CMSC 641 Design & Analysis of Algorithms Spring 2023
Section 01 Homework 4

**Faisal Rasheed Khan**

**VB02734**

vb02734@umbc.edu

**1 A balanced binary-search tree**

The elegance of red-black trees and splay trees may sometimes leave us with a sense of awe. How did they come up with such a nifty scheme to keep the binary search tree balanced? It is actually not that hard to balance a binary search tree. Many simple schemes can give us Θ(log n) amortized running time. They may be lacking in elegance, but quite easy to understand.

Consider the following scheme. First, we store at every node x the number of items in the subtree rooted x. This allows us to determine when the binary search tree becomes unbalanced. Let's define a node x to be horribly unbalanced if the left subtree of x has ≥ 5 times the number of nodes than in its right subtree, or vice versa. All we will do in this scheme is to "fix up the subtree" if we detect that x is horribly unbalanced.

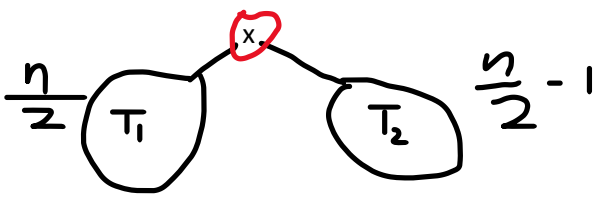
1. Suppose that none of the nodes in a binary search tree T with n items is horribly unbalanced. Argue that the height of T is O(log n). Hint: use recursion/induction.

A. Proof of Induction on height of binary search tree:

Induction Hypothesis P(n):
The height of binary search tree T with n items which is not horribly unbalanced is O(log n)

Base Case:
P(1) holds since height of a binary search tree with one element is 0. Log 1=0.

Induction Case:
Assume P(k) holds for all k<n, k less than n items. We want to show that P(n) holds.
Consider the left subtree $T_1$ of x and the right subtree $T_2$ of x. Each having half the items of n without the root x




Consider the left subtree $T_1$ , k=n/2

By Inductive Hypothesis, height of left subtree=log(k)

height of left subtree=log(n/2)

height of left subtree=log(n)-log 2

Consider the right subtree $T_2$ ,k=n/2 - 1

By Inductive Hypothesis, height of right subtree=log(k)

height of right subtree=log(n/2 - 1)

height of right subtree=log(n-2)-log 2

By induction hypothesis, the subtree with k items has height O(log k)=O(log(n/2))=O(log n)

Thus,

N items=k+1

height of binary search tree T =O(log(n/2 + 1))

=O(log(n+2)-log 2)

=O(log n)

Therefore, P(n) holds.

2. Suppose that during an insertion, we detect that some node x is horribly unbalanced. We can fix the subtree rooted at x by taking apart the subtree and adding every item from this subtree into a balanced tree that is as balanced as possible. Describe how to accomplish this re-balancing in O(m) worst case actual time, where m is the number of items in the subtree rooted at x.
Note: Do not use pseudocode to describe your algorithm. For example, if you want to sort some numbers in an array A, just say ”Sort the values in A.”

A.

i. Let T be the subtree of m elements rooted at x which is unbalanced.
ii. Run the inorder traversal for the subtree rooted at x, This will retrieve the nodes in the sorting order. Store this m elements in an array InO.
iii. Let $T^{|}$ be the balanced subtree to be attached to x.
iv. The median node of the inorder traversal (InO) of subtree will be the root for $T^{|}$
v. The elements left to the median are inserted to the left subtree of $T^{|}$ and the elements right to the median are inserted to the right subtree of $T^{|}$
vi. Recursively repeat the steps iv,v till all the elements are inserted to the subtree.
vii. Attach this balanced subtree $T^{|}$ in place of unbalanced subtree T rooted at x.

To compute the worst case actual time, lets consider the operations done to achieve the balanced subtree.

