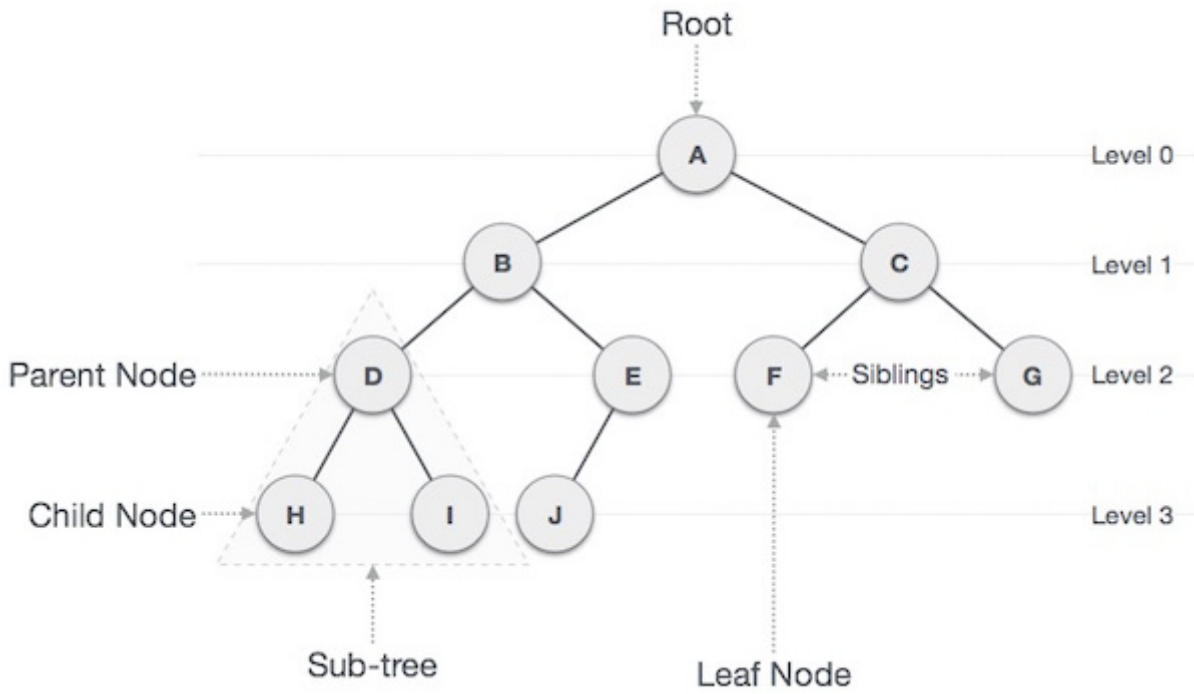
Data Structure Manual (2)

Tree

A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges.



- **Path** − Path refers to the sequence of nodes along the edges of a tree.
- **Root** − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** − Any node except the root node has one edge upward to a node called parent.
- **Child** − The node below a given node connected by its edge downward is called its child node.
- **Leaf** − The node which does not have any child node is called the leaf node.
- **Subtree** − Subtree represents the descendants of a node.

