INTRODUCTION TO DATA STRUCTURE

DATA STRUCTURE- OVERVIEW

- Data Structure is a systematic way to organize data in order to use it efficiently. Following terms are the foundation terms of a data structure.
- **Interface**
  - Each data structure has an interface. Interface represents the set of operations that a data structure supports.
  - An interface only provides the list of supported operations, type of parameters they can accept and return type of these operations.
- **Implementation**
  - Implementation provides the internal representation of a data structure.
  - Implementation also provides the definition of the algorithms used in the operations of the data structure.

DATA STRUCTURE- OVERVIEW

Characteristics of a Data Structure:

- **Correctness**
  - Data structure implementation should implement its interface correctly.
- **Time Complexity**
  - Running time or the execution time of operations of data structure must be as small as possible.
- **Space Complexity**
  - Memory usage of a data structure operation should be as little as possible.

DATA STRUCTURE- OVERVIEW

Need for Data Structure:

- As applications are getting complex and data rich, there are three common problems that applications face now-a-days.
- **Data Search**
  - Consider an inventory of 1 million(10^6) items of a store. If the application is to search an item, it has to search an item in 1 million(10^6) items every time slowing down the search. As data grows, search will become slower.
- **Processor speed**
  - Processor speed although being very high, falls limited if the data grows to billion records.
- **Multiple requests**
  - As thousands of users can search data simultaneously on a web server, even the fast server fails while searching the data.

DATA STRUCTURE- OVERVIEW

- To solve the above-mentioned problems, data structures come to rescue.
- Data can be organized in a data structure in such a way that all items may not be required to be searched, and the required data can be searched almost instantly.

Execution Time Cases:

- There are three cases which are usually used to compare various data structure's execution time in a relative manner.
- **Worst Case**
  - This is the scenario where a particular data structure operation takes maximum time it can take.
  - If an operation's worst case time is $f(n)$ then this operation will not take more than $f(n)$ time where $f(n)$ represents function of n.

DATA STRUCTURE- OVERVIEW

- **Average Case**
  - This is the scenario depicting the average execution time of an operation of a data structure.
  - If an operation takes $f(n)$ time in execution, then m operations will take m*$f(n)$ time.
- **Best Case**
  - This is the scenario depicting the least possible execution time of an operation of a data structure.
  - If an operation takes $f(n)$ time in execution, then the actual operation may take time as the random number which would be maximum as $f(n)$.

Basic Terminology:

- **Data** − Data are values or set of values.
- **Data Item** − Data item refers to single unit of values.
- **Group Items** − Data items that are divided into sub items are called as Group Items.

DATA STRUCTURE- OVERVIEW

- **Elementary Items** − Data items that cannot be divided are called as Elementary Items.
- **Attribute and Entity** − An entity is that which contains certain attributes or properties, which may be assigned values.
- **Entity Set** − Entities of similar attributes form an entity set.
- **Field** − Field is a single elementary unit of information representing an attribute of an entity.
- **Record** − Record is a collection of field values of a given entity.
- **File** − File is a collection of records of the entities in a given entity set.

DATA STRUCTURE- CLASSIFICATION

1. STATIC AND DYNAMIC DATA STRUCTURE:

- **Static Data Structure** are those whose size is fixed at compile time and doesn't grow or shrink at run time. E.g. Array
- **Dynamic Data Structure** are those whose size is not fixed at compile time and that can grow or shrink at run time so as to make efficient use of memory. E.g. Linked List
- **Arrays and Linked list** are basic data structures that are used to implement other data structure such as stack, queue, trees and graphs. So any other data structure can be static or dynamic depending on whether they are implementing using an array or linked list.

2. PRIMITIVE AND NON-PRIMITIVE DATA STRUCTURE:

- **Primitive Data Structure** are those data types which are provided by a programming language as basic building block. So, primitive data types are predefined types of data, which are supported by programming language. E.g. integer, float, character etc.

DATA STRUCTURE- CLASSIFICATION

- **Non-Primitive Data Structure** are those data types which are not defined by programming language but are instead created by the programmer by using primitive data types. E.g. array, linked list, stack, queue, tree etc.

3. LINEAR AND NON-LINEAR DATA STRUCTURE:

- A data structure is said to be **linear** if its elements form a sequence i.e. linear list. So, here while traversing the data elements sequentially, only one element can directly be reached. Example: Array, stack, queue
- A data structure is said to be **non linear** if its elements do not form a sequence. So, here every data items is attached to several other data items. Tree and Graph are examples of non linear data structure.

DATA TYPES AND ABSTRACT DATA TYPE (ADT)

- A **data type** is a classification of data, which can store specific type of information.
- Data types are primarily used in computer programming in which variables are created to store data.
- Each variable is assigned a data type that determine what type of data the variable may contain.
- Data type is a collection of values and set of operation on those values.
- Example: in the statement int a=5, 'a' is a variable of data type integer which stores integer value 5 and allow different mathematical operations like addition, subtraction, multiplication and division.

ADT:

- We can perform some specific operations with data structure, so data structure with these operations are called **Abstract Data Types (ADT).**
- ADT have their own data type and methods to manipulate data.
- It is a useful tool for specifying the logical properties of a data type.

DATA TYPES AND ABSTRACT DATA TYPE (ADT)

- A data type or data structure is the implementation of ADT.
- So ADT allow us to work with abstract idea behind a data type or data structure without getting bagged down in the implementation detail. The abstraction in this case is called an abstract data type.

Advantage of abstract data type:

- **Encapsulation:**
  - Encapsulation is a process by which we can combine code and data and it manipulates into a single unit.
  - While working with ADT, the user doesn't require knowing how the implementation works because implementation is encapsulated in a simple interface.
- **Flexibility:**
  - If different implementations of ADT are equivalent then they can be interchangeable in the code. So user have flexibility to select the one which is most efficient in different situations.

DATA TYPES AND ABSTRACT DATA TYPE (ADT)

- **Localization if change:**
  - If ADT is used then if changes are made to the implementation then it doesn't require any changes in the code.
- Example: Stack is ADT which have pop and push operation for deleting and inserting element.

DATA STRUCTURE OPERATIONS

- **Traversing** :- Accessing each record so that some item in the record may be processed.
- **Searching** :- Finding the location of the record with the given key value
- **Inserting** :- Adding new record to the data structure
- **Deleting** :- Removing record from the data structure
- **Sorting** :- Arranging the record in some logical order
- **Merging** :- Combining the record of different sorted file into single sorted file.

Example:

- An array A having n element is ADT which have following operations
  - Create(A) :- create an array A
  - Insert(A,x) :- insert an element x into an array A in any location
  - Delete(A,x) :- remove element x

DATA STRUCTURE OPERATIONS

- Traverse(A):- access each element of an array A
- Modify(A,x,y) :- Change the old element x with new element y
- Search(A,x) :- search element x in an array A
- Merge (A,B,C) :- Combine the elements of array B and C and store in new array A

1. Which of this best describes an array?

a) A data structure that shows a hierarchical behavior

Container of objects of similar types

c) Arrays are immutable once initialized

d) Array is not a data structure

2. How do you initialize an array in C?

a) int arr[3] = (1,2,3);

b) int arr(3) = {1,2,3};

int arr[3] = {1,2,3};

d) int arr(3) = (1,2,3);

3. How do you instantiate an array in Java?

a) int arr[] = new int(3);

b) int arr[];

int arr[] = new int[3];

d) int arr() = new int(3);

4. Which of the following is the correct way to declare a multidimensional array in Java?

a) int[] arr;

b) int arr[[]];

c) int[][]arr;

d) int[[]] arr;

5. What is the output of the following Java code

```
public class array
{
        public static void main(String args[]
        {
                int []arr = {1,2,3,4,5};
                System.out.println(arr[2]);
                System.out.println(arr[4]);
        }
}
```

a) 3 and 5

b) 5 and 3

c) 2 and 4

d) 4 and 2

6. What is the output of the following Java code?

```
public class array
{
        public static void main(String args[])
        {
                int []arr = {1,2,3,4,5};
                System.out.println(arr[5]);
        }
}
```

a) 4

b) 5

c) ArrayIndexOutOfBoundsException

d) InavlidInputException

7. When does the ArrayIndexOutOfBoundsException occur?

a) Compile-time

b) Run-time

c) Not an error

d) Not an exception at all

13. Elements in an array are accessed _____

a) randomly

b) sequentially

c) exponentially

d) logarithmically

9. What are the advantages of arrays?

a) Objects of mixed data types can be stored

b) Elements in an array cannot be sorted

c) Index of first element of an array is 1

d) Easier to store elements of same data type

10. What are the disadvantages of arrays?

a) Data structure like queue or stack cannot be implemented

b) There are chances of wastage of memory space if elements inserted in an array are lesser than the allocated size

c) Index value of an array can be negative

d) Elements are sequentially accessed

11. Assuming int is of 4bytes, what is the size of int arr[15];?

a) 15

b) 19

c) 11

d) 60

12. In general, the index of the first element in an array is _____

a) 0

b) -1

c) 2

d) 1

THE STACK

INTRODUCTION

- A stack is an ordered collection of homogenous data elements where the insertion and deletion operation occurs at only one ends called **top** of the stack.
- As all the insertion and deletion is performed from top of the stack, the last item that is inserted in a stack is the first one to be deleted.
- Therefore stack is also called as **last in first out (LIFO)** data structure.
- The insertion (addition) operation is referred to as PUSH and the deletion (remove) operation as POP.
- A stack is said to be empty or underflow, if the stack contains no elements. At this point, the **top** of the stack is present at the bottom of the stack.
- In addition, it is overflow when the stack becomes full, i.e. , no other elements can be pushed onto the stack. At this point, **top** pointer is at the highest location of the stack.
- Consider an array of size[4]with the name stack i.e. stack[4] then the insertion and deletion operation that can be performed on stack is shown below:

INTRODUCTION

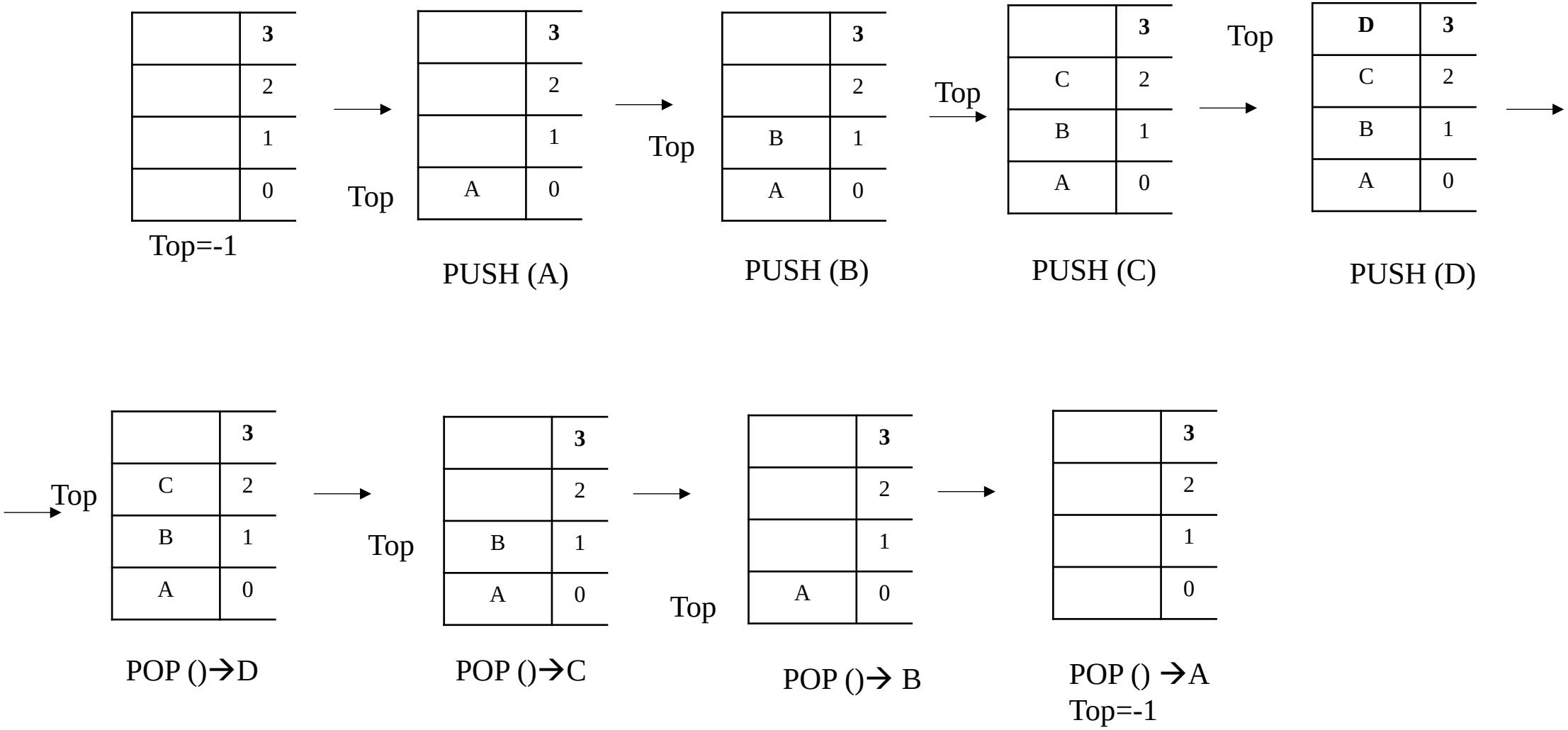


Fig: PUSH and POP operation

STACK AS AN ADT/ OPERATION ON STACK

- Stack is generally implemented with two basic operation such as PUSH and POP.
- These two operations are also called as primitive operation on stack.

PUSH operation

- The processing of adding a new element to the top of stack is called PUSH operation.
- For each PUSH operation, the value of top is increased by 1.
- If the array is full and no new element can be accommodate, then the stack overflows condition occurs.
- E.g. PUSH (A,S)→ insert element A at the top of the stack, S

POP operation:

- The process of deleting an element from the top of the stack is called POP operation.
- After every POP operation, the stack is decremented by one.
- If there is no element in the stack and the POP operation is performed then the stack underflow condition occurs.

STACK AS AN ADT/ OPERATION ON STACK

- E.g. POP(S) → deleting the element from top of the stack, S

- Also, we can define following additional operation that can be performed on stack.
- **TOP (S) →** Return the elements at the top of the stack.
- **ISEMPTY (S) →** Check the stack, S, is empty or not. If the stack is empty it returns true (1) otherwise false (0)
- **ISFULL (S) →** Check the stack, S, is full or not. If the stack is full it returns true (1) otherwise false (0)
- **MAKENULL (S) →** Make stack S is an empty stack.
- **SEARCH (X,S) →** Get the location of item X in stack S.
- **TRAVERSE (S) →** Read each elements only once from the stack S
- **MAX (S) →** Get the maximum value from the stack, S
- **MIN (S) →** Get the minimum value from the stack, S

ALGORITHM FOR PUSH AND POP OPERATION

PUSH OPERATION:

- Assuming a stack named STACK of size MAX with the TOP pointer that points to the top most element of the stack and DATA is the data item to be pushed.

1.If TOP==(MAX-1) then

   a. Display “Stack Overflow”
   b. Exit

2.Read DATA

3.TOP=TOP+1

4.STACK[TOP]← DATA

5.Exit

ALGORITHM FOR PUSH AND POP OPERATION

POP OPERATION:

- Assuming a stack named STACK of size MAX with the TOP pointer that points to the top most element of the stack and DATA is the data item to be popped.

1.If TOP==-1 then

   a. Display “Stack underflow”

   b. Exit

2.DATA←STACK[TOP]

3.TOP=TOP-1

4.Exit

STACK IMPLEMENTATION

- Static Implementation (Using Arrays)
- Dynamic Implementation (Using Pointer)

Static Implementation:

- Static implementation using array is very simple technique but is not a flexible way, as the size of the stack has to be declared during the program design, because after that, the size can not be varied.
- Moreover, static implementation is not an efficient method when resource optimization is concerned.

Dynamic Implementation:

- when a stack is implemented by using link list i.e. uses pointer to implement the stack, then the stack is said to be dynamic.
- In this case memory is dynamically allocated to the stack and the memory size can grow and shrink at run time.

