ALAGAPPA UNIVERSITY

[Accredited with 'A+' Grade by NAAC (CGPA:3.64) in the Third Cycle
and Graded as Category–I University by MHRD-UGC]
(A State University Established by the Government of Tamilnadu)

KARAIKUDI – 630 003

DIRECTORATE OF DISTANCE EDUCATION

B.Sc (COMPUTER SCIENCE)

III - SEMESTER

13033

DATA STRUCTURES AND ALGORITHMS

Author:
Dr.A. Sumathi,
Assistant Professor,
Department of CSE,SRC Sastra University,
Kumbakonam – 612 001

SYLLABI-BOOK MAPPING TABLE

DATA STRUCTURES AND ALGORITHMS

Syllabi	Mapping in Book
BLOCK I : INTRODUCTION	
UNIT -I: **Introduction to Data Structure** : Types of Data Structure , Primitive data types Algorithms**:** –Time and space Complexity of algorithms	1-10
UNIT-II: **Arrays:** Array initialization, Definition of Array, Characteristic of Array ,One-dimensional Array, Two-dimensional array and Multi dimensional array	11-18
BLOCK II : LINEAR DATA STRUCTURE	
UNIT-III: Stack : Stack related terms, Operations on a stack,	19-26
UNIT-IV: Representation of Stack: Implementation of a stack – application of Stack. expression Evaluation Polish notation.	27-32
UNIT V-: **Queues:** Operations on queue Circular Queue, Representation of Queues, Application of Queues	33-45
UNIT-VI: **List**: Merging lists, Linked list, Single linked list, Double Linked List, Header Linked list	46-59
UNIT-VII: **Operation on Linked List :** Insertion and Deletion of linked list	60-66
UNIT-VIII: **Traversal:** Traversing a linked list , Representation of linked list.	67-71
BLOCK : III NON-LINEAR DATA STRUCTURE	
UNIT-IX: **Trees:** Binary Trees, Types of Binary trees, Binary Tree Representation	72-81
UNIT-X: **Binary Tree operations / Applications :** Traversing Binary Trees, Binary Search tree	82-87

CONTENTS

BLOCK – I INTRODUCTION

UNIT- I INTRODUCTION TO DATA STRUCTURE

Structure

1. 1 Introduction

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently. Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc. Data Structures are widely used in almost every aspect of Computer Science i.e. operating System, Compiler Design, Artificial intelligence, Graphics and many more.

Need for data structure:

A data structure helps you to understand the relationship of one data element with the other and organize it within the memory. Sometimes the organization might be simple. E.g.: List of names of months in a year –Linear Data Structure, List of historical places in the world- Non-Linear Data Structure. A data structure helps you to analyze the data, store it and organize it in a logical and mathematical manner.

Data and Information

1. Raw facts gathered about a condition, event, idea, entity or anything else which is bare and random, is called data. Information refers to facts concerning a particular event or subject, which are refined by processing.
2. Data are simple text and numbers, while information is processed and interpreted data.

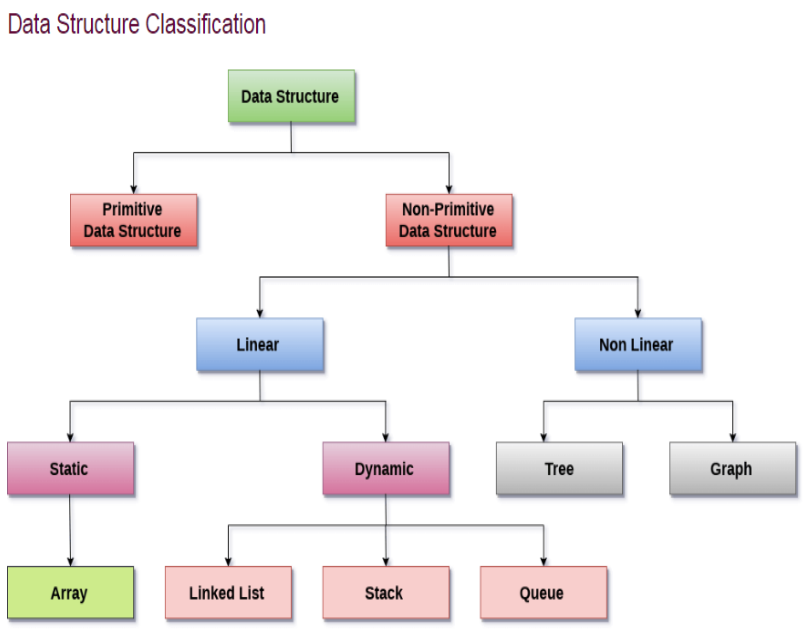
Data is in an unorganized form, i.e. it is randomly collected facts and figures which are processed to draw conclusions. On the other hand, when the data is organized, it becomes information, which presents data in a better way and gives meaning to it.

1.2 Objectives

After going through the unit you will be able to;

- Discuss Primitive data types
- Understand composite data types
- Explain Linear and non-linear data structure
- Analyze abstract data types

1.3 Types of Data Structures



Primitive Data Structures

Primitive Data Structures are the basic data structures that directly operate upon the machine instructions. They have different representations on different computers. For example, integer, character, and string are all primitive data types. Programmers can use these data types when creating variables in their programs.

Non-primitive Data Structures

Non-primitive data structures are more complicated data structures and are derived from primitive data structures.

1.3.1 Linear Data Structures

A data structure is called linear if all of its elements are arranged in the linear order. In linear data structures, the elements are stored in non-hierarchical way where each element has the successors and predecessors except the first and last element.

Types of Linear Data Structures are given below:

- **Arrays:** An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double. The elements of array share the same variable name but each one carries a different index number known as subscript. The array can be one dimensional, two dimensional or multidimensional.
- **Linked List:** Linked list is a linear data structure which is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memory locations. Each node of the list contains a pointer to its adjacent node.
- **Stack:** Stack is a linear list in which insertion and deletions are allowed only at one end, called **top**. A stack is an abstract data type (ADT), can be implemented in most of the programming languages. It is named as stack because it behaves like a real-world stack, for example: - piles of plates or deck of cards etc.
- **Queue:** Queue is a linear list in which elements can be inserted only at one end called **rear** and deleted only at the other end called **front**.It is an abstract data structure, similar to stack. Queue is opened at both end therefore it follows First-In-First-Out (FIFO) methodology for storing the data items.

1.3.2 Non Linear Data Structures

This data structure does not form a sequence i.e. each item or element is connected with two or more other items in a non-linear arrangement. The data elements are not arranged in sequential structure.

Types of Non Linear Data Structures are given below:

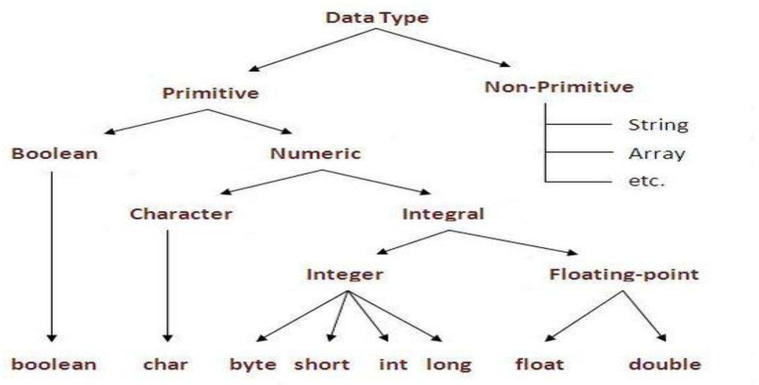
- **Trees:** Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottommost nodes in the hierarchy are called **leaf node** while the topmost node is called **root node**. Tree data structure is based on the parent-child relationship among the nodes. Each node contains pointers that points to the child node. Each node in the tree can have more than one children except the leaf nodes whereas each node can have at most one parent except the root node. Trees can be classified into many categories which will be discussed later.

- **Graphs:** Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges. A graph is different from tree in the sense that a graph can have cycle while the tree cannot have the one.

Primitive Data Types

Basically, primitive data types, as the name is self-explanatory, are the **data types that already come with the programming language** completely ready for the programmer's use.

NOTES



1.4 Algorithms

An algorithm is a procedure having well defined steps for solving a particular problem. Algorithm is finite set of logic or instructions, written in order for accomplish the certain predefined task. It is not the complete program or code, it is just a solution (logic) of a problem, which can be represented either as an informal description using a Flowchart or Pseudo code.

Characteristics of an Algorithm

An algorithm must follow the mentioned below characteristics:

- **Input:** An algorithm must have 0 or well defined inputs.
- **Output:** An algorithm must have 1 or well defined outputs, and should match with the desired output.
- **Feasibility:** An algorithm must be terminated after the finite number of steps.
- **Independent:** An algorithm must have step-by-step directions which is independent of any programming code.
- **Unambiguous:** An algorithm must be unambiguous and clear. Each of their steps and input/outputs must be clear and lead to only one meaning.

The major categories of algorithms are given below:

- **Sort:** Algorithm developed for sorting the items in certain order.
- **Search:** Algorithm developed for searching the items inside a data structure.

- **Delete:** Algorithm developed for deleting the existing element from the data structure.
- **Insert:** Algorithm developed for inserting an item inside a data structure.
- **Update:** Algorithm developed for updating the existing element inside a data structure.

1.4.1 Time and space complexity of algorithms

Sometimes, there are more than one way to solve a problem. We need to learn how to compare the performance different algorithms and choose the best one to solve a particular problem. While analysing an algorithm, we mostly consider time complexity and space complexity. Time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. Similarly, Space complexity of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input.

The performance of algorithm is measured on the basis of following properties:

- **Time complexity:** It is a way of representing the amount of time needed by a program to run to the completion.

- **Space complexity:** It is the amount of memory space required by an algorithm, during a course of its execution. Space complexity is required in situations when limited memory is available and for the multi user system.

Each algorithm must have:

- **Specification:** Description of the computational procedure.
- **Pre-conditions:** The condition(s) on input.
- **Body of the Algorithm:** A sequence of clear and unambiguous instructions.
- **Post-conditions:** The condition(s) on output.

Example: Design an algorithm to multiply the two numbers x and y and display the result in z.

- Step 1 START
- Step 2 declare three integers x, y & z
- Step 3 define values of x & y
- Step 4 multiply values of x & y
- Step 5 store the output of step 4 in z
- Step 6 print z
- Step 7 STOP

Asymptotic analysis

Asymptotic analysis of an algorithm refers to defining the mathematical foundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.

Usually, the time required by an algorithm falls under three types

- **Best Case** − Minimum time required for program execution.
- **Average Case** − Average time required for program execution.
- **Worst Case** − Maximum time required for program execution.

Asymptotic Notations

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O Notation
- Ω Notation
- θ Notation

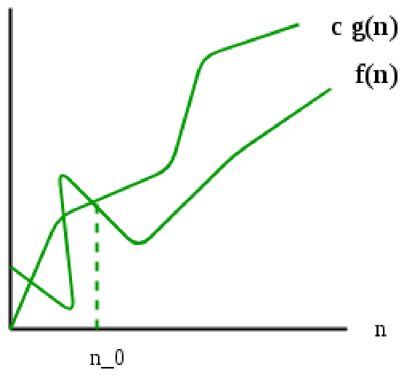
Big O Notation

The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. It is used for estimating the rate of function growth. It was introduced by Paul Bachmann in 1894.

f(n) is O(g(n)) if there exist positive numbers c and N such that

$f(n) \leq c.g(n), \forall n \geq N$

i.e., f is not larger than c.g(n) for sufficiently large ns. g(n) is an upper-bound on the value of f(n), where g(n) is the given function and f(n) is the analysing function.



f(n) = O(**g(n)**)

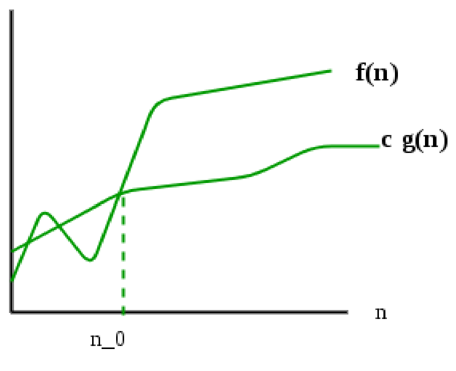
Ω Notation

Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.

f(n) is Ω(g(n)) if there exist positive numbers c and N such that

f(n) $\geq c.g(n), \forall\, n \geq N$.

i.e., c.g(n) is a lower bound on the size of f(n). Ω provides a lower bound for the value of f(n).



f(n) = Omega(g(n))

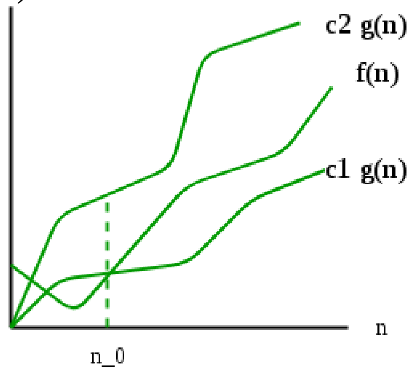
Θ Notation

The theta notation bounds a function from above and below, so it defines exact asymptotic behavior.

The function f(n) is Θ(g(n)), if there exist positive numbers c1,c2 and N such that

$$c1.g(n) \leq f(n) \leq c2.g(n), \forall n \geq N$$

i.e., f(n) is Θ(g(n)) if f(n) is O(g(n)) and f(n) is Ω(g(n)).
i.e., f(n) is Θ(g(n)) if and only if g(n) is both an upper bound and lower bound on f(n).



f(n) = theta(g(n))

1.5 Check your progress questions

1. What is data structure?
2. What is time and space complexity?

1.6 Check your progress questions

1) A **data structure** is a specialized format for organizing, processing, retrieving and storing **data**.
2) **Time complexity** of an algorithm quantifies the amount of **time** taken by an algorithm to run as a function of the length of the input. Similarly, **Space complexity** of an algorithm quantifies the amount of **space** or memory taken by an algorithm to run as a function of the length of the input.

1.7 Summary

- Each programming language provides various data types and each data type is represented differently within the computer's memory.
- The memory requirement of a data type determines the permissible range of values for that data type.
- The data types can be classified into several categories, including primitive data types and composite data types.
- The data types provided by a programming language are known as primitive data types or in-built data types.
- In addition to primitive and composite data types, programming languages allow the user to define new data types (or user-defined data types) as per his requirements.
- Generally, handling small problems is much easier than handling comparatively larger problems.
- The size of each module is kept as small as possible and if required, other modules are invoked from it.
- Second, a well-designed modular program has modules independent of each other's, implementation, which will make the program easily modifiable.
- An abstract data type (ADT) is an extension of a modular design in a way that the set of operations of an ADT are defined at a formal, logical level, and nowhere in ADT's definition, it is mentioned how these operations are implemented.
- The basic idea of ADT is that the implementation of the set of operations are written once in the program and the part of program which needs to perform an operation on ADT accomplishes this by invoking the required operation.
- If there is a need to change the implementation details of an ADT, the change will be completely transparent to the programs using it.
- The logical or mathematical model used to organize the data in main memory is called a data structure.
- These features should be kept in mind while choosing a data structure for a particular situation.
- The choice of a data structure depends on its simplicity and effectiveness in processing of data.
- Data structures are divided into two categories, namely, linear data structure and non-linear data structure.

- A linear data structure is one in which its elements form a sequence. It means each element in the structure has a unique predecessor and a unique successor.
- A finite collection of homogenous elements is termed as an array.
- The elements of an array are always stored in a contiguous memory locations irrespective of the array size.
- A stack is a linear list of data elements in which the addition of a new element or the deletion of an element occurs only at one end.
- A queue is a linear data structure in which the addition or insertion of a new element occurs at one end, called 'Rear', and deletion of an element occurs at other end, called 'Front'.
- A tree consists of multiple nodes, with each node containing zero, one or more pointers to other nodes called child nodes.

1.8 Keywords

- **Traversing:** It means accessing all the data elements one by one to process all or some of them.
- **Substitution method**: In this method, a reasonable guess for the solution is made and it is proved through mathematical induction.
- **Recursion tree:** In this method, recurrences are represented as a tree whose nodes indicate the cost that is incurred at the various levels of recursion.
- **Searching:** It is the process of finding the location of a given data element in the data structure.
- **Insertion**: It means adding a new data element in the data structure. A new element can be inserted anywhere in the structure, such as in the beginning, in the end, or in the middle.
- **Deletion:** It means removing any existing data element from the data structure.
- **Sorting:** It is the process of arranging all the elements of a data structure in a logical order such as ascending or descending order.
- **Merging:** It is the process of combining the elements of two sorted data structures into a single sorted data structure.

1.9 Self-Assessment Questions and Exercises

Short-Answer Questions

1. Write a short note on abstract data types.
2. Write some points of differences between linear and non-linear data structures.
3. What are the different operations on data structures?
4. What are algorithms?
5. What are the various asymptotic notations?

6. Explain big O notation.

Long-Answer Questions

1. "The data types provided by a programming language are known as primitive data types or in-built data types. Different programming languages provide different set of primitive data types." Discuss in detail.
2. "If there is a need to change the implementation details of an ADT, the change will be completely transparent to the programs using it." Explain.
3. Write a detailed note on Algorithm Design Techniques.
4. What do you mean by time and space complexity of algorithms?

1.8 Further Readings

Storer, J.A. 2012. An Introduction to Data Structures and Algorithms. New York: Springer Publishing.
Preiss, Bruno. 2008. Data Structures and Algorithms with Object-Oriented Design Patterns in C++. London: John Wiley and Sons.
Pandey, Hari Mohan. 2009. Data Structures and Algorithms. New Delhi: Laxmi Publications.
Goodrich Michael, Tamassia Roberto and Michael H. Goldwasser. 2014. Data
Structures and Algorithms in Java. London: John Wiley and Sons.
McMillan, Michael. 2007. Data Structures and Algorithms Using C#. Cambridge, UK: Cambridge University Press.

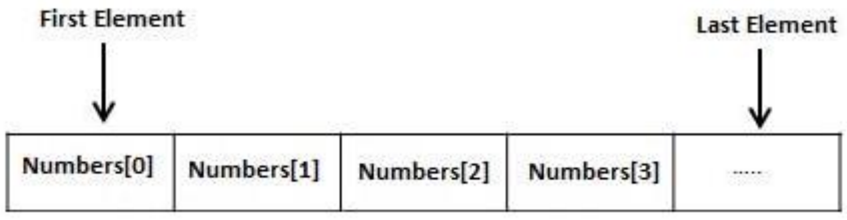
UNIT - II ARRAYS

Structure

2.1 Introduction

Array stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index. All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



2.2 Objectives

After going through this unit, the learner will be able to understand

- Array initialization and characterization
- Types of Arrays

2.3 Arrays

An array is a data structure that contains a group of elements. Typically these elements are all of the same data type, such as an integer, float, character or string. Arrays are commonly used in computer programs to organize data so that a related set of values can be easily sorted or searched.

```
int no[3]={10,20,30};
float no[3]={1.5,2.7,8.9};
```

2.3.1 Definition

An **array** is a collection of elements of the same type placed in contiguous memory locations that can be individually referenced by using an index to a unique identifier.

2.3.2 Initialization

An array can be initialized in two ways.

- By declaring and initializing it simultaneously.
  int arr[] = { 10, 20, 30, 40 }
- By declaring it first and initializing it later during run time.

2.3.3 Characteristics

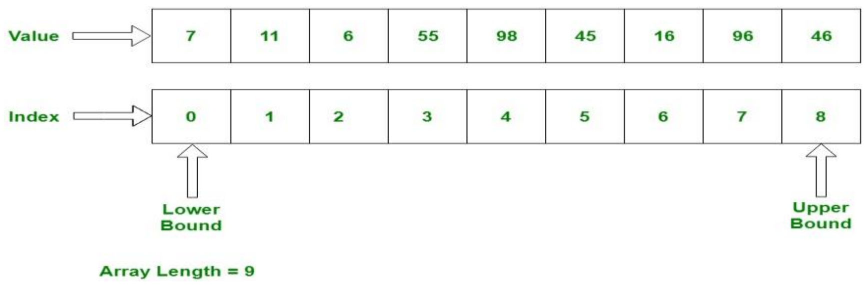
- An array holds elements that have the same data type
- Array elements are stored in subsequent memory locations
- Two-dimensional array elements are stored row by row in subsequent memory locations
- Array name represents the address of the starting element
- Array size should be mentioned in the declaration.
- Array size must be a constant expression and not a variable.

2.4 Types of Arrays

- One dimensional (1-D) arrays or Linear arrays.
- Multi-dimensional arrays.
  - ➢ Two dimensional (2-D) arrays or Matrix arrays.
  - ➢ Three dimensional arrays.

2.4.1 Single-Dimensional Arrays

A single-dimensional array is defined as an array in which only one subscript value is used to access its elements.

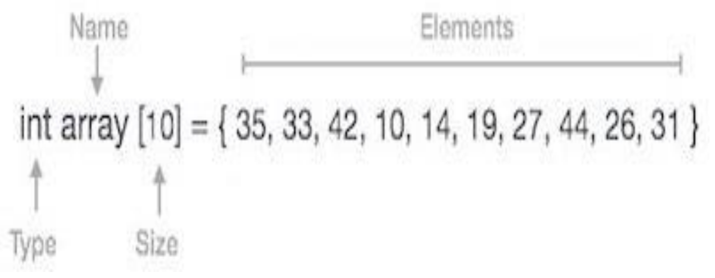


Before using an array in a program, it needs to be declared. The syntax of declaring a single-dimensional array in C is as follows:

data_type array_name[size];

where,

data_type = data type of elements to be stored in array
array_name = name of the array
size = the size of the array indicating that the lower bound of the array is 0 and the upper bound is size-1. Hence, the value of the subscript ranges from 0 to size-1.



For example, in the statement int array[10], an integer array of ten elements is declared and the array elements are indexed from 0 to 9. Once the compiler reads a single-dimensional array declaration, it allocates a specific amount of memory for the array. Memory is allocated to the array at the compile-time before the program is executed.

Working with single dimensional array

An array can be initialized in two ways. It can be done by declaring and initializing it simultaneously or by accepting elements of the already declared array from the user. Once an array is declared and initialized, the elements stored in it can be accessed any time. These elements can be accessed by using a combination of the name of an array and subscript value.

A program to illustrate the initialization of two arrays and display their elements is as follows:

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

```
#define MAX 5
void main()
{
        int A[MAX]={1,2,3,4,5};
        int B[MAX], i;
        clrscr();
        printf("Enter the elements of array B:\n");
        for (i=0;i<MAX;i++)
        {
                printf("Enter the element: ");
                scanf("%d", &B[i]);
        }
        printf("Elements of array A: \n");
        for (i=0;i<MAX;i++)
                printf("%d\t", A[i]);
        printf("\nElements of array B: \n");
        for (i=0;i<MAX;i++)
                printf("%d\t", B[i]);
        getch();
}
```

The output of the program is as follows:

```
Enter the elements of array b:
Enter a value: 6
Enter a value: 7
Enter a value: 8
Enter a value: 9
Enter a value: 10
Elements of array a:
        1 2 3 4 5
Elements of array b:
        6 7 8 9 10
```

2.4.2 Multi-Dimensional Arrays

Multi-dimensional arrays can be described as 'arrays of arrays'. A multi-dimensional array of dimension n is a collection of elements, which are accessed with the help of n subscript values. Most of the high-level languages, including C, support arrays with more than one dimension. However, the maximum limit of an array dimension is compiler dependent.

Two dimensional Arrays

A two-dimensional array is one in which two subscript values are used to access an array element. They are useful when the elements being processed are to be arranged in rows and columns (matrix form).

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

The syntax of declaring a two-dimensional array in C is as follows:

data_type array_name [row_size][column_size];

For example, in the statement int a[3][3], an integer array of three rows and three columns is declared. Once a compiler reads a two-dimensional array declaration, it allocates a specific amount of memory for this array.

Working with Two-dimensional Arrays

Once a two-dimensional array is declared and initialized, the array elements can be accessed anytime. Same as one-dimensional arrays, two-dimensional array elements can also be accessed by using a combination of the name of the array and subscript values. The only difference is that instead of one subscript value, two subscript values are used. The first subscript indicates the row number and the second subscript indicates the column number of a two-dimensional arrays.

A program to illustrate the traversal of a matrix (two-dimensional array) and finding the sum of its elements is as follows:

```
#include<stdio.h>
#include<conio.h>
#define MAX 10
/*Function prototype*/
void traverse(int [][MAX], int,int);
void main()
{
int ARR[MAX][MAX], i, j, m, n;
clrscr();
printf("Enter the number of rows and columns of a matrix A: ");
scanf("%d%d", &m, &n);
printf("Enter the elements of matrix A: \n");
for(i=0;i<m;i++)
for(j=0;j<n;j++)
scanf("%d", &ARR[i][j]);
traverse(ARR, m, n);
getch();
}

/*Function to find sum of elements of the matrix*/
void traverse(int ARR[][MAX], int m, int n)
{
int i, j, sum=0;
printf("Matrix A is: ");
for(i=0;i<m;i++)
{
printf("\n");
for(j=0;j<n;j++)
{
printf("%d ", ARR[i][j]);
sum=sum+ARR[i][j];
}
}
printf("\nSum of elements of a matrix is: %d", sum);
}
```

The output of the program is as follows:

```
Enter the number of rows and columns of a matrix A: 3 3
Enter the elements of matrix A:
1 2 3
4 5 6
7 8 9
Matrix A is:
1 2 3
4 5 6
7 8 9
Sum of elements of a matrix is: 45
```

2.5 Check your Progress

1. What is an array?
2. Write the syntax 1D array.
3. Limitations of array.

2.6 Answers to Check Your Progress Questions

1. An **array** is collection of items stored at contiguous memory locations. The idea is to store multiple items of same type together.
2. **Syntax:** datatype array_name[size];
3. **Array** is Static **data structure**. Memory can be allocated at compile time only Thus if after executing program we need more space for storing additional information then we cannot allocate additional space at run time

2.7 Summary

- Array is one of the data types that can be used for storing a list of elements.
- Arrays can be defined as a fixed-size sequence of elements of the same data type.
- The programmer can access a particular element of an array by using one or more indices or subscripts.
- If only one subscript is used then the array is known as a single-dimensional array.
- A single-dimensional array is defined as an array in which only one subscript value is used to access its elements.
- Once an array is declared and initialized, various operations such as traversing, searching, insertion, deletion, sorting, and merging can be performed on an array.
- To perform any operation on an array, the elements of the array need to be accessed.
- The process of accessing each element of an array is known as traversal.
- Multi-dimensional arrays can be described as 'arrays of arrays'. A multidimensional array of dimension n is a collection of elements, which are accessed with the help of n subscript values.
- Most of the high-level languages, including C, support arrays with more than one dimension.
- All the elements in an array are always stored next to each other.
- Each element in a single-dimensional array is associated with a unique subscript value, starting from 0 to size-1.
- Two-dimensional arrays are represented in a linear form to enable the storage of elements in the contiguous memory locations.
- There are two ways in which a two- dimensional array can be represented in the linear forms which are row-major order and column-major order.

2.8 Keywords

- **Array**: Array is a static data structure.
- **1D Array**: An array contains only one subscript.
- **2D Array**: An array contains two subscripts.

