

Data Structures & Algorithm (CS-201)

1. **INSTRUCTOR:**

Shahid Iqbal Lone

e_mail: loneshahid@yahoo.com

2. **COURSE BOOK:**

Tanenbaum Aaron M, Langsam Yedidyah, Augenstein J Moshe, *Data Structures using C*.

3. **LIST OF REFERENCE MATERIAL:**

1. Tremblay J.P and Sorenson P.G, *An introduction to data structures with applications*, Tata McGraw Hill, 2nd Edition.
2. Gilberg, F Richard & Forouzan, A Behrouz, *Data Structures A Pseudocode approach with C*, Thomson Brooks/Cole Publications, 1998.
3. Seymour Lipschutz, *Theory and Problems of data Structures*, Schaum's Series, Tata McGraw Hill, 2004.

4. **OBJECTIVES:**

With a dynamic learn-by-doing focus, this document encourages students to explore data structures by implementing them, a process through which students discover how data structures work and how they can be applied. Providing a framework that offers feedback and support, this text challenges students to exercise their creativity in both programming and analysis. Each laboratory work creates an excellent hands-on learning opportunity for students. Students will be expected to write C-language programs, ranging from very short programs to more elaborate systems. Since one of the goals of this course is to teach how to write large, reliable programs. We will be emphasizing the development of clear, modular programs that are easy to read, debug, verify, analyze, and modify.

5. **PRE-REQUISITE:**

A good knowledge of c-language, use of Function and structures.

Data:

Data are simply values or set of values. A data item refers to a single unit of values.

Data items that are divided into sub items are group items; those that are not are called elementary items. For example, an employee's name may be divided into three sub items – first name, middle name and last name- but the social security number would normally be treated as a single item.

An entity is something that has certain attributes or properties which may be assigned values. The values themselves may be either numeric or non-numeric.

Example:

Attributes:	Name	Age	sex	Social Society number
Values:	Hamza	20	M	134-24-5533
	Ali Rizwan	23	M	234-9988775
	Fatima	20	F	345-7766443

Entities with similar attributes (e.g. all the employees in an organization) form an entity set. Each attribute of an entity set has a range of values, the set of all possible values that could be assigned to the particular attribute.

The term "*information*" is sometimes used for data with given attributes, of, in other words meaningful or processed data.

A field is a single elementary unit of information representing an attribute of an entity, a record is the collection of field values of a given entity and a file is the collection of records of the entities in a given entity set.

Data Structure:

Data may be organized in many different ways; the logical or mathematical model of a particular organization of data is called a **data structure**. The choice of a particular data model depends on the two considerations first; it must be rich enough in structure to mirror the actual relationships of the data in the real world. On the other hand, the structure should be simple enough that one can effectively process the data when necessary.

Categories of Data Structure:

The data structure can be classified in to major types:

- ❖ Linear Data Structure
- ❖ Non-linear Data Structure

1. Linear Data Structure:

A data structure is said to be linear if its elements form a sequence or in other word a linear list.

These are basically two ways of representing such linear structure in memory.

a) *One way is to have the linear relationships between the elements represented by means of sequential memory location.*

These linear structures are called **arrays**.

b) *The other way is to have the linear relationship between the elements represented by means of pointers or links.*

These linear structures are called **linked lists**.

The common examples of linear data structure are

- **Arrays**
- **Queues**
- **Stacks**
- **Linked lists**

2. Non-linear Data Structure:

This structure is mainly used to represent data containing a hierarchical relationship between elements.

e.g. graphs, family **trees** and table of contents.

Arrays:

The simplest type of data structure is a linear (or one dimensional) array. A list of a finite number n of similar data referenced respectively by a set of n consecutive numbers, usually 1, 2, 3 n . if we choose the name **A** for the array, then the elements of A are denoted by subscript notation

$a_1, a_2, a_3 \dots a_n$

or by the parenthesis notation

A (1), A (2), A (3) A (n)

or by the bracket notation

A [1], A [2], A [3] A [n]

Example:

A linear array **A[8]** consisting of numbers is pictured in following figure.



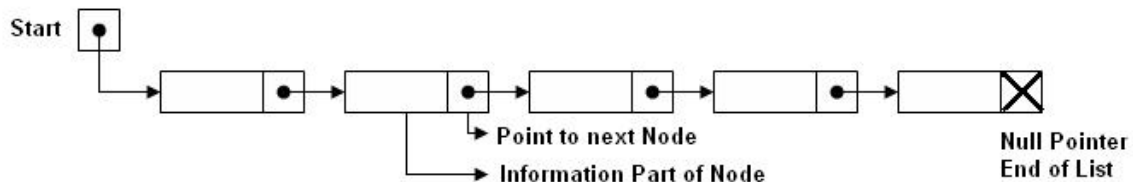
Linked List:

A linked list, or one way list is a linear collection of data elements, called **nodes**, where the linear order is given by means of pointers. Each **node** is divided into two parts:

- The first part contains the **information** of the element/node
- The second part contains the address of the next node (**link /next pointer field**) in the list.

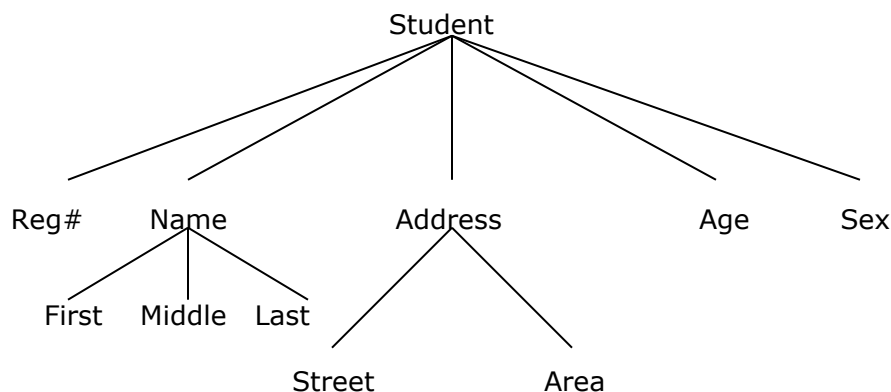
There is a special pointer **Start/List** contains the address of first node in the list. If this special pointer contains null, means that List is empty.

Example:



Tree:

Data frequently contain a hierarchical relationship between various elements. The data structure which reflects this relationship is called a **rooted tree graph** or, simply, a **tree**.



Graph:

Data sometimes contains a relationship between pairs of elements which is not necessarily hierarchical in nature, e.g. an airline flights only between the cities connected by lines. This data structure is called **Graph**.

Queue:

A queue, also called **FIFO** system, is a linear list in which deletions can take place only at one end of the list, the **Front** of the list and insertion can take place only at the other end **Rear**.

Stack:

It is an ordered group of homogeneous items of elements. Elements are added to and removed from the **top** of the stack (the most recently added items are at the top of the stack). The last element to be added is the first to be removed (**LIFO**: Last In, First Out).

Data Structure Operator:

The data appearing in our data structures are processed by means of certain operations. In fact, the particular data structure that one chooses for a given situation depends largely in the frequency with which specific operations are performed.

The following four operations play a major role in this text:

- ❖ **Traversing**: accessing each record/node exactly once so that certain items in the record may be processed. (This accessing and processing is sometimes called "**visiting**" the record.)
- ❖ **Searching**: Finding the location of the desired node with a given key value, or finding the locations of all such nodes which satisfy one or more conditions.
- ❖ **Inserting**: Adding a new node/record to the structure.
- ❖ **Deleting**: Removing a node/record from the structure.

ARRAYS

INTRODUCTION

An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier. Arrays are preferred for situations which require similar type of data items to be stored together. It is a **finite** collection of **similar** elements stored in adjacent memory locations. By **finite** we means that there are specific number of elements in an array and **similar** implies that all the elements in an array are of the same type. For example, an array may contain all integers or all float values.

Thus an array is collection of variables of the same type that are referred by a common name. The elements of the array are referenced respectively by an **index set** containing **n** consecutive numbers. An array with **n** number of elements is referenced using an index that ranges from **0** to **n-1**. The lowest index of an array is called its **lower bound** and highest index is called the **upper bound**. The number of elements in an array is called its **range**. The elements of an array **A[n]** containing **n** elements are referenced as **A[0]**, **A[1]**, **A[2]**, **A[3]**, **A[n-1]** where **0** is the **lower bound** and **n-1** is the **upper bound** of the array.

For Example: **int A[10] = { 5, 6, 12, 4, 15, 45, 87, 1, 9, 13 };**

Where each element is 4 bytes long.

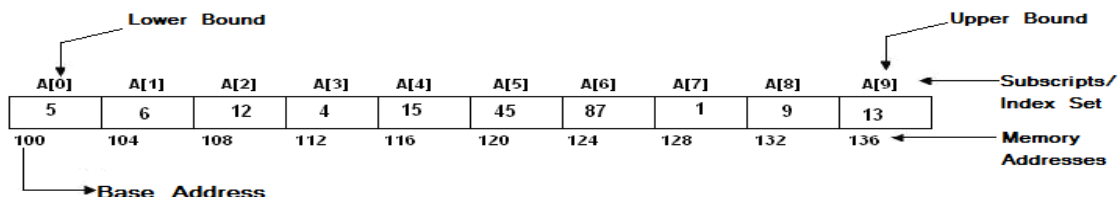


Figure : 3-1

Representation of (Single Dimension) Linear Array in Memory

The memory of computer is simply a sequence of addressed locations. Let **A** be a Linear array stored in the memory of computer as shown in Figure: 3-1. As previously noted, the elements of **A** are stored in successive memory cells. Accordingly the computer does not need to keep track of the address of every element of **A**, but needs to keep track only first element of the array i.e. **A[0]**, which is also called **Base Address** of the array **A** denoted by:

Base (**A**) is called Base Address of **A**

Using this Base Address **Base (A)**, computer calculates the address of any element of the array by using following formula:

$$\text{Add (A[k])} = \text{Base (A)} + w (K - \text{Lower bound})$$

Where **w** is the size, (number of bytes) of each cell (depends on data type of the array) and **K** is the **index number/subscript** of the element of **A**.

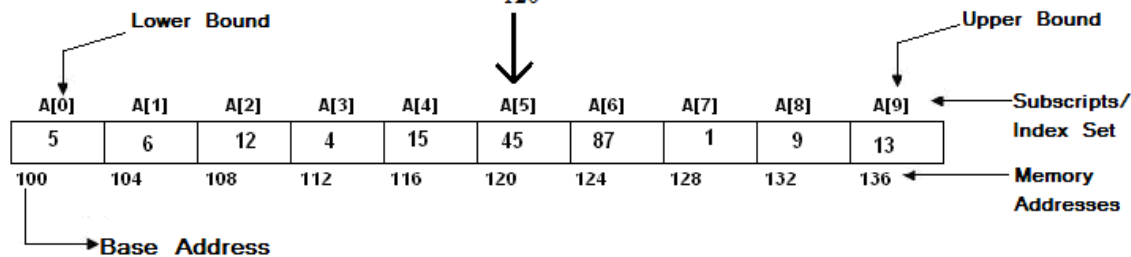
For example:

Find address of A[5], Assume that the base address of this array = 100,

$$w = 4 \quad \text{and} \quad K = 5$$

$$\text{Add (A[5])} = 100 + 4 (5 - 0)$$

$$= 120$$



Declaration of the Arrays: Any Linear array declaration contains:

Data type, **array name** and the **array size**.

Examples:

int a[20], b[3], c[7];

float f[5], c[2];

char m[4], n[20];

Initialization of an array is the process of assigning initial values. Typically declaration and initialization are combined.

Examples:

float, b[3]={2.0, 5.5, 3.14}; char name[4]= {'E','m','r','e'}; int c[10]={0};

Two-Dimensional Array

A two-dimensional array is a collection of elements placed in M rows and N columns denoted as $M \times N$ elements such that each element is specified by a pair of integers (such as j, k), called subscripts, where j represents to row and k represents to column of the element. The two-dimensional array is also called *matrix*. Mathematically, a matrix \mathbf{A} is a collection of elements $\mathbf{A}_{j,k}$ with the property that

$$0 \leq j < M \quad \text{and} \quad 0 \leq k < N$$

A matrix is said to be of order $M \times N$, Where M is total number of Rows and N is total numbers of columns in the two dimensional array / matrix. Matrix can be conveniently represented by two-dimensional array. The various operations that can be performed on a matrix are: addition, multiplication, transposition, and finding the determinant of the matrix.

An example of 2D array can be `int A[2][3]` containing 2 rows and 3 columns (Here $M = 2$ and $N = 3$) and `A[0][1]` is an element placed at 0th row and 1st column in the array.

A two-dimensional array can thus be represented as given in following Fig:

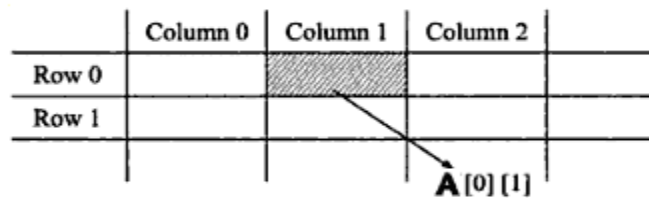


Fig. Representation of 2-D array

Representation of (Two-Dimensional) Array in Memory

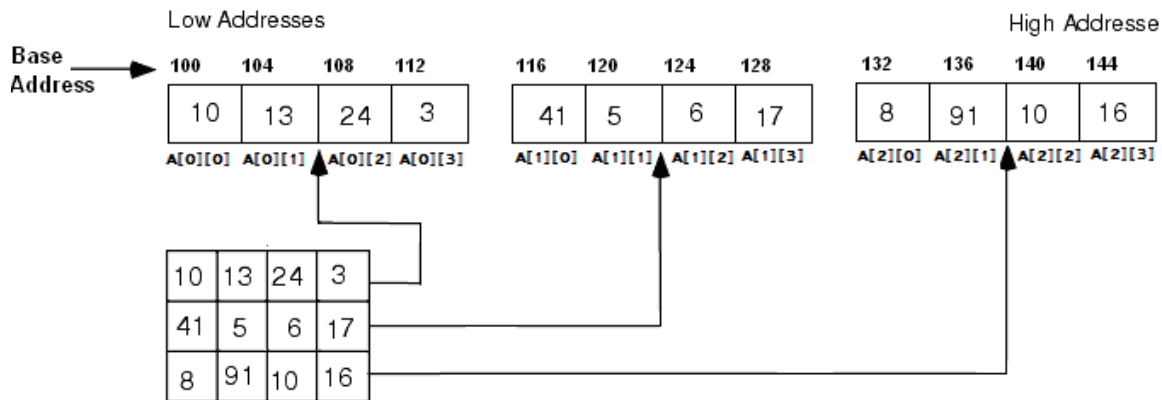
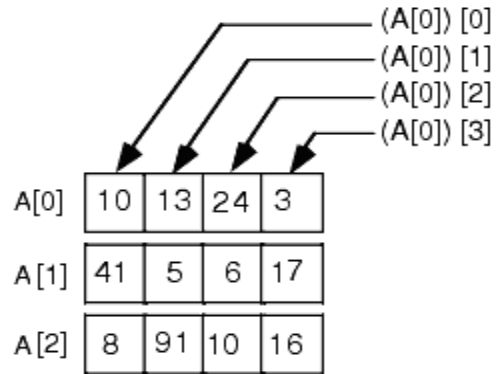
Let \mathbf{A} be a two-dimensional $m \times n$ array. Although \mathbf{A} is pictured as a rectangular array of elements with m rows and n columns, the array will be represented in memory by a block of $m.n$ sequential memory locations. Matrices are also stored in contiguous/sequential memory locations. Specially, the programming language will store the array \mathbf{A} in memory either:

- Row-Major-Order.
- Column-Major-Order

In row-major-order, elements of a matrix are stored on row-by-row basis, that is, all the elements in first row are stored sequentially first, then the second row will be stored in successive locations and so on. On the other hand, in column-major-order, all the elements in first columns are stored sequentially first, then the second columns will be stored in successive locations and so on. For example consider the following matrix:

Row-Major-Order:

```
int A[3][4] = { 10, 13, 24, 3,
                41, 5, 6, 17,
                8, 91, 10, 16};
```



Now we find the address of a particular element in Two-Dimensional array stored as Row-Major-Order using the following formula.

$$\mathbf{A[j][k]} = \text{Base}(\mathbf{A}) + \mathbf{W} [\mathbf{N} (\mathbf{j} - \text{Row_LowBound}) + (\mathbf{k} - \text{Col_LowBound})]$$

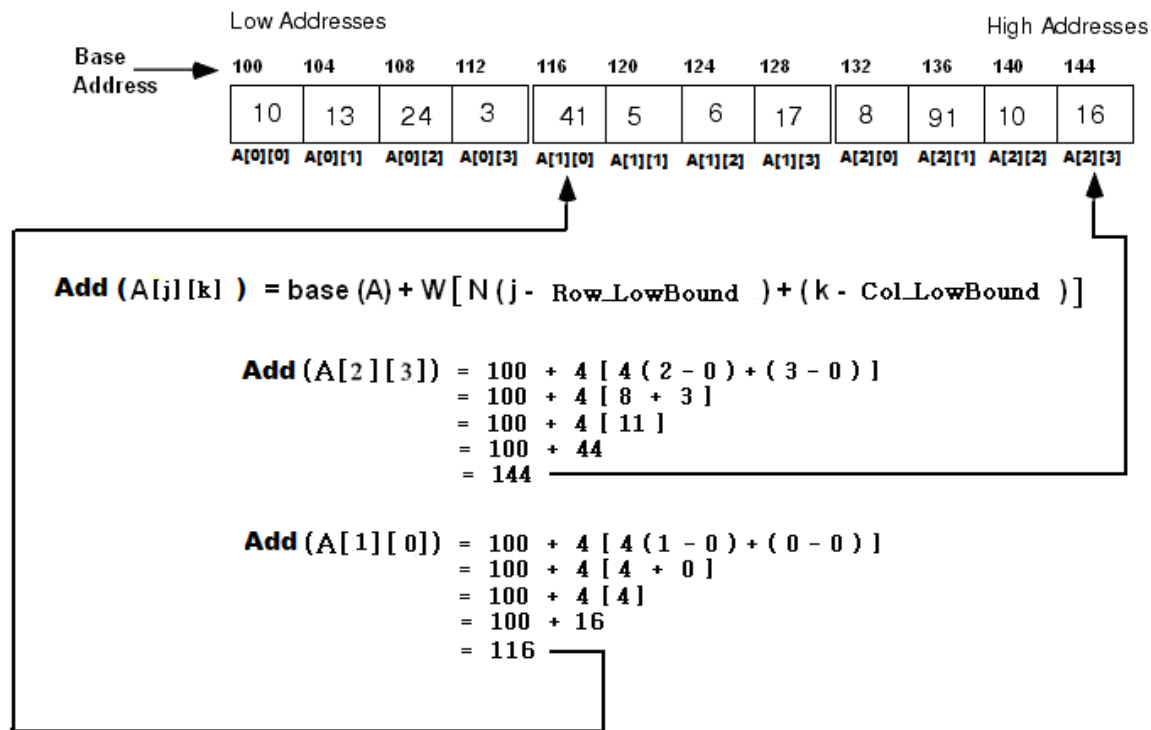
Where \mathbf{W} = Size of element (Number of bytes in each element)
 \mathbf{N} = Total Number of columns in the matrix.

Consider that following two-dimensional array is stored in computer's memory:

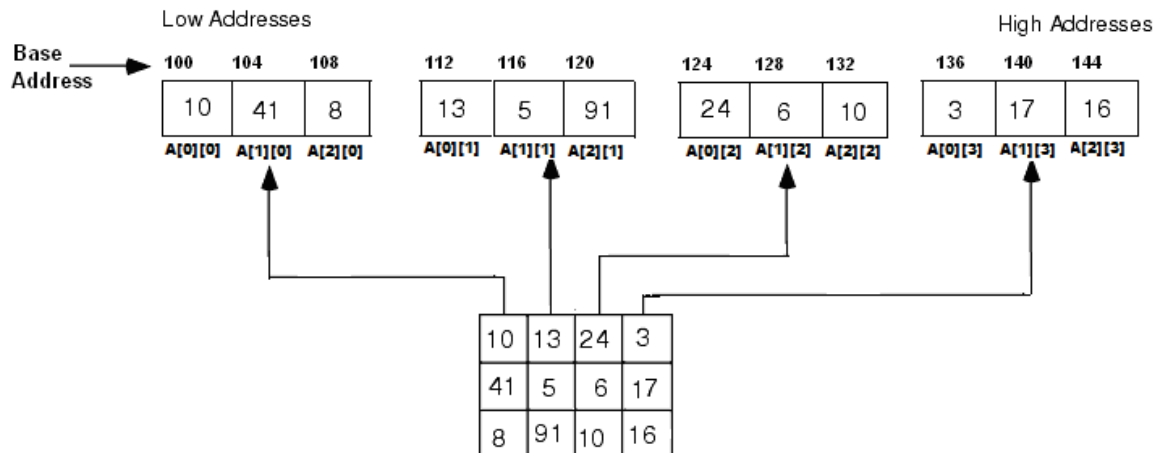
```
int A[3][4] = { 10, 13, 24, 3, 41, 5, 6, 17, 8, 91, 10, 16 };
```

Here $\mathbf{M} = 3$ and $\mathbf{N} = 4$

Let us assume that the base address of the array matrix is 100. Since $\mathbf{W} = 4$ (as array of integer type whose cell size is 4 bytes long). Therefore according to the formula, addresses of $\mathbf{A(2, 3)^{th}}$ and $\mathbf{A(1, 0)^{th}}$ elements in the array matrix will be:



3.3.3 Column-Major-Order:



Now we find the address of a particular element in Two-Dimensional array stored as Column-Major-Order using the following formula.

$$A[j][k] = \text{Base}(A) + W [M (k - \text{Col_LowBound}) + (j - \text{Row_LowBound})]$$

Where W = Size of element (Number of bytes in each element)

M = Total Number of Rows in the matrix.

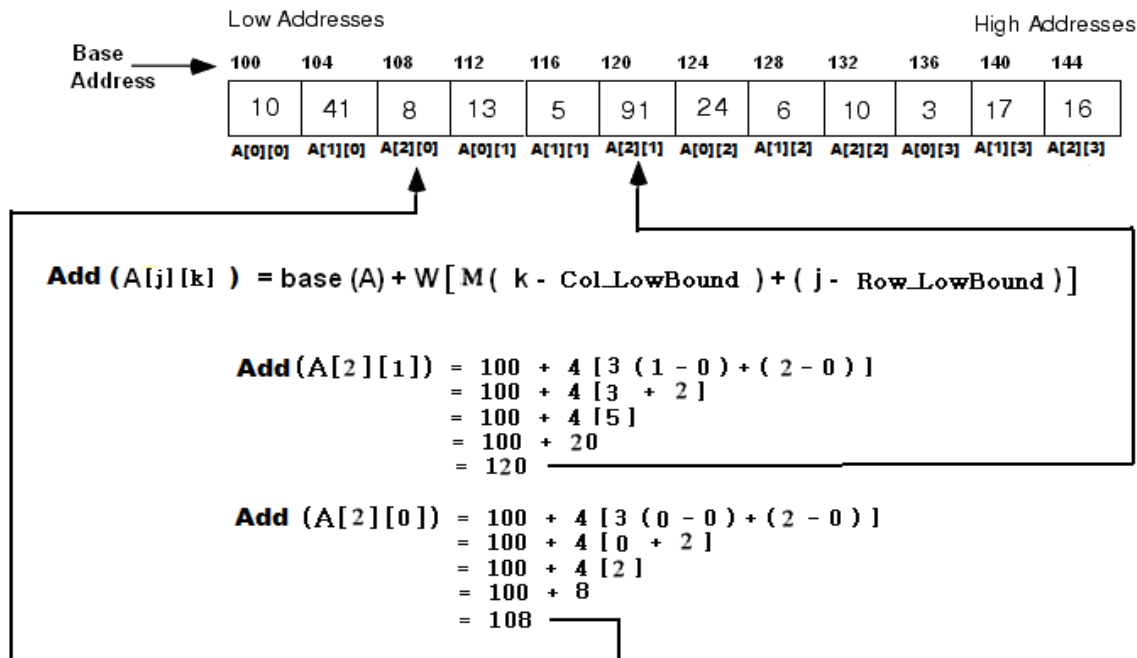
Consider that following two-dimensional array is stored in computers memory:

int $A[3][4] = \{ 10, 13, 24, 3, 41, 5, 6, 17, 8, 91, 10, 16 \};$

Here $M = 3$, $N = 4$ and $W = 4$

DATA STRUCTURES

Let us assume that the base address of the array matrix is 100. Therefore according to the formula, addresses of $A(2, 1)^{th}$ and $A(2, 0)^{th}$ elements in the array matrix will be:



Operations on array

- 1- **Traversing:** means to visit all the elements of the array in an operation is called traversing.
- 2- **Insertion:** means to put values into an array
- 3- **Deletion / Remove:** to delete a value from an array.
- 4- **Sorting:** Re-arrangement of values in an array in a specific order is called sorting.
- 5- **Searching:** The process of finding a particular element of an array is called searching.

There two popular searching techniques:

Linear search and binary search and will be discussed later.

1. Traversing in Linear Array:

It means processing or visiting each element in the list exactly once;

Let 'A' is an array stored in the computer memory. If we want to display the contents of 'A', it has to be traversed i.e. by accessing and processing each element of 'A' exactly once.

Algorithm: (Traverse a Linear Array) Here LA is a Linear array with lower boundary LB and upper boundary UB. This algorithm traverses LA applying an operation Process to each element of LA.

1. [\[Initialize counter.\]](#) Set $K=LB$.
2. Repeat Steps 3 and 4 while $K \leq UB$.
3. [\[Visit element.\]](#) Apply PROCESS to $LA[K]$.
4. [\[Increase counter.\]](#) Set $k=K+1$.
- [\[End of Step 2 loop.\]](#)
5. Exit.

The alternate algorithm for traversing is

Algorithm: (Traverse a Linear Array) This algorithm traverse a linear array LA with lower bound Lb and upper bound UB.

1. Repeat for $K=LB$ to UB
 Apply PROCESS to $LA[K]$.
[\[End of loop.\]](#)
2. Exit.

This program will traverse each element of the array to calculate the sum and then calculate & print the average of the following array of integers.

(4, 3, 7, -1, 7, 2, 0, 4, 2, 13)

```
#include<iostream.h>
```

```
#define size 10
```

```
int main()
```

```
{ int x[10]={4,3,7,-1,7,2,0,4,2,13}, i, sum=0, LB=0, UB=size;
```

```
  float av;
```

```
  for(i=LB; i<UB; i++) sum = sum + x[i];
```

```
  av = (float)sum/size;
```

```
  cout<< "The average of the numbers= "<< av<<endl;  return 0;
```

```
}
```

2. Insert an Item into Linear Array:

Inserting refers to the addition of a new element in an array, say '**A**'. Inserting an element at the end of the array can be easily done. On the other hand, if an element is to be inserted at the beginning or in the middle of array '**A**', then the elements (from insertions point inclusively) must be moved downward to new locations, to accommodate the new element. Assume, values inside the '**A**' are in sorted order and it is needed to insert the new item at such location in '**A**' so that the sorted order of '**A**' should not be disturbed. See the algorithm:

The following set of algorithms inserts a data element **ITEM** at **Kth** position in a linear array. Here **A** is a Linear Array with **SIZE** locations, **N** is the number of total values stored in **A** and **K** is a positive integer (location where to insert) such that **K** ≤ **N**.

This algorithm inserts an element **ITEM** into the **Kth** position in **A**.

Algorithm: INSERT

1. if **N** = **SIZE** then write: "Overflow, Array is Full. " and End
2. Get **ITEM** from user to be inserted in array
[Find location in sorted array]
3. Call Find_Location_To_Insert (**A**, **N**, **K**, **ITEM**)
[Above algorithm finds the location **K** where to insert]
4. Call INSERT (**A**, **N**, **K**, **ITEM**)
5. End

Algorithm: Find_Location_To_Insert(A**, **N**, **K**, **ITEM**)**

1. **LB** = 0 and **UB** = **N** - 1
2. Repeat Step-3 for **K** = **LB** to **UB**.
3. If **A**[**K**] > **ITEM** then return **K**. [Insertion at front or in middle]
[End of Step 2 loop.]
4. return **K**. [Insertion at the end of array]
5. End.

Algorithm: INSERT (A**, **N**, **K**, **ITEM**)**

1. Set **J** = **N** - 1. [Initialize counter.]
2. Repeat Step-3 and 4 while **J** ≥ **K**.
3. Set **A**[**J**+1] = **A**[**J**]. [Move **J**th element downward.]
4. Set **J** = **J**-1. [Decrease counter.]
[End of Step 2 loop.]
5. Set **A**[**K**] = **ITEM**. [Insert element.]
6. Set **N** = **N**+1. [Reset **N**.]
7. End.

3. Deletion refers to the operation of removing one of the element from the array '**A**'. Deleting an item at the end of the array presents no difficulties. But, deleting an item from beginning or somewhere in the middle of array would require that each subsequent element be moved one location upward in order to fill up the array.

The following algorithm deletes the **Kth** element from a linear array **A** with **N** values stored in it.

