KG REDDY College of Engineering & Technology AN AUTONOMOUS INSTITUTION

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING

ADVANCED DATA STRUCTURE LAB MANUAL

Subject Code: KG2458103

Regulation:KGR24

Academic Year: 2024-25

I M. TECH I SEMESTER

COMPUTER SCIENCE AND ENGINEERING

KG REDDY COLLEGE OF ENGINEERING AND TECHNOLOGY

Affiliated To JNTUH, Chilkur,(V), Moinabad(M) R. R Dist, TS-501504

KG REDDY
College of Engineering & Technology
AN AUTONOMOUS INSTITUTION

VISION AND MISSION OF THE INSTITUTION

VISION:

To become an institution which is internationally recognized for its holistic approach to engineering, innovative teaching and learning culture, research and entrepreneurial ecosystem, and sustainable social impact in the community.

MISSION:

- To offer undergraduate and post-graduate programs which are supported through industry relevant curriculum and innovative teaching and learning processes that would help students succeed in their professional careers.
- To provide faculty and students with an ecosystem that fosters innovation, research, entrepreneurship, and international exposure through strategic partnerships with government organizations and collaboration with industries.
- To provide holistic learning environment to students which will contribute to their personal and professional growth and enable them to become leaders in their respective fields.
- To contribute to the development of the region by using our technological expertise to work with nearby communities and support them in their social and economic development.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

VISION:

To be recognized as a department of excellence by stimulating a learning environment in which students and faculty will thrive and grow to achieve their professional, institutional and societal goals.

MISSION:

• To provide high quality technical education to students that will enable life-long learning and build expertise in advanced technologies in Computer Science and Engineering.
• To promote research and development by providing opportunities to solve complex engineering problems in collaboration with industry and government agencies.
• To encourage professional development of students that will inculcate ethical values and leadership skills through entrepreneurship while working with the community to address societal issues.

PROGRAM EDUCATIONAL OBJECTIVES

PEO 1: Graduates will provide solutions to difficult and challenging issues in their profession by applying computer science and engineering theory and principles.
PEO 2: Graduates have successful careers in computer science and engineering fields or will be able to successfully pursue advanced degrees.
PEO 3: Graduates will communicate effectively, work collaboratively and exhibit high levels of professionalism, moral and ethical responsibility.
PEO 4: Graduates will develop the ability to understand and analyze engineering issues in a broader perspective with ethical responsibility towards sustainable development.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PROGRAM OUTCOMES

• **PO I: Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
• **PO II: Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
• **PO III: Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
• **PO IV: Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
• **PO V: Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
• **PO VI: The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
• **PO VII: Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of and need for sustainable development.
• **PO VIII: Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
• **PO IX: Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
• **PO X: Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

• **PO XI: Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
• **PO XII: Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PROGRAM SPECIFIC OUTCOMES

PSO1: The Computer Science and Engineering graduates are able to analyze, design, develop, test and apply management principles, mathematical foundations in the development of intelligent systems with computational solutions, make them to expert in designing the secure application and hardware prototype

PSO2: The graduating student will be analyze the contemporary research issues in different areas of computer science & engineering and explore research gaps, analyze and carry out research in the specialized/emerging areas.

PSO3: Develop their skills to solve problems in the broad area of programming concepts and appraise environmental and social issues with ethics and manage different projects in multi-disciplinary field to conducive in cultivating skills for successful career, entrepreneurship and higher studies.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ADVANCED DATA STRUCTURES LAB

M.Tech CSE
Prerequisites:
1. A course on Computer Programming & Data Structures

Course Objectives:

1. Introduces the basic concepts of Abstract Data Types.
2. Reviews basic data structures such as stacks and queues.
3. Introduces a variety of data structures such as hash tables, search trees, tries, heaps, graphs, and B-trees.
4. Introduces sorting and pattern matching algorithms.
5. Able to identify and use a suitable data structure and algorithm to solve a real world problem

Course Outcomes:

1. Ability to select the data structures that efficiently model the information in a problem.
2. Ability to assess efficiency trade-offs among different data structure implementations or Combinations.
3. Implement and know the application of algorithms for sorting and pattern matching.
4. Apply Non-linear data structures: Trees and Graphs.
5. Design programs using a variety of data structures, including hash tables, binary and general tries, heaps and B-trees.

List of Programs

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.

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

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.

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

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

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

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.

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.

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

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

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

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

TEXT BOOKS:

1. Fundamentals of Data structures in C, E. Horowitz, S. Sahni and Susan Anderson Freed, 2nd Edition, Universities Press
2. Data Structures Using C – A.S. Tanenbaum, Y. Langsam, and M.J. Augenstein, PHI/Pearson education.
3. Introduction to Data Structures in C, Ashok Kamthane, 1st Edition, Pearson.

REFERENCES:

1. The C Programming Language, B.W. Kernighan, Dennis M. Ritchie, PHI/Pearson Education
2. C Programming with problem solving, J.A. Jones & K. Harrow, Dreamtech Press
3. Data structures: A Pseudocode Approach with C, R.F. Gilberg And B.A. Forouzan, 2nd Edition,
Cengage Learning

}

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.

```
// C program to demonstrate insert

// operation in binary

// search tree.

#include <stdio.h>

#include <stdlib.h>


struct node {

        int key;

        struct node *left, *right;

};


// A utility function to create a new BST node

struct node* newNode(int item)

{

        struct node* temp

                = (struct node*)malloc(sizeof(struct node));

        temp->key = item;

        temp->left = temp->right = NULL;

        return temp;

}
```

```
// A utility function to do inorder traversal of BST
void inorder(struct node* root)
{
    if (root != NULL) {
        inorder(root->left);
        printf("%d \n", root->key);
        inorder(root->right);
    }
}

/* A utility function to insert
a new node with given key in
* BST */
struct node* insert(struct node* node, int key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL)
        return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
```

```
        /* return the (unchanged) node pointer */
        return node;
}

// Driver Code
int main()
{
        /* Let us create following BST
                    50
                /       \
                30      70
                / \ / \
        20 40 60 80 */
        struct node* root = NULL;
        root = insert(root, 50);
        insert(root, 30);
        insert(root, 20);
        insert(root, 40);
        insert(root, 70);
        insert(root, 60);
        insert(root, 80);

        // print inoder traversal of the BST
        inorder(root);
```

```
        return 0;
}
```

