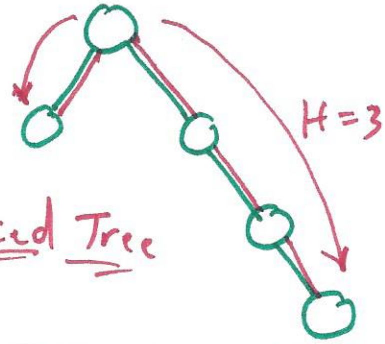


Not Balanced Tree

5-23-20 : Recursive Functions.

A recursive function is one that calls itself.

example:

```
void display()
{
  cout<<"Hello recursive."<<endl;
  display();
}
```

* There is no way to stop recursive calls. It is like an infinite loop.

```
void display(int n)   // 3
{ if(n>0) // base case
  { cout<<"Hello recursive."<<endl;
    display(n-1); // recursive call
  }
} // end func.
```

output:

display(3) first call → Hello recursive

display(2) → Hello recursive

display(1) → Hello recursive

display(0)

n=0 STOP

Factorial example: $0! = 1$

$1! = 1$

$5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$

$n! = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n$

$n! = (n-1)! \times n$

```
int fact (int n) // n is a positive integer
{   if (n <= 1)
          return 1 ; // base case
    else
          return n * fact(n-1); // recursive call
} // end fact
```

fact(4)

4 × fact(3)

4 × 3 × fact(2)

4 × 3 × 2 × fact(1)

4 × 3 × 2 × 1 × fact(0)

4 × 3 × 2 × 1 × 1 = 24

Iterative Implementation of factorial:

```
int fact(int n)   4
{ int i, f=1;
  for( i =1 ; i<=n ; i++)
       f=f*i;
  return f;   // 1×2×3×4
}
```

Which of the two functions is faster? Iterative Method

Any recursive function can be written as iterative function.

Every recursion should have the following characteristics:

1- A simple base case which we have a solution for and return value.

2- A way of getting our problem closer to the base case (simple problem).

3- A recursive call which passes the simpler problem back into the func.

*Recursion is like proof by mathematical induction:

- base case should be true
- If a statement is true for k, then we show it is true for k+1.

	0th	1	2	3rd	4th	5th	6th	....		
Fibonacci Sequence:	0	1	1	2	3	5	8	13	21	34 ...
							(n-2)	(n-1)	n	

```
int fib (int n)
{
    if(n<=1) // base case
        return n;
    else
        return fib(n-1)+ fib(n-2); //recursive call
} //end fib
```

fib(4)

fib(3) + fib(2)

fib(2) + fib(1) + fib(1) + fib(0)

fib(1) + fib(0) + 1 + 1 + 0

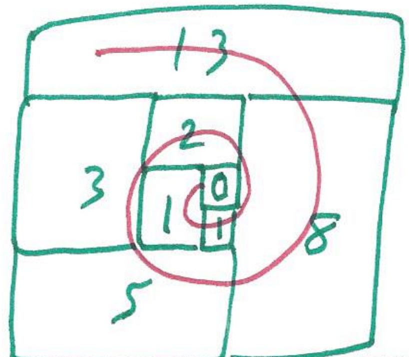
1 + 0 + 1 + 1 + 0 = 3

* Dividing two consecutive fib. numbers, eventually we get Golden Ratio. ($\varphi = 1.618034$

2 3 → 1,5
3 5 → 1,66
5 8 → 1,6
⋮ ⋮
233 377 → 1.618 → Golden Ration
⋮ ⋮

$$X_n = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}} \qquad \varphi = 1.618034$$

$$X_6 = \frac{(1.618034)^6 - (1-1.618034)^6}{\sqrt{5}} = \underline{\underline{8}}$$ So 6th fib. number is 8.



Recursive linked list operation:

```
int NumberList::countNodes(Node *nodeptr) const
{   if(nodeptr != NULL)
        return 1+ countNodes(nodeptr->next);
    else
        return 0;
}
```

To check if a number is prime. (greater than 1 and can be only divided by itself and 1).

```
bool isPrime(int p, int i)   2
{  if(p==i) return 1;
   if(p%i ==0) return 0;
   return isPrime(p, i+1);
} //end prime
```

Back to BST:

```
void IntBinaryTree::insert(TreeNode *& nodeptr, TreeNode *& newNode)
{ if(nodeptr ==NULL) // It is at the end of branch and insertion point
        nodeptr=newNode; // has been found.
                         // insert data
   else if(newNode->data < nodeptr->data)
           insert(nodeptr->left, newNode); // Search left
        else
           insert(nodeptr->right, newNode);
 } //end insert
```

***& nodeptr**: nodeptr is a reference to a pointer to a TreeNode structure. This means that any action performed on nodeptr is actually performed on the argument that was passed into nodeptr.

