Data Structures and Algorithms
Part 2

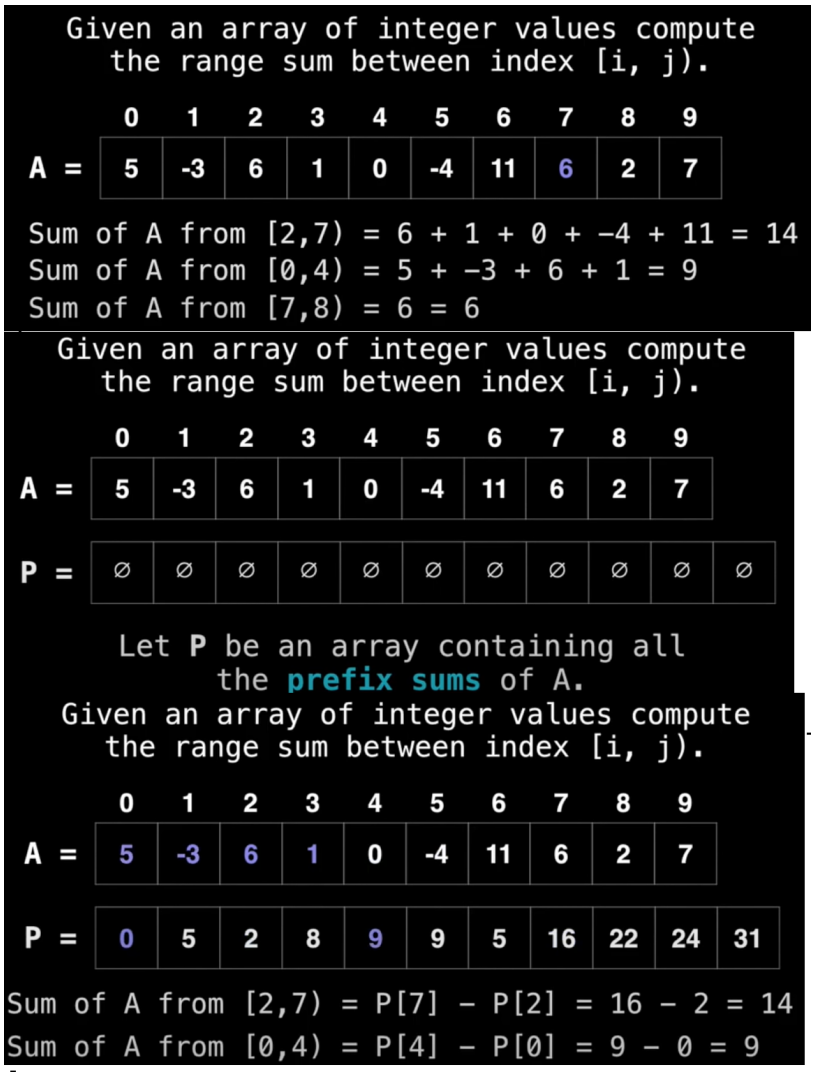
Iñaki Lakunza

April 4, 2024

Contents

1 Fenwick Tree (Binary Index Tree)

1.1 Introduction



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 cam 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.

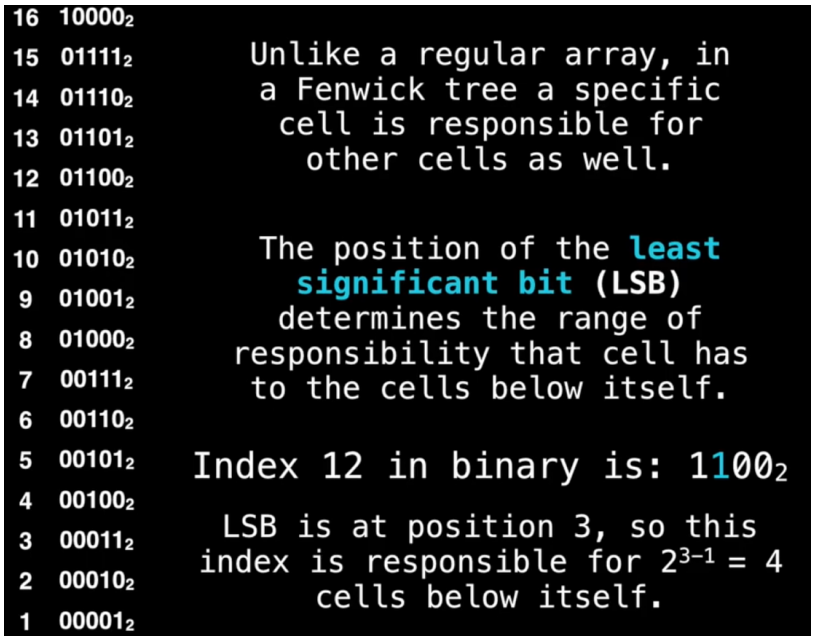
Construction	O(n)
Point Update	O(log(n))
Range Sum	O(log(n))
Range Update	O(log(n))
Adding Index	N/A
Removing Index	N/A

Figure 1:

