# Data and File Structures

## (MCA-102)

### Unit – 2

[Tree]

by

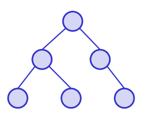
**Dr. Sunil Pratap Singh**
**(Assistant Professor, BVICAM, New Delhi)**

**2021**



## Tree

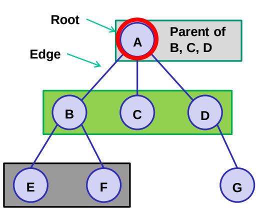
- A Tree is a non-linear data structure which organizes data in a hierarchical structure.
- In Tree, every individual element is called a *node* which stores the data value.
- Each node is connected by an *edge* to another node.
- Example:
  - Tree with 6 nodes.



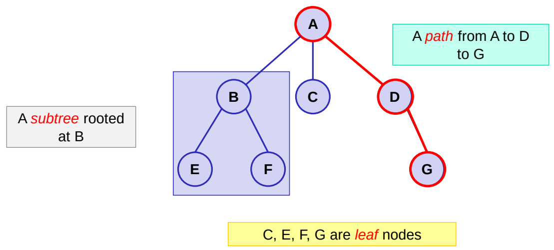


## Tree Terminology

- Root
- Rooted Tree
- Degree of a Node
- Degree of a Tree
- Leaf Node / Terminal Node
- Non-Terminal Node / Internal Node
- Siblings
- Level
- Height of a Tree
- Path
- Subtree
- Forest

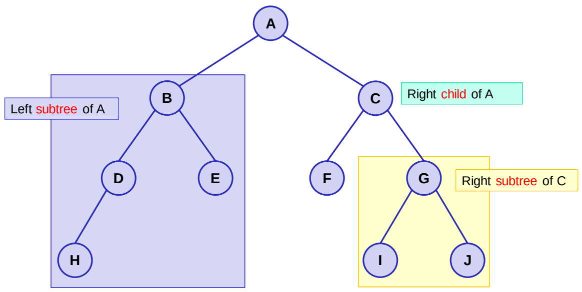


## Tree Terminology: Example





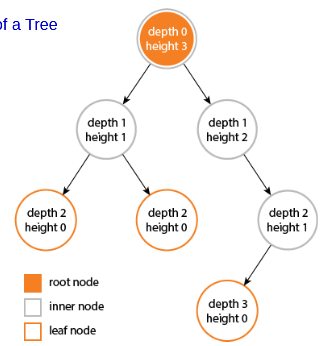

## Tree Terminology: Example





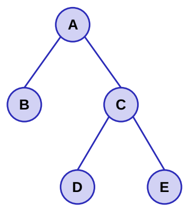

## Tree Terminology: Example

Depth and Height of a Tree



## Binary Tree

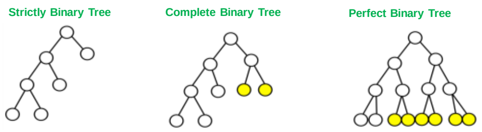
- In a normal tree, every node can have any number of children.
- A binary tree is tree in which each node can have a maximum of 2 children.
- Example:



## Types of Binary Tree

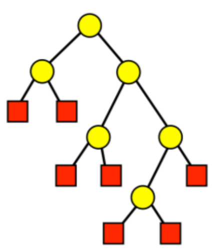
- Strictly (Full) Binary Tree
  - If every node has either 0 or 2 children, a binary tree is called **Strictly Binary Tree**.
- Complete Binary Tree
  - A **Complete Binary Tree** is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible
- Perfect Binary Tree
  - A binary tree is **Perfect Binary Tree** in which all internal nodes have two children and all leaves are at same level.



## Extended Binary Tree

- The full binary tree obtained by adding dummy nodes (external nodes) to a binary tree is called **Extended Binary Tree**.



## Properties of Binary Tree

- A binary tree with $n$ internal nodes has exactly $n+1$ external nodes.
- For any non-empty binary tree with $n_0$ leaf nodes and $n_2$ nodes of degree 2, $n_0 = n_2 + 1$
- The maximum number of nodes on level $i$ of a binary tree is $2^i$, $i \geq 0$.
- The maximum number of nodes in a binary tree of height $k$ is $2^{k+1} - 1$.
- The height of a binary tree with $n$ nodes is at least $\lceil \log_2(n+1) - 1 \rceil$ and at most $n - 1$.
- The number of distinct binary trees with $n$ nodes is $\frac{(2n)!}{(n+1)!\, n!}$.



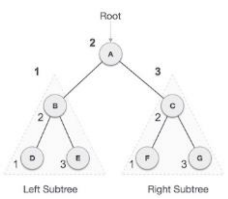
## Binary Tree Traversal

- Traversal is a process to visit all the nodes of a tree and may print their values too.
- There are three ways which we use to traverse a tree:
  - In-order Traversal
  - Pre-order Traversal
  - Post-order Traversal



## In-order Traversal

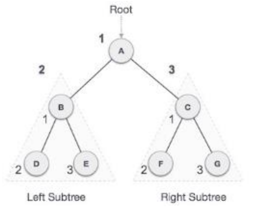
- In this traversal method, the left subtree is visited first, then the root and later the right sub-tree.
- We should always remember that every node may represent a subtree itself.




Output: D – B – E – A – F – C – G

## Pre-order Traversal

- In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

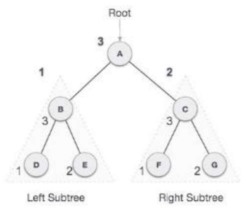



Output: A – B – D – E – C – F – G



## Post-order Traversal

- In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.




Output: D – E – B – F – G – C – A



## Inorder Traversal with Stack (Non-Recursive)

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 = poppedItem->right

   c. Go to step 3.

5) If current is NULL and Stack is empty then we are done.

