CMSC 641 Design & Analysis of Algorithms Spring 2023
Section 01 Homework 6

Faisal Rasheed Khan

VB02734

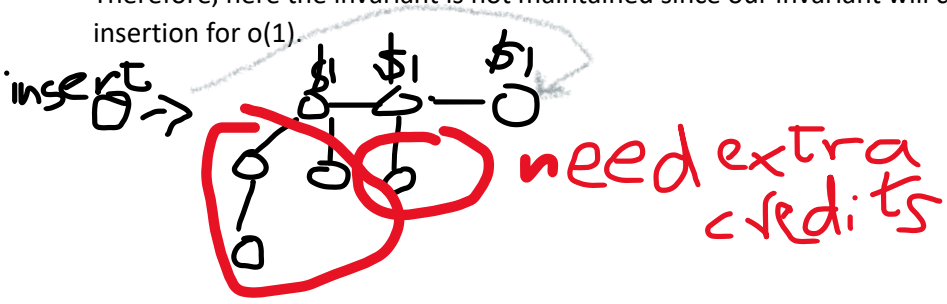
vb02734@umbc.edu

1 Frequent Consolidation

In Fibonacci Heaps, the consolidate procedure is called only during ExtractMin(). That makes the data structure rather "messy". In this problem, we explore some ways that might make Fibonacci Heaps tidier. Throughout, we will keep the same invariant as before: $1 credit on each root in the root list and $2 on each marked node.

1. Recall that in the standard implementation of Fibonacci Heaps, when we insert an item x, we just make x its own heap (with a single item) and add that heap to the root list. Suppose that we modify the Insert operation and have it call the consolidate procedure after each new item is added to the root list. Explain why we cannot maintain the invariant and still have Insert run in Θ(1) amortized time.

A. Here, we are doing consolidation after every insert. Consolidation occurs for the trees with the same degree.
To perform the consolidation we need o(log n) which would be done for $1 credit which is on the root.
But here still the insertion is not o(1) we need to pay extra credits for the future operations.
For example degrees are not in sorted order.
We need extra log n+1 credits to consolidate merging operations which would take o(1) insert time.
Therefore, here the invariant is not maintained since our invariant will change to perform insertion for o(1).



2. The astute observer would correctly claim that the previous question is silly because if we consolidate after every insertion, then the roots in the root list would always have unique degree. If we keep the roots sorted by degree in the root list (starting with degree 0), then we can easily combine the newly inserted x with the heaps already in the root list. The process will be very similar to the ripple-carry adder in the binary counter example for amortized analysis.
For example, if the root list has roots with degree 0, 1, 2, 4 and 7, then we fist merge the new heap (which has degree 0) with the degree 0 root. This makes a root with degree 1, which we merge with the existing root with degree 1. That makes a root with degree 2 which

gets merged and forms a root with degree 3. After that we stop merging and the resulting root list has roots with degree 3, 4 and 7. We can just use the $1 stored at each root to pay for the merges, add $1 to the new root and Insert would still run in Θ(1) amortized time. Explain what happens to the amortized running times of Extract-Min, Decrease-Key and Merge operations under such a scheme.

A. Since the process of the merging of the heaps are same as ripple carry adder, insert takes o(1) amortized time by merging using 1$ credit and the new root will get $1 credit.
Extract-Min:
The extract-min will take O(log n) amortized time since the consolidation process is done similar to the ripple carry adder and takes o(log n) to merge with sorted degrees.
Decrease-key:
The decrease-key will take o(log n) amortized time , i.e we need to look through the tree for searching the element as the credits are consumed for the other operations.
Merge:
The merge operations are paid by the credits, therefor the amortized time to merge is O(1).

3. Another variation of the previous scheme is to keep an array Root[ ] indexed by degree so that Root[i] points to the unique heap where the root has degree i (if one exists). That is, we keep an array of roots instead of a list of roots. Does the binary counter scheme still work to keep Insertion at Θ(1) amortized running time? Explain.
Explain what happens to the amortized running times of Extract-Min, Decrease-Key and Merge operations under our new scheme.

