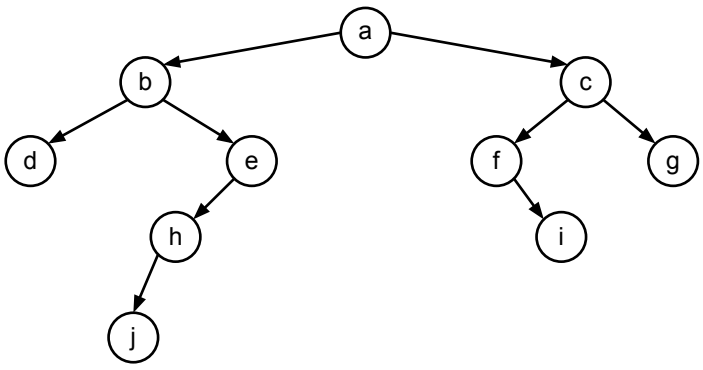
Section 8

CSCI E-22

Will Begin Shortly

Binary trees

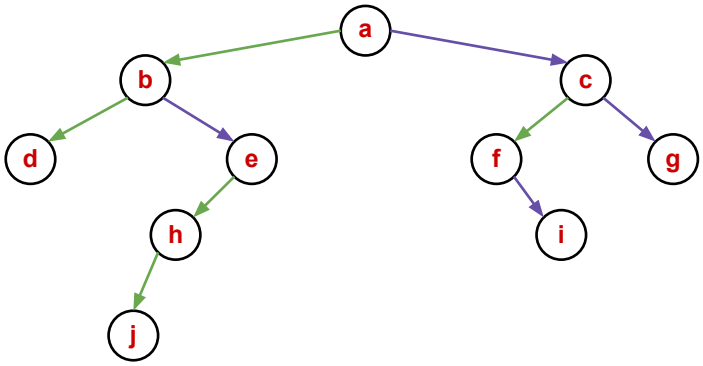
```
public class LinkedTree {
  private class Node {
    private int key;
    private LLList data;
    private Node left;
    private Node right;
    ...
  }

  private Node root;

  ...
}
```



1. What are the ancestors of node *h*?
2. What are the descendants of node *c*?
3. What is the depth of node *i*?
4. What is the height of this tree?

Binary trees

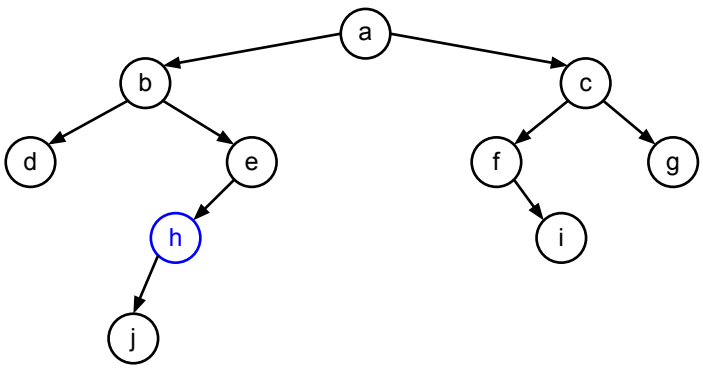
```
public class LinkedTree {
  private class Node {
    private int key;
    private LLList data;
    private Node left;
    private Node right;
    ...
  }

  private Node root;

  ...
}
```



1. What are the ancestors of node *h*?
2. What are the descendants of node *c*?
3. What is the depth of node *i*?
4. What is the height of this tree?

Binary trees

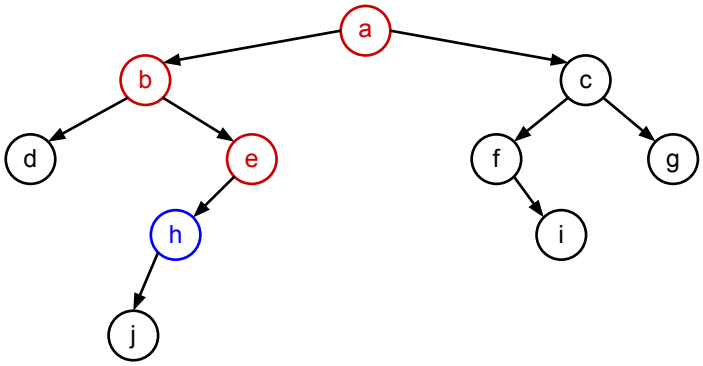
```
public class LinkedTree {
  private class Node {
    private int key;
    private LLList data;
    private Node left;
    private Node right;
    ...
  }

  private Node root;

  ...
}
```



1. What are the ancestors of node *h*?
2. What are the descendants of node *c*?
3. What is the depth of node *i*?
4. What is the height of this tree?

Binary trees

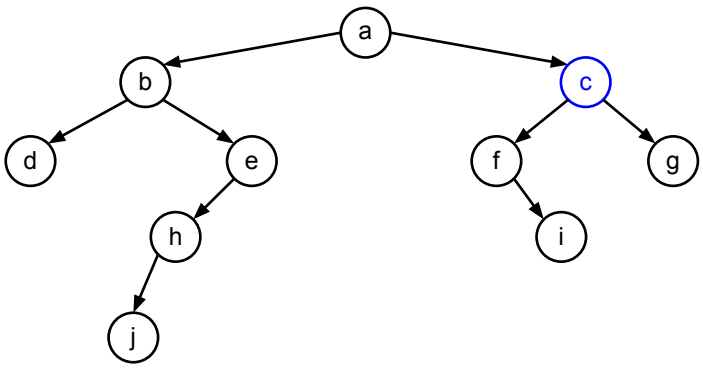
```
public class LinkedTree {
  private class Node {
    private int key;
    private LLList data;
    private Node left;
    private Node right;
    ...
  }

  private Node root;

  ...
}
```



1. What are the ancestors of node *h*? e, b, & a
2. What are the descendants of node *c*?
3. What is the depth of node *i*?
4. What is the height of this tree?

Binary trees

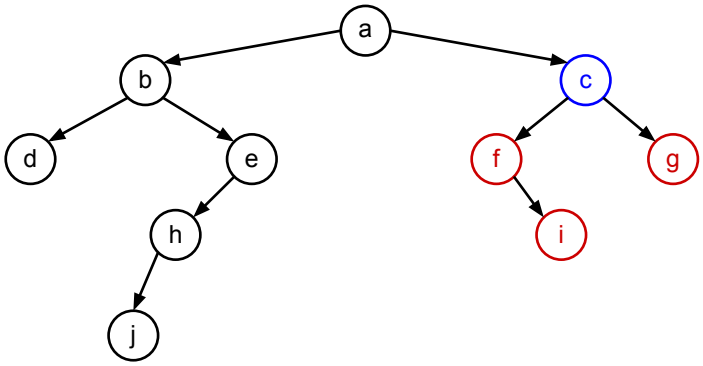
```
public class LinkedTree {
  private class Node {
    private int key;
    private LLList data;
    private Node left;
    private Node right;
    ...
  }

  private Node root;

  ...
}
```



1. What are the ancestors of node *h*? e, b, & a
2. What are the descendants of node *c*?
3. What is the depth of node *i*?
4. What is the height of this tree?

Binary trees

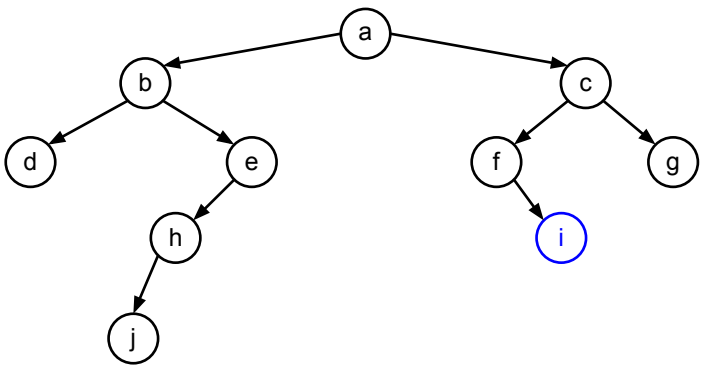
```
public class LinkedTree {
  private class Node {
    private int key;
    private LLList data;
    private Node left;
    private Node right;
    ...
  }

  private Node root;

  ...
}
```



1. What are the ancestors of node *h*? e, b, & a
2. What are the descendants of node *c*? f, g, & i
3. What is the depth of node *i*?
4. What is the height of this tree?

Binary trees

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

  private Node root;

  ...
}
```

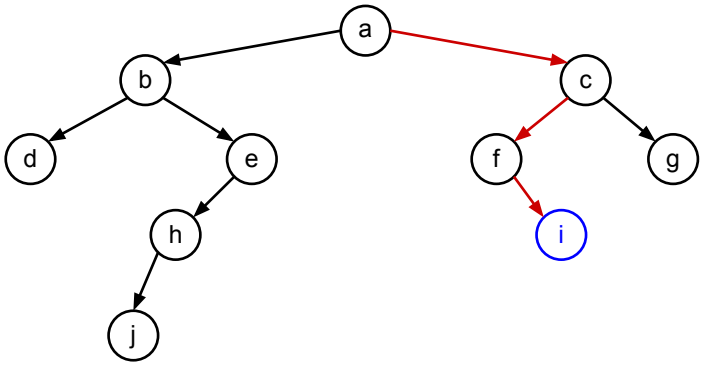
1. What are the ancestors of node *h*? e, b, & a
2. What are the descendants of node *c*? f, g, & i

3. What is the depth of node *i*?

4. What is the height of this tree?

Binary trees

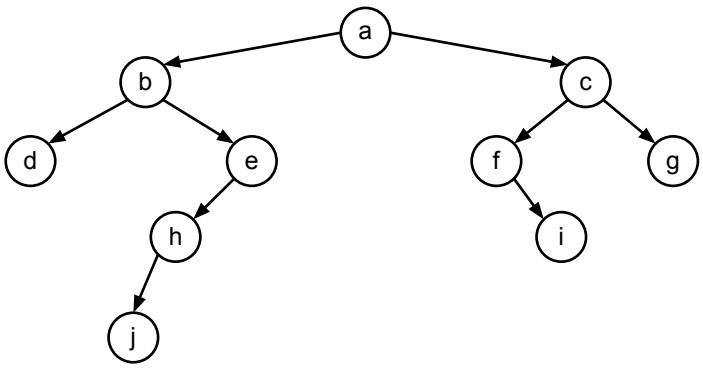
```
public class LinkedTree {
  private class Node {
    private int key;
    private LLList data;
    private Node left;
    private Node right;
    ...
  }

  private Node root;

