

Data Structures and Algorithms

Part 2

Iñaki Lakunza

April 3, 2024

Contents

1 Fenwick Tree (Binary Index Tree)	2
1.1 Introduction	2
1.2 Fenwick Tree Complexity	3
1.3 Fenwick Tree Range Queries	3
1.4 Range Query Algorithm	4
1.5 Fenwick Tree Point Updates	5
1.6 Fenwick Tree Construction	6
2 Suffix Arrays	8
2.1 Introduction	8
2.2 The Longest Common Prefix (LCP) array	8
2.3 Using SA/LCP array to find unique substrings	9
2.4 Longest Common Substring (LCS)	10
2.4.1 LCS example	12
2.5 Longest Repeated Substring (LRS)	14
3 Balanced Binary Search Trees (BBSTs)	15
3.1 Introduction	15
3.2 Complexity	15
3.3 Tree Rotations	15
3.4 Inserting Elements into an AVL tree	17
3.5 Node insertion in AVL trees	19
3.6 Removing Elements from an AVL Tree	21
3.7 Augmenting BST Removal Algorithm for AVL Tree	22

1 Fenwick Tree (Binary Index Tree)

1.1 Introduction

Given an array of integer values compute the range sum between index $[i, j]$.

0	1	2	3	4	5	6	7	8	9
A = 5	-3	6	1	0	-4	11	6	2	7

Sum of A from $[2, 7] = 6 + 1 + 0 + -4 + 11 = 14$

Sum of A from $[0, 4) = 5 + -3 + 6 + 1 = 9$

Sum of A from $[7, 8) = 6 = 6$

Given an array of integer values compute the range sum between index $[i, j]$.

0	1	2	3	4	5	6	7	8	9
A = 5	-3	6	1	0	-4	11	6	2	7

P =	\emptyset									
-----	-------------	-------------	-------------	-------------	-------------	-------------	-------------	-------------	-------------	-------------

Let P be an array containing all the **prefix sums** of A .

Given an array of integer values compute the range sum between index $[i, j]$.

0	1	2	3	4	5	6	7	8	9
A = 5	-3	6	1	0	-4	11	6	2	7

P =	0	5	2	8	9	9	5	16	22	24	31
-----	---	---	---	---	---	---	---	----	----	----	----

Sum of A from $[2, 7) = P[7] - P[2] = 16 - 2 = 14$

Sum of A from $[0, 4) = P[4] - P[0] = 9 - 0 = 9$

In this example, for instance, we want to query a range and find the sum of that range. One thing we can do would be to start at the position and scan up to where we want to stop, and then sum all the individual values in that range.

That is fine, we can do this, but, however, this will soon get very slow, since we are doing linear queries.

However, if we do something like compute all prefix sums for the array A , we can do our queries in constant time.

So, if we set the index 0 of our array P to be zero, and then we go in our array and add that element to the current prefix sum, we get five and then five plus minus three gives two, and so on.

So, this is an elementary form of dynamic programming, since we are calculating out prefix sums.

And then, if we want to find the sum, we just can get the difference between those two indexes in the P array. **Which requires a constant time**

However, there is a slight flaw in this: when we want to update a value in our original array A , for instance we want to update index 4, inserting the value 3. We will have to recompute all those prefix sums, which will make us go linear time. So, this is why the Fenwick Tree was created.

What is a Fenwick Tree

A **Fenwick Tree** (also called Binary Indexed Tree), is a data structure that supports sum range queries as well as setting values in a static array and getting the value of the prefix sum up some index efficiently.

Construction	O(n)
Point Update	O(log(n))
Range Sum	O(log(n))
Range Update	O(log(n))
Adding Index	N/A
Removing Index	N/A

Figure 1:

1.2 Fenwick Tree Complexity

Even though its construction requires linear time, points and updates are logarithmic, range sum queries are also logarithmic. But we can add or remove elements from the array

1.3 Fenwick Tree Range Queries

```

16 100002
15 011112
14 011102
13 011012
12 011002
11 010112
10 010102
 9 010012
 8 010002
 7 001112
 6 001102
 5 001012
 4 001002
 3 000112
 2 000102
 1 000012

Unlike a regular array, in
a Fenwick tree a specific
cell is responsible for
other cells as well.

The position of the least
significant bit (LSB)
determines the range of
responsibility that cell has
to the cells below itself.

Index 12 in binary is: 11002
LSB is at position 3, so this
index is responsible for  $2^{3-1} = 4$ 
cells below itself.

```

Unlike a regular array, in a Fenwick Tree a specific cell is responsible for other cells as well. Important to note that Fenwick Trees are indexed using binary values

The position of the **least significant bit (LSB)** determines the range of responsibility that cell has to the cells below itself.

Similarly for 10, for instance, its binary representation is 1010_2 , its least significant bit is at position 2, so this cell is responsible for two cells below itself. 11's least significant bit (1011_2) is at position 1, so this cell is only responsible for 1 cell below

16	10000 ₂
15	01111 ₂
14	01110 ₂
13	01101 ₂
12	01100 ₂
11	01011 ₂
10	01010 ₂
9	01001 ₂
8	01000 ₂
7	00111 ₂
6	00110 ₂
5	00101 ₂
4	00100 ₂
3	00011 ₂
2	00010 ₂
1	00001 ₂

So, due to this arrangement, **all odd numbers have their first least significant bit set in the ones position, to they are only responsible for themselves.** And depending on the value of the others they will get more or less responsibility, as shown in the image on the left.

In a Fenwick tree we may compute the **prefix sum** up to a certain index, which ultimately lets us perform range sum queries.

Idea: Suppose we want to find the prefix sum of $[1, i]$, them we **start at i and cascade downwards** until we reach zero adding the value at each of the indices we encounter.

For example, if we want to find the prefix sum up to index 7, if we look at index 7, we get the value at that position, and then we want to cascade downwards. So the next one below is 6 and the next one below is 4 (**we are going downwards using the responsibilities**).

