Binary Search

- Search algorithm that finds the position of a target value in sorted values.
  - We need to sort the data first

- Binary search for 3

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>7</td>
<td>28</td>
<td>32</td>
<td>37</td>
<td>40</td>
</tr>
</tbody>
</table>
Binary Search

- Two approach
  - Using iteration
  - Using recursive function

- Check “BinarySearch.cpp”

### Time complexity

<table>
<thead>
<tr>
<th>Class</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worse case performance</td>
<td>Log N</td>
</tr>
<tr>
<td>Best case performance</td>
<td>1</td>
</tr>
<tr>
<td>Average case performance</td>
<td>Log N</td>
</tr>
</tbody>
</table>
Binary Search

- Limitations?
  - Need to sort entire data whenever adding new values

Binary Search Trees

- Construct binary tree with the given collection data
- Allow very fast lookup, addition, removal of items.
- Reflect structural relationship in the data
- Efficient insertion and searching
Binary Search Trees

- Binary search tree or Ordered binary tree

A set of nodes, where each node contains a "left" pointer, a "right" pointer, and a data element.
Binary Search Trees

- Binary search tree (BST) or Ordered binary tree
  - A type of binary tree where the nodes are arranged in order
  - all elements in its left sub-tree are less-or-equal to the node (<=), and all the elements in its right sub-tree are greater than the node (>)

Binary Search Trees

- Insertion
  - Add the first one as a root
  - If a new value is less than the node’s value
    - If a current node has no left child, insert the new value
    - Otherwise, handle the left child with the same algorithm
  - If a new value is greater than the node’s value
    - If a current node has no right child, insert the new value
    - Otherwise, handle the right child with the same algorithm
Binary Search Trees

Example

```
  1  10  8  4  6  3  2  5
```

Binary Search Trees

Example

```
  1  10  8  4  6  3  2  5
```

```
  1  10
     |
      8
    /  |
   4   6
  /|
 3 2 5
```
Binary Search Trees

- **Deletion**
  - **Case 1**: remove a node that has no child
    - Delete it and set the link of the parent to NULL

  ![Diagram showing case 1](http://www.algolist.net/Data_structures/Binary_search_tree/Removal)

- **Case 2**: remove a node that has one child
  - Node is cut from the tree, and link single child (with its subtree) directly to the parent of the removed node.

  ![Diagram showing case 2](http://www.algolist.net/Data_structures/Binary_search_tree/Removal)
Binary Search Trees

- **Deletion**
  - Case 3: remove a node that has two child
    - The most complex case

http://www.algolist.net/Data_structures/Binary_search_tree/Removal

Binary Search Trees

- Same set of values can be represented as different binary search tree
  - Choose minimum element from the right subtree
  - Replace 5 by 19
  - Hang 5 as a left child
Binary Search Trees

- Find a minimum value in the right subtree
- Replace value of the node to be removed with the found minimum

Right subtree contains a duplicate
- Apply remove to the right subtree to remove a duplicate
Binary Search Trees

- Time Complexity

<table>
<thead>
<tr>
<th></th>
<th>Average</th>
<th>Worse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search</td>
<td>Log N</td>
<td>N</td>
</tr>
<tr>
<td>Insertion</td>
<td>Log N</td>
<td>N</td>
</tr>
<tr>
<td>Deletion</td>
<td>Log N</td>
<td>N</td>
</tr>
</tbody>
</table>

Binary Tree

- Tree data structure, where each node has at most two children (left child and right child)
- Not necessary to be ordered.
Binary Tree

Tree Terminology

- Depth of a node: number of edges from the root to the node
- Height of a node: number of edges from the node to the deepest leaf
- Height of a tree: a height of the root
- Full binary tree: A full binary tree is a tree in which every node other than the leaves has two children
- Complete binary tree: a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

Full Binary Tree

Complete Binary Tree
Binary Tree Traversals

- Process that visits all the nodes in the tree
- Because a tree is nonlinear data structure, there is no unique traversal

Binary Tree

- Two types of traversal (Search)
  - Depth-first traversal
  - Breadth-first traversal
Depth-first traversal

- Pre-Order traversal
  - Visit the parent first and then left and right children
- In-Order traversal
  - Visit the left child, then the parent and the right child
- Post-Order traversal
  - Visit left child, then the right child and then the parent

Breadth-first traversal

- One type Breadth-first traversal
  - Level-Order traversal
    - Visit nodes by levels from top to bottom and from left to right
Traversals on Binary Search Tree

- **Tree traversal**
  - Depth-first:
    - Pre-order: F, B, A, D, C, E, G, I, H
    - In-order: A, B, C, D, E, F, G, H, I
    - Post-order: A, C, E, D, B, H, I, G, F
    - Breadth-first: F, B, G, A, D, I, C, E, H

**Examples**
- **Pre-Order**
  - 8, 5, 9, 7, 1, 12, 2, 4, 11, 3
- **In-Order**
  - 9, 5, 1, 7, 2, 12, 8, 4, 3, 11
- **Post-Order**
  - 9, 1, 2, 12, 7, 5, 3, 11, 4, 8
- **Level-Order**
  - 8, 5, 4, 9, 7, 11, 1, 12, 3, 2
Implementation - Recursive

- **DFS: Pre-Order**

```c
void preorder(BinaryTreeNode* t) {
    if (t != NULL){
        display(t->data)
        preorder(t->left)
        preorder(t->right)
    }
}
```

Implementation - Recursive

- **DFS: In-Order**

```c
void inorder(BinaryTreeNode* t) {
    if (t != NULL){
        inorder(t->left)
        display(t->data)
        inorder(t->right)
    }
}
```
Implementation - Recursive

- **DFS: Post-Order**

```c
void postorder(BinaryTreeNode* t) {
    if (t != NULL) {
        postorder(t->left);
        postorder(t->right);
        display(t->data);
    }
}
```

- **DFS: Pre-Order**

```
Push the root node on a stack
while (stack is not empty) {
    pop from the stack
    push the right child node to the stack
    push the left child node to the stack
}
```
Implementation

- **DFS: In-Order**

  1) Create an empty stack S.
  2) Initialize current node as root
  3) Push the current node to S and set current = current->left until current is NULL
  4) If current is NULL and stack is not empty then
      a) Pop the top item from stack.
      b) Print the popped item, set current = popped_item->right
      c) Go to step 3.
  5) If current is NULL and stack is empty then we are done.

- **DFS: Post-Order**

  While root is not NULL
  
  - Push root’s right child and then root to stack
    - set root as root’s left child
  
  End While

  Pop an item from stack and set it as root
  
  - If the popped item has a right child and the right child is at top of stack, then remove the right child from stack, push the root back and set root as root’s right child.
  - Else print root’s data and set root as NULL

  Repeat above while stack is not empty.
Implementation

- Breadth-First Traversal

```plaintext
Put the root node on a queue
while (queue is not empty){
    dequeue the next node
    enqueue the left child node
    enqueue the right child node
}
```