Assignments 5-7

Trees: Binary tree, Binary Search Tree, Heap, Priority Queue, Huffman Tree

Data Structures, Fall 2018

TA: Marija Stanojevic

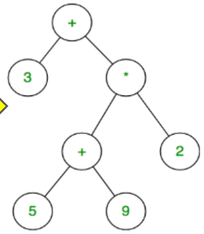
Introduction

Binary tree is set of nodes T, so that:

- T is empty or
- T consists of root node with two subtrees T_L, T_R which are binary trees

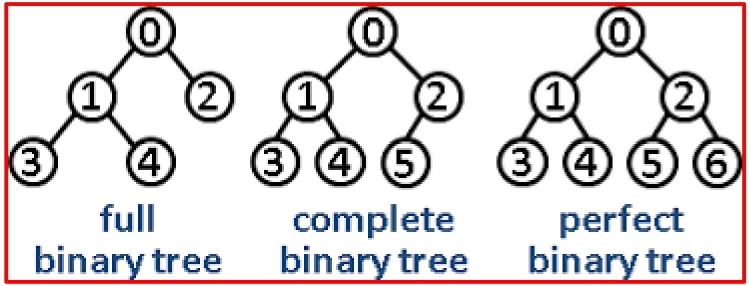
Types:

- Expression Tree
- Binary Search Tree
- Huffman Tree



Special properties:

- **Full tree**: each node has 0 or 2 children
- **Perfect tree**: full tree of height n with 2^n - 1 nodes
- **Complete tree**: perfect binary tree that is missing some nodes from rightmost part of the n-th level



Tree

```
public int depth () { depth(root); }
private int depth(Node<E> lR) {
  if (lR == null) return 0;
  return 1 + Math.max(depth(lR.left), depth(lR.right));
}
```

```
public int size () { size(root); }
private int size(Node<E> lR) {
  if (lR == null) return 0;
  return 1 + size(lR.left) + size(lR.right);
}
```

- Depth: number of layers
- Size: number of nodes
- **PreOrder**: root => left => right subtree
- **InOrder**: left subtree => root => right subtree
- **PostOrder**: left => right subtree => root
- Order is important when transforming tree into list or reading expression tree

```
void printPreorder(Node node) {
    if (node == null)
        return;
    System.out.print(node.key +" ");
    printPreorder(node.left);
    printPreorder(node.right);
}
```

```
void printInorder(Node node) {
    if (node == null)
        return;
    printInorder(node.left);
    System.out.print(node.key + " ");
    printInorder(node.right);
}
```

```
void printPostorder(Node node) {
    if (node == null)
        return;
    printPostorder(node.left);
    printPostorder(node.right);
    System.out.print(node.key+" ");
}
```

Binary Search Tree (BST)

- Binary tree such that all the values in left subtree are smaller and in right subtree are bigger
- InOrder of BST is a sorted list
- Perfect/Complete BST for fast search, time O(log n)
    - Finds element among 1,000,000 examples in max 20 steps

```
public E find (E target) {
    return find(root, target);
}
private E find(Node<E> lRoot, E t) {
    if (lR == null)
        return null;
    int cmp = t.compareTo(lR.data);
    if (cmp == 0) {
        return lR.data;
    }
    if (cmp < 0) {
        return find(lr.left, t);
    }
    return find(lr.right, t);
} //time: avg=O(log n), worst=O(n)
```

```
public void add (E item) {
    root = add(root, item);
}
private void add(Node<E> lR, E val) {
    if (lR == null)
        return new Node<E>(item);
    int cmp = val.compareTo(lR.data);
    if (cmp < 0) {
        lr.left = add(lr.left, val);
        return lr.left;
    }
    if (cmp > 0) {
        lr.right = add(lr.right, val);
        return lr.right;
    }
} //time: avg=O(log n), worst=O(n)
```

```
public void delete (E item) {
  root = delete(root, item);
}
private void delete(Node<E> lRoot, E
val) {
  if (lR == null) return null;
  if (t.compareTo(lR.data) < 0) {
    lr.left = delete(lr.left, val);
    return lr.left;
  }
  if (t.compareTo(lR.data) > 0) {
    lr.right= delete(lr.right, val);
    return lr.right;
  }
  if (lR.left==null) return lR.right;
  if (lR.right==null) return lR.left;
  if (lR.left.right == null) {
    lR.data = lR.left.data;
    lR.left = lR.left.left;
  } else {
    lR.data =
      delLargestChild(lR.left);
  }
  return lR;
} //time: avg=O(log n), worst=O(n)
```

