Maps and Binary Search

- Definitions — what is a map (or function)?
- Specification
- List-based representation (singly linked)
- Sorted block representation
  - binary search, performance of binary search
- Performance comparison

Reading: Lambert and Osborne, Sections 13.1, 13.3, and 10.2
1. 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 \}\n
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 \}\n
domain — set of values which can be taken on by the first item of a binary relation
eg.  \{lassie, babushka, benji, felix, tweety\}\n
codomain — set of values which can be taken on by the second item of a binary relation
eg.  \{dog, cat, human, bird\}
Example

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.

\[
\text{Affiliation} = \{ \langle \text{Turing}, \text{Manchester} \rangle, \langle \text{Von Neumann}, \text{Princeton} \rangle, \langle \text{Knuth}, \text{Stanford} \rangle, \langle \text{Minsky}, \text{MIT} \rangle, \langle \text{Dijkstra}, \text{Texas} \rangle, \langle \text{McCarthy}, \text{Stanford} \rangle \}\n\]

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)
2. **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.
  
  eg. 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?”)
3. Map Specification

- **Constructor**
  1. `Map()`: create a new map that is undefined for all domain elements.

- **Checkers**
  2. `isEmpty()`: return `true` if the map is empty (undefined for all domain elements), `false` otherwise.
  3. `isDefined(d)`: return `true` if the image of `d` is defined, `false` otherwise.

- **Manipulators**
  4. `assign(d,c)`: assign `c` as the image of `d`.
  5. `image(d)`: return the image of `d` if it is defined, otherwise throw an exception.
  6. `deassign(d)`: if the image of `d` is defined return the image and make it undefined, otherwise throw an exception.
4. 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...

```java
public class MapLinked {

    private ListLinked list;

    public MapLinked () {
        list = new ListLinked();
    }
}
```
4.1 Pairs

