

Data Structures & Algorithms

(EECP 2280)

Semester 1/2020-21



ENGINEERING DEPARTMENT

EECP2280	Data Structures & Algorithms	3 Credit Hours
Prerequisites:	None	
Goal	Provide student with wide range of experiences in using the algorithms and data structures that underpins much of today's computing and connecting these experiences with the computer problem solving techniques.	
Objectives		Outcomes

At the end of the semester, this course should enable the students to:

1. Specify and implement Abstract Data Type (ADT).
2. Evaluate and implement data structure
3. Justify the selection of an appropriate data structure/algorithm

The students should be able to understand the following:

4. 1. Produce a simple specification of an Abstract Data Type
5. 2. Describe alternating methods of implementing ADT
6. 3. Implement ADT using simple data types
7. 4. To describe in detail how basic operations are implemented on data structures.
8. 5. Describe a typical application where such a data structure may be found.
9. 6. Write code to implement a section of data structure.
10. 7. Review current estimates of the resources and time requirements for a/an data structure / algorithm.
8. Justify the selection of the most appropriate data structure / algorithm for a given application.

Table of contents

Chapter I	7
Abstract data types and other basic concepts	7
1.1 What is a Data Structure?	8
1.2 Why we need Data Structures?	8
1.3 Which Program is efficient?	8
1.4 Data Structure Selection:	9
1.5 Data Types:	9
1.6 Abstract Data Types – ADT:	9
1.6.1 Data Structures:	10
1.6.2 Classes & Objects:	10
1.6.3 Difference between Data Structure & Class:	11
1.7 Algorithms:	11
1.7.1 Characteristics of an Algorithm:	11
1.7.2 How to write algorithms?	12

1.8 Complexity of an Algorithm:	13
1.8.1 What effects run time of an algorithm?	13
1.8.2 Asymptotic Notations:	14
1.8.3 Analysis of Algorithms:	14
1.8.4 How to calculate time complexity of any algorithm or program?	14
Chapter II	16
Implementation of Arrays	16
2.1 Array:	17
2.2 Representation of Array in Memory:	17
2.3 Operations on Arrays:	18
2.3.1 TRAVERSING:	18
2.3.2 INSERTION:	19
2.3.3 DELETION:	20
2.4 Two Dimensional Arrays:	21
2.4.1 Traversing of 2d array:	22
Chapter III	23
Searching & Sorting	23
3.1 Searching:.....	24
3.1.1 Sequential / Linear Search:	24
3.1.2 Binary Search:	25
3.2 Sorting:	27
3.2.1 Exchange Sort:	27

3.2.2 Bubble Sort:	29
3.2.3 Selection Sort:	31
3.2.4 Insertion Sort:	33
Chapter IV	34
Stack	34
4.1 Introduction:	35
4.2 Use of Stack in Computers:	35
4.3 Representation of Stack in Memory:-	36
4.4 Stack Operations:	36
4.4.1 PUSH (Stack, Item, Top, N) [1]	36
4.4.2 POP (Stack, Top, Item) [1]	37
4.5 Implementation of Stack: Recursion	37
4.5.1 FACTORIAL (F, N)[1]	38
4.6 Evaluation of Arithmetic Expressions using STACK:	38
4.6.1 Expression Notations:	38
4.6.2 Use of Stack in evaluation of Expression:	39
Chapter V	44
Queues	44
5.1 INTRODUCTION:	45
5.2 Structure of a QUEUE:	46
5.3 INSERT_QUEUE (Q, N, REAR, FRONT, DATA) [1]	46

5.4 DELETE_QUEUE (Q, N, FRONT, REAR, DATA) [1]	47
5.5 Applications of Queue:	47
5.5.1 in Operating System:	47
5.6 Priority Queues:	49

Chapter VI 50

Link List 50

6.1 INTRODUCTION:	51
6.2 REPRESENTATION OF LINKED LIST IN MEMORY:	52
6.3 Operations of Linked List:	53
6.3.1 Insertion in the Link List:	53
6.3.2 Deletion from the Link List:	57
6.3.3 Advantages of Using Linked list:	59
6.3.4 Disadvantages of link list:	59

Chapter VII 59

Trees 59

7.1 INTRODUCTION:	60
7.2 Binary Tree:	61
7.3 Full Binary Tree:	62
7.4 Complete Binary Tree:	62
7.5 Advantages of trees:	63
7.6 Structure of a Node of Tree:	63
7.7 Binary Search Tree:	64

7.7.1 Insertion in a Binary Search Tree:	64
7.7.2 TREE-INSERT (Tree, node) [1]	64
7.7.3 Traversing in BST:	65
References	67

LIST OF FIGURES

FIGURE 1-1 CLASS REPRESENTATION	10
FIGURE 3-1 EXCHANGE SORT [GOOGLE.COM/IMAGES]	28
FIGURE 3-2 BUBBLE SORT [GOOGLE.COM/IMAGES].....	31
FIGURE 3-3 SELECTION SORT [GOOGLE.COM/IMAGES]	31
FIGURE 3-4 INSERTION SORT [GOOGLE.COM/IMAGES]	34
FIGURE 5.1: STATE DIAGRAM [GOOGLE.COM/IMAGES]	48
FIGURE 5.2: SCHEDULING QUEUES	48
FIGURE 5.3: PRIORITY QUEUES	49

FIGURE 7.1: GENERAL TREE	61
FIGURE 7.2: BINARY TREE	62
FIGURE 7.2: COMPLETE AND FULL BINARY TREE	63
FIGURE 7.3: BST	65

Chapter I

Abstract data types and other basic concepts

This chapter emphasizes on the basic idea of data structure and what is the significance of using data structures, different type of data structures, calculating the run time complexity of any implementation of different data structures.

Students will understand ADT and different ways of implementing ADT.

At the end of the chapter, students are expected to satisfactorily meet the following course objectives and learning outcomes:

OBJECTIVES	LEARNING OUTCOMES (LO)
<ol style="list-style-type: none"> 1. Specify and implement Abstract Data Type (ADT). 2. Justify the selection of an appropriate data structure/algorithm. 	<ol style="list-style-type: none"> 1. Produce a simple specification of an Abstract Data Type. 2. Describe alternating methods of implementing ADT. 3. Review current estimates of the resources and time requirements for a data structure algorithm.

1.1 What is a Data Structure?

Data Structure is a specific organization of data to be stored in memory, so that it can be used efficiently in our programs. Organization means that data should be arranged in a way, which should be easily accessible.

1.2 Why we need Data Structures?

An efficient program executes faster and helps minimize the usage of resources like memory and disk etc. Computers are getting more powerful with the increase in Processor speed, fast networks and large size storage disks. Therefore people want to have software for all type of applications from simple to most complex. Now if the application will be complex than there is need for more resources. Here resources all the times does not means that we always need a more powerful computer with more hardware resources. As an Engineer & Programmer, our try should be to achieve this target with the help of good and efficient programming, data structures and algorithm.

1.3 Which Program is efficient?

A Program is said to be efficient if it solves the problem within its resource constraints. The important resources

1. Time
2. Hard disk
3. Memory
4. Network Bandwidth
5. Other Computer hardware

Suppose that we have some program that solves the problem but takes 10 days, then it will be of no use. Similarly if the data to be stored in disk is too huge then again we can have the problem of resources. So we have to write programs considering the resources to achieve best possible solution as soon as possible.

There is always cost associated with resources like CPU and memory. But as long as data structures and programs are concerned, we have to spend time for this. Consumption of all these requirements will decide that the solution/program/software you have provided is best suitable or not. If its advantages are not obtained by the organization/company for which software was developed, then either software is not good or specifications of computer are not good.

1.4 Data Structure Selection:

We have multiple types of Data Structures available. Each data structure has costs and benefits. Any data structure used in your program will have some benefits. For this, you have to pay price. That can be computer resources or the time.

There are three basic things associated with data structures. A data structure requires:

1. Memory space for each data item it stores
2. Time to perform each basic operation
3. Programming effort

1.5 Data Types:

Data type is an identification construct used in programming languages to identify specific type of data which will be stored in computer memory. It also refers to the size of memory allocated for that specific data type.

Following are the most commonly used data types in C/C++ and lot many other 3GLs.

1. Character Size = 1 byte
2. Integer Size = 2 bytes
3. Float Size = 4 bytes
4. Double Size = 8 bytes

All these data types and many other are built-in in most of the languages for storing different types of data.

1.6 Abstract Data Types – ADT:

An ADT can be defined as an abstract model of implementation of certain data with the operations that are performed on that data.

ADTs are implemented in the form of two types:

1. Data Structures.
2. Classes & Objects.

1.6.1 Data Structures:

Following data structures we will implement in coming chapters.

1. Array
2. Stack
3. Linked Lists
4. Queue
5. Tree

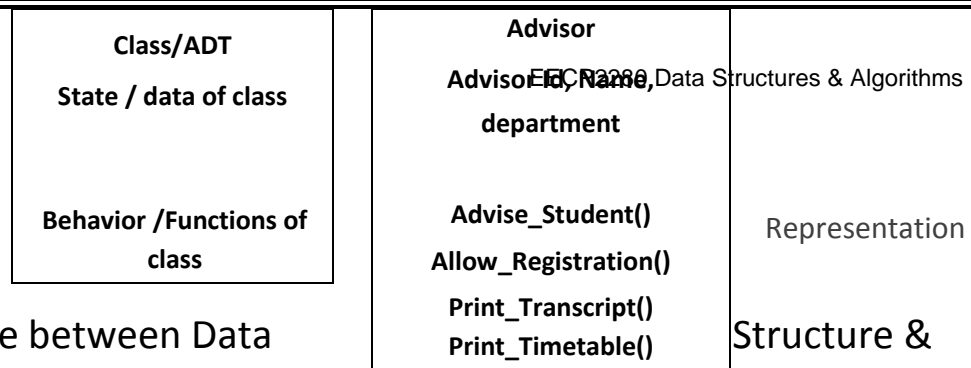
1.6.2 Classes & Objects:

An object is a composition of related state (Data) and behavior (Functionality). Software objects are often used to model the real-world objects that you find in everyday life.e.g. In software for Advising & Registration in our college are Student, Advisor, Degree, Course, Teacher, Timetable etc are Objects. For each real time object, Classes are made first as a model, and then, in software Objects are created whenever and how many are required.

So in Advising and Registration system of our college, classes will be the same as real time objects i.e. Student, Advisor, Degree, Course, Teacher and Timetable etc.(See Figure 1.1)

So a class is a model/prototype from which objects are created. A class defines the state and behavior of a real-world object.

Figure 1-1 Class



1.6.3 Difference between Data Class:

Data Structure is only the organization of related data while a class combines both related data as well as functions to be performed on that data. A class can compose a Data structure inside the Data part of it and in the behavior part can define the functionality to be performed on that data structure. This is possible due to the reason that Data structure did not have its own behavior inside.

1.7 Algorithms:

The word *algorithm* comes from the name of the 9th-century Persian mathematician Abu Abdullah Muhammad bin Musa al-Khwarizmi. Algorithm simply is a method/procedure/technique to perform a specific functionality/task. It consists of a set of instructions or steps to complete the required task.

Algorithms are always written as a design artifact for developing software. It has the same purpose or importance as a Map/architecture of a house has for building it, a circuit diagram/schematic diagram has for making an electronic product etc. Algorithms are used as a design of computer programs/software and once algorithms are written/prepared, then it means design of software is ready, which will be translated only into computer programs using a programming language e.g. C/C++/Java etc.

So an algorithm is the model and the program is its concrete shape to run in computer. Algorithms are more important than Programs itself as these work as design and if there will be an error in the design of an algorithm, it will lead to errors for sure in program. For solving a single problem in computer, there can be multiple algorithms and each will be successful in completing the job with some benefits/drawbacks.