Algorithm:

1. If **N = 0** then write: "Underflow. Array is Empty. " and End
2. Get **ITEM** from user to be deleted from array
[Find location in sorted array]
3. Call Find_Location_To_Delete(A, N, K, ITEM)
4. If found Call DELETE (A, N, K, ITEM) [if K = -1 then not found]
Else write: "Not Found in array"
5. End

Algorithm: Find_Location_To_Delete(A, N, K, ITEM)

1. Repeat Step 2 for I = LB to UB.
2. If **A[I] = ITEM** then K = I and return K. [Deletion from front or from middle]
[End of Step 2 loop.]
3. K = -1 and return K. [ITEM Not found in array]
4. End.

Algorithm: DELETE (A, N, K, ITEM)

1. Set ITEM = A[K].
2. Repeat for J=K to N-1:
Set A[J] = A[J+1] [Move J + 1 element upward.].
[End of loop.]
3. Set N = N-1. [Reset the number "N" of element in A.]
4. End.

Source code:

```
#include<iostream.h>
#include<process.h>
int const SIZE=10;
int A[SIZE], N=0; // global variables,
// no need to use them as arguments/parameters
// Function definition before main( ) function
bool IsFull( ) { if (N==SIZE) return true; else return false;}
bool IsEmpty( ) { if (N==0) return true; else return false; }

int Find_Location_To_Insert(int ITEM )
{
    int LB=0,UB=N-1;
    for(int K=LB;K<=UB;K++) { if(A[K]>ITEM) return K ; } // Insertion at front or in middle
    return K; // Insertion at the end
}

void Insert( int K, int ITEM)
{
    int J=N-1;
    while (J>=K){A[J+1]=A[J]; J--;} // shifting elements one position downward
    A[K]=ITEM; // insert Item at Kth location
    N++; // increase N by one, so that one more element has been inserted
}
```

```

void Traverse( )
{
    int LB=0,UB=N-1;
    for( int l=LB;l<=UB;l++ ) cout<<A[l]<<"\t";
}

int Find_Location_To_Delete( int ITEM )
{
    int LB=0,UB=N-1;
    for( int K=LB;K<=UB;K++ )
        { if(A[K] = ITEM) return K; } // found at location K

    K= -1; return K; // not found
}

int Delete( int K, int ITEM )
{
    ITEM = A[K];
    for ( int J=K ; J<=N-1 ; J++ )
        A[J] = A[J+1]; // Move J + 1 element upward
    N--; //Reset N by decreasing it value by 1
    return ITEM;
}

int main( )
{
    int ch, ITEM, K, loop=1;
    while( loop )
        { cout<<"\n\n\n\n\n";
          cout<<"\t1- insert value\n";
          cout<<"\t2- delete value\n";
          cout<<"\t3- traverse array\n";
          cout<<"\t4- exit\n\n";
          cout<<"\n\t\tyour choice :";   cin>>ch;
          switch( ch )
              { case 1: if( IsFull( ) ) {cout<<"\n\nArray is full\n\n"; break;}
                  cout<<"\n\n For Insertion, Put a value : ";
                  cin>>ITEM;
                  K=Find_Location_To_Insert( ITEM );
                  Insert( K, ITEM );
                  break;
                case 2: if( IsEmpty( ) ) {cout<<"\n\nArray is Empty\n\n"; break;}
                  cout<<"\n\n For Deletion, Put a value : ";
                  cin>>ITEM;
                  K= Find_Location_To_Delete( ITEM );
                  if( K == -1 ) cout<<"\n\n"<<ITEM<<" Not Found in array \n";
                  else cout<<"\n\n "<<Delete( K, ITEM )<<" deleted from array\n ";
                  break;
                case 3: if( IsEmpty( ) ) {cout<<"\n\nArray is Empty\n\n"; break;}
                  Traverse( );
                  break;
                case 4: loop=0; break;
                default: cout<<"\n\nInvalid choice\n";
              } // end of switch
        } // end of while loop
    return 0;
} // end of main( )

```

4. Sorting in Linear Array:

Sorting an array is the ordering the array elements in **ascending** (increasing -from min to max) or **descending** (decreasing – from max to min) order.

Example:

- {2 1 5 7 4 3} → {1, 2, 3, 4, 5, 7} **ascending** order
- {2 1 5 7 4 3} → {7, 5, 4, 3, 2, 1} **descending** order

Bubble Sort:

The technique we use is called "*Bubble Sort*" because the smaller value gradually bubbles their way up to the top of array like air bubble rising in water, while the large values sink to the bottom of array.

This technique is to make several passes through the array. On each pass, successive pairs of elements are compared. If a pair is in increasing order (or the values are identical), we leave the values as they are. If a pair is in decreasing order, their values are swapped in the array.



Bubble Sort

Pass = 1	Pass = 2	Pass = 3	Pass=4
<u>2</u> 1 5 7 4 3	<u>1</u> 2 5 4 3 7	<u>1</u> 2 4 3 5 7	<u>1</u> 2 3 4 5 7
1 <u>2</u> 5 7 4 3	1 <u>2</u> 5 4 3 7	1 <u>2</u> 4 3 5 7	1 <u>2</u> 3 4 5 7
1 2 <u>5</u> 7 4 3	1 2 <u>5</u> 4 3 7	1 2 <u>4</u> 3 5 7	1 2 <u>3</u> 4 5 7
1 2 5 <u>7</u> 4 3	1 2 4 <u>5</u> 3 7	1 2 3 <u>4</u> 5 7	1 2 3 4 <u>5</u> 7
1 2 5 4 <u>7</u> 3	1 2 4 3 <u>5</u> 7		
1 2 5 4 3 <u>7</u>			

- Underlined pairs show the comparisons. For each pass there are **size-1** comparisons.
- Total number of comparisons= **(size-1)²**

Algorithm: (Bubble Sort) BUBBLE (DATA, N)
 Here DATA is an Array with N elements. This algorithm sorts the elements in DATA.

1. for pass=1 to N-1.
2. for (i=0; i<= N-Pass; i++)
3. If DATA[i]>DATA[i+1], then:
 - Interchange DATA[i] and DATA[i+1].
 - [End of If Structure.]
- [End of inner loop.]
- [End of Step 1 outer loop.]
4. Exit.

/ This program sorts the array elements in the ascending order using bubble sort method */*

```
#include <iostream.h>
#define SIZE 6
void BubbleSort(int [ ], int);
int main()
{
    int a[SIZE]= {77,42,35,12,101,6};
    int i;
    cout<< "The elements of the array before sorting\n";
    for (i=0; i<= SIZE-1; i++)
        cout<< a[i]<<"\t";

    BubbleSort(a, SIZE);
    cout<< "\n\nThe elements of the array after sorting\n";
    for (i=0; i<= SIZE-1; i++)
        cout<< a[i]<<"\t";

    return 0;
}
```

```
void BubbleSort(int A[ ], int N)
{
    int i, pass, hold;
    for (pass=1; pass<= N-1; pass++)
    {
        for (i=0; i<= SIZE-pass; i++)
        {
            if(A[i] >A[i+1])
            {
                hold =A[i];
                A[i]=A[i+1];
                A[i+1]=hold;
            }
        }
    }
}
```

Home Work

Write a program to determine the median of the array given below:
(9, 4, 5, 1, 7, 78, 22, 15, 96, 45,25)

Note that the median of an array is the middle element of a sorted array.

5. Searching in Linear Array:

The process of finding a particular element of an array is called **Searching**". If the item is not present in the array, then the search is unsuccessful.

There are two types of search (**Linear search** and **Binary Search**)

Linear Search:

The linear search compares each element of the array with the **search key** until the search key is found. To determine that a value is not in the array, the program must compare the search key to every element in the array. It is also called "**Sequential Search**" because it traverses the data sequentially to locate the element.

Algorithm: (Linear Search)

LINEAR (A, SKEY)

Here **A** is a Linear Array with N elements and SKEY is a given item of information to search. This algorithm finds the location of SKEY in **A** and if successful, it returns its location otherwise it returns -1 for unsuccessful.

1. Repeat for i = 0 to N-1
2. if(A[i] = SKEY) return i [Successful Search]
[End of loop]
3. return -1 [Un-Successful]
4. Exit.

/ This program use linear search in an array to find the LOCATION of the given Key value */*

/ This program is an example of the Linear Search*/*

```
#include <iostream.h>
```

```
#define N 10
```

```
int LinearSearch(int [ ], int); // Prototyping
```

```
int main()
```

```
{
```

```
int A[N]= {9, 4, 5, 1, 7, 78, 22, 15, 96, 45}, Skey, LOC;
```

```
cout<< "Enter the Search Key: ";
```

```
cin>>Skey;
```

```
LOC = LinearSearch( A, Skey); // call a function
```

```
if(LOC == -1)
```

```
cout<< "The search key is not in the array\n Un-Successful Search\n";
```

```
else
```

```
cout<< "The search key "<<Skey<< " is at location "<<LOC<<endl;
```

```
return 0;
```

```
}
```

```
int LinearSearch (int b[ ], int skey) // function definition
```

```
{
```

```
int i;
```

```
for (i=0; i<= N-1; i++) if(b[i] == skey) return i;
```

```
return -1;
```

```
}
```


Binary Search:

It is useful for the large arrays. The binary search algorithm can only be used with **sorted array** and eliminates one half of the elements in the array being searched after each comparison. The algorithm locates the middle element of the array and compares it to the search key. If they are equal, the search key is found and array subscript of that element is returned. Otherwise the problem is reduced to searching one half of the array. If the search key is less than the middle element of array, the first half of the array is searched. If the search key is not the middle element of in the specified sub array, the algorithm is repeated on one quarter of the original array. The search continues until the sub array consist of one element that is equal to the search key (search successful). But if Search-key not found in the array then the value of END of new selected range will be less than the START of new selected range. This will be explained in the following example:

Binary Search		
Search-Key = 22		Search-Key = 8
A[0]	3	Start=0 End = 9 Mid=int(Start+End)/2 Mid= int (0+9)/2 Mid=4
A[1]	5	
A[2]	9	
A[3]	11	Start=4+1 = 5 End = 9 Mid=int(5+9)/2 = 7
A[4]	15	
A[5]	17	Start = 5 End = 7 - 1 = 6 Mid = int(5+6)/2 =5
A[6]	22	
A[7]	25	
A[8]	37	Start = 5+1 = 6 End = 6 Mid = int(6 + 6)/2 = 6
A[9]	68	
Found at location 6 Successful Search		Start=0 End = 3 Mid=int(0+3)/2 = 1
		Start = 1+1 = 2 End = 3 Mid = int(2+3)/2 =2
		Start = 2 End = 2 - 1 = 1
		End is < Start Un-Successful Search

Algorithm: (Binary Search)

Here **A** is a sorted Linear Array with N elements and SKEY is a given item of information to search. This algorithm finds the location of SKEY in **A** and if successful, it returns its location otherwise it returns -1 for unsuccessful.

BinarySearch (A, SKEY)

1. [Initialize segment variables.]
Set START=0, END=N-1 and MID=INT((START+END)/2).
2. Repeat Steps 3 and 4 while START ≤ END and A[MID]≠SKEY.
3. If SKEY < A[MID]. Then
Set END=MID-1.
Else Set START=MID+1.
[End of If Structure.]
4. Set MID=INT((BEG+END)/2).
[End of Step 2 loop.]
5. If A[MID]= SKEY then Set LOC= MID
Else:
Set LOC = -1
[End of IF structure.]
6. return LOC and Exit

```
#include <iostream.h>
#define N 10
int BinarySearch(int [ ], int); // Function Prototyping
int main()
{
    int A[N]= {3, 5, 9, 11, 15, 17, 22, 25, 37, 68}, SKEY, LOC;
    cout<< "Enter the Search Key\n";
    cin>>SKEY;
    LOC = BinarySearch(A, SKEY); // Function call
    if(LOC == -1)
        cout<< "The search key is not in the array\n";
    else
        cout<< "The search key "<<Skey<< " is at location "<<LOC<<endl;
    return 0;
}
int BinarySearch (int A[], int skey)
{
    int START=0, END= N-1, MID=int((START+END)/2), LOC;
    while(START <= END && A[MID] != skey)
    {
        if(skey < A[MID])
            END = MID - 1;
        Else
            START = MID + 1;
        MID=int((START+END)/2)
    }
    If(A[MID] == skey) LOC=MID else LOC= -1;
    Return LOC;}
```

Computational Complexity of Binary Search

The **Computational Complexity** of the **Binary Search** algorithm is measured by the maximum (worst case) number of Comparisons it performs for searching operations.

The searched array is divided by 2 for each comparison/iteration.

Therefore, the maximum number of comparisons is measured by:

$\log_2(n)$ where n is the size of the array

Example:

If a given sorted array 1024 elements, then the maximum number of comparisons required is:

$\log_2(1024) = 10$ (only 10 comparisons are enough)

Computational Complexity of Linear Search

Note that the **Computational Complexity** of the **Linear Search** is the maximum number of comparisons you need to search the array. As you are visiting all the array elements in the worst case, then, the number of comparisons required is:

n (n is the size of the array)

Example:

If a given an array of 1024 elements, then the maximum number of comparisons required is:

$n-1 = 1023$ (As many as 1023 comparisons may be required)

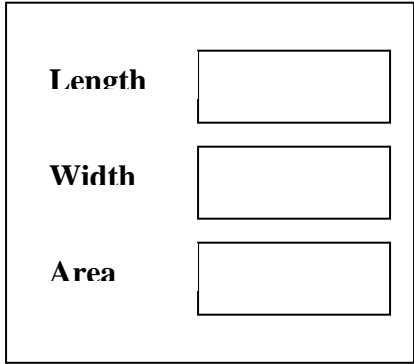
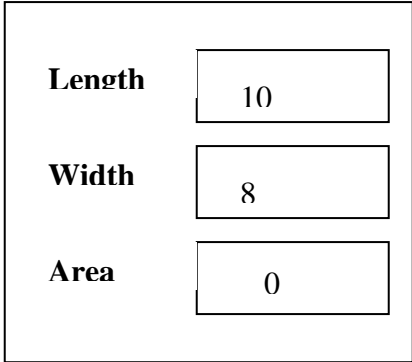
D A T A S T R U C T U R E S

Structures A structure is a collection of variables under a single name. These variables can be of different types, and each has a name which is used to select it from the structure. A structure is a convenient way of grouping several pieces of related information together. They are most commonly used for record-oriented data.

Example: How to declare a structure

```
struct Rectangle    // this is type/name for structure
{
    float Length;
    float width;
    float area;
};
```

NOTE: declaration of structure does not occupy space in memory. One has to create the variables for the struct and variable will take spaces in memory. For example:

Following instruction will just occupy space Struct Rectangle Rect;	Following instruction will occupy space and also initialize members. Struct Rectangle Rect={10, 8, 0};
Rect 	Rect 

Here is an other example of structure declaration.

```
struct Student {
    char name[20];
    char course[30];
    int age;
    int year;
};
```

```
struct Student S1;    // Here s1 is a variable of Student type and has four members.
```

A structure is usually declared before main() function. In such cases the structure assumes global status and all the functions can access the structure. The members themselves are not variables they should be linked to structure variables in order to make them meaningful members. The link between a member and a variable is established using the membership operator ‘.’ Which is also known as dot operator.

For example:

```
strcpy(S1.name, "Nazir Hussain");
strcpy(S1.course, "CS-214 Data Structures");
S1.age = 21;
S1.year = 1989;
```

Note: following is the work to do in the Lab.

1- Run this program and examine its behavior.

In the following program you will see the way to initialize the structure variables.

```
#include <iostream.h>
#include <conio.h>
#include <iomanip.h>

struct student
{ int ID;           // 4   bytes
  char name[10];    // 10  bytes
  float grade;      // 4   bytes
  int age;          // 4
  char phone[10];   // 10
  char e_mail[16];  // 16
};
// Prototyping of the functions
void display(struct student);

void main()
{
  struct student s1={55,"Amir Ali",3.5f,23,"6535418","amir@yahoo.com"};
  struct student s2={26,"Mujahid",2.9888f,25,"5362169", "muj@hotmail.com"};
  struct student s3={39,"M Jamil",3.108f,30,"2345677","jam@hotmail.com"};
  struct student s4={44,"Dilawar",2.7866f,31,"5432186","dil@hotmail.com"};
  struct student s5={59,"S.Naveed",2.9f,27,"2345671","navee@yahoo.com"};
  cout<<"                      Students Records Sheet\n";
  cout<<"                      ~~~~~~\n\n";
  cout<<"ID#      NAME      GRADE  AGE   PHONE      E-MAIL\n";
  cout<<"~~~      ~~~~      ~~~~~  ~~~   ~~~~~      ~~~~~~\n";
  display(s1); // structure pass to function
  display(s2); // structure pass to function
  display(s3);
  display(s4);
  display(s5);
}

void display(struct student s)
{ cout<<setw(3)<< s.ID <<setw(12)<< s.name <<setw(8)<< setiosflags
  (ios::showpoint)<<setprecision(2)<< s.grade<<setw(5)<< s.age
  <<setw(10)<< s.phone<< setw(18)<<s.e_mail<<endl;
}
```

2- Run this program and examine its behavior.

// In this program you will see the structures (members Manipulation), // Passing structures to functions:

```
#include <iostream.h>
#include <iomanip.h>

struct STU_GRADES
{ char name [30];
  int exam1;
  int exam2;
  int exam3;
  int final;
  float sem_ave;
  char letter_grade;
};

//inputs the data items for a student, structure
//is passed by reference
struct STU_GRADES get_stu ( )
{
    struct STU_GRADES student;
    cout << "\n\n\n Enter the information for a student\n";
    cout << "   Name: ";
    cin.getline (student.name, 30, '\n');
    cout << "   Exam1: ";
    cin >> student.examl;
    cout << "   Exam2: ";
    cin >> student.exam2;
    cout << "exam3: ";
    cin >> student.exam3;
    cout << "final: ";
    cin >> student.final;
    return student;
}

//displays a student's info.
//structure is passed by value
void print_stu (struct STU_GRADES stu)
{
    cout << "\n\n\nGrade report for: " << stu.name<<endl;
    cout << "\nexam 1\texam 2\texam 3\tfinal\n";
    cout << stu.examl << "\t" << stu.exam2 << "\t"
        << stu.exam3 << "\t" << stu.final;
    cout << "\n\n\nsemester average: " << setiosflags (ios::fixed)
        << setprecision (2) << stu.sem_ave;
    cout << "\nsemester grade: " << stu.letter_grade;
}

float calc_ave (int ex1, int ex2, int ex3, int final)
{
    float ave;

    ave = float (ex1 + ex2 + ex3 + final)/4.0f;
    return ave;
}
```

```

char assign_let (float average)
{
    char let_grade;

    if (average >= 90)
        let_grade = 'A';
    else if (average >= 80)
        let_grade = 'B';
    else if (average >= 70)
        let_grade = 'C';
    else if (average >= 60)
        let_grade = 'D';
    else let_grade = 'F';

    return let_grade;
}

int main()
{
    struct STU_GRADES stu;
    char more;

    do
    {
        //pass the entire structure
        stu= get_stu ( );
        //pass elements of the strucutre
        stu.sem_ave = calc_ave (stu.examl, stu.exam2,
                               stu.exam3, stu.final);
        //pass elements of the structure
        stu.letter_grade = assign_let (stu.sem_ave);
        //pass the entire structure
        print_stu (stu);
        cout << "\n\n Enter another student? (y/n)   ";
        cin >> more;
        //grab the carriage return since
        //character data is input next
        cin.ignore (80,'\n');
    } while (more == 'y' || more == 'Y');

    return 0;
}

```

Pointers

Pointers are a fundamental part of C. If you cannot use pointers properly then you have basically lost all the power and flexibility that C allows. The secret to C is in its use of pointers.

C uses *pointers* a lot. **Why?:**

- It is the only way to express some computations.
- It produces compact and efficient code.
- It provides a very powerful tool.

C uses pointers explicitly with:

- Arrays,
- Structures,
- Functions.

NOTE: Pointers are perhaps the most difficult part of C to understand. C's implementation is slightly different from other languages.

What is a Pointer?

A pointer is a variable which contains the address in memory of another variable. We can have a pointer to any variable type.

The operator **&** gives the “address of a variable”.

The *indirection* or dereference operator ***** gives the “contents of an object *pointed to* by a pointer”.

To declare a pointer to a variable do:

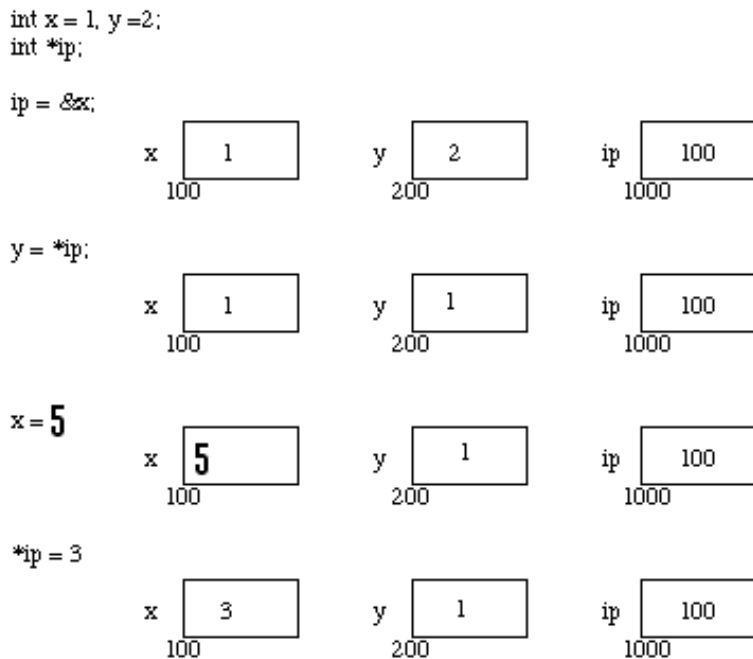
```
int *p;
```

NOTE: We must associate a pointer to a particular type: You can't assign the address of a **short int** to a **long int**, for instance.

Consider the effect of the following code:

```
int x = 1, y = 2;
int *ip;
ip = &x;
y = *ip;
x = 5;
*ip = 3;
```

It is worth considering what is going on at the *machine level* in memory to fully understand how pointer work. Consider following Fig. Assume for the sake of this discussion that variable x resides at memory location 100, y at 200 and ip at 1000. Note A pointer is a variable and thus its values need to be stored somewhere. It is the nature of the pointers value that is *new*.



Now the assignments $x = 1$ and $y = 2$ obviously load these values into the variables. **ip** is declared to be a *pointer to an integer* and is assigned to the address of x ($\&x$). So ip gets loaded with the value 100 which is the address of x.

Next y gets assigned to the *contents of ip*. In this example ip currently *points* to memory location 100 -- the location of x. So y gets assigned to the values of x -- which is 1. After that assignment of 5 to variable x.

Finally we can assign a value 3 to the contents of a pointer ($*ip$).

DATA STRUCTURES

IMPORTANT: When a pointer is declared it does not point anywhere. You must set it to point somewhere before you use it.

So ...

```
int *ip;
*ip = 50;
```

will generate an error (program crash!!).

The correct use is:

```
int *ip;
int x;
ip = &x; // setting the pointer
*ip = 100;
```

Here is another example program which will describes the usage of pointers and the contents of pointers.

```
#include <stdio.h>
int main ( )
{  int  a=3,  b=5,  S=0,  D=0,  M=0;
   int  *p1,  *p2,  *p3,  *p4,  *p5; // five pointers are declared
   // assigning address of a, b, S, D and M to these pointers
   p1 = &a; p2= &b; p3 = &S; p4 = &D; p5=&M;
   *p3 = *p1 + *p2; // same as s = a + b;
   cout<<*p3;      // it prints      8
   cout<< p1;      // it prints the address of a
   D = *p1 - b;     // it calculates -2
   cout<<"\n"<<*p4; // it prints      -2
   *p5 = a * *p2;   // it calculates  15
   cout<<"\n"<<M;   // it prints      15
}
```

The above program has been discussed in the class lecture in detail. If you still has some confusion, contact the instructor through e-mail.

Pointers and Arrays

Pointers and arrays are very closely linked in C.

Hint: think of array elements arranged in consecutive memory locations / successive.

Consider the following:

```
int a[12]= { 5, 2 , 6, 9, 12, 7, 56, 34, 11, 76, 37, 55,69};
```

The elements of the above given array will be stored as pictured in the following figure. Each element of the array occupies 4 bytes. Assume first element is stored at address 100, second element will store at address 104 and so on:

100	104	108	112	116	120	124	128	132	136	140	144
5	2	6	9	12	7	56	34	76	37	55	69
a[0]	a[1]	a[2]	a[3]	a[11]

Following is the c-language code which prints contents of all elements using pointers.

```
#include <iostream.h>
int main( )
{
    int a[12]= { 5, 2 , 6, 9, 12, 7, 56, 34, 11, 76, 37, 55,69};
    int i, *p;

    // printing array elements using index / subscript
    for ( i = 0 ; i < 12 ; i++)    cout<<a[i]<<"\t";

    // Following will store the address of a[0] into p (pointer)
    p = a; // same as p = a[0];
    for ( i = 0 ; i < 12 ; i++)
    { cout<<*p<<"\t"; // prints the contents of the address
      p++; // it shift the pointer to next element of the array
    }
    return 0;
}
```

WARNING: There is no bound checking of arrays and pointers so you can easily go beyond array memory and overwrite other things.

C however is much more fine in its link between arrays and pointers.

For example we can just type

```
p = a; // here p is pointer and a is an array.
instead of p = &a[0];
```

A pointer is a variable. We can do `p = a` and `p++`. An Array is not a variable. So `a = p` and `a++` ARE ILLEGAL.

This stuff is very important. Make sure you understand it. We will see a lot more of this.

Pointer and Functions

Let us now examine the close relationship between pointers and C's other major parts. We will start with functions.

When C passes arguments to functions it passes them by value.

There are many cases when we may want to alter a passed argument in the function and receive the new value back once the function has finished. C uses pointers explicitly to do this. The best way to study this is to look at an example where we must be able to receive changed parameters.

Let us try and write a function to swap variables around?

Pointers provide the solution: *Pass the address of the variables to the functions and access address in function.*

Thus our function call in our program would look like this:

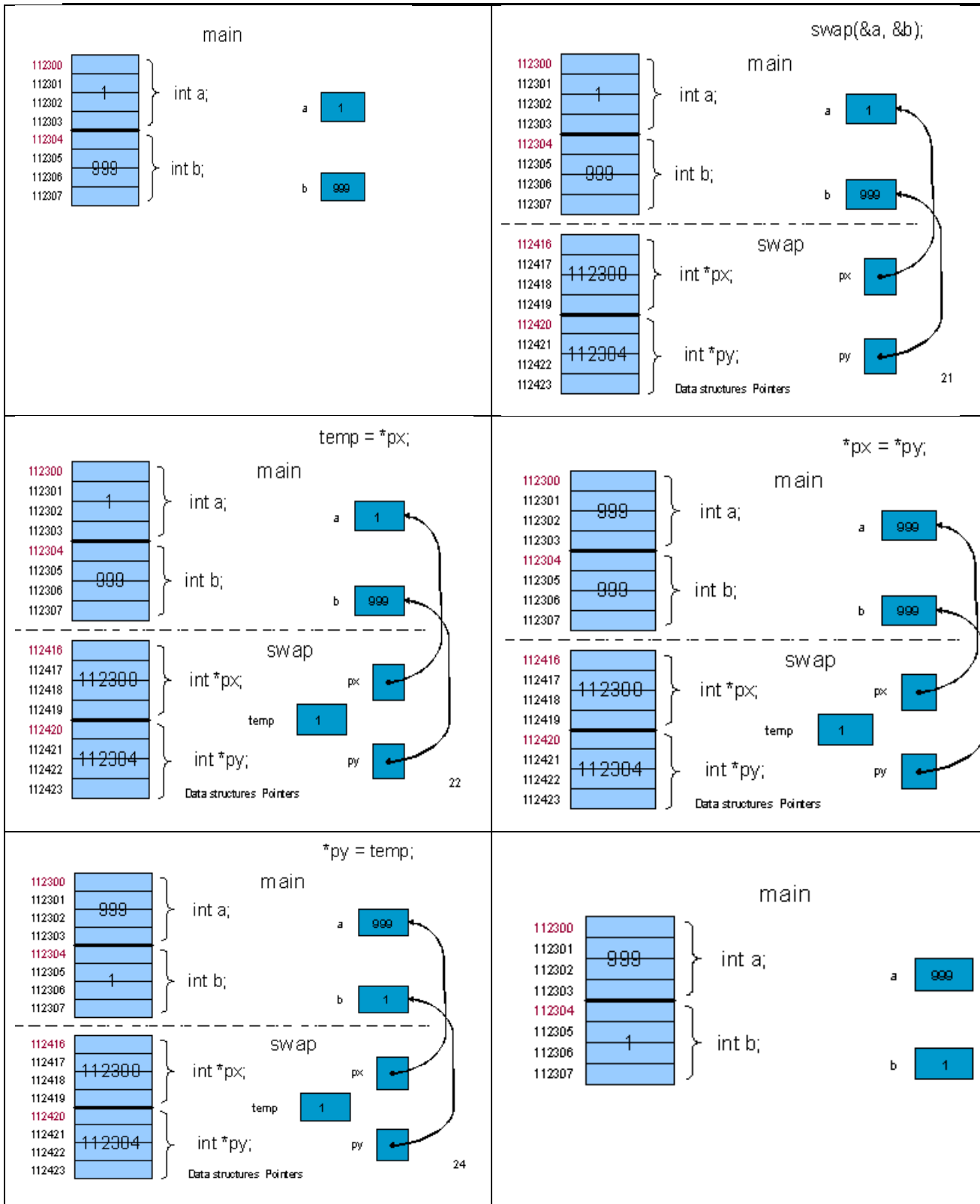
```
swap(&x, &y);
```

The Code to swap is fairly straightforward:

```
// This program swap / interchange the values of two variables
#include <stdio.h>
void swap( int *, int *); // function prototyping
int main( )
{ int    a, b;
  a = 1;
  b = 999;
  cout<<"    a  =  "<< a << "    and  b= "<<b<<endl;
  swap( &a, &b);
  cout<< "\n After Swaping the new values of  a    and  b \n";
  cout<<"    a  =  "<< a << "    and  b= "<<b<<endl;
  return 0;
}

void swap(int *px, int *py)
{ int temp;
  /* contents of pointer */
  temp = *px;
  *px = *py;
  *py = temp;
}
```

The explanation of the above program is given on the next page:

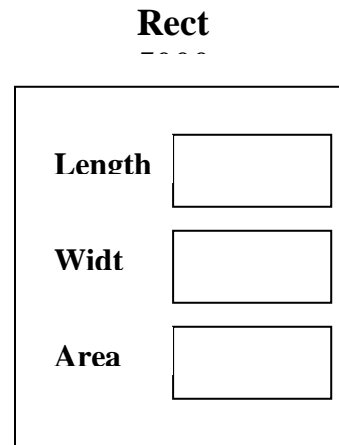


Pointers and Structures

1- Program to describe the pointer to stuctures

```
#include <iostream.h>
struct rectangle
{
    float length;
    float width;
    float area;
};

struct rectangle Rect;
Rect.length = 12.5;
Rect.width = 6.78;
Rect.area = Rect.length * Rect.width;
Struct rectangle *P;
P = &Rect;
cout<<"\n Length= "<<P->length;
cout<<"\n Width= "<<P-> width;
cout<<"\n Area= "<<P-> area;
return 0;
}
```



2- Another program to describe the pointer to structures.

