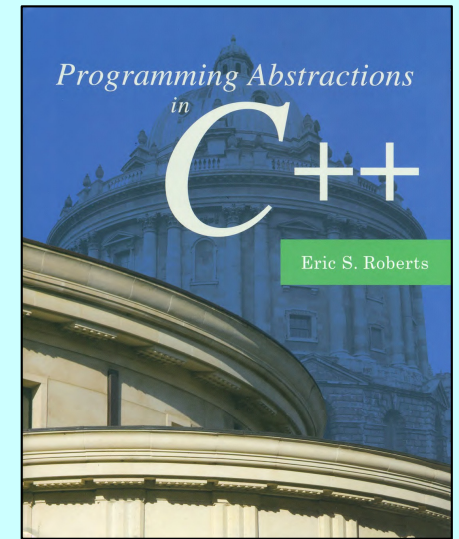


CHAPTER 15

Maps

A map was a fine thing to study when you were disposed to think of something else

—George Eliot, *Middlemarch*, 1874



15.1 Implementing maps using vectors

15.2 Lookup tables

15.3 Hashing

15.4 Implementing the `HashMap` class



The **Map**<*type*, *type*> Class

- A *map* is conceptually similar to a *dictionary* in real life (and in Python), which allows you to look up a word to find its meaning.
- The **Map** class is a generalization of this idea that provides an association between an identifying tag called a *key* (e.g., the word in a dictionary) and an associated *value* (e.g., the definition of the word in the dictionary), which may be a much larger and more complicated structure.
- Map declaration: `Map<key type, value type> map;`
- The type for the keys stored in a **Map** must define a natural ordering, usually through a `less` function and/or `<` operator so that *the keys can be compared and ordered*.
- E.g., two most commonly used maps:

```
Map<string, string> dictionary;  
Map<string, double> symbolTable;
```



Methods in the Map Classes

- A *map* associates *keys* and *values*. The Stanford library offers two flavors of maps, **Map** and **HashMap** (more about *hash* later), both of which implement the following methods:

map.size()

Returns the number of key/value pairs in the map.

map.isEmpty()

Returns **true** if the map is empty.

map.put(key, value) *or* **map[key] = value;**

Makes an association between **key** and **value**, discarding any existing one.

map.get(key) *or* **map[key]**

Returns the most recent value associated with **key**.

map.containsKey(key)

Returns **true** if there is a value associated with **key**.

map.remove(key)

Removes **key** from the map along with its associated value, if any.

map.clear()

Removes all key/value pairs from the map.

Exercise – State Code

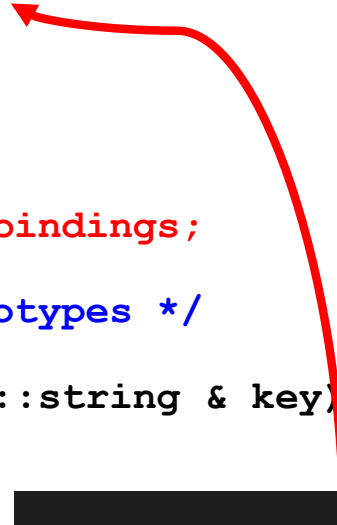
AK Alaska	HI Hawaii	ME Maine	NJ New Jersey	SD South Dakota
AL Alabama	IA Iowa	MI Michigan	NM New Mexico	TN Tennessee
AR Arkansas	ID Idaho	MN Minnesota	NV Nevada	TX Texas
AZ Arizona	IL Illinois	MO Missouri	NY New York	UT Utah
CA California	IN Indiana	MS Mississippi	OH Ohio	VA Virginia
CO Colorado	KS Kansas	MT Montana	OK Oklahoma	VT Vermont
CT Connecticut	KY Kentucky	NC North Carolina	OR Oregon	WA Washington
DE Delaware	LA Louisiana	ND North Dakota	PA Pennsylvania	WI Wisconsin
FL Florida	MA Massachusetts	NE Nebraska	RI Rhode Island	WV West Virginia
GA Georgia	MD Maryland	NH New Hampshire	SC South Carolina	WY Wyoming

