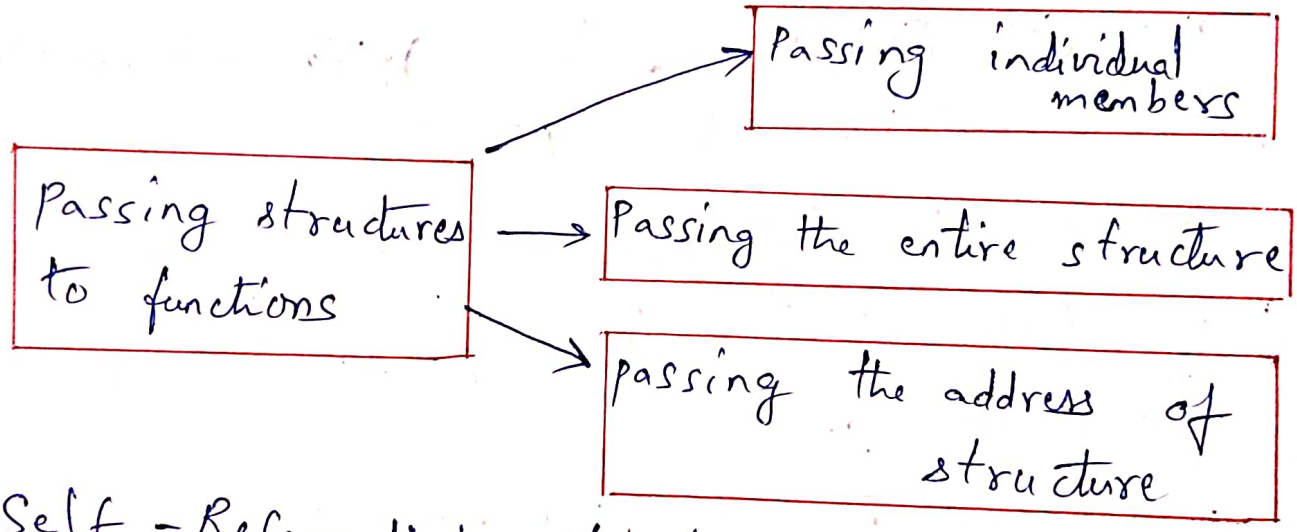
Structures and Functions

→ For structures to be fully useful, we must have a mechanism to pass them to functions and return them.

→ A function may access the members of a structure in three ways as shown below:



Self-Referential Structures:

→ Self-referential structures are those structures that contain a reference to the data of its same type.

→ That is in addition to other data, contains a pointer to a data that is of the same type as that of the structure.

→ For example, consider the structure node given below

```
struct node
{
int val;
struct node *next;
};
```

Here, the structure node will contain two types of data: an integer val and a pointer next.

Actually, self-referential structure is the foundation of the other data structures.

Pointer Basics:

→ A pointer is a variable that stores a memory address.

→ Pointers are used to store the addresses of other variables or memory items.

→ Pointers are useful for another type of parameter passing, usually referred to as "pass by Address".

→ Pointers are essential for dynamic memory allocation.

Declaring pointers:

→ Use the "*" operator.

General syntax:

```
data_type *ptr_name;
```

Example:

```
int *pnum;
char *pch;
```

→ In the above example, pnum is a pointer, and its type will be specified as "pointer to int", because it stores the address of an integer variable.

→ We can also say its type is int*

→ The type is important while pointers are all the same size, as they store a memory address.

```
double *dptr; // a pointer to a double
char *c1; // a pointer to a character
float *fptr; // a pointer to a float.
```

Pointer to structures

→ It holds the address of the entire structure.

→ It is used to create complex data structures such as linked lists, trees, graphs and so on.

→ The members of the structure can be accessed using a special operator called as an "arrow" "→" operator.

Declaration:

```
struct tagname *ptr;
```

Accessing pointers to structures:

```
ptr → membername;
```

Example program for usage of pointers to structures

```
#include <stdio.h>
struct student{
    int sno;
    char sname[30];
    float marks;
    };
main(){
    struct student s;
    struct student *st;
    printf("enter sno, sname, marks:");
    scanf("%d%s%f", &s.sno, &s.sname,
                    &s.marks);
    st = &s;
    printf("details of the student are");
    printf("Number = %d\n", st->sno);
    printf("name = %s\n", st->sname);
    printf("marks = %f\n", st->marks);
    getch();
    }
```

Output:

```
enter sno, sname, marks: 1 Michael 99
details of the student are:
Number = 1
name =  Michael
marks =  99.000000
```

Introduction to Data Structures

→ Data structures is a way of collecting and organising data in such a way that we can perform operations on these data in an effective way.

