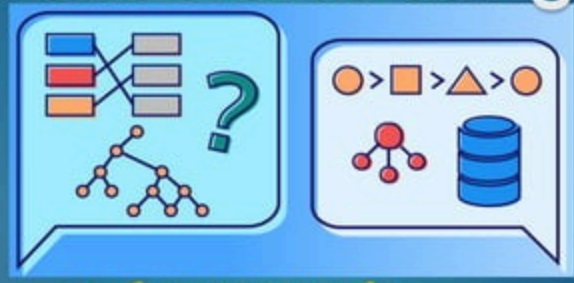
Data Structures using C++



Prof. K ADISESHA (Ph. D)

Learning Objective

DATA STRUCTURES

- Introduction
- Types of Data structures
- Linear Data structure
- Stacks and Queues
- Linked lists
- Non-Linear Data structure

Introduction

Data Structures:

Data

- ***Data is a collection of facts, numbers, letters or symbols that the computer process into meaningful information.***

Data structure

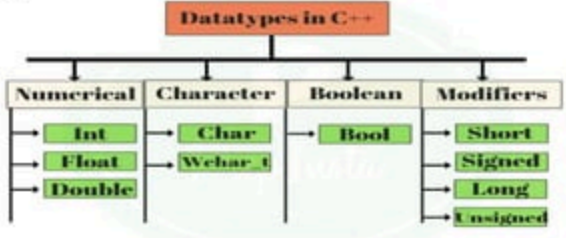
- ***Data structure is representation of the logical relationship existing between individual elements of data.***
- ***Data structure is a specialized format for organizing and storing data in memory that considers not only the elements stored but also their relationship to each other.***

Introduction

Data Structures:

Data structure affects the design of both structural & functional aspects of a program.

- A algorithm is a step by step procedure to solve a particular function.
  - Program=algorithm + Data Structure

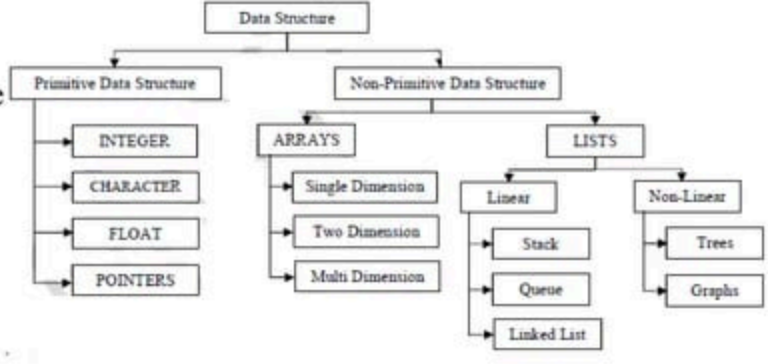


Data Structures

Classification of Data Structure:

***Data structure are normally divided into two broad categories*:**

- Primitive Data Structure
- Non-Primitive Data Structure

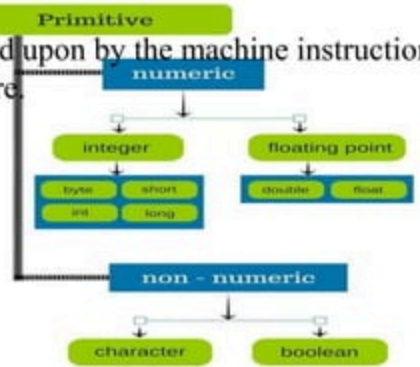


Data Structures

Primitive Data Structure:

Data structures that are directly operated upon the machine-level instructions are known as primitive data structures:

- There are basic structures and directly operated upon by the machine instructions.
- The Data structures that fall in this category are,
  - *Integer*
  - *Floating-point number*
  - *Character constants*
  - *string constants*
  - *pointers etc.,*



Data Structures

Primitive Data Structure:

Data structures that are directly operated upon the machine-level instructions are known as primitive data structures:

- The most commonly used operation on data structure are broadly categorized into following types:
  - ***Create***
  - ***Selection***
  - ***Updating***
  - ***Destroy or Delete***

Data Structures

Non-Primitive Data Structure:

The Data structures that are derived from the primitive data structures are called Non-primitive data structure:

- There are more sophisticated data structures
- The non-primitive data structures emphasize on structuring of a group of homogeneous (same type) or heterogeneous (different type) data items:
  - ***Linear Data structures***
  - ***Non-Linear Data structures***

Data Structures

Non-Primitive Data Structure:

Linear Data structures

Linear Data structures are kind of data structure that has homogeneous elements.

- The data structure in which elements are in a sequence and form a liner series.
- Linear data structures are very easy to implement, since the memory of the computer is also organized in a linear fashion.
- Some commonly used linear data structures are:
  - ***Stack***
  - ***Queue***
  - ***Linked Lists***

Data Structures

Non-Primitive Data Structure:

Non-Linear Data structures

A Non-Linear Data structures is a data structure in which data item is connected to several other data items.

- Non-Linear data structure may exhibit either a hierarchical relationship or parent child relationship.
- The data elements are not arranged in a sequential structure.
- Some commonly used non-linear data structures are:
  - ***Trees***
  - ***Graphs***

Data Structures

Non-Primitive Data Structure:

The most commonly used operation on data structure are broadly categorized into following types:

- ***Traversal***
- ***Insertion***
- ***Selection***
- ***Searching***
- ***Sorting***
- ***Merging***
- ***Destroy or Delete***

Data Structures

Differences between Data Structure:

The most commonly used differences between on data structure are broadly categorized into following types:

- A ***primitive data structure*** is generally a basic structure that is usually built into the language, such as an integer, a float.

- A ***non-primitive data structure*** is built out of primitive data structures linked together in meaningful ways, such as a or a linked-list, binary search tree, AVL Tree, graph etc.

Arrays

Arrays:

An array is defined as a set of finite number of homogeneous elements or same data items:

- Declaration of array is as follows:
- Syntax: ***Datatype Array_Name [Size];***
- Example: ***int arr[10];***

  - ✓ Where int specifies the data type or type of elements arrays stores.
  - ✓ "arr" is the name of array & the number specified inside the square brackets is the number of elements an array can store, this is also called sized or length of array.

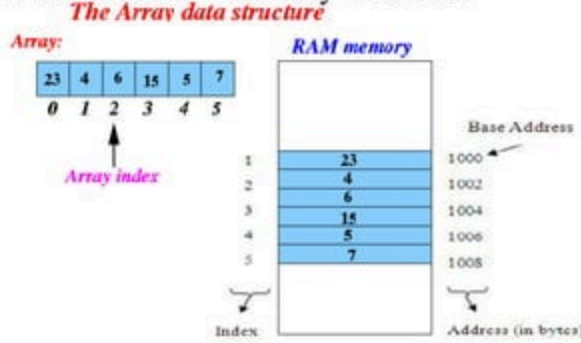
Arrays

Arrays:

Represent a Linear Array in memory:

- The elements of linear array are stored in consecutive memory locations.
- It is shown below:

  int A[5]={23, 4, 6, 15, 5, 7}



Arrays

Calculating the length of the array:

The elements of array will always be stored in the consecutive (continues) memory location.

- The number of elements that can be stored in an array, that is the size of array or its length is given by the following equation:
  - A[n] is the array size or length of n elements.
  - The length of the array can be calculated by:

    L = UB – LB + 1
  - To Calculate the address of any element in array:

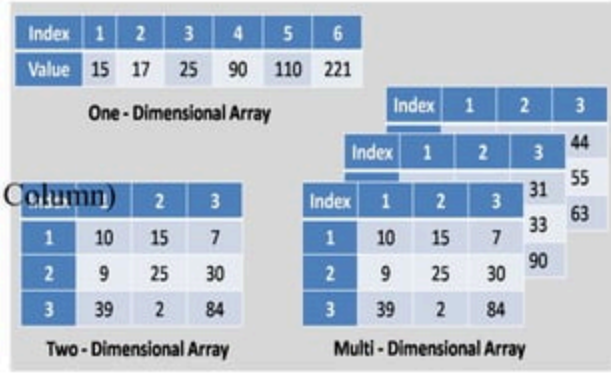
    Loc(A[P])=Base(A)+W(P-LB)
  - Here, UB is the largest Index and LB is the smallest index
- Example: If an array A has values 10, 20, 30, 40, 50, stored in location 0,1, 2, 3, 4 the UB = 4 and LB=0 Size of the array L = 4 – 0 + 1 = 5

Arrays

Types of Arrays:

The elements of array will always be stored in the consecutive (continues) memory location. The various types of Arrays are:

- **Single Dimension Array:**
  - Array with one subscript
  - Ex: int A[i];
- **Two Dimension Array**
  - Array with two subscripts (Rows and Column)
  - Ex: int A[i][j];
- **Multi Dimension Array:**
  - Array with Multiple subscripts
  - Ex: int A[i][j]..[n];

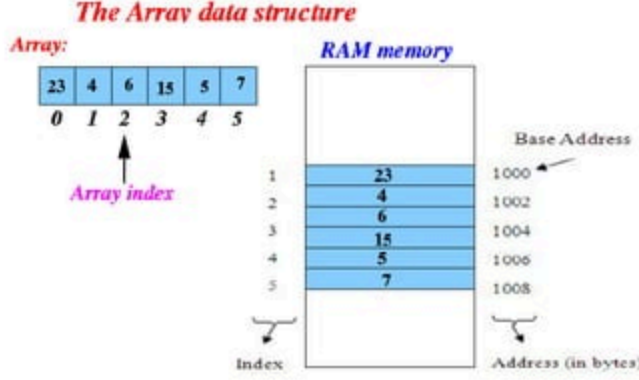


Arrays

Basic operations of Arrays:

Some common operation performed on array are:

- ***Traversing***
- ***Searching***
- ***Insertion***
- ***Deletion***
- ***Sorting***
- ***Merging***



Arrays

Traversing Arrays:

***Traversing: It is used to access each data item exactly once so that it can be processed*:**

- We have linear array A as below:

1	2	3	4	5
10	20	30	40	50

- Here we will start from beginning and will go till last element and during this process we will access value of each element exactly once as below:

A [0] = 10
A [1] = 20
A [2] = 30
A [3] = 40
A [4] = 50

ALGORITHM: Traversal (A, LB, UB) A is an array with Lower Bound LB and Upper Bound UB.

```
Step 1:    for LOC = LB to UB do
Step 2:        PROCESS A [LOC]
           [End of for loop]
Step 3:    Exit
```

Arrays

Program:To find the frequency of presence element in the given array:

```
#include<iostream.h>
#include<iomanip.h>
#include<conio.h>
class frequency
{
     private:
               int  n, m[100], ele, freq;
     public:
               void getdata();
               void findfreq();
               void display();
};
void frequency::getdata()
{
     cout<<"Enter the size of the array: ";
     cin>>n;
     cout<<"Enter "<<n<<" elements into the array:   ";
     for(int i=0; i<n; i++)
          cin>>m[i];
     cout<<"Enter the search element:   ";
     cin>>ele;
}
void frequency::findfreq()
{
     freq = 0;
     for(int i=0; i<n; i++)
          if(ele==m[i])
               freq++;
}
```

```
void frequency::display()
{
     if(freq > 0)
          cout<<"Frequency of "<<ele<<" is "<<freq;
     else
          cout<<ele<<" does not exist";
}
```

```
void main()
{
     frequency F;
     clrscr();
     F.getdata();
     F.findfreq();
     F.display();
     getch();
}
```

```
Enter the size of the array: 5
Enter 5 elements into the array:   10 50 40 30 40
Enter the search element:   40
Frequency of 40 is 2
```

Arrays

Insertion into Array:

***Insertion: It is used to add a new data item in the given collection of data items*:**

- We have linear array A as below:

1	2	3	4	5
10	20	50	30	15

- New element to be inserted is 100 and location for insertion is 3.
- So shift the elements from 5th location to 3rd location downwards by 1 place.
- And then insert 100 at 3rd location

Arrays

Insertion into Array:

Insertion into Array:

- Insertion 100 into Array at Pos=3

A [0] = 10
A [1] = 20
A [2] = 50
A [3] = 30
A [4] = 15

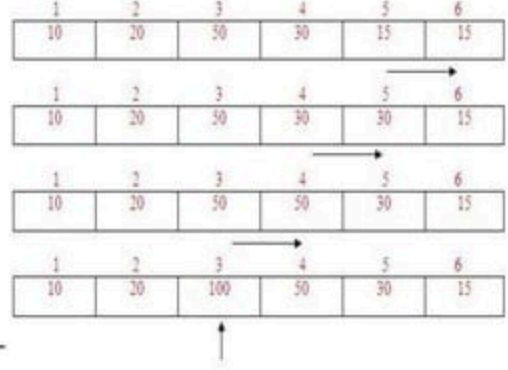
ALGORITHM: Insert (A, N, ITEM, Pos) A is an array with N elements. ITEM is the element to be inserted in the position Pos.

Steo 1: for I = N-1 down to Pos
A[I + 1] = A[I]
[End of for loop]

Step 2: A [Pos] = ITEM

Step 3: N = N+1

Step 4: Exit



Arrays

Insertion into Array: Add a new data item in the given array of data:

***Insertion into Array*:**

A [0] = 10
A [1] = 20
A [2] = 50
A [3] = 30
A [4] = 15

```
#include<iostream.h>
#include<iomanip.h>
class insertion
{
     private:  int  n, m[100], ele, p;
     public:
               void getdata();
               void insert();
               void display();
};
void insertion::getdata()
{
     cout<<"How many elements? ";
     cin>>n;
     cout<<"Enter the elements: ";
     for(int i=0; i<n; i++)
          cin>>m[i];
     cout<<"Enter the element to be inserted: ";
     cin>>ele;
     cout<<"Enter the position ( 0 to "<<n<<"): ";
     cin>>p;
}
```

```
void insertion::insert()
{
     if(p > n)
     {
          cout<<p<<" is an invalid position";
          exit(0);
     }
     for(int i=n-1; i>=p; i--)
          m[i+1] = m[i];
     m[p] = ele;
     n++;
     cout<<ele<<" is successfully inserted"<<endl;
}

void insertion::display()
{
     cout<<"The array after the insertion is ";
     for(int i=0; i<n; i++)
          cout<<setw(4)<<m[i];
}
void main()
{
     insertion I;
     I.getdata();
     I.insert();
     I.display();
}
```

```
How many elements? 5
Enter the elements: 20 30 40 50 60
Enter the element to be inserted: 10
Enter the position ( 0 to 5): 0
10 is successfully inserted into position 0
The array after the insertion is    10   20   30   40   50   60
```

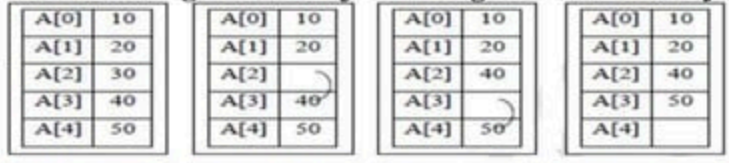
Arrays

Deletion from Array:

Deletion: It is used to delete an existing data item from the given collection of data items:

- Deletion 30 from Array at Pos 3

A [0] = 10
A [1] = 20
A [2] = 30
A [3] = 40
A [4] = 50



ALGORITHM: Delete (A, N, ITEM, Pos) A is an array with N elements. ITEM is the element to be deleted in the position Pos and it is stored into variable Item.

```
Step 1:   ITEM = A [Pos]
Step 2:   for I = Pos down to N-1
              A[ I ] = A[I+1]
          [End of for loop]
Step 3:   N = N-1
Step 4:   Exit
```

Arrays

Deletion from Array:

A [0] = 10
A [1] = 20
A [2] = 30
A [3] = 40
A [4] = 50

```
#include<iostream.h>
#include<iomanip.h>
#include<conio.h>
class deletion
{
     private:
          int m[100], n, ele, p;
     public:
          void getdata();
          void remove();
          void display();
};
void deletion::getdata()
{
     cout<<"How many elements? ";
     cin>>n;
     cout<<"Enter the elements: ";
     for(int i=0; i<n; i++)
          cin>>m[i];
     cout<<"Enter the position (0 to "<<n-1<<"): ";
     cin>>p;
}
```

```
How many elements? 5
Enter the elements: 10 20 30 40 50
Enter the position (0 to 4): 0
The element 10 at position 0 is successfully removed
The array after the deletion is   20  30  40  50
```

```
void deletion::remove()
{
     if(p > n-1)
     {
          cout<<p<<" is an invalid position";
          exit(0);
     }
     ele = m[p];
     for(int i=p+1; i<n; i++)
          m[i-1] = m[i];
     n--;
     cout<<ele<<" is successfully removed"<<endl;
}
     void deletion::display()
     {
          cout<<"The array after deletion is ";
          for(int i=0; i<n; i++)
               cout<<setw(4)<<m[i];
     }
     void main()
     {
          deletion D;
          clrscr();
          D.getdata();
          D.remove();
          D.display();
          getch();
     }
```

Arrays

Searching in Arrays:

Searching: It is used to find out the location of the data item if it exists in the given collection of data items:

- E.g. We have linear array A as below:

1	2	3	4	5
10	20	50	30	35

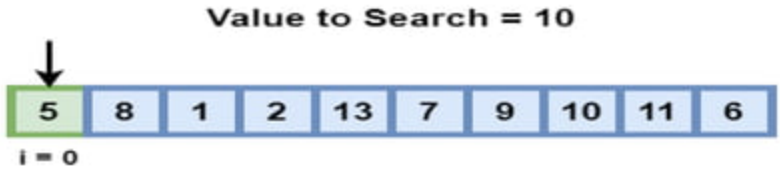
- Suppose item to be searched is 35. We will start from beginning and will compare 35 with each element.
- This process will continue until element is found or array is finished.
- Types of searching Algorithms:
  - ***Linear searching***
  - ***Binary Searching***

Arrays

Linear search:

Linear Searching: Also called Sequential Searching.