So, this way the prefix sum up to index 7 (including it), is the value at index 7, the value at index 6 and the value at index 4.

If we want to compute the index sum up to (and including) 11, for instance, we would have to get the values at indexes 11, 10 and 8.

On the other hand, if we want to do, for instance, the index sum between (and including) indexes 11 and 15, first, we would compute the prefix sum of $[1, 15]$, the we would compute the prefix sum of $[1, 11]$, and then we would get the difference. So, for the first step we would need the values at indexes 15, 14, 12 and 8. And for the second step we would need the values at indexes 10 and 8. And then we would subtract the first one with the second.

Notice that in the worst case the cell we are querying has a binary representation of all ones (numbers of the form $2^n - 1$). Hence, it is easy to see that in the worst case a range query might make us have to do two queries that cost $\log_2(n)$ operations.

1.4 Range Query Algorithm

To do a range query from $[i, j]$, both inclusive, in a Fenwick tree of size N :

```
function prefixSum(i);
    sum := 0
```

```

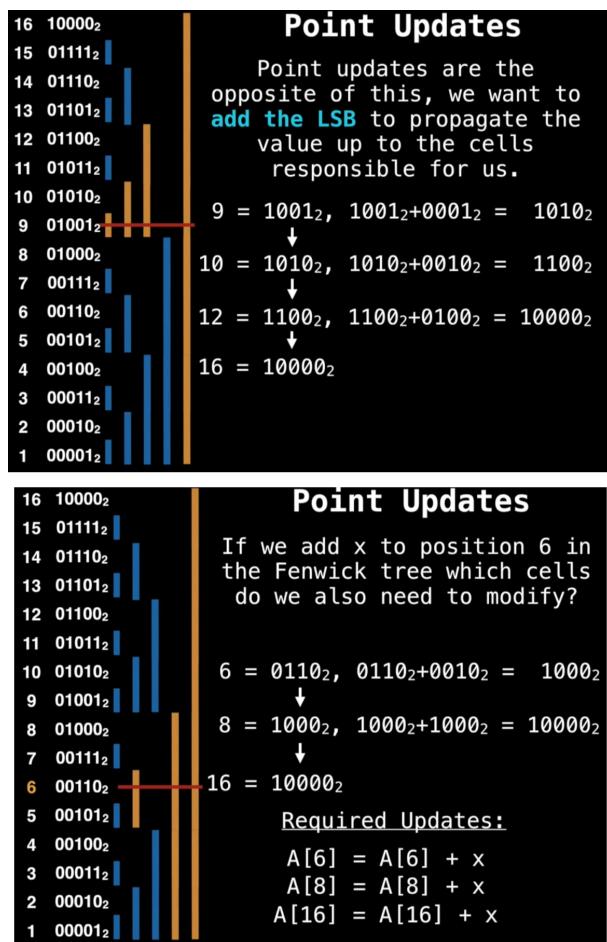
while i != 0:
    sum = sum + tree[i]
    i = i - LSB(i)
return sum

function rangeQuery(i, j):
    return prefixSum(j) - prefixSum(i-1)

```

Where **LSB** return the value of the least significant bit.

1.5 Fenwick Tree Point Updates



This way, it is like drawing a horizontal line at the index that we want, and we see which are the responsible elements for that index, as it can be seen in the image at the right.

So, this way we know which are the cells we have to modify in each case. The algorithm looks the following way:

```

function add (i, x):
    while i < N:
        tree[i] = tree[i] + x
        i = i + LSB(i)

```

When doing range queries, if we started on 13 (1101_2) for instance, we analyzed the value at index 13, then we removed the last significant beat so we got 1100_2 , so 12, so we analyzed the value at index 12, then we removed again the least significant, getting 1000_2 , so we analyzed the value at index 8.

Now, for doing point updates, instead of removing the last significant bit to get its index value , we will be adding the least significant bit. For instance, if we want to add a value at index 9 (1001_2), we have to find out all the cells which are responsible for that index, in this case 9. So, we start at 9 (1001_2) and find its least significant bit (in this case 1_2), so we get 1010_2 , so 10.

Then we find its least significant bit and add it, so 10_2 , and so we get 1100_2 , so 12. Add to its its least significant bit, so 100_2 , so we get 10000_2 , so 16.

And then we would do again the same thing, and then we are out of bounds, so we would know its time to stop.

Where **LSB** returns the value of the least significant bit. For instance $\text{LSB}(12) = 4$, because $12_{10} = 1100_2$ and the least significant bit of 1100_2 is 100_2 , or 4 in base ten.

1.6 Fenwick Tree Construction

Naive Construction

Let A be an array of values. For each element in A at index i do a point update on the Fenwick tree with a value of $A[i]$. There are n elements for a total of $O(n \log(n))$. But we should be able to do a better construction.

Linear Construction

Idea: Add the value in the current cell to the **immediate cell** that is responsible for us. This resembles what we did for point updates but only once cell at a time.

This will make the ‘cascading’ effect in range queries possible by propagating the value in each cell throughout the tree.

Let i be the current index, the immediate cell above us is at position j given by $j := i + \text{LSB}(i)$. Where LSB is the **Least Significant Bit** of i .