  ...
}
```



1. What are the ancestors of node *h*? e, b, & a
2. What are the descendants of node *c*? f, g, & i
3. What is the depth of node *i*? 3, since the path from the root contains 3 links
4. What is the height of this tree?

Binary trees

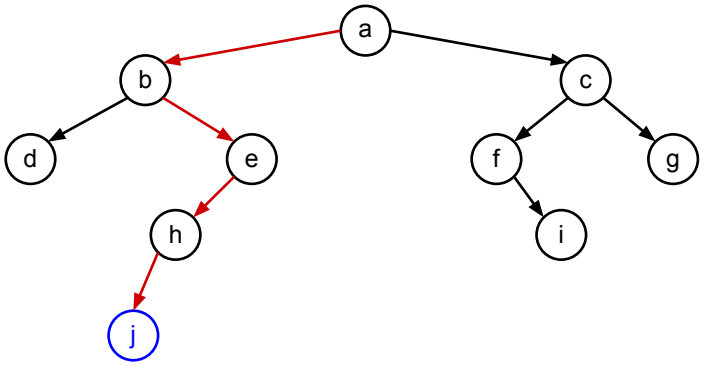
```
public class LinkedTree {
  private class Node {
    private int key;
    private LLList data;
    private Node left;
    private Node right;
    ...
  }

  private Node root;

  ...
}
```



1. What are the ancestors of node *h*? e, b, & a
2. What are the descendants of node *c*? f, g, & i
3. What is the depth of node *i*? 3, since the path from the root contains 3 links
4. What is the height of this tree?

Binary trees

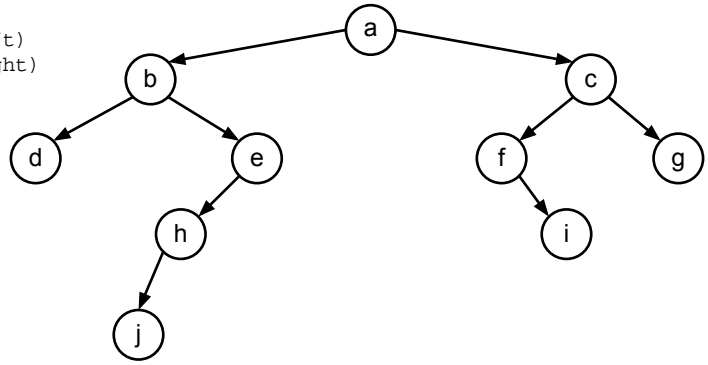
```
public class LinkedTree {
  private class Node {
    private int key;
    private LLList data;
    private Node left;
    private Node right;
    ...
  }

  private Node root;

  ...
}
```



1. What are the ancestors of node *h*? e, b, & a
2. What are the descendants of node *c*? f, g, & i
3. What is the depth of node *i*? 3, since the path from the root contains 3 links
4. What is the height of this tree? 4, since j is deepest and its depth is 4

Binary tree traversals

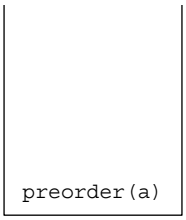
```
preorder(node) {
  print(node.key)
  preorder(node.left)
  preorder(node.right)
}
```

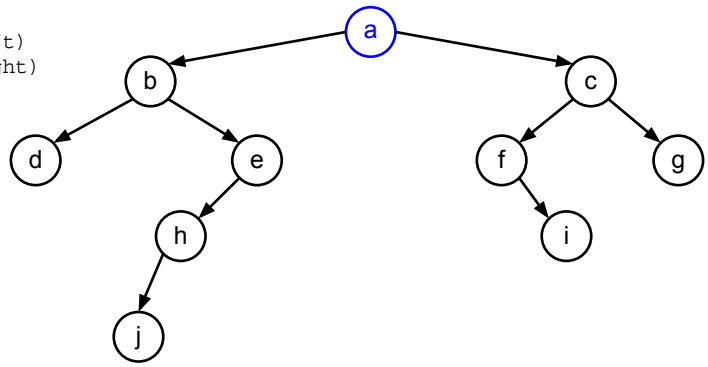


- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if a **pre-order traversal** is made?

Binary tree traversals

```
preorder(node) {
  print(node.key)
  preorder(node.left)
  preorder(node.right)
}
```



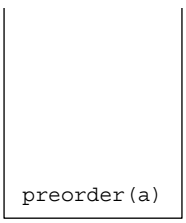


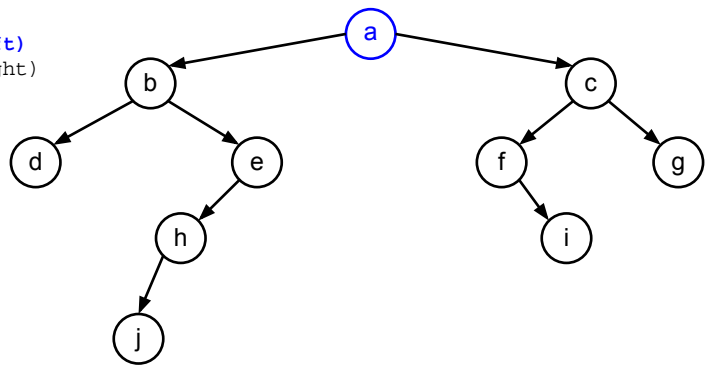
- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if a **pre-order traversal** is made?

a

Binary tree traversals

```
preorder(node) {
  print(node.key)
  preorder(node.left)
  preorder(node.right)
}
```



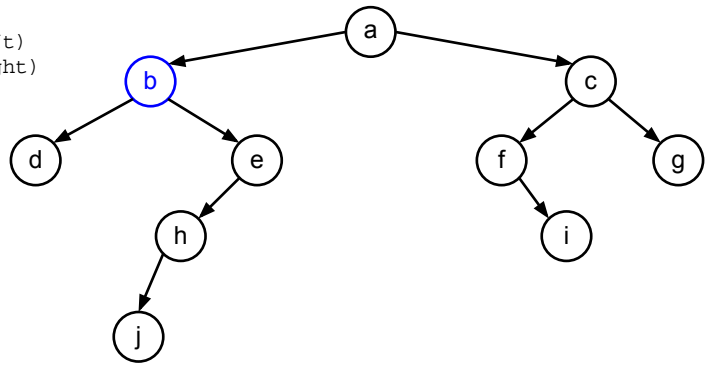


- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if a **pre-order traversal** is made?

a

Binary tree traversals

```
preorder(node) {
  print(node.key)
  preorder(node.left)
  preorder(node.right)
}
```



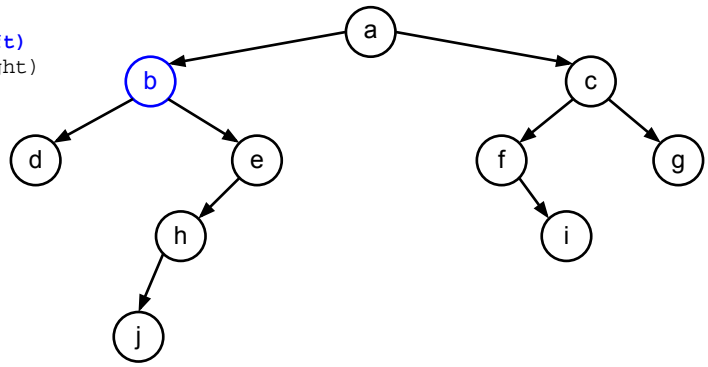

```
preorder(b)
preorder(a)
```

- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if a **pre-order traversal** is made?

a b

Binary tree traversals

```
preorder(node) {
  print(node.key)
  preorder(node.left)
  preorder(node.right)
}
```



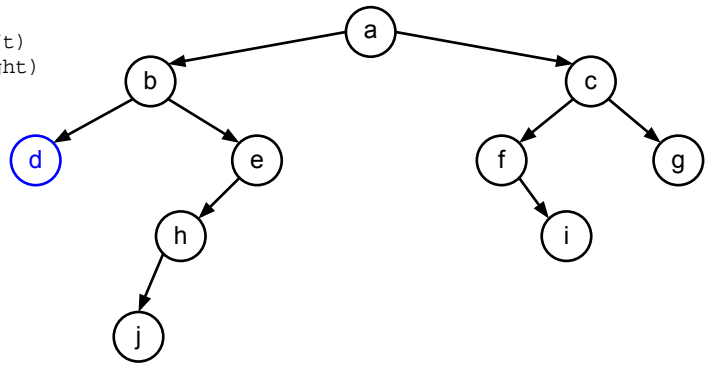

```
preorder(b)
preorder(a)
```

- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if a **pre-order traversal** is made?

a b

Binary tree traversals

```
preorder(node) {
  print(node.key)
  preorder(node.left)
  preorder(node.right)
}
```



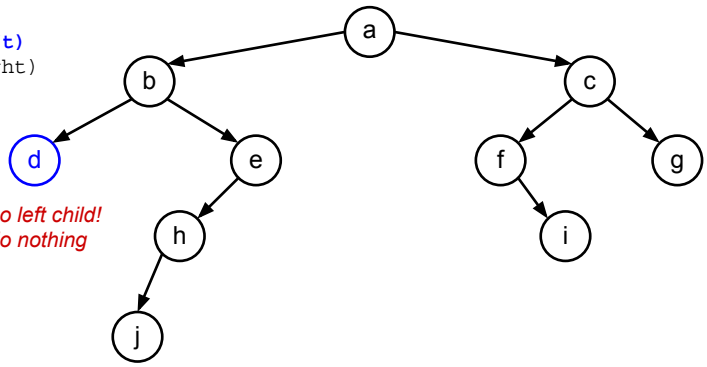

```
preorder(d)
preorder(b)
preorder(a)
```

- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if a **pre-order traversal** is made?

a b d

Binary tree traversals

```
preorder(node) {
  print(node.key)
  preorder(node.left)
  preorder(node.right)
}
```



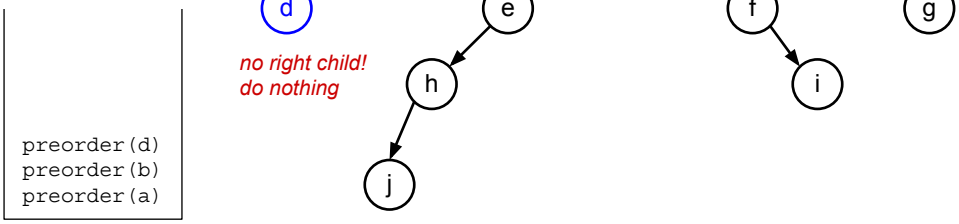

```
preorder(d)
preorder(b)
preorder(a)
```

- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if a **pre-order traversal** is made?

a b d

Binary tree traversals

```
preorder(node) {
  print(node.key)
  preorder(node.left)
  preorder(node.right)
}
```

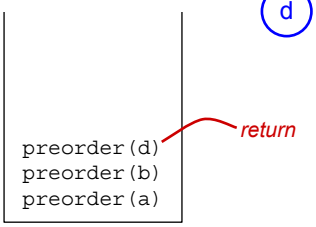


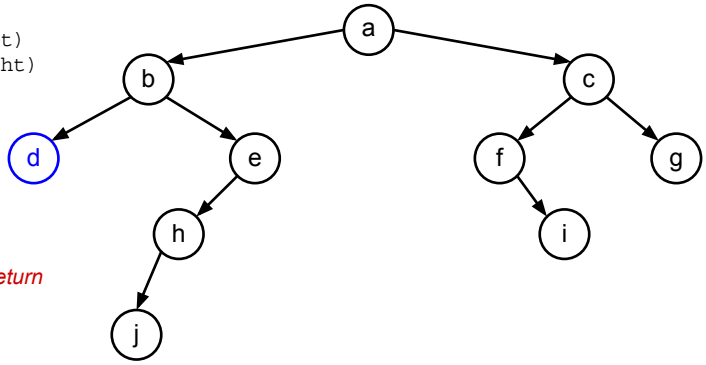
- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if a **pre-order traversal** is made?

a b d

Binary tree traversals

