Summary previous lecture

- **Trees** combine the advantages of an ordered array and a linked list
- **Binary Tree** (at most 2 childs per node)
- **BST**: left (resp. right) subtree has smaller (resp. greater) values than its root node

![General Tree](image1)

![Binary Tree](image2)

![Binary Search Tree (BST)](image3)
class Node {
    public int key;  // key-id value
    public Node left; // left child
    public Node right; // right child

    // constructor
    public Node(int key) {
        this.key = key;
        left = null;
        right = null;
    }
}

class Tree {
    private Node root; // Ref. root

    public Tree() { // constructor
        root = null;
    }
    // methods...Here
}
Binary Search Trees (BST)-- find method(s)

public Node find(int i) {
    Node current = root;  // start at root
    while (current!=null && current.key!=i) {
        if (i<current.key) {  // go left
            current = current.left;
        } else {  // go right
            current = current.right;
        }
    }
    return current;  // return Node or null
}

public Node recfind(Node current, int i) {
    if (current==null) return null;
    else if (i<current.key) // search left
        recfind(current.left,i);
    else if (i>current.key) // search right
        recfind(current.right,i);
    else  // find the target
        return current;
}

Iterative version

Recursive version

Complexity: # of levels O(logN)
• Insert a new node in the tree, there are 2 cases to consider
  • **Case1**: the BST tree is empty → we need to create a new node as root
  • **Case2**: the BST tree is not empty → we need to search for the position to insert the new node, similarly to the find method we start from the root and we keep comparing recursively.

• **Example**: insert 7 in this tree:

Since 7<10, 7 needs to be inserted to the left subtree
Since 7>5, 7 needs to be inserted to the right subtree
Since 5's right subtree is null, node for 7 is created
Binary Search Trees (BST)-- insert method

```java
public void recinsert(Node current, int i) {
    if (i<current.key) {
        // search left
        if (current.left==null) // node needs to be inserted
            current.left= new Node(i);
        else
            recinsert(current.left,i); // keep searching
    } else if (i>=current.key) {
        // search right
        if (current.right==null) // node needs to be inserted
            current.right= new Node(i);
        else
            recinsert(current.right,i); // keep searching
    }
}

public void insert(int i) {
    if (root==null) // node needs to be inserted (case 1)
        root= new Node(i);
    else // (case 2)
        recinsert(root,i); // initiate the recursion
}
Some algorithms need to visit all nodes in tree → Tree traversal

“Traversal” is process of visiting nodes

Recursion can be used very conveniently

- Visit current node
- Visit (recursively) all nodes in leftsubtree
- Visit (recursively) all nodes in rightsubtree

However order of visits matters! … 3 main different ways to perform the traversal:

- In-order (most commonly used for BST)
- Pre-order
- Post-order
1. Visit the left subtree
2. Visit the node (ex: display it)
3. Visit the right subtree

```
public void inOrder(Node current) {
    if (current != null) {
        inOrder(current.left); //1
        System.out.print(current.key + " "); //2
        inOrder(current.right); //3
    }
}
```
in-Order Traversal

Result is (in-Order) : 20 30 40 50 60
pre-Order Traversal

1- Visit the node (ex: display it)
3- Visit the left subtree
3- Visit the right subtree

```java
public void preOrder(Node current) {
    if (current!=null){
        System.out.print(current.key+" "); //1
        preOrder(current.left);          //2
        preOrder(current.right);         //3
    }
}
```

Result is (In-Order) : 1 5 7 10 14 16
Result is (Pre-Order) : 10 5 1 7 14 16
post-Order Traversal

1- Visit the left subtree
2- Visit the right subtree
3- Visit the node (ex: display it)

```java
public void postOrder(Node current) {
    if (current!=null){
        postOrder(current.left);  //1
        postOrder(current.right);  //2
        System.out.print(current.key+" ");  //3
    }
}
```

Result is (In-Order) : 1 5 7 10 14 16
Result is (Pre-Order) : 10 5 1 7 14 16
Result is (Post-Order):  1 7 5 16 14 10
Traversal: application

- A binary tree (not a BST) can be used to represent an arithmetic expression that involves the binary operators +, -, /, *.
- The root node holds an operator and each other nodes hold either a variable or another operator.
- Each subtree is a valid algebraic expression.
- Example: Binary Tree to represent: A*(B+C) … infix notation

- in-Order?
  - A*B+C (no parentheses)
- pre-Order?
  - *A+BC (prefix notation)
- post-Order?
  - ABC+* (postfix notation- RPN)
Traversal: application

- We can construct the tree using the postfix expression as input: \( ABC+* \)
- push(A), push(B), push(C)
- When we encounter the operand '+':
  \( x=\text{pop}, \ y=\text{pop} \) and create a tree with
  the operator + in its root
  \( (x \text{ right child, } y \text{ left child, here } x=C, y=B) \)
- push(tree);
- When we encounter another operand (here *):
  \( x=\text{pop}, \ y=\text{pop} \) and create a new tree with *
  in its root (here \( x \) is the tree above, \( y \) is A)
  and push(tree)

After a final pop, the resulting tree is a complete representation of the arithmetic expression
BST: finding maximum and minimum value

- Easy procedure- For the minimum, go to the left child of the root, then go to the left child of that child, and so on
- For the maximum, do the same to the right

```java
public Node minimum()
{
    Node current,last;
    current=root;
    while(current!=null){ //until the bottom
        last=current;
        current=current.left;
    }
    return last;
}
```
Motivations: How to show tree structures?
- How does the tree look like after insertion/removal of nodes?
- Easy approach → rotate the tree by 90 degrees
  - Most left is root
  - Right subtree of root is up
  - Left subtree of root is down

```java
public void recShow(Node current, String indStr) {
    if (current != null) {
        recShow(current.right, indStr + " ");
        System.out.println(indStr + current.key);
        recShow(current.left, indStr + " ");
    }
}
```

```java
public void show() {
    System.out.println("The tree looks like: ");
    recShow(root, " ");
}
```