## Preorder Traversal with Stack (Non-Recursive)

**1)** Create an empty Stack S.

**2) Push** the root node to S.

**3)** While the Stack is not empty, then

- **a. Pop** the top item from S and print.
- **b. Push** the poppedItem->right item to S.
- **c. Push** the poppedItem->left item to S.

## Postorder Traversal with Stack (Non-Recursive)

**1)** Create two empty Stacks **S1** and **S2**.

**2) Push** the root node to **S1**.

**3)** While the Stack **S1** is not empty, then

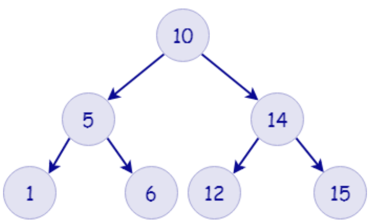
- **a. Pop** the top item from **S1** and Push it into **S2**.
- **b. Push** the poppedItem->left item to **S1**.
- **c. Push** the poppedItem->right item to **S1**.

**4) Pop** out all the items from Stack **S2** and Print.

## Example

Find **Inorder**, Preorder and Postorder traversal sequence for the given tree.



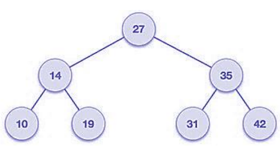
## Representation of Binary Tree

- Using Array
- Using Linked List



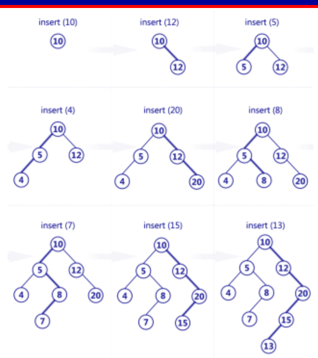
## Binary Search Tree

- A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties:
  - The left sub-tree of a node has a key less than or equal to its parent node's key.
  - The right sub-tree of a node has a key greater than to its parent node's key.






## Insertion in Binary Search Tree



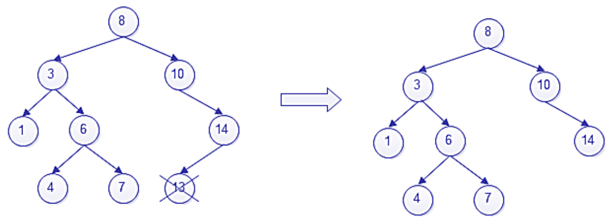
## Deletion in Binary Search Tree

- To delete an element from, there are three cases:
  - The node to be deleted has no children.
  - The node to be deleted has 1 child node.
  - The node to be deleted has 2 child nodes.



## Deletion in Binary Search Tree

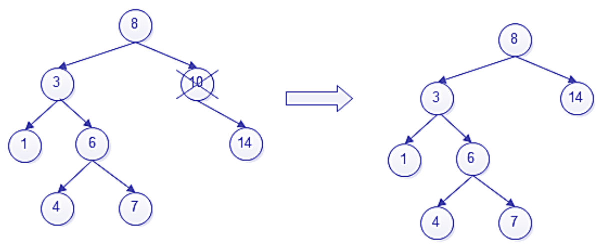
- The node to be deleted has no children.
  - Simply delete the node.





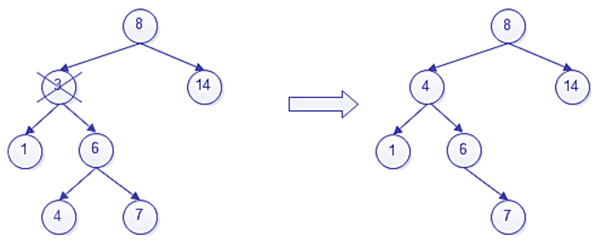

## Deletion in Binary Search Tree

- The node to be deleted has 1 child node.
  - Replace the node with its child node and delete it.



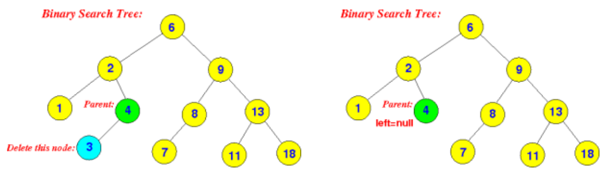
## Deletion in Binary Search Tree

- The node to be deleted has 2 child nodes.
    - Find its in-order successor node, and replace it with in-order successor, then delete it.



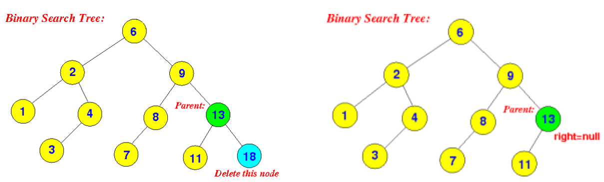

 U2.25

## Deletion in Binary Search Tree




 U2.26

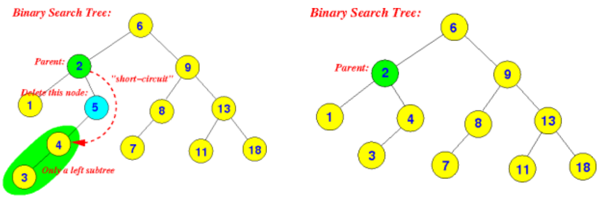
## Deletion in Binary Search Tree




 U2.27

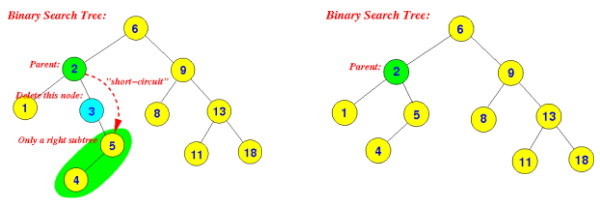
 

## Deletion in Binary Search Tree





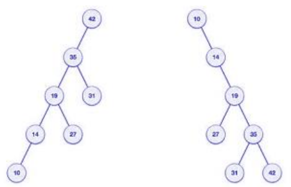

## Deletion in Binary Search Tree






## AVL Tree

- What if the input to binary search tree comes in a sorted (ascending or descending) manner? It will then look like this:

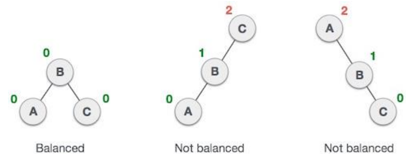


If input 'appears' non-increasing manner    If input 'appears' in non-decreasing manner

- Named after their inventor Adelson, Velski & Landis, AVL trees are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the Balance Factor.

