```cpp
#include <chrono>
#include <iostream>
#include <random>
#include <vector>
#include "HashChain.hpp"

void TestFind(){
 std::mt19937 prbg;

  for (int n = 1000; n <= 50000; n += 1000) {

    SepHash* hash = new SepHash();
    std::vector<int> seq;
    for (int i = 0; i < n; ++i) {
      std::uniform_int_distribution<int> rand(0, i);

      //int num = rand(prbg);                // Generate a random number
      hash->insert(i);
      //auto iter = linear_search(seq, num); // Find the insertion point
      seq.push_back(i);
    }

    for (int i = n; i < n *2; ++i) {
      std::uniform_int_distribution<int> rand(0, i);

      //int num = rand(prbg);                // Generate a random number
      //hash->insert(i);
      //auto iter = linear_search(seq, num); // Find the insertion point
      seq.push_back(i);
    }

    auto start = std::chrono::system_clock::now();
    for (int f = 0; f < seq.size(); f++){
        int* test = hash->find(f);
    }
    // Get the current system time in nanoseconds.
    auto stop = std::chrono::system_clock::now();

    // Print the number of nanoseconds each test takes.
    std::cout << n << "," << (stop - start).count() << std::endl;
  }

}


int main()
{
  TestFind();
}
```



```cpp
#include <chrono>
#include <iostream>
#include <random>
#include <vector>
#include "HashChain.hpp"


void TestInsert(){
 std::mt19937 prbg;

  for (int n = 1000; n <= 500000; n += 10000) {

    // Get the starting time point. The type is deduced because it's hard
    // to spell (it is std::chrono::system_clock::time_point).
    auto start = std::chrono::system_clock::now();

    // The actual test.
    SepHash* hash = new SepHash();
    //std::vector<int> seq;
    for (int i = 0; i < n; ++i) {
      std::uniform_int_distribution<int> rand(0, i);

      //int num = rand(prbg);                // Generate a random number
      hash->insert(i);
      //auto iter = linear_search(seq, num); // Find the insertion point
      //seq.insert(iter, num);
    }

    // Get the current system time in nanoseconds.
    auto stop = std::chrono::system_clock::now();

    // Print the number of nanoseconds each test takes.
    std::cout << n << "," << (stop - start).count() << std::endl;
  }

}

int main()
{
  TestInsert();
}
```

```
#include "HashChain.hpp"
#include <iostream>

void TestAdd(){
        Bucket bucket;
        bucket.Add(4);
        //std::cout << "yay";
}

void TestBegin(){
        Bucket bucket;
        bucket.Add(4);
        bucket.Add(6);
        auto test = bucket.Begin();
        if(*test != 4){
                std::cout << "TestGet broke" << std::endl;
        }
}

void TestEnd(){
        Bucket bucket;
        bucket.Add(4);
        bucket.Add(6);
        auto test = bucket.End();
        if(*test != 6){
                std::cout << "TestEnd broke" << std::endl;
        }

}

void TestFind(){
        Bucket bucket;
        bucket.Add(4);
        bucket.Add(6);
        int* found = bucket.Find(6);

        if (!found){
                std::cout << "TestEnd broke" << std::endl;
        }

        if (*found != 6){
                std::cout << "TestEnd broke" << std::endl;
        }
}

void BucketTest(){
        TestAdd();
        TestBegin();
        TestEnd();
}

void TestInsert(){
        SepHash table;
        table.insert(5);
        table.insert(10);
        if (*table.Table[5].Begin() != 5){
                 std::cout << "TestInsert broke" << std::endl;
        }

        if (*table.Table[2].Begin() != 10){
                 std::cout << "TestInsert broke" << std::endl;
        }
}

void TestRehash(){
        SepHash table;
        std::vector<int> values = {50, 25, 100, 10, 75, 76, 74, 4, 99};
        for (int i = 0; i < values.size(); i++){
                table.insert(values[i]);
        }

        if (*table.Table[2].Begin() != 50){
                std::cout << "TestRehash broke" << std::endl;
        }

        if (*table.Table[9].Begin() != 25){
                std::cout << "TestRehash broke" << std::endl;
        }
}

void TestCopyConstruct(){
        SepHash table;
        table.insert(5);
        table.insert(10);
        SepHash newTable = SepHash(table);

        if (*newTable.Table[5].Begin() != 5){
                std::cout << "TestCopyConstruct broke" << std::endl;
        }

        if (*newTable.Table[2].Begin() != 10){
                std::cout << "TestCopyConstruct broke" << std::endl;
        }
}

void TestCopyAssign(){
        SepHash table;
        table.insert(5);
        table.insert(10);
        SepHash newTable;
        newTable.insert(50);
        newTable.insert(100);

        newTable = table;

        if (*newTable.Table[5].Begin() != 5){
                std::cout << "TestCopyAssign broke" << std::endl;
        }

        if (*newTable.Table[2].Begin() != 10){
                std::cout << "TestCopyAssign broke" << std::endl;
        }
}

void TestTableFind(){
        SepHash table;
        std::vector<int> values = {50, 25, 100, 10, 75, 76, 74, 4, 99};
        for (int i = 0; i < values.size(); i++){
                table.insert(values[i]);
        }

        int* found = table.find(25);

        if (!found){
                std::cout << "TestEnd broke" << std::endl;
        }

        if (*found != 25){
                std::cout << "TestEnd broke" << std::endl;
        }
}


void SepHashTest(){
        TestInsert();
        TestRehash();
        TestCopyConstruct();
