CSE 143
Lecture 17

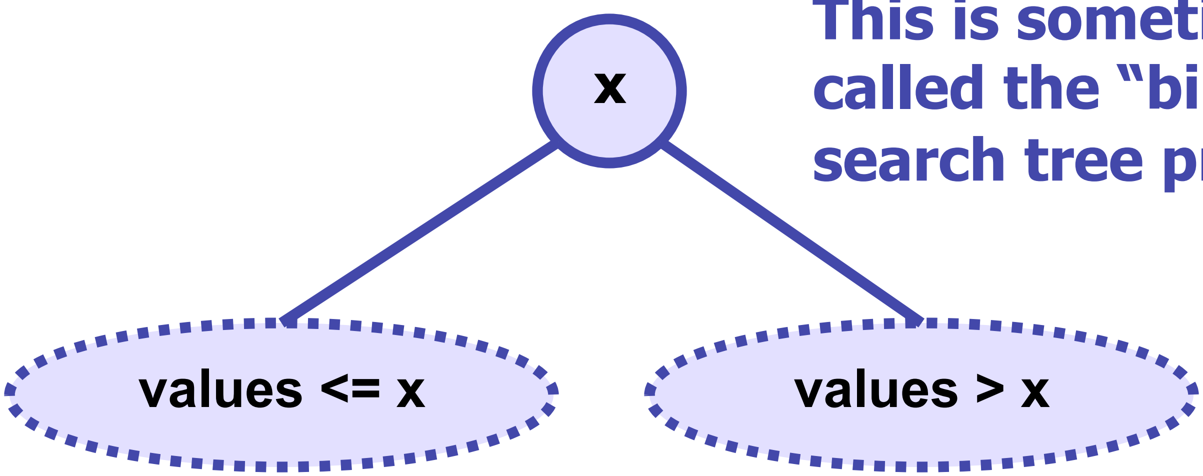
Binary Search Trees
and
Comparable

slides created by Ethan Apter
http://www.cs.washington.edu/143/

Binary Search Tree (BST)

- Binary search tree: a binary tree on which you can perform binary search

- For every subtree in a binary search tree, the following property holds:



This is sometimes called the "binary search tree property"

- Remember: this property holds for *every subtree*, not just for the overall root

Duplicate Values

- We must handle duplicates somehow

- We're going to handle duplicates by putting them in the left subtree (as shown on the previous slide)

- But there are also other options:
  - we could choose to not allow duplicates in our tree
  - we could choose to put the duplicates in the right subtree

- It doesn't really matter which one we choose, so long as we're consistent

BST of Names

- Let's create a BST of names (Strings)

- How will we compare one name to another?

- One name is *less than* another when the former comes before the latter alphabetically
  - so "A" < "B"

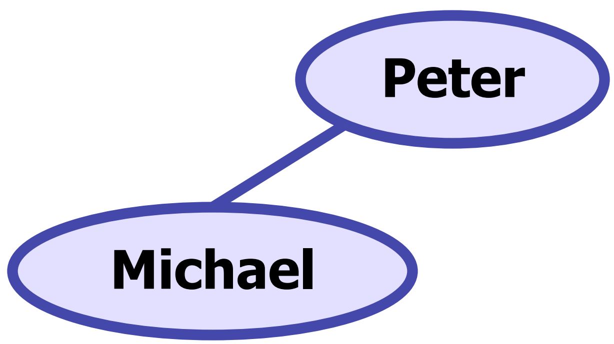
- One name is *greater than* another when the former comes after the latter alphabetically
  - so "B" > "A"

BST of Names

- Let's start with an empty tree and add the following names in the given order:
  - Peter, Michael, Kim, Morgan, Baron, and Toby

