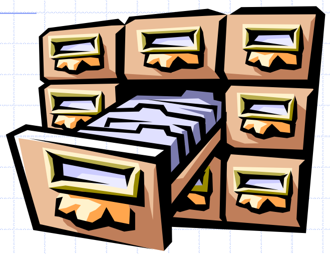
Lecture 8:
Maps, Hash Tables and Dictionaries

(Chapter 9, Sections 9.1, 9.2, 9.5 from the book)

Agenda

- Maps
  - Map ADT
  - List-Based Map Implementation
- Hash Tables
  - Bucket Array and Hash Functions
  - Collision Handling
- Dictionaries
  - Dictionary ADT
  - Dictionary Implementations

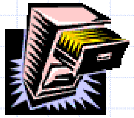
Maps



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
  - student-record database

The Map ADT



- get(k): if the map M has an entry e=(k,v) with key k, return its associated value v; else, return **null**;
- put(k, v): If M does not have an entry (k, v) then add it to the map M and return **null**; else, replace with v the existing value of the entry with key equal to k and 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(): return the number of entries in M;
- isEmpty(): test whether M is empty;
- entrySet(): return an iterable collection of the entries in M
- keySet(): return an iterable collection of the keys in M
- values(): return an iterator of the values in M

Exercise 1 – Map ADT

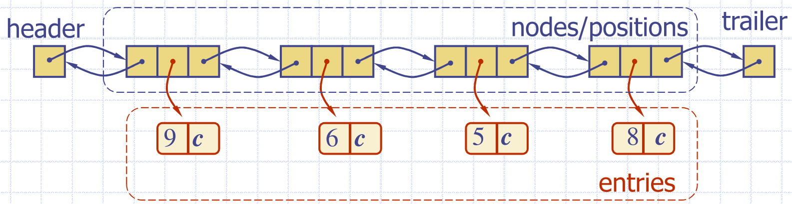
- Starting from the empty map, show the output and map after each of the following operations:
- isEmpty(); put(5,*A*); put(7,*B*); remove(5); put(2,*C*); put(8,*D*); put(2,*E*); put(2,*G*); remove(10); get(7); get(4); put(1,*D*); get(2); size(); remove(2); get(2); isEmpty();

Exercise 1 – Map ADT – Answer

Operation	***Output***	***Map***
isEmpty()	**true**	Ø
put(5,*A*)	**null**	(5,*A*)
put(7,*B*)	**null**	(5,*A*),(7,*B*)
remove(5)	*A*	(7,*B*)
put(2,*C*)	**null**	(7,*B*),(2,*C*)
put(8,*D*)	**null**	(7,*B*),(2,*C*),(8,*D*)
put(2,*E*)	*C*	(7,*B*),(2,*E*),(8,*D*)
put(2,*G*)	*E*	(7,*B*),(2,*G*),(8,*D*)
remove(10)	**null**	(7,*B*),(2,*G*),(8,*D*)
get(7)	*B*	(7,*B*),(2,*G*),(8,*D*)
get(4)	**null**	(7,*B*),(2,*G*),(8,*D*)
put(1,*D*)	**null**	(7,*B*),(2,*G*),(8,*D*),(1,*D*)
get(2)	*G*	(7,*B*),(2,*G*),(8,*D*),(1,*D*)
size()	4	(7,*B*),(2,*G*),(8,*D*),(1,*D*)
remove(2)	*G*	(7,*B*),(8,*D*),(1,*D*)
get(2)	**null**	(7,*B*),(8,*D*),(1,*D*)
isEmpty()	**false**	(7,*B*),(8,*D*),(1,*D*)

A Simple List-Based 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 get(k) Algorithm

```
Algorithm get(k):
   B = S.positions() {B is an iterator of the positions in S}
   while B.hasNext() do
        p = B.next()   { the next position in B }
        if p.element().getKey() == k       then
                return p.element().getValue()
   return null {there is no entry with key equal to k}
```


The put(k,v) Algorithm

```
Algorithm put(k,v):
B  = S.positions()
while B.hasNext() do
   p = B.next()
   if p.element().getKey() == k  then
        t = p.element().getValue()
        S.set(p,(k,v))
        return t        {return the old value}
S.addLast((k,v))
n = n + 1       {increment variable storing number of entries}
return null     { there was no entry with key equal to k }
```

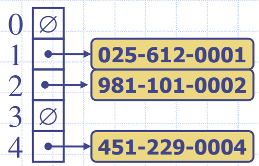

The remove(k) Algorithm

```
Algorithm remove(k):
B =S.positions()
while B.hasNext() do
   p = B.next()
   if p.element().getKey() = k  then
        t = p.element().getValue()
        S.remove(p)
        n = n – 1       {decrement number of entries}
        return t        {return the removed value}
return null             {there is no entry with key equal to k}
```

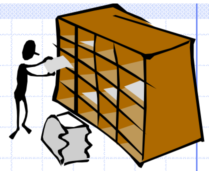

Performance of a List-Based Map

- Performance:
  - put, get and remove 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)

Hash Tables



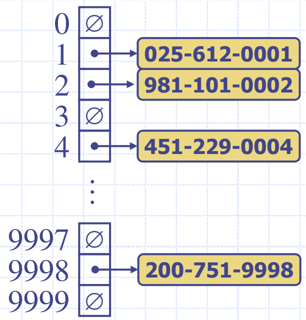
Hash Tables



- Hash tables are used to implement a map
- Hash table consists of two main components:
  - Bucket array – is an array A of size N, where each cell of A is thought of as a "bucket" (that is a collection of key-value pair)
  - 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
- When implementing a map with a hash table, the goal is to store item (k,v) at index $i = h(k)$

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



- The problem not addressed in here: collisions, i.e. more than one person can have SSN ended with the same four digits – we will talk about this later!

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 $\rightarrow [0, N-1]$

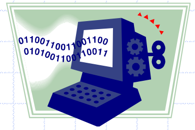
Hash Functions

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

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

- Hash codes assigned to the keys should avoid collisions
- The goal of the hash function is to "disperse" the keys in an apparently random way

Hash Codes

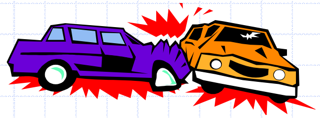


- Memory address:
  - We reinterpret the memory address of the key object as an integer (default hash code of all Java objects)
  - Good in general, except for numeric and string keys – Why?
- Integer cast:
  - We reinterpret the bits of the key as an integer
  - Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., byte, short, int and float in Java)
- 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 Java)

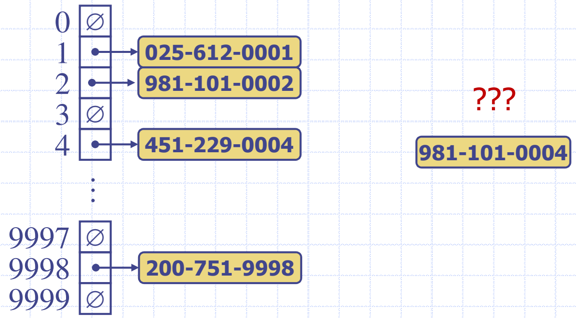
Compression Functions

- A good hash function should ensure that the probability of two different keys getting hashed to the same bucket is $1/N$
- Division:
  - $h_2(y) = y \bmod N$
  - The size N of the bucket array is usually chosen to be a prime
- Multiply, Add and Divide (MAD):
  - $h_2(y) = [(ay + b) \bmod p] \bmod N$
  - N is the size of bucket array
  - p is a prime number greater than N
  - a and b are integers chosen at random from the interval $[0, p-1]$ with $a > 0$

Collision Handling

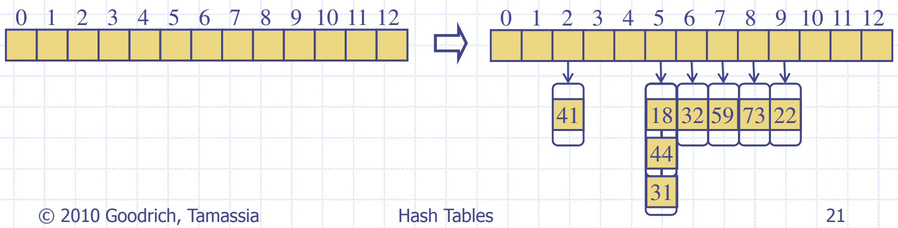


- Collisions occur when different elements are mapped to the same bucket



Separate Chaining

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



Map with Separate Chaining

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

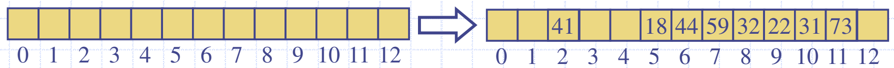
```
Algorithm get(k):
return A[h(k)].get(k)

Algorithm put(k,v):
t = A[h(k)].put(k,v)
if t == null then                    {k is a new key}
    n = n + 1
return t

Algorithm remove(k):
t = A[h(k)].remove(k)
if t ≠ null then                     {k was found}
    n = n - 1
return t
```


Open addressing

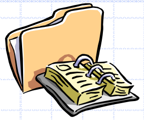
- Open addressing: the colliding item is placed in a different cell of the table:
  - Linear probing
  - Double hashing
  - Quadratic probing (individual study, p.396)
- Each table cell inspected is referred to as a "probe"
- Colliding items lump together, causing future collisions to cause a longer sequence of probes

Linear Probing

- Linear probing: handles collisions by placing the colliding item in the next (circularly) available table cell:
  - If we try to insert an entry (k,v) into a cell $A[i]$ that is already occupied, where $i=h(k)$, then we try next at $A[(i+1) \bmod N]$.
  - If $A[(i+1) \bmod N]$ is taken then we try $A[(i+2) \bmod N]$, and so on, until we find an empty cell.
- Example:
  - $h(k) = k \bmod 13$
  - 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
- 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.getKey () == k
      return c.getValue()
    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 $AVAILABLE$, which replaces deleted elements

- remove(k)
  - We search for an entry with key k
  - If such an entry (k, v) is found, we replace it with the special item $AVAILABLE$ and we return element v
  - Else, we return $null$
- put(k, v)
  - 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, v) in cell i

Java Hash Table Implementation with linear probing (p.397)

```
/** Search method - returns index of found key or -(a + 1), where a is the index of the first
   empty or available slot found. */
  protected int findEntry(K key) throws InvalidKeyException {
    int avail = -1;
    checkKey(key);
    int i = hashValue(key); int j = i;
    do {
      Entry<K,V> e = bucket[i];
      if ( e == null) {
        if (avail < 0)
          avail = i;            // key is not in table
        break; }
      if (key.equals(e.getKey())) // we have found our key
        return i;   // key found
      if (e == AVAILABLE) {           // bucket is deactivated
        if (avail < 0)
          avail = i; }  // remember that this slot is available
      i = (i + 1) % capacity;          // keep looking
    } while (i != j);
    return -(avail + 1);  // first empty or available slot }
```

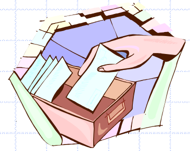

Java Hash Table Implementation with linear probing (p.397)

```
/** Returns the value associated with a key. */
 public V get (K key) throws InvalidKeyException {
   int i = findEntry(key);  // helper method for finding a key
   if (i < 0) return null;  // there is no value for this key, so reutrn null
   return bucket[i].getValue();     // return the found value in this case
 }

/** Removes the key-value pair with a specified key. */
 public V remove (K key) throws InvalidKeyException {
   int i = findEntry(key);       // find this key first
   if (i < 0) return null;       // nothing to remove
   V toReturn = bucket[i].getValue();
   bucket[i] = AVAILABLE;               // mark this slot as deactivated
   n--;
   return toReturn;
 }
```

Java Hash Table Implementation with linear probing (p.397)

```
/** Put a key-value pair in the map, replacing previous one if it exists. */
  public V put (K key, V value) throws InvalidKeyException {
    int i = findEntry(key); //find the appropriate spot for this entry
    if (i >= 0)//  this key has a previous value
      return ((HashEntry<K,V>) bucket[i]).setValue(value); // set new value
    if (n >= capacity/2) {
      rehash(); // rehash to keep the load factor <= 0.5
      i = findEntry(key); //find again the appropriate spot for this entry
    }
    bucket[-i-1] = new HashEntry<K,V>(key, value); //convert to proper index
    n++;
    return null;         // there was no previous value
  }
```


Double Hashing



- Double hashing uses a secondary hash function $h'(k)$
- If h maps some key k to a cell $A[i]$, with $i=h(k)$, that is already occupied, then we iteratively try the buckets:

  $$A[(i + f(j)) \bmod N] \text{ for } j = 1,2, \dots , N-1$$

  where $f(j) = j \cdot h'(k)$
- The secondary hash function $h'(k)$ 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:

  $$h'(k) = q - k \bmod q$$

  where: $q < N$ and q is a prime

Example of Double Hashing

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

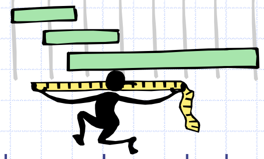
k	$h(k)$	$h'(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		

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

⇩

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

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: $1 / (1 - \alpha)$
- In practice, hashing is very fast provided the load factor is not close to 100%
- Applications of hash tables:
  - small databases
  - compilers
  - browser caches

Exercise 2 – Hash tables

- Draw the 7-entry hash table that results from using a hash function:

$$h(i) = (3i+5) \bmod 7,$$

to hash the keys 5, 6, 11, 4, 1, 15

assuming collisions are handled by:

a) Chaining

b) Linear Probing

c) Double hashing using the secondary hash function

$h'(i) = 5 - i \bmod 5$

Exercise 2 – Hash tables – Answer

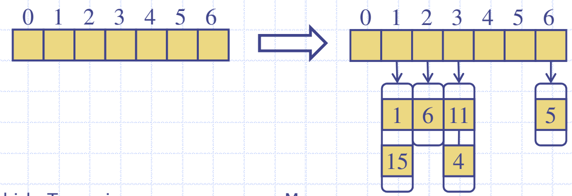
- Draw the 7-entry hash table that results from using a hash function:

$$h(i) = (3i+5) \bmod 7,$$

to hash the keys 5, 6, 11, 4, 1, 15

assuming collisions are handled by:

a) Chaining



Exercise 2 – Hash tables – Answer

- Draw the 7-entry hash table that results from using a hash function:

$$h(i) = (3i+5) \bmod 7,$$

to hash the keys 5, 6, 11, 4, 1, 15

assuming collisions are handled by:

b) Linear Probing

0	1	2	3	4	5	6

⇒

0	1	2	3	4	5	6
	1	6	11	4	15	5

Exercise 2 – Hash tables – Answer

- Draw the 7-entry hash table that results from using a hash function:

$$h(i) = (3i+5) \bmod 7,$$

to hash the keys 5, 6, 11, 4, 1, 15

assuming collisions are handled by:

c) Double hashing using the secondary hash function

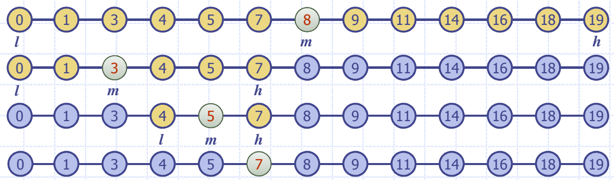
$h'(i) = 5 - i \bmod 5$

0	1	2	3	4	5	6

⇒

0	1	2	3	4	5	6
15	1	6	11	4		5

Dictionaries



Dictionary ADT

- The dictionary ADT models a searchable collection of key-value entries
- 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

- Dictionary ADT methods:
  - get(k): if the dictionary D has an entry with key k, returns it, else, returns **null**
  - getAll(k): returns an iterable collection of all entries with key k
  - put(k, v): inserts into the dictionary D and returns the entry (k, v)
  - remove(e): removes the entry e from the dictionary and returns the removed entry; an error occurs if entry is not in the dictionary D
  - entrySet(): returns an iterable collection of the entries in the dictionary
  - size(), isEmpty()

Example

Operation	Output	Dictionary
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,C),(8,D),(2,E)
get(7)	(7,B)	(5,A),(7,B),(2,C),(8,D),(2,E)
get(4)	**null**	(5,A),(7,B),(2,C),(8,D),(2,E)
get(2)	(2,C)	(5,A),(7,B),(2,C),(8,D),(2,E)
getAll(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(get(5))	(5,A)	(7,B),(2,C),(8,D),(2,E)
get(5)	**null**	(7,B),(2,C),(8,D),(2,E)

A List-Based Dictionary

- A log file or audit trail is a dictionary implemented by means of an unsorted sequence
  - We store the items of the dictionary in a sequence (based on a doubly-linked list or array), in arbitrary order
- Performance:
  - put takes $O(1)$ time since we can insert the new item at the beginning or at the end of the sequence
  - get and remove 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 log file is effective only for dictionaries of small size or for dictionaries on which insertions are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation) – better than the map

The getAll and put Algorithms

```
Algorithm getAll(k)
Create an initially-empty list L
for e: D do
    if e.getKey() == k  then
        L.addLast(e)
return L

Algorithm put(k,v)
Create a new entry e = (k,v)
S.addLast(e)   {S is unordered}
return e
```


The remove Algorithm

```
Algorithm remove(e):
{ We don't assume here that e stores its position in S }
B = S.positions()
while B.hasNext() do
    p = B.next()
    if p.element() == e then
        S.remove(p)
        return e
return null     {there is no entry e in D}
```


Hash Table Implementation

- 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.

Ordered Search Table

- A search table is a dictionary implemented by means of a sorted (ordered) array
  - We store the items of the dictionary in an array-based sequence, sorted by key
  - We use an external comparator for the keys
- Performance:
  - get takes $O(\log n)$ time, using binary search
  - put takes $O(n)$ time since in the worst case we have to shift $n/2$ items to make room for the new item
  - remove takes $O(n)$ time since in the worst case we have to shift $n/2$ items to compact the items after the removal
- A search table is effective only for dictionaries of small size or for dictionaries on which searches are the most common operations, while insertions and removals are rarely performed (e.g., credit card authorizations)