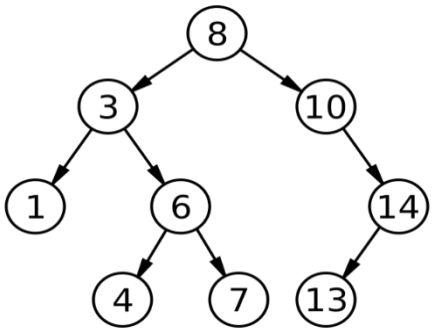
Binary Search Tree

Definition:

A binary search tree is a binary tree. It may be empty; if not then it satisfies following properties:

1. Each node has exactly one key and the keys in the tree are distinct.
2. The keys (if any) in the left subtree are smaller than the key in the root.
3. The keys (if any) in the right subtree are larger than the key in the root.
4. The left and right subtree are also binary search tree.

Understanding with an example:



In this figure, it is a Binary Search tree because of satisfying all four properties. 8 is root node and left sub-tree are smaller than root. And right subtree is larger than the root and left and right subtree are also binary search tree.

Node Structure:

Each node contains three parts:

- Data: The value stored in the node.
- Left Child: A pointer to the left child node
- Right Child: A pointer to the right child node.

```
typedef struct binarySearchTree{
        int data;
        struct binarySearchTree  * leftChild;
        struct binarySearchTree  *rightChild;
} binarySearchTree;
```

Operations: Traversal, Insertion, Deletion, Searching

Searching in BST:

Since the definition of binary search tree is recursive , it is recursive, it is easiest to describe a recursive search method. Suppose we wish to search for a node whose key is K.

- We begin by examining the root node. If the root is NULL , K is not exist in the tree. Otherwise we compare K with the key in the root. if k equals to the root's key then terminates Successfully.
- If K is less than root's key, we search the left subtree of the root. Similarly, if K is larger than root's key value, we search the right subtree if the root.
- This process is repeated until K is found or the remaining subtree is NULL, If K is not found after a null subtree is reached the we can say K is not present in the tree.

Algorithm

```
binarySearchTree * search(binarySearchTree *root, int  key)
{
        if(!root) return NULL;
        else if(K==root→data.key)  return &(root→data);
        else if(k<root→data.key) return search(root→leftChild,K);
        return search(root→rightChild,K);
}
```

Complexity:

- Time complexity: O(h) [where h is height of the tree]
- Space complexity: O(h) due to recursive function calls.

Example:

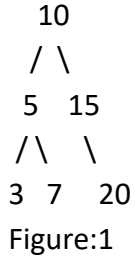


Figure:1

if you search for 7:
then K=7,

1. [K<10] → Go to the left subtree.
2. [K>5] → Go to the right subtree.

3. [K==7] → Key found; return the address of the data field in the node.

Insertion in BST:

To insert, we must first verify that the key is different from those of existing values. To do this we search the tree, if the search is unsuccessful then we insert the value.

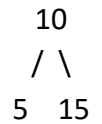


Figure 2

If we want to add 7 in the above tree then first compare 7 with root(10) ,We can see that 7 is less than 10 go to the left subtree now 7 is greater then 5 goto right subtree of 5 but we received NULL so search unsuccessful now insert 7 as the right child of 5.

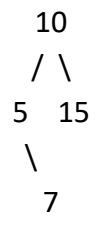


Figure 3

If we want to add 20 in the above tree then first compare 20 with root(10) ,We can see that 7 is greater than 10 goto the right subtree now 10 is greater than 51 goto right subtree of 15 but we received NULL so search unsuccessful now insert 20 as the right child of 15.

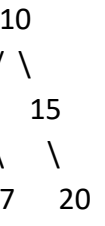


Figure 4 (Final Tree)

Analysis of insertion:

The time required to search the tree for K is O(h) where h is its height. The remainder of the algorithm takes Θ(1) time , so the overall time needed by insert is O(h).

Deletion in BST:

Deletion of a leaf is quite easy. For example to delete 20 from the tree of figure 4 the right field of its parent is set to 0(NULL) and the freed . this gives us the tree of figure 3.

The deletion of a nonleaf that has only one child is also easy . The node to be deleted is freed, and its single child takes the place of freed node, so to delete 5 from the figure 3 , we simply change the pointer from the parent node to the single child node , generating figure 5.

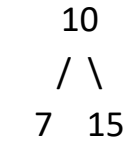


Figure 5

When The pair to be deleted is in a nonleaf node that has two children, the node to be deleted is replaced by either largest pair in its left subtree or the smallest one in its right subtree. Then we proceed to delete this replacing node from the subtree from which it was taken. For instance ,if we wish to delete 10 from the tree of figure 5. Then we can replace it either 7 or 15. Now node 7 is moved into the root and figure 6 is generated.

