SYLLABUS

Tree Algorithms: Fenwick Tree, Segment Tree – Applications-Range Sum Queries. Treap- Applications- Kth Largest Element in an Array.

Trie: Introduction, Suffix Tree, Applications-Index Pairs of a String, Longest word with all prefixes, top K frequent words.

Tree Algorithms

Applications

1. **Fenwick Tree.**
2. **Segment Tree – Applications-Range Sum Queries.**
3. **Treap- Applications- Kth Largest Element in an Array.**

Fenwick Tree

Fenwick Tree- also known as **Binary Indexed Tree (BIT)**—were invented by *Peter M.Fenwick* in 1994.

Fenwick Tree provides a way to represent an array of numbers in an array, allowing prefix sums to be calculated efficiently.

For example, an array is [2, 3, -1, 0, 6] the length 3 prefix [2, 3, -1] with sum 2 + 3 + -1 = 4. Calculating prefix sums efficiently is useful in various scenarios.

Let's start with a simple problem.

We are given an array a[], and we want to be able to perform two types of operations on it.

1. *Change the value* stored at an index i. (This is called a point update operation)
2. *Find the sum of a prefix* of length k. (This is called a range sum query)

A straightforward implementation of the above would look like this.

```
int a[] = {2, 1, 4, 6, -1, 5, -32, 0, 1};
void update(int i, int v)   //assigns value v to a[i]
{
   a[i] = v;
}
int prefixsum(int k)    //calculate the sum of all a[i] such that 0 <= i < k
{
  int sum = 0;
  for(int i = 0; i < k; i++)
      sum += a[i];
  return sum;
}
```

This is a perfect solution, but unfortunately the time required to calculate a prefix sum is proportional to the length of the array, so this will usually time out when large number of such intermingled operations are performed.

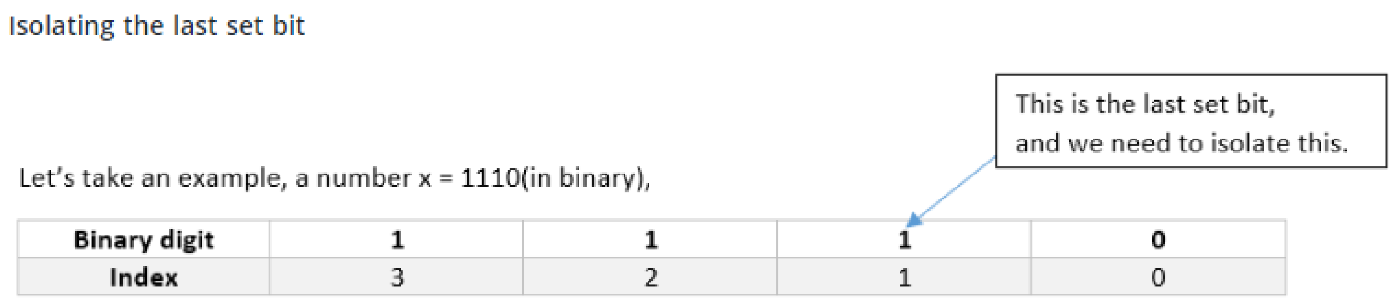
Can we do better than this? Off course.

One efficient solution is to use *segment tree* that can perform both operation in O(logN) time.

Using *binary Indexed tree* also, we can perform both the tasks in O(logN) time.

But then why learn another data structure when segment tree can do the work for us. It’s because *binary indexed trees require less space and are* ***very easy to implement*** *during programming contests*.

Before starting with binary indexed tree, we need to understand a particular bit manipulation trick. Here it goes.



Isolating the last set bit

Let’s take an example, a number x = 1110(in binary),

Binary digit	1	1	1	0
Index	3	2	1	0

This is the last set bit, and we need to isolate this.

How to isolate?

x & (-x) gives the last set bit in a number x. How?

Example:-

x = 10(*in decimal*) = 1010(*in binary*)

The last set bit is given by x & (-x) = (10)1(0) & (01)1(0) = 0010 = 2(in decimal)

<u>**Basic Idea of Binary Indexed Tree:**</u>

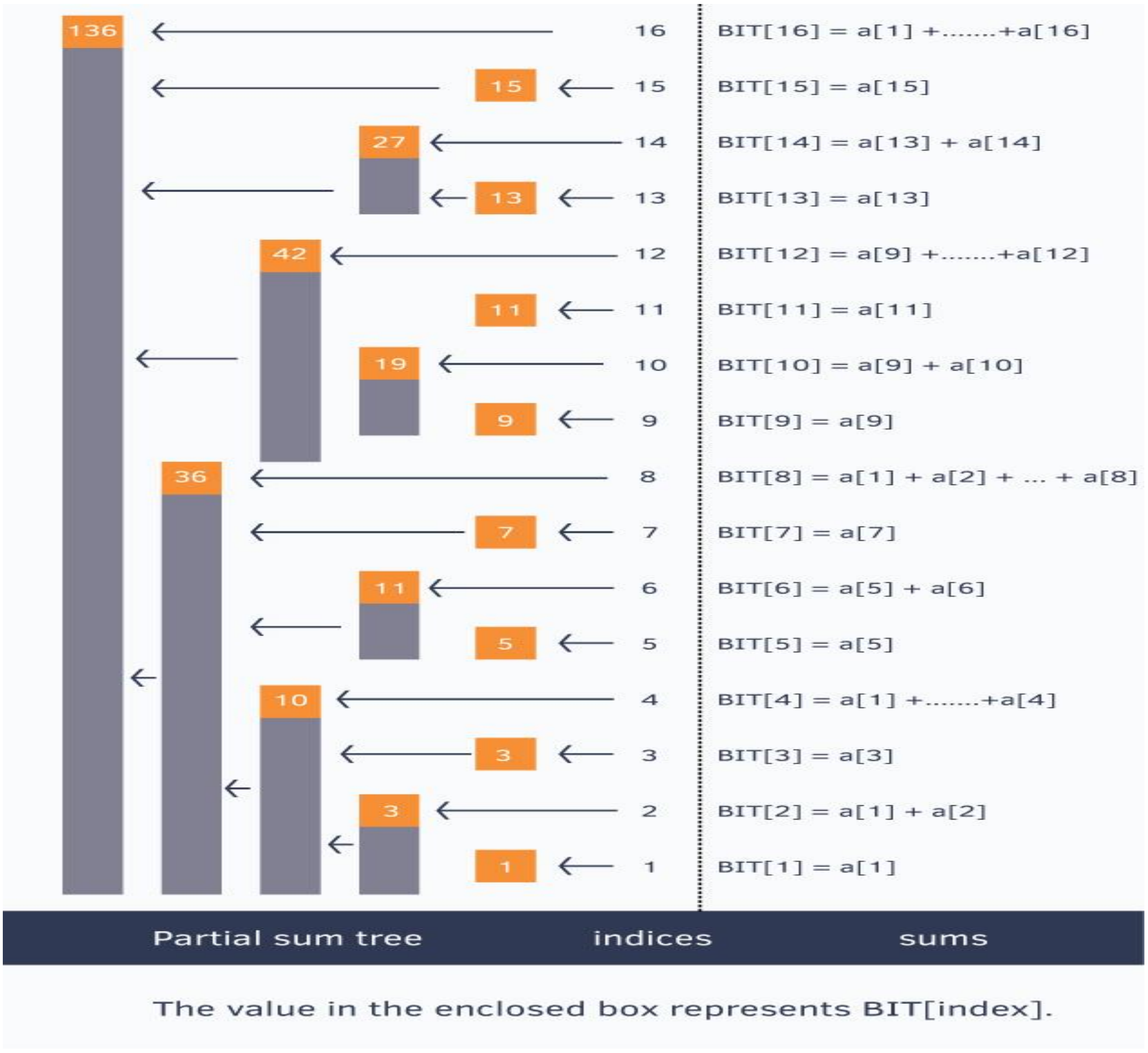
We know the fact that each integer can be represented as sum of powers of two.

Similarly, for a given array of size N, we can maintain an array BIT[] such that, at any index we can store sum of some numbers of the given array. This can also be called a partial sum tree.

Let’s use an example to understand how BIT[] stores partial sums.

//for ease, we make sure our given array is 1-based indexed

int a[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};



The above picture shows the binary indexed tree, each enclosed box of which denotes the value BIT[index] and each BIT[index] stores partial sum of some numbers.

Notice

BIT[X]={ a[x] -----if x is odd

a[1]+……….+a[x] ------- if x is power of 2 }

To generalize this, every index i in the BIT[] array stores the cumulative sum from the index **i to i - (1 << r) + 1** (*both inclusive*), where **r** represents the last set bit in the index i

Applications

Fenwick tree can be used to **calculate Range Sum,** i.e., finding the sum within range. Sum(1,7) in the below example array is

Value	5	2	9	-3	5	20	10	-7	2	3	-4	0	-2	15	5
Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

5 + 2 + 9 + (-3) + 5 + 20 + 10 = 48 in traditional way, but by using Fenwick tree we can do this in minimum time. The following method demonstrates that Range Sum Calculation.

```
int sum(int i)
{
        int sum = 0;
        while ( i > 0 )
        {
                sum +=BIT[i];
                i -= i & -i;        //flip the last set bit
        }
        return sum;
}
```

Example: - to find sum from the range 1 to 7 we use

Sum(7) = sum(00111)

= BIT[00111] + BIT[00110] +BI T[00100]

= BIT[7] +BIT[6] + BIT[4]

= range(7,7) + range(5,6) + range(1,4)

= 10 + 25 + 12 =48

To compute the sum (7) while loop iterates only three times

Sum (8) = sum (01000)

= BIT [01000]

= BIT [8]

=range (8,8) =41

To compute sum(8) while loop iterates only once. This is faster to compute the range sums in large arrays.

Updation

When we Update the array recomputing the Fenwick tree will not be too costly, see the below code for updation.

```
void add(int i, int k)
 {
        while (i < T.length)
        {
                T[i] += k;
                i += i & -i;      // add last set bit
        }
}
```

Example :- add(4,10) i.e., making index 4 value to be 7

BIT[00100] = 13 + 10 =23

BIT[01000] = 41 + 10 = 51

BIT[10000] is out of array index, function holds.

<u>Java Program to implement Fenwick tree</u>: FenWickTree.java

```
import java.util.*;
class FenWickTree
{
        int[] nums;
        int[] BIT;
        int n;
        public FenWickTree(int[] nums)
        {
                this.nums = nums;
                n = nums.length;
                BIT = new int[n + 1];
                for (int i = 0; i < n; i++)
                        init(i, nums[i]);
        }
        public void init(int i, int val)
        {
                i++;
                while (i <= n)
                {
                        BIT[i] += val;
                        i += (i & -i);
                }
        }
        void update(int i, int val)
        {
                int diff = val - nums[i];
                nums[i] = val;
                init(i, diff);
        }

        public int getSum(int i)
        {
                int sum = 0;
                i++;
                while (i > 0)
                {
                        sum += BIT[i];
                        i -= (i & -i);

                }
                return sum;
        }

        public int sumRange(int i, int j)
        {
                return getSum(j) - getSum(i - 1);
        }
```

```
	public static void main(String args[] )
	{
		Scanner scan = new Scanner(System.in);
		int n=scan.nextInt();
		int q=scan.nextInt();
		int[] nums=new int[n];
		for(int i=0; i<n; i++)
		{
			nums[i] = scan.nextInt();
		}
		FenWickTree ft =new FenWickTree(nums);
		while(q-->0)
		{
			int opt=scan.nextInt();
			if(opt==1)
			{
				int s1 = scan.nextInt();
				int s2 = scan.nextInt();
				System.out.println(ft.sumRange(s1,s2));
			}
			else
			{
				int ind = scan.nextInt();
				int val= scan.nextInt();
				ft.update(ind,val);
			}
		}
	}
}
```

Sample input:

```
8 5
1 2 13 4 25 16 17 8
1 2 6
1 0 7
2 2 18
2 4 17
1 2 7
```

Output:

```
 75
 86
 80
```

Segment Tree

- Segment tree or Segtree is basically a binary tree used for storing the intervals or segments.
- Each node in the segment tree represents an interval.
  Consider an array A of size N and a corresponding Segtree T:
- The root of T will represent the whole array A[0:N-1].
- Each leaf in the Segtree T will represent a single element A[i] such that 0 <= i < N.
- The internal nodes in the Segtree T represent union of elementary intervals A[i:j] where 0 <= i < j < N.
- The root of the Segtree will represent the whole array A[0:N-1]. Then we will break the interval or segment into half and the two children of the root will represent the A[0:(N-1) / 2] and A[(N-1) / 2 + 1:(N-1)].

- So, in each step we will divide the interval into half and the two children will represent the two halves, so the height of the segment tree will be log2N. There are N leaves representing the N elements of the array. The number of internal nodes is N-1. So total number of nodes are 2*N - 1.

- Once we have built a Segtree we cannot change its structure i.e., its structure is static. We can update the values of nodes but we cannot change its structure. Segment tree is recursive in nature. Because of its recursive nature, Segment tree is very easy to implement. Segment tree provides two operations:

- Update: In this operation we can update an element in the Array and reflect the corresponding change in the Segment tree.

- Query: In this operation we can query on an interval or segment and return the answer to the problem on that particular interval.

Implementation:

Since a segment tree is a binary tree, we can use a simple linear array to represent the segment tree. In almost any segment tree problem we need to think about what we need to store in the segment tree?

