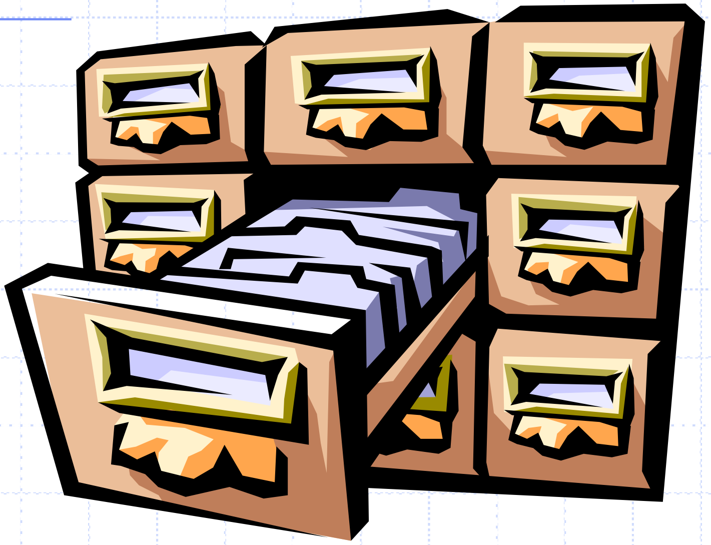
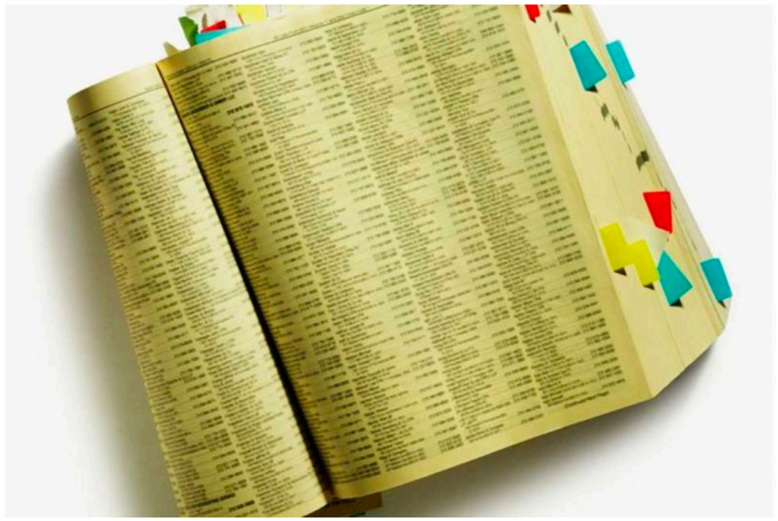
Maps



Maps

- A map models a searchable collection of key-value entries
- The main operations of a map are for searching, inserting, and deleting items
- Multiple entries with the same key are not allowed
- Applications:
  - address book (yellowpage)
  - student-record database



Entry ADT

- An entry stores a key-value pair (k,v)
- Methods:
  - key(): return the associated key
  - value(): return the associated value
  - setKey(k): set the key to k
  - setValue(v): set the value to v
- We call this “item” or “element” or “record” exchangeably.
- Then, MAP stores multiple a collection of Entries

The Map ADT

- find(k): if the map M has an entry with key k, return and iterator to it; else, return special iterator end
- put(k, v): if there is no entry with key k, insert entry (k, v), and otherwise set its value to v. Return an iterator to the new/modified entry
- erase(k): if the map M has an entry with key k, remove it from M

