**The due date for submitting this assignment has passed**
Due on 2022-03-29, 23:59 IST.
**You may submit any number of times before the due date. The final submission will be considered for grading.**
Note: The assignment will be evaluated after the deadline passes. You will get your score/feedback after the deadline. Until then the score will be shown as [tbd].

1) Consider the Quick sort algorithm in which the partitioning procedure splits elements into two sub-arrays and each subarray contains at least one third of the elements. Let T(n) be the number of comparisons required to sort array of n elements. 1 point

- $T(n) <= 2T(n/3) + cn$
- $T(n) <= T(n/3) + T(2n/3) + cn$
- $T(n) <= 2T(n/3) + cn$
- $T(n) <= T(n/3) + T(n/3) + cn$

No, this answer is incorrect.

Score: 0

**Feedback:**

Each call of the Partition procedure splits the array into two subarrays with n/3 and 2n/3 elements, in this scenario partition takes O(n) time and Quicksort will apply on these subarrays again. Hence, recurrence T(n) = T(n/3) + T(2n/3) + cn holds.

Accepted Answers

$T(n) <= T(n/3) + T(2n/3) + cn$

2) Within functions correctly inserts an element after it creates a linked list, where each node of linked list is an object of class Node and a LL1 to the system stacks. 1 point

head is the first node in the linked list and value is the value that needs to be inserted after c node.
The function insertAfterHelper(head, value) is exactly similar to the beginning of the function insert.

```
class node:
    def __init__(self, x = None):
        self.value = x
        self.next = None

def insert(self):
    if(self.isEmpty()):
        return False;
    else:
        temp=self.pop()

def insertAfterHelper(head, value, c):
    if c == 0:
        insertAfterHelper(head, value)

    pos = head
    i=1
    while (pos!=None and i<c):
        pos = pos.next
        i=i+1

    if (i==c):
        temp = pos.next
        pos.next = Node(value)
        pos.next.next = temp
    else:
        raise Exception("Not enough nodes")
```

```
def insertAfterHelper(head, value, c):
    if c == 0:
        insertAfterHelper(head, value)

    pos = head
    i=1
    while (pos!=None and i<c):
        pos = pos.next
        i=i+1

    if (i==c):
        temp = pos.next
        pos.next = Node(value)
        pos.next.next = temp
    else:
        raise Exception("Not enough nodes")
```

```
def insertAfterHelper(head, value, c):
    if (c == 1):
        insertAfterHelper(head, value)

    pos = head
    i=1
    while (pos!=None and i<c):
        pos = pos.next
        i=i+1

    if (i==c):
        temp = pos.next
        pos.next = Node(value)
        pos.next.next = temp
    else:
        raise Exception("Not enough nodes")
```

```
def insertAfterHelper(head, value, c):
    if (c == 1):
        insertAfterHelper(head, value)

    pos = head
    i=1
    while (pos!=None and i<c):
        pos = pos.next
        i=i+1

    if (i==c):
        temp = pos.next
        pos.next = Node(value)
        pos.next.next = temp
    else:
        raise Exception("Not enough nodes")
```

No, this answer is incorrect.

Score: 0

**Feedback:**

Option (A) is incorrect because implementation doesn't insert node after 'c' position.

Option (B) is incorrect because due to the 'while' loop condition, 'i==c' new node will be added after 'c+1' position.

Option (D) is incorrect because it makes a little linked list one less than given position 'c'. Then the condition 'pos = pos.next' inside 'for' loop will cause an error.

Option (E) is incorrect because of this error in updated pseudo code 'c' does.

Accepted Answers

```
def insertAfterHelper(head, value, c):
    if (c == 1):
        insertAfterHelper(head, value)

    pos = head
    i=1
    while (pos!=None and i<c):
        pos = pos.next
        i=i+1

    if (i==c):
        temp = pos.next
        pos.next = Node(value)
        pos.next.next = temp
    else:
        raise Exception("Not enough nodes")
```

3) Which of the following code can be used to **enqueue**? 1 point

```
class Queue:
    def __init__(self):
        self.items = []
    def enqueue(self, key):
        self.items.append(key)
    def dequeue(self):
        return self.items.pop()
    def display(self):
        for i in range(len(self.items)):
            print(self.items[i])
```

```
class Queue:
    def __init__(self):
        self.items = []
    def enqueue(self, key):
        self.items.insert(0,key)
    def dequeue(self):
        return self.items.pop()
    def display(self):
        for i in range(len(self.items)):
            print(self.items[i])
```

```
class Queue:
    def __init__(self):
        self.items = []
    def enqueue(self, key):
        self.items.insert(len(self.items),key)
    def dequeue(self):
        return self.items.pop()
    def display(self):
        for i in range(len(self.items)):
            print(self.items[i])
```

```
class Queue:
    def __init__(self):
        self.items = []
    def enqueue(self, key):
        self.items.append(key)
    def dequeue(self):
        return self.items.pop(0)
    def display(self):
        for i in range(len(self.items)):
            print(self.items[i])
```

No, this answer is incorrect.