An Illustrative Mapping Application using vector

- Suppose that you want to write a program that displays the name of a US state given its two-letter postal abbreviation.
- This program is an ideal application for the **StringMap** class because what you need is a map between two-letter codes and state names. Each two-letter code uniquely identifies a particular state and therefore serves as a key for the **StringMap**; the state names are the corresponding values.
- To implement this program in C++, you need to perform the following steps, which are illustrated on the following slide:
 1. Create a **StringMap** containing the key/value pairs.
 2. Read in the two-letter abbreviation to translate.
 3. Call **get** on the **StringMap** to find the state name.
 4. Print out the name of the state.

The Vector-Based `stringmap` Interface

```
/* Private section */  
  
private:  
  
/*  
 * Type: KeyValuePair  
 * -----  
 * This type combines a key and a value into a single structure.  
 */  
  
    struct KeyValuePair {  
        std::string key;  
        std::string value;  
    };  
  
/* Instance variables */  
  
    Vector<KeyValuePair> bindings;  
  
/* Private function prototypes */  
  
    int findKey(const std::string & key) const;  
  
};  
  
#endif
```



`vector< pair<string, string> > bindings;`

Vector-Based Code for StringMap

```
/*  
 * Private method: findKey  
 * -----  
 * Returns the index at which the key appears, or -1 if the key is not found.  
 */  
  
int StringMap::findKey(const string & key) const {  
    for (int i = 0; i < bindings.size(); i++) {  
        if (bindings.get(i).key == key) return i;  
    }  
    return -1;  
}
```

```
size_t findKey(const string & key) const {  
    for (auto i=0; i < bindings.size(); i++) {  
        if (bindings[i].first == key)  
            return i;  
    }  
    return SIZE_MAX;  
}
```

Vector-Based Code for StringMap

```
/*
 * Implementation notes: put, get
 * -----
 * These methods use findKey to search for the specified key.
 */

string StringMap::get(const string & key) const {
    int index = findKey(key);
    return (index == -1) ? "" : bindings.get(index).value;
}

void StringMap::put(const string & key, const string & value) {
    int index = findKey(key);
    if (index == -1) {
        KeyValuePair entry;
        entry.key = key;
        index = bindings.size();
        bindings.add(entry);
    }
    bindings[index].value = value;
}
```


C++ standard

```
void put(const string & key, const string & value) {  
    auto idx = findKey(key);  
    if (idx == SIZE_MAX)  
        bindings.push_back( pair {key, value} );  
    else  
        bindings[idx].second = value;  
}  
  
string get(const string & key) const {  
    auto idx = findKey(key);  
    return (idx == SIZE_MAX) ? "" : bindings[idx].second;  
}
```

Element access:

<u>operator[]</u>	Access element (public member function)
<u>at</u>	Access element (public member function)
<u>front</u>	Access first element (public member function)
<u>back</u>	Access last element (public member function)
<u>data</u>	Access data (public member function)

Modifiers:

<u>assign</u>	Assign vector content (public member function)
<u>push_back</u>	Add element at the end (public member function)
<u>pop_back</u>	Delete last element (public member function)
<u>insert</u>	Insert elements (public member function)
<u>erase</u>	Erase elements (public member function)
<u>swap</u>	Swap content (public member function)
<u>clear</u>	Clear content (public member function)
<u>emplace</u>	Construct and insert element (public member function)
<u>emplace_back</u>	Construct and insert element at the end (public member function)

Implementation Strategies for Maps

- There are several strategies you might choose to implement the map operations **get** and **put**. Those strategies include:
 1. **Linear search**. Keep track of all the key/value pairs in a vector. In this model, both the **get** and **put** operations run in $O(N)$ time.
 2. **Binary search**. If you keep the vector sorted by the key, you can use binary search to find the key. Using this strategy improves the performance of **get** to...?
 $O(\log N)$. But to keep the vector sorted, **put** runs in...?
 $O(N)$. Why?
 3. **Table lookup in a grid**. In this specific example, you can store the state names (two-character code) in a 26×26 **Grid<string>** in which the first and second indices correspond to the two letters in the code. Because you can now find any code in a single step, this strategy is $O(1)$, although this performance comes at a cost in memory space (only 50/676 occupied).