1.2 Fenwick Tree Complexity

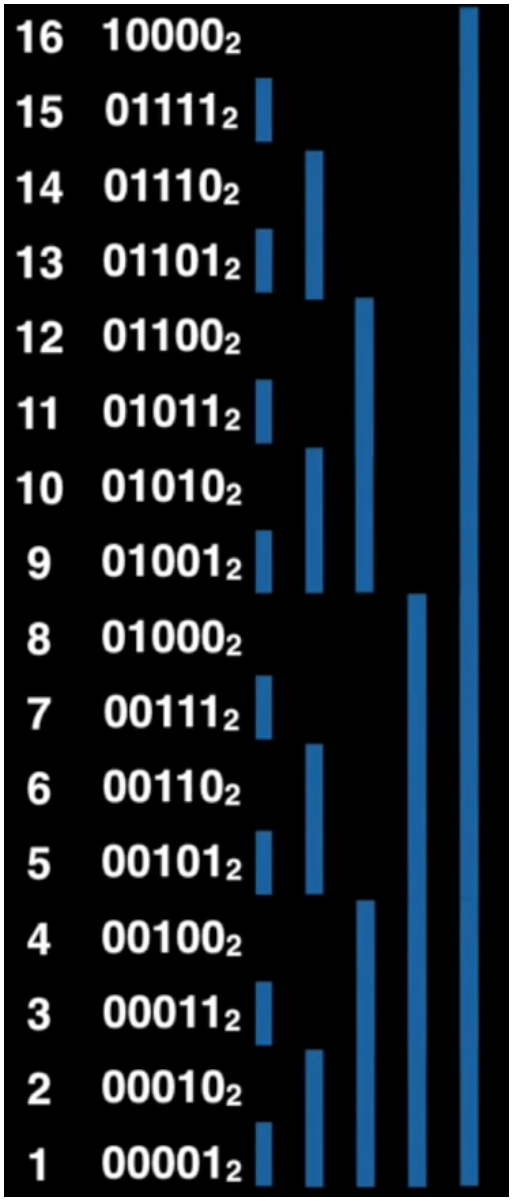
Even though its construction requires linear time, points and updates are logarithmic, range sum queries are also logarithmic. But we can add or remove elements from the array

1.3 Fenwick Tree Range Queries



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 cells is only responsible for 1 cell below



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:

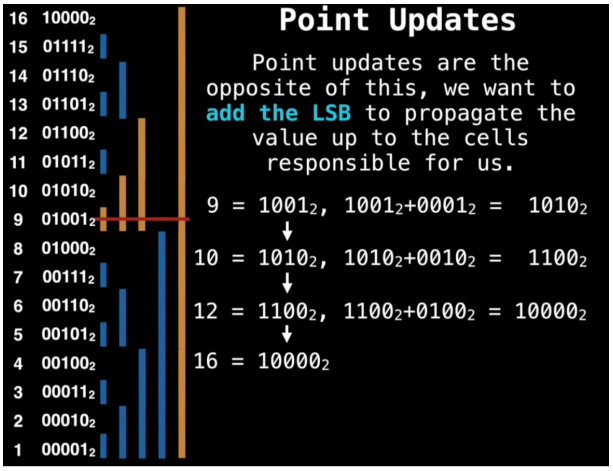
```
function prefixSum(i);
    sum := 0
```

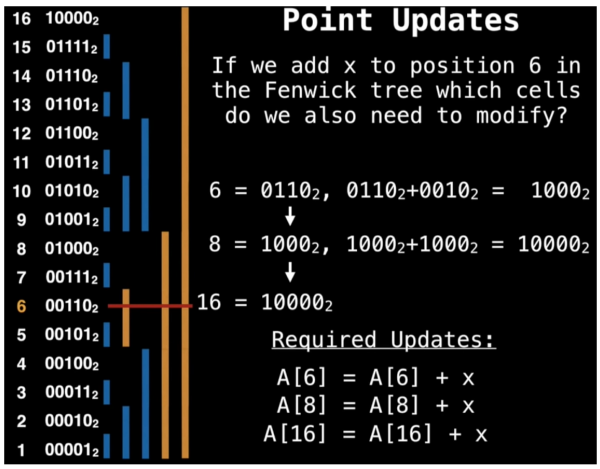
```
    while i != 0:
        sum = sum + tree[i]
        i = i - LSB(i)
    return sum

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





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.

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 see 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)
```

Where **LSB** returns the value of the least significant bit. For instance **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 $0(nlog(n))$. 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 $j := i + LSB(i)$. Where LSB is the **Least Signigicant Bit** of i.

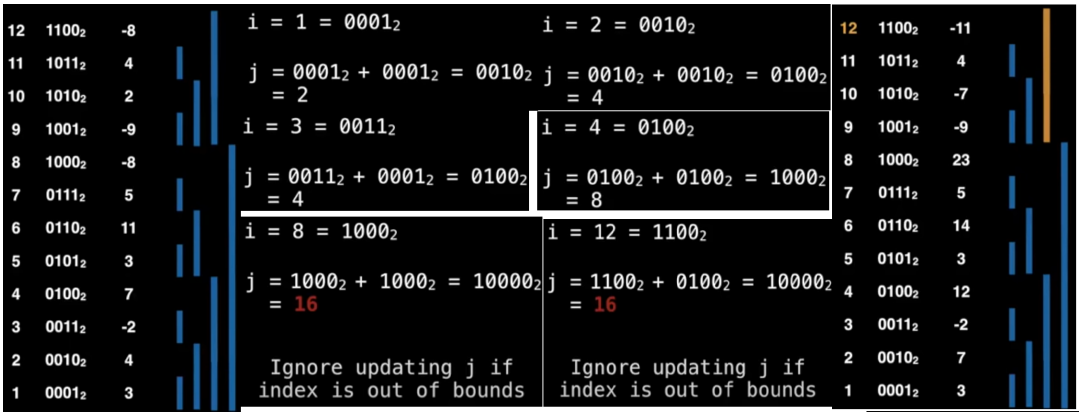


Figure 2:

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.

