Lab deadlines

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

On June the 6th (Monday after the exam) I will sit in my office (5469) from 1-3 and you can show me your lab in person

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 binary search tree

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

A hash table

- Very fast if you choose a good hash function: O(1)

A skip list

- Quite simple to implement
- Only appropriate for imperative languages
- Probabilistic performance only, typically: O(log n)

Implementing queues, stacks, priority queues

Queues:

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

Stacks:

- a dynamic array (in an imperative language)
- a list (in a functional language)

Priority queues:

- a binary heap (in an imperative language)
- a skew heap (in a functional language)

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 (invariant, 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: Purely Functional Data Structures
- Book: Algorithms by Sedgewick
- 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 complexity analysis

Dynamic arrays, hash tables, skew heaps etc. have *amortised* complexity

- e.g. skew heaps: O(log n) amortised complexity means a sequence of n operations takes O(n log n) time

We analysed the complexity of dynamic arrays, but for e.g. skew heaps it's too hard

Amortised analysis gives us cute tools to calculate amortised complexity

- e.g. the *banker's method*: there is a "piggy bank" containing a number of units of time
- operations can choose to put time into the piggy bank; this time is counted as part of their running time
- expensive operations can empty the piggy bank and use the time in it, which is subtracted from their actual running time
- by saving and taking out exactly the right amount of time from the piggy bank, you can analyse the operations as if they had non-amortised complexity

Splay trees

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

- The main operation: *splaying* uses rotations to move a given node to the root of the tree
- Insertion: use BST insertion and splay the new node
- Deletion: use BST deletion and splay the parent of the deleted node
- Lookup: use BST lookup and splay the closest node

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

Too hard to analyse complexity for this course (amortised)

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
- Based on 2-3 trees, but with “fingers” that give you constant-time access to the leftmost and rightmost nodes of the tree
- http://www.soi.city.ac.uk/~ross/papers/FingerTree.pdf
- `Data.Sequence` in GHC

The limits of sorting

All the sorting algorithms we've seen take at least O(n log n) time. Is there a reason for that?

Yes! It turns out any sorting algorithm based on *comparing* list elements (e.g. a[i] < a[j]) must take O(n log n) time, and there's a lovely proof

- See http://www.bowdoin.edu/~ltoma/teaching/cs231/fall07/Lectures/sortLB.pdf
- or http://www.cse.chalmers.se/edu/course/DIT960/slides/limits-of-sorting.pdf

Sorting algorithms *not* based on comparisons can break this barrier

- e.g., sorting algorithms for integers can get to O(n) time by using modular arithmetic (see *radix sort*)

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
- The cache is a fast area of memory built into the processor, which stores the values of recently-accessed memory locations
- It's ~32KB, and reading data stored in it takes ~1 clock cycle

If your program accesses the same or nearby memory locations frequently (good *locality*), it will run faster because the things it reads are more likely to be in the cache

- The elements of an array are stored contiguously in memory – this makes it much better for locality than e.g. a linked list
- Accessing the elements of an array in increasing or decreasing 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 to read them
- This is one reason why quicksort is quick!

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":

- L1 cache reference, 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!

The exam

The exam

Monday 30th of May, 14:00 – 18:00,

Hörsalsvägen

The exam

You can bring a fusklapp, one A4 piece of paper

6 questions

On each question, you can get a G or a VG (or nothing)

To get a G on the exam, get a G on 3 questions

To get a VG on the exam, get a VG on 5 questions

- Normally: answer right = VG, answer slightly wrong = G
- But some questions have extra parts for VG!

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

What you need to know: the following!

Data structures

Dynamic arrays

Queue, stack and deque implementations

- Queues using a circular array (imperative) or pair of lists (functional)

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

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

Binary heaps, skew 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

Graphs (weighted, unweighted, directed, undirected)

- Represented using adjacency lists

Algorithms

The algorithms that are part 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

- 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

Good luck!