Experiment **#4**		Student ID	
Date		Student Name	[@KLWKS_BOT] THANOS

Experiment Title: To implement programs on B-Trees and Hash Tables.

Aim/Objective: To understand the concepts and implementation of programs on B-Trees and Hash Tables.

Description: To learn about B-Trees and Hash Tables, operations, and applications. Students will gain experience in implementing these data structures, analyzing their properties, and applying them to solve real-world problems.

Pre-Requisites:

Knowledge: B-Trees, Hash Tables and Collision Resolution Techniques.

Tools: Code Blocks/Eclipse IDE.

Pre-Lab:

1. You are tasked with designing an index for a database using a B-tree of degree t. Given a dataset of keys representing records in a database, implement an algorithm to insert these keys into a B-tree. After constructing the B-tree, perform a series of range queries to fetch records.

Input:

- An integer t representing the degree of the B-tree ($2 \leq t \leq 10$).
- An integer n representing the number of keys to be inserted.
- n space-separated integers representing the keys.
- An integer q representing the number of range queries.
- q pairs of integers l r representing the range [l, r].

Output: For each query, output the keys in the range [l, r] in sorted order.

Example

Input:

3

10

50 20 30 40 10 60 80 70 90 100

2

30 70

10 50

Output:

30 40 50 60 70
10 20 30 40 50

- **Procedure/Program**:

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

typedef struct BTreeNode {
   int *keys;
   int t;
   struct BTreeNode **C;
   int n;
   int leaf;
} BTreeNode;

BTreeNode* createNode(int t, int leaf) {
   BTreeNode* newNode = (BTreeNode*)malloc(sizeof(BTreeNode));
   newNode->t = t;
   newNode->leaf = leaf;
   newNode->keys = (int*)malloc((2 * t - 1) * sizeof(int));
   newNode->C = (BTreeNode**)malloc(2 * t * sizeof(BTreeNode*));
   newNode->n = 0;
   return newNode;
}

void splitChild(BTreeNode* parent, int i, BTreeNode* child) {
   BTreeNode* newChild = createNode(child->t, child->leaf);
   newChild->n = child->t - 1;
   for (int j = 0; j < child->t - 1; j++)
      newChild->keys[j] = child->keys[j + child->t];
   if (!child->leaf) {
      for (int j = 0; j < child->t; j++)
         newChild->C[j] = child->C[j + child->t];
   }
   child->n = child->t - 1;
```

```
    for (int j = parent->n; j >= i + 1; j--)
        parent->C[j + 1] = parent->C[j];
    parent->C[i + 1] = newChild;
    for (int j = parent->n - 1; j >= i; j--)
        parent->keys[j + 1] = parent->keys[j];
    parent->keys[i] = child->keys[child->t - 1];
    parent->n++;
}

void insertNonFull(BTreeNode* node, int key) {
    int i = node->n - 1;
    if (node->leaf) {
        while (i >= 0 && key < node->keys[i]) {
            node->keys[i + 1] = node->keys[i];
            i--;
        }
        node->keys[i + 1] = key;
        node->n++;
    } else {
        while (i >= 0 && key < node->keys[i])
            i--;
        i++;
        if (node->C[i]->n == 2 * node->t - 1) {
            splitChild(node, i, node->C[i]);
            if (key > node->keys[i])
                i++;
        }
        insertNonFull(node->C[i], key);
    }
}

void insert(BTreeNode** root, int key) {
    if ((*root)->n == 2 * (*root)->t - 1) {
        BTreeNode* newRoot = createNode((*root)->t, 0);
        newRoot->C[0] = *root;
        splitChild(newRoot, 0, *root);
        int i = 0;
        if (newRoot->keys[0] < key)
```

