Module-1: Introduction to Data Structures

Notes

Structure:

Unit-1.1: Introduction to Data Structures

1.1.1 Definition and Types of Data structure

1.1.2 Algorithm Design

1.1.3 Complexity, Time-Space Tradeoffs

1.1.4 Use of pointers in data structures

Unit-1.2: Arrays

1.2.1 Array Definition and Analysis

1.2.2 Representation of Linear Arrays in Memory

1.2.3 Traversing of Linear Arrays

1.2.4 Insertion and Deletion

1.2.5 Single Dimensional Arrays

1.2.6 Two Dimensional Arrays

1.2.7 Multidimensional Arrays

1.2.8 Function Associated with Arrays

Unit-1.3: Strings

1.3.1 Character String in C

1.3.2 Character String Operations

1.3.4 Arrays as parameters

Unit-1.4: Array Implementation

1.4.1 Implementing One Dimensional Array

1.4.2 Sparse matrix

Notes

Unit-1.1: Introduction to Data Structures

1.1.1 Definition and Types of Data structure

Data structure is a representation of data and the operations allowed on that data. A data structure is a way to store and organize data to facilitate the access and modifications. The data structure name indicates itself that organizing the data in memory. There are many ways of organizing the data in the memory. Data Structure are the method of representing of logical relationships between individual data elements related to the solution of a given problem. To structure the data in memory, 'n' number of algorithms were used, and all these algorithms are known as Abstract data types.

A data structure is a type to store, organize, process, retrieve to facilitate efficient access and modification. Here are several basic and advanced structure types.

The choice of data model depends on two consideration:

- It must be rich enough in structure to represent the relationship between data elements.
- The structure should be simple enough that one can effectively process the data when necessary.

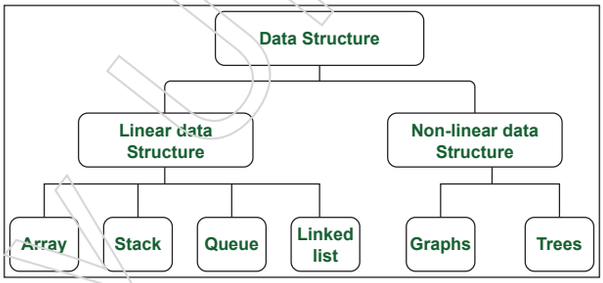


Fig 1.1: Types of data structures

There are two types of data structures: The primitive data structures are primitive data types. The int, char, float, double, and pointer are the primitive data structures that can hold a single value. The non-primitive data structure is divided into two types: Linear data structure & Non-linear data structure.

Other than linear and non-linear there are below more types of data structures available.

Homogeneous or Heterogeneous data structures: When all data items in each repository are of the same type or of different types. The simplest example of such type of data structures is an Array. Heterogeneous Data Structures are those data structures that contain a variety or dissimilar type of data, for e.g. a data structure that can contain various data of different data types like integer, float and character. The examples of such data structures include structures, union etc.

Static or dynamic data structures: This type tells how the data structures are compiled. Static data structures have fixed sizes and Dynamic data structures have flexible size. In Static data structure, the content of the data structure can be modified

Notes

but without changing the memory space allocated to it. Dynamic data structure can be modified during the operations performed on it. Dynamic data structures are designed to facilitate change of data structures in the run time.

Linear Data Structures: In Linear data structure, values are arranged in linear fashion. The data structures used for this purpose are Arrays, Linked list, Stacks, and Queues. In these data structures, one element is connected to only one another element in a linear form.

- **Array:** A collection of items when we store at adjoining memory locations is called an Array. Same type of elements gets stored together so that the position of each element can be calculated or retrieved easily. Arrays can be of fixed or flexible length. It is a finite group of data and is allocated contiguous memory locations. To access an element within the array can be accessed by an index key.

Example: The syntax for storing and displaying the values in an array typically looks something like this:

```
arrayname[0] = “This “;

arrayname[1] = “is “;

arrayname[2] = “pretty simple.”;

print arrayname[0];

print arrayname[1];

print arrayname[2];
```

The above commands would print the first three values of the array, or “This is pretty simple.” By using a “while” or “for” loop, the programmer can tell the program to output each value in the array until the last value has been reached. So not only do arrays help manage memory more efficiently, they make the programmer’s job more efficient as well.

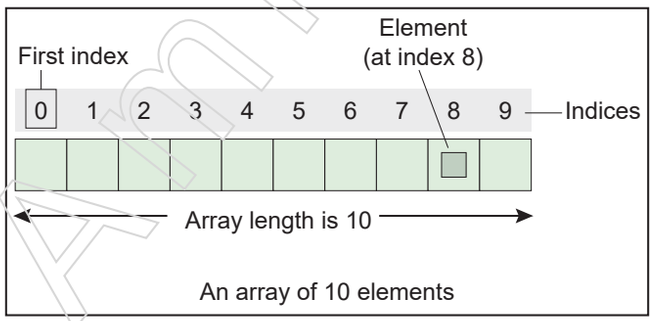


Fig 1.2: Array

- **Stack:** When a collection of items is stored in linear order which follows a particular order in which the operations are applied is called stack. This order could be last in first out (LIFO) or first in first out (FIFO). In LIFO, elements are added to top and removed from top.

Notes

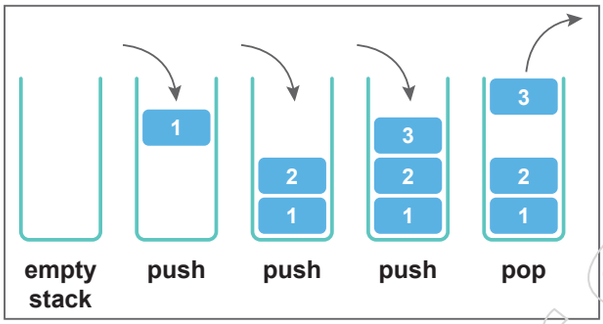


Fig 1.3: Stack

- **Linked-list:** Linked list is also a linear data structure and it is variable-sized data structures that means we can insert or delete data from beginning, middle or end of linked list. Linked list contains data elements along with its pointer which points to the next data element. In this case memory locations are not contiguous. In linked list

Linked List contains a link called first. Each link carries a data field and a link field. Link field is called next. Each link is linked with its next link using its next link. Last link carries a link as null to mark the end of the list. The various types of linked list are Simple Linked List, Doubly Linked List, Circular Linked List.

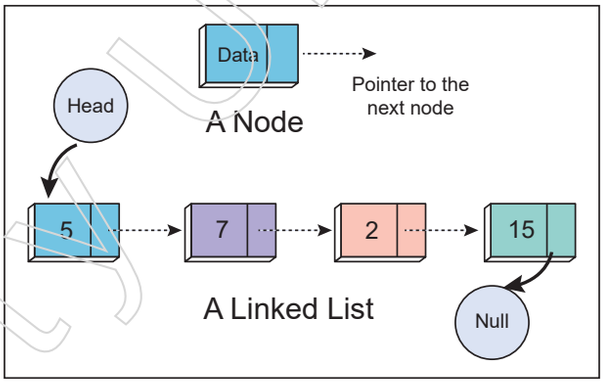


Fig 1.4: Linked list

- **Queue:** A queue stores a collection of items like a stack, though, the operation is done only by first in first out (FIFO) order. That means we can add elements to back and remove an element from front.

Notes

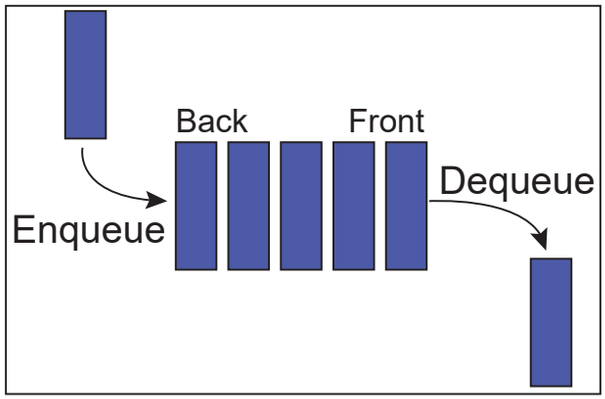


Fig 1.5: Queue

Non-Linear Data Structures: The data values in this structure are not arranged in order.

- **Hash tables:** Hash tables, also known as hash map, stores a collection of data in an associative manner. Array items are associated with keys. Items are in (key, value) format. It is an unordered list which use a 'hash function' to insert and search.
- **Tree:** Data is organized in branches.
- **Graph:** A more general branching structure, with less strict connection conditions than for a tree

Liner Vs. Non Linear Data Structure:

1.	In a linear data structure, data elements are arranged in a linear order where each and every element is attached to its previous and next adjacent.	In a non-linear data structure, data elements are attached in hierarchically manner.
2.	In linear data structure, single level is involved.	Whereas in non-linear data structure, multiple levels are involved.
3.	Its implementation is easy in comparison to non-linear data structure.	While its implementation is complex in comparison to linear data structure.
4.	In linear data structure, data elements can be traversed in a single run only.	While in non-linear data structure, data elements can't be traversed in a single run only.
5.	In a linear data structure, memory is not utilized in an efficient way.	While in a non-linear data structure, memory is utilized in an efficient way.
6.	Its examples are: array, stack, queue, linked list, etc.	While its examples are: trees and graphs.
7.	Applications of linear data structures are mainly in application software development.	Applications of non-linear data structures are in Artificial Intelligence and image processing.

Table 1.1 Linear Vs. Non linear data structure

Notes

1.1.2 Algorithm Design

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e., an algorithm can be implemented in more than one programming language.

Characteristics of an Algorithm

Not all procedures can be called an algorithm. An algorithm should have the following characteristics −

- Unambiguous − Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- Input − An algorithm should have 0 or more well-defined inputs.
- Output − An algorithm should have 1 or more well-defined outputs and should match the desired output.
- Finiteness − Algorithms must terminate after a finite number of steps.
- Feasibility − Should be feasible with the available resources.
- Independent − An algorithm should have step-by-step directions, which should be independent of any programming code.

1.1.3 Complexity, Time-Space Tradeoffs

Algorithm Analysis

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following − Asymptotic Analysis is the big idea that handles above issues in analyzing algorithms. In Asymptotic Analysis, we evaluate the performance of an algorithm in terms of input size (we don't measure the actual running time). We calculate, how the time (or space) taken by an algorithm increases with the input size. For example, let us consider the search problem (searching a given item) in a sorted array. One way to search is Linear Search (order of growth is linear) and the other way is Binary Search (order of growth is logarithmic).

To understand how Asymptotic Analysis solves the above mentioned problems in analyzing algorithms, let us say we run the Linear Search on a fast computer A and Binary Search on a slow computer B and we pick the constant values for the two computers so that it tells us exactly how long it takes for the given machine to perform the search in seconds. Let's say the constant for A is 0.2 and the constant for B is 1000 which means that A is 5000 times more powerful than B. For small values of input array size n, the fast computer may take less time. But, after a certain value of input array size, the Binary Search will definitely start taking less time compared to the Linear Search even though the Binary Search is being run on a slow machine. The reason is the order of growth of Binary Search with respect to input size is logarithmic while the order of growth of Linear Search is linear. So the machine dependent constants can always be ignored after a certain value of input size.

A complete analysis of the running time of an algorithm involves the following steps:

Notes

- Implement the algorithm completely.
- Determine the time required for each basic operation.
- Identify unknown quantities that can be used to describe the frequency of execution of the basic operations.
- Develop a realistic model for the input to the program.
- Analyze the unknown quantities, assuming the modelled input.
- Calculate the total running time by multiplying the time by the frequency for each operation, then adding all the products.
- A Priori Analysis − This is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.
- A Posterior Analysis − This is an empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

Algorithm Complexity

Suppose X is an algorithm and n is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.

- Time Factor − Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
- Space Factor − Space is measured by counting the maximum memory space required by the algorithm.

Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components −

- A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.
- A variable part is a space required by variables; whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.

Time Complexity

Time Complexity of an algorithm is the representation of the amount of time required by the algorithm to execute to completion. Time requirements can be denoted or defined as a numerical function t(N), where t(N) can be measured as the number of steps, provided each step takes constant time.

For example, in case of addition of two n-bit integers, N steps are taken. Consequently, the total computational time is t(N) = c*n, where c is the time consumed for addition of two bits. Here, we observe that t(N) grows linearly as input size increases.

Notes

In divide and conquer approach, a problem is divided into smaller problems, then the smaller problems are solved independently, and finally the solutions of smaller problems are combined into a solution for the large problem.

Generally, divide-and-conquer algorithms have three parts –

- Divide the problem into a number of sub-problems that are smaller instances of the same problem.
- Conquer the sub-problems by solving them recursively. If they are small enough, solve the sub-problems as base cases.
- Combine the solutions to the sub-problems into the solution for the original problem.
- Divide and conquer approach supports parallelism as sub-problems are independent. Hence, an algorithm, which is designed using this technique, can run on the multiprocessor system or in different machines simultaneously.
- In this approach, most of the algorithms are designed using recursion, hence memory management is very high. For recursive function stack is used, where function state needs to be stored.

Following are some problems, which are solved using divide and

conquer approach.

- Finding the maximum and minimum of a sequence of numbers
- Strassen's matrix multiplication
- Merge sort
- Binary search

Problem Statement

The Max-Min Problem in algorithm analysis is finding the maximum and minimum value in an array.

Solution

To find the maximum and minimum numbers in a given array numbers[] of size n, the following algorithm can be used. First we are representing the naive method and then we will present divide and conquer approach.

1.1.4 Use of Pointers in Data Structures

Any variable we deal with has memory location(address) to store its value, and the pointer is something like a variable that holds that memory address of the variable. One can easily find the page using the location referred to there. Such pointers usage helps in the dynamic implementation of various data structures such as stack or list. Optimization of our code and improving the time complexity of one algorithm. Since using pointers helps to reduce the time needed by an algorithm to copy data from one place to another. Since it used the memory locations directly, any change made to the value will be reflected at all the locations.

Like we have pointers to int, char and other data-types, we also have pointers pointing to structures. These pointers are called structure pointers.

Notes

Now, how to define a pointer to a structure? The answer is below:

```
struct structure_name
{
   data-type member-1;
   data-type member-1;
   data-type member-1;
   data-type member-1;
};
int main()
{
   struct structure_name *ptr;
}
```

Notes

Unit-1.2: Arrays

1.2.1 Array Definition and Analysis

An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).

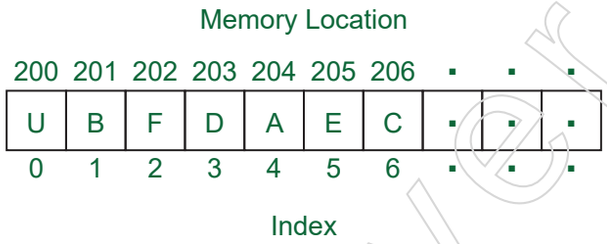


Fig 1.6: Index

1.2.2 Representation of Linear Arrays in Memory

A linear array is a list of finite numbers of elements stored in the memory. In a linear array, we can store only homogeneous data elements. Elements of the array form a sequence or linear list, that can have the same type of data.

Each element of the array is referred by an index set. The total number of elements in the array list, is the length of an array. We can access these elements with the help of the index set. For example, consider an array of employee names with the index set from 0 to 7, which contains 8 elements as shown in fig:

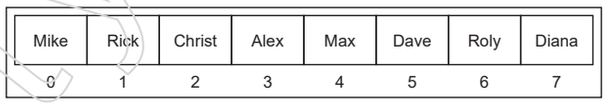


Fig 1.7: Array of employee names along with index value

Now, if we want to retrieve the name 'Alex' from the given array, we can do so by calling "employees [3]". It gives the element stored in that location, that is 'Alex', and so on.

We can calculate the length, or a total number of elements, of a linear array (LA), by the given formula:

Length (LA)=UB-LB+1

Here, UB refers to the upper bound, the largest index set of the given array list. LB refers to the lower bound, smallest index set of the given array list.

For example, we have an array of employees, in which lower bound is 153 and the upper bound is 234, so we can calculate the total number of elements in the list, by giving the formula.

234-153+1=82

Notes

1.2.3 Traversing of Linear Arrays

We can process each element of an array with the help of an index set. The array elements are stored in contiguous (i.e. one after the other) memory locations. All the array elements must be of one particular data types like either int or, float or, char or, double etc. or they can be of any user-defined data types like structures and unions.

Example:

```
/*Array traversal*/
#include <stdio.h>
 int main()
 {
   int arr[5],i;
   /* Getting Array Elements From User*/
   printf("Enter array elements: ");
   for(i=0;i<5;i++){
     scanf("%d",&arr[i]);
     printf("\n");
   }
   /* Traversing Array and Displaying them */
   for(i=0;i<5;i++){
     printf("\n arr[%d] = %d",i, arr[i]);
   }
   return 0;
 }
```

Output:

```
Enter array elements: 10 20 30 40 50
arr[0] = 10
arr[0] = 20
arr[0] = 30
arr[0] = 40
arr[0] = 50
```

1.2.4 Insertion and Deletion

Array insertion, of an element at the end of the list, is quite simple. We can do so by providing the unallocated space to the new element. On the other hand, if we

Notes

need to insert an element in the middle of an array, lots of internal shifting is required to insert the new element. i.e., half of the elements must be moved downward, to the new location, to enter the new element.

Similarly, deletion of the element from the end of the array, is quite simple. But, if we need to delete an element from the middle of the array, then on average, half of the elements must be moved upward, to fill the blank space, after deleting the element from the specified location.

1.2.5 Single Dimensional Arrays

An array is a collection of elements of one specific type in a horizontal fashion. The array in contention here is that of the one-dimensional array in C programming.

Anything having one-dimension means that there is only one parameter to deal with. In regular terms, it is the length of something. Similarly, as far as an array is concerned, one dimension means it has only one value per location or index.

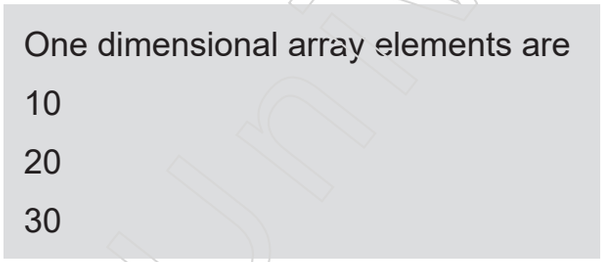


Fig 1.8: 1-D array

As you can see in the example given above, firstly, you need to declare the elements that you want to be in the specified array. Secondly, the location of each element needs to particularize as well, since that is where the elements will be stored respectively.

Syntax of declaring one dimensional array in C:

data_type array_name[array_size];

Here is the general form to initialize values to one dimensional array in C: data_type array_name[array_size] = {comma_separated_element_list};

Example:

```
/* One Dimensional Array */
#include<stdio.h>
int main()
{
    int arr[5] = {1, 2, 3, 4, 5};
    int i;
    for(i=0; i<5; i++)
    printf("\nElements of One dimensional array is: %d", arr[i]);
```

Notes

```
    return 0;
}
```

Output:

Elements of One dimensional array is: 1

Elements of One dimensional array is: 2

Elements of One dimensional array is: 3

Elements of One dimensional array is: 4

Elements of One dimensional array is: 5

1.2.6 Two Dimensional Arrays

An array keeps track of multiple pieces of information in linear order, a one-dimensional list. However, the data associated with certain systems (a digital image, a board game, etc.) lives in two dimensions. To visualize this data, we need a multi-dimensional data structure, that is, a multi-dimensional array. A two-dimensional array is really nothing more than an array of arrays (a three-dimensional array is an array of arrays of arrays). Think of your dinner. You could have a one-dimensional list of everything you eat: (lettuce, tomatoes, steak, mashed potatoes, cake, ice cream) Or you could have a two-dimensional list of three courses, each containing two things you eat: (lettuce, tomatoes) and (steak, mashed potatoes) and (cake, ice cream)

In the case of an array, our old-fashioned one-dimensional array looks like this: int[] myArray = {0,1,2,3};

And a two-dimensional array looks like this:

int[][] myArray = { {0,1,2,3}, {3,2,1,0}, {3,5,6,1}, {3,8,3,4} };

For our purposes, it is better to think of the two-dimensional array as a matrix. A matrix can be thought of as a grid of numbers, arranged in rows and columns, kind of like a bingo board. We might write the two-dimensional array out as follows to illustrate this point:

```
int[][] myArray = {  {0, 1, 2, 3},
                     {3, 2, 1, 0},
                     {3, 5, 6, 1},
                     {3, 8, 3, 4}  };
```

Example:

```
#include<stdio.h>
int main(){
int a=0,b=0;
int arr[4][3]={{1,2,3},{2,3,4},{3,4,5},{4,5,6}};
//traversing 2D array
```

Notes

```
for(a=0;a<4;a++){
 for(b=0;b<3;b++){
   printf("arr[%d] [%d] = %d \n",a,b,arr[a][b]);
 }j
}
return 0;
}
```

Output:

```
arr[0] [0] = 1
arr[0] [1] = 2
arr[0] [2] = 3
arr[1] [0] = 2
arr[1] [1] = 3
arr[1] [2] = 4
arr[2] [0] = 3
arr[2] [1] = 4
arr[2] [2] = 5
arr[3] [0] = 4
arr[3] [1] = 5
arr[3] [2] = 6
```

1.2.7 Multidimensional Arrays

In C programming, you can create an array of arrays. These arrays are known as multidimensional arrays. For example, x is a two-dimensional (2d) array. The array can hold 12 elements. You can think the array as a table with 3 rows and each row has 4 columns.

	Column 1	Column 2	Column 3	Column 4
Row 1	× [0] [0]	× [0] [1]	× [0] [2]	× [0] [3]
Row 2	× [1] [0]	× [1] [1]	× [1] [2]	× [1] [3]
Row 3	× [2] [0]	× [2] [1]	× [2] [2]	× [2] [3]

Fig 1.9: Multidimensional array

Multidimensional arrays use additional subscripts for indexing. A three-dimensional array, for example, uses three subscripts:

- The first references array dimension 1, the row.

Notes

- The second references dimension 2, the column.
- The third references dimension 3. This illustration uses the concept of a page to represent dimensions 3 and higher.

Example:

```
/*Multi-dimensional array*/
#include<stdio.h>
int main()
{
int a, b, c;
int arr[3][3][3]=
        {
          {
                {1, 2, 3},
                {4, 5, 6},
                {7, 8, 9}
          },
          {
                {11, 12, 13},
                {14, 15, 16},
                {17, 18, 19}
                },
          {
                {21, 22, 23},
                {24, 25, 26},
                {27, 28, 29}
                },
        };
printf("3D Array Elements:\n\n");
for(a=0;a<3;a++)
{
    for(b=0;b<3;b++)
    {
            for(c=0;c<3;c++)
```

Notes

```
                {
                printf("%d\t",arr[a][b][c]);
                }
                printf("\n");
        }
        printf("\n");
}
return 0;
}
```

Output:

3D Array Elements:

1 2 3

4 5 6

7 8 9

11 12 13

14 15 16

17 18 19

21 22 23

24 25 26

27 28 29

1.2.8 Function Associated with Arrays

Array Functions in C is a type of data structure that holds multiple elements of the same data type. The size of an array is fixed, and the elements are collected in a sequential manner. There can be different dimensions of arrays and C programming does not limit the number of dimensions in an Array. Traversing, searching, insertion, deletion, sorting are different functions associated with array.

Traversing an Array means going through each element of an Array exactly once. We start from the first element and go to the last element. An example of such a program that performs traversing operation on a linear Array is given below in C language.

Notes

In an Array, the search operation is used to find a particular data item or element in. We can perform searching in an unsorted array with the help of traversal of the Array. The linear traversal from the first element to the last element can be used to search if a given number is present in an Array and can also be used to find its position if present.

Sorting operation is performed to sort an Array into either ascending or descending.

Example:

```
#include <stdio.h>
#define MAX_SIZE 100    // Maximum array size
int main()
{
   int arr[MAX_SIZE];
   int size;
   int i, j, temp;

   /* Input size of array */
   printf("Enter size of array: ");
   scanf("%d", &size);
   /* Input elements in array */
   printf("Enter elements in array: ");
   for(i=0; i<size; i++)
   {
      scanf("%d", &arr[i]);
   }
   for(i=0; i<size; i++)
   {
      /*  Place currently selected element array[i]   */
      for(j=i+1; j<size; j++)
      {
          /*  Swap if currently selected array element is not at its correct position.
*/
         if(arr[i] > arr[j])
         {
            temp     = arr[i];
```

Notes

```
            arr[i] = arr[j];
            arr[j] = temp;
        }
      }
    }

    /* Print the sorted array */
    printf("\nElements of array in ascending order: ");
    for(i=0; i<size; i++)
    {
        printf("%d\t", arr[i]);
    }
    return 0;
}
```

Output:

```
Enter size of array: 5
Enter elements in array: 20 10 30 50 70
Elements of array in ascending order: 10    20    30    50    70
```

Unit-1.3: Strings

Notes

1.3.1 Character String in C

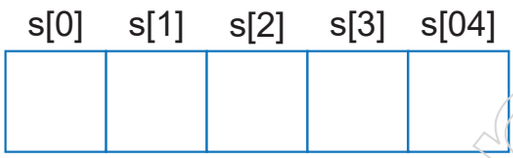
In C programming, a string is a sequence of characters terminated with a null character \0.

For example: char c [] = “c string”;

When the compiler encounters a sequence of characters enclosed in the double quotation marks, it appends a null character \0 at the end by default.

Here is how you can declare strings:

char s[5];



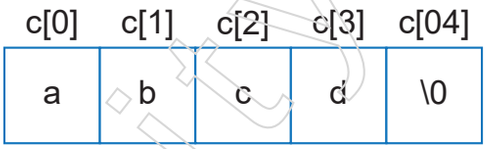
You can initialize strings in several ways.

char c[] = “abcd”;

char c[50] = “abcd”;

char c[] = {‘a’, ‘b’, ‘c’, ‘d’, ‘\0’};

char c[5] = {‘a’, ‘b’, ‘c’, ‘d’, ‘\0’};



Arrays and strings are second-class citizens in C; they do not support the assignment operator once it is declared.

For example: char c[100];

c = “C programming”; // Error! array type is not assignable.

1.3.2 Character String Operations

Different library functions are available for String manipulation as described below:

- **strcpy :** strcpy copies a string, including the null character terminator from the source string to the destination. This function returns a pointer to the destination string.
- **strcat :** This function appends a source string to the end of a destination string. This function returns a pointer to the destination string, or NULL pointer on error.

Notes

- **strcmp :** Two strings are compred with this function. If the first string is greater than the second, it returns a number greater than zero. If the second string is greater, it returns a number less than zero. If the strings are equal, it returns 0.
- **strlen :** This function returns the length of a string, not counting the null character at the end. That is, it returns the character count of the string, without the terminator.

We can use the scanf() function to read a string.

The scanf() function reads the sequence of characters until it encounters whitespace (space, newline, tab, etc.).

Example: scanf() to read a string

```
#include <stdio.h>
int main()
{
   char name[20];
   printf("Enter name: ");
   scanf("%s", name);
   printf("Your name is %s.", name);
   return 0;
}
```

Output:

```
Enter name: Avinab
Your name is Avinab.
```

How to read a line of text?

You can use the fgets() function to read a line of string. You can use puts() to display the string. The puts() function is used to write a line or string to the output stream. It prints the passed string with a newline and returns an integer value.

Example of fgets():

```
#include <stdio.h>
#define MAX 15
int main()
{
   char buf[MAX];
   printf("Enter a string: ");
   fgets(buf, MAX, stdin);
```

Notes

```
    printf("The string is: %s\n", buf);
     return 0;
}
```

Output:

```
Enter a string: Learning C
The string is: Learning C
```

Example of puts():

```
#include<stdio.h>
int main()
{
    //string initialization
    char str1[10] = "C";
    char str2[10] = "Language";
     puts(str1);
    puts(str2); //not specifically adding a newline
     return 0;
}
```

Output:

```
C
Language
```

1.3.3 Arrays as Parameters

If you want to pass a single-dimension array as an argument in a function, you will have to declare a formal parameter in one of following three ways and all three declaration methods produce similar results because each tells the compiler that an integer pointer is going to be received. Similarly, you can pass multi-dimensional arrays as formal parameters.

Example: Passing one dimensional arrays to functions

// Program to calculate the sum of array elements by passing to a function

Example:

```
#include <stdio.h>
float calculateSum(float age[]);
int main() {
    float result, age[] = {23.4, 55, 22.6, 3, 40.5, 18};
```

Notes

```
    // age array is passed to calculateSum()
    result = calculateSum(age);
    printf("Result = %.2f", result);
    return 0;
}
float calculateSum(float age[]) {
  float sum = 0.0;
  int i;
  for( i = 0; i < 6; ++i) {
                sum += age[i];
  }
  return sum;
}
```

Output:

Result = 162.50

Example : Passing two-dimensional arrays

```
#include <stdio.h>
void displayNumbers(int num[2][2]);
int main()
{
    int num[2][2], a, b;
    printf("Enter 4 numbers:\n");
    for ( a = 0; a < 2; ++a)
        for ( b = 0; b < 2; ++b)
            scanf("%d", &num[a][b]);
    // passing multi-dimensional array to a function
    displayNumbers(num);
    return 0;
}
void displayNumbers(int num[2][2])
{
     int i, j;
```

Notes

```
    printf("Numbers you have entered:\n");
    for (i = 0; i < 2; ++i) {
       for (j = 0; j < 2; ++j) {
          printf("%d\t", num[i][j]);
       }
      printf("\n");
     }
}
```

Output:

Enter 4 numbers:

2 3 4 5

Numbers you have entered:

2 3

4 5

Notes

Unit-1.4: Array Implementation

1.4.1 Implementing One Dimensional Array

To implement an array, first we need to declare the data type and then declare array name and mention size of array.

Syntax: dataType arrayName[arraySize];

For example: float mark [5];

Here, we declared an array (name of array is mark) of floating-point type and size is 5 i.e., it can hold 5 floating-point values. It is important to note that the size and type of an array cannot be changed once it is declared.

A variable can be initialized in its declaration.

