Shubham Laxman Hippargi
224CS3014

DSAD Assignment #1

Code:

```
#include <iostream>

#include <fstream>

#include <vector>

#include <list>

#include <cmath>

#include <string>

#include <algorithm>


using namespace std;


// Function to process token to 10 characters

string processToken(const string &token)

{

    string result = token;

    if (result.length() > 10)

    {

        result = result.substr(0, 10); // Truncate

    }

    while (result.length() < 10)

    {

        result += '*'; // Pad with '*'

    }

    return result;

}


// Function to extract tokens from a file

vector<string> extractTokens(const string &filename)

{

    ifstream file(filename);

    vector<string> tokens;
```

```
    string word;
    if (!file.is_open())
    {
        cerr << "Error opening file!" << endl;
        return tokens;
    }
    string forbidden = ",. ";
    while (file >> word)
    {
        string token;
        for (char c : word)
        {
            if (forbidden.find(c) != string::npos)
            {
                if (!token.empty())
                {
                    tokens.push_back(processToken(token));
                    token.clear();
                }
            }
            else
            {
                token += c;
            }
        }
        if (!token.empty())
        {
            tokens.push_back(processToken(token));
        }
    }
    file.close();
    return tokens;
}
```

```
// Open Hashing (Unsorted Chaining)
class OpenHashTableUnsorted
{
private:
    int m;
    vector<list<string>> table;
    int probeCount = 0;

    int hashFunction(int key)
    {
        const double A = (sqrt(5) - 1) / 2;
        return floor(m * (key * A - floor(key * A)));
    }

public:
    OpenHashTableUnsorted(int size) : m(size), table(size) {}

    int calculateKey(const string &token)
    {
        int key = 0;
        for (char c : token)
        {
            key += int(c);
        }
        return key;
    }

    void insert(const string &token)
    {
        int key = calculateKey(token);
        int index = hashFunction(key);
        table[index].push_back(token);
    }
```

```
    bool search(const string &token)
    {
        int key = calculateKey(token);
        int index = hashFunction(key);
        for (const auto &item : table[index])
        {
            probeCount++;
            if (item == token)
            {
                return true;
            }
        }
        return false;
    }

    void remove(const string &token)
    {
        int key = calculateKey(token);
        int index = hashFunction(key);
        table[index].remove(token);
    }

    int getProbeCount()
    {
        return probeCount;
    }

    void resetProbeCount()
    {
        probeCount = 0;
    }
};

// Open Hashing (sorted Chaining)
```

```
class OpenHashTableSorted
{
private:
    int m;
    vector<list<string>> table;
    int probeCount = 0;

    int hashFunction(int key)
    {
        const double A = (sqrt(5) - 1) / 2;
        return floor(m * (key * A - floor(key * A)));
    }

public:
    OpenHashTableSorted(int size) : m(size), table(size) {}

    int calculateKey(const string &token)
    {
        int key = 0;
        for (char c : token)
        {
            key += int(c);
        }
        return key;
    }

    void insert(const string &token)
    {
        int key = calculateKey(token);
        int index = hashFunction(key);
        table[index].insert(lower_bound(table[index].begin(), table[index].end(), token), token);
    }

    bool search(const string &token)
```

```
    {
        int key = calculateKey(token);
        int index = hashFunction(key);
        return find(table[index].begin(), table[index].end(), token) != table[index].end();
    }

    void remove(const string &token)
    {
        int key = calculateKey(token);
        int index = hashFunction(key);
        table[index].remove(token);
    }

    int getProbeCount()
    {
        return probeCount;
    }

    void resetProbeCount()
    {
        probeCount = 0;
    }
};

// Closed Hashing (Linear Probing)
class ClosedHashTable
{
private:
    int m;
    vector<string> table;
    int probeCount = 0;

public:
    ClosedHashTable(int size) : m(size), table(size, "") {}
```

```
int calculateKey(const string &token)
{
   int key = 0;
   for (char c : token)
   {
      key += int(c);
   }
   return key;
}

int hashFunction(int key, int i)
{
   return (key + i) % m;
}

void insert(const string &token)
{
