

Computer Science S-22

Data Structures

Harvard Summer School, 2026
David G. Sullivan, Ph.D.

Introduction: Course Overview; Object-Oriented Programming.....	2
Recursion; Recursive Backtracking	23
A First Look at Sorting and Algorithm Analysis	51
Divide-and-Conquer Sorting Algorithms; Distributive Sorting	71
Linked Lists	93
Lists, Stacks, and Queues	127
Binary Trees; Huffman Encoding	164
Search Trees	187
Heaps and Priority Queues	206
Hash Tables	224
Graphs.....	240

Introduction: Course Overview, Java Review

Computer Science E-22/S-22
Harvard Extension School

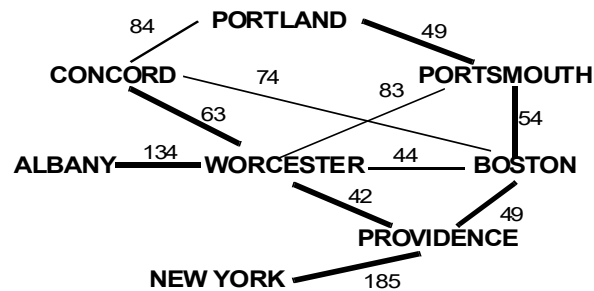
David G. Sullivan, Ph.D.

Welcome to Computer Science E-22/S-22!

- We will study fundamental *data structures*.
 - ways of imposing order on a collection of information
 - sequences: lists, stacks, and queues
 - trees
 - hash tables
 - graphs
- We will also:
 - study *algorithms* related to these data structures
 - learn how to *compare* data structures & algorithms
- Goals:
 - learn to think more intelligently about programming problems
 - acquire a set of useful tools and techniques

Sample Problem I: Finding Shortest Paths

- Given a set of routes between pairs of cities, determine the shortest path from city A to city B.



Sample Problem II: A Data "Dictionary"

- Given a large collection of data, how can we arrange it so that we can efficiently:
 - add a new item
 - search for an existing item
- Some data structures provide better performance than others for this application.
- More generally, we'll learn how to characterize the *efficiency* of different data structures and their associated algorithms.

Prerequisites

- A good working knowledge of Java
 - comfortable with object-oriented programming concepts
 - comfortable with arrays
 - some prior exposure to recursion would be helpful
 - if your skills are weak or rusty, you may want to consider first taking CSCI E-10b
- Reasonable comfort level with mathematical reasoning
 - mostly simple algebra, but need to understand the basics of logarithms (we'll review this)
 - will do some simple proofs

Requirements

- Lectures
- Sections
 - optional but highly recommended
 - start this week
 - on Zoom; one section will be recorded
- Five problem sets
 - plan on 10-20 hours per week!
 - code in Java
 - must be your own work
 - see syllabus for the collaboration policy
 - grad-credit students will do extra problems
- Midterm exam
- Final exam

Exams are *not* online.
They are taken either on campus or
with an in-person proctor (see syllabus).

Additional Administtrivia

- Instructor: Dave Sullivan
- TAs: see website mentioned below
- Office hours and contact info. will be available on the Web:
<https://cscie22.sites.fas.harvard.edu>
- For questions on content, homework, etc.:
 - use Ed Discussion on Canvas
 - send e-mail to cscie22-staff@lists.fas.harvard.edu

Review: What is an Object?

- An *object* groups together:
 - one or more data values (the object's *fields* – also known as *instance variables*)
 - a set of operations that the object can perform (the object's *methods*)
- In Java, we use a *class* to define a new type of object.
 - serves as a "blueprint" for objects of that type
 - simple example:

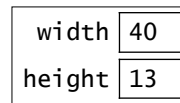
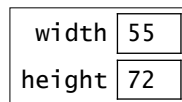
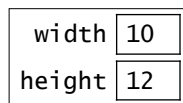
```
public class Rectangle {  
    // fields  
    private int width;  
    private int height;  
  
    // methods  
    public int getWidth() {  
        return this.width;  
    }  
    ...  
}
```

Class vs. Object

- The Rectangle class is a blueprint:

```
public class Rectangle {  
    // fields  
    private int width;  
    private int height;  
    // methods  
    ...  
}
```

- Rectangle objects are built according to that blueprint:



(You can also think of the methods as being inside the object, but we won't show them in our diagrams.)

Creating and Using an Object

- We create an object by using the new operator and a special method known as a *constructor*:

```
Rectangle r1 = new Rectangle(10, 30);
```

- Once an object is created, we can call one of its methods by using *dot notation*:

```
int width1 = r1.getWidth();
```

- The object on which the method is invoked is known as the *called object* or the *current object*.

Two Types of Methods

- Methods that belong to an object are referred to as *instance methods* or *non-static methods*.
 - they are invoked on an object

```
int width1 = r1.getwidth();
```
 - they have access to the fields of the called object
- *Static* methods do *not* belong to an object – they belong to the class as a whole.
 - they have the keyword *static* in their header:

```
public static int max(int num1, int num2) {  
    ...  
}
```
 - they do *not* have access to the fields of the class
 - outside the class, they are invoked using the class name:

```
int result = Math.max(5, 10);
```

Encapsulation

- Our classes should provide proper *encapsulation*.
- We limit direct access to the internals of an object by making the fields *private*:

```
public class Rectangle {  
    private int width;  
    private int height;  
    ...  
}
```

 - private components of a class can only be accessed directly by code within the class itself
- We provide limited *indirect* access through methods that are labeled *public*.

```
public int getwidth() {  
    return this.width;  
}
```

 - public components can be accessed anywhere

Encapsulation (cont.)

- `getWidth()` is an *accessor* method that can be used to obtain information about an object, but *not* to change it:

```
public int getWidth() {  
    return this.width;  
}
```
- we use the keyword `this` to access the fields and methods that are inside the called object
- A class can also provide *mutator* methods that change the called object, but only in appropriate ways:

```
public void setWidth(int newWidth) {  
    if (newWidth <= 0) {  
        throw new IllegalArgumentException();  
    }  
    this.width = newWidth;  
}
```
- throwing an exception prevents an inappropriate change by ending the method prematurely

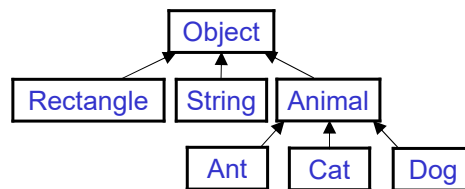
Inheritance

- We can define a class that explicitly *extends* another class:

```
public class Animal {  
    private String name;  
    // other field definitions go here  
  
    public String getName() {  
        return this.name;  
    }  
    // other method definitions go here  
}  
  
public class Dog extends Animal {  
    ...  
}
```
- We say that `Dog` is a *subclass* of `Animal`, and `Animal` is a *superclass* of `Dog`.
- A class *inherits* the fields and methods of the class that it extends.

The object Class

- If a class does not explicitly extend another class, it implicitly extends Java's object class.
- The object class includes methods that all classes must possess. For example:
 - `toString()`: returns a string representation of the object
 - `equals()`: is this object equal to another object?
- The process of extending classes forms a hierarchy of classes, with the object class at the top of the hierarchy:



Polymorphism

- An object can be used wherever an object of one of its superclasses is called for.
- For example:

```
Animal a = new Dog();  
Animal[] zoo = new Animal[100];  
zoo[0] = new Ant();  
zoo[1] = new Cat();  
...
```
- The name for this capability is *polymorphism*.
 - from the Greek for "many forms"
 - the same code can be used with objects of different types

A Bag Data Structure

- A bag is just a container for a group of data items.
 - analogy: a bag of candy
- The positions of the data items don't matter (unlike a sequence).
 - {3, 2, 10, 6} is equivalent to {2, 3, 6, 10}
- The items do *not* need to be unique (unlike a set).
 - {7, 2, 10, 7, 5} isn't a set, but it is a bag

A Bag Data Structure (cont.)

- The operations we want our bag to support:
 - `add(item)`: add `item` to the bag
 - `remove(item)`: remove one occurrence of `item` (if any) from the bag
 - `contains(item)`: check if `item` is in the bag
 - `numItems()`: get the number of items in the bag
 - `grab()`: get an item at random, without removing it
 - reflects the fact that the items don't have a position (and thus we can't say "get the 5th item in the bag")
 - `toArray()`: get an array containing the current contents of the bag

Implementing a Bag Using a Class

- To implement our bag data structure, we define a class:

```
public class ArrayBag {  
    private Object[] items;  
    private int numItems;  
    // constructors go here  
    public boolean add(Object item) {  
        ...  
    }  
}
```

- Each object of this class will represent an entire bag of items.
- The items themselves are stored in an array of type Object.
 - allows us to store *any* type of object in the bag, thanks to the power of polymorphism:

```
ArrayBag bag = new ArrayBag();  
bag.add("hello");  
bag.add(new Rectangle(20, 30));
```

Memory Management: Looking Under the Hood

- To understand how data structures are implemented, you need to understand how memory is managed.
- There are three main types of memory allocation in Java.
- They correspond to three different regions of memory.

Memory Management, Type I: Static Storage

- Static storage is used for *class variables*, which are declared *outside any method* using the keyword `static`:

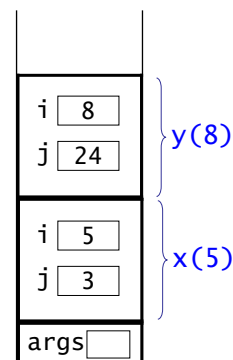
```
public class MyMethods {  
    public static int numCompares;  
    public static final double PI = 3.14159;  
}
```

- There is only one copy of each class variable.
 - shared by all objects of the class
 - Java's version of a global variable
- The Java runtime allocates memory for class variables when the class is first encountered.
 - this memory stays fixed for the duration of the program

Memory Management, Type I: Stack Storage

- Method parameters and other local variables are stored in a region of memory known as *the stack*.
- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {  
    public static int y(int i) {  
        int j = i * 3;  
        return j;  
    }  
    public static int x(int i) {  
        int j = i - 2;  
        return y(i + j);  
    }  
    public static void  
    main(String[] args) {  
        System.out.println(x(5));  
    }  
}
```



- When a method completes, its stack frame is removed.

Memory Management, Type III: Heap Storage

- Objects (including arrays) are stored in a region of memory known as *the heap*.
- Memory on the heap is allocated using the new operator:

```
int[] values = new int[3];
ArrayBag b = new ArrayBag();
```
- new returns the memory address of the start of the object.
- This memory address – which is referred to as a *reference* – is stored in the variable that represents the object:

values

0x23a

0	0	0
---	---	---

- We will often use an arrow to represent a reference:

values

—

 →

0	0	0
---	---	---

Heap Storage (cont.)

- An object remains on the heap until there are no remaining references to it.
- Unused objects are automatically reclaimed by a process known as *garbage collection*.
 - makes their memory available for other objects

Two Constructors for the ArrayBag Class

```
public class ArrayBag {
    private Object[] items;
    private int numItems;
    public static final int DEFAULT_MAX_SIZE = 50;

    public ArrayBag() {
        this.items = new Object[DEFAULT_MAX_SIZE];
        this.numItems = 0;
    }
    public ArrayBag(int maxSize) {
        ...
    }
}
```

- A class can have multiple constructors.
 - the parameters must differ in some way
- The first one is useful for small bags.
 - creates an array with room for 50 items.
- The second one allows the client to specify the max # of items.

Two Constructors for the ArrayBag Class

```
public class ArrayBag {
    private Object[] items;
    private int numItems;
    public static final int DEFAULT_MAX_SIZE = 50;

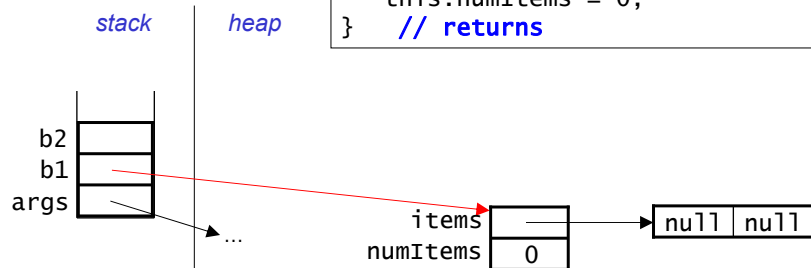
    public ArrayBag() {
        this.items = new Object[DEFAULT_MAX_SIZE];
        this.numItems = 0;
    }
    public ArrayBag(int maxSize) {
        if (maxSize <= 0) {
            throw new IllegalArgumentException(
                "maxSize must be > 0");
        }
        this.items = new Object[maxSize];
        this.numItems = 0;
    }
    ...
}
```

- If the user inputs an invalid maxSize, we throw an exception.

Example: Creating Two ArrayBag Objects

```
// client
public static void main(String[] args) {
    ArrayBag b1 = new ArrayBag(2);
    ArrayBag b2 = new ArrayBag(4);
    ...
}
```

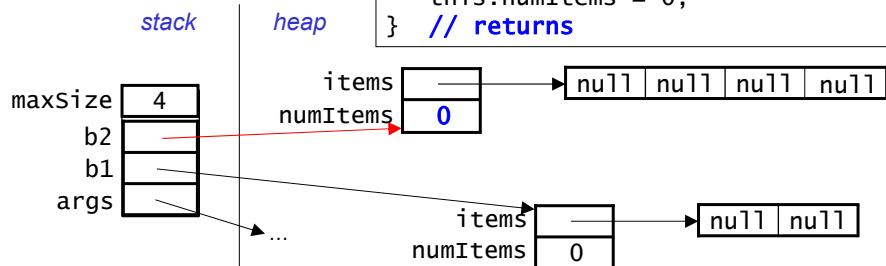
```
// constructor
public ArrayBag(int maxSize) {
    ... // error-checking
    this.items = new Object[maxSize];
    this.numItems = 0;
} // returns
```



Example: Creating Two ArrayBag Objects

```
// client
public static void main(String[] args) {
    ArrayBag b1 = new ArrayBag(2);
    ArrayBag b2 = new ArrayBag(4);
    ...
}
```

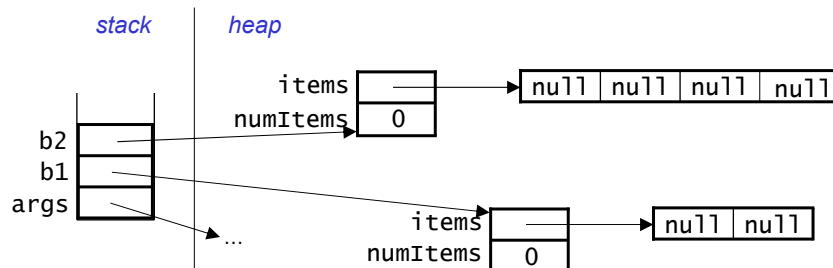
```
// constructor
public ArrayBag(int maxSize) {
    ... // error-checking
    this.items = new Object[maxSize];
    this.numItems = 0;
} // returns
```



Example: Creating Two ArrayBag Objects

```
// client
public static void main(String[] args) {
    ArrayBag b1 = new ArrayBag(2);
    ArrayBag b2 = new ArrayBag(4);
    ...
}
```

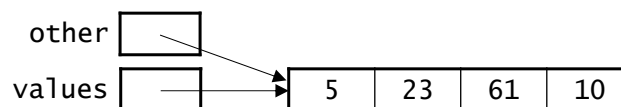
- After the objects have been created, here's what we have:



Copying References

- A variable that represents an array or object is known as a *reference variable*.
- Assigning the value of one reference variable to another reference variable copies the reference to the array or object. It does *not* copy the array or object itself.

```
int[] values = {5, 23, 61, 10};
int[] other = values;
```



- Given the lines above, what will the lines below output?
`other[2] = 17;`
`System.out.println(values[2] + " " + other[2]);`

Passing an Object/Array to a Method

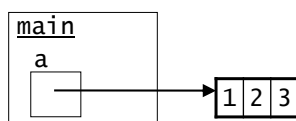
- When a method is passed an object or array as a parameter, the method gets a copy of the *reference* to the object or array, *not* a copy of the object or array itself.
- Thus, any changes that the method makes to the *internals* of the object/array will still be there when the method returns.
- Consider the following:

```
public static void main(String[] args) {  
    int[] a = {1, 2, 3};  
    triple(a);  
    System.out.println(Arrays.toString(a));  
}  
  
public static void triple(int[] n) {  
    for (int i = 0; i < n.length; i++) {  
        n[i] = n[i] * 3;  
    }  
}
```

What is the output?

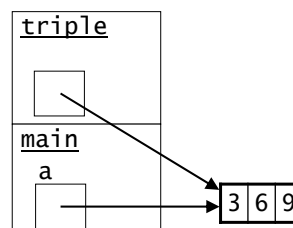
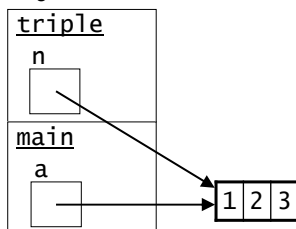
Passing an Object/Array to a Method (cont.)

before method call

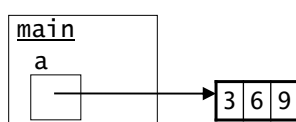


```
public static void main(String[] args) {  
    int[] a = {1, 2, 3};  
    triple(a);  
    System.out.println(...);  
}
```

during method call

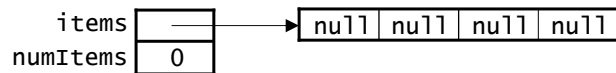


after method call

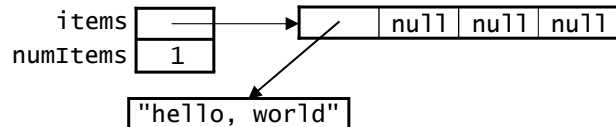


Adding Items to an ArrayBag

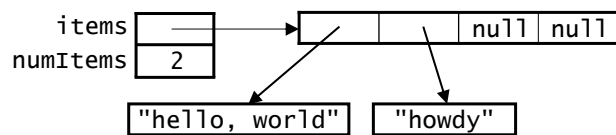
- We fill the array from left to right. Here's an empty bag:



- After adding the first item:

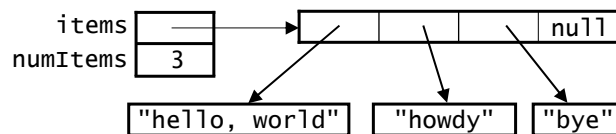


- After adding the second item:

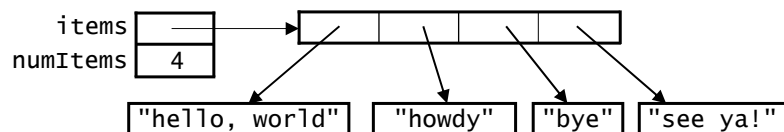


Adding Items (cont.)

- After adding the third item:



- After adding the fourth item:



- At this point, the ArrayBag is full!
 - it's non-trivial to "grow" an array, so we don't!
 - additional items cannot be added until one is removed

A Method for Adding an Item to a Bag

```
public class ArrayBag {
    private Object[] items;
    private int numItems;
    ...
    public boolean add(Object item) {
        if (item == null) {
            throw new IllegalArgumentException("no nulls");
        } else if (this.numItems == this.items.length) {
            return false; // no more room!
        } else {
            this.items[this.numItems] = item;
            this.numItems++;
            return true; // success!
        }
    }
    ...
}
```

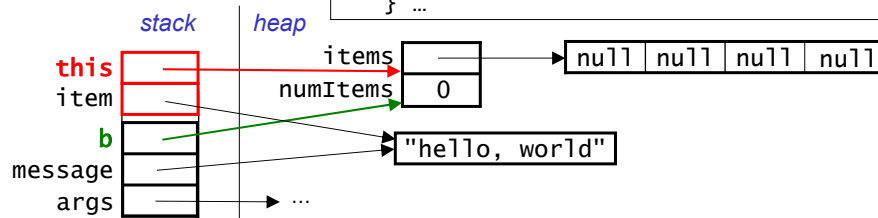
- takes an object of any type!
- returns a boolean to indicate if the operation succeeded

- Initially, `this.numItems` is 0, so the first item goes in position 0.
- We increase `this.numItems` because we now have 1 more item.
 - and so the *next* item added will go in the correct position!

Example: Adding an Item (cont.)

```
public static void main(String[] args) {
    String message = "hello, world";
    ArrayBag b = new ArrayBag(4);
    b.add(message);
    ...
}

public boolean add(Object item) {
    ...
    else {
        this.items[this.numItems] = item;
        this.numItems++;
        return true;
    }
    ...
}
```

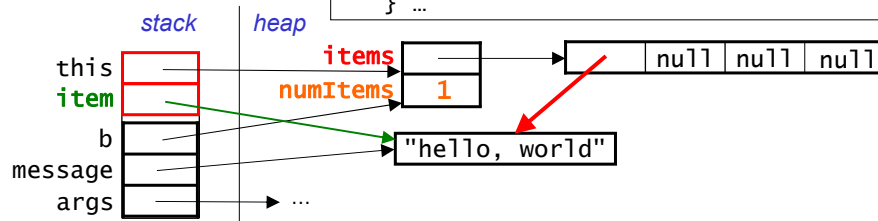


- `add`'s stack frame includes:
 - `item`, which stores a copy of the reference passed as a param.
 - `this`, which stores a reference to the called `ArrayBag` object

Example: Adding an Item

```
public static void main(String[] args) {
    String message = "hello, world";
    ArrayBag b = new ArrayBag(4);
    b.add(message);
    ...
}
```

```
public boolean add(Object item) {
    ...
    else {
        this.items[this.numItems] = item;
        this.numItems++;
        return true;
    } ...
}
```

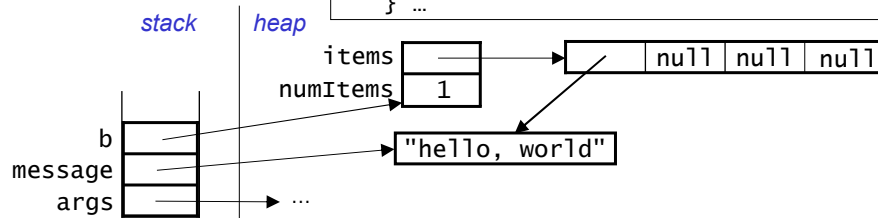


- The method modifies the `items` array and `numItems`.
 - note that the array holds a copy of the *reference* to the item, not a copy of the item itself.

Example: Adding an Item

```
public static void main(String[] args) {
    String message = "hello, world";
    ArrayBag b = new ArrayBag(4);
    b.add(message);
    ...
}
```

```
public boolean add(Object item) {
    ...
    else {
        this.items[this.numItems] = item;
        this.numItems++;
        return true;
    } ...
}
```



- After the method call returns, `add`'s stack frame is removed from the stack.

A Type Mismatch

- Here are the headers of two ArrayBag methods:

```
public boolean add(Object item)
public Object grab()
```

- Polymorphism allows us to pass String objects into add():

```
ArrayBag stringBag = new ArrayBag();
stringBag.add("hello");
stringBag.add("world");
```

- However, this will not work:

```
String str = stringBag.grab(); // compiler error
```

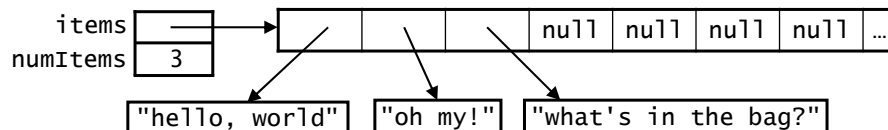
- the return type of grab() is Object
- Object isn't a subclass of String, so polymorphism doesn't help!

- Instead, we need to use a *type cast*:

```
String str = (String)stringBag.grab();
```

- this cast doesn't actually change the value being assigned
- it just reassures the compiler that the assignment is okay

Extra Practice: Determining if a Bag Contains an Item

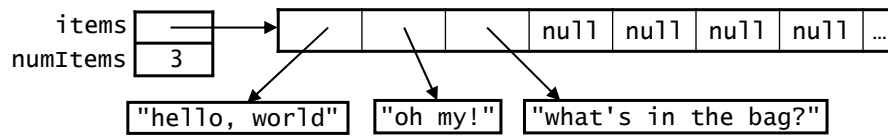


- Let's write the ArrayBag contains() method together.
- should return true if an object equal to item is found, and false otherwise.

```
_____ contains(_____ item) {
```

```
}
```

Would this work instead?



- Let's write the `ArrayBag` `contains()` method together.
 - should return `true` if an object equal to `item` is found, and `false` otherwise.

```
public boolean contains(Object item) {  
    for (int i = 0; i < this.items.length; i++) {  
        if (this.items[i].equals(item)) { // not ==  
            return true;  
        }  
    }  
    return false;  
}
```

Another Incorrect `contains()` Method

```
public boolean contains(Object item) {  
    for (int i = 0; i < this.numItems; i++) {  
        if (this.items[i].equals(item)) {  
            return true;  
        } else {  
            return false;  
        }  
    }  
    return false;  
}
```

- What's the problem with this?

Recursion and Recursive Backtracking

Computer Science E-22
Harvard Extension School

David G. Sullivan, Ph.D.

Iteration

- When we encounter a problem that requires repetition, we often use *iteration* – i.e., some type of loop.
- Sample problem: printing the series of integers from n_1 to n_2 , where $n_1 \leq n_2$.
 - example: `printSeries(5, 10)` should print the following:
5, 6, 7, 8, 9, 10

- Here's an iterative solution to this problem:

```
public static void printSeries(int n1, int n2) {  
    for (int i = n1; i < n2; i++) {  
        System.out.print(i + ", ");  
    }  
    System.out.println(n2);  
}
```

Recursion

- An alternative approach to problems that require repetition is to solve them using *recursion*.
- A recursive method is a method that calls itself.
- Applying this approach to the print-series problem gives:

```
public static void printSeries(int n1, int n2) {  
    if (n1 == n2) {  
        System.out.println(n2);  
    } else {  
        System.out.print(n1 + ", ");  
        printSeries(n1 + 1, n2);  
    }  
}
```

Tracing a Recursive Method

```
public static void printSeries(int n1, int n2) {  
    if (n1 == n2) {  
        System.out.println(n2);  
    } else {  
        System.out.print(n1 + ", ");  
        printSeries(n1 + 1, n2);  
    }  
}
```

- What happens when we execute `printSeries(5, 7)`?

```
printSeries(5, 7):  
    System.out.print(5 + ", ");  
printSeries(6, 7):  
    System.out.print(6 + ", ");  
printSeries(7, 7):  
    System.out.print(7);  
    return  
    return  
return
```

Recursive Problem-Solving

- When we use recursion, we solve a problem by reducing it to a simpler problem of the same kind.
- We keep doing this until we reach a problem that is simple enough to be solved directly.
- This simplest problem is known as the *base case*.

```
public static void printSeries(int n1, int n2) {  
    if (n1 == n2) {                // base case  
        System.out.println(n2);  
    } else {  
        System.out.print(n1 + ", ");  
        printSeries(n1 + 1, n2);  
    }  
}
```

- The base case stops the recursion, because it doesn't make another call to the method.

Recursive Problem-Solving (cont.)

- If the base case hasn't been reached, we execute the *recursive case*.

```
public static void printSeries(int n1, int n2) {  
    if (n1 == n2) {                // base case  
        System.out.println(n2);  
    } else {                        // recursive case  
        System.out.print(n1 + ", ");  
        printSeries(n1 + 1, n2);  
    }  
}
```

- The recursive case:
 - reduces the overall problem to one or more simpler problems of the same kind
 - makes recursive calls to solve the simpler problems

Structure of a Recursive Method

```
recursiveMethod(parameters) {  
    if (stopping condition) {  
        // handle the base case  
    } else {  
        // recursive case:  
        // possibly do something here  
        recursiveMethod(modified parameters);  
        // possibly do something here  
    }  
}
```

- There can be multiple base cases and recursive cases.
- When we make the recursive call, we typically use parameters that bring us closer to a base case.

Tracing a Recursive Method: Second Example

```
public static void mystery(int i) {  
    if (i <= 0) {        // base case  
        return;  
    }  
    // recursive case  
    System.out.println(i);  
    mystery(i - 1);  
    System.out.println(i);  
}
```

- What happens when we execute `mystery(2)`?

A Recursive Method That Returns a Value

- Simple example: summing the integers from 1 to n

```
public static int sum(int n) {  
    if (n <= 0) {  
        return 0;  
    }  
    int rest = sum(n - 1);  
    return n + rest;  
}
```

- Example of this approach to computing the sum:

```
sum(6) = 6 + sum(5)  
       = 6 + 5 + sum(4)  
       ...
```

Tracing a Recursive Method

```
public static int sum(int n) {  
    if (n <= 0) {  
        return 0;  
    }  
    int rest = sum(n - 1);  
    return n + rest;  
}
```

- What happens when we execute `int x = sum(3);` from inside the `main()` method?

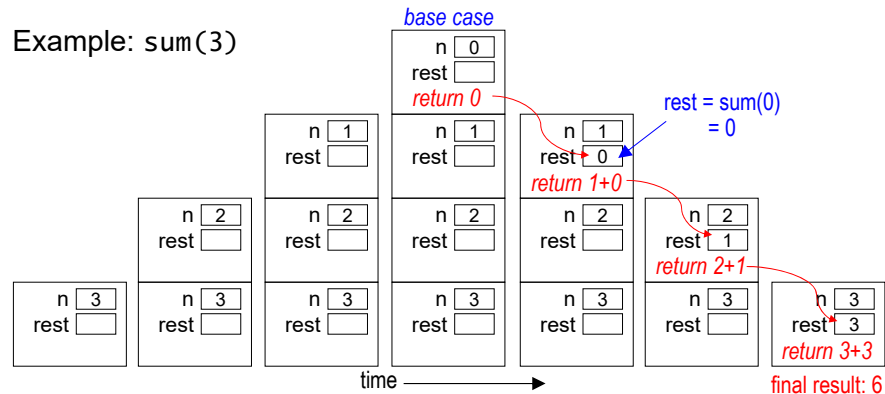
```
main() calls sum(3)  
  sum(3) calls sum(2)  
    sum(2) calls sum(1)  
      sum(1) calls sum(0)  
        sum(0) returns 0  
      sum(1) returns 1 + 0 or 1  
    sum(2) returns 2 + 1 or 3  
  sum(3) returns 3 + 3 or 6  
main() assigns 6 to x
```

Tracing a Recursive Method on the Stack

```
public static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    int rest = sum(n - 1);
    return n + rest;
}
```

The final result gets built up **on the way back** from the base case!

Example: sum(3)



Infinite Recursion

- We have to ensure that a recursive method will eventually reach a base case, regardless of the initial input.
- Otherwise, we can get *infinite recursion*.
 - produces *stack overflow* - there's no room for more frames on the stack!
- Example: here's a version of our `sum()` method that uses a different test for the base case:

```
public static int sum(int n) {
    if (n == 0) {
        return 0;
    }
    int rest = sum(n - 1);
    return n + rest;
}
```

- what values of `n` would cause infinite recursion?

Designing a Recursive Method

1. Start by programming the base case(s).
 - *What instance(s) of this problem can I solve directly (without looking at anything smaller)?*
2. Find the recursive substructure.
 - *How could I use the solution to **any smaller version** of the problem to solve the overall problem?*
3. Solve the smaller problem using a recursive call!
 - **store its result in a variable**
4. Do your one step.
 - build your solution from the result of the recursive call
 - **use concrete cases to figure out what you need to do**

Processing a String Recursively

- A string is a recursive data structure. It is either:
 - empty ("")
 - a single character, followed by a string
- Thus, we can easily use recursion to process a string.
 - process one or two of the characters ourselves
 - make a recursive call to process the rest of the string
- Example: print a string vertically, one character per line:

```
public static void printVertical(String str) {  
    if (str == null || str.equals("")) {  
        return;  
    }  
  
    System.out.println(str.charAt(0)); // first char  
    printVertical(str.substring(1));   // rest of string  
}
```

Short-Circuited Evaluation

- The second operand of both the `&&` and `||` operators will not be evaluated if the result can be determined on the basis of the first operand alone.
- `expr1 || expr2`
if `expr1` evaluates to `true`, `expr2` is not evaluated, because we already know that `expr1 || expr2` is `true`
 - example from the last slide:

```
if (str == null || str.equals("")) {  
    return;  
}  
// if str is null, we won't check for empty string.  
// This prevents a null pointer exception!
```
- `expr1 && expr2`
if `expr1` evaluates to , `expr2` is not evaluated, because we already know that `expr1 && expr2` is .

Counting Occurrences of a Character in a String

- `numOccur(c, s)` should return the number of times that the character `c` appears in the string `s`
 - `numOccur('n', "banana")` should return 2
 - `numOccur('a', "banana")` should return 3
- Take the approach outlined earlier:
 - base case: empty string (or null)
 - delegate `s.substring(1)` to the recursive call
 - we're responsible for handling `s.charAt(0)`

Recursive Counting

```
public static int numOccur(char c, String s) {  
    if (s == null || s.equals("")) {  
        return 0;  
    } else {  
        int rest = numOccur(c, s.substring(1));  
        // do our one step!  
        ...  
    }  
}
```

Determining Our One Step

```
public static int numOccur(char c, String s) {  
    if (s == null || s.equals("")) {  
        return 0;  
    } else {  
        int rest = numOccur(c, s.substring(1));  
        // do our one step!  
    }  
}
```

- In our one step, we take care of `s.charAt(0)`.
 - we build the solution to the larger problem on the solution to the smaller problem (in this case, `rest`)
 - does what we do depend on the value of `s.charAt(0)`?
- ***Use concrete cases to figure out the logic!***

Consider this concrete case...

```
public static int numOccur(char c, String s) {  
    if (s == null || s.equals("")) {  
        return 0;  
    } else {  
        int rest = numOccur(c, s.substring(1));  
        // do our one step!  
        ...  
    }  
}
```

`numOccur('r', "recurse")`

What value is eventually assigned to rest?
(i.e., what does the recursive call return?)

```
public static int numOccur(char c, String s) {  
    if (s == null || s.equals("")) {  
        return 0;  
    } else {  
        int rest = numOccur(c, s.substring(1));  
        // do our one step!  
        ...  
    }  
}
```

`numOccur('r', "recurse")`

```
numOccur('r', "recurse")  
c = 'r', s = "recurse"  
int rest = ???
```

Consider Concrete Cases

`numOccur('r', "recurse")` # first char is a match

- what is its solution?
- what is the next smaller subproblem?
- what is the solution to that subproblem?
- how can we use the solution to the subproblem?
What is our one step?

`numOccur('a', "banana")` # first char is not a match

- what is its solution?
- what is the next smaller subproblem?
- what is the solution to that subproblem?
- how can we use the solution to the subproblem?
What is our one step?

Now complete the method!

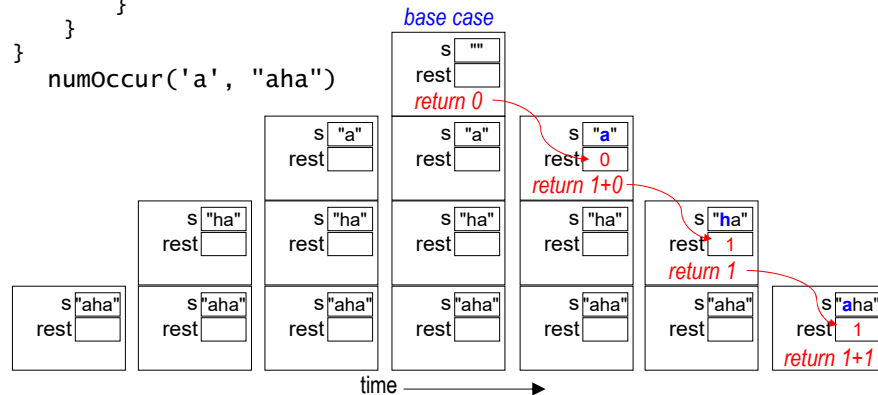
```
public static int numOccur(char c, String s) {  
    if (s == null || s.equals("")) {  
        return 0;  
    } else {  
        int rest = numOccur(c, s.substring(1));  
        if (s.charAt(0) == c) {  
            return _____;  
        } else {  
            return _____;  
        }  
    }  
}
```

Tracing a Recursive Method on the Stack

```
public static int numOccur(char c, String s) {
    if (s == null || s.equals("")) {
        return 0;
    } else {
        int rest = numOccur(c, s.substring(1));
        if (s.charAt(0) == c) {
            return 1 + rest;
        } else {
            return rest;
        }
    }
}
```

numOccur('a', "aha")

The final result gets built up **on the way back** from the base case!



Common Mistake

- This version of the method does *not* work:

```
public static int numOccur(char c, String s) {
    if (s == null || s.equals("")) {
        return 0;
    }

    int count = 0;
    if (s.charAt(0) == c) {
        count++;
    }

    numOccur(c, s.substring(1));
    return count;
}
```

Another Faulty Approach

- Some people make count "global" to fix the prior version:

```
public static int count = 0;

public static int numOccur(char c, String s) {
    if (s == null || s.equals("")) {
        return 0;
    }
    if (s.charAt(0) == c) {
        count++;
    }
    numOccur(c, s.substring(1));
    return count;
}
```

- Not recommended, and not allowed on the problem sets!
- Problems with this approach?

Removing Vowels From a String

- `removeVowels(s)` - removes the vowels from the string `s`, returning its "vowel-less" version!
`removeVowels("recursive")` should return `"rcrsv"`
`removeVowels("vowel")` should return `"vwl"`
- Can we take the usual approach to recursive string processing?
 - base case: empty string
 - delegate `s.substring(1)` to the recursive call
 - we're responsible for handling `s.charAt(0)`

Applying the String-Processing Template

```
public static String removeVowels(String s) {  
    if (s.equals("")) {    // base case  
        return _____;  
    } else {                // recursive case  
        String rem_rest = _____;  
        // do our one step!  
    }  
}
```

Consider Concrete Cases

`removeVowels("after")` # first char is a vowel

- what is its solution?
- what is the next smaller subproblem?
- what is the solution to that subproblem?
- how can we use the solution to the subproblem?
What is our one step?

`removeVowels("recurse")` # first char is not a vowel

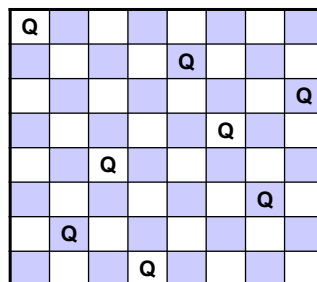
- what is its solution?
- what is the next smaller subproblem?
- what is the solution to that subproblem?
- how can we use the solution to the subproblem?
What is our one step?

removeVowels()

```
public static String removeVowels(String s) {  
    if (s.equals("")) { // base case  
        return "";  
    } else { // recursive case  
        String rem_rest = removeVowels(s.substring(1));  
        if ("aeiou".indexOf(s.charAt(0)) != -1) {  
            _____  
        } else {  
            _____  
        }  
    }  
}
```

The n-Queens Problem

- **Goal:** to place n queens on an $n \times n$ chessboard so that no two queens occupy:
 - the same row
 - the same column
 - the same diagonal.
- Sample solution for $n = 8$:



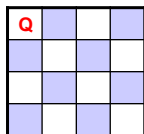
- This problem can be solved using a technique called *recursive backtracking*.

Recursive Strategy for n-Queens

- `findSolution(row)` – to place a queen in the specified row:
 - try one column at a time, looking for a "safe" one
 - if we find one: – place the queen there
 - *make a recursive call* to go to the next row
 - if we can't find one: – *backtrack* by returning from the call
 - try to find another safe column in the previous row

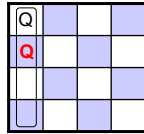
- Example:

- row 0:

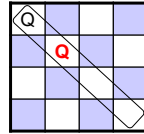


col 0: safe

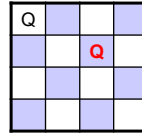
- row 1:



col 0: same col



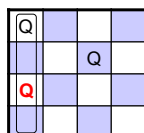
col 1: same diag



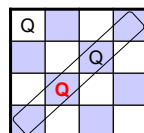
col 2: safe

4-Queens Example (cont.)

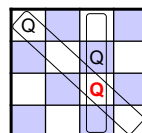
- row 2:



col 0: same col



col 1: same diag

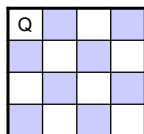


col 2: same col/diag

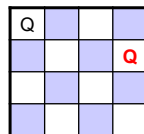


col 3: same diag

- We've run out of columns in row 2!
- *Backtrack* to row 1 by returning from the recursive call.
 - pick up where we left off
 - we had already tried columns 0-2, so now we try column 3:



we left off in col 2

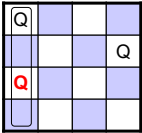


try col 3: safe

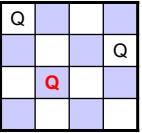
- Continue the recursion as before.

4-Queens Example (cont.)

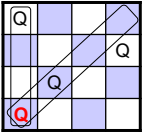
- row 2:



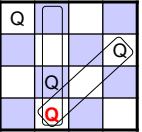
col 0: same col



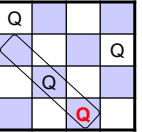
col 1: safe
- row 3:



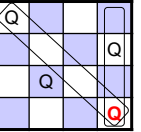
col 0: same col/diag



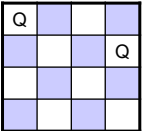
col 1: same col/diag



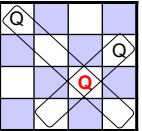
col 2: same diag



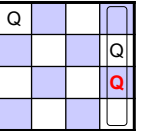
col 3: same col/diag
- Backtrack to row 2:



we left off in col 1



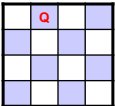
col 2: same diag

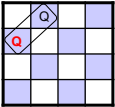


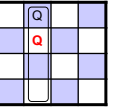
col 3: same col
- Backtrack to row 1. No columns left, so backtrack to row 0!

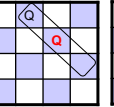
4-Queens Example (cont.)

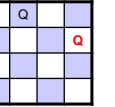
- row 0:

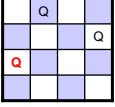

- row 1:

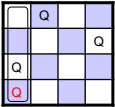


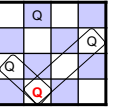


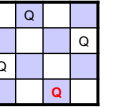



- row 2:


- row 3:





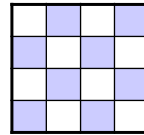


A solution!

A Blueprint Class for an N-Queens Solver

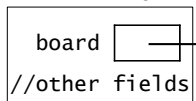
```
public class NQueens {
    private boolean[][] board; // state of the chessboard
    // other fields go here...

    public NQueens(int n) {
        this.board = new boolean[n][n];
        // initialize other fields here...
    }
    ...
}
```



- Here's what the object looks like initially:

NQueens object



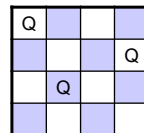
false	false	false	false
false	false	false	false
false	false	false	false
false	false	false	false

A Blueprint Class for an N-Queens Solver

```
public class NQueens {
    private boolean[][] board; // state of the chessboard
    // other fields go here...

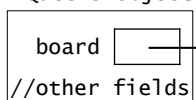
    public NQueens(int n) {
        this.board = new boolean[n][n];
        // initialize other fields here...
    }

    private void placeQueen(int row, int col) {
        this.board[row][col] = true;
        // modify other fields here...
    }
}
```



- Here's what it looks like after placing some queens:

NQueens object



true	false	false	false
false	false	false	true
false	true	false	false
false	false	false	false

A Blueprint Class for an N-Queens Solver

```
public class NQueens {
    private boolean[][] board; // state of the chessboard
    // other fields go here...

    public NQueens(int n) {
        this.board = new boolean[n][n];
        // initialize other fields here...
    }

    private void placeQueen(int row, int col) {
        this.board[row][col] = true;
        // modify other fields here...
    }

    private void removeQueen(int row, int col) {
        this.board[row][col] = false;
        // modify other fields here...
    }

    private boolean isSafe(int row, int col) {
        // returns true if [row][col] is "safe", else false
    }

    private boolean findSolution(int row) {
        // see next slide!
    }
    ...
}
```

private helper methods
that will only be called
by code within the class.

Making them private
means we don't need
to do error-checking!

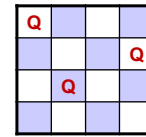
Recursive-Backtracking Method

```
private boolean findSolution(int row) {
    if (row == this.board.length) {
        this.displayBoard();
        return true;
    }
    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);
            if (this.findSolution(row + 1)) {
                return true;
            }
            this.removeQueen(row, col);
        }
    }
    return false;
}
```

- takes the index of a row (initially 0)
- uses a loop to consider all possible columns in that row
- makes a recursive call to move onto the next row
- returns true if a solution has been found; false otherwise

Tracing findSolution()

```
private boolean findSolution(int row) {
    if (row == this.board.length) {
        // code to process a solution goes here...
    }
    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);
            if (this.findSolution(row + 1)) {
                return true;
            }
            this.removeQueen(row, col);
        }
    }
    return false;
}
```



Note: row++
will not work
here!

We can pick up
where we left off,
because row and
col are stored in
the stack frame!

backtrack!

row: 3
col: 0,1,2,3,4
return false

row: 2
col: 0,1,2,3,4
return false

row: 2
col: 0,1

row: 2
col: 0,1

row: 1
col: 0,1,2

row: 1
col: 0,1,2

row: 1
col: 0,1,2

row: 1
col: 0,1,2,3

row: 1
col: 0,1,2,3

row: 1
col: 0,1,2,3

row: 0
col: 0

row: 0
col: 0

row: 0
col: 0

row: 0
col: 0

row: 0
col: 0

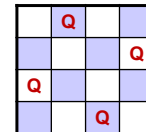
row: 0
col: 0

row: 0
col: 0

time →

Once we place a queen in the last row...

```
private boolean findSolution(int row) {
    if (row == this.board.length) {
        this.displayBoard();
        return true;
    }
    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);
            if (this.findSolution(row + 1)) {
                return true;
            }
            this.removeQueen(row, col);
        }
    }
    return false;
}
```



row: 3
col: 0,1,2

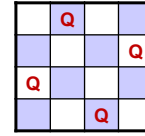
row: 2
col: 0

row: 1
col: 0,1,2,3

row: 0
col: 1

time →

...we make one more recursive call...

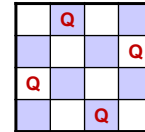


```
private boolean findSolution(int row) {
    if (row == this.board.length) {
        this.displayBoard();
        return true;
    }
    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);
            if (this.findSolution(row + 1)) {
                return true;
            }
            this.removeQueen(row: 4);
        }
    }
    return false;
}
```

row: 3 col: 0,1,2	row: 3 col: 0,1,2
row: 2 col: 0	row: 2 col: 0
row: 1 col: 0,1,2,3	row: 1 col: 0,1,2,3
row: 0 col: 1	row: 0 col: 1

time →

...and hit the base case!

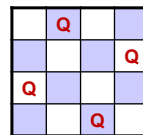


```
private boolean findSolution(int row) {
    if (row == this.board.length) {
        this.displayBoard();
        return true;
    }
    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);
            if (this.findSolution(row + 1)) {
                return true;
            }
            this.removeQueen(row: 4, return true);
        }
    }
    return false;
}
```

row: 3 col: 0,1,2	row: 3 col: 0,1,2
row: 2 col: 0	row: 2 col: 0
row: 1 col: 0,1,2,3	row: 1 col: 0,1,2,3
row: 0 col: 1	row: 0 col: 1

time →

true is sent back...

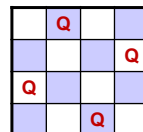


```
private boolean findSolution(int row) {
    if (row == this.board.length) {
        this.displayBoard();
        return true;
    }
    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);
            if (this.findSolution(row + 1)) { // if (true)
                return true;
            }
            this.removeQueen(row: 4, col: 1);
        }
    }
    return false;
}
```

row: 3 col: 0,1,2	row: 3 col: 0,1,2	row: 3 col: 0,1,2
row: 2 col: 0	row: 2 col: 0	row: 2 col: 0
row: 1 col: 0,1,2,3	row: 1 col: 0,1,2,3	row: 1 col: 0,1,2,3
row: 0 col: 1	row: 0 col: 1	row: 0 col: 1

time →

...and all the earlier calls also return true!