1.7.1 Characteristics of an Algorithm:

Algorithm must have the following characteristics.

1. An algorithm must be **P**recise.
2. An Algorithm must be **E**ffective.

3. An algorithm must terminate.

In short an algorithm must be PET, Precise, Effective and Terminate. Note that while the three characteristics just given are necessary for an algorithm but are not sufficient. An incorrect algorithm might possess all three characteristics, yet still contain incorrect instruction that prevent it from accomplishing the task for which it was designed.

1.7.2 How to write algorithms?

There are two common ways of writing algorithms:

1. Pseudo code

2. Flow Charts

1.7.2.1 *Pseudo code:*

Pseudo code is just like a natural language sentences written in the form of steps/instructions to perform some specific activity/task. Pseudo code doesn't have any standard way of writing. It is an English like representation of code required for an algorithm.

Layout of an algorithm written in pseudo code may have the following form.

ALGORITHM Algorithm_name

Statements of Algorithm

END

Example No.1:

ALGORITHM PURCHASE

1. INPUT Item_Price, Discount_Rate
2. SET Discount = Item_Price * Discount_Rate
3. SET Item_Price = Item_Price –Discount
4. OUTPUT Item_Price

END

Example No.2:

ALGORITHM CALCULATE_TAX

1. INPUT Item_Price
2. INPUT Sales_tax_rate
3. SET Sales_tax = Item_Price * Sales_tax_rate
4. SET Final_Price = Item_Price + Sales_tax
5. OUTPUT Final_Price

END

Example No.3:

ALGORITHM CALCULATE_PAY

1. INPUT Days_worked
 2. INPUT Working_days
 3. INPUT Daily_pay_rate
 4. if Days_worked \leq Working_days then
 - 4.1 SET Gross_pay = Days_worked * Daily_pay_rate
 5. IF Days_Worked > Working_days then
 - 5.1 SET Gross_pay = Daily_pay_rate * Days_worked + 1.5 * (Days_worked - Working_days)
 6. OUTPUT Gross_pay
- END

1.7.2.2 Flowcharts:

Same what you studied in Computer programming using C.

1.8 Complexity of an Algorithm:

Algorithmic complexity is concerned about how fast or slow particular algorithm performs. We define complexity as a numerical function $T(n)$ - time versus the input size n .

Complexity of an algorithm is a measure of the amount of time and/or space required by an algorithm for an input of a given size (n).

1.8.1 What effects run time of an algorithm?

- (a) Computer used, the hardware platform
- (b) Representation of abstract data types
- (c) Efficiency of compiler
- (d) Competence of implementer
- (e) Complexity of underlying algorithm
- (f) Size of the input

1.8.2 Asymptotic Notations:

The goal of computational complexity is to classify algorithms according to their performances. We will represent the time function $T(n)$ using the "big-O" notation to express an algorithm runtime complexity. For example, the following statement

$T(n) = O(n^2)$, says that an algorithm has a quadratic time complexity.

1.8.3 Analysis of Algorithms:

The term analysis of algorithms is used to describe approaches to the study of the performance of algorithms

The worst-case runtime complexity of the algorithm is the function defined by the maximum number of steps taken on any instance of size a .

The best-case runtime complexity of the algorithm is the function defined by the minimum number of steps taken on any instance of size a .

The average case runtime complexity of the algorithm is the function defined by an average number of steps taken on any instance of size a .

1.8.4 How to calculate time complexity of any algorithm or program?

The most common metric for calculating time complexity is Big O notation. This removes all constant factors so that the running time can be estimated in relation to N as N approaches infinity. In general you can think of it like this:

```
1.      statement;
```

Is constant. The running time of the statement will not change in relation to N .

```
1.      for (i= 0; i< N; i++ )
2.      statement;
```

Is linear. The running time of the loop is directly proportional to N . When N doubles, so does the running time.

```
1.      for (i= 0; i< N; i++ ){
2.      for ( j = 0; j < N; j++)
3.      statement;
4.      }
```

Is quadratic. The running time of the two loops is proportional to the square of N . When N doubles, the running time increases by $N * N$.

```
5.      while ( low <= high ){
6.      mid =( low + high ) / 2;
7.
```

```
8.         if ( target < list[mid])
9.             high = mid - 1;
10.        else if ( target > list[mid])
11.            low = mid + 1;
12.        else break;
13.    }
```

Is logarithmic. The running time of the algorithm is proportional to the number of times N can be divided by 2. This is because the algorithm divides the working area in half with each iteration.

Chapter II

Implementation of Arrays

This chapter emphasizes on the array data structures, different types of arrays, their implementation and representation in memory.

At the end of the chapter, this course should enable the students to:

OBJECTIVES	LEARNING OUTCOMES (LO)
------------	------------------------

<ol style="list-style-type: none">1. Evaluate and implement data structure2. Justify the selection of an appropriate data structure/algorithm.	<ol style="list-style-type: none">1. Produce a simple specification of an Abstract Data Type.2. Implement ADT using simple data types.3. Write code to implement a section of data structure.
---	---

2.1 Array:

Array is the most basic and simplest type of Data Structure; used to store a collection of data items all of the same type. Array is given a name and each element of the array can be accessed using its name and its index number or position number in the array.

Array_name [index number of element]

The **index numbers** are started from **0** and goes up to **SIZE – 1**.

Starting / First index of the array is called Lower Bound (**LB**).

Ending / Last index of the array is called Upper Bound (**UB**).

Length of the array = $UB - LB + 1$

2.2 Representation of Array in Memory:

Array is allocated the contiguous memory locations i.e. in a sequence. The address of the first element of the array is called Base Address. Address of the any element of one-d array can be calculated using the formula.

$$LOC (X[K]) = Base(X) + W(K-LB)$$

Base is the starting address of array X in memory.

W is the weight or size of 1 element of array, W depends on the data type of Array.

Character - $W = 1$

Integer - $W = 2$

Float - $W = 4$

Double - $W = 8$

Array X – One dimensional integer Array

Index	X[0]	X[1]	X[2]	X[3]	X[4]
Memory Location	2000	2002	2004	2006	2008
Data	2	5	8	9	11

Example:

$$\text{Loc}(X[3]) = 2000 + 2(3-0)$$

$$= 2003 + 3 = 2006$$

2.3 Operations on Arrays:

There are three general operations performed on Arrays.

1. Traversing

2. Insertion

3. Deletion

2.3.1 TRAVERSING:

Traversing is simply accessing or visiting the elements of an array.

Algorithm:

(Traversing an array) TRAVERSE (X, LB, UB)[1]

X is an array, UB is the upper bound and LB is the lower Bound.

1. [Initialize counter] Set K = LB
2. Repeat steps While K <= UB
 - 2.1 [Visit Element] Output X [K]
 - 2.2 [Increase Counter] Set K = K + 1
- [End of step-2 loop]

3. End

2.3.2 INSERTION:

Insertion can be done in two ways in an array.

- Insertion at End

- Insertion at a specific location

2.3.2.1 Insertion at End:

Algorithm:

INSERT (X, UB, N, ITEM) [1]

X is an array, UB is upper bound, N is the total number of elements and ITEM is the element to Insert.

1. [Insert element] Set $X[UB + 1] = \text{ITEM}$
2. [Reset N] Set $N = N + 1$
3. End

2.3.2.2 Insertion at a specific location:

Algorithm:

INSERT (X, UB,N, K, ITEM)[1]

X is an array, UB is upper bound, number of elements in array are N and ITEM is the element to Insert, K is the position/Index to Insert.

1. [Initialize Counter] Set $J = UB$
2. Repeat While $J \geq K$
 - 2.1 [Move Jth element downward] Set $X[J + 1] = X[J]$
 - 2.2 [Decrease Counter] Set $J = J - 1$

[End of Step-2 loop]
3. [Insert element] Set $X[K] = \text{ITEM}$
4. [Reset N] Set $N = N + 1$
5. End

2.3.3 DELETION:

Deletion can be done in two ways in an array.

- Deletion at End
- Deletion from a specific location

2.3.3.1 *Deletion at End:*

Algorithm:

DELETE (X, UB, N, ITEM) [1]

X is an array, number of elements in array are N and ITEM is to hold the deleted element, and UB Upper bound

1. [Insert element] Set $ITEM = X[UB]$
2. [Reset N] Set $N = N - 1$
3. $UB = UB - 1$
4. End

2.3.3.2 *Deletion at a specific location:*

DELETE (X, N, UB, K, ITEM) [1]

X is an array, N is total elements, ITEM is to hold the deleted element, K is the position/Index to Delete and UB is upper bound.

1. Set $ITEM = X[K]$
2. Set $J = K$.
3. Repeat while $J < UB$
 - 3.1 [Move J + 1 element upward] Set $X[J] = X[J + 1]$
 - 3.2 $J = J + 1$

[End of loop]

4. [Reset N] Set $N = N - 1$
5. UB=UB-1
6. End

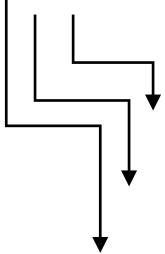
2.4 Two Dimensional Arrays:

One dimensional array is storing just a list of data. But a 2 dimensional array has the capacity to store a table with rows and columns. For accessing a 2 dimensional array we need two Index numbers. One is to specify Row and other is for Column.

	Col 0	Col 1	Col 2	Col 3	Col 4
Row 0	X[0][0]	X[0][1]	X[0][2]	X[0][3]	X[0][4]
Row 1	X[1][0]	X[1][1]	X[1][2]	X[1][3]	X[1][4]
Row 2	X[2][0]	X[2][1]	X[2][2]	X[2][3]	X[2][4]
Row 3	X[3][0]	X[3][1]	X[3][2]	X[3][3]	X[3][4]

Row Index

But as we know that stored in the memory



Array Name

Memory is Linear, so a 2-dimensional array will be either in row major or column major.

The difference between the orders lies in which elements of an array are contiguous in memory. In a rowmajor order, the consecutive elements of a row reside next to each other, whereas the same holds true for consecutive elements of a column in a column-major order.

Let's suppose array is of Integer with 3 rows and 2 columns stored in row major representation: `int Marks[3][2],`

Following formula will be used for calculating the address of any location.

$$\text{ADDR}(r, c) = B + W * (Nc * r + c)$$

B = Base address of array.

W = Weight, means size of an element. It is defined with respect to Data type of array.

Nc = number of columns in a row. r

= Row Index, c = Column Index

Example: For above shown array, find address for `X[2][0]`

$$\text{ADDR}(r, c) = B + W * (Nc * r + c)$$

$$\text{Address}(X[2][0]) = 2000 + 2 * (2 * 2 + 0)$$

$$= 2000 + 2 * (4)$$

$$= 2000 + 8$$

$$= 2008$$

2.4.1 Traversing of 2d array:

(Traversing a 2D array) TRAVERSE_2D (X, LB_R, LB_C, UB_R, UB_C) [1]

X is a 2D array, UB_R is Upper Row bound, UB_C is upper Column bound, LB_R is lower Row Bound, LB_C is the lower Column Bound.

1. [Initialize Row counter] Set R = LB_R
2. [Initialize Column counter] Set C = LB_C
3. Repeat While R <= UB_R
 - 3.1 Repeat While C <= UB_C
 - 3.1.1 [Visit Element] Output X [R][C]
 - 3.1.2 [Increase Column Counter] Set C = C + 1
 - [END OF LOOP]
- 3.2 [Increase Row Counter] Set R = R+1, C=LB_C
- [End of step-3 loop]
4. End

Chapter III

Searching & Sorting

This chapter emphasizes on the different algorithms for searching and sorting like linear search, binary search and insertion sort, bubble sort, selection sort, exchange sort and their implementation.

