Computer College, Department of Computer Science, Kingdom of Saudi Arabia

Data Structures (CS-214)

- ***INSTRUCTOR:***

Shahid Iqbal Lone
e_mail: loneshahid@yahoo.com

- ***COURSE BOOK:***

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

- ***LIST OF REFERENCE MATERIAL:***

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

- ***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.

- ***PRE-REQUISITE:***

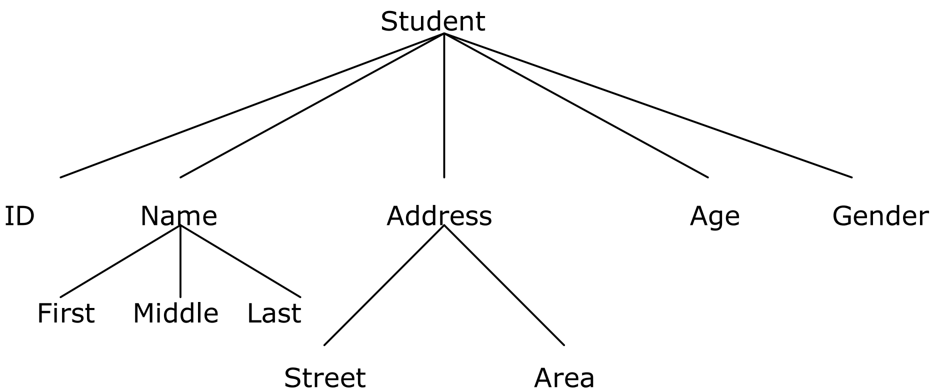
A good knowledge of c-language, use of Function and structures.

Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

Data:

Data are simply collection of facts and figures. Data are 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, a student's name may be divided into three sub items – [first name, middle name and last name] but the ID of a student would normally be treated as a single item.



In the above example (ID, Age, Gender, First, Middle, Last, Street, Area) are elementary data items, whereas (Name, Address) are group data items.

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**	**Gender**	**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:

In computer science, a data structure is a particular way of storing and organizing data in a computer's memory so that it can be used efficiently. 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 whenever necessary.

Created with nitro PDF professional
download the free trial online at nitropdf.com/professional

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 any sequence. There 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

$\mathbf{A}_1$, $\mathbf{A}_2$, $\mathbf{A}_3$ . . . . $\mathbf{A}_n$

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.

1	2	3	4	5	6	7	8
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]

```
int A[8] = {1, 2, 3, 4, 5, 6, 7, 8};
```

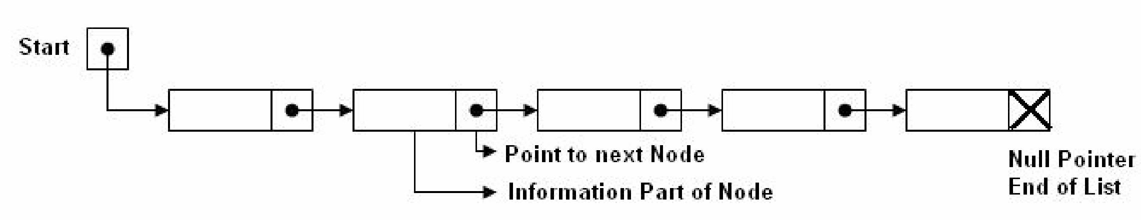
Linked List:

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.

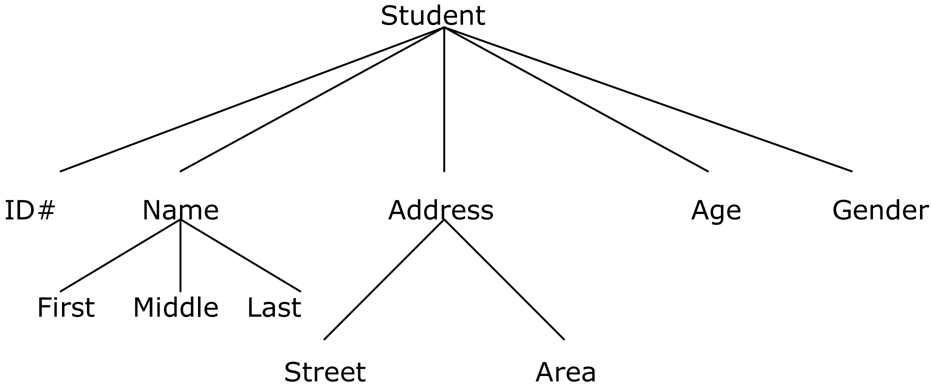
There is a special pointer ***Start/List*** contains the address of first node in the list. If this special pointer contains null, means that List is empty.

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 ***Font*** 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 Structures Operations:

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.

Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

ARRAYS IN C (Single Dimension)

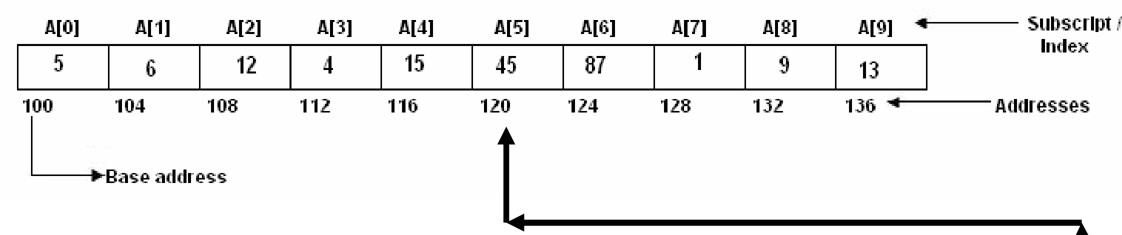
Arrays are preferred for situations which require similar type of data items to be stored together.

An array is a **finite** collection of **similar** elements stored in adjacent memory locations. By **finite** we mean 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 characters.

Thus an array is a 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**[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};
```



Representation of Linear Arrays in Memory

The memory of computer is simply a sequence of addressed locations as shown in Fig. Let **A** be a linear array stored in the memory of computer as given in above figure:

Add(**A**[k]) = address of the **A**[k] element of the array **A**

As the elements of the array **A** are stored in consecutive memory cells, the computer does not need to keep track of the address of every element of the array. It only needs to keep track of the address of the first element of the array which is denoted by:

Base(**A**)

It is called the base address of the **A**. Using this base address Base(**A**), the computer calculates the address of any element of the array by using the following formula:

Add(**A**[k]) = Base(**A**) + w(k-lower bound)

where w is the size of the data type of the array **A** and k is the index number.

Using the figure given on top of this page we are calculation the address of A[5]

Add(**A**[5]) = 100 + 4 (5 - 0)

= 120

 Shahid Iqbal (Lecturer) Computer College
Qassim University Kingdom of Saudi Arabia

Decleration of the Arrays: Any array declaration contains: the ***array name***, the *element* ***type*** 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 ARRAYS

A two-dimensional array is a collection of elements placed in **m** rows and **n** columns. There are two subscripts in the syntax of 2-D array in which one specifies the number of rows and the other the number of columns. In a two-dimensional array each element is itself an array.

The two-dimensional array is also called a **matrix**. Mathematically, a matrix A is a collection of elements 'a_{ij}' for all i and j's such that $0 \le i < M$ and $0 \le j < N$.

A matrix is said to be of order M × **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 a 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 arr[2][3] containing 2 rows and 3 columns (here M = 2 and N = 3) and arr[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 Fig.

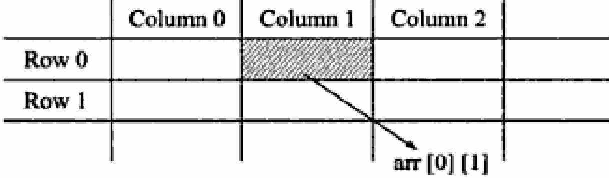


Fig. *Representation of 2-D array in memory*

A two-dimensional array differentiates between the logical and physical view of data. A 2-D array is a logical data structure that is useful in programming and problem solving. For example, such an array is useful in describing an object that is physically two-dimensional such as a map or checkerboard.

Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

Now let us calculate the address of an element in the following 2-D array:

For Example:

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

then how this 2-D array will be presented externally and in the memory of the computer using Row-Major-Order

A[0]	10	13	24	3
A[1]	41	5	6	17
A[2]	8	91	10	16

(A[0]) [0], (A[0]) [1], (A[0]) [2], (A[0]) [3]

Each elemen of A is four bytes long.

Low Addresses — High Addresses

Base Address →

100	104	108	112	116	120	124	128	132	136	140	144
10	13	24	3	41	5	6	17	8	91	10	16
A[0][0]	A[0][1]	A[0][2]	A[0][3]	A[1][0]	A[1][1]	A[1][2]	A[1][3]	A[2][0]	A[2][1]	A[2][2]	A[2][3]

10	13	24	3
41	5	6	17
8	91	10	16

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

Stored as Row-Major-Order

Here M=3 and **N** = 4

Low Addresses — High Addresses

Base Address →

100	104	108	112	116	120	124	128	132	136	140	144
10	13	24	3	41	5	6	17	8	91	10	16
A[0][0]	A[0][1]	A[0][2]	A[0][3]	A[1][0]	A[1][1]	A[1][2]	A[1][3]	A[2][0]	A[2][1]	A[2][2]	A[2][3]

Formula to calculate/find the address of A[j] [k]th element of a 2-D array of M × N dimension is

A[j] [k] = base (A) + W [N (j - Row_LowBound) + (k - Col_LowBound)]

where W = size of element
N = number of columns

Let us assume that the base address of the array matrix is 100. Since W= 4 (as array is of integer type whose size is 4), therefore, according to the formula, address of (2, 3)th element in the array matrix will be

```
LOC (A[2][3]) = 100 + 4 [ 4 ( 2 - 0 ) + ( 3 - 0 ) ]
              = 100 + 4 [ 8 + 3 ]
              = 100 + 4 [ 11 ]
              = 100 + 44
              = 144

LOC (A[1][0]) = 100 + 4 [ 4 (1 - 0 ) + ( 0 - 0 ) ]
              = 100 + 4 [ 4 + 0 ]
              = 100 + 4 [ 4 ]
              = 100 + 16
              = 116
```


Qassim University
Shahid Iqbal (Lecturer) Computer College
Qassim University Kingdom of Saudi Arabia

Stored as Column-Major-Order

10	13	24	3
41	5	6	17
8	91	10	16

Similarly, for the column major order representation, let us consider the same matrix.

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

Here M=3 and N = 4

A(j, k) = Base(A) + W[M(k−1) + (j−1)]

Low Addresses — High Addresses

Base Address →

100	104	108	112	116	120	124	128	132	136	140	144
10	41	8	13	5	91	24	6	10	3	17	16
A[0][0]	A[1][0]	A[2][0]	A[0][1]	A[1][1]	A[2][1]	A[0][2]	A[1][2]	A[2][2]	A[0][3]	A[1][3]	A[2][3]

Formula to calculate/find the address of A[j] [k]th element of a 2-D array of M × N dimension is

A[j] [k] = base (A) + W [M (k - Col_LowBound) + (j - Row_LowBound)]

where W = size of element

M = number of Rows

Let us assume that the base address of the array matrix is 100. Since W= 4 (as array is of integer type whose size is 4), therefore, according to the formula, address of (2, 3)th element in the array matrix will be

```
LOC (A[2][1]) = 100 + 4 [3 (1 - 0 ) + ( 2 - 0 ) ]
              = 100 + 4 [3 + 2 ]
              = 100 + 4 [5 ]
              = 100 + 20
              = 120

LOC (A[2][0]) = 100 + 4 [3 (0 - 0 ) + (2 - 0 ) ]
              = 100 + 4 [0 + 2 ]
              = 100 + 4 [2 ]
              = 100 + 8
              = 108
```

Dynamic Arrays

Dynamic array allocation is actually a combination of pointers and dynamic memory allocation. Whereas static arrays are declared prior to runtime and are reserved in stack memory, dynamic arrays are created in the heap using the ***new*** and released from the heap using ***delete*** operators.

Start by declaring a pointer to whatever data type you want the array to hold. In this case I've used int :

```
int *my_array;
```

This C++ statement simply declares an integer pointer. Remember, a pointer is a variable that holds a memory address. Declaring a pointer doesn't reserve any memory for the array - that will be accomplished with ***new***. The following C++ statement requests 10 integer-sized elements be reserved in the heap with the first element address being assigned to the pointer *my_array:*

```
my_array = new int[10];
```

The ***new*** operator is requesting 10 integer elements from the heap. There is a possibility that there might not be enough memory left in the heap, in which case your program would have to properly

handle such an error. Assuming everything went OK, you could then use the dynamically declared array just like the static array.

Dynamic array allocation is nice because the size of the array can be determined at runtime and then used with the ***new*** operator to reserve the space in the heap. To illustrate I'll uses dynamic array allocation to set the size of its array at runtime.

```
// array allocation to set the size of its array at runtime.
#include <iostream.h>
int main ()
{
  int i,n;
  int * p;
  cout << "How many numbers would you like to type? ";
  cin >> i;
  p= new int[i]; // it takes memory at run-time from Heap
  if (p == NULL)
    cout << "Error: memory could not be allocated";
  else
  {
    for (n=0; n<i; n++)
    {
      cout << "Enter number: ";
      cin >> p[n];
    }

      int *k=p; // to hold the base address of dynamic array
      cout << "You have entered: \n";
      for (n=0; n<i; n++)
      {  cout << *k<< ", "; k++;}
        cout<<"\n";
     delete[] p; // it release the memory to send it back to Heap
  }
  return 0;
}
```

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 (Ascending / Descending) is called sorting.
5- **Searching:** The process of finding the location of a particular element in an array is called searching. There are two popular searching techniques/mechanisms :
Linear search and binary search and will be discussed later.

a) Traversing in Linear Array:

It means processing or visiting each element in the array exactly once;
Let **'A'** is an array stored in the computer's 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≤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 (using for loop) 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    //  another way        int const  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;
}
```

Created with nitro PDF professional
download the free trial online at nitropdf.com/professional

b) 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 bigger value gradually bubbles their way up to the top of array like air bubble rising in water, while the small 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
2 1 5 7 4 3	1 2 5 4 3 7	1 2 4 3 5 7	1 2 3 4 5 7
1 2 5 7 4 3	1 2 5 4 3 7	1 2 4 3 5 7	1 2 3 4 5 7
1 2 5 7 4 3	1 2 5 4 3 7	1 2 4 3 5 7	1 2 3 4 5 7
1 2 5 7 4 3	1 2 4 5 3 7	1 2 3 4 5 7	
1 2 5 4 7 3	1 2 4 3 5 7		
1 2 5 4 3 7			

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

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 */*

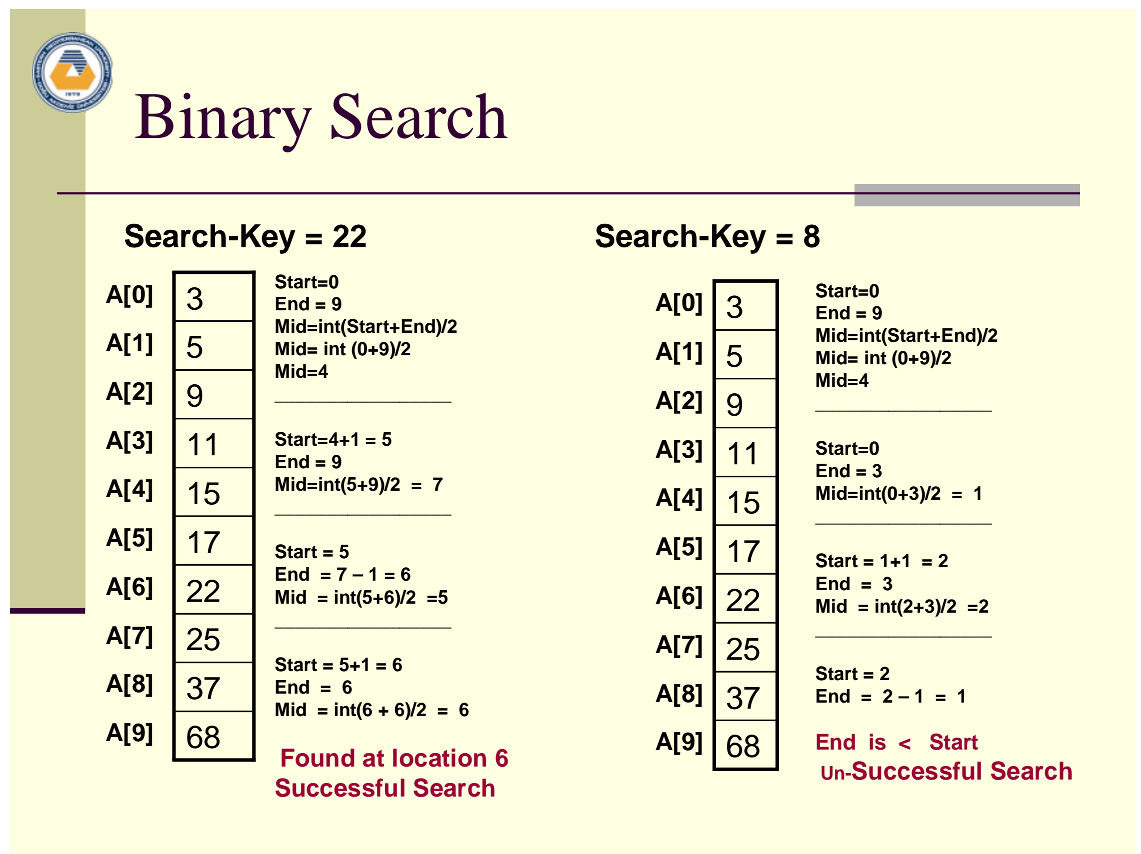
```
/* This program is an example of the Linear Search*/
#include <iostream.h>
int const N=10;
int LinearSearch(int [ ], int);   // Function Prototyping
int main()
 {  int A[N]= {9, 4, 5, 1, 7, 78, 22, 15, 96, 45}, Skey, LOC;
    cout<<" Enter the Search Key\n";
    cin>>Skey;
    LOC = LinearSearch( 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;
}
```


Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

Binary Search:

It is useful for the large sorted 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:



Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

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((START +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

```
// C++ Code  for Binary Search
#include <iostream.h>
int const 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:

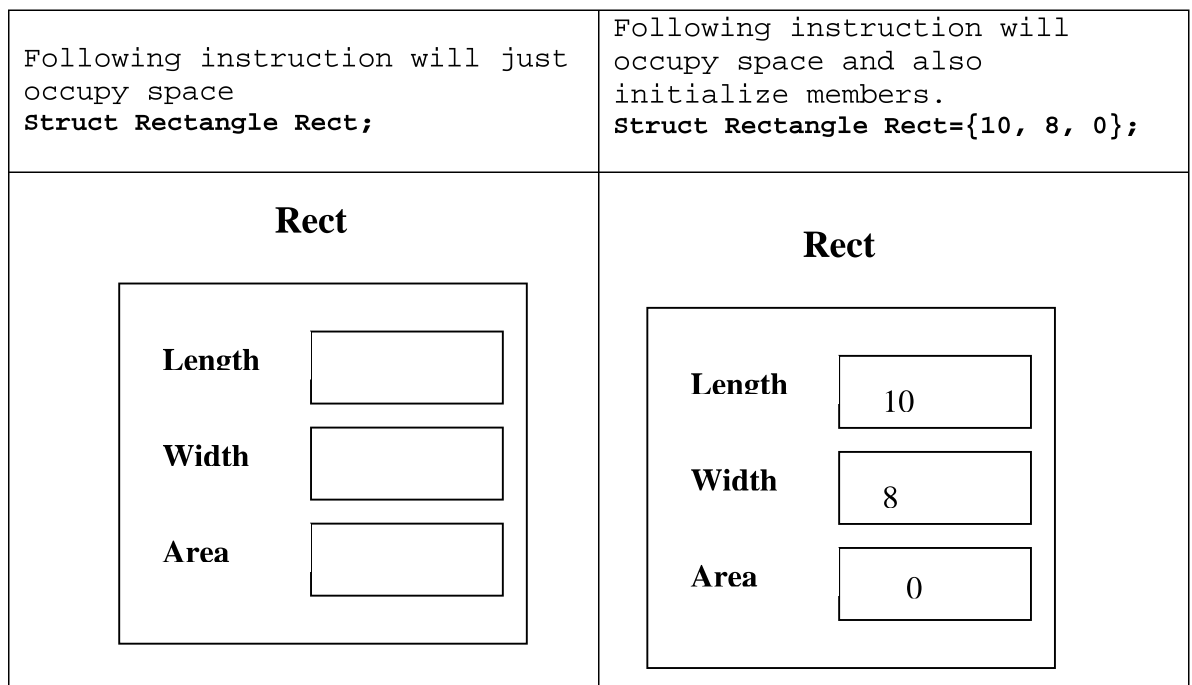
n-1 = 1023 (As many as 1023 comparisons may be required)

Structures A structure is a collection of logically related variables under a single unit/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 **Length** [ ] **Width** [ ] **Area** [ ]	**Rect** **Length** 10 **Width** 8 **Area** 0

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 / membership 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\n Enter the information for a student\n";
       cout << "  Name: ";
       cin.getline (student.name, 30, '\n');
       cout << "  Exam1: ";
       cin >> student.exam1;
       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.exam1 << "\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.exam1, 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\n Enter another student? (y/n)    ";
            cin >> more;
            //grab the carriage return since
            //character data is input next
            cin.ignore ( );
      } while (more == 'y' || more == 'Y');

      return 0;
}
```


Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

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;
```

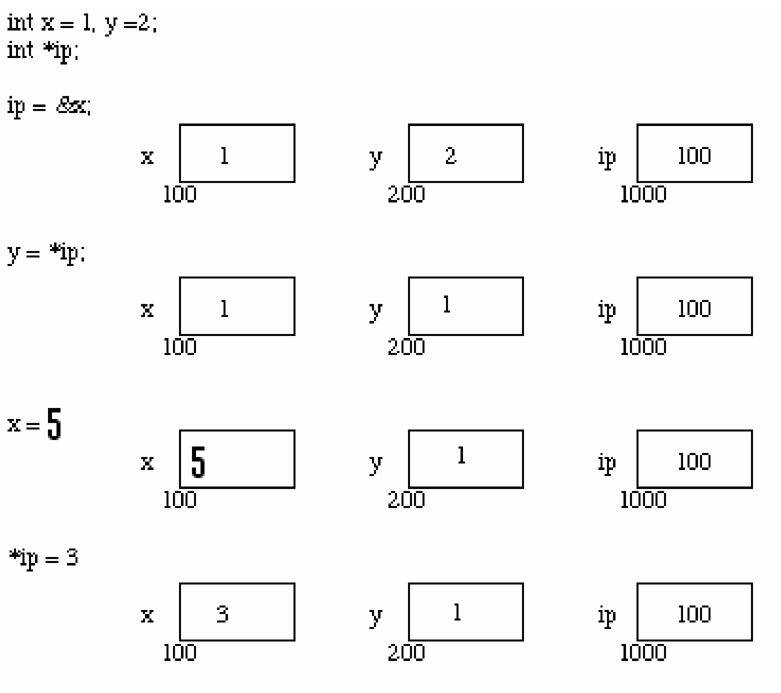
NOTE: We must associate a pointer to a particular type: You can't assign the address of a **short int** to a **long int**, for instance.

```
        long int   A=10;     short int  B= 5;     long int  *p = &A;  //   worng  assignment;
Similarly:         float   *T = &B;   //   wong  pointer assignment.
```

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).

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 <iostream.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<<endl;   //  it prints     8
   cout<< p1<<endl;    //   it   prints the address of    a
   D = *p1  -  b;      //   it    calculates    -2
   cout<< *p4<<endl;   // it  prints       -2
   *p5  =   a  * *p2;  //   it calculates    15
   cout<< M<<endl;  //  it prints    15
   return 0;
}
```

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

Pointers and Arrays

Pointers and arrays are very closely linked in C.
Hint: think of array elements arranged in consecutive / successive memory locations.
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 (due to *int)*. 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]<< ",  ";

    // 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<< ",  "; // 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 much 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 to 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:

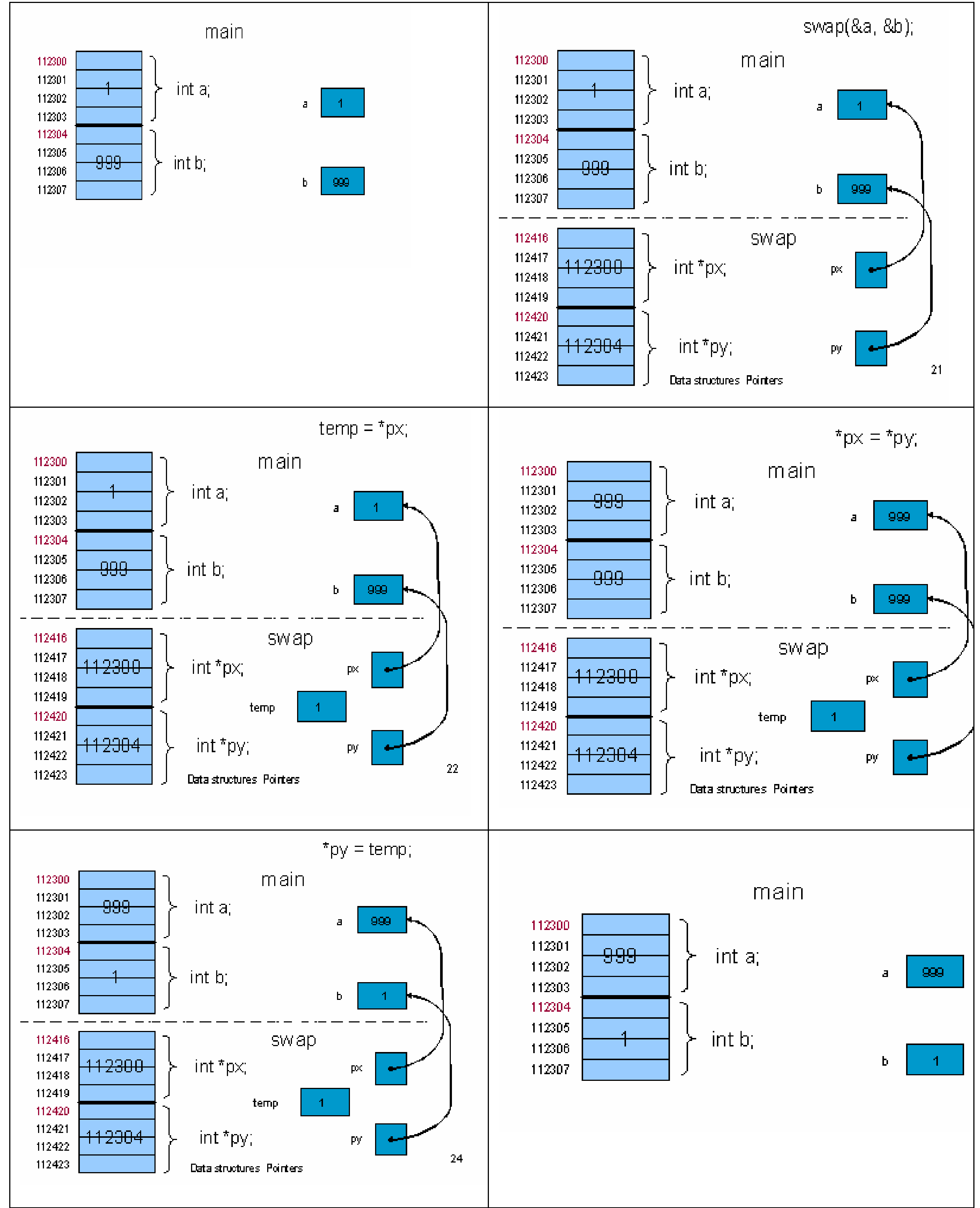
```
swap(&x, &y);
```

The Code to swap is fairly straightforward:

```
// This program swap / interchange the values of two variables
#include <iostream.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:



Shahid Iqbal (Lecturer) Computer College
Qassim University Kingdom of Saudi Arabia

Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

Pointers and Structures

1- Program to describe the pointer to stuctures

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

Rect

Length

Widt

Area

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

struct tag my_struct;           /* define the structure */
void show_name(struct tag *p);  /* function prototype */
int main( )
{
    struct tag *st_ptr;         /* a pointer to a structure */
    strcpy(my_struct.lname,"Shahid");
    strcpy(my_struct.fname,"Hamza");
    cout<<my_struct.fname<<endl;
    cout<<my_struct.lname<<endl;
    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<<endl;   /* p points to a structure */
    cout<<p->lname<<endl;
    cout<<p->age<<endl;
}
```

My_struct 2000

lname shahid

fname Hamza

age 19

st_ptr 2000

p 2000

Created with nitro PDF professional
download the free trial online at nitropdf.com/professional

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

the −> 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.
```

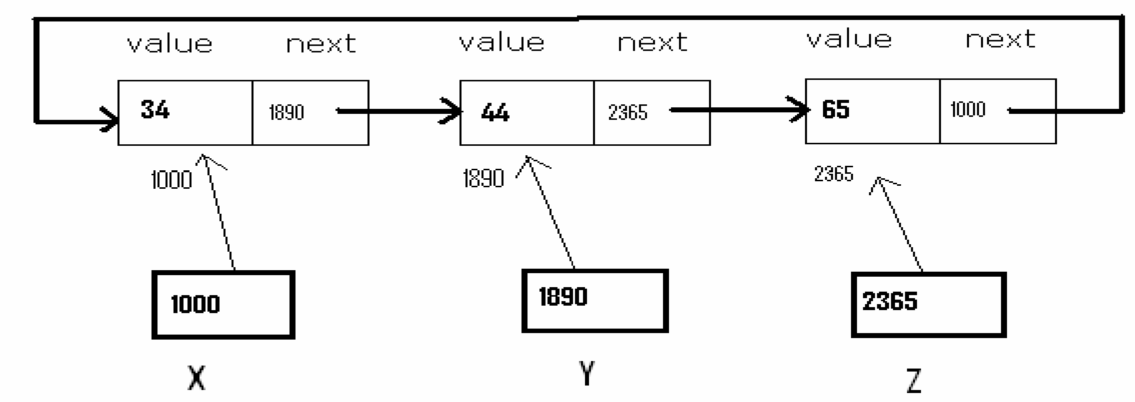
See another example, which is a little bit complex:

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

        // Allocate the pointees        In  Simple  c-language
        x = new (Node);  // (struct Node*) malloc(sizeof(Node));
        y = new (Node);  // (struct Node*) malloc(sizeof(Node));
        z = new (Node);  // (struct Node*) malloc(sizeof(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;
}
```



Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

Home Work

Exercise -1

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

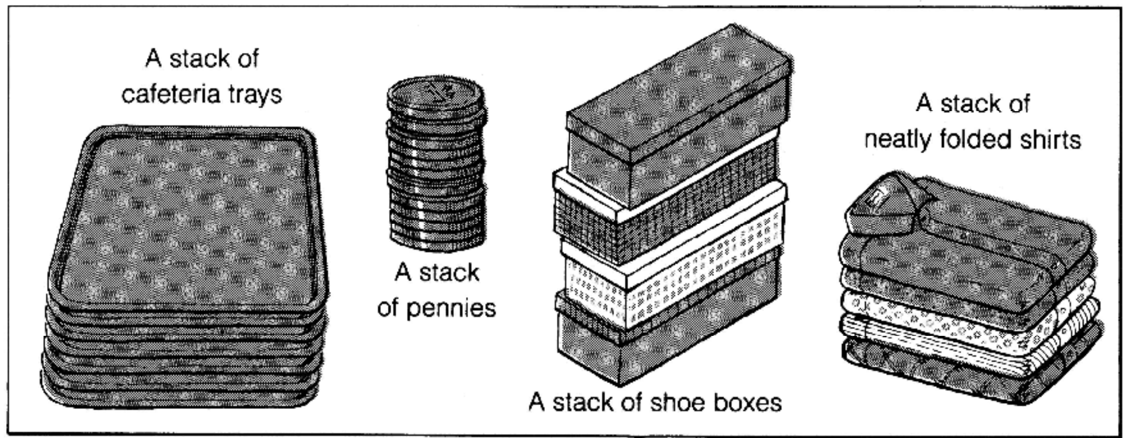
Exercise - 2

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 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).



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:

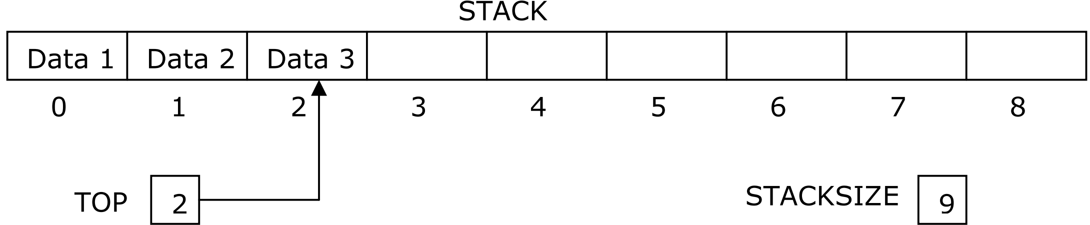
- ***Push()*** is the term used to insert/add an element into a stack.
- ***Pop()*** is the term used to delete/remove an element from a stack.

