CSCI 6057
Advanced Data Structures

ASSIGNMENT - 1

Balaji Sukumaran (bl664064@dal.ca)

Banner ID: B00948977

Table of contents

1. [10 marks] In the *clearable list* problem, we maintain a singly-linked list under the following two update operations:

   `INSERT`, which inserts an element into the list, and the newly inserted element will become the list head;

   `CLEAR`, which removes all the elements from the list.

   It is easy to see that `INSERT` can be performed in $O(1)$ worst-case time, while `CLEAR` can take $O(n)$ time in the worst case, where n is the number of operations performed so far starting from an initially empty list.

   Your task is to show that the amortized times of performing `INSERT` and `CLEAER` are both $O(1)$.

Solution:

For a clear-list problem, with two operations i.e., INSERT and CLEAR

Pseudo code:

```
INSERT(E):      //E is the new element value
        Node <- CREATE NODE(E)
        Node.next <- Head
        Head <- Node
```

It is clear that for each insert operation it takes constant time ***O(1)***

```
CLEAR(L):       //L is the list
        temp <- L.HEAD
        temp_next <- NULL
        While temp != NULL :
                temp_next <- temp.next
                deallocate(temp)
                temp<-temp_next
        L <- NULL
```

In order to deallocate every elements of the list of n elements it takes ***O(n)*** worst case complexity.

Proof:

By Aggregate Analysis:

We know that, worst case complexity of :

Clear: ***O(n)***

Insert: ***O(1)***

Based on this, If we perform **m** insert operations and **1** clear operation, the total time will be ***O(m+n).***

So, for a series of m operations the average cost per operation is

$$O\left(\frac{m+n}{m}\right) = O\left(1+\frac{n}{m}\right) \quad \text{------> (1)}$$

We know that $n \leq m$ because

n: number of insert operations before clear

m: number of operations

so, $\frac{n}{m} \leq 1$

$= O\left(1+\frac{n}{m}\right) = O(1+1) = O(2)$ ------> from (1)

$= O(1)$

After disregarding the constant, the amortized time of Insert and clear is ***O(1)***

By Accounting Method:

We assign different credits for the operations INSERT and CLEAR operations.

INSERT: **$2** is assigned where **$1** is to cover the actual cost and the other $1 to be stored as credit.

CLEAR: When the clear is called it will use the pre-paid credits for its operations. Moreover, it can only clear the number of elements inserted so we will already have enough credits.

Hence, for a series of operations the amortized cost of both the operation is ***O(1)***

By Potential Method:

$a_i = c_i + \phi(D_i) - \phi(D_{i-1})$ ------> (1)

a_i = amortized cost

c_i = actual cost

$\phi(D_i) - \phi(D_{i-1})$ = change in potential

For INSERT operation:

c_i = 1 since we are just inserting in the head of the list.

ϕ is defined by adding an element to the list which increases the potential by 1,

So,

$\phi(D_i) = \phi(D_{i-1}) + 1$ ------> (2)

By (1) and (2),

$a_i = c_i + \phi(D_i) - \phi(D_{i-1})$

$= 1 + (\phi(D_{i-1}) + 1) - \phi(D_{i-1})$

$= 2$ ------> (3)

The amortized cost remains a constant, ***O(1)***

For CLEAR operation:

Actual cost, C = O(n) - for removing n elements from the list.

$\phi(D_{i-1}) = n$ - before clearing the list the potential ϕ is n

$\phi(D_i) = 0$ - after clearing the list the potential ϕ is reduced to 0

$a_i = c_i + \phi(D_i) - \phi(D_{i-1})$

$= \mathrm{O(n)} + 0 - n$ = O(n) – n

Here the -n cancels the actual cost ***O(n)***. since we have been adding 1 for insert. These units add up to pay for the cost of clear. So, the amortized cost of INSERT and CLEAR remains ***O(1)*** over a sequence of operations.

