Maps and Dictionaries

Data structures and Algorithms

Acknowledgement:
These slides are adapted from slides provided with *Data Structures and Algorithms in C++*
Goodrich, Tamassia and Mount (Wiley, 2004)

Outline

- Maps (9.1)
- Hash tables (9.2)
- Dictionaries (9.3)

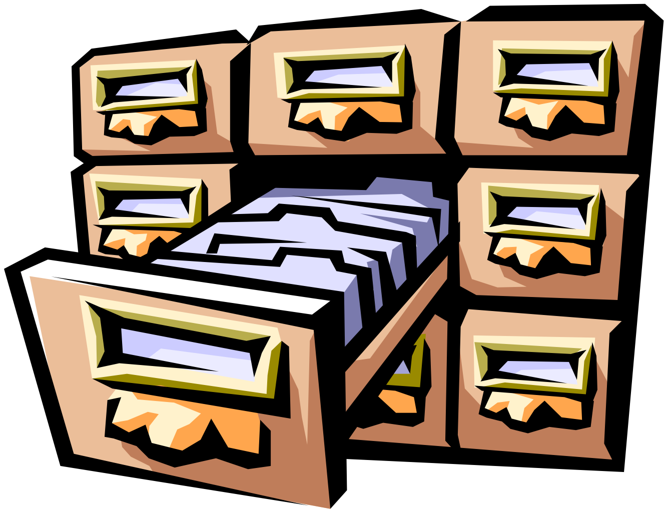
Maps & Dictionaries

- Map ADT and Dictionary ADT:
  - model a searchable collection of key-value entries
  - main operations are searching, inserting, and deleting entries

- Map: multiple entries with the same key are **not** allowed
- Map applications:
  - address book
  - student-record database

- Dictionary: multiple entries with the same key **are** allowed
- Dictionary applications:
  - word-definition pairs
  - credit card authorizations
  - DNS mapping of host names (e.g., datastructures.net) to internet IP addresses (e.g., 128.148.34.101)

Maps



The Map ADT

Map ADT methods:

- get(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
- remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null
- size(), isEmpty()
- keys(): return an iterator of the keys in M
- values(): return an iterator of the values in M

Example

Operation	***Output***	***Map***
isEmpty()	**true**	Ø
put(5,*A*)	**null**	(5,*A*)
put(7,*B*)	**null**	(5,*A*),(7,*B*)
put(2,*C*)	**null**	(5,*A*),(7,*B*),(2,*C*)
put(8,*D*)	**null**	(5,*A*),(7,*B*),(2,*C*),(8,*D*)
put(2,*E*)	*C*	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
get(7)	*B*	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
get(4)	**null**	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
get(2)	*E*	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
size()	4	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
remove(5)	*A*	(7,*B*),(2,*E*),(8,*D*)
remove(2)	*E*	(7,*B*),(8,*D*)
get(2)	**null**	(7,*B*),(8,*D*)
isEmpty()	**false**	(7,*B*),(8,*D*)

http://kengine.sourceforge.net/tutorial/g/stdmap-eng.htm

```
#include <iostream>
#include <map>
#include <string>
using namespace std;

typedef map<string, string> TStrStrMap;
typedef pair<string, string> TStrStrPair;

int main(int argc, char *argv[])
{
    TStrStrMap tMap;

    tMap.insert(TStrStrPair("yes", "no"));
    tMap.insert(TStrStrPair("up", "down"));
    tMap.insert(TStrStrPair("left", "right"));
    tMap.insert(TStrStrPair("good", "bad"));

    string key;
    cout << "Enter word: " << endl;
    cin >> key;
```

```
    string strValue = tMap[key];
    if(strValue!="")
        cout << "Opposite: " << strValue << endl;   // Show value
    else
    {
        TStrStrMap::iterator p;
        bool bFound=false;
        // Show key
        for(p = tMap.begin(); p!=tMap.end(); ++p) {
                string strKey= p->second;
                if( key == strKey)  {
                        // Return key
                        std::cout << "Opposite: " << p->first << std::endl;
                        bFound = true;
                }
        }
        if(!bFound)                     // If not found opposite word
                cout << "Word not in map." << endl;
    }
    return 0;
}
```

Dictionary ADT

- The dictionary ADT models a searchable collection of key-value entries: ordered and unordered.
- The main operations of a dictionary are searching, inserting, and deleting items
- Multiple items with the same key **are** allowed
- Applications:
  - word-definition pairs
  - credit card authorizations
  - DNS mapping of host names (e.g., datastructures.net) to internet IP addresses (e.g., 128.148.34.101)
- Dictionary ADT methods:
  - find(k): if the dictionary has an entry with key k, returns it, else, returns null
  - findAll(k): returns an iterator of all entries with key k
  - insert(k, o): inserts and returns the entry (k, o)
  - remove(e): remove the entry e from the dictionary
  - entries(): returns an iterator of the entries in the dictionary
  - size(), isEmpty()