- It is used to find out the location of the data item if it exists in the given collection of data items.
- Example Searching element 33 from the array of elements:



Arrays

Linear search:

Linear Searching: Also called Sequential Searching.

ALGORITHM: Linear_Search (A, N, Element) A is an array with N elements. Element is the being searched in the array.

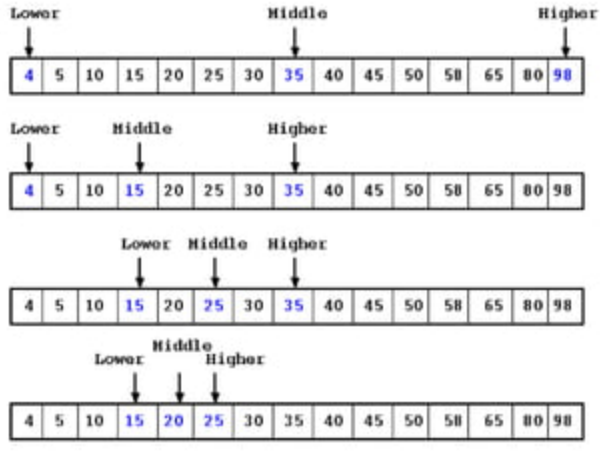
```
Step 1:     LOC = -1 [Assume the element does not exist]
Step 2:     for I = 0 to N-1 do
                if ( Element = A [ I ] ) then
                    LOC = I
            Goto Step 3
                [End if]
            [End of for loop]
Step 3: if ( LOC >= 0 ) then
            Print Element, "Found in Location", LOC
        else
            Print Element, "Not Found"
Step 4:     Exit
```

Arrays

Binary Searching:

The binary search algorithm can be used with only sorted list of elements.

- Binary Search first divides a large array into two smaller sub-arrays and then recursively operate the sub-arrays.
- Binary Search basically reduces the search space to half at each step

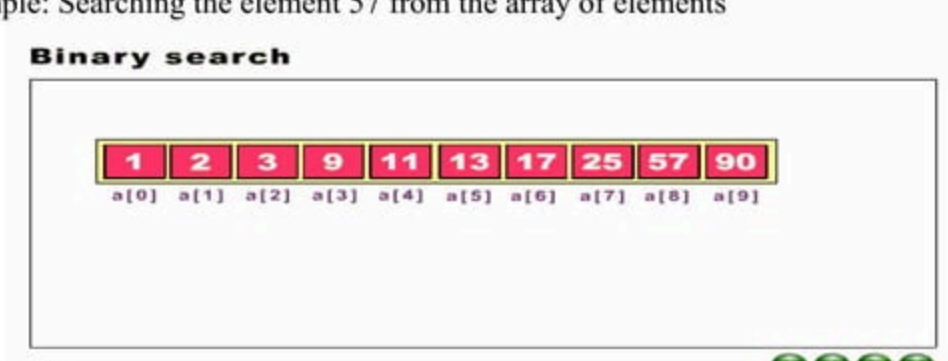


Arrays

Binary Searching:

The binary search algorithm can be used with only sorted list of elements.

- Example: Searching the element 57 from the array of elements



Arrays

Binary Searching:

➢ Example:

	BEG & END	MID = (BEG+END)/2	Compare	Location
A[0] A[1] A[2] A[3] A[4] 12 23 39 47 57 BEG MID END	BEG = 0 END = 4	MID = (0+4)/2 MID = 2	67>39 (Does not match)	LOC = -1
The search element i.e. 67 is greater than the element in the middle position i.e. 39 then continues the search to the right portion of the middle element.				
A[0] A[1] A[2] A[3] A[4] 12 23 39 47 57 BEGMID END	BEG = 3 END = 4	MID = (3+4)/2 MID = 3	67>47 (Does not match)	LOC = -1
The search element i.e. 67 is greater than the element in the middle position i.e. 47 then continues the search to the right portion of the middle element.				
A[0] A[1] A[2] A[3] A[4] 12 23 39 47 57 BEGMID END	BEG = 4 END = 4	MID = (4+4)/2 MID = 4	67>57 (Does not match)	LOC = -1
The search element i.e. 67 is greater than the element in the middle position i.e. 57 then continues the search to the right portion of the middle element.				
A[0] A[1] A[2] A[3] A[4] 12 23 39 47 57 ENDBEG	BEG = 5 END = 4	Since the condition (BEG <= END) is false the comparison ends		
We get the output as 67 **Not Found**				

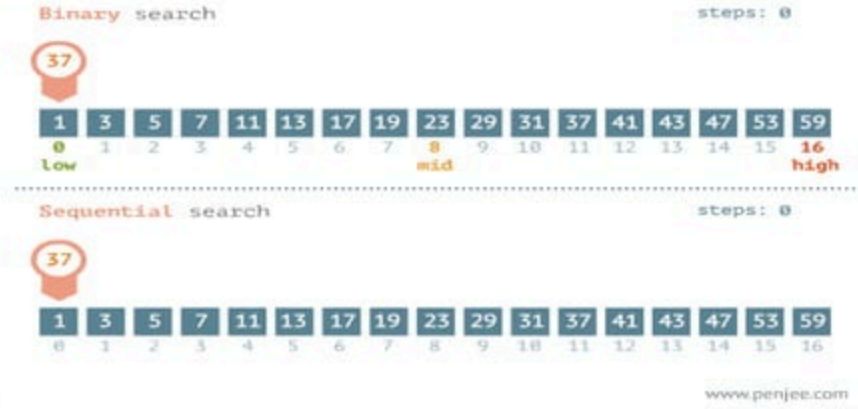
Arrays

Binary Searching: **ALGORITHM**: Binary_Search (BEG, END, MID ELE) A is an array with N elements. Let BEG, END, MID denote Beginning, end and middle location of the array

```
Step 1:     Set BEG=LB, END=UB LOC = -1
Step 2:     While(BEG <= END)
                MID = (BEG+END)/2
            if ( ELE = A [ MID ] ) then
                LOC = MID
            Goto Step 3
            else
                if( ELE < A[MID])
                    END=MID-1;
                else
                    BEG=MID+1;
                [End if]
            [End if]
            [End of While loop]
Step 3:     if ( LOC >= 0 ) then
                Print Element, "Found in Location", LOC
            else
            Print Element, "Search is Unsuccessful"
Step 4:     Exit
```

Arrays

Difference in Searching:



Arrays

Difference in Searching:

	Linear Search	Binary Search
1	This can be used in sorted and unsorted array	This can be used only in sorted array
2	Array elements are accessed sequentially	One must have direct access to the middle element in the sub list.
3	Access is very slow	Access is faster.
4	This can be used in single and multi dimensional array	Used only in single dimensional array.
5	This technique is easy and simple in implementing	Complex in operation

Arrays

Sorting in Arrays:

A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements:

- The comparison operator is used to decide the new order of element in the respective data structure.
- Types of Sorting Algorithms are:

❖ ***Bubble Sort***	❖ *Heap Sort*
❖ ***Insertion Sort***	❖ *Radix Sort*
❖ ***Selection Sort***	❖ *Bucket Sort*
❖ *Merge Sort*	❖ *Shell Sort*
❖ *Quick Sort*	

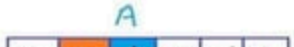
Arrays

Bubble Sort in Arrays:

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

8 5 3 1 4 7 9

Bubble Sort



2, 7, 4, 1, 5, 3
⇒ 2, 4, 1, 5, 3, 7
⇒ 2, 1, 4, 3, 5, 7

```
for i ← 0 to n-2
{
   if (A[i] > A[i+1])
   {
      swap(A[i], A[i+1])
   }
}
```

Arrays

Insertion Sorting:

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time.

