Assignment 6

Question 1

In this question, you are required to implement a hashmap of strings.

The required strategies are:

- Open Addressing: When a hash collision happened, find another available slot to store the key-value pair
- Linear Probing: In finding another available slot, using linear probing; that is to check `slot[current + 1 % capacity]` until found an available slot.
- Dummy Erasing: A problem for open addressing is that remove an key-value pair from the map will result in an empty slot which will affect the linear probing procedure. A common workaround is to make a special mark to the deleted entry such that it can be reused later but the probing procedure should not stop.

Detailed requirements are in the comments of the template code. Please have a careful look.

The required interfaces are

```
    /*
     * return_type operator[] ( const std::string & key ) const
     * -----------------------------------------------------------
     * Get a reference to the value associated with the key if it exists in the
hashmap.
     *
     * - Return `std::nullopt` if the key does not exist
     * - You can use `std::ref ( value )` to create a `std::reference_wrapper`
     *   to value
     * - See https://en.cppreference.com/w/cpp/utility/optional for documents of
`std::optional`
     * - See
https://en.cppreference.com/w/cpp/utility/functional/reference_wrapper for
documents
     *   of `std::reference_wrapper`
     * - See https://en.cppreference.com/w/cpp/utility/functional/ref for
documents of `std::ref`
     */
    return_type operator[] ( const std::string & key ) const;

    /*
     * void insert ( const std::string & key, const std::string& value )
     * ------------------------------------------------------------------
     * Insert a new item to the hashmap.
     *
     * - If key exists, simply replace the value.
     * - Otherwise, if `load_factor ( ) >= MAX_LOAD_FACTOR`, do `rehash ( )`
first
     * - Then use linear probing and open addressing to locate a new bucket for
the key-value pair
     */
    void insert ( const std::string & key, const std::string& value );
```

```
    /*
     * void erase ( const std::string & key )
     * ----------------------------------------------------------------
     * Erase an existing key-value pair
     *
     * - If the key exists, remove the key.
     * - If the key does not exist, do nothing.
     * - If this operation results in an empty slot in the middle, the
     *   handling strategy is to mark the slot as dummy to make sure that
     *   later probing can be done successfully.
     */
    void erase ( const std::string & key );

    /*
     * bool contains ( const std::string & key ) const
     * ----------------------------------------------------------------
     * Check whether the hashmap contains the given key.
     */
    bool contains ( const std::string & key ) const;

    /*
     * bool empty ( ) const
     * ----------------------------------------------------------------
     * Check whether the hashmap is empty or not.
     */
    bool empty ( ) const;


    /*
     * size_t size ( ) const
     * ----------------------------------------------------------------
     * Return the size of the hashmap (number of used buckets)
     */
    size_t size ( ) const;

    /*
     * size_t capacity ( ) const
     * ----------------------------------------------------------------
     * Return the capacity of the hashmap (number of all buckets)
     */
    size_t capacity ( ) const;

    /*
     * void clear ( )
     * ----------------------------------------------------------------
     * Clear all hashmap entries and reset capacity to default.
     */
    void clear ( );

    /*
     * void rehash ( )
     * ----------------------------------------------------------------
     * Grow the hashmap capacity to 3 times (we should have `3 * original`
     * capacity after this operation) and rehash all existing entries.
     */
    void rehash ( );
```

```
    /*
     * float load_factor () const
     * ------------------------------------------------------------------
     * Get the load factor of the current hashmap.
     */
    float load_factor () const;
```

Question 2

In this question, you are required to implement a priority queue using binary heap. You should not use the binary heap related functions from STL.

The required interfaces are

```
    /*
     * Copy Constructor
     * ------------------------------------------------------
     * Copy construct a new priority queue. You should write
     * it correctly on yourself.
     */
    PriorityQueue ( const T& that );


    /*
     * Copy Assignment Operator
     * ------------------------------------------------------
     * Copy Assignment the priority queue. You should write it
     * correctly on yourself.
     */
    PriorityQueue & operator= ( const T& that );

    /*
     * void push ( const T& element )
     * ------------------------------------------------------
     * Push a new element to the priority queue.
     */
    void push ( const T& element );

    /*
     * T pop ( )
     * ------------------------------------------------------
     * Pop the element with the highest priority in the queue.
     * and return it;
     * No need for checking whether the queue is empty.
     */
    T pop ( );

    /*
     * const T& top () const
     * ------------------------------------------------------
     * Return a reference to the element with the highest
     * priority.
     * No need for checking whether the queue is empty.
     */
    const T& top () const;

    /*
```

```
     * bool empty () const
     * ------------------------------------------------------
     * Check whether the queue is empty
     */
    bool empty () const;

    /*
     * void clear ()
     * ------------------------------------------------------
     * Clear all elements in the queue
     */
    void clear ();

    /*
     * size_t size () const
     * ------------------------------------------------------
     * Return the size of the queue.
     */
    size_t size () const;
```

Question 3

In questions 3, you are going to help us complete the implementation of a `Graph` class.

The graph is directed. The basic constructors and structures design are provided.

The graph is organized in `std::unordered_map < std::string, Node > inner`, you can use `inner.at(name)` to visit a specific node.

Each node is something like:

```
using NodePtr = struct Node *;

struct Node {
    std::string name;
    std::vector < NodePtr > edges;

    explicit Node ( std::string name );
};
```

You can access the connected node via

```
for (Node * i : inner.at(name).edges) {
    std::cout << name << " -> " << i->name << std::endl;
}
```

You need to finish the implementation of the following interfaces:

```
 /*
     * std::vector< std::string > dfs_order ( const std::string & root ) const
     * ------------------------------------------------------------------------
     * Using `std::stack` to run DFS.
     * Return the DFS order starting from the root.
     *
     * - The order is recorded when a node is pushed to stack
     * - For edges, please visit them in the order as they are in the vector.
```

```
     * - Already visited nodes will not be pushed to stack again.
     * - For details, see examples.
     */
    std::vector< std::string > dfs_order ( const std::string & root ) const;

    /*
     * std::vector< std::string > bfs_order ( const std::string & root ) const
     * ------------------------------------------------------------------------
     * Using `std::queue` to run BFS.
     * Return the BFS order starting from the root.
     *
     * - The order is recorded when a node is pushed to queue.
     * - For edges, please visit them in the order as they are in the vector.
     * - Already visited nodes will not be pushed to queue again.
     * - For details, see examples.
     */
    std::vector< std::string > bfs_order ( const std::string & root ) const;
```

File Tree

We will deliver an archive with the following structure, when submitting, please also keep the same structure.

```
.
├── lib
|   └── <some files>       # do not modify
├── res
|   └── <some files>         # do not modify
├── Assignment-6.pro      # do not modify
└── src
    ├── testing
    |   └── <some files>     # do not modify
    ├── graph.cpp         # change it
    ├── graph.h           # do not modify
    ├── priorityqueue.cpp   # do not modify
    ├── priorityqueue.h     # change it
    ├── main.cpp            # change it
    ├── stringmap.cpp           # change it
    ├── stringmap.h             # do not modify
```