VIT®
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

Continuous Assessment Test key – 1

Course Name & code: **Data Structures and Algorithms & BCSE202L**
Programme Name & Branch : CSE

B2/TB2

Q.No.	Question	Max Marks

1.	a. Evaluate the Asymptotic recurrence relation for the given function
	
	int fib(int n) {
	if (n <= 1)
	{
	return n;
	}
	return fib(n - 1) + fib(n - 2);
	}
	
	When the value of n
	n is 0 or 1, Fibonacci number in constant time i.e.
	$T(n)=O(1)$ if $n\leq 1$
	
	There are two recurrence relations - one takes input n−1 and other takes n−2. Once we get the result of these two recursive calls, add them together in constant time i.e.
	$T(n)=T(n-1)+T(n-2)+O(1)$
	
	Combining case, get
	$T(n)=\{O(1)$ if $n\leq 1$
	$T(n-1)+T(n-2)+O(1)$ otherwise
	
	b. Find, the time complexity of subsequent recurrence relation, using the substitution method.
	$T(n)=\begin{cases}1 & n=0\\ 4T(n-1)+\log n & n>0\end{cases}$
	
	$O(4^n \log n)$
		10 (4+6)

2.	a) Recall, the Master's theorem for finding the time complexity of decreased recurrence relation.	10 (3+7)

For decreasing functions of the form

$T(n)=aT(n-b)+f(n)$
where:
n = input size (or the size of the problem)
a = count of subproblems in the decreasing recursive function
n-b = size of each subproblem (Assuming size of each subproblem is same)

Here a, b, and k are constants that satisfy the following conditions:

- a>0, b>0
- k>=0

Case 1) If a<1, then T(n) = θ(n^{k}) $T(n)=\theta(nk)$

Case 2) If a=1, then T(n) = θ(n^{k+1}) $T(n)=\theta(nk+1)$

Case 3) If a>1, then T(n) = θ(n^{n\over b} * f(n)) $T(n)=\theta(nbn*f(n))$

b) Having an array of unsorted elements, define an algorithm to search an element using binary search, and write the best and worst case of time-complexity with an example.

Binary Search Algorithm: The basic steps to perform Binary Search are:

- sort the given array in ascending order
- Begin with the mid element of the whole array as a search key.
- If the value of the search key is equal to the item then return an index of the search key.
- Or if the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half.
- Otherwise, narrow it to the upper half.
- Repeatedly check from the second point until the value is found or the interval is empty.

Binary Search Algorithm can be implemented in the following two ways

1. Iterative Method
2. Recursive Method

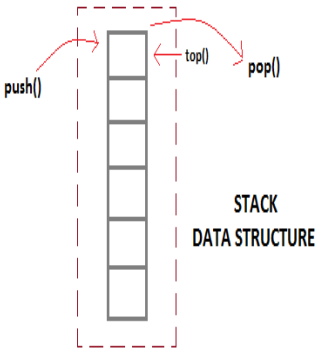
Time Complexity: O(log n)

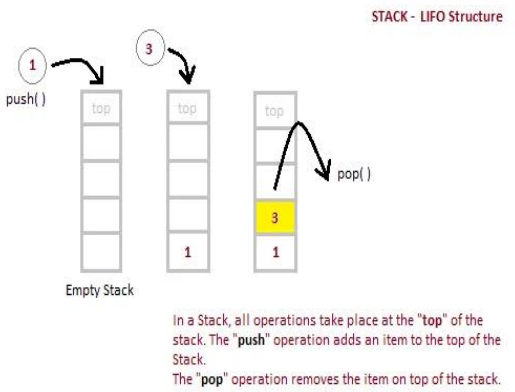
Best case - O (1)

Worst-case - O (logn)

3.	Covert the following infix expression to pre-fix & Postfix with its algorithm using stack in detail. x * y / (a * z) + i	10 (5+5)

