Structure of this week's classes

Huffman Tree

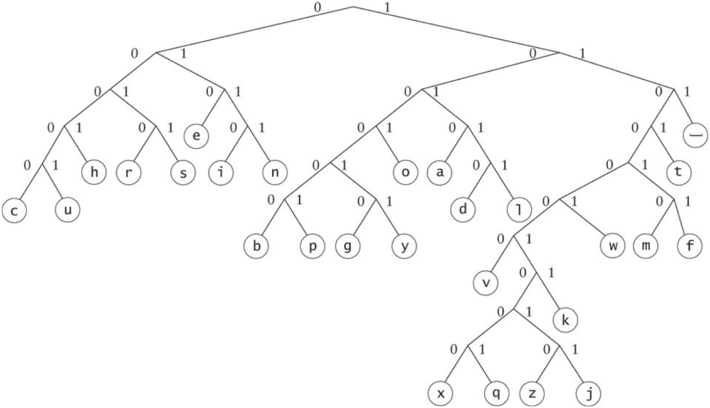
Binary Tree Pre-Order Traversal

Implementing a Binary Search Tree
- Add
- Delete

Huffman

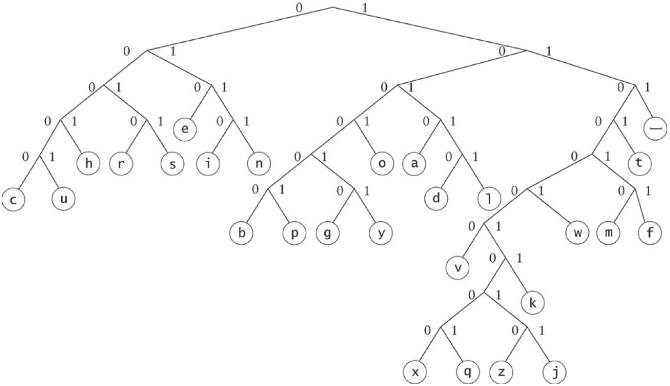
- A Huffman tree represents Huffman codes for characters that might appear in a text file
- As opposed to ASCII or Unicode, Huffman code uses different numbers of bits to encode letters; more common characters use fewer bits
- Many programs that compress files use Huffman codes

Huffman Tree



To form a code, traverse the tree from the root to the chosen character, appending 0 if you turn left, and 1 if you turn right.

Huffman Tree



Examples: d : 10110 e : 010

Huffman Coding

- Encode every character in the vocabulary as a 0-1 string
- Fixed-length encoding vs. Variable-length encoding
- *Uniquely-decodable* encoding:
  - Given the huffman code for each character, there is only 1 way to decode a sequence
  - e.g., $\{A{=}0, B{=}1, C{=}11, D = 10\}$ is not uniquely decodable, because 1011 can be decoded as both DC and BABB

Prefix Coding

- Prefix coding refers to any way of encoding where no character is a prefix of another character
- e.g., $\{A = 1, B = 01, C = 001, D = 0001, E = 0000\}$
- A prefix coding can guarantee unique decoding of a string. (Why?)
- Any encoding represented by a *full binary tree* (any node has 0 or 2 children) must be a prefix encoding. (Why?)

Using the Shortest Way of Encoding

- For example, if $\{A = 00, B = 01, C = 10, D = 11\}$, BAC is encoded as 010010
- If $\{A = 0000, B = 0101, C = 1010, D = 1111\}$ instead, BAC is encoded as 010100001010
- How to design a way of encoding that minimizes the encoded lengths for any char sequence such as BAC?
- Answer: Huffman coding (the more frequent char should be encoded with fewer bits)
- Recitation: greedy algorithm for Huffman coding and proof of its optimality

Huffman Tree

Binary Tree Pre-Order Traversal

Implementing a Binary Search Tree
- Add
- Delete

`toString()` Method

The `toString` method generates a string representing a preorder traversal in which each local root is indented a distance proportional to its depth

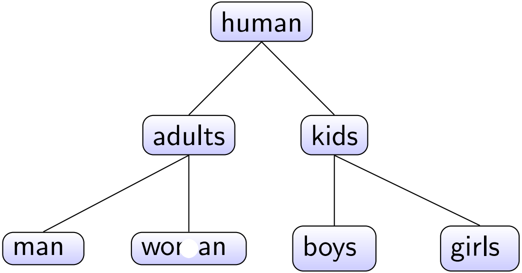
```
public String toString(Node<E> root) {
      StringBuilder sb = new StringBuilder();
      preOrderTraverse(root, 1, sb);
      return sb.toString();
}
```

preOrderTraverse Method

```java
private void preOrderTraverse(Node<E> node, int depth,
StringBuilder sb) {
      for (int i = 1; i < depth; i++) {
        sb.append("  ");
      }
      if (node == null) {
        sb.append("null\n");
      } else {
        sb.append(node.value);
        sb.append("\n");
        preOrderTraverse(node.l_child, depth + 1, sb);
        preOrderTraverse(node.r_child, depth + 1, sb);
      }
}
```

`preOrderTraverse` Method (cont.)

