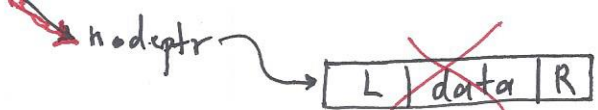
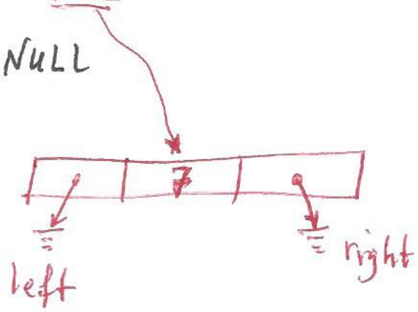
```
    } //end if
  } //end func.
```



Inserting a Node In BST:

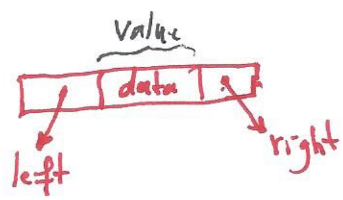
```
void IntBinaryTree:: insertNode (int num)
{    TreeNode *newNode = nullptr; // NULL
     newNode = new TreeNode;
     newNode -> data = num;
     newNode -> left = NULL;
     newNode -> right = NULL;
     insert (root, newNode);
} //end func.
```



Week 15, SAT:

```
// Deleting a Node in BST
void IntBinaryTree :: remove (int num)
{   deleteNode (num, root);
}
void IntBinaryTree :: deleteNode (int num , TreeNode *& nodeptr)
{   if(num < nodeptr->data)
        deleteNode (num, nodeptr->left)
    else if (num > nodeptr->data)
        deleteNode (num, nodeptr->right)
    else
        makeDeletion (nodeptr); // I will post it in Canvas
} //end func.
```

```
// Class IntBinaryTree
#ifndef INTBINARYTREE_H
#define INTBINARYTREE_H
class IntBinaryTree
{ private:  struct TreeNode
            {  int value;
               TreeNode *left;
               TreeNode *right;
            };
```



```
    TreeNode *root;
    // private member functions
    void insert (TreeNode *& , TreeNode *&); // prototype
    void destroySubTree ( TreeNode * ); // prototype
    void deleteNode (int, TreeNode *& ); // "
    void makeDeletion ( TreeNode *& ) ; // "
    void displayInOrder ( TreeNode * ) Const ; // "
    void displayPreOrder ( TreeNode *) Const ; // "
    void displayPostOrder ( "      " ) "  ; // "
public:
    IntBinaryTree ( ) // default Constructor
    { root = nullptr ; }
    // Destructor
    ~ IntBinaryTree ( )
    { destroySubTree (root); }
    // Binary Tree Operations
    void insertNode ( int );
    void searchNode ( int );
    void remove (int);
    void displayInOrder ( ) Const
    { displayInOrder (root); }
    void displayPreOrder ( ) Const
    { displayPreOrder (root); }
    void displayPostOrder ( ) Const
    { displayPostOrder (root); }
}; //end class
#endif
```

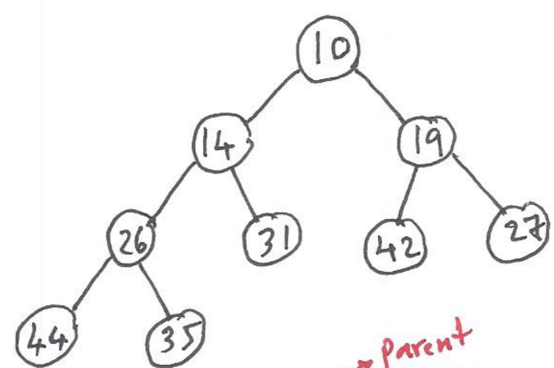
```
// Driver of IntBinaryTree
#include <iostream>
#include "IntBinaryTree.h"
using namespace std;

int main ( )
{   IntBinaryTree tree;
    tree.insertNode (5);
```

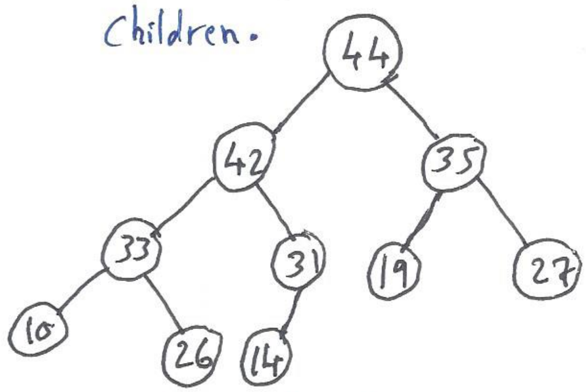
```
    tree.insertNode ( 3);
     "        "     (12);
    tree.displayInOrder ( );
    tree.remove (12);
      :      :      :
      return 0;
}
```

Heap Tree : It is a special case of balanced binary tree data structure, where the root-node value is compared with its children.