STACK APPLICATION

- It is used to reverse the string.
- It is used to implement function call.
- It is used to maintaining the undo list for an application.
- It is used for checking the validity for expression
- It is used for converting infix expression to post fix or prefix expression
- To keep page visited history in web browser.
- Sorting
- Backtracking

INFIX, PREFIX AND POSTFIX NOTATION

- **Infix notation:** The operator symbol is placed between two operand. E.g. a+b
- **Prefix notation:** The operator symbol is placed before two operand E.g. +ab
- **Postfix Notation:** The operator is placed after two operand E.g. ab+
- Using infix notation, if the expression consists of more than one operator and brackets, the precedence rule (BODMAS) should be applied to decide which operator or which section of expression are evaluated first.
- So, the computer usually evaluates an infix expression first by converting it to postfix and evaluating the postfix expression by using stack.
- As soon as an operator appears in the postfix expression during scanning of postfix expression the topmost operands are popped off and are calculated by applying the encountered operator.

Operator Precedence

Exponential operator	^	Highest precedence (**Note : ^ = $**)
Multiplication/Division	*, /	Next precedence
Addition/Subtraction	+, -	Least precedence

CONVERSION OF INFIX TO POSTFIX EXPRESSION

ALGORITHM FOR CONVERTING INFIX EXPRESSION TO POSTFIX EXPRESSION USING STACK

- Suppose Q is an arithmetic expression written in infix notation, and P is the expression written in postfix notation, then algorithm that finds the equivalent postfix expression is given below:

1. Scan Q from left to right for each element of Q until end of infix expression is encountered
2. If scanned element is
    a. **Operand:**
        I. Add the operand to the postfix expression, P
    b. **Operator:**
        I. If the precedence of the operator on the top of the stack is greater or equal to precedence of scanned operator, pop the operator from the stack and append it to the postfix expression, P. Repeat this step until the operator at the top of the stack has precedence less than the scanned operator or stack become empty.

CONVERSION OF INFIX TO POSTFIX EXPRESSION

II. Push the scanned operator on the top of the stack.

c. Left parenthesis:

I. Push the left parenthesis on to the top of stack.

d. Right parenthesis

I. Pop stack element one by one and append to the postfix expression, until a last parenthesis is popped.

3. POP all the entries from the stack and append them to postfix expression

CONVERSION OF INFIX TO POSTFIX EXPRESSION

Example: Consider the following arithmetic infix expression P

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

Following table shows the character (operator, operand or parenthesis) scanned, status of the stack and postfix expression Q of the infix expression P.

Character Scanned	Stack	Postfix String
A		A
+	+	A
(	+ (	A
B	+ (	AB
/	+ (/	AB
C	+ (/	ABC
-	+ (-	ABC/
(	+ (- (	ABC/
D	+ (- (	ABC/D
*	+ (- (*	ABC/D
E	+ (- (*	ABC/DE
$	+ (- (* $	ABC/DE
F	+ (- (* $	ABC/DEF
)	+ (-	ABC/DEF$*
+	+ (+	ABC/DEF$* -
G	+ (+	ABC/DEF$* - G
)	+	ABC/DEF$* - G+
*	+ *	ABC/DEF$* - G+
H	+ *	ABC/DEF$* - G + H
		ABC/DEF$*- G + H * +

CONVERSION OF INFIX TO POSTFIX EXPRESSION

Example:

Input string:
a+b*c+(d*e+f)*g

Character scanned	Operator Stack	Postfix String
a		a
+	+	a
b	+	ab
*	+ *	ab
c	+ *	abc
+	+	abc * +
(	+ (	abc * +
d	+ (	abc * + d
*	+ (*	abc * + d
e	+ (*	abc * + de
+	+ (+	abc * + de *
f	+ (+	abc * + de * f
)	+	abc * + de * f +
*	+ *	abc * + de * f +
g	+ *	abc * + de * f + g
		abc * + de * f + g * +

CONVERSION OF INFIX TO POSTFIX EXPRESSION

Example: Input string ((A - (B+C)) * D) $ (E + F)

Character scanned	Operator stack	Postfix string
(	(	
(	((	
A	((	A
-	((-	A
(	((- (	A
B	((- (	AB
+	((- (+	AB
C	((- (+	ABC
)	((-	ABC+
)	(	ABC+ -
+	(*	ABC+ -
D	(*	ABC+ - D
)		ABC+ - D*
$	$	ABC+ - D*
(	$ (	ABC+ - D*
E	$ (	ABC+ - D*E
+	$ (+	ABC+ - D*E
F	$ (+	ABC+ - D*EF
)	$	ABC+ - D*EF+
		ABC+ - D*EF+$

For practice:

1. (A-B)-C+D*E+F

ANS: AB-C-DE*+F+

EVALUATION OF POSTFIX EXPRESSION

- Once the infix expression is converted into postfix expression, we can write the **algorithm** for evaluating postfix expression as below:

1. Scan the elements in postfix expression from left to right until end of the postfix expression is encountered.
2. If the scanned element is:
    a. Operand : PUSH it into stack
    b. Operator :
        I. POP the top two values
        II. Operate the two values by using the operator (just below the top of the stack operand to top of the stack operand)
        III. PUSH the result back to the stack.
3. Get the result from top of the stack.

EVALUATION OF POSTFIX EXPRESSION

Example 6 2 3 + - 3 8 2 / + * 2 $ 3 +

Char Scanned	Operand1	Operand2	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	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	7	7	7	7
2	1	7	7	7,2
$	7	2	49	49
3	7	2	49	49,3
+	49	3	52	52

EVALUATION OF POSTFIX EXPRESSION

Example: 6 5 2 3 + 8 * + 3 + *

Char Scanned	Operand 1	Operand 2	Value	Operator
6				6
5				6,5
2				6,5,2
3				6,5,2,3
+	2	3	5	6,5,5
8	2	3	5	6,5,5,8
*	5	8	40	6,5,40
+	5	40	45	6,45
3	5	40	45	6,45,3
+	45	3	48	6,48
*	6	48	288	288

EVALUATION OF POSTFIX EXPRESSION

Example:

- ABC+*DE/- where A=5, B=6, C=2, D=12, E=4

Solution:

Given expression becomes: 5 6 2 +* 12 4 / -

Char Scanned	Operand 1	Operand 2	Value	Stack
5				5
6				5, 6
2				5,6,2
+	6	2	8	5,8
*	5	8	40	40
12	5	8	40	40, 12
4	5	8	40	40, 12, 4
/	12	4	3	40, 3
-	40	3	37	37

INFIX TO PREFIX CONVERSION

- Suppose Q is an arithmetic expression written in infix notation then the algorithm that finds the equivalent prefix expression is given below:

1. Reverse the element of arithmetic expression, Q
2. Replace → (by), { by } , [by] ,) by (, } by { and] by [
3. Convert the reversed arithmetic expression Q into postfix expression.
4. Scan Q from left to right for each element of Q until end of infix expression is encountered
5. If scanned element is
    a. Operand:
        i. Add the operand to the postfix expression, P
    b. Operator:
        i. If the precedence of the operator on the top of the stack is greater or equal to precedence of scanned operator, pop the operator from the stack and append it to the postfix expression, P. Repeat this step until the operator at the top of the stack has precedence less than the scanned operator or stack become empty.

INFIX TO PREFIX CONVERSION

ii. Push the scanned operator on the top of the stack.

c) Left parenthesis:

i. Push the left parenthesis on to the top of stack.

d)Right parenthesis

i. Pop stack element one by one and append the to the postfix expression, until a last parenthesis is popped.

6.POP all the entries from the stack and append them to postfix expression

7.Reverse the obtained postfix expression element to obtain required prefix expression.

INFIX TO PREFIX CONVERSION

Example:

Convert into prefix expression : (A + B * C)

Solution: Reverse the above infix expression:

) C * B + A (

Replacing (by) and) by (, we get

(C * B + A)

Scanned Character	Stack	Postfix Expression
(	(	
C	(	C
*	(*	C
B	(*	C B
+	(+	C B *
A	(+	C B * A
)		C B * A +

We know, Prefix is equal to reverse of postfix expression. So, the required prefix expression is:

+ A * B C

INFIX TO PREFIX CONVERSION

Example:

Convert into prefix expression (A+B)*(C-D)

Solution: Reverse the above infix expression:

)D-C(*)B+A(

Replacing (by) and) by (, we get

(D-C)*(B+A)

We know, Prefix is equal to reverse of postfix expression. So, the required prefix expression is:

*+AB-CD

Scanned Character	Stack	Postfix Expression
(	(	
D	(	D
-	(-	D
C	(-	DC
)		DC-
*	*	DC-
(	*(	DC-
B	*(	DC-B
+	*(+	DC-B
A	*(+	DC-BA
)	*	DC-BA+
		DC-BA+*

EVALUATION OF PREFIX EXPRESSION

1. Accept Prefix expression
2. Reverse the input prefix expression or scan the element in prefix expression from right to left until the end of the prefix expression is encountered.
3. If the scanned character is operand, PUSH it onto the stack.
4. If it is operator, then POP two elements from stack and perform the operation performed by operator (from top of the stack operand to just below the top of stack operand) and PUSH the result back to stack.
5. Get result from top of the stack

EVALUATION OF PREFIX EXPRESSION

Example:

Evaluate the following prefix expression: a) /+5,3-4,2 b) +5*3,2

Char Scanned	Operand 1	Operand 2	Value	Stack
2				2
4				2,4
-	4	2	2	2
3	4	2	2	2,3
5	4	2	2	2,3,5
+	5	3	8	2,8
/	8	2	4	4

Char Scanned	Operand 1	Operand 2	Value	Stack
2				2
3				2,3
*	3	2	6	6
5	3	2	6	6,5
+	5	6	11	11

POSTFIX TO INFIX CONVERSION

1. Scanned the POSTFIX expression from left to right
2. If scanned character is
    a. Operand
        i. PUSH it into the stack
    b. Operator
        i. Pop the top two values from the stack
        ii. Put the operator, with the values as arguments and form a string
        iii. Encapsulate the resulted string with parenthesis
        iv. PUSH the resulted string back to stack
3. The value in the stack is the desired infix string.

POSTFIX TO INFIX CONVERSION

Example:

ABC-+DE-FG-H+/*

Expression	Stack
ABC-+DE-FG-H+/*	Null
BC-+DE-FG-H+/*	A
C-+DE-FG-H+/*	A,B
-+DE-FG-H+/*	A,B,C
+DE-FG-H+/*	A,(B-C)
DE-FG-H+/*	(A+(B-C))
E-FG-H+/*	(A+(B-C)),D
-FG-H+/*	(A+(B-C)),D,E
FG-H+/*	(A+(B-C)),(D-E)
G-H+/*	(A+(B-C)),(D-E),F
-H+/*	(A+(B-C)),(D-E),F,G
H+/*	(A+(B-C)),(D-E),(F-G)
+/*	(A+(B-C)),(D-E),(F-G),H
/*	(A+(B-C)),(D-E),((F-G)+H)
*	(A+(B-C)),((D-E)/((F-G)+H))
	(A+(B-C))*((D-E)/((F-G)+H))

1. Process of inserting an element in stack is called _____

a) Create

Push

c) Evaluation

d) Pop

2. Process of removing an element from stack is called _____

a) Create

b) Push

c) Evaluation

Pop

3. In a stack, if a user tries to remove an element from an empty stack it is called _____

Underflow

b) Empty collection

c) Overflow

d) Garbage Collection

4. Pushing an element into stack already having five elements and stack size of 5, then stack becomes _____

a) Overflow

b) Crash

c) Underflow

d) User flow

5. Entries in a stack are “ordered”. What is the meaning of this statement?

a) A collection of stacks is sortable

b) Stack entries may be compared with the ‘<‘ operation

c) The entries are stored in a linked list

d) There is a Sequential entry that is one by one

6. Which of the following is not the application of stack?

a) A parentheses balancing program

b) Tracking of local variables at run time

c) Compiler Syntax Analyzer

d) Data Transfer between two asynchronous process

7. Consider the usual algorithm for determining whether a sequence of parentheses is balanced. The maximum number of parentheses that appear on the stack AT ANY ONE TIME when the algorithm analyzes: (()(())(())))?

a) 1

b) 2

c) 3

d) 4 or more

8. Consider the usual algorithm for determining whether a sequence of parentheses is balanced. Suppose that you run the algorithm on a sequence that contains 2 left parentheses and 3 right parentheses (in some order). The maximum number of parentheses that appear on the stack AT ANY ONE TIME during the computation?

a) 1

b) 2

c) 3

d) 4 or more

9. What is the value of the postfix expression 6 3 2 4 + – *?

a) 1

b) 40

c) 74

d) -18

10. Here is an infix expression: 4 + 3*(6*3-12). Suppose that we are using the usual stack algorithm to convert the expression from infix to postfix notation. The maximum number of symbols that will appear on the stack AT ONE TIME during the conversion of this expression?

a) 1

b) 2

c) 3

4

11. The postfix form of the expression (A+ B)*(C*D- E)*F / G is?

a) AB+ CD*E – FG /**

b) AB + CD* E – F **G /

AB + CD* E – *F *G /

d) AB + CDE * – * F *G /

12. The data structure required to check whether an expression contains a balanced parenthesis is?

Stack

b) Queue

c) Array

d) Tree

13. What data structure would you most likely see in non recursive implementation of a recursive algorithm?

a) Linked List

b) Stack

c) Queue

d) Tree

14. The process of accessing data stored in a serial access memory is similar to manipulating data on a _____

a) Heap

b) Binary Tree

c) Array

b) Stack

15. The postfix form of A*B+C/D is?

a) *AB/CD+

b) AB*CD/+

c) A*BC+/D

d) ABCD+/*

16. Which data structure is needed to convert infix notation to postfix notation?

a) Branch

b) Tree

c) Queue

Stack

17. The prefix form of A-B/ (C * D ^ E) is?

a) -/*^ACBDE

b) -ABCD*^DE

-A/B*C^DE

d) -A/BC*^DE

18. What is the result of the following operation?

Top (Push (S, X))

X

b) X+S

c) S

d) XS

19. The prefix form of an infix expression (p + q) – (r * t) is?

a) + pq – *rt

b) – +pqr * t

– +pq * rt

d) – + * pqrt

20. Which data structure is used for implementing recursion?

a) Queue

Stack

c) Array

d) List

21. The result of evaluating the postfix expression 5, 4, 6, +, *, 4, 9, 3, /, +, * is?

a) 600

350

c) 650

d) 588

22. Convert the following infix expressions into its equivalent postfix expressions.

(A + B ΛD)/(E – F)+G

(A B D Λ + E F – / G

b) (A B D +Λ E F – / G +)

c) (A B D Λ + E F/- G +)

d) (A B D E F + Λ / – G +)

23. Convert the following Infix expression to Postfix form using a stack.

x + y * z + (p * q + r) * s, Follow usual precedence rule and assume that the expression is legal.

xyz*+pq*r+s*+

b) xyz*+pq*r+s+*

c) xyz+*pq*r+s*+

d) xyzp+**qr+s*+

24. Which of the following statement(s) about stack data structure is/are NOT correct?

a) Linked List are used for implementing Stacks

b) Top of the Stack always contain the new node

Stack is the FIFO data structure

d) Null link is present in the last node at the bottom of the stack

25. Consider the following operation performed on a stack of size 5.

After the completion of all operation, the number of elements present in stack is?

```
Push(1);
Pop();
Push(2);
Push(3);
Pop();
Push(4);
Pop();
Pop();
Push(5);
```

a) 1

b) 2

c) 3

d) 4

26. Which of the following is not an inherent application of stack?

a) Reversing a string

b) Evaluation of postfix expression

c) Implementation of recursion

d) Job scheduling

27. The type of expression in which operator succeeds its operands is?

a) Infix Expression

b) Prefix Expression

c) Postfix Expression

d) Both Prefix and Postfix Expressions

28. Assume that the operators +,-, x are left associative and ^ is right associative. The order of precedence (from highest to lowest) is ^, x, +, -. The postfix expression for the infix expression a + b x c – d ^ e ^ f is?

a) a b c x + d e f ^ ^ –

a b c x + d e ^ f ^ –

c) a b + c x d – e ^ f ^

d) – + a x b c ^ ^ d e f

29. If the elements “A”, “B”, “C” and “D” are placed in a stack and are deleted one at a time, what is the order of removal?

a) ABCD

DCBA

c) DCAB

d) ABDC

8. Consider you have a stack whose elements in it are as follows.

5 4 3 2 << top

Where the top element is 2.

You need to get the following stack

6 5 4 3 2 << top

The operations that needed to be performed are (You can perform only push and pop):

Push(pop()), push(6), push(pop())

b) Push(pop()), push(6)

c) Push(pop()), push(pop()), push(6)

d) Push(6)

QUEUE

QUEUE - INTRODUCTION

- A QUEUE is logically a First In First Out (FIFO) linear data structure.
- It is a homogeneous collection of elements in which new elements are added at one end called **rear,** and the existing elements are deleted from other end called **front.**

QUEUE AS AN ADT/ OPERATION ON QUEUE

- There are two basic operations/ primitive operation that can be performed on queue.

