Pointers

Introduction

- A pointer is a derived data type. This is the one which stores the address of data in memory, we will be in position to access the data directly and do calculations over it.
- The standard concept is, access the data from memory using variable name it gets the data and operations are done over them.
- But the pointer is different that the accessing is done by address the data is stored so that it will be advantage of decreasing the instructions and overheads of standard usage.

Definition:

A pointer is a variable which contains the address of another variable.

Advantages of pointer

- Enables us to access a variable that is defined outside the function.
- Can be used to pass information back and forth between a function and its reference point.
- More efficient in handling data tables.
- Reduces the length and complexity of a program.
- Sometimes pointers also increases the execution speed.

Declaring of a pointer variable

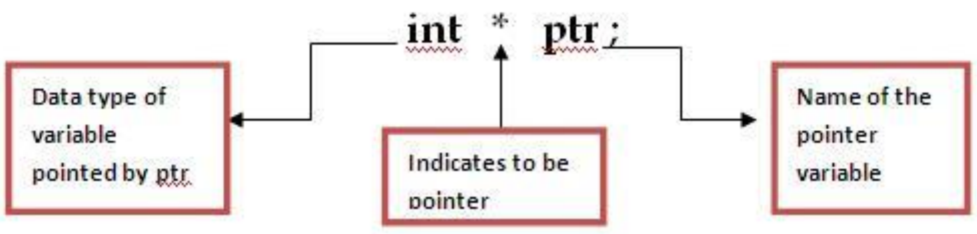
General form:

data_type *pointer_name;

where,

- The asterisk (*) tells that the variable pointer_name is a pointer variable.
- Pointer_name is a identifier.
- pointer_name needs a memory location.
- pointer_name points to a variable of type data_type which may be int, float, double etc..

Example:


int * ptr ;
Data type of variable pointed by ptr
Indicates to be pointer
Name of the pointer variable

where

- ptr is not an integer variable but ptr can hold the address of the integer variable i.e. it is a **pointer to an integer variable'** ,but the declaration of pointer variable does not make them point to any location .
- We can also declare the pointer variables of any other data types .

For example:

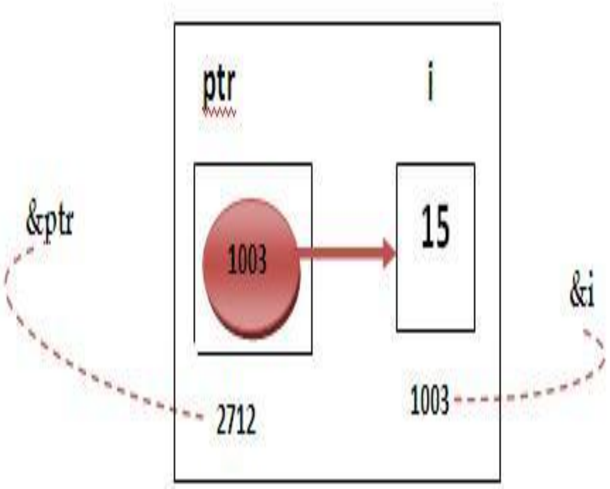
double * dptr;
char * ch;
float * fptr;

<u>Dereference operator (*)</u>

The unary operator (*) is the dereferencing pointer or indirection pointer , when applied to the pointer can access the value the pointer points to.

Example:

```
        int  i= 15;
        int * ptr;
        ptr=&i;
printf("Value of  i  is :%d",i);
printf("Address of  i  is :%d",&i);
printf("Value of  i  is :%d",*ptr);
printf("address of i  is :%d",ptr);
printf("address of pointer is :%x",&ptr);
```



```
Value of  i  is : 15
Address of i is : 1003
Value of i:  15
Address  of  i  is  : 1003
Address  of  ptr  is : 2712
```

<u>The null pointer</u>

- Sometimes it is useful to make our pointers initialized to nothing . This is called a **null pointer**.
- A null pointer is the pointer does not point to any valid reference or memory address .We assign a pointer a null value by setting it to address 0:
- **Example**

```
    int *iptr;
    iptr=0;        or              iptr = NULL;
```

- This statement assign address 0 to iptr . This statement describe a special preprocessor define called NULL that evaluates to 0.
- Hence A null pointer is a pointer type which has a special value that indicate it is not pointing to any valid reference.