→ A data structure is basically a group of data elements that are put together under one name, and which defines a particular way of storing and organizing data in a computer so that it can be used efficiently.

Examples of Data structures:

Arrays, linked lists, queues, stacks, binary trees, and hash tables.

→ A data structure can hold different types of data within one single object.

→ The data structures can also be classified on arrangement of data items in the data base:

Linear Data structures:

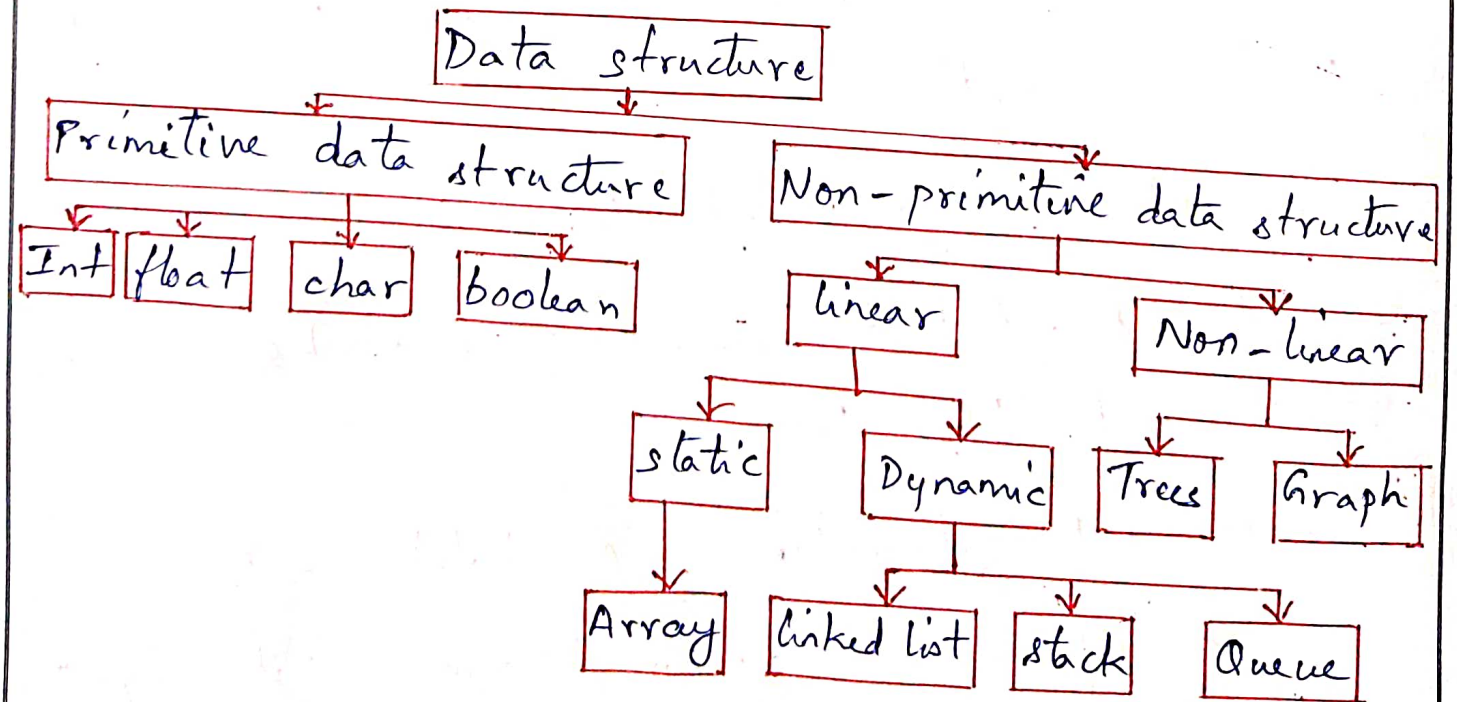
In linear data structures, the data items are arranged in a linear sequence

Ex: Array.

Non-linear data structure:

In non-linear data structure, the data items are not in sequence.

Ex: Tree, Graph



Unit III

STACKS

Abstract Data Type (ADT):

→ A data type specifies the type of data that a variable can store such as integer, float, etc.

→ Abstract Data type is a special kind of data type whose behaviour is defined by a set of values and set of operations.