Enqueue ():

- It refers to the addition of an item in the queue.
- Items are always inserted at the **rear** end of queue
- Whenever we insert a data items the value of rear is increased by 1 i.e. rear = rear+1

Dequeue ():

- It refers to the deletion of an item from the queue
- Item are always deleted from the **front** end of queue
- Whenever an item is deleted from the queue the value of **front** is increased by 1 i.e. front = front+1

QUEUE AS AN ADT/ OPERATION ON QUEUE

- However, some more additional operations that can be performed on queue are:
- **Make Empty (Q) :** Create an empty queue, Q
- **Isempty (Q):** Returns true if the queue, Q, is empty otherwise false.
- **Isfull (Q) :** Returns true if the queue, Q, is full otherwise false.
- **Size (Q) :** Returns the number of items in the queue, Q
- **Front (Q) :** Return the object that is at the front of the queue without removing it.
- **Traverse (Q) :** Visit all the elements stored in the queue, Q
- **Search (K,Q) :** Search for the location of K in queue, Q

IMPLEMENTATION OF QUEUE

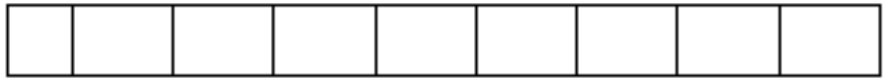
- Static Implementation (Array Implementation)
- Dynamic Implementation (Linked List Implementation)

TYPES OF QUEUE

- Linear Queue or Simple Queue
- Circular Queue
- Double ended Queue (De-Queue)
- Priority Queue : Priority queue is generally implemented using linked list.

LINEAR QUEUE

- **Enqueue** operation will insert an element to queue, at the rear end, by incrementing the array index.
- **Dequeue** operation will delete from the front end by incrementing the array index and will assign the deleted value to a variable.
- Initially front and rear is set to -1.
- The queue is empty whenever **rear < front** or both the rear and front is equal to -1.
- Total number of elements in the queue at any time is equal to **rear-front+1,** when implemented using arrays.
- Below are the few operations in the queue.



Queue is empty

rear=-1,front=-1

LINEAR QUEUE

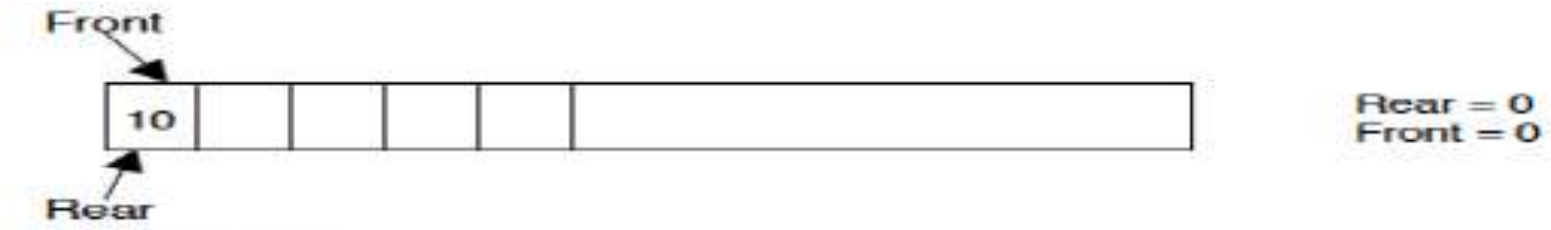


Fig: Enqueue(10)

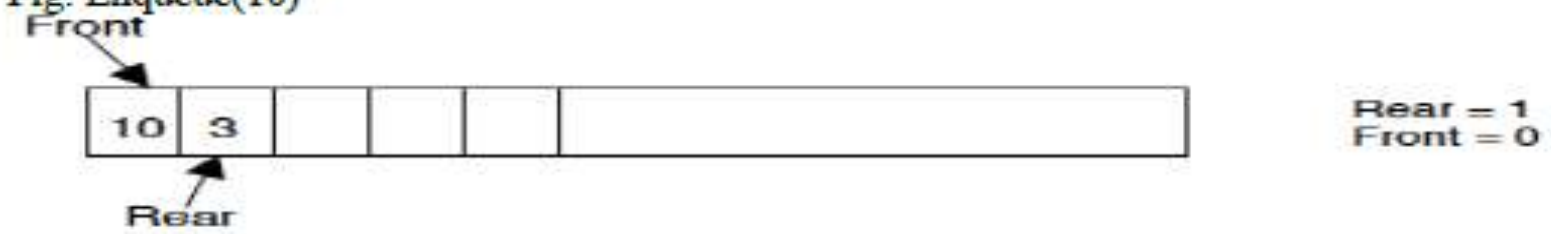


Fig: Enqueue(3)

Fig: Enqueue(41)

LINEAR QUEUE

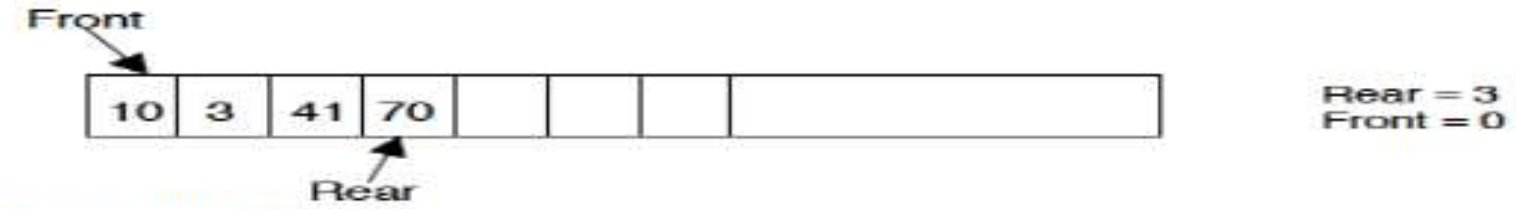


Fig: Enqueue(70)

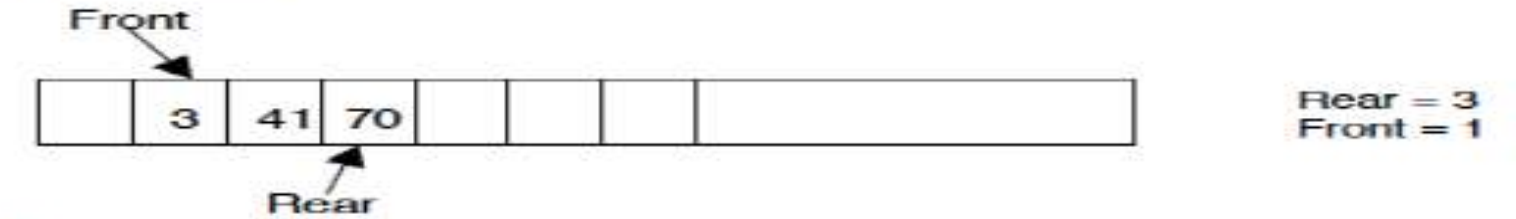


Fig: x = Dequeue()[i.e. x = 10]

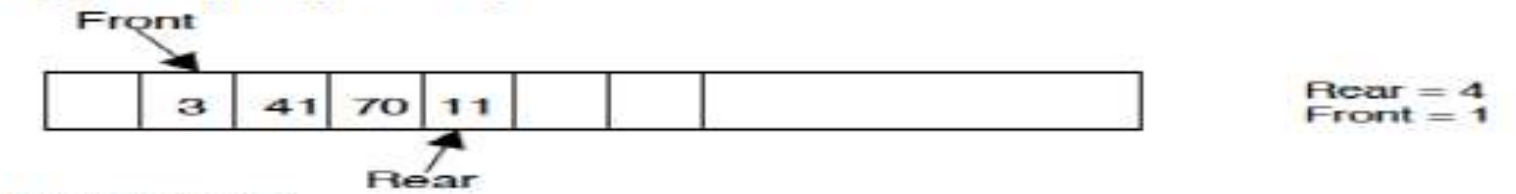


Fig: Enqueue(11)

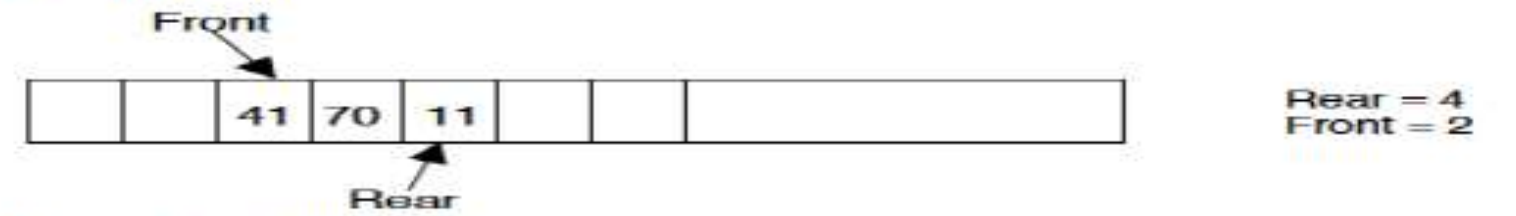


Fig: x = Dequeue()[i.e. x = 3]

LINEAR QUEUE

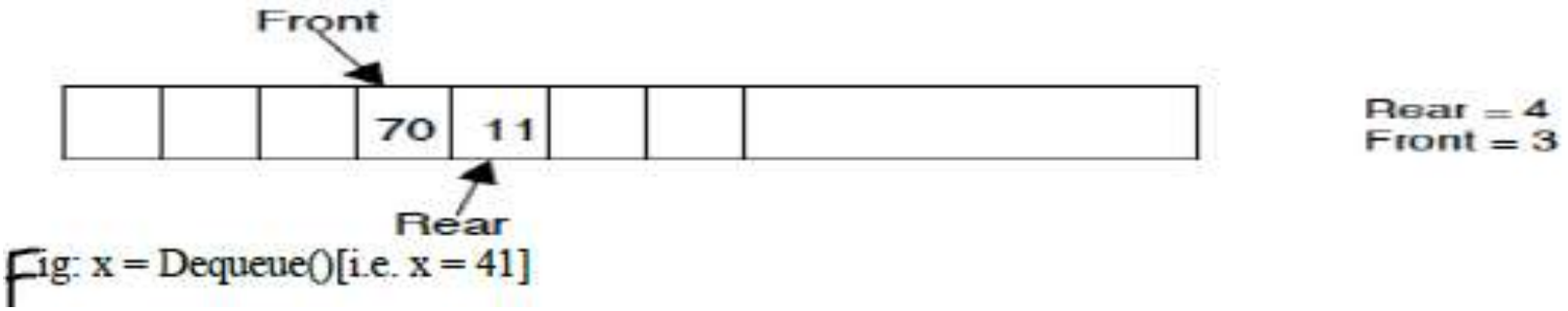


Fig: x = Dequeue()[i.e. x = 41]

- ***Note: During The insertion of first element in the queue, we always increment the front by one.***
- If we try to dequeue an element from queue when it is empty, underflow occurs.
- If we try to enqueue an element to queue , overflow occurs when the queue is full.

Overflow Condition: If Rear = MAX-1
Underflow Condition: If front = -1 and rear = -1 (Initial Condition) or rear < front
One Element: If rear = front.
Number of Elements present in a Queue : rear – front +1

LINEAR QUEUE- ALGORITHM FOR QUEUE OPERATIONS

- Let Q be the arrays of some specified size say MAX. **rear** and **front** are two points for element insertion and deletion.

Inserting an element into QUEUE (Enqueue)

1. If (rear>= MAX-1)
   a. Display “Queue Overflow”
   b. Exit
2. Else
   a. If (front == -1 && rear ==-1) [first time insertion]
   b. Front = 0
3. rear =rear +1
4. Input the value to be inserted and assign to variable “data”.
5. Q[rear] = data
6. Exit

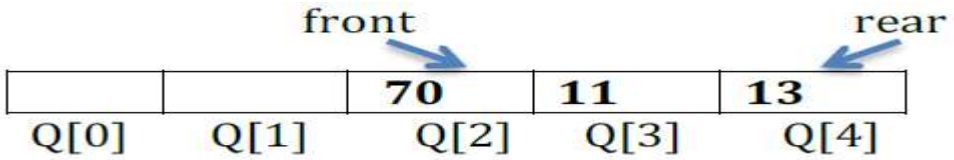
LINEAR QUEUE- ALGORITHM FOR QUEUE OPERATIONS

Deleting an element from QUEUE (Dequeue)

1. If (rear < front or (front == -1 && rear == -1))
   a. Display “Queue is empty”
   b. Exit
2. Else
   a. Data = Q [front]
3. If (front ==rear)
   a. front = -1
   b. rear = -1
4. Else
   a. front = front +1
5. Exit

CIRCULAR QUEUE

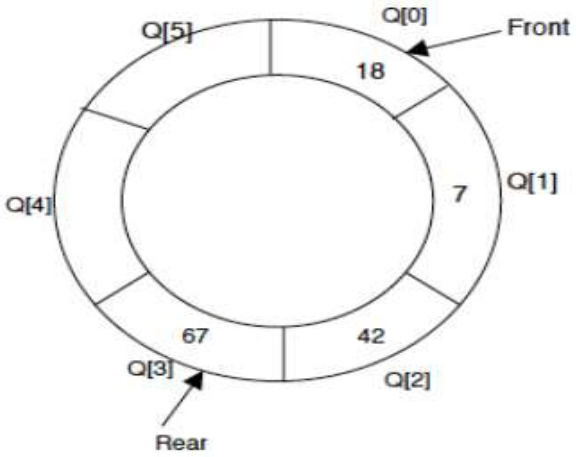
- Suppose a queue has maximum size 5, say 5 elements pushed and 2 elements popped.



- Now if we attempt to add more elements, even though 2 queue cells are free, the elements cannot be pushed.
- Because in a queue, elements are always inserted at the rear end and hence rear points to last location of the queue which indicates queue full.
- This limitation can be overcome if we use circular queue.
- In circular queues the elements Q[0], Q[1], Q[2],……, Q[n-1] is represented in a circular fashion.
- A circular queue is one in which the insertion of a new element is done at the very first location of the queue if the last location at the queue is full.

CIRCULAR QUEUE

- Suppose Q is a queue array of 6 elements. Enqueue() and Dequeue() operation can be performed on circular. The following figure will illustrate the same.



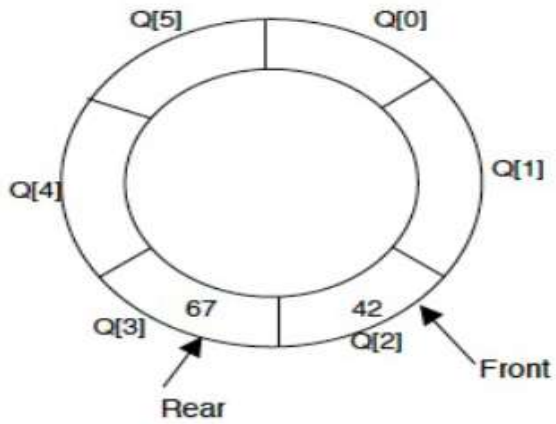


Fig1: A Circular Queue After inserting 18,7,42,67. **Fig2**: Circular Queue after popping 18,7.

- After inserting an element at last location Q[5], the next element will be inserted at the very first location (i.e., Q[0]) that is circular queue is one in which the first element comes just after the last element.

CIRCULAR QUEUE

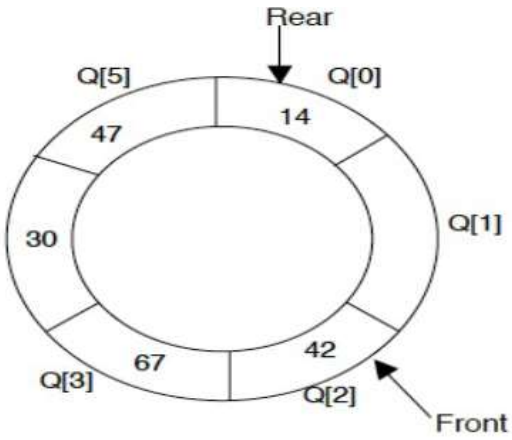


Fig3: Circular Queue after pushing 30, 47, 14

- At any time the relation will calculate the position of the element to be inserted.
  - rear = (rear+1) % MAX [MAX = size]
- After deleting an element from circular queue the position of the front end is calculated by the relation
  - front = (front + 1) % MAX

ALGORITHM FOR CIRCULAR QUEUE

- Let Q be the arrays of some specified size say **MAX**. **front** and **rear** are two pointers where the elements are deleted and inserted. **DATA** is the element to be inserted. Initially front==-1 and rear==-1.

