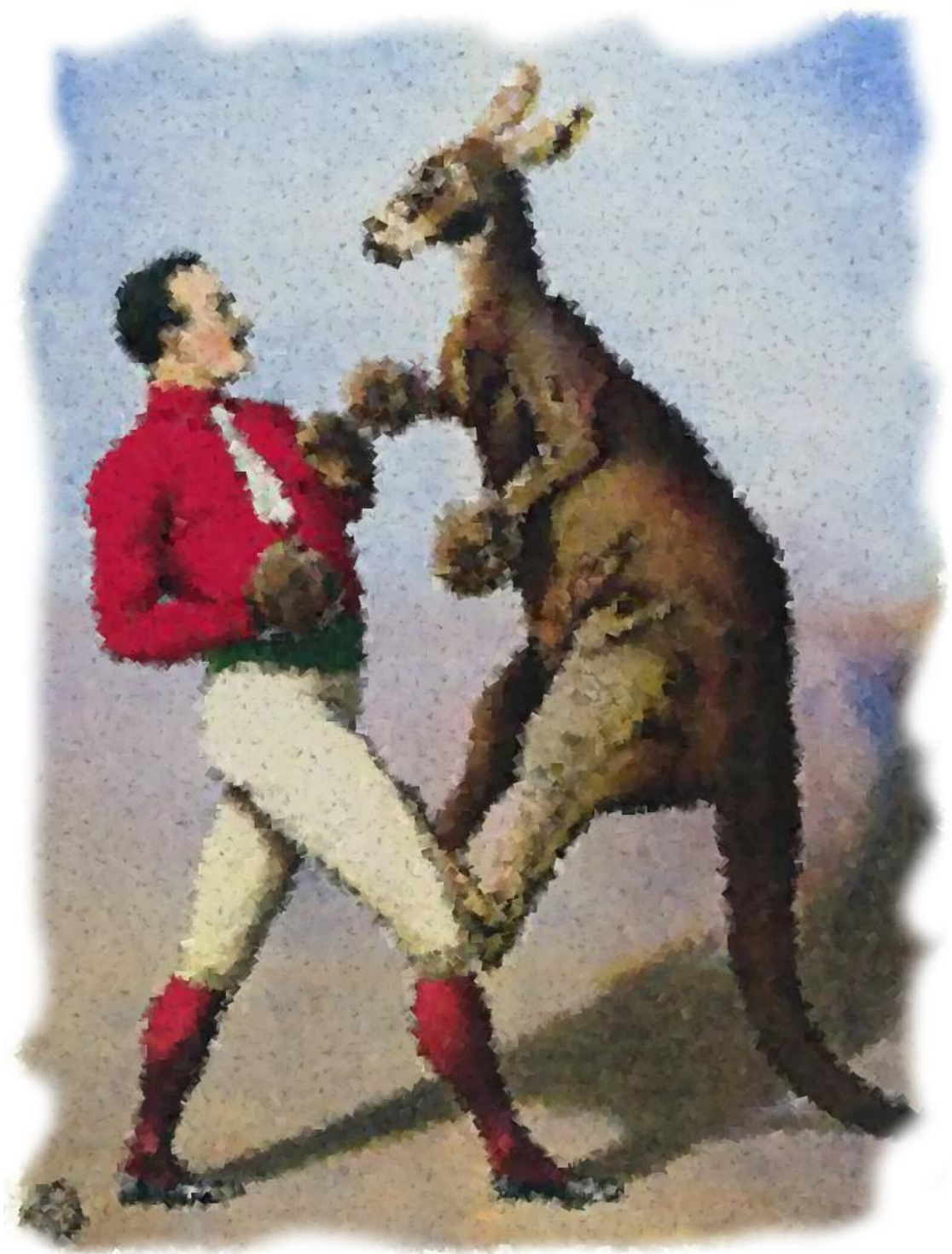


*I'm a Kingaroo! I own dis ring! I hop to places never seen.
Who be you? And what dis thing? Stops for spaces in-between?*



I hoppet and then I boppit

How to Hop in Hash Tables

previously by Michael Locuff

In this cool and easy quest, the first of its kind, you get to play with two kinds of probing techniques in hash tables. The first one does linear probing for a vacant cell and the second does quadratic probing.