To retrieve elements in sorting order we did inorder traversal which takes O(m) time to complete and to store the m elements we need array of size m.

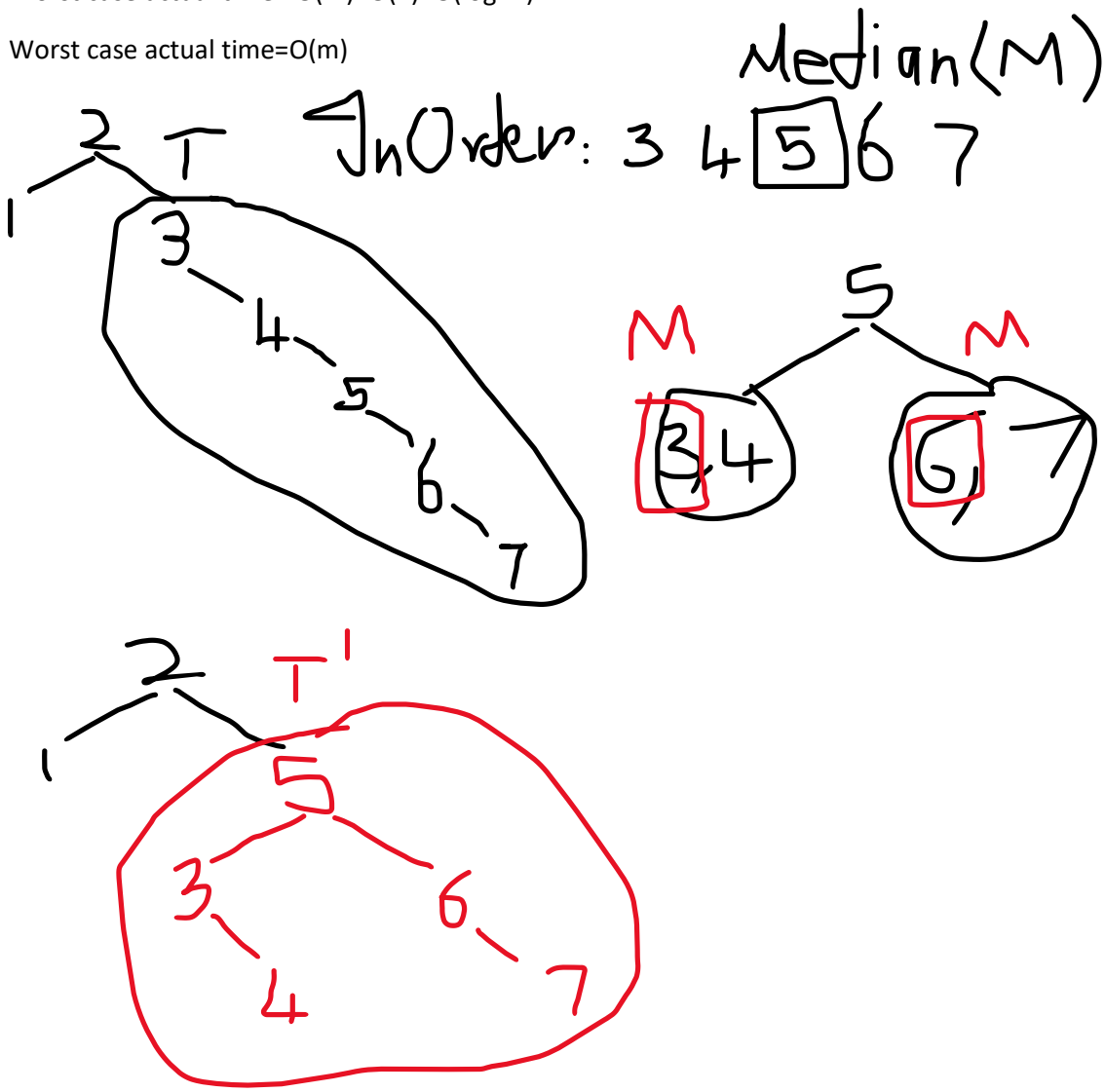
Median element can be known in O(1) time as elements are in sorted order.