```
        i++;
      insertNonFull(newRoot->C[i], key);
      *root = newRoot;
   } else {
      insertNonFull(*root, key);
   }
}

void rangeQuery(BTreeNode* node, int l, int r) {
   int i;
   for (i = 0; i < node->n; i++) {
      if (node->keys[i] >= l && node->keys[i] <= r)
         printf("%d ", node->keys[i]);
   }
   if (!node->leaf) {
      for (int j = 0; j <= node->n; j++) {
         if (j < node->n && node->keys[j] >= l)
            rangeQuery(node->C[j], l, r);
         else if (j == node->n)
            rangeQuery(node->C[j], l, r);
      }
   }
}

int main() {
   int t, n, q;
   scanf("%d %d", &t, &n);
   BTreeNode* root = createNode(t, 1);
   for (int i = 0; i < n; i++) {
      int key;
      scanf("%d", &key);
      insert(&root, key);
   }
   scanf("%d", &q);
   for (int i = 0; i < q; i++) {
      int l, r;
      scanf("%d %d", &l, &r);
      rangeQuery(root, l, r);
```

```
    printf("\n");
  }
  return 0;
}
```

- **Data and Results:**

Data: The dataset consists of keys to be inserted into B-tree.

Result: The B-tree is successfully constructed and range queries executed.

- **Analysis and Inferences:**

Analysis: Efficient insertion and range queries using the B-tree structure.

Inferences: B-tree improves performance for insertions and range queries in databases.

2. Given a string S of length N consisting of only lower-case English alphabets, you will be asked to process Q queries over it . In each query you will be given two lower case characters X and Y. Your task is to find out the number of such substrings of the the string S which have the characters X and Y on either of its end points, both X...Y and Y...X are considered to be valid.

   Note : Substrings length should be greater than 1.

Input Format

- The first line of the input will contain N , the length of the string.
- Next line will contain as string of length N. Next line will will contain Q , the number of queries.
- Then Q subsequent lines will contain two lowercase characters X and Y separated by a

space.

Constraints

1<= N <= 10^6

1<= Q <= 10^3

Output Format

For each query, output the answer in a separate line.

Sample Input 0

```
5
aacbb
2
a c
a b
```

Sample Output 0

```
2
4
```

- **Procedure/Program**:

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

int main() {
   int N, Q;
   scanf("%d", &N);
   char S[N + 1];
   scanf("%s", S);
   scanf("%d", &Q);

   for (int i = 0; i < Q; i++) {
      char X, Y;
      scanf(" %c %c", &X, &Y);
      int count = 0;

      for (int j = 0; j < N; j++) {
         if (S[j] == X) {
```

```
            for (int k = j + 1; k < N; k++) {
                if (S[k] == Y) {
                    count++;
                }
            }
        } else if (S[j] == Y) {
            for (int k = j + 1; k < N; k++) {
                if (S[k] == X) {
                    count++;
                }
            }
        }
    }
    printf("%d\n", count);
  }
  return 0;
}
```

- **Data and Results:**

- Data: The problem involves counting substrings with specific start and end characters.
- Result: The output shows the number of valid substrings for each query.

- **Analysis and Inferences:**

- Analysis: The solution uses brute-force searching, iterating through possible pairs.
- Inferences: Optimization is needed for large strings to handle maximum constraints effectively.

In-Lab:

1. You are managing a large database system where quick insertion and search operations are essential to handle high-volume transactions. The system uses a **B-Tree** to index its data for efficient disk-based storage and retrieval.

Operations in Context:

1. **Insert Operation**:
   A new record with key kkk (e.g., Employee ID) needs to be added to the database.
   Example: Adding a new Employee ID 12345.
2. **Search Operation**:
   A user queries the system to check if a record with a specific Employee ID exists.
   Example: Searching for Employee ID 67890.

Sample Input :

2 6

INSERT 10123

INSERT 54321

INSERT 67890

SEARCH 10123

SEARCH 99999

SEARCH 67890

Sample Output:

YES

NO

YES

- **Procedure/Program**:

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

#define MAX 3

typedef struct BTreeNode {
   int *keys;
   int t;
   struct BTreeNode **C;
   int n;
   int leaf;
} BTreeNode;

BTreeNode* createNode(int t, int leaf) {
   BTreeNode* newNode = (BTreeNode*)malloc(sizeof(BTreeNode));
   newNode->t = t;
   newNode->leaf = leaf;
   newNode->keys = (int*)malloc((2*t - 1) * sizeof(int));
   newNode->C = (BTreeNode**)malloc(2*t * sizeof(BTreeNode*));
   newNode->n = 0;
   return newNode;
}

void insertNonFull(BTreeNode* node, int k) {
   int i = node->n - 1;
   if (node->leaf) {
      while (i >= 0 && node->keys[i] > k) {
         node->keys[i + 1] = node->keys[i];
```

