**Contribution to Wikipedia**

Username is Azx0987.

The list of Wikipedia articles in which I have contributed are as follows:-

**1. Heap:-**

The Wikipedia link of this article is  <https://en.wikipedia.org/wiki/Heap_(data_structure)>

The things which I have contributed in this Wikipedia page are as follows:-

In the Heap introduction page, I have added:-

In other words, a binary heap is a complete binary tree which satisfies the heap ordering property. The ordering can be one of two types:

1. the min-heap property: the value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root.

2. the max-heap property: the value of each node is less than or equal to the value of its parent, with the maximum-value element at the root.

Throughout this chapter the word ""heap"" will always refer to a min-heap. In a heap the highest (or lowest) priority element is always stored at the root, hence the name ""heap"". A heap is not a sorted structure and can be regarded as partially ordered. As you see from the Heap-diagram, there is no particular relationship among nodes on any given level, even among the siblings. Since a heap is a complete binary tree, it has a smallest possible height - a heap with N nodes always has O(log N) height. A heap is useful data structure when you need to remove the object with the highest (or lowest) priority.

In the implementations and operations section, I have added:-

meld(h1,h2): Return the heap formed by taking the union of the item-disjoint heaps h1 and h2. Melding destroys h1 and h2.

size: return the number of items in the heap.

isEmpty(): returns true if the heap is empty, false otherwise.

buildHeap(list): builds a new heap from a list of keys.

ExtractMin(): Returns the node of minimum value after removing it from the heap

Union(): Creates a new heap by joining two heaps given as input.

Shift-up: Move a node up in the tree, as long as needed (depending on the heap condition: min-heap or max-heap)

Shift-down: Move a node down in the tree, similar to Shift-up.

In the variants section, I have added:-

Ternary Heap

Treap

In the applications section, I have added:-

Priority Queue: A priority queue is an abstract concept like "a list" or "a map"; just as a list can be implemented with a linked list or an array, a priority queue can be implemented with a heap or a variety of other methods.

Order statistics: The Heap data structure can be used to efficiently find the kth smallest (or largest) element in an array.

**2. Binary Search Tree:-**

The Wikipedia link of this article is https://en.wikipedia.org/wiki/Binary\_search\_tree

The things which I have added in Binary Search Tree Wikipedia are as follows:-

I have added 2 headings in the Binary Search tree Wikipedia page with their explanation:-

1.  Determining whether a tree is a BST or not.

2.  Difference between binary tree and binary search tree.

**Determining whether a tree is a BST or not**

Sometimes we already have a binary tree that we need to determine whether it is a BST or not. This is an interesting problem and can be really solved with a simple recursive solution.

The BST property- every node on the right subtree has to be larger than the current node and every node on the left subtree has to be smaller (or equal) than the current node - is the key to figuring out whether a tree is a BST or not. On a first thought it might look like we can simply traverse the tree and at every node check whether the node contains a value larger than the value at the left child and smaller than the value on the right child, and if this condition holds for all the nodes in the tree then we have a BST. This is the so called Greedy approach, making a decision based on local properties. But this approach clearly won't work for the following tree:

     20

    /  \

  10    30

       /  \

      5    40

In the tree above, at every node the condition that the node contains a value larger than its left child and smaller than its right child hold, still its not a BST: the value 5 is on the right subtree of the node containing 20, a violation of the BST property!

So how do we solve this? It turns out that instead of making a decision based solely on a node and its children's values, we also need information flowing down from the parent as well. In the case of the tree above, if we could remember about the node containing the value 20 we could see that the node with value 5 is violating the BST property contract.

So the condition we need to check at each node is that: a) if the node is the left child of its parent, then it must be smaller (or equal) than the parent and it must pass down the value from its parent to its right subtree to make sure none of the nodes in that subtree is greater the parent, and similarly b) if the node is the right child of its parent, then it must be larger than the parent and it must pass down the value from its parent to its left subtree to make sure none of the nodes in that subtree is greater than the parent.

A simple but elegant recursive solution in Java can explain this further:

   public static boolean isBST(TreeNode node, int leftData, int rightData)

   {

   if (node == null)

       return true;

   if (node.getData() > leftData || node.getData() <= rightData)

       return false;

   return (isBST(node.left, node.getData(), rightData) && isBST(node.right, leftData, node.getData()));

   }

The initial call to this function can be something like this:

if(isBST(root, Integer.MAX\_VALUE, Integer.MIN\_VALUE))

    System.out.println("This is a BST.");

else

    System.out.println("This is NOT a BST!");

Essentially we keep creating a valid range (starting from [ MIN\_VALUE, MAX\_VALUE]) and keep shrinking it down foe each node as we go down recursively.

**Difference between binary tree and binary search tree**:-

Binary tree: In short, a binary tree is a tree where each node has up to two leaves. In binary tree, a left child node and a right child node contains value which can be either greater, less, or equal to parent node.

    3

  /   \

4     5

Binary Search Tree: In binary search tree, the left child contains nodes with values less than the parent node and where the right child only contains nodes with values greater than or equal to the parent.

    4

   / \

  3   5

In the Binary-search-tree property, I have added two points:-

1. For each node x, every value found in the left subtree of x is less than or equal to the value found in x.

2. For each node x, every value found in the right subtree of x is greater than or equal to the value found in x.

In the See also section, I have added Red Black Trees and AVL Trees.

**See also:**

Red black trees

AVL Trees

In the introduction part of BST Wikipedia page, the old content was(old one):-

In computer science, a binary search tree (BST), sometimes also called an ordered or sorted binary tree, is a node-based binary tree data structure which has the following properties:[1]

The left subtree of a node contains only nodes with keys less than the node's key.

The right subtree of a node contains only nodes with keys greater than the node's key.

The left and right subtree each must also be a binary search tree.

There must be no duplicate nodes.

Generally, the information represented by each node is a record rather than a single data element. However, for sequencing purposes, nodes are compared according to their keys rather than any part of their associated records.

The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient. Binary search trees are a fundamental data structure used to construct more abstract data structures such as sets, multisets, and associative arrays.

The changes which I have made in the introduction part of the BST Wikipedia page are as follows(new one):-

A binary search tree (BST) is a binary tree where each node has a Comparable key (and an associated value) and satisfies the restriction that the key in any node is larger than the keys in all nodes in that node's left subtree and smaller than the keys in all nodes in that node's right sub-tree. A binary search tree (BST), also known as an ordered binary tree, is a node-based data structure in which each node has no more than two child nodes. Each child must either be a leaf node or the root of another binary search tree. The left sub-tree contains only nodes with keys less than the parent node; the right sub-tree contains only nodes with keys greater than the parent node. In computer science, a binary search tree (BST), sometimes also called an ordered or sorted binary tree. It is also a dynamic data structure, which means, that its size is only limited by amount of free memory in the operating system and number of elements may vary during the program run. Main advantage of binary search trees is rapid search, while addition is quite cheap. The common properties of Binary Search Tree are as follows:-:[1]

The left subtree of a node contains only nodes with keys less than the node's key.

The right subtree of a node contains only nodes with keys greater than the node's key.

The left and right subtree each must also be a binary search tree.

Each node can have up to two successor nodes.

There must be no duplicate nodes.

A unique path exists from the root to every other node.

Generally, the information represented by each node is a record rather than a single data element. However, for sequencing purposes, nodes are compared according to their keys rather than any part of their associated records.

The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient. The other advantages are:-

Binary Search Tree is fast in insertion and deletion etc when balanced.

Very efficient and its code is easier than other data structures.

Stores keys in the nodes in a way that searching, insertion and deletion can be done efficiently.

Implementation is very simple in Binary Search Trees.

Nodes in tree are dynamic in nature.

Binary search trees are a fundamental data structure used to construct more abstract data structures such as sets, multisets, and associative arrays. Some of their disadvantages are as follows:

The shape of the binary search tree totally depends on the order of insertions, and it can be degenerated.

When inserting or searching for an element in binary search tree, the key of each visited node has to be compared with the key of the element to be inserted or found, i.e., it takes a long time to search an element in a binary search tree.

The keys in the binary search tree may be long and the run time may increase.

The old content of Compete tree and a Degenerate tree in BST types in BST Wikipedia page was(old one):-

Two other titles describing binary search trees are that of a complete and degenerate tree.

A complete tree is a tree with n levels, where for each level d <= n - 1, the number of existing nodes at level d is equal to 2d. This means all possible nodes exist at these levels. An additional requirement for a complete binary tree is that for the nth level, while every node does not have to exist, the nodes that do exist must fill from left to right.

A degenerate tree is a tree where for each parent node, there is only one associated child node. What this means is that in a performance measurement, the tree will essentially behave like a linked list data structure.

The changes which I have made in  Compete tree and a Degenerate tree in BST types section in BST Wikipedia page is(new one):-

A complete binary tree is a binary tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right. In complete binary tree, all nodes are far left as possible. It is a tree with n levels, where for each level d <= n - 1, the number of existing nodes at level d is equal to 2d. This means all possible nodes exist at these levels. An additional requirement for a complete binary tree is that for the nth level, while every node does not have to exist, the nodes that do exist must fill from left to right.

A degenerate tree is a tree where for each parent node, there is only one associated child node. It is unbalanced and, in the worst case, performance degrades to that of a linked list. If your added node function does not handle re-balancing, then you can easily construct a degenerate tree by feeding it with data that is already sorted. What this means is that in a performance measurement, the tree will essentially behave like a linked list data structure.

**3. Linked List:-**

The Wikipedia link of this article is https://en.wikipedia.org/wiki/Linked\_list

The things which I have added in the linked list Wikipedia page are as follows:-

I have added advantages of disadvantages of using Linked List.

The advantages and disadvantages of using linked list are as follows:-

Advantages:-

It is a dynamic data structure. The memory is created at run time while running a program.

Insertion and deletion nodes operations are very easy in linked list. We can insert node and delete node easily.

Linear data structures such as stacks and queues can be easily implemented using linked list.

The access time is faster in linked list and it can be expanded in constant time without memory overhead.

Disadvantages:-

Wastage of memory takes place in linked list as pointers requires extra memory for storage.

In Linked list no random access is given to user, we have to access each node sequentially.

In linked list, it takes a lot of time to access an element as individual nodes are not stored in contiguous memory allocations.

Reverse traversing is very difficult in linked list. In case if we are using singly linked list then it is very difficult to traverse linked list from end. If using doubly linked list then though it becomes easier to traverse from end but still it increases again storage space for back pointer.

I have also added, When to use LinkedList over ArrayList ?

When to use LinkedList over ArrayList ?[edit]

LinkedList and ArrayList are 2 distinct implementations of the List interface. LinkedList implements it with a doubly-linked list whereas ArrayList implements it with a dynamically

resizing array. As with standard linked list and array operations, the various methods will have different algorithmic runtimes.

For LinkedList<E> :-

get(int index) is O(n)

add(E element) is O(1)

add(int index, E element) is O(n)

remove(int index) is O(n)

Iterator.remove() is O(1) <--- main benefit of LinkedList<E>

ListIterator.add(E element) is O(1) <--- main benefit of LinkedList<E>

For ArrayList<E> :-

get(int index) is O(1) <--- main benefit of ArrayList<E>

add(E element) is O(1) amortized, but O(n) worst-case since the array must be resized and copied

add(int index, E element) is O(n - index) amortized, but O(n) worst-case (as above)

remove(int index) is O(n - index) (i.e. removing last is O(1))

Iterator.remove() is O(n - index)

ListIterator.add(E element) is O(n - index)

LinkedList<E> allows for constant-time insertions or removals using iterators, but only sequential access of elements. In other words, you can walk the list forwards or backwards, but finding a position in the list takes time proportional to the size of the list.

ArrayList<E>, on the other hand, allow fast random read access, so you can grab any element in constant time. But adding or removing from anywhere but the end requires shifting all the latter elements over, either to make an opening or fill the gap. Also, if you add more elements than the capacity of the underlying array, a new array (twice the size) is allocated, and the old array is

copied to the new one, so adding to an ArrayList is O(n) in the worst case but constant on average.

So depending on the operations you intend to do, you should choose the implementations accordingly. Iterating over either kind of List is practically equally cheap. (Iterating over an ArrayList is technically faster, but unless you're doing something really performance-sensitive, you shouldn't worry about this -- they're both constants.)

The main advantages of using a LinkedList arise when you re-use existing iterators to insert and remove elements. These operations can then be done in O(1) by changing the list locally only. In an array list, the remainder of the array needs to be moved (i.e. copied). On the other side, seeking in a LinkedList means following the links in O(n), whereas in an ArrayList the desired position can be computed mathematically and accessed in O(1).

**4. LCP Array :-**

The Wikipedia link of this article is https://en.wikipedia.org/wiki/LCP\_array

The things which I have added in the LCP Array Wikipedia page are as follows:-

In the introduction part I have added:-

In other words, it is the length of prefix that is common between the two consecutive suffixes in an sorted suffix array. To simplify , see this way , we have created an suffix array and we have ordered suffixes . We need to find out the number of consecutive characters common in the two suffixes (from starting of suffix).

example:-

LCP of a and aabba is 1.

LCP of abaabba and abba is 2.

In the history part I have added:-

Gene Myers: Former V.P. (Vice president) of Informatics Research at Celera Genomics

Udi Manber: V.P. (Vice president) of engineering at Google.

I have also added 2 headings with their explanation. The First heading which I have added is Difference between Suffix Array and LCP Array ?

Difference between Suffix Array and LCP Array ?

Suffix array : Represents the lexicographic rank of each suffix of an array.

LCP array : Contains the maximum length prefix match between two consecutive suffixes, after they are sorted lexicographically.

The Second heading which I have added is LCP Array usage in finding the number of occurrences of a pattern.

LCP Array usage in finding the number of occurrences of a pattern

In order to find the number of occurrences of a given string P (length m) in a text T (length N),

You must use binary search against the suffix array of T.

You should speed up the LCP array usage as an auxiliary data structure. More specifically, you generate a special version of the LCP array (LCP-LR below) and use that.

The issue with using standard binary search (without the LCP information) is that in each of the O(log N) comparisons you need to make, you compare P to the current entry of the suffix array, which means a full string comparison of up to m characters. So the complexity is O(m\*log N).

The LCP-LR array helps improve this to O(m+log N), in the following way:

At any point during the binary search algorithm, you consider, as usual, a range (L,...,R) of the suffix array and its central point M, and decide whether you continue your search in the left sub-range (L,...,M) or in the right sub-range (M,...,R). In order to make the decision, you compare P to the string at M. If P is identical to M, you are done, but if not, you will have compared the first k characters of P and then decided whether P is lexicographically smaller or larger than M. Let's assume the outcome is that P is larger than M. So, in the next step, you consider (M,...,R) and a new central point M' in the middle:

             M ...... M' ...... R

             |

      we know:

         lcp(P,M)==k

The trick now is that LCP-LR is precomputed such that a O(1)-lookup tells you the longest common prefix of M and M', lcp(M,M').

You know already (from the previous step) that M itself has a prefix of k characters in common with P: lcp(P,M)=k. Now there are three possibilities:

Case 1: k < lcp(M,M'), i.e. P has fewer prefix characters in common with M than M has in common with M'. This means the (k+1)-th character of M' is the same as that of M, and since P is lexicographically larger than M, it must be lexicographically larger than M', too. So we continue in the right half (M',...,R).

Case 2: k > lcp(M,M'), i.e. P has more prefix characters in common with M than M has in common with M'. Consequently, if we were to compare P to M', the common prefix would be smaller than k, and M' would be lexicographically larger than P, so, without actually making the comparison, we continue in the left half (M,...,M').

Case 3: k == lcp(M,M'). So M and M' are both identical with P in the first k characters. To decide whether we continue in the left or right half, it suffices to compare P to M' starting from the (k+1)-th character.

We continue recursively.

The overall effect is that no character of P is compared to any character of the text more than once. The total number of character comparisons is bounded by m, so the total complexity is indeed O(m+log N).

Obviously, the key remaining question is how did we precompute LCP-LR so it is able to tell us in O(1) time the lcp between any two entries of the suffix array? As you said, the standard LCP array tells you the lcp of consecutive entries only, i.e. lcp(x-1,x) for any x. But M and M' in the description above are not necessarily consecutive entries, so how is that done?

The key to this is to realize that only certain ranges (L,...,R) will ever occur during the binary search: It always starts with (0,...,N) and divides that at the center, and then continues either left or right and divide that half again and so forth. If you think of it: Every entry of the suffix array occurs as central point of exactly one possible range during binary search. So there are exactly N distinct ranges (L...M...R) that can possibly play a role during binary search, and it suffices to precompute lcp(L,M) and lcp(M,R) for those N possible ranges. So that is 2\*N distinct precomputed values, hence LCP-LR is O(N) in size.

Moreover, there is a straight-forward recursive algorithm to compute the 2\*N values of LCP-LR in O(N) time from the standard LCP array – I'd suggest posting a separate question if you need a detailed description of that.

To sum up:

It is possible to compute LCP-LR in O(N) time and O(2\*N)=O(N) space from LCP.

Using LCP-LR during binary search helps accelerate the search procedure from O(M\*log N) to O(M+log N).

You can use two binary searches to determine the left and right end of the match range for P, and the length of the match range corresponds with the number of occurrences for P.