Data Structure
Trees
Binary Search Tree

Abdul Mateen
Assistant Professor
Department of Computer Science, PU

Binary Tree Time Complexity

Statistically, what is the most frequent function?

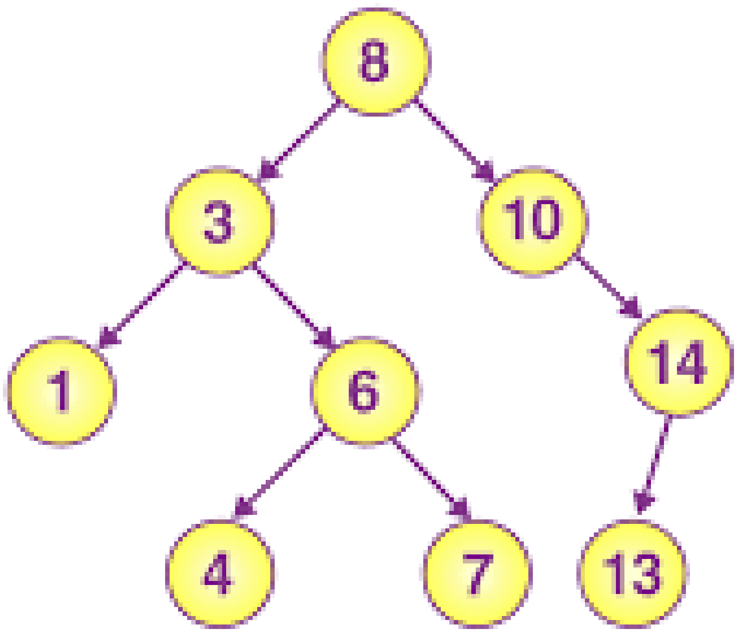
Time complexity of search function in worst case

Challenge: Can we reduce the time complexity of searching?

Binary Search Tree (BST)

A BST is a binary tree where:

- For any node, all nodes in the left subtree have smaller values.
- All nodes in the right subtree have larger values.



BST Operations

- Fundamental Operations in BST
- **Insertion**: Add a new element to the tree
- **Search**: Find whether an element exists in the tree
- **Deletion**: Remove an element from the tree while maintaining BST properties

Insertion in BST

- Start at the root
- Compare the new value with the current node:
  - If smaller, go to the left subtree
  - If larger, go to the right subtree. Insert at the appropriate null position

Insert Function

```
void insert(int value){
  root = insert(root, value);
}

TreeNode insert(TreeNode* root, int val){
  if (!root)  return new TreeNode(val);

  if (root->value > value)
     root->left=insert(root->left, val);

  else
     root->right=insert(root->right, val);
  return root;
}
```

Search in BST

- Start at the root
- Compare the target value with the current node:
  - If equal, return the node
  - If smaller, search the left subtree
  - If larger, search the right subtree
- Repeat until the value is found or a null node is reached

Search Function

```
void search(int value){
  return search (root, value);
}

TreeNode* search(TreeNode* root, int val){
  if (!root)  return NULL;

  if (root->value > val)
     return search(root->left, val);

  return  search(root->right,val);
}
```

Deletion in BST

- Locate the node to delete
- Handle one of three cases:
  - **No children**: Simply remove the node.
  - **One child**: Replace the node with its child
  - **Two children**: Replace the node with its **in-order successor** (smallest node in the right subtree)
- Ensure the BST properties are maintained

Delete Function

```
void delNode(int value){
  root = delNode (root, value);
}
TNode* delNode (TNode* root,int val){
  if (!root)   return NULL;
  if (root->value > value)
      root->left= delete (root->left, val);
  else if (root->value<value)
      root->right=delete(root->right,val);
  else if (!root->left && !root->right){
      delete root;       return NULL;
  }
  …
```

Delete Function (cont...)

```
else if (!root->left){
    TreeNode* temp = root->right;
    delete root;
    return temp;
}
else if (!root->right){
    TreeNode* temp = root->left;
    delete root;
    return temp;
}
```

Delete Function (cont...)

```
    else{
        TreeNode* temp = findSmallest(root->right);
        root->value = temp->value;
        root->right=delete(root->right, temp->val);
    }
    return root;
}
```

Complexity of BST Functions

- Insert Complexity
  - Best Case : O(LogN)
  - Worst Case : O(N)
- Search Complexity
- Deletion Complexity

Factor of Complexity of BST

- Height varies from Log N or N
- Maintain height to Log N
- Height Balanced Tree
  - AVL Tree
  - Red Black Tree
  - Splay Trees