→ The word Abstract indicates that we can use these data types, we can perform different operations but how these operations are working that is totally hidden from the user.

→ ADT is made with primitive data types.

→ Some examples of ADT are stack, Queue, etc.

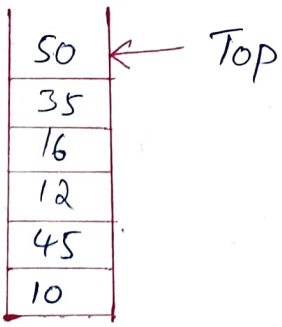
Stack ADT

Stack is a linear data structure in which the operations are performed based on LIFO principle.

→ It is a collection of similar data items in which both insertion and deletion operations are performed based on LIFO principle.

Example:

If we want to create a stack by inserting 10, 45, 12, 16, 35 and 50. Then 10 becomes bottom-most element and 50 is the top most element.



Operations on stack:

1. PUSH (To insert an element into stack)
2. POP (To delete an element from the stack)
3. DISPLAY (To display elements of the stack).

Stack data structure can be implemented in two ways. They are

(i) Using Arrays
(ii) Using Linked list

→ When a stack is implemented using an array, that stack can organize only limited number of elements.

→ When a stack is implemented using a linked list that stack can organize unlimited number of elements.

Stack Using Array

→ A stack data structure can be implemented and represented as a linear array.

→ Every stack has a variable called TOP associated with it, which is used to store the address of the top most element of the stack.

→ It is this position where the element will be added to or deleted from.

→ There is another variable called MAX, which is used to store the maximum number of elements that the stack can hold.

→ If TOP = NULL, then it indicates the stack is empty.

→ If TOP = MAX - 1, then the stack is full.

Stack Operations

4. isfull() is used to check if stack is full,
5. isempty() to check if stack is empty.

1. Push Operation:

step(i): check if stack is full.

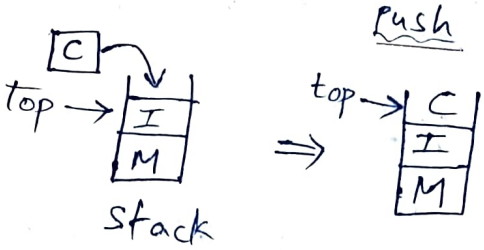
step(ii): If full, print "stack overflow" and exit.

step(iii): Else, increment top to point next empty space

step(iv): Add data element to the stack at the location pointed by top.

step(v): Return success.

Push operation



Note: If linked list is used to implement stack, then

step (iii): We need to allocate memory dynamically.

2. Pop operation

→ In an Array implementation, pop() operation does not remove data element, instead top is decremented to its lower position in the stack to point to the next value.

→ In linked list implementation, pop() actually removes data element and deallocates memory space. step (i): check if stack empty.

step (ii): If stack is empty, print "stack underflow" and exit.

step (iii): If not empty, access the data element at which top is pointing.

step (iv): Decrease the value of top by 1.

step (v): Return success.

Implement stack using linked list

stack		Linked list
push()	≡	insert_at_beg() of SLL
pop()	≡	delete_at_beg() of SLL
display()	≡	display() of SLL.
top	≡	st

✱ See all the codes given in separate word document.

Applications of stack:

1. Reverse a string
2. parentheses handling
3. 'Undo operation
4. Back traversing of web pages
5. Search history
6. Call history
7. App notifications
8. Conversion of infix expression to postfix expression.
9. Evaluation of postfix expression
10. Recursion Implementation

Infix to postfix conversion

Introduction:

Infix expression : It is of the form <operand> <operator> <operand.

Example: a + b.

→ Here we can see that the operator (+) is inbetween the operands ('a' and 'b').

Advantages:

→ Readable and solvable by humans.

→ We can easily distinguish the order of operators and can use parenthesis.

Disadvantages:

→ Computer cannot differentiate the operators and paranthesis easily.

Therefore we need postfix expression.

We use stack data structure to convert infix expression to postfix expression.

★ Convert the infix expression a*b/c +d into postfix expression.

Sol:-) a * b/c + d

⇒ ab*/c +d

⇒ ab*c/ +d

⇒ ab*c/d+ is the postfix expression

★ Convert the infix expression A*B*(C/D+F%G-H)+I^J into postfix expression.

