Computer Science 3A - CSC3A10

Lecture 8: Maps and Hash Tables

Academy of Computer Science and Software Engineering
University of Johannesburg

1. Map ADT
   - Map ADT Properties
   - Map ADT Example
   - List-Based Map Implementation

2. Hash Table ADT
   - Hash Table Properties
   - Hash Table Example
   - Hash Functions
   - Collision Handling
   - Linear Probing
   - Double Hashing
   - Performance of Hashing

Maps ADT

Map ADT Properties

- A map models a search-able 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

Map ADT Methods

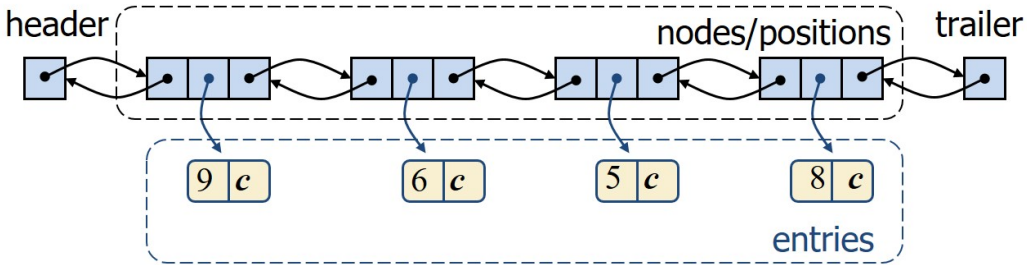
- **get(k)**: if the map M has an entry with key k, return its associated value; else, return *null*
- **put(k, v)**: insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
- **remove(k)**: if the map M has an entry with key k, remove 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

Map ADT 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*)

List-Based Map Implementation

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

```
1 Algorithm get(k):
2    For each position p in S.positions() do
3      if   p.element().getKey() = k then
4        return p.element().getValue()
5   return null {there is no entry with key equal to k}
```

get(k)

The put(k,v) Algorithm

```
1 Algorithm put(k,v):
2 For each position p in S.positions() do
3   if p.element().getKey() = k  then
4     t = p.element().getValue()
5     B.set(p,(k,v))
6     return t  {return the old value}
7 S.addLast((k,v))
8 n = n + 1   {increment variable storing number of entries}
9 return null {there was no previous entry with key equal to k}
```

put(k,v)

The remove(k) Algorithm

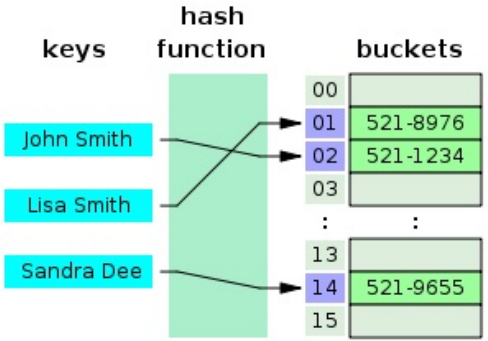
```
1 Algorithm remove(k):
2 for each position p in S.positions() do
3   if p.element().getKey() = k  then
4     t = p.element().getValue()
5     S.remove(p)
6     n = n − 1   {decrement number of entries}
7     return t  {return the removed value}
8 return null {there is no entry with key equal to k}
```

remove(k)

Performance of list-Based Map

- *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

Hash Table ADT

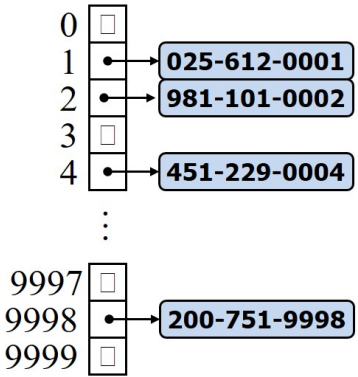


Hash Table Properties

- 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 $i = h(k)$
- 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

Hash Table 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



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 is applied first, and the compression function is applied next on the result, i.e. $h(x) = h2(h1(x))$
- 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
- **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)

Hash Codes II

- **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)
- **Polynomial accumulation**:
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits) $a_0 a_1 ... a_{n-1}$
  - We evaluate the polynomial $p(z) = a_0 + a_1 z + a_2 z^2 + ... + a_{n-1} z^{n-1}$ 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)

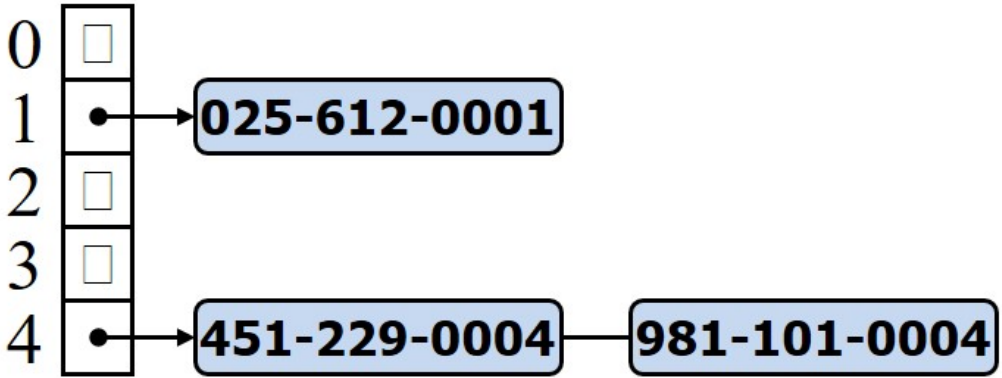
Compression Functions

- **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

- Collisions occur when different elements are mapped to the same cell
- **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

Collision Handling II



Map Methods with Separate Chaining used for Collisions

```
1  Algorithm get(k):
2  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
3  return A[h(k)].get(k)   {delegate the get to the list-based map at A[h(
       k)]}
4
5  Algorithm put(k,v):
6  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
7  t = A[h(k)].put(k,v)  {delegate the put to the list-based map at A[h(k)
       ]}
8  if t = null then     {k is a new key}
9    n = n + 1
10 return t
```

get and put with collisions

Map Methods with Separate Chaining used for Collisions II

```
1 Algorithm remove(k):
2 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
3 t = A[h(k)].remove(k)        {delegate the remove to the list-based map
      at A[h(k)]}
4 if t != null then              {k was found}
5   n = n - 1
6 return t
```

remove with collisions

Linear Probing

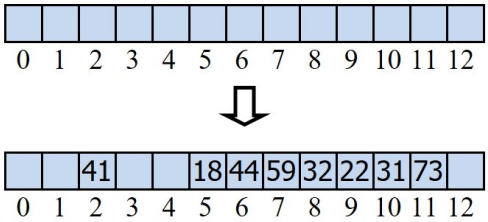
- **Open addressing**: the colliding item is placed in a different cell of the table
- Linear probing handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a "probe"
- Colliding items lump together, causing future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

Linear Probing II

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
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

Search with Linear Probing

```
1  Algorithm get(k)
2    i = h(k)
3    p = 0
4    repeat
5      c = A[i]
6      if c == null
7        return null
8       else if c.key() = k
9        return c.element()
10     else
11       i = (i + 1) mod N
12     p = p + 1
13   until p == N
14   return null
```

get with Linear Probing

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*

Updates with Linear Probing II

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 $(i + jd(k)) \bmod N$ for $j = 0, 1, ..., N - 1$
- 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

Double Hashing II

Common choice for the secondary hash function:

- $d(k) = q - k \bmod q$
- where $q < N$ and q is a prime

The possible values for $d(k)$ are $1, 2, ..., q$

Double Hashing Example

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

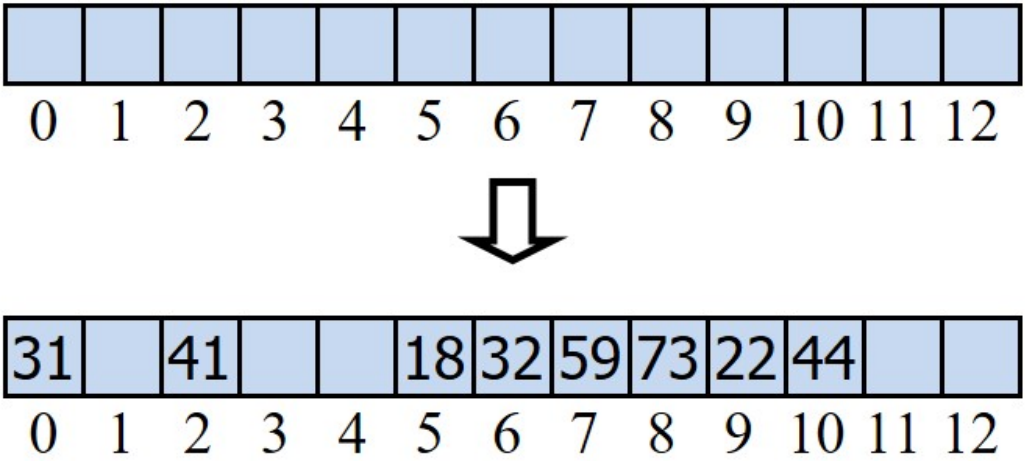
- $N = 13$
- $h(k) = k \bmod 13$
- $d(k) = 7 - k \bmod 7$

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

Double Hashing Example II

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		

Double Hashing Example II



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 $1/(1-\alpha)$

Performance of Hashing II

- 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