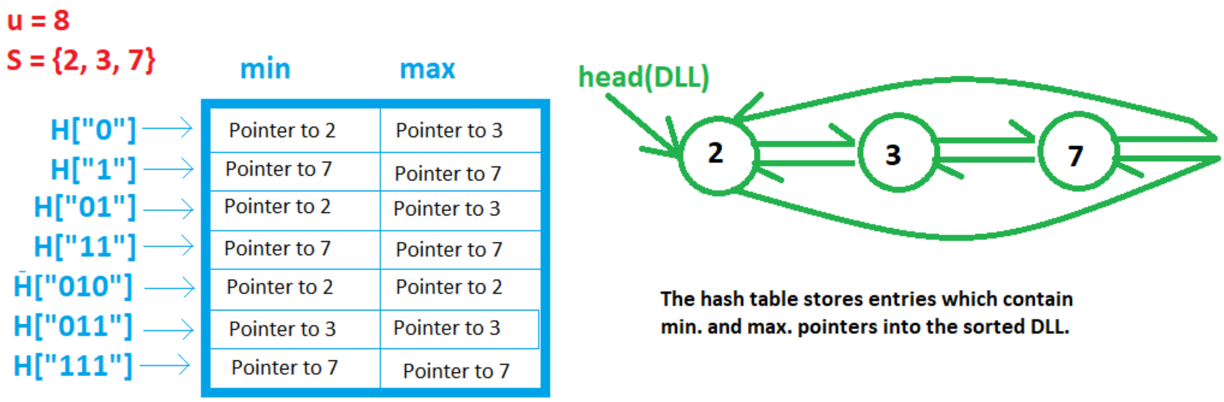
DESIGN AND ANALYSIS OF ALGORITHMS
Homework 2

Aman Choudhary
MTech Coursework, CSA 2020
Sr No: 17920

October 21, 2020

1 Space Complexity

Our assumption of the entire data structure.



There are n elements stored in the sorted doubly linked list (DLL).
Hence, the space occupied by the DLL = $O(n)$

When n is small,
The size of hash table $H = O(n \log u)$
Total Space Complexity = $O(n) + O(n \log u) = O(n \log u)$.

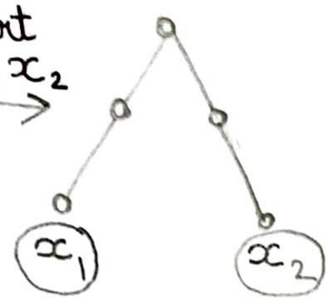
When n approaches u,
The size of hash table $H = O(u)$
Total Space Complexity = $O(n) + O(u) = O(u)$.

We give the reasoning behind the size of hash table H, in next page.

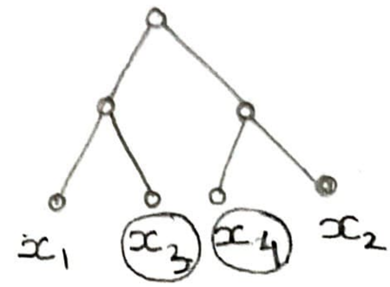
We want to look at what could be the maximum number of prefixes in the hash table at any point, when $|S| = n$.

Case: $u = 4$, $\log_2 u = 2$

Insert x_1, x_2 →



Insert x_3, x_4 →



2 ($\log_2 u$) prefixes inserted per element

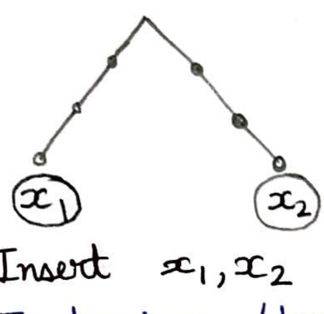
1 prefixes ($\log_2 u - 1$) inserted per element

	x_1	x_2	x_3	x_4
Prefixes / element →	2	2	1	1
Size of H after insertion →	2	4	5	6

$n=0$ $n=2$ $n=4$

We claim that only when elements are inserted in this manner, maximum no. of prefixes get pushed in H at every step.