At the end of the chapter, this course should enable the students to:

OBJECTIVES	LEARNING OUTCOMES (LO)

<ol style="list-style-type: none">1. Evaluate and implement data structure2. Justify the selection of an appropriate data structure/algorithm.	<ol style="list-style-type: none">1. Implement ADT using simple data types.2. Describe a typical application where such a data structure may be found.
---	---

3.1 Searching:

One of the most common and most time consuming operations in Computer science is searching. For accessing a specific data out of a large amount of data is a critical operation because as fast it will be searched out, as fast the operation can be completed. Searching is a critical operation on which all other operations like deletion, insertion, manipulation etc are dependent.

There are two common techniques of searching used.

1. Linear Search / Sequential Search
2. Binary Search

3.1.1 Sequential / Linear Search:

In this technique of searching data is searched sequentially i.e. each element is checked from the start till data is found or data finished. The sequential search is used whenever the list is not ordered. Generally, the method is used for smaller lists or lists that are not searched for often. In the sequential search, we usually start from the beginning and check each element until we find the target or we are sure that the element is not in the list.

3.1.1.1 LINEAR_SEARCH (DATA, N, ITEM, LOC) [1]

[Data is linear array, N is the total no. of elements; ITEM is the value to be found & LOC is for storing Location if found]

1. [Initialize counter] Set LOC = 0
 [Search for ITEM]
2. Repeat while DATA [LOC] \neq ITEM and LOC < N
 2.1 Set LOC = LOC + 1
 [End of Loop]
3. if LOC = N then [Check if Successful ?]
 3.1 Set LOC = 0
 3.2 Print "Element not in list"
 Else
 3.3 Print "Element present and Location is " LOC.
4. End

3.1.2 Binary Search:

It is a more efficient technique to search a specific item from list of items. It is mostly used for relatively large lists or table of records that are sorted in ascending or descending order. The Binary search algorithm is only operable on the sorted data.

This algorithm begins by searching the required value from the middle of the list. If the required value is in the middle of the list then search process terminates at that point. If the list is sorted in ascending order and the required value is greater than the value at the middle then search will be confined to the right side of middle otherwise if required value is less than middle then search will be confined to the left. This process will continue till the required value is found or data finishes. The efficiency of Binary Search is very high as compared to linear search as number of comparisons are very less as compared to the Sequential search.

Example:

Number to search (KEY) = 212 in given array X.

1	2	3	4	5
23	34	108	212	347

Iteration-1:

LB = 1, UB = 5, MID = $(LB+UB) / 2 = 1 + 5 / 2 = 3$

MID

Check if KEY = X [MID] (NO)

Then check

KEY > X [MID] (YES)

Then

Move your search to the Right side. I.e.

1	2	3	4	5
23	34	108	212	347

Iteration-2:

MID

$$LB = MID + 1 = 4$$

$$UB = N = 5$$

$$MID = 4 + 5 / 2 = 4$$

1	2	3	4	5
23	34	108	212	347

3.1.2.1 Binary Search (DATA, LB, UB, ITEM, LOC) [1]

[DATA is a sorted array of data, LB is lower bound of data, UB is the upper Bound, ITEM is the key to search,

LOC is the Location where ITEM is found.]

[Initialize Segment variables i.e. BEG, END, MID]

1. Set BEG:= LB
2. Set END:= UB
3. Set MID:= Integer ((BEG + END)/2)
4. Repeat while BEG <= END and DATA[MID] != ITEM
 - 4.1 If ITEM < DATA[MID] then
 - 4.1.1 Set END:= MID – 1
 - 4.2 ELSE
 - 4.2.1 Set BEG:= MID + 1

[End of If Structure]
- 4.3 Set: MID = Integer ((BEG + END)/2)

[End of Step 2 loop]
5. If DATA[MID] = ITEM then

- 5.1 Set LOC:= MID
- 6. Else
- 6.1 Set LOC:= NULL
 [End Of If Structure]
- 7. End

3.2 Sorting:

The Process of arranging data in a specified order (Ascending, Descending) is called sorting. There are several methods of sorting data.

1. Exchange Sort
2. Bubble sort
3. Insertion sort
4. Selection sort

3.2.1 Exchange Sort:

It is the simplest and slowest algorithm of sorting.

3.2.1.1 Exchange Sort (DATA, N) [1]

[DATA is the data to be sorted, N is the number of data elements in the array]

1. [Initialize counter] Set count_E = 1
2. Repeat While count_E < N

[Initialize the Segment variable counter for internal loop]

- 2.1 Set count_I := 1
- 2.2 Repeat while count_I < N
 - [Comparing & if upper value > lower ,Swap both]
 - 2.2.1 If DATA[count_I] > DATA[count_I + 1]

2.2.1.1 Set $HOLDER := DATA[count_I]$

2.2.1.2 Set $DATA[count_I] = DATA[count_I + 1]$

2.2.1.3 Set $DATA[Count_I + 1] = HOLDER$

[End of If]

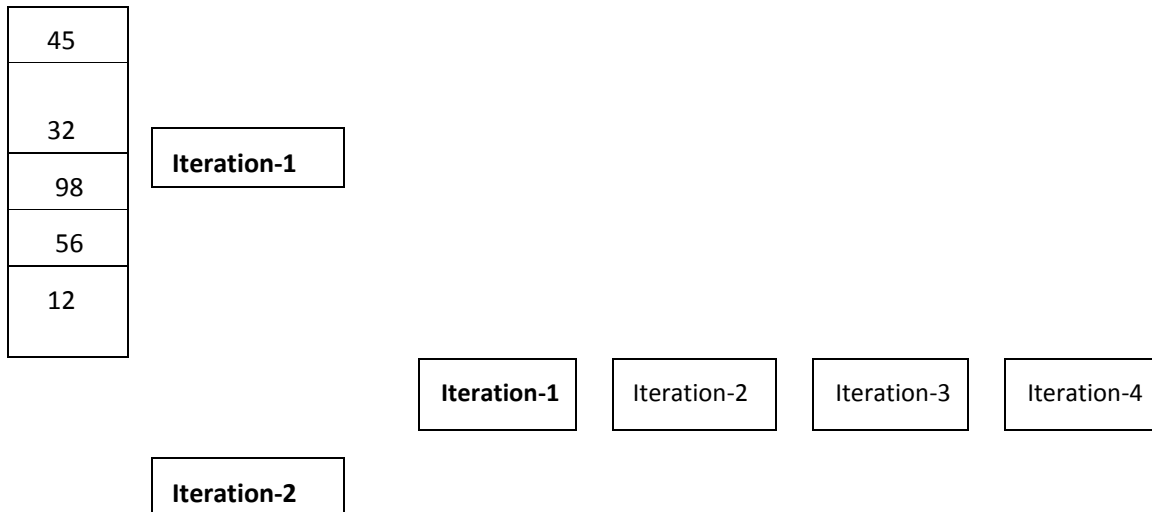
2.2.2 Set $count_I := count_I + 1$

[End of Internal loop]

2.3 Set $count_E := count_E + 1$

[End of External loop]

3. End



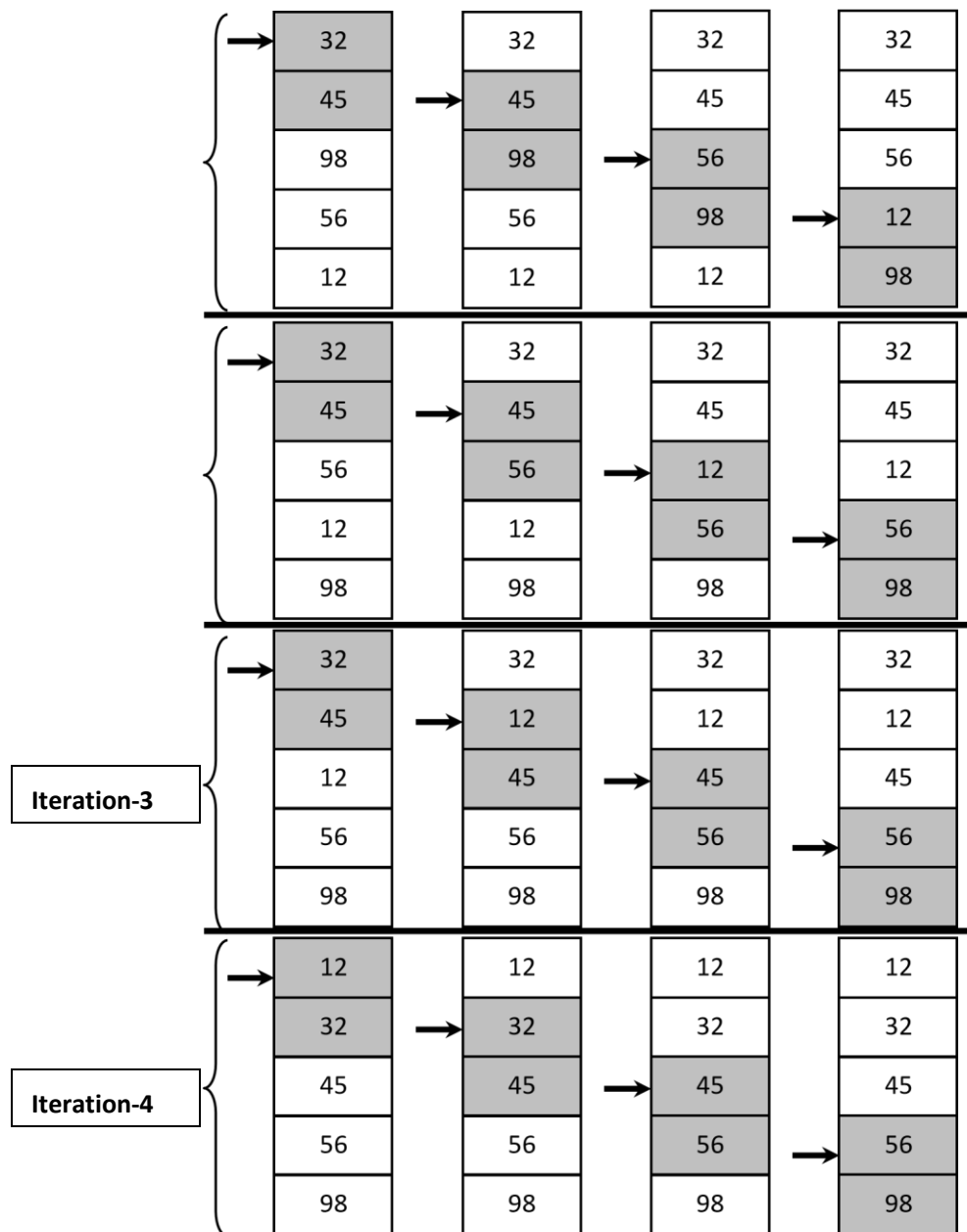


Figure 3-1 Exchange Sort [Google.com/images]

3.2.2 Bubble Sort:

It is also a very commonly used algorithm for sorting. It is slow like the exchange sort and commonly used with small amount of data. But a bit fast than the Exchange sort.