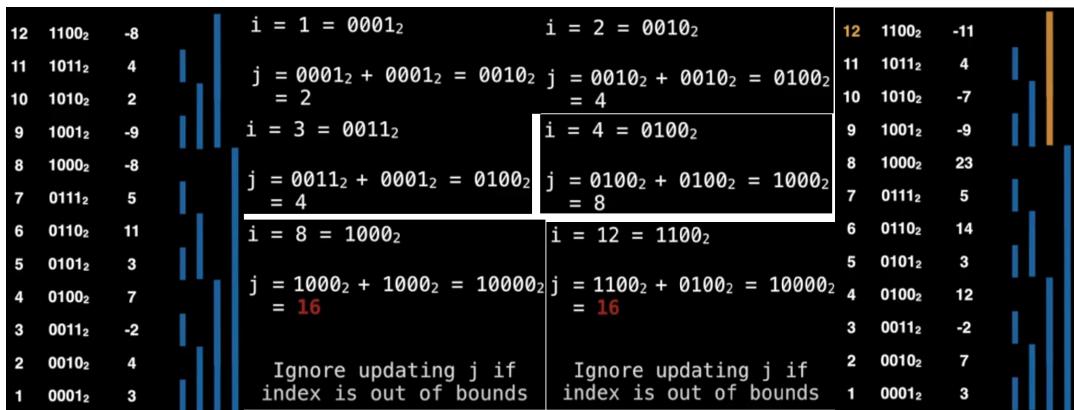


Figure 2:

So, this way we get the values of the Fenwick tree. And so we can now perform point and range query updates as required.

And the construction algorithm has the following form:

```
# Make sure values are 1-based!
function construct(values):
    N := length(values)

    # Clone the values array since we are
    # doing in place operations
    tree = deepCopy(values)

    for i = 1, 2, 3, ... N:
        j := i + LSB(i)
        if j < N:
            tree[j] = tree[j] + tree[i]

    return tree
```

2 Suffix Arrays

2.1 Introduction

A **suffix** is a substring at the end of a string of characters. For our purposes, suffixes are non empty. For example, if we have the word ‘HORSE’, we will have 5 possible and different suffixes: E, SE, RSE, ORSE and HORSE.

A **suffix array** is an array which contains all the **sorted** suffixes of a string.

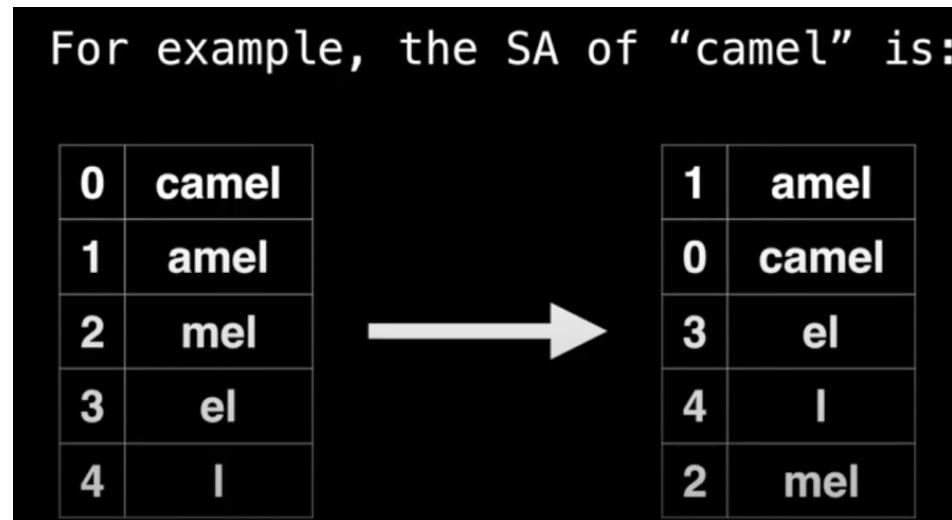


Figure 3:

The actual ‘**suffix array**’ is the array of sorted indices. This provides a compressed representation of the sorted suffixed without actually needing to store the suffixes.

What is a Suffix Array

The suffix array provides a space efficient alternative to a **suffix tree** which itself is a compressed version of a **tree**.

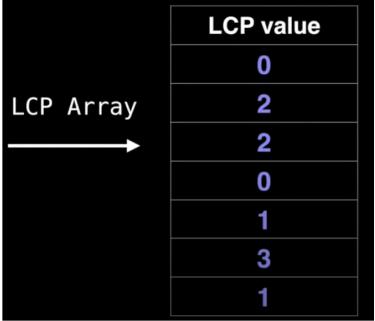
NOTE: suffix arrays can do everything suffix trees can, with some additional information such as a Longest Common Prefix (LCP) array.

2.2 The Longest Common Prefix (LCP) array

The LCP arrays is an array where each index stores how many characters two sorted suffix have in common with each other.

Begin by constructing the suffix array:

Sorted Index	LCP value	Suffix
5	0	AB
0	2	ABABBAB
2	2	ABBAB
6	0	B
4	1	BAB
1	3	BABBAB
3	1	BBAB



LCP value
0
2
2
0
1
3
1

For example, we have the string ‘ABABBAB’. Notice how the very first entry that we placed in our LCP array (which is the middle column on this figure) is 0. This is because this index is undefined, so we ignore it for now.

We begin by looking at the first two suffixed and seeing how many characters they have in common with each other. We notice that this is two, so we place a 2 in the first index of our LCP array.

Now we move on to the next suffix, analyzing the current row’s suffix and the previous one and we notice how it also has two characters in common. So we place another 2.

And then the next one does not have anyone in common with the previous one, so we place a 0. And so on.

In summary, the LCP array is an array in which every index tracks **how many characters two sorted adjacent suffixes have in common**. Although it may seem very simple, a lot of information can be derived from such a simple construction.

By convention, $LCP[0]$ is undefined, but for most purposes it is fine to set it to zero.

NOTE: There exists many methods for efficiently constructing the LCP array in $O(n \log(n))$ and $O(n)$.

2.3 Using SA/LCP array to find unique substrings

The problem of finding/counting all the unique substrings of a string is a commonplace problem in computer science.

The naive algorithm generates all substrings and places them in a set resulting in a $O(n^2)$ algorithm.

A better approach is to use the **LCP array**. This provides not only a quick but also a space efficient solution.

For instance, if we have the word ‘AZAZA’ the all possible $n(n + 1)/2$ substrings are: A, AZ, AZAZ, AZAZA, Z, ZA, ZAZ, ZAZA, A, AZ, AZA, Z, ZA, A

It is clear that there are some duplicate substrings. The number of unique substrings is 9.