Sounds scary? Well, it don't have to be. After you're done reading the reference material in Canvas (Locuff's notes), your text, or other reliable sources, you'll see there's nothing to fear about these two new beasts.

In fact, this quest turned out to be so easy that I got worried if you folks will start complaining. I asked the Quest Master. He say

"Well Then. Turn off the lights."

"What a kewl idea! I much much likes."

From this quest on, you're gonna have to learn to fly with your own inner lights. No more feedback or diagnostic debug output from the site. You'll have to train your good sense to figure out what's right to implement where. Maybe I'll speak up just a liiiiittle bit in this one. But I better learn how to pipe down asap.

'Cuz in a few weeks, you're gonna be flyin on yer own and you can't rely on me to stick around for like ever.



Overview of Probing in Hash Tables

Hash tables¹ are the backing stores for your dictionary objects, like in Python or JS. I'm not gonna review the idea behind hash tables here. Hit your favorite reference material. Or you can simply watch a cool and laid-back discussion we had about it in our CS2A class: Starring a few of our very own: Jack Morgan and Chayan Tronson in lead roles, recorded and produced by another of our own, Darshan Patel.

https://www.youtube.com/watch?v=uGLm3CCt_TA&t=34m24s

I hope it brings back good memories.

¹ What do stilts say about hash tables?

In this quest, you'll implement two different ways to overcome the hash table collision problem. By *collision* I mean the situation when two items hash and index to the same value. Keep in mind that a hash table typically has multiple items with the same hash because the hash function is many-to-one. When you ask for a particular record, you can imagine the equivalent of a (likely linear) search happening for the desired key over all keys with the same hash as yours. (Is this always right? Discuss why. Or why not. Or when not.)

There are many ways to tackle the collision problem. For example, you could make a vector, linked list, or tree of all items with the same value, but different on a secondary key, etc. You can get as fancy as you want.²

But most of us would try a few simple techniques first.

If an item indexes into a location that is already occupied, then simply move it to the next available location. See Figure 1.