Inserting m elements doesn’t take more than O(log m) time, since we are inserting in the sorting order and its height would be O(log m).

Worst case actual time= Inorder time + Median element time + Insert time

Worst case actual time= O(m)+O(1)+O(log m)

Worst case actual time=O(m)




3. (revised) When you a insert a new key k, you may encounter several nodes that are horribly unbalanced nodes on the path from the root to where k gets inserted. Where should you perform a re-balance first? at the horribly unbalanced node closest to the root? or closest to k? Justify your response.

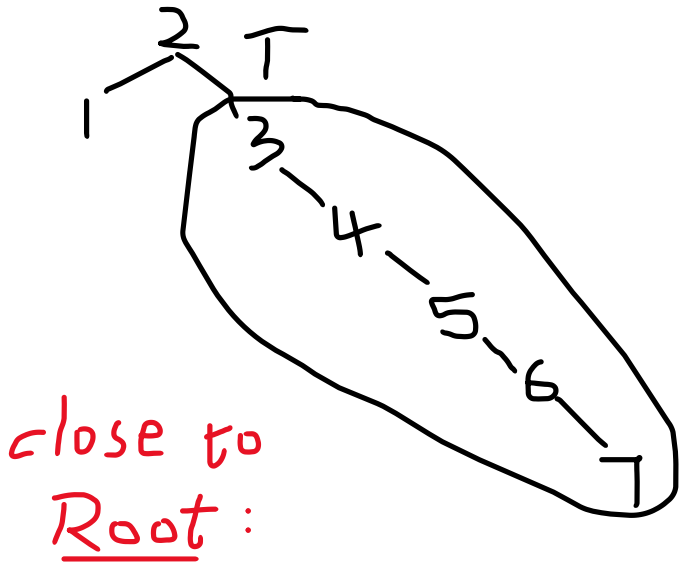
A.

- The re-balance should be performed at the horribly unbalanced node closest to new key k.
- When u balance at the horribly unbalanced node closest to the root you might end up balancing the subtree of the root again and again when its get unbalanced which is expensive, O(m) time complexity.

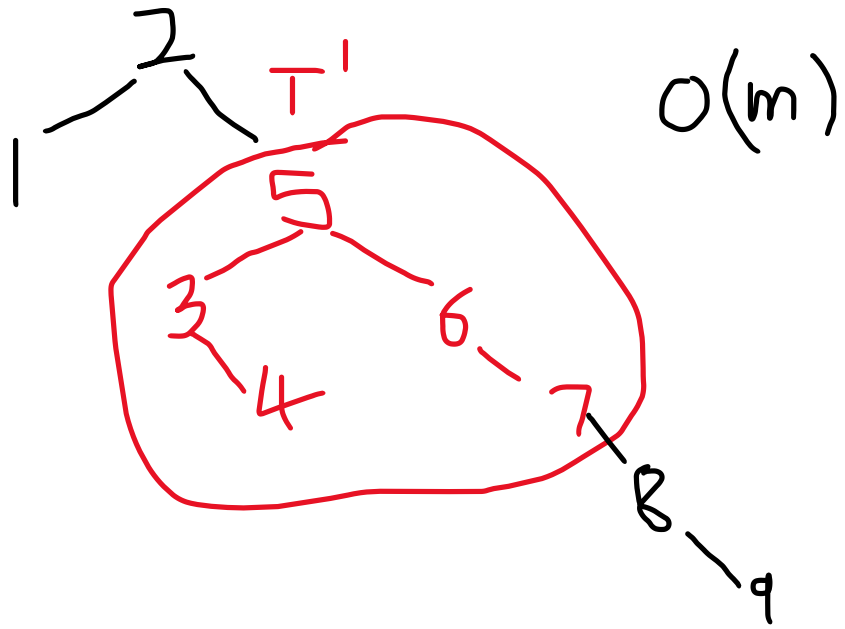
- When u balance at the horribly unbalanced node closest to key k, you just need rotations to balance and it ensures to maintain balance for the not seen keys in future. Here u need not to balance entire subtree

Consider the following keys:2,1,3,4,5,6 Binary search tree and insert 7.