A. The binary counter scheme also works here as the indexes represents the degree of the tree. the process of the merging of the heaps are same as ripple carry adder, insert takes o(1) amortized time by merging using 1$ credit and the new root will get $1 credit.
Extract-Min:
The extract-min will take O(1) amortized time since the consolidation process is done similar to the ripple carry adder. The min element can be accessed from the top. And we know the position of degrees for the array.
Decrease-key:
The decrease-key will take o(1) amortized time , i.e the invariant is maintained by cascading cuts property and the credits are paid off.
Merge:
The merge operations are paid by the credits, therefor the amortized time to merge is O(1). Its because of the array accessing time O(1).

2 All Links before Finds

Suppose that we use the disjoint-set union data structure for a sequence of m Make-Set, Find-Set and Link operations, but we are told that all of the Link operations are performed before any Find-Set operation. (As usual, assume that there are n Make-Set operations.) Furthermore, suppose that we implement the data structure using both Union-by-Rank and Path Compression. Show that the total actual running time for all m operations is O(m).

Note: Recall that the Link operation is given the two roots of the trees to be merged. Thus, you do not need to perform any Find-Set operations to execute a Link. (This is just an exercise. It would be unclear how an application using this data structure would know which items are roots.)

Hint: Think about paying for the path compression during a Find-Set.

Note: There is no limit on the number of Find-Set operations. While we do know that m is the total number of operations (including Find-Sets), you cannot assume a relationship between m and the number of items n. I.e., the number of Find-Set operations could greatly exceed the number of items n.

A. Make-set = create set {x}
Find-set = return name of set that contains x
Link(name 1,name2) = union of name1 with name2
Union(x,y) : join sets containg x with y

Observations
1. If x is not a root, then rank(x) << rank(parent(x)).
2. If x becomes a non-root, then rank(x) becomes fixed forever.
3. If x is a root, size(x) ≥ 2^rank(x). This is an easy proof by induction on rank. When the rank of the root is 0, the tree has 1 item and the claim holds. A rank of r + 1 can only be achieved by linking two trees with roots with rank r. This means the size of the tree doubles while the rank increases by 1.
4. For r ≥ 0, there are at most n/2^r items that ever achieve rank r during the entire run of the data structure.
5. For all items x, rank(x) ≤ ⌊log n⌋. Otherwise, size(x) > n.

<u>Union-by rank:</u>
Rank r must have atleast 2^r descendants
All link operations are performed before Find-sets, o(1).

Make-set: O(1)
Link : o(1)
Union : o(log n)
Find-Set o(log n) amortized
m operations could be large and we don't have a relation b/w n and m
Worst case find-set operations are o(log n) and after that we can access the array easily, but link operations are done before find-set, total m operations, consider all link operations take m operations without find-set then

Time= m*o(1)
Time=O(m)

Considering for other cases,
time complexity = O(m)+O(1) +O(log n)
therefore, time Complexity =O(m)

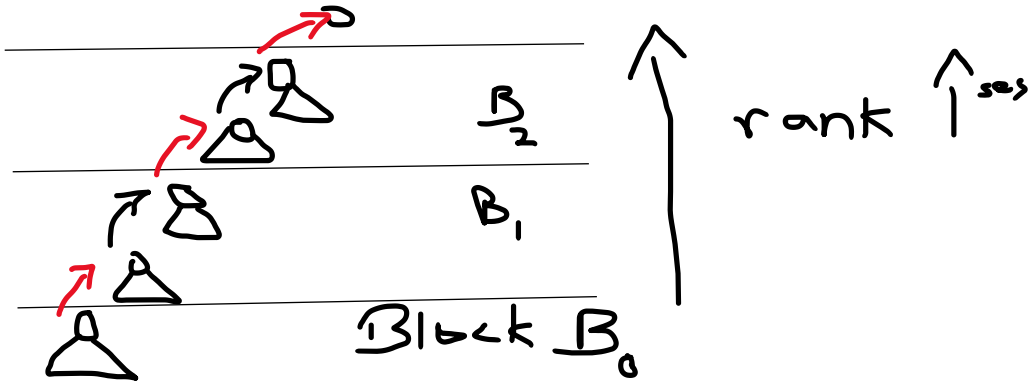
<u>Path Compression:</u>

Invariant: charge the node unless rank(y) << rank (p(y)), after that find-set will pay off

Since, after the path is compressed no need to traverse the log n path of the tree.
Divide ranks into blocks

B0	B1	B2	B3	B4

If rank(y) and rank(p(y)) are in different blocks, we think rank(y) << rank (p(y)), then Find-set pays



