SYLLABUS:-

Tree Algorithms: Fenwick Tree, Segment Tree – Applications-Range Sum Queries. Treap-Applications- K^{th} Largest Element in an Array. Page No: (01-34)

Trie: Introduction, Suffix Tree, Applications-Index Pairs of a String, Longest word with all prefixes, top K frequent words. Page No: (35-72)

Tree Algorithms:

Applications:

1. **Fenwick Tree.**
2. **Segment Tree – Applications-Range Sum Queries.**
3. **Treap- Applications- K^{th} Largest Element in an Array.**

1. **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.

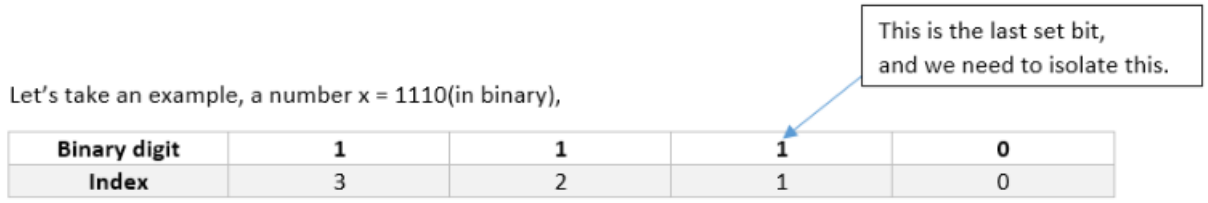
```
int a[] = {2, 1, 4, 6, -1, 5, -32, 0, 1};
void update(int i, int v)   //assigns value v to a[i]
{
    a[i] = v;
}
int prefixsum(int k)    //calculate the sum of all a[i] such that 0 <= i < k
{
   int sum = 0;
   for(int i = 0; i < k; i++)
        sum += a[i];
   return sum;
}
```

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



Binary digit	1	1	1	0
Index	3	2	1	0

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)

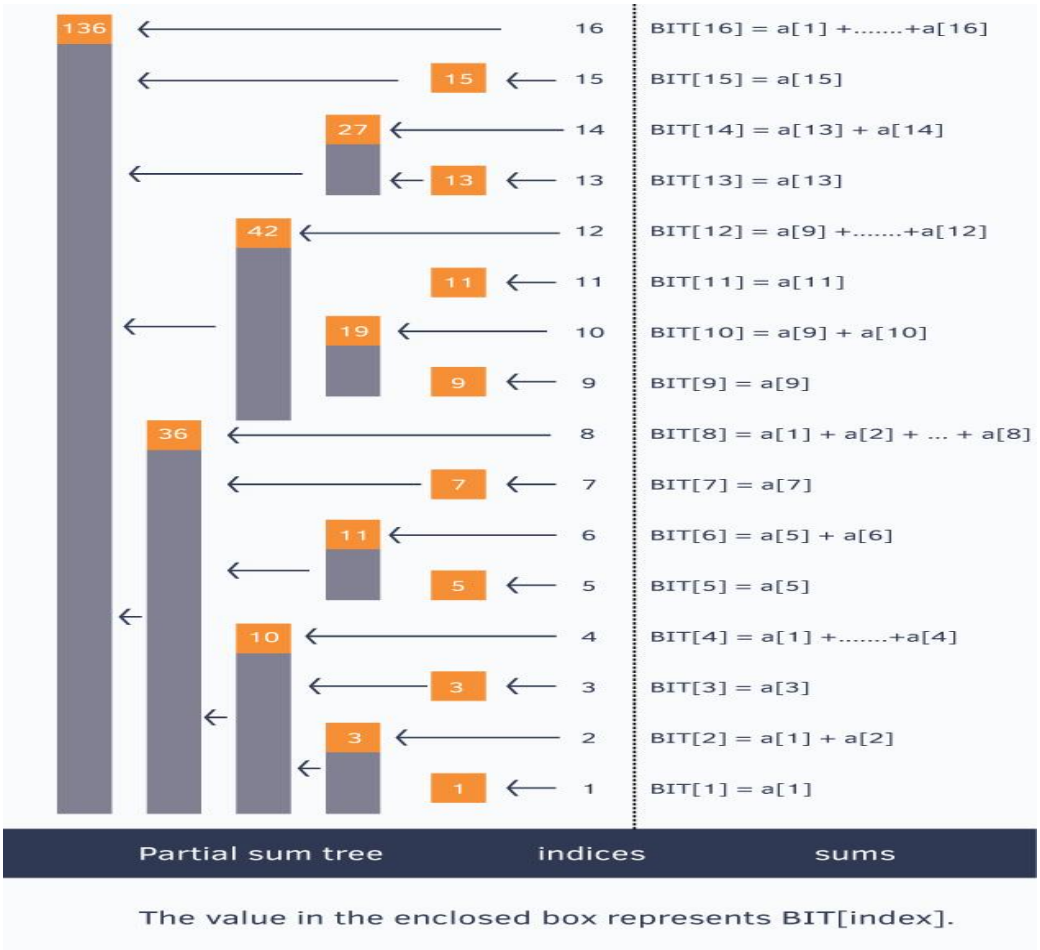
Basic Idea of Binary Indexed Tree:

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.

Java Program to implement Fenwick tree: 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

Out put :

75
86
80

2. 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.

Maximum number of nodes in segment Tree:

The **maximum number of nodes** in a Segment Tree depends on the size of the input array.

For an array of size **n**:

1. The **height** of the Segment Tree is $\lceil \log_2(n) \rceil$.
2. The **maximum number of nodes** in the Segment Tree is $2 \times 2^{\lceil \log_2(n) \rceil} - 1$.

Explanation:

- A segment tree is a **complete binary tree** if n is a power of 2.
- The **total number of nodes** in a perfect binary tree with n leaves is $2n-1$.
- If n is not a power of 2, we consider the next power of 2 greater than nnn, say $m=2^x$.
- The total nodes required will be $2m-1$.

Formula:

Max nodes=$2 \times 2^{\lceil \log_2(n) \rceil} - 1$.

Example Calculations:

For n=6

$\log_2(6)=3$
Maximum nodes = $2 \times 2^3 - 1 = 15$

For n=8
$\log_2(8)=3$
Maximum nodes = = $2 \times 2^3 - 1 = 15$

For n=10
$\log_2(10)=4$
Maximum nodes = $2 \times 2^4 - 1 = 31$.

Note: The number of nodes in a Segment Tree is at most O(4n) in the worst case.

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 children 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 children nodes.

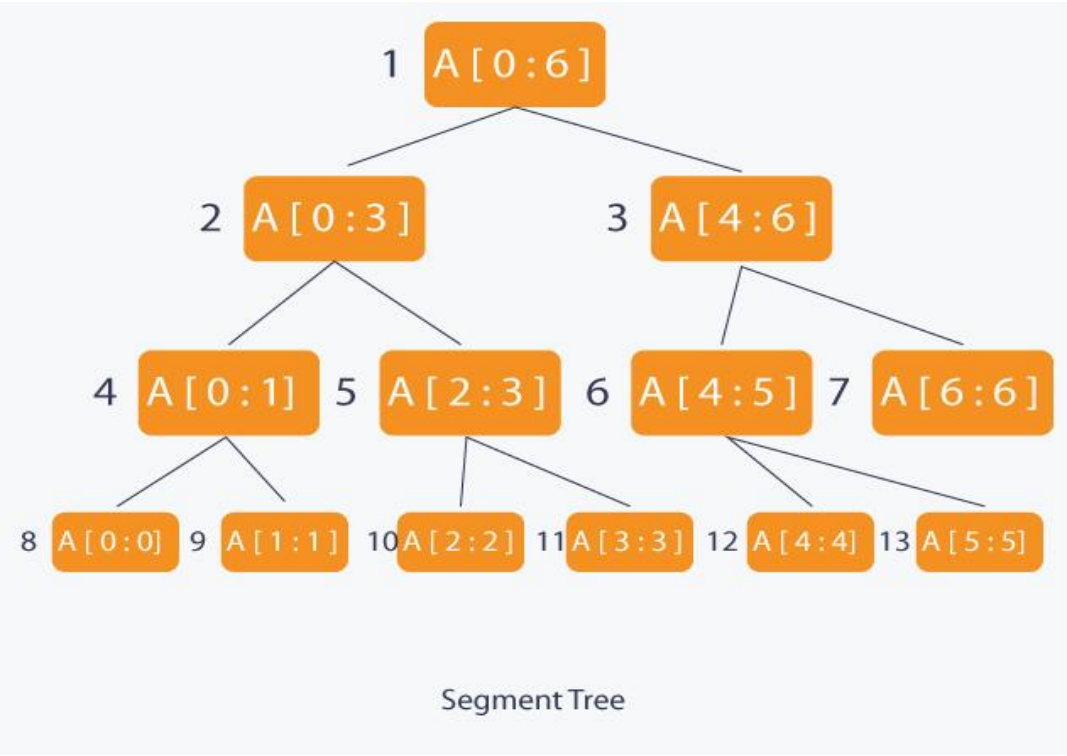
Once we know what we need to store in the segment tree we can build the tree using **recursion (bottom-up approach)**. We will start with the leaves and go up to the root and update the corresponding changes in the nodes that are in the path from leaves to root. Leaves represent a single element. In 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.

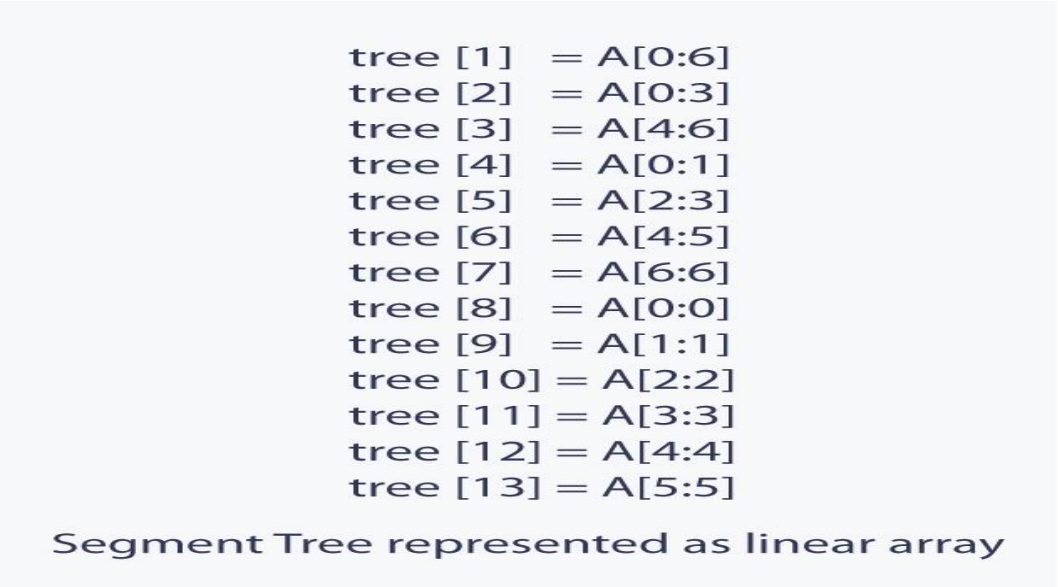
To make a query on the segment tree we will be given a range from **l to r**. We will recurse on the tree starting from the root and check if the interval represented by the node is completely in

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



Segment Tree represented as linear array

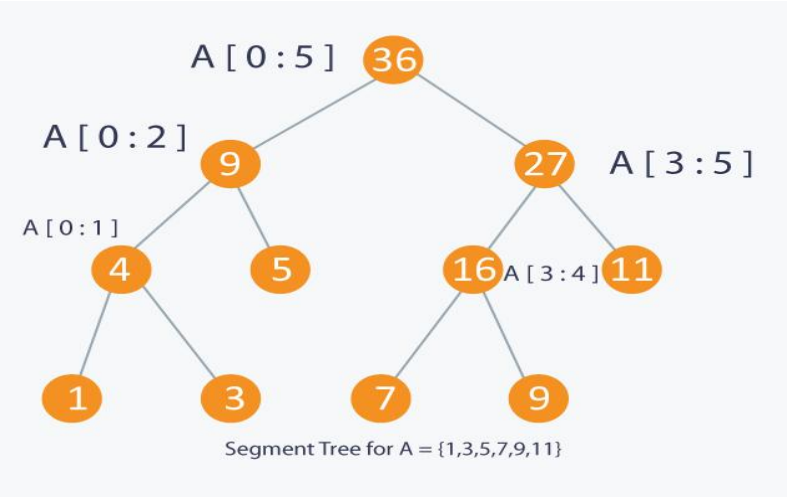
Applications: - Segment trees are used to find range sum queries.

