#include <iostream>

#include <vector>

using namespace std;

namespace linear\_probing {

using Entry = struct Entry;

void add(int key);

int notPresent=0;

std::vector<Entry> table;

int totalSize;

int tomb = -1;

int size;

/\*\* Node object that holds key \*/

struct Entry {

explicit Entry(int key = notPresent) : key(key) {} ///< constructor

int key; ///< key value

};

/\*\*

\* @brief Hash a key. Uses the STL library's `std::hash()` function.

\* @param key value to hash

\* @return hash value of the key

\*/

size\_t hashFxn(int key) {

std::hash<int> hash;

return hash(key);

}

/\*\* Performs linear probing to resolve collisions

\* @param key key value to hash

\* @return hash value of the key

\*/

int linearProbe(int key, bool searching) {

int i = 0;

Entry entry;

do {

int index = ((key + i) % totalSize);

//if (searching) {

if (entry.key == notPresent) {

table[index].key=key;

cout<<table[0].key<<table[1].key<<table[2].key;

return 0;

}

else

i++;

} while (entry.key != notPresent);

return notPresent;

}

/\*\* Finds empty spot

\* @param entry instance to check in

\* @param key key value to hash

\* @return hash value of the key

\*/

/\*\* Looks for a matching key

\* @param entry instance to check in

\* @param key key value to hash

\* @return hash value of the key

\*/

void display() {

for (int i = 0; i < totalSize; i++) {

cout << " ";

cout << table[i].key;

cout << " ";

}

}

/\*\* Rehashes the table into a bigger table

\* @returns None

\*/

/\*\* Adds entry using linear probing. Checks for load factor here

\* @param key key value to hash and add

\*/

void add(int key) {

int index = linearProbe(key, false);

cout<<"index="<<index;

table[index].key = key;

// Load factor greater than 0.5 causes resizing

}

/\*\* Removes key. Leaves tombstone upon removal.

\* @param key key value to hash and remove

\*/

void remove(int key) {

int index = linearProbe(key, true);

if (index == notPresent) {

cout << "key not found" << endl;

}

cout << "Removal Successful, leaving tomb" << endl;

table[index].key = tomb;

size--;

}

/\*\* Information about the adding process

\* @param key key value to hash and add

\*/

void addInfo(int key) {

add(key);

cout << "New table: ";

//display();

}

/\*\* Information about removal process

\* @param key key value to hash and remove

\*/

void removalInfo(int key) {

cout << "Initial table: ";

display();

cout << endl;

cout << "hash of " << key << " is " << hashFxn(key) << " % "

<< totalSize << " == " << hashFxn(key) % totalSize;

cout << endl;

remove(key);

cout << "New table: ";

display();

}

} // namespace linear\_probing

using linear\_probing::Entry;

using linear\_probing::table;

using linear\_probing::totalSize;

int main() {

int cmd = 0, hash = 0, key = 0;

cout << "Enter the initial size of Hash Table. = ";

cin >> totalSize;

table = std::vector<Entry>(totalSize);

bool loop = true;

while (loop) {

cout << endl;

cout << "PLEASE CHOOSE -" << endl;

cout << "1. Add key. (Numeric only)" << endl;

cout << "2. Remove key." << endl;

cout << "5. Display Hash table." << endl;

cout << "6. Exit." << endl;

cin >> cmd;

switch (cmd) {

case 1:

cout << "Enter key to add = ";

cin >> key;

linear\_probing::addInfo(key);

break;

case 2:

cout << "Enter key to remove = ";

cin >> key;

linear\_probing::removalInfo(key);

break;

case 3:

linear\_probing::display();

break;

default:

loop = false;

break;

}

cout << endl;

}

return 0;

}