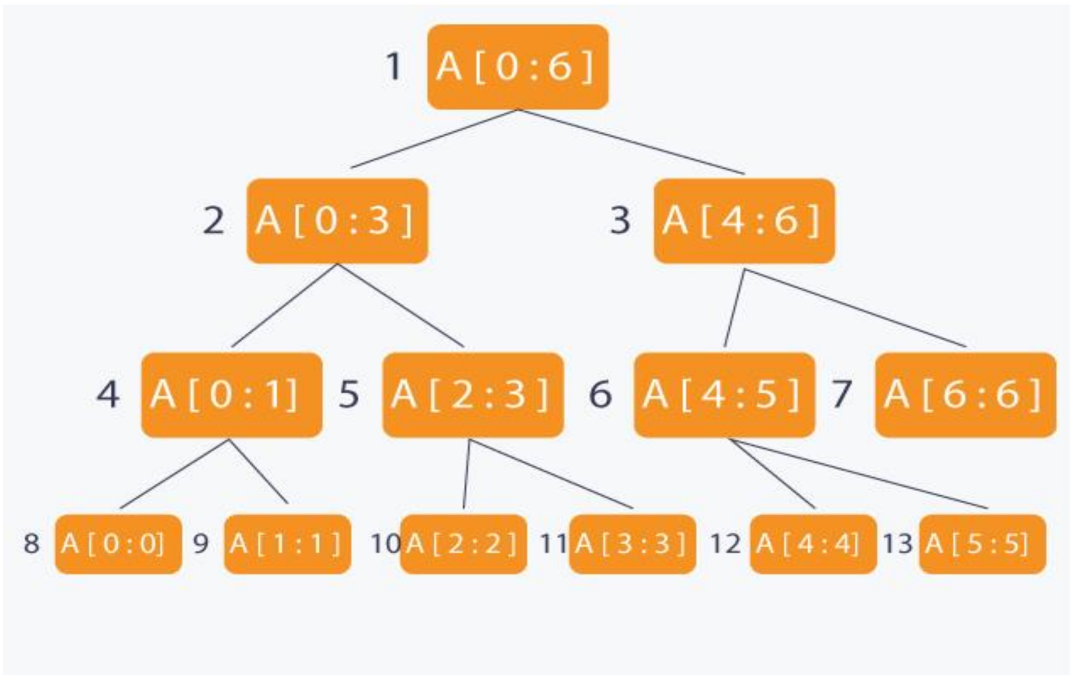
For example, if we want to find the sum of all the elements in an array from index left to right, then at each node (except leaf nodes) we will store the sum of its child nodes. If we want to find the minimum of all the elements in an array from index left to right, then at each node (except leaf nodes) we will store the minimum of its child nodes.

Once we know what we need to store in the segment tree we can build the tree using **recursion (bottom-up approach)**. We will start with the leaves and go up to the root and update the corresponding changes in the nodes that are in the path from leaves to root. Leaves represent a single element. In each step we will merge two children to form an internal node. Each internal node will represent a union of its children’s intervals. Merging may be different for different problems. So, recursion will end up at root which will represent the whole array.

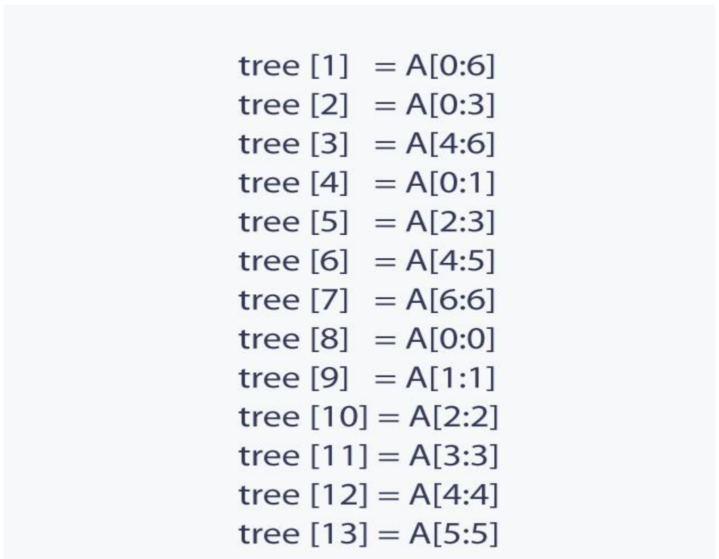
For update, we simply have to search the leaf that contains the element to update. This can be done by going to either on the left child or the right child depending on the interval which contains the element. Once we found the leaf, we will update it and again use the bottom-up approach to update the corresponding change in the path from leaf to root.

To make a query on the segment tree we will be given a range from **l to r**. We will recurse on the tree starting from the root and check if the interval represented by the node is completely in the range from **l to r**. If the interval represented by a node is completely in the range from **l to r**, we will return that node's value.

The segtree of array **A** of size **7** will look like :



Segment Tree



tree [1] = A[0:6]
tree [2] = A[0:3]
tree [3] = A[4:6]
tree [4] = A[0:1]
tree [5] = A[2:3]
tree [6] = A[4:5]
tree [7] = A[6:6]
tree [8] = A[0:0]
tree [9] = A[1:1]
tree [10] = A[2:2]
tree [11] = A[3:3]
tree [12] = A[4:4]
tree [13] = A[5:5]

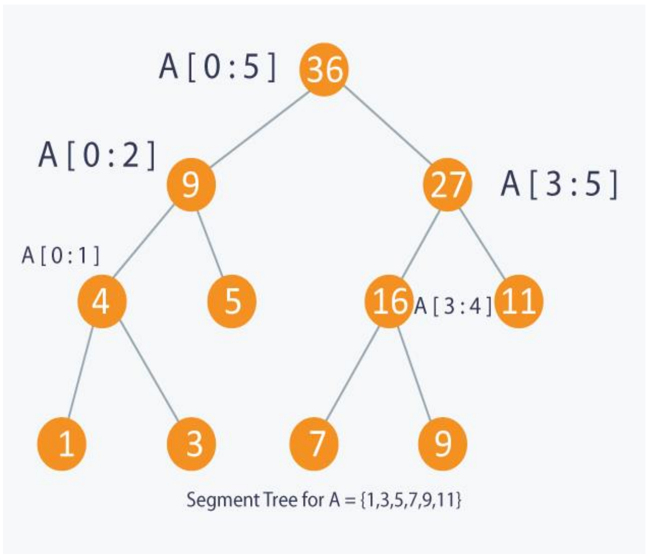
Segment Tree represented as linear array

<u>Applications</u>: - Segment trees are used to find range sum queries.

Let us see how to use segment tree and what we will store in the segment tree in this problem. As we know that each node of the segtree will represent an interval or segment. In this problem we need to find the sum of all the elements in the given range. So, in each node we will store the sum of all the elements of the interval represented by the node. How do we do that? We will build a segment tree using recursion (bottom-up approach) as explained above. Each leaf will have a single element. All the internal nodes will have the sum of both of its children.

```
void build(int node, int start, int end)
{
        if(start == end)
        {
                // Leaf node will have a single element
                tree[node] = A[start];
        }
        else
        {
                int mid = (start + end) / 2;
                // Recurse on the left child
                build(2*node, start, mid);
                // Recurse on the right child
                build(2*node+1, mid+1, end);
                // Internal node will have the sum of both of its children
                tree[node] = tree[2*node] + tree[2*node+1];
        }
}
```

In the above code we will start from the root and recurse on the left and the right child until we reach the leaves. From the leaves we will go back to the root and update all the nodes in the path. **node** represent the current node we are processing. Since segment tree is a binary tree. **2*node** will represent the left node and **2*node + 1** represent the right node. **start** and **end** represents the interval represented by the node. Complexity of build() is **O(N)**.



Segment Tree for A = {1,3,5,7,9,11}

To update an element, we need to look at the interval in which the element is and recurse accordingly on the left or the right child.

```
void update(int node, int start, int end, int idx, int val)
{
   if(start == end)
   {
      // Leaf node
      A[idx] += val;
      tree[node] += val;
   }
   else
   {
      int mid = (start + end) / 2;
      if(start <= idx and idx <= mid)
      {
         // If idx is in the left child, recurse on the left child
         update(2*node, start, mid, idx, val);
      }
      else
      {
         // if idx is in the right child, recurse on the right child
         update(2*node+1, mid+1, end, idx, val);
      }
      // Internal node will have the sum of both of its children
      tree[node] = tree[2*node] + tree[2*node+1];
   }
}
```

Complexity of update will be O(logN).

<u>To query on a given range, we need to check 3 conditions.</u>

1. range represented by a node is completely inside the given range
2. range represented by a node is completely outside the given range
3. range represented by a node is partially inside and partially outside the given range

If the range represented by a node is completely outside the given range, we will simply return 0. If the range represented by a node is completely inside the given range, we will return the value of the node which is the sum of all the elements in the range represented by the node. And if the range represented by a node is partially inside and partially outside the given range, we will return sum of the left child and the right child. Complexity of query will be O(logN).

```
int query(int node, int start, int end, int l, int r)
{
   if(r < start or end < l)
   {
      // range represented by a node is completely outside the given range
      return 0;
   }
```

```
    if(l <= start and end <= r)
    {
        // range represented by a node is completely inside the given range
        return tree[node];
    }
    // range represented by a node is partially inside and partially outside the given range
    int mid = (start + end) / 2;
    int p1 = query(2*node, start, mid, l, r);
    int p2 = query(2*node+1, mid+1, end, l, r);
    return (p1 + p2);
}
```

Updating an interval (Lazy Propagation):

Sometimes problems will ask you to update an interval from **l to r**, instead of a single element. One solution is to update all the elements one by one. Complexity of this approach will be **O(N)** per operation since where are **N** elements in the array and updating a single element will take **O(logN)** time.

To avoid multiple call to update function, we can modify the update function to work on an interval.

```
void updateRange(int node, int start, int end, int l, int r, int val)
{
    // out of range
    if (start > end or start > r or end < l)
        return;
    // Current node is a leaf node
    if (start == end)
    {
        // Add the difference to current node
        tree[node] += val;
        return;
    }
    // If not a leaf node, recur for children.
    int mid = (start + end) / 2;
    updateRange(node*2, start, mid, l, r, val);
    updateRange(node*2 + 1, mid + 1, end, l, r, val);
    // Use the result of children calls to update this node
    tree[node] = tree[node*2] + tree[node*2+1];
}
```

Let's be Lazy i.e., do work only when needed. How? When we need to update an interval, we will update a node and mark its child that it needs to be updated and update it when needed. For this we need an array lazy[] of the same size as that of segment tree. Initially all the elements of the lazy[] array will be 0 representing that there is no pending update. If there is non-zero element lazy[k] then this element needs to update node k in the segment tree before making any query operation.

To update an interval, we will keep 3 things in mind.

1. If current segment tree node has any pending update, then first add that pending update to current node.
2. If the interval represented by current node lies completely in the interval to update, then update the current node and update the lazy[] array for children nodes.
3. If the interval represented by current node overlaps with the interval to update, then update the nodes as the earlier update function

Since we have changed the update function to postpone the update operation, we will have to change the query function also. The only change we need to make is to check if there is any pending update operation on that node. If there is a pending update operation, first update the node and then work same as the earlier query function.

```
void updateRange (int node, int start, int end, int l, int r, int val)
{
   if(lazy[node] != 0)
   {
      // This node needs to be updated
      tree[node] += (end - start + 1) * lazy[node];    // Update it
      if(start != end)
      {
         lazy[node*2] += lazy[node];                // Mark child as lazy
         lazy[node*2+1] += lazy[node];              // Mark child as lazy
      }
      lazy[node] = 0;                               // Reset it
   }
   if(start > end or start > r or end < l)         // Current segment is not within range [l, r]
      return;
   if(start >= l and end <= r)
   {
      // Segment is fully within range
      tree[node] += (end - start + 1) * val;
      if(start != end)
      {
         // Not leaf node
         lazy[node*2] += val;
         lazy[node*2+1] += val;
      }
      return;
   }
   int mid = (start + end) / 2;
   updateRange(node*2, start, mid, l, r, val);        // Updating left child
   updateRange(node*2 + 1, mid + 1, end, l, r, val);   // Updating right child
   tree[node] = tree[node*2] + tree[node*2+1];        // Updating root with max value
}
```

```
int queryRange(int node, int start, int end, int l, int r)
{
   if(start > end or start > r or end < l)
      return 0;         // Out of range
   if(lazy[node] != 0)
   {
      // This node needs to be updated
      tree[node] += (end - start + 1) * lazy[node];          // Update it
      if(start != end)
      {
         lazy[node*2] += lazy[node];         // Mark child as lazy
         lazy[node*2+1] += lazy[node];    // Mark child as lazy
      }
      lazy[node] = 0;                // Reset it
   }
   if(start >= l and end <= r)         // Current segment is totally within range [l, r]
      return tree[node];
   int mid = (start + end) / 2;
   int p1 = queryRange(node*2, start, mid, l, r);         // Query left child
   int p2 = queryRange(node*2 + 1, mid + 1, end, l, r); // Query right child
   return (p1 + p2);
}
```