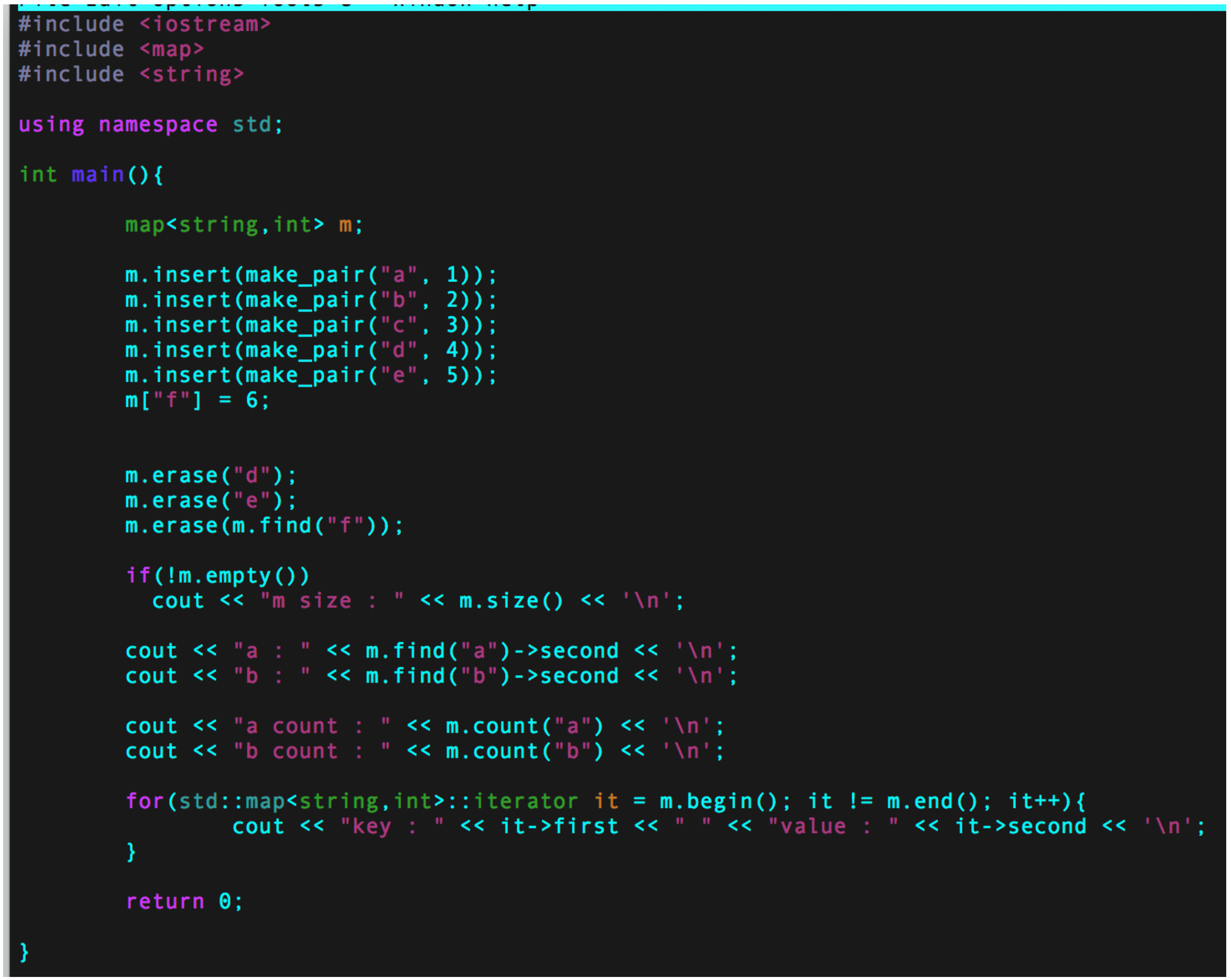
- size(), empty()
- begin(), end(): return iterators to beginning and end of M

- Important Issue?
  Using what data structure and algorithm, we implement Map?

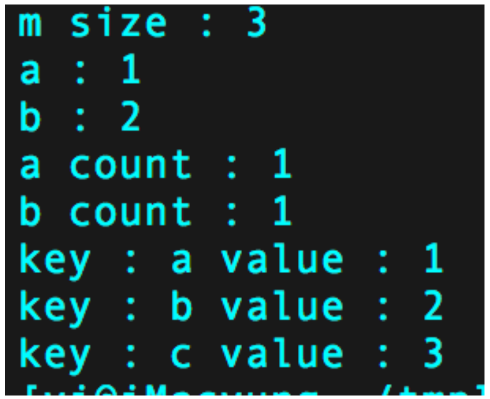
Example

Operation	***Output***	***Map***
empty()	**true**	Ø
put(5,A)	[(5,A)]	(5,A)
put(7,B)	[(7,B)]	(5,A),(7,B)
put(2,C)	[(2,C)]	(5,A),(7,B),(2,C)
put(8,D)	[(8,D)]	(5,A),(7,B),(2,C),(8,D)
put(2,E)	[(2,E)]	(5,A),(7,B),(2,E),(8,D)
find(7)	[(7,B)]	(5,A),(7,B),(2,E),(8,D)
find(4)	end	(5,A),(7,B),(2,E),(8,D)
find(2)	[(2,E)]	(5,A),(7,B),(2,E),(8,D)
size()	4	(5,A),(7,B),(2,E),(8,D)
erase(5)	—	(7,B),(2,E),(8,D)
erase(2)	—	(7,B),(8,D)
find(2)	end	(7,B),(8,D)
empty()	**false**	(7,B),(8,D)