2.9 Self-Assessment questions

Short-Answer Questions

1. What is an array? Give an example
2. Write a short note on single dimensional arrays.
3. Give the syntax for declaring single dimensional array
4. Write in brief about multi-dimensional arrays

5. Write a short note on two dimensional arrays
6. Give the syntax for declaring two dimensional array
7. What do you mean by memory representation of single-dimensional arrays?
8. Describe the memory representation of a two-dimensional array.
9. What are some disadvantages of arrays?
10. Write a snippet to find the largest number in an array.

Long answer questions

1. "Once an array is declared and initialized, various operations such as traversing, searching, insertion, deletion, sorting, and merging can be performed on an array." Discuss. Also explain how can an operation be performed on an array?
2. Write a program to illustrate the traversal of an array.
3. "Multi-dimensional arrays can be described as 'arrays of arrays'. A multidimensional array of dimension n is a collection of elements, which are accessed with the help of n subscript values. " Discuss.
4. "A two-dimensional array is one in which two subscript values are used to access an array element." Discuss two-dimensional arrays in detail.
5. Write a program to illustrate the traversal of a matrix (two-dimensional array) and finding the sum of its elements.

2.10 Further Readings

Storer, J.A. 2012. An Introduction to Data Structures and Algorithms. NewYork: Springer Publishing.
Preiss, Bruno. 2008. Data Structures and Algorithms with Object-Oriented Design Patterns in C++. London: John Wiley and Sons.
Pandey, Hari Mohan. 2009. Data Structures and Algorithms. New Delhi: Laxmi Publications.
Goodrich Michael, Tamassia Roberto and Michael H. Goldwasser. 2014. Data
Structures and Algorithms in Java. London: John Wiley and Sons.
McMillan, Michael. 2007. Data Structures and Algorithms Using C#.Cambridge, UK: Cambridge University Press.

BLOCK –II LINEAR DATA STRUCTURE

NOTES

UNIT- III STACK

Structure

3.0 Introduction

A stack is a linear data structure in which an element can be added or removed only at one end called the top of the stack. A stack works on the principle of last in first out, and is also known as a last-in-first out (LIFO) list.

3.1 Objectives

After going through this unit, you will be able to:

- Discuss stack and its definition
- Explain the various stack related terms
- Analyse the operations on stack

3.2 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). An element can be added or removed only at one end called the top of the stack. The last element added to the stack is the first element to be removed, that is, the elements are removed in the opposite order in which they are added to the stack.

3.2.1 Implementation of stack

There are two ways to implement a stack:

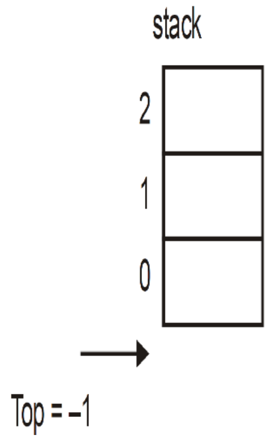
- Using array
- Using linked list

3.3 Stack Related Terms

NOTES

When a stack is organized as an array, a variable named Top is used to point to the top element of the stack. Initially, the value of Top is set as -1 to indicate an empty stack. Before inserting a new element onto a stack, it is necessary to test the condition of overflow. Overflow occurs when a stack is full and there is no space for a new element and an attempt is made to push a new element. If a stack is not full then the push operation can be performed successfully. To push an item onto a stack, Top is incremented by one and the element is inserted at that position.
Similarly, before removing the top element from the stack, it is necessary to check the condition of underflow. Underflow occurs when a stack is empty and an attempt is made to pop an element. If a stack is not empty, POP operation can be performed successfully. To POP (or remove) an element from a stack, the element at the top of the stack is assigned to a local variable and then Top is decremented by one. The total number of elements in a stack at a given point of time can be calculated from the value of Top as follows.

Number of elements = Top + 1



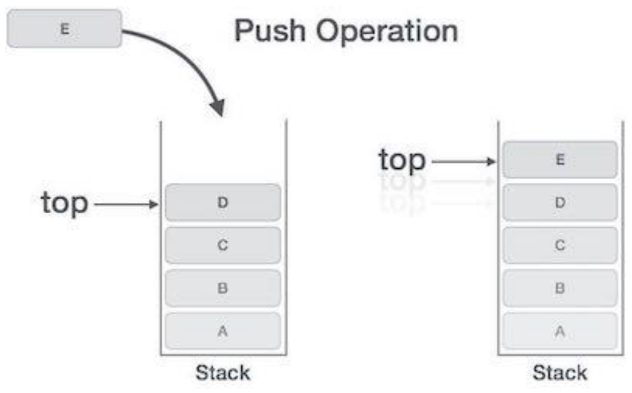
Empty stack

3.4 Operations on stack

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.

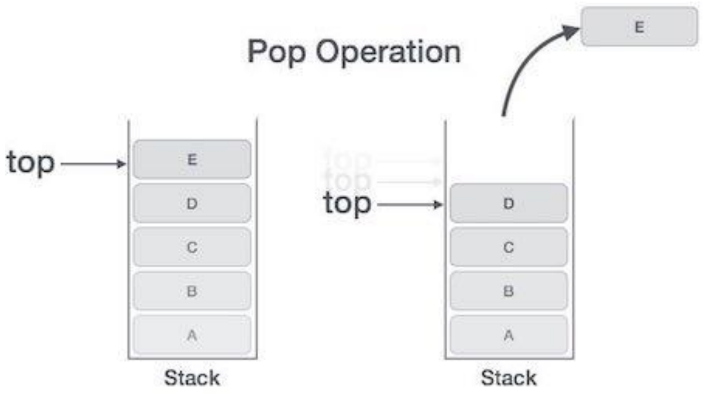
NOTES



Algorithm for push

```
Algorithm Push( )
{
    if (TOP =n-1)
    {
            Print"stack is full"
        }
        Else
        {
                Top=top+1
            S[ TOP ] =x
  }
}
```

- **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.



Algorithm for pop

```
Algorithm Pop()
{
        if ( TOP == -1)
        {
        Print"stack is empty"
        }
        Else
        {

                y= s[ TOP ]
                TOP = TOP -1
            }
}
```

Stacktop

This function returns the top-most element of the stack when it is not empty. Otherwise it says “Underflow”.

```
Algorithm StackTop()
{
        if(top==-1)
        {
                Print “Empty Stack”
        }
        else
        {
                Print A[top]
        }
}
```

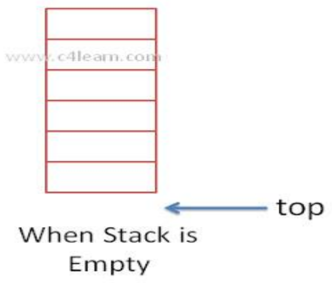
- **Peep:** This algorithm is used to display the ith element from the top of the stack. So, it can display only when the stack has elements. Otherwise, if the stack underflows, then the peep operation is not possible.

```
Algorithm Peep( i )
{
        if ( TOP > -1)
        {
                if( (TOP – i +1) > 0)
                        { print S[ TOP – i +1 ] }
                else
                        print “Invalid position”
        }
        else
        {
                print “Stack Underflow,can’t display”
        }
   }
```

Algorithm for isEmpty

This algorithm checks if the queue is empty by checking the value of TOP. If it is equal to -1, it display Empty; else it displays not empty.

```
Algorithm IsEmpty()

{

        if ( TOP == -1)

        { print “Stack is Empty” }

        else
        { print " Stack is not Empty" }
}
```



IsFull:

This algorithm checks if the queue is Full by checking the value of TOP. If it is equal to n-1, it display Full; else it displays not Full.

```
Algorithm IsFull()
{
        if ( TOP == n -1)
        { print "Stack is Full" }
        else
        { print " Stack is not Full" }
}
```

A program to implement a stack as an array is as follows:

```
// C program for array implementation of stack
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>

// A structure to represent a stack
struct Stack {
    int top;
    unsigned capacity;
    int* array;
};

// function to create a stack of given capacity. It initializes size of
// stack as 0
struct Stack* createStack(unsigned capacity)
{
    struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
    stack->capacity = capacity;
    stack->top = -1;
    stack->array = (int*)malloc(stack->capacity * sizeof(int));
    return stack;
}
```

```
// Stack is full when top is equal to the last index
int isFull(struct Stack* stack)
{
   return stack->top == stack->capacity - 1;
}

// Stack is empty when top is equal to -1
int isEmpty(struct Stack* stack)
{
   return stack->top == -1;
}

// Function to add an item to stack.  It increases top by 1
void push(struct Stack* stack, int item)
{
   if (isFull(stack))
      return;
   stack->array[++stack->top] = item;
   printf("%d pushed to stack\n", item);
}

// Function to remove an item from stack.  It decreases top by 1
int pop(struct Stack* stack)
{
   if (isEmpty(stack))
      return INT_MIN;
   return stack->array[stack->top--];
}

// Function to return the top from stack without removing it
int peek(struct Stack* stack)
{
   if (isEmpty(stack))
      return INT_MIN;
   return stack->array[stack->top];
}

// Driver program to test above functions
int main()
{
   struct Stack* stack = createStack(100);
     push(stack, 10);
   push(stack, 20);
   push(stack, 30);
    printf("%d popped from stack\n", pop(stack));
     return 0;
}
```

3.5 Check Your Progress Questions

1. Define Stack.
2. List out the operations on stack.

3.6 Answers to Check Your Progress Questions

1. 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.
2. Push(), Pop(), isEmpty(), isFull()

3.5 Summary

- A stack can be organized (represented) in the memory either as an array or as a singly-linked list.
- Though array representation is a simple technique, it provides less flexibility and is not very efficient with respect to memory utilization.
- When a stack is organized as an array, a variable named Top is used to point to the top element of the stack.
- An array representation of a stack is static, but linked list representation is dynamic in nature
- When a stack is organized as an array, a variable named Top is used to point to the top element of the stack. Initially, the value of Top is set as -1 to indicate an empty stack.
- Overflow occurs when a stack is full and there is no space for a new element and an attempt is made to push a new element.
- When a stack is organized as an array, a variable named Top is used to point to the top element of the stack.
- Similarly, before removing the top element from the stack, it is necessary to check the condition of underflow.
- To POP (or remove) an element from a stack, the element at the top of the stack is assigned to a local variable and then Top is decremented by one.

3.7 Key Words

- **Overflow**: It occurs when a stack is full and there is no space for a new element and an attempt is made to push a new element.
- **Stack**: It is an abstract data type that serves as a collection of elements, with two principal operations: push, which adds an element to the collection, and pop, which removes the most recently added element that was not yet removed.
- **Underflow**: An error condition that occurs when an item is called for from the stack, but the stack is empty. Contrast with stack overflow.

3.8 Self-Assessment Questions and Exercise

Short-Answer Questions

1. What is a stack?

2. How is a stack organized?
3. What do you mean by overflow in a stack?
4. Explain stack underflow.
5. Give the algorithm to push elements into stack.
6. Give the algorithm to pop elements from stack.

Long-Answer Questions

1. Write a program to implement a stack as an array.
2. How can you insert elements in a stack? Explain.
3. Write a note on the operations on stack.

3.13 Further Readings

Storer, J.A. 2012. An Introduction to Data Structures and Algorithms. New York: Springer Publishing.
Preiss, Bruno. 2008. Data Structures and Algorithms with Object-Oriented Design Patterns in C++. London: John Wiley and Sons.
Pandey, Hari Mohan. 2009. Data Structures and Algorithms. New Delhi: Laxmi
Publications.
Goodrich Michael, Tamassia Roberto and Michael H. Goldwasser. 2014. Data
Structures and Algorithms in Java. London: John Wiley and Sons.
McMillan, Michael. 2007. Data Structures and Algorithms Using C#. Cambridge, UK: Cambridge University Press.

UNIT – IV REPRESENTATION OF STACK

NOTES

Structure

4.1 Introduction

A stack is an abstract data type and it serves as a collection of elements, with two primary principal operations: push, and pop. Push adds an element to the collection and pop removes the most recently added element. As in this unit, we will see the application and implementation of stacks.

4.2 Objectives

After going through this unit, you will be able to:

- Explain what are stacks
- Discuss the application of stacks
- Analyse the implementation of stacks

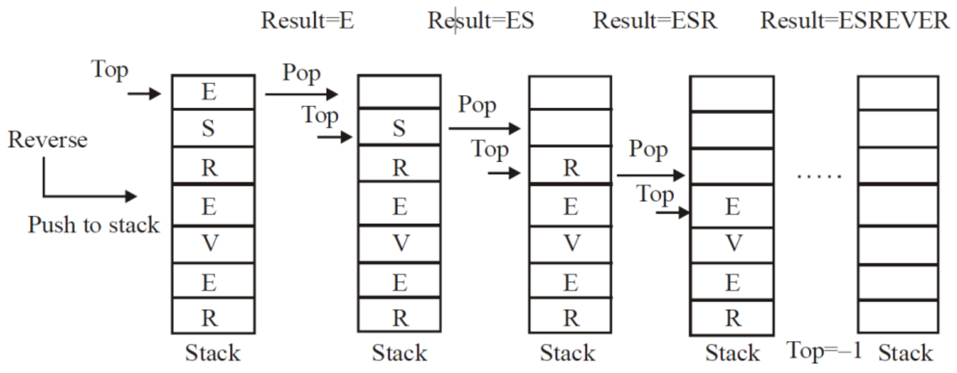
4.2 Application and Implementation of Stack

Stacks are used where the last-in-first-out principle is required like reversing strings, checking whether the arithmetic expression is properly parenthesized, converting infix notation to postfix and prefix notations, evaluating postfix expressions, implementing recursion and function calls, etc.

Reversing Strings

A simple application of stacks is reversing strings. To reverse a string, the characters of a string are pushed onto a stack one by one as the string is read from left to right. Once all the characters of the string are pushed onto the stack, they are popped one by one. Since the character last pushed in comes out first, subsequent POP operations result in reversal of the string.

For example, to reverse a string 'REVERSE', the string is read from left to right and its characters are pushed onto a stack, starting from the letter R, then E, V, E, and so on



Algorithm for string reversal

```
reversal(s, str)
    1. Set i = 0
    2. While(i < length_of_str)
        Push str[i] onto the stack
        Set i = i + 1
    End While
    3. Set i = 0
    4. While(i < length_of_str)
        Pop the top element of the stack and store it in str[i]
        Set i = i + 1
    End While
    5. Print "The reversed string is: ", str
    6. End
```

4.3.1 Converting Infix Notation to Postfix and Prefix or Polish Notations

The general way of writing arithmetic expressions is known as infix notation where the binary operator is placed between two operands on which it operates. For simplicity, expressions containing unary operators have been ignored.

For example, the expressions 'a+b' and '(a–c)*d', '[(a+b)*(d/f)–f]' are in infix notation. The order of evaluation of these expressions depends on the parentheses and the precedence of operators. For example, the order of evaluation of the expression '(a+b)*c' is different from that of 'a+(b*c)'. As a result, it is difficult to evaluate an expression in an infix notation. Thus, the arithmetic expressions in the infix notation are converted to another notation which can be easily evaluated by a computer system to produce a correct result.

A Polish mathematician Jan Lukasiewicz suggested two alternative notations to represent an arithmetic expression. In these notations, the operators can be written either before or after the operands on which they operate. The notation in which an operator occurs before its operands is known as the prefix notation (also known as Polish notation). For example,

the expressions '+ab' and '*– acd' are in prefix notation. On the other hand, the notation in which an operator
occurs after its operands is known as the postfix notation (also known as Reverse Polish or suffix notation). For example, the expressions 'ab+' and 'ac–d*' are in postfix notation.

Advantages of Prefix notation

A characteristic feature of prefix and postfix notations is that the order of evaluation of expression is determined by the position of the operator and operands in the expression. That is, the operations are performed in the order in which the operators are encountered in the expression. Hence, parentheses are not required for the prefix and postfix notations. Moreover, while evaluating the expression, the precedence of operators is insignificant. As a result, they are compiled faster than the expressions in infix notation.

Conversion of infix to prefix notation

In infix to prefix conversion the infix notation is scanned in reverse order, that is, from right to left. Therefore, the stack in this case stores the operators and the closing (right) parenthesis.

Algorithm for infix to prefix conversion

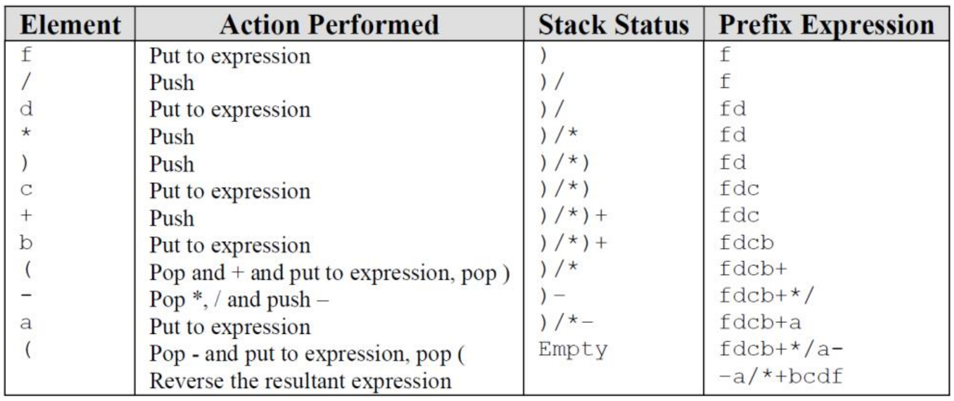
```
infixtoprefix(s, infix, prefix)
        1. Set i = 0
        2. While (i < number_of_symbols_in_infix)
                If infix[i] is a whitespace or comma
                Set i = i + 1 go to step 2
                If infix[i] is an operand, add it to prefix
                Else If infix[i] = ')', push it onto the stack
                Else If infix[i] is an operator, follow these steps:
                        i. For each operator on the top of stack whose
                precedence is greater
                        than or equal to the precedence of the current
                        operator, pop the operator from stack and add it
                        to prefix
                        ii. Push the current operator onto the stack Else If
                        infix[i] = '(', follow these steps:
                                i. Pop each operator from top of the stack
                                and add it to prefix until ')' is
                                encountered in the stack
                                ii. Remove ')' from the stack and do not
                        add it to prefix
                End If
                Set i = i + 1
        End While
        3. Reverse the prefix expression
        4. End
```

NOTES

For example, consider the conversion of the following infix expression to a prefix expression:

a-(b+c)*d/f

The step-wise conversion of the expression **a-(b+c)*d/f** into its equivalent prefix expression is shown. Note that initially ')' is pushed onto the stack, and '(' is inserted in the beginning of the infix expression. Since the infix expression is scanned from right to left, but elements are inserted in the resultant expression from left to right, the prefix expression needs to be reversed.



Element	Action Performed	Stack Status	Prefix Expression
f	Put to expression	)	f
/	Push	) /	f
d	Put to expression	) /	fd
*	Push	) /*	fd
)	Push	) /*)	fd
c	Put to expression	) /*)	fdc
+	Push	) /*)+	fdc
b	Put to expression	) /*)+	fdcb
(	Pop and + and put to expression, pop)	) /*	fdcb+
–	Pop *, / and push –	) –	fdcb+*/
a	Put to expression	) /*–	fdcb+a
(	Pop - and put to expression, pop (	Empty	fdcb+*/a-
	Reverse the resultant expression		-a/*+bcdf

The equivalent prefix expression is –a/*+bcdf.

4.4 Check Your Progress

1) What is a characteristic feature of prefix and postfix notation?
2) Where are stacks used?
3) Where is stack used during evaluation?

4.5 Summary

- Stacks are used where the last-in-first-out principle is required like reversing strings.
- A simple application of stacks is reversing strings. To reverse a string, the characters of a string are pushed onto a stack one by one as the string is read from left to right.
- Once all the characters of the string are pushed onto the stack, they are popped one by one.
- Since the character last pushed in comes out first, subsequent POP operations result in reversal of the string.
- The general way of writing arithmetic expressions is known as infix notation where the binary operator is placed between two operands on which it operates.
- A characteristic feature of prefix and postfix notations is that the order of evaluation of expression is determined by the position of the operator and operands in the expression.
- To convert an arithmetic expression from an infix notation to a postfix notation, the precedence and associativity rules of operators should always kept in mind.

- The conversion of an infix expression to a prefix expression is similar to the conversion of infix to postfix expression.
- In a computer system, when an arithmetic expression in an infix notation needs to be evaluated, it is first converted into its postfix notation.
- Evaluation of postfix expressions is also implemented through stacks. Representation of Stack
- Since the postfix expression is evaluated in the order of appearance of operators, parentheses are not required in the postfix expression.
- During evaluation, a stack is used to store the intermediate results of evaluation.
- Since an operator appears after its operands in a postfix expression, the expression is evaluated from left to right.
- If the element is an operand, it is pushed onto the stack.
- If two or more stacks are needed in a program, then it can be accomplished in two ways.

4.6 Keywords

- **Stack**: It is an abstract data type that serves as a collection of elements, with two principal operations- push and pop.
- **Reversing Strings**: It is a simple application of stacks. To reverse a string, the characters of a string are pushed onto a stack one by one as the string is read from left to right.
- **Prefix Notation**: It is also called polish notation. It is simply a mathematical notation where the operators precede the operands.

4.7 Self-assessment questions and exercise

Short-Answer Questions

1. Write the algorithm for reversing a string.
2. Write a short note on stacks.
3. Discuss the application of stacks.
4. Discuss the advantages of polish notation.

Long-Answer Questions

1. What do you mean by implementation of stack? Discuss in detail.
2. Write a program to reverse a given string using stacks.
3. Write a detailed note on Conversion of infix to prefix notation.
4. State and explain an algorithm to convert an expression from infix notation to prefix notation.

4.8 Further Readings

Storer, J.A. 2012. An Introduction to Data Structures and Algorithms. New

York: Springer Publishing.
Preiss, Bruno. 2008. Data Structures and Algorithms with Object-Oriented Design Patterns in C++. London: John Wiley and Sons.
Pandey, Hari Mohan. 2009. Data Structures and Algorithms. New Delhi: Laxmi
Publications.
Goodrich Michael, Tamassia Roberto and Michael H. Goldwasser. 2014. Data
Structures and Algorithms in Java. London: John Wiley and Sons.
McMillan, Michael. 2007. Data Structures and Algorithms Using C#.
Cambridge, UK: Cambridge University Press.

UNIT- V QUEUES

NOTES

Structure

5.1 Introduction

A Queue is an abstract data structure which is somewhat similar to Stacks. But unlike stacks, a queue is open at both its ends. One end of a queue is always used to ins rt data (called enqueue) and the other is used to remove data (called dequeue). Queue follows the basic and simple First-In-First-Out methodology, which means that the data item stored first will be accessed first.

5.2 Objectives

After going through this unit, you will be able to:

- Understand queues
- Discuss the representation of queues
- Analyse circular queues and deque
- Explain about priority queues
- List the applications of queue.

5.3 Queues

A queue is a linear data structure in which a new element is inserted at one end and an element is deleted from the other end. The end of the queue from which the element is deleted is known as the Front and the end at which a new element is added is known as the Rear.

5.3.1 Operations on Queues

Operations	Description
enqueue()	This function defines the operation for adding an element into queue.
dequeue()	This function defines the operation for removing an element from queue.
init()	This function is used for initializing the queue.
Front	Front is used to get the front data item from a queue.
Rear	Rear is used to get the last item from a queue.

5.4 Representation of Queues

Like stacks, queues can be represented in the memory by using an array or a singly linked list. We will discuss how a queue can be implemented using an array.

Array Implementation of a Queue

When a queue is implemented as an array, all the characteristics of an array are applicable to the queue. Since an array is a static data structure, the array representation of a queue requires the maximum size of the queue to be predetermined and fixed. As we know that a queue keeps on changing as elements are inserted or deleted, the maximum size should be large enough for a queue to expand or shrink.

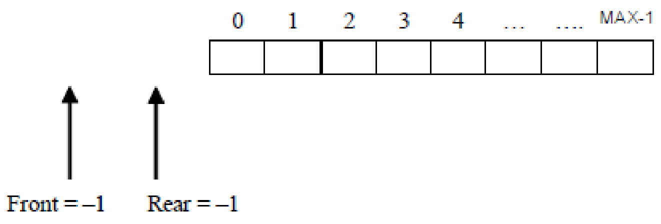
```
struct queue
{
        int item[MAX];
        int Front;
        int Rear;
};
```

The representation of a queue as an array needs an array to hold the elements of the queue and two variables Rear and Front to keep track of the rear and the front ends of the queue, respectively. Initially, the value of Rear and Front is set to –1 to indicate an empty queue. Before we insert a new element in the queue, it is necessary to test the condition of overflow. A queue is in a condition of overflow (full) when Rear is equal to MAX–1, where MAX is the maximum size of the array. If the queue is not full, the insert operation can be performed. To insert an element in the queue, Rear is incremented by one and the element is inserted at that position.

Similarly, before we delete an element from a queue, it is necessary to test the condition of underflow. A queue is in the condition of underflow (empty) when the value of Front is –1. If a queue is not empty, the delete operation can be performed. To delete an element from a queue, the element referred by Front is assigned to a local variable and then Front is incremented by one. The total number of elements in a queue at a given point of time can be calculated from the values of Rear and Front given as follows:

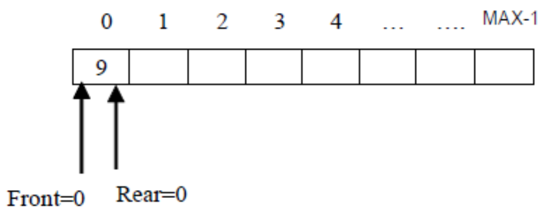
Number of elements = Rear – Front + 1

To understand the implementation of a queue as an array in detail, consider a queue stored in the memory as an array named Queue that has MAX as its maximum number of elements. Rear and Front store the indices of the rear and front elements of Queue. Initially, Rear and Front are set to –1 to indicate an empty queue.



(a) An Empty Queue

Whenever a new element has to be inserted in a queue, Rear is incremented by one and the element is stored at Queue[Rear]. Suppose an element 9 is to be inserted in the queue. In this case, the rear is incremented from –1 to 0 and the element is stored at Queue[0]. Since it is the first element to be inserted, Front is also incremented by one to make it to refer to the first element of the queue. For subsequent insertions, the value of Rear is incremented by one and the element is stored at Queue[Rear].

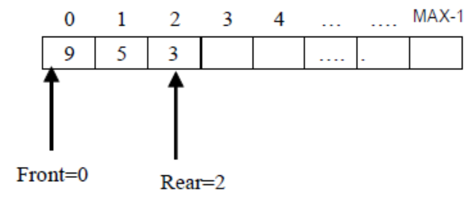


(b) Queue after Inserting the First Element

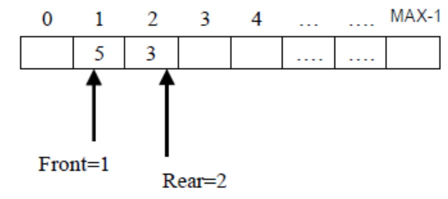
However, Front remains unchanged. Observe that the front and rear elements of the queue are the first and last elements of the list,

NOTES

respectively. Whenever, an element is to be deleted from a queue, Front is incremented by one. Suppose that an element is to be deleted from Queue. Then, here it must be 9. It is because the deletion is always made at the front end of a queue. Deletion of the first element results in the queue as shown in Similarly, deletion of the second element results in the queue.

