The Java collections framework (not on exam)

Java collections framework

Many data structures implemented as classes, all found in java.util.*

- Lists: ArrayList, LinkedList
- Queues/stacks: ArrayDeque, LinkedList
- Sets/maps: HashSet, HashMap, TreeSet, TreeMap
  - also: IdentityHashMap, LinkedHashMap, LinkedHashSet, and more
- PriorityQueue

These classes implement various interfaces

- Collection<E> – a collection of values
- Iterator<E> - for iterating over all values
- List<E> - for sequential and random access

The Collection interface

Most data structures implement Collection, which provides methods for adding and removing values, checking for membership of a value etc.:

```
interface Collection<E> extends Iterable<E> {
    boolean add(E e);
    boolean addAll(Collection<E> c);
    void clear();
    boolean contains(Object o);
    boolean isEmpty();
    boolean remove(Object o);
    boolean removeAll(Collection<?> c);
    int size();
    Object[] toArray();
    ...
}
```

The List interface

`List` is used for collections where elements occur in an *order* (e.g. linked lists, dynamic arrays):

```
interface List<E> extends Collection<E> {
    // Add an element at a specific index
    boolean add(int index, E e);
    // Remove the element at a given index
    boolean remove(int index);
    // Get the element at a given index
    E get(int index);
    // Modify the element at a given index
    E set(int index, E element);
    // Return a slice of the list,
    // containing only elements from index “from”
    // to index “to”
    List<E> sublist(int from, int to);
    ...
}
```

Use of sublist

Iterating through all elements between index *from* and index *to*:

```
for(E element: list.sublist(from, to)) {
    …
}
```

Deleting all elements between index *from* and index *to*:

```
list.sublist(from, to).clear();
```

This works because `sublist` returns a *view* of the underlying structure, not a copy!

Performance

`LinkedList` and `ArrayList` implement all `List` operations, even those that have bad performance!

The following examples work even though they are not really efficient:

```
int sum(LinkedList<Integer> list) {
  int total = 0;
  // Using a linked list for random access :(
  for (int i = 0; i < list.size(); i++)
    total += list.get(i);
  return total;
}

void deleteFirst(ArrayList<Integer> list) {
  // Deleting from the front of a dynamic array :(
  list.delete(0);
}
```

The Set interface

The `Set` interface is just the same as `Collection`:

```
interface Set<E> extends
Collection<E> { }
```

This is because Collection already contains set operations: add, remove, contains etc.

The difference is: if a class implements Set, it should be guaranteed that duplicate elements are ignored

The Map interface

Important methods of Map<K, V>:

```
boolean containsKey(K key);
V get(K key);
void put(K key, V value);
V remove(K key);
Set<K> keySet();
Collection<V> valueSet();
Set<Entry<K,V>> entrySet();
// Entry<K,V> has methods
// getKey, getValue, setValue
```

For example, you can iterate over all keys of a map like so:

```
HashMap<K, V> map = …;
for (K key: map.keySet()) {
    …
}
```

Iterators

In Java, you can loop over all kinds of data structures:

```
ArrayList<E> list;
for (E x: list) { … }
```

```
LinkedList<E> list;
for (E x: list) { … }
```

```
HashMap<K, V> map;
for (K x: map.keySet()) { … }
```

That's because all these structures implement `Iterable`.

Iterator and Iterable

An `Iterator` is an object that represents a sequence of items being looped over:

```
interface Iterator<E> {
  // Return the next item in the iteration
  E next();
  // Return true if the iteration has more elements
  boolean hasNext();
}
```

Usage: each time you call `next()`, it gives you the next element in the sequence

- Once you have gone through all elements in the sequence, `hasNext()` will return false

An `Iterable` has one method for producing an iterator:

```
Iterator<E> iterator();
```

Iterator and Iterable

Suppose list is an ArrayList:

```
ArrayList<Integer> list;
```

Then the following code loops through all elements of the list:

```
Iterator<Integer> iter = list.iterator();
while (iter.hasNext()) {
  int x = iter.next();
  …
}
```

