30/30

CMSC 641, Design and Analysis of Algorithms, Spring 2010

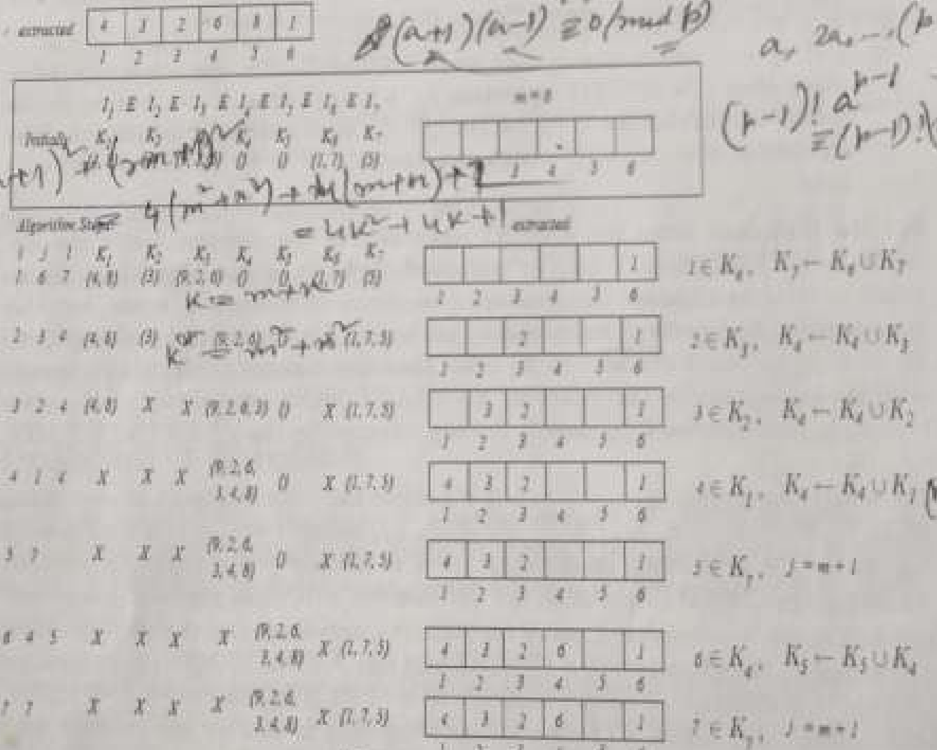
Sandipan Dey,

Homework Assignment - 3

March 2, 2010

Problem 1 (Offline minimum) Solution

Part (a)



Part (b)

Proof

The OFF-LINE-MINIMUM algorithm deals with n INSERT (I) operations ($\bigcup_{k=1}^{m} I_k$) and m EXTRACT-MIN (E) operations. To prove that the algorithm is correct, we need to prove that $\forall j \in \{1, 2, \ldots m\}$, $extract[j]$ contains the key returned by j^{th} E.

For $i = 1$, the algorithm considers the entire sequence $I_1, E, I_2, E, \ldots I_m, E, I_{m+1}$. It first finds a $j | i \in K_j$. There can be couple of cases:

1. $j = m + 1$, which means that the element 1 is inserted after the last EXTRACT-MIN, in which case it will NOT be part of the *extracted* array, since it will never get a chance to be extracted. The algorithm also does nothing ($j \neq m + 1$ check on line 3 ensures it), simply proceeds to the next larger element. Since the elements $\{1, 2, \ldots n\}$ are considered in the increasing order (ensured by the for loop in line 1), this element will never be considered again. Hence, this behavior is correct.
2. $j \neq m + 1$, which means that some EXRACT-MIN operation has taken place after this INSERT operation I_j. 1 being the smallest element in the set S, the immediate E operation (j^{th} E) must extract this element. The algorithm also correctly assigns $extracted[j] \leftarrow i$ at line 4, where $i = 1$ here.

