COMP 2012 Object-Oriented Programming and Data Structures

Lab 7 Hashing

Page maintained by

TANG, Mingkai
Last Modified:
11/20/2022 12:36:32

Homepage

Course Homepage

Introduction

In lab 7, we will implement a hash table using open addressing for collision handling. In this lab, you can practice adding, searching, deleting, and rehashing operations of the hash table. You can find the lecture note on hashing HERE. You can download the skeleton code HERE.

Background

After grading the midterm exam, the scores are uploaded to a database. Imagine that you are the developer of the database. You need to implement a database based on a hash table that can support three operations:

- Add a student to the database. A student is represented by a pair `(name, score)` where `name` is a string to represent the name of the student, and `score` is an integer to represent the score of the student. `name` only contains lowercase letters.
- Search for the score of a student given the `name`.
- Delete a student from the database given the `name`.

Enum, Function, and Class Introduction:

This section will introduce each enum and class in detail.

Class Student

```
class Student {
  public:
    Student(const std::string &name, int score)
        : name_(name), score_(score) {}
    Student(const Student &student)
        : name_(student.name()), score_(student.score()) {}

    const std::string& name() const {return name_;}
    int score() const {return score_;}

  private:
    std::string name_;
    int score_;
};
```

We use this class to represent the information of a student.

Enum Class HashCellState

```
enum class HashCellState {
  Empty,
  Active,
  Deleted
};
```

This enum class describes the state of a hash cell. Here we use *lazy deletion* by marking the state of each hash cell, which is described on P.32 in lecture note. The usage of the enum class has been described in lab6.

Struct HashCell

```
struct HashCell {
  HashCell() = default;
  ~HashCell() {
    delete student;
    student = nullptr;
  }
  HashCellState state;
  Student *student;
};
```

This structure maintains the information of a hash cell. `state` represents the state of a hash cell and `student` is a pointer pointing to the data put into the hash cell.

Enum Class ProbeMode

```
enum class ProbeMode {
  Linear,
  Quadratic,
  DoubleHash
};
```

You have learned three common strategies for resolving collisions in open addressing. This enum class represents these three strategies.

Class HashTable

```
class HashTable {
  public:
    HashTable(int init_size, int maximum_probe_num,
              ProbeMode probe_mode);
    ~HashTable();

    bool add(const std::string &name, int score);
    int search(const std::string &name) const;
    bool remove(const std::string &name);

    static int hashFunction(const std::string &name, int base, int mod);
    static int getNextHashTableSize(int now_hash_table_size);
    void print() const;

  private:
    void reHashUntilSuccess();
    bool reHash(int rehash_table_size);

    HashCell *table_;
    int (*probe_func_)(int hash_value1, int hash_value2, int num_probe, int
hash_table_size);

    int base_1_;
    int base_2_;

    int hash_table_size_;
    int maximum_probe_num_;
    ProbeMode probe_mode_;
};
```

This class represents a hash table. The data members are described as follows.

- `table_`: This pointer points to a dynamically allocated array of hash cells.
- `base_1_` and `base_2_`: These two variables will be used to calculate the hash function. `base_1_` is set to `37` and `base_2_` is set to `41`.
- `hash_table_size_` : This variable represents the number of hash cells.
- `maximum_probe_num_`: This variable represents the maximum number of probes when inserting a new element. If the number of probes larger or equal than `maximum_probe_num_` and still can not find a position to insert, it is assumed that the hash table is too full, and needs to be rehashed. **Note that it is a bit different from P.33 in lecture note which uses a load factor to determine when to rehash**. After rehashing, add the data back to the new hash table. More detail will be given in Task 2 and Task 5.
- `probe_mode_`: This variable represents the probe mode.
- `probe_func_`: This variable is a function pointer to represent the probe function. More detail will be given in Task 1.

```
HashTable(int init_size, int maximum_probe_num, ProbeMode probe_mode);
~HashTable();
```

These two functions are the constructor and the destructor of the hash table. `init_size` is the initial size of the hash table. `maximum_probe_num` is the maximum number of probes for inserting elements into the hash table. `probe_mode` is the probe mode in the hash table. You need to implement these two functions in Task 1.

```
bool add(const std::string &name, int score);
```

This function adds new data to the hash table. If the `name` is already in the hash table, you cannot insert the student again into the hash table; simply return `false`. If it can find a proper position to insert, insert it and return `true`. You need to implement it in Task 2 and Task 4.

```
int search(const std::string &name) const;
```

This function searches the score given by the `name` of the student. If the student can be found, return the student's `score`; otherwise, return `-1`. You need to implement it in Task 3.

```
bool remove(const std::string &name);
```

This function deletes the data given by the `name` of the student. If the student can be found, delete it from the hash table and return `true`; otherwise, return `false`. You need to implement it in Task 4.

```
static int hashFunction(const std::string &name, int base, int table_size);
```