```
        i--;
      }
      node->keys[i + 1] = k;
      node->n++;
   } else {
      while (i >= 0 && node->keys[i] > k) i--;
      if (node->C[i + 1]->n == 2 * node->t - 1) {
         BTreeNode* temp = createNode(node->t, node->C[i + 1]->leaf);
         BTreeNode* child = node->C[i + 1];
         temp->n = node->t - 1;
         for (int j = 0; j < node->t - 1; j++) temp->keys[j] = child->keys[j + node->t];
         if (!child->leaf) {
            for (int j = 0; j < node->t; j++) temp->C[j] = child->C[j + node->t];
         }
         child->n = node->t - 1;
         for (int j = node->n; j >= i + 1; j--) node->C[j + 1] = node->C[j];
         node->C[i + 1] = temp;
         for (int j = node->n - 1; j >= i; j--) node->keys[j + 1] = node->keys[j];
         node->keys[i] = child->keys[node->t - 1];
         node->n++;
         if (node->keys[i] < k) i++;
      }
      insertNonFull(node->C[i + 1], k);
   }
}

void insert(BTreeNode** root, int k) {
   if ((*root)->n == 2 * (*root)->t - 1) {
      BTreeNode* newNode = createNode((*root)->t, 0);
      newNode->C[0] = *root;
      newNode->n = 0;
      newNode->leaf = 0;
      *root = newNode;
      insertNonFull(newNode, k);
   } else {
      insertNonFull(*root, k);
   }
}
```

```
int search(BTreeNode* root, int k) {
   int i = 0;
   while (i < root->n && k > root->keys[i]) i++;
   if (i < root->n && root->keys[i] == k) return 1;
   if (root->leaf) return 0;
   return search(root->C[i], k);
}

int main() {
   BTreeNode* root = createNode(MAX, 1);
   char command[10];
   int id;