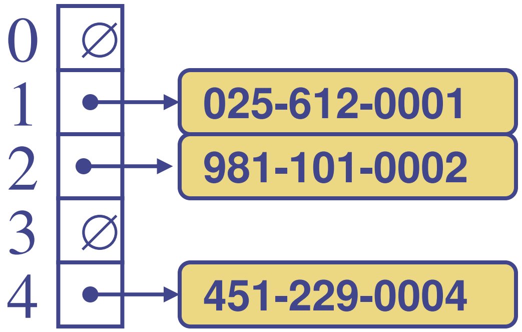
Example

Operation	***Output***	***Dictionary***
insert(5,*A*)	(5,*A*)	(5,*A*)
insert(7,*B*)	(7,*B*)	(5,*A*),(7,*B*)
insert(2,*C*)	(2,*C*)	(5,*A*),(7,*B*),(2,*C*)
insert(8,*D*)	(8,*D*)	(5,*A*),(7,*B*),(2,*C*),(8,*D*)
insert(2,*E*)	(2,*E*)	(5,*A*),(7,*B*),(2,*C*),(8,*D*),(2,*E*)
find(7)	(7,*B*)	(5,*A*),(7,*B*),(2,*C*),(8,*D*),(2,*E*)
find(4)	**null**	(5,*A*),(7,*B*),(2,*C*),(8,*D*),(2,*E*)
find(2)	(2,*C*)	(5,*A*),(7,*B*),(2,*C*),(8,*D*),(2,*E*)
findAll(2)	(2,*C*),(2,*E*)	(5,*A*),(7,*B*),(2,*C*),(8,*D*),(2,*E*)
size()	5	(5,*A*),(7,*B*),(2,*C*),(8,*D*),(2,*E*)
remove(find(5))	(5,*A*)	(7,*B*),(2,*C*),(8,*D*),(2,*E*)
find(5)	**null**	(7,*B*),(2,*C*),(8,*D*),(2,*E*)

Implement Dictionary ADT

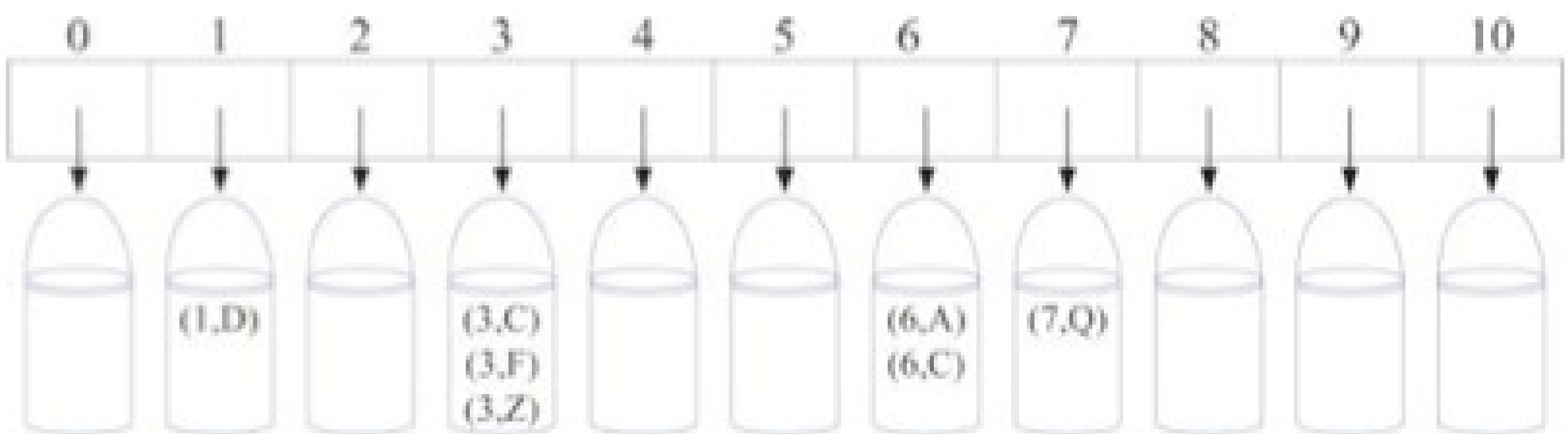
- Unordered dictionary
  - List-based dictionary
  - Hash table
- Ordered dictionary
  - Array-based dictionary – search table
  - Skip list

