**Hashtable Practice**

Create a project using the ChainedHashtable.h code given below, or use the following repl.it link. <https://repl.it/@ringodarius/ChainedHashtable> Fork the project into your own account. Use the code to answer the question below.

**Basic understanding**

1. Observe the declaration of the htable variable in the protected section of the ChainedHashtable class. This is a list of LinkedList header nodes. Note that there is the use of unique\_ptr and shared\_ptr. Is the LinkedList header node unique or shared? Is the list of Nodes unique or shared? **The LinkedList header node is shared, but the list of Nodes is unique.**
2. Observe the getHash method. What is the algorithm used to enter the value into the table? **Using a modulo operator between the max/capacity and the value itself.**
3. Take a look at the constructor. What is the default capacity of the hashtable? **21**
4. Observe the implementation of the insert method. What would the value h be if the user were inserting the value 10? (assume the default capacity of the hashtable) **It would be 10, as 10%21 is still 10.**
5. What gets created if that particular element does not exist? **A front or new pointer to a new list gets created.**
6. What would be the positions of the following values? 18, 21, 25; **4, 0, 4;**
7. Create a program to insert those values into the hashtable. The display the hashtable to check your answers.
8. Observe the insert method again. What happens when there is a collision when the LinkedList header already exists? **It adds the value to the linked list at that value.**
9. Does the new item get placed before or after the item in the LinkedList header? **It gets placed before.**
10. Where would the value 31 be placed in the hashtable? How about 42, 52, 60? **31 would be placed in the 3rd list. 42 is the 0th list. The 52 is the 3rd list. 60 the 4th list.**
11. Add those values to your hashtable and run the program to check your answers.

**Contains/Delete**

1. Observe the contains() method. What is used to find the LinkedListheader where the item is contained? **They hash the data to get the table.**
2. What happens when the item is not in that particular list? Why would it not need to search in the rest of the array? **The function returns false, being it is not there. Due to the insert function, there is no way for the data to reach a different array besides being hashed and put into the one checked.**
3. Observe the remove() method. What pointers are updated if the item to be removed is the head of the list? **If the item to be removed is the head of the list, then the pointer that leads to the list itself is changed to be the first node.**
4. What pointers are updated when the item to be removed is in the middle of the list? **The pointer previous to the item to be removed is pointed to the one after the one to be removed.**

**Why Prime Number?**

1. Create a hashtable with a non-prime number for the capacity by using the following code.

ChainedHashtable<int>ht(128);

for (int i = 0; i < 100; i++) {

ht.insert(rand() % 1000);

}

cout << ht << endl;

1. Observe the following:
   1. About how many positions had collisions? **32**
   2. What was the longest chain of collisions? **3 long**
   3. How many positions had that same amount of collisions? **28 positions had 1 collision , 3 had 2 collisions, 1 had 3 collisions**
2. Change the capacity of the hashtable to the prime number (127) and answer the same questions…
   1. About how many positions had collisions? **24 positions**
   2. What was the longest chain of collisions? **collisions**
   3. How many positions had that same amount of collisions? **21 positions had 1 collision, 3 had 3 collisions.**
3. Was there an advantage to having a capacity of a prime number? Why or why not? **There were less collisions, about a quarter less collisions, and this would help keep the hashmap isolated so that when you find a value you can get the correct one easily.**
4. Change the random number to be rand() % 10 instead of 1000. Run the program. Is this a very efficient use of a function? Why or why not? **No, because there are a lot of unused positions this way.**

**Chained Hashtable Code**

#pragma once

#include <iostream>

#include <iomanip>

#include <memory>

#include <sstream>

using namespace std;

template<class Type>

struct Node {

Node(Type data): data(data), next(nullptr){}

Type data;

shared\_ptr <Node<Type>> next;

};

template<class Type>

class ChainedHashtable;

template<class Type>

ostream& operator<<(ostream& out, const ChainedHashtable<Type> &t);

template<class Type>

class ChainedHashtable {

protected:

int sz; //Number of elements in table

int capacity; //Table size

unique\_ptr<shared\_ptr<Node<Type>>[]> htable; //An Array of LinkedList Node Headers

public:

ChainedHashtable(int capacity = 21) :

sz(0), capacity(capacity), htable(new shared\_ptr<Node<Type>>[capacity]) {}

void insert(Type e); //Inserts the element in the table

bool remove(Type e); //Removes an element from the table

bool contains(Type e); //Returns true if the value exists

bool empty() { return sz == 0; } //Returns true if the list is empty

int size() { return sz; } //Returns the number of elements in the table

int getHash(Type data) { return data % capacity; } //Hash based upon mod division

friend ostream& operator<< <>(ostream& out, const ChainedHashtable<Type> &t); //Displays the Hashtable

};

template<class Type>

void ChainedHashtable<Type>::ChainedHashtable::insert(Type data) {

int h = getHash(data);

if (htable[h]) {

auto temp = make\_shared<Node<Type>>(data);

temp->next = htable[h];

htable[h] = temp;

}

else {

htable[h] = make\_shared<Node<Type>>(data);

}

sz++;

}

template<class Type>

bool ChainedHashtable<Type>::remove(Type data) {

if (empty()) {

throw runtime\_error("Table is empty");

}

int t = getHash(data);

//If the node is the first item in the list

if (htable[t] == nullptr) {

return false;

}

if (htable[t]->data == data) {

htable[t] = htable[t]->next;

sz--;

return true;

}

auto curr = htable[t]->next;

auto prev = htable[t];

while (curr) {

if (curr->data == data) {

prev->next = curr->next;

sz--;

return true;

}

curr = curr->next;

prev = prev-> next;

}

return false;

}

template<class Type>

bool ChainedHashtable<Type>::contains(Type e) {

int t = getHash(data);

auto curr = htable[t];

while (curr) {

if (curr->data == data) {

return true;

}

curr = curr->next;

}

return false;

}

template<class Type>

ostream& operator<<(ostream& out, const ChainedHashtable<Type> &t) {

for (int i = 0; i < t.capacity; i++) {

out << std::setw(8) << i << ": ";

auto curr = t.htable[i];

while (curr) {

out << curr->data << " ";

if (curr->next) out << "-> ";

curr = curr->next;

}

out << endl;

}

return out;

}