Week 14, Recursive Functions:

A recursive function is one that calls itself.

ex#1,

```
void display()
{ Cout << "Hello Recursion!" << endl;
  display();
}
```

⇒ output:

Hello Recursion!
Hello Recursion!
⋮ ⋮
Infinite recursive calls.

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

ex#2,

```
void display (int n)   → 3/ 2/ 1/ 0
{ if (n>0) // base case
  { Cout << "Hello Recursion!" << endl;
    display (n-1); // recursive call
  }
} // end func.
```

n = 3

Hello Recursion!
Hello Recursion!
Hello Recursion!

Factorial: $0! = 1$, $1! = 1$, $3! = 1 \times 2 \times 3$

$$n! = \overbrace{1 \times 2 \times 3 \times \cdots \times (n-1)}^{(n-1)!} \times n$$

$$\boxed{n! = n \times (n-1)!}$$

```
int fact (int n)   → 4/ 3/ 2/ 1
{ if (n <= 1)   // base case
      return 1;
  else
      return n * fact(n-1); // recursive
                            // call
} // end fact
```

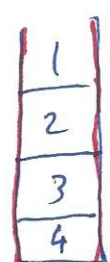
⇒ output: n = 4

4 * fact(3)

4 * 3 * fact(2)

4 * 3 * 2 * fact(1)

4 * 3 * 2 * 1 = 24



Stack

Any recursive function can be written in Iterative Implementation Code.

- Iterative Implementation of factorial:

```
int fact (int n)   → 4
{ int i, f=1;
  for( i=1 ; i<=n ; i++)
      f = f*i;
  return f;
} // end func.
```

⇐

$f = 1 \times 1$

$f = 1 \times 2$

$f = 1 \times 2 \times 3$

$f = 1 \times 2 \times 3 \times 4$

$f = 24$

Which one of the two methods is faster? Iterative method is faster than recursive method.

In recursive method we may have call stack and this slows down the process.

Every recursion should have the following characteristics:

1- A simple base case which we have a solution for it 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 function.

- Recursion is like proof by induction (Mathematical Induction).

For example, prove that sum of n positive integers is equal to $\frac{n(n+1)}{2}$.

$$P(n) \rightarrow 1+2+3+\cdots+n = \frac{n(n+1)}{2}$$

Base Case: $n=1 \rightarrow 1 = \frac{1(1+1)}{2} \rightarrow 1=1$ ✓

We assume that $P(K)$ is true then we prove $P(K+1)$ is true.

$$P(K) = 1+2+3+\cdots+K = \frac{K(K+1)}{2}$$

$$P(K+1) = \underbrace{1+2+3+\cdots+K}+K+1 = \frac{(K+1)(K+2)}{2}$$ → inductive case

$$\frac{K(K+1)}{2} + \frac{2\cdot(K+1)}{2\times 1} = \frac{K(K+1)+2(K+1)}{2} = \frac{(K+1)(K+2)}{2}$$ ✓

Fibonacci Sequence:

Son of bonacci is Italian Math. He changed the numeric system from I, II, III, IV, ... to 1, 2, 3, 4, ...

0th	1st	2nd	3rd	4th	5th	6th	7th	...		
0	1	1	2	3	5	8	13	21	34	....
					n-2	n-1	n			

$$\boxed{fib(n) = fib(n-2) + fib(n-1)}$$

```
int fib(int n)   →4
{  if (n<=1)  // base case
        return n;
   else
        return fib(n-2)+fib(n-1); // recursive call
} //end fib
```

If $n=4$, What is the 4th fib number?

$$fib(4)$$

$$fib(2) + fib(3)$$

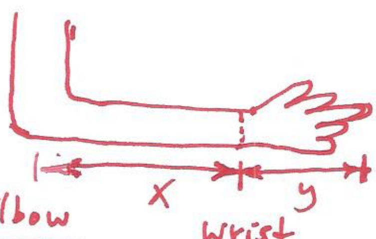
$$fib(0) + fib(1) + fib(1) + fib(2)$$

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

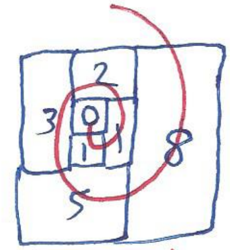
$$0 + 1 + 1 + 0 + 1 = 3 \longrightarrow \text{4th fib. number is } 3$$

- Dividing two Consecutive fib. numbers, eventually we get Golden Ratio.