2. [10 marks] In class we discussed the following strategy of supporting insert and delete over resizable arrays: double the array size when inserting into a full array and halve the array size when a deletion would cause the array to become less than half full. We showed that this solution would not guarantee $O(1)$ amortized time for insert or delete by constructing a counterexample. When constructing this counterexample, we assume, for simplicity, that n is a power of 2.

   Now, construct a counterexample for the general case in which n is not necessarily a power of 2. Argue why it is a counterexample.

Solution:

For resizable array:

The growing and shrinking phase of the resizable array depends on the loading factor α if it is small we should shrink the size of the resizable array and if it is 1 we should increase the size of the array.

The General Idea:

- For insertion we simply have to insert the data and if the array is full we will double the size of array (creating and new array and transferring the data to new array) and insert.
- For deletion we can remove the element but the capacity of the resizable array should be monitored and if it is half filled we can shrink the array (create an array with half the size and move all the elements to the new array)
- **Essentially, we are trying to maintain the Load factor $\alpha \leq 1/2$**

But this solution does NOT guarantee an ***O*(1)** amortized time for insert or delete.

let us consider the following counterexample:

Initial Setup:

Consider an empty array having an initial capacity of 1.

1. Perform a series of **m** insert operations where **m** is a significant number. The array will resize several times during these operations.
2. Each resizing involves copying all existing elements to a new, larger array. The resizes occur at capacities 1, 2, 4, 8, ..., up to m.
3. The number of copy operations for each resizing forms a series like 1 + 2 + 4 + 8 + ... + m/2 + m. Takes ***O*(m)**.

Toggling Insert and Delete Operations:

After the initial ***m*** inserts, we perform a series of ***k*** cycles where each cycle consists of ***j*** insert operations followed by ***j*** delete operations.

Here, please note that for every insert in the cycle causes the array to resize (double its size), there will eventually be a delete operation that could potentially cause the array to resize (halve its size) if the resizing strategy is to shrink when the array is less than half full.

During the toggling,

- ***Insert***: a single resize will involve copying m, m+1, m+2, ..., m+j-1 elements to a new array, assuming **j** inserts cause **j** resizes, which is worst-case.
- ***Delete***: the array will shrink, involving copying m/2, m/4, ..., elements back to a smaller array. Again, this is the worst-case scenario assuming each delete after the threshold leads to a resize.

Overall Time Complexity

- The initial series of **m** insert operations contributes ***O*(m)** complexity.
- Each cycle of **j** insert operations followed by **j** delete operations contributes to the complexity. This toggling can approach ***O*(j^2)** complexity per cycle due to the potential resizing on each operation.
- If we perform **k** such cycles, and **j** is proportional to **m** where **j** ≈ **m**, then the complexity of the **k** cycles is **k** times ***O*(j^2)**, which can approach ***O*(k X m^2)**.
- Combining this with the initial ***O*(m)** operations, and if **k** is proportional to m (i.e., k ≈ m, the total complexity for all operations can approach ***O*(m^3)**, which is can be worse than ***O*(n^2)**.

3. [15 marks] In class we saw the example of incrementing an initially 0 binary counter. Now, suppose that the counter is expressed as a base-3 number. That is, it has digits from $\{0, 1, 2\}$.

(i) [5 marks] Write down the pseudocode for `INCREMENT` for this counter. The running time should be proportional to the number of digits that this algorithm changes.

Solution:

Pseudo code

A is an array storing the base-3 counter with bits from right to left in a such a way the least significant bit on the left and most significant bit on the right

```
INCREMENT(A[0.....k-1]):
    i <- 0
    while  i < k and A[i] =2:
        A[i] <- 0
        i <- i+1
    if i < k :
        A[i] <- A[i] + 1
```

The worst case time complexity of the algorithm is O(min(c,k)) where c <= k and c is the number of bits changed by the algorithm

Now, Consider numbers from 0 till 18

Base-10	Base-3
0	000
1	001
2	002
3	010
4	011
5	012
6	020
7	021
8	022
9	100
10	101
11	102
12	110
13	111
14	112
15	120
16	121
17	122
18	200

- The right most bit is flipped every time - 3^0
- The second right most bit is flipped after 3rd increment - 3^1
- The third right most bit is flipped after 9th increment - 3^2

For i = 0,1,… bit A[i] flips $\frac{n}{3^i}$ times in a sequence of n increments.

