TAPI DIPLOMA ENGINEERING COLLEGE, SURAT
COMPUTER ENGENEERING DEPARTMENT
SUBJECT: Data Structure and Algorithm (4330704)
VIRTUAL LAB

Practical-1 Stacks and Queues

AIM:- Gain a asic understanding of Stacks and Queues as an abstract data type.

Link to perform this experiment: - https://ds1-iiith.vlabs.ac.in/exp/stacks-queues/recap-arrays.html

THEORY:-
Time required: - Around 1.00 hours

Objectives:-
Gain a basic understanding of Stacks and Queues as an abstract data type
Understand stack and queue operations and associated time complexity through interactive animations
Understand applications of Stacks and Queues
Gain the concept of stacks
Understand the basic operations of stacks
Practice the operations of stacks
Gain the concept of Queues
Understand the basic operations of Queues
Practice the operations of Queues

Arrays:-
Arrays are a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.
Instead of declaring individual variables, such as value0, value1, ..., and value99, you declare one array variable such as value and use value[0], value[1], and ..., value[99] to represent individual variables. A specific element in an array is accessed by an index.
All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.
Arrays Visualization

Time and Space Complexity

Time complexity of an algorithm gives the measure of time taken by it to run as a function of the length of the input. Similarly, Space complexity of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input.

Recall that suppose our input is an array of N elements, and our algorithm iterates through the array once, time complexity will be O(N). If I run two embedded loops to traverse the array N times, time complexity will be O(N2).

Insertion : What would be the time for inserting an element into a stack? Since you're inserting an element to the top of the stack, time is constant O(1). Deletion : Similar to insertion, deletion takes contant time O(1) too. Search : How much time would it take to search for a particular element inside a stack? Searching for a particular element would mean traversing the entire stack in the worst case, so search operation takes linear time O(n).

Applications of Stacks

Expression Evaluation Stack is used to evaluate prefix, postfix and infix expressions.

Expression Conversion An expression can be represented in prefix, postfix or infix notation. Stack can be used to convert one form of expression to another.

Syntax Parsing Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code.

Parenthesis Checking Stack is used to check the proper opening and closing of parenthesis.

String Reversal Stack is used to reverse a string. We push the characters of string one by one into stack and then pop characters from stack.

Applications of Queues

Serving Requests

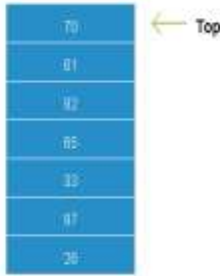
Serving requests on a single shared resource, like a printer, CPU task scheduling etc.

In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served.

Virtual Labs

Stacks and Queues

Observe: Stack A with random numbers is given. Observe how the prime numbers from stack A are popped and pushed into stack B whereas the others are pushed into stack C.



Stack A   Stack B   Stack C

Observations

Min. Speed   Max. Speed

Start   Reset   Pause

Virtual Labs

Stacks and Queues

Instructions

Observations

45 is pushed into the stack

Enter an element   Push   pop   Clear

PRACTICAL-2 Infix and Postfix

AIM:- This module explains Infix to Postfix conversion of expressions, and the evaluation of Postfix expressions.

Link to perform this experiment: - https://ds1-iiith.vlabs.ac.in/exp/infix-postfix/index.html.

THEORY:-

Time required: - Around 1hr 10min

Objectives:-

Formal Definitions of Infix and Postfix expressions.
Basic concepts behind Infix to Postfix conversion.
Conversion methods from Infix to Postfix.
Evaluation method of Postfix Expressions.

Definition of an Infix Expression

The operator is present between the operands on which it is acted upon. It is also the expression obtained from inorder travesal of an expression tree.

(A + B) * C
1 + 2 * 4 + 5 / 2 - 1

Definition of a Postfix Expression (RPN)

The operator is present after the operands that it is acted upon.
It is also the expression obtained from postorder traversal of an expression tree.
It is also called the Reverse Polish Notation (RPN).

2 3 + (RPN of 2 + 3)
A B + C * (RPN of (A + B) * C)

Algorithm:

1.Parenthesize the expression following operator precedence.
2.Priority(^) > Priority(/) > Priority(*) > Priority(+) >= Priority(-)
3.Starting from the lowest level in the parenthesized expression,
4.Rearrange the expression to RPN as (3 + 4) becomes (3 4 +).
5.Go to subsequent upper levels.
6.Thereby, convert the whole expression to RPN.