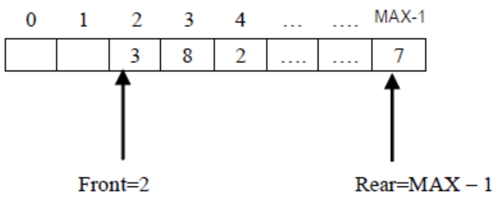


(c) *Queue after Inserting a few Elements*



(d) *Queue after Deleting the First Element*

Observe that after deleting the second element from the queue, the values of Rear and Front are equal. Here, it is apparent that when values of Front and Rear are equal other than –1, there is only one element in the queue. When this only element of the queue is deleted, both Rear and Front are again made equal to –1 to indicate an empty queue. Further, suppose that some more elements are inserted and Rear reaches the maximum size of the array. This means that the queue is full and no more elements can be inserted in it even though the space is vacant on the left of the Front.



(*f*) *Queue having Vacant Space though Rear = MAX – 1*

We can use a one-dimensional array A of size n to represent a queue in memory. To manipulate the elements in the queue, we can use 2 pointers called as Front and Rear (f and r respectively). These pointers are initialized to -1. Basic Conditions:

Empty Queue : f = -1, r = -1

Full Queue : r = n-1, where n is the size of the queue.

Single element in Queue: f = r

Algorithm for enqueue

```
Algorithm Enqueue( )
{
        if ( r ==n-1)
        {
```

NOTES

```
            **Print"queue is full"**
            **}**
           **else**
           **{**
                **If(f= =-1)**
               **{**
                  **f=0;**
               **}**
               **r=r+1;**
               **q[r]=x**
          **}**
**}**
```

Algorithm for dequeue

```
Algorithm Dequeue()
{
           if ( f==-1 && r==-1)
            {
             Print"queue is empty"
            }
          else
         {
         Y=q[p]
             If(f==r)
           {
             F=-1;
            }
         else
         {
             f=f+1;
         }
     }
 }
```

Algorithm Display()

```
{
           if ( f ==-1 && r==-1 )
           {
             Print"queue is empty"
            }
          else
          {
             for(i=f;i<=r;i++)
            {
               Print q[i]
            }
          }
}
```

<u>Algorithm IsEmpty()</u>

```
{
        if( f ==  -1 )
        {
                print " Queue is Empty "
        }
        else
        {
                print "Queue is not Empty"
        }
}
```

<u>Algorithm IsFull()</u>

```
{
        if ( r == n-1 )
        {
                print " Queue is full "
        }
        else
        {
                print " Queue is not full "
        }
}
```

<u>Front()</u>

This algorithm specifies the position of Front of the queue and says what element is stored there.

Algorithm Front()

```
{
        if( f == -1)
                Print "Queue Underflow"
        else
                Print " Element A[f] is in position front = f"
}
```

<u>Rear()</u>

This algorithm specifies the position of Rear of the queue and says what element is stored there.

Algorithm Rear()

```
{
        if( f == -1)
                Print "Queue Underflow"
        else
                Print " Element A[r] is in position front = r"
}
```

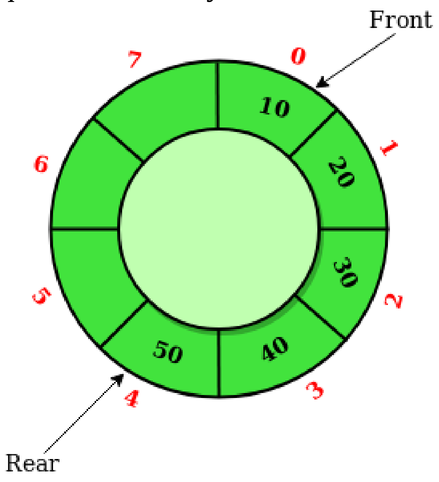
NOTES

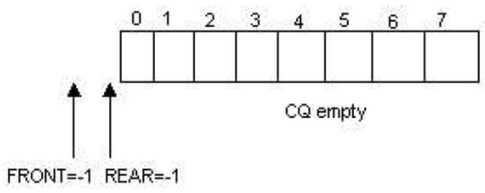
Disadvantage of Ordinary Queue / Need for Circular Queue

In an ordinary queue, after a few deletions, some space is available in the beginning of the list. But, the space can't be used for inserting new data just because the queue full condition i.e., (Rear = n-1) is satisfied. This problem is solved in the Circular Queue.

5.5 Circular Queue

Once the value of the rear reaches the maximum size of the queue, no more elements can be inserted. However, there may be the possibility that the space on the left of the front index is vacant. Hence, in spite of space on the left of front being empty, the queue is considered full. This wastage of space in the array implementation of a queue can be avoided by shifting the elements to the beginning of the array if space is available. In order to do this, the values of the Rear and Front indices have to be changed accordingly. However, this is a complex process and difficult to implement. An alternative solution to this problem is to implement a queue as a circular queue.
The array implementation of a circular queue is similar to the array implementation of the queue. The only difference is that as soon as the rear index of the queue reaches the maximum size of the array, Rear is reset to the beginning of the queue, provided it is free. The circular queue is full only when all the locations in the array are occupied.





(a) *An Empty Queue*

To understand the operations on a circular queue, consider a circular queue represented in the memory by the array CQueue[MAX]. Rear and Front are used to store the indices of the rear and front elements of

CQueue, respectively. Initially, both Rear and Front are set to NULL to indicate an empty queue.

Whenever an element is to be inserted in a circular queue, Rear is incremented by one. However, if the value of the rear index is MAX-1, instead of incrementing Rear, it is reset to the first index of the array if space is available in the beginning. Hence, if any location to the left of the front index is empty, the elements can be added to the queue at an index starting from 0. A queue is considered full in the following cases:

- When the value of Rear equals the maximum size of the array and Front is at the beginning of the array.
- When the value of Front is one more than the value of Rear.

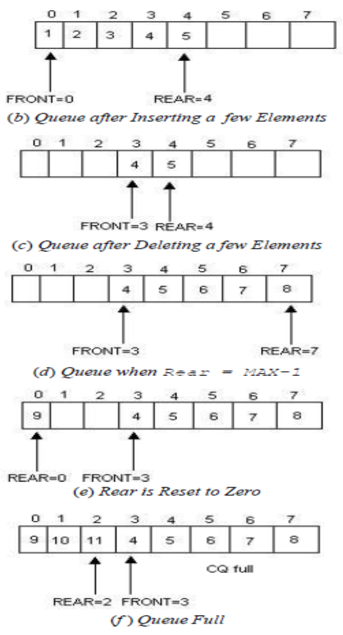
Algorithm for circular enqueue

```
enqueue(q, val)

1.  If ((q->Rear = MAX-1 AND q->Front = 0) OR
    (q->Rear + 1 = q->Front))
        Print "Overflow: Queue is full!" and go to step 5
   End If //check if circular queue is full
2. If q->Rear = MAX-1 // check if rear is MAX-1
       Set q->Rear = 0
   Else
      Set q->Rear = q->Rear + 1 //increment rear by one
   End If
3. Set q->CQueue[q->Rear] = val //val is the value to be inserted in the
queue
4. If q->Front = -1 //check if queue is empty
     Set q->Front = 0
    End If
5. End
```

Algorithm for circular dequeue

```
deque(q)
        1. If q->Front = -1
                Print "Underflow: Queue is empty!"
                Return 0 and go to step 5
        End If
        2. Set del_val = q->CQueue[q->Front] //del_val is the value to be
deleted
        3. If q->Front = q->Rear // check if there is one element in the
queue
                Set q->Front = q->Rear = -1
        Else
                If q->Front = MAX-1
                        Set q->Front = 0
                Else
                        Set q->Front = q->Front +1
                End If
        End If
        4. Return del_val
```

5. End



(b) Queue after Inserting a few Elements

(c) Queue after Deleting a few Elements

(d) Queue when Rear = MAX−1

(e) Rear is Reset to Zero

(f) Queue Full

5.6 Applications of queues

There are numerous applications of queues in computer science. Various real-life applications such as railway ticket reservation and the banking system are implemented using queues. One of the most useful applications of a queue is in simulation. Another application of a queue is in the operating system, to implement various functions like CPU scheduling in a multiprogramming environment, device management (printer or disk), etc. Besides, there are several algorithms like level order traversal of binary tree, etc., that use queues to solve problems efficiently.

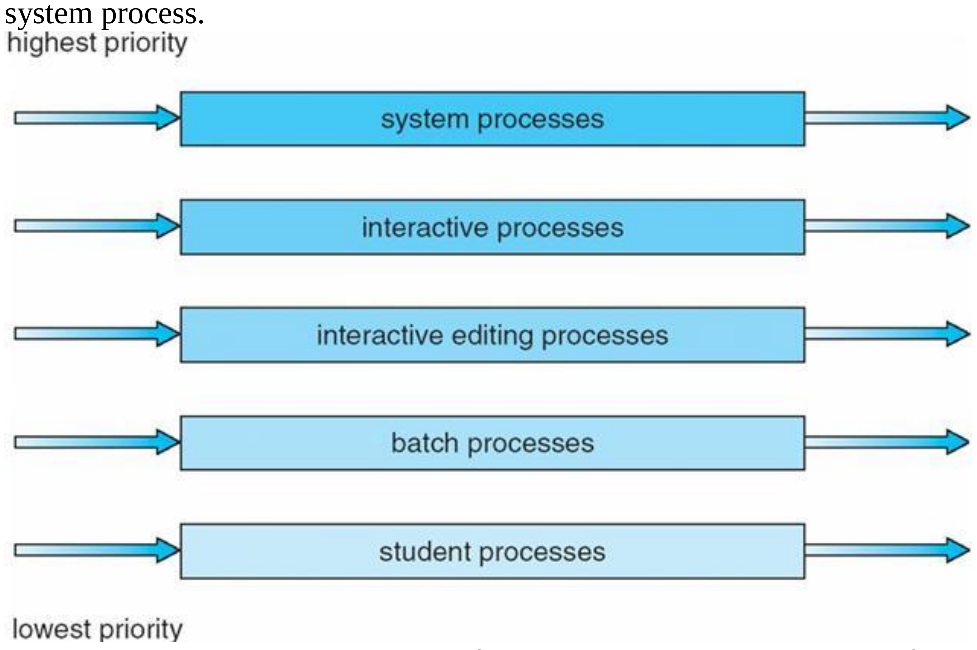
Simulation

Simulation is the process of modelling a real-life situation through a computer program. Its main use is to study a real-life situation without actually making it occur. It is mainly used in areas like military operations, scientific research, etc., where it is expensive or dangerous to experiment with the real system. In simulation, corresponding to each object and action, there is a counterpart in the program. The objects that are studied are represented as data and the actions are represented as operations on the data. By supplying different data, we can observe the result of the program. If the simulation is accurate, the result of the program represents the behaviour of the actual system accurately. Consider a ticket reservation system having four counters. If a customer arrives at time ta and a counter is free, then the customer will get the ticket immediately. However, it is not always possible that a counter is free. In that case, a new customer goes to the queue having fewer customers. Assume that the time required to issue the ticket is t. Then the total time spent by the customer is equal to the time t (time required to issue the ticket) plus the time spent waiting in line. The average time spent in the line by the customer can be computed by a program simulating the customer action. This program can be implemented using a queue, since while one customer is being serviced, the others are kept waiting.

CPU Scheduling in a Multiprogramming Environment

As we know, in a multiprogramming environment, multiple processes run concurrently to increase CPU utilization. All the processes that are residing in the memory and are ready to execute are kept in a list referred to as a ready queue. It is the job of the scheduling algorithm to select a process from the processes and allocate the CPU to it.

Let us consider a multiprogramming environment where the processes are classified into three different groups, namely system processes, interactive processes and batch processes. Some priority is associated with each group of processes. The system processes have the highest priority, whereas the batch processes have the least priority. To implement a multiprogramming environment, a multi-level queue scheduling algorithm is used. In this algorithm, the ready queue is partitioned into multiple queues. The processes are assigned to the respective queues. The higher priority processes are executed before the lower priority processes. For example, no batch process can run unless all the system processes and interactive processes are executed. If a batch process is running and a system process enters the queue, then batch process would be pre-empted to execute this system process.



In this algorithm, the processes of a lower priority may starve if the number of processes in a higher-priority queue is high. Starvation can be prevented by two ways. One way is to time-slice between the queues, that is, each queue gets a certain interval of time. Another way is using a multi-level feedback queue algorithm. In this algorithm, processes are not assigned permanently to a queue; instead, they are allowed to move between the queues. If a process uses too much CPU time, it is moved to lower priority. Similarly, a process that has been waiting for too long in a lower-priority queue is moved to the higher-priority queue. To implement multiple programming environments, a priority queue using multiple queues can be used.

Round Robin Algorithm

The Round Robin algorithm is one of the CPU scheduling algorithms designed for time-sharing systems. In this algorithm, the CPU is allocated to a process for a small time interval called time quantum (generally from 10 to 100 milliseconds). Whenever a new process enters, it is inserted at the end of the ready queue. The CPU scheduler picks the first process from the ready queue and processes it until the time quantum elapses. Then, the CPU switches to the next process in the queue and the first process is inserted at the end of the queue if it has not been finished. If the process is finished before the time quantum, the process itself releases the CPU voluntarily and the process gets deleted from the ready queue. This process continues until all the processes are finished. When a process is finished, it is deleted from the queue. To implement the Round Robin algorithm, a circular queue can be used. Suppose there are n processes, such as P1, P2, …, Pn served by the CPU. Different processes require different execution time. Suppose, sequence of processes arrivals is arranged according to their subscripts, i.e., P1 comes first, then P2. Therefore, Pi comes after Pi□1 where 1< i □ n. Round Robin algorithm first decides a small unit of time called time quantum or time slice represented by τ. A time quantum generally starts from 10 to 100 milliseconds. CPU starts services from P1. Then, P1 gets CPU for τ instant of time; afterwards CPU switches to process P2 and so on. Now, during time-sharing, if a process finishes its execution before the finding of its time quantum, the process then simply releases the CPU and the next process waiting will get the CPU immediately. When CPU reaches the end of time quantum of Pn it returns to P1 and the same process will be repeated.

5.7 Check Your Progress Questions

1) What is a priority queue?
2) What is simulation?

5.8 Answers to Check Your Progress Questions

1) A priority queue is a type of queue in which each element is assigned a priority and the elements are added or removed according to that priority.
2) Simulation is the process of modelling a real-life situation through a computer program.

5.9 Summary

- A queue is a linear data structure in which a new element is inserted at one end and an element is deleted from the other end.
- The end of the queue from which the element is deleted is known as the Front and the end at which a new element is added is known as the Rear.
- Before inserting a new element in the queue, it is necessary to check whether there is space for the new element.
- If there is no element in the queue, the queue is said to be in the condition of underflow.
- Like stacks, queues can be represented in the memory by using an array or a singly linked list.
- When a queue is implemented as an array, all the characteristics of an array are applicable to the queue.
- Since an array is a static data structure, the array representation of a queue requires the maximum size of the queue to be predetermined and fixed.
- Whenever a new element has to be inserted in a queue, Rear is incremented by one and the element is stored at Queue[Rear].
- Whenever, an element is to be deleted from a queue, Front is incremented by one.
- Whenever an element is deleted from the queue, a temporary pointer is created, which is made to point to the node pointed to by Front.
- The info and next fields of each node represent the element of the queue and a pointer to the next element in the queue, respectively.
- In the case of a queue represented as an array, once the value of the rear reaches the maximum size of the queue, no more elements can be inserted.
- The array implementation of a circular queue is similar to the array implementation of the queue. The only difference is that as soon as the rear index of the queue reaches the maximum size of the array, Rear is reset to the beginning of the queue, provided it is free.
- A priority queue is a type of queue in which each element is assigned a priority and the elements are added or removed according to that priority.
- In the multiple queue representation of a priority queue, a separate queue for each priority is maintained.
- If the size of each queue is the same, then instead of multiple one-dimensional arrays, a single two-dimensional array can be used where the row number shows the priority and the column number shows the position of the element within the queue.
- One of the most useful applications of a queue is in simulation.
- Simulation is the process of modelling a real-life situation through a computer program.

- If the simulation is accurate, the result of the program represents the behaviour of the actual system accurately.
- To implement a multiprogramming environment, a multi-level queue scheduling algorithm is used.
- If a batch process is running and a system process enters the queue, then batch process would be pre-empted to execute this system process.
- To implement multiple programming environments, a priority queue using multiple queues can be used.

5.10 Key Words

- **Input Restricted Deque**: It allows insertion of elements only at one end but deletion can be done at both ends.
- **Output Restricted Deque**: It allows deletion of elements only at one end but insertion can be done at both ends.
- **Circular queue**: The useful property of a circular buffer is that it does not need to have its elements shuffled around when one is consumed. (If a non-circular buffer were used then it would be necessary to shift all elements when one is consumed.) In other words, the circular buffer is well-suited as a FIFO buffer while a standard, non-circular buffer is well suited as a LIFO buffer.

5.11 Self-Assessment Questions and Exercises

Short-Answer Questions

1. What are queues?
2. Write a short note about representation of queues.
3. What is a circular queue?
4. List few basic operations performed on queues.

Long-Answer Questions

1. Write a program to implement a queue as an array.
2. Differentiate between circular queue and deque.
3. Write a program to implement a circular queue.
4. Write a program to illustrate the insertion and deletion operations on a simple deque.

5.12 Further Readings

Storer, J.A. 2012. An Introduction to Data Structures and Algorithms. New York: Springer Publishing.

Preiss, Bruno. 2008. Data Structures and Algorithms with Object-Oriented Design Patterns in C++. London: John Wiley and Sons.

Pandey, Hari Mohan. 2009. Data Structures and Algorithms. New Delhi: Laxmi Publications.

Goodrich Michael, Tamassia Roberto and Michael H. Goldwasser. 2014. Data Structures and Algorithms in Java. London: John Wiley and Sons.

NOTES

UNIT- VI LIST

Structure

6.1 Introduction

A list or sequence is an abstract data type that represents a countable number of ordered values, where the same value may occur more than once. A linked list is a sequence of data structures, which are held together by links. A Linked List is a sequence of links which contains items. Each link contains a connection to another link. A Linked list is the second most-used data structure after an array.

6.2 Objectives

After going through this unit, you will be able to:

- Discuss linked list
- Explain singly-linked list
- Analyse doubly-linked list
- Understand merging list

6.3 Merging list

Merge lists or algorithms are a family of algorithms that take multiple sorted lists as a medium of input and in turn produce a single list as an output. This output contains all the elements of the inputs lists in a neatly sorted out order. These algorithms are then used as subroutines in various sorting algorithms, which most famously merge sort.

Dynamic data structure

A dynamic data structure is one in which the memory for elements is allocated dynamically during run-time. The successive elements of a dynamic data structure need not be stored in contiguous memory locations but they are still linked together by means of some linkages or references.

Whenever a new element is inserted, the memory for the same is allocated dynamically and is linked to the data structure. The elements can be inserted as the long as memory is available. Thus there is no upper limit on the number of elements in the data structure. Similarly, whenever an element is deleted from the data structure, memory is de-allocated so that it can be reused in the future.

6.4 Linked list

A linked list is a linear collection of homogeneous elements called nodes. Successive nodes of a linked list need not occupy adjacent memory locations. The linear order between nodes is maintained by means of pointers. In linked lists, insertion or deletion of nodes do not require shifting of existing nodes as in the case of arrays; they can be inserted or deleted merely by adjusting the pointers or links.

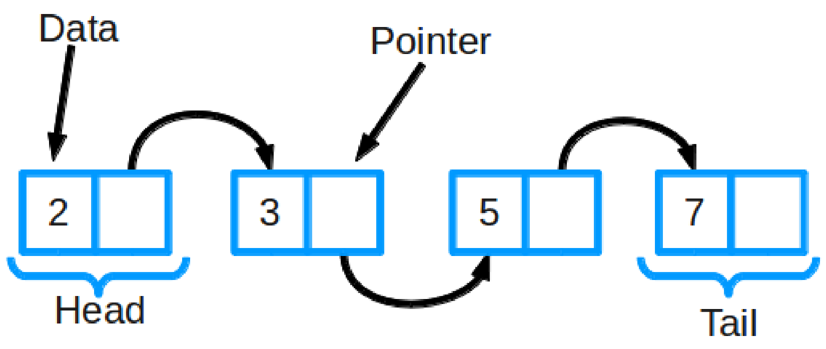
Types of linked list

Depending on the number of pointers in a node or the purpose for which the pointers are maintained, a linked list can be classified into various types such as

- singly-linked list
- circular-linked list
- doubly-linked list

6.5 Singly-Linked Lists

A singly-linked list is also known as a linear linked list. In it, each node consists of two fields, viz. 'data' and 'next'. The 'data' field contains the data and the 'next' field contains the address of memory location where the subsequent node is stored. The last node of the singly-linked list contains NULL in its 'next' field which indicates the end of the list.



A linked list contains a list pointer variable 'Head' that stores the address of the first node of the list. In case, the 'Head' node contains NULL, the list is called an empty list or a null list.

NOTES

Operations

Since each node of the list contains only a single pointer pointing to the next node, not to the previous node—allowing traversing in only one direction—hence, it is also referred to as a one-way list.

A number of operations can be performed on singly-linked lists. These operations include traversing, searching, inserting and deleting nodes, reversing, sorting, and merging linked lists. Before implementing these operations, it is important to understand how the node of a linked list is created.

The structure of a node in linked list is as follows:

```
typedef struct node
{
        int data; /*to store integer type data*/
        struct node *next; /*to store a pointer to nextnode*/
} Node;
Node *nptr;
```

Algorithm for creating a new node

```
create_node()
    1. Allocate memory for nptr //nptr is a pointer to new node
    2. If nptr = NULL
           Print "Overflow: Memory not allocated!" and go to step 7
    End If
    3. Read item //item is the value to be inserted in the new node
    4. Set nptr->info = item
    5. Set nptr->next = NULL
    6. Return nptr //returning pointer nptr
7. End
```

The following operations are performed on a Single Linked List

- Insertion
- Deletion
- Display

Before we implement actual operations, first we need to set up an empty list. First, perform the following steps before implementing actual operations.

Step 1 - Include all the **header files** which are used in the program.
Step 2 - Declare all the **user defined functions**.
Step 3 - Define a **Node** structure with two members **data** and **next**
Step 4 - Define a Node pointer **'head'** and set it to **NULL**.
Step 5 - Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Insertion

In a single linked list, the insertion operation can be performed in three ways. They are as follows...

Inserting At Beginning of the list
Inserting At End of the list
Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the single linked list...

Step 1 - Create a **newNode** with given value.
Step 2 - Check whether list is **Empty** (**head** == **NULL**)
Step 3 - If it is **Empty** then, set **newNode→next** = **NULL** and **head** = **newNode**.
Step 4 - If it is **Not Empty** then, set **newNode→next** = **head** and **head** = **newNode**.

Inserting At End of the list

We can use the following steps to insert a new node at end of the single linked list...

Step 1 - Create a **newNode** with given value and **newNode → next** as **NULL**.
Step 2 - Check whether list is **Empty** (**head** == **NULL**).
Step 3 - If it is **Empty** then, set **head** = **newNode**.
Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize
with **head**.
Step 5 - Keep moving the **temp** to its next node until it reaches to the last node in
the list (until **temp → next** is equal to **NULL**).
Step 6 - Set **temp → next** = **newNode**.

Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the single linked list...

Step 1 - Create a **newNode** with given value.
Step 2 - Check whether list is **Empty** (**head** == **NULL**)
Step 3 - If it is **Empty** then, set **newNode → next** = **NULL** and **head** = **newNode**.
Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize
with **head**.
Step 5 - Keep moving the **temp** to its next node until it reaches to the node after
which we want to insert the newNode (until **temp1 → data** is equal
to **location**, here location is the node value after which we want to insert
the newNode).
Step 6 - Every time check whether **temp** is reached to last node or not. If it is
reached to last node then display **'Given node is not found in the list!!!
Insertion not possible!!!'** and terminate the function. Otherwise move
the **temp** to next node.

Step 7 - Finally, Set '**newNode → next** = **temp → next**' and '**temp →**
next = **newNode**'

NOTES

Deletion

In a single linked list, the deletion operation can be performed in three ways. They are as follows...

Deleting from Beginning of the list

Deleting from End of the list

Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the single linked list...

Step 1 - Check whether list is **Empty** (**head** == **NULL**)

Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

Step 3 - If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.

Step 4 - Check whether list is having only one node (**temp → next** == **NULL**)

Step 5 - If it is **TRUE** then set **head** = **NULL** and delete **temp** (Setting **Empty** list conditions)

Step 6 - If it is **FALSE** then set **head** = **temp → next**, and delete **temp**.

Deleting from End of the list

We can use the following steps to delete a node from end of the single linked list...

Step 1 - Check whether list is **Empty** (**head** == **NULL**)

Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and
terminate the function.

Step 3 - If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and
initialize '**temp1**' with **head**.

Step 4 - Check whether list has only one Node (**temp1 → next** == **NULL**)

Step 5 - If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate the
function. (Setting **Empty** list condition)

Step 6 - If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next
node. Repeat the same until it reaches to the last node in the list.
(until **temp1 → next** == **NULL**)

Step 7 - Finally, Set **temp2 → next** = **NULL** and delete **temp1**.

NOTES

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the single linked list...

Step 1 - Check whether list is **Empty** (**head** == **NULL**)

Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

Step 3 - If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.

Step 4 - Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.

Step 5 - If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.

Step 6 - If it is reached to the exact node which we want to delete, then check whether list is having only one node or not

Step 7 - If list has only one node and that is the node to be deleted, then set **head** = **NULL** and delete **temp1** (**free(temp1)**).

Step 8 - If list contains multiple nodes, then check whether **temp1** is the first node in the list (**temp1 == head**).

Step 9 - If **temp1** is the first node then move the **head** to the next node (**head = head → next**) and delete **temp1**.

Step 10 - If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == NULL**).

Step 11 - If **temp1** is last node then set **temp2 → next** = **NULL** and delete **temp1** (**free(temp1)**).

Step 12 - If **temp1** is not first node and not last node then set **temp2 → next** = **temp1 → next** and delete **temp1** (**free(temp1)**).

Displaying a Single Linked List

We can use the following steps to display the elements of a single linked list...

Step 1 - Check whether list is **Empty** (**head** == **NULL**)

Step 2 - If it is **Empty** then, display **'List is Empty!!!'** and terminate the function.

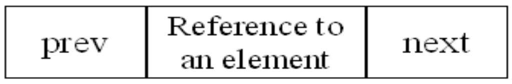
Step 3 - If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.

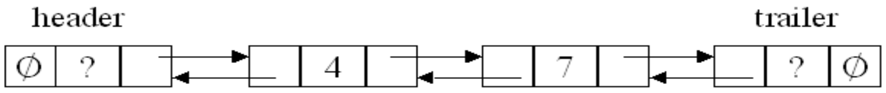
NOTES

Step 4 - Keep displaying **temp → data** with an arrow (**--->**) until **temp** reaches to the last node
Step 5 - Finally display **temp → data** with arrow pointing to **NULL** (**temp → data ---> NULL**).

6.6 Doubly-Linked Lists

In a singly-linked list, each node contains a pointer to the next node and it has no information about its previous node. Thus, one can traverse only in one direction, i.e., from beginning to end. However, sometimes it is required to traverse in the backward direction, i.e., from end to beginning. This can be implemented by maintaining an additional pointer in each node of the list that points to the previous node. Such type of a linked list is called doubly-linked list.
Each node of a doubly-linked list consists of three fields—prev, info, and next. The info field contains the data, the prev field contains the address of the previous node, and the next field contains the address of the next node.

