Homework 5 Solutions
Fundamental Algorithms, Fall 2005, Professor Yap

Due: Wed Dec. 14, in class.
SOLUTION PREPARED BY Instructor and T.A.s

INSTRUCTIONS:

- Please try to give clear but brief answers. Usually, the length of your answer will go down in proportion to the amount of thought given to the problem.

Question 1 **(10 Points)**

Our "Counter Example" analysis can yield the exact cost to increment from any initial counter value to any final counter value. What is the *exact* number of work units to increment a counter from 71 to 254?

SOLUTION: From the lecture notes we know that for the "Counter Example" the total charge is equal to the sum of the total cost and the change in potential. The total charge is 366; each increment has charge 2 and we need to increment $254 - 71 = 183$ times. The initial potential is 4, since the binary representation of 71 is 1000111, and the final potential is 7, since the binary represenation of 254 is 11111110; remember that the potential of the counter is defined to be the number of one's in the counter. Thus the total change in potential is 3 and hence the total cost is 363.

Question 2 **(20 Points)**

Consider the insertion of the following sequence of keys into an initially empty tree: $-1, 1, -2, 2, -3, 3, \ldots, -n, n$. Let T_n be the final splay tree.
(a) Show T_n for $n = 1, 2, 3$.
(b) Make a conjecture about the shape of T_n, and prove it.

SOLUTION: (b) The conjecture on the shape is shown in the Figure 1 (a).

The proof is by induction on n. The base case $n = 1$ is obvious. Suppose the conjecture holds for T_{n-1}. Now we insert $-n$ and then n; these steps and the resulting tree are shown in Figure 1 (b).

Question 3 **(40 Points)**

(a) Carry out the following operations on a Fibonacci heap, starting from an initially empty structure. Show the heap after each operation. Please indicate any marked nodes with an "X".
Insert(a,10) i.e., insert item a with key 10
Insert(b,20)
Insert(c,30)
Insert(d,40)
Insert(e,15)
Insert(f,50)
Insert(g,35)
Insert(h,45)
deleteMin()
decreaseKey(c,14) i.e., decrease the key of c to 14
deleteMin().

(b) We want to do a variation of Fibonacci heaps. The idea is to combine the root list H and the array A used in the consolidation process. In this way, we avoid the consolidation process altogether. First, all sibling lists must be ordered by degree, and furthermore, there is at most one child of each degree. Second, the rootlist H is treated like a sibling list (of some imaginary superroot). This implies that the rootlist and sibling lists have length at most $D(n) = O(\lg n)$. Third, each sibling list (including rootlist) has a potential that is equal to the number of nodes in the sibling list, plus the maximum degree among the siblings. The