This static function is implemented in `hash_table.h/cpp`. It defines a hash function for compute an index for the `name`. The return value represents the string's *home position* using the following formula.

$$\sum_{i=0}^{len-1} base^{len-i-1} \cdot (name[i] - 'a') \% table\_size$$

where $base$, $name$, $table\_size$ are the function's parameter, and len is the size of $name$. It is the same as the formula on P.14 in the lecture note when the $base = 37$.

```
static int getNextHashTableSize(int now_hash_table_size);
```

This static function is implemented in `hash_table.h/cpp`. When it needs to rehash, the size of the table will be the smallest prime number that is larger than two times the original table size. `now_hash_table_size` is the original size of the hash table. The function will return the smallest prime number larger than `2*now_hash_table_size`. This function is used in `void HashTable::reHashUntilSuccess()`.

```
bool reHash(int rehash_table_size);
```

This function tries rehashing in a hash table whose size is `rehash_table_size`. If the new hash table is built successfully, return `true`; otherwise, return `false`. You need to implement it in Task 5.

```
void reHashUntilSuccess();
```

This function is implemented in `hash_table.h/cpp`. It continuously tries to do rehashing with a larger size of the hash table until rehashing successfully. It is implemented by calling `int HashTable::GetNextHashTableSize(int now_hash_table_size)` and `bool HashTable::reHash(int rehash_table_size)`

```
void print() const;
```

This function is implemented in `hash_table.h/cpp`. It prints the information of `HashTable` for debugging and correctness checking.

Lab tasks

You can download the skeleton code HERE.

Task 1

```
HashTable(int init_size, int maximum_probe_num, ProbeMode probe_mode);
```

Implement the constructor. The parameters are:

- `init_size` is the initial size of the hash table.
- `maximum_probe_num` is the maximum number of probes for inserting elements in the hash table.
- `probe_mode` is the probe mode in the hash table.

Hint: You can implement the function based on this scheme:

1. Set `base_1_` to `37` and set `base_2_` to `41`.
2. Initialize `hash_table_size_`, `maximum_probe_num_` and `probe_mode_` by the input parameters.
3. Allocate memory to `table_` and initialize the value of each hash cells.
4. Assign a probe function to `probe_func_` according to `probe_mode_`. You should write the function as a lambda expression. The type of lambda expression should be the same as `probe_func_`. You can write statement like `probe_func_=<lambda expression>`. The lambda expression should have four integers `int hash_value1`, `int hash_value2`, `int num_probe`, `int hash_table_size`, as the input parameters and returns an integer representing the position to probe. The meaning of the four input parameters is described as follows.
    - `hash_value1` represents the hash function value. It is the same as `hash(k)` on P.22, 25, 28 in the lecture note.
    - `hash_value2` represents the second hash function value in double hashing. It is the same as `hash_2(k)` on p28 in the lecture note. For linear and quadratic probe modes, you don't need to use `hash_value2`. We keep this variable because we want to use the same function pointer for three different probe functions.
    - `num_probe` represents the current number of probes. It is the same as `i` on P.22, 25, 28 in the lecture note.
    - `hash_table_size` represents the size of the hash table. It is the same as `m` on P.22, 25, 28 in the lecture note

    To check the correctness of the probing position in other tasks in this lab, you need to use the printing statement: `std::cout << "Probing for the position: " << p << std::endl;` in the lambda function, where `p` is the probe position which will return in the lambda function.

```
~HashTable();
```

Implement the destructor. Hint: You need to free the memory of table_.

Task 2

```
bool add(const std::string &name, int score);
```

In this task, you only need to implement adding new elements to the hash table and do not need to consider rehashing. We will modify this function again for considering rehashing in Task 5. Test cases 1-3 guarantee no need to use rehashing.

Hint: You can implement the function based on this scheme:

1. Calculate `hash_value1` by calling `HashFunction(const std::string &name, int base, int table_size)` where the value of `base` equals `base_1_`.
2. If the `probe_mode_` is `DoubleHash`, calculate `hash_value2` by calling `HashFunction(const std::string &name, int base, int table_size)` where the value of `base` equals `base_2_`. If hash_value2 equals 0, it will probe the same hash cell repeatedly. To avoid this situation, if hash_value2 equals 0, set it to 1 directly. **Note that the formula to calculate `hash_value2` is different from the one on P.30 in lecture note.**
3. Enumerate `num_probe` from `0` to `maximum_probe_num_-1`, call `probe_func_(hash_value1, hash_value2, num_probe, hash_table_size_)` to get the position to probe. If the `state` of the hash cell is `Active`, and the name of the student is the same as the name in the parameter, returning `false`. If the `state` of the hash cell is `Empty` or `Deleted`, insert the data into the hash cell and modify the `state` to `Active` and return `true`.
4. If it still cannot find the position for insertion, return `false`. This statement will be replaced by performing rehashing in Task 5.

