ADS LAB PROGRAMS

Experiment-1:

1. Write a program to perform the following operations:

a) Insert an element into a binary search tree.
b) Delete an element from a binary search tree.
c) Search for a key element in a binary search tree.

Program:

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

/* Structure of a BST node */
struct node
{
   int data;
   struct node *left;
   struct node *right;
};

/* Create a new node */
struct node* createNode(int value)
{
   struct node *newNode = (struct node*)malloc(sizeof(struct node));
   newNode->data = value;
   newNode->left = NULL;
   newNode->right = NULL;
   return newNode;
}

/* Insert an element into BST */
struct node* insert(struct node *root, int value)
{
   if (root == NULL)
      return createNode(value);

   if (value < root->data)
      root->left = insert(root->left, value);
   else if (value > root->data)
      root->right = insert(root->right, value);

   return root;
}

/* Search an element in BST */
void search(struct node *root, int key)
{
   if (root == NULL)
```

```c
    {
        printf("Element not found\n");
        return;
    }

    if (root->data == key)
    {
        printf("Element found\n");
        return;
    }
    else if (key < root->data)
        search(root->left, key);
    else
        search(root->right, key);
}

/* Find minimum value node */
struct node* findMin(struct node *root)
{
    while (root->left != NULL)
        root = root->left;
    return root;
}

/* Delete an element from BST */
struct node* deleteNode(struct node *root, int key)
{
    if (root == NULL)
        return root;

    if (key < root->data)
        root->left = deleteNode(root->left, key);
    else if (key > root->data)
        root->right = deleteNode(root->right, key);
    else
    {
        if (root->left == NULL)
        {
            struct node *temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL)
        {
            struct node *temp = root->left;
            free(root);
            return temp;
        }

        struct node *temp = findMin(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
```

```
    return root;
}

/* Inorder traversal */
void inorder(struct node *root)
{
    if (root != NULL)
    {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

/* Main function */
int main()
{
    struct node *root = NULL;
    int choice, value;

    while (1)
    {
        /* Display tree BEFORE menu */
        printf("\n\nCurrent BST (Inorder): ");
        inorder(root);
        printf("\n");

        printf("\n--- Binary Search Tree Operations ---");
        printf("\n1. Insert");
        printf("\n2. Delete");
        printf("\n3. Search");
        printf("\n4. Exit");
        printf("\nEnter your choice: ");
        scanf("%d", &choice);

        switch (choice)
        {
            case 1:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                root = insert(root, value);
                break;

            case 2:
                printf("Enter value to delete: ");
                scanf("%d", &value);
                root = deleteNode(root, value);
                break;

            case 3:
                printf("Enter value to search: ");
                scanf("%d", &value);
                search(root, value);
```

```
          break;

        case 4:
          exit(0);

        default:
          printf("Invalid choice\n");
      }
    }

  return 0;
}
```

Output:

```
"C:\Users\abhin\OneDrive\De


Current BST (Inorder): 22 25 31 35

--- Binary Search Tree Operations ---
1. Insert
2. Delete
3. Search
4. Exit
Enter your choice: 1
Enter value to insert: 26


Current BST (Inorder): 22 25 26 31 35

--- Binary Search Tree Operations ---
1. Insert
2. Delete
3. Search
4. Exit
Enter your choice: 2
Enter value to delete: 26


Current BST (Inorder): 22 25 31 35

--- Binary Search Tree Operations ---
1. Insert
2. Delete
3. Search
4. Exit
Enter your choice: 3
Enter value to search: 31
Element found
```

Experiment-2:

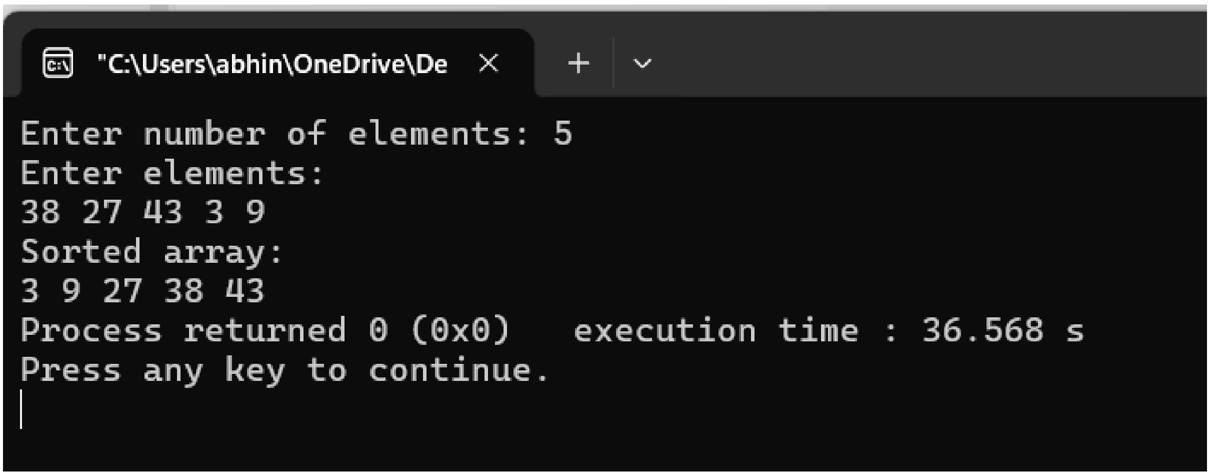
2. Write a program for implementing the following sorting methods:
a) Merge sort   b) Heap sort   c) Quick sort

a) Merge sort
Program:

```
#include <stdio.h>

// Function to merge two subarrays
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];

   // Copy data to temp arrays
   for (i = 0; i < n1; i++)
      L[i] = arr[left + i];
   for (j = 0; j < n2; j++)
      R[j] = arr[mid + 1 + j];

   // Merge the temp arrays
   i = 0; j = 0; k = left;
   while (i < n1 && j < n2) {
      if (L[i] <= R[j])
         arr[k++] = L[i++];
      else
         arr[k++] = R[j++];
   }

   // Copy remaining elements
   while (i < n1)
      arr[k++] = L[i++];
   while (j < n2)
      arr[k++] = R[j++];
}

// Merge sort function
void mergeSort(int arr[], int left, int right) {
   if (left < right) {
      int mid = (left + right) / 2;
      mergeSort(arr, left, mid);
      mergeSort(arr, mid + 1, right);
      merge(arr, left, mid, right);
   }
}

int main() {
   int n, i;
```

```
    int arr[50];

    printf("Enter number of elements: ");
    scanf("%d", &n);

    printf("Enter elements:\n");
    for (i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    mergeSort(arr, 0, n - 1);

    printf("Sorted array:\n");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);

    return 0;
}
```

Output:



----

b) Heap sort

Program:

```
#include <stdio.h>

// Function to swap two numbers
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Heapify function
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);
    }
}

// Heap sort function
void heapSort(int arr[], int n) {
    int i;