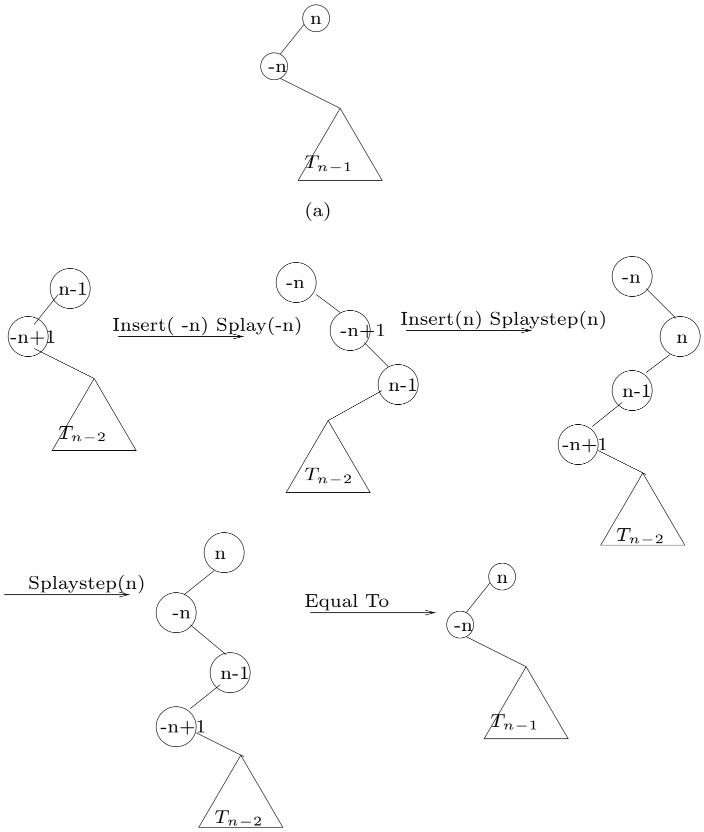

Figure 1: (a)The tree T_n. (b) Inserting $-n$ and n in T_{n-1}

potential of the data structure $\Phi(H)$ is the sum of the potentials of all sibling lists, including the rootlist, plus twice the number of marked nodes. Marking is as before.

Show that we can perform insert, union and decreaseKey in amortized constant time.

SOLUTION: (a) The root list of the heap after the insertions is a set of trees where the root of each tree has degree zero; the roots of these trees contain the key values $10, 20, 30, 40, 15, 50, 35, 45$ corresponding to the items a to h.

The deleteMin() operation deletes 'a' from the heap. The resulting heap after doing decreaseKey(c,14) and deleteMin() is shown in Figure 2.

(b) As was told in the class the above potential function does not work. The reason is that the change in potential is always non-negative and thus insufficient to pay for the cost which in the worst case can be $O(\lg n)$.

Let $t(H)$ denote the number of tress in the root list; $H.len$ be one plus the maximum degree among all the trees in root list; and $m(H)$ the total number of marked nodes in the whole heap. The potential function is then defined as $\Phi(H) = t(H) + H.len + 2m(H)$.

An equivalent way to treat the root lists is to think of them as binary numbers of bit length $H.len$, where the i'th bit is one iff there is a tree with degree i in the root list; let $B(H)$ represent this binary number. It is not hard to see that under this interpretation union of two heaps H and H' is just adding the two binary numbers $B(H)$ and $B(H')$. In this interpretation, the potential $\Phi(H)$ means that we assign two credits to each one and one credit to each zero in $B(H)$. We will show that the amortized cost of adding two binary numbers is a constant and hence the amortized cost of `union(H, H')` is constant.

Consider the following special cases:

- Adding a one, with credit 2, to a zero with credit 1. Using the credit of zero we can pay for the cost of addition and place the remaining 2 credits on the output bit.

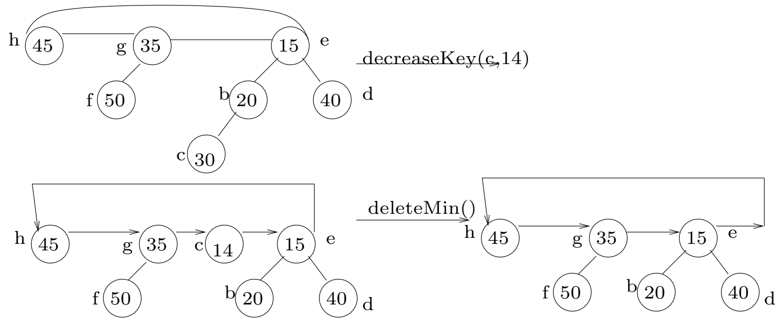

Figure 2: The heap after doing insertions, deleteMin(), decreaseKey, and another deleteMin()

- Adding a one, with credit 2, to another one with credit 2. In this case we use one credit to pay for the cost of addition and break the remaining 3 into one credit on the zero and 2 credits on the one.

Thus our potential scheme always ensures the above invariants while adding the two binary numbers; if a carry bit is produced it always has 2 credits and from the above two cases we know that this is sufficient to pay for the cost and maintain the credits. Thus we have enough credit to pay for the change in potential and the cost and hence the amortized charge is a constant.

An alternative argument is the following: Let $B(H)$ be longer than $B(H')$. Suppose l' is the largest bit in $B(H)$, beyond the largest bit of $B(H')$, that is modified; thus $0 \leq l' \leq H.len - H'.len$. Then the change in potential is less than $3 - l' - H'.len$; we must have converted $l' - 1$ ones in $B(H)$ that lie to the left of the largest bit of $B(H')$ to zeros, converted a zero to one, and possibly increased the length of $B(H)$ by one, however, at the same time we have destroyed the counter H' and thus lost the potential corresponding to its length. The cost is $l' + H'.len$. Thus the amortized cost is $3 = O(1)$.