Java lets us write this as a for-loop instead!

```
for (int x: list) {
   …
}
```

ListIterator

Classes that implement List support a souped-up iterator:

- ListIterator<E> iterator();

ListIterator is an extension of Iterator that supports:

- modifying the current element during iteration
- adding and removing elements at the current position
- going back and forward in the list

So you can use this to do anything you would want to do with a linked list

Miscellaneous bits and bobs

SortedSet, NavigableSet: extra interfaces that are implemented for sets that are based on ordering

- SortedSet<E> subSet(E from, E to): return all items between from and to
- E floor(E e): return the greatest item less than e
- E pollFirst(): remove and return the smallest element
- etc.

SortedMap, NavigableMap: similar but for maps

java.util.Collections: many useful functions defined in a generic way, e.g., binary search, sorting, copying, shuffling, etc.

Java collections framework – summary

A large number of data structures, but organised into relatively few interfaces (Collection, List, etc.)

- Depending on the data structure, not all operations may be supported *efficiently* – you should be aware of which operations each class efficiently supports

Iterators are very handy, use them!

- List and Collection both extend Iterable

To find your away around a particular data structure class:

- Look at what interfaces it implements and what they are for
- Look for operations that return interesting views on the data structure

What we haven't had time for
(not on exam)

Splay trees

Splay trees are a balanced BST having *amortised* O(log n) complexity

- The main operation: *splaying* moves a given node to the root of the tree (by doing rotations)
- Insertion: use BST insertion and splay the new node
- Lookup: use BST lookup and splay the closest node
- Deletion: totally different (and much simpler) algorithm than for BSTs!

It turns out that splaying after every operation keeps the tree balanced enough

Because lookup does splaying, *frequently-used values are quicker to access!*

Complexity requires amortised complexity analysis

Amortised complexity analysis

Amortised complexity analysis: how to calculate the amortised complexity of a data structure?

- In principle: calculate the cost of any sequence of operations
- We did this for dynamic arrays but for more complex structures it gets too hard

One approach: the *banker's method*

- Imagine your data structure as a coin-operated machine – every time you put in a coin it executes one instruction
- In "normal" data structures, number of coins charged for each operation = number of instructions executed = complexity
- In the banker's method, we charge *extra coins* for the cheap operations, and *save them up for later*
- When we get to an expensive operation, we use the *saved up coins* to pay for it
- For dynamic arrays, if we charge an extra coin for each add operation, we save up enough coins to pay for the resize – we can always charge a constant number of coins per add and this means we have amortised O(1) complexity

Interesting data structures: splay trees, skew heaps

Probabilistic complexity

Some data structures rely on random choice

- Skip lists, for example
- We saw that hash tables can be modelled this way

There are more data structures like this:

- Treap: binary search tree where the shape is random
- Randomised meldable heap

Typically, you can get fast, simple data structures – but analysing the performance (i.e. telling whether it is fast) is difficult!

Real-world performance

Constant factors are important!

Perhaps the most important factor: the processor's *cache*

- It takes about 200 clock cycles for the processor to read data from memory
- But recently-accessed parts of memory are stored in the processor's *cache*, a fast memory of ~32KB, and take only ~1 clock cycle to read!

If your program accesses the same or nearby memory locations frequently (good *locality*), it will run much faster because much of the data it reads will be in the cache

- Arrays have much better locality than linked lists: their elements are stored contiguously in memory
- Accessing the elements of an array in a linear order is especially good – the processor has special circuitry to detect this, and will start *prefetching* the array, reading elements into the cache before your program asks for them
- Quicksort and mergesort have much better locality than heapsort

Real-world performance

"Latency numbers every programmer should know":

- https://dzone.com/articles/every-programmer-should-know
- L1 cache reference (~ reading one variable), 0.5ns
- Sending a packet USA → Europe → USA, 150ms

Multiply the times by a billion to get "human scale":

- Reading from L1 cache, 0.5s (one heart beat)
- Reading from main memory, 100s
- Reading from SSD, 1.7 days
- Reading from hard drive, 4 months!
- Send a packet USA → Europe → USA, 5 years!