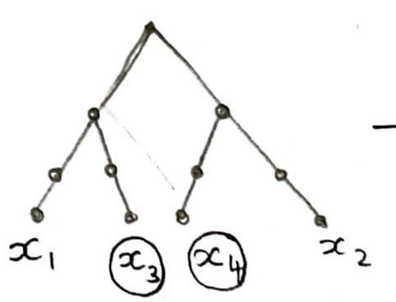
Case: $u = 8$, $\log_2 u = 3$



Insert x_1, x_2

3 prefixes ($\log_2 u$) inserted per element

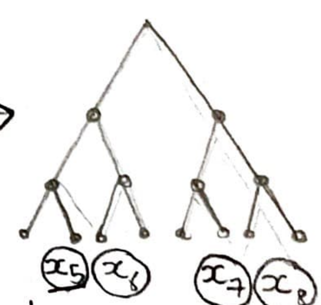
→



Insert x_3, x_4

2 prefixes ($\log_2 u - 1$) inserted per element

→



Insert x_5, x_6 x_7, x_8

1 prefix ($\log_2 u - 2$) inserted per element.

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
Prefixes / element →	3	3	2	2	1	1	1	1
Size of H after insertion →	3	6	8	10	11	12	13	14

$n=0$ $n=2$ $n=4$ $n=8$

We can show that if $n = 2^k$,
then the no. of prefixes in hash table is,

$$(\log u)(2) + (\log u - 1)(2^1) + (\log u - 2)(2^2) + (\log u - 3)(2^3) + \cdots + (\log u - (k-1))(2^{k-1}) =$$

$$= \log u\,(2) + \log u\,(2^1 + 2^2 + 2^3 + \cdots + 2^{k-1}) - (1.2^1 + 2.2^2 + 3.2^3 + \cdots + (k-1).2^{k-1})$$

$$= (2\log u) + \log u\left[\frac{2(2^{k-1}-1)}{2-1}\right] - \left[\sum_{i=1}^{k-1} i.2^i\right]$$

$$= (2\log u) + \log u\,(2^k - 2) - \left[\sum_{i=1}^{k-1} i.2^i\right] \quad \hookrightarrow S'$$

$$= (2^k \log u) - S' \quad \text{——} \; (1)$$

$$S' = 1.x^1 + 2x^2 + \cdots + nx^n$$

$$xS' = 1.x^2 + \cdots + (n-1)x^n + n.x^{n+1}$$

$$S'(1-x) = (x^1 + x^2 + \cdots + x^n) - n.x^{n+1}$$

$$S' = \frac{1}{(1-x)}\left[\left(\frac{x(x^n-1)}{x-1}\right) - n.x^{n+1}\right]$$

Putting, $x = 2$, $n = k-1$

$$S' = \sum_{i=1}^{k-1} i.2^i = 2^k(k-2) + 2 \quad \text{——} \; (2)$$

Put ② in ①,

Size of $H = (2^k \log u) - (2^k(k-2)+2)$

$$\boxed{\text{Size of } H = 2^k(\log u - k + 2) - 2} \quad \text{—— ③}$$

For $n = 2^k$, we calculated the above result. So, we can safely say for $n \leq 2^k$, size of H will be less than given by eq^n ③,

Now, $n = 2^k$

$k = \log_2 n$

$$\boxed{\text{Size of } H = n(\log u - \log n + 2) - 2}$$

Case: when 'n' is small $(n \leq 2)$

In this case $\log u$ dominates the result, and the size of $H \approx n \log u$, which is intitutive result we get that for every element, all $\log_2 u$ prefixes are being inserted. But in reality all prefixes are inserted only for first 2 elements.

Case: when 'n' is closer to 'u'

As n approaches u, log u and log n get subtracted and $(\log u - \log n)$ approaches 0.

So, $\boxed{\text{Size of } H = 2u - 2 = O(u)}$ when n reaches closer to u.

2 Insertion

2.1 Insertion into DLL

$//head(DLL)$ points to first node of DLL.
$//x$ is the element to be inserted.
$//X$ is the pointer returned to x after insertion into DLL.