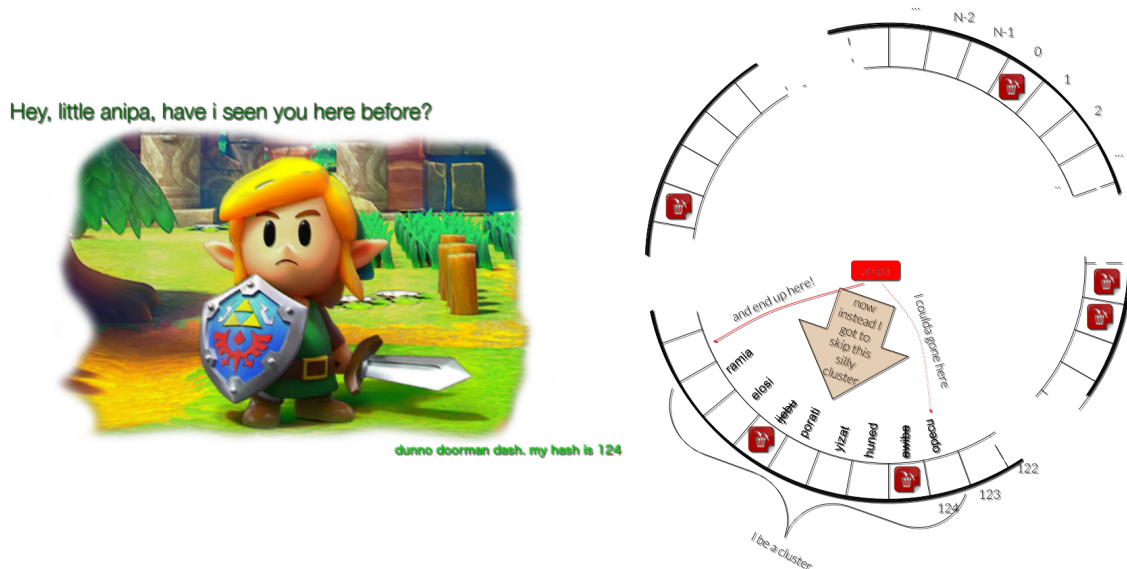


Figure 1. Probing for a vacant cell (Assume $N \gg 124$)

This technique is called *linear probing*. This is the first way in which you will try and solve the collision problem.

You must implement a class called `Hash_Table_LP`. When trying to search for an item in it, you must essentially probe for it linearly starting at the item's indexed *hash*³ in your array. When trying to insert a new element, this is how you eventually end up at a vacant cell in the array after having looked over all non-vacant cells that didn't match.

² Is this how they do it in Hanoi?

³ How is this like an ant?

Figure 2 shows a fuzzy photo of the Hash_Table_LP template class.

```
template <typename T>
class Hash_Table_LP {
protected:
    struct Entry {
        T _data; // payload
        enum STATE { ACTIVE, VACANT, DELETED } _state;
        Entry(const T &d = T(), STATE st = VACANT) : _data(d), _state(st) {}
    };

    static const size_t DEFAULT_INIT_CAPACITY = 3; // first odd prime. (Don't change)
    vector<Entry> _elems;
    size_t _size; // doesn't count deleted elems
    size_t _num_non_vacant_cells; // does
    float _max_load_factor;

    virtual size_t _get_hash_modulus(const T &item) const; // uses Hash(item), ext.
    virtual void _rehash();

    // Most common overrides
    virtual bool set_max_load_factor(float x);
    virtual float _get_biggest_allowed_max_load_factor() const;
    virtual size_t _find_pos(const T &item) const;
    virtual void _grow_capacity();

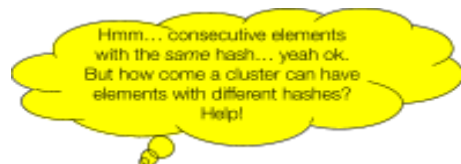
public:
    Hash_Table_LP(size_t n = DEFAULT_INIT_CAPACITY);
    size_t get_size() const { return _size; }
    bool is_empty() const { return _size == 0; }
    bool contains(const T &item) const;
    T &find(const T &item);
    bool clear();
    bool insert(const T &item);
    bool remove(const T &item);

    class Table_full_exception : public exception {
    public: const string to_string() const throw() { return string("Table full exception"); }
    };
    class Not_found_exception : public exception {
    public: const string to_string() const throw() { return string("Not found exception"); }
    };

    friend class Tests;
```

Figure 2. The Hash_Table_LP Class

As you may have been able to guess already, this linear probing technique is susceptible to the formation of clusters. A cluster is a collection of consecutive array elements that may or may not have the same hash. Once you enter a cluster, you are doomed to repeated collisions until you break out of it.



Just like a COVID-19 hotspot, the likelihood that a cluster will grow is directly related to its size. The bigger the cluster, the more likely that it's going to get even bigger (without proper cluster

distancing measures). Why? Hit our [sub](#) and discuss the surprisingly efficient way in which a cluster grows (but please don't bring covid into this any more. Hash Table clusters are interesting enough by themselves).

How do we keep the cluster sizes optimally small (and thus better scattered)? One way is to make sure we always maintain sufficient free space in the table, which lets us implement effective cluster distancing measures.

So we deliberately introduce some redundancy into our data structure to avoid worst case behaviors. It's kinda like underbooking your cruise seats so you never have to make a potential passenger wait endlessly while you look for a free room.

