CSCI 136:
Data Structures
and
Advanced Programming

Lecture 31

Hash tables, part 3

Instructor: Dan Barowy

Williams

Announcements

Move Fri hours to Sat, 12-2

Next week is last week!

Return to graphs, final exam review

Outline

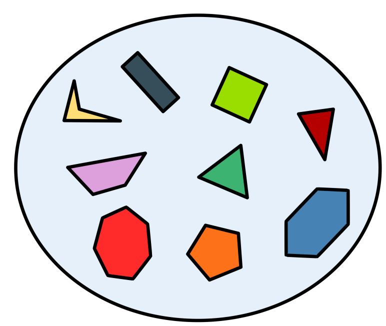
Sets

hashCodes revisited

Lab 10 part 2 overview

Set

A **set** is a an abstract data type that stores **at most one copy** of each unique value, in **no particular order**.



Set ADT operations

Essential `Set<T>` operations are:

```
public void add(T value)                          // add
public T remove(T value)                          // remove
public boolean contains(T value)                  // contains
public int size()                                 // # unique Ts
public void addAll(Structure<E> other)            // union
public boolean containsAll(Structure<E> other)    // is subset
public void removeAll(Structure<E> other)         // difference
public void retainAll(Structure<E> other)         // intersection
```

Set implementation

The `structure5` `SetList<T>` implements `Set<T>` using a **list**.

Is this a **good** or **bad** choice? **Worst case** analysis:

`List<T>` (assuming no order)

`add` :	**O(n)**
`remove` :	**O(n)**
`contains` :	**O(n)**
`size` :	**O(1)**

`Hashtable<T,?>`

`add` :	**O(n)**
`remove` :	**O(n)**
`contains` :	**O(n)**
`size` :	**O(1)**

Set implementation

As with QuickSort, **worst-case analysis** is **misleading** for hash tables!

Is this a **good** or **bad** choice? **Average case** analysis:

`List<T>` (assuming no order)

`add` :	**O(n)**
`remove` :	**O(n)**
`contains` :	**O(n)**
`size` :	**O(1)**

Hashtable complexity

Load factor is a ratio ***n/k***, where ***n*** is the **number of elements** in a hash table and where ***k*** is the **number of buckets**.

Method	Successful	Unsuccessful
Linear probes	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)}\right)$	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$
Double hashing	$\frac{1}{\alpha}\ln\frac{1}{(1-\alpha)}$	$\frac{1}{1-\alpha}$
External chaining	$1+\frac{1}{2}\alpha$	$\alpha+e^{-\alpha}$

Figure 15.11 Expected theoretical performance of hashing methods, as a function of α, the current load factor. Formulas are for the number of association compares needed to locate the correct value or to demonstrate that the value cannot be found.

Why is **load factor** effectively a **constant**?

Set implementation

As with QuickSort, **worst-case analysis** is **misleading** for hash tables!

Is this a **good** or **bad** choice? **Average case** analysis:

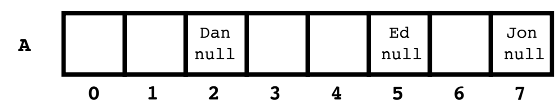
`List<T>` (assuming no order)		`Hashtable<T,?>`	
`add` :	O(n)	`add` :	O(1)
`remove` :	O(n)	`remove` :	O(1)
`contains` :	O(n)	`contains` :	O(1)
`size` :	O(1)	`size` :	O(1)

https://en.wikipedia.org/wiki/Best%2C_worst_and_average_case#Data_structures

Obstacles?

A **set** stores at most one unique value of type `T`.

A **map** stores at most one unique key of type `K` along with a value `V`.



We can **repurpose** a map and **store nothing** in the value.

Let's implement `SetHashtable<T>`

hashCode

hashCode

The **`hashCode`** method defines a hash function for a given type. In Java, all classes inherit a `hashCode` method from `Object`.

For built-in types, Java supplies **good default** `hashCode`s. E.g., `String`, `Character`, `Integer`, `Double`, etc.

For user-defined types (i.e., classes that you implement), the default `hashCode` is **usually inappropriate**.

If you intend to use your class as a key in a Map, you **should override both** `hashCode` and `equals`.

hashCode

Be aware of the **rules** when overriding `hashCode`!

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)`

Suspiciously similiar demo app.

Lab 10 part 2 Overview

Recap & Next Class

Today we learned:

Sets

hashCodes revisited

Lab 10 part 2 overview

Next class:

Back to graphs