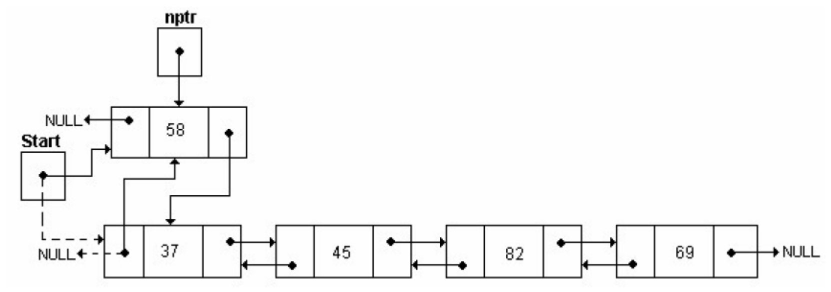




Since a doubly-linked list allows traversing in both forward and backward directions, it is also referred to as a two-way list.
The structure of a node in doubly linked list is as follows:

```
typedef struct node
{
        int info; /*to store integer type data*/
        struct node *next; /*to store a pointer to next node*/
        struct node *prev; /*to store a pointer to previous node*/
}Node;
Node *nptr; /*nptr is a pointer to node*/
```

Insertion
Insertion in the beginning
To insert a new node in the beginning of a doubly-linked list, a pointer, for example nptr to new node is created. The next field of the new node is made to point to the existing first node and prev field of the existing first node (that has become the second node now) is made to point to the new node. After that, Start is modified to point to the new node.



```
insert_beg(Start)
        1. Call create_node() //creating a new node pointed to by nptr
        2. If Start != NULL
                Set nptr->next = Start //inserting node in the beginning
                Set Start->prev = nptr
        End If
        3. Set Start = nptr //making Start to point to new node
        4. End
```

In a double linked list, we perform the following operations...

- Insertion
- Deletion
- Display

Insertion

In a double linked list, the insertion operation can be performed in three ways as follows...

- Inserting At Beginning of the list
- Inserting At End of the list
- Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the double linked list...

Step 1 - Create a **newNode** with given value and **newNode → previous** as **NULL**.
Step 2 - Check whether list is **Empty** (**head** == **NULL**)
Step 3 - If it is **Empty** then, assign **NULL** to **newNode → next** and **newNode** to **head**.
Step 4 - If it is **not Empty** then, assign **head** to **newNode → next** and **newNode** to **head**.

Inserting At End of the list

We can use the following steps to insert a new node at end of the double linked list...

Step 1 - Create a **newNode** with given value and **newNode → next** as **NULL**.
Step 2 - Check whether list is **Empty** (**head** == **NULL**)
Step3 - If it is **Empty**, then assign **NULL** to **newNode →**
previous and **newNode** to **head**.
Step 4 - If it is **not Empty**, then, define a node pointer **temp** and initialize with **head**.
Step 5 - Keep moving the **temp** to its next node until it reaches to the last node in the
list (until **temp → next** is equal to **NULL**).
Step 6 - Assign **newNode** to **temp → next** and **temp** to **newNode → previous**.

Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the double linked list...

Step 1 - Create a **newNode** with given value.
Step 2 - Check whether list is **Empty** (**head** == **NULL**)
Step 3 - If it is **Empty** then, assign **NULL** to both **newNode → previous** &
newNode → next and set **newNode** to **head**.
Step 4 - If it is **not Empty** then, define two node pointers **temp1** & **temp2** and
initialize **temp1** with **head**.
Step 5 - Keep moving the **temp1** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
Step 6 - Every time check whether **temp1** is reached to the last node. If it is reached to the last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp1** to next node.
Step 7- Assign **temp1 → next** to **temp2**, **newNode** to
temp1 → next, **temp1** to **newNode → previous**,
temp2 to **newNode → next** and **newNode** to **temp2 → previous**.

Deletion

In a double linked list, the deletion operation can be performed in three ways as follows...

- Deleting from Beginning of the list
- Deleting from End of the list
- Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the double linked list...

Step 1 - Check whether list is **Empty** (**head** == **NULL**)
Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and
terminate the function.
Step 3 - If it is not Empty then, define a Node pointer **'temp'** and initialize with **head**.
Step 4 - Check whether list is having only one node
(**temp → previous** is equal to **temp → next**)
Step 5 - If it is **TRUE**, then set **head** to **NULL** and delete **temp**
(Setting **Empty** list conditions)
Step 6 - If it is **FALSE**, then assign **temp →**
next to **head**, **NULL** to **head → previous** and delete **temp**.

Deleting from End of the list

We can use the following steps to delete a node from end of the double linked list...

Step 1 - Check whether list is **Empty** (**head** == **NULL**)
Step 2 - If it is **Empty**, then display **'List is Empty!!! Deletion is not possible'** and terminate the function.
Step 3 - If it is not Empty then, define a Node pointer **'temp'** and initialize with **head**.
Step 4 - Check whether list has only one Node
(**temp → previous** and **temp → next** both are **NULL**)
Step 5 - If it is **TRUE**, then assign **NULL** to **head** and delete **temp**. And terminate
from the function. (Setting **Empty** list condition)
Step 6 - If it is **FALSE**, then keep moving **temp** until it reaches to the last node in the list. (until **temp → next** is equal to **NULL**)
Step 7 - Assign **NULL** to **temp → previous → next** and delete **temp**.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the double linked list...

Step 1 - Check whether list is **Empty** (**head** == **NULL**)
Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and
terminate the function.
Step 3 - If it is not Empty, then define a Node pointer **'temp'** and initialize with **head**.
Step 4 - Keep moving the **temp** until it reaches to the exact node to be deleted or to the
last node.
Step 5 - If it is reached to the last node, then display **'Given node not found in the list!**
Deletion not possible!!!' and terminate the fuction.
Step 6 - If it is reached to the exact node which we want to delete, then check whether

list is having only one node or not
Step 7 - If list has only one node and that is the node which is to be deleted then
set **head** to **NULL** and delete **temp** (**free(temp)**).
Step 8 - If list contains multiple nodes, then check whether **temp** is the first node in the
list (**temp == head**).
Step 9 - If **temp** is the first node, then move the **head** to the next node (**head** = **head →**
next), set **head** of **previous** to **NULL** (**head → previous = NULL**) and
delete **temp**.
Step 10 - If **temp** is not the first node, then check whether it is the last node in the list
(**temp → next == NULL**).
Step 11 - If **temp** is the last node then set **temp** of **previous** of **next** to **NULL** (**temp →**
previous → next = NULL) and delete **temp** (**free(temp)**).
Step 12 - If **temp** is not the first node and not the last node, then set **temp** of **previous** of **next** to **temp** of **next** (**temp → previous → next = temp → next**), **temp** of **next** of **previous** to **temp** of **previous** (**temp → next → previous = temp → previous**) and delete **temp** (**free(temp)**).

Displaying a Double Linked List

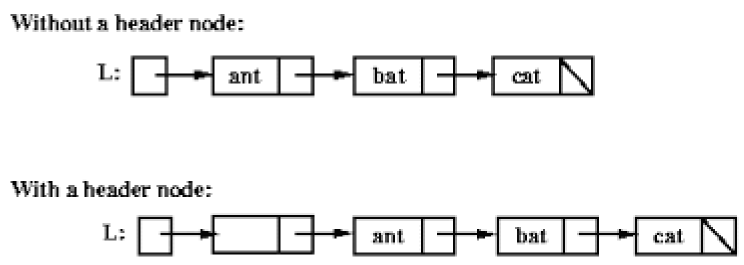
We can use the following steps to display the elements of a double linked list...

Step 1 - Check whether list is **Empty** (**head** == **NULL**)
Step 2 - If it is **Empty**, then display **'List is Empty!!!'** and terminate the function.
Step 3 - If it is not Empty, then define a Node pointer **'temp'** and initialize with **head**.
Step 4 - Display **'NULL <--- '**.
Step 5 - Keep displaying **temp → data** with an arrow (<===>) until **temp** reaches to the last node
Step 6 - Finally, display **temp → data** with arrow pointing to **NULL** (**temp → data ---> NULL**).

6.7 Header Linked List

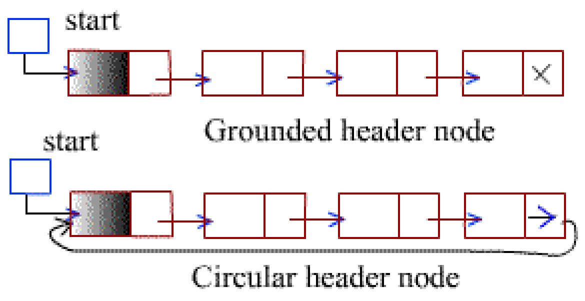
A header linked list is a linked list that contains a special note at the front of the list. This special node is called a headed node and it does not contain any actual data item that is included in the list but generally contains some useful information about the entire linked list.
The following are two kinds of widely used header lists:

NOTES



Grounded Header List

A grounded header list is a header list where the last node contains the null pointer. (The term “grounded” comes from the fact that many texts use the electrical ground symbol to indicate the null pointer.)



6.8 Check Your Progress Questions

1. What is the main advantage of a circular linked list over linear linked list?
2. How can a circular linked list be traversed?

6.9 Answers to Check Your Progress Questions

1. The main advantage of a circular linked list over a linear linked list is that by starting with any node in the list, its predecessor nodes can be reached.
2. A circular linked list can be traversed in the same way as a linear linked list, except the condition for checking the end of list.

6.10 Summary

- A dynamic data structure is one in which the memory for elements is allocated dynamically during run-time.
- Whenever a new element is inserted, the memory for the same is allocated dynamically and is linked to the data structure.
- A linked list is a linear collection of homogeneous elements called nodes.

NOTES

- In linked lists, insertion or deletion of nodes do not require shifting of existing nodes as in the case of arrays; they can be inserted or deleted merely by adjusting the pointers or links.
- Depending on the number of pointers in a node or the purpose for which the pointers are maintained, a linked list can be classified into various types such as singly-linked list, circular-linked list and doubly-linked list.
- As memory is allocated dynamically to the linked list, a new node can be inserted anytime in the list.
- While creating a linked list or inserting an element into a linked list, if a request for a new node arrives, the memory manager searches through the free- storage list for the block of desired size
- A number of operations can be performed on singly-linked lists. These operations include traversing, searching, inserting and deleting nodes, reversing, sorting, and merging linked lists. Creating a node means defining its structure, allocating memory to it, and its initialization.
- A linear linked list, in which the next field of the last node points back to the first node instead of containing NULL, is termed as a circular linked list.
- The main advantage of a circular linked list over a linear linked list is that by starting with any node in the list, its predecessor nodes can be reached.
- A circular linked list can be traversed in the same way as a linear linked list, except the condition for checking the end of list.
- In a singly-linked list, each node contains a pointer to the next node and it has no information about its previous node. Thus, one can traverse only in one direction, i.e., from beginning to end.

6.8 Key Words

- **Linked list:** It is a linear collection of homogeneous elements called nodes.
- **Dynamic data structure:** It is one in which the memory for elements is allocated dynamically during run-time

6.12 Self-Assessment Questions and Exercises

Short-Answer Questions

1. What is a linked list?
2. Explain singly-linked list.
3. What do you mean by doubly-linked list?
4. Differentiate between merging list and header linked list.

Long Answer Questions

1. “A dynamic data structure is one in which the memory for elements is allocated dynamically during run-time.” Explain.
2. “The last node of the singly-linked list contains NULL in its ‘next’ field which indicates the end of the list.” Explain with examples.
3. “A number of operations can be performed on singly-linked lists.” Elabotrate.
4. “All the operations that are performed on singly- linked lists can also be performed on doubly-linked lists.” Explain.

6.13 Further Readings

Storer, J.A. 2012. An Introduction to Data Structures and Algorithms. New York: Springer Publishing.
Preiss, Bruno. 2008. Data Structures and Algorithms with Object-Oriented Design Patterns in C++. London: John Wiley and Sons.
Pandey, Hari Mohan. 2009. Data Structures and Algorithms. New Delhi: Laxmi Publications.
Goodrich Michael, Tamassia Roberto and Michael H. Goldwasser. 2014. Data Structures and Algorithms in Java. London: John Wiley and Sons.
McMillan, Michael. 2007. Data Structures and Algorithms Using C#. Cambridge, UK: Cambridge University Press.

UNIT- VII OPERATIONS ON LINKED LIST

Structure

7.1 Introduction

Now that you have got an understanding of the basic concepts behind linked list and their types, its time to dive into the common operations that can be performed. This unit will basically discuss the insertion and deletion operations on the linked lists.

7.2 Objectives

After going through this unit, you will be able to:

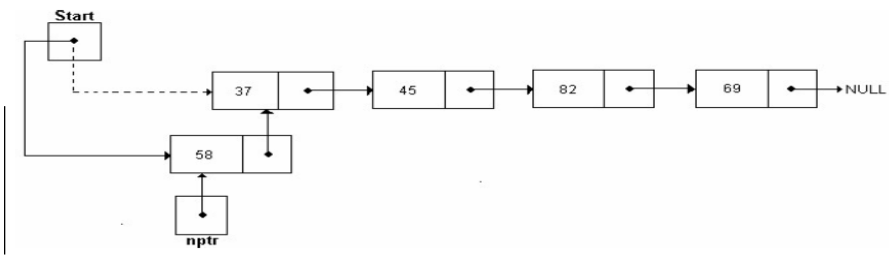
- Discuss lists
- Analyse insertion and deletion of operators in linked lists

7.3 Insertion and Deletion Operations in Linked List

To insert a node in the beginning of a list, the next field of the new node (pointed to by nptr) is made to point to the existing first node and the Start pointer is modified to point to the new node.

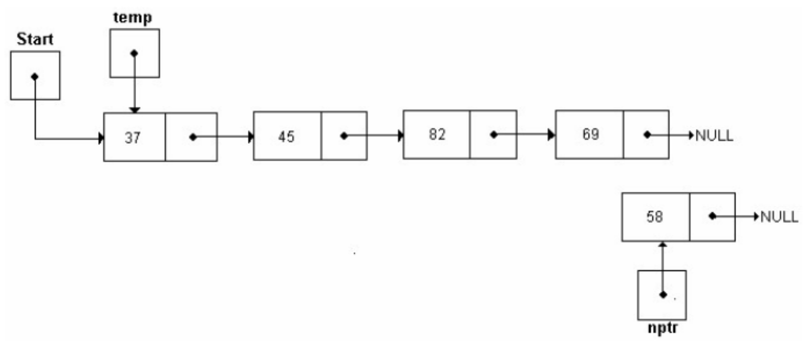
Algorithm for insertion in the beginning of linked list

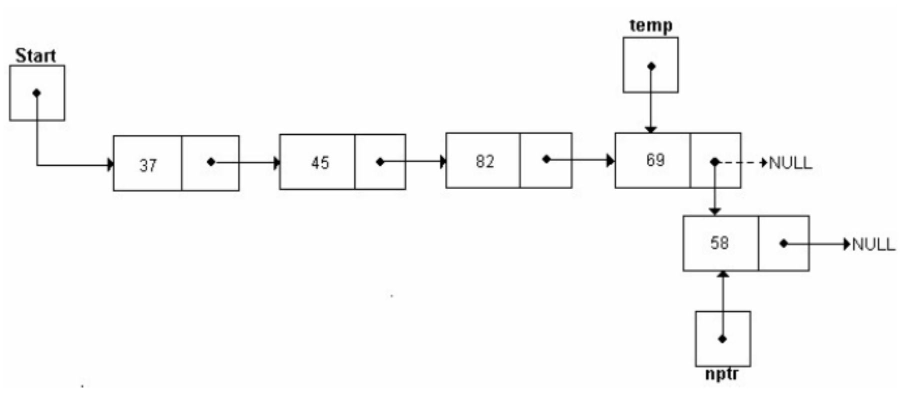
```
insert_beg(Start)
        1. Call create_node() //creating a new node pointed to by nptr
        2. Set nptr->next = Start
        3. Set Start = nptr //Start pointing to new node
        4. End
```



Insertion at end

To insert a node at the end of a linked list, the list is traversed up to the last node and the next field of this node is modified to point to the new node. However, if the linked list is initially empty then the new node becomes the first node and Start points to it.



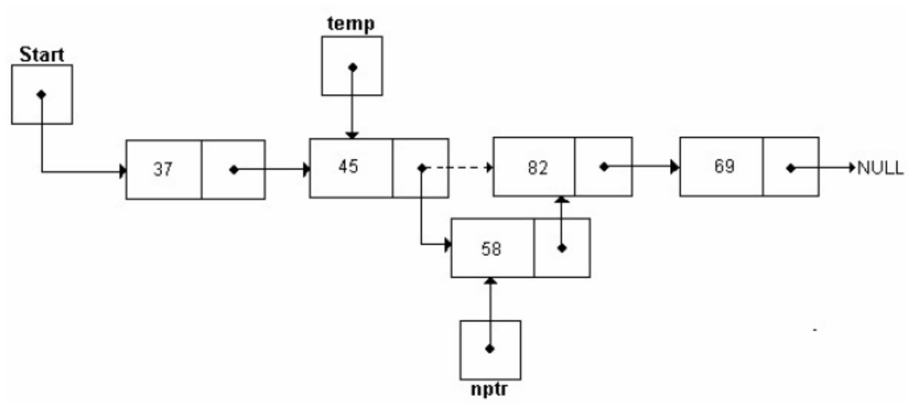


Algorithm for insertion in the end of linked list

```
insert_end(Start)
        1. Call create_node() //creating a new node pointed to by nptr
        2. If Start = NULL //checking for empty list
                Set Start = nptr //inserting new node as the first node
        Else
                Set temp = Start
                While temp->next != NULL //traversing up to the last node
                        Set temp = temp->next
                End While
                Set temp->next = nptr //appending new node at the end
        End If
    3.  End
```

Insertion at a specified position

To insert a node at a position pos as specified by the user, the list is traversed up to pos-1 position. Then the next field of the new node is made to point to the node that is already at the pos position and the next field of the node at pos-1 position is made to point to the new node.



Algorithm for insertion in the in specified position of linked list

```
insert_pos(Start)
    1. Call create_node() //creating a new node pointed to by nptr
    2. Set temp = Start
    3. Read pos //position at which the new node is to be inserted
    4. Call count_node(temp) //counting total number of nodes in count variable
    5. If (pos > count + 1 OR pos = 0)
        Print "Invalid position!" and go to step 7
    End If
    6. If pos = 1
        Set nptr->next = Start
        Set Start = nptr //inserting new node as the first node
    Else
        Set i = 1
        While i < pos - 1 //traversing up to the node at pos-1 position
            Set temp = temp->next
            Set i = i + 1
        End While
        Set nptr->next = temp->next //inserting new node at pos position
        Set temp->next = nptr
    End If
7. End
```

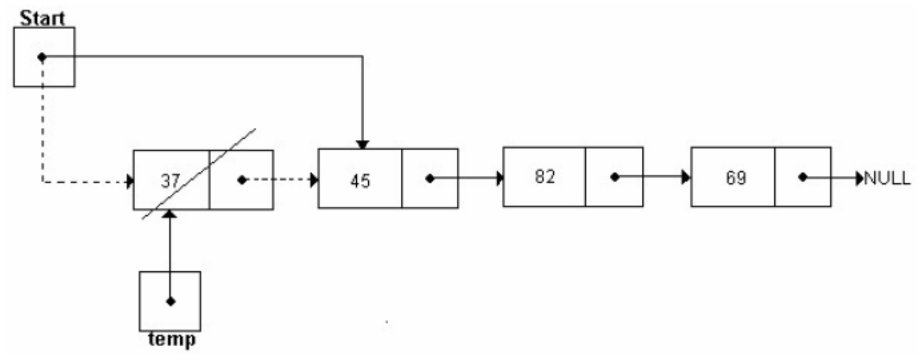
Deletion

Like insertion, nodes can be deleted from the linked list at any point of time and from any position. Whenever a node is deleted, the memory occupied by the node is de-allocated. It must be noted that while performing deletions, the immediate predecessor of the node to be deleted must be keep track of. Thus, two temporary pointer variables are used (except in case of deletion from beginning), while traversing the list.

Note: A situation where the user tries to delete a node from an empty linked list is termed as underflow.

Deletion from beginning

To delete a node from the beginning of a linked list, the address of the first node is stored in a temporary pointer variable temp and Start is modified to point to the second node in the linked list. After this, the memory occupied by the node pointed to by temp is de-allocated.

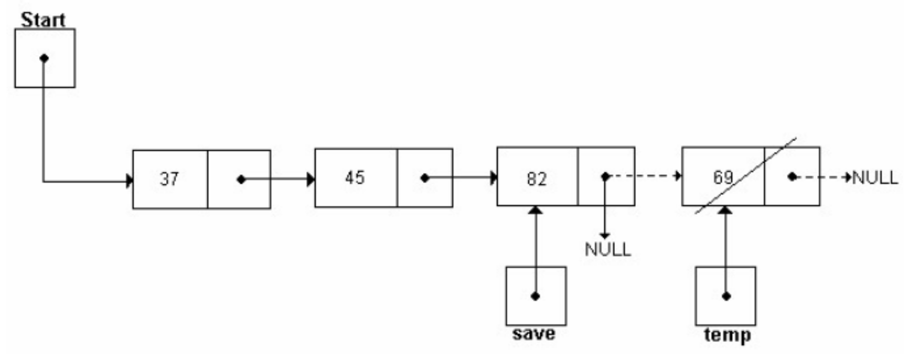


Algorithm for Deletion from beginning of linked list

```
delete_beg(Start)
        1. If Start = NULL //checking for underflow
                Print "Underflow: List is empty!" and go to step 5
        End If
        2. Set temp = Start //temp pointing to the first node
        3. Set Start = Start->next //moving Start to point to the second node
        4. Deallocate temp //deallocating memory
5. End
```

Deletion from end

To delete a node from the end of a linked list, the list is traversed up to the last node. Two pointer variables save and temp are used to traverse the list where save points to the node as previously pointed to by temp. At the end of traversing, temp points to the last node and save points to the second last node. Then the next field of the node pointed to by save is made to point to NULL and the memory occupied by the node pointed to by temp is de-allocated.

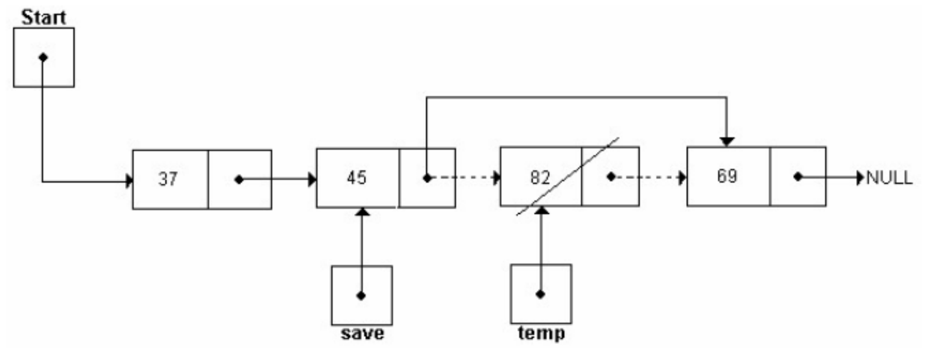


Algorithm for Deletion from the end of linked list

```
delete_end(Start)
        1. If Start = NULL //checking for underflow
                Print "Underflow: List is empty!" and go to step 6
        End If
        2. Set temp = Start //temp pointing to the first node
        3. If temp->next = NULL //deleting the only node of the list
                Set Start = NULL
        Else
                While (temp->next) != NULL //traversing up to the last
        node
                        Set save = temp //save pointing to node previously
                        Set temp = temp->next //moving temp to point to
                next node
                End While
        End If
        4. Set save->next = NULL //making new last node to point to
NULL
        5. Deallocate temp //deallocating memory
6. End
```

Deletion from a specified position

To delete a node from a position pos as specified by the user, the list is traversed up to pos position using pointer variables temp and save. At the end of traversing, temp points to the node at pos position and save points to the node at pos-1 position. Then the next field of the node pointed to by save is made to point to the node at pos+1 position and the memory occupied by the node as pointed to by temp is de-allocated.



Algorithm for deletion from a specified position of linked list

```
delete_pos(Start)
        1. If Start = NULL //checking for underflow
                Print "Underflow: List is empty!" and go to step 8
        End If
        2. Set temp = Start
        3. Read pos //position of the node to be deleted
        4. Call count_node(Start) //counting total number of nodes in count
variable
        5. If pos > count OR pos = 0
                Print "Invalid position!" and go to step 8
        End If
        6. If pos = 1
                Set Start = temp->next //deleting the first node
        Else
                Set i = 1
                While i < pos //traversing up to the node at position pos
                        Set save = temp
                        Set temp = temp->next
                        Set i = i + 1
                End While
```

```
            Set save->next = temp->next //deleting the node at position
        pos
        End If
        7. Deallocate temp //deallocating memory
8. End
```

7.4 Check Your Progress Questions

1. What is Linked list?
2. Advantages of Linked list.

7.5 Answers to Check Your Progress Questions

1. A **linked list** is a linear data structure, in which the elements are not stored at contiguous memory locations. Li**nked list** consists of nodes where each node contains a data field and a reference (**link**) to the next node in the **list**.
2. **Linked List** can grow and shrink during run time. Insertion and Deletion Operations are Easier. Efficient Memory Utilization, i.e no need to pre-allocate memory. Faster Access time, can be expanded in constant time without memory overhead.

7.6 Summary

- To insert a node at the end of a linked list, the list is traversed up to the last node and the next field of this node is modified to point to the new node.
- To insert a node at a position pos as specified by the user, the list is traversed up to pos-1 position
- Like insertion, nodes can be deleted from the linked list at any point of time and from any position.
- Whenever a node is deleted, the memory occupied by the node is deallocated.
- It must be noted that while performing deletions, the immediate predecessor of the node to be deleted must be keep track of.
- To delete a node from the beginning of a linked list, the address of the first node is stored in a temporary pointer variable temp and Start is modified to point to the second node in the linked list.
- To delete a node from the end of a linked list, the list is traversed up to the last node.
- To delete a node from a position pos as specified by the user, the list is traversed up to pos position using pointer variables temp and save.

.

NOTES

7.7 Key Words

- **Underflow:** It is a situation where the user tries to delete a node from an empty linked list.
- Header Linked list: A **Header linked list** is one more variant of **linked list**. In **Header linked list**, we have a special node present at the beginning of the **linked list**. This special node is used to store number of nodes present in the **linked list**.

7.8 Self-Assessment Questions and Exercises

Short-Answer Questions

1. What are lists?
2. Write an algorithm to insert a node at the end in linked lists.
3. Write an algorithm to delete a node in the linked list?

Long-Answer Questions

1. Write a detailed note on insertion and deletion of operations in linked list.
2. Write a detailed report on insertion and deletion in doubly-linked lists.

7.9 Further Readings

Storer, J.A. 2012. An Introduction to Data Structures and Algorithms. New York: Springer Publishing.
Preiss, Bruno. 2008. Data Structures and Algorithms with Object-Oriented Design Patterns in C++. London: John Wiley and Sons.
Pandey, Hari Mohan. 2009. Data Structures and Algorithms. New Delhi: Laxmi Publications.
Goodrich Michael, Tamassia Roberto and Michael H. Goldwasser. 2014. Data Structures and Algorithms in Java. London: John Wiley and Sons.
McMillan, Michael. 2007. Data Structures and Algorithms Using C#. Cambridge, UK: Cambridge University Press.