Conversion without Stack Example

Expression	Steps
A / B ^ C + D * E	1. Given infix expression
((A / (B ^ C)) + (D * E))	2. Move the operator next to the immediate closing paranthesis
((A B C ^ /) + (D E *))	3. Remove all paranthesis
A B C ^ / D E * +	4.Resultant postfix expression

Infix and Postfix

Instructions

Question: Validate the given expression: 84 ^ 82 - 95 / 97 + 41 * 31

Step 1: (84 ^ 82) - 95 / 97 + 41 * 31

Step 2: (84 ^ 82) - (95 / 97) + 41 * 31

Step 3: (84 ^ 82) - (95 / 97) + (41 * 31)

Step 4: ((84 ^ 82) - (95 / 97)) + (41 * 31)

Step 5: (((84 ^ 82) - (95 / 97)) + (41 * 31))

Observations:

Demonstration Finished

Min. Speed Max. Speed

Next Reset Pause

Algorithm for the Whole Conversion

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else:
4. 1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack (or the stack is empty or the stack contains a '('), push it.
5. 2 Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
6. If the scanned character is an '(', push it to the stack.
7. If the scanned character a is n ')', pop the stack and and output it until a '(' is encountered, and discard both the parenthesis.
8. Repeat steps 2-6 until infix expression is scanned.
9. Pop and output from the stack until it is empty.

PRACTICAL-3 Bubble Sort

AIM: - Given an unsorted array of numbers, generate a sorted array of numbers by applying Bubble Sort.

Link to perform this experiment: - https://ds1-iiith.vlabs.ac.in/exp/bubble-sort/analysis/stability-of-bubble-sort.html.

Time Required:- Around 1.00 hours

Objectives of the Experiment:-

Optimize the Bubble Sort algorithm to achieve better performance.

Demonstrate knowledge of time complexity of Bubble Sort by counting the number of operations involved in each iteration.

Compare Bubble Sort with other sorting algorithms and realize Bubble Sort as a stable comparison sorting algorithm.

Algorithm of Bubble Sort:-
Let's have a final look at the consolidated algorithm to sort an array of N elements:

STEP 1 : Compare the ith and (i+1)th element, where i=first index to i=second last index.

STEP 2 : Compare the pair of adjacent elements. If ith element is greater than the (i+1)th element, swap them.

STEP 3 : Run steps 1 and 2 a total of N-1 times to attain the final sorted array.

Analysis:-
Running Time of Bubble Sort

Lets assume that we are sorting N elements of a given array using SIMPLE Bubble Sort.

To complete one iteration, we traverse the array exactly once. Since we perform N-1 comparisons in an iteration, time complexity of completing one iteration is O(N).
In regular Bubble Sort, we run N-1 iterations, which is O(N), to sort our array. Hence overall time complexity becomes O(N*N). Note that even if array is fully sorted initially, regular Bubble Sort will take O(N2) time to complete.

Best and Worst Cases for Optimized Bubble Sort

For regular Bubble Sort, time complexity will be O(N2) in all cases.

For optimized Bubble Sort :

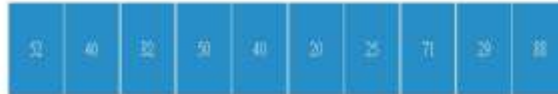
In best case scenario, we will have an already sorted array. We will have to run one iteration (N-1 comparisons) to determine this. Time complexity will be O(N) in this case.
In worst case (reverse sorted array) we will have to run N iterations to sort our array. Total comparisons performed will be (N-1)+(N-2)+(N-3)....+2+1. Hence overall time complexity becomes O(N2).
Try out the demo below and look out for the number of comparisons performed for sorted, reverse sorted and randomly generated array using optimized Bubble Sort. Notice how the number of comparisons always remains between O(N) and O(N2)!

Space Complexity of Bubble Sort

While swapping two elements, we need some extra space to store temporary values. Other than that, the sorting can be done in-place. Hence space complexity is O(1) or constant space.

Bubble Sort

Instructions



Observations

Min. Speed Max. Speed

Start Reset Pause