map in C++



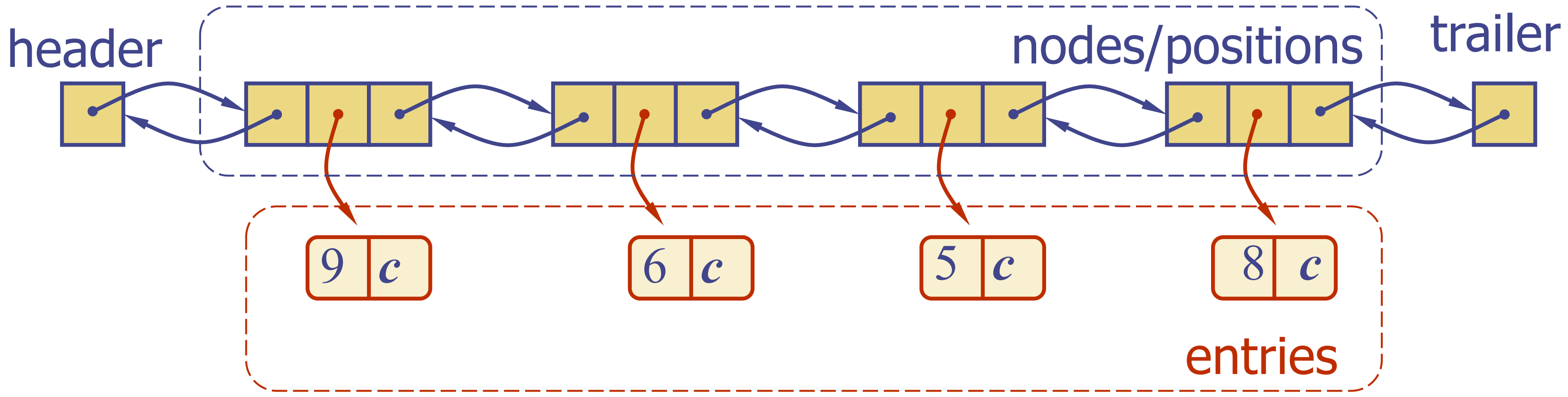
```cpp
#include <iostream>
#include <map>
#include <string>

using namespace std;

int main(){

        map<string,int> m;

        m.insert(make_pair("a", 1));
        m.insert(make_pair("b", 2));
        m.insert(make_pair("c", 3));
        m.insert(make_pair("d", 4));
        m.insert(make_pair("e", 5));
        m["f"] = 6;


        m.erase("d");
        m.erase("e");
        m.erase(m.find("f"));

        if(!m.empty())
          cout << "m size : " << m.size() << '\n';

        cout << "a : " << m.find("a")->second << '\n';
        cout << "b : " << m.find("b")->second << '\n';

        cout << "a count : " << m.count("a") << '\n';
        cout << "b count : " << m.count("b") << '\n';

        for(std::map<string,int>::iterator it = m.begin(); it != m.end(); it++){
                cout << "key : " << it->first << " " << "value : " << it->second << '\n';
        }

        return 0;

}
```



```
m size : 3
a : 1
b : 2
a count : 1
b count : 1
key : a value : 1
key : b value : 2
key : c value : 3
```

A Simple List-Based Map

- An easiest way of implementing Map
- We can efficiently implement a map using an unsorted list
  - We store the items of the map in a list S (based on a doubly-linked list), in arbitrary order



The find Algorithm

```
Algorithm find(k):
    for each p in [S.begin(), S.end()) do
        if p→key() = k then
            return p
    return S.end() {there is no entry with key equal to k}
```

We use p→key() as a shortcut for (*p).key()

The put Algorithm

```
Algorithm put(k,v):
    for each p in [S.begin(), S.end()) do
        if p→key() = k then
            p→setValue(v)
            return p
    p = S.insertBack((k,v)) {there is no entry with key k}
    n = n + 1    {increment number of entries}
    return p
```

The erase Algorithm

```
Algorithm erase(k):
    for each p in [S.begin(), S.end()) do
        if p.key() = k  then
            S.erase(p)
            n = n – 1         {decrement number of entries}
```

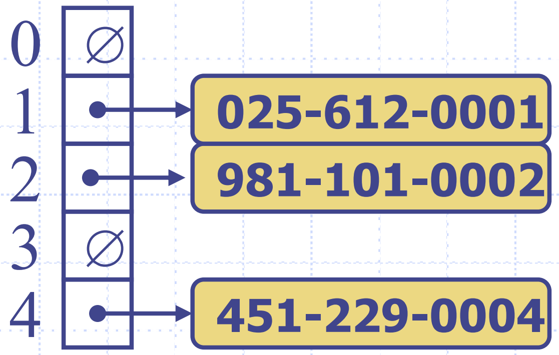
Performance of a List-Based Map

- Performance:
  - put takes $O(n)$ time since we need to determine whether it is already in the sequence
  - find and erase take $O(n)$ time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key

- The unsorted list implementation is effective only for maps of small size or for maps in which puts are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)

- Can we improve?