## AVL Tree

Balance Factor = height(left-subtree) − height(right-subtree)
or
Balance Factor = height(right-subtree) − height(left-subtree)




- If the difference in the height of left/right and right/left sub-trees is more than 1, the tree is balanced using some rotation techniques.



## How Rotation Works to Balance the Tree

- Let the newly inserted node be **w**:
  1. Perform standard BST insert for **w**.
  2. Starting from **w**, travel up and find the first unbalanced node. Let **z** be the first unbalanced node, **y** be the child of **z** that comes on the path from **w** to **z** and **x** be the grandchild of **z** that comes on the path from **w** to **z**.
  3. Re-balance the tree by performing appropriate rotations on the sub-tree rooted with **z**.
     - There can be 4 possible cases that needs to be handled as **x**, **y** and **z** can be arranged in 4 ways.

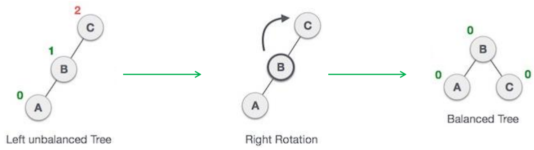


## Insertion in AVL Tree

- 4 Possible Cases for Unbalanced Node:
  1. Left-Left Case: x is the left child of y and y is the left child of z
  2. Left-Right Case: x is the right child of y and y is the left child of z
  3. Right-Left Case: x is the left child of y and y is the right child of z
  4. Right-Right Case: x is the right child of y and y is the right child of z

## Example: Right Rotation (LL Case)

- If a tree becomes unbalanced, when a node is inserted in the left subtree of the left subtree, then we perform a single right rotation.
- Example: Insert C, B and A

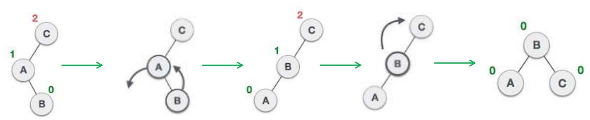



- In our example, node C has become unbalanced as A is inserted in the left subtree of C's left subtree. We performed the right rotation by making C as the right-subtree of B.



## Example: Left Right Rotation (LR Case)

- The LR Rotation is combination of left rotation followed by right rotation.
- Example: Insert C, A and B

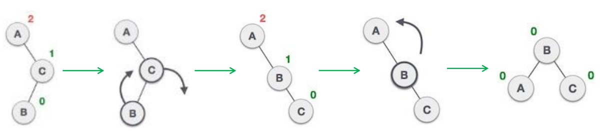


- In our example, node C has become unbalanced as B is inserted in the right subtree of C's left subtree.
  - Perform the left rotation on the left subtree of C. This makes A, the left subtree of B.
  - Perform the right-rotation on the tree, making B the new root node of this subtree.



## Example: Right Left Rotation (RL Case)

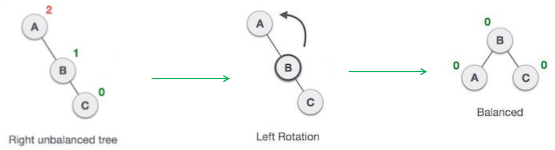
- The RL Rotation is combination of right rotation followed by left rotation.
- Example: Insert A, C and B



- In our example, node A has become unbalanced as B is inserted in the left subtree of A's right subtree.
  - Perform the right rotation along C. This makes C, the right subtree of B.
  - Perform the left rotation on the tree, making B the new root node of this subtree.

## Example: Left Rotation (RR Case)

- If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation.
- Example: Insert A, B and C




- In our example, node A has become unbalanced as C is inserted in the right subtree of A's right subtree. We performed the left rotation by making A as the left-subtree of B.

## Example: Complex Situation of LL Case

```
         z                                       y
        / \                                    /   \
       y   T4      Right Rotate (z)           x      z
      / \          - - - - - - - - ->        /  \   /  \
     x   T3                                 T1  T2  T3  T4
    / \
  T1   T2
```

## Example: Complex Situation of LR Case

```
     z                               z                           x
    / \                            /   \                        /  \
   y   T4  Left Rotate (y)        x    T4  Right Rotate(z)    y      z
  / \      - - - - - - - - ->    /  \      - - - - - - - ->  / \    / \
T1   x                          y    T3                    T1  T2 T3  T4
    / \                        / \
  T2   T3                    T1   T2
```

## Example: Complex Situation of RL Case

```
   z                              z                              x
  / \                            / \                            / \
T1   y   Right Rotate (y)     T1   x    Left Rotate(z)        z     y
    / \  - - - - - - - - ->       / \  - - - - - - - - ->    / \   / \
   x   T4                       T2   y                      T1 T2 T3 T4
  / \                               / \
T2   T3                           T3   T4
```

## Example: Complex Situation of RR Case

```
  z                                y
 / \                             /   \
T1  y     Left Rotate(z)        z     x
   / \   - - - - - - - ->      / \   / \
  T2  x                       T1 T2 T3  T4
     / \
    T3  T4
```