For decreaseKey the amortized cost analysis is similar to the one in the lecture notes; in this case the release in potential is due to the loss of marked nodes while doing the cascading cut. The analysis holds if we know the exact place in the root list where to insert the tree produced by cutting. For this purpose we introduce a degree register D associated with the root list. From now on we assume that all our linked lists are doublly linked lists.

Let $T_1, \ldots, T_k$ be the trees in the root list with degrees $d_1 < d_2 < \cdots < d_k$. The degree register is a list of pair of pointers where the length of the list is $d_k + 1$. Let $D[i]$, $i = 0, 1, 2, \ldots, d_k$, represent the i'th pair in the register and $D[i, j]$, $j = 1, 2$ the j'the element in the pair $D[i]$. The interpretation of these two pointers is the following.

- $D[i, 1]$ points to the tree T_j if $i = d_j$; if $d_j < i < d_{j+1}$ then it points to T_j; and is null if $i = 0$ and there is not tree with degree 0.
- $D[i, 2]$ points to a list of all trees in the heap whose root have degree i.

For each node x with degree i in the heap there are three additional pointers:

- $x.degPtr$ that points to the entry $D[i]$;
- $x.degPtrS$ that points to its successor in the list $D[i, 2]$;
- $x.degPtrP$ that points to its predecessor in the list $D[i, 2]$.

Now we describe the details of all the operations in this augmented data structure.

`makeQueue()`: create two empty lists: first, the root-list and second the degree register. `insert(x, H)` is a special case of union. `union`(H_1, H_2): Let D_i be the degree register corresponding to H_i and $l_i := H_i.len - 1$ with the assumption $l_1 \geq l_2$. We give a rough outline of the algorithm.

- Let $i = 0$, $prev = NULL$, and $T = NULL$

- While $i \neq l_1$
  Concatenate the two lists $D_1[i, 2]$ and $D_2[i, 2]$.
  Merge the trees at $D_1[i, 1]$, $D_2[i, 1]$ and T, the tree of degree i from the previous merge. The output, if any, is stored in $D_1[i, 1]$ and let T be the tree that is "carried" over to the next merge. If there is no merging to be done then we update the $D_1[i, 1]$ to point to $prev$; this is needed to maintain the first property for $D_1[i]$. Let $prev = D_1[i, 1]$.

Intuitively, we are doing "binary addition" of the lists D_1 and D_2 and T represents the carries.

`decreaseKey(x, k, H)`: we cut x if it is not a root; if x is the child of the root and k is less than the value at the root then we switch x with its parent; otherwise, we just decrement the key at x to k. We can insert the trees, produced during the cascading cut, in the root-list using the $x.degPtr$. Moreover, if the parent of x has degree i we have to move it from the list $D[i, 2]$ to the list $D[i - 1, 2]$, but this can be done using $x.degPtr$ and the two additional pointers $x.degPtrS$ and $x.degPtrP$ in constant time.

The `deleteMin()` operation can be implemented using the above operations.

Question 4 (30 Points)

Consider the LCS problem where we are constructing the matrix L corresponding to input strings X, Y.

(a) Suppose $L[i, j] = 89$. What are all the possible values of $L[i - 1, j - 1]$? What are all the possible values of $L[i, j + 1]$? You must prove your claims.

(b) Consider the string $X_n = 010^2 10^3 1 \cdots 0^{n-1} 10^n 1$ and $Y_n = 101^2 01^3 0 \cdots 1^{n-1} 01^n 0$. Compute $L(X_4, Y_4)$. Next, prove that $L(X_n, Y_n) = 2n - 1$.

(c) You can represent $LCS(X, Y)$ by a suitable digraph whose size is $O(mn)$ where $|X| = m, |Y| = n$. This graph can be constructed from the matrix L used to solve the LCS problem. Please describe an algorithm to construct this graph.

(d) Instead of $O(mn)$, give a sharper bound on the space used by this graph. HINT: express the bound in terms of $L(X, Y) = k$. pp (e) Given a string Z, how fast can you check whether $Z \in LCS(X, Y)$ using this graph G?