```
preorder(node) {
  print(node.key)
  preorder(node.left)
  preorder(node.right)
}
```



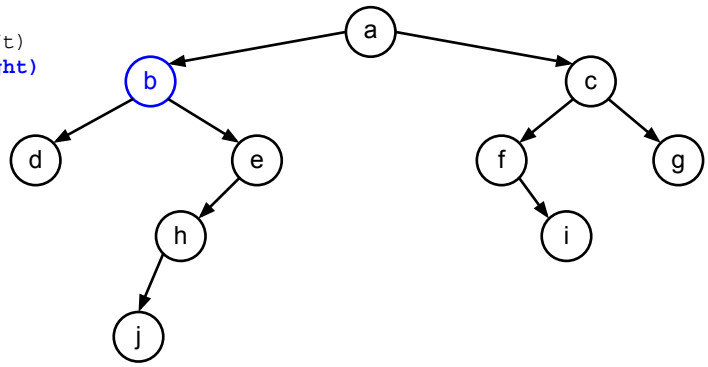


- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if a **pre-order traversal** is made?

a b d

Binary tree traversals

```
preorder(node) {
  print(node.key)
  preorder(node.left)
  preorder(node.right)
}
```



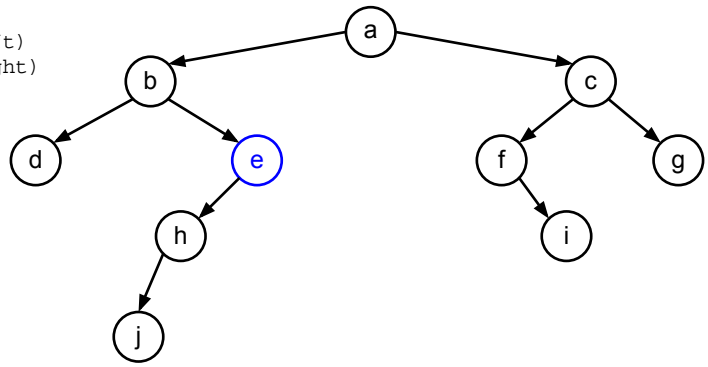

```
preorder(b)
preorder(a)
```

- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if a **pre-order traversal** is made?

a b d

Binary tree traversals

```
preorder(node) {
  print(node.key)
  preorder(node.left)
  preorder(node.right)
}
```



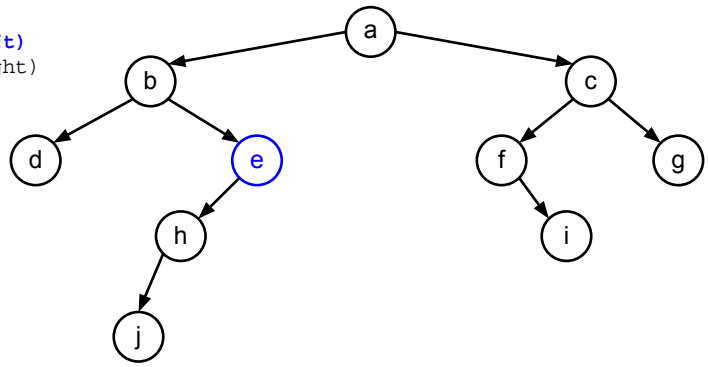

```
preorder(e)
preorder(b)
preorder(a)
```

- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if a **pre-order traversal** is made?

a b d e

Binary tree traversals

```
preorder(node) {
  print(node.key)
  preorder(node.left)
  preorder(node.right)
}
```



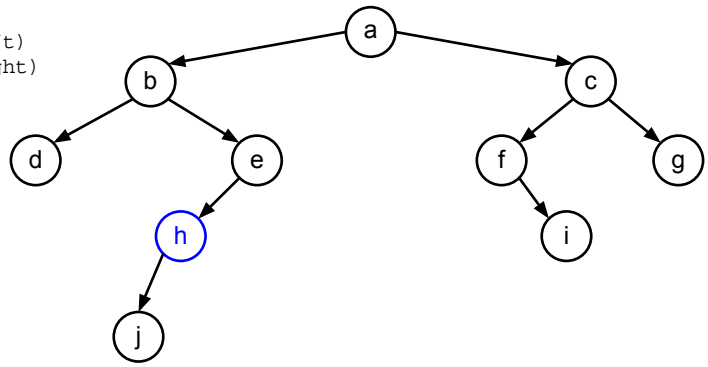

```
preorder(e)
preorder(b)
preorder(a)
```

- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if a **pre-order traversal** is made?

a b d e

Binary tree traversals

```
preorder(node) {
  print(node.key)
  preorder(node.left)
  preorder(node.right)
}
```




```
preorder(h)
preorder(e)
preorder(b)
preorder(a)
```

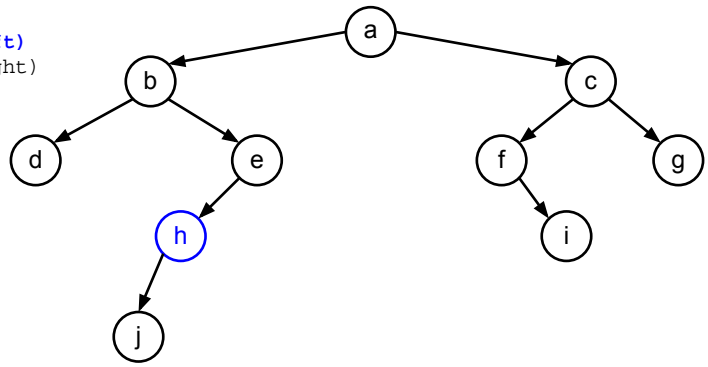
- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if a **pre-order traversal** is made?

a b d e h

Binary tree traversals

```
preorder(node) {
  print(node.key)
  preorder(node.left)
  preorder(node.right)
}
```

```
preorder(h)
preorder(e)
preorder(b)
preorder(a)
```



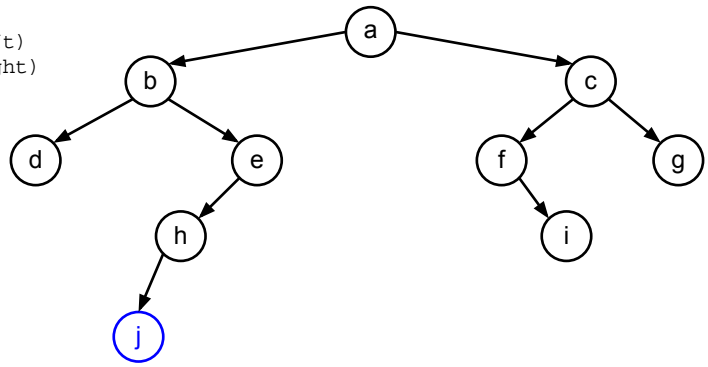
- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if a **pre-order traversal** is made?

a b d e h

Binary tree traversals

```
preorder(node) {
  print(node.key)
  preorder(node.left)
  preorder(node.right)
}
```

```
preorder(j)
preorder(h)
preorder(e)
preorder(b)
preorder(a)
```

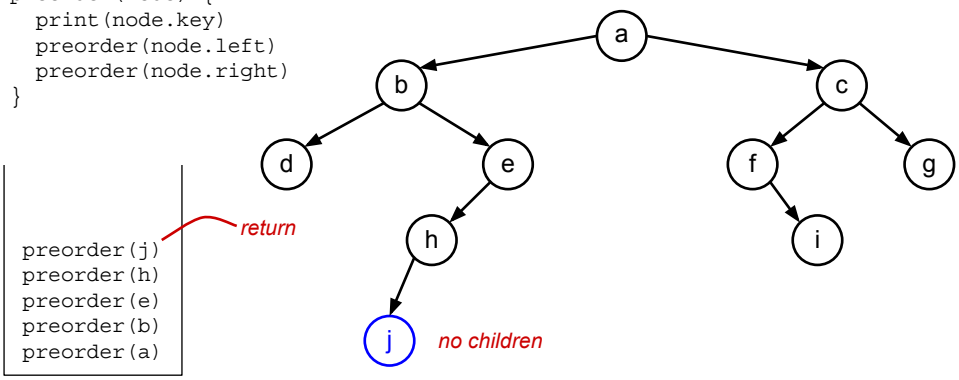


- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if a **pre-order traversal** is made?

a b d e h j

Binary tree traversals

```
preorder(node) {
  print(node.key)
  preorder(node.left)
  preorder(node.right)
}
```

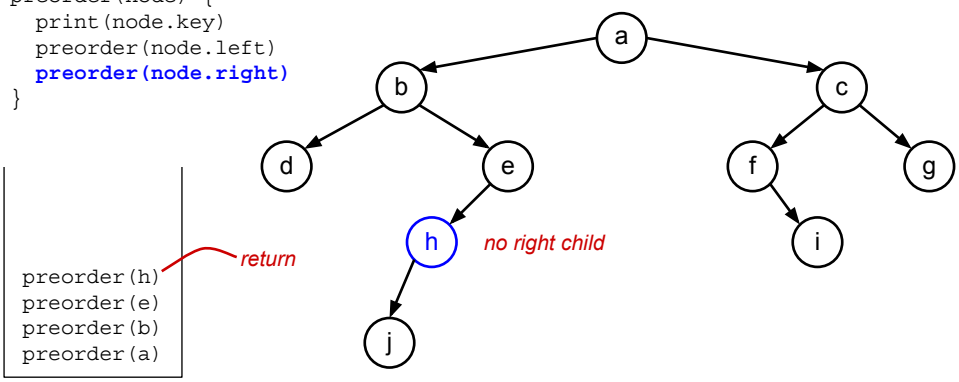


- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if a **pre-order traversal** is made?

a b d e h j

Binary tree traversals

```
preorder(node) {
  print(node.key)
  preorder(node.left)
  preorder(node.right)
}
```

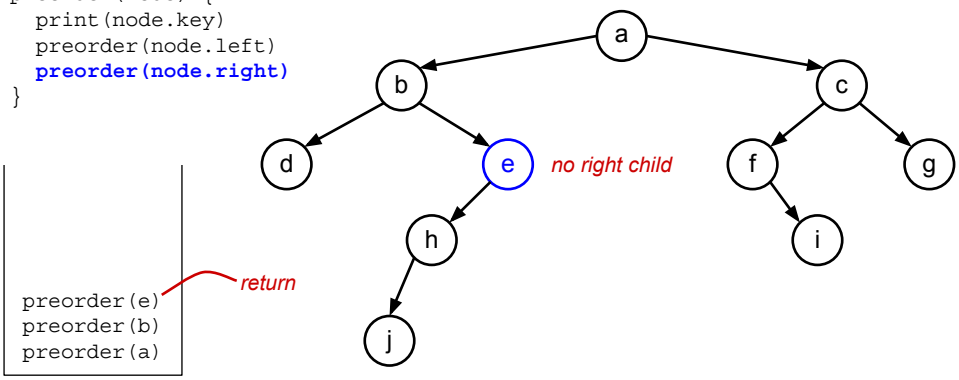


- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if a **pre-order traversal** is made?

a b d e h j

Binary tree traversals

```
preorder(node) {
  print(node.key)
  preorder(node.left)
  preorder(node.right)
}
```

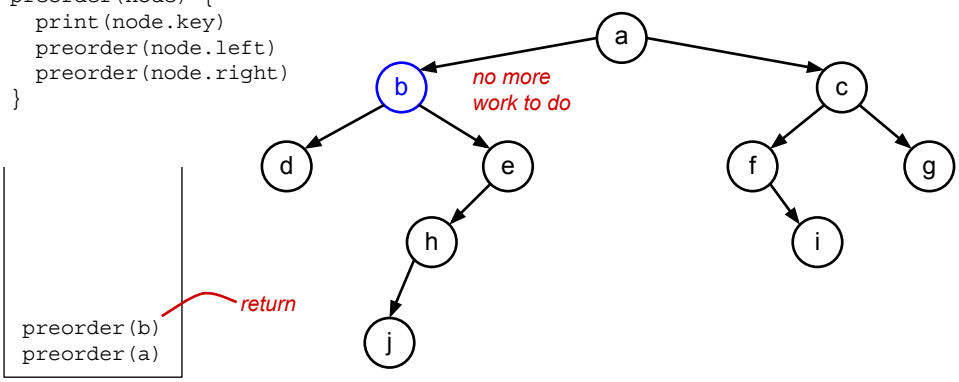


- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if a **pre-order traversal** is made?

a b d e h j

Binary tree traversals

```
preorder(node) {
  print(node.key)
  preorder(node.left)
  preorder(node.right)
}
```

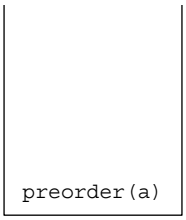


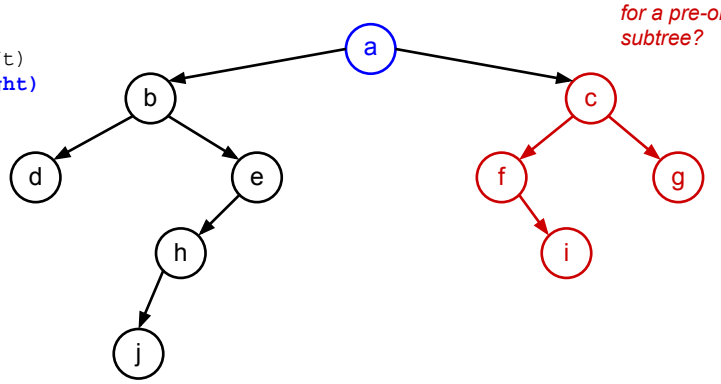
- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if a **pre-order traversal** is made?

a b d e h j

Binary tree traversals

```
preorder(node) {
  print(node.key)
  preorder(node.left)
  preorder(node.right)
}
```





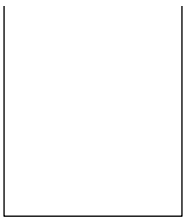
you do the rest—what's the output for a pre-order traversal on this subtree?

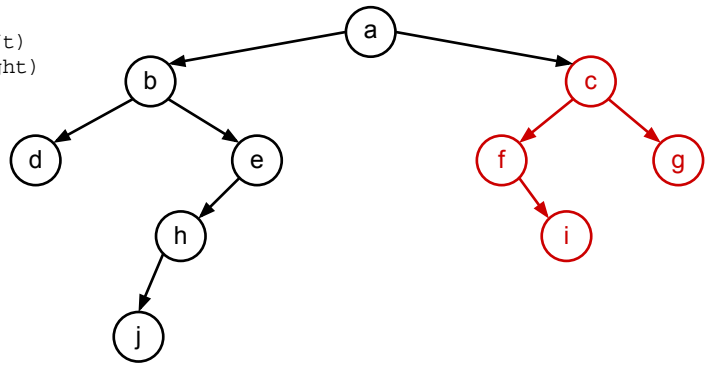
- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if a **pre-order traversal** is made?

a b d e h j

Binary tree traversals

```
preorder(node) {
  print(node.key)
  preorder(node.left)
  preorder(node.right)
}
```



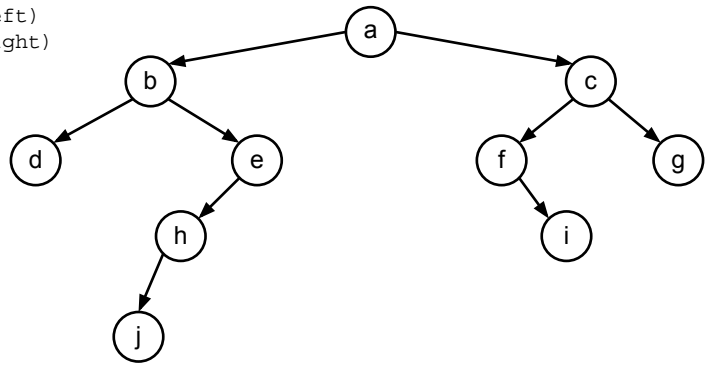


- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if a **pre-order traversal** is made?

a b d e h j c f i g

Binary tree traversals

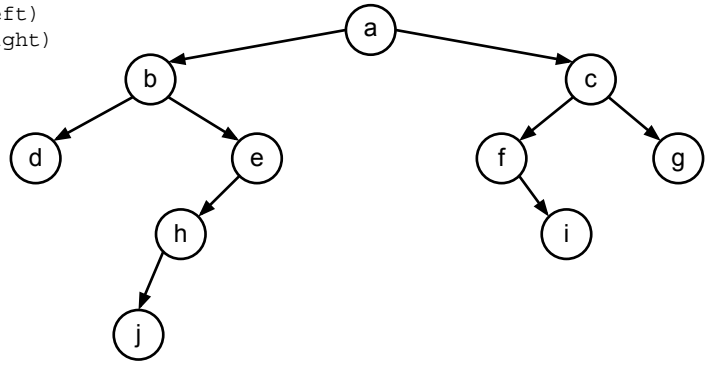
```
postorder(node) {
  postorder(node.left)
  postorder(node.right)
  print(node.key)
}
```



- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if a **post-order traversal** is made?

Binary tree traversals

```
postorder(node) {
  postorder(node.left)
  postorder(node.right)
  print(node.key)
}
```

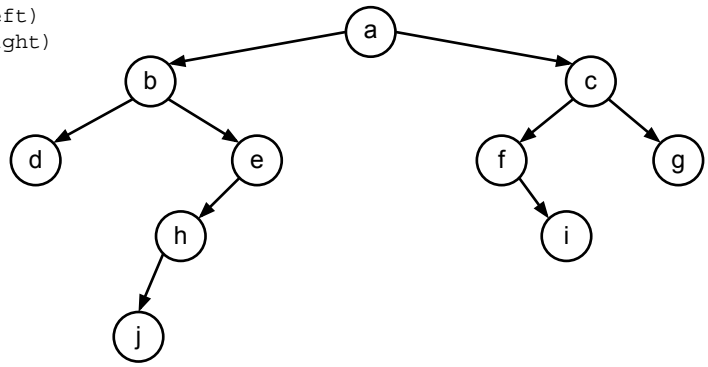


- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if a **post-order traversal** is made?

d

Binary tree traversals

```
postorder(node) {
  postorder(node.left)
  postorder(node.right)
  print(node.key)
}
```

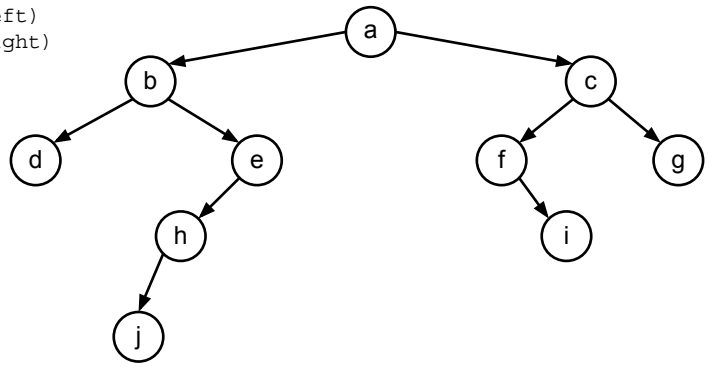


- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if a **post-order traversal** is made?

d j

Binary tree traversals

```
postorder(node) {
  postorder(node.left)
  postorder(node.right)
  print(node.key)
}
```

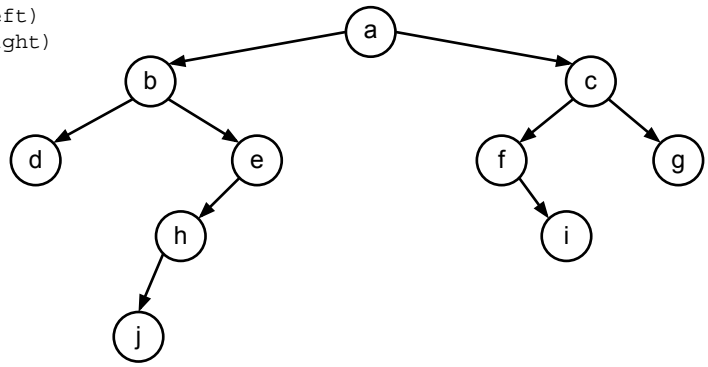


- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if a **post-order traversal** is made?

d j h

Binary tree traversals

```
postorder(node) {
  postorder(node.left)
  postorder(node.right)
  print(node.key)
}
```

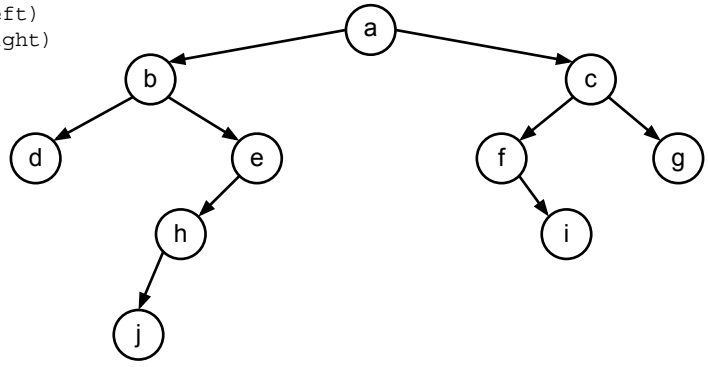


- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if a **post-order traversal** is made?

d j h e

Binary tree traversals