```
private boolean findSolution(int row) {
    if (row == this.board.length) {
        this.displayBoard();
        return true;
    }
    for (int col = 0; col < this.board.length; col++) {
        if (this.isSafe(row, col)) {
            this.placeQueen(row, col);
            if (this.findSolution(row + 1)) { // if (true)
                return true;
            }
            this.removeQueen(row: 4, col: 1);
        }
    }
    return false;
}
```

row: 3 col: 0,1,2	row: 3 col: 0,1,2	row: 3 col: 0,1,2 return true	row: 2 col: 0 return true
row: 2 col: 0	row: 2 col: 0	row: 2 col: 0	row: 1 col: 0,1,2,3
row: 1 col: 0,1,2,3	row: 1 col: 0,1,2,3	row: 1 col: 0,1,2,3	row: 0 col: 1
row: 0 col: 1	row: 0 col: 1	row: 0 col: 1	row: 0 col: 1

time →

Using a "Wrapper" Method

- The key recursive method is private:

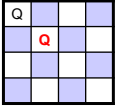
```
private boolean findSolution(int row) {  
    ...  
}
```

- We use a separate, public "wrapper" method to start the recursion:

```
public boolean findSolution() {  
    return this.findSolution(0);  
}
```

- an example of overloading – two methods with the same name, but different parameters
- this method takes no parameters
- it makes the initial call to the recursive method and returns whatever that call returns
- it allows us to ensure that the correct initial value is passed into the recursive method

Recursive Backtracking in General

- Useful for *constraint satisfaction problems*
 - involve assigning values to variables according to a set of constraints
 - n-Queens: variables = Queen's position in each row
constraints = no two queens in same row/col/diag
 - many others: factory scheduling, room scheduling, etc.
- Backtracking greatly reduces the number of possible solutions that we consider.
 - ex: 
 - there are 16 possible solutions that begin with queens in these two positions
 - backtracking doesn't consider any of them!
- Recursion makes it easy to handle an arbitrary problem size.
 - stores the state of each variable in a separate stack frame

Template for Recursive Backtracking

```
// n is the number of the variable that the current  
// call of the method is responsible for  
boolean findSolution(int n, possibly other params) {  
    if (found a solution) {  
        this.displaySolution();  
        return true;  
    }  
  
    // loop over possible values for the nth variable  
    for (val = first to last) {  
        if (this.isValid(val, n)) {  
            this.applyValue(val, n);  
            if (this.findSolution(n+1, other params)) {  
                return true;  
            }  
            this.removeValue(val, n);  
        }  
    }  
    return false;    // backtrack!  
}
```

Note: n++ will not work here!

Template for Finding Multiple Solutions

(up to some target number of solutions)

```
boolean findSolutions(int n, possibly other params) {  
    if (found a solution) {  
        this.displaySolution();  
        this.solutionsFound++;  
        return (this.solutionsFound >= this.target);  
    }  
  
    // loop over possible values for the nth variable  
    for (val = first to last) {  
        if (isValid(val, n)) {  
            this.applyValue(val, n);  
            if (this.findSolutions(n+1, other params)) {  
                return true;  
            }  
            this.removeValue(val, n);  
        }  
    }  
    return false;  
}
```

Data Structures for n-Queens

- Three key operations:
 - `isSafe(row, col)`: check to see if a position is safe
 - `placeQueen(row, col)`
 - `removeQueen(row, col)`
- In theory, our 2-D array of booleans would be sufficient:


```
public class NQueens {
    private boolean[][] board;
```
- It's easy to place or remove a queen:


```
private void placeQueen(int row, int col) {
    this.board[row][col] = true;
}
private void removeQueen(int row, int col) {
    this.board[row][col] = false;
}
...
```
- Problem: `isSafe()` takes a lot of steps. What matters more?

Additional Data Structures for n-Queens

- To facilitate `isSafe()`, add three arrays of booleans:


```
private boolean[] colEmpty;
private boolean[] upDiagEmpty;
private boolean[] downDiagEmpty;
```
- An entry in one of these arrays is:
 - `true` if there are no queens in the column or diagonal
 - `false` otherwise
- Numbering diagonals to get the indices into the arrays:

$$\text{upDiag} = \text{row} + \text{col}$$

$$\text{downDiag} = (\text{boardSize} - 1) + \text{row} - \text{col}$$

	0	1	2	3
0	0	1	2	3
1	1	2	3	4
2	2	3	4	5
3	3	4	5	6

	0	1	2	3
0	3	2	1	0
1	4	3	2	1
2	5	4	3	2
3	6	5	4	3

Using the Additional Arrays

- Placing and removing a queen now involve updating four arrays instead of just one. For example:

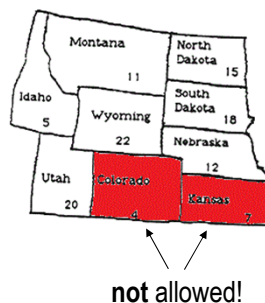
```
private void placeQueen(int row, int col) {  
    this.board[row][col] = true;  
    this.colEmpty[col] = false;  
    this.upDiagEmpty[row + col] = false;  
    this.downDiagEmpty[  
        (this.board.length - 1) + row - col] = false;  
}
```

- However, checking if a square is safe is now more efficient:

```
private boolean isSafe(int row, int col) {  
    return (this.colEmpty[col]  
        && this.upDiagEmpty[row + col]  
        && this.downDiagEmpty[  
            (this.board.length - 1) + row - col]);  
}
```

Recursive Backtracking II: Map Coloring

- We want to color a map using **only four colors**.
- Bordering states or countries **cannot** have the same color.
 - example:



Applying the Template to Map Coloring

```

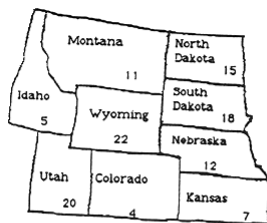
boolean findSolution(n, perhaps other params) {
    if (found a solution) {
        this.displaySolution();
        return true;
    }
    for (val = first to last) {
        if (this.isValid(val, n)) {
            this.applyValue(val, n);
            if (this.findSolution(n + 1, other params)) {
                return true;
            }
            this.removeValue(val, n);
        }
    }
    return false;
}

```

template element	meaning in map coloring
n	
found a solution	
val	
isValid(val, n)	
applyValue(val, n)	
removeValue(val, n)	

Map Coloring Example

consider the states in alphabetical order. colors = { red, yellow, green, blue }.



We color Colorado through Utah without a problem.

Colorado:

Idaho:

Kansas:

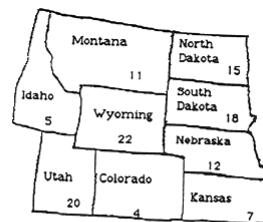
Montana:

Nebraska:

North Dakota:

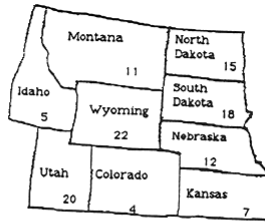
South Dakota:

Utah:



No color works for Wyoming, so we backtrack...

Map Coloring Example (cont.)



Now we can complete the coloring:

Recursion vs. Iteration

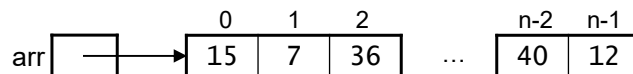
- Some problems are much easier to solve using recursion.
- Other problems are just as easy to solve using iteration.
- Recursion is a bit more costly because of the overhead involved in invoking a method.
 - also: in some cases, there may not be room on the stack
- Rule of thumb:
 - if it's easier to formulate a solution recursively, use recursion, unless the cost of doing so is too high
 - otherwise, use iteration

A First Look at Sorting and Algorithm Analysis

Computer Science E-22
Harvard Extension School

David G. Sullivan, Ph.D.

Sorting an Array of Integers



- Ground rules:
 - sort the values in increasing order
 - sort “in place,” using only a small amount of additional storage
- Terminology:
 - position: one of the memory locations in the array
 - element: one of the data items stored in the array
 - element i : the element at position i
- Goal: minimize the number of **comparisons** C and the number of **moves** M needed to sort the array.
 - move = copying an element from one position to another
example: `arr[3] = arr[5];`

Defining a Class for our Sort Methods

```
public class Sort {  
    public static void bubbleSort(int[] arr) {  
        ...  
    }  
    public static void insertionSort(int[] arr) {  
        ...  
    }  
    ...  
}
```

- Our sort class is simply a collection of methods like Java's built-in Math class.
- Because we never create Sort objects, all of the methods in the class must be *static*.
 - outside the class, we invoke them using the class name:
e.g., `Sort.bubbleSort(arr)`

Defining a Swap Method

- It would be helpful to have a method that swaps two elements of the array.
- Why won't the following work?

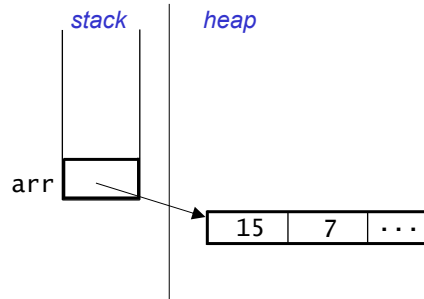
```
private static void swap(int a, int b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

An Incorrect Swap Method

```
private static void swap(int a, int b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

- Trace through the following lines to see the problem:

```
int[] arr = {15, 7, ...};  
swap(arr[0], arr[1]);
```



A Correct Swap Method

- This method works:

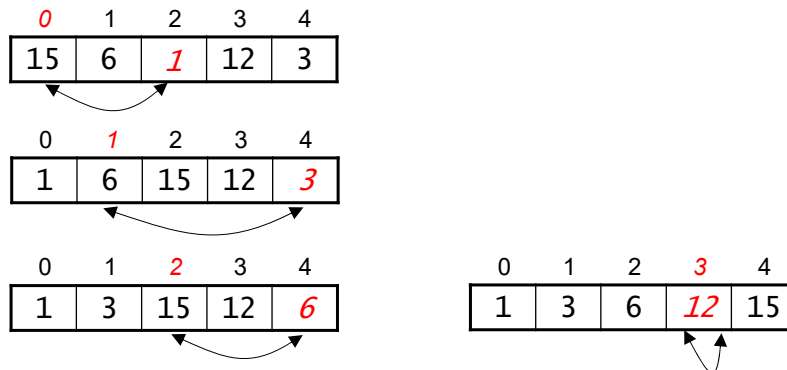
```
private static void swap(int[] arr, int a, int b) {  
    int temp = arr[a];  
    arr[a] = arr[b];  
    arr[b] = temp;  
}
```

- Trace through the following with a memory diagram to convince yourself that it works:

```
int[] arr = {15, 7, ...};  
swap(arr, 0, 1);
```

Selection Sort

- Basic idea:
 - consider the positions in the array from left to right
 - for each position, find the element that belongs there and swap it with the element that's currently there
- Example:



Why don't we need to consider position 4?

Selecting an Element

- When we consider position i , the elements in positions 0 through $i - 1$ are already in their final positions.

example for $i = 3$:

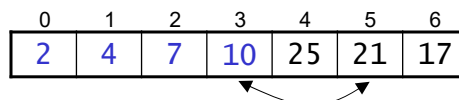
0	1	2	3	4	5	6
2	4	7	21	25	10	17

- To select an element for position i :
 - consider elements $i, i+1, i+2, \dots, \text{arr.length} - 1$, and keep track of `indexMin`, the index of the smallest element seen thus far

`indexMin`: 3, 5

0	1	2	3	4	5	6
2	4	7	21	25	10	17

- when we finish this pass, `indexMin` is the index of the element that belongs in position i .
- swap `arr[i]` and `arr[indexMin]`:



Implementation of Selection Sort

- Use a helper method to find the index of the smallest element:

```
private static int indexSmallest(int[] arr, int start) {  
    int indexMin = start;  
    for (int i = start + 1; i < arr.length; i++) {  
        if (arr[i] < arr[indexMin]) {  
            indexMin = i;  
        }  
    }  
    return indexMin;  
}
```

- The actual sort method is very simple:

```
public static void selectionSort(int[] arr) {  
    for (int i = 0; i < arr.length - 1; i++) {  
        int j = indexSmallest(arr, i);  
        swap(arr, i, j);  
    }  
}
```

Time Analysis

- Some algorithms are much more efficient than others.
- The *time efficiency* or *time complexity* of an algorithm is some measure of the number of operations that it performs.
 - for sorting, we'll focus on comparisons and moves
- We want to characterize how the number of operations depends on the size, n , of the input to the algorithm.
 - for sorting, n is the length of the array
 - how does the number of operations grow as n grows?
- We'll express the number of operations as functions of n
 - $C(n)$ = number of comparisons for an array of length n
 - $M(n)$ = number of moves for an array of length n

Counting Comparisons by Selection Sort

```
private static int indexSmallest(int[] arr, int start){
    int indexMin = start;
    for (int i = start + 1; i < arr.length; i++) {
        if (arr[i] < arr[indexMin]) {
            indexMin = i;
        }
    }
    return indexMin;
}

public static void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        int j = indexSmallest(arr, i);
        swap(arr, i, j);
    }
}
```

- To sort n elements, selection sort performs $n - 1$ passes:
 on 1st pass, it performs ____ comparisons to find `indexSmallest`
 on 2nd pass, it performs ____ comparisons
 ...
 on the $(n-1)$ st pass, it performs 1 comparison
- Adding them up: $C(n) = 1 + 2 + \dots + (n - 2) + (n - 1)$

Counting Comparisons by Selection Sort (cont.)

- The resulting formula for $C(n)$ is the sum of an arithmetic sequence:

$$C(n) = 1 + 2 + \dots + (n - 2) + (n - 1) = \sum_{i=1}^{n-1} i$$

- Formula for the sum of this type of arithmetic sequence:

$$\sum_{i=1}^m i = \frac{m(m+1)}{2}$$

- Thus, we can simplify our expression for $C(n)$ as follows:

$$\begin{aligned} C(n) &= \sum_{i=1}^{n-1} i \\ &= \frac{(n-1)((n-1)+1)}{2} \\ &= \frac{(n-1)n}{2} \end{aligned}$$

$$C(n) = n^2/2 - n/2$$

Focusing on the Largest Term

- When n is large, mathematical expressions of n are dominated by their “largest” term — i.e., the term that grows fastest as a function of n .

• example:

n	$n^2/2$	$n/2$	$n^2/2 - n/2$
10	50	5	45
100	5000	50	4950
10000	50,000,000	5000	49,995,000

- In characterizing the time complexity of an algorithm, we'll focus on the largest term in its operation-count expression.
 - for selection sort, $C(n) = n^2/2 - n/2 \approx n^2/2$
- In addition, we'll typically ignore the coefficient of the largest term (e.g., $n^2/2 \rightarrow n^2$).

Big-O Notation

- We specify the largest term using big-O notation.
 - e.g., we say that $C(n) = n^2/2 - n/2$ is $O(n^2)$

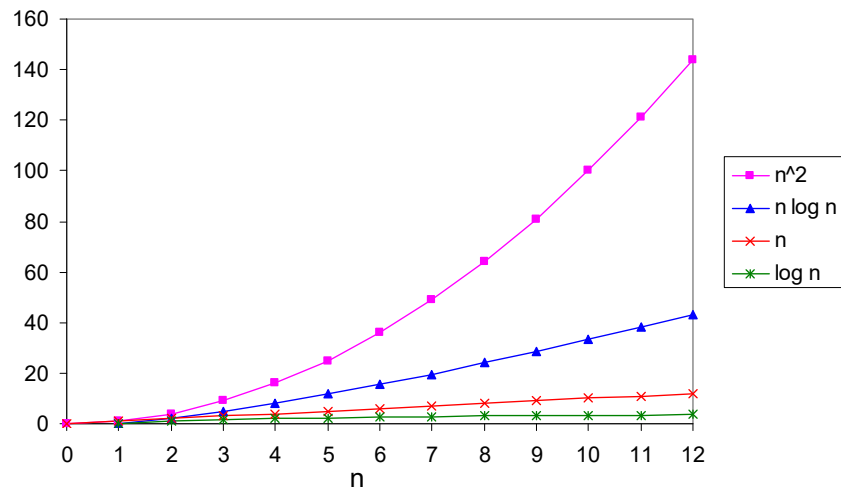
- Common classes of algorithms:

<u>name</u>	<u>example expressions</u>	<u>big-O notation</u>
constant time	1, 7, 10	$O(1)$
logarithmic time	$3\log_{10}n$, $\log_2n + 5$	$O(\log n)$
linear time	$5n$, $10n - 2\log_2n$	$O(n)$
$n \log n$ time	$4n\log_2n$, $n\log_2n + n$	$O(n \log n)$
quadratic time	$2n^2 + 3n$, $n^2 - 1$	$O(n^2)$
exponential time	2^n , $5e^n + 2n^2$	$O(c^n)$

- For large inputs, efficiency matters more than CPU speed.
 - e.g., an $O(\log n)$ algorithm on a slow machine will outperform an $O(n)$ algorithm on a fast machine

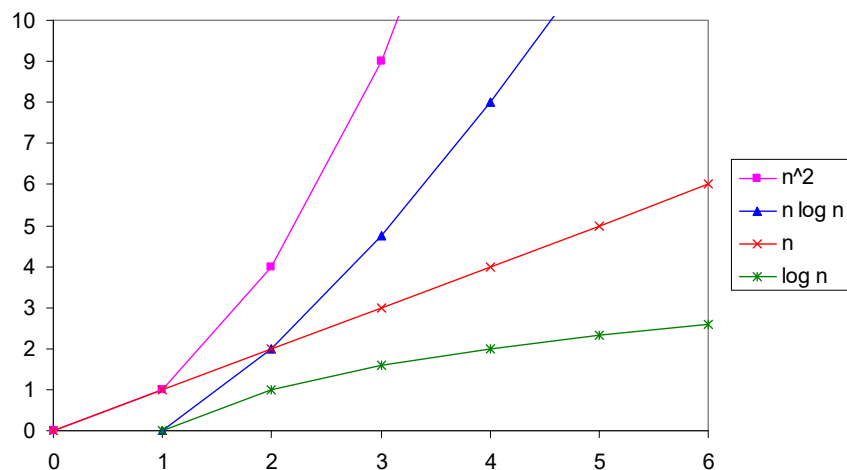
Ordering of Functions

- We can see below that: n^2 grows faster than $n \log_2 n$
 $n \log_2 n$ grows faster than n
 n grows faster than $\log_2 n$



Ordering of Functions (cont.)

- Zooming in, we see that: $n^2 \geq n$ for all $n \geq 1$
 $n \log_2 n \geq n$ for all $n \geq 2$
 $n > \log_2 n$ for all $n \geq 1$



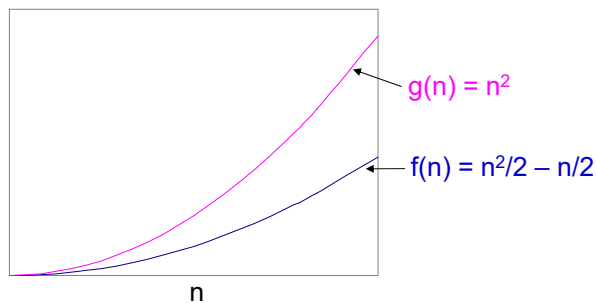
Big-O Time Analysis of Selection Sort

- **Comparisons:** we showed that $c(n) = n^2/2 - n/2$
 - selection sort performs $O(n^2)$ comparisons
- **Moves:** after each of the $n-1$ passes, the algorithm does one swap.
 - $n-1$ swaps, 3 moves per swap
 - $M(n) = 3(n-1) = 3n-3$
 - selection sort performs $O(n)$ moves.
- **Running time (i.e., total operations):** ?

Mathematical Definition of Big-O Notation

- $f(n) = O(g(n))$ if there exist positive constants c and n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$
- Example: $f(n) = n^2/2 - n/2$ is $O(n^2)$, because

$$\underbrace{n^2/2 - n/2}_{c=1} \leq \underbrace{n^2}_{n_0=0} \text{ for all } n \geq 0.$$



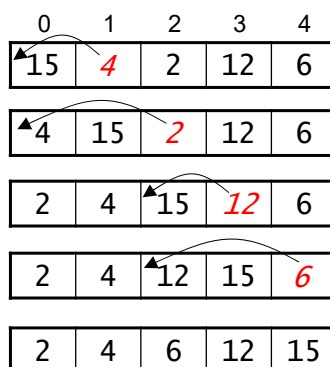
- Big-O notation specifies an *upper bound* on a function $f(n)$ as n grows large.

Big-O Notation and Tight Bounds

- Strictly speaking, big-O notation provides an upper bound, *not* a tight bound (upper and lower).
- Example:
 - $3n - 3$ is $O(n^2)$ because $3n - 3 \leq n^2$ for all $n \geq 1$
 - $3n - 3$ is also $O(2^n)$ because $3n - 3 \leq 2^n$ for all $n \geq 1$
- However, it is common to use big-O notation to characterize a function as closely as possible – as if it specified a tight bound.
 - for our example, we would say that $3n - 3$ is $O(n)$
 - this is how you should use big-O in this class!

Insertion Sort

- Basic idea:
 - going from left to right, “insert” each element into its proper place with respect to the elements to its left
 - “slide over” other elements to make room
- Example:



Comparing Selection and Insertion Strategies

- In selection sort, we start with the *positions* in the array and *select* the correct elements to fill them.
- In insertion sort, we start with the *elements* and determine where to *insert* them in the array.
- Here's an example that illustrates the difference:

0	1	2	3	4	5	6
18	12	15	9	25	2	17

- Sorting by selection:
 - consider position 0: find the element (2) that belongs there
 - consider position 1: find the element (9) that belongs there
 - ...
- Sorting by insertion:
 - consider the 12: determine where to insert it
 - consider the 15; determine where to insert it
 - ...

Inserting an Element

- When we consider element i , elements 0 through $i - 1$ are already sorted with respect to each other.

example for $i = 3$:

0	1	2	3	4
6	14	19	9	...

- To insert element i :
 - make a copy of element i , storing it in the variable `toInsert`:

`toInsert`

9

0	1	2	3
6	14	19	9

- consider elements $i-1$, $i-2$, ...
 - if an element $>$ `toInsert`, slide it over to the right
 - stop at the first element $\leq$ `toInsert`

`toInsert`

9

0	1	2	3
6		14	19

- copy `toInsert` into the resulting "hole":

0	1	2	3
6	9	14	19

Insertion Sort Example (done together)

description of steps

12	5	2	13	18	4
----	---	---	----	----	---

Implementation of Insertion Sort

```
public class Sort {  
    ...  
    public static void insertionsort(int[] arr) {  
        for (int i = 1; i < arr.length; i++) {  
            if (arr[i] < arr[i-1]) {  
                int toInsert = arr[i];  
  
                int j = i;  
                do {  
                    arr[j] = arr[j-1];  
                    j = j - 1;  
                } while (j > 0 && toInsert < arr[j-1]);  
  
                arr[j] = toInsert;  
            }  
        }  
    }  
}
```

Time Analysis of Insertion Sort

- The number of operations depends on the contents of the array.
- best case:** array is sorted
 - each element is only compared to the element to its left
 - we never execute the do-while loop!
 - $C(n) = \text{_____}$, $M(n) = \text{_____}$, running time = _____
- worst case:** array is in reverse order
 - each element is compared to *all* of the elements to its left:
 - $\text{arr}[1]$ is compared to 1 element ($\text{arr}[0]$)
 - $\text{arr}[2]$ is compared to 2 elements ($\text{arr}[0]$ and $\text{arr}[1]$)
 - ...
 - $\text{arr}[n-1]$ is compared to $n-1$ elements
 - $C(n) = 1 + 2 + \dots + (n - 1) = \text{_____}$
 - similarly, $M(n) = \text{_____}$, running time = _____
- average case:** elements are randomly arranged
 - on average, each element is compared to *half* of the elements to its left
 - still get $C(n) = M(n) = \text{_____}$, running time = _____

← also true if array is almost sorted

Shell Sort

- Developed by Donald Shell
- Improves on insertion sort
 - takes advantage of the fact that it's fast for almost-sorted arrays
 - eliminates a key disadvantage: an element may need to move many times to get to where it belongs.
- Example: if the largest element starts out at the beginning of the array, it moves one place to the right on *every* insertion!

0	1	2	3	4	5	...	1000
999	42	56	30	18	23	...	11
- Shell sort uses larger moves that allow elements to quickly get close to where they belong in the sorted array.

Sorting Subarrays

- Basic idea:
 - use insertion sort on subarrays that contain elements separated by some increment $incr$
 - increments allow the data items to make larger “jumps”
 - repeat using a decreasing sequence of increments
- Example for an initial increment of 3:

0	1	2	3	4	5	6	7
36	18	10	27	3	20	9	8

- three subarrays:
 - 1) elements 0, 3, 6
 - 2) elements 1, 4, 7
 - 3) elements 2 and 5
- Sort the subarrays using insertion sort to get the following:

0	1	2	3	4	5	6	7
9	3	10	27	8	20	36	18

- Next, we complete the process using an increment of 1.

Shell Sort: A Single Pass

- We *don't* actually consider the subarrays one at a time.
- For each element from position $incr$ to the end of the array, we insert the element into its proper place with respect to the elements *from its subarray* that come before it.

- The same example ($incr = 3$):

0	1	2	3	4	5	6	7
36	18	10	27	3	20	9	8
27	18	10	36	3	20	9	8
27	3	10	36	18	20	9	8
27	3	10	36	18	20	9	8
9	3	10	27	18	20	36	8
9	3	10	27	8	20	36	18

Inserting an Element in a Subarray

- When we consider element i , the other elements in its subarray are already sorted with respect to each other.

example for $i = 6$:
($\text{incr} = 3$)

0	1	2	3	4	5	6	7
27	3	10	36	18	20	9	8

the other element's in 9's subarray (the 27 and 36) are already sorted with respect to each other

- To insert element i :
 - make a copy of element i , storing it in the variable `toInsert`:

	0	1	2	3	4	5	6	7
toInsert							9	
	27	3	10	36	18	20	9	8

- consider elements $i - \text{incr}$, $i - (2 * \text{incr})$, $i - (3 * \text{incr})$, ...
 - if an element $>$ `toInsert`, slide it right *within the subarray*
 - stop at the first element $\leq$ `toInsert`

	0	1	2	3	4	5	6	7
toInsert							9	
		3	10	27	18	20	36	8

- copy `toInsert` into the "hole":

0	1	2	3	4
9	3	10	27	18
				...

The Sequence of Increments

- Different sequences of decreasing increments can be used.
- Our version uses values that are one less than a power of two.
 - $2^k - 1$ for some k
 - ... 63, 31, 15, 7, 3, 1
 - can get to the next lower increment using integer division:
 $\text{incr} = \text{incr} / 2;$
- Should avoid numbers that are multiples of each other.
 - otherwise, elements that are sorted with respect to each other in one pass are grouped together again in subsequent passes
 - repeat comparisons unnecessarily
 - get fewer of the large jumps that speed up later passes
 - example of a bad sequence: 64, 32, 16, 8, 4, 2, 1
 - what happens if the largest values are all in odd positions?

Implementation of Shell Sort

```
public static void shellSort(int[] arr) {
    int incr = 1;
    while (2 * incr <= arr.length) {
        incr = 2 * incr;
    }
    incr = incr - 1;
    while (incr >= 1) {
        for (int i = incr; i < arr.length; i++) {
            if (arr[i] < arr[i-incr]) {
                int toInsert = arr[i];

                int j = i;
                do {
                    arr[j] = arr[j-incr];
                    j = j - incr;
                } while (j > incr-1 &&
                    toInsert < arr[j-incr]);

                arr[j] = toInsert;
            }
        }
        incr = incr/2;
    }
}
```

(If you replace incr with 1 in the for-loop, you get the code for insertion sort.)

Time Analysis of Shell Sort

- Difficult to analyze precisely
 - typically use experiments to measure its efficiency
- With a bad interval sequence, it's $O(n^2)$ in the worst case.
- With a good interval sequence, it's better than $O(n^2)$.
 - at least $O(n^{1.5})$ in the average and worst case
 - some experiments have shown average-case running times of $O(n^{1.25})$ or even $O(n^{7/6})$

- Significantly better than insertion or selection for large n:

n	n^2	$n^{1.5}$	$n^{1.25}$
10	100	31.6	17.8
100	10,000	1000	316
10,000	100,000,000	1,000,000	100,000
10^6	10^{12}	10^9	3.16×10^7

- We've wrapped insertion sort in another loop and increased its efficiency! The key is in the larger jumps that Shell sort allows.

Practicing Time Analysis

- Consider the following static method:

```
public static int mystery(int n) {
    int x = 0;
    for (int i = 0; i < n; i++) {
        x += i;           // statement 1
        for (int j = 0; j < i; j++) {
            x += j;
        }
    }
    return x;
}
```

- What is the big-O expression for the number of times that statement 1 is executed as a function of the input n ?

What about now?

- Consider the following static method:

```
public static int mystery(int n) {
    int x = 0;
    for (int i = 0; i < 3*n + 4; i++) {
        x += i;           // statement 1
        for (int j = 0; j < i; j++) {
            x += j;
        }
    }
    return x;
}
```

- What is the big-O expression for the number of times that statement 1 is executed as a function of the input n ?

Practicing Time Analysis

- Consider the following static method:

```
public static int mystery(int n) {
    int x = 0;
    for (int i = 0; i < n; i++) {
        x += i;           // statement 1
        for (int j = 0; j < i; j++) {
            x += j;       // statement 2
        }
    }
    return x;
}
```

- What is the big-O expression for the number of times that **statement 2** is executed as a function of the input **n**?
value of i number of times statement 2 is executed

Bubble Sort

- Perform a sequence of passes from left to right
 - each pass swaps adjacent elements if they are out of order
 - larger elements “bubble up” to the end of the array
- At the end of the kth pass:
 - the k rightmost elements are in their final positions
 - we don’t need to consider them in subsequent passes.
- Example:

	0	1	2	3	4
	28	24	37	15	5
<i>after the first pass:</i>	24	28	15	5	37
<i>after the second:</i>	24	15	5	28	37
<i>after the third:</i>	15	5	24	28	37
<i>after the fourth:</i>	5	15	24	28	37

Implementation of Bubble Sort

```
public class Sort {
    ...
    public static void bubbleSort(int[] arr) {
        for (int i = arr.length - 1; i > 0; i--) {
            for (int j = 0; j < i; j++) {
                if (arr[j] > arr[j+1]) {
                    swap(arr, j, j+1);
                }
            }
        }
    }
}
```

- Nested loops:
 - the **inner loop** performs a single pass
 - the **outer loop** governs:
 - the number of passes (`arr.length - 1`)
 - the ending point of each pass (the current value of `i`)

Time Analysis of Bubble Sort

- Comparisons** (n = length of array):
 - they are performed in the inner loop
 - how many repetitions does each execution of the inner loop perform?*

<u>value of i</u>	<u>number of comparisons</u>	
$n - 1$	$n - 1$	} $1 + 2 + \dots + n - 1 =$
$n - 2$	$n - 2$	
...	...	
2	2	
1	1	

```
public static void bubbleSort(int[] arr) {
    for (int i = arr.length - 1; i > 0; i--) {
        for (int j = 0; j < i; j++) {
            if (arr[j] > arr[j+1]) {
                swap(arr, j, j+1);
            }
        }
    }
}
```

Time Analysis of Bubble Sort

- **Comparisons:** the k th pass performs $n - k$ comparisons, so we get $C(n) = \sum_{i=1}^{n-1} i = n^2/2 - n/2 = O(n^2)$
- **Moves:** depends on the contents of the array
 - in the worst case:
 - $M(n) =$
 - in the best case:
- **Running time:**
 - $C(n)$ is always $O(n^2)$, $M(n)$ is never worse than $O(n^2)$
 - therefore, the largest term of $C(n) + M(n)$ is $O(n^2)$
- Bubble sort is a quadratic-time or $O(n^2)$ algorithm.
 - can't do much worse than bubble!

Sorting II: Divide-and-Conquer Algorithms, Distributive Sorting

Computer Science E-22
Harvard Extension School

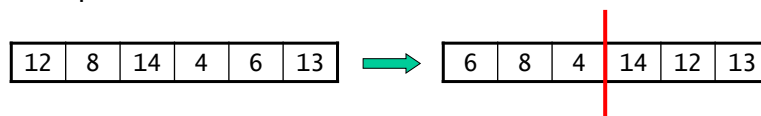
David G. Sullivan, Ph.D.

Quicksort

- Like bubble sort, quicksort uses an approach based on swapping out-of-order elements, but it's more efficient.
- A recursive, divide-and-conquer algorithm:
 - *divide*: rearrange the elements so that we end up with two subarrays that meet the following criterion:

each element in left array $\leq$ each element in right array

example:

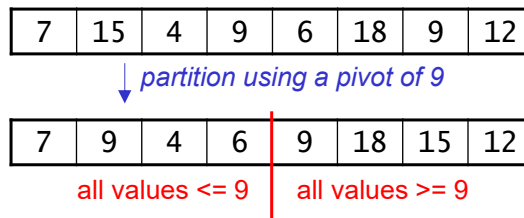


- *conquer*: apply quicksort recursively to the subarrays, stopping when a subarray has a single element
- *combine*: nothing needs to be done, because of the way we formed the subarrays

Partitioning an Array Using a Pivot

- The process that quicksort uses to rearrange the elements is known as *partitioning* the array.
- It uses one of the values in the array as a *pivot*, rearranging the elements to produce two subarrays:
 - left subarray: all values $\leq$ pivot
 - right subarray: all values $\geq$ pivot

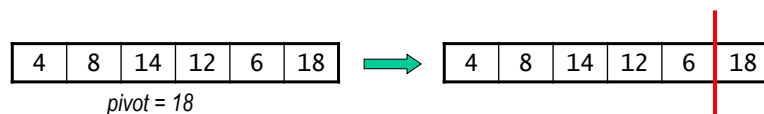
equivalent to the criterion on the previous page.



- The subarrays will *not* always have the same length.
- This approach to partitioning is one of several variants.

Possible Pivot Values

- First element or last element
 - risky, can lead to terrible worst-case behavior
 - especially poor if the array is almost sorted



- Middle element (what we will use)
- Randomly chosen element
- Median of three elements
 - left, center, and right elements
 - three randomly selected elements
 - taking the median of three decreases the probability of getting a poor pivot

arr

--

 →

7	15	4	9	6	18	9	12
---	----	---	---	---	----	---	----

first last

pivot = 9

- $i = \text{first} - 1$ i
- $j = \text{last} + 1$ j
- | | | | | | | | |
|---|----|---|---|---|----|---|----|
| 7 | 15 | 4 | 9 | 6 | 18 | 9 | 12 |
|---|----|---|---|---|----|---|----|

- increment i until $\text{arr}[i] \geq \text{pivot}$
- decrement j until $\text{arr}[j] \leq \text{pivot}$

	i					j	
7	15	4	9	6	18	9	12

- | | | | | | | | |
|---|---|---|---|---|----|----|----|
| | i | | | | | j | |
| 7 | 9 | 4 | 9 | 6 | 18 | 15 | 12 |

from prev. page:

	i					j		
	7	9	4	9	6	18	15	12

- Find:

7	9	4	9	6	18	15	12
---	---	---	---	---	----	----	----

- Swap:

7	9	4	6	9	18	15	12
---	---	---	---	---	----	----	----

- Find:

7	9	4	6	9	18	15	12
---	---	---	---	---	----	----	----

- Subarrays: **left** = from first to j, **right** = from j+1 to last

first			j		i			last
7	9	4	6		9	18	15	12

Partitioning Example 2

- Start
(pivot = 13):

24	5	2	13	18	4	20	19
----	---	---	----	----	---	----	----

 i j
- Find:

24	5	2	13	18	4	20	19
----	---	---	----	----	---	----	----

 i j
- Swap:

4	5	2	13	18	24	20	19
---	---	---	----	----	----	----	----

 i j
- Find:

4	5	2	13	18	24	20	19
---	---	---	----	----	----	----	----

 i j
and now the indices are equal, so we return j .
- Subarrays:

4	5	2	13	18	24	20	19
---	---	---	----	----	----	----	----

 i j

Partitioning Example 3 (done together)

- Start
(pivot = 5):

4	14	7	5	2	19	26	6
---	----	---	---	---	----	----	---

 i j
- Find:

4	14	7	5	2	19	26	6
---	----	---	---	---	----	----	---

Partitioning Example 4

- Start
(pivot = 15):

8	10	7	15	20	9	6	18
---	----	---	----	----	---	---	----

ij
- Find:

8	10	7	15	20	9	6	18
---	----	---	----	----	---	---	----

partition() Helper Method

```
private static int partition(int[] arr, int first, int last)
{
    int pivot = arr[(first + last)/2];
    int i = first - 1; // index going left to right
    int j = last + 1;  // index going right to left
    while (true) {
        do {
            i++;
        } while (arr[i] < pivot);
        do {
            j--;
        } while (arr[j] > pivot);
        if (i < j) {
            swap(arr, i, j);
        } else {
            return j; // arr[j] = end of left array
        }
    }
}
```

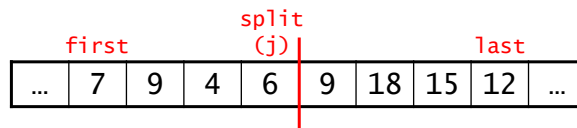
			first						last			
...	7	15	4	9	6	18	9	12	...			

Implementation of Quicksort

```
public static void quickSort(int[] arr) { // "wrapper" method
    if (arr.length <= 1) {
        return;
    }
    qSort(arr, 0, arr.length - 1);
}

private static void qSort(int[] arr, int first, int last) {
    int split = partition(arr, first, last);

    if (first < split) { // if left subarray has 2+ values
        qSort(arr, first, split); // sort it recursively!
    }
    if (last > split + 1) { // if right has 2+ values
        qSort(arr, split + 1, last); // sort it!
    }
} // note: base case is when neither call is made!
```



A Quick Review of Logarithms

- $\log_b n$ = the exponent to which b must be raised to get n
 - $\log_b n = p$ if $b^p = n$
 - examples: $\log_2 8 = 3$ because $2^3 = 8$
 $\log_{10} 10000 = 4$ because $10^4 = 10000$
- Another way of looking at $\log_2 n$:
 - let's say that you repeatedly divide n by 2 (using integer division)
 - $\log_2 n$ is an upper bound on the number of divisions needed to reach 1
 - example: $\log_2 18$ is approx. 4.17
 $18/2 = 9$ $9/2 = 4$ $4/2 = 2$ $2/2 = 1$

A Quick Review of Logs (cont.)

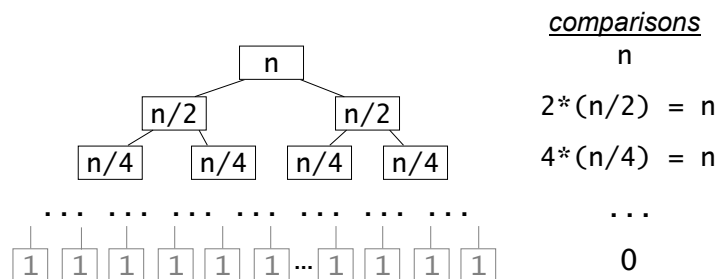
- $O(\log n)$ algorithm – one in which the number of operations is proportional to $\log_b n$ for any base b
- $\log_b n$ grows much more slowly than n

n	$\log_2 n$
2	1
1024 (1K)	10
1024*1024 (1M)	20
1024*1024*1024 (1G)	30

- Thus, for large values of n :
 - a $O(\log n)$ algorithm is much faster than a $O(n)$ algorithm
 - $\log n \ll n$
 - a $O(n \log n)$ algorithm is much faster than a $O(n^2)$ algorithm
 - $n * \log n \ll n * n$ it's also faster than a $O(n^{1.5})$ algorithm like Shell sort
 - $n \log n \ll n^2$

Time Analysis of Quicksort

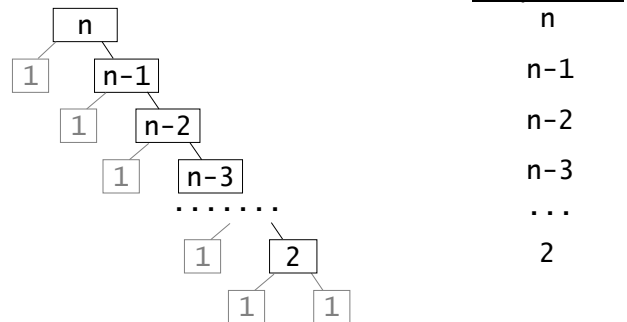
- Partitioning an array of length n requires approx. n comparisons.
 - most elements are compared with the pivot once; a few twice
- **best case:** partitioning always divides the array in half
 - repeated recursive calls give:



- at each "row" except the bottom, we perform n comparisons
- there are _____ rows that include comparisons
- $C(n) = ?$
- Similarly, $M(n)$ and running time are both _____

Time Analysis of Quicksort (cont.)

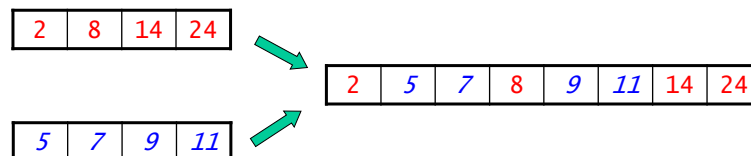
- **worst case:** pivot is always the smallest or largest element
 - one subarray has 1 element, the other has $n - 1$
 - repeated recursive calls give:



- $c(n) = \sum_{i=2}^n i = O(n^2)$. $M(n)$ and run time are also $O(n^2)$.
- **average case** is harder to analyze
 - $C(n) > n \log_2 n$, but it's still $O(n \log n)$

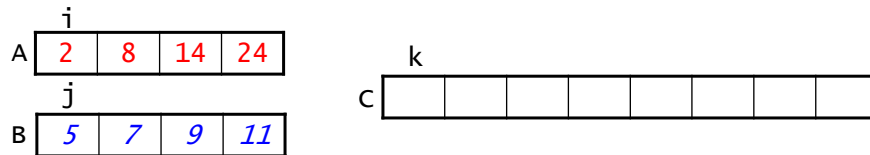
Mergesort

- The algorithms we've seen so far have sorted the array in place.
 - use only a small amount of additional memory
- Mergesort requires an additional temporary array of the same size as the original one.
 - it needs $O(n)$ additional space, where n is the array size
- It is based on the process of *merging* two sorted arrays.
 - example:

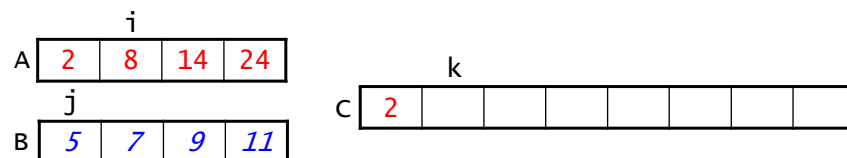


Merging Sorted Arrays

- To merge sorted arrays A and B into an array C, we maintain three indices, which start out on the first elements of the arrays:

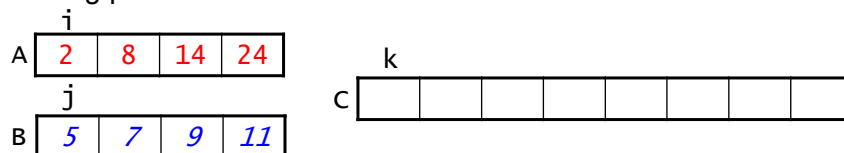


- We repeatedly do the following:
 - compare $A[i]$ and $B[j]$
 - copy the smaller of the two to $C[k]$
 - increment the index of the array whose element was copied
 - increment k

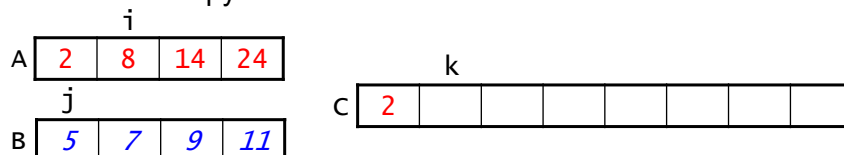


Merging Sorted Arrays (cont.)

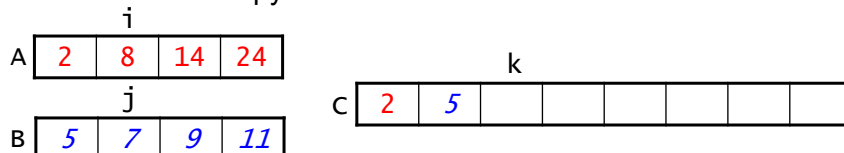
- Starting point:



- After the first copy:

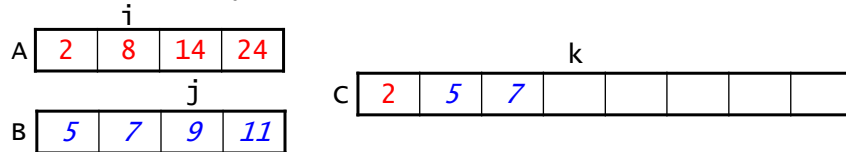


- After the second copy:

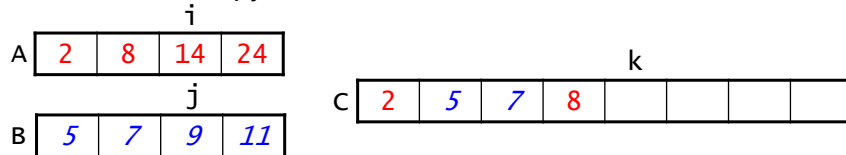


Merging Sorted Arrays (cont.)

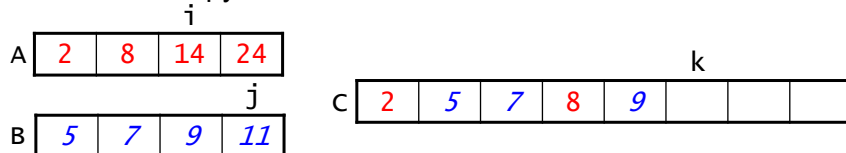
- After the third copy:



- After the fourth copy:

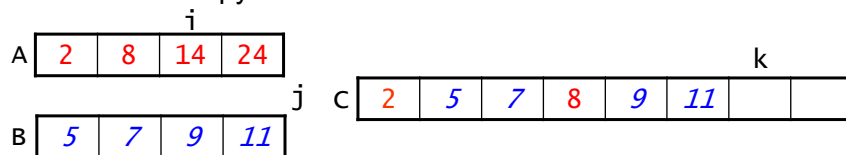


- After the fifth copy:

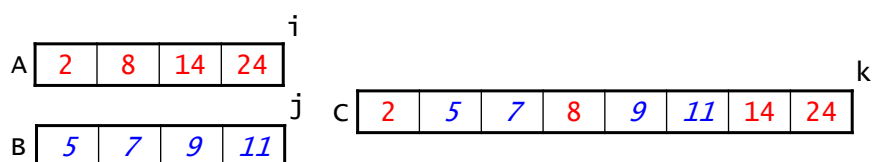


Merging Sorted Arrays (cont.)

- After the sixth copy:

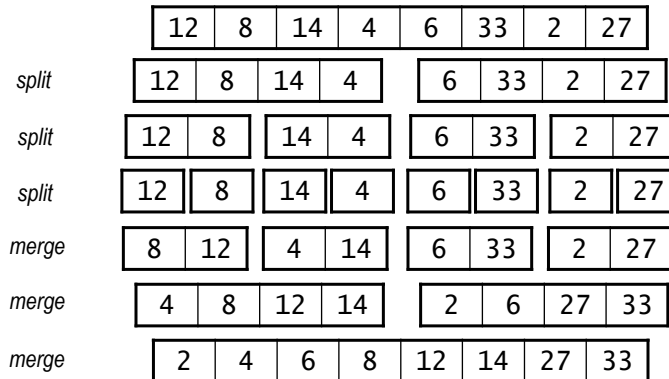


- There's nothing left in B, so we simply copy the remaining elements from A:



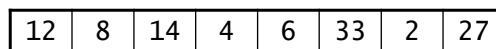
Divide and Conquer

- Like quicksort, mergesort is a divide-and-conquer algorithm.
 - divide*: split the array in half, forming two subarrays
 - conquer*: apply mergesort recursively to the subarrays, stopping when a subarray has a single element
 - combine*: merge the sorted subarrays

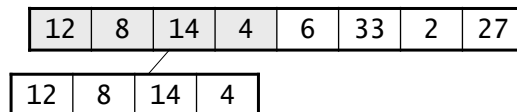


Tracing the Calls to Mergesort

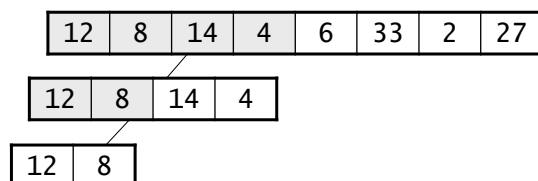
the initial call is made to sort the entire array:



split into two 4-element subarrays, and make a recursive call to sort the left subarray:

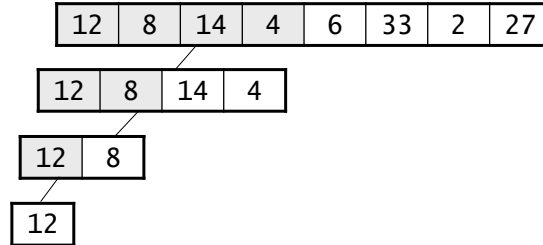


split into two 2-element subarrays, and make a recursive call to sort the left subarray:

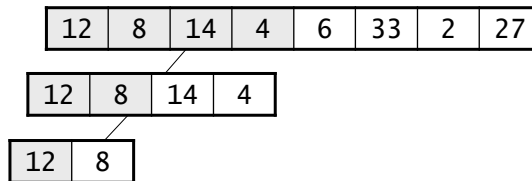


Tracing the Calls to Mergesort

split into two 1-element subarrays, and make a recursive call to sort the left subarray:

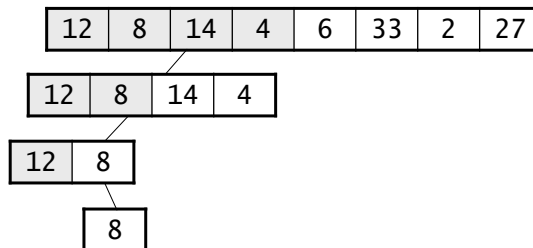


base case, so return to the call for the subarray {12, 8}:

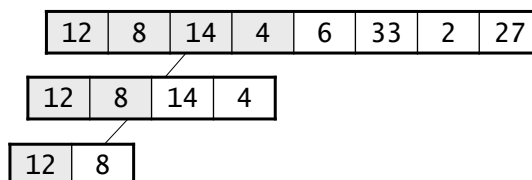


Tracing the Calls to Mergesort

make a recursive call to sort its right subarray:

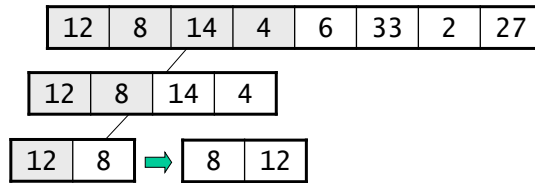


base case, so return to the call for the subarray {12, 8}:

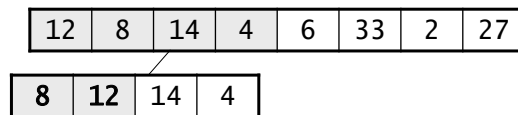


Tracing the Calls to Mergesort

merge the sorted halves of {12, 8}:

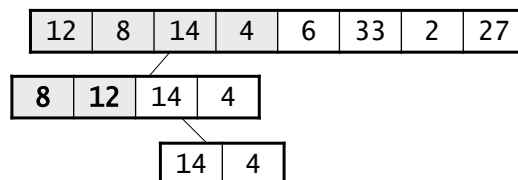


end of the method, so return to the call for the 4-element subarray, which now has a sorted left subarray:

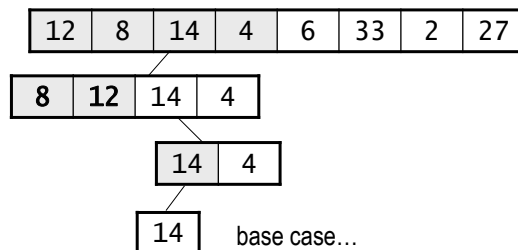


Tracing the Calls to Mergesort

make a recursive call to sort the right subarray of the 4-element subarray

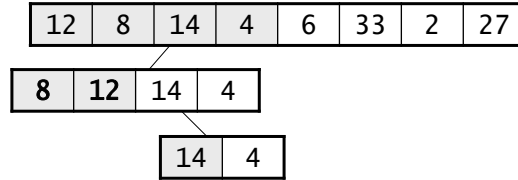


split it into two 1-element subarrays, and make a recursive call to sort the left subarray:

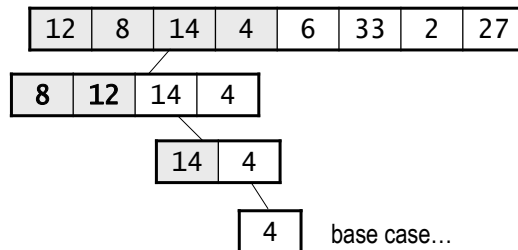


Tracing the Calls to Mergesort

return to the call for the subarray {14, 4}:

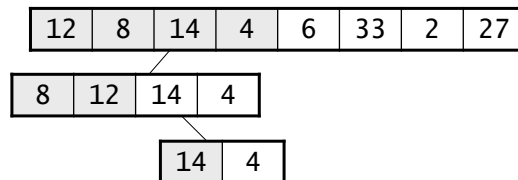


make a recursive call to sort its right subarray:

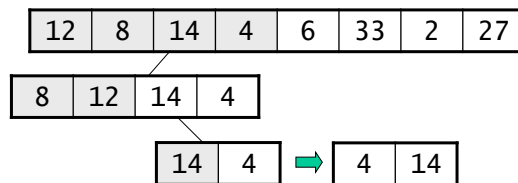


Tracing the Calls to Mergesort

return to the call for the subarray {14, 4}:

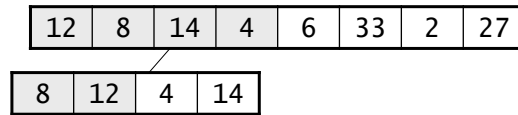


merge the sorted halves of {14, 4}:

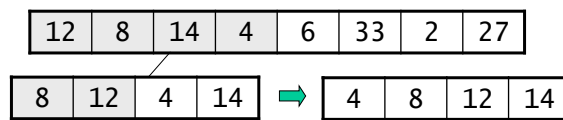


Tracing the Calls to Mergesort

end of the method, so return to the call for the 4-element subarray, which now has two sorted 2-element subarrays:

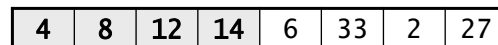


merge the 2-element subarrays:

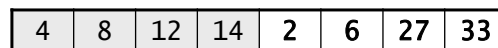


Tracing the Calls to Mergesort

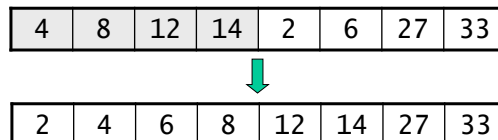
end of the method, so return to the call for the original array, which now has a sorted left subarray:



perform a similar set of recursive calls to sort the right subarray. here's the result:

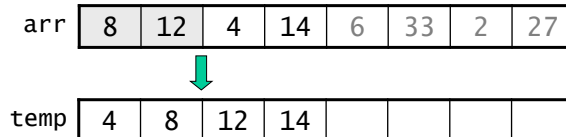


finally, merge the sorted 4-element subarrays to get a fully sorted 8-element array:

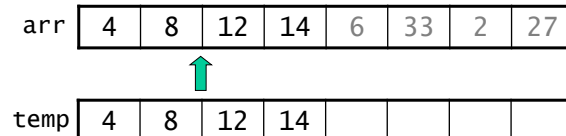


Implementing Mergesort

- In theory, we could create new arrays for each new pair of subarrays, and merge them back into the array that was split.
- Instead, we'll create a temp. array of the same size as the original.
 - pass it to each call of the recursive mergesort method
 - use it when merging subarrays of the original array:



- after each merge, copy the result back into the original array:



A Method for Merging Subarrays

