Welcome to CS 225

Data Structures

lab_hash

✓ Hash tables

✓ Collision resolution:

- Separate chaining
- Linear probing
- Double hashing

Now: Implementation

Separate Chaining

1. insert
2. find
3. remove
4. resizeTable

Linear Probing Hash Table

1. insert
2. findIndex
3. remove
4. resizeTable

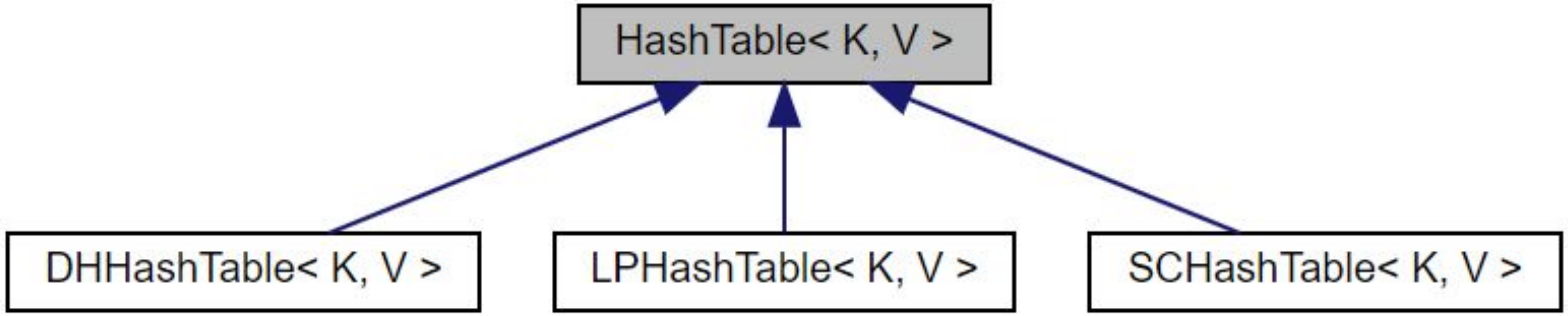
Double Hashing Hash Table

1. insert
2. findIndex
3. remove

The following files (and ONLY those files!!) are used for grading this lab:

dhhashtable.cpp
lphashtable.cpp
schashtable.cpp

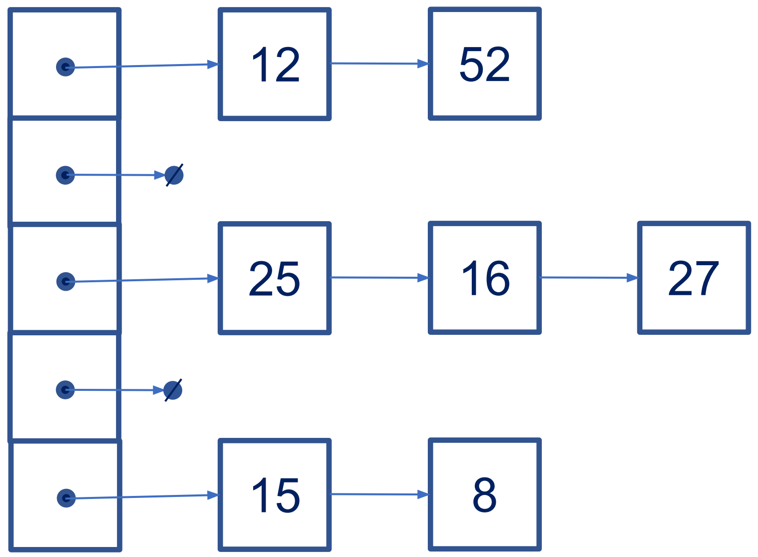
HashTable< K, V > Class



Separate Chaining (Open hashing)

