Stack & Queue

Stack:→ A stack is a list of elements in which an element may be inserted or deleted only at one end, called the top of the stack.

This means, in particular, that elements are removed from a stack in the reverse order of that in which they were inserted into the stack. Accordingly, stacks are also called Last-in-first-out (LIFO) lists.

Special terminology is used for 2 basic operations associated with stacks.

a) "Push" is the term used to insert an element into a stack.

b) "Pop" is the term used to delete an element from a stack.

Algorithm:→

PUSH (STACK, TOP, MAXSTK, ITEM)

S1:→ If TOP ≥ MAXSTK-1, then
Print "Overflow" & return

S2→ Set TOP = TOP+1

S3→ Set STACK[TOP] = ITEM

S4→ Return

2

POP (STACK, TOP, ITEM)

S1:→ If TOP = -1, then
Print "Underflow" & Return

S2:→ Set ITEM = STACK [TOP]

S3:→ Set TOP = TOP - 1

S4:→ Return

Polish Notation:→ The process of writting the operators of an expression either before their operands or after them is called the Polish Notation. The Polish notations are classified into 3 catagories.
i> Infix, ii> Prefix iii> Postfix.

i> **Infix:→** When the operators exists b/w 2 operands then the expression is called infix expression.
Ex- (a+b)*c, a+(b*c).

ii> **Prefix:→** When the operators are written before their operands then the resulting expression is called the prefix expression.
Ex:→ (A+B)*C = [+AB]*C = *+ABC
A+(B*C) = A+[*BC] = +A*BC
(A+B)/(C-D) = (+AB)/(-CD) = /+AB-CD

ii) <u>Postfix</u>:→ When the operators come after their operands the resulting expression is called the reverse Polish notation or Postfix expression.

<u>Ex</u>- $A+(B*C) = A+(BC*) = ABC*+$

<u>Evaluation of a Postfix Expression</u>:→

Suppose P is an arithmetic expression written in postfix notation.

<u>Algorithm</u>:→

<u>S1</u>:→ Add ')' at the end of P.

<u>S2</u>:→ Scan P from left to right & repeat S3 & S4 for each element of P until ')' is encountered.

<u>S3</u>:→ If an operand is encountered, Push it on stack.

<u>S4</u>:→ If an operator ⊗ is encountered, then

If an operator ⊗ is encountered, then

ⓐ Remove the 2 top element of stack, where A is top & B is next top element

ⓑ Evaluate B ⊗ A

ⓒ Push the result of ⓑ back on stack

end of if

end of loop

<u>S5</u> Exit.

4

Ex→ Consider the following arithmetic exp P wri
in postfix notation:

P: 5, 6, 2, +, *, 12, 4, /, −

Symbol	STACK
1) 5	5
2) 6	5, 6
3) 2	5, 6, 2
4) +	5, 8
5) *	40
6) 12	40, 12
7) 4	40, 12, 4
8) /	40, 3
9) −	37
10))	

Ex→ Consider the following arithmetic exp P, written in postfix notation:

P: 12, 7, 3, −, /, 2, 1, 5, +, *, +

Evaluate P using algo.

P: 12, 7, 3, −, /, 2, 1, 5, +, *, +

Symbol	STACK
1) 12	12
2) 7	12, 7
3) 3	12, 7, 3
4) −	12, 4
5) /	3
6) 2	3, 2
7) 1	3, 2, 1
8) 5	3, 2, 1, 5
9) +	3, 2, 6
10) *	3, 12
11) +	15
12))	

Transforming Infix Expression into Postfix Expression:→

Suppose Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent Postfix expression P.

S1:→ Push "(" onto STACK & add ")" to the end of Q.

S2:→ Scan Q from left to right & repeat steps 3 to steps 6 for each element of Q until STACK is empty.

S3:→ If an operand is encountered, add it to P.

S4:→ If a left parenthesis is encountered, push it onto STACK.

S5:→ If an operator ⊗ is encountered, then
(a) Repeatedly Pop from STACK & add to P each operator which has the same precedence as or higher precedence than ⊗
(b) Add ⊗ to STACK.
end of if

S6:→ If a ")" is encountered, then
(a) Repeatedly Pop from STACK & add to P each operator until a left parenthesis is encountered.

(b) Remove the "(". [Do not add '(' to P].
end of if.
end of loop.

S7:→ Exit.

Ex→ Consider the following arithmetic expression Q:

$$Q: A + (B * C - (D / E \uparrow F) * G) * H$$

Using the algorithm transform Q into its equivalent postfix expression P.

Soln:->

Symbol	STACK	Expression P
1) A	(	A
2) +	(+	A
3) (	(+(	A
4) B	(+(	AB
5) *	(+(*	AB
6) C	(+(*	ABC
7) -	(+(-	ABC*
8) (	(+(-(	ABC*
9) D	(+(-(	ABC*D
10) /	(+(-(/	ABC*D
11) E	(+(-(/	ABC*DE
12) ↑	(+(-(/↑	ABC*DE
13) F	(+(-(/↑	ABC*DEF
14))	(+(-	ABC*DEF↑/
15) *	(+(-*	ABC*DEF↑/
16) G	(+(-*	ABC*DEF↑/G
17))	(+	ABC*DEF↑/G*-
18) *	(+*	ABC*DEF↑/G*-
19) H	(+*	ABC*DEF↑/G*-H
20))	—	ABC*DEF↑/G*-H*+

8

Ex> Consider the following infix expression Q

Q : ((A+B) * D) ↑ (E−F)

Soln:>

Symbol	STACK	Expression P
1> (	((	
2> (	(((	
3> A	(((	A
4> +	(((+	A
5> B	(((+	AB
6>)	((	AB+
7> *	((*	AB+
8> D	((*	AB+D
9>)	(	AB+D*
10> ↑	(↑	AB+D*
11> (	(↑(	AB+D*
12> E	(↑(	AB+D*E
13> −	(↑(−	AB+D*E
14> F	(↑(−	AB+D*EF
15>)	(↑	AB+D*EF−
16>)	—	AB+D*EF−↑

Queue: A queue is a linear list of elements in which deletions can take place only at one end called the front & insertion can take place only at other end called rear.

Queue are also called First-in-first-out (FIFO) list, since the first element in a queue will be the first element out of the queue.

Linear Queue Insertion:

LQInsert(Queue, Element, Size)

S1 → If front = 0 & rear = Size-1 then
  Print "Overflow" & exit.

S2 → If front = -1 then
  front=0, rear=0

S3 → else
  rear=rear+1

S4 → Queue[rear]=element

S5 Return.

Deletion:->

LQDelete (Queue, Element)

S1:-> If front = -1 then
print "Underflow" & exit

S2:-> Element = Queue[front]

S3:-> If front = rear then
front = -1 & rear = -1

S4:-> else
front = front + 1

S5:-> Return (Element).

Circular Queue:-> Let we have an array Q that ~~contain~~ contains n elements in which Q[0] comes after Q[n-1] in the array. When this technique is used to construct a queue then the queue is called circular queue.

Insertion:->

CQInsert (Queue, Size, Item)

S1:-> If front = 0 & rear = size-1 or front = rear+1
then Print "Overflow" & exit

S2:-> If front = -1 then
front = 0 & rear = 0

S3 else if rear = Size-1 then
rear = 0

S4 else
rear = rear + 1

S5 Queue[rear] = element

S6 Return.

Deletion:→

CQDelete (Queue, size, Element)

S1:→ If front = -1 then
print "Underflow" & exit

S2:→ Element = Queue [front]

S3:→ If front = rear then
front = -1 & rear = -1

S4:→ else if front = size - 1 then
front = 0

S5:→ else
front = front + 1

S6:→ Return (Element)

Deque:→ A dequeue is a linear list in which elements can be added or removed at either end but not in the middle. The term deque is a contraction of the name double-ended queue.

There are 2 types of dequeue.

i) Input-restricted dequeue &

ii) Output-restricted deque.

i) **Input-restricted deque:→** An input restricted deque is a deque which allows insertions at only one end of the list but also allows deletions at both ends of the list.

(12)

ii) Output-restricted dequeue → An output-restricted deque is a deque which allows deletions at only one end of the list but allows insertions at both ends of the list.

Algorithm to insert any element in a deque :→

Insert_front() algo :→

Ins_f (Deque, item, size)

S1 :→ If front = 0 & rear = size-1 or front = rear+1 then
print "Overflow" & exit.

S2 :→ If front = -1 & rear = -1 then
front = 0 & rear = 0

S3 → Else if front = 0 then
front = size-1

S4 → Else
front = front-1

S5 :→ Deque[front] = item

S6 :→ Return.

Insert_rear() algo :→

Ins_r (Deque, item, size)

S1 :→ If front = 0 & rear = size-1 or front = rear+1 then
print "Overflow" & exit

S2 :→ If front = -1 & rear = -1 then
front = 0, rear = 0

S3 :→ Else if rear = size-1 then
rear = 0

S4 :→ Else
rear = rear+1

S5 :→ Deque[rear] = item

S6 :→ Return.

Algorithm to delete an element from the deque:→

delet_front() algo:→

del_f (Deque, item, size)

S1:→ If front = -1 & rear = -1 then
print "Underflow" & exit

S2:→ item = Deque[front]

S3:→ if front = rear then
front = -1 & rear = -1

S4:→ else if front = size-1 then
front = 0

S5:→ else front = front+1

S6:→ Return (item)

delet_rear() algo:→

del_r (Deque, item, size)

S1:→ If front = -1 & rear = -1 then
print "Undelflow" & Exit.

S2:→ item = Deque[rear]

S3:→ if front = rear then
front = -1 & rear = -1

S4:→ else if rear = 0 then
rear = size-1

S5:→ else
rear = rear-1

S6:→ Return (item).

Priority Queue:→ A priority queue is collection of elements such that each element has been assigned a priority & such that the order in which elements are deleted & processed comes from the following rules:

ii) An elements with the same priority are processed according to order in which they were added to the queue.

i) An element of higher priority is processed before any element of lower priority.

There are 2 types of priority Queue.

i) **Ascending Priority Queue:→** An ascending priority queue is a collection of items into which items can be inserted arbitrarily & from which only the smallest item can be removed.

A queue may be viewed as ascending priority queue whose elements are ordered by time of insertion. The element that was inserted 1st has the lowest insertion time value & is the only item that can be retrieved.

ii) <u>Descending Priority Queue</u> :→ An descending priority queue is a collection of items into which item can be inserted arbitrarily & from which only the largest item can be removed.

A stack may be viewed as descending priority queue whose elements are ordered by times of insertion. The element that was inserted last has the greatest insertion time value & is the only item that can be retrieved.