<u>Initialization of pointers</u>

Initializing a pointer variable is a important thing, it is done as follows:

Step 1: Declare a data variable

Step 2:Declare a Pointer variable

Step 3:Assign address of data variable to pointer variable using & operator and assignment operator.

Example:

```
 int x;
int *p
p = &x;
```

Pointer arithmetic

Pointer arithmetic operations are different from normal arithmetic operations.

The operations allowed to on Pointers are:

1. Add or subtract integers to/from a pointer. The result is a pointer.
2. Subtract two pointers to the same type. The result is an int.
3. Comparison of two pointers

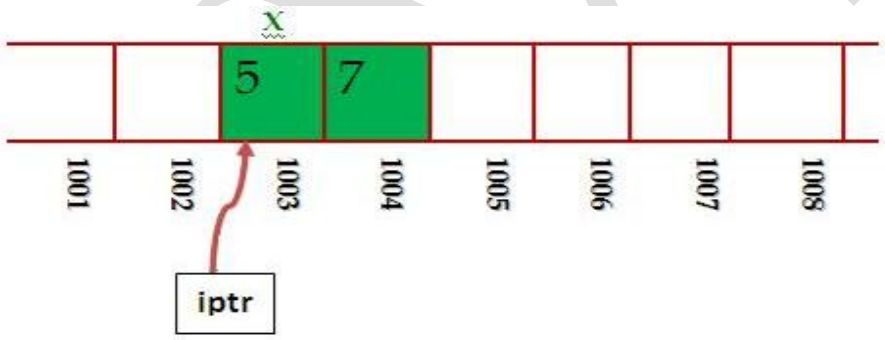
Note: Multiplication, addition, division of two pointers not allowed.

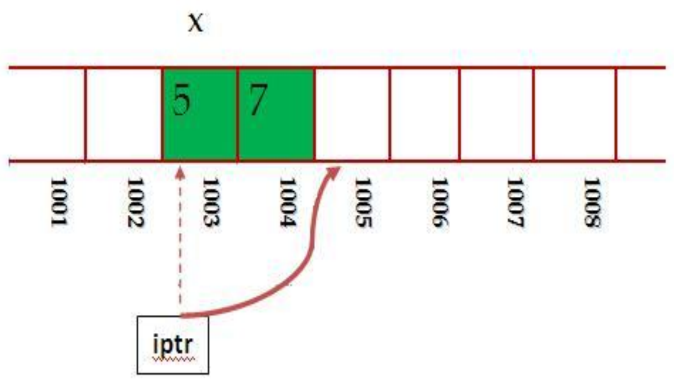
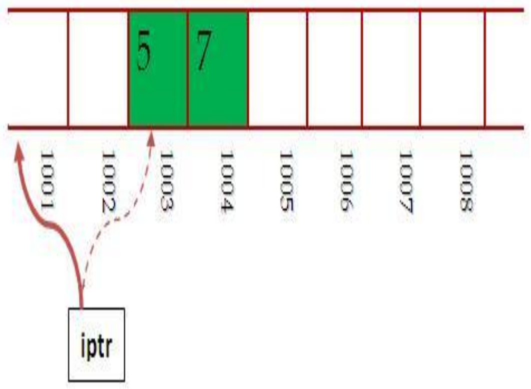
1. **Add or subtract integers to/from a pointer**

- ➢ Simple addition or subtractions operations can be performed on pointer variables.
- ➢ If ***iPtr** points to an integer, ***iPtr + 1** is the address of the next integer in memory after ***iPtr**. ***iPtr - 1** is the address of the previous integer before ***iPtr**

Examples:

int a=57;
int *iptr=&a;
iptr= iptr +1;
iptr = iptr +(2B)
iptr =1003+2
iptr =1005
	int a=57;
	int *iptr=&a;
	iptr= iptr -1;
	iptr = iptr -(2B)
	iptr =1003-2
	iptr =1001



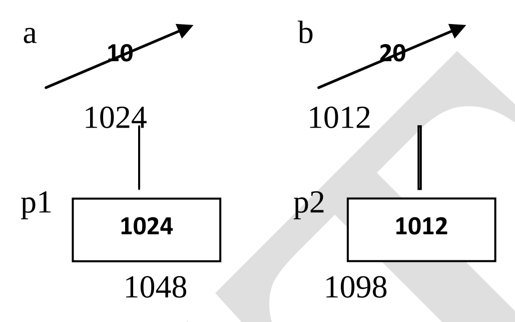
	

Note: Pointer is always incremented or decremented as per the type of value it is pointing to.

