6.046
PROBLEM SET 7

JOHN WANG

Collaborators: Xinran Liu

1. Problem 7-1

1.1. Problem A.

Problem 1.1. *Write pseudoode implementations of the Bib-Insert, Bib-Delete, and Bib-Min functions for the Bib Tree.*

Solution First, if $u == 2$, then it is the base size and it contains an array $A[0, 1]$ of two bits. Otherwise, we shall set $u = 2^{2^k}$ for some integer $k \geq 1$ so that $u \geq 4$. The summary and blocks structures of the tree are defined as in the original vEB structure.

Also note that we will use a helper function BibSuccessor for our delete function. In essence, the BibSuccessor function finds the next larger element in a Bib Tree. We also include a BibMember function which checks if an element x is a member of the Bib-tree. This function is used in part C of the problem set.

```
def BibInsert(tree, x):
    if global_min == null || x < global_min:
        global_min = x
    if tree.u == 2:
        tree.A[x] = 1
    else:
        BibInsert(tree.blocks[high(x)], low(x))
        BibInsert(tree.summary, high(x))

def BibMin(tree):
    if tree is root:
        return global_min
    if tree.u == 2:
        if tree.A[0] == 1:
            return 0
        elif tree.A[1] == 1:
            return 1
        else:
            return null
    else:
        i = BibMin(tree.summary)
        if i == null:
            return null
        else :
            j = BibMin(tree.blocks[i])
            return index(i,j)

def BibSuccessor(tree, x):
    if tree.u == 2:
        if x == 0 and tree.A[1] == 1:
            return 1
        else:
            return null
    else:
        i = BibSuccessor(tree.blocks[high(x)], low(x))
```

```
        if i == null:
            return index(high(x), i)
        else:
            i = BibSuccessor(tree.summary, high(x))
            if i == null:
                return null
            else:
                j = BibMin(tree.blocks[i])
                return index(i,j)

def BibDelete(tree, x):
    if tree.u == 2:
        tree.A[x] = 0
        if x == global_min:
            global_min = BibMin(tree)
    else:
        BibDelete(tree.blocks[high(x)], low(x))
        if tree.blocks[high(x)][0] is empty and
                BibSuccessor(tree.blocks[high(x)], 0) == null:
            BibDelete(tree.summary, high(x))
        if x == global_min:
            global_min = BibMin(tree)

def BibMember(tree, x):
    if tree.u == 2:
        return tree.A[x]
    else:
        return BibMember(tree.blocks[high(x)], low(x))
```

□

1.2. **Problem B.**

Problem 1.2. *Write the recurrences for the run-times of the operations in the revised data structure and solve the recurrences.*

Solution Let us examine BibInsert. We see that there are operations and checks in the first half of the code that can be performed in $O(1)$ time. However, if $tree.u! = 2$ then we will have two perform two BibInserts, one on tree.blocks[high(x)] and one on tree.summary. Notice that each of these structures is size $\sqrt{n}$. This means that we can write down the following recurrence $T(n) = 2T(\sqrt{n}) + O(1)$. Using the substitution of $2^k = n$ and $S(k) = T(2^k) = T(n)$, we find that $S(k) = 2S(k/2) + O(1)$. This recurrence solves out to $S(k) = O(k)$ by the first case of the master theorem. This implies that $T(n) = O(\lg n)$.

Now let us examine BibMin. The first half of the if statement can be performed in $O(1)$ time. However, in the second half, there is the possibility that one will have to perform BibMin twice, once on the tree.summary structure and a second time on the tree.blocks[i] structure. Since each of these have size $\sqrt{n}$, we obtain the recurrence $T(n) = 2T(\sqrt{n}) + O(1)$. Since we have just solved this, we find that $BibMin$ takes $O(\lg n)$ time. However, notice that if BibMin is called on the top-level tree (the root tree), then it will only take $O(1)$ time. We implement BibMin for other trees because we want to use it in our BibDelete method. However, in many applications, BibMin will only take $O(1)$ time.

Next, we shall look at BibSuccessor, which obtains the next larger element in a tree, given a starting element. We see that the first half of the if statement takes $O(1)$ time. The second half of the if statement will take $T(\sqrt{n})$ time to find the successor in tree.blocks[high(x)], because tree.blocks[high(x)] is of size $\sqrt{n}$. In the worst case, the algorithm will go on to another BibSuccessor, this time on tree.summary which is of size $\sqrt{n}$, and will then call BibMin once on tree.blocks[i], which is of size $\sqrt{n}$. The BibMin function will take $O\lg\sqrt{n}$ time. Thus, the recurrence for the BibSuccessor function comes out to $T(n) = 2T(\sqrt{n}) + O(\lg\sqrt{n})$. Noticing that $\lg\sqrt{n} = (1/2)\lg n = O(\lg n)$, we have the currence $T(n) = 2T(\sqrt{n}) + O(\lg n)$. We use the substitution of $2^k = n$ and $S(k) = T(2^k) = T(n)$ as before, and we find this recurrence transforms to $S(k) = 2(k/2)+O(k)$. Using the master theorem, we find that $S(k) = O(k \lg k)$ so that $T(n) = O(\lg n \lg\lg n)$.

Now let us examine BibDelete. The first part of the if statement requires $O(1)$ time to evaluate. In the worst case, BibDelete will call itself twice, first on tree.blocks[high(x)] and second on tree.summary.