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

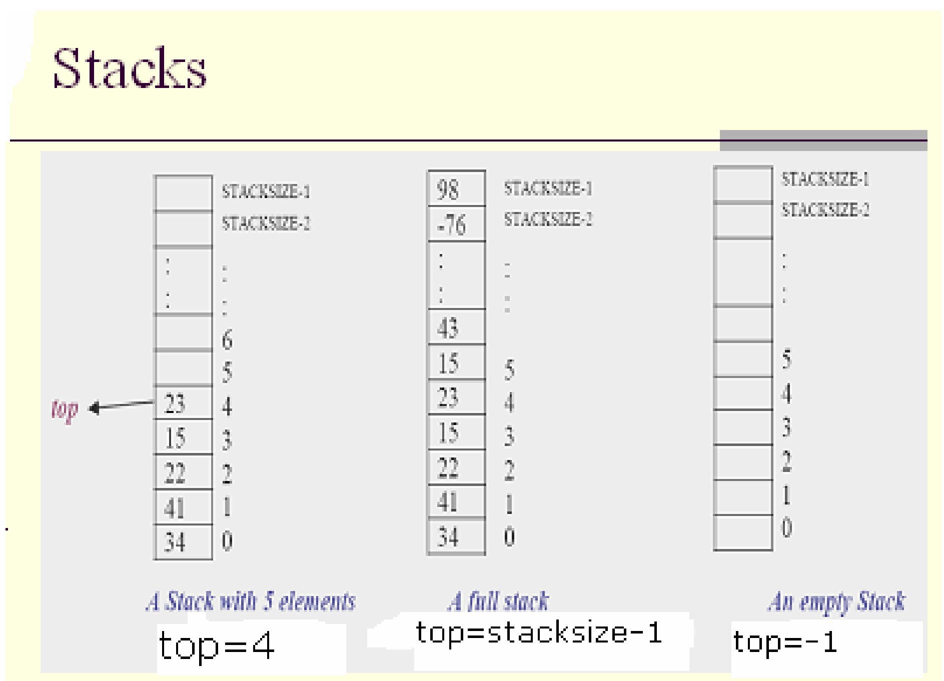
There are two ways to represent *Stack* in memory. One is using array and other is using linked list.

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.



Created with nitro PDF professional
download the free trial online at nitropdf.com/professional



Push Operation

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

Index	Before PUSH	After PUSH
STACKSIZE-1		
STACKSIZE-2		
:	:	:
:	:	:
6		
5		15
4	23	23
3	15	15
2	22	22
1	41	41
0	34	34

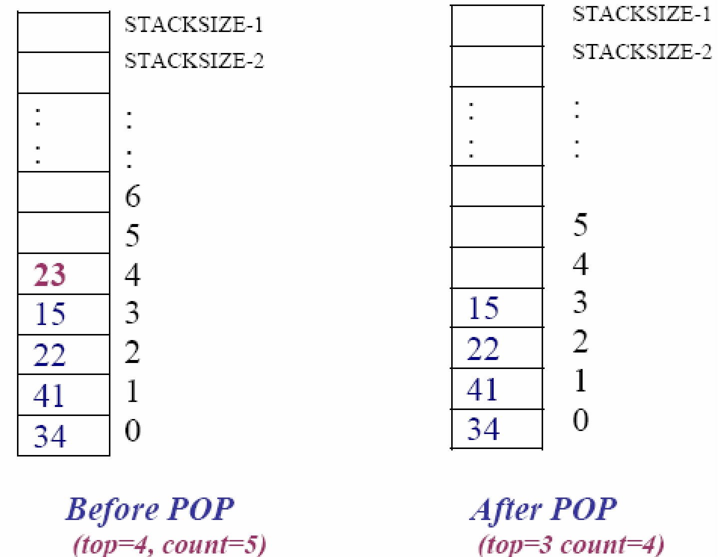
Before PUSH
(top=4, count=5)

After PUSH
(top=5, count= 6)

Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

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.

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

- `Push`: Places an element/value on *top* of the stack.
- `Pop`: Removes value/element from *top* of the stack.
- `IsEmpty`: Reports whether the stack is Empty or not.
- `IsFull`: Reports whether the stack is Full or not.

1. Run this program and examine its behavior.

```
// A Program that exercise the operations on Stack Implementing Array
// i.e. (Push, Pop, Traverse)
#include <conio.h>
#include <iostream.h>
#include <process.h>
#define  STACKSIZE  10    //   int const STACKSIZE = 10;
//  global  variable and array  declaration
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;
   while( 1 )
    {
      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:  exit(0);
        default:
                cout<< "\n\n\t Invalid Choice: \n";
       } // end of  switch block

      }  // end of while loop

 }  // end of of main() function


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--]<<endl;} while(TopTemp>= 0);
 }
```

1- Run this program and examine its behavior.

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

#include <iostream.h.h>
#include <process.h>

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

struct node *TOP = NULL;

void push (int x)
{ struct node *NewNode;
  NewNode = new (node);    // (struct node *) malloc(sizeof(node));

  if(NewNode==NULL) { cout<<"\n\n Memeory Crash\n\n";
                      return; }
   NewNode->info = x;
   NewNode->next = NULL;
```


Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

```
    if(TOP == NULL) TOP = NewNode;
   else
    { 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;
   while(1)
   { cout<<"\n\n\n\n\n\n    ***** Stack  Operations *****\n";
     cout<<"\n\n    1-  Push Item  \n    2- Pop Item \n";
     cout<<"    3- Traverse/Print stack-values\n    4- Exit\n\n";
     cout<<"\n   Your Choice --> ";
       cin>>ch;
       switch(ch)
       { case 1:
                    cout<<"\nPut a value: ";
                    cin>>item;
                    Push(item);
                    break;
            case 2:
                    if(IsEmpty()) {cout<<"\n\n Stack is Empty\n";
                                   break;
                                  }
                     T= Pop();
                     cout<< T->info <<"\n\n has been deleted \n";
                     break;
            case 3:
                     if(IsEmpty()) {cout<<"\n\n Stack is Empty\n";
                                    break;
                                   }
                        Traverse();
                        break;
            case 4:
                        exit(0);
       } // end of switch block
   }   // end of loop
   return 0;
} // end of main function
```

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 prescience 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 © 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 ©
   b) Add © 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*+**

Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

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 © 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** © **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 – 9 / 3

And its postfix is:

5 2 + 3 * 9 3 / –

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

5 2 + 3 * 8 4 / – **$**

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

Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

Following code will transform an infix arithmetic expression into Postfix arithmetic expression. You will also see the Program which evaluates a Postfix 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.h>
#include<conio.h>
#include<string.h>
int main()
{ int const null=-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
  int c=0;    // used as an  index  for  P
  int top=null;  // it assign   -1   to   top
  int k,i;
  cout<<"Put an arithematic INFIX _Expression\n\n\t\t";
  cin.getline(Q,99); // reads an 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
  stack[++top]='(';  // This statement will push first (  on  Stack

  while(top!= null)
    {
      for(i=0;i<=k;i++)
       {
        switch(Q[i])
        {
         case '+':
         case '-':
                for(;;)
                 {
                  if(stack[top]!='(' )
                  { P[c++]=stack[top--];n++; }
                  else
                   break;
                 }

               stack[++top]=Q[i];
               break;

         case '*':
         case '/':
         case '%':
                for(;;)
                 {if(stack[top]=='(' || stack[top]=='+' ||
                     stack[top]=='-') break;
                  else
                  { P[c++]=stack[top--]; n++; }
```


Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

```
                }

               stack[++top]=Q[i];
               break;

        case '^':
               for(;;)
                {
                 if(stack[top]=='(' || stack[top]=='+' ||
                    stack[top]=='-' || stack[top]=='/' ||
                    stack[top]=='*' || stack[top]=='%')  break;
                 else
                   { P[c++]=stack[top--]; n++; }
                }

               stack[++top]=Q[i];
               break;


       case '(':
              stack[++top]=Q[i];
              break;


       case ')':
              for(;;)
               {
                if(stack[top]=='(' )  {top--; break;}
                else   { P[c++]=stack[top--]; n++;}
               }
               break;

       default :   //  it  means that read  item is an oprand
              P[c++]=Q[i];
               n++;
     } //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\nPOSTFIX EXPRESION IS \n\n\t\t"<<P<<endl;

} //END OF MAIN FUNCTION
```


Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

```
// 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 raise-to-the-power

#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include<math.h>
#include <stdlib.h>
#include <ctype.h>

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

struct node *TOP = NULL;

void push (int x)
{  struct node *Q;
              // in c++        Q = new node;
   Q = (struct node *) malloc(sizeof(node)); // creation of new node
   Q->info = x;
   Q->next = NULL;
   if(TOP == NULL) TOP = Q;
   else
    { Q->next = TOP;
     TOP=Q;
    }
}

struct node* pop ()
{ struct node *Q;
    if(TOP==NULL)  { cout<<"\nStack is empty\n\n";
                     exit(0);
                   }
    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 arithmatic expression  end  with $: \n ";
     while(1)
      { t=getche(); // this will read one character and store it in t
      if(isdigit(t)) push(t-'0'); // this will convert char  into int
      else if(t==' ')continue;
      else if(t=='$') break;
```

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

      case '-':
              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<<"Error unknown operator";
             } //  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\nThe result of this expression is = "<<Q->info<<endl;
      return 0;
} //  end of  main function
```


Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

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.
        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, then: [Circular, so set Front = 0 ]
                       Set FRONT = 0
               Else:
                       Set FRONT:=FRONT+1.
               [End of If structure.]

        5. Return ITEM
```

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

Step	Pointers	0	1	2	3	4	5
(a) Initially QUEUE is Empty	Front = -1, Rear = -1, Count = 0						
(b) A, B, C are Enqueued / Inserted	Front = 0, Rear = 2, Count = 3	A	B	C			
(c) A is Deleted / Dequeue	Front = 1, Rear = 2, Count = 2		B	C			
(d) D, E, F are Enqueued / Inserted	Front = 1, Rear = 5, Count = 5		B	C	D	E	F
(e) B and C are Deleted / Dequeue	Front = 3, Rear = 5, Count = 3				D	E	F
(f) G is Enqueued / Inserted	Front = 3, Rear = 0, Count = 4	G			D	E	F
(g) D and E are Deleted / Dequeue	Front = 5, Rear = 0, Count = 2	G					F
(h) H and I are Enqueued / Inserted	Front = 5, Rear = 2, Count = 4	G	H	I			F

Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

(i) F is Deleted / Dequeue

Front = 0
Rear = 2
Count = 3

G	H	I			
0	1	2	3	4	5

(j) G and H are Deleted / Dequeue

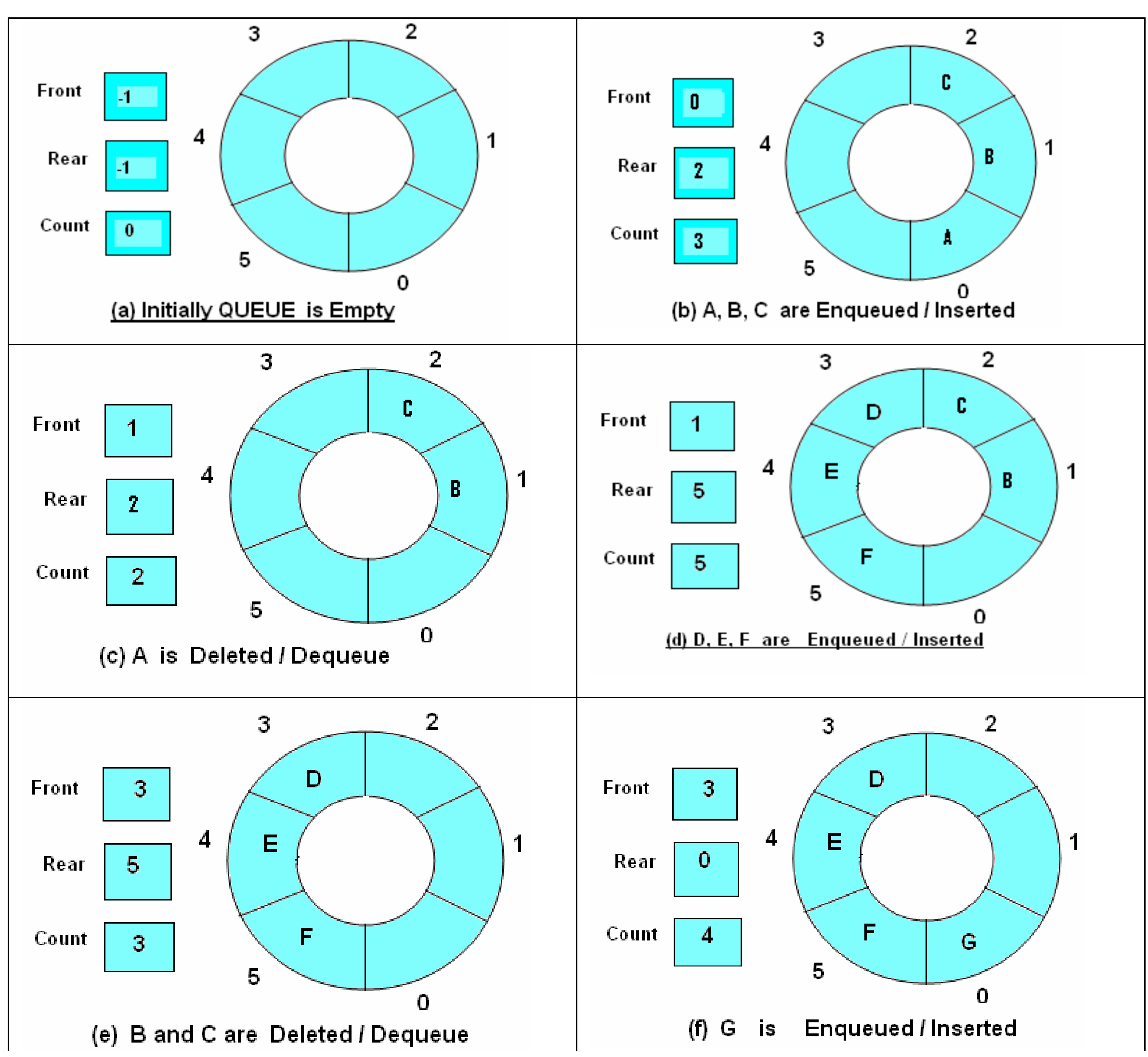
Front = 2
Rear = 2
Count = 1

		I			
0	1	2	3	4	5

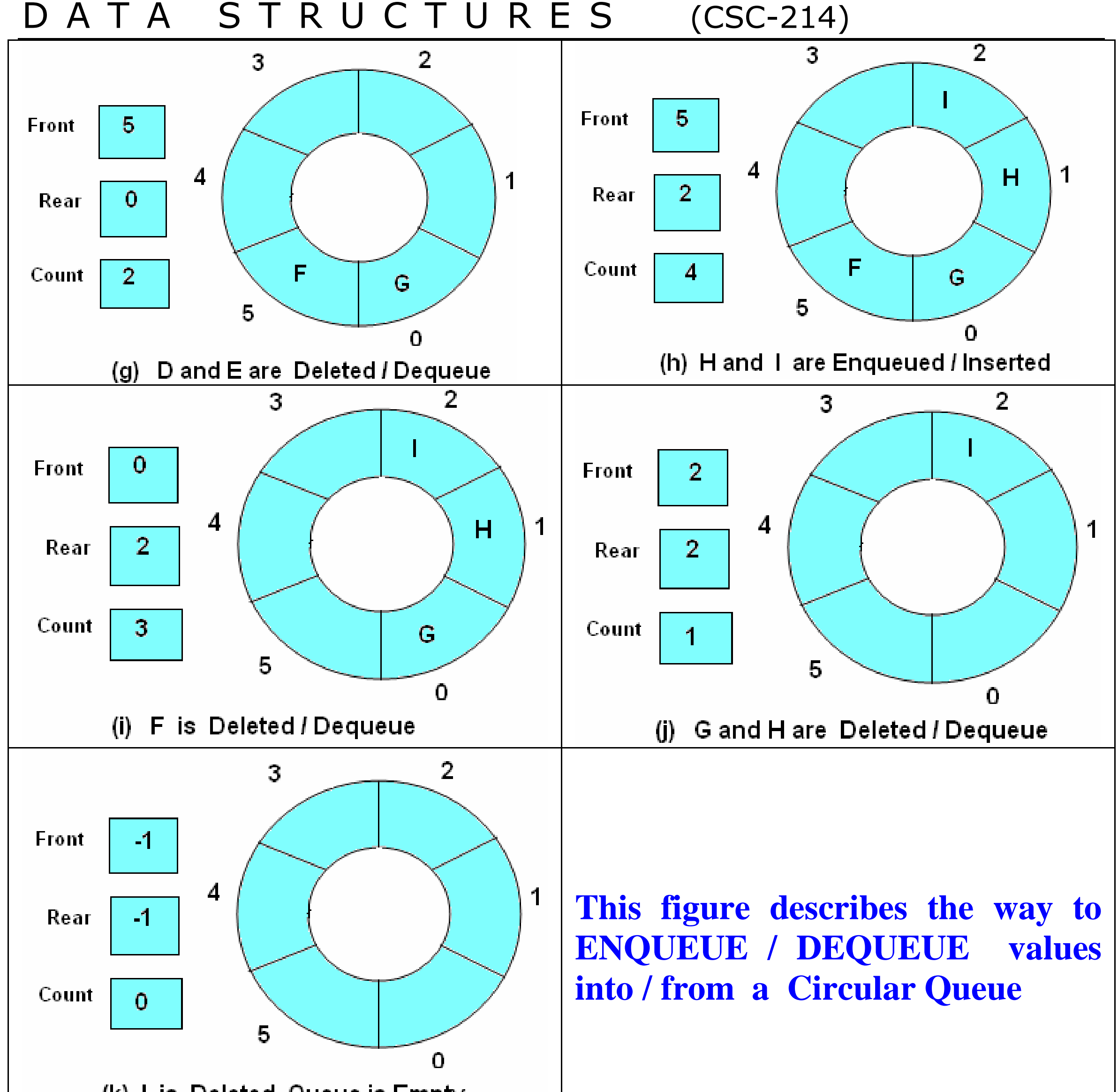
(k) I is Deleted. Queue is ***Empty***

Front = -1
Rear = -1
Count = 0

0	1	2	3	4	5



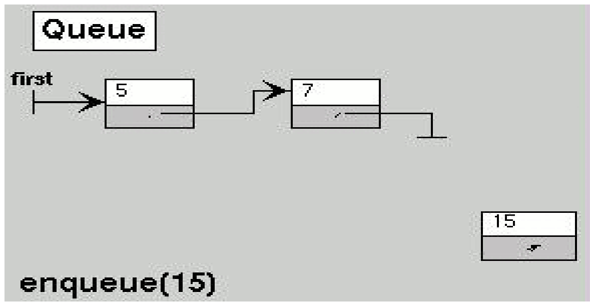
Created with nitroPDF professional
download the free trial online at nitropdf.com/professional



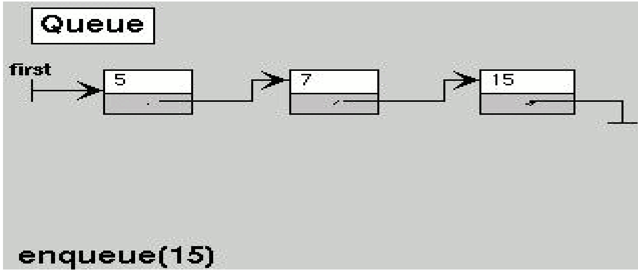
This figure describes the way to ENQUEUE / DEQUEUE values into / from a Circular Queue

Created with nitro PDF professional
download the free trial online at nitropdf.com/professional

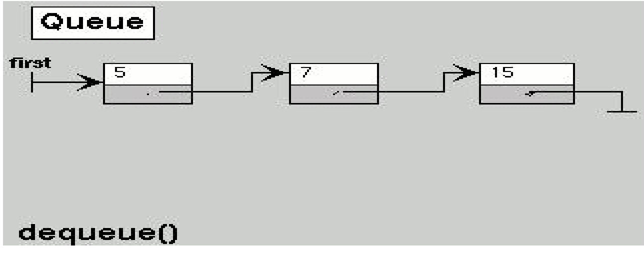
Following is the graphical presentation of Queue using pointers



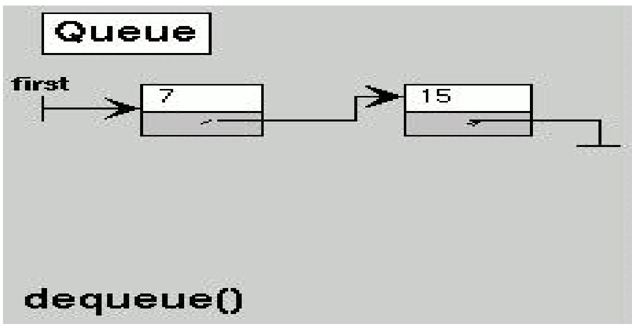
Before inserting a new element



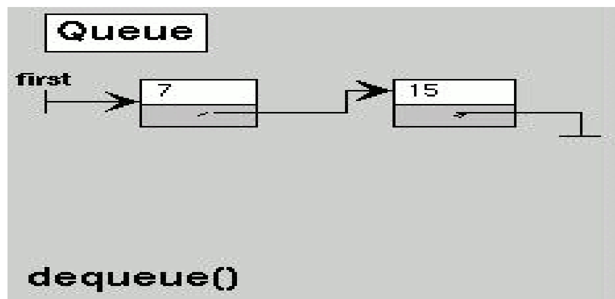
After inserting element 15



Before removing the first element



After removing the first lement



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

```
#include<iostream.h>
#include <process.h>
#define MAXSIZE 10    // int const  MAXSIZE = 10;

//   Global  declarations and available to every
      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++;

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

int Dequeue()
{
      if(IsEmpty()) { cout<<"\n\nQUEUE is empty\n"; 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()
      { int i;
        if(IsEmpty()) cout<<"\n\nQUEUE is empty\n";
        else
          { 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;
  while(1)
      {
        cout<<"\n\n\n\n QUEUE operation\n\n";
        cout<<"1-insert value \n 2-deleted value\n";
        cout<<"3-Traverse QUEUE \n 4-exit\n\n";
        cout<<"\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<<t<<  " deleted \n";
                  break;
      case 3:
                  cout<<"\n queue state\n";
                  Traverse(); break;

      case 4:exit(0);
      }
}
      return 0;
}
```

```
// A Program that exercise the operations on QUEUE
// using POINTER (Dynamic Binding)
// Programed by SHAHID LONE

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

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

QUEUE *rear=NULL, *front=NULL;

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

void main(void)
{ int ITEM, choice;
  while( 1 )
    {
      cout<<"     *******  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);
```


Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

```
      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:   exit(0);
                    break;

          default:  cout<<"\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;
      // in c++     NewNode = new QUEUE;
      NewNode = (struct QUEUE *) malloc( sizeof(struct 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;
                    }
   int ITEM = front->val;
    if(front == rear ) front=rear=NULL;
    else front = front-> pNext;
   return(ITEM);
}
void Traverse(void)
{ if(front == NULL) {cout<< " \n <Underflow>  QUEUE is empty\n";
                     return; }
  QUEUE f = front;
   while(f!=rear)
    { cout front->val << ", ";
      f=f->pNext;
    }
}
```

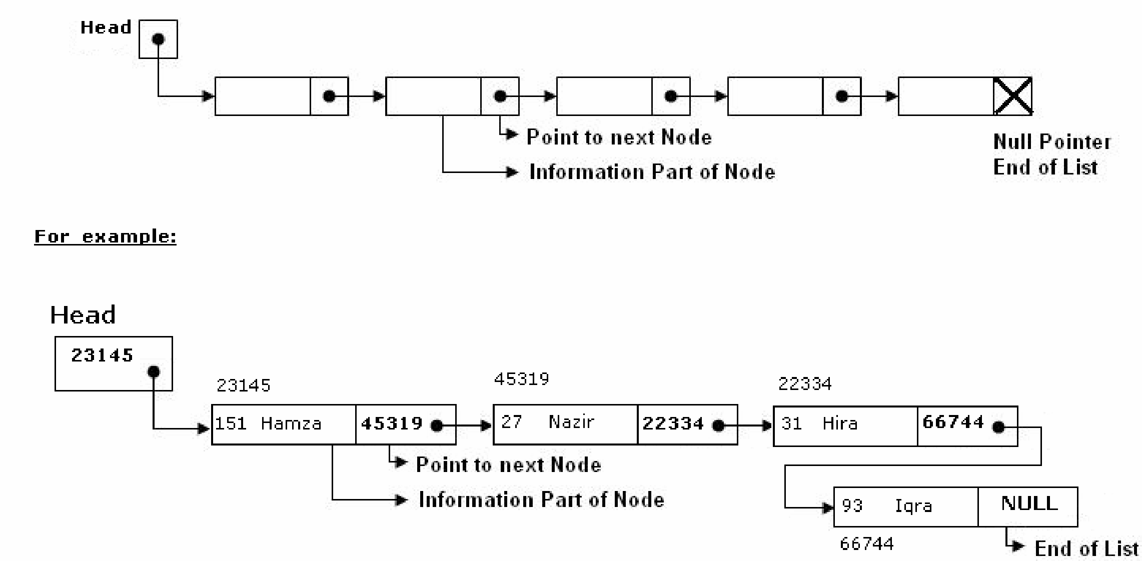

Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

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:



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:

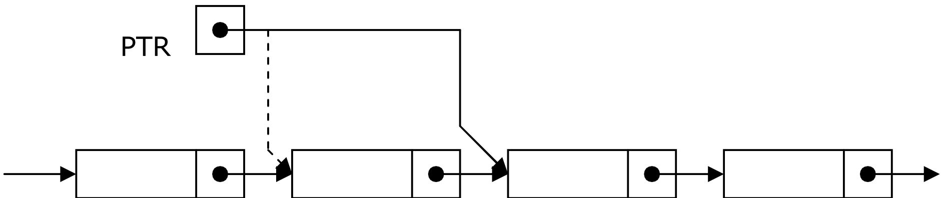
There are several operations associated with linked list i.e.

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,

Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

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 point to the node currently being processed.

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

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, PREV, CURR, SCAN)
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 and 4 while PTR≠NULL:
3. if ITEM = PTR->INFO then:
   Set LOC=PTR, and return. [Search is successful.]
   [End of If structure.]
4. Set PTR=PTR->NEXT
   [End of Step 2 loop.]
5. Set LOC=NULL, and return. [Search is unsuccessful.]
6. Exit.

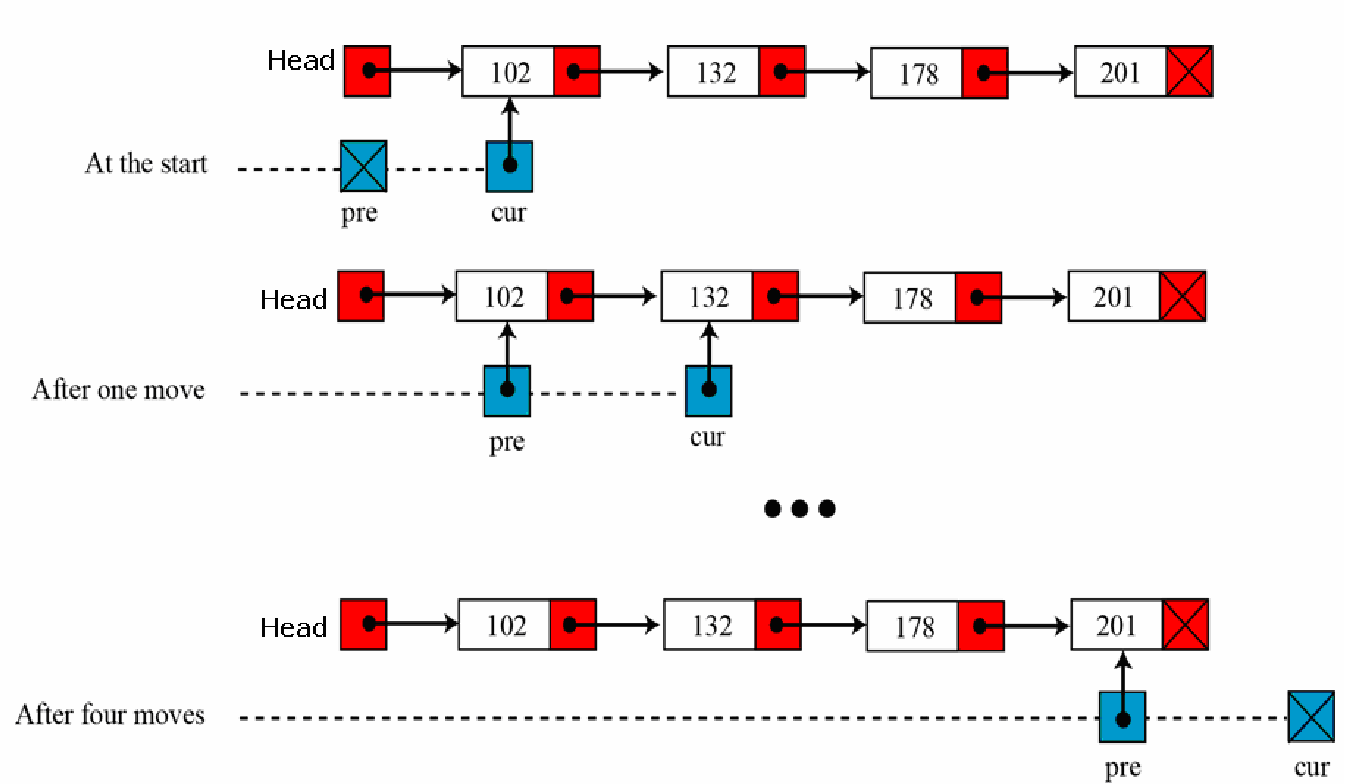
Shahid Iqbal (Lecturer) Computer College
Qassim University Kingdom of Saudi Arabia

Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

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).

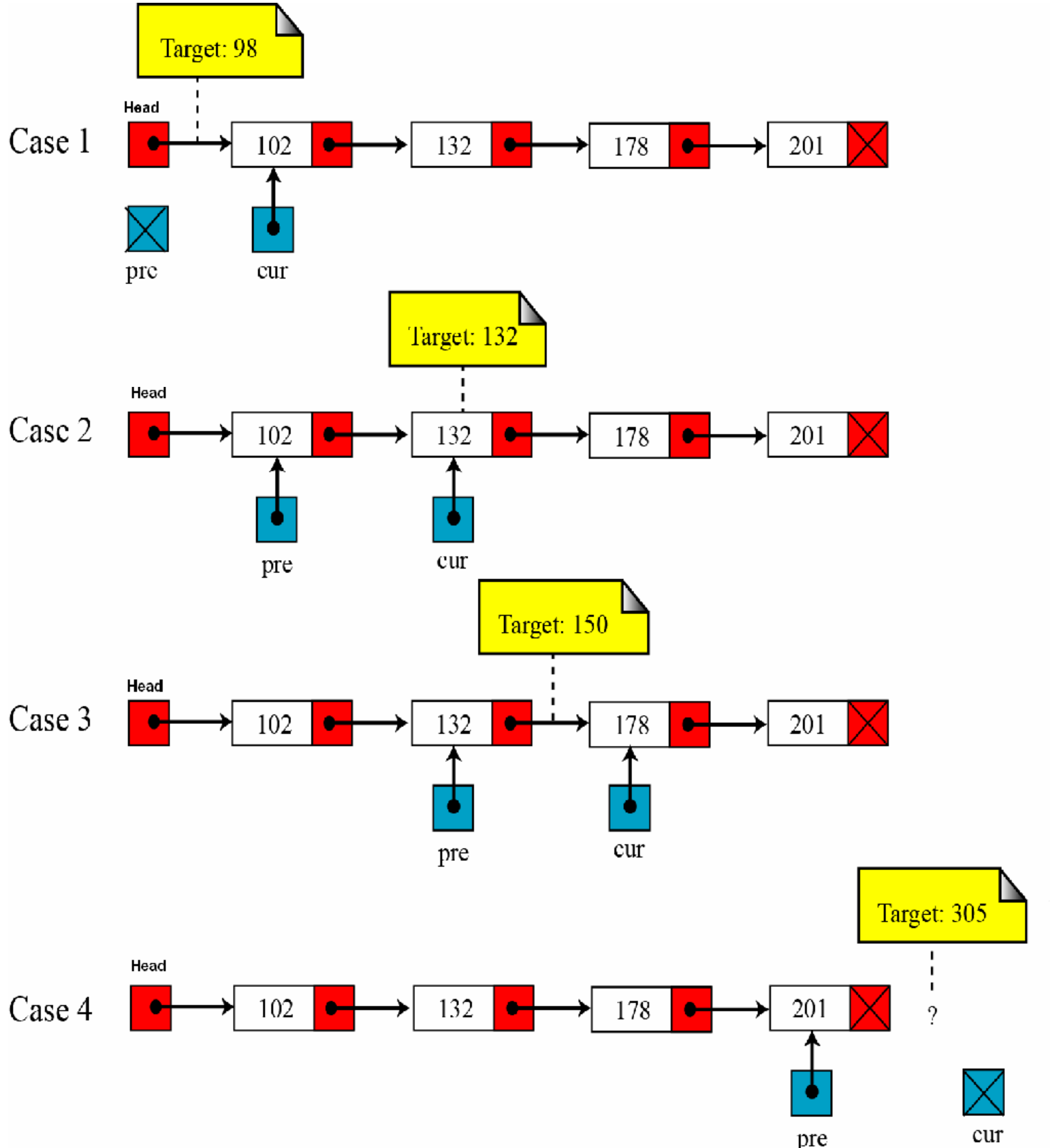
Basic Search Concept
Assume there is a sorted linked list and we wish that after each insertion/deletion this list should always be sorted. Given a target value, the search attempts to locate the requested node in the linked list.
Since nodes in a linked list have no names, we use two pointers, **pre** (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

Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

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

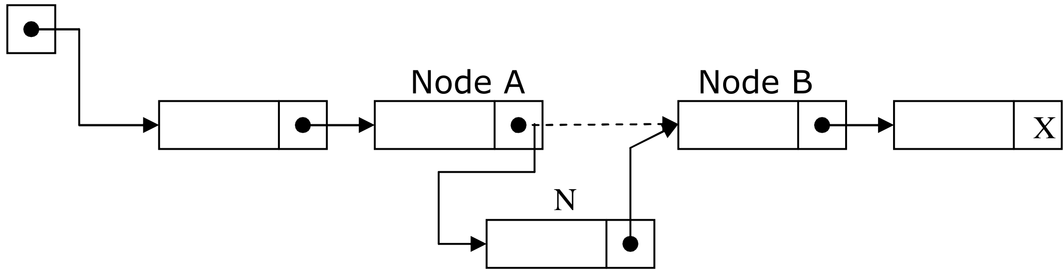


Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

Insertion into a Linked List:

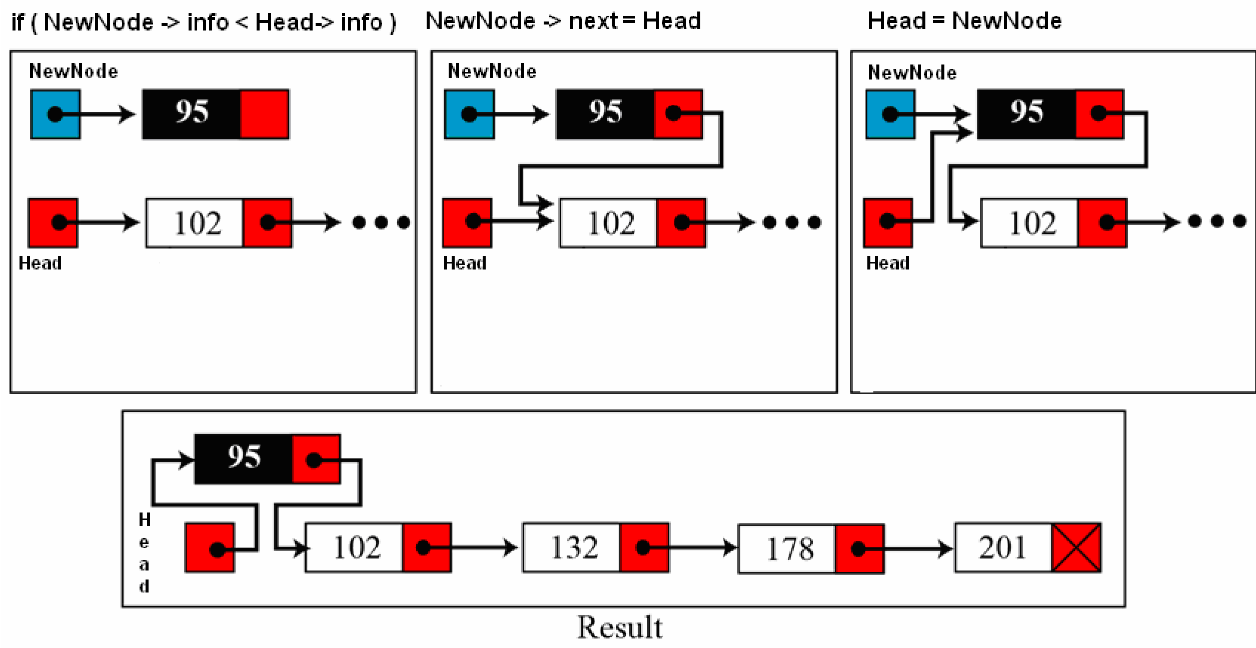
If a node N is to be inserted into the list between nodes **A** and **B** in a linked list named LIST.

Its schematic diagram would be;



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->info < Head->info)*** then new node is inserted at the beginning / Top of the list.



Result

Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

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(CurrNode =HEAD; CurrNode != NULL; CurrNode = CurrNode ->NEXT)
```
{ if(NewNode->INFO <= CurrNode ->INFO)
    {
        break the loop
    }
  PrevNode = CurrNode;
} [ end of loop ]
```
   [Insert after PREV node (in middle or at end) of the list]
8. Set **NewNode**->NEXT = PrevNode->NEXT and
9. Set PrevNode->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.
4. for(CurrNode =HEAD; CurrNode != NULL; CurrNode = CurrNode ->NEXT)
```
{ if (ITEM = CurrNode ->INFO ) then:
    {
        break the loop
    }
  Set PrevNode = CurrNode;
} [ end of loop ]
```
5. if(CurrNode = NULL) then write : Item not found in the list and return
6. [delete the current node from the list]
   Set PrevNode ->NEXT = CurrNode->NEXT
7. Exit.

Created with nitro PDF professional
download the free trial online at nitropdf.com/professional

```
// A Program that exercise the operations on Liked List
// Programed by SHAHID LONE
#include<iostream.h>
#include <malloc.h>
#include <process.h>

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

struct node *Prev,*Curr;

void AddNode(int ITEM)
{
  struct node *NewNode;
  NewNode = new node;
  //   NewNode=(struct node*)malloc(sizeof(struct node));
  NewNode->info=ITEM;  NewNode->next=NULL;
  if(Head==NULL) { Head=NewNode; return; }
  if(NewNode->info < Head->info)
      { NewNode->next = Head; Head=NewNode; return;}

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

  }//  end of DeleteNode function
```


Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

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

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

int main()
{ int inf, ch;
  while(1)
  { cout<< " \n\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: exit(0);
      }  // end of switch
  } //  end of while loop
 return 0;
} // end of main ( ) function
```


Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

```
// Program designed by SHAHID IQBAL LONE    Qassim University  K.S.A
// This program describes the basics about Read/Write structures into
//  a file.  The file which is going to be used is a binary file.
//  The logic is already discussed in the Class. Students should try to
//  run this program and discuss with me if they have some queries

#include<iostream.h>
#include<fstream.h>
#include<stdlib.h>             // for exit keyword
#include<iomanip.h>
#include<conio.h>

// **********  GLOBAL STRUCTURE TEMPLATE  *****************/
 struct  student
      { int id;
          char name[20];
          int m1;
          int m2;
          int m3;
      };
//***************  GLOBAL  VARIABLES  ***************/

   const int MAX=50;   // size available in the array
   int n= -1;    // used as index for insertion of new records in array
   student Data[MAX];  // Array to hold all the records
   int page_no;    // used for page numbers with headings in Report

/*********************  FUNCTIONS PROTOTYPES  *******************/

void LoadList();
void AddStudent(void);
void SaveList(void);
void DisplayList(void);
void Heading(void);

/*************************MAIN FUNCTION****************************/

void main(void)
{
      short int Choice=0;
      LoadList();
      while( 1 )
      {   system ("cls"); // this statement clears the screen
            cout<<"                  Main Menu"<<endl;
            cout<<"                  ~~~~~~~~~"<<endl;
            cout<<"\n\t[1]  Add a New Students's Record";

            cout<<"\n\n\t[2]  Display Students List";
            cout<<"\n\n\t[3]  Store   Students List into File";
            cout<<"\n\n\t[4]  Exit Program";
            cout<<"\n\n\n\t\tEnter your choice --> ";
            cin>>Choice;

            switch( Choice )
            {
            case 1:
```


Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

```
                        AddStudent( );
                        break;

            case 2:     page_no=0;
                      DisplayList( );
                      break;

            case 3:     SaveList( );
                        break;
            case 4:
                        SaveList( );
                        cout<<"\n\n\n GoodBye!!!\n";
                        exit(0);
            default:
                        cout<<"\nInvalid Choice...";
            }

      }

}  // end of main() function


/******************  FUNCTION  DEFINITIONS  ********************/

void AddStudent(void)
{


    if(n==MAX-1)
            { cout<< "\n\n [ OverFlow !! ]";
              cout<< "\n [ can not store new record ]\n\n\\n";
              return;
            }
       n++;
       cout<<"\n\n\n\n\n\t\t DATA FOR  "<<n+1<<" STUDENT\n\n";
       cout<< " PUT ID: "; cin>> Data[n].id;
       cout<< " PUT NAME: "; cin.ignore(); // clears input buffer
       cin.getline(Data[n].name, 20);
       cout<< " put marks of three subjects separated by space: ";
       cin>>Data[n].m1>>Data[n].m2>>Data[n].m3;

}  // End of function

void SaveList(void)
{     ofstream pFile;
    pFile.open("student.dat",ios::binary);
      if(pFile==NULL)
            {
            cout<<"Cannot Open File \n Data not saved into file\n\n";
                  exit(0);
            }

         for (int j=0; j<=n ; j++ )
                     pFile.write((char*) &Data[j],sizeof(Data[j]));
         pFile.close();
}
```

```
void LoadList( )
{    ifstream piFile;
     piFile.open("student.dat",ios::binary);
      if(piFile==NULL) {cout<< "New file will be created\n"; return;}

      n++;
      // Read First Record from file
      piFile.read( (char*) &Data[n],sizeof(Data[n]));

      while( piFile)    // if(pFile.eof( ))break;
      {      n++;

        piFile.read( (char*) &Data[n],sizeof(Data[n]));

      }
      n--;
      piFile.close();
}

void  DisplayList(void)
 {
      if(n == -1)
            { cout<< "\n\nUnderFlow !! [Empty Array] ";
              cout<< "\n Nothing to Display \n\n";
              return;
            }
     int i=0,tot;
     double per;
     char grade;
     Heading( );

     while (i<=n)
          { if( i % 20 ==  0 && i != 0 )
               { cout<< "\n\n Press a key for next Page: .... ";
                 Heading( )
               }
            tot=Data[i].m1 + Data[i].m2 + Data[i].m3;
            per = tot * 100.0 / 300.0;
            if(per >= 80.0) grade='A';
             else if(per>= 70.0) grade='B';
            else if(per>= 60.0) grade= 'C';
             else if(per>= 50.0) grade= 'D';
              else grade='F';

              // print record
      cout<<setw(5)<<Data[i].id<<setw(3)<<" "<<setw(20)
      <<setiosflags(ios::left)<<Data[i].name<<setw(6)
      <<resetiosflags(ios::left)<<Data[i].m1<<setw(9)<<Data[i].m2
      <<setw(9)<<Data[i].m3<<setw(8)<<tot<<setw(10)
      <<setiosflags(ios::fixed)<<setiosflags(ios::showpoint)
      <<setprecision(2)<<per<<setw(5)<<grade<<endl;

      i++;
      } // end of while loop
     system("pause"); //  it makes halt and need to press any key
}
```


Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