   int key = calculateKey(token);
   for (int i = 0; i < m; i++)
   {
      int index = hashFunction(key, i);
      if (table[index] == "")
      {
         table[index] = token;
         return;
      }
      probeCount++;
   }
}

bool search(const string &token)
{
   int key = calculateKey(token);
```

```
    for (int i = 0; i < m; i++)
    {
        int index = hashFunction(key, i);
        probeCount++;
        if (table[index] == token)
        {
            return true;
        }
        else if (table[index] == "")
        {
            return false;
        }
    }
    return false;
}

void remove(const string &token)
{
    int key = calculateKey(token);
    for (int i = 0; i < m; i++)
    {
        int index = hashFunction(key, i);
        if (table[index] == token)
        {
            table[index] = "";
            return;
        }
    }
}

int getProbeCount()
{
    return probeCount;
}
```

```
    void resetProbeCount()
    {
        probeCount = 0;
    }
};

// Binary Search Tree
struct Node
{
    string key;
    Node *left;
    Node *right;

    Node(string key) : key(key), left(nullptr), right(nullptr) {}
};

class BinarySearchTree
{
private:
    Node *root;
    int probeCount = 0;

    Node *insert(Node *node, const string &key)
    {
        probeCount++;
        if (node == nullptr)
        {
            return new Node(key);
        }
        if (key < node->key)
        {
            node->left = insert(node->left, key);
        }
```

```
    else if (key > node->key)
    {
        node->right = insert(node->right, key);
    }
    return node;
}

bool search(Node *node, const string &key)
{
    probeCount++;
    if (node == nullptr)
    {
        return false;
    }
    if (key == node->key)
    {
        return true;
    }
    else if (key < node->key)
    {
        return search(node->left, key);
    }
    else
    {
        return search(node->right, key);
    }
}

Node *remove(Node *node, const string &key)
{
    probeCount++;
    if (node == nullptr)
    {
        return node;
```

```
    }
    if (key < node->key)
    {
        node->left = remove(node->left, key);
    }
    else if (key > node->key)
    {
        node->right = remove(node->right, key);
    }
    else
    {
        if (node->left == nullptr)
        {
            Node *temp = node->right;
            delete node;
            return temp;
        }
        else if (node->right == nullptr)
        {
            Node *temp = node->left;
            delete node;
            return temp;
        }
        Node *temp = findMin(node->right);
        node->key = temp->key;
        node->right = remove(node->right, temp->key);
    }
    return node;
}

Node *findMin(Node *node)
{
    while (node->left != nullptr)
    {
```

```
            node = node->left;
        }
        return node;
    }

public:
    BinarySearchTree() : root(nullptr) {}

    void insert(const string &key)
    {
        root = insert(root, key);
    }

    bool search(const string &key)
    {
        return search(root, key);
    }

    void remove(const string &key)
    {
        root = remove(root, key);
    }

    int getProbeCount()
    {
        return probeCount;
    }

    void resetProbeCount()
    {
        probeCount = 0;
    }
};
```

```
// Function to convert decimal to ternary
vector<int> decimalToTernary(int num)
{
    vector<int> ternary;
    while (num > 0)
    {
        ternary.push_back(num % 3);
        num /= 3;
    }
    reverse(ternary.begin(), ternary.end());
    return ternary;
}

// Main function
int main()
{
    // Example input values
    int m;
    cout << "enter the hash table size:" << endl;
    cin >> m;

    int n;
    cout << "enter the number of insertions:" << endl;
    cin >> n;
    int I = 10;                          // Operation sequence in decimal
    vector<int> M = {1, 2, 3, 4};                  // Methods to investigate (all four)
    string filename = "C:/Users/Shubham/Downloads/try/sample.txt"; // Input file

    vector<string> tokens = extractTokens(filename);

    // Ternary representation of I
    vector<int> operations = decimalToTernary(I);

    // Investigate each method M
```

```
for (int method : M)
{
  if (method == 1)
  {
    cout << "Method 1: Open Hashing with Unsorted Chaining\n";
    OpenHashTableUnsorted hashTable(m);
    for (int i = 0; i < n && i < tokens.size(); ++i)
    {
      hashTable.insert(tokens[i]);
    }

    for (int op : operations)
    {
      if (op == 0)
      { // Search