- This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted.

6 5 3 1 8 7 2 4

Arrays

Insertion Sorting:

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted.

- This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted.

A

2	7	7	1	5	3
0	1	2	3	4	5

i	value	hole
1	2	1
1	2	0
2	4	2
2	4	1

```
InsertionSort(A, n)
{
  for i ← 1 to n-1
  {
    value ← A[i]
    hole ← i
    while(hole>0 && A[hole-1]>value
    {
      A[hole] ← A[hole-1]
      hole ← hole-1
    }
    A[hole] ← value
  }
}
```

Arrays

Insertion Sorting: **ALGORITHM: Insertion Sort (A, N) A is an array with N unsorted elements.**

- Step 1: for I=1 to N-1
- Step 2: J = I

  While(J >= 1)

  if (A[J] < A[J-1]) then

  Temp = A[J];

  A[J] = A[J-1];

  A[J-1] = Temp;

  [End if]

  J = J-1

  [End of While loop]

  [End of For loop]
- Step 3: Exit

6 5 3 1 8 7 2 4

Arrays

Selection Sort:

- The algorithm works as follows:
- Find the minimum value in the list
- Swap it with the value in the first position.
- Repeat the steps above for the remainder of the list (starting at the second position and advancing each time).

8	5	2	6	9	3	1	4	0	7

```
    void SelectionSort(int A[], int n)
     {
        int i, j, small, temp;
(1)     for (i = 0; i < n-1; i++) {
(2)       small = i;
(3)       for (j = i+1; j < n; j++)
(4)         if (A[j] < A[small])
(5)         small = j;
(6)     temp = A[small];
(7)     A[small] = A[i];
(8)     A[i] = temp;
        }
       }
```

Arrays

Selection Sort:

Yellow is smallest number found
Blue is current item
Green is sorted list

Arrays

Merging from Array:

Merging: It is used to combine the data items of two sorted files into single file in the sorted form.

- We have sorted linear array A as below:

1	2	3	4	5	6
10	40	50	80	95	100

- And sorted linear array B as below:

1	2	3	4
20	35	45	90

- After merging merged array C is as below:

1	2	3	4	5	6	7	8	9	10
10	20	35	40	45	50	80	90	95	100

Arrays

Two dimensional array:

A two dimensional array is a collection of elements and each element is identified by a pair of subscripts. (A[3] [3]).

- The elements are stored in continuous memory locations.
- The elements of two-dimensional array as rows and columns.
- The number of rows and columns in a matrix is called as the order of the matrix and denoted as MxN.
- The number of elements can be obtained by multiplying number of rows and number of columns.

	A[0]	A[1]	A[2]
A[0]	10	20	30
A[1]	40	50	60
A[2]	70	80	90

Arrays

Representation of Two Dimensional Array:

A two dimensional array is a collection of elements and each element is identified by a pair of subscripts. (A[m] [n])

- A is the array of order m x n. To store m*n number of elements, we need m*n memory locations.
- The elements should be in contiguous memory locations.
- There are two methods:
  - ***Row-major method***
  - ***Column-major method***

	A[0]	A[1]	A[2]
A[0]	10	20	30
A[1]	40	50	60
A[2]	70	80	90

Arrays

Representation of Two Dimensional Array:

Row-Major Method:

- All the first-row elements are stored in sequential memory locations and then all the second-row elements are stored and so on. Ex: A[Row][Col]

	A[0]	A[1]	A[2]
A[0]	10	20	30
A[1]	40	50	60
A[2]	70	80	90

1000	10	A[0][0]
1002	20	A[0][1]
1004	30	A[0][2]
1006	40	A[1][0]
1008	50	A[1][1]
1010	60	A[1][2]
1012	70	A[2][0]
1014	80	A[2][1]
1016	90	A[2][2]

Row-Major Method

Column-Major Method:

- All the first column elements are stored in sequential memory locations and then all the second-column elements are stored and so on. Ex: A [Col][Row]

1000	10	A[0][0]
1002	40	A[1][0]
1004	70	A[2][0]
1006	20	A[0][1]
1008	50	A[1][1]
1010	80	A[2][1]
1012	30	A[0][2]
1014	60	A[1][2]
1016	90	A[2][2]

Col-Major Method

Arrays

Calculating the length of the 2D-Array

A two dimensional array is a collection of elements and each element is identified by a pair of subscripts. (A[m] [n])

- The size of array or its length is given by the following equation:
  A[i][j] is the array size or length of m*n elements.
- To Calculate the address of i*j th element in array:
  - **Row-Major Method:** ***Loc(A[i][j])=Base(A)+W[n(i-LB)+(j-LB)]***
  - **Col-Major Method:** ***Loc(A[i][j])=Base(A)+W[(i-LB)+m(j-LB)]***

  Here, ***W*** is the number of words per memory location and ***LB*** is the smallest index

Arrays

Advantages of Array

A two dimensional array is a collection of elements and each element is identified by a pair of subscripts. (A[m] [n])

- It is used to represent multiple data items of same type by using single name.
- It can be used to implement other data structures like linked lists, stacks, queues, tree, graphs etc.
- Two-dimensional arrays are used to represent matrices.
- Many databases include one-dimensional arrays whose elements are records.

Arrays

Disadvantages of Array

A two dimensional array is a collection of elements and each element is identified by a pair of subscripts. (A[m] [n])

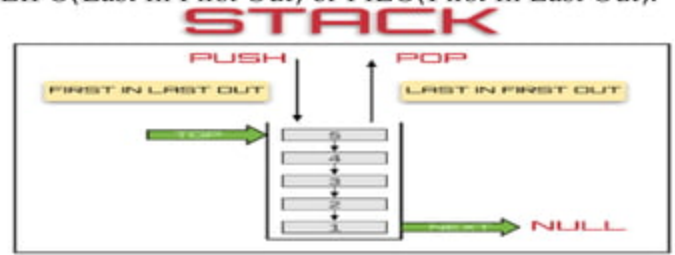
- We must know in advance the how many elements are to be stored in array.
- Array is static structure. It means that array is of fixed size. The memory which is allocated to array cannot be increased or decreased.
- Array is fixed size; if we allocate more memory than requirement then the memory space will be wasted.
- The elements of array are stored in consecutive memory locations. So insertion and deletion are very difficult and time consuming.

Stack

Stack:

Stack is a linear data structure which follows a particular order in which the operations are performed

- Insertion of element into stack is called PUSH and deletion of element from stack is called POP.
- The order may be LIFO(Last In First Out) or FILO(First In Last Out).

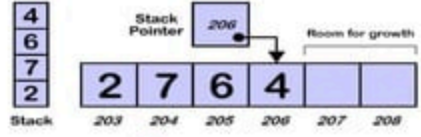


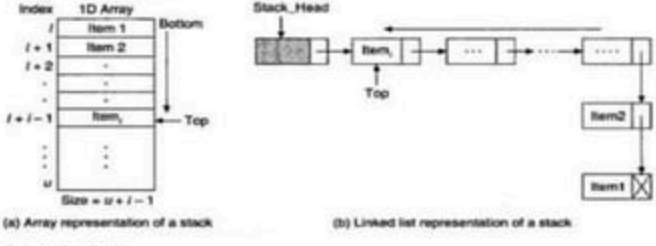
Stack

Stack:

Memory Representation of Stack

- The stack can be implemented into two ways:
  - Using arrays (Static implementation)
  - Using pointer (Dynamic





Stack

Operation on Stacks:

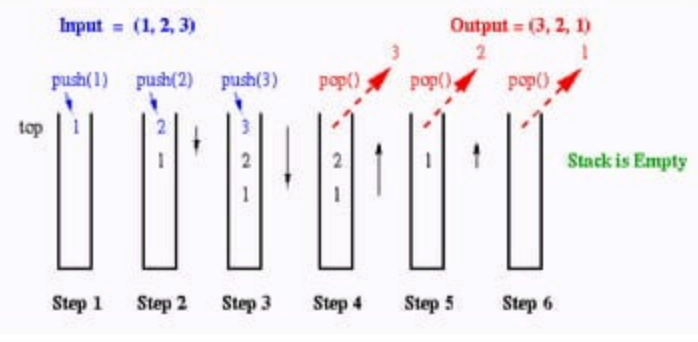
The various operation performed on Stacks are:

- **Stack( ):** It creates a new stack that is empty. It needs no parameter and returns an empty stack.
- **push(item):** It adds a new item to the top of the stack.
- **pop( ):** It removes the top item from the stack.
- **peek( ):** It returns the top item from the stack but does not remove it.
- **isEmpty( ):** It tests whether the stack is empty.
- **size( ):** It returns the number of items on the stack.