Hash Tables



Recall the Map ADT

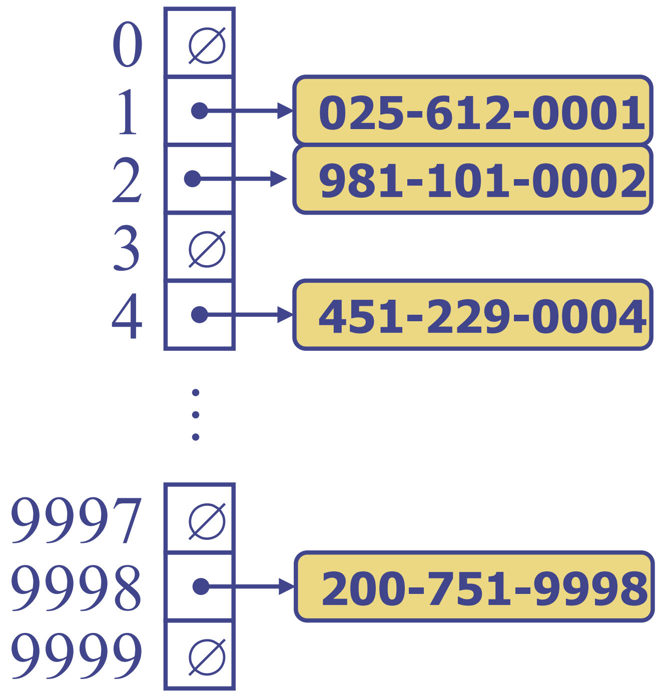
- find(k): if the map M has an entry with key k, return its associated value; else, return null
- put(k, v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
- erase(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null
- size(), empty()
- entrySet(): return a list of the entries in M
- keySet(): return a list of the keys in M
- values(): return a list of the values in M

What about this idea?

- We design a table for a map storing entries as (SSN, Name), where SSN (social security number) is a nine-digit positive integer

- Our table uses an array of size $N = 10{,}000$ and classify each person based on the last four digits of his/her SSN

- ***Do you think that we can speed up searching using this method?***



Hash Table: Overview

- Use key as an “address” for a value

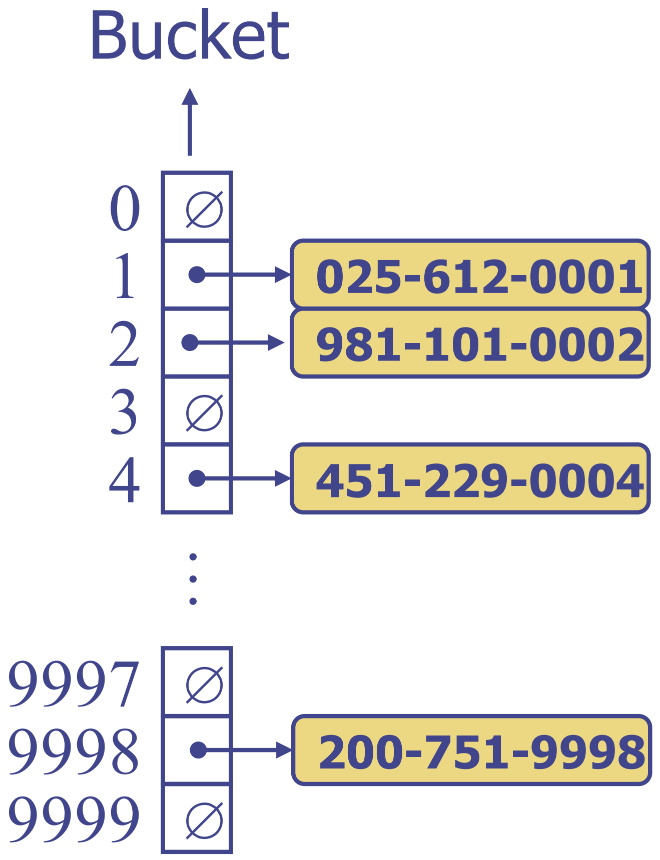
- Worst-case performance: still O(n)
- But, usually expected performance: O(1)
  - Practically very fast