## Example: Drawing AVL Tree

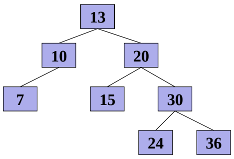
- Draw AVL Tree by inserting the values: 15, 20, 24, 10, 13, 7, 30, 36

## Example: Drawing AVL Tree

- AVL Tree by inserting the values: 15, 20, 24, 10, 13, 7, 30, 36



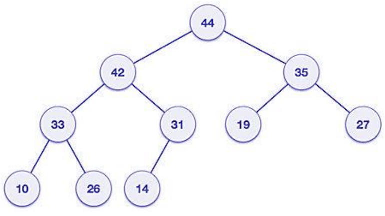
## Heap

- Heap is a special balanced binary tree with special characteristics.
- Heap can be defined as a collection of keys (data elements) which satisfies the following characteristics:
  - Ordering: Nodes must be arranged in a order according to values.
  - Structural: All levels in a heap must full, except last level and nodes must be filled from left to right strictly (Complete Binary Tree)
- There are two types of heap:
  - Max Heap
  - Min Heap

## Max Heap

- When the value of the root node is greater than or equal to either of its children, it is called Max Heap.
- Max heap is used for **Heap Sort**

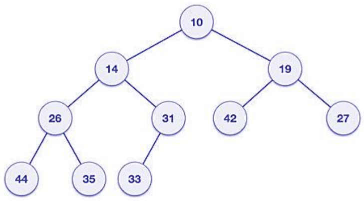


## Min Heap

- When the value of the root node is less than or equal to either of its children, it is called Min Heap.
- Min heap is used to implement **Priority Queue**.
- It may also be used to implement **Heap Sort**.



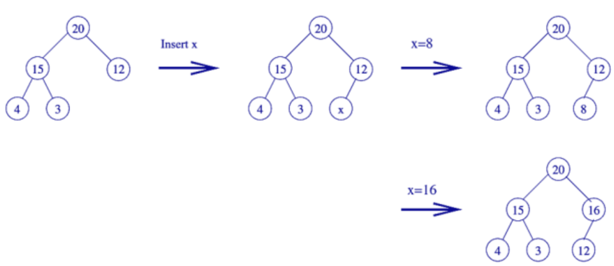



## Max Heap Construction: Algorithm

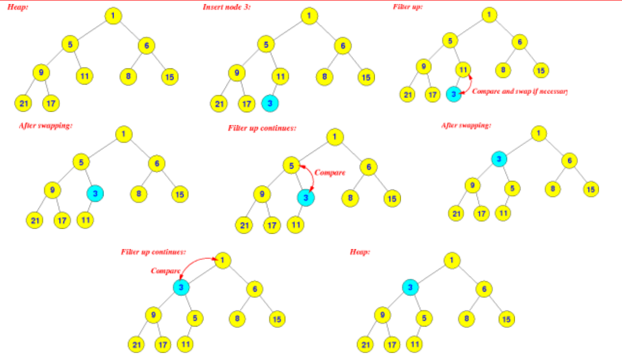
- First increase the heap size by 1, so that it can store the new element.
- Insert the new element at the end of the Heap.
- This newly inserted element may distort the properties of Heap for its parents. So, in order to keep the properties of Heap, heapify this newly inserted element following a bottom-up approach.



## Max Heap Insertion: Example



## Min Heap Insertion: Example





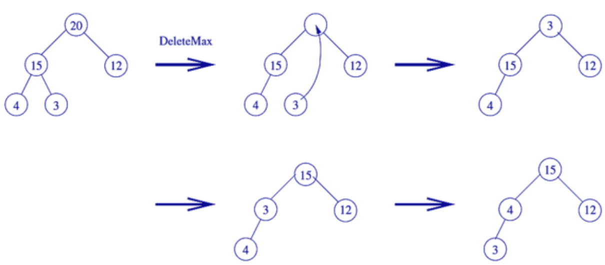

## Max Heap Deletion: Algorithm

**Deletion of Root Node**

- Replace the root or element to be deleted by the last element.
- Delete the last element from the Heap.
- Since, the last element is now placed at the position of the root node. So, it may not follow the heap property. Therefore, heapify the last node placed at the position of root.



## Max Heap Deletion (Root): Example



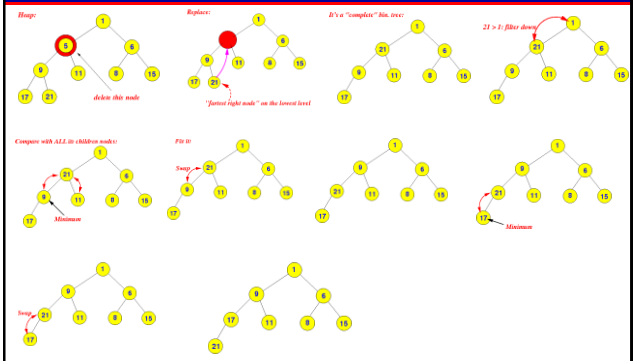
## Max Heap Deletion (Specific): Algorithm

**Deletion of a Specific Node**

- Delete a node from the array.
- Replace the deletion node with the "farthest right node" on the lowest level of the Binary Tree
- Heapify (fix the heap):
  - If the value in replacement node is greater then its parent node, filter the replacement node UP the binary tree.
  - Else Filter the replacement node DOWN the binary tree