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+1, start, mid);
      // Recurse on the right child
      build(2*node+2, mid+1, end);
      // Internal node will have the sum of both of its children
      tree[node] = tree[2*node+1] + tree[2*node+2];
   }
}
```

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+1** will represent the left node and **2*node + 2** represent the right node. **start** and **end** represents the interval represented by the node. Complexity of build() is **O(N)**.



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+1, start, mid, idx, val);
        }
        else
        {
            // if idx is in the right child, recurse on the right child
            update(2*node+2, 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+1, start, mid, l, r);
    int p2 = query(2*node+2, 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+1, start, mid, l, r, val);
    updateRange(node*2 + 2, mid + 1, end, l, r, val);
    // Use the result of children calls to update this node
    tree[node] = tree[node*2+1] + tree[node*2+2];
}
```

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+1] += lazy[node];                // Mark child as lazy
         lazy[node*2+2] += 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+1] += val;
         lazy[node*2+2] += val;
      }
      return;
   }
   int mid = (start + end) / 2;
   updateRange(node*2+1, start, mid, l, r, val);        // Updating left child
   updateRange(node*2 +2, mid + 1, end, l, r, val);   // Updating right child
   tree[node] = tree[node*2+1] + tree[node*2+2];        // 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+1] += lazy[node];        // Mark child as lazy
         lazy[node*2+2] += 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+1, start, mid, l, r);         // Query left child
    int p2 = queryRange(node*2 +2, mid + 1, end, l, r); // Query right child
    return (p1 + p2);
}
```

Example-1:

Java program for segment tree (get, update get range sum example):

SegmentTree.java

```
import java.util.*;

class SegmentTree
{
    private int[] tree;
    private int[] nums;
    private int n;

    // Constructor to initialize the Segment Tree
    public SegmentTree(int[] nums)
    {
        this.nums = nums;
        this.n = nums.length;
        // The size of the segment tree is 2 * 2^ceil(log2(n)) - 1
        int height = (int) Math.ceil(Math.log(n) / Math.log(2));
        int maxSize = 2 * (int) Math.pow(2, height) - 1;
        this.tree = new int[maxSize];
        buildTree(0, 0, n - 1);
    }

    // Build the Segment Tree
    private void buildTree(int treeIndex, int left, int right)
     {
        if (left == right)
        {
            tree[treeIndex] = nums[left];
            return;
```

```
    }
    int mid = left + (right - left) / 2;
    buildTree(2 * treeIndex + 1, left, mid); // Build left subtree
    buildTree(2 * treeIndex + 2, mid + 1, right); // Build right subtree
    tree[treeIndex] = tree[2 * treeIndex + 1] + tree[2 * treeIndex + 2]; // Merge results
}

// Query the total sum in the range [queryLeft, queryRight]
public int queryRange(int queryLeft, int queryRight)
{
    return queryRange(0, 0, n - 1, queryLeft, queryRight);
}

private int queryRange(int treeIndex, int left, int right, int queryLeft, int queryRight)
{
    // If the current segment is completely outside the query range
    if (right < queryLeft || left > queryRight)
    {
        return 0;
    }
    // If the current segment is completely inside the query range
    if (left >= queryLeft && right <= queryRight)
     {
        return tree[treeIndex];
    }
    // If the current segment overlaps with the query range
    int mid = left + (right - left) / 2;
    int leftSum = queryRange(2 * treeIndex + 1, left, mid, queryLeft, queryRight);
    int rightSum = queryRange(2 * treeIndex + 2, mid + 1, right, queryLeft, queryRight);
    return leftSum + rightSum;
}

// Update the value at index `index` to `newValue`
public void update(int index, int newValue)
 {
    int diff = newValue - nums[index];
    nums[index] = newValue;
    updateTree(0, 0, n - 1, index, diff);
}

private void updateTree(int treeIndex, int left, int right, int index, int diff)
 {
    // If the index is outside the current segment
```

```
    if (index < left || index > right)
    {
       return;
    }
    // If the index is within the current segment
    tree[treeIndex] += diff;
    if (left != right)
    {
       int mid = left + (right - left) / 2;
       updateTree(2 * treeIndex + 1, left, mid, index, diff); // Update left subtree
       updateTree(2 * treeIndex + 2, mid + 1, right, index, diff); // Update right subtree
    }
}

// Get the total sum of the entire array
public int getTotalSum()
 {
   return tree[0]; // The root of the segment tree contains the total sum
}

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)
      {
         // call sumrange(s1,s2)
         int s1 = scan.nextInt();
         int s2 = scan.nextInt();
         System.out.println(st.queryRange(s1,s2));
      }
      else{
         int ind = scan.nextInt();
```

```
            int val= scan.nextInt();
            st.update(ind,val);
          }
       }

    }
}
```

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
```

Example-2:

Java program for segment tree (getMax, update getMaxrange): `SegmentTreeMax.java`

```
import java.util.*;

class SegmentTreeMax
{
   private int[] tree;
   private int[] nums;
   private int n;

   // Constructor to initialize the Segment Tree
   public SegmentTreeMax(int[] nums)
   {
      this.nums = nums;
      this.n = nums.length;
      // The size of the segment tree is 2 * 2^ceil(log2(n)) - 1
      int height = (int) Math.ceil(Math.log(n) / Math.log(2));
      int maxSize = 2 * (int) Math.pow(2, height) - 1;
      this.tree = new int[maxSize];
      buildTree(0, 0, n - 1);
```

```
    }

    // Build the Segment Tree
    private void buildTree(int treeIndex, int left, int right)
{
        if (left == right) {
            tree[treeIndex] = nums[left]; // Leaf node stores the value
            return;
        }
        int mid = left + (right - left) / 2;
        buildTree(2 * treeIndex + 1, left, mid); // Build left subtree
        buildTree(2 * treeIndex + 2, mid + 1, right); // Build right subtree
        // Merge results: store the maximum of left and right subtrees
        tree[treeIndex] = Math.max(tree[2 * treeIndex + 1], tree[2 * treeIndex + 2]);
    }

    // Query the maximum value in the range [queryLeft, queryRight]
    public int queryRangeMax(int queryLeft, int queryRight) {
        return queryRangeMax(0, 0, n - 1, queryLeft, queryRight);
    }

    private int queryRangeMax(int treeIndex, int left, int right, int queryLeft, int queryRight) {
        // If the current segment is completely outside the query range
        if (right < queryLeft || left > queryRight) {
            return Integer.MIN_VALUE; // Return minimum value to avoid affecting the result
        }
        // If the current segment is completely inside the query range
        if (left >= queryLeft && right <= queryRight) {
            return tree[treeIndex];
        }
        // If the current segment overlaps with the query range
        int mid = left + (right - left) / 2;
        int leftMax = queryRangeMax(2 * treeIndex + 1, left, mid, queryLeft, queryRight);
        int rightMax = queryRangeMax(2 * treeIndex + 2, mid + 1, right, queryLeft, queryRight);
        return Math.max(leftMax, rightMax);
    }

    // Update the value at index `index` to `newValue`
    public void update(int index, int newValue)
     {
        nums[index] = newValue;
        updateTree(0, 0, n - 1, index, newValue);
    }
```

```
    private void updateTree(int treeIndex, int left, int right, int index, int newValue) {
        // If the index is outside the current segment
        if (index < left || index > right) {
            return;
        }
        // If the current segment is a leaf node
        if (left == right) {
            tree[treeIndex] = newValue;
            return;
        }
        // If the index is within the current segment
        int mid = left + (right - left) / 2;
        updateTree(2 * treeIndex + 1, left, mid, index, newValue); // Update left subtree
        updateTree(2 * treeIndex + 2, mid + 1, right, index, newValue); // Update right subtree
        // Merge results: update the current node with the maximum of left and right subtrees
        tree[treeIndex] = Math.max(tree[2 * treeIndex + 1], tree[2 * treeIndex + 2]);
    }

    // Get the maximum value of the entire array
    public int getMax() {
        return tree[0]; // The root of the segment tree contains the maximum value
    }

    public static void main(String[] args)
     {
            int[] nums = {1, 3, 5, 7, 9, 11};
        SegmentTreeMax st = new SegmentTreeMax(nums);

        // Query the maximum value in the range [1, 4]
        System.out.println("Maximum value in range [1, 4]: " + st.queryRangeMax(1, 4)); // Output:
9

        // Update the element at index 2 to 10
        st.update(2, 10);

        // Query the maximum value in the range [1, 4] after update
        System.out.println("Maximum value in range [1, 4] after update: " + st.queryRangeMax(1,
4)); // Output: 10

        // Get the maximum value of the entire array
        System.out.println("Maximum value of the array: " + st.getMax()); // Output: 11
```

```
 }
}
```

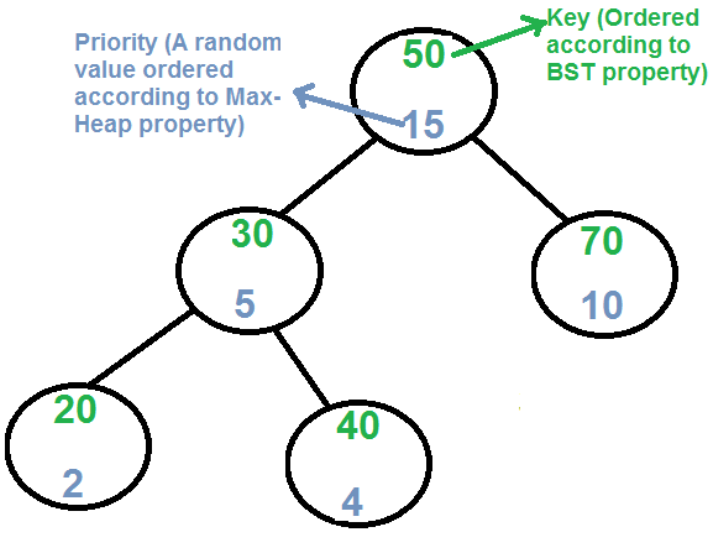
3. Treap:

- 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 abinary 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.



The following are the features of Treap Datastructure.

- 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.

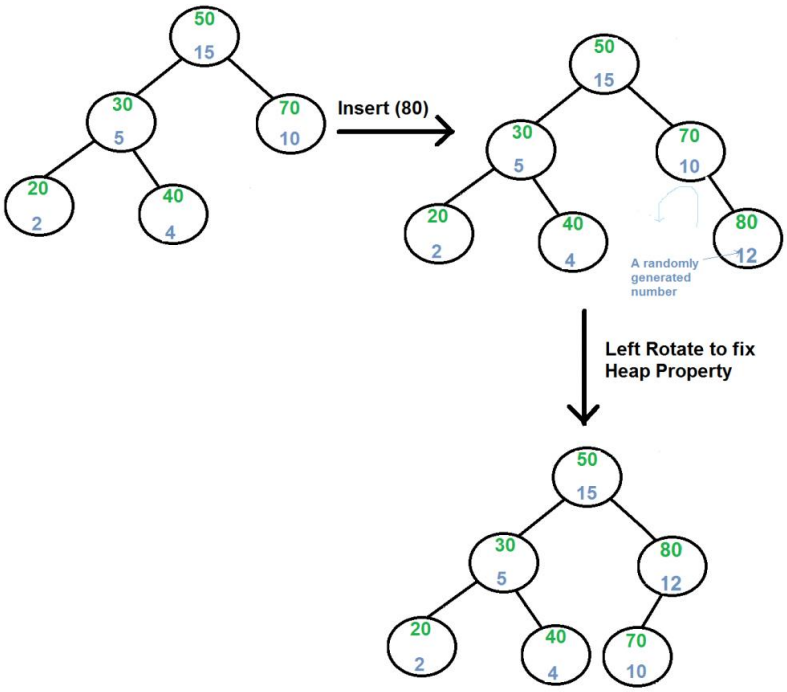
- It is simply a binary search tree, therefore to print a sorted order of keys, traverse it in the same way as we would conventional BSTs. Searching for a treap is similar to searching for a tree.

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 BSTinsert.
3. Use rotations to make sure that inserted node's priority follows max heap property.