3.2.2.1 *Bubble_Sort(DATA, N) [1]*

[DATA is the data to be sorted, N is the number of data elements in the array]

1. [Initialize counter] Set count_E = 1

2. Repeat While count_E < N

[Initialize the Segment variable counter for internal loop]

2.1 Set count_I := 1

2.2 Repeat while count_I ≤ N – count_E

[Comparing consecutive values and if upper > lower, Swap]

2.2.1 If DATA[count_I] > DATA[count_I + 1]

2.2.1.1 Set HOLDER := DATA[count_I]

2.2.1.2 Set DATA[count_I] = DATA[count_I + 1]

2.2.1.3 Set DATA[Count_I + 1] = HOLDER

[End of If]

2.2.2 Set count_I := count_I + 1

[End of Internal loop]

2.3 Set count_E := count_E + 1

[End of External loop]

3. End

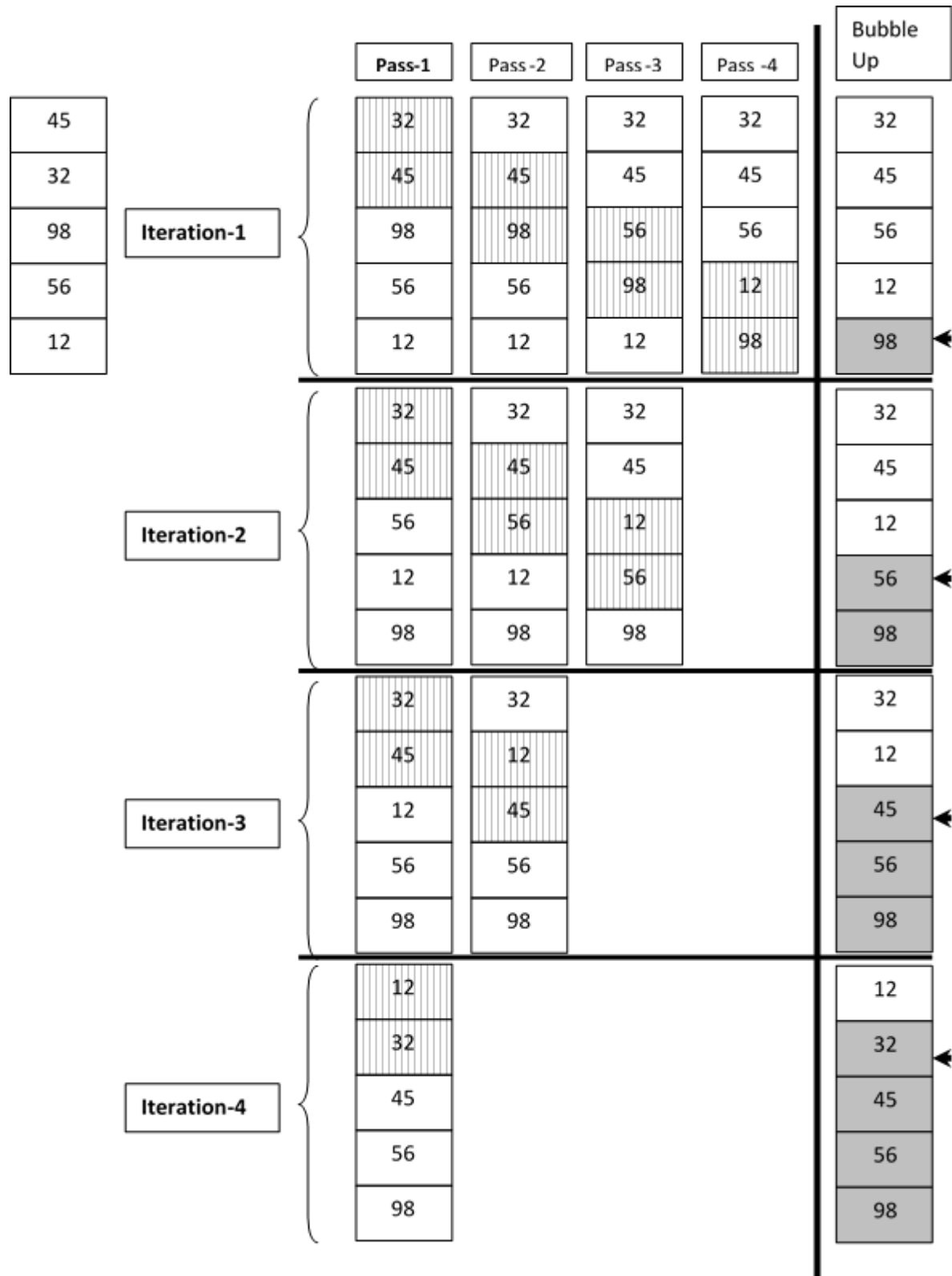


Figure 3-2 Bubble Sort [Google.com/images]

3.2.3 Selection Sort:

Selection sort works by selecting the smallest unsorted item remaining in the list and then swapping it with the item in the next position to be filled, i.e. the smallest is find out and swapped with value at index 1, next smallest out of unsorted data will be detected and swapped with value at index 2 and so on until all the items are sorted. N-1 iterations are required to sort an array of N items.

Example:

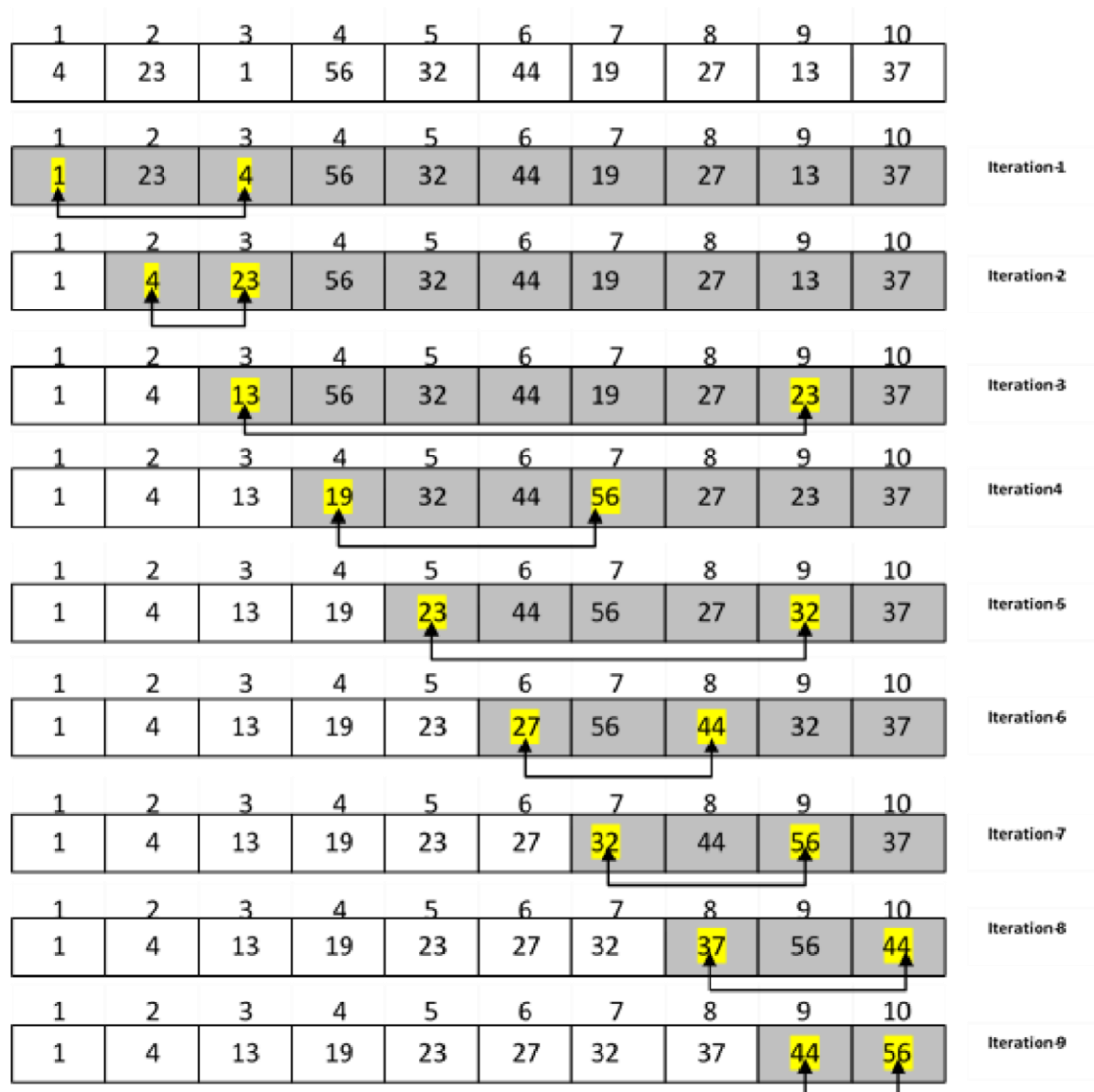


Figure 3-3 Selection Sort [Google.com/images]

3.2.3.1 *Selection_sort(Data, N)[1]*

[Data is the list of data, N is the number of elements in the Data]

1. [Initialize the counter] Set COUNT = 1

2. Repeat While COUNT < N

2.1 Set Temp = Data [COUNT] 2.2

Set INDEX = COUNT

2.3 Repeat While INDEX <= N

2.3.1 IF Temp > DATA [INDEX] Then

2.3.1.1 Set Temp = Data [INDEX]

2.3.1.2 Set LOC = INDEX

2.3.2 Set INDEX = INDEX + 1

[End of internal loop]

[Swap / Interchange values]

2.4 Set Temp = DATA[LOC]

2.5 Set DATA[LOC] = DATA [COUNT]

2.6 Set DATA[COUNT] = Temp

2.7 Set COUNT = COUNT+1

[End of step-2 loop]

3. End

3.2.4 Insertion Sort:

Insertion sort works by inserting each item in the list into its proper position. It is implemented using Two separate lists, one is the source list and the other one the Final sorted list. But this cause extra memory consumption so In-place sort is done i.e. the sorting is done in the same source array.

3.2.4.1 INSERTION_SORT (Data, N) [1]

[Data is the data to be sorted; N is the total number of elements in the Data]

1. Repeats for K = 1 to N by 1

1.1 Set TEMP = Data [K]

1.2 Set Count = K

1.3 Repeat steps WHILE (Count > 0 AND TEMP < Data [Count - 1])

1.3.1 Set Data [Count] = Data [Count-1]

1.3.2 Set Count = Count – 1

[End of Internal Repeat WHILE]

1.4 Set Data[Count] = Temp

[End of external Repeat FOR]

2. End

A[1]	A[2]	A[3]	A[4]	A[5]
16	17	2	8	18
A[1]	A[2]	A[3]	A[4]	A[5]
16	17	2	8	18
A[1]	A[2]	A[3]	A[4]	A[5]
16	17	2	8	18
A[1]	A[2]	A[3]	A[4]	A[5]
16	17	2	8	18
A[1]	A[2]	A[3]	A[4]	A[5]
2	16	17	8	18
A[1]	A[2]	A[3]	A[4]	A[5]
2	8	16	17	18
A[1]	A[2]	A[3]	A[4]	A[5]
2	8	16	17	18

Figure 3-4 Insertion Sort [Google.com/images]

Chapter IV

Stack

This chapter emphasizes on the different algorithms for Stack data structures and applications of Stack in Computer Science.

At the end of the chapter, this course should enable the students to:

OBJECTIVES	LEARNING OUTCOMES (LO)
------------	------------------------

<ol style="list-style-type: none">1. Evaluate and implement data structure2. Justify the selection of an appropriate data structure/algorithm.	<ol style="list-style-type: none">1. Describe alternating methods of implementing ADT.2. Implement ADT using simple data types.3. To describe in detail how basic operations are implemented on data structures.4. Describe a typical application where such a data structure may be found.5. Describe a typical application where such a data structure may be found.
	<ol style="list-style-type: none">6. Justify the selection of the most appropriate data structure / algorithm for a given application.

4.1 Introduction:

Stack is a special kind of linear list in which only two operations, insertion & Deletion can be performed & in a

Stack these operations may occur only at its Top end. The items in it are stored & retrieved in Last in First out (LIFO) manner.

Definition: A stack is a linear list in which all additions and deletions are restricted to one end called the Top. If you were to insert a data series to a stack and then remove them, their order would be reversed. The reversing attribute has led to stacks being known as Last In First Out data structure.