Sol:-) A*B*(C/D+F%G-H)+I^J

→ First priority for '(' and ')'
→ Second priority for '^'
→ third priority for '*', '/' and '%'
→ Last priority for '+' and '-'
→ If equal priority, consider the operator that comes first for evaluation (from left to right), based on associativity.

$A * B * (CC/D + F\%.G - H) + I \hat{} J$

$\Rightarrow A * B * (CD/ + F\%.G - H) + I \hat{} J$

$\Rightarrow A * B * (CD/ + FG\%. - H) + I \hat{} J$

$\Rightarrow A * B * (CD/FG\%. + - H) + I \hat{} J$

$\Rightarrow A * B * CD/FG\%. + H - + I \hat{} J$

$\Rightarrow AB * * CD/FG\%. + H - + IJ \hat{}$

$\Rightarrow AB * CD/FG\%. + H - * + IJ \hat{}$

$\Rightarrow AB * CD/FG\%. + H - * IJ \hat{} +$ is the postfix expression.

★★ By observing the input infix expression and the final result of postfix expression, we develop an algorithm to convert an infix expression to postfix expression.

Algorithm for infix to postfix conversion using stack

1). Scan the given infix expression from left to right.

2) If scanned character is operand then print it.

3) If scanned character is an operator and stack is empty (or) top of stack is '(' (or) scanned character is '(' then push it onto stack.

4) If scanned operator has equal (or) low priority when compared to top of stack (when '(' not on top) then pop and print operators until condition fails.

5) Now push scanned operator onto stack.

6) If scanned character is ')' then pop all elements until '('. Discard both '(' and ')'.

7). Repeat steps 2 to 6 until end of expression.

8) At the end of infix expression, pop and print all elements from stack.

9) Resultant expression is postfix expression.

Evaluation of postfix expression

→ Let us first convert a given infix expression to postfix.

→ then let us evaluate that obtained postfix expression.

✡ **Problem**: (i) Convert the given infix expression to postfix (ii) Evaluate the obtained postfix expression.