Processors are fast, communication (with networks, with hard drives, even with RAM) is slow!

Summing up

Basic ADTs

Maps: maintain a key/value relationship

- An array is a sort of map where the keys are array indices

Sets: like a map but with only keys, no values

Queue: add to one end, remove from the other

Stack: add and remove from the same end

Deque: add and remove from either end

Priority queue: add, remove minimum

Implementing maps and sets

A hash table

- Very fast if you choose a good hash function: O(1)
- *Unordered*: can't be used to, e.g., find all values in a given range

A binary search tree

- Good performance if you can keep it balanced: O(log n)

A skip list

- Quite simple to implement
- Probabilistic performance only, typically: O(log n)

A trie or radix tree

- Tries: very simple to implement, O(w) performance
- Radix trees: a bit harder, but O(min(w, log n)) performance
- But only for certain types of input data

Implementing queues, stacks, priority queues

Stacks:

- a dynamic array
- a linked list

Queues:

- a circular array
- a linked list

Priority queues:

- a binary heap
- a leftist heap

Graphs

Very useful in modelling many problems

- A wide variety: directed, undirected, weighted, unweighted, cyclic, acyclic

But can be a bit hard to program with

- Problem: how to represent? Solution (usually): adjacency list
- Problem: want to visit nodes, but only visiting each node once. Solution: DFS
- Problem: want to visit nodes, but only visiting each node after its neighbours. Solution: topological sort
- Problem: graph is cyclic. Solution: SCCs tell you where the cycles are

Sorting

Sorted lists are a very low-tech data structure

- Create by using a sorting algorithm
- Can use binary search to find values efficiently

But they are surprisingly useful

- Sorted lists can implement *read-only* sets or maps
- Or a priority queue, if you don't need to *interleave* adds and removeMins
- You can also do range queries ("all values between x and y"), or prefix queries like in tries

What we have studied

The data structures and ADTs above

+ algorithms that work on these data structures (sorting, Dijkstra's, etc.)

+ complexity

+ data structure design (invariants, etc.)

You can apply these ideas to your *own* programs, data structures, algorithms etc.

- Using appropriate data structures to simplify your programs + make them faster
- Taking ideas from existing data structures when you need to build your own

Data structure design

How to design your own data structures?

- This takes *practice!*

Study other people's ideas:

- http://en.wikipedia.org/wiki/List_of_data_structures
- Book: Programming Pearls
- Book: The Algorithm Design Manual
- Study your favourite language's standard library

Data structure design

First, identify what operations the data structure must support

- Often there's an existing data structure you can use
- Or perhaps you can adapt an existing one?

Then decide on:

- A representation (tree, array, etc.)
- An invariant

These hopefully drive the rest of the design!

Data structure design

Finally, remember the First and Second Rules of Program Optimisation:

1. Don't do it.
2. (For experts only!): Don't do it yet.

Keep things simple!

- No point optimising your algorithms to have O(log n) complexity if it turns out $n \leq 10$
- *Profile* your program to find the bottlenecks are
- Use big-O complexity to get a handle on performance before you start implementing it
- Decide which operations need to be efficient and use that to pick out the correct data structure

