Maps

- Definitions — what is a map (or function)?
- Specification
- List-based representation (singly linked)
- Sorted block representation

Reading: Weiss, Section 6.8

What is a Map (or Function)?

Some definitions...

relation — set of n-tuples

eg. $\{\langle 1, i, a\rangle, \langle 2, ii, b\rangle, \langle 3, iii, c\rangle, \langle 4, iv, d\rangle, \ldots\}$

binary relation — set of pairs (2-tuples)

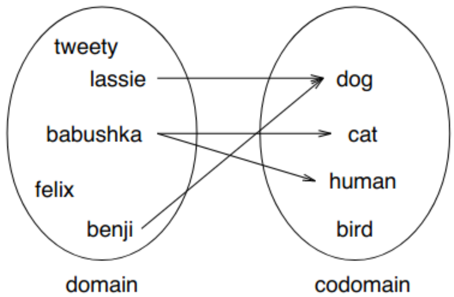
eg. $\{\langle lassie, dog\rangle, \langle babushka, cat\rangle, \langle benji, dog\rangle, \langle babushka, human\rangle, \ldots\}$

domain — set of values which can be taken on by the first item of a binary relation

eg. $\{lassie, babushka, benji, felix, tweety\}$

codomain — set of values which can be taken on by the second item of a binary relation

eg. $\{dog, cat, human, bird\}$



dog is called the **image** of *lassie* under the relation

map (or function) — binary relation in which each element in the domain is mapped to *at most one* element in the codomain (*many-to-one*)

eg.

$$\text{Affiliation} = \{ \begin{array}{cccc} \langle & \text{Turing} & , & \text{Manchester} & \rangle \\ \langle & \text{Von Neumann} & , & \text{Princeton} & \rangle \\ \langle & \text{Knuth} & , & \text{Stanford} & \rangle \\ \langle & \text{Minsky} & , & \text{MIT} & \rangle \\ \langle & \text{Dijkstra} & , & \text{Texas} & \rangle \\ \langle & \text{McCarthy} & , & \text{Stanford} & \rangle\} \end{array}$$

Shorthand notation: eg. affiliation(Knuth) = Stanford

partial map — not every element of the domain has an image under the map (ie, the image is undefined for some elements)

Aside: Why Study Maps?

A Java method is a function or map — why implement our own map as an ADT?

- Create, modify, and delete maps during use.
- a map of affiliations may change over time — Turing started in Cambridge, but moved to Manchester after the war.

  A Java program cannot modify itself (and therefore its methods) during execution (some languages, eg Prolog, can!)
- Java methods just return a result — we want more functionality (eg. ask "is the map defined for a particular domain element?")

Map Specification

1. **Constructor**
2. *Map()*: create a new map that is undefined for all domain elements.
3. **Checkers**
4. *isEmpty()*: return *true* if the map is empty (undefined for all domain elements), *false* otherwise.
5. *isDefined(d)*: return *true* if the image of d is defined, *false* otherwise.
6. **Manipulators**
7. *assign(d,c)*: assign c as the image of d.
8. *image(d)*: return the image of d if it is defined, otherwise throw an exception.
9. *deassign(d)*: if the image of d is defined return the image and make it undefined, otherwise throw an exception.

List-based Representation

A map can be considered to be a list of pairs. Providing this list is *finite*, it can be implemented using one of the techniques used to implement the list ADT.

Better still, it can be built *using* the list ADT!

(Providing it can be done efficiently — recall the example of *overwrite*, using *insert* and *delete*, in a text editor based on the list ADT.)

Question: Which List ADT should we use?

- Require arbitrarily many assignments.
- Do we need *previous*?

Implementation. . .

```
public class MapLinked {

  private ListLinked list;

  public MapLinked () {
    list = new ListLinked();
  }








}
```

Pairs

We said a (finite) map could be considered a list of pairs — need to define a Pair object…

```
public class Pair {

  public Object item1;       // the first item (or domain item)
  public Object item2;       // the second item (or codomain item)

  public Pair (Object i1, Object i2) {
    item1 = i1;
    item2 = i2;
    }
```

```
  // determine whether this pair is the same as the object passed
  // assumes appropriate ``equals'' methods for the components
  public boolean equals(Object o) {
    if (o == null)  return false;
    else if (!(o instanceof Pair))  return false;
    else  return item1.equals( ((Pair)o).item1) &&
                 item2.equals( ((Pair)o).item2);
  }

  // generate a string representation of the pair
  public String toString() {
    return "< "+item1.toString()+" , "+item2.toString()+" >";
  }
}
```

Example — Implementation of *image*

```
public Object image (Object d) throws ItemNotFound {
  WindowLinked w = new WindowLinked();
  list.beforeFirst(w);
  list.next(w);
  while (!list.isAfterLast(w) &&
         !((Pair)list.examine(w)).item1.equals(d) )  list.next(w);
  if (!list.isAfterLast(w))  return ((Pair)list.examine(w)).item2;
  else throw new ItemNotFound("no image for object passed");
}
```

Notes:

1. `!list.isAfterLast(w)` must precede `list.examine(w)` in the condition for the loop — why??

2. Note use of parentheses around casting so that the field reference (eg `.item1`) applies to the cast object (`Pair` rather than `Object`).

3. Assumes appropriate *equals* methods for each of the items in a pair.

Performance

Map and *isEmpty* make trivial calls to constant-time list ADT commands.

The other four operations all require a sequential search within the list $\rightarrow$ linear in the size of the defined domain ($O(n)$)

Performance using (singly linked) List ADT

Operation	
Map	1
isEmpty	1
isDefined	n
assign	n
image	n
deassign	n

If the maximum number of pairs is predefined, and we can specify a total ordering on the domain, better efficiency is possible...

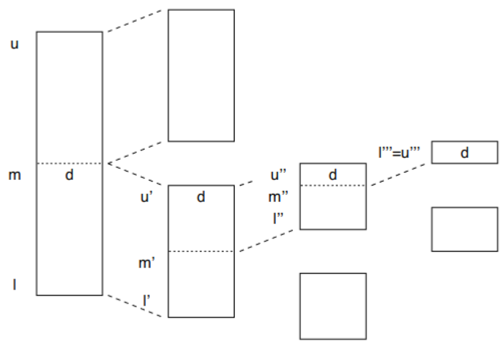
Sorted-block Representation

Some of the above operations take linear time because they need to search for a domain element. The above program does a linear search.

Q: Are any more efficient searches available for arbitrary *linked* list?

Binary Search

An algorithm for binary search...



Assume `block` is defined as:

```
private Pair[] block;
```

Then binary search can be implemented as follows…

```
protected int bSearch (Object d, int l, int u) {
  if (l == u) {
    if (d.toString().compareTo(block[l].item1.toString()) == 0)
      return l;
    else return -1;
  }
  else {
    int m = (l + u) / 2;
    if (d.toString().compareTo(block[m].item1.toString()) <= 0)
      return bSearch(d,l,m);
    else  return bSearch(d,m+1,u);
  }
}
```

Note: `compareTo` is an instance method of String — returns 0 if its argument matches the String, a value < 0 if the String is lexicographically less than the argument, and a value > 0 otherwise.

Exercise: Can `bSearch` be implemented using only the abstract operations of the list ADT?

Performance of Binary Search

One way of looking at the problem, to get a feel for it, is to consider the biggest list of pairs we can find a solution for with m calls to `bSearch`.

Calls to `bSearch`	Size of list
1	1
2	1 + 1
3	2 + 1 + 1
4	4 + 2 + 1 + 1
⋮	
m	$(2^{m-2} + 2^{m-3} + \cdots + 2^1 + 2^0) + 1$
	$= (2^{m-1} - 1) + 1$
	$= 2^{m-1}$

It can be shown (see Exercises) that T_n is $O(\log n)$.

Comparative Performance of Operations

isDefined and *image* simply require binary search, therefore they are $O(\log n)$ — much better than singly linked list representation.

However, since the block is sorted, both *assign* and *deassign* may need to move blocks of items to maintain the order. Thus they are

$$\max(O(\log n), O(n)) = O(n).$$

In summary...

Operation	Linked List	Sorted Block
Map	1	1
isEmpty	1	1
isDefined	n	$\log n$
assign	n	n
image	n	$\log n$
deassign	n	n

Sorted block may be best choice if:

1. map has fixed maximum size
2. domain is totally ordered
3. map is fairly static — mostly reading (*isDefined, image*) rather than writing (*assign, deassign*)

Otherwise linked list representation is probably better.

Summary

- A map (or function) is a many-to-one binary relation.
- Implementation using linked list
  - can be arbitrarily large
  - reading from and writing to the map takes linear time
- Sorted block implementation
  - fixed maximum size
  - requires ordered domain
  - reading is logarithmic, writing is linear