Data Structures

Trees III

CS284

Binary Search Trees

Find

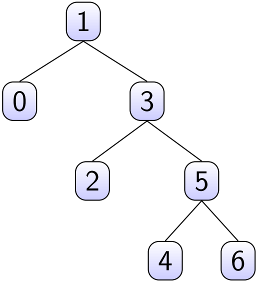
Add

Largest Item

Delete

Overview of a Binary Search Tree

- *Empty*
- $Node(i, lt, rt)$
  - lt and rt are binary search trees and
  - i is greater than all values in lt
  - i is less than all values in rt



Interface `SearchTree<E>`

```
public interface SearchTree<E extends Comparable<E>>  {

    // false if the item was already in the tree.
    boolean add(E item);

    boolean contains(E target);

    // If not found null is returned.
    E find(E target);

    // If not found null is returned.
    E delete(E target);

    // true if the object was in the tree, false otherwise
    boolean remove(E target);
}
```

BinarySearchTree Class

```java
public class BinarySearchTree<E extends Comparable<E>>
          extends BinaryTree<E>
         implements SearchTree<E> {
    // Data Fields

    /** Return value from the public add method. */
    protected boolean addReturn;
    /** Return value from the public delete method. */
    protected E deleteReturn;
    ...
}
```

Binary Search Trees

Find

Add

Largest Item

Delete

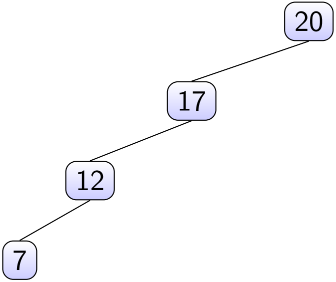
Recursive Algorithm for Searching a Binary Search Tree

Search a BST for a target `key`

```
let rec find key = function
  | Empty -> false
  | Node(i,lt,rt) when key=i -> true
  | Node(i,lt,rt)  ->
      if (key<i)
      then find key lt
      else find key rt
```

Performance

- Search in a BST is generally $\mathcal{O}(\log n)$
- If a tree is not very full, performance will be worse
- Searching a BST with only left subtrees, for example, is $\mathcal{O}(n)$



Implementing the `find` Method

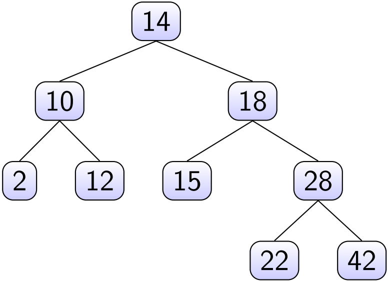
```java
public E find(E target)
    { return find(root, target); }

private E find(Node<E> localRoot, E target) {
    if (localRoot == null)
        { return null; }

    // Compare target with data field at the root.
    int compResult = target.compareTo(localRoot.data);
    if (compResult == 0) {
        return localRoot.data;
    } else if (compResult < 0) {
        return find(localRoot.left, target);
    } else {
        return find(localRoot.right, target);
    }
}
```

Binary Search Trees

- Find
- Add
- Largest Item
- Delete

Insert `key` into a Binary Search Tree `t` – Tree Expressions

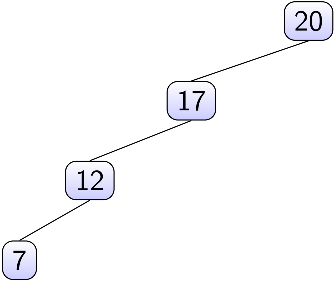
```
exception Duplicate_key

let rec add key = function
  | Empty -> Node(key,Empty,Empty)
  | Node(i,lt,rt) when key=i -> raise Duplicate_key
  | Node(i,lt,rt)  ->
      if (key<i)
      then Node(i,add key lt,rt)
      else Node(i,lt,add key rt)
```



- Insert 11
- Insert 17

Performance

- Insertion is $\mathcal{O}(n)$



- Could be better if tree were "balanced"

Insertion into a Binary Search Tree

Defined using two operations (the second is the helper):

- `public boolean add(E item)`
- `private Node<E> add(Node<E> localRoot, E item)`

```
public boolean add(E item) {
    root = add(root, item);
    return addReturn;
}
```

Insertion into a Binary Search Tree

```java
private Node<E> add(Node<E> localRoot, E item) {
    if (localRoot == null) {
        // item is not in the tree, insert it.
        addReturn = true;
        return new Node<E>(item);
    } else if (item.compareTo(localRoot.data) == 0) {
        // item is equal to localRoot.data
        addReturn = false;
        return localRoot;
    } else if (item.compareTo(localRoot.data) < 0) {
        // item is less than localRoot.data
        localRoot.left = add(localRoot.left, item);
        return localRoot;
    } else {
        // item is greater than localRoot.data
        localRoot.right = add(localRoot.right, item);
        return localRoot;
    }
}
```

