Dr. Yihsiang Liow (March 12, 2024)

Contents

debug: chap.tex

Chapter 108

Hash tables

debug: hashtable.tex

108.1 Hash tables

A hash table (aka unordered associative array) is very easy to understand, at least for the simple case. Suppose we have an collection of name-height data that I want to keep in a container for later searching (for instance).

Name	Height
Abe	6.5
Tom	5.5
Annie	5.9

The key in this case is the name (of course). The idea that I'm going to talk about has nothing to do with name-height. In general, it works for any so–called key-value table. I'm going to call the above collection (vaguely) **key–value pairs**.

key–value pairs

Now if you think about an array, you can quickly associate values with index values:

Index	Height
0	6.5
1	5.5
2	5.9

The problem of course is that I have names and not index values! Of course I can have another array of names to tie Abe, Annie, Tom to index values:

Index	Name
0	Abe
1	Tom
2	Annie

But that's a pain. Why? Because if you want to search for Annie's height, you would have to search the name array to see that Annie's index is 1, and *then* go to the height array to find Annie's height. If you add another name to the name array, then you might need some kind of organization for fast searching. You might think: "Hey ... let's sort the name array!" The problem is that the index values will then change and I would have to change the height array. Bad idea!

Another way is to *somehow* associate names to index values using some kind of numeric function. In fact, you should know that the characters of the names are already associated with integer values: they have ASCII codes. For instance `A` has ASCII value of 65. Try this C/C++ statement:

```
std::cout << int('A') << '\n';
```

They have been around for a long time: the ASCII codes were designed by IEEE in 1960s. So if I take inspiration from the base 10 representation of integers, I can do this to convert Abe to an integer:

$$\text{Abe} \rightarrow \text{int}(\texttt{A}) \cdot 10^0 + \text{int}(\texttt{b}) \cdot 10^1 + \text{int}(\texttt{e}) \cdot 10^2$$

where int means the ASCII value of the relevant character. The value you get is

```
11145
```

Now what I'm going to do is to create an array of name-height values, say that array has size 10. The index values are of course from 0 to 9. No problem: the above index value, I just do mod 10 and this will give me an integer value from 0 to 9. Right? This means that the name `Abe` will be associated with index

```
11145 % 10 = 5
```

Such a function that takes data and produce an integer value is a hash function. A hash function of course has a finite range. In our case, it's from 0 to 9. (You'll see later that frequently, you want to have the option of expanding the range.) Because we're associating names to index values 0 to 9, we of course want to make sure that names go to different index values. Clearly with only 10 rows in your array, you cannot avoid hashing names to the same index values, especially if you're looking at 1000 names! But say you have only three names. Clearly you do want the three names to hash to different index values. You'll see that a good quality that we want for hash functions is that they should "randomly scatter" values in the range that they can handle.

First let me put Abe and his height into our array, or hash table. The above has hash function hashes Abe to 5. So the array looks like this:

0		
1		
2		
3		
4		
5	Abe	6.5
6		
7		
8		
9		

(The index value is included just for convenience. Of course arrays do not contain index values.) The C++ code would look like this:

```
class NameHeight
{
public:
    std::string name_;
    double height_;
};

class Hashtable
{
public:
    NameHeight table_[10];
};
```

OK. Now let's put Tom into our hash table. The hash value of Tom is

```
12094
```

and when I take mod 10, I get 4.

0		
1		
2		
3		
4	Tom	5.5
5	Abe	6.5
6		
7		
8		
9		

Let's take a pause and see how this container help us find data. So ... if you want to know Abe's height, all you need to do is to use our hash function, compute the hash value of Abe, which would be 5, and you find what you're looking for. Period. Easy, right?

If you don't allow the empty string as a name, you can initializ the names of your hash table to `""`. But what if you allow the empty string as a name? Well you can for instance include a flag in your `NameHeight` class to indicate if there's valid data or not. Something like this:

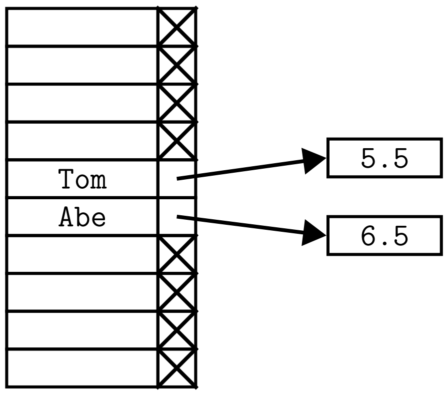
```
class NameHeight
{
public:
    bool available_; // or not_available if you like
    std::string name_;
    double height_;
};

class Hashtable
{
public:
    NameHeight table_[10];
};
```

If so, the hash table now looks like this

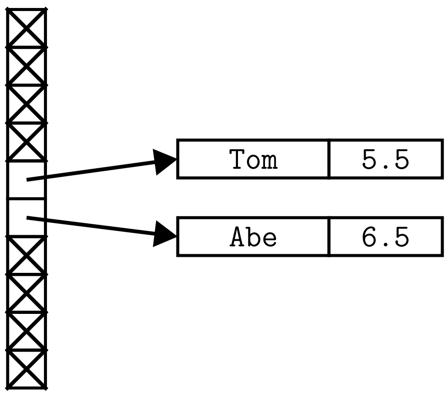
0	true		
1	true		
2	true		
3	true		
4	false	Tom	5.5
5	false	Abe	6.5
6	true		
7	true		
8	true		
9	true		

It's also possible to have a hash table that contains the keys, but the data can be placed in the heap, i.e., the hash table is an array of key-pointer values. In that case, the hash table would take less space – don't forget that a huge array makes memory management difficult because the array of values must be contiguous. And this is a good thing.



In this case, we don't even need the availability flag since a `NULL` pointer would tell us that the row is available.

But ... we can take this one step further ... another option is to have a hash table not be name-pointer rows, but of pointers all altogether. Here's what I mean (in pictures):



At this point, we have two (extreme) options for our hash table:

- An array of key-value pairs, each with an "available" field.
- An array of pointers to key-value pairs, using `NULL` to denote the fact that an array element is available.

Very soon we'll see that there are other options.

108.2 Hash Collision

Recall that I want to put the following into our hash table:

Name	Height
Abe	6.5
Tom	5.5
Annie	5.9

Unfortunately, when I hash `Annie` using our hash function, I also get `5`. What I have here is a hash collision: Two keys are hashed to the same integer value!!! And `Abe` is already at index `5`.

Now what?!?!

108.3 Collision Resolution

One thing I can do is to just go onto the *next* index value. Don't forget that the index values are from 0 to 9. So if I hash to `9` and there's someone at `9`, and I compute the *next* index value to get `10`, I'm outside the array. In other words, I need to mod by 10 (the size of the array).

Computation of new hash values in case of a collision is sometimes called **probing**. There are many different methods for probing. The "*go to next one with mod*" is called **linear probing**.

probing

linear probing

Going back to our example, if I use linear probing, since `Annie` is hashed to `5`, the row occupied by `Abe`, I will put `Annie` at index `6`.