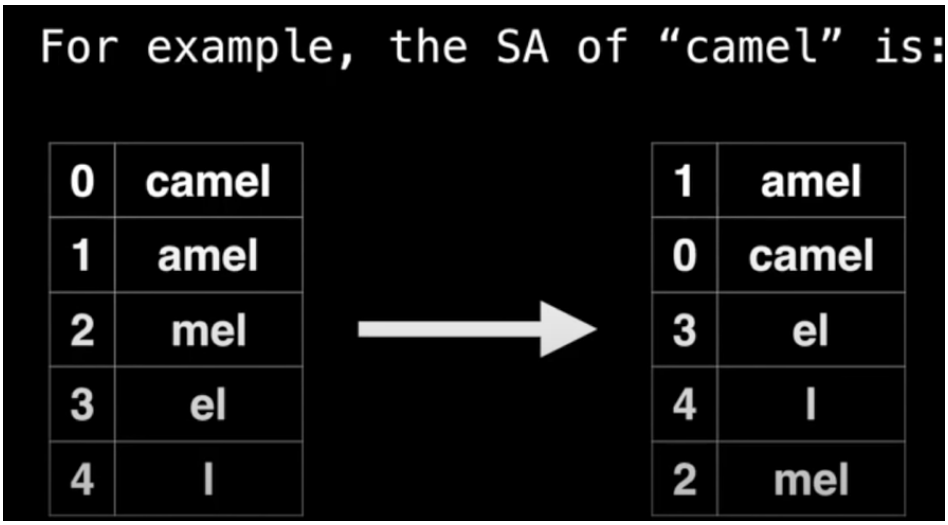


Figure 3:

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 Array →

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 purposed it is fine to set it to zero.

NOTE: There exists many methods for efficiently construction the LCP array in $0(nlog(n))$ and $0(n)$.

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 $0(n^2)$ 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'

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

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

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 postion 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 statemenet 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 $S_1 =' daca'$.

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 $0(n_1 * n_2 * ... * n_m)$, 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 $0(n_1 + n_2 + ... + n_m)$.

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**: $T = S_1 + \text{`}\#' + S_2 + \text{`}\$' + S_3 + \text{`}\%' = abca\#bcad\$daca\%$.

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.

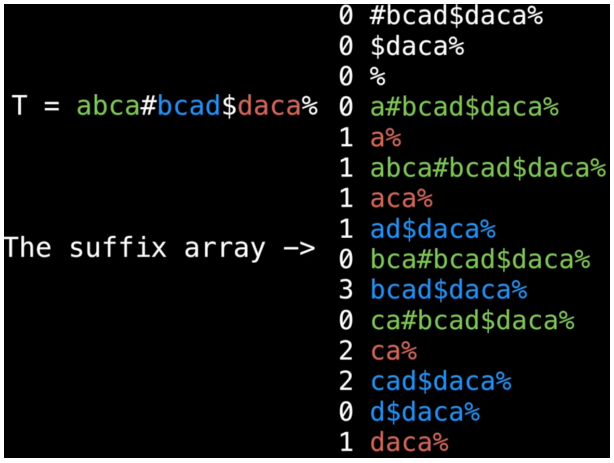


Figure 4:

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 eacj 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 $0(n)$ time for all windows.

Alternatively, we can use min range query DS such as a segment tree to perform queries in $log(n)$ time, which may be easier but slightly slower running for a total of $0(nlog(n))$.

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

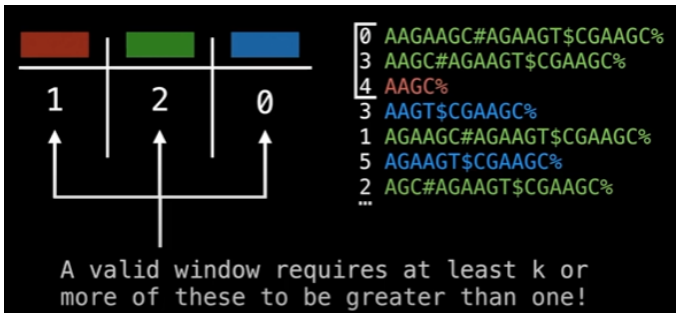


Figure 5:

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 = AABC
S_2 = BCDC
S_3 = BCDE
S_4 = 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 = AABC#BCDC$BCDE%CDED&

And the correct possible answers are: BCD, CDE

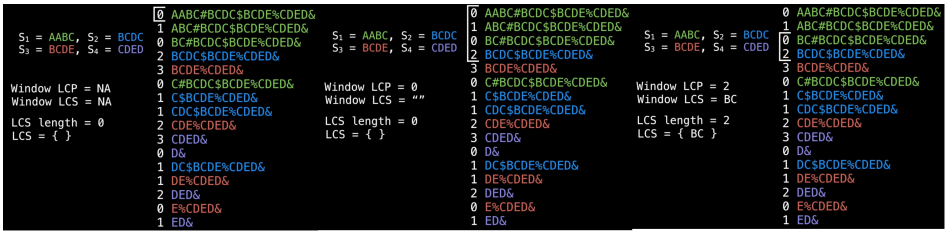


Figure 6:

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 lCS 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.

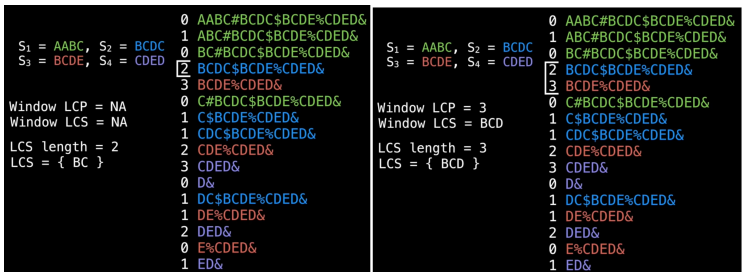


Figure 7:

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.