    // Build heap
    for (i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // Extract elements from heap
    for (i = n - 1; i > 0; i--) {
        swap(&arr[0], &arr[i]);
        heapify(arr, i, 0);
    }
}

int main() {
    int n, i;
    int arr[50];

    printf("Enter number of elements: ");
    scanf("%d", &n);

    printf("Enter elements:\n");
    for (i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    heapSort(arr, n);

    printf("Sorted array:\n");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);

    return 0;
}
```

Output:

```
"C:\Users\abhin\OneDrive\De

Enter number of elements: 5
Enter elements:
38 27 43 3 5
Sorted array:
3 5 27 38 43
Process returned 0 (0x0)   execution time : 11.590 s
Press any key to continue.
```

----

c) Quick sort

Program:

```
#include <stdio.h>

// Function to swap
void swap(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}

// Partition function
int partition(int arr[], int low, int high) {
   int pivot = arr[high];
   int i = low - 1;
   int j;

   for (j = low; j < high; j++) {
      if (arr[j] <= pivot) {
         i++;
         swap(&arr[i], &arr[j]);
      }
   }
   swap(&arr[i + 1], &arr[high]);
   return i + 1;
}

// Quick sort function
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);
   }
}
```

```c
int main() {
   int n, i;
   int arr[50];

   printf("Enter number of elements: ");
   scanf("%d", &n);

   printf("Enter elements:\n");
   for (i = 0; i < n; i++)
      scanf("%d", &arr[i]);

   quickSort(arr, 0, n - 1);

   printf("Sorted array:\n");
   for (i = 0; i < n; i++)
      printf("%d ", arr[i]);

   return 0;
}
```

Output:

```
"C:\Users\abhin\OneDrive\De
Enter number of elements: 5
Enter elements:
10 9 8 1 5
Sorted array:
1 5 8 9 10
Process returned 0 (0x0)   execution time : 16.873 s
Press any key to continue.
```

Experiment-3:

3. Write a program to perform the following operations:
a) Insert an element into a B- tree.
b) Delete an element from a B- tree.
c) Search for a key element in a B- tree.

Program:

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

#define T 2   // Minimum degree

/* B-Tree node structure */
struct BTreeNode
{
   int keys[2*T-1];
   struct BTreeNode *child[2*T];
   int n;
   int leaf;
};

/* Create a new B-Tree node */
struct BTreeNode* createNode(int leaf)
{
   struct BTreeNode *node = (struct BTreeNode*)malloc(sizeof(struct BTreeNode));
   node->leaf = leaf;
   node->n = 0;

   for (int i = 0; i < 2*T; i++)
      node->child[i] = NULL;

   return node;
}

/* Traverse and display B-Tree */
void traverse(struct BTreeNode *root)
{
   if (root != NULL)
   {
      int i;
      for (i = 0; i < root->n; i++)
      {
         if (!root->leaf)
            traverse(root->child[i]);
         printf("%d ", root->keys[i]);
      }
```

```
        if (!root->leaf)
            traverse(root->child[i]);
    }
}

/* Search a key */
struct BTreeNode* search(struct BTreeNode *root, int key)
{
    int i = 0;
    while (i < root->n && key > root->keys[i])
        i++;

    if (i < root->n && root->keys[i] == key)
        return root;

    if (root->leaf)
        return NULL;

    return search(root->child[i], key);
}

/* Split child */
void splitChild(struct BTreeNode *x, int i, struct BTreeNode *y)
{
    struct BTreeNode *z = createNode(y->leaf);
    z->n = T - 1;

    for (int j = 0; j < T - 1; j++)
        z->keys[j] = y->keys[j + T];

    if (!y->leaf)
        for (int j = 0; j < T; j++)
            z->child[j] = y->child[j + T];

    y->n = T - 1;

    for (int j = x->n; j >= i + 1; j--)
        x->child[j + 1] = x->child[j];

    x->child[i + 1] = z;

    for (int j = x->n - 1; j >= i; j--)
        x->keys[j + 1] = x->keys[j];

    x->keys[i] = y->keys[T - 1];
    x->n++;
}

/* Insert in non-full node */
```

```
void insertNonFull(struct BTreeNode *x, int k)
{
   int i = x->n - 1;

   if (x->leaf)
   {
      while (i >= 0 && k < x->keys[i])
      {
         x->keys[i + 1] = x->keys[i];
         i--;
      }
      x->keys[i + 1] = k;
      x->n++;
   }
   else
   {
      while (i >= 0 && k < x->keys[i])
         i--;

      if (x->child[i + 1]->n == 2*T - 1)
      {
         splitChild(x, i + 1, x->child[i + 1]);
         if (k > x->keys[i + 1])
            i++;
      }
      insertNonFull(x->child[i + 1], k);
   }
}

/* Insert a key */
struct BTreeNode* insert(struct BTreeNode *root, int k)
{
   if (root == NULL)
   {
      root = createNode(1);
      root->keys[0] = k;
      root->n = 1;
      return root;
   }

   if (root->n == 2*T - 1)
   {
      struct BTreeNode *s = createNode(0);
      s->child[0] = root;
      splitChild(s, 0, root);

      int i = 0;
      if (s->keys[0] < k)
         i++;
```

```c
        insertNonFull(s->child[i], k);

        return s;
    }
    else
    {
        insertNonFull(root, k);
        return root;
    }
}

/* Simple delete (leaf-only for clarity) */
void deleteKey(struct BTreeNode *root, int k)
{
    int i;
    for (i = 0; i < root->n && root->keys[i] < k; i++);

    if (i < root->n && root->keys[i] == k)
    {
        if (root->leaf)
        {
            for (int j = i; j < root->n - 1; j++)
                root->keys[j] = root->keys[j + 1];
            root->n--;
            printf("Key deleted\n");
        }
        else
            printf("Deletion supported only for leaf nodes (simplified)\n");
    }
    else
    {
        if (root->leaf)
            printf("Key not found\n");
        else
            deleteKey(root->child[i], k);
    }
}

/* Main function */
int main()
{
    struct BTreeNode *root = NULL;
    int choice, key;

    while (1)
    {
        /* Display tree BEFORE menu */
        printf("\nCurrent B-Tree: ");
        traverse(root);
```

```c
        printf("\n");

        printf("\n--- B-Tree Operations ---");
        printf("\n1. Insert");
        printf("\n2. Delete");
        printf("\n3. Search");
        printf("\n4. Exit");
        printf("\nEnter choice: ");
        scanf("%d", &choice);

        switch (choice)
        {
            case 1:
                printf("Enter key to insert: ");
                scanf("%d", &key);
                root = insert(root, key);
                break;

            case 2:
                printf("Enter key to delete: ");
                scanf("%d", &key);
                if (root != NULL)
                    deleteKey(root, key);
                break;

            case 3:
                printf("Enter key to search: ");
                scanf("%d", &key);
                if (root != NULL && search(root, key))
                    printf("Key found\n");
                else
                    printf("Key not found\n");
                break;

            case 4:
                exit(0);

            default:
                printf("Invalid choice\n");
        }
    }
}
```

