CSCI212
Computer Science

Binary Trees/Heaps/Binary Search Trees

Tree Terminology

- A ***tree*** is a non-linear abstract data type that stores elements hierarchically.

- With the exception of the top element (**root**), each element (**node**) in a tree has a ***parent*** element and zero or more ***children*** elements (**siblings**).

- It is usually depicted by placing elements inside circles, ovals

  or rectangles, and by drawing connections (**edges**) between parents and children with straight lines.

- A node that has 0 children is called a **leaf.**

Tree Terminology

- A ***path*** *(*a sequence of nodes such that any 2

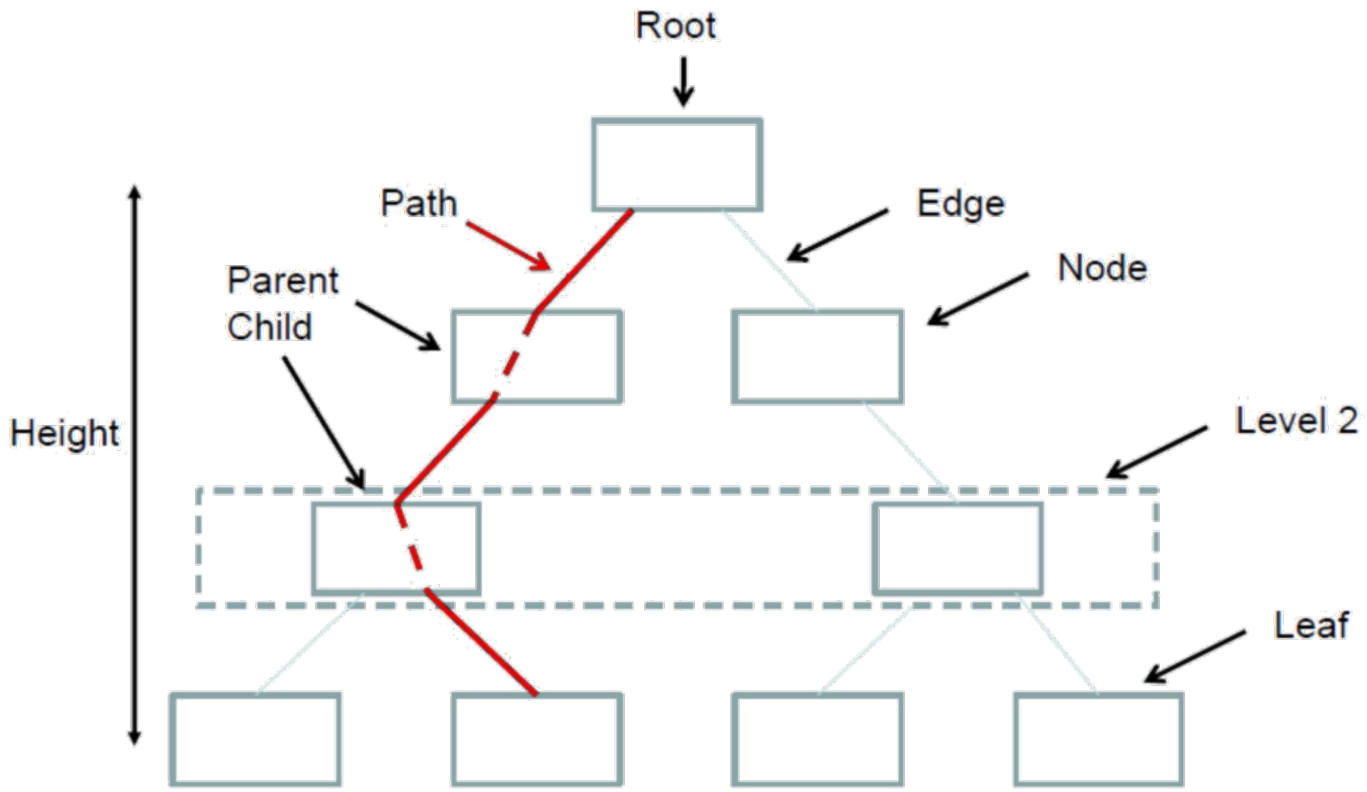
  consecutive nodes in the sequence form an edge) exists from the root to any node or leaf