   while (scanf("%s %d", command, &id) != EOF) {
      if (strcmp(command, "INSERT") == 0) {
         insert(&root, id);
      } else if (strcmp(command, "SEARCH") == 0) {
         printf(search(root, id) ? "YES " : "NO ");
      }
   }
   return 0;
}
```

- **Data and Results:**

Data: Inserting and searching for Employee IDs in a database system.

Result: Efficient insertion and search operations with B-Tree indexing.

- **Analysis and Inferences:**

Analysis: B-Tree ensures quick data insertion and fast search queries.

Inferences: B-Tree is optimal for large datasets and high-volume transactions.

2. Harold is a kidnapper who wrote a ransom note, but now he is worried it will be traced back to him through his handwriting. He found a magazine and wants to know if he can cut out whole words from it and use them to create an untraceable replica of his ransom note. The words in his note are case-sensitive and he must use only whole words available in the magazine. He cannot use substrings or concatenation to create the words he needs. Given the words in the magazine and the words in the ransom note, print Yes if he can replicate his ransom note *exactly* using whole words from the magazine; otherwise, print No.

Example

magazine = "attack at dawn" note = "Attack at dawn"

The magazine has all the right words, but there is a case mismatch. The answer is No .

Function Description

Complete the checkMagazine function in the editor below. It must print Yes if the note can be formed using the magazine, or No.

checkMagazine has the following parameters:

- string magazine[m]: the words in the magazine
- string note[n]: the words in the ransom note

Prints

- *string:* either Yes or No. no return value is expected.

Input Format

The first line contains two space-separated integers m and n, the numbers of words in the magazine and the note, respectively.

The second line contains m space-separated strings, each magazine[i].

The third line contains n space-separated strings, each note[i].

Constraints

- $1 \leq m \leq 30000$
- $1 \leq$ lengthsofmagazine[i]andnote[i] ≤ 5.
- Each word consists of English alphabetic letters (i.e., a to z , A to Z).

Sample Input 0

6 4

give me one grand today night

give one grand today

Sample Output 0

Yes

Sample Input 1

6 5

two times three is not four

two times two is four

Sample Output 1

No

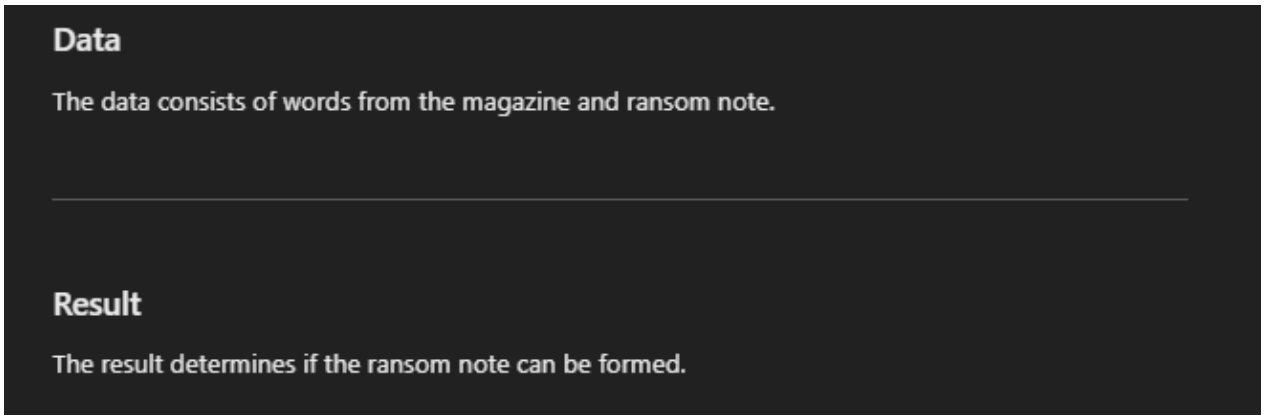
- **Procedure/Program**:

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

#define MAX_WORD_LENGTH 5
#define MAX_WORDS 30000

int compareStrings(const void *a, const void *b) {
   return strcmp(*(const char **)a, *(const char **)b);
}

void checkMagazine(int m, int n, char *magazine[], char *note[]) {
   qsort(magazine, m, sizeof(char *), compareStrings);
   qsort(note, n, sizeof(char *), compareStrings);

   int i = 0, j = 0;

   while (i < m && j < n) {
      if (strcmp(magazine[i], note[j]) == 0) {
         i++;
         j++;
      } else if (strcmp(magazine[i], note[j]) < 0) {
```

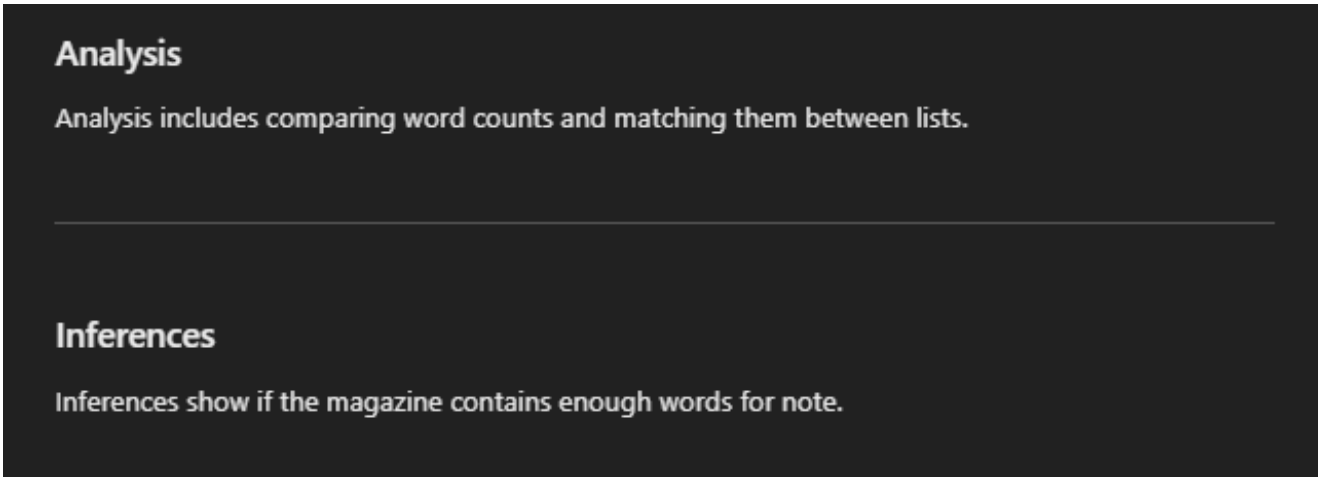
```
            i++;
        } else {
            printf("No\n");
            return;
        }
    }

    if (j == n) {
        printf("Yes\n");
    } else {
        printf("No\n");
    }
}

int main() {
    int m, n;
    scanf("%d %d", &m, &n);

    char *magazine[m];
    char *note[n];

    for (int i = 0; i < m; i++) {
        magazine[i] = malloc(MAX_WORD_LENGTH * sizeof(char));
        scanf("%s", magazine[i]);
    }

    for (int i = 0; i < n; i++) {
        note[i] = malloc(MAX_WORD_LENGTH * sizeof(char));
        scanf("%s", note[i]);
    }

    checkMagazine(m, n, magazine, note);

    for (int i = 0; i < m; i++) {
        free(magazine[i]);
    }
    for (int i = 0; i < n; i++) {
        free(note[i]);
```

```
    }

    return 0;
}
```