Output:

```
"C:\Users\abhin\OneDrive\De

Current B-Tree: 45 52 54 55 56 56 88

--- B-Tree Operations ---
1. Insert
2. Delete
3. Search
4. Exit
Enter choice: 1
Enter key to insert: 99

Current B-Tree: 45 52 54 55 56 56 88 99

--- B-Tree Operations ---
1. Insert
2. Delete
3. Search
4. Exit
Enter choice: 2
Enter key to delete: 99
Key deleted

Current B-Tree: 45 52 54 55 56 56 88

--- B-Tree Operations ---
1. Insert
2. Delete
3. Search
4. Exit
Enter choice: 3
Enter key to search: 54
Key found
```

Experiment-4:

4. Write a program to perform the following operations:

a) Insert an element into a Min-Max heap

b) Delete an element from a Min-Max heap

c) Search for a key element in a Min-Max heap

Program:

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

#define MAX 100

int heap[MAX];
int size = 0;

/* Utility: swap two values */
void swap(int *a, int *b)
{
   int temp = *a;
   *a = *b;
   *b = temp;
}

/* Find level of a node */
int level(int index)
{
   return (int)log2(index + 1);
}

/* Check if node is on min level */
int isMinLevel(int index)
{
   return level(index) % 2 == 0;
}

/* Bubble up for min level */
void bubbleUpMin(int index)
{
   while (index > 2)
   {
      int parent = (index - 1) / 2;
      int grandparent = (parent - 1) / 2;

      if (heap[index] < heap[grandparent])
      {
         swap(&heap[index], &heap[grandparent]);
         index = grandparent;
      }
      else
```

```
        break;
    }
}

/* Bubble up for max level */
void bubbleUpMax(int index)
{
    while (index > 2)
    {
        int parent = (index - 1) / 2;
        int grandparent = (parent - 1) / 2;

        if (heap[index] > heap[grandparent])
        {
            swap(&heap[index], &heap[grandparent]);
            index = grandparent;
        }
        else
            break;
    }
}

/* Insert element */
void insert(int value)
{
    heap[size] = value;
    int index = size;
    size++;

    if (index == 0)
        return;

    int parent = (index - 1) / 2;

    if (isMinLevel(index))
    {
        if (heap[index] > heap[parent])
        {
            swap(&heap[index], &heap[parent]);
            bubbleUpMax(parent);
        }
        else
            bubbleUpMin(index);
    }
    else
    {
        if (heap[index] < heap[parent])
        {
            swap(&heap[index], &heap[parent]);
            bubbleUpMin(parent);
        }
        else
            bubbleUpMax(index);
```

```
    }
}

/* Delete root (minimum element) */
void deleteMin()
{
    if (size == 0)
    {
        printf("Heap is empty\n");
        return;
    }

    heap[0] = heap[size - 1];
    size--;
    printf("Minimum element deleted\n");
}

/* Search element */
void search(int key)
{
    for (int i = 0; i < size; i++)
    {
        if (heap[i] == key)
        {
            printf("Element found\n");
            return;
        }
    }
    printf("Element not found\n");
}

/* Display heap */
void display()
{
    for (int i = 0; i < size; i++)
        printf("%d ", heap[i]);
}

/* Main function */
int main()
{
    int choice, value;

    while (1)
    {
        /* Display heap BEFORE menu */
        printf("\nCurrent Min-Max Heap: ");
        display();
        printf("\n");

        printf("\n--- Min-Max Heap Operations ---");
        printf("\n1. Insert");
        printf("\n2. Delete (Min)");
```

```
        printf("\n3. Search");
        printf("\n4. Exit");
        printf("\nEnter your choice: ");
        scanf("%d", &choice);

        switch (choice)
        {
            case 1:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                insert(value);
                break;

            case 2:
                deleteMin();
                break;

            case 3:
                printf("Enter value to search: ");
                scanf("%d", &value);
                search(value);
                break;

            case 4:
                exit(0);

            default:
                printf("Invalid choice\n");
        }
    }
    return 0;
}
```

Output:

"C:\Users\abhin\OneDrive\De

```
Current Min-Max Heap: 15 35 32 24 25

--- Min-Max Heap Operations ---
1. Insert
2. Delete (Min)
3. Search
4. Exit
Enter your choice: 1
Enter value to insert: 23

Current Min-Max Heap: 15 35 32 24 25 23

--- Min-Max Heap Operations ---
1. Insert
2. Delete (Min)
3. Search
4. Exit
Enter your choice: 2
Minimum element deleted

Current Min-Max Heap: 23 35 32 24 25

--- Min-Max Heap Operations ---
1. Insert
2. Delete (Min)
3. Search
4. Exit
Enter your choice: 3
Enter value to search: 32
Element found
```

Experiment-5:

5. Write a program to perform the following operations:
a) Insert an element into a Lefiist tree
b) Delete an element from a Leftist tree
c) Search for a key element in a Leftist tree

Program:

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

/* Leftist Tree Node */
struct node
{
   int data;
   int npl;              // Null Path Length
   struct node *left;
   struct node *right;
};

/* Create new node */
struct node* createNode(int value)
{
   struct node *temp = (struct node*)malloc(sizeof(struct node));
   temp->data = value;
   temp->npl = 0;
   temp->left = NULL;
   temp->right = NULL;
   return temp;
}

/* Merge two leftist trees */
struct node* merge(struct node *h1, struct node *h2)
{
   if (h1 == NULL)
      return h2;
   if (h2 == NULL)
      return h1;

   /* Ensure min-heap property */
   if (h1->data > h2->data)
   {
      struct node *temp = h1;
      h1 = h2;
      h2 = temp;
   }

   h1->right = merge(h1->right, h2);

   /* Maintain leftist property */
   if (h1->left == NULL ||
```

```
      (h1->right != NULL && h1->left->npl < h1->right->npl))
    {
        struct node *temp = h1->left;
        h1->left = h1->right;
        h1->right = temp;
    }

    if (h1->right == NULL)
        h1->npl = 0;
    else
        h1->npl = h1->right->npl + 1;

    return h1;
}

/* Insert element */
struct node* insert(struct node *root, int value)
{
    struct node *newNode = createNode(value);
    root = merge(root, newNode);
    return root;
}

/* Delete minimum element */
struct node* deleteMin(struct node *root)
{
    if (root == NULL)
    {
        printf("Tree is empty\n");
        return NULL;
    }

    struct node *leftTree = root->left;
    struct node *rightTree = root->right;
    printf("Deleted element: %d\n", root->data);
    free(root);

    return merge(leftTree, rightTree);
}

/* Search element */
void search(struct node *root, int key)
{
    if (root == NULL)
        return;

    if (root->data == key)
    {
        printf("Element found\n");
        return;
    }

    search(root->left, key);
```

```
    search(root->right, key);
}

/* Preorder traversal (display tree) */
void display(struct node *root)
{
    if (root != NULL)
    {
        printf("%d ", root->data);
        display(root->left);
        display(root->right);
    }
}

/* Main function */
int main()
{
    struct node *root = NULL;
    int choice, value, found;

    while (1)
    {
        /* Display tree BEFORE menu */
        printf("\nCurrent Leftist Tree: ");
        display(root);
        printf("\n");

        printf("\n--- Leftist Tree Operations ---");
        printf("\n1. Insert");
        printf("\n2. Delete (Min)");
        printf("\n3. Search");
        printf("\n4. Exit");
        printf("\nEnter your choice: ");
        scanf("%d", &choice);

        switch (choice)
        {
            case 1:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                root = insert(root, value);
                break;

            case 2:
                root = deleteMin(root);
                break;

            case 3:
                printf("Enter value to search: ");
                scanf("%d", &value);
                found = 0;
                search(root, value);
                break;
```

```
        case 4:
          exit(0);

        default:
          printf("Invalid choice\n");
      }
    }

  return 0;
}
```