- Consists of two major components
  - 1. Bucket array
  - 2. Hash function

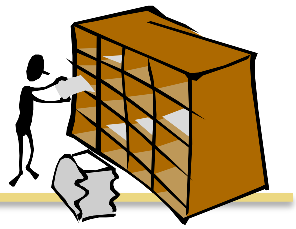
Bucket Array

- An array of size N, where each cell of A is “bucket”

- Two Issues
  - How to choose the bucket size N?
    - Large N?
    - Small N?
  - Keys should be integers, but in practice, not always.
    - Key: string “yiyung”



Hash Functions and Hash Tables



- A hash function h maps keys of a given type to integers in a fixed interval $[0, N - 1]$
- Example:

  $$h(x) = x \bmod N$$

  is a hash function for integer keys
- The integer $h(x)$ is called the hash value of key x

- A hash table for a given key type consists of
  - Hash function h
  - Array (called table or bucket array) of size N
- When implementing a map with a hash table, the goal is to store item (k, o) at index $i = h(k)$

Hash Functions

- A hash function is usually specified as the composition of two functions:

  Hash code:

  h_1: keys $\rightarrow$ integers

  Compression function:

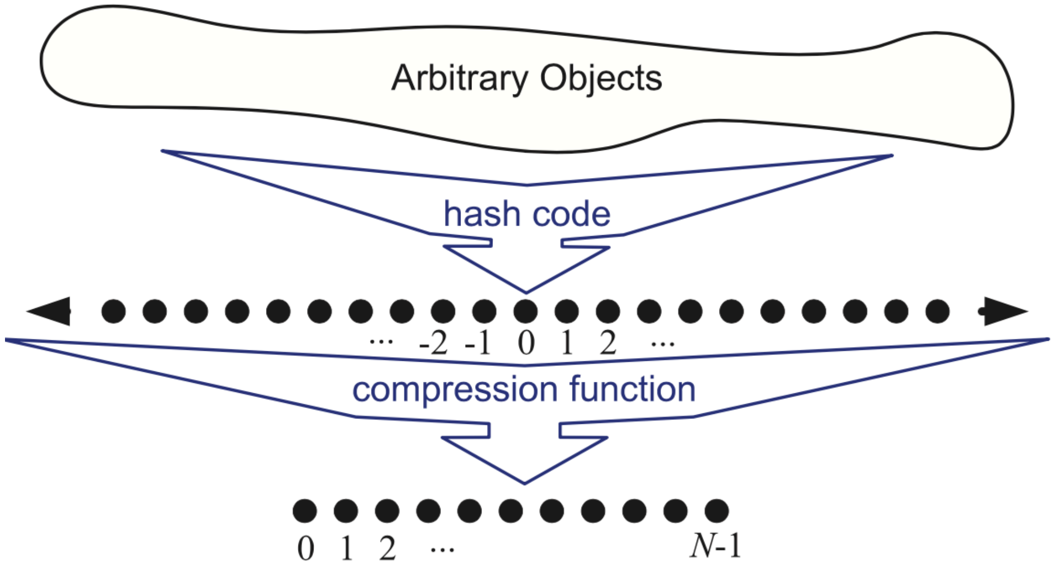
  h_2: integers $\rightarrow$ [0, N - 1]

(Note) Keys can be arbitrary objects, e.g., string “yiyung”

- The hash code is applied first, and the compression function is applied next on the result, i.e.,

$$h(x) = h_2(h_1(x))$$

- The goal of the hash function is to “disperse” the keys in an apparently random way

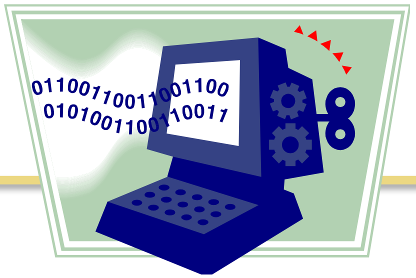


Hash Code and Compress Function

- There are extensive theoretical and experiment research about “good” hash code and compress functions

- In the next 3 slides,
  - We will discuss some basic hash codes and compress functions.
  - Looking at their more details is not the beyond of our scope.