```
private static void merge(int[] arr, int[] temp,
    int leftStart, int leftEnd, int rightStart, int rightEnd) {
    int i = leftStart;    // index into left subarray
    int j = rightStart;   // index into right subarray
    int k = leftStart;    // index into temp

    while (i <= leftEnd && j <= rightEnd) {
        if (arr[i] < arr[j]) {
            temp[k] = arr[i];
            i++; k++;
        } else {
            temp[k] = arr[j];
            j++; k++;
        }
    }

    while (i <= leftEnd) {
        temp[k] = arr[i];
        i++; k++;
    }

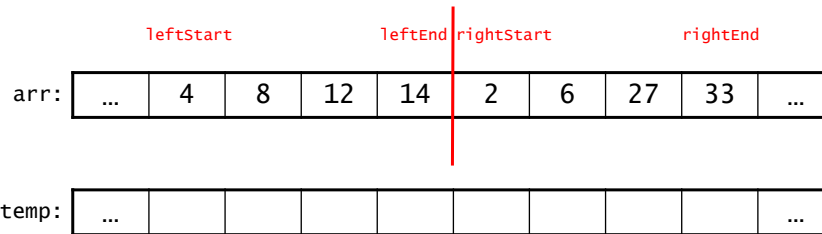
    while (j <= rightEnd) {
        temp[k] = arr[j];
        j++; k++;
    }

    for (i = leftStart; i <= rightEnd; i++) {
        arr[i] = temp[i];
    }
}
```

A Method for Merging Subarrays

```
private static void merge(int[] arr, int[] temp,
    int leftStart, int leftEnd, int rightStart, int rightEnd) {
    int i = leftStart;    // index into left subarray
    int j = rightStart;   // index into right subarray
    int k = leftStart;    // index into temp

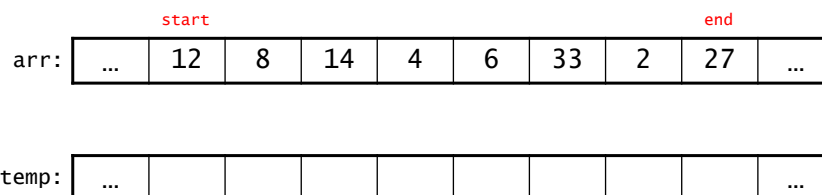
    while (i <= leftEnd && j <= rightEnd) { // both subarrays still have values
        if (arr[i] < arr[j]) {
            temp[k] = arr[i];
            i++; k++;
        } else {
            temp[k] = arr[j];
            j++; k++;
        }
    }
    ...
}
```



Methods for Mergesort

- Here's the key recursive method:

```
private static void mSort(int[] arr, int[] temp, int start, int end){
    if (start >= end) { // base case: subarray of length 0 or 1
        return;
    } else {
        int middle = (start + end)/2;
        mSort(arr, temp, start, middle);
        mSort(arr, temp, middle + 1, end);
        merge(arr, temp, start, middle, middle + 1, end);
    }
}
```



Methods for Mergesort

- Here's the key recursive method:

```
private static void mSort(int[] arr, int[] temp, int start, int end){
    if (start >= end) { // base case: subarray of length 0 or 1
        return;
    } else {
        int middle = (start + end)/2;

        mSort(arr, temp, start, middle);
        mSort(arr, temp, middle + 1, end);

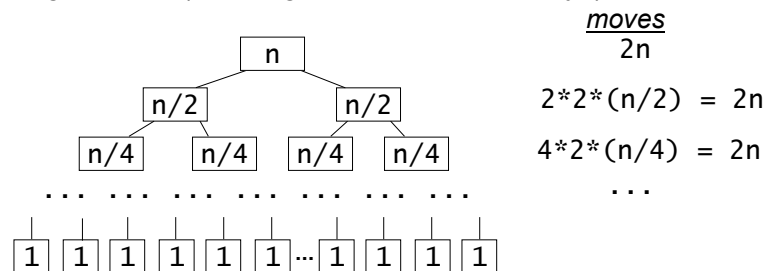
        merge(arr, temp, start, middle, middle + 1, end);
    }
}
```

- We use a "wrapper" method to create the temp array, and to make the initial call to the recursive method:

```
public static void mergeSort(int[] arr) {
    int[] temp = new int[arr.length];
    mSort(arr, temp, 0, arr.length - 1);
}
```

Time Analysis of Mergesort

- Merging two halves of an array of size n requires $2n$ moves. Why?
- Mergesort repeatedly divides the array in half, so we have the following call tree (showing the sizes of the arrays):



- at all but the last level of the call tree, there are $2n$ moves
- how many levels are there?
- $M(n) = ?$
- $C(n) = ?$

Summary: Sorting Algorithms

algorithm	best case	avg case	worst case	extra memory
selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Shell sort	$O(n \log n)$	$O(n^{1.5})$	$O(n^{1.5})$	$O(1)$
bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	best/avg: $O(\log n)$ worst: $O(n)$
mergesort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$

- Insertion sort is best for nearly sorted arrays.
- Mergesort has the best worst-case complexity, but requires $O(n)$ extra memory – and moves to and from the temp. array.
- Quicksort is comparable to mergesort in the best/average case.
 - efficiency is also $O(n \log n)$, but less memory and fewer moves
 - its extra memory is from...
 - with a reasonable pivot choice, its worst case is seldom seen

Comparison-Based vs. Distributive Sorting

- All of the sorting algorithms we've considered have been *comparison-based*:
 - treat the values being sorted as wholes (comparing them)
 - don't "take them apart" in any way
 - all that matters is the relative order of the values
- No comparison-based sorting algorithm can do better than $O(n \log_2 n)$ on an array of length n .
 - $O(n \log_2 n)$ is a *lower bound* for such algorithms
- *Distributive* sorting algorithms do more than compare values; they perform calculations on the values being sorted.
- Moving beyond comparisons allows us to overcome the lower bound.
 - tradeoff: use more memory.

Distributive Sorting Example: Radix Sort

- Breaks each value into a sequence of **m** components, each of which has **k** possible values.
- Examples:

	<u>m</u>	<u>k</u>
• integer in range 0 ... 999	3	10
• string of 15 upper-case letters	15	26
• 32-bit integer	32	2 (in binary)
	4	256 (as bytes)
- Strategy: Distribute the values into "bins" according to their last component, then concatenate the results:

33 41 12 24 31 14 13 42 34

get: 41 31 | 12 42 | 33 13 | 24 14 34

- Repeat, moving back one component each time:

get: | | |

Analysis of Radix Sort

- m = number of components
 k = number of possible values for each component
 n = length of the array
- Time efficiency: $O(m \cdot n)$
 - perform m distributions, each of which processes all n values
 - $O(m \cdot n) < O(n \log n)$ when $m < \log n$
 so we want m to be small
- However, there is a tradeoff:
 - as m decreases, k increases
 - fewer components $\rightarrow$ more possible values per component
 - as k increases, so does memory usage
 - need more bins for the results of each distribution
 - increased speed requires increased memory usage

Big-O Notation Revisited

- We've seen that we can group functions into classes by focusing on the fastest-growing term in the expression for the number of operations that they perform.
 - e.g., an algorithm that performs $n^2/2 - n/2$ operations is a $O(n^2)$ -time or quadratic-time algorithm
- Common classes of algorithms:

<u>name</u>	<u>example expressions</u>	<u>big-O notation</u>
constant time	1, 7, 10	$O(1)$
logarithmic time	$3\log_{10}n$, $\log_2n + 5$	$O(\log n)$
linear time	$5n$, $10n - 2\log_2n$	$O(n)$
$n\log n$ time	$4n\log_2n$, $n\log_2n + n$	$O(n\log n)$
quadratic time	$2n^2 + 3n$, $n^2 - 1$	$O(n^2)$
cubic time	$n^2 + 3n^3$, $5n^3 - 5$	$O(n^3)$
exponential time	2^n , $5e^n + 2n^2$	$O(c^n)$
factorial time	$3n!$, $5n + n!$	$O(n!)$

slower
↓

How Does the Number of Operations Scale?

- Let's say that we have a problem size of 1000, and we measure the number of operations performed by a given algorithm.
- If we double the problem size to 2000, how would the number of operations performed by an algorithm increase if it is:
 - $O(n)$ -time
 - $O(n^2)$ -time
 - $O(n^3)$ -time
 - $O(\log_2n)$ -time
 - $O(2^n)$ -time

How Does the Actual Running Time Scale?

- How much time is required to solve a problem of size n ?
 - assume that each operation requires $1 \mu\text{sec}$ ($1 \times 10^{-6} \text{ sec}$)

time function	problem size (n)					
	10	20	30	40	50	60
n	.00001 s	.00002 s	.00003 s	.00004 s	.00005 s	.00006 s
n^2	.0001 s	.0004 s	.0009 s	.0016 s	.0025 s	.0036 s
n^5	.1 s	3.2 s	24.3 s	1.7 min	5.2 min	13.0 min
2^n	.001 s	1.0 s	17.9 min	12.7 days	35.7 yrs	36,600 yrs

- sample computations:
 - when $n = 10$, an n^2 algorithm performs 10^2 operations.
 $10^2 * (1 \times 10^{-6} \text{ sec}) = .0001 \text{ sec}$
 - when $n = 30$, a 2^n algorithm performs 2^{30} operations.
 $2^{30} * (1 \times 10^{-6} \text{ sec}) = 1073 \text{ sec} = 17.9 \text{ min}$

What's the Largest Problem That Can Be Solved?

- What's the largest problem size n that can be solved in a given time T ? (again assume $1 \mu\text{sec}$ per operation)

time function	time available (T)			
	1 min	1 hour	1 week	1 year
n	60,000,000	3.6×10^9	6.0×10^{11}	3.1×10^{13}
n^2	7745	60,000	777,688	5,615,692
n^5	35	81	227	500
2^n	25	31	39	44

- sample computations:
 - 1 hour = 3600 sec
 that's enough time for $3600 / (1 \times 10^{-6}) = 3.6 \times 10^9$ operations
 - n^2 algorithm:
 $n^2 = 3.6 \times 10^9 \rightarrow n = (3.6 \times 10^9)^{1/2} = 60,000$
 - 2^n algorithm:
 $2^n = 3.6 \times 10^9 \rightarrow n = \log_2(3.6 \times 10^9) \approx 31$

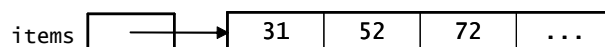
Linked Lists

Computer Science E-22
Harvard University

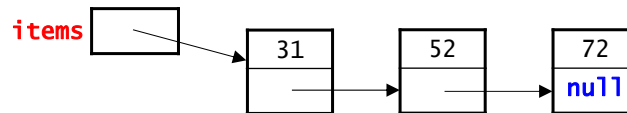
David G. Sullivan, Ph.D.

Representing a Sequence of Data

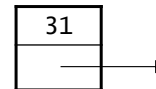
- Sequence – an ordered collection of items (position matters)
 - we will look at several types: lists, stacks, and queues
- Most common representation = an array
- Advantages of using an array:
 - easy and efficient access to *any* item in the sequence
 - `items[i]` gives you the item at position *i* in $O(1)$ time
 - known as *random access*
 - very compact (but can waste space if positions are empty)
- Disadvantages of using an array:
 - have to specify an initial array size and resize it as needed
 - inserting/deleting items can require shifting other items
 - ex: insert 63 between 52 and 72



Alternative Representation: A Linked List

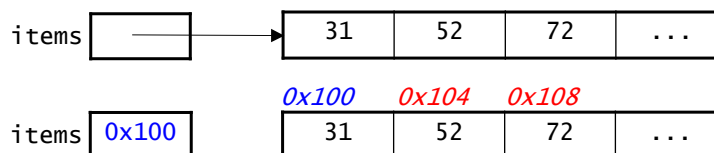


- A linked list stores a sequence of items in separate *nodes*.
- Each node is an object that contains:
 - a single item
 - a "link" (i.e., a reference) to the node containing the next item
- The last node in the linked list has a link value of `null`.
- The linked list as a whole is represented by a variable that holds a reference to the first node.
 - e.g., `items` in the example above

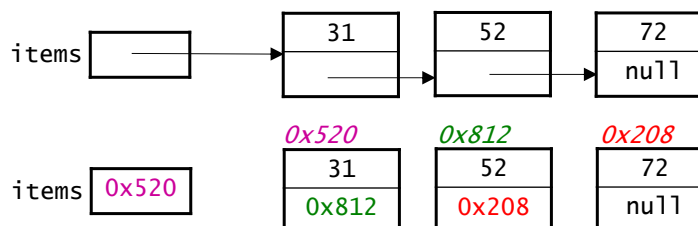


Arrays vs. Linked Lists in Memory

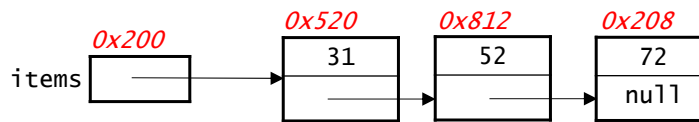
- In an array, the elements occupy consecutive memory locations:



- In a linked list, the nodes are distinct objects.
 - do *not* have to be next to each other in memory
 - that's why we need the links to get from one node to the next!



Linked Lists in Memory

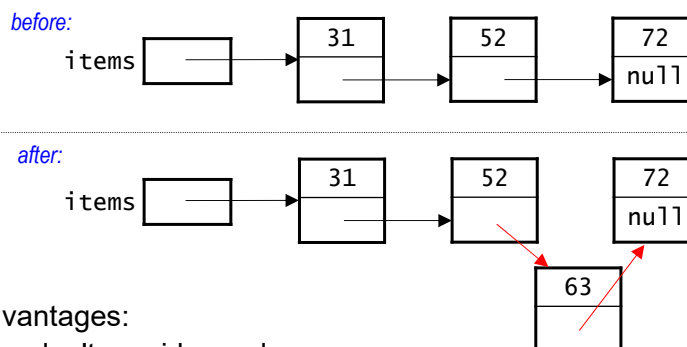


- Here's how the above linked list might actually look in memory:

0x200	0x520	← the variable items
0x204		
0x208	72	} the last node
0x212	null	
0x216		
...	...	
0x520	31	} the first node
0x524	0x812	
0x528		
...	...	
0x812	52	} the second node
0x816	0x208	

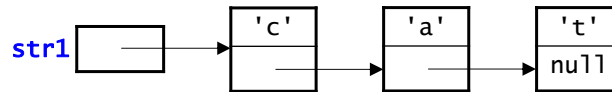
Features of Linked Lists

- They can grow without limit (provided there is enough memory).
- Easy to insert/delete an item – no need to "shift over" other items.
 - for example, to insert 63 between 52 and 72:



- Disadvantages:
 - they don't provide random access
 - need to "walk down" the list to access an item
 - the links take up additional memory

A String as a Linked List of Characters



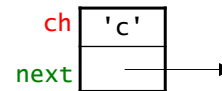
- Each node represents one character.

- Java class for this type of node:

```

public class StringNode {
    private char ch;
    private StringNode next;

    public StringNode(char c, StringNode n) {
        this.ch = c;
        this.next = n;
    }
    ...
}
  
```



same type as the node itself!

- The string as a whole is represented by a variable that holds a reference to the node for the first character (e.g., `str1` above).

A String as a Linked List (cont.)

- An empty string will be represented by a null value.

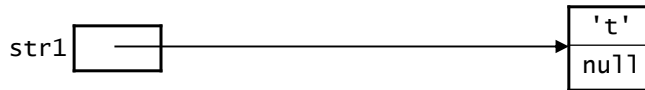
example:

```
StringNode str2 = null;
```

- We will use *static* methods that take the string as a parameter.
 - e.g., we'll write `length(str1)` instead of `str1.length()`
 - outside the class, call the methods using the class name: `StringNode.length(str1)`
- This approach allows the methods to handle empty strings.
 - if `str1 == null`:
 - `length(str1)` will work
 - `str1.length()` will throw a `NullPointerException`

Building a Linked List of Characters I

```
public StringNode(char c,
                  StringNode n) {
    this.ch = c;
    this.next = n;
}
```

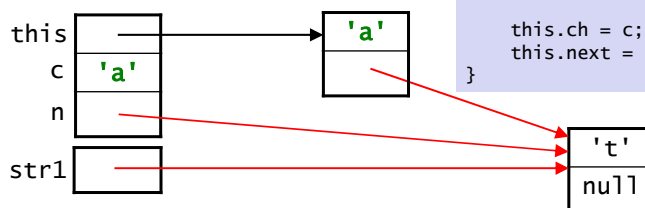


- We can use the `StringNode` constructor to build the linked list from the previous slide.
- One way is to start with the last node and work towards the front:

```
StringNode str1 = new StringNode('t', null);
```

Building a Linked List of Characters II

```
public StringNode(char c,
                  StringNode n) {
    this.ch = c;
    this.next = n;
}
```

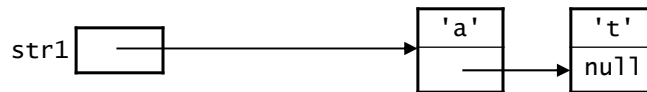


- We can use the `StringNode` constructor to build the linked list from the previous slide.
- One way is to start with the last node and work towards the front:

```
StringNode str1 = new StringNode('t', null);
str1 = new StringNode('a', str1);
```

Building a Linked List of Characters III

```
public StringNode(char c,
                  StringNode n) {
    this.ch = c;
    this.next = n;
}
```

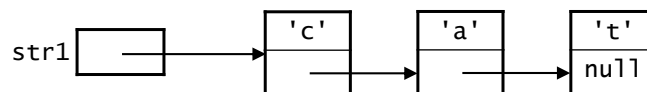


- We can use the `StringNode` constructor to build the linked list from the previous slide.
- One way is to start with the last node and work towards the front:

```
StringNode str1 = new StringNode('t', null);
str1 = new StringNode('a', str1);
```

Building a Linked List of Characters IV

```
public StringNode(char c,
                  StringNode n) {
    this.ch = c;
    this.next = n;
}
```



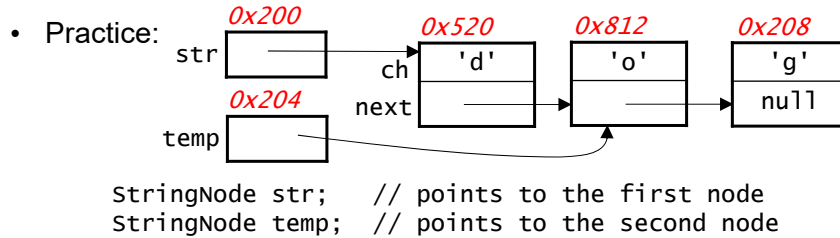
- We can use the `StringNode` constructor to build the linked list from the previous slide.
- One way is to start with the last node and work towards the front:

```
StringNode str1 = new StringNode('t', null);
str1 = new StringNode('a', str1);
str1 = new StringNode('c', str1);
```

- Later, we'll see methods that can be used to build a linked list and add nodes to it.

Review of Variables

- A variable or variable expression represents both:
 - a "box" or location in memory (the *address* of the variable)
 - the contents of that "box" (the *value* of the variable)

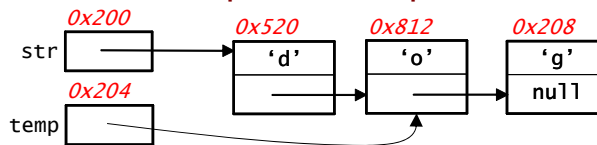


expression	address	value
str	0x200	0x520 (ref to the 'd' node)
str.ch		
str.next		

Assumptions:

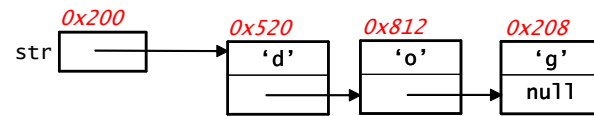
- ch field has the same memory address as the node itself.
- next field comes 2 bytes after the start of the node.

More Complicated Expressions



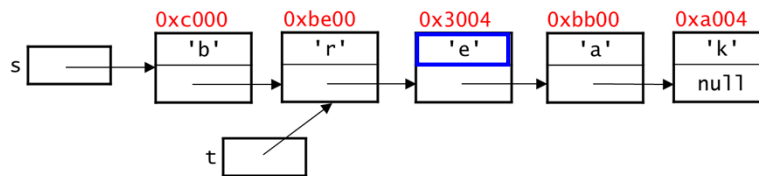
- Example: `temp.next.ch`
- Start with the beginning of the expression: `temp.next`
It represents the next field of the node to which `temp` refers.
 - address =
 - value =
- Next, consider `temp.next.ch`
It represents the `ch` field of the node to which `temp.next` refers.
 - address =
 - value =

What are the address and value of `str.next.next`?

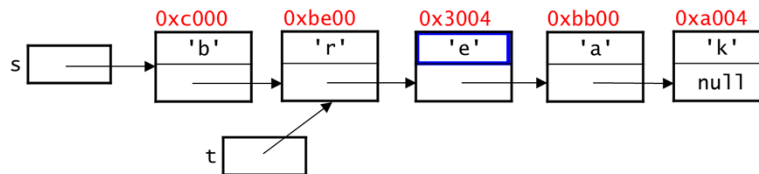


- `str.next` is...
- thus, `str.next.next` is...

What expression using `t` would give us 'e'?



What expression using t would give us 'e'?



Working backwards...

- I know that I need the ch field in the 'e' node
- Where do I have a reference to the 'e' node?
- What expression can I use for the box containing that reference?

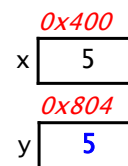
Review of Assignment Statements

- An assignment of the form
`var1 = var2;`
 - takes the value inside var2
 - copies it into var1

- Example involving integers:

```

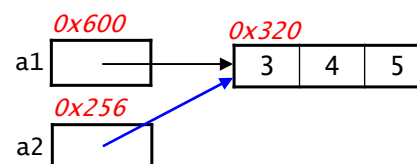
int x = 5;
int y = x;
      5
  
```



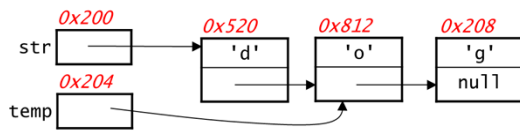
- Example involving references:

```

int[] a1 = {3, 4, 5};
int[] a2 = a1;
          0x320
  
```



What About These Assignments?



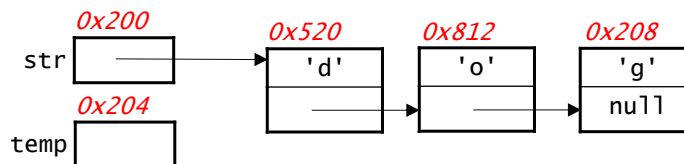
- Identify the two boxes.
- Determine the value in the box specified by the right-hand side.
- Copy that value into the box specified by the left-hand side.

1) `str.next = temp.next;`

2) `temp.next = temp.next.next;`

Writing an Appropriate Assignment

- If `temp` didn't already refer to the 'o' node, what assignment would be needed to make it refer to that node?



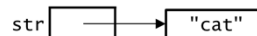
- start by asking: where do I currently have a reference to the 'o' node?
- then ask: what expression can I use for that box?
- then write the assignment:

A Linked List Is a Recursive Data Structure!

- Recursive definition: a linked list is either
 - a) empty or
 - b) a single node, followed by a linked list
- Viewing linked lists in this way allows us to write recursive methods that operate on linked lists.

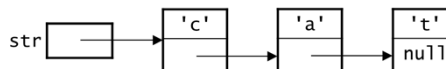
Recursively Finding the Length of a String

- For a Java String object:



```
public static int length(String str) {  
    if (str.equals("")) {  
        return 0;  
    } else {  
        int lenRest = length(str.substring(1));  
        return 1 + lenRest;  
    }  
}
```

- For a linked-list string:



```
public static int length(StringNode str) {  
    if (str == null) {  
        return 0;  
    } else {  
        int lenRest = length(str.next);  
        return 1 + lenRest;  
    }  
}
```

An Alternative Version of the Method

- Original version:

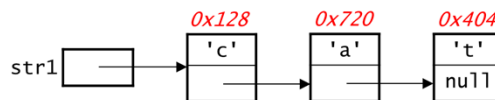
```
public static int length(StringNode str) {
    if (str == null) {
        return 0;
    } else {
        int lenRest = length(str.next);
        return 1 + lenRest;
    }
}
```

- Version without a variable for the result of the recursive call:

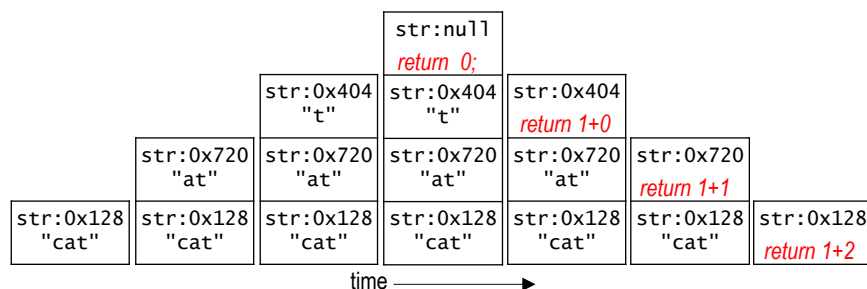
```
public static int length(StringNode str) {
    if (str == null) {
        return 0;
    } else {
        return 1 + length(str.next);
    }
}
```

Tracing length()

```
public static int length(StringNode str) {
    if (str == null) {
        return 0;
    } else {
        return 1 + length(str.next);
    }
}
```

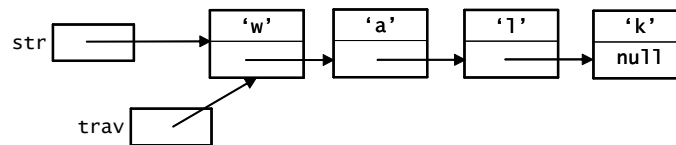


- Example: stringNode.length(str1)



Using Iteration to Traverse a Linked List

- Many tasks require us to traverse or "walk down" a linked list.
- We just saw a method that used recursion to do this.
- It can also be done using iteration (for loops, while loops, etc.).
- We make use of a variable (call it `trav`) that keeps track of where we are in the linked list.

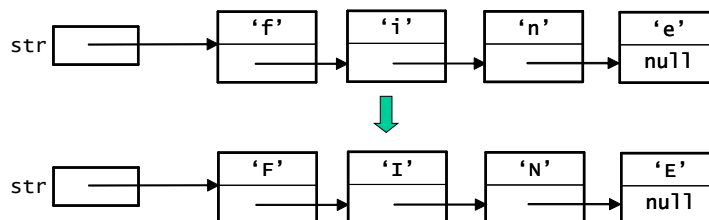


- Template for traversing an entire linked list:

```
StringNode trav = str;    // start with first node
while (trav != null) {
    // process the current node here
    trav = trav.next;    // move trav to next node
}
```

Example of Iterative Traversal

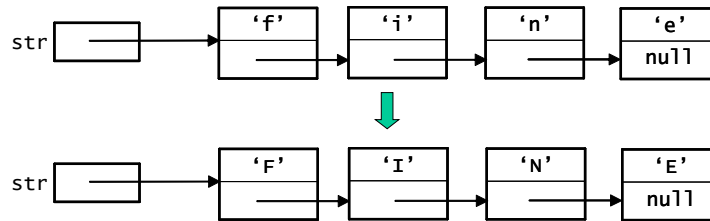
- `toUpperCase(str)`: converting `str` to all upper-case letters



- Similar to the built-in method for Java `String` objects.
- This method processes linked-list strings:
 - uses a loop to process one `StringNode` at a time
 - modifies the internals of the string (unlike the built-in version)
 - thus, it doesn't need to return anything

Example of Iterative Traversal (cont.)

- toUpperCase(str): converting str to all upper-case letters

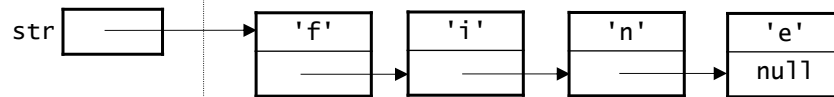


- Here's the method:

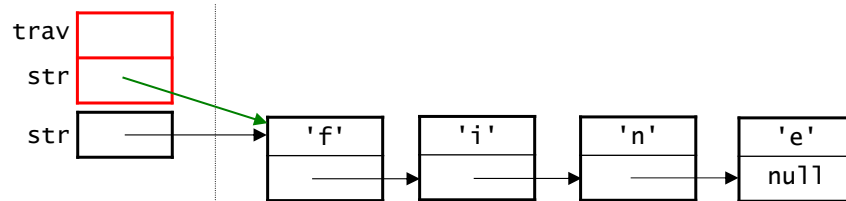
```
public static void toUpperCase(StringNode str) {
    StringNode trav = str;
    while (trav != null) {
        trav.ch = Character.toUpperCase(trav.ch);
        trav = trav.next;
    }
}
```

- uses a built-in static method from the Character class to convert a single char to upper case

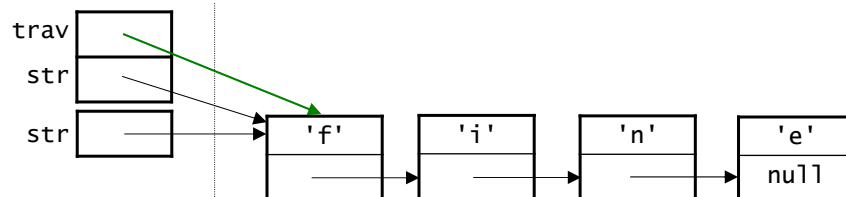
Tracing toUpperCase(): Before the Loop



Calling StringNode.toUpperCase(str) adds a stack frame to the stack:



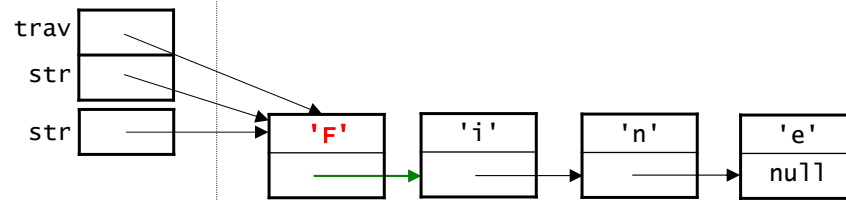
StringNode trav = str;



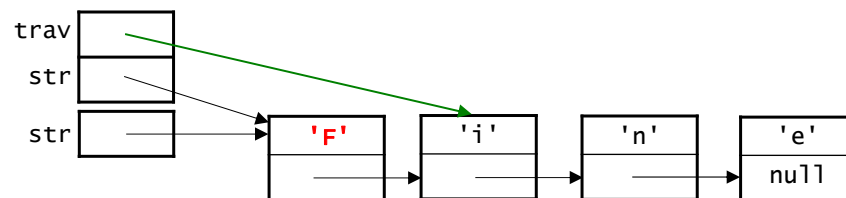
Tracing toUpperCase(): First Iteration of Loop

```
while (trav != null) {  
    trav.ch = Character.toUpperCase(trav.ch);  
    trav = trav.next;  
}
```

after updating trav.ch:



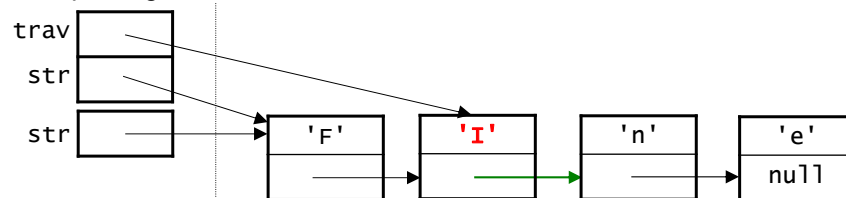
after updating trav:



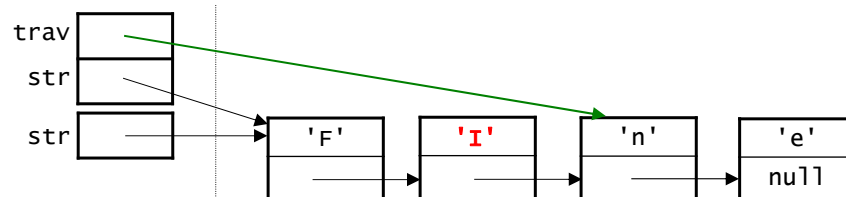
Tracing toUpperCase(): Second Iteration

```
while (trav != null) {  
    trav.ch = Character.toUpperCase(trav.ch);  
    trav = trav.next;  
}
```

after updating trav.ch:



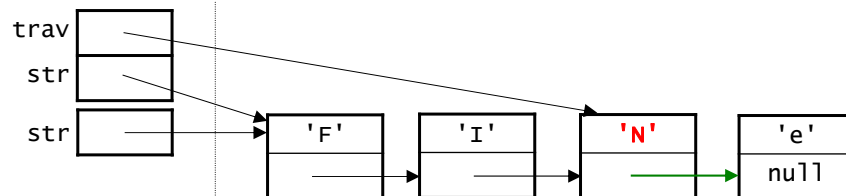
after updating trav:



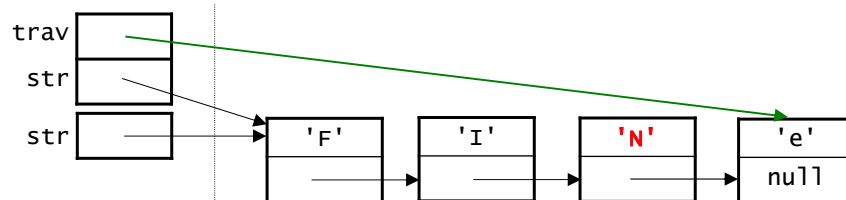
Tracing toUpperCase(): Third Iteration

```
while (trav != null) {  
    trav.ch = Character.toUpperCase(trav.ch);  
    trav = trav.next;  
}
```

after updating trav.ch:



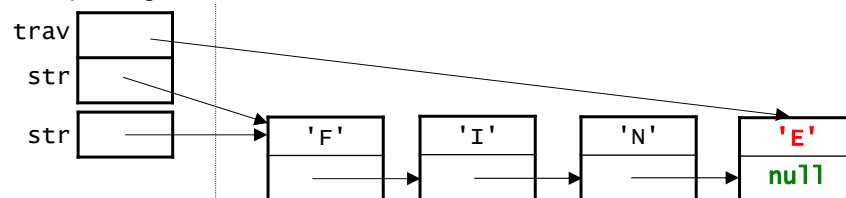
after updating trav:



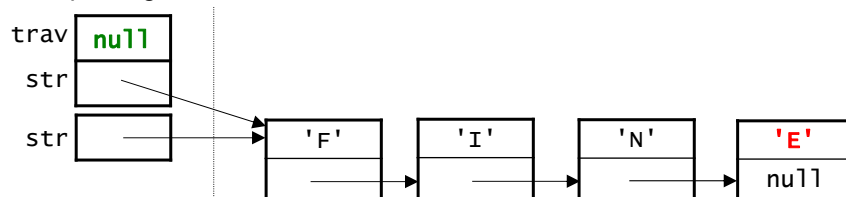
Tracing toUpperCase(): Fourth Iteration

```
while (trav != null) {  
    trav.ch = Character.toUpperCase(trav.ch);  
    trav = trav.next;  
}
```

after updating trav.ch:



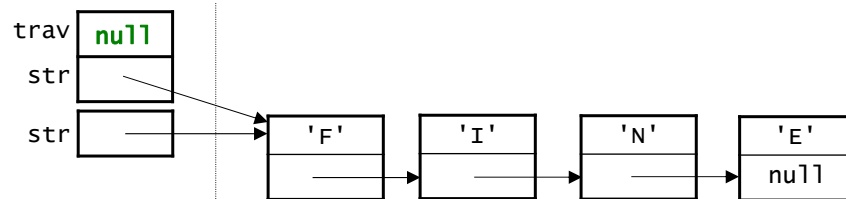
after updating trav:



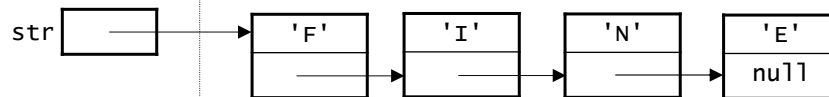
Tracing toupperCase(): Finishing Up

```
while (trav != null) {
    trav.ch = Character.toUpperCase(trav.ch);
    trav = trav.next;
}
```

results of the final iteration:

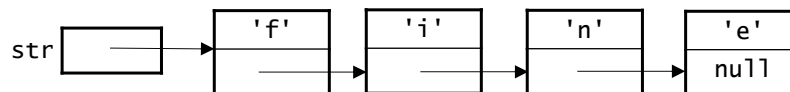


and now `trav == null`, so we end the loop and return:



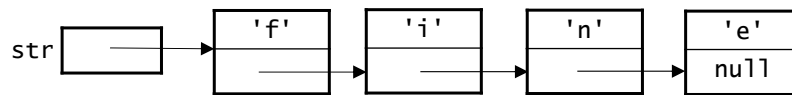
Getting the Node at Position i in a Linked List

- `getNode(str, i)` – should return a reference to the *i*th node in the linked list to which `str` refers



- Examples:
 - `getNode(str, 0)` should return a ref. to the 'f' node
 - `getNode(str, 3)` should return a ref. to the 'e' node
 - `getNode(str.next, 2)` should return a ref. to...?
- More generally, when $0 < i < \text{length of list}$, `getNode(str, i)` is equivalent to `getNode(str.next, i-1)`

Getting the Node at Position i in a Linked List



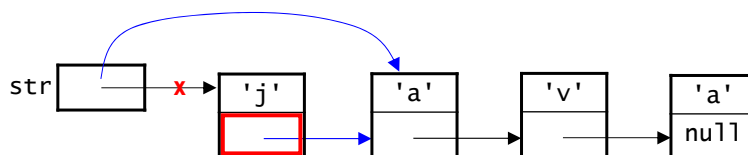
- Recursive approach to getNode(str, i):
 - if $i == 0$, return str (base case)
 - else call getNode(str.next, i-1) and return what it returns!
 - other base case?

- Here's the method:

```
private static StringNode getNode(StringNode str, int i) {  
    if (i < 0 || str == null) { // base case 1: no node i  
        return null;  
    } else if (i == 0) { // base case 2: just found  
        return str;  
    } else {  
        return getNode(str.next, i-1);  
    }  
}
```

Deleting the Item at Position i

- Special case: $i == 0$ (deleting the first item)
- Update our reference to the first node by doing:
 str = str.next;



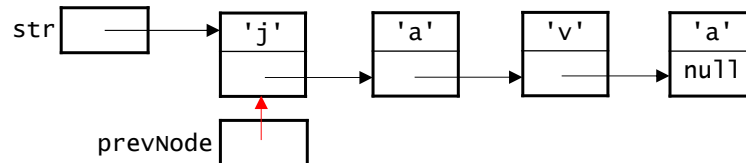
Deleting the Item at Position i (cont.)

- General case: $i > 0$

1. Obtain a reference to the *previous* node:

```
StringNode prevNode = getNode(i - 1);
```

(example for $i == 1$)



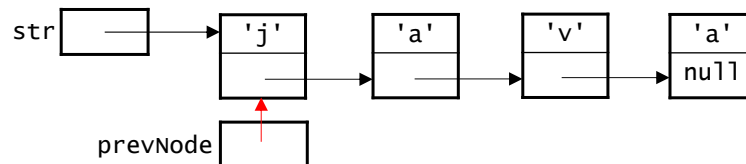
Deleting the Item at Position i (cont.)

- General case: $i > 0$

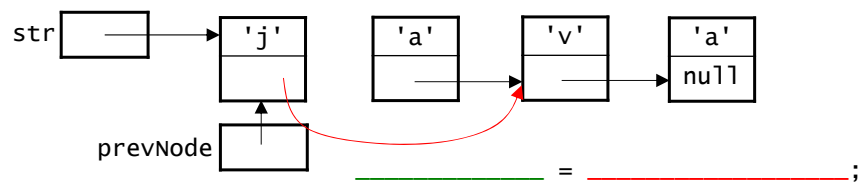
2. Update the references to remove the node

(example for $i == 1$)

before:



after:

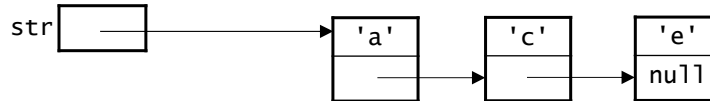


Inserting an Item at Position i

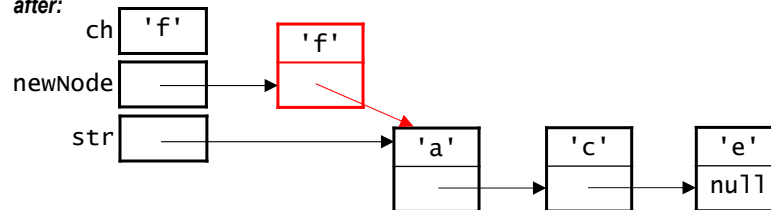
- Special case: $i == 0$ (insertion at the front of the list)
- Step 1: *Create the new node. Fill in the blanks!*

before:

ch 'f'



after:

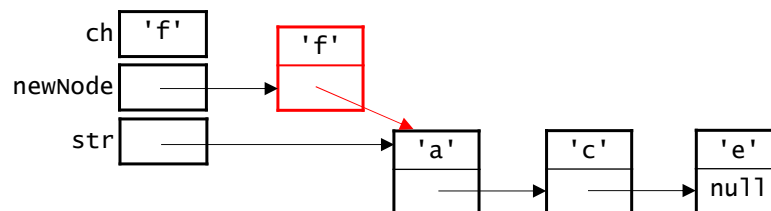


```
StringNode newNode = new StringNode(_____, _____);
```

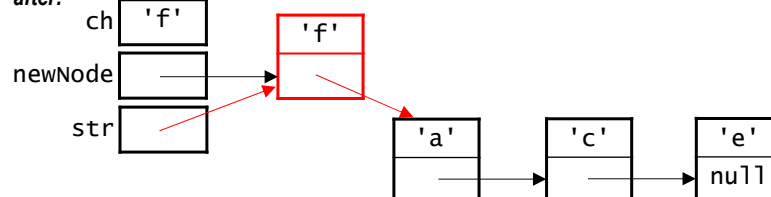
Inserting an Item at Position i (cont.)

- Special case: $i == 0$ (continued)
- Step 2: *Insert the new node. Write the assignment!*

before (result of previous slide):

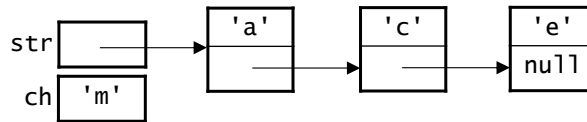


after:

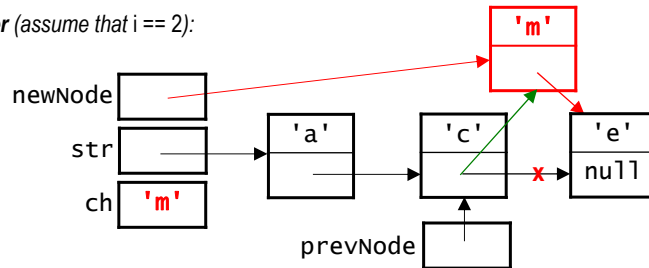


Inserting an Item at Position i (cont.)

- General case: $i > 0$ (insert *before* the item currently in posn i)
before:



after (assume that $i == 2$):



```

StringNode prevNode = getNode(i - 1);
StringNode newNode = new StringNode(ch, _____);
_____ // one more line
  
```

Returning a Reference to the First Node

- Both `deleteChar()` and `insertChar()` return a reference to the first node in the linked list. For example:

```

public static StringNode deleteChar(StringNode str, int i) {
    ...
    if (i == 0) {                // special case
        str = str.next;
    } else {                    // general case
        StringNode prevNode = getNode(str, i-1);
        if (prevNode != null && prevNode.next != null) {
            prevNode.next = prevNode.next.next;
            ...
        }
        return str;
    }
}
  
```

- Clients should call them as part of an assignment:

```

s1 = StringNode.deleteChar(s1, 0);
s2 = StringNode.insertChar(s2, 0, 'h');
  
```

- If the first node changes, the client's variable will be updated to point to the new first node.

Creating a Copy of a Linked List

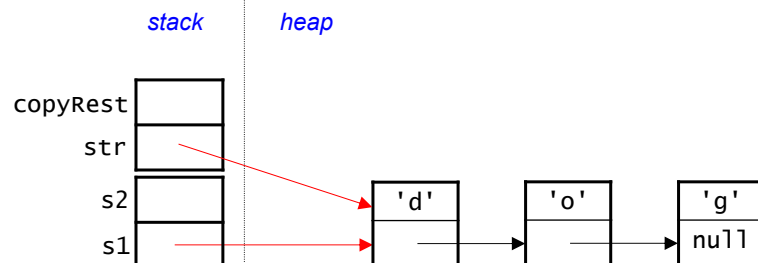
- `copy(str)` – create a copy of *the entire list* to which `str` refers
- Recursive approach:
 - base case: if `str` is empty, return `null`
 - else: – make a recursive call to copy the rest of the linked list
 - create and return a copy of the first node,
with its next field pointing to the copy of the rest

```
public static StringNode copy(StringNode str) {  
    if (str == null) {          // base case  
        return null;  
    }  
    // make a recursive call to copy the rest of the list  
    StringNode copyRest = copy(str.next);  
  
    // create and return a copy of the first node,  
    // with its next field pointing to the copy of the rest  
    return new StringNode(str.ch, copyRest);  
}
```

Tracing `copy()`: the initial call

- From a client: `StringNode s2 = StringNode.copy(s1);`

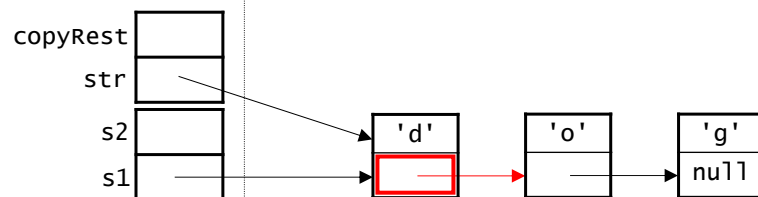
```
public static StringNode copy(StringNode str) {  
    if (str == null) {  
        return null;  
    }  
    StringNode copyRest = copy(str.next);  
    return new StringNode(str.ch, copyRest);  
}
```



Tracing copy(): the initial call

- From a client: `StringNode s2 = StringNode.copy(s1);`

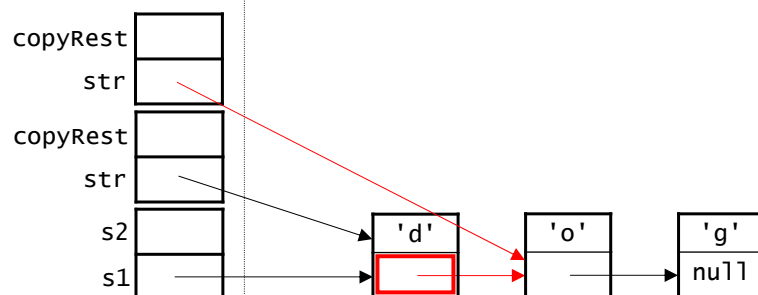
```
public static StringNode copy(StringNode str) {  
    if (str == null) {  
        return null;  
    }  
    StringNode copyRest = copy(str.next);  
    return new StringNode(str.ch, copyRest);  
}
```



Tracing copy(): the initial call

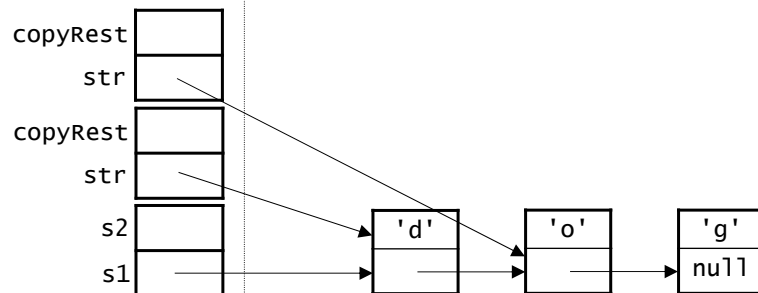
- From a client: `StringNode s2 = StringNode.copy(s1);`

```
public static StringNode copy(StringNode str) {  
    if (str == null) {  
        return null;  
    }  
    StringNode copyRest = copy(str.next);  
    return new StringNode(str.ch, copyRest);  
}
```



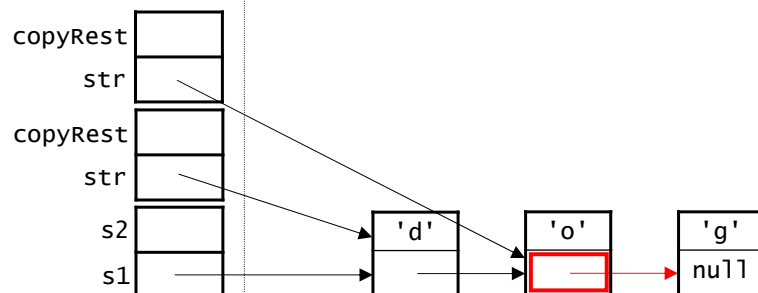
Tracing copy(): the recursive calls

```
public static StringNode copy(...) {
    if (str == null) {
        return null;
    }
    StringNode copyRest = copy(str.next);
    return new StringNode(str.ch,
                          copyRest);
}
```



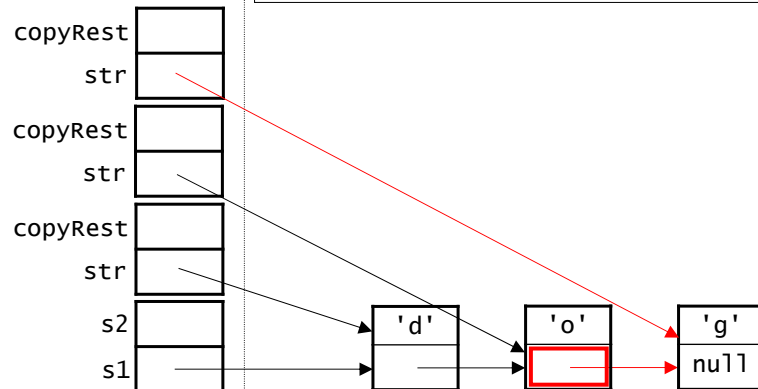
Tracing copy(): the recursive calls

```
public static StringNode copy(...) {
    if (str == null) {
        return null;
    }
    StringNode copyRest = copy(str.next);
    return new StringNode(str.ch,
                          copyRest);
}
```



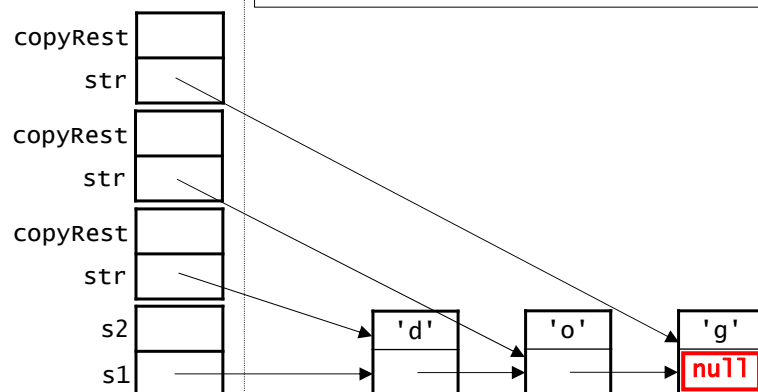
Tracing copy(): the recursive calls

```
public static StringNode copy(...) {
    if (str == null) {
        return null;
    }
    StringNode copyRest = copy(str.next);
    return new StringNode(str.ch,
                          copyRest);
}
```

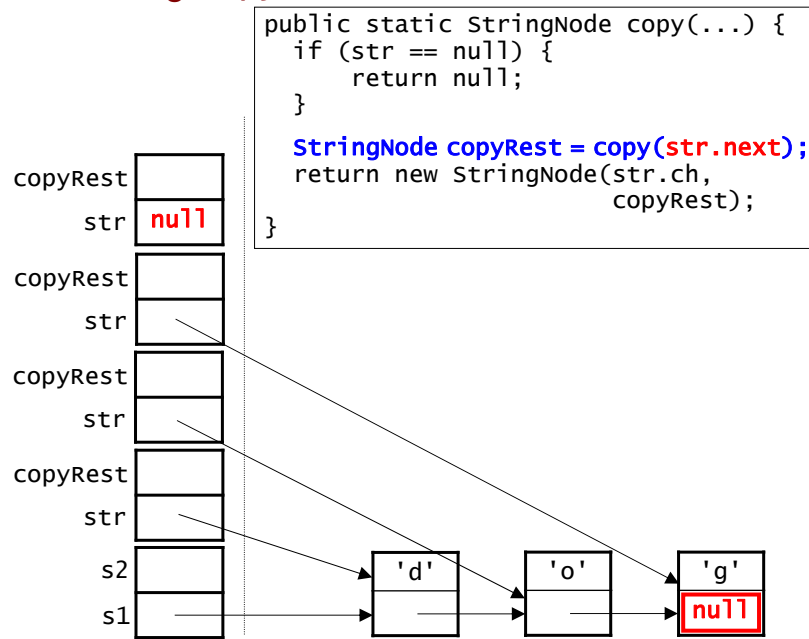


Tracing copy(): the recursive calls

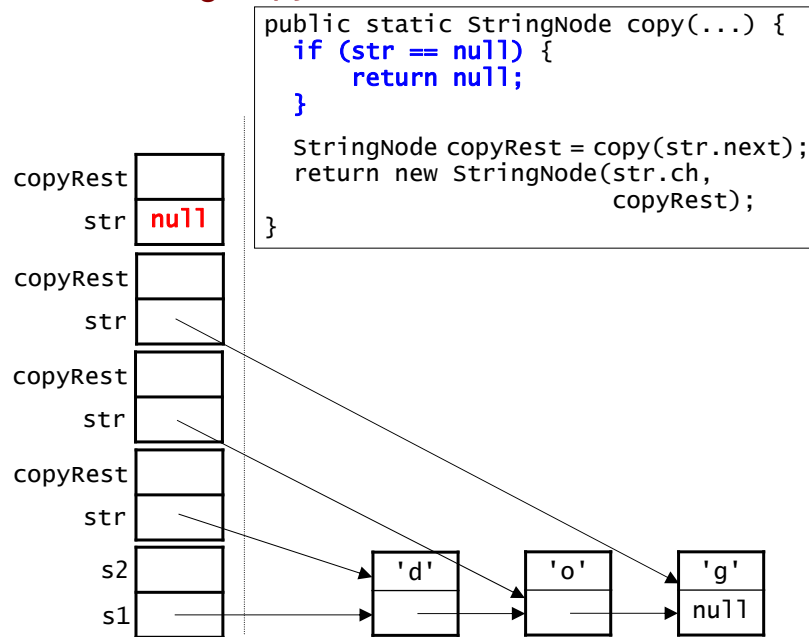
```
public static StringNode copy(...) {
    if (str == null) {
        return null;
    }
    StringNode copyRest = copy(str.next);
    return new StringNode(str.ch,
                          copyRest);
}
```



Tracing copy(): the recursive calls

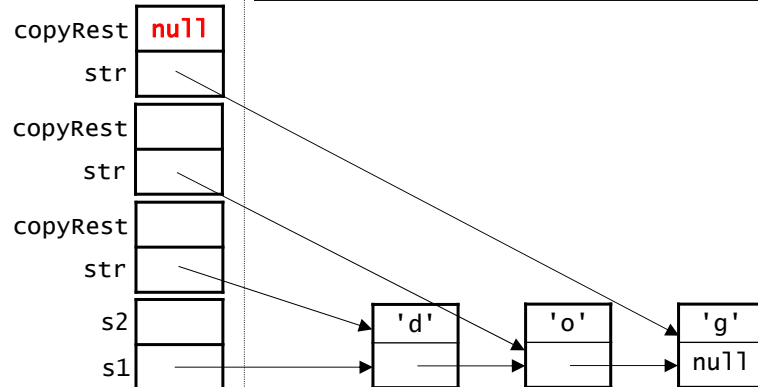


Tracing copy(): the base case



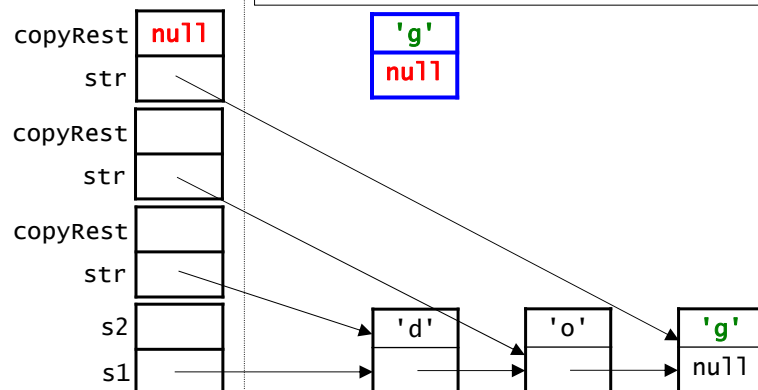
Tracing copy(): returning from the base case

```
public static StringNode copy(...) {
    if (str == null) {
        return null;
    }
    StringNode copyRest = copy(str.next);
    return new StringNode(str.ch,
        copyRest);
}
```



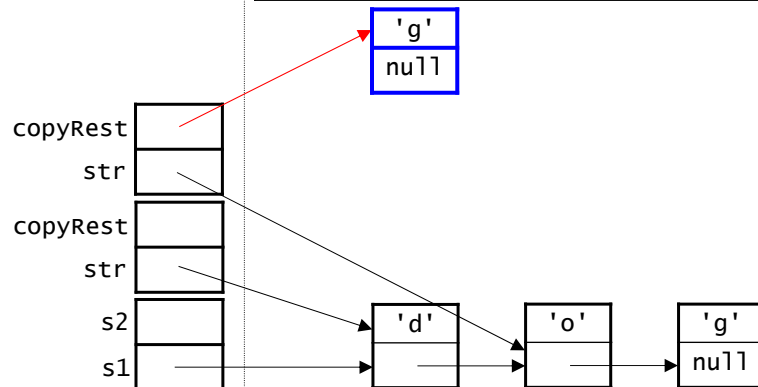
Tracing copy(): returning from the base case

```
public static StringNode copy(...) {
    if (str == null) {
        return null;
    }
    StringNode copyRest = copy(str.next);
    return new StringNode(str.ch,
        copyRest);
}
```



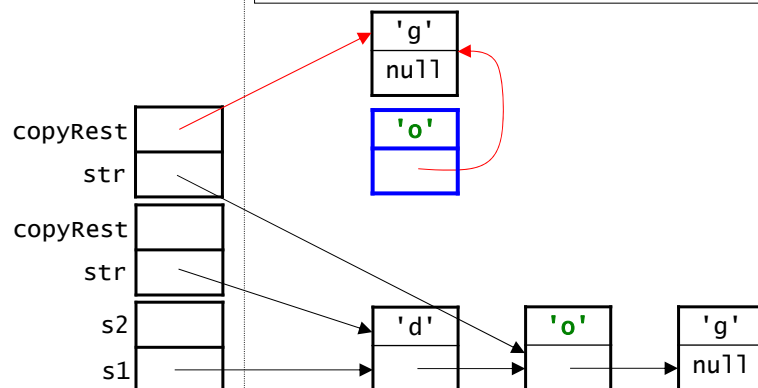
Tracing copy(): returning from the base case

```
public static StringNode copy(...) {
    if (str == null) {
        return null;
    }
    StringNode copyRest = copy(str.next);
    return new StringNode(str.ch,
                          copyRest);
}
```



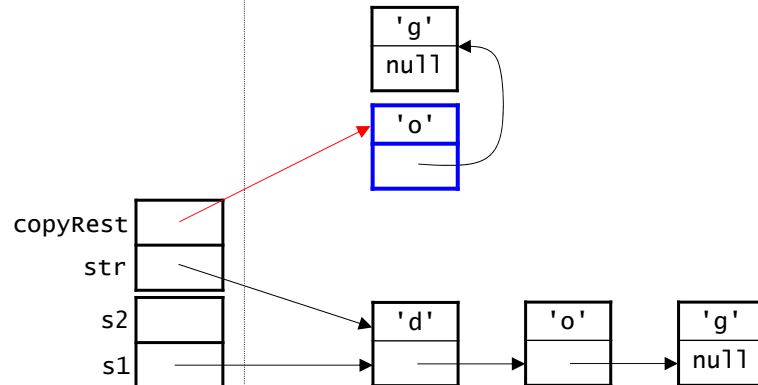
Tracing copy(): returning from the base case

```
public static StringNode copy(...) {
    if (str == null) {
        return null;
    }
    StringNode copyRest = copy(str.next);
    return new StringNode(str.ch,
                          copyRest);
}
```



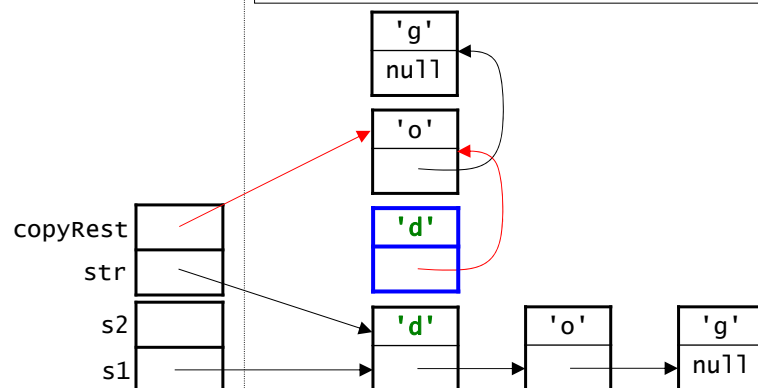
Tracing copy(): returning from the base case

```
public static StringNode copy(...) {
    if (str == null) {
        return null;
    }
    StringNode copyRest = copy(str.next);
    return new StringNode(str.ch,
        copyRest);
}
```



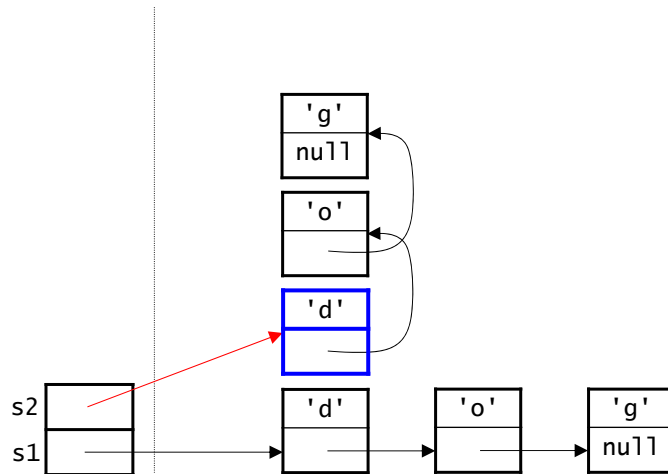
Tracing copy(): returning from the base case

```
public static StringNode copy(...) {
    if (str == null) {
        return null;
    }
    StringNode copyRest = copy(str.next);
    return new StringNode(str.ch,
        copyRest);
}
```



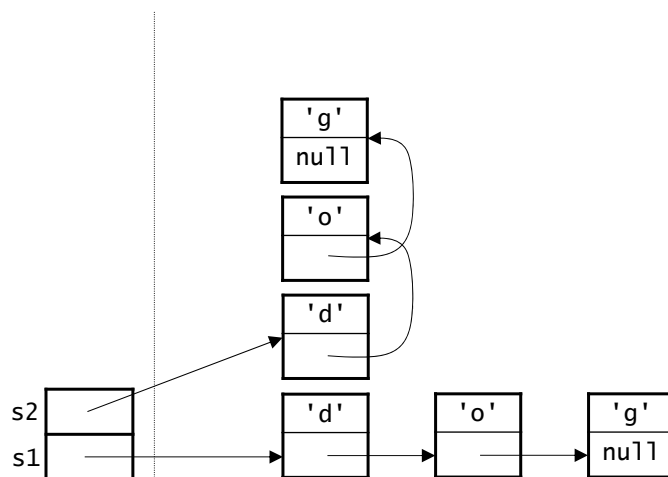
Tracing copy(): returning from the base case

- From a client: `StringNode s2 = StringNode.copy(s1);`



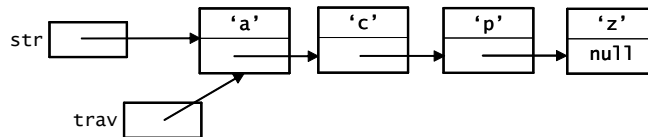
Tracing copy(): Final Result

- `s2` now holds a reference to a linked list that is a copy of the linked list to which `s1` holds a reference.



Using a "Trailing Reference" During Traversal

- When traversing a linked list, one trav may not be enough.
- Ex: insert `ch = 'n'` at the right place in this *sorted* linked list:



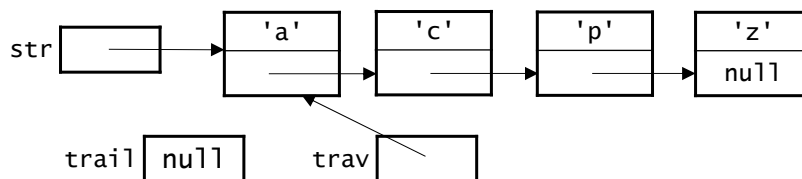
- Traverse the list to find the right position:

```
StringNode trav = str;
while (trav != null && trav.ch < ch) {
    trav = trav.next;
}
```
- When we exit the loop, where will `trav` point? Can we insert 'n'?
- The following changed version doesn't work either. Why not?

```
while (trav != null && trav.next.ch < ch) {
    trav = trav.next;
}
```

Using a "Trailing Reference" (cont.)

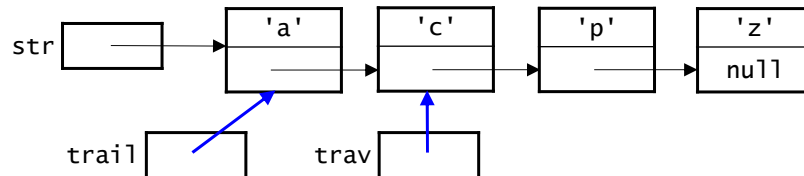
- To get around the problem seen on the previous page, we traverse the list using two different references:
 - `trav`, which we use as before
 - `trail`, which stays one node behind `trav`



```
StringNode trav = str;
StringNode trail = null;
while (trav != null && trav.ch < ch) {
    trail = trav;
    trav = trav.next;
}
// if trail == null, insert at the front of the list
// else insert after the node to which trail refers
```

Using a "Trailing Reference" (cont.)

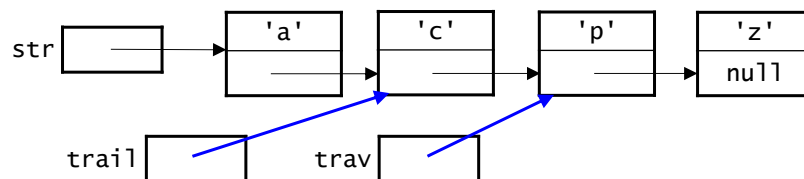
- To get around the problem seen on the previous page, we traverse the list using two different references:
 - trav, which we use as before
 - trail, which stays one node behind trav



```
StringNode trav = str;
StringNode trail = null;
while (trav != null && trav.ch < ch) {
    trail = trav;
    trav = trav.next;
}
// if trail == null, insert at the front of the list
// else insert after the node to which trail refers
```

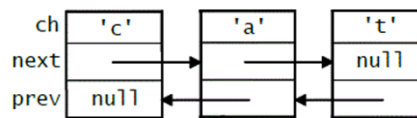
Using a "Trailing Reference" (cont.)

- To get around the problem seen on the previous page, we traverse the list using two different references:
 - trav, which we use as before
 - trail, which stays one node behind trav



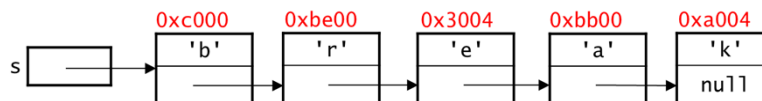
```
StringNode trav = str;
StringNode trail = null;
while (trav != null && trav.ch < ch) {
    trail = trav;
    trav = trav.next;
}
// if trail == null, insert at the front of the list
// else insert after the node to which trail refers
```

Doubly Linked Lists



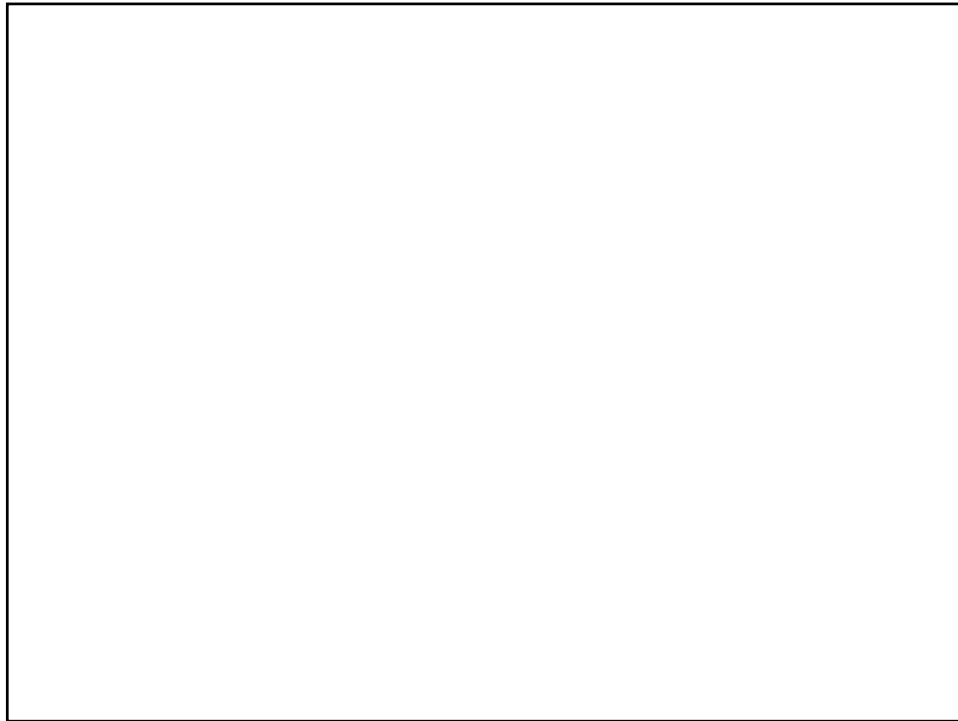
- In a doubly linked list, every node stores *two* references:
 - next, which works the same as before
 - prev, which holds a reference to the previous node
 - in the first node, prev has a value of null
- The prev references allow us to "back up" as needed.
 - remove the need for a trailing reference during traversal!
- Insertion and deletion must update both types of references.

Find the address and value of `s.next.next.ch`

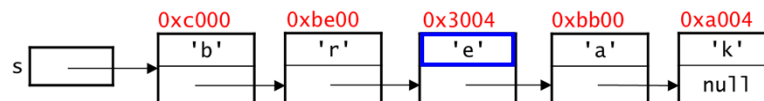


Extra practice!

- | | <u>address</u> | <u>value</u> |
|----|----------------|--------------|
| A. | 0xbe00 | 'r' |
| B. | 0x3004 | 'e' |
| C. | 0xbb00 | 'a' |
| D. | none of these | |



Find the address and value of `s.next.next.ch`



- `s.next` is the next field in the node to which `s` refers
 - it holds a reference to the 'r' node
- thus, `s.next.next` is the next field in the 'r' node
 - it holds a reference to the 'e' node
- thus, `s.next.next.ch` is the ch field in the 'e' node
 - it holds the 'e' !

	<u>address</u>	<u>value</u>
A.	0xbe00	'r'
B.	0x3004	'e'
C.	0xbb00	'a'
D.	none of these	

Lists, Stacks, and Queues

Computer Science E-22
Harvard University

David G. Sullivan, Ph.D.

Representing a Sequence: Arrays vs. Linked Lists

- Sequence – an ordered collection of items (position matters)
 - we will look at several types: lists, stacks, and queues
- Can represent any sequence using an array *or* a linked list

	<i>array</i>	<i>linked list</i>
representation in memory	elements occupy consecutive memory locations	nodes can be at arbitrary locations in memory; the links connect the nodes together
advantages	• provide random access (access to any item in constant time) • no extra memory needed for links	• can grow to an arbitrary length • allocate nodes as needed • inserting or deleting does <i>not</i> require shifting items
disadvantages	• have to preallocate the memory needed for the maximum sequence size • inserting or deleting can require shifting items	• no random access (may need to traverse the list) • need extra memory for links

Abstract Data Types

- An *abstract data type* (ADT) is a model of a data structure that specifies:
 - the characteristics of the collection of data
 - the operations that can be performed on the collection
- It's *abstract* because it doesn't specify *how* the ADT will be implemented.
- A given ADT can have multiple implementations.

The List ADT

- A list is a sequence in which items can be accessed, inserted, and removed *at any position in the sequence*.
- The operations supported by our List ADT:
 - `getItem(i)`: get the item at position *i*
 - `addItem(item, i)`: add the specified item at position *i*
 - `removeItem(i)`: remove the item at position *i*
 - `length()`: get the number of items in the list
 - `isFull()`: test if the list already has the maximum number of items
- Note that we *don't* specify *how* the list will be implemented.

Specifying an ADT Using an Interface

- In Java, we can use an *interface* to specify an ADT:

```
public interface List {  
    Object getItem(int i);  
    boolean addItem(Object item, int i);  
    Object removeItem(int i);  
    int length();  
    boolean isFull();  
}
```

- An interface specifies a set of methods.
 - includes only their headers
 - does *not* typically include the full method definitions
- Like a class, it must go in a file with an appropriate name.
 - in this case: `List.java`
- Methods specified in an interface *must* be public, so we don't need the keyword `public` in the headers.

Implementing an ADT Using a Class

- To implement an ADT, we define a class.
- We specify the corresponding interface in the class header:

```
public class ArrayList implements List {  
    ...
```

 - tells the compiler that the class will define *all* of the methods in the interface
 - if the class doesn't define them, it won't compile
- We'll look at two implementations of the `List` interface:
 - `ArrayList` – uses an array to store the items
 - `LinkedList` – uses a linked list to store the items

Recall: Polymorphism

- An object can be used wherever an object of one of its superclasses is called for.
- For example:

```
Animal a = new Dog();  
Animal[] zoo = new Animal[100];  
zoo[0] = new Ant();  
zoo[1] = new Cat();  
...
```

Another Example of Polymorphism

- An interface can be used as the type of a variable:
`List myList;`
- We can then assign an object of any class that implements the interface:
`List l1 = new ArrayList(20);`
`List l2 = new LLList();`
- This allows us write code that works with *any* implementation of an ADT:

```
public static void processList(List vals) {  
    for (int i = 0; i < vals.length(); i++) {  
        ...  
    }  
}
```

- `vals` can be an object of *any* class that implements `List`
- regardless of which class `vals` is from, we know it has all of the methods in the `List` interface

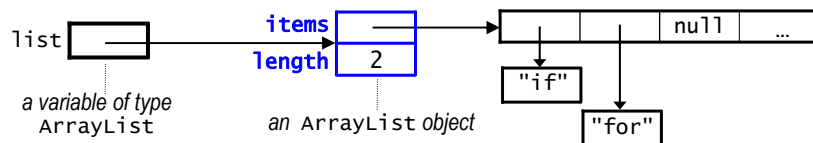
Implementing a List Using an Array

```
public class ArrayList implements List {
    private Object[] items;
    private int length;

    public ArrayList(int maxSize) {
        // code to check for invalid maxSize goes here...
        this.items = new Object[maxSize];
        this.length = 0;
    }

    public int length() {
        return this.length;
    }

    public boolean isFull() {
        return (this.length == this.items.length);
    }
    ...
}
```



Recall: The Implicit Parameter

```
public class ArrayList implements List {
    private Object[] items;
    private int length;

    public ArrayList(int maxSize) {
        this.items = new Object[maxSize];
        this.length = 0;
    }

    public int length() {
        return this.length;
    }

    public boolean isFull() {
        return (this.length == this.items.length);
    }
    ...
}
```

- All non-static methods have an implicit parameter (`this`) that refers to the called object.
- In most cases, we're allowed to omit it!
 - we'll do so in the remaining notes

Omitting The Implicit Parameter

```
public class ArrayList implements List {
    private Object[] items;
    private int length;

    public ArrayList(int maxSize) {
        items = new Object[maxSize];
        length = 0;
    }

    public int length() {
        return length;
    }

    public boolean isFull() {
        return (length == items.length);
    }
    ...
}
```

- In a non-static method, if we use a variable that
 - isn't declared in the method
 - has the name of one of the fieldsJava assumes that we're using the field.

Adding an Item to an ArrayList

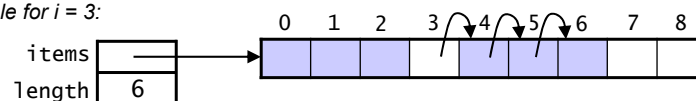
- Adding at position i (shifting items $i, i+1, \dots$ to the right by one):

```
public boolean addItem(Object item, int i) {
    if (item == null || i < 0 || i > length) {
        throw new IllegalArgumentException();
    } else if (isFull()) {
        return false;
    }

    // make room for the new item
    for (int j = length - 1; j >= i; j--) {
        items[j + 1] = items[j];
    }

    items[i] = item;
    length++;
    return true;
}
```

example for $i = 3$:



Adding an Item to an ArrayList

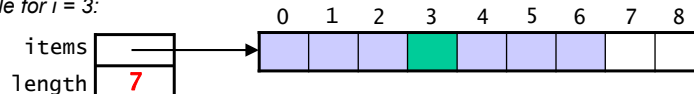
- Adding at position i (shifting items $i, i+1, \dots$ to the right by one):

```
public boolean addItem(Object item, int i) {
    if (item == null || i < 0 || i > length) {
        throw new IllegalArgumentException();
    } else if (isFull()) {
        return false;
    }

    // make room for the new item
    for (int j = length - 1; j >= i; j--) {
        items[j + 1] = items[j];
    }

    items[i] = item;
    length++;
    return true;
}
```

example for $i = 3$:

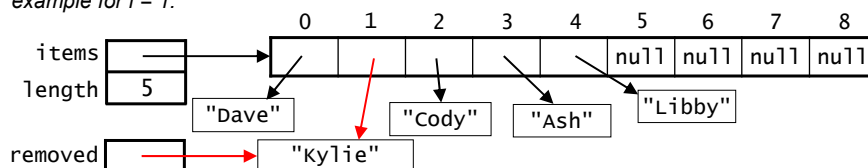


Removing an Item from an ArrayList

- Removing item i (shifting items $i+1, i+2, \dots$ to the left by one):

```
public Object removeItem(int i) {
    if (i < 0 || i >= length) {
        throw new IndexOutOfBoundsException();
    }
    Object removed = items[i];
    // shift items after items[i] to the left
    for (int j = i; j < length - 1; j++) {
        items[j] = items[j + 1];
    }
    items[length - 1] = null;
    length--;
    return removed;
}
```

example for $i = 1$:



Getting an Item from an ArrayList

- Getting item *i* (without removing it):

```
public Object getItem(int i) {  
    if (i < 0 || i >= length) {  
        throw new IndexOutOfBoundsException();  
    }  
    return items[i];  
}
```

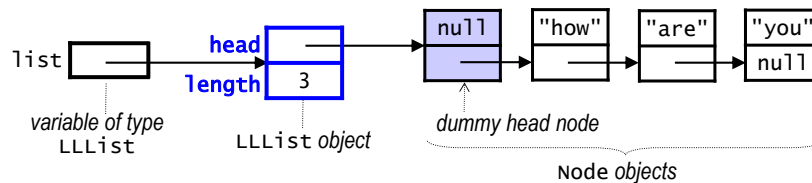
toString() Method for the ArrayList Class

```
public String toString() {  
    String str = "{";  
    if (length > 0) {  
        for (int i = 0; i < length - 1; i++) {  
            str = str + items[i] + ", ";  
        }  
        str = str + items[length - 1];  
    }  
    str = str + "}";  
    return str;  
}
```

- Produces a string of the following form:
 {items[0], items[1], ... }
- Why is the last item added outside the loop?
- Why do we need the if statement?

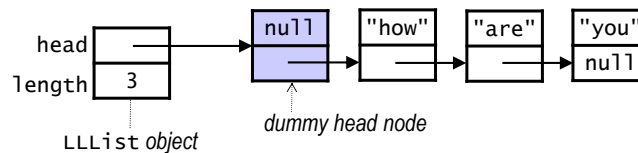
Implementing a List Using a Linked List

```
public class LLList implements List {
    private Node head;
    private int length;
    ...
}
```

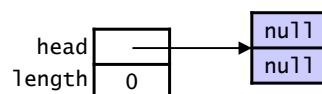


- Differences from the linked lists we used for strings:
 - we "embed" the linked list inside another class
 - users of our `LLLlist` class won't actually touch the nodes
 - we use non-static methods instead of static ones
`myList.length()` instead of `length(myList)`
 - we use a special *dummy head node* as the first node

Using a Dummy Head Node



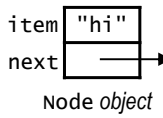
- The dummy head node is always at the front of the linked list.
 - like the other nodes in the linked list, it's of type `Node`
 - it does *not* store an item
 - it does *not* count towards the length of the list
- Using it allows us to avoid special cases when adding and removing nodes from the linked list.
- An empty `LLLlist` still has a dummy head node:



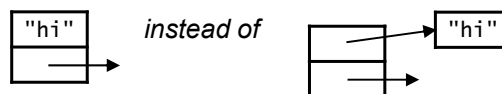
An Inner Class for the Nodes

```
public class LLList implements List {
    private class Node {
        private Object item;
        private Node next;
    }
    private Node(Object i, Node n) {
        item = i;
        next = n;
    }
    ...
}
```

private since only LLList will use it



- We make Node an *inner class*, defining it within LLList.
 - allows the LLList methods to directly access Node's private fields, while restricting access from outside LLList
 - the compiler creates this class file: LLList\$Node.class
- For simplicity, our diagrams may show the items inside the nodes.



Other Details of Our LLList Class

```
public class LLList implements List {
    private class Node {
        // see previous slide
    }
    private Node head;
    private int length;

    public LLList() {
        head = new Node(null, null);
        length = 0;
    }

    public boolean isFull() {
        return false;
    }
    ...
}
```

- Unlike ArrayList, there's no need to preallocate space for the items. The constructor simply creates the dummy head node.
- The linked list can grow indefinitely, so the list is never full!

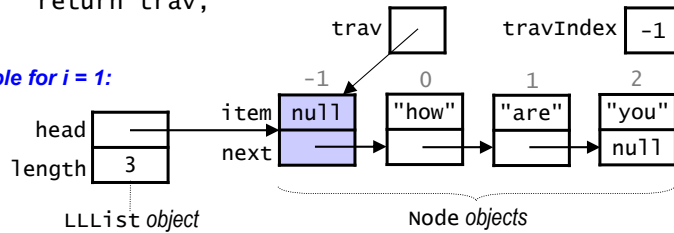
Getting a Node

- Private helper method for getting node i
 - to get the dummy head node, use $i = -1$

```
private Node getNode(int i) {
    // private method, so we assume i is valid!

