Unit -III

Stack & Queue

★ Stack

- array can store multiple elements at a time.
- As array is a flexible DS, we can add & remove elements anywhere in the array.
- Therefore, there is no track that how, where and when elements are added & removed.
- Solution to this problematic flexibility is stack, queue, tree, graph etc.
- With own rules, while performing operations on data elements; record get maintained properly.

Stack

Stack is Data Structure in which addition and removal of an element is allowed at the same end called as top of the stack.

Examples :- ① Book shelf
② a stack of plates.
③ CD / DVD Holder, etc.

- Stack is also known as LIFO (Last in First out) DS.

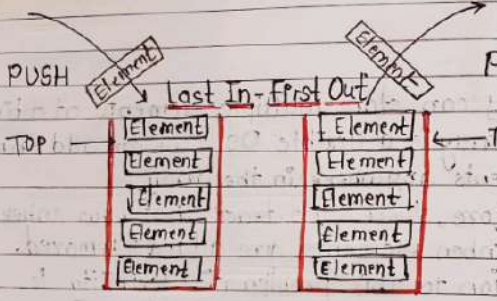


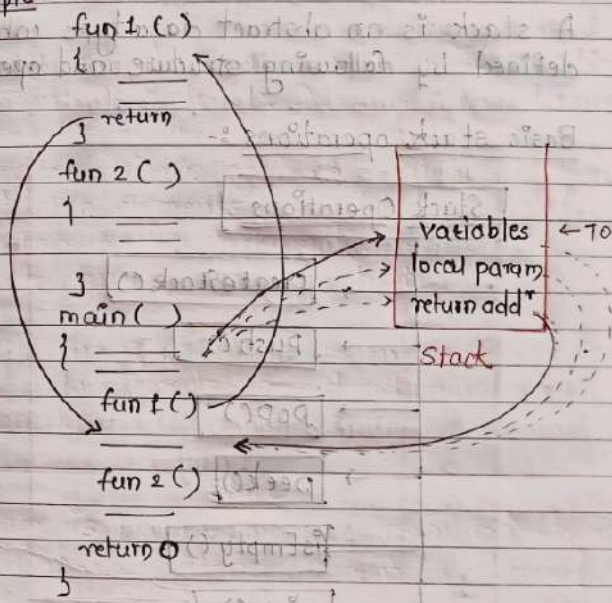
fig: structure & operations of stack.

★ Concept of Implicit and Explicit stack.

① Implicit stack:-

- Implicit stack is used in the process of recursion.
- This stack is not allowed to be implemented by programmer directly. Such stack is present in every computer system.
- This stack is used by the runtime memory environment to hold the information regarding different function calls.

Example:



② Explicit stack

- Programmers are allowed to implement explicit stack as ADT.

Examples:

① any non-recursive version of recursive algo.

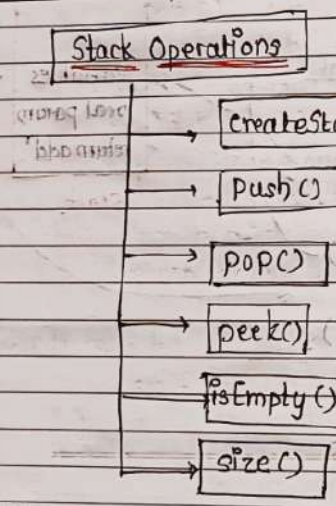
② Non-recursive tree traversal algorithm.

③ Infix to prefix conversion of expression.

★ Stack as ADT

A stack is an abstract data type which is defined by following structure and operations.

Basic stack operations :-



These are all built-in operations to carry out data manipulation and to check the status of the stack.

① CreateStack()
- It creates a new empty stack.
- It does not require any parameter and returns an empty stack.

② isfull()
- It checks whether stack is full. This operation is used to check the status of the stack with the help of top pointer.

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

int isfull()
{
  if (top == MAXSIZE)
    return 1;
  else
    return 0;
}
```

③ push (item)
- It insert new element at the top of the stack.
- It needs the item to be added and returns nothing.

if top >= size-1
"stackoverflow"
Top = Top+1
stack[top] = X

```
int push(int data)
{
  if (!isfull())
  { top = top+1;
    stack[top] = data;
  } else; printf("stack is full");
}
```

④ pop()

- It removes the topmost element from the stack.
- It does not require any parameter and returns the item. The stack is modified.

```
int pop()
{
  int data;
  if (!isEmpty())
  {
    data = stack[top];
    top = top - 1;
    return data;
  }
  else
    printf("Stack is empty");
}
```

if Top=-1
stack underflow
return stack[top]
top=top-1

⑤ isEmpty()

- It is used to test whether the stack is empty or not.
- It does not require any parameter & returns a Boolean value.

```
if top == -1
  (stack is empty); return 1;
else
  return 0;
```

```
int isEmpty()
{
  if (top==-1)
    return 1;
  else
    return 0;
}
```

⑥ peek()

- It returns the topmost element from the stack but does not remove it.
- It does not require any parameter. The stack is not modified.

```
int peek()
{
  return stack[top];
}
```

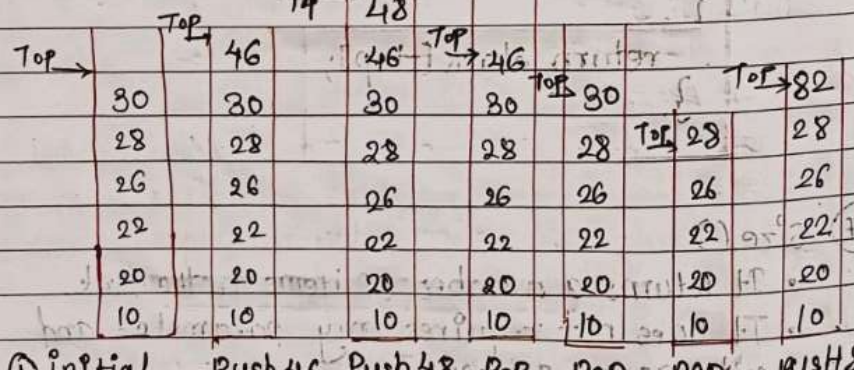
⑦ size()

- It returns a number of items in the stack.
- It does not requires any parameter and returns an integer.

Example:- ①

Stack operation	Stack content	Return Value
S.isEmpty()	[ ]	True
S.push ("A")	[A]	
S.push (5)	[A,5]	
S. peek ()	[A,5]	5
S. push ("B")	[A,5,B]	
S. size()	[A,5,B]	3
S. pop ()	[A,5]	B

Example ②:- The stack contain 10, 20, 22, 26, 28, 30 with 30 being at top of the stack.
Show diagrammatically the effect of PUSH 46 PUSH 48 POP POP POP PUSH 82.



★ Implementation of stack using sequential organization.

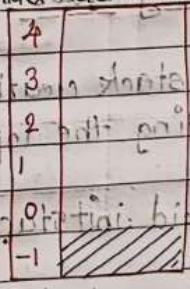
- Stack is generally represented in two ways:
  - static
  - dynamic
- Static implementation of stack is done using array. (as size of array is fixed)
- Dynamic representation is done using linked list.

Stack using array

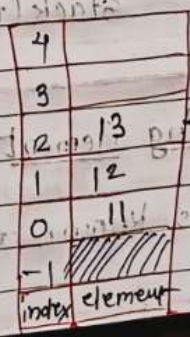
"Stack is a DS in which the topmost element is pointed by the integer variable top"

int arr[5];

- if top = -1
  then stack is empty.



Push (11)
Push (12)
Push (13)
} top = top+1



push-(14)	4	15	← top
push (15)	3	14	
	2	13	
	1	12	
	0	11	
	-1		

if top == arraysize
then "stack is full"

if top = arraysize → can't add element (as full)
if top = -1 → can't remove element (as empty)

① Initialising Stack :

- After stack creation, initialize it with top by setting the top at position -1.

```
void initstack ()
{
    stack[top] = -1;
}
```

② Inserting Element in the Stack :

- It is known as push operation on stack.

- Before inserting element in stack, first we check whether it is full or not.

if full ⟶ ove Stack Overflow.

e.g. ① function to check stack is full or not.

```
int isfull()
{
  if (top == size-1)
      return(1);
  else
      return(0);
}
```

② Function to add (push) element.

```
Void push()
{
 if (isfull()==1)
    {
      cout << "\n\t Stack is over flow";
    }
 else
    {
      cout << "\nEnter an element to add in stack:";
      cin >> ele;
      top++;
      stack[top] = ele;
    }
}
```

③ Deleting Element from the stack

- The deletion of element from the stack is known as pop.
- Before deleting, check whether stack is empty or not?

if stack is empty ~~(-1)~~ stack underflow.

① function to check whether stack is empty or not

```
int is_empty()
{
    if (top == -1)
        return (1);
    else
        return (0);
}
```

② function to delete (pop) an element

```
void pop()
{ if (is_empty() == 1)
  { cout << "\n\t Stack is underflow"
  }
  else
  { cout << "\n Element popped:" << stack[top];
    top--;
  }
}
```

④ Displaying Elements of stack

- As stack is known as LIFO structure, the elements are displayed in reverse order
- first, check stack is empty or not,

```
void display()
{
  int i;
  if (is_empty() == 1)
  {
    cout << "Stack is underflow";
  }
  else
  {
    cout << "\n Stack Elements";
    for (i = top; i >= 0; i--)
    {
      cout << "\n " << stack[i];
    }
  }
}
```

★ Algorithm of stack implementation using Array.

Step1: Start
Step2: Display menu:
1: push
2: pop
3: display
4: Exit
Step3: read choice
Step4: if choice 1
call push fun°
if choice 2
call pop fun°
if choice 3
call display fun°
if choice 4
exit
default: display invalid choice
Step5: Read choice again
Step6: if choice between 1–3
repeate step 4
else
Stop.

★ Stack as an ADT using Linked Organization

- Stack elements are placed one above other.
- In same way, in stack as linked list, we will place nodes one above other.

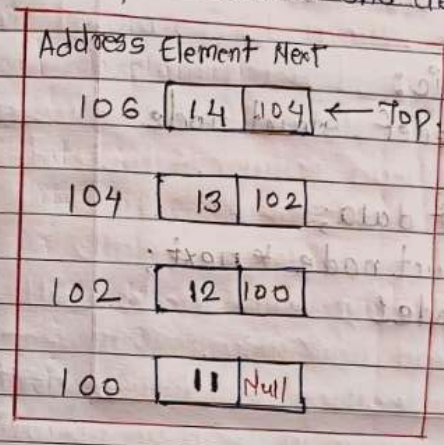


fig: Stack as a linked list.

Advantages of Dynamic implementation of stack

① No memory wastage
② No memory shortage
③ no limitation on number of elements.

① Initialising stack & list:

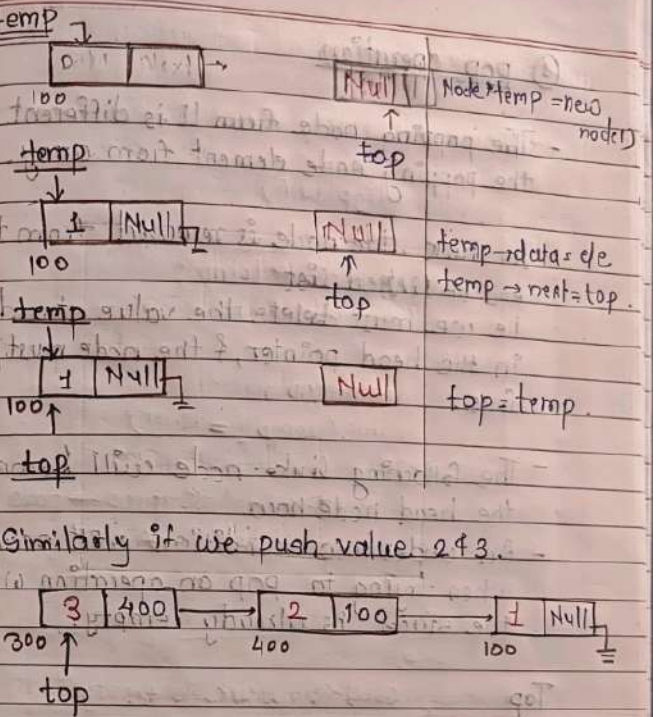
Create a class of stack & struct for list.

```
Class stacktest
{
  public:
  typedef struct node
  {
   int data;
   struct node *next;
  }node1
};
```

② push function

- Linked list implementation for push fun?

```
node1 *top;
void push(int ele)
{
   struct node *temp;
   temp = (struct node *) malloc (sizeof
          (struct node));
   if (temp == NULL)
      cout << "stack is overflow"
   temp → data = ele;
   temp → next = top;
   top = temp;
}
```



Similarly if we push value 2 & 3.

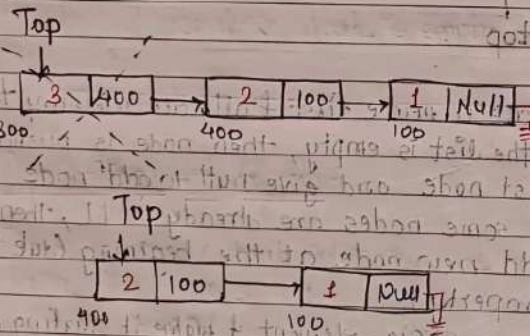
(i) Create a node first & allocate memory to it
(ii) if the list is empty, then node is pushed as first node and give null to add' node.
(iii) if some nodes are already in LL, then add new node at the begining (due to stack property)
- assig. element & make it starting node.
(iv) An overflow conditions occurs when trying to push an operation if the stack is already full.

③ pop operation

- The popping node from LL is different from the popping node element from array.

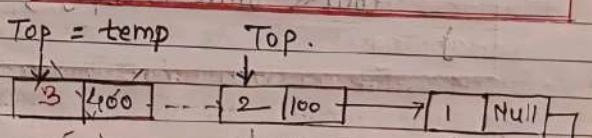
- In stack, the node is removed from the end of linked list
  i.e we must delete the value stored in the head pointer, & the node must get free.

- The following linked node will become the head node now.
- An underflow condition will occur when trying to pop an operation when the stack is already empty.




```
void pop()
{
    struct node *temp;
    int item;
    if (top == NULL)
    {
        cout << "\n Stack is Underflow";
        return;
    }
    temp = top;
    item = temp -> data;
    top = top -> next;
    free (temp);

    cout << "Element poped" << item;
}
```



④ Display function

- Display in stack elements.

```
Void display()
{
  struct node *temp;
  if (top != NULL)
  {
    temp = top;
    while (temp != NULL)
    {
      cout << "\n" << temp->data;
      temp = temp->next;
    }
  }
  else
    cout << "stack is empty";
}
```

* Applications of Stack

Stack is very useful data structure in managing data. Applications are:

1) Recursion
2) Converting expression from infix to postfix or prefix form.
3) Evaluating postfix or prefix form.

* Recursion

Recursion: Calling a function inside itself is called as recursion. Such function is called as recursive function.

```
Void recurse()
{
  .....
  recurse();
}
int main()
{
  .....
  recurse();
  .....
}
```

recursive call

- The execution of recursion continues unless and until some specific condition is met to prevent its repetition.
- To avoid infinite recursion, if else statement can be used.

① Use of recursion:

- When a function is called, memory for the function is allocated on the stack space of the process and it is a run time activity.

- When fun? is called, activation record i.e a stack frame is created and pushed on top of stack & when the fun? returns the, activation record is popped.

② Advantages of Recursion

- to reduce the size of program by minimizing the code.
- easy to maintain fun? calling related info.
- Evaluation of stack can be implemented through recursion.
- Infix, prefix, post-fix notation can be evaluated with the help of recursion.

③ Disadvantages of Recursion

- It takes more time because of stack overlapping
- Stack overflow may occur in recursive program.
- memory requirement is more at every recursive call, seperate memory address is allocated for the variables
- It may leads to indefinite call if exit condition does not work
- efficiency is less.

④ Examples of Recursion

Recursion

Direct Recursion . Indirect Recursion

Direct recursion : Recursion is said to be direct when function calls itself directly.
i.e. fun^n body contains an explicit call to itself.
e.g:- factorial function.

Indirect Recursion: Recursion is said to be indirect when it calls other function that in turns call it.

- The fun^n involved in indirect recursion are mutually recursive functions.
- Comparatively, it is simpler and safer to use direct recursion as it is much easier to understand and apply.

Converting expressions from Infix to Postfix / Prefix form

- Notation is the way of writing arithmetic expressions.

① **Polish Notations** : It is the way of expressing arithmetic expressions that avoids the use of brackets to define priorities for evaluation of operators.

- There are 3 ways to write expressions.
- All 3 ways are different but equivalent notations
  i.e. even though they are different, they do not change the essence / output.

Polish Notations

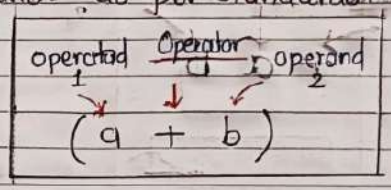
→ 1. Infix Notation
→ 2. Prefix Notation (polish)
→ 3. Postfix Notation (reverse-polish)

fig: Polish Notations

- The way of using operators in the expression decides name of notations.

★ Infix Notation

- It is usual and basic way of writing an expression.
- Here, operators are placed in between the operands as per standards.



Here a & b = operands
and + = operator.

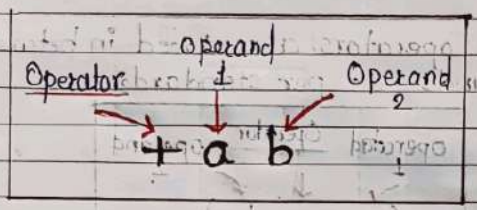
- Human readable format: easy to read, write and speak.
- Sometimes, not so easy for computing devices which leads to conversion of infix format into other formats such as prefix & postfix.

★ Prefix (Polish) Notation

- It is another way to write an airthmetic expression in which operators are placed before the operands.

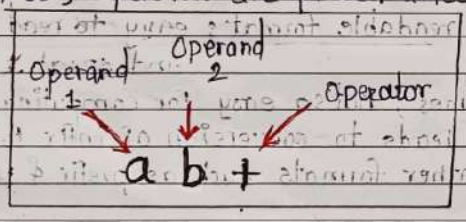
- This notation is considered as equivalent to it's infix notation.

- Prefix notation is also known as Polish Notation



★ Postfix (Reversible - Polish) Notation

- Here, operators are placed after operands.



- This notations is considered as equivalent to it's infix notation

- Postfix notation is also known as Reversed Polish Notation

Need of Prefix and postfix conversion

- Infix notations considered as easy to read for humans, whereas pre-/postfix notations is easier to parse for a machine.

- Benefits :-
  1) It is very convenient for evaluating formulas on computer with stacks.
  2) No any issues regarding operator precedence.
  3) Less overhead of brackets
  4) Reduce ambiguity.

★ Conversion of Infix to Postfix Expression

- Infix is a regular format whereas postfix means operators are placed after operands.

  e.g. company ⟶ appoint a stu engg.
  qualification ⟶ Engineering
  i.e. every engineer completed engg will not be appointed.
  other para ⟶ knowledge, sincerity etc.

- keeping operators at end is the basic rule

- Here, stack is used as mediator bet" source string (infix) and target string (postfix)
- stack will hold operators for some time.

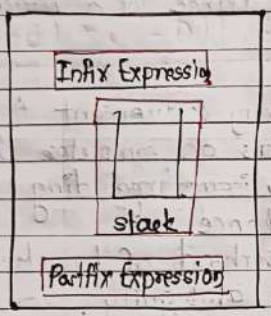


fig: Infix to Postfix conversion.

① Algorithm of Infix to Postfix

Step1: The I/p string (infix notation) is scanned from left to right.

Step2: If the scanned ch is space or tab, it is skipped.

Step3: If the scanned char ch is digit or alphabet, then it is appended to post fix string

Step4: If scanned ch is opening paranthesis '(', it is pushed to stack.

Step5: If scanned ch is operator:

ⓐ all operators from top of the stack are popped having more or same priority than ch and appended to postfix string.

ⓑ if have less priority operator, then it is pushed in the stack again with ch.

Step6: If scanned ch is closing paranthesis ')', then
- all operators above the opening paranthesis are appended to postfix string.

Step7: After Evaluations of all characters, the remaining operators from stack are appended to postfix string.

② Examples :-

Q. Consider infix string : ((A+B) * (C-D)) / E convert it into postfix form.

Ans ((A+B) * (C-D)) / E

char. from infix	stack	Postfix Expression
(	(	
(	((	
A	((	A
+	((+	A
B	((+	AB
)	(	AB+
*	(*	AB+
E	(*E	AB+
C	(*C	AB+C
-	(*C-	AB+C
D	(*C-	AB+CD
)	(*	AB+CD-
)		AB+CD-*
/	/	AB+CD-*
E	/	AB+CD-*E
		AB+CD-*E/

Q. Explain stepwise conversion using stack for the given infix expression to postfix expression.
A * B + C * D

Ans →

Character Scanned	Stack	Postfix
A	Empty.	A
*	*	A
B	*	AB
+	+	AB*
C	+	AB*C
*	+*	AB*C
D	+*	AB*CD
	empty	AB*CD*+

Q. Explain conversion using stack for infix to postfix
A * (B + C) * D

Char. scanned	stack	postfix
A		A
*	*	A
(	*(	A
B	*(	AB
+	*(+	AB
C	*(+	ABC
)	*	ABC+
*	*	ABC+*
D	*	ABC+*D
	Empty	ABC+*D*

(2) Postfix Expression Evaluation

- The evaluation of this is easy
- Postfix Expression is without parantheis
- It can be evaluated as 2 operands & 1 operator at a time
- easier for the complier.

Rules

1. Read elements left → right and push element if it is an operand.

2. If element is operator then
   - pop two elements from stack
   - Evaluate expression by inserting operators in operands.

3. Push the result of evaluation Repeat it till end of Expression

4. Print the popped element as a result.

Algorithm

Step 1: Read postfix expression from Left → Right.

Step 2: If operand is encountered, push it is stack

Step 3: If operator is encountered,
- Pop two elements
  - A → Top element
  - B → Next element.
- Evaluate B operator A

Step 4: Push result into stack.

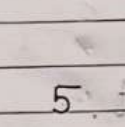
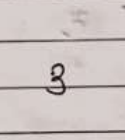
Step 5: Read next postfix element; if not end of postfix string

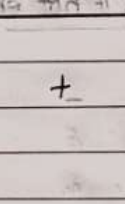
Step 6: Repeat from Step 2

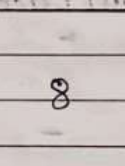
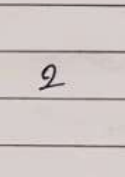
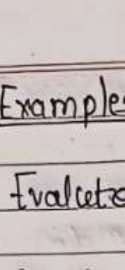
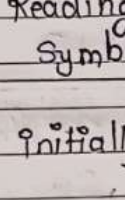
Step 7: Print the result popped from stack.

Examples

Q Evaluate postfix expression: 53+82-*

Reading Symbol	Stack operations	Evaluated part of Expn
Initially	Empty	—
5	push(5) 	—
3	push(3) 	—

+	val1=pop(); val2=pop(); R=5+3=8; push(8) 	val2=pop(); // 3
		val2=pop(); // 5
		R=5+3;
		push(8)

8	push(8) 	(5+3)
2	push(2) 	(5+3)
	val1=pop(); val2=pop(); R=8-2=6; push(8-2) 	(5+3), (8-2)
*	val1=pop(); val2=pop(); R=val2*val1; push(R) 	(6*8) (5+3) * (8-2)
End of Expression	result=pop()	Display 48.

postfix.

Q. Evaluate the following Exprⁿ & show stack after every step. In tabular form.

Given A=5, B=6, C=2, D=12, E=4

ABC+*DE\-

→ Exprⁿ become 562+*124\-

Reading	stack	Evaluated
5	5	
6	6,5	
2	2,6,5	
+	8,5	6+2=8
*	40	5*8=40
12	12,40	
4	4,12,40	
\	3,40	12/4=3
−	37	40-3=37

Q. if values of A, B, C & D are 2, 3, 4 & 5 resp. Calculate the value of following postfix Exp'
i) AB * C - D +
ii) ABC + * D -

→

Reading	stack	Evaluated
① 23*4-5+		
2	2	
3	3, 2	
*	6	2*3=6
4	4, 6	
-	2	6-4=2
5	5, 2	
+	7	2+5=7
② 234+*5-		
2	2	
3	3, 2	
4	4, 3, 2	
+	7, 2	3+4=7
*	14	2*7=14
5	5, 14	
-	9	14-5=9

Q. Evaluate postfix Expr^n
A: 6, 2, 3, +, -, 3, 8, 2, /, +, *, 2
↑, 3, +

ans:- 172

Q. Consider following postfix Exp^n & Evaluate:
10, 2, *, 15, 3, /, +, 12, 3, 2, ↑, /, +

ans:- 46

* Converting an Infix into Prefix Expression.

- In conversion, stack is used as mediator bet^n source (infix) string & target (Prefix) string.
- Stacks will hold operators for some time.

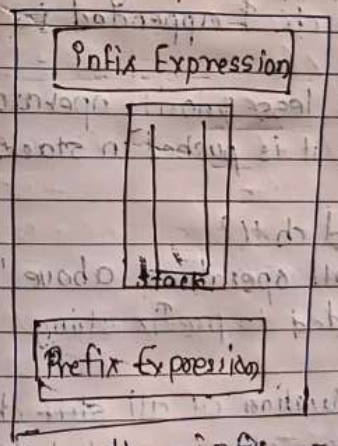


fig: Converting infix → prefix.

① Algorithm

step1: The infix expression is first reversed.

step2: The input string (infix) is scanned from left → right

step3: If scanned ch is space/tab, it is skipped.

step4: If scanned ch is digit or alphabet then appended to prefix string

step5: If the scanned ch is ')', it is pushed to stack

step6: If the scanned ch is operator:

(i) all operators from top of the stack are popped having more priority than ch & appended to prefix string

(ii) When less priority operator is found then it is pushed in stack again

step7: If scanned ch '('
then all operators above ')' are appended to prefix string

step8: After evaluation of all ~~str~~ characters, remaining operators appended to prefix string

② Examples

Q. Convert the given infix string to prefix expression & show the details of stack at each step.
(A - B / C) * (D * E - F)

→ first reverse string
(F - E * D) * (C / B - A)

Char. scanned	stack	Postfix
(	+(	
F	((	F
-	(-	F
E	(-	FE
*	(-*	FED
)	Empty	FED*-
*	*(	FED*-
(	*(	FED*-
C	*((	FED*-C
/	*(/	FED*-C
B	*(/	FED*-CB
-	*(-	FED*-CB/
A	*(-	FED*-CB/A
)	*	FED*-CB/A-
End	Empty	FED*-CB/A-*

Now, reverse the Expression

Prefix Expression :- *-A/BC-*DEF

Q. Convert infix string ((A+B) * (C-D)) / (E+F) into prefix string with stack. Show the content of stack.

→ first reverse string.

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

char scanned	stack	Postfix
(	(	
F	(	F
+	(+	F
E	(+	FE
)	Empty	FE+
/	/(	FE+
(	/((	FE+
(	/((	FE+
D	/((	FE+D
−	/((−	FE+D
C	/((−	FE+DC
)	/(−	FE+DC−
*	/(*	FE+DC−
(	/(*(	FE+DC−
B	/(*(	FE+DC−B
+	/(*(+	FE+DC−B
A	/(*(+	FE+DC−BA
)	/(*	FE+DC−BA+
)	/	FE+DC−BA+*
end	Empty	FE+DC−BA+*/

/*+AB−CD+EF

Q. Infix:- (F−E*D) * (C/B−A)

prefix: * −F*ED−/CBA.

Evaluation of Prefix Expression

① Algorithm

Step1: Accept a prefix string

Step2: Start scanning from right → left.

Step3: If it is an operand, push it in stack

Step4: If it is an operator, pop opnd1, opnd2 & perform the operation.

Push result in stack.

Step5: Repeat these steps until arr of ilp prefix strings ends.

Q. Evaluate prefix Expression for following

−*+4325

i.e −*+ABCD

consider A=4, B=3, C=2 & D=5

symbol	op1	op2	value	opndstack
5				5
2				2,5
3				3,2,5
4				4,3,2,5
+	4	3	7	7,2,5
*	7	2	14	14,5
-	14	5	9	9
				Result.

Q. -+8/632

ans - it is 8

Q. -+7*45+20

ans - 25

Q. 321+*74+-33+1

Ans - 3

Converting an Prefix into Postfix Expression

① Algorithm

Step 1: Read the prefix expression in reverse order (from right to left)

Step 2: If the symbol is operand then push it onto the stack.

Step 3: If the symbol is an operator then
- pop two operands from the stack
- create a string by concatenating the 2 operands & the operator after them.

Step 4: string = operand 1 operand 2 operator
push the resultant string back to stack.

Step 5: Repeat the above steps until end of prefix expr^n.

Q. Converts the following prefix expression into postfix.

*+a-bc/-de+-fgh

Prefix	stack
*+a-bc/-de+-fgh	h
*+a-bc/-de+-fg	gh
*-a-bc/-de+-f	fgh
*-a-bc/-de+-	fg-
*-a-bc/-de+	fg-h
*-a-bc/-de	fg-h+
*-a-bc/-d	efg-h+
*-a-bc/-	defg-h+
*-a-bc/	de-fg-h+
*-a-bc	fg-h+ de-fg-h+
	de-fg-h+/

Q. prefix to postfix.

① *+AB-CD

② *+AB-C D

③ *+AB- C, D

④ *+AB CD-

⑤ *+A B, CD-

⑥ *+ A, B, CD-

⑦ * AB+, CD-

⑧ Empty

AB+ CD- *

ans.

Q. 2

1) *-A|BC-|AKL

2) *-A|BC-|AK

3) *-A|BC-|A

4) *-A|BC-|

5) *-A|BC-

6) *-A|BC

7) *-A|B

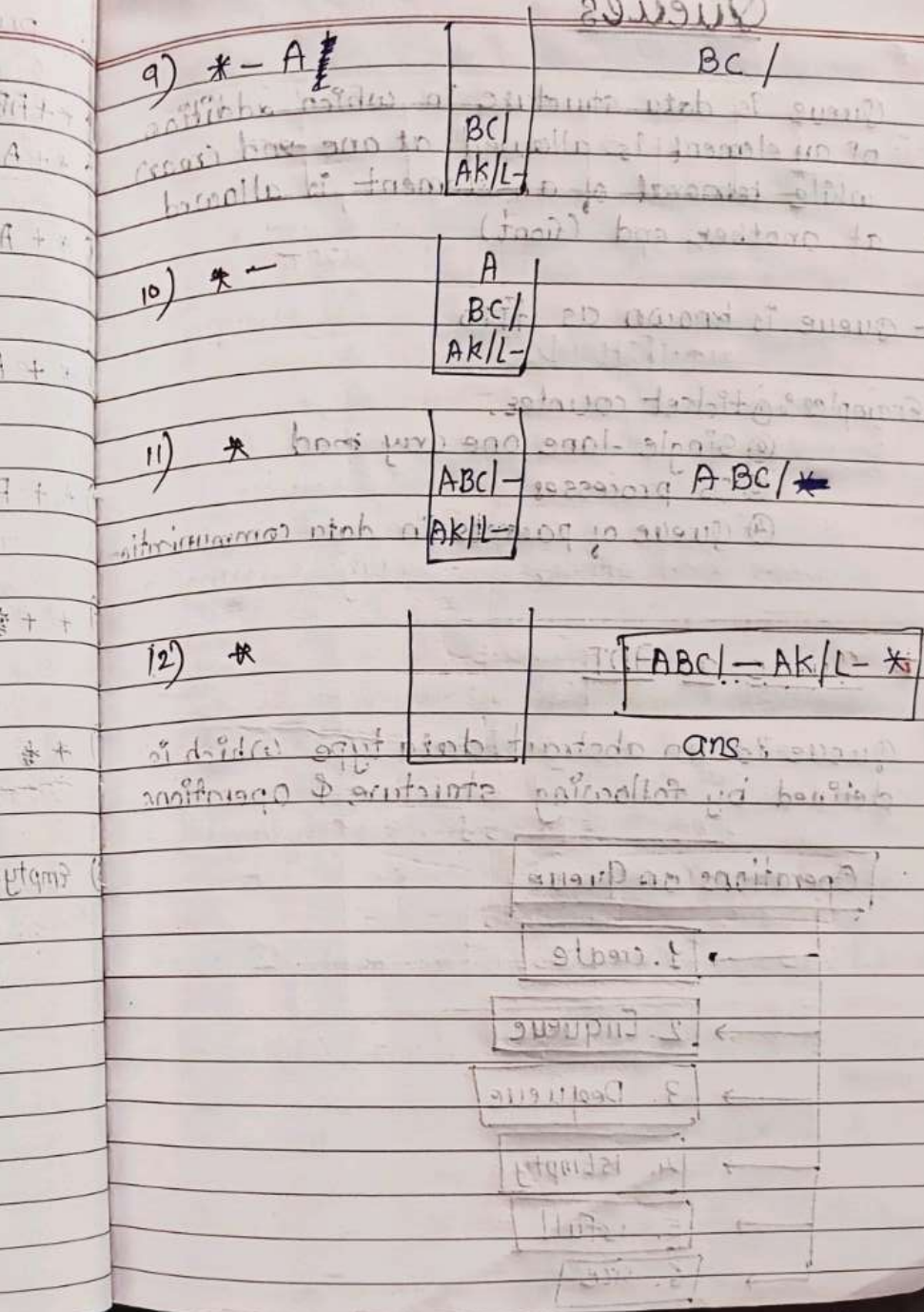
8) *-A|

9) *-A

10) *-

11) *

12) *



Queues

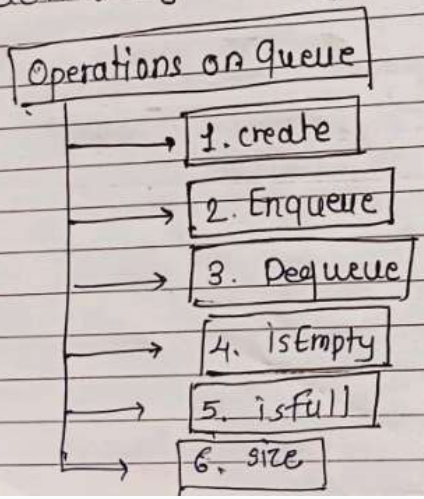
Queue is data structure in which addition of an element is allowed at one end (rear) while removal of an element is allowed at another end (front)

- Queue is known as FIFO

Examples: ① ticket counter
② Single-lane one way road
③ OS processes.
④ Queue of packets in data communicatio

★ Queue as ADT

Queue is an abstract data type which is defined by following structure & operations.



- create(): creates and initializes new queue that is empty.
  - It does not require any parameter & returns an empty queue.

- Enqueue (item): Add a new element to the rear of queue.
  - It requires the element to be added & returns nothing.

- Dequeue (~~item~~): Removes the element from front of queue.
  - It does not require any param & returns deleted item.

- isEmpty(): checks whether queue is empty or not.
  - It does not require param & returns boolval

- Is Full(): checks whether queue is full or not
  - It does not require param & returns boolval.

- Size(): returns total no. of elements present
  - returns integer.

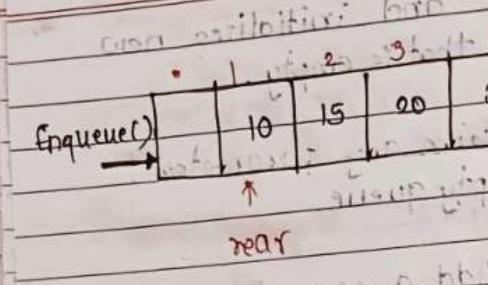


fig: FIFO DS

Example

Consider q is a queue that has been created and starts out empty.

Queue operation	Queue Content	Return value
q.IsEmpty()	[]	True
q.enqueue("A")	[A]	
q.enqueue(5)	[A, 5]	
q.dequeue()	[5]	A
q.enqueue("B")	[5, B]	
q.size()	[5, B]	2

Difference between Stack and Queue

Parameter	Stack	Queue
Operation end	Elements are added & deleted from same end.	Elements are added & deleted from different ends.
Pointer	Sigle top pointer is used	two pointers used: front & rear ends.
Order	follows LIFO order	follows FIFO order
Operations	push and pop	enqueue & dequeue
visualization	Vertical visualization	Horizontal visualization.
Examples	Book shelf	tickets at theatre.

Implementation of Queue

- using arrays
- using linked list

★ Implementation of Queue using Arrays

- Queue is linear DS ; hence can be represented using array.
- It is also known as static implementation of queue.
- Two pointers used :
  - front → point to first element
  - rear → point to last element

~~front will~~

Example:

```
int arr[5];
```

- Initially queue is empty.
- front & rear both will point to -1
- if front & rear both are at -1
  - ↳ queue is Empty.

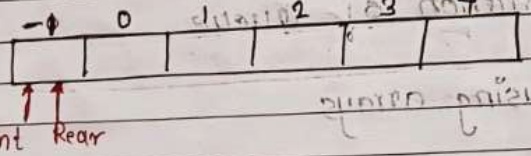
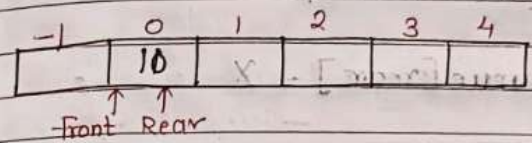


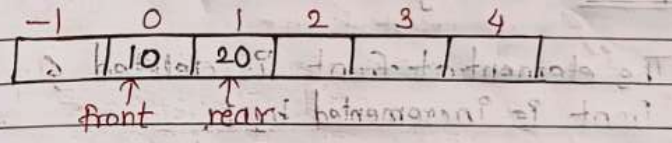
fig: Initialization of queue.

① Enqueue : Inserting an element in Queue.

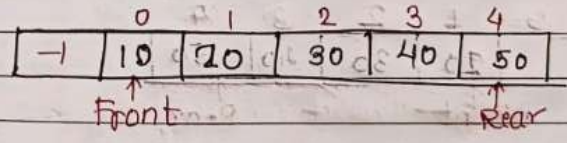
- While adding 1st element
  - ↳ front and rear are incremented by 1
  - ↳ at rear position new element added.



- from here, for every addition, rear will be incremented & element will be added to rear end.



- When rear is reached to SIZE minus 1
  - ↳ the queue is considered as FULL



Algorithm

Step 1: if rear = SIZE-1
then Write "Queue is Overflow"

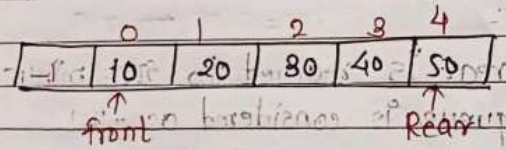
Step 2: rear = rear + 1

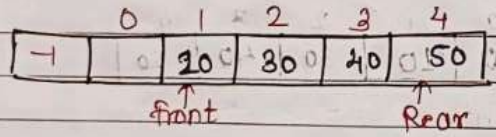
Step 3: Queue[rear] = X

Step 4: If front = -1
then front = 0

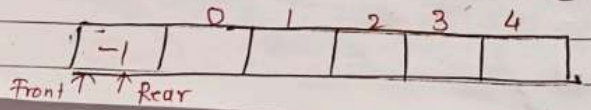
② Dequeue : Deleting an element from Queue

- The element at front is deleted & front is incremented by 1.





- If single element is present at time of deletion.
↳ then front & rear set to -1 (Empty)



Algorithm

Step 1: If front = -1
then Write "Queue is Underflow"

Step 2: Return Queue[front]

Step 3: If front = rear
then
front = -1
rear = -1
Else
front = front + 1

★ Implementation of Queue using Linked Organization

① Initialization and creation of node

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

struct node *F;
struct node *R;
```

② Insertion Operation

- The new element added becomes last element of queue.

Algorithm:-

step1: Create a new node pointer.

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

step2: If the queue is empty,
- then new node added will be both front & rear
- next point of F & R point to Null

```
ptr → data = val;
if (front == NULL)
{
    front = ptr;
    rear = ptr;
    front → next = NULL;
    rear → next = NULL;
}
else
{
    rear → next = ptr
    rear = ptr
}
    rear → next = NULL
```

③ Deletion Operation

- always first element of the queue is removed.

Algorithm:-

Step1:- check if queue is empty or not.
if front == NULL
print underflow message.

Step2:- if queue is not empty:
↳ delete front pointer element.
- for deleting copy node front → ptr
- make front pointer ~~poi~~ → front's next node
- free the node pointed as ptr.

```
void delete (struct Node * ptr)
{
    if (front == NULL)
    print ("Underflow")
    else
    {
        ptr = front;
        front = front → next
        free (ptr);
    }
}
```

④ Display function

Algorithm:

step 1: Check if queue contains at least one element or not

step 2: If queue is empty print
↳ "No elements — underflow"

step 3: Else,
- define node pointer *
- Initialize it with front.

step 4: Display data until next node becomes NULL

```
void Display()
{ struct node *temp;
  if ((front == NULL) && (rear == NULL))
  {
     print("underflow")
  }
  else
  { temp = front
    while (temp != NULL)
    {
       print (*temp → data);
       temp = temp → next
    }
  }
```

★ Circular Queues

Circular queue is a linear Data Structure in which the operations are carried out on the basis of FIFO (First In First Out) principal and the last position is connected back to the first position to make a circle.

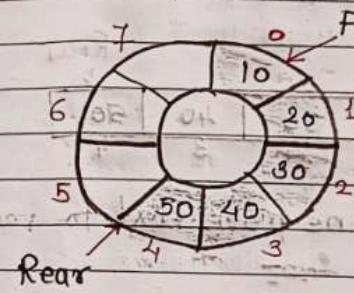


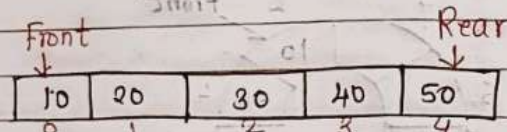
fig: Circular Queue

- In simple queue, when elements are deleted the front is incremented.
  - The spaces which get free after deletion can not be reutilized.
  - It lead to wastage of memory.

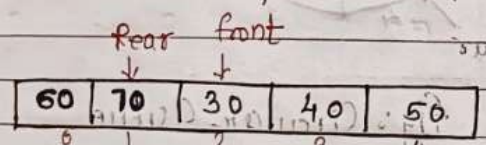
- To solve this problems
  ↳ DS provides circular queue.

① Array Representation of Circular Queue

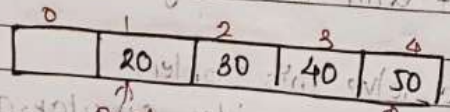
- Here when rear reaches to SIZE-1 position, it is not considered that queue is full.
- Only Two situations
  ① Front is at position 0 and Rear is at position SIZE-1

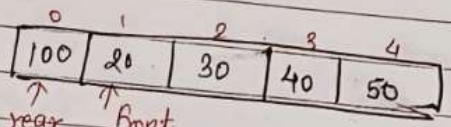


② Front is one place next to rear



- Here, while adding elements,
  - if rear reach to last position
  - if there are vacant position at the begin
  - then rear can shift to first position.





② Algorithm to Implement Circular queue

Step 1:- start
Step 2:- Show options 1. Insert 2. Delete 3. display
Step 3:- Accept choice
Step 4:- As per choice call a fun^n
enQueue(), deQueue(), display()
Step 5:- exit

③ Algorithm of Enqueue (Insertion)

Step 1:- If queue is full print msg & exit.
Accept element.
If rear is not at last position
Step 2:- To enqueue an element x into queue
Increment rear by 1
if rear is equal to n
set rear to 0
Step 3:- If front is -1
set front to 0
Step 4:- set queue[rear] to x.

④ Algorithm to Dequeue (Deletion)

To dequeue an element from the queue, do the following.

- Check if queue is empty by checking if front is -1 → Queue Empty.
- Set x to queue[front]
- If front is equal to rear
  set front & rear to -1
- otherwise
  - Increment front by 1
  - if front is equal to n then set front to 0
  - return x.

⑤ Advantages

- We can insert n numbers of elements new item to the location from where previous items is deleted.
- No memory wastage
- In circular queue,
  - we can insert n numbers of elements continuously but condition is that we must used deletion.
  - where as in simple queue continuously insertion is not possible.

⑥ Difference between Simple Queue & Circular Queue

Parameter	Simple Queue	Circular Queue
Presentation	Linear	Circular
memory	memory is wasted	No memory wastage
Front	Rear can not shift last to first position.	Rear can shift last to first position
Position	Rear is always next to front	Rear can be set previous to front.

★ Double Ended Queue (DEQUE)

- Double Ended Queue is data structure in which the insertion and deletion operations are allowed at both the ends (front and Rear)

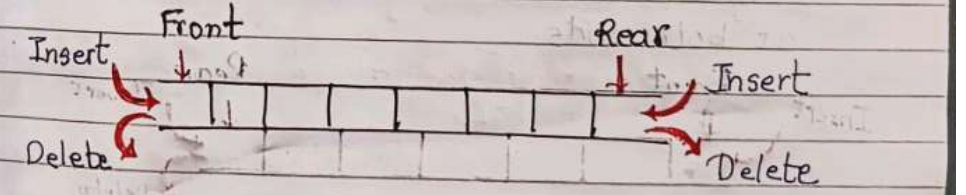


fig: Dequeue

① Representation of Deque

2 ways to represent DEQUE

1. Input restricted DEQUE
2. Output restricted DEQUE

* Input restricted DEQUE

- In this queue, the insertion operation is allowed at only one end while deletion operation is allowed at both the ends.

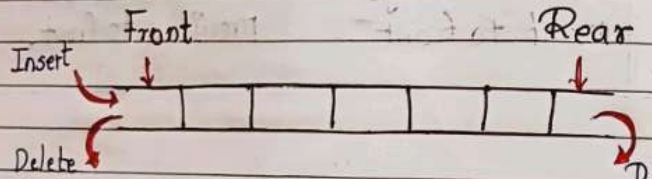


fig: Input Restricted DEQUE

* Output Restricted DEQUE

- In this type, deletion operation is allowed at only one end while insertion is allowed at both ends.

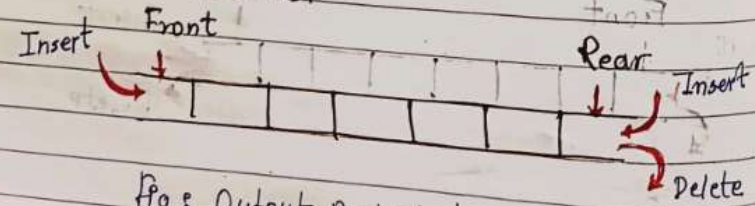


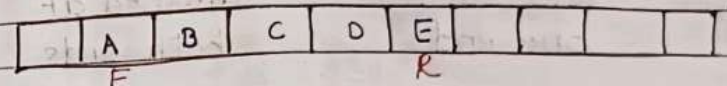
fig: Output Restricted DEQUE

Example:- Consider a dequeue given below which has LEFT = 1, Right = 5

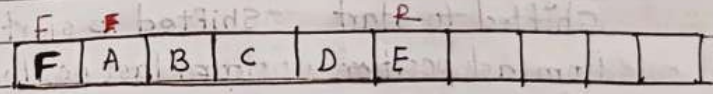
A B C D E

Now Perform following operations

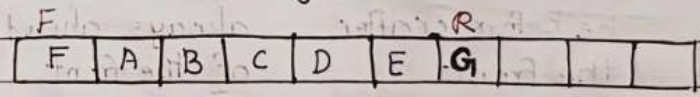
1) add F on the left
2) Add G on the right
3) Add H on the right
4) Delete 2 alphabets from left
4) Add I on right



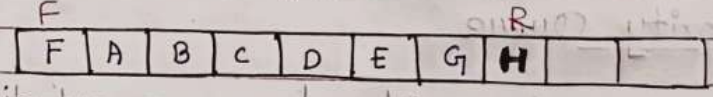
① Add F on the left



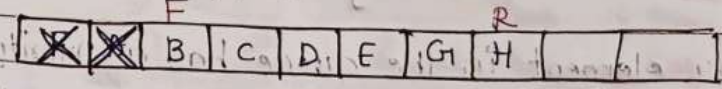
② Add G on the right



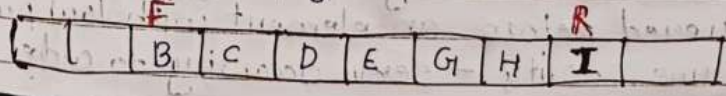
③ Add H on the right



④ Delete 2 alphabets from Left



⑤ Add I on the right



★ Difference between Circular Queue and Double-ended Queue

Parameter	Circular Queue	Double ended Queue
Insertion	allowed at one end	allowed at both ends
Deletion	allowed at one end	allowed at both ends
Rear pointer	Rear can be shifted to start from last position	Rear cannot be shifted to start from last position
Position	Rear Pointer can be before or after the front	Rear Pointer is always ahead of the front

★ Priority Queue

Priority Queue is considered as an extention of queue with following properties

① Every element in the queue has a priority associated with it
② An element with high priority is dequeued before an element with low priority
③ For same priority - Serve according to order

① Advantages
1) Simplicity
2) Reasonable support for priority.
3) Important task do not have to wait for completion of less priority task.
4) Suitable for applications with varying time and resource requirements.

② Applications
① used in Operating system for job scheduling.
② Graph algorithm
- Dijkstra's algo
- Prims algo
③ Data Compression - Huffman code
④ Heap sort

③ Types of Priority Queue

2 Types
- 1. Ascending Priority Queue
- 2. Descending Priority Queue.

★ Ascending priority Queue :- elements can be added randomly but only the removal of smallest element is allowed first.

2) Descending priority Queue :- removal of largest element is allowed first.

④ Elements of priority queue :-

- There may be numbers, characters or various types of complex structures which can be ordered on specific field.

for example :-

Records of telephone directory.

- The priority values need not be part of actual queue elements. These may be external values.

⑤ Implementation of priority Queue

* Using Array

```
struct data
{
    int element;
    int priority;
};
```

Insert() and delete() can be carried out

* Using Heaps

- Heap is a specialized tree-based data structure
- Performance of heaps is better as compared to arrays or LL.

⑥ Array Representation of priority queue :-

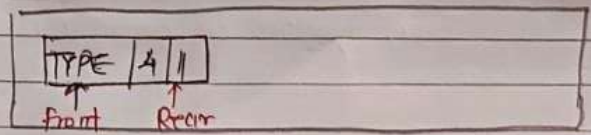
Rules :-

- The lower priority number indicates the higher priority.
- Add jobs in the queue and arrange them priority wise.
- When two jobs have same priority, they will be added order-wise.

add following jobs :-

- TYPE 4 1
- SWAP 3 2
- COPY 5 3

① add TYPE 4 1

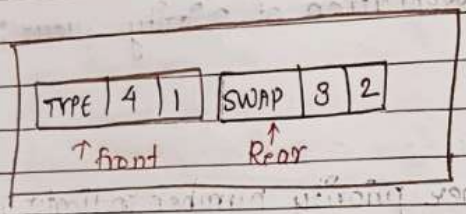


- front & rear points to job TYPE with priority 4.

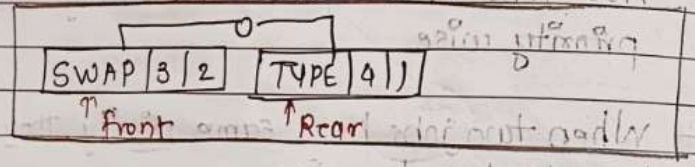
① Now add

SWAP	3	2

Here Job SWAP with priority 3.



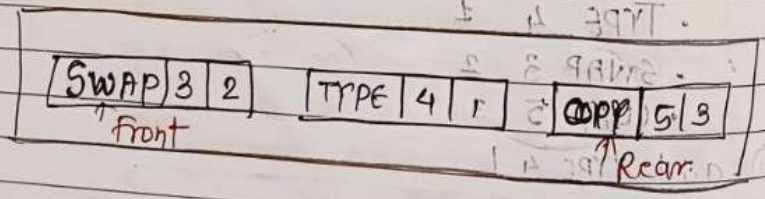
- The priority of SWAP is more that TYPE hence SWAP should be placed before TYPE



③ Now add

COPY	5	3

i.e Job COPY with priority 5



★ Algorithm of priority queue.

★ Algorithm of insert :-

Step 1 : If queue is full print the message and exit.
Step 2 : Accept element
step 3 : Search position for element as per priority by moving existing elements.
step 4 : Set new element at proper position.

★ Algorithm of delete :-

Step 1 : If queue is empty print the msg & exit.
step 2 : print front element as deleted.
step 3 : P = PQ[front]
step 4 : Increment front by 1.
step 5 : return P.

★ Algorithm to display :-

Step 1 : If queue is empty print the msg & exit
Step 2 : print elements from front to rear.

TREES

Tree is Data structure in which one node is connected with several nodes and in turn these several nodes are connected with several other nodes.

- Tree is very powerful as well as flexible DS as it's provides flexibility in traversing.
- Tree where one node is connected with several nodes, Hence we can jump from one to many other nodes

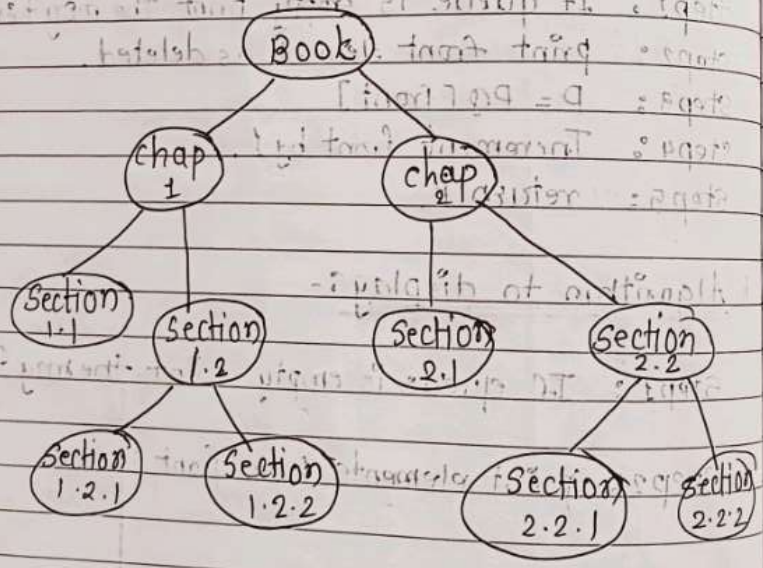


fig: Tree structure of book content

A Tree may be defined as a finite set 'T' of one or more nodes such that there is a node designated as the root of the tree and other nodes are divided into $n >= 0$ disjoint sets $T_1, T_2 \cdots T_n$ are called subtrees.

① Advantages of Tree

- Reflects structural relationships in bet^n data.
- Hierarchies can be represented by trees.
- Searching & Insertion can be done efficiently in trees.
- flexible DS, allowing to move sub-trees.

② Tree: Concept and Terminology.

Terminologies

1) Node
2) Root
3) Leaf nodes
4) Degree of node
5) Degree of Tree
6) Degree of level of a node
7) Depth / Height of a tree
8) Edge
9) path
10) parent
11) child
12) Ancestor
13) Descendents
14) Sibling
15) left & right subtree.