You need to understand the meaning of each step before writing functions for the following tasks.

Task 3

```
int search(const std::string &name) const;
```

In this task, you need to implement a function to search for `score` given the student's `name`. If the student can be found, return the `score`. Otherwise, return `-1`.

Hint:

- If you probe a `Deleted` hash cell, continue.
- If you probe an `Active` hash cell, check whether the `name` is the same as the parameter. If yes, return the `score`.
- If you probe an `Empty` hash cell, the student is not in the hash table; return `-1`.
- If the number of probes is equal to the size of the hash table and still cannot find the student, the student is not in the hash table; return `-1`.

Task 4

```
bool remove(const std::string &name);
```

In this task, you need to implement a function to delete a student whose `name` is equal to the input parameter. If the student can be found, delete the data from the hash table and return the `true`. Otherwise, return `false`.

Hint:

- If you probe a `Deleted` hash cell, continue.
- If you probe an `Active` hash cell, check whether the `name` is the same as the parameter. If yes, delete the `student` object, set the `student` pointer to `nullptr` and modify the `state` of the hash cell to `Deleted` and return `true`.
- If you probe an `Empty` hash cell, the student is not in the hash table; return `false`.
- If the number of probes is equal to the size of the hash table and still cannot find the student, the student is not in the hash table; return `false`.

Task 5

```
bool reHash(int rehash_table_size);
```

You need to implement a function to rehash the hash table. The size of the new hash table is `rehash_table_size`. If the rehash is successful, return `true`. Otherwise, return `false`.

Hint:

1. In the beginning, allocate a new space of size `rehash_table_size` for the new hash table.
2. Enumerate all `Active` hash cells in the original hash table from lower to higher index and insert them into the new hash table. If the number of the probes is larger or equal than `maximum_probe_num_` and still cannot find a position to insert, it means the new size is not big enough. Delete the new hash table and return `false`.
3. If all elements are inserted successfully, delete the original hash table, modify `table_` to point to the new hash table and update `hash_table_size_`. Return `true`.

```
bool add(const std::string &name, int score);
```

This function should be modified so that if the number of probes larger or equal than `maximum_probe_num_` and still cannot find a position to insert, call `reHashUntilSuccess()` to do rehashing. After rehashing, you should try to insert the student in the new hash table again. Note that the insertion might still fail. You need to call `reHashUntilSuccess()` again to expand the hash table in this situation again. This process will be repeated until the element can be inserted successfully.

Notes

1. For `bool add(const std::string &name, int score)` and `bool reHash(int rehash_table_size)` , the number of the maximum probes is `maximum_probe_num_`, because we assume that if the number of the probes larger or equal than `maximum_probe_num_` and still cannot find the position to insert, the hashtable is too full, and needs to be rehashed.
2. For `int search(const std::string &name) const` and `bool remove(const std::string &name)` , the number of the maximum probes is `hash_table_size_`, because if we cannot find the student after `hash_table_size_` probes, the student is impossible to appear in the hash table.

Expected Output

The expected output is shown in `test_x_expected_output.txt`, where `x` can be 1 to 6.

Submission and Grading

Deadline: 11:59 ~~AM~~ PM, Nov 20, 2022.

Please submit the following files to ZINC by zipping the following 1 file. ZINC usage instructions can be found here.

- `hash_table.cpp`

There are 7 test cases in total. Test cases 1-3 don't need to rehash. Test cases 4-6 need to rehash. Test case 7 is the same as test case 6, except that test case 7 checks memory leakage.

Notes:

- You may submit your file multiple times, but only the last submission will be graded. **You do NOT get to choose which version we grade.** If you submit after the deadline, a late penalty will be applied according to the submission time of your last submission.
- Submit early to avoid any last-minute problems. Only ZINC submissions will be accepted.
- The ZINC server will be very busy on the last day, especially in the last few hours, so you should expect you would get the grading result report not-very-quickly. However, as long as your submission is successful, we will grade your latest submission with all test cases after the deadline.
- In the grading report, pay attention to various errors reported. For example, **under the "make" section, if you see a red cross, click on the STDERR tab to see the compilation errors.** You must fix those before seeing any program output for the test cases below.
- Make sure you submit the correct file yourself. You can download your file back from ZINC to verify. Again, **we only grade what you uploaded last to ZINC**.

Compilation Requirement

It is **required** that your submissions can be compiled and run successfully in our online auto-grader ZINC. It won't be graded if we cannot compile your work. Therefore, for parts you cannot finish, just put in a dummy implementation so that your whole program can be compiled for ZINC to grade the other parts you have done.