The summation would look like,

= n + n/3 + n/9 + n/27

= n + n/3^1 + n/3^2 + n/3^3…..

= $\frac{n}{1-\frac{1}{3}}$ by, S = a+ar+ar^2 …. S= a/(1-r)

= $\frac{3}{2}n$

Aggregate Analysis:

Assuming that it takes constant time to flip a single bit, the total cost O(n). for a series of n operations, the amortized cost per operation is O(n)/n = O(1)

(ii) [5 marks] Define the actual cost of `INCREMENT` to be the exact number of digits changed during the execution of this algorithm. Let $C(n)$ denote the total cost of calling `INCREMENT` n times over an initially 0 base-3 counter. Use amortized analysis to show that $C(n) \leq (3/2)n$ for any positive integer n. To simplify your work, assume that the counter will not overflow during this process.

Note: I will give the potential function as a hint after the problem statement, though you will learn more if you try to come up with your own potential function.

Hint: Let D_i be the counter after the i-th `INCREMENT`, and Let $|D_i|_d$ be the number of digit d in D_i for $d \in \{0, 1, 2\}$. Define $\Phi(D_i) = |D_i|_1/2 + |D_i|_2$.

Solution:

Let,

D_i – be the counter after the 'i' increment.

$|D_i|_d$ – be the total number of digits d in D_i for d in {0,1,2}

b_i – number of bit 2 in D_i

Observations: For finding the potential function, lets observe our algorithm

1. it basically flips all the bit 2s to 0s on the right and transition the first non 2-bit {0,1} to 0 -> 1 and 1 -> 2.
2. All the other bits remain static, and it doesn't have to account for when we define the potential function.
3. From Hint, $\phi(D_i) = \frac{|D_i|_1}{2} + |D_i|_2$

Scenario 1: All the 2s on the right are flipped to 0 and first non 2-bit {0,1} is 0 and it is transitioned to 1.

{0 | 1 | 2}, {0 | 1 | 2}, {0 | 1 | 2} …, 0, 2,2,2,2…

Increments to

{0 | 1 | 2}, {0 | 1 | 2}, {0 | 1 | 2} …, 1, 0,0,0,0…

So, for the current potential $\phi(D_i)$ the previous potential $\phi(D_{i-1})$ will be reduced by number of 2s [$-\ b_i$] and the potential is increased by ½ accounting for switching 0 to 1.

$$\phi(D_i) = \phi(D_{i-1}) - b_i + \frac{1}{2}$$

Scenario 2: All the 2s on the right are flipped to 0 and first non 2-bit {0,1} is 1 and it is transitioned to 2.

{0 | 1 | 2}, {0 | 1 | 2}, {0 | 1 | 2} …, 1, 2,2,2,2…

Increments to

{0 | 1 | 2}, {0 | 1 | 2}, {0 | 1 | 2} …, 2, 0,0,0,0…

So, for the current potential $\phi(D_i)$ the previous potential $\phi(D_{i-1})$ will be reduced by number of 2s $[-\ b_i]$ and the potential is increased by 1 accounting for switching 1 to 2. But since we already had bit 1 we can reduce the potential by ½.

$$\phi(D_i) = \phi(D_{i-1}) - b_i + 1 - \frac{1}{2}$$

$$\phi(D_i) = \phi(D_{i-1}) - b_i - \frac{1}{2}$$

Regardless of the scenario we know the potential will be the following:

$$\phi(D_i) = \phi(D_{i-1}) - b_i - \frac{1}{2} \quad \text{---------- (1)}$$

Actual cost $c_i = b_i + 1$ ---------- (2)