For example: int value = 25;

The value 25 is called an initializer. Similarly, an array can be initialized in its declaration. A list of initial values for array elements can be specified. They are separated with commas and enclosed within braces.

For example: int age [5] = {23, 56, 87, 92, 38};

In this declaration, age [0] is initialized to 23, age [1] is initialized to 56, and so on. There must be at least one initial value between braces. If too many initial values are specified, a syntax error will occur. If the number of initial values is less than the array size, the remaining array elements will be initialized to zero.

1.4.2 Sparse Matrix

A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. If most of the elements of the matrix have 0 value, then it is called a sparse matrix.

- **Storage:** There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.
- **Computing time:** Computing time can be saved by logically designing a data structure traversing only non-zero elements.

For example:

0 0 3 0 4

0 0 5 7 0

0 0 0 0 0

0 2 6 0 0

Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases. So, instead of storing zeroes with non-zero elements, we only store non-zero elements. This means storing non-zero elements with triples- (Row, Column, value).

Notes

Sparse Matrix Representations can be done in many ways following are two common representations:

- Array representation
- Linked list representation

Implementation of Array Representation of Sparse Matrix

```
/* C Program to check Matrix is a Sparse Matrix or Not */
#include<stdio.h>
 int main()
{
    int i, j, rows, columns, a[10][10], Total = 0;
    printf("\n Enter Number of rows and columns  :  ");
    scanf("%d %d", &i, &j);
    printf("\n Enter the Matrix Elements:  \n");
    for(rows = 0; rows < i; rows++)
    {
            for(columns = 0;columns < j;columns++)
    {
                    scanf("%d", &a[rows][columns]);
    }       }
            for(rows = 0; rows < i; rows++)
    {
            for(columns = 0; columns < j; columns++)
            {
                    if(a[rows][columns] == 0)
                    {
                            Total++;
                    }
             }
    }
    if(Total > (rows * columns)/2)
    {
            printf("\n The Matrix that you entered is a Sparse Matrix ");
    }
```

Notes

```
        else
        {
        printf("\n The Matrix that you entered is Not a Sparse Matrix ");
        }
        return 0;
    }
```

Output:

Enter Number of rows and columns : 5 4

Enter the Matrix Elements:

0 0 3 0 4

0 0 5 7 0

0 0 0 0 0

0 2 6 0 0

The Matrix that you entered is a Sparse Matrix

Module End Questions

1. How can we describe an array in the best possible way?
   a) The Array shows a hierarchical structure.
   b) Arrays are immutable.
   c) Container that stores the elements of similar types
   d) The Array is not a data structure
2. What is a dynamic array?
   a) A variable size data structure
   b) An array which is created at runtime
   c) The memory to the array is allocated at runtime
   d) An array which is reallocated everytime whenever new elements have to be added
3. Which of the following is the advantage of the array data structure?
   a) Elements of mixed data types can be stored.
   b) Easier to access the elements in an array
   c) Index of the first element starts from 1.
   d) Elements of an array cannot be sorted
4. The number of items used by the dynamic array contents is its _____
   a) Physical size

Notes

b) Capacity

c) Logical size

d) Random size

5. Array is divided into two parts in _____

a) Hashed Array Tree

b) Geometric Array

c) Bounded-size dynamic array

d) Sparse Array

6. In which of the following cases dynamic arrays are not preferred?

a) If the size of the array is unknown

b) If the size of the array changes after few iterations

c) If the memory reallocation takes more time i.e. expensive

d) If the array holds less number of elements

7. Which of the following is not the method to represent Sparse Matrix?

a) Dictionary of Keys

b) Linked List

c) Array

d) Heap

8. What will be the output of the following C code?

```
#include <stdio.h>
void f(int a[][3])
  {
     a[0][1] = 3;
     int i = 0, j = 0;
     for (i = 0; i < 2; i++)
     for (j = 0; j < 3; j++)
     printf("%d", a[i][j]);
  }
  void main()
  {
     int a[2][3] = {0};
     f(a);
  }
```

Notes

a) 0 3 0 0 0 0

b) Junk 3 junk junk junk junk

c) Compile time error

d) All junk values

9. What will strcmp() function do?

a) compares the first n characters of the object

b) compares the string

c) undefined function

d) copies the string

10. Which of the following function is correct that finds the length of a string?

a)
```
int xstrlen(char *s)
{
   int length=0;
   while(*s!='\0')
   {    length++; s++; }
   return (length);
}
```

b)
```
int xstrlen(char s)
{
   int length=0;
   while(*s!='\0')
      length++; s++;
   return (length);
}
```

c)
```
int xstrlen(char *s)
{
   int length=0;
   while(*s!='\0')
      length++;
   return (length);
}
```

d)
```
int xstrlen(char *s)
{
   int length=0;
```

```
    while(*s!='\0')
       s++;
    return (length);
}
```

Answer key:

1. c, 2. a, 3. b, 4. c, 5. c, 6. d, 7. d, 8. a, 9. b, 10. a

Notes

Notes

Module-2: Stacks and Queues

Structure:

Unit-2.1: Stack

Notes

2.1.1 Definition

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).

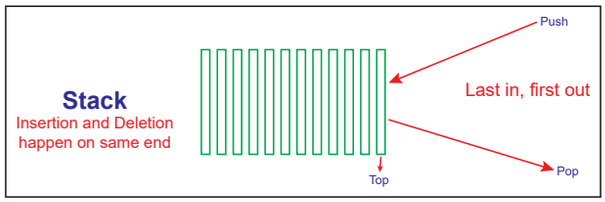


Fig 2.1 Stack

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO (Last in First Out) or FILO (First In Last Out).

Mainly the following three basic operations are performed in the stack:

- Push: Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- Pop: Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- Peek or Top: Returns top element of stack.
- isEmpty: Returns true if stack is empty, else false.

2.1.2 Array Representation of Stack

In array implementation, the stack is formed by using the array. All the operations regarding the stack are performed using arrays. Let us see how each operation can be implemented on the stack using array data structure.

Adding an element into the top of the stack is referred to as push operation. Push operation involves following two steps.

1. Increment the variable Top so that it can now refer to the next memory location.
2. Add element at the position of incremented top. This is referred to as adding new element at the top of the stack.

Stack is overflown when we try to insert an element into a filled stack therefore, our main function must always avoid stack overflow condition.

We are given a stack of elements: 12 , 08 , 21 , 33 , 18 , 40.

Notes

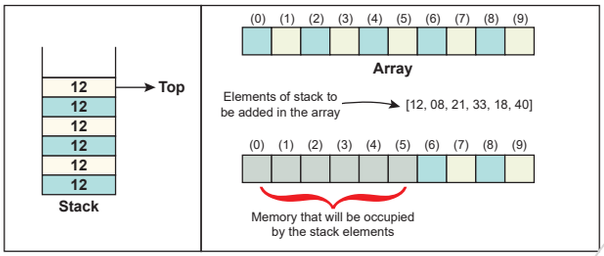


Fig 2.2 Sack overflow

2.1.3 Operations Associated with Stacks- Push & Pop

A stack is a simple last-in, first-out (LIFO) data structure. That is, the last data element stored in a stack is the first data element retrieved from the stack. The common analogy is a stack of plates in a cafeteria: when you go through the line, you pop the top plate off the stack; the dish washer (stepping away from reality a bit), pushes a single clean plate on top of the stack. So a stack supports two basic operations: push and pop. Some stacks also provide additional operations: size (the number of data elements currently on the stack) and peek (look at the top element without removing it).

Based on these operations, the snapshots shown in Figure 3 illustrate the appearance of a stack as data (characters) are stored in or pushed on to it.

Operation	Picture	Execution
Stack is empty	3 2 1 0 ← sp = 0 st	
push('A');	3 2 1 0 A ← sp = 1 st	st[0] = 'A'; sp = 0 + 1;
push('B');	3 2 1 B 0 A ← sp = 2 st	st[1] = 'B'; sp = 1 + 1;
push('C');	3 2 C 1 B 0 A ← sp = 3 st	st[2] = 'C'; sp = 2 + 1;

Notes

The push operation illustrated. Each call to the push function (left column) pushes a data element on to the stack. The main instruction in the push function is st [sp++] = data, where "data" is the function argument. The middle column abstractly illustrates how the stack (the array and the stack pointer) appear after each call to the push function. The right column breaks the behaviour of the push function in to two steps.

Similarly, the data (characters) stored in a stack can be retrieved from or popped off it.

Operation	Picture	Execution
data = pop();	3 2 C ← sp = 2 1 B 0 A st	sp = 3 - 1; return sp[2];
data = pop();	3 2 C 1 B ← sp = 1 0 A st	sp = 2 - 1; return sp[2];
data = pop();	3 2 C 1 B 0 A ← sp = 0 st	sp = 1 - 1; return sp[2];

2.1.4 Polish Expressions

The name comes from the Polish mathematician/logician Lukasiewicz, who introduced it. There are 3 different ways to write an algebraic expression:

- Infix form
- Prefix form
- Postfix form

2.1.5 Conversion of Infix to Postfix

To covert from infix to postfix, we need to scan through an expression and need to get one token at a time.

Step1) Fix a priority level for each operator.

For example, from high to low:

3. - (unary negation)

2. * /

Notes

1. + - (subtraction)

Thus, high priority corresponds to high number in the table.

Step 2) If the token is an operand, do not stack it. Pass it to the output.

Step 3) If token is an operator or parenthesis, do the following:

-- Pop the stack until you find a symbol of lower priority number than the current one. An incoming left parenthesis will be considered to have higher priority than any other symbol. A left parenthesis on the stack will not be removed uniess an incoming right parenthesis is found.

The popped stack elements will be written to output.

--Stack the current symbol.

-- If a right parenthesis is the current symbol, pop the stack down to (and including) the first left parenthesis. Write all the symbols except the left parenthesis to the output (i.e. write the operators to the output).

-- After the last token is read, pop the remainder of the stack and write any symbol (except left parenthesis) to output.

Convert A * (B + C) * D to postfix notation.

Move	Current Token	Stack	Output
1	A	empty	A
2	*	*	A
3	(	(*	A
4	B	(*	A B
5	+	+(*	A B
6	C	+(*	A B C
7	)	*	A B C +
8	*	*	A B C + *
9	D	*	A B C + * D
10		empty	

2.1.6 Infix to Prefix (and vice versa),

Step 1. Push “)” onto STACK, and add “(“ to end of the A

Step 2. Scan A from right to left and repeat step 3 to 6 for each element of A until the STACK is empty.

Step 3. If an operand is encountered add it to B

Step 4. If a right parenthesis is encountered push it onto STACK

Step 5. If an operator is encountered, then:

Notes

a. Repeatedly pop from STACK and add to B, each operator (on the top of STACK) which has same or higher precedence than the operator.

b. Add operator to STACK.

Step 6. If left parenthesis is encountered then

a. Repeatedly pop from the STACK and add to B (each operator on top of stack until a left parenthesis is encountered)

b. Remove the left parenthesis.

Step 7. Exit

Expression = (A+B^C)*D+E^5

Step 1. Reverse the infix expression.

5^E+D*)C^B+A(

Step 2. Make Every '(' as ')' and every ')' as '('

5^E+D*(C^B+A)

Step 3. Convert expression to postfix form.

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

Expression	Stack	Output	Comment
5^E+D*(C^B+A)	Empty	-	Initial
^E+D*(C^B+A)	Empty	5	Print
E+D*(C^B+A)	^	5	Push
+D*(C^B+A)	^	5E	Push
D*(C^B+A)	+	5E^	Pop and Push
*(C^B+A)	+	5E^D	Print
(C^B+A)	+*	5E^D	Push
C^B+A)	+*(	5E^D	Push
^B+A)	+*(	5E^DC	Print
B+A)	+*(^	5E^DC	Push
+A)	+*(^	5E^DCB	Print
A)	+*(+	5E^DCB^	Pop And Push
)	+*(+	5E^DCB^A	Print
End	+*	5E^DCB^A+	Pop Until '('
End	Empty	5E^DCB^A+*+	Pop Every element

Notes

2.1.7 Application of Stacks Recursion

"Recursion" is technique of solving any problem by calling same function again and again until some breaking (base) condition where recursion stops, and it starts calculating the solution from there on. By using stack, many programming languages implement recursion. When a function calls another function, the value of all variables need to be saved, hence, when the called function returns, the values are exactly as they were left. We know when a recursive function calls itself, it is a series of nested functions. It may not have any local variables, but always have a return address and the stack facilitates this. To store and restore the recursive function and arguments, stack is used.

For example calculating factorial of a given number.

```
int fact(int n){
   if(n == 0){
      return 1;
   }
   return (n * fact(n-1)); //function fact() is calling itself
}
```

2.1.8 Polish Expression and Their Compilation

Notation is the way to write arithmetic expression. The name comes from the Polish mathematician/logician Lukasiewicz, who introduced it. There are 3 different ways to write an algebraic expressions:

- Infix form
- Prefix form
- Postfix form

Infix form	Prefix form	Postfix form
a × b	× ab	ab ×
a + b × c	+a × bc	abc × +
(a + b) × c	× + abc	ab + c ×

Fig 2.3 Infix, Prefix and Postfix

Infix form

In infix form, the operators are used in between operands. Though it is easy for human to read and understand this form easily but the same is not true for computing device.

For examples

(3 * 7)

Notes

((1 + 3) * 2)

((1 + 3) * (2 - 3))

Main Feature: Between two operands lies the binary operator.

Prefix form (Polish Notation)

In prefix form, the operator precedes the two operands which means operator is written before operands. Prefix notation is called as Polish notation. For better understanding, just think of an infix notation, (a+b) and if the same expression we want to write in prefix notation then it would be like +ab.

More examples of prefix form,

(* 3 7) or simply * 3 7

(* (+ 1 3) 2) or simply * + 1 3 2

(* (+ 1 3) (- 2 3)) or simply * + 1 3 - 2 3

Postfix form (Reversed Polish Notation)

In postfix form, as the name suggests, the operator is post fixed or lies after the two operands. For better understanding, just think of an infix notation, (a+b) and if the same expression we want to write in postfix form, then it would be like ab+

More examples:

(3 7 *) or simply 3 7 *

((1 3 +) 2 *) or simply 1 3 + 2 *

((1 3 +) (2 3 -) *) or simply 1 3 + 2 3 - *

2.1.9 Conversion of Infix Expression to Prefix and Postfix Expression

Algorithm to convert infix to prefix expression:

In this case, we have operator's stack, output's stack and one input string. Operator's stack works as FILO (First In Last Out). Output's stack works as FIFO (First In First Out).

Steps of conversion of infix expression to prefix

- Reverse the input string.
- Make every '(' as ')' and every ')' as '('
- Convert expression to postfix form.
- Reverse the expression.

Infix expression: (P + (Q*R) / (S-T))

Notes

Symbol Scanned	Stack	Output
)	)	-
)	))	-
T	))	T
-	))-	T
S	))-	ST
(	)	-ST
/	)/	-ST
)	)/)	-ST
R	)/)	R-ST
*	)/)*	R-ST
Q	)/)*	QR-ST
(	)/	*QR-ST
+	)+	/*QR-ST
P	)+	P/*QR-ST
(	Empty	+P/*QR-ST

Prefix expression: +P/*QR-ST

Algorithm to convert infix to postfix expression:

In this case, we use the stacks to convert infix to postfix. We have operator's stack, output's stack and one input string. Operator's stack works as FILO (First In Last Out). Output's stack works as FIFO (First In First Out).

Steps of conversion of infix expression to postfix

- Scan input string from left to right character by character.
- If the character is an operand, put it into output stack.
- If the character is an operator and operator's stack is empty, push operator into operators' stack.
- If the operator's stack is not empty, there may be following possibilities.
- If the precedence of scanned operator is greater than the topmost operator of operator's stack, push this operator into operand's stack.
- If the precedence of scanned operator is less than or equal to the topmost operator of operator's stack, pop the operators from operand's stack until we find a low precedence operator than the scanned character. Never pop out ('(') or (')') whatever may be the precedence level of scanned character.
- If the character is opening round bracket ('('), push it into operator's stack.
- If the character is closing round bracket (')'), pop out operators from operator's stack until we find an opening bracket ('(').

Notes

- Now pop out all the remaining operators from the operator's stack and push into output stack.

Infix expression: (A/(B-C)*D+E)

Symbol Scanned	Stack	Output
(	(	-
A	(	A
/	(/	A
(	(/(	A
B	(/(	AB
-	(/(-	AB
C	(/(-	ABC
)	(/	ABC-
*	(*	ABC-/
D	(*	ABC-/D
+	(+	ABC-/D*
E	(+	ABC-/D*E
)	Empty	ABC-/D*E+

Postfix expression: ABC-/D*E+

2.1.10 Tower of Hanoi Problem

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e., a disk can only be moved if it is the uppermost disk on a stack.
3. No disk may be placed on top of a smaller disk.

Take an example for 2 disks:

Let rod 1 = 'A', rod 2 = 'B', rod 3 = 'C'.

Step 1: Shift first disk from 'A' to 'B'.

Step 2: Shift second disk from 'A' to 'C'.

Step 3: Shift first disk from 'B' to 'C'.

The pattern here is:

Shift 'n-1' disks from 'A' to 'B'.

Notes

Shift last disk from ‘A’ to ‘C’.

Shift ‘n-1’ disks from ‘B’ to ‘C’.

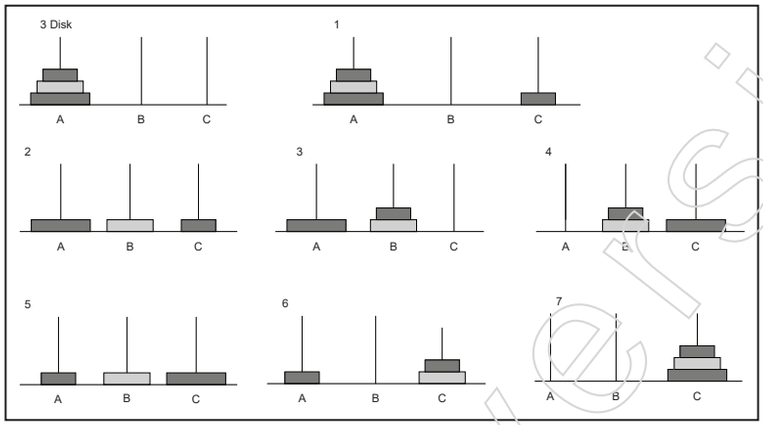


Fig 2.4 Tower of Hanoi

Example:

```
/* C program for Tower of Hanoi using Recursion */
#include <stdio.h>
 void towers(int, char, char, char);
 int main()
{
    int num;
     printf(“Enter the number of disks: “);
    scanf(“%d”, &num);
    printf(“The sequence of moves involved in the Tower of Hanoi are :\n”);
    towers(num, ‘A’, ‘C’, ‘B’);
    return 0;
}
void towers(int num, char frompeg, char topeg, char auxpeg)
{
    if (num == 1)
    {
        printf(“\n Move disk 1 from peg %c to peg %c”, frompeg, topeg);
        return;
    }
    towers(num - 1, frompeg, auxpeg, topeg);
```

```
    printf("\n Move disk %d from peg %c to peg %c", num, frompeg, topeg);
    towers(num - 1, auxpeg, topeg, frompeg);
}
```

Output:

Enter the number of disks: 3

The sequence of moves involved in the Tower of Hanoi are :

Move disk 1 from peg A to peg C

Move disk 2 from peg A to peg B

Move disk 1 from peg C to peg B

Move disk 3 from peg A to peg C

Move disk 1 from peg B to peg A

Move disk 2 from peg B to peg C

Move disk 1 from peg A to peg C

Notes

Notes

Unit-2.2: Queue

2.2.1 Definition

Queue is an abstract data structure, somewhat like Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

A Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO). A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first. The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

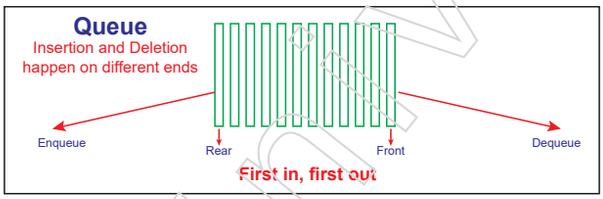


Fig 2.5 Queue

2.2.2 Representation of Queue

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure.

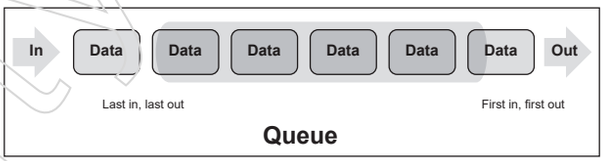


Fig 2.6 LIFO and FIFO

2.2.3 Operations of Queue

Queue operations involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues:

Notes

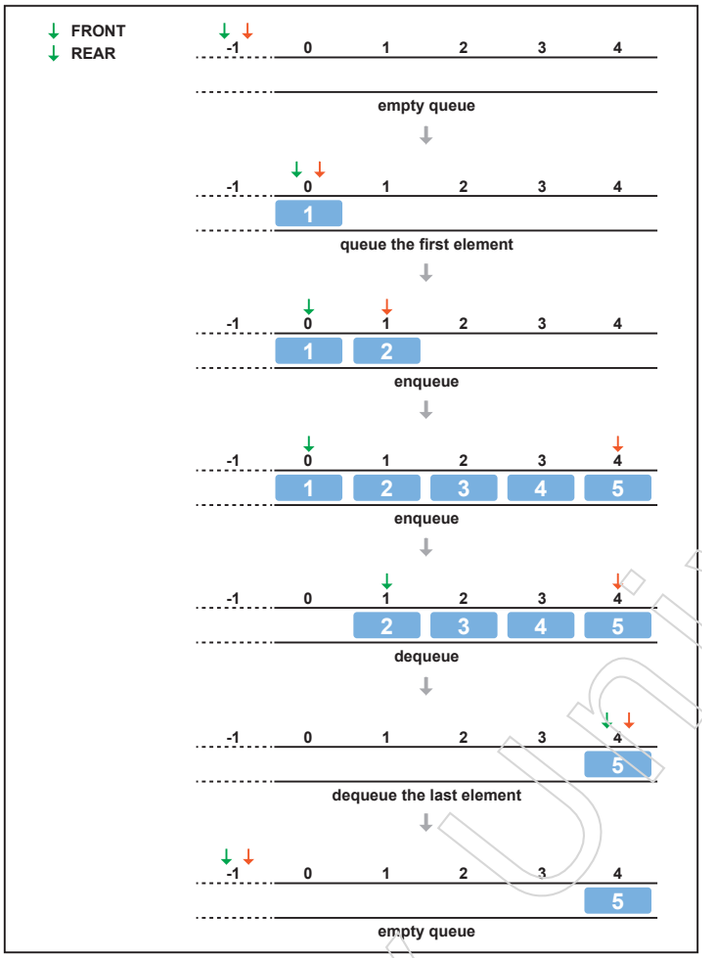


Fig 2.7 Operations on queue

Enqueue Operation

- check if the queue is full
- for the first element, set the value of FRONT to 0
- increase the REAR index by 1
- add the new element in the position pointed to by REAR

Dequeue Operation

- check if the queue is empty
- return the value pointed by FRONT
- increase the FRONT index by 1
- for the last element, reset the values of FRONT and REAR to -1

Few more functions are required to make the above-mentioned queue operation efficient. These are −

- peek() − Gets the element at the front of the queue without removing it.
- isfull() − Checks if the queue is full.

Notes

- isempty() − Checks if the queue is empty.

2.2.4 Priority Queue

Priority Queue is an extension of queue with following properties.

1. Every item has a priority associated with it.
2. An element with high priority is dequeued before an element with low priority.
3. If two elements have the same priority, they are served according to their order in the queue.

A typical priority queue supports following operations.

insert(item, priority): Inserts an item with given priority.

getHighestPriority(): Returns the highest priority item.

deleteHighestPriority(): Removes the highest priority item.

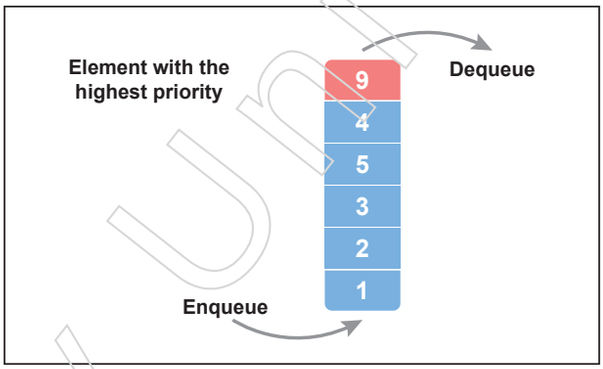


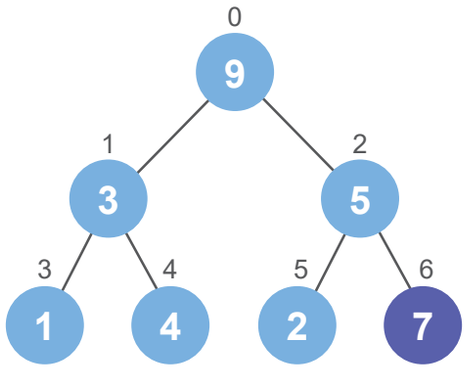
Fig 2.8 Enque and deque

Priority Queue Operations

1. Inserting an Element into the Priority Queue

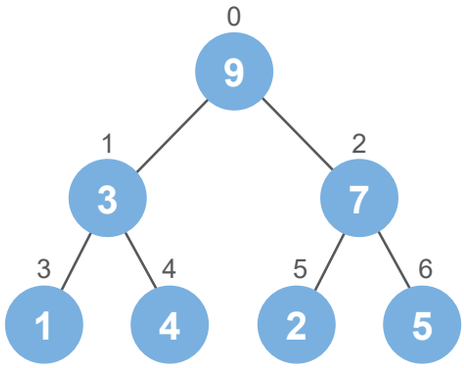
Inserting an element into a priority queue (max-heap) is done by the following steps.

- Insert the new element at the end of the tree.



Heapify the tree.

Notes



Algorithm for insertion of an element into priority queue (max-heap)

If there is no node,

create a newNode.

else (a node is already present)

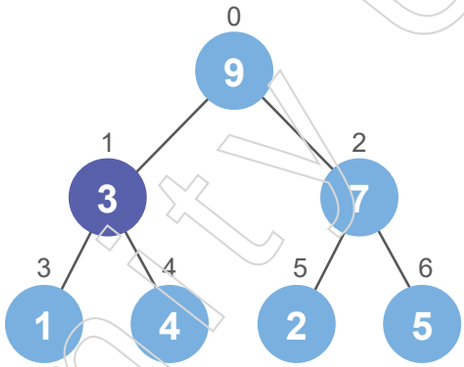
insert the newNode at the end (last node from left to right.)

heapify the array

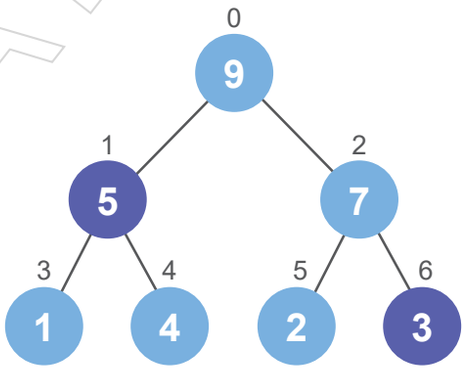
2. Deleting an Element from the Priority Queue

Deleting an element from a priority queue (max-heap) is done as follows:

Select the element to be deleted.

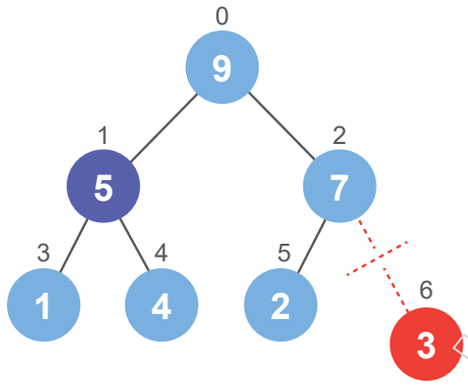


Swap it with the last element.

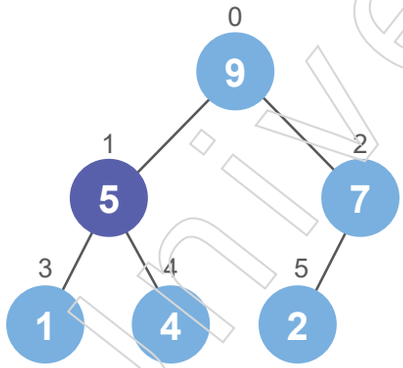


Remove the last element.

Notes



Heapify the tree.