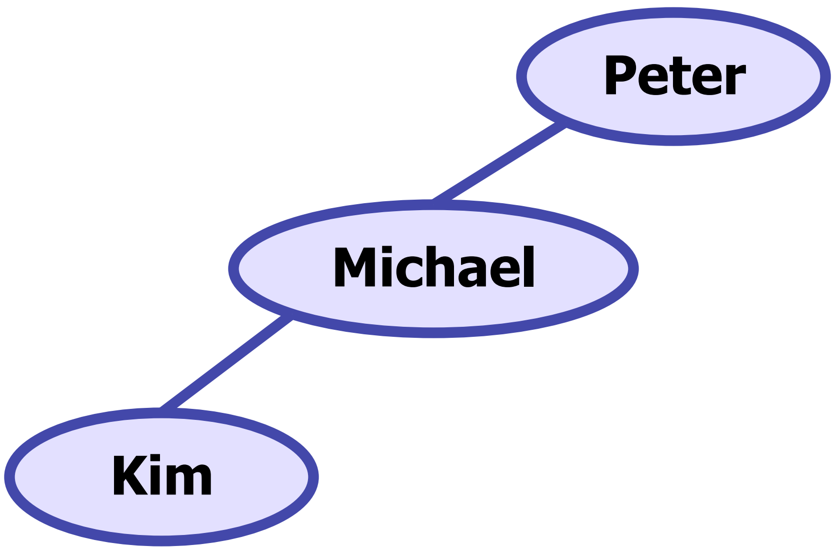
BST of Names

- Let's start with an empty tree and add the following names in the given order:
  - Peter, Michael, Kim, Morgan, Baron, and Toby



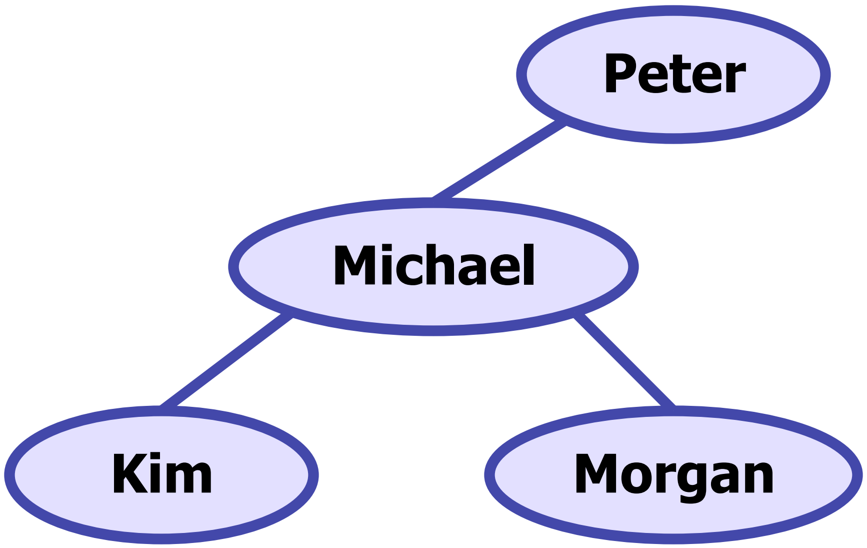
BST of Names

- Let's start with an empty tree and add the following names in the given order:
  - Peter, Michael, Kim, Morgan, Baron, and Toby