Output

20

30

40

50

60

70

80

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

A) /* C program for Merge Sort */

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


// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r)
{
        int i, j, k;
        int n1 = m - l + 1;
        int n2 = r - m;
```

```
/* create temp arrays */
int L[n1], R[n2];

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

/* Merge the temp arrays back into arr[l..r]*/
i = 0; // Initial index of first subarray
j = 0; // Initial index of second subarray
k = l; // Initial index of merged subarray
while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
                arr[k] = L[i];
                i++;
        }
        else {
                arr[k] = R[j];
                j++;
        }
        k++;
}
```

```
        /* Copy the remaining elements of L[], if there
        are any */
        while (i < n1) {
                arr[k] = L[i];
                i++;
                k++;
        }

        /* Copy the remaining elements of R[], if there
        are any */
        while (j < n2) {
                arr[k] = R[j];
                j++;
                k++;
        }
}

/* l is for left index and r is right index of the
sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
        if (l < r) {
                // Same as (l+r)/2, but avoids overflow for
                // large l and h
```

```
        int m = l + (r - l) / 2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}

/* UTILITY FUNCTIONS */
/* Function to print an array */
void printArray(int A[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n");
}

/* Driver code */
int main()
{
    int arr[] = { 12, 11, 13, 5, 6, 7 };
    int arr_size = sizeof(arr) / sizeof(arr[0]);
```

```
    printf("Given array is \n");
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);

    printf("\nSorted array is \n");
    printArray(arr, arr_size);
    return 0;
}
```

Output

Given array is

12 11 13 5 6 7

Sorted array is

5 6 7 11 12 13

B) // Heap Sort in C

```
#include <stdio.h>

// Function to swap the position of two elements

void swap(int* a, int* b)
{

    int temp = *a;
```

```
    *a = *b;

    *b = temp;
}

// To heapify a subtree rooted with node i
// which is an index in arr[].
// n is size of heap
void heapify(int arr[], int N, int i)
{
    // Find largest among root, left child and right child

    // Initialize largest as root
    int largest = i;

    // left = 2*i + 1
    int left = 2 * i + 1;

    // right = 2*i + 2
    int right = 2 * i + 2;

    // If left child is larger than root
    if (left < N && arr[left] > arr[largest])
```

```
        largest = left;

    // If right child is larger than largest
    // so far
    if (right < N && arr[right] > arr[largest])

        largest = right;

    // Swap and continue heapifying if root is not largest
    // If largest is not root
    if (largest != i) {

        swap(&arr[i], &arr[largest]);

        // Recursively heapify the affected
        // sub-tree
        heapify(arr, N, largest);
    }
}

// Main function to do heap sort
void heapSort(int arr[], int N)
{

    // Build max heap
```

```
	for (int i = N / 2 - 1; i >= 0; i--)

		heapify(arr, N, i);

	// Heap sort
	for (int i = N - 1; i >= 0; i--) {

		swap(&arr[0], &arr[i]);

		// Heapify root element to get highest element at
		// root again
		heapify(arr, i, 0);
	}
}

// A utility function to print array of size n
void printArray(int arr[], int N)
{
	for (int i = 0; i < N; i++)
		printf("%d ", arr[i]);
	printf("\n");
}

// Driver's code
int main()
```

```
{
    int arr[] = { 12, 11, 13, 5, 6, 7 };
    int N = sizeof(arr) / sizeof(arr[0]);

    // Function call
    heapSort(arr, N);
    printf("Sorted array is\n");
    printArray(arr, N);
}

// This code is contributed by _i_plus_plus_.
```

Output

```
Sorted array is
5 6 7 11 12 13
```

C) QUICK SORT

```
#include <stdio.h>

// Function to swap two elements
void swap(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}
```

```
// Partition the array using the last element as the pivot
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++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

// Function to implement 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);
    }
}

// Function to print the array
```

```
void printArray(int arr[], int size) {
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Driver program
int main() {
    int arr[] = { 10, 7, 8, 9, 1, 5 };
    int n = sizeof(arr) / sizeof(arr[0]);
    quickSort(arr, 0, n - 1);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

Output

Sorted array:

1 5 7 8 9 10

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.

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

#define MAX 4
#define MIN 2

struct btreeNode {
    int val[MAX + 1], count;
    struct btreeNode *link[MAX + 1];
};

struct btreeNode *root;

/* creating new node */
struct btreeNode * createNode(int val, struct btreeNode *child) {
    struct btreeNode *newNode;
    newNode = (struct btreeNode *)malloc(sizeof(struct btreeNode));
    newNode->val[1] = val;
    newNode->count = 1;
    newNode->link[0] = root;
    newNode->link[1] = child;
    return newNode;
}

/* Places the value in appropriate position */
void addValToNode(int val, int pos, struct btreeNode *node,
                    struct btreeNode *child) {
    int j = node->count;
    while (j > pos) {
        node->val[j + 1] = node->val[j];
        node->link[j + 1] = node->link[j];
        j--;
    }
    node->val[j + 1] = val;
    node->link[j + 1] = child;
```

```
    node->count++;
}

/* split the node */
void splitNode (int val, int *pval, int pos, struct btreeNode *node,
  struct btreeNode *child, struct btreeNode **newNode) {
    int median, j;

    if (pos > MIN)
        median = MIN + 1;
    else
        median = MIN;

    *newNode = (struct btreeNode *)malloc(sizeof(struct btreeNode));
    j = median + 1;
    while (j <= MAX) {
        (*newNode)->val[j - median] = node->val[j];
        (*newNode)->link[j - median] = node->link[j];
        j++;
    }
    node->count = median;
    (*newNode)->count = MAX - median;

    if (pos <= MIN) {
        addValToNode(val, pos, node, child);
    } else {
        addValToNode(val, pos - median, *newNode, child);
    }
    *pval = node->val[node->count];
    (*newNode)->link[0] = node->link[node->count];
    node->count--;
}