```
void Heading( )
{
     system ("cls"); // this statement clears the screen
     page_no++;
       // Print heading
       cout<<"\t\t\t\tStudents Records List \t\t  Page-No: "<<
       page_no<<endl;
       cout<<"\t\t\t\t~~~~~~~~~~~~~~~~~~~~~~\n\n";
     cout<<setw(8)<<"   ID.  "<<setw(20)<<"      N A M E       ";
     cout<<setw(9)<<" Marks-1 "<<setw(9)<<" Marks-2 " <<setw(9)
     <<" Marks-3 "<<setw(7)<<" Total "<<setw(9)<<"  Per%  "<<setw(6)
     <<" Grade";
      cout<<endl<<endl;;
}
```

Home Work:

Students have to change the above program to store/write the nodes of a linked list into a file and reverse read nodes from file to create a linked list in memory.

Tree:

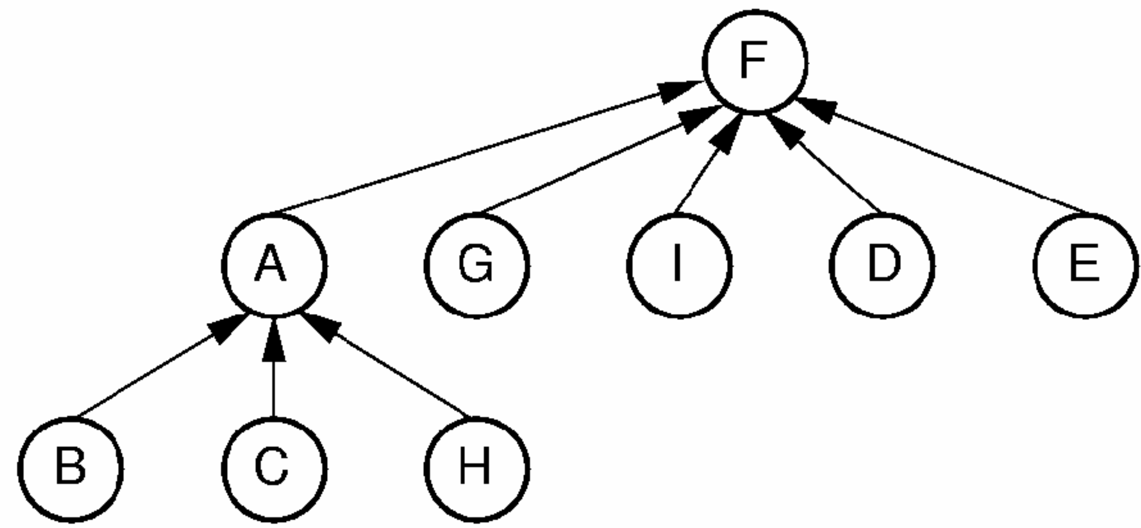
So far, we have been studying mainly linear types of data structures: arrays, lists, stacks and queues. Now we defines a nonlinear data structure called ***Tree.*** This structure is mainly used to represent data containing a hierarchical relationship between nodes/elements e.g. family trees and tables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

There are two main types of tree:

- General Tree
- Binary Tree

General Tree:

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



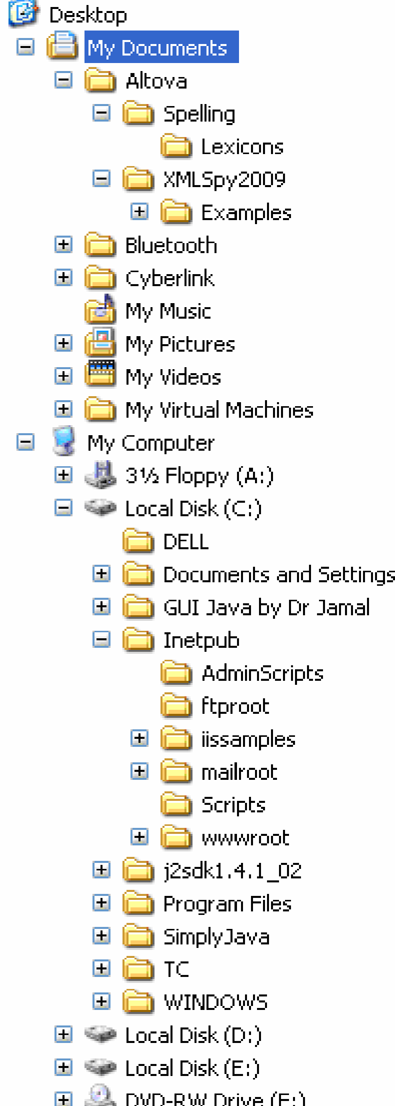
This following figure is a general tree where *root* is "*Visual Programming*".

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 has 0-child , 1-child or maximum of 2-children. A *Binary* Tree **T** is defined as 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 $\mathbf{T_1}$ and $\mathbf{T_2}$.

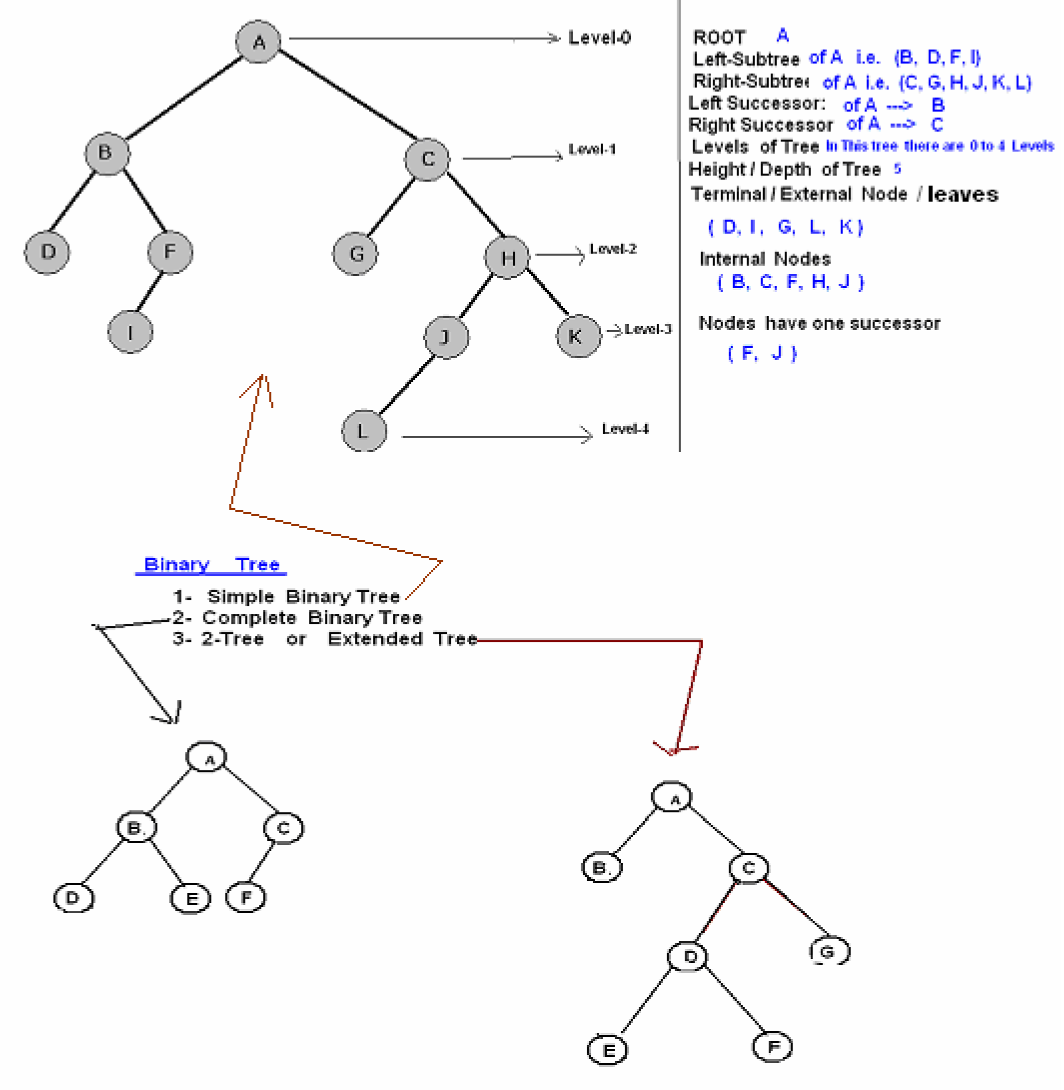
If **T** does contain a *root* **R**, then the two trees $\mathbf{T_1}$ and $\mathbf{T_2}$ are called, respectively, the **left sub tree** and **right sub tree** of **R**.

If T_1 is non empty, then its node is called the left successor of **R**; similarly, if T_2 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 S_1 and right successor S_2, then N is called the **parent(or father)** of S_1 and S_2. Analogously, S_1 is called the left child (or son) of N, and S_2 is called the right child (or son) of N. Furthermore, S_1 and S_2 are said to **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 **leaves**, and a path ending in a leaves 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.

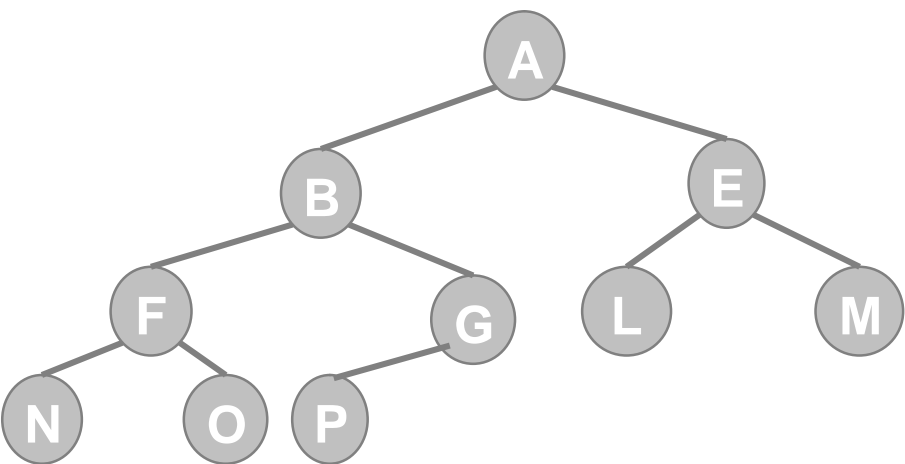


Shahid Iqbal (Lecturer) Computer College
Qassim University Kingdom of Saudi Arabia
Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

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.

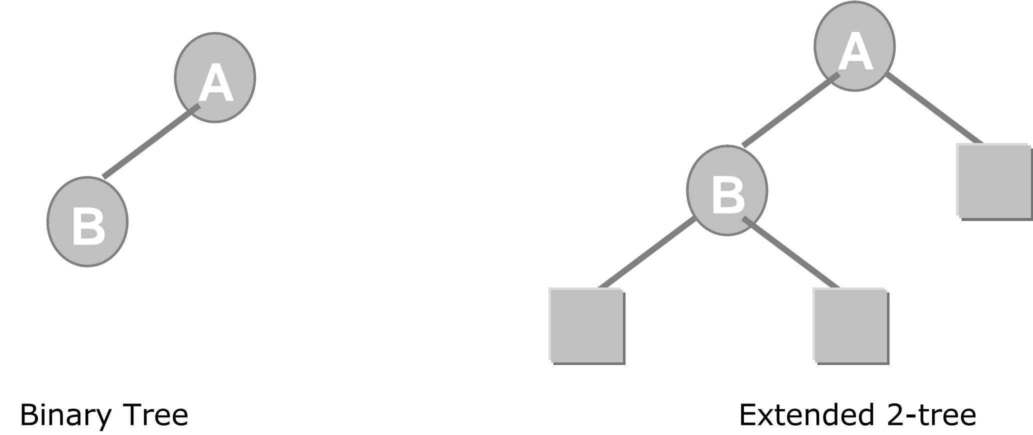
*A tree is said to be **complete** if all its levels, except possibly the last have the maximum number of possible nodes, and if all the nodes at the last level appear as far left as possible.*



Extended Binary Tree: 2-Tree:

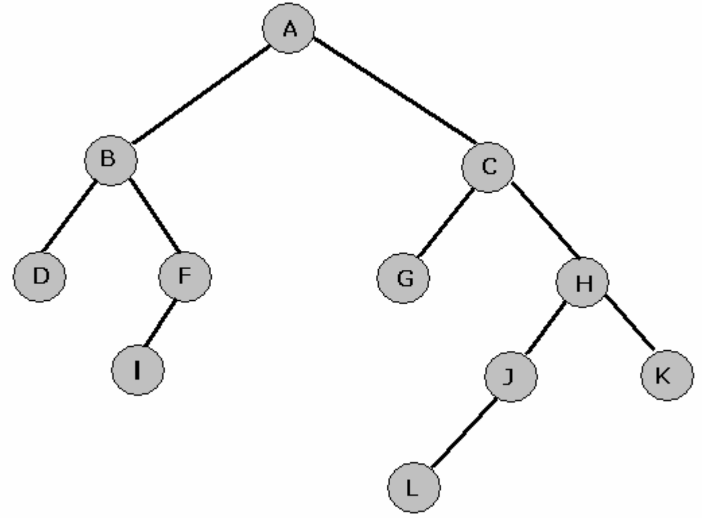
A binary tree T is said to be a 2-tree or an extended binary tree if each node N has either 0 or 2 children.

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

Shahid Iqbal (Lecturer) Computer College
Qassim University Kingdom of Saudi Arabia



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.

Preorder Traversal:

Algorithm: PREORDER (pRoot)

First time this function is called by passing original ***root*** into ***pRoot***. Here ***pRoot*** is pointers pointing to current root. This algorithm does a preorder traversal of T, applying by rcurrsively calling same function and updating ***proot*** to traverse in a way i.e. (**NLR**) Node, Left, Right.

1. If (pRoot NOT = NULL) then: [does child exist ?]

i- Apply PROCESS to pRoot-> info. e.g. Write: pRoot -> info

[recursive call by passing address of left child to update ***pRoot***]

ii- PREORDER (pRoot -> Left)

[recursive call by passing address of right child to update ***pRoot***]

iii- PREORDER(pRoot -> Right)

[End of If structure.]

2. Exit.

Inorder Traversal:

Algorithm: INORDER (pRoot)

First time this function is called by passing original ***root*** into ***pRoot***. Here ***pRoot*** is pointers pointing to current root. This algorithm does a Inorder traversal of T, applying by rcurrsively calling same function and updating ***proot*** to traverse in a way i.e. (**LNR**) Left, Node, Right.

1. If (pRoot NOT = NULL) then: [does child exist ?]

[recursive call by passing address of left child to update ***pRoot***]

i- PREORDER (pRoot -> Left)

ii- Apply PROCESS to pRoot-> info. e.g. Write: pRoot -> info

[recursive call by passing address of right child to update ***pRoot***]

iii- PREORDER(pRoot -> Right)

[End of If structure.]

2. Exit.

Postorder Traversal:

Algorithm: POSTORDER (pRoot)

First time this function is called by passing original ***root*** into ***pRoot***. Here ***pRoot*** is pointers pointing to current root. This algorithm does a Postorder traversal of T, applying by rcurrsively calling same function and updating ***proot*** to traverse in a way i.e. (**LRN**) Left, Right, Node.

1. If (pRoot NOT = NULL) then: [does child exist ?]

[recursive call by passing address of left child to update ***pRoot***]

i- PREORDER (pRoot -> Left)

[recursive call by passing address of right child to update ***pRoot***]

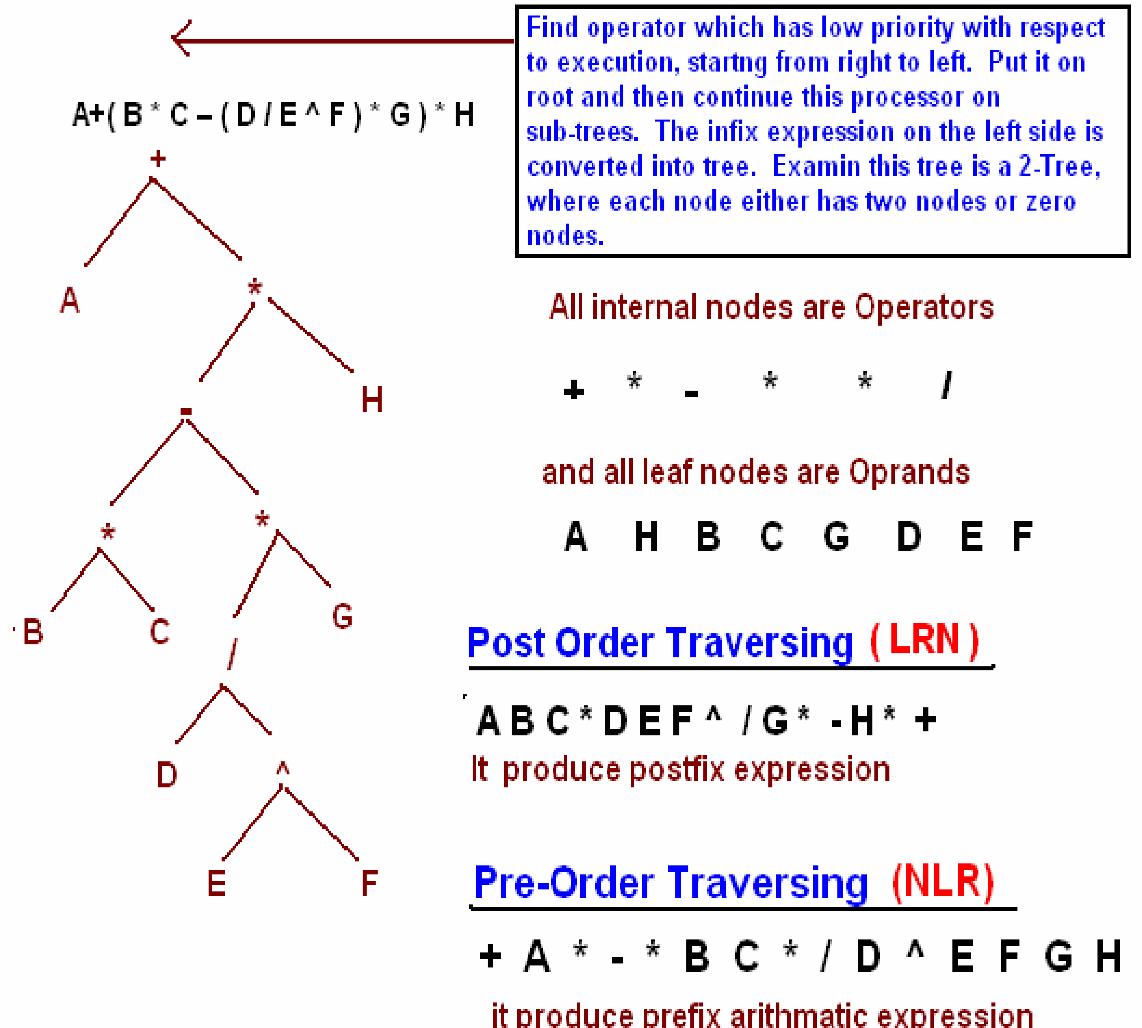
ii- PREORDER(pRoot -> Right)

iii- Apply PROCESS to pRoot-> info. e.g. Write: pRoot -> info

[End of If structure.]

2. Exit.

Preparing a Tree from an infix 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 (using arguments), called ***base criteria,*** for which the procedure / function does not call itself.
(2) Each time the procedure / function does call itself, control must be closer to the base criteria.

Created with nitro PDF professional
download the free trial online at nitropdf.com/professional

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( ); // prototyping
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

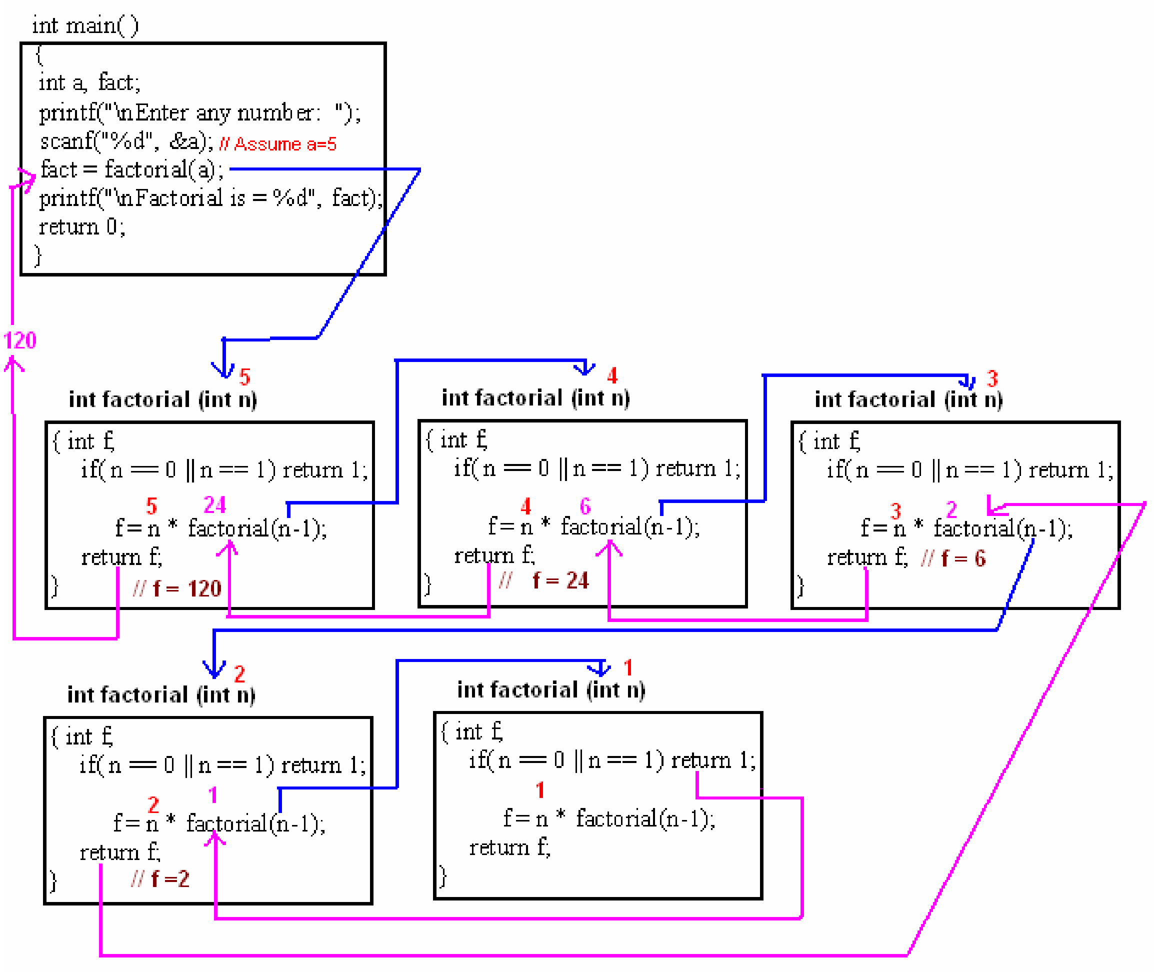
4! -> 4*3*2*1 = 24   **5!** -> 5*4*3*2*1 = 120   **6!** -> 6*5*4*3*2*1 = 720

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;
}
```



