DS Assignment V

1A) For the following set of numbers perform radix sort. Show the sublists at each iteration and the numbers in the array.

5764, 493, 7159, 7100, 1959, 8813, 4302, 3855, 6767, 1313

Iteration 1 (LSD)

7100, 4302, 493, 8813, 1313, 5764, 3855, 6767, 7159, 1959

Iteration 2 (Second digit)

7100, 4302, 8813, 1313, 3855, 7159, 1959, 5764, 6767, 493

Iteration 3 (Third digit)

7100, 7159, 4302, 1313, 493, 5764, 6767, 8813, 3855, 1959

Iteration 4 (MSD)

493, 1313, 1959, 3855, 4302, 5764, 6767, 7100, 7159, 8813

1B) Write a program to sort character strings using merge sort.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE 50
#define initString(size) (char *)malloc(size * sizeof(char));

typedef char * String;

void merge (String *arr, int start, int mid, int end) {
     String *temp = (String *)calloc(end + 1, sizeof(String));

     int i;
     for (i = start; i <= end; ++i)
       *(temp + i) = initString(SIZE);

     i = start;
     int s = start, m = mid + 1;
```

```
    while ((s <= mid) && (m <= end)) {
       if (strcmp(*(arr + s), *(arr + m)) < 0)
          strcpy(*(temp + (i++)), *(arr + (s++)));
       else
          strcpy(*(temp + (i++)), *(arr + (m++)));
    }
    while (s <= mid)
       strcpy(*(temp + (i++)), *(arr + (s++)));

    while (m <= end)
       strcpy(*(temp + (i++)), *(arr + (m++)));


    for (i = start; i <= end; ++i)
       strcpy(*(arr + i), *(temp + i));

}

void mergeSort (String *arr, int start, int end) {
    int mid;
    if (start < end) {
        mid = (start + end)/2;
        mergeSort(arr, start, mid);
        mergeSort(arr, mid + 1, end);
        merge(arr, start, mid, end);
    }
}

void printArray (String *arr, int n) {
    printf("\n\t");
    int i;
    for (i = 0; i < n; ++i)
        printf("%s  ", *(arr + i));
    printf("\n");
}

int main (int argc, const char * argv[]) {

    String *arr = (String *)calloc(5, sizeof(String));

    int i;
    for (i = 0; i < 5; ++i)
        *(arr + i) = initString(SIZE);

    strcpy(*arr, "Merge");
    strcpy(*(arr + 1), "Radix");
    strcpy(*(arr + 2), "Quick");
    strcpy(*(arr + 3), "Heap");
    strcpy(*(arr + 4), "Insertion");

    printArray(arr, 5);    // Merge  Radix  Quick  Heap  Insertion

    mergeSort(arr, 0, 4);

    printArray(arr, 5);    // Heap  Insertion  Merge  Quick  Radix
}
```

2A) Write a function to delete a node from BST.

```
typedef struct TNode {
     int data;
     struct TNode *left;
     struct TNode *right;
} TNODE_t;

typedef TNODE_t * TNODE_p_t;

TNODE_p_t minValueNode (TNODE_p_t root) {
     TNODE_p_t temp = root;
     while (temp->left != NULL)
         temp = temp->left;
     return temp;
}

TNODE_p_t delete (TNODE_p_t root, int item) {

     if (root == NULL)
         return root;

     if (item < root->data)
         root->left = delete(root->left, item);

     else if (item > root->data)
         root->right = delete(root->right, item);

     else {
        if (root->left == NULL) {
            TNODE_p_t temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL) {
            TNODE_p_t temp = root->left;
            free(root);
            return temp;
        }

        TNODE_p_t temp = minValueNode(root->right);
        root->data = temp->data;
        root->right = delete(root->right, temp->data);
     }
     return root;
}
```

2B) Write a function to insert a node N as the left child of node T.

