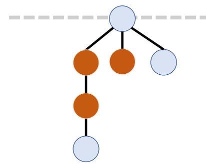
Example sheet week 7/8

Data structures. Amortized analysis.
Algorithms—DJW—2016/2017

Questions labelled FS are from Dr Stajano's list of exercises. Questions labelled $*$ involve more coding or more thinking and are best tackled after the other questions, if you have time.

Question 1 (FS49). In the Fibonacci heap, suppose that `decreasekey()` only cuts out (if necessary) the node whose key has been decreased, i.e. it doesn't use the `loser` flag and it doesn't recursively inspect the node's parents. Show that it would be possible for a node with n descendants to have more than $O(\log n)$ children.

Question 2. Give a sequence of operations that would result in a Fibonacci heap of this shape. (The three darker nodes are losers.) What is the shortest sequence of operations you can find?



Question 3*. In a Fibonacci heap, can a node x aquire a child node y, then lose it, then gain it again?

Question 4. Prove the corollary in Section 6.3 of the handout, namely, that in a Fibonacci heap with n items the maximum degree is $O(\log n)$.

Question 5. In the flat forest implementation of DisjointSet, using the weighted union heuristic, prove the following:

(i) After an item has had its 'parent' pointer updated k times, it belongs to a set of size $\geq 2^k$.
(ii) If the DisjointSet has n items, each item has had its 'parent' pointer updated $O(\log n)$ times.
(iii) Starting with an empty DisjointSet, any sequence of m operations of which n are `add_singleton()` takes $O(m + n \log n)$ time in aggregate.

Question 6*. Sketch out how you might implement the lazy forest DisjointSet, so as to efficiently support "Given an item, print out all the other items in the same set".

Question 7. In Tick 1, you implemented a heap using an array that expands as necessary to hold the data. You were invited to implement it by doubling the size of the array whenever it fills up. Prove that the amortized cost of adding an item to the array is $O(1)$.

Question 8. Consider a stack that, in addition to `push()` and `pop()`, supports `flush()` which pops all items from the stack. Explain why the amortized cost of each of these operations is $O(1)$.

Question 9*. Consider a k-bit binary counter. This supports a single operation, `inc()`, which adds 1. When it overflows i.e. when the bits are all 1 and `inc()` is called, the bits all reset to 0. The bits are stored in an array, $A[0]$ for the least significant bit, $A[k-1]$ for the most significant.

(i) Give pseudocode for `inc()`.
(ii) Explain why the worst-case cost of `inc()` is $O(k)$.
(iii) Starting with the counter at 0, what is the aggregate cost of n calls to `inc()`?

(iv) Let $\Phi(A)$ be the number of 1s in A. Use this potential function to calculate the amortized cost of `inc()`.

Question 10*. In a binomial heap with n items, show that the amortized cost of `push()` is $O(1)$ and the amortized cost of `popmin()` is $O(\log n)$. *[Hint: use your answer to Question 9].*

Question 11*. Consider an undirected graph with n vertices, where the edges can be coloured blue or white, and which starts with no edges. The graph can be modified using these operations:

- `insert_white_edge(u,v)` inserts a white edge between vertices u and v
- `colour_edges_of(v)` colours blue all the white edges that touch v
- `colour_edge(u,v)` colours the edge u–v blue
- `is_blue(u,v)` returns True if and only if the edge u–v is blue

Give an efficient algorithm that supports these operations, and analyse its amortized cost.

Extend your algorithm to also support the following operation, and analyse its amortized cost:

- `are_blue_connected(u,v)` returns True if and only if u and v are connected by a blue path

Extend your algorithm to also support the following operation, and analyse its amortized cost.

- `remove_blue_from_component(v)` deletes all blue edges between pairs of nodes in the blue-connected component containing v