Miscellaneous

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 and for skew heaps.

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

Standard libraries

- Java:
  - *List*, ArrayList, LinkedList
  - Stack
  - *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

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

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

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

A couple of things we didn’t have time for

Sorting and trees

- Sorting which doesn't compare elements pairwise:
  - 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. All words which begin with a certain prefix can be found in one sub tree.

Splay trees

Splay trees are a self-balancing 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)

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.

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!
Simple algorithms tend to have worse complexity but better constant factors. Keep in mind that for small data the simple algorithm can be fastest.

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!