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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Output</th>
<th>Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>isEmpty()</td>
<td>true</td>
<td>Ø</td>
</tr>
<tr>
<td>put(5, A)</td>
<td>null</td>
<td>(5, A)</td>
</tr>
<tr>
<td>put(7, B)</td>
<td>null</td>
<td>(5, A), (7, B)</td>
</tr>
<tr>
<td>put(2, C)</td>
<td>null</td>
<td>(5, A), (7, B), (2, C)</td>
</tr>
<tr>
<td>put(8, D)</td>
<td>null</td>
<td>(5, A), (7, B), (2, C), (8, D)</td>
</tr>
<tr>
<td>put(2, E)</td>
<td>C</td>
<td>(5, A), (7, B), (2, E), (8, D)</td>
</tr>
<tr>
<td>get(7)</td>
<td>B</td>
<td>(5, A), (7, B), (2, E), (8, D)</td>
</tr>
<tr>
<td>get(4)</td>
<td>null</td>
<td>(5, A), (7, B), (2, E), (8, D)</td>
</tr>
<tr>
<td>get(2)</td>
<td>E</td>
<td>(5, A), (7, B), (2, E), (8, D)</td>
</tr>
<tr>
<td>size()</td>
<td>4</td>
<td>(5, A), (7, B), (2, E), (8, D)</td>
</tr>
<tr>
<td>remove(5)</td>
<td>A</td>
<td>(7, B), (2, E), (8, D)</td>
</tr>
<tr>
<td>remove(2)</td>
<td>E</td>
<td>(7, B), (8, D)</td>
</tr>
<tr>
<td>get(2)</td>
<td>null</td>
<td>(7, B), (8, D)</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>false</td>
<td>(7, B), (8, D)</td>
</tr>
</tbody>
</table>
// std::map example
// opposite words

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

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Output</th>
<th>Dictionary</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>insert(5, A)</code></td>
<td><code>(5, A)</code></td>
<td><code>(5, A)</code></td>
</tr>
<tr>
<td><code>insert(7, B)</code></td>
<td><code>(7, B)</code></td>
<td><code>(5, A), (7, B)</code></td>
</tr>
<tr>
<td><code>insert(2, C)</code></td>
<td><code>(2, C)</code></td>
<td><code>(5, A), (7, B), (2, C)</code></td>
</tr>
<tr>
<td><code>insert(8, D)</code></td>
<td><code>(8, D)</code></td>
<td><code>(5, A), (7, B), (2, C), (8, D)</code></td>
</tr>
<tr>
<td><code>insert(2, E)</code></td>
<td><code>(2, E)</code></td>
<td><code>(5, A), (7, B), (2, C), (8, D), (2, E)</code></td>
</tr>
<tr>
<td><code>find(7)</code></td>
<td><code>(7, B)</code></td>
<td><code>(5, A), (7, B), (2, C), (8, D), (2, E)</code></td>
</tr>
<tr>
<td><code>find(4)</code></td>
<td><code>null</code></td>
<td><code>(5, A), (7, B), (2, C), (8, D), (2, E)</code></td>
</tr>
<tr>
<td><code>find(2)</code></td>
<td><code>(2, C)</code></td>
<td><code>(5, A), (7, B), (2, C), (8, D), (2, E)</code></td>
</tr>
<tr>
<td><code>findAll(2)</code></td>
<td><code>(2, C), (2, E)</code></td>
<td><code>(5, A), (7, B), (2, C), (8, D), (2, E)</code></td>
</tr>
<tr>
<td><code>size()</code></td>
<td><code>5</code></td>
<td><code>(5, A), (7, B), (2, C), (8, D), (2, E)</code></td>
</tr>
<tr>
<td><code>remove(find(5))</code></td>
<td><code>(5, A)</code></td>
<td><code>(7, B), (2, C), (8, D), (2, E)</code></td>
</tr>
<tr>
<td><code>find(5)</code></td>
<td><code>null</code></td>
<td><code>(7, B), (2, C), (8, D), (2, E)</code></td>
</tr>
</tbody>
</table>
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
A hash function $h$ maps keys of a given type to integers in a fixed interval $[0, N - 1]$

- Example: $h(x) = x \mod N$
  - 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 $h(x) =$ last four digits of $x$. 

<table>
<thead>
<tr>
<th>Array Index</th>
<th>Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>1</td>
<td>025-612-0001, 981-101-0002</td>
</tr>
<tr>
<td>2</td>
<td>451-229-0004</td>
</tr>
<tr>
<td>3</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>4</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>9997</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>9998</td>
<td>200-751-9998</td>
</tr>
<tr>
<td>9999</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
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)
    $$a_0a_1 \ldots a_{n-1}$$
  - We evaluate the polynomial
    $$p(z) = a_{n-1} + a_{n-2}z + a_{n-3}z^2 + \ldots + a_0z^{n-1}$$
    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
    $$p_0(z) = a_{n-1}$$
    $$p_i(z) = a_{n-i-1} + zp_{i-1}(z)$$
    ($i = 1, 2, \ldots, n-1$)
  - We have $p(z) = p_{n-1}(z)$
Compression Functions

**Division:**
- \( h_2(y) = y \mod N \)
- 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):**
- \( h_2(y) = (ay + b) \mod N \)
- \( N \) is prime, \( a \) and \( b \) are nonnegative integers such that
  \( a \mod N \neq 0 \)
  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

<table>
<thead>
<tr>
<th>Cell</th>
<th>Contacts</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∅</td>
</tr>
<tr>
<td>1</td>
<td>025-612-0001</td>
</tr>
<tr>
<td>2</td>
<td>∅</td>
</tr>
<tr>
<td>3</td>
<td>∅</td>
</tr>
<tr>
<td>4</td>
<td>451-229-0004</td>
</tr>
</tbody>
</table>

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) \quad \text{\{delegate the put to the list-based map at \( A[h(k)] \}\}
\]

if \( t = \text{null} \) 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) \quad \text{\{delegate the remove to the list-based map at \( A[h(k)] \}\}
\]

if \( t \neq \text{null} \) 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:**
- \( h(x) = x \mod 13 \)
- 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
```

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

```
41 18 44 59 32 22 31 73
```

```
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 ← h(k)
p ← 0
repeat
    c ← A[i]
    if c = ∅
        return null
    else if c.key () = k
        return c.element()
    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 object, called \textit{AVAILABLE}, which replaces deleted elements.

- \textbf{remove}(k):
  - We search for an entry with key \textit{k}.
  - If such an entry \((k, o)\) is found, we replace it with the special item \textit{AVAILABLE} and we return element \textit{o}.
  - Else, we return \textit{null}.

- \textbf{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 \textit{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 \(h(k,i) = (h(k) + i* d(k)) \mod N\) for \(i = 0, 1, \ldots, N - 1\)

- 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:
  \[ d_2(k) = q - (k \mod q) \]
  where
  - \(q < N\)
  - \(q\) is a prime

- The possible values for \(d_2(k)\) are
  \[ 1, 2, \ldots, q \]
Example of Double Hashing

Consider a hash table storing integer keys that handles collision with double hashing

- $N = 13$
- $h(k) = k \mod 13$
- $d(k) = 7 - k \mod 7$

Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order
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.