2.Subtract two pointers to the same type.

In C subtraction of two pointers are allowed.

Example:

int a=10,b=20;
int *p1,*p2;
p1=&a;
p2=&b;
int res=p1-p2;
 	
res=1024-1012
res=12
	 a
	10
	1024
	b
	20
	1012
	p1
	1024
	1048
	p2
	1012
	1098

3.Comparison between two Pointers :

I. **A pointer comparison is** valid only if the **two pointers are pointing to same array.**
II. All Relational Operators can be used for comparing pointers of **same type.**
III. **All Equality and Inequality Operators** can be used with all Pointer types.
IV. **Two Pointers addition,subtraction and division is not possible.**

Pointer Comparison example:

```
#include<stdio.h>
 void main()
 {
 int *ptr1,*ptr2;
 ptr1 = (int *)1000;
 ptr2 = (int *)2000;
 if(ptr2 > ptr1)
    printf("Ptr2 is far from ptr1");
 }
```

Pointers and functions

- ➢ Pointers are often passed to a function as arguments.
- ➢ Allows data items within the calling program to be accessed by the function, altered, and then returned to the calling program in altered form.(Called address or by location).
- ➢ When arguments are passed to a function by value.

– The data items are copied to the function.
– Changes are not reflected in the calling program.

Example:

```
#include<stdio.h>
void swap (int *x, int *y);
void  main()
 {
        int a, b;
        a = 5;
        b = 20;
        swap (&a, &b);
        printf ("\n a=%d, b=%d", a, b);

}
void swap (int *x, int *y)
{
        int t;
        t = *x;
        *x = *y;
        *y = t;
}
```

Pointer and arrays

When an array is declared,

- The compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations.
- The base address is the location of the first element (index 0) of the array.
- The compiler also defines the array name as a constant pointer to the first element.
- There is a strong relationship between the pointer and the arrays.
- Any operation which can be performed using array subscripting can also be performed using pointers .
- The pointer version will be generally faster than the array version of that program.
- To access the address of any element use (a+i).
- To access the element of array use *(a+i).

Program to read and display array elements using pointers

```
#include<stdio.h>

 void main()
{
       int a[100], i, n;
        printf("enter number of elements");
        scanf ("%d", &n);
        printf("enter array elements");
        for (i=0; i<n; i++)
```

```
                scanf("%d",(a+i));
        printf("enter array elements are");
        for (i=0; i<n;i++)
                printf("%d",*(a+i));
}
```

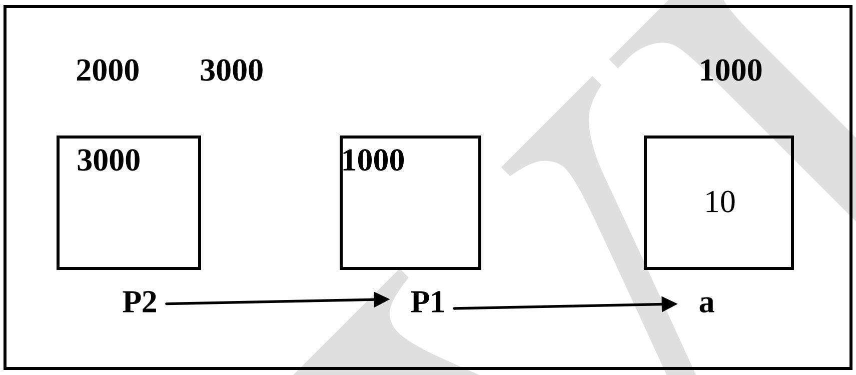
Pointers to pointers

- C allows the use of pointers that point to pointers, i.e in turn, point to data (or even to other pointers). In order to do that, add an asterisk (*) for each level of reference in their declarations:

Example

```
    int a=10;
    int *p1=&a;
    int **p2=&p1;
```

Pictorial representation of above example is:



- In the above example p1 is a pointer variable to a and p2 is the pointer to pointer variable for p1.
- p2 has the type int** and a value 3000
- p1 has the type int* and a value 1000
- a has the type int and a value 10

Pointers and structures

- Like array of integers, array of pointer etc, array of structure variables is also available in C.
- And to make the use of array of structure variables efficient, use **pointers of structure type**.