$X = Insert(head(DLL), x)$

1. If $head(DLL) == NULL$, this means that the DLL is empty. We insert the new element x right away into DLL. This step takes $O(1)$ time.
2. Otherwise, the DLL is not empty. We need to locate exactly where x must be inserted. We call $P = predecessor(x)$, which returns a pointer P to the predecessor of x in DLL. This step takes $O(\log\log u)$ time (shown later).
   - 2.a If $P == NULL$, this means that x is less than every element in S. So, we insert x at the beginning of DLL. This step takes $O(1)$ time.
   - 2.b Otherwise, we insert x, next to the element pointed to by P. This step takes $O(1)$ time.
3. We return a pointer X to the newly inserted element x.

Time Complexity of Insertion into DLL = $O(\log\log u)$

2.2 Insertion into Hash Table

$//H$ is the Hash Table.
$//X$ points to newly inserted element x.

$Insert(H, X)$

1. We compute the set A which contains all the prefixes of binary representation of x. For simplicity, we assume all string operations required to compute a single prefix takes $O(1)$ time. Therefore, this step takes a total of $O(\log u)$ time.
2. For every prefix $a \in A$, we do the following:
   - 2.a If the hash table H does not contain an entry for a, we set:
     $H[a].min = X$
     $H[a].max = X$
   - 2.b If the hash table H already contains an entry for a, we modify the entry as:
     $if(\quad X->val \quad < \quad (H[a].min)->val \quad)$
     $\qquad H[a].min = X$
     $if(\quad X->val \quad > \quad (H[a].max)->val \quad)$
     $\qquad H[a].max = X$

   Step 2 takes $O(1)$ time for every prefix $a \in A$. Since, there are $\log_2 u$ prefixes in A, the total time for Step 2 is $O(\log u)$.

Time Complexity of Insertion into Hash Table = $O(\log u)$

Total Time Complexity = $O(\log\log u) + O(\log u) = O(\log u)$.

3 Longest Common Prefix

1. We know that every element in U in binary representation requires $O(\log u)$ bits. In order to find the longest common prefix of x, we can perform binary search on the range: 1 to $\log_2 u$, i.e. the possible lengths of the prefixes.

2. The idea is that if the hash table H contains a prefix of x of length l, then we can say:

   a) H also contains all prefixes of x with length $< l$. We don't need to worry about them, so we completely discard that range of lower prefix lengths from our search.

   b) Moreover, there is a possibility that we could find a prefix of x with length $> l$. If that is so, this prefix of length l was derived from it in the first place. Hence, we pursue for greater prefix lengths.

3. On the other hand, if the hash table H does not contain any prefix of x of length l, then we can say:

   a) H cannot contain any prefixes of x with length $> l$. Because, if that would have been the case, then a prefix of length l would have existed in H. Hence, we completely discard that range of greater prefix lengths from our search.

   b) Although we couldn't find a prefix of length l, there is a possibility that we may still find a prefix of x with length $< l$. Hence, we proceed in search of lesser prefix lengths.

4. We know that binary search on k elements, takes $O(\log k)$ time. Hence for searching on prefix lengths from range 1 to $\log u$, **we will need to query the hash table $O(\log \log u)$ times**.

//x is the element whose LCP we wish to compute.
//lcp is the longest common prefix string of x returned by algorithm.