SOLUTION:

(a) $L[i - 1, j - 1]$ can only be 89 or 88. It must be 89 if $x_i \neq y_j$. $L[i, j + 1]$ can be $89, 88$ or 90.

(b) The string $(01)^{n-1}0$, or the string $(10)^{n-1}1$, which has length $2n-1$ is a common subsequence of both X_n and Y_n; thus $L(X_n, Y_n) \geq 2n - 1$. Also observe that there are n 1's in X_n and the same number of 0's in Y_n. Since the last bit of X_n and Y_n do not match only one of them can be present in $LCS(X_n, Y_n)$, i.e., either the last 1 of X_n or the last 0 of Y_n gets dropped. Thus $L(X_n, Y_n) \leq 2n-1$, giving us the desired equality.

(c) We will construct a DAG $G_{m,n}$ corresponding to the matrix L recursively. More precisely, we will construct the adjacency list representation of the $G_{m,n}$. With each index (i, j) we will associate a graph $G_{i,j}$, though we will not construct it, and using these graphs we will construct the final graph. Each edge in $G_{i,j}$ will be either labelled with a character from the alphabet or the empty string. We will use a boolean matrix M where $M[i, j] = 0$ iff we have not constructed $G_{i,j}$.

Construct
  Input: Weight matrix $L[m, n]$ corresponding strings X, Y.
  Output: $G_{m,n}$ the graph representing $LCS(X, Y)$.
  1 Initialize $V = (0, 0)$; $E = \Phi$; all entries in M to $false$.
  2 RecursiveCG(m,n)

RecursiveCG
  Input: $m, n \in \mathbb{N}$.
  1 If $L[i, j] = 0$ or $M[i, j] = true$ Return.
  2 Add (i, j) to V.
  3 If $X_i = Y_j$
    If $L[i - 1, j - 1] = 0$
      Add the edge $((i, j), (0, 0))$ with label X_i to E.
    else
      Add the edge $((i, j), (i - 1, j - 1))$ with label X_i to E.
      RecursiveCG$(i - 1, j - 1)$.
  4 If $L[i, j] = L[i - 1, j]$
    Add the edge $((i, j), (i - 1, j))$ with empty string as the label to E.
    RecursiveCG$(i - 1, j)$.
  5 If $L[i, j] = L[i, j - 1]$
    Add the edge $((i, j), (i, j - 1))$ with empty string as the label to E.
    RecursiveCG$(i, j - 1)$.

Clearly, it takes $O(mn)$ space and time to construct the graph. Note that if we do not use the matrix M then we have an exponential time algorithm. Also observe that it is the reversed graph that represents $LCS(X, Y)$, but this can be obtained from the graph constructed above.

(d)Now we give a version of the above procedure that gets rid of all the edges that are labelled with empty strings. In the recursive calls now we also pass the vertex in whose adjacency list we want to add an edge. Moreover, we only add edges when we see a match. the

RecursiveCG
  Input: $m, n \in \mathbb{N}$ and vertex u.
  1 If $L[i, j] = 0$ or $M[i, j] = true$ Return.
  2 If $X_i = Y_j$
    If $L[i - 1, j - 1] = 0$
      Add the edge $(u, (0, 0))$ with label X_i to E.
    else
      Add $(i - 1, j - 1)$ to V.
      Add the edge $(u, (i - 1, j - 1))$ with label X_i to E.
      RecursiveCG$(i - 1, j - 1, (i - 1, j - 1))$.
  3 If $L[i, j] = L[i - 1, j]$
    RecursiveCG$(i - 1, j, u)$.
  4 If $L[i, j] = L[i, j - 1]$
    RecursiveCG$(i, j - 1, u)$.

Clearly, the longest path in this graph has length k, the length of $LCS(X, Y)$. The width at any level of the graph is bounded by at most $m + n$. Thus $|V| = O((m + n)k)$.

NOTE: As stated in the class, we cannot ensure the bound of $O(k|\Sigma|)$ since merging of the adjacent vertices of the graph, so that the width at any level is bounded by $|\Sigma|$, engenders new strings that are not present in $LCS(X, Y)$.

(e) A BFS on the graph based upon the labels on the edges will give us a way to verify whether a string is in $LCS(X, Y)$ or not. The complexity is govenrned by the number of edges in the graph, i.e., $O(k^2(m + n)^2)$.