/* sets the value val in the node */
int setValueInNode(int val, int *pval,
  struct btreeNode *node, struct btreeNode **child) {

    int pos;
    if (!node) {
        *pval = val;
```

```
        *child = NULL;
        return 1;
    }

    if (val < node->val[1]) {
        pos = 0;
    } else {
        for (pos = node->count;
            (val < node->val[pos] && pos > 1); pos--);
        if (val == node->val[pos]) {
            printf("Duplicates not allowed\n");
            return 0;
        }
    }
    if (setValueInNode(val, pval, node->link[pos], child)) {
        if (node->count < MAX) {
            addValToNode(*pval, pos, node, *child);
        } else {
            splitNode(*pval, pval, pos, node, *child, child);
            return 1;
        }
    }
    return 0;
}

/* insert val in B-Tree */
void insertion(int val) {
    int flag, i;
    struct btreeNode *child;

    flag = setValueInNode(val, &i, root, &child);
    if (flag)
        root = createNode(i, child);
}

/* copy successor for the value to be deleted */
void copySuccessor(struct btreeNode *myNode, int pos) {
    struct btreeNode *dummy;
    dummy = myNode->link[pos];
```

```
    for (;dummy->link[0] != NULL;)
        dummy = dummy->link[0];
    myNode->val[pos] = dummy->val[1];

}

/* removes the value from the given node and rearrange values */
void removeVal(struct btreeNode *myNode, int pos) {
    int i = pos + 1;
    while (i <= myNode->count) {
        myNode->val[i - 1] = myNode->val[i];
        myNode->link[i - 1] = myNode->link[i];
        i++;
    }
    myNode->count--;
}

/* shifts value from parent to right child */
void doRightShift(struct btreeNode *myNode, int pos) {
    struct btreeNode *x = myNode->link[pos];
    int j = x->count;

    while (j > 0) {
        x->val[j + 1] = x->val[j];
        x->link[j + 1] = x->link[j];
    }
    x->val[1] = myNode->val[pos];
    x->link[1] = x->link[0];
    x->count++;

    x = myNode->link[pos - 1];
    myNode->val[pos] = x->val[x->count];
    myNode->link[pos] = x->link[x->count];
    x->count--;
    return;
}

/* shifts value from parent to left child */
void doLeftShift(struct btreeNode *myNode, int pos) {
    int j = 1;
```

```
    struct btreeNode *x = myNode->link[pos - 1];

    x->count++;
    x->val[x->count] = myNode->val[pos];
    x->link[x->count] = myNode->link[pos]->link[0];

    x = myNode->link[pos];
    myNode->val[pos] = x->val[1];
    x->link[0] = x->link[1];
    x->count--;

    while (j <= x->count) {
        x->val[j] = x->val[j + 1];
        x->link[j] = x->link[j + 1];
        j++;
    }
    return;
}

/* merge nodes */
void mergeNodes(struct btreeNode *myNode, int pos) {
    int j = 1;
    struct btreeNode *x1 = myNode->link[pos], *x2 = myNode->link[pos - 1];

    x2->count++;
    x2->val[x2->count] = myNode->val[pos];
    x2->link[x2->count] = myNode->link[0];

    while (j <= x1->count) {
        x2->count++;
        x2->val[x2->count] = x1->val[j];
        x2->link[x2->count] = x1->link[j];
        j++;
    }

    j = pos;
    while (j < myNode->count) {
        myNode->val[j] = myNode->val[j + 1];
        myNode->link[j] = myNode->link[j + 1];
        j++;
```

```
        }
        myNode->count--;
        free(x1);
}

/* adjusts the given node */
void adjustNode(struct btreeNode *myNode, int pos) {
        if (!pos) {
                if (myNode->link[1]->count > MIN) {
                        doLeftShift(myNode, 1);
                } else {
                        mergeNodes(myNode, 1);
                }
        } else {
                if (myNode->count != pos) {
                        if(myNode->link[pos - 1]->count > MIN) {
                                doRightShift(myNode, pos);
                        } else {
                                if (myNode->link[pos + 1]->count > MIN) {
                                        doLeftShift(myNode, pos + 1);
                                } else {
                                        mergeNodes(myNode, pos);
                                }
                        }
                } else {
                        if (myNode->link[pos - 1]->count > MIN)
                                doRightShift(myNode, pos);
                        else
                                mergeNodes(myNode, pos);
                }
        }
}

/* delete val from the node */
int delValFromNode(int val, struct btreeNode *myNode) {
        int pos, flag = 0;
        if (myNode) {
                if (val < myNode->val[1]) {
                        pos = 0;
                        flag = 0;
```

```
        } else {
            for (pos = myNode->count;
                (val < myNode->val[pos] && pos > 1); pos--);
             if (val == myNode->val[pos]) {
                flag = 1;
            } else {
                flag = 0;
            }
        }
        if (flag) {
            if (myNode->link[pos - 1]) {
                copySuccessor(myNode, pos);
                flag = delValFromNode(myNode->val[pos], myNode->link[pos]);
                if (flag == 0) {
                    printf("Given data is not present in B-Tree\n");
                }
            } else {
                removeVal(myNode, pos);
            }
        } else {
            flag = delValFromNode(val, myNode->link[pos]);
        }
        if (myNode->link[pos]) {
            if (myNode->link[pos]->count < MIN)
                adjustNode(myNode, pos);
        }
    }
    return flag;
}

/* delete val from B-tree */
void deletion(int val, struct btreeNode *myNode) {
    struct btreeNode *tmp;
    if (!delValFromNode(val, myNode)) {
        printf("Given value is not present in B-Tree\n");
        return;
    } else {
        if (myNode->count == 0) {
            tmp = myNode;
            myNode = myNode->link[0];
```

```
                free(tmp);
            }
        }
        root = myNode;
        return;
}

/* search val in B-Tree */
void searching(int val, int *pos, struct btreeNode *myNode) {
        if (!myNode) {
            return;
        }

        if (val < myNode->val[1]) {
            *pos = 0;
        } else {
            for (*pos = myNode->count;
                (val < myNode->val[*pos] && *pos > 1); (*pos)--);
            if (val == myNode->val[*pos]) {
                printf("Given data %d is present in B-Tree", val);
                return;
            }
        }
        searching(val, pos, myNode->link[*pos]);
        return;
}