// It is assumed that structure variable is stored at location 2000
// in memory

```
#include <iostream.h>
#include <string.h>
```

```
struct tag{
    char lname[20];          /* the structure type */
    char fname[20];          /* last name */
    int age;                  /* first name */
};                             /* age */

struct tag my_struct;        /* define the structure */
void show_name(struct tag *p); /* function prototype */

int main(void)
{
    struct tag *st_ptr;      /* a pointer to a structure */
    strcpy(my_struct.lname,"Shahid");
    strcpy(my_struct.fname,"Hamza");
    cout<< my_struct.fname<<"\t";
    cout<< my_struct.lname<<"\t";

    my_struct.age = 19;
    st_ptr = &my_struct;    /* points the pointer to my_struct */
    show_name(st_ptr);       /* pass the pointer to function*/
    return 0;
}

void show_name(struct tag *p)
{
    cout<< p->fname<<"\t";  /* p points to a structure */
    cout<< p->lname<<"\t";
    cout<< p->age;
}
```

My_struct 2000

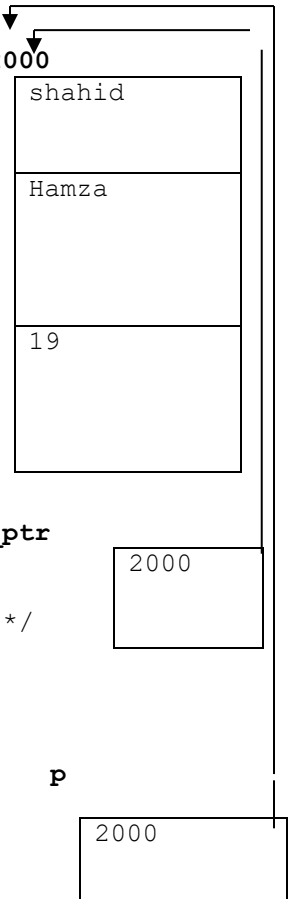
lname

fname

age

st_ptr

P



These are fairly straight forward and are easily defined. Consider the following:

the \rightarrow operator lets us access a member of the structure pointed to by a pointer. *i.e.*:

`p -> fname` will access the member "fname" of "p" pointer.

`P -> age` will access the member "age" of "p" pointer.

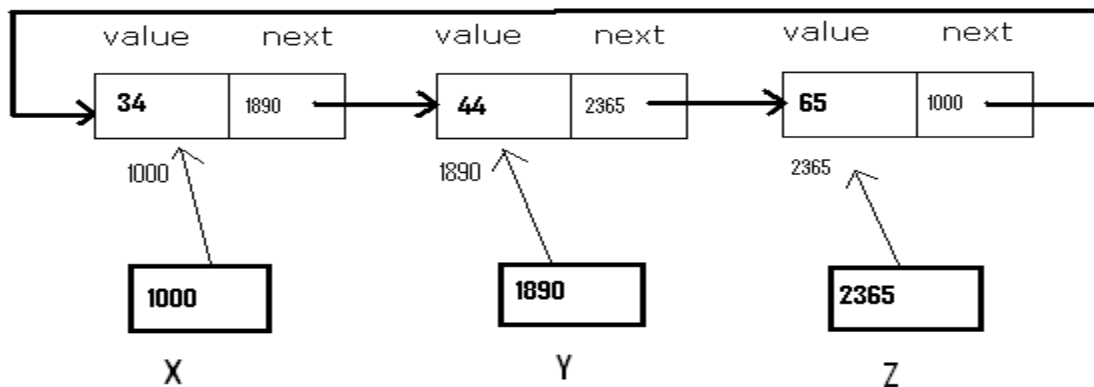
See another example, which is a little bit complex:

```
struct Node {
    int value;
    struct Node* next;
};
// Allocate the pointers
struct Node *x;
struct Node *y;
struct Node *z;

// Allocate the pointees
x = (struct Node*) malloc(sizeof(Node)); // x = new Node;
y = (struct Node*) malloc(sizeof(Node)); // y = new Node;
z = (struct Node*) malloc(sizeof(Node)); // z = new Node;

// Put the numbers in the pointees
x->value = 34;
y->value = 44;
z->value = 65;

// Put the pointers in the pointees
x->next = y;
y->next = z;
z->next = x;
}
```



Home Work

Exercise -1

Write a C program to read through a 10 elements into dynamic array of integer type using pointers. Search through this array to find the biggest and smallest value.

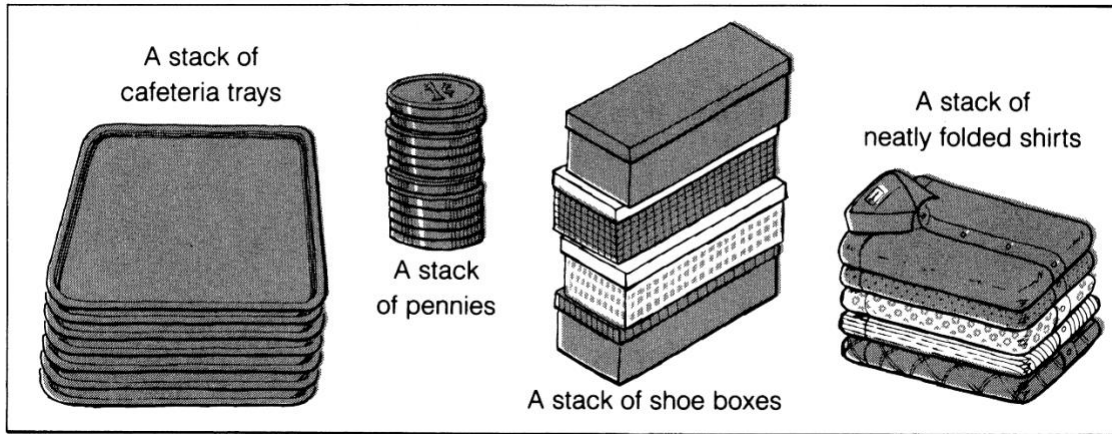
Exercise - 2

Write a program that takes three variable (a, b, c). Rotates the values stored so that value **a** goes to **b**, **b** to **c** and **c** to **a**.

Note: make a function which takes pointers of these variables and using pointers it rotates the values.

STACKS:

It is an ordered group of homogeneous items or elements. Elements are added to and removed from the top of the stack (the most recently added items are at the top of the stack). The last element to be added is the first to be removed (LIFO: Last In, First Out).



A stack is a list of elements in which an element may be inserted or deleted only at one end, called **TOP** of the stack. The elements are removed in reverse order of that in which they were inserted into the stack.

Basic operations:

These are two basic operations associated with stack:

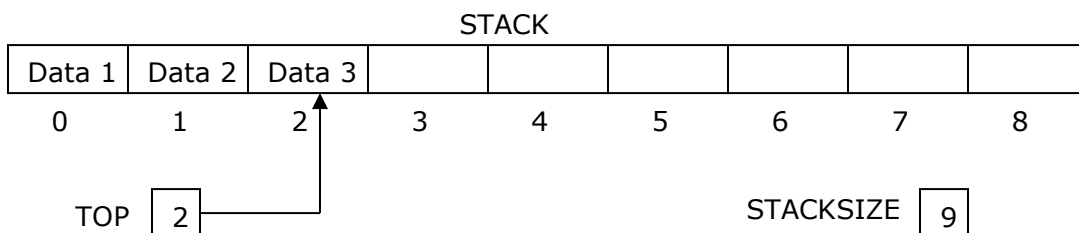
- **Push()** is the term used to insert/add an element into a stack.
- **Pop()** is the term used to delete/remove an element from a stack.
- **Peak()** is the term to return/get the value on the Top of stack without deleting it.

Other names for stacks are **piles** and **push-down lists**.

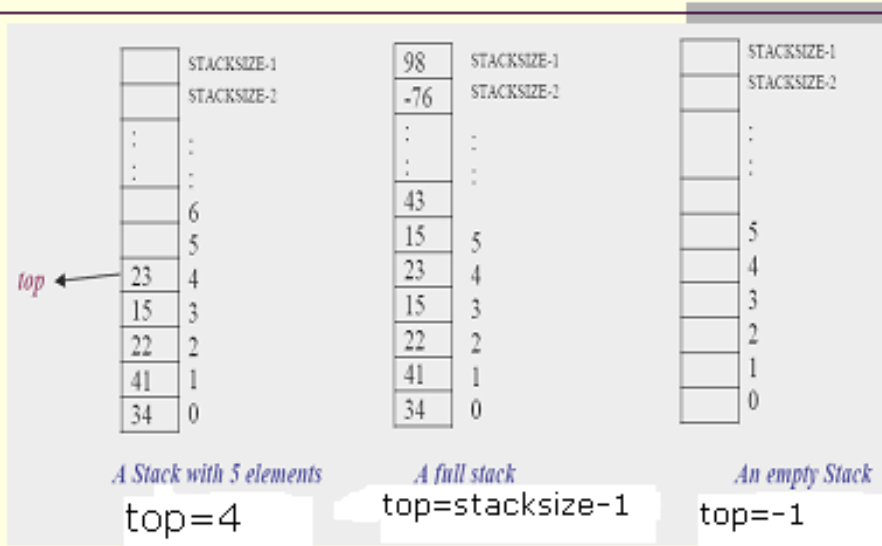
There are two ways to represent *Stack* in memory. One is using array and other is using linked structure.

Array representation of stacks:

Usually the stacks are represented in the computer by a linear array. In the following algorithms/procedures of pushing and popping an item from the stacks, we have considered, a linear array **STACK**, a variable **TOP** which contain the location of the top element of the stack; and a variable **STACKSIZE** which gives the maximum number of elements that can be hold by the stack.



Stacks



Push Operation

Push an item onto the top of the stack (*insert an item*)

	STACKSIZE-1
	STACKSIZE-2
:	:
:	:
	6
	5
23	4
15	3
22	2
41	1
34	0

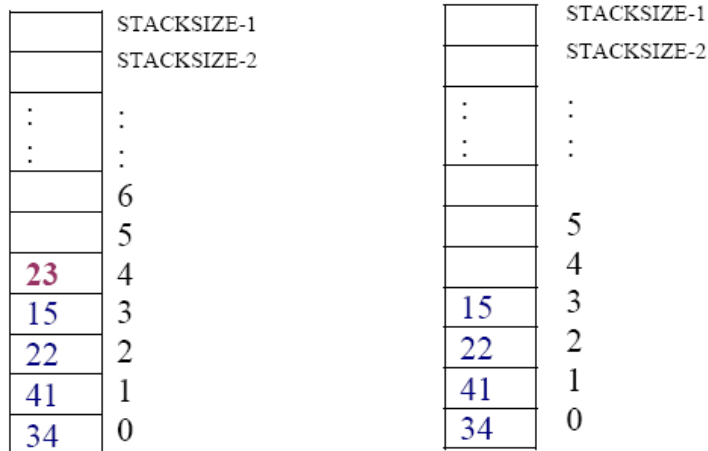
Before PUSH
(top=4, count=5)

	STACKSIZE-1
	STACKSIZE-2
:	:
:	:
15	5
23	4
15	3
22	2
41	1
34	0

After PUSH
(top=5, count= 6)

Pop Operation

Pop an item off the top of the stack (*delete an item*)



Before POP
(top=4, count=5)

After POP
(top=3 count=4)

Algorithm for PUSH:

Algorithm: PUSH(STACK, TOP, STACKSIZE, ITEM)

1. [STACK already filled?
If TOP=STACKSIZE-1, then: Print: OVERFLOW / Stack Full, and Return.
2. Set TOP:=TOP+1. [Increase TOP by 1.]
3. Set STACK[TOP]=ITEM. [Insert ITEM in new TOP position.]
4. RETURN.

Algorithm for POP:

Algorithm: POP(STACK, TOP, ITEM)

This procedure deletes the top element of STACK and assigns it to the variable ITEM.

1. [STACK has an item to be removed? Check for empty stack]
If TOP=-1, then: Print: UNDERFLOW/ Stack is empty, and Return.
2. Set ITEM=STACK[TOP]. [Assign TOP element to ITEM.]
3. Set TOP=TOP-1. [Decrease TOP by 1.]
4. Return.

Algorithm for Peak:

Algorithm: PEAK(STACK, TOP, ITEM)

This procedure returns the top element of STACK without deleting (Pop) it.

1. [STACK has an item to be removed? Check for empty stack]
If TOP=-1, then: Print: UNDERFLOW/ Stack is empty, and Return.
2. Set ITEM=STACK[TOP]. [Assign TOP element to ITEM.]
3. Return ITEM.

Here are the minimal operations we'd need for an abstract stack (and their typical names):

- `Push()` : Places an element/value on the *top* of the stack.
- `Pop()` : Removes value/element from the *top* of the stack.
- `Peak()` : value/element from the *top* of the stack without deleting it.
- `IsEmpty()` : Reports whether the stack is Empty or not.
- `IsFull()` : Reports whether the stack is Full or not.

```
// Run this program and examine its behavior.
// A Program that exercise the operations on Stack Implementing Array
// i.e. (Push, Pop, Traverse)
#include <iostream>
using namespace std;
#define STACKSIZE 10
// global variables declation
int Top=-1;
int Stack[STACKSIZE];

void Push(int);    // functions prototyping
int Pop(void);
bool IsEmpty(void);
bool IsFull(void);
void Traverse(void);

int main( )
{ int item, choice, loop=1;
  while( loop )
  {
    cout<<"\n\n\n\n\n";
    cout<<"      *****  STACK  OPERATIONS  ***** \n\n";
    cout<<"  1-  Push item \n  2-  Pop Item \n";
    cout<<"  3-  Traverse / Display Stack Items \n  4-  Exit.";
    cout<<" \n\n\t Your choice ---> ";
    cin>>choice;
    switch(choice)
    { case 1: if(IsFull()) cout<<"\n Stack Full/Overflow\n";
              else
              { cout<<"\n Enter a number: "; cin>>item;
                Push(item);}
              break;
      case 2: if(IsEmpty())cout<<"\n (Stack is empty) \n";
              else
              {item=Pop();
                cout<<"\n deleted from Stack = "<<item<<endl;}
              break;
      case 3: if(IsEmpty())cout<<"\n (Stack is empty) \n";
              else
              { cout<<"\n List of Item pushed on Stack:\n";
                Traverse();
              }
              break;

      case 4: loop=0; break;
      default:
                cout<<"\n\n\t Invalid Choice: \n";
    } // end of switch block
  } // end of while loop
} // end of of main() function
```

```
// function definitions
void Push(int item)
{   Stack[++Top] = item; }

int Pop( )
{   return Stack[Top--]; }

bool IsEmpty( )
{ if(Top == -1 ) return true; else return false; }

bool IsFull( )
{ if(Top == STACKSIZE-1 ) return true; else return false; }

void Traverse( )
{ int TopTemp = Top;
  do{ cout<<Stack[TopTemp--]<<"\n";} while(TopTemp>= 0);
}

//
// A Program that exercise the operations on Stack
// Implementing POINTER (Linked Structures) (Dynamic Binding)
// This program provides you the concepts that how STACK is
// implemented using Pointer/Linked Structures

#include <iostream>
using namespace std;

struct node { int info;
              struct node *next;
            };
struct node *TOP = NULL;

void Push (int x)
{ struct node *NewNode = new node;
  if(NewNode == NULL) { cout<<"\n\n Memeory Crash\n\n"; return; }
  NewNode->info = x;
  NewNode->next = TOP;
  TOP=NewNode;
}

struct node* Pop ()
{ struct node *T;
  T=TOP;
  TOP = TOP->next;
  return T;
}

void Traverse()
{ struct node *T;
  for( T=TOP ; T!=NULL ;T=T->next) cout<<T->info<<endl;
}

bool IsEmpty()
{ if(TOP == NULL) return true; else return false; }
```

```

int main ()
{ struct node *T;
  int item, ch, loop=1;

  while(loop)
  {
    cout<<"\n\n\n\n\n";
    cout<<"      *****  STACK  OPERATIONS  ***** \n\n";
    cout<<" 1-  Push item \n 2-  Pop Item \n";
    cout<<" 3-  Traverse / Display Stack Items \n 4-  Exit.";
    cout<<" \n\n\t Your choice ---> ";
    cin>>ch;

    switch(ch)
    { case 1:      cout<<"\n Enter a number: "; cin>>item;
                Push(item);
                break;
      case 2:
                if(IsEmpty()) {cout<<"\n\n Stack is Empty\n";
                               break;
                               }
                T= Pop();
                cout<<"\n\n"<<T->info<<" has been deleted \n";
                delete T; // release the memory
                break;
      case 3:
                if(IsEmpty()) {cout<<"\n\n Stack is Empty\n";
                               break;
                               }
                Traverse();
                break;
      case 4:
                loop=0; break;

    } // end of switch block
  } // end of loop
  return 0;
} // end of main function

```

Application of the Stack (Arithmetic Expressions)



INFIX, POSTFIX AND PREFIX NOTATIONS

Infix	Postfix	Prefix
A+B	AB+	+AB
A+B-C	AB+C-	-+ABC
(A+B)*(C-D)	AB+CD-*	*+AB-CD

Infix, Postfix and Prefix notations are used in many calculators. The easiest way to implement the Postfix and Prefix operations is to use stack. Infix and prefix notations can be converted to postfix notation using stack.

The reason why postfix notation is preferred is that you don't need any parenthesis and there is no precedence problem.

Stacks are used by compilers to help in the process of converting infix to postfix arithmetic expressions and also evaluating arithmetic expressions. Arithmetic expressions consisting variables, constants, arithmetic operators and parentheses.

Humans generally write expressions in which the operator is written between the operands (**3 + 4**, for example). This is called infix notation. Computers "prefer" postfix notation in which the operator is written to the right of two operands. The preceding infix expression would appear in postfix notation as **3 4 +**.

To evaluate a complex infix expression, a compiler would first convert the expression to postfix notation, and then evaluate the postfix version of the expression. We use the following three levels of precedence for the five binary operations.

Precedence	Binary Operations
Highest	Exponentiations (^)
Next Highest	Multiplication (*), Division (/) and Mod (%)
Lowest	Addition (+) and Subtraction (-)

For example:

(66 + 2) * 5 - 567 / 42

to postfix

66 22 + 5 * 567 42 / -

Transforming Infix Expression into Postfix Expression:

The following algorithm transform the infix expression **Q** into its equivalent postfix expression **P**. It uses a stack to temporary hold the operators and left parenthesis. The postfix expression will be constructed from left to right using operands from **Q** and operators popped from STACK.

Algorithm: Infix_to_PostFix(Q, P)

Suppose **Q** is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression **P**.

1. Push "(" onto STACK, and add ")" to the end of **Q**.
2. Scan Q from left to right and repeat Steps 3 to 6 for each element of Q until the STACK is empty:
3. If an operand is encountered, add it to **P**.
4. If a left parenthesis is encountered, push it onto STACK.
5. If an operator \odot is encountered, then:
 - a) Repeatedly pop from STACK and add to **P** each operator (on the top of STACK) which has the same or higher precedence/priority than \odot
 - b) Add \odot to STACK.
 [End of If structure.]
6. If a right parenthesis is encountered, then:
 - a) Repeatedly pop from STACK and add to **P** each operator (on the top of STACK) until a left parenthesis is encountered.
 - b) Remove the left parenthesis. [Do not add the left parenthesis to **P**.]
 [End of If structure.]
 [End of Step 2 loop.]
7. Exit.

Convert **Q**: **A+(B * C - (D / E ^ F) * G) * H** into postfix form showing stack status .
 Now add ")" at the end of expression **A+(B * C - (D / E ^ F) * G) * H)**
 and also Push a "(" on Stack.

Symbol Scanned	Stack	Expression Y
	(
A	(A
+	(+	A
((+(A
B	(+(AB
*	(+(*	AB
C	(+(*	ABC
-	(+(-	ABC*
((+(-(ABC*
D	(+(-(ABC*D
/	(+(-(/	ABC*D
E	(+(-(/	ABC*DE
^	(+(-(/^	ABC*DE
F	(+(-(/^	ABC*DEF
)	(+(-	ABC*DEF^/
*	(+(-*	ABC*DEF^/
G	(+(-*	ABC*DEF^/G
)	(+	ABC*DEF^/G*-
*	(+*	ABC*DEF^/G*-
H	(+*	ABC*DEF^/G*-H
)	empty	ABC*DEF^/G*-H*+

Evaluation of Postfix Expression:

If **P** is an arithmetic expression written in postfix notation. This algorithm uses STACK to hold operands, and evaluate **P**.

Algorithm: This algorithm finds the VALUE of **P** written in postfix notation.

1. Add a Dollar Sign "\$" at the end of **P**. [This acts as sentinel.]
2. Scan **P** from left to right and repeat Steps 3 and 4 for each element of **P** until the sentinel "\$" is encountered.
3. If an operand is encountered, put it on STACK.
4. If an operator \odot is encountered, then:
 - a) Remove the two top elements of STACK, where **A** is the top element and **B** is the next-to-top-element.
 - b) Evaluate **B** \odot **A**.
 - c) Place the result of (b) back on STACK.

[End of If structure.]

[End of Step 2 loop.]
5. Set VALUE equal to the top element on STACK.
6. Exit.

For example: Following is an infix arithmetic expression
 $(5 + 2) * 3 - 8 / 4$

And its postfix is:

5 2 + 3 * 8 4 / -

Now add "\$" at the end of expression as a sentinel.

Scanned Elements	Stack	Action to do
5	5	Pushed on Stack
2	5, 2	Pushed on Stack
+	7	POP the two top elements and calculate $5 + 2$ and push the result back on Stack
3	7, 3	Pushed on Stack
*	21	POP the two top elements and calculate $7 * 3$ and push the result on Stack
8	21, 8	Pushed on Stack
4	21, 8, 4	Pushed on Stack
/	21, 2	POP the two top elements and calculate $8 / 2$ and push the result on Stack
-	19	POP the two top elements and calculate $21 - 2$ and push the result on Stack
\$	19	Sentinel \$ encounter, Result is on top of the STACK

Here is the code which transform an infix arithmetic expression into Postfix arithmetic expression.

```
// This program provides you the concepts that how an infix
// arithmetic expression will be converted into post-fix expression
// using STACK

// Conversion Infix Expression into Post-fix
// NOTE: ^ is used for raise-to-the-power

#include<iostream>
#include<cstring>
using namespace std;

int Top = -1;
char Q[100],P[100],Stack[100]; //Q is infix and P is Postfix array
int n=0; // used to count item inserted in P

void Push(char ch) { Stack[++Top] = ch; }
char Pop( ) { return Stack[Top--];}
char Peak( ) { return Stack[Top]; }

int main()
{ int k,i;
  cout<<"Put an arithematic INFIX _Expression\n\n\t\t";
  cin.getline(Q,99); // Function gets infix expression into Q as string

  k=strlen(Q); // it calculates the length of Q and store it in k
  // following two lines will do initial work with Q and stack
  strcat(Q,""); // This function add ) at the and of Q
  Push('('); // This statement will push first ( on Stack )

  while(Top!= -1)
  {for(i=0;i<=k;i++)
  {
    switch(Q[i])
    {case '+':
     case '-':
      while(1)
      { if(Peak() != '(' ) { P[n++]=Pop(); }
        else break;
      }
      Push(Q[i]); break;

    case '*':
    case '/':
    case '%':
      while(1)
      {if(Peak() == '(' || Peak() == '+' || Peak() == '-') break;
        else
          P[n++]=Pop();
      }
      Push(Q[i]); break;

    case '^':
      while(1)
      { if( Peak() == '(' || Peak() == '+' ||
          Peak() == '-' || Peak() == '/' ||
          Peak() == '*' || Peak() == '%') break;
        else
          P[n++] = Pop();
      }
      Push(Q[i]);
```

```
break;
```

```
case '(':
    Push(Q[i]);
    break;
```

```
case ')':
    while(1)
    {
        if(Stack[Top]=='(' ) {Top--; break;}
        else { P[n++]=Pop(); }
    }
    break;
```

```
default :    // it means that read item is an operand
            P[n++]=Q[i];
    } //END OF SWITCH
    } //END OF FOR LOOP
    } //END OF WHILE LOOP
```

```
P[n]='\0'; // this statement will put string terminator at the
           // end of P which is Postfix expression
cout<<"\n\n POSTFIX EXPRESION IS \n\n\t\t"<<P<<endl;
return 0;
} //END OF MAIN FUNCTION
```

D A T A S T R U C T U R E S

```
// This program provides you the concepts that how a post-fixed
// expression is evaluated using STACK. In this program you will
// see that linked structures (pointers are used to maintain the stack.

// NOTE: ^ is used for rais-to-the-power

#include <iostream>
#include <conio.h>
#include <math.h>
#include <stdlib.h>
#include <ctype.h>
using namespace std;

struct node {    int info;
                 struct node *next;
                };

struct node *TOP = NULL;

void push (int x)
{ struct node *NN = new node; // creation of new node
  NN->info = x;

  NN->next = TOP;
  TOP = NN;
}

struct node* pop ()
{ struct node *Q;
  if(TOP==NULL) { cout<<"\n Stack is Empty \n\n";
                  system("pause");
                  exit(0); // to stop the execution
                }
  else
  { Q = TOP;
    TOP = TOP->next;
    return Q;
  }
}

int main(void)
{char t;
 struct node *Q, *A, *B;
 cout<<"\n\n Put a post-fix expression (only numbers), $ at end of exp: \n";

 while(1)
 {   t=getche(); // this will read one character and store it in t

     if(isdigit(t)) push(t-'0'); // this will convert char to int
     else if(t==' ')continue;
     else if(t=='$') break;
     else
     {
        A= pop();
        B= pop();

        switch (t)
        {
            case '+':
                push(B->info + A->info);
                break;

            case '-':
```

D A T A S T R U C T U R E S

```

                                push(B->info - A->info);
                                break;
        case '*':
                                push(B->info * A->info);
                                break;
        case '/':
                                push(B->info / A->info);
                                break;
        case '^':
                                push(pow(B->info, A->info));
                                break;
        default:
                                cout<<"\n\n Error unknown operator \n\n";
                                } // end of switch
    } // end of if structure
} // end of while loop

Q=pop(); // this will get top value from stack which is result
cout<<"\n\n\n The result of this expression is = "<<Q->info<<endl;
delete Q, A, B; // release memory
return 0;
} // end of main function
```

Queue:

A queue is a linear list of elements in which deletion can take place only at one end, called the **front**, and insertions can take place only at the other end, called the **rear**. The term "front" and "rear" are used in describing a linear list only when it is implemented as a queue.

Queue is also called **first-in-first-out (FIFO)** lists. Since the first element in a queue will be the first element out of the queue. In other words, the order in which elements enters a queue is the order in which they leave.

There are main two ways to implement a queue :

1. Circular queue using array
2. Linked Structures (Pointers)

Primary queue operations:

Enqueue: insert an element at the rear of the queue

Dequeue: remove an element from the front of the queue

Following is the algorithm which describes the implementation of Queue using an Array.

Insertion in Queue:

Algorithm: ENQUEUE(Queue, MAXSIZE, FRONT, REAR, COUNT, ITEM)
 This algorithm inserts an element ITEM into a circular queue.