Potential Method:

$a_i = c_i + \phi(D_i) - \phi(D_{i-1})$

$= b_i + 1 + \phi(D_{i-1}) - b_i - \frac{1}{2} - \phi(D_{i-1})$ ---------- After applying (1) and (2)

$= 3/2$ ---------- (3)

Amortization:

$$\sum_{i=0}^{n} a_i = \sum_{i=0}^{n} c_i + \sum_{i=0}^{n} (\phi(D_i) - \phi(D_{i-1}))$$

$$= \sum_{i=0}^{n} c_i + \phi(D_i) - \phi(D_0)$$

We know that $\phi(D_0) = 0$

$$\sum_{i=0}^{n} c_i = \sum_{i=0}^{n} a_i - \phi(D_i)$$

From (3), and as we know $\phi(D_i)$ potential is always positive,

$$C(n) \leq \frac{3}{2}n$$

(iii) [5 marks] Prove that for any c such that $C(n) \leq cn$ for any positive integer n, the inequality $c \geq 3/2$ holds. Again, assume that the counter will not overflow during this process.

Solution:

We have to prove that for any c, in C(n) ≤ cn for any positive integer n, such that c ≥ $\frac{3}{2}$ holds and lets consider constant 'a' a small positive integer a>0. Where c – a -------- (1) will NOT satisfy in the above equation.

We can prove this by assuming in *cn*, $c = \frac{3}{2} - a.$ --------(2)

Such that $C(n) > (\frac{3}{2} - a)n$ holds true.

For base-3 Counter we know that:

- The right most bit is flipped every time - 3^0
- The second right most bit is flipped after 3rd increment - 3^1
- The third right most bit is flipped after 9th increment - 3^2
- And so on.

C(n) = $\sum_{k=0}^{\partial} 3^k$

Where $\partial = \lceil -\log_3 a \rceil$

$-\partial \leq \log_3 a$

$3^{-\partial} \leq a$ ---------- (3)

C(n) = $\frac{3^{\partial+1}-1}{2}$

C(n) ≤ $3^{\partial+1} - 1$ ---------- (4)

In (3) in (1)

(c - a)n ≤ (c - $3^{-\partial}$)n

($\frac{3}{2} - a$)n ≤ ($\frac{3}{2}$ - $3^{-\partial}$)n

For n = 3^{∂}

$\leq (\frac{3}{2} - 3^{-\partial})\ 3^{-\partial} = \frac{3^{\partial+1}}{2} - 1$

$\leq\ 3^{\partial+1}$ -1 = C(n)

Hence (C - a)n $\leq$ C(n) or C(n) > $(\frac{3}{2} - a)$n

4. [15 marks] Let the binary representation of a positive integer n be $n_{k-1}n_{k-2}\cdots n_0$. Thus $k = \lceil \lg(n+1) \rceil$.

   Suppose that we maintain a data structure that is a collection of arrays (NOT resizable arrays) $A_0, A_1, \cdots, A_{k-1}$ that store n integers in total.

   The following three invariants hold for this collection:

   (a) The elements in each A_i are sorted in ascending order;

   (b) The size of each A_i is 2^i, i.e., the maximum number of integers that can be stored in A_i is 2^i;