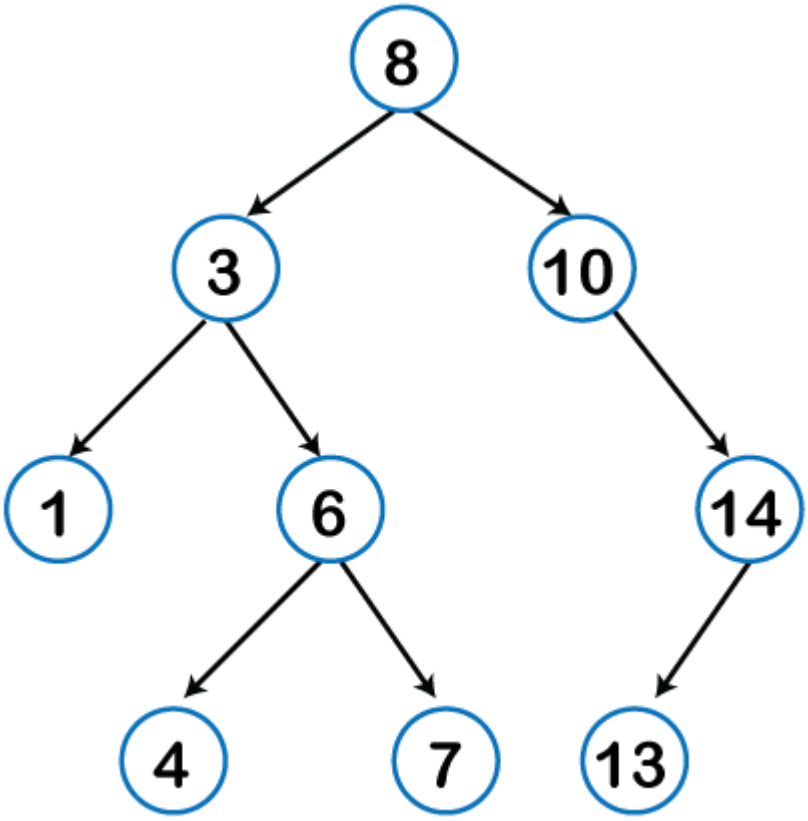
- **Visiting** − Visiting refers to checking the value of a node when control is on the node.
- **Traversing** − Traversing means passing through nodes in a specific order.
- **Levels** − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **keys** − Key represents a value of a node based on which a search operation is to be carried out for a node.

Applications of trees

The following are the applications of trees:

- **Storing naturally hierarchical data:** Trees are used to store the data in the hierarchical structure. For example, the file system. The file system stored on the disc drive, the file and folder are in the form of the naturally hierarchical data and stored in the form of trees.
- **Organize data:** It is used to organize data for efficient insertion, deletion and searching. For example, a binary tree has a `O(log(n))` time for searching an element.
- **Trie:** It is a special kind of tree that is used to store the dictionary. It is a fast and efficient way for dynamic spell checking.
- **Heap:** It is also a tree data structure implemented using arrays. It is used to implement priority queues.
- **B-Tree and B+Tree:** B-Tree and B+Tree are the tree data structures used to implement indexing in databases.
- **Routing table:** The tree data structure is also used to store the data in routing tables in the routers.

Binary Tree



Node structure (binary tree)

```
struct node
{
  int key;
  struct node *left;
  struct node *right;
}
```

A binary tree in which each node has exactly zero or two children is called **a full binary tree**.

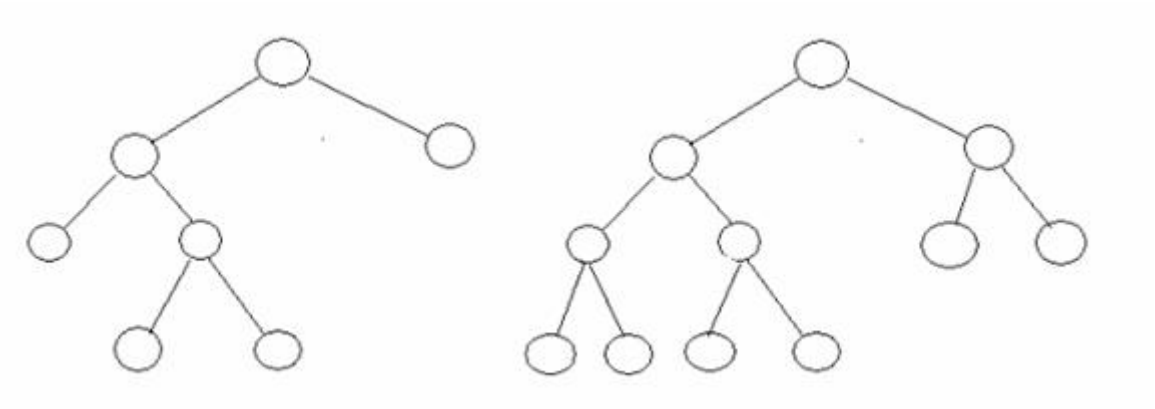
In a full tree, there are no nodes with exactly one child.