1. [Queue already filled?]
 If COUNT = MAXSIZE then: [COUNT is number of values in the Queue]
 Write: OVERFLOW, and Return.
2. [Find new value of REAR.]
 If COUNT = 0, then: [Queue initially empty.]
 Set FRONT = 0 and REAR = 0
 Else: if REAR = MAXSIZE - 1, then:
 Set REAR = 0
 Else:
 Set REAR = REAR + 1.
 [End of If Structure.]
3. Set Queue[REAR] = ITEM. [This insert new element.]
4. COUNT = COUNT + 1 [Increment to Counter.]
5. Return.

Deletion in Queue:

Algorithm: DEQUEUE(Queue, MAXSIZE, FRONT, REAR, COUNT, ITEM)
 This procedure deletes an element from a queue and assigns it to the variable ITEM.

1. [Queue already empty?]
 If COUNT = 0, then: Write: UNDERFLOW, and Return -1.
2. Set ITEM = Queue[FRONT].
3. Set COUNT = COUNT - 1
4. [Find new value of FRONT.]
 If COUNT = 0, then: [There was one element and has been deleted]
 Set FRONT = -1, and REAR = -1.
 Else if FRONT = MAXSIZE - 1, then: [Circular, so set Front = 0]
 Set FRONT = 0
 Else:
 Set FRONT = FRONT + 1.
 [End of If structure.]
5. Return ITEM

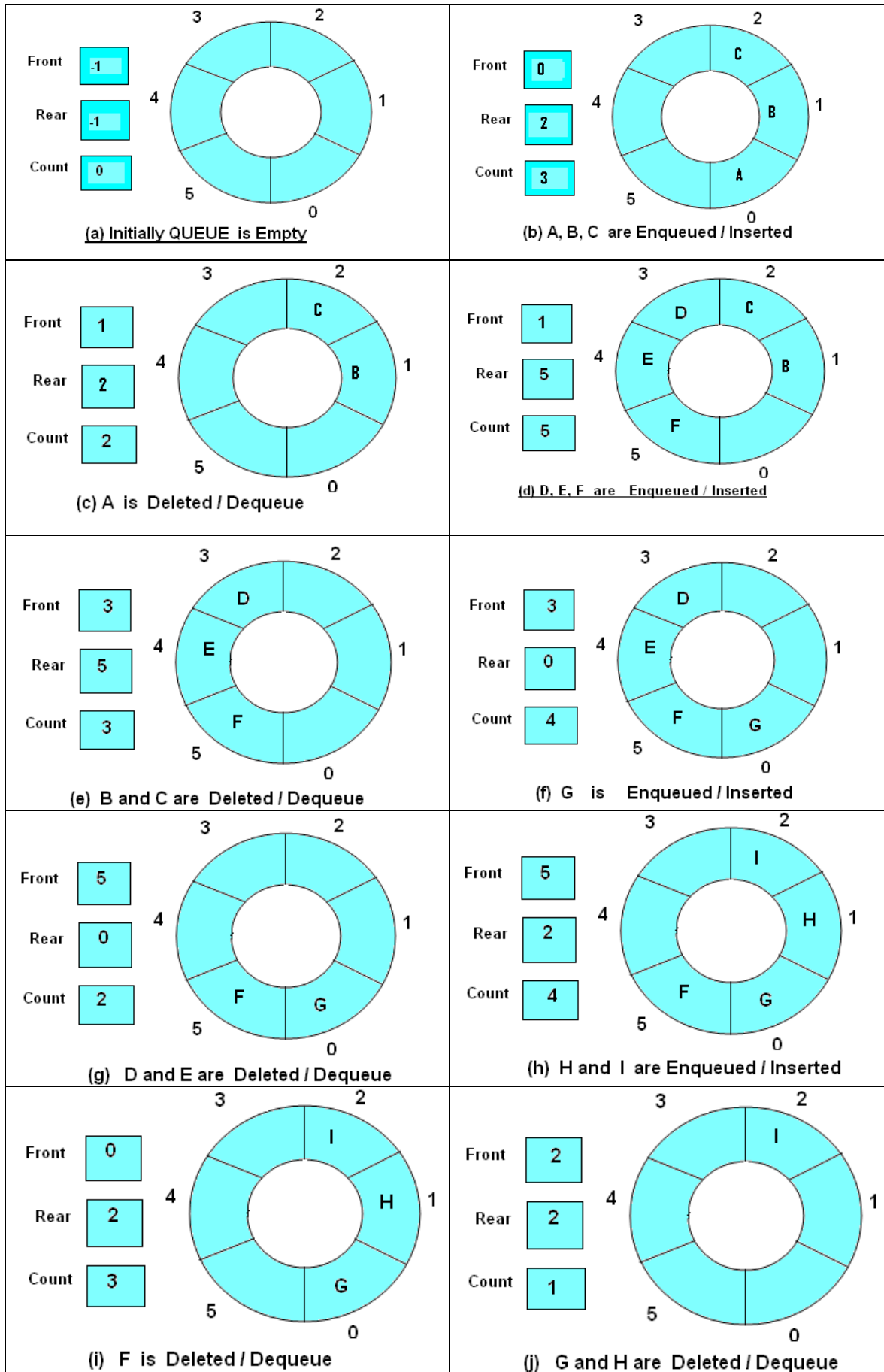
D A T A S T R U C T U R E S

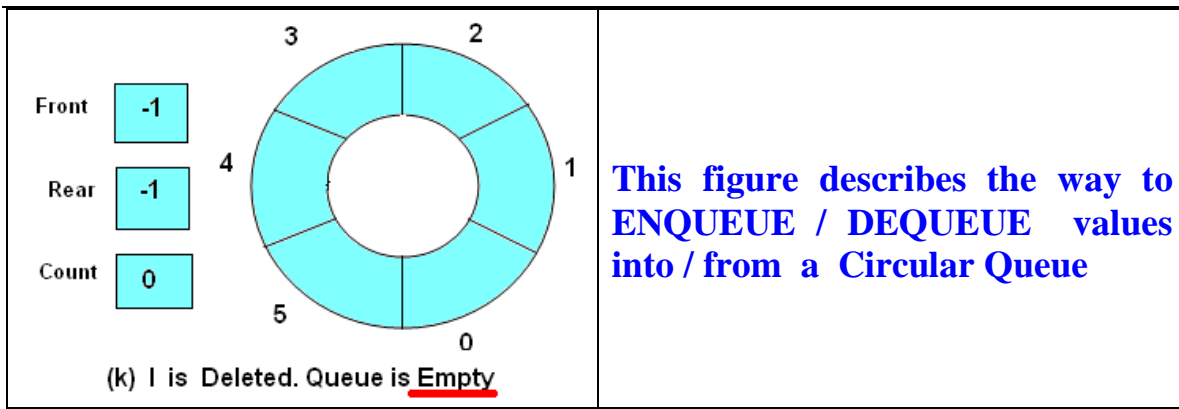
Following Figure shows that how a queue may be maintained by a circular array with **MAXSIZE = 6** (Six memory locations). Observe that queue always occupies consecutive locations except when it occupies locations at the beginning and at the end of the array. If the queue is viewed as a circular array, this means that it still occupies consecutive locations. Also, as indicated by **Fig(k)**, the queue will be empty only when **Count = 0** or (**Front = Rear** but not null) and an element is deleted. For this reason, **-1 (null)** is assigned to **Front** and **Rear**.

MaxSize = 6

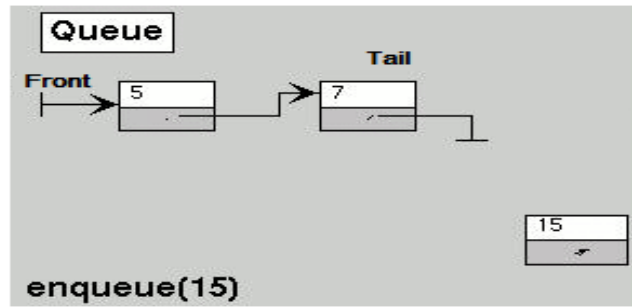
(a) Initially QUEUE is Empty	Front = -1 Rear = -1 Count = 0	<table><tr><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table>							0	1	2	3	4	5
0	1	2	3	4	5									
(b) A, B, C are Enqueued / Inserted	Front = 0 Rear = 2 Count = 3	<table><tr><td>A</td><td>B</td><td>C</td><td></td><td></td><td></td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table>	A	B	C				0	1	2	3	4	5
A	B	C												
0	1	2	3	4	5									
(c) A is Deleted / Dequeue	Front = 1 Rear = 2 Count = 2	<table><tr><td></td><td>B</td><td>C</td><td></td><td></td><td></td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table>		B	C				0	1	2	3	4	5
	B	C												
0	1	2	3	4	5									
(d) D, E, F are Enqueued / Inserted	Front = 1 Rear = 5 Count = 5	<table><tr><td></td><td>B</td><td>C</td><td>D</td><td>E</td><td>F</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table>		B	C	D	E	F	0	1	2	3	4	5
	B	C	D	E	F									
0	1	2	3	4	5									
(e) B and C are Deleted / Dequeue	Front = 3 Rear = 5 Count = 3	<table><tr><td></td><td></td><td></td><td>D</td><td>E</td><td>F</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table>				D	E	F	0	1	2	3	4	5
			D	E	F									
0	1	2	3	4	5									
(f) G is Enqueued / Inserted	Front = 3 Rear = 0 Count = 4	<table><tr><td>G</td><td></td><td></td><td>D</td><td>E</td><td>F</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table>	G			D	E	F	0	1	2	3	4	5
G			D	E	F									
0	1	2	3	4	5									
(g) D and E are Deleted / Dequeue	Front = 5 Rear = 0 Count = 2	<table><tr><td>G</td><td></td><td></td><td></td><td></td><td>F</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table>	G					F	0	1	2	3	4	5
G					F									
0	1	2	3	4	5									
(h) H and I are Enqueued / Inserted	Front = 5 Rear = 2 Count = 4	<table><tr><td>G</td><td>H</td><td>I</td><td></td><td></td><td>F</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table>	G	H	I			F	0	1	2	3	4	5
G	H	I			F									
0	1	2	3	4	5									
(i) F is Deleted / Dequeue	Front = 0 Rear = 2 Count = 3	<table><tr><td>G</td><td>H</td><td>I</td><td></td><td></td><td></td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table>	G	H	I				0	1	2	3	4	5
G	H	I												
0	1	2	3	4	5									
(j) G and H are Deleted / Dequeue	Front = 2 Rear = 2 Count = 1	<table><tr><td></td><td></td><td>I</td><td></td><td></td><td></td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table>			I				0	1	2	3	4	5
		I												
0	1	2	3	4	5									
(k) I is Deleted. Queue is <i>Empty</i>	Front = -1 Rear = -1 Count = 0	<table><tr><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table>							0	1	2	3	4	5
0	1	2	3	4	5									

Another way to Represent Circular QUEUE

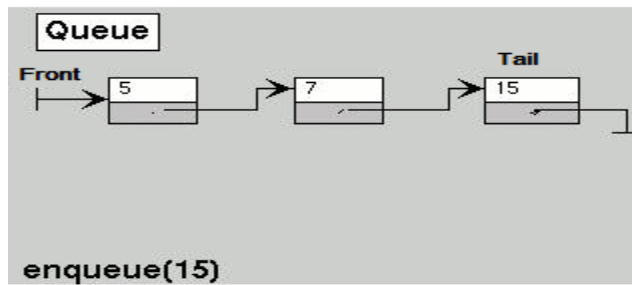




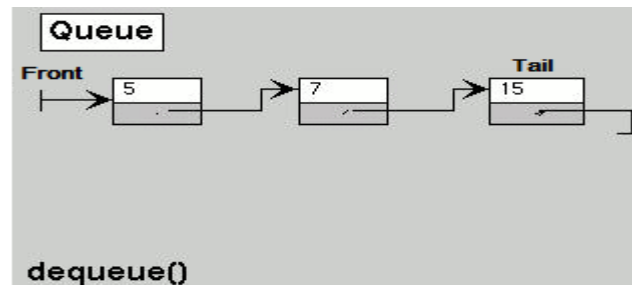
Following is the graphical presentation of Queue using Linked Structures



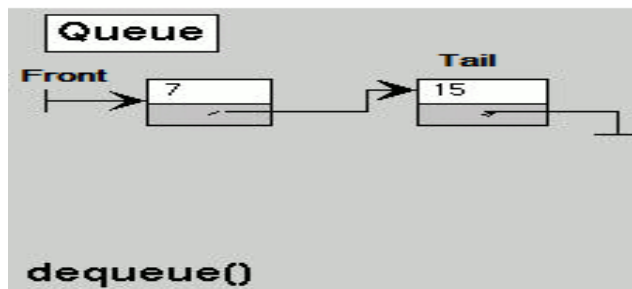
Before inserting a new element



After inserting element 15



Before removing the first element



After removing the first element

// c-language code to implement Circular QUEUE using array

```
#include<iostream>
using namespace std;
#define MAXSIZE 10

// Global items declaration
int Queue[MAXSIZE];
int front = -1;
int rear = -1;
int count =0;

bool IsEmpty(){if(count==0)return true; else return false; }

bool IsFull() { if( count== MAXSIZE) return true; else return false;}

void Enqueue(int ITEM)
{ if(IsFull()) { cout<<"\n QUEUE is full\n"; return;}

    if(count == 0) rear = front= 0; // first item to enqueue
    else if(rear == MAXSIZE -1) rear=0 ; // Circular, rear set to zero
    else rear++; // otherwise increase rear by one

    Queue[rear]=ITEM;
    count++;
}

int Dequeue()
{
    if(IsEmpty()) return -1;
    int ITEM= Queue[front];
    count--;
    if(count == 0 ) front = rear = -1;
    else if(front == MAXSIZE -1) front=0;
    else front++;
    return ITEM;
}

void Traverse()
{ if(IsEmpty()) { cout<<"\n\n QUEUE is empty\n"; return; }
    int i = front;
    while(1)
    { cout<<Queue[i]<<"\t";
        if (i == rear) break;
        else if(i == MAXSIZE -1) i = 0;
        else i++;
    }
}

int main()
{
    int choice,ITEM, loop=1;
    while(loop)
    {
        cout<<"\n\n\n\n\t\t\t\t\t QUEUE Operation\n\n";
        cout<<"\t 1-insert value \n\n\t 2-deleted value\n\n";
        cout<<"\t 3-Traverse QUEUE \n\n\t 4-exit\n\n\n\n";
        cout<<"\t\t\t\t\t your choice: "; cin>>choice;
    }
}
```

```

switch(choice)
{
    case 1:
        cout<<"\n put a value: ";
        cin>>ITEM;
        Enqueue(ITEM);
        break;
    case 2:
        ITEM=Dequeue();
        if(ITEM == -1) cout<<"\n\n QUEUE is empty\n";
        else
            cout<<ITEM<<" has been deleted\n";
        break;
    case 3:
        cout<<"\n Queue Traverse\n";
        Traverse(); break;

        case 4:      loop=0; break;
} // End of Switch
} // End of while
return 0;
} // End of main()

```

**// A Program that exercise the operations on QUEUE
// using POINTER (Dynamic Binding)**

```

#include <iostream>
using namespace std;

struct QUEUE
{ int val;
  QUEUE *pNext;
};

QUEUE *rear=NULL, *front=NULL;

void Enqueue(int);
int Dequeue(void);
void Traverse(void);

int main(void)
{ int ITEM, choice, loop=1;
  while( loop )
  {
      cout<<"\n\n\n\n      ***** QUEUE UNSING POINTERS ***** \n";
      cout<<"\n\n\t ( 1 ) Enqueue \n\t ( 2 ) Dequeue \n";
      cout<<"\t ( 3 ) Print queue \n\t ( 4 ) Exit.";
      cout<<" \n\n\n\t Your choice ---> ";
      cin>>choice;
      switch(choice)
      {
          case 1:
              cout<<"\n Enter a number: ";
              cin>>ITEM;
              Enqueue(ITEM);
              break;
          case 2:
              ITEM = Dequeue();
              if(ITEM) cout<<" \n Deleted from Q = "<<ITEM<<endl;
              break;
      }
  }
}

```

```

        case 3:
            Traverse();
            break;

    case 4:
        loop=0;
        break;

    default:
        printf("\n\n\t Invalid Choice: \n");
    } // end of switch block
} // end of while loop
} // end of of main() function

void Enqueue (int ITEM)
{
    struct QUEUE *NewNode = new QUEUE;

    NewNode->val = ITEM;
    NewNode->pNext = NULL;

    if (rear == NULL)
        front = rear= NewNode;
    else
    {
        rear->pNext = NewNode;
        rear = NewNode;
    }
}

int Dequeue(void)
{ if(front == NULL) { cout<<" \n <Underflow>  QUEUE is empty\n";
                    return 0;
                    }
  QUEUE *T = front;
  int ITEM = front->val;
  if(front == rear ) front=rear=NULL;
  else front = front-> pNext;
  delete T;  return  ITEM;
}

void Traverse(void)

{ if(front == NULL) {cout<<" \n <Underflow>  QUEUE is empty\n";
                    return; }
  QUEUE *F = front;
  do
  { cout<< F->val << "\t";
    F=F->pNext;
  } while(F != rear);
  cout<< F->val; // last value when F == rear
}

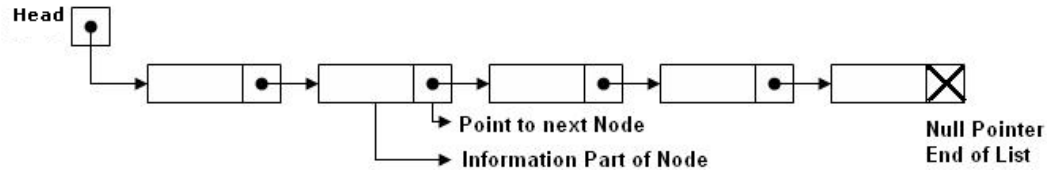
```

Linked List:

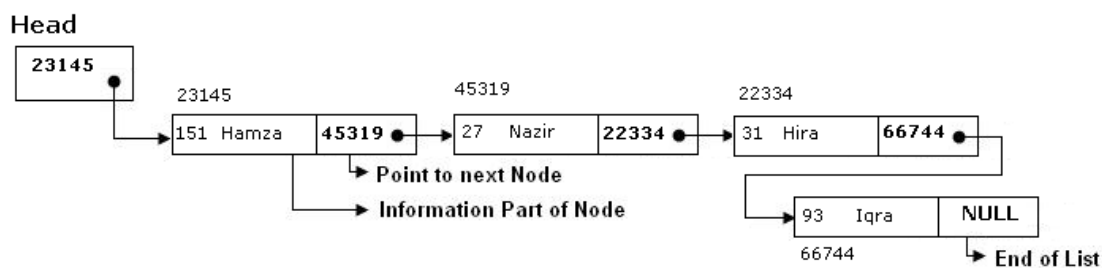
A linked list or one way list is a linear collection of data elements, called **nodes**, where the linear order is given by means of "**pointers**". Each node is divided into two parts.

- The first part contains the information of the element.
- The second part called the link field contains the address of the next node in the list.

To see this more clearly lets look at an example:



For example:



The **Head** is a special pointer variable which contains the address of the first node of the list. If there is no node available in the list then **Head** contains **NULL** value that means, List is empty. The left part of the each node represents the information part of the node, which may contain an entire record of data (e.g. ID, name, marks, age etc). the right part represents pointer/link to the next node. The next pointer of the last node is **null** pointer signal the end of the list.

Advantages:

List of data can be stored in arrays but linked structures (pointers) provide several advantages.

A linked list is appropriate when the number of data elements to be represented in data structure is unpredictable. It also appropriate when there are frequently insertions & deletions occurred in the list. Linked lists are dynamic, so the length of a list can increase or decrease as necessary.

Operations on Linked List:

There are several operations associated with linked list i.e.

a) Traversing a Linked List

Suppose we want to traverse LIST in order to process each node exactly once. The traversing algorithm uses a pointer variable PTR which points to the node that is currently being processed. Accordingly, PTR->NEXT points to the next node to be processed so,

PTR=HEAD [Moves the pointer to the first node of the list]
 PTR=PTR->NEXT [Moves the pointer to the next node in the list.]



Algorithm: (Traversing a Linked List) Let LIST be a linked list in memory. This algorithm traverses LIST, applying an operation PROCESS to each element of list. The variable PTR points to the node currently being processed.

b) Searching a Linked List:

Let list be a linked list in the memory and a specific ITEM of information is given to search. If ITEM is actually a key value and we are searching through a LIST for the record containing ITEM, then ITEM can appear only once in the LIST.

Search for wanted ITEM in List can be performed by traversing the list using a pointer variable PTR and comparing ITEM with the contents PTR->INFO of each node, one by one of list.

Algorithm: SEARCH(INFO, NEXT, HEAD, ITEM)

LIST is a linked list in the memory. This algorithm finds the location LOC of the node where ITEM first appear in LIST, otherwise sets LOC=NULL.

1. Set PTR=HEAD.
2. Repeat Step 3 while PTR≠NULL:
3. if ITEM = PTR->INFO then:
 Set LOC=PTR, and return LOC. [Search is successful.]
- 3A. Set PTR = PTR->Next
 [End of If structure.]
- [End of Step 2 loop.]
4. Set LOC=NULL, and return LOC. [Search is unsuccessful.]
5. Exit.

Search Linked List for insertion and deletion of Nodes:

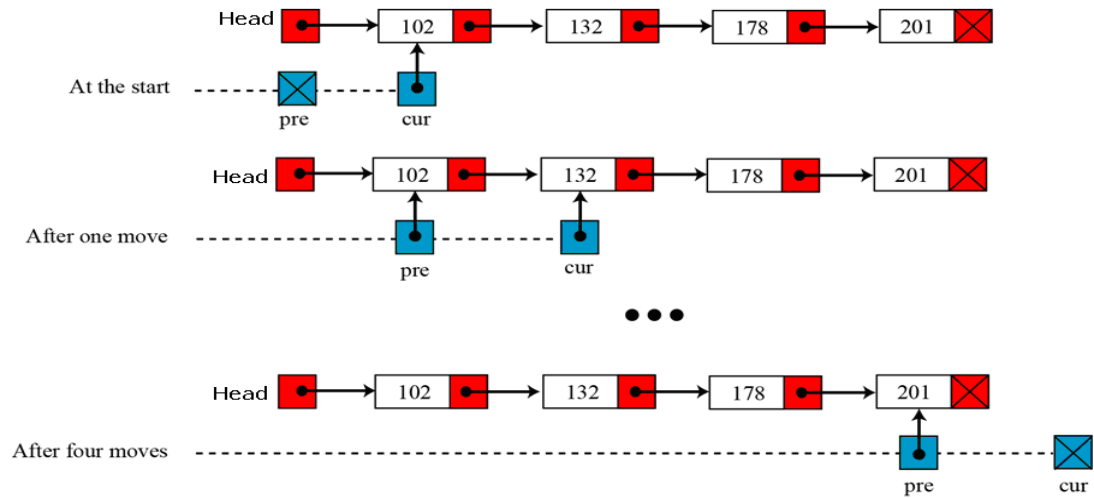
Both insertion and deletion operations need searching the linked list.

- To add a new node, we must identify the logical predecessor (address of previous node) where the new node is to be inserting.
- To delete a node, we must identify the location (addresses) of the node to be deleted and its logical predecessor (previous node).

Basic Search Concept

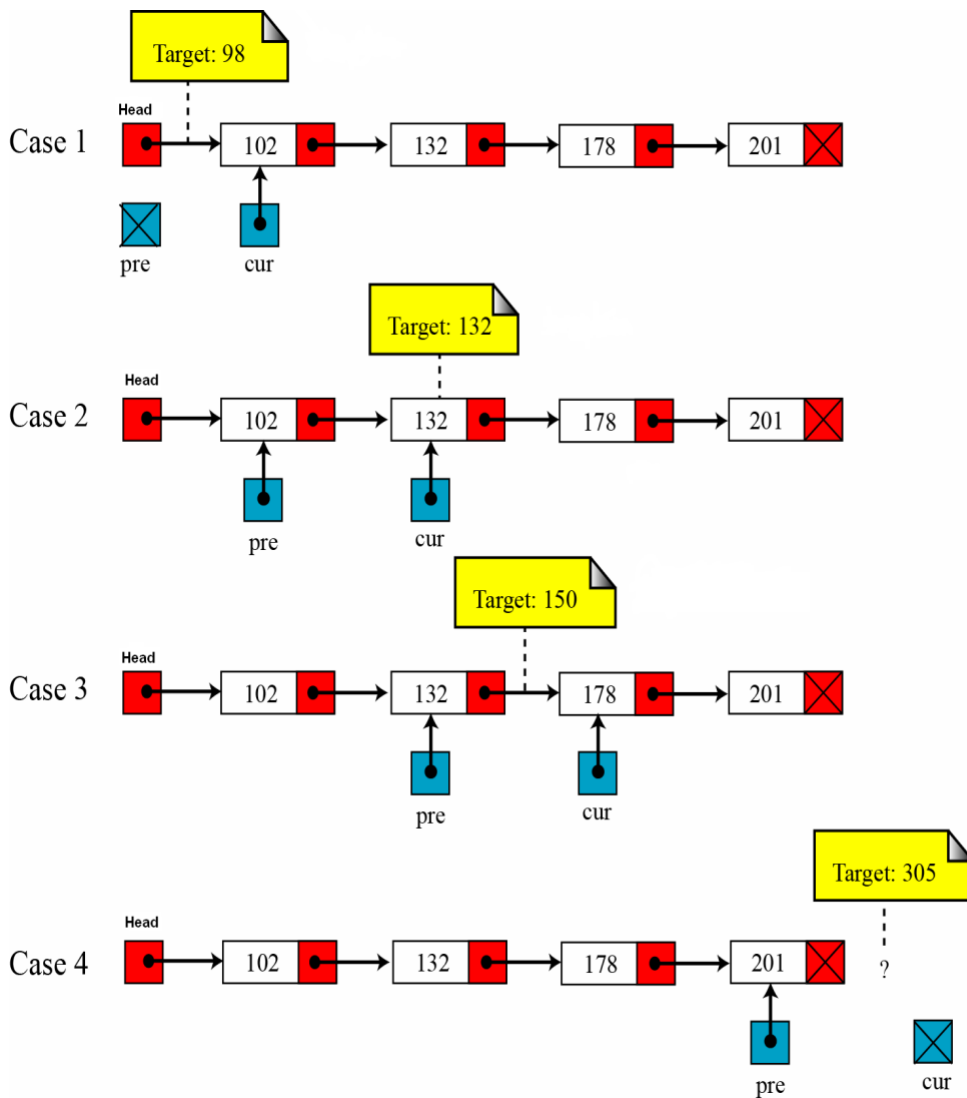
Assume there is a sorted linked list and we wish that after each insertion/deletion this list should always be sorted. Given a target value, the search attempts to locate the requested node in the linked list.

Since nodes in a linked list have no names, we use two pointers, **pre** (for previous) and **cur** (for current) nodes. At the beginning of the search, the **pre** pointer is **null** and the **cur** pointer points to the first node (**Head**). The search algorithm moves the two pointers together towards the end of the list. Following Figure shows the movement of these two pointers through the list in an extreme case scenario: when the target value is larger than any value in the list.



Moving of **pre** and **cur** pointers in searching a linked list

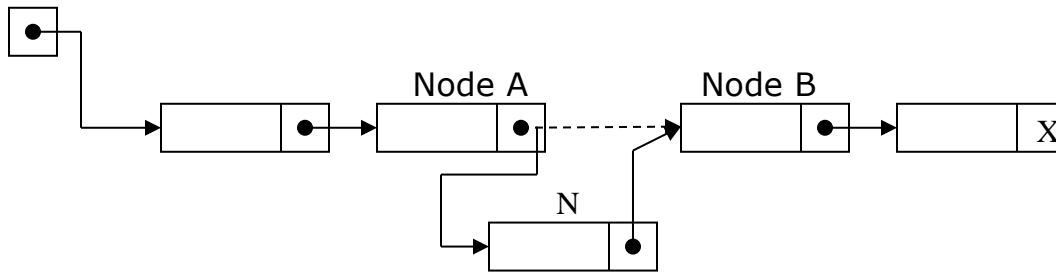
Values of **pre** and **cur** pointers in different cases



Insertion into a Linked List:

If a node N is to be inserted into the list between nodes **A** and **B** in a linked list named LIST. Its schematic diagram would be;

Head



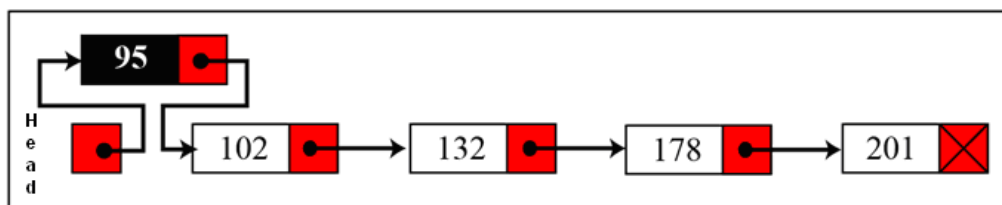
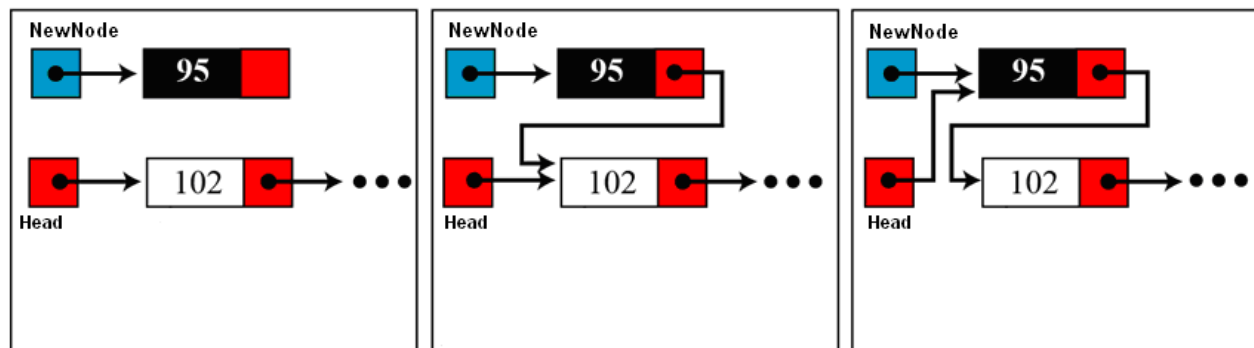
Inserting at the Beginning of a List:

If the linked list is sorted list and new node has the least low value already stored in the list i.e. (**if $New \rightarrow info < Head \rightarrow info$**) then new node is inserted at the beginning / Top of the list.