Example:

```
#include <stdio.h>
#include <string.h>
struct  student
{
char name[20];
int age;
float percentage;
};
struct student s1={"madan",12,89.5};
```

```
 void show_name(struct student  *p);
 void  main()
 {
 struct student *ptr;
  ptr = &s1;
 show_name(ptr);
 }
 void show_name(struct student *p)
 {
 printf("\n%s ", p->name);
 printf("%d ", p->usn);
 printf("%f\n", p->percentage);
}
```

<u>Memory Allocation</u>

Memory Allocation in C is of 2 types

1. Static Memory allocation
2. Dynamic Memory allocation

<u>1.Static Memory Allocation</u>

- If the memory is allocated for various variables during compilation time, then it is called as static memory allocation and memory allocated cannot be extended and cannot be reduced, it is fixed.

Example:

```
int a;
     int a[10];
     int *p;
```

Advantage: efficient execution time.

Disadvantage:

1. The memory is allocated during compilation time.Hence,the memory allocated is fixed and cannot be altered during execution time.
2. Leads to underutilization if more memory is allotted.
3. Leads to overflow if less memory is allocated.

<u>2.Dynamic Memory allocation</u>

- Dynamic memory allocation is the process of allocating memory during run time(execution time).
- Data structures can grow and shrink to fit changing data requirements.
- Additional storage can be allocated whenever needed.
- We can de-allocate the dynamic space whenever we done with them.
- To implementing dynamic memory the following 4 functions are used.

1. malloc( ): Allocates requested number of bytes and returns a pointer to the first byte of the allocated space.
2. calloc( ): Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory.
3. realloc( ): Modifies the size of previously allocated space.
4. free( ): Frees previously allocated space.

1. **malloc( ):**
   - ➢ Allocates requested number of bytes and returns a pointer to the first byte of the allocated space.

Syntax:

ptr=(datatype *)malloc(size);

where,

- ✓ ptr is a pointer variable of type datatype
- ✓ datatpe can be any of the basic datatype or user define datatype
- ✓ Size is number of bytes required.

Example:

```
int  *p;
p=(int *)malloc(sizeof(int));
```

2. **calloc( ):**

- ➢ It allocates a contiguous block of memory large enough to contain an array of elements of specified size. So it requires two parameters as number of elements to be allocated and for size of each element. It returns pointer to first element of allocated array.

Syntax:

ptr=(datatype *)calloc(n,size);

where,

- ✓ ptr is a pointer variable of type datatype
- ✓ datatype can be any of the basic datatype or user define datatype
- ✓ n is number of blocks to be allocated
- ✓ Size is number of bytes required.

Example:

```
int  *p;
p=(int *)calloc(sizeof(5,int));
```

<u>3.realloc( )</u>

- ➢ realloc() changes the size of block by deleting or extending the memory at end of block.
- ➢ If memory is not available it gives complete new block.

Syntax:

ptr=(datatype *)realloc(ptr,size);

where,

- ✓ ptr is a pointer to a block previously allocated memory either using malloc() or calloc()
- ✓ Size is new size of the block.

Example:

```
int  *p;
p=(int *)calloc(sizeof(5,int));
p=(int *)realloc(p,sizeof(int *8));
```

If enough space doesn‟t exist in memory of current block to extend, new block is allocated for the full size of reallocation, then copies the existing data to new block and then frees the old block.

4.free()

- This function is used to de-allocate(or free) the allocated block of memory which is allocated by using functions malloc(),calloc(),realloc().

Syntax:

```
free(ptr);
```

Note:

It is the responsibility of a programmer to de-allocate memory whenever not required by the program and initialize **ptr** to **Null.**

Difference between Static and Dynamic memory allocation

Static Memory Allocation	Dynamic Memory Allocation
1.Memory is allocated during compilation time 2.The size of the memory to be allocated is fixed during compilation time and cannot be altered during execution time 3.Execution is faster 4.Used only when the data size is fixed and known in advance before processing.	1.Memory is allocated during run time 2.When required memory can be allocated and when not required memory can be de-allocated 3.Execution is slower 4.Can be used when memory requirement is unpredictable.

INTRODUCTION TO DATA STRUCTURES

Data Structure:

- An implementation of abstract data type is data structure i.e. a mathematical or logical model of a particular organization of data is called data structure.
- A data structure deals with the study of how data is organized in memory so that it can be retrieved and manipulated efficiently.

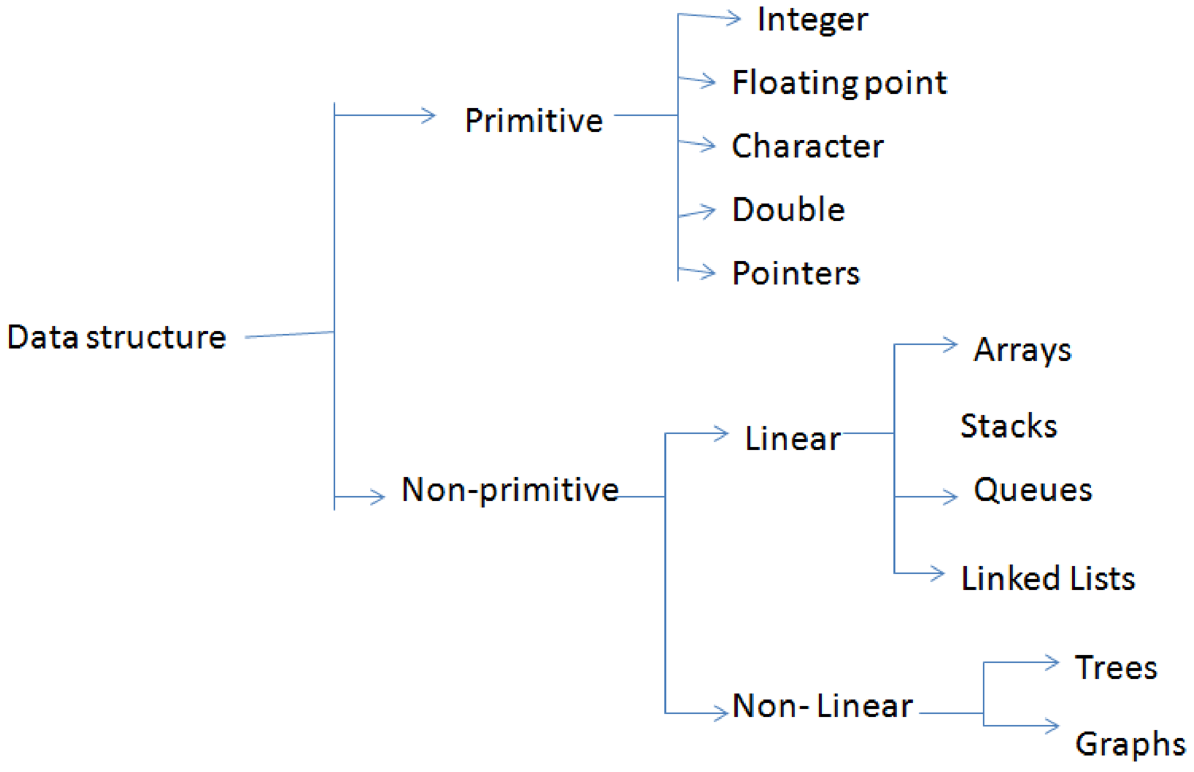
Data structures and its use:

The main focus of data structures are:

- ✓ The study of how data is organized in the memory.
- ✓ How efficiently data can be retrieved and manipulated.
- ✓ The possible ways in which different data items are logically related.

Types of data structure:

A data structure can be broadly classified as shown bellow



I. Primitive data structure

- The data structures, that are directly operated upon by machine level instructions i.e. the fundamental data types such as int, float, double in case of 'c' are known as primitive data structures.

Data Types and Sizes:

There are only a few basic/fundamental data types available in C:

char- a single byte, capable of holding one character in the local character set.

int - an integer, typically reflecting the natural size of integers on the host machine.

float- single-precision floating point.

double- double-precision floating point.

void- does not return any value.

(i) Non-primitive data structure

The data structures, which are not primitive, are called non-primitive data structures.

There are two types of-primitive data structures.

The non-primitive data types cannot be manipulated by machine instructions.

- **Linear Data Structures:-**

A list, which shows the relationship of adjacency between elements, is said to be linear data structure. The most, simplest linear data structure is a 1-D array, but because of its deficiency, list is frequently used for different kinds of data.