The example of Stack is the Currency/Cash put by a Cashier in a Cash machine.

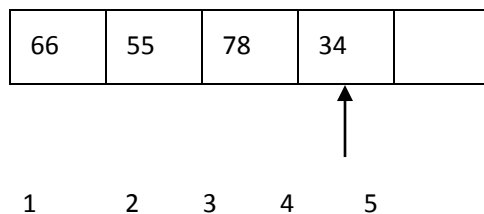
- Cash is inserted in Cash box/machine from Top.
- Which note is first taken out? Top one, means the currency note which is entered at last will be taken out first, so it is called LIFO. So Insertion and Deletion both is done from Top.

4.2 Use of Stack in Computers:

1. In Programming we used it in Recursion and in Evaluation of mathematical Instructions.
2. Memory Management in Operating Systems.
3. Protocol Stack used in Network communication, e.g. TCP/IP
4. In Microprocessors, there is a specific register named stack, used for program execution.
5. Some Microprocessors have a Stack based Instruction Set Architecture, in which always operands are pushed into Stack and then Popped for performing operations.
6. Page visited history of a browser.
7. Undo sequence of a normal editor.

4.3 Representation of Stack in Memory:-

Stack is mostly represented as a linear array in memory, so for implementation of Stack Arrays are used as a base structure.



In Stack Insertion and deletion is only done at the TOP of Stack.

4.4 Stack Operations:

1. PUSH is the term used to insert an element in to a Stack.

2. POP is the term used to delete an element from a Stack.

Before Pushing an Item into A Stack, it is tested whether or not there is any room in Stack to store the new item. If there is no room in the Stack, than the Stack condition is known as OVERFLOW.

Before removing or deleting an item from Stack it is tested whether or not at least one element exists in the Stack, if no element exists in Stack i.e. the Stack is empty, the condition is known as UNDERFLOW.

4.4.1 PUSH (Stack, Item, Top, N) [1]

[Stack is the stack, Item is the item to be inserted, TOP is the top of the stack, N is the size of the stack]

1. [Check for Overflow condition] IF Top \geq N THEN

1.1 OUTPUT "Stack Overflow"

[End of IF]

2. ELSE

2.1 [Increment 1 into TOP] SET Top = Top + 1

2.2 [Add Element Item into Stack] SET Stack[Top] = Item

[End of ELSE]

3. END

4.4.2 POP (Stack, Top, Item) [1]

[Stack is the Stack, Top is top of stack, Item is the item which is deleted]

1. [Check for under flow condition] IF Top = 0 THEN

1.1 OUTPUT "Stack Underflow"

[End of IF]

2. ELSE

2.1 [Assign top of Stack value to Item to store deleted value]

SET Item = Stack [Top]

2.2 [Decrement Top of Stack to delete item] SET Top = Top – 1

[End of ELSE]

3. END

4.5 Implementation of Stack: Recursion

Recursion is a fundamental concept in Programming. Every problem, which is solvable iteratively, may also be done through Recursion. A Procedure that calls itself inside its definition is called Recursive Procedure and the process of calling the procedure is called the Recursion. It is easier to define a Recursive function. Only the procedure, which we want to make recursive, should have two properties.

1. There must be criteria for exiting from the Recursive Procedure. i.e. Base Criteria.
2. Each time the procedure is called, the condition must get closer to the Base criteria.

Let us consider an example first to understand the Recursion.

Factorial to a number:

We know that the Factorial of a number N can be written like this:

$$N! = N * (N-1)!$$

You can visualize the recursive call/execution of function as under:

factorial(5) = 5 * factorial(4)

= 5 * (4 * factorial(3)) PUSHs

= 5 * (4 * (3 * factorial(2)))

= 5 * (4 * (3 * (2 * factorial(1))))

= 5 * (4 * (3 * (2 * (1 * factorial(0)))))

= 5 * (4 * (3 * (2 * (1 * 1)))) POPs

= 5 * (4 * (3 * (2 * 1)))

= 5 * (4 * (3 * 2))

= 5 * (4 * 6)

$$= 5 * 24$$

$$= 120$$

4.5.1 FACTORIAL (F, N)[1]

[F is for Storing result i.e. Factorial value, N is the number whose Factorial is to find]

1. IF N = 0 THEN

1.1 SET F := 1

[End of If]

2. CALL FACTORIAL (F, N-1)

3. SET F := N * F

4. END

4.6 Evaluation of Arithmetic Expressions using STACK:

An Expression is made up of Operands, Arithmetic Operators and Parenthesis. Operands can be Variable / constant.

Expressions are evaluated from Left to Right. Following Precedence rules are followed:

1. If expression has Parenthesis, then they are evaluated first.
2. Exponential (^) is given highest priority.
3. Multiplication (*) and Division (/) have next Highest priority.
4. Addition (+) and subtraction (-) has lowest priority.

4.6.1 Expression Notations:

INFIX:

In this notation Operators are placed between the two operands.

POSTFIX:

In this notation the Operator is placed after the Operands.

INFIX

POSTFIX

$A + B$ $AB +$ X / Y $XY /$ $(X + Y) * Z$ $XY + Z *$ $(A + B) * (C - D)$ $A B + C D - *$ $A \wedge B * C - D + E / F$ $A B \wedge C * D - E F / +$

4.6.2 Use of Stack in evaluation of Expression:

The computer evaluates an arithmetic expression given in INFIX notation by converting it into the POSTFIX notation. The conversion of expression from INFIX to POSTFIX & then evaluation of a POSTFIX expression both involves the use of Stack.

4.6.2.1 Conversion from INFIX to POSTFIX:

First we will do some working examples and understand the method of conversion of an instruction from Infix to Postfix:

Example No. 1:

Expression = $A + (B * C)$

Steps	Symbol Scanned	Stack	Output
1	A	Empty	A
2	+	+	A
3	(+ (A
4	B	+ (A B
5	*	+ (*	A B
6	C	+ (*	A B C
7)	+	A B C *

8	End	Empty	A B C * +
---	-----	-------	-----------

Example No. 2:

Expression = $A + B * C + (D * E + F) * G$

Steps	Symbol Scanned	Stack	Output
1	A	Empty	A
2	+	+	A
3	B	+	A B
4	*	+ *	A B
5	C	+ *	A B C
6	+	+	A B C * +
7	(+ (A B C * +
8	D	+ (A B C * + D
9	*	+ (*	A B C * + D
10	E	+ (*	A B C * + D E
11	+	+ (+	A B C * + D E *
12	F	+ (+	A B C * + D E * F
13)	+	A B C * + D E * F +
14	*	+ *	A B C * + D E * F +
15	G	+ *	A B C * + D E * F + G
16	End	Empty	A B C * + D E * F + G * +

Example No. 3:

Expression = $A + B * C - D / E * F$

Steps	Symbol Scanned	Stack	Output
-------	----------------	-------	--------

1	A	Empty	A
2	+	+	A
3	B	+	A B
4	*	+ *	A B
5	C	+ *	A B C
6	-	-	A B C * +
7	D	-	A B C * + D
8	/	- /	A B C * + D
9	E	- /	A B C * + DE
10	*	- *	A B C * + DE/
11	F		A B C * + DE/F*-

Example No. 4:

Expression = $(A + B * C - D) / (E * F)$

Steps	Symbol Scanned	Stack	Output
1	((
2	A	(A
3	+	(+	A
4	B	(+	AB
5	*	(+ *	AB

6	C	(+ *	ABC
7	-	(-	ABC*+
8	D	(-	ABC*+D
9)		ABC*+D-
10	/	/	ABC*+D-
11	(/ (ABC*+D-
12	E	/ (ABC*+D-E
13	*	/ (*	ABC*+D-E
14	F	/ (*	ABC*+D-EF
15)		ABC*+D-EF*/

4.6.2.1 POSTFIX (Q , P) [1]

[Q is the expression in Infix and P is the expression in Postfix] 1.

REPEAT WHILE there is element available in Q

- 1.1 Scan an element from Q (from Left to Right)
- 1.2 IF an Operand is encountered add it to P
- 1.3 IF a left Parenthesis ('(') is encountered Then PUSH it into Stack
- 1.4 IF an Operator is encountered Then
 - 1.4.1 IF Operator there in stack on the TOP is of higher Precedence or equal precedence than POP it and add to P, repeat until the condition satisfies.
 - 1.4.2 PUSH Operator to Stack

[End of If structure]
- 1.5 IF a right parenthesis (')') is encountered, then
 - 1.5.1 POP all the operators before the left parenthesis and add to P in the order of POP.
 - 1.5.2 POP the left Parenthesis and discard it.

[End of If structure]

[End of loop]

2. END

4.6.2.2 Evaluation of Expression:

Example No. 1:

Expression: **1 2 3 * + 4 -**

Symbol	Operand 1	Operand 2	Value	Stack
1				1
2				1, 2
3				1, 2, 3
*	2	3	6	1, 6
+	1	6	7	7
4				7, 4
-	7	4	3	3

Example No. 2:

Expression = **6 2 3 + - 3 8 2 / + * 2 ^ 3 +**

Symbol	Operand 1	Operand 2	Value	Stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8

2	6		1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2	1	7	7	7,2
^	7	2	49	49
3	7	2	49	49,3
+	49	3	52	52

4.6.2.3 EVALUATE_POSTFIX (P , RESULT) [1]

[P is the expression in Postfix notation and Result is the variable to hold the result]

1. REPEAT WHILE there is an element in P
 - 1.1 SCAN an element from P (from Left to Right)
 - 1.2 IF an Operand is encountered then
 - 1.2.1 PUSH into Stack

[End of If structure]
 - 1.3 IF an Operator is encountered then
 - 1.3.1 POP two TOP elements from Stack.
 - 1.3.2 Perform the operation based on Operator and store in RESULT.
 - 1.3.3 PUSH RESULT into Stack

[End of If structure]

[End of step 1 loop]
2. POP stack and assign it to RESULT.
3. END

Chapter V

Queues

This chapter emphasizes on the different algorithms for Queue data structures and applications of Queue in Computer Science.

At the end of the chapter, this course should enable the students to:

OBJECTIVES	LEARNING OUTCOMES (LO)
3. Evaluate and implement data structure 4. Justify the selection of an appropriate data structure/algorithm.	7. Describe alternating methods of implementing ADT. 8. Implement ADT using simple data types. 9. To describe in detail how basic operations are implemented on data structures. 10. Describe a typical application where such a data structure may be found. 11. Describe a typical application where such a data structure may be found.
	12. Justify the selection of the most appropriate data structure / algorithm for a given application.

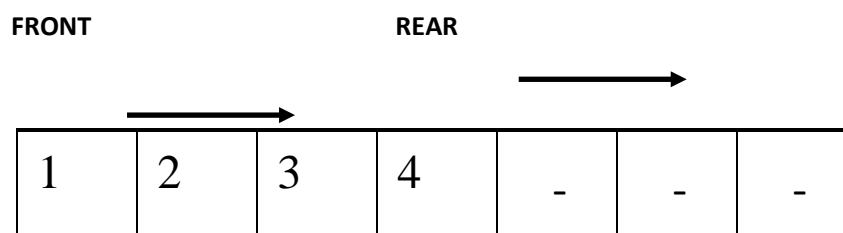
5.1 INTRODUCTION:

A Queue is a Linear Data structure and its realization is exactly similar to real world Queues. Stack access mechanism is LIFO, but Queues are always accessed in FIFO (First In First Out) manner. Definition: A queue is a linear list in which data can only be inserted at one end, called the rear, and deleted from the other end, called as front. These restrictions ensure that the data are processed through the queue in the order in which they are received.