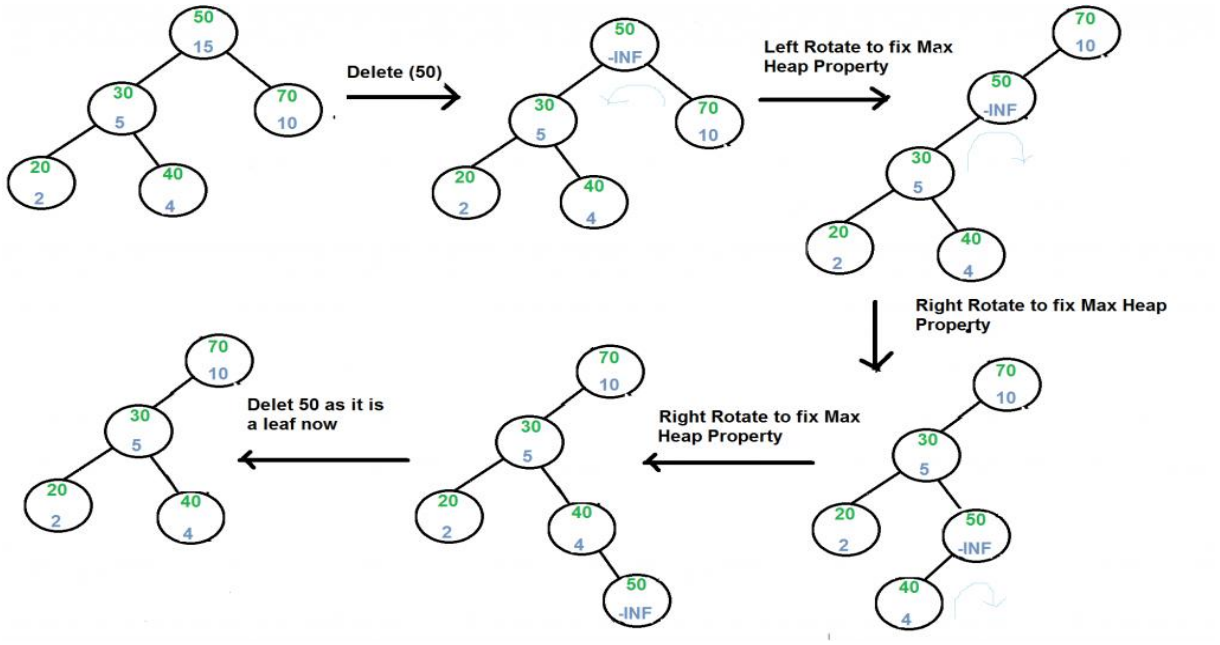


2. Deletion in Treap

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 a node Example:

Assume we have inserted 5, 2, 8, 1, 3 with the following random priorities:

```
      5 (90)
     /     \
  2 (75)   8 (80)
   /   \
1(40) 3(50)
```

Step 1: Delete 2

2 has two children (1 and 3).

Compare priority of left (40) and right (50) → Right child has higher priority.

Left rotation on 2 moves 3 up:

```
      5 (90)
     /     \
  3 (50)   8 (80)
  /
2 (75)
/
1(40)
```

Now, 2 has only one child (1), so replace 2 with 1.

Step 2: Final Treap After Deletion

```
      5 (90)
     /     \
```

```
      3 (50)   8 (80)
     /
   1(40)
```

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);
}
```

4. Left Rotation in Treap:

Example:

```
10 (50)
     \
     20 (80)
     /
   15 (30)
```

Node values (key) with random priority values (in brackets):

10 has priority 50
20 has priority 80 (higher than 10)
15 has priority 30
Since 20 has higher priority than 10, a left rotation at 10 is required.

Applying Left Rotation on Node 10
Now, let's apply leftRotate(10) using the given method:

```
public static TreapNode leftRotate(TreapNode x)
{
   TreapNode y = x.right;  // y = 20
   TreapNode T2 = y.left;  // T2 = 15
   y.left = x;           // Move x under y (20 becomes new root)
   x.right = T2;          // Assign 15 as right child of x
   return y;             // Return new root (y)
```

```
}
```

Step-by-Step Execution
Identify nodes:

x = 10
y = x.right = 20
T2 = y.left = 15
Perform rotation:
20 (y) becomes the new root.
10 (x) moves down to become the left child of 20.
15 (T2) becomes the right child of 10.

After Left Rotation

```
    20 (80)   <-- New Root
   /
 10 (50)
   \
    15 (30)
```

5. Right Rotation in Treap:

```
    20 (50)
   /
 10 (80)
   \
    15 (30)
```

Node values (key) with random priority values (in brackets):

20 has priority 50
10 has priority 80 (higher than 20)
15 has priority 30
Since 10 has higher priority than 20, a right rotation at 20 is required.

Applying Right Rotation on Node 20:

Now, let's apply rightRotate(20) using the given method:

```
public static TreapNode rightRotate(TreapNode
)
 {
   TreapNode x = y.left;  // x = 10
   TreapNode T2 = x.right;  // T2 = 15
   x.right = y;  // Move y under x (10 becomes new root)
   y.left = T2;  // Assign 15 as left child of y
   return x;  // Return new root (x)
}
```

Step-by-Step Execution Identify nodes:

y = 20
x = y.left = 10
T2 = x.right = 15

Perform rotation:
10 (x) becomes the new root.
20 (y) moves down to become the right child of 10.
15 (T2) becomes the left child of 20.

After Right Rotation:

```
  10 (80)
    \
     20 (50)
      /
   15 (30)
```

Java program for Treap (Insert,Seacrh and delete Operations) Treap.java

```
import java.util.*;

class TreapNode
    {
        int key, priority;
        TreapNode left, right;
    }
class Treap
 {
   public static TreapNode rightRotate(TreapNode y)
    {
        TreapNode x = y.left;
        TreapNode T2 = x.right;
        x.right = y;
        y.left = T2;
        return x;
     }
   public static TreapNode leftRotate(TreapNode x)
    {
        TreapNode y = x.right;
        TreapNode T2 = y.left;
        y.left = x;
        x.right = T2;
```

```
        return y;
    }
public static TreapNode newNode(int key)
 {
        TreapNode temp = new TreapNode();
        temp.key = key;
        //Generates a random priority (between 0 and 99) using Math.random().
         //This priority ensures that the Treap maintains heap properties.
        temp.priority = (int)(Math.random() * 100);
        temp.left = temp.right = null;
        return temp;
 }
public static TreapNode insertNode(TreapNode root, int key)
 {
        if (root == null)
         {
                return newNode(key);
         }
        //If key is smaller than or equal to the current node's key, insert it into the left subtree.
        if (key <= root.key)
        {
                root.left = insertNode(root.left, key);
        //If the newly inserted node (now in root.left) has a higher priority than the root node:
        //Perform a right rotation to bring it up.
                if (root.left.priority > root.priority)
                {
                    root = rightRotate(root);
                }
        }
        //If key is greater than the root's key, insert it into the right subtree.
         else
         {
                root.right = insertNode(root.right, key);
          //If the newly inserted node (now in root.right) has a higher priority than the root:
          //Perform a left rotation to bring it up.
                if (root.right.priority > root.priority)
                 {
                        root = leftRotate(root);
                  }
          }
        return root;
 }
 public static TreapNode deleteNode(TreapNode root, int key)
```

```
{
    System.out.println("DeleteNode key " + key + " root.key " + root.key);
 if (root == null)
 return root;

 if (key < root.key)//If key is smaller, move left.
 root.left = deleteNode(root.left, key);
 else if (key > root.key) //If key is greater, move right.
 root.right = deleteNode(root.right, key);

 //If no left child, replace root with root.right.
 else if (root.left == null)
 {
    TreapNode temp = root.right;
    root = temp;
 }

 //If no right child, replace root with root.left.
 else if (root.right == null)
 {
     TreapNode temp = root.left;
     root = temp;
 }

 //Left rotation is performed if the right child has a higher priority.
 //The node gets pushed down, and deletion continues.

 else if (root.left.priority < root.right.priority)
 {
      root = leftRotate(root);
      root.left = deleteNode(root.left, key);
 }

 //Right rotation is performed if the left child has a higher priority.
 //The node gets pushed down, and deletion continues.
 else
 {
     root = rightRotate(root);
     root.right = deleteNode(root.right, key);
 }
  return root;
}
public static TreapNode search(TreapNode root, int key)
```

```
    {
        if (root == null || root.key == key)
          return root;
        if (root.key < key)
          return search(root.right, 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);
         }
    }
  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);
    }
 }
```

Input:-

6
2 3 4 5 1 7

Output:-

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)

K^{th} 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 suppose to print the K'th largest height in the sorted order of heights[].

Not the K'th distinct height.

Explanation

1. Convert k-th Largest to k-th Smallest

- The **inorder traversal** of a **Binary Search Tree (BST)** gives elements in **sorted order** (ascending).
- To find the **k-th largest element**, we convert it into finding the **(n - k + 1)-th smallest element**.
- Formula:

  $k = n - p + 1$
  - n = Total number of elements.
  - p = k-th largest element to find.
  - k = Equivalent smallest element.

  Example Calculation:

  - Given n = 6, p = 3 (Find 3rd largest).
  - k = 6 - 3 + 1 = 4 → Find **4th smallest element** instead.

2. Construct the Treap

- A **Treap** is a **BST** where nodes also have **random priorities**.
- Nodes are inserted using **BST rules**:
  - **Left subtree** contains **smaller** values.
  - **Right subtree** contains **larger** values.
- After insertion, rotations **maintain heap order**:
  - **Max Heap Property:** Parent priority must be **greater** than children's priority.
  - **Rotations (Left/Right) are performed** when heap order is violated.

3. Perform Inorder Traversal

- **Inorder traversal (Left → Root → Right)** visits elements in **sorted order**.

- **Decrement k for each visited node**:
  - When k == 0, **print the node's value** (this is our answer).
- The traversal ensures that we efficiently find the **k-th smallest element**, which corresponds to the **p-th largest**.

Time Complexity Analysis

Operation	Complexity
Insertion in Treap	**O(log n)** (Expected)
Inorder Traversal	**O(n)** (Worst-case)
Finding k-th element	**O(k) ≈ O(log n)**

Java Program for Kth 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);
  }
  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);
        }
        inorder(root);
    }
}
```

Example-1:
Input =
6 3
2 4 3 1 6 5
Output =
4

Example-2:
Input =
6 2
3 2 1 5 6 4
Output =
5

Trie Data Structure:

Introduction, Suffix Tree,

Applications

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

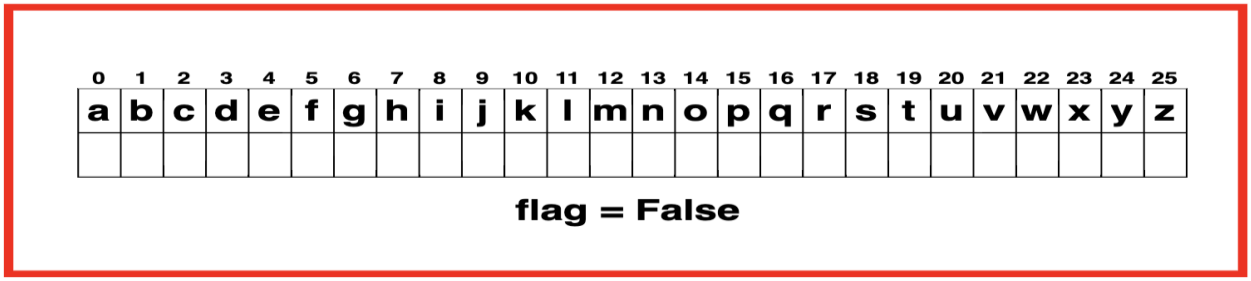
Trie Introduction:

The Trie data structure is used to efficiently store and retrieve a set of strings.

It organises strings such that common prefixes are shared among strings, making operations like searching for words with a given prefix efficient. Trie allows for quick retrieval of all strings with a given prefix, making it highly efficient for autocomplete and predictive text applications.