Inserting an element to circular queue:

1. if ((front ==0 && rear ==MAX-1) OR front =rear+1)
   a. Display " Queue is Full"
   b. Exit
2. If (front == -1 && rear == -1)
   a. front =0
   b. rear = 0
3. else
   a. Rear = (rear+1) % MAX
4. Input the value to be inserted and assign to variable "DATA"
5. Q[rear] = DATA
6. Repeat steps 2 t0 5 if we want to insert more elements
7. Exit.

ALGORITHM FOR CIRCULAR QUEUE

Deleting an element from a circular queue:

1.if (front ==-1 && rear ==-1)

   a. Display " Queue is Empty"

   b. Exit

2.Else

   a. DATA = Q[front]

3.If (rear==front)

   a. front = -1

   b. rear = -1

4.Else

   a. front = (front + 1) % MAX

5.Repeat steps 1 t0 4 if we want to delete more elements

6.Exit.

DEQUES

- A deque is a homogeneous list in which elements can be added or inserted (called enqueue operation) and deleted or removed from both the ends (which is called dequeue operation).
- That is, we can add a new element at the rear or front end and also we can remove an element from both front and rear end.
- Hence, it is called double ended Queue.

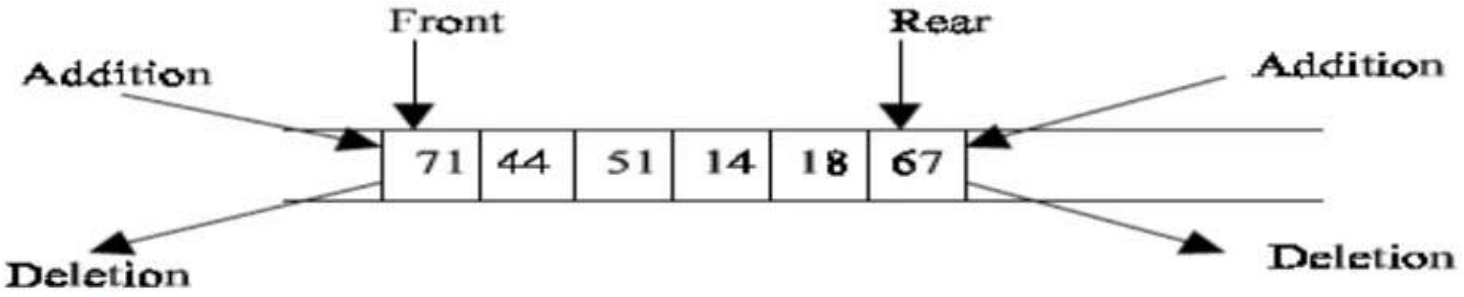


Figure 2: A Deque

DEQUES

- There are two types of deque depending upon the restriction to perform insertion or deletion operations at the two ends. They are:

Input restricted deque:

- An input restricted deque is a deque, which allows insertion at only one end, rear end, but allows deletion at both ends, rear and front end of the lists.

Output restricted deque:

- An output restricted deque is a deque, which allows deletion at only one end, front end, but allows insertion at both ends, rear and front end of the lists.

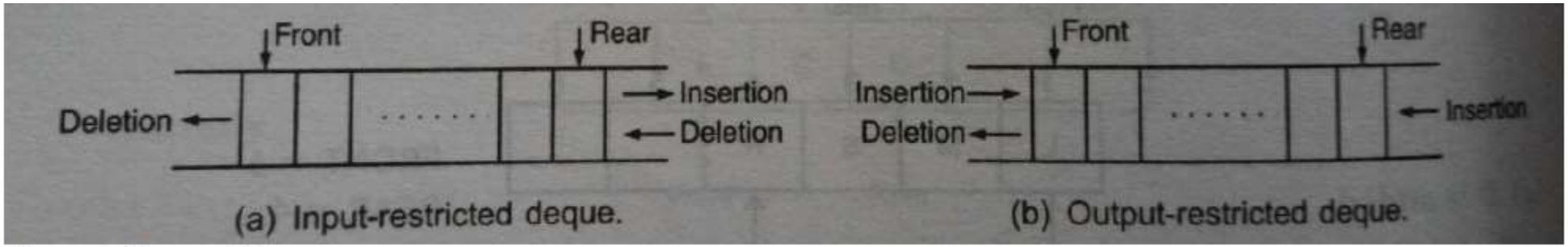


Figure 3: Types of Deques:

DEQUES

- The possible operation performed on deque is : (Deque ADT)
  - Add an element at the rear end (insert_rear)
  - Add an element at the front end (insert_front)
  - Delete an element from the front end (delete_front)
  - Delete an element from the rear end (delete_rear)
- Only 1st, 3rd and 4th operations are performed by input-restricted deque and 1st, 2nd and 3rd operations are performed by output – restricted deque.

ALGORITHM FOR INSERTING AN ELEMENT IN DEQUES

- Let Q be the queue of size **MAX** . **front** and **rear** are two pointers where the addition and deletion of elements occurred. Let **DATA** be the element to be inserted. Initially front == -1 and rear == -1.

Insert an element at the rear end of the deque:

1.If ((front ==0 && rear==MAX-1) OR (front==rear+1)

   a. Display “Queue Full”

   b. Exit

2.If (front == -1 && rear == -1)

   a. Front =0

   b. Rear =0

3.Else

   a. Rear=(Rear+1)%MAX

4.Input DATA to be inserted

5.Q[rear] = DATA

6.Exit

ALGORITHM FOR INSERTING AN ELEMENT IN DEQUES

Insert an element at the front end of the deque:

1. If ((front ==0 && rear==MAX-1) OR (front==rear+1))
   a. Display “Queue Full”
   b. Exit
2. If (front == -1 && rear == -1)
   a. Front =0
   b. Rear =0
3. Else
   a. If (front ==0)
      i. front=MAX-1
   b. else
      i. front=front-1
4. Input DATA to be inserted
5. Q[front] = DATA
6. Exit

ALGORITHM FOR DELETING AN ELEMENT IN DEQUES

- Let Q be the queue of size **MAX** . **front** and **rear** are two pointers where the addition and deletion of elements occurred. Let **DATA** will contain the element just deleted. Initially front == -1 and rear == -1.

Delete an element from the rear end of the deque:

1. If (front == -1 && rear == -1)
   a. Display “Queue Underflow”
   b. Exit
2. DATA = Q[rear]
3. If (front == rear)
   a. Front = -1
   b. Rear = -1
4. If (rear ==0)
   a. rear = MAX-1
5. Else
   a. rear = rear-1
6. Exit

ALGORITHM FOR DELETING AN ELEMENT IN DEQUES

Delete an element from the front end of the deque:

1. If (front == -1 && rear == -1)
   a. Display “Queue Underflow”
   b. Exit
2. DATA = Q[front]
3. If (front == rear)
   a. Front = -1
   b. Rear = -1
4. Else
   1. Front=(front+1)%MAX
5. Exit

PRIORITY QUEUES

- Priority queue is a queue where each element is assigned a priority.
- In priority queue, the elements are deleted and processed by following rules.
  - An element of higher priority is processed before any element of lower priority
  - Two elements with the same priority are processed according to the order in which they were inserted to the queue.
- For example, Consider a manager who is in process of checking and approving files in a first come first basis. In between, if any urgent file (with a high priority) comes, he will process the urgent file next and continue with the other low urgent files.

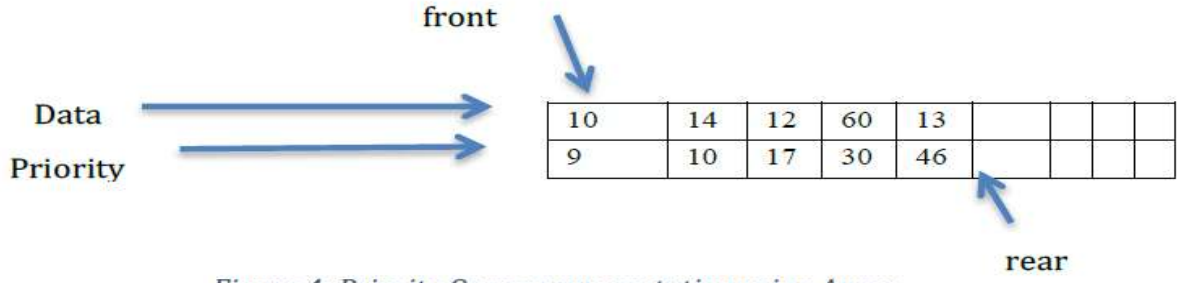


Figure 4: Priority Queue representation using Array

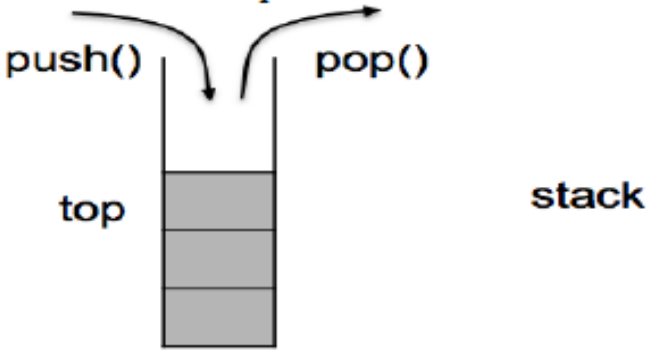
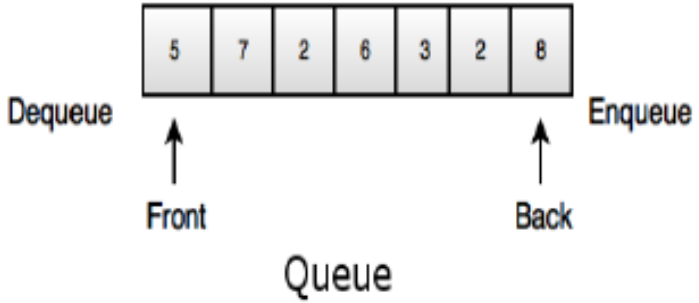
PRIORITY QUEUES

- Above figure gives the pictorial representation of priority queue using arrays after adding 5 elements with its corresponding priorities.
- Here the priorities of data are in ascending order.
- Always we may not be pushing the data in an ascending order.
- From the mixed priority list it is difficult to find the highest priority element if the priority queue is implemented using arrays.
- It is better to implement the priority queue using linked list where a node can be inserted at anywhere in the list.

APPLICATION OF QUEUES

- Round robin techniques for processor, scheduling is implemented using queue.
- Printer server routines (in drivers) are designed using queues.
- All type of customer service type software (e.g. Ticket reservation) are designed using queue to give proper service to the customers.
- When a resource is shared among multiple consumers. Example includes CPU scheduling, Disk Scheduling
- Scheduler (e.g. in operating system): maintains a queue of processes awaiting a slice of machine time
- when data is transferred asynchronously between two processes.

STACK VS QUEUES

SN	Stack	Queue
1	Stack is an ordered list where in all insertions and deletions are performed at the one end called top. 	Queue is an ordered list where in insertions are performed at one end called rear and deletions are performed at another end called front. 
2	Stacks follow Last In First Out (LIFO) order.	Queues following First In First Out (FIFO) order.
3	Stack operations are called push and pop.	Queue operations are called enqueue and dequeue.
4	Associated with stack there is one variable called top.	Associated with queues there are two variables called front and rear.
5	Stack is full can be represented by the condition, Top = MAX-1	Queue is full can be represented by the condition, rear = MAX-1.
6	Stack is empty is represented by the condition, Top = -1	Queue is Empty is represented by the condition, front = -1 and rear = -1
7	To insert an element into the stack top is incremented by 1.	To insert an element into the queue rear is incremented by 1.
8	To delete an element from the stack top is decremented by 1.	To delete an element from the queue front is incremented by 1.
9	Collection of dinner plates at a wedding reception is an example of stack.	People standing in a file to board a bus is an example of queue.

1. A linear list of elements in which deletion can be done from one end (front) and insertion can take place only at the other end (rear) is known as _____

a) Queue

b) Stack

c) Tree

d) Linked list

2. The data structure required for Breadth First Traversal on a graph is?

a) Stack

b) Array

a) Queue

d) Tree

3. A queue follows _____

a) FIFO (First In First Out) principle

b) LIFO (Last In First Out) principle

c) Ordered array

d) Linear tree

4. Circular Queue is also known as _____

a) Ring Buffer

b) Square Buffer

c) Rectangle Buffer

d) Curve Buffer

5. If the elements “A”, “B”, “C” and “D” are placed in a queue and are deleted one at a time, in what order will they be removed?

a) ABCD

b) DCBA

c) DCAB

d) ABDC

6. A data structure in which elements can be inserted or deleted at/from both ends but not in the middle is?

a) Queue

b) Circular queue

c) Deque

d) Priority queue

7. A normal queue, if implemented using an array of size MAX_SIZE, gets full when?

a) Rear = MAX_SIZE – 1

b) Front = (rear + 1)mod MAX_SIZE

c) Front = rear + 1

d) Rear = front

8. After performing these set of operations, what does the final list contain?

```
InsertFront(10);
InsertFront(20);
InsertRear(30);
DeleteFront();
InsertRear(40);
InsertRear(10);
DeleteRear();
InsertRear(15);
display();
```

a) 10 30 10 15

b) 20 30 40 15

c) 20 30 40 10

d) 10 30 40 15

9. Which of the following is not the type of queue?

a) Ordinary queue

b) Single ended queue

c) Circular queue

d) Priority queue

10. In a circular queue, how do you increment the rear end of the queue?

a) rear++

b) (rear+1) % CAPACITY

c) (rear % CAPACITY)+1

d) rear- -

11. What is the term for inserting into a full queue known as?

a) overflow

b) underflow

c) null pointer exception

d) program won’t be compiled

12. What is the time complexity of enqueue operation?

a) O(logn)

b) O(nlogn)

c) O(n)

d) O(1)

13. What does the following Java code do?

```
public Object function()
{
        if(isEmpty())
        return -999;
        else
        {
                Object high;
                high = q[front];
                return high;
        }
}
```

a) Dequeue

b) Enqueue

Return the front element

d) Return the last element

14. What is the need for a circular queue?

effective usage of memory

b) easier computations

c) to delete elements based on priority

d) implement LIFO principle in queues

15. What is the space complexity of a linear queue having n elements?

O(n)

b) O(nlogn)

c) O(logn)

d) O(1)

16. In linked list implementation of queue, if only front pointer is maintained, which of the following operation take worst case linear time?

a) Insertion

b) Deletion

c) To empty a queue

Both Insertion and To empty a queue

17. In linked list implementation of a queue, where does a new element be inserted?

a) At the head of link list

b) At the center position in the link list

At the tail of the link list

d) At any position in the linked list

18. In linked list implementation of a queue, front and rear pointers are tracked. Which of these pointers will change during an insertion into a NONEMPTY queue?

a) Only front pointer

Only rear pointer

c) Both front and rear pointer

d) No pointer will be changed

19. In linked list implementation of a queue, front and rear pointers are tracked. Which of these pointers will change during an insertion into EMPTY queue?

a) Only front pointer

b) Only rear pointer

Both front and rear pointer

d) No pointer will be change

20. In linked list implementation of a queue, from where is the item deleted?

At the head of link list

b) At the center position in the link list

c) At the tail of the link list

d) Node before the tail

21. In linked list implementation of a queue, the important condition for a queue to be empty is?

FRONT is null

b) REAR is null

c) LINK is empty

d) FRONT==REAR-1

22. The essential condition which is checked before insertion in a linked queue is?

a) Underflow

b) Overflow

c) Front value

d) Rear value

23. The essential condition which is checked before deletion in a linked queue is?

a) Underflow

b) Overflow

c) Front value

d) Rear value

24.Which of the following is true about linked list implementation of queue?

a) In push operation, if new nodes are inserted at the beginning of linked list, then in pop operation, nodes must be removed from end

b) In push operation, if new nodes are inserted at the beginning, then in pop operation, nodes must be removed from the beginning

c) In push operation, if new nodes are inserted at the end, then in pop operation, nodes must be removed from end

d) In push operation, if new nodes are inserted at the end, then in pop operation, nodes must be removed from beginning

25. With what data structure can a priority queue be implemented?

a) Array

b) List

c) Heap

d) Tree

26. Which of the following is not an application of priority queue?

a) Huffman codes

b) Interrupt handling in operating system

c) Undo operation in

d) Bayesian spam filter

27. What is the time complexity to insert a node based on key in a priority queue?

a) O(nlogn)

b) O(logn)

c) O(n)

d) $O(n^2)$

29. What is a dequeue?

a) A queue with insert/delete defined for both front and

b) A queue implemented with a doubly linked list

c) A queue implemented with both singly and doubly linked lists

d) A queue with insert/delete defined for front side of the queue

LIST

LINKED LIST

LIST- INTRODUCTION

- A list is a collection of homogeneous set of elements or objects.
- If the size of list is fixed at compile time and do not grow or shrink at runtime then it is called static list however if the size of the list is not fixed at compile time and grow and shrink at runtime then it is called dynamic list.
- A list is said to be empty when it contains no elements
- The number of elements currently stored is called the length of the list.
- The beginning of the list is called head and the end of the list is called the tail.