```
typedef struct TNode {
     int data;
     struct Node *left;
     struct Node *right;
} TNODE_t;

typedef TNODE_t * TNODE_p_t;

void insertLeft (TNODE_p_t T, TNODE_p_t N) {
     if (T->left == NULL)
           T->left = N;
     else {
           N->left = T->left;
           T->left = N;
     }
}
```

3B) What is an extended binary tree? Show with an example how an extended binary tree can be transformed into a red-black tree. Derive an expression for the height of red-black tree in terms of the number of internal nodes n.

An ***extended binary tree*** is a transformation of any binary tree into a complete binary tree. This transformation consists of replacing every null subtree of the original tree with "special nodes." The nodes from the original tree are then *internal nodes*, while the "special nodes" are *external nodes*.

A ***red-black tree*** is an extended binary tree in which every node has a color, either ***red*** or ***black***. The conditions that must satisfy for transforming a Extended Binary Tree to Red Black Tree:

1. A child and it's parent cannot both be red.
2. Every path from the root to an external node (descendent) contains the same number of black nodes
3. The root is black.

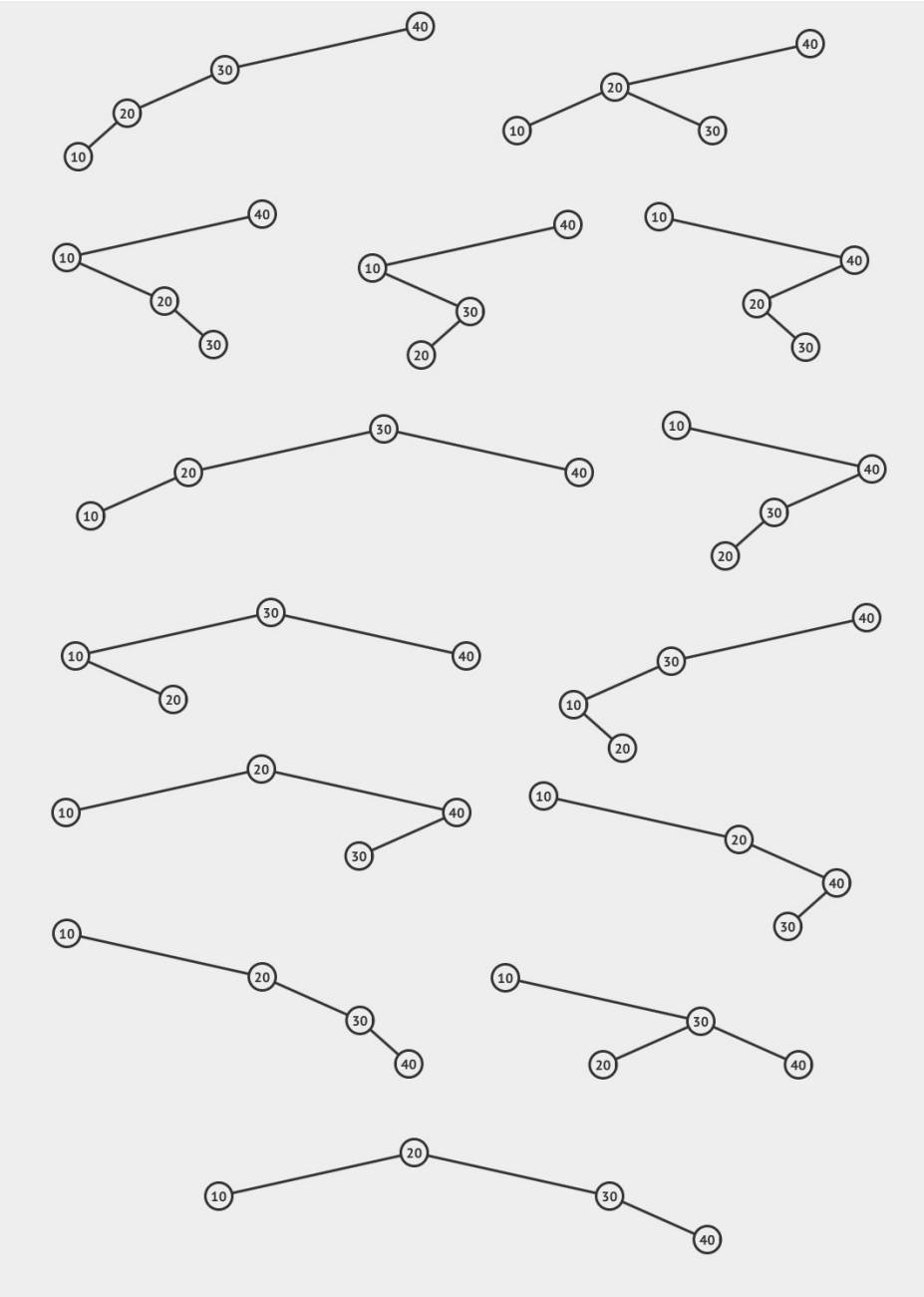
The number of internal nodes in a RBT of black-height ***k*** is **>= 2^k - 1** and **<= 2^{2k} - 1**

3A) For the key set `(10, 20, 30, 40)` how many possible binary search tree exist? Draw all of them and show the height in each case. How do you find the optimal binary search tree among them?

Number of BSTs possible for **n** elements =

$$\frac{(2n)!}{n!\,(n+1)!}$$

So, for 4 elements the number of trees possible = 8!/4!*5! = 14



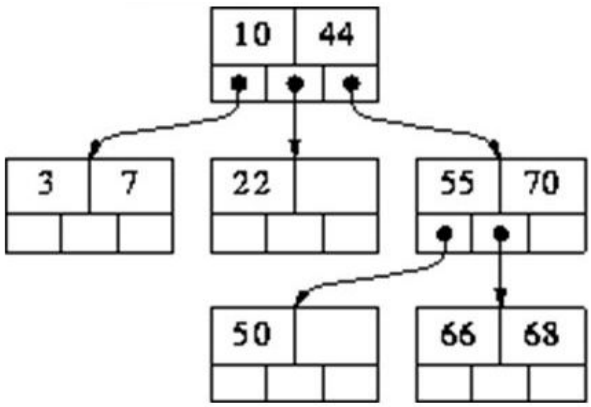
An optimal binary search tree is a binary search tree for which the nodes are arranged on levels such that the tree cost is minimum.

All trees above with root 20 or 30 are optimal.