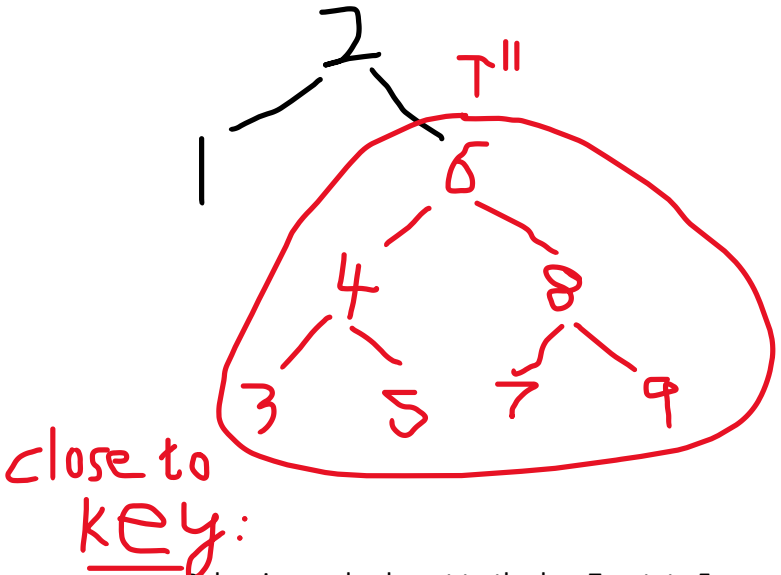
Inserting 7 makes unbalanced tree



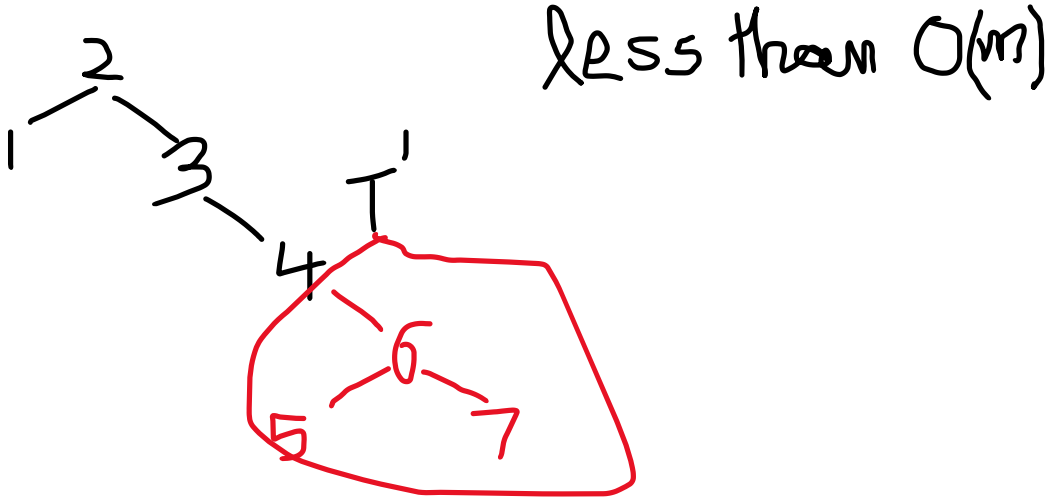

Balancing node closest to the root and consider new keys 8,9, will become unbalanced



