Programming Assignment 3

Lead TAs: Aamil Khan Mahar and Faaiz Umer

Deadline: November 8, 2024

Contents

1 Hash Table (30 Marks)

1.1 Overview

A hash table is a data structure that maps keys to values for efficient data retrieval. This task covers various collision handling strategies like linear probing, quadratic probing, double hashing, and separate chaining.

1.2 Class Definition

```
template <typename T>
class HashTable {
public:
    enum CollisionHandling { LINEAR_PROBING,
        QUADRATIC_PROBING, DOUBLE_HASHING, SEPARATE_CHAINING
        };
    HashTable(CollisionHandling strategy);
    ~HashTable();
    void insert(int key, T value);
    T search(int key);
    void remove(int key);
    void printTable();
private:
    CollisionHandling collisionStrategy;
    std::vector<KeyValue<T>> probingTable;
    std::vector<std::vector<KeyValue<T>>> chainingTable;
    int hashFunction1(int key);
    int hashFunction2(int key);
    void insertLinearProbing(int key, T value);
    void searchLinearProbing(int key);
    void removeLinearProbing(int key);
};
```

2 Task 2: MinHeap (15 Marks)

2.1 Overview

In this part you will be implementing a Min Heap data structure. Refer to heap.h which provides all the required definitions. Write your implementation in heap.cpp using the boiler code

2.2 Function Descriptions

- `MinHeap(int cap)`: Constructor that initializes the heap with the given capacity and allocates memory for the heap array.
- `int parent(int i)`: Returns the index of the parent node of the node at index `i` using the formula: `(i - 1) / 2`.
- `int left(int i)`: Returns the index of the left child of the node at index `i` using the formula: `2 * i + 1`.
- `int right(int i)`: Returns the index of the right child of the node at index `i` using the formula: `2 * i + 2`.
- `int extractMin()`: Removes and returns the smallest element (root) of the heap. After removal, the heap is reorganized to maintain the min-heap property.
- `void decreaseKey(int i, int new_val)`: Decreases the value of the node at index `i` to `new_val`. If the new value is smaller, the function ensures the heap property is maintained.
- `int getMin()`: Returns the smallest element in the heap without removing it.
- `void deleteKey(int i)`: Deletes the element at index `i` by first decreasing its value to negative infinity, then calling `extractMin()` to remove it from the heap.
- `void insertKey(int k)`: Inserts a new key `k` into the heap. If the heap is full, it resizes dynamically. The heap property is restored after insertion.
- `shared_ptr<int> getHeap()`: Returns a shared pointer to the internal heap array, allowing external code to access the heap contents.

3 Task 3: Sorting Algorithms (15 Marks)

3.1 Overview

In this part of the assignment, you will implement two sorting algorithms: **Insertion Sort** and **Merge Sort**. Both algorithms receive an **unsorted vector of longs** as the only parameter and must return a **sorted vector** upon successful execution.

The provided **test cases** will not only check if the functions return a correctly sorted vector but will also evaluate the **execution time** of your implementations. If the sorting time exceeds the **benchmark time** defined in the test cases, your solution will not pass.

You are free to create as many **helper functions** as you need to assist with your implementation.

3.2 Generator for Testing (Optional Manual Testing)

A file named `generator.cpp` is provided to help you visualize the output of your sorting algorithms. This generator creates sequences of numbers based on user input (either sorted or random) and allows the user to choose a sorting algorithm to apply to these sequences.

However, please note that the `generator.cpp` file is not mandatory to use and is only provided for your convenience. It is not part of the final test cases. Your sorting functions must also work for **negative numbers**, as the final test cases will validate this.

3.3 Implementation Rules

The **header file** `sorts.h` contains the declarations for the sorting functions. You are required to implement all the functions inside `sorts.cpp` only. Do not modify any other provided files. Additionally, you have been provided with an implementation of the **LinkedList** data structure in `LinkedList.h` and `LinkedList.cpp`, which must be used for the **Merge Sort** implementation.

Note: It is compulsory to pass the individual test cases for both **Insertion Sort** and **Merge Sort** to receive credit for this task.

3.4 Task 1: Insertion Sort (Array-Based)

3.4.1 Description

In this task, you will implement an **array-based insertion sort**. The elements from the unsorted vector will first be **transferred to an array**, sorted using insertion sort, and then **transferred back to a vector** (either the original vector or a new one). The sorted vector will be returned at the end.

3.4.2 Function Signature

```
vector<long> InsertionSort(vector<long> nums);
```

3.4.3 Example Implementation

```
vector<long> InsertionSort(vector<long> nums) {
    for (int i = 1; i < nums.size(); i++) {
        long key = nums[i];
        int j = i - 1;
        // Move elements greater than key one position to
            the right
        while (j >= 0 && nums[j] > key) {
            nums[j + 1] = nums[j];
            j--;
        }
        nums[j + 1] = key;
    }
    return nums;
}
```

3.5 Task 2: Merge Sort (Linked List-Based)

3.5.1 Description

In this task, you will implement **Merge Sort** using a **Linked List**. You will first **transfer all elements from the unsorted vector to a linked list**. After that, you will sort the elements within the linked list using the **merge sort algorithm** as taught in class. Finally, you will **transfer the sorted elements back to a vector** (either the original vector or a new one) and return the sorted vector.

3.5.2 Function Signature

```
vector<long> MergeSort(vector<long> nums);
```

3.5.3 Example Implementation

```
vector<long> MergeSort(vector<long> nums) {
    // Create a linked list from the input vector
    LinkedList<long> list;
    for (long num : nums) {
        list.insert(num);
    }

    // Sort the linked list using merge sort
    list.sort();

    // Convert the sorted linked list back to a vector
    return list.toVector();
}
```

3.5.4 Important Notes

- You are required to use the **LinkedList** data structure provided in `LinkedList.h` and `LinkedList.cpp` for the **Merge Sort** implementation.
- You must not modify any of the provided files except for `sorts.cpp`.
- Ensure your implementation can handle **negative numbers**, as the final test cases will verify this.

3.6 Testing and Visualization

Although not mandatory, you can use the `generator.cpp` file to visualize the output of your sorting functions. This file generates a sequence of numbers based on user input and allows you to select a sorting algorithm to apply to these numbers. The results of your implementation will be displayed on the screen.

Note: The generator is provided only for visualization purposes. The **final test cases** will be provided separately, and passing them is necessary to receive credit for this task.

3.7 Testing and Visualization

Although not mandatory, you can use the `generator.cpp` file to visualize the output of your sorting functions. This file generates a sequence of numbers based on user input and allows you to select a sorting algorithm to apply to these numbers. The results of your implementation will be displayed on the screen.

Note: The generator is provided only for visualization purposes. The **final test cases** will be provided separately, and passing them is necessary to receive credit for this task.

4 Task 4: Student Database (40 Marks)

4.1 Overview

The Student Database system manages student records using an M-ary tree structure. Each student is uniquely identified by their roll number and has additional attributes such as batch, school code, name, age, GPA, and major.

4.2 Class Definition

```
class Student {
public:
    string rollNumber;
    int batch;
    string schoolCode;
    int schoolRollNumber;
    string name;
    int age;
    double gpa;
    string major;
};
```

4.3 Implementation Instructions

Please use the M-ary tree implementation from PA2 to build the Student Database (DB). The structure of the StudentDB class will be as follows:

```
Tree<Vector<Tree<Student>>> LUMS;
```

This **nested M-ary tree** structure allows for the efficient organization of students by batch, school. Each inner tree holds the student records (as 'Student' objects) for a specific category. You must properly initialize, insert, search, and display student records within this nested tree structure.

4.4 Methods

Below are the required methods to be implemented for the **StudentDatabase** class:

- `StudentDatabase()`: Initializes the M-ary tree-based student database.
- `void addStudent(const Student student)`: Adds a new student to the appropriate tree within the database.
- `bool searchStudent(const string rollNumber)`: Searches for a student by their roll number.
- `void displayStudent(const string rollNumber)`: Displays the details of a specific student.
- `void displayBatch(const int batch)`: Displays all students from a specific batch.

- `void displayBatchAndSchool(const int batch, const string schoolCode)`: Displays students belonging to a specific batch and school.
- `void displayAll()`: Displays all student records in the database.
- `void read_csv(const string filename)`: Reads student data from a CSV file and populates the database.
- `void interface()`: The Function starts the menu to interact with the DB

4.5 Tree Structure

The M-ary tree used here allows each node to have multiple children. This structure ensures:

- **Scalable storage:** Can store large numbers of student records.
- **Efficient categorization:** Group students by batch, school, or other relevant fields.
- **Faster look-ups:** Allows faster searches by grouping records hierarchically.

5 Submission Guidelines

- Submit only the required .cpp files.
- Zip your submission with the name `PA3_<roll number>.zip`.
- Upload the zip file on LMS before the deadline.
- You have 4 "free" late days across the semester.
- Penalties for late submissions:
  - 10% penalty for up to 24 hours late.
  - 20% penalty for up to 2 days late.
  - 30% penalty for up to 3 days late.
  - No submissions accepted after 3 days.

6 Running Test Files

- For the Hashing test:

```
g++ --std=c++17 -w -o Test1 Test1.cpp
./Test1
```

- For the MinHeap test:

```
g++ --std=c++17 -w -o Test2 Test2.cpp
./Test2
```

- For the Insertion Sorting test:

```
g++ --std=c++17 -w -o Test3_1 Test3_1.cpp
./Test3_1
```