1- Min Heap Tree : $\boxed{\underbrace{P}_{\text{parent}}\text{-value} \leq \underbrace{C}_{\text{child}}\text{-value}}$, where the value of the root node is less than or equal to either of its children.



2- Max Heap Tree : $\boxed{\overbrace{P}^{\text{parent}}\text{-value} \geq \overbrace{C}^{\text{child}}\text{-value}}$, where the value of the root node is greater than or equal to the either of its children.



- Max Heap Tree Implementation Algorithm :

We insert one element at a time for Max. Heap tree. At any point, heap must maintain its property. While insertion, we also assume that we are inserting a Node in an already heapified tree.

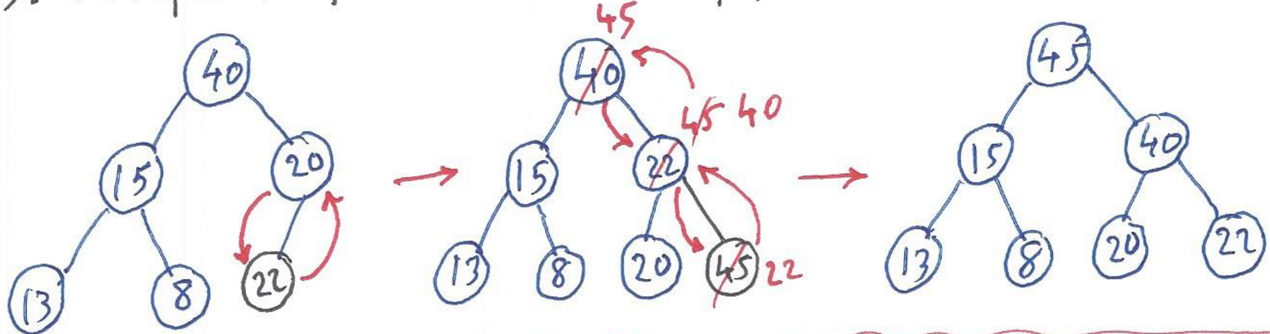
Step 1: Create a node at the end of heap

Step 2: Assign a new value to the node

Step 3: Compare the value of this child node with its parent.

Step 4: If the value of the parent node is less than child value, then swap them.

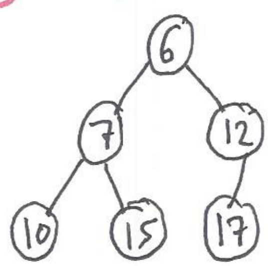
Step 5: We repeat step 3 and 4 until the heap property holds.



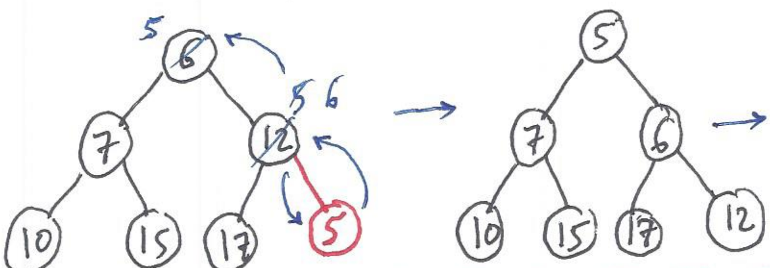
- We can implement heap as a tree or as an array.

For Tree : 1- Top to Bottom
2- left to right
3- We fill the tree

- **Array Implementation :** Min Heap Tree (Top to Bottom and left to right)



0	1	2	3	4	5	6	7
	6	7	12	10	15	17	

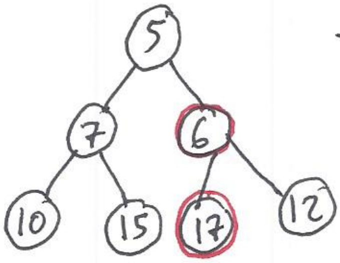


0	1	2	3	4	5	6	7
	5	7	6	10	15	17	12

- k th element of the array :
  - Its left child is located at $2*k$ index.
  - Its right child " " " $2*k+1$ index.
  - Its parent is located at $k/2$ index ($k/2$ is integer division)