Stack

Stack Conditions:

The various operation performed on Stacks depend on the following conditions:



Stack

PUSH Operation:

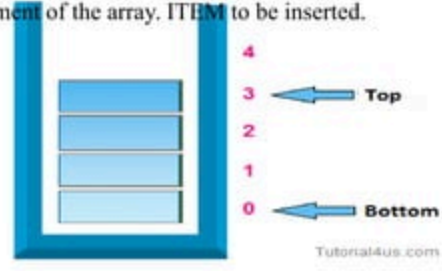
The process of adding one element or item to the stack is represented by an operation called as the PUSH operation:

ALGORITHM:

PUSH (STACK, TOP, SIZE, ITEM)

STACK is the array with N elements. TOP is the pointer to the top of the element of the array. ITEM to be inserted.

```
Step 1: if TOP = N then          [Check Overflow]
            PRINT " STACK is Full or Overflow"
            Exit
        [End if]
Step 2: TOP = TOP + 1            [Increment the TOP]
Step 3: STACK[TOP] = ITEM        [Insert the ITEM]
Step 4: Return
```

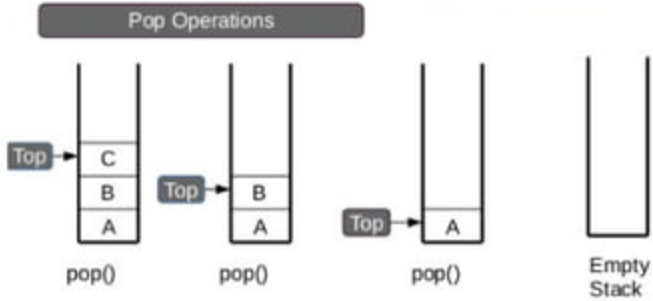


Stack

POP Operation:

The process of deleting one element or item from the stack is represented by an operation called as the POP operation.

When elements are removed continuously from a stack, it shrinks at same end i.e., top:



Stack

POP Operation:

The process of deleting one element or item from the stack is represented by an operation called as the POP operation.

ALGORITHM: POP (STACK, TOP, ITEM)

STACK is the array with N elements. TOP is the pointer to the top of the element of the array. ITEM to be DELETED.

Step 1: if TOP = 0 then [Check Underflow]
PRINT " STACK is Empty or Underflow"
Exit
[End if]
Step 2: ITEM = STACK[TOP] [copy the TOP Element]
Step 3: TOP = TOP - 1 [Decrement the TOP]
Step 4: Return

Stack

PEEK Operation:

The process of returning the top item from the stack but does not remove it called as the PEEK operation.

ALGORITHM: PEEK (STACK, TOP)

STACK is the array with N elements. TOP is the pointer to the top of the element of the array.

Step 1: if TOP = NULL then [Check Underflow]
PRINT " STACK is Empty or Underflow"
Exit
[End if]
Step 2: Return (STACK[TOP] [Return the top element of the stack]
Step 3:Exit

Stack

Application of Stacks:

- ***It is used to reverse a word.*** You push a given word to stack – letter by letter and then pop letter from the stack.
- ***"Undo" mechanism*** in text editor.
- ***Backtracking:*** This is a process when you need to access the most recent data element in a series of elements. Once you reach a dead end, you must backtrack.
- ***Language Processing:*** Compiler' syntax check for matching braces in implemented by using stack.
- ***Data Conversion*** of decimal number to binary.
- To solve ***tower of Hanoi.***
- Conversion of ***infix expression into prefix and postfix.***
- To sort elements using ***Quick sort***
- ***Runtime memory management.***

Infix	Prefix	Postfix
a + b	+ b a	a b +
(a + b) * (c + d)	* + d c + b a	a b + c d + *
b * b - 4 * a * c	- * c * a 4 * b b	b b * 4 a * c * -

Stack

Arithmetic Expression:

An expression is a combination of operands and operators that after evaluation results in a single value.

- Operand consists of constants and variables.
- Operators consists of {, +, -, *, /,),] etc.

➢ Expression can be

❖ **Infix Expression**: If an operator is in between two operands, it is called infix expression.

- ✓ **Example: a + b,** where a and b are operands and + is an operator.

❖ **Postfix Expression:** If an operator follows the two operands, it is called postfix expression.

- ✓ **Example: ab +**

❖ **Prefix Expression:** an operator precedes the two operands, it is called prefix expression.

- ✓ **Example: +ab**

Stack

Arithmetic Expression:

Three notations for the given arithmetic expression are listed below:

Infix: ((A + ((B ^ C) – D)) * (E – (A/C)))

Prefix: * + A – ^ BCD – E/AC

Postfix: ABC ^ D – + EAC / – *

EXAMPLE: Let us illustrate the procedure *InfixToPostfix* with the following arithmetic expression:
Input: (A + B)^ C - (D * E) / F) (infix form)

Read symbol	*Stack*	*Output*
Initial	(	
1	((	
2	((	A
3	((+	A
4	((+	AB
5	(	AB+
6	(^	AB+
7	(^	AB + C
8	(–	AB + C ^
9	(– (	AB + C ^
10	(– (	AB + C ^ D
11	(– (*	AB + C ^ D
12	(– (*	AB + C ^ DE
13	(–	AB + C ^ DE *
14	(– /	AB + C ^ DE *
15	(– /	AB + C ^ DE * F
16		AB + C ^ DE * F / –

Output: A B + C ^ DE * F / – (postfix form)

Queue

Queue:

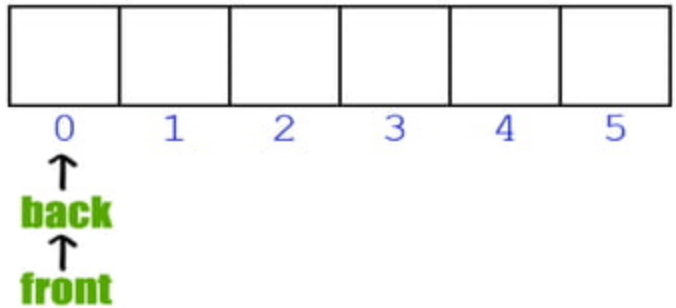
A queue is an ordered collection of items where an item is inserted at one end called the "rear" and an existing item is removed at the other end, called the "front".

- Queue is also called as FIFO list i.e. First-In First-Out.
- In the queue only two operations are allowed enqueue and dequeue.
- Enqueue means to insert an item into back of the queue.
- Dequeue means removing the front item.The people standing in a railway reservation row are an example of queue.

Queue

Queue:

A queue is an ordered collection of items where an item is inserted at one end called the "rear" and an existing item is removed at the other end, called the "front".

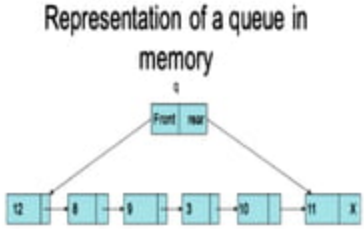


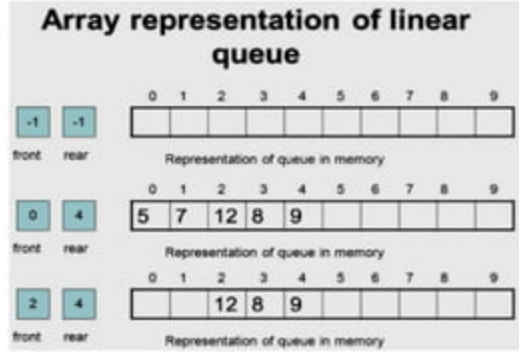
Queue

Memory Representation of Queue:

The queue can be implemented into two ways:

- Using arrays (Static implementation)
- Using pointer (Dynamic





Queue

Memory Representation of Queue:

Memory Representation of a queue using array:

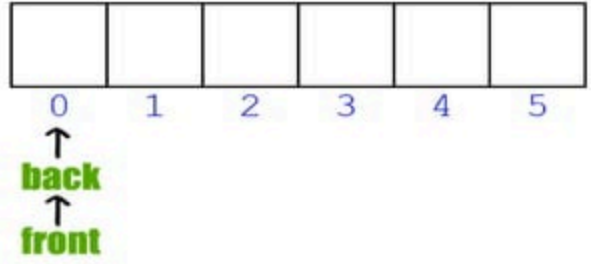
- Queue is represented in memory using linear array.
- Let QUEUE is a array, two pointer variables called FRONT and REAR are maintained.
- The pointer variable FRONT contains the location of the element to be removed or deleted.
- The pointer variable REAR contains location of the last element inserted.
- The condition FRONT = NULL indicates that queue is empty.
- The condition REAR = N-1 indicates that queue is full.