Java program for segment tree: **SegmentTree.java**

```
import java.util.*;
public class SegmentTree
{
        class SegmentTreeNode
        {
                int start, end;
                SegmentTreeNode left, right;
                int sum;

                public SegmentTreeNode(int start, int end)
                {
                        this.start = start;
                        this.end = end;
                        this.left = null;
                        this.right = null;
                        this.sum = 0;
                }
        }

        SegmentTreeNode root = null;
        public SegmentTree(int[] nums)
        {
                root = buildTree(nums, 0, nums.length-1);
                printTree(root);
        }

        public void printTree(SegmentTreeNode root)
        {
                if(root == null)
                        return;

                printTree(root.left);
                printTree(root.right);
        }

        private SegmentTreeNode buildTree(int[] nums, int start, int end)
        {
                if (start > end)
                {
                        return null;
                }
                else
                {
                        SegmentTreeNode ret = new SegmentTreeNode(start, end);
                        if (start == end)
                        {
                                ret.sum = nums[start];
                        }
                        else
                        {
                                int mid = (start  + end) / 2;
                                ret.left = buildTree(nums, start, mid);
                                ret.right = buildTree(nums, mid + 1, end);
                                ret.sum = ret.left.sum + ret.right.sum;
                        }
                        return ret;
                }
        }

        void update(int i, int val)
        {
                update(root, i, val);
        }

        void update(SegmentTreeNode root, int pos, int val)
        {
                if (root.start == root.end)
                {
                        root.sum = val;
                }
                else
                {
                        int mid = (root.start + root.end) / 2;
                        if (pos <= mid)
                        {
                                update(root.left, pos, val);
                        }
                        else
                        {
                                update(root.right, pos, val);
                        }
                        root.sum = root.left.sum + root.right.sum;
                }
                printTree(root);
        }

        public int sumRange(int i, int j)
        {
                return sumRange(root, i, j);
        }

        public int sumRange(SegmentTreeNode root, int start, int end)
        {
                if (root.start == start && root.end == end)
                {
                        return root.sum;
                }
                else
```

```
                {
                        if (end <= mid)
                        {
                                return sumRange(root.left, start, end);
                        }
                        else if (start >= mid+1)
                        {
                                return sumRange(root.right, start, end);
                        }
                        else
                        {
                                return sumRange(root.left, start, mid) + sumRange(root.right,
                                                                                mid+1, end);
                        }
                }
        }
}

class SegmentTreeTest
{
        public static void main(String args[] )
        {
                Scanner scan = new Scanner(System.in);
                int n=scan.nextInt();
                int q=scan.nextInt();
                int[] nums=new int[n];
                for(int i=0; i<n; i++)
                {
                         nums[i] = scan.nextInt();
                }
                SegmentTree st = new SegmentTree(nums);
                while(q != 0)
                {
                        int opt=scan.nextInt();
                        if(opt==1)
                        {
                                int s1 = scan.nextInt();
                                int s2 = scan.nextInt();
                                System.out.println(st.sumRange(s1,s2));
                        }
                        else
                        {
                                int ind = scan.nextInt();
                                int val= scan.nextInt();
                                st.update(ind, val);
                        }
                        q--;
                }
        }
}
```

Sample input :-

```
8 5
4 2 13 4 25 16 17 8
1 2 6
1 0 7
2 2 18
2 4 17
1 2 7
```

Output :-

```
75
89
80
```

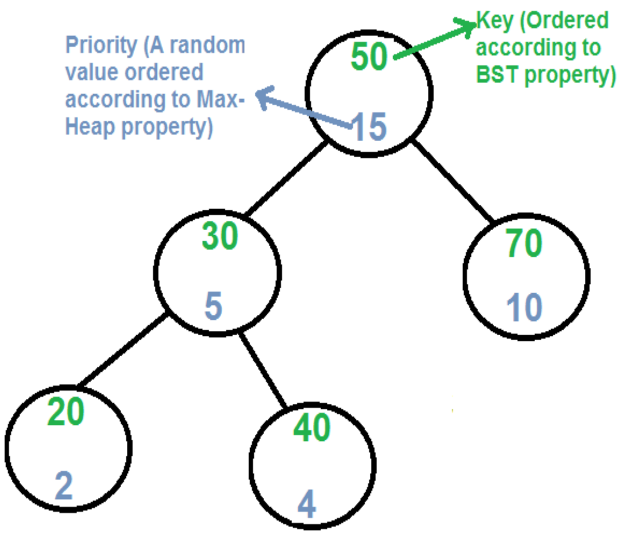
Treap

- A treap is a data structure which combines binary tree and binary heap (hence the name: tree + heap ⇒ Treap).
- Treap is a Balanced Binary Search Tree, but not guaranteed to have height as O(log n).
- The idea is to use Randomization and Binary Heap property to maintain balance with high probability.
- The expected time complexity of search, insert and delete is O(log n).

- More specifically, treap is a data structure that stores pairs(X,Y) in a binary tree in such a way that it is a binary search tree by X and a binary heap by Y.
- If some node of the tree contains values (X_0,Y_0), all nodes in the left subtree have $X<=X_0$ all nodes in the right subtree have $X_0<= X$, and all nodes in both left and right subtrees have $Y<=Y_0$. Here X denotes the key value and Y denotes the priority or weights.

Every node of Treap maintains two values.

1) **Key** Follows standard BST ordering (left is smaller and right is greater)
2) **Priority(or weights)** Randomly assigned value that follows Max-Heap property.


Priority (A random value ordered according to Max-Heap property)
Key (Ordered according to BST property)
50
15
30
5
70
10
20
2
40
4

The following are the features of Treap Data structure.

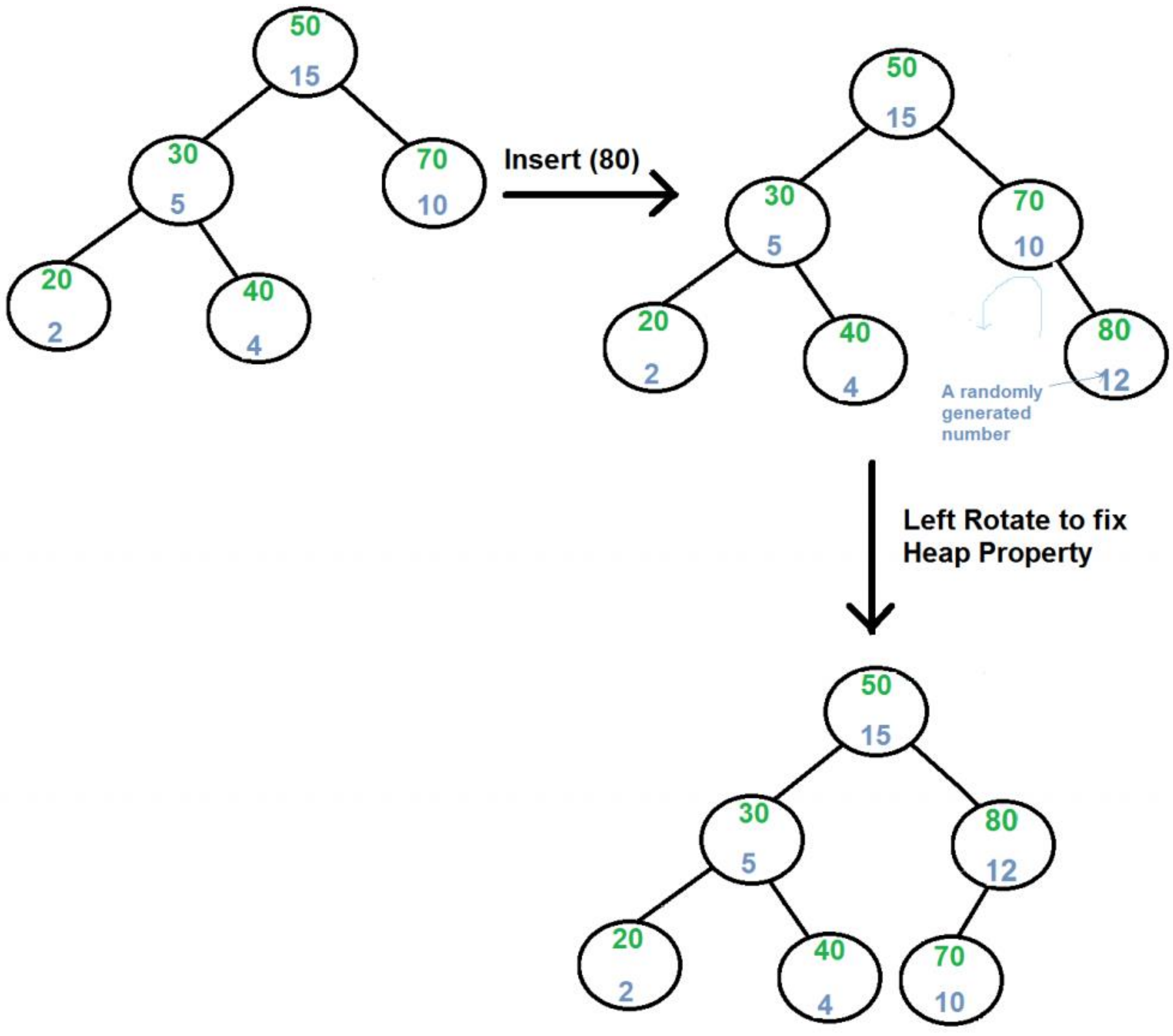
- It is a randomized data structure
- Because of the randomized weights, there is a strong possibility that the tree will be balanced regardless of the sequence in which we add, remove, etc.
- It is simply a binary search tree, therefore to print a sorted order of keys, traverse it in the same way as we would conventional BSTs. Searching for a treap is similar to searching for a tree.

Following are the algorithms for basic operations on Treap:

1. Insertion in Treap

To insert a new key x into the treap, generate a random priority y for x. Binary search for x in the tree, and create a new node at the leaf position where the binary search determines a node for x should exist. Then as long as x is not the root of the tree and has a larger priority number than its parent z, perform a tree rotation that reverses the parent-child relation between x and z.

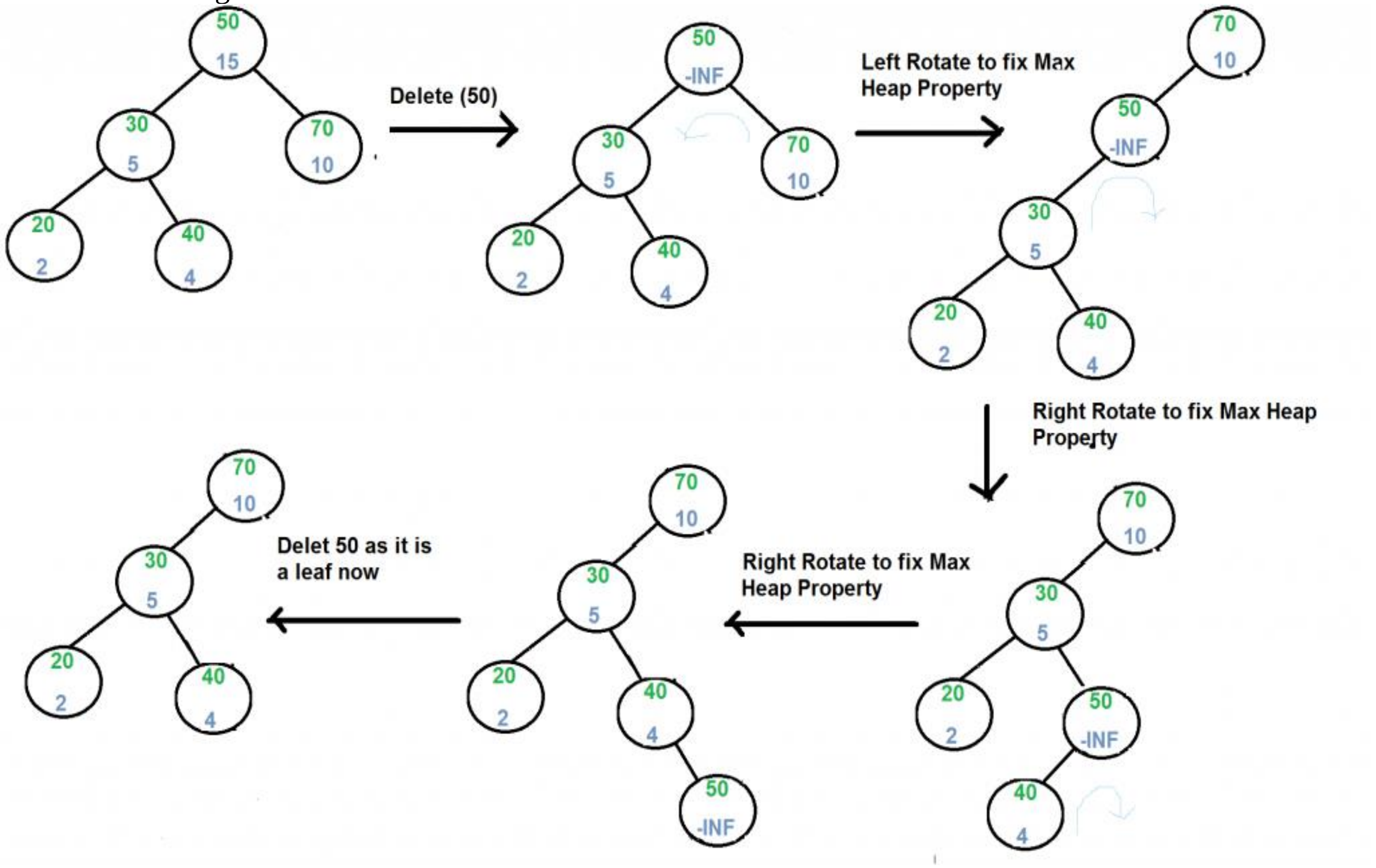
1. Create new node with key equals to x and value equals to a random value.
2. Perform standard BST insert.
3. Use rotations to make sure that inserted node's priority follows max heap property.



2. Deletion in Treap

To delete a node x from the treap, remove it if it is a leaf of the tree. If x has a single child, z, remove x from the tree and make z be the child of the parent of x (or make z the root of the tree if x had no parent). Finally, if x has two children, swap its position in the tree with its immediate successor z in the sorted order, resulting in one of the previous cases. In this last case, the swap may violate the heap-ordering property for z, so additional rotations may need to be performed to restore this property.

1. If node to be deleted is a leaf, delete it.
2. Else replace node's priority with minus infinite (-INF), and do appropriate rotations to bring the node down to a leaf.


Delete (50)
Left Rotate to fix Max Heap Property
Right Rotate to fix Max Heap Property
Right Rotate to fix Max Heap Property
Delet 50 as it is a leaf now
50 15 30 5 70 10 20 2 40 4
-INF

3. Searching in Treap

To search for a given key value, apply a standard search algorithm in a binary search tree, ignoring the priorities.

```
public Node search(Node root, int key)
{
   // Base Cases: root is null or key is present at root
   if (root==null || root.key==key)
       return root;
    // Key is greater than root's key
   if (root.key < key)
      return search(root.right, key);
    // Key is smaller than root's key
   return search(root.left, key);
}
```

Java program for Treap: **Treap.java**

```
import java.util.*;

// A Treap Node
class TreapNode
{
	int key, priority;
	TreapNode left, right;
}

class treap
{
	public static TreapNode rightRotate(TreapNode y)
	{
		System.out.println("rightRotate " + y.key);
		TreapNode x = y.left;
		TreapNode T2 = x.right;

		// Perform rotation
		x.right = y;
		y.left = T2;

		// Return new root
		return x;
	}

	// A utility function to left rotate subtree rooted with x
	public static TreapNode leftRotate(TreapNode x)
	{
		System.out.println("leftRotate " + x.key);

		TreapNode y = x.right;
		TreapNode T2 = y.left;

		// Perform rotation
		y.left = x;
		x.right = T2;

		// Return new root
		return y;
	}

	/* Utility function to add a new key */
	public static TreapNode newNode(int key)
	{
		TreapNode temp = new TreapNode();
		temp.key = key;
		temp.priority = (int)(Math.random() * 100);
		temp.left = temp.right = null;
```

```
        return temp;
    }

    /* Recursive implementation of insertion in Treap
    1) Create new node with key equals to key and value equals to a random value.
    2) Perform standard BST insert.
    3) Use rotations to make sure that inserted node's priority follows max heap property.
    */
    public static TreapNode insertNode(TreapNode root, int key)
    {
        // If root is null, create a new node and return it
        if (root == null) {
            return newNode(key);
        }

        // If key is smaller than root
        if (key <= root.key)
        {
            // Insert in left subtree
            root.left = insertNode(root.left, key);

            // Fix Heap property if it is violated
            //It is Max heap. To change to min heap, change the condition to less than
            if (root.left.priority > root.priority)
            {
                root = rightRotate(root);
            }
        }
        else
        {
            // If key is greater
            // Insert in right subtree
            root.right = insertNode(root.right, key);

            // Fix Heap property if it is violated
            //It is Max heap. To change to min heap, change the condition to less than
            if (root.right.priority > root.priority)
            {
                root = leftRotate(root);
            }
        }
        return root;
    }

    /* Recursive implementation of Delete() */
    public static TreapNode deleteNode(TreapNode root, int key)
    {

        if (root == null)
            return root;
        if (key < root.key)
            root.left = deleteNode(root.left, key);
        else if (key > root.key)
            root.right = deleteNode(root.right, key);

        // IF KEY IS AT ROOT
        // If left is NULL
        else if (root.left == null)
        {
            TreapNode temp = root.right;
            root = temp; // Make right child as root
        }
        // If Right is NULL
        else if (root.right == null)
        {
            TreapNode temp = root.left;
            root = temp; // Make left child as root
        }
        // If key is at root and both left and right are not NULL
        // If priority of right child is greater, perform left rotation at node
        else if (root.left.priority < root.right.priority)
        {
            root = leftRotate(root);
            root.left = deleteNode(root.left, key);
        }
        else    // If priority of left child is greater, perform right rotation at node
        {
            root = rightRotate(root);
            root.right = deleteNode(root.right, key);
        }
        return root;
    }

    // Search a given key in a given BST
    public static TreapNode search(TreapNode root, int key)
    {
        // Base Cases: root is null or key is present at root
        if (root == null || root.key == key)
            return root;

        // Key is greater than root's key
        if (root.key < key)
            return search(root.right, key);

        // Key is smaller than root's key
        return search(root.left, key);
    }

    static void preorder(TreapNode root)
    {
        if (root != null)
        {
            System.out.println("key: " + root.key + " | priority: " + root.priority);
            preorder(root.left);
            preorder(root.right);
        }
    }

    // 6
    // 2 4 3 1 7 5

    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int arr[] = new int[n];
        for(int i=0;i<n;i++)
        {
            arr[i] = sc.nextInt();
        }
        TreapNode root = null;
        for(int a:arr)
        {
            root = insertNode(root,a);
        }
        preorder(root);
        System.out.println("Enter item to search ");
        int key = sc.nextInt();

        TreapNode result = search(root, key);
        if(result != null)
            System.out.println("Search result "+ result.key + " " + result.priority);
        else
            System.out.println("Key " + key + " not found");

        do
        {
            System.out.println("Enter item to delete ");
            key = sc.nextInt();
            root = deleteNode(root, key);
            System.out.println("After delete");
            preorder(root);
        } while(key != -1 && root != null);
    }
}
```

<u>Input:-</u>

```
6
2 3 4 5 1 7
```

<u>Output:-</u>

```
key: 2 | priority: 94
key: 1 | priority: 47
key: 7 | priority: 85
key: 5 | priority: 23
key: 4 | priority: 14
key: 3 | priority: 6

Enter item to search
2
Search result 2 94
Enter item to delete
2
After delete
key: 7 | priority: 85
key: 1 | priority: 47
key: 5 | priority: 23
key: 4 | priority: 14
key: 3 | priority: 6
```

(note : here only partial output is shown this is for your understanding purpose)

Kth Largest Element in an Array Using Treap:

Given an integer array nums and an integer k, return the kth largest element in the array.

Note that it is the kth largest element in the sorted order, not the kth distinct element.

You must solve it in O(n) time complexity.

Example 1:

Input: nums = [3,2,1,5,6,4], k = 2

Output: 5

Example 2:

Input: nums = [3,2,3,1,2,4,5,5,6], k = 4

Output: 4

Note: You are supposed to print the K'th largest height in the sorted order of heights[].

Not the K'th distinct height.

Explanation

- In the first example, we can see that *k*=2, so we have to find the second largest element in this array. If we rearrange the array in descending order [9,7,6,4,3,2,1][9,7,6,4,3,2,1], we can see that the element 7 is in the second position, so it is considered the second largest element.
- In the second example, we can see that *k*=4, so we have to find the fourth largest element in this array. If we rearrange the array in descending order [90,65,34,12][90,65,34,12], we can see that the element 12 is in the fourth position, so it is considered the fourth largest element.

Java Program for K^th Largest Element in an Array Using Treap:

KthLargest.java

```
import java.util.*;
class TreapNode
{
        int data;
        int priority;
        TreapNode left;
        TreapNode right;
                TreapNode(int data)
                {
                        this.data = data;
                        this.priority = new Random().nextInt(1000);
                        this.left = this.right = null;
                }
}

class KthLargest
{
        static int k;
        public static TreapNode rotateLeft(TreapNode root)
        {
                TreapNode R = root.right;
                TreapNode X = root.right.left;
                R.left = root;
                root.right = X;
                return R;
        }

        public static TreapNode rotateRight(TreapNode root)
        {
                TreapNode L = root.left;
                TreapNode Y = root.left.right;
                L.right = root;
                root.left = Y;
                return L;
        }

        public static TreapNode insertNode(TreapNode root, int data)
        {
                if (root == null)
                {
                        return new TreapNode(data);
                }
                if (data < root.data)
                {
                        root.left = insertNode(root.left, data);
                        if (root.left != null && root.left.priority > root.priority)
                        {
```

```
                root = rotateRight(root);
            }
        }
        else
        {
            root.right = insertNode(root.right, data);
            if (root.right != null && root.right.priority > root.priority)
            {
                root = rotateLeft(root);
            }
        }
        return root;
    }

    static void inorder(TreapNode root)
    {
        if (root == null)
            return;
        inorder(root.left);
        k--;
        if(k==0)
        {
            System.out.print(root.data);
            return;
        }
        inorder(root.right);
    }

    static void printTreap(TreapNode root)
    {
        if (root == null)
            return;
        printTreap(root.left);
        System.out.println(root.data + " " + root.priority);
        printTreap(root.right);
    }

    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int p = sc.nextInt();
        k=n-p+1;
        int arr[] = new int[n];
        for(int i=0;i<n;i++)
        {
            arr[i] = sc.nextInt();
        }
        TreapNode root = null;
        for(int a:arr)
```

```
                {
                        root = insertNode(root,a);
                }
                printTreap(root);
                inorder(root);
        }
}
```

Test case=1
Input =6 3
2 4 3 1 6 5
Output =4

Test case=2
Input =6 2
3 2 1 5 6 4
Output =5

Trie Data Structure

Trie Introduction

- Trie data structure is defined as a Tree based data structure that is used for storing some collection of strings and performing efficient search operations on them.
- The word Trie is derived from reTRIEval, which means finding something or obtaining it.
- Trie follows some property that If two strings have a common prefix then they will have the same ancestor in the trie.
- A trie can be used **to sort a collection of strings** alphabetically as well as **search whether a string with a given prefix** is present in the trie or not.
- A Trie data structure is used for storing and retrieval of data and the same operations could be done using another data structure which is Hash Table but Trie can perform these operations more efficiently than a Hash Table.
- Moreover, Trie has its own advantage over the Hash table. A Trie data structure can be used for prefix-based searching whereas a Hash table can't be used in the same way.

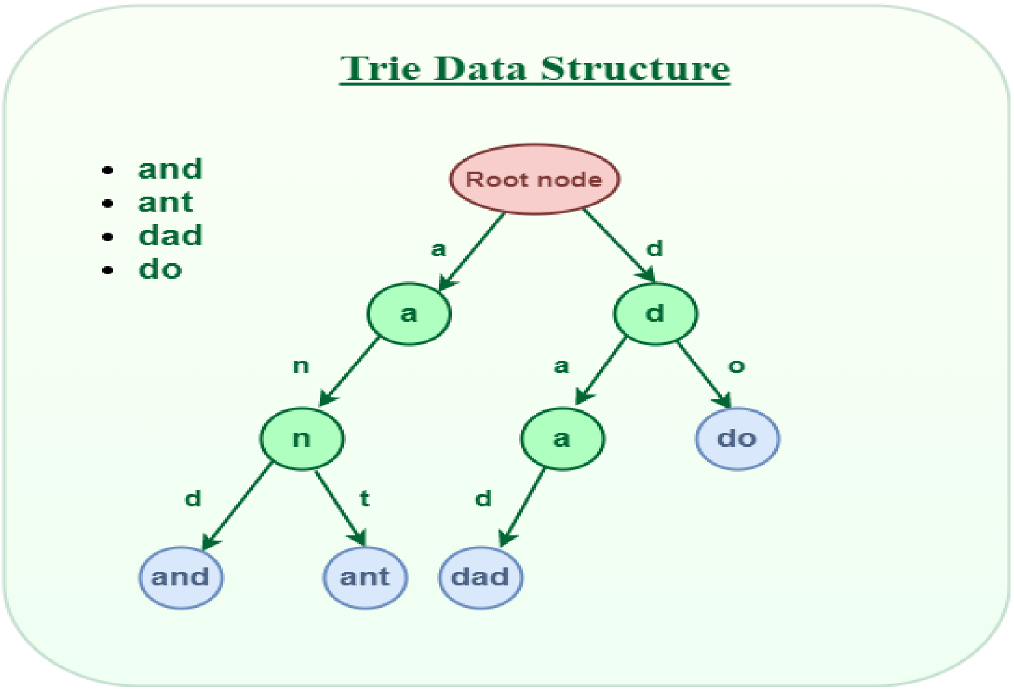
Advantages of Trie Data Structure over a Hash Table:

- We can efficiently do prefix search (or auto-complete) with Trie.
- We can easily print all words in alphabetical order which is not easily possible with hashing.
- There is no overhead of Hash functions in a Trie data structure.
- Searching for a String even in the large collection of strings in a Trie data structure can be done in O(L) Time complexity, Where L is the number of words in the query string. This searching time could be even less than O(L) if the query string does not exist in the trie.

Properties of a Trie Data Structure:

- Now we already know that Trie has a tree-like structure. So, it is very important to know its properties. Below are some important properties of the Trie data structure:
- There is one root node in each Trie.
- Each node of a Trie represents a string and each edge represents a character.
- Every node consists of hashmaps or an array of pointers, with each index representing a character and a flag to indicate if any string ends at the current node.
- Trie data structure can contain any number of characters including alphabets, numbers, and special characters. But for this article, we will discuss strings with characters a-z. Therefore, only 26 pointers need for every node, where the 0th index represents 'a' and the 25th index represents 'z' characters.
- Each path from the root to any node represents a word or string.

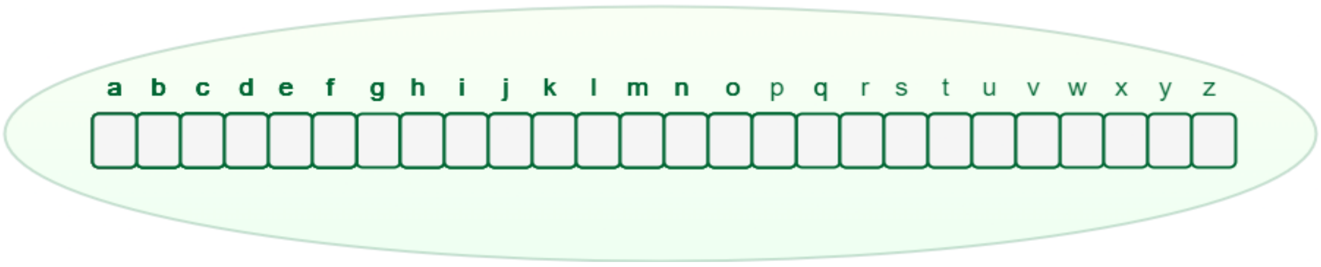
Below is a simple example of Trie data structure.



How does Trie Data Structure work?

We already know that the Trie data structure can contain any number of characters including alphabets, numbers, and special characters. But for this example, we will discuss strings with characters a-z. Therefore, only 26 pointers need for every node, where the 0th index represents ‘a’ and the 25th index represents ‘z’ characters.

Any lowercase English word can start with a-z, then the next letter of the word could be a-z, the third letter of the word again could be a-z, and so on. So for storing a word, we need to take an array (container) of size 26 and initially, all the characters are empty as there are no words and it will look as shown below.



Let’s see how a word **“and”** and **“ant”** is stored in the Trie data structure:

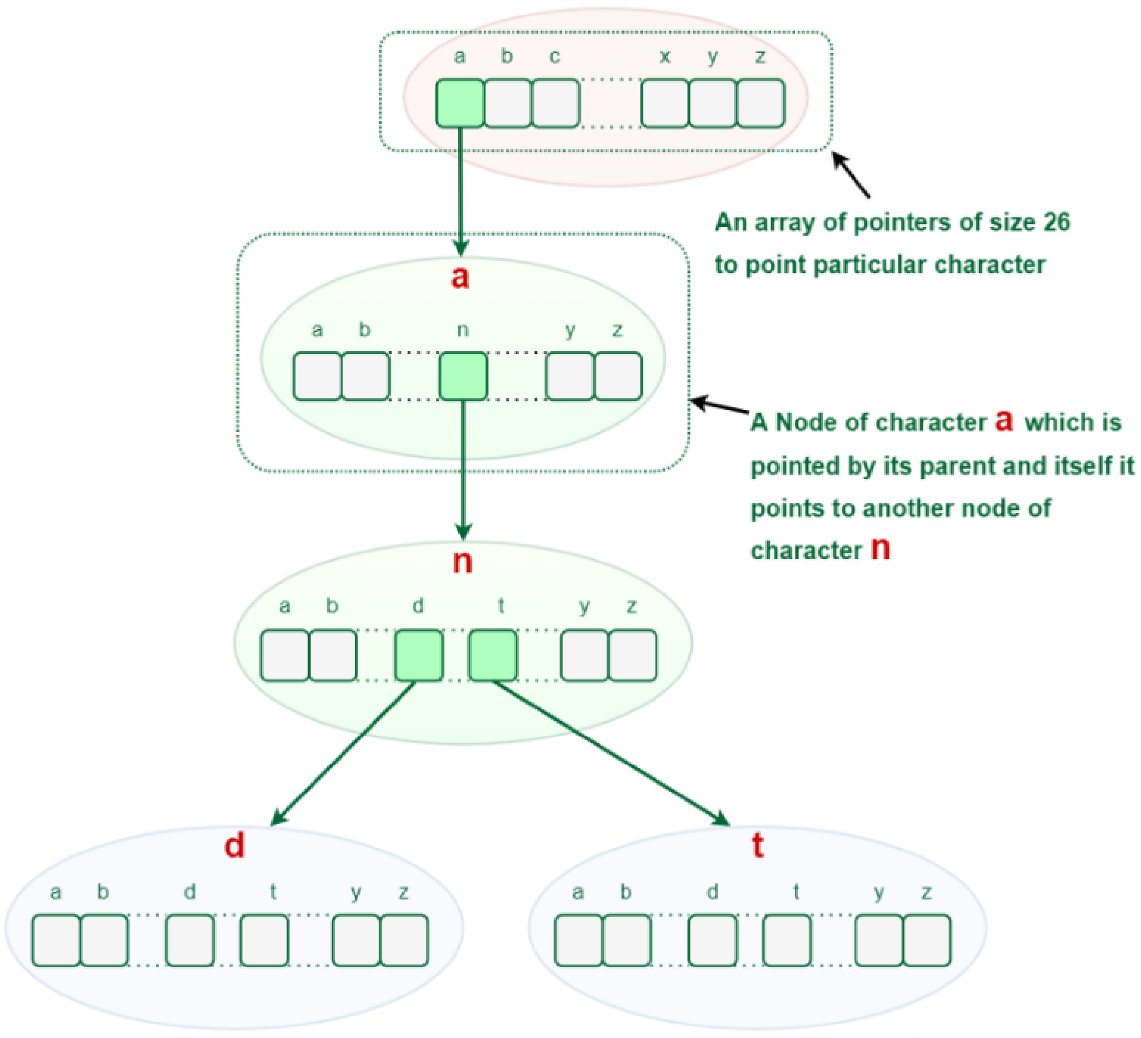
1. **Store “and” in Trie data structure**:

   ➢ The word “and” starts with “a“, So we will mark the position “a” as filled in the Trie node, which represents the use of “a”.

- ➢ After placing the first character, for the second character again there are 26 possibilities, So from "a", again there is an array of size 26, for storing the 2nd character.
- ➢ The second character is "n", So from "a", we will move to "n" and mark "n" in the 2nd array as used.
- ➢ After "n", the 3rd character is "d", So mark the position "d" as used in the respective array.

2. **Store "ant" in the Trie data structure:**
   - ➢ The word "ant" starts with "a" and the position of "a" in the root node has already been filled. So, no need to fill it again, just move to the node 'a' in Trie
   - ➢ For the second character 'n' we can observe that the position of 'n' in the 'a' node has already been filled. So, no need to fill it again, just move to node 'n' in Trie.
   - ➢ For the last character 't' of the word, The position for 't' in the 'n' node is not filled. So, filled the position of 't' in 'n' node and move to 't' node.

After storing the word "and" and "ant" the Trie will look like this:


a b c x y z
An array of pointers of size 26 to point particular character
a
a b n y z
A Node of character a which is pointed by its parent and itself it points to another node of character n
n
a b d t y z
d
a b d t y z
t
a b d t y z

Representation of Trie Node

Every Trie node consists of a character pointer array or hash map and a flag to represent if the word is ending at that node or not. But if the words contain only lower-case letters (i.e. a-z), then we can define Trie Node with an array instead of a hash map.

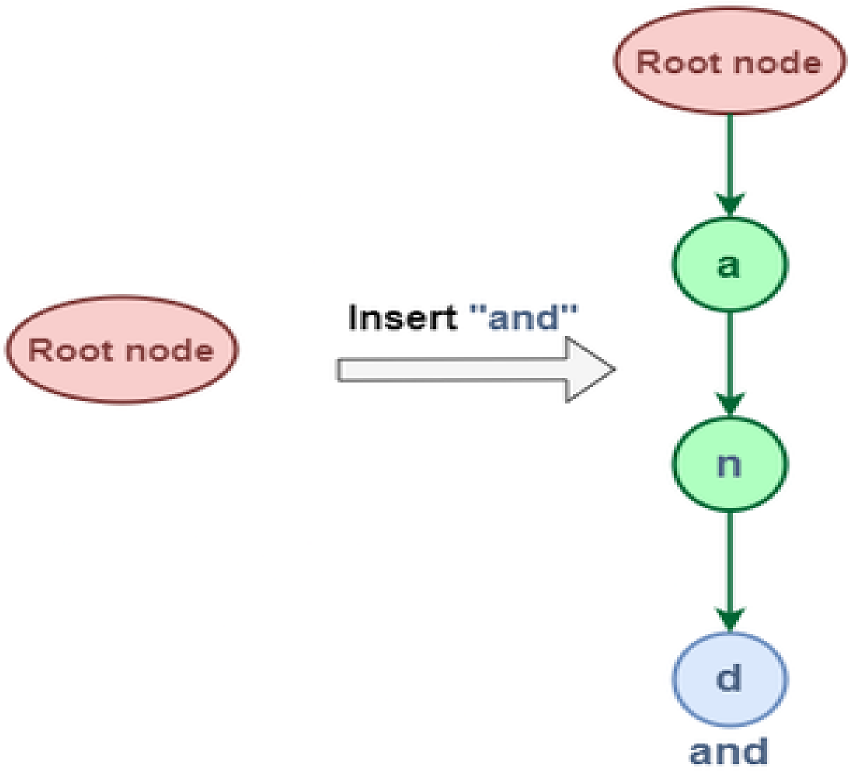
```
static class TrieNode
{
        TrieNode[] children = new TrieNode[NUM_CHARS];
        // isEndOfWord is true if the node represents end of a word
        boolean isEndOfWord;
        TrieNode()
        {
                isEndOfWord = false;
                for (int i = 0; i < NUM_CHARS; i++)
                        children[i] = null;
        }
};
```

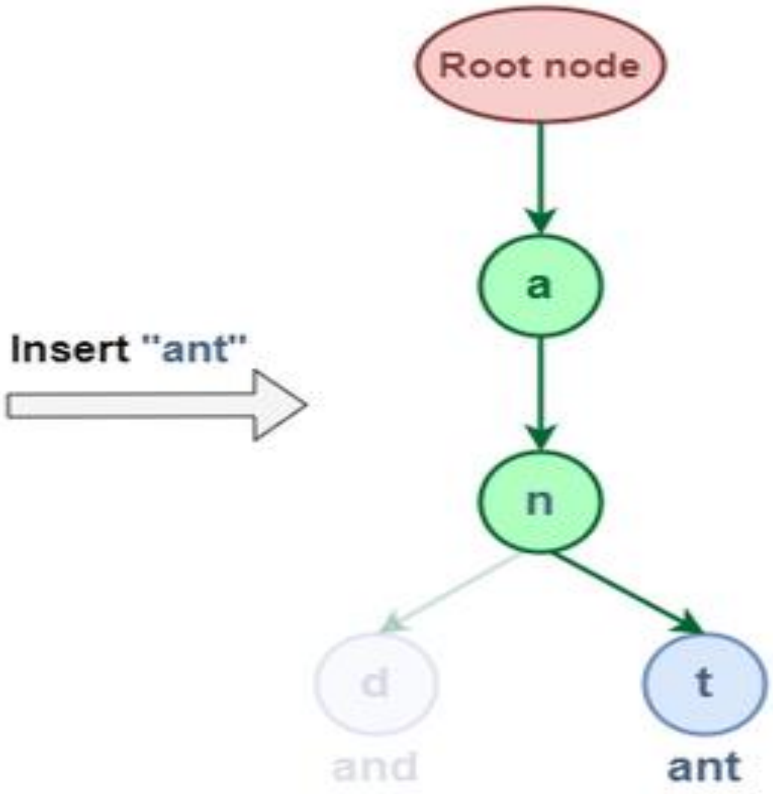
Basic Operations on Trie Data Structure:

1. Insertion.
2. Search.
3. Deletion.

1. Insertion in Trie Data Structure

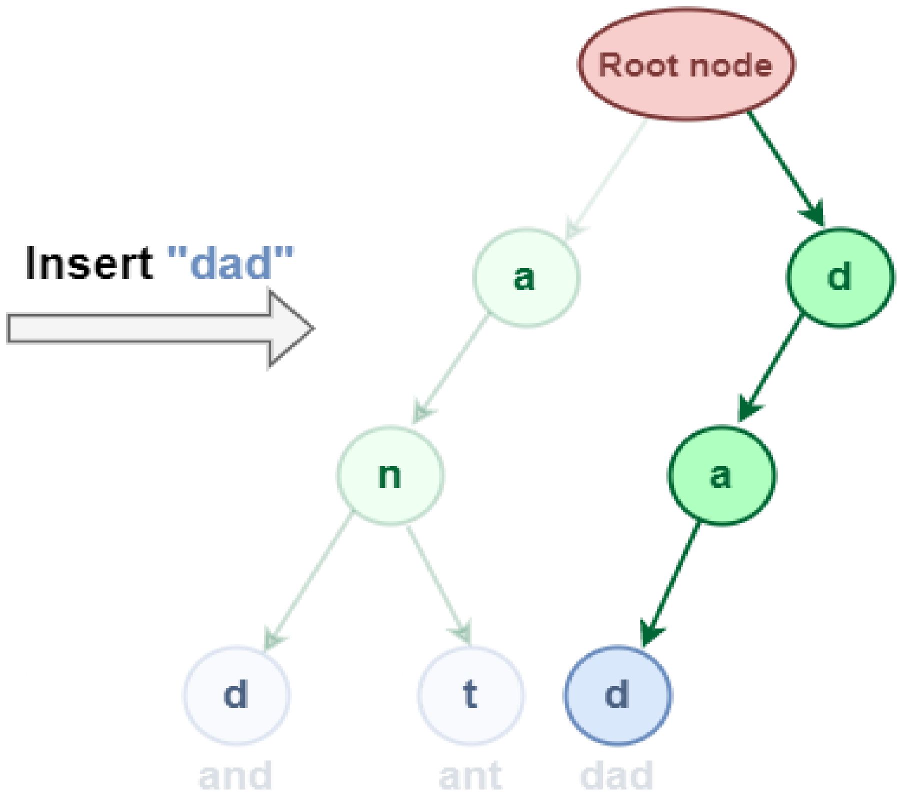
This operation is used to insert new strings into the Trie data structure. Let us see how this works: **Let us try to Insert "and" & "ant" in this Trie:**

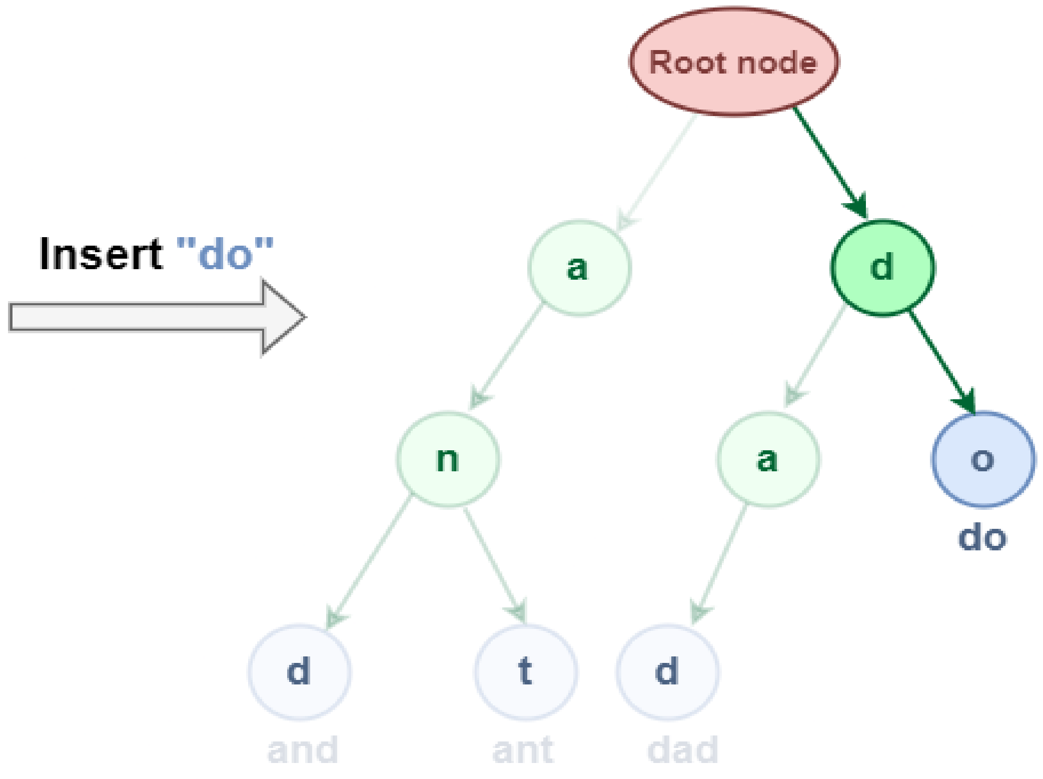




From the above representation of insertion, we can see that the word “and” & “ant” have shared some common node (i.e “an”) this is because of the property of the Trie data structure that If two strings have a common prefix then they will have the same ancestor in the trie.

Now let us try to Insert “dad” & “do”:





<u>Implementation of Insertion in Trie data structure:</u>

<u>Algorithm:</u>

- Define a function insert(TrieNode *root, string &word) which will take two parameters one for the root and the other for the string that we want to insert in the Trie data structure.
- Now take another pointer currentNode and initialize it with the root node.
- Iterate over the length of the given string and check if the value is NULL or not in the array of pointers at the current character of the string.
    - ✓ If It’s NULL then, make a new node and point the current character to

      this newly created node.
    - ✓ Move the curr to the newly created node.
- Finally, increment the wordCount of the last currentNode, this implies that there is a string ending currentNode.

2. Searching in Trie Data Structure

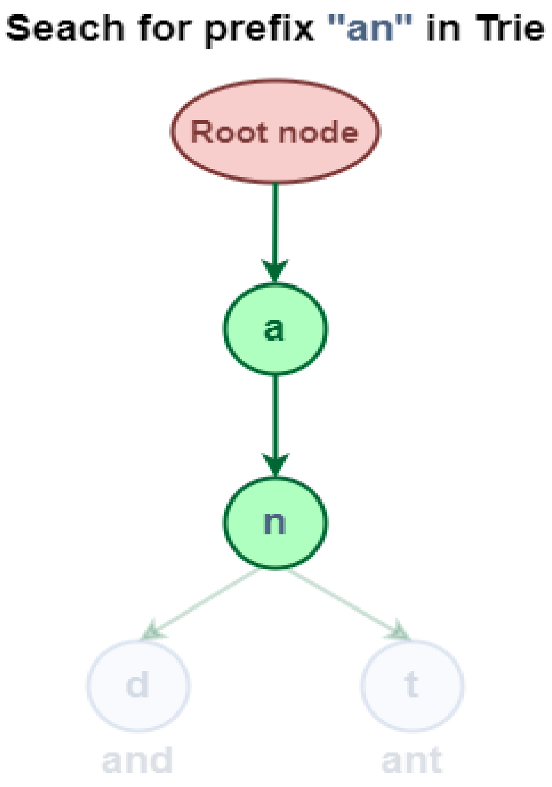
Search operation in Trie is performed in a similar way as the insertion operation but the only difference is that whenever we find that the array of pointers in **curr node** does not point to the **current character** of the **word** then return false instead of creating a new node for that current character of the word.
This operation is used to search whether a string is present in the Trie data structure or not. There are two search approaches in the Trie data structure.

1. Find whether the given word exists in Trie.
2. Find whether any word that starts with the given prefix exists in Trie.

There is a similar search pattern in both approaches. The first step in searching a given word in Trie is to convert the word to characters and then compare every character with the trie node from the root node. If the current character is present in the node, move forward to its children. Repeat this process until all characters are found.

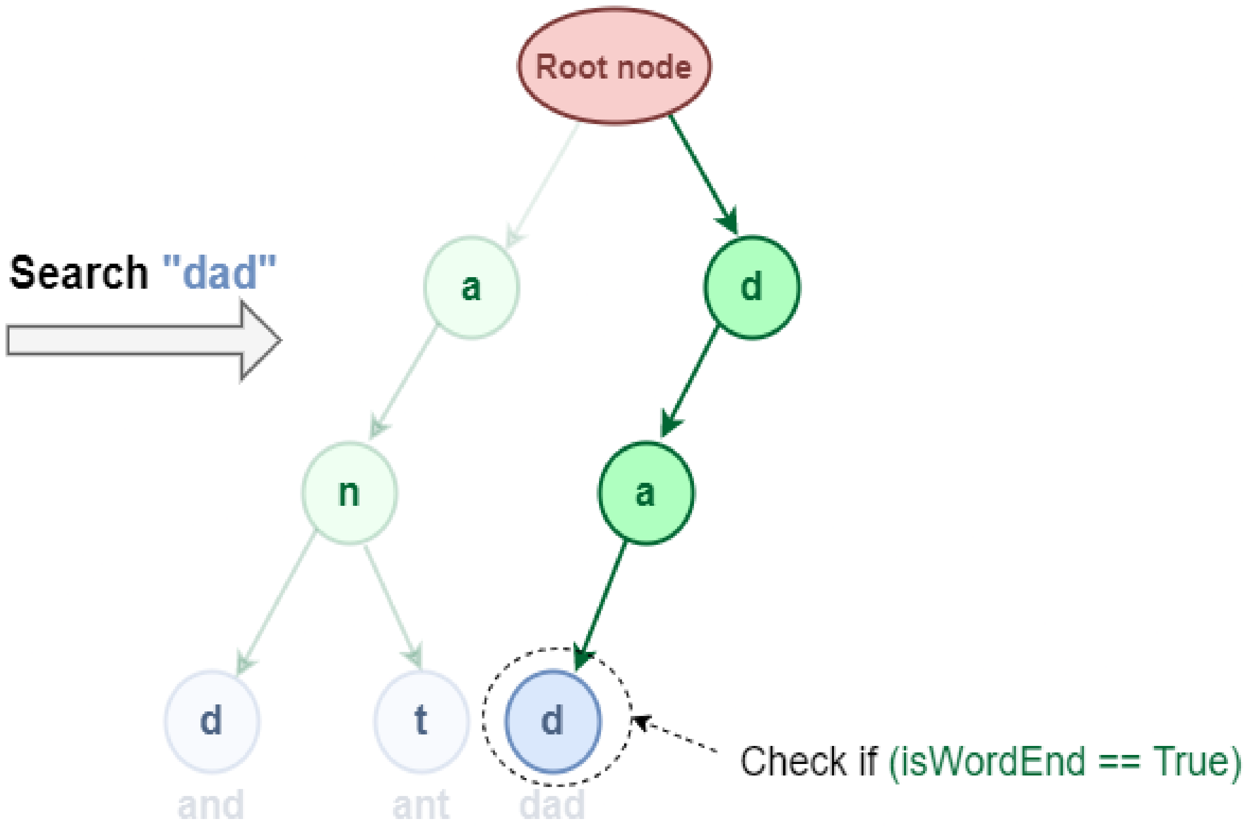
Searching Prefix in Trie Data Structure:

Search for the prefix "an" in the Trie Data Structure.



Searching complete word in Trie Data Structure

It is similar to prefix search but additionally, we have to check if the word is ending at the last character of the word or not.

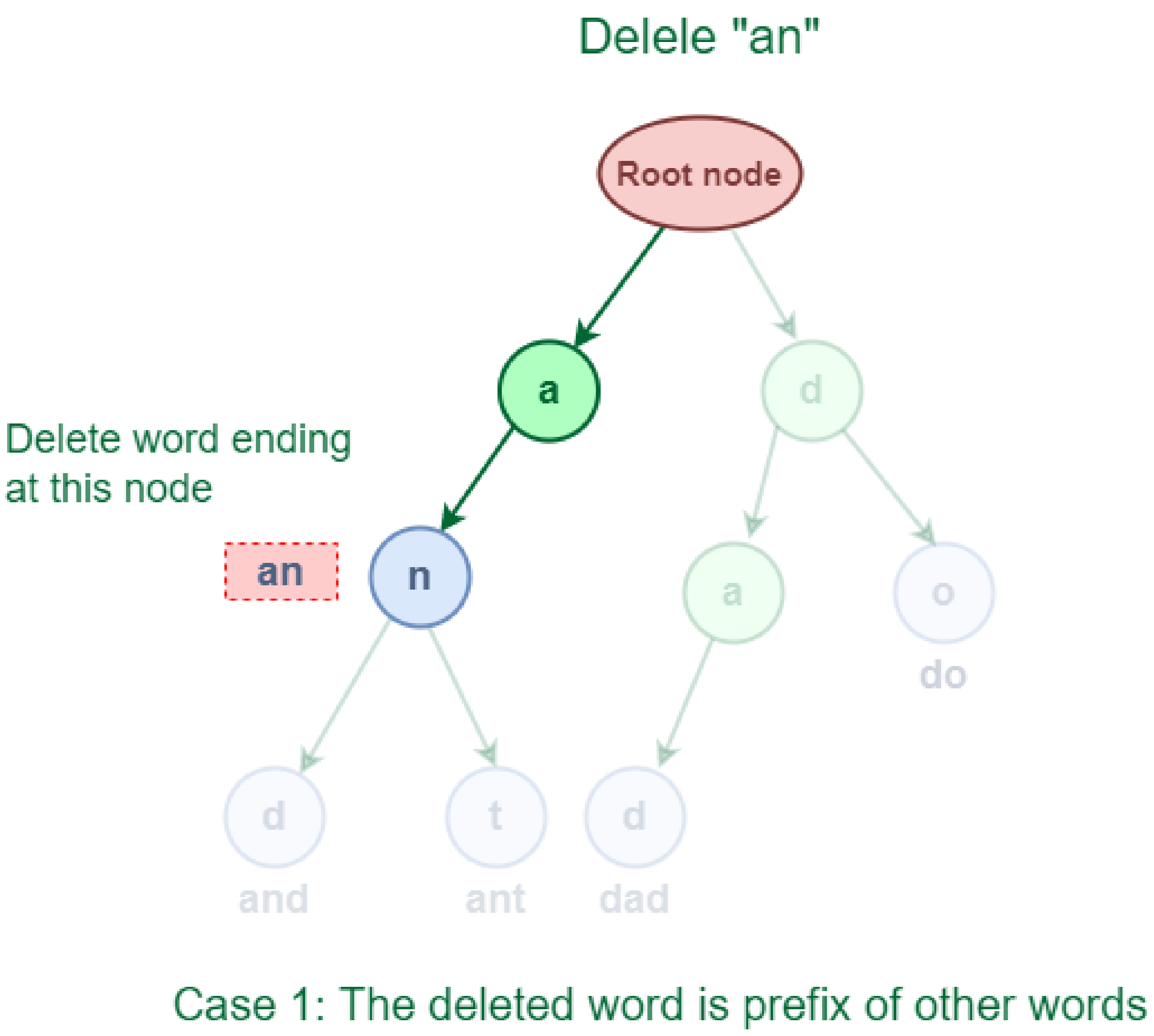


3. Deletion in Trie Data Structure

This operation is used to delete strings from the Trie data structure. There are three cases when deleting a word from Trie.

1. The deleted word is a prefix of other words in Trie.
2. The deleted word shares a common prefix with other words in Trie.
3. The deleted word does not share any common prefix with other words in Trie.

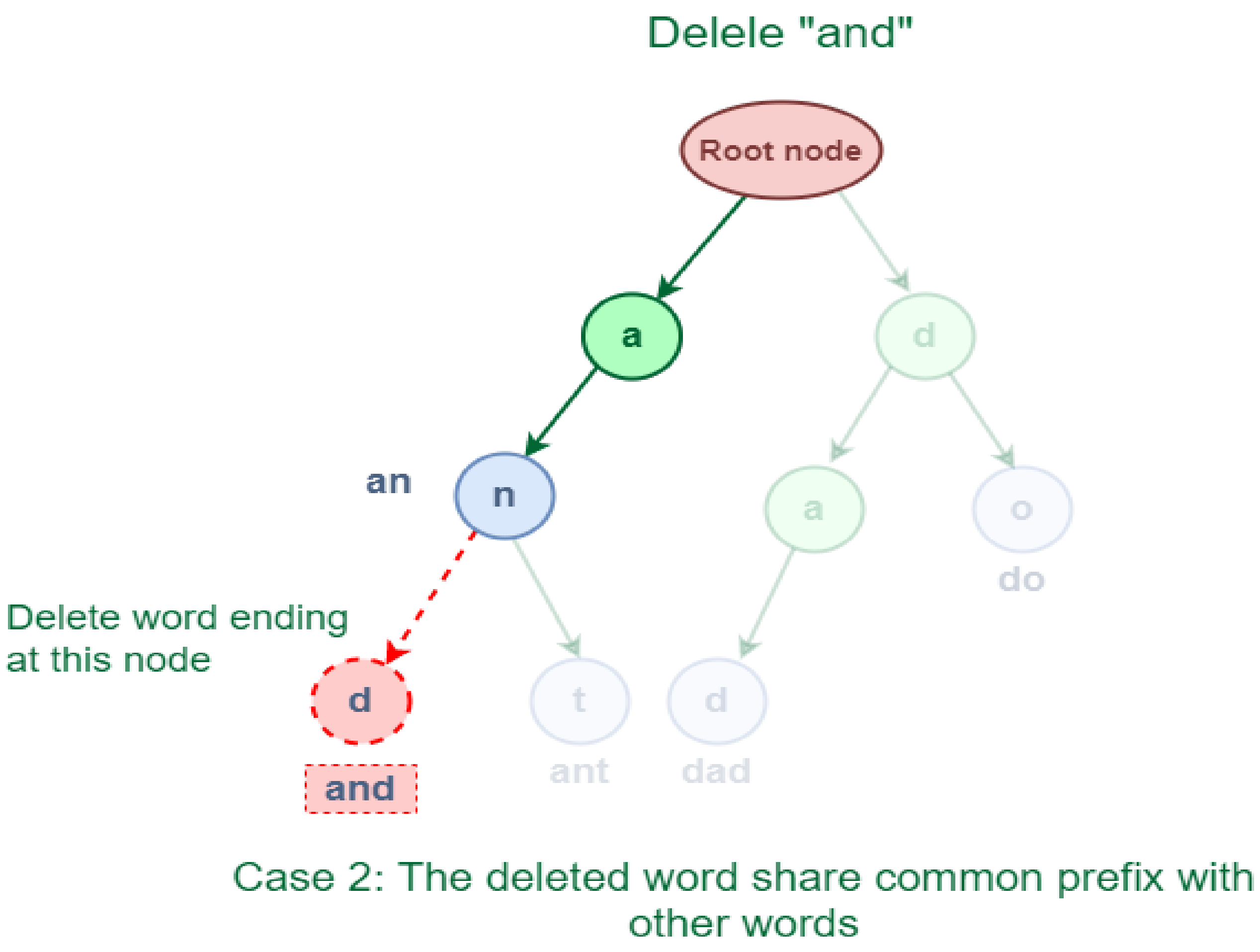
1. The deleted word is a prefix of other words in Trie.
As shown in the following figure, the deleted word “an” share a complete prefix with another word “and” and “ant“.


Delele "an"
Root node
a
d
Delete word ending at this node
an
n
a
o
do
d
t
d
and
ant
dad
Case 1: The deleted word is prefix of other words

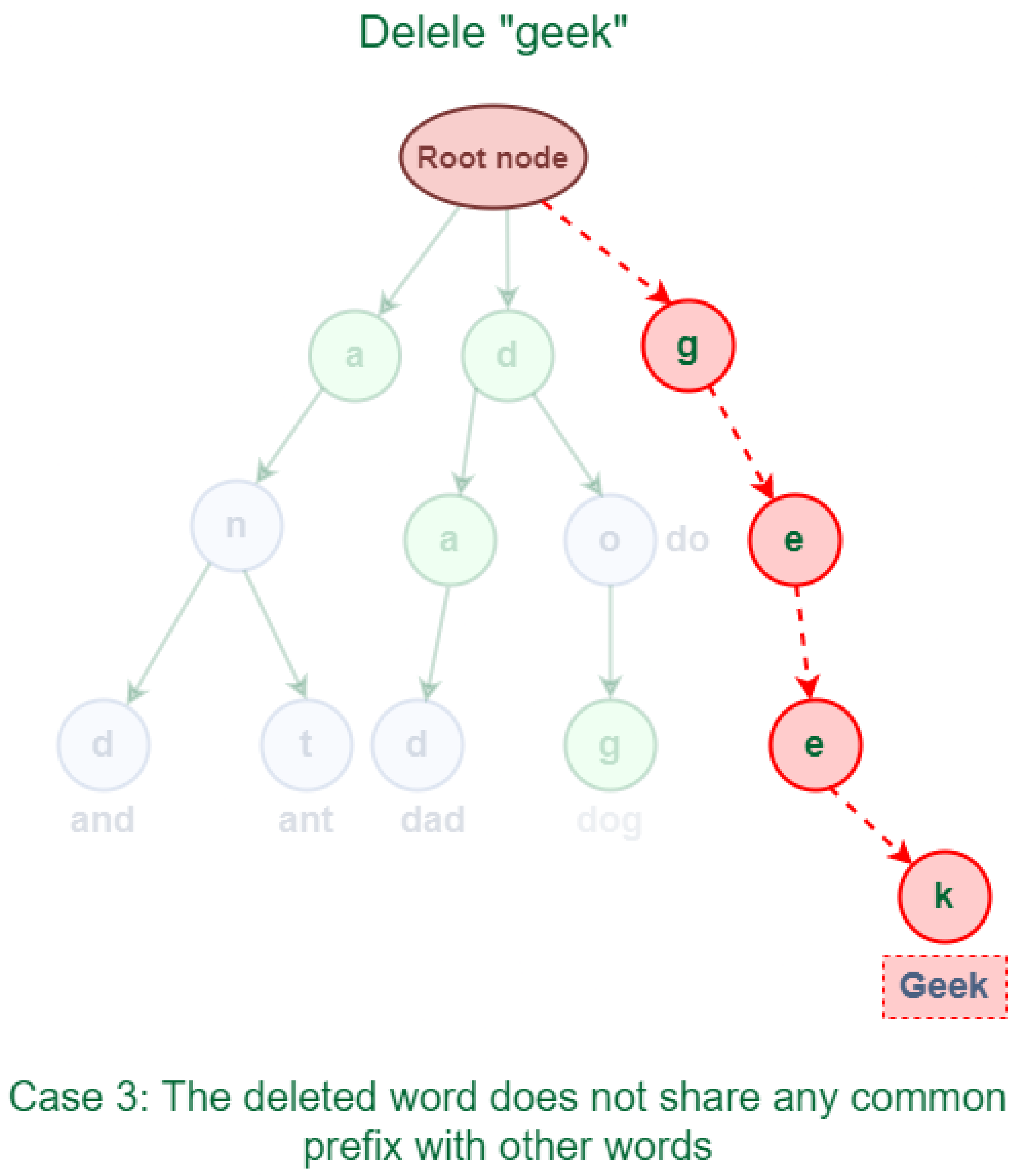
An easy solution to perform a delete operation for this case is to just decrement the wordCount by 1 at the ending node of the word.

2. The deleted word shares a common prefix with other words in Trie.

- As shown in the following figure, the deleted word “and” has some common prefixes with other words ‘ant’. They share the prefix ‘an’.
- The solution for this case is to delete all the nodes starting from the end of the prefix to the last character of the given word.


Delele "and"
Root node
a
d
an
n
a
o
do
Delete word ending at this node
d
t
d
and
ant
dad
Case 2: The deleted word share common prefix with other words

3. The deleted word does not share any common prefix with other words in Trie
As shown in the following figure, the word “geek” does not share any common prefix with any other words.


Delele "geek"
Root node
a
d
g
n
a
o
do
e
e
d
t
d
g
and
ant
dad
dog
k
Geek
Case 3: The deleted word does not share any common prefix with other words

<u>Java Pogram to implement Trie Data Structure</u>: **Trie.java**

```
import java.util.*;
class Trie
{
        static final int NUM_CHARS = 26;
        // To handle prefix deletion
        static boolean isDeleted = false;
        // trie node
        static class TrieNode
        {
                TrieNode[] children = new TrieNode[NUM_CHARS];
                // isEndOfWord is true if the node represents end of a word
                boolean isEndOfWord;
                TrieNode()
                {
                        isEndOfWord = false;
                        for (int i = 0; i < NUM_CHARS; i++)
                                children[i] = null;
                }
        };
        static TrieNode root;
        // If not present, inserts key into trie
        // If the key is prefix of trie node, just marks leaf node
        static void insert(String key)
        {
                int level;
                int length = key.length();
                int index;
                TrieNode currentNode = root;
                for (level = 0; level < length; level++)
                {
                        index = key.charAt(level) - 'a';
                        if (currentNode.children[index] == null)
                                currentNode.children[index] = new TrieNode();
                        currentNode = currentNode.children[index];
                }
                // mark last node as leaf
                currentNode.isEndOfWord = true;
        }
        // Returns true if key (prefix or complete word) is present in trie, else false
        static boolean search(String key)
        {
                int level;
                int length = key.length();
                int index;
                TrieNode currentNode = root;
                for (level = 0; level < length; level++)
                {
                        index = key.charAt(level) - 'a';
```

```
            if (currentNode.children[index] == null)
                return false;
            currentNode = currentNode.children[index];
        }
        // To check if prefix exists in the Trie
        // return true;
        // To check for complete word
        return (currentNode.isEndOfWord);
    }
    // Returns true if root has no children, else false
    static boolean isEmpty(TrieNode root)
    {
        for (int i = 0; i < NUM_CHARS; i++)
            if (root.children[i] != null)
                return false;
        return true;
    }
    // Recursive function to delete a key from given Trie
    static TrieNode delete(TrieNode root, String key, int depth)
    {
        // If tree is empty
        if (root == null)
            return null;
        // If last character of key is being processed
        if (depth == key.length())
        {
            // isDeleted is true if it is end of word and false otherwise
            isDeleted = root.isEndOfWord;
            // This node is no more end of word after removal of given key
            if (root.isEndOfWord)
                root.isEndOfWord = false;
            // If given is not prefix of any other word
            if (isEmpty(root))
            {
                return null;
            }
            return root;
        }
        // If not last character, recur for the child obtained using ASCII value
        int index = key.charAt(depth) - 'a';
        if (root.children[index] == null)
            return null;
        root.children[index] = delete(root.children[index], key, depth + 1);
        /* If root does not have any child (its only child got deleted), and it is not end
        of another word.*/
        if (isEmpty(root) && root.isEndOfWord == false)
        {
            return null;
        }
        return root;
```

```
}
// To check if current node is leaf node or not
static boolean isLeafNode(TrieNode root)
{
        return root.isEndOfWord == true;
}
// print Trie
static void print(TrieNode root, char[] str, int level)
{
        // If node is leaf node, it indicates end of string,
        // so, a null character is added and string is printed
        if (isLeafNode(root))
        {
                for (int k = level; k < str.length; k++)
                        str[k] = 0;
                System.out.println(str);
        }
        int i;
        for (i = 0; i < NUM_CHARS; i++)
        {
                // if NON-NULL child is found add parent key to str and
                // call the print function recursively for child node
                if (root.children[i] != null)
                {
                        str[level] = (char) (i + 'a');
                        print(root.children[i], str, level + 1);
                }
        }
}
public static void main(String args[])
{
        Scanner sc=new Scanner(System.in);
        String keys[]=sc.nextLine().split(" ");
        root = new TrieNode();
        // Construct trie
        int i;
        for (i = 0; i < keys.length ; i++)
                insert(keys[i]);
        char[] str = new char[50];
        String word;
label1: while(true)
        {
          int opt = sc.nextInt();
          sc.nextLine();
          switch(opt){
          case 4:
                System.out.println("Content of Trie: ");
                print(root, str, 0);
                break;
          case  1:
```

```
                    String s = sc.nextLine();
                    insert(s);
                    System.out.println("Content of Trie: ");
                    print(root, str, 0);
                    break;
               case 2:
                     word = sc.next();
                     if(search(word) == true)
                          System.out.println(word + " is present ");
                     else System.out.println(word + " is not present");
                     break;
               case 3:
                     word = sc.next();
                     if(delete(root, word, 0) != null & isDeleted == true)
                          System.out.println(word + " is deleted ");
                     else
                          System.out.println(word + " is not present in Trie to be deleted");
                     System.out.println("Content of Trie after deletion: ");
                     print(root, str, 0);
                     break;
              case 5:
                    break label1;
              }

      }
}
}
```

Sample Test Case:

```
abc def ghi    //input
4              //input
Content of Trie:
abc
def
ghi
1              //input
Pqr            //input
Content of Trie:
abc
def
ghi
pqr
2              //input
abc            //input
abc is present
1              //input
mno            //input
Content of Trie:
abc
def
```

```
ghi
mno
pqr
3              //input
abc            //input
abc is deleted
Content of Trie after deletion:
def
ghi
mno
pqr
5
```

Applications

1. Index Pairs of a String
2. Longest word with all prefixes
3. Top K frequent words.

1. Index Pairs of a String:

Given a text string and words (a list of strings), return all index pairs [i, j] so that the substring text[i]...text[j] is in the list of words.

Example 1:
Input: text = "thestoryofleetcodeandme", words = ["story","fleet","leetcode"]
Output: [[3,7],[9,13],[10,17]]

Example 2:
Input: text = "ababa", words = ["aba","ab"]
Output: [[0,1],[0,2],[2,3],[2,4]]
Explanation:
Notice that matches can overlap, see "aba" is found in [0,2] and [2,4].

Java Program for Index Pairs of a String using Trie DS: **IndexPairs.java**

```
import java.util.*;
class Solution
{
        public int[][] indexPairs(String text, String[] words)
        {
                /*initializing tire and put all word from words into Trie.*/
                Trie trie=new Trie();
                for(String s:words)
                {
                        Trie cur=trie;
                        for(char c:s.toCharArray())
                        {
                                if(cur.children[c-'a']==null)
                                {
                                        cur.children[c-'a']=new Trie();
                                }
                                cur=cur.children[c-'a'];
                        }
                        cur.end=true;       /*mark there is a word*/
                }

                /*if text is "ababa", check "ababa","baba","aba","ba","a" individually.*/
                int len=text.length();
                List<int[]> list=new ArrayList<>();
                for(int i=0;i<len;i++)
                {
                        Trie cur=trie;
                        char cc=text.charAt(i);
                        int j=i;   /*j is our moving index*/

                        while(cur.children[cc-'a']!=null)
                        {
                                cur=cur.children[cc-'a'];
                                if(cur.end)
                                {   /*there is a word ending here, put into our list*/
                                        list.add(new int[]{i,j});
                                }
                                j++;
                                if(j==len)
                                {  /*reach the end of the text, we stop*/
                                        break;
                                }
                                else
                                {
                                        cc=text.charAt(j);
                                }
                        }
                }
                /*put all the pairs from list into array*/
                int size=list.size();
                int[][] res=new int[size][2];
                int i=0;
                for(int[] r:list)
                {
                        res[i]=r;
                        i++;
                }
                return res;
        }
}
class Trie
{
        Trie[] children;
        boolean end;   /*indicate whether there is a word*/
        public Trie()
        {
                end=false;
                children=new Trie[26];
        }
}

class IndexPairs
{
        public static void main(String args[])
        {
                Scanner sc=new Scanner(System.in);
                String org=sc.nextLine();
                String arr[]=sc.nextLine().split(" ");
```

```
		int res[][]=new Solution().indexPairs(org, arr);
		for(int[] ans: res)
		{
			System.out.println(ans[0]+" "+ans[1]);
		}
	}
}
```

Sample Input-1:

```
---------------
thekmecandngitcolleges
kmec ngit colleges
```

Sample Output-1:

```
----------------
3 6
10 13
14 21
```

Sample Input-2:

```
---------------
xyxyx
xyx xy
```

Sample Output-2:

```
----------------
0 1
0 2
2 3
2 4
```

2. Longest word with all prefixes

Given an array of strings words, find the **longest** string in words such that **every prefix** of it is also in words.

- For example, let words = ["a", "app", "ap"].

  The string "app" has prefixes "ap" and "a", all of which are in words.

Return *the string described above. If there is more than one string with the same length, return the* ***lexicographically smallest*** *one, and if no string exists, return "".*

Example 1:

Input: words = [“k”,”ki”,”kir”,”kira”, “kiran”]

Output: “kiran”

Explanation: “kiran” has prefixes “kira”, “kir”, “ki”, and “k”, and all of them appear in words.

Example 2:

Input: words = [“a”, “banana”, “app”, “appl”, “ap”, “apply”, “apple”]

Output: “apple”

Explanation: Both “apple” and “apply” have all their prefixes in words. However, “apple” is lexicographically smaller, so we return that.

Example 3:

Input: words = [“abc”, “bc”, “ab”, “qwe”]

Output: “ ”

Solution:

Use a **trie** to store all the words. Then do depth first search on the **trie**. Search the nodes from left to right to make sure that the leftmost nodes (with lexicographically smallest letters) are visited first. For the words that all nodes are ends of words, update the longest word. Finally, return the longest word.

Java Program for Longest word with all prefixes using Trie DS: **LongestWord.java**

```
import java.util.*;
class trieHelper
{
	Trie root = new Trie();
	String res = "";
	public String longestWord(String[] words)
	{
		for (String word : words)
			addWord(word);
		for (String word : words)
			searchPrefix(word);
		return res;
	}
	private void searchPrefix(String word)
	{
		System.out.println("start res " + res + " word " + word);
		Trie cur = root;
		for (char c : word.toCharArray())
		{
			cur = cur.children[c-'a'];
			if (!cur.isWord) return;
		}
		if(res.equals(""))
		{
			res = word;
			return;
		}
		System.out.println("res " + res + " word " + word);
		if (res.length() < word.length() || res.length() == word.length() &&
								res.compareTo(word) > 0)
			res = word;
	}
	private void addWord(String word)
	{
		Trie cur = root;
		for(char c : word.toCharArray())
		{
			if (cur.children[c-'a'] == null)
				cur.children[c-'a'] = new Trie();
			cur = cur.children[c-'a'];
		}
		cur.isWord = true;
	}
}
class Trie
{
	Trie[] children = new Trie[26];
	boolean isWord;
}

class LongestWord
{
	public static void main(String args[])
	{
		Scanner sc=new Scanner(System.in);
		String arr[]=sc.nextLine().split(" ");
		System.out.println(new trieHelper().longestWord(arr));
	}
}
```

Sample Test case =1
input = k kmi km kmit kme kmec ksj ksjc ks kmecs
output = "kmecs"

Sample Test case =2
input = t tanker tup tupl tu tuple tupla
output = "tupla"

Sample Testcase =3
input = abc bc ab abcd
output = ""

3. Top K frequent words

We are given an array of strings words and an integer k, we have to return k strings which has highest frequency.
We need to return the answer which should be sorted by the frequency from highest to lowest and the words which has the same frequency sort them by their alphabetical order.

Example-1:
Input:
words = ["i","love","writing","i","love","coding"], k = 2
Output:
["i","love"]
Explanation: "i" and "love" are the two most frequent words.

Example-2:
Input:
words = ["it","is","raining","it","it","raining","it","is","is"], k = 3
Output:
["it","is","raining"]

Approach:
In this method, we will use Trie data structure and bucket sort

Trie data structure is like a tree which is used to store strings. It is also known as the digital tree or prefix tree. There can be maximum of 26 children in each node of the trie and the root node is always the null node.

Sorting is used for sorting elements by arranging them into multiple groups.

Algorithm:

- In this question the least frequency of any word can be greater than or equal to 1 and the maximum frequency is less than or equal to the length of the given vector.
- Then for each group we will define a trie to store the words of the same frequency.
- We don't have to sort the words inside each group as the words will be arranged in the alphabetical order inside trie.

Java Program for Top K frequent words:

```
import java.util.*;
class Node
{
	public char c;
	public boolean isWord;
	public int count;
	public Node[] children;
	public String str;
	public Node(char c)
	{
		this.c = c;
		this.isWord = false;
		this.count = 0;
		children = new Node[26];
		str = "";
	}
}

class Trie
{
	public Node root;
	public Trie()
	{
		this.root = new Node('\0');
	}
	public void insert(String word)
	{
		Node curr = root;
		for(int i  = 0;i<word.length();i++)
		{
			char c = word.charAt(i);
			if(curr.children[c-'a'] == null)
				curr.children[c-'a'] = new Node(c);
			curr = curr.children[c-'a'];
		}
		curr.isWord = true;
		curr.count +=1;
		curr.str = word;
	}
	public void traverse(Node root, PriorityQueue<Node> pq)
	{
		if(root.isWord == true)
			pq.offer(root);
		for(int i = 0;i<26;i++)
			if(root.children[i]!=null)
				traverse(root.children[i], pq);
	}
}
```

```
public class FrequentWord
{
        public static void main(String[] args)
        {
                Scanner sc = new Scanner(System.in);
                String line1 = sc.nextLine();
                int p = sc.nextInt();
                String[] words = line1.split(",");
                Trie t = new Trie();
                PriorityQueue<Node> pq = new PriorityQueue<>( (a,b)->
                {
                        if(a.count!=b.count)
                                return b.count-a.count;
                        return a.str.compareTo(b.str);
                } );

                for(int i = 0;i<words.length;i++)
                {
                        t.insert(words[i]);
                }
                t.traverse(t.root, pq);

                List<String> res = new ArrayList<>();
                int k = 0;
                while(k++<p)
                        res.add(pq.poll().str);

                System.out.println(res);
        }
}
```

Sample Test Cases

case =1

```
input =ball,are,case,doll,egg,case,doll,egg,are,are,egg,case,are,egg,are,case
3
output =[are, case, egg]
```

case =2

```
input =ball,are,case,doll,egg,case,doll,egg,are,are,egg,case,are,egg,are
3
output =[are, egg, case]
```

Suffix tree

A suffix tree is a data structure that is used to store all the suffixes of a given string in an efficient manner. It can be used to solve a wide range of string problems in linear time complexity, making it a powerful tool in competitive programming.

A suffix tree represents all the suffixes of a string in a tree-like data structure. The root of the tree represents the empty string, and each leaf of the tree represents a suffix of the string. Each internal node of the tree represents a common substring of two or more suffixes.

To build a suffix tree, we can start by creating a root node and adding edges to represent each character in the input string. Then, we add additional edges to represent all possible suffixes of the string, splitting the edges wherever there is a branching in the suffixes. This process continues until all suffixes have been added.

Once the suffix tree has been built, we can use it to search for substrings in the original string. We can start at the root of the tree and traverse the edges based on the characters in the substring we are searching for. If we can successfully traverse all the characters in the substring, we have found a match.

Suffix trees are very useful for solving a variety of string-related problems, such as finding the longest repeated substring in a string or finding all occurrences of a pattern in a text. They are also used in bioinformatics to analyze DNA sequences.

Here are some important concepts and operations/applications related to suffix trees:

Construction: A suffix tree can be constructed from a given string using various algorithms such as Ukkonen's algorithm. These algorithms use a concept called implicit suffix tree, where the tree is constructed as suffixes are added one by one.

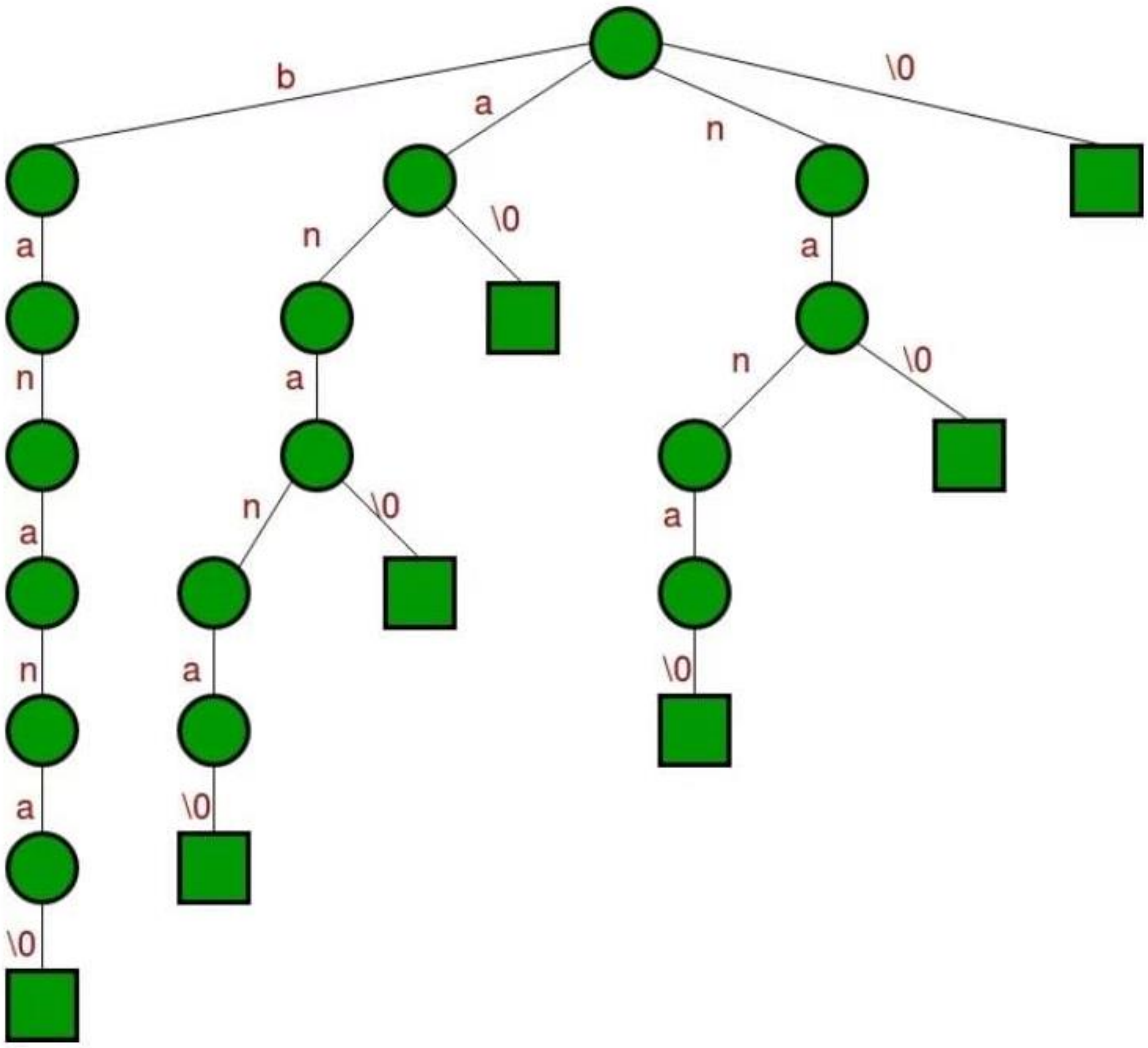
Searching: Once a suffix tree is constructed, it can be used to search for a given pattern in the string. This is done by traversing the tree based on the characters in the pattern. If the pattern is found, the tree can be used to find all the occurrences of the pattern in the string.

Longest common substring: A suffix tree can be used to find the longest common substring between two strings. This is done by first constructing a suffix tree for both strings and then finding the deepest node in the tree that has children from both strings.

Longest repeated substring: A suffix tree can also be used to find the longest repeated substring in a given string. This is done by finding the deepest node in the tree that has more than one leaf node.

Counting substrings: A suffix tree can be used to count the number of occurrences of all substrings of a given string in linear time complexity. This is done by counting the number of leaf nodes in the subtree rooted at each internal node.

These are some of the important concepts and operations related to suffix trees in competitive programming.



The above is the sample suffixtree for the word banana

Java program to implement suffixtree:

```
import java.util.*;
class Suffix
{
        static class SuffixTrieNode
        {
                static final int MAX_CHAR = 26;
                SuffixTrieNode[] children = new SuffixTrieNode[MAX_CHAR];
                SuffixTrieNode()
                {
                        for (int i = 0; i < MAX_CHAR; i++)
                                children[i] = null;
                }
                void insertSuffix(String s)
                {
                        if (s.length() > 0)
                        {
                                char cIndex = (char) (s.charAt(0) - 'a');
                                if (children[cIndex] == null)
                                {
                                        System.out.println("children null");
                                        children[cIndex] = new SuffixTrieNode();
                                }
                                children[cIndex].insertSuffix(s.substring(1));
                        }
                }
        }
        static class Suffix_trie
        {
                static final int MAX_CHAR = 26;
                SuffixTrieNode root;
                Suffix_trie(String s)
                {
                        root = new SuffixTrieNode();
                        for (int i = 0; i < s.length(); i++)
                                root.insertSuffix(s.substring(i));
                }
                int _countNodesInTrie(SuffixTrieNode node)
                {
                        if (node == null)
                                return 0;
                        int count = 0;
                        for (int i = 0; i < MAX_CHAR; i++)
                        {
                                if (node.children[i] != null)
                                        count += _countNodesInTrie(node.children[i]);
                        }
                        return (1 + count);
                }
```

```
		int countNodesInTrie()
		{
			return _countNodesInTrie(root);
		}

		static int countDistinctSubstring(String str)
		{
				Suffix_trie sTrie = new Suffix_trie(str);
				return sTrie.countNodesInTrie();
		}
		public static void main(String args[])
		{
				Scanner sc=new Scanner(System.in);
				System.out.println("Enter any string to construct suffix tree");
				String str=sc.nextLine();
				System.out.println("Count of distinct substrings is "
									+ (countDistinctSubstring(str) - 1));
		}
	}
}
```