UNIT – VIII TRAVERSAL

Structure

8.1 Introduction

In this unit, you will learn about the traversing and representation of linked lists. Traversing means to access the elements of the lists for searching an element, find the position for insertion etc.

8.2 Objectives

- After going through this unit, you will be able to:
- Discuss traversing linked lists
- Explain the representation of linked lists
- Analyse memory allocation

8.3 Traversing Linked Lists

Traversing

Traversing a list means accessing the elements of a linked list, one by one, to process all or some of the elements. For example, to display values of the nodes, the number of nodes counted, or a particular item in the list is searched, then traversing is required. A list can be traversed by using a temporary pointer variable temp, which will point to the node that is currently being processed. Initially, temp points to the first node, processes that element, moves temp point to the next node using the statement temp=temp->next, processes that element, and moves on as long as the last node is not reached, that is, until temp becomes
NULL.

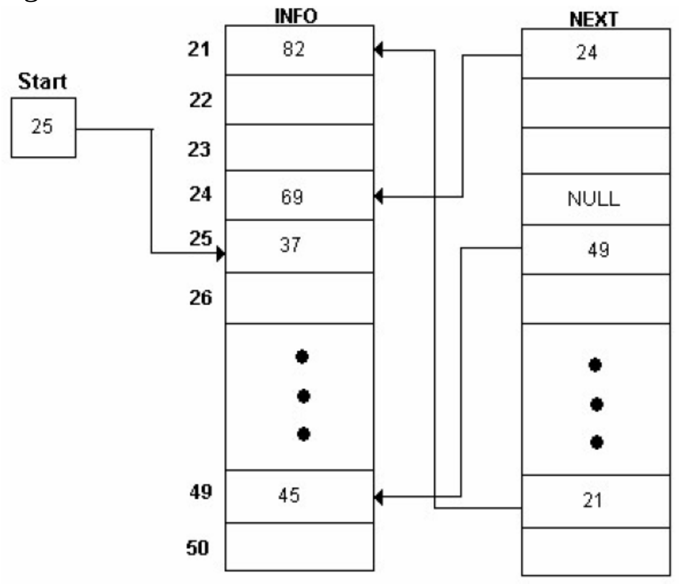
Algorithm for Traversing a linked List

```
traverse(Start)
        1. If Start = NULL //Start points to the first node of list
                Print "List is empty!!" and go to step 4
        End If
        2. Set temp = Start //initialising temp with Start
        3. While temp != NULL
```

```
            Print temp->info //displaying value of each node
            Set temp = temp->next //moving temp to point to next node
        End While
    3. End
```

8.4 Representation of Linked List

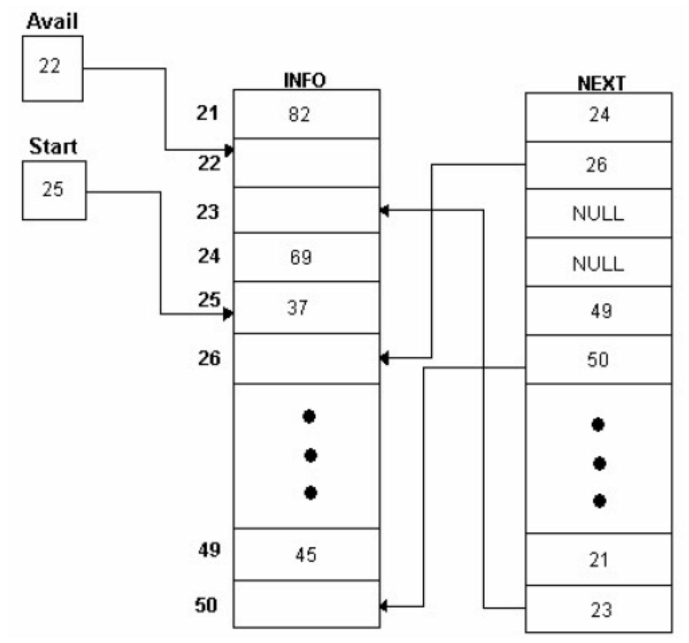
To maintain a linked list in the memory, two parallel arrays of equal size are used. One array (INFO) is used for the 'data' field and another array (NEXT) is used for the 'next' field of the nodes of a list. Values in the arrays are stored such that the 'ith' location in arrays 'INFO' and 'NEXT' contain the 'data' and 'next' fields of a node of the list respectively. In addition, a pointer variable 'Start' is maintained in memory that stores the starting address of the list.



The pointer variable Start contains 25, which is the address of first node of the list. This node stores the value 37 in array INFO and its corresponding element in array NEXT stores 49 which is the address of next node in the list and so on. Finally, the node at address 24 stores value 69 in array INFO and NULL in array NEXT, thus, it is the last node of the list. It must be noted that values in array INFO are stored randomly and array NEXT is used to keep a track of the values in the list.

Memory allocation

As memory is allocated dynamically to the linked list, a new node can be inserted anytime in the list. For this, the memory manager maintains a special linked list known as a free-storage list or memory bank or free pool which consists of unused memory cells. This list keeps a track of the free space available in the memory and a pointer to this list is stored in a pointer variable. Note that the end of the free-storage list is also denoted by storing NULL in the last available block of memory.



Avail contains 22, hence, INFO[22] is the starting point of the free-storage list. Since NEXT[22] contains 26, INFO[26] is the next free memory location. Similarly, other free spaces can be accessed and the NULL in NEXT[23] indicates the end of free-storage list.

While creating a linked list or inserting an element into a linked list, if a request for a new node arrives, the memory manager searches through the freestorage list for the block of desired size. If the block of desired size is found, it returns a pointer to that block. However, sometimes there is no space available, i.e., the free-storage list is empty. This situation is termed as overflow and the memory manager replies accordingly

8.5 Check Your Progress Questions

1) What happens whenever a node is deleted?
2) What is done to insert a node at the end of a linked list?
3) What is done to delete a node from the end of a circular linked list?

8.6 Answers to Check Your Progress Questions

1) Whenever a node is deleted, the memory occupied by the node is deallocated.
2) To insert a node at the end of a linked list, the list is traversed up to the last node and the next field of this node is modified to point to the new node.
3) To delete a node from the end of a circular linked list, two pointer variables save and temp are used.

8.7 Summary

- A number of operations can be performed on singly-linked lists. These operations include traversing, searching, inserting and deleting nodes, reversing, sorting, and merging linked lists.
- Creating a node means defining its structure, allocating memory to it, and its initialization.
- To define a node containing an integer data and a pointer to next node in C language, a self-referential structure can be used whose definition is as follows:
- Traversing a list means accessing the elements of a linked list, one by one, to process all or some of the elements.
- To maintain a linked list in the memory, two parallel arrays of equal size are used.
- As memory is allocated dynamically to the linked list, a new node can be inserted anytime in the list.
- While creating a linked list or inserting an element into a linked list, if a request for a new node arrives, the memory manager searches through the free- storage list for the block of desired size.
- If the block of desired size is found, it returns a pointer to that block.

8.8 Key Words

- **Traversing a list**: It means accessing the elements of a linked list, one by one, to process all or some of the elements.
- **Linked List**: It is an ordered set of data elements, each containing a link to its successor (and sometimes its predecessor).

8.9 Self-Assessment Questions and Exercises

Short-Answer Questions

1. Write a short note on linked lists.
2. How are linked lists represented?
3. List the operations performed on linked lists.

Long-Answer Questions

1. “Traversing a list means accessing the elements of a linked list, one by one, to process all or some of the elements.” Explain.
2. “As memory is allocated dynamically to the linked list, a new node can be inserted anytime in the list.” Discuss.
3. “While creating a linked list or inserting an element into a linked list, if request for a new node arrives, the memory manager searches through the free- storage list for the block of desired size.” Discuss.

8.10 Further Readings

NOTES

Storer, J.A. 2012. An Introduction to Data Structures and Algorithms. New York: Springer Publishing. Preiss, Bruno. 2008. Data Structures and Algorithms with Object-Oriented Design Patterns in C++. London: John Wiley and Sons. Pandey, Hari Mohan. 2009. Data Structures and Algorithms. New Delhi: Laxmi Publications.

Goodrich Michael, Tamassia Roberto and Michael H. Goldwasser. 2014. Data Structures and Algorithms in Java. London: John Wiley and Sons.

McMillan, Michael. 2007. Data Structures and Algorithms Using C#. Cambridge, UK: Cambridge University Press.

BLOCK –III NON-LINEAR DATA STRUCTURE

NOTES

UNIT- IX TREES

Structure

9.1 Introduction

A tree is a widely used abstract data type also called an ADT, or data structure. This ADT simulates a hierarchical tree structure that has a root value and subtrees of children with a parent node; these are represented as a set of linked nodes. A binary tree is made up of nodes, where each node contains a ‘left’ and ‘right’ reference, and a data element. The topmost node in the tree is called the root. Nodes that go with the same parent are called siblings. In this unit you will learn in detail about trees.

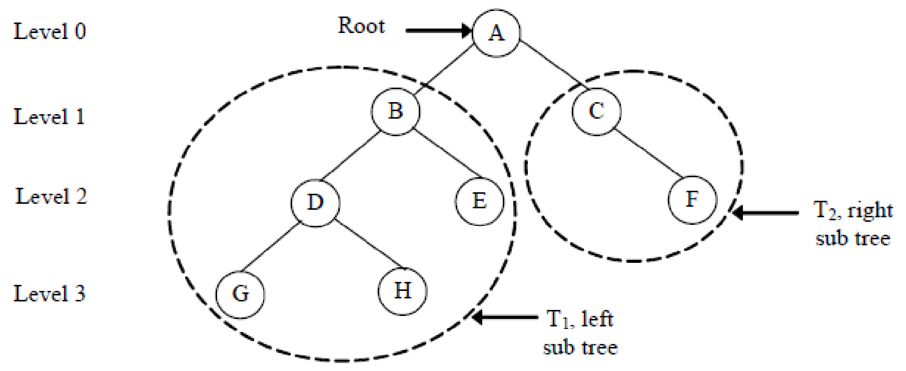
9.2 Objectives

After going through this unit, you will be able to:

- Discuss binary trees
- Analyse the nature of binary trees
- Explain the forms of binary trees
- Understand the concept of extended binary tree

9.3 Binary Trees

A binary tree is a special type of tree, which can either be empty or has finite set of nodes such that one of the nodes is designated as root node and remaining nodes are partitioned into two sub trees of root node known as left sub tree and right sub tree. The nonempty left sub tree and the right sub tree are also binary trees. Unlike general tree each node in binary tree is restricted to have only two child nodes.



The topmost node A is the root node of the tree T. Each node in this tree has zero or at the most two child nodes. The nodes A, B and D have two child nodes, node C has only one child node, and the nodes G, H, E and F are leaf nodes having no child nodes. The nodes B, C, D are internal nodes having child as well as parent nodes. Some basic terms associated with binary trees are:

- **Ancestor and descendant**: Node N1 is said to be an ancestor of node N2. N1 is the parent node of N2 or so on, whereas, node N2 is said to be the descendant of node N1. The node N2 is said to be left descendant of node N1 if it belongs to left sub tree of N1 and is said to be the right descendant of N1 if it belongs to right sub tree of N1. In binary tree shown here, node A is ancestor of node H and node H is left descendent of node A.
- **Degree of a node**: Degree of a node is equal to the number of its child nodes. In binary tree shown here, the nodes A, B and D have degree 2, node C has degree 1 and nodes G, H, E and F have degree 0.
- **Level**: Since binary tree is a multilevel data structure, each node belongs to a particular level number. In binary tree shown here, the root node A belongs to level 0, its child nodes belongs to level 1, child nodes of nodes B and C belong to level 2, and so on.
- **Depth (or height):** Depth of the binary tree is the highest level number of any node in a binary tree. In binary tree shown here, the nodes G and H are nodes with highest level number 3. Hence, the depth of the binary tree is 3.
- **Siblings**: The nodes belonging to the same parent node are known as sibling nodes. In binary tree shown here, nodes B and C are sibling nodes as they have same parent node, that is, A. Similarly, the nodes D and E are also sibling nodes.
- **Edge**: Edge is a line connecting any two nodes. In binary tree shown here, there exists an edge between nodes A and B, whereas, there is no edge between the nodes B and C.
- **Path**: Path between the two nodes x and y is a sequence of consecutive edges being followed from node x to y. In binary tree shown here, the path between the nodes A and H is A->B->D->H. Similarly, the path from A to F is A->C->F.

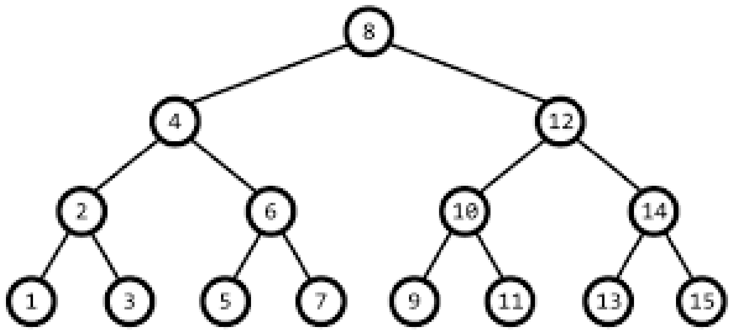
9.4 Types of Binary Tree

There are various forms of binary trees that are formed by imposing certain restrictions on them. Some of the variations of binary trees are—complete binary tree and extended binary tree.

Complete Binary Tree

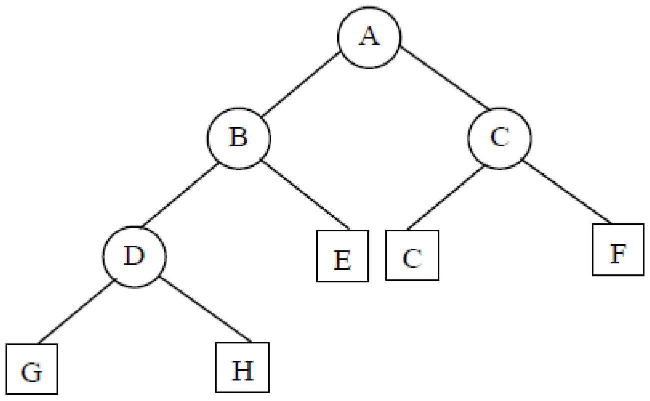
A binary tree is said to be complete binary tree if all the leaf nodes of the tree are at the same level. Thus, the tree has maximum number of nodes at all the levels.

At any level n of binary tree, there can be at the most 2^n nodes.



Extended Binary Tree

A binary tree is said to be extended binary tree (also known as 2-tree) if all of its nodes are of either zero or two degree. In this type of binary trees, the nodes with degree two (also known as internal nodes) are represented as circles and nodes with degree zero (also known as external nodes) are represented as squares.



Full binary tree: Every node other than leaf nodes has 2 child nodes.
Perfect binary tree: All nodes have two children and all leaves are at the same level.

Binary Tree Traversal

It is the process of visiting each node in the tree exactly once. Two main Traversals are:

Breadth First Traversal
Depth First Traversal

Breadth First Traversal:

Visiting each node starting from the lowest(Root) level and moving down level by level, visiting nodes on each level from left to right.

four possible ways are:

1. Top-down, Left-to-right
2. Top-down, right-to-left
3. Bottom-up, Left-to-right
4. Bottom-up, right-to-left

Top-down, Left-to-right BFT:

Queue structure is used.

After a node is visited, its children, if any, are placed at the end of the queue, and the node at the beginning of the queue is visited.

For a node on level n, its children are on level n+1, by placing these children at the end of the queue, they are visited after all nodes from level n are visited.

All nodes on level n must be visited before visiting nodes on level n+1.

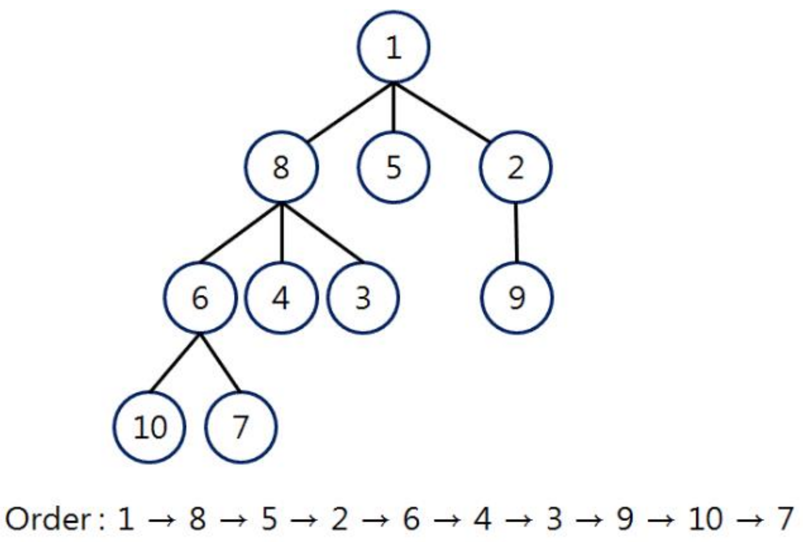
Algorithm BFT (root)

```
{
    p=root
    if( p!=NULL)
    {
        Enqueue(p)
        while( Queue is not empty)
        {
            p=Dequeue()
            visit(p)
            if( p->left != NULL)
            {
                Enqueue(p->left)
            }
            if( p->right != NULL)
            {
                Enqueue(p->right)
            }

        }

    }

}
```

Approach:

- Take a Empty Queue.
- Start from the root, insert the root into the Queue.
- Now while Queue is not empty,
  - Extract the node from the Queue and insert all its children into the Queue.
  - Print the extracted node.

NOTES



Depth First Traversal:

DFT proceeds as far as possible to the left, then backs up until the first cross-road, goes one step to the right and again as far as possible to the left, we repeat this process until all nodes are visited. Thus there are three tasks:

V – visiting a node
L - traversing the left subtree
R – traversing the right subtree

Types of DFT:

1. Preorder Tree Traversal [VLR]
2. Inorder Tree Traversal [LVR]
3. Postorder Tree Traversal [LRV]

We can solve using Recusive and Non-recursive algorithms: Recursive Algorithms:

Inorder Traversal:

Steps:

- Traverse the Left subtree
- Process the root node
- Traverse the right subtree

Algorithm Inorder(Node p)

```
{
        if (p!=NULL)
        {
                Inorder(p->left)
                visit(p)
                Inorder(p->right)
        }

}
```

Inorder Traversal leads to infix expression

Preorder Traversal:

Steps:

- Process the root node
- Traverse the Left subtree
- Traverse the right subtree

Algorithm Preorder(Node p)

```
{
        if (p!=NULL)
        {
                visit(p)
                Preorder(p->left)
                Preorder(p->right)
        }

}
```

Preorder Traversal leads to Prefix expression

Postorder Traversal:

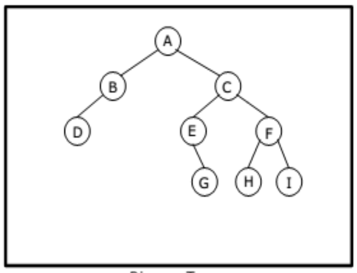
Steps:

- Traverse the Left subtree
- Traverse the right subtree
- Process the root node

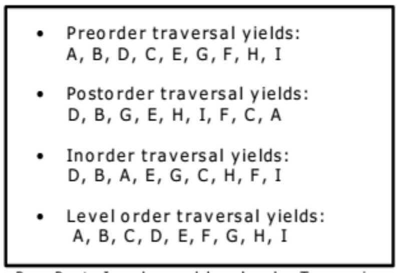
Algorithm Postorder(Node p)

```
{
        if (p!=NULL)
        {
                Postorder(p->left)
                Postorder(p->right)
                visit(p)
        }

}
```

Traverse the following binary tree in pre, post, inorder and level order.



Binary Tree



- Preorder traversal yields:
  A, B, D, C, E, G, F, H, I
- Postorder traversal yields:
  D, B, G, E, H, I, F, C, A
- Inorder traversal yields:
  D, B, A, E, G, C, H, F, I
- Level order traversal yields:
  A, B, C, D, E, F, G, H, I

Pre, Post, Inorder and level order Traversing

NOTES

9.5 Binary Tree Representations

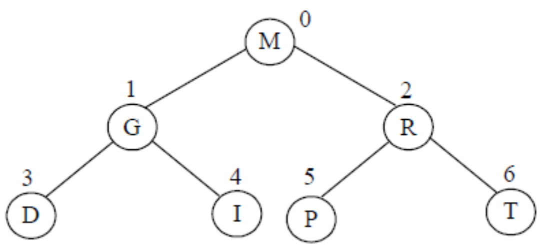
Like stacks and queues, binary trees can also be represented in two ways in memory

- Array (sequential) representation
- Linked representation

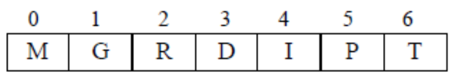
In array representation, memory is allocated at compile time and in linked representation, memory is allocated dynamically.

Array representation

In array representation binary tree is represented sequentially in memory by using single one-dimensional array. A binary tree of height n may comprise at most 2(n+1)-1 nodes, hence an array of maximum size 2(n+1)-1 is used for representing such a tree. All the nodes of the tree are assigned a sequence number (from 0 to (2(n+1)-1)-1) level by level. That is, the root node at level 0 is assigned a sequence number 0, and then nodes at level 1 are assigned sequence number in ascending order from left to right, and so on. For example, the nodes of a binary tree of height 2, having 7 (2(2+1)-1) nodes can be numbered.



(a) Ordering of Nodes of Binary Tree



0	1	2	3	4	5	6
M	G	R	D	I	P	T

(b) Nodes of Binary Tree Stored in an Array

The numbers assigned to the nodes indicates the position (index value) of an array at which that particular node is stored. The array representation of this tree It can be observed that if any node is stored at position p, then its left child node is stored at 2*p+1 position and its right child node is stored at 2*p+2 position.

It is to be noted that array representation of an array may lead to wastage of memory as if a node is absent in the tree its corresponding location in the array will be also be left empty.

Advantages:

We can easily locate the parent, left and right child of any node. It is convenient for performing Heapsort.

Disadvantages:

- This representation is efficient only for full trees. For incomplete trees, it leads to free locations.
- The data may overflow the array if too little space is allocated.
- Memory space may be wasted if too much space is allocated.

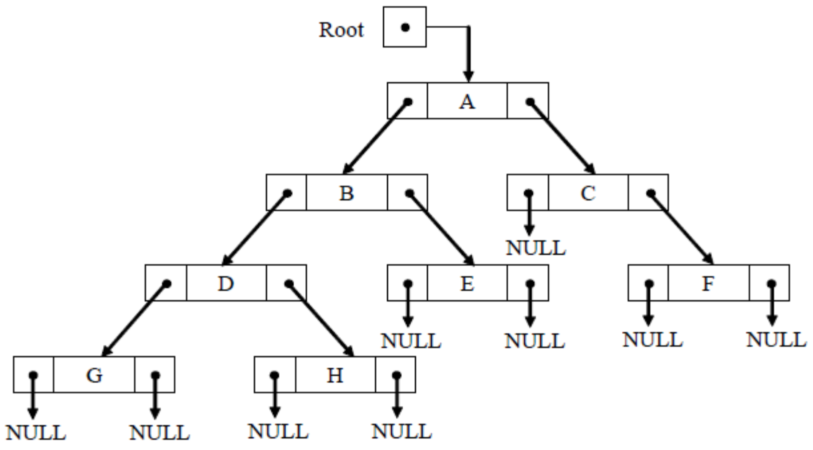
- Trees often change and it may be hard to predict how many nodes will be created during program execution. So, this may be inconvenient.

Linked Representation

Linked representation is one of the most common and important way of representing a binary tree in memory. The linked representation of a binary tree is implemented by using a linked list having info part and two pointers. The info part contains the data value and two pointers, left and right are used to point to the left and right sub tree of a node, respectively.

Structure of tree represented by using linked list

```
struct node
{
        int info;
        struct node *left;
        struct node *right;
}Node;
```



In linked representation, a pointer variable Root of Node type is used to point to the root node of a tree. Root variable is used for accessing the root and the subsequent nodes of a binary tree. Since binary tree is empty in the beginning, the pointer variable Root is initialized with NULL

9.6 Check Your Progress Questions

1. What is the depth of binary tree?
2. What are sibling nodes?

9.7 Answers to Check Your Progress Questions

1. Depth of the binary tree is the highest level number of any node in a binary tree.
2. The nodes belonging to the same parent node are known as sibling nodes.

NOTES

9.8 Summary

- A binary tree is a special type of tree, which can either be empty or has finite set of nodes.
- The nonempty left sub tree and the right sub tree are also binary trees.
- Degree of a node is equal to the number of its child nodes.
- Since binary tree is a multilevel data structure, each node belongs to a particular level number.
- Depth of the binary tree is the highest level number of any node in a binary tree.
- The nodes belonging to the same parent node are known as sibling nodes.
- There are various forms of binary trees that are formed by imposing certain restrictions on them.
- Some of the variations of binary trees are—complete binary tree and extended binary tree.
- A binary tree is said to be complete binary tree if all the leaf nodes of the tree are at the same level.
- A binary tree is said to be extended binary tree (also known as 2-tree) if all of its nodes are of either zero or two degree.
- Like stacks and queues, binary trees can also be represented in two ways in memory—array (sequential) representation and linked representation.
- In array representation binary tree is represented sequentially in memory by using single one-dimensional array.
- The numbers assigned to the nodes indicates the position (index value) of an array at which that particular node is stored.
- Linked representation is one of the most common and important way of representing a binary tree in memory.

9.9 Key Words

- **Complete Binary tree**: It is a binary tree if all the leaf nodes of the tree are at the same level.
- **Extended Binary tree**: It is a binary tree if all of its nodes are of either zero or two degree.

9.10 Self-Assessment Questions and Exercises

Short-Answer Questions

1. Write a short note on binary trees.
2. Show a diagrammatic representation of a binary tree.
3. What do you mean by ancestor and descendent?

Long-Answer Questions

1. Write a detailed note on the forms of binary trees.
2. How is a complete binary tree different from an extended binary tree? Explain.

3. Write a detailed note on Binary tree representations.
4. “Linked representation is one of the most common and important way of representing a binary tree in memory.” Explain.

9.11 Further Readings

NOTES

Storer, J.A. 2012. An Introduction to Data Structures and Algorithms. New York: Springer Publishing.
Preiss, Bruno. 2008. Data Structures and Algorithms with Object-Oriented Design Patterns in C++. London: John Wiley and Sons.
Pandey, Hari Mohan. 2009. Data Structures and Algorithms. New Delhi: Laxmi Publications.
Goodrich Michael, Tamassia Roberto and Michael H. Goldwasser. 2014. Data Structures and Algorithms in Java. London: John Wiley and Sons.
McMillan, Michael. 2007. Data Structures and Algorithms Using C#. Cambridge, UK: Cambridge University Press.

UNIT- X BINARY TREE OPERATIONS

Structure

10.1 Introduction

This unit will explain about binary tree operations and applications. A binary tree can be traversed in three different ways— in-order, pre-order and post-order traversal.