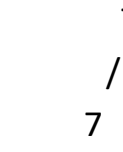


Figure 6

Now we must delete the second 7. Since the second 7 is leaf so the left field of root is set to 0(NULL).

Figure 7

References:

1. https://www.geeksforgeeks.org/binary-search-tree-data-structure/
2. Cormen, Thomas H., et al. *Introduction to algorithms*. MIT press, 2022.

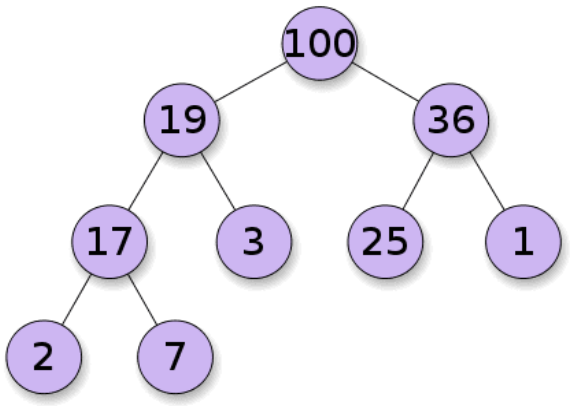
Heap Data Structure

Definition: Heap is an almost complete binary tree (All the levels in the tree are completely filled except possibly the last level) .

Types:

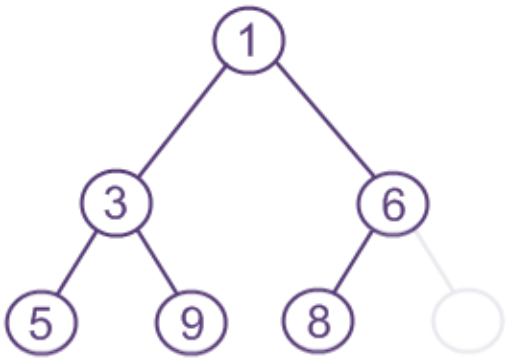
1. Max heap: For every node k, the value of node is less than or equal to its parent value.

A[parent(k)]>= A[k]



2. Min heap: for every node k, the value of node is greater than or equal to its parents.

A[parent(k)]<= A[k]

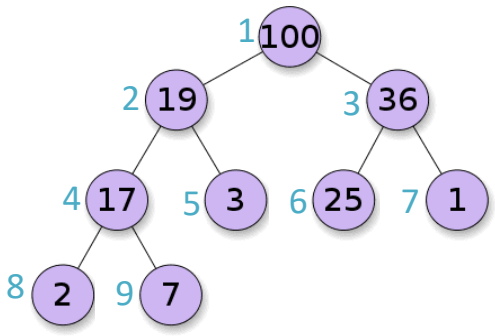


To build a heap from a tree it must follows Structural property and ordering property.

Heap as an array:

1. The first element in the array corresponds to index i=1;
2. Index of Parent node = i/2.
3. Index of left child of parent: 2*i
4. Index of right child of parent: 2*i+1

Example:



100	19	36	17	3	25	1	2	7

Heapify:

Heapify is a key operation in heap data structures. It ensures that a binary tree satisfies the heap property:

- **Max-Heap Property**: Every parent node is greater than or equal to its children.
- **Min-Heap Property**: Every parent node is less than or equal to its children.

Heapify is used in operations like heap construction, insertion, and deletion.

Max Heapify:

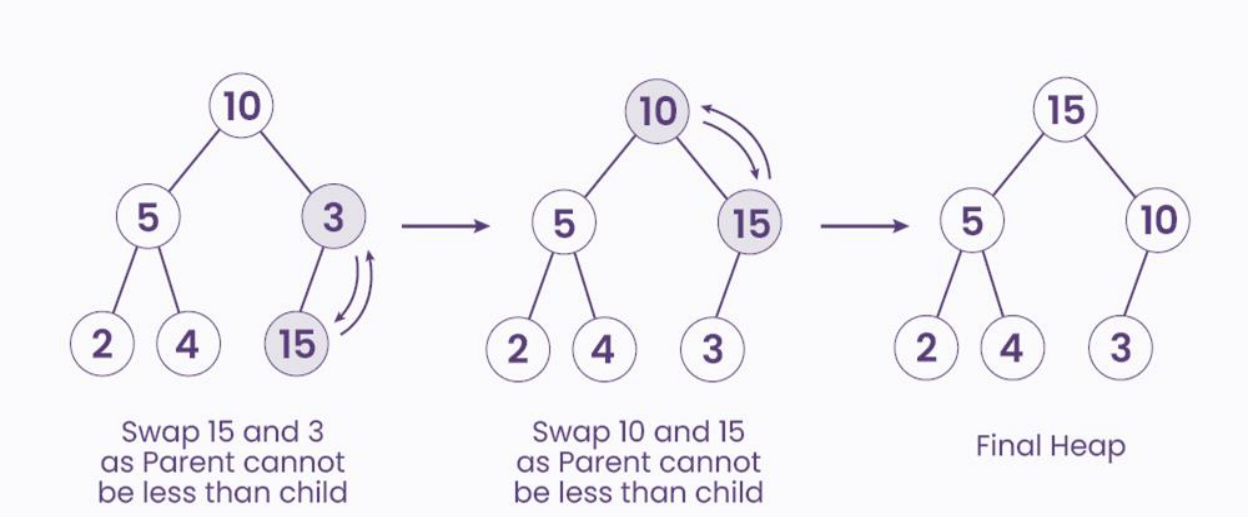
```
BUILD-MAX-HEAP(A, n)
1  A.heap-size = n
2  for i = ⌊n/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

Max heap also known as max tree i.e. the key value in each node is no smaller than the key values in its children.

```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4      largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7      largest = r
8  if largest ≠ i
9      exchange A[i] with A[largest]
10     MAX-HEAPIFY(A, largest)
```

The procedure **MAX-HEAPIFY** maintains the max-heap property in an array A with a heap-size attribute and an index i. It assumes the binary trees rooted at LEFT(i) and RIGHT(i) are max-heaps but allows A[i] to violate the property by being smaller than its children. MAX-HEAPIFY makes A[i] "float down" until the subtree rooted at i satisfies the max-heap property.

At each step, it finds the largest among A[i], A[LEFT(i)], and A[RIGHT(i)]. If A[i] is largest, the subtree is already a max-heap. Otherwise, it swaps A[i] with the largest child and recursively calls itself on the affected subtree. This ensures the max-heap property is restored throughout the tree.



Insertion in max heap:

The insertion operation in a **max-heap** involves adding a new element while maintaining the heap property, where every parent node is greater than or equal to its child nodes. Here's a detailed explanation of the process:

Steps for Inserting an Element into a Max-Heap

1. **Add the Element**:
   - Insert the new element at the end of the heap (to maintain the complete binary tree structure).
2. **Heapify Up (Bubble Up)**:
   - Compare the newly added element with its parent.
   - If the new element is greater than its parent, swap them.
   - Repeat this process until the heap property is restored or the element reaches the root.
3. **Stop Condition**:
   - The element is no longer greater than its parent.
   - The element becomes the root.

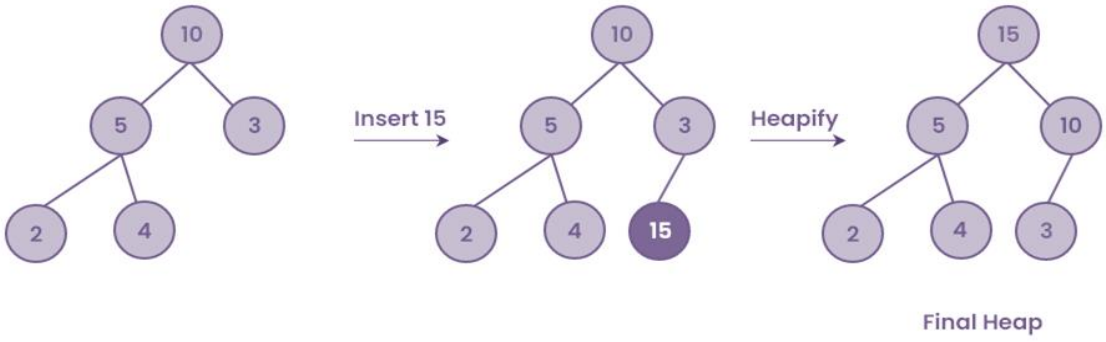


Figure: Insertion of Heap

Deletion in Max Heap:

Deleting an element from a heap, particularly the root (the maximum element in a max-heap or the minimum in a min-heap), is one of the most common operations. The process involves maintaining the heap property after the deletion. Here's a detailed explanation of the deletion process in a heap:

Steps for Deletion in a Heap

Deleting the Root Element (Max for Max-Heap, Min for Min-Heap):

1. **Remove the Root**:
    - Replace the root with the last element in the heap.
    - This ensures the heap remains a complete binary tree.
2. **Heapify Down (Bubble Down)**:
    - Compare the new root with its children.
    - Swap it with the largest child (for max-heap) or the smallest child (for min-heap) to restore the heap property.
    - Repeat this process until the heap property is restored or the element reaches a leaf.
3. **Stop Condition**:
    - The element is larger (max-heap) or smaller (min-heap) than its children.
    - The element reaches a leaf node.

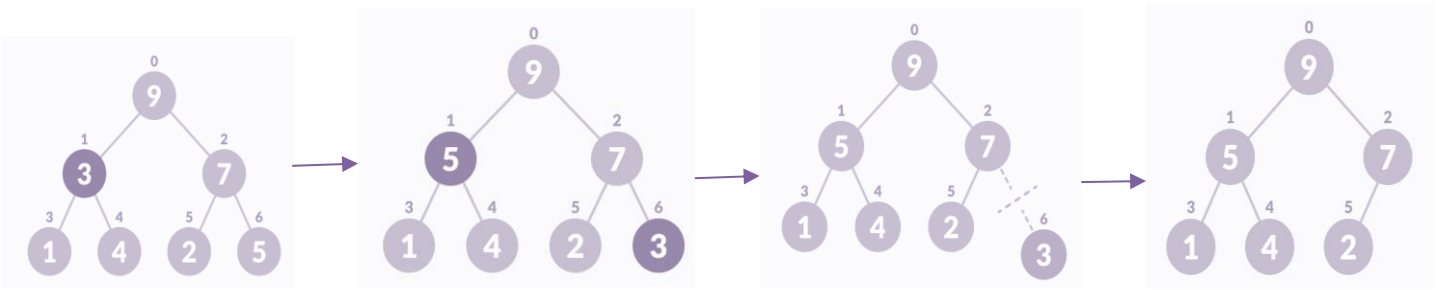


Figure: Deletion of Heap

Heapsort:

Heap Sort: Detailed Explanation

Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure. It is an efficient algorithm with a time complexity of O(n log n), making it suitable for large datasets. Heap sort works by repeatedly building a heap and extracting the root element (maximum or minimum, depending on the heap type).

Steps of Heap Sort

1. **Build a Heap**:
    - Convert the input array into a binary heap.
    - For a max-heap, the largest element is at the root. For a min-heap, the smallest element is at the root.
2. **Extract the Root**:
    - Swap the root element with the last element in the heap.
    - Reduce the size of the heap (exclude the last element, which is now sorted).
    - Restore the heap property by heapifying down the new root.
3. **Repeat**:
    - Continue extracting the root and heapifying until all elements are sorted.

Algorithm

1. Building the Heap

To build a heap from an array:

- Start from the last non-leaf node and apply heapify down.
- A node at index i has:
    - Left child: 2i
    - Right child: 2i+1

2. Heapify

Heapify is a process of restoring the heap property:

- Compare a node with its children.
- Swap it with the larger child (for max-heap) or smaller child (for min-heap).
- Repeat the process until the heap property is restored.

3. Sorting

- Swap the root (maximum/minimum) with the last element.
- Reduce the heap size.
- Heapify the root.

Pseudocode

Heapify Function