        TestCopyAssign();
```

```
        TestTableFind();
}

int main(){
        //do stuff
        BucketTest();
        SepHashTest();
        return 0;
}
```



```
#include <vector>
#include <list>
#include <iostream>

struct Bucket {
        std::vector<int> List;
        int Size = 0;

        void Add(int val){
                List.push_back(val);
                Size++;
        }

        int* Find(int val){
                int* output;
                for (int i = 0; i < Size; i++){
                        output = &List[i];
                        if (*output == val){
                                return output;
                        }
                }
                return nullptr;
        }


        //I do this so I can play with using a linked list vs a vector without c
hanging the hash table code
        int* Begin(){
                return &List[0];
        }

        int* End(){
                return &List[Size-1];
        }

};

struct SepHash {//a seperate chained hash table
        private:

        int KeyCount;

        int Hash(int key){
                return key % bucket_count();
        }
        public:
        std::vector<Bucket> Table;

        SepHash(){
                KeyCount = 0;
                Bucket emptyBucket = Bucket();
                std::vector<Bucket> newTable(8, emptyBucket);
                Table = newTable;
        }

        SepHash(const SepHash & other){
                KeyCount = other.key_count();
                Table = other.Table;
        }

        SepHash & operator=(const SepHash & other){
                KeyCount = other.key_count();
                Table = other.Table;
        }

        int load() const{
                return key_count() / bucket_count();
        }
```

```
        int key_count() const {
                return KeyCount;
        }

        int bucket_count() const{
                return Table.size();
        }

        void insert(int key){
                int hash = Hash(key);
                if (load() > .75){
                        rehash(bucket_count() * 2);
                }
                Table[hash].Add(key);
                KeyCount++;
        }

        void rehash(int size){
                Bucket emptyBucket = Bucket();
                std::vector<Bucket> oldTable = Table;
                //we have to do it this way to make the hash function work right
                KeyCount = 0;
                std::vector<Bucket> newTable(size, emptyBucket);
                Table = newTable;
                for (int i = 0; i < oldTable.size(); i++){
                        Bucket thisBucket = oldTable[i];
                        for (int i = 0; i < thisBucket.Size; i++){
                                //insert(thisBucket.List[i]);
                                int newHash = Hash(thisBucket.List[i]);
                                Table[newHash].Add(thisBucket.List[i]);
                                KeyCount++;
                        }
                }
        }

        int* find(int val){
                int hash = Hash(val);
                return Table[hash].Find(val);
        }

};
```



```
#include "HashOpen.hpp"
#include <iostream>

void TestInsert(){
        OpenHash table;
        table.insert(5);
        table.insert(10);
        if (table.Table[5] != 5){
                std::cout << "TestInsert broke" << std::endl;
        }

        if (table.Table[2] != 10){
                std::cout << "TestInsert broke" << std::endl;
        }
}

void TestRehash(){
        OpenHash table;
        std::vector<int> values = {50, 25, 100, 10, 75, 76, 74, 4, 99};
        for (int i = 0; i < values.size(); i++){
                table.insert(values[i]);
        }

        if (table.Table[2] != 50){
                std::cout << "TestRehash broke" << std::endl;
        }

        if (table.Table[9] != 25){
                std::cout << "TestRehash broke" << std::endl;
        }
}

void TestCopyConstruct(){
        OpenHash table;
        table.insert(5);
        table.insert(10);
        OpenHash newTable = OpenHash(table);

        if (newTable.Table[5] != 5){
                std::cout << "TestCopyConstruct broke" << std::endl;
        }

        if (newTable.Table[2] != 10){
                std::cout << "TestCopyConstruct broke" << std::endl;
        }
}

void TestCopyAssign(){
        OpenHash table;
        table.insert(5);
        table.insert(10);
        OpenHash newTable;
        newTable.insert(50);
        newTable.insert(100);

        newTable = table;

        if (newTable.Table[5] != 5){
                std::cout << "TestCopyAssign broke" << std::endl;
        }

        if (newTable.Table[2] != 10){
                std::cout << "TestCopyAssign broke" << std::endl;
        }
}


void TestTableFind(){
        OpenHash table;
```

```
        std::vector<int> values = {50, 25, 100, 10, 75, 76, 74, 4, 99};
        for (int i = 0; i < values.size(); i++){
                table.insert(values[i]);
        }

        int* found = table.find(25);

        if (!found){
                std::cout << "TestEnd broke" << std::endl;
        }

        if (*found != 25){
                std::cout << "TestEnd broke" << std::endl;
        }
}


void OpenHashTest(){
        TestInsert();
        TestRehash();
        TestCopyConstruct();
        TestCopyAssign();
        TestTableFind();
}

int main(){
        //do stuff
        //BucketTest();