        hashTable.search(tokens[n]);
      }
      else if (op == 1)
      { // Insert
        hashTable.insert(tokens[n]);
      }
      else if (op == 2)
      { // Delete
        hashTable.remove(tokens[n]);
      }
    }
    cout << "Probes: " << hashTable.getProbeCount() << endl;
    hashTable.resetProbeCount();
    for (int i = 0; i < n && i < tokens.size(); ++i)
    {
      hashTable.search(tokens[i]);
    }
    int totalProbesSuccessfulSearch = hashTable.getProbeCount();
    double avgProbesSuccessfulSearch = totalProbesSuccessfulSearch / static_cast<double>(n);
```

```
        cout << "Average probes for successful search: " << avgProbesSuccessfulSearch << endl;


        // Unsuccessful search probe count
        hashTable.resetProbeCount();
        for (int i = n; i < n + 5 && i < tokens.size(); ++i)
        { // Searching for non-inserted tokens
          hashTable.search(tokens[i]);
        }
        int totalProbesUnsuccessfulSearch = hashTable.getProbeCount();
        double avgProbesUnsuccessfulSearch = totalProbesUnsuccessfulSearch / static_cast<double>(5); // Assuming
5 unsuccessful searches
        cout << "Average probes for unsuccessful search: " << avgProbesUnsuccessfulSearch << endl;


        // Insert probe count
        hashTable.resetProbeCount();
        for (int i = n; i < n + 5 && i < tokens.size(); ++i)
        { // Inserting additional tokens
          hashTable.insert(tokens[i]);
        }
        int totalProbesInsert = hashTable.getProbeCount();
        double avgProbesInsert = totalProbesInsert / static_cast<double>(5); // Assuming 5 insertions
        cout << "Average probes for insert: " << avgProbesInsert << endl;


        // Delete probe count
        hashTable.resetProbeCount();
        for (int i = 0; i < n && i < tokens.size(); ++i)
        {
          hashTable.remove(tokens[i]);
        }
        int totalProbesDelete = hashTable.getProbeCount();
        double avgProbesDelete = totalProbesDelete / static_cast<double>(n);
        cout << "Average probes for delete: " << avgProbesDelete << endl;
      }
    // Implement the other methods similarly (method == 2, 3, 4)...
```

```
else if (method == 2)
{
    cout << "Method 2: Open Hashing with Sorted Chaining\n";
    OpenHashTableSorted hashTable(m);
    for (int i = 0; i < n && i < tokens.size(); ++i)
    {
        hashTable.insert(tokens[i]);
    }
    for (int op : operations)
    {
        if (op == 0)
        { // Search
            hashTable.search(tokens[n]);
        }
        else if (op == 1)
        { // Insert
            hashTable.insert(tokens[n]);
        }
        else if (op == 2)
        { // Delete
            hashTable.remove(tokens[n]);
        }
    }
    cout << "Probes: " << hashTable.getProbeCount() << endl;
    hashTable.resetProbeCount();
    for (int i = 0; i < n && i < tokens.size(); ++i)
    {
        hashTable.search(tokens[i]);
    }
    int totalProbesSuccessfulSearch = hashTable.getProbeCount();
    double avgProbesSuccessfulSearch = totalProbesSuccessfulSearch / static_cast<double>(n);
    cout << "Average probes for successful search: " << avgProbesSuccessfulSearch << endl;

    // Unsuccessful search probe count
```

```
        hashTable.resetProbeCount();
        for (int i = n; i < n + 5 && i < tokens.size(); ++i)
        { // Searching for non-inserted tokens
          hashTable.search(tokens[i]);
        }
        int totalProbesUnsuccessfulSearch = hashTable.getProbeCount();
        double avgProbesUnsuccessfulSearch = totalProbesUnsuccessfulSearch / static_cast<double>(5); // Assuming 5 unsuccessful searches
        cout << "Average probes for unsuccessful search: " << avgProbesUnsuccessfulSearch << endl;

        // Insert probe count
        hashTable.resetProbeCount();
        for (int i = n; i < n + 5 && i < tokens.size(); ++i)
        { // Inserting additional tokens
          hashTable.insert(tokens[i]);
        }
        int totalProbesInsert = hashTable.getProbeCount();
        double avgProbesInsert = totalProbesInsert / static_cast<double>(5); // Assuming 5 insertions
        cout << "Average probes for insert: " << avgProbesInsert << endl;

        // Delete probe count
        hashTable.resetProbeCount();
        for (int i = 0; i < n && i < tokens.size(); ++i)
        {
          hashTable.remove(tokens[i]);
        }
        int totalProbesDelete = hashTable.getProbeCount();
        double avgProbesDelete = totalProbesDelete / static_cast<double>(n);