Paid by Find-set=

Paid by Node=

Link : o(1)
Union : 2 Find-sets + 1 link
Find-Set o(log log n) amortized
Make set O(1) + node charges =o(log log n)
m operations could be large and we don't have a relation b/w n and m
time complexity = O(m)+O(1) +O(log log n)
therefore, time Complexity =O(m)

3 Offline Minimum

Do Problem 19-1 in CLRS (Problem 21-1 in the third edition), except in part a use the following sequence of operations:

4 6 E 3 1 2 E 7 E 5 E 9 E 8 E E

Note: The premise of this problem is that you can turn the extract-min question around when the problem is offline. Instead of asking

Which item will this call to Extract-Min return?

you ask

Which Extract-Min call will return this item?

For example, if the item in question is 1 (the smallest item), then it will be returned by some call to Extract-Min (unless 1 is inserted after the last call to Extract-Min). So, the question is which call will return 1.

A. a.Given Sequence S=4 6 E 3 1 2 E 7 E 5 E 9 E 8 E E

The breakdown sequence S= I_1 , E, I_2 , I_3 , E, I_4 , E, I_5 , E, I_6 , E, E

I_1 ={4,6}

E

I_2 ={3,1,2}

E

I_3 ={7}

E

I_4 ={5}

E

I_5 ={9}

E

I_6 ={8}

E

E

➔ m=7

for i =1 to n

j, such that i∈ k_j ,

k_1 ={4,6}

extracted[1]=4

k_1 ={6}

k[2]=k[1] U k[2]

k[2]={6,3,1,2}

j, such that i∈ k_j ,

k_2 ={6,3,1,2}

extracted[2]=1

k_2 ={6,3,2}

k[3]=k[2] U k[3]

k[3]={6,3,2,7}

j, such that i∈ k_j ,

k_3 ={6,3,2,7}

extracted[3]=2

k_3 ={6,3,7}

k[4]=k[3] U k[4]

k[4]={6,3,7,5}

j, such that i∈ k_j ,

k_4 ={6,3,7,5}

extracted[4]=3

k_4 ={6,7,5}

k[5]=k[4] U k[5]

k[5]={6,7,5,9}

j, such that i∈ k_j ,

k_5 ={6,7,5,9}

extracted[5]=5

k_5 ={6,7,9}
k[6]=k[5] U k[6]
k[6]={6,7,9,8}

j, such that i∈ k_j ,
k_6 ={6,7,9,8}
extracted[6]=6
k_6 ={7,9,8}

j, such that i∈ k_j ,
k_7 ={7,9,8}
extracted[7]=7
k_7 ={9,8}

The first call to Extract-Min returns 4

The second call to Extract-Min returns 1

The third call to Extract-Min returns 2

The fourth call to Extract-Min returns 3

The fifth call to Extract-Min returns 5

The sixth call to Extract-Min returns 6

The seventh call to Extract-Min returns 7

Extracted = 4,1,2,3,5,6,7

b. The above method we are implementing is similar to the min-heap process. Where if we encounter input E we return the minimum element from the set and store it in the extracted array.

If E is encountered we return the minimum element, else we insert the value to the min heap we are creating and including all elements as a set.

Therefore, array extracted by offline minimum is correct.

c.

Implement Offline minimum as disjoint dataset as :

- Traverse through the given sequence, make-heap sets till the input is not E, this will take o(log n) time for inserting the link.
- If the input element is E, perform Extract-Min which will be the top element and takes o(1) time.
- Perform the union operations for the different elements to be inserted in the data structure, this will take o(log n).
- Repeat the process.

- The time taken would be o(n) for n input sequence + o(log n) for inserting the elements according to heap + o(1) for extract-min
- Therefor the time would be o(n).

References:

https://archive.org/details/introduction-to-algorithms-by-thomas-h.-cormen-charles-e.-leiserson-ronald.pdf/page/427/mode/2up

https://archive.org/details/AlgorithmDesign1stEditionByJonKleinbergAndEvaTardos2005PDF/page/n201/mode/2up

09-Fibonacci-Heaps-post.pdf - Google Drive

06.1-Amortized-Analysis-post.pdf - Google Drive

11-Disjoint-Set-Union-post.pdf - Google Drive

Notes on Disjoint Set Union.pdf - Google Drive

CMSC 641-01 Spring 2023 Shared Folder - Google Drive