If(pre == NULL) { **NewNode** ->Next =Head; **Head**=**NewNode**;}

if (**NewNode** -> info < **Head**-> info) **NewNode** -> next = **Head**

Head = **NewNode**



Result

Inserting a new node in list:

The following algorithm inserts an ITEM into LIST.

Algorithm: INSERT(ITEM)

[This algorithm add newnodes at any position (Top, in Middle and at End) in the List]

1. Create a **NewNode** node in memory
2. Set **NewNode** -> INFO =ITEM. [Copies new data into INFO of new node.]
3. Set **NewNode** -> NEXT = NULL. [Copies NULL in NEXT of new node.]
4. If **HEAD**=NULL, then **HEAD**=**NewNode** and return. [Add first node in list]
5. if **NewNode**-> INFO < **HEAD**->INFO
 then Set **NewNode**->NEXT=**HEAD** and **HEAD**=**NewNode** and return
 [Add node on top of existing list]
6. PrevNode = NULL, CurrNode=NULL;
7. for(Curr =**HEAD**; Curr != NULL; Curr = Curr->NEXT)
 { if(NewNode->INFO <= Curr->INFO)
 {
 Return Curr and Prev
 and break the loop
 }
 Prev = Curr;
 } [end of loop]
 [Insert after PREV node (in middle or at end) of the list]
8. Set **NewNode**->NEXT = Prev->NEXT and
9. Set Prev->NEXT= **NewNode**.
10. Exit.

Delete a node from list:

The following algorithm deletes a node from any position in the LIST.

Algorithm: DELETE(ITEM)

LIST is a linked list in the memory. This algorithm deletes the node where ITEM first appear in LIST, otherwise it writes "NOT FOUND"

1. if **Head** =NULL then write: "Empty List" and return [Check for Empty List]
2. if ITEM = **Head** -> info then: [Top node is to delete]
 Set **Head** = **Head** -> next and return
3. Set PrevNode = NULL, CurrNode=NULL, Scan=NULL;
4. for(Scan =**HEAD**; Scan != NULL; Scan = Scan->NEXT)
 { if (ITEM = Scan->INFO) then:
 {
 Set CurrNode = Scan
 and break the loop
 }
 Set PrevNode = Scan;
 } [end of loop]
 [delete the current node from the list]
 Set PrevNode ->NEXT = CurrNode->NEXT
5. Exit.

```
// A Program that exercise the operations on Liked List
// Programed by SHAHID LONE
#include<iostream>
using namespace std;

struct node
{
    int info;
    struct node *next;
};

struct node *Head=NULL;

struct node *Prev,*Curr;

void AddNode(int ITEM)
{
    struct node *NewNode = new node;
    NewNode->info=ITEM;  NewNode->next=NULL;

    if(Head==NULL) { Head=NewNode; return; }
    if(NewNode->info < Head->info)
        { NewNode->next = Head; Head=NewNode; return;}

    for(Curr = Head ; Curr != NULL ; Curr = Curr->next)
    {
        if( NewNode->info < Curr->info) break;
        Prev = Curr;
    }

    NewNode->next = Prev->next;
    Prev->next = NewNode;
} // end of AddNode function

void DeleteNode()
{
    int inf;
    if(Head==NULL){ cout<<"\n\n empty linked list\n"; return;}

    cout<<"\n Put the info to delete: ";
    cin>>inf;

    if(inf == Head->info) // First top node to delete
        { Head = Head->next; return;}

    Prev = Curr = Scan = NULL;
    for( Curr = Head ; Curr != NULL ; Curr = Curr->next )
        { if( Curr->info == inf) break;
          Prev = Curr;
        }

    if( Curr->info != inf)
        cout<<"\n "<<inf<<" not found in list\n";
        else
            { Prev->next = Curr->next; }

} // end of DeleteNode function
```

```

void Traverse()
{
    for( Curr = Head; Curr != NULL ; Curr = Curr->next )

        cout<< Curr->info<<" , \t";
} // end of Traverse function

int main()
{ int inf, ch, loop=1;
  while(loop)
  { cout<<" \n\n\n Linked List Operations\n\n";
    cout<<" 1- Add Node \n 2- Delete Node \n";
    cout<<" 3- Traverse List \n 4- exit\n";
    cout<<"\n\n Your Choice: ";
    cin>>ch;
    switch(ch)
    { case 1: cout<<"\n Put info/value to Add: ";
              cin>>inf;
              AddNode(inf);
              break;

              case 2: DeleteNode(); break;

              case 3: cout<<"\n Linked List Values:\n";
                      Traverse(); break;
              case 4: loop=0; break;
            } // end of switch
    } // end of while loop
  return 0;
} // end of main ( ) function

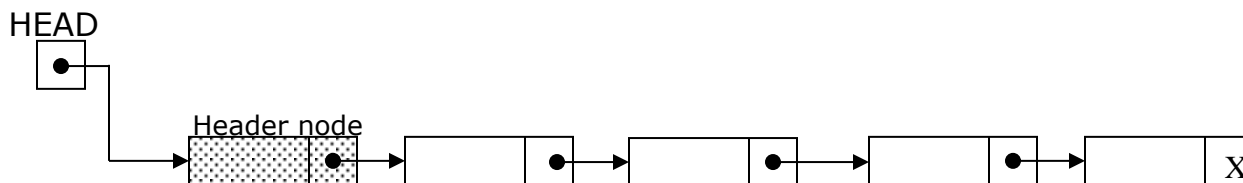
```

Header Linked Lists:

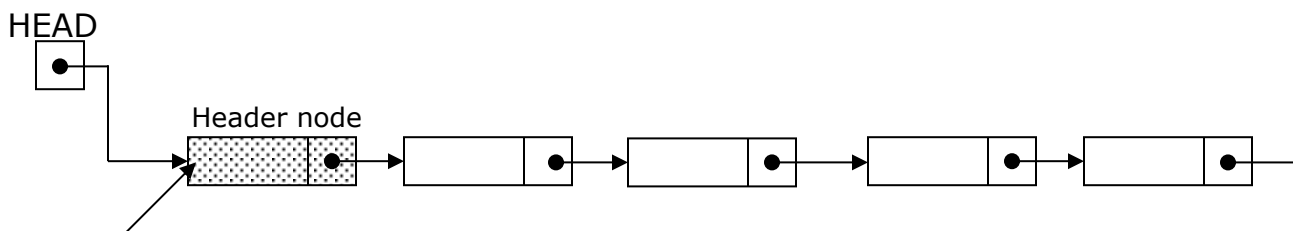
A *header linked list* is a linked list which always contains a special node, called the *header node*, at the beginning of the list.

The following are two kinds of widely used header lists:

- ❖ A *Grounded header list* is a header list where the last node contains the null pointers (the term '*grounded*' comes from the fact that texts use the electrical ground symbol to indicate the null pointer.)
- ❖ A *Circular header list* is a header list where the last node points back to the header node.



Grounded Linked List



Circular Linked List

Circular linked list are frequently used instead of ordinary linked lists because many operation are much easier to state and implement using header lists. This comes from the following two properties of circular header list.

- The null pointers are not used and hence all pointers contain valid addresses.
- Every (ordinary) node has a predecessor so the first node may not require a special case.

a) Traversing a Circular Header List:

The following algorithm shows how a circular header list may be traversed.

Algorithm: (Traversing a circular header list)
 Let LIST be a circular header list in memory. This algorithm traverses LIST, applying a operation PROCESS in each node of LIST.

1. Set CURR=HEAD->Next [Initializes the pointer CURR.]
2. Repeat Step 3 and 4 while CURR ≠ HEAD:
3. Apply PROCESS to CURR->INFO
4. Set CURR=CURR->Next. [CURR now points to the next node.]
- [End of Step 2 loop.]
5. Exit.

b) Searching a Circular Header List:

The following algorithm finds the location LOC of the first node where ITEM appears in the LIST (which is an ordinary circular header linked list)

Algorithm: SRCHHL(INFO, NEXT, HEAD, ITEM, LOC)
 LIST is a circular header list in memory. This algorithm finds the location LOC of the node where ITEM first appear in LIST or sets LOC=NULL.

1. Set CURR = HEAD->Next
2. Repeat while CURR->INFO ≠ ITEM and CURR ≠ HEAD.
 Set CURR = CURR->Next [CURR now points to the next node.]
 [End of loop.]
3. If CURR->INFO = ITEM, then:
 Set LOC = CURR.
 Else:
 Set LOC = NULL.
 [End of IF Structure.]
4. Return LOC and EXIT.

c) Deleting a node in Circular Header List:

The following algorithm finds the location of node N which contain ITEM (to be deleted) and also the location (PREV) of the preceding node N, to delete an ITEM when LIST is a circular header linked list.

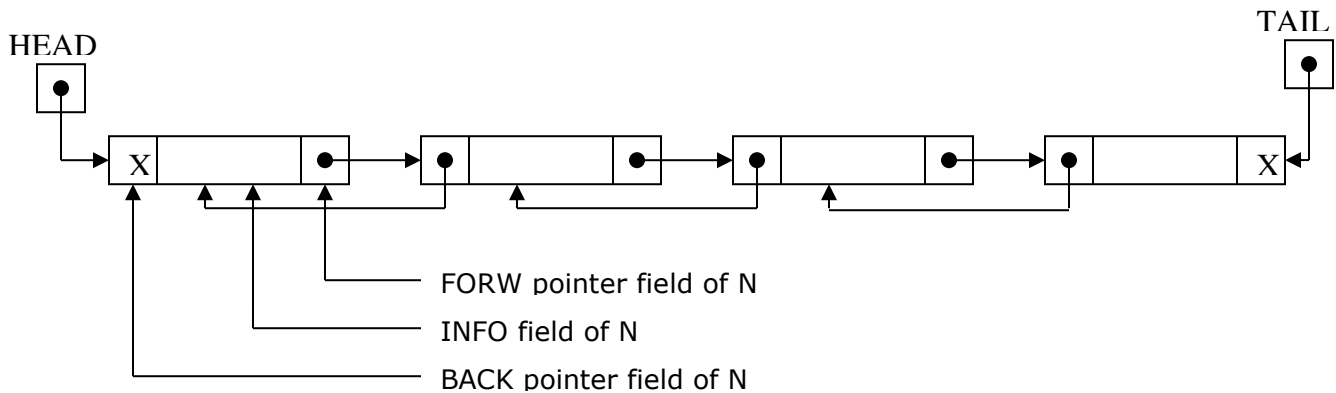
Algorithm: DELL_HL (HEAD, ITEM, CURR, PREV)

1. Set PREV = HEAD and CURR = HEAD->Next [Initializes pointers.]
2. Repeat While CURR->INFO ≠ ITEM and CURR ≠ HEAD.
 Set PREV = CURR and CURR = CURR->Next [Updates pointers.]
 [End of loop.]
3. If CURR->INFO = ITEM, then:
 Set PREV->Next = CURR->Next
 Else:
 Write: "Item not in the list"
 [End of If Structure.]
4. Exit.

TWO WAY LISTS / DOUBLY LINK LIST:

A *two way list* is a list structure which can be traversed in two directions: in the usual forward direction from the beginning of the list to the end, or in the backward direction from the end of the list to the beginning. Any node N has immediate access to both the next node and the previous node in the list. Hence one is able to delete a node N from the list without traversing any part of the list. A two way list is a linear collection of data element called node where each node is divided into three parts.

- An information field INFO which contain the data of N.
- A pointer field FORW which contains the location of the next node in the list.
- A pointer field BACK which contains the location of the preceding node in the list.



Struct student
{

Int id;
Char name[20];
Float gpa;

Student *Back;
Student *FORW;

}

for (Curr = Head ; Curr != NULL; Curr = Curr ->FORW)

{

Apply process to curr->info;

}

for (Curr = Tail ; Curr != NULL; Curr = Curr ->BACK)

{

Apply process to curr->info;

}

Traversing:

Using the variable HEAD and the pointer field FORW, we can traverse a two way list in the forward direction. On the other hand using the variable TAIL and the pointer field BACK, we can traverse in the backward direction.

Two Way lists in memory:

Two way lists may be maintain in memory by means of pointer in the same way as one way lists except that now we require two pointers, FORW and BACK, instead of one i.e. NEXT, and two list pointer variables HEAD and TAIL.

Two way Circular Header Lists:

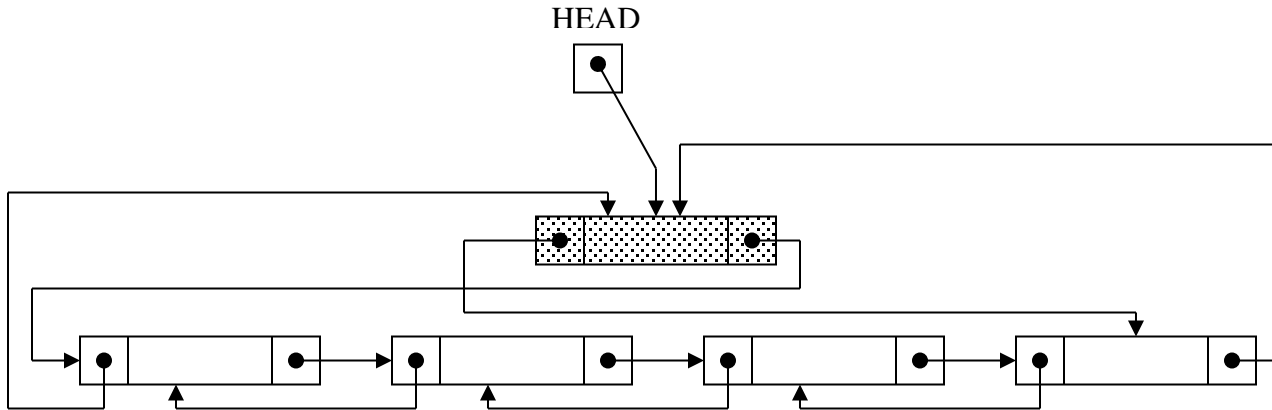
A two way circular header list consists of a header node consisting of a forward (FORW) pointer and a back (BACK) pointer pointing towards the two end of the LIST.

At requires only one list pointer variable HEAD which points to the header node.

The last end of the list points back to the header node maintains the two way circular header lists.

Operation on Two Way Lists:

Let LIST be two way lists in memory.



a) Traversing:

To traverse a LIST in order to process each node exactly once, then the simple algorithm for traversing a circular header list is used as discussed before.

The following algorithms shows how a two-way circular header linked list may be traversed.

Algorithm: (Traversing two-way circular header linked list in **Forward Direction**)

1. Set CURR=HEAD->FORW [Initializes the pointer CURR.]
2. Repeat Step 3 and 4 while CURR ≠ HEAD
3. Apply PROCESS to CURR->INFO
4. Set CURR=CURR->FORW. [CURR now points to the next node.]
- [End of Step 2 loop.]
5. Exit.

Algorithm: (Traversing two-way Circular Header linked list in **Backward Direction**)

1. Set CURR=HEAD->BACK [Initializes the pointer CURR.]
2. Repeat Step 3 and 4 while CURR ≠ HEAD
3. Apply PROCESS to CURR->INFO
4. Set CURR=CURR->BACK. [CURR now points to the next node.]
- [End of Step 2 loop.]
5. Exit.

b) Searching:

If we are given an ITEM of information a key value and its location LOC is to be found in Two-Way Header Linked LIST. Similar algorithm for traversing a circular header linked list and then comparing the given ITEM with each node and giving the location LOC of the ITEM, are used. In this case, another advantage is that we can traverse the list in the backward direction also.

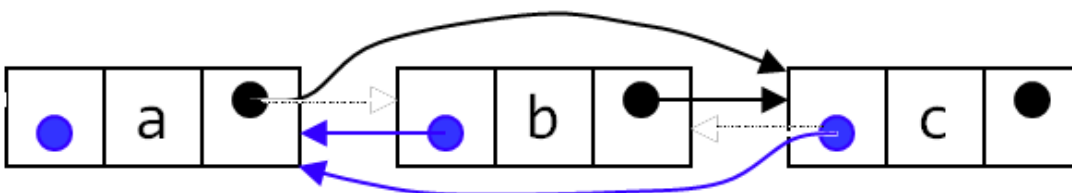
c) Deleting:

If we are given the location LOC of a node N in LIST, and we want to delete it from the list, N is deleted from the list by changing the value of the forward (FORW) pointer of its previous node and back (BACK) pointer of its next node.

Algorithm: DELTWL(INFO, FORW, BACK, HEAD, ITEM)

1. If HEAD ->FORW = HEAD and HEAD->BACK = HEAD then:
Write: "List is Empty" and return
2. Get ITEM from user for which Node is to delete
3. Repeat for(CURR = HEAD->FORW ; CURR != HEAD ; CURR = CURR-> FORW)
 - { if (CURR->INFO = ITEM) then:
 - {if(CURR -> BACK = HEAD && CURR->FORW = HEAD) then:
 - {HEAD -> BACK = HEAD
 - HEAD -> FORW = HEAD
 - return;
 - }
 - else
 - { PREV = CURR -> BACK
 - POST = CURR -> FORW
 - POST -> BACK = PREV
 - PREV -> FORW = POST
 - return
 - }
4. Write: "Node not found."
5. return
6. Exit.

Deletion a Node from Doubly Linked List

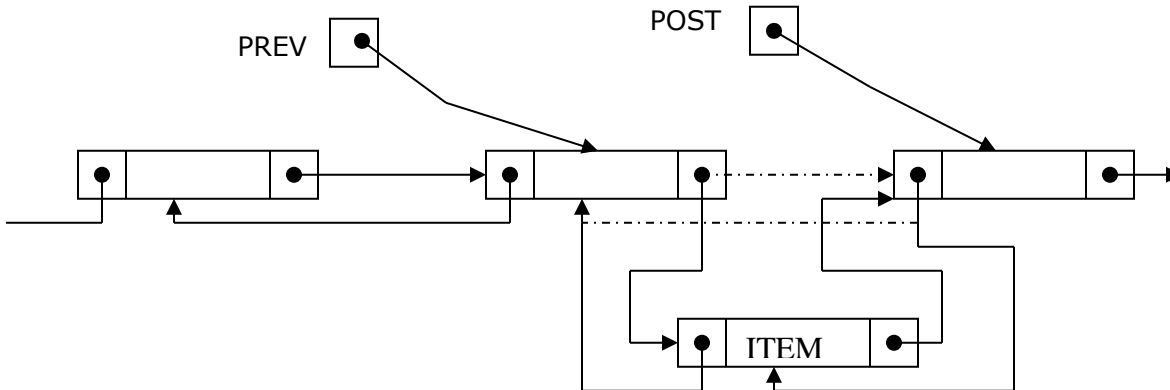


d) Inserting:

If we are given the location PREV and POST of adjacent nodes A and B in LIST, and an ITEM of information between nodes A and B.

Now, the node NEWNODE with contents ITEM is inserted into the list by changing the following four pointers

- The FORW pointer of the NEWNODE with FORW pointer of PREV
- The BACK pointer of the NEWNODE with PREV
- The FORW pointer of the PREV with NEWNODE
- The BACK pointer of the POST with NEWNODE



Algorithm: INSTWL(INFO, FORW, BACK, START, AVAIL, LOCA, LOCB, ITEM)

1. Create a NEWNODE in Memory
2. Furnish the Data for NEWNODE
3. [Insert node into list.]
 - if(HEAD->BACK equal to HEAD) [This will be the first node to add]
 - I. set HEAD->BACK= NEWNODE
 - II. set HEAD->FORW= NEWNODE
 - III. set NEWNODE ->FORW = HEAD
 - IV. set NEWNODE ->BACK=HEAD
 - V. return
4. Repeat for(CURR = HEAD->FORW ; CURR != HEAD ; CURR = CURR->FORW)
 - I. if(CURR->INFO = NEWNODE -> INFO) then Write: "Duplicate Entry" and return
 - II. if(CURR->INFO > NEWNODE ->INFO)
 - { PREV = CURR->BACK;
 - POST = PREV->FORW; // or POST = CURR
 - break the loop
 - }
 - else
 - { POST = CURR->FORW;
 - PREV = CURR;
 - }

[end of loop]
5. NEWNODE -> FORW = POST;
6. PREV -> FORW = CURR-> BACK = NEWNODE
7. NEWNODE -> BACK = PREV;
8. exit

C++ Code (2-WAY HEADER LINKED LIST):

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<iomanip.h>
struct student
{
    int ID;
    char name[15];
    int m1,m2,m3;
    struct student *back;
    struct student *forw;
};
struct student *HEAD = new student;

void Initilize()
{
    HEAD->back=HEAD;
    HEAD->forw=HEAD;
    HEAD->ID=HEAD->m1=HEAD->m2=HEAD->m3=0;
}

void Finish()
{
    struct student *CURR= HEAD->back, *Temp;

    while (CURR!=HEAD)
    {
        Temp = CURR;
        CURR = CURR->back;
        delete Temp;
    }
    delete CURR; // it deletes header node
}

char FindGrade( float p )
{
    if( p >= 80 ) return 'A';
    else if( p >= 70 ) return 'B';
    else if( p >= 60 ) return 'C';
    else return 'F';
}

void Summary()
{
    system("cls");
    cout<<"\n\n\t\t\t S U M M R Y\n";
    cout<<"\n\t\t Total Students in List   : "<<HEAD->ID;
    cout<<"\n\t\t No. of  A-Grade students  : "<<HEAD->m1;
    cout<<"\n\t\t No. of  B-Grade students  : "<<HEAD->m2;
    cout<<"\n\t\t No. of  C-Grade students  : "<<HEAD->m3;
    cout<<"\n\t\t No. of  F-Grade students  : "
        <<HEAD->ID - (HEAD->m1+HEAD->m2+HEAD->m3);
    cout<<"\n\n\n";
    //getch();
}

void Display(struct student *CURR )
```

```

{char grd;
    int tot=CURR->m1+CURR->m2+CURR->m3;
    float per=tot*100.0f/300.0f;
    grd = FindGrade(per);
    cout<<"\n\t\t Name      : "<<CURR->name;
    cout<<"\n\t\t ID #      : "<<CURR->ID;
    cout<<"\n\t\t Sub #01   : "<<CURR->m1;
    cout<<"\n\t\t Sub #02   : "<<CURR->m2;
    cout<<"\n\t\t Sub #03   : "<<CURR->m3;
    cout<<"\n\t\t Total    : "<<tot;
    cout<<"\n\t\t %age     : "<<per;
    cout<<"\n\t\t Grade    : "<<grd;
}
void add()
{ struct student *newnode = new student;
  system("cls");
  cout<<"\n\n\n\t\t Enter the data for student\n";
  cout<<"\n\tPut the ID # ";cin>>newnode->ID;
  cout<<"\n\tPut the Name : ";cin.ignore();cin.getline(newnode->name,20);
  cout<<"\n\tPut the Marks of three Subjects : ";
  cin>>newnode->m1>>newnode->m2>>newnode->m3;
  float per= (newnode->m1+newnode->m2+newnode->m3)*100/300.0f;
  if(HEAD->back==HEAD) // add first node in the list
  { HEAD->back= HEAD->forw = newnode;
    newnode->forw= newnode->back=HEAD;
    HEAD->ID++;
    if( FindGrade(per) == 'A') HEAD->m1++;
    else if( FindGrade(per) == 'B') HEAD->m2++;
    else if( FindGrade(per) == 'C') HEAD->m3++;
    return;
  }
  struct student *CURR, *POST = NULL, *PREV = NULL;
  for(CURR = HEAD->forw ; CURR != HEAD ; CURR = CURR->forw)
  {   if(CURR->ID == newnode->ID)
      {
          cout<<"\n\n\n\t Duplicate student ID # "
              <<newnode->ID<<" already exist.\n\n";
          getch(); return;
      }
      else
      {
          if(CURR->ID > newnode->ID)
          { PREV = CURR->back;
            POST = PREV->forw; // or POST = CURR
            break; // break the loop
          }
          else
          { POST = CURR->forw;  PREV = CURR; }
      }
  } // end of loop

  newnode->forw = POST;  PREV->forw = CURR->back=newnode;
  newnode->back=PREV;
  HEAD->ID++;
  if( FindGrade(per) == 'A') HEAD->m1++;
  else if( FindGrade(per) == 'B') HEAD->m2++;
  else if( FindGrade(per) == 'C') HEAD->m3++;
} // End of Add() function

```

```

void Traverse()
{if(HEAD->forw==HEAD && HEAD->back==HEAD)
    {cout<<"\n\n\t\t List is Empty..";
     getche();return;
    }
    system("cls");
    struct student *CURR;
    cout<<"\n\n ID#      Name      Sub#01  Sub#02  Sub#03   Total      Per%  Grade\n";
    for(CURR=HEAD->forw;CURR!=HEAD;CURR=CURR->forw)
        { int tot=CURR->m1+CURR->m2+CURR->m3;
          float per=(float)tot*100/300.0f;
          cout <<setw(5)<<CURR->ID<<setw(3)<<" " <<setw(15)
              <<setiosflags(ios::left) <<CURR->name<<setiosflags(ios::right)<<setw(10)
              <<setiosflags(ios::right)<<CURR->m1<<setw(10)<<CURR->m2<<setw(10)
              <<CURR->m3<<setw(10)<<tot<<setw(10)<<setiosflags(ios::fixed)
              <<setiosflags(ios::showpoint)<<setprecision(2)<<per<<setw(3)
              <<FindGrade(per)<<endl;
          }
        getch();
    }

void remove()
{if(HEAD->forw==HEAD && HEAD->back==HEAD)
    {cout<<"\n\n\t\t List is Empty.."; getch();return; }
    struct student *CURR;
    int temp_ID;
    system("cls");
    cout<<"\n\n Enter the ID # "; cin>>temp_ID;
    for(CURR=HEAD->forw;CURR!=HEAD;CURR=CURR->forw)
        { if(CURR->ID==temp_ID)
            { float per= (CURR->m1+CURR->m2+CURR->m3)*100/300.0f;
              HEAD->ID--; // Dcrease one from total students

              if( FindGrade(per) == 'A') HEAD->m1--;
              else if( FindGrade(per) == 'B') HEAD->m2--;
              else if( FindGrade(per) == 'C') HEAD->m3--;
              system("cls");
              cout<<"\n\n Following record has been deleted \n\n";
              Display(CURR);
              if(CURR->back==HEAD && CURR->forw==HEAD)
                  {HEAD->back=HEAD; HEAD->forw=HEAD;
                   Display(CURR);
                   delete CURR;          return;
                  }
              else
                  {struct student *PREV = CURR->back;
                   student *POST = CURR->forw;
                   POST->back = PREV; PREV->forw = POST;
                   delete CURR; return;
                  }
              } // end of outer if
        } // end of loop
    cout<<"\n\n"<<temp_ID<<"\t\t ID NO not found.\n";
    getch();
}

```

```

void search()
{
    if(HEAD->back==HEAD && HEAD->forw==HEAD)
        {cout<<"\n\n\t List is Empty";getche();return;}
    system("cls");
    struct student *CURR;
    int temp_ID;
    cout<<"\n\n\tEnter the ID # ";cin>>temp_ID;
    for(CURR = HEAD->back;CURR!=HEAD;CURR=CURR->back)
        {if(temp_ID==CURR->ID)break;
        }
    if(CURR == HEAD)
        {cout<<"\n\t ID No not found.";getche();return;}
    else
        Display(CURR);
}

void main()
{ Initilize();
  int choice, loop=1;
  while(loop)
      {system("cls");
       cout<<"\n\n\n\n\t\t\t Main Menu\n\n";
       cout<<"\n\t\t[0] -- Summary of Linked List";
       cout<<"\n\t\t[1] -- Add a student";
       cout<<"\n\t\t[2] -- Remove a Record";
       cout<<"\n\t\t[3] -- Traverse";
       cout<<"\n\t\t[4] -- Search ";
       cout<<"\n\t\t[5] -- Exit";
       cout<<"\n\n\t\t Your choice --> ";
       cin>>choice;
       switch(choice)
           {   case 0:
                     Summary();
                     break;
               case 1:
                     add();
                     break;
               case 2:
                     remove();
                     break;
               case 3:
                     Traverse();
                     break;
               case 4:
                     search();
                     break;
               case 5:
                     Finish();
                     loop=0; break;
               default:
                     cout<<"\n\tInvalid Choice." ;
                     getche();
           } // end of switch block

