Final Exam, CS520, Fall '09, McConnell

1. Write a Java or C++ method for RANDOMIZED-SELECT. Assume that you have available an implemention of RANDOMIZED-PARTITION that you can call.

   Solution: *See page 186 for the equivalent pseudocode.*

2. One way to build a leftist heap is to insert the elements one at a time. This takes $\Theta(n \log n)$. Get a better bound by taking a divide-and-conquer approach. Be sure to state your recurrence before analyzing the running time.

   Solution: *As a base case of one element, create a leftist heap node. Otherwise, recursively construct a leftist heap on half the elements, recursively construct one on the other half, and then merge them.* $T(n) = 2T(n/2) + \log n = O(n)$ *by the master theorem.*

3. Illustrate our reduction of 3-CNF-SAT to CLIQUE on the following example: $(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x3) \wedge (x_1 \vee x_2 \vee x_3)$

   Solution: *See Figure 34.14 on page 1005.*

4. Amortized bound for the splay operation.

   (a) State the credit invariant.

   Solution: *Each node y has* $\lfloor log_2 size(y) \rfloor$ *credits.*

   (b) How many credits in our pocket do we start the splay operation with?

   Solution: $\lfloor 3 \log_2 n \rfloor$ *credits. Tarjan had one greater to cover a possible odd rotation at the end, but we ignored this because its cost is subsumed by other costs.*

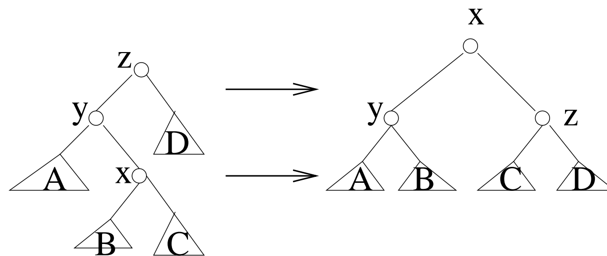
   (c) What is the rule about when we can spend some of them and how many we can spend?

   Solution: *Every time the splayed node's credit requirement increases by one, we can spend three credits.*

   (d) Why is it that if we stick to this rule, we won't go broke before reaching the top?

   Solution: *The splayed node's credit requirement can increase by at most* $\lfloor \log_2 n \rfloor$*, and three times this is what we start out with.*

   (e) Explain how the following rotation is paid for in the case when x's credit requirement doesn't go up. Your analysis should explain the relation between x, y, and z's initial credit requirements. It should then tell what relation these have to their final credit requirements.



Solution: *Since z's initial size is x's final size, and y's initial size is at least x's, x, y, and z, each have the same credit requirement initially. Since x's final size is the same as z's initial size, this is the same as x's final credit requirement. Afterward, one of y and z has less than half of x's size, so its credit requirement is less than x's. Its credit requirement went down, so one of its credits can be used to pay for the rotation without violating the credit invariant.*

(f) What data structure could you use to implement the credits in a program without altering the asymptotic time bound?

Solution: *The credits are for the analysis only. Implementing them in the program would be pointless.*

5. Recall that when we did the analysis of the splay operation using the splay credits, I asked if the analysis were complete, and everybody agreed that it was. I surprised you by saying that I didn't agree. I pointed out that, in addition to showing how to maintain the credit invariant during a splay, we also had to maintain it after a concatenation of two paths if we were to claim that this operation also takes $O(\log n)$, amortized. I pointed out that overlooked cases is the most common cause of bogus proofs.

   Are you convinced of the $O(\log^2 n)$ amortized bound for linking/cutting tree operations? **You shouldn't be. The most important operations were the linking and cutting operations, but we neglected to do an amortized analysis on them.** I wondered if someone would call me on this, or if I would get away with it. (I got away with it.)

   (a) (Review) State the credit invariant we used to bound the number of splices during an expose. Our analysis mentioned piglets stealing the solid edge from a sibling, and used credits that can each pay for one splice (theft).

   Solution: *If the solid edge is held by a runt, the solid edge has a credit on it.*

   (b) (Review) Show that you can maintain the invariant during an expose, where you start out with a pocket full of $2\lfloor \log_2 n \rfloor$ credits.