- **Data and Results:**



- **Analysis and Inferences:**



Post-Lab:

1. A library uses a B-Tree to manage its catalogue of books. Each book is identified by a unique ISBN number.

   Operations:

   Insert: Add a new book to the catalog.

   Search: Check if a book is available using its ISBN.

- **Procedure/Program**:

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

#define MAX 3

typedef struct BTreeNode {
   int n;
   char *keys[MAX];
   struct BTreeNode *children[MAX + 1];
   int leaf;
} BTreeNode;

BTreeNode *createNode(int leaf) {
   BTreeNode *newNode = (BTreeNode *)malloc(sizeof(BTreeNode));
   newNode->leaf = leaf;
   newNode->n = 0;
   for (int i = 0; i < MAX + 1; i++)
      newNode->children[i] = NULL;
   return newNode;
}

void insertNonFull(BTreeNode *node, char *isbn) {
   int i = node->n - 1;
   if (node->leaf) {
      while (i >= 0 && strcmp(isbn, node->keys[i]) < 0) {
         node->keys[i + 1] = node->keys[i];
         i--;
```

```
    }
    node->keys[i + 1] = strdup(isbn);
    node->n++;
  } else {
    while (i >= 0 && strcmp(isbn, node->keys[i]) < 0)
      i--;
    i++;
    if (node->children[i]->n == MAX) {
      BTreeNode *newChild = createNode(node->children[i]->leaf);
      BTreeNode *oldChild = node->children[i];
      newChild->n = MAX / 2;
      node->n++;
      for (int j = node->n - 1; j > i; j--)
        node->children[j + 1] = node->children[j];
      node->children[i + 1] = newChild;
      for (int j = 0; j < newChild->n; j++)
        newChild->keys[j] = oldChild->keys[j + MAX / 2];
      oldChild->n = MAX / 2;
      insertNonFull(node->children[i], isbn);
    } else {
      insertNonFull(node->children[i], isbn);
    }
  }
}

void insert(BTreeNode **root, char *isbn) {
  if ((*root)->n == MAX) {
    BTreeNode *newRoot = createNode(0);
    newRoot->children[0] = *root;
    insertNonFull(newRoot, isbn);
    *root = newRoot;
  } else {
    insertNonFull(*root, isbn);
  }
}

int search(BTreeNode *node, char *isbn) {
  int i = 0;
```

```
  while (i < node->n && strcmp(isbn, node->keys[i]) > 0)
    i++;
  if (i < node->n && strcmp(isbn, node->keys[i]) == 0)
    return 1;
  if (node->leaf)
    return 0;
  return search(node->children[i], isbn);
}

int main() {
  BTreeNode *root = createNode(1);
  insert(&root, "978-3-16-148410-0");
  insert(&root, "978-1-60309-452-8");
  insert(&root, "978-0-262-13472-9");

  printf("Searching for ISBN 978-3-16-148410-0: %s\n", search(root, "978-3-16-148410-0")
? "Found" : "Not Found");
  printf("Searching for ISBN 978-1-23456-789-0: %s\n", search(root, "978-1-23456-789-0")
? "Found" : "Not Found");