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

```java
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()+" >";
}
}
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.
4.3 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 ⇒ linear in the size of the defined domain \(O(n)\)

**Performance using (singly linked) List ADT**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Map</em></td>
<td>1</td>
</tr>
<tr>
<td><em>isEmpty</em></td>
<td>1</td>
</tr>
<tr>
<td><em>isDefined</em></td>
<td>(n)</td>
</tr>
<tr>
<td><em>assign</em></td>
<td>(n)</td>
</tr>
<tr>
<td><em>image</em></td>
<td>(n)</td>
</tr>
<tr>
<td><em>deassign</em></td>
<td>(n)</td>
</tr>
</tbody>
</table>

If the maximum number of pairs is predefined, and we can specify a total ordering on the domain, better efficiency is possible...
5. 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?
5.1 Party Games...

Q: I’ve chosen a number between 1 and 1000. What is it?

Q: I’ve chosen a number between 1 and 1000. If you make an incorrect guess I’ll tell whether its higher or lower. You have 10 guesses. What is it?

Q: I’m going to choose a number between 1 and $n$. You have 5 guesses. What is the maximum value of $n$ for which you are certain to get my number right?

Exercise: Write a recursive Java method $guessrange(m)$ that returns the maximum number $n$ for which you can always obtain a correct answer with $m$ guesses.
5.2 Binary Search

An algorithm for binary search…
Assume block is defined as:

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

Then binary search can be implemented as follows...
// recursive implementation of binary search
// uses String representations generated by toString()
// for comparison
// returns index to the object if found, or -1 if not found

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?
5.3 Performance of Binary Search

We will illustrate performance in two ways.

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.

<table>
<thead>
<tr>
<th>Calls to bSearch</th>
<th>Size of list</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1 + 1</td>
</tr>
<tr>
<td>3</td>
<td>2 + 1 + 1</td>
</tr>
<tr>
<td>4</td>
<td>4 + 2 + 1 + 1</td>
</tr>
<tr>
<td>\vdots</td>
<td>( (2^{m-2} + 2^{m-3} + \cdots + 2^1 + 2^0) + 1 )</td>
</tr>
<tr>
<td>( m )</td>
<td>( (2^{m-1} - 1) + 1 )</td>
</tr>
<tr>
<td></td>
<td>( = 2^{m-1} )</td>
</tr>
</tbody>
</table>

That is, \( n = 2^{m-1} \) or \( m = \log_2 n + 1 \).
This view ignores the “intermediate” size lists — those which aren’t a maximum size for a particular number of calls.

An alternative is to look at the number of calls needed for increasing input size. Can be expressed as a recurrence relation...
\[
\begin{align*}
T_1 &= 1 \\
T_2 &= 1 + T_1 = 2 \\
T_3 &= 1 + T_2 = 3 \\
T_4 &= 1 + T_2 = 3 \\
T_5 &= 1 + T_3 = 4 \\
T_6 &= 1 + T_3 = 4 \\
T_7 &= 1 + T_4 = 4 \\
T_8 &= 1 + T_4 = 4 \\
T_9 &= 1 + T_5 = 5 \\
\vdots
\end{align*}
\]

The rows for which \( n \) is an integer power of 2...

\[
\begin{align*}
T_1 &= 1 \\
T_2 &= 1 + T_1 = 2 \\
T_4 &= 1 + T_2 = 3 \\
T_8 &= 1 + T_4 = 4 \\
\vdots
\end{align*}
\]

\ldots correspond to those in the earlier table.
For these rows we have

\[
\begin{align*}
T_{2^0} &= 1 \\
T_{2^m} &= 1 + T_{2^{m-1}} \\
&= 1 + 1 + T_{2^{m-2}} \\
&\vdots \\
&= 1 + 1 + \cdots + 1 \\
&= m + 1
\end{align*}
\]

Substituting \( n = 2^m \) or \( m = \log_2 n \) once again gives

\[
T_n = \log_2 n + 1.
\]

What about the cases where \( n \) is not an integer power of 2?

⇒ Exercises.

It can be shown (see Exercises) that \( T_n \) is \( O(\log n) \).
6. 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...

<table>
<thead>
<tr>
<th>Operation</th>
<th>Linked List</th>
<th>Sorted Block</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Map</code></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><code>isEmpty</code></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><code>isDefined</code></td>
<td>$n$</td>
<td>$\log n$</td>
</tr>
<tr>
<td><code>assign</code></td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td><code>image</code></td>
<td>$n$</td>
<td>$\log n$</td>
</tr>
<tr>
<td><code>deassign</code></td>
<td>$n$</td>
<td>$n$</td>
</tr>
</tbody>
</table>
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.
7. Arrays as Maps

We have seen two representations for maps:

- linked list — linear time accesses.
- sorted block — logarithmic for reading, linear for writing.

One very frequently used subtype of the map is an array. An array is simply a map (function) whose domain is a cross product of (that is, tuples from) sets of ordinals.
We will assume all domain items are tuples of integers.

ey. The array

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>6.6</td>
<td>2.8</td>
<td>0.4</td>
<td>6.0</td>
<td>0.1</td>
</tr>
<tr>
<td>true</td>
<td>3.4</td>
<td>7.2</td>
<td>9.6</td>
<td>4.0</td>
<td>9.9</td>
</tr>
</tbody>
</table>

could be represented by the map

\[
\{ \langle 0, 1 \rangle, 6.6 \}, \langle 0, 2 \rangle, 2.8 \}, \langle 0, 3 \rangle, 0.4 \}, \ldots , \langle 1, 4 \rangle, 4.0 \}, \langle 1, 5 \rangle, 9.9 \} .
\]
We will also assume the arrays are bounded in size, so we can store the items in a contiguous block of memory locations. (This can be simulated in Java using a 1-dimensional array.)

An *addressing function* can be used to translate the array indices into the actual location of the item in the block.

Accesses are more efficient for this subtype of maps — *constant time in all operations*.

⇒ good example of a subtype over which operations are more efficient.
7.1 Specification

- **Constructors**
  1. `Array()`: creates a new array that is undefined everywhere.

- **Manipulators**
  2. `assign(d, c)`: assigns `c` as the image of `d`.
  3. `image(d)`: returns the image of tuple `d` if it is defined, otherwise throws an exception.
7.2 Lexicographically Ordered Representations

Lexicographic Ordering with 2 Indices

Pair \( \langle i, j \rangle \) is lexicographically earlier than \( \langle i', j' \rangle \) if \( i < i' \) or \( (i = i' \text{ and } j < j') \).

Best illustrated by an array with indices of type char:
- first index: \( a, \ldots, d \)
- second index: \( a, \ldots, e \)

Then entries are indexed in the order

\[ \langle a, a \rangle, \langle a, b \rangle, \langle a, c \rangle, \langle a, d \rangle, \langle a, e \rangle, \langle b, a \rangle, \langle b, b \rangle, \ldots \langle d, d \rangle, \langle d, e \rangle \]

\[ \Rightarrow \text{ ‘alphabetic’ order (lexicon} \approx \text{ dictionary).} \]
Also called \textit{row-major} order.

Implementation straightforward — indexed block (from 1 to 20 in the example).

Wish to access entries in constant time.

Addressing function: \( \alpha : 1..m \times 1..n \to \mathbb{N} \)

\[
\alpha(i, j) = (i - 1) \times n + j \quad 1 \leq i \leq m, \ 1 \leq j \leq n
\]
Reverse-lexicographic Order

Similar to lexicographic, but indices swapped around...

Pair \( \langle i, j \rangle \) is reverse-lexicographically earlier than \( \langle i', j' \rangle \) if \( j < j' \) or \( (j = j' \text{ and } i < i') \).

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>13</td>
<td>17</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>6</td>
<td>10</td>
<td>14</td>
<td>18</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td>7</td>
<td>11</td>
<td>15</td>
<td>19</td>
</tr>
<tr>
<td>d</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

Also called column-major order.

Addressing function:

\[
\alpha(i, j) = (j - 1) \times m + i \quad 1 \leq i \leq m, \ 1 \leq j \leq n
\]
7.3 Shell-ordered Representation

An alternative to lexicographic ordering that has advantages in terms of extendibility.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>10</td>
<td>17</td>
</tr>
<tr>
<td>b</td>
<td>4</td>
<td>3</td>
<td>6</td>
<td>11</td>
<td>18</td>
</tr>
<tr>
<td>c</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>12</td>
<td>19</td>
</tr>
<tr>
<td>d</td>
<td>16</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
</tr>
</tbody>
</table>

Built up shell by shell.

The \( k \)-th shell contains indices \( \langle i, j \rangle \) such that \( k = \max(i, j) \).
Notice that the $k$th shell “surrounds” a block containing $(k - 1)^2$ cells, and forms a block containing $k^2$ cells.

$\Rightarrow$ To find entries in the first half of the shell, add to $(k - 1)^2$. To find entries in the second half of the shell, subtract from $k^2$.

Addressing function:

$$\alpha(i, j) = \begin{cases} 
(k - 1)^2 + i & i < k \\
 k^2 + 1 - j & \text{otherwise} 
\end{cases} \quad k = \max(i, j).$$
Disadvantage

May waste a lot of space. Worst case is a one-dimensional array of size $n$ — wastes $n^2 - n$ cells.

A related problem occurs with all these representations when only a small number of the entries are used.

eg. matrices in which most entries are zero.

In this case more complex schemes can be used — trade space for performance. See Wood, Sec. 4.4.

Advantage

- *All* arrays use the same addressing function — independent of number of rows and columns.
- Extendibility . . .
7.4 Extendibility

In lexicographic ordering new rows can be added (if memory is available) \textit{without changing} the values assigned to existing cells by the addressing function.

\[
\alpha(i, j) = (i - 1) \times \frac{n}{\text{no change}} + j \quad 1 \leq i \leq m, \ 1 \leq j \leq n
\]

We say the lexicographic addressing function is \textit{row extendible}.

Adding a row takes \(O(\text{size of row})\).
However it is not *column extendible*. Adding a new column means changing the values, *and hence locations*, of existing entries.

**Q:** What is an example of a worst case array for adding a column?

This is $O(\text{size of array})$ time operation.
Similarly, reverse lexicographic ordering is column extendible.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>13</td>
<td>17</td>
<td>21</td>
<td>25</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>6</td>
<td>10</td>
<td>14</td>
<td>18</td>
<td>22</td>
<td>26</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td>7</td>
<td>11</td>
<td>15</td>
<td>19</td>
<td>23</td>
<td>27</td>
</tr>
<tr>
<td>d</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>24</td>
<td>28</td>
</tr>
</tbody>
</table>

\[ \alpha(i, j) = (j - 1) \times m + i \quad 1 \leq i \leq m, \ 1 \leq j \leq n \]

\[ \text{no change} \]

\[ \text{...but not row extendible.} \]
Shell ordering, on the other hand, is both row and column extendible...

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>10</td>
<td>17</td>
<td>26</td>
</tr>
<tr>
<td>b</td>
<td>4</td>
<td>3</td>
<td>6</td>
<td>11</td>
<td>18</td>
<td>27</td>
</tr>
<tr>
<td>c</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>12</td>
<td>19</td>
<td>28</td>
</tr>
<tr>
<td>d</td>
<td>16</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>20</td>
<td>29</td>
</tr>
<tr>
<td>e</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>36</td>
<td>35</td>
<td>34</td>
<td>33</td>
<td>32</td>
<td>31</td>
</tr>
</tbody>
</table>

This is because the addressing function is independent of $m$ and $n$...

$$\alpha(i, j) = \begin{cases} 
(k - 1)^2 + i & i < k \\
(k^2 + 1 - j) & \text{otherwise}
\end{cases}$$

for $1 \leq i \leq m$, $1 \leq j \leq n$. 
7.5 Performance

<table>
<thead>
<tr>
<th>Operation</th>
<th>Lexicographic</th>
<th>Reverse-lexicographic</th>
<th>Shell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Assign</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Image</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Accesses are more efficient for this subtype of maps — *constant time in all operations*.

Disadvantages:

- restricted domain (integers).
- lexicographical based representations are not easily extendible — they require moving elements should additional columns/rows be added.
- shell-ordered representation waste space for non-square arrays.
8. 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
- Arrays are a commonly used subtype of maps which can be treated more efficiently
  - implemented using a 1D block of memory and an addressing function