      } // end of loop
} // end of main()

```

Tree:

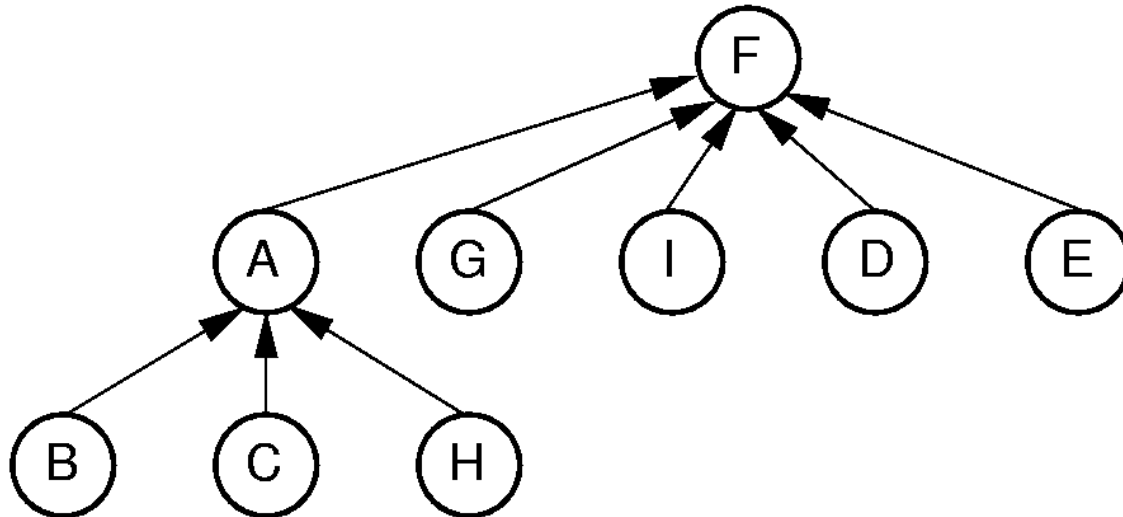
So far, we have been studying mainly linear types of data structures: arrays, lists, stacks and queues. Now we define a nonlinear data structure called **Tree**. This structure is mainly used to represent data containing a hierarchical relationship between nodes/elements e.g. family trees and tables of contents.

There are two main types of tree:

- General Tree
- Binary Tree

General Tree:

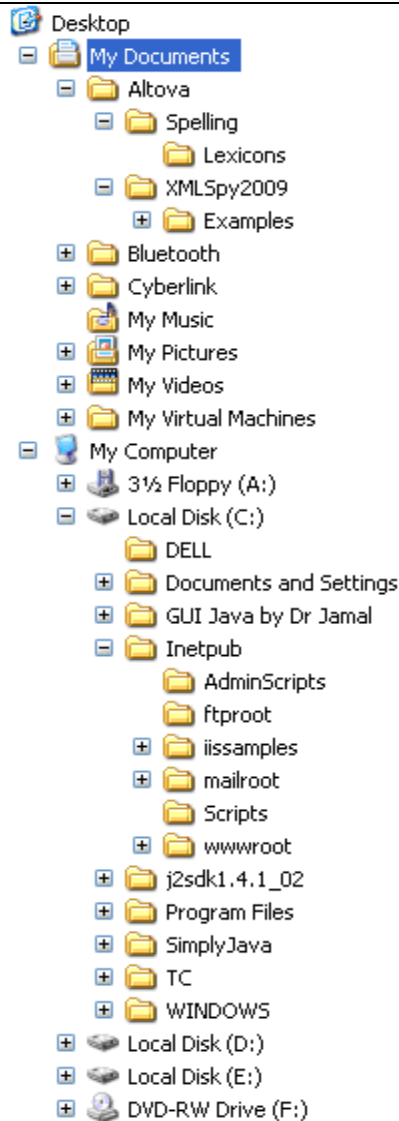
A tree where a node can have any number of children / descendants is called General Tree. For example:



This following figure is a general tree where root is "Visual Programming".

Visual Programming	
1 Introduction to the Visual Studio .NET IDE	30
1.1 Introduction	34
1.2 Visual Studio .NET Integrated Development Environment (IDE) Overview	34
1.3 Menu Bar and Toolbar	37
1.4 Visual Studio .NET Windows	39
1.4.1 Solution Explorer	39
1.4.2 Toolbox	40
1.4.3 Properties Window	42
1.5 Using Help	42
1.6 Simple Program: Displaying Text and an Image	44
2 ASP .NET, Web Forms and Web Controls	48
2.1 Introduction	49
2.2 Simple HTTP Transaction	50
2.3 System Architecture	52
2.4 Creating and Running a Simple Web Form Example	53
2.5 Web Controls	66
2.5.1 Text and Graphics Controls	67
2.5.2 AdRotator Control	71
2.5.2.1 X-axis Control	73
2.5.2.2 Y-axis Control	75
2.5.3 Validation Controls	76
2.6 Session Tracking	87
2.6.1 Cookies	88
2.6.2 Session Tracking with HttpSessionState	97
2.7 Case Study: Online Guest Book	100
2.8 Case Study: Connecting to a Database in ASP .NET	113

Following figure is also an example of general tree where root is "Desktop".



Binary Tree:

A tree in which each element may have 0-child, 1-child or maximum of 2-children. A *Binary Tree* **T** is defined as a finite set of elements, called *nodes*, such that:

- a) **T** is empty (called the null tree or empty tree.)
- b) **T** contains a distinguished node **R**, called the *root* of **T**, and the remaining nodes of **T** form an ordered pair of disjoint binary trees **T₁** and **T₂**.

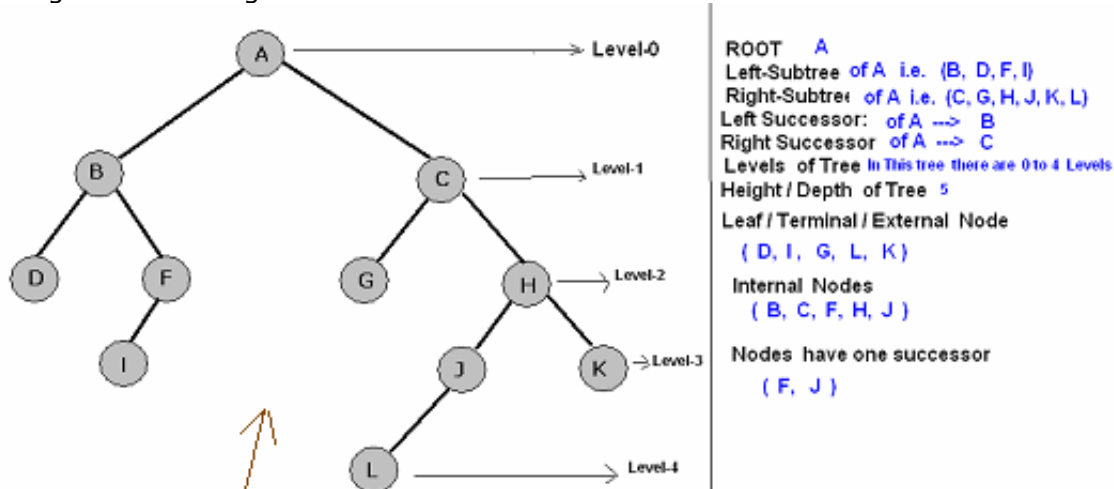
If **T** does contain a *root* **R**, then the two trees **T₁** and **T₂** are called, respectively, the **left sub tree** and **right sub tree** of **R**.

If **T₁** is non empty, then its node is called the left successor of **R**; similarly, if **T₂** is non empty, then its node is called the **right successor** of **R**. The nodes with no successors are called the **terminal nodes**. If **N** is a node in **T** with left successor **S₁** and right successor **S₂**, then **N** is called the **parent(or father)** of **S₁** and **S₂**. Analogously, **S₁** is called the left child (or son) of **N**, and **S₂** is called the right child (or son) of **N**. Furthermore, **S₁** and **S₂** are said to be **siblings (or brothers)**. Every node in the binary tree **T**, except the root, has a unique parent, called the predecessor of **N**. The line drawn from a node **N** of **T** to a successor is called an **edge**, and a sequence of consecutive edges is called a path. A terminal node is called a **leaf**, and a path ending in a leaf is called a **branch**.

The **depth (or height)** of a tree **T** is the maximum number of nodes in a branch of **T**. This turns out to be 1 more than the largest level number of **T**.

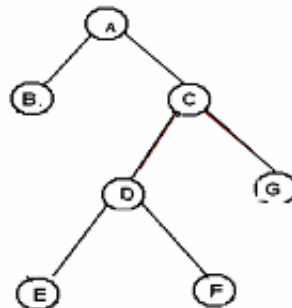
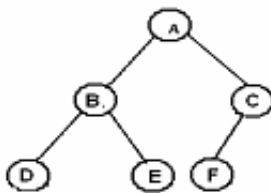
Level of node & its generation:

Each node in binary tree T is assigned a level number, as follows. The root R of the tree T is assigned the level number 0, and every other node is assigned a level number which is 1 more than the level number of its parent. Furthermore, those nodes with the same level number are said to belong to the same generation.



Binary Tree

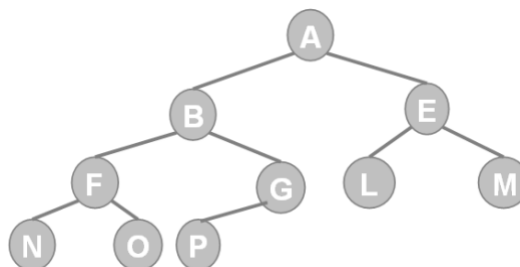
- 1- Simple Binary Tree
- 2- Complete Binary Tree
- 3- 2-Tree or Extended Tree



Complete Binary Tree:

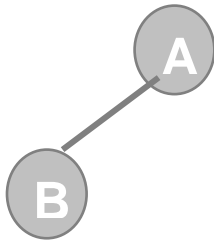
Consider a binary tree **T**. each node of T can have at most two children. Accordingly, one can show that level **n-2** of T can have at most two nodes.

A tree is said to be **complete** if all its levels, except possibly the last have the maximum number of possible nodes, and if all the nodes at the last level appear as far left as possible.

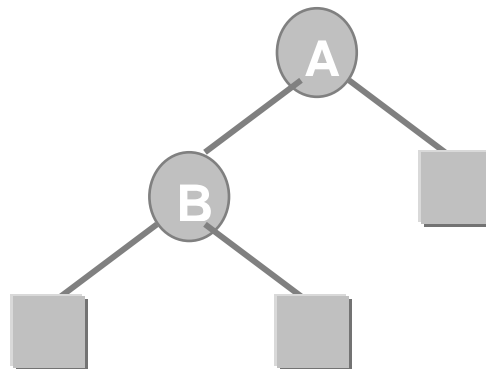


Extended Binary / Tree: 2-Tree:

A binary tree **T** is said to be a 2-tree or an extended binary tree if each node **N** has either 0 or 2 children. In such a case, the nodes, with 2 children are called internal nodes, and the node with 0 children are called external node.



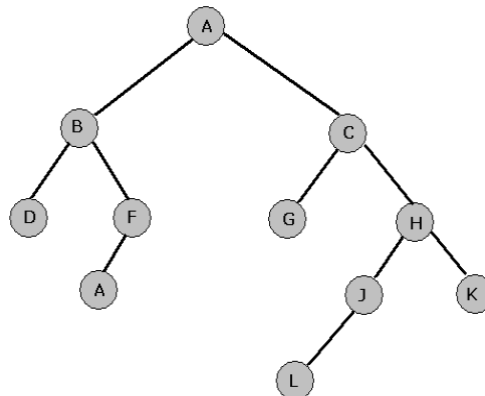
Binary Tree



Extended 2-tree

Traversing of Binary Tree:

A traversal of a tree **T** is a systematic way of accessing or visiting all the node of **T**. There are three standard ways of traversing a binary tree **T** with root **R**. these are :

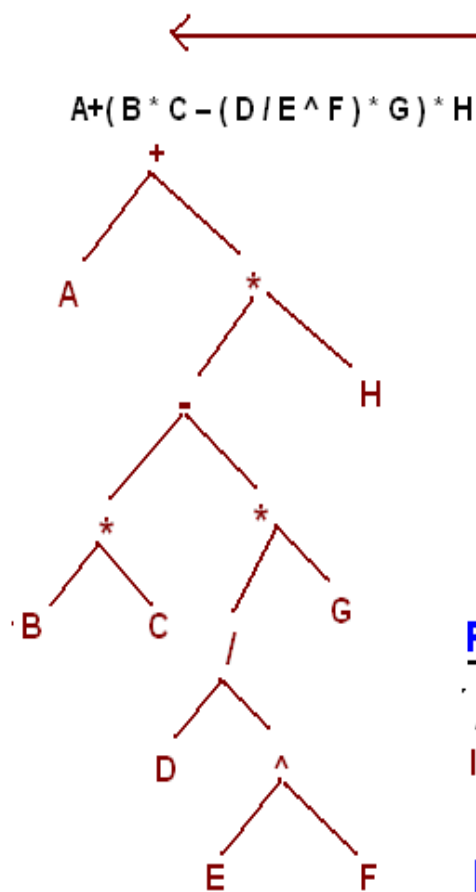


- **Preorder (N L R):** **(A B D F I C G H J L K)**
 - a) Process the node/root.
 - b) Traverse the Left sub tree.
 - c) Traverse the Right sub tree.
- **Inorder (L N R):** **(D B I F A G C L J H K)**
 - a) Traverse the Left sub tree.
 - b) Process the node/root.
 - c) Traverse the Right sub tree.
- **Postorder (L R N):** **(D I F B G L J K H C A)**
 - a) Traverse the Left sub tree.
 - b) Traverse the Right sub tree.
 - c) Process the node/root.
- **Descending order (R N L):** **(K H J L C G A F I B D)**

(Used in Binary Search Tree, will be discussed later)

 - a) Traverse the Right sub tree.
 - b) Process the node/root.
 - c) Traverse the Left sub tree.

Preparing a Tree from an infix arithmetic expression



Find operator which has low priority with respect to execution, starting from right to left. Put it on root and then continue this processor on sub-trees. The infix expression on the left side is converted into tree. Examine this tree is a 2-Tree, where each node either has two nodes or zero nodes.

All internal nodes are Operators

+ * - * * /

and all leaf nodes are Operands

A H B C G D E F

Post Order Traversing (LRN)

A B C * D E F ^ / G * - H * +

It produce postfix expression

Pre-Order Traversing (NLR)

+ A * - * B C * / D ^ E F G H

it produce prefix arithmetic expression

Recursion:

For implementing tree traversal logics as stated, two approaches are used, i.e. use stacks or recursive functions. In this lecture notes, we will see only recursion approach to implement these algorithms. Students will also use stacks to implement these algorithms as homework.

Recursion is an important concept in computer science. Many algorithms can be best described in terms of recursion. A recursive function / procedure containing a **Call** statement to itself. To make a function recursive one must consider the following properties:

- (1) There must be certain criteria (using arguments), called **base criteria**, for which the procedure does not call itself.
- (2) Each time the procedure does call itself, it must be closer to the base criteria.

Following examples helps to clarify of these ideas:

/* The following program demonstrates function call of itself. The following function does not follows the first property i.e. (recursion must has some criteria). Endless execution */

```
#include <stdio.h>

void disp( );
int main( )
{
    printf("\nHello");
    disp( ); /* A function, call */
    return 0;
}

void disp( )
{ printf("Hello\t");
  disp( ); /* A function call to itself without any criteria */
}
```

Don't run above program, it is still an explanation thus program is not valid logically.

The second property of recursion i.e. (after each cycle/iteration control must reach closer to the base criteria) and it is ignored here, so the following program is logically invalid.

```
#include <stdio.h>
void numbers(int );
int main( )
{
    int i=10;
    numbers(n);
    return 0;
}

void numbers(int n )
{
    printf("Hello\t");
    if(n >= 1 ) numbers(n+1); // this needs explanation
    // after each iteration, control is going far from base criteria
}
```

Factorial Function:

The product of the positive integers from ***n to 1***, inclusive, is called "*n_factorial*" and is usually denoted by ***n!***.

It is also convenient to define $0! = 1$, so that the function is defined for all nonnegative integers. Thus we have:

0! = 1	1! = 1	2! -> 2*1 = 2	3! -> 3*2*1 = 6
4! -> 4*3*2*1 = 24	5! -> 5*4*3*2*1 = 120	6! -> 6*5*4*3*2*1 = 720	

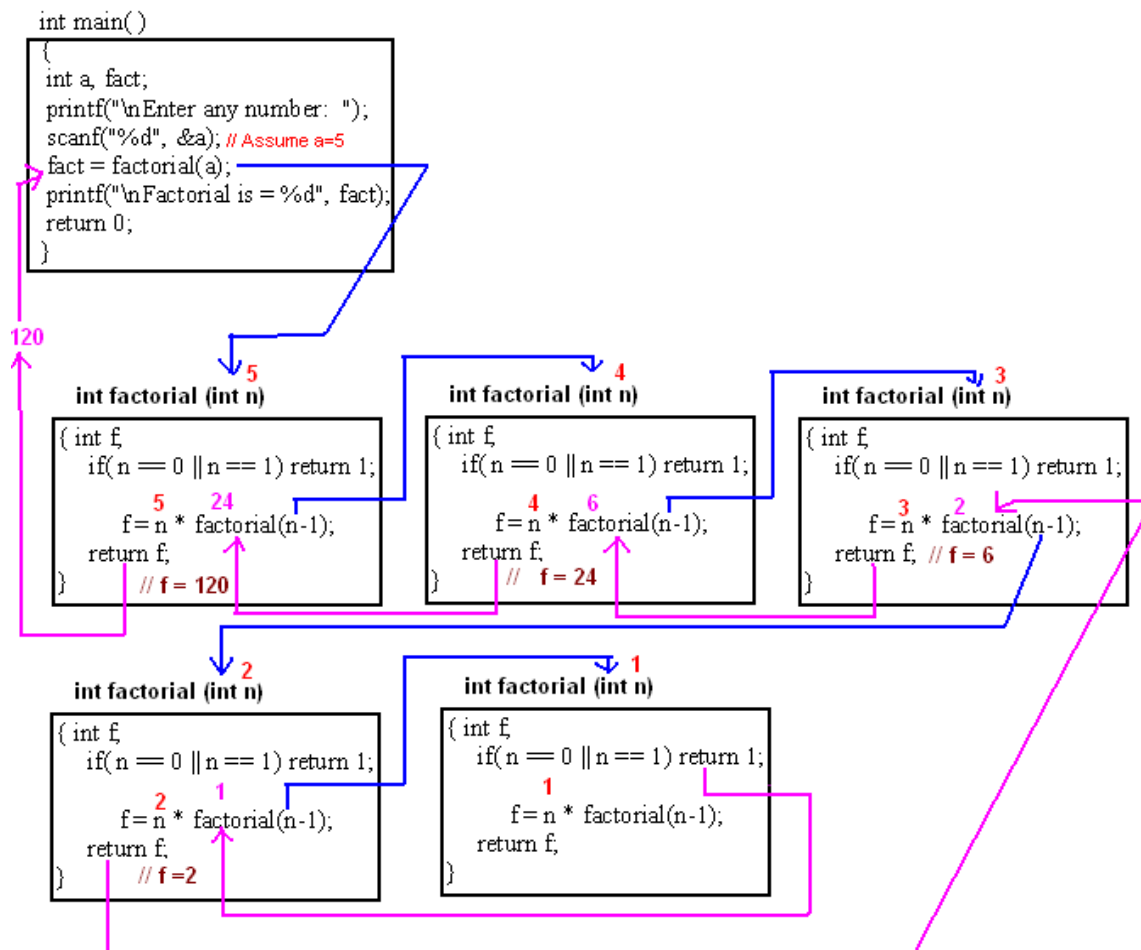
DATA STRUCTURES

Program to find the factorial of the given number using for loop:

```
#include <stdio.h>
int factorial(int);
void main
{ int a, fact;
  printf("\nEnter any number: ");
  scanf("%d", &a);
  fact = factorial(a);
  printf("\nFactorial is = %d", fact);
}
int factorial(int n)
{   int f = 1, i;
    for( i = n; i>=1; i--)
        f = f * i;
    return f;
}
```

To find the factorial of a given number using recursion

```
#include <stdio.h>
int factorial(int);
int main( )
{ int a, fact;
  printf("\nEnter any number: ");
  scanf("%d", &a);
  fact = factorial(a);
  printf("\nFactorial is = %d", fact);
  return 0;
}
int factorial(int n)
{ int f;
  if( n == 0 || n == 1) return 1;
  f = n * factorial(n-1);
  return f;
}
```

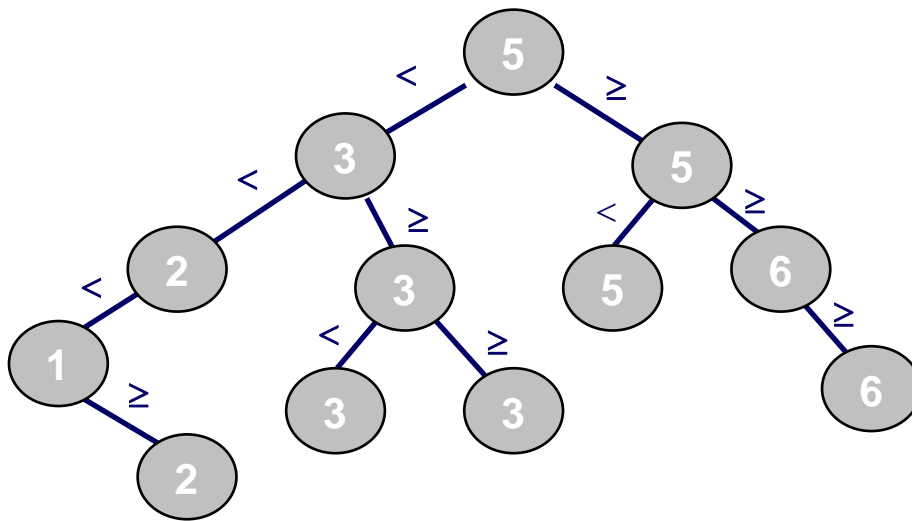


Binary Search Tree:

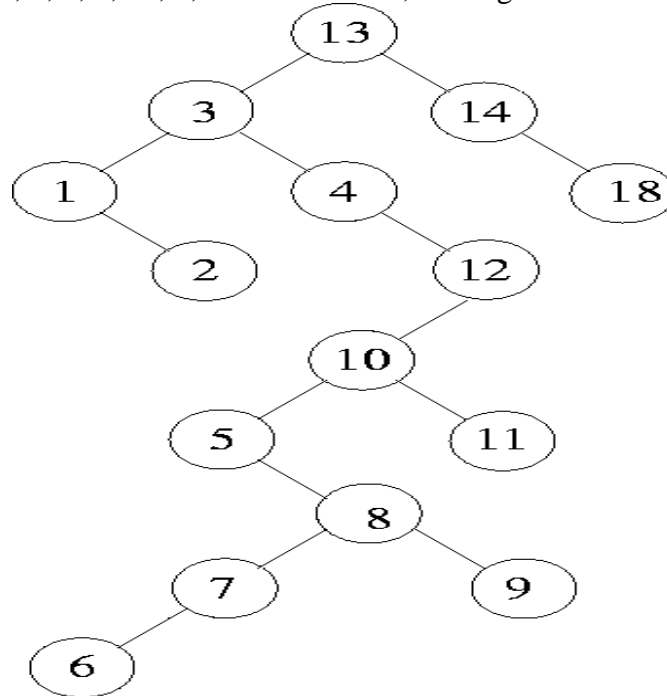
Suppose T is a binary tree, the T is called a binary search tree or binary sorted tree if each node N of T has the following property:

The values of at N (node) is greater than every value in the left sub tree of N and is less than every value in the right sub tree of N.

Binary Search Tree using these values: (50, 30, 55, 25, 10, 35, 31, 20, 53, 60, 62)



Following figure shows a **Binary Search Tree**. Notice that this tree is obtained by inserting the values 13, 3, 4, 12, 14, 10, 5, 1, 8, 2, 7, 9, 11, 6, 18 in that order, starting from an empty tree.

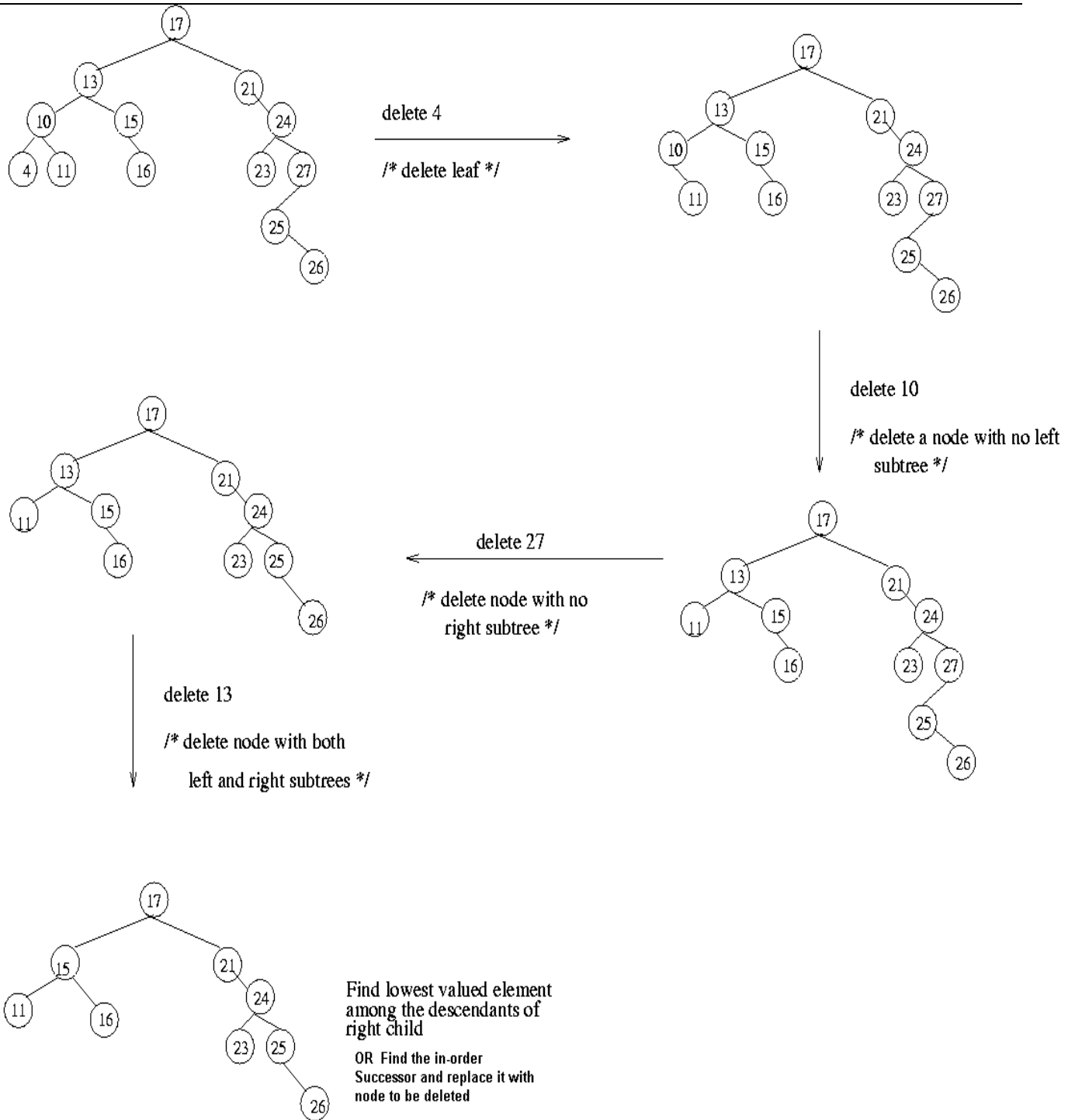


- **Sorting:** Note that inorder traversal of a binary search tree always gives a sorted sequence of the values. This is a direct consequence of the BST property. This provides a way of sorting a given sequence of keys: first, create a BST with these keys and then do an inorder traversal of the BST so created.

Inorder Travers (LNR) : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 15, 18

- **Search:** is straightforward in a BST. Start with the root and keep moving left or right using the BST property. If the key we are seeking is present, this search procedure will lead us to the key. If the key is not present, we end up in a null link.
- **Insertion** in a BST is also a straightforward operation. If we need to insert an element **n**, we traverse tree starting from root by considering the above stated rules. Our traverse procedure ends in a null link. It is at this position of this null link that **n** will be included. .
- **Deletion in BST:** Let **x** be a value to be deleted from the BST and let **N** denote the node containing the value **x**. Deletion of an element in a BST again uses the BST property in a critical way. When we delete the node **N** containing **x**, it would create a "gap" that should be filled by a suitable existing node of the BST. There are two possible candidate nodes that can fill this gap, in a way that the BST property is not violated: (1). Node containing highest valued element among all descendants of left child of **N**. (2). Node containing the lowest valued element among all the descendants of the right child of **N**.

Following figure on next page illustrates several scenarios for deletion in BSTs.



