CS601: Principles of Software Development

Data Structures Cont. Hash Table.

Olga A. Karpenko

Parts of this presentations are based on the Java Solftware Solutions book by Lewis&Loftus and on the slides of Prof. Galles.

Announcements

- Graduate tutor: Xue Wang

  MF 3-5pm, Tue 2-4pm, CS labs, 4th floor
- Orientation for International Students: Friday, Sep 1, 3:30-4:45, Harney 136
- Lab0 will be out tonight
- Accept invitation on github to join USF-CS601-Fall2017 and to get access to lab0 starter code

Collection: Set

- Contains no duplicate elements

```
{ 1, 2, 3, 4 } ←Okay
{ 1, 2, 2, 3 } ←Not okay
```

- Main implementations
  - `HashSet`
  - `TreeSet`

Collection -> Set -> HashSet

- No guarantee on order
  - Iteration does not go in sorted order
  - Iteration order may change over time
- Basic operations are constant time (fast)
- Uses HashMap internally

Example

```
HashSet<String> words = new HashSet<>();

words.add("cat");
words.add("mail");
words.add("dog");
words.add("cat"); // won't add it again
System.out.println(words); //don’t know the
order
```

HashSet

- Sometimes HashSet is **slower** than an ArrayList
  - if you want to **iterate** on elements for example

Collection -> Set -> TreeSet

- Guarantees sorted order
- Iteration goes in sorted order
- Can easily navigate forward and backward
- Basic operations are log(n) time (decent)
- Choose over HashSet only if need to maintain sorted order

TreeSet: Example

```
TreeSet<String> wordsTree = new TreeSet<String>();

wordsTree.add("cat");
wordsTree.add("mail");
wordsTree.add("dog");
String firstElement = wordsTree.first();
String test = wordsTree.lower("mail");
```

TreeSet: Sort Order

- Sorts using the natural sort order of the type of its elements
  - alphabetical order for strings
  - numerical order for integers
  - unless a comparator is provided to define sort order

Maps

Map

- Maps keys to values
  - Keys must be unique
  - Values need not be unique
- Allows to quickly find the entry by key

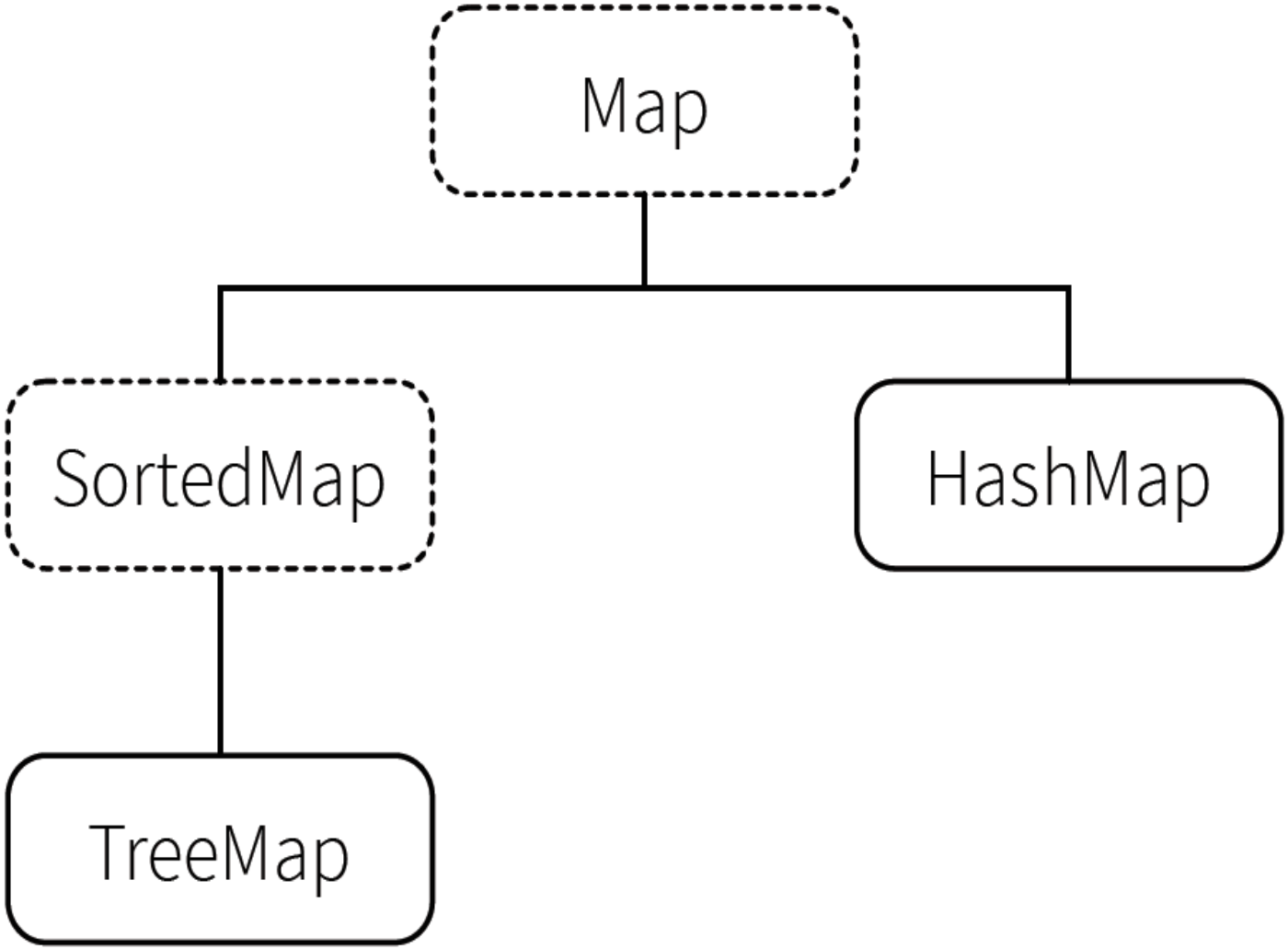
Example: Dictionary

- Associates a particular key with a value
  - Keys: words
  - Values: definitions

Another Example of a Map

- Phone Book
  - Keys are names
  - Values are phone numbers

Map Implementations



Map -> HashMap

- Basic operations are constant-time (fast)
- Need to specify:
  - the type of the keys
  - the type of the values
- Entry: (key, value) pair

HashMap: Dictionary

```
import java.util.HashMap;

Map<String, String>  dict = new HashMap<>();

dict.put("smart", "Having a quick-witted
intelligence");

dict.put("university", "A school that offers
courses leading to a degree");
```

HashMap: Dictionary

- Find if the word is in the dictionary
- Returns a value or null if not in the map

```
String value = dict.get("bat");
```

- Keys are case sensitive

`dict.get("Smart");` will return null

HashMap: Phone Book

```
Map<String, Integer>  phoneBook = new
HashMap<>();

phoneBook.put("Jones", 5103495832);
phoneBook.put("Patel", 9258341112);
```

- Can't use int, types need to be Classes
- A map has no intrinsic ordering (no indices)
- `phoneBook.get(1);` will not work!

HashMap Methods

- put(key, value)
- get(key) - > value
- containsKey(key) -> boolean
- containsValue(value) ->boolean
- isEmpty()
- remove(key)
- keySet()

Iterating Over the Keys

- Use keySet() method and iterate
- Print all the keys and values:

```
for (String key : phoneBook.keySet()) {
  System.out.println("Key:" + key);
  System.out.println("Data:" + phoneBook.get(key));
}
```

Iterating Over the Values

- Use entrySet() method and iterate

```
Map<String, Object> map = new HashMap<String, Object>();

for (Map.Entry<String, Object> entry : map.entrySet()) {
    System.out.println("key=" + entry.getKey());
    System.out.println("value=" + entry.getValue());
}
```

HashMap: Another Example

- Key: a word in the array
- Value: the number of times the word occurs in the array

```
String[] words = {"cat", "mail", "door", "cat"};

Map<String, Integer> hMap = new HashMap<String,
Integer>();

// continued on the next page
```

HashMap: Another Example

```
// continued..

for (String word : words) {
        Integer count = hMap.get(word);
        if (count == null)
             hMap.put(word, new Integer(1));
         else {
                count = count + 1;
               hMap.put(word, count);
        }
}
```

Map -> TreeMap

- Entries are sorted in ascending key order
- Basic operations are log-time

TreeMap: Additional Methods

- Object firstKey(), Object lastKey()
- SortedMap headMap(Object toKey)
  - Returns a part of the map where all keys are less than toKey
- SortedMap tailMap(Object fromKey)
  - Returns a part of the map where all keys are larger than fromKey
- SortedMap subMap(Object fromKey, Object toKey)
- See posted example

How to implement a Map

- Different approaches
- For lab0 you will implement Map using a Hash table

Map Implementation

- Store data in unsorted *non-contiguous array*

- How do we decide where to store each element?

*To make the diagram cleaner, we show only keys in this example

Hash Function

- We need a function that
  - Takes a key
  - Maps it to the index of the array where the element is stored
- Called *Hash Function*
  - If hash(key) == i, we say that the key *hashes* to i

Hash Function

- We want different keys to hash to different values
  - Not possible. Too many possible keys!

Collisions

- If two keys hash to the same value, a *collision* occurs
- How to minimize collisions?
  - Pick a hash function that distributes the keys evenly through the array

Hash Table

- Array of size N

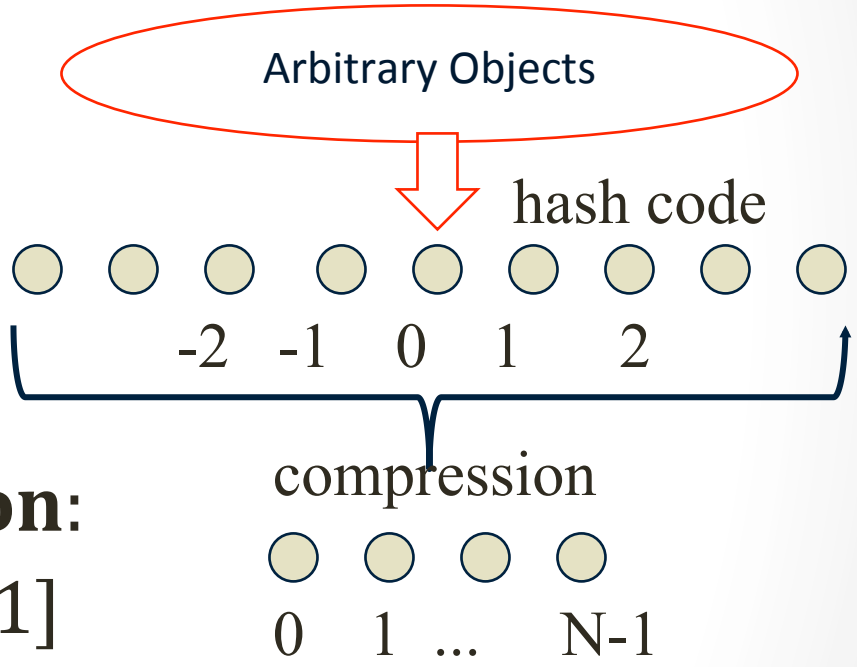
- Hash Function
  - Maps keys to indices in the array
  - The goal is to “disperse” the keys in a random way

Hash Function

- Usually specified as the composition of two functions:

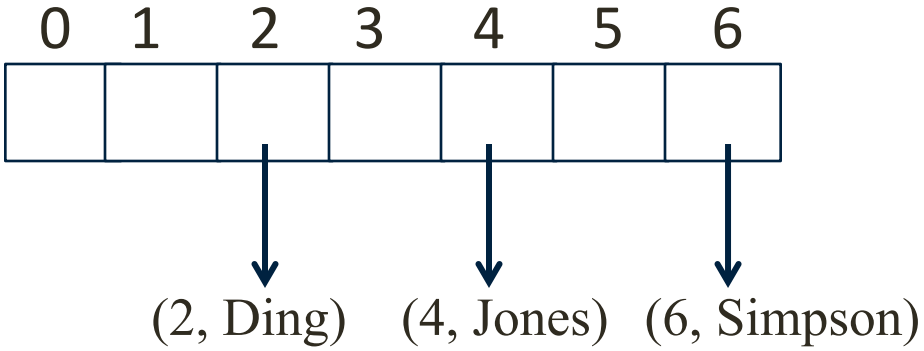
Hash code:
h_1: keys → integers

Compression function:
h_2: integers → $[0, N-1]$



Integer Keys

- Consider an easy case first:
- Well distributed integers in the range [0, N-1] -> need only the array A
- Insert entry with key k into A[k]
- (4, "Jones"), (6, "Simpson"), (2, ”Ding")



Integer Keys

- Integers in a different range
- Need to compute a compression function

Compression Functions: Division

$h_2(k) = k \bmod N$

Where N is the size of the table

Example

- (7, "Jones"), (3, "Nielson"), (13, "Lee");

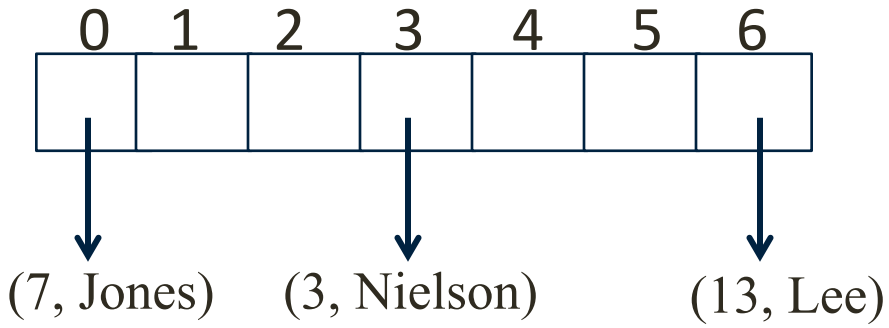


Table size is 7. Hash(key) = key mod 7.

Compression Functions: Division

$h_2(k) = k \bmod N$

- The size *N* of the hash table is usually chosen to be a prime to avoid patterns

- Example: N = 10, Keys: 200, 220, 230, 240, 250..
- Making N prime helps prevent patterns of this sort

Compression Functions: MAD

$h_2(k) = (a*k + b) \bmod N$

- a and b are nonnegative integers such that

  $a \bmod N \neq 0$

  Otherwise, every integer would map to the same value b

Hash Code for Strings

- Often the keys are Strings
- How can we map a String to an integer?
  - Different approaches exist
  - We will use *polynomial hash code*

Polynomial Hash Code

- Assume the key is a tuple $(x_0, x_1, ..., x_{k-1})$
- x_i is an ASCII code of character at position i
- The code should take into account positions

  $x_0 * a^{k-1} + x_1 * a^{k-2} + .. + x_{k-2}*a + x_{k-1}$,

  for some constant a

- Used in hashCode() method in class String

Polynomial Hash Code

H	e	l	l	o
72	101	108	108	111

$$h(\text{"Hello"}) = 72a^4 + 101a^3 + 108a^2 + 108a^1 + 111a^0$$

- The choice of a is important
  a = 31, 33, 37, 39 work well in practice
- Take mod N after computing the hash code

http://www.mathcs.emory.edu/~cheung/Courses/323/Syllabus/Map/hash-func.html

Hash Function

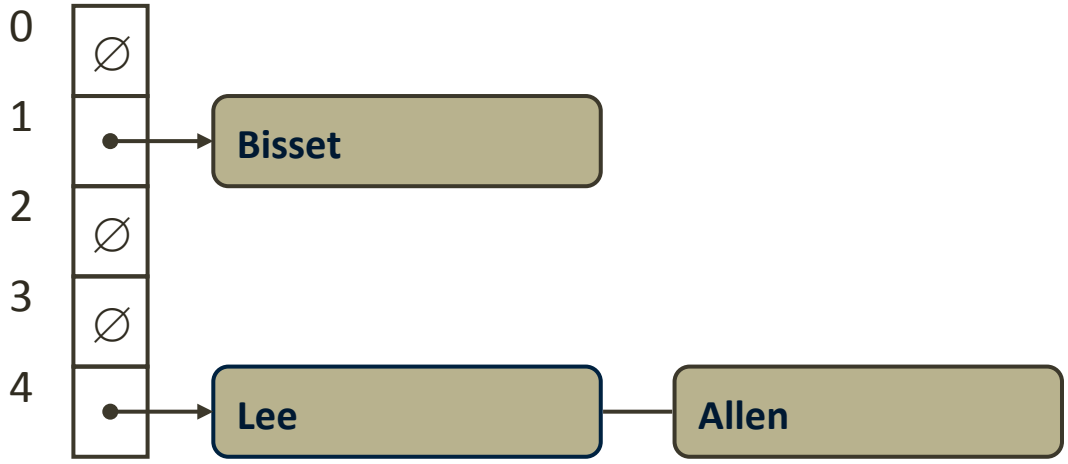
- We'd like to ensure that different keys will always hash to different values.
- Why is this not possible?
  - Too many possible keys

Handling Collisions

- When two keys hash to the same value, a collision occurs
- We cannot avoid collisions
- We can minimize them by picking a hash function that distributes keys evenly through the array
- Two Approaches
  - Open Hashing
  - Closed Hashing

Open Hashing: Separate Chaining

- Each cell in the table has a pointer to the linked list of entries that map to this index:



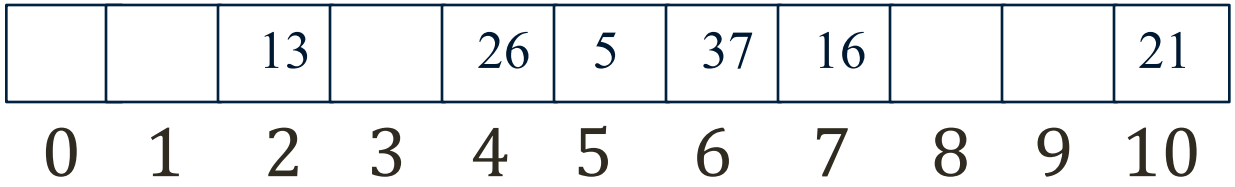
Closed Hashing

- All values are stored at the array
- The number of elements limited to the table size
- The colliding item is placed in a different cell of the table

Closed Hashing: Linear Probing

- If collision happens, it places the entry in the *next available table cell (circularly)*
- Example: h(k) = k mod 11

Inserted keys 13, 26, 5, 37, 16, 21.
Insert key = 15:

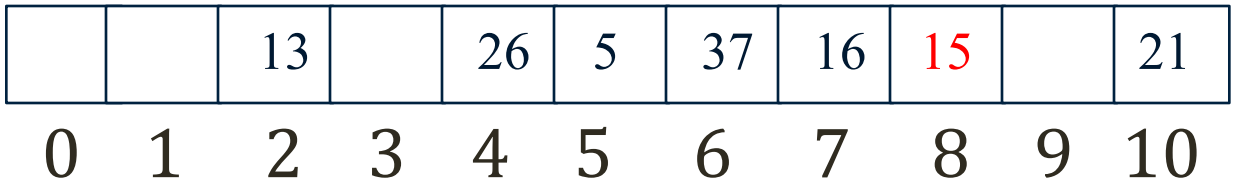


Closed Hashing: Linear Probing

Example: h(k) = k mod 11

Inserted keys 13, 26, 5, 37, 16, 21.

Insert key = 15: will try indices 4, 5, 6, 7 before inserting at index 8.



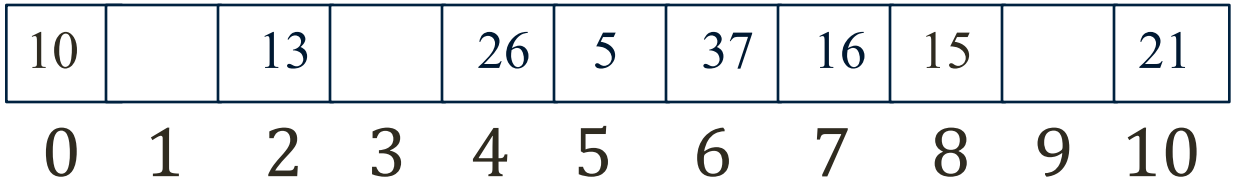
Closed Hashing: Linear Probing

Example: h(k) = k mod 11

Inserted keys 13, 26, 5, 37, 16, 21, 15

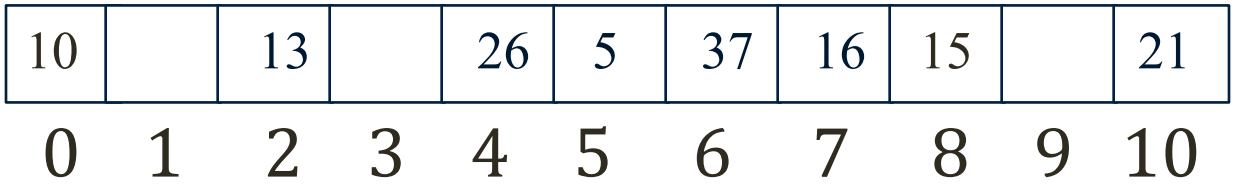
Insert key = 10: will try index 10

before inserting at index 0.

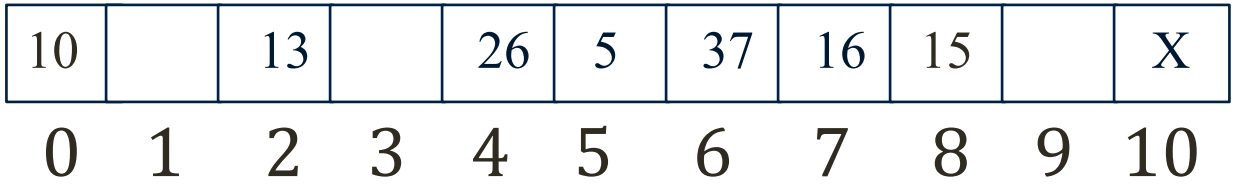


delete(key) with Linear Probing

- Find the entry given the key
- Replace the element with a marker (isDeleted)
- Example: before deleting key = 21



- Example: after deleting key = 21



get(key) with Linear Probing

- Start searching at cell $h(k)$
- Probe consecutive locations and check
  - Found an entry with another key, keep searching
  - Found a cell where isDeleted = true, keep searching
  - Found the entry with key k – done, return value
  - Found an empty cell (where isDeleted is false) - done, return null
  - Tried searching N times, did not find – return null

put(key) with Linear Probing

- Start searching at cell $h(k)$
- Probe consecutive locations and check
  - Found an empty cell or isDeleted cell - insert the entry
  - Tried N times with no luck – the table is full

Lab 0

- Implement a map using a hash table that uses closed hashing with linear probing
- Required to use provided starter code
- Implement classes Hotel and Review
- Submit to github (see instructions)