CS 331 Final Exam
Question Paper

Section 1    Objective Question

This section contains fill-in the blank and true/false questions  10 * 2 = 20 Marks

1. The worst-case running time for quicksort is ________

2. Quicksort is a well-known sorting algorithm developed by C. A. R. Hoare on average, makes ________ comparisons to sort n items

3. Merge sort has an average case performance of__________.

4. Finding the shortest path from one node to another, like ______________

5. _________________ is used for finding minimum spanning trees

6. Linked list enables dynamic storage. True/False

7. Arrays enable static storage. True/False

8. An example of the divide and conquer algorithmic is merge sort. True/False

9. The root node of a tree has n/2 parents. True/False

10. Recursion is implemented using stacks. True/False

Section 2    Essay Questions

This section contains long answer questions. 3 * 10 = 30 Marks

1. Define singly linked list. Explain singly linked list with an example.

2. Explain the logic of Merge Sort with its algorithm. Also give its complexity analysis.

3. Define breadth-first search and depth first search algorithm. Give the pseudo code for each.

-------X---X---X-------

End of Question Paper
Section 1  Objective Question

Answers
1. The worst-case running time for quicksort is $O(n^2)$
2. Quicksort is a well-known sorting algorithm developed by C. A. R. Hoare on average, makes $O(n \log n)$ comparisons to sort $n$ items
3. Merge sort has an average case performance of $O(n \log n)$.
4. Finding the shortest path from one node to another, like Dijkstra's algorithm
5. Prim's algorithm is used for finding minimum spanning trees
6. Linked list enables dynamic storage T/F == T
7. Arrays enable static storage T/F == T
8. An example of the divide and conquer algorithmic is merge sort T/F = T
9. The root node of a tree has $n/2$ parents T/F == F
10. Recursion is implemented using stacks T/F ===F

Section 2 Essay Questions
Answers

1. Define singly linked list. Explain singly linked list with an example.

Ans. A linked list is a series of nodes in memory such that:

1. There is a starting node.
2. Each node contains a pointer that points to the next or child node.
3. If a node does not have a child node then its pointer is set to NULL.
4. Each node contains data, maybe a lot of it.
5. The linked list also has functions that manage the list by performing additions, deletions, changing the data of a node, returning the number of nodes, etc.

The singly-linked list is the most basic of all the linked data structures. A singly-linked list is simply a sequence of dynamically allocated objects, each of which refers to its
successor in the list. Figure below shows several of the most common singly-linked list variants.

Figure (a) is a **basic singly-linked list**. Each element of the list refers to its successor and the last element contains the null reference. One variable, labeled head is used to keep track of the list. While it is easy to add elements at the head of the list, to add elements at the other end (the tail) we need to locate the last element. If the basic singly-linked list is used, the entire list needs to be traversed in order to find its tail.

Figure (b) shows a way in which to make **adding elements to the tail** of a list more efficient. The solution uses a second variable, tail, which refers to the last element of the list. This time efficiency comes at the cost of the additional space used to store the variable tail.

Figure (c) illustrates **two common programming tricks**. There is an extra element at the head of the list called a sentinel. This element is never used to hold data and it is always present. The principal advantage of using a sentinel is that it simplifies the programming of certain operations.

Figure (d) shows a variation in which a single variable is used to keep track of the list, but this time the variable, tail, refers to the last element of the list. Since the list is circular in this case, the first element follows the last element of the list. Therefore, it is relatively simple to insert both at the head and at the tail of this list. This variation minimizes the storage required, at the expense of a little extra time for certain operations.

2. Explain the logic of Merge Sort with its algorithm. Also give its complexity analysis.
Ans. Merge Sort is a comparison-based sorting algorithm. It is stable, meaning that it preserves the input order of equal elements in the sorted output. It is a sorting algorithm that is based on divide and conquer algorithm design paradigm. The Divide and Conquer methodology is as follows:

**Divide:**
- Divide input data I into two disjoint subsets I1, I2
- Solve the sub problems associated with I1, I2

**Conquer:**
- Combine the solutions for I1, I2 into a solution for I

The **Merge sort** algorithm is as follows:

```pseudo
mergeSort (S, C)
- Input: sequence S with n elements and comparator C
- Output: sequence S sorted according to C
- If S.size () > 1
  (S1, S2) ← partition (S, n/2)
  mergeSort (S1, C)
  mergeSort (S2, C)
  S ← merge (S1, S2)
```

**Merge algorithm:**

The algorithm for merging two sorted arrays is as follows:
- Start.
- Compare the values in the arrays of size one each.
- Combine the two arrays such that the resultant array is sorted.
- Keep repeating the process until the result is a single sorted array.
- End.

**Analysis of merge sort:**

a) The height h of merge sort tree is O(log n)
   - At each recursive call we divide the sequence into half
b) The overall amount of work done at the nodes of depth i is O(n)
   - We partition and merge 2i sequences of size n/2i
   - We make 2i+1 recursive calls
c) Thus the total running time of merge sort is O(n log n)

3. Define breadth-first search and depth first search algorithm. Give the pseudo code for each.
Ans. **Breadth-first search** (BFS) is a graph search algorithm that begins at the root node and explores all the neighbouring nodes. Then for each of those nearest nodes, it explores their unexplored neighbour nodes, and so on, until it finds the goal.

Pseudo code:

```cpp
BFS(const std::vector<Vertex>& graph, int start, int end) {
    std::queue<int> next;
    std::map<int, int> parent;
    parent[start] = -1;
    next.push(start);
    while (!next.empty()) {
        int u = next.front();
        next.pop(); // Here is the point where you can examine the u-th vertex of graph
        // For example:
        if (u == end) return true;
        for (std::vector<int>::const_iterator j = graph[u].out.begin(); j != graph[u].out.end(); ++j) {
            // Look through neighbors.
            int v = *j;
            if (parent.count(v) == 0) {
                // If v is unvisited.
                parent[v] = u;
                next.push(v);
            }
        }
    }
    return false;
}
```

**Depth-first search** (DFS) is an algorithm for traversing or searching a tree, tree structure, or graph. Intuitively, one starts at the root (selecting some node as the root in the graph case) and explores as far as possible along each branch before backtracking.

Pseudo Code:

```cpp
dfs(graph G) {
    list L = empty
    tree T = empty
    choose a starting vertex x
    search(x)
    while(L is not empty) {
        remove edge (v, w) from beginning of L
    }
}
```
if w not yet visited
    { 
    add (v, w) to T
    search(w)
    }
}

search(vertex v)
{
    visit v
    for each edge (v, w)
        add edge (v, w) to the beginning of L
}