STATIC IMPLEMENTATION OF LIST

- Static implementation can be implemented using arrays.
- It is very simple method but it has static implementation.
- Once a size is declared, it cannot be change during the program execution.
- It is also not efficient for memory utilization.
- When array is declared, memory allocated is equal to the size of the array.
- The vacant space of array also occupies the memory space.
- In this cases, if we store fewer arguments than declared, the memory is wasted and if more elements are stored than declared, array cannot be expanded.
- It is suitable only when exact numbers of elements are to be stored.

SOME COMMON OPERATIONS PERFORMED ON STATIC LIST

- Creating of an array
- Inserting new element at required position
- Deletion of any element
- Modification of any element
- Traversing of an array
- Merging of arrays

LIST (ARRAY) AS AN ADT

- Let A be an LIST of array implementation and it has n elements then it satisfied the following operations:
  - CREATE (A) : Create an array A
  - INSERT (A,X): Insert an element X into an array A in any location
  - DELETE (A,X) : Delete an element X from an array A
  - MODIFY (A,X,Y) : Modify element X by Y of an array A
  - TRAVERSE (A) : Access all elements of an array A
  - MERGE (A,B) : Merging elements of A and B into a third array C.

  Thus by using a one dimensional array we can perform above operations thus an array acts as an ADT.

INSERTION OF AN ELEMENT IN ONE-DIMENSIONAL ARRAY

- **Insertion at the end of an array:**
  - Providing the memory space allocated for the array enough to accommodate the additional; element can easily do insertion at the end of an array.
- **Insertion at the required position**
  - For inserting the element at required position, element must be moved downwards to new locations to accommodate the new element and keep the order of the elements.
  - For inserting an element into a linear array **insert(a, len, pos, num)** where **a** is a linear array, **len** be total number of elements with an array, **pos** is the position at which number **num** will be inserted.

INSERTION OF AN ELEMENT IN ONE-DIMENSIONAL ARRAY

Algorithm to insert new element in a list:

1.[initialize the value of i] set **i=len-1**

2.Repeat for **i= len-1** down to **pos**

[shift the elements right by 1 position]

Set **a[i+1] = a [i]**

[end of loop]

3.[insert the element at required position]

Set **a[pos] = num**

4.[reset len] set **len=len+1**

5.Display the new list of arrays

6.end

DELETION OF AN ELEMENT FROM ONE-DIMENSIONAL ARRAY

- Deleting an element at the end of an array presents no difficulties, but deleting element somewhere in the middle of the array would require to shift all the elements to fill the space emptied by the deletion of the element, then the element following it were moved left by one location.
- **num** is the item deleted and **len** is the no of element in the array. **pos** is the position from where the data item is deleted.

Algorithm:

1. Set **num = a[pos]**
2. Repeat for **j= pos to len-1**
    a. Shift elements 1 position left
    b. Set **a[j] = a[j+1]**
3. Reset **len=len-1**
4. Display the new list of element of array
5. end

DYNAMIC IMPLEMENTATION OF LIST

- In static implementation of memory allocation, we cannot alter (increase or decrease) the size of an array and the memory allocation is fixed.
- So we have to adopt an alternative strategy to allocate memory only when it is required.
- There is a special data structure called linked list that provides a more flexible storage system and it does not required the use of array.
- The advantage of a list over an array occurs when it is necessary to insert or delete an element in the middle of a group of other elements.

LINKED LIST

- A linked list is a linear collection of specially designed data structure, called **nodes,** linked to one another by means of **pointer.**
- Each node is divided into 2 parts: the first part contains information of the element and the second part contains address of next node in the link list.

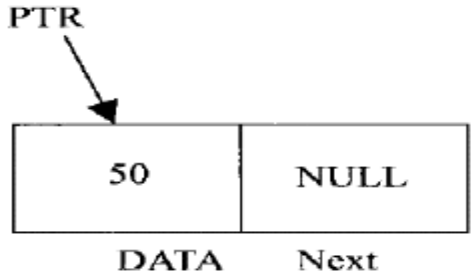


Fig1: Node

Fig2: Linked List

- The left part of each node contains the data items and the right part represents the address of the next node.
- The next pointer of the last node contains a special value, called the NULL pointer, which does not point to any address of the node.

LINKED LIST

- That is NULL pointer indicates the end of linked list.
- START pointer will hold the address of the 1st node in the list

  START = NULL if there is no list (i.e. NULL list or empty list)

Suppose we want to store a list of integer numbers using linked list. Then it can be schematically represented as

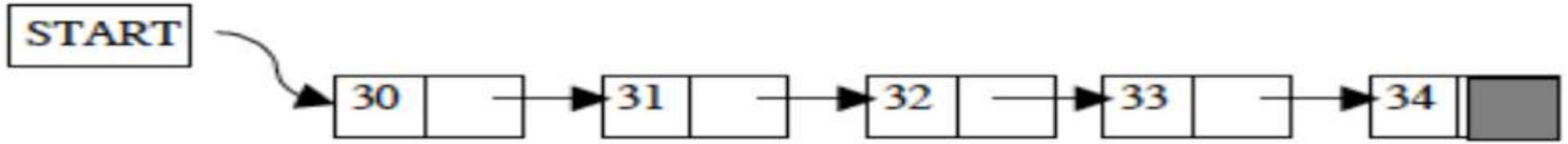


Figure 5: Representation of Linked List

We can declare linear linked list as follows

```
struct Node{
   int data; //instead of data we also use info
   struct Node *Next; //instead of Next we also use link
};
typedef struct Node *NODE;
```

ADVANTAGES AND DISADVANTAGES OF LINKED LIST

ADVANTAGES:

- Link list are dynamic data structure. That is they can grow or shrink during the execution of a program.
- Efficient memory utilization: in linked list memory is not pre-allocated. Memory is allocated whenever it is required. And it is deallocated when it is not needed.
- Insertion and deletion are easier and efficient. Linked list provides flexibility in inserting a data item at a specified position and deletion of a data item from the given position.
- Many complex applications can be easily carried out with linked list.

DISADVANTAGES:

- More memory: to store an integer number, a node with integer data and address field is allocated. That is more memory space is needed.
- Access to any arbitrary data item is little bit cumbersome and also time consuming.

OPERATIONS ON LINKED LIST

The primitive operations performed on linked list is as follows:

- **Creation :**
  - It is used to create a linked list.
  - Once a linked list is created with one node, insertion operation can be used to add more elements in a node.
- **Insertion:**
  - It is used to insert a new node at any specified location in the linked list.
  - A new node may be inserted
    - At the beginning of the linked list
    - At the end of the linked list
    - At any specified position in between in a linked list
- **Deletion:**
  - it is used to delete an item (or node) from the linked list.

OPERATIONS ON LINKED LIST

- A node may ne deleted from the
  - Beginning of a linked list
  - End of a linked list
  - Specified location of the linked list

- **Traversing:**
  - It is the process of going through all the nodes from one end to another end of a linked list.
- **Concatenation:**
  - It is the process appending the second list to the end of the first list.
- **Searching:**
  - It is the process of finding the location of searched item.

TYPES OF LINKED LIST

- Following are the types of Linked list depending upon the arrangements of the nodes.
  - Singly Linked List
  - Doubly Linked List
  - Circular Linked List
    - Circular singly linked list
    - Circular doubly linked list.

SINGLY LINKED LIST

- All the nodes in a singly linked list are arranged sequentially by linking with pointer.
- A singly linked list can grow or shrink, because it is a dynamic data structure.
- The following figure explains the different operations on singly linked list.

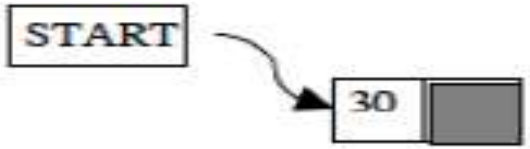


Fig4: Create a node with Data (30)

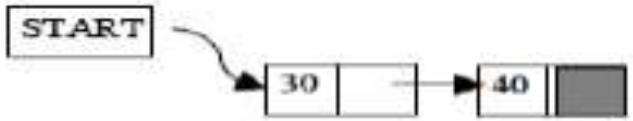


Fig5: Insert a node with Data (40) at the end

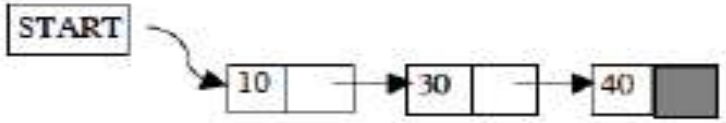


Fig6: Insert a node with Data (10) at the beginning position

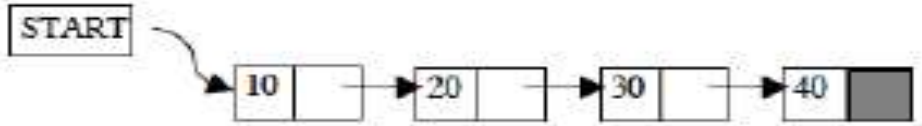


Fig7: Insert a node with Data (20) at 2nd

SINGLY LINKED LIST

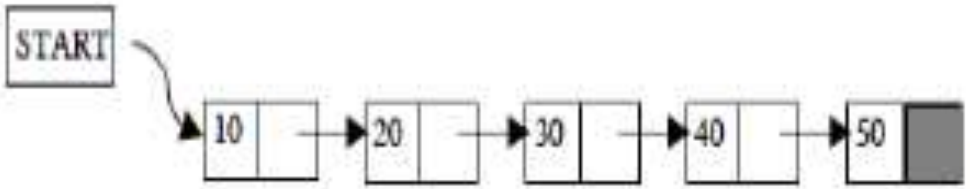


Fig8: Insert a node with Data (50) at the end

Output → 10, 20, 30, 40, 50

Fig9: Traversing the nodes from left to right

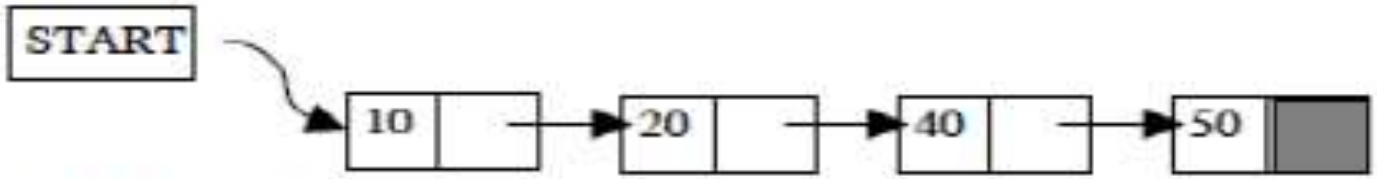


Fig10: Delete 3rd node from the

ALGORITHM FOR INSERTION A NODE IN SLL

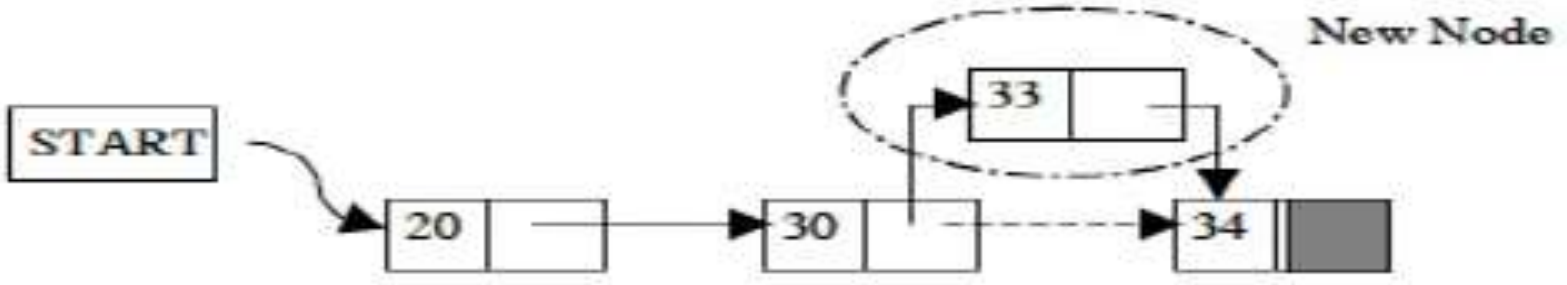


Fig11:

Figure 6: Insertion of New Node at specific position of SLL

- Suppose START is the first position in linked list. Let DATA be the element to be inserted in the new node. POS is the position where the new node is to be inserted. TEMP is a temporary pointer to hold the node address.

ALGORITHM FOR INSERTION A NODE IN SLL

Insert a Node at the beginning of Linked List:

1. Input DATA to be inserted
2. Create a NewNode
3. NewNode→ DATA = DATA
4. If(START == NULL)
   a. NewNode→next = NULL
5. Else
   a. NewNode→next = START
6. START = NewNode
7. Exit

ALGORITHM FOR INSERTION A NODE IN SLL

Insert a Node at the end of Linked List:

1. Input DATA to be inserted
2. Create a NewNode
3. NewNode→ DATA = DATA
4. NewNode→next = NULL
5. If(START == NULL)
   a. START = NewNode
6. Else
   a. TEMP = START
   b. While (TEMP→ next!=NULL)
      i. TEMP=TEMP→next
   c. TEMP→next = NewNode
7. Exit

ALGORITHM FOR INSERTION A NODE IN SLL

Insert a Node at any specified position of Linked List:

1. Input DATA and POS to be inserted
2. Initialize TEMP=START and i=0
3. Repeat the step 3 while (i<POS-1)
   a. TEMP = TEMP→ next
   b. If(TEMP ==NULL)
      a. Display “Node in the list less than the position”
      b. Exit
   c. i=i+1
4. Create a NewNode
5. NewNode→ DATA =DATA
6. NewNode → next=TEMP→next
7. TEMP→next =NewNode
8. Exit

ALGORITHM FOR DISPLAY ALL NODES

- Suppose START is the address of the first node in the linked list.

1. If (START ==NULL)
   a. Display “The list is Empty”
   b. Exit
2. Initialize TEMP = START
3. Repeat the step 4 and 5 until (TEMP==NULL)
4. Display “TEMP→DATA”
5. TEMP=TEMP→next
6. Exit

ALGORITHM FOR DELETING A NODE

- Suppose START is the first position in linked list. Let DATA be the element to be deleted. TEMP, HOLD is a temporary pointer to hold the node address.

Deletion from the beginning:

1. If (START==NULL)
   a. Display “List is empty”
   b. exit
2. Else
   a. HOLD=START
   b. START=START→next
   c. Display the deleted node as HOLD→DATA
   d. Set free the node HOLD, which is deleted
   e. Exit

ALGORITHM FOR DELETING A NODE

Deletion from the end:

1. If (START==NULL)
   a. Display “List is empty”
   b. exit
2. Else IF (START→NEXT ==NULL)
   a. HOLD=START
   b. START =NULL
   c. Display the deleted node as HOLD→DATA
   d. Set free the node HOLD, which is deleted
   e. Exit

a. ELSE
   a. HOLD = START
   b. WHILE(HOLD→next!=NULL)
      a. TEMP=HOLD
      b. HOLD=HOLD→next
   c. TEMP→NEXT=NULL
   d. Display the deleted node as HOLD→DATA
   e. Set free the node HOLD, which is deleted
   f. Exit

ALGORITHM FOR DELETING A NODE

Deletion from the given position:

1. Input the position POS from where data is to be deleted
2. Set i=0
3. If (START==NULL)
    a. Display “The list is empty”
    b. Exit
4. Else
    a. If (POS==0)
        i. HOLD = START
        ii. START=START→next
        iii. Display deleted data as HOLD→DATA
        iv. Set free the node HOLD, which is deleted
        v. Exit

ALGORITHM FOR DELETING A NODE

b. Else
    i. HOLD=START
    ii. WHILE(i<POS)
        i. TEMP =HOLD
        ii. HOLD = HOLD→next
        iii. IF(HOLD==NULL)
            i. Display Position not found
            ii. Exit
        iv. i=i+1
    iii. TEMP→next=HOLD→next
    iv. Display deleted data as HOLD→DATA
    v. Set free the node HOLD, which is deleted
    vi. Exit

ALGORITHM FOR SEARCHING A NODE

- Suppose START is the address of the first node in the linked list and DATA is the information to be searched. After searching, if the DATA is found, POS will contain the corresponding position in the list.

