The exam

Lab deadlines

If you've missed the final deadline for a lab, don't panic!

On June the 12th I will sit in my office (5463) from 1-3 and you can show me your lab in person

The exam
Friday 5th of June, 14:00 – 18:00,
Väg och vatten

The exam

You can bring a fusklapp, handwritten on both sides

6 questions, to pass: answer 3 questions

- There might be parts marked “for VG” - you don't need to answer those!

For a VG:

- Answer 5 questions
- If there are any parts marked “for VG”, you *do* need to answer them!

Best preparation: do the exercises, make sure you understand the labs, read the sample exam

What you need to know: the following!

Data structures

Arrays, dynamic arrays

Queue, stack and deque implementations

Binary trees, binary search trees, AVL trees, red-black trees, 2-3 trees

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

Hash tables

- Rehashing, linear probing, linear chaining –
  not how to construct a good hash function

Graphs (weighted, unweighted, directed, undirected), adjacency lists, adjacency matrices

Binary heaps, leftist heaps

Algorithms

Data structure algorithms (e.g., list insertion, BST lookup)

Graph algorithms:

- breadth-first and depth-first search
- Dijkstra's and Prim's algorithms (using a priority queue)

Selection sort, insertion sort

- In-place versions

Quicksort, mergesort

- Strategies for choosing the pivot – 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

Data structure invariants

Summing up

Basic data structures

Arrays: good for random access

- dynamic arrays: resizeable

Trees: good for hierarchical data

- special case: binary trees

Graphs: good for cyclic data

- many variants: weighted, directed, etc.

(Linked lists: good for sequential access

- Fallen out of favour a bit!)

Some data structures are special

In machine language, the memory is an array of integers

- To the processor, everything is an array

In imperative languages, the memory is an *object graph* with references being edges

- To an imperative language, everything is a graph (or an array)

In functional languages, any algebraic data type is a tree-like structure

- To a functional language, everything is a tree (or a function)

Everything else is built from whatever primitive data structures your programming language supports

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 binary search tree

- Good performance if you can keep it balanced
- Has good random *and* sequential access: the best of both worlds

A hash table

- Very fast if you choose a good hash function

Implementing queues, stacks, priority queues

Queues:

- a circular array
- a pair of lists (in a functional language)

Stacks:

- a dynamic array

Priority queues:

- a binary heap
- a leftist heap

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

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: Purely Functional Data Structures
- 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

What we haven't had time for

Amortised data structures

We briefly mentioned *amortised* complexity:

- e.g. dynamic arrays
- adding an element normally takes O(1) time
- but occasionally it can take O(n) time
- but the O(n) case happens rarely enough that *on average* adding an element takes O(1) time
- and so we say that it takes *amortised* O(1) time

Amortised complexity

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

- The tree sometimes becomes unbalanced but this happens rarely enough that the average time per operation is still O(log n)

Skew heaps are a priority queue having amortised O(log n) merge

- Similar to leftist heaps but simpler, and faster in practice!

See book chapters 21 (splay trees) and 22 (skew heaps)

Probabilistic algorithms

Sometimes it helps to make *random choices*

- Example: quicksort with a random pivot has *expected* O(n log n) complexity

Probabilistic algorithms and data structures use randomness in their implementation

- Downside: harder to analyse, small chance of poor performance (but if the probability is low enough…)

Skip lists: a nice map-like data structure with O(log n) expected complexity

Randomised splay tree: a balanced BST with O(log n) expected complexity

Functional data structures

Zippers: allow you to update functional data structures efficiently

- http://www.haskell.org/haskellwiki/Zipper

Finger trees: a sequence data type with an impressive list of features:

- O(1) access near the front and back of the sequence
- O(log n) random access
- O(log n) concatenation and splitting
- http://www.soi.city.ac.uk/~ross/papers/FingerTree.pdf
- `Data.Sequence` in GHC

Good luck!