The Lookup-Table Strategy

	A	B	C	D	E	F
A						
B						
C	California					
D					Delaware	
E						
F						
G	Georgia					
H						
I	Iowa			Idaho		
J						
K						
L	Louisiana					
M	Massachusetts			Maryland	Maine	

The Lookup-Table Strategy

- Why is it $O(1)$?

```
string getStateName(string key, Grid<string> & grid) {  
    char row = key[0] - 'A';  
    char col = key[1] - 'A';  
    if (!grid.inBounds(row, col) || grid[row][col] == "") {  
        error("No state name for " + abbr);  
    }  
    return grid[row][col];  
}
```

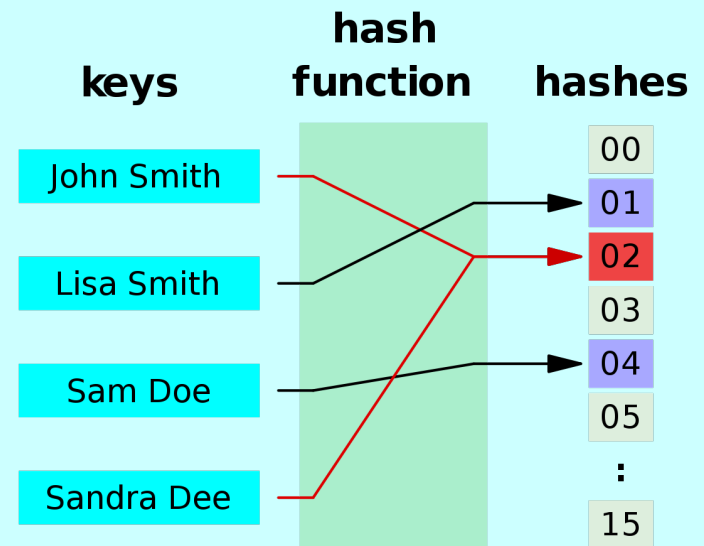
The Idea of Hashing

- The lookup-table strategy from the preceding slide shows that one can make the **get** and **put** operations run very quickly, even to the point that the cost of finding a key is independent of the number of keys in the table. This $O(1)$ performance is possible only if **you know where to look for a particular key**.
- To get a sense of how you might achieve this goal in practice, it helps to think about how you find a word in a dictionary. Most dictionaries have **thumb index** that indicate where each letter appear. Words starting with *A* are in the *A* section, and so on.
- The most common implementations of maps use a strategy called **hashing**, which is conceptually similar to the thumb index in a dictionary. **The critical idea is that you can improve performance enormously if you use the key to figure out where to look.**



Hash Maps

- The **StringMap** class implemented using the hashing strategy is called a *hash map* (sometimes a *hash table*).
- The implementation requires the existence of a free function (called a *hash function*) that transforms a key into a non-negative integer (called a *hash code*), preferably in constant time. The hash code tells the implementation where it should look for a particular key, thereby reducing the search time dramatically.
- The important things to remember about hash codes are:
 1. From a certain hash function, every key has a hash code, and only one hash code.
 2. If two keys are different, they ideally should not, but practically could, **have the same hash code (called collision)**.