Searching a Tree for a number:

```
bool IntBinaryTree::SearchNode(int num)
{
  TreeNode *nodeptr=root;
  while(nodeptr)
  {  if(nodeptr->data ==num)
           return true;
     else if(num < nodeptr->data)
               nodeptr = nodeptr->left;
          else
               nodeptr =nodeptr->right;
  } //end while
} //end func.
```

```
// The displayInOrder member func.
void IntBinaryTree :: displayInOrder (TreeNode *nodeptr) Const
{   if (nodeptr)                          NULL = nullptr
    {   displayInOrder( nodeptr->left); // recursive call
        cout << nodeptr->data <<endl;
        displayInOrder(nodeptr->right); // recursive call
    }
}
```

```
// The displayPreOrder func., root-left-right      left | data | right
void IntBinaryTree :: displayPreOrder (TreeNode *nodeptr) Const
{   if(nodeptr)
    {     cout << nodeptr->data <<endl;
          displayPreOrder (nodeptr->left);
             "        "     (  "    ->right);
    }
} //end
```

```
// The displayPostOrder func. left-right-root
void IntBinaryTree :: displayPostOrder (TreeNode *nodeptr) Const
{   if (nodeptr)
    {   displayPostOrder (nodeptr->left);
           "       "     (  "    "  right);
        cout<<nodeptr-data <<endl;
    }
} //end
```

```
// Destructor
~IntBinaryTree ()
{
    destroySubTree (root);
}
// destroySubTree is called by the Destructor
// It deletes all the nodes in the tree
void IntBinaryTree::destroySubTree (TreeNode *nodeptr)
{   if (nodeptr)
    {      if (nodeptr->left)
                  destroySubTree(nodeptr->left);
           if(nodeptr->right)
                  destroySubTree (nodeptr->right)
           delete nodeptr; //    nodeptr -> | L | data | R |
    }
} //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;
      "      " right = NULL;
   insert (root, newNode);
}
```

```
#include <iostream>
#include "IntBinaryTree.h"
using . . . . ;
int main()
{
    IntBinaryTree   tree;
    tree.insertNode (5);
     "  .  "        (3);
     "  .  "        (12);
    tree.displayInOrder (root);
    tree.remove(12);
     :     :     :
    retur 0;
}
```

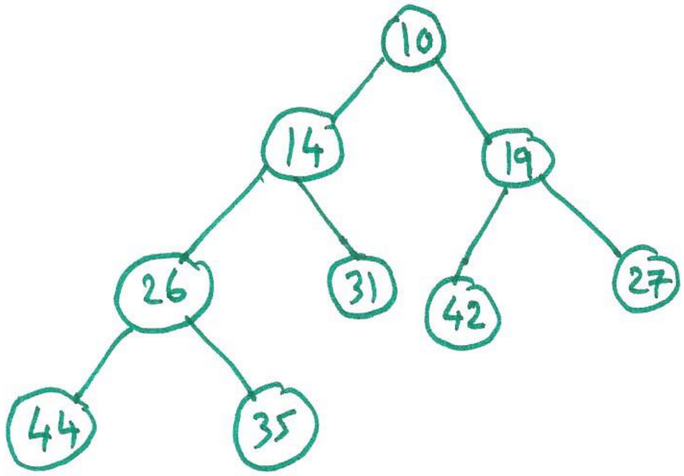
Deleting a Node :

```
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 in Canvas.
}
```

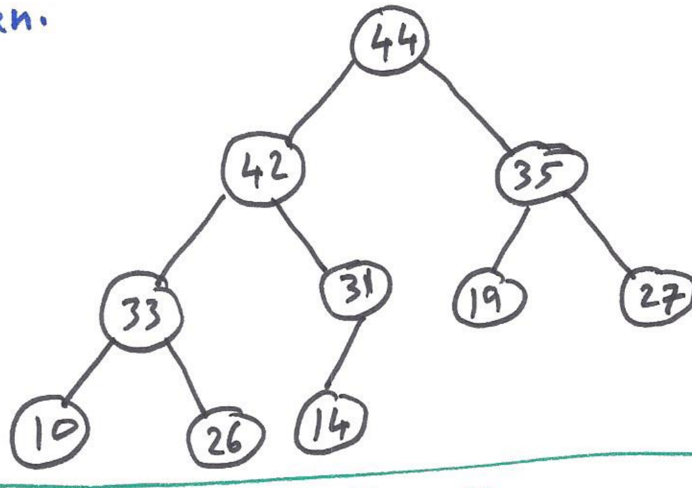
Heap Tree: It is a special case of balanced binary tree data structure, where the root-node value is compared with its children.

1- **Min Heap Tree:** P-value $\leq$ C-value, where the value of the root node is less than or equal to either of its children.

parent → P, child → C



2- **Max. Heap Tree:** P-value $\geq$ C-value
Where the value of the root node is greater than or equal to either of its children.