A complete binary tree is a tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right.

A complete binary tree of the height h has between 2h and 2(h+1)-1 nodes.

Here are some examples:

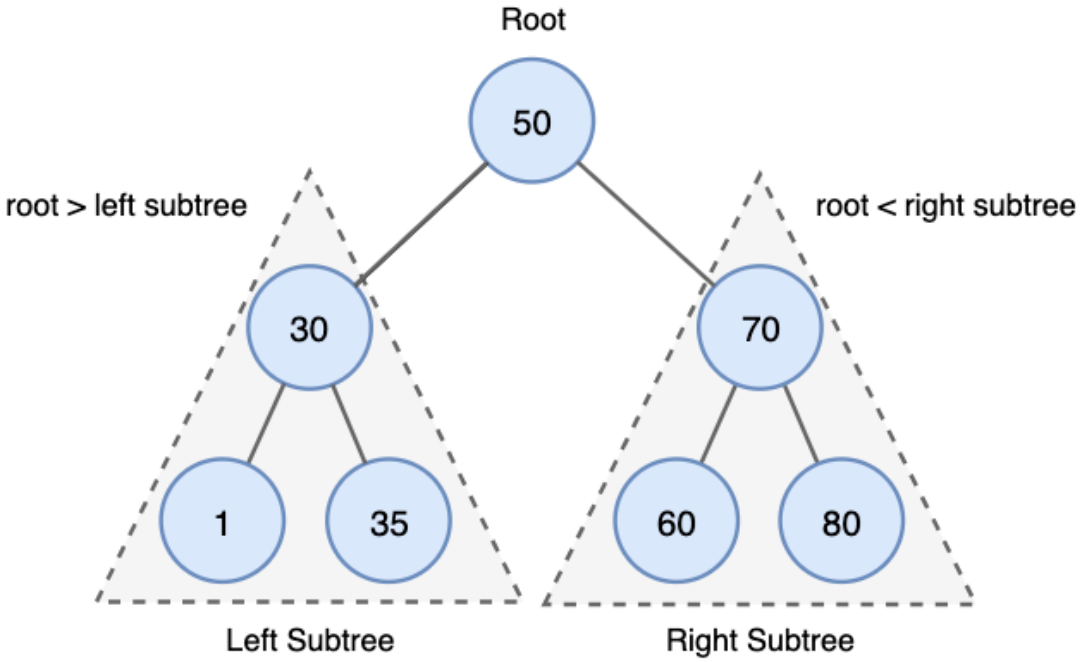
[Full Tree] **[Complete Tree]**



Binary Search Tree (BST)

Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.

- It is called a binary tree because each tree node has a maximum of two children.
- It is called a search tree because it can be used to search for the presence of a number in `O(log(n))` time.



Properties of a Binary Search tree

- Each node has a maximum of up to two children
- The value of all the nodes in the left sub-tree is less than the value of the root
- Value of all the nodes in the right subtree is greater than or equal to the value of the root
- This rule is recursively valid for all the left and right subtrees of the root

Tree Traversal

Traversing a tree means visiting every node in the tree.

You might, for instance, want to add all the values in the tree or find the largest one.

For all these operations, you will need to visit each node of the tree.