/* B-Tree Traversal */
void traversal(struct btreeNode *myNode) {
        int i;
        if (myNode) {
            for (i = 0; i < myNode->count; i++) {
                traversal(myNode->link[i]);
                printf("%d ", myNode->val[i + 1]);
            }
            traversal(myNode->link[i]);
        }
}

int main() {
```

```
	int val, ch;
	while (1) {
		printf("1. Insertion\t2. Deletion\n");
		printf("3. Searching\t4. Traversal\n");
		printf("5. Exit\nEnter your choice:");
		scanf("%d", &ch);
		switch (ch) {
			case 1:
				printf("Enter your input:");
				scanf("%d", &val);
				insertion(val);
				break;
			case 2:
				printf("Enter the element to delete:");
				scanf("%d", &val);
				deletion(val, root);
				break;
			case 3:
				printf("Enter the element to search:");
				scanf("%d", &val);
				searching(val, &ch, root);
				break;
			case 4:
				traversal(root);
				break;
			case 5:
				exit(0);
			default:
				printf("U have entered wrong option!!\n");
				break;
		}
		printf("\n");
	}
}
```

KG REDDY
College of Engineering
& Technology
AN AUTONOMOUS INSTITUTION

OUTPUT:

```
1. Insertion 2. Deletion
3. Searching 4. Traversal
5. Exit
Enter your choice:1
Enter your input:70

1. Insertion 2. Deletion
3. Searching 4. Traversal
5. Exit
Enter your choice:1
Enter your input:17

1. Insertion 2. Deletion
3. Searching 4. Traversal
5. Exit
Enter your choice:1
Enter your input:67

1. Insertion 2. Deletion
3. Searching 4. Traversal
5. Exit
Enter your choice:1
Enter your input:89

1. Insertion 2. Deletion
3. Searching 4. Traversal
5. Exit
```

```
Enter your choice:4
17 67 70 89

1. Insertion 2. Deletion
3. Searching 4. Traversal
5. Exit
Enter your choice:3
Enter the element to search:70
Given data 70 is present in B-Tree

1. Insertion 2. Deletion
3. Searching 4. Traversal
5. Exit
Enter your choice:2
Enter the element to delete:17

1. Insertion 2. Deletion
3. Searching 4. Traversal
5. Exit
Enter your choice:4
67 70 89

1. Insertion 2. Deletion
3. Searching 4. Traversal
5. Exit
Enter your choice:5
```

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

```
int size = 0;

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

//function to heapify the tree
void heapify(int array[], int size, int i)
{
 if (size == 1){
   printf("Only one element in the heap");
 }
 else{
   int largest = i;

   // Finding the left and the right child
   int l = 2 * i + 1;
   int r = 2 * i + 2;

   //Setting the largest out of root, left child & right child
   if (l < size && array[l] > array[largest])
     largest = l;
   if (r < size && array[r] > array[largest])
     largest = r;

   // If index is not equal to largest, perform swap
   if (largest != i){
     swap(&array[i], &array[largest]);
```

```
    heapify(array, size, largest);
   }
 }
}

//function to insert elements in the heap
void insert(int array[], int newNum){
 //checking if it is the first element
 if (size == 0){
   array[0] = newNum;
   size += 1;
 }
 else{
   array[size] = newNum;
   size += 1;
   for (int i = size / 2 - 1; i >= 0; i--)
     //heapifying the tree
     heapify(array, size, i);
 }
}

//function to delete from heap
void deleteRoot(int array[], int num)
{
 int i;
 for (i = 0; i < size; i++)
   //search if element is present in the heap
   if (num == array[i])
     break;
  //swapping the last element with the root element
 swap(&array[i], &array[size - 1]);
 size -= 1;

 for (int i = size / 2 - 1; i >= 0; i--)
   //heapifying the tree once again
   heapify(array, size, i);
}
```

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

```
//C++ program for leftist heap / leftist tree

#include <bits/stdc++.h>

using namespace std;


// Node Class Declaration

class LeftistNode

{

public:

        int element;

        LeftistNode *left;

        LeftistNode *right;

        int dist;

        LeftistNode(int & element, LeftistNode *lt = NULL,

                                LeftistNode *rt = NULL, int np = 0)

        {

                this->element = element;

                right = rt;

                left = lt,

                dist = np;

        }
```

```
};


//Class Declaration
class LeftistHeap
{
public:
        LeftistHeap();
        LeftistHeap(LeftistHeap &rhs);
        ~LeftistHeap();
        bool isEmpty();
        bool isFull();
        int &findMin();
        void Insert(int &x);
        void deleteMin();
        void deleteMin(int &minItem);
        void makeEmpty();
        void Merge(LeftistHeap &rhs);
        LeftistHeap & operator =(LeftistHeap &rhs);
private:
        LeftistNode *root;
        LeftistNode *Merge(LeftistNode *h1,
                                LeftistNode *h2);
        LeftistNode *Merge1(LeftistNode *h1,
                                    LeftistNode *h2);
        void swapChildren(LeftistNode * t);
```

```
        void reclaimMemory(LeftistNode * t);
        LeftistNode *clone(LeftistNode *t);
};

// Construct the leftist heap
LeftistHeap::LeftistHeap()
{
        root = NULL;
}

// Copy constructor.
LeftistHeap::LeftistHeap(LeftistHeap &rhs)
{
        root = NULL;
        *this = rhs;
}

// Destruct the leftist heap
LeftistHeap::~LeftistHeap()
{
        makeEmpty( );
}

/* Merge rhs into the priority queue.
rhs becomes empty. rhs must be different
```

```
from this.*/
void LeftistHeap::Merge(LeftistHeap &rhs)
{
    if (this == &rhs)
        return;
    root = Merge(root, rhs.root);
    rhs.root = NULL;
}

/* Internal method to merge two roots.
Deals with deviant cases and calls recursive Merge1.*/
LeftistNode *LeftistHeap::Merge(LeftistNode * h1,
                                 LeftistNode * h2)
{
    if (h1 == NULL)
        return h2;
    if (h2 == NULL)
        return h1;
    if (h1->element < h2->element)
        return Merge1(h1, h2);
    else
        return Merge1(h2, h1);
}

