Solomiya Pobutska
Final Exam CISC 3130
05.20.20

I.

```
public class BinaryTree {
    public int element;
    public node left_node;
    public node right_node;
    public static node Node(int val)
    {
        node new_node = new node();
        new_node.element = val;
        new_node.left_node = null;
        new_node.right_node = null;
        return new_node;
    }

    public static int delete_Leaf_node(node root) {
        if (root == null)
            {
            return 0;
            }
        if (root.left_node == null && root.right_node == null) {
            root = null;
            return 1;
            }
        else {
            return (delete_Leaf_node(root.left_node) +
delete_Leaf_node(root.right_node));
          }
    }

    public static int find_Node_Count(node root) {
        if (root == null) {
            return 0;
        }
        else {
            return (find_Node_Count(root.left_node) + 1 +
find_Node_Count(root.right_node));
        }
    }
    public static void main(String args[]) {
        int count = delete_Leaf_node(root);
        int total = find_Node_Count(root);
        System.out.print("Deleted nodes are : " + count );
        System.out.print("Nodes left : "+ (total - count));
    }
}
```

II.

```
public  class Linkedlist {
        node head;
        node sort_node;

class node {
    int data;
    node nextptr;

    public node(int data) {
        this.data = data;
    }
}

    void push(int data) {
        node node_data = new node(data);
        node_data.nextptr = head;
        head = node_data;
    }

      void Sort(node headref) {
        sort_node = null;
        node curr = headref;

        while (curr != null)
        {
            node nextptr = curr.nextptr;
            sortInsert(curr);
            curr = nextptr;
        }
               head = sort_node;
    }

     void sortInsert(node node_ptr)  {

        if (sort_node == null || sort_node.data >= node_ptr.data)
        {
            node_ptr.nextptr = sort_node;
            sort_node = node_ptr;
        }
        else
        {
            node curr = sort_node;
            while (curr.nextptr != null && curr.nextptr.data < node_ptr.data)
            {
                curr= curr.nextptr;
            }
            node_ptr.nextptr = curr.nextptr;
            curr.nextptr = node_ptr;
        }
    }

       void print(node head) {
            while (head != null)
        {
            System.out.print(head.data + " ");
            head = head.nextptr;
```

```
  }
}
```

III. (a) attached as a picture
III. (b)

Hashing:

If the sequential search is done in array or list in average case it take O(n). Hashing improve search in the sequential file, because the values are stored as key value. By calculating the key value using hash function the value can be directly accessed without searching entire array. Hashing enables to search the file in O(1). Hashing make use of data structure hash table. Hash table store data in key value format. The key is the index of the array which is computed using hash function Each value has its own unique index. One of the hash function used is modulo operator. If the array size is m and the value is k, the key or unique index is k%m.

Chaining:

If multiple values map to same index it result in clashes. The values map to same index 0. It results in clashes. In case of clashes, if the values are stored to next free cell, then in worst case searching takes O(n). In order to avoid clashes chaining is used. Chaining improve the searching of sequential file. •In chaining the values that map to same index are stored as linked list. So searching has to be done only in the slot or index mapped by the key. Each index corresponds to a linked list that stores the values that map to it.

Move to front:

In move to front methodology the file accessed is moved to the head of list. This methodology improves searching if the same file is accessed repeatedly as it will be at the head of the list.

Transposition methodology:

In transposition methodology the file accessed is swapped with its predecessor. This methodology improves searching as the most frequently accessed files will be at list front.

IV. (a)

Sort	**Worst Case**	**Best Case**	**Inplace?**
Bubble	O(n^2)	O(n)	Yes
Insertion	O(n^2)	O(n)	Yes
Selection	O(n^2)	O(n^2)	Yes
Merge Sort	O(nlog(n)).	O(nlog(n)).	No
Radix Sort	O(n+k).	O(nk).	No
Heap Sort	O(nlog(n))	O(nlog(n)).	Yes

IV. (b)

SHELL Sort :

- It is based on insertion sort algorithm.
- In this spacing between element is known as gap/interval.
- It compare the element that are distant apart rather than adjacent.
- In every pass, gap is reduced by 1 till we reach last pass when gap is 1 and then it works as an insertion sort.
- Formula gap = floor (n/2) where n= number of element in array.

Indexed Sequential sort :

- In this sort first index file is created.
- Index file contain the specific group or division of required records when the index is obtained.
- Partial indexing takes less time to search the element

Address calculation sort

- Uses the address table to store the value
- First we define the address or range for the calculation.
- In record we have the key, data and link.
- Now we have to define range like 0-20, 21-40 and 41-60.
- Insert the element to particular range.
- Sort elements.

V.

```
public class ReturnByLevel {

  LinkedList<Integer>levelOrderList[];

  NodeByLevel(int size){
     levelOrderList = new LinkedList[size];
     for (int i = 0; i < size; i++) {
         levelOrderList[i] = new LinkedList<>();
     }
 }

 public void PrintByValue(Node root , int val){

     int valueLevel = 0;
     LinkedList<Node> queue = new LinkedList<>();
     Node marker = new Node(00);
     int lev = 1;
     queue.add(root);
     queue.add(marker);

     while (queue.size()!=0){
         Node node = queue.poll();

         if(node == marker){

             if(queue.isEmpty() == true) break;

             queue.add(marker);
             lev=lev+1;

         }else {
```

```
                levelOrderList[lev].add(node.Data);
                if(node.Data == val)
                    valueLevel = lev;
            }

            if(node.left!=null){
                queue.add(node.left);
            }
            if(node.right!=null){
                queue.add(node.right);
            }
        }
        System.out.println(levelOrderList[valueLevel]);

    }
```

VI.

Project #1
Let assume I had to sort students grades from highest to lowest grade really fast before the class starts. In this case I'd use QuickSort. Quick Sort is a very fast, it has small constant and its worst case is O(n^2), it can also be reduced to nlog(n) and it performs better with less memory, such as grades.

Project#2
Let assume I had to sort a huge amount of files which were send to me overnight, they're all unordered and in different places. In this case i'd use Merge Sort. Merge Sort works better with large data structures. It is stable, unlike quicksort and heapsort, and can be easily adapted to operate on linked lists and very large lists.

Project#3
Let assume I was given a small list of some data. My job was to make sure it was all sorted correctly by some criterions. However, I've noticed not all data was on it's correct place. In this case I'd use Insertion sort to quickly sort small data. Insertion sort is often used when array in almost sorted, and when only few elements are misplaced. For almost sorted files it takes nearly O(n) time.