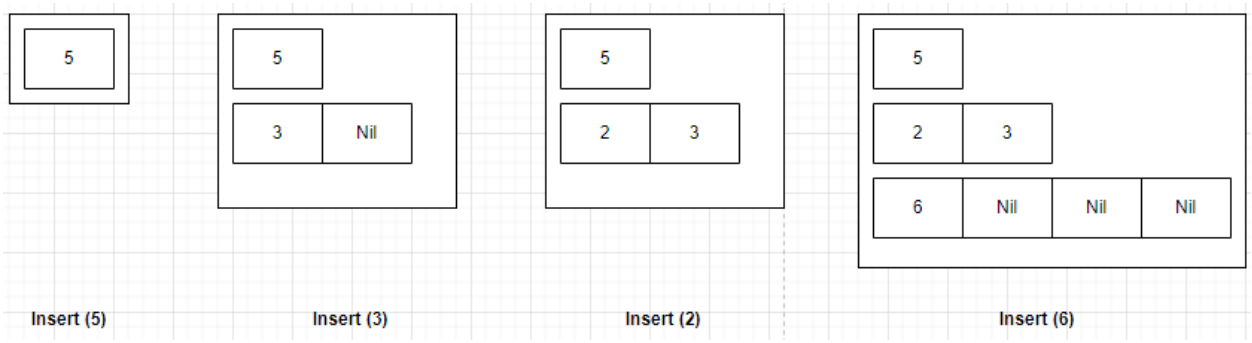
   (c) Each A_i is either full or empty: if n_i is 0, then A_i is empty; otherwise A_i is full.

   Although each array is sorted, we cannot make any assumptions on the relationship between integers in different arrays. To simplify your work, you can assume that integers stored in this data structure are distinct, even under insertion and deletion (it is easy to handle duplicates).

Solution:

Pre-Work: Based on the description given lets us try and visualize the data structure initially it is empty

Now, let's try to insert a series of integers, 5,3,2,6



In order to hold the invariant rule (c) [Ai is either full or empty], when creating a new array we will initialize some placeholder for all the cells such as INT_VALUE in diagram it is Nil.

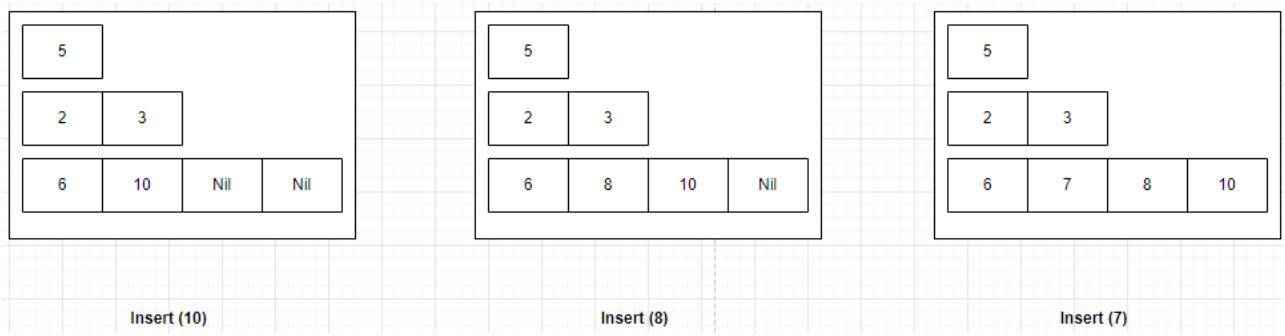
Observations:

<u>For Insertion:</u>

1. When inserting we have to navigate to the last element in the last array. Refer, insert (3) and insert (2)

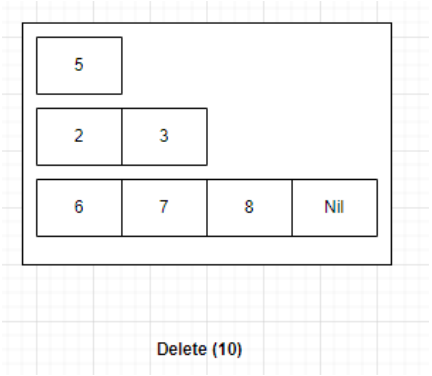
2. If this array has space we have can insert in the correct location. Without violating the sorting invariant rule (a)
3. If the last array is full create a new array of size 2^{k+1} with INT_VALUE as initial value for all elements and insert the data in the array's first location. Refer insert (6)