Text: ‘AZAZA’

LCP	Sorted Suffixes
0	A
1	AZA
3	AZAZA
0	ZA
2	ZAZA

A, AZ, AZA, AZAZ, AZAZA, Z, ZA, ZAZ, ZAZA, A, AZ, AZA, Z, ZA, A

We will use the information in the LCP array to figure out which of those substrings really were duplicate. Remember that the LCP array represents the number of common characters in common have at the beginning of the word two adjacent suffixes.

So, if the LCP value at a certain index is, for instance, 5, there are five characters in common between those two suffixes, in other words, there are five repeated substrings between those two suffixes, since they come from the same larger string.

So, if we inspect the first LCP position at index 1, we see that has a value of one. We know that the repeated string is the first character, and so we know that A is a repeated substring.

The next LCP value is 3, so there are three repeated substrings between AZA and AZAZA: A, AZ and AZA.

The next interesting LCP value is 2, so there are two repeated substrings here we can eliminate: Z and ZA.

We can then come up with an interesting way of counting all unique substrings: We know how many substrings there are, which is $\frac{n(n+1)}{2}$ and we also know the number of duplicate strings, which is the sum of all the LCP values. So, if we subtract these two values, we will get the number of unique substrings in a string:

$$\frac{n(n+1)}{2} - \sum_{i=1}^n LCP[i]$$

For instance, if the text is ‘AZAZA’, which has $n = 5$:

$$\frac{5(5+1)}{2} - (1 + 3 + 0 + 2 + 0) = 9$$

2.4 Longest Common Substring (LCS)

The problem statement is: suppose we have n strings, how do we find the **longest common substring** that appears in at least $2 \leq k \leq n$ of the strings? For instance $n = 3$, $k = 2$ with the strings being $S_1 = 'abca'$, $S_2 = 'bcad'$ and $S_3 = 'daca'$.

The longest common substring does not require to be unique, they might be multiple.

The traditional approach of solving this problem is to use dynamic programming running in $O(n_1 * n_2 * \dots * n_m)$, where n_i is the length of the string S_i .

This approach might be okay if we have a few small strings, but rapidly gets unwieldy.

An alternative method is to use a suffix array which can find the solution in $O(n_1 + n_2 + \dots + n_m)$.

To find the LCS, we first create a new larger string T which is the concatenation of all the strings S_i separated by **unique sentinels**: $T = S_1 + '#' + S_2 + '$' + S_3 + '%' = abca\#bcad\$daca%$.

NOTE: The sentinels must be unique and lexicographically less than any of the characters contained in any of the strings S_i . In this case the characters #, \$ and % are less than any alphabetic character contained within S_1 , S_2 and S_3 .

Once we have the string T , we are able to construct the suffix array for T , this procedure can be accomplished in linear time with a linear suffix array construction algorithm.

We are displaying both the LCP array on the leftmost column and the sorted suffixes of T as they would appear in the suffix array on the right column.

```

          0 #bcad$daca%
          0 $daca%
          0 %
T = abca#bcad$daca% 0 a#bcad$daca%
1 a%
1 abca#bcad$daca%
1 aca%
1 ad$daca%
0 bca#bcad$daca%
3 bcad$daca%
0 ca#bcad$daca%
2 ca%
2 cad$daca%
0 d$daca%
1 daca%

```

The suffix array ->

Figure 4:

We can see that the suffixes starting with sentinel values got sorted to the top, because they were lexicographically less than all the characters in the string, as expected.

We want to ignore these strings since we inserted them in the T string ourselves.

Given that we now have the suffix array and the LCP array constructed, we are going to look for K strings, each from different colors whom share the largest LCP value.

For instance, if $k = 3$, we have 3 strings, this means that we need one string of each color, we can achieve a maximum of 2 if we sell the following three adjacent suffixes:

```

ca#bcad$daca%
ca%
cad$daca%

```

Notice that each of them is from a different color, meaning that they came from different original strings. And that the minimum LCP value in the selected window (note that we must ignore the first entry in the window) is two, with the string ‘ca’, which is shared amongst all three of the strings that we were given.

Now we will change the value of k to be 2. Since $K = 2$, we want to have two suffixes of different colors and the maximum longest common prefix value between them.

In this case there is a unique solution for $k = 2$, which is ‘bac’, with a length of 3. It is obtained with the following words:

```

bca#bcad$daca%
bcad$daca%

```

Shared between S_1 and S_2 but not S_3 .

Things can get more messy when suffixes of different colours are not exactly adjacent.

We should use a **sliding window** to capture the correct amount of suffix colours. At each step advance the bottom endpoint or adjust the top endpoint such that the window contains exactly K suffixes of different colours.

For each valid window we perform a range query on the LCP array between the bottom and top endpoints. The LCS will be the maximum LCP value for all possible windows.

The **minimum sliding range query problem** can be solved in a total of $O(n)$ time for all windows.

Alternatively, we can use min range query DS such as a segment tree to perform queries in $O(n \log(n))$ time, which may be easier but slightly slower running for a total of $O(n \log(n))$.

Additionally, we will need a DS (hashtable) to track the colours in our sliding window:

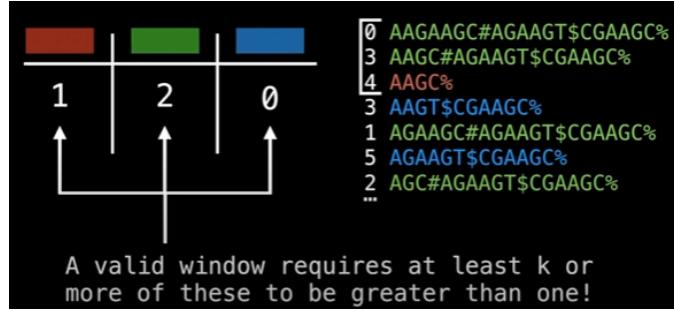


Figure 5:

The idea is to expand the window to acquire the correct amount of each colour and retract the window when the correct amount of colours is met.