        OpenHashTest();
        return 0;
}
```



```
#include <vector>
#include <list>
#include <iostream>

struct OpenHash {//a seperate chained hash table
        private:

        int KeyCount;

        int Hash(int key){
                return key % int_count();
        }
        public:
        std::vector<int> Table;

        OpenHash(){
                KeyCount = 0;
                std::vector<int> newTable(8);
                Table = newTable;
        }

        OpenHash(const OpenHash & other){
                KeyCount = other.key_count();
                Table = other.Table;
        }

        OpenHash & operator=(const OpenHash & other){
                KeyCount = other.key_count();
                Table = other.Table;
        }

        int load() const{
                return key_count() / int_count();
        }

        int key_count() const {
                return KeyCount;
        }

        int int_count() const{
                return Table.size();
        }

        void insert(int key){
                if (load() > .75){
                        rehash(int_count() * 2);
                }
                DumbInsert(key);
        }

        void DumbInsert(int key){
                int hash = Hash(key);
                if (Table[hash]){
                        int counter = hash +1;
                        while (counter != hash){
                                if (!Table[counter]){
                                        Table[counter] = key;
                                        break;
                                }
                                counter++;

                                if (counter == Table.size()){
                                        counter = 0;
                                }
                        }
                }else{
                        Table[hash] = key;
                }
                KeyCount++;
```

```
        }

        void rehash(int size){
                int emptyint = int();
                std::vector<int> oldTable = Table;
                //we have to do it this way to make the hash function work right
                KeyCount = 0;
                std::vector<int> newTable(size, emptyint);
                Table = newTable;
                for (int i = 0; i < oldTable.size(); i++){
                        DumbInsert(oldTable[i]);//lest us bypass rehashing

                }
        }

        int* find(int val){
                int hash = Hash(val);
                int* output = &Table[hash];
                if (!output){
                        return nullptr;
                }else{
                        while (*output != val){
                                if (!Table[hash]){
                                        return nullptr;
                                }
                                hash++;
                                output = &Table[hash];
                                if (hash == Table.size()){
                                        hash = 0;
                                }
                        }
                        return output;
                }
        }

};
```



```
#include <chrono>
#include <iostream>
#include <random>
#include <vector>
#include "HashOpen.hpp"

void TestFind(){
 std::mt19937 prbg;

  for (int n = 1000; n <= 50000; n += 1000) {

    OpenHash* hash = new OpenHash();
    std::vector<int> seq;
    for (int i = 0; i < n; ++i) {
      std::uniform_int_distribution<int> rand(0, i);

      //int num = rand(prbg);                // Generate a random number
      hash->insert(i);
      //auto iter = linear_search(seq, num); // Find the insertion point
      seq.push_back(i);
    }

    for (int i = n; i < n *2; ++i) {
      std::uniform_int_distribution<int> rand(0, i);

      //int num = rand(prbg);                // Generate a random number
      //hash->insert(i);
      //auto iter = linear_search(seq, num); // Find the insertion point
      seq.push_back(i);
    }

    auto start = std::chrono::system_clock::now();
    for (int f = 0; f < seq.size(); f++){
        int* test = hash->find(f);
    }
    // Get the current system time in nanoseconds.
    auto stop = std::chrono::system_clock::now();

    // Print the number of nanoseconds each test takes.
    std::cout << n << "," << (stop - start).count() << std::endl;
  }

}


int main()
{
  TestFind();
}
```

```cpp
#include <chrono>
#include <iostream>
#include <random>
#include <vector>
#include "HashOpen.hpp"


void TestInsert(){
 std::mt19937 prbg;

  for (int n = 1000; n <= 500000; n += 10000) {

    // Get the starting time point. The type is deduced because it's hard
    // to spell (it is std::chrono::system_clock::time_point).
    auto start = std::chrono::system_clock::now();

    // The actual test.
    OpenHash* hash = new OpenHash();
    //std::vector<int> seq;
    for (int i = 0; i < n; ++i) {
      std::uniform_int_distribution<int> rand(0, i);

      //int num = rand(prbg);                // Generate a random number
      hash->insert(i);
      //auto iter = linear_search(seq, num); // Find the insertion point
      //seq.insert(iter, num);
    }

    // Get the current system time in nanoseconds.
    auto stop = std::chrono::system_clock::now();

    // Print the number of nanoseconds each test takes.
    std::cout << n << "," << (stop - start).count() << std::endl;
  }

}

int main()
{
  TestInsert();
}
```