(1) Hash Codes



- Memory address:
  - We reinterpret the memory address of the key object as an integer
- Integer cast:
  - We reinterpret the bits of the key as an integer
  - Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., byte, short, int and float in C++)
- Component sum:
  - We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum the components (ignoring overflows)
  - Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type (e.g., long and double in C++)
  - But, not good for strings
    - "temp01" and "temp10"
    - "stop", "tops", "pots", "spot"

Polynomial Hash Code

- Polynomial accumulation:
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)

    $$a_0\, a_1 \ldots a_{n-1}$$

  - We evaluate the polynomial

    $$p(z) = a_0 + a_1 z + a_2 z^2 + \ldots$$
    $$\ldots + a_{n-1} z^{n-1}$$

    at a fixed value z, ignoring overflows

  - Especially suitable for strings (e.g., the choice $z = 33$ gives at most 6 collisions on a set of 50,000 English words)

- Polynomial $p(z)$ can be evaluated in $O(n)$ time using Horner's rule:
  - The following polynomials are successively computed, each from the previous one in $O(1)$ time

    $$p_0(z) = a_{n-1}$$

    $$p_i(z) = a_{n-i-1} + z p_{i-1}(z)$$
    $$(i = 1, 2, \ldots, n - 1)$$

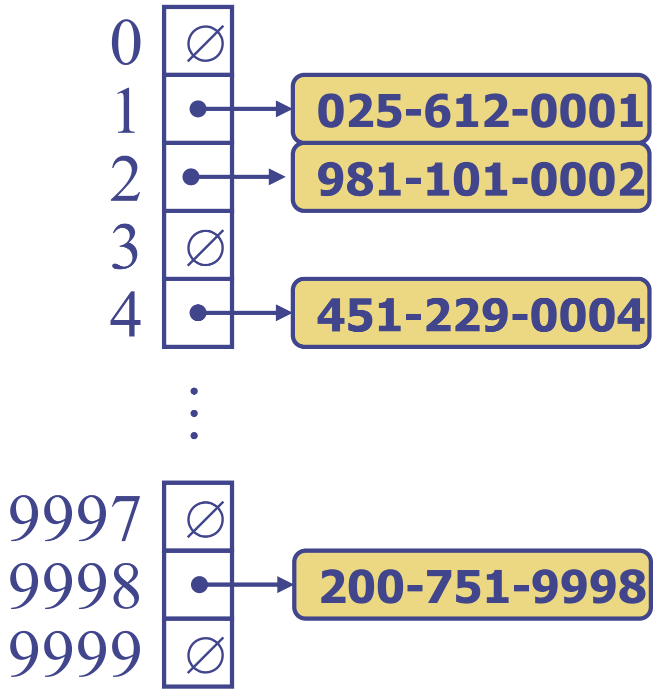
- We have $p(z) = p_{n-1}(z)$

- Lots of research about "good hash code"

(2) Compression Functions

- Division:
  - $h_2(y) = |y| \bmod N$
  - The size N of the hash table is usually chosen to be a prime
  - The reason has to do with number theory and is beyond the scope of this course
- Multiply, Add and Divide (MAD):
  - $h_2(y) = |ay + b| \bmod N$
  - a and b are nonnegative integers such that $a \bmod N \neq 0$
    - Otherwise, every integer would map to the same value b