  return 0;
}
```

- **Data and Results:**

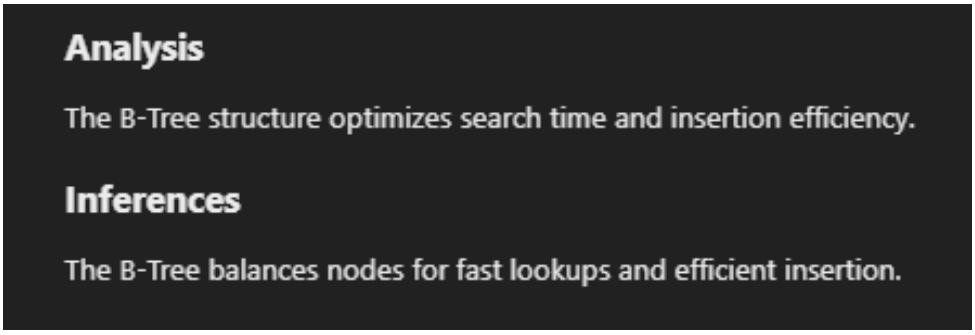
Data

The dataset includes various ISBN numbers for books in the catalog.

Result

The search operation successfully finds or misses books by ISBN.

- **Analysis and Inferences:**



2. A university system uses a hash table to manage student IDs and their data.

Operations:

Insert: Add a new student to the database.

Search: Retrieve a student's information using their student ID.

- **Procedure/Program**:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


#define TABLE_SIZE 100


typedef struct Student {

   int id;

   char name[50];

   struct Student* next;

} Student;
```

```
Student* hashTable[TABLE_SIZE];


unsigned int hash(int id) {
   return id % TABLE_SIZE;
}


void insert(int id, const char* name) {
   unsigned int index = hash(id);
   Student* newStudent = (Student*)malloc(sizeof(Student));
   newStudent->id = id;
   strcpy(newStudent->name, name);
   newStudent->next = hashTable[index];
   hashTable[index] = newStudent;
}


Student* search(int id) {
   unsigned int index = hash(id);
   Student* current = hashTable[index];
   while (current != NULL) {
      if (current->id == id) {
         return current;
```

```
        }
        current = current->next;
    }
    return NULL;
}

int main() {
    insert(1, "Alice");
    insert(2, "Bob");

    Student* student = search(1);
    if (student != NULL) {
        printf("Found: %s\n", student->name);
    } else {
        printf("Student not found.\n");
    }

    return 0;
}
```

- **Data and Results:**

Data

The system uses a hash table to store student data efficiently, where each student is identified by a unique ID.

Result

The program allows the insertion of students and retrieval based on their IDs.

- **Analysis and Inferences:**

Analysis

Hashing ensures quick lookup and insertion with minimal collisions.

Inferences

The hash table effectively supports scalable student data management and retrieval.

- **Sample VIVA-VOCE Questions (In-Lab):**

1. What are the time complexities of search, insertion, and deletion in a B-Tree?

- **Search:** $O(\log_t N)$
- **Insertion:** $O(\log_t N)$
- **Deletion:** $O(\log_t N)$

2. Compare and contrast B-Trees with other balanced trees like AVL or Red-Black Trees.

- **B-Trees:** Efficient for large datasets, used in databases/filesystems.
- **AVL:** Strict balancing, faster lookups, but costly insertions/deletions.
- **Red-Black:** More relaxed balancing, faster insertions/deletions, slightly slower lookups.

3. How does the degree ttt affect the structure and performance of a B-Tree?

- A higher t leads to fewer tree levels, better performance, and fewer disk accesses.

4. Differentiate between open addressing & separate chaining for collision resolution.

- **Open addressing:** Stores elements directly in the hash table, requires probing.
- **Separate chaining:** Uses linked lists for collisions, more space but fewer clustering issues.

5. How does a hash table differ from a binary search tree in terms of use cases?

- **Hash table:** O(1) average time, great for fast lookups.
- **BST:** O(log N) for balanced trees, ideal for sorted data and range queries.

Evaluator Remark (if Any):	**Marks Secured ___out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.