Let's insert few more elements. 10, 8, 7

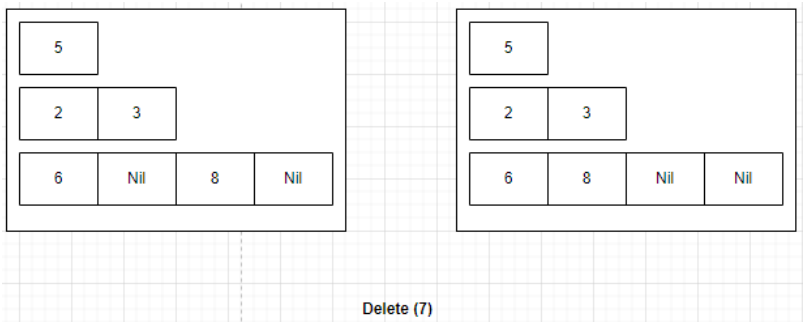


<u>For Deletion:</u>

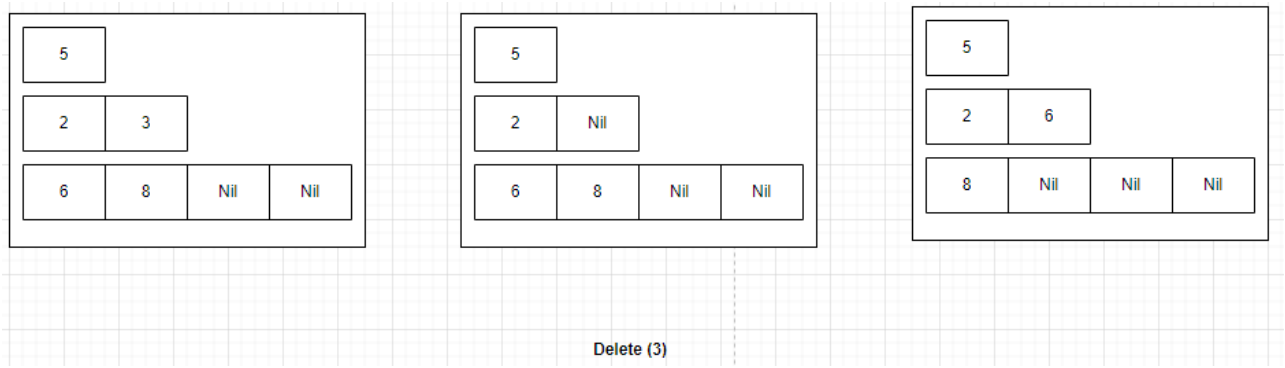
Case 1: if we have to delete the last element of the last array just assign NIL, deleting 10



Case 2: If we have to delete the intermediate element in last array we just have to left shift all the elements from right. Deletion (7)



Case 3: If we have to delete any other element apart from last array we just have to left shift all the elements from right and below and sort the current array. Here after deleting (3) the A1 should have been sorted



(i) [5 marks] Describe an $O(\lg^2 n)$-time algorithm that, given an integer value x, find out whether it is one of the integers stored in this data structure. Show your analysis of the running time. You are not required to give pseudocode, but feel free to give pseudocode if it helps you explain your algorithm.

Solution:

Search Operation

Please note that:

1. Each array within our data structure is sorted.
2. Elements in the different arrays are not sorted and are in no relation to each other, except they will be moved to fill holes when deletion

Algorithm Description:

Perform Binary Search on every array, leveraging the sorted nature of each array.

Time Complexity:

This takes **O(log n)** per array.

As there are k = [log (n+1)] arrays, we need to perform binary search k times.

The total time complexity becomes

= O(k X log n)

= O(log n X log n)

= O($log^2 n$)

(ii) [5 marks] Describe an algorithm that supports the insertion of an integer x into this data structure, such that the three invariants hold after the insertion. In most cases, your algorithm will move elements between arrays and will even allocate one more array if k increases. Your algorithm is expected to be efficient in the amortized sense. Show your analysis. Again, pseudocode is not required but feel free to give it.

