Programming Practice to Perform CRT

Linear Probing Code:

CPP:

```
#include <iostream>
using namespace std;

class HashTable {
private:
   int *hashTable;
   int tableSize;
   int currentSize;

public:
   HashTable(int size) {
      tableSize = size;
      hashTable = new int[tableSize];
      for (int i = 0; i < tableSize; i++) {
         hashTable[i] = -1;  // -1 indicates an empty slot
      }
      currentSize = 0;
   }
   int hashFunction(int key) {
      return key % tableSize;
   }
   void insert(int key) {
      if (currentSize >= tableSize) {
         cout << "Hash Table is full" << endl;
         return;
      }
      int hashValue = hashFunction(key);
      while (hashTable[hashValue] != -1) {
         hashValue = (hashValue + 1) % tableSize;  // Linear probing
      }
      hashTable[hashValue] = key;
      currentSize++;
   }
   bool search(int key) {
      int hashValue = hashFunction(key);
      int initialHash = hashValue;

      while (hashTable[hashValue] != -1) {
         if (hashTable[hashValue] == key) {
            return true;
         }
         hashValue = (hashValue + 1) % tableSize;
         if (hashValue == initialHash) {  // Came back to the start
            break;
         }
      }
```

```
        return false;
    }
    void display() {
        for (int i = 0; i < tableSize; i++) {
            if (hashTable[i] != -1) {
                cout << "Index " << i << ": " << hashTable[i] << endl;
            } else {
                cout << "Index " << i << ": Empty" << endl;
            }
        }
    }
    ~HashTable() {
        delete[] hashTable;
    }
};
int main() {
    HashTable ht(7);
    ht.insert(10);
    ht.insert(20);
    ht.insert(15);
    ht.insert(7);
    ht.insert(30);

    ht.display();

    cout << "Search 15: " << (ht.search(15) ? "Found" : "Not Found") << endl;  // Output: Found
    cout << "Search 25: " << (ht.search(25) ? "Found" : "Not Found") << endl;  // Output: Not Found

    return 0;
}
```

Java:

```
class HashTable {
    private int[] hashTable;
    private int tableSize;
    private int currentSize;

    public HashTable(int size) {
        tableSize = size;
        hashTable = new int[tableSize];
        for (int i = 0; i < tableSize; i++) {
            hashTable[i] = -1;  // -1 indicates an empty slot
        }
        currentSize = 0;
    }

    private int hashFunction(int key) {
        return key % tableSize;
```

```
}

public void insert(int key) {
    if (currentSize >= tableSize) {
        System.out.println("Hash Table is full");
        return;
    }

    int hashValue = hashFunction(key);
    while (hashTable[hashValue] != -1) {
        hashValue = (hashValue + 1) % tableSize;  // Linear probing
    }
    hashTable[hashValue] = key;
    currentSize++;
}

public boolean search(int key) {
    int hashValue = hashFunction(key);
    int initialHash = hashValue;

    while (hashTable[hashValue] != -1) {
        if (hashTable[hashValue] == key) {
            return true;
        }
        hashValue = (hashValue + 1) % tableSize;
        if (hashValue == initialHash) {  // Came back to the start
            break;
        }
    }
    return false;
}

public void display() {
    for (int i = 0; i < tableSize; i++) {
        if (hashTable[i] != -1) {
            System.out.println("Index " + i + ": " + hashTable[i]);
        } else {
            System.out.println("Index " + i + ": Empty");
        }
    }
}

public static void main(String[] args) {
    HashTable ht = new HashTable(7);
    ht.insert(10);
    ht.insert(20);
    ht.insert(15);
```

```
        ht.insert(7);
        ht.insert(30);

        ht.display();

        System.out.println("Search 15: " + ht.search(15));  // Output: true
        System.out.println("Search 25: " + ht.search(25));  // Output: false
    }
}
```

Quadratic Probing Code:

CPP

```
#include <iostream>
using namespace std;

class HashTable {
private:
    int *hashTable;
    int tableSize;
    int currentSize;

public:
    HashTable(int size) {
        tableSize = size;
        hashTable = new int[tableSize];
        for (int i = 0; i < tableSize; i++) {
            hashTable[i] = -1;  // -1 indicates an empty slot
        }
        currentSize = 0;
    }

    int hashFunction(int key) {
        return key % tableSize;
    }

    void insert(int key) {
        if (currentSize >= tableSize) {
            cout << "Hash Table is full" << endl;
            return;
        }

        int hashValue = hashFunction(key);
        int i = 0;
        while (hashTable[(hashValue + i * i) % tableSize] != -1) {
            i++;
```

```
        }
        hashTable[(hashValue + i * i) % tableSize] = key;
        currentSize++;
    }

    bool search(int key) {
        int hashValue = hashFunction(key);
        int i = 0;
        while (hashTable[(hashValue + i * i) % tableSize] != -1) {
            if (hashTable[(hashValue + i * i) % tableSize] == key) {
                return true;
            }
            i++;
            if (i == tableSize) {  // To avoid infinite loops
                break;
            }
        }
        return false;
    }

    void display() {
        for (int i = 0; i < tableSize; i++) {
            if (hashTable[i] != -1) {
                cout << "Index " << i << ": " << hashTable[i] << endl;
            } else {
                cout << "Index " << i << ": Empty" << endl;
            }
        }
    }

    ~HashTable() {
        delete[] hashTable;
    }
};

int main() {
    HashTable ht(7);
    ht.insert(10);
    ht.insert(20);
    ht.insert(15);
    ht.insert(7);
    ht.insert(30);

    ht.display();

    cout << "Search 15: " << (ht.search(15) ? "Found" : "Not Found") << endl;  // Output: Found
    cout << "Search 25: " << (ht.search(25) ? "Found" : "Not Found") << endl;  // Output: Not Found
```

```
    return 0;
}
```

Java

```
class HashTable {
    private int[] hashTable;
    private int tableSize;
    private int currentSize;

    public HashTable(int size) {
        tableSize = size;
        hashTable = new int[tableSize];
        for (int i = 0; i < tableSize; i++) {
            hashTable[i] = -1;  // -1 indicates an empty slot
        }
        currentSize = 0;
    }

    private int hashFunction(int key) {
        return key % tableSize;
    }

    public void insert(int key) {
        if (currentSize >= tableSize) {
            System.out.println("Hash Table is full");
            return;
        }

        int hashValue = hashFunction(key);
        int i = 0;
        while (hashTable[(hashValue + i * i) % tableSize] != -1) {
            i++;
        }
        hashTable[(hashValue + i * i) % tableSize] = key;
        currentSize++;
    }

    public boolean search(int key) {
        int hashValue = hashFunction(key);
        int i = 0;
        while (hashTable[(hashValue + i * i) % tableSize] != -1) {
            if (hashTable[(hashValue + i * i) % tableSize] == key) {
                return true;
            }
            i++;
```

```
            if (i == tableSize) {  // To avoid infinite loops
                break;
            }
        }
        return false;
    }

    public void display() {
        for (int i = 0; i < tableSize; i++) {
            if (hashTable[i] != -1) {
                System.out.println("Index " + i + ": " + hashTable[i]);
            } else {
                System.out.println("Index " + i + ": Empty");
            }
        }
    }

    public static void main(String[] args) {
        HashTable ht = new HashTable(7);
        ht.insert(10);
        ht.insert(20);
        ht.insert(15);
        ht.insert(7);
        ht.insert(30);

        ht.display();

        System.out.println("Search 15: " + ht.search(15));  // Output: true
        System.out.println("Search 25: " + ht.search(25));  // Output: false
    }
}
```

Double Hashing Code:

CPP

```
#include <iostream>
using namespace std;

class HashTable {
private:
    int *hashTable;
    int tableSize;
    int currentSize;

public:
    HashTable(int size) {
```

```
    tableSize = size;
    hashTable = new int[tableSize];
    for (int i = 0; i < tableSize; i++) {
        hashTable[i] = -1;  // -1 indicates an empty slot
    }
    currentSize = 0;
}

int hashFunction1(int key) {
    return key % tableSize;
}

int hashFunction2(int key) {
    return 7 - (key % 7);  // Secondary hash function
}

void insert(int key) {
    if (currentSize >= tableSize) {
        cout << "Hash Table is full" << endl;
        return;
    }

    int hashValue = hashFunction1(key);
    int i = 0;
    while (hashTable[(hashValue + i * hashFunction2(key)) % tableSize] != -1) {
        i++;
    }
    hashTable[(hashValue + i * hashFunction2(key)) % tableSize] = key;
    currentSize++;
}

bool search(int key) {
    int hashValue = hashFunction1(key);
    int i = 0;
    while (hashTable[(hashValue + i * hashFunction2(key)) % tableSize] != -1) {
        if (hashTable[(hashValue + i * hashFunction2(key)) % tableSize] == key) {
            return true;
        }
        i++;
        if (i == tableSize) {  // To avoid infinite loops
            break;
        }
    }
    return false;
}

void display() {
```

```
        for (int i = 0; i < tableSize; i++) {
            if (hashTable[i] != -1) {
                cout << "Index " << i << ": " << hashTable[i] << endl;
            } else {
                cout << "Index " << i << ": Empty" << endl;
            }
        }
    }

    ~HashTable() {
        delete[] hashTable;
    }
};

int main() {
    HashTable ht(7);
    ht.insert(10);
    ht.insert(20);
    ht.insert(15);
    ht.insert(7);
    ht.insert(30);

    ht.display();

    cout << "Search 15: " << (ht.search(15) ? "Found" : "Not Found") << endl;  // Output: Found
    cout << "Search 25: " << (ht.search(25) ? "Found" : "Not Found") << endl;  // Output: Not Found

    return 0;
}
```

Java

```
class HashTable {
    private int[] hashTable;
    private int tableSize;
    private int currentSize;

    public HashTable(int size) {
        tableSize = size;
        hashTable = new int[tableSize];
        for (int i = 0; i < tableSize; i++) {
            hashTable[i] = -1;  // -1 indicates an empty slot
        }
        currentSize = 0;
    }

    private int hashFunction1(int key) {
```

```
    return key % tableSize;
}

private int hashFunction2(int key) {
    return 7 - (key % 7);  // Secondary hash function (must not be 0)
}

public void insert(int key) {
    if (currentSize >= tableSize) {
        System.out.println("Hash Table is full");
        return;
    }

    int hashValue = hashFunction1(key);
    int i = 0;
    while (hashTable[(hashValue + i * hashFunction2(key)) % tableSize] != -1) {
        i++;
    }
    hashTable[(hashValue + i * hashFunction2(key)) % tableSize] = key;
    currentSize++;
}

public boolean search(int key) {
    int hashValue = hashFunction1(key);
    int i = 0;
    while (hashTable[(hashValue + i * hashFunction2(key)) % tableSize] != -1) {
        if (hashTable[(hashValue + i * hashFunction2(key)) % tableSize] == key) {
            return true;
        }
        i++;
        if (i == tableSize) {  // To avoid infinite loops
            break;
        }
    }
    return false;
}

public void display() {
    for (int i = 0; i < tableSize; i++) {
        if (hashTable[i] != -1) {
            System.out.println("Index " + i + ": " + hashTable[i]);
        } else {
            System.out.println("Index " + i + ": Empty");
        }
    }
}
```

```
    public static void main(String[] args) {
        HashTable ht = new HashTable(7);
        ht.insert(10);
        ht.insert(20);
        ht.insert(15);
        ht.insert(7);
        ht.insert(30);

        ht.display();

        System.out.println("Search 15: " + ht.search(15));  // Output: true
        System.out.println("Search 25: " + ht.search(25));  // Output: false
    }
}
```

Chaining Code:

CPP

```
#include <iostream>
#include <list>
using namespace std;

class HashTable {
private:
    list<int> *hashTable;
    int tableSize;

public:
    HashTable(int size) {
        tableSize = size;
        hashTable = new list<int>[tableSize];  // Initialize hash table with an array of linked lists
    }

    int hashFunction(int key) {
        return key % tableSize;
    }

    void insert(int key) {
        int hashValue = hashFunction(key);
        hashTable[hashValue].push_back(key);  // Insert key into the linked list of the computed hash bucket
    }

    bool search(int key) {
        int hashValue = hashFunction(key);
        for (auto x : hashTable[hashValue]) {
```

```
            if (x == key) {
                return true;
            }
        }
        return false;
    }

    void deleteKey(int key) {
        int hashValue = hashFunction(key);
        auto it = hashTable[hashValue].begin();
        while (it != hashTable[hashValue].end()) {
            if (*it == key) {
                hashTable[hashValue].erase(it);  // Erase key from the linked list
                cout << "Key " << key << " deleted." << endl;
                return;
            }
            ++it;
        }
        cout << "Key " << key << " not found." << endl;
    }

    void display() {
        for (int i = 0; i < tableSize; i++) {
            cout << "Bucket " << i << ": ";
            for (auto x : hashTable[i]) {
                cout << x << " -> ";
            }
            cout << "null" << endl;
        }
    }

    ~HashTable() {
        delete[] hashTable;
    }
};

int main() {
    HashTable ht(7);
    ht.insert(10);
    ht.insert(20);
    ht.insert(15);
    ht.insert(7);
    ht.insert(30);

    ht.display();

    cout << "Search 15: " << (ht.search(15) ? "Found" : "Not Found") << endl;  // Output: Found
```

```
    cout << "Search 25: " << (ht.search(25) ? "Found" : "Not Found") << endl;  // Output: Not Found

    ht.deleteKey(15);
    ht.display();

    return 0;
}
```

Java

```
import java.util.LinkedList;

class HashTable {
    private LinkedList<Integer>[] hashTable;
    private int tableSize;

    public HashTable(int size) {
        tableSize = size;
        hashTable = new LinkedList[tableSize];
        for (int i = 0; i < tableSize; i++) {
            hashTable[i] = new LinkedList<>();  // Initialize each bucket as a linked list
        }
    }

    private int hashFunction(int key) {
        return key % tableSize;
    }

    public void insert(int key) {
        int hashValue = hashFunction(key);
        hashTable[hashValue].add(key);  // Insert key into the linked list of the computed hash bucket
    }

    public boolean search(int key) {
        int hashValue = hashFunction(key);
        return hashTable[hashValue].contains(key);  // Search in the linked list
    }

    public void delete(int key) {
        int hashValue = hashFunction(key);
        if (hashTable[hashValue].contains(key)) {
            hashTable[hashValue].remove((Integer) key);  // Remove the key from the linked list
            System.out.println("Key " + key + " deleted.");
        } else {
            System.out.println("Key " + key + " not found.");
        }
    }
```

```
    public void display() {
        for (int i = 0; i < tableSize; i++) {
            System.out.print("Bucket " + i + ": ");
            for (int key : hashTable[i]) {
                System.out.print(key + " -> ");
            }
            System.out.println("null");
        }
    }

    public static void main(String[] args) {
        HashTable ht = new HashTable(7);
        ht.insert(10);
        ht.insert(20);
        ht.insert(15);
        ht.insert(7);
        ht.insert(30);

        ht.display();

        System.out.println("Search 15: " + ht.search(15));  // Output: true
        System.out.println("Search 25: " + ht.search(25));  // Output: false

        ht.delete(15);
        ht.display();
    }
}
```