0	`true`		
1	`true`		
2	`true`		
3	`true`		
4	`false`	`Tom`	`5.5`
5	`false`	`Abe`	`6.5`
6	`false`	`Annie`	`5.9`
7	`true`		
8	`true`		
9	`true`		

One step up is **quadratic probes**. Suppose you hash to value h. If there's a collision, you look at $h + 1^2$. If there's another collision, you look at $h + 2^2$. If there's again another collision, you look at $h + 3^2$. Etc. That's it. More generally, instead of using x^2, you can use $ax^2 + bx + c$. For instance, say whenever you have a hash collision, you use the quadratic probe with $a = 1$, $b = 1$, $c = 0$. If you have a collision with value h, then the next value to try is $h + 1^2 + 1^1$. If that's again a collision, you look at $h + 2^2 + 2^1$. And if this is again a collision, you look at $h + 3^2 + 3^1$. Etc.double hashing. Here's how it works. You need two hash functions. Let's say I call them h and h'. Suppose your key if k. The first hash value you look at is $h(k)$. If there's a collision, you look at $h(k) + h'(k)$. If there's another colliion, you look at $h(k) + 2h'(k)$. If there's yet another collision, you look at $h(k) + 3h'(k)$. Etc.

Exercise 108.3.1. Analyze this probing technique. Instead of double hashing with $h(k) + ih'(k)$, use $h(k) + (ai^2 + bi + c)h'(k)$ □

Exercise 108.3.2. Suppose you have a hash function for integer values, say h'. For instance our hash function actually can hash integer values: you simply convert the integer to an integer string. If you have a collision at $h(k)$, look at $h'(h(k))$. Now if you have a collision at $h(k)$, look at $h'(h(k)+1)$. If that's again a collision, look at $h'(h(k)+2)$. Etc. Analyze this probing technique.
□

Exercise 108.3.3. Suppose you have a hash function for integer values, say h'. For instance our hash function actually can hash integer values: you simply convert the integer to an integer string. Now if you have a collision at $h(k)$, look at $h'(h(k))$. If that's again a collision, look at $h'(h'(h(k)))$. Etc. Analyze this probing technique. □

Exercise 108.3.4. Suppose you have a hash function for integer values, say h'. For instance our hash function actually can hash integer values: you simply convert the integer to an integer string. Now if you have a collision at $h(k)$, look at $h'(h(k)+1)$. If that's again a collision, look at $h'(h(k)+2))$. Etc. Analyze this probing technique. □

Exercise 108.3.5. Suppose you have a hash function for integer values, say h'. For instance our hash function actually can hash integer values: you simply convert the integer to an integer string. Now if you have a collision at $h(k)$, look at $h'(h(k)+1)$. If that's again a collision, look at $h'(h'(h(k)+1)+2)$. Etc. Analyze this probing technique. □

108.4 Connection between hash table and buckets

Do you remember bucket sort?

You can and *should* think of hash tables as a kind of bucketing structure. What we have seen so far, each row in the array is a bucket. A hash basically place a key into a bucket. Each bucket has either one or zero key. You can think of probes as a way to overflow from a bucket to another.

(Later we will see that there's another method to let buckets have more than one key.)

108.5 Avoiding/minimizing collisions

In general you want your hash functions to randomly scatter the keys you are interested in to different index values. In other words, you do not want to have too many collisions. The best is not to have collisions at all. Why? Because if there's a collision, then you need to probe.

The technical term to use is that you have your hash functions to behave like a uniform random distribution.

In general, designing a good hash function is not easy. Our hash function:

$$\text{Abe} \rightarrow (\text{int}(\texttt{A}) \cdot 10^0 + \text{int}(\texttt{b}) \cdot 10^1 + \text{int}(\texttt{e}) \cdot 10^2) \mod 10$$

is in fact pretty bad. Why? Because if you mod by 10 like the above, you actually get

$$\text{Abe} \rightarrow (\text{int}(\texttt{A}) \cdot 10^0) \mod 10$$

That's why `Annie` was hashed to the same index as `Abe`.

It's because of this that table/array sizes should be primes. So for instance we could have choosen 13 for a table/array size.

108.6 Insert, Delete, Find

It's clear what you need to do for insert and find. When you're given the key k, you hash to get $h(k)$. If that's available, you put your key-value there. If it's occupied, you probe using whatever method you have decided to use. If your probe reaches the first hash value, you are in trouble. At that point, you should probably throw an exception.

What about delete? You might think that all you need to do is to find it and then mark the row as available. Hang on there ...

This means that you're breaking the chain of collided keys. For instance if you're looking for a key that is hashed to 3 and there are four keys hashed to 3 and resolved using linear probes, then the three keys are at index values 3, 4, 5, 6. If your key is at index 5, then marking that row as available, when you search for the key at 6, your search algorithm is going to stop at 5 and say it's not found. Duh.

There are several pretty obvious options.

OPTION 1. Instead of a flag for each row saying AVAILABLE/NOT-AVAILABLE, you can have a flag that says AVAILABLE/NOT-AVAILABLE/DELETED. In that case the available flag cannot (of course) cannot be a boolean. During a search, if you see a row that is DELETED, you have to continue. Your search ends if either you have found the key at a row that is marked NOT-AVAILABLE or a row that is marked AVAILABLE or you reached your first hash value. Rows which are marked as DELETED are sometimes called tombstones.

The earlier table becomes:

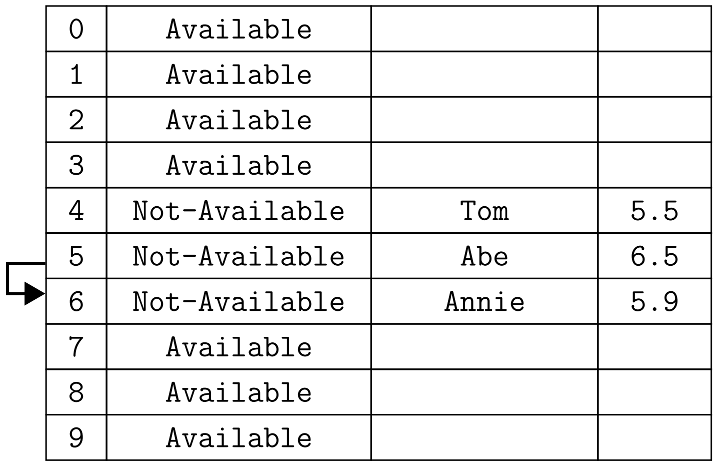
0	Available		
1	Available		
2	Available		
3	Available		
4	Not-Available	Tom	5.5
5	Not-Available	Abe	6.5
6	Not-Available	Annie	5.9
7	Available		
8	Available		
9	Available		

OPTION 2. The second option is to organize the chain of collided keys by

actually moving the keys to overwrite the row to be deleted. For instance say your key is hashed to 3 and there four collided keys are index 3, 4, 5, 6. Say your key is at index 4. Then you have to move the data at index 5 to index 4 and index 6 to index 5 and then mark row 6 as available. If there are not too many colliding keys and long probe sequences, this is usually not that bad. If the probe sequences are getting long, we will have to to a complete restructuring of the hash table – see next section.