	Algorithm of Infix to Prefix 1. Step 1. Push ")" onto STACK, and add "(" to end of the A 2. Step 2. Scan A from right to left and repeat step 3 to 6 for each eleme nt of A until the STACK is empty 3. Step 3. If an operand is encountered add it to B 4. Step 4. If a right parenthesis is encountered push it onto STACK 5. Step 5. If an operator is encountered then: 6. a. Repeatedly pop from STACK and add to B each operator (on the top of STACK) which has same 7. or higher precedence than the operator. 8. b. Add operator to STACK 9. Step 6. If left parenthesis is encontered then 10. a. Repeatedly pop from the STACK and add to B (each operator on top of stack until a left parenthesis is encounterd) 11. b. Remove the left parenthesis 12. Step 7. Exit **+ / * x y * a z i or + / * x y (* a z) i** Infix to Postfix Conversion **1.** Scan the Infix string from left to right. **2.** Initialize an empty stack. **3.** If the scanned character is an operand, add it to the Postfix string. **4.** If the scanned character is an operator and if the stack is empty push th character to stack. **5.** If the scanned character is an Operator and the stack is not empty, com precedence of the character with the element on top of the stack. **6.** If top Stack has higher precedence over the scanned character pop the s else push the scanned character to stack. Repeat this step until the stack empty and top Stack has precedence over the character. **7.** Repeat 4 and 5 steps till all the characters are scanned. **8.** After all characters are scanned, we have to add any character that the s may have to the Postfix string. **9.** If stack is not empty add top Stack to Postfix string and Pop the stack. Repeat this step as long as stack is not empty. **x y *(a z *)/ i + or x y * a z * / i +**	
4.	a)Explain, which one of the linear ADT suits well for reversing a string with its operations in detail. **Stack**	10 (7+3)






```
void Stack::push(int x)
{
   if(top >= 10)
   {
      cout << "Stack Overflow \n";
   }
   else
   {
      a[++top] = x;
      cout << "Element Inserted \n";
   }
}


int Stack::pop()
{
   if(top < 0)
   {
      cout << "Stack Underflow \n";
      return 0;
   }
   else
   {
      int d = a[top--];
      return d;
   }
}
```

b) Distinguish, the circular queue from Queue ADT.

S.NO.	LINEAR QUEUE	CIRCULAR QUEUE
1.	Arranges the data in a linear pattern.	Arranges the data in a circular ord where the rear end is connected wi front end.
2.	The insertion and deletion operations are fixed i.e, done at	Insertion and deletion are not fixe it can be done in any position.

		the rear and front end respectively.		
	3.	Linear queue requires more memory space.	It requires less memory space.	
	4.	In the case of a linear queue, the element added in the first position is going to be deleted in the first position. The order of operations performed on any element is fixed i.e, FIFO.	In the case of circular queue, the order of operations performed on an element may change.	
	5.	It is inefficient in comparison to a circular queue.	It is more efficient in comparison to linear queue.	
	6.	In a linear queue, we can easily fetch out the peek value.	In a circular queue, we cannot fetch out the peek value easily.	
	7.	Application- People standing for the bus.	Application- Computer-controlled traffic signal	
		Cars lined on a bridge.	In CPU scheduling and memory management.	

5. Consider the following numbers 23,34,12, 18,36,34,45, 56, 3. Using quick sort arrange them in ascending order. Explain the algorithm and time complexity for the same. — 10

23,34,12, 18,36,34,45, 56, 3

Pivot can be 23 or 3 …
Recursive call of function-based pivots index

```
function partitionFunc(left, right, pivot)
  leftPointer = left
  rightPointer = right - 1

  while True do
    while A[++leftPointer] < pivot do
      //do-nothing
    end while

    while rightPointer > 0 && A[--rightPointer] > pivot do
      //do-nothing
    end while

    if leftPointer >= rightPointer
      break
    else
      swap leftPointer,rightPointer
    end if

  end while

  swap leftPointer,right
```

```
  return leftPointer

end function
```

```
procedure quickSort(left, right)

  if right-left <= 0
    return
  else
    pivot = A[right]
    partition = partitionFunc(left, right, pivot)
    quickSort(left,partition-1)
    quickSort(partition+1,right)
  end if

end procedure
```

The average time complexity of quick sort is O(N log(N)).