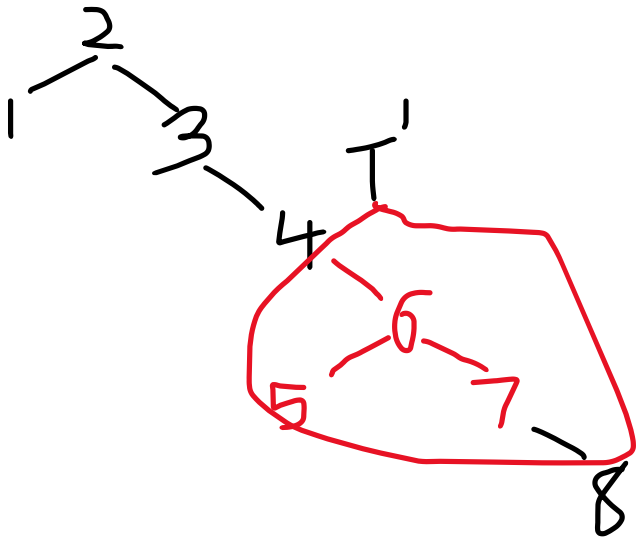

Balancing node closest to the root



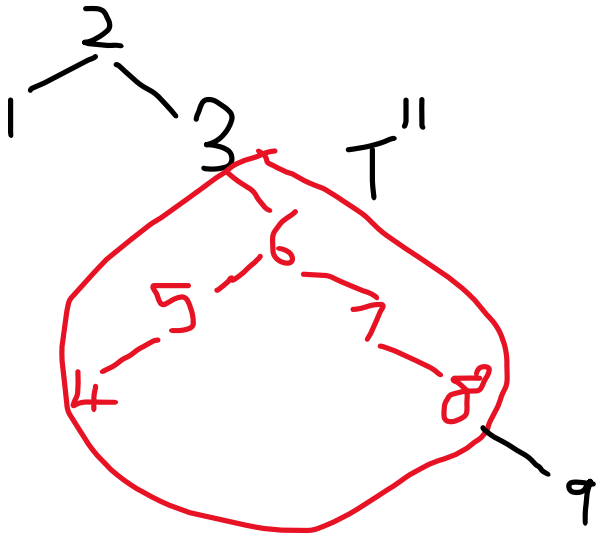

Balancing node closest to the key 7, rotate 5



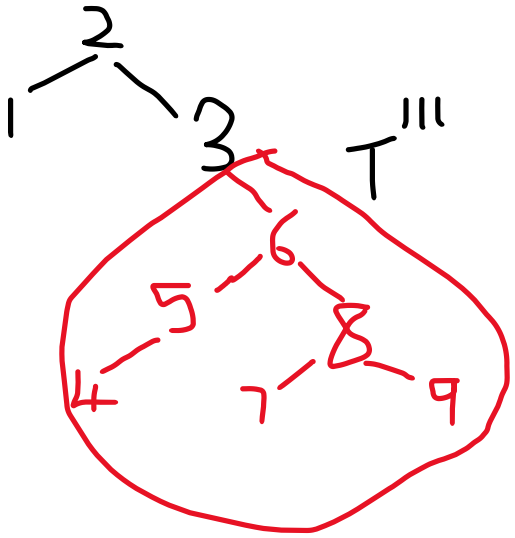

Balancing node closest to the key and consider new keys 8,9, becomes unbalanced




Rotate 4




Rotate 7




Balancing node closest to key doesn't require O(m) time everytime as compared to balancing node closest to root.

4. Devise an amortized analysis for this data structure using the accounting method. Your analysis must show that the amortized running time of insert and search are O(log n).
Note: You must state an invariant for your accounting scheme and argue that the invariant is maintained after each insert and search operation, including operations that perform re-balancing operations.