Hash Tables



Hash table

- Expected time of search, put: $O(1)$
- Bucket array
- Hash function



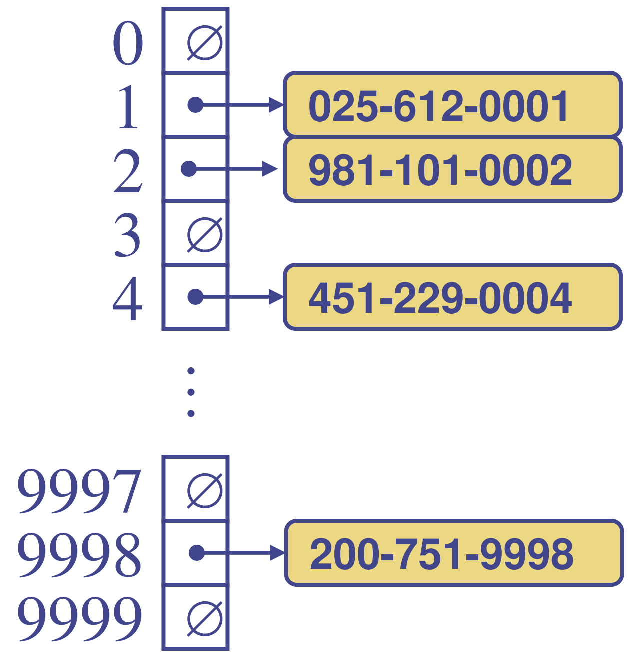
Hash Functions and Hash Tables

- 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) of size N

When implementing a map with a hash table, the goal is to store item (k, o) at index $i = h(x)$

Example

- We design a hash table for a map storing entries as (SSN, Name), where SSN (social security number) is a nine-digit positive integer
- Our hash table uses an array of size $N = 10{,}000$ and the hash function $h(x) = \text{last four digits of } x$



Hash Functions

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

  Hash code:

  h_1: keys $\rightarrow$ integers

  Compression function:

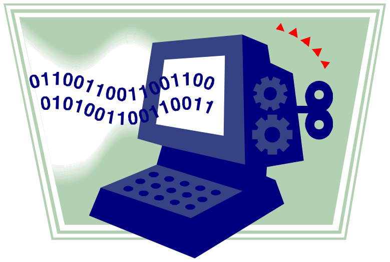
  h_2: integers $\rightarrow [0, N-1]$

- The hash code map is applied first, and the compression map 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
  - minimize collisions

Hash Codes



- Memory address:
  - We reinterpret the memory address of the key object as an integer
  - Good in general, except for numeric and string keys (same key should have the same hash code)
- 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/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)

Hash Codes (cont.)

- Polynomial accumulation:
  - Order is important
  - 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

    $$p(z) = a_{n-1} + a_{n-2} z + a_{n-3} z^2 + \ldots + a_0 z^{n-1}$$

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

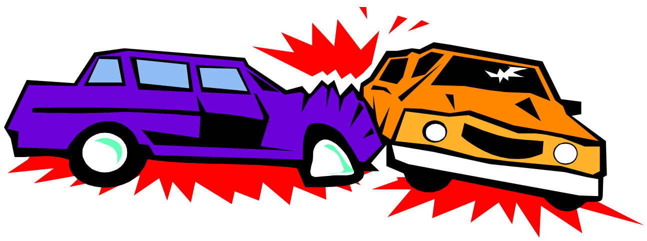
Compression Functions

- Division:
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime
    - Reason: reduce collisions
    - How: number theory and is beyond the scope of this course
- Multiply, Add and Divide (MAD):
  - $h_2(y) = (ay + b) \bmod N$
  - N is prime, a and b are nonnegative integers such that