A Trie node is a data structure used to construct Trie. Each node contains the following components:

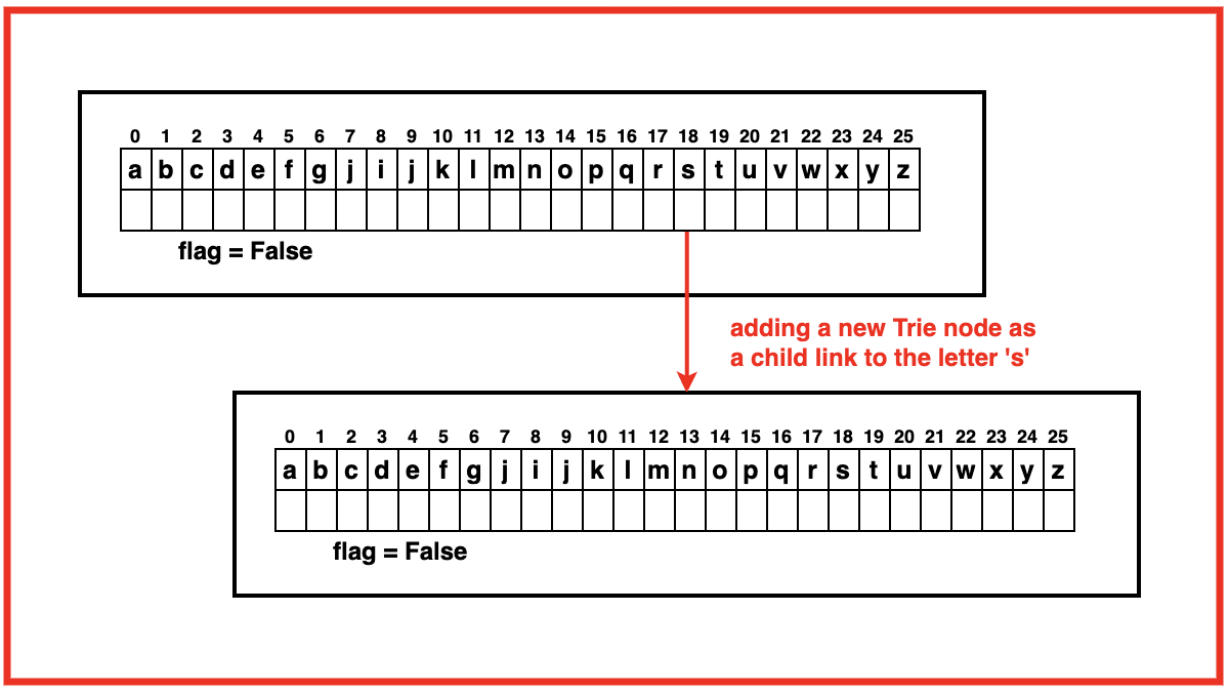
- Links to Child Nodes: A Trie node contains an array of pointers called “links” or “pointer to children” for each letter of the lowercase alphabet. These pointers represent connections to child nodes corresponding to each letter of the alphabet. For instance, the link at index 0 corresponds to the child node representing the letter 'a', the link at index 1 corresponds to 'b', and so forth.
- Flag for End of Word: Each Trie node contains a boolean flag indicating whether the node marks the end of a word. This flag is essential for distinguishing between prefixes and complete words stored in the Trie.



Each node in the trie nodes support several operations:

- **Contains Key:** This operation checks whether a specific letter (or key) exists as a child node of the current Trie node. It returns true if the letter is present, indicating a valid path in the Trie.

- **Get Child Node:** Given a letter, this operation retrieves the corresponding child node of the current Trie node. If the letter is present, it returns the pointer to the child node; otherwise, it returns nullptr, signifying the absence of the letter.
- **Put Child Node:** This operation establishes a connection between the current Trie node and a child node representing a particular letter. It sets the link at the corresponding index to point to the provided child node.
- **Set End Flag:** Marks the current Trie node as the end of a word. This flag is crucial for determining whether a string stored in the Trie terminates at this node, indicating a complete word.
- **Is End of Word:** Checks whether the current Trie node signifies the end of a word by examining the end flag. It returns true if the node marks the end of a word; otherwise, it returns false.



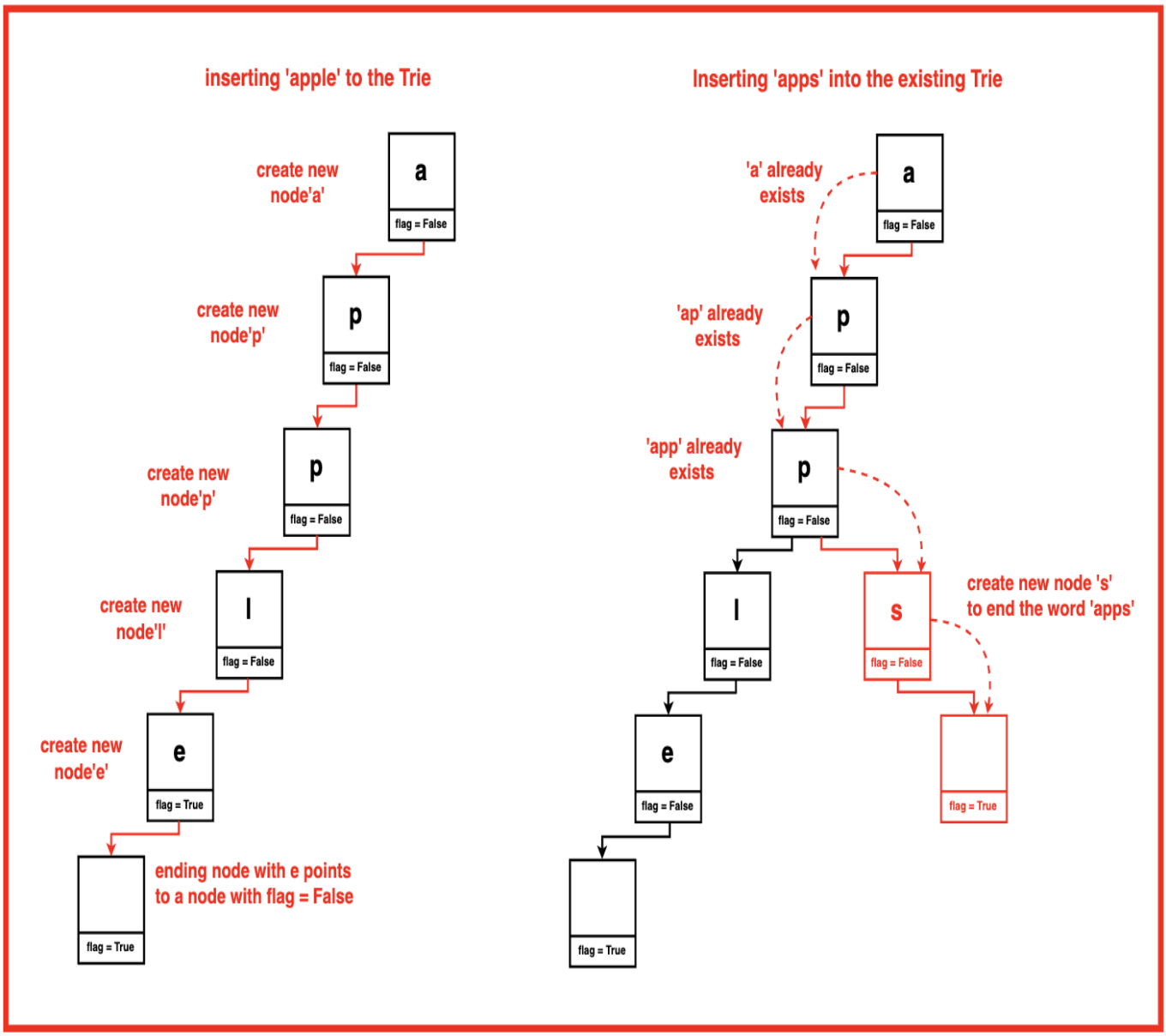
Algorithm 1: To Insert a Node in the Trie:

Step 1: Start at the root node.

Step 2: For each character in the word:

- Check if the current node has a child node corresponding to the character.
- If not, create a new node and link it as a child of the current node.
- Move to the child node corresponding to the character.
-

Step 3: Once all characters are inserted, mark the end of the word by setting the flag of the last node to true.



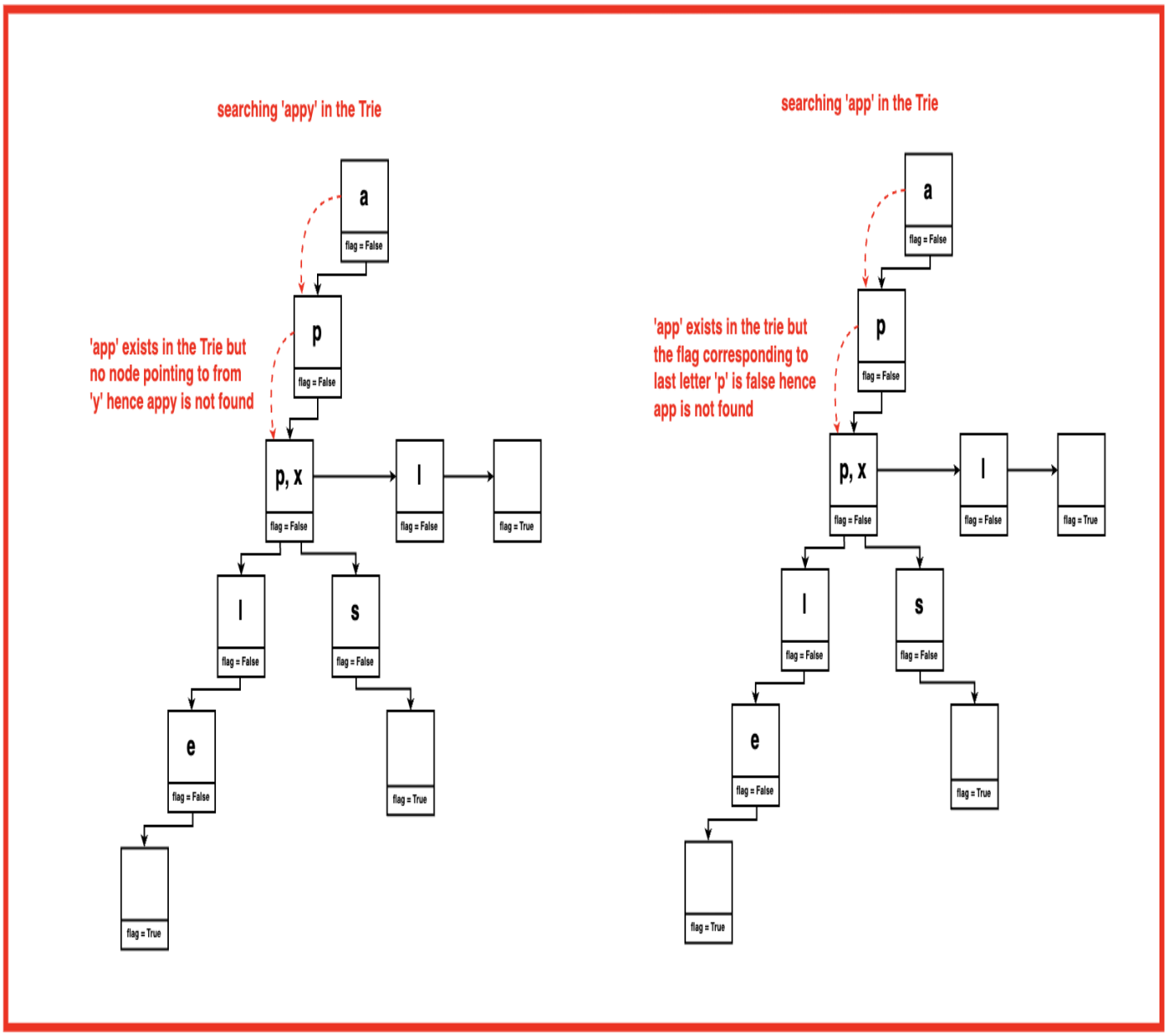
Algorithm 2: To Search for a word in the Trie:

Step 1:Start at the root node.

Step 2: For each character in the word:

- Check if the current node has a child node corresponding to the character.
- If not, the word is not in the Trie.
- Move to the child node corresponding to the character.

Step 3: After processing all characters, check if the flag of the last node is set to true. If yes, the word is found; otherwise, it is not.