```
/* This program is about to implement different operations on Binary Search Tree.
```

```
*/
```

```
#include <iostream.h>
```

```
#include <stdlib.h>
```

```
#include <process.h>
```

```
struct NODE
```

```
{
```

```
    int    info;
```

```
    struct NODE *Left;
```

```
    struct NODE *Right;
```

```
};
```

```
struct NODE *Root = NULL; // initially Root is NULL
```

```
void AttachNode( struct NODE *pRoot, struct NODE *pNew )
```

```
{
```

```
    if( Root == NULL ) // to attach first node with tree
```

```
    {
```

```
        Root = pNew; // attaches node on first root
```

```
    }
```

```
    else
```

```
    {
```

```
        if (pNew->info < pRoot->info )
```

```
        { // traverse to left sub-tree and find null at left
```

```
            if( pRoot->Left != NULL)
```

```
                AttachNode( pRoot->Left, pNew ); // recursive call
```

```
            else
```

```
                pRoot->Left = pNew; // attaches node on left
```

```
        }
```

```
    else
```

```
    { // traverse to right sub-tree and find null at right
```

```
        if( pRoot->Right != NULL)
```

```
            AttachNode( pRoot->Right, pNew ); // recursive call
```

```
        else
```

```
            pRoot->Right = pNew; // attaches node on left
```

```
    }
```

```
}
```

```
}
```



```
void Insert(int x)
{
    struct NODE *NewNode= new NODE;
    NewNode->Left = NULL;
    NewNode->Right= NULL;
    NewNode->info = x;
    AttachNode( Root, NewNode );
}
```

```
void Pre_Order(struct NODE *pRoot)
{
    if (pRoot)
    {
        cout<<pRoot->info<<" ";
        Pre_Order(pRoot->Left);
        Pre_Order(pRoot->Right);
    }
}
```

```
void Post_Order(struct NODE *pRoot)
{
    if (pRoot)
    {
        Post_Order(pRoot->Left);
        Post_Order(pRoot->Right);
        cout<<pRoot->info<<" ";
    }
}
```

```
void In_Order(struct NODE *pRoot)
{
    if( pRoot )
    {
        if(pRoot->Left) In_Order(pRoot->Left);
        cout<<pRoot->info<<" ";
        if(pRoot->Right) In_Order(pRoot->Right);
    }
}
```

```
void DisplayDescending(struct NODE *pRoot)
{

```

```
if( pRoot )
{
    if(pRoot->Right) DisplayDescending(pRoot->Right);
    cout<<pRoot->info<<" ";
    if(pRoot->Left) DisplayDescending(pRoot->Left);
}
}
```

```

void DeleteTree( struct NODE *pRoot) // This function deletes all nodes in the tree
{
    if( pRoot )
    {
        if(pRoot->Right)
        {
            DeleteTree(pRoot->Right);
        }

        if(pRoot->Left)
        {
            DeleteTree(pRoot->Left);
        }

        free( pRoot );
    }
}

int main( void )
{
    int ch, item, loop=1;
    while( loop )
    {
        cout<<"\n\n\n  Binary Search Tree Functions\n\n";
        cout<<"\n1. Insert a New Node";
        cout<<"\n2. Remove Existing Node");
        cout<<"\n3. In-Order Traverse  (Ascending Order)";
        cout<<"\n4. Pre-Order Traverse ";
        cout<<"\n5. Post-Order Traverse ";
        cout<<"\n6. Display in Descending Order  (Reverse)";
        cout<<"\n7. Exit";
        cout<<"\nEnter you choice: ";
        cin>>ch;

        switch(ch)
        {
            case 1:
                cout<<"\n\n put a number: ";    cin>>item;
                Insert(item);
                break;

            case 2:
                // Remove(); // This function is not defined.
                break; // Students shall write this function as home
work.

            case 3:

```

```
ORDER)\n";
    cout<<"\n\n\n In-Order Traverse (ASCENDING
ORDER)\n";
    In_Order(Root);
    cout<<"\n\n";
    break;

case 4:
    cout<<"\n\n\n Pre-Order Traverse \n";
    Pre_Order(Root);
    cout<<"\n\n";
    break;
case 5:
    cout<<"\n\n\n Post-Order Traverse \n";
    Post_Order(Root);
    cout<<"\n\n";
    break;

case 6:
    cout<<"\n\n\nDESCENDING ORDER (Reverse )\n";
    DisplayDescending(Root);
    cout<<"\n\n";
    break;

case 7:
    DeleteTree(Root);
    loop=0;
    break;

default:
    cout<<"\n\nInvalid Input";

