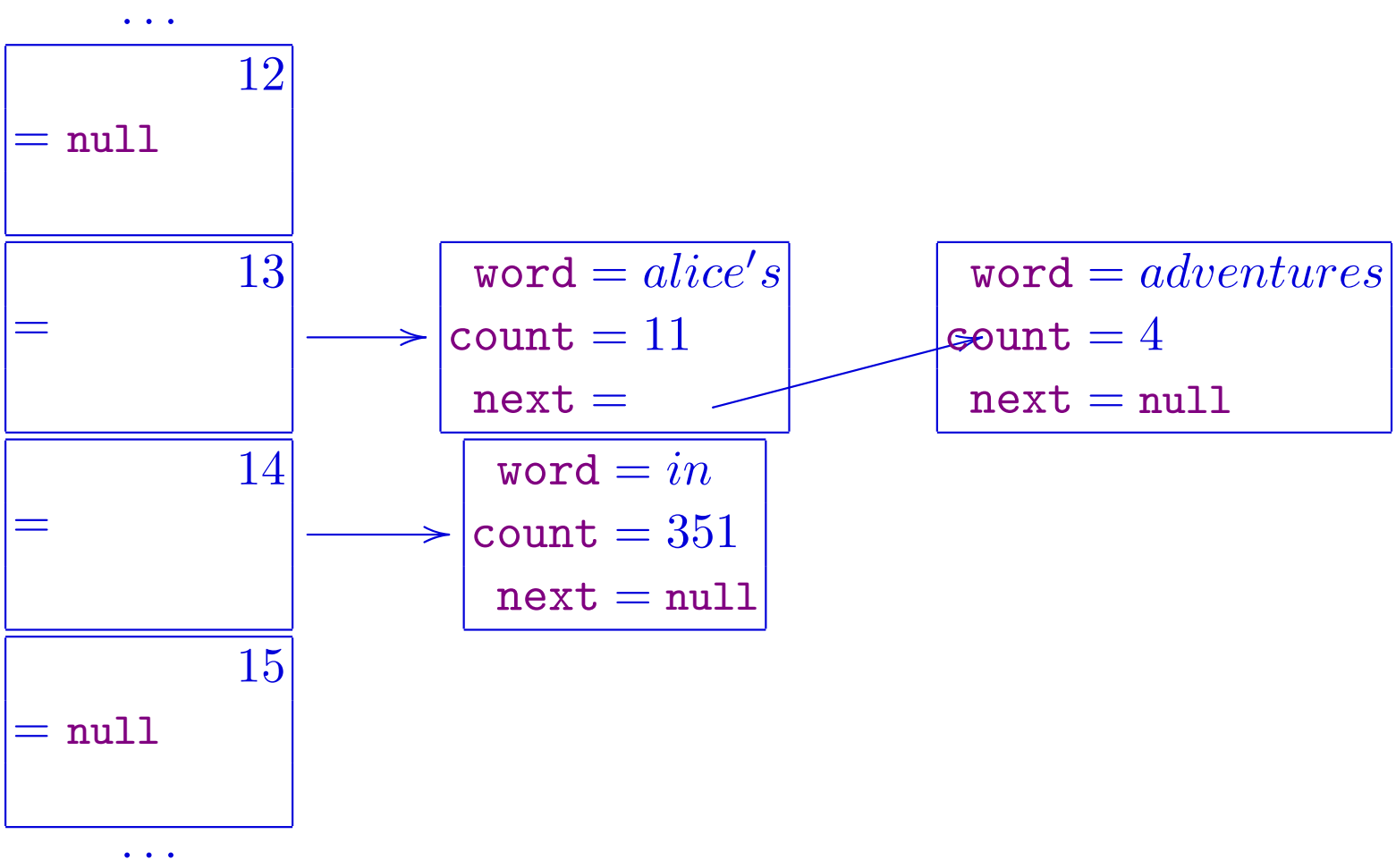
IP Lecture 17: Hash Tables and Binary Trees

Joe Pitt-Francis

—with thanks to Mike Spivey & Gavin Lowe—

Hash table


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

Complexity

Given a suitable hash function

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

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 $loadFactor = \texttt{size\_/N}$ 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.

Resizing

We now need to treat `N` as a variable. If the load factor reaches some value `MaxLoadFactor`, we resize the table.

```
/** # entries in the hash table */
private var N = 100 // Was "val"

/** Max load factor allowed */
private val MaxLoadFactor = 0.75

/** max # elements before resizing needed */
private var threshold = (N*MaxLoadFactor).toInt

def add(word: String) = {
  if(size >= threshold) resize
  ... // as before
}
```

Resizing the table

To resize the table, we create a new table and copy the data across.

```
/** resize the hash table, by doubling its size */
private def resize = {
  val oldN = N; N = 2*N; threshold = 2*threshold
  // Note "table" must be var for the reassignment:
  val oldTable = table; table = new Array[HashBag.Node](N)
  // copy entries
  for(i <- 0 until oldN){
    var n = oldTable(i) // iterate over list, copying entries
    while(n != null){
      val h = hash(n.word)
      table(h) = new HashBag.Node(n.word, n.count, table(h))
      n = n.next
    }
  }
}
```

Maybe we should store the raw hash to avoid recalculating it.

Testing the bag for resizing

```
import org.scalatest.FunSuite

class HashBagTest extends FunSuite{
  val rbag = new HashBagResizable
  test("add resizable"){
    rbag.add("a"); rbag.add("a")
    assert(rbag.count("a")==2 && rbag.count("b")==0)
  }

  test("resize resizable"){
    for(i <- 0 to 200) rbag.add("element"+i)
    assert(rbag.N === 400) // requires N to be public
    for(i <- 201 to 500) rbag.add("element"+i)
    assert(rbag.N === 800) // requires N to be public
    for(i <- 0 to 500) assert(rbag.count("element"+i)==1)
    assert(rbag.count("a")==2 && rbag.count("b")==0)
  }
}
```

Amortized complexity

Resizing is an expensive operation: iterating through the buckets takes time $O(\texttt{N})$; iterating through all the entries takes time $O(\texttt{size})$; but $\texttt{size} \approx \texttt{MaxLoadFactor} \times \texttt{N}$, so this is $O(\texttt{N})$ in total.

However, we can share the cost of the resizing out between all of the `add` operations that have happened since the last resizing. There must have been at least $\texttt{size}/2 \approx \texttt{N} \times \texttt{MaxLoadFactor}/2$ such `add` operations. Hence the average cost for each such operation is $O(1)$.

So all the operations are amortized $O(1)$!

Hash codes

All classes include a function

```
def hashCode : Int
```

Scala provides a default implementation of `hashCode`, in the class `Any`. Most classes in the API override this to provide appropriate definitions, giving a hash code for the object in question (as an `Int`).

We can use this to define a more polymorphic version of `hash`:

```
private def hash(x: T) : Int = x.hashCode.abs % N
```

Hash codes

When defining a new class, it makes sense to override the default implementation of `hashCode`. In other cases, for an object with fields `a`, `b`, `c`, `d` and `e`, a suggested hash functon is

$$p^5 + \texttt{a.hashCode} \times p^4 + \texttt{b.hashCode} \times p^3 + \\ \texttt{c.hashCode} \times p^2 + \texttt{d.hashCode} \times p + \texttt{e.hashCode}$$

where p is an odd prime; and similarly for other numbers of fields.

The hash code should agree with your definition of equality: if `this.equals(that)` then `this.hashCode` should equal `that.hashCode`.

If you define a class as a `case` class, the compiler will create a suitable definition for you.

Hash table variations

The idea is always to keep the number of collisions (items which hash to the same number) as low as possible. There are two main ways to resolve collisions when they occur.

1. Chaining.

   Using a hash table in which each entry is a reference to a (small) linked list is known as chaining or open hashing.

   This is the normal way hash tables are implemented.

2. Probing.

   Rather than putting elements in a linked list, all elements could be stored in the array itself. When adding a new element would create a collision, we search for a nearby empty slot (either linearly or using a second hash function). The process of probing for an empty slot needs to be predictable and terminate quickly.

Exercise: implement linear probing.

Binary trees

In the Functional Programming course, you studied binary search trees. We can also implement binary search trees in Scala.

We will use a binary search tree to implement a bag of strings; but very similar techniques could be used to implement a set or a mapping.

The abstract state

Abstractly, a bag of strings is a mapping

state: $count : String \to Int$

DTI: $\{st \mid count(st) > 0\}$ is finite

such that $count(st)$ gives the number of occurrences of st in the bag.

Initially, the bag is empty

init: $\forall st \bullet count(st) = 0$

Specifying the operations

```
/** Add an occurrence of word
  * post: count = count_0 ⊕ {word → count_0(word) + 1} */
def add(word:String)

/** Find the count stored for word
  * post: count = count_0 ∧ returns count(word) */
def count(word:String) : Int
```

Specifying the operations

```
/** Print the bag
  * post: count = count_0 ∧
  *       prints word → count(word)
  *       for each word s.t.  count(word) > 0,
  *       in alphabetical order */
def printBag

/** Delete one occurrence of word
  * post: if count_0(word) = 0 then count = count_0
  *       else count = count_0 ⊕ {word → count_0(word) − 1} */
def delete(word:String)
```

Binary trees

In Haskell we might define a suitable type of binary trees as

```
data BSTree = Tree String Int BSTree BSTree | Null
```

In Scala, we can define a similar type.

```
object BinaryTreeBag{
  private class Tree(var word: String, var count: Int,
                     var left: Tree, var right: Tree)
}
```

(We put the definition inside the companion object.)

However, we can change the values of fields of a `Tree`, so our trees will be mutable (whereas Haskell trees are immutable); this provides for some different algorithms.

A class for binary trees

```
class BinaryTreeBag{
  // Define shorthands, so we can write "Tree" rather than
  // "BinaryTreeBag.Tree"
  private type Tree = BinaryTreeBag.Tree
  private def Tree(word: String, count: Int, left: Tree, right: Tree) =
    new BinaryTreeBag.Tree(word, count, left, right)

  private var root : Tree = null
  ...
}
```

The abstraction function

Define $T(t)$ to be the set of `Tree` nodes reachable from t.

$$T(\texttt{null}) = \{\}$$

$$T(t) = \{t\} \cup T(t.\texttt{left}) \cup T(t.\texttt{right})$$

Each object represents

Abs: $count =$

$$\{st \to 0 \mid st \in \texttt{String}\} \oplus \{t.\texttt{word} \to t.\texttt{count} \mid t \in T(\texttt{root})\}$$

(The first term is necessary because we've specified *count* to be a *total* function, whereas the second term produces only a *partial* function.)

The datatype invariant

The datatype invariant captures that: the tree satisfies the normal ordering property for a binary search tree; we only store information about words that have at least one occurrence in the bag; and the tree is acyclic and finite.

DTI: $\forall t \in T(\texttt{root}) \bullet$

$(\forall t' \in T(t.\texttt{left}) \bullet t'.\texttt{word} < t.\texttt{word}) \wedge$

$(\forall t' \in T(t.\texttt{right}) \bullet t'.\texttt{word} > t.\texttt{word}) \wedge$

$t.\texttt{count} > 0 \wedge$

$t \notin T(t.\texttt{left}) \cup T(t.\texttt{right}) \wedge$

T(root) is finite

Searching in the tree

We can write a function `count` that returns the number of occurrences of a given word in the bag.

Here's a recursive version.

```
/** Find the count stored for a particular word */
def count(word: String) : Int = countInTree(word, root)

/** Find the count stored for a particular word, within t */
private def countInTree(word: String, t: Tree) : Int =
  if(t == null) 0
  else if(word == t.word) t.count
  else if(word < t.word) countInTree(word, t.left)
  else countInTree(word, t.right)
```

Summary

- Resizing hash tables;
- `hashCode`;
- Representing binary trees using reference-linked nodes;
- Searching in binary search tree.
- Next time: Implementing and testing tree operations.