Queue

Types of Queues

Queue can be of four types:

- ***Simple Queue***
- ***Circular Queue***
- ***Priority Queue***
- ***De-queue (Double Ended Queue)***



Queue

Types of Queues

Simple Queue:

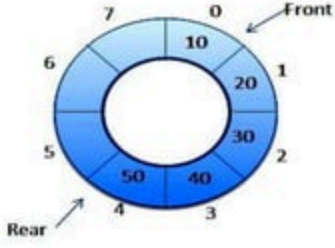
- **Simple Queue:** In simple queue insertion occurs at the rear end of the list and deletion occurs at the front end of the list.

Queue

Types of Queues

Circular Queue:

- **Circular Queue:** A circular queue is a queue in which all nodes are treated as circular such that the last node follows the first node.

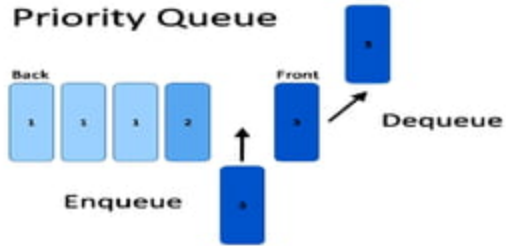


Queue

Types of Queues

Priority Queue:

- A **priority queue** is a queue that contains items that have some present priority. An element can be inserted or removed from any position depending upon some priority.

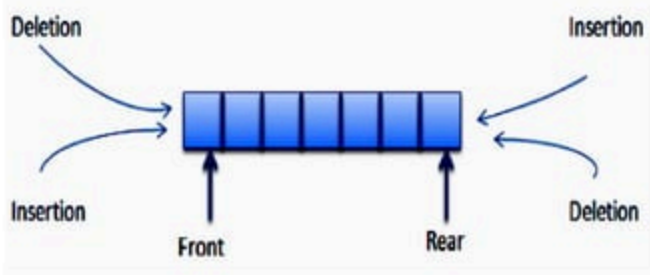


Queue

Types of Queues

Dequeue Queue:

- **Dequeue:** It is a queue in which insertion and deletion takes place at the both ends.

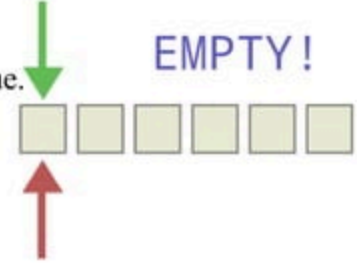


Queue

Operation on Queues

The various operation performed on Queue are :

- ***Queue( ):*** It creates a new queue that is empty.
- ***enqueue(item):*** It adds a new item to the rear of the queue.
- ***dequeue( ):*** It removes the front item from the queue.
- ***isEmpty( ):*** It tests to see whether the queue is empty.
- ***size( ):*** It returns the number of items in the queue.



Queue

Queue Insertion Operation (ENQUEUE):

ALGORITHM: ENQUEUE (QUEUE, REAR, FRONT, ITEM)

QUEUE is the array with N elements. FRONT is the pointer that contains the location of the element to be deleted and REAR contains the location of the inserted element. ITEM is the element to be inserted.

Step 1: if REAR = N-1 then [Check Overflow]
PRINT "QUEUE is Full or Overflow"
Exit
[End if]

Step 2: if FRONT = NULL then [Check Whether Queue is empty]
FRONT = -1
REAR = -1
else
REAR = REAR + 1 [Increment REAR Pointer]

Step 3: QUEUE[REAR] = ITEM [Copy ITEM to REAR position]

Step 4: Return

Front **Rear**

Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
8	12	4	18	34

Q[0]	Q[1]	Q[2]	Q[3]	Q[4]

Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
8	12			

Queue

Queue Deletion Operation (DEQUEUE):

ALGORITHM: DEQUEUE (QUEUE, REAR, FRONT, ITEM)

QUEUE is the array with N elements. FRONT is the pointer that contains the location of the element to be deleted and REAR contains the location of the inserted element. ITEM is the element to be inserted.

Step 1: if FRONT = NULL then [Check Whether Queue is empty]
PRINT "QUEUE is Empty or Underflow"
Exit
[End if]
Step 2: ITEM = QUEUE[FRONT]
Step 3: if FRONT = REAR then [if QUEUE has only one element]
FRONT = NULL
REAR = NULL
else
FRONT = FRONT + 1 [Increment FRONT pointer]
Step 4: Return

Front **Rear**

Q[0]	Q[1]	Q[2]	Q[3]	Q[4]

Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
8				

Q[0]	Q[1]	Q[2]	Q[3]	Q[4]
8	12			

Queue

Application of Queue:

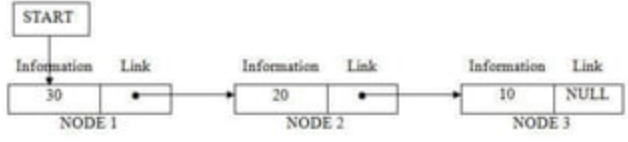
- ***Simulation***
- ***Various features of Operating system***
- ***Multi-programming platform systems.***
- ***Different types of scheduling algorithms***
- ***Round robin technique algorithms***
- ***Printer server routines***
- ***Various application software's is also based on queue data structure.***

Lists

Lists (linked list):

A lists (Linear linked list) can be defined as a collection of variable number of data items called nodes.

- Lists are the most commonly used non-primitive data structures.
- Each nodes is divided into two parts:
  - The first part contains the information of the element.
  - The second part contains the memory address of the next node in the list. Also called Link part.

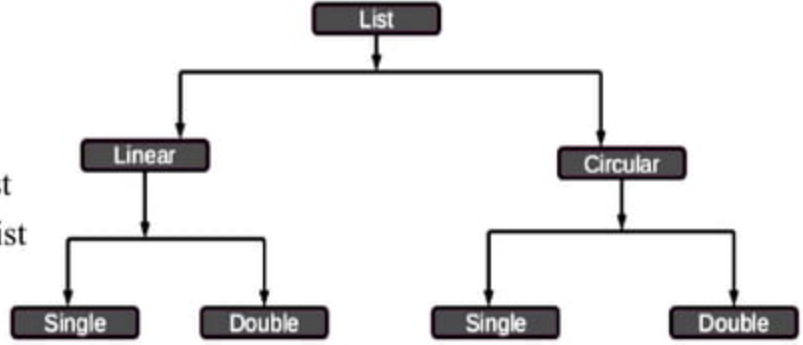


Lists

Lists (linked list):

Types of linked lists:

- Single linked list
- Doubly linked list
- Single circular linked list
- Doubly circular linked list



Lists

Single linked list:

The link field of the last node contains the memory address of the first node, such a linked list is called circular linked list:

- The information field contains the data of that node.
- The link field contains the memory address of the next node.
- The last link field contains the memory address as null (\).

➢ There is only one link field in each node, the linked list is called singly linked list.

A graphical representation of a list.

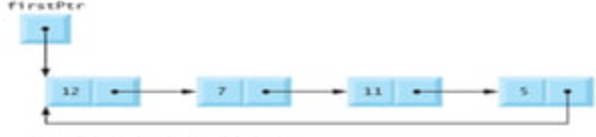
Lists

Single circular linked list:

A singly linked list contains two fields in each node - an information field and the linked field:

- The information field contains the data of that node.
- The link field contains the memory address of the next node
- The last link field contains the memory address of the first node.

- In a circular linked list every node is accessible from a given node.



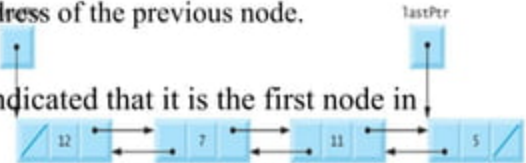
Circular, singly linked list.