Sol:- $3 * 2 / 3 - 4 * (5 \wedge 2 + 7)$

$\Rightarrow 3 * 2 / 3 - 4 * (5\ 2 \wedge + 7)$

$\Rightarrow 3 * 2 / 3 - 4 * 5\ 2 \wedge 7 +$

$\Rightarrow 3\ 2 * / 3 - 4 * 5\ 2 \wedge 7 +$

$\Rightarrow 3\ 2 * 3 / - 4 * 5\ 2 \wedge 7 +$

$\Rightarrow 3\ 2 * 3 / - 4\ 5\ 2 \wedge 7 + *$

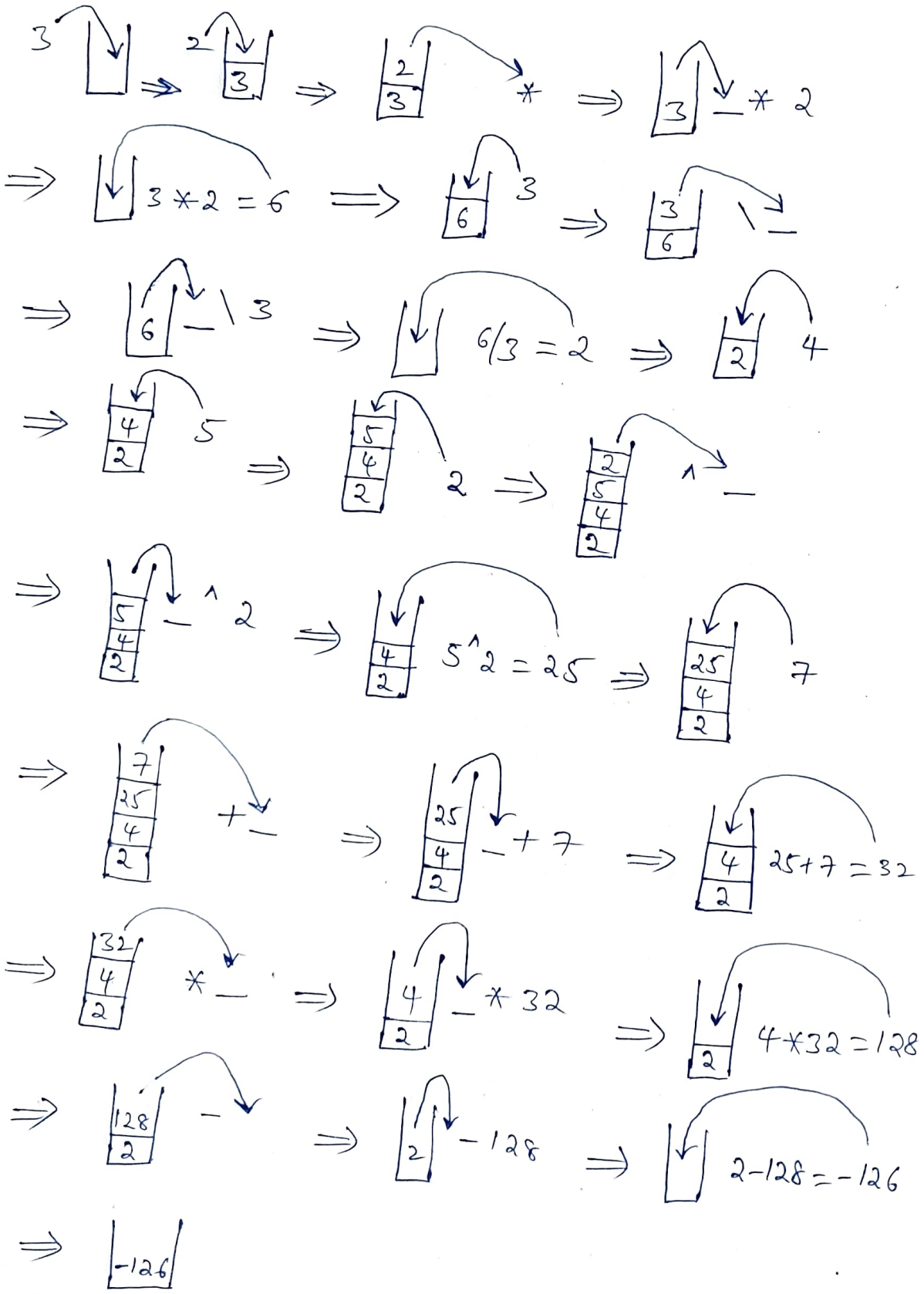
$\Rightarrow$ 32 * 3 / 4 5 2 ^ 7 + * − is the postfix expression.

By observing this we can develop rules to evaluate a postfix expression.

Example: In postfix we have 32* means actually we are supposed to perform 3*2

Evaluation of postfix expression

32 * 3 / 4 5 2 ^ 7 + * -



∴ -126 is final result stored in stack.

Based on the previous process used we develop:

Algorithm for evaluation of postfix expression:

1. Scan given postfix expression from left to right.
2. If scanned character is operand push it onto stack.
3. If scanned character is operator then pop two elements. If first popped element was A and second one was B, then perform B operator A.
4. Push above result onto stack.
5. Repeat steps 2, 3 and 4 until end of expression.
6. Pop the final result from stack and print it.

Stack Application: Balanced Delimiters

An expression is balanced if each opening bracket is closed by the same type of closing bracket in the exact same order.

Ex: ({ [ ] }) — Balanced

( ) { }] (— Not balanced

Algorithm to check balanced Delimiter

1. Scan the given expression from left to right.
2. If scanned character is open parentheris '{' or '(' or '[' then push it onto stack.
3. If scanned character is closed parentheris
   (i) If stack is empty then print unbalanced and terminate the process. Otherwise pop an element from the stack.

(ii) Compare popped open parenthesis with scanned closed parenthesis. If pair is unmatched print unbalanced and exit.

4) Repeat steps 2 and 3 until end of expression.

5) At the end of expression if stack is empty print balanced. Otherwise print unbalanced.

Unit-IV Queues

Queue ADT

Queue is a linear data structure in which insertion, deletion and accessing of data elements are performed using First in First Out (FIFO) prinaple.

Applications of Queue:

1. Spooling in printers : The documents to be printed are in Queue.
2. CPU scheduling
3. Network routing
4. In music player to arrange and play songs in sequence.

Types of Queue

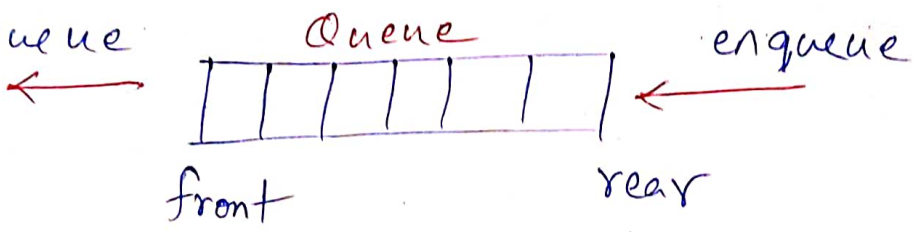
1. Simple Queue / Linear Queue
2. Circular Queue
3. Double Ended Queue (DEQUE), and so on....