Score: 0

**Feedback:**

In Queue, an element is inserted(enqueue operation) from one side (after 'Rear' and) and removed(dequeue operation) from another side (after 'Front' or Queue's Base) in first-out (FIFO) concept. Option (A) is the correct implementation for queue.

Option (B) is wrong because enqueue and dequeue both operations are performing from one end.

Option (C) is wrong because an enqueue operation, 'insert(0, key)' takes linear operations, but for key in queue, key is inserted at the rear end.

Option (D) is incorrect because of the wrong dequeue operation, the wrong way of using 'del' operation with the list.

Accepted Answers

```
class Queue:
    def __init__(self):
        self.items = []
    def enqueue(self, key):
        self.items.insert(0,key)
    def dequeue(self):
        return self.items.pop(0)
    def display(self):
        for i in range len(self.items):
            print(self.items[i])
```

4) Consider a Quicksort implementation where we find the median and use the median as a pivot. Assume that we have a O(n) time find the median of an unsorted list. What will be the worst-case time complexity of this modified Quicksort with suitable assumptions? 1 point

- $O(n)$
- $O(n \log n)$
- $O(n \log n)$
- $O(n^2)$

No, this answer is incorrect.

Score: 0

**Feedback:**

If a median element is selected as a pivot, then the list will be divided into two equal lists (size of the two equal lists is floor(n/2) and ceil(n/2)). So, partition takes O(n) time for each partition and in (log2 n) steps (n, n/2, n/4, ..., 1). Quicksort will be completed. Hence, Option (c) is correct.

Accepted Answers

$O(n \log n)$

5) 1 point

```
def partition(arr, low, up):
    i = low-1
    j = up
    pivot = arr[low]
    while(1):
        while(arr[i]<=pivot and i<up):
            i = i+1
        while(arr[j]>pivot):
            j = j-1
        if(i<j):
            temp = arr[i] = arr[i], arr[j]
            i = i+1
            j = j-1
        else:
            break
    arr[low], arr[j] = arr[j], pivot
    return j
```

The above function is used as a partitioning function for sorting an array through Quicksort. What will be the value of Arr[j] after the call **partition(0, 10, 8, 10)**, (7, 2, 10), 6), 10) ?

- [6, 10, 7, 2, 4, 13, 16, 21, 32, 19]
- [4, 7, 2, 6, 10, 16, 19, 21, 32]
- [6, 10, 7, 2, 13, 19, 21, 32, 16]
- [10, 8, 10, 7, 2, 13, 21, 32, 19]

No, this answer is incorrect.

Score: 0

**Feedback:**

Option (B) is correct, execute the code sequence and observe the output.

Accepted Answers

[4, 7, 2, 6, 10, 16, 19, 21, 32]

6) **Linear probing** is an open addressing criteria in computer programming for resolving hash collisions in hash tables. Linear probing operates by looking the original hash index and adding successive values (mod hash) until a free slot is found.

An example sequence of linear probing is:

hash(key) = hash(key) % table_size

where mis a value of hash table, and hash(x) the hash function.

**Hash function:**

Let h(k) = k mod m be a hash function that maps an element k to an integer in {0, ..., m-1} where m is the size of the table. Let the i^ th probe position for a value k be given by the function

h(k,i) = (h(k) + i) mod m

The values of h(k), h(k+1), ..., h(k+m-1) for a fixed k form a full cycle of the integers from 0 to m-1. Hence, with the hash table after inserting at most m-1 elements, it is possible to find an empty slot by searching through all the slots in the table.

Consider inserting the keys 76, 93, 92, 65, 87, 137 into a hash table of size m = 11 using linear probing. Use primary hash function h(k)=k mod m. What will be the hash table after inserting all keys in given order? Suppose hash table is:
[None,None,None,None,None,None,None,None,None,None,None]

- [None,None,None,76,93,92,65,None,137,87,None]
- [None,None,None,76,93,92,65,137,None,87,None]
- [None,137,None,76,93,92,65,None,None,87,None]
- [None,None,None,76,93,92,65,137,None,87,None]

No, this answer is incorrect.

Score: 0

**Feedback:**

In linear probing, if the slots generated by the hash function is already occupied, then we will check the next location (by adding 1 until the free slot is not found. In the given question, initially all slots in the hash table are empty.