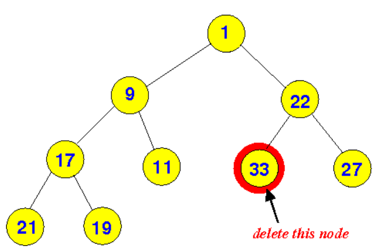


## Min Heap Deletion (Specific): Example

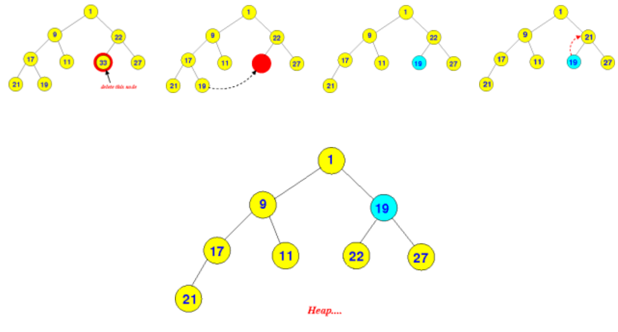




## Min Heap Deletion: Example



## Min Heap Deletion: Example




 U2.55

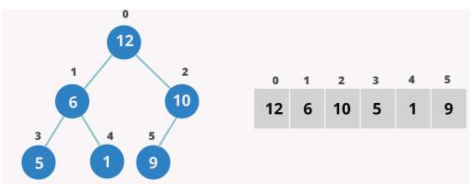
## Heap Sort: Step-by-Step Process

- In max-heaps, largest element will always be at the root. Heap Sort uses this property of heap to sort the array.
- Heap sort is an in-place algorithm, i.e., does not use any extra space, like merge sort.
- Complexity: O(n log n)
- Procedure:
  1) Build a max-heap of elements in array.
  2) Swap the root element with last element of array.
  3) Reduce the size of the heap by 1 and heapify the root element so that we have highest element at root.
  4) Repeat the steps 2 and 3, until all the items of the list are sorted.

 U2.56

## Heap Sort: Working Example

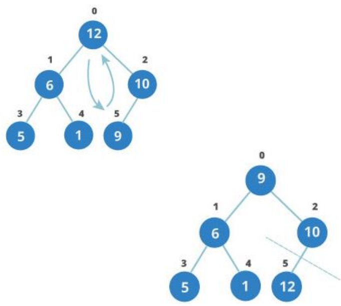
- Let the elements of the array are: 1, 12, 9, 5, 6, 10. For these given elements, the max heap is constructed as follows:




 U2.57

## Heap Sort: Working Example

- Swapping the root with last element, and decreasing the size of heap:

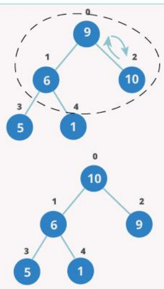


| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 9 | 6 | 10 | 5 | 1 | 12 |



## Heap Sort: Working Example

- Heapify the root element:



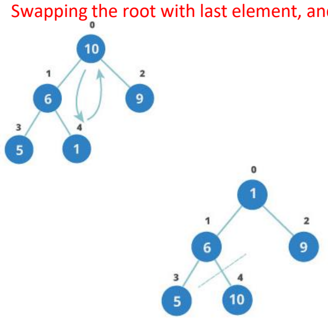
| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 9 | 6 | 10 | 5 | 1 | 12 |

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 10 | 6 | 9 | 5 | 1 | 12 |



## Heap Sort: Working Example

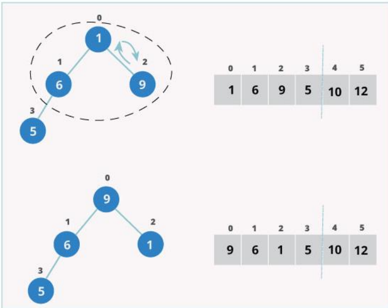
- Swapping the root with last element, and decreasing the size of heap:



| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 6 | 9 | 5 | 10 | 12 |

## Heap Sort: Working Example

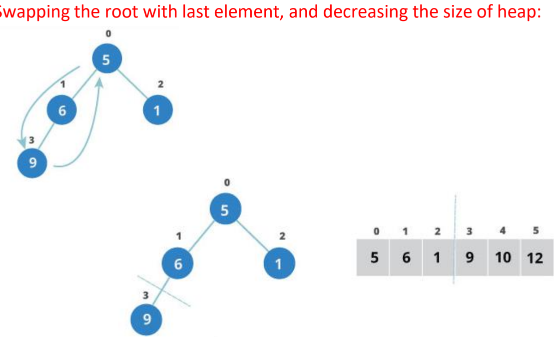
- Heapify the root element:






## Heap Sort: Working Example

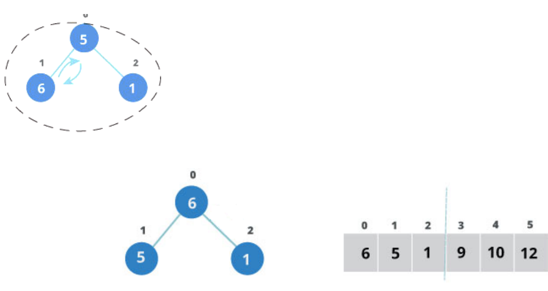
- Swapping the root with last element, and decreasing the size of heap:





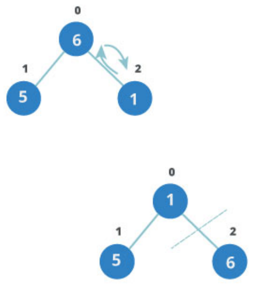

## Heap Sort: Working Example

- Heapify the root element:



## Heap Sort: Working Example

- Swapping the root with last element, and decreasing the size of heap:

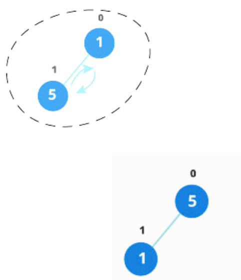



| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 5 | 6 | 9 | 10 | 12 |

## Heap Sort: Working Example

- Heapify the root element:

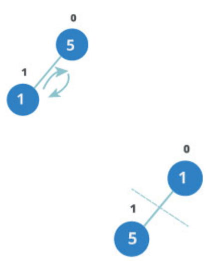



| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 5 | 1 | 6 | 9 | 10 | 12 |

## Heap Sort: Working Example

- Swapping the root with last element, and decreasing the size of heap:



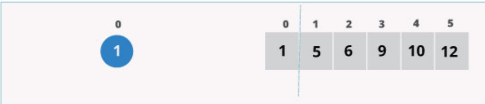

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 5 | 6 | 9 | 10 | 12 |

## Heap Sort: Working Example

- Heapify the root element:




- Final sorted elements:

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 5 | 6 | 9 | 10 | 12 |

 U2.67

## Program Code for Heap Sort

```
for(i=n-1; i>=0; i--)
{
    swap(a[0], a[i])
    heapify(a, 0, i)
}
```

 U2.68

## Priority Queue

- Consider a networking application where four requests arrived to the queue in the order of R1 requires 20 units of time, R2 requires 2 units of time, R3 requires 10 units of time and R4 requires 5 units of time. Queue is as follows:


- Now, check waiting time for each request to be complete.
  - R1 : 20 units of time
  - R2 : 22 units of time
  - R3 : 32 units of time
  - R4 : 37 units of time
- **Average waiting time for all requests = (20+22+32+37)/4 ≈ 27 units of time**
- That means, if we use a normal queue data structure to serve these requests the average waiting time for each request is **27 units of time**.

 U2.69

## Priority Queue

- Now, consider another way of serving these requests. If we serve according to their required amount of time.
- Then, the waiting time for each request to be complete will be as follows:
  - R2 : 2 units of time
  - R4 : 7 units of time
  - R3 : 17 units of time
  - R1 : 37 units of time
- **Average waiting time for all requests = (2+7+17+37)/4 ≈ 15 units of time**
- **Priority queue is a variant of queue data structure in which insertion is performed in the order of arrival and deletion is performed based on the priority.**



## Types of Priority Queue

- Max Priority Queue
- Min Priority Queue



## Max Priority Queue

- In max priority queue, elements are inserted in the order in which they arrive the queue and always maximum value is removed first from the queue.
- For example, assume that we insert in order 8, 3, 2, 5 and they are removed in the order 8, 5, 3, 2.

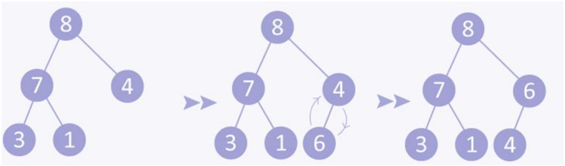
## Min Priority Queue

- In min priority queue, elements are inserted in the order in which they arrive the queue and always minimum value is removed first from the queue.
- For example, assume that we insert in order 8, 3, 2, 5 and they are removed in the order 2, 3, 5, 8.



## Insertion in Priority Queue

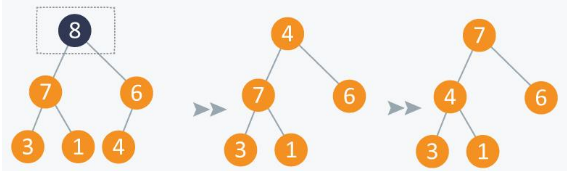
- Initially there are 5 elements in Priority Queue.
- Insert value 6.






## Deletion in Priority Queue

- Extract Maximum



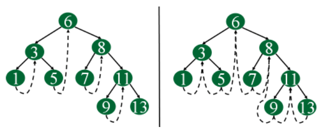
## Threaded Binary Tree

- When a binary tree is represented using linked list representation, we use NULL pointer for nodes which do not have children.
- In any binary tree linked list representation, there are more number of NULL pointer than actual pointers.
- A. J. Perlis and C. Thornton proposed new binary tree called "Threaded Binary Tree", which make use of NULL pointer to improve its traversal processes.
- The idea of Threaded Binary Trees is to make in-order traversal faster and do it without recursion.
- To convert Binary Tree into Threaded Binary Tree, first find the in-order traversal of that tree.



## Threaded Binary Tree

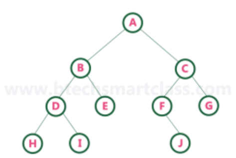
- **One Way Threading**:
  - Each node is threaded towards either the in-order predecessor or successor (left **OR** right) means all right null pointers will point to in-order successor **OR** all left null pointers will point to in-order predecessor.
- **Two Way Threading**:
  - Each node is threaded towards both, in-order predecessor and successor (left **AND** right) which means all right null pointers will point to in-order successor **AND** all left null pointers will point to in-order predecessor.





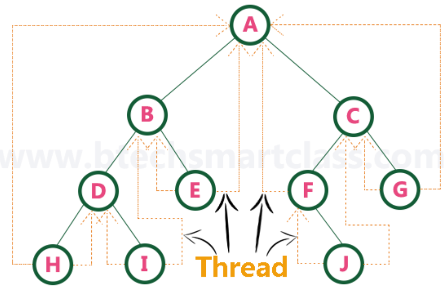

## Threaded Binary Tree: Example

- In-Order : H D I B E A F J C G




- Left child pointers of nodes H, I, E, F, J and G are NULL.
- These NULLs are replaced by address of their in-order predecessor, respectively (I to D, E to B, F to A, J to F and G to C), but here the node H does not have its in-order predecessor, so it points to the root node A.
- Right child pointers of nodes H, I, E, J and G are NULL.
- These NULLs are replaced by address of their in-order successor, respectively (H to D, I to B, E to A, and J to C), but here the node G does not have its in-order successor, so it points to the root node A.