Queues have immense applications in Computer Industry. Major applications are:

1. It has so many applications in Operating Systems for CPU scheduling, Process Scheduling, I/O scheduling, Interrupt handling etc.
2. Priority queues in Operating Systems
3. In a Network when many computers will be sharing few printers then Print jobs will be queued in Printer buffers.
4. Buffers on MP3 players and portable CD players, iPod playlist etc.
5. Any data which need to be processed in order in a device to device communication or a Network communication is maintained in a Queue.
6. Software for generating Tokens for queuing in Hospitals and Banks etc.

5.2 Structure of a QUEUE:



- INSERTIONS are made from Rear. The insertion into a queue is called as Enqueue.
- DELETIONS are made from Front. The deletion from a queue is called as Dequeue.

5.3 INSERT_QUEUE (Q, N, REAR, FRONT, DATA) [1]

[Q is the Queue, N is the size of Queue, FRONT and REAR is the Tracking Pointer of the Queue, DATA is the data to be inserted.]

1. [Check for Space in Queue]

IF FRONT = 1 and REAR = N, or IF FRONT = REAR + 1 then

1.1 OUTPUT "OVERFLOW"

2. Else

[Find new value for REAR]

2.1 IF FRONT = NULL, then

2.1.1 Set FRONT = 1

2.1.2 Set REAR = 1

2.2 ELSE IF REAR = N, then

2.2.1 Set REAR = 1

2.3 ELSE

2.3.1 Set REAR = REAR + 1

[End of IF structure]

[Insert new Element]

2.4 Set Q [REAR] = DATA

3. END

5.4 DELETE_QUEUE (Q, N, FRONT, REAR, DATA) [1]

[Q is the Queue, N is the size, FRONT and REAR are the queue pointers, DATA is to hold the data to be deleted.]

1. [Check for Underflow] IF FRONT = NULL, then

1.1 OUTPUT "UNDERFLOW"

2. Else

2.1 Set DATA = Q [FRONT]

[Move FRONT to its new position]

2.2 IF FRONT = REAR, then

2.2.1 Set FRONT = NULL

2.2.2 Set REAR = NULL

2.3 Else IF FRONT = N, then

2.3.1 Set FRONT = 1

2.4 Else

2.4.1 Set FRONT = FRONT + 1

[End of IF structure]

3. End

5.5 Applications of Queue:

5.5.1 in Operating System:

Following is the flow of a Process execution in Operating system.

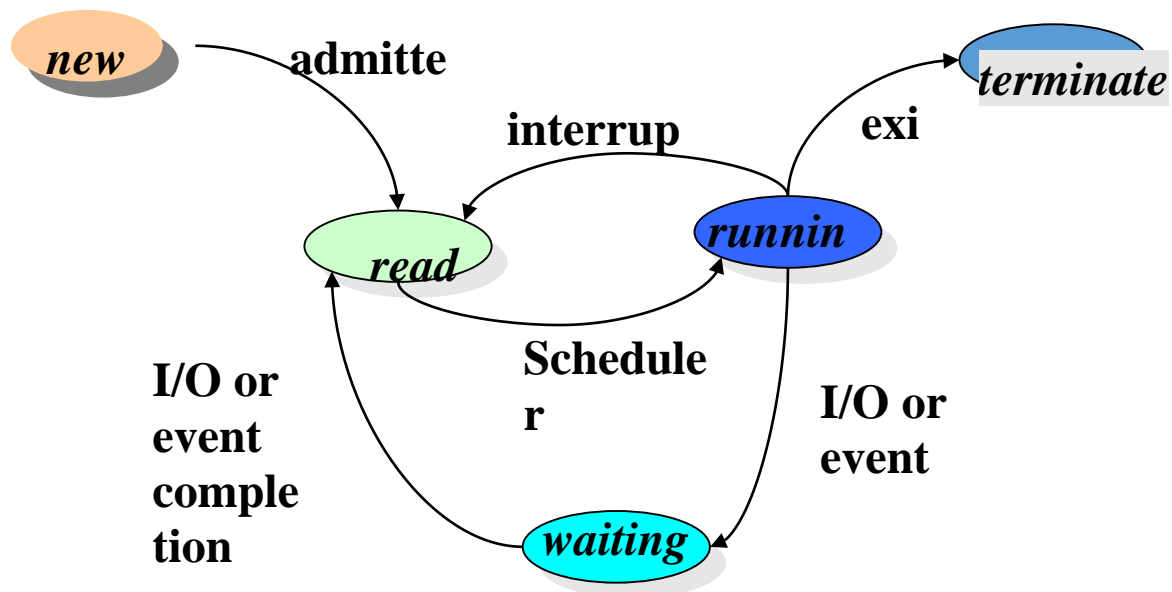


Figure 5.1: State Diagram [google.com/images]

For scheduling Operating system implements several techniques, but most common is First Come First Server, which is actually FIFO means Queue. So for scheduling of processes, OS maintains certain queues which are:

- Job Queue: set of all processes in the system
- Ready Queue: set of ready for execution
- I/O Device Queues: set of processes waiting for an I/O device.

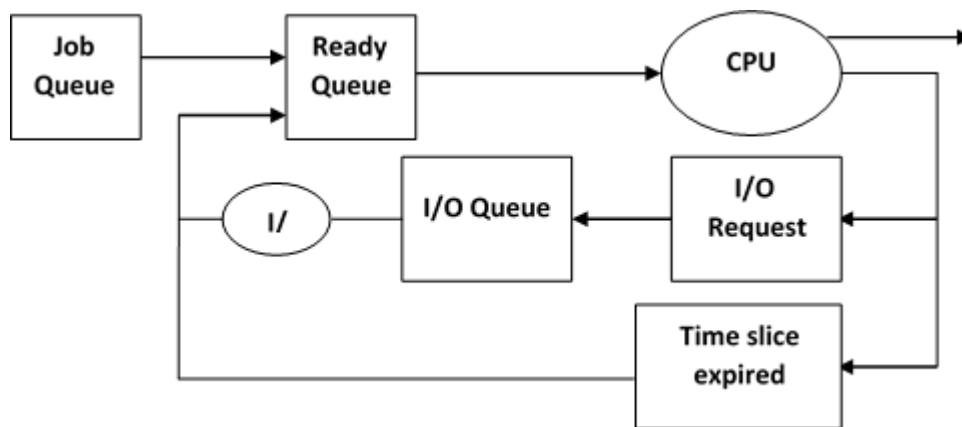


Figure 5.2: Scheduling Queues[google.com/images]

5.6 Priority Queues:

A Priority QUEUE is a collection of elements such that each element has been assigned a Priority and the processing of the elements is done based on the priorities assigned to elements under the following rules:

1. An element of higher priority is processed before any element of lower priority.
2. If two elements have the same priority than the order in which they are inducted into the row will be followed.

These types of QUEUE are used commonly in Operating Systems of different types.

Priority Queues are maintained in the computer in various forms:

1. in a single list, in which the nodes are placed in the queue according to the order of their priorities.
2. Separate Queues are maintained for each level of priorities.

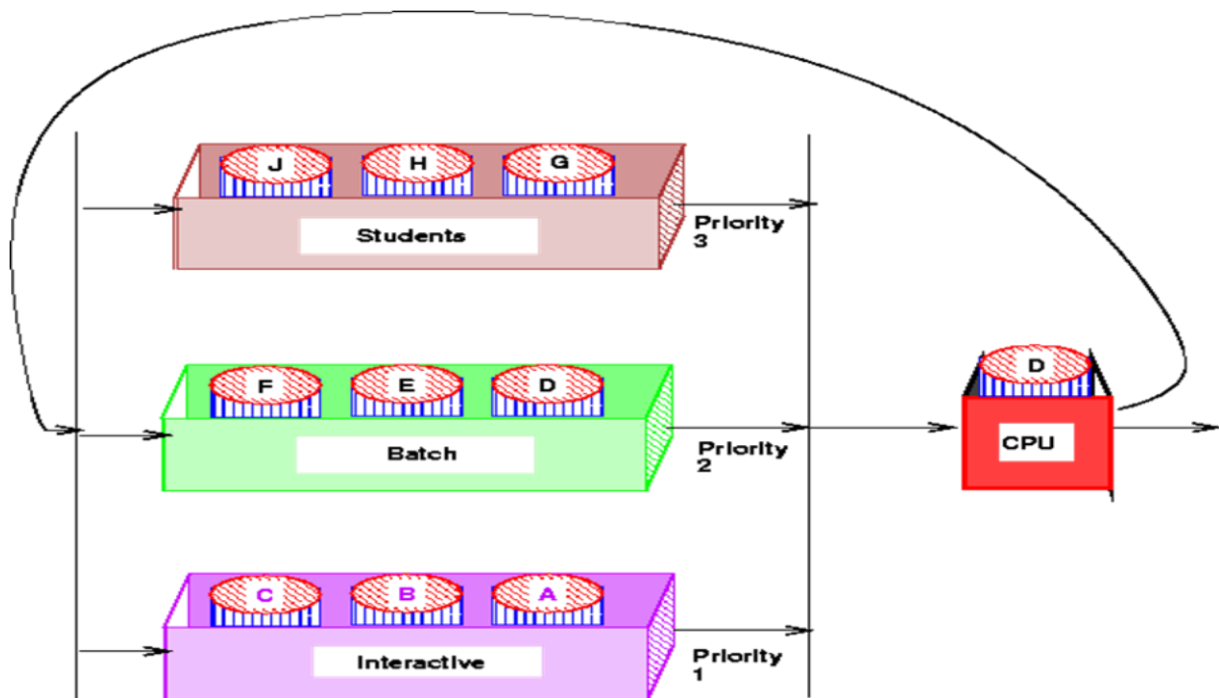


Figure 5.3: Priority Queues[google.com/images]

Chapter VI Link List

This chapter emphasizes on the different algorithms for Link List data structures and applications of Link List in Computer Science.

At the end of the chapter, this course should enable the students to:

OBJECTIVES	LEARNING OUTCOMES (LO)
------------	------------------------

<ol style="list-style-type: none">1. Evaluate and implement data structure2. Justify the selection of an appropriate data structure/algorithm.	<ol style="list-style-type: none">1. Describe alternating methods of implementing ADT.2. Implement ADT using simple data types.3. To describe in detail how basic operations are implemented on data structures.4. Describe a typical application where such a data structure may be found.5. Describe a typical application where such a data structure may be found.6. Justify the selection of the most appropriate data structure / algorithm for a given application.

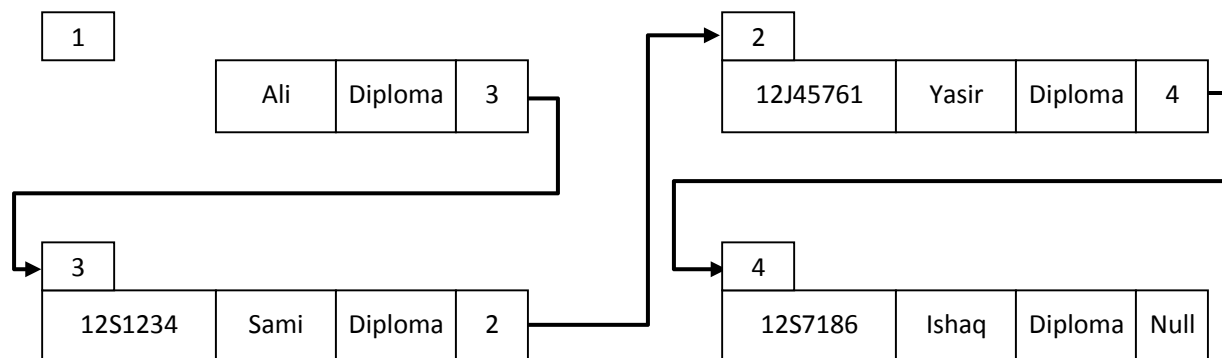
6.1 INTRODUCTION:

A Linked list is an ordered collection of data in which each element contains the location of the next element. The elements in a linked list are called as nodes. Each record is called a node. A data item may be stored anywhere in the memory. A special item, called Pointer, is added to each record. It is used to contain a link to the next record. Thus an item in the list is linked logically with the next item by assigning to it the memory address of next item.