- The ***depth*** is the distance from a node to the root

  of the tree. The depth of the root is 0
  (The depth of other nodes is 1 + the depth of its parent)

- The **level** of a node is the ***number of edges + 1*** along a path from the root to the node
- The ***height*** of a tree is the length of the longest path from the root to a leaf

Tree Terminology



Tree Terminology

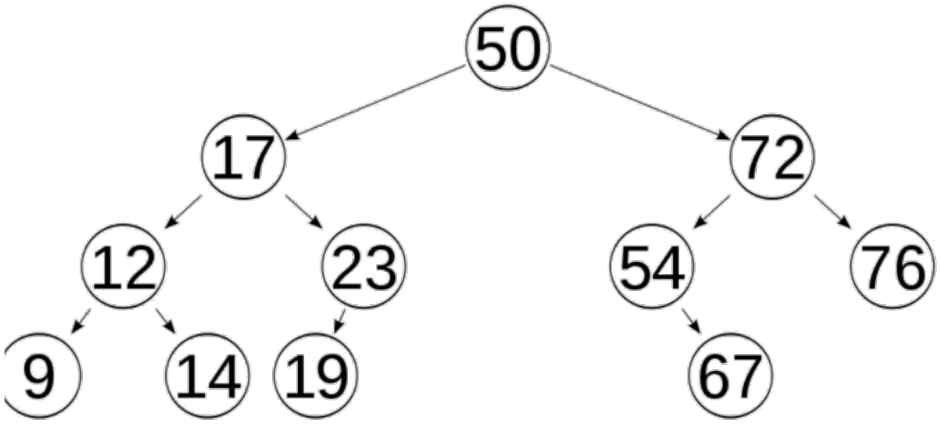
- A node is the ***ancestor*** of another node if it is on the path between the root and the other node, and a node that can be reached along a path away from its ancestor is a ***descendant***
- A **sub-tree** is a portion of a tree that can be viewed as a complete tree in itself

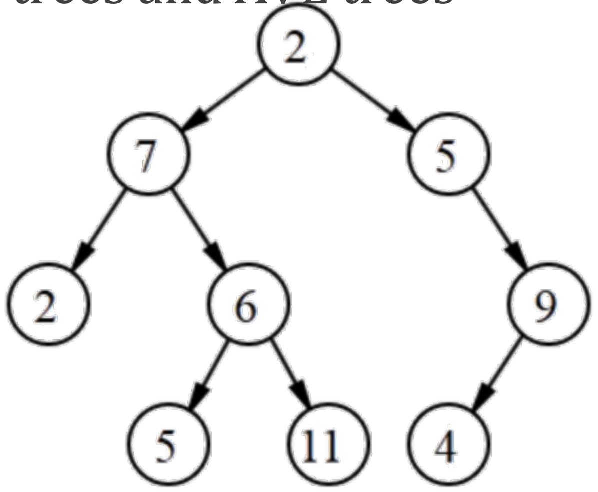
- *The inheritance relation between classes* *in Java* *forms a* *tree* *with the* *Object class* *as the* *root* *and* *ancestor* *of all* *other classes*

- A Binary Tree is a tree with a maximum of 2 children per parent

Balanced Binary Tree

- A *balanced tree* is a binary tree in which the depth of the two sub-trees of every node never differ by more than 1
- Or a tree is considered height-balanced if the height of the left sub-tree and the height of the right sub-tree do not differ by more than one level of hierarchy.
- Two well-known examples are red-black trees and AVL trees