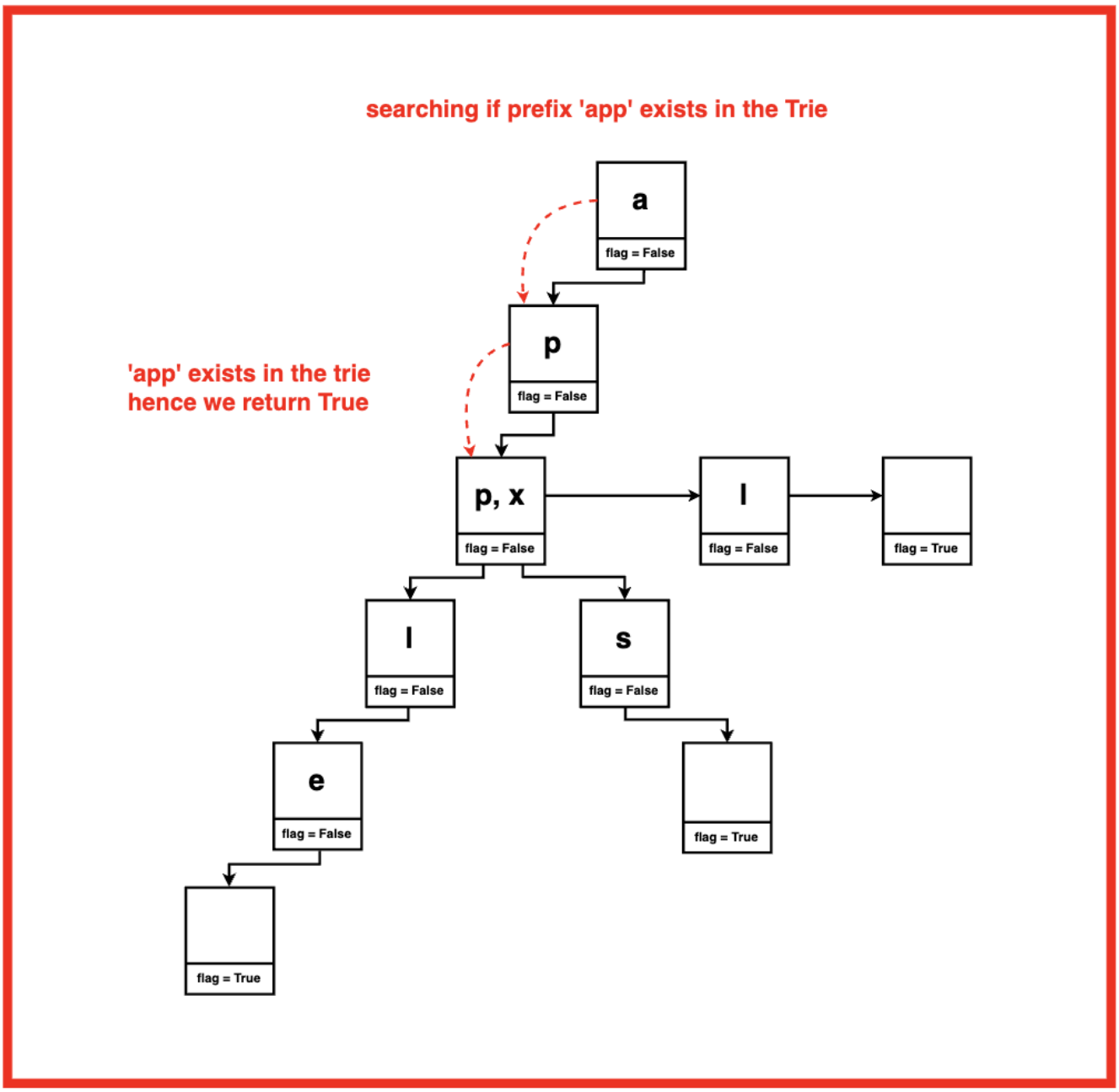
Algorithm 3: Check if Trie contains prefix:

Step 1:Start at the root node.

Step 2: For each character in the prefix:

- Check if the current node has a child node corresponding to the character.
- If not, there is no word with the given prefix.
- Move to the child node corresponding to the character.

Step 3: If all characters of the prefix are found, return true indicating the existence of words with the given prefix.


searching if prefix 'app' exists in the Trie
'app' exists in the trie
hence we return True
a
flag = False
p
flag = False
p, x
flag = False
l
flag = False
flag = True
l
flag = False
s
flag = False
e
flag = False
flag = True
flag = True

Algorithm 4 Delete a String form Trie:

1. Start at the root
- Begin from the root of the Trie.

2. Recursively traverse each character of the word
- For each character ch in the word:
  - Go to the corresponding child node (node.get(ch)).
  - If the child does not exist → **word not present**, return false.

3. Base Case – End of word reached
- When you reach the last character:
  - If isEnd() is false, the word isn't actually stored → return false.
  - Otherwise, **unset the end flag** (mark that this node is no longer the end of a valid word).

4. Check if current node has no other children
- If the current node has no children and isn't end of another word:
  - Return true to indicate this node can be deleted from its parent.

5. Backtrack and delete unnecessary nodes
- If a child node returns true (can be deleted):
  - Set the reference to that child to null.
  - Then check the current node: if it has no other children and isn't the end of another word, return true.

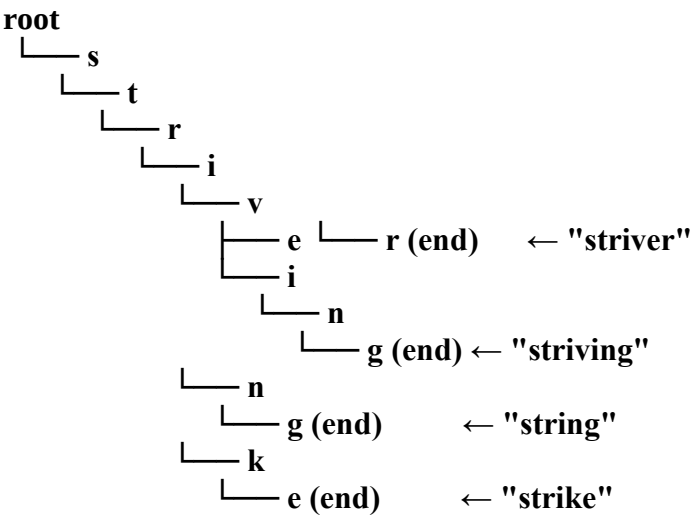
6. Return status
- Return true only if nodes are safely deletable.
- Return false if the word was not found.

Words inserted into the Trie:
1. striver
2. striving
3. string
4. strike

All words share a common prefix: stri

Step-by-Step: Trie Before Deletion

Let's say the Trie looks like this (simplified tree format):



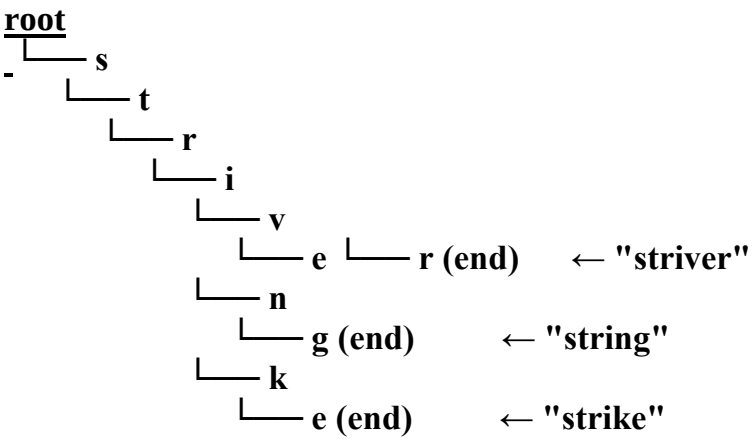
Delete("striving")

We want to delete **only "striving"**, not affecting "striver" or "string".

 Recursive Steps:

1. Start at root → 's' → 't' → 'r' → 'i' → 'v' → 'i' → 'n' → 'g'
2. We reach 'g' which is the end of "striving":
   - flag = true → we **unset it** → flag = false
3. Now check: does 'g' have any children? → **No**
   - So 'g' can be deleted from 'n'
4. Check 'n': any children left? → **No** → delete 'n' from 'i'
5. Check 'i': any children left? → `'e'` (from `"striver"`) → ✗ **Keep** `'i'`

Trie After Deletion of "striving"



We only remove nodes **if**:

- They're **not used by other words**, and
- They're **not marking the end of another word**

This makes Trie deletion both **safe and efficient**!

Pseudocode Summary:

```
delete(word):
   if not search(word): return false
   call deleteHelper(root, word, index = 0)
   return true

deleteHelper(currentNode, word, index):
   if index == word.length:
      currentNode.unsetEnd()
      return currentNode has no children
   ch = word.charAt(index)
   next = currentNode.get(ch)
   if next is null: return false
   shouldDelete = deleteHelper(next, word, index+1)
   if shouldDelete:
      currentNode.removeLink(ch)
      return currentNode has no children and is not end of any other word
   return false
```

Complexity Analysis

Time Complexity:

Insertion: O(N) where N is the length of the word being inserted. This is because we have to iterate over each letter of the word to find its corresponding node or create a node accordingly.

Search: O(N) where N is the length of the word being searched for. This is because in Trie search we traverse over each letter for the word from the root, checking if the current node contains a node at the index of the next letter. This process repeats until we reach the end of the word or encounter a node without the next letter.
Prefix Search: O(N) where N is the length of the prefix being searched for. Similar to searching for words, in prefix search we also iterate over each letter of the word to find its corresponding node.

Space Complexity: O(N) where N is the total number of characters across all unique words inserted into the Trie. For each character in a word, a new node may need to be created leading to space proportional to the number of characters.

Advantages of Trie:

1. **Fast Search Time (O(m)):**
   Time complexity for searching a word is **O(m)** where *m* is the length of the word.
   Unlike hash tables or binary search trees, the lookup time **doesn't depend on the number of stored words**, only on the word's length.

2. Efficient Prefix Queries

- Easily supports operations like:
  - startsWith(prefix)
  - Auto-complete systems
  - Word prediction
- Efficiently finds all words that share a common prefix.

3. Sorted/Ordered Output

- Trie stores data **in lexicographical order** (alphabetically sorted), which makes it ideal for:
  - Suggestion systems
  - Dictionary lookup

4. Avoids Duplicate Storage

- Shared prefixes are stored once, saving memory compared to storing all full words individually (especially useful with large word sets and many common prefixes).

5. Safe from Hash Collisions

- Unlike HashMap or HashSet, Trie **does not rely on hash functions**, so:
  - No hash collisions
  - No need for a good hash function
  - Predictable structure

6. Supports Advanced Search Features

- With some modifications, Tries can support:
  - Wildcard matching (?, *)
  - Longest prefix match (used in IP routing)
  - Ternary search for space optimization

- Word frequency tracking
- Suffix Trie/Tree for substring queries

7. Space-Time Tradeoff Customizable

- Tries can be optimized for space using:
  - **Compressed Trie (Radix Trie)**
  - **Ternary Search Trie**
  - **Suffix Tree (for substrings)**

Write a java Program to Implement a Trie Data Structure which supports the following Four operations:

Search (word): To check if the string `word` is present in the Trie or not.
Insert (word): To insert a string `word` in the Trie.
Start With(word): To check if there is a string that has the prefix `word`.
Delete(word) : To Delete a sting 'word' in the Trie.