1. Input the DATA to be searched
2. Initialize TEMP = START; POS =0
3. Repeat the step 4, 5 and 6 until (TEMP == NULL)
4. If (TEMP → DATA ==DATA)
   a. Display “The data is found at POS”
   b. Exit
5. TEMP = TEMP→next
6. POS = POS+1
7. If (TEMP == NULL)
   a. Display “The data is not found in the list”
8. Exit

STACK USING LINKED LIST

- The following figures shows that the implementation of stack using linked list.

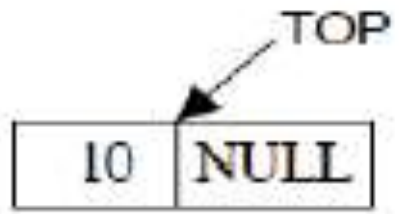


Fig13: Push (10)

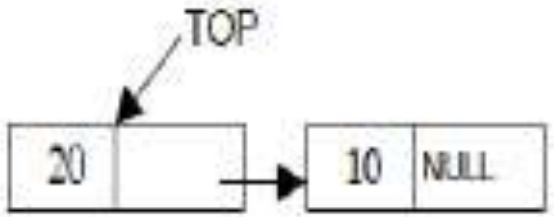


Fig14: Push (20)

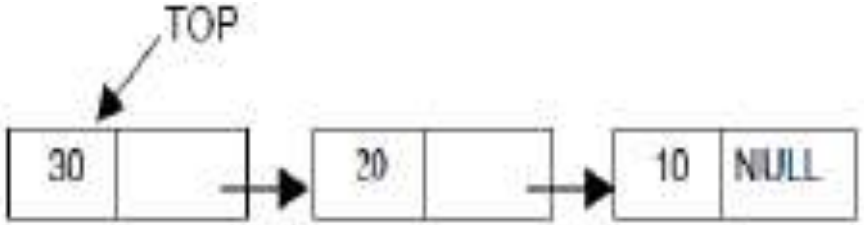


Fig15: Push (30)

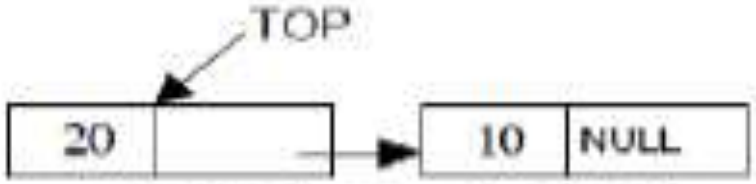


Fig16: X = Pop (i.e. X=30)

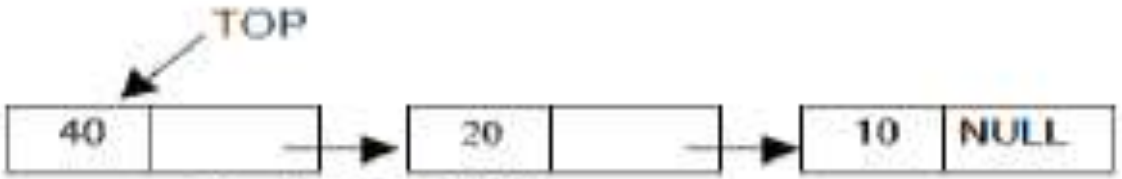


Fig17: Push (40)

STACK USING LINKED LIST

Algorithm for PUSH operation:

- Suppose TOP is a pointer, which is pointing towards the topmost element of the stack. TOP is NULL when the stack is empty. DATA is the data item to be pushed.

1. Input the DATA to be pushed
2. Create a new Node
3. NewNode→DATA=DATA
4. NewNode→next = TOP
5. TOP=NewNOde
6. exit

STACK USING LINKED LIST

Algorithm for POP operation:

- Suppose TOP is a pointer, which is pointing towards the topmost element of the stack. TOP is NULL when the stack is empty. TEMP is pointer variable to hold any nodes address. DATA is the information on the node which is just deleted.

1. If (TOP ==NULL)
    a. Display “The stack is empty”
2. Else
    a. TEMP = TOP
    b. Disply “The popped elemet TOP→DATA”
    c. TOP = TEMP→Next
    d. TEMP → next = NULL
    e. Free the TEMP node
3. Exit

QUEUE USING LINKED LIST

The following figure shows that the implementation issues of Queue using linked list.

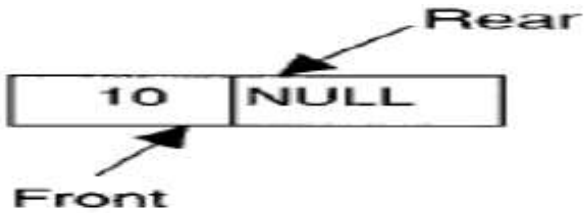


Fig18: Enqueue (10)

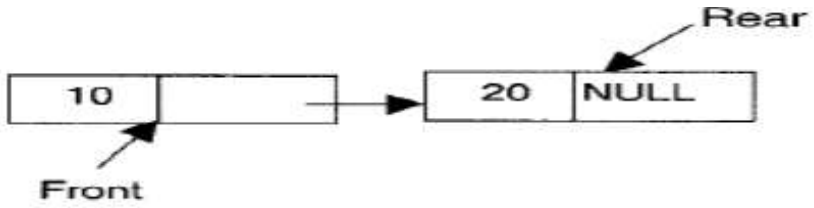


Fig19: Enqueue (20)

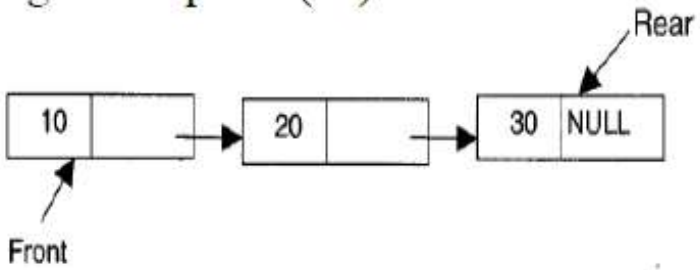


Fig20: Enqueue (30)

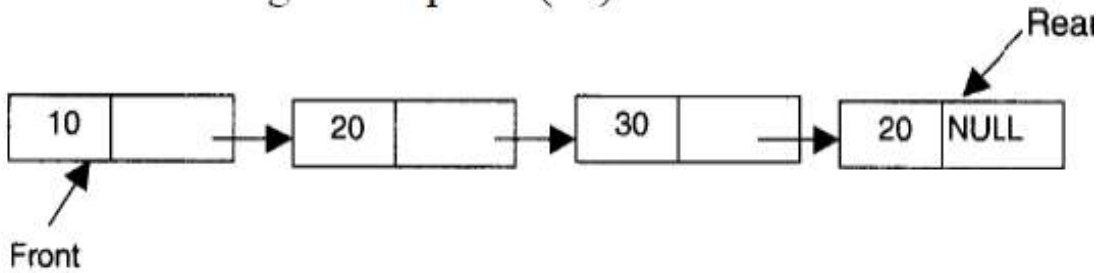


Fig21: Enqueue (20)

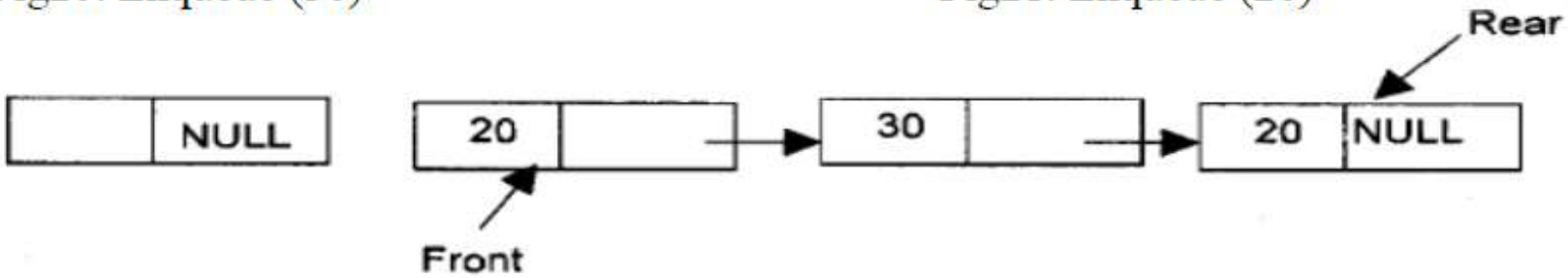


Fig22: X = Dequeue (i.e. X = 10)

ALGORITH FOR ENQUEUE AN ELEMENT INTO A QUEUE

- REAR is a pointer in queue where the new elements are added. FRONT is a pointer, which is pointing to the queue where the elements are popped. DATA is an element to be pushed.

1. Input the DATA element to be pushed
2. Create a New Node
3. NewNode → DATA = DATA
4. NewNode → next = NULL
5. If (REAR == NULL)
    a. REAR = NewNode
    b. FRONT = NewNode
6. Else, REAR→next = NewNOde
7. Rear=NewNode
8. Exit

ALGORITH FOR DEQUEUE AN ELEMENT FROM A QUEUE

- REAR is a pointer in queue where the new elements are added. FRONT is a pointer, which is pointing to the queue where the elements are popped. DATA is an element popped from the queue.

1. If (FRONT == NULL)
    a. Display “The queue is empty”
2. Else
    a. Display “ the popped element is FRONT → DATA”
    b. If (FRONT != REAR)
        i. FRONT = FRONT → Next
    c. Else,
        i. Front = NULL
3. exit

ADVANTAGE AND DISADVANTAGE OF SINGLY LINKED LIST

Advantages:

- Accessibility of a node in the forward direction is easier
- Insertion and deletion of node are easier

Disadvantages:

- Can insert only after a referenced node
- Removing node requires pointer to previous node
- Can traverse list only in the forward direction.

DOUBLY LINKED LIST

- A doubly linked list is one in which all nodes are linked together by multiple links which help in accessing both the successor (next) and predecessor (previous) node for any arbitrary node within the list.
- Every nodes in the doubly linked list has three fields: LeftPointer (Prev), RightPointer (Next) and DATA
- **Prev** will point to the node in the left side i.e. **Prev** will hold the address of the previous node.
- **Next** will point to the node in the right side i.e. **Next** will hold the address of next node.
- **DATA** will store the information of the node.

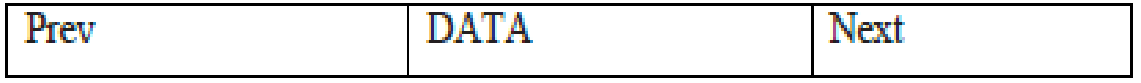


Fig23: Typical Doubly Linked list node

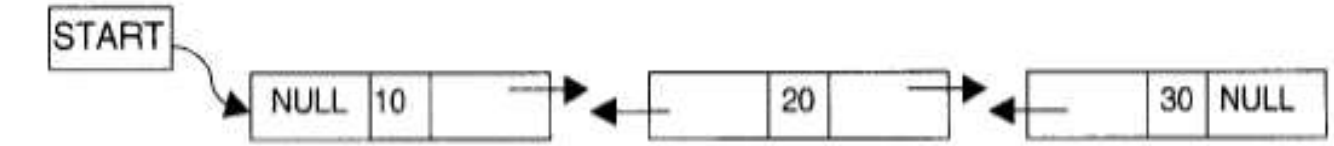


Figure 8: Representation of Doubly Linked List (DLL)

REPRESENTATION OF DOUBLY LINKED LIST

Following declaration can represent doubly linked list

```
struct Node
{
    int data;
    struct Node *Next;
    struct Node *Prev;
};
typedef struct Node *NODE;
```

All the primitive operations performed on singly linked list can also be performed on doubly linked list. The following figures shows that the insertion and deletion of nodes.

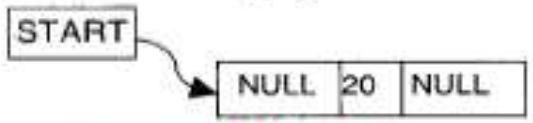


Fig25: Add (20)

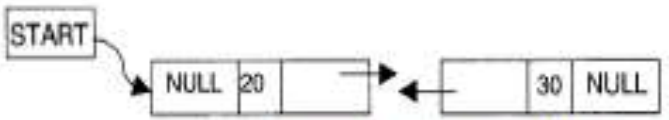


Fig26: Insert (30) at the end

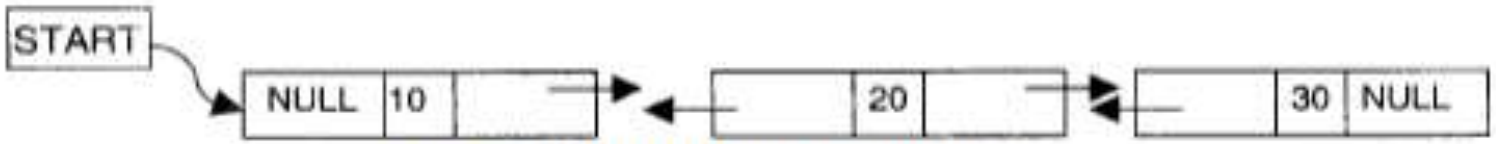


Fig27: Insert (10) at the Beginning

Fig28: Delete a node at the 2nd position (Delete 20 at 2nd position)

ALGORITHM FOR INSERTING A NODE INTO DLL

- Suppose START is the first position in linked list. Let DATA be the element to be inserted in the new node. TEMP is a temporary pointer to hold the node address and POS is the position where the NewNode is to be inserted.

At the beginning:

1.Input DATA element to be inserted

2.Create NewNode

3.NewNode → DATA =DATA

4.NewNode→prev = NULL

5.If (SART == NULL)

a. NewNode → next = NULL

6.Else

a. NewNode→next = START

b. START→prev = NewNode

7.Start=NewNode

8.exit

ALGORITHM FOR INSERTING A NODE INTO DLL

At the end:

1. Input DATA element to be inserted
2. Create NewNode
3. NewNode → DATA =DATA
4. NewNOde→next = NULL
5. If (SART == NULL)
   a. NewNode→prev = NULL
   b. START=NewNode
6. Else
   a. TEMP = START
   b. While (TEMP→next != NULL)
      a. TEMP = TEMP → next
   c. NewNode→ prev = TEMP
   d. TEMP→ next= NewNode
7. exit

ALGORITHM FOR INSERTING A NODE INTO DLL

At the specified location:

1. Input DATA and POS
2. Create NewNode
3. NewNode → DATA =DATA
4. TEMP = START; i=1
5. while((i< POS-1) && (TEMP!=NULL))
   a. TEMP = TEMP→ next ; i=i+1
6. If ((TEMP !=NULL) && (i==POS-1))
   a. NewNode→ prev = TEMP
   b. NewNOde→next = TEMP→next
   c. (TEMP→ next)→prev = NewNode
   d. TEMP→next = NewNode
7. Else
   a. Display “Position not found”
8. exit

ALGORITHM FOR DELETING A NODE FROM DLL

- Suppose START is the address of the first node in the linked list. Let DATA is the number to be deleted. TEMP and PTR is the temporary pointer to hold the address of the node.