- Inorder Traversal
    1. First, visit all the nodes in the left subtree
    2. Then the root node
    3. Visit all the nodes in the right subtree
- Preorder Traversal

1. Visit root node
2. Visit all the nodes in the left subtree
3. Visit all the nodes in the right subtree

- Postorder Traversal
  1. Visit all the nodes in the left subtree
  2. Visit all the nodes in the right subtree
  3. Visit the root node

Example of Traversal Code in C

```
void inorder_traversal(struct Node* root) {
  if (root == NULL) return;
  inorder_traversal(root->left);
  printf("%d ->", root->key);
  inorder_traversal(root->right);
}

void preorder_traversal(struct Node* root) {
  if (root == NULL) return;
  printf("%d ->", root->key);
  preorder_traversal(root->left);
  preorder_traversal(root->right);
}

void postorder_traversal(struct Node* root) {
  if (root == NULL) return;
  postorder_traversal(root->left);
  postorder_traversal(root->right);
  printf("%d ->", root->key);
}
```

BST Time Complexities

Operation	Best Case Complexity	Average Case Complexity	Worst Case Complexity
Search	`O(log(n))`	`O(log(n))`	`O(n)`
Insertion	`O(log(n))`	`O(log(n))`	`O(n)`
Deletion	`O(log(n))`	`O(log(n))`	`O(n)`

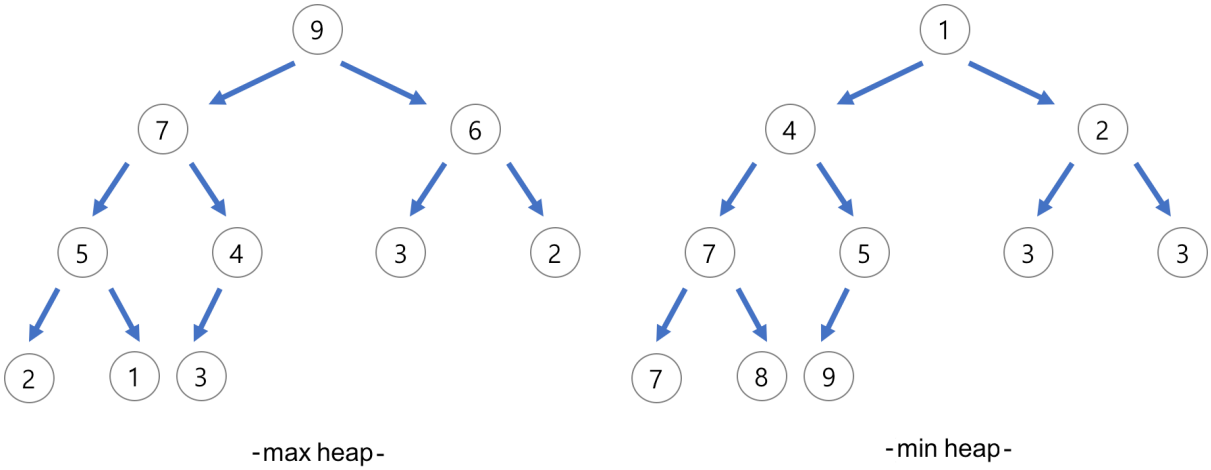
Heap

A heap is a tree-based data structure in which all the nodes of the tree are in a specific order.