Information Part	Pointer to Next Node

Each node in Linked List consists of at least two fields, the first field contains the data or value called Information field (more than one data fields can be used in order to store information). The second field contains the pointer to the next node.

The elements in a single linked list can be visited starting from beginning and towards its end. The last node will have nothing/NULL in its Pointer to next node, which will represent that this is the last node of Linked list.



The last node of the list contains a NULL value in its Pointer field, which depicts that this is the end of the Linked List.

6.2 REPRESENTATION OF LINKED LIST IN MEMORY:

Linked List is also a Linear Data Structure but its storage is different from that of linear arrays. The items of linked list are stored in memory in scattered form but these are linked together through pointers in the form of a chain. The following example explains this concept using a 'c' structure:

```
struct Student
{
    char id[10]; char
    name[35]; char
    degree[10];

    Student * next;
};
```

In the above example, structure "Student" is defined with Three fields:

- "id", "name" and "degree" are the data/Information fields.
- "next" is a pointer to the object "Student". It is used to store the memory addresses. It contains the memory address of the next object in the chain.

We need one variable of pointer type that points to the Starting Object of the linked list.

```
struct Student * Start;
```

Memory allocation in Link list is dynamic; means there is no need to specify that how many nodes will be inserted and allocating memory in the start. As it is wastage of memory. In Linked list we will allocate memory on the node basis and whenever it will be required.

IN C/C++, nodes of the linked list are created by using the malloc() function or "new" operator during program execution whenever we have to create a node in memory. The malloc() and "new" operator allocates memory for the specified object and returns its memory address to the pointer variable.

6.3 Operations of Linked List:

Link List has the following operations:

1. Insertion
2. Deletion
3. Traversing

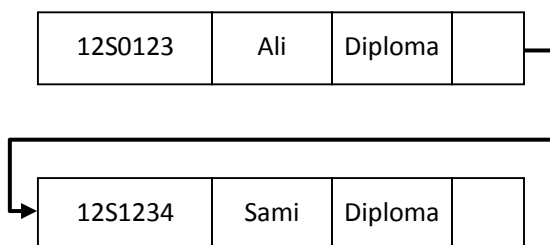
6.3.1 Insertion in the Link List:

The Insertion can be done at three points:

1. Insertion at End
2. Insertion at the Start
3. Insertion at any specific point

6.3.1.1 Insertion at the End:[1]

The Insertion of a new node in the Linked list is simply done by creating a new node and assigning the address of the new node to the already existing end of List node.



1. Create a new node.



2. Assign data to it.

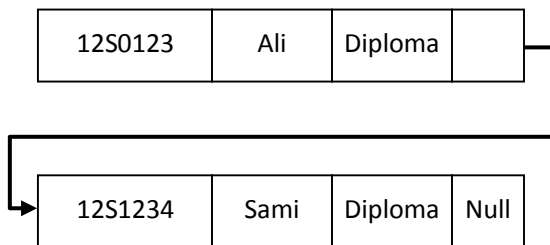
12S01203	Qusai	Diploma	
----------	-------	---------	--

3. Traverse the list and reach at the end where the pointer is NULL and assign the address of newly created node to the end of the List node.

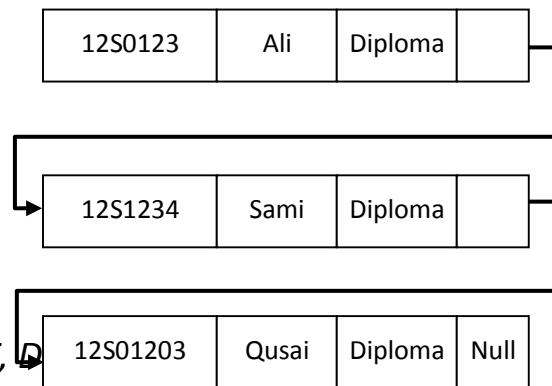
Before Insertion

After Insertion

Start



Start



6.3.1.2 INSERT_END_OF_LIST (START, DATA)

[START is the starting pointer of the List, Data is the information part of the node]

1. Create the New Node TEMP.
2. Set TEMP [INFO] = Data [Insert Data in the Information part of node]
3. Set Temp [NEXT] = NULL [Insert NULL in the NEXT part of List]
4. IF START = NULL Than
 - 4.1 Set START = Temp
5. Else
 - 5.1 Set PTR = START [Initialize PTR]
 - 5.2 Repeat While PTR [NEXT] != NULL [Go to the End of List]
 - 5.2.1 Set PTR = PTR [NEXT]
 - 5.3 Set PTR [NEXT] = TEMP [Attach the newly created node at end]

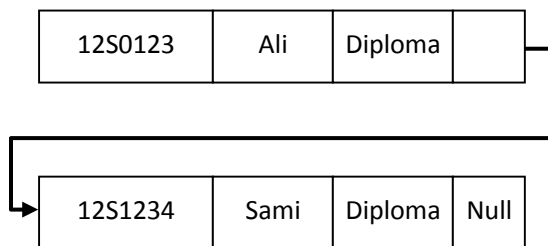
[End of if-else structure]

6. END

6.3.1.3 Insertion at the Start:

The Insertion of a new node in the Link list is simply done by creating a new node and assigning the START to the Next part of newly created node and shifting start on the newly created node.

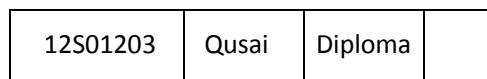
Start



1. Create a new node.

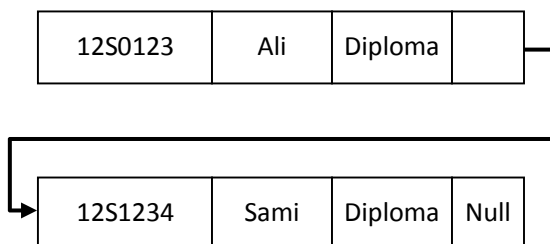


2. Assign data in it.

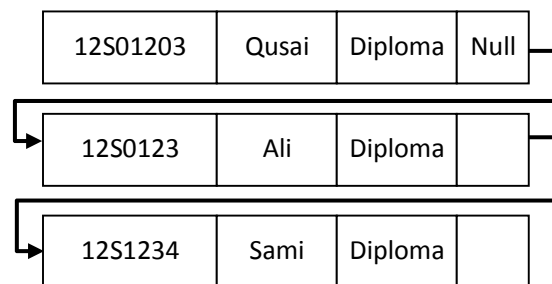


3. Assign the START to the Next part of newly created node and shifting start on the newly created node.

Before insertion



After Insertion



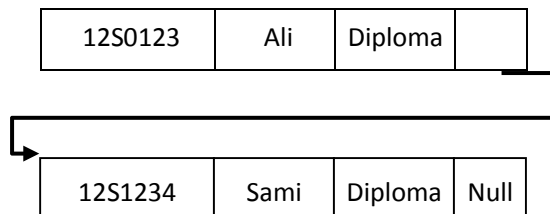
6.3.1.4 INSERT_START_OF_LIST (START, Data)[1]

[START is the starting pointer of the List, Data is the information part of the node]

1. Create the New Node TEMP.
2. Set TEMP [INFO] = Data [Insert Data in the Information part of node]
3. Set TEMP [NEXT] = START [Attach the newly created node at Start]
4. Set START = TEMP
5. END

6.3.1.5 Insertion at the Any Specific Point:

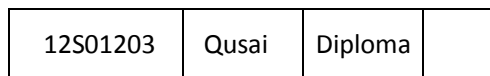
The Insertion of a new node in the Link list is simply done by creating a new node and traversing till the specific point and then inserting the node there.



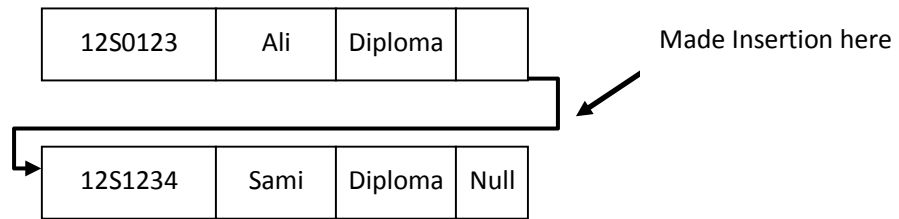
1. Create a new node.



2. Assign data in it.



3. Traversing till the point where Insertion have to be made and then inserting the new node in the List.



6.3.1.6 INSERT_AT_SPECIFIC_POINT (START, Data, Target_data)[1]

[START is the starting pointer of the List, Data is the information part of the node]

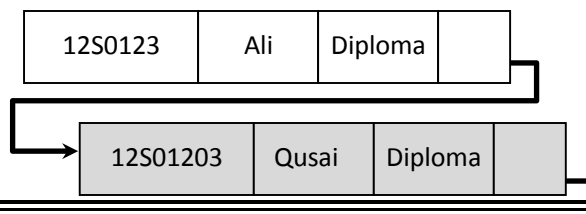
1. Create the New Node TEMP.
2. Set TEMP [INFO] = Data [Insert Data in the Information part of node]
3. Set PTR = START [Initialize PTR]
4. Repeat While PTR [INFO] != Target_data [Go to the Specific point of List]
 - 4.1 Set PTR = PTR [NEXT]

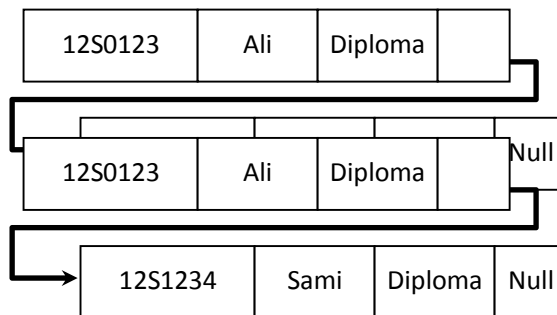
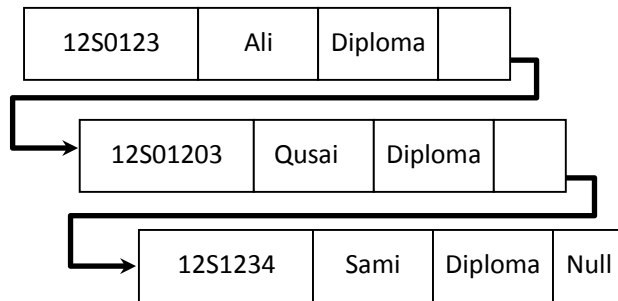
[Attach the newly created node at end]
5. Set TEMP [NEXT] = PTR [NEXT]
6. Set PTR [NEXT] = TEMP
7. End

6.3.2 Deletion from the Link List:

The process of deletion in Link List is straightforward. The pointer (NEXT) value of the node to be deleted is assigned to the Pointer (NEXT) of the Preceding Node.

Preceding Node.





6.3.2.1 DELETION_LINKLIST (START, Data)[1]

[START is the starting pointer of the List, Data is the information part of the node to be deleted]

[Initialize the temporary Node pointers]

1. Set CURRENT = START

2. Set PREVIOUS = START

[Go to the Specific node of List to be deleted]

3. Repeat While CURRENT [INFO]! = Data and CURRENT! = NULL

3.1 Set PREVIOUS = CURRENT

3.2 Set CURRENT = CURRENT [NEXT]

[End of loop]

4. IF CURRENT = NULL Then

4.1 PRINT "NOT FOUND"

5. ELSE

5.1 Set PREVIOUS [NEXT] = CURRENT [NEXT]

[End of IF Else structure]

6. END

6.3.3 Advantages of Using Linked list:

- Linked list is dynamic in nature.
- Linked list can grow and shrink during Runtime.
- Insertion and deletion operations are easy.
- Efficient memory utilization, no need to pre allocate memory.
- Faster access time and can be expanded without memory overhead.
- Linear data structures such as stack and queue can be easily implemented by linked list.

