Linked Lists

An array is a list of elements with a fixed size, accessed by index. A more flexible data structure is the **linked list**.

A linked list is composed of **nodes**, each of which contains a piece of data and a reference to the next node.

- A linked list can grow and shrink as much as needed.
- Adding an element to a linked list is $\Theta(1)$.
- Finding an element in a linked list is $\Theta(n)$.

Doubly Linked Lists

A **doubly linked list** allows us to traverse the list backwards and forwards. Each node has a reference to both the next and the previous node.

Removing the last element of a singly linked list is $\Theta(n)$.

Queues

A **queue** is a data structure in with two operations:

- **add** puts an element at the end of the queue.
- **get** takes an element from the front of the queue.

We sometimes call a queue a FIFO: first-in, first-out.

We can implement a queue with an array or a linked list.

Stacks

A **stack** is a data structure with two operations:

- **push** puts an element on top of the stack.
- **pop** takes an element from the top of the stack.

Stacks are also called LIFO queues: last-in, first-out.

We can implement a stack with an array or a linked list.

Dictionaries

A **dictionary** is an abstract data structure that stores a data element with a **key** and provides two operations:

- **add** puts an element in the dictionary.
- **find** takes a key and returns the corresponding element.

Keys and elements are comparable to words and definitions: you find a definition in the dictionary by searching for the word (the key).

Dictionaries and Linked Lists

We can implement a dictionary using a linked list.

- **add** is easy. It’s an $\Theta(1)$ operations.
- We can implement **find** by scanning the list. It’s an $\Theta(n)$ operation.

We’ve seen algorithms that can search a list in $\Theta(\lg n)$ operations. Can we make a better dictionary?

Hash Tables

A **hash table** is a collection of buckets. A **hash function** maps keys to buckets. Typically, the buckets are linked lists and the buckets are held in an array; the data in the hashtable is an array of linked lists.

To add an element to a hash table, we:

- Run the hash function on the key. (Should be $\Theta(1)$)
- Index the array and retrieve the correct bucket. ($\Theta(1)$)
- Add the element to the bucket (linked list). ($\Theta(1)$)

Adding an element to the hashtable is $\Theta(1)$.
Hash Table Operations: Find

To find an element in a hash table, we:
- Run the hash function on the key. (Should be $\Theta(1)$)
- Index the array and retrieve the correct bucket. ($\Theta(1)$)
- Find the element to the bucket (linked list). (?)

If we choose the hash function and the number of buckets carefully, we can make the average length of each bucket constant with respect to the number of elements in the hash table.

Finding an element in a properly implemented hash table is $\Theta(1)$.

Java Collections

The Java standard library provides implementations of all these data structures, and many more besides.

A trip through the java.util package documentation will teach you a lot about interfaces, inheritance and the behavior of data structures.

All of the Java data structures work with elements and keys of type `Object`. This means:
- You should always cast the return value of a `get` to the correct class.
- You need to use the wrapper classes (Integer, etc.) to put primitive types into the data structures.

java.util.Hashtable

Java's implementation of a hash table is the class `Hashtable` in java.util.Hashtable (and other “hashing” classes, like HashMap and HashSet) relies on the `hashCode` method of Object:

```java
public int hashCode()
```

Every class in Java inherits `hashCode`. We can override `hashCode`, if necessary, but it's best to leave hashing to the experts. All of the built-in classes (String, Integer, etc.) have suitable hash functions.

Generic Programming

Constantly casting the results of data structure `get` methods is a pain. It's also unnecessary: we almost always want a collection of uniform type (or parent type, at least). E.g., a linked list of String, or a hash table mapping Integer keys (employee IDs) to Personnel objects.

Generic programming allows us to strongly type higher-level data structures, using syntax similar to creating an array:

```java
List<String> list = new LinkedList<String>;
```

The latest version of Java (1.5) allows you to do this.