A. <u>Amortized analysis using Accounting Method:</u>

Amortized time = Actual Time + Credits Added – Credits Removed
Actual cost of $i^{th}$ operation is $c_i$ and the amortized cost of $i^{th}$ operation is $\hat{c}_i$ ,then

$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$ for all sequences of n operations

Invariant: Each node has (log (n+1)) credits

Claim1: Amortized running cost of insert is O(log n)

Proof:

- Each node upon insertion has (log (n+1)) credits. These credits are used to pay for future operations whenever the binary search tree gets horribly unbalance for the node to be inserted.
- Whenever the node which is to be inserted makes the binary search tree horribly unbalance, then the rotations to balance the tree uses these credits such that the invariant is maintained.
- After every insertion the credit pays off for the horribly unbalance node at most log n credits and invariant is maintained.
- Amortized time = O(log n) + log n – log n
- Amortized time = O(log n)
- Therefore, the invariant is maintained for insert and also for its rebalancing operations.

Claim2: Amortized running cost of search is O(log n)

Proof:

- Each node has (log (n+1)) credits. These credits are used to pay for future operations.
- Whenever the node is being searched it deducts a credit from the available credits of (log(n+1)) such that the invariant is maintained.
- Amortized time = O(log n) + log n – log n
- Amortized time = O(log n)
- Therefore, the invariant is maintained for search operations.

**2 Sorted Array with Sidekick**

Storing items in a sorted array has the advantage that binary search can find an item in Θ(log n) time. However, insertion into a sorted array is cumbersome and takes Θ(n) time.

A Sorted Array with Sidekick (SAwS) combines a sorted array with a smaller unsorted array, the sidekick. New items inserted in a SAwS are added to the sidekick array, which takes O(1) actual time since the sidekick is not sorted. When the number of items in the sidekick grows beyond √ n, the sidekick is sorted and merged into the sorted array. After this merger, the new sidekick is empty. To search in a SAwS simply do a binary search in the main array and a linear search in the sidekick.

Notes:

• Throughout this problem, let n be the number of items currently stored in the SAwS. This means n changes when items are added to or removed from the SAwS. You should analyze the running times in terms of n.

• Just assume that the arrays have enough space allocated. It is possible to combine this data structure with dynamic tables, but let’s keep it simple.

• (revised) Again, for simplicity, just assume that in our use case we will never insert duplicate keys into this data structure. (It is possible to handle duplicates, but it gets rather complicated when we delete a key that has duplicates still in the SAwS.)

• You know nothing about the type, distribution or range of the keys. So you cannot use counting sort, bucket sort, radix sort or any linear-time sorting algorithm.

• You may assume without further comment that in the worst case sorting m items takes Θ(m log m) time, binary search in a sorted array of m items takes Θ(log m) time, finding the median of an unsorted array of m items takes Θ(m) time and merging two sorted arrays of m and l items takes Θ(m + l) time.

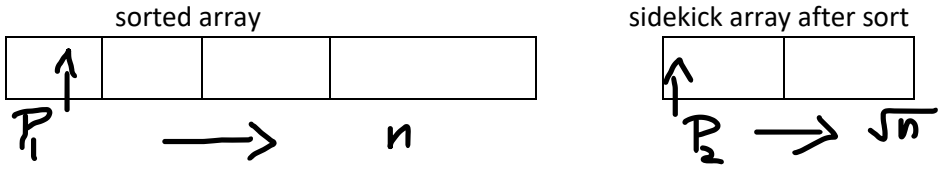
• Do not write pseudo-code. If you want to sort an array A, just say "Sort the array A."