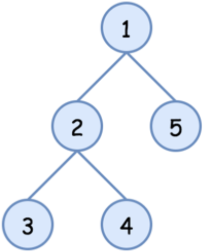
```
human
  adults
    men
      null
      null
    women
      null
      null
  kids
    boys
      null
      null
    girls
      null
      null
```

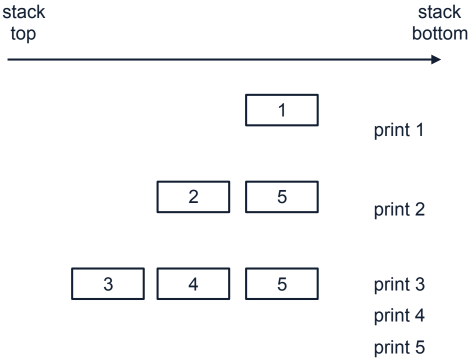


PreOrderTraverse

- Recursive traversal is trivial
- Can we use a stack to traverse the tree?
  - PreOrder is Root $\rightarrow$ L $\rightarrow$ R
  - Every time, pops the current node $\rightarrow$ pushes its right child $\rightarrow$ pushes its left child, until there are nodes in the stack
  - Let's look at one specific example

An Example of PreOrder Traversal





PreOrder Traversal: Iterative

```java
public void preOrderTraverse_iter(Node<E> root, StringBuilder sb) {
    root.set_depth(1);
    Stack<Node<E>> stack = new Stack<Node<E>>();
    stack.push(root);

    while(!stack.isEmpty()){
        Node<E> temp = stack.pop();
        int this_depth = temp.get_depth();

        for (int i = 1; i < this_depth; i++) {
            sb.append("  ");
        }
        if (temp == null)
            sb.append("null\n");
        else {
            sb.append(temp.value);
            sb.append("\n");
        }
```

PreOrder Traversal: Iterative

```
            if(temp.r_child != null){
                temp.r_child.set_depth(this_depth + 1);
                stack.push(temp.r_child);
            }
            if(temp.l_child != null){
                temp.l_child.set_depth(this_depth + 1);
                stack.push(temp.l_child);
            }
        }
    }
```

Huffman Tree

Binary Tree Pre-Order Traversal

Implementing a Binary Search Tree
- Add
- Delete

`BinarySearchTree` Class

```
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;
    ...
}
```

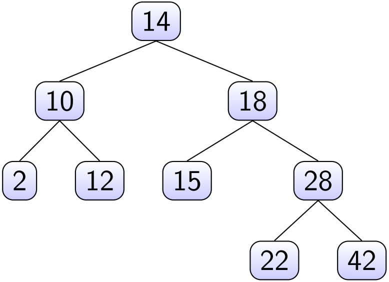
Huffman Tree

Binary Tree Pre-Order Traversal

Implementing a Binary Search Tree
- Add
- Delete

Insert `key` into a Binary Search Tree

- If the key is already in the tree, returns;
- Otherwise, attach the key as a child of the leaf node;



- Insert 11
- Insert 17

Insert `key` into a Binary Search Tree

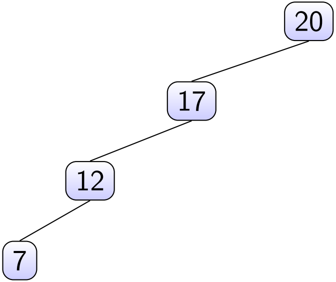
```java
public Node insertIntoBST(Node<E> root, E target) {

    if(root == null) {
        addReturn = true;
        return new Node(target, null, null);
    }

     if(target.compareTo(root.value) < 0) {
        addReturn = false;
        root.l_child = insertIntoBST(root.l_child, target);
    } else {
        root.r_child = insertIntoBST(root.r_child, target);
    }
    return root;
}
```