2.4.1 LCS example

We will consider the following 4 strings:

$$S_1 = \text{AABC}$$

$$S_2 = \text{BCDC}$$

$$S_3 = \text{BCDE}$$

$$S_4 = \text{CDED}$$

And our task is to find the Longest Common Substrings (LCS) that appears in at least two of the strings ($k = 2$). We get the following T string:

$$T = \text{AABC}\#\text{BCDC\$BCDE\%CDED\&}$$

And the correct possible answers are: BCD, CDE

$S_1 = \text{AABC}$, $S_2 = \text{BCDC}$				
$S_3 = \text{BCDE}$, $S_4 = \text{CDED}$				
Window LCP = NA	Window LCP = 0	Window LCP = 0	Window LCP = 2	Window LCP = 2
Window LCS = NA	Window LCS = ""	Window LCS = ""	Window LCS = BC	Window LCS = BC
LCS length = 0	LCS length = 0	LCS length = 0	LCS length = 2	LCS length = 2
LCS = {}	LCS = {}	LCS = {}	LCS = { BC }	LCS = { BC }
0 D\$				
1 DC\\$BCDE\%CDED\&				
1 DE\%CDED\&				
2 DED\$				
0 E\%CDED\&				
1 ED\&				

Figure 6:

The first step in finding the longest common substrings of our 4 strings is to build the suffix array and the LCP array, which are displayed on the above image.

The window LCP and the window LCS will track the longest common prefix and the longest common substring values for the current window. And the ICS length and the LCS set will track the best values so far.

Initially, the window starts at the top and we want the window to contain two different colors ($k = 2$). So our rule is to expand downwards when we do not meet this criteria. For the first

substrings, we see how as we expand down we still get strings of the color green, so we have to expand until we also group a different color, until we get the state in the middle picture.

Here we are in to perform a range query for our window. However, our query is not fruitful because the window longest common prefix value is zero, so there is not longest common substring here. When we satisfy the window color criteria as we do now, we decrease its size, until we arrive to the state of the image on the right. The current window contains a longest common prefix length of 2. So we obtain the longest common substring and add it to our solution set.

Now, we keep decreasing our window size since we meet the color requirement.

$S_1 = \text{AABC}$, $S_2 = \text{BCDC}$ $S_3 = \text{BCDE}$, $S_4 = \text{CDED}$ Window LCP = NA Window LCS = NA LCS length = 2 LCS = { BC }	$0 \text{ AABC}\#BCDC\$BCDE\%CDED\&$ $1 \text{ ABC}\#\text{BCDC\$BCDE\%CDED\&}$ $0 \text{ BC}\#\text{BCDC\$BCDE\%CDED\&}$ $2 \text{ BCDC\$BCDE\%CDED\&}$ 3 BCDE\%CDED\& $0 \text{ C\$BCDC\$BCDE\%CDED\&}$ $1 \text{ C\$BCDE\%CDED\&}$ $1 \text{ CDC\$BCDE\%CDED\&}$ 2 CDE\%CDED\& 3 CDED\& 0 D\& $1 \text{ DC\$BCDE\%CDED\&}$ 1 DE\%CDED\& 2 DED\& 0 E\%CDED\& 1 ED\&	$0 \text{ AABC}\#\text{BCDC\$BCDE\%CDED\&}$ $1 \text{ ABC}\#\text{BCDC\$BCDE\%CDED\&}$ $0 \text{ BC}\#\text{BCDC\$BCDE\%CDED\&}$ $2 \text{ BCDC\$BCDE\%CDED\&}$ 3 BCDE\%CDED\& Window LCP = 3 Window LCS = BCD LCS length = 3 LCS = { BCD }	$0 \text{ C\$BCDC\$BCDE\%CDED\&}$ $1 \text{ C\$BCDE\%CDED\&}$ $1 \text{ CDC\$BCDE\%CDED\&}$ 2 CDE\%CDED\& 3 CDED\& 0 D\& $1 \text{ DC\$BCDE\%CDED\&}$ 1 DE\%CDED\& 2 DED\& 0 E\%CDED\& 1 ED\&
--	--	---	---

Figure 7:

Now, our window size is too small, since it does not fulfil the color requirement, as it can be seen in the above image at the left. We need two different color strings so we expand once more. And we arrive at the state on the right.

We have found an LCP value of 3, which is larger than our current best value, so we update the solution set to have the string BCD.

$S_1 = \text{AABC}$, $S_2 = \text{BCDC}$ $S_3 = \text{BCDE}$, $S_4 = \text{CDED}$ Window LCP = 2 Window LCS = CD LCS length = 3 LCS = { BCD }	$0 \text{ AABC}\#BCDC\$BCDE\%CDED\&$ $1 \text{ ABC}\#\text{BCDC\$BCDE\%CDED\&}$ $0 \text{ BC}\#\text{BCDC\$BCDE\%CDED\&}$ $2 \text{ BCDC\$BCDE\%CDED\&}$ 3 BCDE\%CDED\& $0 \text{ C\$BCDC\$BCDE\%CDED\&}$ $1 \text{ C\$BCDE\%CDED\&}$ $1 \text{ CDC\$BCDE\%CDED\&}$ 2 CDE\%CDED\& 3 CDED\& 0 D\& $1 \text{ DC\$BCDE\%CDED\&}$ 1 DE\%CDED\& 2 DED\& 0 E\%CDED\& 1 ED\&	$S_1 = \text{AABC}$, $S_2 = \text{BCDC}$ $S_3 = \text{BCDE}$, $S_4 = \text{CDED}$ Window LCP = 3 Window LCS = CDE LCS length = 3 LCS = { BCD, CDE }	$0 \text{ AABC}\#\text{BCDC\$BCDE\%CDED\&}$ $1 \text{ ABC}\#\text{BCDC\$BCDE\%CDED\&}$ $0 \text{ BC}\#\text{BCDC\$BCDE\%CDED\&}$ $2 \text{ BCDC\$BCDE\%CDED\&}$ 3 BCDE\%CDED\& $0 \text{ C\$BCDC\$BCDE\%CDED\&}$ $1 \text{ C\$BCDE\%CDED\&}$ $1 \text{ CDC\$BCDE\%CDED\&}$ 2 CDE\%CDED\& 3 CDED\& 0 D\& $1 \text{ DC\$BCDE\%CDED\&}$ 1 DE\%CDED\& 2 DED\& 0 E\%CDED\& 1 ED\&
--	--	--	---

Figure 8:

Now we have to decrease the window size, since we fulfill the window color requirement. Then expand it, since we do not fulfill it, and then again since we have just two blue substrings. Now we fulfill it, and we see that the window LCP is 1.