As a consequence of this redundancy, our hash table backing data stores (vectors) will no longer have 100% occupancy rates. Some fraction of cells in the vectors will always have to be vacant. This is the overhead of the hash table.

Load Factor

When your hash table has relatively few elements, then the fraction of its vacant cells is high. Incoming elements get inserted without too much effort. You can say that the load on the hash table is low because it's not really doing a great deal of work.

As it gets used, you may insert and delete elements, and clusters begin to grow. Note that a cluster, once formed, is permanent and never shrinks until a rehash event. No cell within a cluster is a vacant cell, and a cell, once marked non-VACANT, never gets marked VACANT again. And so the hash table's load can only increase with use.

This lets us define a property called the *load factor*. This is the ratio of non vacant cells to the actual capacity of the backing data store. Since the best programmers also tend to be the *most lazy*⁴, this is the quantity we want to control from getting way outta hand. Any time an insert operation looks like the next one's gonna take a lot of work, we'll simply rehash the whole table and bring the load factor down again.

How much? Let's say you decide that the maximum load your employees can handle while doing a decent job is 75%. Then just before the rehash, your table would have been at 75% load using a backing store of N cells. After doubling the size of the array, you would have a backing store of



⁴ If you're just ordinary lazy, you don't wanna work if you don't gotta. If you're like extraordinary lazy, you don't want anyone to work if they don't gotta. But if you're like... the *most lazy*, then you don't even want dem machines to work if they don't hafta.

2N cells, but still only 0.75N elements. Thus the load factor would go from 0.75 to 0.375 (half the value).

As it happens, 0.75 is the default maximum load factor you should allow users to set in your linear probing hash tables.⁵ You will give your users the ability to set their own maximum load factors to any number they want as long as that number is *reasonable* ($0 < n \leq 0.75$). Like a black Ford.

To incorporate this into your implementation, calculate the load factor in your `insert()` method (after inserting). If the load factor of your hash table has gotten way too big (as big as your `_max_load_factor`), you would say (Maybe to yourself):

"Ok. That's it. Things are getting too waaay close for comfort. I gotta breathe man!"

At this time you must `rehash()` your table by growing its capacity to twice⁶ its previous size and reinserting all currently active elements into your new backing store.

To minimize the likelihood of your floating point result being different from mine, try to model your load-factor calculation and comparison like I do in Figure 3.

```
if (_num_non_vacant_cells > _elems.size() * _max_load_factor)
    _rehash();
```

Figure 3. Checking if the load factor > `_max_load_factor`



Hash_Table_LP Details

Obviously, not all of the detail below is reward-worthy.⁷

`_size` and `_num_non_vacant_cells`

`_size` is what users of your class think to be the number of elements actually in the table. Even though they both start off the same, the latter does not get adjusted upon a delete.

`_get_biggest_allowed_maximum_load_factor`

This private method should simply return the floating point value 0.75. In subclasses of `Hash_Table_LP`, you'll have the opportunity to override this virtual function to return different thresholds if you want.

the constructor

It not only sizes the backing data store (`_elems`), but also initializes the member variables.

⁵ Hit the reference material or our sub to talk about the derivation of this optimal value. Why 0.75?

⁶ Why double? And is it always double? Exactly?

⁷ But you still gotta try and do everything to make the customer a happy chappie. Make 'em yell "Hooray" in ecstasy.

It should also set the table's max load factor to the optimal value, which is also the biggest allowed value.⁸

When you have considerable visibility into the domain in which you might deploy a hash table, you get to make such fine adjustments as deliberately underloading or overloading your table because your budget may dictate more compute/less memory or vice-versa. How's that for fun?

`set_max_load_factor`

As discussed already, it should fail and return false if the given load factor is either non-positive or greater than your biggest allowed max load factor.

`_get_hash_modulus`

The hash modulus of an item `x` of type `T` is defined as the result of the modulus operation `Hash<T>(X) % _elems.size()`

Sometimes people loosely call this the hash of `x`. No. The functor⁹ `Hash<T>(X)` returns the hash of `X`. The remainder after modding with the size of the backing store is the hash modulus. It is guaranteed to be bounded by the array size.