```
postorder(node) {
  postorder(node.left)
  postorder(node.right)
  print(node.key)
}
```

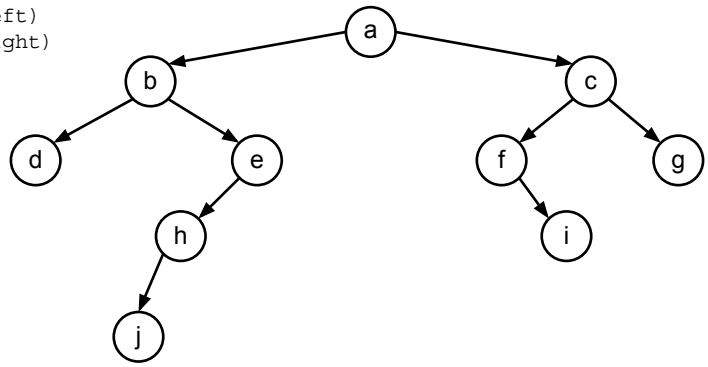


- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if a **post-order traversal** is made?

d j h e b

Binary tree traversals

```
postorder(node) {
  postorder(node.left)
  postorder(node.right)
  print(node.key)
}
```

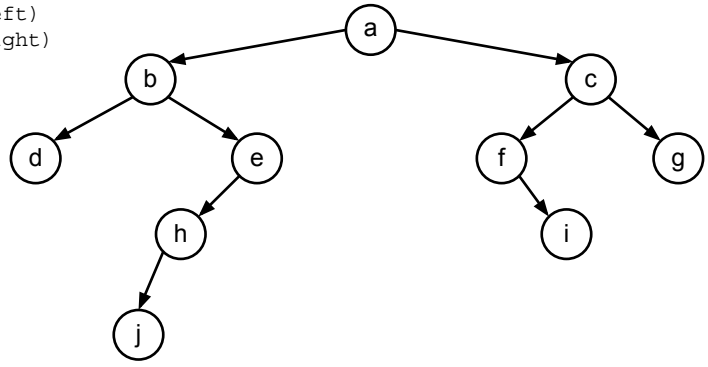


- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if a **post-order traversal** is made?

d j h e b i

Binary tree traversals

```
postorder(node) {
  postorder(node.left)
  postorder(node.right)
  print(node.key)
}
```

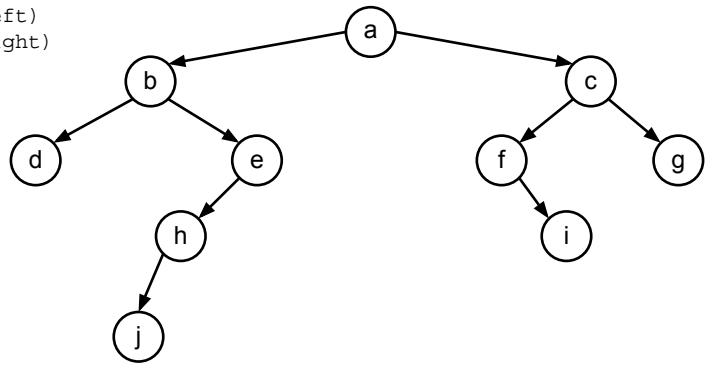


- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if a **post-order traversal** is made?

d j h e b i f

Binary tree traversals

```
postorder(node) {
  postorder(node.left)
  postorder(node.right)
  print(node.key)
}
```

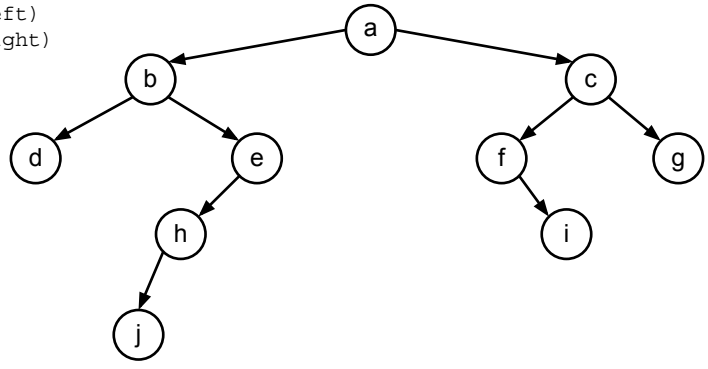


- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if a **post-order traversal** is made?

d j h e b i f g

Binary tree traversals

```
postorder(node) {
  postorder(node.left)
  postorder(node.right)
  print(node.key)
}
```

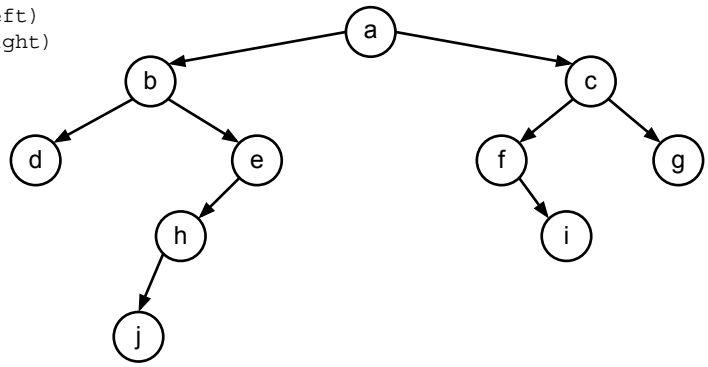


- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if a **post-order traversal** is made?

d j h e b i f g c

Binary tree traversals

```
postorder(node) {
  postorder(node.left)
  postorder(node.right)
  print(node.key)
}
```

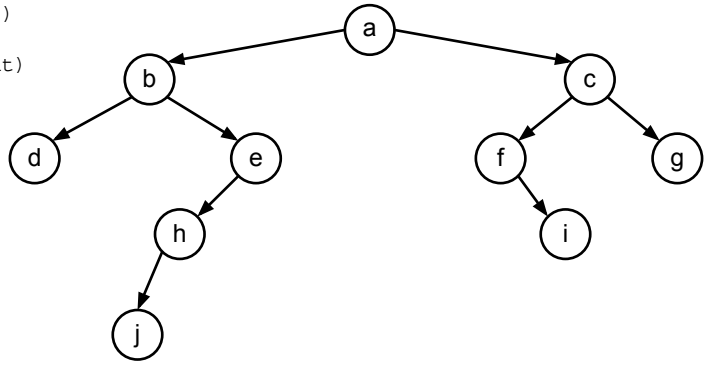


- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if a **post-order traversal** is made?

d j h e b i f g c a

Binary tree traversals

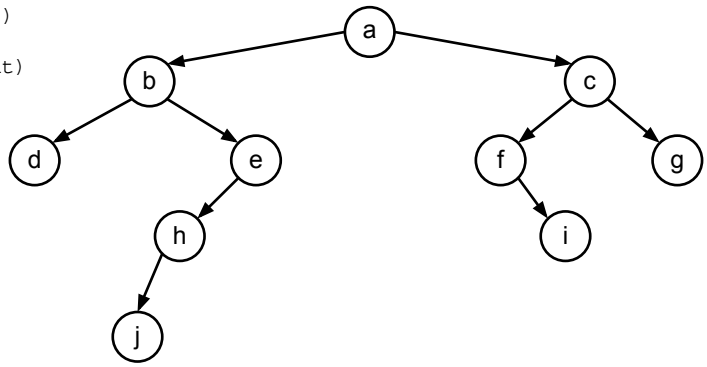
```
inorder(node) {
  inorder(node.left)
  print(node.key)
  inorder(node.right)
}
```



- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if an **in-order traversal** is made?

Binary tree traversals

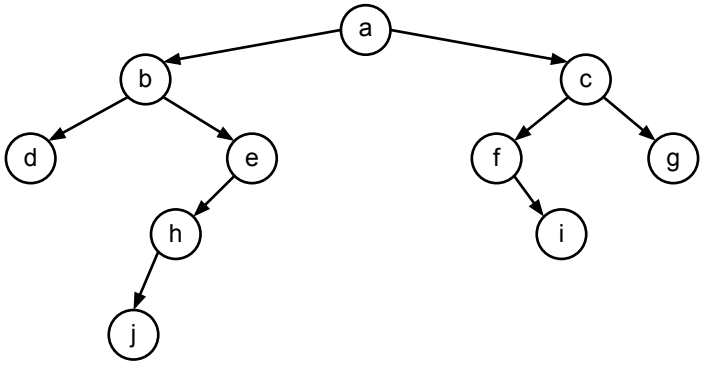
```
inorder(node) {
  inorder(node.left)
  print(node.key)
  inorder(node.right)
}
```



- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if an **in-order traversal** is made?

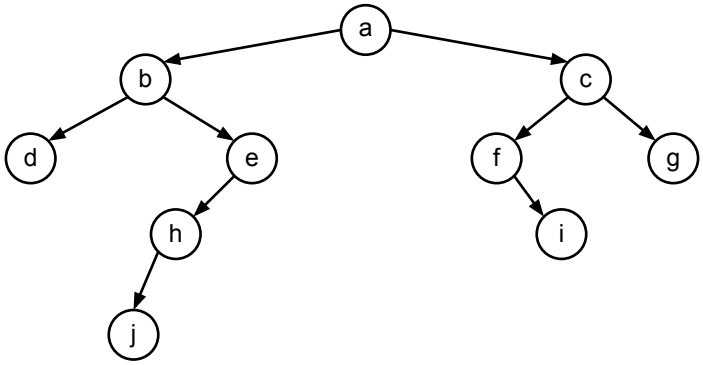
d b j h e a f i c g

Binary tree traversals



- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if a **level-order traversal** is made?

Binary tree traversals

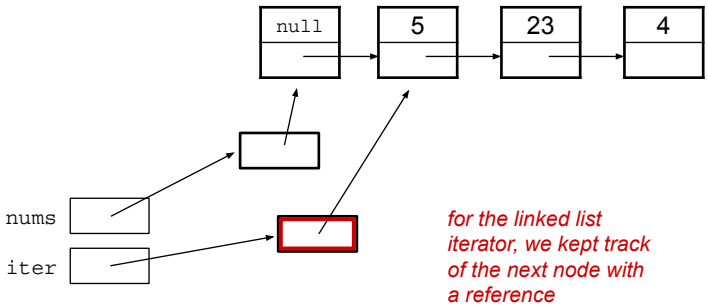


- We want to traverse the tree and print the key of a node when visited
- In what order would the keys be printed if a **level-order traversal** is made?

a b c d e f g h i j

Iterators for linked lists

```
public class LLList ... {
  private class Node {
    private Object item;
    private Node next;
  }

  private class LLListIterator ... {
    private Node nextNode;

    public LLListIterator() {
      nextNode = head.next;
    }

    public boolean hasNext() {
      return nextNode != null;
    }

    public Object next() {
      Object item = nextNode.item;
      nextNode = nextNode.next;
      return item;
    }
  }
}
```



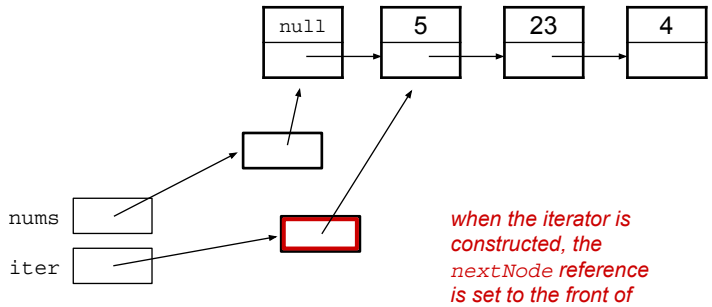
for the linked list iterator, we kept track of the next node with a reference