Algorithm for deletion of an element in the priority queue (max-heap)

If nodeToBeDeleted is the leafNode

remove the node

Else swap nodeToBeDeleted with the lastLeafNode

remove noteToBeDeleted

heapify the array

Applications of Priority Queue:

1) CPU Scheduling
2) Graph algorithms like Dijkstra's shortest path algorithm, Prim's Minimum Spanning Tree, etc
3) All queue applications where priority is involved.

```
/*C Program to Implement Priority Queue to Add and Delete Elements */
#include <stdio.h>
#include <stdlib.h>
 #define MAX 5
 void insert_by_priority(int);
void delete_by_priority(int);
void create();
```

Notes

```
void check(int);
void display_pqueue();
 int pri_que[MAX];
int front, rear;
 void main()
{
    int n, ch;
     printf("\n1 - Insert an element into queue");
    printf("\n2 - Delete an element from queue");
    printf("\n3 - Display queue elements");
    printf("\n4 - Exit");
     create();
     while (1)
    {
        printf("\nEnter your choice: ");
        scanf("%d", &ch);
         switch (ch)
        {
        case 1:
            printf("\nEnter value to be inserted: ");
            scanf("%d",&n);
            insert_by_priority(n);
            break;
        case 2:
            printf("\nEnter value to delete: ");
            scanf("%d",&n);
            delete_by_priority(n);
            break;
        case 3:
            display_pqueue();
            break;
        case 4:
```

Notes

```
            exit(0);
        default:
            printf("\nChoice is incorrect, Enter a correct choice:");
        }
    }
}
/* Function to create an empty priority queue */
void create()
{
    front = rear = -1;
}
 /* Function to insert value into priority queue */
void insert_by_priority(int data)
{
    if (rear >= MAX - 1)
    {
        printf("\nQueue overflow no more elements can be inserted");
  return;
    }
    if ((front == -1) && (rear == -1))
    {
        front++;
        rear++;
        pri_que[rear] = data;
        return;
    }
    else
        check(data);
    rear++;
}
 /* Function to check priority and place element */
void check(int data)
```

Notes

```
{
    int i,j;
     for (i = 0; i <= rear; i++)
    {
        if (data >= pri_que[i])
        {
            for (j = rear + 1; j > i; j--)
            {
                pri_que[j] = pri_que[j - 1];
            }
            pri_que[i] = data;
            return;
        }
    }
    pri_que[i] = data;
}
 /* Function to delete an element from queue */
void delete_by_priority(int data)
{
    int i;
     if ((front==-1) && (rear==-1))
    {
        printf("\nQueue is empty no elements to delete");
        return;
    }
     for (i = 0; i <= rear; i++)
    {
        if (data == pri_que[i])
        {
            for (; i < rear; i++)
            {
                pri_que[i] = pri_que[i + 1];
```

Notes

```
            }
         pri_que[i] = -99;
        rear--;
         if (rear == -1)
            front = -1;
        return;
        }
    }
    printf("\n%d Not found in queue to delete", data);
}
 /* Function to display queue elements */
void display_pqueue()
{
    if ((front == -1) && (rear == -1))
    {
        printf("\nQueue is empty");
        return;
    }
     for (; front <= rear; front++)
    {
        printf(" %d ", pri_que[front]);
    }
     front = 0;
}
```

Output:

1 - Insert an element into queue

2 - Delete an element from queue

3 - Display queue elements

4 - Exit

Enter your choice: 1

Enter value to be inserted: 10

Enter your choice: 1

Notes

Enter value to be inserted: 5

Enter your choice: 1

Enter value to be inserted: 25

Enter your choice: 3

25 10 5

Enter your choice: 2

Enter value to delete: 10

Enter your choice: 3

25 5

Enter your choice: 4

2.2.5 Circular Queue

Circular Queue is also a linear data structure, which follows the principle of FIFO(First In First Out), but instead of ending the queue at the last position, it again starts from the first position after the last, hence making the queue behave like a circular data structure.

In case of a circular queue, head pointer will always point to the front of the queue, and tail pointer will always point to the end of the queue.

Initially, the head and the tail pointers will be pointing to the same location, this would mean that the queue is empty.

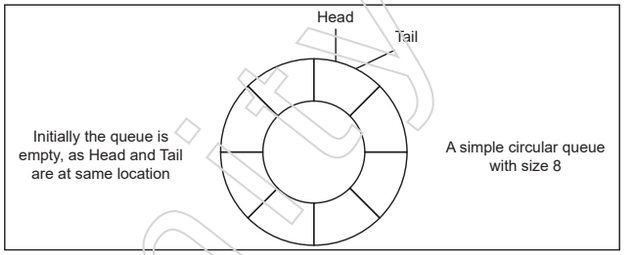


Fig 2.8 Circular queue

New data is always added to the location pointed by the tail pointer, and once the data is added, tail pointer is incremented to point to the next available location.

Circular Queue works by the process of circular increment i.e. when we try to increment the pointer and we reach the end of the queue, we start from the beginning of the queue.

Here, the circular increment is performed by modulo division with the queue size. That is,

if REAR + 1 == 5 (overflow!), REAR = (REAR + 1)%5 = 0 (start of queue)

Notes

2.2.6 Deque

Deque or Double Ended Queue is a generalized version of Queue data structure that allows insert and delete at both ends.

Operations on Deque:

Mainly the following four basic operations are performed on queue:

insertFront(): Adds an item at the front of Deque.

insertLast(): Adds an item at the rear of Deque.

deleteFront(): Deletes an item from front of Deque.

deleteLast(): Deletes an item from rear of Deque.

Applications of Deque:

Since Deque supports both stack and queue operations, it can be used as both. The Deque data structure supports clockwise and anticlockwise rotations in O(1) time which can be useful in certain applications.

```
/*static circular queue*/
#include <stdio.h>
#define size 5
void insertq(int[], int);
void deleteq(int[]);
void display(int[]);
int front =  - 1;
int rear =  - 1;
int main()
{
   int n, ch;
   int queue[size];
   do
   {
printf("\n\n Circular Queue:\n1. Insert \n2. Delete\n3. Display\n0. Exit");
      printf("\nEnter Choice 0-3? : ");
      scanf("%d", &ch);
      switch (ch)
      {
         case 1:
```

Notes

```
            printf("\nEnter number: ");
            scanf("%d", &n);
            insertq(queue, n);
            break;
        case 2:
            deleteq(queue);
            break;
        case 3:
            display(queue);
            break;
      }
   }while (ch != 0);
}
void insertq(int queue[], int item)
{
   if ((front == 0 && rear == size - 1) || (front == rear + 1))
   {
      printf("queue is full");
      return;
   }
   else if (rear ==  - 1)
   {
      rear++;
      front++;
   }
   else if (rear == size - 1 && front > 0)
   {
      rear = 0;
   }
   else
   {
      rear++;
```

Notes

```
    }
    queue[rear] = item;
}
void display(int queue[])
{
    int i;
    printf("\n");
    if (front > rear)
    {
        for (i = front; i < size; i++)
        {
            printf("%d ", queue[i]);
        }
        for (i = 0; i <= rear; i++)
            printf("%d ", queue[i]);
    }
    else
    {
        for (i = front; i <= rear; i++)
            printf("%d ", queue[i]);
    }
}
void deleteq(int queue[])
{
    if (front ==  - 1)
    {
        printf("Queue is empty ");
    }
    else if (front == rear)
    {
        printf("\n %d deleted", queue[front]);
```

Notes

```
        front =  - 1;
        rear =  - 1;
    }
    else
    {
        printf("\n %d deleted", queue[front]);
        front++;
    }
}
```

Module End Questions

1. Which one of the following is the process of inserting an element in the stack?
   a) Insert
   b) Add
   c) Push
   d) None of the above
2. What is the value of the postfix expression 6 3 2 4 + – *?
   a) 1
   b) 40
   c) 74
   d) -18
3. When the user tries to delete the element from the empty stack then the condition is said to be a ____
   a) Underflow
   b) Garbage collection
   c) Overflow
   d) None of the above
4. Entries in a stack are "ordered". What is the meaning of this statement?
   a) A collection of stacks is sortable
   b) Stack entries may be compared with the '<' operation
   c) The entries are stored in a linked list
   d) There is a Sequential entry that is one by one
5. If the size of the stack is 10 and we try to add the 11th element in the stack then the condition is known as___
   a) Underflow

b) Garbage collection

c) Overflow

d) None of the above

6. Which one of the following is not the application of the stack data structure

a) String reversal

b) Recursion

c) Backtracking

d) Asynchronous data transfer

7. Which data structure is mainly used for implementing the recursive algorithm?

a) Queue

b) Stack

c) Binary tree

d) Linked list

8. Which data structure is required to convert the infix to prefix notation?

a) Stack

b) Linked list

c) Binary tree

d) Queue

9. Which of the following is the prefix form of A+B*C?

a) A+(BC*)

b) +AB*C

c) ABC+*

d) +A*BC

10. Which of the following is not the correct statement for a stack data structure?

a) Arrays can be used to implement the stack

b) Stack follows FIFO

c) Elements are stored in a sequential manner

d) Top of the stack contains the last inserted element

Answer key:

1. c, 2. d, 3. a, 4. d, 5. c, 6. d, 7. b, 8. a, 9. d, 10. b

Module-3: Programming with Linked Lists

Notes

Structure:

Notes

Unit-3.1: Singly linked lists

3.1.1 Introduction and Representation of linked lists in Memory

Linked list is a linear data structure. Linked list is defined as collection of objects called nodes and nodes are stored randomly in memory. A node contains two fields, one field contains data and the other field contains address of the node. The last node points to NULL. Unlike array here the list is not contiguous in memory. Empty node cannot be present here.

Singly Linked is collection of ordered set of nodes. According to the need of the program, the number of nodes varies. Linked list components are not stored at a contiguous location. The elements are linked using pointers. Singly link list can traverse only in one direction.

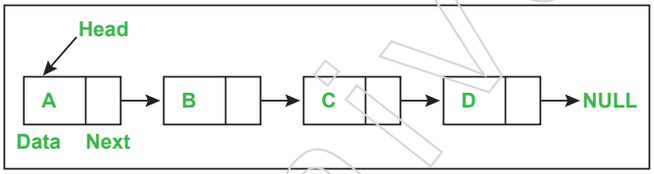


Fig 3.1 Singly link list

In the above figure, the arrow represents the link and the data part contains values. A linked list is characterized by a pointer to the first node of the linked list. In the figure, next defines the address of the next node. The first node is termed the head. If the linked list is empty, then the value of head is NULL.

The list is not necessary to be contiguously present in the memory. The node can reside anyplace in the memory and linked together to make a list. This achieves optimized utilization of space. List size is limited to the memory size and does not require to be declared in advance. Empty node cannot be appear in the linked list. We can store values of primitive types or objects in the singly linked list. Data part of the node stores genuine information that is to be epitomized by the node while the link part of the node stores the address of its current successor.

One way chain or singly linked list can be traversed only in one direction. In other words, we can say that each node includes only next pointer, therefore we cannot traverse the list in the reverse direction. Let us take one example for better understanding. In below figure we can see a singly linked list which shows the marks obtained by the student in three subjects.

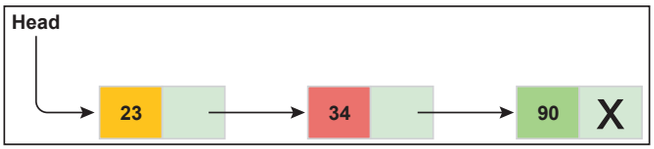


Fig 3.2 Implementation of Singly link list

In the above figure, the arrow symbolizes the links which hold the address of next node. The last node in the list is pinpointed by the null pointer which is present in the address part of the last node. We can have as many elements as possible we require, in the data part of the list.

3.1.2 Traversing, Searching, Insertion, Deletion from linked list

Notes

Operations on Singly Linked List:

Node Creation:

```
struct node
{
   int data;
   struct node *next;
};
struct node *head, *ptr;
ptr = (struct node *)malloc(sizeof(struct node *));
```

Insertion

The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following categories.

Insertion at beginning

By this process we can insert any element at the beginning of the list. For doing this, we need to make the new node as the head of the list.

Steps:

Allocate memory for new node : newNode = malloc(sizeof(struct node));

Store data: newNode->data = 4;

Change next of new node to point to head: newNode->next = head;

Change head to point to recently created node: head = newNode;

Example:

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
newNode->next = head;
head = newNode;
```

Insertion at end of the list

By this process, we can insert a node at the end of the list. The new node has to be added after the last node. Let us think of a link list like 10->20->30->40 and now we want to add a new node 50 at the end of the list, so the list will become 10->20->30->40->50. For doing this, we have to traverse the list till the end and we need to make change of next of last node to new node.

Notes

Steps:

- Allocate memory for new node
- Store data
- Traverse to last node
- Change next of last node to recently created node

Example:

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
newNode->next = NULL;
struct node *temp = head;
while(temp->next != NULL){
  temp = temp->next;
}
temp->next = newNode;
```

This process contains insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics we have to implemented for each scenario.

Insertion after a specified node

By this process, we can insert a node after a specific node in the list. We already have pointer to a node, we need to make adjustment of pointers for this.

Steps:

- Allocate memory and store data for new node
- Traverse to node just before the required position of new node
- Change next pointers to include new node in between

Example:

```
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
struct node *temp = head;
for(int i=2; i < position; i++) {
   if(temp->next != NULL) {
      temp = temp->next;
   } }
```

Notes

```
newNode->next = temp->next;

temp->next = newNode;
```

Traversing

Traversing is the most widespread operation that is performed in almost every situation of singly linked list. Traversing requires visiting each node of the list once to carry out some operation on that. This will be done by using the subsequent statements.

Example:

```
ptr = head;
      while (ptr!=NULL)
        {
           ptr = ptr -> next;
        }
```

Algorithm

- STEP 1: SET PTR = HEAD
- STEP 2: IF PTR = NULL

WRITE “EMPTY LIST”

GOTO STEP 7

END OF IF

- STEP 4: REPEAT STEP 5 AND 6 UNTIL PTR != NULL
- STEP 5: PRINT PTR→ DATA
- STEP 6: PTR = PTR → NEXT

[END OF LOOP]

- STEP 7: EXIT

Example:

```
#include<stdio.h>
#include<stdlib.h>
void create(int);
void traverse();
struct node
{
   int data;
   struct node *next;
};
struct node *head;
```

Notes

```
void main ()
{
   int choice,item;
   do
   {
      printf("\n1.Append List\n2.Traverse\n3.Exit\n4.Enter your choice\n");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:
         printf("\nEnter the item\n");
         scanf("%d",&item);
         create(item);
         break;
         case 2:
         traverse();
         break;
         case 3:
         exit(0);
         break;
         default:
         printf("\nPlease enter valid choice\n");
      }
   }while(choice != 3);
}
void create(int item)
   {
      struct node *ptr = (struct node *)malloc(sizeof(struct node *));
      if(ptr == NULL)
      {
         printf("\nOVERFLOW\n");
      }
```

Notes

```
        else
        {
            ptr->data = item;
            ptr->next = head;
            head = ptr;
            printf("\nNode inserted\n");
        }
    }
void traverse()
  {
        struct node *ptr;
        ptr = head;
        if(ptr == NULL)
        {
            printf("Empty list..");
        }
        else
        {
            printf("The list is..\n");
            while (ptr!=NULL)
            {
                printf("\n%d",ptr->data);
                ptr = ptr -> next;
            }
        }
    }
```

Output:

1. Append List
2. Traverse
3. Exit
4. Enter your choice

1

Enter the item

Notes

100

Node inserted

1. Append List

2. Traverse

3. Exit

4. Enter your choice

1

Enter the item

200

Node inserted

1. Append List

2. Traverse

3. Exit

4. Enter your choice

1

Enter the item

300

Node inserted

1. Append List

2. Traverse

3. Exit

4. Enter your choice

2

The list is..

300

200

100

1. Append List

2. Traverse

3. Exit

4. Enter your choice

3

Deletion

Notes

Like insertion, the Deletion of a node from a singly linked list can be achieved at different positions. Based on the position of the node being deleted, the operation is classified into the following categories.

Deletion at beginning

By this process, deletion of a node from the beginning of the list can be done. This is the easiest operation among all. It just requires a few adjustments in the node pointers.

- Point head to the second node

```
head = head->next;
```

Deletion at the end of the list

By this process we can delete the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios.

Steps:

- Traverse to second last element
- Change its next pointer to null

```
struct node* temp = head;
while(temp->next->next!=NULL){
  temp = temp->next;
}
temp->next = NULL;
```

Deletion after specified node

By this process we can delete a node after a specified node in the list. we require to skip the desired number of nodes to reach the node after which the node will be deleted. For doing so we need to traverse through the list.

Steps:

- Traverse to element before the element to be deleted
- Change next pointers to exclude the node from the chain

```
for(int i=2; i< position; i++) {
   if(temp->next!=NULL) {
      temp = temp->next;
   }
}
temp->next = temp->next->next;
```

Searching

While searching, we need to match each element of the list with the given element. If the element is discovered on any of the location then location of that element is

Notes

returned else null is returned.

Example:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{     int data;
    struct node *next;
};
struct node *head;
void begininsert ();
void lastinsert ();
void otherinsert();
void begin_delete();
void last_delete();
void other_delete();
void display();
void search();
void main ()
{   int choice =0;
   while(choice != 9)
   {
printf("\nChoose one option from the following list\n");
printf("\n1.Insert at beginning \n2.Insert at last\n3.Insert at any other location\
n4.Delete node from beginning\n5.Delete node from last\n6.Delete node after specified
location\n7.Search for an element\n8.Display list\n9.Exit\n");
printf("\nEnter your choice: ");
scanf("\n%d",&choice);
switch(choice)
      {
         case 1:
         begininsert();
         break;
         case 2:
```

Notes

```
            lastinsert();
            break;
            case 3:
            otherinsert();
            break;
            case 4:
            begin_delete();
            break;
            case 5:
            last_delete();
            break;
            case 6:
            other_delete();
            break;
            case 7:
            search();
            break;
            case 8:
            display();
            break;
            case 9:
            exit(0);
            break;
            default:
            printf("Please enter valid choice..");
        }     } }
    void begininsert()
    {
    struct node *ptr;
       int item;
       ptr = (struct node *) malloc(sizeof(struct node *));
       if(ptr == NULL)
```

Notes

```
    {       printf("\nOVERFLOW");
    }
    else
    {   printf("\nEnter value\n");
        scanf("%d",&item);
        ptr->data = item;
        ptr->next = head;
        head = ptr;
        printf("\nNode inserted");
    } }
void lastinsert()
{
    struct node *ptr,*temp;
    int item;
    ptr = (struct node*)malloc(sizeof(struct node));
    if(ptr == NULL)
    {       printf("\nOVERFLOW");
    }
    else
    {   printf("\nEnter value: \n");
        scanf("%d",&item);
        ptr->data = item;
        if(head == NULL)
        {  ptr -> next = NULL;
           head = ptr;
           printf("\nNode inserted");
        }
        else
        {
  temp = head;
  while (temp -> next != NULL)
           {
```

Notes

```
        temp = temp -> next;
            }
            temp->next = ptr;
            ptr->next = NULL;
            printf("\nNode inserted");
             }     } }
void otherinsert()
{
 int i,loc,item;
    struct node *ptr, *temp;
    ptr = (struct node *) malloc (sizeof(struct node));
    if(ptr == NULL)
    {         printf("\nOVERFLOW");
    }
    else
    {
      printf("\nEnter element value");
        scanf("%d",&item);
        ptr->data = item;
        printf("\nEnter the location after which you want to insert ");
        scanf("\n%d",&loc);
        temp=head;
        for(i=0;i<loc;i++)
        {
       temp = temp->next;
            if(temp == NULL)
            {     printf("\ncan't insert\n");
                return;
            }       }
        ptr ->next = temp ->next;
        temp ->next = ptr;
        printf("\nNode inserted");
```

Notes

```
    }
}
void begin_delete()
{
    struct node *ptr;
    if(head == NULL)
    {
        printf("\nList is empty.\n");
    }
    else
    {
        ptr = head;
        head = ptr->next;
        free(ptr);
        printf("\nNode deleted from the beginning.\n");
    }
}
void last_delete()
{
    struct node *ptr,*ptr1;
    if(head == NULL)
    {
        printf("\nlist is empty");
    }
    else if(head -> next == NULL)
    {
        head = NULL;
        free(head);
        printf("\nOnly node of the list deleted\n");
    }
   else
    {
```

Notes

```
        ptr = head;
        while(ptr->next != NULL)
        {
            ptr1 = ptr;
            ptr = ptr ->next;
        }
        ptr1->next = NULL;
        free(ptr);
        printf("\nDeleted Node from the last.\n");
    }   }
void other_delete()
{   struct node *ptr,*ptr1;
    int loc,i;
printf("\n Enter the location of the node after which you want to perform deletion
\n");
    scanf("%d",&loc);
    ptr=head;
    for(i=0;i<loc;i++)
    {
        ptr1 = ptr;
        ptr = ptr->next;
         if(ptr == NULL)
        {
            printf("\nCan't delete");
            return;
        }   }
    ptr1 ->next = ptr ->next;
    free(ptr);
    printf("\nDeleted node %d ",loc+1);
}
void search()
{   struct node *ptr;
```

Notes

```
int item,i=0,flag;
ptr = head;
if(ptr == NULL)
{
   printf("\nEmpty List\n");
}
else
{
   printf("\nEnter item which you want to search?\r");
   scanf("%d",&item);
   while (ptr!=NULL)
   {
      if(ptr->data == item)
      {
         printf("item found at location %d ",i+1);
         flag=0;
      }
      else
      {
         flag=1;
      }
      i++;
      ptr = ptr -> next;
   }
   if(flag==1)
   {
      printf("Item not found\n");
   }    }      }
void display()
{
   struct node *ptr;
```

Notes

```
ptr = head;
if(ptr == NULL)
{
    printf("Nothing to print");
}
else
{
    printf("\nThe list is:\n");
    while (ptr!=NULL)
    {
        printf("\n%d",ptr->data);
        ptr = ptr -> next;
    }    } }
```

After compiling the above program, the below output will be shown

Output:

Choose one option from the following list

1. Insert at beginning
2. Insert at last
3. Insert at any other location
4. Delete node from beginning
5. Delete node from last
6. Delete node after specified location
7. Search for an element
8. Display list
9. Exit

Enter your choice:1

Enter value

10

Node inserted

Choose one option from the following list

1. Insert at beginning
2. Insert at last
3. Insert at any other location

Notes

4. Delete node from beginning
5. Delete node from last
6. Delete node after specified location
7. Search for an element
8. Display list
9. Exit

Enter your choice:1

Enter value

20

Node inserted

Choose one option from the following list

1. Insert at beginning
2. Insert at last
3. Insert at any other location
4. Delete node from beginning
5. Delete node from last
6. Delete node after specified location
7. Search for an element
8. Display list
9. Exit

Enter your choice:1

Enter value

30

Node inserted

Choose one option from the following list

1. Insert at beginning
2. Insert at last
3. Insert at any other location
4. Delete node from beginning
5. Delete node from last
6. Delete node after specified location
7. Search for an element
8. Display list

Notes

9. Exit

Enter your choice:1

Enter value

40

Node inserted

Choose one option from the following list

1. Insert at beginning
2. Insert at last
3. Insert at any other location
4. Delete node from beginning
5. Delete node from last
6. Delete node after specified location
7. Search for an element
8. Display list
9. Exit

Enter your choice:8

The list is:

40

30

20

10

Choose one option from the following list

1. Insert at beginning
2. Insert at last
3. Insert at any other location
4. Delete node from beginning
5. Delete node from last
6. Delete node after specified location
7. Search for an element
8. Display list
9. Exit

Enter your choice:2

Enter value:

60

Notes

Node inserted

Choose one option from the following list

1. Insert at beginning
2. Insert at last
3. Insert at any other location
4. Delete node from beginning
5. Delete node from last
6. Delete node after specified location
7. Search for an element
8. Display list
9. Exit

Enter your choice:8

Enter value

40

30

20

10

50

Choose one option from the following list

1. Insert at beginning
2. Insert at last
3. Insert at any other location
4. Delete node from beginning
5. Delete node from last
6. Delete node after specified location
7. Search for an element
8. Display list
9. Exit

Enter your choice:4

Node deleted from the beginning.

Choose one option from the following list

1. Insert at beginning
2. Insert at last

Notes

3. Insert at any other location
4. Delete node from beginning
5. Delete node from last
6. Delete node after specified location
7. Search for an element
8. Display list
9. Exit

Enter your choice:8

The list is:

30

20

10

50

Choose one option from the following list

1. Insert at beginning
2. Insert at last
3. Insert at any other location
4. Delete node from beginning
5. Delete node from last
6. Delete node after specified location
7. Search for an element
8. Display list
9. Exit

Enter your choice: 9

3.1.3 Garbage Collection and Compaction

Garbage collection is a kind of automatic memory management. It is marking unreachable memory blocks (garbage) as free. Compaction is moving reachable memory blocks close together, so that there are not free memory blocks between them. Some programming languages have built-in garbage collector while some programming languages require functions to manage garbage collection.

When a node is deleted, some memory space becomes reusable. This memory space should be obtainable for future use. One way to do this is to instantly insert the free space into availability list. But this method may be time exhausting for the operating system. So an alternative method is used which is called 'Garbage Collection'. This method is described below: In this method the OS collects the deleted space time to time onto the availability list. This process occurs in two steps. In first step, the OS goes

Notes

through all the lists and tags all those cells which are currently being utilized. In the second step, the OS goes through all the lists again and collects untagged space and adds this collected space to availability list. The garbage collection may happen when small amount of free space is left in the system or no free space is left in the system or when CPU is idle and has time to do the garbage collection.

Garbage collector is the magic component that makes the illusion of infinite memory possible. It typically comprises of two semi-independent components known as mutator and collector.

Mutator

Usually, all components of a language runtime that are accountable for executing application code, allocating/de-allocating new objects, and managing execution contexts get considered as the mutators. Most acceptably large programs have more than one mutator threads. Precise number of mutator threads is generally irrelevant to the garbage collection approaches.

Collector

The collector executes the garbage collection code. It realizes the garbage (read: unreachable) objects and reclaims their storage. In concurrent GCs, there may be more than one collector threads. Exact number of collectors is mostly inappropriate to the approaches that we are going to discuss. Most GCs suggest control over the behaviours of collectors through configuration settings.

Unit-3.2: Doubly linked list

Notes

3.2.1 Introduction to Doubly Linked List

An extra pointer has been added to implement a Doubly Linked List, usually termed as previous pointer, together with next pointer and data which are there in singly linked list.

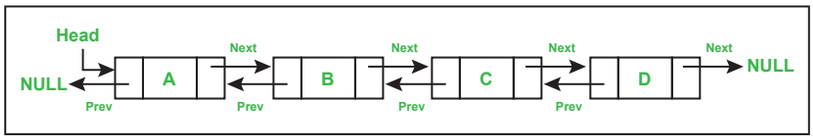


Fig 3.3 Doubly linked list

Below is the process of creating node in Doubly Link List.

```
struct Node {
   int data;
   struct Node* next; // Pointer to next node in Doubly Link List
   struct Node* prev; // Pointer to previous node in Doubly Link List
};
```

Advantages over singly linked list:

1. The main advantage is we can traverse in both forward and in backward direction.
2. The delete operation in Doubly Link List is more effective if pointer to the node to be deleted is given.
3. We can swiftly insert a new node before a given node.

In singly linked list, to delete a node, pointer to the previous node is needed. To get this previous node, the list has to be traversed. In Doubly Link List, we can get the earlier node using previous pointer itself.

Disadvantages over singly linked list

1. The main disadvantage is it require extra space to store previous pointer.
2. All operations involve an additional pointer previous to be maintained. For example, in insertion, we ought to modify previous pointers together with next pointers. For example in following functions for insertions at various positions, we need 1 or 2 extra steps to set previous pointer.

Operations on doubly linked list

The operations of doubly link list includes insertion, deletion and display operations. Node can be added in different ways.

Insertion of a node at the beginning:

The new node is all the time added before the head of the given Linked List. And recently added node becomes the new head of Doubly Link List.

Notes

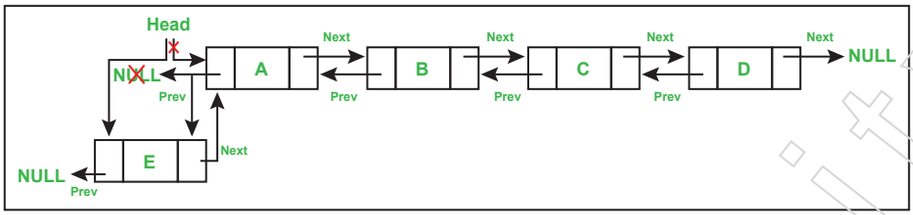


Fig 3.4 Insertion at begining

For example if the given Linked List is 20->10->1->5->2 and we include an item 4 at the front, then the Linked List becomes 4->20->10->1->5->2. We need to call push() function which adds at the front of the list. The push() must receive a pointer to the head pointer, because push must change the head pointer to point to the new node

Steps:

1. allocate node
2. insert the data
3. Make next of new node as head and previous as NULL
4. change prev of head node to new node
5. move the head to point to the new node

Insertion of a node after a specified node

By this way the new node can be inserted after a specified node.

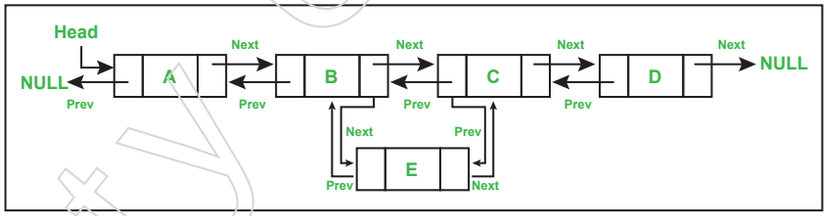


Fig 3.5 Insertion after specified node

Steps:

1. check if the given prev_node is NULL
2. allocate node
3. insert in the data
4. Make next of new node as next of prev_node
5. Make the next of prev_node as new_node
6. Make prev_node as previous of new_node
7. Change previous of new_node's next node

Insertion of node at the end

When we insert a node, the new node is always added after the last node. We need to traverse the list in order to change the nest of last node to new node.

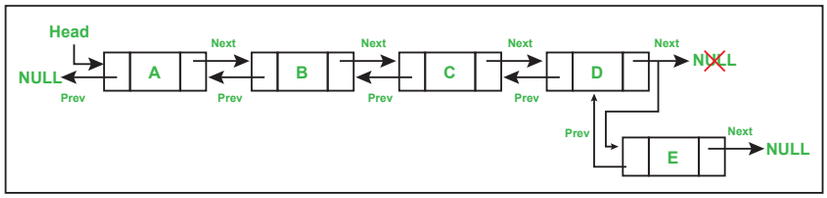


Fig 3.6 Insertion at the end

Steps

Let the pointer to this given node be next_node and the data of the new node to be added as new_data.

1. Check if the next_node is NULL or not. If it's NULL, return from the function because any new node cannot be added before a NULL
2. Allocate memory for the new node, let it be called new_node
3. Set new_node->data = new_data
4. Set the previous pointer of this new_node as the previous node of the next_ node, new_node->prev = next_node->prev
5. Set the previous pointer of the next_node as the new_node, next_node->prev = new_node
6. Set the next pointer of this new_node as the next_node, new_node->next = next_node;
7. If the previous node of the new_node is not NULL, then set the next pointer of this previous node as new_node, new_node->prev->next = new_node
8. Else, if the prev of new_node is NULL, it will be the new head node. So, make (*head_ref) = new_node.

Deletion

Deletion can be done at three places. Deletion at beginning, deletion at middle and deletion at end. All the cases can be done in two steps if the pointer of the node to be deleted and the head pointer is known.

Steps:

1. If we want to delete the first node then we need to make the second node as head node.
2. If a node is deleted then connect the next node and previous node of the deleted node.

Let us understand the deletion process with the help of few pictures.

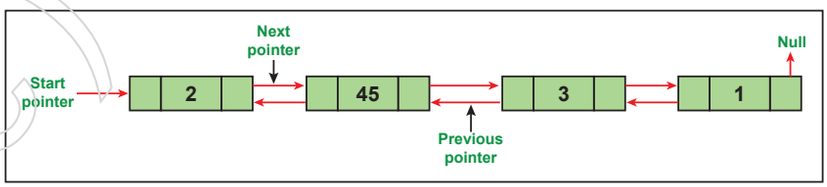


Fig 3.7: Original list

Notes

After deletion of head node:

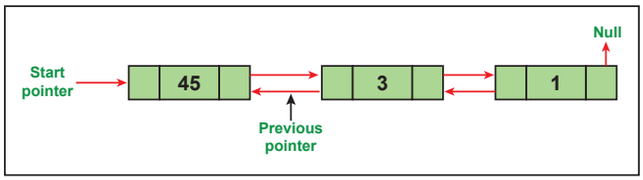


Fig 3.8 After head deletion

After deletion of middle node:

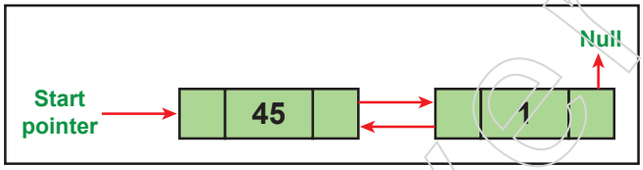


Fig 3.9 After middle node deletion

After deletion of last node:

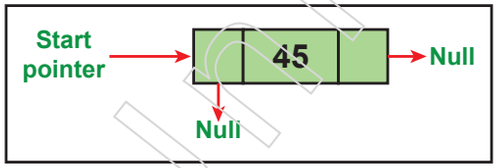


Fig 3.10 After last node deletion

Searching

While searching, we need to compare each node with the given node and if found we need to return the location of the item in the list, if not found then null has to be returned.

Traversing

Traversing simply means visiting each node in order to perform some specific operation.

Example:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   struct node *prev;
   struct node *next;
   int data;
};
struct node *head;
```

```
void beginninginsert();
void lastinsert();
void insertionother();
void deletion_beginning();
void deletion_last();
void deletion_specified();
void display();
void search();
void main ()
{
int choice =0;
   while(choice != 9)
   {
printf("\nChoose one option from the following list:");
printf("\n1.Insert in beginning\n2.Insert at last\n3.Insert at any other location\
n4.Delete from Beginning\n5.Delete from last\n6.Delete the node after a specified data\
n7.Search\n8.Display\n9.Exit\n");
printf("\nEnter your choice: ");
      scanf("\n%d",&choice);
      switch(choice)
      {
         case 1:
         beginninginsert();
         break;
         case 2:
         lastinsert();
         break;
         case 3:
         insertionother();
         break;
         case 4:
         deletion_beginning();
         break;
```

