Section 7

CSCI E-22

Will Begin Shortly

Searching a list

Suppose we have a list of items *in sorted order*, in both the `ArrayList` implementation and the `LLList` implementation.

If we want to find a single item in the list, what can we do?

Searching a list

Suppose we have a list of items *in sorted order*, in both the `ArrayList` implementation and the `LLList` implementation.

If we want to find a single item in the list, what can we do?

- We could make a linear pass through the list, checking whether each item in the list is the one we want.
- Or…?

Searching a list

Suppose we have a list of items *in sorted order*, in both the `ArrayList` implementation and the `LLList` implementation.

If we want to find a single item in the list, what can we do?

- We could make a linear pass through the list, checking whether each item in the list is the one we want.
- Or…? We could try to take advantage of the fact that the list is sorted by using binary search!

Searching a list

Suppose we have a list of items *in sorted order*, in both the `ArrayList` implementation and the `LLList` implementation.

If we want to find a single item in the list, what can we do?

- We could make a linear pass through the list, checking whether each item in the list is the one we want.
- Or…? We could try to take advantage of the fact that the list is sorted by using binary search!

What is the efficiency if we use binary search on the array implementation?

Searching a list

Suppose we have a list of items *in sorted order*, in both the `ArrayList` implementation and the `LLList` implementation.

If we want to find a single item in the list, what can we do?

- We could make a linear pass through the list, checking whether each item in the list is the one we want.
- Or…? We could try to take advantage of the fact that the list is sorted by using binary search!

What is the efficiency if we use binary search on the array implementation?

For the array implementation, our efficiency will be O(logn).

Searching a list

Suppose we have a list of items *in sorted order*, in both the `ArrayList` implementation and the `LLList` implementation.

If we want to find a single item in the list, what can we do?

- We could make a linear pass through the list, checking whether each item in the list is the one we want.
- Or…? We could try to take advantage of the fact that the list is sorted by using binary search!

What is the efficiency if we use binary search on the array implementation?

For the array implementation, our efficiency will be O(logn).

How about the linked list implementation?

Searching a list

Suppose we have a list of items *in sorted order*, in both the `ArrayList` implementation and the `LLList` implementation.

If we want to find a single item in the list, what can we do?

- We could make a linear pass through the list, checking whether each item in the list is the one we want.
- Or…? We could try to take advantage of the fact that the list is sorted by using binary search!

What is the efficiency if we use binary search on the array implementation?

For the array implementation, our efficiency will be O(logn).

How about the linked list implementation?

Our efficiency drops to O(n * logn), since we may need to traverse the list, which is an O(n) operation on a linked list, in each of the O(logn) iterations of binary search.

Searching a list

Suppose we have a list of items *in sorted order*, in both the `ArrayList` implementation and the `LLList` implementation.

If we want to find a single item in the list, what can we do?

- We could make a linear pass through the list, checking whether each item in the list is the one we want.
- Or…? We could try to take advantage of the fact that the list is sorted by using binary search!

What is the efficiency if we use binary search on the array implementation?

For the array implementation, our efficiency will be O(logn).

How about the linked list implementation?

Our efficiency drops to O(n * logn), since we may need to traverse the list, which is an O(n) operation on a linked list, in each of the O(logn) iterations of binary search.

What would be a better way to search if we knew we were using the linked list implementation?

Searching a list

Suppose we have a list of items *in sorted order*, in both the `ArrayList` implementation and the `LLList` implementation.

If we want to find a single item in the list, what can we do?

- We could make a linear pass through the list, checking whether each item in the list is the one we want.
- Or…? We could try to take advantage of the fact that the list is sorted by using binary search!

What is the efficiency if we use binary search on the array implementation?

For the array implementation, our efficiency will be O(logn).

How about the linked list implementation?

Our efficiency drops to O(n * logn), since we may need to traverse the list, which is an O(n) operation on a linked list, in each of the O(logn) iterations of binary search.

What would be a better way to search if we knew we were using the linked list implementation?

Our standard linear search would be better. We'd simply traverse the linked list once, which would yield an O(n) efficiency.

Searching a list

However, implementing a linear search with an `LLList` is not as straightforward as it might seem. Let's say that we are using an instance of our `LLList` class in which the items are in sorted order, and that our `search()` method is *external* to the `LLList` class. Let's also assume that the objects in the `LLList` implement the `Comparable` interface.

