**Theory Final Exam**

**Marks**

1. Write the time complexity of the following code segments with proper explanation. **10**

| void fun(int l,int r)  {  int mid = (l+r)/2;  for(int i = l ; i <= r ; i++)  {  cout<<i<<endl;  }  if(l<r){  fun(l,mid);  fun(mid+1,r);  }  }  int main()  {  int n;  cin>>n;  fun(0,n-1);  } |
| --- |
| for(int i = 1 ; i <= n/2 ; i++)  {  for(int j = 1 ; j <= n ; j = j + i)  {  cout<<i<<" "<<j<<endl;  }  } |

**Ans:**

**For 1st code block:**

This is a recursive function.

For level 1, loop has to iterate over left to right, n’th number of time

For level 2, left to mid & mid to right n/2 th time

For level 3, left to mid & mid to right (n/2) / 2 th time

………………………………………………………..

Worst case number of level or recursion k’ th time is ( n / (2^k) )

So recursive time complexity worst case, is O ( log2(n) )

Each level iteration time complexity worst case is O(n) // ignoring constant

Therefore total time complexity is = O ( n \* log2(n) )

**For 2nd code block:**

Outer Loop runs for i = 1 to i = n/2, and increment is i++  
Worst case the loop will run for n/2 times.

So time complexity in worst case = O(n/2) => O(n) // Ignoring the constant

The inner loop runs for j = 1 to j <= n in the worst-case.  
Here the increment is j = j + i,

for 1st iteration i = 1, j = 1, 2, 3, 4 ……. to j <= n, increment = j+1

for 2nd iteration i = 2, j = 1, 3, 5, 7 ……. to j <= n, increment = j+2

for 3rd iteration i = 3, j = 1, 4, 7, 10 ……. to j <= n, increment = j+3

……………………………………………..

Sum of this arithmetic series = ( n \* (n + 1) / 2 )

Time complexity in worst case = O( n \* log(n))

Therefore total time complexity = O(n) \* O( n \* log(n) ) => O (n^2 \* log (n) )

2. Suppose you are implementing a linked-list where you want to maintain a floating point number and a character in each node. Each node will contain a next pointer and also a next\_to\_next pointer that will keep track of the node that is next to the next node. What will the node class look like? **10**

class Node{

// write your variables

};

**Ans:**

#include<bits/stdc++.h>

using namespace std;

class Node

{

// write your variables

public:

float f;

char ch;

Node \* next;

Node \* next\_to\_next;

};

class CustomLinkedList

{

public:

Node \* head;

CustomLinkedList()

{

head = NULL;

}

Node \* create\_new\_node(float f1, char ch1)

{

Node \* newnode = new Node;

newnode->next = NULL;

newnode->next\_to\_next = NULL;

newnode->ch = ch1;

newnode->f = f1;

return newnode;

}

void insert\_at\_head(float f1, char ch1)

{

Node \* newnode = create\_new\_node(f1, ch1);

if(head == NULL){

head = newnode;

return;

}

Node \* tempnode = head;

Node \* next\_tempnode = head->next;

head = newnode;

head->next = tempnode;

if(next\_tempnode != NULL){

head->next\_to\_next = next\_tempnode;

}

}

void print\_all()

{

if(head == NULL){

cout << "Empty!\n";

return;

}

Node \* a = head;

if(a->next\_to\_next != NULL){

cout << "Next to next Float " << a->next\_to\_next->f << " Next to next Char " << a->next\_to\_next->ch <<"\n";

}

while(a != NULL){

cout << "Float " << a->f << " Char " << a->ch <<"\n";

a = a->next;

}

}

};

int main()

{

CustomLinkedList cll;

cll.insert\_at\_head(4.1, 'a');

cll.insert\_at\_head(4.2, 'b');

cll.insert\_at\_head(4.3, 'c');

cll.insert\_at\_head(4.4, 'd');

cll.insert\_at\_head(4.5, 'e');

cll.print\_all();

return 0;

}

3. Write the main difference between linear and non-linear data structures. Compare between Stack, Queue and Deque. Are stack, queue, deque linear or non-linear data structure? What about a tree? **10**

**Ans:**

The main difference difference between linear and non-linear data structure is,

In a linear data structure, the data elements follow sequential memory allocation. Each element must take the next memory space from the previous element. For example integer array elements take 4 bytes of space if int a[1] takes the address 1234 then next integer int a[2] must take address 1238. In linear data structure we can traverse through each element without the necessity to know each element address.

In a non-linear data structure the data elements or nodes are scattered throughout the memory and are not organized in a sequential pattern. The only way to traverse the next node is to have a pointer pointed at the next node’s memory address. Linked list Tree etc data structure uses non-linear memory allocation.