```
def heapify(arr, n, i):
```

```
    largest = i  # Assume the root is the largest
    left = 2 * i + 1  # Left child index
    right = 2 * i + 2  # Right child index

    # Check if left child exists and is greater than root
    if left < n and arr[left] > arr[largest]:
        largest = left

    # Check if right child exists and is greater than largest
    if right < n and arr[right] > arr[largest]:
        largest = right

    # If largest is not root, swap and continue heapifying
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)
```

Heap Sort Function

```
def heap_sort(arr):
    n = len(arr)

    # Step 1: Build a max-heap
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    # Step 2: Extract elements one by one
    for i in range(n - 1, 0, -1):
        # Swap the root (largest element) with the last element
        arr[0], arr[i] = arr[i], arr[0]

        # Heapify the reduced heap
        heapify(arr, i, 0)
```

Time Complexity

1. **Building the Heap**: O(n)
2. **Heapify Operations**: O(log n) for each of the nn elements.
    - Total: O(n log n)

Space Complexity

- In-place sorting: O(1) auxiliary space.

Advantages

- No need for additional memory (in-place sorting).
- Efficient for large datasets.

Disadvantages

- Not a stable sort (relative order of equal elements is not preserved).
- Slightly slower than QuickSort on average.

References:

[1]. Cormen, Thomas H., et al. *Introduction to algorithms*. MIT press, 2022.

[2]. https://en.wikipedia.org/wiki/Heapsort

[3]. https://www.geeksforgeeks.org/introduction-to-max-heap-data-structure/