In this lab, you will implement some applications of symbol tables. You may use `std::map` from STL (see the end of this document for a link to its documentation).

Exercise 1: *Sparse vectors*

A *sparse vector* is a vector in which most elements are zero. To save space, we store only the nonzero elements along with their indices in a symbol table. The memory usage is proportional to the number of nonzero entries, rather than the total size of the vector.

Implement `Svector` class to represent a sparse vector of `double`s with the following methods:

Function	Description
`set(int i, double x)`	Set the i-th entry to x (if $x = 0$, remove the entry if it exists).
`get(int i) const`	Return the i-th entry (if not present, return 0).
`dot(const Svector& that) const`	Return the dot product of `this` vector with `that`.
`norm() const`	Return the Euclidean norm $\|x\| = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$.
`add(const Svector& that) const`	Return the sum of `this` vector and `that`.
`scale(double alpha)`	Multiply `this` vector by the scalar `alpha`.
`print() const`	Print the non-zero entries of vector in the form `{(i1,x1), (i2,x2), ...}`

The operations `set` and `get` should run in $O(\log n)$ time in the worst case, where n is the number of nonzero entries in the vector. Computing the dot product of two vectors should take time proportional to the total number of nonzero entries in both vectors.

Following is a sample usage of the `Svector` class:

```
#include "Svector.h"
int main() {
    Svector v1, v2;

    v1.set(1, 3.0);  v1.set(3, 4.0);  v1.set(10, 5.0);
    std::cout << "v1 = "; v1.print();

    v2.set(1, 2.0);  v2.set(2, 7.0);  v2.set(3, 1.0);
    std::cout << "v2 = "; v2.print();

    std::cout << "v1.get(2) = " << v1.get(2) << " (should be 0)\n";
    std::cout << "v1 . v2 = " << v1.dot(v2) << "\n";
    std::cout << "||v1|| = " << v1.norm() << "\n";

    Svector v3 = v1.add(v2);   std::cout << "v1 + v2 = "; v3.print();
    v1.scale(2.0);             std::cout << "2 * v1 = ";  v1.print();

    // test removing entry by setting to 0
    v1.set(3, 0.0);    std::cout << "After setting v1[3] = 0: "; v1.print();
}
```

Exercise 2: *LRU cache*

An LRU *(least recently used)* cache is a data structure that stores a collection of items with a limited capacity, and evicts the least recently accessed item when the capacity is exceeded.

Design a data structure that supports the following two operations: `access` and `remove`: The `access` operation inserts an item into the data structure if it is not already present, and marks it as the most recently accessed. The `remove` operation deletes and returns the item that was least recently accessed. Both operations should run in $O(\log n)$ time in the worst case, where n is the number of items currently stored in the data structure.

Hint: Maintain the items in order of access using a doubly linked list. Use a symbol table with keys = *items* and values = *their locations in the linked list (i.e., iterators to the corresponding nodes)*. When an item is accessed or removed, update the linked list and the symbol table accordingly:

- When you `access` an element, remove it from its current position (if it exists) and re-insert it at the beginning of the list. Update its entry in the symbol table to point to the new position.
- When you `remove` an element, delete it from the end of the list and remove its entry from the symbol table.

Implement a class `LRU` with the following methods:

Function	**Description**
`access(int item)`	Insert `item` if not already present, or update it as most recently accessed.
`remove()`	Remove and return the least recently accessed item.
`print() const`	Print the items in order of most recent to least recent access.
`contains(int item) const`	Return `true` if `item` is present.
`size() const`	Return the number of items in the cache.
`empty() const`	Return `true` if the cache is empty.

Following is a sample usage of the `LRU` class:

```
#include "LRU.h"
int main() {
    LRU cache;
    cache.access(10);  cache.access(20);  cache.access(30);
    std::cout << "Cache after 10,20,30: "; cache.print(); // Expected: 30 20 10

    cache.access(20); // move 20 to front
    std::cout << "After accessing 20: "; cache.print(); // Expected: 20 30 10

    std::cout << "Remove LRU = " << cache.remove() << std::endl;
    std::cout << "Cache after remove: "; cache.print(); // Expected: 20 30

    std::cout << "Contains 10? " << cache.contains(10) << "\n"; // Expected: 0 (false)
    std::cout << "Size = " << cache.size() << "\n"; // Expected: Size = 2
    std::cout << "Empty = " << cache.empty() << "\n"; // Expected: Empty = 0 (false)
}
```

Exercise 3: *Mutable string*

Create a data type `Mstring` that supports the following operations on a string in $O(\log n)$ time in the worst case, where n is the length of the string:

Function	Description
`get(int i)`	Return the i-th character of the string.
`insert(int i, char c)`	Insert `c` so that it becomes the i-th character of the string.
`remove(int i)`	Delete the i-th character from the string.
`print() const`	Print the entire string.

Hint: Represent the string using a balanced binary search tree (BST), where each node stores a key–value pair: *key* = a real number between 0 and 1, *value* = a character.

The characters of the string are stored in the inorder order of the BST.