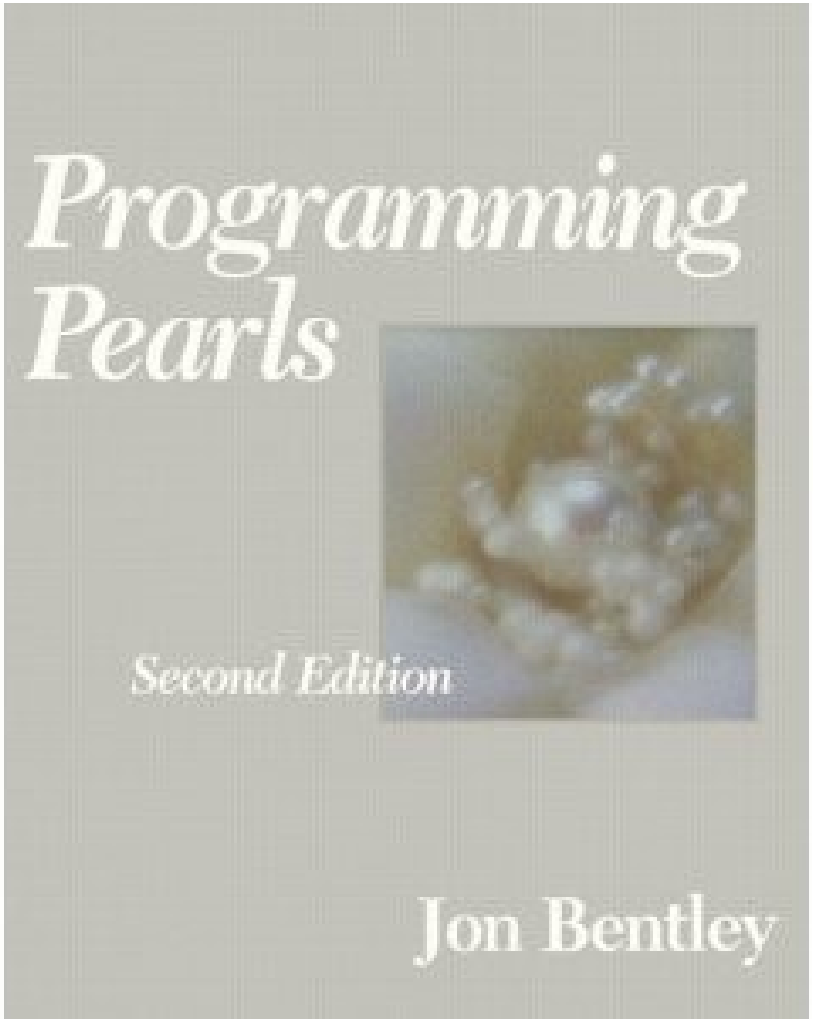
Programming Pearls

A classic computer science book – beautiful solutions to various programming problems, many of them involving data structures

A fun read, not a textbook

It helped me learn how to think like a computer scientist

A great book to read next!