/* Internal method to merge two roots.
```

Assumes trees are not empty, and h1's root contains smallest item.*/

```
LeftistNode *LeftistHeap::Merge1(LeftistNode * h1,
                                                   LeftistNode * h2)
{
        if (h1->left == NULL)
                h1->left = h2;
        else
        {
                h1->right = Merge(h1->right, h2);
                if (h1->left->dist < h1->right->dist)
                        swapChildren(h1);
                h1->dist = h1->right->dist + 1;
        }
        return h1;
}

// Swaps t's two children.
void LeftistHeap::swapChildren(LeftistNode * t)
{
        LeftistNode *tmp = t->left;
        t->left = t->right;
        t->right = tmp;
}
```

```
/* Insert item x into the priority queue, maintaining

heap order.*/

void LeftistHeap::Insert(int &x)

{

        root = Merge(new LeftistNode(x), root);

}


/* Find the smallest item in the priority queue.

Return the smallest item, or throw Underflow if empty.*/

int &LeftistHeap::findMin()

{

        return root->element;

}


/* Remove the smallest item from the priority queue.

Throws Underflow if empty.*/

void LeftistHeap::deleteMin()

{

        LeftistNode *oldRoot = root;

        root = Merge(root->left, root->right);

        delete oldRoot;

}


/* Remove the smallest item from the priority queue.

Pass back the smallest item, or throw Underflow if empty.*/
```

```
void LeftistHeap::deleteMin(int &minItem)
{
    if (isEmpty())
    {
        cout<<"Heap is Empty"<<endl;
        return;
    }
    minItem = findMin();
    deleteMin();
}

/* Test if the priority queue is logically empty.
Returns true if empty, false otherwise*/
bool LeftistHeap::isEmpty()
{
    return root == NULL;
}

/* Test if the priority queue is logically full.
Returns false in this implementation.*/
bool LeftistHeap::isFull()
{
    return false;
}
```

```
// Make the priority queue logically empty
void LeftistHeap::makeEmpty()
{
    reclaimMemory(root);
    root = NULL;
}

// Deep copy
LeftistHeap &LeftistHeap::operator =(LeftistHeap & rhs)
{
    if (this != &rhs)
    {
        makeEmpty();
        root = clone(rhs.root);
    }
    return *this;
}

// Internal method to make the tree empty.
void LeftistHeap::reclaimMemory(LeftistNode * t)
{
    if (t != NULL)
    {
        reclaimMemory(t->left);
        reclaimMemory(t->right);
```

```
                delete t;
        }
}

// Internal method to clone subtree.
LeftistNode *LeftistHeap::clone(LeftistNode * t)
{
        if (t == NULL)
                return NULL;
        else
                return new LeftistNode(t->element, clone(t->left),
                                                        clone(t->right), t->dist);
}

//Driver program
int main()
{
        LeftistHeap h;
        LeftistHeap h1;
        LeftistHeap h2;
        int x;
        int arr[]= {1, 5, 7, 10, 15};
        int arr1[]= {22, 75};

        h.Insert(arr[0]);
```

```
h.Insert(arr[1]);
h.Insert(arr[2]);
h.Insert(arr[3]);
h.Insert(arr[4]);
h1.Insert(arr1[0]);
h1.Insert(arr1[1]);

h.deleteMin(x);
cout<< x <<endl;

h1.deleteMin(x);
cout<< x <<endl;

h.Merge(h1);
h2 = h;

h2.deleteMin(x);
cout<< x << endl;

return 0;
}
```

Output

1

22

5

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

```
#include
<stdio.h>
        #include <stdlib.h>
        #define MAX_BUFFER_SIZE 127
        typedef struct __node_t___{
                int value;
                int degree;
                struct __node_t___*child;
                struct __node_t___*parent;
                struct __node_t___*brother;
        } node_t;
        typedef struct __heap_t___{
                node_t *head;
        } heap_t;
        node_t *node_init( int value )
        {
                node_t *node = malloc( sizeof ( node_t ) );
                node->value = value;
                node->degree = 0;
                return node;
        }
        heap_t *heap_init()
        {
                heap_t *heap = malloc( sizeof ( heap_t ) );
                heap->head = NULL;
                return heap;
        }
        node_t *heap_merge( heap_t *heap1, heap_t *heap2 )
        {
                if( heap1->head == NULL ) return heap2->head;
                if( heap2->head == NULL ) return heap1->head;
```

```
	node_t *head;
	node_t *h1It = heap1->head;
	node_t *h2It = heap2->head;
	node_t *tail;
	if( heap1->head->degree <= heap2->head->degree )
	{
		head = heap1->head;
		h1It = h1It->brother;
	}
	else
	{
		head = heap2->head;
		h2It = h2It->brother;
	}
	tail = head;
	while( h1It != NULL && h2It != NULL )
	{
		if( h1It->degree <= h2It->degree )
		{
			tail->brother = h1It;
			h1It = h1It->brother;
		}
		else
		{
			tail->brother = h2It;
			h2It = h2It->brother;
		}
		tail = tail->brother;
	}
	tail->brother = ( h1It != NULL ) ? h1It : h2It;
	return head;
}
node_t *heap_union( heap_t *original, heap_t *uni )
{
	node_t *new_head = heap_merge( original, uni );
	original->head = NULL;
	uni->head = NULL;
	if( new_head == NULL )
		return NULL;
	node_t *prev = NULL;
```

```
	node_t *aux = new_head;
	node_t *next = new_head->brother;
	while( next != NULL )
	{
		if( aux->degree != next->degree || ( next->brother != NULL &&
next->brother->degree == aux->degree ) )
		{
			prev = aux;
			aux = next;
		}
		else
		{
			if( aux->value < next->value )
			{
				aux->brother = next->brother;
				next->parent = aux;
				next->brother = aux->child;
				aux->child = next;
				aux->degree++;
			}
			else
			{
				if( prev == NULL )
					new_head = next;
				else
					prev->brother = next;
				aux->parent = next;
				aux->brother = next->child;
				next->child = aux;
				next->degree++;
				aux = next;
			}
		}
		next = aux->brother;
	}
	return new_head;
}
void heap_insert( heap_t *heap, int value )
{
	node_t *node = node_init( value );
```

