Jeremiah McDonald

28 February 2025

Algorithms

Stacks and Queues

1. **Stacking**: Given an empty stack, what will be the contents of the stack after the following operations?
   * push(8): The push() operation adds the element to the stack.

We push element 8 into the stack. Stack: [8]

* + push(2): We push element 2 into the stack. Stack: [8, 2]. The first element in the stack is 8 and the second element on top of the 8 is 2.
  + pop(): The pop() operation removes the top element of the stack, which is 2. Stack: [8]
  + push(pop() \* 2): First the pop() operation removes the element 8 from the stack, now the stack is currently empty []. Then it pushes (adds) the popped element (which is 8) \* 2 to the stack.

Stack: [16]

* + push(10): The push() operation adds 10 to the stack on top of 16. Stack [16, 10]
  + push(pop()/2): First the pop() operation removes the top element 10 from the stack leaving Stack: [16]. Then it divides the removed (popped) element (10) by 2 and adds that to the stack, Stack: [16, 5]

1. **Queueing**: Given an empty queue, what will be the contents of the queue after the following operations?

Errors in syntax: Stacks use push() to insert and pop() to remove.

Queues us enqueue() to insert(at the back) and dequeue() to remove(from the front).

* + enqueue(4): The enqueue() operation adds the element 4 to the back of the queue. Queue[4].
  + enqueue(dequeue() + 4): First the dequeue operation removes the element 4 from the queue leaving the queue empty, Queue: []. Then it takes the removed element (which was 4) and adds 4 to it before adding the new element to the queue, Queue: [8].
  + enqueue(8): The enqueue operation adds 8 to the back of the queue, Queue: [8, 8].
  + enqueue(dequeue()/2): First the element at the front of the queue (8) gets removed leaving Queue:[8]. Then it takes the removed element and divides that by 2 before adding the new element into the back of the queue, Queue[8, 4].
  + dequeue(): This removes the front element from the queue which is 8. The queue is now Queue:[4].
  + dequeue(): This removes the front element from the queue which is 4 leaving the queue empty, Queue: [].

1. **Find in deque**: We discussed that using a doubly-linked list, you are able to search and element in O(n/2), the same is true for a deque.

Given a Deque q and element y, provide an algorithm that finds the position in the deque in which the element x is stored in O(n/2).

int searchDeque(int y){

if(size == 0){

return;

}

Node leftNode = head;

Node rightNode = tail;

int left = 0;

int right = size – 1;

while(left <= right){

if(leftNode.data == y){

return left;

}

if(rightNode.data == y){

return right;

}

leftNode = leftNode.next;

rightNode = rightNode.next;

left++;

right--;

}

return -1; // if no element is found

}

**7. Algorithm Analysis**: For each of the algorithms you wrote for problems 4-6, explain their time complexity and space complexity using Big-O notation. Explain how you arrived at your answer.

**Question 4: isBalanced Algorithm:**

Time Complexity: Best Case: Worst Case: Final:

n is the length of the input String: O(n) O(n) Θ(n)

1. Iterating though the input string: O(n)
   1. Main for loop iterates through each character of the input string once.
2. Inside Loop:
   1. Constant Operations: O(1)
      1. Mathematical operations have constant time complexity.
   2. Adding an element to the “Stack”: O(1)
      1. If head is null: new element becomes head
      2. If head is not null: set temp = head, head = new element, new element points to temp (previous head).
   3. Removing an element from the “Stack”: O(1)
      1. Set temp pointer to head, updated head pointer to next element
      2. return the value of the removed element.

Space Complexity: Best Case: Worst Case: Final:

n is the length of the input string. O(n) O(n) Θ(n)

1. Space for LinkedList “Stack”: O(n)
   1. Stores at most n characters, every opening backet is added to the linked list.
   2. The variables c and top do not change with the size of the input making the space complexity O(1).

**Question 5: Decode Algorithm:**

Time Complexity: Best Case: Worst Case: Final:

j is the length of the input String:

n is the length of the output String: O(j) O(n) O(n)

1. Main Outer Loop: O(j) where j is the length of the input string
   1. Iterates through each character in the string.
2. Inside Loop:
   1. Constant Operation: O(1)
   2. Adding element to the LinkedList “Stack” (as stated above) O(1)
   3. Removing element from the LinkedList “Stack” (as stated above) O(1)
   4. Creating the tempString: O(k) where k is the length of the previous substring
      1. This converts a char array into a StringBuilder, which has to go through every char in the array.
   5. Repeat (the name I defined for the multiplier): O(repeat \*k)
      1. When encountering a closing bracket ‘]’, the current substring is repeated repeat times.
      2. In the worst case, the repeat value is large, and k is the length of the substring k.
      3. This means we can view this as the worst case for repeat is the length of the output string O(n)

The overall time complexity is O(j + n) Since j ≤ n in the worse case, O(j + n) = O(n).

Space Complexity: Best Case: Worst Case: Final:

j is the length of the input String:

n is the length of the output String: O(j) O(n) O(n)

1. Two LinkedLists “stacks” O(j)
   1. Stores numbers(multiplies) and substrings
   2. In the worst case every number continues to stack storing the length of the input j.
2. Output String: O(n)
   1. StringBuilder grows to the final output

Final Space complexity: O(j) + O(n) and since j ≤ n in the worst case, O(j + n) = O(n).