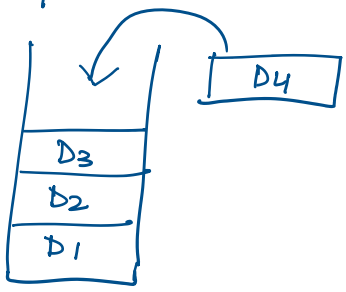
Stack and Queue

13 May 2024 15:48

STACK

(i) What is Stack?

→ A stack is a linear Data Structure that follows LIFO (Last-In-First-Out) principle. It can be defined as a container in which insertion & deletion can be done from the one end known as the top of the stack.



(ii) Operations implemented on Stacks –

(a.) push()
(b.) pop()
(c.) isEmpty()
(d.) isFull()
(e.) peek()
(f.) count()
(g.) change()
(h.) display()

(iii) Stack Overflow & Underflow –

Stack Overflow: When a stack is full & you try to push something onto it

Underflow: When a stack is empty & you try to pop something from it.

(iv) Applications of Stack –

(a.) String Reversal – To reverse a string, first push all the strings into a stack one-by-one until we reach null character
After that, start pulling all the characters one by one until we reach the bottom of the stack

(b.) UNDO/REDO – Whenever an action is performed, push the state onto the stack. When the user wants to undo an action, pop the state from the stack & if the user wants to redo, push the undone state back onto the stack.

(c.) Recursion – It means that the function is calling itself again. To maintain the previous states, the compiler creates a system stack in which all the previous records of the function are saved.

the previous states. The compiler creates a system stack in which all the previous records of the function are saved.

(d.) Balancing of Symbols - If the current character is a starting bracket then push it to the stack.
If the current character is a closing bracket then pop from stack & if the popped character is the matching starting bracket then fine else paranthesis is not balanced.

(e.) Backtracking - By using a stack for backtracking you can backtrack by undoing the choices made.

(v) Algorithm to PUSH items in a Stack

Step1 : Start
Step2 : Check whether the stack is full (top = size-1)
2.1 : If it is full then display "Stack full cannot insert".
Step3 : If the stack is not full then increment the top value by 1 and set stack[top] to value (top++)
stack[top] = value
Step 4 : End / Stop

(vi) Algorithm to POP items from a Stack

Step 1 : Start
Step 2 : Check whether the stack is Empty (top == -1)
2.1 : If the stack empty then display "Stack is empty, cannot delete"
Step3 : If the stack is not empty, then delete stack[top] & decrement top value by one (top --)
Step 4 : End / Stop

(vii) Algorithm to print all values from a Stack

Step1 : Start
Step2 : Check if the stack is empty
2.1 : Print "Stack is empty" and Stop
Step3 : While stack is not empty -
3.1 : Pop the top element from the stack

Step 3: While stack is not empty -

3.1: Pop the top element from the Stack

3.2: Print the popped element

Step 4: Stop / End

(viii) Prefix / Postfix / Infix

(a.) (a+b)x c

Infix - ab+c* ✓

Postfix - *+abc ✓

(b.) a x (b+c)

Infix - *a+bc ✓

Postfix - abc+* ✓

(c.) a+b x c-d

P - ab+cd-* ✗ I - *+ab-cd ✗

(d.) (a x b)-(c+d)

P - ab*cd+- ✓ I - -*ab+cd ✓

(e.) a x b x (c+d)

P - ab*cd+* ✓ I - **ab+cd ✓

(f.) (a+b) x (c-d)

P - ab+cd-* ✓ I - *+ab-cd ✓

(g.) a x b x c x d

P - abxc*d* ✓ I - ***abcd ✓

(h.) (a+b) x ((c+d)-e

P - ab+cd+*e- ✓ I - -*+ab+cde ✓

(i.) a+(b x (c-d))

P - abcd-*+ ✓ I - +a*b-cd ✓

(j.) ((a x b)+c) x d

P - ab*c+d* ✓ I - *+*abcd ✓

(k.) a+b+c+d+e

P - ab+c+d+e+ ✓ I - ++++abcd ✓

(l.) a x (b x (c x (d x e)))

P - abcde**** ✓ I - *a*b*c*de ✓

(m.) ((a+b) x c) -(d x e)

P - ab+c*de*- ✓ I- - *+abc*de ✓

(n.) a+b+c x d-e

P - ab+cd*+e- ✓ I- -++ab*cde ✓

(o) (a x b) + (c x d) -e

P - ab*cd*+e- ✓ I- -+*ab*cde ✓

(ix) Implementation using Stack

(A+ B/C *(D+C) -f)

