Tree Task 1

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

// Define the structure for a binary tree node
typedef struct TreeNode {
   int data;
   struct TreeNode* left;
   struct TreeNode* right;
} TreeNode;

// Function to create a new node with the given data
TreeNode* createNode(int data) {
   TreeNode* newNode = (TreeNode*)malloc(sizeof(TreeNode));
   newNode->data = data;
   newNode->left = NULL;
   newNode->right = NULL;
   return newNode;
}

// Function to insert a node into the binary tree
TreeNode* insert(TreeNode* root, int data) {
   if (root == NULL) {
      return createNode(data);
   }
   if (data < root->data) {
      root->left = insert(root->left, data);
   } else if (data > root->data) {
      root->right = insert(root->right, data);
   }
   return root;
}

// Pre-order traversal: Root, Left, Right
void preOrderTraversal(TreeNode* root) {
   if (root == NULL) return;
   printf("%d ", root->data);
   preOrderTraversal(root->left);
   preOrderTraversal(root->right);
}

// In-order traversal: Left, Root, Right
void inOrderTraversal(TreeNode* root) {
   if (root == NULL) return;
   inOrderTraversal(root->left);
   printf("%d ", root->data);
   inOrderTraversal(root->right);
}

// Post-order traversal: Left, Right, Root
void postOrderTraversal(TreeNode* root) {
   if (root == NULL) return;
   postOrderTraversal(root->left);
   postOrderTraversal(root->right);
   printf("%d ", root->data);
}
```

```
// Function to find the height of the binary tree
int findHeight(TreeNode* root) {
   if (root == NULL) return 0;
   int leftHeight = findHeight(root->left);
   int rightHeight = findHeight(root->right);
   return (leftHeight > rightHeight) ? leftHeight + 1 : rightHeight + 1;
}

// Function to count the number of leaf nodes in the binary tree
int countLeafNodes(TreeNode* root) {
   if (root == NULL) return 0;
   if (root->left == NULL && root->right == NULL) return 1;
   int leftLeaves = countLeafNodes(root->left);
   int rightLeaves = countLeafNodes(root->right);
   return leftLeaves + rightLeaves;
}

int main() {
   TreeNode* root = NULL;
   int values[] = {4, 2, 6, 1, 3, 5, 7};
   for (int i = 0; i < sizeof(values) / sizeof(values[0]); i++) {
      root = insert(root, values[i]);
   }

   printf("Pre-order traversal: ");
   preOrderTraversal(root);
   printf("\n");

   printf("In-order traversal: ");
   inOrderTraversal(root);
   printf("\n");

   printf("Post-order traversal: ");
   postOrderTraversal(root);
   printf("\n");

   int height = findHeight(root);
   printf("Height of the binary tree: %d\n", height);

   int leafCount = countLeafNodes(root);
   printf("Number of leaf nodes: %d\n", leafCount);

   return 0;
}
```

Searching and sorting

Binary search

```
#include <stdio.h>
int binarySearchIterative(int arr[], int size, int key) {
   int left = 0;
   int right = size - 1;

   while (left <= right) {
      int mid = left + (right - left) / 2;

      if (arr[mid] == key) {
         return mid;  // Key found at index mid
      }
      if (arr[mid] < key) {
         left = mid + 1;
      } else {
         right = mid - 1;
      }
   }

   return -1;  // Key not found in the array
}
int binarySearchRecursive(int arr[], int left, int right, int key) {
   if (left <= right) {
      int mid = left + (right - left) / 2;

      if (arr[mid] == key) {
         return mid;  // Key found at index mid
      }
      if (arr[mid] < key) {
         return binarySearchRecursive(arr, mid + 1, right, key);
      } else {
         return binarySearchRecursive(arr, left, mid - 1, key);
      }
   }

   return -1;  // Key not found in the array
}

int main() {
   int arr[] = {12, 34, 45, 67, 78, 89};
   int size = sizeof(arr) / sizeof(arr[0]);
   int key = 45;

   int result = binarySearchIterative(arr, size - 1, key);
   if (result != -1) {
      printf("Element found at index: %d\n", result);
   } else {
      printf("Element not found in the array.\n");
   }

   return 0;
}
```

Linear

```
#include <stdio.h>
int linearSearchIterative(int arr[], int size, int key) {
   for (int i = 0; i < size; i++) {
      if (arr[i] == key) {
         return i;  // Key found at index i
      }
   }
   return -1;  // Key not found in the array
}
int linearSearchRecursive(int arr[], int start, int end, int key) {
   if (start > end) {
      return -1;  // Key not found in the array
   }