Searching a list

However, implementing a linear search with an `LLList` is not as straightforward as it might seem. Let's say that we are using an instance of our `LLList` class in which the items are in sorted order, and that our `search()` method is *external* to the `LLList` class. Let's also assume that the objects in the `LLList` implement the `Comparable` interface.

Remember, this means they must provide an instance method `compareTo()` that allows two objects to be compared (i.e., to determine which object "comes before" another when sorting).

`a.compareTo(b)` returns a negative integer when `a` comes before `b`, zero if they are equal, and a positive integer when `a` comes after `b`

Searching a list

However, implementing a linear search with an `LLList` is not as straightforward as it might seem. Let's say that we are using an instance of our `LLList` class in which the items are in sorted order, and that our `search()` method is *external* to the `LLList` class. Let's also assume that the objects in the `LLList` implement the `Comparable` interface.

Here's one way we could write our search() method with this framework:

```
public static boolean search(Comparable item, LLList list) {
    if (item == null || list == null)
        return false;

    for (int i = 0; i < list.length(); i++) {
        Comparable listItem = (Comparable)list.getItem(i);
        if (item.equals(listItem))
            return true;
        if (item.compareTo(listItem) < 0)
            return false;
    }

    return false;
}
```

Remember, this means they must provide an instance method `compareTo()` that allows two objects to be compared (i.e., to determine which object "comes before" another when sorting).

`a.compareTo(b)` returns a negative integer when `a` comes before `b`, zero if they are equal, and a positive integer when `a` comes after `b`

Searching a list

However, implementing a linear search with an `LLList` is not as straightforward as it might seem. Let's say that we are using an instance of our `LLList` class in which the items are in sorted order, and that our `search()` method is *external* to the `LLList` class. Let's also assume that the objects in the `LLList` implement the `Comparable` interface.

Here's one way we could write our search() method with this framework:

```
public static boolean search(Comparable item, LLList list) {
    if (item == null || list == null)
        return false;

    for (int i = 0; i < list.length(); i++) {
        Comparable listItem = (Comparable)list.getItem(i);
        if (item.equals(listItem))
            return true;
        if (item.compareTo(listItem) < 0)
            return false;
    }

    return false;
}
```


Notice any problems with this implementation?

Searching a list

However, implementing a linear search with an `LLList` is not as straightforward as it might seem. Let's say that we are using an instance of our `LLList` class in which the items are in sorted order, and that our `search()` method is *external* to the `LLList` class. Let's also assume that the objects in the `LLList` implement the `Comparable` interface.

Here's one way we could write our search() method with this framework:

```
public static boolean search(Comparable item, LLList list) {
    if (item == null || list == null)
        return false;

    for (int i = 0; i < list.length(); i++) {
        Comparable listItem = (Comparable)list.getItem(i);
        if (item.equals(listItem))
            return true;
        if (item.compareTo(listItem) < 0)
            return false;
    }

    return false;
}
```

Notice any problems with this implementation?

It's $O(n^2)$! Every call to `getItem()` begins at the first node of the list and traverses the list until it reaches node `i`.

Searching a list

However, implementing a linear search with an `LLList` is not as straightforward as it might seem. Let's say that we are using an instance of our `LLList` class in which the items are in sorted order, and that our `search()` method is *external* to the `LLList` class. Let's also assume that the objects in the `LLList` implement the `Comparable` interface.

Here's one way we could write our search() method with this framework:

```
public static boolean search(Comparable item, LLList list) {
    if (item == null || list == null)
        return false;

    for (int i = 0; i < list.length(); i++) {
        Comparable listItem = (Comparable)list.getItem(i);
        if (item.equals(listItem))
            return true;
        if (item.compareTo(listItem) < 0)
            return false;
    }

    return false;
}
```

Notice any problems with this implementation?

It's $O(n^2)$! Every call to `getItem()` begins at the first node of the list and traverses the list until it reaches node `i`.

Any ideas on how we could fix this?

Searching a list

However, implementing a linear search with an `LLList` is not as straightforward as it might seem. Let's say that we are using an instance of our `LLList` class in which the items are in sorted order, and that our `search()` method is *external* to the `LLList` class. Let's also assume that the objects in the `LLList` implement the `Comparable` interface.