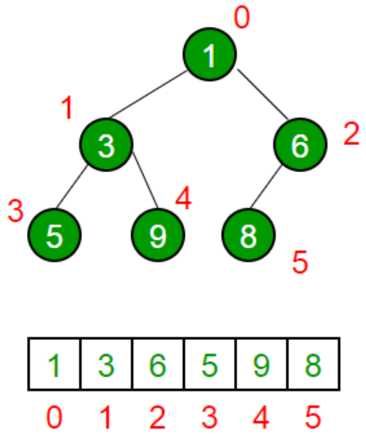
```java
public E findLargestChild () {
    if (lR == null)
        return null;
    return findLargestChild(root);
}
private E findLargestChild(Node<E> lR) {
    if (lR.right == null) {
            return lR.data;
    }
    return findLargestChild(lR.right);
} // for smallest child go left
```

```java
public boolean equals (BinarySearchTree bst) {
    if (bst != null) return equals(root, bst.root);
} // have to check if BST exists
private boolean equals (Node<E> lR1, Node <E> lR2) {
    if (lR1 == null && lr2 == null) return true;
    if (lR1 != null && lr2 != null)
            return lR1.data == lR2.data &&
             equals(lR1.left, lR2.left) &&
             equals(lR1.right, lR2.right);
    return false;
}
```

```java
public E delLargestChild () {
    if (root == null) return null;
    if (root.right == null)
        return root;
    return delLargestChild(root);
}
private E delLargestChild(Node<E> lR) {
    if (lR.right.right == null) {
        E val = lR.right.data;
            lR.right = lR.right.left;
            return lR;
    } else {
        return delLargestChild(lR.right);
    }
}
```

```java
public void copy (BinarySearchTree bst) {
    if (bst != null) root = copy(root, bst.root);
}
private void copy (Node<E> lR1, Node <E> lR2) {
    if (lR2 != null) {
        lR1 = new Node(lR2.data);
              lR1.left = copy(lR1.left, lR2.left);
              lR1.right = copy(lR1.right, lR2.right);
    }
    return null;
} // copies into empty tree
```

Heap



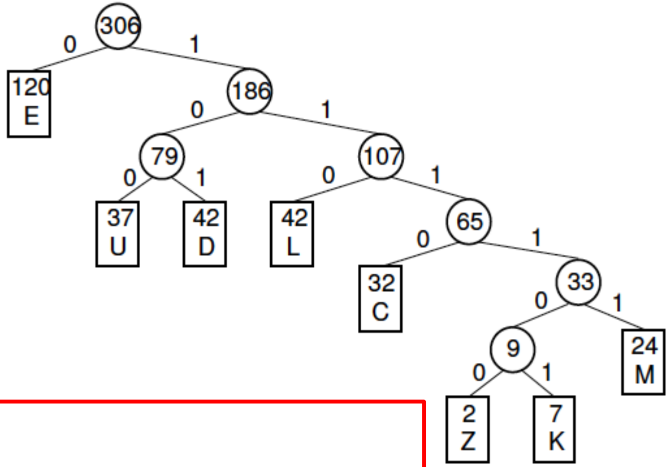
- **Heap is complete binary tree**
- **MIN heap:** children are bigger than parent
- **MAX heap:** children are smaller than parent
- Find / insert / remove time: O(log n)
- Used for Heapsort algorithm
- Usually stored as array to make operations easier and take less space

- **Priority queue is implemented with heap**
- **Priority queue** is queue which is sorted by priority of tasks (not by FIFO)
  - more important elements will be served first
- In java API generic class: **PriorityQueue**
- Main methods: offer, peek, poll and remove


Heaps
HackerRank

Huffman tree

- Used for binary encoding
- More frequent elements have shorter codes




```
public class HuffmanTree
implements Comparable<Node> {
public class Node {
    int freq;
    E symbol;
    Node left;
    Node right;
    public Node(int f, E s,
Node<E> l, Node<E> r) {
        freq = f;
        symbol = s;
              left = l;
              right = r;
    }
}
```

```
Node root = null;
public int compareTo (Node x) {
            return freq - x.freq;
}
public void createTree(PriorityQueue<Node<E>> p) {
  if (p != null)
    while (p.size() > 1) {
      Node val1 = p.pool();
            Node val2 = p.pool();
      root = new Node(val1.f + val2.f, null, val1, val2);
      p.offer(root);
   } //In java it is best to create tree from priorityQueue
}  // which elements are sorted ascendingly by freq
```

Huffman Encoding