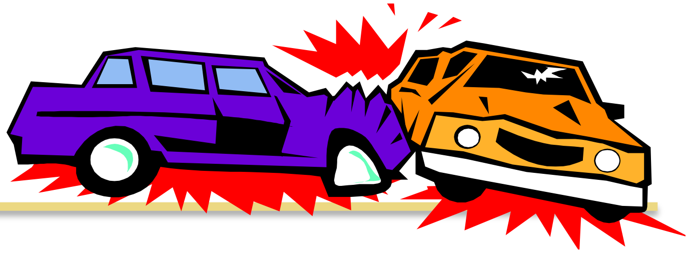
Collision Handling



Insert the entry: 032-637-0004

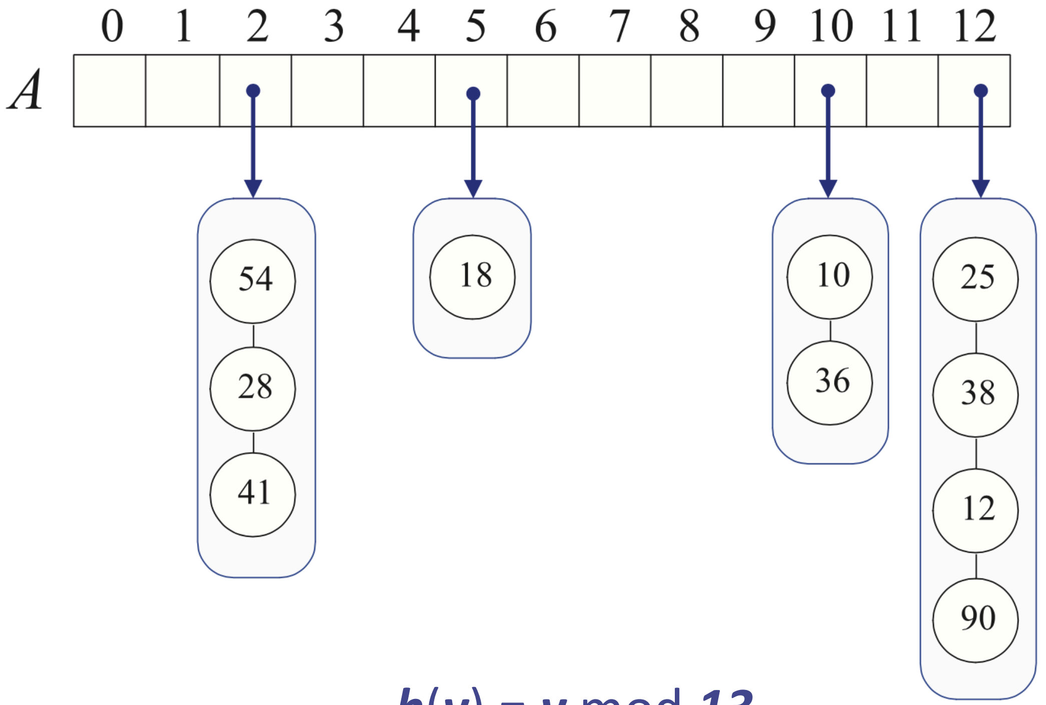
Collisions occur when different elements are mapped to the same cell

Collision Handling



- Separate Chaining: let each cell in the table point to a linked list of entries that map there

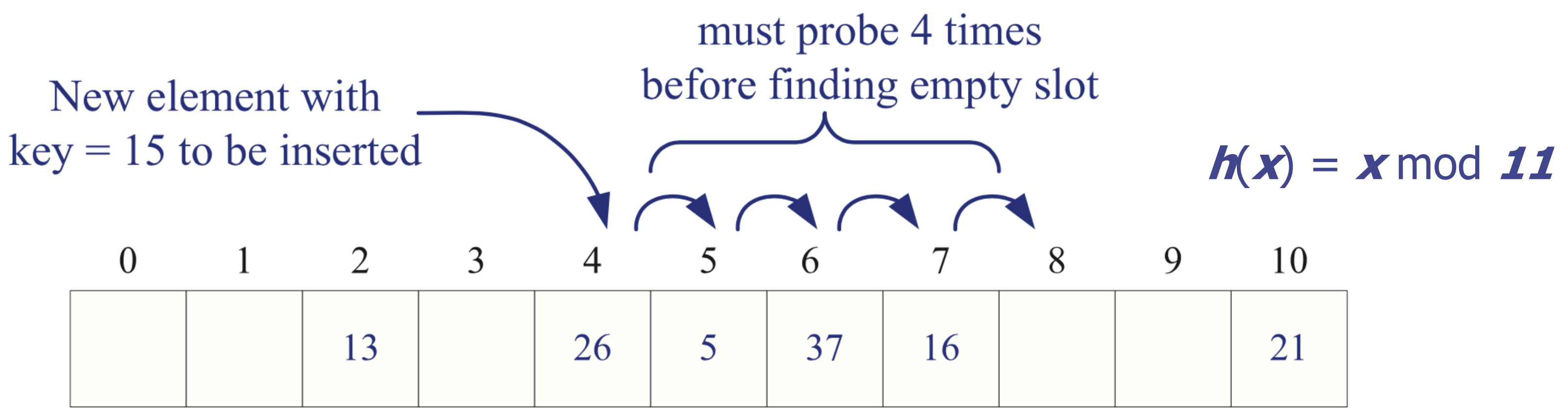
- Separate chaining is simple, but requires additional memory outside the table



