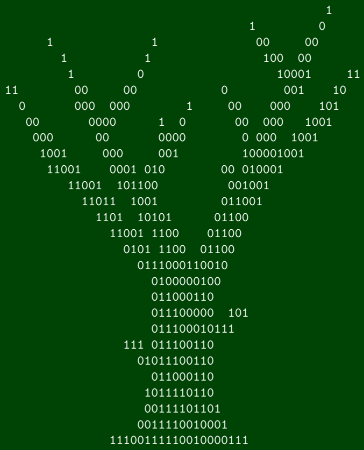
CS 2

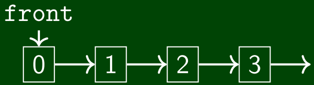
Introduction to Programming Methods

Trees


1
1 0
1 1 00 00
1 1 100 00
1 0 10001 11
11 00 00 0 001 10
0 000 000 1 00 000 101
00 0000 1 0 00 000 1001
000 00 0000 0 000 1001
1001 000 001 100001001
11001 0001 010 00 010001
11001 101100 001001
11011 1001 011001
1101 10101 01100
11001 1100 01100
0101 1100 01100
0111000110010
0100000100
011000110
011100000 101
011100010111
111 011100110
01011100110
011000110
1011110110
00111101101
0011110010001
11100111110010000111

Outline

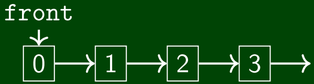
Consider the following standard `LinkedList`:



Recall the definition of a `ListNode`

```
public class Node {
   public int data;
   public Node next;

   public Node(int data, Node next) {
      this.data = data;
      this.next = next;
   }
}
```

Consider the following standard `LinkedList`:



Recall the definition of a `ListNode`

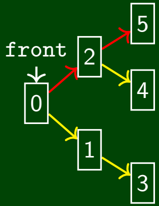
```java
public class Node {
   public int data;
   public Node next;

   public Node(int data, Node next) {
      this.data = data;
      this.next = next;
   }
}
```

What if we added more fields?

- Multiple data fields?
- Multiple “next” fields?

Nodes with Multiple next Fields

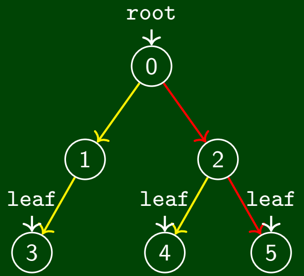
```
public class Node {
   public int data;
   public Node next1;
   public Node next2;

   public Node(int data, Node next1, Node next2) {
      this.data = data;
      this.next1 = next1;
      this.next2 = next2;
   }
}
```



(yellow is next2; red is next1)

Binary Trees

```
public class Node {
   public int data;
   public Node left;
   public Node right;

   public Node(int data, Node left, Node right) {
      this.data = data;
      this.left = left;
      this.right = right;
   }
}
```



(red is right; yellow is left)

Back To `LinkedLists`

Consider the following `LinkedList` of a mathematical expression:

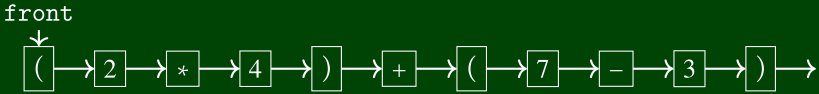
`front`

$$\downarrow$$

$$\boxed{(} \rightarrow \boxed{2} \rightarrow \boxed{*} \rightarrow \boxed{4} \rightarrow \boxed{)} \rightarrow \boxed{+} \rightarrow \boxed{(} \rightarrow \boxed{7} \rightarrow \boxed{-} \rightarrow \boxed{3} \rightarrow \boxed{)} \rightarrow$$

What's bad about it?

Consider the following LinkedList of a mathematical expression:

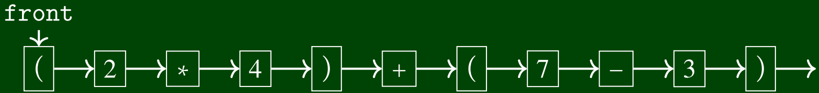


What's bad about it?

- It doesn't really help us with the structure
- Looking at it doesn't really show us what's going on

Back To `LinkedLists`

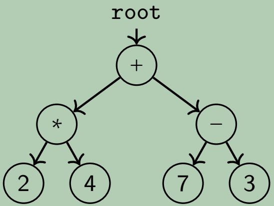
Consider the following `LinkedList` of a mathematical expression:



What's bad about it?

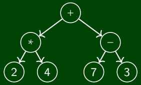
- It doesn't really help us with the structure
- Looking at it doesn't really show us what's going on

What about this structure instead?



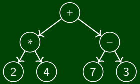
Now we can see the order of operations much more clearly!

- Parsing (Programming Languages, Math, etc.)

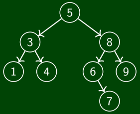


Uses of Trees

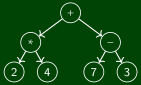
- Parsing (Programming Languages, Math, etc.)



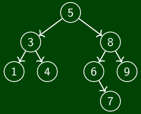
- Implementing `TreeSet`



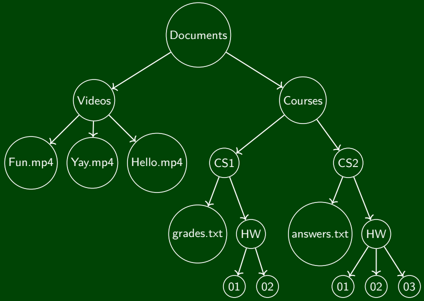
- Parsing (Programming Languages, Math, etc.)



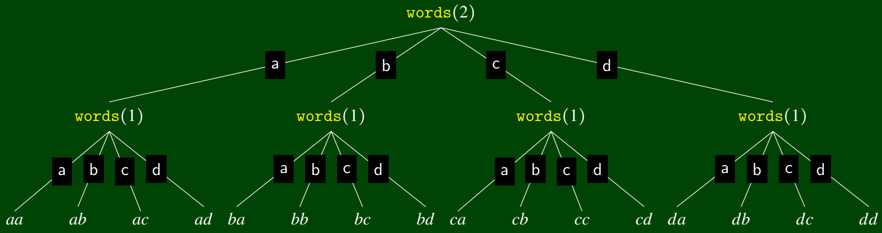
- Implementing `TreeSet`



- Directory File Structure



- Recursive Trees (including things like games of Tic-Tac-Toe)



Printing A LinkedList (Again)

```java
public void print() {
   Node current = this.front;
   while (current != null) {
      System.out.print(current.data + " ");
      current = current.next;
   }
}
```

We'd like to figure out how to print trees. Since `LinkedLists` are "simpler versions of trees", they might help.

Printing A `LinkedList` (Again)

```
public void print() {
   Node current = this.front;
   while (current != null) {
      System.out.print(current.data + " ");
      current = current.next;
   }
}
```

We'd like to figure out how to print trees. Since `LinkedList`s are "simpler versions of trees", they might help.

How do we go in every direction in a tree?

USE RECURSION!

To print a `LinkedList`...

- Print the **front** of the list
- Print the **next** of the list (recursively)

Code

```
public void print() {
   print(this.front);
}

public void print(Node c) {
   if (c != null) {
      System.out.print(c.data + " ");
      print(c.next);
   }
}
```

To print a `BinaryTree`...

- Print the **root** of the tree
- Print the **left** of the tree (recursively)
- Print the **right** of the tree (recursively)

Code

```
public void print() {
   print(this.root);
}

public void print(Node c) {
   if (c != null) {
      System.out.print(c.data + " ");
      print(c.left);
      print(c.right);
   }
}
```

Binary Tree `method`

Binary Tree methods are just normal recursive functions. The base case/recursive calls will always be similar.

Writing a Binary Tree Method

- The base case is `current == null`.
- First recursive case is `method(current.left)`
- Second recursive case is `method(current.right)`

```
public type method(...) {
   return method(this.root, ...);
}
private type method(TreeNode current, ...) {
   if (current == null) { /* DO BASE CASE */ }

   // Do the left recursive case:
   type leftResult = method(current.left, ...);

   // Do the right recursive case:
   type rightResult = method(current.right, ...);

   /* Use the left and right results... */
   return ...;
}
```

`contains()`

Write a method, in the `IntTree` class, called `contains()`:

```
public boolean contains(int value);
```

that returns true if the tree contains `value` and false otherwise.

contains()

Write a method, in the IntTree class, called contains():

```
public boolean contains(int value);
```

that returns true if the tree contains value and false otherwise.

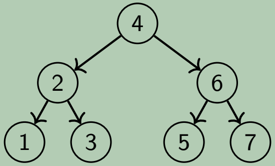
```
public boolean contains(int value) {
   return contains(this.root, value);
}
private boolean contains(Node current, int value) {
   /* If the tree is null, it definitely doesn't contain value... */
   if (current == null) { return false; }

   /* If current *is* value, we found it! */
   else if (current.data == value) { return true; }

   else {
      boolean leftContainsValue = contains(current.left, value);
      boolean rightContainsValue = contains(current.right, value);
      return leftContainsValue || rightContainsValue;
   }
}
```

Recall contains()

```
private boolean contains(IntTreeNode current, int value) {
   /* If the tree is null, it definitely doesn't contain value... */
   if (current == null) { return false; }

   /* If current *is* value, we found it! */
   else if (current.data == value) { return true; }

   else {
      return contains(current.left, value) ||
             contains(current.right, value);
   }
}
```

Runtime of contains(7)

Consider the following tree:



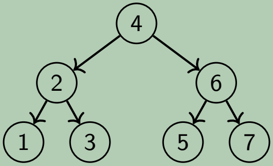
Back to contains

Recall contains()

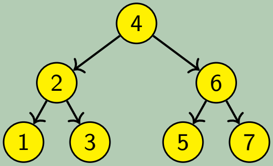
```java
private boolean contains(IntTreeNode current, int value) {
   /* If the tree is null, it definitely doesn't contain value... */
   if (current == null) { return false; }

   /* If current *is* value, we found it! */
   else if (current.data == value) { return true; }

   else {
      return contains(current.left, value) ||
             contains(current.right, value);
   }
}
```

Runtime of contains(7)

Consider the following tree:



Which nodes do we visit for contains(**7**)

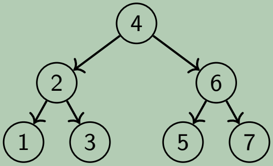


Recall `contains()`

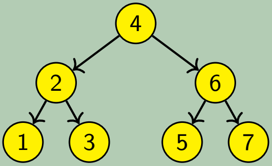
```
private boolean contains(IntTreeNode current, int value) {
   /* If the tree is null, it definitely doesn't contain value... */
   if (current == null) { return false; }

   /* If current *is* value, we found it! */
   else if (current.data == value) { return true; }

   else {
      return contains(current.left, value) ||
               contains(current.right, value);
   }
}
```

Runtime of `contains(7)`

Consider the following tree:



Which nodes do we visit for `contains`(**7**)



That makes the code $\mathcal{O}(n)$. Can we do better?

In general, **we can't do better**. BUT, sometimes, we can!

Definition (Binary SEARCH Tree (BST))

A binary tree is a **BST** when an **in-order traversal of the tree** yields a sorted list.

In general, **we can't do better**. BUT, sometimes, we can!

Definition (Binary SEARCH Tree (BST))

A binary tree is a **BST** when an **in-order traversal of the tree** yields a sorted list.

To put it another way, a binary tree is a **BST** when:

- All data "to the left of" a node is less than it
- All data "to the right of" a node is greater than it
- All sub-trees of the binary tree are also BSTs

Example (Which of the following are BSTs?)

In general, **we can't do better**. BUT, sometimes, we can!

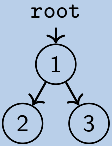
Definition (Binary SEARCH Tree (BST))

A binary tree is a **BST** when an **in-order traversal of the tree** yields a sorted list.

To put it another way, a binary tree is a **BST** when:

- All data "to the left of" a node is less than it
- All data "to the right of" a node is greater than it
- All sub-trees of the binary tree are also BSTs

Example (Which of the following are BSTs?)



In general, **we can't do better**. BUT, sometimes, we can!

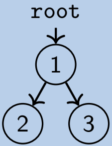
Definition (Binary SEARCH Tree (BST))

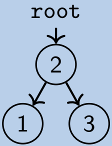
A binary tree is a **BST** when an **in-order traversal of the tree** yields a sorted list.

To put it another way, a binary tree is a **BST** when:

- All data "to the left of" a node is less than it
- All data "to the right of" a node is greater than it
- All sub-trees of the binary tree are also BSTs

Example (Which of the following are BSTs?)





In general, **we can't do better**. BUT, sometimes, we can!

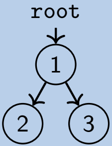
Definition (Binary **SEARCH** Tree (BST))

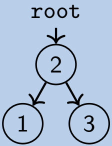
A binary tree is a **BST** when an **in-order traversal of the tree** yields a sorted list.

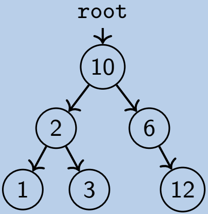
To put it another way, a binary tree is a **BST** when:

- All data "to the left of" a node is less than it
- All data "to the right of" a node is greater than it
- All sub-trees of the binary tree are also BSTs

Example (Which of the following are BSTs?)







Doing Better!

In general, **we can't do better**. BUT, sometimes, we can!

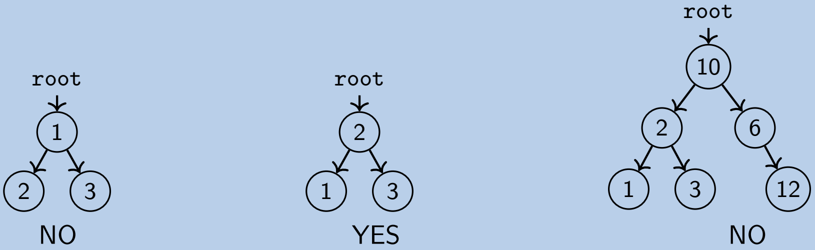
Definition (Binary SEARCH Tree (BST))

A binary tree is a **BST** when an **in-order traversal of the tree** yields a sorted list.

To put it another way, a binary tree is a **BST** when:

- All data "to the left of" a node is less than it
- All data "to the right of" a node is greater than it
- All sub-trees of the binary tree are also BSTs

Example (Which of the following are BSTs?)



Write `contains()` for a BST

Fix `contains` so that it takes advantage of the BST properties.

Write contains() for a BST

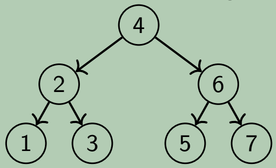
Fix contains so that it takes advantage of the BST properties.

Recall contains()

```
private boolean contains(IntTreeNode current, int value) {
   /* If the tree is null, it definitely doesn't contain value... */
   if (current == null) { return false; }

   /* If current *is* value, we found it! */
   else if (current.data == value) { return true; }

   else if (current.data < value) {
      return contains(current.right, value);
   }
   else {
      return contains(current.left, value);
   }
}
```

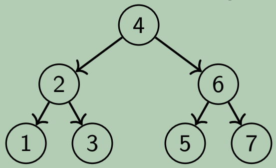
Runtime of (better) `contains(7)`

Consider the following tree:

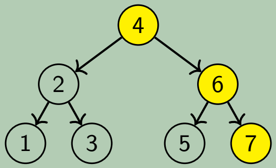


Runtime of (better) `contains(7)`

Consider the following tree:

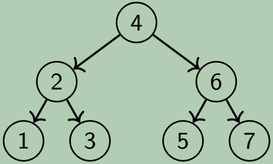


Which nodes do we visit for `contains`(**7**)

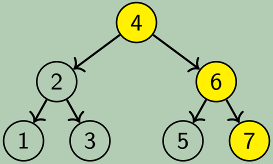


Runtime of (better) `contains(7)`

Consider the following tree:



Which nodes do we visit for `contains`(**7**)

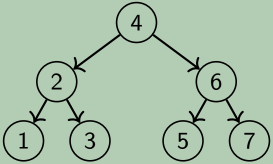


That makes the code $\log n$. Much better!

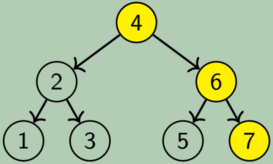
WARNING!

Runtime of (better) `contains(7)`

Consider the following tree:



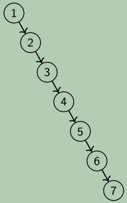
Which nodes do we visit for `contains`(**7**)



That makes the code $\log n$. Much better!

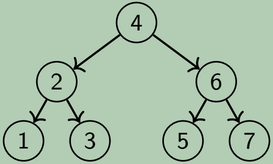
WARNING!

Consider the following tree:

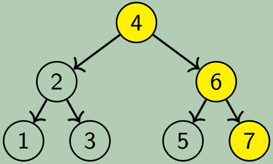


Runtime of (better) `contains(7)`

Consider the following tree:



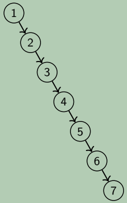
Which nodes do we visit for `contains`(**7**)



That makes the code $\log n$. Much better!

WARNING!

Consider the following tree:



This is the same tree, but now **we have to visit all the nodes!**

Adding to a BST!

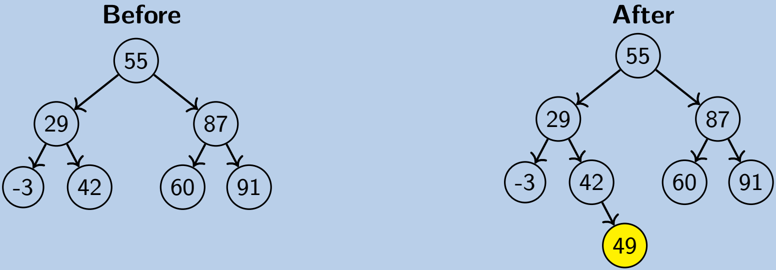
`add`

Write a method `add` in the BST class with the following signature:

```
public void add(int value);
```

that preserves the BST property.

Example (`tree.add(49)`)



Adding to a BST (Attempt #1)

Attempt #1

```java
public void add(int value) {
   add(this.root, value);
}
private void add(IntTreeNode current, int value) {
   if (current == null) {
      current = new IntTreeNode(value);
   }
   else if (current.data > value) {
      add(current.left, value);
   }
   else if (current.data < value) {
      add(current.right, value);
   }
}
```

What's wrong with this solution?

Just like with `LinkedLists` where we must change `front` or `.next`, we're not actually changing anything here. We're discarding the result.

Consider the following code:

```
public static void main(String[] args) {
   String s = "hello world";
   s.toUpperCase();
   System.out.println(s);
}
```

Consider the following code:

```
public static void main(String[] args) {
   String s = "hello world";
   s.toUpperCase();
   System.out.println(s);
}
```

OUTPUT

```
>> hello world
```

Consider the following code:

```
public static void main(String[] args) {
    String s = "hello world";
    s.toUpperCase();
    System.out.println(s);
}
```

OUTPUT

```
>> hello world
```

```
public static void main(String[] args) {
    String s = "hello world";
    s = s.toUpperCase();
    System.out.println(s);
}
```

x = change(x)

Consider the following code:

```
public static void main(String[] args) {
   String s = "hello world";
   s.toUpperCase();
   System.out.println(s);
}
```

OUTPUT

```
>> hello world
```

```
public static void main(String[] args) {
   String s = "hello world";
   s = s.toUpperCase();
   System.out.println(s);
}
```

OUTPUT

```
>> HELLO WORLD
```

We must `USE` the result; otherwise, it gets discarded

x = change(x)

If you want to write a method that can change the object that a variable refers to, you must do three things:

1. Pass in the original state of the object to the method
2. Return the new (possibly changed) object from the method
3. Re-assign the caller's variable to store the returned result

```
p = change(p); // in main
public static Point change(Point thePoint) {
   thePoint = new Point(99, -1);
   return thePoint;
}
```

Adding to a BST (Fixed)

Fixed Attempt

```
public void add(int value) {
   this.root = add(this.root, value);
}
private IntTreeNode add(IntTreeNode current, int value) {
   if (current == null) {
      current = new IntTreeNode(value);
   }
   else if (current.data > value) {
      current.left = add(current.left, value);
   }
   else if (current.data < value) {
      current.right = add(current.right, value);
   }
   return current;
}
```

This works because we **always update the result**, **always return the result**, and **always update the root**.