Deletion From the Beginning:

1. IF (START==NULL)
    1. Display “List is empty”
    2. Exit
2. Else,
    a. TEMP = START
    b. START = START→ Next
    c. START → Prev = NULL
    d. Display the deleted data as TEMP→DATA
    e. Free the TEMP
    f. Exit

ALGORITHM FOR DELETING A NODE

Deletion from the end:

1. If (START==NULL)
   a. Display “List is empty”
   b. exit
2. Else IF (START→NEXT ==NULL)
   a. HOLD=START
   b. START =NULL
   c. Display the deleted node as HOLD→DATA
   d. Set free the node HOLD, which is deleted
   e. Exit
3. ELSE
   a. HOLD = START
   b. WHILE(HOLD→next!=NULL)
      a. TEMP=HOLD
      b. HOLD=HOLD→next
   c. TEMP→next=NULL
   d. HOLD→prev=NULL
   e. Display the deleted node as HOLD→DATA
   f. Set free the node HOLD, which is deleted
   g. Exit

ALGORITHM FOR DELETING A NODE

Deletion from the given position:

1. Input the position POS from where data is to be deleted
2. Set i=0
3. If (START==NULL)
    a. Display “The list is empty”
    b. Exit
4. Else
    a. If (POS==0)
        i. HOLD = START
        ii. START=START→next
        iii. START→prev =NULL
        iv. HOLD→next =NULL
        v. Display deleted data as HOLD→DATA
        vi. Set free the node HOLD, which is deleted
        vii. Exit

ALGORITHM FOR DELETING A NODE

b. Else
   i. HOLD=START
   ii. WHILE(i<POS)
      i. TEMP =HOLD
      ii. HOLD = HOLD→next
      iii. IF(HOLD==NULL)
         i. Display Position not found
         ii. Exit
      iv. i=i+1
   iii. TEMP→next=HOLD→next
   iv. (HOLD→next)→prev=TEMP
   v. Display deleted data as HOLD→DATA
   vi. HOLD→next =NULL
   vii. HOLD→prev=NULL
   viii. Set free the node HOLD, which is deleted
   ix. Exit

CIRCULAR LINKED LIST

- A circular linked list is one, which has no beginning and no end.
- A singly linked list can be made a circular linked list by simply storing the address of the very first node in the linked field of the last node.
- Circular linked lists also make our implementation easier, because they eliminate the boundary conditions associated with the beginning and end of the list, thus eliminating the special case code required to handle these boundary conditions.

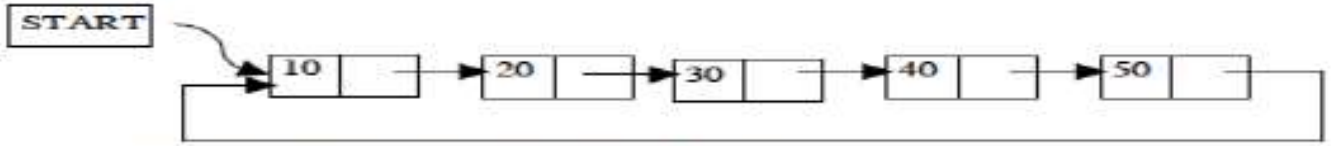


Figure 11: Representation of Singly Circular Linked List (SCLL)

A circular doubly linked list has both the successor pointer and predecessor pointer in circular manner as shown in the following Fig.

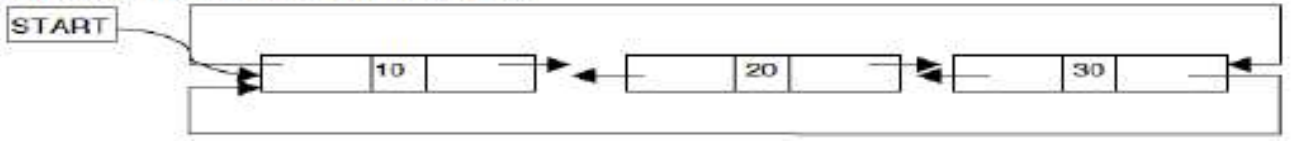


Figure 12: Representation of Doubly Circular Linked List(DCLL)

INSERTION ALGORITHM INTO CIRCULAR LINKED LIST

- Suppose START is the first position in linked list. Let DATA be the element to be inserted in the new node. LAST indicated the last node.

At the beginning:

1. Create a New Node
2. NewNOde→DATA = DATA
3. If (START == NULL)
   a. NewNode→next = NewNode
   b. START = NewNode
   c. LAST = NewNode
4. Else
   a. NewNode → next = START
   b. START = NewNode
   c. LAST → next = NewNode
5. Exit

INSERTION ALGORITHM INTO CIRCULAR LINKED LIST

At the end:

1. Create a New Node
2. NewNOde→DATA = DATA
3. If (START == NULL)
   a. NewNode→next = NewNode
   b. START = NewNode
   c. LAST = NewNode
4. Else
   a. LAST → next = NewNode
   b. LAST = NewNode
   c. LAST → next = START
5. Exit

DELETION ALGORITHM FROM CIRCULAR LINKED LIST

At the beginning:

1. Declare a temporary node , PTR
2. If (START == NULL)
    a. Display " Empty Circular queue"
    b. Exit
3. PTR = START
4. START = START →next
5. Print , element deleted is PTR→ DATA
6. LAST→next = START
7. Free PTR
8. Exit

DELETION ALGORITHM FROM CIRCULAR LINKED LIST

At the end:

1. Declare a temporary node , PTR
2. If (START == NULL)
   a. Display " Empty Circular queue"
   b. Exit
3. PTR = START
4. While (PTR! = LAST)
   a. PTR1 = PTR
   b. PTR = PTR →next
5. Print , element deleted is PTR→ DATA
6. LAST = PTR1
7. LAST→next = START
8. Free PTR

APPLICATION OF LINKED LIST IN COMPUTER SCIENCE

- Implementation of stacks and queues
- Implementation of graphs : Adjacency list representation of graphs is most popular which is uses linked list to store adjacent vertices.
- Dynamic memory allocation : We use linked list of free blocks.
- Maintaining directory of names
- Performing arithmetic operations on long integers
- Manipulation of polynomials by storing constants in the node of linked list
- representing sparse matrices

APPLICATION OF LINKED LIST IN REAL WORLD

- Image viewer – Previous and next images are linked, hence can be accessed by next and previous button.
- Previous and next page in web browser – We can access previous and next url searched in web browser by pressing back and next button since, they are linked as linked list.
- Music Player – Songs in music player are linked to previous and next song. you can play songs either from starting or ending of the list.

1. A linear collection of data elements where the linear node is given by means of pointer is called?

a) Linked list

b) Node list

c) Primitive list

d) Unordered list

2. Consider an implementation of unsorted singly linked list. Suppose it has its representation with a head pointer only. Given the representation, which of the following operation can be implemented in O(1) time?

```
i) Insertion at the front of the linked list
ii) Insertion at the end of the linked list
iii) Deletion of the front node of the linked list
iv) Deletion of the last node of the linked list
```

a) I and II

b) I and III

c) I, II and III

d) I, II and IV

3. In linked list each node contains a minimum of two fields. One field is data field to store the data second field is?

a) Pointer to character

b) Pointer to integer

c) Pointer to node

d) Node

4. What would be the asymptotic time complexity to add a node at the end of singly linked list, if the pointer is initially pointing to the head of the list?

a) O(1)

b) O(n)

c) θ(n)

d) Both O(n) and θ(n)

5. What would be the asymptotic time complexity to insert an element at the front of the linked list (head is known)?

a) O(1)

b) O(n)

c) O(n2)

d) O(n3)

6. What would be the asymptotic time complexity to find an element in the linked list?

a) O(1)

b) O(n)

c) $O(n^2)$

d) $O(n^4)$

7. What would be the asymptotic time complexity to insert an element at the second position in the linked list?

a) O(1)

b) O(n)

c) $O(n^2)$

d) $O(n^4)$

8. Consider the following definition in c programming language. Which of the following c code is used to create new node?

```
struct node
{
    int data;
    struct node * next;
}
typedef struct node NODE;
NODE *ptr;
```

a) ptr = (NODE*)malloc(sizeof(NODE));

b) ptr = (NODE*)malloc(NODE);

c) ptr = (NODE*)malloc(sizeof(NODE*));

d) ptr = (NODE)malloc(sizeof(NODE));

9. What kind of linked list is best to answer questions like "What is the item at position n?"

a) Singly linked list

b) Doubly linked list

c) Circular linked list

d) Array implementation

10. Linked list is considered as an example of _____ type of memory allocation.

a) Dynamic

b) Static

c) Compile time

d) Heap

11. In Linked List implementation, a node carries information regarding _____

a) Data

b) Link

c) Data and Link

d) Node

12. Linked list data structure offers considerable saving in _____

a) Computational Time

b) Space Utilization

c) Space Utilization and Computational Time

d) Speed Utilization

13. Which of the following points is/are not true about Linked List data structure when it is compared with an array?

a) Arrays have better cache locality that can make them better in terms of performance

b) It is easy to insert and delete elements in Linked List

c) Random access is not allowed in a typical implementation of Linked Lists

d) Access of elements in linked list takes less time than compared to arrays

14. What does the following function do for a given Linked List with first node as head?

```
void fun1(struct node* head)
{
    if(head == NULL)
    return;
    fun1(head->next);
    printf("%d  ", head->data);
}
```

a) Prints all nodes of linked lists

b) Prints all nodes of linked list in reverse order

c) Prints alternate nodes of Linked List

d) Prints alternate nodes in reverse order

15. Given pointer to a node X in a singly linked list. Only one pointer is given, pointer to head node is not given, can we delete the node X from given linked list?

a) Possible if X is not last node

b) Possible if size of linked list is even

c) Possible if size of linked list is odd

d) Possible if X is not first node

16. You are given pointers to first and last nodes of a singly linked list, which of the following operations are dependent on the length of the linked list?

a) Delete the first element

b) Insert a new element as a first element

c) Delete the last element of the list

d) Add a new element at the end of the list

17. In the worst case, the number of comparisons needed to search a singly linked list of length n for a given element is?

a) log2 n

b) n/2

c) log2 n – 1

d) n

18. Which of the following is not a disadvantage to the usage of array?

a) Fixed size

b) There are chances of wastage of memory space if elements inserted in an array are lesser than the allocated size

c) Insertion based on position

d) Accessing elements

19. What is the time complexity of inserting at the end in dynamic arrays?

a) O(1)

b) O(n)

c) O(logn)

d) Either O(1) or O(n)

20. Which of these is not an application of a linked list?

a) To implement file systems

b) For separate chaining in hash-tables

c) To implement non-binary trees

d) Random Access of elements

21. Which of the following is false about a doubly linked list?

a) We can navigate in both the directions

b) It requires more space than a singly linked list

c) The insertion and deletion of a node take a bit longer

d) Implementing a doubly linked list is easier than singly linked list

22. What is the worst case time complexity of inserting a node in a doubly linked list?

a) O(nlogn)

b) O(logn)

O(n)

d) O(1)

23. What differentiates a circular linked list from a normal linked list?

You cannot have the ‘next’ pointer point to null in a circular linked list

b) It is faster to traverse the circular linked list

c) In a circular linked list, each node points to the previous node instead of the next node

d) Head node is known in circular linked list

24. Which of the following application makes use of a circular linked list?

a) Undo operation in a text editor

b) Recursive function calls

Allocating CPU to resources

d) Implement Hash Tables

25. Which of the following is false about a circular linked list?

a) Every node has a successor

Time complexity of inserting a new node at the head of the list is O(1)

c) Time complexity for deleting the last node is O(n)

d) We can traverse the whole circular linked list by starting from any point

RECURSION

INTRODUCTION

- Recursion is the powerful technique to write repeatable logic.
- Recursion is defined by implementing function.
- A function said to be recursive is it calls again and again with reduced input and has a base condition to stop the process.
- The basic idea behind recursion is to break a problem into smaller version of itself and then build up a solution for entire problem.
- A recursive definition consists of two parts:
  - **Identifying Base / Ground/ Anchor case:**
    - It is a terminating condition for the problem while designing a recursive function.
  - **Identifying recursive/ inductive step:**
    - Each time the function call itself, it must be close to the base case.

INTRODUCTION

Example 1:

- Consider F(n) which find the sum of first n natural number. Mathematically the function can be defined as:

  F (n) = 1+2+3+4+5+……..+n

- Now recursive definition of this function can be given as:

  Put n = 1, F (1) = 1

  n = 2, F (2) = 1 + 2 = F(1) +2

  n=3, F(3) = 1+ 2+ 3 = F(2) + 3

  n=N, F(N) = F(N-1) + N

i.e. $F(N)=\begin{cases} 1 & \text{If n=1 (Base Case)} \\ F(N-1)+N & \text{If n>1 (Inductive step)} \end{cases}$

Note: So, using this recursive function we can generate a sequence of numbers 1,3,6,10,15,…. etc .which includes the sum of first 1,2,3,4,5,…. Natural numbers.

INTRODUCTION

Example 2:

- Consider F(n) which find the factorial of number n.
- Now recursive definition of this function can be given as:

Put n=0, F(0) =1

n = 1, F (1) = 1*1 = 1*F(0)

n = 2, F (2) = 2*1 = 2*F(1)

n=3, F(3) = 3*2*1 = 3*F(2)

n=N, F(N) = N*F(N-1)

i.e. $F(N)=\begin{cases} 1 & \text{If n=0 (Base Case)} \\ N * F(N-1) & \text{If n>0 (Inductive step)} \end{cases}$

Note: So, using this recursive function we can generate a sequence of numbers 1,1,2,6,24,…. etc .which includes the FACTORIAL OF 0,1,2,3,4,5,…. .

TYPES OF RECURSION

- **Direct recursion and Indirect Recursion**
  - A recursion is said to be direct if a function calls itself
  - Example:

```
Public void abc()
{
        abc();
}
```

  - A function is said to be indirect recursion if it contains a call to another function which ultimately call it.
  - Example:

```
Public void abc()
{
        xyz();
}
```

```
Public int xyz()
{
        abc();
}
```

TYPES OF RECURSION

- **Tail recursion and Non Tail Recursion**
  - A recursion is said to be tail recursion if there are no pending operation to be performed on return from the recursive call, otherwise it is called non tail recursion.
  - Example:

```
Public int fact(int n)
{
        if (n==0) return 1;
        return (n * fact (n-1));
}
```

Algorithm 1

```
Public int fact(int n, int res)
{
        if (n==0) return res;
        return fact (n-1, n*res);
}
```

Algorithm 2

  - Algorithm 1 is non tail recursion because it has pending operation i.e. multiplication to be performed on the return from each recursive call, but algorithm 2 is tail recursive since no such pending operation needs to be performed from each recursive call.

TYPES OF RECURSION

- **Nested Recursion:**
  - A recursion is said to be nested recursion if a function is not only defined in terms of itself, but also is used as one of the parameters.

```
Public int H(int n)
{
        if (n==0) return 0;
        else if (n>4) return n;
        else return (H(2+H(2n)));
}
```

  - So the above example, the recursive method have recursive call inside recursive function, so it is nested recursion.