Notes

```
            case 5:
            deletion_last();
            break;
            case 6:
            deletion_specified();
            break;
            case 7:
            search();
            break;
            case 8:
            display();
            break;
            case 9:
            exit(0);
            break;
            default:
            printf("Please enter valid choice. ");
        }     } }
void beginninginsert()
{
  struct node *ptr;
  int item;
  ptr = (struct node *)malloc(sizeof(struct node));
  if(ptr == NULL)
  {
     printf("\nOVERFLOW");
  }
  else
  {
   printf("\nEnter Item value: ");
   scanf("%d",&item);
    if(head==NULL)
```

Notes

```
    {
        ptr->next = NULL;
        ptr->prev=NULL;
        ptr->data=item;
        head=ptr;
    }
    else
    {
        ptr->data=item;
        ptr->prev=NULL;
        ptr->next = head;
        head->prev=ptr;
        head=ptr;
    }
    printf("\nNode inserted\n");
} }
void lastinsert()
{
    struct node *ptr,*temp;
    int item;
    ptr = (struct node *) malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter value:");
        scanf("%d",&item);
         ptr->data=item;
        if(head == NULL)
        {
```

Notes

```
            ptr->next = NULL;
            ptr->prev = NULL;
            head = ptr;
        }
        else
        {
          temp = head;
          while(temp->next!=NULL)
          {
             temp = temp->next;
          }
          temp->next = ptr;
          ptr ->prev=temp;
          ptr->next = NULL;
          }        }
      printf("\nNode inserted\n");
    }
void insertionother()
{
   struct node *ptr,*temp;
   int item,loc,i;
   ptr = (struct node *)malloc(sizeof(struct node));
   if(ptr == NULL)
   {
      printf("\n OVERFLOW");
   }
   else
   {
      temp=head;
      printf("Enter the location: ");
      scanf("%d",&loc);
      for(i=0;i<loc;i++)
      {
```

Notes

```
            temp = temp->next;
            if(temp == NULL)
            {
                printf("\n There are less than %d elements", loc);
                return;
            }       }
        printf("Enter value:");
        scanf("%d",&item);
        ptr->data = item;
        ptr->next = temp->next;
        ptr -> prev = temp;
        temp->next = ptr;
        temp->next->prev=ptr;
        printf("\nNode inserted\n");
    } }
void deletion_beginning()
{
    struct node *ptr;
    if(head == NULL)
    {
        printf("\n UNDERFLOW");
    }
    else if(head->next == NULL)
    {
        head = NULL;
        free(head);
        printf("\nNode deleted\n");
    }
    else
    {
        ptr = head;
        head = head -> next;
        head -> prev = NULL;
```

Notes

```
        free(ptr);
        printf("\nNode deleted\n");
    }   }
void deletion_last()
{
    struct node *ptr;
    if(head == NULL)
    {
        printf("\n UNDERFLOW");
    }
    else if(head->next == NULL)
    {
        head = NULL;
        free(head);
        printf("\nNode deleted\n");
    }
    else
    {
        ptr = head;
        if(ptr->next != NULL)
        {
            ptr = ptr -> next;
        }
        ptr -> prev -> next = NULL;
        free(ptr);
        printf("\nNode deleted\n");
    } }
void deletion_specified()
{
    struct node *ptr, *temp;
    int val;
    printf("\n Enter the data after which the node is to be deleted : ");
    scanf("%d", &val);
```

Notes

```
    ptr = head;
    while(ptr -> data != val)
    ptr = ptr -> next;
    if(ptr -> next == NULL)
    {
        printf("\nCan't delete\n");
    }
    else if(ptr -> next -> next == NULL)
    {
        ptr ->next = NULL;
    }
    else
    {
        temp = ptr -> next;
        ptr -> next = temp -> next;
        temp -> next -> prev = ptr;
        free(temp);
        printf("\nNode deleted\n");
    }    }
void display()
{
    struct node *ptr;
    printf("\n The List is : \n");
    ptr = head;
    while(ptr != NULL)
    {
        printf("%d\n",ptr->data);
        ptr=ptr->next;
    } }
void search()
{
    struct node *ptr;
    int item,i=0,flag;
```

Notes

```
ptr = head;
if(ptr == NULL)
{
   printf("\nEmpty List\n");
}
else
{
   printf("\nEnter item which you want to search: ");
   scanf("%d",&item);
   while (ptr!=NULL)
   {
      if(ptr->data == item)
      {
         printf("\nItem found at location %d ",i+1);
         flag=0;
         break;
      }
      else
      {
         flag=1;
      }
      i++;
      ptr = ptr -> next;
   }
   if(flag==1)
   {
      printf("\nItem not found\n");        }    }     }
```

Output:

Choose one option from the following list:

1. Insert in beginning
2. Insert at last
3. Insert at any random location
4. Delete from Beginning

Notes

5. Delete from last
6. Delete the node after the given data
7. Search
8. Display
9. Exit

Enter your choice: 1

Enter Item value: 1

Node inserted

Choose one option from the following list:

1. Insert in beginning
2. Insert at last
3. Insert at any random location
4. Delete from Beginning
5. Delete from last
6. Delete the node after the given data
7. Search
8. Display
9. Exit

Enter your choice: 1

Enter Item value: 2

Node inserted

Choose one option from the following list:

1. Insert in beginning
2. Insert at last
3. Insert at any random location
4. Delete from Beginning
5. Delete from last
6. Delete the node after the given data
7. Search
8. Display
9. Exit

Enter your choice: 1

Enter Item value: 3

Node inserted

Notes

Choose one option from the following list:

1. Insert in beginning
2. Insert at last
3. Insert at any random location
4. Delete from Beginning
5. Delete from last
6. Delete the node after the given data
7. Search
8. Display
9. Exit

Enter your choice: 8

The list is:

3

2

1

Choose one option from the following list :

1. Insert in beginning
2. Insert at last
3. Insert at any random location
4. Delete from Beginning
5. Delete from last
6. Delete the node after the given data
7. Search
8. Display
9. Exit

Enter your choice: 2

Enter value: 89

Node inserted

Choose one option from the following list:

1. Insert in beginning
2. Insert at last
3. Insert at any random location
4. Delete from Beginning
5. Delete from last

Notes

6. Delete the node after the given data
7. Search
8. Display
9. Exit

Enter your choice: 4

Node deleted

Choose one option from the following list:

1. Insert in beginning
2. Insert at last
3. Insert at any random location
4. Delete from Beginning
5. Delete from last
6. Delete the node after the given data
7. Search
8. Display
9. Exit

Enter your choice: 8

2

1

89

Choose one option from the following list.

1. Insert in beginning
2. Insert at last
3. Insert at any random location
4. Delete from Beginning
5. Delete from last
6. Delete the node after the given data
7. Search
8. Display
9. Exit

Enter your choice: 9

Notes

Unit-3.3: Circular linked list

3.3.1 Introduction to Circular Linked List

Circular linked list is a linked list where all nodes are connected to form a circle i.e. the last node is connected to the first node. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list. To understand the complete traversal we need to reach the first node again.

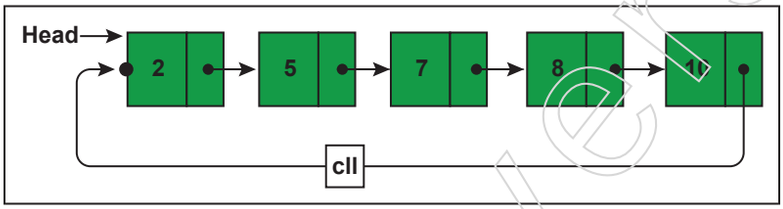


Fig 3.11 Circular linked list

Advantages of Circular Linked Lists:

1. Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
2. Useful for implementation of queue. Unlike this implementation, we don't need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.
3. Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.

3.3.2 Operations on Circular Linked List

In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.

We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

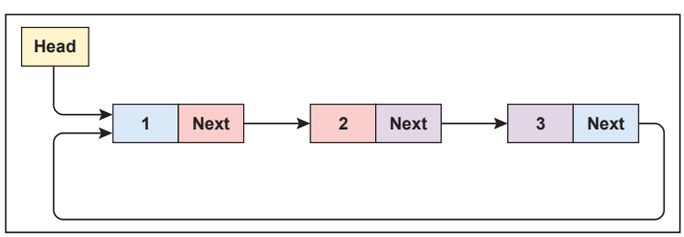


Fig 3.12 Memory representation

In the following image, memory representation of a circular linked list containing marks of a student in 4 subjects. However, the image shows a glimpse of how the circular list is being stored in the memory. The start or head of the list is pointing to the element with the index 1 and containing 13 marks in the data part and 4 in the next part. Which means that it is linked with the node that is being stored at 4th index of the list.

However, due to the fact that we are considering circular linked list in the memory therefore the last node of the list contains the address of the first node of the list.

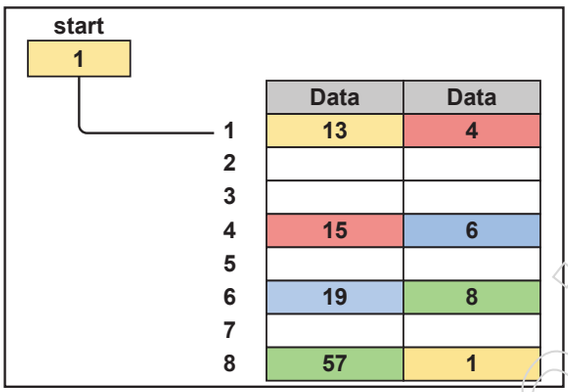


Fig 3.13 Addresing in circular linked list

However, due to the fact that we are considering circular linked list in the memory therefore the last node of the list contains the address of the first node of the list.

Insertion

Insertion at beginning

There are two scenario in which a node can be inserted in circular singly linked list at beginning. Either the node will be inserted in an empty list or the node is to be inserted in an already filled list.

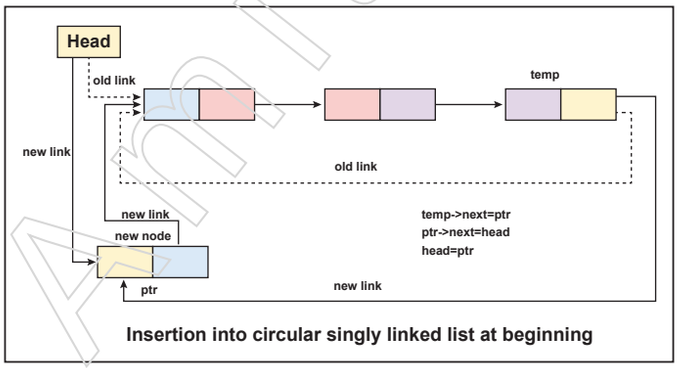


Fig 3.14 Insertion at the begining

Firstly, allocate the memory space for the new node by using the malloc method of C language.

```
struct node *ptr = (struct node *)malloc(sizeof(struct node));
```

Notes

In the first scenario, the condition **head == NULL** will be true. Since, the list in which, we are inserting the node is a circular singly linked list, therefore the only node of the list (which is just inserted into the list) will point to itself only. We also need to make the head pointer point to this node. This will be done by using the following statements.

```
If (head == NULL)
    {
      head = ptr;
      ptr -> next = head;
    }
```

In the second scenario, the condition head == NULL will become false which means that the list contains at least one node. In this case, we need to traverse the list in order to reach the last node of the list. This will be done by using the following statement.

```
temp = head;
        while(temp->next != head)
          temp = temp->next;
```

At the end of the loop, the pointer temp would point to the last node of the list. Since, in a circular singly linked list, the last node of the list contains a pointer to the first node of the list. Therefore, we need to make the next pointer of the last node point to the head node of the list and the new node which is being inserted into the list will be the new head node of the list therefore the next pointer of temp will point to the new node ptr. Which will be done by using temp -> next = ptr;

Algorithm

- Step 1: IF PTR = NULL

Write OVERFLOW

Go to step 11

[End of if]

- Step 2: SET NEW_NODE = PTR
- Step 3: SET PTR = PTR -> NEXT
- Step 4: SET NEW_NODE -> DATA = VAL
- Step 5: SET TEMP = HEAD
- Step 6: Repeat Step 8 while TEMP -> NEXT != HEAD
- Step 7: SET TEMP = TEMP -> NEXT

[END OF LOOP]

- Step 8: SET NEW_NODE -> NEXT = HEAD
- Step 9: SET TEMP → NEXT = NEW_NODE

Notes

- Step 10: SET HEAD = NEW_NODE
- Step 11: EXIT

Insertion at the end

We can insert node at end of circular link list.

Steps:

1. create a new node say node
2. make node->next = last->next
3. last-> next = node
4. last = node

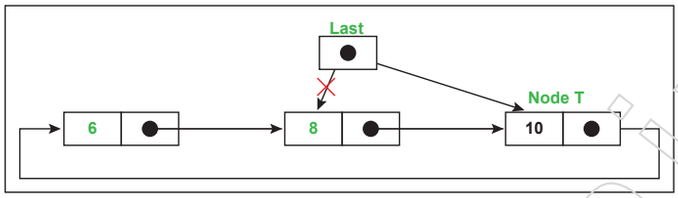


Fig 3.15 Insertion at end

Insertion in between nodes

Steps:

1. create a new node say node
2. search the node after which new node is to be inserted (say A)
3. make node->next = A->next
3. A-> next = node

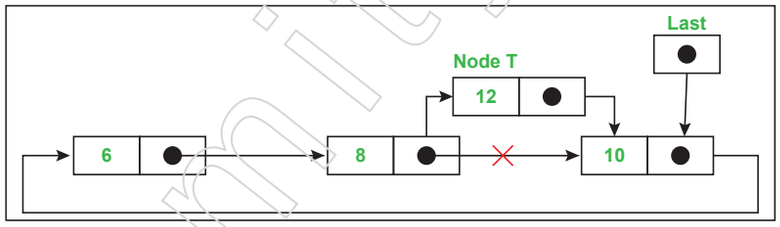


Fig 3.16 Insertion in between nodes

Deletion

Deletion can be done in two ways. Deletion at the beginning and deletion at the end.

Steps:

1. Define two pointers curr and prev if the node is not empty and initialize the pointer curr with the head node.
2. Then traverse the list using curr to find the node to be deleted and before moving curr to next node, we need to set prev = curr everytime.

3. If the node is found, check if it is the only node in the list. If yes, set head = NULL and free(curr).
4. If the list has more than one node, check if it is the first node of the list (curr == head). If yes, then move prev until it reaches the last node. After prev reaches the last node, set head = head -> next and prev -> next = head. Delete curr.
5. If curr is not the first node, check if it is the last node in the list (curr -> next == head).
6. If curr is the last node. Set prev -> next = head and delete the node curr using free(curr).
7. If the node to be deleted is neither the first node nor the last node, then set prev -> next = temp -> next and delete curr.

Searching

We need to compare each element of the node with the given node and return the location at which the node is present otherwise return null.

Traverse

Traversing means visiting each and every element of the list to perform some specific operation.

Example:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node *next;
};
struct node *head;
void begininsert ();
void lastinsert ();
void begindelete();
void lastdelete();
void random_delete();
void display();
void search();
void main ()
{
```

Notes

```
int choice =0;
while(choice != 7)
{
   printf("\nChoose one option from the following list:\n");
printf("\n1.Insert in beginning\n2.Insert at end\n3.Delete from beginning\n4.Delete from end\n5. Search for an element \n6.Display\n7.Exit\n");
printf("\nEnter your choice:\n");
scanf("\n%d",&choice);
    switch(choice)
    {
      case 1:
      begininsert();
      break;
      case 2:
      lastinsert();
      break;
      case 3:
      begindelete();
      break;
      case 4:
      lastdelete();
      break;
      case 5:
      search();
      break;
      case 6:
      display();
      break;
      case 7:
      exit(0);
      break;
      default:
```

Notes

```
            printf("Enter valid choice.");
        }
    }
}
void begininsert()
{
    struct node *ptr,*temp;
    int item;
    ptr = (struct node *)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter the node value:");
        scanf("%d",&item);
        ptr -> data = item;
        if(head == NULL)
        {
            head = ptr;
            ptr -> next = head;
        }
        else
        {
            temp = head;
            while(temp->next != head)
                temp = temp->next;
            ptr->next = head;
            temp -> next = ptr;
            head = ptr;
        }
```

Notes

```
        printf("\nNode inserted\n");
    }
}
void lastinsert()
{
    struct node *ptr,*temp;
    int item;
    ptr = (struct node *)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW\n");
    }
    else
    {
        printf("\nEnter Data:");
        scanf("%d",&item);
        ptr->data = item;
        if(head == NULL)
        {
            head = ptr;
            ptr -> next = head;
        }
        else
        {
            temp = head;
            while(temp -> next != head)
            {
                temp = temp -> next;
            }
            temp -> next = ptr;
            ptr -> next = head;
        }
```

Notes

```
            printf("\nNode inserted\n");
        }
    }
    void begindelete()
    {
        struct node *ptr;
        if(head == NULL)
        {
            printf("\nUNDERFLOW");
        }
        else if(head->next == head)
        {
            head = NULL;
            free(head);
            printf("\nNode deleted\n");
        }
        else
        {   ptr = head;
            while(ptr -> next != head)
                ptr = ptr -> next;
            ptr->next = head->next;
            free(head);
            head = ptr->next;
            printf("\nNode deleted\n");
        }
    }
    void lastdelete()
    {
        struct node *ptr, *preptr;
        if(head==NULL)
        {
            printf("\nUNDERFLOW");
```

Notes

```
    }
    else if (head ->next == head)
    {
       head = NULL;
       free(head);
       printf("\nNode deleted\n");
     }
    else
    {
       ptr = head;
       while(ptr ->next != head)
       {
          preptr=ptr;
          ptr = ptr->next;
       }
       preptr->next = ptr -> next;
       free(ptr);
       printf("\nNode deleted\n");
     }
}
void search()
{
    struct node *ptr;
    int item,i=0,flag=1;
    ptr = head;
    if(ptr == NULL)
    {
       printf("\nEmpty List\n");
    }
    else
    {
       printf("\nEnter item which you want to search: \n");
```

Notes

```
        scanf("%d",&item);
        if(head ->data == item)
        {
        printf("Item found at location %d",i+1);
        flag=0;
        }
        else
        {
        while (ptr->next != head)
        {
           if(ptr->data == item)
           {
              printf("Item found at location %d ",i+1);
              flag=0;
              break;
           }
           else
           {
              flag=1;
           }
           i++;
           ptr = ptr -> next;
        }
        }
        if(flag != 0)
        {
           printf("Item not found\n");
        }
     }
}
void display()
{
```

Notes

```
    struct node *ptr;
    ptr=head;
    if(head == NULL)
    {
        printf("\nNothing to display.");
    }
    else
    {
        printf("\n The list is : \n");

        while(ptr -> next != head)
        {

            printf("%d\n", ptr -> data);
            ptr = ptr -> next;
        }
        printf("%d\n", ptr -> data);
    }
}
```

Output:

Choose one option from the following list:

1. Insert in beginning
2. Insert at end
3. Delete from beginning
4. Delete from end
5. Search for an element
6. Display
7. Exit

Enter your choice: 1

Enter the node data: 10

Node inserted

Choose one option from the following list:

Notes

1. Insert in beginning
2. Insert at end
3. Delete from beginning
4. Delete from end
5. Search for an element
6. Display
7. Exit

Enter your choice: 1

Enter the node data: 20

Node inserted

Choose one option from the following list:

1. Insert in beginning
2. Insert at end
3. Delete from beginning
4. Delete from end
5. Search for an element
6. Display
7. Exit

Enter your choice: 1

Enter the node data: 30

Node inserted

Choose one option from the following list:

1. Insert in beginning
2. Insert at end
3. Delete from beginning
4. Delete from end
5. Search for an element
6. Display
7. Exit

Enter your choice: 4

Node deleted

Choose one option from the following list:

1. Insert in beginning
2. Insert at end

Notes

3. Delete from beginning
4. Delete from end
5. Search for an element
6. Display
7. Exit

Enter your choice: 5

Enter item which you want to search

20

Item found at location 1

Choose one option from the following list:

1. Insert in beginning
2. Insert at end
3. Delete from beginning
4. Delete from end
5. Search for an element
6. Display
7. Exit

Enter your choice: 7

Notes

Unit-3.4: Generalized List

A generalized list, A. is a finite sequence of elements. (b.c)): a list of length two; its first element is the atom a. and its second element is the linear list (b,c). (3) B = (A.A. ()) a list of length three whose first two elements are the list A. and the third element is the null list. A generalized linked list encompasses structures or elements with everyone containing its own pointer. It's generalized if the list can have any deletions, insertions, and similar inserted effectively into it.

A generalized list L is a finite sequence of n elements (n ≥ 0). The element ei is either an atom (single element) or another generalized list. The elements ei that are not atoms, they will be sub-list of L. Suppose L is ((A, B, C), ((D, E), F), G). Here L has three elements sub-list (A, B, C), sub-list ((D, E), F), and atom G. Again sub-list ((D, E), F) has two elements one sub-list (D, E) and atom F.

Notes

Unit-3.5: Applications and Implementation of Linked List

3.5.1 Applications of Linked List-Polynomial Representation Using Linked list and Basic Operation

To represent a polynomial of any degree we can use the linked list. If a single variable is used, the information field node contains two parts. One coefficient of variable and the other for degree of variable. A polynomial can be represented in an array or in a linked list by simply storing the coefficient and exponent of each term. However, for any polynomial operation , such as addition or multiplication of polynomials , you will find that the linked list representation is more easier to deal with. It would be very easy to represent the polynomial using a linked list structure, where each node can hold information pertaining to a single term of the polynomial. Each node will need to store the variable x, the exponent and the coefficient for each term. It often does not matter whether the polynomial is in x or y. This information may not be very crucial for the intended operations on the polynomial. Thus we need to define a node structure to hold two integers, exp and cof. Compare this representation with storing the same polynomial using an array structure. In the array we have to have keep a slot for each exponent of x, thus if we have a polynomial of order 50 but containing just 6 terms, then a large number of entries will be zero in the array. You will also see that it would be also easy to manipulate a pair of polynomials if they are represented using linked lists.

Example:

Polynomial: $3x^3 - 4x^2 + 2x - 9$

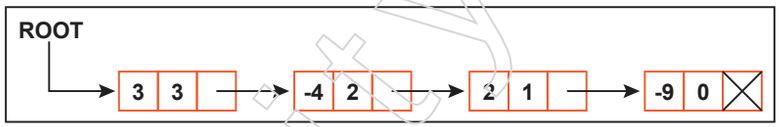


Fig 3.17 Polynomial in memory

In the above example, the external pointer ROOT point to the first node of the linked list. The first node of the linked list contains the data of the variable with highest degree. First node points to the next node with lower degree of variable. When we need to do operations such as addition or subtraction then representation of a polynomial using linked list is beneficial.

Example:

Input:

1st number = $5x^2 + 4x^1 + 2x^0$

2nd number = $-5x^1 - 5x^0$

Output:

$5x^2-1x^1-3x^0$

Input:

Notes

1st number = $5x^3 + 4x^2 + 2x^0$

2nd number = 5x^1 - 5x^0

Output:

$5x^3 + 4x^2 + 5x^1 - 3x^0$

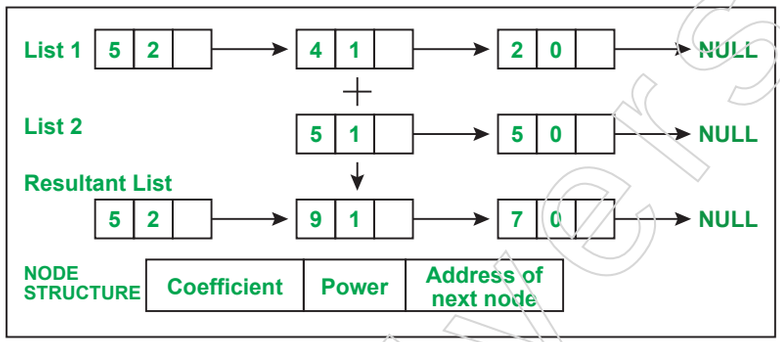


Fig 3.18 Polynomial using linked list

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct node
{
   int cof;
   int exp;
   struct node *link;
};
struct node *create(struct node *q);
struct node *insert(struct node *ptr, struct node *p);
void display(char const *tag, struct node *ptr);
void err_exit(char const *tag);
struct node *create(struct node *q)
{
   int i, n;
   printf("Enter the number of nodes: ");
   if (scanf("%d", &n) != 1)
      err_exit("Read error (number of nodes)");
```

Notes

```
    for (i = 0; i < n; i++)
    {
        struct node *ptr = (struct node *)malloc(sizeof(struct node));
        if (ptr == 0)
          err_exit("Out of memory (1)");
        printf("enter the coefficient and exponent respectively: ");
        if (scanf("%d%d", &ptr->cof, &ptr->exp) != 2)
                err_exit("Read error (coefficient and exponent)");
        ptr->link = NULL;
        q = insert(ptr, q);
        display("after input", q);
    }
    return q;
}
struct node *insert(struct node *ptr, struct node *p)
{
    struct node *temp, *b;
    if (p == NULL)
        p = ptr;
    else
    {
        display("insert: p   = ", p);
        display("insert: ptr = ", ptr);
        if (p->exp < ptr->exp)
        {
            ptr->link = p;
            p = ptr;
        }
        else
        {
            temp = p;
            while ((temp->link != NULL) && (temp->link->exp < ptr->exp))
```

Notes

```
            display("insert: tmp = ", temp),
            temp = temp->link;
        display("insert: post loop", temp);
        b = temp->link;
        temp->link = ptr;
        ptr->link = b;
      }
    }
    return p;
}
void display(char const *tag, struct node *ptr)
{
    struct node *temp;
    const char *pad = "";
    temp = ptr;
    printf("%s: ", tag);
    while (temp != NULL)
    {
        printf("%s%d x ^ %d", pad, temp->cof, temp->exp);
        temp = temp->link;
        pad = " + ";
    }
    putchar('\n');
}
int main(void)
{
    printf("Enter the first polynomial:\n");
    struct node *p1 = NULL, *p2 = NULL;
    p1 = create(p1);
    printf("Enter the second polynomial:\n");
    p2 = create(p2);
    display("p1", p1);
```

Notes

```
    display("p2", p2);
    return 0;
}
void err_exit(char const *tag)
{
    fprintf(stderr, "%s\n", tag);
    exit(1);
}
```

3.5.2 Stack and Queue Implementation Using Linked List

Implement a stack applying single linked list concept. all the single linked list operations operate based on Stack operations LIFO(last in first out) and with the help of that knowledge we are going to implement a stack using single linked list. By means of single linked lists means what we are storing the information in the form of nodes and we need to follow the stack rules and we need to implement using single linked list nodes so what are the rules we need to follow in the implementation of a stack a simple rule that is last in first out and all the operations we should perform so with the help of a top variable only with the help of top variables are how to insert the elements let's see

A stack can be simply implemented through the linked list. In stack Implementation, a stack includes a top pointer which is "head" of the stack where pushing and popping items occurs at the head of the list. first node has null in link field and second node link have first node address in link field and so on and last node address in "top" pointer.