## Threaded Binary Tree: Example







## m-way Tree

- The concept of two-way search tree (BST) can be extended to create an m-way search tree. The m-way tree has following properties:
  - Each node has any number of children from 2 to $M$, i.e., all nodes have degree $<= M$, where $M >= 2$
  - Each node has keys ($K_1$ to $K_n$) and pointers to its children ($P_0$ to $P_n$), i.e., number of keys is one less than the number of pointers. The keys are ordered, i.e., $K_i < K_{i+1}$ for $1 <= i < n$
  - The subtree pointed by a pointer $P_i$ has key values less than the key value of $K_{i+1}$ for $1 <= i < n$
  - The subtree pointed by a pointer $P_n$ has key values greater than the key value of $K_n$
  - All subtrees pointed by pointers $P_i$ are m-way trees.



## B-Tree

- B-Tree is a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time.
- B-Tree was developed in the year of 1972 by Bayer and McCreight with the name **Height Balanced m-way Search Tree**.
- When data volume is large and does not fit in memory, an extension of the binary search tree to disk-based environment is the B-tree.
- Since the B-tree is always balanced (all leaf nodes appear at the same level), it is an extension of the balanced binary search tree.

## B-Tree

- The B in B-Tree technically doesn't represent a word. However some common characteristics can be summarized with words that begin with B, which is most likely the origin of the name.
  - **Balanced** – this is a self balancing data structure, which means that performance can be guaranteed when B-Trees are utilized.
  - **Broad** – as opposed to binary search trees, which grow vertically, B-Trees expand horizontally, so saying that they are broad is a suitable description.
  - **Bayer** – lastly the creator of B-Trees was named Bayer Rudolf. In all actuality this is probably the reason why B-Trees got their name.



## B-Tree vs. Binary Search Tree

| Basis for Comparison | B-Tree | Balanced Binary Search Tree |
|---|---|---|
| Essential Constraint | A node can have at max M number of child nodes(where M is the order of the tree). | A node can have at max 2 number of subtrees. |
| Use | It is used when data is stored on disk. | It is used when data is stored on RAM. |
| Height of the Tree | $\log_M N$ (where M is the order of the M-way tree) | $\log_2 N$ |



## B-Tree

- When the number of data elements (keys) are more, the data is read from disk in the form of blocks.
- Disk access time is very high compared to main memory access time.
- The main idea of using B-Trees is to **reduce** the number of disk accesses.
  - Most of the tree operations (search, insert, delete, max, min) require **O(h)** disk accesses where **h** is height of the tree.
  - Height of B-Trees is kept low by putting maximum possible keys in a B-Tree node.
  - Since each disk access exchanges a whole block of information between memory and disk rather than a few bytes, a node of the B-tree is expanded to hold more than two child pointers, up to the block capacity
  - Since **h** is low for B-Tree, total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees (like AVL Tree).

## B-Tree

- B-Trees are a good example of a data structure for external memory.
- B-Trees are commonly used in databases and files systems.
- Most database management systems have implemented the B-tree or its variants.



## Properties of B-Tree

- A height-balanced m-way tree is called as B-Tree.
- A B-Tree of order m, has following properties:
  - The root has at least two children. If the tree contains only a root, i.e., it is a leaf node then it has no children.
  - Each internal node (except the root) has between $\lceil m/2 \rceil$ and $m$ children.
    - Internal nodes stores up to $m - 1$ keys.
  - All paths from the root to leaves have the same length, i.e., all leaves are at the same level making it height balanced.
  - Leaves store between $\lceil m/2 \rceil - 1$ and $m$-1 data records

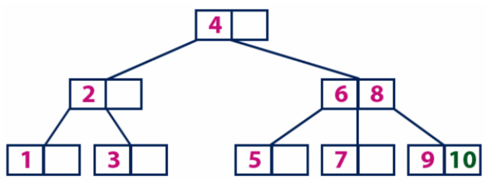


## Insertion in B-Tree

- Inserting into a B-tree starts out by "find"ing the leaf in which to insert.
- If there is room in the leaf for another data item, then we're done.
- If the leaf already has m-1 items, then there's no room.
  - Split the overfull node in half and pass the middle (median) value up to the parent for insertion there.
  - If the value passed up to the parent causes the parent to be over-full, then it too splits and passes the middle value up to its parent.

## Example of B-Tree

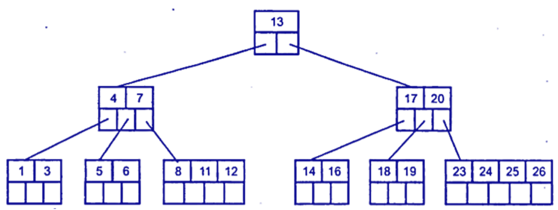
- Construct a B-tree of order 3 by inserting numbers from 1 to 10






## Example of B-Tree

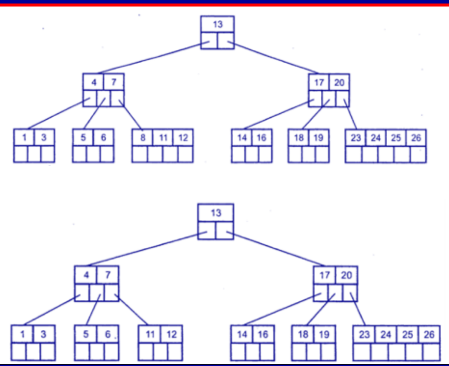
- Construct a B-Tree of order 5 for following numbers:
  3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25, 19