10.2 Objectives

After going through this unit, you will be able to:

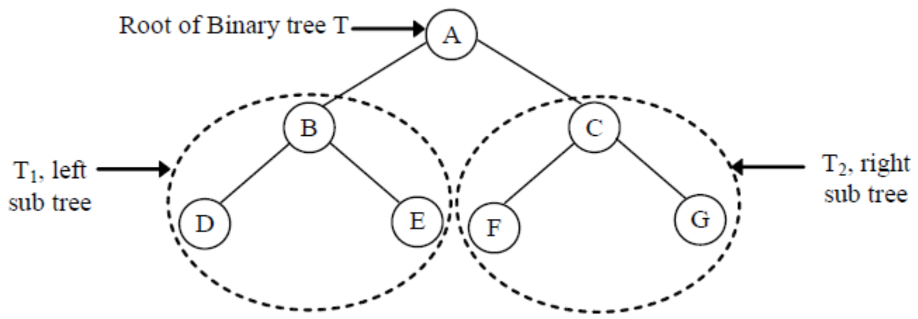
- Discuss about traversing a binary tree
- Analyse what happens when a root node is visited in pre-order traversal
- Understand the different ways in which a tree can be traversed
- Explain binary search tree

10.3 Traversing Binary trees

Traversing a binary tree refers to the process of visiting each and every node of the tree exactly once. The three different ways in which a tree can be traversed are

- in-order
- pre-order
- post-order traversal

The main difference in these traversal methods is based on the order in which the root node is visited. Note that in all the traversals the left sub tree is always traversed before the traversal of the right sub tree.



Pre-order

In pre-order traversal, the root node is visited before traversing its left and right sub trees. Steps for traversing a nonempty binary tree in pre-order are as follows:

- Visit the root node R.
- Traverse the left sub tree of root node R in pre-order.
- Traverse the right sub tree of root node R in pre-order.

For example, in binary tree T, the root node A is traversed before traversing its left sub tree and the right sub tree. In the left sub tree T1, the root node B (of left sub tree T1) is traversed before traversing the nodes D and E. After traversing the root node of binary tree T and traversing the left sub tree T1, the right sub tree T2 is also traversed following the same procedure. Hence, the resultant pre-order traversal of the binary tree T is A, B, D, E, C, F, G.

In-order

In in-order traversal, the root node is visited after the traversal of its left sub tree and before the traversal of its right sub tree. Steps for traversing a nonempty binary tree in in-order are as follows:

- Traverse the left sub tree of root node R in in-order.
- 2. Visit the root node R.
- Traverse the right sub tree of root node R in in-order.

For example, in binary tree T, the left sub tree T1 is traversed before traversing the root node A. In the left sub tree T1, the node D is traversed before traversing its root node B (of left sub tree T1). After traversing the node D and B, the node E is traversed. Once the traversal of left sub tree T1 and the root node A of binary tree T is complete, the right sub tree T2 is traversed following the same procedure. Hence, the resultant in-order traversal of the binary tree T is D, B, E, A, F, C, G.

Post-order

In post-order traversal, the root node is visited after traversing its left and right sub trees. Steps for traversing a nonempty binary tree in post-order are as follows:

- Traverse the left sub tree of root node R in post-order.
- 2. Traverse the right sub tree of root node R in post-order.
- Visit the root node R.

For example, in binary tree T, the root node A is traversed after traversing its left sub tree and the right sub tree. In the left sub tree T1, the root node B (of left sub tree T1) is traversed after traversing the nodes D and E. Similarly, the nodes of right sub tree T2 are traversed following the same procedure. After traversing the left sub tree (T1) and right sub tree (T2), the root node A of binary tree T is traversed. Hence, the resultant post-order traversal of the binary tree T is D, E, B, F, G, C, A.

10.4 Binary Search Tree

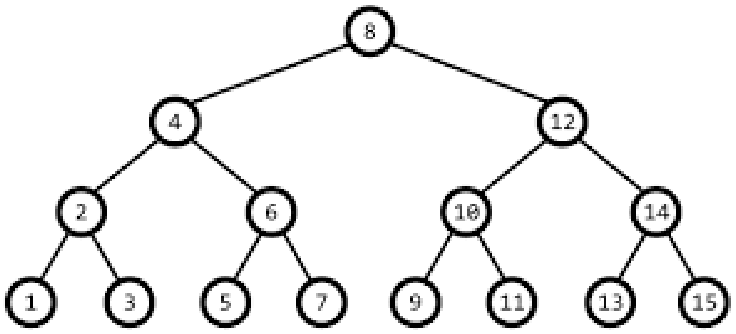
Binary search tree, also known as binary sorted tree, is a kind of a binary tree, which satisfies the following conditions

- The data value in each node is a key (unique) value, that is, no two nodes can have identical values.
- The data values in the nodes of the left sub tree, if exists, is smaller than the value in the root node.
- The data values in the nodes of the right sub tree, if exists, is greater than the value in the root node.
- The left and the right sub trees, if exists, are also binary search trees.

In other words, values in the left sub tree of a root node are smaller than the value of the root node, and the values in the right sub tree are greater than the value of the root node. This rule is applicable on all the subsequent sub trees in a binary search tree. In addition, each and every value in binary search tree is unique, that is, no two nodes in it can have identical values.

Searching a Node

Searching an element in a binary search tree is easy, since the elements in this tree are arranged in a sorted order. The element to be searched is compared with the value in the root node. If the element is smaller than the value in the root node, then the searching will proceed to the left sub tree and if the element is greater than the value in the root node, then the searching will proceed to the right sub tree. This process is repeated until either the element to be searched is found or NULL value is encountered.



Algorithm for searching a node in a binary search tree

```
search(item, ptr)
    1. If !(ptr)
        Print "Element not found!" and go to step 3
    End If
    2. If item < ptr->info
        Call search(item, ptr->left)
    Else If item > ptr->info
        Call search(item, ptr->right)
    Else
```

```
        Print "Element found."
    End If
3. End
```

10.5 Traversing a Binary Search Tree

Traversing a binary search tree is same as traversing a binary tree. That is, binary search tree can also be traversed in three different ways—pre-order, in-order and post-order. For example, consider the tree shown in Figure 10.2. The pre-order, in-order and post-order traversal of this tree is as follows:

Pre-order traversal: 66 40 30 20 35 50 45 55 90 75 70 80 110 100 120

In-order traversal: 20 30 35 40 45 50 55 66 70 75 80 90 100 110 120

Post-order traversal: 20 35 30 45 55 50 40 70 80 75 100 120 110 90 66

It can be observed that when a binary search tree is traversed in in-order, it results in the sequence of elements in ascending order. The algorithms for traversing tree in pre-order, in-order and post-order are recursive in nature.

Algorithm for pre order traversal

```
preorder(ptr)
    1. If ptr != NULL
        Print ptr->info //ptr is temporary pointer initialised with
    Root
        Call preorder(ptr->left)
        Call preorder(ptr->right)
    End If
2. End
```

Algorithm for post order traversal

```
inorder(ptr)
    1. If ptr != NULL
        Call inorder(ptr->left) //ptr is temporary pointer initialised
    with Root
        Print ptr->info
        Call inorder(ptr->right)
    End If
2. End
```

Algorithm for in order traversal

```
postorder(ptr)
    1. If ptr != NULL
        Call postorder(ptr->left) //ptr is temporary pointer initialised
    with Root
        Call postorder(ptr->right)
        Print ptr->info
    End If
2. End
```

10.6 Check Your Progress Questions

1) What does traversing a binary tree refer to?
2) When is the root node visited in pre-order traversal?
3) Why is searching an element in a binary search tree easy?

10.7 Answers to Check Your Progress Questions

1) Traversing a binary tree refers to the process of visiting each and every node of the tree exactly once.
2) In pre-order traversal, the root node is visited before traversing its left and right sub trees.
3) Searching an element in a binary search tree is easy, since the elements in this tree are arranged in a sorted order.

10.8 Summary

- Traversing a binary tree refers to the process of visiting each and every node of the tree exactly once.
- The three different ways in which a tree can be traversed are— in-order, pre-order and post-order traversal.
- The main difference in these traversal methods is based on the order in which the root node is visited.
- In pre-order traversal, the root node is visited before traversing its left and right sub trees.
- In in-order traversal, the root node is visited after the traversal of its left sub tree and before the traversal of its right sub tree.
- In post-order traversal, the root node is visited after traversing its left and right sub trees.
- There are various operations that can be performed on the binary search trees. Some of these are searching a node, insertion of a new node, traversal of a tree and deletion of a node.
- Searching an element in a binary search tree is easy, since the elements in this tree are arranged in a sorted order.

10.9 Key Words

- **Traversing:** It is the process of visiting each and every data item of the data structure exactly once.
- **Binary tree:** It is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child.

10.10 Self-Assessment Questions and Exercises

Short-Answer Questions

1) What is binary search tree?
2) What do you mean by traversing binary trees?
3) How do you search a node in binary tree?

Long-Answer Questions

1) "The element to be searched is compared with the value in the root node. If the element is smaller than the value in the root node, then the searching will proceed to the left sub tree and if the element is greater than the value in the root node, then the searching will proceed to the right sub tree." Explain in detail.

2) Write a program to illustrate the various operations performed on binary search tree. NOTE: In case of deletion of a node with two child nodes, largest value from left sub tree (in-order predecessor) is used for replacement.

3) Write the algorithms for Pre-order, In-order and Post order traversal.

10.11 Further Readings

Storer, J.A. 2012. An Introduction to Data Structures and Algorithms. New York: Springer Publishing.

Preiss, Bruno. 2008. Data Structures and Algorithms with Object-Oriented Design Patterns in C++. London: John Wiley and Sons.

Pandey, Hari Mohan. 2009. Data Structures and Algorithms. New Delhi: Laxmi Publications.

Goodrich Michael, Tamassia Roberto and Michael H. Goldwasser. 2014. Data Structures and Algorithms in Java. London: John Wiley and Sons.

McMillan, Michael. 2007. Data Structures and Algorithms Using C#. Cambridge, UK: Cambridge University Press.

UNIT - XI
OPERATIONS ON BINARY TREE

Structure

11.1 Introduction

We considered a particular kind of a binary tree called a Binary Search Tree (BST). A binary tree is a binary search tree (BST) if and only if an in order traversal of the binary tree results in a sorted sequence. The idea of a binary search tree is that data is stored according to an order, so that it can be retrieved very efficiently. This unit will explain the various types of operations on binary tree.

11.2 Objectives

After going through this unit, you will be able to:

- Discuss the operations in binary tree
- Understand the insertion and deletion operations
- List the hashing techniques
- Analyse the division remained method

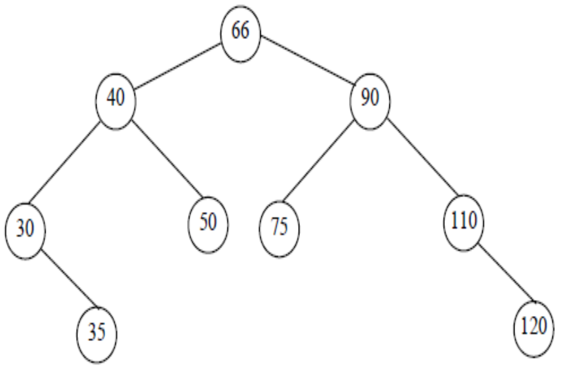
11.3 Insertion and Deletion Operations

Insertion in a binary search tree is similar to the procedure for searching an element in a binary search tree. The difference is that in case of insertion, appropriate null pointer is searched where new node can be inserted. The process of inserting a node in a binary search tree can be divided into two steps—in the first step, the tree is searched to determine the appropriate position where the node is to be inserted and in the second step, the node is inserted at this searched position.

Here are two cases of insertion in a tree

- o Insertion into an empty tree.
- o Insertion into a nonempty tree.

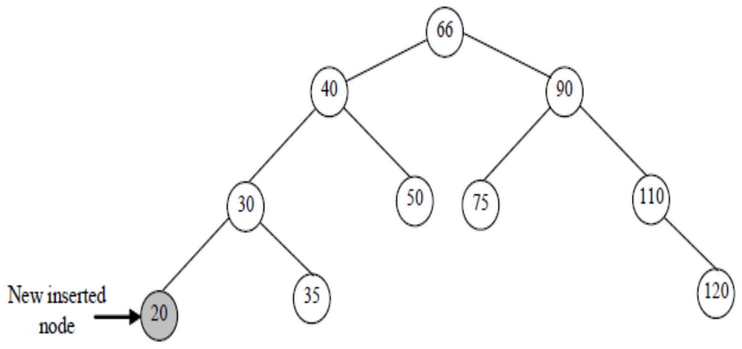
In case the tree is initially empty, the new node to be inserted becomes its root node. In case the tree is nonempty, appropriate position is determined for insertion. For this, first of all the value in the new node is compared with the root node of the tree. If the value in the new node is less than the value in the root node, the new node is added as the left leaf if the left sub tree is empty, otherwise the search continues in the left sub tree. On the other hand, if the value in the new node is greater than the value in the root node, the new node is added as the right leaf if the right sub tree is empty, otherwise the search continues in the right sub tree.



For example, consider the sample binary search tree shown

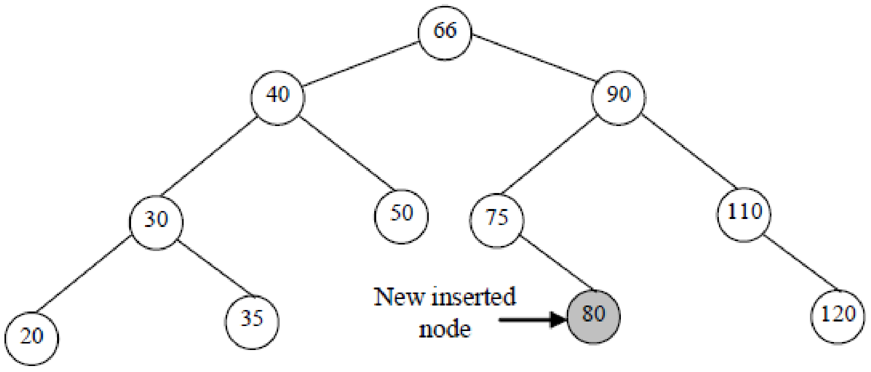
For inserting elements 20 and 80, follow the steps given:

- Compare 20 with the value in the root node, that is, 66. Since 20 is smaller than 66, move to the left sub tree.
- 2. Finding that the left pointer of root node is non-null, compare 20 with the value (40) in this node. Since 20 is smaller than 40, move to the left sub tree.
- Again, the left pointer of the current node is non-null, compare 20 with the value (30) in this node. Since 20 is smaller than 30, move to the left sub tree.
- Now, the left pointer is null, thus 20 will be inserted at this position.



Steps for inserting the element 80 are as follows:

- Compare 80 with the value in root node 66. Since 80 is greater than 66, move to the right sub tree.
- 2. Finding that the right pointer of root node is non-null, compare 80 with the value (90) in this node. Since 80 is smaller than 90, move to the left sub tree.
- Again, the left pointer of the current node is non-null, compare 80 with the value (75) in this node. Since 80 is greater than 75, move to the right sub tree.
- Now, the right pointer is null, thus 80 will be inserted at this position.



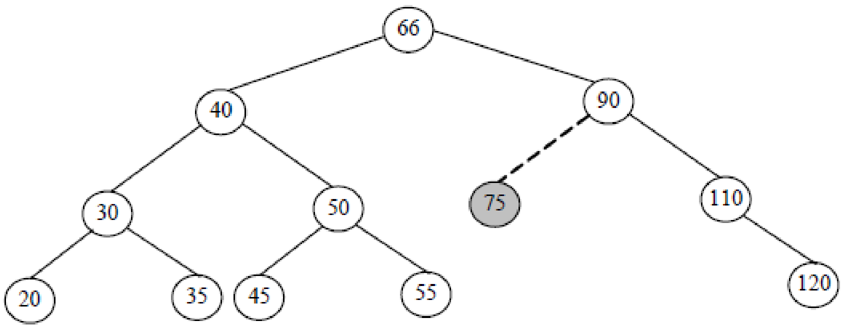
Algorithm for insertion into binary search tree

```
insert_node(item, ptr)
    1. If !(ptr)
        Allocate memory for ptr
        Set ptr->info = item
        Set ptr->left = NULL
        Set ptr->right = NULL
    Else
        If item < ptr->info
            Call insert_node(item, ptr->left)
        Else
            Call insert_node(item, ptr->right)
        End If
    End If
2. End
```

Deleting a Node

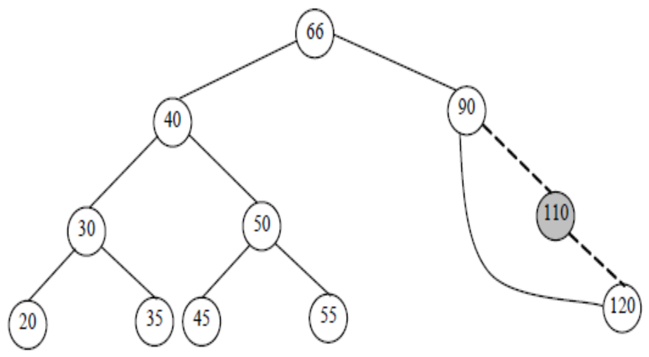
Deletion of a node from a binary search tree involves two steps—first, searching the desired node and second, deleting the node. Whenever a node is deleted from a tree, it must be ensured that the tree remains a binary search tree, that is, the sorted order of the tree must not be disturbed. The node being deleted may have zero, one or two child nodes. On the basis of the number of child nodes to be deleted, there are three cases of deletion which are discussed as follows:

Case 1: If the node to be deleted has no child node, it is deleted by making its parent's pointer pointing to NULL and de-allocating memory allocated to it.



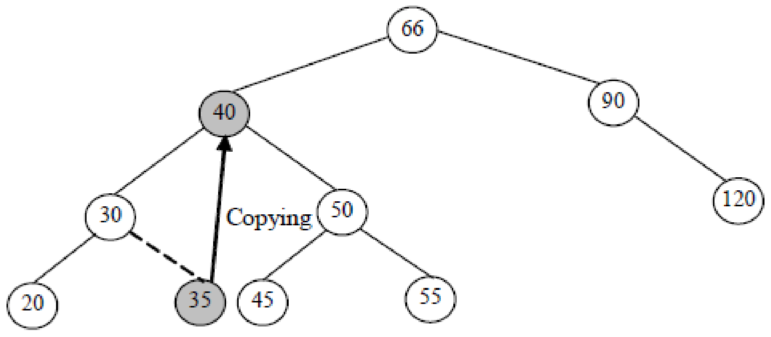
The node with value 75 is to be deleted. Since this node has no child node, its parent's (90) left pointer will be made to point to NULL and the memory space of the node (75) is de-allocated.

Case 2: If the node to be deleted has only one child node, it is deleted by adjusting its parent's pointer pointing to its only child and de-allocating memory allocated to it.

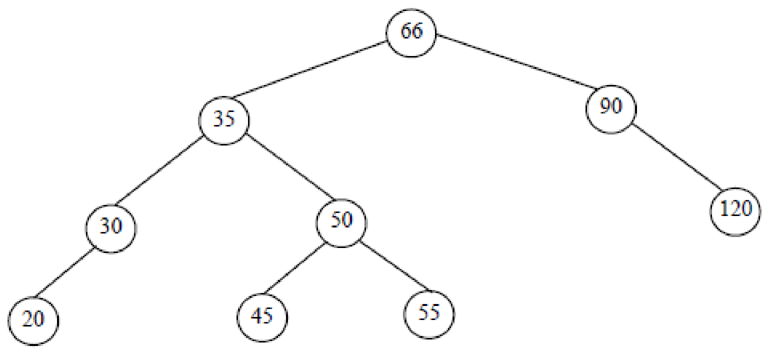


The node with value 110 is to be deleted. Since this node has one child node, its parent's (90) right pointer will be made to point to its child node (120) and memory space of the node (110) is de-allocated.

Case 3: If the node to be deleted has two child nodes, it is deleted by replacing its value by largest value in the left sub tree (in-order predecessor) or bysmallest value in the right sub tree (in-order successor). The node whose value is used for replacement is then deleted using case 1 or case 2.



The node with value 40 is to be deleted. Since this node has two sub trees or child nodes, a value has to be searched from its sub trees which can be used for its replacement. The value that will be used for replacement can either be largest value from its left sub tree (35) or smallest value from its right sub tree (45). Suppose, the value 35 is selected for this purpose, then value 35 is copied in the node with the value 40. After this, the right pointer of parent node (30) of the node used for replacement (35) is made to point to NULL and memory allocated to the node with value 35 is de-allocated. As a result of deletion of this node the order of tree is maintained. The final structure of the tree after deletion of node 40 will be:



Algorithm for deleting a node in a binary search tree

```
del_node(item, ptr)
    1. If !(ptr)
        Print "Item does not exist." and go to step 3
    2. If item < ptr->info
        Call del_node(item,&(ptr->left))
    Else
        If item > ptr->info
            Call del_node(item,&(ptr->right))
        Else
            If item = ptr->info
                Set save = ptr
                If save->right = NULL
                    Set ptr = save->left
                    Deallocate save
                Else
                    If save->left = NULL
                        Set ptr = save->right
                        Deallocate save
                    Else
                        Call del(&(save->left),save)
                    End If
                End If
            End If
        End If
    End If
4.  End
```

del(p, q) // q is the node to be deleted, p is the node whose value //is used for replacing the value in q and p is de-allocated

```
    1. If p->right != NULL
        Call del(&(p->right),q)
    Else
        Set delnode = p
        Set q->info = p->info
```

```
            Set p = p->left
            Deallocate delnode
    End If
    2. End
```

NOTES

11.4 Hashing techniques

Hashing (also known as hash addressing) is generally applied to a file F containing R records. Each record contains many fields, out of these one particular field may uniquely identify the records in the file. Such a field is known as primary key (denoted by k). The values k1, K2, in the key field are known as keys or key values.

Whenever a key is to be inserted in the hash table, a hash function is applied on it, which yields an index for the key. It is then inserted at that index in the hash table. However, since there is a finite number of locations in the hash table and virtually an infinite number of keys to be stored, it is quite possible that two distinct keys hash to the same index in the hash table. The situation in which a key hashes to an index which is already occupied by another key is called collision. It should be resolved by finding some other location to insert the new key. This process of finding another location is called collision resolution.

Although, a variety of collision resolution techniques have been developed, however, the possibility of collision should be minimized. It makes the study of hash function an implied requirement because the hash function is responsible for specifying the location for a new key. Therefore, the topic of hashing comprises two major sub-parts, namely,

- Hash functions.
- Collision resolutions.

Since, the keys are inserted by applying hash functions on them, searching a key in the hash table is straightforward. Simply, the same hash function is applied on the key to be searched, which yields the index at which it may be found. Then the key at that location is compared with the desired key and if they are matched, the search is successful.

Hash Functions

A hash function h is simply a mathematical formula that maps the key to some slot in the hash table T. Thus, we can say that the key k hashes to slot h(k), or h(k) is the hash value of key k. If the size of the hash table is N, then the index of the hash table ranges from 0 to N-1. A hash table with N slots is denoted by T[N]. If the input keys are integers, then applying hash function on them is simple.

However, if the input keys are strings, then they are first converted into integers before applying the hash function. For this, the numeric (ASCII) code associated with characters can be used in converting character values into integers.

There are a number of hash functions available, however, the one which is easy to compute and ensures that two distinct values hash to different location in the hash table is desirable.

However designing a hash function with zero collisions is impossible so, a hash function with minimum collisions is desirable.

Division-remainder method
The division-remainder is the simplest and most commonly used method. In this method, the key k is divided by the number of slots N in the hash table, and the remainder obtained after division is used as an index in the hash table. That is, the hash function is

h(k) = k mod N

Where, mod is the modulus operator.
For example, consider a hash table with N=101. The hash value of the key value 132437 can be calculated as follows:

h(132437)=132437 mod 101 = 26

Multiplication method
In multiplication method, first the key k is multiplied by a constant C, where 0 < C < 1, and the fractional part of the product kC is extracted. In the second step, this fractional part is multiplied by N, and the floor of the result is taken as the hash value. The floor of a value x denoted by floor(x) is the largest integer less than or equal to x.
That is, the hash function is

h(k) = floor(N (kC mod 1))

Where, kC mod 1 represents the fractional part of kC, calculated as kC – floor(kC).

For example, consider a hash table with N=101 and C=0.6180339. The hash value of the key 132437 can be calculated as follows:

h(132437) = floor(101 * ((132437 * 0.6180339…) mod 1))
= floor(101 * (81850.5673680698… mod 1))
= floor(101 * 0.5673680698…)
= floor(57.3041750498…)
= 57

Folded key method
The folded key is also a two-step process. In the first step, the key k is divided into several groups from the left most digits, where each group contains n number of digits, except the last one which may contain lesser number of digits. In the next step, these groups are added together, and the hash value is obtained by ignoring the last carry (if any).

For example, if the hash table has 100 slots, then each group will have two digits; and the sum of the groups after ignoring the last carry will also be a 2-digit number between 0 and 99. The hash value for the key value 132437 is computed as follows:

- The key value is first broken down into a group of 2-digit numbers from the left most digits. Therefore, the groups are 13, 24, and 37.
- 2. These groups are then added like 13 + 24 + 37 = 74. The sum 74 is now used as the hash value for the key value 132437.

Similarly, the hash value of another key value, say 6217569, can be calculated as follows:

- The key value is first broken down into a group of 2-digit numbers from the left most digits. Therefore, the groups are 62, 17, 56, and 9.
- These groups are then added, like, 62 + 17 + 56 + 9 = 144. The sum 44 after ignoring the last carry 1 is now used as the hash value for the key value 6217569.

Mid-square method

This method also operates in two steps. First, the square of the key k (that is, k2) is calculated, and then some of the digits from left and right ends of k2 are removed. The number obtained after removing the digits is used as the hash value. Note that the digits at the same position of k2 must be used for all the keys.

For example, consider a hash table with N=1000. The hash value of the key value 132437 can be calculated as follows:

- The square of the key value is calculated, which is, 17539558969.
- 2. The hash value is obtained by taking 5th, 6th and 7th digits counting from right, which is, 955.

Similarly, the hash value of another key value, say 6217569, can be calculated as follows:

- The square of the key value is calculated, which is, 38658164269761
- 2. The hash value is obtained by taking 5th, 6th and 7th digits counting from right, which is, 426.

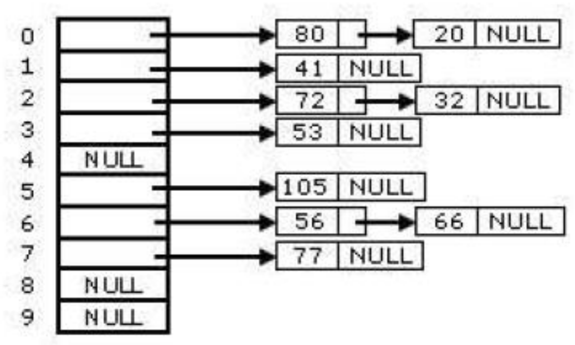
Collision Resolution Techniques

The main problem associated with most hashing functions is that they do not yield distinct hash addresses for distinct keys, because the number of key values is much larger than the number of available locations in the hash table. Due to this sometimes the problem called collision occurs. Since one cannot eliminate collisions altogether, one needs some mechanisms to deal with them. There are several ways for resolving collisions, the two most common techniques used are separate chaining and open addressing.

