Data Structures and Algorithms Assignment 2

**Stack and queue**

1. Use examples to explain the sorting algorithms.

Sorting algorithms are a set of instructions that take an array or list as an input and arrange the items into a particular order.

Sorts are most commonly in numerical or a form of alphabetical (called lexicographical) order, and can be in ascending (A-Z, 0-9) or descending (Z-A, 9-0) order.

Some of the most common sorting algorithms are:

Selection Sort

Bubble Sort

Insertion Sort

Merge Sort

Quick Sort

Heap Sort

Counting Sort

Radix Sort

Bucket Sort

Let us go through some of the examples:

**Merger sort:**

Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

MergeSort(arr[], l, r)

If r > l

1. Find the middle point to divide the array into two halves:

middle m = l+ (r-l)/2

2. Call mergeSort for first half:

Call mergeSort(arr, l, m)

3. Call mergeSort for second half:

Call mergeSort(arr, m+1, r)

4. Merge the two halves sorted in step 2 and 3:

Call merge(arr, l, m, r)

# Python program for implementation of MergeSort

def mergeSort(arr):

if len(arr) > 1:

# Finding the mid of the array

mid = len(arr)//2

# Dividing the array elements

L = arr[:mid]

# into 2 halves

R = arr[mid:]

# Sorting the first half

mergeSort(L)

# Sorting the second half

mergeSort(R)

i = j = k = 0

# Copy data to temp arrays L[] and R[]

while i < len(L) and j < len(R):

if L[i] < R[j]:

arr[k] = L[i]

i += 1

else:

arr[k] = R[j]

j += 1

k += 1

# Checking if any element was left

while i < len(L):

arr[k] = L[i]

i += 1

k += 1

while j < len(R):

arr[k] = R[j]

j += 1

k += 1

# Code to print the list

def printList(arr):

for i in range(len(arr)):

print(arr[i], end=" ")

print()

# Driver Code

if \_\_name\_\_ == '\_\_main\_\_':

arr = [12, 11, 13, 5, 6, 7]

print("Given array is", end="\n")

printList(arr)

mergeSort(arr)

print("Sorted array is: ", end="\n")

printList(arr)

Bubble sort eg:

## Method Definition

def bubbleSort(array, n):

for i in range(n):

for j in range(0, n-i-1):

## Checking for adjacent elements

if array[j] > array[j+1]:

array[j], array[j+1] = array[j+1], array[j]

## Driver Code

array = [70, 20, 50, 30, 90, 5, 15]

n = len(array)

bubbleSort(array, n) ## Function Call

print("Sorted Array :")

for i in range(0,n):

print(array[i], end = " ")

Insertion sort eg.

## Method Definition

def insertionSort(array, n):

for i in range(1, n):

key = array[i]

j = i - 1

while j>=0 and key < array[j]:

array[j+1] = array[j]

j = j - 1

array[j+1] = key

## Driver code

array = [75, 90, 100, 95, 85, 50, 100, 110, 7]

n = len(array)

insertionSort(array, n)

print("Sorted Array")

for i in range(0,n):

print(array[i], end = " ")

2. What Are the Benefits of Stacks?

* Stack is easy to learn and implement for beginners.
* Stacks are used to solving problems that work on recursion.
* It allows you to control how memory is allocated and deallocated.
* Stack Helps you to manage the data in a Last In First Out(LIFO) method which is not possible with Linked list and array. When a function is called the local variables are stored in a stack, and it is automatically destroyed once returned.

3. What is the difference between a stack and a queue?

|  |  |
| --- | --- |
| **Stack** | **Queue** |
| Stacks are based on the LIFO principle, i.e., the element inserted at the last, is the first element to come out of the list. | Queues are based on the FIFO principle, i.e., the element inserted at the first, is the first element to come out of the list. |
| Insertion and deletion in stacks takes place only from one end of the list called the top. | Insertion and deletion in queues takes place from the opposite ends of the list. The insertion takes place at the rear of the list and the deletion takes place from the front of the list. |
| Insert operation is called push operation. | Insert operation is called enqueue operation. |
| Delete operation is called pop operation. | Delete operation is called dequeue operation. |
| In stacks we maintain only one pointer to access the list, called the top, which always points to the last element present in the list. | In queues we maintain two pointers to access the list. The front pointer always points to the first element inserted in the list and is still present, and the rear pointer always points to the last inserted element. |
| Stack is used in solving problems works on recursion. | Queue is used in solving problems having sequential processing. |

4. What are the different forms of queues?

There are four different types of queues:

* Simple Queue
* Circular Queue
* Priority Queue
* Double Ended Queue
* Simple Queue
* In a simple queue, insertion takes place at the rear and removal occurs at the front. It strictly follows the FIFO (First in First out) rule.
* Circular Queue
* In a circular queue, the last element points to the first element making a circular link.
* The main advantage of a circular queue over a simple queue is better memory utilization. If the last position is full and the first position is empty, we can insert an element in the first position. This action is not possible in a simple queue.
* Priority Queue
* A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority. If elements with the same priority occur, they are served according to their order in the queue.
* Deque (Double Ended Queue)In a double ended queue, insertion and removal of elements can be performed from either from the front or rear. Thus, it does not follow the FIFO (First In First Out) rule.

5. Why should I use Stack or Queue data structures instead of Arrays or

Lists, and when should I use them?.

* Well , it completely depends upon the application and the requirements. You can very well implement the LIFO and FIFO functionality using an array, but you are not sure as to the size of the array your app would require , so you end up using a stack/queue.
* Example 1: DFS for very large graph : you would require to use a stack as you donot know the number of nodes at each level.
* Example 2: BFS for large graphs: same reason as above( you donot know the numbers of neighbors a vertex may have)
* Example 3: Simulating an infinite capacity router : You donot know as to what maximum length queue can achieve during the simulation.
* Access order is a simple conceptual difference between array and stack/queue. Thats surely a benefit of these data structures over array if you want constrained access.
* Along with that memory allocation place is also an important factor.
* Efficient memory usage of your program greatly depend on what data structure you choose in you design.
* Array is always allocated on process stack. So if you need an array of size lets say 1kib, that much memory of you process stack is consumed. If you are working in kernel space (on Linux kernel mode stack is limited to 8 kib max), its always better to minimize stack size. So it is advised to use stack/queue or other data structures which can use heap memory with best efficiency.
* Talking about user space where you can have good lot of memory for process stack, it is occupied until process terminates. Where as if you use stack/queue or any other data structure, you can free that memory as soon as you are done with it. This improves memory utilization.

6. What is the significance of Stack being a recursive data structure?

* The answer of this question is the nature of the stack, it follow the last in first out simply(LIFO mechanism).
* In the recursion the same function calls itself when the last calls terminates it returned the value(if it has returned type if it does not have any return type then simply return to parent caller function) to the parent caller function again parent caller function returned the value to its parent function and so on…
* if we look at the return(value in case of return type function if it don’t have return type then only return towards the caller fuction) nature of recursion it also follows the last in first out (last called fuction comes out first from the programme) thats why we use the stack to maintain the hierarchy of the recursive methods(functions).