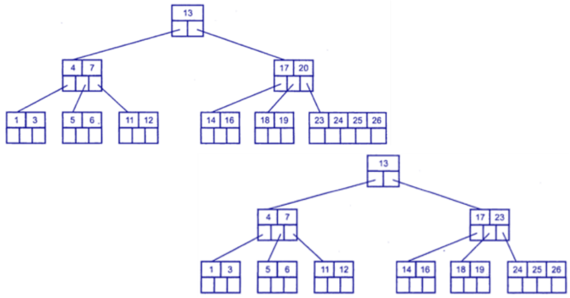

## Deletion in B-Tree

- Delete 8



## Deletion in B-Tree

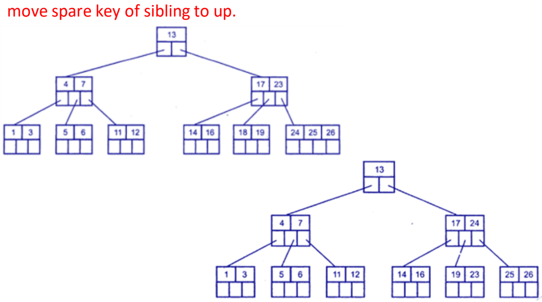
- Delete 20, which is not a leaf node so find its **successor** which is 23. Hence 23 will be moved up to replace 20.



## Deletion in B-Tree

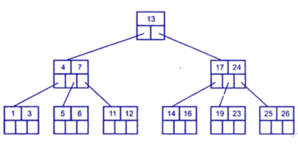
- Delete 18, which causes the node with **only one key**. The sibling node to immediate right has an extra key. In such case borrow a key from parent and move spare key of sibling to up.



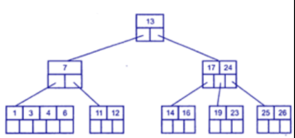
## Deletion in B-Tree

- Delete 5. This node has no extra keys nor siblings to left or right. In such a situation, we can combine this node with one of the siblings. That means remove 5 and combine 6 with node 1, 3. To make tree balanced, we have to move parent's key down. Hence move 4 down.




But, again internal node of 7 contains only 1 key, which is not allowed in B-Tree. We will then borrow a key from siblings but sibling has no spare key.

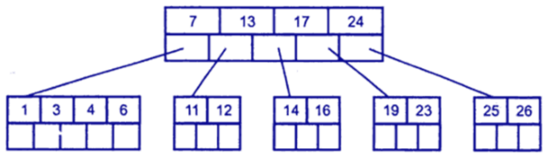
Hence, we need to combine 7 with 13 and 17, 24.



## Deletion in B-Tree



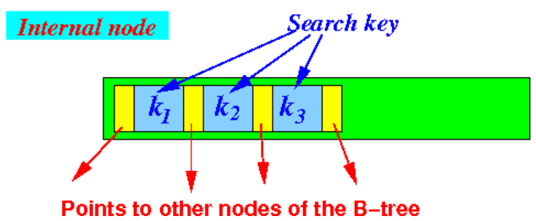



## B+ Tree

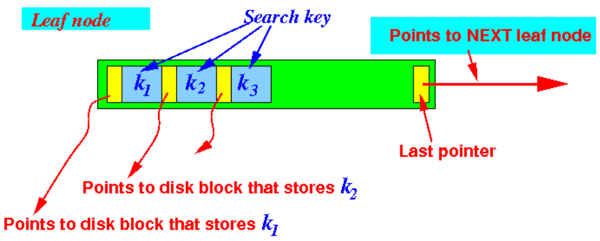
- B+ Tree is a variation of basic **B-Tree**.
- Leaves are connected to form a Linked List of keys in sequential order.
- B+ Tree has two parts:
  - ***Index Set*** that constitutes **Internal Nodes**
  - ***Sequence Set*** that constitutes **Leaves**
- **The linked Leaves can be accessed sequentially in addition to accessing them directly.**
  - This allows for extremely efficient range queries.



## B+ Tree: Structure of Internal Node

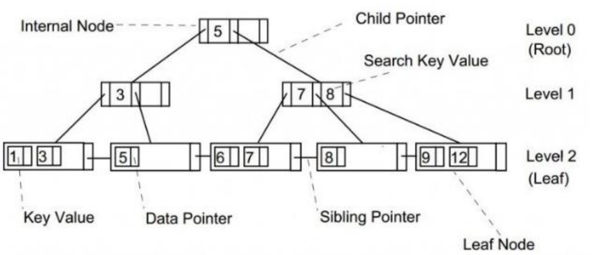


## B+ Tree: Structure of Leaf Node





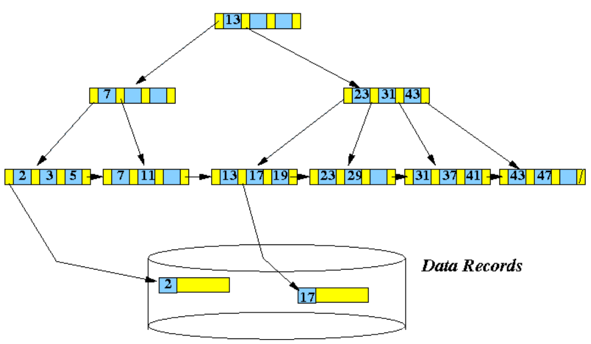

## Structure of B+ Tree




*B+ trees don't store data pointer in interior nodes, they are ONLY stored in leaf nodes.*



## Example of B+ Tree



## B Tree vs. $B^+$ Tree

- The leaf nodes in a B-Tree are not linked.
- $B^+$ Trees do not store data pointer in interior nodes.
- In B Tree, Internal Nodes and Leaves, both, store the search keys.
- $B^+$ Tree is efficient due to traversal performed with sibling pointers.



## B* Tree

- B*-tree is a variant of a B-tree that requires each internal node to be at least 2/3 full, rather than at least half full.



## Bibliography


- E. Horowitz and S. Sahani, "Fundamentals of Data Structures in C"
- Mark Allen Weiss, "Data Structures and Algorithm Analysis in C"
- R. S. Salaria, "Data Structure & Algorithms Using C"
- Schaum's Outline Series, "Data Structure"