Stack, Queue and Deque are linear data structure. The comparison between Stack, Queue, and Deque is given below

|  | Stack | Queue | Deque |
| --- | --- | --- | --- |
| Follows | Last In First Out | First in First Out | Both |
| Operations | push(), pop(), top() | enqueue(), dequeue(), front() | push\_back(), push\_front(),  pop\_back(),  pop\_front() |
| Used | Undo redo, recursive function call | Bill payment, line | Algorithms that need both stack and queue functions |

Stack, Queue and Deque are linear data structures as they all follow sequential memory allocation rule.

A Tree is a non-linear data structure, the nodes of a tree do not sequentially vacate memory locations. They are connected by pointers.

4. Between singly linked list and doubly linked list which is better for implementing Stack and Queue? What about Deque? **10**

**Ans:**

For implementing stack, A singly linked list is sufficient and memory efficient. Although we can implement stack using doubly linked list also.

Stack has mainly three major operations, push(), pop() and top(). Which we can easily implement using a singly linked list. For push() operation we can use insert\_at\_head() function where we need to know the next node of head, but not necessarily the previous node of any node. Similarly for pop() we can use delete\_at\_head() and for top() we can show the head -> data.

Similarly a singly linked list is enough to implement queue.

For Queue we have three major operations, enqueue(), dequeue() and front(). We need to maintain head pointer as well as tail pointer. For enqueue() operation we need to push after tail and update tail. For dequeue() operation we need to use delete\_at\_head(). And similarly we can return head -> data for front().

And for a Deque we need to use a Doubly linked list.

Deque has all the functionalities of Stack and Queue combined. In this case we need to have some major operations, they are push\_front(), push\_back(), pop\_front(), pop\_back(), front\_item(), tail\_item(). As we can see that Dequeue needs to do operations from both ends front and back, we need to have next and previous pointer pointing next and previous node of our current node. A doubly linked list has next and previous node that can help implementing these operations.

This is why a Doubly linked list is ideal for implementing Deque.

5. Convert the infix expression to postfix expression using a stack. You need to show all the steps. **10**  
 **a\*b+c\*d+e**

**Ans:**

The Algorithm to convert the infix expression to postfix expression:

1. Traverse the expression from left to right
2. If operand found add to answer string
3. If ‘(‘ found push to stack
4. If ‘)’ found pop from stack and add to answer string until ‘(‘ found
5. If operator found pop from stack and add to answer string until  
   an operator with less precedence is found

Infix Expression: a \* b + c \* d + e

| step | Symbol | Stack | Postfix |
| --- | --- | --- | --- |
| 1 | a |  | a |
| 2 | \* | \* | a |
| 3 | b | \* | ab |
| 4 | + | + | ab\* |
| 5 | c | + | ab\*c |
| 6 | \* | + \* | ab\*c |
| 7 | d | + \* | ab\*cd |
| 8 | + | + | ab\*cd\*+ |
| 9 | e | + | ab\*cd\*+e |
| 10 |  |  | ab\*cd\*+e+ |

6. Compare the memory usage of Array, Singly Linked-list and Doubly Linked-list with necessary explanation. **10**

**Ans:**

The comparison between memory usage of Array, Singly Linked List and Doubly Linked List is explained below

**Arrays:**

Arrays are more memory efficient than linked list in case of memory space allocation. They take contagious memory locations. This makes accessing arrays most efficient. However they are fixed size and need to dynamically change the size of array and copy that elements which can be difficult. Also if the array size is 4 and there is no 4 adjacent blocks are available in the memory arrays cannot store data.

**Singly Linked List:**

Singly linked list takes more space than arrays. It needs to have a additional pointer to point the next node location. However they do not need to use contagious memory locations to store data. They also can increase size as needed as they are not fixed size. But accessing a node in the singly linked list is more difficult as we need to traverse the full linked list.

**Doubly Linked List:**

Doubly linked list takes more space than a singly linked list as it needs to store additional two pointers next node and previous node. It has all the properties of a singly linked list. Although it is the most memory consuming This provides bidirectional traversal capabilities at the cost of increased memory usage.

7. Suppose you are implementing a stack in a scenario where numbers are added in sorted order so that the stack is always sorted. Sometimes you need to quickly search if a value exists in the stack or not. Array or Linked-list which implementation for stack will you prefer in this scenario? Give necessary explanations. **10**

**Ans:**

In case of the scenario where numbers are added in sorted order so that the stack is always sorted I think Array based Stack implementation would be better if we need to quickly search if a value exists in the stack or not.

Given that the stack will be sorted, if we apply Array based implementation we can use Binary Search to check if a value exists on the stack. The time complexity will be worst case O(log(n))

But if we use Linked list based Stack implementation, we have to traverse the full stack every time to search a value. Because even if the stack is pushed in a sorted order the nodes are not sequentially organized in memory so we can not access it like an array. We need to traverse them. Worst case complexity will be O(n).