    Node trav = _____;
    int travIndex = -1;
    while ( _____ ) {
        travIndex++;
        _____;
    }
    return trav;
}
```

example for $i = 1$:

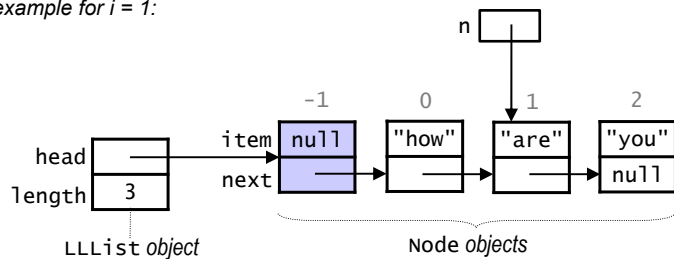


Getting an Item

```
public Object getItem(int i) {
    if (i < 0 || i >= length) {
        throw new IndexOutOfBoundsException();
    }

    Node n = getNode(i);
    return _____;
}
```

example for $i = 1$:

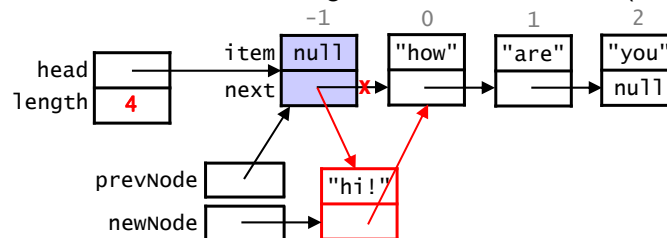


Adding an Item to an LLList

```
public boolean addItem(Object item, int i) {
    if (item == null || i < 0 || i > length) {
        throw new IllegalArgumentException();
    }
    Node newNode = new Node(item, null);
    Node prevNode = getNode(i - 1);
    newNode.next = prevNode.next;
    prevNode.next = newNode;

    length++;
    return true;
}
```

- This works even when adding at the front of the list ($i = 0$):



addItem() Without a Dummy Head Node

```
public boolean addItem(Object item, int i) {
    if (item == null || i < 0 || i > length) {
        throw new IllegalArgumentException();
    }
    Node newNode = new Node(item, null);

    if (i == 0) { // case 1: add to front
        newNode.next = head;
        head = newNode;
    } else { // case 2: i > 0
        Node prevNode = getNode(i - 1);
        newNode.next = prevNode.next;
        prevNode.next = newNode;
    }

    length++;
    return true;
}
```

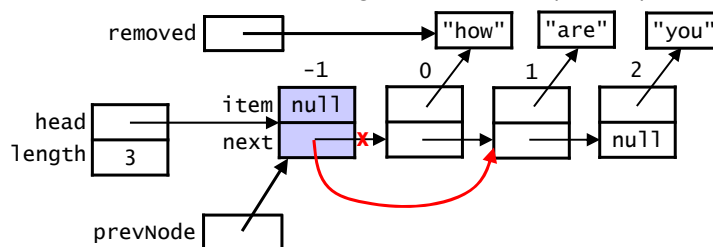
(the gray code shows what we would need to add if we didn't have a dummy head node)

Removing an Item from an LLList

```
public Object removeItem(int i) {
    if (i < 0 || i >= length) {
        throw new IndexOutOfBoundsException();
    }
    Node prevNode = getNode(i - 1);
    Object removed = prevNode.next.item;
    // what line goes here?

    length--;
    return removed;
}
```

- This works even when removing the first item ($i = 0$):



toString() Method for the LLList Class

```
public String toString() {
    String str = "{";

    // what should go here?
```

```
    str = str + "}";
    return str;
}
```

Efficiency of the List ADT Implementations

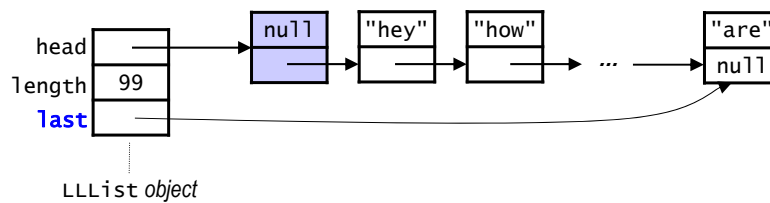
n = number of items in the list

	ArrayList	LLList
getItem()	only one case:	best: worst: average:
addItem()	best: worst: average:	best: worst: average:

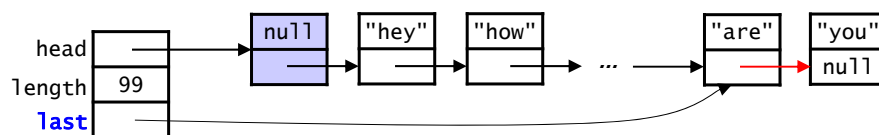
Example of Using a Reference to the Last Node

```
mylist.addItem("you", 99)
```

- before the call is made:



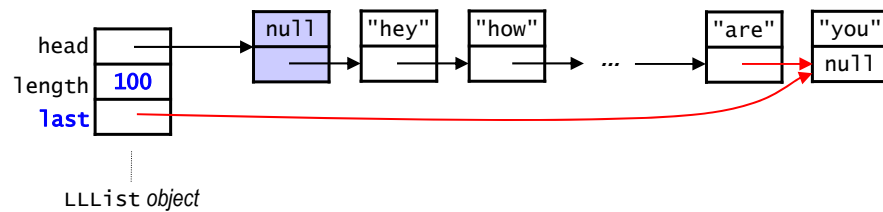
- use **last** to add the new item's node to the end of the linked list:



Example of Using a Reference to the Last Node (cont.)

```
mylist.addItem("you", 99)
```

- after the call is made:



Efficiency of the List ADT Implementations (cont.)

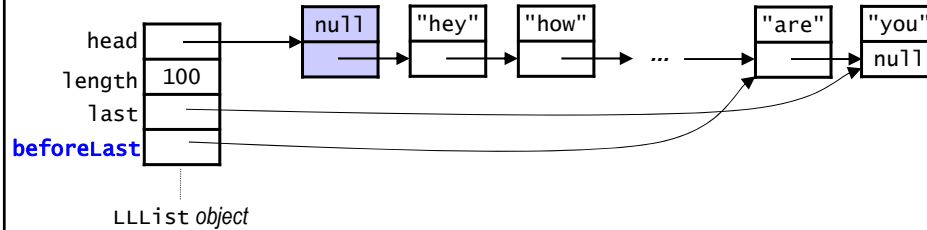
n = number of items in the list

	ArrayList	LLLlist
removeItem()	best: worst: average:	best: worst: average:
space efficiency		

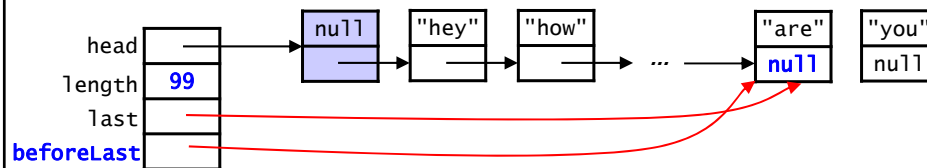
A Reference to the Second-to-Last Node Doesn't Help

```
mylist.removeItem(99)
```

- before the call is made:



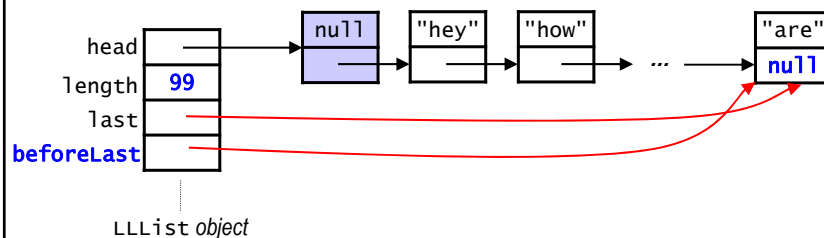
- we can use beforeLast to remove the last node and update last:



A Reference to the Second-to-Last Node Doesn't Help

```
mylist.removeItem(99)
```

- but in order to update beforeLast, we need to walk down the linked list!



Counting the Number of Occurrences of an Item

```
public class MyClass {
    public static int numOccur(List l, Object item) {
        int numOccur = 0;
        for (int i = 0; i < l.length(); i++) {
            Object itemAt = l.getItem(i);
            if (itemAt.equals(item)) {
                numOccur++;
            }
        }
        return numOccur;
    } ...
}
```

- This method works fine if we pass in an ArrayList object.
 - time efficiency (as a function of the length, n) = ?
- However, it's *not* efficient if we pass in an LLList.
 - each call to `getItem()` calls `getNode()`
 - to access item 0, `getNode()` accesses 2 nodes (dummy + node 0)
 - to access item 1, `getNode()` accesses 3 nodes
 - to access item i , `getNode()` accesses $i+2$ nodes
 - $2 + 3 + \dots + (n+1) = ?$

Solution: Provide an Iterator

```
public class MyClass {
    public static int numOccur(List l, Object item) {
        int numOccur = 0;
        ListIterator iter = l.iterator();
        while (iter.hasNext()) {
            Object itemAt = iter.next();
            if (itemAt.equals(item)) {
                numOccur++;
            }
        }
        return numOccur;
    } ...
}
```

- We add an `iterator()` method to the List interface.
 - it returns a separate *iterator object* that can efficiently iterate over the items in the list
- The iterator has two key methods:
 - `hasNext()`: tells us if there are items we haven't seen yet
 - `next()`: returns the next item *and* advances the iterator

An Interface for List Iterators

- Here again, the interface only includes the method headers:

```
public interface ListIterator { // in ListIterator.java
    boolean hasNext();
    Object next();
}
```
- We can then implement this interface for each type of list:
 - LLListIterator for an iterator that works with LLLists
 - ArrayListIterator for an iterator for ArrayLists
- We use the interfaces when declaring variables in client code:

```
public class MyClass {
    public static int numOccur(List l, Object item) {
        int numOccur = 0;
        ListIterator iter = l.iterator();
        ...
    }
}
```

 - doing so allows the code to work for any type of list!

Using an Inner Class for the Iterator

```
public class LLList {
    private Node head;
    private int length;

    private class LLListIterator implements ListIterator {
        private Node nextNode; // points to node with the next item
        public LLListIterator() {
            nextNode = head.next; // skip over dummy head node
        }
        ...
    }

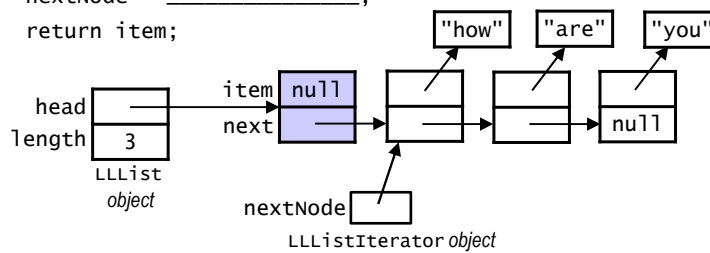
    public ListIterator iterator() {
        return new LLListIterator();
    }
    ...
}
```

- Using an inner class gives the iterator access to the list's internals.
- The iterator() method is an LLList method.
 - it creates an instance of the inner class and returns it
 - its return type is the interface type
 - so it will work in the context of client code

Full LLListIterator Implementation

```
private class LLListIterator implements ListIterator {
    private Node nextNode;    // points to node with the next item
    public LLListIterator() {
        nextNode = head.next; // skip over the dummy head node
    }
    public boolean hasNext() {
        return (nextNode != null);
    }
    public Object next() {
        // throw an exception if nextNode is null

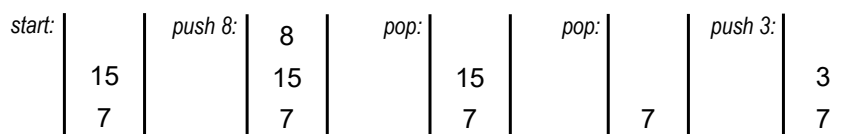
        Object item = _____;
        nextNode = _____;
        return item;
    }
}
```



Stack ADT



- A stack is a sequence in which:
 - items can be added and removed only at one end (the *top*)
 - you can only access the item that is currently at the top
- Operations:
 - push: add an item to the top of the stack
 - pop: remove the item at the top of the stack
 - peek: get the item at the top of the stack, but don't remove it
 - isEmpty: test if the stack is empty
 - isFull: test if the stack is full
- Example: a stack of integers



A Stack Interface: First Version

```
public interface Stack {
    boolean push(Object item);
    Object pop();
    Object peek();
    boolean isEmpty();
    boolean isFull();
}
```

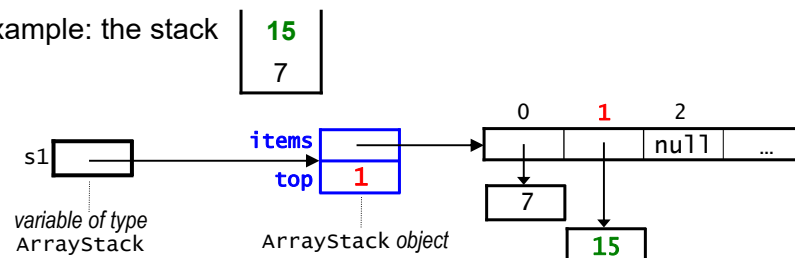
- `push()` returns `false` if the stack is full, and `true` otherwise.
- `pop()` and `peek()` take no arguments, because we know that we always access the item at the top of the stack.
 - return `null` if the stack is empty.
- The interface provides no way to access/insert/delete an item at an arbitrary position.
 - encapsulation allows us to ensure that our stacks are only manipulated in appropriate ways

Implementing a Stack Using an Array: First Version

```
public class ArrayStack implements Stack {
    private Object[] items;
    private int top;    // index of the top item

    public ArrayStack(int maxSize) {
        // code to check for invalid maxSize goes here...
        items = new Object[maxSize];
        top = -1;
    }
    ...
}
```

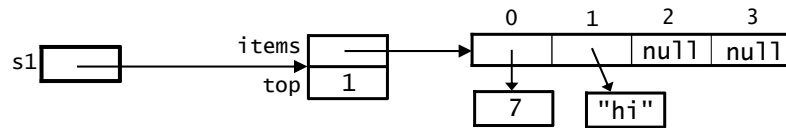
- Example: the stack



- Items are added from left to right (top item = the rightmost one).
 - `push()` and `pop()` won't require any shifting!

Collection Classes and Data Types

```
public class ArrayStack implements Stack {
    private Object[] items;
    private int top;    // index of the top item
    ...
}
```



- So far, our collections have allowed us to add objects of any type.

```
ArrayStack s1 = new ArrayStack(4);
s1.push(7);    // 7 is turned into an Integer object for 7
s1.push("hi");
String item = s1.pop();    // won't compile
String item = (String)s1.pop();    // need a type cast
```

- We'd like to be able to limit a given collection to one type.

```
ArrayStack<String> s2 = new ArrayStack<String>(10);
s2.push(7);    // won't compile
s2.push("hello");
String item = s2.pop();    // no cast needed!
```

Limiting a Stack to Objects of a Given Type

- We can do this by using a *generic* interface and class.

- Here's a generic version of our Stack interface:

```
public interface Stack<T> {
    boolean push(T item);
    T pop();
    T peek();
    boolean isEmpty();
    boolean isFull();
}
```

- It includes a *type variable* **T** in its header and body.
 - used as a placeholder for the actual type of the items

A Generic ArrayStack Class

```
public class ArrayStack<T> implements Stack<T> {
    private T[] items;
    private int top;    // index of the top item
    ...
    public boolean push(T item) {
        ...
    }
    ...
}
```

- Once again, a type variable **T** is used as a placeholder for the actual type of the items.
- When we create an ArrayStack, we specify the type of items that we intend to store in the stack:

```
ArrayStack<String> s1 = new ArrayStack<String>(10);
ArrayStack<Integer> s2 = new ArrayStack<Integer>(25);
```

- We can still allow for a mixed-type collection:

```
ArrayStack<Object> s3 = new ArrayStack<Object>(20);
```

Using a Generic Class

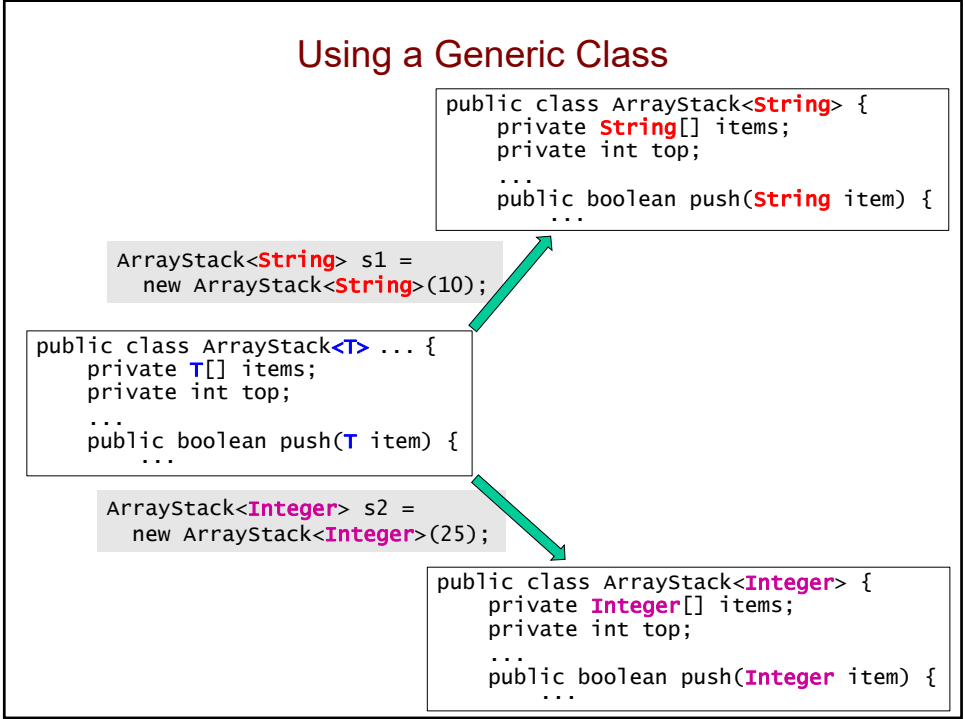
```
public class ArrayStack<String> {
    private String[] items;
    private int top;
    ...
    public boolean push(String item) {
        ...
    }
}
```

```
ArrayStack<String> s1 =
    new ArrayStack<String>(10);
```

```
public class ArrayStack<T> ... {
    private T[] items;
    private int top;
    ...
    public boolean push(T item) {
        ...
    }
}
```

```
ArrayStack<Integer> s2 =
    new ArrayStack<Integer>(25);
```

```
public class ArrayStack<Integer> {
    private Integer[] items;
    private int top;
    ...
    public boolean push(Integer item) {
        ...
    }
}
```



ArrayStack Constructor

- Java doesn't allow you to create an object or array using a type variable. Thus, we *cannot* do this:

```
public ArrayStack(int maxSize) {  
    // code to check for invalid maxSize goes here...  
    items = new T[maxSize]; // not allowed  
    top = -1;  
}
```

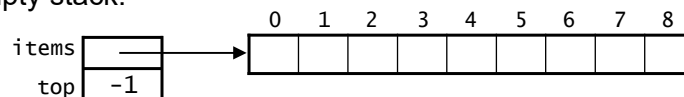
- Instead, we do this:

```
public ArrayStack(int maxSize) {  
    // code to check for invalid maxSize goes here...  
    items = (T[])new Object[maxSize];  
    top = -1;  
}
```

- The cast generates a compile-time warning, but we'll ignore it.
- Java's built-in ArrayList class takes this same approach.

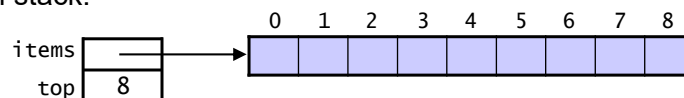
Testing if an ArrayStack is Empty or Full

- Empty stack:



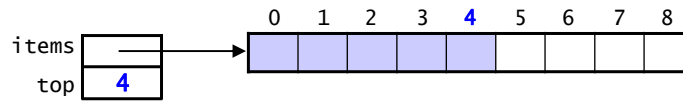
```
public boolean isEmpty() {  
    return (top == -1);  
}
```

- Full stack:



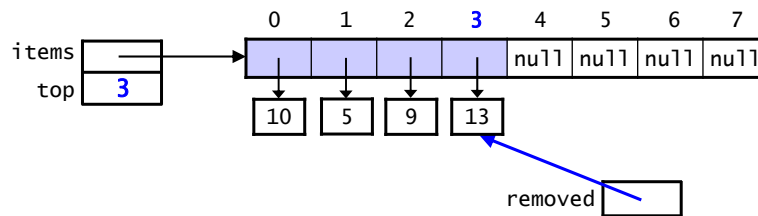
```
public boolean isFull() {  
    return (top == items.length - 1);  
}
```

Pushing an Item onto an ArrayStack



```
public boolean push(T item) {  
    // code to check for a null item goes here  
    if (isFull()) {  
        return false;  
    }  
    top++;  
    items[top] = item;  
    return true;  
}
```

ArrayStack pop() and peek()



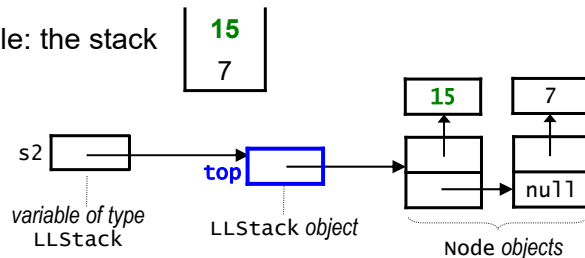
```
public T pop() {  
    if (isEmpty()) {  
        return null;  
    }  
  
    _____ removed = items[top];  
    items[top] = null;  
    top--;  
    return removed;  
}
```

- peek just returns `items[top]` without decrementing `top`.

Implementing a Generic Stack Using a Linked List

```
public class LLStack<T> implements Stack<T> {
    private Node top;    // top of the stack
    ...
}
```

- Example: the stack



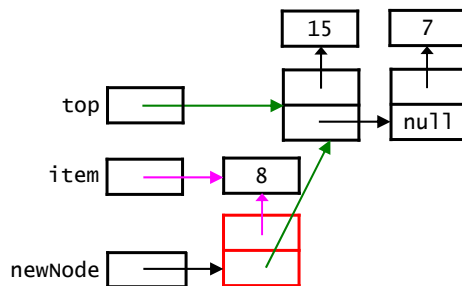
- Things worth noting:
 - our LLStack class needs only a single field: a reference to the first node, which holds the top item
 - top item = leftmost item (vs. rightmost item in ArrayStack)
 - we don't need a dummy node
 - only one case: always insert/delete at the front of the list!

Other Details of Our LLStack Class

```
public class LLStack<T> implements Stack<T> {
    private class Node {
        private T item;
        private Node next;
        ...
    }
    private Node top;
    public LLStack() {
        top = null;
    }
    public boolean isEmpty() {
        return (top == null);
    }
    public boolean isFull() {
        return false;
    }
}
```

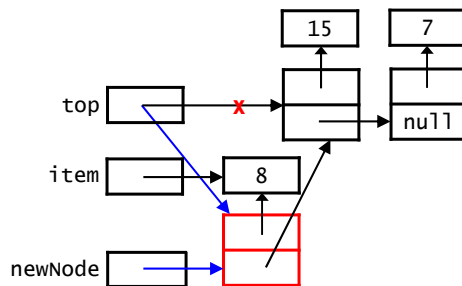
- The inner node class uses the type parameter T for the item.
- We don't need to preallocate any memory for the items.
- The stack is never full!

LLStack push()



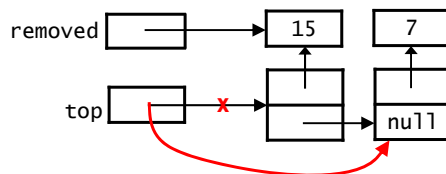
```
public boolean push(T item) {
    // code to check for a null item goes here
    Node newNode = new Node(item, top);
    top = newNode;
    return true;
}
```

LLStack push()



```
public boolean push(T item) {
    // code to check for a null item goes here
    Node newNode = new Node(item, top);
    top = newNode;
    return true;
}
```

LLStack pop() and peek()



```

public T pop() {
    if (isEmpty()) {
        return null;
    }
    T removed = _____;
    _____;
    return removed;
}

public T peek() {
    if (isEmpty()) {
        return null;
    }
    return top.item;
}

```

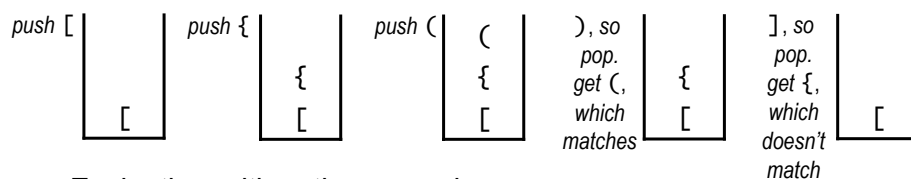
Efficiency of the Stack Implementations

	ArrayStack	LLStack
push()	$O(1)$	$O(1)$
pop()	$O(1)$	$O(1)$
peek()	$O(1)$	$O(1)$
space efficiency	$O(m)$ where m is the <i>anticipated</i> maximum number of items	$O(n)$ where n is the number of items currently on the stack

Applications of Stacks

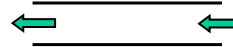
- Converting a recursive algorithm to an iterative one
 - use a stack to emulate the runtime stack
- Making sure that delimiters (parens, brackets, etc.) are balanced:
 - push open (i.e., left) delimiters onto a stack
 - when you encounter a close (i.e., right) delimiter, pop an item off the stack and see if it matches
 - example:

$5 * [3 + \{(5 + 16 - 2)\}]$



- Evaluating arithmetic expressions

Queue ADT



- A queue is a sequence in which:
 - items are added at the rear and removed from the front
 - first in, first out (FIFO) (vs. a stack, which is last in, first out)
 - you can only access the item that is currently at the front
- Operations:
 - insert: add an item at the rear of the queue
 - remove: remove the item at the front of the queue
 - peek: get the item at the front of the queue, but don't remove it
 - isEmpty: test if the queue is empty
 - isFull: test if the queue is full
- Example: a queue of integers
 - start:* 12 8
 - insert 5:* 12 8 5
 - remove:* 8 5

Our Generic Queue Interface

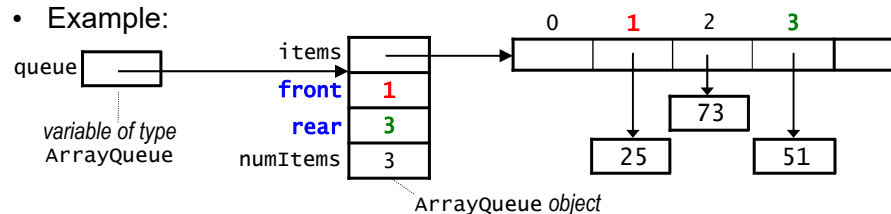
```
public interface Queue<T> {  
    boolean insert(T item);  
    T remove();  
    T peek();  
    boolean isEmpty();  
    boolean isFull();  
}
```

- `insert()` returns `false` if the queue is full, and `true` otherwise.
- `remove()` and `peek()` take no arguments, because we always access the item at the front of the queue.
 - return `null` if the queue is empty.
- Here again, we will use encapsulation to ensure that the data structure is manipulated only in valid ways.

Implementing a Queue Using an Array

```
public class ArrayQueue<T> implements Queue<T> {  
    private T[] items;  
    private int front;  
    private int rear;  
    private int numItems;  
    ...  
}
```

- Example:



- We maintain two indices:
 - `front`: the index of the item at the front of the queue
 - `rear`: the index of the item at the rear of the queue

Avoiding the Need to Shift Items

- Problem: what do we do when we reach the end of the array?

example: a queue of integers:

front								rear	
54	4	21	17	89	65				

the same queue after removing two items and inserting two:

front								rear	
		21	17	89	65	43	81		

we have room for more items, but shifting to make room is inefficient

- Solution: maintain a *circular queue*. When we reach the end of the array, we wrap around to the beginning.

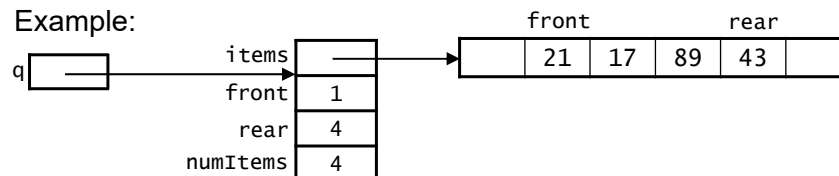
insert 5: wrap around!

rear		front							
5		21	17	89	65	43	81		

Maintaining a Circular Queue

- We use the mod operator (%) when updating front or rear:
 $\text{front} = (\text{front} + 1) \% \text{items.length};$
 $\text{rear} = (\text{rear} + 1) \% \text{items.length};$

- Example:



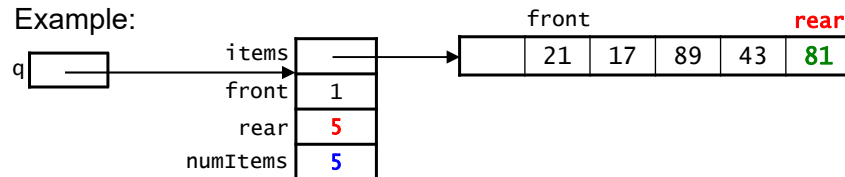
Maintaining a Circular Queue

- We use the mod operator (%) when updating front or rear:

$\text{front} = (\text{front} + 1) \% \text{items.length};$

$\text{rear} = (\text{rear} + 1) \% \text{items.length};$

- Example:



- `q.insert(81):` // rear is not at end of array
 - $\text{rear} = (\text{rear} + 1) \% \text{items.length};$
 - $= (4 + 1) \% 6$
 - $= 5 \% 6 = 5$ (% has no effect)

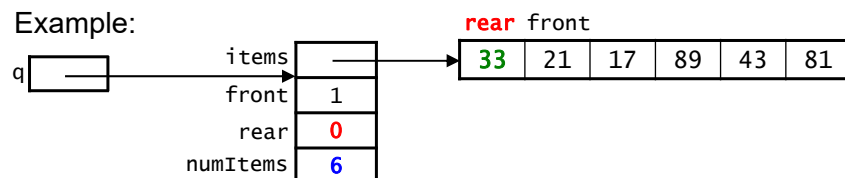
Maintaining a Circular Queue

- We use the mod operator (%) when updating front or rear:

$\text{front} = (\text{front} + 1) \% \text{items.length};$

$\text{rear} = (\text{rear} + 1) \% \text{items.length};$

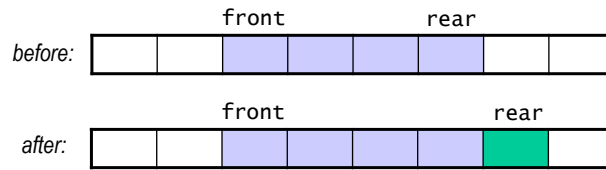
- Example:



- `q.insert(81):` // rear is not at end of array
 - $\text{rear} = (\text{rear} + 1) \% \text{items.length};$
 - $= (4 + 1) \% 6$
 - $= 5 \% 6 = 5$ (% has no effect)
- `q.insert(33):` // rear is at end of array
 - $\text{rear} = (\text{rear} + 1) \% \text{items.length};$
 - $= (5 + 1) \% 6$
 - $= 6 \% 6 = 0$ wrap around!

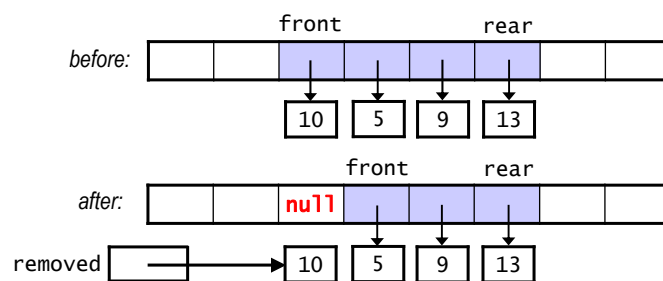
Inserting an Item in an ArrayQueue

- We increment rear before adding the item:



```
public boolean insert(T item) {
    // code to check for a null item goes here
    if (isFull()) {
        return false;
    }
    rear = (rear + 1) % items.length;
    items[rear] = item;
    numItems++;
    return true;
}
```

ArrayQueue remove()



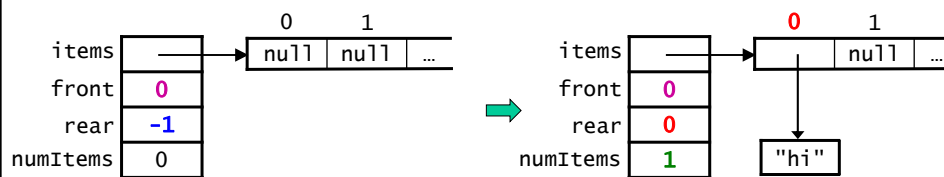
```
public T remove() {
    if (isEmpty()) {
        return null;
    }
    T removed = _____;

    numItems--;
    return removed;
}
```

Constructor

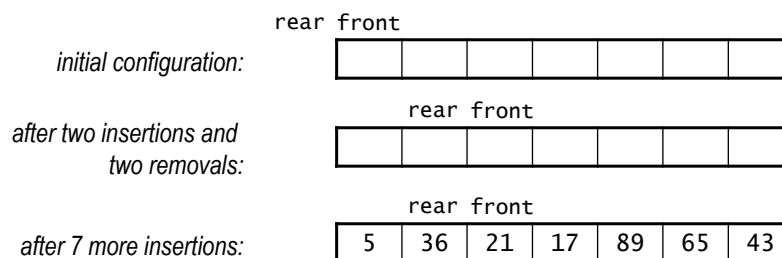
```
public ArrayQueue(int maxSize) {
    // code to check for an invalid maxSize goes here...
    items = (T[])new Object[maxSize];
    front = 0;
    rear = -1;
    numItems = 0;
}
```

- When we insert the first item in a newly created ArrayQueue, we want it to go in position 0. Thus, we need to:
 - start rear at **-1**, since then it will be incremented to **0** and used to perform the insertion
 - start front at **0**, since it is not changed by the insertion



Testing if an ArrayQueue is Empty or Full

- In both empty and full queues, rear is one "behind" front:



- This is why we maintain numItems!

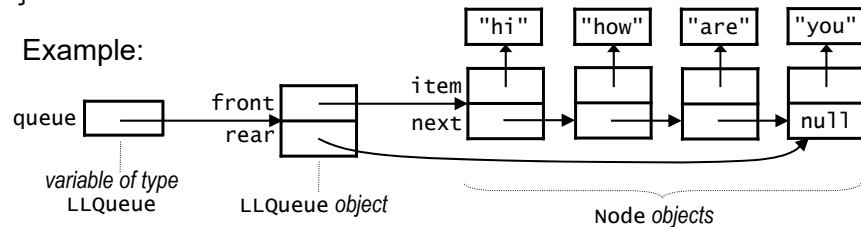
```
public boolean isEmpty() {
    return (numItems == 0);
}

public boolean isFull() {
    return (numItems == items.length);
}
```

Implementing a Queue Using a Linked List

```
public class LLQueue<T> implements Queue<T> {
    private Node front;    // front of the queue
    private Node rear;     // rear of the queue
    ...
}
```

- Example:



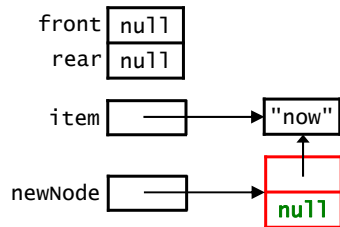
- In a linked list, we can efficiently:
 - remove the item at the front
 - add an item to the rear (if we have a ref. to the last node)
- Thus, this implementation is simpler than the array-based one!

Other Details of Our LLQueue Class

```
public class LLQueue<T> implements Queue<T> {
    private class Node {
        private T item;
        private Node next;
        ...
    }
    private Node front;
    private Node rear;

    public LLQueue() {
        front = null;
        rear = null;
    }
    public boolean isEmpty() {
        return (front == null);
    }
    public boolean isFull() {
        return false;
    }
    ...
}
```

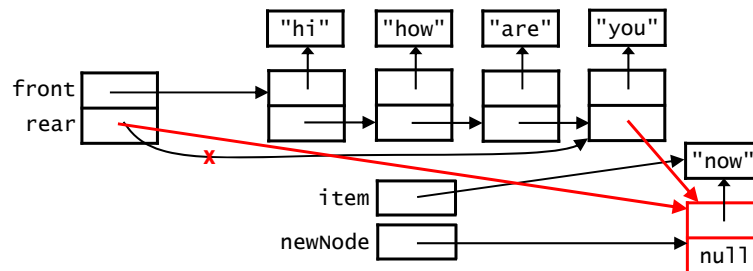
Inserting an Item in an Empty LLQueue



The next field in the newNode will be null regardless of whether the queue is empty. Why?

```
public boolean insert(T item) {
    // code to check for a null item goes here
    Node newNode = new Node(item, null);
    if (isEmpty()) {
        front = newNode;
        rear = newNode;
    } else {
        // we'll add this later!
    }
    return true;
}
```

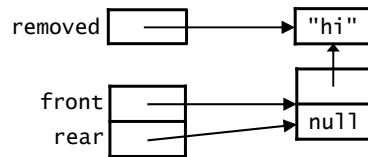
Inserting an Item in a Non-Empty LLQueue



```
public boolean insert(T item) {
    // code to check for a null item goes here
    Node newNode = new Node(item, null);
    if (isEmpty()) {
        front = newNode;
        rear = newNode;
    } else {
        // we'll add this later!
    }
    return true;
}
```

- A. rear = newNode;
rear.next = newNode;
- B. rear.next = newNode;
rear = newNode;
- C. either A or B
- D. neither A nor B

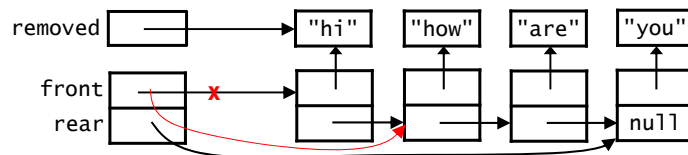
Removing from an LLQueue with One Item



```

public T remove() {
    if (isEmpty()) {
        return null;
    }
    T removed = _____;
    if (front == rear) {    // removing the only item
        front = null;
        rear = null;
    } else {
        // we'll add this later
    }
    return removed;
}
  
```

Removing from an LLQueue with Two or More Items



```

public T remove() {
    if (isEmpty()) {
        return null;
    }
    T removed = _____;
    if (front == rear) {    // removing the only item
        front = null;
        rear = null;
    } else {
    }
    return removed;
}
  
```

Efficiency of the Queue Implementations

	ArrayQueue	LLQueue
insert()	$O(1)$	$O(1)$
remove()	$O(1)$	$O(1)$
peek()	$O(1)$	$O(1)$
space efficiency	$O(m)$ where m is the <i>anticipated</i> maximum number of items	$O(n)$ where n is the number of items currently in the queue

Applications of Queues

- first-in first-out (FIFO) inventory control
- OS scheduling: processes, print jobs, packets, etc.
- simulations of banks, supermarkets, airports, etc.

Binary Trees and Huffman Encoding

Computer Science E-22
Harvard University

David G. Sullivan, Ph.D.

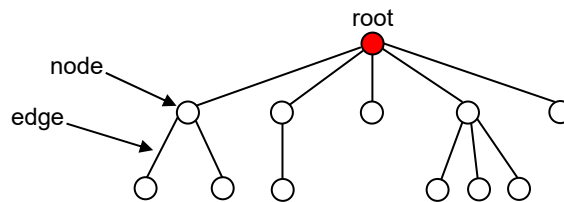
Motivation: Implementing a Dictionary

- A *data dictionary* is a collection of data with two main operations:
 - *search* for an item (and possibly delete it)
 - *insert* a new item
- If we use a *sorted* list to implement it, efficiency = $O(n)$.

<i>data structure</i>	<i>searching for an item</i>	<i>inserting an item</i>
a list implemented using an array	$O(\log n)$ using binary search	$O(n)$ because we need to shift items over
a list implemented using a linked list	$O(n)$ using linear search (binary search in a linked list is $O(n \log n)$)	$O(n)$ ($O(1)$ to do the actual insertion, but $O(n)$ to find where it belongs)

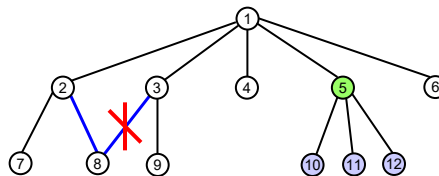
- In the next few lectures, we'll look at how we can use a *tree* for a data dictionary, and we'll try to get better efficiency.
- We'll also look at other applications of trees.

What Is a Tree?



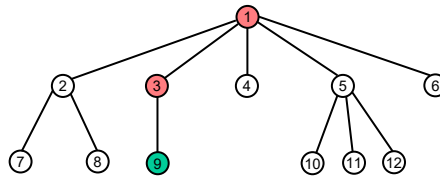
- A tree consists of:
 - a set of *nodes*
 - a set of *edges*, each of which connects a pair of nodes
- Each node may have one or more *data items*.
 - each data item consists of one or more fields
 - *key field* = the field used when searching for a data item
 - data items with the same key are referred to as *duplicates*
- The node at the "top" of the tree is called the *root* of the tree.

Relationships Between Nodes



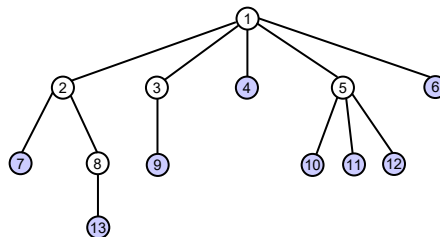
- If a node N is connected to nodes directly below it in the tree:
 - N is referred to as their *parent*
 - they are referred to as its *children*.
 - example: node 5 is the parent of nodes 10, 11, and 12
- Each node is the child of *at most one* parent.
- Nodes with the same parent are *siblings*.

Relationships Between Nodes (cont.)



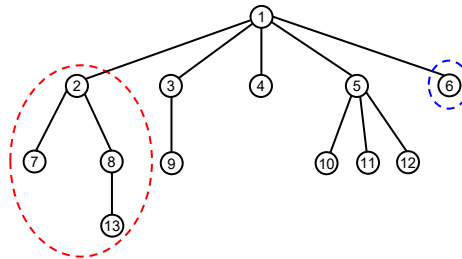
- A node's *ancestors* are its parent, its parent's parent, etc.
 - example: node 9's ancestors are 3 and 1
- A node's *descendants* are its children, their children, etc.
 - example: node 1's descendants are *all* of the other nodes

Types of Nodes



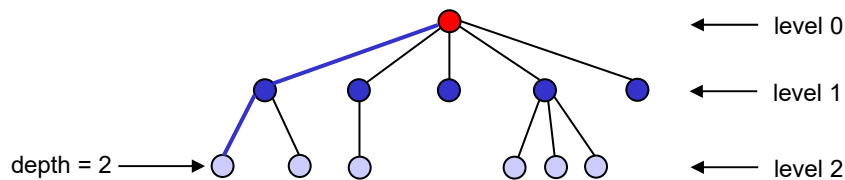
- A *leaf node* is a node without children.
- An *interior node* is a node with one or more children.

A Tree is a Recursive Data Structure



- Each node in the tree is the root of a smaller tree!
 - refer to such trees as *subtrees* to distinguish them from the tree as a whole
 - example: node 2 is the root of the subtree circled above
 - example: node 6 is the root of a subtree with only one node
- We'll see that tree algorithms often lend themselves to recursive implementations.

Path, Depth, Level, and Height

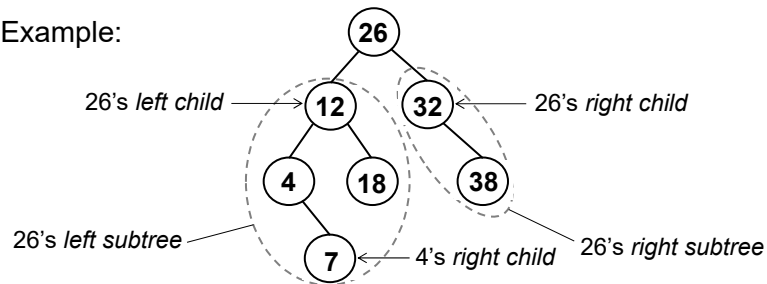


- There is exactly one *path* (one sequence of edges) connecting each node to the root.
- *depth* of a node = # of edges on the path from it to the root
- Nodes with the same depth form a *level* of the tree.
- The *height* of a tree is the maximum depth of its nodes.
 - example: the tree above has a height of 2

Binary Trees

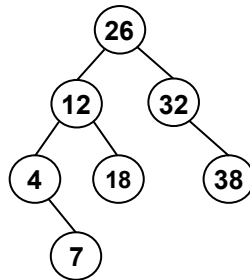
- In a *binary tree*, nodes have *at most two* children.
 - distinguish between them using the direction *left* or *right*

- Example:



- Recursive definition: a binary tree is either:
 - empty, or
 - a node (the root of the tree) that has:
 - one or more pieces of data (the key, and possibly others)
 - a *left subtree*, which is itself a binary tree
 - a *right subtree*, which is itself a binary tree

Which of the following are true?



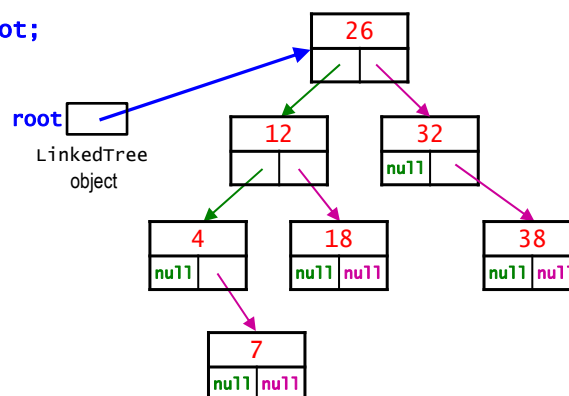
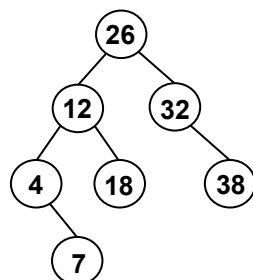
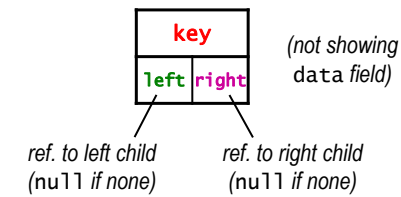
- A. This tree has a height of 4.
- B. There are 3 leaf nodes.
- C. The 38 node is the right child of the 32 node.
- D. The 12 node has 3 children.

Representing a Binary Tree Using Linked Nodes

```
public class LinkedTree {
    private class Node {
        private int key;           // limit ourselves to int keys
        private LList data;        // list of data for that key
        private Node left;        // reference to left child
        private Node right;       // reference to right child
        ...
    }
    private Node root;
    ...
}
```

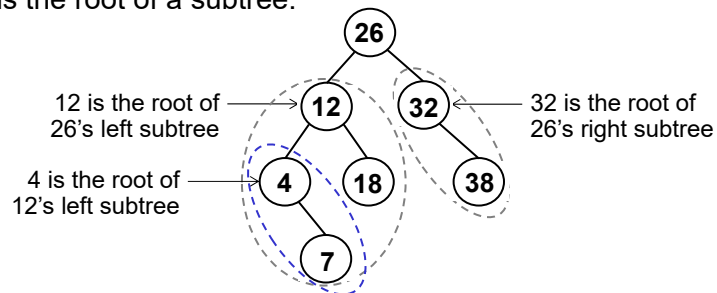
Representing a Binary Tree Using Linked Nodes

```
public class LinkedTree {
    private class Node {
        private int key;
        private LList data;
        private Node left;
        private Node right;
        ...
    }
    private Node root;
    ...
}
```



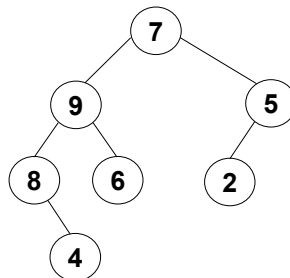
Traversing a Binary Tree

- Traversing a tree involves *visiting* all of the nodes in the tree.
 - visiting a node = processing its data in some way
 - example: print the key
- We'll look at four types of traversals.
 - each visits the nodes in a different order
- To understand traversals, it helps to remember that every node is the root of a subtree.



1: Preorder Traversal

- preorder traversal of the tree whose root is N:
 - 1) visit the root, N
 - 2) recursively perform a preorder traversal of N's left subtree
 - 3) recursively perform a preorder traversal of N's right subtree



- *preorder* because a node is visited *before* its subtrees
- The root of the tree as a whole is visited first.

Implementing Preorder Traversal

```
public class LinkedTree {
    ...
    private Node root;

    public void preorderPrint() {
        if (root != null) {
            preorderPrintTree(root);
        }
        System.out.println();
    }

