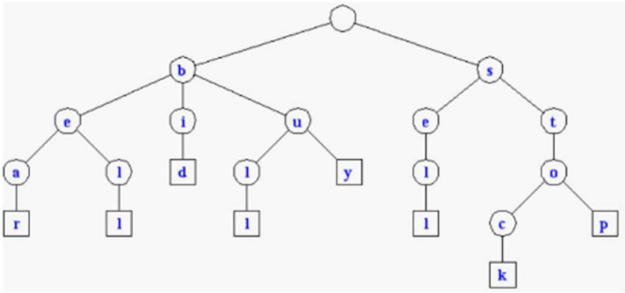
What is Trie?

Trie is an efficient information retrieval data structure. A Trie is a tree in which each node has many children. The value at each node consists of 2 things.

1. A character
2. A Boolean to say whether this character represents the end of a word.

Tries are also known as Prefix Trees.



Representation: -

```
public class TrieNode {
	private char c;
	private HashMap<Character, TrieNode> children = new HashMap<>();
	private boolean isLeaf;

	public TrieNode() {
	}

	public TrieNode(char c) {
		this.c = c;
	}

	public HashMap<Character, TrieNode> getChildren() {
		return children;
	}

	public void setChildren(HashMap<Character, TrieNode> children) {
		this.children = children;
	}

	public boolean isLeaf() {
		return isLeaf;
	}

	public void setLeaf(boolean isLeaf) {
		this.isLeaf = isLeaf;
	}
}
```

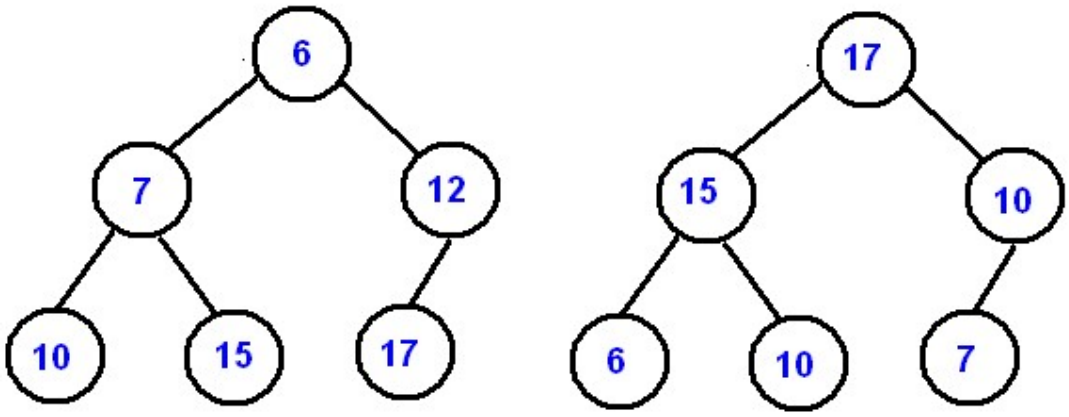
Basic Operations: -

```
public class Trie {
	private TrieNode root;

	public Trie() {
		root = new TrieNode();
	}

	public void insert(String word) {
		HashMap<Character, TrieNode> children = root.getChildren();
		for (int i = 0; i < word.length(); i++) {
			char c = word.charAt(i);
			TrieNode node;
			if (children.containsKey(c)) {
				node = children.get(c);
			} else {
				node = new TrieNode(c);
				children.put(c, node);
			}
			children = node.getChildren();
			if (i == word.length() - 1) {
				node.setLeaf(true);
			}
		}
	}

	public boolean search(String word) {
		HashMap<Character, TrieNode> children = root.getChildren();
		TrieNode node = null;
		for (int i = 0; i < word.length(); i++) {
			char c = word.charAt(i);
			if (children.containsKey(c)) {
				node = children.get(c);
				children = node.getChildren();
			} else {
				node = null;
				break;
			}
		}
		if (node != null && node.isLeaf()) {
			return true;
		} else {
			return false;
		}
	}
}
```

A **binary heap** is a complete binary tree which satisfies the heap ordering property.

There are two types of heap: -

1. **Min heap:** every parent is less than or equal to its children
2. **Max heap:** Every parent is greater than or equal to its children



A binary heap must be a complete tree, children are added at each level from left to right, and usually implemented as arrays. The maximum or minimum value will always be at the root of the tree, this is the advantage of using a heap.

For Heapify, the process of converting a binary tree into a heap, is often has to be done after an insertion or deletion.

To implement the heaps as arrays: -

1. We put the root at array[0]
2. Then we traverse each level from left to right, and so the left child of the root would go into array[1],its right child would be into array[2],etc.

For the node at array[i], we can get left child using this formula **(2i + 1)**, for the right child we can use this one **(2i + 2),** and for parent item **floor((i — 1) / 2)**.This works just with complete binary trees.

To insert into heap: -

1. Always add new items to the end of the array
2. Then we have to fix the heap(heapify process)
3. We compare the new item against its parent
4. If the item is greater than its parent, we swap it with its parent
5. We then rinse and repeat

Implementation: -

```
public class Heap {

	private int[] heap;
	private int size;

	public Heap(int capacity) {
		heap = new int[capacity];
	}

	public void insert(int value) {
		if (isFull()) {
			throw new IndexOutOfBoundsException("Heap is full");
		}
		heap[size] = value;
		fixHeapAbove(size);
		size++;
	}

	private void fixHeapAbove(int index) {
		int newValue = heap[index];
		while (index > 0 && newValue > heap[getParent(index)]) {
			heap[index] = heap[getParent(index)];
			index = getParent(index);
		}
		heap[index] = newValue;
	}

	public boolean isFull() {
		return heap.length == size;
	}

	public int getParent(int index) {
		return (index - 1) / 2;
	}
}
```

Default implementations in Java: -

```
public class HeapUsingJava {
    static PriorityQueue<Integer> minHeap = new PriorityQueue<>();
    static PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Comparator.reverseOrder());
}
```