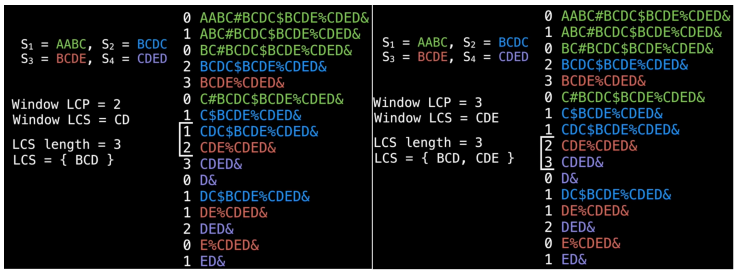


Figure 8:

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

The brute force method requires $0(n^2)$ 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.

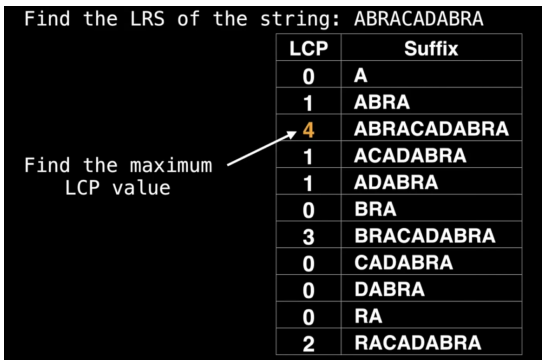


Figure 9:

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 Tress 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 satisfies, a series of tree rotations are normally applied.

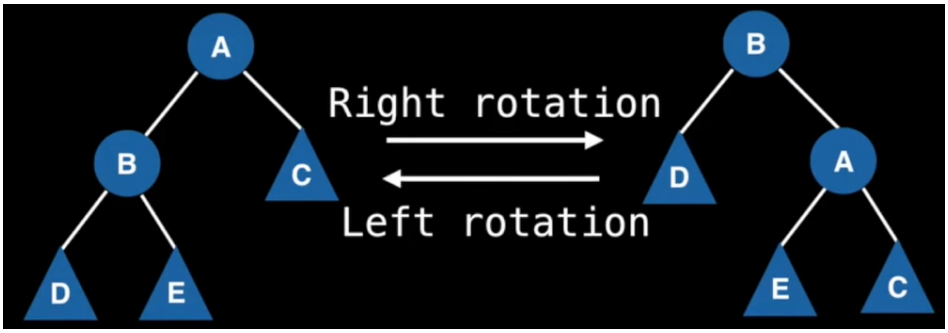


Figure 11:

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.

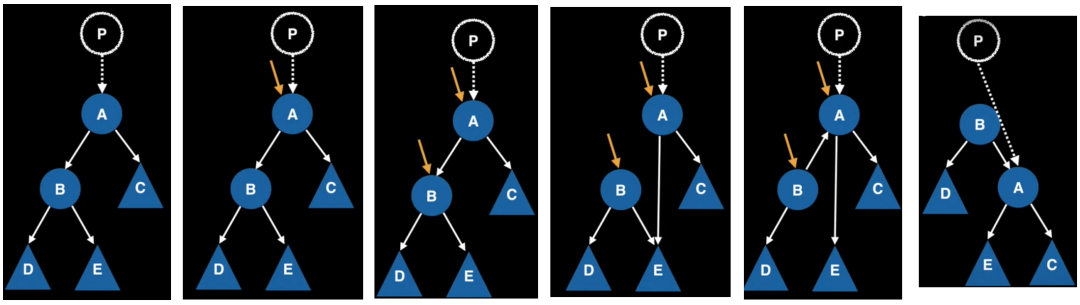


Figure 12:

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:

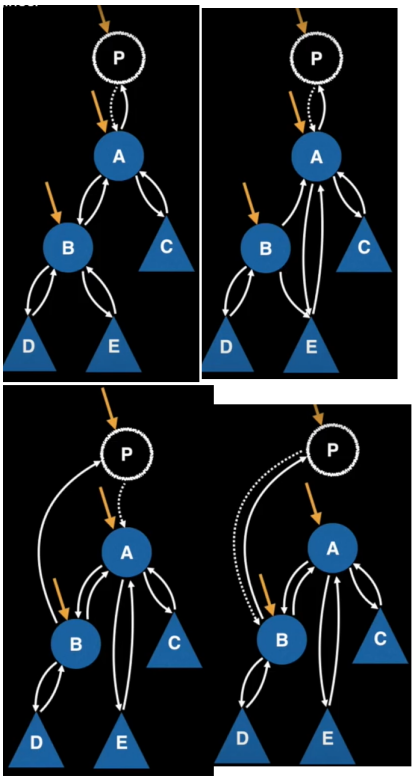
```
function rightRotate(A):
    B := A.left
    A.left = B.right
    B.right = A
    return B
```

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 $0(log(n))$ 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:

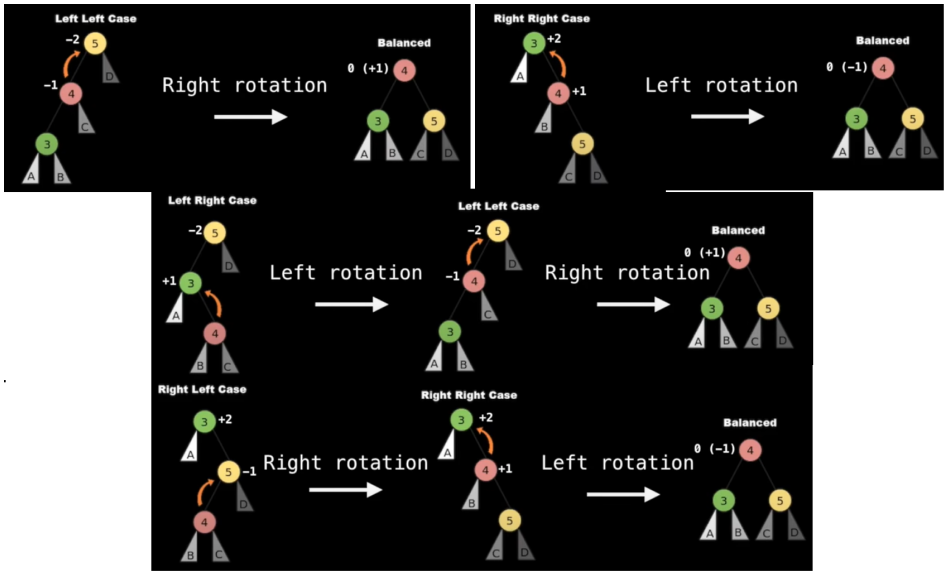


Figure 13:

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.left, 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)
```

4 Indexed Priority Queue

4.1 Introduction

An **Indexed Priority Queue** is a traditional priority queue variant which on top of the regular Priority Queue operations supports **quick updates and deletions** of **key-value pairs**.

One of the big problem that Index Priority Queue solves is that it is able to quickly look up dynamically changes of values in our priority queue on the fly, which is often very useful.

As an example, we can image an hospital where people are waiting to be attended. Depending on their urgency level they will have a higher or a lower priority, so the order should be followed by their priority level.

But, it might be possible that while they are attending the first patient another one arrives which has higher priority than the one who should go 3^{rd}, so we have to modify that priority on the fly. And it might happen that the one who was going in 5^{th} suddenly needs to be attended much more urgently, so we have to augment its priority and change the order.

Indexed priority queues are really useful for this kind of tasks, since we have to be able to **dynamically update the priority** (value) of certain people (keys).

The **Indexed Priority Queue (IPQ)** data structure lets us do this efficiently. The first step to using a IPQ is to assign index values to all the keys forming a bidirectional mapping.

We have to assing to each element a unique key-index value, so we can work with them.

NOTE: This mapping is intended to be bidirectional, so we can use a bidirectional hashtable to be able to flip back and fort between an elements key and its key-index (in the case of the hospital, for instance, the key would be the name of the patient, 'Mary' for instance, and its key index would be its priority level, 2, for instance).

Why are we mapping keys to indexes in the domain $[0, N)$?
Typically, priority queues are implemented as heaps under the hood, which internally use arrays which we wan to facilitate indexing into.

NOTE: Often the keys themselves are integers in the range $[0, N)$, so there is no need for the mapping, but it is handy to be able to support any type of key (like names).

We can think of the indexed priority queue as an abstract data structure with certain operations we want it to support. Here is a list with some of the operations we want our IPQ to support. Take into account that if k is the key we want to update, we first have to get the key's index: $ki = map[k]$, and then we have to use ki with the priority queue:

- *delete(ki)*
- *valueOf(ki)*
- *contains(ki)*
- *peekMinKeyIndex(ki)*
- *pollMinKeyIndex(ki)*
- *peekMinValue()*

- *poolMinValue()*
- *insert(ki, value)*
- *update(ki, value)*
- *decreaseKey(ki, value)*
- *increaseKey(ki, value)*

4.2 IPQ complexity

Operation	Indexed Binary Heap PQ
`delete(ki)`	O(log(n))
`valueOf(ki)`	O(1)
`contains(ki)`	O(1)
`peekMinKeyIndex()`	O(1)
`pollMinKeyIndex()`	O(log(n))
`peekMinValue()`	O(1)
`pollMinValue()`	O(log(n))
`insert(ki, value)`	O(log(n))
`update(ki, value)`	O(log(n))
`decreaseKey(ki, value)`	O(log(n))
`increaseKey(ki, value)`	O(log(n))

Figure 14:

We will cover the binary heap implementation for creating our Indexed Priority Queue.

Notice how all operations require a logarithmic or a constant time.

In a regular priority queue the remove and update operations are linear because we are maintaining a mapping for the positions at where our values exist within the heap.

4.3 Remembering Binary Heaps

Recall that a very common way to represent a binary heap is with an array, since every node is indexed sequentially.

Whenever we want to insert a value in our priority queue we have to do it **in the insertion position a the bottom right** of the binary tree. Doing this ensures that a **complete tree structure** is always maintained.

Once we have inserted it, we have to check if the **heap invariant** is preserved. If not, we have to bubble up our value until it is preserved, swapping up values until we are fine.

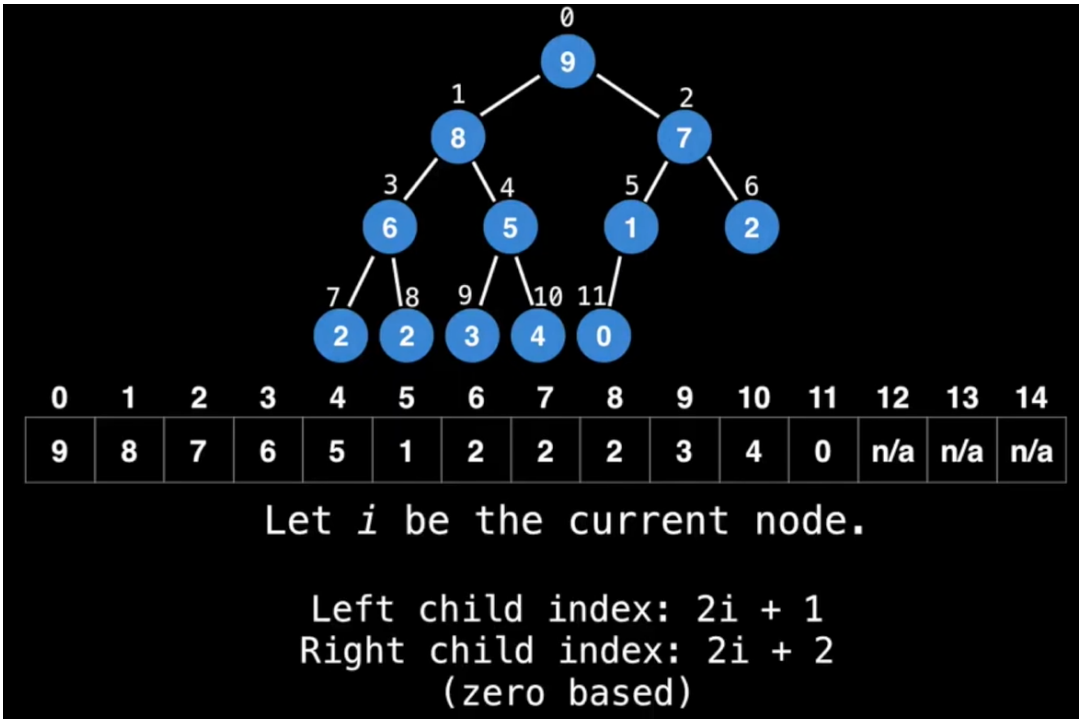


Figure 15:

On the other hand, in a traditional priority queue, if we want to remove an element, we have to search for the element we want to remove and then swap it with the last node, and then we have to ensure that the **heap invariant** is preserved. So we have to bubble up or bubble down the swapped value.

For more information and some examples, check the Priority Queue section.

4.4 Implementing an Indexed Priority Queue with a Binary Heap

We will suppose that N people with different priorities we need to serve, the lower the priority value the higher priority we want to give that person. We will assume that priorities can dynamically change.

To figure out who to serve next we will use a **Min Indexed Priority Queue**, to sort by lowest first.

The first step always consists on arbitrarily assigning each person a unique index value between $[0, N)$. These are the key index values in the second columns besides each persons name (key).

Then, we have to place initial values inside the IPQ. This will be maintained by the IPQ once inserted. Note that values can be any comparable value, not only integers.

When we insert (ki, v) paints into an IPQ, we sort by the value associated with each key.

In the heap below we are sorting by smallest value, since we are working with a min heap.

If we want to access a value for a given key k, we first have to figure out what its key index ki

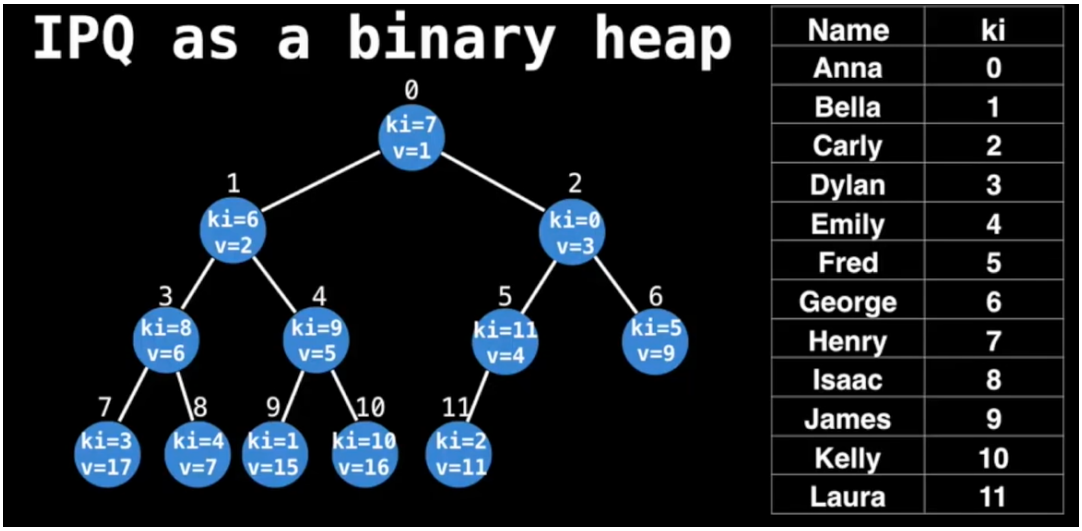


Figure 16:

is, and then we will lookup in our **values** array maintained by our indexed priority queue.

How do we get the index of the node for a particular key?
We will need to maintain some additional information: a **position map (pm)** we can use to tell us the index of a node in the heap for any given key index.

For convenience, we will store the position map as an array called *pm* inside the priority queue:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	-1	-1	-1
pm	2	9	11	7	8	6	1	0	3	4	10	5	-1	-1	-1

Figure 17:

For instance, 'Dylan' has a key index of 3, so $pm[3] = 7$, so we know where Dylan is in the heap, in this case is in the node at index 7.

And, the key index ki of George is 6, so we know that the position index of George is $pm[6] = 1$, so George is found at position index 1, which is the left node of the root node.

How do we go from knowing the position of a node to its key and ki value?
This inverse lookup is quite an useful operation, for example if we want to node the key associated to the root node, for instance, which we know that it is located at index 0.

To be able to do this, we will need to maintain an **inverse lookup table**, which can also be known as **inverse map (im)**.

For example in the node at index 2, we see that we have $ki = 0$, since from the inverse map array we get the associated key index.

Now, since we know the key index, we can do a lookup in our **bidirectional** hashtable and find the element's key, which in this example represents the key 'Anna'.

For instance, we know that the node located at index 3 belongs to the key 'Isaac'.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	-1	-1	-1
pm	2	9	11	7	8	6	1	0	3	4	10	5	-1	-1	-1
im	7	6	0	8	9	11	5	3	4	1	10	2	-1	-1	-1

Figure 18:

4.5 Insertion in PIQ

Insertion is almost similar to what we have seen until now. But we will have to **update the position map** *pm* **and the inverse map** *im* **to reflect the movement of the key-value pairs**.

Supposing we want to insert the key 'Mary' in our priority queue, we first have to assign a unique key index *ki* value, for instance 12, and then we will insert the new key-value pair at the insertion position, which is at the lower right of the tree.

When inserting we have to update our arrays (*vals*, *pm* and *im*) at index 12.

Then we have to check if the heap invariant is satisfied or not. In our example it is not, since we have a value which is lower than the one we have at index 5 (look at Figure 16).

So, we have to swap the newely inserted value upwards until the heap invariant is satisfied. **When we do the swapping we have to update the position and inverse maps** *pm* **and** *im*.

The values array does not to be touched, since it gets indexed by the key-index value that we get from the map, and not from the node index per-se, as they do the *pm* **and** *im* **arrays.** .

The heap invariant is not satisfied yet in our example, so we need to keep swapping upwards and do the necessary changes in the arrays. We have to do this until the heap invariant is satisfied.

Insertion pseudo code