For the 2nd case, after the INSERT operation of the element 1 and the immediate (j^{th}) EXTRACT-MIN is evaluated correctly by the algorithm, the algorithm tries to consider the remaining sequence of operations again, but this time without the particular I and E. This is done by the line 6, which performs $K_l \leftarrow K_l \cup K_j$ (since the keys in K_j other than the element i can only be considered for extraction by the following EXTRACT-MINS) and destroys K_j, since it already found $extract[j]$, namely the key returned by the j^{th} EXTRACT-MIN.

Therefore, for iterations $i = 2 \ldots n$ it considers only the sequence of operations $I_1, E, I_2, E, \ldots I_{j-1}, E, I_{j+1}, E, \ldots I_m, E, I_{m+1}$, where $l = j + 1$ in this case (it can be $> j + 1$ in other cases when $j + 1$ is already destroyed). Hence after removing the INSERT operation for the element 1 (it's not physically removed, but will never be considered, since i is strictly increasing) and the corresponding $extracted[j]$, the sequence of n INSERT and m EXTRACT-MIN operations get reduced to a different (smaller) sequence of $n-1$ INSERT and $m-1$ EXTRACT-MIN operations, hence a smaller subproblem that is exactly similar and on it the algorithm will work for the iterations $i = 2$ to n.

By applying the same logic for the smaller subproblem with $n - 1$ INSERT an $m - 1$ EXTRACT-MIN operations (consdered by the algorithm steps $\forall i =$

$2 \ldots n$), we can divide it into 2 parts again, one for $i = 2$ and the other for still smaller subproblem $i = 3 \ldots n$ and argue that the algorithm works correctly for $i = 2$. Continuing in this manner, $\forall i = k \ldots n$, each time we can divide the current problem into another subproblem with strictly non-increasing size in the sequence of operations (handled by the algorithm in iterations $i = k+1 \ldots n$) and prove the correctness of the k^{th} iteration. But i is increasing, hence we are done when we have $i = n$.

Part (c)

Implementation

- Start with each element as a singleton set in a disjoint set forest, with total n elements.
- In order to form sets K_j, $j = 1 \ldots m+1$ (in the worst case last $n-1$ of them possibly empty), $n-1$ UNIONs in the worst case.
- Line 2 basically then reduces to $j \leftarrow FIND-SET(i)$ and we have n such operations.
- Line 5 reduces to $l \leftarrow next(j)$, operation which is executed for n times in the worst case.
- Line 6 reduces to $l \leftarrow LINK(j, l)$ operation which is also executed for n times in the worst case.

✓

Hence, total number of operations $= m\prime = O(n)$

$\Rightarrow$ amortized time $= O(m\prime \log^* n) = O(n \log^*(n))$

or to provide a tighter bound, the amortized time $= O(n\alpha(n))$, where α is the inverse of the Ackerman function.

10

✓

Problem 2 Solution

As it can be seen from the figure 1, starting with $2^n + 1$ INSERT operations, followed by an EXTRACT-MIN (with CONSOLIDATE) operations, followed by $2^n - 1$ DELETE operations can create a Fibonacci Heap of height n, with n nodes (a chain).

Note that DELETE operation uses DECREASE-KEY + EXTRACT-MIN, where none of the DECREASE-KEY operation here can have cascade-cut, since every non-root node will have its child deleted only once.

Problem 3 Solution

Part (a)

10

Algorithm 1 Algorithm FIB-HEAP-CHANGE-KEY

$FIB-HEAP-CHANGE-KEY(H, x, k)$

```
1: if k < key[x] then
2:   call FIB-HEAP-DECREASE-KEY(H, x, k).
3: else if k == key[x] then
4:   return {do nothing}.
5: else {increase key}
6:   for each child y of x do
7:     call CUT(H, y, x).
8:   end for
9:   key[x] ← k.
10:  call CASCADING-CUT(H, x).
11: end if
```

- Lines $1-2$ have an amortized cost of $O(1)$, so have lines $3-4$ (comparison cost).
- Let's analyze the amortized cost for lines $5-10$, i.e., for the increase-key operation.

  By the potential method, potential before increase-key $= t(H) + 2m(H)$.