76 mod 11 = 5 which is free. So, 76 will be stored at the 5th index.

93 mod 11 = 5 which is free. So, 93 will be stored at the 6th index.

92 mod 11 = 5 which is free. So, 92 will be stored at the 7th index.

65 mod 11 = 5 which is not free. Hence, 65 will be stored at the 8th index.

87 mod 11 = 10 which is free. So, 87 will be stored at the 10th index.

137 mod 11 = 7 which is free. So, 137 will be stored at the 7th index.

76 mod 11 = 5 which is not free. [(7 - 5 mod 11) = 2] (approximately hash function) [= 2 mod 11 = 2] again slot is not free. [(= 2 mod 11) = 3 which is free. 76 will be stored at the 10th index.

Hence, option (b) is correct.

Accepted Answers

[None,None,None,76,93,92,65,137,None,87,None]

7) In a quicksort algorithm, we choose the last element of the array as pivot for partitioning. For which of the following array will the algorithm exhibit the worst-case behavior? 1 point

- [5, 3, 1, 7, 9, 13, 24, 46, 67]
- [67, 46, 24, 13, 9, 7, 1, 3, 5]
- [9, 5, 46, 3, 13, 67, 1, 24, 7]
- [9, 46, 7, 13, 67, 1, 24, 3, 5]

No, this answer is incorrect.

Score: 0

**Feedback:**

While input is sorted in any order then they partition procedure will divide the list into two sub-lists with 1 and n-1 elements, in that case, the number of partition levels will be n^2. And in the best case, the list will always divide into two equal sub-lists, due to the pivot selection. Hence, option (a) and (d) is incorrect and option (b) is the worst case behavior of quicksort.

Accepted Answers

[5, 3, 1, 7, 9, 13, 24, 46, 67]
[67, 46, 24, 13, 9, 7, 1, 3, 5]

8) In a heap tree, what is the upper bound for worst-case running time for finding the size of the tree? 1 point

- O(1)
- O(log n)
- O(n log n)
- O(n)

No, this answer is incorrect.

Score: 0

**Feedback:**

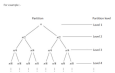
For finding the size of a binary tree, we have to traverse each element from starting. Hence, option (d) is correct.

Accepted Answers

O(n)

9) For Quicksort on an input array, when an element always selects the first element as a pivot, how many partitioning levels (required to sort an array) are there, respectively? 1 point

For example:-




- log n
- log n,log n
- n, n
- n, n

No, this answer is incorrect.

Score: 0

**Feedback:**

In the worst-case, the list will be divided into two sub-lists with 1 and n-1 elements, in that case, the number of partition levels will be 'n'. And in the best case, the list will always divide into two equal sub-lists, due to the pivot selection. Hence, option (d) is correct.

Accepted Answers

n,log n

10) Consider two lists L1 and L2 both containing n integers. You want to find if lists L1 and L2 are disjoint. Two lists are disjoint if there is no integer common in both the lists.

For example,

L1 = [1,2,3,4,5] L2 = [6,7,8,9,10] (L1 and L2 are disjoint)

L1 = [1, 2, 3, 4, 5] L2 = [5, 6, 7, 8, 9] (L1 and L2 are not disjoint as 5 is common in both the lists)

Assume that elements of both lists are not sorted. You want to find if lists L1 and L2 are disjoint. Which of the following statements is/are correct?

- For every integer in L1 search if it is present in L2 or not.
- Sort list L1 then for each integer in L2 use binary search to find if it is present in L1 or not.
- Sort both lists L1 and L2 then use merge like process to compare integers in both the lists.
- Sort list L1 then for each integer in L2 search if it present in L1 or not.

No, this answer is incorrect.

Score: 0

**Feedback:**

Option (A) is not minimal as if L1 are self-sorted and for L2, L1 remains, overall complexity will be O(n^2).

Option (B) sorting of list L1 will take O(n log n). Since then searching each element of L2 in L1 using binary search takes O(n log n). Hence, overall complexity will be O(n log n).

Option (C) sorting of list L1 and L2 will take O(n log n). Since then merge process takes O(n). Hence, overall complexity will be O(n log n).

Option (D) sorting of list L1 will take O(n log n). Since then searching each element of L2 in L1 is linear, takes O(n^2) time. Hence, overall complexity will be O(n^2).

Hence, Option (b) and (c) are the efficient solution.

Accepted Answers

Sort list L1 then for each integer in L2 use binary search to find if it is present in L1 or not.
Sort both lists L1 and L2 then use merge like process to compare integers in both the lists.