Here's one way we could write our search() method with this framework:

```
public static boolean search(Comparable item, LLList list) {
    if (item == null || list == null)
        return false;

    for (int i = 0; i < list.length(); i++) {
        Comparable listItem = (Comparable)list.getItem(i);
        if (item.equals(listItem))
            return true;
        if (item.compareTo(listItem) < 0)
            return false;
    }

    return false;
}
```

Notice any problems with this implementation?

It's $O(n^2)$! Every call to `getItem()` begins at the first node of the list and traverses the list until it reaches node `i`.

Any ideas on how we could fix this?

Use an iterator! That way, our method will be able to complete its task using only a single pass of the underlying linked list.

Searching a list

Here's our `LLList search()` method implemented using an iterator:

```
public static boolean search(Comparable item, LLList list) {
    if (item == null || list == null) {
        return false;
    }

    ListIterator iter = list.iterator();
    while (iter.hasNext()) {
        Comparable listItem = (Comparable)iter.next();
        if (item.equals(listItem)) {
            return true;
        }

        if (item.compareTo(listItem) < 0) {
            // we went past what we're looking for
            return false;
        }
    }

    return false;
}
```

Searching a list

Here's our `LLList search()` method implemented using an iterator:

```
public static boolean search(Comparable item, LLList list) {
    if (item == null || list == null) {
        return false;
    }

    ListIterator iter = list.iterator();
    while (iter.hasNext()) {
        Comparable listItem = (Comparable)iter.next();
        if (item.equals(listItem)) {
            return true;
        }

        if (item.compareTo(listItem) < 0) {
            // we went past what we're looking for
            return false;
        }
    }

    return false;
}
```

Because the iterator has a reference to the underlying linked list, we can use it to iterate over the internals of the `LLList` *as if* we had that direct access!

In other words, with an iterator, we can perform an O(n) search on a List implemented with *any* underlying data structure.

Stacks

- A *stack* is a collection that follows the last-in, first-out (LIFO) rule.
  - `push(item)` is used to insert a new item at the top
  - `pop()` is the only way to remove an item, and it removes the topmost item
  - `peek()` is used to access the topmost item without popping it

Stacks

- A *stack* is a collection that follows the last-in, first-out (LIFO) rule.
  - `push(item)` is used to insert a new item at the top
  - `pop()` is the only way to remove an item, and it removes the topmost item
  - `peek()` is used to access the topmost item without popping it
- In lecture, we covered two implementations of the `Stack` interface: `ArrayStack` and `LLStack`.
  - **For both implementations, what is the time complexity of `push()`, `pop()`, and `peek()`?**

Stacks

- A *stack* is a collection that follows the last-in, first-out (LIFO) rule.
  - `push(item)` is used to insert a new item at the top
  - `pop()` is the only way to remove an item, and it removes the topmost item
  - `peek()` is used to access the topmost item without popping it
- In lecture, we covered two implementations of the `Stack` interface: `ArrayStack` and `LLStack`.
  - For both implementations, what is the time complexity of `push()`, `pop()`, and `peek()`?
    - The performance is O(1) for all operations in both implementations.
    - For `ArrayStack`, `push()` and `pop()` does array access & updating an integer.
    - For `LLStack`, `push()` and `pop()` involve modifying a few references.

Stacks

- A *stack* is a collection that follows the last-in, first-out (LIFO) rule.
  - `push(item)` is used to insert a new item at the top
  - `pop()` is the only way to remove an item, and it removes the topmost item
  - `peek()` is used to access the topmost item without popping it
- In lecture, we covered two implementations of the `Stack` interface: `ArrayStack` and `LLStack`.
  - For both implementations, what is the time complexity of `push()`, `pop()`, and `peek()`?
    - The performance is O(1) for all operations in both implementations.
    - For `ArrayStack`, `push()` and `pop()` does array access & updating an integer.
    - For `LLStack`, `push()` and `pop()` involve modifying a few references.
  - **How do both implementations use memory?**

Stacks

- A *stack* is a collection that follows the last-in, first-out (LIFO) rule.
  - `push(item)` is used to insert a new item at the top
  - `pop()` is the only way to remove an item, and it removes the topmost item
  - `peek()` is used to access the topmost item without popping it
- In lecture, we covered two implementations of the `Stack` interface: `ArrayStack` and `LLStack`.
  - For both implementations, what is the time complexity of `push()`, `pop()`, and `peek()`?
    - The performance is O(1) for all operations in both implementations.
    - For `ArrayStack`, `push()` and `pop()` does array access & updating an integer.
    - For `LLStack`, `push()` and `pop()` involve modifying a few references.