```
int[] numsArray = {5, 23, 4};
LLList nums = new LLList(numsArray);

ListIterator iter = nums.iterator();
System.out.println("printing all items...");

while (iter.hasNext()) {
  Object item = iter.next();
  System.out.println(item);
}
```

Iterators for linked lists

```
public class LLList ... {
  private class Node {
    private Object item;
    private Node next;
  }

  private class LLListIterator ... {
    private Node nextNode;

    public LLListIterator() {
      nextNode = head.next;
    }

    public boolean hasNext() {
      return nextNode != null;
    }

    public Object next() {
      Object item = nextNode.item;
      nextNode = nextNode.next;
      return item;
    }
  }
}
```

skip over dummy head node of linked list

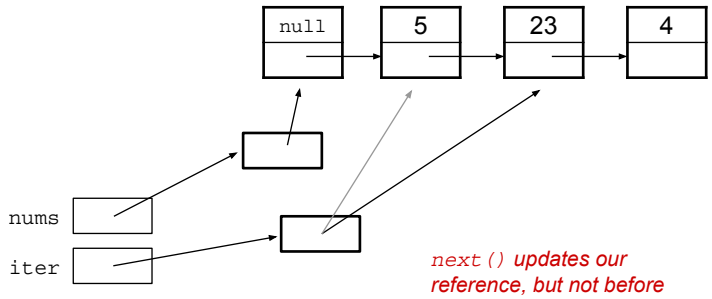


when the iterator is constructed, the `nextNode` reference is set to the front of the list

```
int[] numsArray = {5, 23, 4};
LLList nums = new LLList(numsArray);

ListIterator iter = nums.iterator();
System.out.println("printing all items...");

while (iter.hasNext()) {
  Object item = iter.next();
  System.out.println(item);
}
```

Iterators for linked lists

```
public class LLList ... {
  private class Node {
    private Object item;
    private Node next;
  }

  private class LLListIterator ... {
    private Node nextNode;

    public LLListIterator() {
      nextNode = head.next;
    }

    public boolean hasNext() {
      return nextNode != null;
    }

    public Object next() {
      Object item = nextNode.item;
      nextNode = nextNode.next;
      return item;
    }
  }
}
```



*`next()` updates our reference, but not before **saving the item** for returning*

```
int[] numsArray = {5, 23, 4};
LLList nums = new LLList(numsArray);

ListIterator iter = nums.iterator();
System.out.println("printing all items...");

while (iter.hasNext()) {
  Object item = iter.next();
  System.out.println(item);
}
```

Iterators for binary trees

- Just like linked list iterators, binary tree iterators give consecutive access to values in nodes
- Binary tree iterators should satisfy the following interface:

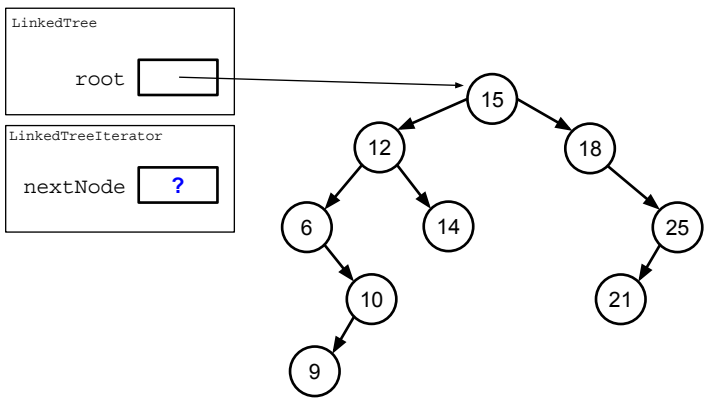
```
public interface LinkedTreeIterator {
  boolean hasNext();
  int next();          // assume tree stores integers
}
```

- Depending on the traversal, we will write a class that implements `LinkedTreeIterator` and express the logic of the traversal **in three places**: constructor, `hasNext()` and `next()`
- Like `LLList`, we implement the iterator as a private inner class

Iterators for binary trees

- Start by copying `nextNode` and the `hasNext()` code from the linked list iterator
- When an iterator is constructed, what node should `nextNode` point to?
  - In other words, which node is visited first in a pre-order traversal?

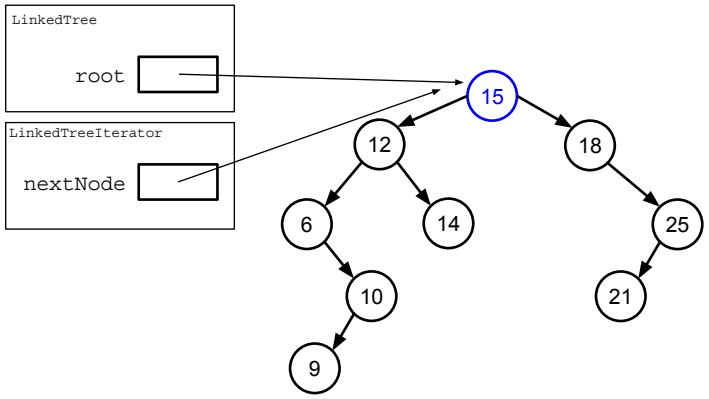
```
public class LinkedTree ... {
  private Node root;

  private class PreorderIterator ... {
    private Node nextNode;

    public PreorderIterator() {
      nextNode = ???
    }

    public boolean hasNext() {
      return nextNode != null;
    }

    public int next() {
      ???
    }
  }
}
```



Iterators for binary trees

- Start by copying `nextNode` and the `hasNext()` code from the linked list iterator
- When an iterator is constructed, what node should `nextNode` point to? root of the tree
  - In other words, which node is visited first in a pre-order traversal? here, the 15 node

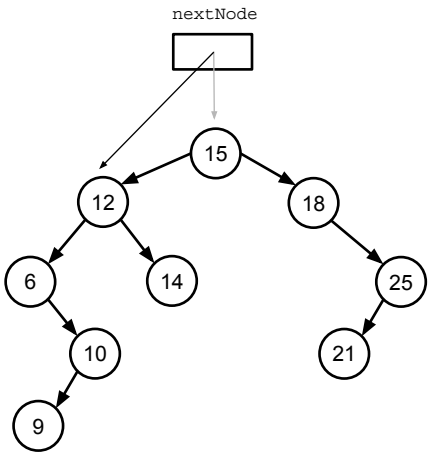
```
public class LinkedTree ... {
  private Node root;

  private class PreorderIterator ... {
    private Node nextNode;

    public PreorderIterator() {
      nextNode = root;
    }

    public boolean hasNext() {
      return nextNode != null;
    }

    public int next() {
      ???
    }
  }
}
```



Iterators for binary trees: the first call to `next()`

- For the tree on the right, the first two keys in the pre-order traversal are 15, then 12
- After an iterator is constructed, the first call to `next()` would need to create a variable for the key of the 15 node, then make `nextNode` point to the 12 node

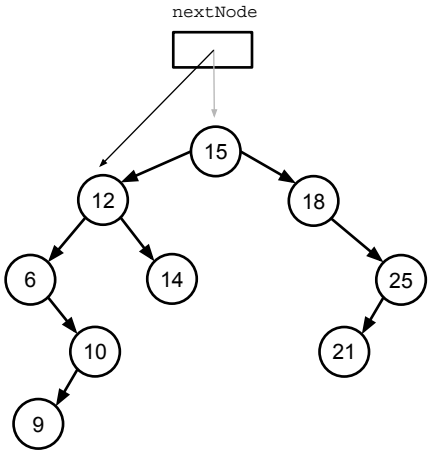
```
private class PreorderIterator ... {
  private Node nextNode;

  public int next() {

    ???

  }
}
```



Iterators for binary trees: the first call to `next()`

- For the tree on the right, the first two keys in the pre-order traversal are 15, then 12
- After an iterator is constructed, the first call to `next()` would need to create a variable for the key of the 15 node, then make `nextNode` point to the 12 node

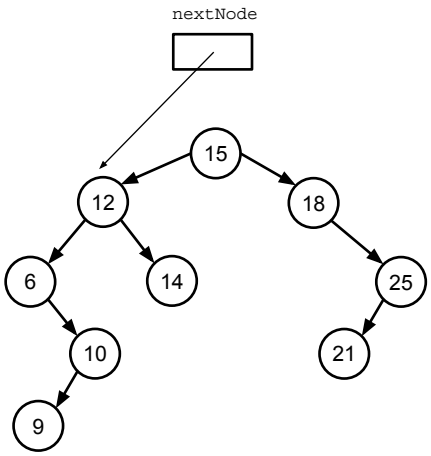
```
private class PreorderIterator ... {
  private Node nextNode;

  public int next() {
    int toReturn = nextNode.key;
    nextNode = nextNode.left;
    return toReturn;
  }
}
```



Iterators for binary trees: the first call to `next()`

- For the tree on the right, the first two keys in the pre-order traversal are 15, then 12
- After an iterator is constructed, the first call to `next()` would need to create a variable for the key of the 15 node, then make `nextNode` point to the 12 node

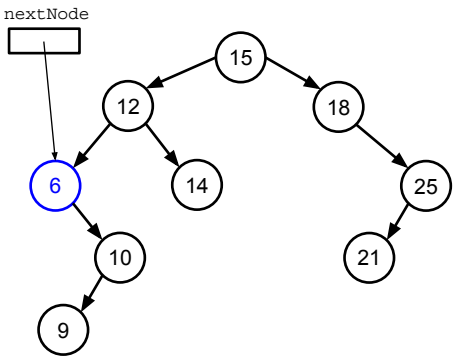
```
private class PreorderIterator ... {
  private Node nextNode;

  public int next() {
    int toReturn = nextNode.key;
    nextNode = nextNode.left;
    return toReturn;
  }
}
```

Iterators for binary trees: the first call to `next()`

- For the tree on the right, the first two keys in the pre-order traversal are 15, then 12
- After an iterator is constructed, the first call to `next()` would need to create a variable for the key of the 15 node, then make `nextNode` point to the 12 node