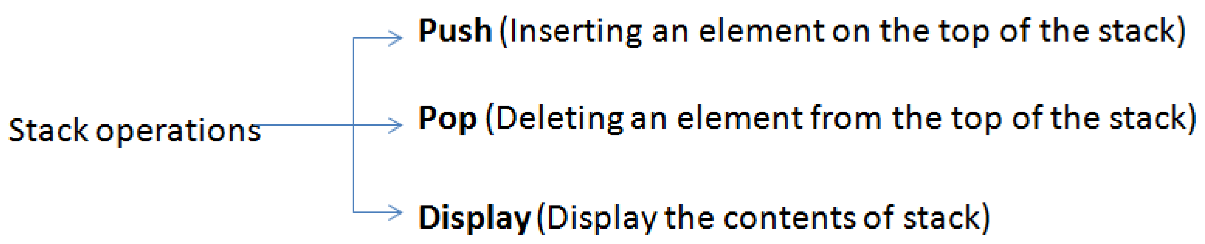
- **Non-linear data structure:-**

A list, which doesn't show the relationship of adjacency between elements, is said to be non-linear data structure.

Stacks

➔ A stack is a linear data structure in which an element may be inserted or deleted only at one end called the top end of the stack i.e. the elements are removed from a stack in the reverse order of that in which they were inserted into the stack.

➔ A stack follows the principle of last-in-first-out (LIFO) system. The various operations that can be performed on stacks are shown below:


Stack operations
Push (Inserting an element on the top of the stack)
Pop (Deleting an element from the top of the stack)
Display (Display the contents of stack)

- **Representation of Stacks**

A stack may be represented by means of a one way list or a linear array. We need to define two more variables the size of stack i.e, MAX and the entry or exit of stack i.e., top.

Push: inserting the element to the stack

- Here we need to check the stack overflow condition which means whether the stack is full.
- To insert an element two activities has to be done
  1. Increment the top by 1
  2. Insert the element to the stack at the position top.

Pop: deleting an element from the stack

- Here we need to check the stack underflow condition which means whether the stack is empty or not.
- To delete the element we need to perform following operations.
  1. Access the top element from the stack.
  2. Decrement top by 1

Display: printing the elements of the stack

- Here we need to check the stack underflow condition which means whether the stack is empty or not. If empty there will be no elements to display.
- Display the elements from the 0th position of stack till the stack top.

- **Application of Stacks**

There are two applications of stacks.

a) Recursion: A recursion function is a function which calls itself. The problems like towers of Hanoi, tree manipulation problems etc can be solved using recursion.

b) Arithmetic/Evaluation of Expression: The conversion of an expression in the form of either postfix or prefix can be easily evaluated.

c) Conversions of expressions: Evaluation of infix expressions will be very difficult and hence it needs to be converted to prefix or postfix expression which needs the use of stack.

Queue

- ➔ Queue is a linear data structure in which insertion can take place at only one end called rear end and deletion can take place at other end called top end.
- ➔ The front and rear are two terms used to represent the two ends of the list when it is implemented as queue.

- ➔ Queue is also called First In First Out (FIFO) system since the first element in queue will be the first element out of the queue.
- ➔ Types of queues are

1. Queue (ordinary queue)
2. Circular queue
3. Double ended queue
4. Priority queue

Queue

Here the elements are inserted from one end and deleted from other end. The inserting end is the rear end and the deleting end is the front end.

The operations that can be performed on queue are.

- ∗ Insert an element at rear end
- ∗ Delete an element from the front end
- ∗ Display elements

Circular Queue

In circular queue the elements of queue can be stored efficiently in an array so as to wrap around so that the end of the queue is followed by front of queue.

The operations that can be performed on queue are.

- ∗ Insert an element at rear end
- ∗ Delete an element from the front end
- ∗ Display elements

Double Ended Queue

A Double Ended Queue is in short called as Deque (pronounced as Deck or dequeue). A deque is a linear queue in which insertion and deletion can take place at either ends but not in the middle. The operations that can be performed are

- ∗ Insert an item from the front end
- ∗ Insert an item from rear end.
- ∗ Delete element from front end
- ∗ Delete an element from rear end
- ∗ Display the elements

Priority Queue

A priority queue is a collection of elements such that each element has been assigned a priority value such that the order in which elements are deleted and processed based on the assigned priority.