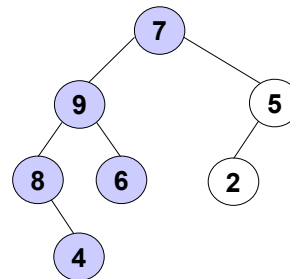
    private static void preorderPrintTree(Node root) {
        System.out.print(root.key + " ");
        if (root.left != null) {
            preorderPrintTree(root.left);
        }
        if (root.right != null) {
            preorderPrintTree(root.right);
        }
    }
}
```

*Not always the
same as the root
of the entire tree.*

- `preorderPrintTree()` is a static, recursive method that takes the root of the tree/subtree that you want to print.
- `preorderPrint()` is a non-static "wrapper" method that makes the initial call. It passes in the root of the entire tree.

Tracing Preorder Traversal

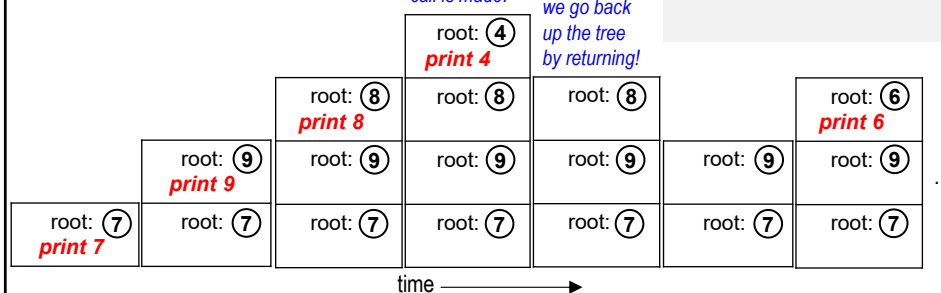
```
void preorderPrintTree(Node root) {
    System.out.print(root.key + " ");
    if (root.left != null) {
        preorderPrintTree(root.left);
    }
    if (root.right != null) {
        preorderPrintTree(root.right);
    }
}
```



order in which nodes are visited:

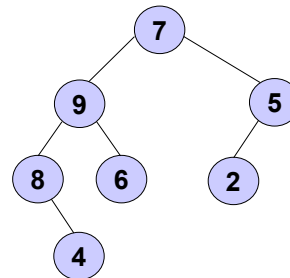
*base case, since
neither recursive
call is made!*

*we go back
up the tree
by returning!*



Using Recursion for Traversals

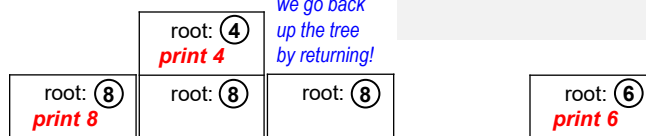
```
void preorderPrintTree(Node root) {
    System.out.print(root.key + " ");
    if (root.left != null) {
        preorderPrintTree(root.left);
    }
    if (root.right != null) {
        preorderPrintTree(root.right);
    }
}
```



*base case, since
neither recursive
call is made!*

*we go back
up the tree
by returning!*

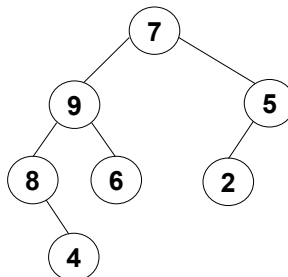
order in which nodes are visited:



- Using recursion allows us to easily go back up the tree.
- Using a loop would be harder. Why?

2: Postorder Traversal

- postorder traversal of the tree whose root is N:
 - 1) recursively perform a postorder traversal of N's left subtree
 - 2) recursively perform a postorder traversal of N's right subtree
 - 3) visit the root, N



- *postorder* because a node is visited *after* its subtrees
- The root of the tree as a whole is visited last.

Implementing Postorder Traversal

```
public class LinkedTree {
    ...
    private Node root;

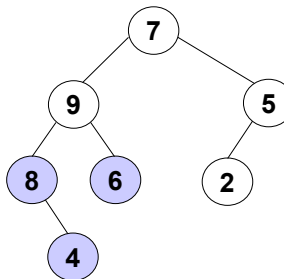
    public void postorderPrint() {
        if (root != null) {
            postorderPrintTree(root);
        }
        System.out.println();
    }

    private static void postorderPrintTree(Node root) {
        if (root.left != null) {
            postorderPrintTree(root.left);
        }
        if (root.right != null) {
            postorderPrintTree(root.right);
        }
        System.out.print(root.key + " ");
    }
}
```

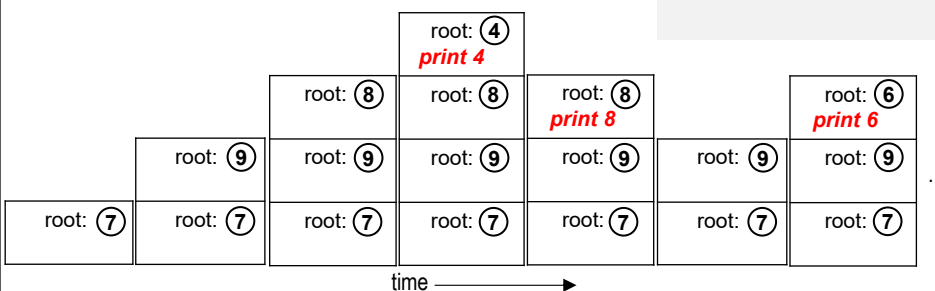
- Note that the root is printed *after* the two recursive calls.

Tracing Postorder Traversal

```
void postorderPrintTree(Node root) {
    if (root.left != null) {
        postorderPrintTree(root.left);
    }
    if (root.right != null) {
        postorderPrintTree(root.right);
    }
    System.out.print(root.key + " ");
}
```

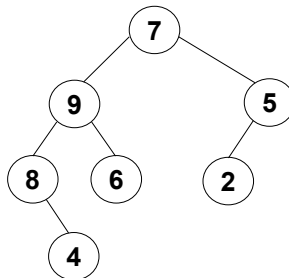


order in which nodes are visited:



3: Inorder Traversal

- inorder traversal of the tree whose root is N:
 - 1) recursively perform an inorder traversal of N's left subtree
 - 2) visit the root, N
 - 3) recursively perform an inorder traversal of N's right subtree



- The root of the tree as a whole is visited between its subtrees.
- We'll see later why this is called *inorder* traversal!

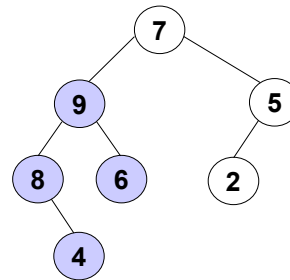
Implementing Inorder Traversal

```
public class LinkedTree {  
    ...  
    private Node root;  
    public void inorderPrint() {  
        if (root != null) {  
            inorderPrintTree(root);  
        }  
        System.out.println();  
    }  
    private static void inorderPrintTree(Node root) {  
        if (root.left != null) {  
            inorderPrintTree(root.left);  
        }  
        System.out.print(root.key + " ");  
        if (root.right != null) {  
            inorderPrintTree(root.right);  
        }  
    }  
}
```

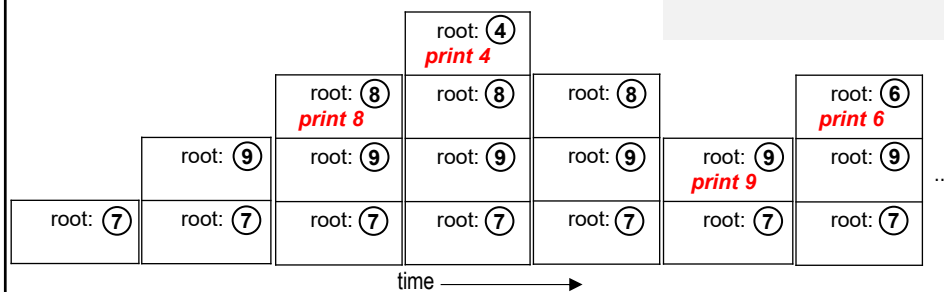
- Note that the root is printed *between* the two recursive calls.

Tracing Inorder Traversal

```
void inorderPrintTree(Node root) {
    if (root.left != null) {
        inorderPrintTree(root.left);
    }
    System.out.print(root.key + " ");
    if (root.right != null) {
        inorderPrintTree(root.right);
    }
}
```

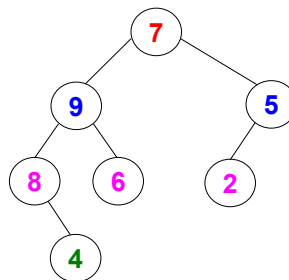


order in which nodes are visited:



Level-Order Traversal

- Visit the nodes one level at a time, from top to bottom and left to right.



- Level-order traversal of the tree above: **7 9 5 8 6 2 4**
- We can implement this type of traversal using a queue.

Tree-Traversal Summary

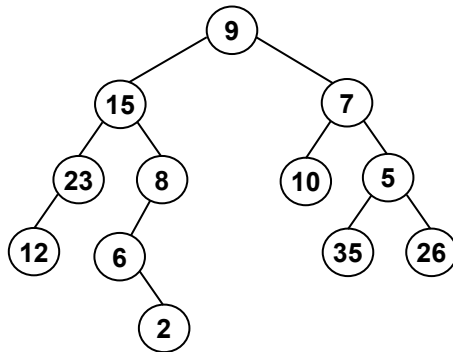
preorder: root, left subtree, right subtree

postorder: left subtree, right subtree, root

inorder: left subtree, root, right subtree

level-order: top to bottom, left to right

- Perform each type of traversal on the tree below:

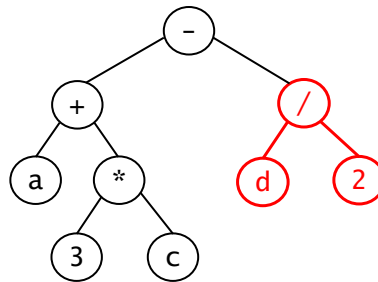


Tree Traversal Puzzle

- preorder traversal: A M P K L D H T
- inorder traversal: P M L K A H T D
- Draw the tree!
- What's one fact that we can easily determine from one of the traversals?
- How could we determine the nodes in each of the root's subtrees?
- What are the roots of each subtree?

Using a Binary Tree for an Algebraic Expression

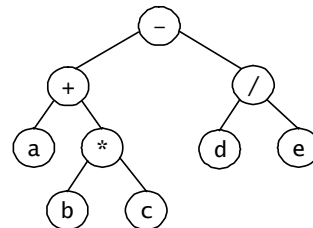
- We'll restrict ourselves to fully parenthesized expressions using the following binary operators: $+$, $-$, $*$, $/$
- Example: $((a + (3 * c)) - (d / 2))$



- Leaf nodes are variables or constants.
- Interior nodes are operators.
 - their children are their operands

Traversing an Algebraic-Expression Tree

- Inorder gives conventional algebraic notation.
 - print '(' before the recursive call on the left subtree
 - print ')' after the recursive call on the right subtree
 - for tree at right: $((a + (b * c)) - (d / e))$
- Preorder gives functional notation.
 - print '('s and ')'s as for inorder, and commas after the recursive call on the left subtree
 - for tree above: `subtr(add(a, mult(b, c)), divide(d, e))`
- Postorder gives the order in which the computation must be carried out on a stack calculator.
 - for tree above: push a, push b, push c, multiply, add,...



Fixed-Length Character Encodings

- A character encoding maps each character to a number.
- Computers usually use fixed-length character encodings.
 - ASCII - 8 bits per character

char	dec	binary
'a'	97	01100001
'b'	98	01100010
...	...	...
't'	116	01110100

example: "bat" is stored in a text file as the following sequence of bits:
01100010 01100001 01110100

- Unicode - 16 bits per character
(allows for foreign-language characters; ASCII is a subset)
- Fixed-length encodings are simple, because:
 - all encodings have the same length
 - a given character always has the same encoding

A Problem with Fixed-Length Encodings

- They tend to waste space.
- Example: an English newspaper article with only:
 - upper and lower-case letters (52 characters)
 - spaces and newlines (2 characters)
 - common punctuation (approx. 10 characters)
 - total of 64 unique characters → only need ____ bits
- We could gain even more space if we:
 - gave the most common letters shorter encodings (3 or 4 bits)
 - gave less frequent letters longer encodings (> 6 bits)

Variable-Length Character Encodings

- Variable-length encodings *compress* a text file by:
 - using encodings of different lengths for different characters
 - assigning shorter encodings to frequently occurring characters
- Example: if we had only four characters

e	01
o	100
s	111
t	00

"test" would be encoded as
00 01 111 00 → 000111100

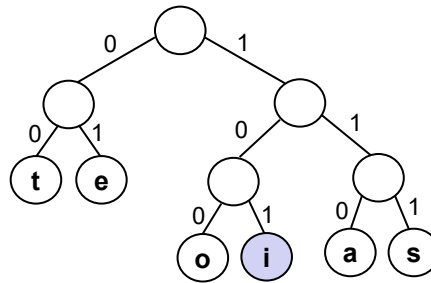
- Challenge: when reading a document, how do we determine the boundaries between characters?
 - how do we know how many bits the next character has?
- One requirement: no character's encoding can be the prefix of another character's encoding (e.g., couldn't have 00 and 001).

Huffman Encoding

- One type of variable-length encoding
- Based on the actual character frequencies in a given document
 - different documents have different encodings
- Huffman encoding uses a binary tree:
 - to determine the encoding of each character
 - to *decode* / *decompress* an encoded file
 - putting it back into ASCII

Huffman Trees

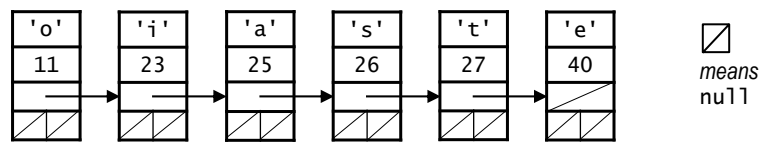
- Example for a text with only six characters:



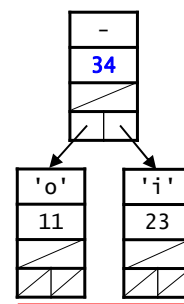
- Left branches are labeled with a 0, right branches with a 1.
- Leaf nodes are characters.
- To get a character's encoding, follow the path from the root to its leaf node.
 - example: i = ?

Building a Huffman Tree

- Begin by reading through the text to determine the frequencies.
- Create a list of nodes containing (character, frequency) pairs for each character in the text – *sorted by frequency*.

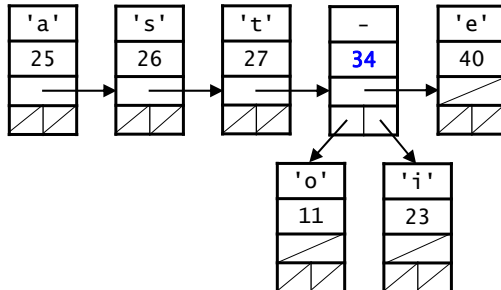


- Remove and "merge" the nodes with the two lowest frequencies, forming a new node that is their parent.
 - left child = lowest frequency node
 - right child = the other node
 - frequency of parent = sum of the frequencies of its children
 - in this case, $11 + 23 = 34$



Building a Huffman Tree (cont.)

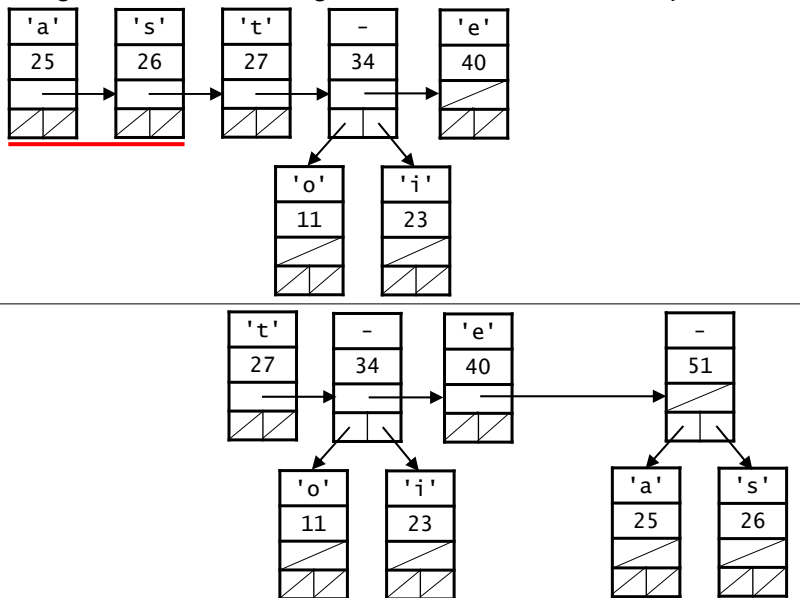
4) Add the parent to the list of nodes (maintaining sorted order):



5) Repeat steps 3 and 4 until there is only a single node in the list, which will be the root of the Huffman tree.

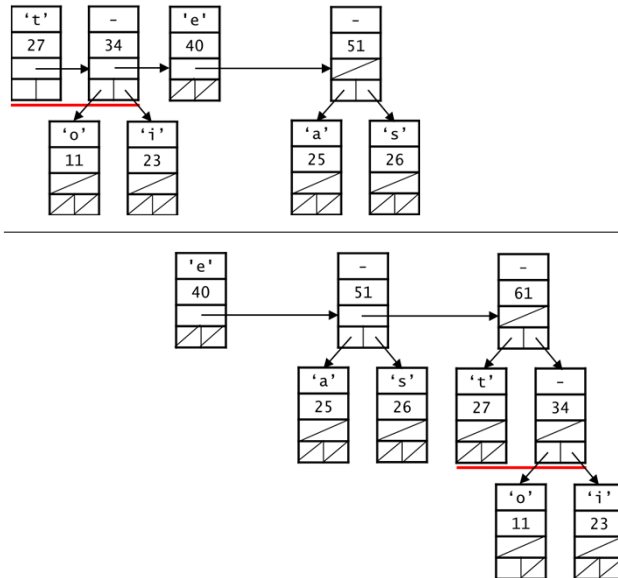
Completing the Huffman Tree Example I

• Merge the two remaining nodes with the lowest frequencies:



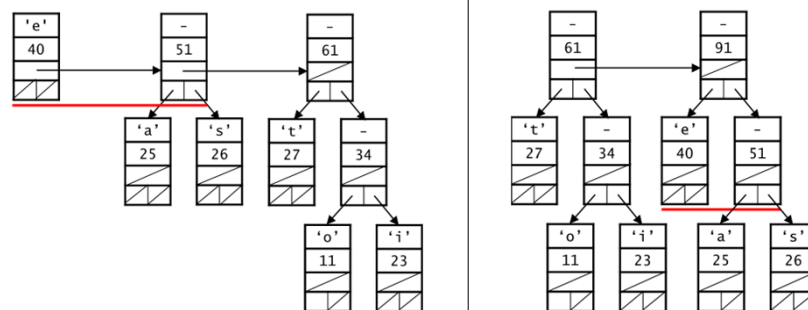
Completing the Huffman Tree Example II

- Merge the next two nodes:



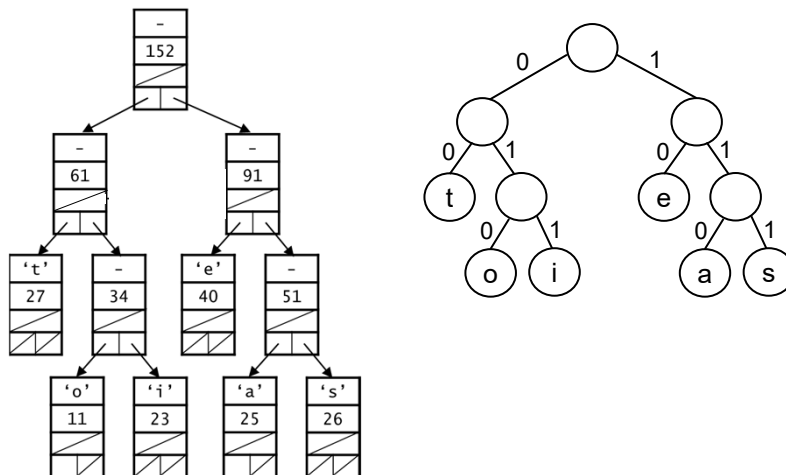
Completing the Huffman Tree Example II

- Merge again:



Completing the Huffman Tree Example IV

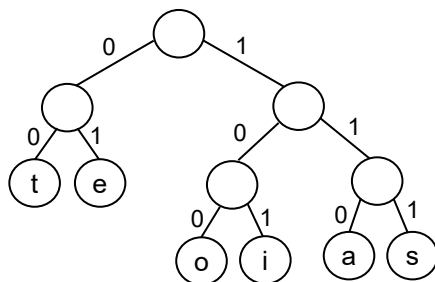
- The next merge creates the final tree:



- Characters that appear more frequently end up higher in the tree, and thus their encodings are shorter.

The Shape of the Huffman Tree

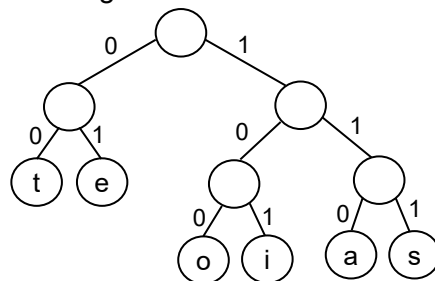
- The tree on the last slide is fairly symmetric.
- This won't always be the case!
 - depends on the character frequencies
- For example, changing the frequency of 'o' from 11 to 21 would produce the tree shown below:



- This is the tree that we'll use in the remaining slides.

Huffman Encoding: Compressing a File

- 1) Read through the input file and build its Huffman tree.
- 2) Write a file header for the output file.
 - include the character frequencies so the tree can be rebuilt when the file is decompressed
- 3) Traverse the Huffman tree to create a table containing the encoding of each character:



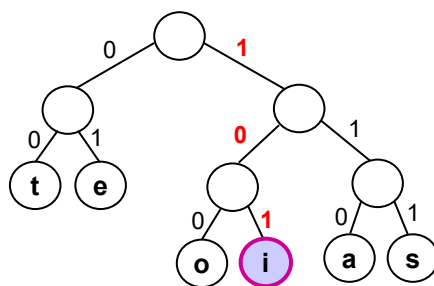
a	
e	
i	101
o	100
s	111
t	

- 4) Read through the input file a second time, and write the Huffman code for each character to the output file.

Huffman Decoding: Decompressing a File

- 1) Read the frequency table from the header and rebuild the tree.
- 2) Read one bit at a time and traverse the tree, starting from the root:
 - when you read a bit of 1, go to the right child
 - when you read a bit of 0, go to the left child
 - when you reach a leaf node, record the character, return to the root, and continue reading bits

The tree allows us to easily overcome the challenge of determining the character boundaries!

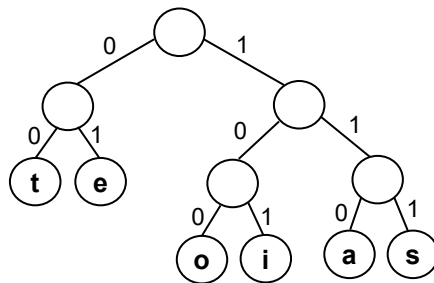


example: **101**111110000111100
first character = i

What are the next three characters?

- 1) Read the frequency table from the header and rebuild the tree.
- 2) Read one bit at a time and traverse the tree, starting from the root:
 - when you read a bit of 1, go to the right child
 - when you read a bit of 0, go to the left child
 - when you reach a leaf node, record the character,
 - return to the root, and continue reading bits

The tree allows us to easily overcome the challenge of determining the character boundaries!

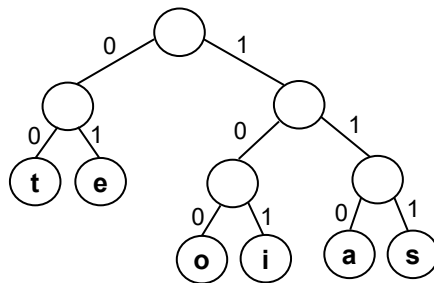


example: 101111110000111100
first character = i (101)

Huffman Decoding: Decompressing a File

- 1) Read the frequency table from the header and rebuild the tree.
- 2) Read one bit at a time and traverse the tree, starting from the root:
 - when you read a bit of 1, go to the right child
 - when you read a bit of 0, go to the left child
 - when you reach a leaf node, record the character,
 - return to the root, and continue reading bits

The tree allows us to easily overcome the challenge of determining the character boundaries!



example: 101111110000111100

101 = right,left,right = i
111 = right,right,right= s
110 = right,right,left = a
00 = left,left = t
01 = left,right = e
111 = right,right,right= s
00 = left,left = t

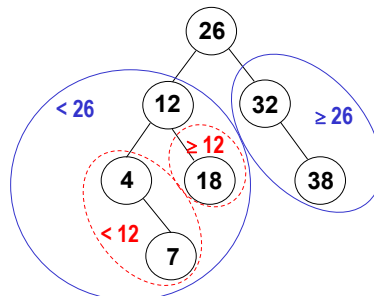
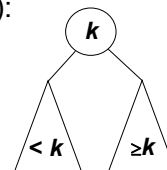
Search Trees

Computer Science E-22
Harvard University

David G. Sullivan, Ph.D.

Binary Search Trees

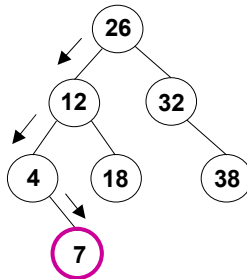
- Search-tree property: for each node k (k is the key):
 - all nodes in k 's left subtree are $< k$
 - all nodes in k 's right subtree are $\geq k$
- Our earlier binary-tree example is a search tree:



- With a search tree, an inorder traversal visits the nodes in order!
 - in order of increasing key values

Searching for an Item in a Binary Search Tree

- Algorithm for searching for an item with a key k :
 - if $k ==$ the root node's key, you're done
 - else if $k <$ the root node's key, search the left subtree
 - else search the right subtree
- Example: search for 7



Implementing Binary-Tree Search

```
public class LinkedTree {    // Nodes have keys that are ints
    ...
    private Node root;

    public LList search(int key) {    // "wrapper method"
        Node n = searchTree(root, key); // get Node for key
        if (n == null) {
            return null;    // no such key
        } else {
            return n.data;    // return list of values for key
        }
    }

    private static Node searchTree(Node root, int key) {
        if (key == null) {
            return null;
        } else if (key < root.key) {
            return searchTree(root.left, key);
        } else if (key > root.key) {
            return searchTree(root.right, key);
        } else {
            return root;
        }
    }
}
```

two base cases
(order matters!)

two
recursive cases

Inserting an Item in a Binary Search Tree

- `public void insert(int key, Object data)`
will add a new (key, data) pair to the tree
- Example 1: a search tree containing student records
 - key = the student's ID number (an integer)
 - data = a string with the rest of the student record
 - we want to be able to write client code that looks like this:

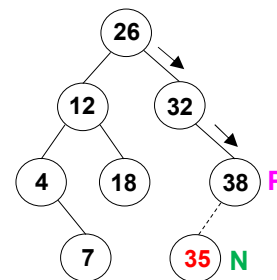
```
LinkedTree students = new LinkedTree();
students.insert(23, "Jill Jones,sophomore,comp sci");
students.insert(45, "Al Zhang,junior,english");
```
- Example 2: a search tree containing scrabble words
 - key = a scrabble score (an integer)
 - data = a word with that scrabble score

```
LinkedTree tree = new LinkedTree();
tree.insert(4, "lost");
```

Inserting an Item in a Binary Search Tree (cont.)

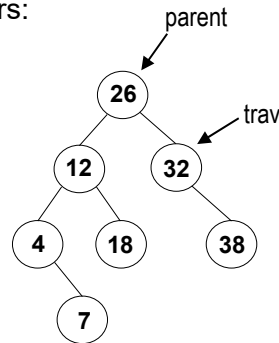
- To insert an item (k, d) ,
we start by searching for k .
- If we find a node with key k , we add d to the list of data values for that node.
 - example: `tree.insert(4, "sail")`
- If we don't find k , the last node seen in the search becomes the parent **P** of the new node **N**.
 - if $k < \mathbf{P}$'s key, make **N** the left child of **P**
 - else make **N** the right child of **P**
- *Special case*: if the tree is empty,
make the new node the root of the tree.
- **Important**: The resulting tree is still a search tree!

example:
`tree.insert(35,
"photooxidizes")`



Implementing Binary-Tree Insertion

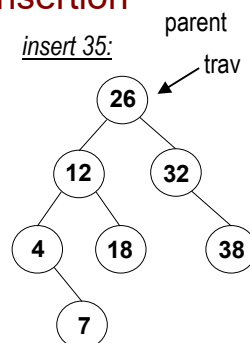
- We'll implement part of the `insert()` method together.
- We'll use iteration rather than recursion.
- Our method will use two references/pointers:
 - `trav`: performs the traversal down to the point of insertion
 - `parent`: stays one behind `trav`
 - like the `trail` reference that we sometimes use when traversing a linked list



Implementing Binary-Tree Insertion

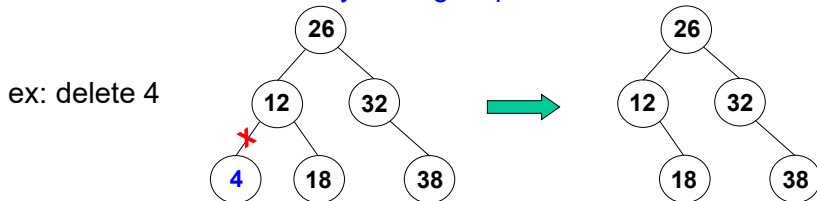
```
public void insert(int key, Object data) {
    Node parent = null;
    Node trav = root;
    while (trav != null) {
        if (trav.key == key) {
            trav.data.addItem(data, 0);
            return;
        }
        // what should go here?
    }
```

```
    Node newNode = new Node(key, data);
    if (root == null) { // the tree was empty
        root = newNode;
    } else if (key < parent.key) {
        parent.left = newNode;
    } else {
        parent.right = newNode;
    }
}
```

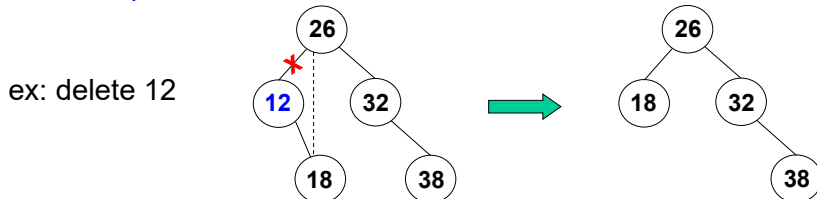


Deleting Items from a Binary Search Tree

- Three cases for deleting a node x
- Case 1:** x has no children.
Remove x from the tree by setting its parent's reference to null.



- Case 2:** x has one child.
Take the parent's reference to x and make it refer to x 's child.

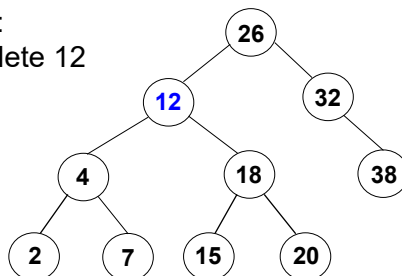


Deleting Items from a Binary Search Tree (cont.)

- Case 3:** x has two children
 - we can't give both children to the parent. why?
 - instead, we leave x 's node where it is, and we replace its key and data with those from another node
 - the replacement must maintain the search-tree inequalities

ex:
delete 12

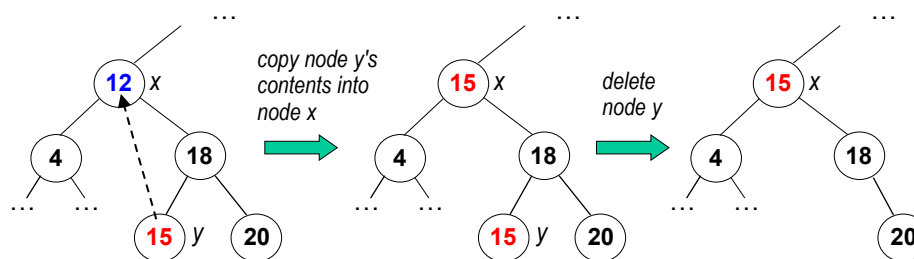
two options: which ones?



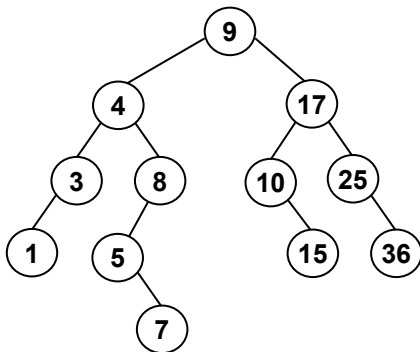
Deleting Items from a Binary Search Tree (cont.)

- **Case 3:** x has two children (continued):
 - replace x 's key and data with those from the smallest node in x 's right subtree—call it y
 - we then delete y
 - it will either be a leaf node or will have one right child. why?
 - thus, we can delete it using case 1 or 2

ex: delete 12



Which Node Would Be Used To Replace 9?



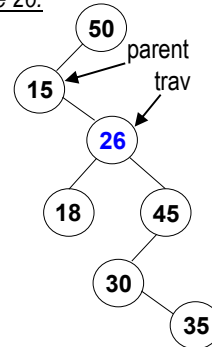
- A. 4
- B. 8
- C. 10
- D. 15
- E. 17

Implementing Deletion

```
public LList delete(int key) {
    // Find the node and its parent.
    Node parent = null;
    Node trav = root;
    while (trav != null && trav.key != key) {
        parent = trav;
        if (key < trav.key) {
            trav = trav.left;
        } else {
            trav = trav.right;
        }
    }

    // Delete the node (if any) and return the removed items.
    if (trav == null) { // no such key
        return null;
    } else {
        LList removedData = trav.data;
        deleteNode(trav, parent); // call helper method
        return removedData;
    }
}
```

delete 26:



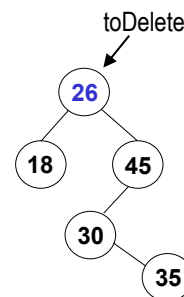
Implementing Case 3

```
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left != null && toDelete.right != null) {
        // Find a replacement - and
        // the replacement's parent.
        Node replaceParent = toDelete;

        // Get the smallest item
        // in the right subtree.
        Node replace = toDelete.right;
        // what should go here?

        // Replace toDelete's key and data
        // with those of the replacement item.
        toDelete.key = replace.key;
        toDelete.data = replace.data;

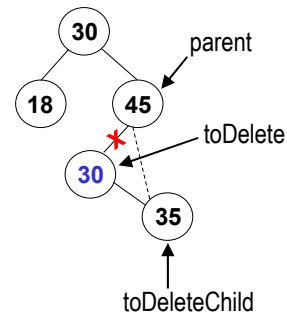
        // Recursively delete the replacement
        // item's old node. It has at most one
        // child, so we don't have to
        // worry about infinite recursion.
        deleteNode(replace, replaceParent);
    } else {
        ...
    }
}
```



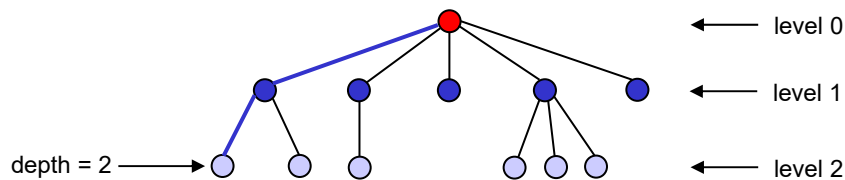
Implementing Cases 1 and 2

```
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left != null && toDelete.right != null) {
        ...
    } else {
        Node toDeleteChild;
        if (toDelete.left != null)
            toDeleteChild = toDelete.left;
        else
            toDeleteChild = toDelete.right;
        // Note: in case 1, toDeleteChild
        // will have a value of null.

        if (toDelete == root)
            root = toDeleteChild;
        else if (toDelete.key < parent.key)
            parent.left = toDeleteChild;
        else
            parent.right = toDeleteChild;
    }
}
```



Recall: Path, Depth, Level, and Height



- There is exactly one *path* (one sequence of edges) connecting each node to the root.
- *depth* of a node = # of edges on the path from it to the root
- Nodes with the same depth form a *level* of the tree.
- The *height* of a tree is the maximum depth of its nodes.
 - example: the tree above has a height of 2

Efficiency of a Binary Search Tree

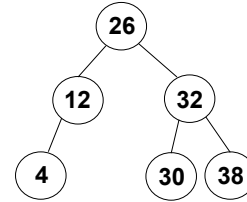
- For a tree containing n items, what is the efficiency of any of the traversal algorithms?
 - you process all n of the nodes
 - you perform $O(1)$ operations on each of them
- Search, insert, and delete all have the same time complexity.
 - insert is a search followed by $O(1)$ operations
 - delete involves either:
 - a search followed by $O(1)$ operations (cases 1 and 2)
 - a search partway down the tree for the item, followed by a search further down for its replacement, followed by $O(1)$ operations (case 3)

Efficiency of a Binary Search Tree (cont.)

- Time complexity of searching:
 - best case:
 - worst case:
 - you have to go all the way down to level h before finding the key or realizing it isn't there
 - along the path to level h , you process $h + 1$ nodes
 - average case:
- What is the height of a tree containing n items?

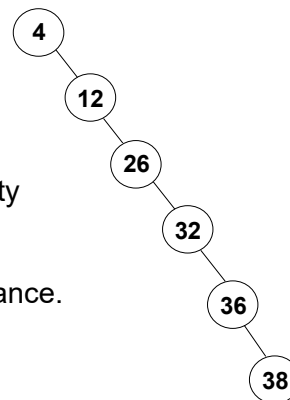
Balanced Trees

- A tree is *balanced* if, for *each* of its nodes, the node's subtrees have the same height or have heights that differ by 1.
 - example:
 - 26: both subtrees have a height of 1
 - 12: left subtree has height 0
right subtree is empty (height = -1)
 - 32: both subtrees have a height of 0
 - all leaf nodes: both subtrees are empty
- For a balanced tree with n nodes, height = $O(\log n)$
 - each time that you follow an edge down the longest path, you cut the problem size roughly in half!
- Therefore, for a *balanced* binary search tree, the worst case for search / insert / delete is $O(h) = O(\log n)$
 - the "best" worst-case time complexity



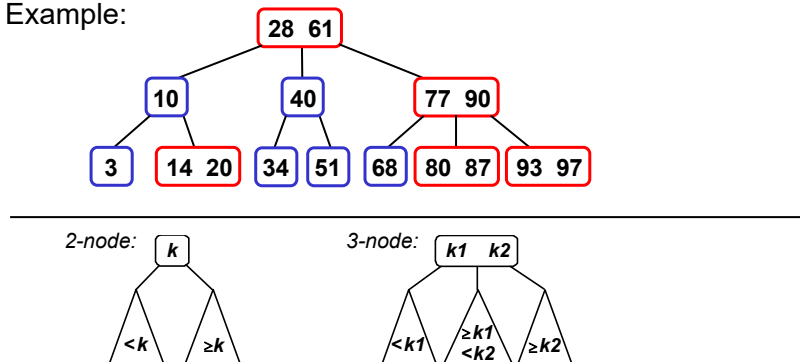
What If the Tree Isn't Balanced?

- Extreme case: the tree is equivalent to a linked list
 - height = $n - 1$
- Therefore, for an unbalanced binary search tree, the worst case for search / insert / delete is $O(h) = O(n)$
 - the "worst" worst-case time complexity
- We'll look next at search-tree variants that take special measures to ensure balance.



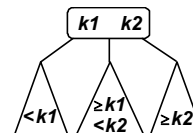
2-3 Trees

- A 2-3 tree is a balanced tree in which:
 - all nodes have equal-height subtrees (perfect balance)
 - each node is either
 - a **2-node**, which contains one data item and 0 or 2 children
 - a **3-node**, which contains two data items and 0 or 3 children
 - the keys form a search tree
- Example:

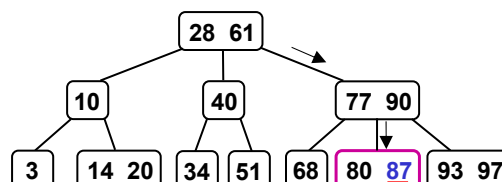


Search in 2-3 Trees

- Algorithm for searching for an item with a key k :
 - if $k ==$ one of the root node's keys, you're done
 - else if $k <$ the root node's first key
 - search the left subtree
 - else if the root is a 3-node and $k <$ its second key
 - search the middle subtree
 - else
 - search the right subtree

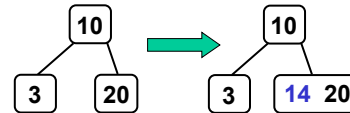


- Example: search for 87

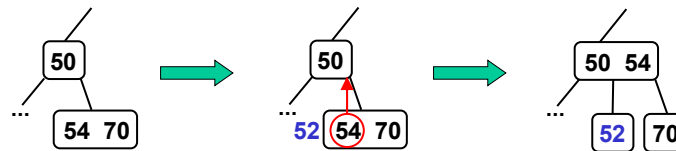


Insertion in 2-3 Trees

- Algorithm for inserting an item with a key k :
 - search for k , but don't stop until you hit a leaf node
 - let L be the leaf node at the end of the search
 - if L is a 2-node
 - add k to L , making it a 3-node
 - else if L is a 3-node
 - split L into two 2-nodes containing the items with the smallest and largest of: k , L 's 1st key, L 's 2nd key
 - the middle item is "sent up" and inserted in L 's parent

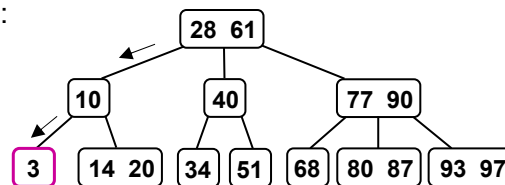


example: add 52

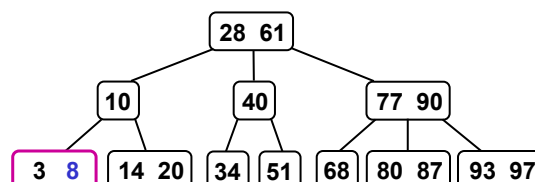


Example 1: Insert 8

- Search for 8:

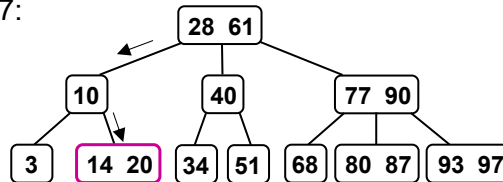


- Add 8 to the leaf node, making it a 3-node:

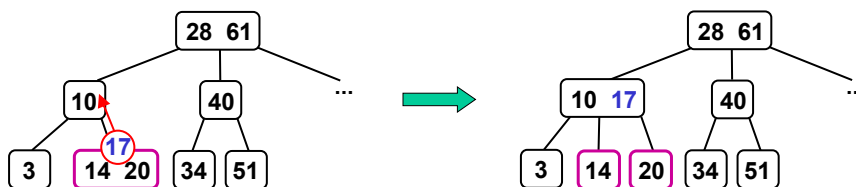


Example 2: Insert 17

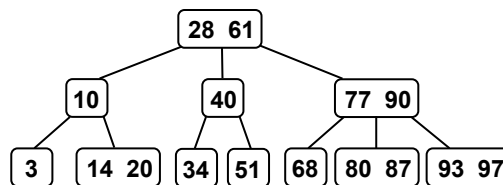
- Search for 17:



- Split the leaf node, and send up the middle of 14, 17, 20 and insert it the leaf node's parent:



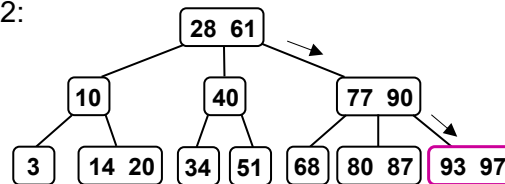
Example 3: Insert 92



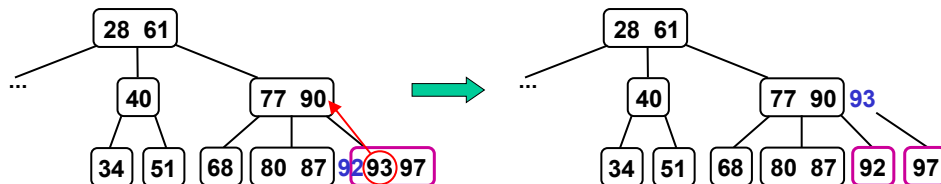
- In which node will we initially try to insert it?

Example 3: Insert 92

- Search for 92:



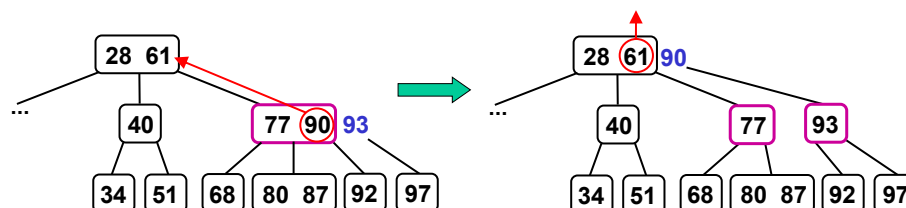
- Split the leaf node, and send up the middle of 92, 93, 97 and insert it the leaf node's parent:



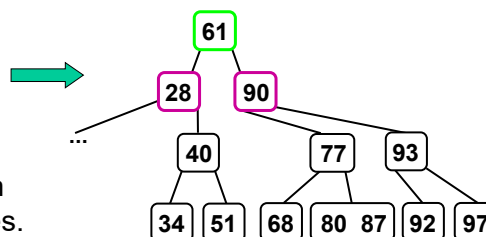
- In this case, the leaf node's parent is also a 3-node, so we need to split it as well...

Example 3 (cont.)

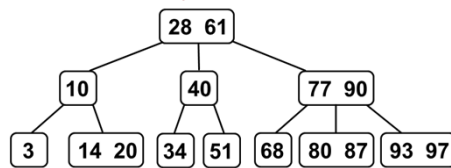
- We split the [77 90] node and we send up the middle of 77, 90, 93:
- We try to insert it in the root node, but the root is also full!



- Then we split the root, which increases the tree's height by 1, but the tree is still balanced.
- This is only case in which the tree's height increases.



Efficiency of 2-3 Trees



- A 2-3 tree containing n items has a height $h \leq \log_2 n$.
- Thus, search and insertion are both $O(\log n)$.
 - search visits at most $h + 1$ nodes
 - insertion visits at most $2h + 1$ nodes:
 - starts by going down the full height
 - in the worst case, performs splits all the way back up to the root
- Deletion is tricky – you may need to coalesce nodes! However, it also has a time complexity of $O(\log n)$.
- Thus, we can use 2-3 trees for a $O(\log n)$ -time data dictionary!

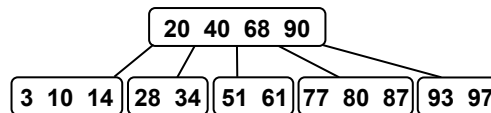
External Storage

- The balanced trees that we've covered don't work well if you want to store the data dictionary externally – i.e., on disk.
- Key facts about disks:
 - data is transferred to and from disk in units called *blocks*, which are typically 4 or 8 KB in size
 - disk accesses are slow!
 - reading a block takes ~10 milliseconds (10^{-3} sec)
 - vs. reading from memory, which takes ~10 nanoseconds
 - in 10 ms, a modern CPU can perform millions of operations!

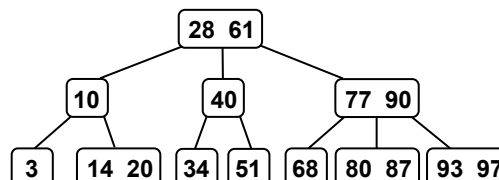
B-Trees

- A B-tree of order m is a tree in which each node has:
 - at most $2m$ entries (and, for internal nodes, $2m + 1$ children)
 - at least m entries (and, for internal nodes, $m + 1$ children)
 - exception: the root node may have as few as 1 entry
 - a 2-3 tree is essentially a B-tree of order 1
- To minimize the number of disk accesses, we make m as large as possible.
 - each disk read brings in more items
 - the tree will be shorter (each level has more nodes), and thus searching for an item requires fewer disk reads
- A large value of m doesn't make sense for a memory-only tree, because it leads to many key comparisons per node.
- These comparisons are less expensive than accessing the disk, so large values of m make sense for on-disk trees.

Example: a B-Tree of Order 2



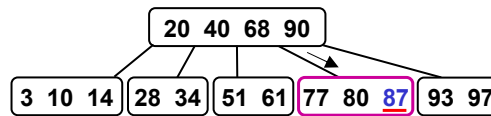
- $m = 2$: at most $2m = 4$ items per node (and at most 5 children)
at least $m = 2$ items per node (and at least 3 children)
(except the root, which could have 1 item)
- The above tree holds the same keys this 2-3 tree:



- We used the same order of insertion to create both trees:
51, 3, 40, 77, 20, 10, 34, 28, 61, 80, 68, 93, 90, 97, 87, 14

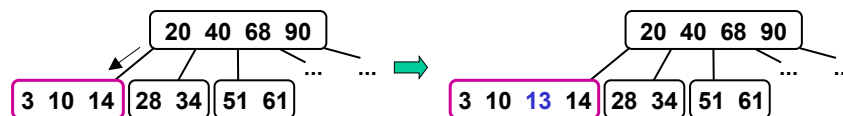
Search in B-Trees

- Similar to search in a 2-3 tree.
- Example: search for 87



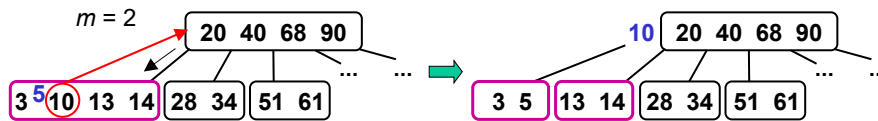
Insertion in B-Trees

- Similar to insertion in a 2-3 tree:
 - search for the key until you reach a leaf node
 - if a leaf node has fewer than $2m$ items, add the item to the leaf node
 - else split the node, dividing up the $2m + 1$ items:
 - the smallest m items remain in the original node
 - the largest m items go in a new node
 - send the middle entry up and insert it (and a pointer to the new node) in the parent
- Example of an insertion without a split: insert 13

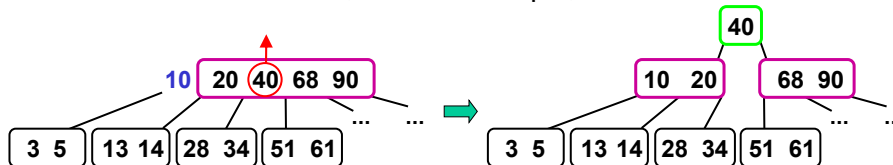


Splits in B-Trees

- Insert 5 into the result of the previous insertion:

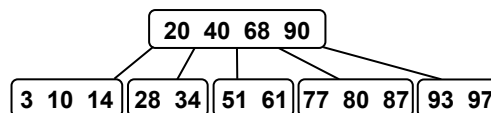


- The middle item (the 10) is sent up to the root.
The root has no room, so it is also split, and a new root is formed:



- Splitting the root increases the tree's height by 1, but the tree is still balanced. This is only way that the tree's height increases.
- When an internal node is split, its $2m + 2$ pointers are split evenly between the original node and the new node.

Analysis of B-Trees



- All internal nodes have at least m children (actually, at least $m+1$).
- Thus, a B-tree with n items has a height $\leq \log_m n$, and search and insertion are both $O(\log_m n)$.
- As with 2-3 trees, deletion is tricky, but it's still logarithmic.

Search Trees: Conclusions

- Binary search trees can be $O(\log n)$, but they can degenerate to $O(n)$ running time if they are out of balance.
- 2-3 trees and B-trees are *balanced* search trees that guarantee $O(\log n)$ performance.
- When data is stored on disk, the most important performance consideration is reducing the number of disk accesses.
- B-trees offer improved performance for on-disk data dictionaries.

Heaps and Priority Queues

Computer Science E-22
Harvard University

David G. Sullivan, Ph.D.

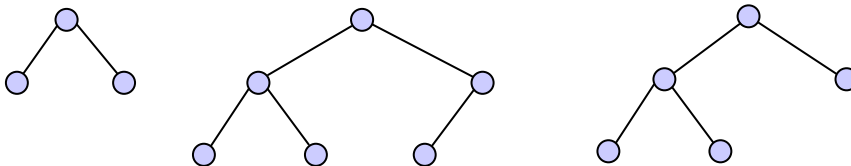
Priority Queue

- A *priority queue* (PQ) is a collection in which each item has an associated number known as a *priority*.
 - ("Ann Cudd", 10), ("Robert Brown", 15), ("Dave Sullivan", 5)
 - use a higher priority for items that are "more important"
- Example application: scheduling a shared resource like the CPU
 - give some processes/applications a higher priority, so that they will be scheduled first and/or more often
- Key operations:
 - *insert*: add an item (with a position based on its priority)
 - *remove*: remove the item with the highest priority
- One way to implement a PQ efficiently is using a type of binary tree known as a *heap*.

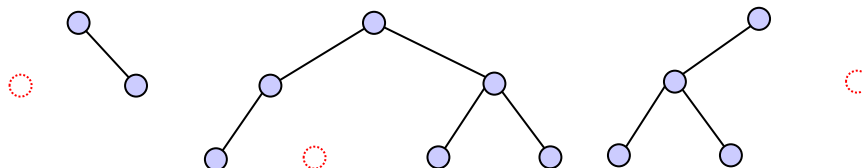
Complete Binary Trees

- A binary tree of height h is *complete* if:
 - levels 0 through $h - 1$ are fully occupied
 - there are no “gaps” to the left of a node in level h

- Complete:

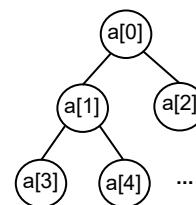


- Not complete (○ = missing node):

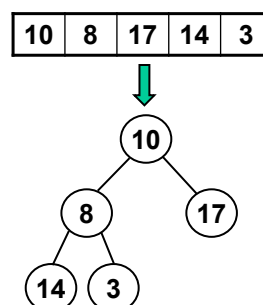
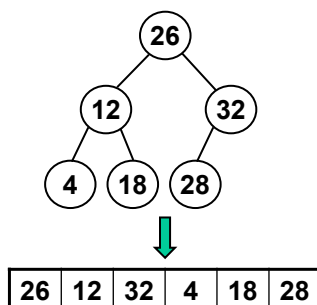


Representing a Complete Binary Tree

- A complete binary tree has a simple array representation.
- The tree's nodes are stored in the array in the order given by a level-order traversal.
 - top to bottom, left to right

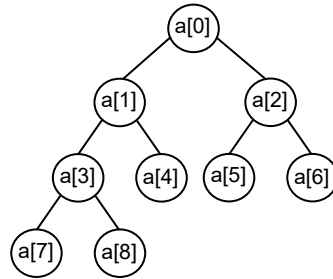


- Examples:



Navigating a Complete Binary Tree in Array Form

- The root node is in $a[0]$
- Given the node in $a[i]$:
 - its left child is in $a[2*i + 1]$
 - its right child is in $a[2*i + 2]$
 - its parent is in $a[(i - 1)/2]$ (using integer division)



- Examples:
 - the left child of the node in $a[1]$ is in $a[2*1 + 1] = a[3]$
 - the left child of the node in $a[2]$ is in $a[2*2 + 1] = a[5]$
 - the right child of the node in $a[3]$ is in $a[2*3 + 2] = a[8]$
 - the right child of the node in $a[2]$ is in _____
 - the parent of the node in $a[4]$ is in $a[(4 - 1)/2] = a[1]$
 - the parent of the node in $a[7]$ is in _____

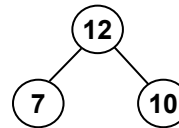
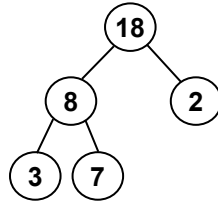
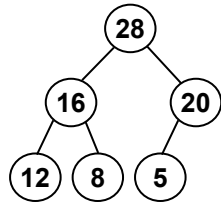
What is the left child of 24?

- Assume that the following array represents a complete tree:

0	1	2	3	4	5	6	7	8
26	12	32	24	18	28	47	10	9

Heaps

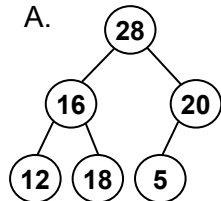
- Heap: a complete binary tree in which each interior node is greater than or equal to its children
 - examples:



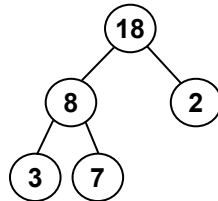
- The largest value is always at the root of the tree.
- The smallest value can be in *any* leaf node - there's no guarantee about which one it will be.
- We're using *max-at-top* heaps.
 - in a *min-at-top* heap, every interior node $\leq$ its children

Which of these is a heap?

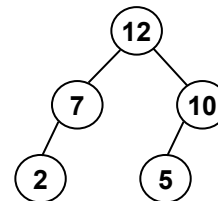
• A.



B.



C.



D. more than one (which ones?)

E. none of them

How to Compare Objects

- We need to be able to compare items in the heap.
- If those items are objects, we can't just do something like this:

```
if (item1 < item2)
```

Why not?

- Instead, we need to use a method to compare them.

An Interface for Objects That Can Be Compared

- The Comparable interface is a built-in generic Java interface:

```
public interface Comparable<T> {  
    public int compareTo(T other);  
}
```

- It is used when defining a class of objects that can be ordered.
- Examples from the built-in Java classes:

```
public class String implements Comparable<String> {  
    ...  
    public int compareTo(String other) {  
        ...  
    }  
    public class Integer implements Comparable<Integer> {  
        ...  
        public int compareTo(Integer other) {  
            ...  
        }  
    }
```

An Interface for Objects That Can Be Compared (cont.)

```
public interface Comparable<T> {  
    public int compareTo(T other);  
}
```

- `item1.compareTo(item2)` should return:
 - a negative integer if `item1` "comes before" `item2`
 - a positive integer if `item1` "comes after" `item2`
 - 0 if `item1` and `item2` are equivalent in the ordering
- These conventions make it easy to construct appropriate method calls:

numeric comparison

`item1 < item2`

`item1 > item2`

`item1 == item2`

comparison using compareTo

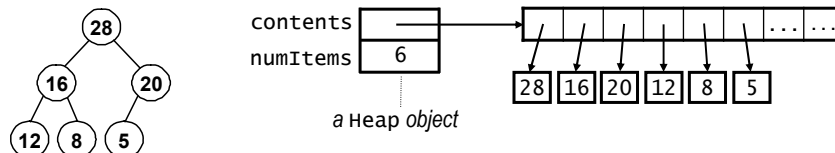
`item1.compareTo(item2) < 0`

`item1.compareTo(item2) > 0`

`item1.compareTo(item2) == 0`

Heap Implementation

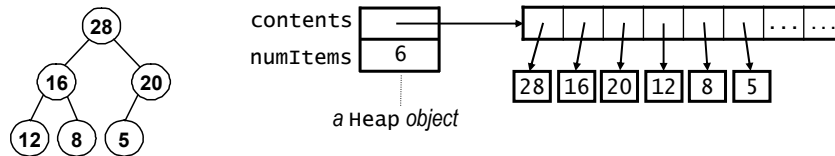
```
public class Heap<T> extends Comparable<T> {  
    private T[] contents;  
    private int numItems;  
  
    public Heap(int maxSize) {  
        contents = (T[])new Comparable[maxSize];  
        numItems = 0;  
    }  
    ...  
}
```



- `Heap` is another example of a generic collection class.
 - as usual, `T` is the type of the elements
 - extends `Comparable<T>` specifies `T` must implement `Comparable<T>`
 - must use `Comparable` (not `Object`) when creating the array

Heap Implementation (cont.)

```
public class Heap<T extends Comparable<T>> {
    private T[] contents;
    private int numItems;
    ...
}
```



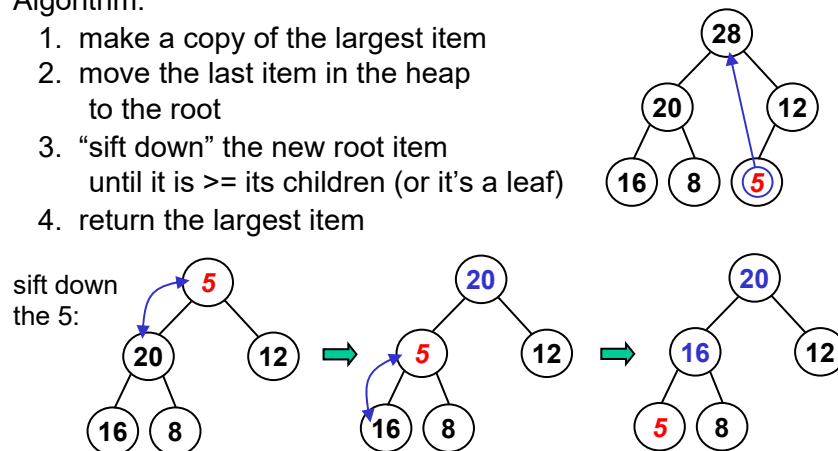
- The picture above is a heap of integers:


```
Heap<Integer> myHeap = new Heap<Integer>(20);
```

 - works because `Integer` implements `Comparable<Integer>`
 - could also use `String` or `Double`

Removing the Largest Item from a Heap

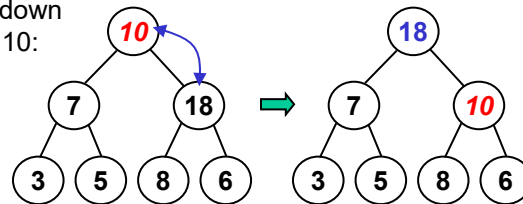
- Remove and return the item in the root node.
- In addition, need to move the largest remaining item to the root, while maintaining a complete tree with each node $\geq$ children
- Algorithm:
 - make a copy of the largest item
 - move the last item in the heap to the root
 - "sift down" the new root item until it is $\geq$ its children (or it's a leaf)
 - return the largest item



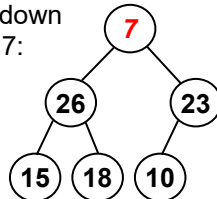
Sifting Down an Item

- To sift down item x (i.e., the item whose key is x):
 - compare x with the larger of the item's children, y
 - if $x < y$, swap x and y and repeat
- Other examples:

sift down
the 10:



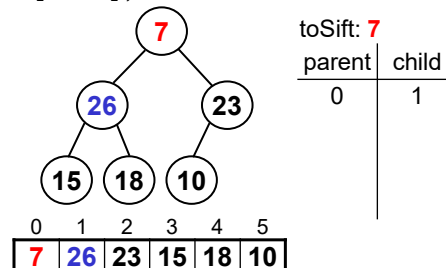
sift down
the 7:



siftDown() Method

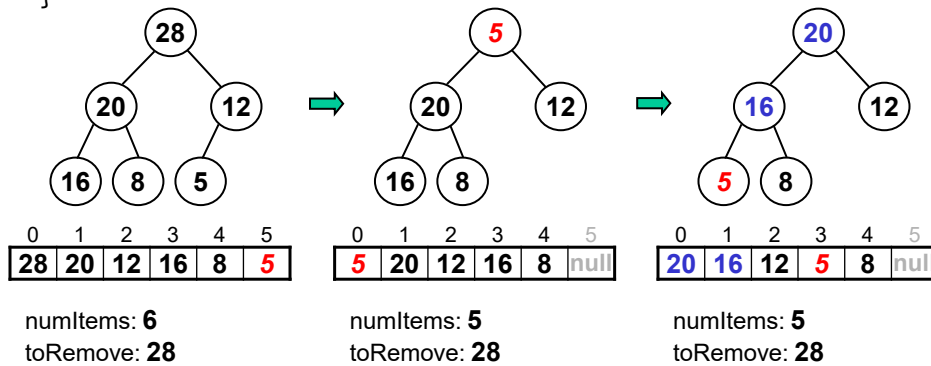
```
private void siftDown(int i) {    // assume i = 0
    T toSift = contents[i];
    int parent = i;
    int child = 2 * parent + 1;
    while (child < numItems) {
        // If the right child is bigger, set child to be its index.
        if (child < numItems - 1 &&
            contents[child].compareTo(contents[child + 1]) < 0) {
            child = child + 1;
        }
        if (toSift.compareTo(contents[child]) >= 0) {
            break; // we're done
        }
        // Move child up and move down one level in the tree.
        contents[parent] = contents[child];
        parent = child;
        child = 2 * parent + 1;
    }
    contents[parent] = toSift;
}
```

- We don't actually swap items. We put the sifted item in place at the end.



remove() Method

```
public T remove() {
    // check for empty heap goes here
    T toRemove = contents[0];
    contents[0] = contents[numItems - 1];
    contents[numItems - 1] = null;
    numItems--;
    siftDown(0);
    return toRemove;
}
```

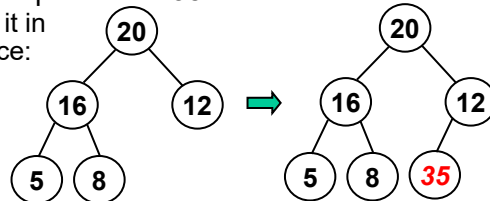


Inserting an Item in a Heap

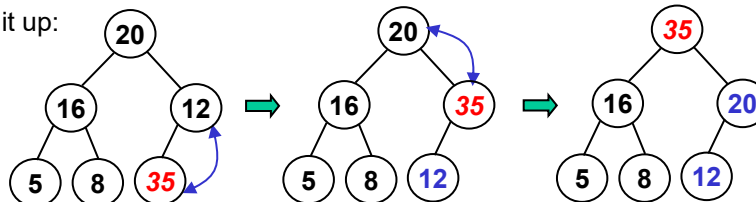
- Algorithm:
 1. put the item in the next available slot (grow array if needed)
 2. "sift up" the new item until it is $\leq$ its parent (or it becomes the root item)

- Example: insert 35

put it in place:

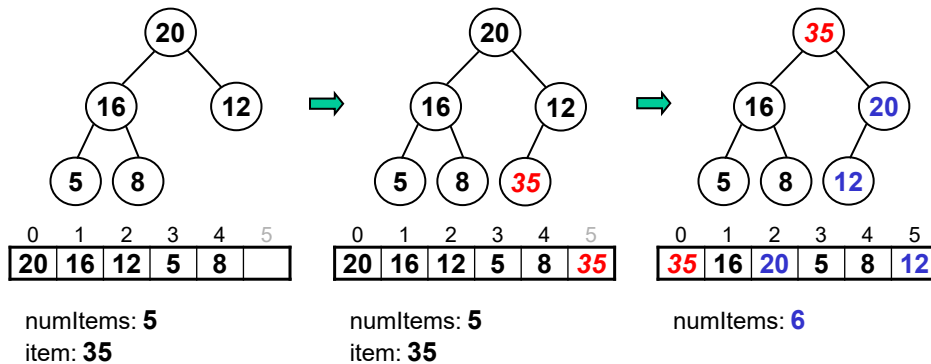


sift it up:

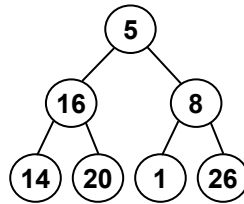


insert() Method

```
public void insert(T item) {
    if (numItems == contents.length) {
        // code to grow the array goes here...
    }
    contents[numItems] = item;
    siftUp(numItems);
    numItems++;
}
```



Time Complexity of a Heap



- A heap containing n items has a height $\leq \log_2 n$. Why?
- Thus, removal and insertion are both $O(\log n)$.
 - remove: go down at most $\log_2 n$ levels when sifting down; do a constant number of operations per level
 - insert: go up at most $\log_2 n$ levels when sifting up; do a constant number of operations per level
- This means we can use a heap for a $O(\log n)$ -time priority queue.

Using a Heap for a Priority Queue

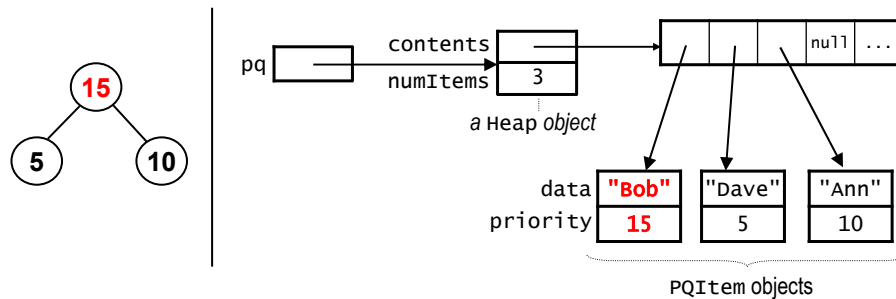
- Recall: a *priority queue* (PQ) is a collection in which each item has an associated number known as a *priority*.
 - ("Ann Cudd", 10), ("Robert Brown", 15), ("Dave Sullivan", 5)
 - use a higher priority for items that are "more important"
- To implement a PQ using a heap:
 - order the items in the heap according to their priorities
 - every item in the heap will have a priority $\geq$ its children
 - the highest priority item will be in the root node
 - get the highest priority item by calling `heap.remove()`!
- For this to work, we need a "wrapper" class for items that we put in the priority queue.
 - will group together an item with its priority
 - with a `compareTo()` method that compares priorities!

A Class for Items in a Priority Queue

```
public class PQItem implements Comparable<PQItem> {  
    // group an arbitrary object with a priority  
    private Object data;  
    private int priority;  
    ...  
  
    public int compareTo(PQItem other) {  
        // error-checking goes here...  
        return (priority - other.priority);  
    }  
}
```

- Example: `PQItem item = new PQItem("Dave Sullivan", 5);`
- Its `compareTo()` compares PQItems based on their priorities.
- `item1.compareTo(item2)` returns:
 - a negative integer if `item1` has a lower priority than `item2`
 - a positive integer if `item1` has a higher priority than `item2`
 - 0 if they have the same priority

Using a Heap for a Priority Queue



- Sample client code:

```
Heap<PQItem> pq = new Heap<PQItem>(50);
pq.insert(new PQItem("Dave", 5));
pq.insert(new PQItem("Ann", 10));
pq.insert(new PQItem("Bob", 15));
```

```
PQItem mostImportant = pq.remove(); // will get Bob!
```

Using a Heap to Sort an Array

- Recall selection sort: it repeatedly finds the smallest remaining element and swaps it into place:

0	1	2	3	4	5	6
5	16	8	14	20	1	26
0	1	2	3	4	5	6
1	16	8	14	20	5	26
0	1	2	3	4	5	6
1	5	8	14	20	16	26

...

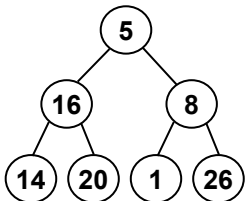
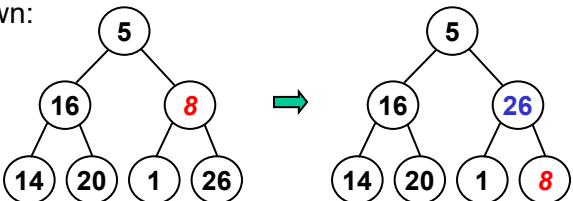
- It isn't efficient, because it performs a linear scan to find the smallest remaining element ($O(n)$ steps per scan).
- Heapsort is a sorting algorithm that repeatedly finds the *largest* remaining element and puts it in place.
- It *is* efficient, because it turns the array into a heap.
 - it can find/remove the largest remaining in $O(\log n)$ steps!

Converting an Arbitrary Array to a Heap

- To convert an array (call it `contents`) with n items to a heap:
 - start with the parent of the last element:
 $\text{contents}[i]$, where $i = ((n - 1) - 1) / 2 = (n - 2) / 2$
 - sift down `contents[i]` and all elements to its left

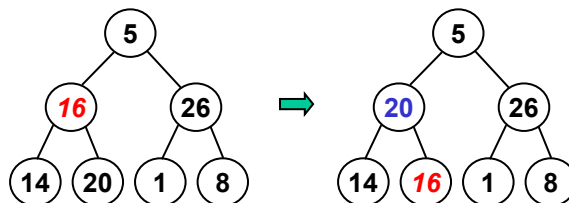
- Example:

0	1	2	3	4	5	6
5	16	8	14	20	1	26

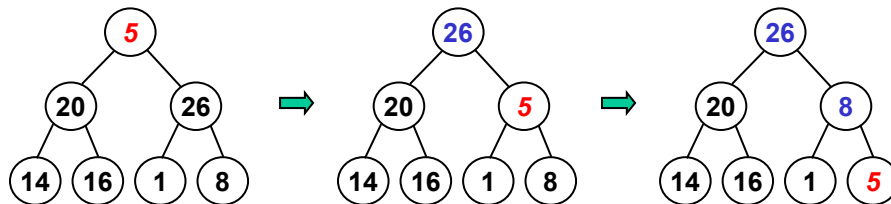

- Last element's parent = $\text{contents}[(7 - 2) / 2] = \text{contents}[2]$.
 Sift it down:
 

Converting an Array to a Heap (cont.)

- Next, sift down `contents[1]`:



- Finally, sift down `contents[0]`:



Heapsort

- Pseudocode:

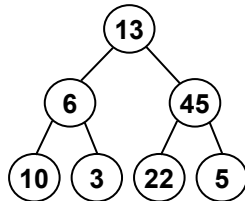
```
heapSort(arr) {  
    // Turn the array into a max-at-top heap.  
    heap = new Heap(arr);  
    endUnsorted = arr.length - 1;  
    while (endUnsorted > 0) {  
        // Get the largest remaining element and put it  
        // at the end of the unsorted portion of the array.  
        largestRemaining = heap.remove();  
        arr[endUnsorted] = largestRemaining;  
        endUnsorted--;  
    }  
}
```

Heapsort Example

- Sort the following array:

0	1	2	3	4	5	6
13	6	45	10	3	22	5

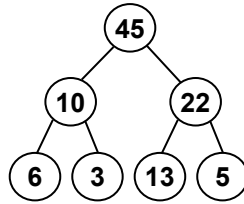
- Here's the corresponding complete tree:



- Begin by converting it to a heap:

Heapsort Example (cont.)

- Here's the heap in both tree and array forms:



0	1	2	3	4	5	6
45	10	22	6	3	13	5

endUnsorted: 6

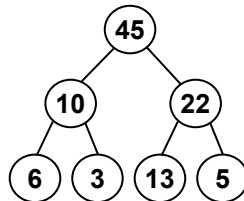
- We begin looping:

```

while (endUnsorted > 0) {
    // Get the largest remaining element and put it
    // at the end of the unsorted portion of the array.
    largestRemaining = heap.remove();
    arr[endUnsorted] = largestRemaining;
    endUnsorted--;
}
  
```

Heapsort Example (cont.)

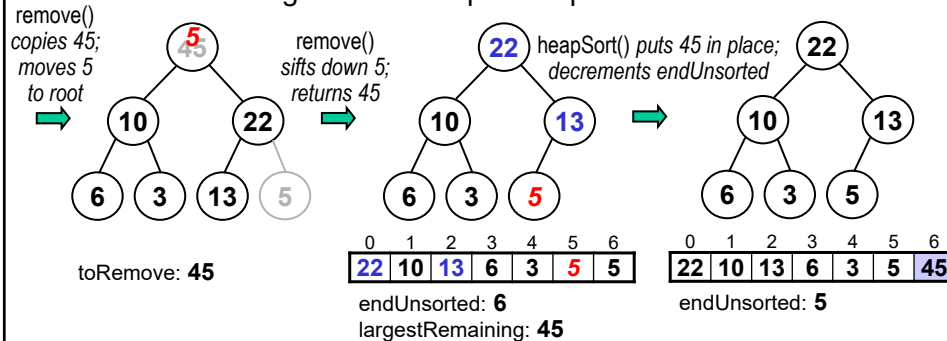
- Here's the heap in both tree and array forms:

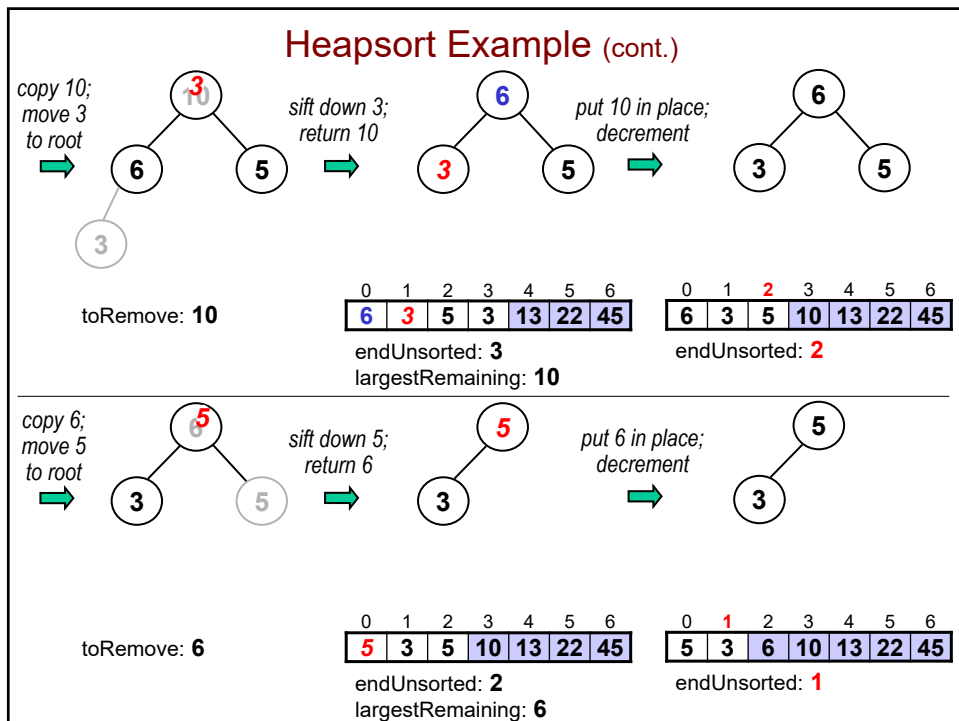
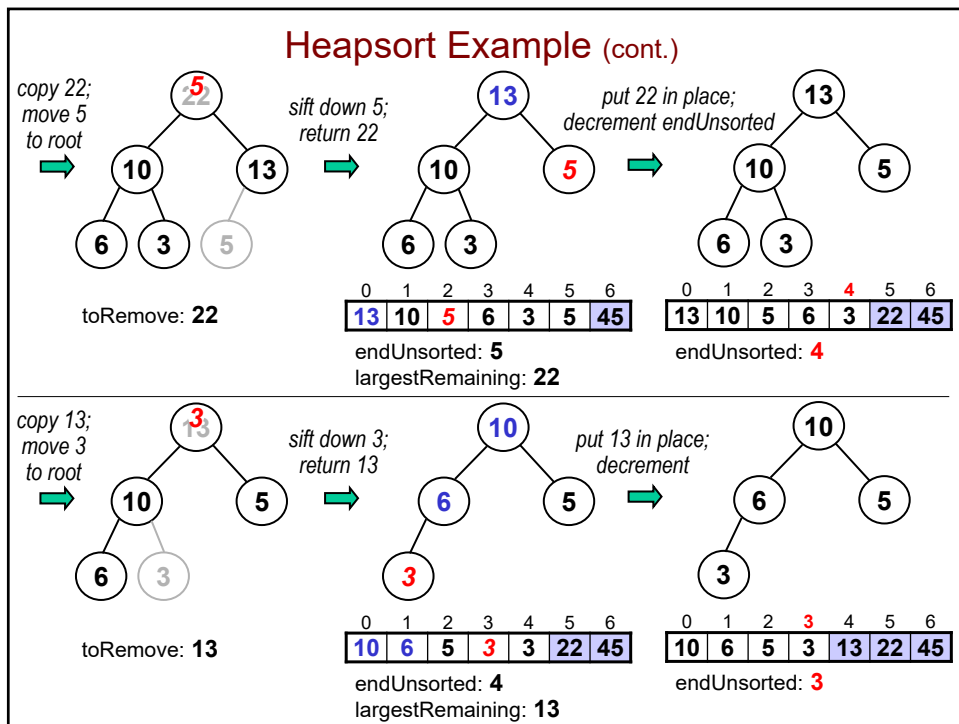


0	1	2	3	4	5	6
45	10	22	6	3	13	5

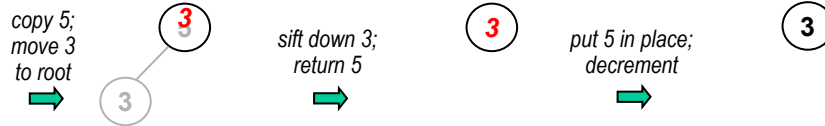
endUnsorted: 6

- Remove the largest item and put it in place:





Heapsort Example (cont.)



toRemove: 5

0	1	2	3	4	5	6
3	3	6	10	13	22	45

endUnsorted: 1
largestRemaining: 5

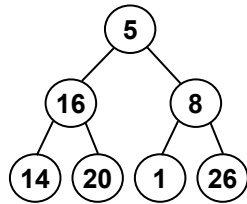
0	1	2	3	4	5	6
3	5	6	10	13	22	45

endUnsorted: 0

- And now we terminate the loop:

```
while (endUnsorted > 0) {
    // Get the largest remaining element and put it
    // at the end of the unsorted portion of the array.
    largestRemaining = heap.remove();
    arr[endUnsorted] = largestRemaining;
    endUnsorted--;
}
```

Efficiency of Heapsort



- Time complexity of going from a heap to a sorted array?
- It can be shown that turning an array into a heap takes $O(n)$ steps.
 - even better than $O(n \log n)$!
 - $n/2$ calls to `siftDown()`, most of which involve small subheaps
- Thus, total time complexity = ?

How Does Heapsort Compare?

algorithm	best case	avg case	worst case	extra memory
selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Shell sort	$O(n \log n)$	$O(n^{1.5})$	$O(n^{1.5})$	$O(1)$
bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$ worst: $O(n)$
mergesort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$

- Heapsort matches mergesort for the best worst-case time complexity, but it has better space complexity.
- Insertion sort is still best for arrays that are almost sorted.
- Quicksort is still typically fastest in the average case.

Hash Tables

Computer Science E-22
Harvard University

David G. Sullivan, Ph.D.

Data Dictionary Revisited

- We've considered several data structures that allow us to store and search for data items using their key fields:

<i>data structure</i>	<i>searching for an item</i>	<i>inserting an item</i>
a list implemented using an array	$O(\log n)$ using binary search	$O(n)$
a list implemented using a linked list	$O(n)$ using linear search	$O(n)$
binary search tree		
balanced search trees (2-3 tree, B-tree, others)		

- We'll now look at hash tables, which can do better than $O(\log n)$.

Ideal Case: Searching = Indexing

- We would achieve optimal efficiency if we could treat the key as an index into an array.
- Example: storing data about members of a sports team
 - key = jersey number (some value from 0-99).
 - class for an individual player's record:

```
public class Player {  
    private int jerseyNum;  
    private String firstName;  
    ...  
}
```
 - store the player records in an array:

```
Player[] teamRecords = new Player[100];
```
- In such cases, search and insertion are $O(1)$:

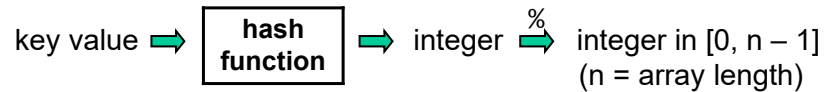
```
public Player search(int jerseyNum) {  
    return teamRecords[jerseyNum];  
}
```

Hashing: Turning Keys into Array Indices

- In most real-world problems, indexing is not as simple as the sports-team example. Why?
 -
 -
 -
- To handle these problems, we perform *hashing*:
 - use a *hash function* to convert the keys into array indices
"sullivan" $\rightarrow$ 18
 - use techniques to handle cases in which multiple keys are assigned the same hash value
- The resulting data structure is known as a *hash table*.

Hash Functions

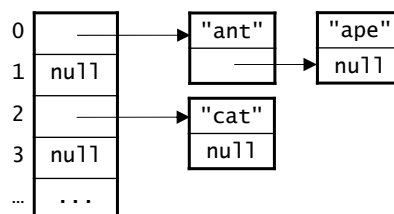
- A hash function defines a mapping from keys to integers.
- We then use the modulus operator to get a valid array index.



- Here's a very simple hash function for keys of lower-case letters:
 $h(\text{key}) = \text{ASCII value of first char} - \text{ASCII value of 'a'}$
 - examples:
 $h(\text{"ant"}) = \text{ASCII for 'a'} - \text{ASCII for 'a'} = 0$
 $h(\text{"cat"}) = \text{ASCII for 'c'} - \text{ASCII for 'a'} = 2$
- $h(\text{key})$ is known as the key's *hash code*.
- A *collision* occurs when items with different keys are assigned the same hash code.

Dealing with Collisions I: Separate Chaining

- Each position in the hash table serves as a *bucket* that can store multiple data items.
- Two options:
 1. each bucket is itself an array
 - need to preallocate, and a bucket may become full
 2. each bucket is a linked list
 - items with the same hash code are "chained" together
 - each "chain" can grow as needed



Dealing with Collisions II: Open Addressing

- When the position assigned by the hash function is occupied, find another open position.
- Example: "wasp" has a hash code of 22, but it ends up in position 23 because position 22 is occupied.
- We'll consider three ways of finding an open position – a process known as *probing*.
- We also perform probing when searching.
 - example: search for "wasp"
 - look in position 22
 - then look in position 23
 - need to figure out when to safely stop searching (more on this soon!)

0	"ant"
1	
2	"cat"
3	
4	"emu"
5	
6	
7	
...	...
22	"wolf"
23	"wasp"
24	"yak"
25	"zebra"

Linear Probing

- Probe sequence: $h(\text{key})$, $h(\text{key}) + 1$, $h(\text{key}) + 2$, ..., wrapping around as necessary.
- Examples:
 - "ape" ($h = 0$) would be placed in position 1, because position 0 is already full.
 - "bear" ($h = 1$): try 1, $1 + 1$, $1 + 2$ – open!
 - where would "zebu" end up?
- Advantage: if there is an open cell, linear probing will eventually find it.
- Disadvantage: get "clusters" of occupied cells that lead to longer subsequent probes.
 - probe length = the number of positions considered during a probe

0	"ant"
1	"ape"
2	"cat"
3	"bear"
4	"emu"
5	
6	
7	
...	...
22	"wolf"
23	"wasp"
24	"yak"
25	"zebra"

Quadratic Probing

- Probe sequence: $h(\text{key})$, $h(\text{key}) + 1^2$, $h(\text{key}) + 2^2$, $h(\text{key}) + 3^2$, ..., wrapping around as necessary.
- Examples:
 - "ape" ($h = 0$): try 0, 0 + 1 – open!
 - "bear" ($h = 1$): try 1, 1 + 1, 1 + 4 – open!
 - "zebu"?
- Advantage: smaller clusters of occupied cells
- Disadvantage: may fail to find an existing open position. For example:

table size = 10

x = occupied

trying to insert a
key with $h(\text{key}) = 0$

offsets of the probe
sequence in italics

0	x		5	x	<i>25</i>
1	x	<i>1 81</i>	6	x	<i>16 36</i>
2			7		
3			8		
4	x	<i>4 64</i>	9	x	<i>9 49</i>

0	"ant"
1	"ape"
2	"cat"
3	
4	"emu"
5	"bear"
6	
7	
...	...
22	"wolf"
23	"wasp"
24	"yak"
25	"zebra"

Double Hashing

- Use two hash functions:
 - h_1 computes the hash code
 - h_2 computes the increment for probing
 - probe sequence: h_1 , $h_1 + h_2$, $h_1 + 2 \cdot h_2$, ...
- Examples:
 - $h_1 = \text{our previous } h$
 - $h_2 = \text{number of characters in the string}$
 - "ape" ($h_1 = 0$, $h_2 = 3$): try 0, 0 + 3 – open!
 - "bear" ($h_1 = 1$, $h_2 = 4$): try 1 – open!
 - "zebu"?
- Combines good features of linear and quadratic:
 - reduces clustering
 - will find an open position if there is one, provided the table size is a prime number

0	"ant"
1	"bear"
2	"cat"
3	"ape"
4	"emu"
5	
6	
7	
...	...
22	"wolf"
23	"wasp"
24	"yak"
25	"zebra"

Removing Items Under Open Addressing

- Problematic example (using linear probing):
 - insert "ape" ($h = 0$): try 0, 0 + 1 – open!
 - insert "bear" ($h = 1$): try 1, 1 + 1, 1 + 2 – open!
 - remove "ape"
 - search for "ape": try 0, 0 + 1 – conclude not in table
 - search for "bear": **try 1 – conclude not in table, but "bear" is further down in the table!**

0	"ant"
1	
2	"cat"
3	"bear"
4	"emu"
5	
...	...
22	"wolf"
23	"wasp"
24	"yak"
25	"zebra"

- To fix this problem, distinguish between:
 - removed positions* that previously held an item
 - empty positions* that have never held an item
- During probing, we *don't* stop if we see a removed position.
ex: search for "bear": try 1 (removed), 1 + 1, 1 + 2 – found!
- We can insert items in either empty or removed positions.

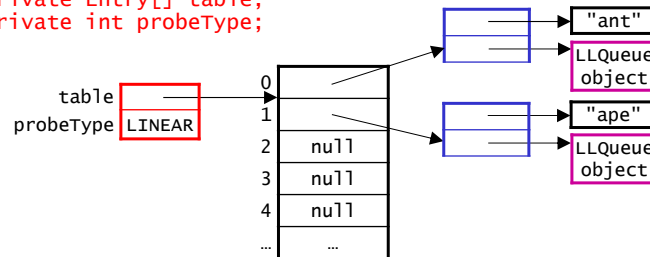
An Interface For Hash Tables

```
public interface HashTable {
    boolean insert(Object key, Object value);
    Queue<Object> search(Object key);
    Queue<Object> remove(Object key);
}
```

- `insert()` takes a key-value pair and returns:
 - `true` if the key-value pair can be added
 - `false` if it cannot be added (referred to as *overflow*)
- `search()` and `remove()` both take a key, and return a queue containing all of the values associated with that key.
 - example: an index for a book
 - key = word
 - values = the pages on which that word appears
 - return `null` if the key is not found

An Implementation Using Open Addressing

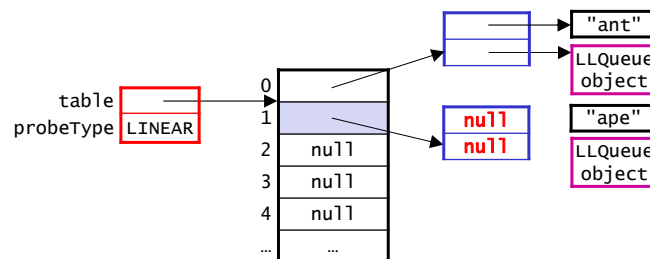
```
public class OpenHashTable implements HashTable {
    private class Entry {
        private Object key;
        private LLQueue<Object> values;
    }
    ...
    private Entry[] table;
    private int probeType;
}
```



- We use a private inner class for the entries in the hash table.
- We use an LLQueue for the values associated with a given key.

Empty vs. Removed

- When we remove a key and its values, we:
 - leave the Entry object in the table
 - set the Entry object's key and values fields to null
 - example: after remove("ape"):



- Note the difference:
 - a truly empty position has a value of null in the table (example: positions 2, 3 and 4 above)
 - a removed position refers to an Entry object whose key and values fields are null (example: position 1 above)

Probing Using Double Hashing

```
private int probe(Object key) {
    int i = h1(key);    // first hash function
    int h2 = h2(key);   // second hash function

    // keep probing until we get an empty position or match
    while (table[i] != null && !key.equals(table[i].key)) {
        i = (i + h2) % table.length;
    }

    return i;
}
```

- It is essential that we:
 - check for `table[i] != null` first. why?
 - call the `equals` method on `key`, not `table[i].key`. why?

Avoiding an Infinite Loop

- The while loop in our probe method could lead to an infinite loop.

```
while (table[i] != null && !key.equals(table[i].key)) {
    i = (i + h2) % table.length;
}
```

- When would this happen?
- We can stop probing after checking n positions (n = table size), because the probe sequence will just repeat after that point.
 - for quadratic probing:
$$(h1 + n^2) \% n = h1 \% n$$
$$(h1 + (n+1)^2) \% n = (h1 + n^2 + 2n + 1) \% n = (h1 + 1) \% n$$
 - for double hashing:
$$(h1 + n*h2) \% n = h1 \% n$$
$$(h1 + (n+1)*h2) \% n = (h1 + n*h2 + h2) \% n = (h1 + h2) \% n$$

Avoiding an Infinite Loop (cont.)