Lists

Doubly linked list:

It is a linked list in which each node is points both to the next node and also to the previous node:

- In doubly linked list each node contains three parts:
  - FORW : It is a pointer field that contains the address of the next node
  - BACK: It is a pointer field that contains the address of the previous node.
  - INFO: It contains the actual data.
- In the first node, if BACK contains NULL, it indicated that it is the first node in the list.
- The in which FORW contains, NULL indicates that the node is the last node.



Doubly linked list.

Lists

Doubly circular linked list:

It is a linked list in which each node is points both to the next node and also to the previous node:

- In doubly linked list each node contains three parts:
  - FORW : It is a pointer field that contains the address of the next node
  - BACK: It is a pointer field that contains the address of the previous node.
  - INFO: It contains the actual data.

Fig. 19.12 | Circular, doubly linked list.

Lists

Operation on Linked List:

The operation that are performed on linked lists are:

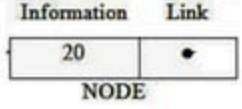
- *Creating a linked list*
- *Traversing a linked list*
- *Inserting an item into a linked list.*
- *Deleting an item from the linked list.*
- *Searching an item in the linked list*
- *Merging two or more linked lists.*

Lists

Creating a linked list:

The nodes of a linked list can be created by the following structure declaration:

```
struct Node
{
    int information;
    struct Node *link;
}*node1, *node2;
```



- Here info is the information field and link is the link field.
- The link field contains a pointer variable that refers the same node structure. Such a reference is called as Self addressing pointer.

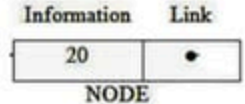
Lists

Operator new and delete:

The nodes of a linked list can be created by the following structure declaration:

- **Operators new** allocate memory space
- **Operators new [ ]** allocates memory space for array.
- **Operators delete** deallocate memory space.
- **Operators delete [ ]** deallocate memory space for array.

```
struct Node
{
    int information;
    struct Node *link;
}*node1, *node2;
```



Traversing a linked list

Traversing is the process of accessing each node of the linked list exactly once to perform some operation.

ALGORITHM: TRAVERS (START, P)

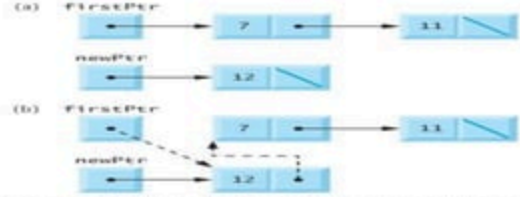
START contains the address of the first node. Another pointer p is temporarily used to visit all the nodes from the beginning to the end of the linked list.

Step 1: P = START
Step 2: while P != NULL
Step 3: PROCESS data (P) [Fetch the data]
Step 4: P = link(P) [Advance P to next node]
Step 5: End of while
Step 6: Return

Lists

Inserting a node into the linked list:

- Inserting a node at the beginning of the linked list
- Inserting a node at the given position.
- Inserting a node at the end of the linked list.



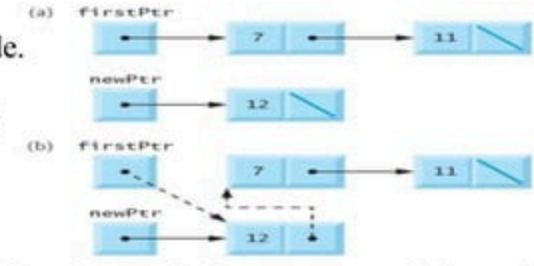
Operation insertAtFront represented graphically.

Lists

Inserting node at Front:

Inserting a node at the beginning of the linked list

1. Create a node.
2. Fill data into the data field of the new node.
3. Mark its pointer field as NULL
4. Attach this newly created node to START
5. Make the new node as the START node.



Operation insertAtFront represented graphically.

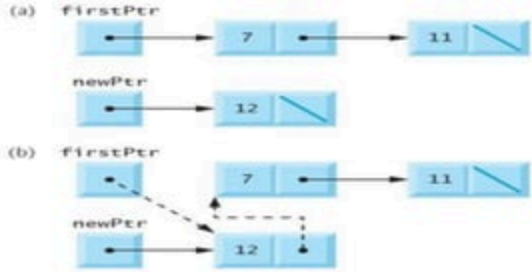
Lists

Inserting node at Front:

Inserting a node at the beginning of the linked list

ALGORITHM: INS_BEG (START, P) START contains the address of the first node.

Step 1: P←new Node;
Step 2: data(P)← num;
Step 3: link(P)← START
Step 4: START←P
Step 5: Return



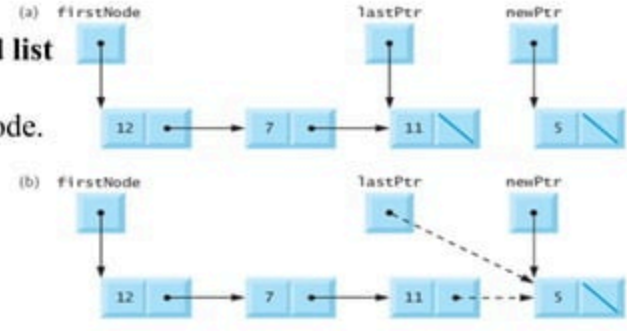
Operation insertAtFront represented graphically.

Lists

Inserting node at End:

Inserting a node at the End of the linked list

1. Create a node.
2. Fill data into the data field of the new node.
3. Mark its pointer field as NULL
4. Attach this newly created node to Last
5. Make the new node as the Last node.



Operation `insertAtBack` represented graphically.

Inserting node at End

ALGORITHM: INS_END (START, P)

START contains the address of the first node.

Step 1: START

Step 2: P ← START [identify the last node]

while P!= null

P ← next (P)

End while

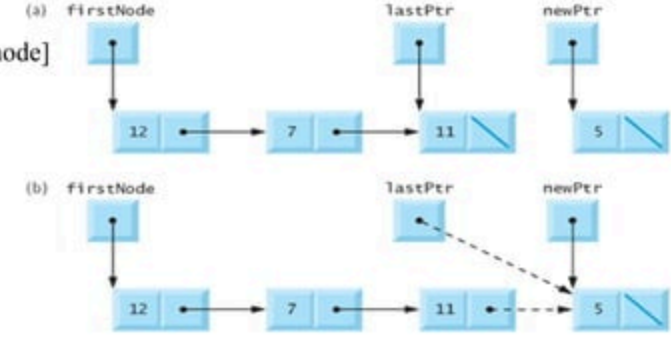
Step 3: N ← new Node;

Step 4: data(N) ← item;

Step 5: link(N) ← null

Step 6: link(P) ← N

Step 7: Return



Operation `insertAtBack` represented graphically.

Inserting node at a given Position

ALGORITHM: INS_POS (START, P1, P2)

START contains the address of the first node. P2 is new node

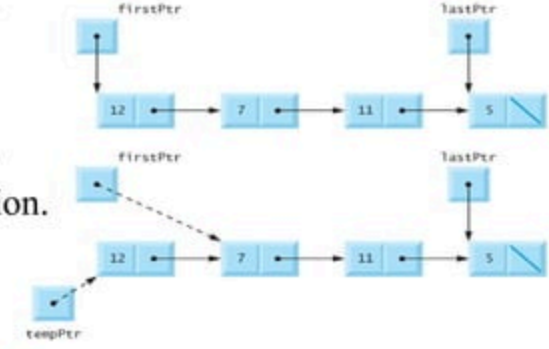
```
Step 1: START
Step 2: P1 ← START [Initialize node]
        Count ← 0
Step 3: while P!= null
            count ← count+1
            P1 ← next (P1)
        End while
Step 4: if (POS=1)
        Call function INS_BEG( )
        else if (POS=Count +1)
        Call function INS_END( )
        else if (POS<=Count)
          P1 ← Start
        For(i=1; i<=pos; i++)
          P1← next(P1);
        end for
        [create] P2← new node
        data(P2) ← item;
        link(P2) ← link(P1)
        link(P1)← P2
        else
        PRINT "Invalid position"
Step 5: Return
```

Deleting an node

Deleting an node:

Deleting an item from the linked list:

- Deletion of the first node
- Deletion of the last node
- Deletion of the node at the give position.



19.8 | Operation removeFromFront represented graphically.