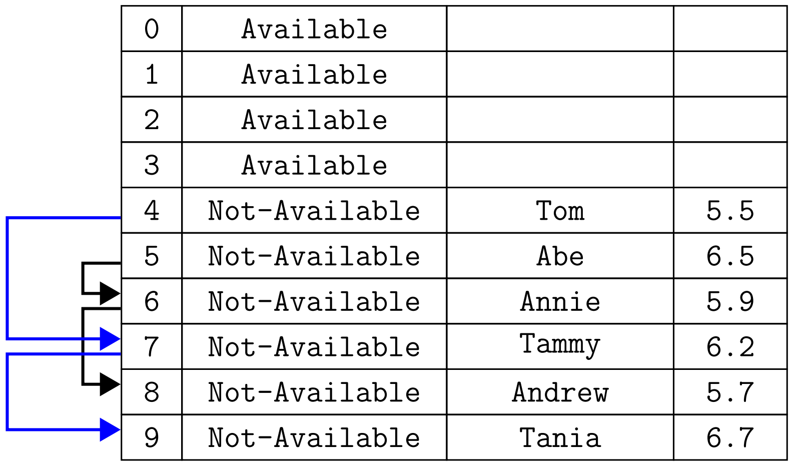
I will stick to OPTION 1.

Going back to out earlier table:



0	`Available`		
1	`Available`		
2	`Available`		
3	`Available`		
4	`Not-Available`	`Tom`	`5.5`
5	`Not-Available`	`Abe`	`6.5`
6	`Not-Available`	`Annie`	`5.9`
7	`Available`		
8	`Available`		
9	`Available`		

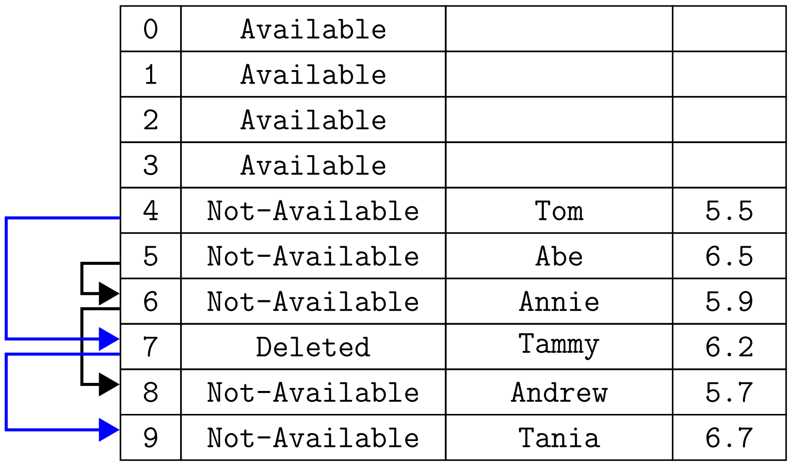
If I add `(Tammy, 6.2)` and then `(Andrew, 5.7)`. and then `(Tania, 6.7)`, this is the resulting table:



0	`Available`		
1	`Available`		
2	`Available`		
3	`Available`		
4	`Not-Available`	`Tom`	`5.5`
5	`Not-Available`	`Abe`	`6.5`
6	`Not-Available`	`Annie`	`5.9`
7	`Not-Available`	`Tammy`	`6.2`
8	`Not-Available`	`Andrew`	`5.7`
9	`Not-Available`	`Tania`	`6.7`

You can think of (Tom, Tammy, Tania) as forming a chain in the hashtable and (Abe, Annie, Andrew) forming another.

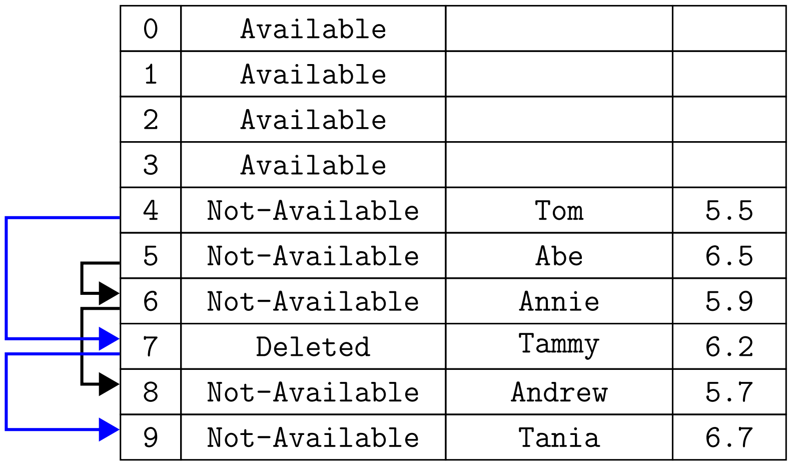
Now if I delete Tammy, the table becomes



0	Available		
1	Available		
2	Available		
3	Available		
4	Not-Available	Tom	5.5
5	Not-Available	Abe	6.5
6	Not-Available	Annie	5.9
7	Deleted	Tammy	6.2
8	Not-Available	Andrew	5.7
9	Not-Available	Tania	6.7

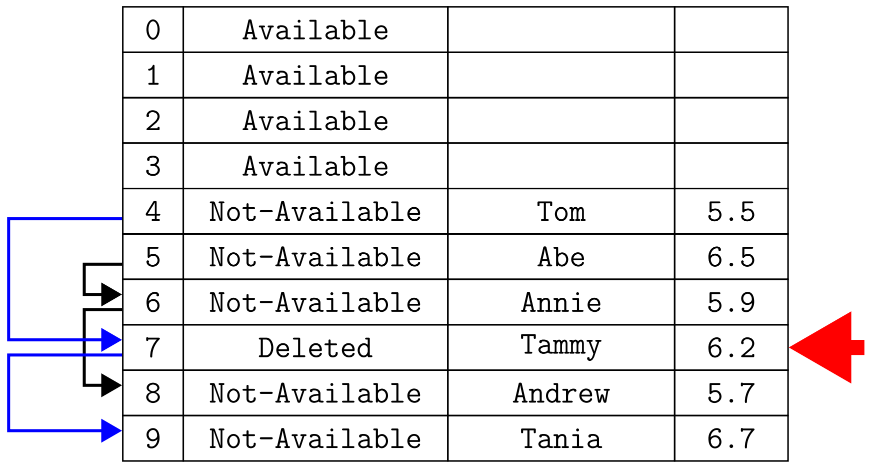
Because of the way in which chains cut into each other's way, during a search, you have to keep probing until you find the key you're looking for or when you hit AVAILABLE (i.e. key is not found).

In the case of insert, you insert at the first row that is DELETED or AVAILABLE if you know for sure that the key does not exist in the table. If you see the key along the way, then you have an error – the key already exists. However if you do *not* know is the key exists, you would have to continue to search until you hit a row that is AVAILABLE. You can (and should) insert the key at the first row that is either DELETED or AVAILABLE. For instance if we insert (Toby, 6.3) into the above table:



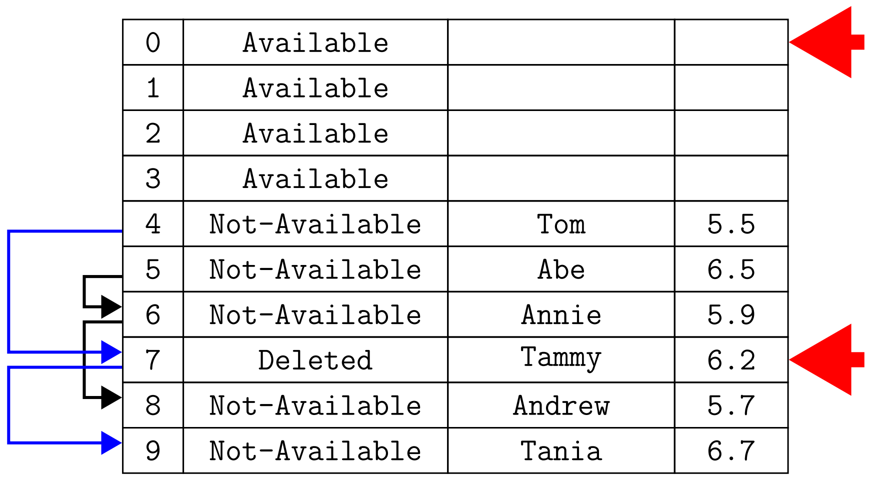
0	Available		
1	Available		
2	Available		
3	Available		
4	Not-Available	Tom	5.5
5	Not-Available	Abe	6.5
6	Not-Available	Annie	5.9
7	Deleted	Tammy	6.2
8	Not-Available	Andrew	5.7
9	Not-Available	Tania	6.7

you would first get the first empty row at index 7 (where the row is `Delete`) – you remember this index:



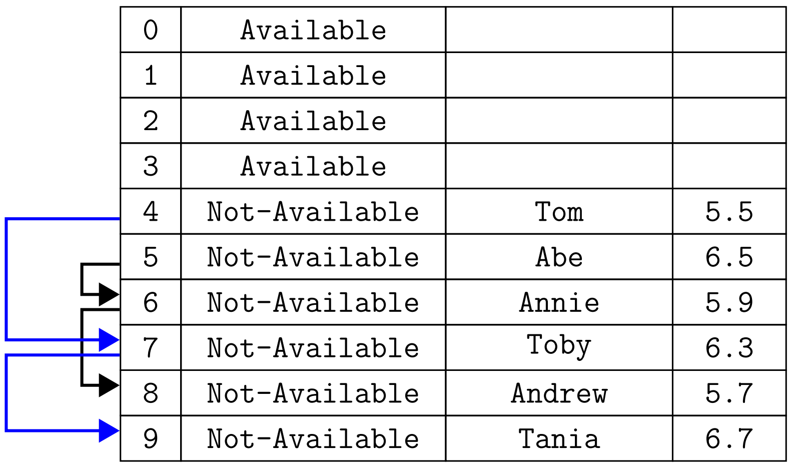
0	Available		
1	Available		
2	Available		
3	Available		
4	Not-Available	Tom	5.5
5	Not-Available	Abe	6.5
6	Not-Available	Annie	5.9
7	Deleted	Tammy	6.2
8	Not-Available	Andrew	5.7
9	Not-Available	Tania	6.7

You would continue until you see the first row that is AVAILABLE – this is index 0:



0	Available		
1	Available		
2	Available		
3	Available		
4	Not-Available	Tom	5.5
5	Not-Available	Abe	6.5
6	Not-Available	Annie	5.9
7	Deleted	Tammy	6.2
8	Not-Available	Andrew	5.7
9	Not-Available	Tania	6.7

Since, up to this point in time, you have not seen Toby, you know that you can safely add (Toby, 6.3) at index 7 and mark that row as NOT-AVAILABLE:



0	Available		
1	Available		
2	Available		
3	Available		
4	Not-Available	Tom	5.5
5	Not-Available	Abe	6.5
6	Not-Available	Annie	5.9
7	Not-Available	Toby	6.3
8	Not-Available	Andrew	5.7
9	Not-Available	Tania	6.7

For delete, this is like search. When the key is found, we mark the row as DELETED.

```
ALGORITHM: HASHTABLE-INSERT
INPUT: hashtable of size n
       hash - hash function
       (key, value) - keyvalue pair to insert

index = -1
compute h = hash(key) % n
while 1:
    if hashtable[h].flag is DELETED:
        index = h
    else if hashtable[h].flag is AVAILABLE:
        if index == -1:
            index = h
        put (key, value) at hashtable[h] and mark that
        row as not available
        return SUCCESS
    else if hashtable[h].flag is NOT-AVAILABLE:
        if hashtable[h].key == key:
            return ERROR (i.e. key already exists)

    apply your probing method to compute the next h value
    if h is a previous h value:
        return ERROR
```

```
ALGORITHM: HASHTABLE-DELETE
INPUT: hashtable of size n
```

```
       key

deleted = false
compute h = hash(key) % n

while 1:

    if hashtable[h].flag is NOT-AVAILABLE:
        if hashtable[h].key is key:
            hashtable[h].flag = DELETED
            return SUCCESS
    else if hashtable[h].flag is AVAILABLE:
        return FAILURE (i.e. key is not found)

    else:
        do nothing

    apply your probing method to compute the next h value
    if h is a previous h value:
        return FAILURE
```

```
ALGORITHM: HASHTABLE-FIND
INPUT: hashtable of size n
       key
OUTPUT: index i where hashtable[h].key is key
            -1 is returned is key is not found

compute h = hash(key) % n

while 1:

    if hashtable[h].flag is AVAILABLE:
        return -1 (i.e., key not found)

    else if hashtable[h].flag is NOT-AVAILABLE:
        if hashtable[h].key is key:
            return h

    else:
        do nothing

    apply your probing method to compute the next h value
    if h is a previous h value:
        return FAILURE
```

(Of course you can also have a find operation that returns a pointer to the relevant row in the table.)

Note that there's since we are marking rows as DELETED, a possible useful operation is to find a row that was deleted and undelete that row – if it has not been already overwritten by an insert operation.

Exercise 108.6.1. In a hash table of size 8, insert the keys a,b,c,d,e,f,g assuming that they are hashed to 24,12,23,45,5,13,7 respectively. Draw the table (leave the value column blank). How many rows were read altogether? Next delete c. How many rows were read to do this? Insert h assuming that h is has a hash value of 19. How many rows were read? For each key in the table, write down the number of rows read to find that key. Which keys can be found with the least number of rows read? Which would require the most number of rows read? Compute the average number of rows read for search of the keys in the table. Rehash the table to another table of size 11. Now compute the new average number of rows read for search of the keys in the table.

Use this table:

0			
1			
2			
3			
4			
5			
6			
7			

Here are the runtimes. In the following n is the number of keys already in the table. Usually the keys is a fraction of the table size. So average space needed is $O(n)$. I will assume that the hash function is "reasonable", i.e., the hash function (after modding by the size of the table) distributes keys uniformly. The following runtimes assumes no rehashing/resizing the table.

- Insert:
  - Worst runtime = $O(n)$
  - Best, average runtime = $O(1)$
- Delete:

    - Worst runtime $= O(n)$
    - Best, average runtime $= O(1)$
- Find:
    - Worst runtime $= O(n)$
    - Best, average runtime $= O(1)$

108.7 Rehashing the table

When there are too many collisions, then either your hash table is too full or the hash function is bad. Assuming your hash function is fine, then it's time to expand your hash table. In other words, you need to have a larger array for the hash table. This of course implies that if you do want your hash table to be expandable, you have to put the table on the heap, i.e., use a dynamic array. Another thing is that you now cannot hardcode the size to mod within your hash function.

Rehashing is of course easy. Just run through your hash table array and perform inserts into the new table. If you are using tombstones (see previous section), of course make sure that you do not include them when you insert into the new table. That's it.

Note that rehashing is (obviously) time consuming if the old table has a large size.

108.8 Two types of hash tables

There are two types of hash tables. What I have talking about is called open address hash table. Later I'll talk about closed address hash tables. But to let you start writing some C++ code ASAP, I'm going to delay explaining closed address hash tables. The API in the next section will apply to both open and closed address hash tables.