Hash Functions

- One extreme example for a hash function is the function that transforms anything to 0. In that case, the hash map turns back into a regular map.
- The other extreme example is the function that transforms the N different keys exactly to $0, 1, \dots, N-1$ (mapping keys to indices) in $O(1)$ time. But imaging how difficult it would be for you to find such a function (see [perfect hash function](#)).
- A simple hash function that simulates the idea of the thumb index in a dictionary:

```
int page = thumbIndexVector[key[0] - 'A'];
```

- Is this hash function good? Why?



Hash Functions

- To achieve the high level of efficiency that hashing offers, a hash function must have the following two properties:
 1. The function must be **inexpensive to compute**.
 2. The function should **distribute keys as uniformly as possible** across the integer range, to minimize **collisions**.
- Hash functions are extremely subtle and often depend on sophisticated mathematics. Most implementations use techniques similar to those for generating pseudorandom numbers, to ensure that the results are **hard to predict**, hence are unlikely to exhibit any higher level of collision than one would expect by chance (e.g., thumb index collisions are not by chance).
- Fortunately, **the correctness of the algorithm is not affected by the collision rate**. Implementations that use poorly designed hash functions run more slowly but nonetheless continue to give correct results.

Example - hash function

```
int hashCode(const string & str) {  
    int hash = 0;  
    int n = str.length();  
    for (int i = 0; i < n; i++) {  
        hash += str[i];  
    }  
    return hash;  
}
```

The hashCode Function for Strings

```
const int HASH_SEED = 5381; /* Starting point for first cycle */
const int HASH_MULTIPLIER = 33; /* Multiplier for each cycle */
const int HASH_MASK = unsigned(-1) >> 1; /* Largest positive integer */

/*
 * Function: hashCode
 * Usage: int code = hashCode(key);
 * -----
 * This function takes a string key and uses it to derive a hash code,
 * which is nonnegative integer related to the key by a deterministic
 * function that distributes keys well across the space of integers.
 * The specific algorithm used here is called djb2 after the initials
 * of its inventor, Daniel J. Bernstein, Professor of Mathematics at
 * the University of Illinois at Chicago.
 */

int hashCode(const string & str) {
    unsigned hash = HASH_SEED;
    int nchars = str.length();
    for (int i = 0; i < nchars; i++) {
        hash = HASH_MULTIPLIER * hash + str[i];
    }
    return (hash & HASH_MASK);
}
```

Things that you only need to know that you don't know. I don't need you to memorize it for the exam.