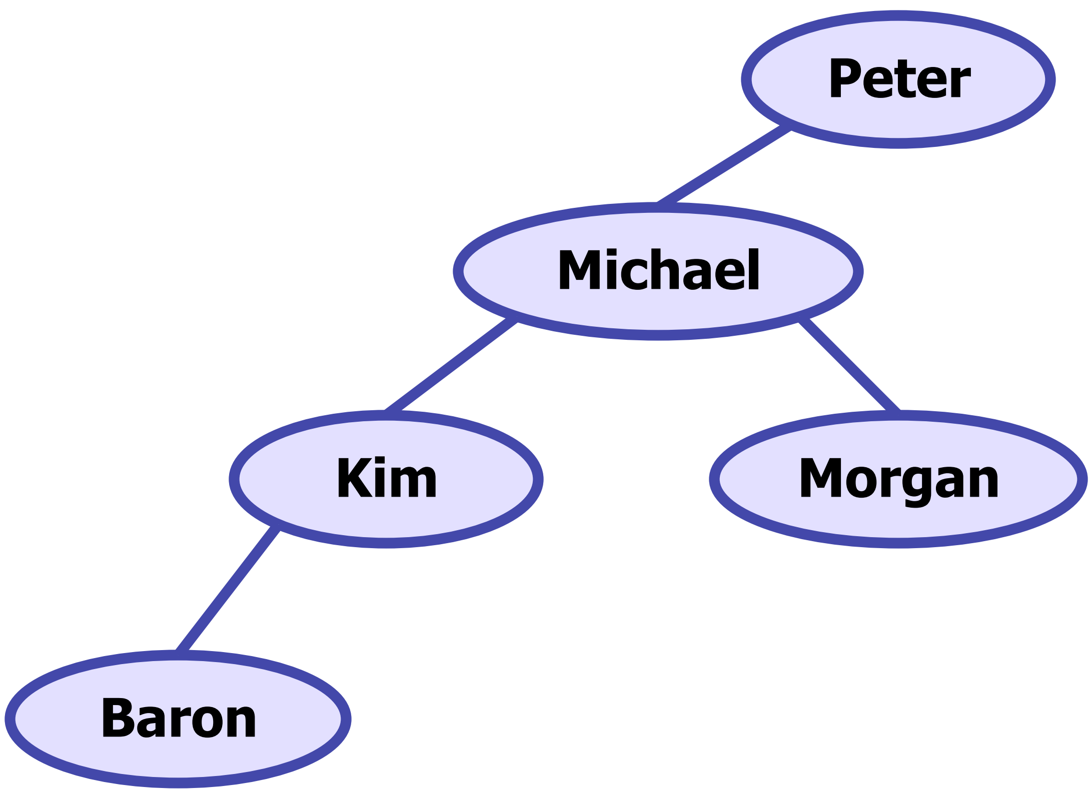
BST of Names

- Let’s start with an empty tree and add the following names in the given order:
  - Peter, Michael, Kim, Morgan, Baron, and Toby