1. Simple Queue:

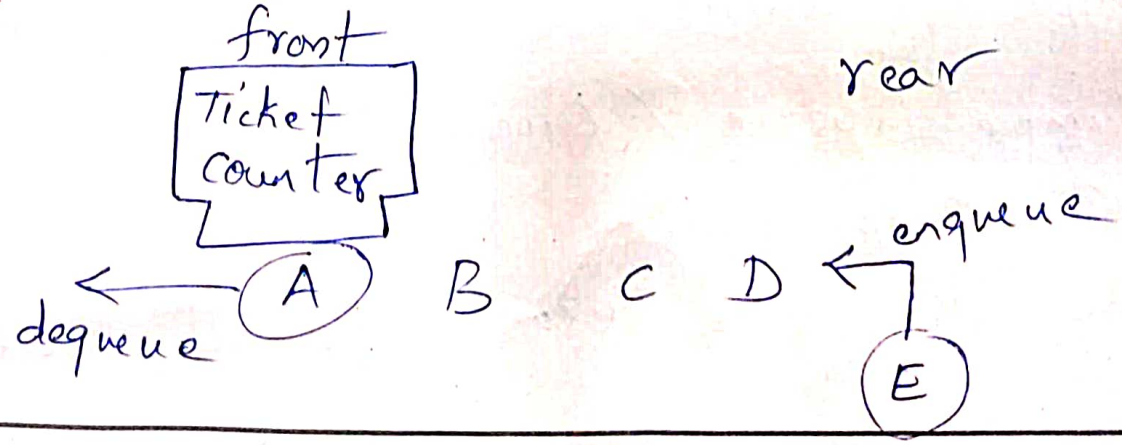
It is the most basic queue in which the insertion can be done at the front of the queue and deletion takes place at the end of the queue.

→ Insertion at the end is known as enqueue

→ Deletion takes place at the front of the queue is known as dequeue.



→ Remember a ticket counter. A person will join the queue at the end. Person at the beginning of the queue will take ticket and leave the queue first.



Subject: Data Structures
Faculty: Dr. P. Michael Preetam Raj
Topic:

Class Notes

Unit No:
Lecture No:
Book Reference:
Date Conducted:

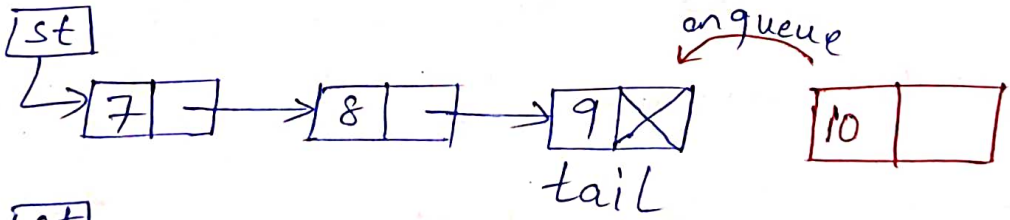
Page No: 4.2

Queue using linked list

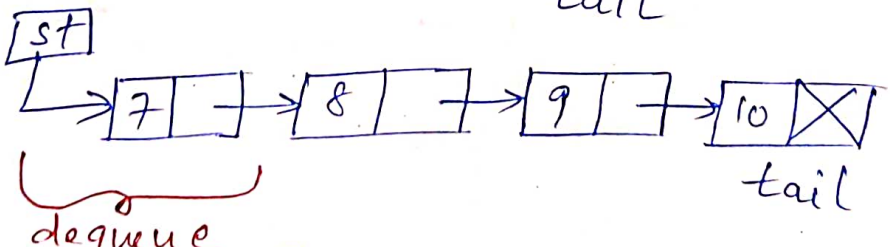
→ We can develop the code for enqueue based on insert_at_beg ( ) function for singly linked list.

→ However it must work for insertion of first element also.

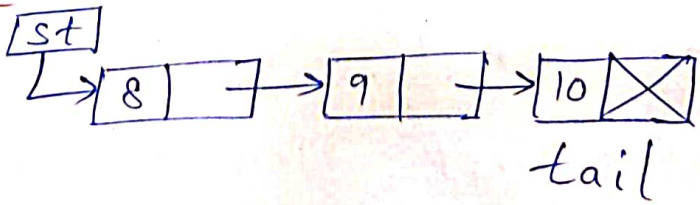
→ Therefore we develop the code based on create ( ) function of the singly linked list. But, it should be modified such that it executes only once (corresponding to insertion of one element) without any loop in it.