There are two different types of priority queue

1. Ascending priority queue: Elements can be inserted in any order but deletion is done in the ascending order of values.
2. Descending priority queue: Elements can be inserted in any order but deletion is done in the descending order of values.

Applications of Queues:

a) Number of print jobs waiting in a queue, when we use network printer. The print jobs are stored in the order in which they arrive. Here the job which is at the front of the queue, gets the services of the network printer.

b) Call center phone system will use a queue to hold people in line until a service representative is free.

c) Buffers on MP3 players and portable CD player, iPod playlist are all implemented using the concept of a queue.

d) Movie ticket counter system, it maintains a queue to take tickets based on first come first serve means who is standing first in a queue, that person will get the ticket first than second person, and so on.

Arrays

Array is a data structure, which is a collection of elements of same datatype.

Advantages :

- Data accessing is faster.
- Simple o understand and use

Disadvantages:

- The size of the array is fixed
- Array items are stored continuously.
- Insertion and deletion of an array element in between is a tedious job.

Linked list

A linked list is a data structure which is collection of zero or more nodes with each node consisting of two field's data and link.

- Data field consists the information of be processed.
- Link field contains the address of the next node.

Types of linked list are

1. Singly Linked List
2. Doubly Linked List
3. Circular Singly Linked List
4. Circular Doubly Linked List

Singly Linked List:

A singly linked list is a linked list, where each node has designated field called link field which contains address of next node. Since there is only one link field it is termed as singly linked list.

The various operations of singly linked list are

- Inserting a node into a list
- Deleting a node from a list
- Search in a list
- Reverse a list
- Display the content of the list

Circular singly linked list

In the singly linked list the last node link field consists of the NULL(\0), if it contains the address of the first node then it is termed as circular linked list.

Doubly Linked List

It is collection of nodes where each node consists of three fields

- Info→ which contains the information to be processed
- Llink→pointer which contains address of the left node or previous node in the list
- Rlink→ pointer which contains address of the right node link or next node in the list

The main purpose of doubly linked list is its very easy to traverse in any directions either forward or backward direction of the list.

Circularly Doubly linked list

It is the variation of doubly linked list where

- Rlink of the last node contains address of first node
- Llink of the first node contains address of the last node

Circularly Doubly linked list with header

It is the variation of the doubly linked list with the header where

- Llink of header contains address of the last node of the list
- Rlink of header contains address of the first node of the list
- Llink of the last node contains address of last but one node of the list
- Rlink of the last node contains address of the first node of list

Application of Linked List:

1) **Polynomial Manipulation like polynomial addition, subtraction, multiplication can be done easily.**
2) **Linked Dictionary**
3) **Addition of long numbers can be done easily**

Trees

A tree is a nonlinear data structure and is generally defined as a nonempty finite set of elements, called nodes such that:

1. It contains a distinguished node called *root* of the tree.
2. The remaining elements of tree form an ordered collection of zero or more disjoint subsets called sub tree

Binary Tree:

A binary tree is defined as a finite set of elements, called nodes, such that:

1) Tree is empty (called the null tree or empty tree) or
2) Tree contains a distinguished node called root node, and the remaining nodes form an ordered pair of disjoint binary trees

Now let us see **"What are the different ways of traversing the tree?"**

The tree can be traversed in 3 different ways:

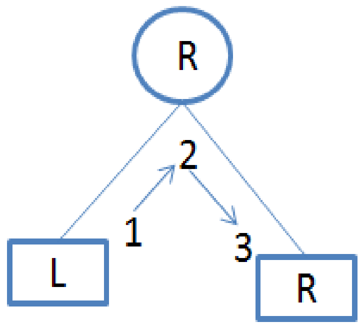
- Inorder traversal
- Preorder traversal
- Postorder traversal

Inorder traversal: Let us see "What is inorder traversing of a binary tree?"

Definition: The inorder traversal of a binary tree can be recursively defined as follows:

1. Traverse the Left subtree in inorder [L]
2. Process the root Node [N]
3. Traverse the Right subtree in inorder [R]

Pictorial representation of inoder traversal is shown below:

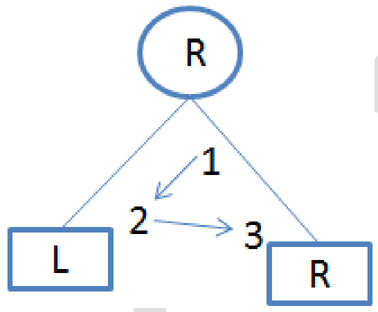


Preorder traversal: Let us see "What is preorder traversing of a binary tree?"