Example 1:

Enter strings:
striver striving string strike
Strings are inserted in Trie.
Enter a word to search:
striving
True
Enter a prefix to check:
stri
True
Enter a word to delete:
striving
Deleted successfully

```
import java.util.*;

 class Trie
  {
    // Node structure for Trie
    static class Node
    {
       // Array to store links to child nodes,each index represents a letter

       Node[] links = new Node[26];

       // Flag indicating if the node marks the end of a word
       boolean flag = false;

       // Check if the node contains a specific key (letter)
       boolean containsKey(char ch)
```

```
    {
        return links[ch - 'a'] != null;
    }

    // Insert a new node with a specific key (letter) into the Trie
    void put(char ch, Node node)
    {
        links[ch - 'a'] = node;
    }

    // Get the node with a specific key (letter) from the Trie
    Node get(char ch)
    {
        return links[ch - 'a'];
    }

    // Set the current node as the end of a word
    void setEnd()
    {
        flag = true;
    }

    void unsetEnd()
    {
        flag = false;
    }

    // Check if the current node marks the end of a word
    boolean isEnd() {
        return flag;
    }

    boolean isEmpty()
    {
        for (int i = 0; i < 26; i++)
        {
            if (links[i] != null) return false;
        }
        return true;
    }
}

// Trie class
private final Node root;

// Constructor to initialize the Trie with an empty root node
public Trie()
{
```

```
    root = new Node();
}

 // Inserts a word into the Tri Time Complexity O(len), where len is the length of the word
public void insert(String word)
{
    Node node = root;
    for (char ch : word.toCharArray())
    {
        if (!node.containsKey(ch))
        {
            // Create a new node for the letter if not present
            node.put(ch, new Node());
        }
         // Move to the next node
        node = node.get(ch);
    }
     // Mark the end of the word
    node.setEnd();
}

// Returns if the word is in the trie
public boolean search(String word)
 {
    Node node = root;
    for (char ch : word.toCharArray())
     {
        if (!node.containsKey(ch))
         {
             // If a letter is not found,the word is not in the Trie
            return false;
        }
         // Move to the next node
        node = node.get(ch);
    }
     // Check if the last node marks the end of a word
    return node.isEnd();
}

 // Returns if there is any word in the trie that starts with the given prefix
public boolean startsWith(String prefix)
 {
    Node node = root;
    for (char ch : prefix.toCharArray())
    {
        if (!node.containsKey(ch))
         {
             // If a letter is not found, there is no word with the given prefix
```

```
            return false;
        }
        // Move to the next node
        node = node.get(ch);
    }
    // The prefix is found in the Trie
    return true;
}

 public boolean delete(String word)
{
    if (!search(word))
    {
        return false; // Word not present in Trie
    }
    deleteHelper(root, word, 0);
    return true; // Word was present and logically deleted
}

private boolean deleteHelper(Node current, String word, int index)
{
    if (index == word.length())
    {
        current.unsetEnd(); // Unset the end of word
        return current.isEmpty(); // If no children, node can be removed
    }

    char ch = word.charAt(index);
    Node next = current.get(ch);
    if (next == null)
    {
        return false;
    }

    boolean shouldDeleteNextNode = deleteHelper(next, word, index + 1);

    if (shouldDeleteNextNode) //If the child node should be deleted:
    {
        //Remove the reference to that child node (ch) effectively deleting that character's node
from the Trie
        current.links[ch - 'a'] = null;

        //!current.isEnd() → This node is not the end of another word
        //current.isEmpty() → This node has no other children
        //If both are true, it means this node is also useless, and can be deleted too.
        //So return true to the previous level.

        return !current.isEnd() && current.isEmpty();
```

```
        }

        return false;
    }

    // Main method for testing
    public static void main(String[] args)
    {
        Trie trie = new Trie();
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter strings:");
        String[] str = sc.nextLine().split(" ");
        for (String s : str)
        {
            trie.insert(s);
        }

        System.out.println("Strings are inserted in Trie.");

        // Search
        System.out.println("Enter a word to search:");
        String searchWord = sc.next();
        System.out.println(trie.search(searchWord) ? "True" : "False");

        // Prefix check
        System.out.println("Enter a prefix to check:");
        String prefix = sc.next();
        System.out.println(trie.startsWith(prefix) ? "True" : "False");

        // Delete
        System.out.println("Enter a word to delete:");
        String deleteWord = sc.next();
        boolean deleted = trie.delete(deleteWord);
        System.out.println(deleted ? "Deleted successfully" : "Word not found");

    }
}
```

Applications:

1. Index Pairs of a String:

Given a string text and an array of strings words, return an array of all index pairs [i, j] so that the substring text[i...j] is in words. Return the pairs [i, j] in sorted order (i.e., sort them by their first coordinate, and in case of ties sort them by their second coordinate).

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].

Approach:

Step 1: Build a Trie from the words array

- Create a Trie (prefix tree) where:
  - Each node has an array of 26 children (for lowercase English letters).
  - Each node tracks whether it is the end of a word (isEndOfWord flag).
- Insert every word from words into the Trie.

Why a Trie?

- It allows **fast prefix matching**.
- Instead of checking every substring against every word, we **traverse the Trie** as we move through text.

Step 2: Traverse the text

- For every starting index i in text:
  - Start from the root of the Trie.
  - Traverse characters from i to the end of the string.
  - At each character:
    - Check if the corresponding child exists in the Trie.
    - If it doesn't → break (no matching word starts here).
    - If it does and the node is the end of a word → we found a match! Add [i, j] to the result list.

Step 3: Sorting (if needed)

- Since we go from left to right (i = 0 to end), and j increases with i, **pairs are naturally in sorted order**.
- But if required by constraints, you can sort the list of pairs explicitly by i, then j.

Time Complexity Analysis:

Let:

- `n = text.length()`
- `L = average/max word length`
- `k = number of words`

Building the Trie:

- O(k * L)

Searching in text:

- Worst-case O(n * L), as for each `i`, we may go up to `L` characters while matching.

So total: **O(k * L + n * L)** — very efficient compared to a brute-force $O(n^2)$.

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

```
import java.util.*;

public class IndexPairs
{
   // Trie Node definition
   static class TrieNode
   {
      TrieNode[] children = new TrieNode[26];
      boolean isEndOfWord = false;
   }

   // Root of the Trie
   TrieNode root = new TrieNode();

   // Insert a word into the Trie
   private void insert(String word)
    {
      TrieNode node = root;
      for (char c : word.toCharArray())
      {
         int idx = c - 'a';
         if (node.children[idx] == null)
            node.children[idx] = new TrieNode();
         node = node.children[idx];
      }
      node.isEndOfWord = true;
   }

   // Main function to get index pairs
   public List<int[]> indexPairs(String text, String[] words)
```

```
{
    // Step 1: Build the Trie
    for (String word : words)
        insert(word);

    List<int[]> result = new ArrayList<>();

    // Step 2: Traverse the text
    for (int i = 0; i < text.length(); i++)
    {
        TrieNode node = root;
        int j = i;
        while (j < text.length())
        {
            int idx = text.charAt(j) - 'a';

            if (node.children[idx] == null)
                break;

            node = node.children[idx];

            if (node.isEndOfWord)
            {
                result.add(new int[]{i, j});
            }
            j++;
        }
    }

    // Step 3: Result is already in sorted order due to traversal logic
    return result;
}

public static void main(String args[])
{
    Scanner sc=new Scanner(System.in);
    String text=sc.nextLine();
    String words[]=sc.nextLine().split(" ");
    IndexPairs solution = new IndexPairs();
    printResult(solution.indexPairs(text,words));
}
    private static void printResult(List<int[]> pairs)
    {
```

```
        for (int[] pair : pairs) {
            System.out.println(Arrays.toString(pair));
        }
    }

}
```

Example-1:
input=
thestoryofleetcodeandme
story fleet leetcode
output=
[3, 7]
[9, 13]
[10, 17]

Example-2:
input=
abcabcabc
abc abca ab
output=
[0, 1]
[0, 2]
[0, 3]
[3, 4]
[3, 5]
[3, 6]
[6, 7]
[6, 8]

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: “ ”

Algorithm/Intuition:

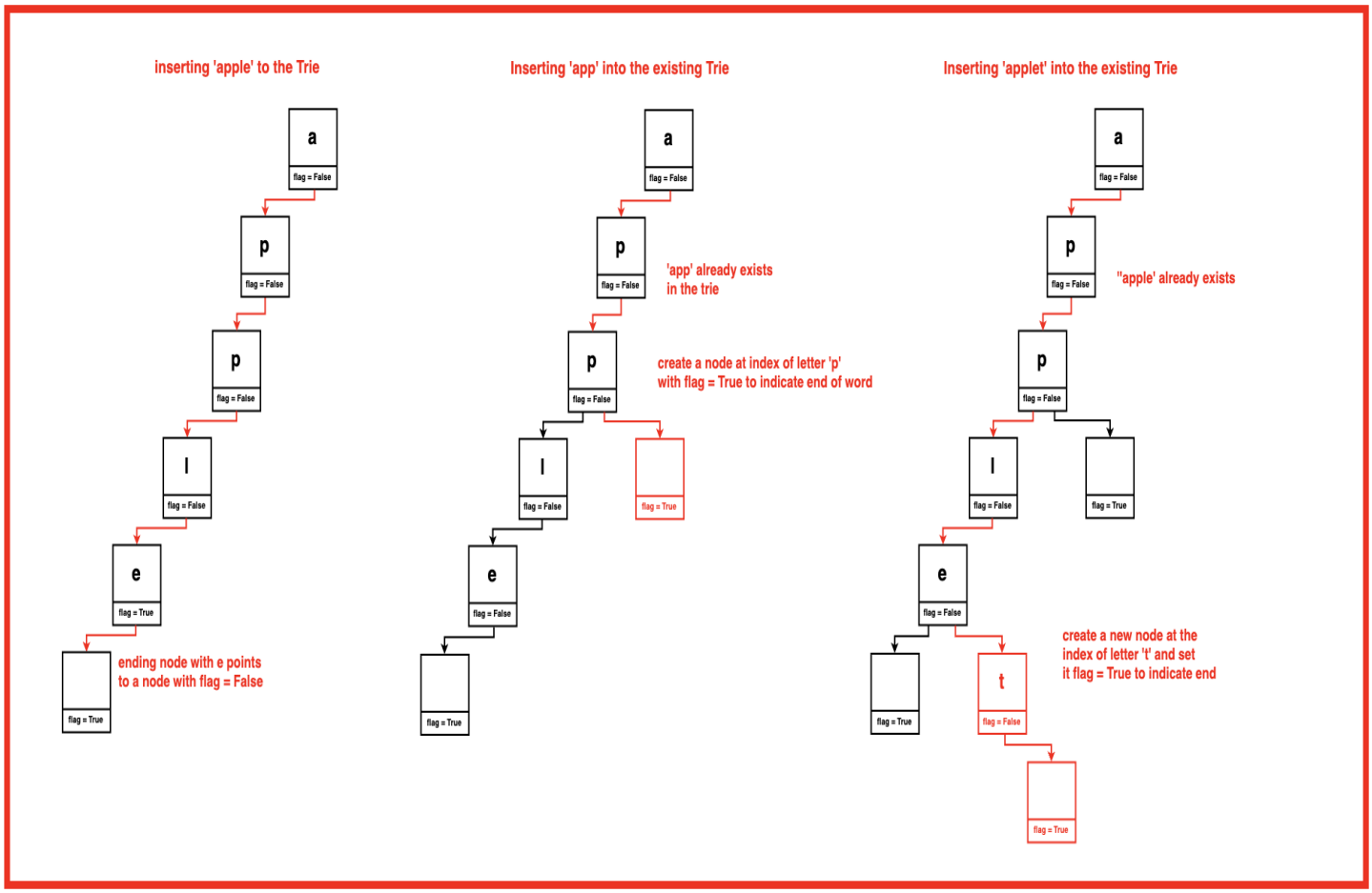
We create a Trie because we need to efficiently check whether every prefix of a given string is present in the array. Without a Trie, we would need to iterate through the array for each prefix check, resulting in a time complexity that is exponential in the length of the strings.

By constructing a Trie from the array of words, we can efficiently check whether each prefix exists in the array. This is because each node in the Trie represents a character, and the paths from the root to each node represent the prefixes present in the array. Traversing the Trie allows us to quickly verify whether a given prefix exists.

The longest string will all prefixes refer to the string in a given set that is longest and also has every prefix present in the same set.