Binary Search Trees

Find

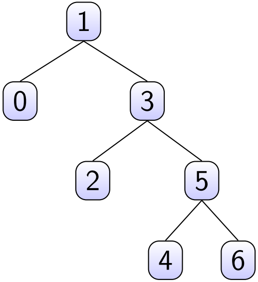
Add

Largest Item

Delete

Specifying `find_max`

```
exception Empty_tree

let rec find_max = function
| Empty -> raise Empty_tree
| Node(i,lt,Empty) -> i
| Node(i,lt,rt) -> find_max rt;;
```



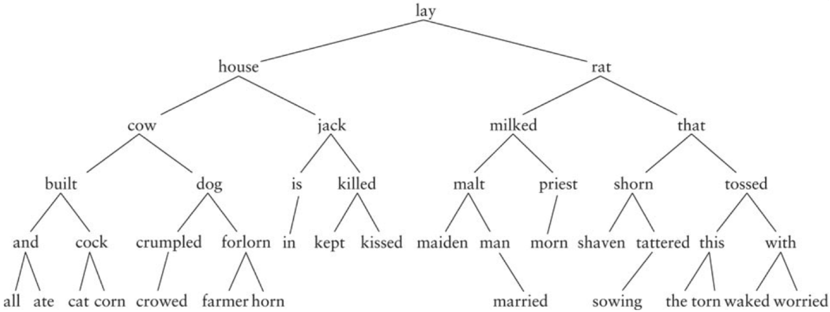
Implementing `findMax`

```
private E findMax(Node<E> current) {
  if (current==null) {
     throw new IllegalArgumentException();
  }
  if (current.right=null) {
      return current.data;
    } else {
      return findMax(current.right)
  }
}
```

Binary Search Trees

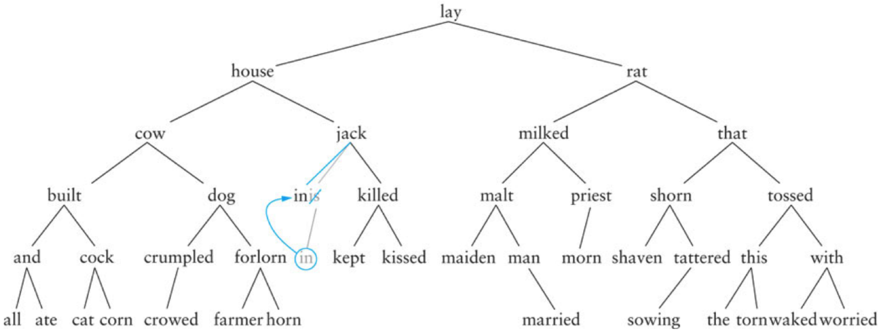
- Find
- Add
- Largest Item
- Delete

Removing from a Binary Search Tree



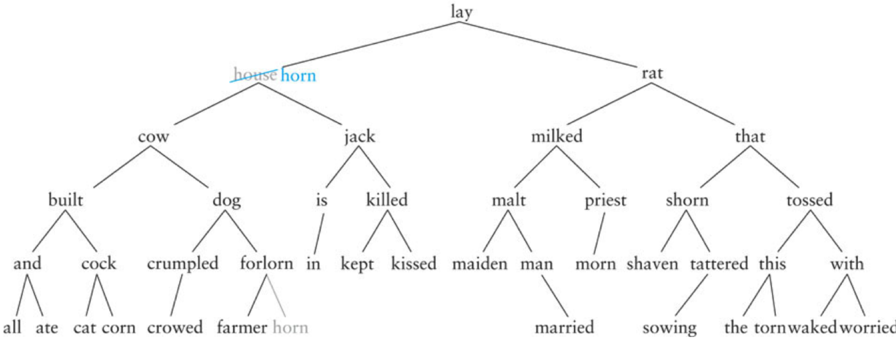
We want to remove “is”

Removing from a Binary Search Tree



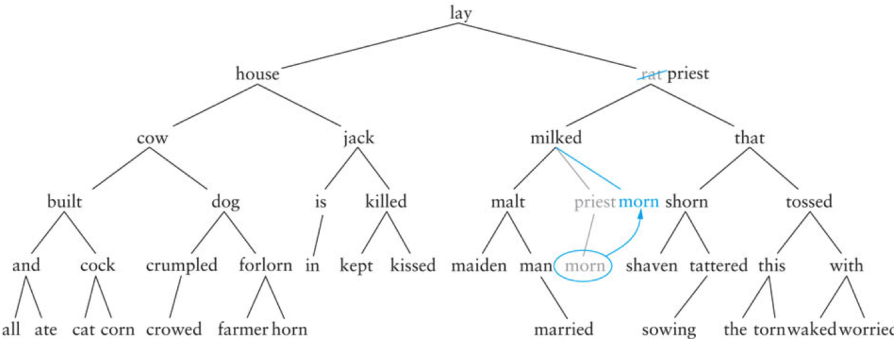
If the item to be removed (eg. "is") has only one child, replace it with this child

Removing from a Binary Search Tree (cont.)



