Assignment 1
Zhaocheng Li / V00832770
CSC425 | Instructor: Valerie King | Fall Term

Question 1:
Given a dynamic table that allow insertion and deletion, and recall that in the ***amortized analysis***, the amortized analysis guarantees the average performance of each operation in the worst case.

and for accounting method, we determine an amortized cost of each operation. When there is more than one type operation, each type if operation may have a different amortized cost.

Consider the insertion operation **TABLE-INSERTION(T, x)** performed on the dynamic tables, we can give a similar function for the deletion operation *TABLE-DELETION(T, x)*, where T is the object representing table, T.table contains a pointer to the block of storage representing the table; T.num contains the number of items in the table; and T.size gives the total number of slots in the table. Moreover, we determine the load factor is 1/4 of the table size.

```
TABLE-DELETION(T, x):
    if T.num <= T.size/4 :
        allocate new-table with T.size/4 slots
        insert all items in T.table into new-table
        free T.table
        T.size = new-table
        T.size = T.size/4
    delete x from table T
    T.num = T.num - 1
```

By accounting method,

- for each insertion, we charge 3 dollars on the inserted element for each insertion.
    - The inserted element spends immediately 1 dollar for insertion operation;
    - and it save 2 dollars itself as credit for re-insertion of itself and another element when the table size doubles.
- for each deletion, we charge 2 dollars on deleted entry for each deletion.
    - The deletion cost 1 dollar immediately,

- Another dollar on the emptied slot is spent for the re-insertion of other one element when table size contracts.

Hence for n INSERTION operations, it has been proven that the total cost is bounded above by 3n, and then the amortized cost is 3 for each insertion. Similarly, for n DELETION operations, the total cost is, in this case, bounded above by 2n, and then the amortized cost is 2 for each deletion.

Moreover, for n operations including INSERTION and DELETION, since each operation requires constant cost, and thus the n operations are O(n), and O(1) per operation.

[Additional Question: 17.4-3]
As what the question describes, we contract the table to the table with 2/3 of the original table size as the load factor drops to & below 1/3.

It seems identical in logic as the method with halving the table size when load factor drops to & below 1/2 at first glance. But there is a crucial difference.

Suppose there are n operations including both DELETION and INSERTION, we assume that we still charge 2 dollars for each DELETION operation. In a desired situation, since the table could contract to the 2/3 of the original size, but double based on it as half of new table. So the table tend to get bigger and bigger and accordingly, in a long term the potential of the following deletion operations WILL NOT cover the cost of copying old element to another table. So charging 2 dollars is not enough.

Instead, if we charge 3 dollar for each DELETION. A dollar for deletion immediately, 2 dollars for the old element to re-inset, since if we assume the old table has size x. After doubling, 1/3 of the new table is 2x/3, which never exceeds the potential 2*(x-2x/3). So for each DELETION operation, the amortized cost is bound by 3, which is constant. And for n operations of INSERTIONs and DELETIONs, the cost is O(n) as well and O(1) for each operation.

Question 2:
Part(a):
Recall that

- A binary search tree rooted at node x is said to be a-balanced, for 0< a

<1, iff both sizes of x's left and right subtree are ≤ a*size(x), and both subtrees are also a-balanced.

- A inorder tress walk on a binary search tree allows us to print out nodes in the order of keys. i.e., it recursively prints out left subtree, root, and right subtree in order.

For this question, we apply **linear size array** and **inorder traversal algorithm**.

For a binary search tree with n nodes, we apply an inorder traversal T on it, and print out all nodes in order into the linear size array. Then we can recursively find the median of the array as the root and break array into three part: left array, root, and right array, and then keep doing it on each smaller arrays. In this case, since we can allocate median using T each time in constant time, the recurrence is

- $T(n)=2T(n/2)+1$

which has the linear solution, meaning it uses linear time. Since the original array is binary search tree visited by inorder traversal, the subarrays must also be binary search tree, and all the subarrays cut in half must satisfy 1/2-banlanced property.

Part(b)
We are proving by induction that any 2/3-balanced tree having nodes <= $(2/3)^{-b}+b$ has the height at most b, where

- Base: for b=0, then we has only one node and $(2/3)^0=1$, thus the assumption holds for the base case.

- Induction Hypothesis: Assuming the assumption is also valid for b=k, and then for b=k+1,

- Induction Step: Since both of subtrees are 2/3-balanced, then by the rules we have

lefttree.size ≤ (2/3)*righttree.size

then

$$\begin{aligned} \text{righttree.size} &> \text{roottree.size}-(2/3)*\text{roottree.size}-1 \\ &> (1-2/3)*\text{roottree.size}-1 \\ &> (1-2/3)*[(2/3)^{-k-1}]+(k+1)-1 \\ &> (1/2)*[(2/3)^{-k}]+k \end{aligned}$$

also

$$(1-1/2)*[(2/3)^{-k}]+k = ((1/2)^{-1} - 1)*[(2/3)^{-k+1}]+k$$
$$\geq (2/3)^{-k+1}+k$$

then

$$\text{righttree.size} > (2/3)^{-k+1}+k$$

It contradicts to the fact since it holds for even smaller values of b and any child of a tree with height b has height b-1.