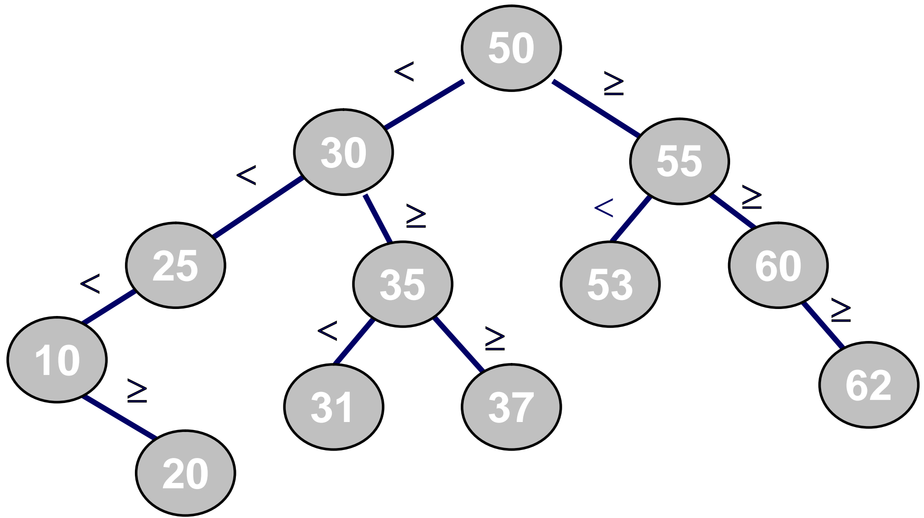
Created with nitro PDF professional
download the free trial online at nitropdf.com/professional

Binary Search Tree:

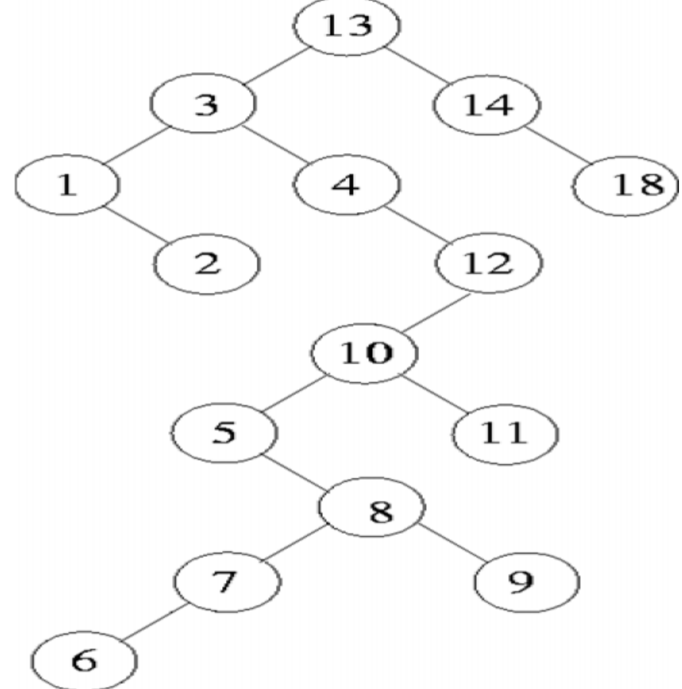
Suppose T is a binary tree, the T is called a binary search tree or binary sorted tree if each node N of T has the following property:

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.



Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

- **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. There are three possible cases to consider:

  Deleting a leaf (node with no children): Deleting a leaf is easy, as we can simply remove it from the tree.

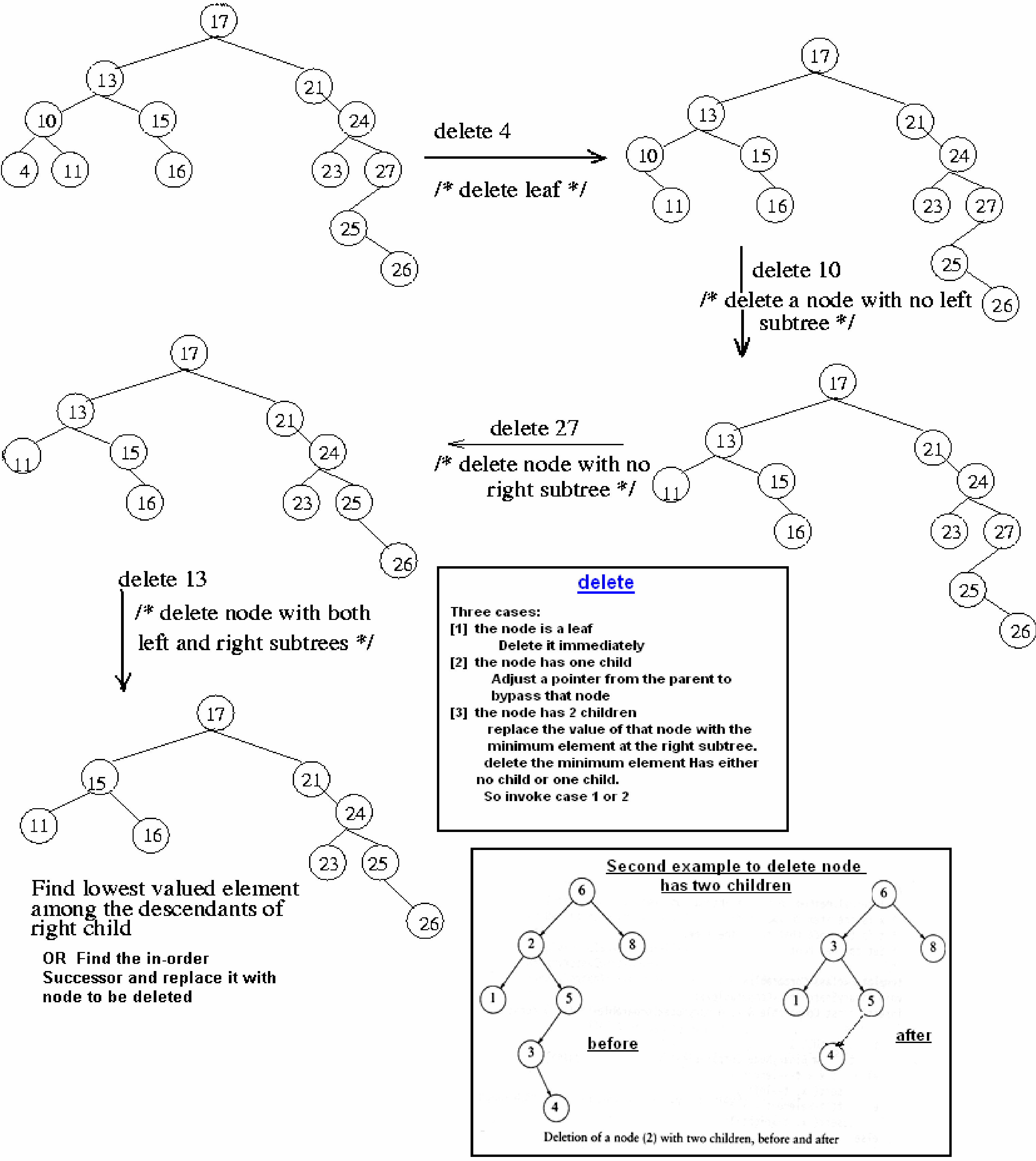
  Deleting a node with one child: Delete it and replace it with its child.

  Deleting a node with two children: Call the node to be deleted "N". Do not delete N. Instead, choose its in-order successor node "S". Replace the value of “N” with the value of “S”. (Note: S itself has up to one child.)

  As with all binary trees, a node's in-order successor is the left-most child of its right subtree. This node will have zero or one child. Delete it according to one of the two simpler cases above.

  Figure on next page illustrates several scenarios for deletion in BSTs.

Created with nitroPDF professional
download the free trial online at nitropdf.com/professional



Shahid Iqbal (Lecturer) Computer College
Qassim University Kingdom of Saudi Arabia
Created with nitro PDF professional
download the free trial online at nitropdf.com/professional

```
/*  This program is about to implement different operations on Binary
    Search Tree.            Programmed by SHAHID LONE   */
#include <stdio.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= (struct NODE *) malloc(sizeof(NODE));
        NewNode->Left = NULL;
        NewNode->Right= NULL;
        NewNode->info = x;
        AttachNode( Root, NewNode );

}
```


Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

```
void Pre_Order(struct NODE *pRoot)
{
        if (pRoot)
        {
                printf("%d\t",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);
                printf("%d\t",pRoot->info);
        }
}


void In_Order(struct NODE *pRoot)
{
        if( pRoot  )
        {
                if(pRoot->Left)  In_Order(pRoot->Left);

                printf("%d\t",pRoot->info);

                if(pRoot->Right)  In_Order(pRoot->Right);

        }
}


void DisplayDescending(struct NODE *pRoot)
{
        if( pRoot )
        {
                if(pRoot->Right)  DisplayDescending(pRoot->Right);

                printf("%d\t",pRoot->info);

                if(pRoot->Left) DisplayDescending(pRoot->Left);

        }
}
```


Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

```
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;
        while( 1 )
        {       printf("\n\n\n   Binary Search Tree Functions\n\n");
                printf("\n1. Insert a New Node");
                printf("\n2. Remove Existing Node");
                printf("\n3. In-Order Traverse  (Ascending Order)");
                printf("\n4. Pre-Order Traverse ");
                printf("\n5. Post-Order Traverse ");
                printf("\n6. Display in Descending Order   (Reverse)");
                printf("\n7. Exit");
                printf("\nEnter you choice: ");
                scanf("%d",&ch);

                switch(ch)
                {
                case 1:
                        printf("\n\n put a number: "); scanf("%d",&item);
                        Insert(item);
                        break;

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

                case 3:
                        printf("\n\n\n In-Order Traverse (ASCENDING ORDER)\n");
                        In_Order(Root);
                        printf("\n\n");
                        break;

                case 4:
```


Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

```
                printf("\n\n\n Pre-Order Traverse \n");
                Pre_Order(Root);
                printf("\n\n");
                break;
        case 5:
                printf("\n\n\n Post-Order Traverse \n");
                Post_Order(Root);
                printf("\n\n");
                break;

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

        case 7:
                 DeleteTree(Root);
                exit(0);

        default:
                printf("\n\nInvalid Input");

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


Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

Sorting:

Sorting and searching are fundamental operations in computer science. Sorting refers to the operation of arranging data in some given order. Such as increasing or decreasing, with numeric data or alphabetically, with string data.

Insertion Sort:

Suppose an array ***A*** with ***N*** elements ***A***[0], ***A***[1], ... ***A***[N-1] is in memory. The insertion sort algorithm scan A from ***A***[0] to ***A***[N-1], inserting each element ***A***[K] into its proper position in the previously sorting sub array A[0], A[1], ...A[K-1]. That is:

Pass 1: A[1] is inserted either before of after A[0] so that: A[0], A[1] is sorted.
Pass 2: A[2] is inserted into its proper place in A[0], A[1], so that A[0], A[1], A[2] are sorted.
Pass 3: A[3] is inserted into its proper place in A[0], A[1], A[2] so that: A[0], A[1], A[2], A[3] are sorted.
. . . .
Pass N-1: A[N-1] is inserted into its proper place in A[0], A[1], ... A[N-1] so that: A[0], A[1], ... A[N-1] are sorted.

This sorting algorithm is frequently used when ***N*** is small. There remains only the problem of deciding how to insert A[K] in its proper place in the subarray A[0], A[1], ... A[K-1]. This can be accomplished by comparing A[K] with A[K-1], comparing A[K] with A[K-2], comparing A[K] with A[K-3], and so on, until first meeting an element A[i] (where ***i*** start from k-1) such that A[i] ≤ A[K]. then each of elements A[K-1], A[K-2], ... A[i+1] is moved forward one location, and A[K] is then inserted in the ***i***+1 st position in the array.

Pass	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
	80	77	33	44	11	88	22	66	55
K=1	77	80	33	44	11	88	22	66	55
K=2	33	77	80	44	11	88	22	66	55
K=3	33	44	77	80	11	88	22	66	55
K=4	11	33	44	77	80	88	22	66	55
K=5	11	33	44	77	80	88	22	66	55
K=6	11	22	33	44	77	80	88	66	55
K=7	11	22	33	44	66	77	80	88	55
K=8	11	22	33	44	55	66	77	80	88
Sorted	11	22	33	44	55	66	77	80	88

Created with nitro PDF professional
download the free trial online at nitropdf.com/professional

The algorithm is simplified if there always is an element A[i] such that A[i] ≤ A[K]; other wise we must constantly check to see if we are comparing A[K] with A[0].

Algorithm: (INSERTION SORT) INSERTION (A, N)
[Where A is an array and N is the Size of values in the array]

1. Repeat steps 2 to 4 for K=1,2,3, ... N-1:
2. Set TEMP = A[K] and i =K-1.
3. Repeat while i >= 0 and TEMP < A[i]
   a) Set A[i+1] = A[i]. [Moves element forward.]
   b) Set i = i -1.
   [End of loop.]
4. Set A[i+1] =TEMP. [Insert element in proper place.]
   [End of Step 2 loop.]
5 Return.