Algorithm 1:

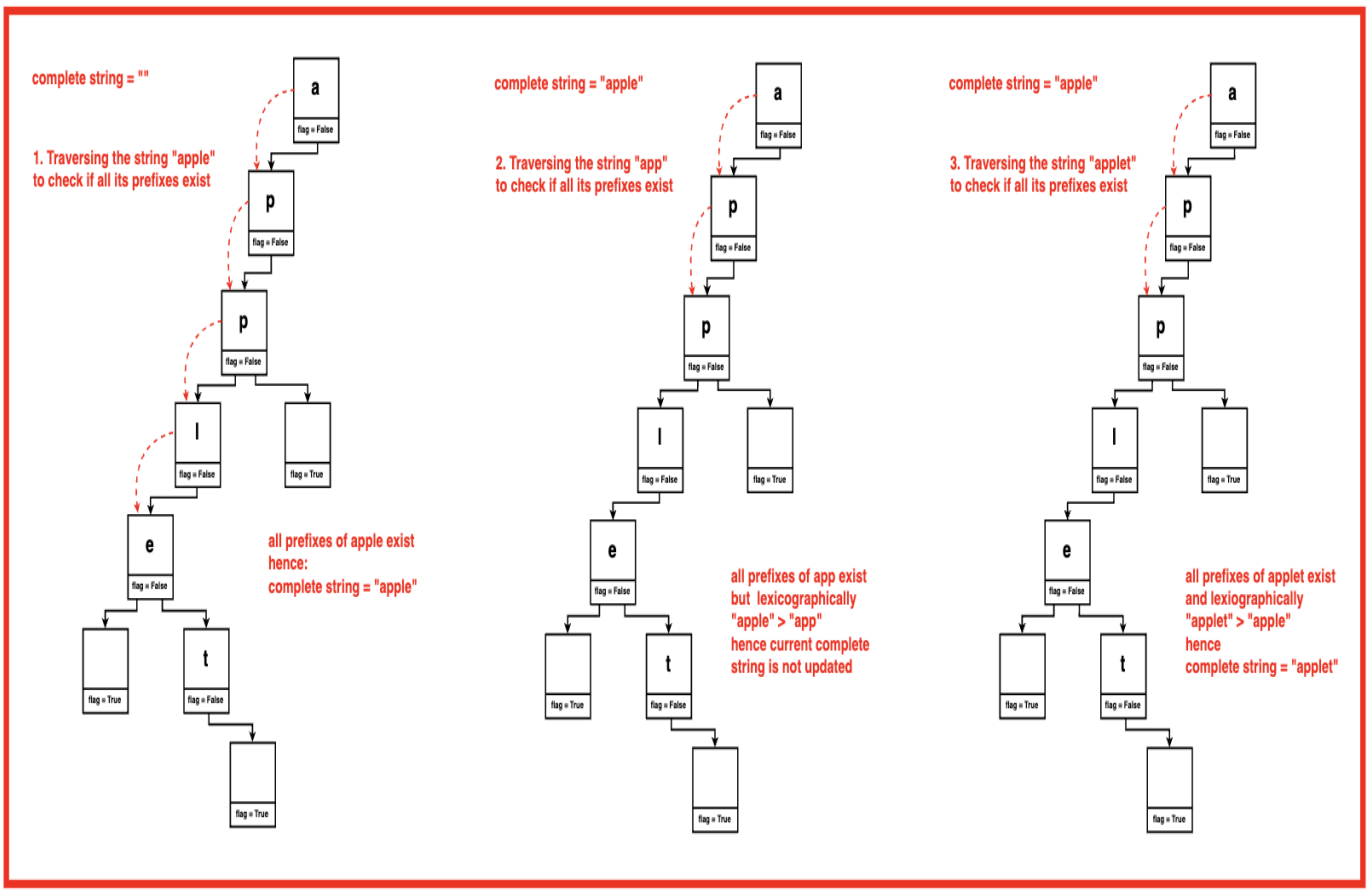
Step 1: Iterate over each word in the input vector and insert each word into the Trie.



Step 2: Initialise an empty string that is set at the complete string initially.

Step 3: Iterate over each word in the input vector again and check if all prefixes of the word exist in the Trie.

- If all prefixes exist: Compare the word's length and the lexicographical order with the longest complete string found so far.
- Update the longest complete string if current word > global complete string length.



Step 4: If no complete string is found, return None. Otherwise return the longest complete string found.

Time Complexity:

O(2N) where N is the total number of characters of all unique words hence the time complexity.

- **Insertion into Trie:** O(N) To insert each word into the Trie, we iterate over all its characters and insert them by creating nodes. To insert all words, the time complexity becomes O(N).
- **Finding Longest Complete String:** O(N): Involves iterating over each word in the input vector and checking if all prefixes in the Trie exist. Hence the time complexity for this is also O(N).

Space Complexity: O(N) where N is the total number of characters of all unique words hence the time complexity.

- Each node of the Trie contains an array of pointers to child nodes, which typically takes constant 26 pointers for lowercase English letters.
- The number of nodes to be created will be proportional to the number of characters in all unique words hence O(N).

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

```
import java.util.*;

class Node
 {
   Node[] links;
   // Array to hold links to  child nodes for each character
   boolean flag;
   // Flag to indicate if it  marks the end of a word

   // Constructor to   initialize the node
   public Node() {
      links = new Node[26];
      // Initialize array for   links to child nodes
      flag = false;
      // Initialize flag as false, indicating it's not the end of a word
   }

   // Method to check if the node contains a link for a given character
   public boolean containsKey(char ch)
   {
      // Check if the link forthe character exists
      return links[ch - 'a'] != null;
   }

   // Method to get the node corresponding to a given character
   public Node get(char ch)
   {
       // Return the corresponding child node
      return links[ch - 'a'];
   }

   // Method to set the link for  a given character to a node
   public void put(char ch, Node node)
   {
      // Set the link for the character to the provided node
      links[ch - 'a'] = node;
   }

   // Method to mark the node as the end of a word
   public void setEnd()
    {
      // Set the flag to indicate the end of a word
      flag = true;
   }

   // Method to check if the node marks the end of a word
```

```
    public boolean isEnd()
    {
        // Return the flag indicating if it's the end of a word
        return flag;
    }
}

class Trie
 {
     // Root node of the Trie
    private Node root;

    // Constructor to initialize the Trie
    public Trie()
    {
        // Create a new root node for the Trie
        root = new Node();
    }

    // Method to insert a word into the Trie
    public void insert(String word)
     {
        Node node = root;
        // Start traversal from the root node
        for (int i = 0; i < word.length(); i++)
         {
            // Iterate through each character of the word
            char ch = word.charAt(i);
             // If the character doesn't exist as a child node
            if (!node.containsKey(ch))
             {
                // Create a new node for the character
                node.put(ch, new Node());
            }
            // Move to the next node
            node = node.get(ch);
        }
         // Mark the last node as the end of the word
        node.setEnd();
    }

    // Method to check if all prefixes of a word exist in the Trie
    public boolean checkIfAllPrefixExists(String word)
     {
        Node node = root;
        // Start traversal from the root node
        boolean flag = true;
        // Initialize flag as true
```

```
        for (int i = 0; i < word.length() && flag; i++)
        {
            // Iterate through each character of the word
            char ch = word.charAt(i);
            if (node.containsKey(ch))
            {
                // If the character exists as a child node
                node = node.get(ch);
                // Move to the next node
                flag = flag && node.isEnd();
                // Update flag based on whether it's the end of a word
            }
            else
             {
                // Return false if the prefix doesn't exist
                return false;
            }
        }
        // Return true if all prefixes exist
        return flag;
    }
}

public class LongestWord
{
    // Function to find the longest complete string in a given array of strings
    public static String completeString(int n, String[] a)
     {
        Trie obj = new Trie();
        // Create a Trie object

        for (String word : a)

            obj.insert(word);

            // Insert each word into the Trie
        String longest = "";
        // Initialize the variable to store the longest complete string
        for (String word : a)
        {
            // Iterate through each word in the array
            if (obj.checkIfAllPrefixExists(word))
             {
                // Check if all prefixes of the word exist
                if (word.length() > longest.length() || (word.length() == longest.length()  &&
word.compareTo(longest) < 0))
                 {
                    // Update the longest string if the current word is longer or lexicographically smaller
```

```
                longest = word;
            }
        }
    }
    // Return "None" if no  complete string found
    if (longest.equals(""))
    return "None";
    // Return the longest complete string
    return longest;
  }

  public static void main(String[] args)
   {

       Scanner sc=new Scanner(System.in);
       String dict[]=sc.nextLine().split(" ");
       System.out.println(completeString(dict.length,dict));
    }

}
```

Example-1:
input = k kmi km kmit kme kmec ksj ksjc ks kmecs
output = "kmecs"

Example-2:
input = t tanker tup tupl tu tuple tupla
output = "tupla"

Example-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:

1.Count Frequencies: Use a HashMap<String, Integer> to count the frequency of each word.

2.Insert into Trie: Insert each unique word into a Trie, and store the frequency at the end node of each word.

3.DFS on Trie: Traverse the Trie using DFS to collect words and their frequencies.

4.Use Priority Queue (Min-Heap):

->Build a min-heap of size k.

->Compare based on:

-> Higher frequency

-> If frequency is same → lexicographically smaller first

Java Program for Top K frequent words:

```
import java.util.*;

class TrieNode
{
   TrieNode[] children;
   boolean isEndOfWord;
   int frequency;  // store frequency at end node
   String word;    // store the word itself at end node for reference

   TrieNode()
   {
      children = new TrieNode[26];
      isEndOfWord = false;
      frequency = 0;
      word = null;
   }
}

class Trie
{
   TrieNode root;

   Trie()
   {
      root = new TrieNode();
   }

   public void insert(String word, int freq)
   {
      TrieNode node = root;
      for (char c : word.toCharArray())
      {
         int idx = c - 'a';
         if (node.children[idx] == null)
          {
            node.children[idx] = new TrieNode();
         }
         node = node.children[idx];
      }
      node.isEndOfWord = true;
      node.frequency = freq;
      node.word = word;
   }

   public void collectWords(TrieNode node, PriorityQueue<StringFrequency> pq, int k)
   {
```

```
        if (node == null) return;

        if (node.isEndOfWord)
         {
            pq.offer(new StringFrequency(node.word, node.frequency));
            if (pq.size() > k) {
                pq.poll(); // remove lowest priority element
            }
        }

        for (TrieNode child : node.children)
         {
            if (child != null)
             {
                collectWords(child, pq, k);
            }
        }
    }
}

class StringFrequency
{
    String word;
    int freq;

    StringFrequency(String word, int freq)
     {
        this.word = word;
        this.freq = freq;
    }
}

// Custom comparator for min-heap
class WordComparator implements Comparator<StringFrequency>
 {
    public int compare(StringFrequency a, StringFrequency b)
     {
        if (a.freq == b.freq)
        {
            return b.word.compareTo(a.word); // reverse for min-heap
        }
        return Integer.compare(a.freq, b.freq); // min-heap based on frequency
    }
}

public class FrequentWord
 {
    public static List<String> topKFrequent(String[] words, int k)
```

```
    {
      // Step 1: Count frequency
      Map<String, Integer> freqMap = new HashMap<>();
      for (String word : words) {
         freqMap.put(word, freqMap.getOrDefault(word, 0) + 1);
      }

      // Step 2: Build Trie
      Trie trie = new Trie();
      for (Map.Entry<String, Integer> entry : freqMap.entrySet())
       {
         trie.insert(entry.getKey(), entry.getValue());
      }
      // Step 3: Use PriorityQueue to find top k
      PriorityQueue<StringFrequency> pq = new PriorityQueue<>(new WordComparator());

      // Step 4: DFS to collect words into priority queue
      trie.collectWords(trie.root, pq, k);

      // Step 5: Build result list from heap (reverse order)
      List<String> result = new ArrayList<>();
      while (!pq.isEmpty()) {
         result.add(pq.poll().word);
      }
      Collections.reverse(result); // since min-heap gives smallest first

      return result;
   }
   public static void main(String[] args)
   {
      Scanner sc=new Scanner(System.in);
      String dict[]=sc.nextLine().split(",");
      int k=sc.nextInt();
      System.out.println(new FrequentWord().topKFrequent(dict,k));
   }
}
```

Example-1:

input =

ball,are,case,doll,egg,case,doll,egg,are,are,egg,case,are,egg,are,case

3

output =

[are, case, egg]

Example-2:

input =

ball,are,case,doll,egg,case,doll,egg,are,are,egg,case,are,egg,are

3

output =

[are, egg, case]

Suffix Tree:

Suffix tree is a compressed trie of all the suffixes of a given string. Suffix trees help in solving a lot of string related problems like pattern matching, finding distinct substrings in a given string, finding longest palindrome etc. In this following points will be covered:

- Compressed Trie
- Suffix Tree

Before going to suffix tree, let's first try to understand what a compressed trie is.
Consider the following set of strings: **{"banana","nabd","bcdef","bcfeg","aaaaaa","aaabaa" }**

A standard trie for the above set of strings will look like:

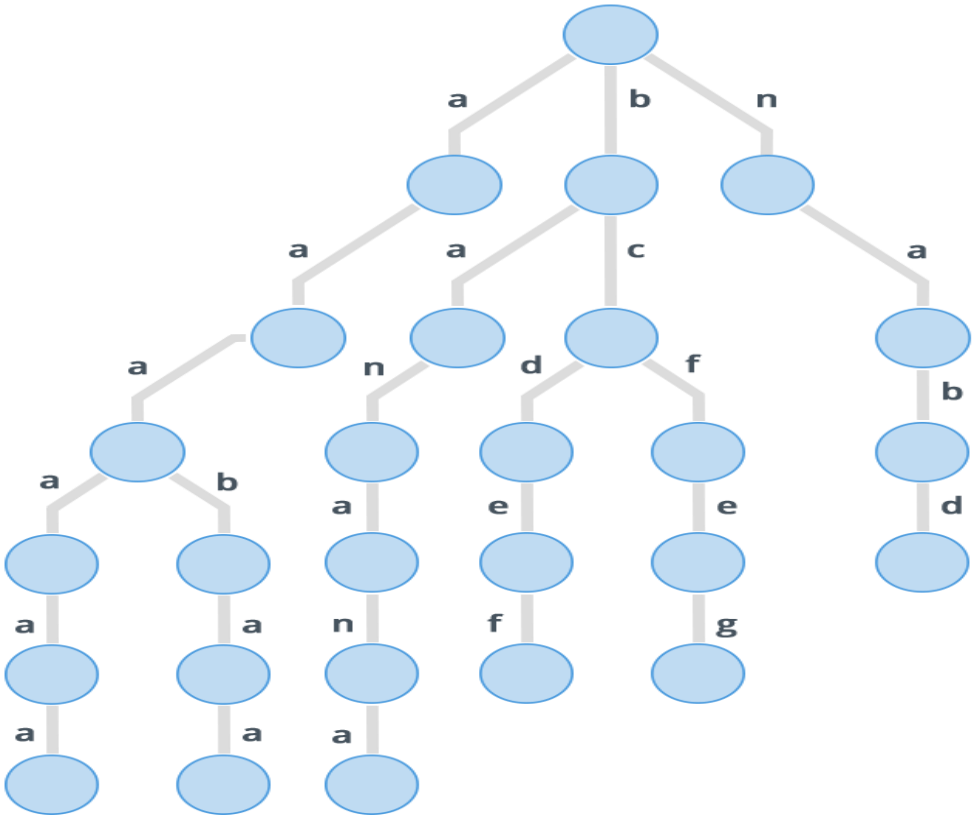


Figure: A standard trie for the above set of strings.