⇒



⇒



Basic functions or operations in Queue

1. enqueue() : Adding an element to Queue
2. dequeue() : Removing an element from Queue
3. isfull() : To check whether queue is full.
4. isempty() : To check whether queue is empty.
5. display() : To display elements in the queue.

We may also use a very simple function peek() : To get the element at the front of the queue without deleting it.

→ We use linked list implementation of queue to overcome the drawbacks associated with array implementation.

→ For array implementation, the amount of data must be specified at the beginning itself.

→ In linked list implementation, we use a pointer (start or front) to access the

first element.

→ We use a pointer (tail or rear) to access the last node.

→ The use of tail pointer reduces the unnecessary traversing from first node to last node for enqueue.

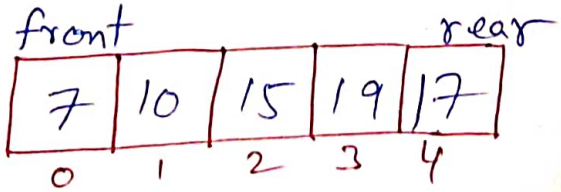
→ As a result, time complexity is reduced from O(1) to O(n).

Array Implementation of Queue:

We enqueue an item at the rear and dequeue item from the front.

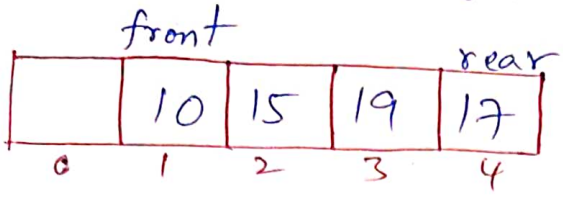
Let us consider an array of size 5

```
int a[5];
```



If we enqueue() now it correctly shall indicate overflow condition.

If we dequeue an element, Then front is incremented.



→ If we enqueue now, it still indicates overflow condition.

→ We can see that memory is not utilized efficiently since there is available memory location (at index 0), however we are unable to insert.

→ To overcome this drawback, we go for circular queue.

Circular Queue:

→ In a circular queue the last node is connected to the first node.

→ Insertion takes place at front whereas deletion takes place at rear end (similar to that of a Queue).

Applications:

→ Memory management.

→ Process scheduling.

Subject: Data Structures
Faculty: Dr. P. Michael Preetam Raj
Topic:

Class Notes

Unit No:
Lecture No:
Book Reference:
Date Conducted:

Array Implementation of Circular Queue:

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

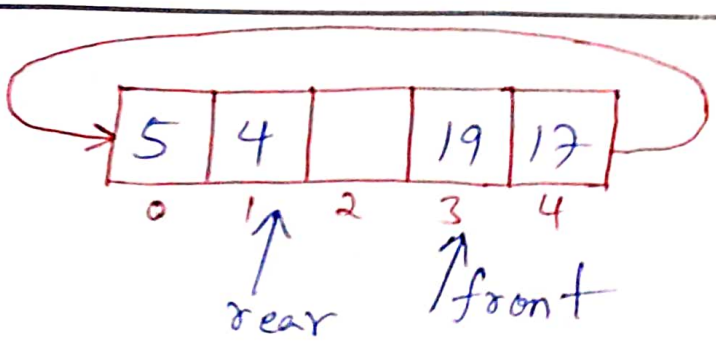
→ Here the circular increment is performed by modulo division with the queue size (N)

that is $$rear = (rear + 1) \% N$$

→ We can wisely modify the Queue using array to Circular queue using array by wisely using the % operator at appropriate places to achieve the circular action.

Example

Linear increment		Circular increment
0	% 5 =	0
1	% 5 =	1
2	% 5 =	2
3	% 5 =	3
4	% 5 =	4
5	% 5 =	0
6	% 5 =	1
7	% 5 =	2



In the above figure, the three (0,1,2) vacant places (created due to three dequeues) can be efficiently utilized in a circular Queue.

→ It can be seen that 5 and 4 were efficiently placed in 0 and 1 index positions.

→ One more element can be placed at index 2.