        cout << "Average probes for delete: " << avgProbesDelete << endl;
      }

      else if (method == 3)
      {
        cout << "Method 3: Closed Hashing with Linear Probing\n";
```

```
ClosedHashTable hashTable(m);
for (int i = 0; i < n && i < tokens.size(); ++i)
{
   hashTable.insert(tokens[i]);
}
for (int op : operations)
{
   if (op == 0)
   { // Search
      hashTable.search(tokens[n]);
   }
   else if (op == 1)
   { // Insert
      hashTable.insert(tokens[n]);
   }
   else if (op == 2)
   { // Delete
      hashTable.remove(tokens[n]);
   }
}
cout << "Probes: " << hashTable.getProbeCount() << endl;
hashTable.resetProbeCount();
for (int i = 0; i < n && i < tokens.size(); ++i)
{
   hashTable.search(tokens[i]);
}
int totalProbesSuccessfulSearch = hashTable.getProbeCount();
double avgProbesSuccessfulSearch = totalProbesSuccessfulSearch / static_cast<double>(n);
cout << "Average probes for successful search: " << avgProbesSuccessfulSearch << endl;

// Unsuccessful search probe count
hashTable.resetProbeCount();
for (int i = n; i < n + 5 && i < tokens.size(); ++i)
{ // Searching for non-inserted tokens
```

```
            hashTable.search(tokens[i]);
        }
        int totalProbesUnsuccessfulSearch = hashTable.getProbeCount();
        double avgProbesUnsuccessfulSearch = totalProbesUnsuccessfulSearch / static_cast<double>(5); // Assuming 
5 unsuccessful searches
        cout << "Average probes for unsuccessful search: " << avgProbesUnsuccessfulSearch << endl;

        // Insert probe count
        hashTable.resetProbeCount();
        for (int i = n; i < n + 5 && i < tokens.size(); ++i)
        { // Inserting additional tokens
            hashTable.insert(tokens[i]);
        }
        int totalProbesInsert = hashTable.getProbeCount();
        double avgProbesInsert = totalProbesInsert / static_cast<double>(5); // Assuming 5 insertions
        cout << "Average probes for insert: " << avgProbesInsert << endl;

        // Delete probe count
        hashTable.resetProbeCount();
        for (int i = 0; i < n && i < tokens.size(); ++i)
        {
            hashTable.remove(tokens[i]);
        }
        int totalProbesDelete = hashTable.getProbeCount();
        double avgProbesDelete = totalProbesDelete / static_cast<double>(n);
        cout << "Average probes for delete: " << avgProbesDelete << endl;
    }

    else if (method == 4)
    {
        cout << "Method 4: Binary Search Tree\n";
        BinarySearchTree bst;
        for (int i = 0; i < n && i < tokens.size(); ++i)
        {
```

```
    bst.insert(tokens[i]);
}
for (int op : operations)
{
    if (op == 0)
    { // Search
        bst.search(tokens[n]);
    }
    else if (op == 1)
    { // Insert
        bst.insert(tokens[n]);
    }
    else if (op == 2)
    { // Delete
        bst.remove(tokens[n]);
    }
}
cout << "Probes: " << bst.getProbeCount() << endl;
bst.resetProbeCount();
for (int i = 0; i < n && i < tokens.size(); ++i)
{
    bst.search(tokens[i]);
}
int totalProbesSuccessfulSearch = bst.getProbeCount();
double avgProbesSuccessfulSearch = totalProbesSuccessfulSearch / static_cast<double>(n);
cout << "Average probes for successful search: " << avgProbesSuccessfulSearch << endl;

// Unsuccessful search probe count
bst.resetProbeCount();
for (int i = n; i < n + 5 && i < tokens.size(); ++i)
{ // Searching for non-inserted tokens
    bst.search(tokens[i]);
}
int totalProbesUnsuccessfulSearch = bst.getProbeCount();
```

```
        double avgProbesUnsuccessfulSearch = totalProbesUnsuccessfulSearch / static_cast<double>(5); // Assuming 5 unsuccessful searches
        cout << "Average probes for unsuccessful search: " << avgProbesUnsuccessfulSearch << endl;

        // Insert probe count
        bst.resetProbeCount();
        for (int i = n; i < n + 5 && i < tokens.size(); ++i)
        { // Inserting additional tokens
          bst.insert(tokens[i]);
        }
        int totalProbesInsert = bst.getProbeCount();
        double avgProbesInsert = totalProbesInsert / static_cast<double>(5); // Assuming 5 insertions
        cout << "Average probes for insert: " << avgProbesInsert << endl;

        // Delete probe count
        bst.resetProbeCount();
        for (int i = 0; i < n && i < tokens.size(); ++i)
        {
          bst.remove(tokens[i]);
        }
        int totalProbesDelete = bst.getProbeCount();
        double avgProbesDelete = totalProbesDelete / static_cast<double>(n);
        cout << "Average probes for delete: " << avgProbesDelete << endl;
      }
    }

    return 0;
}
```

Output:

```
PS C:\Users\Shubham\Downloads\try\output> cd 'c:\Users\Shubham\Downloads\try\output'
PS C:\Users\Shubham\Downloads\try\output> & .\'try.exe'
enter the hash table size:
10
enter the number of insertions:
5
Method 1: Open Hashing with Unsorted Chaining
Probes: 1
Average probes for successful search: 1.2
Average probes for unsuccessful search: 0.8
Average probes for insert: 0
Average probes for delete: 0
Method 2: Open Hashing with Sorted Chaining
Probes: 0
Average probes for successful search: 0
Average probes for unsuccessful search: 0
Average probes for insert: 0
Average probes for delete: 0
Method 3: Closed Hashing with Linear Probing
Probes: 11
Average probes for successful search: 1.6
Average probes for unsuccessful search: 4.4
Average probes for insert: 7.4
Average probes for delete: 0
Method 4: Binary Search Tree
Probes: 28
Average probes for successful search: 2.6
Average probes for unsuccessful search: 4.4
Average probes for insert: 4.4
Average probes for delete: 3
PS C:\Users\Shubham\Downloads\try\output>
```

Page No.
Date

* Observations :-

01. Open Hashing with Unsorted Chaining:-

A. The average probe tends to be low for sucessful searches as elements are searched sequentially in unsorted linked lists at each hash index.

B. Unsucessful searches , the number of size when increased leads to increase in number of probes.

C. Insertion:- Inserts are relatively efficient as new elements are added to the end of list without any sorting

D. Deletion is inefficient

02. Open hashing with Sorted chaining:-

A. The average number of probes is less than that of unsorted chaining due to sorting in chaining process

B. unsecussful search performs better in sorted chaining

C. Insertion requires number of more probes to keep required sorted order

D. Deletion :- is efficient if the element is in front

03. Closed hashing with linear Probing

A. The effect of clustering increases the probe count for sucessful search

B. Unsucess-ful search : if the element does not exist many slots need to be checked.

C. Insertion is efficient in sparse tables, but as the table fills up , more collision occurs.

D. Deletion is inefficient.

04. Binary Search Tree (BST) :-

A. Performs better if the tree is balanced for sucessful searches.

B. Similar to sucessful searches, efficient for balanced tree

C. Insertion is difficult if the tree becomes unbalanced

D. Deletion becomes harder if it has two children

* A. Sorted Chaining is best for efficient searches
  B. Linear Probing is good for small datasets
  C. Binary Search Trees are less efficient if unbalanced