The main advantage of using linked list over an array is that it is feasible to implements a stack that can shrink or grow as much as needed. In using array will put a limitation to the maximum capacity of the array which can lead to stack overflow. Here each new node will be dynamically allocated, so, overflow is not possible.

Stack Operations:

1. push() : Insert the element into linked list nothing but which is the top node of Stack.
2. pop() : Return top element from the Stack and move the top pointer to the second node of linked list or Stack.
3. peek(): Return the top element.
4. display(): Print all element of Stack.

Example:

```
#include <stdio.h>
int MAXSIZE = 8;
int stack[8];
int top = -1;
```

Notes

```
int isempty() {
   if(top == -1)
      return 1;
   else
      return 0;
}
int isfull() {
   if(top == MAXSIZE)
      return 1;
   else
      return 0;
}
int peek() {
   return stack[top];
}
int pop() {
int data;
   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   } else {
      printf("Could not retrieve data, Stack is empty.\n");
   }
}
int push(int data) {
   if(!isfull()) {
      top = top + 1;
      stack[top] = data;
   } else {
      printf("Could not insert data, Stack is full.\n");
   }
```

Notes

```
}
int main() {
   // push items on to the stack
   push(3);
   push(5);
   push(9);
   push(1);
   push(12);
   push(15);
   printf("Element at top of the stack: %d\n" ,peek());
   printf("Elements: \n");
   // print stack data
   while(!isempty()) {
      int data = pop();
      printf("%d\n",data);
   }
   printf("Stack full: %s\n" , isfull()?"true":"false");
   printf("Stack empty: %s\n" , isempty()?"true":"false");
   return 0;
}
```

If we compile and run the above program, it will produce the following result:

Output:

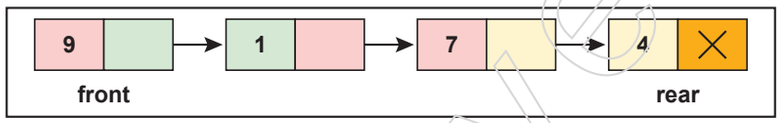
```
Element at top of the stack: 15
Elements:
15
12
1
9
5
3
Stack full: false
Stack empty: true
```

Notes

In a linked queue, every single node of the queue comprises of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.

In the linked queue, there are two pointers preserved in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer comprises the address of the last element of the queue.

Insertion and deletions are completed at rear and front end respectively. If front and rear both are NULL, it signifies that the queue is empty. The linked representation of queue is shown in the following figure.



There are two basic operations which can be employed on the linked queues. The operations are Insertion and Deletion.The insert operation appends the queue by adding an element to the end of the queue. The new element will be the last element of the queue.

Firstly, allocate the memory for the new node ptr by using the following statement.

1. Ptr = (struct node *) malloc (sizeof(struct node));

we insert element into an empty queue. In this case, the condition front = NULL becomes true. Now, the new element will be included as the only element of the queue and the subsequent pointer of front and rear pointer both, will point to NULL.

```
ptr -> data = item;
    if(front == NULL)
    {
       front = ptr;
       rear = ptr;
       front -> next = NULL;
       rear -> next = NULL;
    }
```

Algorithm

- Step 1: Allocate the space for the new node PTR
- Step 2: SET PTR -> DATA = VAL
- Step 3: IF FRONT = NULL

SET FRONT = REAR = PTR

SET FRONT -> NEXT = REAR -> NEXT = NULL

ELSE

Notes

SET REAR -> NEXT = PTR

SET REAR = PTR

SET REAR -> NEXT = NULL

[END OF IF]

- Step 4: END

Deletion

Deletion operation eliminates the element that is first inserted among all the queue elements. Initially it is required to check either the list is empty or not. The condition front == NULL becomes true if the list is empty, in this case, we simply write underflow on the console and make exit.

Else, we will delete the element that is pointed by the pointer front. For this purpose, copy the node pointed by the front pointer into the pointer ptr. Now, shift the front pointer, point to its next node and free the node pointed by the node ptr. This is done by using the following statements.

```
ptr = front;
    front = front -> next;
      free(ptr);
```

Algorithm

- Step 1: IF FRONT = NULL

Write " Underflow "

Go to Step 5

[END OF IF]

- Step 2: SET PTR = FRONT
- Step 3: SET FRONT = FRONT -> NEXT
- Step 4: FREE PTR
- Step 5: END

Module End Questions

Assignment I

Use the following Node definition for all problems:

```
struct Node
{
  typedef int Item;
```

Notes

```
    Item data;
    Node *link;
}
```

1. Write a function to concatenate two linked lists. Given lists l1 = (2, 3, 1) and l2 = (4, 5), after return from concatenate(l1,l2) the list l1 should be changed to be l1 = (2, 3, 1, 4, 5). Your function should not change l2 and should not directly link nodes from l1 to l2 (i.e. the nodes inserted into l1 should be copies of the nodes from l2.)

```
void concatenate(Node*& h1, Node* h2 );
// Precondition: h1 and h2 are head pointers of linked lists.
// The lists may be empty or non-empty.
// Postcondition: A copy of list h2 is concatenated (added to the end)
// of list h1.  List h2 should be unchanged by the function.
// NOTE: The nodes added to the list h1 must be copies of the
// nodes in list h2.
```

2. Write a function to insert a number into a sorted linked list. Assume the list is sorted from smallest to largest value. After insertion, the list should still be sorted. Given the list l1 = (3, 17, 18, 27) and the value 20, on return l1 be the list (3, 17, 18, 20, 27).

```
void insertInOrder(Node*& head_ptr, int value);
// Precondition: head_ptr is the head pointer of a linked list
// sorted in non-decreasing order. The list may be empty or non-empty.
// Postcondition: The number value is inserted in the list.
// The list should be sorted on return from the function.
```

3. Write a function to return the median value in a sorted linked list. If the length i of the list is odd, then the median is the ceiling(i/2) member. For example, given the list (1, 2, 2, 5, 7, 9, 11) as input, your function should return the value 5. If the length of the list is even, then the median is the mean of the i/2 and (i/2)+1 members. Thus, the median of the sorted list (2, 4, 8, 9) is (4+8)/2. Finally, define the median of an empty list to be 0.

```
FILL IN RETURN TYPE listMedian ---- FILL IN ARGUMENTS ----------
// Precondition: ----  FILL IN -------
// Postcondition: ----  FILL IN -------
```

4. Write a function to reverse the nodes in a linked list. Your function should have time complexity O(n), where n is the length of the list. You should create no new nodes.

```
void reverse(Node*& head_ptr);
// Precondition: head_ptr is the head pointer of a linked list.
// The list may be empty or non-empty.
```

Notes

```
// Postcondition: head_ptr points to the list of Nodes in reverse
// order.
```

Linked List Problem and Solutions

1. How do you reverse a linked list?

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

// A Linked List Node
struct Node
{
int data;
struct Node* next;
};
// Helper function to print a given linked list
void printList(struct Node* head)
{
struct Node* ptr = head;
while (ptr)
{
printf("%d —> ", ptr->data);
ptr = ptr->next;
}
printf("NULL\n");
}
// Helper function to insert a new node at the beginning of the linked list
void push(struct Node** head, int data)
{
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = data;
newNode->next = *head;
*head = newNode;
```

Notes

```
}
// Reverses a given linked list by changing its `.next` pointers and
// its head pointer. Takes a pointer (reference) to the head pointer
void reverse(struct Node** head)
{
struct Node* previous = NULL;   // the previous pointer
struct Node* current = *head;   // the main pointer
// traverse the list
while (current != NULL)
{
// tricky: note the next node
struct Node* next = current->next;
current->next = previous;   // fix the current node
// advance the two pointers
previous = current;
current = next;
}
// fix the head pointer to point to the new front
*head = previous;
}
int main(void)
{
// input keys
int keys[] = { 1, 2, 3, 4, 5, 6 };
int n = sizeof(keys)/sizeof(keys[0]);

struct Node *head = NULL;
for (int i = n - 1; i >=0; i--) {
push(&head, keys[i]);
}
reverse(&head);
printList(head);
```

Notes

```
return 0;
}
```

Questions

1. In a circular linked list
   a) Components are all linked together in some sequential manner.
   b) There is no beginning and no end.
   c) Components are arranged hierarchically.
   d) Forward and backward traversal within the list is permitted.
2. Which of the following operations is performed more efficiently by doubly linked list than by singly linked list?
   a) Deleting a node whose location in given
   b) Searching of an unsorted list for a given item
   c) Inverting a node after the node with given location
   d) Traversing a list to process each node
3. A linear collection of data elements where the linear node is given by means of pointer is called?
   a) linked list
   b) node list
   c) primitive list
   d) None of these
4. Linked lists are not suitable to for the implementation of?
   a) Insertion sort
   b) Radix sort
   c) Polynomial manipulation
   d) Binary search
5. Which of these is an application of linked lists?
   a) To implement file systems
   b) For separate chaining in hash-tables
   c) To implement non-binary trees
   d) All of the mentioned
6. In circular linked list, insertion of node requires modification of?
   a) One pointer
   b) Two pointer
   c) Three pointer
   d) None

Notes

7. Which of these is not an application of a linked list?
   a) To implement file systems
   b) For separate chaining in hash-tables
   c) To implement non-binary trees
   d) Random Access of elements
8. A linear collection of data element given by mean of pointer is called
   a) Queue
   b) Linked list
   c) Graph
   d) Stack
9. Which of the following operation is performed more efficiently in doubly linked list?
   a) Searching a node at given position
   b) Inserting a node at given position
   c) Deleting node at given position
   d) None of these
10. Which of the following is two way list
   a) gounded header list
   b) circular header list
   c) linked list with header and trailer node
   d) None of the above

Answer key:

1. b, 2. a, 3. a, 4. d, 5. d, 6. b, 7. d, 8. b, 9. c, 10. d

Module-4: Trees and Graphs

Notes

Structure:

Notes

Unit-4.1: Introduction to Trees

4.1.1 Introduction to Trees

Tree is a hierarchical data structure which stores the information naturally in the form of ladder style. Tree is one of the most effective and advanced data structures. It is a non-linear data structure compared to arrays, linked lists, stack and queue. It signifies the nodes connected by edges.

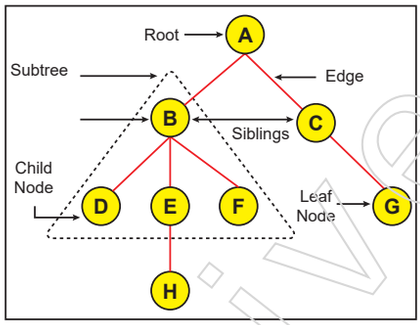


Fig 4.1 Tree

The above figure represents structure of a tree. Tree has 2 subtrees. A is a parent of B and C. B is called a child of A and also parent of D, E, F. Tree is a collection of elements called Nodes, where each node can have arbitrary number of children.

4.1.2 Basic Terminology

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.

Unit-4.2: Binary Trees

Notes

4.2.1 Binary Trees and Their Representation

Binary Tree is a unique data structure used for data storage. Binary tree has at most 2 children. Hence the name binary tree. The topmost node is called root and the nodes that are under the root node are called its children. The node directly above another node is called as parent.

Binary Tree is a unique data structure used for data storage reasons . A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

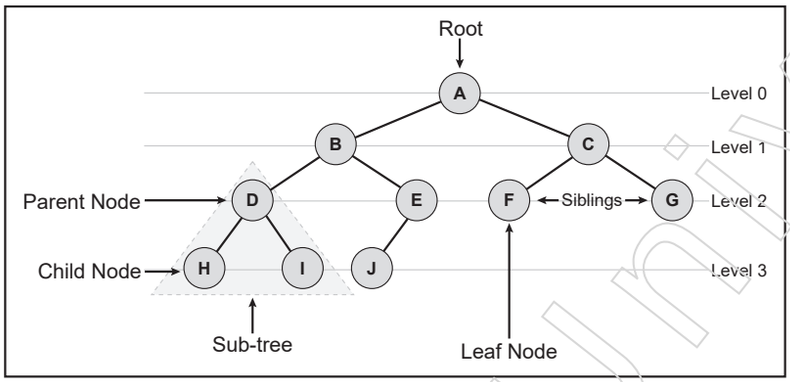


Fig4.2 : Binary tree

A Binary Tree node encompasses following parts.

1. Data
2. Pointer to left child
3. Pointer to right child

Binary Search tree displays a special behaviour. A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value.

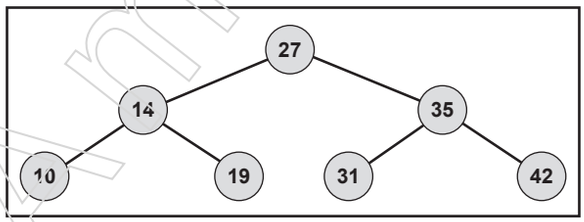


Fig 4.3 : Binary tree

The basic operations that can be completed on a binary search tree data structure, are the following:

- Insert − Inserts an element in a tree/create a tree.
- Search − Searches an element in a tree.
- Preorder Traversal − Traverses a tree in a pre-order manner.

Notes

- Inorder Traversal − Traverses a tree in an in-order manner.
- Postorder Traversal − Traverses a tree in a post-order manner.

Example:

```
#include<stdlib.h>
#include<stdio.h>
struct bin_tree {
int data;
struct bin_tree * right, * left;
};
typedef struct bin_tree node;
void insert(node ** tree, int val)
{
node *temp = NULL;
if(!(*tree))
{
temp = (node *)malloc(sizeof(node));
temp->left = temp->right = NULL;
temp->data = val;
*tree = temp;
return;
}
if(val < (*tree)->data)
{
insert(&(*tree)->left, val);
}
else if(val > (*tree)->data)
{
insert(&(*tree)->right, val);
}
}
void print_preorder(node * tree)
{
```

Notes

```
if (tree)
{
printf("%d\n",tree->data);
print_preorder(tree->left);
print_preorder(tree->right);
}
}
void print_inorder(node * tree)
{
if (tree)
{
print_inorder(tree->left);
printf("%d\n",tree->data);
print_inorder(tree->right);
}
}
void print_postorder(node * tree)
{
if (tree)
{
print_postorder(tree->left);
print_postorder(tree->right);
printf("%d\n",tree->data);
}
}
void deltree(node * tree)
{
if (tree)
{
deltree(tree->left);
deltree(tree->right);
free(tree);
```

Notes

```
}
}
node* search(node ** tree, int val)
{
if(!(*tree))
{
return NULL;
}
if(val < (*tree)->data)
{
search(&((*tree)->left), val);
}
else if(val > (*tree)->data)
{
search(&((*tree)->right), val);
}
else if(val == (*tree)->data)
{
return *tree;
}
}
void main()
{
node *root;
node *tmp;
root = NULL;
/* Inserting nodes into tree */
insert(&root, 9);
insert(&root, 4);
insert(&root, 15);
insert(&root, 6);
insert(&root, 12);
```

Notes

```
insert(&root, 17);
insert(&root, 2);
/* Printing nodes of tree */
printf("Pre Order Display\n");
print_preorder(root);
printf("In Order Display\n");
print_inorder(root);
printf("Post Order Display\n");
print_postorder(root);
/* Search node into tree */
tmp = search(&root, 4);
if (tmp) {
printf("Searched node=%d\n", tmp->data);    }
else
{printf("Data Not found in tree.\n");
}
/* Deleting all nodes of tree */
deltree(root);    }
```

Output:

```
Pre Order Display
9
4
2
6
15
12
17
in Order Display
2
4
6
9
```

Notes

12

15

17

Post Order Display

2

6

4

12

17

15

9

Searched node=4

4.2.2 Expression Evaluation

Evaluate an expression represented by a String. The expression can contain parentheses, you can assume parentheses are well-matched. For simplicity, you can assume only binary operations allowed are +, -, *, and /. Arithmetic Expressions can be written in one of three forms:

Infix Notation: Operators are written between the operands on, e.g. 3+4.

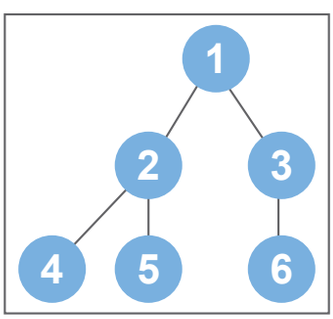
Prefix Notation: Operators are written before the operands, e.g + 3 4

Postfix Notation: Operators are written after operands.

4.2.3 Complete Binary Trees

A complete binary tree is a binary tree in which all the levels are completely filled except possibly the lowest one, which is filled from the left. A complete binary tree is just like a full binary tree, but with two major differences

1. All the leaf elements must lean towards the left.
2. The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.

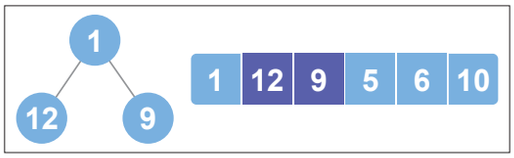


Notes

Steps to create Complete Binary Tree:

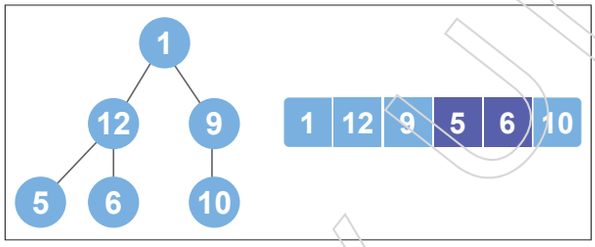
Select the first element of the list to be the root node. (no. of elements on level-I: 1)

Put the second element as a left child of the root node and the third element as the right child. (no. of elements on level-II: 2)



Put the next two elements as children of the left node of the second level. Again, put the next two elements as children of the right node of the second level (no. of elements on level-III: 4) elements).

Keep repeating until you reach the last element.



Example:

```
//Checking if a binary tree is a complete binary tree
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
struct Node {
  int key;
  struct Node *left, *right;
};
// Node creation
struct Node *newNode(char k) {
  struct Node *node = (struct Node *)malloc(sizeof(struct Node));
  node->key = k;
  node->right = node->left = NULL;
```

Notes

```
  return node;
}
// Count the number of nodes
int countNumNodes(struct Node *root) {
  if (root == NULL)
    return (0);
  return (1 + countNumNodes(root->left) + countNumNodes(root->right));
}
// Check if the tree is a complete binary tree
bool checkComplete(struct Node *root, int index, int numberNodes) {
  // Check if the tree is complete
  if (root == NULL)
    return true;
  if (index >= numberNodes)
    return false;
  return (checkComplete(root->left, 2 * index + 1, numberNodes) &&
checkComplete(root->right, 2 * index + 2, numberNodes));
}
int main() {
  struct Node *root = NULL;
  root = newNode(1);
  root->left = newNode(2);
  root->right = newNode(3);
  root->left->left = newNode(4);
  root->left->right = newNode(5);
  root->right->left = newNode(6);
  int node_count = countNumNodes(root);
  int index = 0;
  if (checkComplete(root, index, node_count))
    printf("The tree is a complete binary tree\n");
  else
    printf("The tree is not a complete binary tree\n");
```

}

Notes

4.2.4 Extended Binary trees

Extended binary tree is a type of binary tree in which all the null sub tree of the original tree are replaced with special nodes called external nodes whereas other nodes are called internal nodes

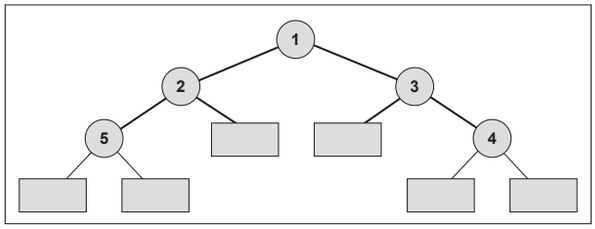


Fig 4.4 : Extended Binary tree

Here the circles represent the internal nodes and the boxes represent the external nodes.

Properties of External binary tree

1. The nodes from the original tree are internal nodes and the special nodes are external nodes.
2. All external nodes are leaf nodes and the internal nodes are non-leaf nodes.
3. Every internal node has exactly two children and every external node is a leaf. It displays the result which is a complete binary tree

Below is an example of making an extended binary tree in C++ by making all the external nodes as '-1'

4.2.5 Traversing Binary Trees

Often we wish to process a binary tree by "visiting" each of its nodes, each time performing a specific action such as printing the contents of the node. Any process for visiting all of the nodes in some order is called a traversal. Any traversal that lists every node in the tree exactly once is called an enumeration of the tree's nodes. Some applications do not require that the nodes be visited in any particular order as long as each node is visited precisely once. For other applications, nodes must be visited in an order that preserves some relationship.

Tree traversal is the process of visiting each node in the tree exactly once. Visiting each node in a graph should be done in a systematic manner. If search result in a visit to all the vertices, it is called a traversal. There are basically three traversal techniques for a binary tree that are,

1. Pre-order traversal
2. In-order traversal
3. Post-order traversal

Notes

1) Pre-order traversal

To traverse a binary tree in pre-order, following operations are carried out:

a. Visit the root.

b. Traverse the left sub tree of root.

c. Traverse the right sub tree of root.

Note: Pre-order traversal is also known as NLR traversal.

2) Inorder traversal

To traverse a binary tree in in-order traversal, following operations are carried out:

a. Traverse the left most sub tree.

b. Visit the root.

c. Traverse the right most sub tree.

3) Postorder traversal

To traverse a binary tree in postorder traversal, following operations are carried out:

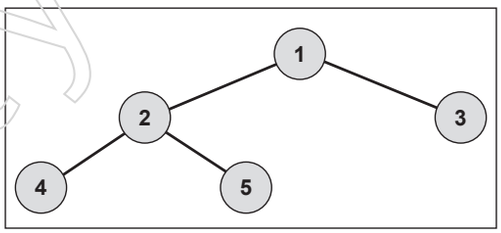
a. Traverse the left sub tree of root.

b. Traverse the right sub tree of root.

c. Visit the root.

Example:

Consider the below figure and let us understand the program of In-order, Pre-order and Post-order traversal.




```
#include <stdio.h>
#include <stdlib.h>
 /* A binary tree node has data, pointer to left child  and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};
```

Notes

```
/* function that allocates a new node with the given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
    malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
     return(node);
}
/* Given a binary tree, print its nodes according to the “bottom-up” post-order traversal. */
void postorder(struct node* node)
{
    if (node == NULL)
       return;
  postorder(node->left);
  postorder(node->right);
  printf(“%d “, node->data);
}
 /* Given a binary tree, print its nodes in in-order*/
void inorder(struct node* node)
{
    if (node == NULL)
         return;
    inorder(node->left);
   printf(“%d “, node->data);
    inorder(node->right);
}
void preorder(struct node* node)
{
    if (node == NULL)
```

Notes

```
        return;
    printf("%d ", node->data);
    preorder(node->left);
    preorder(node->right);
}
int main()
{
    struct node *root  = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
   printf("\nPre-order traversal of binary tree is: ");
    preorder(root);
   printf("\nIn-order traversal of binary tree is: ");
    inorder(root);
   printf("\nPos-torder traversal of binary tree is: ");
    postorder(root);
      getchar();
    return 0;
}
```

Output:

Pre-order traversal of binary tree is: 1 2 4 5 3

In-order traversal of binary tree is: 4 2 5 1 3

Post-order traversal of binary tree is: 4 5 2 3 1

4.2.6 Searching, Insertion and Deletion in Binary Search Tree

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties.

The value of the key of the left sub-tree is less than the value of its parent (root) node's key.

The value of the key of the right sub-tree is greater than or equal to the value of its parent (root) node's key.

Notes

Search Operation

Whenever an element is to be searched, start searching from the root node. Then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

Algorithm:

```
struct node* search(int data){
   struct node *current = root;
   printf("Visiting elements: ");
   while(current->data != data){
      if(current != NULL) {
         printf("%d ",current->data);
         //go to left tree
         if(current->data > data){
            current = current->leftChild;
         }  //else go to right tree
         else {
            current = current->rightChild;
         }
         //not found
         if(current == NULL){
            return NULL;
         }
      }
   }
   return current;
}
```

Insert Operation

Whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Notes

Deletion

Delete function is used to delete the specified node from a binary search tree. However, we must delete a node from a binary search tree in such a way, that the property of binary search tree doesn't violate. There are three situations of deleting a node from binary search tree.

It is the simplest case, in this case, replace the leaf node with the NULL and simple free the allocated space.

In the following image, we are deleting the node 85, since the node is a leaf node, therefore the node will be replaced with NULL and allocated space will be freed.

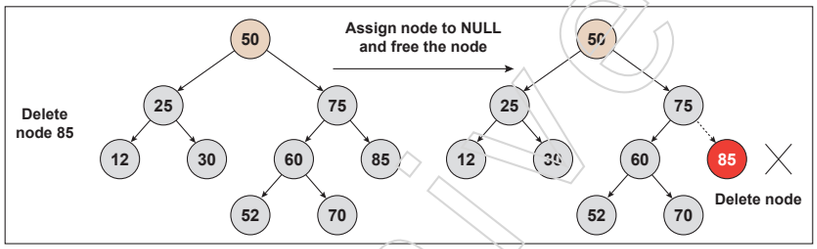


Fig 4.5 : Deletion

In this case, replace the node with its child and delete the child node, which now contains the value which is to be deleted. Simply replace it with the NULL and free the allocated space.

In the following image, the node 12 is to be deleted. It has only one child. The node will be replaced with its child node and the replaced node 12 (which is now leaf node) will simply be deleted.

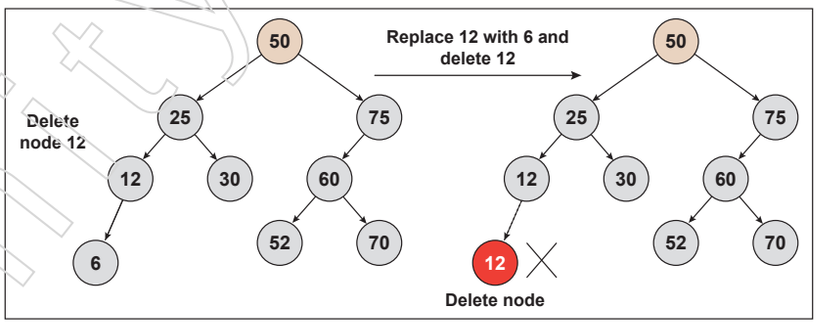


Fig 4.6 : Deletion

Algorithm

1. Define Node which has three attributes namely: data, left and right where left represents the left child of the node and right represents the right child of the node.
2. When a node is created, data will pass to the data attribute of the node and both left and right will be set to null.
3. Define another class which has an attribute root.
   a. Root represents the root node of the tree and initializes it to null.

Notes

4. insert() will insert the new value into a binary search tree:

It checks whether root is null, which means tree is empty. New node will become root node of tree.

a. If tree is not empty, it will compare value of new node with root node. If value of new node is greater than root, new node will be inserted to right subtree. Else, it will be inserted in left subtree.

5. deleteNode() will delete a particular node from the tree:

a. If value of node to be deleted is less than root node, search node in left subtree. Else, search in right subtree.

b. If node is found and it has no children, then set the node to null.

c. If node has one child then, child node will take position of node.

d. If node has two children then, find a minimum value node from its right subtree. This minimum value node will replace the current node.

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
struct node{
   int data;
   struct node *left;
   struct node *right;
};
struct node *root= NULL;
//createNode() will create a new node
struct node* createNode(int data){
   struct node *newNode = (struct node*)malloc(sizeof(struct node));
   newNode->data= data;
   newNode->left = NULL;
   newNode->right = NULL;
   return newNode;
}
//insert() will add new node to the binary search tree
void insert(int data) {
   struct node *newNode = createNode(data);
```

Notes

```
    if(root == NULL){
        root = newNode;
        return;
     }
    else {
        struct node *current = root, *parent = NULL;
        while(true) {
            parent = current;
            if(data < current->data) {
                current = current->left;
                if(current == NULL) {
                    parent->left = newNode;
                    return;
                }
            }
            else {
                current = current->right;
                if(current == NULL) {
                    parent->right = newNode;
                    return;
                }          }        }     } }
  //minNode() will find out the minimum node
struct node* minNode(struct node *root) {
    if (root->left != NULL)
        return minNode(root->left);
    else
        return root;
}
//deleteNode() will delete the given node from the binary search tree
struct node* deleteNode(struct node *node, int value) {
    if(node == NULL){
         return NULL;
```

Notes

```
        }
        else {
            if(value < node->data)
                node->left = deleteNode(node->left, value);
            else if(value > node->data)
                node->right = deleteNode(node->right, value);
            else {
                if(node->left == NULL && node->right == NULL)
                    node = NULL;
                else if(node->left == NULL) {
                    node = node->right;
                }
            else if(node->right == NULL) {
                    node = node->left;
                }
            else {
            struct node *temp = minNode(node->right);
                node->data = temp->data;
                node->right = deleteNode(node->right, temp->data);
                }
            }
            return node;
        }
    }
    //inorder() will perform in-order traversal on binary search tree
    void inorder(struct node *node) {
        //Check whether tree is empty
        if(root == NULL){
            printf("Tree is empty\n");
            return;
        }
        else {
```

Notes

```
            if(node->left!= NULL)
            inorder(node->left);
        printf("%d ", node->data);
        if(node->right!= NULL)
          inorder(node->right);
    }
}
int main()
{
    //Add nodes to the binary tree
    insert(5);
    insert(3);
    insert(7);
    insert(6);
    insert(1);
    insert(9);
    printf("Binary search tree after insertion: ");
    //Displays the binary tree
    inorder(root);
    struct node *deletedNode = NULL;
    //Deletes node 9 which has no child
    deletedNode = deleteNode(root, 9);
    printf("\nBinary search tree after deleting node 9: ");
    inorder(root);
    //Deletes node 3 which has one child
    deletedNode = deleteNode(root, 3);
    printf("\nBinary search tree after deleting node 3: ");
    inorder(root);
    //Deletes node 5 which has two children
    deletedNode = deleteNode(root, 5);
    printf("\nBinary search tree after deleting node 5: ");
    inorder(root);
```

```
    return 0;
}
```

Output:

Binary search tree after insertion:

1 3 5 6 7 9

Binary search tree after deleting node 9:

1 3 5 6 7

Binary search tree after deleting node 3:

1 5 6 7

Binary search tree after deleting node 5:

1 6 7

Notes

Notes

Unit-4.3: General Trees

4.3.1 General Trees

General tree is a tree in which every node can have either zero or many child nodes. It cannot be empty. In general tree, there is no limitation on the degree of a node. The topmost node of a general tree is termed the root node. There are several subtrees in a general tree. The subtree of a general tree is unordered since the nodes of the general tree cannot be ordered according to specific criteria. In a general tree, each node has in-degree(number of parent nodes) one and maximum out-degree(number of child nodes) n.