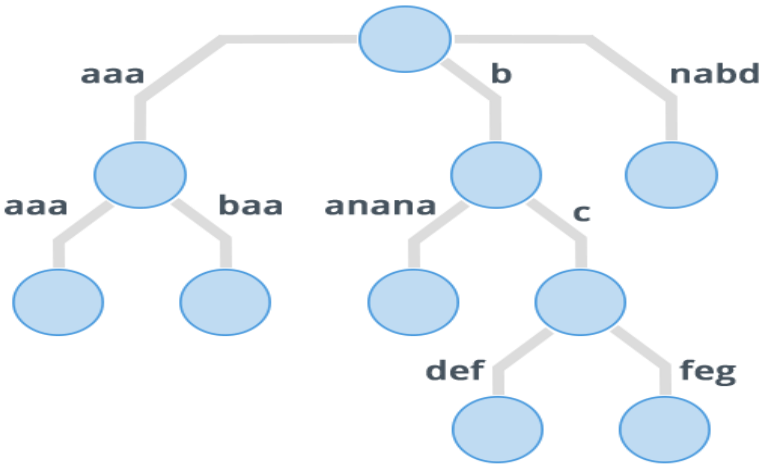


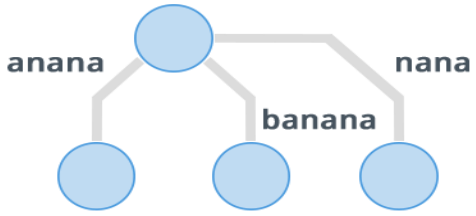
Figure: compressed trie for the given set of strings.

- As it might be clear from the images show above, in a compressed trie, edges that direct to a node having single child are combined together to form a single edge and their edge labels are concatenated.
- So this means that each internal node in a compressed trie has atleast two children. Also it has atmost **N leaves**, where **N i**s the number of strings inserted in the compressed trie.
- **Now both the facts:** Each internal node having atleast two children, and that there are N leaves, implies that there are atmost 2N−1 nodes in the trie.
- So the space complexity of a compressed trie is O(N) as compared to the $O(N^2)$ of a normal trie.

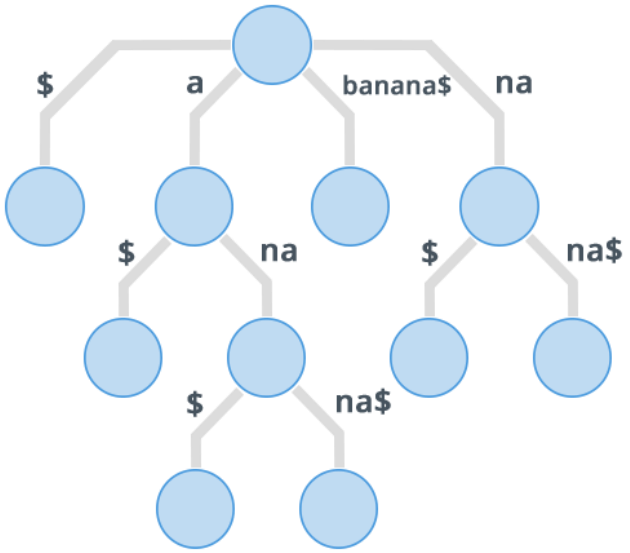
So that is one reason why to use compressed tries over normal tries.

Before going to construction of suffix trees, there is one more thing that should be understood, Implicit Suffix Tree. In Implicit suffix trees, there are atmost N leaves, while in normal one there should be exactly N leaves. The reason for atmost N leaves is one suffix being prefix of another suffix.

Following example will make it clear. Consider the string "banana"
Implicit Suffix Tree for the above string is shown in image below:



To avoid getting an Implicit Suffix Tree we append a special character that is not equal to any other character of the string. Suppose we append $ to the given string then, so the new string is "banana$". Now its suffix tree will be



Drawbacks of Compressed Tries (Patricia Tries)

1. Complex Insertion and Deletion

- **Problem:** Inserting or deleting keys is **more complex** due to:
  - Splitting edges when adding a partial match.
  - Merging nodes when deleting a key.
- This requires handling **substrings on edges**, not just characters, which adds logic.

2. Slower Worst-case Lookup (in practice)

- Although **theoretical lookup is O(k)** (length of key), comparing substrings instead of single characters can:
  - Introduce overhead due to `String.startsWith()`, `substring()`, etc.
  - Be **slower in practice** than standard Trie lookups.

3. Substring Matching is Not Directly Supported

- Compressed Tries are great for **prefix matching**, but:
  - **Substring search** (like in Suffix Trees) isn't natively supported unless all suffixes are inserted.
  - This leads to building a **Compressed Suffix Trie**, which requires even more memory and logic.

---

4. More Memory in Some Scenarios

- If many keys share long prefixes but then diverge quickly, substrings are stored **redundantly** on different edges.
- Although they save space over regular tries in general, they **don't beat suffix trees** in substring-heavy datasets.

---

5. Harder to Implement

- Compared to standard Tries, compressed tries:
  - Require more edge-case handling.
  - Are **not beginner-friendly** to code from scratch.
  - Debugging them can be tricky.

Now let's go to the construction of the suffix trees.

Suffix tree as mentioned previously is a compressed trie of all the suffixes of a given string, so the brute force approach will be to consider all the suffixes of the given string as separate strings and insert them in the trie one by one

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.

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.

Purpose of a Suffix Tree

A **suffix tree** is a compressed trie-like data structure used to **represent all suffixes of a given string**. It allows you to perform **fast string operations**.

Main Goals of a Suffix Tree

Operation	Time Complexity
Check if a substring exists	O(m)
Count occurrences of a substring	O(m)
Find longest repeated substring	O(n)

Operation	Time Complexity
Find longest common substring	O(n + m)
Pattern matching in large texts	O(m + k)

Where:

- n is the length of the text
- m is the length of the pattern
- k is the number of occurrences

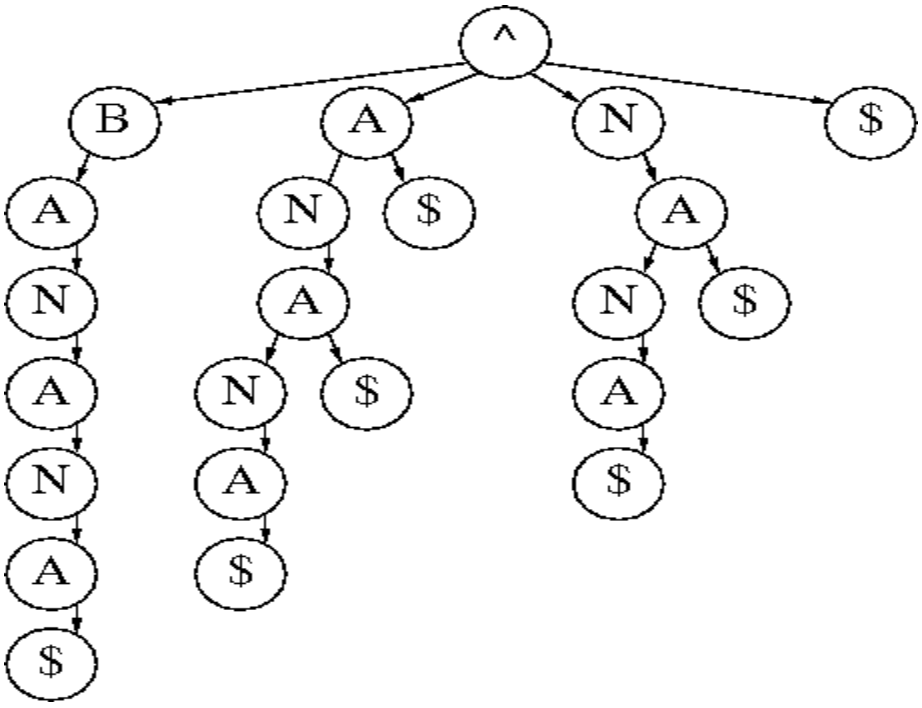
What does it store?

For a string "**banana"**, the suffix tree stores:

banana$
anana$
nana$
ana$
na$
a$

Each suffix becomes a **path in the tree**. The $ is added to make sure each suffix ends uniquely.

Trie Structure (Character-wise)



Key Use Cases

1. Fast Substring Search
Check if "ana" exists in "banana" in linear time (O(m)), better than brute force (O(n*m)).

2. Text Compression
Can help in detecting repeated patterns, enabling better compression.

3. Bioinformatics
Used in DNA sequence analysis for matching patterns in large genome sequences.

4. Longest Repeated Substring
Can find the longest repeated substring in linear time using the structure.

Example:

Let's say you have a string: "banana"

The suffix tree enables:

- **Search** for "ana": YES, found
- **Occurrences of "ana"**: 2 times
- **Longest repeated substring**: "ana" or "na"
- **Longest common substring** (between "banana" and "bandana"): "ana"

Java program to implement suffixtree:

```
import java.util.*;

class SuffixTrie
{
   static final int NUM_CHARS = 26;
   // SuffixTrie node
   static class SuffixTrieNode
   {
      SuffixTrieNode[] children = new SuffixTrieNode[NUM_CHARS];

      // isEndOfWord is true if the node represents end of a word
      boolean isEndOfWord;

      SuffixTrieNode()
      {
         isEndOfWord = false;
         for (int i = 0; i < NUM_CHARS; i++)
            children[i] = null;
      }
   };

   static SuffixTrieNode root;

   // If not present, inserts word into SuffixTrie
   // If the word is prefix of SuffixTrie node, just marks leaf node
```

```
static void insert(String word)
{
    System.out.println("word " + word);
    int level;
    int length = word.length();
    int index;

    SuffixTrieNode currentNode = root;

    for (level = 0; level < length; level++)
    {
        index = word.charAt(level) - 'a';
        if (currentNode.children[index] == null)
            currentNode.children[index] = new SuffixTrieNode();

        currentNode = currentNode.children[index];
    }

    // mark last node as leaf
    currentNode.isEndOfWord = true;
}

// To check if current node is leaf node or not
static boolean isLeafNode(SuffixTrieNode root)
{
    return root.isEndOfWord == true;
}

// print SuffixTrie
static void print(SuffixTrieNode 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);
        System.out.println("Enter any string to construct suffix tree");
        String word=sc.nextLine();
        root = new SuffixTrieNode();

        for (int i = 0; i < word.length(); i++)
            insert(word.substring(i));

        char[] str = new char[50];
        print(root, str, 0);
    }
}
```

Input:

Enter any string to construct suffix tree

Banana

Output:

word banana
word anana
word nana
word ana
word na
word a
a
ana
anana
banana
na
nana