```
private class PreorderIterator ... {
  private Node nextNode;

  public int next() {
    int toReturn = nextNode.key;
    nextNode = nextNode.left;
    return toReturn;
  }
}
```

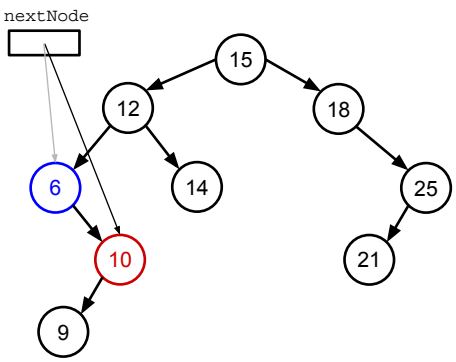


- This works for the first call to `next()`, but it is clear we need more cases—what should `next()` do when `nextNode` points to the 6 node?

Iterators for binary trees: the first call to `next()`

- For the tree on the right, the first two keys in the pre-order traversal are 15, then 12
- After an iterator is constructed, the first call to `next()` would need to create a variable for the key of the 15 node, then make `nextNode` point to the 12 node

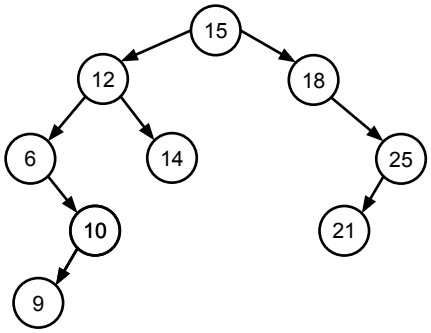
```
private class PreorderIterator ... {
  private Node nextNode;

  public int next() {
    int toReturn = nextNode.key;
    nextNode = nextNode.left;
    return toReturn;
  }
}
```



- This works for the first call to `next()`, but it is clear we need more cases—what should `next()` do when `nextNode` points to the 6 node?
  Since the 6 node has no left child, it should set `nextNode` to `nextNode.right`

Iterators for binary trees: improving `next()`

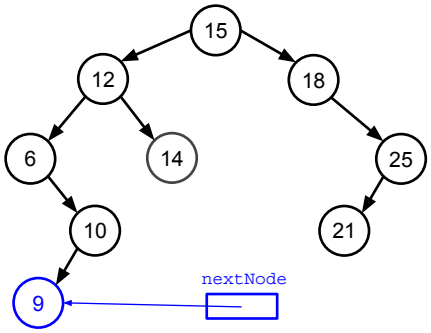
```
public int next() {
  int toReturn = nextNode.key;

  if (nextNode.left != null) {
    nextNode = nextNode.left;
  } else if (nextNode.right != null) {
    nextNode = nextNode.right;
  } else {
    ???
  }

  return toReturn;
}
```



Iterators for binary trees: improving `next()`

```
public int next() {
  int toReturn = nextNode.key;

  if (nextNode.left != null) {
    nextNode = nextNode.left;
  } else if (nextNode.right != null) {
    nextNode = nextNode.right;
  } else {
    ???
  }

  return toReturn;
}
```

What happens when we update `nextNode` to point to the 9 node? Where should `nextNode` point to next?

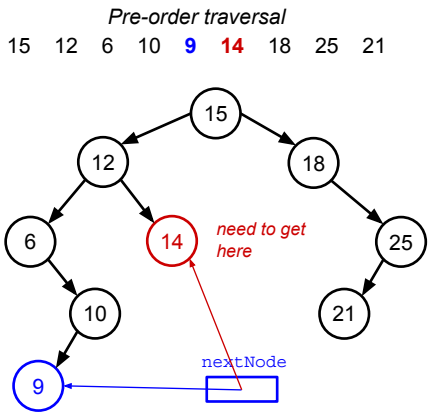


Iterators for binary trees: improving `next()`

```
public int next() {
  int toReturn = nextNode.key;

  if (nextNode.left != null) {
    nextNode = nextNode.left;
  } else if (nextNode.right != null) {
    nextNode = nextNode.right;
  } else {
    ???
  }

  return toReturn;
}
```

What happens when we update `nextNode` to point to the 9 node? Where should `nextNode` point to next?

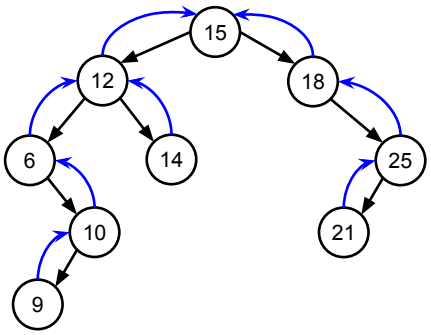
We will need a way to get from the 9 node up to **the 14 node**, the next key in the pre-order traversal!



Iterators for binary trees: `parent` references

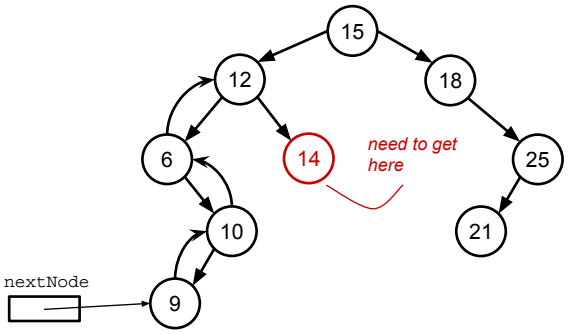
- To enable our binary tree iterator to reach all nodes starting from any node, we can add a `parent` reference to the inner `Node` class
- The methods of the `LinkedTree` class must be changed to properly update the `parent` reference (e.g., when inserting a node)

```
private class Node {
  private int key;
  private LLList data;
  private Node left;
  private Node right;
  private Node parent;
  ...
}
```



Iterators for binary trees: putting it together

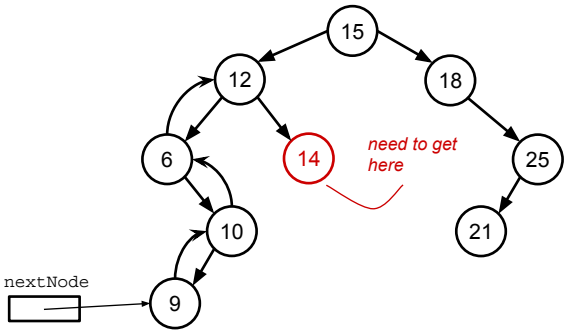
- When `nextNode` is a leaf node, we need to **use the `parent` references** to reach a node above us in the tree with a right child we have not visited yet
- We may need to traverse **more than one parent reference** to get to the node whose key is the next one in the pre-order traversal



Iterators for binary trees: putting it together

- When `nextNode` is a leaf node, we need to **use the `parent` references** to reach a node above us in the tree with a right child we have not visited yet
- We may need to traverse **more than one parent reference** to get to the node whose key is the next one in the pre-order traversal
- Here's one attempt:

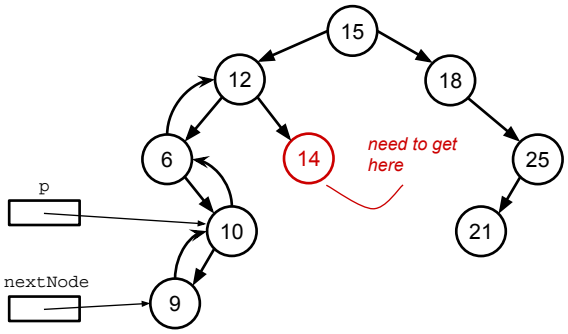
```
Node p = nextNode.parent;

while (p != null && p.right == null) {
  p = p.parent;
}

if (p == null) {
  nextNode = null;
} else {
  nextNode = p.right;
}
```



Iterators for binary trees: putting it together

- When `nextNode` is a leaf node, we need to **use the `parent` references** to reach a node above us in the tree with a right child we have not visited yet
- We may need to traverse **more than one parent reference** to get to the node whose key is the next one in the pre-order traversal
- Here's one attempt:

```
Node p = nextNode.parent;

while (p != null && p.right == null) {
  p = p.parent;
}

if (p == null) {
  nextNode = null;
} else {
  nextNode = p.right;
}
```

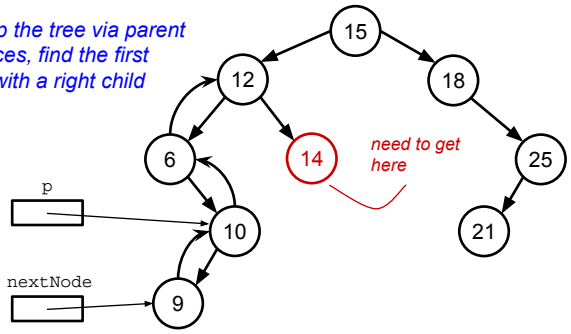


Iterators for binary trees: putting it together

- When `nextNode` is a leaf node, we need to **use the `parent` references** to reach a node above us in the tree with a right child we have not visited yet
- We may need to traverse **more than one parent reference** to get to the node whose key is the next one in the pre-order traversal
- Here's one attempt:

```
Node p = nextNode.parent;

while (p != null && p.right == null) {
  p = p.parent;
}

if (p == null) {
  nextNode = null;
} else {
  nextNode = p.right;
}
```

going up the tree via parent references, find the first parent with a right child



Iterators for binary trees: putting it together

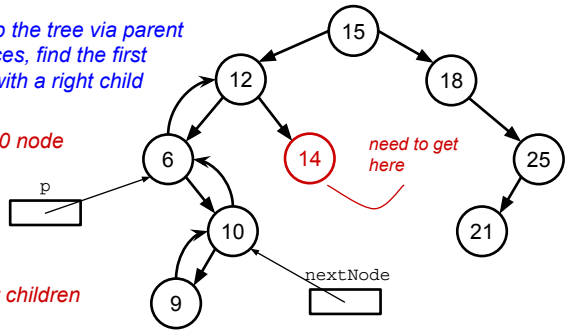
- When `nextNode` is a leaf node, we need to **use the `parent` references** to reach a node above us in the tree with a right child we have not visited yet
- We may need to traverse **more than one parent reference** to get to the node whose key is the next one in the pre-order traversal
- Here's one attempt:

```
Node p = nextNode.parent;

while (p != null && p.right == null) {
  p = p.parent;
}

if (p == null) {
  nextNode = null;
} else {
  nextNode = p.right;
}
```

going up the tree via parent references, find the first parent with a right child

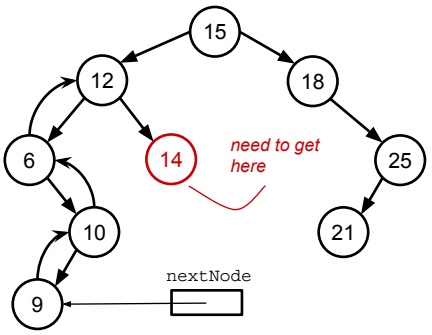
*this **will not** work! it finds the 10 node instead of the 14 node*

*we need to find **unvisited** right children*



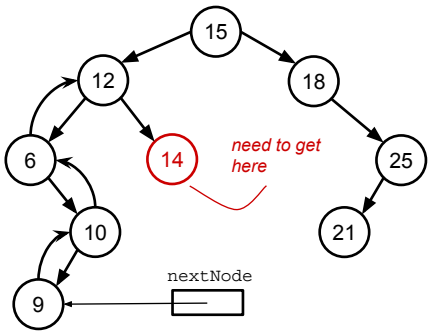