4A) Taking properties of m-way search tree show that binary search trees are 2-way search trees. Draw a sample 3-way search tree and explain the terms such as degree, height and max. no of nodes.

An internal node of a ***m–way*** search tree consists of ***n*** subtrees and ***n-1*** keys where ***2 ≤ n ≤ m.***

For ***m = 2*** there will be two subtrees and one key for internal nodes. Leaf nodes will have 0 subtrees and one key. From one-to-one correspondence, Binary Search Trees are *2-way* search trees.



3–way search tree

Degree = 3,

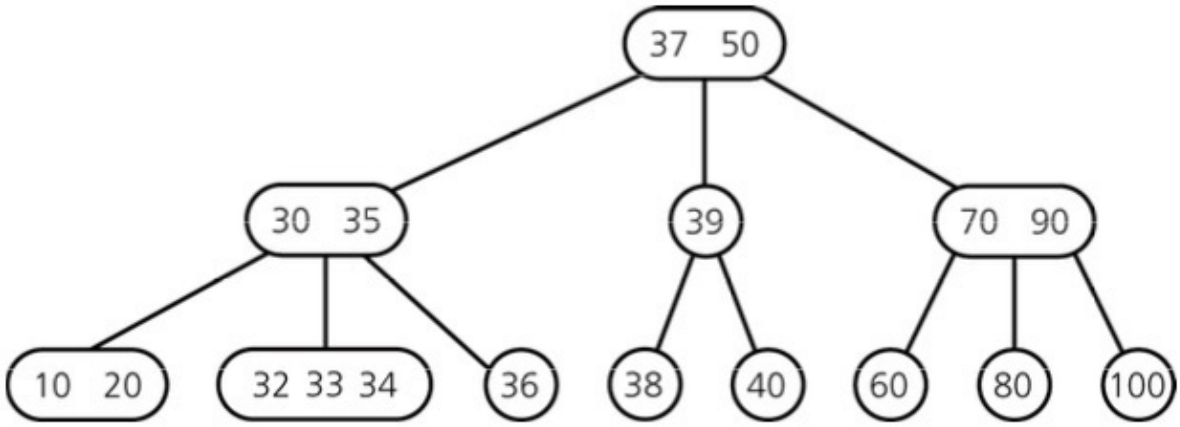
Max number of nodes = $\lfloor (3^{h+1} - 1) / 2 \rfloor$ 'h' being the height (taking root as a level)

Height h = $\lceil \log_3(2) + \log_3(n) - 1 \rceil$ where 'n' is the number of the nodes.

*Note -> In m-way trees, leaf nodes have 0 subtrees, and keys ranging from 1 to m-1. For binary trees, m = 2, hence one key.

— Courtesy YKL7

4B) Draw a B- tree of order 2, 3 and 4 and comment on the maximum number of elements in such a B-tree?



A ***2–3–4 B–Tree*** *is* a self-balancing data structure, i.e. height balanced.

It is a special kind of B-Tree, of Order = 4.

- It consists of nodes with one, two, or three elements, and two, three, or four links.
- A 2-node has one data field, and two links.
- A 3-node has two data fields, and three links.
- A 4-node has three data fields, and four links.

Minimum number of elements = 2 * d^h – 1, where h = height of the tree (taking root as level zero).

Maximum number of elements =
3 (height = 1)
3 + (1 + 2 + 3 + 3) = 12 (height = 2)
3 + (3 * 4) + (1 + 2 + 3 + 3 + 12*3) = 60 (height = 3)
. . .

A 2–3–4 B–Tree must contain at least one node with one element (2–Node), and one node with two elements (3–Node). Rest all nodes have to be 4–Nodes with three data fields, for maximum elements.

** *Calculate at your own risk.*

—- Courtesy YKL7

5) Write a C function to input the number of vertices and edges in an undirected graph. Next, input the edges one by one and to set up the linked adjacency-list representation of the graph. You may assume that no edge is input twice.