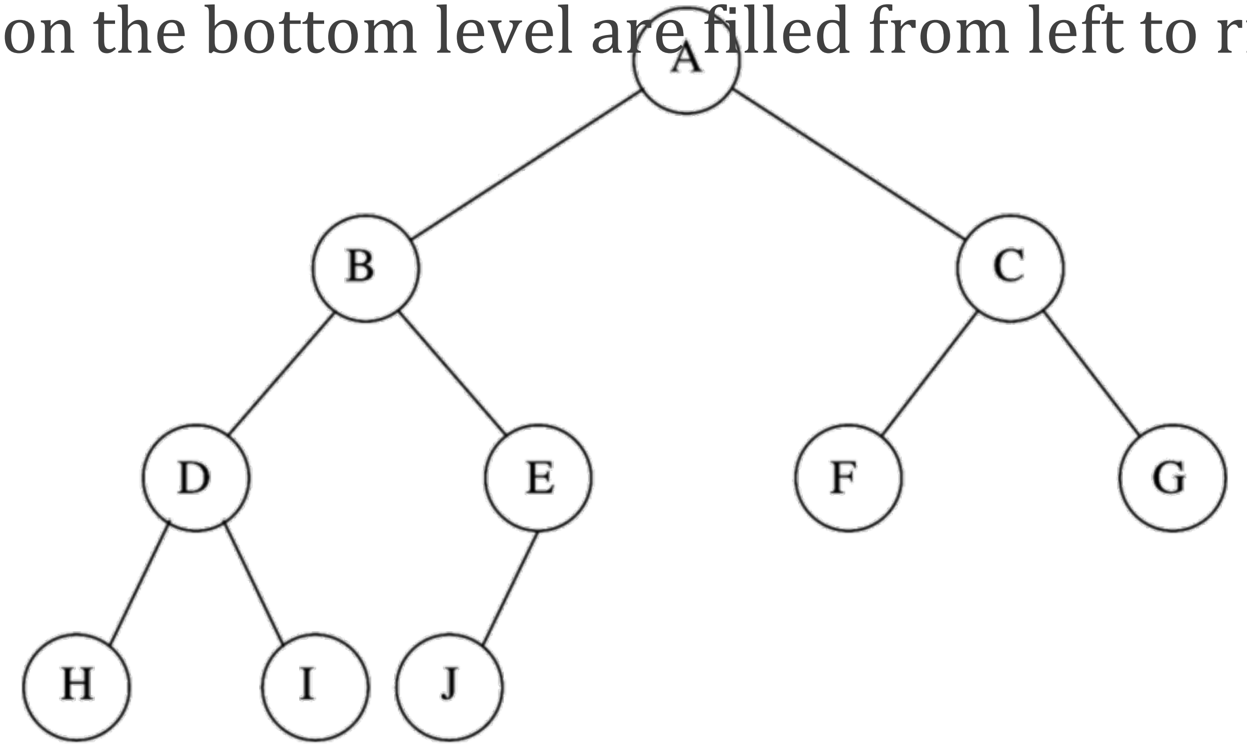
Balanced Tree

Unbalanced Tree

Complete Binary Tree

- A *complete tree* is a tree with every level, except

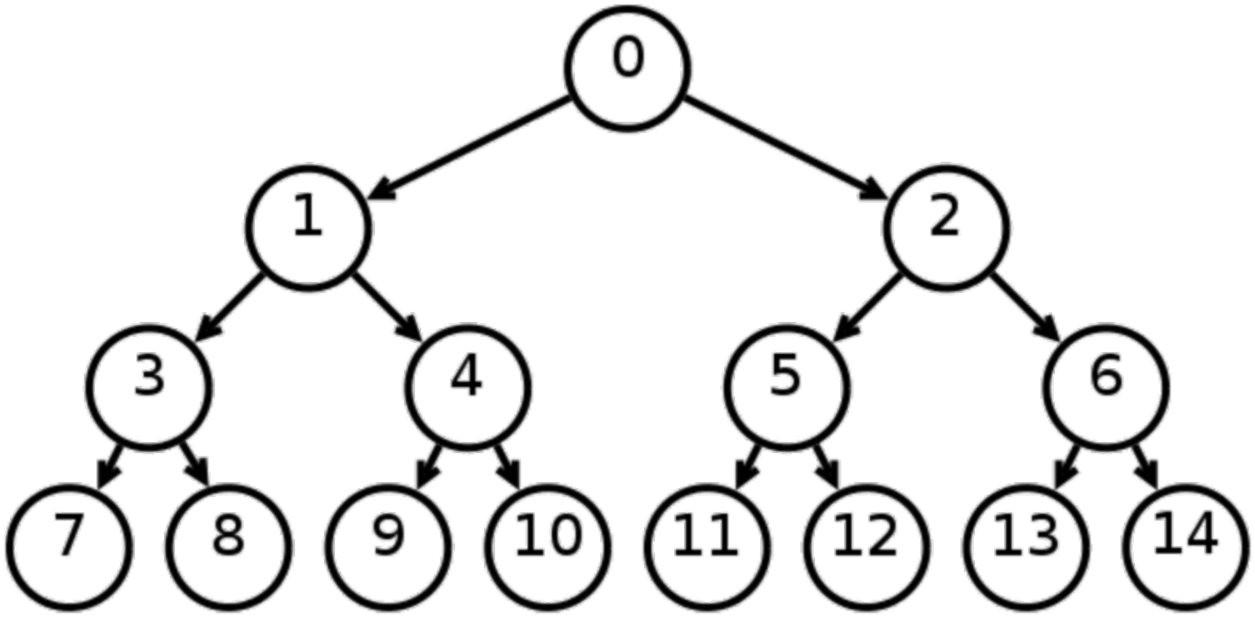
possibly the deepest, are completely filled, and all the leaves on the bottom level are filled from left to right



Full Binary Tree

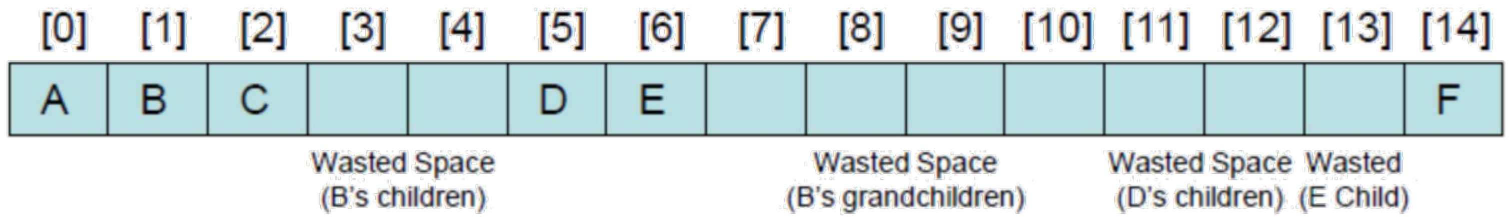
- A *full tree* has all leaves at the same level and

every parent node has exactly the same number of children



Binary Tree – Using An Array

- Each element of the array is an object with a reference to a data element and an int index for each of its two children
- For any element stored in the array in position n:
  - its left child will be stored in position: 2*n + 1
  - its right child will be stored in position: 2*n + 2
- Disadvantages

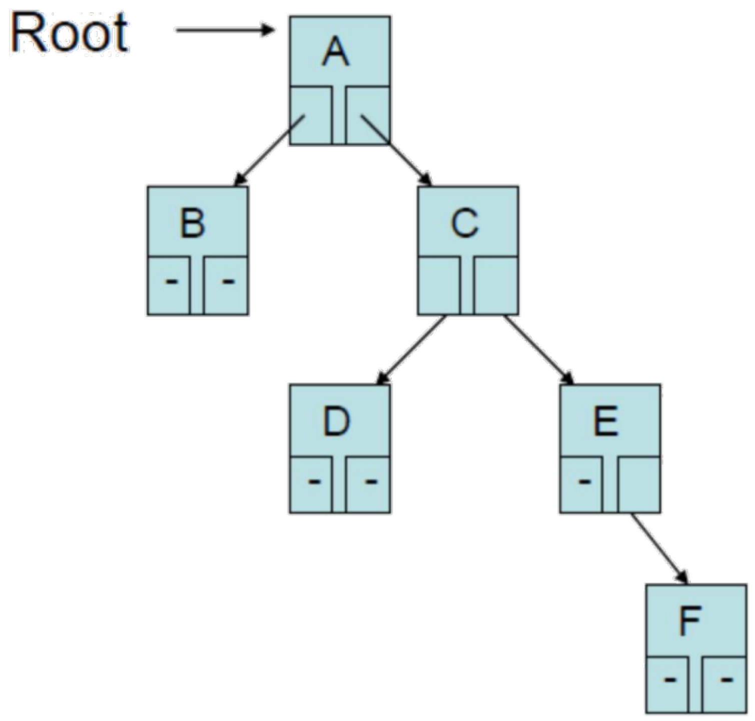

[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14]
A
B
C
D
E
F
Wasted Space (B's children)
Wasted Space (B's grandchildren)
Wasted Space (D's children)
Wasted (E Child)

- If the tree is not complete or nearly complete, the array may have many empty elements - wasted memory
- Removing an element from the array requires shifting the remaining elements and alteration of index values

Binary Tree – Using Linked List

- Consists of a Node class containing a reference to the data

and a left and a right reference to 2 child nodes

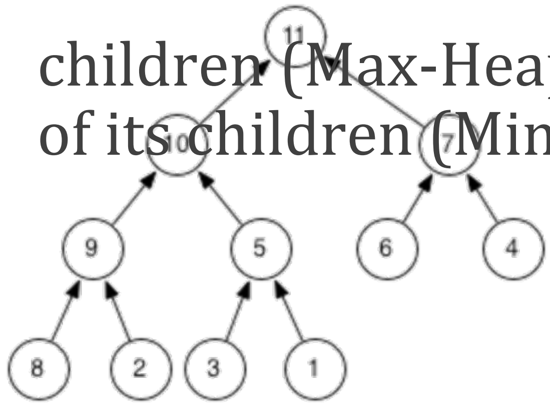


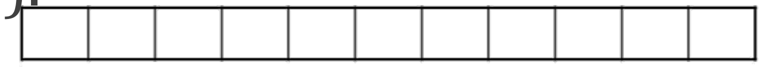
Heap or Binary Heap

- All levels of the tree, except possibly the last one

  (deepest) are fully filled, and, if the last level of the tree is not complete, the nodes of that level are filled from left to right – a complete tree .
- Each node is greater than or equal to each of its

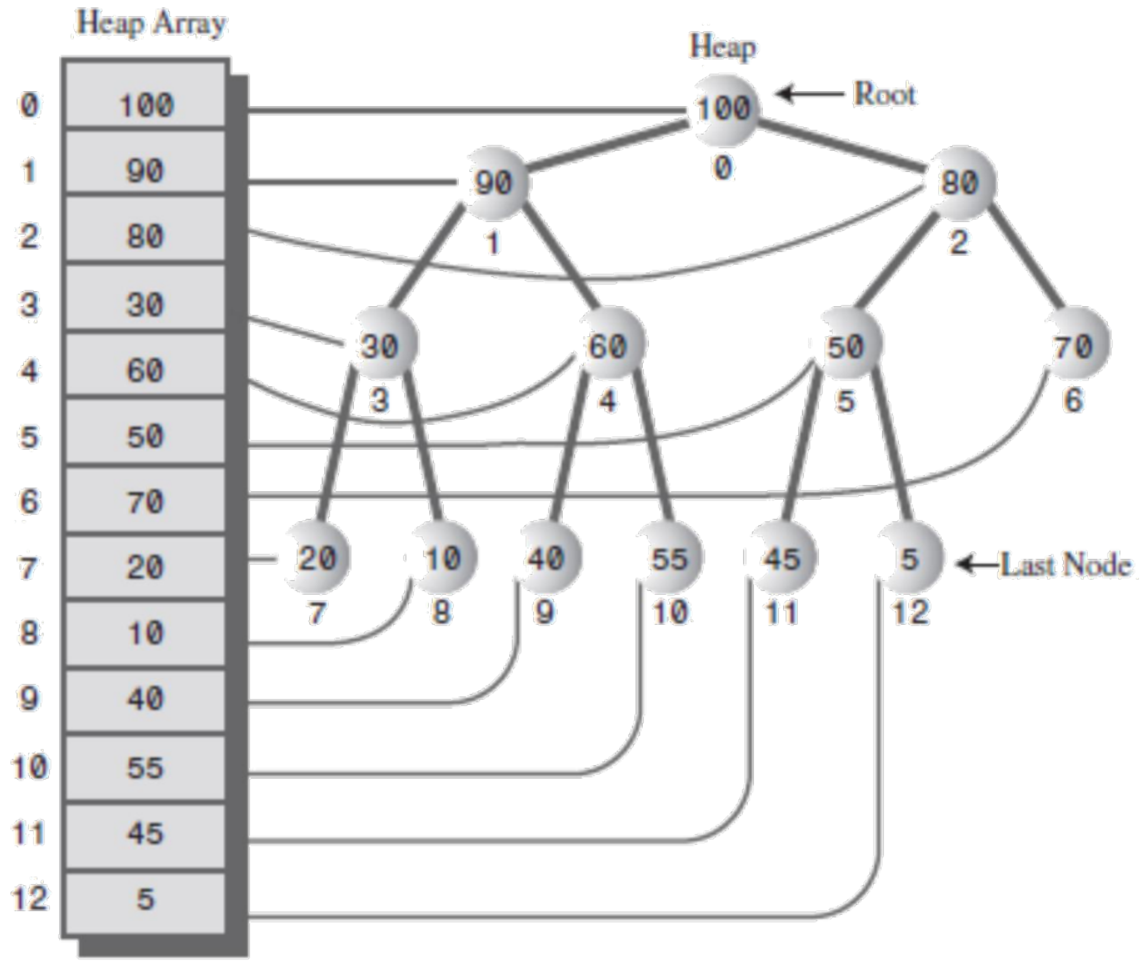
  children (Max-Heap), or less than or equal to each of its children (Min-Heap).





11	**10**	**7**	**9**	**5**	**6**	**4**	**8**	**2**	**3**	**1**
0	1	2	3	4	5	6	7	8	9	10

Max-Heap Representation



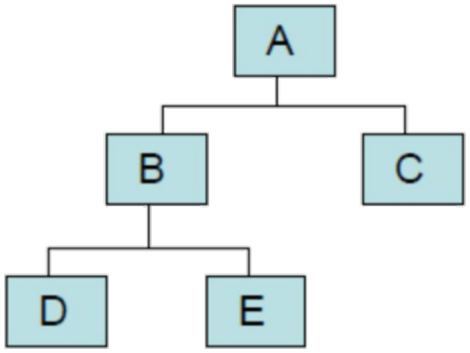
Binary Tree Traversal

- Pre-order (Depth first)
  - Visit node, traverse left child, traverse right child
- In-order (Depth first)

- Traverse left child, visit node, traverse right child
- Post-order (Depth first)
  - Traverse left child, traverse right child, visit node
- Level-order (Breadth first)

- Visit all the nodes at each level, one level at a time

Binary Tree Traversal



- Pre-order: A, B, D, E, C
- In-order: D, B, E, A, C
- Post-order: D, E, B, C, A

- Level-order: A, B, C, D, E

Pre-order Traversal

```
preorder(root);
…
private void preorder(BinaryNode <T> root){
        if (root != null) {
                System.out.println(root.data);
                preorder(root.left);
                preorder(root.right);
        }
}
```

In-order Traversal

```
inorder(root);
…

private void inorder(BinaryNode <T> root){
        if (root != null) {
                inorder(root.left);
                System.out.println(root.data);
                inorder(root.right);
        }
}
```

Post-order Traversal

```
postorder(root);
…

private void postorder(BinaryNode <T> root){
	if (root != null) {
		postorder(root.left);
		postorder(root.right);
		System.out.println(root.data);
	}
}
```

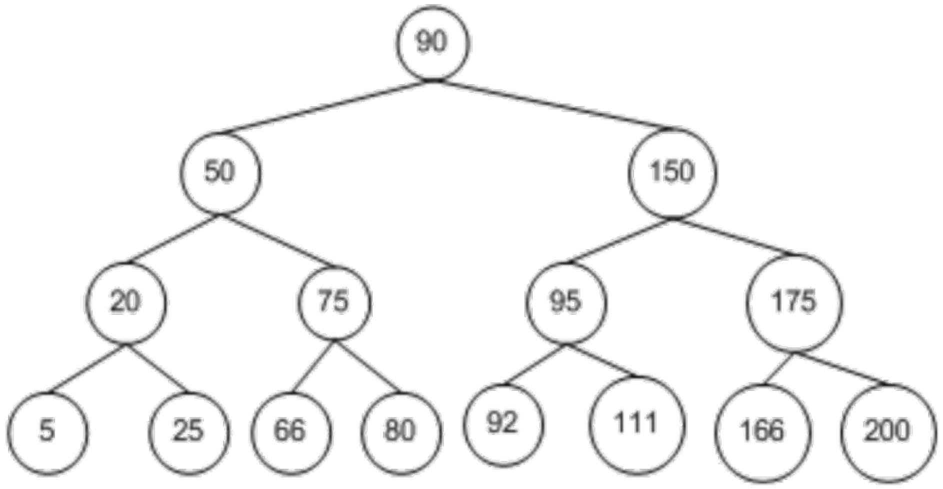
Level-order Traversal

```
private void levelOrderPrint(BinaryNode<T> root) {
     Queue<T> queue = new Queue<T>();
     UnorderedList<T> list = new UnorderedList<T>();
     queue.enQueue((T) root);
     while(!queue.isEmpty()) {
         BinaryNode<T> item = (BinaryNode<T>) queue.deQueue();
         if(item != null) {
              list.addToRear((T) item);
              if(item.left != null)
                   queue.enQueue((T) item.left);
              if(item.right != null)
                   queue.enQueue((T) item.right);
         }
     }
     System.out.println(list);
}
```

Binary Search Tree

- A *binary tree* such that for any node *n*, every

descendant node's value in the left *subtree* of *n* is less than the value of *n*, and every descendant node's value in the right *subtree* is greater than the value of *n*.



Building The Heap

```
ublic static void buildHeap(int[] a,int size) {
     for(int i = size / 2 - 1; i > =0; i--) // start from last parent to first parent
          heapify(a,i,size);

}

ublic static void heapify(int[] a, int i, int size) {
     int l = 2 * i + 1;       // left child
     int r = 2 * i + 2;       // right child
     int largest=i;      // parent
     if(l <= size-1 && a[l] > a[i])        // find the larger of parent and left child
          largest = l;
     else
          largest = i;
     if(r <= size-1 && a[r] > a[largest]) // find the larger of parent and right child
          largest = r;
     if(largest != i) {                   // swap parent and larger child if necessary
          int temp = a[i];
          a[i] = a[largest];
          a[largest] = temp;
          heapify(a, largest, size);    // repeat heapify until all children are in a heap
     }
}
```