```
        heap_t *temp = heap_init();
        temp->head = node;
        heap->head = heap_union( heap, temp );
        free( temp );
}
void heap_remove( heap_t *heap, node_t *head, node_t *bef )
{
        if( head == heap->head )
                heap->head = head->brother;
        else
                bef->brother = head->brother;
        node_t *new_head = NULL;
        node_t *child = head->child;
        while( child != NULL )
        {
                node_t *next = child->brother;
                child->brother = new_head;
                child->parent = NULL;
                new_head = child;
                child = next;
        }
        heap_t *temp = heap_init();
        temp->head = new_head;
        heap->head = heap_union( heap, temp );
        free( temp );
}
node_t *heap_min( heap_t *heap )
{
        if( heap->head == NULL ) return NULL;
        node_t *min = heap->head;
        node_t *min_prev = NULL;
        node_t *next = min->brother;
        node_t *next_prev = min;
        while( next != NULL )
        {
                if( next->value < min->value )
                {
                        min = next;
                        min_prev = next_prev;
                }
```

```
            next_prev = next;
            next = next->brother;
        }
        heap_remove( heap, min, min_prev );
        return min;
}
void heap_free( heap_t *h )
{
        while( heap_min( h ) );
        free( h );
}
int
main()
{
        heap_t *heap = heap_init();
        char buf[MAX_BUFFER_SIZE];
        while( fgets( buf, MAX_BUFFER_SIZE, stdin ) != NULL )
        {
            char op;
            int value;
            sscanf( buf, "%c %d", &op, &value );
            switch( op )
            {
                case 'i': // insert
                    heap_insert( heap, value );
                break;
                case 'a': // remove
                {
                    node_t *min = heap_min( heap );
                    if( min != NULL )
                        printf( "%d\n", min->value );
                }
                break;
                case 'r': // ?
                break;
            }
        }
        heap_free( heap );
        return 0;
}
```

OUTPUT:

```
/**
    * Binomial heap data structure
    * Test case:
    * i 10
    * i 20
    * i 30
    * a
    * i 5
    * a
    * i 3
    * i 1
    * a
    * r
    * a
    ********************************
    * Expected output:
    * 10
    * 5
    * 1
    * 3
    **/
```

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.

```
#include  <stdio.h>
#include <stdlib.h>
struct Node {
  int data;
  struct Node *leftChild;
  struct Node *rightChild;
  int height;
};
int max(int a, int b);
int height(struct Node *N){
  if (N == NULL)
    return 0;
  return N->height;
}
int max(int a, int b){
  return (a > b) ? a : b;
}
struct Node *newNode(int data){
  struct Node *node = (struct Node *) malloc(sizeof(struct Node));
  node->data = data;
  node->leftChild = NULL;
  node->rightChild = NULL;
  node->height = 1;
  return (node);
}
struct Node *rightRotate(struct Node *y){
  struct Node *x = y->leftChild;
  struct Node *T2 = x->rightChild;
  x->rightChild = y;
  y->leftChild = T2;
  y->height = max(height(y->leftChild), height(y->rightChild)) + 1;
  x->height = max(height(x->leftChild), height(x->rightChild)) + 1;
  return x;
```

```
}
struct Node *leftRotate(struct Node *x){
  struct Node *y = x->rightChild;
  struct Node *T2 = y->leftChild;
  y->leftChild = x;
  x->rightChild = T2;
  x->height = max(height(x->leftChild), height(x->rightChild)) + 1;
  y->height = max(height(y->leftChild), height(y->rightChild)) + 1;
  return y;
}
int getBalance(struct Node *N){
  if (N == NULL)
    return 0;
  return height(N->leftChild) - height(N->rightChild);
}
struct Node *insertNode(struct Node *node, int data){
  if (node == NULL)
    return (newNode(data));
  if (data < node->data)
    node->leftChild = insertNode(node->leftChild, data);
  else if (data > node->data)
    node->rightChild = insertNode(node->rightChild, data);
  else
    return node;
  node->height = 1 + max(height(node->leftChild),
              height(node->rightChild));
  int balance = getBalance(node);
  if (balance > 1 && data < node->leftChild->data)
    return rightRotate(node);
  if (balance < -1 && data > node->rightChild->data)
    return leftRotate(node);
  if (balance > 1 && data > node->leftChild->data) {
    node->leftChild = leftRotate(node->leftChild);
    return rightRotate(node);
  }
  if (balance < -1 && data < node->rightChild->data) {
    node->rightChild = rightRotate(node->rightChild);
    return leftRotate(node);
  }
  return node;
```

```
}
struct Node *minValueNode(struct Node *node){
  struct Node *current = node;
  while (current->leftChild != NULL)
    current = current->leftChild;
  return current;
}
void printTree(struct Node *root){
  if (root == NULL)
    return;
  if (root != NULL) {
    printTree(root->leftChild);
    printf("%d ", root->data);
    printTree(root->rightChild);
  }
}
int main(){
  struct Node *root = NULL;
  root = insertNode(root, 22);
  root = insertNode(root, 14);
  root = insertNode(root, 72);
  root = insertNode(root, 44);
  root = insertNode(root, 25);
  root = insertNode(root, 63);
  root = insertNode(root, 98);
  printf("AVL Tree: ");
  printTree(root);
  return 0;
       }
```

Output

AVL Tree: 14 22 25 44 63 72 98

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.