  - How do both implementations use memory?
    - An array has a fixed size, and `ArrayStack` needs to allocate O(*m*) memory beforehand. A linked list only uses the amount of memory for its nodes.

Stacks

- Suppose there are a number of `push()` and `pop()` calls to a stack. The numbers 1 through 8 will be pushed onto the stack in order, with zero or more pops after each push. When a number is popped, it is immediately printed. Which of the following outputs are impossible?
  - `1 2 3 4 8 7 6 5`

Stacks

- Suppose there are a number of `push()` and `pop()` calls to a stack. The numbers 1 through 8 will be pushed onto the stack in order, with zero or more pops after each push. When a number is popped, it is immediately printed. Which of the following outputs are impossible?
  - `1 2 3 4 8 7 6 5` **possible**
    - push 1, pop, push 2, pop, push 3, pop, push 4, pop, push 5, push 6, push 7, push 8, pop...

Stacks

- Suppose there are a number of `push()` and `pop()` calls to a stack. The numbers 1 through 8 will be pushed onto the stack in order, with zero or more pops after each push. When a number is popped, it is immediately printed. Which of the following outputs are impossible?
  - `1 2 3 4 8 7 6 5` **possible**
    - push 1, pop, push 2, pop, push 3, pop, push 4, pop, push 5, push 6, push 7, push 8, pop...
  - `1 3 6 8 7 5 4 2`

Stacks

- Suppose there are a number of `push()` and `pop()` calls to a stack. The numbers 1 through 8 will be pushed onto the stack in order, with zero or more pops after each push. When a number is popped, it is immediately printed. Which of the following outputs are impossible?
  - `1 2 3 4 8 7 6 5` **possible**
    - push 1, pop, push 2, pop, push 3, pop, push 4, pop, push 5, push 6, push 7, push 8, pop...
  - `1 3 6 8 7 5 4 2` **possible**
    - push 1, pop, push 2, push 3, pop, push 4, push 5, push 6, pop, push 7, push 8, pop...

Stacks

- Suppose there are a number of `push()` and `pop()` calls to a stack. The numbers 1 through 8 will be pushed onto the stack in order, with zero or more pops after each push. When a number is popped, it is immediately printed. Which of the following outputs are impossible?
  - `1 2 3 4 8 7 6 5` **possible**
    - push 1, pop, push 2, pop, push 3, pop, push 4, pop, push 5, push 6, push 7, push 8, pop...
  - `1 3 6 8 7 5 4 2` **possible**
    - push 1, pop, push 2, push 3, pop, push 4, push 5, push 6, pop, push 7, push 8, pop...
  - `8 7 6 5 1 2 3 4`

Stacks

- Suppose there are a number of `push()` and `pop()` calls to a stack. The numbers 1 through 8 will be pushed onto the stack in order, with zero or more pops after each push. When a number is popped, it is immediately printed. Which of the following outputs are impossible?
  - `1 2 3 4 8 7 6 5` **possible**
    - push 1, pop, push 2, pop, push 3, pop, push 4, pop, push 5, push 6, push 7, push 8, pop...
  - `1 3 6 8 7 5 4 2` **possible**
    - push 1, pop, push 2, push 3, pop, push 4, push 5, push 6, pop, push 7, push 8, pop...
  - `8 7 6 5 1 2 3 4` **impossible**
    - After 5 is printed, the stack looks like **`[1, 2, 3, 4`** and the only operation we can do is pop 4 off the stack.

Stacks

- Suppose there are a number of `push()` and `pop()` calls to a stack. The numbers 1 through 8 will be pushed onto the stack in order, with zero or more pops after each push. When a number is popped, it is immediately printed. Which of the following outputs are impossible?
  - `1 2 3 4 8 7 6 5` **possible**
    - push 1, pop, push 2, pop, push 3, pop, push 4, pop, push 5, push 6, push 7, push 8, pop...
  - `1 3 6 8 7 5 4 2` **possible**
    - push 1, pop, push 2, push 3, pop, push 4, push 5, push 6, pop, push 7, push 8, pop...
  - `8 7 6 5 1 2 3 4` **impossible**
    - After 5 is printed, the stack looks like **`[1, 2, 3, 4`** and the only operation we can do is pop 4 off the stack.
  - `3 5 4 2 8 7 1 6`

Stacks

- Suppose there are a number of `push()` and `pop()` calls to a stack. The numbers 1 through 8 will be pushed onto the stack in order, with zero or more pops after each push. When a number is popped, it is immediately printed. Which of the following outputs are impossible?
  - `1 2 3 4 8 7 6 5` **possible**
    - push 1, pop, push 2, pop, push 3, pop, push 4, pop, push 5, push 6, push 7, push 8, pop...
  - `1 3 6 8 7 5 4 2` **possible**
    - push 1, pop, push 2, push 3, pop, push 4, push 5, push 6, pop, push 7, push 8, pop...
  - `8 7 6 5 1 2 3 4` **impossible**
    - After 5 is printed, the stack looks like **`[1, 2, 3, 4`** and the only operation we can do is pop 4 off the stack.

  - `3 5 4 2 8 7 1 6` **impossible**
    - The 1 cannot be popped before the 6.
      push 1, push 2, push 3, pop, push 4, push 5, pop, pop, pop, push 6, push 7, push 8, pop, pop, **!!!**

Generics

- The Stack interface is generic, which means the items in the stack can be limited to a specific type in code and checked by the Java compiler.
  - Why is this a safer option than using `Object` everywhere?
    If `Stack` weren't generic, the following code would compile, but we can *prove* it will crash:

```
Stack s = new ArrayStack(10);
s.push(1);
s.push(2);
s.push("three");

int sum = 0;
while (!s.isEmpty()) {
    int n = (int) s.pop();
    sum += n;
}

System.out.println(sum);
```

crashes when string `"three"` is popped off the stack because it is not an `int` and cannot be added to `sum`

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

Generics

- With the generic Stack, the `s.push("three")` call **no longer compiles**.

```
Stack<Integer> s = new ArrayStack<>(10);
s.push(1);
s.push(2);
s.push("three");

int sum = 0;
while (!s.isEmpty()) {
    int n = (int) s.pop();
    sum += n;
}

System.out.println(sum);
```

fails to compile here

```
error: incompatible types: String cannot be converted to Integer
    s.push("three");
           ^
```

Generics

- Generics will not prevent all runtime errors. Here's some code that **will still crash at runtime**:

```
Stack<Integer> s = new ArrayStack<>(10);
int n = s.pop();
System.out.println(n);
```

  - **Why does this crash?**
    See the `pop()` method in `ArrayStack`. If the stack is empty, this method returns `null`. The compiler-accepted line where the crash occurs is the **`int n = s.pop()`** assignment. `null` is a valid value for an `Integer` variable (and any other reference type), but not for a primitive `int`, so a `NullPointerException` is thrown at runtime.

Balancing braces

- In lecture, we briefly mentioned an application for stacks that involves processing a sequence of delimiters (parentheses, braces and brackets) to ensure they are balanced.
- A sequence of delimiters is balanced when every left delimiter has a matching right delimiter in the correct order. For example:
  - {{{}}} is balanced because every opening delimiter has a closing delimiter and there are no extra closing delimiters
  - {{{}{}{}}} is balanced (open brackets can come after closing brackets as long as they are eventually closed)
  - {([}]) is not balanced (mismatching delimiter types)
  - ((((is not balanced (missing closing delimiters)
  -)))) is not balanced (missing opening delimiters)
- Let's write a method `isBalanced()` that takes a string containing delimiters and returns `true` if the delimiters are balanced, and `false` otherwise.

```
...
for (int i = 0; i < s.length(); i++) {
    char c = s.charAt(i);

    if ("{[(".indexOf(c) >= 0) {
        stack.push(c);

    } else if ("}])".indexOf(c) >= 0) {
        if (stack.isEmpty()) {
            return false;
        }

        char stackChar = stack.pop();

        if (
            stackChar == '{' && c != '}' ||
            stackChar == '[' && c != ']' ||
            stackChar == '(' && c != ')'
        ) {
            return false;
        }
    }
}
...
```

when an open delimiter is seen, push it onto the stack

check that the close delimiter and the open delimiter (popped off the stack) are the same type