Separate chaining

In this technique, a linked list of all the key values that hash to the same hash value is maintained. Each node of the linked list contains a key value and the pointer to the next node. Each index i (0<=i<N) in the hash table contains the address of the first node of the linked list containing all the keys that hash to the index i. If there is no key value that hashes to the index i, the slot contains NULL value. Therefore, in this method, a slot in the hash table does not contain the actual key values; rather it contains the address of the first node of the linked list containing the elements that hash to this slot.

Consider the key values 20, 32, 41, 66, 72, 80, 105, 77, 56, and 53 that need to be hashed using the simple hash function h(k) = k mod 10. The keys 20 and 80 hash to index 0, key 41 hashes to index 1, keys 32 and 72 hashes to index 2, key 53 hashes to index 3, key 105 hashes to index 5, keys 66 and 56 hashes to index 6 and finally the key 77 hashes to index 7. The collision is handled using the separate chaining (also known as synonyms chaining) technique as shown



Note that a new element can be inserted either at the beginning or at the end of the list. Generally, the elements are inserted in the beginning of the list because it is simpler to implement, and moreover, it frequently happens that the elements which are added recently are the most likely to be accessed in the near future. The main disadvantage of separate chaining is that it makes use of pointers, which slows down the algorithm a bit because of the time required in allocating and deallocating the memory. Moreover, maintaining another data structure (that is, linked list) in addition to the hash table causes extra overheads.

Open addressing

Unlike the separate chaining method, no separate data structure is used in open addressing because all the key values are stored in the hash table itself. Since, each slot in the hash table contains the key value rather than the address value, a bigger hash table is required in this case as compared to separate chaining. Some value is used to indicate an empty slot. For example, if it is known that all the keys are positive values, then -1 can be used to represent a free or empty slot.

To insert a key value, first the slot in the hash table to which the key value hashes, is determined, using any hash function. If the slot is free, the key value is inserted into that slot. In case the slot is already occupied, then the subsequent slots, starting from the occupied slot, are examined systematically in the forward direction, until an empty slot is found. If no empty slot is found, then an overflow condition occurs.

In case of searching, first the slot in the hash table to which the key value hashes is determined, using any hash function. Then, the key value stored in that slot is compared with the key value to be searched. If they match, the search operation is successful; otherwise alternative slots are examined systematically in the forward direction to find the slot containing the desired key value. If no such slot is found, then the search is unsuccessful.

The process of examining the slots in the hash table to find the location of a key value is known as probing. The various types of probing are linear probing, quadratic probing, and double hashing that are used in open addressing method.

Linear probing

Linear probing is the simplest approach for resolving collisions. It uses the following hash function:

h(k, i) = [h'(k) + i] mod N

Where,

h'(k) is any hash function (for simplicity we use k mod N)

i is the probe number ranging from 0 to N-1

To insert a key k in the hash table, first the slot T[h'(k)] is probed. If this slot is empty, the key is inserted into the slot. Otherwise, the slots T[h'(k)+1], T[h'(k)+2], T[h'(k)+3], and so on (up to T[N-1]) are probed sequentially until an empty slot is found. If no empty slot is found up to T[N-1], we wrap around to slots T[0], T[1], T[2], and so on until an empty slot is found or we finally reach the slot T[h'(k)-1]. The main advantage of linear probing is that as long as the hash table is not full, a free slot can always be found, however, the time taken to find an empty slot can be quite large. To understand linear probing, consider the insertion of the following keys into the hash table with N=10.

126, 75, 37, 56, 29, 154, 10, 99

Further consider that the basic hash function is h'(k)=k mod N.

Step 1: The key value 126 hashes to the slot 6 as follows:

h(126, 0) = (126 mod 10 + 0) mod 10 = (6 + 0) mod 10 = 6 mod 10 = 6

Since slot 6 is empty, it is inserted into this slot.

0	1	2	3	4	5	6	7	8	9
						126			

Step 2: Next, the key value 75 hashes to the slot 5 as follows:

h(75, 0) = (75 mod 10 + 0) mod 10 = (5 + 0) mod 10 = 5 mod 10 = 5

Since slot 5 is empty, it is inserted into this slot.

0	1	2	3	4	5	6	7	8	9
					75	126			

Step 3: Next, the key value 37 hashes to the slot 7 as follows:

h(37, 0) = (37 mod 10 + 0) mod 10 = (7 + 0) mod 10 = 7 mod 10 = 7

Since slot 7 is empty, it is inserted into this slot.

0	1	2	3	4	5	6	7	8	9
					75	126	37		

Step 4: Now, the key value 56 hashes to the slot 6 as follows:

h(56, 0) = (56 mod 10 + 0) mod 10 = (6 + 0) mod 10 = 6 mod 10 = 6

Since slot 6 is not empty, the next probe sequence is computed as follows:

h(56, 1) = (56 mod 10 + 1) mod 10 = (6 + 1) mod 10 = 7 mod 10 = 7

Slot 7 is also not empty, the next probe sequence is computed as follows:

h(56, 2) = (56 mod 10 + 2) mod 10 = (6 + 2) mod 10 = 8 mod 10 = 8

Since slot 8 is empty, 56 is inserted into this slot.

0	1	2	3	4	5	6	7	8	9
					75	126	37	56	

Step 5: Next, the key value 29 hashes to the slot 9 as follows:
h(29, 0) = (29 mod 10 + 0) mod 10 = (9 + 0) mod 10 = 9 mod 10 = 9
Since slot 9 is empty, it is inserted into this slot.

0	1	2	3	4	5	6	7	8	9
					75	126	37	56	29

Step 6: Now, the key value 154 hashes to the slot 4 as follows:
h(154, 0) = (154 mod 10 + 0) mod 10 = (4 + 0) mod 10 = 4 mod 10 = 4
Since slot 4 is empty, 154 is inserted into this slot.

0	1	2	3	4	5	6	7	8	9
				154	75	126	37	56	29

Step 7: Now, the key value 10 hashes to the slot 0 as follows:
h(10, 0) = (10 mod 10 + 0) mod 10 = (0 + 0) mod 10 = 0 mod 10 = 0
Since slot 0 is empty, it is inserted into this slot.

0	1	2	3	4	5	6	7	8	9
10				154	75	126	37	56	29

Step 8: Now, the key value 99 hashes to the slot 9 as follows:
h(99, 0) = (99 mod 10 + 0) mod 10 = (9 + 0) mod 10 = 9 mod 10 = 9
Since slot 9 is not empty, the next probe sequence is computed as follows:
h(99, 1) = (99 mod 10 + 1) mod 10 = (9 + 1) mod 10 = 10 mod 10 = 0
Slot 0 is also not empty, the next probe sequence is computed as follows:
h(99, 2) = (99 mod 10 + 2) mod 10 = (9 + 2) mod 10 = 11 mod 10 = 1
Since slot 1 is empty, 99 is inserted into this slot.

0	1	2	3	4	5	6	7	8	9
10	99			154	75	126	37	56	29

In case of searching also, the same process is followed. The only difference is that instead of finding an empty slot to store a given key value, you find the slot containing the desired key value. The number of probes required in both the cases (insertion and searching) is not more than the number of slots in the hash table. Linear probing is easy to implement, but it has a disadvantage that if the hash table is relatively empty, then blocks (clusters) of occupied slots start forming.
This problem is known as primary clustering in which many such blocks are separated by free slots. For example, in the previous example, the slots

0 and 1 form one cluster of occupied slots, slots 4 to 9 form another cluster of occupied slots. These two clusters are separated by free slots 2 and 3.
Once the clusters are formed, there are more chances that subsequent insertions will also end up in one of the cluster. This further increases the size of the cluster, thereby increasing the number of probes required to find a free slot. The performance gets worse as you insert more and more values in the table. To avoid this problem, two techniques, namely, quadratic probing and double hashing are used.

Quadratic probing

In quadratic probing, the collision function is quadratic instead of linear function of i as in linear probing. That is, it uses the following hash function:

h(k, i) = [h'(k) + i2] mod N

Where,

h'(k) is any hash function (for simplicity we use k mod N) i is the probe number ranging from 0 to N-1

To insert a key k in the hash table, first the slot T[h'(k)] is probed. If this slot is empty, the key is inserted into the slot. Otherwise, the slots h'(k)+i2 are probed. That is, the indexes h'(k)+1, h'(k)+4, h'(k)+9, and so on are considered until an empty slot is found. Quadratic probing can also guarantee a successful insertion of a key as long as the hash table is at most half full, and the size of the table is a prime number. The same probe sequences are followed to search a desired key value in the hash table.

To understand quadratic probing, consider the insertion of the following keys into the hash table with N=11.

Step 1: The key value 126 hashes to the slot 5 as follows:

h(126, 0) = (126 mod 11 + 02) mod 11 = (5 + 0) mod 11 = 5

Since slot 5 is empty, it is inserted into this slot.

0	1	2	3	4	5	6	7	8	9	10
					126					

Step 2: Next, the key value 75 hashes to the slot 9 as follows:

h(75, 0) = (75 mod 11 + 02) mod 11 = (9 + 0) mod 11 = 9

Since slot 9 is empty, it is inserted into this slot.

0	1	2	3	4	5	6	7	8	9	10
					126				75	

Step 3: Next, the key value 37 hashes to the slot 4 as follows:

h(37, 0) = (37 mod 11 + 02) mod 11 = (4 + 0) mod 11 = 4

Since slot 4 is empty, it is inserted into this slot.

0	1	2	3	4	5	6	7	8	9	10
				37	126				75	

Step 4: Now, the key value 56 hashes to the slot 1 as follows:

h(56, 0) = (56 mod 11 + 02) mod 11 = (1 + 0) mod 11 = 1

Since slot 1 is empty, 56 is inserted into this slot.

0	1	2	3	4	5	6	7	8	9	10
	56			37	126				75	

Step 5: Next, the key value 29 hashes to the slot 7 as follows:

h(29, 0) = (29 mod 11 + 02) mod 11 = (7 + 0) mod 11 = 7

Since slot 7 is empty, it is inserted into this slot.

0	1	2	3	4	5	6	7	8	9	10
	56			37	126		29		75	

Step 6: Now, the key value 154 hashes to the slot 0 as follows:

h(154, 0) = (154 mod 11 + 02) mod 11 = (0 + 0) mod 11 = 0

Since slot 0 is empty, 154 is inserted into this slot.

0	1	2	3	4	5	6	7	8	9	10
154	56			37	126		29		75	

Step 7: Now, the key value 10 hashes to the slot 10 as follows:

h(10, 0) = (10 mod 11 + 02) mod 11 = (10 + 0) mod 11 = 10

Since slot 10 is empty, it is inserted into this slot.

0	1	2	3	4	5	6	7	8	9	10
154	56			37	126		29		75	10

Step 8: Now, the key value 99 hashes to the slot 0 as follows:

h(99, 0) = (99 mod 11 + 02) mod 11 = (0 + 0) mod 11 = 0

Since slot 0 is not empty, the next probe sequence is computed as follows:

h(99, 1) = (99 mod 11 + 12) mod 11 = (0 + 1) mod 11 = 1

Slot 1 is also not empty, the next probe sequence is computed as follows:

h(99, 2) = (99 mod 11 + 22) mod 11 = (0 + 4) mod 11 = 4

Slot 4 is also not empty, the next probe sequence is computed as follows:

h(99, 3) = (99 mod 11 + 32) mod 11 = (0 + 9) mod 11 = 9

Slot 9 is also not empty, the next probe sequence is computed as follows:

h(99, 4) = (99 mod 11 + 42) mod 11 = (0 + 16) mod 11 = 5

Slot 5 is also not empty, the next probe sequence is computed as follows:

h(99, 5) = (99 mod 11 + 52) mod 11 = (0 + 25) mod 11 = 3

Since slot 3 is empty, 99 is inserted into this slot.

0	1	2	3	4	5	6	7	8	9	10
154	56		99	37	126		29		75	10

Though quadratic probing eliminates primary clustering, it sometimes results in a milder form of clustering known as secondary clustering where the key values that initially hash to the same position will probe the same sequence of slots. For example, consider a key value 88 that is to be inserted into the hash table. Initially it hashes to slot 0 (as that of the key 99). Therefore, it will follow the same probe sequence, that is, 1, 4, 9, 5, 3.

Double hashing

As you have seen that both the linear and quadratic probing add increments to the initial hash value h'(k) to define a probe sequence. Linear probing adds i, and quadratic probing adds i2 to the initial hash value to find an alternative slot. Both these increments are independent of the key k. The double hashing method, on the other hand, uses a different hash function h''(k) to compute these increments. Therefore, the increments are dependent on the key. Double hashing uses the following hash function:

h(k, i) = [h'(k) + i*h''(k)] mod N

Where,

h'(k) is any hash function (for simplicity we use k mod N) h''(k) is another hash function (for simplicity we use k mod N' where N' is slightly less than N (say N-1 or N-2))

i is the probe number ranging from 0 to N-1

Initially, when a key k is to be inserted into the hash table, the first slot probed is T[h'(k)]. If this slot is empty, the key is inserted into the slot.

Otherwise, alternative slots are searched using another independent hash function (hence the name double hashing). In case of searching also, the same process is followed until the desired key value is found, or all the key values in the table are examined. To understand double hashing, consider the insertion of the following keys into the hash table with N=13.

126, 75, 37, 56, 29, 152, 35, 99

Further, consider that the basic hash function is h'(k)=k mod N and h''(k)=k mod (N-2).

Step 1: The key value 126 is hashes to the slot 9 as follows:

h(126, 0) = (126 mod 13 + 0*(126 mod 11)) mod 13 = (9 + 0) mod 13 = 9

Since slot 9 is empty, it is inserted into this slot.

0	1	2	3	4	5	6	7	8	9	10	11	12
									126			

Step 2: Next, the key value 75 hashes to the slot 10 as follows:

h(75, 0) = (75 mod 13 + 0*(75 mod 11)) mod 13 = (10 + 0) mod 13 = 10

Since slot 10 is empty, it is inserted into this slot.

0	1	2	3	4	5	6	7	8	9	10	11	12
									126	75		

Step 3: Next, the key value 37 hashes to the slot 11 as follows:

h(37, 0) = (37 mod 13 + 0*(37 mod 11)) mod 13 = (11 + 0) mod 13 = 11

Since slot 11 is empty, it is inserted into this slot.

0	1	2	3	4	5	6	7	8	9	10	11	12
									126	75	37	

Step 4: Now, the key value 56 hashes to the slot 4 as follows:

h(56, 0) = (56 mod 13 + 0*(56 mod 11)) mod 13 = (4 + 0) mod 13 = 4

Since slot 4 is empty, 56 is inserted into this slot.

0	1	2	3	4	5	6	7	8	9	10	11	12
				56					126	75	37	

Step 5: Next, the key value 29 hashes to the slot 3 as follows:

h(29, 0) = (29 mod 13 + 0*(29 mod 11)) mod 13 = (3 + 0) mod 13 = 3

Since slot 3 is empty, it is inserted into this slot.

0	1	2	3	4	5	6	7	8	9	10	11	12
			29	56					126	75	37	

Step 6: Now, the key value 152 hashes to the slot 9 as follows:

h(152, 0) = (152 mod 13 + 0*(152 mod 11)) mod 13 = (9 + 0) mod 13 = 9

Since slot 9 is not empty, the next probe sequence is computed as follows:

h(152, 1) = (152 mod 13 + 1*(152 mod 11)) mod 13 = (9 + 9) mod 13 = 5

Since slot 5 is empty, 152 is inserted into this slot.

0	1	2	3	4	5	6	7	8	9	10	11	12
			29	56	152				126	75	37	

Step 7: Now, the key value 35 hashes to the slot 9 as follows:

h(35, 0) = (35 mod 13 + 0*(35 mod 11)) mod 13 = (9 + 0) mod 13 = 9

Since slot 9 is not empty, the next probe sequence is computed as follows:

h(35, 1) = (35 mod 13 + 1*(35 mod 11)) mod 13 = (9 + 2) mod 13 = 11

Slot 11 is also not empty, the next probe sequence is computed as follows:

h(35, 2) = (35 mod 13 + 2*(35 mod 11)) mod 13 = (9 + 4) mod 13 = 0

Since slot 0 is empty, 35 is inserted into this slot.

0	1	2	3	4	5	6	7	8	9	10	11	12
35			29	56	152				126	75	37	

Step 8: Now, the key value 99 hashes to the slot as follows:

h(99, 0) = (99 mod 13 + 0*(99 mod 11) mod 13 = (8 + 0) mod 13 = 8

Since slot 8 is empty, 99 is inserted into this slot.

0	1	2	3	4	5	6	7	8	9	10	11	12
35			29	56	152			99	126	75	37	

Since the increment in double hashing depends on the value of key k, the values that hash to the same initial slot may have different probe sequences. Thus, double hashing almost eliminates the problem of primary and secondary clustering and its performance is very close to the ideal hashing. For example, the key value 35 initially hashes to slot 9 (as that of the key 152). However, the next probe sequence for 35 is 11 (not 5 as in case of 152).

11.5 Check Your Progress Questions

1) What is hashing also known as?
2) What is a hash function?
3) What is the main problem associated with most hashing functions?

11.6 Answers to Check Your Progress Questions

1) Hashing is also known as hash addressing.
2) A hash function h is simply a mathematical formula that maps the key to some slot in the hash table T.
3) The main problem associated with most hashing functions is that they do not yield distinct hash addresses for distinct keys

11.7 Summary

- Insertion in a binary search tree is similar to the procedure for searching an element in a binary search tree.
- The process of inserting a node in a binary search tree can be divided into two steps—in the first step, the tree is searched to determine the appropriate position where the node is to be inserted and in the second step, the node is inserted at this searched position.
- Here are two cases of insertion in a tree—first, insertion into an empty tree and second, insertion into a nonempty tree.
- The process of inserting a node in a binary search tree can be divided into two steps—in the first step, the tree is searched to determine the appropriate position where the node is to be inserted and in the second step, the node is inserted at this searched position.
- Deletion of a node from a binary search tree involves two steps—first, searching the desired node and second, deleting the node.
- If the node to be deleted has two child nodes, it is deleted by replacing its value by largest value in the left sub tree (in-order predecessor) or by smallest value in the right sub tree (in-order successor).
- Hashing (also known as hash addressing) is generally applied to a file F containing R records.
- Whenever a key is to be inserted in the hash table, a hash function is applied on it, which yields an index for the key.
- Since, the keys are inserted by applying hash functions on them, searching a key in the hash table is straightforward.
- A hash function h is simply a mathematical formula that maps the key to some slot in the hash table T.

- There are a number of hash functions available, however, the one which is easy to compute and ensures that two distinct values hash to different location in the hash table is desirable.
- The main problem associated with most hashing functions is that they do not yield distinct hash addresses for distinct keys, because the number of key values is much larger than the number of available locations in the hash table.
- There are several ways for resolving collisions, the two most common techniques used are separate chaining and open addressing.
- In this technique, a linked list of all the key values that hash to the same hash value is maintained.
- The main disadvantage of separate chaining is that it makes use of pointers, which slows down the algorithm a bit because of the time required in allocating and deallocating the memory.
- Unlike the separate chaining method, no separate data structure is used in open addressing because all the key values are stored in the hash table itself.

11.8 Key Words

- **Hash table**: It is a data structure that is used to store keys/value pairs. It uses a hash function to compute an index into an array in which an element will be inserted or searched.
- **Division-remainder method**: It is the simplest and most commonly used method. In this method, the key k is divided by the number of slots N in the hash table, and the remainder obtained after division is used as an index in the hash table.

11.9 Self-Assessment Questions and Exercises

1) What are the advantages of handling exception?
2) What are the types of exceptions?
3) Write a simple program to illustrate exception handling in java using try and catch.
4) What is the use of finally clause? Give examples
5) Write short note on uncaught exceptions
6) Briefly explain about nested try statements.
7) What is the use of throw keyword? Explain with suitable example.
8) What is the use of throws keyword? Explain with suitable example.
9) Distinguish between throw and throws keyword.

11.10 Further Readings

Storer, J.A. 2012. An Introduction to Data Structures and Algorithms. New York: Springer Publishing.

Preiss, Bruno. 2008. Data Structures and Algorithms with Object-Oriented Design Patterns in C++. London: John Wiley and Sons.

Pandey, Hari Mohan. 2009. Data Structures and Algorithms. New Delhi: Laxmi Publications.

Goodrich Michael, Tamassia Roberto and Michael H. Goldwasser. 2014. Data Structures and Algorithms in Java. London: John Wiley and Sons.

McMillan, Michael. 2007. Data Structures and Algorithms Using C#. Cambridge, UK: Cambridge University Press.

NOTES

BLOCK - IV SEARCHING TECHNIQUES

UNIT- XII SEARCHING

Structure

12.1 Introduction

Searching is the process of finding a given value position in a list of provided values. Searching helps to decide whether a search key is present in the data or not. It can be defined as the algorithmic process of finding a particular item in a collection of items. Searching can be done on both internal data structure and on external data structures.

12.2 Objectives

After going through this unit, you will be able to:

- Learn about searching techniques
- Understand about linear search
- Explain binary search
- Discuss about interpolation search

12.3 Searching

While solving a problem, a programmer may need to search a value in an array. The process of finding the occurrence of a particular data item in a list is known as searching. Search is said to be successful or unsuccessful depending on whether the data item is found or not. The searching techniques are:

- Linear search
- Binary search

12.4 Linear Search

Linear search is one of the simplest searching techniques. In this technique, the array is traversed sequentially from the first element until the value is found or the end of the array is reached. While traversing, each element of the array is compared with the value to be searched, and if the value is found the search is said to be successful. This technique is suitable for performing a search in a small array or in an unsorted array.

NOTES

Algorithm for Linear Search

```
linear_search (ARR, size, item)
        1. Set i = 0
        2. While i < size
                If ARR [i] = item //item is the value to be searched
                        Return i and go to step 4
                End If
                Set i = i + 1
        End While
        3. Return -1 //search unsuccessful
4. End
```

Step 1 - Read the search element from the user.

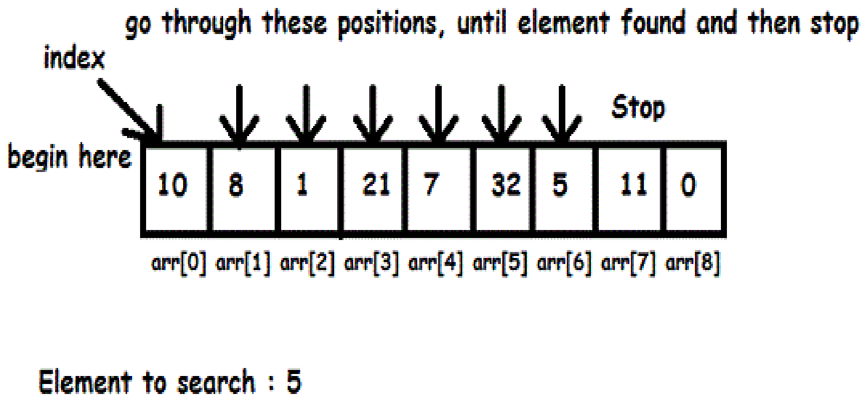
Step 2 - Compare the search element with the first element in the list.

Step 3 - If both are matched, then display "Given element is found!!!" and terminate the function

Step 4 - If both are not matched, then compare search element with the next element in the list.

Step 5 - Repeat steps 3 and 4 until search element is compared with last element in the list.

Step 6 - If last element in the list also doesn't match, then display "Element is not found!!!" and terminate the function.



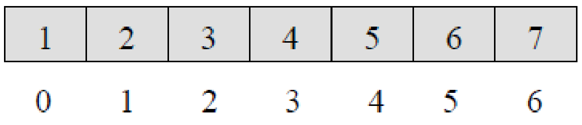
Advantages - Linear Search

When a key element matches the first element in the array, then linear search algorithm is best case because executing time of linear search algorithm is 0 (n), where n is the number of elements in an array.

Disadvantages - Linear Search
Inversely, when a key element matches the last element in the array or a key element doesn't matches any element then linear search algorithm is a worst case.

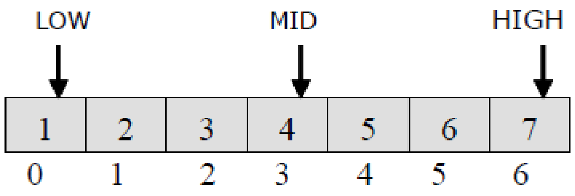
12.5 Binary Search

The binary search technique is used to search for a particular element in a sorted (in ascending or descending order) array. In this technique, the element to be searched (say, item) is compared with the middle element of the array. If item is equal to the middle element, then the search is successful. If item is smaller than the middle element, item is searched in the segment of the array before the middle element. However, if item is greater than the middle element, item is searched in the array segment after the middle element. This process is repeated until the element is found or the array segment is reduced to a single element that is not equal to item.
At every stage of the binary search technique, the array is reduced to a smaller segment. It searches a particular element in the lowest possible number of comparisons. Hence, the binary search technique is used for larger and sorted arrays, as it is faster compared to linear search. For example, consider an array ARR shown here:

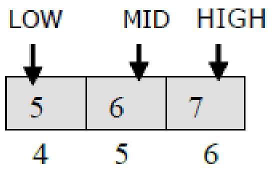


To search an item (say, 7) using binary search in the array ARR with size=7, the following steps are performed.

- Initially, set LOW=0 and HIGH=size–1. The middle of the array is determined using the formula MID=(LOW+HIGH)/2, that is, MID=(0+6)/2, which is equal to 3. Thus, ARR [MID]=4.



- Since the value stored at ARR [3] is less than the value to be searched, that is 7, the search process is now restricted from ARR[4] to ARR[6]. Now LOW is 4 and HIGH is 6. The middle element of this segment of the array is calculated as MID=(4+6)/2, that is, 5. Thus, ARR[MID]=6.



NOTES

- The value stored at ARR[5] is less than the value to be searched, hence the search process begins from the subscript 6. As ARR[6] is the last element, the item to be searched is compared with thisvalue. Since ARR[6] is the value to be searched, the search is successful.

Algorithm for binary search

```
binary_search(ARR, size, item)
        1. Set LOW = 0
        2. Set HIGH = size - 1
        3. While LOW <= HIGH
                Set MID = (LOW + HIGH) / 2
                If ITEM = ARR[MID]
                        Return MID and go to step 5
                Else If item < ARR[MID]
                        Set HIGH = MID – 1
                Else
                        Set LOW = MID + 1
                End If
        End While
        4. Return -1
5. End
```

Advantages

- It is comparatively faster than linear search.
- It can be used for large amount of data.
- It takes lesser amount of time.

Disadvantages

- Binary search requires that the items in the array should be sorted.
- It works only on element types for which there exists a less-than relationship.

12.6 Check Your Progress Questions

1) What is the process of finding the occurrence of a particular data item in a list is known as?
2) What is interpolation search similar to?
3) What happens in an interpolation search if the elements are not uniformly distributed?

12.7 Answers to Check Your Progress Questions

1. The process of finding the occurrence of a particular data item in a list is known as searching.

2. Interpolation search is similar to binary search in the sense that it also applies on sorted arrays.
3. If the elements are not uniformly distributed, then interpolation search gives a very poor performance.