**Questions:**

1. Briefly describe how the sidekick array can be merged into the sorted array in Θ(n) actual time.

A. According to the given information if the sidekick array grows beyond $\sqrt{n}$ it is sorted and merged into the sorted array.
Consider the sidekick array with size $\sqrt{n}$ is sorted using Merge Sort , the time taken is $\theta(\sqrt{n}\ log\sqrt{n})$
Now both the arrays are sorted, we can merge the sidekick array to sorted array using the two pointer technique.
One pointer $p_1$ at first index sorted array and another pointer $p_2$ at the first index of sidekick array.

if($p_1 < p_2$) : add $p_1$ index element and increment $p_1$ by 1
if($p_2 < p_1$) : add $p_2$ index element and increment $p_1$ by 1
$p_1$ increments till n
$p_2$ increments till $\sqrt{n}$
if any of the pointer reaches the end concatenate the remaining left elements to the sorted array




$\theta$(n) > $\theta(\sqrt{n}\ log\sqrt{n})$, need not consider sorting time of sidekick array as its less

For iterating the pointers, the pointer index of sorted array considers all the n elements.

Therefore, Time for merging = $\theta$(n)

2. Using the accounting method, show that the insert and search operations in a SAwS can be accomplished in O( √ n) amortized time. (Ignore the delete operation for now.)

Note: State an invariant for your accounting scheme and argue that the invariant is maintained after each insert and search operation, including operations that merge the sidekick into the sorted array.

A. Amortized analysis using Accounting Method:

Amortized time = Actual Time + Credits Added – Credits Removed
Actual cost of $i^{th}$ operation is $c_i$ and the amortized cost of $i^{th}$ operation is $\hat{c}_i$ ,then

$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$ for all sequences of n operations

Invariant:

Put $1 credit on each insert in sidekick Array when inserted in SAwS Array

Put $2 credit on each element in sidekick Array to Merge and Copy Operations

Total credits for sidekick Array = 3 credits

The cost would not exceed $3\sqrt{n}$ at any time

Claim1: Amortized running cost of insert operation in SAwS Array is O($\sqrt{n}$)

Proof:

- Each element has 3 credits for sidekick array $3\sqrt{n}$. These credits are used to pay for future operations.
- (sidekick array) whenever merging is done when the elements exceed $\sqrt{n}$ these credits will pay for the operations such that the invariant is maintained.
- Amortized time = O(1) + $3\sqrt{n}$ – $2\sqrt{n}$
- Amortized time = O($\sqrt{n}$)
- Therefore, the invariant is maintained.

Claim2: Amortized running cost of search operation in SAwS Array is O($\sqrt{n}$)

Proof:

- Each element has 3 credits for sidekick array $3\sqrt{n}$. These credits are used to pay for future operations.
- The Actual cost is searching for the element in both the arrays i.e. logn + $\sqrt{n}$ logn for sorted array and $\sqrt{n}$ for unsorted array.
- Amortized time = O(1 + logn + $\sqrt{n}$) + 0 - 0
- Amortized time = O($\sqrt{n}$)
- Therefore, the invariant is maintained for search operations.

3. Describe how to implement the delete operation in a SAwS so that search, insert and delete each takes O( √ n) amortized time. The delete operation is given a key x and must find then remove x from the SAwS data structure. For sorted arrays, it is convenient to simply mark the location of x as “removed”. (Assume the programming language supports this.) For unsorted arrays, you can either mark x as removed or move the last item to the vacated

location. However, n is still the number of items in the SAwS and does not include the locations marked “removed”.
Note that a delete makes either the main array or the sidekick array smaller by 1 and that repeated deletions reduce the values of n and √ n significantly. In particular, the size of the sidekick array might exceed √ n not because we added items to the sidekick, but because the value of n is reduced.
Note: You should revise the invariant for your accounting scheme and argue that the invariant is maintained after each insert, delete and search operation, including operations that merge the sidekick into the sorted array

