Part III: Data Structures
If you have been using the examples sheet on the web page, it has now been updated (and will be continually)
Data Structures

- In OOP we saw that there was an advantage in creating classes
  - Allowed us to group primitive types into one entity
  - Allowed us to group together data and the operations allowed on it

- Even without the OOP goodies (inheritance, etc) it is useful to do this for other languages
  - Rename to “data structures”
Why are they in this Course?

- Often we find that data structures provide natural support to parts of an algorithm
  - Think heapsort

- We will be looking at a range of data structures (some that you've already used) from a theoretical standpoint

- We start with a look at how to represent some fairly fundamental data structures
Abstract Data Types

- An ADT is a model of a data structure or type

- It defines the functionality expected of the data structure (but not the implementation)
  - Like a specification of the data structure
  - Maps to the interface notion in Java

- Frees us from implementation details and makes it easy to swap in new algorithms for the operations as they are discovered

- You can make your own of course, but there exists a set of ADTs that you should just know...
ADT 1: List

- A sequence of items
  - `add(item i, position p)`: insert item i into the list in position p.
  - `delete(position p)`: delete the item at p
  - `is_empty()`: returns true iff the list is empty
  - `get(position p)`: get the item at position p
List Types

- **Single**
  - Insert at front: $O(1)$
  - Insert at back: $O(n)$

- **Doubly-Linked**
  - Insert at front and back: $O(1)$
  - General insertion: $O(n)$

- **Circular**
Java List Interface

Part of Collections

List

interface

ArrayList

LinkedLIst

Vector

Array storage

Linked storage (references)
public class MyLinkedList {
    int payload;
    LinkedList next;
    LinkedList previous;
}

References
**Linked List Costs**

- **add**: Traverse the list to find the position, create object, then insert.
  - $O(n)$
  - $O(1)$
  - $O(1)$
  - $O(n)$

- **delete**: Traverse the list to find the position, then delete.
  - $O(n)$
  - $O(1)$
  - $O(n)$

- **isEmpty**: Check the head element.
  - $O(1)$

- **get**: Traverse the list to find the element.
  - $O(n)$
  - $O(n)$
- Think of these as the same thing, but \textbf{Vector} is 'synchronised' and hence slower in normal usage
  - i.e. threadsafe

\[\begin{array}{c}
  1 & 2 & 3 & 4 & 5 & 6 \\
\end{array}\]

- Memory overhead is zero
- Random access

\[\text{1st 2nd}\]
ArrayList: add to end

Increase array size.

\[ 1 2 3 4 \] \rightarrow \[ 1 2 3 4 \],

\[ 1 2 3 4 \]

Making + filling a new array
ArrayList: add to end

Most additions will just be direct \( O(1) \)

Some additions will require expansion \( O(n) \)

Add \( n \) elements in \( O(n) \) time

\( \Rightarrow \) Amortized cost \( O(1) \)
\[
\begin{pmatrix}
1 & 2 & 3 & 4 & 5
\end{pmatrix}
\]

\[
6
\]

1) There is space \( \Rightarrow \) move \( O(n-P) \) element-s

2) Create more space, then (i)
Which to use?

- Right now your default is probably to use LinkedList in java
- For a general usage, ArrayList is usually more efficient

**LinkedList** - good for insertion/deletion in a random pos.
- memory overhead + slow search

**ArrayList** - great for everything except insert in random pos.

Java array list Ξ 10 items initialize,

\[
\begin{array}{ll}
\text{200,000} & \text{LL} \\
\text{438ms} & \text{1458ms}
\end{array}
\]
ADT 2: Stack

- Analogy: A stack of plates (LIFO) - Last in first out
- Add and take from the top
  - push(item i): add i to the top
  - pop(): remove i from the top after returning it
  - top(): get the item on top
  - isEmpty(): return true iff stack is empty
Stack Implementations

- **Linked List**
  - Everything happens at the head so $O(p)$ is now $O(1)$
  - Still have a memory overhead associated with each node

- **Array**
  - Do everything at the tail of the array: good performance characteristics
  - Java's Stack uses Vector as its base
Stack Languages!

- Postscript (language for text and gfx layout on a page: printer speak)

```
7 9 add 2 mul
```

Reverse polish notation:

\[(7 + 9) \times 2\]
ADT 3: Queue

- Analogy: A queue (duh!) (FIFO)  
  - put(item i): add i to the bottom
  - get(): Take top element out of the queue and return it
  - first(): get the item on top
  - isEmpty(): return true iff queue is empty
Aside: Deque

- Sometimes useful to have a double-ended queue (Deque)
  - putFront(item i)
  - putRear(item l)
  - getFront()
  - getRear()

- In many ways Stack and Queue are just crippled versions of Deque!
Deque Implementation

- Doubly linked lists work fine \(O(1)\) for everything
- Arrays need more thought...

\[\text{Circular buffer.}\]
Java's Queue

- As with List, the Queue interface is implemented by different classes
  - LinkedList is the most basic

- You are supposed to select the implementation that best suits your needs
ADT 4: Table

- A *dictionary* or some keys mapped to values
  - `set(key k, value v)`: add the mapping pair `(k->v)`
  - `get(key k)`: return the value for key `k`
  - `delete(key k)`: remove any pair or pairs with the key `k`
  - `isEmpty()`: return true iff table is empty

<table>
<thead>
<tr>
<th>keys</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>K₁</td>
<td>V₁</td>
</tr>
<tr>
<td>K₂</td>
<td>V₂</td>
</tr>
<tr>
<td>K₃</td>
<td>V₃</td>
</tr>
</tbody>
</table>

"map"
Table Naïve Array Implementation

Keys: integers $0 \rightarrow N$

$0 \ 1 \ 2 \ 3 \ 4 \ \ldots \ N$

$\begin{array}{|c|c|c|c|c|}
\hline
V_1 & V_2 & V_3 & V_4 \\
\hline
\end{array}$

$\text{set}(i) \ \{O(1)\}$

$\text{get}(i) \ \{O(1)\}$

Space $O(\text{range})$
get() : Scan list \( O(n) \)
set() : \( O(1) \) iff duplicates
\[
\text{scan list, update or add at end} \quad \mathcal{O}(n)
\]
\( \mathcal{O}(n) \) space.
Smarter Table implementation: Array

\[
\begin{array}{|c|c|c|}
\hline
k_3, v_3 & k_1, v_1 & k_2, v_2 \\
\hline
\end{array}
\]

\[\text{get()} \rightarrow O(\log n)\]

\[\text{set()} \rightarrow O(\log n) \text{ and place shift may give expansion}\]

\[k_3 < k_1 < k_2\]

\[f(n) = f\left(\frac{n}{2}\right) + kn = O(\log n)\]