Linked Data Structures I: Singly-Linked Lists
What’s Wrong With Arrays?

• Java arrays do not “play well” with generics, as we have seen
  – This is the reason for the OSU CSE Components `Array` family

• A Java array (or `Array` from the OSU CSE components) is not ideally suited as a data representation of the various `collection types`
  – Similarly for the Java `collections framework`
What's Wrong With Arrays?

Java arrays do not "play well" with generics, as we have seen. This is the reason for the OSU CSE Components Array family.

A Java array (or Array from the OSU CSE Components) is not ideally suited as a data representation of the various "collection" types.

- Similarly for the Java collections framework

Any type whose abstract mathematical model involves a string or set or multiset (or tree or binary tree?) is a "collection" type.
What’s Wrong With Arrays?

- Java arrays do not “play well” with generics, as we have seen.
- This is the reason for the OSU CSE Array family.

- A Java array (or Array from the OSU CSE components) is not ideally suited as a data representation of the various collection types.
  - Similarly for the Java collections framework.

In addition to Array, examples are Queue, Stack, Sequence, Set, Map, SortingMachine, ...
What's Wrong With Arrays?

• Java arrays do not “play well” with generics, as we have seen.
• This is the reason for the OSU CSE Components Array family.
• A Java array (or Array from the OSU CSE components) is not ideally suited as a data representation of the various collection types.
  – Similarly for the Java collections framework.

This is part of the package `java.util`, and includes many types similar to the OSU CSE components.
Collection Terminology

- **Fixed size** means the size/length of a collection is “inflexible”, i.e., it is determined at initialization of the collection and cannot be incrementally adjusted
  - A classical synonym is *static*; this term unfortunately means other things in Java

- **Dynamic** means the size/length of a collection is “flexible”, i.e., it can be incrementally adjusted by “adding” and “removing” entries, even from the middle
Collection Terminology

• **Direct access** means the entries of a collection (typically with a `string` model) may be accessed by providing an `int` position/index of any entry in the collection
  – A classical but unfortunate synonym is `random access`; nothing random about it!

• **Sequential access** means the entries of a collection (with a `string` model) may be accessed in increasing order of position by accessing the “next” entry in the collection
Collection Terminology

- **Direct access** means the entries of a collection (typically with a string model) may be accessed by providing an integer position/index of an entry.
  - A classical but unfortunate synonym is *random access*; nothing random about it!
- **Sequential access** means the entries of a collection (with a string model) may be accessed in increasing order of position by accessing the “next” entry in the collection.
Key Pros and Cons of Arrays

• Pros:
  – Direct access is fast, i.e., it takes constant time independent of the length of the array

• Cons:
  – Its fixed size limits array utility where dynamic size is important: it can run out of room
  – Adding and removing entries in the middle requires moving array entries, which is slow
  – Initialization may be expensive, especially if many entries are not subsequently used
Fixed Size Can Support Fast Direct Access

• A Java array (or Array from the OSU CSE components) is represented in a contiguous block of memory locations with consecutive memory addresses (IDs), so the memory address of the entry at index $i$ can be directly calculated from the memory address of the first entry, by using simple arithmetic.
Example

Client’s view of an array/Array (entries = <13, 18, 6, 21, 12, 21>):

**`Array` entries = <13, 18, 6, 21, 12, 21>**:

```
13 18 6 21 12 21
  0 1 2 3 4 5
```

*length* = 6

Implementer’s view of an array in memory:

```
... ? 13 18 6 21 12 21 ? ...
```

**base** = 45, **length** = 6
Example:

Client’s view of an array/Arrays:
<13, 18, 6, 21, 12, 21>

Implementer’s view of an array in memory:

0 1 2 3 4 5
length = 6

base = 45, length = 6

If client wants to access the entry at position 3 of the array, how does implementer compute its memory address/ID?
Example

Client’s view of an array: `[13, 18, 6, 21, 12, 21]`

Implementer’s view of an array in memory:

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>18</td>
<td>6</td>
<td>21</td>
<td>12</td>
<td>21</td>
</tr>
</tbody>
</table>
```

`length = 6`

base = 45, length = 6

Every modern computer the JVM runs on provides constant-time access to any memory location given the memory address/ID.
Notice the **Array** Mismatches

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>Queue</td>
<td>✔</td>
<td></td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>Stack</td>
<td>✔</td>
<td></td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>Sequence</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Map</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorting-Machine</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What Can Be Done?

• To represent collections that are dynamic with sequential access, a different approach is needed: not arrays
  – Note: It is an open problem to represent a Sequence, which is dynamic and offers direct access, in a way that is efficient in both execution time of access operations (i.e., constant time) and memory “footprint” (i.e., constant factor overhead)
Dynamic Can Support Fast Sequential Access

- If we want a *dynamic* collection, then we should give up on storing all entries of the collection in contiguous memory locations.

- If we want fast *sequential access*, then we should give up on fast direct access.
  
  - Instead, for every entry in the collection, wherever it is in memory, simply keep a *reference* to (i.e., memory location of) the “next” entry.
Example: Stack2

Client’s view of a Stack:
\[
\text{this} = <13, 6, 18>
\]

Implementer’s view of a Stack2:

- Top: 13
- Length: 3
- Next: 6
- Data: 18
Example:

Client’s view of a Stack:
\[ \text{this} = \langle 13, 6, 18 \rangle \]

Implementer’s view of a Stack2:

This is called a *singly-linked-list* data structure.
Example:

Client’s view of a Stack:
\[ \textit{this} = <13, 6, 18> \]

Implementer’s view of a \texttt{Stack2}:

The abstraction function (correspondence) ...
Example:

Client’s view of a Stack:

```
this = <13, 6, 18>
```

Implementer’s view of a Stack2:

The instance variables (fields) of the data representation for Stack2 are shown here.
Example: \textbf{Stack2}

Client’s view of a \textbf{Stack2}:
\[
\textit{this} = \langle 13, 6, 18 \rangle
\]

Implementer’s view of a \textbf{Stack2}:

The \textbf{Stack} methods only require access to the first entry, i.e., the top, so we keep a reference to its node.
Example:

Client’s view of a Stack:

\textit{this} = \langle 13, 6, 18 \rangle

Implementer’s view of a Stack2:

The \textit{Stack} methods include \textit{length}, so we keep this direct count of the number of nodes in the linked list.
Example:

Client’s view of a Stack:

\[ \text{this} = <13, 6, 18> \]

Implementer’s view of a Stack2:

Each of these objects (a pair of variables) is called a \textit{node} in this \textit{singly-linked-list data structure}: a variable of type \( T \), and a reference to the “next” node.
Declaration of **Node** Class

- A **Node** class is declared as a *nested class* inside the kernel implementation that uses it in a data representation (e.g., **Stack2**, **Queue2**, and similar classes)

```java
private final class Node {
    private T data;
    private Node next;
}
```
• A Node class is declared as a nested class inside the kernel implementation that uses it in a data representation (e.g., Stack2, Queue2, and similar classes)

    private final class Node {
        private T data;
        private Node next;
    }
It works because the instance variable `next` is a *reference variable*, hence is a *reference* to a `Node` object rather than a “nested” object.

```java
private final class Node {
    private T data;
    private Node next;
}
```
Example: Queue2

\[ \text{this} = <9, 21> \]
Example: Queue2

\[ \text{this} = \langle 9, 21 \rangle \]

The abstraction function (correspondence) ...
Example: **Queue2**

\[ this = \langle 9, 21 \rangle \]

Why do we have this “extra” node (we’ll call it a **smart node**)?
Example: Queue 2

\[ \text{this} = <9, 21> \]

Why do we have two references into the linked data structure for Queue, but only one for Stack?
Example: dequeue for Queue2

```java
public final T dequeue() {
    Node p = this.preFront;
    Node q = p.next;
    T result = q.data;
    this.preFront = q;
    this.length--;  
    return result;
}
```
Example: dequeue for Queue2

\( \textit{this} = \langle 9, 21 \rangle \)

The abstraction function (correspondence) ...
Example: dequeue

\( \text{this} = <9, 21> \)

```java
Node p = this.preFront;
Node q = p.next;
T result = q.data;
this.preFront = q;
this.length--; 
return result;
```
Example: dequeue

\[ \text{this} = <9, 21> \]

Node \( p = \text{this}.\text{preFront}; \)
Node \( q = p.\text{next}; \)
\( T \) result = \( q.\text{data}; \)
\( \text{this}.\text{preFront} = q; \)
\( \text{this}.\text{length}--; \)
return result;
Example: dequeue

\[ \text{this} = \langle 9, 21 \rangle \]

```java
Node p = this.preFront;
Node q = p.next;
T result = q.data;
this.preFront = q;
this.length--;
return result;
```
Example: dequeue

\[ \text{this} = \langle 9, 21 \rangle \]

```java
Node p = this.preFront;
Node q = p.next;
T result = q.data;
this.preFront = q;
this.length--; return result;
```
Example: dequeue for Queue2

\[ \text{this} = \langle 9, 21 \rangle \]

Note that this node now plays the role of the smart node!
Example: dequeue

\( \text{this} = <9, 21> \)

Node \( p = \text{this}.\text{preFront}; \)
Node \( q = p.\text{next}; \)
\( T \text{ result} = q.\text{data}; \)
\( \text{this}.\text{preFront} = q; \)
\( \text{this}.\text{length}--; \)
\( \text{return result; } \)
Example: dequeue

\textit{this} = \langle 21 \rangle

Node \( p = \text{this}.\text{preFront}; \)
Node \( q = p.\text{next}; \)
\( T \) \( \text{result} = q.\text{data}; \)
\( \text{this}.\text{preFront} = q; \)
\( \text{this}.\text{length} --; \)
\text{return} \( \text{result}; \)
Example: dequeue for Queue2

\[ \text{this} = \langle 21 \rangle \]
Example: dequeue for Queue2

\[ \text{this} = \langle 21 \rangle \]

With local variable \( p \) out of scope, this node becomes “garbage”. 
Example: dequeue for Queue2

\[\text{this} = <21>\]

The abstraction function (correspondence) ...

preFront

rear

next

length

data

next

9

data

21
Example: enqueue for Queue2

```java
public final void enqueue(T x) {
    Node p = new Node();
    Node q = this.rear;
    p.data = x;
    p.next = null;
    q.next = p;
    this.rear = p;
    this.length++;
}
```
Example: enqueue for Queue2

\[ \text{this} = \langle 21 \rangle, \ x = 74 \]

The abstraction function (correspondence) ...
Node p = new Node();
Node q = this.rear;
p.data = x;
p.next = null;
q.next = p;
this.rear = p;
this.length++;
An instance variable of a reference type in Java is always initialized to `null`; but it is a good idea to be explicit (see later line of code).
Node p = new Node();
Node q = this.rear;
p.data = x;
p.next = null;
q.next = p;
this.rear = p;
this.length++;

Example:
enqueue for Queue2
27 June 2013
OSU CSE
47

p
preFront
this
rear

length

21

next

q

next

p

next
Node p = new Node();
Node q = this.rear;
p.data = x;
p.next = null;
q.next = p;
this.rear = p;
this.length++;
Node p = new Node();
Node q = this.rear;
p.data = x;
p.next = null;
q.next = p;
this.rear = p;
this.length++;
Node p = new Node();
Node q = this.rear;
p.data = x;
p.next = null;
q.next = p;
this.rear = p;
this.length++;

queue for Queue2
length = 74
Node p = new Node();
Node q = this.rear;
p.data = x;
p.next = null;
q.next = p;
this.rear = p;
this.length++;
Node p = new Node();
Node q = this.rear;
p.data = x;
p.next = null;
q.next = p;
this.rear = p;
this.length++;
Example: *enqueue for Queue2*

\[ \text{this} = \langle 21, 74 \rangle, \ x = 74 \]

The abstraction function (correspondence) ...
The “Smart” Node

• To see why we want the extra node at the beginning of the linked list, write the code for `enqueue` without it (but be careful)
  – You should be able to see why it’s a `smart node` rather than a `dummy node` 😄

• Why is the smart node helpful in the representation of a `Queue`, but not of a `Stack`?
Resources

• Wikipedia: Linked Data Structure

• *Big Java Late Objects*, Section 15.2

• *Big Java Late Objects*, Section 16.1 (but not the part about iterators)