A. Implementation of delete operation such that search, insert, delete takes O($\sqrt{n}$):
   - In order to delete the element, we need to find the element where it is present using SAwS data structure.
   - To find the element we first search in the sorted array.
   - We use binary search to find the element in the sorted array.
   - Binary search takes O(logn) time to search the element(n elements).
   - If the element is found then mark location of element x removed.
   - If the element is not found, then we do the iterative search in the sidekick array.
   - The iterative linear search takes O($\sqrt{n}$),( $\sqrt{n}\ elements$).
   - If the element is found then element x is mark as removed.
   - The total time complexity = O(log n) + O($\sqrt{n}$)
   - The total time complexity = O($\sqrt{n}$)

```
BinarySearch(array,element,low,high){
        if(low>high)
                return
        else{
                mid=(low+high)/2
                if(element==array[mid]){
                        //remove element at location mid
                        return
                }
                elseif(element>array[mid])
                        return BinarySearch(array,element,mid+1,high)
                else
                        return BinarySearch(array,element,low,mid-1)
        }
}
```

```
LinearSearch(array,element,size){
    for i=0 to size-1{
        if(array[i]==element){
            //remove element at location i
            return
        }
    }
```

Amortized analysis using Accounting Method:

Amortized time = Actual Time + Credits Added – Credits Removed
Actual cost of $i^{th}$ operation is $c_i$ and the amortized cost of $i^{th}$ operation is $\hat{c}_i$ ,then

$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$ for all sequences of n operations

Invariant:

Put $1 credit on each insert in sidekick Array when inserted in SAwS Array

Put $2 credit on each element in sidekick Array to Merge and Copy Operations with sorted array.

Put $1 credit on each insert in Sorted Array

Total credits for sidekick Array = 3 credits

Total credits for Sorted Array = 3 credits

The cost would not exceed logn + $3\sqrt{n}$ at any time

Claim1: Amortized running cost of insert operation in SAwS Array is O($\sqrt{n}$) after deletion too

Proof:

- Each element has 3 credits for sidekick array $3\sqrt{n}$. These credits are used to pay for future operations.
- (sidekick array) whenever merging is done when the elements exceed $\sqrt{n}$ these credits will pay for the operations such that the invariant is maintained.
- Each element has 1 credit for sorted array. These credits are used to pay for future operations of deletion such that the invariant is maintained.
- Amortized time = O(1) + logn+ $3\sqrt{n} - 2\sqrt{n}$
- Amortized time = O($\sqrt{n}\log n$)
- Amortized time = O($\sqrt{n}$)
- Therefore, the invariant is maintained.

Claim2: Amortized running cost of search operation in SAwS Array is O($\sqrt{n}$) after deletion too

Proof:

- Each element has 3 credits for sidekick array $3\sqrt{n}$. These credits are used to pay for future operations.
- Each element has 1 credit for sorted array. These credits are used to pay for future operations of deletion such that the invariant is maintained.
- The Actual cost is searching for the element in both the arrays i.e. logn + $\sqrt{n}$
  logn for sorted array and $\sqrt{n}$ for unsorted array.
- Amortized time = O(1 + logn + $\sqrt{n}$) + 0 - 0
- Amortized time = O($\sqrt{n}$)
- Therefore, the invariant is maintained for search operations.

Claim3: Amortized running cost of delete operation in SAwS Array is O($\sqrt{n}$)

Proof:

- Each element has 3 credits for sidekick array $3\sqrt{n}$. These credits are used to pay for future operations.
- Each element has 1 credit for sorted array. These credits are used to pay for future operations of deletion such that the invariant is maintained.
- The delete operation consumes credits of sorted array and sidekick array.
- The Actual cost is searching for the element to delete in both the arrays i.e. logn + $\sqrt{n}$
  logn for sorted array and $\sqrt{n}$ for unsorted array.
- Amortized time = O(logn + $\sqrt{n}$) + logn + $3\sqrt{n}$ – (logn + $\sqrt{n}$)
- Amortized time = O($logn + 3\sqrt{n}$)
- Amortized time = O($\sqrt{n}$)
- Therefore, the invariant is maintained for delete operations.
- After delete, insert and search operations are O($\sqrt{n}$ )

**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

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

06.2-Skew-Heaps-post.pdf - Google Drive

CMSC 641-01 Spring 2023 Shared Folder - Google Drive