Implementation of Circular Queue using Linked list:

It is similar to circular linked list. We can easily convert a Queue using linked list to Circular Queue using linked list by creating the link between last node and first node

→ We just need to add

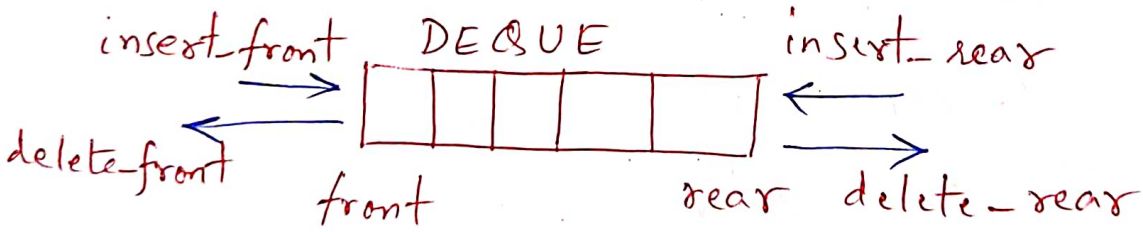
tail->next = st; //Add 1 in each of enqueue & dequeue

at the end of enqueue() and dequeue() functions to achieve the circular Queue (from simple Queue).

Double Ended Queue (DEQUE)

→ It is a list in which the elements can be inserted or deleted at either ends.

→ However, no element can be added or deleted from the middle.



→ In a DEQUE we need to traverse in both directions.

→ For example, when we delete front/insert rear we move from left to right.

→ When we delete rear / insert front we need to travel from right to left.

Subject: Data Structures
Faculty: Dr. Michael
Topic:

Class Notes

Unit No:
Lecture No:
Date Conducted:
Page No:

Priority Queue

→ It is a special type of Queue in which elements are arranged based on their priority.

→ Elements with higher priority values are retrieved before elements with lower priority values.

→ In a priority Queue, each element has a priority value associated with it. When you add an element to the queue, it is inserted in a position based on its priority value.

→ Ex: If you add an element with a high priority value to a priority queue, it may be inserted near the front of the queue, while an element with low priority value may be inserted near the back.

Properties of Priority Queue:

→ Every element in queue has a priority associated with it.

→ An element with high priority is dequeued before an element with low priority.

→ If two elements have the same priority, they are served according to their order in the queue.

Operations of a Priority Queue:

Insertion: Add an element to the queue with an associated priority.

Deletion: Remove the element from the queue that has the highest priority.

is_empty: check whether queue is empty.

is_full: check whether queue is full.

display: Print the elements in the queue.

peek: Retrieves but not removes front element in queue.

Subject: Data Structures
Faculty: Dr. Michael
Topic:

Class Notes

Unit No:
Lecture No:
Date Conducted:
Page No:

Types of Priority Queue:

1. Ascending order priority Queue:

Element with lower priority value is given a higher priority.

Ex: 4, 7, 8, 9, 12 (priority values)

→ 4 is the smallest number, therefore, it will get the highest priority.

→ If we dequeue, 4 will be removed from the queue, (i.e. element with priority 4).

2. Descending order priority queue:

Element with highest priority value is given a higher priority.

Ex: 12, 9, 8, 7, 4 (priority values)

→ 12 is the largest number, therefore, it will get the highest priority.

→ If we dequeue, element with priority value 12 will be removed.

Implementation of Priority Queue using Array :

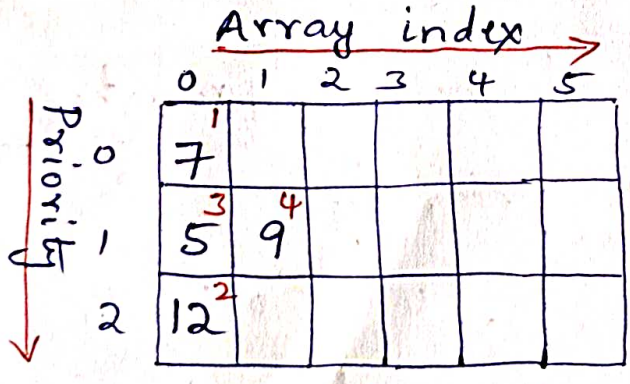
→ A 2D-Array can be used to implement a priority Queue, where each row represents priority value.

→ Let us consider ascending order priority.

If we insert {value, priority} as

{7, 0}, {12, 2}, {5, 1}, {9, 1}
(1) (2) (3) (4)

then we have



When we display the elements in priority Queue 7, 5, 9, 12.