```
private int probe(Object key) {
    int i = h1(key);    // first hash function
    int h2 = h2(key);   // second hash function
    int numChecked = 1;

    // keep probing until we get an empty position or a match
    while (table[i] != null && !key.equals(table[i].key)) {
        if (numChecked == table.length) {
            return -1;
        }
        i = (i + h2) % table.length;
        numChecked++;
    }

    return i;
}
```

Search and Removal

```
public LLQueue<Object> search(Object key) {
    // throw an exception if key == null
    int i = probe(key);
    if (i == -1 || table[i] == null) {
        return null;
    } else {
        return table[i].values;
    }
}

public LLQueue<Object> remove(Object key) {
    // throw an exception if key == null
    int i = probe(key);
    if (i == -1 || table[i] == null) {
        return null;
    }

    LLQueue<Object> removedVals = table[i].values;
    table[i].key = null;
    table[i].values = null;
    return removedVals;
}
```

Insertion

- We begin by probing for the key.
- Several cases:
 1. the key is already in the table (we're inserting a duplicate)
→ add the value to the values in the key's Entry
 2. the key is not in the table: three subcases:
 - a. encountered 1 or more removed positions while probing
→ put the (key, value) pair in the *first* removed position seen during probing. why?
 - b. no removed position; reached an empty position
→ put the (key, value) pair in the empty position
 - c. no removed position or empty position
→ overflow; return false

Tracing Through Some Examples

- Start with the hash table at right with:
 - double hashing
 - our earlier hash functions h1 and h2
- Perform the following operations:
 - insert "bear" (h1 = 1, h2 = 4):
 - insert "bison" (h1 = 1, h2 = 5):
 - insert "cow" (h1 = 2, h2 = 3):
 - delete "emu" (h1 = 4, h2 = 3):
 - search "eel" (h1 = 4, h2 = 3):
 - insert "bee" (h1 = __, h2 = __):

0	"ant"
1	
2	"cat"
3	
4	"emu"
5	"fox"
6	
7	
8	
9	
10	

Dealing with Overflow

- Overflow = can't find a position for an item
- When does it occur?
 - linear probing:
 - quadratic probing:
 -
 -
 - double hashing:
 - if the table size is a prime number: same as linear
 - if the table size is not a prime number: same as quadratic
- To avoid overflow (and reduce search times), grow the hash table when the % of occupied positions gets too big.
 - problem: we need to rehash **all** of the existing items. why?

Implementing the Hash Function

- Characteristics of a good hash function:
 - 1) efficient to compute
 - 2) uses the entire key
 - changing any char/digit/etc. should change the hash code
 - 3) distributes the keys more or less uniformly across the table
 - 4) must be a function!
 - a key must always get the same hash code
- In Java, every object has a hashCode() method.
 - the version inherited from Object returns a value based on an object's memory location
 - classes can override this version with their own

Hash Functions for Strings: version 1

- h_a = the sum of the characters' ASCII values
 - example: $h_a(\text{"eat"}) = 101 + 97 + 116 = 314$
- All permutations of a given set of characters get the same code.
 - example: $h_a(\text{"tea"}) = h_a(\text{"eat"})$
 - could be useful in a Scrabble game
 - allow you to look up all words that can be formed from a given set of characters
- The range of possible hash codes is very limited.
 - example: hashing keys composed of 1-5 lower-case char's (padded with spaces)
 - $26 \times 27 \times 27 \times 27 \times 27 = \text{over 13 million possible keys}$
 - smallest code = $h_a(\text{"a "}) = 97 + 4 \times 32 = 225$
largest code = $h_a(\text{"zzzzz"}) = 5 \times 122 = 610$ } $610 - 225 = 385 \text{ codes}$

Hash Functions for Strings: version 2

- Compute a *weighted* sum of the ASCII values:
$$h_b = a_0b^{n-1} + a_1b^{n-2} + \dots + a_{n-2}b + a_{n-1}$$
where a_i = ASCII value of the i th character
 b = a constant
 n = the number of characters
- Multiplying by powers of b allows the *positions* of the characters to affect the hash code.
 - different permutations get different codes
- We may get arithmetic overflow, and thus the code may be negative. We adjust it when this happens.
- Java uses this hash function with $b = 31$ in the `hashCode()` method of the `String` class.

Hash Table Efficiency

- In the best case, search and insertion are $O(1)$.
- In the worst case, search and insertion are linear.
 - open addressing: $O(m)$, where m = the size of the hash table
 - separate chaining: $O(n)$, where n = the number of keys
- With good choices of hash function and table size, complexity is generally better than $O(\log n)$ and approaches $O(1)$.
- *load factor* = # keys in table / size of the table.
To prevent performance degradation:
 - open addressing: try to keep the load factor $< 1/2$
 - separate chaining: try to keep the load factor < 1
- Time-space tradeoff: bigger tables have better performance, but they use up more memory.

Hash Table Limitations

- It can be hard to come up with a good hash function for a particular data set.
- The items are not ordered by key. As a result, we can't easily:
 - print the contents in sorted order
 - perform a range search (find all values between v_1 and v_2)
 - perform a rank search – get the k th largest itemWe *can* do all of these things with a search tree.

Extra Practice

- Start with the hash table at right with:
 - double hashing
 - $h_1(\text{key}) = \text{ASCII of first letter} - \text{ASCII of 'a'}$
 - $h_2(\text{key}) = \text{key.length}()$
 - shaded cells are removed cells
- What is the probe sequence for "baboon"?**
(the sequence of positions seen during probing)

- A. 1, 2, 5
- B. 1, 6
- C. 1, 7, 2
- D. 1, 7, 3
- E. 1, 7, 2, 8

0	"ant"
1	
2	"cat"
3	
4	"emu"
5	
6	
7	
8	
9	
10	

Extra Practice

- Start with the hash table at right with:
 - double hashing
 - $h1(\text{key}) = \text{ASCII of first letter} - \text{ASCII of 'a'}$
 - $h2(\text{key}) = \text{key.length}()$
 - shaded cells are removed cells

- What is the probe sequence for "baboon"?

($h1 = 1, h2 = 6$) try: $1 \% 11 = 1$

$$(1 + 6) \% 11 = 7$$

$$(1 + 2*6) \% 11 = 2$$

$$(1 + 3*6) \% 11 = 8$$

empty cell, so stop probing

- A. 1, 2, 5
- B. 1, 6
- C. 1, 7, 2
- D. 1, 7, 3
- E. **1, 7, 2, 8**

0	"ant"
1	
2	"cat"
3	
4	"emu"
5	
6	
7	
8	
9	
10	

Extra Practice

- Start with the hash table at right with:
 - double hashing
 - $h1(\text{key}) = \text{ASCII of first letter} - \text{ASCII of 'a'}$
 - $h2(\text{key}) = \text{key.length}()$
 - shaded cells are removed cells

- What is the probe sequence for "baboon"?

($h1 = 1, h2 = 6$) try: $1 \% 11 = 1$

$$(1 + 6) \% 11 = 7$$

$$(1 + 2*6) \% 11 = 2$$

$$(1 + 3*6) \% 11 = 8$$

- If we insert "baboon", in what position will it go?

- A. 1
- B. 7
- C. 2
- D. 8

0	"ant"
1	
2	"cat"
3	
4	"emu"
5	
6	
7	
8	
9	
10	

Extra Practice

- Start with the hash table at right with:
 - double hashing
 - $h1(\text{key}) = \text{ASCII of first letter} - \text{ASCII of 'a'}$
 - $h2(\text{key}) = \text{key.length}()$
 - shaded cells are removed cells

0	"ant"
1	"baboon"
2	"cat"
3	
4	"emu"
5	
6	
7	
8	
9	
10	

- What is the probe sequence for "baboon"?

$(h1 = 1, h2 = 6)$ try: $1 \% 11 = 1$
 $(1 + 6) \% 11 = 7$
 $(1 + 2*6) \% 11 = 2$
 $(1 + 3*6) \% 11 = 8$

- If we insert "baboon", in what position will it go?

A. **1** B. 7 C. 2 D. 8

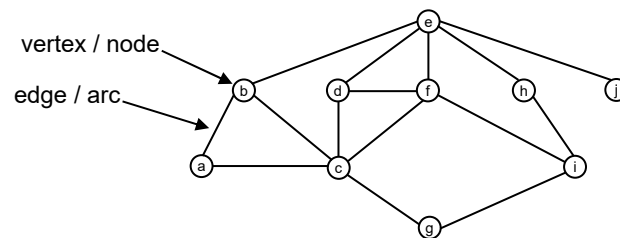

 the first *removed* position seen while probing

Graphs

Computer Science E-22
Harvard Extension School

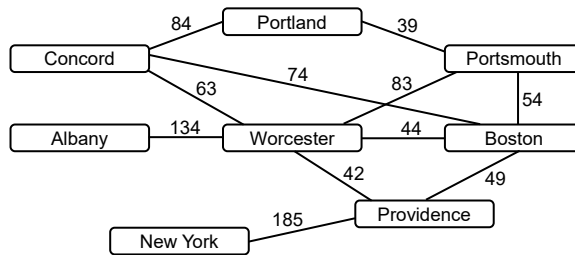
David G. Sullivan, Ph.D.

What is a Graph?



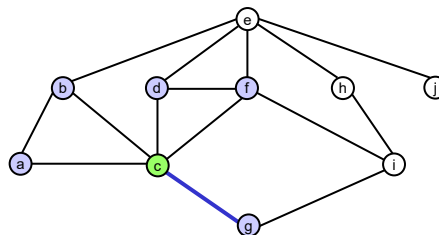
- A graph consists of:
 - a set of *vertices* (also known as *nodes*)
 - a set of *edges* (also known as *arcs*), each of which connects a pair of vertices

Example: A Highway Graph



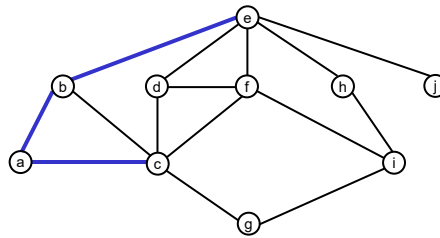
- Vertices represent cities.
- Edges represent highways.
- This is a *weighted* graph, with a *cost* associated with each edge.
 - in this example, the costs denote mileage
- We'll use graph algorithms to answer questions like "What is the shortest route from Portland to Providence?"

Relationships Among Vertices

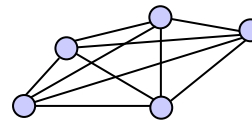
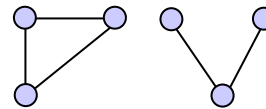


- Two vertices are *adjacent* if they are connected by a single edge.
 - ex: c and g are adjacent, but c and i are not
- The collection of vertices that are adjacent to a vertex v are referred to as v's *neighbors*.
 - ex: c's neighbors are a, b, d, f, and g

Paths in a Graph

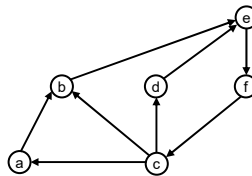


- A *path* is a sequence of edges that connects two vertices.
- A graph is *connected* if there is a path between any two vertices.
 - ex: the six vertices at right are part of a graph that is *not* connected
- A graph is *complete* if there is an edge between every pair of vertices.
 - ex: the graph at right *is* complete



Directed Graphs

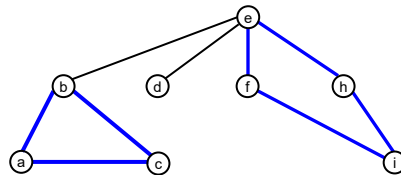
- A *directed* graph has a direction associated with each edge, which is depicted using an arrow:



- Edges in a directed graph are often represented as ordered pairs of the form (start vertex, end vertex).
 - ex: (a, b) is an edge in the graph above, but (b, a) is not.
- In a path in a directed graph, the end vertex of edge i must be the same as the start vertex of edge $i + 1$.
 - ex: $\{(a, b), (b, e), (e, f)\}$ is a valid path.
 $\{(a, b), (c, b), (c, a)\}$ is not.

Cycles in a Graph

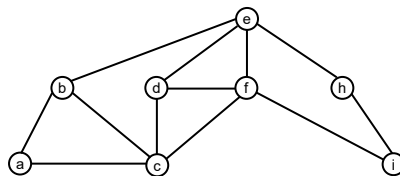
- A *cycle* is a path that:
 - leaves a given vertex using one edge
 - returns to that same vertex using a different edge
- Examples: the highlighted paths below



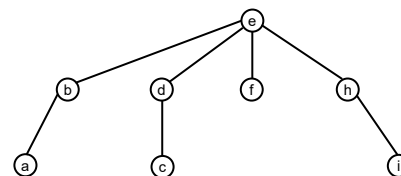
- An *acyclic* graph has no cycles.

Trees vs. Graphs

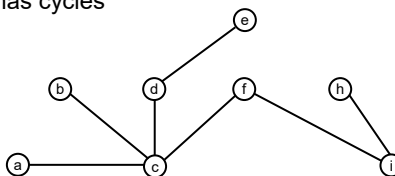
- A tree is a special type of graph.
 - connected, undirected, and acyclic
 - we usually single out one of the vertices to be the root, but graph theory does *not* require this



a graph that is *not* a tree,
because it has cycles



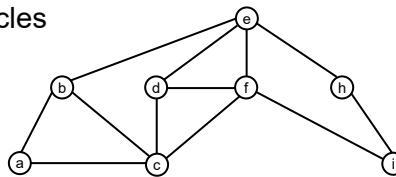
a tree using the same nodes



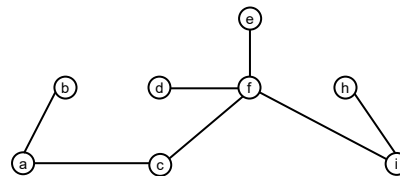
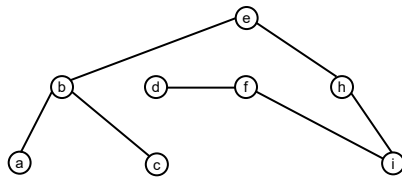
another tree using the same nodes

Spanning Trees

- A spanning tree is a subset of a connected graph that contains:
 - all of the vertices
 - a subset of the edges that form a tree
- Recall this graph with cycles from the previous slide:



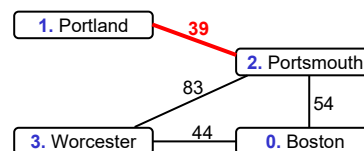
- The trees on that slide were spanning trees for this graph. Here are two others:



Representing a Graph: Option 1

- Use an *adjacency matrix* – a two-dimensional array in which element $[r][c]$ = the cost of going from vertex r to vertex c
- Example:

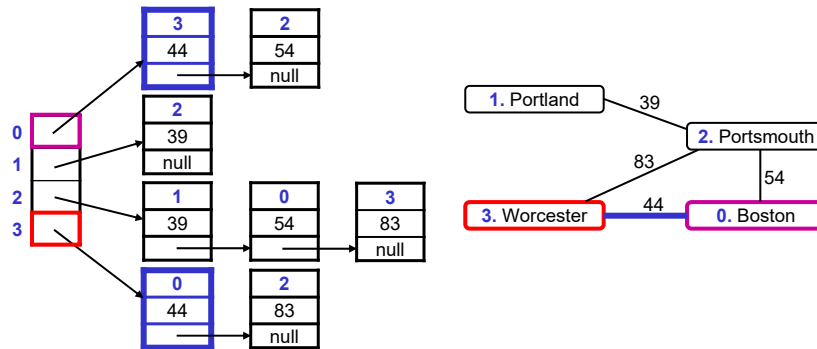
	0	1	2	3
0			54	44
1			39	
2	54	39		83
3	44		83	



- Use a special value to indicate there's no edge from r to c
 - shown as a shaded cell above
 - can't use 0, because an edge may have an actual cost of 0
- This representation:
 - wastes memory if a graph is *sparse* (few edges per vertex)
 - is memory-efficient if a graph is *dense* (many edges per vertex)

Representing a Graph: Option 2

- Use one *adjacency list* for each vertex.
 - a linked list with info on the edges coming from that vertex



- This representation uses less memory if a graph is sparse.
- It uses more memory if a graph is dense.
 - because of the references linking the nodes

Graph Class

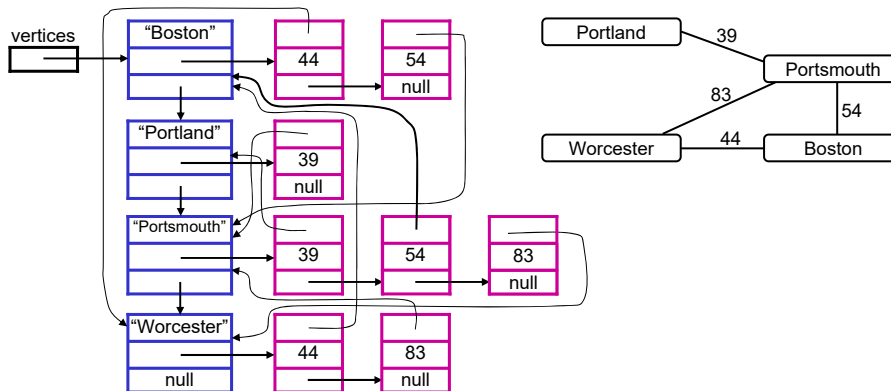
```
public class Graph {
    private class Vertex {
        private String id;
        private Edge edges;           // adjacency list
        private Vertex next;
        private boolean encountered;
        private boolean done;
        private Vertex parent;
        private double cost;
        ...
    }

    private class Edge {
        private Vertex start;
        private Vertex end;
        private double cost;
        private Edge next;
        ...
    }

    private Vertex vertices;
    ...
}
```

The highlighted fields
are shown in the diagram
on the previous page.

Our Graph Representation



- Each Vertex object (shown in blue) stores info. about a vertex.
 - including an adjacency list of Edge objects (the purple ones)
- A Graph object has a single field called `vertices`
 - a reference to a linked list of Vertex objects
 - a linked list of linked lists!

Traversing a Graph

- Traversing a graph involves starting at some vertex and visiting all vertices that can be reached from that vertex.
 - visiting a vertex = processing its data in some way
 - if the graph is connected, all of its vertices will be visited
- We will consider two types of traversals:
 - **depth-first**: proceed as far as possible along a given path before backing up
 - **breadth-first**: visit a vertex
visit all of its neighbors
visit all unvisited vertices 2 edges away
visit all unvisited vertices 3 edges away, etc.

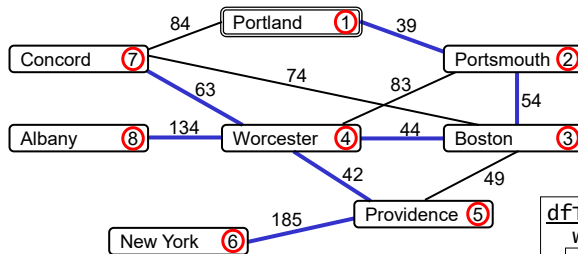
Depth-First Traversal

- Visit a vertex, then make recursive calls on all of its yet-to-be-visited neighbors:

```
private static void dfTrav(Vertex v, Vertex parent) {
    System.out.println(v.id); // visit v
    v.done = true;
    v.parent = parent;      // record where we came from

    // walk down v's adjacency list
    Edge e = v.edges;
    while (e != null) {
        Vertex w = e.end;    // consider each neighbor w
        if (!w.done) {      // if w has not been visited
            dfTrav(w, v);
        }
        e = e.next;
    }
}
```

Example: Depth-First Traversal from Portland

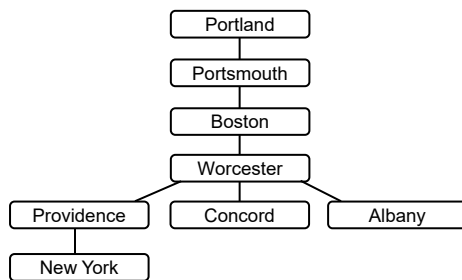
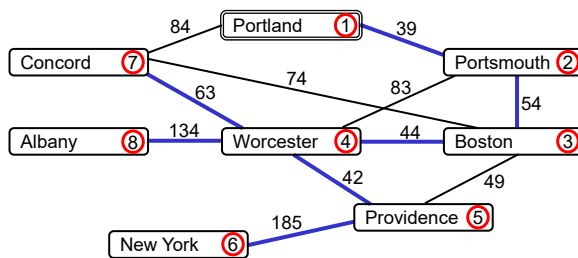


For the examples, we'll assume that the edges in each vertex's adjacency list are sorted by increasing edge cost.

```
void dfTrav(Vertex v, Vertex parent) {
    System.out.println(v.id);
    v.done = true;
    v.parent = parent;
    Edge e = v.edges;
    while (e != null) {
        Vertex w = e.end;
        if (!w.done) {
            dfTrav(w, v);
        }
        e = e.next;
    }
}
```

```
dfTrav(Ptl, null)
w = Pts
dfTrav(Pts, Ptl)
w = Ptl, Bos
dfTrav(Bos, Pts)
w = Wor
dfTrav(Wor, Bos)
w = Pro
dfTrav(Pro, wor)
w = Wor, Bos, NY
dfTrav(NY, Pro)
w = Pro
return
no more neighbors
return
w = Bos, Con
dfTrav(Con, wor)
...
```

Depth-First Spanning Tree

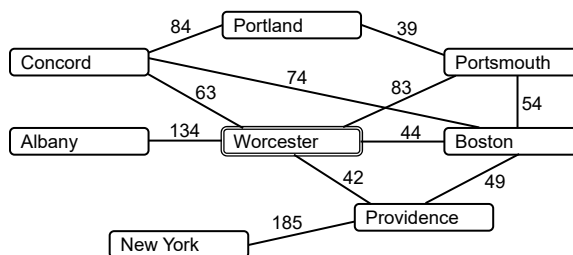


The edges obtained by following the parent references form a spanning tree with the origin of the traversal as its root.

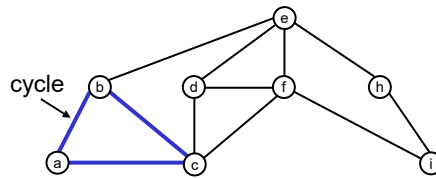
From any city, we can get to the origin by following the roads in the spanning tree.

Another Example: Depth-First Traversal from Worcester

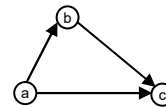
- In what order will the cities be visited?
- Which edges will be in the resulting spanning tree?



Checking for Cycles in an Undirected Graph



- To discover a cycle in an undirected graph, we can:
 - perform a depth-first traversal, marking the vertices as visited
 - if a visited vertex has a neighbor that is (1) not its parent, and (2) already marked as visited, there must be a cycle
- If no cycles found during the traversal, the graph is acyclic.
- This doesn't work for directed graphs:
 - c is a neighbor of both a and b
 - there is no cycle

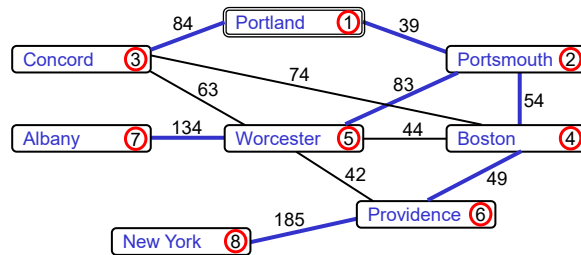


Breadth-First Traversal

- Use a queue to store vertices we've seen but not yet visited:

```
private static void bfTrav(Vertex origin) {
    origin.encountered = true;
    origin.parent = null;
    Queue<Vertex> q = new LLQueue<Vertex>();
    q.insert(origin);
    while (!q.isEmpty()) {
        Vertex v = q.remove();
        System.out.println(v.id);           // visit v.
        // Add v's unencountered neighbors to the queue.
        Edge e = v.edges;
        while (e != null) {
            Vertex w = e.end;
            if (!w.encountered) {
                w.encountered = true;
                w.parent = v;
                q.insert(w);
            }
            e = e.next;
        }
    }
}
```

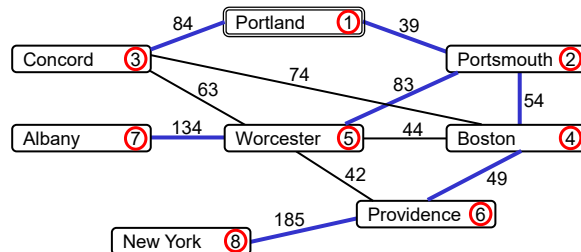
Example: Breadth-First Traversal from Portland



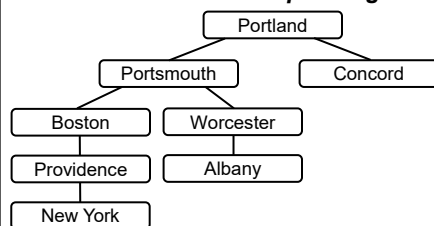
Evolution of the queue:

<u>remove</u>	<u>insert</u>	<u>queue contents</u>
	Portland	Portland
Portland	Portsmouth, Concord	Portsmouth, Concord
Portsmouth	Boston, Worcester	Concord, Boston, Worcester
Concord	<i>none</i>	Boston, Worcester
Boston	Providence	Worcester, Providence
Worcester	Albany	Providence, Albany
Providence	New York	Albany, New York
Albany	<i>none</i>	New York
New York	<i>none</i>	<i>empty</i>

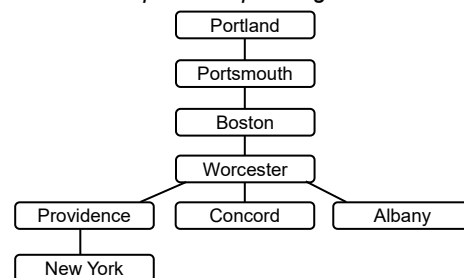
Breadth-First Spanning Tree



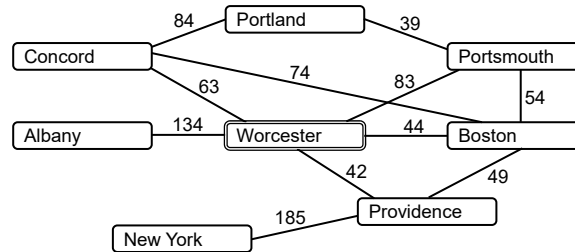
breadth-first spanning tree:



depth-first spanning tree:



Another Example: Breadth-First Traversal from Worcester



Evolution of the queue:

remove

insert

queue contents

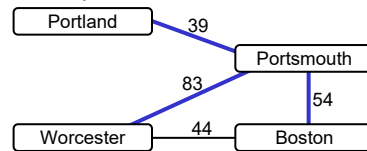
Time Complexity of Graph Traversals

- let V = number of vertices in the graph
 E = number of edges
- If we use an adjacency matrix, a traversal requires $O(V^2)$ steps.
 - why?
- If we use adjacency lists, a traversal requires $O(V + E)$ steps.
 - visit each vertex once
 - traverse each vertex's adjacency list at most once
 - the total length of the adjacency lists is at most $2E = O(E)$
 - for a sparse graph, $O(V + E)$ is better than $O(V^2)$
 - for a dense graph, $E = O(V^2)$, so both representations are $O(V^2)$
- In the remaining notes, we'll assume an adjacency-list implementation.

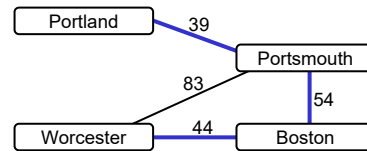
Minimum Spanning Tree

- A *minimum spanning tree* (MST) has the smallest total cost among all possible spanning trees.

- example:*



one possible spanning tree
(total cost = $39 + 83 + 54 = 176$)

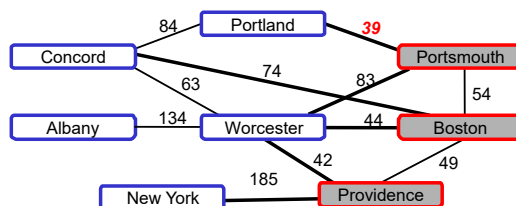


the minimal-cost spanning tree
(total cost = $39 + 54 + 44 = 137$)

- If all edges have unique costs, there is only one MST.
If some edges have the same cost, there may be more than one.
- Example applications:
 - determining the shortest highway system for a set of cities
 - calculating the smallest length of cable needed to connect a network of computers

Building a Minimum Spanning Tree

- Claim:** If you divide the vertices into two disjoint subsets A and B, the lowest-cost edge (v_a, v_b) joining a vertex in A to a vertex in B must be part of the MST.



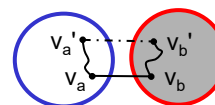
example:
subset A = unshaded
subset B = shaded

The 6 bold edges each join a vertex in A to a vertex in B.

The one with the lowest cost (Portland to Portsmouth) must be in the MST.

Proof by contradiction:

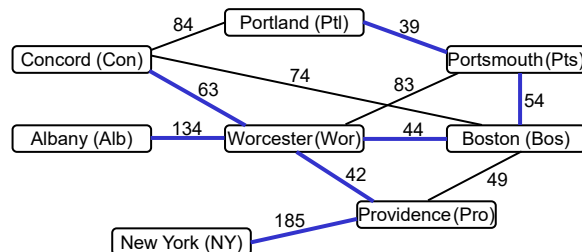
- Assume we can create an MST (call it T) that doesn't include (v_a, v_b) .
- T must include a path from v_a to v_b , so it must include one of the other edges (v_a', v_b') that span A and B, such that (v_a', v_b') is part of the path from v_a to v_b .
- Adding (v_a, v_b) to T introduces a cycle.
- Removing (v_a', v_b') gives a spanning tree with a lower total cost, which contradicts the original assumption.



Prim's MST Algorithm

- Begin with the following subsets:
 - A = any one of the vertices
 - B = all of the other vertices
- Repeatedly do the following:
 - select the lowest-cost edge (v_a, v_b) connecting a vertex in A to a vertex in B
 - add (v_a, v_b) to the spanning tree
 - move vertex v_b from set B to set A
- Continue until set A contains all of the vertices.

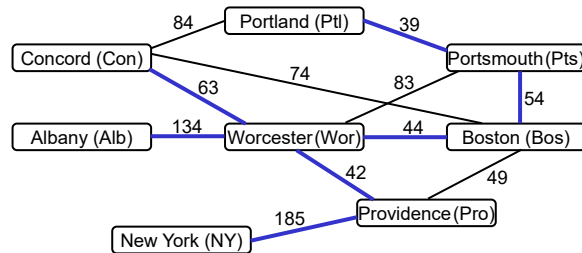
Example: Prim's Starting from Concord



- Tracing the algorithm:

<u>edge added</u>	<u>set A</u>	<u>set B</u>
	{Con}	{Alb, Bos, NY, Ptl, Pts, Pro, Wor}
(Con, Wor)	{Con, Wor}	{Alb, Bos, NY, Ptl, Pts, Pro}
(Wor, Pro)	{Con, Wor, Pro}	{Alb, Bos, NY, Ptl, Pts}
(Wor, Bos)	{Con, Wor, Pro, Bos}	{Alb, NY, Ptl, Pts}
(Bos, Pts)	{Con, Wor, Pro, Bos, Pts}	{Alb, NY, Ptl}
(Pts, Ptl)	{Con, Wor, Pro, Bos, Pts, Ptl}	{Alb, NY}
(Wor, Alb)	{Con, Wor, Pro, Bos, Pts, Ptl, Alb}	{NY}
(Pro, NY)	{Con, Wor, Pro, Bos, Pts, Ptl, Alb, NY}	{}

MST May Not Give Shortest Paths



- The MST is the spanning tree with the minimal *total* edge cost.
- It does not necessarily include the minimal cost path between a pair of vertices.
- Example: shortest path from Boston to Providence is along the single edge connecting them
 - that edge is not in the MST

Implementing Prim's Algorithm

- We use the done field to keep track of the sets.
 - if $v.done == true$, v is in set A
 - if $v.done == false$, v is in set B
- We repeatedly scan through the lists of vertices and edges to find the next edge to add.
 - ➔ $O(EV)$
- We can do better!
 - use a heap-based priority queue to store the vertices in set B
 - priority of a vertex $x = -1 * \text{cost of the lowest-cost edge connecting } x \text{ to a vertex in set A}$
 - why multiply by -1 ?
 - somewhat tricky: need to update the priorities over time
 - ➔ $O(E \log V)$

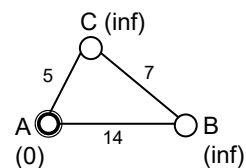
The Shortest-Path Problem

- It's often useful to know the shortest path from one vertex to another – i.e., the one with the minimal total cost
 - example application: routing traffic in the Internet
- For an *unweighted* graph, we can simply do the following:
 - start a breadth-first traversal from the origin, v
 - stop the traversal when you reach the other vertex, w
 - the path from v to w in the resulting (possibly partial) spanning tree is a shortest path
- A breadth-first traversal works for an unweighted graph because:
 - the shortest path is simply one with the fewest edges
 - a breadth-first traversal visits cities in order according to the number of edges they are from the origin.
- Why might this approach fail to work for a *weighted* graph?

Dijkstra's Algorithm

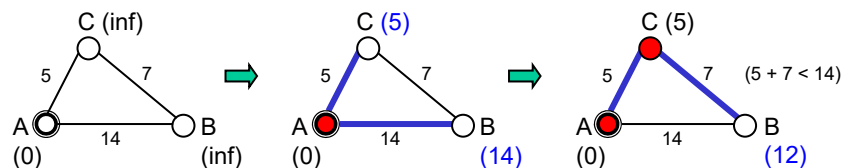
- One algorithm for solving the shortest-path problem for weighted graphs was developed by E.W. Dijkstra.
- It allows us to find the shortest path from a vertex v (the origin) to *all other vertices* that can be reached from v .
- Basic idea:
 - maintain estimates of the shortest paths from the origin to every vertex (along with their costs)
 - gradually refine these estimates as we traverse the graph
- Initial estimates:

	<u>path</u>	<u>cost</u>
the origin itself:	stay put!	0
all other vertices:	unknown	infinity



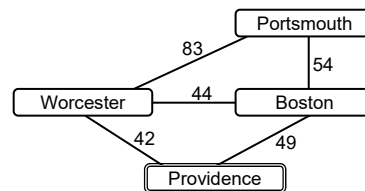
Dijkstra's Algorithm (cont.)

- We say that a vertex w is *finalized* if we have found the shortest path from v to w .
- We repeatedly do the following:
 - find the unfinalized vertex w with the lowest cost estimate
 - mark w as finalized (shown as a filled circle below)
 - examine each unfinalized neighbor x of w to see if there is a shorter path to x that passes through w
 - if there is, update the shortest-path estimate for x
- Example:

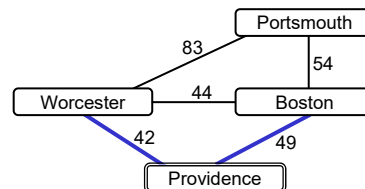


Another Example: Shortest Paths from Providence

- Initial estimates:
- | | |
|------------|----------|
| Boston | infinity |
| Worcester | infinity |
| Portsmouth | infinity |
| Providence | 0 |
- Providence has the smallest unfinalized estimate, so we finalize it.
 - We update our estimates for its neighbors:

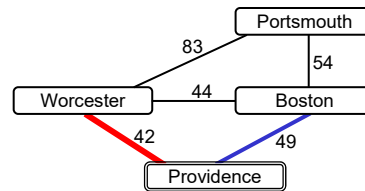


Boston	49 (< infinity)
Worcester	42 (< infinity)
Portsmouth	infinity
Providence	0



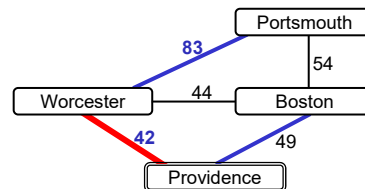
Shortest Paths from Providence (cont.)

Boston	49
Worcester	42
Portsmouth	infinity
Providence	0



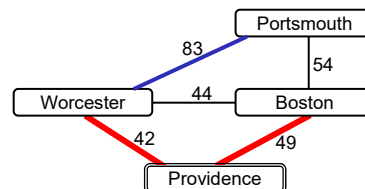
- Worcester has the smallest unfinalized estimate, so we finalize it.
 - any other route from Prov. to Worc. would need to go via Boston, and since $(\text{Prov} \rightarrow \text{Worc}) < (\text{Prov} \rightarrow \text{Bos})$, we can't do better.
- We update our estimates for Worcester's unfinalized neighbors:

Boston	49 (no change)
Worcester	42
Portsmouth	125 ($42 + 83 < \text{infinity}$)
Providence	0



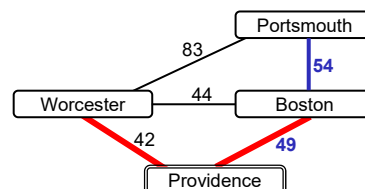
Shortest Paths from Providence (cont.)

Boston	49
Worcester	42
Portsmouth	125
Providence	0



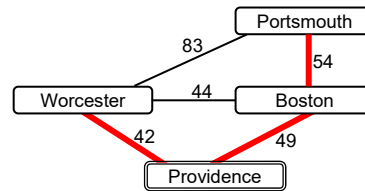
- Boston has the smallest unfinalized estimate, so we finalize it.
 - we'll see later why we can safely do this!
- We update our estimates for Boston's unfinalized neighbors:

Boston	49
Worcester	42
Portsmouth	103 ($49 + 54 < 125$)
Providence	0



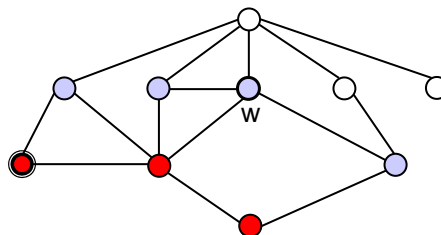
Shortest Paths from Providence (cont.)

Boston	49
Worcester	42
Portsmouth	103
Providence	0



- Only Portsmouth is left, so we finalize it.

Finalizing a Vertex



- origin
- other finalized vertices
- encountered but unfinalized
(i.e., it has a non-infinite estimate)

- Let w be the unfinalized vertex with the smallest cost estimate. Why can we finalize w , before seeing the rest of the graph?
- We know that w 's current estimate is for the shortest path to w that passes through only *finalized* vertices.
- Any shorter path to w would have to pass through one of the other encountered-but-unfinalized vertices, but they are all further away from the origin than w is!
 - their cost estimates may decrease in subsequent stages, but they can't drop below w 's current estimate!

Pseudocode for Dijkstra's Algorithm

```

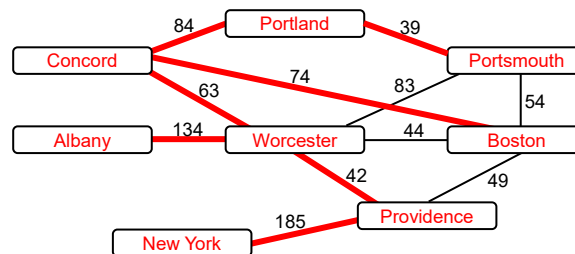
dijkstra(origin)
  origin.cost = 0
  for each other vertex v
    v.cost = infinity;

  while there are still unfinalized vertices with cost < infinity
    find the unfinalized vertex w with the minimal cost
    mark w as finalized

    for each unfinalized vertex x adjacent to w
      cost_via_w = w.cost + edge_cost(w, x)
      if (cost_via_w < x.cost)
        x.cost = cost_via_w
        x.parent = w
  
```

- At the conclusion of the algorithm, for each vertex v:
 - v.cost is the cost of the shortest path from the origin to v
 - if v.cost is infinity, there is no path from the origin to v
 - starting at v and following the parent references yields the shortest path

Example: Shortest Paths from Concord

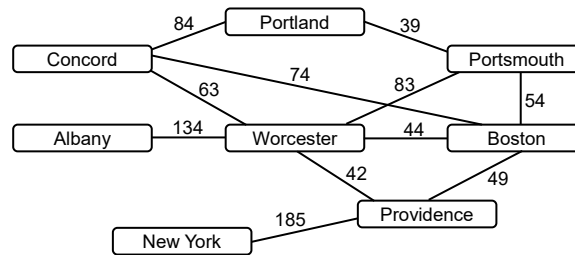


Evolution of the cost estimates (costs in bold have been finalized):

Albany	inf	inf	197	197	197	197	197	
Boston	inf	74	74					
Concord	0							
New York	inf	inf	inf	inf	inf	290	290	290
Portland	inf	84	84	84				
Portsmouth	inf	inf	146	128	123	123		
Providence	inf	inf	105	105	105			
Worcester	inf	63						

Note that the Portsmouth estimate was improved three times!

Another Example: Shortest Paths from Worcester



Evolution of the cost estimates (costs in bold have been finalized):

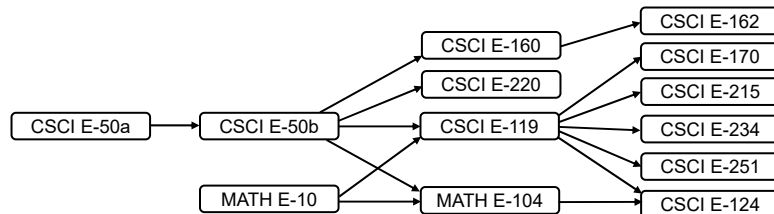
Albany								
Boston								
Concord								
New York								
Portland								
Portsmouth								
Providence								
Worcester								

Implementing Dijkstra's Algorithm

- Similar to the implementation of Prim's algorithm.
- Use a heap-based priority queue to store the unfinalized vertices.
 - priority = ?
- Need to update a vertex's priority whenever we update its shortest-path estimate.
- Time complexity = $O(E \log V)$

Topological Sort

- Used to order the vertices in a directed acyclic graph (a DAG).
- Topological order: an ordering of the vertices such that, if there is directed edge from a to b, a comes before b.
- Example application: ordering courses according to prerequisites



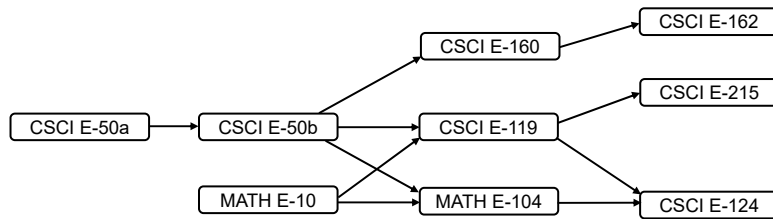
- a directed edge from a to b indicates that a is a prereq of b
- There may be more than one topological ordering.

Topological Sort Algorithm

- A *successor* of a vertex v in a directed graph = a vertex w such that (v, w) is an edge in the graph ($v \rightarrow w$)
- Basic idea: find vertices with no successors and work backward.
 - there must be at least one such vertex. why?
- Pseudocode for one possible approach:

```
topoSort
  S = a stack to hold the vertices as they are visited
  while there are still unvisited vertices
    find a vertex v with no unvisited successors
    mark v as visited
    S.push(v)
  return S
```
- Popping the vertices off the resulting stack gives one possible topological ordering.

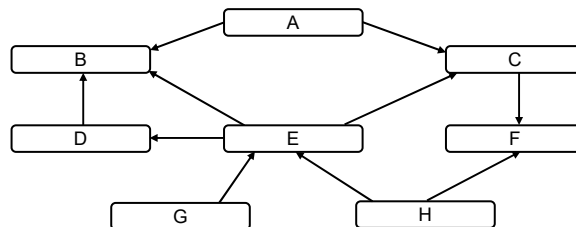
Topological Sort Example



Evolution of the stack:

<u>push</u>	<u>stack contents (top to bottom)</u>
E-124	E-124
E-162	E-162, E-124
E-215	E-215, E-162, E-124
E-104	E-104, E-215, E-162, E-124
E-119	E-119, E-104, E-215, E-162, E-124
E-160	E-160, E-119, E-104, E-215, E-162, E-124
E-10	E-10, E-160, E-119, E-104, E-215, E-162, E-124
E-50b	E-50b, E-10, E-160, E-119, E-104, E-215, E-162, E-124
E-50a	E-50a, E-50b, E-10, E-160, E-119, E-104, E-215, E-162, E-124
one possible topological ordering	

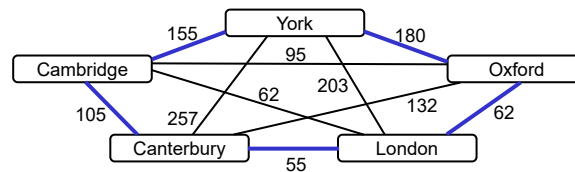
Another Topological Sort Example



Evolution of the stack:

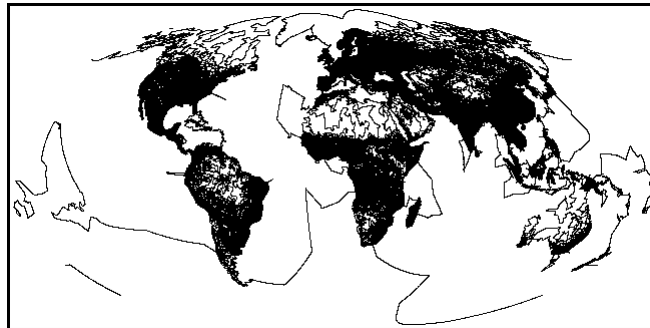
<u>push</u>	<u>stack contents (top to bottom)</u>
-------------	---------------------------------------

Traveling Salesperson Problem (TSP)



- A salesperson needs to travel to a number of cities to visit clients, and wants to do so as efficiently as possible.
- A *tour* is a path that:
 - begins at some starting vertex
 - passes through every other vertex *once and only once*
 - returns to the starting vertex
- TSP: find the tour with the lowest total cost

TSP for Santa Claus



source: <http://www.tsp.gatech.edu/world/pictures.html>

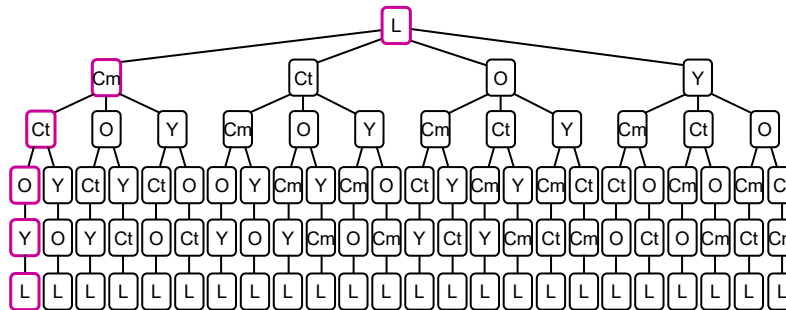
A "world TSP" with 1,904,711 cities.

The figure at right shows a tour with a total cost of 7,516,353,779 meters – which is at most 0.068% longer than the optimal tour.

- Other applications:
 - coin collection from phone booths
 - routes for school buses or garbage trucks
 - minimizing the movements of machines in automated manufacturing processes
 - many others

Solving a TSP: Brute-Force Approach

- Perform an exhaustive search of all possible tours.
 - represent the set of all possible tours as a tree



- The leaf nodes correspond to possible solutions.
 - for n cities, there are $(n - 1)!$ leaf nodes in the tree.
 - half are redundant (e.g., L-Cm-Ct-O-Y-L = L-Y-O-Ct-Cm-L)
- Problem: exhaustive search is intractable for all but small n .
 - example: when $n = 14$, $((n - 1)!)/2 =$ over 3 billion

Solving a TSP: Informed Search

- Focus on the most promising paths through the tree of possible tours.
 - use a function that estimates how good a given path is
- Better than brute force, but still exponential space and time.

Algorithm Analysis Revisited

- Recall that we can group algorithms into classes (n = problem size):

<u>name</u>	<u>example expressions</u>	<u>big-O notation</u>
constant time	1, 7, 10	$O(1)$
logarithmic time	$3\log_{10}n$, $\log_2n + 5$	$O(\log n)$
linear time	$5n$, $10n - 2\log_2n$	$O(n)$
$n \log n$ time	$4n \log_2n$, $n \log_2n + n$	$O(n \log n)$
quadratic time	$2n^2 + 3n$, $n^2 - 1$	$O(n^2)$
n^c ($c > 2$)	$n^3 - 5n$, $2n^5 + 5n^2$	$O(n^c)$
exponential time	2^n , $5e^n + 2n^2$	$O(c^n)$
factorial time	$(n-1)!/2$, $3n!$	$O(n!)$

- Algorithms that fall into one of the classes above the dotted line are referred to as *polynomial-time* algorithms.
- The term *exponential-time algorithm* is sometimes used to include *all* algorithms that fall below the dotted line.
 - algorithms whose running time grows as fast or faster than c^n

Classifying Problems

- Problems that can be solved using a polynomial-time algorithm are considered “easy” problems.
 - we can solve large problem instances in a reasonable amount of time
- Problems that don't have a polynomial-time solution algorithm are considered “hard” or “intractable” problems.
 - they can only be solved exactly for small values of n
- Increasing the CPU speed doesn't help much for intractable problems:

	<u>CPU 1</u>	<u>CPU 2</u> <u>(1000x faster)</u>
max problem size for $O(n)$ alg:	N	1000N
$O(n^2)$ alg:	N	31.6 N
$O(2^n)$ alg:	N	N + 9.97

Dealing With Intractable Problems

- When faced with an intractable problem, we resort to techniques that quickly find solutions that are "good enough".
- Such techniques are often referred to as *heuristic* techniques.
 - heuristic = rule of thumb
 - there's no guarantee these techniques will produce the optimal solution, but they typically work well

Take-Home Lessons

- Object-oriented programming allows us to capture the abstractions in the programs that we write.
 - creates reusable building blocks
 - key concepts: encapsulation, inheritance, polymorphism
- Abstract data types allow us to organize and manipulate collections of data.
 - a given ADT can be implemented in different ways
 - fundamental building blocks: arrays, linked nodes
- Efficiency matters when dealing with large collections of data.
 - some solutions can be *much* faster or more space efficient
 - what's the best data structure/algorithm for *your* workload?
 - example: sorting an almost sorted collection

Take-Home Lessons (cont.)

- Use the tools in your toolbox!
 - interfaces, generic data structures
 - lists/stacks/queues, trees, heaps, hash tables
 - recursion, recursive backtracking, divide-and-conquer
- Use built-in/provided collections/interfaces:
 - `java.util.ArrayList<T>` (implements `List<T>`)
 - `java.util.LinkedList<T>` (implements `List<T>` and `Queue<T>`)
 - `java.util.Stack<T>`
 - `java.util.TreeMap<K, V>` (a balanced search tree)
 - `java.util.HashMap<K, V>` (a hash table)
 - `java.util.PriorityQueue<T>` (a heap)

} implement `Map<K, V>`
- But use them intelligently!
 - ex: `LinkedList` maintains a reference to the last node in the list
 - `list.add(item, n)` will add `item` to the end in $O(n)$ time
 - `list.addLast(item)` will add `item` to the end in $O(1)$ time!