IP Lecture 16: Bit Maps and Hash Tables

Joe Pitt-Francis

—with thanks to Mike Spivey & Gavin Lowe—

Sets of small integers

Suppose we want to represent a set of small integers, in the range `0..N-1`, where `N` is of a moderate size, say at most a few thousand.

For example, if we're writing a program to solve Sudoku puzzles, we might want to record, for each square, the set of values that could legally be placed there, i.e. a set of integers in the range `1..9`.

A particularly efficient way to do this is using a bit map, i.e. an array:

```
val a = new Array[Boolean](N);
```

such that `a(i)` is true iff `i` is in the set.

`Booleans` are initialised to `false` by default, so the above code will correspond to the empty set.

Alternatively, we could use a single bit per entry, hence the name "bit map".

Bit maps

```
/** A set of integers in the range [0..N). */
class BitMapSet(val N: Int){
  private val a = new Array[Boolean](N) // all false initially

  // a(i)=true iff i is in the set; i.e. this represents the set
  // { i | 0 <= i < N && a(i) }

  ...
}
```

Exercise: implement `add`, `remove`, `isIn` and `size` operations.

How can we implement `size` so that it's $O(1)$?

toString

If you use code such as `println(x)` where `x` is an object, Scala applies `toString` operation to `x` to convert it to a `String` (cf. Haskell's `show`).

Scala provides a default definition of `toString`, but we can override this. We would like to produce a string such as `{1, 3, 8}`. The following code does this. We need to be careful with commas.

```
/** Convert to a string. */
override def toString : String = {
  var st = "{" // build up the result in this
  var empty = true // Set to false on first element in string.
                   // Subsequent elements prefixed with ", "
  for(i <- 0 until N) if(a(i)){
    if(empty){ st += i.toString; empty = false }
    else st += ", "+i.toString
  }
  st+"}"
}
```

Testing for equality

We want to be able to test whether sets are equal (i.e. have the same value for `N` and contain the same elements). Scala provides a default definition of `equals`, in `Any`. However, the default definition will test whether the two sets are in fact represented by the same object, which isn't want we want.

The following function tests for equality as sets: i.e. it tests whether $abs(\texttt{this}) = abs(\texttt{that})$.

```
/** Test if this and that are equal. */
def equals(that: BitMapSet) : Boolean = {
  if(N != that.N) return false
  for(i <- 0 until N) if(a(i) != that.a(i)) return false
  true
}
```

Note that `private` variables of an object are visible to other objects of the same class.

Testing for equality

The default definition of `equals` takes an argument of type `Any`. We can override the default definition as follows.

```
override def equals(that: Any) : Boolean = that match {
  case s: BitMapSet => {
    if(N != s.N) return false
    for(i <- 0 until N) if(a(i) != s.a(i)) return false
    true
  }
  case _ => false
}
```

`that` matches the first pattern only if it is a `BitMapSet`. In this case, the variable `s` will be a `BitMapSet`.

Scala defines the operators `==` and `!=` in terms of `equals`, so we can now use those operators to compare `BitMapSets`.

Factory constructors

At present, if we want to create a set corresponding to `{2,5,7}` (with `N = 10`), we have to do something like

```
val s = new BitMapSet(10); s.add(2); s.add(5); s.add(7)
```

It would be nicer to just have to type

```
val s = BitMapSet(10)(2, 5, 7)
```

Factory constructors

The following code allows us to do this.

```
// Companion object
object BitMapSet{
  /** A new BitMapSet over {0..N-1}, containing xs */
  def apply(N: Int)(xs: Int*) : BitMapSet = {
    val s = new BitMapSet(N); for(x <- xs) s.add(x); s
  }
}
```

Recall that `BitMapSet(10)(2, 5, 7)` is shorthand for `BitMapSet.apply(10)(2, 5, 7)`, so it calls the above `apply` function.

The notation `xs: Int*` means that `apply` takes any number of `Int`-valued arguments.

`apply` is called a factory method, because it constructs new `BitMapSet`s. Its definition lives in the companion object because it is not an operation on a particular `BitMapSet`.

Bags

A bag (or multiset) is like a set except it may contain repetitions: the bag $\{\!|\, 3, 3 \,|\!\}$ is different from $\{\!|\, 3 \,|\!\}$.

A bag containing elements of type T can be thought of as a function of type $b : T \to \mathbb{N}$. The idea is that $b(x)$ gives the number of times that x appears in the bag.

If we want to represent a bag containing elements from `0..N-1` we could do so using an array

```
private val a = new Array[Int](N);
```

The idea is that `a(i)` records the number of times that `i` appears in the bag.

Most of the implementation is similar to the `BitMapSet` class.
Exercise: fill in the details.

Bags

But what if the underlying type isn't of the form `0..N-1`, for a moderately sized `N`?

For example, suppose we want to count the number of times words appear in a document; this corresponds to a bag of words (`String`s).

We will use a function

$$\texttt{hash} : T \to \{0 \mathinner{..} \texttt{N} - 1\}$$

We then implement `N` "buckets", each of which can hold several elements. If we want to add an element `x`, we add it to bucket `hash(x)`. If we subsequently want to search for `x`, we need do so only in bucket `hash(x)`.

If the results of `hash` are reasonably evenly distributed over $\{0 \mathinner{..} \texttt{N} - 1\}$, it is likely that each bucket will contain only about $\frac{1}{\texttt{N}}$ of the total elements, making searching quite quick.

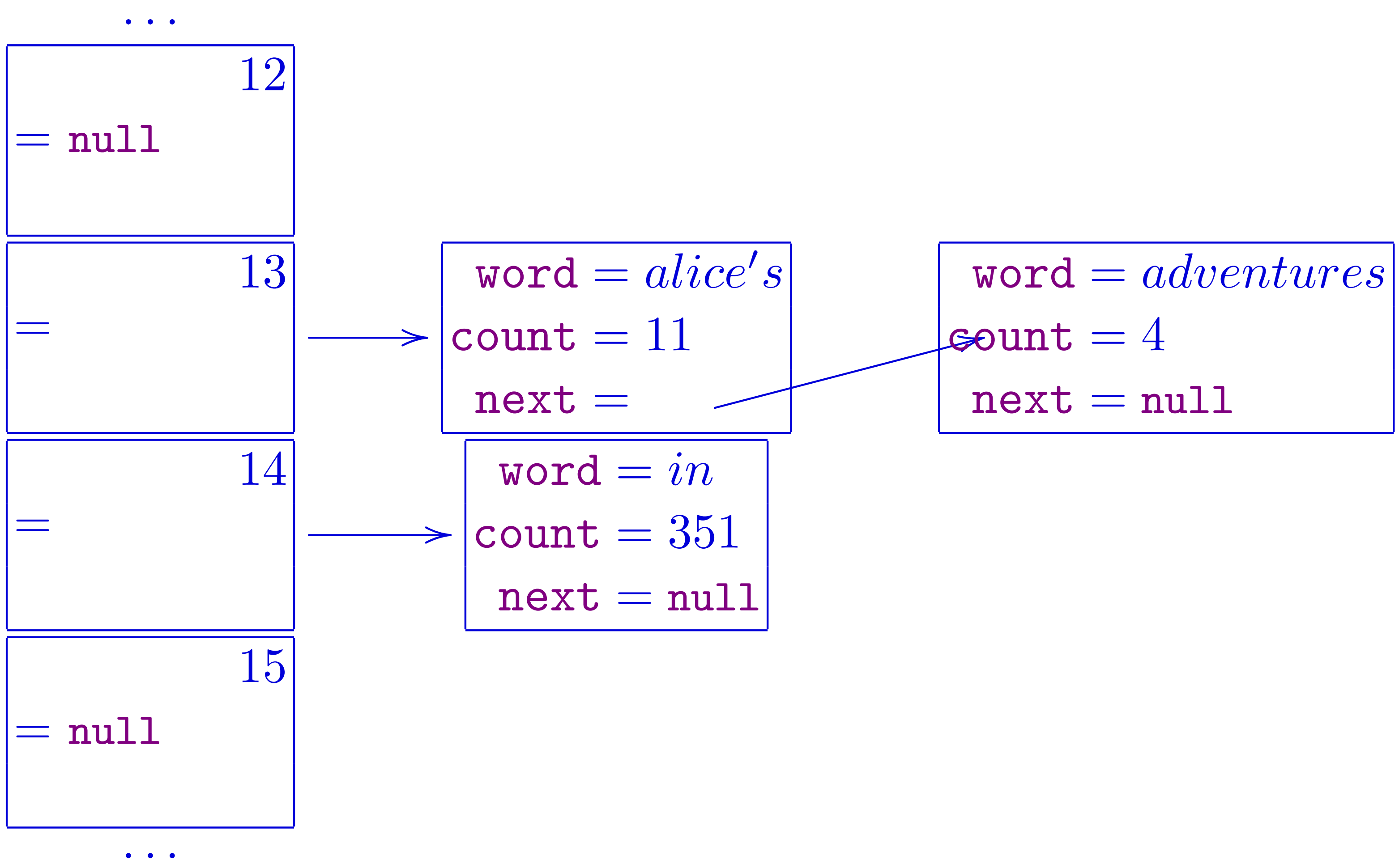
Hash tables

This data structure is known as a hash table.

Each bucket can be implemented as a linked list (without dummy headers, to save space). The hash table itself is then just an array (of size `N`) of such linked lists.

We will illustrate the idea by using a hash table to implement a bag of `String`s.

Hash table


...
12
= null
13
=
14
=
15
= null
...
word = alice's
count = 11
next =
word = adventures
count = 4
next = null
word = in
count = 351
next = null

The hash function

Writing a good hash function is something of a black art. We'll follow a fairly standard technique. The hash of a string $st = c_0 c_1 \ldots c_{n-1}$ will be the value of the polynomial

$$\big(p^n + c_0\texttt{.toInt} \times p^{n-1} + c_1\texttt{.toInt} \times p^{n-2} + \ldots + c_{n-1}\texttt{.toInt}\big) \bmod N$$

where p is an odd prime. We will take $p = 41$.

The above polynomial is equivalent (by Horner's Rule) to

$$\big((\ldots(((p + c_0\texttt{.toInt}) \times p + c_1\texttt{.toInt}) \times p + c_2\texttt{.toInt})\ldots) \times p + c_{n-1}\texttt{.toInt}\big) \\ \bmod N$$

Using laws of mod, this is equivalent to

$$(\ldots(((p + c_0\texttt{.toInt}) \bmod N \times p + c_1\texttt{.toInt}) \bmod N \times p + c_2\texttt{.toInt}) \bmod N \ldots \\ \times p + c_{n-1}\texttt{.toInt}) \bmod N$$

We calculate the result this way to avoid problems with overflow.

The hash function

We can implement the `hash` function in Scala using a loop:

```
private def hash(word: String) : Int = {
  var e = 1;
  for(c <- word) e = (e*41 + c.toInt) % N
  e
}
```

or more concisely using a `foldLeft`:

```
private def hash(word: String) : Int = {
  def f(e: Int, c: Char) = (e*41 + c.toInt) % N
  word.foldLeft(1)(f)
}
```

The hash table

In hash table we store `String`s together with a count for that `String`.

```
// Companion object
object HashBag{
  // Nodes for forming linked lists
  private class Node(val word: String, var count: Int, var next: Node)
}

class HashBag{
  private val N = 100   // # buckets in the hash table
  private var size_ = 0 // # distinct words stored

  // The hash function we will use
  private def hash(word: String) : Int = ...

  private val table = new Array[HashBag.Node](N) // the hash table
  ...
}
```

find

To either add or find the count of a word we need to search for a node containing that word. So let's write a function encapsulating that search.

```
/** Find node containing word in linked list starting at head, or
  * return null if word does not appear */
private def find(word: String, head: HashBag.Node) : HashBag.Node = {
  var n = head
  while(n != null && n.word != word) n = n.next
  n
}
```

add

To add a word `word`, we search for it in entry `h = hash(word)` of the table; if we find an entry for `word`, we increment its count; otherwise we create a new node.

```
/** Add an occurrence of word to the table */
def add(word: String) = {
  val h = hash(word)
  val n = find(word, table(h))
  if(n != null) n.count += 1
  else{
    table(h) = new HashBag.Node(word, 1, table(h))
    size_ += 1
  }
}
```

count

Finding the count for a word is similar.

```
/** The count stored for a particular word */
def count(word: String) : Int = {
  val h = hash(word)
  val n = find(word, table(h))
  if(n != null) n.count else 0
}
```

We could also define a `remove` operation.

Testing the bag and its load factor

```
import org.scalatest.FunSuite

class HashBagTest extends FunSuite{
  val bag  = new HashBag
  test("add"){
    bag.add("a"); bag.add("a");
    assert(bag.count("a")==2 && bag.count("b")==0)
    assert(bag.size == 1)
  }

  test("resize?"){
    /* Note: warning in "add" when load_factor>=3/4*/
    for(i <- 0 to 77) bag.add("element"+i)
    assert(bag.size == 79)
    for(i <- 0 to 77) assert(bag.count("element"+i)==1)
    assert(bag.count("a")==2 && bag.count("b")==0)
  }
}
```

Complexity

Each of the `add` and `count` operations involves traversing the list rooted at `table(h)` where `h = hash(word)`, so takes time $O(len)$ where len is the length of this list.

Let $loadFactor =$ `size_`/`N` be the average length of the lists. If the hash function's results are reasonably evenly distributed, then each operation will take time $O(loadFactor)$, on average.

If $loadFactor$ is bounded, the operations are $O(1)$ on average!

But as $loadFactor$ increases, the operations will become slower. A solution is to resize the hash table, say doubling the number `N` of buckets.

Hash tables in the Scala API

The Scala API includes both sets and mappings implemented with hash tables (`HashSet` and `HashMap`), in both mutable and immutable forms.

These assume that the type `T` of data that you're storing has a suitable definition for `hashCode`.

Summary

- Bit maps;
- Augmenting the state to make operations efficient;
- `toString`, `equals`;
- Factory constructors;
- Hash functions;
- Hash tables;
- Hash tables in the Scala API.
- Next time: Resizing hash tables, Trees.