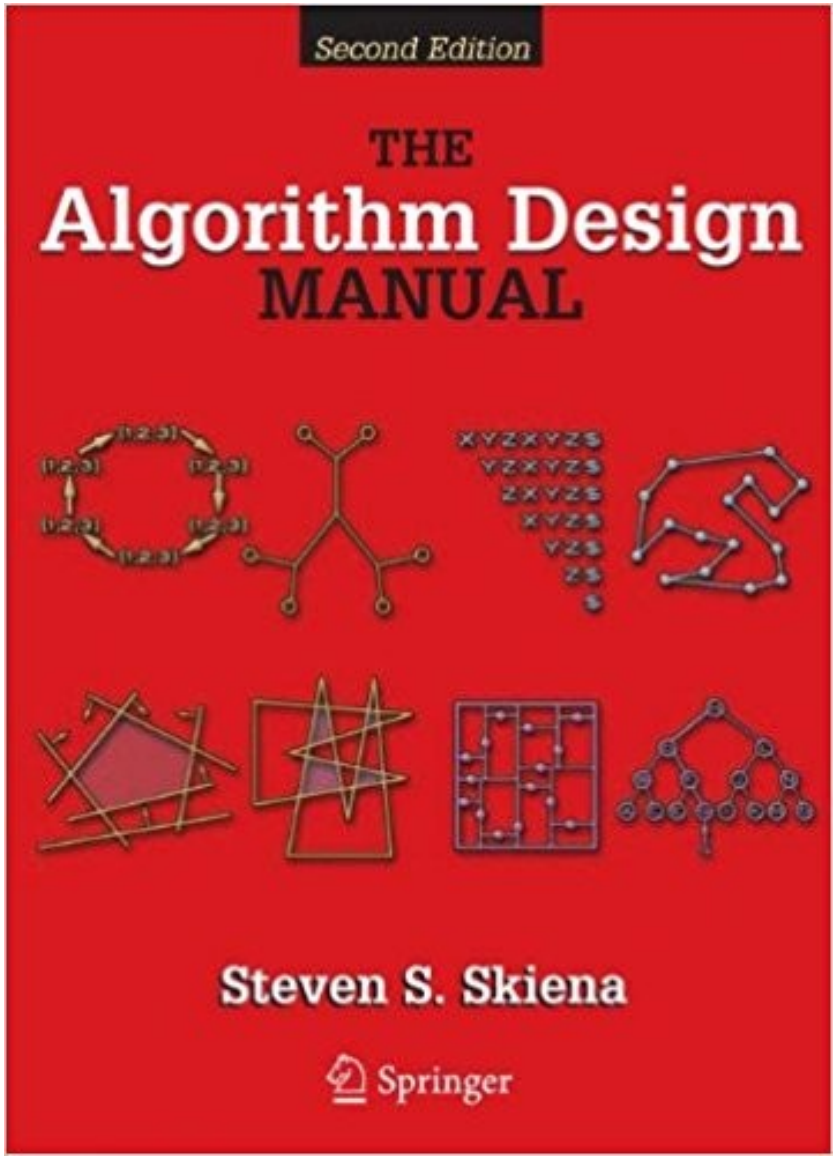
The Algorithm Design Manual

An excellent book about how to use algorithms and data structures to solve real programming problems

Lots of good techniques, and advice on when to use them

"War stories": actual problems that the author needed to solve, false starts, how they were eventually solved

E-book available from the Chalmers library!



The exam

The exam
Tuesday 15th of January,
14:00 – 18:00,
Hörsalsvägen

(re-exams April, August)

The exam

The exam is in English, and you can bring a cheatsheet, one **handwritten** A4 piece of paper

6 questions; you get a 3, 4 or 5 (or U) for *each* question

- Many questions are just right or wrong, where right = 5, wrong = U
- For some questions, you may have to e.g. answer an extra part to get a 5 on that question – this will be clearly stated

The grade you get is decided as follows:

- To get grade 3 on the exam, you must get a 3 on 3 questions
- To get grade 4 on the exam, you must get a 4 on 4 questions
- To get grade 5 on the exam, you must get a 5 on 5 questions

Best preparation: do the exercises, make sure you understand the labs, look at the old exams

- Old exams are in Swedish, but I will upload English translations of selected questions – DIT960/DIT961 exam questions are in English and useful to look at (linked from the course website)

What you need to know: the following!

Data structures

Dynamic arrays

Queue, stack and deque implementations

Binary search trees, AVL trees, 2-3 trees, AA trees

- not deletion for AVL, 2-3 or AA trees – but still for plain BSTs!
- not B-trees

Binary heaps, leftist heaps

Linked lists, skip lists

Hash tables

- Rehashing, linear probing, linear chaining
- Hash functions: the goal of a good hash function, but not how to construct them

Tries, radix trees, suffix trees

Graphs (weighted, unweighted, directed, undirected)

- Represented using adjacency lists

Algorithms

The algorithms that are used in the implementation of each data structure

Graph algorithms:

- breadth-first and depth-first search
- applications of DFS: topological sorting, checking connectedness, SCCs
- Dijkstra's and Prim's algorithms

Insertion sort, quicksort, mergesort, radix sort

- Strategies for choosing the pivot in quicksort – first element, median-of-three, randomised

Theory

Complexity and big-O notation

- For iterative and recursive functions – basically, what's in the complexity hand-in
- Also for functions that use standard data structures and algorithms

Data structure invariants

Typical kinds of question

What is the complexity of the following program?

Perform the following algorithm by hand

- e.g.: Perform Dijkstra's algorithm on this graph
- e.g. insert a value into this AA tree

Design a data structure that supports the following operations… *or:* Implement an algorithm for performing the following task

- Often: getting a 5 requires a better time complexity
- You can *usually* answer in pseudocode: don't need to spell out every last detail, but a programmer should be able to turn it into code without having to think
- You can *usually* use existing data structures and algorithms in your solutions

See old exams for examples!

Good luck!