Thats why I think Array based stack implementation is best suited for this particular scenario.

8. Suppose you are maintaining a head and tail for a singly linked-list. What will be time complexity of **10**

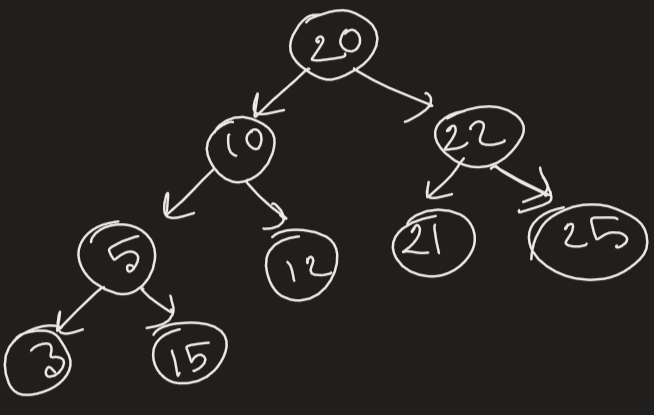
* 1. Inserting a value at the beginning
  2. Inserting a value at the end
  3. Deleting a value at the beginning
  4. Deleting a value at the end
  5. Inserting a value at the mid point
  6. Deleting a value at the mid point

**Ans:**

If we maintain head and tail for a singly linked-list the time complexity of

* 1. Inserting a value at the beginning = O(1)
  2. Inserting a value at the end = O(1)
  3. Deleting a value at the beginning = O(1)
  4. Deleting a value at the end = O(1)
  5. Inserting a value at the mid point = O(n/2) => O(n)
  6. Deleting a value at the mid point = O(n/2) => O(n)

9. Consider the following binary tree in **Fig 1** (node 20 is the root) and answer the given questions. **10**



**Fig: 1**

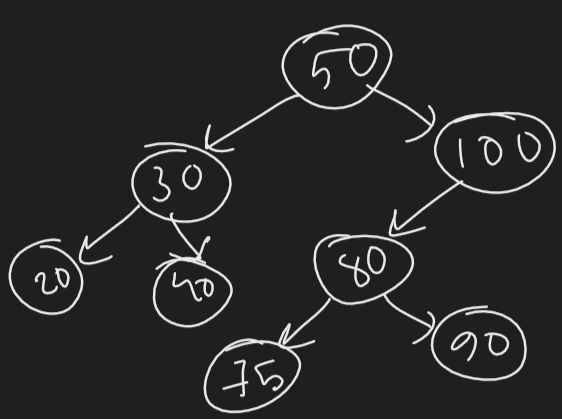
1. Is the tree a Perfect binary tree? Why or why not?
2. Is the tree a Complete binary tree? Why or why not?
3. Is the tree a Binary search tree? Why or why not?
4. Write down the BFS, inorder, preorder and postorder traversal of the tree.

**Ans:**

The Given binary tree question answers are following:

1. Is the tree a Perfect binary tree? Why or why not?  
   - It is not a perfect binary tree. Because a perfect binary tree must have all the levels completely filled. Here level 0 is filled, level 1 is filled but level 2 is not completely filled
2. Is the tree a Complete binary tree? Why or why not?  
   - It is a complete binary tree.Because in a complete binary tree each level must be filled except for the last level. Here level 0 and level 1 are completely filled.
3. Is the tree a Binary search tree? Why or why not?  
   - It is a Binary Search Tree, because all the elements < parent(20) are left side  
   of the parent and all the elements >= parent are on the right side.
4. Write down the BFS, inorder, preorder and postorder traversal of the tree.  
   Inorder- 3, 5, 10, 12, 20, 21, 22, 25  
   Preorder - 20, 10, 5, 3, 12, 22, 21, 25  
   Postorder - 3, 5, 12, 10, 21, 25, 22, 20

10. Write the steps to insert **70** in the following binary search tree in **Fig 2** (node 50 is the root). **10**



**Fig: 2**

**Ans:**

The Insertion Algorithm of BST is,

1. Check the root with value  
   IF value >= root go Right child of root  
   IF value < root go Left child of root
2. Keep on Traverse   
   If value >= root and right child empty, put value to the Right  
   If value < root and left child empty, put value to the Left

| step | value | parent | left | right |
| --- | --- | --- | --- | --- |
| 1 | 70 | 50 | 30 | 100 |
| 2 | 70 | 100 | 80 | NULL |
| 3 | 70 | 80 | 75 | 90 |
| 4 | 70 | 75 | NULL | NULL |
| 5 |  | 75 | **70** | NULL |