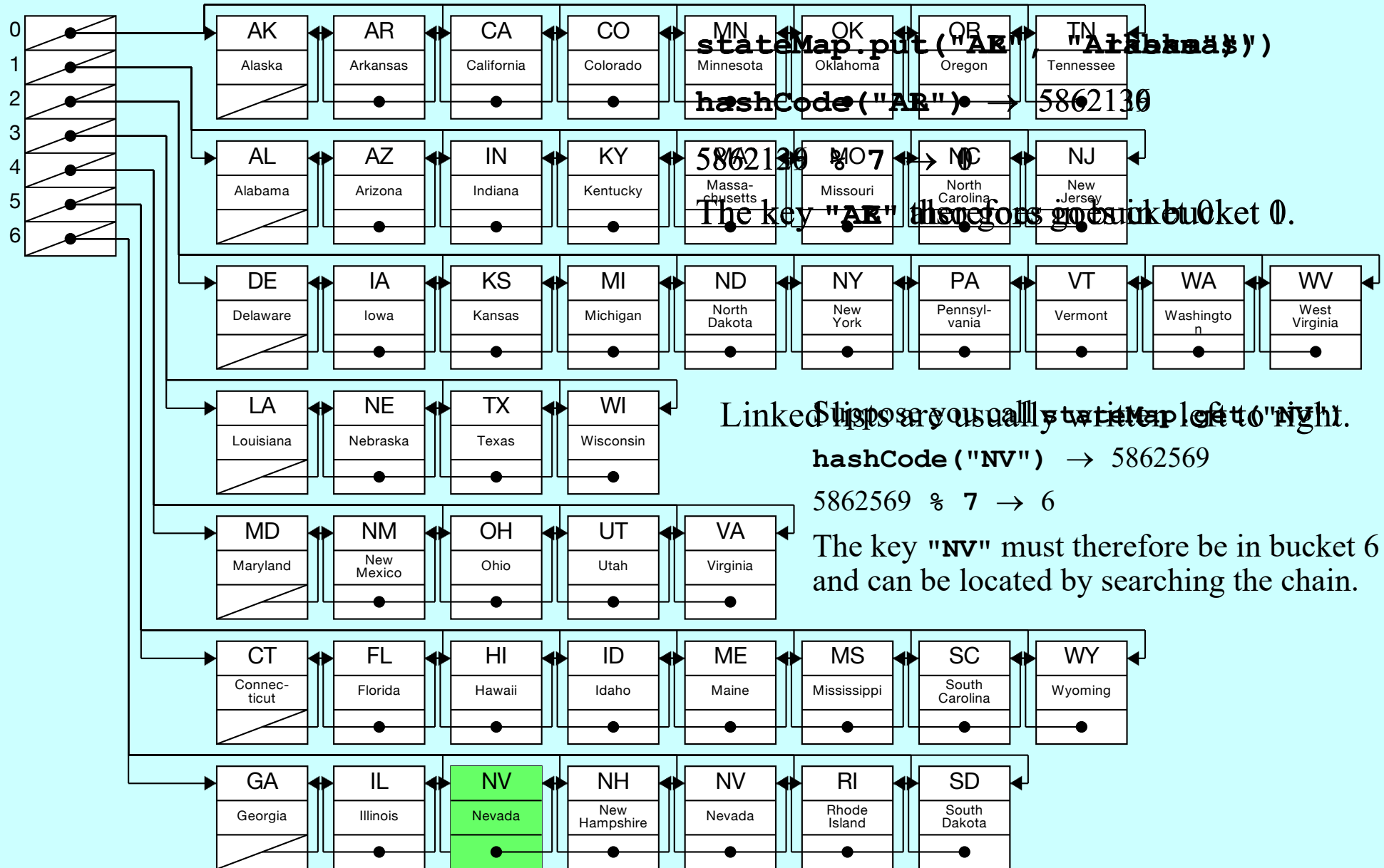
The Bucket Hashing Strategy

- One common strategy for implementing a map is to **use the hash code for each key to select an index into an array** that will contain all the keys with that hash code. Each element of that array is conventionally called a ***bucket***.
- In practice, **the array of buckets is smaller than the number of hash codes**, making it necessary to convert the hash code into a bucket index, typically by executing a statement like

```
int index = hashCode(key) % nBuckets;
```

- The value in each element of the bucket array cannot be a single key/value pair given the chance that different keys fall into the same bucket, i.e., ***collisions***.
- To take account of the possibility of collisions, each element of the bucket array points to a linked list of key/value pairs whose keys fall into that bucket (called **separate chaining**) as illustrated on the next slide. Another strategy, called **open addressing**, uses a single array (Ch.15 Ex.9&10).