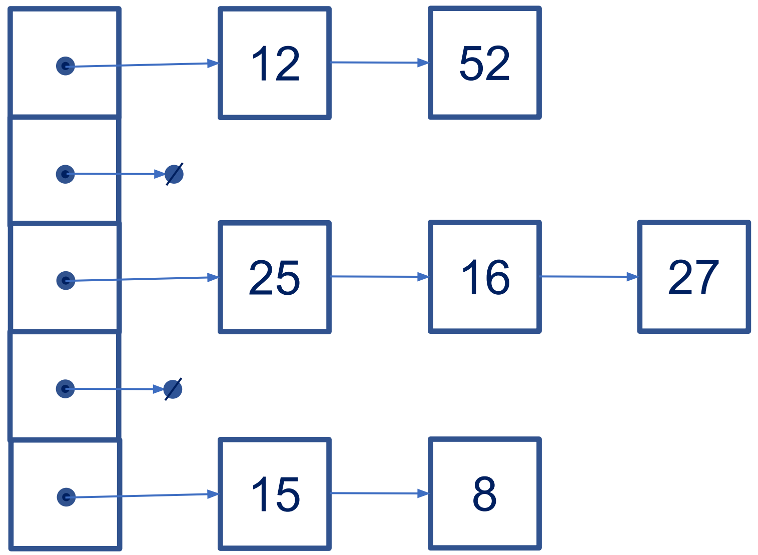
Values are stored outside of the table:

The idea is to make each cell of hash table point to a linked list of records that have same hash value.



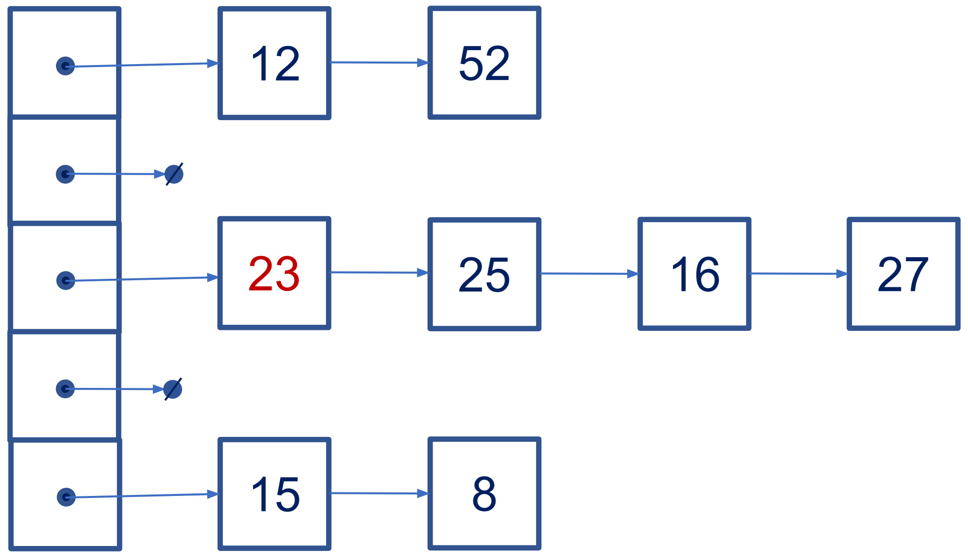
Separate Chaining – insert

Imagine we are inserting new element with key=23; if hash(23)=2, where is new element inserted? What is the optimal running time for insert?



Separate Chaining – insert

Imagine we are inserting new element with key=23; if hash(23)=2, 23 is inserted in the beginning of the linked list on index 2 – inserting in the beginning of the linked list has O(1) running time.



Separate Chaining – insert(key)

?

To calculate hash value for the given key, you can use function declared in `hashes.h` file inside namespace `hashes`:

```
template <class K>
unsigned int hash(const K& key, int size);
```

Given key you can calculate *h(key)=i* and insert element in the beginning of the linked list on index *i*.

What else do you have to do when inserting new element?

Separate Chaining – insert(key)

To calculate hash value for the given key, you can use function declared in `hashes.h` file inside namespace `hashes`:

```
template <class K>
unsigned int hash(const K& key, int size);
```

Given key you can calculate *h(key)=i* and insert element in the beginning of the linked list on index *i*.

What else do you have to do when inserting new element?
Increase element count. Calculate new load factor and resize table if necessary.

Separate Chaining – resizeTable

?

- Load factor = elem / size
- This is called when the load factor for our table is ≥ 0.7
- It should resize the internal array for the hash table.
  (Use the return value of findPrime with a parameter of double the current size to set the size, see other calls to resize for reference.).

What else should happen in function `resizeTable`?