12.8 Summary

- The process of finding the occurrence of a particular data item in a list is known as searching.
- The various search techniques are linear binary, Fibonacci and interpolation searce.hes.
- Linear search is one of the simplest searching techniques. In this technique, the array is traversed sequentially from the first element until the value is found or the end of the array is reached.
- The binary search technique is used to search for a particular element in a sorted (in ascending or descending order) array.
- At every stage of the binary search technique, the array is reduced to a smaller segment.
- Interpolation search is similar to binary search in the sense that it also applies
- In the best case, when the item is found at first position, the search operation terminates successfully with only one comparison.
- In each iteration, binary search algorithm reduces the array to one half. Therefore, for an array containing n elements, there will be log2n iterations.
- The performance of interpolation search is highly dependent on the distribution of elements in the list.
- If the elements are not uniformly distributed, then interpolation search gives a very poor performance.

12.9 Key Words

- **Linear search**: It is one of the simplest searching technique, where the array is traversed sequentially from the first element until the value is found or the end of the array is reached.
- **Binary search**: It is a search technique which is used to search for a particular element in a sorted (in ascending or descending order) of array.

12.10 Self-Assessment Questions and Exercises

Short-Answer Questions

1) Write a short note on searching.
2) What do you mean by linear search?

3) Write a note on binary search.
4) Write an algorithm for binary search.

Long-Answer Questions

1) Write a program to perform linear search.
2) Write a program to perform binary search.
3) "The binary search process is repeated until the element is found or the array segment is reduced to a single element that is not equal to item." Discuss.

12.11 Further Readings

Storer, J.A. 2012. An Introduction to Data Structures and Algorithms. New York: Springer Publishing.

Preiss, Bruno. 2008. Data Structures and Algorithms with Object-Oriented Design Patterns in C++. London: John Wiley and Sons.

Pandey, Hari Mohan. 2009. Data Structures and Algorithms. New Delhi: Laxmi Publications.

Goodrich Michael, Tamassia Roberto and Michael H. Goldwasser. 2014. Data Structures and Algorithms in Java. London: John Wiley and Sons.

McMillan, Michael. 2007. Data Structures and Algorithms Using C#. Cambridge, UK: Cambridge University Press.

BLOCK –V SORTING TECHNIQUES

NOTES

UNIT - XIII SORTING

Structure

13.1 Introduction

Sorting refers to the way in which data is arranged in a particular order. A sorting algorithm is employed to rearrange a given array elements according to a comparison operator on the elements. Sorting improves the efficiency of searching and also simplifies the processing of data. There are two types:

- Internal Sorting- Here all the records of the file to be sorted should be within the main memory at the time of sorting. Information can be retrieved easily. Ex: Bubble, Insertion, Selection, Quick Sort
- External Sorting- Some of the files to be sorted can be kept in secondary storage at the time of sorting. Ex: Merge Sort.

This unit will first begin with the definition of sorting. It will then discuss Bubble Sort, Insertion Sort and Radix Sort.

13.2 Objectives

After going through this unit, you will be able to:

- Discuss sorting
- Describe the process of insertion sorting, bubble sorting and bucket sorting
- Write program using Byte stream classes
- Able to understand about character stream classes

13.3 Definition

The process of arranging the data in some logical order is known as sorting. The order can be ascending or descending for numeric data, and alphabetically for character data. There are two types of sorting, namely, internal sorting and external sorting. If all the data that is to be sorted fits entirely in the main memory, then internal (in-memory) sorting is used.

On the other hand, if all the data that is to be sorted do not fit entirely in the main memory, external sorting is required. An external sorting requires the use of external memory such as disks or tapes during sorting. In external sorting, some part of the data is loaded into the main memory, sorted using any internal sorting technique and written back to the disk in some intermediate file. This process continues until all the data is sorted.

Internal Sorting

There are different internal sorting algorithms such as insertion sort, bubble sort, selection sort, heap sort, merge sort, quick sort and bucket sort. The choice of a particular algorithm depends on the properties of the data and the operations to be performed on the data. For all these algorithms, we will consider an array ARR containing n elements, which are to be sorted in an ascending order.

13.4 Bubble sort

The bubble sort algorithm requires n-1 passes to sort an array. In the first pass, each element (except the last) in the list is compared with the element next to it, and if one element is greater than the other then both the elements are swapped. After the first pass, the largest element in the list is placed at the last position. Similarly, in the second pass the second largest element is placed at its appropriate position. Thus, in each subsequent pass, the next largest element is placed at its appropriate position. Since this algorithm makes the larger values to 'bubble up' to the end of the list, it is named bubble sort.

The bubble sort algorithm possesses an important property. This property is that if a particular pass is made through the list without swapping any items, then there will be no further swapping of elements in the subsequent passes. This property can be used to eliminate the unnecessary passes once the list is sorted in the desired order. For this, a flag variable can be used to detect if any interchange has been made during the pass. We use flag = 0 to indicate that no swaps have occurred in a particular pass, therefore, no further passes are required.

It derives its name from the fact that the smallest data item bubbles up to the top of the sorted array. The array is scanned from bottom up / top down and two adjacent elements are swapped if they are found to be out of order.

NOTES

Algorithm for bubble sort

<u>Algorithm BubbleSort(A,n)</u>

```
    {
        for ( i=0; i<n; i++)
        {
            for ( j=n-1; j>i; j--)
            {
                if ( A[j] < A[j-1] )
                {
                    t=A[j]
                    A[j] =A [j-1]
                    A [j-1] =t
                }
            }
        }
    }
```

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**)

Advantages:

- It's a simple algorithm.
- Efficient way to check if a list is already in order.
- Doesn't use too much memory.

Disadvantages:

- Due to being efficient, the bubble sort algorithm is pretty slow for very large lists of items.

13.5 Insertion Sort

The insertion sort algorithm selects each element and inserts it at its proper position in the earlier sorted sub-list. In the first pass, the element ARR[1] is compared with ARR[0], and if ARR[1] and ARR[0] are not sorted, they are swapped. In the second pass, the element ARR[2] is compared with ARR[0] and ARR[1], and it is inserted at its proper position in the sorted sub-list containing the elements ARR[0], ARR[1]. Similarly, during ith iteration, the element ARR[i] is placed at its proper position in the sorted sub-list containing the elements ARR[0], ARR[1], ARR[2],…, ARR[i-1].

In order to determine the actual position of the element (say, ARR[i]) in the sorted sub-list containing the elements ARR[0], ARR[1], …, ARR[i- 1], the element ARR[i] is compared with all other elements to its left, until an element ARR[j] is found such that ARR[j]<=ARR[i]. Now, to insert the element at its actual position, all the elements ARR[i-1], ARR[i-2], ARR[i- 3],…, ARR[j+1] are shifted one position towards the right to create the space for ARR[i], and then ARR[i] is inserted at (j+1)st position.

Algorithm for Insertion sort

Algorithm Insertion Sort(A,n)

```
{
    for( i=1; i<n; i++)
    {
        t=A[i]
        for( j=i; j>0; j--)
        {
            if( A[j] < A[j-1] )
            {
                A[j]=A[j-1]
            }
            else
                break;
        }
        A[j]=t
    }
}
```

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

NOTES

Advantages:

- The main advantage of the insertion sort is its simplicity.
- It also exhibits a good performance when dealing with a small list.
- The insertion sort is an in-place sorting algorithm so the space requirement is minimal.

Disadvantages:

- With n-squared steps required for every n element to be sorted, the insertion sort does not deal well with a huge list.
- The insertion sort is particularly useful only when sorting a list of few items
-

13.6 Radix Sort

The radix or bucket sort algorithm sorts the numbers by considering individual digits starting from right to left. In the first pass, the numbers are sorted according to the digits at units place. In the second pass, the numbers are sorted according to the digits at tens place, and so on. Since the base of decimal numbers is 10, the radix sort requires ten buckets, numbered 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9 to store the array elements in each pass. The number of passes in the algorithm is equal to the number of digits in the largest number. Therefore, the algorithm first determines the largest number in the list and counts the number of digits in it.
In the first pass, the numbers having 0 at units place are placed in bucket 0. Numbers having 1 at their units place are placed in bucket 1, numbers having 2 at their units place are placed in bucket 2 and so on. The elements are then retrieved from these buckets (starting from bucket 0 till bucket 9) and copied in the original array. At this point, the numbers are sorted based on the digits at units place. This list becomes the input for the second pass. In the second pass, the numbers having 0 at their tens place are placed in bucket 0. Numbers having 1 at their tens place are placed in bucket 1, numbers having 2 at their tens place are placed in bucket 2 and so on. The elements are then retrieved from these buckets (starting from bucket 0 till bucket 9) and copied in the original array. At this point, the numbers are sorted based on the digits at tens place. Now, this array becomes the input for the third pass. This process is repeated d times, where d represents the number of digits in the largest number of the list.

```
Algorithm radixsort()
{
        f=1;
        for(i=1;i<=d;i++)
        {
                for(k=0;k<10;k++)
                {
                        r[k]= -1;
                }
                for (j=0;j<n;j++)
                {
                        ind=(data[j]/f)%10;
                        r[ind]=r[ind]+1;
                        q[ind][r[ind]]=data[j];
                }
                for(k=0,m=0;k<10;k++)
                {
                        for(l=0;l<=r[k];l++)
                        {
                                data[m]=q[k][l];
                                m=m+1;
                        }
                }
                f=f*10;
        }
}
```

Example: Assume the input array is:
10,21,17,34,44,11,654,123
Based on the algorithm, we will sort the input array according to the **one's digit** (least significant digit).
0: 10
1: 21 11
2:
3: 123
4: 34 44 654
5:
6:
7: 17
8:
9:

So, the array becomes 10,21,11,123,24,44,654,17
Now, we'll sort according to the **ten's digit**:
0:
1: 10 11 17
2: 21 123
3: 34
4: 44
5: 654
6:
7:
8:
9:

Now, the array becomes : 10,11,17,21,123,34,44,654
Finally , we sort according to the **hundred's digit** (most significant digit):
0: 010 011 017 021 034 044
1: 123
2:
3:
4:

NOTES

5:
6: 654
7:
8:
9:

The array becomes : 10,11,17,21,34,44,123,654 which is sorted. This is how our algorithm works.

Advantages:

- Fast when the keys are short i.e. when the range of the array elements is less.

Disadvantages:

- Since Radix Sort depends on digits or letters, Radix Sort is much less flexible than other sorts. Hence for every different type of data it needs to be rewritten.
- The constant for Radix sort is greater compared to other sorting algorithms.
  It takes more space compared to Quicksort.

13.4 Check Your Progress Questions

1) Define sorting.
2) List the two types of sorting.
3) What is the important property of the bubble sort algorithm?
4) How does the bucket sort algorithm sort the numbers? Define sorting.
5) List the two types of sorting.
6) What is the important property of the bubble sort algorithm?
7) How does the bucket sort algorithm sort the numbers?

13.5 Answers to Check Your Progress Questions

1) The process of arranging the data in some logical order is known as sorting.
2) There are two types of sorting, namely, internal sorting and external sorting.
3) An important property of the bubble sort algorithm is that if a particular pass is made through the list without swapping any items, then there will be no further swapping of elements in the subsequent passes.
4) The radix or bucket sort algorithm sorts the numbers by considering individual digits starting from right to left.

13.6 Summary

- The process of arranging the data in some logical order is known as sorting. The order can be ascending or descending for numeric data, and alphabetically for character data.
- There are two types of sorting, namely, internal sorting and external sorting
- If all the data that is to be sorted do not fit entirely in the main memory, external sorting is required
- The bubble sort algorithm requires n-1 passes to sort an array. In the first pass, each element (except the last) in the list is compared with the element next to it, and if one element is greater than the other then both the elements are swapped.
- The insertion sort algorithm selects each element and inserts it at its proper position in the earlier sorted sub-list.
- The radix or bucket sort algorithm sorts the numbers by considering individual digits starting from right to left.

13.7 Key Words

- **Sorting**: It refers to arranging data in a particular format.
- **Bucket Sort:** It is a sorting algorithm that works by distributing the elements of an array into a number of buckets.
- **Insertion Sort:** It is a sorting algorithm in which the elements are transferred one at a time to the right position.

13.8 Self-Assessment Questions and Exercises

Short-Answer Questions

1) What are the two types of sorting?
2) What does external sorting require?
3) How does insertion sort work?

Long-Answer Questions

1) Describe the steps to sort the values stored in the array in ascending order using bubble sort.
2) Write a program showing sorting of an array using bubble sort and insertion sort.
3) Examine the steps to sort values using radix sort.

NOTES

13.9 Further Readings

Storer, J.A. 2012. An Introduction to Data Structures and Algorithms. New York: Springer Publishing.
Preiss, Bruno. 2008. Data Structures and Algorithms with Object-Oriented Design Patterns in C++. London: John Wiley and Sons.
Pandey, Hari Mohan. 2009. Data Structures and Algorithms. New Delhi: Laxmi Publications.
Goodrich Michael, Tamassia Roberto and Michael H. Goldwasser. 2014. Data Structures and Algorithms in Java. London: John Wiley and Sons.
McMillan, Michael. 2007. Data Structures and Algorithms Using C#. Cambridge, UK: Cambridge University Press.

UNIT-XIV
OTHER SORTING TECHNIQUES

Structure

14.1 Introduction

In this unit, you will be introduced with other sorting techniques such as selection sort, quick sort and tree sort.

14.2 Objectives

After going through this unit, you will be able to:

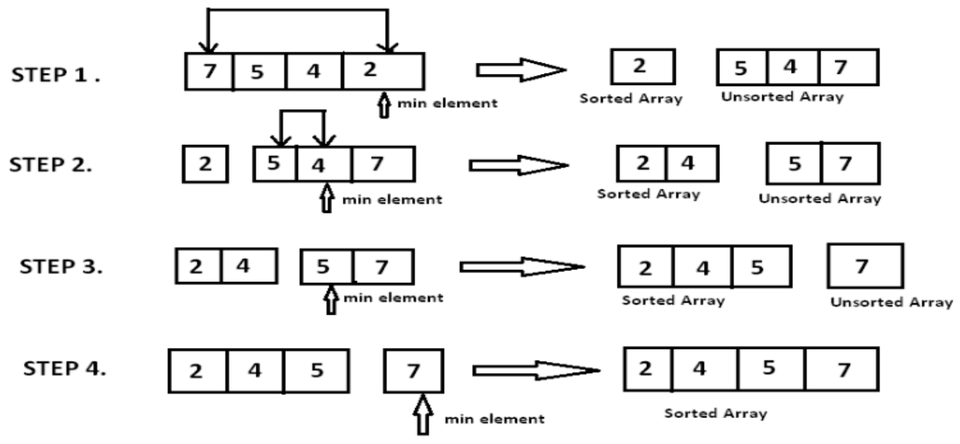
- Define selection sort, quick sort and tree sort
- Write programs using selection, quick and tree sorting techniques

14.3 Selection sort

In selection sort, first, the smallest element in the list is searched and is swapped with the first element in the list (that is, it is placed at the first position). Then, the second smallest element is searched and swapped with the second element in the list (that is, it is placed at the second position), and so on.
Like bubble sort algorithm, the selection sort also requires n-1 passes to sort an array containing n elements. However, there is a slight difference between the selection sort and the bubble sort algorithms. In selection sort, the smallest element is the first one to be placed at its correct position, then the second smallest element takes its position, and so on. Whereas, in bubble sort, the largest element is the first one to be placed at its appropriate position, then the second largest element, and so on.

Algorithm for selection sort

```
Algorithm SelectionSort(A,n)
{
        for(i=0;i<n-1;i++)
        {
                least=i;
                for(j=i+1;j<n;j++)
                {
                        if(data[least]>data[j])
                        {
                                least=j;
                        }
                }

                t=data[i];
                data[i]=data[least];
                data[least]=t;
        }
}
```



Advantages:

- It does not depend on the initial arrangement of data. It is easy to implement.

Disadvantage:

- Sorting shows poor efficiency when dealing with larger lists.

14.4 Quick Sort

Quick sort algorithm also follows the principle of divide-and-conquer. However, it does not simply divide the list into halves. Rather it first picks up a partitioning element, called pivot that divides the list into two sub-lists. This is done in a never that all the elements in the left sub-list are smaller than the pivot, and all the elements in the right sub-list are greater than the pivot. The same process is applied on the left and right sub-lists separately. This process is repeated recursively until each sub-list contains not more than one element.

The main task in quick sort is to find the pivot that partitions the given list into two halves so that the pivot is placed at its appropriate

location in the array. The choice of pivot has a significant effect on the efficiency of quick sort algorithm. The simplest way is to choose the first element as pivot. However, the first element is not a good choice, especially if the given list is already or nearly ordered. For better efficiency, the middle element can be chosen as a pivot. Note that, the first element is as taken as pivot for simplicity.
The steps involved in quick sort algorithm are as follows:

- Initially, three variables pivot, beg and end are taken, such that both pivot and beg refer to the 0th position, and end refers to (n-1)th position in the list.
- Starting with the element referred to by end, the array is scanned from right to left, and each element on the way is compared with the element referred to by pivot. If the element referred to by pivot is greater than the element referred to by end, they are swapped and step 3 is performed. Otherwise, end is decremented by 1 and step 2 is continued.
- Starting with the element referred to by beg, the array is scanned from left to right, and each element on the way is compared with the element referred to by pivot. If the element referred to by pivot is smaller than the element referred to by end, they are swapped and step 2 is performed. Otherwise, beg is incremented by 1 and step 3 is continued.

The first pass terminates when pivot, beg and end all refer to the same array element. This indicates that the pivot element is placed at its final position. The elements to the left of this element are smaller than this element, and elements to its right are greater.

Algorithm for quick sort

```
quick_sort(ARR, size, lb, ub)
        1. Set i = 1 //i is a static integer variable
        2. If lb < ub
                Call splitarray(ARR, lb, ub) //returning an integer value
        pivot
                Print ARR after ith pass
                Set i = i + 1
                Call quick_sort(ARR, size, lb, pivot – 1) //recursive call to
        quick_sort() to
                                                              //sort left sub
                                                      list
                Call quick_sort(ARR, size, pivot + 1, ub); //recursive call to
        quick_sort() to
                                                      //sort right sub list
        Else if (ub=size-1)
                Print “No. of passes: ”, i
        End If
3. End
```

splitarray(ARR, lb, ub) //spiltarray partitions the list into two sub lists such that the elements in left sub list are smaller than ARR[pivot], and elements in the right sub list are greater than ARR[pivot]

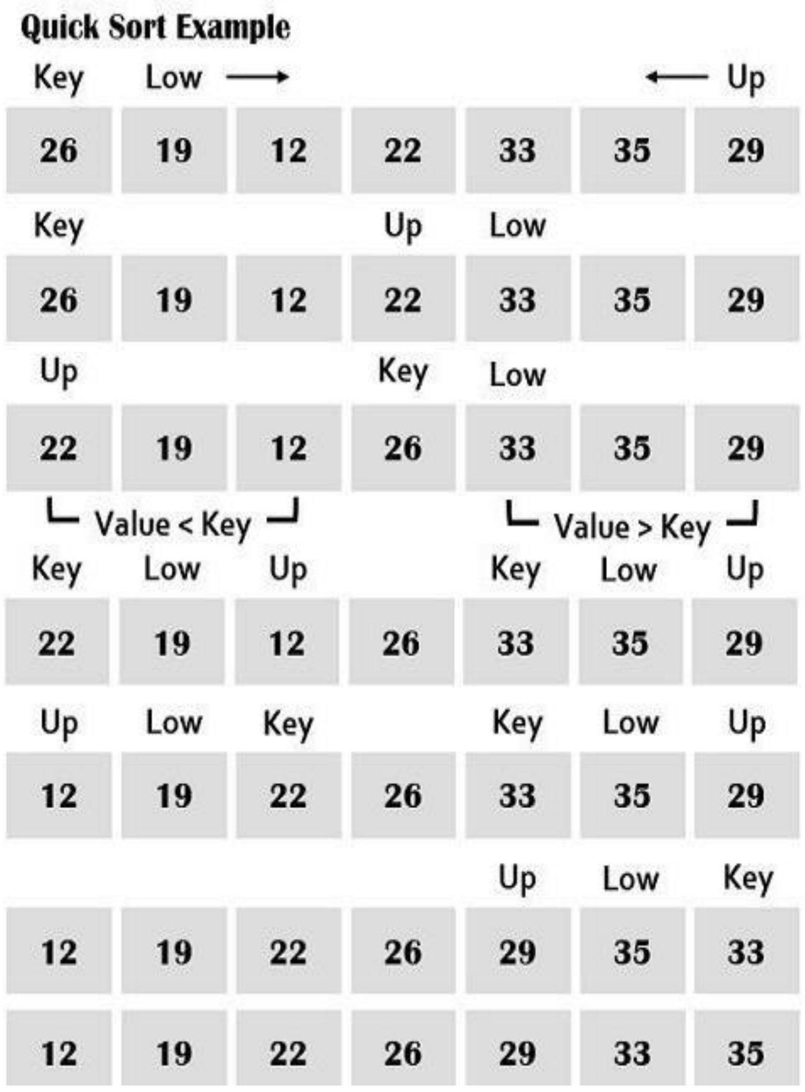
```
    1. Set flag = 0, beg = pivot = lb, end = ub
    2. While (flag != 1)
        While (ARR[pivot] <= ARR[end] AND pivot != end)
            Set end = end – 1
        End While
        If pivot = end
            Set flag = 1
        Else
            Set temp = ARR[pivot]
            Set ARR[pivot] = ARR[end]
            Set ARR[end] = temp
            Set pivot = end
        End If
        If flag != 1
            While (ARR[pivot] >= ARR[beg] AND pivot !=
        beg)
                Set beg = beg + 1
            End While
            If pivot = beg
                Set flag = 1
            Else
                Set temp = ARR[pivot]
                Set ARR[pivot] = ARR[beg]
                Set ARR[beg] = temp
                Set pivot = beg
            End If
        End If
    End While
    3. Return pivot
4. End
```

Advantages:

- One of the fastest algorithm.
- Does not need additional memory
- Quick sort can be easily parallelized due to its divide-and-conquer nature.

Disadvantages

- This algorithm may swap the elements with equal comparison keys (it is not a stable sort).
- Quick sort does work very well on already mostly sorted lists or an lists with lots of similar values. It is recursive.



14.5 Tree Sort

A tree sort is a sort algorithm that builds a binary search tree from the elements to be sorted. It then traverses the tree so that the elements come out in a sorted order. Its typical use is to sort elements online. As an insertion is completed, the set of elements seen so far is available in sorted order.

- Take the elements in an array.
- Create a Binary search tree by inserting data items from the array into the binary search tree.
- Perform in-order traversal on the tree to get the elements in sorted order.

Algorithm for tree sort

```
STRUCTURE BinaryTree
    BinaryTree:LeftSubTree
    Object:Node
    BinaryTree:RightSubTree

PROCEDURE Insert(BinaryTree:searchTree, Object:item)
    IF searchTree.Node IS NULL THEN
        SET searchTree.Node TO item
    ELSE
        IF item IS LESS THAN searchTree.Node THEN
            Insert(searchTree.LeftSubTree, item)
        ELSE
            Insert(searchTree.RightSubTree, item)

PROCEDURE InOrder(BinaryTree:searchTree)
    IF searchTree.Node IS NULL THEN
        EXIT PROCEDURE
    ELSE
        InOrder(searchTree.LeftSubTree)
        EMIT searchTree.Node
        InOrder(searchTree.RightSubTree)

PROCEDURE TreeSort(Collection:items)
    BinaryTree:searchTree
    FOR EACH individualItem IN items
        Insert(searchTree, individualItem)
        InOrder(searchTree)
```

Advantages

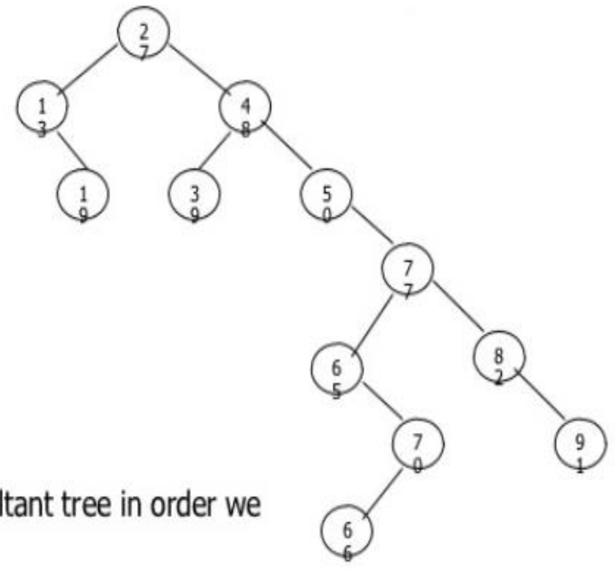
- The main advantage of tree sort algorithm is that we can make changes very easily as in a linked list.
- Sorting in Tree sort algorithm is as fast as quick sort algorithm.

Disadvantages

- The worst case occur when the elements in an array is already sorted.
- In worst case, the running time of tree sort algorithm is 0 (n^2).

Tree sort Example

- Consider the following unsorted list of numbers:
  - 27 48 13 50 39 77 82 91 65 19 70 66
- Creating the binary search tree for this given list:



- Traversing this resultant tree in order we get the sorted list:

13 19 27 39 48 50 65 66 70 77 82 91.

14.11 Check Your Progress Questions

1) What element in the list is searched in selection sort?
2) What principles does the quick sort algorithm follow?
3) What is tree sort?

14.12 Answers to Check Your Progress Questions

1) In selection sort, first, the smallest element in the list is searched and is swapped with the first element in the list (that is, it is placed at the first position).
2) Quick sort algorithm also follows the principle of divide-and-conquer.
3) Tree sort is a sorting algorithm that is based on Binary Search.

14.13 Summary

- In selection sort, first, the smallest element in the list is searched and is swapped with the first element in the list (that is, it is placed at the first position). Then, the second smallest element is searched and swapped with the second element in the list (that is, it is placed at the second position), and so on.
- Like bubble sort algorithm, the selection sort also requires n-1 passes to sort an array containing n elements.
- Quick sort algorithm also follows the principle of divide-and-conquer. However, it does not simply divide the list into halves. Rather it first picks up a partitioning element, called pivot that divides the list into two sub-lists. A tree sort is a sort algorithm that builds a binary search tree from the elements to be sorted, and then traverses the tree so that the elements come out in sorted order.

.

14.14 Key Words

- **Selection Sort**: It is a sorting algorithm that starts by finding the minimum value in the array and moving it to the first position.
- **Quick Sort**: It is a popular sorting algorithm utilizes a divide-and-conquer strategy to quickly sort data items by dividing a large array into two smaller arrays.
- **Tree Sort:** It is a sort algorithm that builds a binary search tree from the elements to be sorted, and then traverses the tree (in-order) so that the elements come out in sorted order

14.15 Self-Assessment Questions and Exercises

Short-Answer Questions

1) Differentiate between bubble sort and selection sort.
2) How does the quick sort algorithm sort data?

Long-Answer Questions

1) Describe the steps to sort the values stored in the array in ascending order using selection sort.
2) Write a program showing sorting of an array using selection sort and quick sort.

14.16 Further Readings

Storer, J.A. 2012. An Introduction to Data Structures and Algorithms. New York: Springer Publishing.
Preiss, Bruno. 2008. Data Structures and Algorithms with Object-Oriented Design Patterns in C++. London: John Wiley and Sons.
Pandey, Hari Mohan. 2009. Data Structures and Algorithms. New Delhi: Laxmi Publications.
Goodrich Michael, Tamassia Roberto and Michael H. Goldwasser. 2014. Data Structures and Algorithms in Java. London: John Wiley and Sons.
McMillan, Michael. 2007. Data Structures and Algorithms Using C#. Cambridge, UK: Cambridge University Press.

*****