- Use the `select(i)` operation on the BST to find the i-th character.
- To insert a character at position i, assign it a key equal to the average of the keys at positions $i-1$ and i.
- To remove a character, simply delete the corresponding key from the BST.

To avoid special cases, you may use two dummy characters at the beginning and end of the string with keys 0 and 1, respectively.

Following is a sample usage of the `Mstring` class:

```
#include "Mstring.h"
int main() {
    Mstring s;

    // Insert characters to form "abcd"
    s.insert(0, 'a');  s.insert(1, 'b');  s.insert(2, 'c');  s.insert(3, 'd');
    std::cout << "String after inserts: "; s.print();  // Expected: abcd

    // Get a character
    std::cout << "s.get(2) = " << s.get(2) << std::endl;  // Expected: c

    // Insert in the middle
    s.insert(2, 'X');
    std::cout << "After insert X at pos 2: "; s.print();  // Expected: abXcd

    // Remove a character
    s.remove(3);
    std::cout << "After remove at pos 3: "; s.print();   // Expected: abXd

    // Remove first and last
    s.remove(0);   s.remove(s.size() - 1);
    std::cout << "After removing first and last: "; s.print();   // Expected: bX
}
```

`std::map` – short summary:

The `map<K,V>` associative container in C++ stores a collection of *key-value* pairs where the keys are unique and and can be accessed in ascending order. It is implemented as red-black tree. It has the following member functions (and more):

Function	Description	Usage example
`insert()`	Inserts a pair of the form `{key,value}` into the map. The key must be unique. Ignore if key is already present in the map.	`erp.insert({"Ali",12345})`
`operator[]`	Returns a reference to the value associated with the given key. If the key is not present, it inserts a pair `{key,x}` where `x` is default value of type `V` and returns a reference to the value `x`.	`cout << erp["Ali"]` `erp["Ali"]=12543`
`find()`	Returns an iterator to the element with the given key. If key is not present in the map, it returns `end()`	`if(erp.find("Zia")==erp.end())` `cout << "Not found"`
`contains()`	Returns true if the map contains the given key (C++20)	`if(!erp.contains("Zia"))` `cout << "Not found"`
`erase()`	Removes the element with the given key	`erp.erase("Ali")`
`size()`	Returns the number of elements in the map	`erp.size()`
`empty()`	Returns true if the map is empty	`erp.empty()`

In the following example, we use `map<string,int>` to store erp id of different students. The keys are the names of the students and the values are their erp ids.

```
#include <iostream>
#include <string>
#include <map>
using std::cout, std::map, std::string;
int main() {
    map<string, int> erp;
    erp["Ali"] = 12345;   // insert pair ("Ali", 12345)
    erp["Ali"] = 13901;   // update value of key "Ali" to 13901
    erp["Beena"] = 12897; // insert pair ("Beena", 12897)
    erp.insert({"Beena", 14593}); // ignore, key "Beena" is already present
    erp.insert({"Chand", 13412}); // insert pair ("Chand", 13412)
    cout << erp["Ali"] << " " << erp["Beena"] << " " << erp["Chand"] "\n"; // // prints 13901 12897 13412

    if(erp.find("Dua")==erp.end())  cout << "Dua not present\n"; // "Dua" is not present
    erp.erase("Beena");     // remove "Beena"

    // Iterate using C++17
    for (auto& [key, value] : erp)
        std::cout << '[' << key << "] = " << value << "; ";
    // C++11 alternative:
    //  for (auto& n : erp)    // p is std::pair<const string, int>
    //      std::cout << n.first << " = " << n.second << "; ";
}
```

`std::list` – short summary:

The `list<T>` container in C++ is a doubly linked list that allows for efficient insertions and deletions from both ends as well as from the middle of the list. Unlike vectors, iterators to elements in a list remain valid even after insertions or deletions of other elements. It has the following member functions (and more):

Function	Description	Usage example
`push_back()`, `pop_back()`	Inserts/remove an element at the end of the list.	`lst.push_back(10)`
`push_front()`, `pop_front()`	Inserts/remove an element at the beginning of the list.	`lst.push_front(20)`
`begin()`, `end()`	Returns an iterator to the first / one-past-the-last element of the list.	`auto it = lst.begin()`
`erase()`	Removes the element at the position pointed to by the given iterator.	`lst.erase(it)`

Following is a sample usage of the `list<int>` class:

```
#include <iostream>
#include <list>
using std::cout, std::list;

int main() {
    list<int> lst;
    lst.push_back(10);   lst.push_back(20);   lst.push_back(30);
    lst.push_front(5);   lst.push_front(1);
    // List now contains: 1, 5, 10, 20, 30

    cout << "List contents: ";
    for (int x : lst)   // iterate using range-based for loop
        cout << x << " ";
    cout << "\n";

    auto it = lst.begin();  ++it; ++it; // iterator to 3rd element (10)
    lst.erase(it); // remove the element at iterator it (i.e., remove 10)

    cout << "After erasing 3rd element: ";
    for (int x : lst)
        cout << x << " ";
    cout << "\n";
}
```