```
# Inserts a value into the min indexed binary heap.
# The key index must not already be in the heap
# and the value must not be null.
function insert(ki, value):
    values[ki] = value
    # 'sz' is the current size of the heap
    pm[ki] = sz
    im[sz] = ki
    swim(sz)
    sz = sz + 1
```

In this code, the first thing we do is to store the value associated with the key inside the values array.

Then we update the position map and the inverse position map to reflect the fact that a new key-value pair has been inserted in the priority queue.

Finally, we move the node up the heap until the heap invariant is satisfied and we augment the variable which stores the current size of the heap. This is done using the following *swim* method:

```
# Swims up node i (zero based) until heap
# invariant is satisfied.
function swim(i):
    for (p = (i-1)/2; i > 0 and less(i, p)):
        swap(i, p)
        i = p
        p = (i-1)/2

function swap(i, j):
    pm[im[j]] = i
    pm[im[i]] = j

    tmp = im[i]
    im[i] = im[j]
    im[j] = tmp

function less(i, j):
    return values[im[i]] < values[im[j]]
```

The swim function begins by finding the index of the parent node ($(i-1)/2$) and walking up the tree ($i > 0$) *and* $less(i, p)$. For every iteration we walk up one layer in the tree if the index of the current node is not the root node and the value of the current node is less than parent node (**remember that this is for the case of the MIN heap**, so we want the small values to be as high as possible in the heap).

Then, to get the node exchange we have to call the *swap* function and provide the index of the current node and the parent node, and then we have to update the current node and the parent node index values ($i = p$ and $p = (i-1)/2$).

The swap method is slightly different than the traditional priority queue. Now we are not actually moving around the values in the array, we are only swapping around index values. This is because **the values array is indexed by the KEY INDEX (ki), not the node index**. So the values array can remain constant while we update the position map pm and the inverse map im.

In the swap method, we first update the positions of where key index values ki in the indexed priority queue. We have to do this by looking at the position map the value given by the search in the inverse map. This is this way because **the position map is the position of which node index a given key index ki is found at**, so we can do a straightforward swap by indexing into the inverse map im to find the key index values and swap the indexes i and j.

Then, we just have to update the key index values associated with the nodes i and j in the inverse map.

4.6 Polling & Removals

Pooling is still $0(log(n))$ in a IPQ, but removing is improved from $0(n)$ in a traditional PQ to $0(log(n))$, since node position lookups are $0(1)$, but repositioning is still $0(log(n))$.

Here is an example of how we remove the root node: (in the example we have removed the root, and then selected to bubble down our node to the left, in order to preserve the heap invariant).

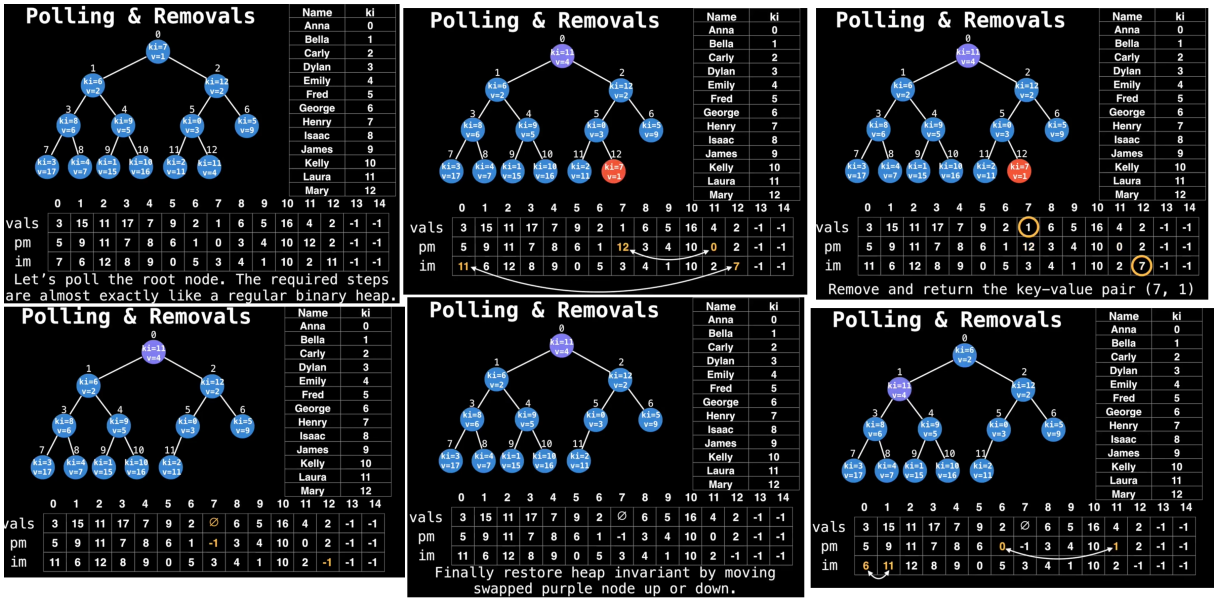


Figure 19:

4.7 Removal Pseudo Code