108.9 API

Like I said, it's not easy designing hash functions. Furthermore, different scenarios might require different hash functions even for the same type of values. For instance strings representing human names might require a hash function that is different from strings representing book titles. It might therefore be a good idea when designing a hash table to have a class that allows you to specify specific hash functions.

We of course want to insert, delete, search in our hashtable. In the following, `h` is a hash table object.

`h.insert(key, value)`	Insert key-value into hash table. Exception is thrown if key is already in the hash table.
`h.erase(key)`	Delete key-value from hash table of given key. Exception is thrown if key is not found.
`h[key]`	Reference to value of given key. If not found exception is thrown.
`h.get(key, default)`	Return value of given key. If not found, default is returned.

So if `height` is a hash table of name-height pairs, we can do this:

```
height["Abe"] = 6.5;
```

Here are some auxiliary methods:

`h.clear()`	Remove all keys.
`h.size()`	Number of keys in the hash table.
`h.keys()`	Returns an iterable list of keys.
`h.values()`	Returns an iterable list of values.
`h.has_key(key)`	Returns true if key is present.
`h.update(h1)`	Update h with hash table h1.

For constructor, we want to allow users to specify initial size. If not specified, let's use a default that is stored as a static in the class that can be modified. Let's say the default is initially set to 97.

108.10 Implementation

Here's an abstract base class:

```
class Hashable
{
public:
    virtual unsigned int hash(unsigned int) = 0;
};
```

This is just to impose a hash method on a subclass. So for instance you can do this:

```
class Name
{
public:
    Name(const std::string & s)
        : s_(s)
    {}

private:
    std::string s_;
};


class HashableName: public Name, Hashable
{
public:
    HashableName(const std::string & s)
        : Name(s)
    {}

    unsigned int hash(unsigned int s)
    {
        unsigned int h = 0;
        ... compute h using s_ ...
        return h % s;
    }
};
```

Here's the class for a row of key-value in the hash table:

```
template < typename Key, typename Value >
class KeyValue
```

```
{
private:
    Key key_;
    Value value_;
};
```

and here's the class for a row in the hash table together with a flag:

```
template < typename Key, typename Value >
class HashtableRow
{
public:
    enum State
    {
        AVAILABLE, NOT_AVAILABLE, DELETED
    };
private:
    State state_;
    KeyValue< Key, Value > keyvalue_;
};
```

And now we have the hash table:

```
template < typename Key, typename Value >
class Hashtable
{
private:
    unsigned int size_;
    HashtableRow< Key, Value > * p_;
};
```

Note that the `HashtableRow` class can be nested in the `Hashtable` class.

Exercise 108.10.1. Implement the above hash table class. Test it thoroughly. □

Exercise 108.10.2. Read up on STL's `pair`. □

Exercise 108.10.3. Read up on STL's `unordered_map`. □

108.11 Open and closed; separate chaining

Now that you know enough about a hash table using an array, like I said earlier. it's time to talk about another type of hash table. The above method is called **open addressing**. Unfortunately, the above is also called **closed hashing**.

open addressing

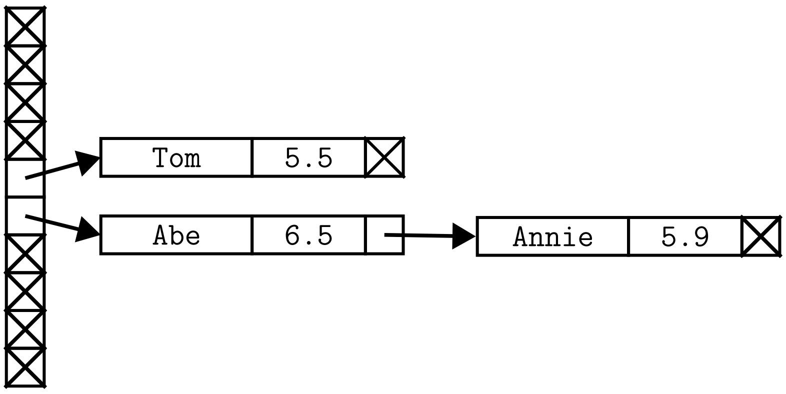
closed hashing

Now I'm going to talk about closed addressing which is sometimes called open hashing and sometimes called separate chaining. Let me put them together so that you can see everything:

- Method 1: Open addressing, closed hashing
- Method 2: Closed addressing, open hashing, separate chaining.

Obviously computer scientists should come up with better names.

What I'm going to do now is to allow each bucket (a row in an array) to have the ability to hold more than one key. This is easy. What I'll do is that each row in the array is now replaced by containers. You can choose any container. But the easiest is to use a linked list, say a singly linked list.



In the above diagram, the array is an array of pointers to singly linked list nodes or `NULL`. Of course you can have linked list objects in the array, whether the linked list object contain pointers or sentinel nodes. The linked lists are called chains.

You can think of the above as 10 buckets. Most of the buckets have no keys. The bucket at index 4 has one key and the bucket at index 5 has two keys.

Inserts, deletes, and search are obvious and easy.

Clearly the worse case is measured by the length of the longest singly linked list in the hash table structure.

When there are many linked lists of long lengths, of course you can rehash with a largest array.

Besides using linked list, you can of course use balanced trees or even hash tables at each bucket.

108.12 Load factor

load factor

The **load factor** of a hash table, usually denoted α, is defined to be

$$\alpha = \frac{n}{m}$$

where n is the number of keys in the table and m is the number of buckets. On the average, the runtime for insert, delete, search is

$$1 + \alpha$$

where the 1 is due to the constant time to compute the hash and accessing the array. All of this assume that the hash function more or less is well behaved and spread the keys uniformly. If the load factor is maintained at a constant value, this means that the runtimes are $O(1)$ on the average.

In the *worse* case insert, delete and search becomes $O(n)$. For instance if all the keys congregate at one hash value and the key you're search for is the last in the chain or last in the probe sequence, the runtime will be $O(n)$. The same reasoning applies to the case of delete and insert.

In general, a hash table usually has a minimum load factor threshold α_{min} (example: 0.25) and a maximum load factor threshold α_{max} (example: 0.75) and the hash table maintain some statistics on the load factor. When the load factor α goes beyond the maximum threshold, the hash table increases the table size and rehash. If the load factor drops below the minimum threshold, the hash table decreases the table size and rehash.

The space requirement is $O(n)$ since a good hash table requires the α be a reasonable proportion of n, for instance $4n$ to maintain a 0.25 load factor.

108.13 C++ STL unordered map

The C++ STL contains the `unordered_map` class template which is a hashtable. You will need to compile your code with at least C++11:

```
g++ *.cpp -std=c++11
```

`unordered_map` uses open hashing (separate chaining), i.e., i.e., it's an array of linked list of key-value pairs. The class:

```
std::unordered_map< X, Y >
```

assumes that type (or class) `X` has `operator==()` (for the obvious reason). Also, each (key, value) pair in the hashtable is a `std::pair< X, Y >` object (you can think of such an object as a 2-tuple). If `pair` is a `std::pair< X, Y >` value, then the first value is `pair.first` has type `X` and the second value is `pair.second` of type `Y`.

To add a key-value pair (k, v) into a `std::unordered_map< X, Y >` object `h`, you do either

```
h.insert({k, v}); // check if error
```

or

```
h[k] = v;
```

Both methods in the above work in the same way if `k` is not in `h`. If `k` is in `h`, then `h.insert({k, v})` will give you an exception while `h[k] = v` will update the value of `k` with `v`.