Hint: At some stage, your algorithm might merge two sorted sequences.

Algorithm Description:

Because of the invariant rule, all the arrays will either be full or empty. So there will be a cascading effect when we try to insert the element into a particular array A^i.

Pseudocode:

```
Insert(x)
  Let k be the number of arrays in the structure
  for i = 0 to k do
    if A^i is not full
      Perform a binary search in A^i to find the correct position for x
      Insert x into A^i at the found position
      Exit the loop
    else
      Find the largest element in A^i, let it be y
      If x < y
        Replace y with x in A^i
        x = y
      Move to the next array
  If all arrays A^0 to A^(k-1) are full
    Create a new array A^k of size 2^k
    Insert x into A^k
  Rebalance the arrays if necessary to maintain all invariants
End
```

Time Complexity:

1. If the array is half empty, this operation takes O(log(2^i)) = O(i) time for an array A^i, which is efficient.
2. Inserting an element into an array of size 2^i takes O(2^i) time in the worst case if you need to shift elements.
3. In the worst case, the element x may cascade through all k arrays if each is full. However, this is a rare event. The cascading move operates in a fashion similar to the increment of a binary counter. For a binary counter, incrementing it N times takes O(N) time, even though some increments take O(k) time when all lower bits flip from 1 to 0. But since these costly flips happen increasingly rarely, the average cost per increment is O(1).
4. Allocating a new array happens when all current arrays are full. This is similar to doubling the size of a dynamic array when it gets full, which has an amortized cost of O(1) per insertion.

Considering all these points the amortized time complexity of the insertion is O(1), even though some individual insertions might take longer due to cascading.

(iii) [5 marks] Finally, show how to support deletion efficiently in the amortized sense and analyze your algorithm. That is, given an integer x, if it can be found as an element of an array in this data structure, remove it. Note that you need to compact the arrays (you cannot leave "holes" between elements) and also maintain the invariants.

Hint: The running time of deletion would be more than that of insertion. Do your best.

Solution:

Algorithm Description:

Step 1. For each array A^i from A^0 to A^{k-1}:

Use binary search to look for x in A^i. This takes O(log(2^i)) time.

If x is found in A^i:

Remove x from A^i, creating a "hole."

Compact A^i to eliminate the hole. This step requires shifting elements, which takes O(2^i) time in the worst case.

To maintain the invariants, check if the removal of x causes A^i to be underfull:

If A^i is now underfull and i != 0, borrow the largest element from A^{i-1} to fill the gap. This may cause A^{i-1} to become underfull, thus recursively applying this borrowing logic up to A^0.

If A^i is A^0 or borrowing is not possible because all previous arrays are empty, no further action is required for compaction.

Exit the loop since x has been deleted.

Step 2. If no more elements remain in the highest array A^{k-1} after deletion and compaction, deallocate A^{k-1}.

Time Complexity:

The deletion algorithm's amortized analysis takes into account the following:

1. Removing
2. Compacting,
3. Maintaining invariants across a sequence of operations.

1. **Finding and Removing x:**

Binary search to find the element takes O($log^2 n$) as proved in the problem (i)

Removing x and compacting the array has a worst-case cost of O(2^i) for array A^i, which directly affects the array where x is found.

2. **Maintaining Invariants:**

Borrowing an element from A^{i-1} to fill a gap in A^i might trigger a cascade of borrowings if A^{i-1} also becomes underfull. However, this cascade is not as costly as the cascading effect in insertions because it does not necessarily involve all arrays.

3. Deallocating Empty Arrays:

If the highest array A^{k-1} becomes empty after deletion, it is deallocated. This operation is infrequent and can be considered O(1) over a sequence of deletions.

The amortized time complexity of deletion is higher than that of insertion, primarily due to the additional steps required to maintain the invariants without leaving "holes."