```
# Deletes the node with the key index ki
# in the heap. The key index ki must exist
# and be present in the heap
function remove(ki):
    i = pm[ki]
    swap(i, sz)
    sz = sz - 1
    sink(i)
    swim(i)
    values[ki] = null
    pm[ki] = -1
    im[sz] = -1
```

The first thing to do is exchange the position of the node we want to remove and the bottom right node, which is always at index position sz, which is the size of our heap.

Once we have exchanged the nodes, the node which was it the position bottom right is now at the position of the node we have to remove, so we need to move it either up or down the heap in order to satisfy the heap invariant.

We do not know if we have to bubble it up or bubble it down, so we first try to move the node up and then down, so if it can go up it will go up and otherwise it will go down (if it can go up it will go and then when trying to swim it, it will remain at the same position since it will see that the heap invariant is satisfied. If it can't go up it will go down. And if the heap invariant is already satisfied, it will be seen by both functions and neither of them will move the node).

Lastly we just have to clean up the values associated to the node we want to remove. We can also return the key-value pair we want to return.

The pseudo code for the sink method is the following:

```
# Sink the node at index i by swapping
# itself with the smallest of the left
# or the right children node.
function sink(i):
    while true:
        left  = 2*i + 1
        right = 2*i +2
        smallest = left

        if right < sz and less(right, left):
            smallest = right

        if left >= sz or less(i, smallest):
            break

        swap(smallest, i)
        i= smallest
```

To sink a node we first want to select the child with smallest value and move it to the left child if there is a tie.

In the next block we try to update the smallest child to be the right child. But first we have to check if the right child's node index is within the size of the heap and its value is actually less than the one of the left node.

The stopping condition is if we are outside the size of the heap or if we can not sink the node any further.

Lastly, we want to make sure we want to swap the current node with whichever was the node with smallest value.

4.8 Updates

Similar to removals, updates in a min index binary heap also take $0(log(n))$ time, due to $0(1)$ lookup time to find the node and $0(log(n))$ time to adjust where the key-value pair should appear in the heap.

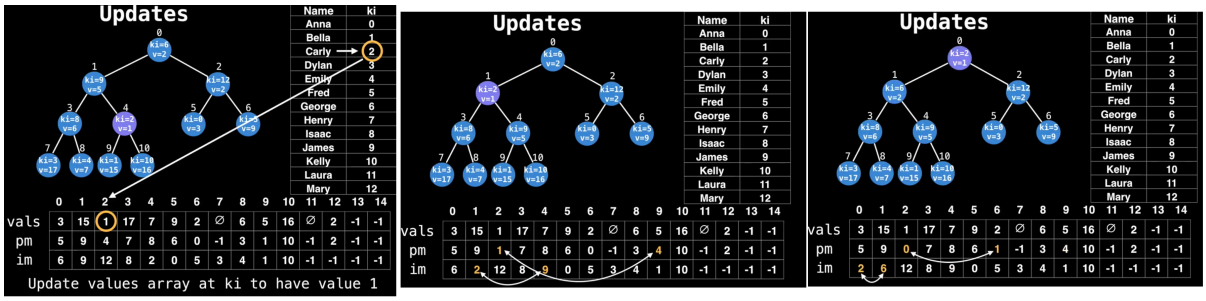


Figure 20:

4.9 Update Pseudo Code

This is the needed pseudo code to update our IPQ:

```
# Updates the value of a key in the binary heap.
# The key must exist and the value must not be null.
function update(ki, value):
    i = pm[ki]
    values[ki] = value
    sink(i)
    swim(i)
```

4.10 Decrease and Increase Key

In many applications (e.g Dijkstra's and Prims algorithm) it is often useful to only update a given key to make its value either always smaller (or larger). In the event that a worse value is given, the value in the IPQ should not be updated.

In such situations it is useful to define a more restrictive form of update operation we call *increaseKey(ki, v)* and *decreaseKey(ki, v)*. These operations are done the following way:

```
# For both these functions assume ki and value
# are valid inputs and we are dealing with a
# MIN indexed binary heap
function decreaseKey(ki, value):
    if less(value, values[ki]):
        values[ki] = value
        swim(pm[ki])

function increaseKey(ki, value):
    if less(values[ki], value):
        values[ki] = value
        sink(pm[ki])
```