`h.find(k)` will return an iterator. The iterator will equal `h.end()` if `k` is not found. Otherwise it will point to the `std::pair` object with the given key (as first value).

```
auto p = h.find(k);
std::cout << (p == h.end() ? "not found\n" : "found\n");
```

As mentioned above C++ unordered map is implemented using open address hashtable, i.e., it's an array of linked list of key-value pairs. Each linked list is also called a bucket. Besides that, the (key, value) pairs on in buckets where each bucket has a linked list of (key, value) pairs. If `h` is an unordered map, then `h.bucket(k)` is `i` where `k` is found in the `i`–th bucket where `i` = 0, ..., `h.bucket_count() - 1`. You can get an iterator to run through a bucket (not the whole hashtable). The following prints all the key-value pairs in bucket i:

```
for (typename T::const_local_iterator p = h.begin(i);
     p != h.end(i); ++p)
{
    std::cout << p->first << ": " << p->second << "  ";
}
```

You also have iterators that runs through all the buckets, i.e., the whole hashtable:

```
for (typename std::unordered_map< X, Y >::const_iterator p = h.begin();
     p != h.end(); ++p)
{
    std::cout << p->first << ": " << p->second << "  ";
}
```

Study the following program carefully.

```
#include <iostream>
#include <string>
#include <unordered_map>

// Height is hashtable with key = std::string, value = double
typedef std::unordered_map<std::string, double> Height;

template < typename T >
void print(const T & h)
{
    std::cout << "(key, value) pairs in h ...\n"; // iterating
                                     // over all entries
    for (typename T::const_iterator p = h.begin();
         p != h.end(); ++p)
    {
        std::cout << p->first << ": " << p->second
                  << " ... "
                  << '\n';
    }
}

template < typename T >
void print_buckets(const T & h)
{
    std::cout << "(key, value) pairs in buckets of h ...";
    for (unsigned int i = 0; i < h.bucket_count(); ++i)
                                     // iterating over all
                                     // buckets
    {
        std::cout << "bucket " << i << ": ";
        for (typename T::const_local_iterator p = h.begin(i);
             p != h.end(i); ++p)
        {
            std::cout << '['
                      << p->first << ": " << p->second
                      << "] ";
        }
        std::cout << '\n';
    }
}

void print_stats(Height & h)
{
    std::cout << "size/buckets/load factor/max load factor:"
              << h.size() << '/'
              << h.bucket_count() << '/'
              << h.load_factor() << '/'
              << h.max_load_factor() << '\n';
}

int main()
{
    Height h;
    h.insert({"John", 5.7});         // insert operation
    h.insert({"Tom", 5.8});
    h["Mary"] = 5.9;
    h["Sue"] = 6.0;
    h["Jane"] = 6.1;
    h["Priscilla"] = 6.1;
    h["Sheila"] = 6.1;
    h["Ashley"] = 6.1;

    h.erase("Tom");                  // delete operation

    std::cout << h["John"] << '\n';  // find key and get value
    h["John"] = 6.2;                 // update by key with new
                                     // value
    std::cout << h["John"] << '\n';

    Height::iterator p = h.find("Mary"); // find by key and
                                     // get iterator
    std::cout << (p != h.end() ? "found" : "not found")
              << '\n';
    print(h);
    print_buckets(h);
    print_stats(h);

    std::cout << "increase size to 641 ...\n"; // changing the
                                 // size
    h.reserve(641);
    print_stats(h);

    std::cout << "clear ...\n";
    h.clear();                   // clear the map
    print_stats(h);

    std::cout << "constructor with size 1000...\n";
    Height h1(1000);             // reserve size of
                                 // approx 1000
    print_stats(h);

    return 0;
}
```

```
5.7
6.2
found
(key, value) pairs in h ...
Ashley: 6.1 ...
Sheila: 6.1 ...
Priscilla: 6.1 ...
Mary: 5.9 ...
John: 6.2 ...
Sue: 6 ...
Jane: 6.1 ...
(key, value) pairs in buckets of h ...bucket 0:
bucket 1:
bucket 2: [Jane: 6.1]
bucket 3:
bucket 4: [Ashley: 6.1]
bucket 5: [John: 6.2] [Sue: 6]
bucket 6: [Mary: 5.9]
bucket 7:
bucket 8: [Sheila: 6.1]
bucket 9:
bucket 10: [Priscilla: 6.1]
size/buckets/load factor/max load factor:7/11/0.636364/1
```

```
increase size to 641 ...
size/buckets/load factor/max load factor:7/661/0.01059/1
clear ...
size/buckets/load factor/max load factor:0/661/0/1
constructor with size 1000...
size/buckets/load factor/max load factor:0/661/0/1
```

For convenience, I'm going to put some of the above code into `Height.h`:

```
#ifndef HEIGHT_H
#define HEIGHT_H

#include <iostream>
#include <string>
#include <unordered_map>

template < typename T >
void print(const T & h)
{
    std::cout << "(key, value) pairs in h ...\n"; // iterating
                                         // over all entries
    for (typename T::const_iterator p = h.begin();
         p != h.end(); ++p)
    {
        std::cout << p->first << ": " << p->second
                  << " ... "
                  << '\n';
    }
}

template < typename T >
void print_buckets(T & h)
{
    std::cout << "(key, value) pairs in buckets of h ...\n";
    for (unsigned int i = 0; i < h.bucket_count(); ++i)
                                         // iterating over all
                                         // buckets
    {
        std::cout << "bucket " << i << ": ";
        for (typename T::const_local_iterator p = h.begin(i);
             p != h.end(i); ++p)
        {
            std::cout << '['
                      << p->first << ": " << p->second
                      << "] ";
        }
        std::cout << '\n';
    }
}

template < typename T >
void print_stats(const T & h)
{
    std::cout << "size/buckets/load factor/max load factor:"
              << h.size() << '/'
              << h.bucket_count() << '/'
              << h.load_factor() << '/'
              << h.max_load_factor() << '\n';
}

#endif
```

Note that the `reserve()` method (to change the number of bucket) asks for 641 buckets, but you might get a number ≥ 641.

xxx	yyy
`std::unordered_map< K, V > h`	`h` is a hashtable of `std::pair< K, V >`
`h.reserve(n)`	
`h.insert({k, v})`	Add `(k, v)` to `h`. If `k` is already in `h`, `h` is not changed.
`h[k]` = v	If `k` is in `h`, its value is changed to `v`. If `k` is not found, `(k,v)` is inserted.
`h.erase(k)`	Remove `k` from `h`.
`h.begin()`	
`h.end()`	
`h.find(k)`	Return `std::unordered_map< X, Y >::iterator`
`h.bucket_count()`	
`h.bucket(i)`	
`h.begin(i)`	
`h.end(i)`	
`h.size()`	Number of key-value pairs in `h`
`h.load_factor()`	
`h.max_load_factor()`	

108.13.1 Hash function

Note that in the above examples, when we use `std::unordered_map`, we don't have to say how to hash the keys. That's because `std::unordered_map` use the built-in `std::hash`. Go ahead and run this:

```
#include <iostream>
#include <string>
#include <functional> // for std::hash

int main()
{
    std::hash< int > h0;
    std::cout << "hash of\n";
    std::cout << "42: " << h0(42) << '\n';
    // or
    std::cout << "42: " << std::hash< int >()(42) << '\n';

    std::cout << "-1: " << std::hash< int >()(-1) << '\n'
              << "3.14: " << std::hash< double >()(3.14) << '\n'
              << "0.0: " << std::hash< double >()(2.0) << '\n'
              << "'a': " << std::hash< char >()('a') << '\n'
              << "true: " << std::hash< bool >()(true) << '\n'
              << "\"hello world\": "
              << std::hash< std::string >()("hello world") << '\n';
    return 0;
}
```

```
[student@localhost 350-hashtable] g++ main.cpp; ./a.out
hash of
42: 42
42: 42
-1: 18446744073709551615
3.14: 5464867211497793177
0.0: 6369015886390043782
'a': 97
true: 1
"hello world": 5577293430985752569
```

The return value of `std::hash` is a `size_t` value, which is like an unsigned integer value that is usually takes up 32 or 64 bits.