Simulating Bucket Hashing

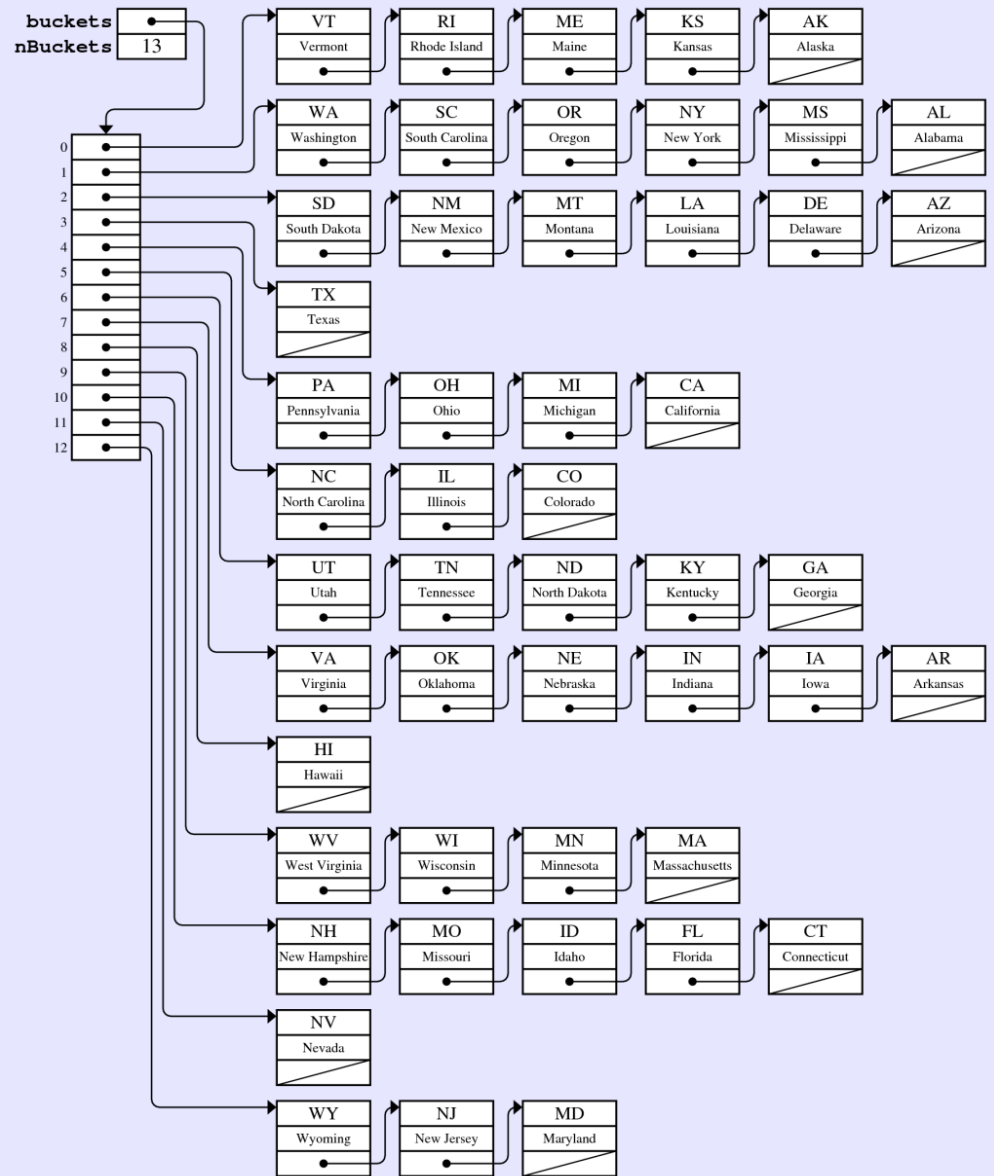


Achieving $O(1)$ Performance

- The simulation on the previous slide uses only seven buckets to emphasize what happens when collisions occur: the smaller the number of buckets, the more likely collisions become.
- In practice, the implementation of **StringMap** would use a much larger value for **nBuckets** to minimize the opportunity for collisions. If the number of buckets is considerably larger than the number of keys, most of the bucket chains will either be empty or contain exactly one key/value pair.
- The ratio of the number of keys to the number of buckets is called the *load factor* of the map. Because a map achieves $O(1)$ performance only if the load factor is small, the library implementation of **HashMap** increases the number of buckets when the map becomes too full. This process is called *rehashing*.
- $O(1)$ is achieved on average, and the worst case is still $O(N)$.