We decrease it and see that we arrive to the state at the above left. The window LCP value is 2, so since it is lower than our maximum we keep going. We decrease it and size it does not satisfy the requirement we increase it, and arrive to the state on the left of the above image.

We see how we get a window LCP of 3, so we add it to our solution set.

And we continue this way until the end. This way we obtain the following solution set: BCD, CDE, with a LCS length of 3.

2.5 Longest Repeated Substring (LRS)

The longest repeated substring is a problem which we can efficiently solve using the information stored in the LCP array.

The brute force method requires $O(n^2)$ time and lots of space. Using the information inside the LCP array saves us time and space.

For example, the longest repeated substring in the word ‘ABRACADABRA’ is ‘ABRA’. It does not matter that the longest common substrings appear disjoint and not overlapping between them, as we have seen in this case, this is permitted.

Find the LRS of the string: ABRACADABRA	
LCP	Suffix
0	A
1	ABRA
4	ABRACADABRA
1	ACADABRA
1	ADABRA
0	BRA
3	BRACADABRA
0	CADABRA
0	DABRA
0	RA
2	RACADABRA

Find the maximum LCP value

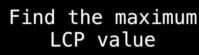


Figure 9:

What we are looking for in the LCP array is the maximum LCP value. Intuitively, we want to do this since we know that the suffixes are already sorted. So, if two adjacent suffixes have a large, longest common prefix value, then they share a good amount of characters with each other.

We also know that if the LCP value at a certain index is greater than zero, then the string shared between the two adjacent suffixes is guaranteed to be repeated, because it is common between two suffixes, each of which start at different locations in the string.

3 Balanced Binary Search Trees (BBSTs)

3.1 Introduction

A **Balanced Binary Search Tree (BBST)** is a **self-balancing** binary search tree. This type of tree will adjust itself in order to maintain a low (logarithmic) height allowing for faster operations such as insertions and deletions.

3.2 Complexity

The Binary Search Tree has on average all operations to require logarithmic time. But in the worst case we still have the linear complexity for all the cases, given that the tree could degrade into a chain for some inputs. One example of these type of inputs is a sequence of increasing numbers.

To avoid these logarithmic worst cases the Balanced Binary Search Trees were invented. Where even in the worst case, for all operations, we get a logarithmic time complexity.

Operation	Average	Worst
Insert	$O(\log(n))$	$O(\log(n))$
Delete	$O(\log(n))$	$O(\log(n))$
Remove	$O(\log(n))$	$O(\log(n))$
Search	$O(\log(n))$	$O(\log(n))$

Figure 10:

3.3 Tree Rotations

Balanced Binary Search Trees rely on the usage of the **tree invariant** and the **tree rotations**.

A **tree invariant** is a property/rule we have to impose on our tree to meet after every rotation. To ensure that the invariant is always satisfied, a series of tree rotations are normally applied.

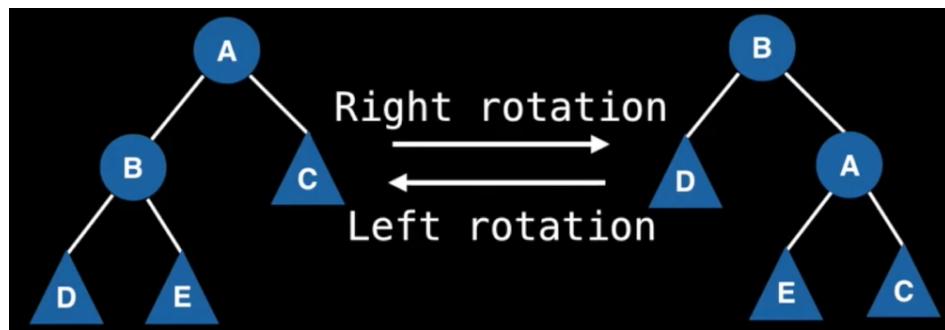


Figure 11:

For example, in the left tree we know that $D < B < E < A < C$, and this remains true for the right subtree, so we did not break the BST invariant and therefore this is a valid transformation.

Also, recalling that all BBSTs are BSTs, so the BST invariant holds. This means that for every node n , $n.left < n$ and $n < n.right$.

(NOTE: The above statement assumes we only have **unique values**, otherwise we would have to consider the case where $n.left \leq n$ and $n \leq n.right$)

It does not matter what the structure of the tree looks, all we care about is that the BST invariant holds. This means we can shuffle/transform/rotate the values and nodes in the tree as we please, as long as the BST invariant remains satisfied.

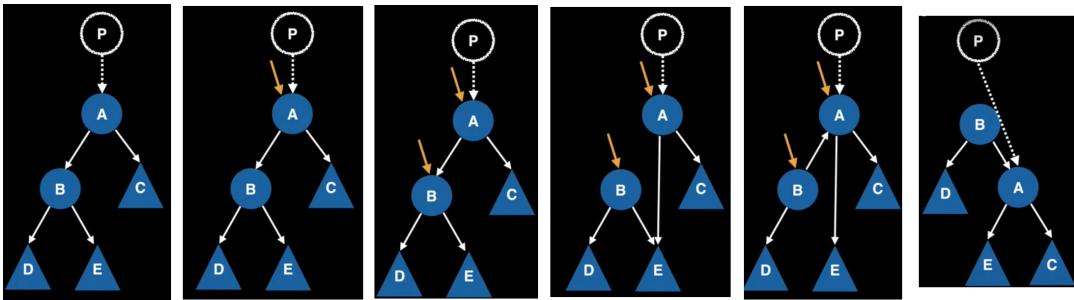


Figure 12:

The example of a procedure of a right rotation is done above.