For `int` and `unsigned int` variable `x`, note that `std::hash(x)` is the same as (size_t)(x). For bool and char values, they are typecasted to size_t values. For double and string So std::hash knows how to has values of the following types: int, double, char, std::string. But what if you want to hash values of other types?

You can define your own hash function for X values where X is the first type (the key type) of the unordered map. There are several ways to do this.

METHOD 1. One way is to create a class or struct where objects are function objects (i.e., they have operator()) that returns an size_t or unsigned int when given a value of type X. You then include this class as the third type parameter for unordered_map:

```
#include <iostream>
#include <string>
#include <unordered_map>
#include "Height.h"

class Hash // or use a struct
{
public:
    unsigned int operator()(const std::string & s) const
    {
        return 3; // a very bad hash function
    }
};

typedef std::unordered_map<std::string, double, Hash> Height;

int main()
{
    Height h;
    h.insert({"John", 5.7});
    h.insert({"Tom", 5.8});
    h["Mary"] = 5.9;
    h["Sue"] = 6.0;

    print(h);
    print_buckets(h);
    print_stats(h);

    return 0;
}
```

```
(key, value) pairs in h ...
Sue: 6 ...
Mary: 5.9 ...
John: 5.7 ...
Tom: 5.8 ...
(key, value) pairs in buckets of h ...
bucket 0:
bucket 1:
bucket 2:
bucket 3: [Sue: 6] [Mary: 5.9] [John: 5.7] [Tom: 5.8]
bucket 4:
size/buckets/load factor/max load factor:4/5/0.8/1
```

METHOD 2. Here's a second method to create a hash function to be used by `unordered_map`. This method is not as flexible as the above method and can only be used when the hashing is performed on a class that is not yet defined. Suppose you have a class `vec2d` where `operator==` is defined. Each `vec2d` has two `double` members that can be accessed by public methods `vec2d::x()` and `vec2d::y()` Note that `std::hash< double >` is already defined. Basically I want to create a hash function on `vec2d` objects using `std::hash< double >`.

I can do this:

```
#include <iostream>
#include <functional> // for std::hash
#include <unordered_map>
#include "vec2d.h"

template <>
struct std::hash< vec2d >
{
    size_t operator()(const vec2d & v) const
    {
        return std::hash< double >{}(v.x() + v.y()) ; // bad hash!
    };
};

typedef std::unordered_map< vec2d, double > temperature;

int main()
{
    temperature h;
    h[vec2d(1.1, 2.2)] = 5.9;
    return 0;
}
```

In this case, note that the class `std::unordered_map< vec2d, double >` does not require us to include the hash. That's because we have extended `std::hash` to be able to handle `vec2d` objects, which is used by default.

Note that this method is not as flexible as the first method. This is because you cannot, for instance, have two different `std::hash< std::string >`.

108.13.2 Size

Now I'm going to resize to a smaller size:

```
#include <iostream>
#include <string>
#include <unordered_map>
#include "Height.h"

class Hash // or use a struct
{
public:
    unsigned int operator()(const std::string & s) const
    {
        return 3; // a very bad hash function
    }
};

typedef std::unordered_map<std::string, double, Hash> Height;

int main()
{
    Height h;
    h.insert({"John", 5.7});
    h.insert({"Tom", 5.8});
    h["Mary"] = 5.9;
    h["Sue"] = 6.0;
    h.reserve(4);
    print(h);
    print_buckets(h);
    print_stats(h);

    return 0;
}
```

```
(key, value) pairs in h ...
Sue: 6 ...
Mary: 5.9 ...
John: 5.7 ...
Tom: 5.8 ...
(key, value) pairs in buckets of h ...
bucket 0:
bucket 1:
bucket 2:
bucket 3: [Sue: 6] [Mary: 5.9] [John: 5.7] [Tom: 5.8]
bucket 4:
size/buckets/load factor/max load factor:4/5/0.8/1
```

108.13.3 Rehash

Now I'm going to rehash with a new bucket size n. The actually bucket size is $\geq n$ (i.e., it might not be exactly n). Note that when the load factor is $>$ the max load factor, the hash table will automatically rehash.

```
#include <iostream>
#include <string>
#include <unordered_map>
#include "Height.h"

typedef std::unordered_map<std::string, double> Height;

int main()
{
    Height h;
    h.insert({"John", 5.7});
    h.insert({"Tom", 5.8});
    h["Mary"] = 5.9;
    h["Sue"] = 6.0;

    print_buckets(h);
    print_stats(h);

    std::cout << "rehash with suggested bucket size 5\n";
    h.rehash(5);
    print_buckets(h);
    print_stats(h);
    return 0;
}
```

```
(key, value) pairs in buckets of h ...
bucket 0:
bucket 1:
bucket 2: [Sue: 6] [Mary: 5.9]
bucket 3: [Tom: 5.8]
bucket 4: [John: 5.7]
size/buckets/load factor/max load factor:4/5/0.8/1
rehash with suggested bucket size 5
(key, value) pairs in buckets of h ...
bucket 0:
bucket 1:
bucket 2: [Sue: 6] [Mary: 5.9]
bucket 3: [Tom: 5.8]
bucket 4: [John: 5.7]
size/buckets/load factor/max load factor:4/5/0.8/1
```

Exercise 108.13.1. The `unordered_map` requires you to use the `first` and `second` members of the `std::pair` class. Write a class `HashTable<X, Y>` that has methods `key()` and `value()`.

Exercise 108.13.2. Download a huge text file (example: a large book from `gutenberg.net`) and compute the frequently of each letter in the file. When you're done with that, do the same for word frequency. Frequency counts like these occur in many areas such as cryptography, text analysis/mining, computational linguistics, etc.

Exercise 108.13.3. We have already talked about the concept of sparse matrices and an implementation using linked lists. Another way to implement sparse matrices is to store ((r, c), v) in a hashtable where at row r and column c (the key is (r, c)), the value is v. Implement a sparse matrix class using hashtables.

Exercise 108.13.4. Suppose you are given a collection X of n integers, say stored as a linked list. If given you a number x, how fast can you find all a and b in the X such that $a + b = x$? (Obviously this can be done in $O(n^2)$ – that's not what I'm looking for.)

Exercise 108.13.5 (Computing a space efficient hash function)**.** There are times when the set of keys are fixed. Say there are 16 keys. If you use a hashtable, you might need a size (number of buckets) much larger than 16 to avoid collision. Add a method to your Hashtable class called `compactify` that uses the default hash function followed by the unsigned int multiplication of `m` such that the resulting hash mod size of table gives you values 0, 1, 2, 3, ..., 15 which implies that you only need a hashtable of size 16. If 0, 1, 2, 3, ..., 15 is impossible, since a range that that is the smallest possible. (See exercise below as well.)