```
/** C implementation for
        Red-Black Tree Insertion
        This code is provided by
        costheta_z **/
#include  <stdio.h>
#include <stdlib.h>

// Structure to represent each
// node in a red-black tree
struct node {
        int d; // data
        int c; // 1-red, 0-black
        struct node* p; // parent
        struct node* r; // right-child
        struct node* l; // left child
};

// global root for the entire tree
struct node* root = NULL;

// function to perform BST insertion of a node
struct node* bst(struct node* trav,
                                        struct node* temp)
{
        // If the tree is empty,
        // return a new node
        if (trav == NULL)
                return temp;

        // Otherwise recur down the tree
        if (temp->d < trav->d)
        {
                trav->l = bst(trav->l, temp);
```

```
		trav->l->p = trav;
	}
	else if (temp->d > trav->d)
	{
		trav->r = bst(trav->r, temp);
		trav->r->p = trav;
	}

	// Return the (unchanged) node pointer
	return trav;
}

// Function performing right rotation
// of the passed node
void rightrotate(struct node* temp)
{
	struct node* left = temp->l;
	temp->l = left->r;
	if (temp->l)
		temp->l->p = temp;
	left->p = temp->p;
	if (!temp->p)
		root = left;
	else if (temp == temp->p->l)
		temp->p->l = left;
	else
		temp->p->r = left;
	left->r = temp;
	temp->p = left;
}

// Function performing left rotation
// of the passed node
void leftrotate(struct node* temp)
{
	struct node* right = temp->r;
	temp->r = right->l;
	if (temp->r)
		temp->r->p = temp;
	right->p = temp->p;
```

```
        if (!temp->p)
                root = right;
        else if (temp == temp->p->l)
                temp->p->l = right;
        else
                temp->p->r = right;
        right->l = temp;
        temp->p = right;
}

// This function fixes violations
// caused by BST insertion
void fixup(struct node* root, struct node* pt)
{
        struct node* parent_pt = NULL;
        struct node* grand_parent_pt = NULL;

        while ((pt != root) && (pt->c != 0)
                && (pt->p->c == 1))
        {
                parent_pt = pt->p;
                grand_parent_pt = pt->p->p;

                /* Case : A
                        Parent of pt is left child
                        of Grand-parent of
                pt */
                if (parent_pt == grand_parent_pt->l)
                {

                        struct node* uncle_pt = grand_parent_pt->r;

                        /* Case : 1
                                The uncle of pt is also red
                                Only Recoloring required */
                        if (uncle_pt != NULL && uncle_pt->c == 1)
                        {
                                grand_parent_pt->c = 1;
                                parent_pt->c = 0;
                                uncle_pt->c = 0;
```

```
                pt = grand_parent_pt;
            }

            else {

                /* Case : 2
                    pt is right child of its parent
                    Left-rotation required */
                if (pt == parent_pt->r) {
                    leftrotate(parent_pt);
                    pt = parent_pt;
                    parent_pt = pt->p;
                }

                /* Case : 3
                    pt is left child of its parent
                    Right-rotation required */
                rightrotate(grand_parent_pt);
                int t = parent_pt->c;
                parent_pt->c = grand_parent_pt->c;
                grand_parent_pt->c = t;
                pt = parent_pt;
            }
        }

        /* Case : B
            Parent of pt is right
            child of Grand-parent of
        pt */
        else {
            struct node* uncle_pt = grand_parent_pt->l;

            /* Case : 1
                The uncle of pt is also red
                Only Recoloring required */
            if ((uncle_pt != NULL) && (uncle_pt->c == 1))
            {
                grand_parent_pt->c = 1;
                parent_pt->c = 0;
                uncle_pt->c = 0;
```

```
                        pt = grand_parent_pt;
                    }
                    else {
                        /* Case : 2
                        pt is left child of its parent
                        Right-rotation  required  */
                        if (pt == parent_pt->l) {
                            rightrotate(parent_pt);
                            pt = parent_pt;
                            parent_pt = pt->p;
                        }

                        /* Case : 3
                            pt is right child of its parent
                            Left-rotation required */
                        leftrotate(grand_parent_pt);
                        int t = parent_pt->c;
                        parent_pt->c = grand_parent_pt->c;
                        grand_parent_pt->c = t;
                        pt = parent_pt;
                    }
                }
            }
}

// Function to print inorder traversal
// of the fixated tree
void inorder(struct node* trav)
{
    if (trav == NULL)
        return;
    inorder(trav->l);
    printf("%d ", trav->d);
    inorder(trav->r);
}

// driver code
int main()
{
    int n = 7;
```

```
        int a[7] = { 7, 6, 5, 4, 3, 2, 1 };

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

            // allocating memory to the node and initializing:
            // 1. color as red
            // 2. parent, left and right pointers as NULL
            // 3. data as i-th value in the array
            struct node* temp
                = (struct node*)malloc(sizeof(struct node));
            temp->r = NULL;
            temp->l = NULL;
            temp->p = NULL;
            temp->d = a[i];
            temp->c = 1;

            // calling function that performs bst insertion of
            // this newly created node
            root = bst(root, temp);

            // calling function to preserve properties of rb
            // tree
            fixup(root, temp);
            root->c = 0;
        }

        printf("Inorder Traversal of Created Tree\n");
        inorder(root);

        return 0;
    }
```

Output

Inorder Traversal of Created Tree

1 2 3 4 5 6 7

Level Order Traversal of Created Tree

6 4 7 2 5 1 3

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

```
#include<stdio.h>

#define size 7

int arr[size];


void init()
{
    int i;
    for(i = 0; i < size; i++)
        arr[i] = -1;
}

void insert(int value)
{
    int key = value % size;

    if(arr[key] == -1)
    {
        arr[key] = value;
        printf("%d inserted at arr[%d]\n", value,key);
    }
    else
    {
        printf("Collision : arr[%d] has element %d already!\n",key,arr[key]);
        printf("Unable to insert %d\n",value);
    }
}

void del(int value)
{
    int key = value % size;
    if(arr[key] == value)
        arr[key] = -1;
    else
        printf("%d not present in the hash table\n",value);
}
```

```
void search(int value)
{
   int key = value % size;
   if(arr[key] == value)
      printf("Search Found\n");
   else
      printf("Search Not Found\n");
}

void print()
{
   int i;
   for(i = 0; i < size; i++)
      printf("arr[%d] = %d\n",i,arr[i]);
}

int main()
{
   init();
   insert(10); //key = 10 % 7 ==> 3
   insert(4);  //key = 4 % 7  ==> 4
   insert(2);  //key = 2 % 7  ==> 2
   insert(3);  //key = 3 % 7  ==> 3 (collision)

   printf("Hash table\n");
   print();
   printf("\n");

   printf("Deleting value 10..\n");
   del(10);
   printf("After the deletion hash table\n");
   print();
   printf("\n");

   printf("Deleting value 5..\n");
   del(5);
   printf("After the deletion hash table\n");
   print();
   printf("\n");
```