Definition: The preorder traversal of a binary tree can be recursively defined as follows:

1. Process the root Node [N]
2. Traverse the Left subtree in preorder [L]
3. Traverse the Right subtree in preorder [R]

Pictorial representation of preoder traversal is shown below:

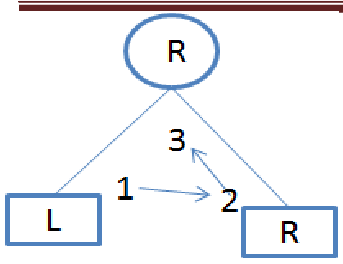


Postorder traversal: Let us see "What is postorder traversing of a binary tree?"

Definition: The postorder traversal of a binary tree can be recursively defined as follows:

1. Traverse the Left subtree in postorder [L]
2. Traverse the Right subtree in preorder [R]
3. Process the root Node [N]

Pictorial representation of postoder traversalis shown below:



Types of binary trees

- **Complete Binary Tree**

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

- **Full binary tree**

A binary tree said to be full if all its level have maximum number of possible node.

- **Extended Binary Tree (Strictly Binary Tree or 2-tree)**

A binary tree is said to be extended binary tree if each node has either 0 or 2 children. In this case the leaf nodes are called external nodes and the node with two children are called internal nodes.

- **Binary Search Trees**

A tree is called binary search tree if each node of the tree has following properties.

The value at a node is greater than every value in the left subtree and is less than every value in the right subtree.

- **Expression tree**

Application of Binary Tree:

1). Symbol Table Construction:

2). Manipulation of the Arithmetic Expression

3). Searching and sorting

4). syntax analysis of compiler design and to display structure of sentence in a language.

Introduction to Preprocessor Directives

5.1 Preprocessor Directives:

- Preprocessor Directives are the lines included in the C program that starts with character #
- These are the instructions(also called as directives) to the preprocessor.
- The # symbol is followed by **directive name or an instruction name.**

Advantages of preprocessor are:

- ✓ Programming becomes simple.
- ✓ Program becomes easy to modify and easy to read.

5.2 Types of preprocessor

1. **Symbolic names**
2. **Macros**
3. **File inclusion**
4. **Conditional compilation**

5.2.1 Symbolic names/symbolic constants

- These are the names which are used to define/give the names for a constant values.
- The ***# define*** directive is used to define the names for a constant value.
- Since it is used to define constant, it is also termed as **defined constant.**

# define name value

Where

- **# define is a directive**
- **name is symbolic name**

Example # define MAX 30

# define PI 3.14

5.2.2 Macros

- Macro is **a name** given to the group of statements.
- When a macro is included in the program, the program replaces the set of instructions defined .
- The ***# define*** directive is used to **define a macro.**
-

Syntax:

# define <macro_name> set of statements

Example1: To find maximum of two variables.

# define max (a, b) ((a>b)?a: b)

(statements for this macro)

macro name

Example2 :

/*Program to find a square of a number using macro */

```
#include<stdio.h>
        #define square (x)    ((x)*(x))
        void main( )
        {
                int m=5;
                printf("square of m is %d",square(m));
        }
```

In the above example square(x) is the name of the macro which contains x*x as a statement.

5.2.3 File inclusion

- It is a include preprocessor directives
- #include specifies to insert the content of the specified files to the program.
- This directive includes a file in to the code.
- It has two possible forms

```
#include <file>
        Or
#include"file"
Example:
#include<stdio.h>
void main()
{
        printf("ECE,EEE,CIV\n");
}
```

5.2.4 Conditional compilation

- The #if, #else ,#endif ,#undef, #ifdef, #ifndef are some of the conditional compilation directives.

#if	it tests a compile time condition
#ifdef	tests for a macro definition
#ifndef	tests whether a macro has been defined or not
#undef	undefines a macro

Example

#include<stdio.h>	
void main( )	
{	
#if((10%2)==0)	
printf(“num is even\n”);	// num is even is executed
#else	
printf(“num is odd\n”);	
#endif	
}	