  Line 7 can increase the number of trees $t(H)$ by at most $D(n)$ (maximum degree of a node in the n-node Fibonacci heap $= O(lg\ n)$).

  Also, if we assume that the number of cascading cut recursive calls line 10 is c, then total decrease in number of marked nodes $= O(c)$, where the same call produces $O(c)$ additional trees, where c is a constant. Hence the potential after increase-key $= (t(H)+D(n)+O(c))+2(m(H)-O(c)))$.

  Hence, the change in potential is at most
  $= (t(H) + D(n) + O(c)) + 2(m(H) - O(c))) - (t(H) + 2m(H))$
  $= D(n) - O(c) = O(lg\ n)$.
- Total amortized time for FIB-HEAP-CHANGE-KEY $= O(lg\ n)$.

Part (b) ?

Deleting a node $\Rightarrow$ Decrease the corresponding key to $-\infty$, followed by Extract-Min, hence has an amortized cost of $O(1) + O(lg\ n) = O(lg\ n)$.

$\frac{1-\cos\alpha}{\sin\alpha} = \frac{1-a+\sqrt{1-a^2}}{a+\sqrt{1-a^2}}$

$1-(a+\sqrt{b})^2 = (1-a^2-b) - 2a\sqrt{b} \equiv c+d - 2c\sqrt{d}$

$a^2+b=1$

If we were to delete $min(r, n[H])$ particular nodes the amortized cost would be $= O(min(r, n[H]).lg\, n)$.

But, since we could delete arbitrary nodes, we hope to do better. Deleting singleton trees and leaf nodes is easy. So, by maintaining a pointer to leaf nodes in each tree, the amortized cost of pruning $min(r, n[H])$ nodes is $O(min(r, n[H]))$.

$\prod_{i=1}^{n+1}(a_i - i)$ $2K(n+1-j)$ $a_{n+1} = n+1$

$c\sqrt{d} = a\sqrt{b} \Rightarrow d = \ldots$

$c^2 + d = 1 - a^2 - b$

$c^2 + \frac{a^2 b}{c^2} = 1 - a^2 - b$

$\Rightarrow c^4 - (1-a^2-b)c^2 + \ldots$

$(a_1-1)(a_2-2)\cdots(a_{j-1}-j)(a_{j+1}-(j+1))\cdots(a_n-n)$

$p_1 \mid f(x_1)$, $f(x_1) = (a_0 + a_1 x_1 + a_2 x_1^2 + \cdots + a_n x_1^n) = q_1 p_1 \equiv 0 \pmod{p_1}$

$p_2 \mid f(x_2)$, $p_2 \mid (a_0 + a_1 x_2 + a_2 x_2^2 + \cdots + a_n x_2^n) = q_2 p_2 \equiv 0 \pmod{p_2}$

even odd

$p_m \mid f(x_m)$, $p_m \mid (a_0 + a_1 x_m + a_2 x_m^2 + \cdots + a_n x_m^n) = q_m p_m \equiv 0 \pmod{p_m}$

p_{m+1}: $\nexists x_i \in \mathbb{Z} \mid p_{m+1} \mid f(x_i)$.

$\Rightarrow (p_{m+1}, f(x_i)) = 1, \quad \forall x_i \in \mathbb{Z}$ $(a_j - j)\, c = 2K$

$(a_1-1)(a_2-2)\cdots(a_n-n)(a_{n+1}-(n+1))$

$p_{m+1} \le p_1 p_2 \cdots p_m + 1$

$p_{m+1} - 1 \le p_1 p_2 \cdots p_m$

$a_1, a_2, \ldots, a_n, a_{n+1} \in \{1, 2, \ldots\}$

$a_j = n+1$

$(n+1-j)$

$f(1)$, $f(2)$, $f(3)$ $p_{m+1}-1$

$(n+1)(j \pm K) - jK - (n+1) + 2(n+1)K$

$f(x_1) f(x_2) \cdots f(x_m) = q_1 q_2 \cdots q_m p_1 p_2 \cdots p_m$