Max. Heap Implementation Algorithm:

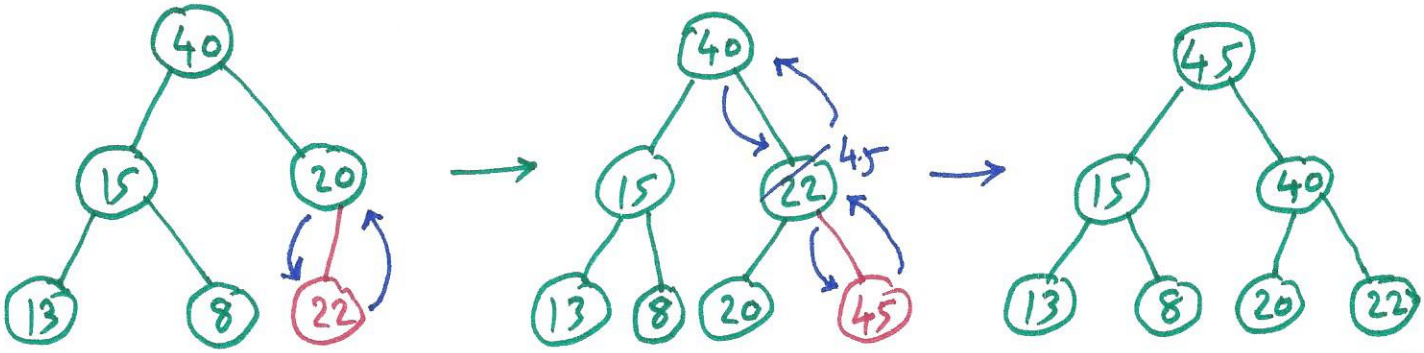
We insert one element at a time for Max. Heap. At any point, heap must maintain its property. While insertion, we also assume that we are inserting a node in an already heapified tree.

1- Create a node at the end of heap.

2- Assign new value to the node.

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

4- If the value of the parent is less than Child value, then swap them.
5- We repeat step 3 & 4 until the heap property holds.

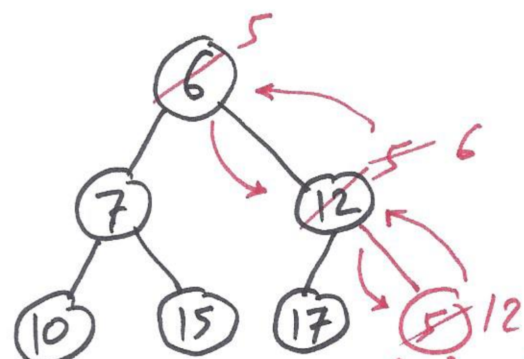


We can implement heap as a tree or as an array.

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

Array Implementation :

Min Heap

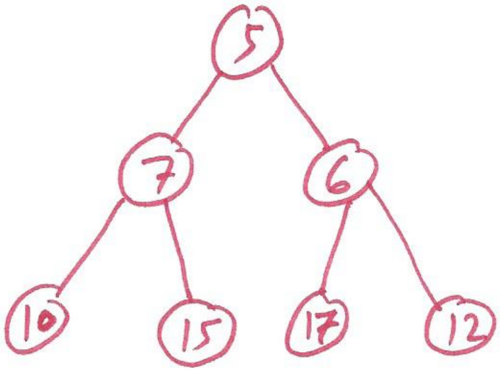


0	1	2	3	4	5	6	7
	6	7	12	10	15	17	

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

* The root is the second item in the array. We skip the index 0, for Convenience implementation.

* Kth element of the array :
  - Its left child is located at $2*k$ index
  - " right " " " " $2*k+1$ index
  - " parent is " " $K/2$ index



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

K=3

- left child is the 6th element
- right " " " 2(3)+1 = 7th element
- parent 3/2 = 1 → 1st element is parent of 6

Max Heap Tree Deletion Algorithm:

Deletion in Max (or Min) Heap always happens at the root to remove the Max (or Min) value.

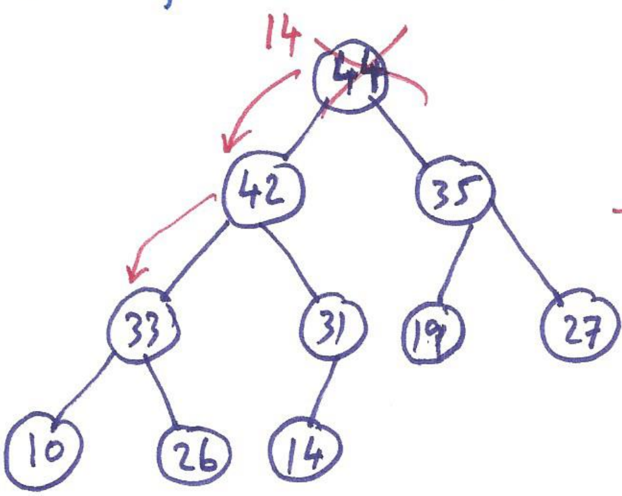
1- Remove root node

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 parent is less than child, then swap them.

5- Repeat Step 3 & 4 until Heap property Holds.



→