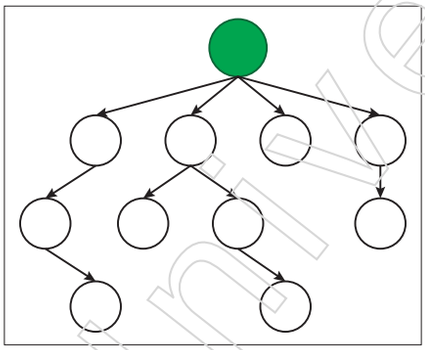


Fig 4.7 : General tree

4.3.2 AVL Tree

AVL tree is a self-balancing binary search tree in which each node maintains extra information called a balance factor whose value is either -1, 0 or +1.

Balance factor of a node in an AVL tree is the difference between the height of the left subtree and that of the right subtree of that node.

Balance Factor = (Height of Left Subtree - Height of Right Subtree) or (Height of Right Subtree - Height of Left Subtree)

The self balancing property of an avl tree is maintained by the balance factor. The value of balance factor should always be -1, 0 or +1.

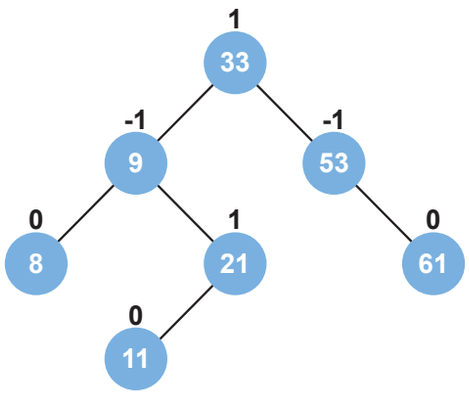


Fig 4.8 : AVL tree

Notes

Various operations that can be performed on an AVL tree are:

Rotating the subtrees in an AVL Tree

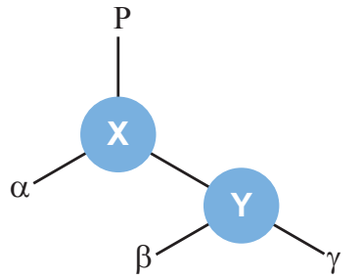
In rotation operation, the positions of the nodes of a subtree are interchanged. There are two types of rotations:

Left Rotate

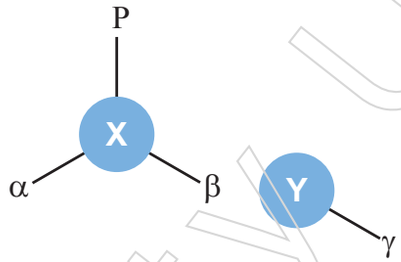
In left-rotation, the arrangement of the nodes on the right is transformed into the arrangements on the left node.

Algorithm:

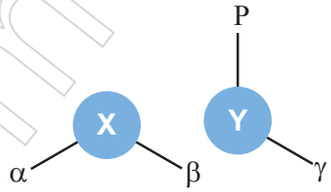
Let the initial tree be:



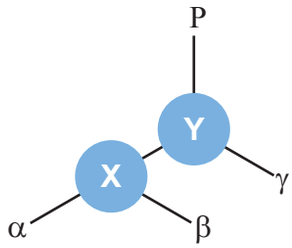
1. If y has a left subtree, assign x as the parent of the left subtree of y.



2. If the parent of x is NULL, make y as the root of the tree.
3. Else if x is the left child of p, make y as the left child of p.
4. Else assign y as the right child of p.



Make y as the parent of x.

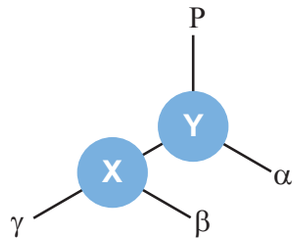


Right Rotate

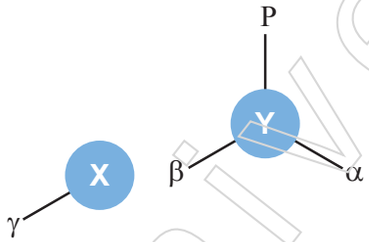
Notes

In left-rotation, the arrangement of the nodes on the left is transformed into the arrangements on the right node.

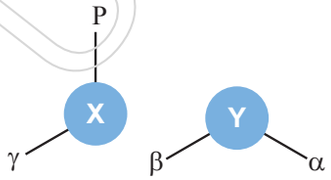
1. Let the initial tree be:



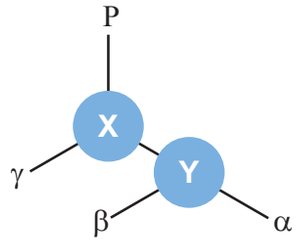
2. If x has a right subtree, assign y as the parent of the right subtree of x.



3. If the parent of y is NULL, make x as the root of the tree.
4. Else if y is the right child of its parent p, make x as the right child of p.
5. Else assign x as the left child of p.



6. Make x as the parent of y.



4.3.3 Threaded Tree

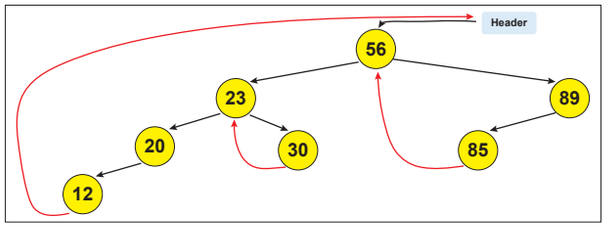
We know that the binary tree nodes may have at most two children. But if they have only one children, or no children, the link part in the linked list representation remains null. Using threaded binary tree representation, we can reuse that empty links by making some threads.

If one node has some vacant left or right child area, that will be used as thread. There are two types of threaded binary tree. The single threaded tree or fully threaded binary tree. In single threaded mode, there are another two variations. Left threaded and right threaded.

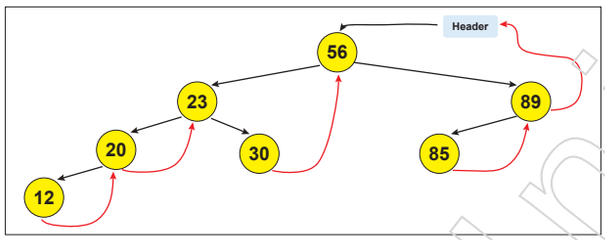
Notes

In the left threaded mode if some node has no left child, then the left pointer will point to its in-order predecessor, similarly in the right threaded mode if some node has no right child, then the right pointer will point to its in-order successor.

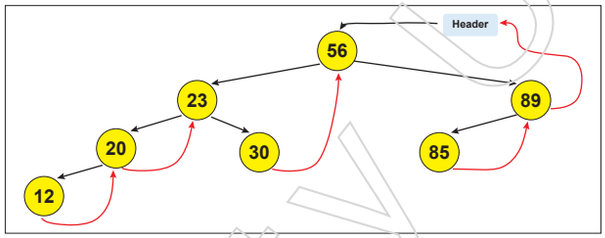
For fully threaded binary tree, each node has five fields. Three fields like normal binary tree node, another two fields to store Boolean value to denote whether link of that side is actual link or thread.



These are the examples of left and right threaded tree



This is the fully threaded binary tree



4.3.4 B Trees

A B-tree is a tree data structure that keeps data sorted and allows searches, insertions, and deletions in logarithmic amortized time. Unlike self-balancing binary search trees, it is optimized for systems that read and write large blocks of data. It is most commonly used in database and file systems.

- B-tree nodes have many more than two children.
- A B-tree node may contain more than just a single element.

The set formulation of the B-tree rules: Every B-tree depends on a positive constant integer called MINIMUM, which is used to determine how many elements are held in a single node.

- Rule 1: The root can have as few as one element (or even no elements if it also has no children); every other node has at least MINIMUM elements.
- Rule 2: The maximum number of elements in a node is twice the value of MINIMUM.
- Rule 3: The elements of each B-tree node are stored in a partially filled array, sorted from the smallest element (at index 0) to the largest element (at the

Notes

final used position of the array).

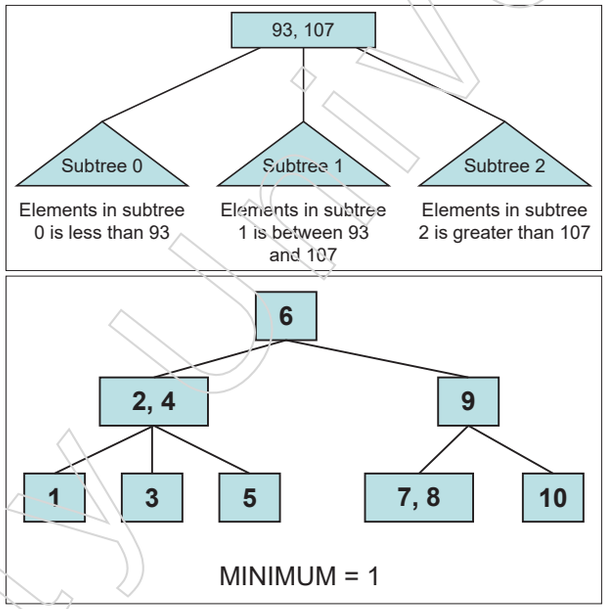
- Rule 4: The number of subtrees below a nonleaf node is always one more than the number of elements in the node.

o Subtree 0, subtree 1, ...

- Rule 5: For any nonleaf node:

  1. An element at index i is greater than all the elements in subtree number i of the node, and
  2. An element at index i is less than all the elements in subtree number i + 1 of the node.

- Rule 6: Every leaf in a B-tree has the same depth. Thus it ensures that a B-tree avoids the problem of a unbalanced tree.



Unit-4.4: Graph

Notes

4.4.1 Introduction

A Graph is a non-linear data structure containing of nodes and edges. The nodes are occasionally also mentioned to as vertices and the edges are lines or arcs that connect any two nodes in the graph.

4.4.2 Graph Theory Terminology

A Graph comprises of a finite set of vertices(or nodes) and set of Edges which connect a pair of nodes. A graph G consists of two sets

- a finite, nonempty set of vertices V(G)
- a finite, possible empty set of edges E(G)
- G(V,E) represents a graph

An undirected graph is one in which the pair of vertices in a edge is unordered, (v0, v1) = (v1,v0)

A directed graph is one in which each edge is a directed pair of vertices, <v0, v1> != <v1,v0>

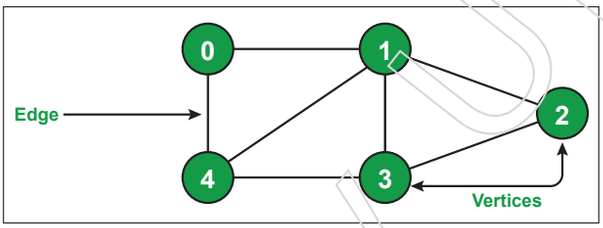


Fig 4.9 :Graph

In the above Graph, the set of vertices V = {0,1,2,3,4} and the set of edges E = {01, 12, 23, 34, 04, 14, 13}.

To represent networks Graphs are used. The networks may include paths in a city or telephone network or circuit network. Graphs are also utilized in social networking sites.

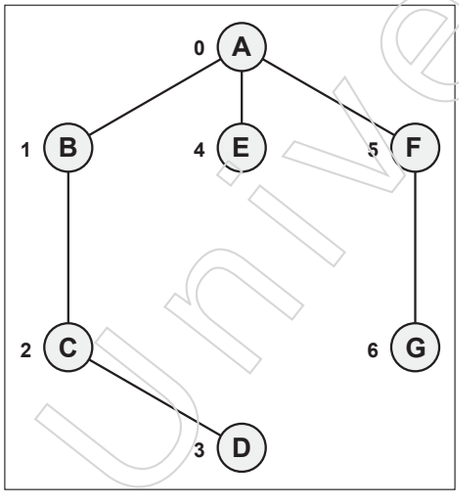
Mathematical graphs can be characterized in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we continue further, let's disseminate ourselves with some important terms:

- **Vertex −** Each node of the graph is represented as a vertex. In the following example, the labelled circle represents vertices. Thus, A to G are vertices. We can signify them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.
- **Edge −** Edge symbolizes a path between two vertices or a line between two

Notes

vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.

- **Adjacency −** Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.
- **Path −** Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.



Basic Operations

Following are basic primary operations of a Graph :

- Add Vertex − Adds a vertex to the graph.
- Add Edge − Adds an edge between the two vertices of the graph.
- Display Vertex − Displays a vertex of the graph.

4.4.3 Sequential Representation of Graph (Adjacency and Path Matrix)

The following two are the most commonly used representations of a graph.

1. Adjacency Matrix
2. Adjacency List

Let us take an undirected graph:

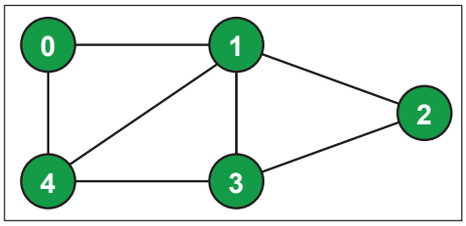


Fig 4.10 : undirected graph

1. Adjacency Matrix:

Adjacency Matrix is a 2D array of size Vertex x Vertex.

- Let G=(V,E) be a graph with n vertices.
- The adjacency matrix of G is a two-dimensional n by n array, say adj_mat
- If the edge (vi, vj) is in E(G), adj_mat[i][j]=1
- If there is no such edge in E(G), adj_mat[i][j]=0
- The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph need not be symmetric.

Adjacency Matrix is a 2D array of size Vertex x Vertex. Let the 2D array be adj_mat[][], a slot adj_mat[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj_mat[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

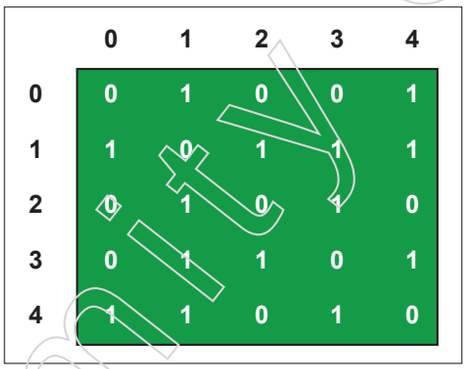


Fig 4.11 : Matrix

2. Adjacency List

For this, an array of lists is used. The size of the array is equal to the number of vertices. Let the array be arr[] and arr[i] represents the list of vertices adjacent to the ith vertex. To represent a weighted graph, the same representation can also be used. The weights of edges can be represented as lists of pairs. Following is the adjacency list representation of the above graph.

Notes

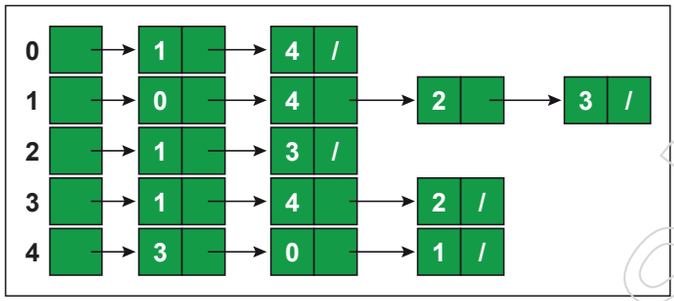


Fig 4.12 : Adjacency list

Data Structure for Adjacency List:

```
#define MAX_VERTICES 50
typedef struct node *node_pointer;
typedef struct node {
   int vertex;
   struct node *link;
};
node_pointer graph[MAX_VERTICES];
int n=0; /* vertices currently in use */
```

4.4.4 Warshall Algorithm

To find solution of all path shortest path problem from a given weighted graph, Floyd-Warshall algorithm is used. The algorithm will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph.

Example:

Input: graph[][] = { {0, 5, INF, 10},
{INF, 0, 3, INF},
{INF, INF, 0, 1},
{INF, INF, INF, 0} }

which represents the following graph

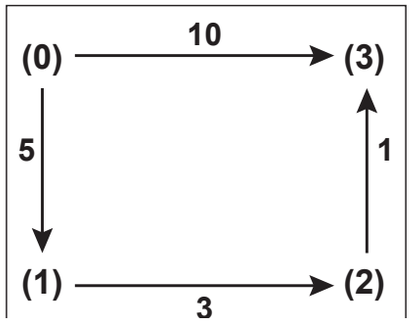


Fig 4.13 : Directed graph

Note that the value of graph[i][j] is 0 if i is equal to j

And graph[i][j] is INF (infinite) if there is no edge from vertex i to j.

Output:

Shortest distance matrix

0	5	8	9
INF	0	3	4
INF	INF	0	1
INF	INF	INF	0

Algorithm:

We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices. For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.

1. k is not an intermediate vertex in shortest path from i to j. We keep the value of dis[i][j] as it is.
2. k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j] if dist[i][j] > dist[i][k] + dist[k][j]

The following figure shows the above optimal substructure property in the all-pairs shortest path problem.

4.4.5 Linked Representation of Graph

In the linked representation, to store the Graph into the computer's memory, an adjacency list is used. Let us take the undirected graph shown in the following figure and check the adjacency list representation.

Notes

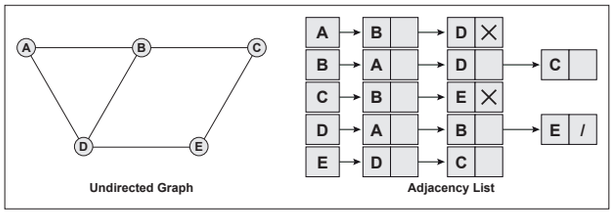


Fig 4.14 : Adjacency list

For each node present in the graph which stores the node value and a pointer to the next adjacent node to the respective node, an adjacency list is maintained. If all the adjacent nodes are traversed then store the NULL in the pointer field of last node of the list. The sum of the lengths of adjacency lists is equal to the twice of the number of edges present in an undirected graph.

Let us take the directed graph shown in the following figure and check the adjacency list representation of the graph. The sum of lengths of all the adjacency lists is equal to the number of edges present in the graph in case of a directed graph.

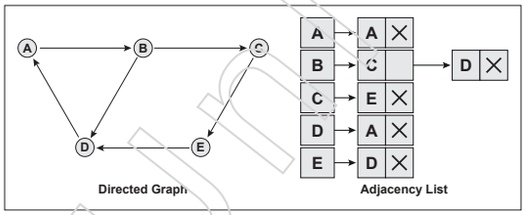


Fig 4.15 : Adjacency list

In the case of weighted directed graph, each node contains an extra field that is called the weight of the node. The adjacency list representation of a directed graph is shown in the following figure.

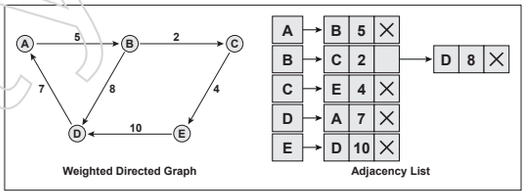


Fig 4.16 : Adjacency list

4.4.6 Different Operations on Graphs

Basic Operations

Following are basic primary operations of a Graph −

- **Add/ Remove Vertex:** This is the simplest operation in the graph. We can add a vertex to a graph. It need not be connected to any other vertex through an edge. When removing a vertex, we must remove all edges originating from and ending at the deleted vertex.
- **Add/ Remove Edge:** This operation adds or removes an edge between two vertices. When all the edges originating from and ending at a vertex are deleted, the vertex becomes an isolated vertex.

Notes

- **Display Vertex:** Displays a vertex of the graph.

4.4.7 Traversing A Graph(DFS, BFS)

DFS

Depth First Search or DFS traverses the graph vertically. It starts with the root node or the first node of the graph and traverses all the child nodes before moving to the adjacent nodes.

Example

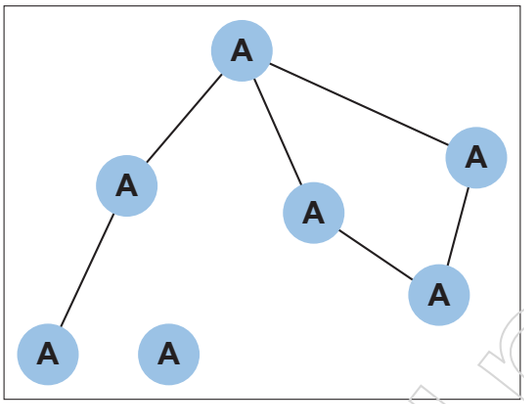


Fig 4.17 : Graph

If we traverse the above graph in DFS fashion then we get result as A -> B -> E -> F -> C -> G -> D. The algorithm starts from node A and traverses all its child nodes. As soon as it encounters B, it seems that it has further child nodes. So, the child nodes of B are traversed before proceeding to the next child node of A.

BFS

Traversal operation in the graph done by this. BFS horizontally traverses the graph which means that it traverses all the nodes at a single level before proceeding to the next level. The BFS algorithm starts at the top of the first node in the graph and then traverses all the adjacent nodes to it. Once all the adjacent nodes are traversed, the algorithm repeats the same procedure for child nodes.

If we traverse the above graph in BFS fashion then we will get as A -> B -> C -> D -> E -> F -> G. The algorithm starts from node A and traverses all its adjacent nodes B, C and D. It marks all the four nodes as visited. Once all the adjacent nodes of A are traversed, the algorithm moves to the first adjacent node of A and repeats the same procedure. In this case, the node is B, and the adjacent nodes to B are E and F. Next, the adjacent nodes to C are traversed. Once all the nodes are visited, the operation ends.

Example:

```
#include <stdio.h>

#include <stdlib.h>

// Define the maximum number of vertices in the graph
```

Notes

```
#define N 6
 struct Graph {
    // An array of pointers to Node to represent an adjacency list
    struct Node* head[N];
};
// to store adjacency list nodes of the graph
struct Node {
    int dest;
    struct Node* next;
};
// to store a graph edge
struct Edge {
    int src, dest;
};
// Function to create an adjacency list from specified edges
struct Graph* createGraph(struct Edge edges[], int n)
{
    unsigned i;
 // allocate storage for the graph data structure
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
 // initialize head pointer for all vertices
    for (i = 0; i < N; i++) {
        graph->head[i] = NULL;
    }
 // add edges to the directed graph one by one
    for (i = 0; i < n; i++)
    {
  // get the source and destination vertex
        int src = edges[i].src;
        int dest = edges[i].dest;
   // allocate a new node of adjacency list from `src` to `dest`
        struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
```

Notes

```
        newNode->dest = dest;
      // point new node to the current head
        newNode->next = graph->head[src];
      // point head pointer to the new node
        graph->head[src] = newNode;
      }
       return graph;
}
 // Function to print adjacency list representation of a graph
void printGraph(struct Graph* graph)
{
    int i;
    for (i = 0; i < N; i++)
    {
        // print current vertex and all its neighbors
        struct Node* ptr = graph->head[i];
        while (ptr != NULL)
        {
            printf("(%d -> %d)\t", i, ptr->dest);
            ptr = ptr->next;
        }
        printf("\n");
    }
}
int main(void)
{
      struct Edge edges[] =
    {
        { 0, 1 }, { 1, 2 }, { 2, 0 }, { 2, 1 },
        { 3, 2 }, { 4, 5 }, { 5, 4 }
    };
    // calculate the total number of edges
```

Notes

```
    int n = sizeof(edges)/sizeof(edges[0]);
    // construct a graph from the given edges
    struct Graph *graph = createGraph(edges, n);
    // Function to print adjacency list representation of a graph
    printGraph(graph);
    return 0;
}
```

Output:

```
(0 -> 1)
(1 -> 2)
(2 -> 1)        (2 -> 0)
(3 -> 2)
(4 -> 5)
(5 -> 4)
```

Unit-4.5: Spanning Trees

Notes

4.5.1 Introduction

When graph G is connected, a depth first or breadth first search starting at any vertex will visit all vertices in G. A spanning tree is any tree that consists solely of edges in G and that includes all the vertices.

E(G): T (tree edges) + N (nontree edges)

where, T: set of edges used during search

N: set of remaining edges

Either dfs or bfs can be used to create a spanning tree

- When dfs is used, the resulting spanning tree is known as a depth first spanning tree
- When bfs is used, the resulting spanning tree is known as a breadth first spanning tree
- While adding a nontree edge into any spanning tree, this will create a cycle

A spanning tree is a minimal subgraph, G', of G such that V(G')=V(G) and G' is connected.

Any connected graph with n vertices must have

at least n-1 edges.

4.5.2 Representation of Spanning tree

By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.

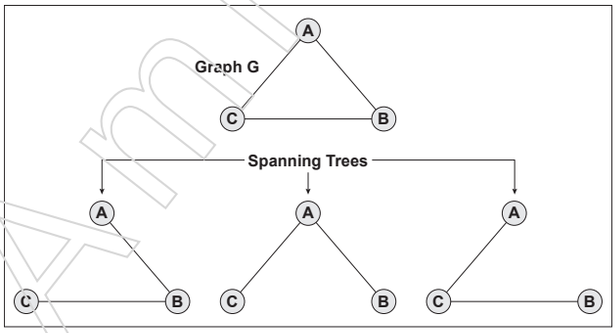


Fig 4.18 : Spanning tree

4.5.3 Constructing a Spanning Tree

We now recognize that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G:

Notes

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Eliminating one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is minimally connected.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is maximally acyclic.

Spanning tree is mostly used to find a minimum path to connect all nodes in a graph. Common application of spanning trees are −

- Civil Network Planning
- Computer Network Routing Protocol
- Cluster Analysis

4.5.4 Prim's Minimum Spanning Tree

Prim's algorithm is a Greedy algorithm. It begins with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already contained in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

A group of edges that connects two set of vertices in a graph is called cut in graph theory. So, at every step of Prim's algorithm, we discover a cut (of two sets, one contains the vertices already included in MST and other contains rest of the vertices), collect the minimum weight edge from the cut and include this vertex to MST Set (the set that contains already included vertices).

The idea behind Prim's algorithm is simple, a spanning tree implies all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a Spanning Tree. And they must be connected with the minimum weight edge to make it a Minimum Spanning Tree.

Algorithm

1. Create a set mstSet that keeps track of vertices already included in MST.
2. Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.
3. While mstSet doesn't include all vertices
   a) Pick a vertex u which is not there in mstSet and has minimum key value.
   b) Include u to mstSet.
   c) Update key value of all adjacent vertices of u. To update the key values, iterate through all adjacent vertices. For every adjacent vertex v, if weight of edge u-v is less than the previous key value of v, update the key value as weight of u-v.

The concept of using key values is to pick the minimum weight edge from cut. The key values are utilized only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges linking them to the set of vertices included in MST.

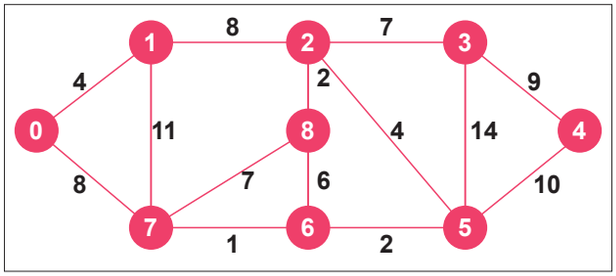
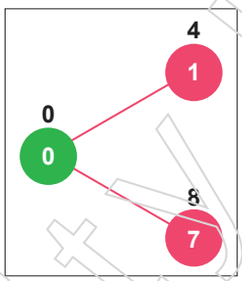


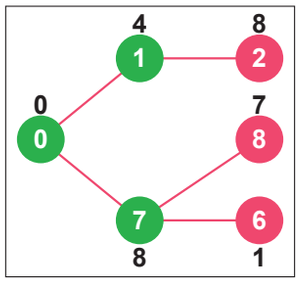
Fig 4.19 : MST

The set mstSet is firstly empty and keys assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with the minimum key value. The vertex 0 is picked, include it in mstSet. So mstSet becomes {0}. After including to mstSet, update key values of adjacent vertices. Adjacent vertices of 0 are 1 and 7. The key values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their key values, only the vertices with finite key values are shown. The vertices encompassed in MST are shown in green color.



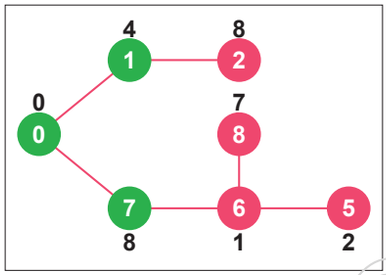
Select the vertex with minimum key value and not already included in MST (not in mstSET). The vertex 1 is picked and added to mstSet. So mstSet now becomes {0, 1}. Update the key values of adjacent vertices of 1. The key value of vertex 2 becomes 8.

Pick the vertex with minimum key value and not even now included in MST (not in mstSET). We can either pick vertex 7 or vertex 2, let vertex 7 is picked. So mstSet now becomes {0, 1, 7}. Update the key values of adjacent vertices of 7. The key value of vertex 6 and 8 becomes finite (1 and 7 respectively).

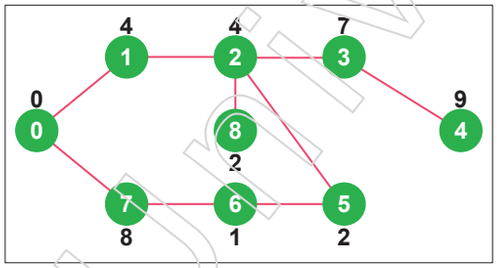


Notes

Pick the vertex with minimum key value and not already included in MST (not in mstSET). Vertex 6 is picked. So mstSet now becomes {0, 1, 7, 6}. Update the key values of adjacent vertices of 6. The key value of vertex 5 and 8 are updated.



We replicate the above steps until mstSet contains all vertices of given graph. Finally, we get the following graph.