Part(c): For either Deletion or Insertion, we need to:

1. Traversal through the tree to find the spot to add-on or the node to be removed.
2. Operating a single Insertion/Deletion operation.
3. Adjust the corresponding subtree from 2/3-balanced to 1/2-balanced.

The worst case in the first step could cost log(n);
The second step costs 1;
And the third step, which is re-binding nodes to satisfy 1/2-balanced once it doesn't after insertion/deletion, can actually cost log(n)

Therefore, the amortized cost is 2log(n)+1 per DELETION/INSERTION operation.

Question 3:
Part(a): Suppose block(i,j)=[B(i,j),...,B(i,j+1)-1]

Rank	0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
Level1	A	B	C	D	E	F	G	H	I	J	K
Level2											
Level3											
Level4											

We are going to need four levels, given the help from the notes and textbook, we now get for level 2-4:
level-2
block(2,0)=[B(2,0),B(2,1)]=[0,2^2] as well
block(2,1)=[B(2,1),B(2,2)]=[2^2,2^4]
block(2,2)=[B(2,2),B(2,3)]=[2^4,2^10]
level-3

block(3,0)=[B(3,0),B(3,1)]=[0,2^4]
block(3,1)=[B(3,1),B(3,2)]=[2^4,2^10]
level-4
block(4,0)=[B(4,0),B(4,1)]=[0,2^10]

Thus by level(x)=min. i^{th} level when rank(x) and rank(p(x)) are in the same block. We have

Nodes	A	B	C	D	E	F	G	H	I	J	K
Level	2	3	2	4	2	2	2	2	2	2	0

Part(b): Similarly, we are going to need 4 levels, this is answer for (i).

Rank	0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
Level1	A	B	C	D	E	F	G	H	I	J	K
Level2											
Level3											
Level4											

(ii) same question as (a),

level-2
block(2,0)=[B(2,0),B(2,1)]=[0,2^2]
block(2,1)=[B(2,1),B(2,2)]=[2^2,2^4]
block(2,2)=[B(2,2),B(2,3)]=[2^4,2^8]
block(2,2)=[B(2,2),B(2,3)]=[2^4,2^10]
level-3
block(3,0)=[B(3,0),B(3,1)]=[0,2^4]
block(3,1)=[B(3,1),B(3,2)]=[2^4,2^10]
level-4
block(4,0)=[B(4,0),B(4,1)]=[0,2^10]

Then we have levels:

Nodes	A	B	C	D	E	F	G	H	I	J	K
Level	2	3	2	4	2	2	2	3	2	2	0

(iii) The time required for amortized analysis will increment and based on the notes, the total cost is considered
$\sum_{i=0}^{\alpha(m,n)} \sum_{j\geq 0} n_{ij}(b_{ij}-1)$, where b_{ij} is then number of level i-1 blocks partitioning block(i,j).
Thus we have,

$$\sum_{i=0}^{\alpha(m,n)+1} \sum_{j\geq 0} n_{ij}(b_{ij}-1) \leq O\big(n\alpha(m,n)\big) + n\sum_{i=0}^{\alpha(m,n)} \sum_{j\geq 0} 2^{B(i-1,j)}$$

$$\leq O\big(n\alpha(m,n)\big) + n\sum_{i=1}^{\alpha(m,n)+1} O(logn)$$

$$= O\big(n\alpha(m,n)\big)$$

Therefore, the running time is bounded by $O\big(n\alpha(m,n)\big)$.

Question 4:

Recall the Even-Shiloach implement two parallel processes A and B, and each update is to delete edge {u, v}

For each deletion, process A detects if the deletion of {u, v} breaks the connected component: in parallel, it searches the graph at u, searches the graph at v. If one search finishes without visiting the other node, output "broken". Such algorithm require running time O(|E| * log|E|) since there are possibly at most 1/2 of the originally connected component is separated.

While the process B tries to restore the BFS if {u,v} is an edge in the BFS. If it cannot be restored, then the BFS breaks into two. Thus its running time is O(|V|* |E|).

Assuming the queries are O(1), then for each update in the worst case, the running time of process A is O(log|E|) since it won't need to go through every edge to detect the connectivity. And the running time of process B is O(|V|) for the same reason.

So the running time in the worst case for Even-Shiloach is O(|V| + log|E|) which is bounded by O(|V|).

Question 5:

The ET-trees should be presented by a table of edges {parent node, child node}, where the node traversed first is the parent node, connecting with two nodes of the tree. Such edges can operate a traversal from both directions. So that when an active occurrence p1 is deleted, we can quickly look up the table and find the edges with p1 being child node and being parent node in constant time, and merging them into new edges {parent of p1, child of p1} in linear time.