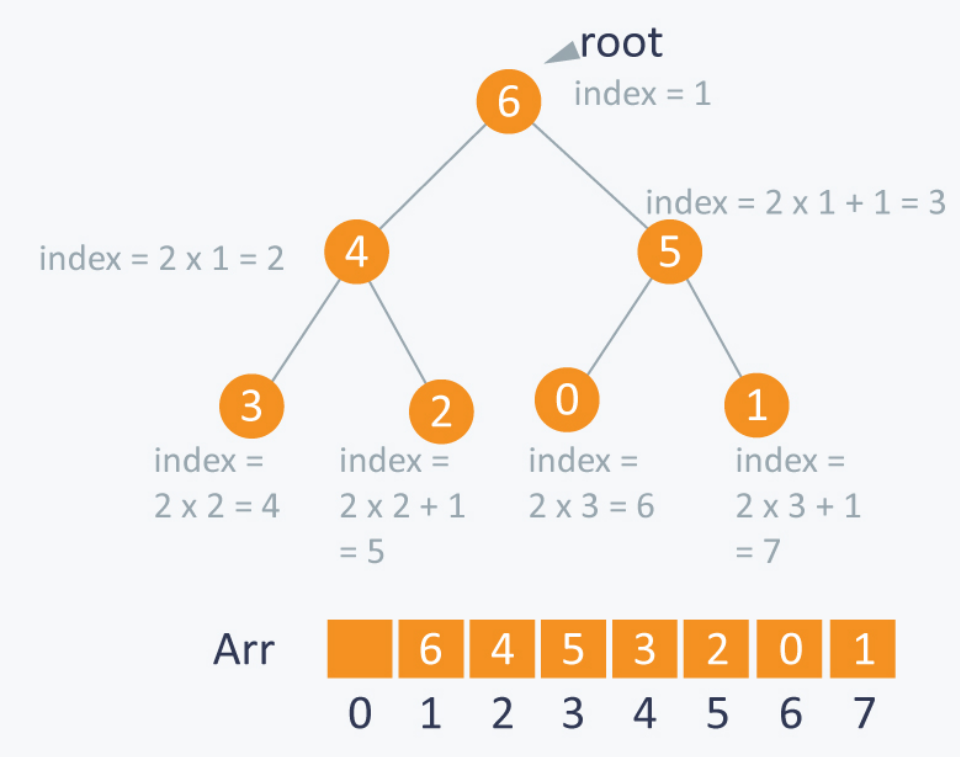
Heap is a special case of balanced binary tree data structure where the root-node key is compared with its children and arranged accordingly. If **α** has child node **β** then **key(α) ≥ key(β)**

As the value of parent is greater than that of child, this property generates **Max Heap**.

Based on this criteria, a heap can be of two types → **Max Heap** and **Min heap**



Heaps are usually built on arrays.



Strengths:

- **Quickly access the smallest item.** Binary heaps allow you to grab the smallest item (the root) in O(1)$O(1)$ time, while keeping other operations relatively cheap (O(lg(n))$O(lg(n))$ time).
- **Space efficient.** Binary heaps are usually implemented with *arrays*, saving the overhead cost of storing pointers to child nodes.

Weaknesses

- **Limited interface.** Binary heaps only provide easy access to the smallest item. Finding other items in the heap takes $O(n)$ time, since we have to iterate through all the nodes.

Uses

- Priority queues are often implemented using heaps. Items are *enqueued* by adding them to the heap, and the highest-priority item can quickly be grabbed from the top.
- One way to efficiently sort an array of items is to make a heap out of them and then remove items one at a time—in sorted order.

Heapify

Heapify is the process of creating a heap data structure from a binary tree represented using an array. It is used to create Min-Heap or Max-heap. Start from the first index of the non-leaf node whose index is given by n/2 – 1. Heapify uses recursion

Heapify in C

```
void heapify(int arr[], int p_idx, int h_size)
{
  int largest = p_idx;
  int l_idx = p_idx * 2 + 1;
  int r_idx = p_idx * 2 + 2;

  /* find a largest one in p, l, r */
  if (l_idx < h_size && arr[l_idx] > arr[largest])
    largest = l_idx;
  if (r_idx < h_size && arr[r_idx] > arr[largest])
    largest = r_idx;
  if (largest != p_idx) { /* base case */
    swap(arr, largest, p_idx); /* swap with larger child */
    heapify(arr, largest, h_size); /* heapify agrain from the child */
  }
}

void build_heap(int arr[], int h_size)
{
  for (int i = (h_size / 2) - 1; i >= 0; i--)
    heapify(arr, i, h_size);
}
```

We can make build_heap, heap_push, heap_pop, heap_sort with heapify.

Operation	Binary Heap
heapify	`O(log(n))`
build_heap	`O(nlog(n))`
heap_push	`O(log(n))`
heap_pop	`O(log(n))`
heap_sort	`O(nlog(n))`

In heap operations, `O(log(n))` is required because heapfiy is required after operations are finished.