Output:

```
"C:\Users\abhin\OneDrive\De

Current Leftist Tree: 22 46 64 52 35 44

--- Leftist Tree Operations ---
1. Insert
2. Delete (Min)
3. Search
4. Exit
Enter your choice: 1
Enter value to insert: 21

Current Leftist Tree: 21 22 46 64 52 35 44

--- Leftist Tree Operations ---
1. Insert
2. Delete (Min)
3. Search
4. Exit
Enter your choice: 2
Deleted element: 21

Current Leftist Tree: 22 46 64 52 35 44

--- Leftist Tree Operations ---
1. Insert
2. Delete (Min)
3. Search
4. Exit
Enter your choice: 3
Enter value to search: 64
Element found
```

Experiment-6:

6. Write a program to perform the following operations:
a) Insert an element into a binomial heap
b) Delete an element from a binomial heap.
c) Search for a key element in a binomial heap

Program:

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

/* Structure of Binomial Heap Node */
struct node {
   int key;
   int degree;
   struct node *parent;
   struct node *child;
   struct node *sibling;
};

struct node *heap = NULL;

/* Create a new node */
struct node* createNode(int key) {
   struct node *newNode = (struct node*)malloc(sizeof(struct node));
   newNode->key = key;
   newNode->degree = 0;
   newNode->parent = NULL;
   newNode->child = NULL;
   newNode->sibling = NULL;
   return newNode;
}

/* Merge two binomial trees */
struct node* mergeTrees(struct node *b1, struct node *b2) {
   if (b1->key > b2->key) {
      struct node *temp = b1;
      b1 = b2;
      b2 = temp;
   }

   b2->parent = b1;
   b2->sibling = b1->child;
   b1->child = b2;
   b1->degree++;

   return b1;
}
```

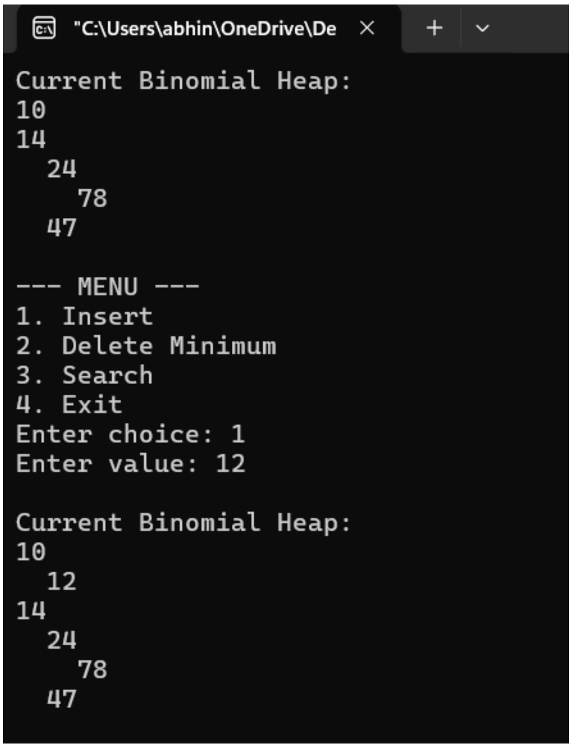
```
/* Merge root lists */
struct node* mergeHeaps(struct node *h1, struct node *h2) {
   if (!h1) return h2;
   if (!h2) return h1;

   struct node *head = NULL;
   struct node *tail = NULL;

   if (h1->degree <= h2->degree) {
      head = h1;
      h1 = h1->sibling;
   } else {
      head = h2;
      h2 = h2->sibling;
   }

   tail = head;

   while (h1 && h2) {
      if (h1->degree <= h2->degree) {
         tail->sibling = h1;
         h1 = h1->sibling;
      } else {
         tail->sibling = h2;
         h2 = h2->sibling;
      }
      tail = tail->sibling;
   }

   tail->sibling = (h1) ? h1 : h2;
   return head;
}

/* Union of two heaps */
struct node* unionHeap(struct node *h1, struct node *h2) {
   struct node *newHeap = mergeHeaps(h1, h2);
   if (!newHeap) return NULL;

   struct node *prev = NULL;
   struct node *curr = newHeap;
   struct node *next = curr->sibling;

   while (next) {
      if ((curr->degree != next->degree) ||
         (next->sibling && next->sibling->degree == curr->degree)) {
         prev = curr;
         curr = next;
      } else {
         if (curr->key <= next->key) {
            curr->sibling = next->sibling;
            mergeTrees(curr, next);
         } else {
```

```
            if (prev == NULL)
                newHeap = next;
            else
                prev->sibling = next;

            mergeTrees(next, curr);
            curr = next;
        }
    }
    next = curr->sibling;
  }

  return newHeap;
}

/* Insert */
void insert(int key) {
  struct node *newNode = createNode(key);
  heap = unionHeap(heap, newNode);
}

/* Find minimum node */
struct node* findMin() {
  struct node *temp = heap;
  struct node *minNode = NULL;
  int min = INT_MAX;

  while (temp) {
    if (temp->key < min) {
      min = temp->key;
      minNode = temp;
    }
    temp = temp->sibling;
  }
  return minNode;
}

/* Delete minimum */
void deleteMin() {
  if (!heap) {
    printf("Heap is empty\n");
    return;
  }

  struct node *minPrev = NULL;
  struct node *minNode = heap;
  struct node *prev = NULL;
  struct node *curr = heap;

  int min = curr->key;