   Solution: *Every time a runt steals the solid edge, you pay for the theft with a credit out of pocket and deposit an additional credit on the edge to maintain the credit invariant. Since a runt is at most half the size of its mom, this can happen at most $\lfloor \log_2 n \rfloor$ times. If a non-runt steals the solid edge, it steals it from a runt sibling, and the credit is freed up to pay for the theft.*

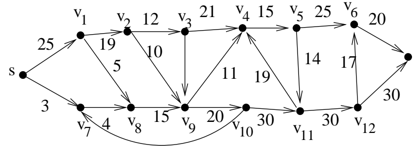
(c) Show how to maintain the credit invariant when we link two trees while using $O(\log n)$ out-of-pocket credits for the operation. Assume that if we make node u of T_1 the parent of root r of T_2, we first perform an expose on u.

Solution: *This can turn some ancestors of u from runts into non-runts, but not the other way around. This doesn't cause them to need more credits. It can turn some siblings into runts, but since the initial expose made sure that they don't have the solid edge, they don't need any new credits.*

(d) Show how to maintain the credit invariant when we cut a tree into two trees. Assume that if we cut between a node r and its parent u, we first perform an expose at u. We've already proven the amortized bound for the cost of this expose, so you should ignore it in the analysis.

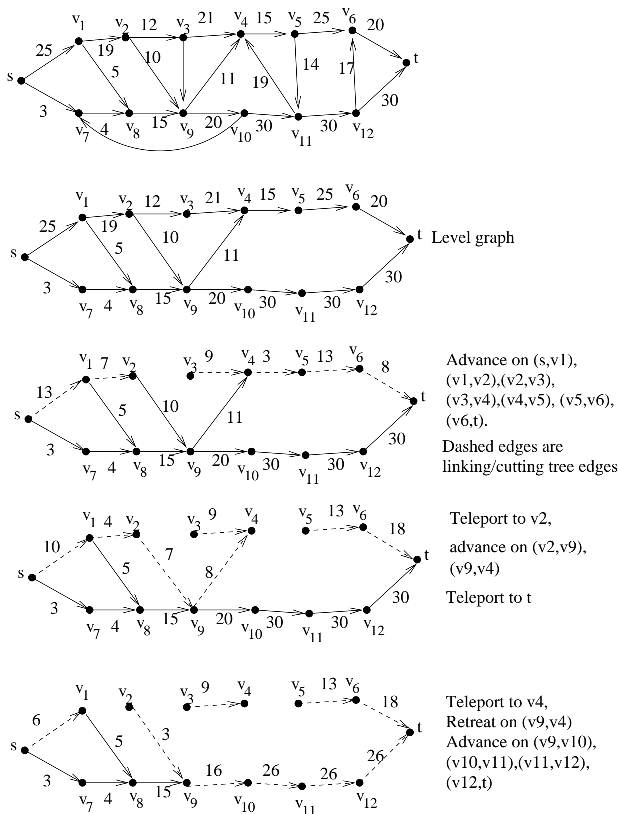
Solution: *This can turn some ancestors of u into runts. They each need a new credit. However, since any node has at most $\lfloor \log_2 n \rfloor$ runt ancestors, the cut costs us $\lfloor \log_2 n \rfloor$ credits out of pocket.*

6. The following depicts a graph on which we plan to use Sleator-Tarjan's approach to max flow using linking/cutting trees.



Sleator-Tarjan leave some decisions open, such as which vertex to advance to next when you're at the root of a linking/cutting tree. To make it possible for me to grade efficiently, whenever you have such a choice, advance to the choice with the smallest subscript.

Show the state of the level graph after each of the first three augmentations. After each, tell which edges you advanced on, which you retreated on, and for each "teleport" you took advantage of using a linking/cutting tree, tell the starting and ending vertex of the teleport.



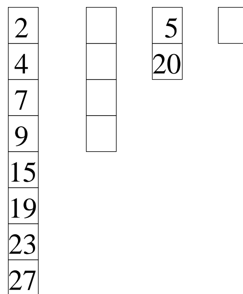
7. Show that the cost of exhausting the level graph is $O(m \log n)$. You may use the fact that linking/cutting tree operations take $O(\log n)$, amortized. **Hint:** *First address the costs incurred in advancing and retreating on edges. Then address the costs incurred in teleportations.*

   Solution: *Each advance inserts an edge into the tree at an amortized cost of $O(\log n)$. It is not removed until a retreat on it or saturation of it. Each retreat or saturation removes it from tree in $O(\log n)$, amortized. It removes it also from the graph, so each edge is inserted and removed at most once, incurring $O(\log n)$, for a total of $O(m \log n)$. Each teleportation is followed by an advance, a retreat, or a saturation, and costs $O(\log n)$, so the amortized cost of the teleportations is bounded by the amortized cost of advances, retreats, and saturations, and must therefore be $O(m \log n)$.*

8. Show that, given a set of elements where each contains a unique binary search key and a unique heap key, a pointer to the next and previous element in sorted order of binary search key and a pointer to the next and previous element in sorted order of heap key, you can construct their treap in $O(n)$ time.