```
//     Programe of Implementation of   Insertion SORT
//     Programmd by SHAHID   LONE
// Insertion Sort

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

const int SIZE = 100;
void InsertionSort( int x[],int );

void main(void)
{      int i,n;
       int   A[SIZE];
       cout<"\nData Collection for Insertion Sort\n";
       cout<< "\nHow many value are to process: ";
       cin>> n;
       cout<<"\nINPUT "<<n<<" values:\n";
       for( i=0; i< n ; i++ ) cin>>A[i];
       InsertionSort( A,n);
       cout<< "\nSORTED ARRAY\n";
       for( i=0; i < n; i++ ) A[i]<<'\t';

}


void InsertionSort( int x[100], int n )
{      int i, k, temp;
       for( k=1; k< n; k++ )
       {  temp = x[k];

             for( i=k-1; i>=0 && temp<x[i]; i-- ) x[i+1] = x[i];
             x[i+1] = temp;
       }
}
```

Selection Sort:

Suppose an array A with N elements A[0], A[1], . . . A[N-1] Is in memory. The Selection sort algorithm for sorting ***A*** works as follows. First find the smallest element in the list and put it in the first position. Then find the second smallest element in the list and put it the second position. And so on.

Pass 1: Find the location LOC of the smallest in the list of N elements A[0], A[1], ... A[N-1], and then interchange A[LOC] and A[0]. Then:
A[0] is sorted.

Pass 2: Find the location LOC of the smallest in the sublist of N-1 elements A[1], A[2], . . . A[N-1], and interchange A[LOC] and A[1]. Then:
A[0], A[1] is sorted. Since A[0] ≤ A[1].

. . . .
. . . .

Pass N-1: Find the location LOC of the smallest A[N-2] and A[N-1], and then interchanged A[LOC] and A[N-1]. Then:
A[0], A[1], A[2], ... A[N-1] is sorted.

Algorithm: MIN(A, K, N, LOC)
[An Array A is in memory. This procedure finds the location Loc of the smallest element among A[K], A[K+1], ...A[N-1].]
1. Set MIN = A[K] and LOC = K. [Initializes pointers.]
2. Repeat for J=K+1, K+2, ... N:
   If MIN > A[J], then: MIN = A[J] and LOC = J.
3. Return LOC.

Algorithm: (SELECTION SORT) SELECTION (A, N)
1. Repeat steps 2 and 3 for K=0,1,2, ... N-2:
2. Call MIN(A, K, N, LOC).
3. [Interchange A[K] and A[LOC].]
   Set TEMP = A[K], A[K] = A[LOC] and A[LOC] = TEMP.
   [End of step 1 loop.]
4. Exit.

Pass	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
K=0, LOC=3	**77**	33	44	**11**	88	22	66	55
K=1, LOC=5	11	**33**	44	77	88	**22**	66	55
K=2, LOC=5	11	22	**44**	77	88	**33**	66	55
K=3, LOC=5	11	22	33	**77**	88	**44**	66	55
K=4, LOC=7	11	22	33	44	**88**	77	66	**55**
K=5, LOC=6	11	22	33	44	55	**77**	**66**	88
K=6, LOC=6	11	22	33	44	55	66	**77**	88
Sorted	11	22	33	44	55	66	77	88

//

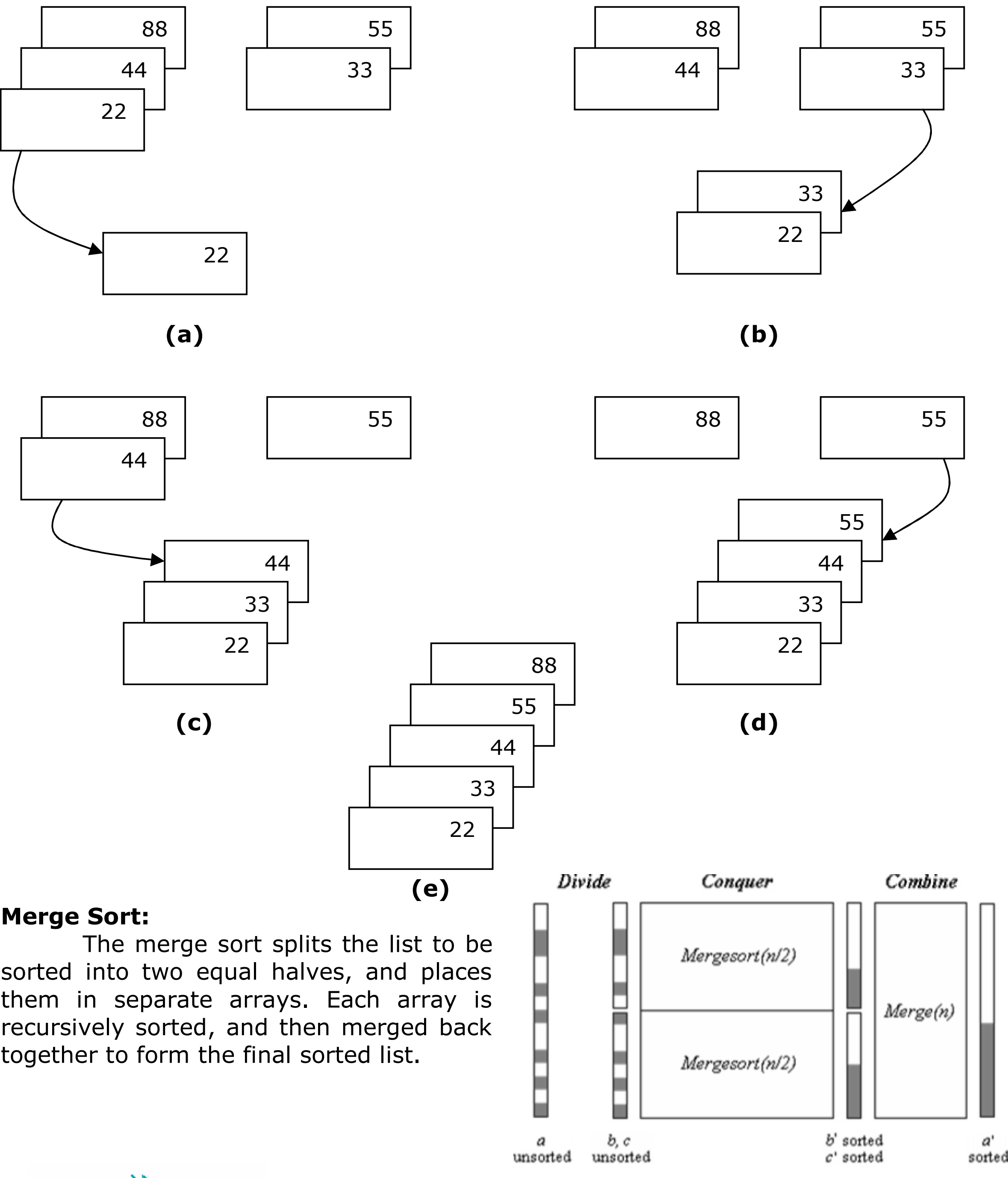
Programe of Implementation of **Selection SORT**

```
//    Programmed  by  SHAHID  LONE
#include<iostream.h>
#include<conio.h>
#define ARRAY_SIZE 100
void SelectionSort( int [], int);
int  MIN(int [], int);

void main(void)
{      int i,n ;
       int   A[ARRAY_SIZE];
       cout<< "\n Selection Sort\n";
       cout<< "How many values are to process: "; cin>>n;
       cout<< "\PUT "<<n<< " Random values\n";
       for( i=0; i< n; i++ ) cin>> nvdata[i] );
       SelectionSort(A, n);
       cout<<"\nSORTED ARRAY\n";
       for( i=0; i< n; i++ ) cout<< A[i]<<"\t";
 }
void SelectionSort( int A[ARRAY_SIZE], int n )
{   int K, loc, temp;
       for( K = 0 ; K < N-1 ; K++ )
       {  loc = MIN (A, K);
             temp = A[K]; A[K] = A[loc];  A[loc] = temp;}
}
int  MIN(int A[ARRAY_ZIZE], int K)
 {  int min, j, loc;
     min = A[K]; loc = K;
     for( j = K+1; j <= K-1 ;   j++)
          if( min > A[j]){min = A[j]; loc = j;}
     return loc;
  }
```

Merging:

Suppose A is sorted list with r elements and B is a sorted list with s elements. The operation that combine the elements of A and B into a single sorted list C with n= r+s elements is called merging. One simple way to merge is to place the elements of C after the elements of A and then use some sorting algorithm on the entire lsit. This method does not take advantage of the fact that A and B are individually sorted. A much more efficient algorithm is merge sort algorithm.
Suppose one is given two sorted decks of cards. The decks are merged as:



(a) (b) (c) (d) (e)

Merge Sort:

The merge sort splits the list to be sorted into two equal halves, and places them in separate arrays. Each array is recursively sorted, and then merged back together to form the final sorted list.

Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

Consider an array then split into two list, then:

Pass 1: | 5 | 4 | 2 | 3 |

Pass 2: | 5 | 4 | | 2 | 3 |

Pass 3: | 4 | 5 | | 2 | 3 |

Pass 4: | 2 | 3 | 4 | 5 |

```
Algorithm: MERGESORT (A, N)
   1. If N=1, Return.
   2. Set N1:=N/2, N2:=N-N1.
   3. Repeat for i=0,1,2 . . . . . (N1-1)
                        Set L [i]=A [i].
   4. Repeat for j=0,1,2 . . . . . (N2-1)
                        Set R [j]=A [N1+j].
   5. Call MERGESORT (L, N1).
   6. Call MERGESORT (R, N2).
   7. Call MERGE (A, L, N1, R,N2).
   8. Return.
```

```
Algorithm: MERGE (A, L, N1, R, N2)
   1. Set i:=0, j:=0.
   2. Repeat for k=0,1,2 . . . . . (N1+N2-1)
                        If i<N1, then:
                              If j=N2 or L [i] ≤ R [j], then:
                                    Set A [k] =L [i].
                                    Set i=i+1;
                        Else:
                              If j < N2, then:
                                    Set A [k]=R [j].
                                    Set j=j+1.
   3. Return.
```


Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

C++ Code (MERGE SORT):

```
#include <iostream.h>
#include <conio.h>
int a[50];
class merge_sort
       {public:
              void merge(int,int,int);
              void sort(int ,int);
       };
void merge_sort::sort(int low,int high)
        {
       int mid;
       if(low<high)
              {mid=(low+high)/2;
               sort(low,mid);
               sort(mid+1,high);
               merge(low,mid,high);
               }
        }
void merge_sort::merge(int low,int mid,int high)
        {int h,i,j,b[50],k;
         h=low;
         i=low;
         j=mid+1;
         while((h<=mid)&&(j<=high))
              {if(a[h]<=a[j])
                     {b[i]=a[h];
                      h++;
                     }
               else
                      {b[i]=a[j];
                       j++;
                      }
                i++;
               }
          if(h>mid)
              {for(k=j;k<=high;k++)
                     {b[i]=a[k];
                      i++;
                      }
              }
          else
               {for(k=h;k<=mid;k++)
                     {b[i]=a[k];
                      i++;
               }
        }
     for(k=low;k<=high;k++) a[k]=b[k];
    }

void main()
   {int num,i;
    merge_sort obj;

cout<<"*****************************************************************"<<endl;
cout<<"                      MERGE SORT PROGRAM"<<endl;

cout<<"*****************************************************************"<<endl;
    cout<<endl<<endl;
    cout<<"How many ELEMENTS you want to sort : ";
    cin>>num;
    cout<<endl;
```


Created with nitroPDF professional
download the free trial online at nitropdf.com/professional

```
     cout<<"Enter the ( "<< num <<" ) numbers :"<<endl;
    for(i=1;i<=num;i++)
        {cin>>a[i] ; }
    obj.sort(1,num);
    cout<<endl;
    cout<<"So, the sorted list (using MERGE SORT) will be :"<<endl;
    cout<<endl<<endl;
    for(i=1;i<=num;i++)
        cout<<a[i]<<" ";
    cout<<endl<<endl<<endl<<endl;
}
```

Radix Sort:

Radix sort is the method that many people intuitively use when alphabetizing a large list of names. (Here the radix is 26, the 26 letters of alphabets.

The basic procedure of radix sort is as follow:

- Based on examining digits in some base-b numeric representation of items (or keys)
- Least significant digit radix sort
    - Processes digits from right to left
- Create groupings of items with same value in specified digit
    - Collect in order and create grouping with next significant digit

55	24	92	40

Input	0	1	2	3	4	5	6	7	8	9
55						55				
24					24					
92			92							
40	40									

Input	0	1	2	3	4	5	6	7	8	9
40					40					
92										92
24			24							
55						55				

Input	0	1	2	3	4	5	6	7	8	9
24	24									
40	40									
55	55									
92	92									

24	40	55	92

Algorithm: RADIXSORT (A, N, R)

1. If N=1, then: Return.
2. Set D=R/10, P=-1.
3. Repeat for i=0,1,2 . . . . (N-1)
   Repeat for j=0,1,2 . . . 9
   Set C [i] [j] := -1.
4. Repeat for i=0,1,2 . . . . (N-1)
   i. Set X=A [i] % R.
   ii. Set m=X/D.
   iii. If m>0, then: Set Z:=1.
   iv. Set C [i] [m] = A [i].
5. Repeat for j=0,1,2 9
   i. Repeat for j=0,1,2 . . . . (N-1)
      1. If C [i] [j] ≠ -1,then:
         a. Set P=P+1.
         b. Set A [P]=C [i] [j].
6. If Z=1, then
   RADIXSORT (A, N, R*10).
7. Return.

C++ Code (RADIX SORT):

```
#include <iostream.h>
#include <conio.h>
int const MAX = 100;
int A[MAX];
int C[MAX][10];
int N;
class Radix_sort
       {public:
              void get();
              void radix(int);
              void display();
       };
void Radix_sort::get()
       {cout<<"\n\tHow many values you want to sort -->";
        cin>>N;
        if(N>MAX)
              {cout<<"\n\n Maximum number of values is "<<MAX<<".\n please try
again....";
               getche();
               get();
              }
        int i=0;
        cout<<"\n Put "<<N<<" values..\n";
        for(i;i<N;i++)
              {cin>>A[i];}
       }
void Radix_sort::radix(int R)
       {int i,j,X,D;
        int Z=0;
        D=R/10;
        for(i=0;i<N;i++)
              {for(j=0;j<10;j++)
                     {C[i][j]=-1;}
              }
        for(i=0;i<N;i++)
```

```
            {X=A[i]%R;
             int m=X/D;
             if(m>0)Z=1;
             C[i][m]=A[i];
            }
        int p=-1;
        for(j=0;j<10;j++)
            {for(i=0;i<N;i++)
                 {if(C[i][j]!=-1)
                      {p++;
                      A[p]=C[i][j];
                      }
                 }
            }

        if(Z==1)
            {R*=10;
             radix(R);
            }
        }
void Radix_sort::display()
       {clrscr();
        cout<<"\n\n\t\t\t Sorted List\n\n";
        for(int i=0;i<N;i++)
            {cout<<A[i]<<"\t";}
       }
void main()
       {
        Radix_sort obj;
        obj.get();
        if(N>1)obj.radix(10);
        obj.display();

       }
```

Quick Sort(An Application of STACKS):

Let A be a list of n data items "sorting A" refers to the operation of rearranging the elements of A so that they are in some logical order. Such as numerically ordered when A contains numerical data, or alphabetically ordered when A contains character data.

Quick sort is an algorithm of the divide-and-conquer type. That is, the problem of sorting a set is reduced to the problem of sorting two smaller sets.

Algorithm: QUICKSORT (A, LEFT, RIGHT)

1. If LEFT ≥ RIGHT, then: Return.
2. Set MIDDLE = PARTITION (A, LEFT, RIGHT).
3. Call QUICKSORT (A, LEFT, MIDDLE-1).
4. Call QUICKSORT (A, MIDDLE+1, RIGHT).
5. Return.

Algorithm: PARTITION (A, LEFT, RIGHT)

1. Set X = A[LEFT].
2. Set I = LEFT.
3. Repeat for j = LEFT+1,LEFT+2, ... RIGHT
   - If A[j] < X, then:
     - Set i = i+1.
     - Call SWAP (A[i], A[j]).
4. Call SWAP (A[i], A[LEFT].
5. Return i.

Created with nitro PDF professional
download the free trial online at nitropdf.com/professional

C++ Code (QUICK SORT):

```
#include <iostream.h>
#include <conio.h>
int const MAX=100;
int A[MAX];
int N;
class quick
       {public:
              void get();
              void quick_sort(int ,int);
              int partition(int , int);
              void display();
       };
void quick::get()
       {cout<<"\n\n How many values you want to sort --> ";
        cin>>N;
        if(N>MAX)
              {cout<<"\n\n  \t  Maximum  number  of  values  is  "<<MAX<<"  so  try
again...";
               getche();
               get();
              }
        cout<<"\n\t\t Put "<<N<<" random values..\n\n";
        for(int i=0;i<N;i++)
              {cin>>A[i];}
       };
void quick::quick_sort(int LEFT, int RIGHT)
       {if(LEFT>=RIGHT)return;
        int MIDDLE=partition(LEFT,RIGHT);
        quick_sort(LEFT,MIDDLE-1);
        quick_sort(MIDDLE+1,RIGHT);
       }
int quick::partition(int LEFT, int RIGHT)
       {int X=A[LEFT];
        int i=LEFT;
        for(int j=LEFT+1;j<=RIGHT;j++)
              {if(A[j]<X)
                     {i++;
                      int temp=A[i];
                      A[i]=A[j];
                      A[j]=temp;
                     }
              }
        int temp=A[i];
        A[i]=A[LEFT];
        A[LEFT]=temp;
        return i;
       }
void quick::display()
       {clrscr();
        cout<<"\n\n\n\n\t\t\t\t Sorted List\n\n";
        cout<<"\t\t\t\t(QUICK SORT)\n\n\n";
        for(int i=0;i<N;i++)
              {cout<<A[i]<<"\t";}
       }
void main()
       {
        quick obj;
        obj.get();
        obj.quick_sort(0,N-1);
        obj.display();

       }
```