6.3.4 Disadvantages of link list:

- Wastage of memory for allocating the pointer space,
- No random access of elements,
- Difficulty in traversing in the reverse manner.

Chapter VII

Trees

This chapter emphasizes on the different algorithms for Tree data structures and applications of Trees in Computer Science.

At the end of the chapter, this course should enable the students to:

OBJECTIVES	LEARNING OUTCOMES (LO)
3. Evaluate and implement data structure 4. Justify the selection of an appropriate data structure/algorithm.	7. Describe alternating methods of implementing ADT. 8. Implement ADT using simple data types. 9. To describe in detail how basic operations are implemented on data structures. 10. Describe a typical application where such a data structure may be found.
	11. Describe a typical application where such a data structure may be found. 12. Justify the selection of the most appropriate data structure / algorithm for a given application.

7.1 INTRODUCTION:

Arrays, Stack, Queue and linked list are linear data structures. In these structures, the elements are in a line. We Insert and Delete elements in and from these data structures in linear order. But there are applications where linear data structures are not appropriate. In such cases, there is need of some non-linear data structure. Tree is one of the non-linear data structures.

A tree consists of a finite set of elements called nodes and a finite set of lines called branches. The number of branches associated with a node is called as the degree of a node. The number of branches coming inside a node is called as the in degree and number of branches going outside is called as the out degree of a node. If the tree is not empty, the first node in a tree is called as Root. The last node in a tree with out-degree zero is called as a leaf. Nodes that are neither root nor leaf are called as internal nodes. The parent nodes are those nodes that have successors and out degree non- zero. Nodes with a predecessor is called as a child node. Two or more nodes of the same parents are called siblings. Any node from root to the node are called ancestor.

Any node from the node to the leaf is called as descendant.

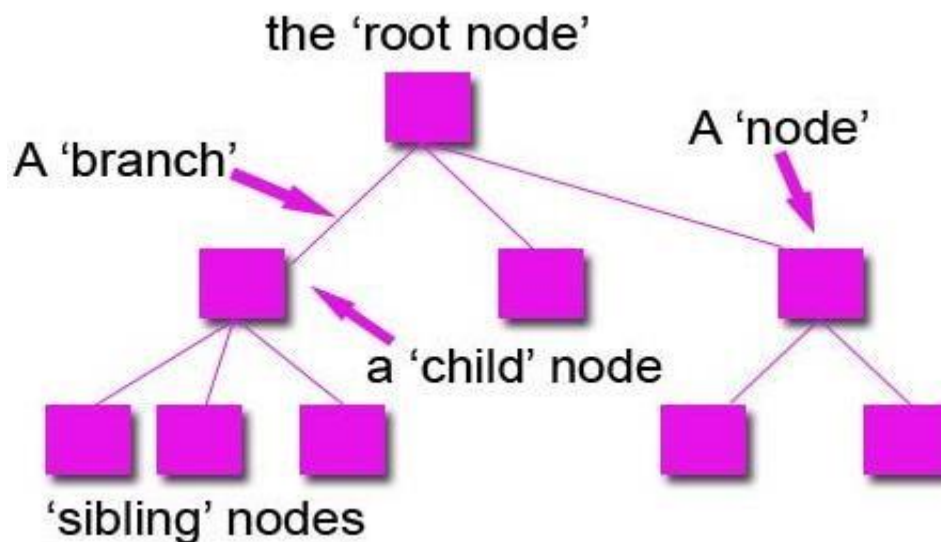


Figure 7.1: General Tree [google.com/images]

The path is a sequence of nodes in which each node is adjacent to the next node. The level is the distance of a node from the root. The height is the level of the leaf in the longest path from root plus one.

There is a relationship between the data that cannot be captured by a linked list or other linear data structure. Here we need a data structure like tree. In some applications, the searching in linear data structures is very tedious. Suppose we want to search a name in a telephone directory having 100000 entries. If this directory is in a linked list manner, we will have to traverse the list from the starting position. We have to traverse on average half of the directory if we find a name. We may not find the required name in the entire directory despite traversing the whole list. Thus it would be better to use a data structure so that the search operation does not take a lot of time. Taking into account such applications, we will now talk about a special tree data structure, known as binary tree.

7.2 Binary Tree:

“A binary tree is a finite set of elements that is either empty or is partitioned into three disjoint subsets. The first subset contains a single element called the root of the tree. The other two subsets are themselves binary trees called the left and right sub-trees”. Each element of a binary tree is called a node of the tree.

Following figure shows a binary tree.

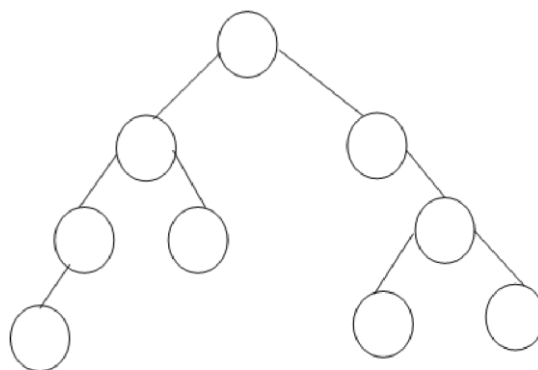


Figure 7.2: Binary Tree[google.com/images]

The definition of tree is of recursive nature. This is due to the fact that we have seen that which definition has been applied to the tree having node A as root, is applied to its subtrees one level downward. Similarly as long as we go down ward in the tree the same definition is used at each level of the tree. And we see three parts

i.e. root, left subtree and right subtree at each level. No nodes in a binary tree can have more than two subtrees.

If there are N nodes, the maximum height of a binary tree can be $H_{\max} = N$. The minimum height is $H_{\min} = \log_2 N + 1$.

7.3 Full Binary Tree:

A full binary tree is a binary tree in which each node has exactly zero or two children, except the leaf.

7.4 Complete Binary Tree:

A complete binary tree is a binary tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right.

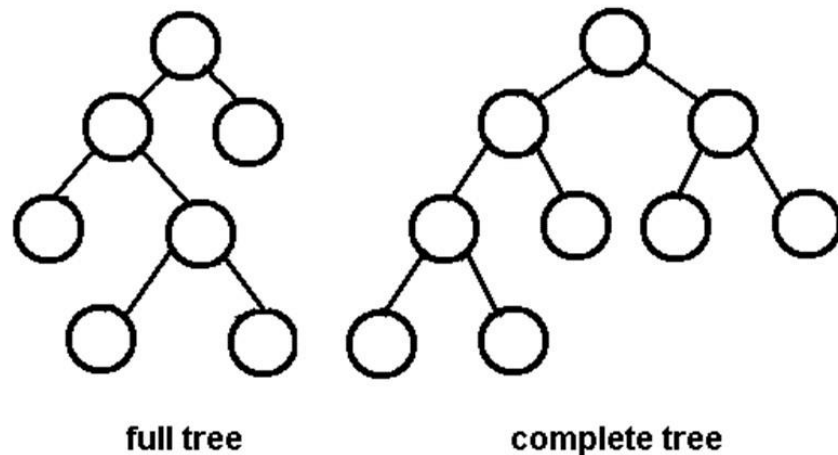


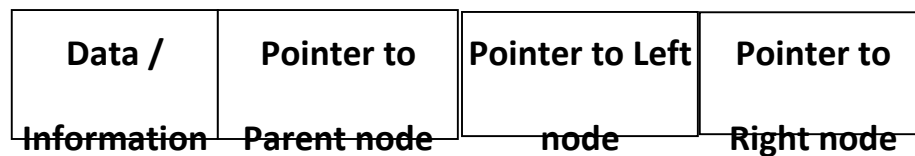
Figure 7.2: Complete and Full Binary Tree[google.com/images]

7.5 Advantages of trees:

Trees are so useful and frequently used, because they have some very serious advantages:

- Trees reflect structural relationships in the data.
- Trees are used to represent hierarchies.
- Trees provide an efficient insertion and searching.
- Trees are very flexible data, allowing to move subtrees around with minimum effort.

7.6 Structure of a Node of Tree:



```
struct node {    int
data;    struct node *
parent;    struct node*
left;    struct node*
right;}
```

7.7 Binary Search Tree:

A binary search tree is a binary tree with the following properties:

1. All Items in the left subtree are less than the root.
2. All items in the right subtree are greater than or equal to the root.
3. Each subtree is itself a binary search tree.

7.7.1 Insertion in a Binary Search Tree:

To insert a node into a BST

1. Find a leaf of the appropriate place and
2. Connect the node to the parent of the leaf.

7.7.2 TREE-INSERT (Tree, node) [1]

```
1.   y = NULL

2.   x = root [Tree]

3.   while x ≠ NULL

      3.1     y = x

      3.2     if Data [node] < Data [x] then

          3.2.1   x = left[x]

      3.3     else

          3.3.1   x = right[x]

4.   Parent[node] = y

5.   if y = NULL then

      5.1     root [Tree] = node

6.   else if Data [node] < Data [y] then

      6.1     left [y] = node

7.   else

      7.1     right [y] = node

8.   End
```

7.7.3 Traversing in BST:

Traversing a tree means visiting all the nodes of a tree in specific order.

There are three different methods of traversing a binary tree:

- Pre-order traversal

- In-order traversal
- Post-order traversal

7.7.3.1 Pre-order traversal:

1. Start at the root node
2. Traverse the left subtree
3. Traverse the right subtree

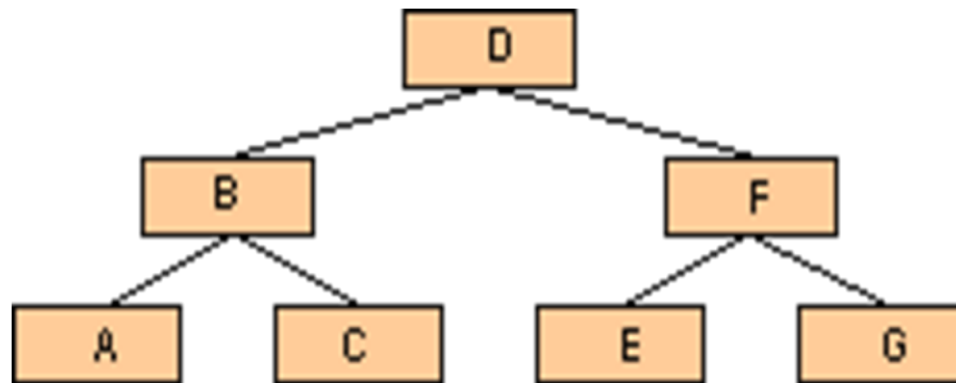


Figure 7.3: BST[google.com/images] The

nodes of this tree would be visited in the order: D B A C F E G

7.7.3.2 In-order traversal:

1. Traverse the left subtree
 2. Visit the root node
 3. Traverse the right subtree
- In figure 7.3 In order traversal is: A B C D E F G (List is sorted)

7.7.3.3 Post-order traversal:

1. Traverse the left subtree
2. Traverse the right subtree

3. Visit the root node

In Figure 7.3 Post-order traversal is: A C B E G F D

1. *Lipschutz Simour (2012) – Data Structures Shaum series.*
2. *Deshpande D.S. Kokde O.G (2008), C and Data Structures Dreamtech Press,1st ed.*
3. *Tenenbaum Aaron M by Tenenbaum (2012) – Data Structures using C++ Prentice Hall*

References