   if (arr[start] == key) {
      return start;  // Key found at index start
   }

   return linearSearchRecursive(arr, start + 1, end, key);
}

int main() {
   int arr[] = {12, 45, 67, 89, 34, 78};
   int size = sizeof(arr) / sizeof(arr[0]);
   int key = 34;

   int result = linearSearchRecursive(arr, 0, size - 1, key);
   if (result != -1) {
      printf("Element found at index: %d\n", result);
   } else {
      printf("Element not found in the array.\n");
   }

   return 0;
}
```

Bubble Sort

```
#include<stdio.h>
#include<stdlib.h>
void bubbleSort(int arr[], int n) {
   for (int i = 0; i < n-1; i++) {
      for (int j = 0; j < n-i-1; j++) {
         if (arr[j] > arr[j+1]) {
            int temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
         }
      }
   }
}
```

Optimized BubbleSort

```
void optimizedBubbleSort(int arr[], int n) {
   int swapped;
   for (int i = 0; i < n-1; i++) {
      swapped = 0;
      for (int j = 0; j < n-i-1; j++) {
         if (arr[j] > arr[j+1]) {
            int temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
            swapped = 1;
         }
      }
      if (swapped == 0)
         break;
   }
}
```

SelectionSort

```
void selectionSort(int arr[], int n) {
   for (int i = 0; i < n-1; i++) {
      int minIndex = i;
      for (int j = i+1; j < n; j++) {
         if (arr[j] < arr[minIndex]) {
            minIndex = j;
         }
      }
      int temp = arr[i];
      arr[i] = arr[minIndex];
      arr[minIndex] = temp;
   }
}
```

Insertion Sort

```
void insertionSort(int arr[], int n) {
   for (int i = 1; i < n; i++) {
      int key = arr[i];
      int j = i - 1;
      while (j >= 0 && arr[j] > key) {
         arr[j + 1] = arr[j];
         j--;
      }
      arr[j + 1] = key;
   }
}
```

Merge Sort

```
void merge(int arr[], int left, int mid, int right) {
   int i, j, k;
   int n1 = mid - left + 1;
   int n2 = right - mid;

   int L[n1], R[n2];

   for (i = 0; i < n1; i++)
      L[i] = arr[left + i];
   for (j = 0; j < n2; j++)
      R[j] = arr[mid + 1 + j];

   i = 0;
```

```
    j = 0;
    k = left;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }

    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;

    return (i + 1);
}
```

Quick Sort

```
void quickSort(int arr[], int low, int high) {
    if (low < high) {
```

```
    int pi = partition(arr, low, high);
    quickSort(arr, low, pi - 1);
    quickSort(arr, pi + 1, high);
  }
}
void heapify(int arr[], int n, int i) {
  int largest = i;
  int left = 2 * i + 1;
  int right = 2 * i + 2;

  if (left < n && arr[left] > arr[largest])
    largest = left;

  if (right < n && arr[right] > arr[largest])
    largest = right;

  if (largest != i) {
    int temp = arr[i];
    arr[i] = arr[largest];
    arr[largest] = temp;

    heapify(arr, n, largest);
  }
}
```

Heap sort

```
void heapSort(int arr[], int n) {
  for (int i = n / 2 - 1; i >= 0; i--)
    heapify(arr, n, i);

  for (int i = n - 1; i > 0; i--) {
    int temp = arr[0];
    arr[0] = arr[i];
    arr[i] = temp;

    heapify(arr, i, 0);
  }
}

int main ()
{
        FILE* fp;
        fp = fopen("random_with_repitions100.txt", "r");

        int *arr;
  int n = sizeof(arr) / sizeof(arr[0]);
        int num,size=0,i=0;
        fscanf(fp, "%d", &size);
        arr = (int*)malloc(sizeof(int)*size);
        while (!feof(fp)&&i<size)
        {
                //fscanf(fp, "%d\n", &num);
                 fscanf(fp, "%d", &arr[i++]);
        }
        fclose(fp);
        printf("%d\n",size);
        //bubbleSort(arr,size);
        //optimizedBubbleSort(arr,size);
```

```
            //selectionSort(arr,size);
            //mergeSort(arr,0,size-1);
            //quickSort(arr,0,size-1);
            //heapSort(arr,size);
            for (i = 0; i <size; i++)
    printf("%d ", arr[i]);
}
```