How is the hash itself calculated? Clearly it's not possible for you to define the `Hash<T>()` method for type values `T` you don't even know yet.

So you don't do it. You're off the hook for this one. A user of your hash table class (that'd be me) will have to define the relevant `Hash<T>` functors for the types they intend to use with your hash table.

For instance, if I want to instantiate a `Song_Entry` hash table, I had better make sure I define

```
template<> size_t Hash<Song_Entry>(const Song_Entry &s) {  
    // return a size_t computed from the key of the s  
}
```

somewhere in my code or the loader will complain about unresolved externals. The above syntax means "Hash is the name of a template function (<>) and following it is a specialization of it for the `Song_Entry` type. A *specialization* is one concrete implementation. Here's another, and one that I actually use:

```
template<> size_t Hash<string>(const string &s) {  
    return hash<string>{}(s)  
}
```

⁸ See https://www.reddit.com/r/cs2c/comments/nhvooolp/find_pos_help/gvzbkxg

⁹ A functor, in case you aren't familiar with it from CS2B, is simply an object of a class that has defined the operator `()`. Yes, c++ lets you override parentheses too. If `My_class` defines `operator ()` and `my_obj` is of type `My_class`, then I'd be able to invoke `my_obj ()` as if it were a function. Cool?

This says that `Hash<string>(string &s)` simply uses a specialization of the library's `std::hash<>` class to calculate the hash of a string using an anonymous functor of type `hash<string>`.

In your testing, you will have to define your own hash methods like you see above. But you shouldn't submit Hash specializations for the `int` and `string` data types since I'll be using my own.

`grow_capacity`

It should double the size of the backing vector. Since your default `Entry` constructor gives you a VACANT entry, the resize will automatically fill the new cells with VACANT entries.

`rehash`

It should save a copy of the current elements, `grow_capacity()` and then insert all ACTIVE elements in the old array into the new hash table.

`_find_pos`

This is a core private helper method that is used by `insert()`, `remove()`, and `find()`. A few important design decisions influence its algorithm. You have to make the same decisions I made in my reference code in order to pass miniquests. What are the engineering implications of these decisions?

Note that `_find_pos()` scans the backing array linearly for a target element. Its scan is not terminated by ACTIVE cells nor DELETED (deleted) cells, but only by VACANT cells. What happens if the `_elems` array has no more VACANT cells? Note that `_find_pos()` cannot rehash a table (it is, in fact, marked `const`). Since the backing store is circular, there is a real risk of getting stuck like an ant on a mobius strip if someone tries to look for an element that is NOT in the table and the array is full.¹⁰

"Well" you could say, "That's what we have the `_max_load_factor` for. Using it will make sure that the table is never more than 75% occupied."

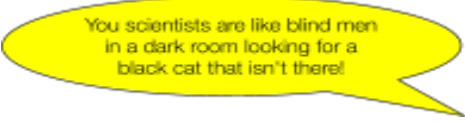
All good. But to be on the safe side, you must code defensively, and let the method immediately fail (return `string::npos`) when asked for the position of an element in a table in which the backing store is completely full (unlikely though it is).

If you put your engineering hats on and survey the situation, you'll find that the following are some of the consequences (some welcome, and some we willingly accept in return for other goods):

1. `_find_pos()` will falsely report that a present element is absent in a backing array that has run out of VACANT cells.

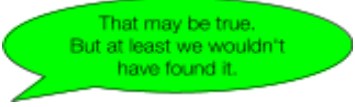
¹⁰ What are some other ways you can handle this situation? Can you get your CS 2B *ant* technique to work here? Why or why not?

2. `_remove()` will fail to remove a present element in a backing array that has run out of VACANT cells.



You scientists are like blind men in a dark room looking for a black cat that isn't there!