```
typedef struct AdjNode {
     int vertex;
     struct AdjNode *next;
} ADJ_NODE_t, *ADJ_NODE_p_t;

typedef struct AdjListNode {
     int count;
     ADJ_NODE_p_t head;
} ADJ_LIST_NODE_t, *ADJ_LIST_NODE_p_t;

ADJ_NODE_p_t createAdjNode (int value) {
     ADJ_NODE_p_t adjNode = (ADJ_NODE_p_t)malloc(sizeof(ADJ_NODE_t));
     adjNode->vertex = value;
     adjNode->next = NULL;
     return adjNode;
}

ADJ_LIST_NODE_p_t createAdjListNode () {
     ADJ_LIST_NODE_p_t adjListNode =
(ADJ_LIST_NODE_p_t)malloc(sizeof(ADJ_LIST_NODE_t));
     adjListNode->count = 0;
     adjListNode->head = NULL;
     return adjListNode;
}

void insertAdjNode (ADJ_NODE_p_t *head, int value) {
     if (*head == NULL) {
       *head = createAdjNode(value);
       return;
     }
     ADJ_NODE_p_t temp = *head;
     while (temp->next != NULL)
       temp = temp->next;
     temp->next = createAdjNode(value);
}
```

```
ADJ_LIST_NODE_p_t * inputAdjList () {

    int n, i, vertex;
    printf("\n\tEnter the number of vertices: ");
    scanf(" %d", &n);

    ADJ_LIST_NODE_p_t *listHeadArr = (ADJ_LIST_NODE_p_t *)calloc(n,
sizeof(ADJ_LIST_NODE_p_t));

    ADJ_LIST_NODE_p_t temp;

    for (i = 0; i < n; ++i) {

        *(listHeadArr + i) = createAdjListNode();
        temp = *(listHeadArr + i);

        printf("\n\tVertex %d, Enter the connected vertices (1 - %d), 0 to
break: ", i+1, n);

        do {
            scanf(" %d", &vertex);
            if (vertex != 0)
                  insertAdjNode(&temp->head, vertex);
        } while (vertex != 0);

    }

    return listHeadArr;
}
```