$lcp = LCP(x)$

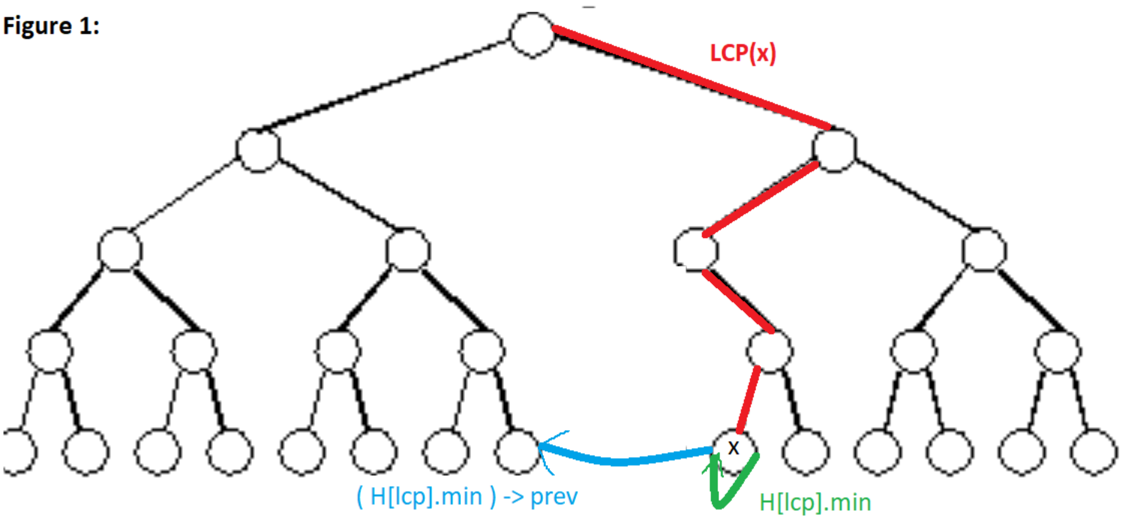
```
s = binary(x)                      //binary string representation of x, computed in O(1) time.
low = 1
high = log_2 u
while(low < high){
        mid = (low + high)/2
        if(H[s(1 : mid)] == NULL)  //s(1:mid) represents the sub-string of s from index 1 to mid.
                high = mid - 1
        else
                low = mid + 1
}
return s(1 : mid)
```

4 Predecessor

1. **Case 1: $x \in S$, and hence $binary(x)$ is present in hash table H.**

   In this case, $lcp = binary(x)$. Hence, from the hash table entry $H[lcp].min$ or $H[lcp].max$, we can reach to node x in sorted DLL. Then, we can simply find the predecessor of x, using the $prev$ pointer of node x.

Figure 1:

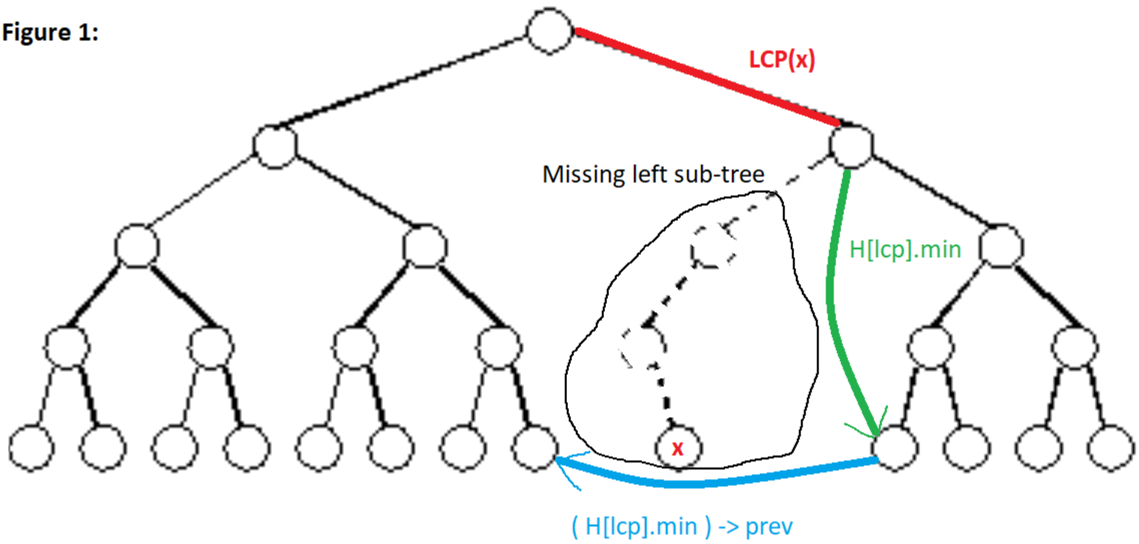


2. **Case 2: $x \notin S$, and hence $binary(x)$ is absent in hash table H.**

   In this case, $lcp = LCP(x)$ is a proper prefix of $binary(x)$. Then, lcp denotes the farthest point in the path, which we can reach, when searching for x. Beyond this point, the path to x in the trie (conceptual) is not present (shown as broken path in figure). This claim is true, because if that was not the case, then we would have got a longer path and hence a larger prefix.