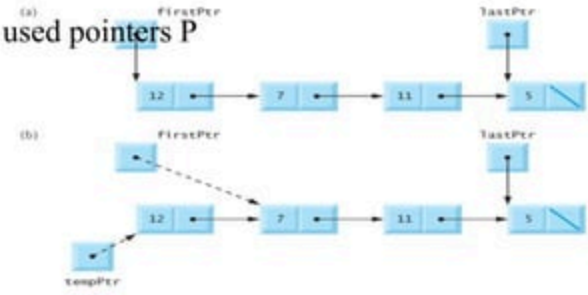
Deleting an node

Deletion of the first node:

This algorithm first copies data in the first node to a variable and delete the first node of the linked list.

ALGORITHM: DEL_BEG(P, START) This used pointers P

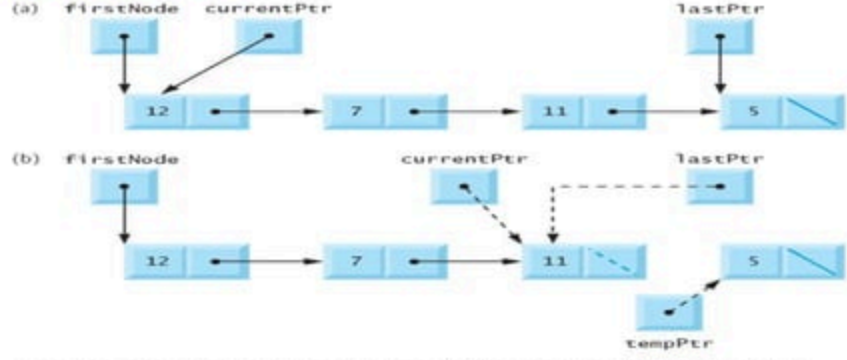
Step 1: START
Step 2: P ← START;
Step 3: PRINT data(P)
Step 4: START ← Link(P)
Step 5: Free(P)
Step 6: STOP



19.8 | Operation removeFromFront represented graphically.

Deleting an node

Deleting node from end:



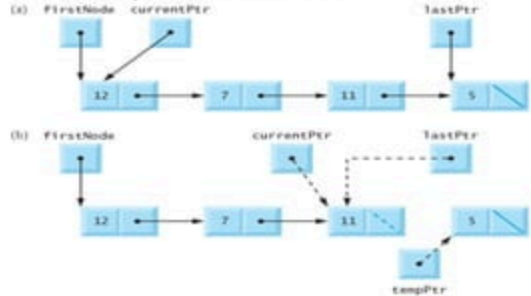
| Operation `removeFromBack` represented graphically.

Deleting an node

Deleting node from end:

ALGORITHM: DEL_END (P1, P2, START) This used two pointers P1 and P2. Pointer P2 is used to traverse the linked list and pointer P1 keeps the location of the previous node of P2.

Step 1: START
Step 2: P2 ← START;
Step 3: while (link(P2) ! = NULL)
    P1 ← P2
    P2 ← link(P2)
    While end
Step 4: PRINT data(p2)
Step 5: link(P1) ← NULL
    Free(P2)
Step 6: STOP



| Operation removeFromBack represented graphically.

Deleting node at a given Position

ALGORITHM: DEL_POS (START, P1, P2)

START contains the address of the first node. P2 is DEL-node

Step 1: START

Step 2: P1 ← START [Initialize node]

Count ← 0

Step 3: while P!= null

count ← count+1

P1 ← next (P1)

End while

Step 4: if (POS=1)

Call function DEL_BEG( )

else if (POS=Count)

Call function DEL_END( )

else if (POS<=Count)

P2 ← Start

For(i=1; i<=pos; i++)

P1 ← P2

P2 ← link(P2)

end for

PRINT data(P2)

link(P1) ← link(P2)

free(P2)

End if

Step 5: Return

Non-Linear Data structures

Non-Linear Data structures:

A Non-Linear Data structures is a data structure in which data item is connected to several other data items:

- The data items in non-linear data structure represent hierarchical relationship.
- Each data item is called node.
- The different non-linear data structures are
  - ***Trees***
  - ***Graphs.***

Non-Linear Data structures

Trees:

A tree is a data structure consisting of nodes organized as a hierarchy:

- Tree is a hierarchical data structure which stores the information naturally in the form of hierarchy style.
- It is a non-linear data structure compared to arrays, linked lists, stack and queue.
- It represents the nodes connected by edges.

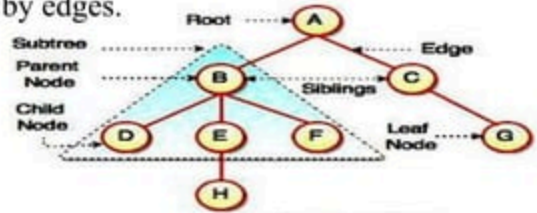


Fig. Structure of Tree

Non-Linear Data structures

Terminology of a Tree:

Field	Description
Root	Root is a special node in a tree. The entire tree is referenced through it. It does not have a parent.
Parent Node	Parent node is an immediate predecessor of a node.
Child Node	All immediate successors of a node are its children.
Siblings	Nodes with the same parent are called Siblings.
Path	Path is a number of successive edges from source node to destination node.
Height of Node	Height of a node represents the number of edges on the longest path between that node and a leaf.
Height of Tree	Height of tree represents the height of its root node.
Depth of Node	Depth of a node represents the number of edges from the tree's root node to the node.
Degree of Node	Degree of a node represents a number of children of a node.
Edge	Edge is a connection between one node to another. It is a line between two nodes or a node and a leaf.

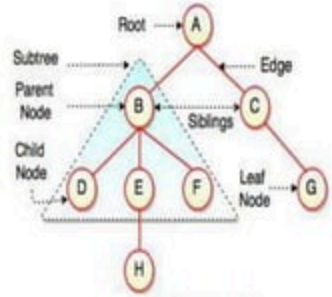


Fig. Structure of Tree

Non-Linear Data structures

Binary Tree:

A binary tree is an ordered tree in which each internal node can have maximum of two child nodes connected to it:

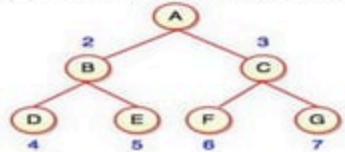


Fig. Binary Tree using Array

- A binary tree consists of:
  - A node (called the root node)
  - Left and right sub trees.
- A Complete binary tree is a binary tree in which each leaf is at the same distance from the root i.e. all the nodes have maximum two subtrees.

Binary tree using array represents a node which is numbered sequentially level by level from left to right. Even empty nodes are numbered.

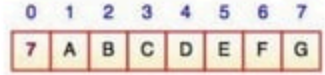


Fig. Location Number of an Array in a Tree

Non-Linear Data structures

Graph:

Graph is a mathematical non-linear data structure capable of representing many kind of physical structures:

- A graph is a set of vertices and edges which connect them.
- A graph is a collection of nodes called vertices and the connection between them called edges.
- Definition: A graph G(V,E) is a set of vertices V and a set of edges E.

Example of graph:

[a] Directed & Weighted Graph

[b] Undirected Graph

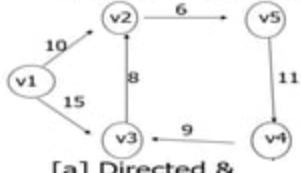
Non-Linear Data structures

Graph:

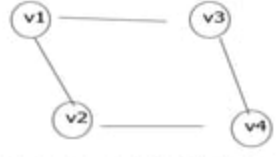
Graph is a mathematical non-linear data structure capable of representing many kind of physical structures:

- An edge connects a pair of vertices and many have weight such as length, cost and another measuring instrument for according the graph.
- Vertices on the graph are shown as point or circles and edges are drawn as arcs or line segment

Example of graph:



[a] Directed & Weighted Graph



[b] Undirected Graph

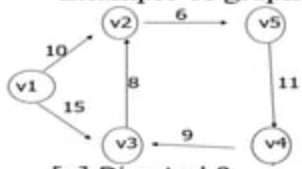
Non-Linear Data structures

Graph:

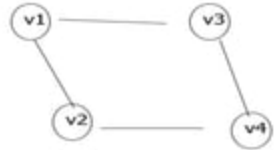
Types of Graphs:

- *Directed graph*
- *Undirected graph*
- *Simple graph*
- *Weighted graph*
- *Connected graph*
- *Non-connected graph*

Example of graph:



[a] Directed & Weighted Graph



[b] Undirected Graph

Data structures

Thank you: