CSE 12 – Basic Data Structures and Object-Oriented Design
Lecture 15

Greg Miranda & Paul Cao, Winter 2021

This lecture is being recorded

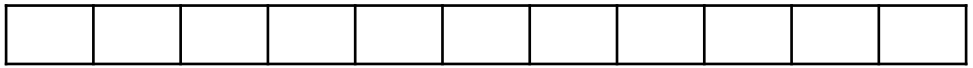
Topics

- Map and HashTable
- Questions on Lecture 15?

Questions on Lecture 15?

Hash table Motivation 1: Two sum problem

- Consider the 2-sum problem: Given an unsorted array (A) of N integers between 0 and 1000,000, find all pairs of elements that sum to a given number T

A 

Method 1:

```
for (i = 0; i<N; i++){
  for (j =i+1;j<N; j++){
    if ((A[i]+A[j])==T)
      store (A[i], A[j]);
  }
}
```

Method 2:

```
bool Hashtable[1000000]={false};
for (i = 0; i<N; i++){
  Hashtable[A[i]]=true;
}
for (i = 0; i<N; i++){
 if (Hashtable[T-A[i]]) == true)
   store (A[i], T-A[i]);
}
```

Which approach is faster?

A. Method 1
B. Method 2
C. They are the same

We have a student record system (i.e. ArrayList) and we will try to insert/delete/search for student

```
class Student{
  int pid;//unique
  double gpa;
  public Student(int pid, double gpa){
    this.pid = pid;
    this.gpa = gpa;
  }
  public boolean equals(Object o){
    //compare if two students are the same
    if (o == null){
      return false;
    }
    if (o instanceof Student){
       Student s = (Student)o;
       if (this.pid == s.pid){
         return true;
       }
    }
    return false;
  }
}
```

```
public class StudentRecord{
  ArrayList<Student> data;
  int capacity;
  int size;
  // ctor etc
  public boolean search(Student key){
    for (int i = 0; i < size; i++){
      if (data[i].equals(key)){
        return true;
      }
    }
    return false;
  }
}
```

What will affect how fast I can search for this key?

A. It depends on the capacity
B. It depends on where the element is in the list
C. It depends on the size
D. A combination of some factors above

We have a student record system (i.e. ArrayList) and we will try to insert/delete/search for student

```
class Student{
  int pid;//unique
  double gpa;
  public Student(int pid, double gpa){
    this.pid = pid;
    this.gpa = gpa;
  }
  public boolean equals(Object o){
    //compare if two students are the same
    if (o == null){
      return false;
    }
    if (o instanceof Student){
       Student s = (Student)o;
       if (this.pid == s.pid){
         return true;
       }
    }
    return false;
  }
}
```

```
public class StudentRecord{
  ArrayList<Student> data;
  int capacity;
  int size;
  // ctor etc
  public boolean search(Student key){
    for (int i = 0; i < size; i++){
      if (data[i].equals(key)){
        return true;
      }
    }
    return false;
  }
}
```

Is the search speed uniform?

A. The size of the ArrayList (i.e. capacity)
B. The number of elements already in the ArrayList (i.e. size)
C. Both A and B

An Improved ArrayList

What if every element of type E defined a method called “magic” that was guaranteed to return a unique int value between 0 and 9999999.

```
public class StudentRecord{
  private ArrayList<Student> data;
  //ctor will make data very big (1M elements)

  public boolean search(Student s){
    position = s.magic();
    if (data.at(position).equals(s)){
      return true;
    }
    return false;
  }
  public boolean add(Student s) {
    int position = s.magic();
    if (data.get(position) == null) {
      data.add(position, s);
      return true;
    }
    return false;
  }
    …
}
```

An Improved ArrayList

What if every element of type E defined a method called “magic” that was guaranteed to return a unique int value between 0 and 9999999.

If this could be done, which of the following is true of this approach?

A. It might allow duplicate elements to be inserted.
B. If there are a lot of elements in the data, it might become slow to check if an element is in the set.
C. If there are a lot of elements in the data, it might become slow to add a new element to the data.
D. B&C only
E. None of the above

```
public class StudentRecord{
  private ArrayList<Student> data;
  //ctor will make data very big (1M elements)

  public boolean search(Student s){
    position = s.magic();
    if (data.at(position).equals(s)){
      return true;
    }
    return false;
  }
  public boolean add(Student s) {
    int position = s.magic();
    if (data.get(position) == null) {
      data.add(position, s);
      return true;
    }
    return false;
  }
    …
}
```

An Improved ArrayList

What if every element of type E defined a method called “magic” that was guaranteed to return a unique int value between 0 and 9999999.

Why is this not possible? What is the problem here?

```
public class StudentRecord{
  private ArrayList<Student> data;
  //ctor will make data very big (1M elements)

  public boolean search(Student s){
    position = s.magic();
    if (data.at(position).equals(s)){
      return true;
    }
    return false;
  }
  public boolean add(Student s) {
    int position = s.magic();
    if (data.get(position) == null) {
      data.add(position, s);
      return true;
    }
    return false;
  }
    …
}
```

An Improved ArrayList

- What if every element of type E defined a method called “magic” that was guaranteed to return a unique int value between 0 and 9999999.

But Java (almost) has this method!

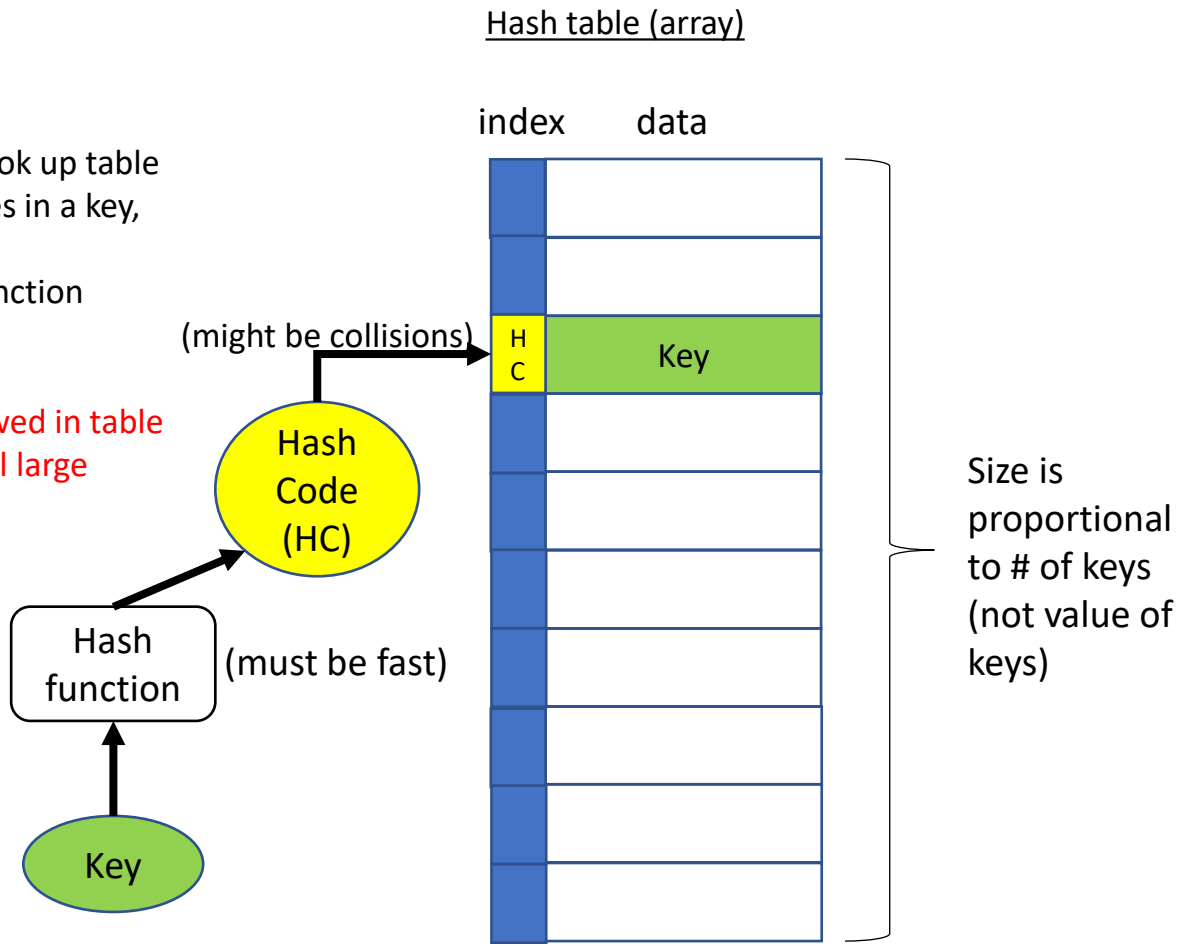
```
public class StudentRecord{
  private ArrayList<Student> data;
  //ctor will make data very big (1M elements)

  public boolean search(Student s){
    position = s.hashCode();
    if (data.at(position).equals(s)){
      return true;
    }
    return false;
  }
  public boolean add(Student s) {
    int position = s.hashCode();
    if (data.get(position) == null) {
      data.add(position, s);
      return true;
    }
    return false;
  }
    …
}
```

Hashing

- Let's modify our array-based look up table
- Need a hash-function h(x): takes in a key, returns an index in the array
- gold standard: random hash function

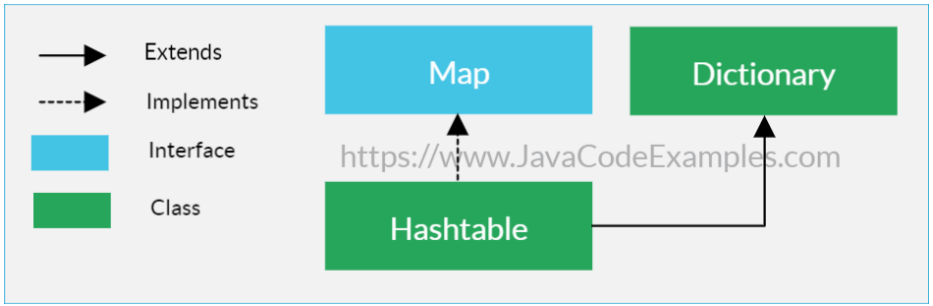
- In general, no null value is allowed in table
- Table size is fixed and in general large



Hashtable

- java.util.Hashtable
- Implementation of the Map interface extension of Dictionary abstract class
- Does not guarantee the order of things
  - Mean the elements may not be returned in the same order in which they

```
public class Hashtable<K,V> extends Dictionary<K,V> implements Map<K,V>, Cloneable, Serializable
```



Map Interface

https://docs.oracle.com/javase/10/docs/api/java/util/Map.html

- A map contains values on the basis of key, i.e. key and value pair.
- Each key and value pair is known as an entry.
- Contains unique keys
- Useful when you have to search, update or delete elements on the basis of a key
- Does not allow duplicate keys, but you can have duplicate values.

Dictionary Abstract Class

- Key/value storage repository and operates like a Map
- Given key and value, you can store the value in a dictionary object.
- Once value stored, you can retrieve it by using its key.

What is a Hash Table?

- A Hash Table is a data structure.
  - Each list known as a bucket. The position of the bucket is identified by calling the hashcode() method. A hashtable contains values based on the key.
- Contains unique elements
- Does not allow null key or value
- Offers fast insertion and searching
- They are limited in size because they are based on arrays
  - Can be resized, but it should be avoided

hashCode

```
public int hashCode()
```

Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by `HashMap`.

The general contract of `hashCode` is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is *not* required that if two objects are unequal according to the `equals(java.lang.Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

As much as is reasonably practical, the hashCode method defined by class `Object` does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java™ programming language.)

Returns:

`a hash code value for this object.`

See Also:

`equals(java.lang.Object), System.identityHashCode(java.lang.Object)`

Generates a number that can be used by a hash function (or simply as the hash value itself)

In general, what is the difference between the value returned by hashCode() and the index location where the item ends up in a particular hash table?

A. Nothing. The value returned by hashCode can be used directly as the index for the item in any hash table
B. The value returned by hashCode might be larger than the size of the hash table
C. The hashCode function might return the same value for two different objects, and indexes in hash tables must always be unique for different objects
D. The hashCode function might return different values for two objects that are considered equal (and for hash tables, two values that are considered equal must have the same hashcode/index value)

hashCode

```
public int hashCode()
```

Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by `HashMap`.

The general contract of `hashCode` is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is *not* required that if two objects are unequal according to the `equals(java.lang.Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

As much as is reasonably practical, the hashCode method defined by class `Object` does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java™ programming language.)

Returns:

a hash code value for this object.

Finite “universe” of objects to store

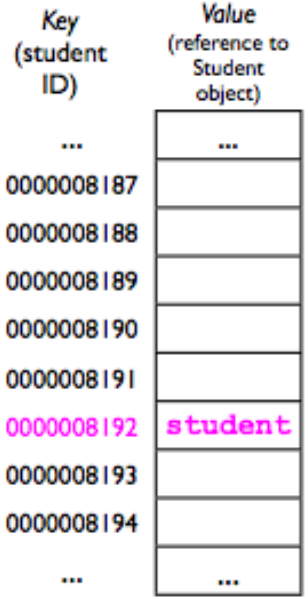
- Sometimes there may exist only a finite **universe** of possible keys/ elements to insert into a hash table.

- Sometimes this finite set of keys is small enough that we can allocate an array big enough to give *every possible key its own slot.*

- In this case, we can make the “search” process for a particular key *trivial*
  - We simply “jump” to the unique array index assigned to that key.
  - This takes only *O(1)* time in the *worst-case.*

Finite “universe” of objects to store

- For example, if we have a UCSD student database and each student has an id, we could allocate an array 4 **billion** elements long.
- We could define the hashCode() function of a student object to return the student's id, then use that value directly in the array.
- When adding a student, we simply insert an entry at his/her unique location.
- (IDs are guaranteed to be unique).



Hash functions

- A hash function maps an object or key to a position in an array (the hash table). What properties are required and/or desired from a good hash function?

Hash function

- To be useful, a hash function must be ***fast***
  - Its performance should not depend on the particular key.
  - Runs in "constant time" (more on this later…)

- A hash function must also be ***deterministic****:*
  - Given the *same value,* it must *always* return the *same array index.* (Otherwise, how would we find something we stored earlier?)

- A “good” hash function should also be **uniform**:
  - Each “slot” *i* in the array should be equally likely to be chosen as any other slot *j.*

Is it a good hash function?

```
int hashFucntion (int studentID){
 return M/2;
} //M is a size of a HT
```

A: Yes,

B: No, it is not fast

C: No, it is not deterministic

D: No, it is not uniform

Make it better…

How do you write a good hash function for objects?

Hash Tables & Hash Functions

- Key values are assigned to elements in a Hash Table using a Hash Function
- A Hash Function helps calculate the index an item should go in
  - Index must be small enough for the arrays size
  - Don’t overwrite other data in the Hash Table
- A Hash Functions job is to store values in an array with a limited size
- It does it in a way that the array doesn’t need to be searched through to find it
  - Enter values in any order
  - Be able to find them using a calculation instead of searching through the array

Hash Table – draw the picture (Separate Chaining)

```
int getIndex(String k) {
    return k.length;
}
```

# of buckets – 6

(i.e. the size of the array)

```
set("Smith", 1);
set("Johnson", 2);
set("Williams", 3);
set("Brown", 4);
set("Jones", 5);
set("Garcia", 6);
set("Miller", 7);
set("Davis", 8);
set("Rodriguez", 9);
set("Martinez", 10);
```