Note that this situation is unlikely to happen if your maximum load factor is less than 1. The trade-off we're talking about concerns the programming experience of an exceedingly small number of programmers



That may be true. But at least we wouldn't have found it.

who ignored warnings, messed around with the private internal settings of the hash table (like its max load factor), filled up its backing store *completely* despite suffering horrible performance on the last few inserts, and then asked `_find_pos()` to behave when asked for an element that isn't there.

Our decision implies that we'd return an incorrect result in this extreme case rather than loop forever. Better this silly programmer's code crash than chew up good machine cycles for no good reason.

What's the alternative? Hit our [sub](#). These are the kinds of decisions you will be making in your life as a programmer.

find and contains

`contains()` should return a boolean value, while `find()` should return a reference to the found item and throw the appropriate exception if it isn't able to.¹¹

insert

Use `_find_pos()` to locate either a VACANT cell or the element itself. If you found the element, and it is ACTIVE, then you can simply return false (you didn't have to do anything).

If you found the element but it was marked deleted, then you can simply reset its state, adjust the hash table's size and return.

If you landed on a VACANT cell you will put the element there, and also need to update your count of non-vacant cells so that your load factor calculations will be correct.

remove

This is just like `insert`.

Well, no. But you get the idea.

Use `_find_pos()` to locate the item. If it isn't there to start with, simply return false (as you would if the table is full). If not, all you have to do is to change the state of the entry from ACTIVE to DELETED.

¹¹ What are the pros and cons of returning a const reference rather than a plain reference? Should you do one rather than the other?

Hash_Table_QP Details

If you've completed the implementation of the `Hash_Table_LP` class successfully, this one will likely just fall outta your bag for almost free.



Figure 4 shows a picture of the `Hash_Table_QP` class.

```
template <typename T>
class Hash_Table_QP : public Hash_Table_LP<T> {
public:
    Hash_Table_QP(size_t n = Hash_Table_LP<T>::DEFAULT_INIT_CAPACITY) : Hash_Table_LP<T>(n) {
        this->_max_load_factor = 0.49;
    }

protected:
    virtual float _get_biggest_allowed_max_load_factor() const;
    virtual size_t _find_pos(const T& item) const;
    virtual void _grow_capacity();

    // Private helper
    static size_t _next_prime(size_t n);

    // Don't modify below
    friend class Tests;
};
```

Figure 4. The `Hash_Table_QP` Class

As you can see, it is a subclass of `Hash_Table_LP`. You can leverage almost everything you did before. You only need to override the constructor and three protected methods in the class def. There's also a new private helper method called `_next_prime()`.

When your workhorse methods (`insert`, `remove`, etc.) are invoked by a user, and they in turn invoke any of the three overridden methods, the compiler will make sure the methods attached to the correct class (whether `LP` or `QP`) are used. Note that we're not accessing the overridden methods through pointers. They're just straight method calls on objects that the compiler can handle in the usual way (i.e. no runtime resolution required).

the constructor

Not much to do here. You can chain back to your base class constructor and have it do all the work. However, once that is done, you must reset the `_max_load_factor` member to the maximum permissible value for quadratic probing. To do that you must first override the `_get_biggest_allowed_max_load_factor()` method to return 0.49 (Why 0.49? Hit your refs and discuss in our sub).

`_find_pos()`

Essentially, this is almost the same as the `_find_pos` from your LP table. However, when you have a collision and need to examine another cell, you don't look at the next higher numbered cell, but rather a cell that is *the next perfect square away* from the current index. What does that mean?