If n =2, then H(2) = H(2 + H(4)) = H(2 + 2 + H(8)) = H(2 + 2 + 8) = H(12) =12
If n = 1 then H(1) = H(2 + H(2)) = H(2 + 12) = H(14) = 14

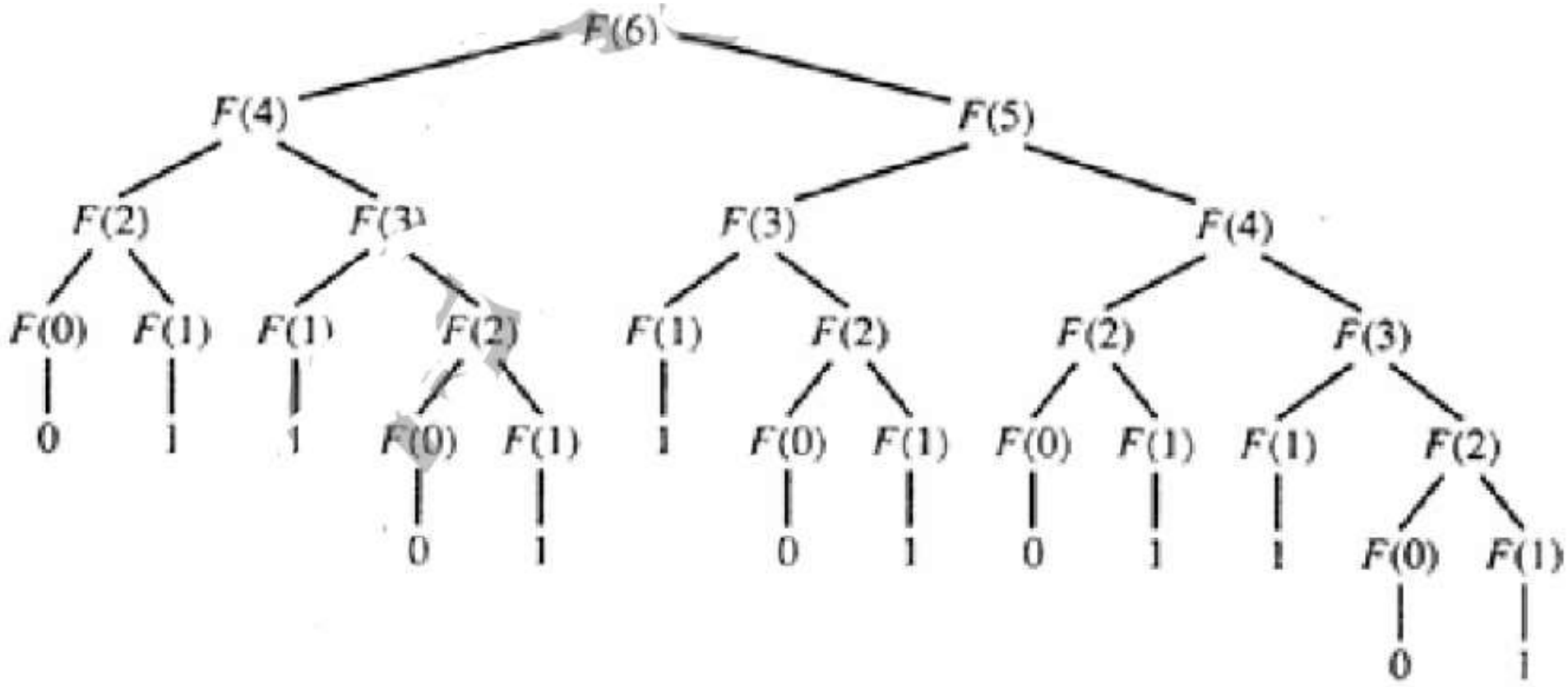
TYPES OF RECURSION

- **Excessive Recursion:**
  - A recursion is said to be excessive recursion if the price for using recursion is slowing down execution time and storing on the run-time stack more things than required in a non –recursive approach.
  - If the recursion is too deep then we may run out of stack.
  - Example: a recursive function for generating sequence of Fibonacci terms.
  - e. F(N)= $\begin{cases} n & \text{If n<2} \\ fib\ (n-2) + fib\ (n-1) & \text{otherwise} \end{cases}$

```
Int fib (int n)
{
        if (n<2) return n;
        else return (fib (n-2) + fib(n-1));
}
```

TYPES OF RECURSION

- **Find the Fib(6):**



TYPES OF RECURSION

- **Find the Fib(6):**

Fib(6) = fib(4) + fib (5)

=fib(2) + fib(3) + fib(5)

=fib(0) + fib(1) + fib(3) +fib(5)

=0+1+fib(3)+ fib(5)

=1 + fib(1) + fib(2) + fib (5)

=1+1+fib(0) + fib(1)+ fib(5)

=2+0+1+fib(5)

=3 + fib(3) + fib(4)

=3 + fib (1) + fib (2) + fib(4)

=3 +1 + fib(2) + fib (4)

=4 + fib (0) + fib (1) + fib (4)

=4+0+1+fib(4)

=5+fib(4)

=5+ fib(2) + fib(3)

=5 + fib(0) + fib(1) + fib(3)

=5 + 0 + 1+fib(3)

=6 + fib(3)

=6+ fib(1) + fib(2)

=6 + 1+ fib(2)

=7+fib (2)

=7+ fib(0) + fib(1)

=7 + 0+1

8

TYPES OF RECURSION

- So for computing 6^{th} term, we have to make call to fib() for 25 times .
- For each call it must make use of some memory resources to make room for the stack frame.
- So, if the recursion is deep, then say fib(1000), then we may run out of memory.
- So it is usually best to develop iterative algorithm while working with large number.

ADVANTAGES AND DISADVANTAGES OF RECURSION

- **Advantages:**
  - We can create simple and easy version of programs using recursion
  - Some specific application are meant for recursion such as binary tree traversal, tower of Hanoi etc.
- **Disadvantages:**
  - It occupies more memory because of implementation of stack
  - It consumes more time to get desired result because they uses calls.
  - The computer may run out of memory if proper precautions are not taken.

APPLICATIONS OF RECURSION

- It is used to calculate factorial of a given number.
- It is used to solve TOH problem
- It is used to translate infix expression into postfix expression
- It is used to check validity of expression
- It is used to calculate term in Fibonacci series.

ITERATIVE VS RECURSION

Iteration	Recursion
1. It is the process of executing a group of statements repeatedly until some specified condition is satisfied.	It is the technique of defining something in term of itself.
2. It uses looping so the steps involved are initialization. Condition, execution and update.	2. The steps involved in recursive procedure are identifying base case and identifying recursive step.
3. Iteration executes very fast and consumes less memory and it can be easy.	Recursion consumes more time to execute and consumes a lots of memory.
4. There are some application in which iteration is not best suited such as TOH, tree traversal as iterative function are difficult to design and take more programming time.	4. There are some applications in which recursion is best suited for designing algorithms such as TOH, tree traversal as recursive function are more efficient and can be understood easily.
5. Any recursive problem can be solved iteratively.	5. Not all problem have recursive solution

ITERATIVE VS RECURSION

Iteration	Recursion

6. Example:
Iterative solution for finding factorial of n number
Int fact (int n)
{
If (n==0) return 1;
Prod =1;
For (int i=n; i>=1; i--)
{
Prod=prod*i;
}
Return prod;
}
	6. Example:
	Recursive solution for finding factorial of n number
	Int fact (int n)
	{
	if (n==0) return 1;
	return (n*fact(n-1));
	}

TOWER OF HANOI

- Tower of Hanoi is a classical problem, which consists of N different sized disc and three towers over which these disc can be mounted.
- The problem of TOH is to move disc from one tower to another with the help of temporary tower.
- If we have three towers A, B and C, all the n disks are mounted on tower A in such a way that a larger disc is always below a smaller one then we have to move disc from tower A to tower C with the help of temporary tower B.
- The condition for playing this game are:

1. We can move only one disc from one tower to another at a time
2. A larger disc can't be placed on the smaller one.
3. One and only one extra tower could be used for temporary storage of discs.

TOWER OF HANOI

- In general, the solution to TOH problem requires 2^N -1 moves of disc, where N is the number of disc.

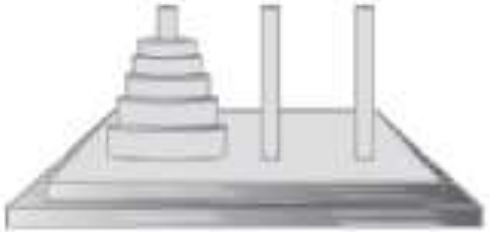
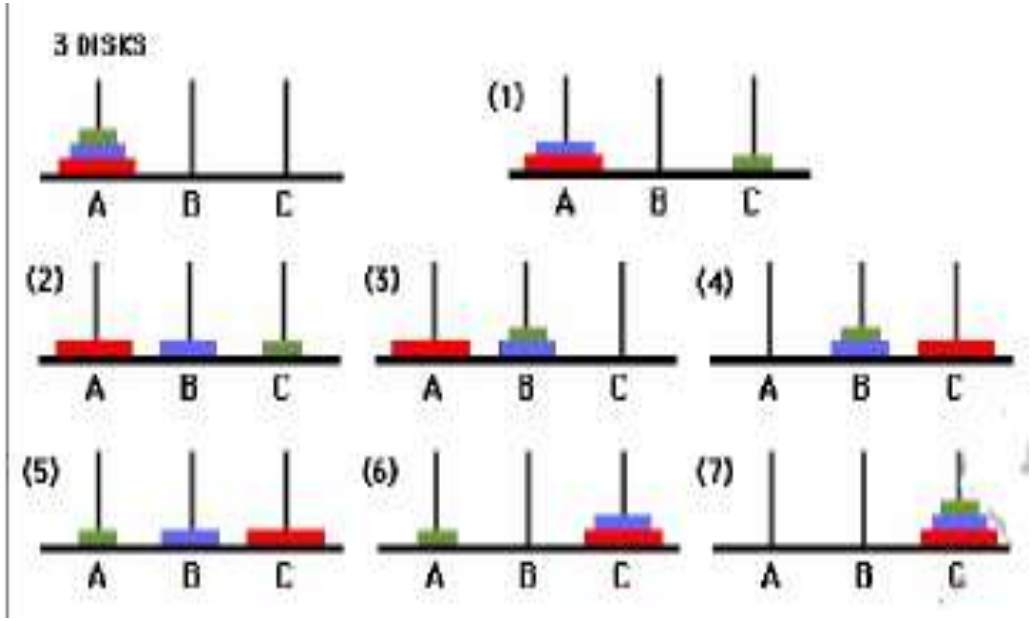


Fig: Initial setup for TOH

- Example: The solution to TOH problem for n=3 is shown in figure below.

So for n=3, the steps to move disk involves:

1. A to C
2. A to B
3. C to B
4. A to C
5. B to A
6. B to C
7. A to C



TOWER OF HANOI- ALGORITHM

- Let us assume that we have N disc and three towers named source, temp and destination. Now the algorithm for solution to TOH problem is

1. if (N==1)
   a. Move a disc from source to destination
   b. Exit
2. Move upper N-1 disc from source to temp using destination as temporary

   Move (N-1, Source, destination, temp)
3. Move largest disc from source to destination
4. Move upper N-1 disc from temp to destination using source as temporary

   Move (N-1, temp, source, destination)
5. Exit

TOWER OF HANOI

Simpler statement of iterative solution:

- Alternating between the smallest and the next smallest disc, follow the steps for appropriate case:

1.For an even number of disc:

- Make the legal move between pegs A and B
- Make the legal move between pegs A and C
- Make the legal move between pegs C and B
- Repeat until complete

2.For an odd number of disc:

- Make the legal move between pegs A and C
- Make the legal move between pegs A and B
- Make the legal move between pegs C and B
- Repeat until complete

VALIDITY OF EXPRESSION

- A statement generally have the following delimiters: parentheses “(“ and “)”, square brackets “[“ and “]”, curly brackets “{“ and “}”, and comment delimiters “/*” and ““*/”.
- A statement is said to be valid for correct matching of delimiters if:
  - The number of left delimiters is equal to right delimiters
  - Each right delimiters is preceded by matching left delimiter.
- Example: {(A+B)-(C*D}} is invalid

  {(A+B)-(C*D)} is valid

ALGORITHM FOR VALIDITY OF EXPRESSION

1. Scan the elements from left to right until the end of statement is encountered.
2. If the scanned element is
    a. (or { or [→ PUSH it onto the stack
    b. / → scan the next character, if this character is * skip all character until */ is found and report an error if the end of statement is reached before */ is reached.
    c.) or } or] → POP the left delimiter from the top of stack
        i. Check if popped left delimiter match the right delimiter, if it doesn't match then display " invalid statement" and exit.
        ii. If during the attempt of POP operation, stack is found to be empty the display "Invalid statement" and exit.
    d. Other character→ ignore them
3. If the scanned element is the end of statement and stack doesn't get empty then display "Invalid statement" and exit.
4. Otherwise, display "Valid statement" and exit

EXAMPLE OF VALIDITY OF EXPRESSION

1. [{(A + B) – (C + D)}]

Scanned Symbol	Stack				
[	[				
{	[	{			
(	[	{	(		
A	[	{	(		
+	[	{	(		
B	[	{	(		
)	[	{			
-	[	{			
(	[	{	(		
C	[	{	(		
+	[	{	(		
D	[	{	(		
)	[	{			
}	[				
]	**Empty**				

Since we have scanned the last element and stack is empty hence it is Valid statement

EXAMPLE OF VALIDITY OF EXPRESSION

2. {(A+B)-(C*D)}]

Since during attempt to pop operation, stack is found to be empty, so it is invalid statement

Practice question:

1. {(A+B)-(C*D})
2. S=T[5] + U/(V*(W+Y));

Scanned symbol	Stack			
{	{			
(	{	(		
A	{	(		
+	{	(		
B	{	(		
)	{			
-	{			
(	{	(		
C	{	(		
*	{	(		
D	{	(		
)	{			
}	Empty			
]				

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

```
#include<stdio.h>
main()
{
    int n;
    n=f1(4);
    printf("%d",n);
}
f1(int x)
{
    int b;
    if(x==1)
        return 1;
    else
        b=x*f1(x-1);
        return b;
}
```

a) 24

b) 4

c) 12

d) 10

2. The data structure used to implement recursive function calls _____

a) Array

b) Linked list

c) Binary tree

d) Stack

3. In the absence of a exit condition in a recursive function, the following error is given _____

a) Compile time error

b) Run time error

c) Logical error

d) No error

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

a) ++++2

b) +++++2

c) +++++

d) 2

```c
#include<stdio.h>
main()
{
    int n,i;
    n=f(6);
    printf("%d",n);
}
f(int x)
{
    if(x==2)
        return 2;
    else
    {
        printf("+");
        f(x-1);
    }
}
```

4

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

a) 10

b) 80

c) 30

d) Error

```c
#include<stdio.h>
int main()
{
    int n=10;
    int f(int n);
    printf("%d",f(n));
}
int f(int n)
{
    if(n>0)
        return(n+f(n-2));
}
```

5

6. The optimal data structure used to solve Tower of Hanoi is _____

a) Tree

b) Heap

c) Priority queue

d) Stack

7. Select the appropriate code for the recursive Tower of Hanoi problem.(n is the number of disks)

```
public void solve(int n, String start, String auxiliary, String end)
{
        if (n == 1)
        {
            System.out.println(start + " -> " + end);
        }
        else
        {
            solve(n - 1, start, end, auxiliary);
            System.out.println(start + " -> " + end);
            solve(n - 1, auxiliary, start, end);
        }
}
```

```
public void solve(int n, String start, String auxiliary, String end)
{
        if (n == 1)
        {
            System.out.println(start + " -> " + end);
        }
        else
        {
            solve(n - 1, start, end, auxiliary);
            System.out.println(start + " -> " + end);
        }
}
```

```
public void solve(int n, String start, String auxiliary, String end)
{
        if (n == 1)
        {
            System.out.println(start + " -> " + end);
        }
        else
        {
            System.out.println(start + " -> " + end);
            solve(n - 1, auxiliary, start, end);
        }
}
```

```
public void solve(int n, String start, String auxiliary, String end)
{
        if (n == 1)
        {
            System.out.println(start + " -> " + end);
        }
        else
        {
            solve(n - 1, auxiliary, start, end);
            System.out.println(start + " -> " + end);
        }
}
```

8. Which among the following is not a palindrome?

a) Madam

b) Dad

c) Malayalam

d) Maadam

9. Which data structure can be used to test a palindrome?

a) Tree

b) Heap

c) Stack

d) Priority queue

10. Recursion is a method in which the solution of a problem depends on _____

a) Larger instances of different problems

b) Larger instances of the same problem

c) Smaller instances of the same problem

d) Smaller instances of different problems

11. Which of the following problems can’t be solved using recursion?

a) Factorial of a number

b) Nth fibonacci number

c) Length of a string

d) Problems without base case

12. Recursion is similar to which of the following?

a) Switch Case

b) Loop

c) If-else

d) if elif else

13. In recursion, the condition for which the function will stop calling itself is _____

a) Best case

b) Worst case

c) Base case

d) There is no such condition

14. Which of the following statements is true?

a) Recursion is always better than iteration

b) Recursion uses more memory compared to iteration

c) Recursion uses less memory compared to iteration

d) Iteration is always better and simpler than recursion

15. What will happen when the below code snippet is executed?

```
void my_recursive_function()
{
   my_recursive_function();
}
int main()
{
   my_recursive_function();
   return 0;
}
```

a) The code will be executed successfully and no output will be generated

b) The code will be executed successfully and random output will be generated

c) The code will show a compile time error

d) The code will run for some time and stop when the stack overflows

16. Suppose the first fibonnaci number is 0 and the second is 1. What is the sixth

a) 5

b) 6

c) 7

d) 8

17. Which of the following is not a fibonnaci number?

a) 8

b) 21

c) 55

d) 14

18. Which of the following option is wrong?

a) Fibonacci number can be calculated by using Dynamic programming

b) Fibonacci number can be calculated by using Recursion method

c) Fibonacci number can be calculated by using Iteration method

d) No method is defined to calculate Fibonacci number

19. Which of the following recurrence relations can be used to find the nth fibonacci number?

a) $F(n) = F(n) + F(n - 1)$

b) $F(n) = F(n) + F(n + 1)$

c) $F(n) = F(n - 1)$

d) $F(n) = F(n - 1) + F(n - 2)$

20. Which of the following recursive formula can be used to find the factorial of a number?

a) fact(n) = n * fact(n)

b) fact(n) = n * fact(n+1)

c) fact(n) = n * fact(n-1)

d) fact(n) = n * fact(1)