If the item to be removed (eg. "house") has two children, replace it with the largest item in its left subtree – the inorder predecessor

Removing from a Binary Search Tree (cont.)



- The inorder predecessor is not always located at a leaf
- Consider removing "rat": its inorder predecessor is "priest" so we have to (recursively!) remove "priest"

Specifying `delete`

```
exception Not_found

let rec delete key = function
  | Empty -> raise Not_found
  | Node(i,lt,rt) when key=i -> join lt rt
  | Node(i,lt,rt) ->
       if key < i
         then Node(i, delete key lt, rt)
         else Node (i,lt,delete key rt)
and join l r =
   match l, r with
     | Empty, r -> r
     | l, Empty -> l
     | l, r ->
         let m = find_max l
         in Node(m,delete m l,r)
```

Specifying `delete`

```
exception Not_found

let rec delete key = function
  | Empty -> raise Not_found
  | Node(i,lt,rt) when key=i -> join lt rt
  | Node(i,lt,rt) ->
       if key < i
         then Node(i, delete key lt, rt)
         else Node (i,lt,delete key rt)
and join l r =
   match l, r with
     | Empty, r -> r
     | l, Empty -> l
     | l, r ->
         let m = find_max l
         in Node(m,delete m l,r)
```

Specifying `delete`

```
exception Not_found

let rec delete key = function
  | Empty -> raise Not_found
  | Node(i,lt,rt) when key=i -> join lt rt
  | Node(i,lt,rt) ->
      if key < i
        then Node(i, delete key lt, rt)
        else Node (i,lt,delete key rt)
and join l r =
   match l, r with
     | Empty, r -> r
     | l, Empty -> l
     | l, r ->
        let m = find_max l
        in Node(m,delete m l,r)
```

Specifying `delete`

```
exception Not_found

let rec delete key = function
  | Empty -> raise Not_found
  | Node(i,lt,rt) when key=i -> join lt rt
  | Node(i,lt,rt) ->
       if key < i
         then Node(i, delete key lt, rt)
         else Node (i,lt,delete key rt)
and join l r =
   match l, r with
     | Empty, r -> r
     | l, Empty -> l
     | l, r ->
         let m = find_max l
         in Node(m,delete m l,r)
```

Implementing the `delete` Method

Defined using two operations (the second is the helper):

- `public E delete(E target)`
- `private Node<E> delete(Node<E> localRoot, E item)`

```
public E delete(E target) {
  root = delete(root, target);
  return deleteReturn;
}
```

Implementing the `delete` Method

```java
private Node <E> delete(Node <E> localRoot , E item) {
    if (localRoot == null) { // item is not in the tree.
        deleteReturn = null;
        return localRoot;
    }

    // Search for item to delete.
    int compResult = item.compareTo(localRoot.data);
    if (compResult < 0) {
        // item is smaller than localRoot.data.
        localRoot.left = delete(localRoot.left , item);
        return localRoot;
    } else if (compResult > 0) {
        // item is larger than localRoot.data.
        localRoot.right = delete(localRoot.right , item);
        return localRoot;
    } else { // E == localRoot.data => join
    ...
}}
```

Implementing the `delete` Method (cont.)

```
      else { // E == localRoot.data
       deleteReturn = localRoot.data;
       if (localRoot.left == null) {
           return localRoot.right;
       } else if (localRoot.right == null) {
           return localRoot.left;
       } else {  // localRoot has 2 children
           if (localRoot.left.right == null) {
               localRoot.data = localRoot.left.data;
               localRoot.left = localRoot.left.left;
               return localRoot;
           } else {
               localRoot.data = findMax(localRoot.left);
               return localRoot;
           }
       }
   }
}
```

```
private E findMax(Node<E> parent) {
  // If the right child has no right child,
  // it is the inorder predecessor
    if (parent.right.right=null) {
       E returnValue = parent.right.data;
       parent.right = parent.right.left;
       return returnValue;
  } else {
    return findMax(parent.right)
  }
}
```