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;
}
```

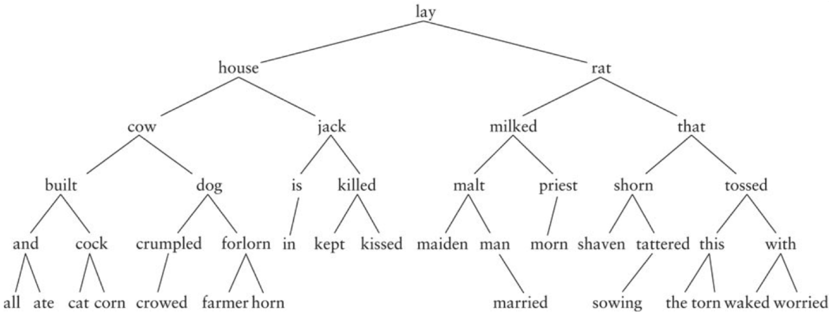
Huffman Tree

Binary Tree Pre-Order Traversal

Implementing a Binary Search Tree

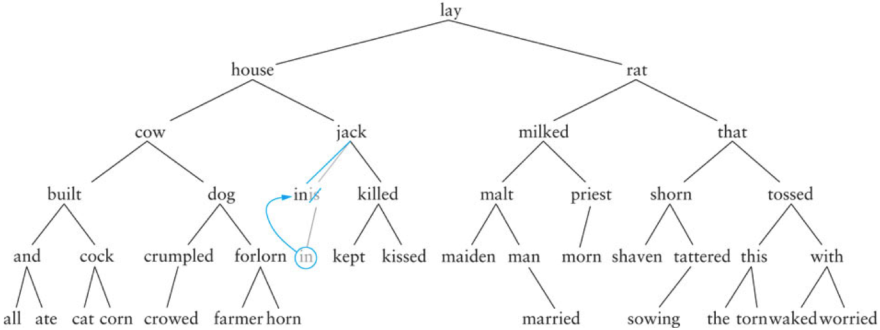
- Add
- Delete

Removing from a Binary Search Tree



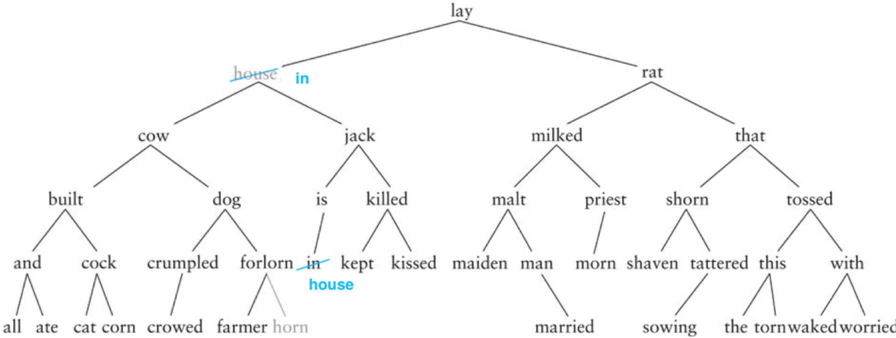
We want to remove “is”

Removing from a Binary Search Tree



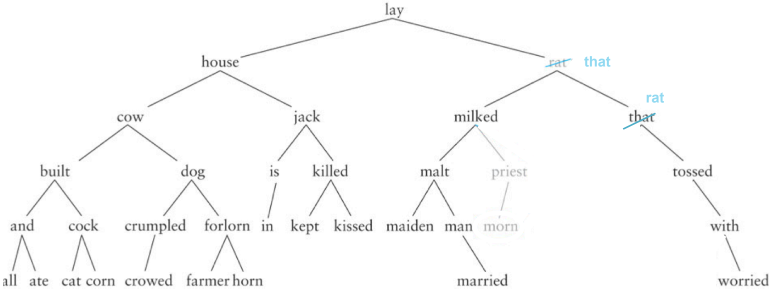
Replacing 'is': if the right child is null, replace it with the left child;

Removing from a Binary Search Tree (cont.)



If the right subtree of the item to remove (eg. “house”) is not null, swap it with the smallest item in the right sub-tree, then continue removing the item from the right sub-tree

Removing from a Binary Search Tree (cont.)



- The smallest item in the right sub-tree is not always located at a leaf
- Consider removing “rat”: we have to (recursively!) remove “rat”

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 = deleteNode(root, target);
    return root.value;
}
```

Implementing the `delete` Method (cont.)

```
    private Node<E> deleteNode(Node<E> root, E target) {
        if (root == null)
          return null;

        if (target.compareTo(root.value) < 0)
            root.l_child = deleteNode(root.l_child, target);

        // If the key to be deleted is greater than the
        // root's key, then it lies in right subtree
        else if (target.compareTo(root.value) > 0)
            root.r_child = deleteNode(root.r_child, target);

        // if key is same as root's key, then this is the node
        // to be deleted
        else {
```

Implementing the `delete` Method (cont.)

```
            if (root.r_child == null) {
                root = root.l_child;
            }
            else {
                // node with two children: Get the inorder
                // successor (smallest in the right subtree)
                Node<E> right = minValueNode(root.r_child);

                // Copy the inorder successor's data to this node
                root.set_value(right.value);

             // Delete the inorder successor
                root.r_child = deleteNode(root.r_child, right.value);
            }
        }

        return root;
    }
```