   Solution: *The element x with the highest heap key must be a leaf, and one of its neighbors in binary search tree order must be its parent. Remove x and recursively build the treap on the remaining element. The nodes don't move in memory, they just form tree links. You still know where the parents are. Figure out which of the parents you can make x a child of to create a larger treap, in $O(1)$ time. The time is $T(n) = T(n-1) + 1 = O(n)$.*

9. Here is an example of a way to maintain a set S of integers:



It consists of a set of arrays, each of which has a size that is a power of two, each of which is either empty or fill, and each of which is sorted if it is full. It supports the following operations in $o(n^\epsilon)$ **amortized** time for any $\epsilon > 0$.

- **Lookup:** Given an integer i, determine whether it occurs in S;
- **Insert:** Given an integer i, insert it to S if it is not already in S;

When you insert an element, you put it in the rightmost array, if that array is empty. Otherwise, if the next array over is empty, you put the two elements in that array in sorted order. Otherwise, you put them in sorted order in an auxiliary array of size two, and perform the `merge` operation of mergesort with the next array to get them into a sorted array of size 4, which can either be stored in the third array from the right or merged with it to get a sorted array of size eight, etc.

(a) Derive a tight worst-case bound for `Lookup`.

Solution: *Do binary search in each array. Each array has at most n elements, and there are $O(\log n)$ of them, so the cost is $O(\log^2 n)$. This bound is tight, because the largest half of the arrays each take $\Omega(\log_2 n)$ time.*

(b) Derive a tight worst-case bound for `Insert`.

Solution: *If merge takes at most cn operations, then the worst case is $c\sum_{i=0}^{\log_2 n} 2^i$, which is $O(n)$, since this is a geometric series with largest term $O(n)$. The bound is tight because a merge on the largest array involves all n elements.*

(c) We can get a better amortized bound for `Insert`. Suppose we number the arrays $(0, 1, 2, \ldots$ from the right. We maintain the invariant that if an element is in array i, it has $\lfloor \log_2 n \rfloor + 1 - i$ credits on it, where n is the largest number of elements the data structure ever has. ($\lfloor \log_2 n \rfloor + 1$ is the number of bits in n.) Each credit pays for $O(1)$ operations.

Derive an $o(n^\epsilon)$ amortized bound for insertion. You should briefly explain why the credits suffice to cover all the costs.

Solution: *When an element is involved in a merge, it moves left, freeing up a credit. One credit is freed for each element involved in a merge, and, since the merge takes time proportional to the number of these elements,*

it is paid for by the credits. The initial credit requirement on the inserted element is $O(\log_2 n)$, so the amortized cost of `Insert` *is $O(\log_2 n)$.*

10. We saw how to use linking/cutting trees to find the least common ancestor of two vertices in a tree in $O(\log n)$ time, amortized. This was an example of an *on-line algorithm*, where we don't know the queries in advance.

    The *off-line least-common ancestors problem* gives a tree and a set of pairs of pointers to vertices. For each pair, we must find the least common ancestor. This differs from the on-line version in that we know the queries in advance.

    Theorem: For m queries on a tree of size n, it takes $O(n+m)$ time to find the m least-common ancestors.

    (This is a result by Harel and Tarjan, using a union-find data structure that's due to Gabow and Tarjan.)

    Here's a new problem. Given an array of n integers and a set of m intervals on the array, find the minimum integer in each interval. Each interval is represented with the starting and ending index of the interval.

    Show that this problem can be solved in $O(n+m)$ time.

    Solution *Reduce it to the least common ancestors problem, as follows. Construct a treap out of the array elements, where the index of the element is the binary search key and the value of the element is the heap key. For interval $[i, j]$, the minimum value in the interval is stored at the least common ancestor of the nodes corresponding to positions i and j.*