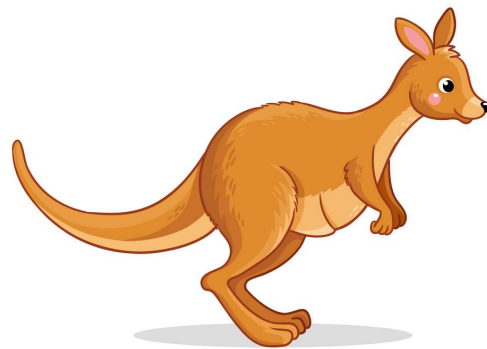
Suppose you're looking for an item X , and the hash modulus of X is K . If you find that `_elems[K]` is occupied, then

- the next location you try will be `_elems[K+1]` (so far the same as in LP)
- If that is also occupied, then you try `_elems[K+4]` ($2^2 = \text{next perfect square after 1}$)
- If that is also occupied, then you try `_elems[K+9]` ($3^2 = 9$)
- etc.

How can you be sure that when you hop around your table like that you'll eventually end up at a location you've never seen?¹² Discuss in our [sub.](#)

Leverage the useful fact that the next perfect square can be found simply by adding the next odd number to the previous one. Don't hit the math library. See:

- $0 + 1 = 1$
- $1 + 3 = 4$
- $4 + 5 = 9$
- $9 + 7 = 16$



Cool 'nuff?

`next_prime()`

If you're here, it means you have probably found a way to ensure that you won't end up hopping endlessly.¹³

`next_prime(size_t n)` should return the least prime number greater than n . You need to test every number $> n$ for primality and return the first one that passes. Use a simple variation of a common test for primality: A number is prime if it leaves a non-zero remainder when divided by any integer > 1 that is less than its square root (This is the only place where you should call a math library function, and BTW, why square root?)

But rather than test for divisibility by every positive integer $< \text{sqrt}(val)$, you can skip a candidate and move on to testing its successor as soon as you find that it's not a prime. Since

¹² And what happens if you do?

¹³ If your table has a load factor < 0.5 and the size of its backing store is a prime number, then you can show that you don't gotta hop endlessly no matter what.

you know that every number (except 1) has a prime factorization,¹⁴ we can come up with an improved algorithm:



A number N is composite (not prime) if:

- it is divisible by 2 or 3
- or it is divisible by some other prime less than \sqrt{N}

"But there's the rub" you say. "How do I find all primes less than \sqrt{N} ? Do I recurse? Do I XYZ?"

"Whoa there, Buster. Stop right there."

Here's a cool way to test for divisibility by all primes less than some number. Exploit the fact that every prime number has at least one neighbor which is divisible by 6. Since a prime > 2 is necessarily odd, both its neighbors must be even. In addition, if one of its neighbors is not divisible by 3, the other must be. Leverage this to come up with this test:

A number N is composite (not prime) if:

- it is divisible by 2 or 3
- or if it is divisible by either $(6k-1)$ or $(6k+1)$ for any positive $k \leq$ a sixth of¹⁵ \sqrt{N}

Together, $(6k+1)$ and $(6k-1)$ cover all known primes ≥ 5 for various positive values of k . Thus, if a number is not divisible by any of them, it can't be divisible by any prime. Of course the less efficient way is to test for divisibility by all primes $\leq \sqrt{N}$. The challenge is to do it with as few as or fewer tests than required by this algorithm.

This technique leveraged information about the two immediate neighbors of the candidate. Can you get more information by looking further? If so, where do you draw the line between spelling out special cases and letting the computer crunch? This kind of reasoning is another kind of seasoning you will be adding to your code sandwiches in the future.

`grow_capacity()`

What can I say? It's the same as its elpie cuzzin, but it got to set the size to the next prime $> 2N$.



¹⁴Curious why? You may want to check out Patrick's Math class.

¹⁵ Wee..elll... That's just loosetongue for a sixth of the ceiling of the root of N . But you get the point. In Salestongue you might say *"Add the Acme test screener to your cart (Improves run time by as much as 500%)"*

Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to <https://quests.nonlinearmedia.org>
2. Enter the secret password for this quest in the box.
3. Drag and drop your source files¹⁶ into the button and press it.

Wait for me to complete my tests and report back (usually a minute or less).

Happy Questin While You're Restin,

&



¹⁶ Hash_Table_LP.h, Hash_Table_QP.h