A	B	
2	3	$\rightarrow B/A = \frac{3}{2} = 1.5$
3	5	$\rightarrow 5/3 = 1.66$
5	8	$\rightarrow 8/5 = 1.6$
⋮	⋮	
233	377	$\rightarrow \frac{377}{233} = 1.618\ldots \rightarrow$ Golden Ration (φ)
⋮	⋮	⋮



$\frac{x}{y} = 1.618\ldots$



"Snail"

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

X_n → nth fib

Golden Ratio $= \varphi = 1.618\ldots$

Find the 6th fib number.
$n = 6$

$$X_6 = \frac{(1.618\ldots)^6 - (1 - 1.618\ldots)^6}{\sqrt{5}} = 8$$

6th fib. number is 8

Recursive Linked List operation:

```
int NumberList:: CountNodes (Node *nptr) Const
{   if (nptr != NULL)
            return 1+ CountNodes (nptr->next);
    else
        return 0;
}
```

example : To check if a number is prime.

* prime number is greater than 1 and can be only divided by itself or 1.

```
bool isPrime ( int P, int i ) → 2
{  if ( P==i ) return 1 ; // base Case
   if ( P%i ==0 ) return 0;
       return isPrime ( P, i+1 )
} //end func.
```

Class Activity 1: Write a recursive function that return sum of the n positive integer numbers from 1 + n (inclusive). e.g n = 5 → 1+2+3+4+5 = 15

```
int Sum ( int n )
{  if ( n==1 )
        return 1;
   else return n + Sum(n-1);
} //end func.
```

Class Activity : Compute the sum of all digits of a given positive number. (recursively)
e.g 7826 → 7+8+2+6 = 23

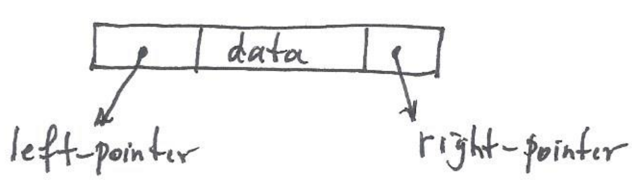
```
int totalDigits ( int n )
{  if ( n==0 )
        return 0;
   else return n%10 + totalDigits( n/10 );
}
```

- Recursive functions have less codes and makes debugging easier.

Back to BST :

```
struct TreeNode
{  int data;
   TreeNode *left;
   TreeNode *right;
};
```

Node :



```
Void IntBinaryTree :: insert ( TreeNode *& nodeptr , TreeNode *& newNode )
{ if ( nodeptr == NULL ) // It is at the end of branch and insertion point has
                          // been found.
        nodeptr = newNode ; // insert data
    else if ( newNode -> data < nodeptr -> data )
                insert ( nodeptr -> left , newNode ) ; // Search left
        else
                insert ( nodeptr -> right , newNode ) ; // Search right
} // 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 ) // nodeptr is pointing to a Node
   {   if ( nodeptr -> data == num )
                return true;
       else if ( num < nodeptr -> data )
                nodeptr = nodeptr -> left;
            else
                nodeptr = nodeptr -> right;
   } //end while
} //end func
```

// To displayInOrder member function , Left - root - Right

```
Void IntBinaryTree :: displayInOrder ( TreeNode *nodeptr ) Const
{     if ( nodeptr )
      {     displayInOrder ( nodeptr -> left ) ; // recursive Call
            Cout << nodeptr -> data << endl; // root
```

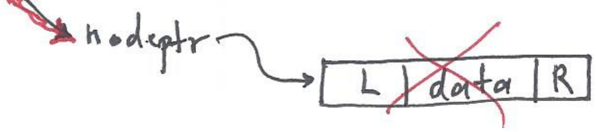
```
        displayInOrder (nodeptr->right); // recursive call
    }
} // end func.
```

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

```
// The displayPostOrder func.  , Left-Right-Root
void IntBinaryTree :: displayPostOrder ( TreeNode *nodeptr) Const
{   if(nodeptr)
    {   displayPostOrder(nodeptr->left);
        displayPostOrder (nodeptr->right);
        Cout << nodeptr->data <<endl; // rood
    } // end if
} // end func.
```

```
// Destructor
~IntBinaryTree ( )
{   destroySubTree (root);
}
// destroySubTree is called by destructor, it deletes all 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;
```

```
    } //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.
```