Separate Chaining – resizeTable

- Load factor = elem / size
- This is called when the load factor for our table is ≥0.7
- It should resize the internal array for the hash table.
  (Use the return value of findPrime with a parameter of double the current size to set the size, see other calls to resize for reference.).

What else should happen in function `resizeTable`?
Hash value of every element will change once we resize the table (since `size` is changing). So we need to calculate new hash value for each element and insert in them in the resized container in the proper place.

Separate Chaining – remove(key) & find(key)

Remove(key) finds the element, removes from the list and decreases the number of elements.

Find(key) finds the element and returns the value corresponding to it.

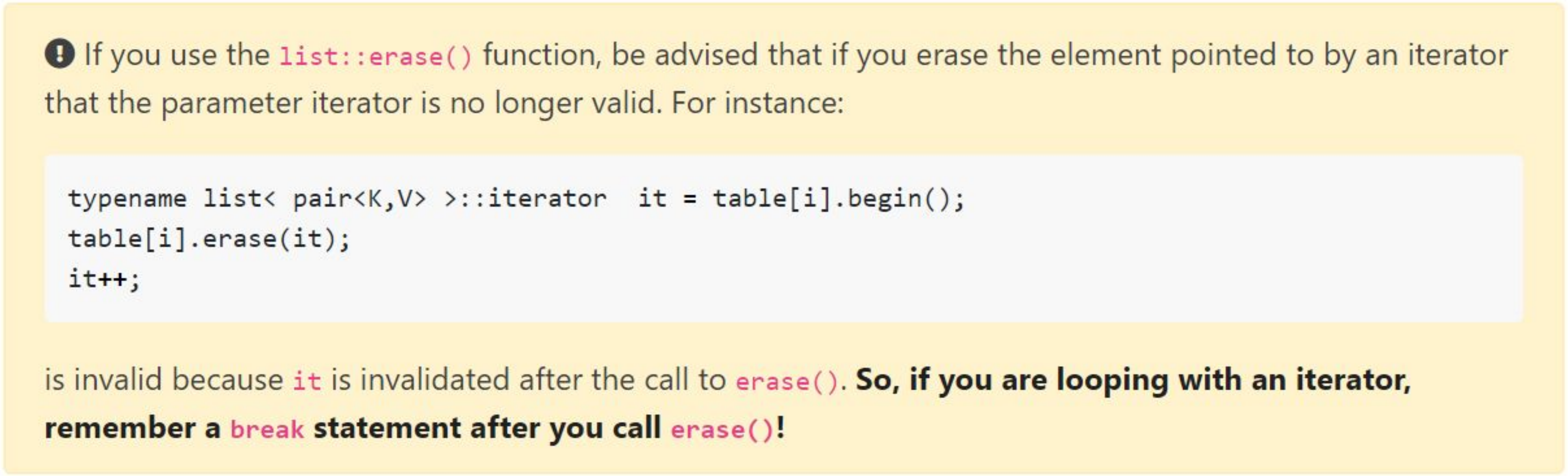
How can you find the element in the table?

Separate Chaining – remove(key) & find(key)

How can you find the element in the table?

Use hash function to get the index of the list element should be in and then go over every element in the list to find the key.

Remember:



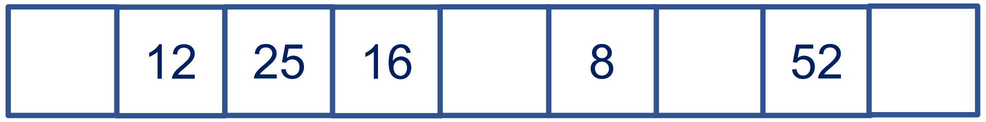
If you use the `list::erase()` function, be advised that if you erase the element pointed to by an iterator that the parameter iterator is no longer valid. For instance:

```
typename list< pair<K,V> >::iterator  it = table[i].begin();
table[i].erase(it);
it++;
```

is invalid because `it` is invalidated after the call to `erase()`. **So, if you are looping with an iterator, remember a `break` statement after you call `erase()`!**

Closed Hashing

All entry records are stored in the bucket array itself.