    } // end of switch
    } // end of while loop
} // end of main( ) function
```

AVL Tree:

All BST operations are $O(d)$, where d is tree depth. The minimum d is

$$d = \lfloor \log N \rfloor$$

For a binary tree with N nodes:

- What is the best case tree?
- What is the worst case tree?

So, the best case running time of BST operations is $O(\log N)$ and Worst case running time is $O(N)$

- What happens when you Insert elements are in ascending order?

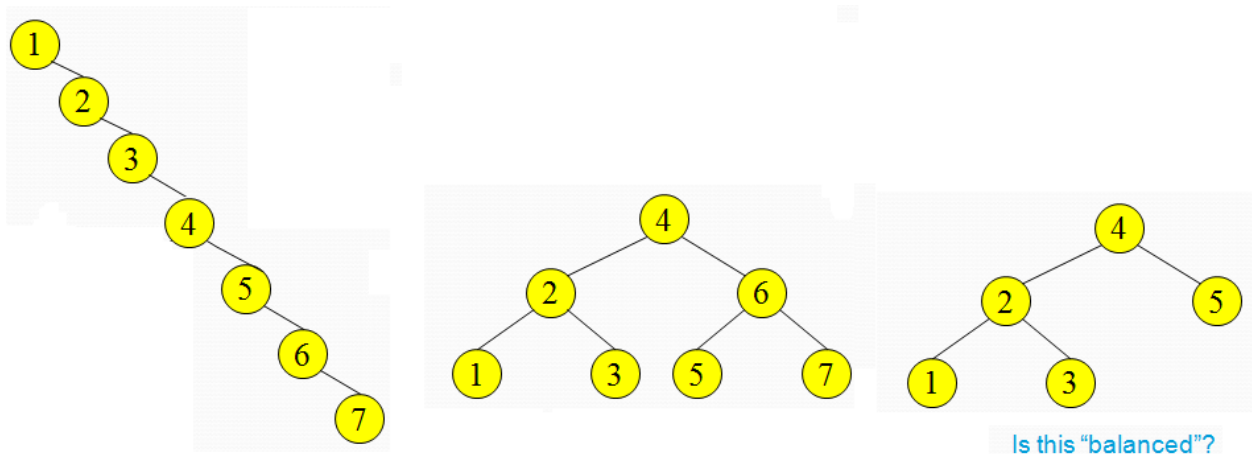
Insert:

into an empty BST. (Worst Case)

- Problem: Lack of "balance":
Compare depths of left and right subtree.

See the following example:

Balanced and unbalanced BST



Balancing Binary Search Tree:

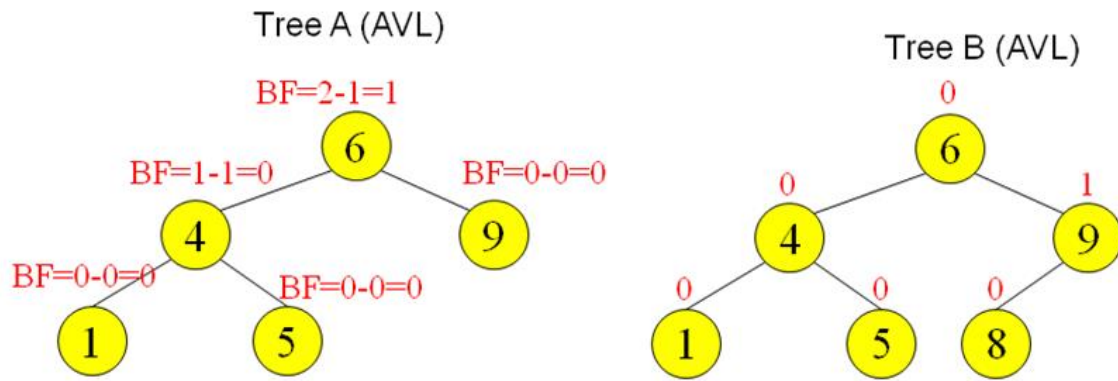
Many algorithms exist for keeping binary search trees balanced.

- Adelson-Velskii and Landis (AVL) trees (height-balanced trees)
- Splay trees and other self-adjusting trees.
- B-trees and other multiway search trees.

Definition

- Binary tree.
- If T is a nonempty binary tree with T_L and T_R as its left and right subtrees, then T is an AVL tree if
 1. T_L and T_R are AVL trees, and
 2. The difference between $|h_L - h_R|$ may has $(-1, 0, 1)$, where h_L and h_R are the heights of T_L and T_R , respectively.
- An AVL search tree is a binary search tree that is also an AVL tree.

Balance Factor:



X = is a node

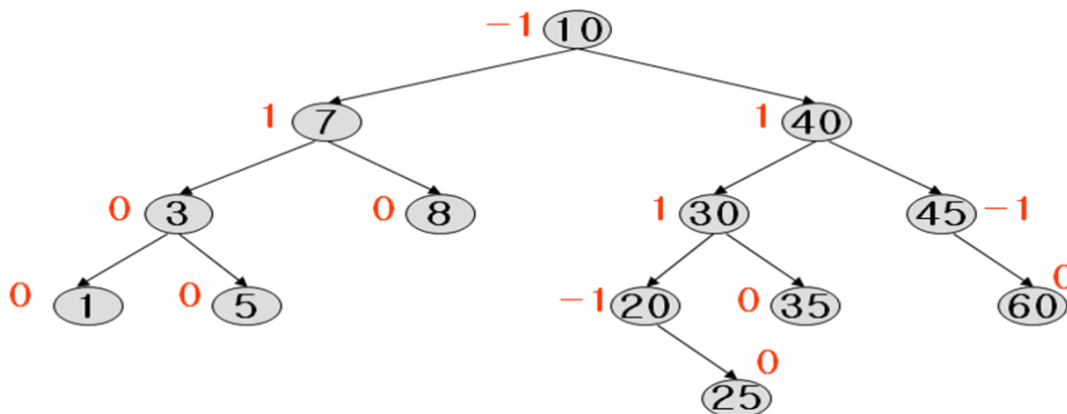
$\text{Height}(x)$ = Number of Edges on the longest path

The **balance factor** $\text{bf}(x)$ of a node x is defined as

$\text{height}(x \rightarrow \text{left_subtree}) - \text{height}(x \rightarrow \text{right_subtree})$

Balance factor of each node in an AVL tree can be -1 , 0 , or 1

AVL Tree with Balance Factors:

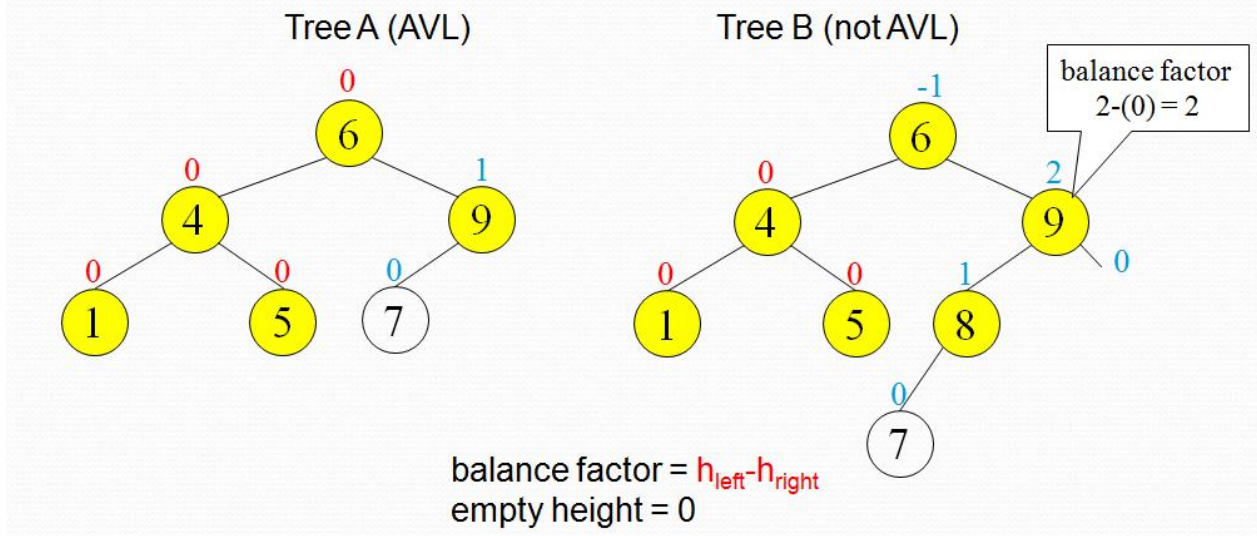


- Is this an AVL tree?
- What is the balance factor for each node in this AVL tree?
- Is this an AVL search tree?

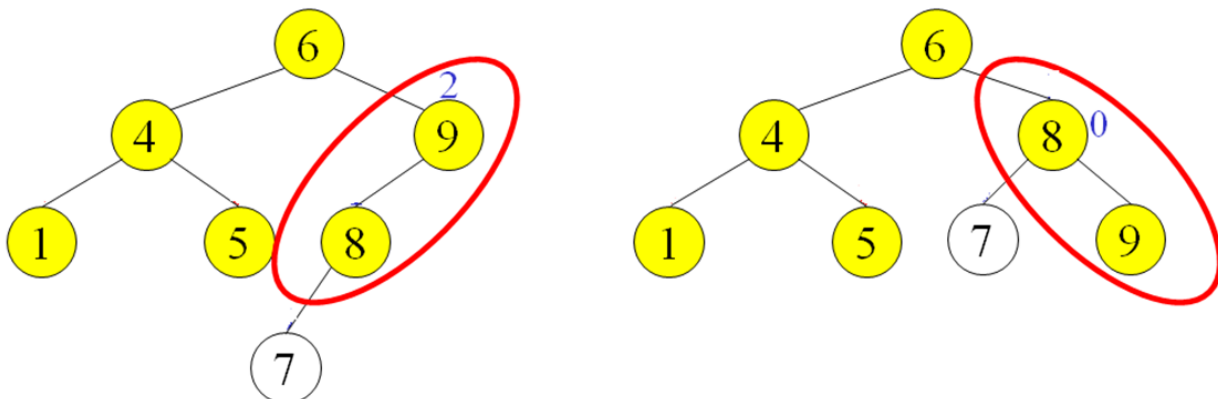
Note: Whenever there is an **insertion** or **deletion** operation occurs in a Binary Search Tree, it may cause to make BST imbalance in height.

- If condition as stated above violated after a node insertion or deletion
 - Which nodes do we need to rotate?
 - Only nodes on path from insertion/deletion point to root may have their balance altered
- **Rebalance the tree through rotation at the deepest node with balance violated**
 - The entire tree will be rebalanced
- Violation cases at node A (deepest node)

Balance factor after Insert 7:



Single Rotation in an AVL Tree:



Insert and Rotation in AVL Trees:

- Insert operation may cause balance factor to become 2 or -2 for some node
 - only nodes on the path from insertion point to root node have possibly changed in height
 - So after the Insert, go back up to the root node by node, updating heights
 - If a new balance factor (the difference $h_{\text{left}} - h_{\text{right}}$) is 2 or -2, adjust tree by *rotation* around the node

Rotation:

Definition

- To switch *children* and *parents* among two or three adjacent nodes to **restore balance of a tree**.
- A rotation may change the depth of some nodes, but does not change their relative ordering. (inorder traversal of the tree is maintained)

Rotations in AVL Tree:

Let the node that needs rebalancing be α .

There are 4 cases:

require single rotation :

1. Insertion into **left subtree of left** child of α . (LL) (+tive balance factor)
2. Insertion into **right subtree of right** child of α . (RR) (-tive balance factor)

require double rotation :

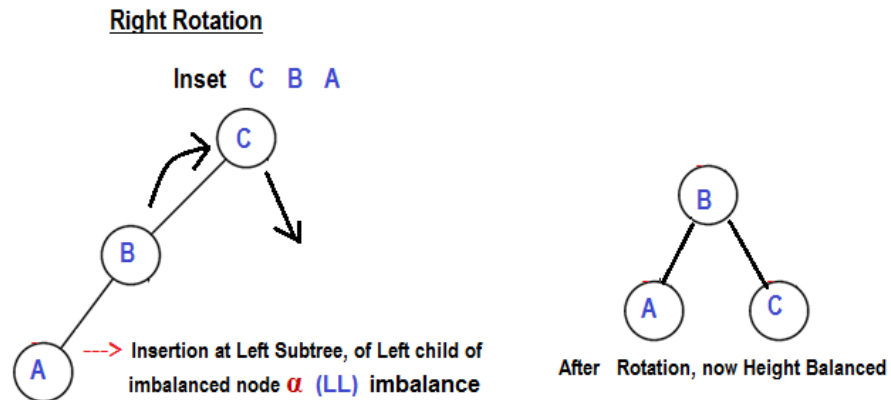
3. Insertion into **right subtree of left** child of α . (RL) (+tive balance factor)
4. Insertion into **left subtree of right** child of α . (LR) (-tive balance factor)

The rebalancing is performed through four separate rotation algorithms. Left rotation, right rotation, Right left double rotation, Left Right double rotation

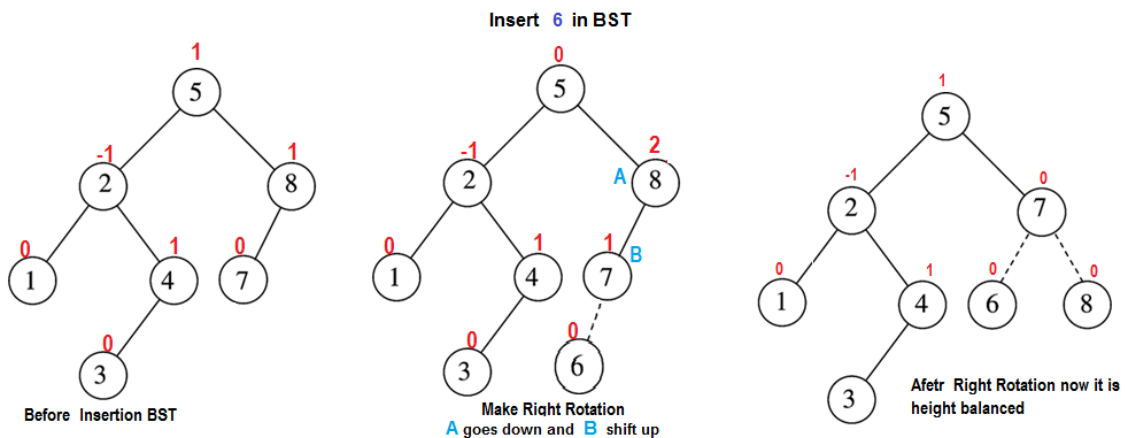
Single and Double Rotations

- **Single rotations**: the transformations done to correct LL and RR imbalances
- **Double rotations**: the transformations done to correct LR and RL imbalances
 - The transformation **to correct LR imbalance** can be achieved by an **RR rotation followed by an LL rotation**
 - The transformation **to correct RL imbalance** can be achieved by an **LL rotation followed by an RR rotation**

Right Rotation: (Insertion at Left Subtree of Left Child of α LL imbalance)

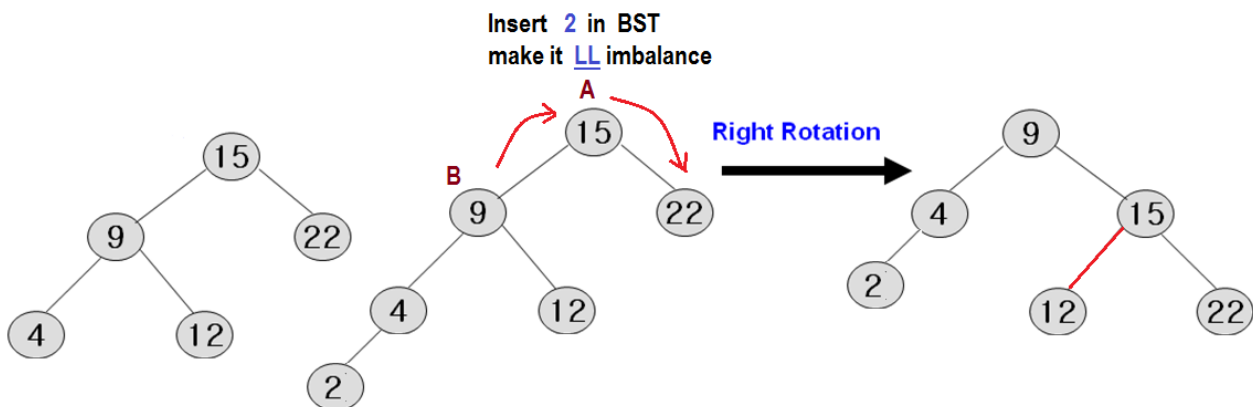


2nd Example: (Insertion at Left Subtree of Left Child of α LL imbalance)



Third Example of LL imbalance Right Rotation:

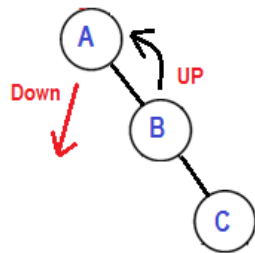
- In a binary search tree, pushing a node A down and to the right to balance the tree.
- A's left child B replaces A, and the right child B becomes left of A



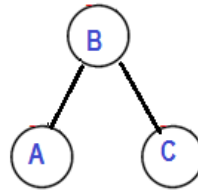
Left Rotation: (Insertion at Right Subtree of Right Child of A RR imbalance)

Left Rotation

Inset A B C



---> Left subtree, of left child of imbalanced node A LL imbalance

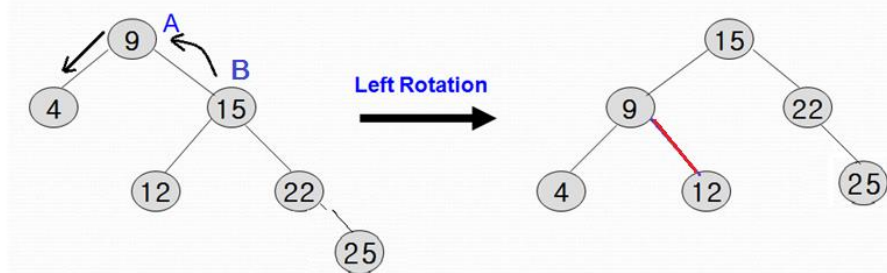
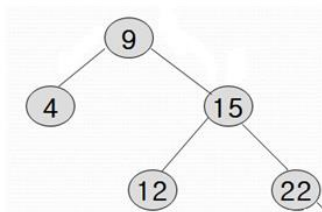


After Left Rotation, now Height Balanced

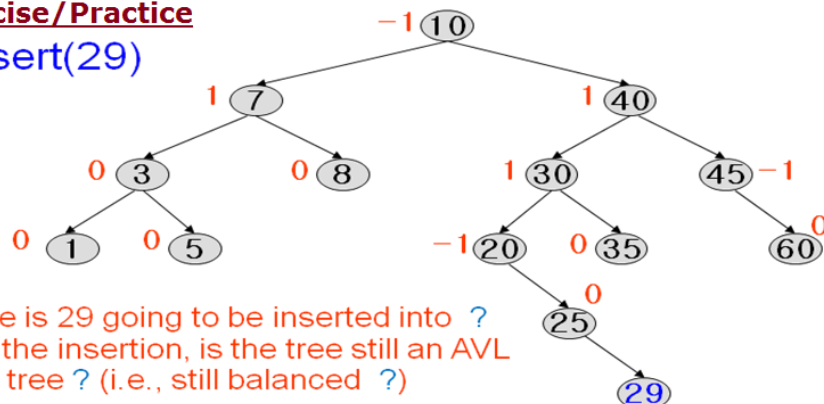
Second Example (RR) imbalance Left Rotation

- In a binary search tree, pushing the node (A) (who's balance factor disturbed) down to the left to make balance of tree.
- A's right child (B) replaces (A), and the right child's left child becomes A's right child.

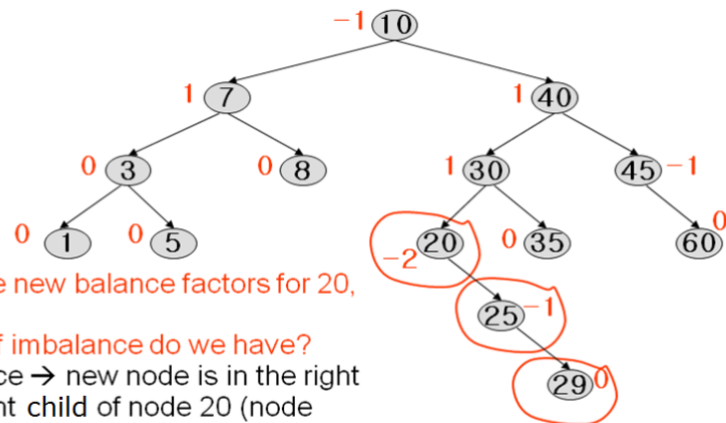
Inset 25 in BST
make height RR imbalance



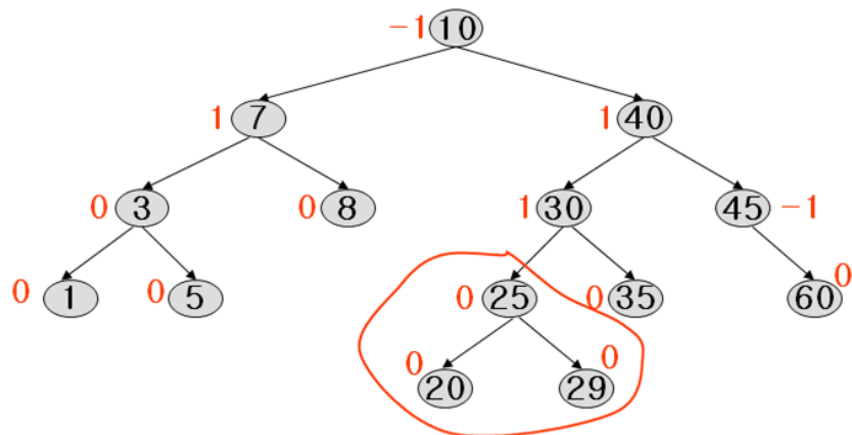
Exercise/Practice Insert(29)



- Where is 29 going to be inserted into ?
- After the insertion, is the tree still an AVL search tree ? (i.e., still balanced ?)



- What are the new balance factors for 20, 25, 29?
- What type of imbalance do we have?
- RR imbalance → new node is in the right subtree of right child of node 20 (node with bf = -2) → what rotation do we need?
- What would the left subtree of 30 look like after left rotation?

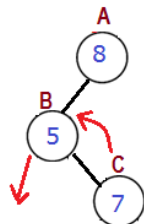


- After the Left rotation, is the resulting tree an AVL search tree ?

Double Rotation

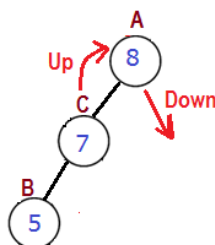
Left-Right Rotation

Insert 8 5 7 in BST



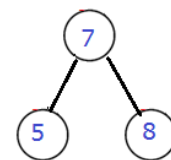
Insertion at Right subtree, of Left child of imbalanced node **RL** imbalance

First
Left-Rotation between B and C



Still not Balanced

Second
Right-Rotation between A and C

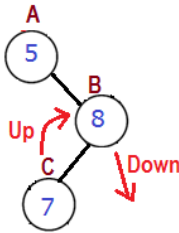


After Right Rotation, now Height Balanced

Double Rotation

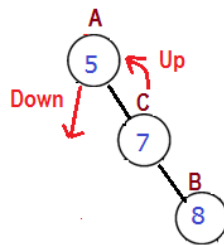
Right-Left Rotation

Insert 5 8 7 in BST



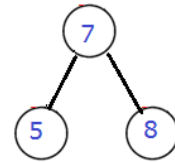
Insertion at Left subtree of Right child of imbalanced node LR imbalance

First
Right-Rotation between B and C



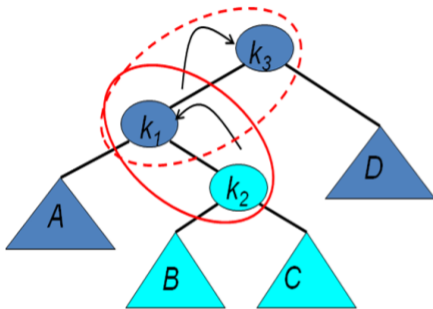
Still not Balanced

Second
Left-Rotation between A and C



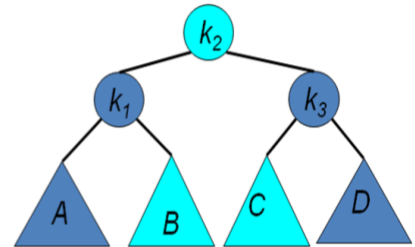
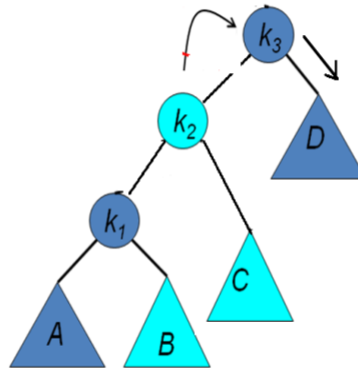
After Left Rotation, now Height Balanced

Double rotation: Left right

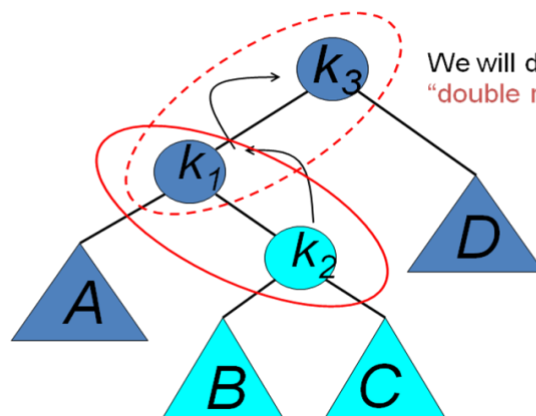


Left rotation between K1 and k2

After that right rotation between k2 and k3

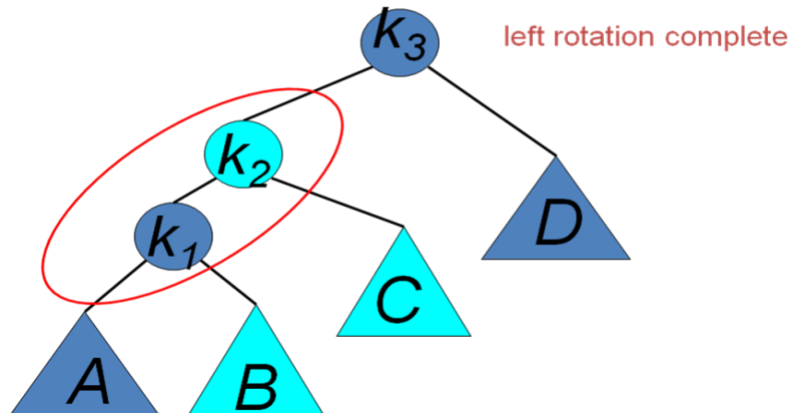


Double rotation: Left right (steps)

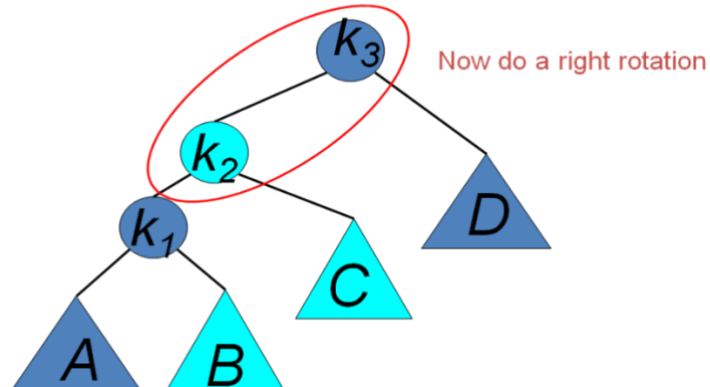


We will do a left-right
"double rotation" . . .

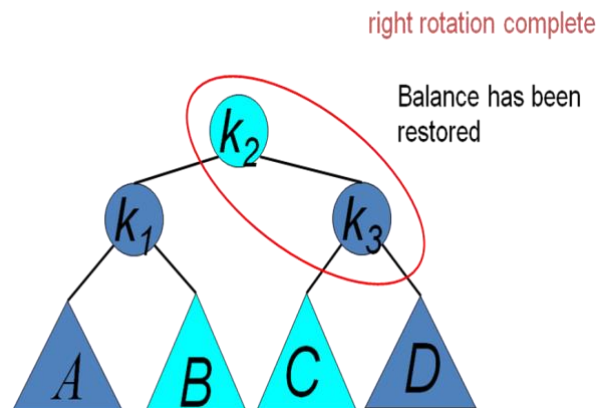
Double rotation: Left right (steps)



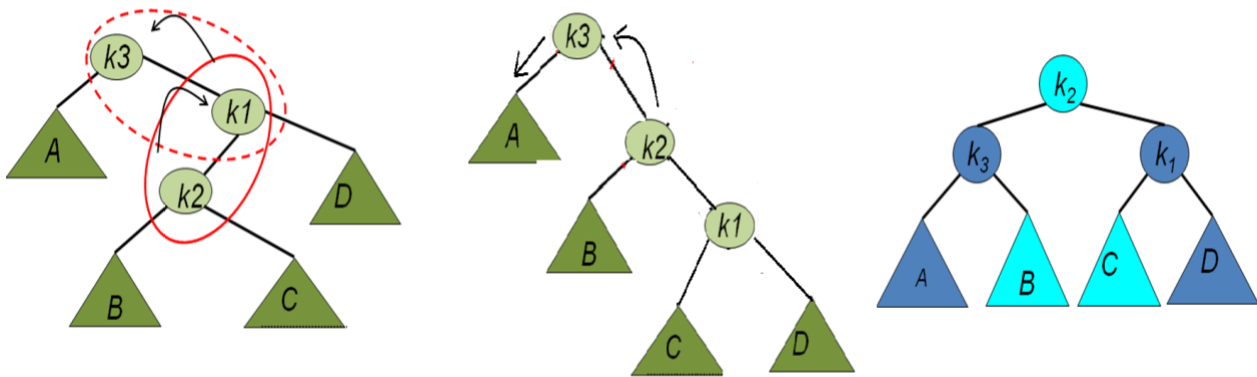
Double rotation: Left right (steps)



Double rotation: Left right (steps)



Double rotation : Right-Left Rotation

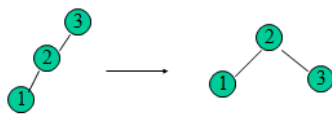


Right rotation between K1 and k2

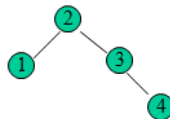
After that left rotation between k2 and k3

Example-1 of AVL Tree Construction:

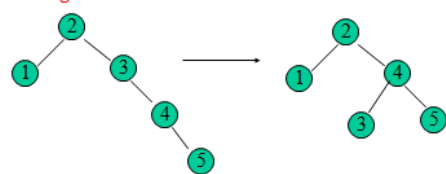
- Inserting 3, 2, 1, into empty AVL tree



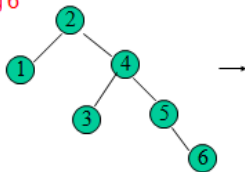
- Inserting 4



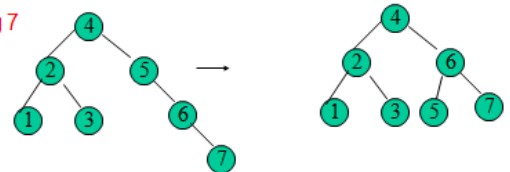
- Inserting 5



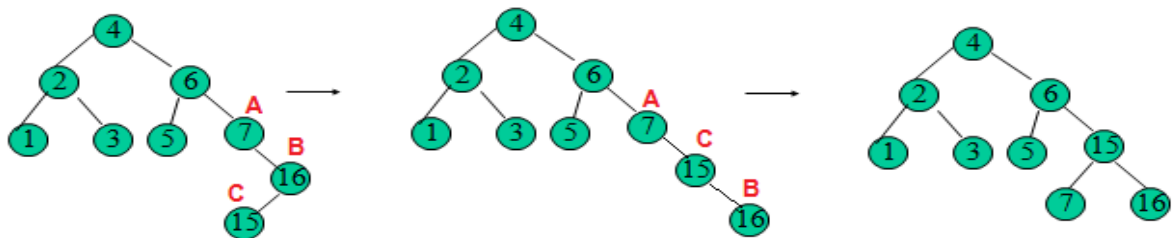
- Inserting 6



- Inserting 7



- Inserting 16 and 15 (Double Rotation [Right-Left])

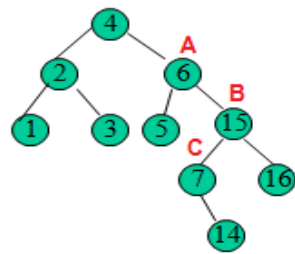


Right Rotation in B and C

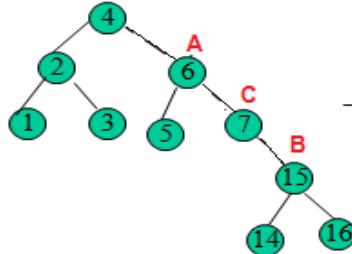
Left Rotation in A and C

Tree is Height balanced

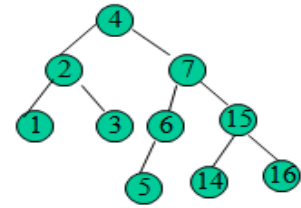
- Inserting 14 (Double Rotation [Right-Left])



Right Rotation in **B** and **C**



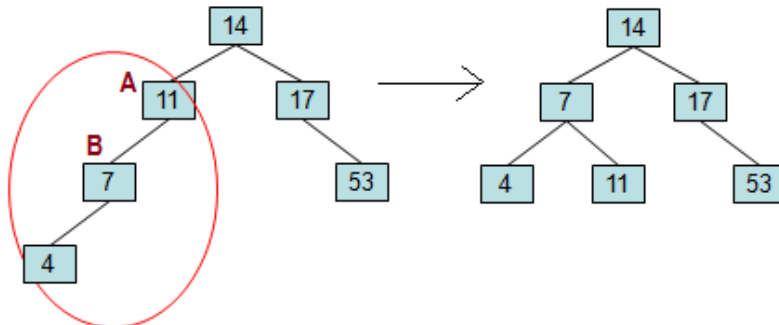
Left Rotation in **A** and **C**



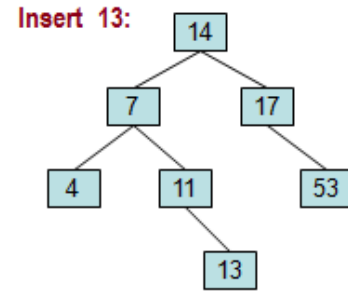
Tree is Height balanced

Example-2 of AVL Tree Construction:

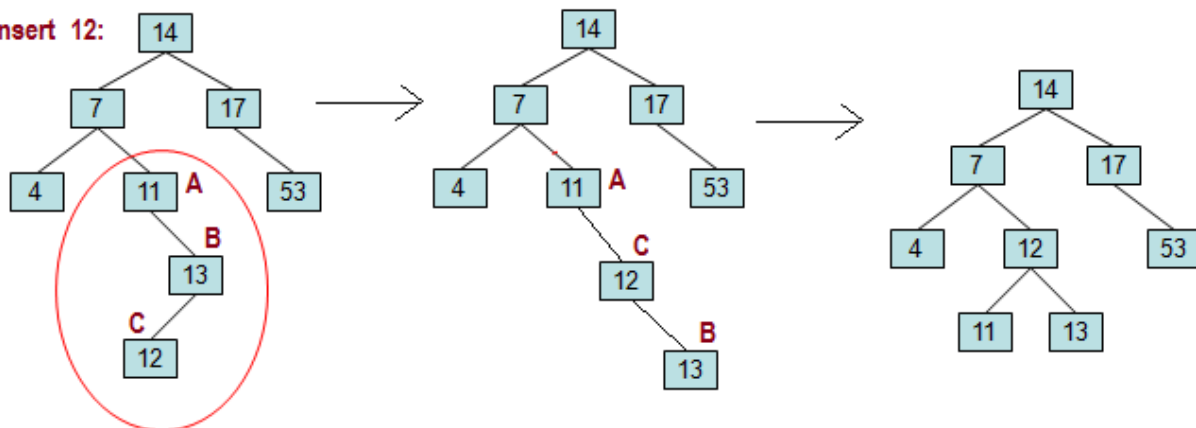
Insert 14, 17, 11, 7, 53, 4



Right Rotation in **A** and **B**



Insert 12:

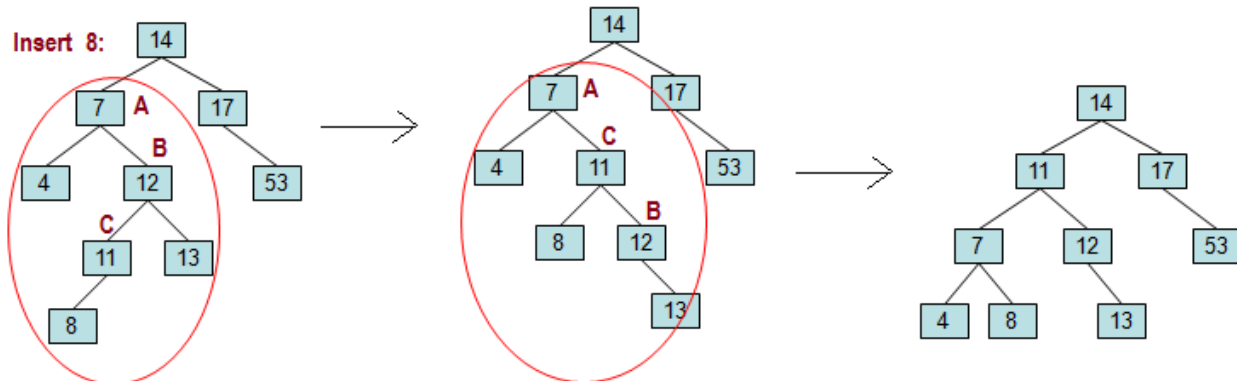


Right Rotation in **B** and **C**

Left Rotation in **A** and **C**

Now the AVL tree is balanced.

Insert 8:

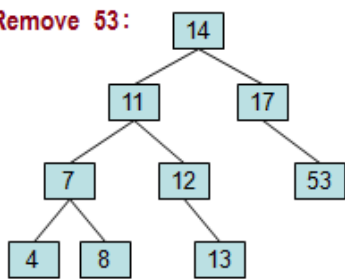


Right Rotation in **B** and **C**

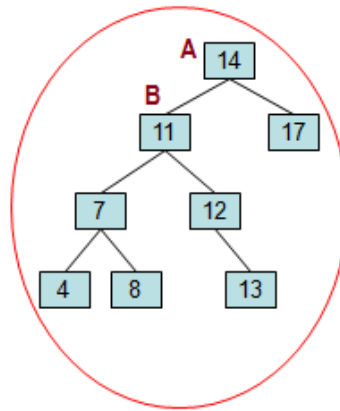
Left Rotation in **A** and **C**

• Now the AVL tree is balanced.

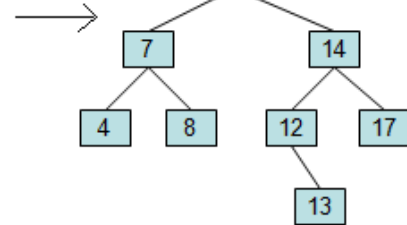
Remove 53:



• Now remove 53, unbalanced

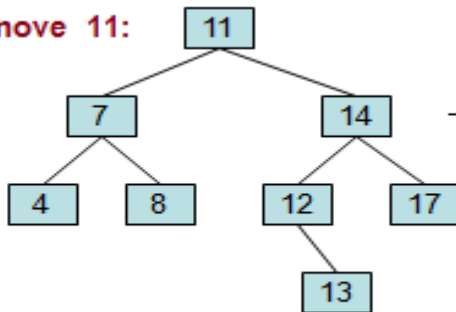


Right Rotation in **A** and **B**

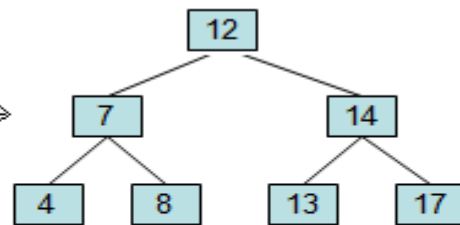


• Balanced!

Remove 11:

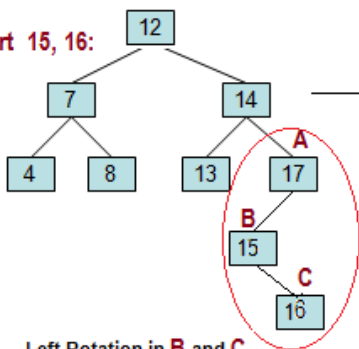


• Now Remove 11

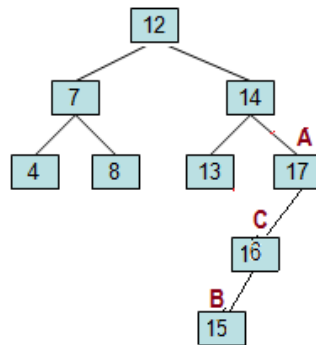


replace it with the smallest in its Right branch
Still Balanced

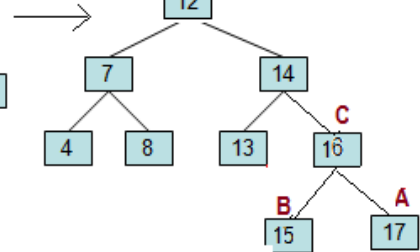
Insert 15, 16:



Left Rotation in **B** and **C**



Right Rotation in **A** and **C**

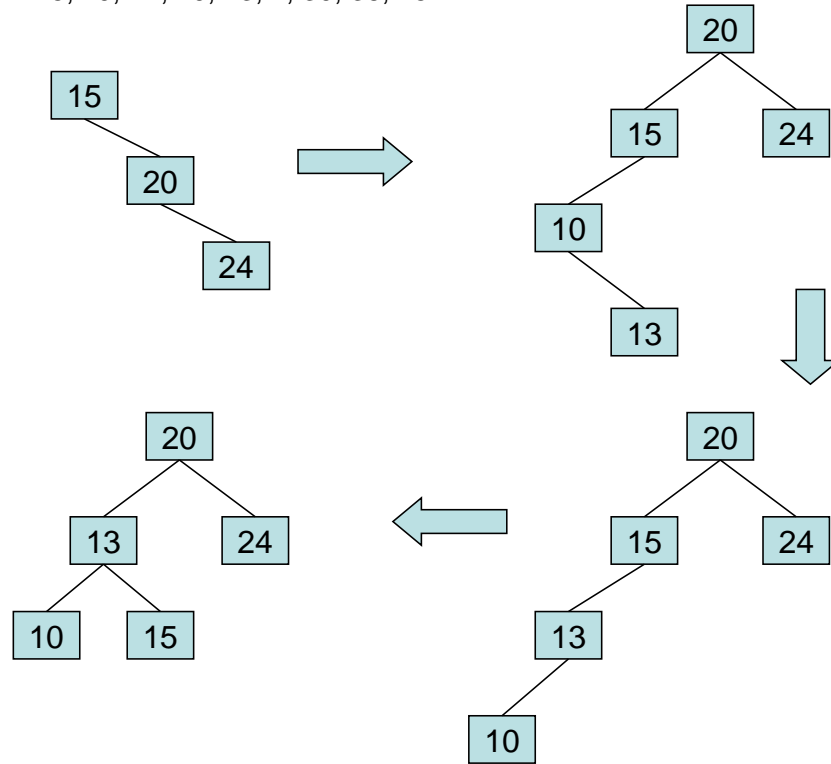


• Balanced!

Exercises

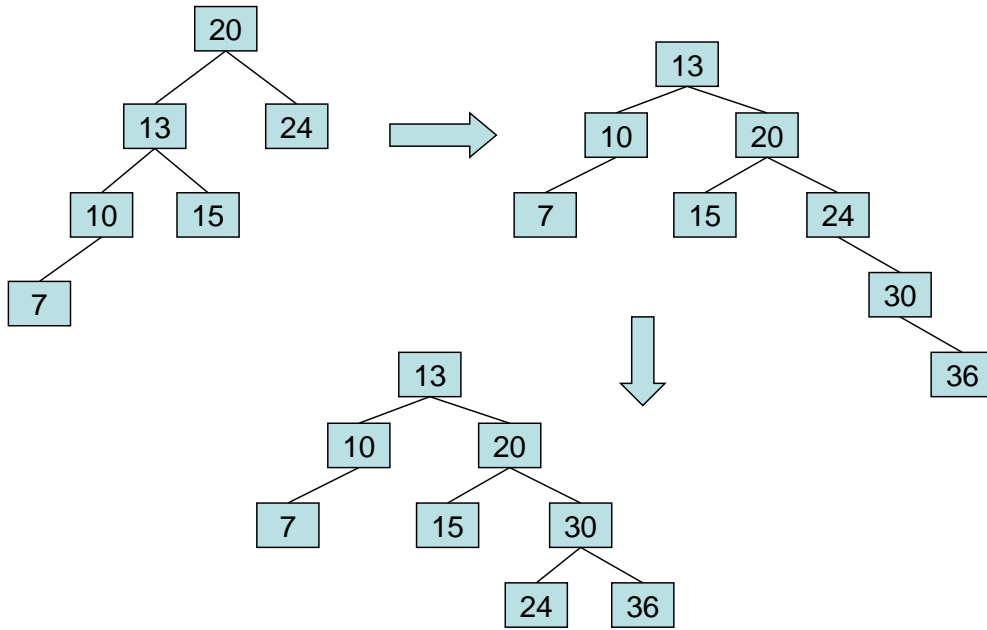
Build an AVL tree with the following values:
15, 20, 24, 10, 13, 7, 30, 36, 25

15, 20, 24, 10, 13, 7, 30, 36, 25

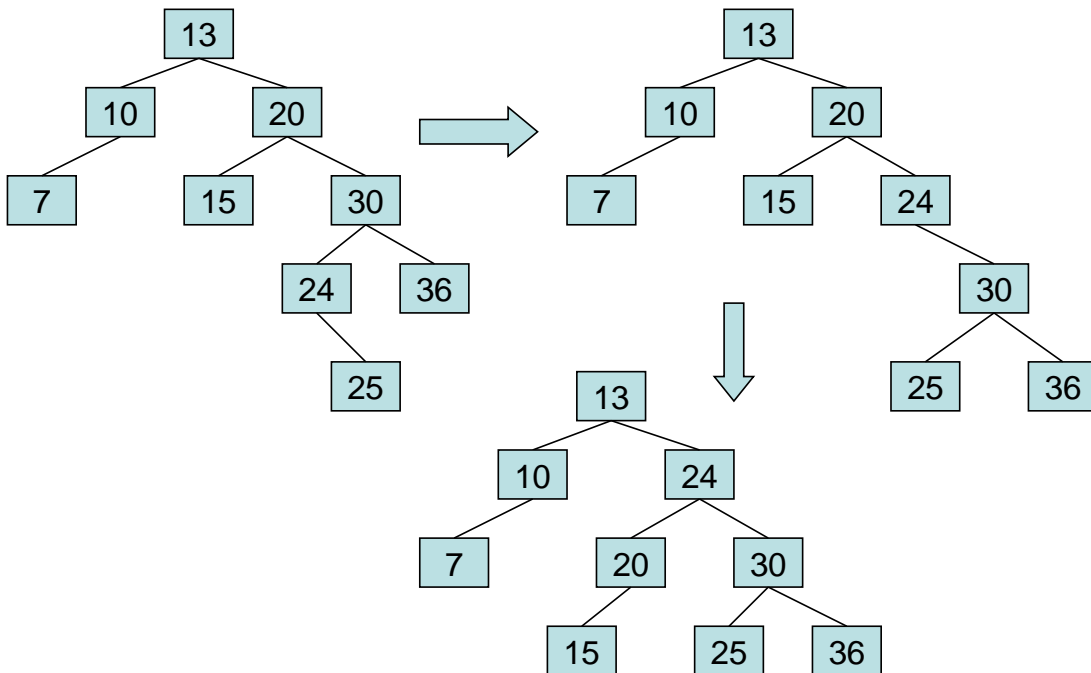


DATA STRUCTURES

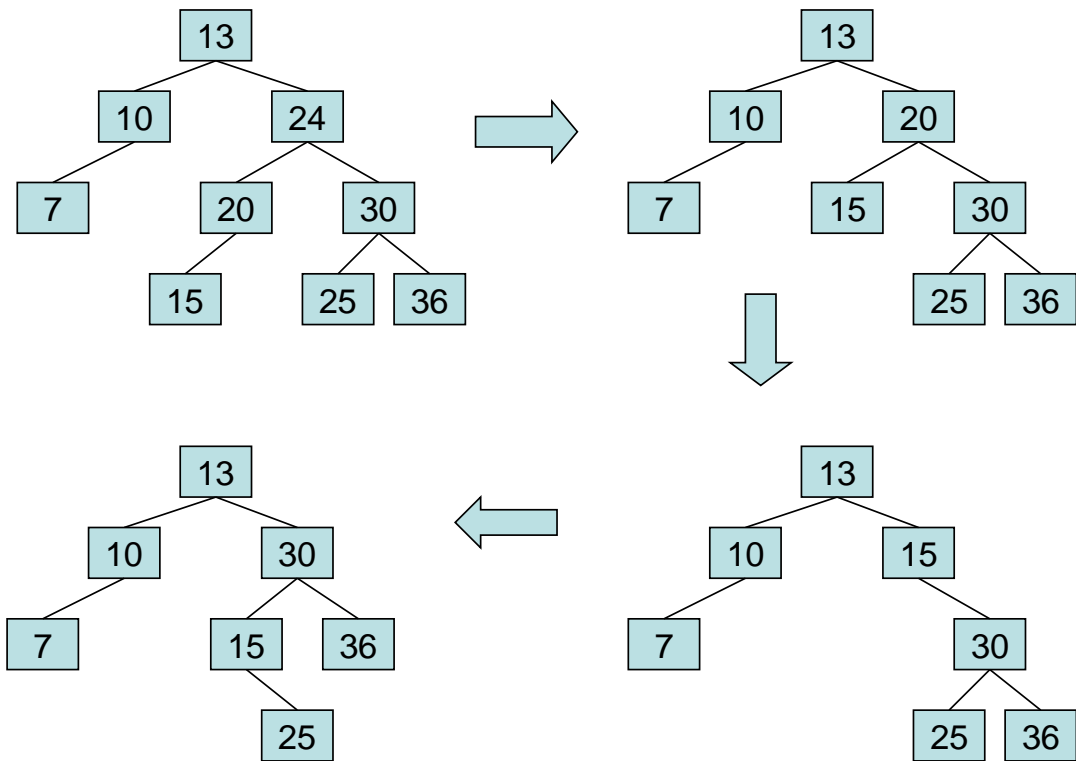
15, 20, 24, 10, 13, 7, 30, 36, 25



15, 20, 24, 10, 13, 7, 30, 36, 25



Remove 24 and 20 from the AVL tree.



Implementation of AVL Tree

```

1  struct AvlNode
2  {
3      Comparable element;
4      AvlNode *left;
5      AvlNode *right;
6      int      height;
7
8      AvlNode( const Comparable & theElement, AvlNode *lt,
9              AvlNode *rt, int h = 0 )
10         : element( theElement ), left( lt ), right( rt ), height( h )
11     };
1
12 /**
13  * Return the height of node t or -1 if NULL.
14  */
15 int height( AvlNode *t ) const
16 {
17     return t == NULL ? -1 : t->height;
18 }
19
20 /**
21  * Internal method to insert into a subtree.
22  * x is the item to insert.
23  * t is the node that roots the subtree.
24  * Set the new root of the subtree.
25  */
26 void insert( const Comparable & x, AvlNode * & t )
27 {
28     if( t == NULL )
29     {
30         t = new AvlNode( x, NULL, NULL );
31     }
32     else if( x < t->element )
33     {
34         insert( x, t->left );
35         if( height( t->left ) - height( t->right ) == 2 )
36         {
37             if( x < t->left->element )
38                 rotateWithLeftChild( t ); ← Case 1
39             else
40                 doubleWithLeftChild( t ); ← Case 2
41         }
42     }
43     else if( t->element < x )
44     {
45         insert( x, t->right );
46         if( height( t->right ) - height( t->left ) == 2 )
47         {
48             if( t->right->element < x )
49                 rotateWithRightChild( t ); ← Case 4
50             else
51                 doubleWithRightChild( t ); ← Case 3
52         }
53     }
54     else
55         ; // Duplicate; do nothing
56     t->height = max( height( t->left ), height( t->right ) ) + 1;
57 }

```

Single Rotation (Case 1)

```

1  /**
2   * Rotate binary tree node with left child.
3   * For AVL trees, this is a single rotation for case 1.
4   * Update heights, then set new root.
5   */
6  void rotateWithLeftChild( AvlNode * & k2 )
7  {
8      AvlNode *k1 = k2->left;
9      k2->left = k1->right;
10     k1->right = k2;
11     k2->height = max( height( k2->left ), height( k2->right ) ) + 1;
12     k1->height = max( height( k1->left ), k2->height ) + 1;
13     k2 = k1;
14 }
```

Double Rotation (Case 2)

```

1  /**
2   * Double rotate binary tree node: first left child
3   * with its right child; then node k3 with new left child.
4   * For AVL trees, this is a double rotation for case 2.
5   * Update heights, then set new root.
6   */
7  void doubleWithLeftChild( AvlNode * & k3 )
8  {
9      rotateWithRightChild( k3->left );
10     rotateWithLeftChild( k3 );
11 }
```