CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Data structures

Summary and exam

Dr. Alex Gerdes

DIT961 - VT 2018

Summing up

Basic data structures

- Arrays: good for random access
  - dynamic arrays: resizeable
- Linked lists: good for sequential access
  - many variants – doubly linked, etc.
- Trees: good for hierarchical data
  - special case: binary trees
- Graphs: good for cyclic data
  - many variants: weighted, directed, etc.

Basic ADTs

- 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
- 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

Implementing stacks and (priority) queues

- Queues:
  - a linked list
  - a circular array
  - a pair of lists (in a functional language)

- Stacks:
  - a dynamic array or linked list
  - a list (in a functional language)

- Priority queues:
  - a binary heap
  - a leftist or skew heap (in a functional language)

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)

Standard libraries

- Java:
  - List, ArrayList, LinkedList
  - Stack (deprecated use LinkedList)
  - Deque, LinkedList, ArrayDeque
  - Set, HashSet, TreeSet
  - Map, HashMap, TreeMap
  - PriorityQueue
  - Comparator, Comparable, equals and hashCode in Object
  - Iterator
- Haskell:
  - Data.Map, Data.Set, Data.Queue, Data.PriorityQueue
  - Eq, Ord

Amortised data structures

- We 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

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
  - 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, 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!
  - First check that your idea works and that the implementation is *correct:* test!
  - 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 didn't have time for

- *Splay trees* are a balanced BST having amortised $O(\log n)$ complexity
- *Skip lists*: a nice map-like data structure with $O(\log n)$ expected complexity
- *Bucket sort*: have a bucket for each equality class and put each element in right bucket
- *Radix sort*: for integers do a bucket sort for each digit
- *Prefix trees*: for strings, each node has a child for each possible first character of the rest of the string

Exam

Exam

- Signing up is mandatory
- Next week Thursday 8:30 - 12:30 in SB-MU
- Language:
  - Written in English
  - Solutions should either be in English or Swedish (or Dutch ;-)
- Allowed aids:
  - English dictionary
  - Fusklapp: one hand-written sheet of A4 (both sides)

Exam format

- 6 questions,
  - no points but a U, G or VG per question
  - G: at least three questions with a G or VG
  - VG: at least five questions with a VG
- Betygsgränser
  - VG: 100% correct
  - G: correct, may contain (very) minor mistake
  - U: if you make a (big) mistake
- Some questions have extra requirements for a VG
  - For example lower complexity
  - ’For a VG only’ parts: more work, sometimes harder

What you need to know?

Data structures

- Arrays, dynamic arrays
- Linked lists (single-linked, doubly-linked)
- Queue and stack implementations using arrays or linked lists
- Binary trees, binary search trees, AVL trees, AA 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, skew and leftist heaps

Algorithms

- Data structure algorithms (e.g., list insertion, BST lookup)
- Binary search
- Tree traversal: in-order, pre-order, post- order
- Graph algorithms:
  - breadth-first and depth-first search
  - Dijkstra's and Prim's algorithms (using a priority queue)

Sorting algorithms

- Bubblesort, 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

Good luck!