0	1	2	3	4	5	6	7
	5	7	6	10	15	17	12

k=3

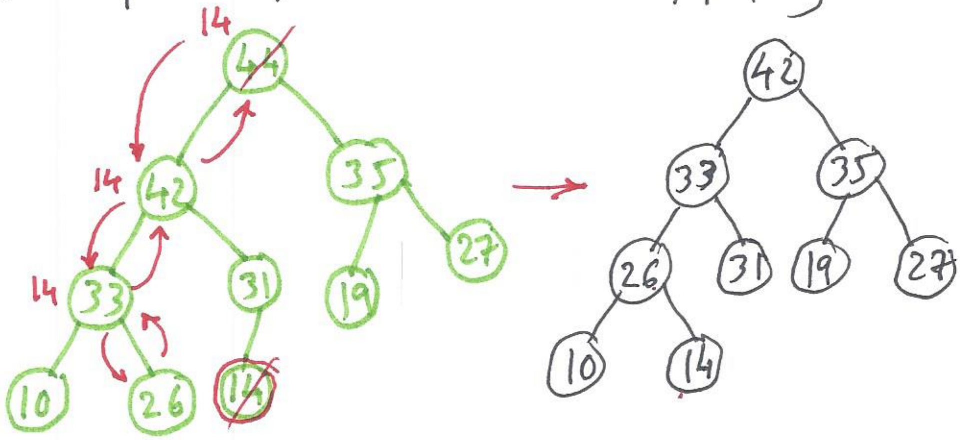


- Left child is the $2*3 = 6$th element ✓
- Right " " " $2*3+1 = 7$th element ✓
- parent $3/2 = 1$ → The 1st element is parent of 6 ✓

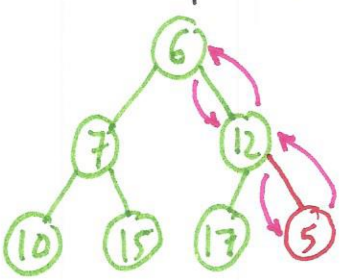
- Max Heap Tree Deletion Algorithm:

Deletion in Max (or Min) heap always happens at the root to remove Max (or Min) value. (Top to Bottom and left to right).

1- Remove root node
2- Move the last element of the last level to root.
3- Compare the value of this child's node with its parent.
4- If the value of the parent is less than child, then swap them.
5- Repeat steps 3 & 4 until the heap property holds.

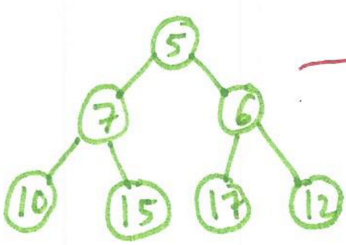


Insert: The new element is appended to the end of the heap (as the last element of the array). We should hold the Heap property at any Node.



0	1	2	3	4	5	6	7
	6	7	12	10	15	17	5

0	1	2	3	4	5	6	7
	6	7	5	10	15	17	12



→

0	1	2	3	4	5	6	7
	5	7	6	10	15	17	12

- Application of Heaps:

1- Heap Sort: Heap sort uses Binary Heap to sort an array in $O(\overbrace{n \log n}^{<n^2})$, time Complexity.

2- Priority Queue: Priority Queue can be efficiently implemented using Heap because it supports insert(), delete(), extractMax(), and decreasekey() Operations in $O(\log n)$ time Complexity.

3- Graph Algorithms: The priority queues are especially used in Graphs algs. like Dijkstra's shortest path and prim's Min Spanning Tree.

- Priority Queue: It is a special type of Queue in which each element is associated with a priority value. And elements are served on the basis of their priority. That is the higher priority elements are served first. If elements with the same prority, they are served according to their order in the queue.

Types of priority Queue: 1- Ascending priority Queue
2- Descending priority Queue

Ascending priority Queue: It gives the highest priority to the lower number in that Queue. For example, if we have: 4, 8, 12, 45, 35, 20, first we arrange them in ascending order. The list will be 4, 8, 12, 20, 35, 45
4 is the smallest number then priority Queue treats 4 as the highest priority.

4	8	12	20	35	45	...

has the highest priority and 45 has the lowest priority.

Descending priority Queue: We sort them in descending order:

45, 35, 20, 12, 8, 4

Here 45 has the highest priorty and 4 has the lowest priority.

The Implementation of Priority Queue in Data structure:

- Linked List
- Binary Heap
- Arrays
- BST (Binary Search Tree)

The Binary Heap is the most efficient to implement the priority Queue.

Hashing: It is a Searching technique. The time it takes for the Search is $O(1)$ which is a Constant time.