```
public int next() {
  int toReturn = nextNode.key;

  if (nextNode.left != null) {
      nextNode = nextNode.left;
  } else if (nextNode.right != null) {
      nextNode = nextNode.right;
  } else {
      Node p = nextNode.parent;
      Node c = nextNode;

      while (p != null && p.right == null || p.right == c) {
          c = p;
          p = p.parent;
      }

      if (p == null) {
          nextNode = null;
      } else {
          nextNode = p.right;
      }
  }

  return toReturn;
}
```

- Use two references: one to the parent, and one "behind"
- Allows us to find right children **we have not already visited**
- Stop the loop on **the first ancestor** whose right child is **not on our path** to the root



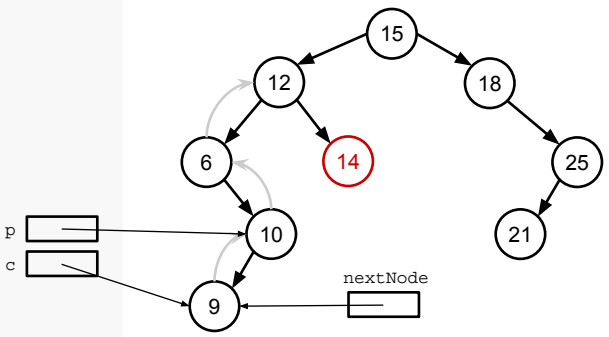
```
public int next() {
  int toReturn = nextNode.key;

  if (nextNode.left != null) {
      nextNode = nextNode.left;
  } else if (nextNode.right != null) {
      nextNode = nextNode.right;
  } else {
      Node p = nextNode.parent;
      Node c = nextNode;

      while (p != null && p.right == null || p.right == c) {
          c = p;
          p = p.parent;
      }

      if (p == null) {
          nextNode = null;
      } else {
          nextNode = p.right;
      }
  }

  return toReturn;
}
```

- Use two references: one to the parent, and one “behind”
- Allows us to find right children **we have not already visited**
- Stop the loop on **the first ancestor** whose right child is **not on our path** to the root



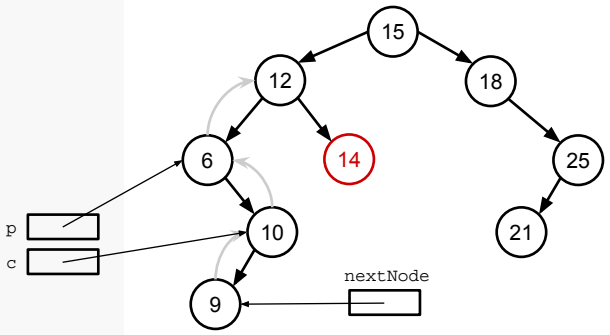

```
public int next() {
  int toReturn = nextNode.key;

  if (nextNode.left != null) {
      nextNode = nextNode.left;
  } else if (nextNode.right != null) {
      nextNode = nextNode.right;
  } else {
      Node p = nextNode.parent;
      Node c = nextNode;

      while (p != null && p.right == null || p.right == c) {
          c = p;
          p = p.parent;
      }

      if (p == null) {
          nextNode = null;
      } else {
          nextNode = p.right;
      }
  }

  return toReturn;
}
```

- Use two references: one to the parent, and one “behind”
- Allows us to find right children **we have not already visited**
- Stop the loop on **the first ancestor** whose right child is **not on our path** to the root



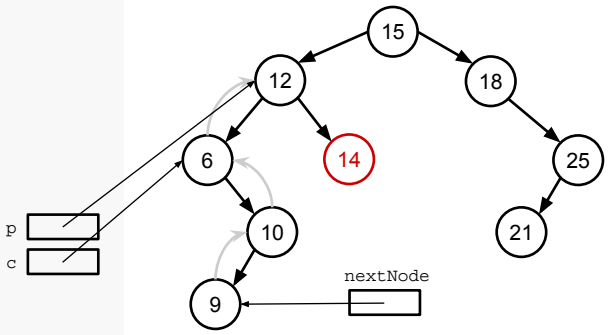
```
public int next() {
  int toReturn = nextNode.key;

  if (nextNode.left != null) {
      nextNode = nextNode.left;
  } else if (nextNode.right != null) {
      nextNode = nextNode.right;
  } else {
      Node p = nextNode.parent;
      Node c = nextNode;

      while (p != null && p.right == null || p.right == c) {
          c = p;
          p = p.parent;
      }

      if (p == null) {
          nextNode = null;
      } else {
          nextNode = p.right;
      }
  }

  return toReturn;
}
```

- Use two references: one to the parent, and one “behind”
- Allows us to find right children **we have not already visited**
- Stop the loop on **the first ancestor** whose right child is **not on our path** to the root



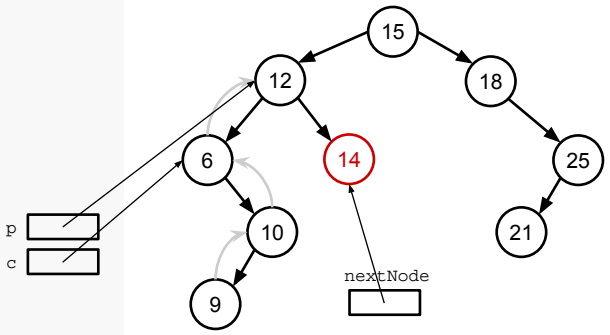

```
public int next() {
  int toReturn = nextNode.key;

  if (nextNode.left != null) {
      nextNode = nextNode.left;
  } else if (nextNode.right != null) {
      nextNode = nextNode.right;
  } else {
      Node p = nextNode.parent;
      Node c = nextNode;

      while (p != null && p.right == null || p.right == c) {
          c = p;
          p = p.parent;
      }

      if (p == null) {
          nextNode = null;
      } else {
          nextNode = p.right;
      }
  }

  return toReturn;
}
```

- Use two references: one to the parent, and one “behind”
- Allows us to find right children **we have not already visited**
- Stop the loop on **the first ancestor** whose right child is **not on our path** to the root



```
public int next() {
  int toReturn = nextNode.key;

  if (nextNode.left != null) {
      nextNode = nextNode.left;
  } else if (nextNode.right != null) {
      nextNode = nextNode.right;
  } else {
      Node p = nextNode.parent;
      Node c = nextNode;

      while (p != null && p.right == null || p.right == c) {
          c = p;
          p = p.parent;
      }

      if (p == null) {
          nextNode = null;
      } else {
          nextNode = p.right;
      }
  }

  return toReturn;
}
```

- Use two references: one to the parent, and one “behind”
- Allows us to find right children **we have not already visited**
- Stop the loop on **the first ancestor** whose right child is **not on our path** to the root

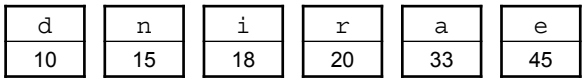


Huffman encoding

- We are given a document where all characters are drawn from a set of 6 characters, with the frequencies shown here
- Let’s create a Huffman tree from this table of frequencies and then use it to decode a binary string
- To create a Huffman tree, create nodes for each character, then, keeping the nodes in sorted order, repeatedly combining the two lowest-frequency nodes into a subtree

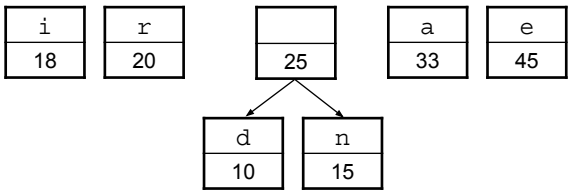
character	frequency
e	45
a	33
r	20
i	18
n	15
d	10

Huffman encoding



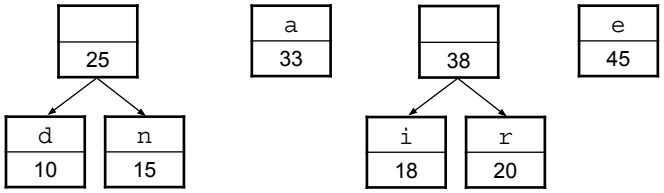
character	frequency
e	45
a	33
r	20
i	18
n	15
d	10

Huffman encoding



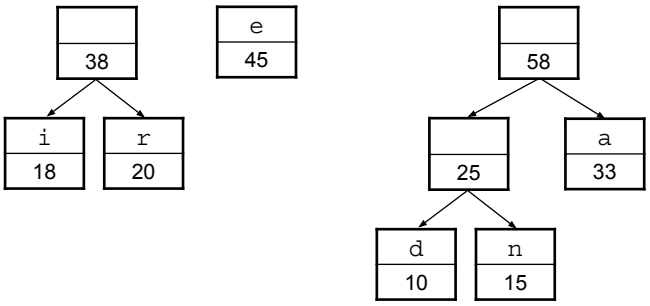
character	frequency
e	45
a	33
r	20
i	18
n	15
d	10

Huffman encoding



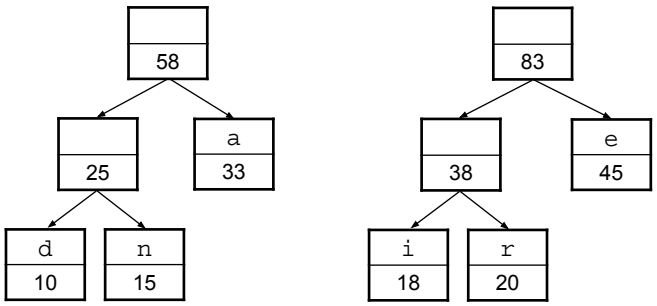
character	frequency
e	45
a	33
r	20
i	18
n	15
d	10

Huffman encoding



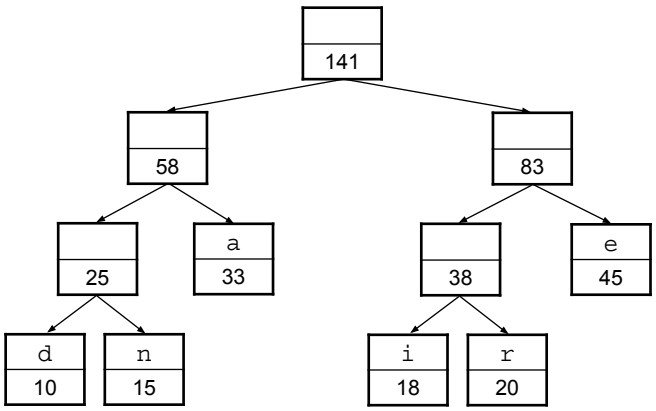
character	frequency
e	45
a	33
r	20
i	18
n	15
d	10

Huffman encoding



character	frequency
e	45
a	33
r	20
i	18
n	15
d	10

Huffman encoding

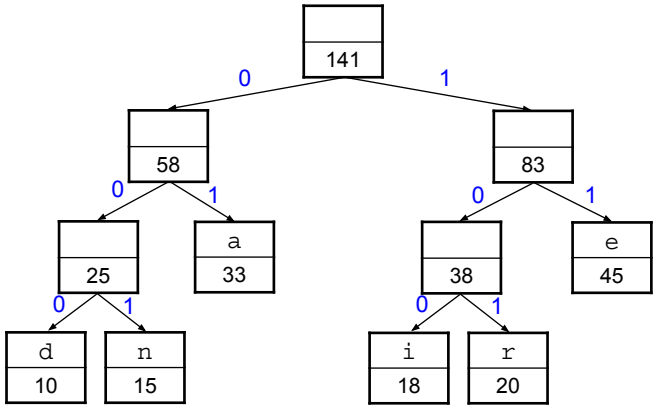


character	frequency
e	45
a	33
r	20
i	18
n	15
d	10

Huffman encoding

- Let's use the tree to decode the following binary string:

```
00101000100101
```



Huffman encoding

- Let's use the tree to decode the following binary string:

```
00101000100101
```

n a d i r