  while (curr) {
    if (curr->key < min) {
```

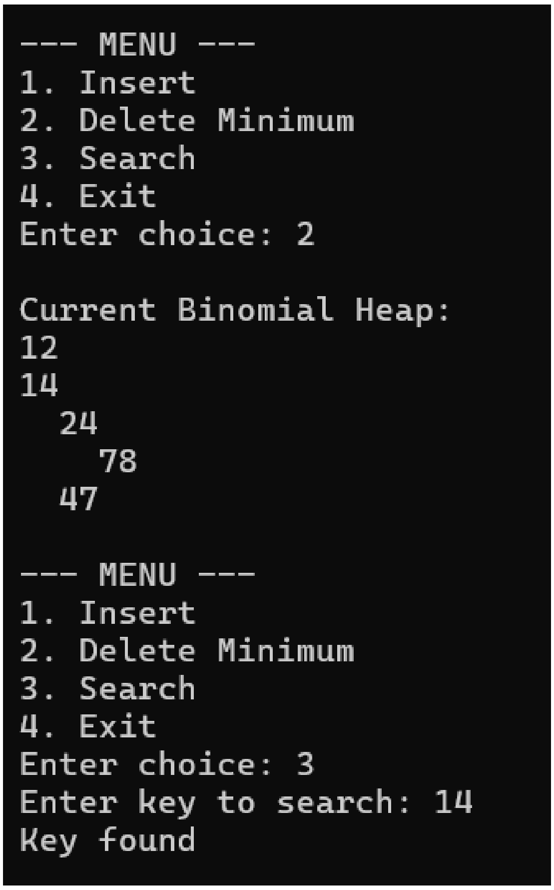
```
            min = curr->key;
            minNode = curr;
            minPrev = prev;
        }
        prev = curr;
        curr = curr->sibling;
    }

    if (minPrev)
        minPrev->sibling = minNode->sibling;
    else
        heap = minNode->sibling;

    struct node *child = minNode->child;
    struct node *rev = NULL;

    while (child) {
        struct node *next = child->sibling;
        child->sibling = rev;
        child->parent = NULL;
        rev = child;
        child = next;
    }

    heap = unionHeap(heap, rev);
    free(minNode);
}

/* Search */
struct node* search(struct node *root, int key) {
    if (!root) return NULL;
    if (root->key == key) return root;

    struct node *found = search(root->child, key);
    if (found) return found;

    return search(root->sibling, key);
}

/* Display */
void display(struct node *h, int level) {
    while (h) {
        for (int i = 0; i < level; i++)
            printf("  ");
        printf("%d\n", h->key);
        display(h->child, level + 1);
        h = h->sibling;
    }
}

/* Main */
int main() {
    int choice, value;
```

```
    while (1) {
        printf("\nCurrent Binomial Heap:\n");
        display(heap, 0);

        printf("\n--- MENU ---\n");
        printf("1. Insert\n");
        printf("2. Delete Minimum\n");
        printf("3. Search\n");
        printf("4. Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value: ");
                scanf("%d", &value);
                insert(value);
                break;

            case 2:
                deleteMin();
                break;

            case 3:
                printf("Enter key to search: ");
                scanf("%d", &value);
                if (search(heap, value))
                    printf("Key found\n");
                else
                    printf("Key not found\n");
                break;

            case 4:
                exit(0);

            default:
                printf("Invalid choice\n");
        }
    }
}
```

Output:



```
"C:\Users\abhin\OneDrive\De

Current Binomial Heap:
10
14
  24
    78
  47

--- MENU ---
1. Insert
2. Delete Minimum
3. Search
4. Exit
Enter choice: 1
Enter value: 12

Current Binomial Heap:
10
  12
14
  24
    78
  47
```



```
--- MENU ---
1. Insert
2. Delete Minimum
3. Search
4. Exit
Enter choice: 2

Current Binomial Heap:
12
14
  24
    78
  47

--- MENU ---
1. Insert
2. Delete Minimum
3. Search
4. Exit
Enter choice: 3
Enter key to search: 14
Key found
```

Experiment-7:

7. Write a program to perform the following operations:
a) Insert an element into a AVL tree.
b) Delete an element from a AVL search tree.
c) Search for a key element in a AVL search tree.

Program:

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

/* AVL Tree Node */
struct node {
   int key;
   struct node *left;
   struct node *right;
   int height;
};

/* Get height */
int height(struct node *n) {
   if (n == NULL)
      return 0;
   return n->height;
}

/* Maximum */
int max(int a, int b) {
   return (a > b) ? a : b;
}

/* Create new node */
struct node* newNode(int key) {
   struct node* node = (struct node*)malloc(sizeof(struct node));
   node->key = key;
   node->left = NULL;
   node->right = NULL;
   node->height = 1;
   return node;
}

/* Right Rotation */
struct node* rightRotate(struct node *y) {
   struct node *x = y->left;
   struct node *T2 = x->right;

   x->right = y;
   y->left = T2;

   y->height = max(height(y->left), height(y->right)) + 1;
   x->height = max(height(x->left), height(x->right)) + 1;
```

```c
    return x;
}

/* Left Rotation */
struct node* leftRotate(struct node *x) {
    struct node *y = x->right;
    struct node *T2 = y->left;

    y->left = x;
    x->right = T2;

    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;

    return y;
}

/* Balance Factor */
int getBalance(struct node *n) {
    if (n == NULL)
        return 0;
    return height(n->left) - height(n->right);
}

/* Insert */
struct node* insert(struct 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;   // No duplicates

    node->height = 1 + max(height(node->left), height(node->right));

    int balance = getBalance(node);

    // LL
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // RR
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // LR
    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }
```

```
    // RL
    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    return node;
}

/* Find minimum */
struct node* minValueNode(struct node* node) {
    struct node* current = node;
    while (current->left != NULL)
        current = current->left;
    return current;
}

/* Delete */
struct node* deleteNode(struct 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)) {
            struct node *temp = root->left ? root->left : root->right;

            if (temp == NULL) {
                temp = root;
                root = NULL;
            } else
                *root = *temp;

            free(temp);
        } else {
            struct node* temp = minValueNode(root->right);
            root->key = temp->key;
            root->right = deleteNode(root->right, temp->key);
        }
    }

    if (root == NULL)
        return root;

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

    // LL
    if (balance > 1 && getBalance(root->left) >= 0)
```

```
        return rightRotate(root);

    // LR
    if (balance > 1 && getBalance(root->left) < 0) {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }

    // RR
    if (balance < -1 && getBalance(root->right) <= 0)
        return leftRotate(root);

    // RL
    if (balance < -1 && getBalance(root->right) > 0) {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}

/* Search */
int search(struct node* root, int key) {
    if (root == NULL)
        return 0;
    if (root->key == key)
        return 1;
    if (key < root->key)
        return search(root->left, key);
    return search(root->right, key);
}

/* Display Tree (Sideways) */
void display(struct node *root, int space) {
    if (root == NULL)
        return;

    space += 5;
    display(root->right, space);

    printf("\n");
    for (int i = 5; i < space; i++)
        printf(" ");
    printf("%d\n", root->key);

    display(root->left, space);
}

/* Main */
int main() {
    struct node* root = NULL;
    int choice, value;
```

```
    while (1) {
        printf("\nCurrent AVL Tree:\n");
        display(root, 0);

        printf("\n--- MENU ---\n");
        printf("1. Insert\n");
        printf("2. Delete\n");
        printf("3. Search\n");
        printf("4. Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value: ");
                scanf("%d", &value);
                root = insert(root, value);
                break;

            case 2:
                printf("Enter value to delete: ");
                scanf("%d", &value);
                root = deleteNode(root, value);
                break;

            case 3:
                printf("Enter value to search: ");
                scanf("%d", &value);
                if (search(root, value))
                    printf("Key found\n");
                else
                    printf("Key not found\n");
                break;

            case 4:
                exit(0);

            default:
                printf("Invalid choice\n");
        }
    }
}
```

Output:

```
Current AVL Tree:

          36

     35

          34

33

     32

          31

--- MENU ---
1. Insert
2. Delete
3. Search
4. Exit
Enter choice: 1
Enter value: 30

Current AVL Tree:

          36

     35

          34

33

          32

     31

          30
```

```
--- MENU ---
1. Insert
2. Delete
3. Search
4. Exit
Enter choice: 2
Enter value to delete: 30

Current AVL Tree:

          36

     35

          34

33

          32

     31

--- MENU ---
1. Insert
2. Delete
3. Search
4. Exit
Enter choice: 3
Enter value to search: 32
Key found
```

Experiment-8:

8. Write a program to perform the following operations:
a) Insert an element into a Red-Black tree.
b) Delete an element from a Red-Black tree.
c) Search for a key element in a Red-Black tree.