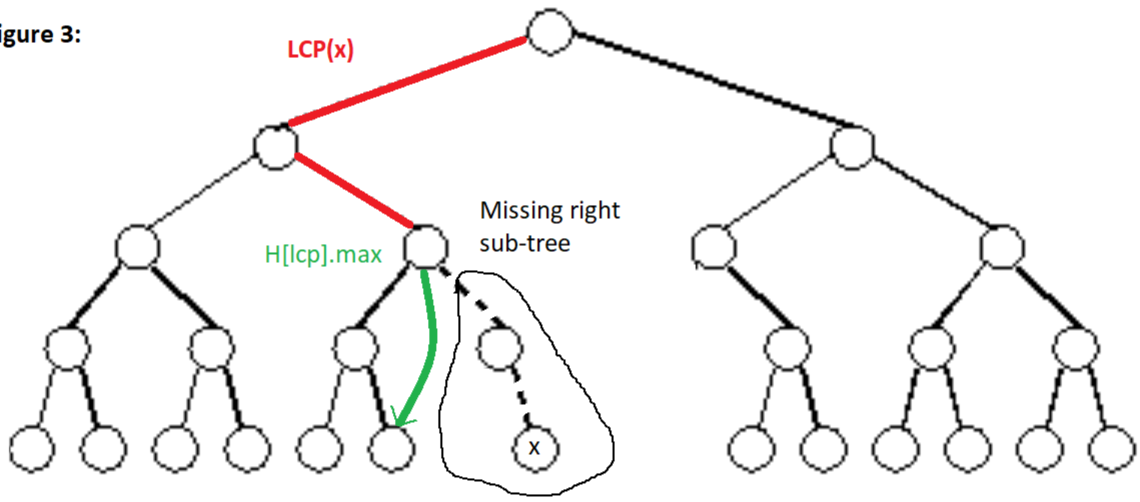
   (a) When x should have lied in the left sub-tree (if it were present in tree), we can see that the entire left sub-tree of $LCP(x)$ is missing. So, we can find the successor of x by querying the hash table for the minimum element of the right sub-tree of $LCP(x)$. Now, the element stored just before this successor in the sorted DLL, would be the predecessor of x.

Figure 1:



(b) When x should have lied in the right sub-tree (if it were present in tree), we can see that the entire right sub-tree of $LCP(x)$ is missing. So, we can find the predecessor of x by querying the hash table for the maximum element of the left sub-tree of $LCP(x)$.

Figure 3:



$//x$ is the element whose predecessor is being computed.
$//P$ is the pointer to the predecessor of x returned by the algorithm.

$P = Predecessor(x)$

1. $s = binary(x)$ //binary string representation of x
2. $lcp = LCP(x)$ //longest common prefix string of x
3. $if(lcp == s)$ //**Case 1**
4. return $(H[lcp].min)->prev$
5. $else\ if(x < (H[lcp].min)->val)$ //**Case 2(a)**
6. return $(H[lcp].min)->prev$
7. $else$ //**Case 2(b)**
8. return $H[lcp].max$

We are assuming all string operations like conversion of x into its binary representation, and string comparisons take constant time, $O(1)$. Therefore, every step in the above algorithm takes $O(1)$ time, except for step 2, where we compute $LCP(x)$. Step 2 takes $O(\log\log u)$ time.
Hence the overall time complexity of finding predecessor is $O(\log\log u)$.

NOTES

1. Ideas and different approaches pertaining to solving of problems were discussed with:
   Shashank Singh (M.Tech Coursework, CSA 2020).
2. CLRS 3rd edition was extensively used as reference material.