```
    printf("Searching value 4..\n");
    search(4);
    printf("Searching value 10..\n");
    search(10);

    return 0;
}
```

OUTPUT:

10 inserted at arr[3]
4 inserted at arr[4]
2 inserted at arr[2]
Collision : arr[3] has element 10 already!
Unable to insert 3
Hash table
arr[0] = -1
arr[1] = -1
arr[2] = 2
arr[3] = 10
arr[4] = 4
arr[5] = -1
arr[6] = -1

Deleting value 10..
After the deletion hash table
arr[0] = -1
arr[1] = -1
arr[2] = 2
arr[3] = -1
arr[4] = 4
arr[5] = -1
arr[6] = -1

Deleting value 5..
5 not present in the hash table
After the deletion hash table
arr[0] = -1
arr[1] = -1
arr[2] = 2

arr[3] = -1
arr[4] = 4
arr[5] = 10 inserted at arr[3]
4 inserted at arr[4]
2 inserted at arr[2]
Collision : arr[3] has element 10 already!
Unable to insert 3
Hash table
arr[0] = -1
arr[1] = -1
arr[2] = 2
arr[3] = 10
arr[4] = 4
arr[5] = -1
arr[6] = -1

Deleting value 10..
After the deletion hash table
arr[0] = -1
arr[1] = -1
arr[2] = 2
arr[3] = -1
arr[4] = 4
arr[5] = -1
arr[6] = -1

Deleting value 5..
5 not present in the hash table
After the deletion hash table
arr[0] = -1
arr[1] = -1
arr[2] = 2
arr[3] = -1
arr[4] = 4
arr[5] = 10 inserted at arr[3]
4 inserted at arr[4]
2 inserted at arr[2]
Collision : arr[3] has element 10 already!
Unable to insert 3
Hash table

arr[0] = -1
arr[1] = -1
arr[2] = 2
arr[3] = 10
arr[4] = 4
arr[5] = -1
arr[6] = -1

Deleting value 10..
After the deletion hash table
arr[0] = -1
arr[1] = -1
arr[2] = 2
arr[3] = -1
arr[4] = 4
arr[5] = -1
arr[6] = -1

Deleting value 5..
5 not present in the hash table
After the deletion hash table
arr[0] = -1
arr[1] = -1
arr[2] = 2
arr[3] = -1
arr[4] = 4
arr[5] = -1
arr[6] = -1

Searching value 4..
Search Found
Searching value 10..
Search Not Found

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

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

// Function to implement the KMP algorithm
void KMP(const char* text, const char* pattern, int m, int n)
{
    // base case 1: pattern is NULL or empty
    if (*pattern == '\0' || n == 0) {
        printf("The pattern occurs with shift 0");
    }

    // base case 2: text is NULL, or text's length is less than that of pattern's
    if (*text == '\0' || n > m) {
        printf("Pattern not found");
    }

    // next[i] stores the index of the next best partial match
    int next[n + 1];

    for (int i = 0; i < n + 1; i++) {
        next[i] = 0;
    }

    for (int i = 1; i < n; i++)
    {
        int j = next[i];

        while (j > 0 && pattern[j] != pattern[i]) {
            j = next[j];
        }

        if (j > 0 || pattern[j] == pattern[i]) {
            next[i + 1] = j + 1;
        }
    }

    for (int i = 0, j = 0; i < m; i++)
    {
        if (*(text + i) == *(pattern + j))
```

```
        {
            if (++j == n) {
                printf("The pattern occurs with shift %d\n", i - j + 1);
            }
        }
        else if (j > 0)
        {
            j = next[j];
            i--;    // since `i` will be incremented in the next iteration
        }
    }
}

// Program to implement the KMP algorithm in C
int main(void)
{
    char* text = "ABCABAABCABAC";
    char* pattern = "CAB";

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

    KMP(text, pattern, n, m);

    return 0;
}
```

Output:

The pattern occurs with shift 2
The pattern occurs with shift 8

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

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main()
{
 int i,j,k,n,m,flag=0;
 char t[40],p[30];
 clrscr();
 printf("Enter text: ");
 gets(t);
 printf("\nEnter pattern: ");
 gets(p);
 n=strlen(t);
 m=strlen(p);
 for(i=0;i<=n-m;i++)
 {
  j=0;
  while(j<m && p[j]==t[j+i])
   {
     j++;
     if(j==m)
     {
      flag=1;
      k=i+1;
     }
     else
      flag=0;
   }
 }
 if(flag==1)
   printf("\nPattern found at position: %d\n ",k);
 else
   printf("\nPattern not found in text \n");
  getch();
}
```

Output:

```
Enter text: mca programme

Enter pattern: programme

Pattern found at position: 5
```

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

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

# define NO_OF_CHARS 256

// A utility function to get maximum of two integers
int max(int a, int b) {
return (a > b) ? a : b;}

// The preprocessing function for Boyer Moore's bad character heuristic
void badCharHeuristic(char *str, int size, int badchar[NO_OF_CHARS]){
int i

// Initialize all occurrences as -1
for (i = 0; i < NO_OF_CHARS; i++)
badchar[i] = -1
// Fill the actual value of last occurrence of a character
for (i = 0; i < size; i++)
badchar[(int) str[i]] = i;
}

void search(char *txt, char *pat) {
int m = strlen(pat);
int n = strlen(txt);

int badchar[NO_OF_CHARS];

badCharHeuristic(pat, m, badchar);

int s = 0; // s is shift of the pattern with respect to text
while (s <= (n - m)) {
int j = m - 1;

while (j >= 0 && pat[j] == txt[s + j])
j--;

if (j < 0) {
```

```
            printf("\n pattern occurs at shift = %d", s);

            s += (s + m < n) ? m - badchar[txt[s + m]] : 1;

        }

        else
            s += max(1, j - badchar[txt[s + j]]);
    }
}

int main() {
    char txt[] = "ABAAABCD";
    char pat[] = "ABC";
    search(txt, pat);
return 0;
}
```

Output:

pattern occurs at shift = 4