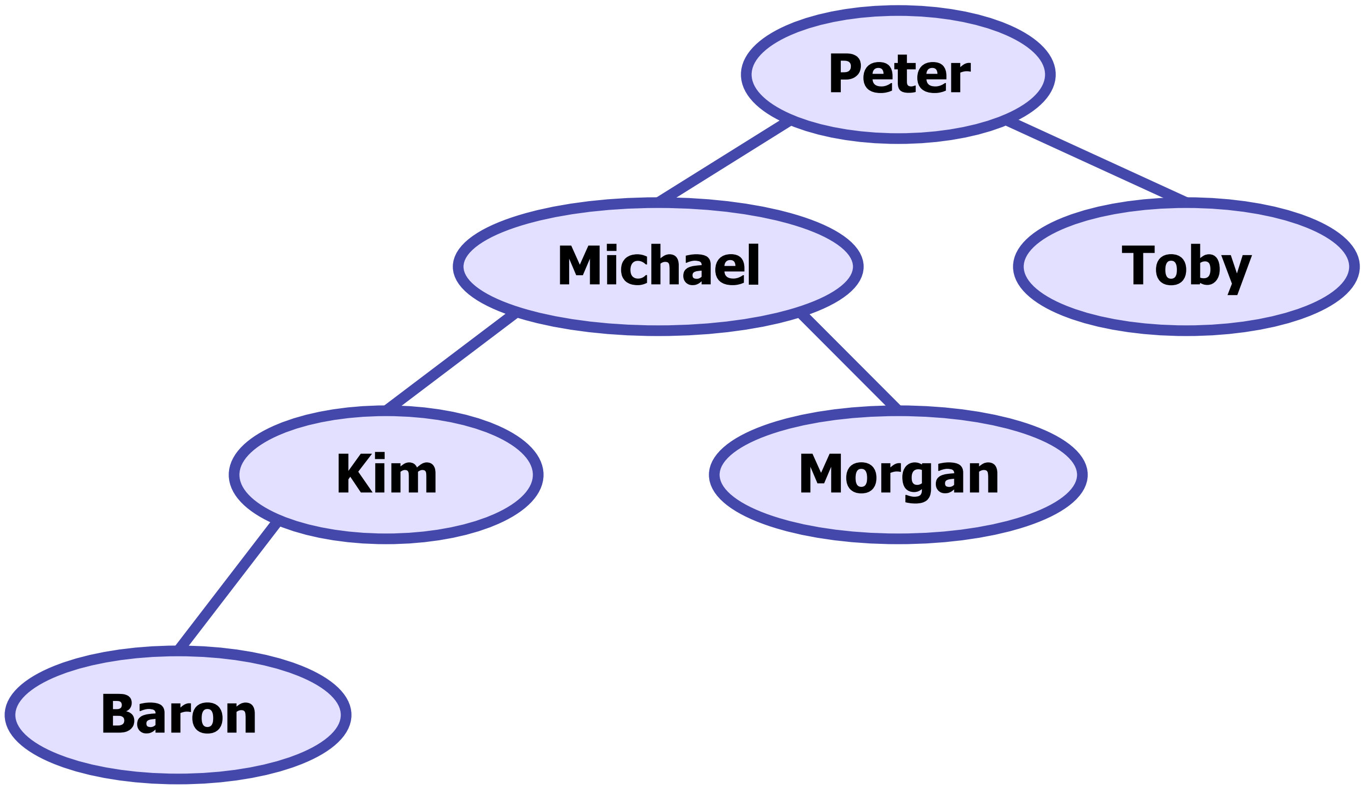
BST of Names

- Let's start with an empty tree and add the following names in the given order:
  - Peter, Michael, Kim, Morgan, Baron, and Toby



BST of Names

- Let's start with an empty tree and add the following names in the given order:
  - Peter, Michael, Kim, Morgan, Baron, and Toby



BST of Names

- What's the in-order traversal of this tree?
  - Baron, Kim, Michael, Morgan, Peter, Toby
  - notice that this is also sorted order!

- Why does this happen?

- in-order traversal
  - left subtree, current node, right subtree

- in-order traversal in a BST
  - values <= current value, current value, values > current value

left subtree **current node** **right subtree**

add

- Let's write an add method that will preserve the binary search tree property

```
public void add(int value) {
    ...
}
```

- Like most recursive tree methods, we'll need a private helper method to keep track of our current node

- We also know to use "x = change(x)" because we're modifying the tree

add

- Here's our updated add method

```
public void add(int value) {
    overallRoot = add(overallRoot, value);
}

private IntTreeNode add(IntTreeNode root, int value) {
    ...
}
```

- Now we have to write the rest of the code

add

- What's the simplest tree we could add a node to?
  - the empty tree
  - this is our base case

- For the empty tree, we'll just return a new node

- What about our recursive case(s)?
  - if the value to add is less than or equal to the value of the current node
    - ...add a new node to the left subtree
  - if the value to add is greater than the value of the current node
    - ...add a new node to the right subtree

add

- Here's our completed add method

```
public void add(int value) {
   overallRoot = add(overallRoot, value);
}

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

BST Wrap-up

- We've seen that BSTs can handle different kinds of objects
  - **`String`**s are sorted alphabetically
  - **`int`**s are sorted by non-decreasing value

- **`String`**s can be compared to other **`String`**s

- **`int`**s can be compared to other **`int`**s

- So we could even define our own class to put in a BST, so long as we can compare different instances of this class
  - you'll have to write a class like this on your final

PerceivedTemperature

- Let's write a simple class called **`PerceivedTemperature`** that keeps track of both the perceived temperature and actual temperature.

- Constructor code

```
public class PerceivedTemperature {
   private int pTemp;
   private int aTemp;

   public class PerceivedTemperature(int pt, int at) {
      pTemp = pt;
      aTemp = at;
   }
}
```

PerceivedTemperature

- Let's also give it a few more simple methods

```
public int getPerceivedTemperature() {
    return pTemp;
}

public int getActualTemperature() {
    return aTemp;
}

public String toString() {
    return pTemp + " (" + aTemp + ") " + degrees;
}
```

- It's still a simple class, but now it has some functionality

Some Client Code

- Let's also make some simple client code

```
import java.util.*;
public class PerceivedTemperatureMain {
   public static void main(String[] args) {
      List<PerceivedTemperature> temps =
            new ArrayList<PerceivedTemperature>();
      temps.add(new PerceivedTemperature(104, 103));
      temps.add(new PerceivedTemperature(104, 101));
      temps.add(new PerceivedTemperature(88, 90));
      System.out.println(temps);
   }
}
```

- Which produces the following output:

```
[104 (103) degrees, 104 (101) degrees, 88 (90) degrees]
```

Some Client Code

- But what if we wanted to sort our list before printing?
- There's a static method in the Java library called **`Collections.sort`** that takes a list as a parameter
- But if we add the following line to our client code:

```
Collections.sort(temps);
```

- We get a compiler error
  - we haven't told Java how to sort **`PerceivedTemperature`**s!

Comparable<T>

- If we want our class to be compatible with tools like **`Arrays.sort`** and **`Collections.sort`**, we need to tell Java how our class is ordered

- Java provides the **`Comparable<T>`** interface:

```
public interface Comparable<T> {
    public int compareTo(T other);
}
```

- **`compareTo`** is not a method in **`Object`** because some things aren't comparable
  - when is one **`Scanner`** greater than another **`Scanner`**?
  - when is one **`Map`** less than another **`Map`**?

compareTo

- What does **`compareTo`** return?

- Java's convention is:
  - if **`compareTo`** returns a negative number, it means "less"
  - if **`compareTo`** returns a zero, it means "equal"
  - if **`compareTo`** returns a positive number, it means "greater"

- Some examples:
  - if **`x.compareTo(y)`** returns -7, then x < y
  - if **`x.compareTo(y)`** returns 0, then x == y
  - if **`x.compareTo(y)`** returns 37, then x > y

compareTo

- For the **PerceivedTemperature** class, the single most important piece of data is the perceived temperature

- We'll use this to write an attempt of **compareTo**:

```
public int compareTo(PerceivedTemperature other) {
   return pTemp - other.pTemp;
}
```

- Notice that the above version works correctly
  - e.g. if pTemp < other.pTemp, it returns a negative number

- But what if the perceived temperatures are equal?
  - in this case, let's break ties by the actual temperatures

compareTo

- Final version of **`compareTo`**:

```
public int compareTo(PerceivedTemperature other) {
    if (pTemp == other.pTemp)
        return aTemp - other.aTemp;
    else
        return pTemp - other.pTemp;
}
```

- Now if we try to compile our client code...

- ...we still get a compiler error! But why? We wrote a correct **`compareTo`**!
  - we forgot to implement the **`Comparable`** interface

Implementing `Comparable`

- We need to have **`PerceivedTemperature`** implement the **`Comparable`** interface:

```
public class PerceivedTemperature implements
        Comparable<PerceivedTemperature> {
    ...
}
```

Notice we have to say what it is comparable to (namely, other `PerceivedTemperature` objects)

- Remember: by implementing the interface we're promising Java that we've written a **`compareTo`** method. Only after we make this promise will Java let us use **`Collections.sort`**

Updated Client Code

- Here's our updated our client code:

```
import java.util.*;
public class PerceivedTemperatureMain {
   public static void main(String[] args) {
      List<PerceivedTemperature> temps =
            new ArrayList<PerceivedTemperature>();
      temps.add(new PerceivedTemperature(104, 103));
      temps.add(new PerceivedTemperature(104, 101));
      temps.add(new PerceivedTemperature(88, 90));
      Collections.sort(temps);
      System.out.println(temps);
   }
}
```

- Which now produces the following output:

```
[88 (90) degrees, 104 (101) degrees, 104 (103) degrees]
```

`compareTo` and `double`s

- Suppose **`PerceivedTemperature`** had stored its temperatures as **`double`**s instead of as **`int`**s.

- How would this affect **`compareTo`**?
  - **`compareTo`** is supposed to return an **`int`**, but now our subtraction operations return **`double`**s

- An attempt at fixing **`compareTo`** by casting to **`int`**s

```
public int compareTo(PerceivedTemperature other) {
    if (pTemp == other.pTemp)
        return (int)(aTemp - other.aTemp);
    else
        return (int)(pTemp - other.pTemp);
}
```

This doesn't always work!

`compareTo` and `doubles`

- Simple casting will return the wrong answer if the difference between the two temperatures is both non-zero and less than one
  - any difference strictly between -1.0 and 1.0 is converted to 0

- We need to check if the difference between the temperatures is non-zero

- The easiest way to do this is with > and <
  - e.g. **`if (pTemp > other.pTemp)`**

compareTo and doubles

- One solution (assuming temperatures are doubles):

```
public int compareTo(PerceivedTemperature other) {
    if (pTemp == other.pTemp)
        return compareDoubles(aTemp, other.aTemp);
    else
        return compareDoubles(pTemp, other.pTemp);
}

private int compareDoubles(double d1, double d2) {
    if (d1 < d2)
        return -1;
    else if (d1 > d2)
        return 1;
    else
        return 0;
}
```