Lab - Experiment 10

Objective:

• To understand and implement AVL Trees with self-balancing properties.

• To apply heap sort for efficient sorting.

• To implement a priority queue using a heap structure.

Assignment 1st: AVL Tree Implementation

Tasks:

• Implement insertion in an AVL tree. Ensure that after each insertion, the tree remains balanced using rotations (left rotation, right rotation, left-right rotation, right-left rotation).
• Implement deletion in an AVL tree with the necessary rebalancing steps.

Testing: Insert and delete a series of values, displaying the tree structure after each operation.

Code:

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

typedef struct Node {
  int key;
  struct Node *left, *right;
  int height;
} Node;

// Utility function to get the height of the tree
int height(Node *N) { return (N == NULL) ? 0 : N->height; }

// Function to get the balance factor of a node
int getBalance(Node *N) {
  return (N == NULL) ? 0 : height(N->left) - height(N->right);
}

// Helper function to create a new node
Node *newNode(int key) {
  Node *node = (Node *)malloc(sizeof(Node));
  node->key = key;
  node->left = node->right = NULL;
  node->height = 1;
  return node;
}

// Right rotate subtree rooted with y
Node *rightRotate(Node *y) {
  Node *x = y->left;
  Node *T2 = x->right;
  x->right = y;
  y->left = T2;
  y->height = 1 + fmax(height(y->left), height(y->right));
```

```
  x->height = 1 + fmax(height(x->left), height(x->right));
  return x;
}
// Left rotate subtree rooted with x
Node *leftRotate(Node *x) {
  Node *y = x->right;
  Node *T2 = y->left;
  y->left = x;
  x->right = T2;
  x->height = 1 + fmax(height(x->left), height(x->right));
  y->height = 1 + fmax(height(y->left), height(y->right));
  return y;
}
// Recursive function to insert a key in the AVL Tree
Node *insert(Node *node, int key) {
  if (node == NULL)
    return newNode(key);
  if (key < node->key)
    node->left = insert(node->left, key);
  else if (key > node->key)
    node->right = insert(node->right, key);
  else
    return node;
  node->height = 1 + fmax(height(node->left), height(node->right));
  int balance = getBalance(node);
  if (balance > 1 && key < node->left->key)
    return rightRotate(node);
  if (balance < -1 && key > node->right->key)
    return leftRotate(node);
  if (balance > 1 && key > node->left->key) {
    node->left = leftRotate(node->left);
    return rightRotate(node);
  }
  if (balance < -1 && key < node->right->key) {
    node->right = rightRotate(node->right);
    return leftRotate(node);
  }
  return node;
}
// Recursive function to delete a node from the AVL Tree
Node *deleteNode(Node *root, int key) {
  if (root == NULL)
    return root;
  if (key < root->key)
    root->left = deleteNode(root->left, key);
  else if (key > root->key)
    root->right = deleteNode(root->right, key);
  else {
    if ((root->left == NULL) || (root->right == NULL)) {
      Node *temp = root->left ? root->left : root->right;
      if (temp == NULL) {
        temp = root;
        root = NULL;
      } else
        *root = *temp;
      free(temp);
    } else {
      Node *temp = root->right;
      while (temp->left != NULL)
        temp = temp->left;
```

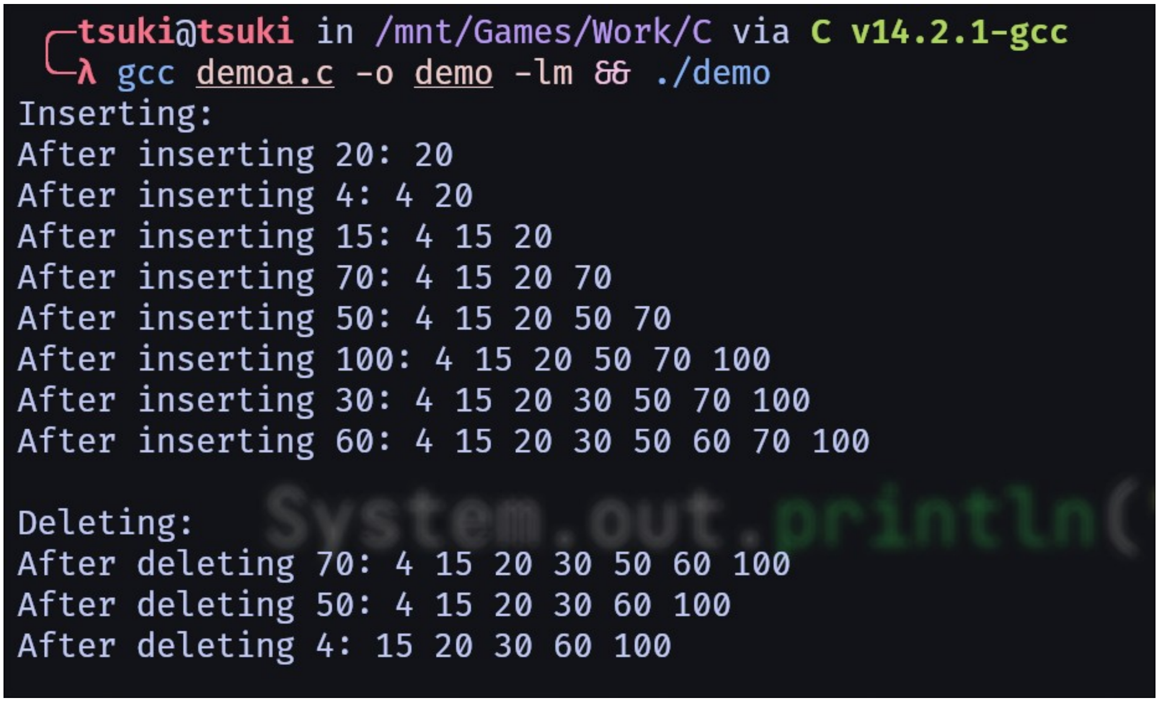
```
      root->key = temp->key;
      root->right = deleteNode(root->right, temp->key);
    }
  }
  if (root == NULL)
    return root;

  root->height = 1 + fmax(height(root->left), height(root->right));
  int balance = getBalance(root);

  if (balance > 1 && getBalance(root->left) >= 0)
    return rightRotate(root);
  if (balance > 1 && getBalance(root->left) < 0) {
    root->left = leftRotate(root->left);
    return rightRotate(root);
  }
  if (balance < -1 && getBalance(root->right) <= 0)
    return leftRotate(root);
  if (balance < -1 && getBalance(root->right) > 0) {
    root->right = rightRotate(root->right);
    return leftRotate(root);
  }
  return root;
}
// Helper function to print the tree in-order
void printInOrder(Node *root) {
  if (root != NULL) {
    printInOrder(root->left);
    printf("%d ", root->key);
    printInOrder(root->right);
  }
}
// Main function to test the AVL Tree insertion and deletion
int main() {
  Node *root = NULL;

  int keys[] = {20, 4, 15, 70, 50, 100, 30, 60};
  int i;
  printf("Inserting:\n");
  for (i = 0; i < 8; i++) {
    root = insert(root, keys[i]);
    printf("After inserting %d: ", keys[i]);
    printInOrder(root);
    printf("\n");
  }
  printf("\nDeleting:\n");
  int deleteKeys[] = {70, 50, 4};
  for (i = 0; i < 3; i++) {
    root = deleteNode(root, deleteKeys[i]);
    printf("After deleting %d: ", deleteKeys[i]);
    printInOrder(root);
    printf("\n");
  }
  return 0;
}
```

Output:-



Assignment 2nd: Heap Sort Implementation

Tasks:

• Build a max heap from an array of unsorted elements.

• Implement heap sort by repeatedly removing the root element (maximum value) and re-heapifying the tree.

Testing: Demonstrate heap sort with an example array, showing each step and the final sorted output.

Code:

```
#include <stdio.h>
// Function to swap two elements
void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}
// Function to maintain the max-heap property for a subtree rooted at index i
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) {
    swap(&arr[i], &arr[largest]);
    heapify(arr, n, largest);
  }
}

// Function to build a max heap from an array
```

```
void buildMaxHeap(int arr[], int n) {
  for (int i = n / 2 - 1; i >= 0; i--)
    heapify(arr, n, i);
}
// Function to print the array
void printArray(int arr[], int n) {
  for (int i = 0; i < n; i++)
    printf("%d ", arr[i]);
  printf("\n");
}
// Function to perform heap sort
void heapSort(int arr[], int n) {
  buildMaxHeap(arr, n);
  printf("Max heap built: ");
  printArray(arr, n);
  for (int i = n - 1; i >= 0; i--) {
    swap(&arr[0], &arr[i]);
    printf("After moving max to end (index %d): ", i);
    printArray(arr, n);
    heapify(arr, i, 0);
    printf("After re-heapifying: ");
    printArray(arr, n);
  }
}
// Main function to demonstrate heap sort
int main() {
  int arr[] = {12, 11, 13, 5, 6, 7};
  int n = sizeof(arr) / sizeof(arr[0]);
  printf("Original array: ");
  printArray(arr, n);
  heapSort(arr, n);
  printf("Sorted array: ");
  printArray(arr, n);
  return 0;
}
```

Output:-

```
"demoa.c" 70L, 1482B written
:!gcc demoa.c -o demoa && ./demoa
Original array: 12 11 13 5 6 7
Max heap built: 13 11 12 5 6 7
After moving max to end (index 5): 7 11 12 5 6 13
After re-heapifying: 12 11 7 5 6 13
After moving max to end (index 4): 6 11 7 5 12 13
After re-heapifying: 11 6 7 5 12 13
After moving max to end (index 3): 5 6 7 11 12 13
After re-heapifying: 7 6 5 11 12 13
After moving max to end (index 2): 5 6 7 11 12 13
After re-heapifying: 6 5 7 11 12 13
After moving max to end (index 1): 5 6 7 11 12 13
After re-heapifying: 5 6 7 11 12 13
After moving max to end (index 0): 5 6 7 11 12 13
After re-heapifying: 5 6 7 11 12 13
Sorted array: 5 6 7 11 12 13
```

Assignment 3rd: Priority Queue Using Heap

Tasks:

• Implement a priority queue using a heap structure.

• Implement functions to insert elements with a priority and to remove the highest priority element.

Testing: Insert elements with varying priorities and demonstrate removing elements in priority order.

Code:-

```
#include <stdio.h>
#include <stdlib.h>
// Define a structure for a priority queue element
typedef struct {
    int data;
    int priority;
} Element;
// Priority Queue structure with a dynamic array for the heap and size tracking
typedef struct {
    Element *elements;
    int size;
    int capacity;
} PriorityQueue;
// Helper function to swap two elements in the heap
void swap(Element *a, Element *b) {
    Element temp = *a;
    *a = *b;
    *b = temp;
}
// Function to create a priority queue with a specified capacity
PriorityQueue* createPriorityQueue(int capacity) {
    PriorityQueue *pq = (PriorityQueue *)malloc(sizeof(PriorityQueue));
    pq->elements = (Element *)malloc(capacity * sizeof(Element));
    pq->size = 0;
    pq->capacity = capacity;
    return pq;
}
// Function to maintain max-heap property for the priority queue
void heapifyDown(PriorityQueue *pq, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    if (left < pq->size && pq->elements[left].priority > pq->elements[largest].priority)
        largest = left;
    if (right < pq->size && pq->elements[right].priority > pq->elements[largest].priority)
        largest = right;
    if (largest != i) {
        swap(&pq->elements[i], &pq->elements[largest]);
        heapifyDown(pq, largest);
    }
}
```

```c
// Function to maintain max-heap property for insertion
void heapifyUp(PriorityQueue *pq, int i) {
    int parent = (i - 1) / 2;
    if (i > 0 && pq->elements[i].priority > pq->elements[parent].priority) {
        swap(&pq->elements[i], &pq->elements[parent]);
        heapifyUp(pq, parent);
    }
}
// Function to insert a new element with a given priority
void insert(PriorityQueue *pq, int data, int priority) {
    if (pq->size == pq->capacity) {
        printf("Priority Queue is full. Cannot insert element.\n");
        return;
    }
    pq->elements[pq->size].data = data;
    pq->elements[pq->size].priority = priority;
    pq->size++;
    heapifyUp(pq, pq->size - 1);
}
// Function to remove and return the element with the highest priority
Element removeMax(PriorityQueue *pq) {
    if (pq->size == 0) {
        printf("Priority Queue is empty.\n");
        return (Element){-1, -1};  // Return a default value indicating an empty queue}
    Element max = pq->elements[0];
    pq->elements[0] = pq->elements[pq->size - 1];
    pq->size--;
    heapifyDown(pq, 0);
    return max;
}
// Function to print the priority queue
void printPriorityQueue(PriorityQueue *pq) {
    for (int i = 0; i < pq->size; i++) {
        printf("Data: %d, Priority: %d\n", pq->elements[i].data, pq->elements[i].priority);
    }
    printf("\n");}
// Main function to demonstrate the priority queue
int main() {
    PriorityQueue *pq = createPriorityQueue(10);
    // Insert elements with varying priorities
    insert(pq, 10, 2);
    insert(pq, 20, 5);
    insert(pq, 30, 1);
    insert(pq, 40, 3);
    insert(pq, 50, 4);
    printf("Priority Queue after insertions:\n");
    printPriorityQueue(pq);
    // Remove elements in priority order
    printf("Removing elements in priority order:\n");
    while (pq->size > 0) {
        Element max = removeMax(pq);
        printf("Removed element with data: %d, priority: %d\n", max.data, max.priority);
        printPriorityQueue(pq);}
    // Clean up
    free(pq->elements);
    free(pq);
    return 0;
}
```

Output:-

```
:!gcc demoa.c -o demoa && ./demoa
Priority Queue after insertions:
Data: 20, Priority: 5
Data: 50, Priority: 4
Data: 30, Priority: 1
Data: 10, Priority: 2
Data: 40, Priority: 3

Removing elements in priority order:
Removed element with data: 20, priority: 5
Data: 50, Priority: 4
Data: 40, Priority: 3
Data: 30, Priority: 1
Data: 10, Priority: 2

Removed element with data: 50, priority: 4
Data: 40, Priority: 3
Data: 10, Priority: 2
Data: 30, Priority: 1

Removed element with data: 40, priority: 3
Data: 10, Priority: 2
Data: 30, Priority: 1

Removed element with data: 10, priority: 2
Data: 30, Priority: 1

Removed element with data: 30, priority: 1
```