Rehashing

FIGURE 15-9 Hash table containing the state abbreviations



Private Section of the `StringMap` Class

```
/* Private section */

private:

/* Type definition for cells in the bucket chain */

    struct Cell {
        std::string key;
        std::string value;
        Cell* link;
    };

/* Instance variables */

    Cell* *buckets;      /* Dynamic array of pointers to cells */
    int nBuckets;         /* The number of buckets in the array */
    int count;            /* The number of entries in the map */

/* Private method prototypes */

    Cell* findCell(int bucket, std::string key);

/* Make copying illegal */

    StringMap(const StringMap & src) { }
    StringMap & operator=(const StringMap & src) { return *this; }
```

The `stringmap.cpp` Implementation

```
/*
 * File: stringmap.cpp
 * -----
 * This file implements the stringmap.h interface using a hash table
 * as the underlying representation.
 */

#include <string>
#include "stringmap.h"
using namespace std;

/*
 * Implementation notes: StringMap constructor and destructor
 * -----
 * The constructor allocates the array of buckets and initializes each
 * bucket to the empty list. The destructor frees the allocated cells.
 */

StringMap::StringMap() {
    nBuckets = INITIAL_BUCKET_COUNT;
    buckets = new Cell*[nBuckets];
    for (int i = 0; i < nBuckets; i++) {
        buckets[i] = NULL;
    }
}
```

The `stringmap.cpp` Implementation

```
StringMap::~~StringMap() {
    for (int i = 0; i < nBuckets; i++) {
        Cell *cp = buckets[i];
        while (cp != NULL) {
            Cell *oldCell = cp;
            cp = cp->link;
            delete oldCell;
        }
    }
    delete [] buckets;
}

/*
 * Implementation notes: get
 * -----
 * This method calls findCell to search the linked list for the matching
 * key.  If no key is found, get returns the empty string.
 */

string StringMap::get(const string & key) const {
    Cell *cp = findCell(hashCode(key) % nBuckets, key);
    return (cp == NULL) ? "" : cp->value;
}
```

The `stringmap.cpp` Implementation

```
/*
 * Implementation notes: put
 * -----
 * The put method calls findCell to search the linked list for the
 * matching key. If a cell already exists, put simply resets the
 * value field. If no matching key is found, put adds a new cell
 * to the beginning of the list for that chain.
 */

void StringMap::put(const string & key, const string & value) {
    int bucket = hashCode(key) % nBuckets;
    Cell *cp = findCell(bucket, key);
    if (cp == NULL) {
        cp = new Cell;
        cp->key = key;
        cp->link = buckets[bucket];
        buckets[bucket] = cp;
    }
    cp->value = value;
}
```

The `stringmap.cpp` Implementation

```
/*
 * Private method: findCell
 * Usage: Cell *cp = findCell(bucket, key);
 * -----
 * Finds a cell in the chain for the specified bucket that matches key.
 * If a match is found, the return value is a pointer to the cell
 * containing the matching key. If no match is found, findCell
 * returns NULL.
 */

StringMap::Cell *StringMap::findCell(int bucket, const string & key) const {
    Cell *cp = buckets[bucket];
    while (cp != NULL && key != cp->key) {
        cp = cp->link;
    }
    return cp;
}
```



TUTORIAL

Implementing the **HashMap** class

To implement a more general **HashMap** (`unordered_map` in C++), start from **StringMap**:

Implementing the missing methods for maps like **size**, **isEmpty**, **containsKey**, **remove**, and **clear**.

- Generalizing the key and value types using templates.
- The algorithm for implementing hash maps imposes several requirements on the type used to represent keys, as follows:
 - The key type must be assignable so that the code can store copies of the keys in the cells.
 - The key type must support the comparison operator `==` so that the code can tell whether two keys are identical.
 - At the time the template for **HashMap** is expanded for a specific key type defined by the client, other than the built-in types like **string** and **int**, a version of the **hashCode** function that produces a non-negative integer for every value of the key type must also be **provided by the client**.

The End