Exercise 108.13.6. The assembly language code generated by a compiler sometimes will use hashing for switch cases when there is a large collection of case labels which contiguous values. The following is similar. Write a `SwitchCase` class that allows you to add (c, f) where c is a switch case label (an integer) and f is a function object that takes an integer for function call.

```
class Function
{
public:
    Function(int i) : i_(i) {}
    int operator(int x) { std::cout << i_ + x << '\n'; }
};

int main()
{
    SwitchCase machine;
    Function f(42);
    Function g(43);
    Function h(647);

    machine.add(1, f);
    machine.add(6, g);
    machine.add(1001, h);

    machine(6, 1000); // Executes g(1000) which prints 1043

    return 0;
}
```

Note that hashtables are not good at range searches, i.e., if you need a collection of iterations to entries with key values in a range, then a hashtable is the wrong data structure.

108.14 SHA256

As root install the OpenSSL library:

```
dnf -y install openssl-devel
```

The following is how you should compile your program:

```
g++ main.cpp -I/opt/ssl/include/ -L/opt/ssl/lib/ -lcrypto
```

```
#include <iostream>
#include <iomanip>
#include <string>
#include <sstream>
#include <openssl/evp.h>

bool sha256_hexdigest(const std::string & unhashed, std::string & hashed)
{
    bool success = false;
    EVP_MD_CTX * context = EVP_MD_CTX_new();

    if (context != NULL)
    {
        if (EVP_DigestInit_ex(context, EVP_sha256(), NULL))
        {
            if (EVP_DigestUpdate(context,
                                 unhashed.c_str(), unhashed.length()))
            {
                unsigned char hash[EVP_MAX_MD_SIZE];
                unsigned int lengthOfHash = 0;
                if (EVP_DigestFinal_ex(context, hash, &lengthOfHash))
                {
                    std::stringstream ss;
                    for (unsigned int i = 0; i < lengthOfHash; ++i)
                    {
                        ss << std::hex << std::setw(2) << std::setfill('0')
                           << (int)hash[i];
                    }
                    hashed = ss.str();
                    success = true;
                }
            }
        }
        EVP_MD_CTX_free(context);
    }
    return success;
}

int main(int argc, char ** argv)
{
    std::string input, output;

    input = "hello world";
    if (sha256_hexdigest(input, output))
    {
        std::cout << "hello world: " << output << std::endl;
    }
    else
    {
        std::cout << "FAILURE!" << std::endl;
    }

    input = "hello worle";
    if (sha256_hexdigest(input, output))
    {
        std::cout << "hello worle: " << output << std::endl;
    }
    else
    {
        std::cout << "FAILURE!" << std::endl;
    }

    return 0;
}
```

```
[student@localhost 350-hashtable] g++ main1.cpp -I/opt/ssl/include/ -L/opt/ssl/l
ib/ -lcrypto
[student@localhost 350-hashtable] ./a.out
hello world: b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9
hello worle: 0fc30e735a0228a31cbbb969988b4f50e02e737f979f091d7d224b765443f5d4
```

The input is a string and the output is a hex string.

The above example is mainly for an input that is an `std::string`. For optimization, here's a version where the input is a character array and a length variable which is also better for hashing general data such that integers, doubles, etc.:

```
#include <iomanip>
#include <iostream>
#include <cstring>
#include <sstream>
#include <string>
#include <openssl/evp.h>

bool sha256_hexdigest(const char * unhashed, size_t unhashed_length,
                   std::string & hashed)
{
    bool success = false;
    EVP_MD_CTX * context = EVP_MD_CTX_new();

    if (context != NULL)
    {
        if (EVP_DigestInit_ex(context, EVP_sha256(), NULL))
        {
            if (EVP_DigestUpdate(context,
                                 unhashed, unhashed_length))
            {
                unsigned char hash[EVP_MAX_MD_SIZE];
                unsigned int lengthOfHash = 0;
                if (EVP_DigestFinal_ex(context, hash, &lengthOfHash))
                {
                    std::stringstream ss;
                    for (unsigned int i = 0; i < lengthOfHash; ++i)
                    {
                        ss << std::hex << std::setw(2) << std::setfill('0')
                           << (int)hash[i];
                    }
                    hashed = ss.str();
                    success = true;
                }
            }
        }
        EVP_MD_CTX_free(context);
    }
    return success;
}

int main(int argc, char ** argv)
{
    char input[1024];
    std::string output;

    strcpy(input, (const char *)"hello world");
    if (sha256_hexdigest(input, strlen(input), output))
    {
        std::cout << "hello world: " << output << std::endl;
    }
    else
    {
        std::cout << "FAILURE!" << std::endl;
    }

    strcpy(input, (const char *)"hello worle");
    if (sha256_hexdigest(input, strlen(input), output))
    {
        std::cout << "hello worle: " << output << std::endl;
    }
    else
    {
        std::cout << "FAILURE!" << std::endl;
    }

    int x;
    x = 42;
    if (sha256_hexdigest((const char *)(&x), sizeof(int), output))
    {
        std::cout << x << ": " << output << std::endl;
    }
    else
    {
        std::cout << "FAILURE!" << std::endl;
    }

    x = 43;
    if (sha256_hexdigest((const char *)(&x), sizeof(int), output))
    {
        std::cout << x << ": " << output << std::endl;
    }
    else
    {
        std::cout << "FAILURE!" << std::endl;
    }

    double d;
    d = 3.1415;
    if (sha256_hexdigest((const char *)(&d), sizeof(double), output))
    {
        std::cout << d << ": " << output << std::endl;
    }
    else
    {
        std::cout << "FAILURE!" << std::endl;
    }

    d = 3.1414;
    if (sha256_hexdigest((const char *)(&d), sizeof(double), output))
    {
        std::cout << d << ": " << output << std::endl;
    }
    else
    {
        std::cout << "FAILURE!" << std::endl;
    }

    return 0;
}
```

```
[student@localhost 350-hashtable] g++ main2.cpp -I/opt/ssl/include/ -L/opt/ssl/l
ib/ -lcrypto
[student@localhost 350-hashtable] ./a.out
hello world: b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9
hello worle: 0fc30e735a0228a31cbbb969988b4f50e02e737f979f091d7d224b765443f5d4
42: e8a4b2ee7ede79a3afb332b5b6cc3d952a65fd8cffb897f5d18016577c33d7cc
43: 7c9bf6a88aed1539a3276462bb9e977bede22cf2c89f96bf61f590da5504ccc7
3.1415: 3e85911ac35b0603359864a15a82c9108cac2335847d38d3dab3b4fc5e754dbf
3.1414: 3e75b7cdcf84ae4c9b73f8d79d7c3ec26f2a1ebb8c5104c04cd0303c4296e9f5
```

Note that in the function above SHA256 was chosen (see the `EVP_sha256()` in the above code). Other options includes

```
const EVP_MD * EVP_md5(void);
const EVP_MD * EVP_sha512(void);
```

See `https://linux.die.net/man/3/evp_sha512`

In the examples above the code

```
std::stringstream ss;
for (unsigned int i = 0; i < lengthOfHash; ++i)
{
    ss << std::hex << std::setw(2) << std::setfill('0')
       << (int)hash[i];
}
hashed = ss.str();
```

forms a hex string which is the string returned. Each of the `hash[i]` is an integer.

Index