Example:

```
#include <limits.h>
#include <stdbool.h>
#include <stdio.h>
#define V 5   // Number of vertices in the graph
// function to find the vertex with minimum key value, from the set of vertices not yet included in MST
int minKey(int key[], bool mstSet[])
{
    int min = INT_MAX, min_index,v;
   for ( v = 0; v < V; v++)
      if (mstSet[v] == false && key[v] < min)
         min = key[v], min_index = v;
   return min_index;
}
// function to print the constructed MST stored in parent[]
int printMST(int parent[], int graph[V][V])
```

Notes

```
{
   int i;
    printf("Edge \tWeight\n");
    for ( i = 1; i < V; i++)
        printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
}
/*function to construct and print MST for a graph represented using adjacency matrix representation */
void primMST(int graph[V][V])
{
    int parent[V], i, count;
    // Key values used to pick minimum weight edge in cut
    int key[V], v;
    // To represent set of vertices not yet included in MST
    bool mstSet[V];
    for ( i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;
 /*always include first 1st vertex in MST.Make key 0 so that this vertex is picked as first vertex. */
    key[0] = 0;
// First node is always root of MST
    parent[0] = -1;
    for ( count = 0; count < V - 1; count++) {
/*pick the minimum key vertex from the set of vertices not yet included in MST */
        int u = minKey(key, mstSet);
        // Add the picked vertex to the MST Set
        mstSet[u] = true;
  /*Update key value and parent index of the adjacent vertices of the picked vertex. Consider only those vertices which are not yet included in MST */
        for ( v = 0; v < V; v++)
          if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
             parent[v] = u, key[v] = graph[u][v];
    }
```

Notes

```
    printMST(parent, graph);
}
void main()
{
    int graph[V][V] = { { 0, 2, 0, 6, 0 },
                        { 2, 0, 3, 8, 5 },
                        { 0, 3, 0, 0, 7 },
                        { 6, 8, 0, 0, 9 },
                        { 0, 5, 7, 9, 0 } };
    primMST(graph);
}
```

Output:.

Edge Weight

0 - 1 2

1 - 2 3

0 - 3 6

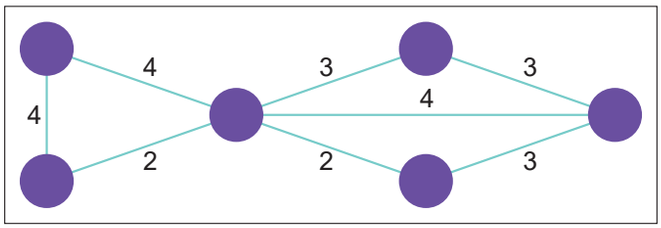
1 - 4 5

4.5.5 Kruskal's Algorithm

Steps:

1. Build a minimum cost spanning tree T by adding edges to T one at a time.
2. Select the edges for inclusion in T in non decreasing order of the cost
3. An edge is added to T if it does not form a cycle
4. Since G is connected and has n>0 vertices, exactly n-1 edges will be selected

Example



Algorithm

This algorithm will create spanning tree with minimum weight from a given weighted graph.

Notes

1. Begin
2. Create the edge list of given graph, with their weights.
3. Sort the edge list according to their weights in ascending order.
4. Draw all the nodes to create skeleton for spanning tree.
5. Pick up the edge at the top of the edge list (i.e. edge with minimum weight).
6. Remove this edge from the edge list.
7. Connect the vertices in the skeleton with given edge. If by connecting the vertices, a cycle is created in the skeleton, then discard this edge.
8. Repeat steps 5 to 7, until n-1 edges are added or list of edges is over.
9. Return

Module End Questions

1. The number of edges from the root to the node is called _____ of the tree.
   a) Height
   b) Depth
   c) Length
   d) Width
2. The number of edges from the node to the deepest leaf is called _____ of the tree.
   a) Height
   b) Depth
   c) Length
   d) Width
3. What is a full binary tree?
   a) Each node has exactly zero or two children
   b) Each node has exactly two children
   c) All the leaves are at the same level
   d) Each node has exactly one or two children
4. What is a complete binary tree?
   a) Each node has exactly zero or two children
   b) A binary tree, which is completely filled, with the possible exception of the bottom level, which is filled from right to left
   c) A binary tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right
   d) A tree In which all nodes have degree 2

Notes

5. Which of the following is not an advantage of trees?

   a) Hierarchical structure

   b) Faster search

   c) Router algorithms

   d) Undo/Redo operations in a notepad

6. Which of the following is incorrect with respect to binary trees?

   a) Let T be a binary tree. For every k ≥ 0, there are no more than 2k nodes in level k

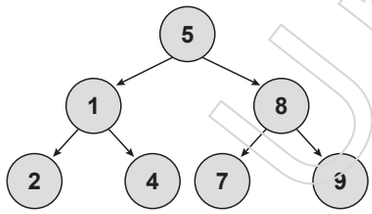
   b) Let T be a binary tree with λ levels. Then T has no more than 2λ – 1 nodes

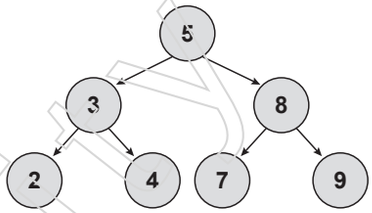
   c) Let T be a binary tree with N nodes. Then the number of levels is at least ceil(log (N + 1))

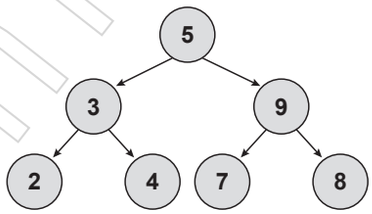
   d) Let T be a binary tree with N nodes. Then the number of levels is at least floor(log (N + 1))

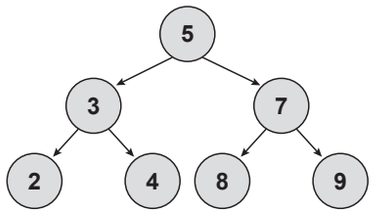
7. Construct a binary search tree by using postorder sequence given below.

   Postorder: 2, 4, 3, 7, 9, 8, 5.

   a)


   b)


   c)


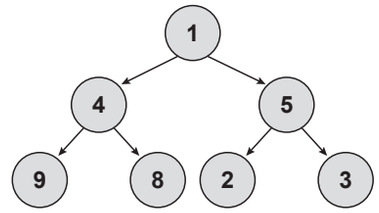
   d)


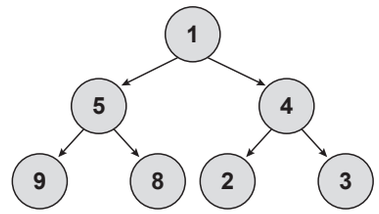
8. Construct a binary tree using inorder and level order traversal given below.

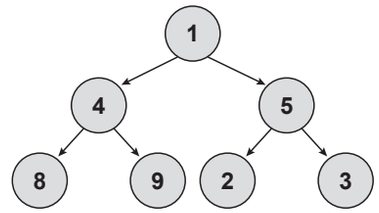
   Inorder Traversal: 3, 4, 2, 1, 5, 8, 9

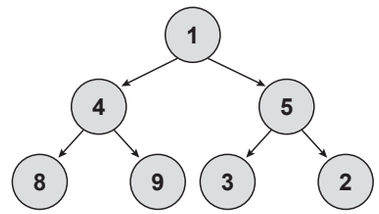
   Level Order Traversal: 1, 4, 5, 9, 8, 2, 3

Notes

a) 

b) 

c) 

d) 

9. What is an AVL tree?

   a) a tree which is balanced and is a height balanced tree

   b) a tree which is unbalanced and is a height balanced tree

   c) a tree with three children

   d) a tree with atmost 3 children

10. Given an empty AVL tree, how would you construct AVL tree when a set of numbers are given without performing any rotations?

    a) just build the tree with the given input

    b) find the median of the set of elements given, make it as root and construct the tree

    c) use trial and error

    d) use dynamic programming to build the tree

Answer key:

1. b, 2. a, 3. a, 4. c, 5. d, 6. d, 7. b, 8. a, 9. a, 10. b

Notes

Module-5: Searching and Sorting Techniques

Structure:

Unit-5.1: Sorting Techniques

Notes

5.1.1 Insertion Sort

Sorting describes organizing data in a particular format. Sorting algorithm stipulates the way to arrange data in a certain order. Most widespread orders are in numerical or lexicographical order. A Sorting Algorithm is applied to rearrange a given array or list elements allowing to a comparison operator on the elements.

Insertion sort is a uncomplicated sorting algorithm that operates similar to the way you sort playing cards in your hands. The array is virtually separated into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the proper position in the sorted part.

Algorithm

To sort an array of size n (in ascending order)

1. Iterate from arr[1] to arr[n] over the array.
2. Compare the current element or key to its predecessor.
3. If the key element is lesser than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Example:

12, 11, 13, 5, 6

Let us loop for i = 1 (second element of the array) to 4 (last element of the array)

i = 1. Since 11 is smaller than 12, move 12 and insert 11 before 12

11, 12, 13, 5, 6

i = 2. 13 will remain at its position as all elements in A[0..I-1] are smaller than 13

11, 12, 13, 5, 6

i = 3. 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.

5, 11, 12, 13, 6

i = 4. 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.

5, 6, 11, 12, 13

Example:

```
/* Insertion sort ascending order */
#include <stdio.h>
int main()
{
```

Notes

```
int n, array[1000], c, d, t, flag = 0;
printf("Enter number of elements: \n");
scanf("%d", &n);
printf("Enter %d integers: \n", n);
for (c = 0; c < n; c++)
  scanf("%d", &array[c]);
for (c = 1 ; c <= n - 1; c++) {
  t = array[c];
  for (d = c - 1 ; d >= 0; d--) {
    if (array[d] > t) {
      array[d+1] = array[d];
      flag = 1;
    }
    else
      break;
  }
  if (flag)
    array[d+1] = t;
}
printf("Sorted list in ascending order: \n");
for (c = 0; c <= n - 1; c++) {
  printf("%d\n", array[c]);
}
return 0;
}
```

Output:

Enter number of elements:

5

Enter 5 integers:

1 2 5 3 4

Sorted list in ascending order:

1

Notes

2

3

4

5

5.1.2 Bubble Sort

Bubble Sort is the plainest sorting algorithm that operates by constantly swapping the adjacent elements if they are in wrong order.

Example:

First pass:

(5 1 4 2 8) –> (1 5 4 2 8), here, algorithm compares the first two elements, and swaps since 5 > 1.

(1 5 4 2 8) –> (1 4 5 2 8), Swap since 5 > 4

(1 4 5 2 8) –> (1 4 2 5 8), Swap since 5 > 2

(1 4 2 5 8) –> (1 4 2 5 8), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

(1 4 2 5 8) –> (1 4 2 5 8)

(1 4 2 5 8) –> (1 2 4 5 8), Swap since 4 > 2

(1 2 4 5 8) –> (1 2 4 5 8)

(1 2 4 5 8) –> (1 2 4 5 8)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one whole pass without any swap to know it is sorted.

Third Pass:

(1 2 4 5 8) –> (1 2 4 5 8)

(1 2 4 5 8) –> (1 2 4 5 8)

(1 2 4 5 8) –> (1 2 4 5 8)

(1 2 4 5 8) –> (1 2 4 5 8)

Example:

```
/* C program for implementation of Bubble sort */

#include <stdio.h>

int main()

{

  int array[100], n, c, d, swap;
```

Notes

```
  printf("Enter number of elements\n");
  scanf("%d", &n);
  printf("Enter %d integers\n", n);
  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);
  for (c = 0 ; c < n - 1; c++)
  {
    for (d = 0 ; d < n - c - 1; d++)
    {
      if (array[d] > array[d+1])
      {
        swap       = array[d];
        array[d]   = array[d+1];
        array[d+1] = swap;
      }
    }
  }
  printf("Sorted list in ascending order:\n");
  for (c = 0; c < n; c++)
     printf("%d\n", array[c]);
  return 0;
}
```

Output:

```
Enter number of elements
5
Enter 5 integers
20 30 50 40 70
Sorted list in ascending order:
20
30
40
50
```

70

Notes

5.1.3 Selection Sort

The selection sort algorithm arranges an array by continually finding the minimum element (considering ascending order) from unsorted part and placing it at the beginning. The algorithm holds two subarrays in a given array.

1. The subarray which is already sorted.
2. Remaining subarray which is unsorted.

In every iteration of selection sort, the least element (considering ascending order) from the unsorted subarray is selected and moved to the sorted subarray.

Explanations of the above steps:

arr[] = 64 25 12 22 11

// Find the minimum element in arr[0...4]

// and place it at beginning

11 25 12 22 64

// Find the minimum element in arr[1...4]

// and place it at beginning of arr[1...4]

11 12 25 22 64

// Find the minimum element in arr[2...4]

// and place it at beginning of arr[2...4]

11 12 22 25 64

// Find the minimum element in arr[3...4]

// and place it at beginning of arr[3...4]

11 12 22 25 64

Example:

```
/* C program for implementation of selection sort */
#include <stdio.h>
int main()
{
  int array[100], n, c, d, position, t;
  printf(“Enter number of elements\n”);
  scanf(“%d”, &n);
  printf(“Enter %d integers\n”, n);
  for (c = 0; c < n; c++)
```

Notes

```
    scanf("%d", &array[c]);
  for (c = 0; c < (n - 1); c++) // finding minimum element (n-1) times
  {
    position = c;
    for (d = c + 1; d < n; d++)
    {
      if (array[position] > array[d])
        position = d;
    }
    if (position != c)
    {
      t = array[c];
      array[c] = array[position];
      array[position] = t;
    }
  }
  printf("Sorted list in ascending order:\n");
  for (c = 0; c < n; c++)
    printf("%d\n", array[c]);
  return 0;
}
```

Output:

```
Enter number of elements
5
Enter 5 integers
9 4 1 2 6
Sorted list in ascending order:
1
2
4
6
9
```

Notes

5.1.4 Quick Sort

QuickSort is a Divide and Conquer algorithm. It chooses an element as pivot and partitions the given array around the picked pivot. There are many distinct versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
   if (low < high)
   {
      /* pi is partitioning index, arr[pi] is now
         at right place */
      pi = partition(arr, low, high);
      quickSort(arr, low, pi - 1);  // Before pi
      quickSort(arr, pi + 1, high); // After pi
   }
}
```

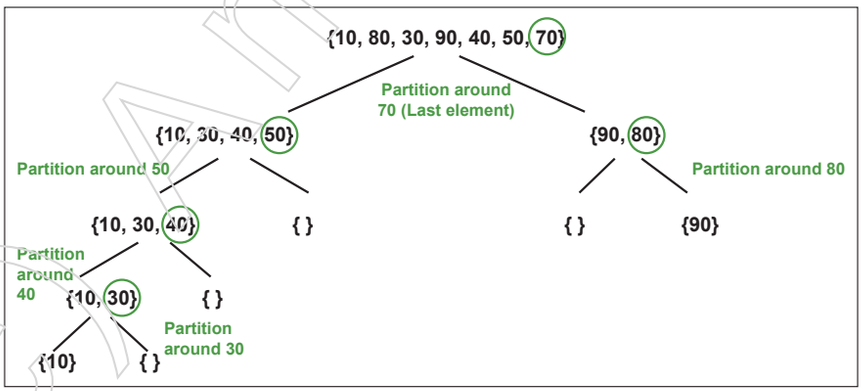


Fig 5.1 : Quick sort

Example:

Notes

```
/* C implementation of QuickSort */
#include <stdio.h>
#include <stdbool.h>
#define MAX 7
int intArray[MAX] = {4,6,3,2,1,9,7};
void printline(int count) {
   int i;
   for(i = 0;i < count-1;i++) {
      printf("=");
   }
   printf("=\n");
}
void display() {
   int i;
   printf("[");

   // navigate through all items
   for(i = 0;i < MAX;i++) {
      printf("%d ",intArray[i]);
   }
   printf("]\n");
}
void swap(int num1, int num2) {
   int temp = intArray[num1];
   intArray[num1] = intArray[num2];
   intArray[num2] = temp;
}
int partition(int left, int right, int pivot) {
   int leftPointer = left -1;
   int rightPointer = right;
   while(true) {
      while(intArray[++leftPointer] < pivot) {
```

Notes

```
      //do nothing
    }
  while(rightPointer > 0 && intArray[--rightPointer] > pivot) {
      //do nothing
    }
    if(leftPointer >= rightPointer) {
      break;
    } else {
      printf(" item swapped :%d,%d\n", intArray[leftPointer],intArray[rightPointer]);
      swap(leftPointer,rightPointer);
    }
  }
 printf(" pivot swapped :%d,%d\n", intArray[leftPointer],intArray[right]);
  swap(leftPointer,right);
  printf("Updated Array: ");
  display();
  return leftPointer;
}
void quickSort(int left, int right) {
  if(right-left <= 0) {
    return;
  } else {
    int pivot = intArray[right];
    int partitionPoint = partition(left, right, pivot);
    quickSort(left,partitionPoint-1);
    quickSort(partitionPoint+1,right);
  }
}
int main() {
  printf("Input Array: ");
  display();
  printline(50);
```

Notes

```
   quickSort(0,MAX-1);
   printf("Output Array: ");
   display();
   printline(50);
}
```

If we compile and run the above program, it will produce the following result :

Output:

```
Input Array: [4 6 3 2 1 9 7 ]
pivot swapped :9,7
Updated Array: [4 6 3 2 1 7 9 ]
 pivot swapped :4,1
Updated Array: [1 6 3 2 4 7 9 ]
 item swapped :6,2
 pivot swapped :6,4
Updated Array: [1 2 3 4 6 7 9 ]
 pivot swapped :3,3
Updated Array: [1 2 3 4 6 7 9 ]
Output Array: [1 2 3 4 6 7 9 ]
```

5.1.5 Merge Sort

Merge Sort is a Divide and Conquer algorithm. It divide up the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. The merge() function is used for combining two halves. The merge(arr, l, m, r) is a key process that undertakes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

MergeSort(arr[], l, r)

If r > l

1. Find the middle point to divide the array into two halves:

   middle m = (l+r)/2

2. Call mergeSort for first half:

   Call mergeSort(arr, l, m)

3. Call mergeSort for second half:

   Call mergeSort(arr, m+1, r)

4. Merge the two halves sorted in step 2 and 3:

Notes

Call merge(arr, l, m, r)

The following diagram from wikipedia shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes come into action and start merging arrays back till the complete array is merged.

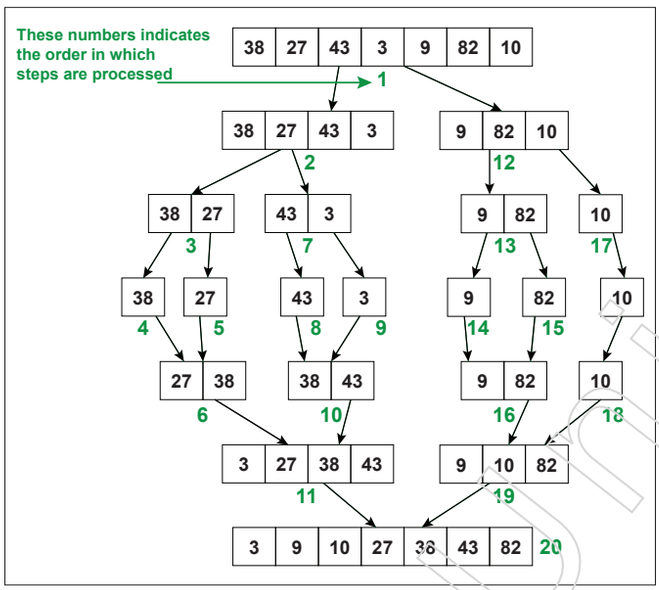


Fig 5.2 : Merge sort

Example:

```
// C program for Merge Sort
#include <stdio.h>
#define max 10
int a[11] = { 10, 14, 19, 26, 27, 31, 33, 35, 42, 44, 0 };
int b[10];
void merging(int low, int mid, int high) {
  int l1, l2, i;
  for(l1 = low, l2 = mid + 1, i = low; l1 <= mid && l2 <= high; i++) {
    if(a[l1] <= a[l2])
      b[i] = a[l1++];
    else
      b[i] = a[l2++];
  }
  while(l1 <= mid)
```

Notes

```
      b[i++] = a[l1++];
   while(l2 <= high)
      b[i++] = a[l2++];
   for(i = low; i <= high; i++)
      a[i] = b[i];
}
void sort(int low, int high) {
   int mid;
   if(low < high) {
      mid = (low + high) / 2;
      sort(low, mid);
      sort(mid+1, high);
      merging(low, mid, high);
   } else {
      return;
   }
}
int main() {
   int i;
   printf("List before sorting\n");
   for(i = 0; i <= max; i++)
      printf("%d ", a[i]);
   sort(0, max);
   printf("\nList after sorting\n");
   for(i = 0; i <= max; i++)
      printf("%d ", a[i]);
}
```

If we compile and run the above program, it will produce the following result :

Output

```
List before sorting
10 14 19 26 27 31 33 35 42 44 0
List after sorting
```

0 10 14 19 26 27 31 33 35 42 44

Notes

5.1.6 Heap sort

Heap sort is a comparison-based sorting procedure based on Binary Heap data structure. It is analogous to selection sort where we first find the maximum element and place the maximum element at the end. We replicate the same process for the remaining elements. Let us first define a Complete Binary Tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible. A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is larger (or smaller) than the values in its two children nodes. The previous is called as max heap and the concluding is called min-heap. The heap can be exemplified by a binary tree or array.

Heap Sort Algorithm for sorting in increasing order:

1. Build up a next head from the input data.
2. At this point, the biggest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of the tree.
3. Repeat step 2 while size of heap is greater than 1.

Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom-up order.

Lets understand with the help of an example:

Input data: 4, 10, 3, 5, 1

```
        4(0)
       /   \
    10(1)   3(2)
   /   \
 5(3)   1(4)
```

The numbers in bracket represent the indices in the array representation of data.

Applying heapify procedure to index 1:

```
        4(0)
       /   \
    10(1)   3(2)
   /   \
 5(3)   1(4)
```

Applying heapify procedure to index 0:

```
        10(0)
       /   \
```

Notes

5(1) 3(2)

/ \

4(3) 1(4)

The heapify procedure calls itself recursively to build heap in top down manner.

Example:

```
/*C program for implementation of Heap Sort*/
#include<stdio.h>
int temp;
 void heapify(int arr[], int size, int i)
{
int largest = i;
int left = 2*i + 1;
int right = 2*i + 2;
 if (left < size && arr[left] >arr[largest])
largest = left;
 if (right < size && arr[right] > arr[largest])
largest = right;
if (largest != i)
{
temp = arr[i];
   arr[i]= arr[largest];
   arr[largest] = temp;
heapify(arr, size, largest);
}
}
 void heapSort(int arr[], int size)
{
int i;
for (i = size / 2 - 1; i >= 0; i--)
heapify(arr, size, i);
for (i=size-1; i>=0; i--)
```

Notes

```
{
temp = arr[0];
   arr[0]= arr[i];
   arr[i] = temp;
heapify(arr, i, 0);
}
}
void main()
{
int arr[] = {1, 10, 2, 3, 4, 1, 2, 100,23, 2};
int i;
int size = sizeof(arr)/sizeof(arr[0]);
heapSort(arr, size);
printf("printing sorted elements\n");
for (i=0; i<size; ++i)
printf("%d\n",arr[i]);
}
```

Output:

```
printing sorted elements
1
1
2
2
2
3
4
10
23
100
```

5.1.7 Partition Exchange Sort

Quick Sort is also established on the concept of Divide and Conquer, just like merge sort. But in quick sort all the intense lifting(major work) is done while dividing the

Notes

array into subarrays, while in case of merge sort, all the genuine work occurs during merging the subarrays. In case of quick sort, the blend step does utterly nothing. It is similarly called partition-exchange sort. This algorithm splits the list into three main parts:

1. Elements less than the Pivot element
2. Pivot element(Central element)
3. Elements greater than the pivot element

Pivot element can be any element from the array, it can be the first element, the last element or any random element. In this tutorial, we will take the rightmost element or the last element as pivot.

For example: In the array {52, 37, 63, 14, 17, 8, 6, 25}, we take 25 as pivot. So after the first pass, the list will be changed like this:

{6 8 17 14 25 63 37 52}

Hence after the first pass, pivot will be set at its position, with all the elements smaller to it on its left and all the elements larger than to its right. Now 6 8 17 14 and 63 37 52 are considered as two separate subarrays, and same recursive logic will be applied on them, and we will keep doing this until the complete array is sorted.

Following are the steps involved in quick sort algorithm:

1. After choosing an element as pivot, which is the last index of the array in our case, we split up the array for the first time.
2. In quick sort, we call this partitioning. It is not simple breaking down of array into 2 subarrays, but in case of partitioning, the array elements are so positioned that all the elements smaller than the pivot will be on the left side of the pivot and all the elements greater than the pivot will be on the right side of it.
3. And the pivot element will be at its final sorted position.
4. The elements to the left and right, may not be sorted.
5. Then we pick subarrays, elements on the left of pivot and elements on the right of pivot, and we perform partitioning on them by choosing a pivot in the subarrays.

Let's consider an array with values {9, 7, 5, 11, 12, 2, 14, 3, 10, 6}

Below, we have a pictorial representation of how quick sort will sort the given array.

Notes

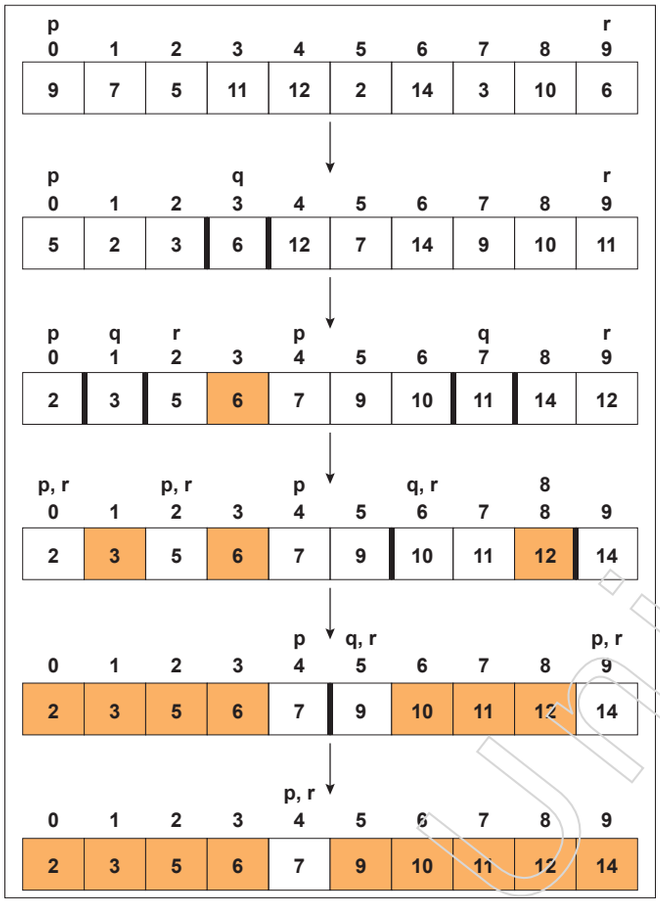


Fig 5.3 : Partition exchange sort

Example:

```
#include <stdio.h>
int partition(int a[], int beg, int end);
void quickSort(int a[], int beg, int end);
int main()
{
   int i;
   int arr[10]={90,23,101,45,65,28,67,89,34,29};
   quickSort(arr, 0, 9);
   printf("\n The sorted array is: \n");
   for(i=0;i<10;i++)
   printf(" %d\t", arr[i]);
}
int partition(int a[], int beg, int end)
```

Notes

```
{
    int left, right, temp, loc, flag;
    loc = left = beg;
    right = end;
    flag = 0;
    while(flag != 1)
    {
        while((a[loc] <= a[right]) && (loc!=right))
        right--;
        if(loc==right)
        flag =1;
        else if(a[loc]>a[right])
        {
            temp = a[loc];
            a[loc] = a[right];
            a[right] = temp;
            loc = right;
        }
        if(flag!=1)
        {
            while((a[loc] >= a[left]) && (loc!=left))
            left++;
            if(loc==left)
            flag =1;
            else if(a[loc] < a[left])
            {
                temp = a[loc];
                a[loc] = a[left];
                a[left] = temp;
                loc = left;
            }
        }
    }
```

Notes