Symbol	Stack	Postfix
C	C	
A	(	A
+	(+	A
B	C+	AB
/	(+/	AB
C	(+/	ABC
*	(+*	ABC/
(	(+*(	ABC/
D	(+*C	ABC/D
+	(+*C+	ABC/D
C	(+*C+	ABC/DC
)	(+*C+)	ABC/DC+
-	(+*-	ABC/DC+*+
F	(-	ABC/DC+*+F
)	(-)	ABC/DC+*+F-

(a+b)xCc-d) → Cd-C) x(b+a)

Symbol	Stack	Prefix
(	(	
D	(	D
-	(-	D
C	(-	DC
)	(-)	DC-
X	*	DC-
(	(	DC -
B	(*	DC-B
+	(*+	DC-B
A	(*	DC-BA
)	(*)	DC-B+A*

→ *+a b-cd

(x) Time Complexity of Stack

(a.) Push : Best - O(1) Worst - O(1)

(b.) Pop : Best - O(1) Worst - O(1)

(c.) Peek/Top : Best - O(1) Worst - O(1)

(d.) Search : Best - O(1) Worst - O(n)

(c) Peek/Top : Best - O(1) Worst - O(n)

(d) Search : Best - O(1) Worst - O(n)

(e) Space Complexity : Best - O(1) Worst - O(n)

QUEUE

(i) A queue can be defined as an ordered list which enables insert operations to be performed at one end called REAR & delete operations to be performed on the other end called FRONT.

FIFO : First-In-First-Out

(ii) Application of Queue

→ Used as waiting lists for a single shared resource like Printer, Disk, CPU

→ Used as buffers in most applications like MP3 Media Player, CP Player, etc.

→ Used to maintain playlist in media players

→ In operating systems for handling interrupts

TYPES OF QUEUE

① Simple Queue or Linear Queue

- Strictly follows the FIFO Rule

(i) Algorithm to Enqueue in Linear Queue

Step 1 : Start

Step 2 : If the Queue is full then return "Queue Overflow"

Step 3 : If the Queue is not full then place the new-item at the rear end of the queue

Step 4 : Move the rear pointer to the next position

Step 5 : End / Stop

(ii) Algorithm to Dequeue in Linear Queue

Step 1 : Start

Step 2 : If the Queue is empty then return "Queue Overflow"

Step1 : Start
Step2 : If the Queue is empty then return "Queue Overflow"
Step3 : Else, remove the item from the front-end of the Queue
Step4 : Move the front pointer to the next position
Step5 : End /Stop

② Circular Queue

(i) Algorithm to Insert data in Circular Queue

Step1 : Start
Step2 : If the rear is pointing to the position one before the front, then return "Queue Overflow"
Step3 : Else, place the new item at the rear end of the queue
Step4 : Update the rear pointer to the next Circular position
Step5 : End

(ii) Algorithm to delete data from Circular Queue

Step1 : Start
Step2 : If the front is equal to rear, return "Queue Underflow"
Step3 : Else, remove the item from the front end of the queue
Step4 : Update the front pointer to the next circular position
Step5 : End

③ Priority Queue

(i) Algorithm to Insert data in Priority Queue

Step1 : Start
Step2 : Create a newnode with the item & its priority
Step3 : If the priority queue is empty, Insert the newnode as the firstnode
Step4 : Else, traverse the priority queue to find the correct position to Insert the newnode based on its priority.
Step5 : Insert the newnode in its appropriate position
Step6 : End /Stop

(ii) Algorithm to delete data from Priority Queue

Step 1 : Start
Step 2 : If the Priority Queue is empty, return "Underflow"
Step 3 : Else, remove the node with the highest priority
Step 4 : Return the item stored in the removed node
Step 5 : End

(4) Deque or Double-Ended Queue

(i) Algorithm for Insertion at Front of Deque

Step 1 : Start
Step 2 : Create newnode containing the item to be inserted
Step 3 : If the Deque is empty, set both the front and rear to point to the newnode
Step 4 : Else, set the newnode's next pointer to point to the current frontnode and update the front pointer to point to the newnode
Step 5 : End

(ii) Algorithm for Insertion at Rear of Deque

Step 1 : Start
Step 2 : Create a newnode with data to be inserted
Step 3 : If the Deque is empty set both front and rear to point to the newnode
Step 4 : Else, set the current rear's next pointer to point to the newnode & update the rear pointer to point to the newnode
Step 5 : End

(iii) Algorithm for Deletion from Front of Deque

Step 1 : Start
Step 2 : If the deque is empty, return "Underflow"
Step 3 : Else, remove the node pointed to by the front pointer
Step 4 : Update the front pointer to point to the next node
Step 5 : End

(iv) Algorithm for Deletion from Rear of Deque.

Step 5 : End

(iv) Algorithm for Deletion from Rear of Deque

Step 1 : Start

Step 2 : If the Deque is empty, return "Underflow"

Step 3 : Else, remove the node pointed to by the rear pointer

Step 4 : Update the rear pointer to point to the previous node

Step 5 : Stop / End