First, we have to notice how there are directed edges pointing downwards and another node P above A , which may or may not exist. This is why there is a dotted line on that edge.

If a node A does have a parent node P , then it is important that we take it into account when doing the rotation.

In either case, we start with a pointer reference to node A , as it can be seen in the second image (starting from the left), which is marked by the orange arrow. Then we will want a point to node B

After that is set, A 's left pointer will point to B 's right child, then we will change B 's right pointer to point to node A . And so, we will successfully have completed a right rotation. Rearranging the nodes of the tree, we get the tree on the image on the right.

The function looks the following way:

```
function rightRotate(A):
    B := A.left
    A.left = B.right
    B.right = A
    return B
```

NOTE: It is possible that before the rotation, node A had a parent whose left/right pointer referenced it. It is very important that this link be updated to reference B . This is usually done on the recursive callback using the *return* value of *rotateRight*.

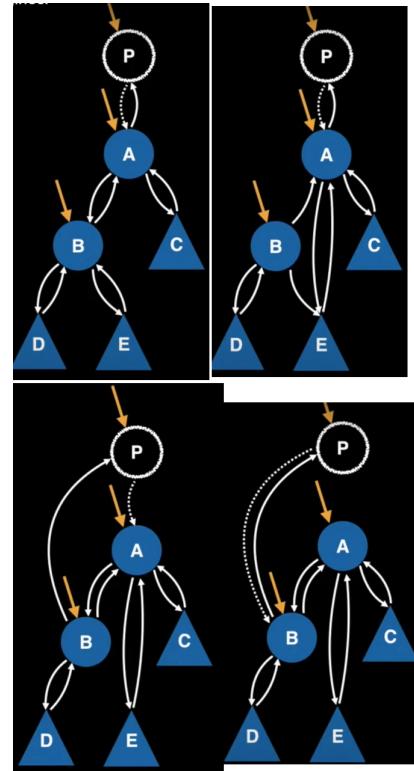
In some BBST implementations where we often need to access the parent/uncle nodes (such as RB trees), it is convenient for nodes to not only have a reference to the left and the right child

nodes, but also the parent node. This can complicate tree rotations because instead of updating 3 pointer, now we would have to update 6.

```

function rightRotate(A):
    P := A.parent
    B := A.left
    A.left = B.right
    if B.right != null:
        B.right.parent = A
    B.right = A
    A.parent = B
    B.parent = P
    # Update parent down link
    if P != null:
        if P.left == A:
            P.left = B
        else:
            P.right = B
    return B

```



In this example, where we also have a parent link, every node is doubly linked.

We start off with a pointer or referencing *A* and the first thing we want to do is to also reference node *P* and *B*, so we do not lose them as we shuffle around pointers.

Next, we will adjust the left subtree of *A* to make *A*'s left pointer reference *B*'s right subtree. (Throughout this example we assume *B* is not null, we could add an extra *if* statement to check for this condition).

It would be a mistake not to assume *B*'s right subtree is not null, we have to check against this before setting *B*'s right child's parent to reference *A*.

Next, we have to make *A* the right subtree of *B*, so this means that the right pointer of *B* needs to reference *A*.

Now we make *A*'s parent pointer or reference *B*.

And the last thing we need to do is to adjust the reference to the parent node *P*. SO we have to make *B*'s parent pointer to reference *P*.

And so, now the last remaining thing is to make *P*'s left or right pointer to reference the successor node *B*.

Notice that we have to check that *P* is not null since it might not exist. And then, readjusting the tree would see that we have made right the wanted rotation.

3.4 Inserting Elements into an AVL tree

An **AVL Tree** is one of many types of **Balanced Binary Search Trees (BBSTs)** which allow for logarithmic $O(\log(n))$ insertion, deletion and search operations.

The property which keeps an AVL tree balanced is called the **Balance Factor (BF)**:

$$BF(node) = H(node.right) - H(node.left)$$

Where $H(x)$ is the height of node x . Recall that $H(x)$ is calculated as the **number of edges** between x and the furthest leaf.

The invariant in the AVL which forces it to remain balanced is the requirement that the balance factor is always either -1, 0 or +1.

Node Information we have to store

- The actual value we are storing in the node. NOTE: This value must be comparable so we know how to insert it.
- A value storing this node's **balance factor**
- The **height** of this node in the tree.
- Pointers to the **left/right child nodes**.

Obviously, as the tree evolves, we will be updating these values.

What if the BF of a node is $\notin \{-1, 0, 1\}$? How do we restore the AVL tree invariant?
If a node's $BF \notin \{-1, 0, 1\}$, then the BF of that node is $+ - 2$, which can be adjusted using **tree rotations**.

So, the balance of the node should always be -1, 0 or +1. We will always be having one of these four cases when this is not true, so we will have to solve it using tree rotations as it is shown below:

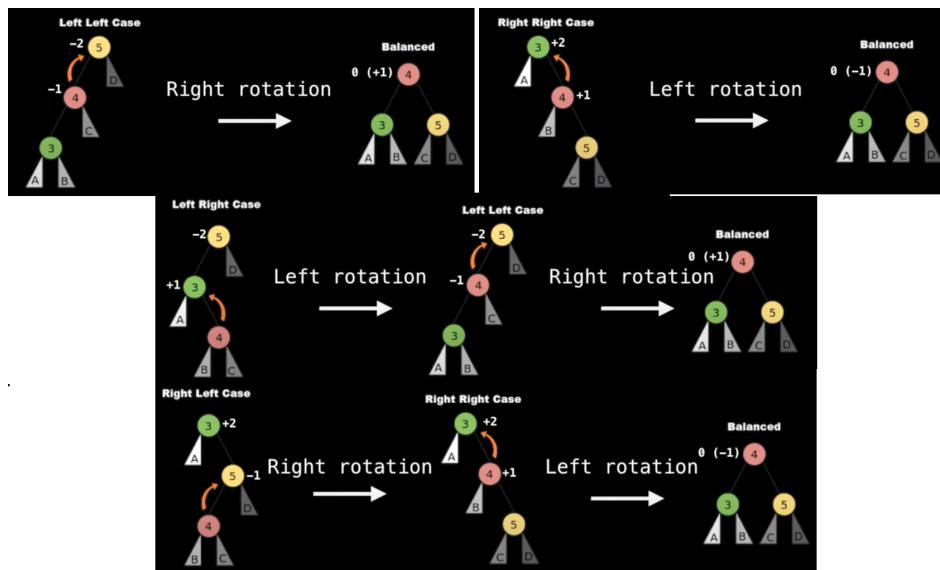


Figure 13:

3.5 Node insertion in AVL trees

Public facing insert method:

```
# Public facing insert method. Returns true
# on successful insert and false otherwise
function insert(value):

    if value == null:
        return false

    # Only insert unique values
    if !contains(root, value):
        root = insert(root, value)
        nodeCount = nodeCount + 1
        return True

    # Value already exists in tree
    return False
```

Private insert recursive case

```
function insert(node, value):
    if node == null: return Node(value)

    # Invoke the comparator function in whatever
    # programming language we are using.
    cmp := compare(value, node.value)

    if cmp < 0:
        node.left = insert(node.left, value)
    else:
        node.right = insert(node.right, value)

    # Update balance factor and height values
    update(node)

    # Rebalance tree
    return balance(node)
```

Update method

```
function update(node):

    # Variables for left/right subtree heights
    lh := -1
```

```

rh := -1
if node.left != null: lh = node.left.height
if node.right != null: rh = node.right.height

# Update this node's height
node.height = 1 + max(lh, rh)

# Update balance factor
node.bf = rh - lh

```

Balance method

```

function balance(node):

    # Left heavy subtree
    if node.bf == -2:
        if node.left.bf <= 0:
            return leftLeftCase(node)
        else:
            return leftRightCase(node)

    # Right heavy subtree
    else if node.bf == +2:
        if node.right.bf >= 0:
            return rightRightCase(node)
        else:
            return rightLeftCase(node)

    # Node has balance factor of -1, 0 or +1
    # which we do not need to balance
    return node

```

All we have to do are calls to the left rotation and right rotation methods that we have just seen.

Notice that the left right and right left cases call the left left and right right case methods respectively, since they reduce to those cases after a first rotation:

```

function leftLeftCase(node):
    return rightRotation(node)

function leftRightCase(node):
    node.left = leftRotation(node.left)
    return lrftLeftCase(node)

function rightRightCase(node):
    return leftRotation(node)

```

```
function rightLeftCase(node):
    node.right = rightRotation(node.right)
    return rightRightCase(node)
```

But, we are dealing with **any** AVL tree, so we need to augment the method to update the height and balance rate values for the nodes we are moving around when we do the rotations. This is a subtle detail **we must not forget, otherwise our height and balance factor value will be inconsistent with the left rotation cases:**

```
function rightRotate(A):
    B := A.left
    A.left = B.right
    B.right = A
    # After rotation, update balance
    # factor and height values !!!
    update(A)
    update(B)
    return B
```

And we should work similarly with the left rotation.

3.6 Removing Elements from an AVL Tree

Removing elements from an AVL tree is almost identical to removing elements from a regular binary search tree.

Removing elements from a Binary Search Tree (BST) can be seen as a two-step process:

1. **Find** the element we wish to remove (if it exists).
2. **Replace** the node we want to remove with its successor (if any) to maintain the BST invariant.

Recall the **BST invariant**: left subtree has smaller elements and right subtree has larger elements.

When searching our BST for a node with a particular value, one of the following four things will happen:

1. We hit a **null node** at which point we know the value does not exist within our BST.
2. Comparator value **equal to 0** (found it!).
3. Comparator value **less than 0** (the value, if it exists, is in the left subtree).
4. Comparator value **greater than 0** (the value, if it exists, is in the right subtree).

In the remove phase we will have one of the following four cases:

- Node to remove is a leaf node.
- Node to remove has a right subtree but no left subtree.
- Node to remove has a left subtree but no right subtree.
- Node to remove has both, a left subtree and a right subtree.

And so, we will have to act according to each case:

- **Case I, Node to remove is a leaf node:** If the node we wish to remove is a leaf node the we may do so without side effects.
- **Cases II & III, either the left/right child node is a subtree:** The successor of the node we are trying to remove in these cases will be the **root node of the left/right subtree**. It may be the case that we are removing the root node of the BST, in which case its immediate child becomes the new root.
- **Case IV, Node to remove has both, a left subtree and a right subtree:** We have to decide in which subtree will the successor of the node we are trying to remove be.

We will be using BOTH. The successor can either be the largest value in the left subtree or the smallest value in the right subtree.

Once the successor node has been identified (if it exists), we replace the value of the node to remove with the value in the successor node.

NOTE: We do not have to forget to remove the duplicate value of the successor node that still exists in the tree at this point! One strategy to resolve this is by calling the function again recursively, but with the value to remove as the value in the successor node.

(THIS IS WELL EXPLAINED AND HAS SOME EXAMPLES IN THE BINARY SEARCH TREE SECTION)

3.7 Augmenting BST Removal Algorithm for AVL Tree

Augmenting the removal algorithm from a plain BST implementation to an AVL tree is just as easy as adding two lines of code:

```
function remove(node, value):  
    #=====  
    # Code for BST item removal here  
    #=====  
  
    # Update balance factor  
    update(node)  
  
    # Rebalance tree  
    return balance(node)
```