```
    return loc;
}
void quickSort(int a[], int beg, int end)
{
    int loc;
    if(beg<end)
    {
        loc = partition(a, beg, end);
        quickSort(a, beg, loc-1);
        quickSort(a, loc+1, end);
    }
}
```

Output:

The sorted array is:

23 28 29 34 45 65 67 89 90 101

5.1.8 Shell Sort

Shell sort is a highly useful sorting algorithm and is established on insertion sort algorithm. This algorithm precludes large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.

Knuth's Formula

h = h * 3 + 1

where, h is interval with initial value 1

Let us consider the subsequent example to come up with an idea of how shell sort works. We take over the same array we have used in our previous examples. For our example and ease of understanding, we take the period of 4. Make a virtual sub-list of all values located at the interval of 4 positions. Here these values are {35, 14}, {33, 19}, {42, 27} and {10, 44}

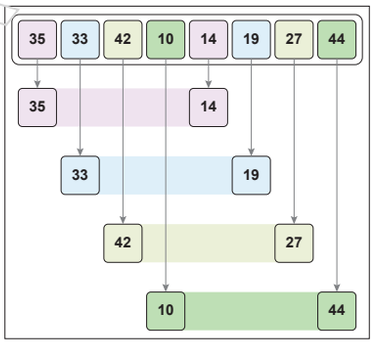
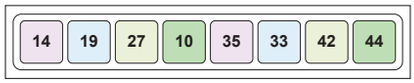


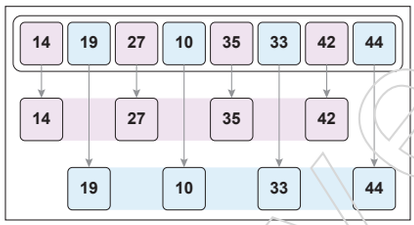
Fig 5.4 : Shell sort

Notes

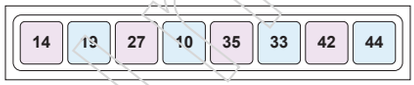
We compare values in each sub-list and swap them (if necessary) in the original array. After this step, the new array should look like this.



Then, we take interval of 1 and this gap generates two sub-lists - {14, 27, 35, 42}, {19, 10, 33, 44}



We compare and swap the values, if required, in the original array. After this step, the array should look like this



Ultimately, we sort the remainder of the array using interval of value 1. Shell sort uses insertion sort to sort the array. Following is the step-by-step depiction

We see that it obliged only four swaps to sort the rest of the array.

Algorithm:

Step 1 − Initialize the value of h

Step 2 − Divide the list into smaller sub-list of equal interval h

Step 3 − Sort these sub-lists using insertion sort

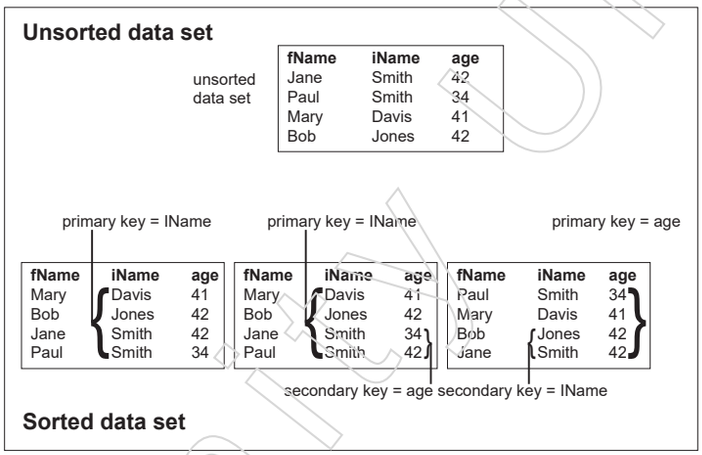
Step 3 − Repeat until complete list is sorted

5.1.9 Sorting on Different Keys

Sorting keys indicate the criteria used to execute the sort. The psort operator permits to set a primary sorting key and multiple secondary sorting keys.

The psort operator manipulates the sorting keys to determine the sorting order of a data set. The sort operator first sorts the records by the primary sorting key. If multiple records have the same primary key value, the psort operator then sorts these records by any secondary keys.

You must define a single primary sorting key for the psort operator. You might optionally define as many secondary keys as required by your job. Note, however, that each record field can be used only once as a sorting key. Therefore, the total number of primary and secondary sorting keys must be less than or equal to the total number of fields in the record.



This figure also indicates the results of three sorts using different combinations of sorting keys. In this figure, the IName field represents a string field and the age field represents an integer. By default, the psort operator applies a case-sensitive algorithm for sorting. This implies that uppercase strings be found before lowercase strings in a sorted data set. You can utilize an option to the psort operator to select case-insensitive sorting. You can use the member function APT_PartitionSortOperator::setKey() to override this default, to perform case-insensitive sorting on string fields.

By default, the psort operator APT_PartitionSortOperator uses ascending sort order, so that smaller values appear before larger values in the sorted data set. You can use an option to the psort operator to select descending sorting.

Notes

5.1.10 External Sorting

To handle massive amounts of data, External sorting algorithm is used. It is a kind of sorting algorithm as the name suggests. After sorting when the date do not fit into the main memory then they must reside in external memory. This sorting technique uses hybrid sort-merge strategy. In this process, small chunk of data which can be fit into main memory are read, sorted and written to a temporary file. Later in merge phase, the sorted sub-files are combined into a single larger file.

External merge sort algorithm is one of the example of external sorting which sorts chunks that can fit in RAM and then merges the sorted chunks together. For this, we need to first divide the file into runs such that the size of run is small enough to fit into main memory. Then using merge sort, sort each run in the main memory. Later we need to merge the resulting runs together into bigger runs, until the file is sorted. To learn external sort algorithm we need to first understand Merge sort (for sorting individual runs) and Merge K sorted array (for merging sorted runs).

Inputs:

input_file : Name of input file. input.txt

output_file : Name of output file, output.txt

run_size : Size of a run (can fit in RAM)

num_ways : Number of runs to be merged

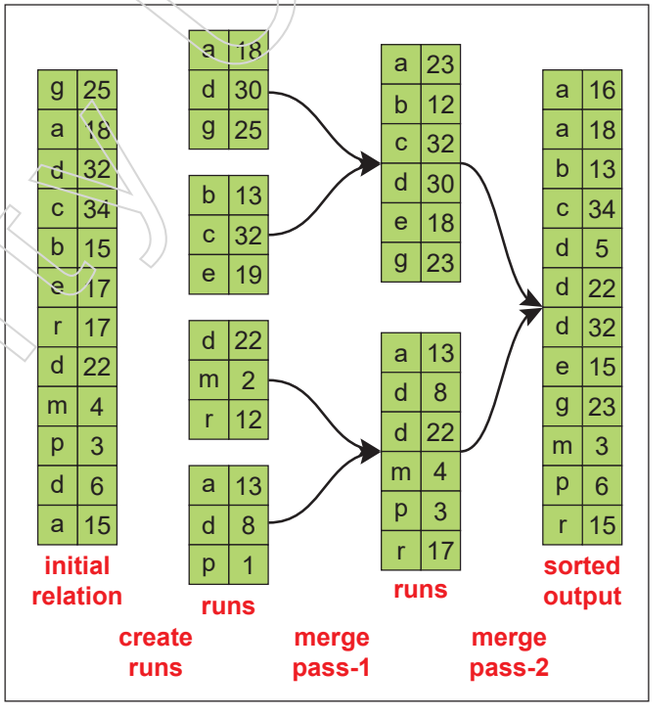


Fig 5.5 : External Sort

Stage 1: First we need to create a number of sorted runs and need to sort each of them. In this stage, we perform the sorting operation on the disk blocks. After completing the steps of Stage 1, proceed to Stage 2.

Steps:

Notes

i = 0;

repeat

read either M blocks or the rest of the relation having a smaller size;

sort the in-memory part of the relation;

write the sorted data to run file Ri;

i =i+1;

Until the end of the relation

Stage 2: In Stage 2, we need to merge the runs. Let us think the total number of runs as N which is less than M. Here, we can allocate one block to each run and still have some extra space to hold one block of output. We perform the operation as follows:

Steps:

read one block of each of N files Ri into a buffer block in memory;

repeat

select the first tuple among all buffer blocks (where selection is made in sorted order);

write the tuple to the output, and then delete it from the buffer block;

if the buffer block of any run Ri is empty and not EOF(Ri)

then read the next block of Ri into the buffer block;

Until all input buffer blocks are empty

After completing Stage 2, we will get a sorted output. The output file is then buffered for minimizing the disk-write operations. As this algorithm merges N runs, that's why it is known as an N-way merge.

Notes

Unit-5.2: Searching Techniques

5.2.1 Linear Search

Searching is an procedure or a technique that discovers the place of a given element or value in the list. Any search is believed to be successful or unsuccessful depending upon whether the element that is being searched is found or not.

This is the easiest method for searching. In this technique of searching, the element to be uncovered in searching the elements to be found is searched sequentially in the list. This method can be performed on a sorted or an unsorted list (usually arrays). In case of a sorted list searching starts from 0th element and remains until the element is found from the list or the element whose value is greater than (assuming the list is sorted in ascending order), the value being searched is achieved.

It is a simple algorithm that searches for a certain item inside a list. It operates looping on each element O(n) unless and until a match occurs or the end of the array is achieved.

- algorithm Seqnl_Search(list, item)
- Pre: list != ;
- Post: return the index of the item if found, otherwise: 1
- index <- fi
- while index < list.Cnt and list[index] != item //cnt: counter variable
- index <- index + 1
- end while
- if index < list.Cnt and list[index] = item
- return index
- end if
- return: 1
- end Seqnl_Search

Example:

```
#include<stdio.h>
int main()
{
   int a[20],i,x,n;
   printf(“How many elements?”);
   scanf(“%d”,&n);
   printf(“Enter array elements:n”);
   for(i=0;i<n;++i)
      scanf(“%d”,&a[i]);
```

Notes

```
    printf("nEnter element to search:");
    scanf("%d",&x);
    for(i=0;i<n;++i)
       if(a[i]==x)
          break;
    if(i<n)
       printf("Element found at index %d",i);
    else
       printf("Element not found");
    return 0;
}
```

5.2.2 Binary Search

Binary search is a very fast and efficient searching technique. It requires the list to be in sorted order. In this method, to search an element you can compare it with the present element at the center of the list. If it matches, then the search is successful otherwise the list is divided into two halves: one from the 0th element to the middle element which is the center element (first half) another from the center element to the last element (which is the 2nd half) where all values are greater than the center element.

The searching mechanism proceeds from either of the two halves depending upon whether the target element is greater or smaller than the central element. If the element is smaller than the central element, then searching is done in the first half, otherwise searching is done in the second half.

Algorithm

- algorithm Binary_Search(list, item)
- Set L to 0 and R to n: 1
- if L > R, then Binary_Search terminates as unsuccessful
- else
- Set m (the position in the mid element) to the floor of (L + R) / 2
- if Am < T, set L to m + 1 and go to step 3
- if Am > T, set R to m: 1 and go to step 3
- Now, Am = T,
- the search is done; return (m)

Example:

```
/* C program to implement Binary Search */
#include <stdio.h>
```

Notes

```
int main()
{
  int c, first, last, middle, n, search, array[100];
printf("Enter number of elements\n");
scanf("%d", &n);
printf("Enter %d integers\n", n);
  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);
  printf("Enter value to find\n");
  scanf("%d", &search);
  first = 0;
  last = n - 1;
  middle = (first+last)/2;
  while (first <= last) {
    if (array[middle] < search)
      first = middle + 1;
    else if (array[middle] == search) {
      printf("%d found at location %d.\n", search, middle+1);
      break;
    }
    else
      last = middle - 1;
    middle = (first + last)/2;
  }
  if (first > last)
    printf("Not found! %d isn't present in the list.\n", search);
  return 0;
}
```

Output:

```
Enter number of elements
5
Enter 5 integers
```

Notes

1 2 3 4 5

Enter value to find

1

1 found at location 1.

5.2.3 Hashing

Hashing is an essential Data Structure which is designed to use a particular function called the Hash function which is used to map a given value with a particular key for swifter access of elements. The efficiency of mapping depends of the effectiveness of the hash function used. Hash table or hash map is a data structure utilized to store key-value pairs. It is a collection of items stored to make it effortless to find them later. It utilizes a hash function to calculate an index into an array of buckets or slots from which the anticipated value can be discovered. It is an array of list where each list is recognized as bucket. It comprises value based on the key. Hash table is employed to implement the map interface and extends Dictionary class. Hash table is synchronized and encompasses only unique elements.

Features :

- Hashing is the process of mapping enormous amount of data item to smaller table with the help of hashing function.
- Hashing is also known as Hashing Algorithm or Message Digest Function.
- It is a method to convert a range of key values into a range of indexes of an array.
- It is used to facilitate the next level searching method when matched with the linear or binary search.
- Hashing allows to update and retrieve any data entry in a constant time O(1).
- Constant time O(1) indicates the operation does not depend on the size of the data.
- Hashing is utilized with a database to enable items to be retrieved more quickly.
- It is used in the encryption and decryption of digital signatures.

Example: hashIndex = key % noOfBuckets

Insert: Move to the bucket corresponds to the above calculated hash index and insert the new node at the end of the list.

Delete: To delete a node from hash table, calculate the hash index for the key, move to the bucket corresponds to the calculated hash index, search the list in the current bucket to find and remove the node with the given key (if found).

Notes

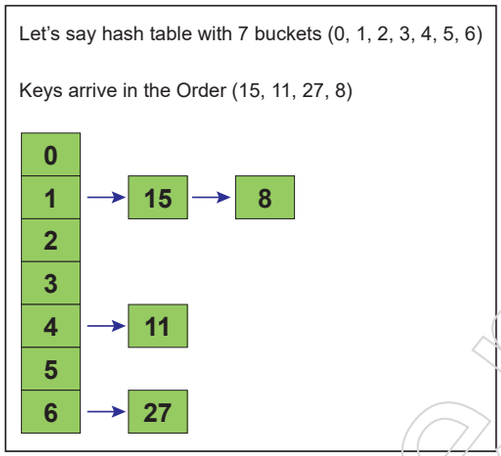


Fig 5.6 Hash table

Example:

```
/* C program to implement hashing with chaining */
   #include<stdio.h>
   #include<stdlib.h>
   #include<conio.h>
   /* Node for storing an item in a Linked List */
   struct node
   {
    int key;
    int value;
    struct node *next;
   };
   /* For storing a Linked List at each index of Hash Table  */
   struct arrayitem
   {
    struct node *head;
    struct node *tail;
     };
   struct arrayitem *array;
   int size = 0;  // Determines the no. of elements present in Hash Table
   int max = 10;// Determines the maximum capacity of Hash Table array
  /* This function creates an index corresponding to the every given key */
```

Notes

```
int hashcode(int key)
{
 return (key % max);
}
struct node* get_element(struct node *list, int find_index);
void remove_element(int key);
void rehash();
void init_array();
void insert(int key, int value)
{
        float n = 0.0;
 /* n => Load Factor, keeps check on whether rehashing is required or not */
 int index = hashcode(key);
 /* Extracting Linked List at a given index */
 struct node *list = (struct node*) array[index].head;
 /* Creating an item to insert in the Hash Table */
 struct node *item = (struct node*) malloc(sizeof(struct node));
 item->key = key;
 item->value = value;
 item->next = NULL;
 if (list == NULL)
     {
/* Absence of Linked List at a given Index of Hash Table */
        printf("Inserting %d(key) and %d(value) \n", key, value);
        array[index].head = item;
        array[index].tail = item;
        size++;
        }
     else
     {
        /* A Linked List is present at given index of Hash Table */
        int find_index = find(list, key);
```

Notes

```
            if (find_index == -1)
             {
    //Key not found in existing linked list
            array[index].tail->next = item;
    array[index].tail = item;
    size++;
                    }
else
      {
    // Key already present in linked list
   struct node *element = get_element(list, find_index);
   element->value = value;
            }
            }
            n = (1.0 * size) / max;
    if (n >= 0.75)

        {
            //rehashing
            printf("going to rehash\n");
            rehash();
    }
   }
  void rehash()
   {
    struct arrayitem *temp = array;
    /* temp pointing to the current Hash Table array */
    int i = 0, n = max;
    size = 0;
    max = 2 * max;
    array = (struct arrayitem*) malloc(max * sizeof(struct node));
    init_array();
```

Notes

```
    for (i = 0; i < n; i++)
        {
/* Extracting the Linked List at position i of Hash Table array */
struct node* list = (struct node*) temp[i].head;
if (list == NULL)
{
/* if there is no Linked List, then continue */
continue;
}
else
{
/* Presence of Linked List at i, keep moving and accessing the Linked List item until the end, Get one key and value at a time and add it to new Hash Table array.*/
while (list != NULL)
{
insert(list->key, list->value);
  list = list->next;
    }                    }          }
temp = NULL;
   }
  //This function finds the given key in the Linked List
 int find(struct node *list, int key)
{
int retval = 0;
struct node *temp = list;
while (temp != NULL)
 {
if (temp->key == key)
{
return retval;
}
temp = temp->next;
```

Notes

```
retval++;
}
return -1;
   }
   /* Returns the node (Linked List item) located at given find_index  */
struct node* get_element(struct node *list, int find_index)
   {
int i = 0;
struct node *temp = list;
while (i != find_index)
{
temp = temp->next;
i++;
}
return temp;
   }
   /* To remove an element from Hash Table */
void remove_element(int key)
{
int index = hashcode(key);
struct node *list = (struct node*) array[index].head;
if (list == NULL)
{
printf("This key does not exists\n");
}
else
{
int find_index = find(list, key);
if (find_index == -1)
{
printf("This key does not exists\n");
}
```

Notes

```
else
 {
struct node *temp = list;
if (temp->key == key)
 {
array[index].head = temp->next;
printf("This key has been removed\n");
return;
}
while (temp->next->key != key)
  {
temp = temp->next;
}
if (array[index].tail == temp->next)
  {
temp->next = NULL;
array[index].tail = temp;
}
 else
 {
temp->next = temp->next->next;
}
printf("This key has been removed\n");
}   }      }
   /* To display the contents of Hash Table */
void display()
{
int i = 0;
for (i = 0; i < max; i++)
{
struct node *temp = array[i].head;
if (temp == NULL)
```

Notes

```
{
printf("array[%d] has no elements\n", i);
}

else
{
printf("array[%d] has elements-: ", i);
while (temp != NULL)
{
printf("key= %d  value= %d\t", temp->key, temp->value);
temp = temp->next;
}
printf("\n");
}
}
 }
  /* For initializing the Hash Table */
 void init_array()
{
int i = 0;
for (i = 0; i < max; i++)
 {
array[i].head = NULL;
array[i].tail = NULL;
   }
   }
 /* Returns size of Hash Table */
   int size_of_array()
   {
return size;
   }
void main()
```

Notes

```
{
int choice, key, value, n, c;
clrscr();
array = (struct arrayitem*) malloc(max * sizeof(struct arrayitem*));
init_array();
do {
printf("Implementation of Hash Table in C chaining with Singly Linked List \n\n");
printf("MENU-: \n1.Inserting item in the Hash Table""\n2.Removing item from the Hash Table""\n3.Check the size of Hash Table" "\n4.To display a Hash Table""\n\n Please enter your choice -: ");
scanf("%d", &choice);
switch(choice)
{
case 1:
printf("Inserting element in Hash Table\n");
printf("Enter key and value-:\t");
scanf("%d %d", &key, &value);
insert(key, value);
 break;
case 2:
printf("Deleting in Hash Table \nEnter the key to delete-:");
scanf("%d", &key);
remove_element(key);
break;
case 3:
n = size_of_array();
printf("Size of Hash Table is-:%d\n", n);
break;
case 4:
display();
break;
default:
printf("Wrong Input\n");
```

Notes

```
}
printf("\nDo you want to continue-:(press 1 for yes)\t");
scanf("%d", &c);
}while(c == 1);
getch();
   }
```

5.2.4 Hash Function

- A fixed process converts a key to a hash key is known as a Hash Function.
- This function takes a key and maps it to a value of a certain length which is called a Hash value or Hash.
- Hash value symbolizes the original string of characters, but it is normally lesser than the original.
- It transfers the digital signature and then both hash value and signature are sent to the receiver. Receiver utilizes the same hash function to generate the hash value and then compares it to that received with the message.
- If the hash values are same, the message is communicated without errors.

5.2.4 Collision Resolution Technique

The hash function is utilized to compute the index of the array. The hash value is used to store the key in the hash table, as an index. The hash function can return the identical hash value for two or more keys. When two or more keys are provided the same hash value, it is called a collision. To handle this collision, we use collision resolution techniques.

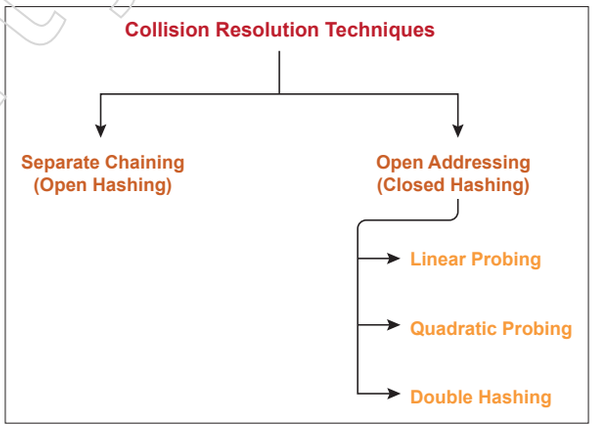


Fig 5.7 Collission resoluition technique

There are two types of collision resolution techniques.

1. Separate chaining (open hashing)
2. Open addressing (closed hashing)

1. Separate chaining

Notes

In this technique, a linked list is produced from the slot in which collision has occurred, after which the new key is injected into the linked list. This linked list of slots looks like a chain, so it is termed separate chaining. It is applied more when we do not know how many keys to insert or delete.

Time complexity

a. Its worst-case complexity for searching is o(n).

b. Its worst-case complexity for deletion is o(n).

Advantages of separate chaining

a. It is simple to implement.

b. The hash table never fills full, so we can add additional elements to the chain.

c. It is less sensitive to the function of the hashing.

Disadvantages of separate chaining

a. In this, cache performance of chaining is not good.

b. The memory wastage is too much in this method.

c. It requires more space for element links.

2. Open addressing

Open addressing is collision-resolution method that is applied to regulate the collision in the hashing table. There is no key stored outside of the hash table. Consequently, the size of the hash table is always greater than or equal to the number of keys. It is also called closed hashing.

The following techniques are used in open addressing:

a. Linear probing

b. Quadratic probing

c. Double hashing

Linear probing

In this, when the collision occurs, we perform a linear probe for the next slot, and this probing is performed until an empty slot is found. In linear probing, the worst time to search for an element is O(table size). The linear probing gives the best performance of the cache but its problem is clustering. The main advantage of this technique is that it can be easily calculated.

Disadvantages of linear probing

1. The main problem is clustering.
2. It takes too much time to find an empty slot.

Quadratic probing

In this, when the collision occurs, we probe for i2th slot in ith iteration, and this probing is accomplished until an empty slot is discovered. The cache performance in quadratic probing is lower than the linear probing. Quadratic probing also diminishes

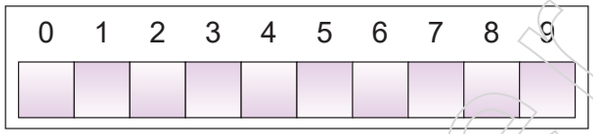
Notes

the problem of clustering.

Double hashing

In this, you utilize another hash function, and probe for (i * hash 2(x)) in the ith iteration. It requires longer to determine two hash functions. The double probing provides the very poor the cache performance, but there has no clustering problem in it.

Example of Hashing Techniques



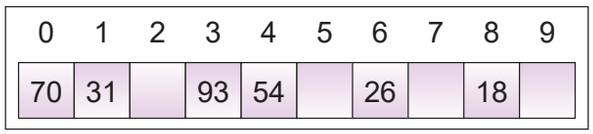
The above figure shows the hash table with the size of n = 10. Each position of the hash table is called as Slot. In the above hash table, there are n slots in the table, names = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. Slot 0, slot 1, slot 2 and so on. Hash table contains no items, so every slot is empty.

As we know the mapping between an item and the slot where item belongs in the hash table is termed the hash function. The hash function receives any item in the collection and returns an integer in the range of slot names between 0 to n-1.

Suppose we have integer items {26, 70, 18, 31, 54, 93}. One common method of determining a hash key is the division method of hashing and the formula is Hash Key = Key Value % Number of Slots in the Table

Division method or reminder method takes an item and separates it by the table size and returns the remainder as its hash value.

Data Item	Value % No. of Slots	Hash Value
26	26 % 10 = 6	6
70	70 % 10 = 0	0
18	18 % 10 = 8	8
31	31 % 10 = 1	1
54	54 % 10 = 4	4
93	93 % 10 = 3	3



After computing the hash values, we can insert each item into the hash table at the designated position as shown in the above figure. In the hash table, 6 of the 10 slots

Notes

are inhabited, it is described to as the load factor and denoted by, λ = No. of items / table size. For example , λ = 6/10.

It is easy to search for an item using hash function where it computes the slot name for the item and then checks the hash table to see if it is appear. Constant amount of time O(1) is required to compute the hash value and index of the hash table at that location.

Linear Probing

- Take the above example, if we insert next item 40 in our collection, it would have a hash value of 0 (40 % 10 = 0). But 70 also had a hash value of 0, it becomes a problem. This problem is called as Collision or Clash. Collision creates a problem for hashing technique.
- Linear probing is utilized for resolving the collisions in hash table, data structures for maintaining a collection of key-value pairs.
- Linear probing was developed by Gene Amdahl, Elaine M. McGraw and Arthur Samuel in 1954 and analyzed by Donald Knuth in 1963.
- It is a component of open addressing scheme for using a hash table to resolve the dictionary problem.
- The simplest method is called Linear Probing. Formula to compute linear probing is:

  P = (1 + P) % (MOD) Table_size

  For example,

0	1	2	3	4	5	6	7	8	9
70	31		93	54		26		18	

If we insert next item 40 in our collection, it would have a hash value of 0 (40 % 10 = 0). But 70 also had a hash value of 0, it becomes a problem.

Linear probing solves this problem:

P = H(40)

44 % 10 = 0

Position 0 is occupied by 70. so we look elsewhere for a position to store 40.

Using Linear Probing:

P= (P + 1) % table-size

0 + 1 % 10 = 1

But, position 1 is occupied by 31, so we look elsewhere for a position to store 40.

Using linear probing, we try next position : 1 + 1 % 10 = 2

Position 2 is empty, so 40 is inserted there.

Notes

0	1	2	3	4	5	6	7	8	9
70	31	40	93	54		26		18	

Module End Questions

1. The worst-case occur in linear search algorithm when
   a) Item is somewhere in the middle of the array
   b) Item is not in the array at all
   c) Item is the last element in the array
   d) Item is the last element in the array or item is not there at all
2. If the number of records to be sorted is small, then ..... sorting can be efficient.
   a) Merge
   b) Heap
   c) Selection
   d) Bubble
3. The complexity of the sorting algorithm measures the ..... as a function of the number n of items to be sorter.
   a) average time
   b) running time
   c) average-case complexity
   d) case-complexity
4. Which of the following is not a limitation of binary search algorithm?
   a) must use a sorted array
   b) requirement of sorted array is expensive when a lot of insertion and deletions are needed
   c) there must be a mechanism to access middle element directly
   d) binary search algorithm is not efficient when the data elements more than 1500.
5. The Average case occurs in the linear search algorithm
   a) when the item is somewhere in the middle of the array
   b) when the item is not the array at all
   c) when the item is the last element in the array
   d) Item is the last element in the array or item is not there at all
6. Binary search algorithm cannot be applied to
   a) sorted linked list

Notes

b) sorted binary trees

c) sorted linear array

d) pointer array

7. Sorting algorithm can be characterized as

a) Simple algorithm which require the order of n2 comparisons to sort n items.

b) Sophisticated algorithms that require the O(nlog2n) comparisons to sort items.

c) Both of the above

d) None of the above

8. ……. is putting an element in the appropriate place in a sorted list yields a larger sorted order list.

a) Insertion

b) Extraction

c) Selection

d) Distribution

9. Which of the following sorting algorithm is of priority queue sorting type?

a) Bubble sort

b) Insertion sort

c) Merge sort

d) Selection sort

10. Partition and exchange sort is ……..

a) quick sort

b) tree sort

c) heap sort

d) bubble sort

Answer key:

1. d, 2. c, 3. b, 4. d, 5. a, 6. a, 7. c, 8. a, 9. d, 10. a

Assignment :

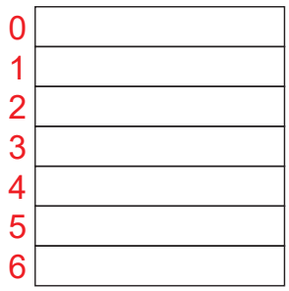
Using the hash function 'key mod 7', insert the following sequence of keys in the hash table- 50, 700, 76, 85, 92, 73 and 101

Use linear probing technique for collision resolution.

Step-01:

Notes

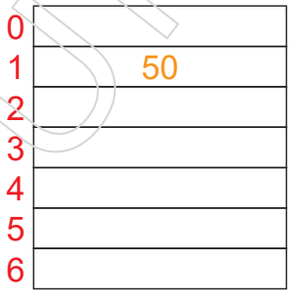
- Draw an empty hash table.
- For the given hash function, the possible range of hash values is [0, 6].
- So, draw an empty hash table consisting of 7 buckets as



Step-02:

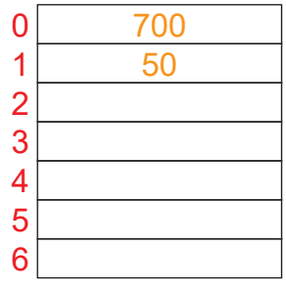
Insert the given keys in the hash table one by one.

- The first key to be inserted in the hash table = 50.
- Bucket of the hash table to which key 50 maps = 50 mod 7 = 1.
- So, key 50 will be inserted in bucket-1 of the hash table as-



Step-03:

- The next key to be inserted in the hash table = 700.
- Bucket of the hash table to which key 700 maps = 700 mod 7 = 0.
- So, key 700 will be inserted in bucket-0 of the hash table as-



Step-04:

- The next key to be inserted in the hash table = 76.
- Bucket of the hash table to which key 76 maps = 76 mod 7 = 6.
- So, key 76 will be inserted in bucket-6 of the hash table as-

Notes

0	700
1	50
2	
3	
4	
5	
6	76

Step-05:

- The next key to be inserted in the hash table = 85.
- Bucket of the hash table to which key 85 maps = 85 mod 7 = 1.
- Since bucket-1 is already occupied, so collision occurs.
- To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
- The first empty bucket is bucket-2.
- So, key 85 will be inserted in bucket-2 of the hash table as-

0	700
1	50
2	85
3	92
4	
5	
6	76

Step-06:

- The next key to be inserted in the hash table = 92.
- Bucket of the hash table to which key 92 maps = 92 mod 7 = 1.
- Since bucket-1 is already occupied, so collision occurs.
- To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
- The first empty bucket is bucket-3.
- So, key 92 will be inserted in bucket-3 of the hash table as-

Notes

0	700
1	50
2	85
3	92
4	73
5	
6	76

Step-07:

- The next key to be inserted in the hash table = 73.
- Bucket of the hash table to which key 73 maps = 73 mod 7 = 3.
- Since bucket-3 is already occupied, so collision occurs.
- To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
- The first empty bucket is bucket-4.
- So, key 73 will be inserted in bucket-4 of the hash table as-

0	700
1	50
2	85
3	92
4	73
5	101
6	76

Step-08:

The next key to be inserted in the hash table = 101.

- Bucket of the hash table to which key 101 maps = 101 mod 7 = 3.
- Since bucket-3 is already occupied, so collision occurs.
- To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
- The first empty bucket is bucket-5.
- So, key 101 will be inserted in bucket-5 of the hash table as-