Collision resolution strategies:

1. Linear probing
2. Double hashing

LPHashTable<K,V> class

A HashTable implementation that uses linear probing as a collision resolution strategy.

//Storage for our LPHashTable.

```
    std::pair<K, V>** table;
```

/** Flags for whether or not to probe forward when looking at a particular cell in the table. This is a dynamic array of booleans that represents if a slot is (or previously was) occupied. This allows us determine whether or not we need to probe forward to look for our key. */

```
    bool* should_probe;   //why do we need this?
```

/**Helper function to determine the index where a given key lies in the LPHashTable. If the key does not exist in the table, it will return -1.*/

```
    int findIndex(const K& key) const;
```

// inherited from HashTable

```
    virtual void resizeTable();
```

LPHashTable<K,V> - insert(key):

Hash function: h(key, size) = key % size

14			31		12	27

Size =7

Insert: 33;

```
i = h(key = 33, size = 7) =  33 % 7 = 5
```

Calculate `i=(i+1) mod size,` until `array[i]` becomes an empty slot:

```
i = (5+1) % 7 = 6
i = (6+1) % 7 = 0
i = (0+1) % 7 = 1
```

14	33		31		12	27

findIndex(33)

Is this correct?

14	33		31		12	27

Hash function: h(key, size) = key % size

```
1. i = h(33, 7);
2. if (array[i] == 33) return i;
3. Calculate: i=(i+1) mod size;
   • if (array[i] == 33) return i;
   • if (array[i] == empty slot)
         return ‘element not in the table’
```

findIndex(33)

Is this correct?

14	33		31		12	27

Hash function: h(key, size) = key % size

```
1. i = h(33, 7);
2. if (array[i] == 33) return i;
3. Calculate: i=(i+1) mod size;
   • if (array[i] == 33) return i;
   • if (array[i] == empty slot)
           return ‘element not in the table’
```

What if we delete 14 before find(33)?

	33		31		12	27

`findIndex(33)`

14	33		31		12	27
T	T	F	T	F	T	T

Hash function: h(key, size) = key % size

```
Probe until you find your element;
Stop if you find empty index where nothing was
inserted from the start.
```

This is why we need: `bool* should_probe;`

`findIndex(33)`

	33		31		12	27

T	T	F	T	F	T	T

Hash function: h(key, size) = key % size

```
Probe until you find your element;
Stop if you find empty index where nothing was
inserted from the start.
```

This is why we need: `bool* should_probe;`

```
Removing 14 doesn't remove
should_probe=True!
```

LPHashTable<K,V> class

A HashTable implementation that uses linear probing as a collision resolution strategy.

ResizeTable follows same logic as in Separate changing table.

What else do you have to resize apart from resizing hash table?

LPHashTable<K,V> class

A HashTable implementation that uses linear probing as a collision resolution strategy.

ResizeTable follows same logic as in Separate changing table.

What else do you have to resize apart from resizing hash table?
resize to new size should_probe array.

Remember: Hash value of every element will change once we resize the table (since `size` is changing). So we need to calculate new hash value for each element and insert in them in the resized container in the proper place.

DHHashTable<K,V> class

A HashTable implementation that uses double hashing as a collision resolution strategy.

Given two hash functions h_1 and h_2, the i_{th} location in the bucket sequence for value k in the hash table is:

$$h(i,k) = \left(h_1(k) + i * h_2(k)\right) \, mod \; size$$

Check bucket	Iteration
4	0
7	1
10	2
0	3

Let's say we want to insert 12, h(1) = 4, h(2) = 3, and the size is 13

Why would you use double hashing instead of linear probing?

Second hash function is declared in hashes.h: `secondary_hash`

Double hashing:

$$h(i,k) = \left(h_1(k) + i * h_2(k)\right) \, mod \; size$$

S = { 16, 8, 4, 13, 29, 11, 22 }

```
h1(k) = k % 7
h2(k) = 5 - (k % 5)
```

	8	16		4		13

	8	16	29	4		13

	8	16	29	4	11	13

22	8	16	29	4	11	13

DHHashTable<K,V> class

A HashTable implementation that uses double hashing as a collision resolution strategy.

Implementation for functions in DHHashTable is very similar to LPHashTable.

Try to identify parts of functions that will be affected by double hashing.