These are Bib structures of size $\sqrt{n}$, so the running time will require $2T(\sqrt{n})$. Finally, one could also run BibSuccessor on tree.blocks[high(x)] and BibMin on the tree. The BibSuccessor function is run on a Bib tree of size $\sqrt{n}$, so the running time is $O(\lg\sqrt{n}\lg\lg\sqrt{n})$. The BibMin function will require $O(\lg n)$ time. Thus, the recurrence is:

$$T(n) = 2T(\sqrt{n}) + O(\lg\sqrt{n}\lg\lg\sqrt{n}) + O(\lg n) \tag{1.1}$$

$$= 2T(\sqrt{n}) + O(\lg n\lg\lg n) \tag{1.2}$$

$$S(k) = 2S(k/2) + O(k\lg k) \tag{1.3}$$

Where we have used the substitution $2^k = n$ and $S(k) = T(2^k) = T(n)$. Solving this recurrence, we find that $S(k) = O(k\lg^2 k)$ using the master theorem. Therefore, we find that BibDelete has running time of $T(n) = O(\lg n\lg^2\lg n)$.

Now, we will give the running time of BibMember, which will be used in part C of the problem set. We see that tree.blocks[high(x)] is of size $\sqrt{n}$, so the recurrence becomes $T(n) = T(\sqrt{n}) + O(1)$. Using the same substitution as in class, we know that the recurrence evalutes to $T(n) = O(\lg\lg n)$. Therefore, we see that checking if an element is a member in the Bib-Tree requires only $O(\lg\lg n)$ time. □

1.3. **Problem C.**

Problem 1.3. *Ben Bitdiddle decides to use the Bib-Tree data structure in his implementation of Prim's algorithm. You can assume that the input graphs of interest all have integer weights in the range from 1 to n and that edge weights are allowed to repeat. Show how Ben would make use of his Bib-Tree and provide an analysis of time complexity for this implementation of Prim.*

Solution First, since edge weights are allowed to repeat, Ben needs to use both a Bib-Tree and an array storing pointers to all of the vertices. The array will have indices from 1 to n and will be much like a hash-table with chaining. If two vertices have the same weight i, they will both will be stored in the array at position i. Position i will hold a pointer to the first vertex, then the first vertex will contain a pointer to the second vertex. In this way, we can deal with duplicate edge weights without changing the Bib-Tree data structure. Since insert and delete operations from this array will take $O(1)$ amortized time (similar to the analysis of hash tables if the edge weights are evenly distributed), we can assume that the array doesn't add anything to the runtime of the Bib-Tree.

Now, we shall provide pseudocode for Prim's algorithm, assuming that the StructInsert and StructDelete functions will insert and delete from both the BibTree and the array. These functions operate by first calling BibInsert and BibDelete, then walking up the appropriate chain in the array and deleting an vertex and reordering the pointers. The TreeMin function performs a StructMin operation, which requires $O(1)$ time since the min is performed on the root of the Bib-Tree. Also, the StructFind function evaluates the BibMember function on the Bib-Tree. If the BibMember function returns 0, then StructFind returns false and true otherwise. We also know from the analysis in part B that the BibMember function takes $O(\lg\lg n)$ time, which means that StructFind takes the same amount of time.

Here is the pseudo code for Ben's implementation of Prim's:

```
def BenPrim(G, w, r):
    T = []
    for each u in G.vertices:
        u.key = n
        u.parent = null
    r.key = 0
    struct = Structure()
    for each u in G.vertices:
        StructInsert(struct, u)
    while StructMin(tree) != null:
        u = StructMin(tree)
        StructDelete(struct, u)
        if u.parent != null:
            T.append((u, u.parent))
        for each v in G.Adj[u]:
            if StructFind(struct, u) and w(u,v) < v.key:
                StructDelete(tree, v)
                v.parent = u
```

```
            v.key = w(u,v)
            StructInsert(tree, v)
    return T
```

Since StructInsert, StructDelete, and StructMin all handle inputs where v is a vertex, we are able to decrease the key by first delete v from the structure, then inserting an updated key back into the structure. Note that these functions handle vertices through the separate array structure storing the vertex for a give key, and the Bib-Tree which stores the key values themselves.

For an analysis of runtime, we assume there are V vertices and E edges. We see that the initialization requires $O(V)$ time. Inserting all the vertices into the structure takes $O(V)O(\lg n) = O(V \lg n)$ time. Next, we loop through all vertices in the graph, and perform a StructMin operation and a StructDelete operation. Moreover, we perform an additional $O(E)$ StructDelete and StructInsert operations when updating the key values of a vertex (we know it is $O(E)$ because there are only E possible edges to go through in the adjacency matrix). Since $V = O(E)$, we see that the total running time is $O(V)+O(E)O(\lg n \lg^2 \lg n+\lg n) = O(V) + O(E \lg n \lg^2 \lg n)$. Therefore the total running time of the algorithm is $O(V + E \lg n \lg^2 \lg n)$.

For the proof of correctness, we cite the proof of Prim's algorithm shown in class. Since our algorithm merely implements Prim's using a particular implementation of a min-priority queue, the proof of correctness still holds. We still always have a subset of an MST and can show using a loop invariant that the final result from the algorithm must therefore be an MST. □