CS2110 Recitation Week 8. Hashing

Hashing: An implementation of a set. It provides O(1) expected time for set operations

Set operations

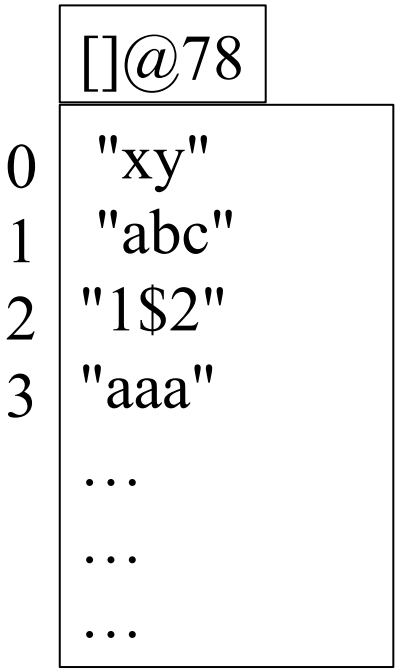
- Make the set empty
- Add an element to the set
- Remove an element from the set
- Get the size of the set (number of elements in it)
- Tell whether a value is in the set
- Tell whether the set is empty.

Note: We work here with a set of Strings. But the elements of the set could be anything at all; only the "hash function" would change.

What's wrong with using an array?

Keep set elements in b[0..n-1]

n | 4 | b | []@78



- Adding an element requires testing whether it is in the array. Expected time O(n)
- Removing an element requires moving elements down. Expected time O(n)
- Testing whether an element is in: expected time O(n)

Keeping the array sorted presents its own problems.

ArrayList and Vector implemented using arrays; same problems

Solution: Let an element appear anywhere in b

Keep set elements anywhere in b

b []@xy

A **hash function** says where to put an element

In example to right:

"abc" hashes to 0

"1$2" hashes to 100

"235" and "aaa" both hash to 60

Collision: string hashes to occupied place

Solution: Put in next available space

	[]@xy
0	"abc"
60	"235"
61	"aaa"
100	"1$2"
999	"xy"

Array elements: **null** or of type HashEntry

```
/** An instance is an element in hash array */
private static class HashEntry {
    public String  element;     // the element
    public boolean isInSet;     // = "element is in set"

    /** Constructor: an element that is in the set  iff b */
    public HashEntry( String e, boolean b) {
        element=  e;
        isInSet=  b;
    }
}
```

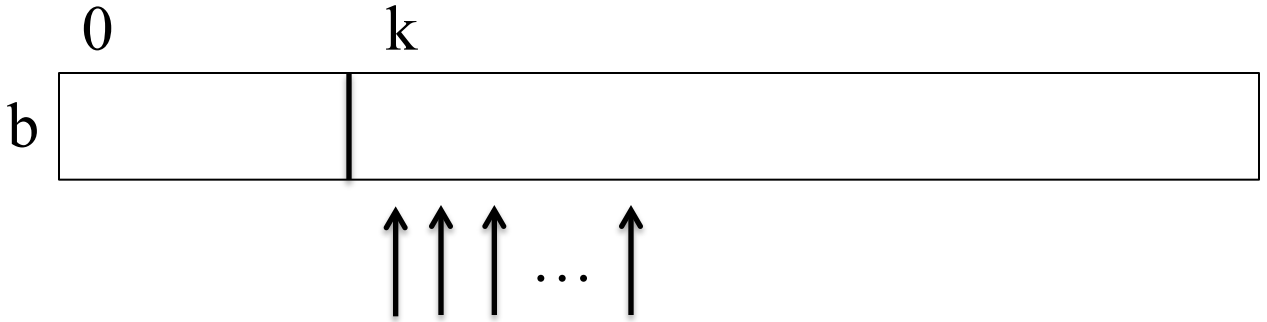
HashEntry object says whether it is in set. To remove an element, set field isInSet to **false**.

Hashing with **linear probing**

To add string "bc" to set:

```
int k=  hashCode("bc");
```

We discuss hash functions later

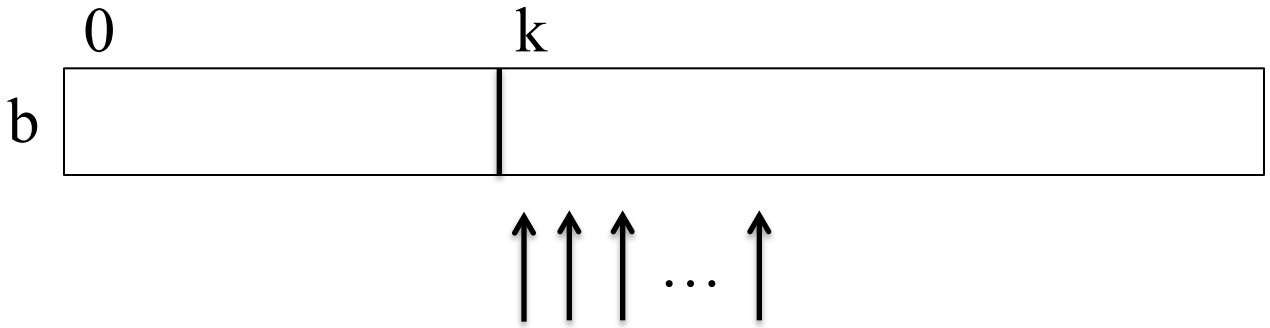


Check elements b[k], b[k+1], … until:
null or element containing "bc" is found

Probe: checking one element

Basic fact

```
int k=  hashCode(s);
```



Suppose k = hashCode(s).

Suppose s is in set.

Let b[j] be first (with wraparound) null element at or after b[k].

Then s is in one of elements b[k..j-1] (with wraparound)

Basic fact relies on never setting an element to **null**

procedure add

```
/** Add s to this set (if not in) */
public void add(String s) {
    int k=  hashCode(s);
    while (b[k] != null  &&  !b[k].element.equals(s))
        k= (k+1) % b.length();

    if (b[k] = = null) {
        b[k]=  new HashEntry(s, true);
        size= size+1;
        return;
    }

    // s is in b[k] –but it may not be in the set
    if (!b[k].isInSet) {
        b[k].isInSet=  true;
        size=  size + 1;
    }
}
```

% (remainder)
gives wraparound

procedure remove

```
/** Remove from this set (if it is in) */
public void remove(String s) {
    int k=  hashCode(s);
    while (b[k] != null  &&  !b[k].element.equals(s))
          k= (k+1) % b.length();

    if (b[k] = = null  ||  !b[k].isInSet) {
        return;

    // s is in b[k] and is in the set; remove it
    b[k].isInSet=  false;
    size=  size - 1;
}
```

Load factor

lf = (number of elements that are not null) / b.length

Estimate of how full array is:

close to 0: relatively empty Close to 1: too full

Somebody proved:

> Under certain independence assumptions (about the hash function), the average number of probes in adding an element is **1 / (1 – lf)**

Array half full? Addition expected to need only 2 probes!

E.g. size 2000, 1000 elements are **null**.
Only 2 probes! Wow!

Procedure rehash

```
/** Rehash array b */
 private void rehash( ) {
      HashEntry[] oldb= b;  // copy of array b
      // Create a new, empty array
        b= new HashEntry[nextPrime(4 * size)];
        size= 0;

      // Copy active elements from oldb to b
        for (int i= 0;  i != oldb.length;  i= i+1)
            if (oldb[i] !== null  &&  oldb[i].isInSet)
                b.add(oldb[i].element);
}
```

Size of new array: first prime larger than 4 * (size of set)
Why a prime? Next slides

Quadratic probing

Linear probing: Look at k, k+1, k+2 …

Clustering: because 2 strings that hash to k, k+1 have almost same probe sequence

Instead, use **quadratic probing**:

b[k]

b[k+1^2]

b[k+2^2]

b[k+3^2]

…

Removes primary clustering

Efficient calculation:

$(i+1)^2 - i^2$

= <arithmetic>

$2*i - 1$

Get to next probe with mult and add

Quadratic probing

b[k]

$b[k+1^2]$

$b[k+2^2]$

$b[k+3^2]$

…

Someone proved: If

1. Size of array is a prime
2. Load factor <= 1/2

Then

- A new element can be added
- Probe sequence never probes same elements twice

Two facts hold for linear probing even if size is not prime.
But quadratic probing requires prime size

Hash function

Want a hash function that doesn't put too many elements at the same position.

Class String has a good hash function

s.hashCode()

The specs define it as (with n the length of s):

$s[0]*31^{n-1} + s[1]*31^{n-1} + \ldots + s[n-1]$

Time is O(n)!
Extremely long strings? Create you own hash function,
But it's not easy to create a good one.

Java's hashcode-equals contract

HashCode and equals are implemented in class Object.

HashCode in Object: usually implemented by converting internal address (pointer) to an integer

General contract for HashCode:

- During an execution, c.hashCode() should consistently return same value unless info used in calculating c.equals(…) is changed
- c1.equals(c2) true? Then c1.hashCode() = c2.hashCode()
- c1.equls(c2) false? For best performance c1.hashCode() != c2.hashCode() —but not required.

Override equals? Then override hashCode also if you are going to use it.

Summary

We presented basics of hashing, although there are a few other ideas you should be aware of. We summarize, giving references to the text by Carrano and elsewhere for more information. Carrano does Hashing in Chapter 21, 523–546.

Describe basic idea of hashing (524–526).

- **hash table** (the array)
- **hash function**: Given search key, compute a **hash code**: an integer. Integer is then changed (Carrano says *compressed*) to be in range of hash table, usually using remainder function.
- **Perfect hash function**: maps each search key into a different integer that is an index in the hash table.
- **Good hash function properties**: (1) minimize collisions, (2) Be fast to compute.

Summary

Java hash functions: String provides hashCode function. It's >= 0. Each wrapper class provides hashCode function for the values that it wraps; for class Integer, it is the wrapped int, so it can be negative. (page 528–530).

Cryptographic hash function (*visit Wikipedia*). Produce a fixed-size bit string for an arbitrary block of data such that any change to the data will, with high probability, change the hash value. Critical for information security applications, like digital signatures. Not easy to come up with good ones. The widely used MD5 Message-Digest Algorithm (by Ron Rivest of MIT) produces a 16-byte hash value, but it has flaws.

Summary

Hash table size n: Best n is a prime > 2. Then compression of hash code using % n provides indices that are distributed uniformly in 0..n-1 (page 531).
Be careful with h % n: it not the modulus operation. If h < 0, h% n is in 1-n..0, so add n or use absolute value.

Load factor lf: Ratio of number of occupied hash-table elements to size of hash-table. Proved: for linear or quadratic probing, under certain independence conditions, the average number of probes in adding an element is at most **1 / (1 – lf)**. So if hash table is half full, only 2 probes expected! Keep it at most half full by making bigger hash table when necessary.

Summary

Collision: Occurs when two different search keys hash and are then compressed to same index. Two general ways to proceed:

1. **Open addressing**: Use
   - **Linear probing**. Has problem of primary clustering
   - **Quadratic probing**. Hash table size should be a prime
   - With both linear and quadratic probing, don't remove a deleted element from hash table. It must stay there with a flag indicating it is not in set.
2. **Separate chaining**: Entry in hash table is head of a linked list of all keys that hash to same index. Takes more space but eliminates many collisions (page 539–542).