In array:
$$\overbrace{\boxed{5 \mid 10 \mid 2 \mid 30 \mid \;\; \mid \;\; \mid \cdots \mid 50}}^{n}$$

The Best case: $O(1)$
The worst case: $O(n)$

In a 2D array:
$$n \left\{ \overbrace{\begin{bmatrix} 5 & 7 & \cdots & 45 \\ 12 & 9 & \cdots & 100 \\ \vdots & \vdots & \ddots & \vdots \\ 71 & 92 & \cdots & 34 \end{bmatrix}}^{n} \right. \longrightarrow \text{The worst case } n \times n \longrightarrow O(n^2)$$

In hashing $O(1)$ is constant and it does not depend on n.
In hashing the data is organized with a help of a table which is called Hash table, denoted by HT and the hash table is stored in an array. To determine whether a particular item with a key, say x is in the table, we apply a func. h called the hash func., to the key x, that is we compute $h(x)$. The func. h is an arithmetic func. and $h(x)$ gives the address of the item in the hash table (HT). If the size HT is m, then $0 \le h(x) < m$ or (0 to $m-1$).
So to check an item in the table, we look at the entry $HT[h(x)]$ in the hash table. There is no particular order when we store indexes in the hash table (HT).

For example: Key $= 6, 17, 26, 12, 20, 27, \ldots$, $\underbrace{h(k_i)}$ $= k_i \% m$, $m = 10$

The values we want to store in HT.

Hash func. which give the index of the key in HT.

$k_i \% m \longrightarrow$ We are using division method. (m is the size of the array)

There are four methods to calculate the index:

1- Division 2- folding 3- Mid Square 4- Multiplication

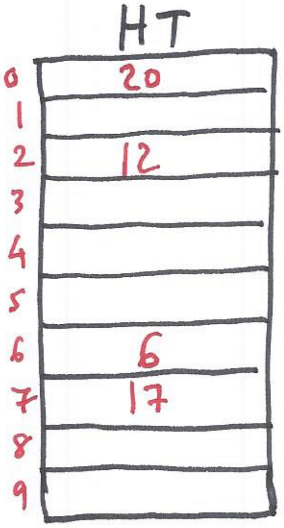
We continue with the above example using division method.

Key $= 6, 17, 26, 12, 20, 27, \ldots$ $h(k_i) = k_i \% m$, $m = \underline{10}$ size of array

$h(6) = 6 \% 10 = \underline{6}$ is location or index in HT

$h(17) = 17 \% 10 = \underline{7}$ " " " " "

$h(26) = 26 \% 10 = \underline{6} \longrightarrow$ Collision happens



$h(12) = 12 \% 10 = 2$ index in HT

$h(\underline{20}) = 20\%10 = \underline{0}$ index in HT

$h(27) = 27\%10 = \underline{7} \rightarrow$ Collision happens

To resolve Collision, We have two methods:

1- Open Hashing (or closed Addressing) → We use "chaining method" (Linked List)

2- Closed Hashing (or open Addressing)

a- Linear probing

b- Quadratic probing

c- double hashing.

---

ex #1a: Key Values: KV = 3, 2, 9, 6, 11, 13, 7, 12

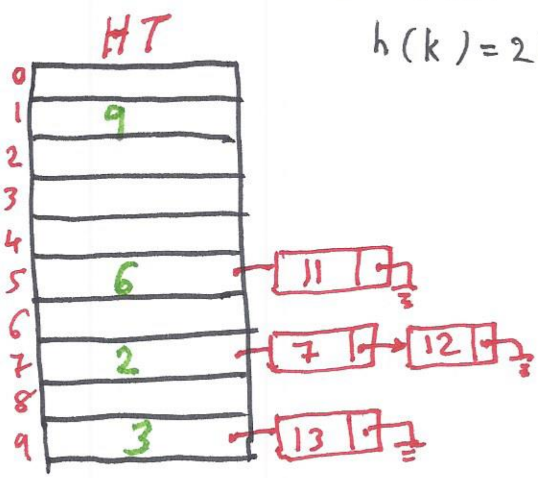
Hash func. → $H(k) = 2k+3$, $\boxed{m = 10}$ Size of HT

We use Division method (%) and open Hashing to store these values (KV).

We want to store these values in HT.

Division method: $h(k_i) = k_i \% m$

$h(k) = 2k+3 \longrightarrow 2k+3 \% m$



Key	Location H or index
3	$[2\times3+3]\%10 = 9$
2	$[2\times2+3]\%10 = 7$
9	$[2\times9+3]\%10 = 1$
6	$[2\times6+3]\%10 = 5$
11	$[2\times11+3]\%10 = 5$ Collision
13	$[2\times13+3]\%10 = 9$ Collision
7	$[7\times2+3]\%10 = 7$ Collision
12	$[12\times2+3]\%10 = 7$ Collision