$h(y) = y \bmod 13$

Open Addressing: Linear Probing

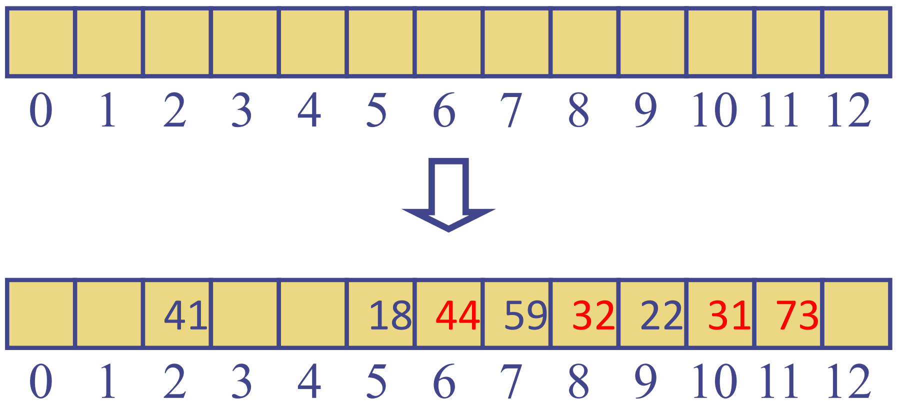
- Open addressing: the colliding item is placed in a different cell of the table
- Linear probing: handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a "probe"
- Colliding items lump together, causing future collisions to cause a longer sequence of probes



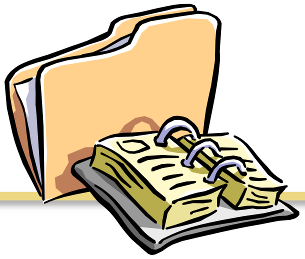
must probe 4 times before finding empty slot
New element with key = 15 to be inserted
$h(x) = x \bmod 11$
0 1 2 3 4 5 6 7 8 9 10
13
26
5
37
16
21

Linear Probing: Example

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



Search with Linear Probing



- Consider a hash table ***A*** that uses linear probing
- find(***k***)
  - We start at cell ***h***(***k***)
  - We probe consecutive locations until one of the following occurs
    - An item with key ***k*** is found, or
    - An empty cell is found, or
    - ***N*** cells have been unsuccessfully probed

```
Algorithm find(k)
    i ← h(k)
    p ← 0
    repeat
        c ← A[i]
        if c = ∅
            return null
        else if c.key() = k
            return c.value()
        else
            i ← (i + 1) mod N
            p ← p + 1
    until p = N
    return null
```

Updates with Linear Probing

- To handle insertions and deletions, we introduce a special marker, called ***AVAILABLE***, which replaces deleted elements
  - Avoids a lot of shift operations
- erase(***k***)
  - We search for an entry with key ***k***
  - If such an entry (***k, o***) is found, we replace it with the special item ***AVAILABLE*** and we return element ***o***
  - Else, we return ***null***
- put(***k, o***)
  - We throw an exception if the table is full
  - We start at cell ***h***(***k***)
  - We probe consecutive cells until one of the following occurs
    - A cell ***i*** is found that is either empty or stores ***AVAILABLE***, or
    - ***N*** cells have been unsuccessfully probed
  - We store (***k, o***) in cell ***i***

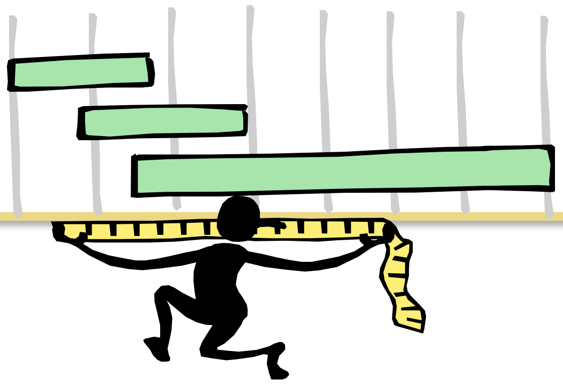
Other Issues

- Search with Linear Probing
  - Clustering problem
- Other open addressing method
  - Quadratic Probing, Double Hashing (the details in the book)

- The load factor $\boldsymbol{a} = \boldsymbol{n}/\boldsymbol{N}$ affects the performance of a hash table

- Keeping the load factor below a certain threshold is vital
  - Open addressing (requires $\boldsymbol{a} < 0.5$)
  - Separate-chaining (requires $\boldsymbol{a} < 0.9$)
  - Resize the hash table, i.e., rehashing a new table

Performance of Hashing



- In the worst case, searches, insertions and removals on a hash table take $O(n)$ time
- The worst case occurs when all the keys inserted into the map collide
- The load factor $\alpha = n/N$
- Assuming that the hash values are like random numbers, it can be shown that the expected number of probes for an insertion with open addressing is
  $1 / (1 - \alpha)$
- But, when well designed, the expected running time of all the MAP ADT operations in a hash table is $O(1)$
- In practice, hashing is very fast provided the load factor is not close to 100%

Questions?