Program:

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

#define RED 1
#define BLACK 0

/* Node structure */
struct node {
    int data;
    int color;
    struct node *left, *right, *parent;
};

struct node *root = NULL;
struct node *NIL;

/* Create NIL node */
void initNIL() {
    NIL = (struct node*)malloc(sizeof(struct node));
    NIL->color = BLACK;
    NIL->left = NIL->right = NIL->parent = NULL;
}

/* Create new node */
struct node* createNode(int data) {
    struct node *n = (struct node*)malloc(sizeof(struct node));
    n->data = data;
    n->color = RED;
    n->left = n->right = n->parent = NIL;
    return n;
}

/* Left Rotate */
void leftRotate(struct node *x) {
    struct node *y = x->right;
    x->right = y->left;

    if (y->left != NIL)
        y->left->parent = x;

    y->parent = x->parent;

    if (x->parent == NIL)
```

```
        root = y;
    else if (x == x->parent->left)
        x->parent->left = y;
    else
        x->parent->right = y;

    y->left = x;
    x->parent = y;
}

/* Right Rotate */
void rightRotate(struct node *y) {
    struct node *x = y->left;
    y->left = x->right;

    if (x->right != NIL)
        x->right->parent = y;

    x->parent = y->parent;

    if (y->parent == NIL)
        root = x;
    else if (y == y->parent->right)
        y->parent->right = x;
    else
        y->parent->left = x;

    x->right = y;
    y->parent = x;
}

/* Fix after insert */
void fixInsert(struct node *z) {
    while (z->parent->color == RED) {
        if (z->parent == z->parent->parent->left) {
            struct node *y = z->parent->parent->right;

            if (y->color == RED) {
                z->parent->color = BLACK;
                y->color = BLACK;
                z->parent->parent->color = RED;
                z = z->parent->parent;
            } else {
                if (z == z->parent->right) {
                    z = z->parent;
                    leftRotate(z);
                }
                z->parent->color = BLACK;
                z->parent->parent->color = RED;
                rightRotate(z->parent->parent);
            }
        } else {
            struct node *y = z->parent->parent->left;
```

```
        if (y->color == RED) {
          z->parent->color = BLACK;
          y->color = BLACK;
          z->parent->parent->color = RED;
          z = z->parent->parent;
        } else {
          if (z == z->parent->left) {
            z = z->parent;
            rightRotate(z);
          }
          z->parent->color = BLACK;
          z->parent->parent->color = RED;
          leftRotate(z->parent->parent);
        }
      }
    }
  root->color = BLACK;
}

/* Insert */
void insert(int data) {
  struct node *z = createNode(data);
  struct node *y = NIL;
  struct node *x = root;

  while (x != NIL) {
    y = x;
    if (z->data < x->data)
      x = x->left;
    else
      x = x->right;
  }

  z->parent = y;

  if (y == NIL)
    root = z;
  else if (z->data < y->data)
    y->left = z;
  else
    y->right = z;

  fixInsert(z);
}

/* Search */
int search(struct node *root, int key) {
  if (root == NIL)
    return 0;
  if (root->data == key)
    return 1;
  if (key < root->data)
```

```
        return search(root->left, key);
    return search(root->right, key);
}

/* Minimum */
struct node* minimum(struct node *node) {
    while (node->left != NIL)
        node = node->left;
    return node;
}

/* Transplant */
void transplant(struct node *u, struct node *v) {
    if (u->parent == NIL)
        root = v;
    else if (u == u->parent->left)
        u->parent->left = v;
    else
        u->parent->right = v;
    v->parent = u->parent;
}

/* Fix delete */
void fixDelete(struct node *x) {
    while (x != root && x->color == BLACK) {
        if (x == x->parent->left) {
            struct node *w = x->parent->right;

            if (w->color == RED) {
                w->color = BLACK;
                x->parent->color = RED;
                leftRotate(x->parent);
                w = x->parent->right;
            }

            if (w->left->color == BLACK && w->right->color == BLACK) {
                w->color = RED;
                x = x->parent;
            } else {
                if (w->right->color == BLACK) {
                    w->left->color = BLACK;
                    w->color = RED;
                    rightRotate(w);
                    w = x->parent->right;
                }
                w->color = x->parent->color;
                x->parent->color = BLACK;
                w->right->color = BLACK;
                leftRotate(x->parent);
                x = root;
            }
        } else {
            struct node *w = x->parent->left;
```

```
      if (w->color == RED) {
        w->color = BLACK;
        x->parent->color = RED;
        rightRotate(x->parent);
        w = x->parent->left;
      }

      if (w->right->color == BLACK && w->left->color == BLACK) {
        w->color = RED;
        x = x->parent;
      } else {
        if (w->left->color == BLACK) {
          w->right->color = BLACK;
          w->color = RED;
          leftRotate(w);
          w = x->parent->left;
        }
        w->color = x->parent->color;
        x->parent->color = BLACK;
        w->left->color = BLACK;
        rightRotate(x->parent);
        x = root;
      }
    }
  }
  x->color = BLACK;
}

/* Delete */
void delete(int key) {
  struct node *z = root;
  while (z != NIL && z->data != key) {
    if (key < z->data)
      z = z->left;
    else
      z = z->right;
  }

  if (z == NIL) {
    printf("Key not found\n");
    return;
  }

  struct node *y = z;
  int yColor = y->color;
  struct node *x;

  if (z->left == NIL) {
    x = z->right;
    transplant(z, z->right);
  } else if (z->right == NIL) {
    x = z->left;
```