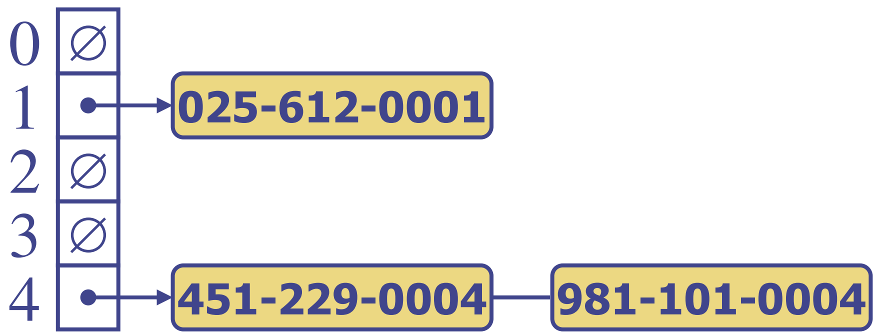
    $a \bmod N \neq 0$

    Otherwise, every integer would map to the same value b

Collision Handling

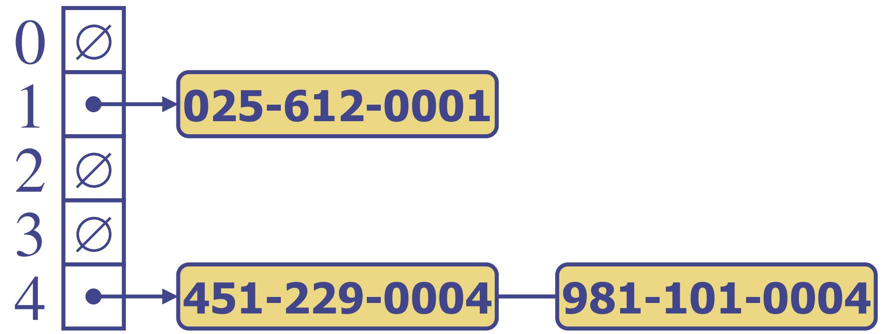


- Collisions occur when different elements are mapped to the same cell
- Ways to handle collisions
  - Separate chaining
  - Linear probing
  - Double hashing



Separate chaining

Separate chaining



- We let each cell in the table point to a linked list of entries that map there
- Load factor: n/N < 1
- Separate chaining is simple, but requires additional memory outside the table
- Example:
  - Assume you have a hash table H with N=9 slots (H[0,8]) and let the hash function be h(k) = k mod N.
  - Demonstrate (by picture) the insertion of the following keys into a hash table with collisions resolved by chaining.
    - 5, 28, 19, 15, 20, 33, 12, 17, 10

Map Methods with Separate Chaining used for Collisions

- Delegate operations to a list-based map at each cell:

Algorithm get(k):
Output: The value associated with the key k in the map, or **null** if there is no entry with key equal to k in the map
return $A[h(k)]$.get(k) {delegate the get to the list-based map at $A[h(k)]$}

Algorithm put(k,v):
Output: If there is an existing entry in our map with key equal to k, then we return its value (replacing it with v); otherwise, we return **null**
$t = A[h(k)]$.put(k,v) {delegate the put to the list-based map at $A[h(k)]$}
if t = **null then** {k is a new key}
  $n = n + 1$
return t

Algorithm remove(k):
Output: The (removed) value associated with key k in the map, or **null** if there is no entry with key equal to k in the map
$t = A[h(k)]$.remove(k) {delegate the remove to the list-based map at $A[h(k)]$}
if $t \neq$ **null then** {k was found}
  $n = n - 1$
return t

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

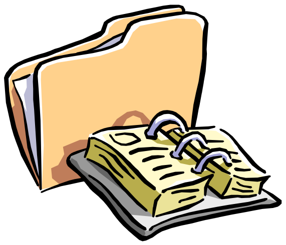
- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

0	1	2	3	4	5	6	7	8	9	10	11	12

⇩

		41			18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12

Search with Linear Probing



- Consider a hash table A that uses linear probing
- get(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 *get*(k)
    $i \leftarrow h(k)$
    $p \leftarrow 0$
    repeat
        $c \leftarrow A[i]$
        if $c = \varnothing$
            return *null*
        else if $c.key() = k$
            return $c.element()$
        else
            $i \leftarrow (i + 1) \bmod N$
            $p \leftarrow p + 1$
    until $p = N$
    return *null*

Updates with Linear Probing

- To handle insertions and deletions, we introduce a special object, called ***AVAILABLE***, which replaces deleted elements
- remove(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 entry (k, o) in cell i

Double Hashing

- Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series
  $h(k,i) = (h(k) + i*d(k)) \bmod N$
  for $i = 0, 1, \dots, N-1$
- The secondary hash function $d(k)$ cannot have zero values
- The table size N must be a prime to allow probing of all the cells

- Common choice of compression function for the secondary hash function:
  $d_2(k) = q - (k \bmod q)$
  where
  - $q < N$
  - q is a prime
- The possible values for $d_2(k)$ are
  $1, 2, \dots, q$

Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $d(k) = 7 - k \bmod 7$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		

0	1	2	3	4	5	6	7	8	9	10	11	12

⇩

31		41			18	32	59	73	22	44		
0	1	2	3	4	5	6	7	8	9	10	11	12

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 $\alpha = n/N$ affects the performance of a hash table
- 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
  $$1 / (1 - \alpha)$$
- The expected running time of all the dictionary ADT operations in a hash table is $O(1)$
- In practice, hashing is very fast provided the load factor is not close to 100%
- Applications of hash tables:
  - small databases
  - compilers
  - browser caches
- Open addressing is not faster than chaining method if space is an issue.

Hash Table Implementation of Dictionary ADT

- Unordered dictionaries.
- We can also create a hash-table dictionary implementation.
- If we use separate chaining to handle collisions, then each operation can be delegated to a list-based dictionary stored at each hash table cell.