```
        transplant(z, z->left);
    } else {
        y = minimum(z->right);
        yColor = y->color;
        x = y->right;

        if (y->parent == z)
            x->parent = y;
        else {
            transplant(y, y->right);
            y->right = z->right;
            y->right->parent = y;
        }

        transplant(z, y);
        y->left = z->left;
        y->left->parent = y;
        y->color = z->color;
    }

    free(z);

    if (yColor == BLACK)
        fixDelete(x);
}

/* Display tree (sideways) */
void display(struct node *root, int space) {
    if (root == NIL)
        return;

    space += 5;
    display(root->right, space);

    printf("\n");
    for (int i = 5; i < space; i++)
        printf(" ");
    printf("%d(%c)\n", root->data, root->color == RED ? 'R' : 'B');

    display(root->left, space);
}

/* Main */
int main() {
    int choice, value;
    initNIL();
    root = NIL;

    while (1) {
        printf("\nCurrent Red-Black Tree:\n");
        display(root, 0);

        printf("\n--- MENU ---\n");
```

```
        printf("1. Insert\n");
        printf("2. Delete\n");
        printf("3. Search\n");
        printf("4. Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value: ");
                scanf("%d", &value);
                insert(value);
                break;

            case 2:
                printf("Enter value to delete: ");
                scanf("%d", &value);
                delete(value);
                break;

            case 3:
                printf("Enter value to search: ");
                scanf("%d", &value);
                if (search(root, value))
                    printf("Key found\n");
                else
                    printf("Key not found\n");
                break;

            case 4:
                exit(0);

            default:
                printf("Invalid choice\n");
        }
    }
}
```

Output:

```
Current Red-Black Tree:

     20(R)

10(B)

--- MENU ---
1. Insert
2. Delete
3. Search
4. Exit
Enter choice: 1
Enter value: 30

Current Red-Black Tree:

     30(R)

20(B)

     10(R)

--- MENU ---
1. Insert
2. Delete
3. Search
4. Exit
```

```
Enter choice: 2
Enter value to delete: 10

Current Red-Black Tree:

     30(R)

20(B)

--- MENU ---
1. Insert
2. Delete
3. Search
4. Exit
Enter choice: 3
Enter value to search: 20
Key found

Current Red-Black Tree:

     30(R)

20(B)
```

Experiment-9:

9. Write a program to implement all the functions of a dictionary using hashing.

Program:

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

#define SIZE 10   // Size of hash table

// Structure for dictionary node
struct Node {
    char word[30];
    char meaning[100];
    struct Node *next;
};

// Hash table
struct Node *hashTable[SIZE];

// Hash function
int hashFunction(char word[]) {
    int sum = 0;
    for (int i = 0; word[i] != '\0'; i++) {
        sum += word[i];
    }
    return sum % SIZE;
}

// Create a new node
struct Node* createNode(char word[], char meaning[]) {
    struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
    strcpy(newNode->word, word);
    strcpy(newNode->meaning, meaning);
    newNode->next = NULL;
    return newNode;
}

// Insert word into dictionary
void insert(char word[], char meaning[]) {
    int index = hashFunction(word);
    struct Node *newNode = createNode(word, meaning);

    if (hashTable[index] == NULL) {
        hashTable[index] = newNode;
    } else {
        struct Node *temp = hashTable[index];
        while (temp->next != NULL) {
            temp = temp->next;
```

```
        }
        temp->next = newNode;
    }

    printf("Word inserted successfully.\n");
}

// Search for a word
void search(char word[]) {
    int index = hashFunction(word);
    struct Node *temp = hashTable[index];

    while (temp != NULL) {
        if (strcmp(temp->word, word) == 0) {
            printf("Word found!\nMeaning: %s\n", temp->meaning);
            return;
        }
        temp = temp->next;
    }

    printf("Word not found in dictionary.\n");
}

// Delete a word
void deleteWord(char word[]) {
    int index = hashFunction(word);
    struct Node *temp = hashTable[index];
    struct Node *prev = NULL;

    while (temp != NULL) {
        if (strcmp(temp->word, word) == 0) {
            if (prev == NULL) {
                hashTable[index] = temp->next;
            } else {
                prev->next = temp->next;
            }
            free(temp);
            printf("Word deleted successfully.\n");
            return;
        }
        prev = temp;
        temp = temp->next;
    }

    printf("Word not found. Cannot delete.\n");
}

// Display dictionary
void display() {
```

```
    printf("\nDictionary Contents:\n");
    for (int i = 0; i < SIZE; i++) {
        struct Node *temp = hashTable[i];
        if (temp != NULL) {
            printf("Index %d:\n", i);
            while (temp != NULL) {
                printf("  %s : %s\n", temp->word, temp->meaning);
                temp = temp->next;
            }
        }
    }
}

// Main function
int main() {
    int choice;
    char word[30], meaning[100];

    // Initialize hash table
    for (int i = 0; i < SIZE; i++) {
        hashTable[i] = NULL;
    }

    do {
        printf("\n--- Dictionary Using Hashing ---\n");
        printf("1. Insert Word\n");
        printf("2. Search Word\n");
        printf("3. Delete Word\n");
        printf("4. Display Dictionary\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter word: ");
                scanf("%s", word);
                printf("Enter meaning: ");
                scanf(" %[^\n]", meaning);
                insert(word, meaning);
                break;

            case 2:
                printf("Enter word to search: ");
                scanf("%s", word);
                search(word);
                break;

            case 3:
```

```
            printf("Enter word to delete: ");
            scanf("%s", word);
            deleteWord(word);
            break;

        case 4:
            display();
            break;

        case 5:
            printf("Exiting program.\n");
            break;

        default:
            printf("Invalid choice.\n");
      }

   } while (choice != 5);

   return 0;
}
```

Output:

```
--- Dictionary Using Hashing ---
1. Insert Word
2. Search Word
3. Delete Word
4. Display Dictionary
5. Exit
Enter your choice: 1
Enter word: Apple
Enter meaning: A fruit
Word inserted successfully.

--- Dictionary Using Hashing ---
1. Insert Word
2. Search Word
3. Delete Word
4. Display Dictionary
5. Exit
Enter your choice: 1
Enter word: book
Enter meaning: A collection of pages
Word inserted successfully.
```

```
--- Dictionary Using Hashing ---
1. Insert Word
2. Search Word
3. Delete Word
4. Display Dictionary
5. Exit
Enter your choice: 2
Enter word to search: Apple
Word found!
Meaning: A fruit

--- Dictionary Using Hashing ---
1. Insert Word
2. Search Word
3. Delete Word
4. Display Dictionary
5. Exit
Enter your choice: 3
Enter word to delete: Apple
Word deleted successfully.
```

Experiment-10:

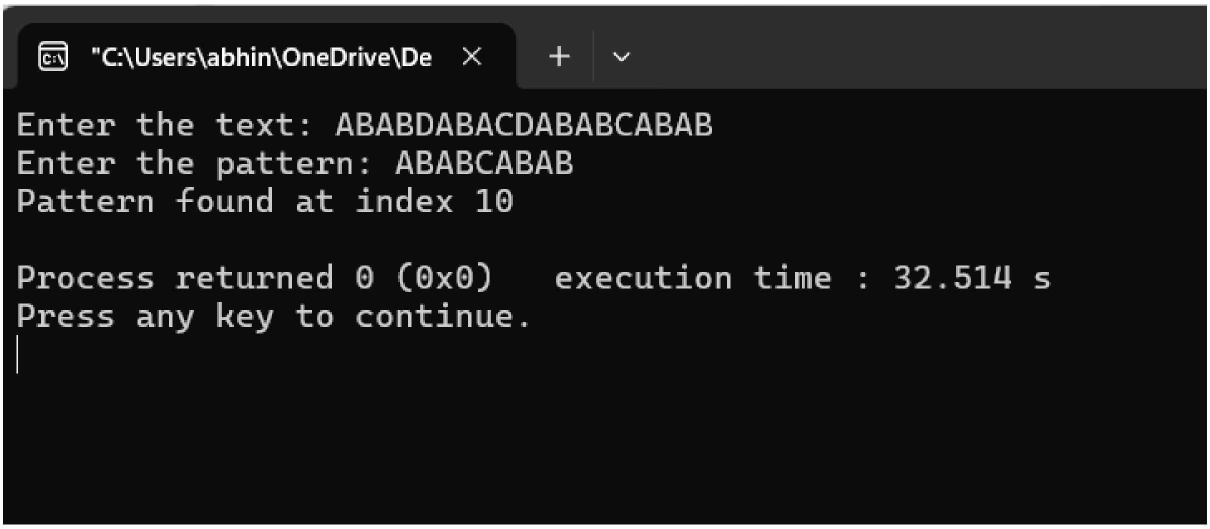
10. Write a program for implementing Knuth-Morris-Pratt pattern matching algorithm.

Program:

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

// Function to compute LPS array
void computeLPS(char pattern[], int m, int lps[]) {
    int length = 0;   // length of previous longest prefix suffix
    lps[0] = 0;       // lps[0] is always 0

    int i = 1;
    while (i < m) {
        if (pattern[i] == pattern[length]) {
            length++;
            lps[i] = length;
            i++;
        } else {
            if (length != 0) {
                length = lps[length - 1];
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }
}

// KMP search function
void KMPSearch(char text[], char pattern[]) {
    int n = strlen(text);
    int m = strlen(pattern);

    int lps[m];

    // Compute LPS array
    computeLPS(pattern, m, lps);

    int i = 0; // index for text
    int j = 0; // index for pattern

    while (i < n) {
        if (pattern[j] == text[i]) {
            i++;
            j++;
```

```
        }

        if (j == m) {
            printf("Pattern found at index %d\n", i - j);
            j = lps[j - 1];
        }
        else if (i < n && pattern[j] != text[i]) {
            if (j != 0) {
                j = lps[j - 1];
            } else {
                i++;
            }
        }
    }
}

// Main function
int main() {
    char text[100], pattern[50];

    printf("Enter the text: ");
    gets(text);

    printf("Enter the pattern: ");
    gets(pattern);

    KMPSearch(text, pattern);

    return 0;
}
```

Output:



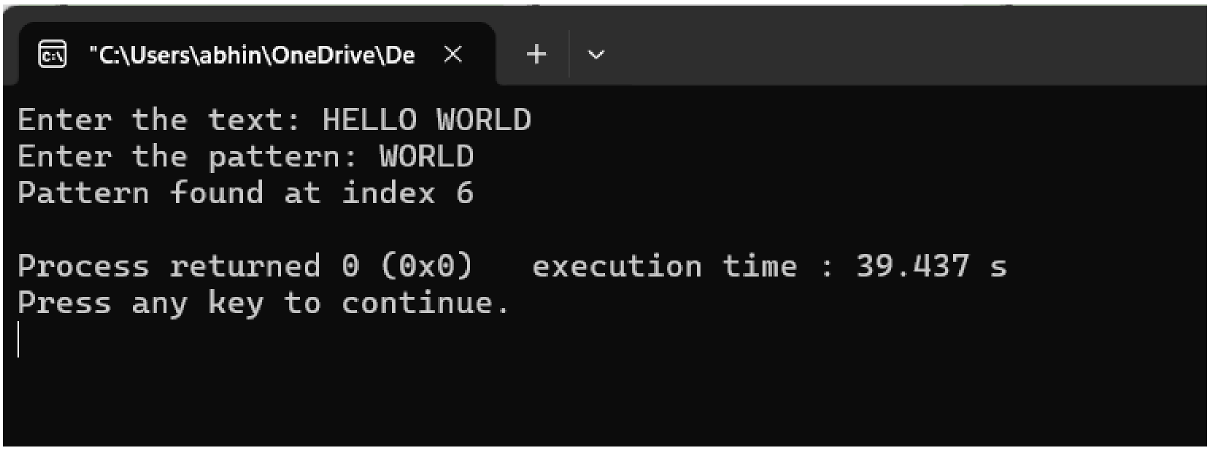
Experiment-11:

11. Write a program for implementing Brute Force pattern matching algorithm.

Program:

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

int main() {
   char text[100], pattern[50];
   int i, j;
   int found = 0;

   printf("Enter the text: ");
   gets(text);

   printf("Enter the pattern: ");
   gets(pattern);

   int n = strlen(text);
   int m = strlen(pattern);

   // Brute force pattern matching
   for (i = 0; i <= n - m; i++) {
      for (j = 0; j < m; j++) {
         if (text[i + j] != pattern[j]) {
            break;
         }
      }

      if (j == m) {
         printf("Pattern found at index %d\n", i);
         found = 1;
      }
   }

   if (!found) {
      printf("Pattern not found\n");
   }

   return 0;
}
```

Output:


"C:\Users\abhin\OneDrive\De
Enter the text: HELLO WORLD
Enter the pattern: WORLD
Pattern found at index 6
Process returned 0 (0x0)   execution time : 39.437 s
Press any key to continue.

Experiment-12:

12. Write a program for implementing Boyer pattern matching algorithm.

Program:

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

#define MAX 256   // Total ASCII characters

// Function to create bad character table
void badCharTable(char pattern[], int m, int badChar[]) {
   int i;

   // Initialize all values as -1
   for (i = 0; i < MAX; i++)
      badChar[i] = -1;

   // Fill actual values of last occurrence
   for (i = 0; i < m; i++)
      badChar[(int)pattern[i]] = i;
}

// Boyer–Moore search function
void boyerMoore(char text[], char pattern[]) {
   int n = strlen(text);
   int m = strlen(pattern);

   int badChar[MAX];
   badCharTable(pattern, m, badChar);

   int shift = 0;  // shift of the pattern

   while (shift <= (n - m)) {
      int j = m - 1;

      // Compare from right to left
      while (j >= 0 && pattern[j] == text[shift + j])
         j--;

      // If pattern matches
      if (j < 0) {
         printf("Pattern found at index %d\n", shift);

         // Shift pattern
         shift += (shift + m < n) ?
                 m - badChar[text[shift + m]] : 1;
      }
      else {
```

```
            // Shift using bad character rule
            int bcIndex = badChar[text[shift + j]];
            shift += (j - bcIndex > 1) ? j - bcIndex : 1;
        }
    }
}

// Main function
int main() {
    char text[100], pattern[50];

    printf("Enter the text: ");
    gets(text);

    printf("Enter the pattern: ");
    gets(pattern);

    boyerMoore(text, pattern);

    return 0;
}
```

Output: