Course:

Data structure & Algorithm

Synthesis Report

New Algorithms and Data Structures

Part 2

Guide teachers

Lê Ngọc Thành

Tạ Việt Phương

Phạm Trọng Nghĩa

Academic year

2020-2021

Thank you

Contents

Branch and Bound

Contents:

I. Introduction.

Branch and Bound is a special method for solving optimization problems[1].

The mechanism is that it will enumerate all the sets of candidate solutions by the state space search[2] procedure. A solution will include a rooted tree with the nodes that form a full solution for the problems.

The differences between Backtracking and Branch and Bound strategy are Branch and Bound do not limit us to any particular war of traversing the tree and it is only used for an optimization problem.

Branch and Bound is a alternative method for the optimization problems where the Greedy and Dynamic Programming fail to solve.

Some terms:

Active node: The node that is generated but whose children is not.

E-Node/Live Node: Is an active node whose childran are being explored.

Dead node: is the node that has been discarded that no need to be explored further.

The term "branch" refers to the method of generating all the children nodes of "E-Note" before considering with the "active nodes"

[1] An optimization problem is the problem of finding the best solution from all feasible solutions.

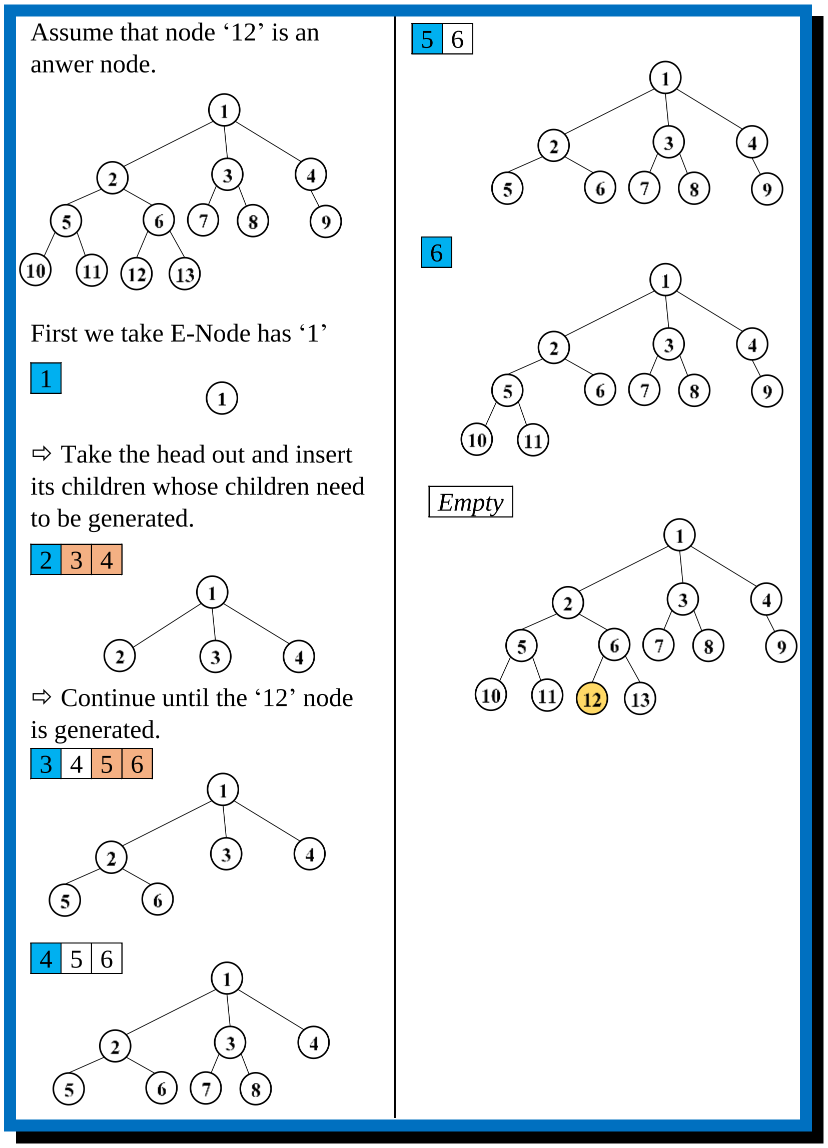
[2] State Space Search is the progress of finding a goal state with the desired property. With this procedure, problems are often modeled as a state space – a set of states that problems can be in which forms a graph where 2 states are connected if there is an operation that can be performed to traverse the first state into the second.

II. Methodologies.

There are 3 methods corresponding to the 3 way of exploring the branches:

- FIFO-BB:

- First-In-First-Out Branch and Bound is a **BFS** and children of E-Node are inserted in a **queue**.
- The operations of list of E-Nodes are the same as a **queue**:
    - *Enqueue()* Removes the head of the queue.
    - *Dequeue()* Adds the node to the tail of the queue.
- Example



Assume that node ‘12’ is an anwer node.

First we take E-Node has ‘1’

⇨ Take the head out and insert its children whose children need to be generated.

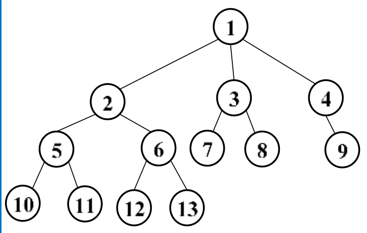
⇨ Continue until the ‘12’ node is generated.

- LIFO-BB:
  - Last-In-First-Out Branch and Bound is a **DFS** and children of E-Node are inserted in a **stack**.
  - The operations of list of E-Nodes are the same as a **stack**:

  o *Pop()* Removes the top of the stack.

  O *Push()* Adds the node to the top of the stack.
  - Example:

Assume that node ‘12’ is an anwer node.

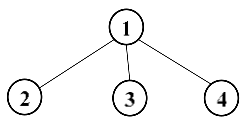


First we take E-Node has ‘1’

1

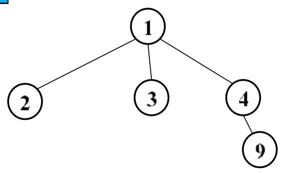
⇨ Take the head out and insert its children.

2 3 4

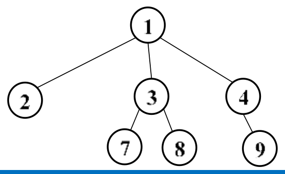


⇨ Continue until the ‘12’ node is generated.

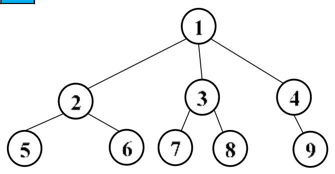
2 3



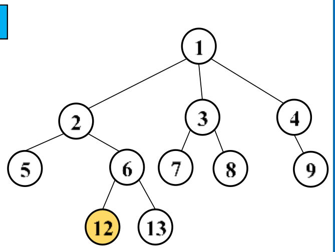
2



5 6



5



- *Notice that*: LIFO-BB will generate nodes tends to lean to the right and in contrast, FIFO-BB tends to lean to the left.

- Least Cost-BB(LC-BB)

- The Least-Cost Branch and Bound is considered the most "intelligent" as it selects the next node based on a Heuristic Cost Function[3]. It picks the next E-Node is the active node with the least cost and avoid all the others.
- In order to identify the next E-Node to spread the branches for more candidate solutions, it will be checked with the upper and lower estimated bound to make sure the solution is better than the current best one.

- To calculate the cost of the list of E-Node (or bounds):

$$\hat{C}(x) = f(x) + \hat{g}(x)$$

$\hat{C}$(x) – It is the estimated minimum cost to reach the good node.

F(x) – The total number of cost from the initial E-Node.

$\hat{G}$(x) – The optimal total number of cost of nodes that are not explored from the rest of the full solution(a branch).

- You can also apply the same procedure for finding maximum optimization problems by consider the cost is negative.

- Example:

[3] A heuristic function, is a function that calculates an approximate cost to a problem (or ranks alternatives).

8-puzzle problem:
You ask to transform the initial form of the puzzle into the specific given form with the least number of moves as much as possible.

Initial State

1	2	3
5	6	
7	8	4

$\longrightarrow$

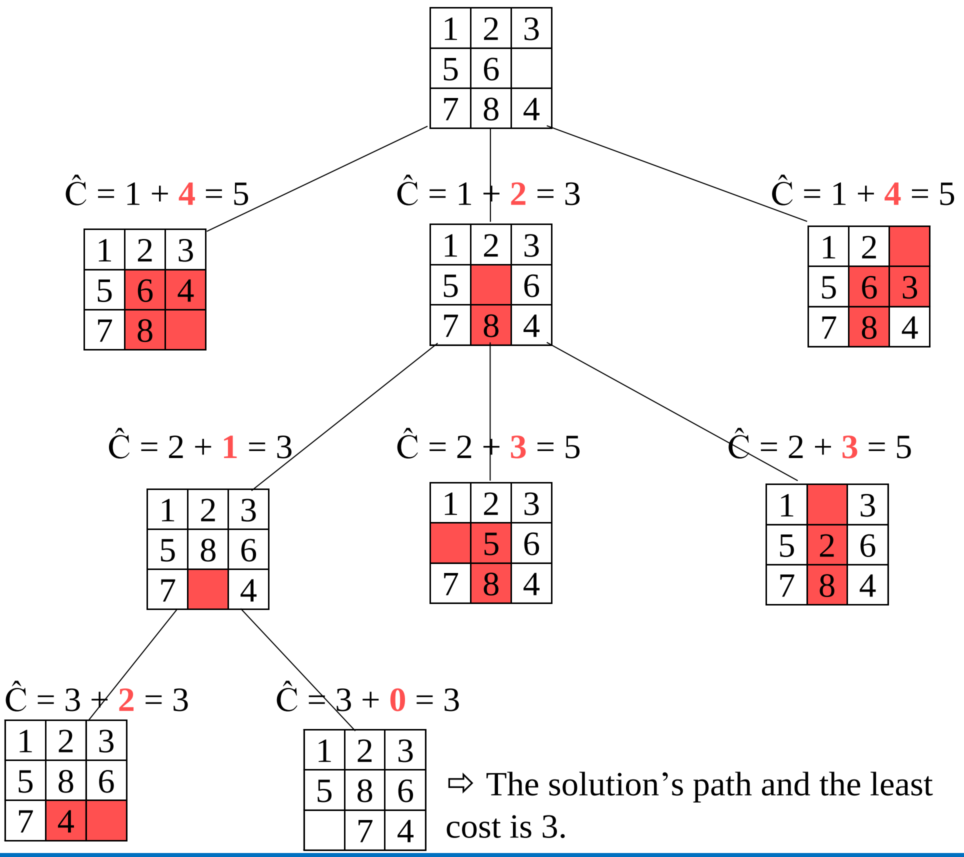
Final State

1	2	3
5	8	6
	7	4

Now, we should idenity the cost function and factors of this problem.

$$\hat{C}(x) = f(x) + \hat{g}(x)$$

f(x) = Number of moves so far.
$\hat{g}(x)$ = Number of non-blank misplaced titles from the Final State.



⇨ The solution's path and the least cost is 3.

- So we have comprehend that LIFO and FIFO Branch and Bound both have the selection node for the next E-Node is rigid and blind. That is the reason why these two are not frequently used as often as LC-BB because Least-Cost method gives the faster result.

III. Application

* I/O Knapsack problem.

- **Problem:**

Given N items with weights W[0..n-1], values V[0..n-1] and a knapsack with capacity M, select the items such that:

- The sum of weights taken into the knapsack is less than or equal to M.
- The sum of values (or benefits) of the items in the knapsack is maximum among all the possible combinations.
- Example: N = 4, M = 15, V[]= {10, 10, 12, 18}, W[]= {2, 4, 6, 9}

- **Solution:**

- When talking about Branch and Bound algorithm, it is impossible not to mention the **Knapsack Problem** because the Least-Cost BB method will be a handy tool to solve this problem in one of the most efficient way.

- Let's define that a combination of a solution with the **1/0** is represent the value is included in the sack or not:

1 – the value is included and **0** – the value is not included.

Ex: S = {1,0,1,0} => $\mathbf{x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 0}$

- So basically, the problem want you to find the combination of items to put in the knapsack that give as much as possible values:

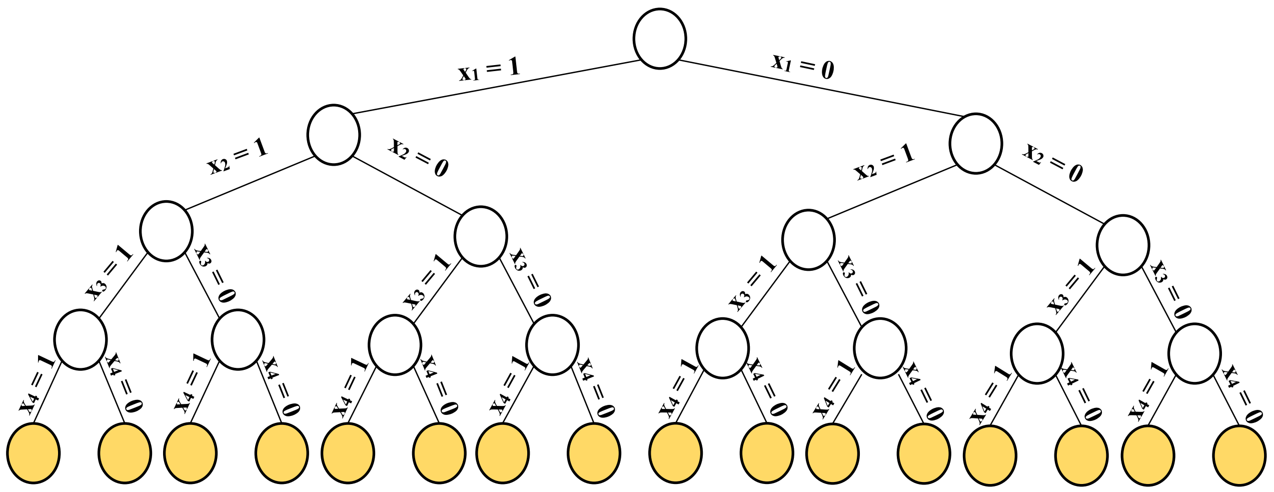
+ $\sum_{i=1}^{n} V_i x_i$ has to be **maximized.**

+ But has the **constraint**: $\sum_{i=1}^{n} W_i x_i \leq M$

+ Brute-Force Solution:

- If you attempt to solve the problem by enumbering all the possible solutions and choose the one that has the most values.
- Because we have to make N choices, so we will have the set of 2^n number of possible solutions.

- So with the example above, which is N = 4 the state space tree will look like this:



⇨ There are $2^4 = 16$ solutions just only for 4 items and keep in mind that the complexity will increase exponentially.

+ Apply Least-Cost BB:

- So the idea for the implementation is that we will evaluate the **upper bound(U)** for each node while generating them. In another word, each time we generate a node, the upper bound will be evaluated following the properties of the node.
- Each node will represent the circumstance that an item is included in the sack or not.

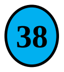
- To find the upper bound, we must use the concept in the Fractional Knapsack problem. Which will use the method like the Greedy method where we choose the items that have the best **value by weight**, so everytime we make a choice then we choose a combination that give us the largest value.

- Example: N = 4, M = 15, V[]= {10, 10, 12, 18}, W[]= {2, 4, 6, 9}

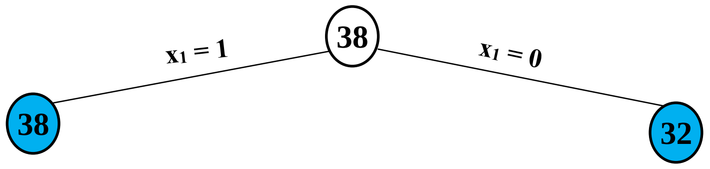
For easy to evaluate the upper bound, we first consider **value by weight** of each items so we can observe which item gives us the most value: **{V / W} = {5, 2.5, 2, 2}**

⇨ The initial root node's upper bound will be U = 10 + 10 + 12 + 18/9***3** = **38**

- The reason that the U contain a fractional is because when we attempt to utilize the capacitiy of the sack to give us the maxium value, the total of weight will be: w = 2 + 4 + 6 + **3** = **16** <= **M** .The remain **3** weight left of the sack will be utilized by taking **3** time of the most value by weight currently which is **18/9**.

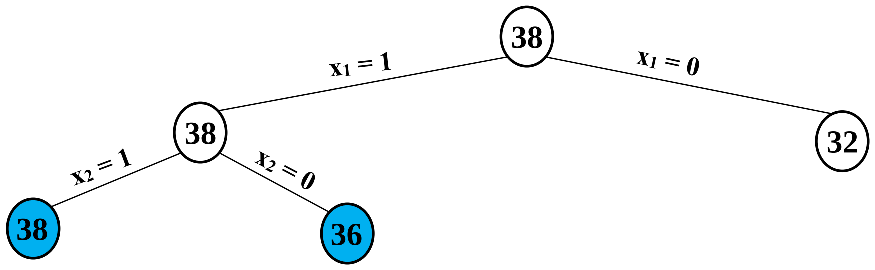


⇨ From here, the most promising candidate node is **38**, so we will continue to generate more nodes from it.



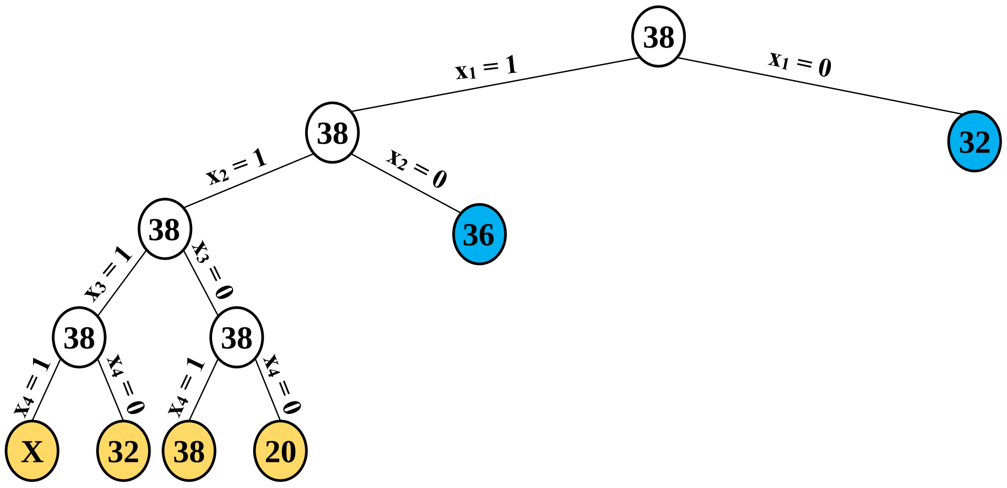
⇨ The upper bound in case x_1 is included is still U = **38** and not included is U = 10 + 12 + 18/9***5** = **32**(w = 4 + 6 + **5** = **15**)

⇨ From here, the most promising candidate node is **38**, so we will continue to generate more nodes from it.



- ⇨ The upper bound in case x_2 is included is still U = 38 and not included is U = 10 + 12 + 18/9*7 = 36 (w = 2 + 6 + 7 = **15**)
- ⇨ From here, the most promising candidate node is 38, so we will continue to generate more nodes from it.

- ⇨ Repeat the same action until a full solution are form properly:



- ⇨ The best solution so far will give us 38 total of values
- ⇨ So the remains candidate nodes's upper bound 36 and 32 is lesser than our current solution so we do not need to generate more cases from them to search more optimal solutions.

So with that, the best optimal solution will be **S = {1, 1, 0, 1}** and gives **38** total of values.

As you can see in the Branch and Bound method when compare it with the **Brute-Force Solution**, it only need to search for 4 solutions instead of 16 to find the maximize solution, it remove some branches of searching. This help us reach the niche solution for the problem faster.

- **Implement:**

- First, we need to define the struct *Node* for a node in the state space tree:

```
struct Node
{
	int x, values, weights, UB;
};
```

- Sort all the items in descending order by their value per weight in order to evaluate the upper bound of a node later:

```
void sortValuebyWeight(int* V, int* W, int N)
{
	int v, w, j;
	for (int i = 1; i < N; i++)
	{
		v = V[i];
		w = W[i];
		j = i - 1;
		while (j >= 0 && (double)V[j] / W[j] < (double)v / w)
		{
			V[j + 1] = V[j];
			W[j + 1] = W[j];
			j--;
		}

		V[j + 1] = v;
		W[j + 1] = w;
	}
}
```

Apply a simple *Insertion Sort* or any sorting algorithm will also work well.

- Next, define a function to evaluate the upper bound values of the items:

```
int upperbound(Node& p, int* V, int* W, int N, int M)
{
	int UB = p.values;
	int w = p.weights;

	int i = p.x + 1;
	while (i < N && w + W[i] < M) {
		UB += V[i];
		w += W[i];
		i++;
	}

	if (i < N)
		UB += (M - w) * V[i] / W[i];

	p.UB = UB;
	return UB;
}
```

This function mainly uses Greedy solution to find an upper bound on maximum value.

- Set up some operations of the priority queue that we are going to use for storing some nodes to search for some further solutions:

```
void enqueue(Node* queue, int N, Node p)
{
	if (p.UB < queue[N - 1].UB || N - 1 == -1) {
		queue[N] = p;
		return;
	}

	queue[N - 1] = queue[N - 2];
	enqueue(queue, N - 1, p);
}
```

Put a node into the appropriate position in the priority queue by the upper bound value of the node.

```
Node dequeue(Node* queue, int N)
{
	if (N == 1)
		return queue[0];

	Node p = dequeue(queue, N - 1);
	queue[N - 2] = queue[N - 1];
	queue[N - 1].UB = -1;
	return p;
}
```

Pop out the node from the head of the queue.

```
bool isEmpty(Node* queue)
{
	return queue[0].UB == -1 ? true : false;
}
```

Check whether the queue is empty or not.

```
Node* initQueue(int N)
{
	Node* queue = new Node[N];
	for (int i = 0; i < N; i++)
		queue[i].UB = -1;

	return queue;
}
```

Initialize the queue.

- And finally, the function will find the optimal solution for the problem:

```
int solveKapsack(int* V, int* W, int N, int M)
{
	sortValuebyWeight(V, W, N);
	Node* queue = initQueue(N);

	Node p_pre{ -1, 0, 0 };
	p_pre.UB = upperbound(p_pre, V, W, N, M);
	enqueue(queue, N, p_pre);

	int maxValue = 0;
	Node p{ 0 };
	while (!isEmpty(queue))
	{
		p_pre = dequeue(queue, N);
		if (p_pre.x == N - 1||p_pre.UB < maxValue)
			continue;

		p.x = p_pre.x + 1;

		//x is included in the sack
		p.values = p_pre.values + V[p.x];
		p.weights = p_pre.weights + W[p.x];
		p.UB = upperbound(p, V, W, N, M);
		//if the node larger than the current values,
		//put it into the queue for further searches
		if (p.UB >= maxValue && p.weights <= M)
			enqueue(queue, N, p);

		if (p.values >= maxValue && p.weights <= M)
			maxValue = p.values;

		//x is not included in the sack
		p.values = p_pre.values;
		p.weights = p_pre.weights;
		p.UB = upperbound(p, V, W, N, M);

		if (p.UB >= maxValue && p.weights <= M)
			enqueue(queue, N, p);
	}

	delete[]queue;
	return maxValue;
}
```

IV. References

https://en.wikipedia.org/wiki/Branch_and_bound https://en.wikipedia.org/wiki/State_space_search

https://academyera.com/branch-and-bound

https://www.sciencedirect.com/science/article/pii/S1572528616000062

Knapsack problem:

https://www.geeksforgeeks.org/0-1-knapsack-using-least-count-branch-and-bound/

https://www.geeksforgeeks.org/implementation-of-0-1-knapsack-using-branch-and-bound/

https://www.youtube.com/watch?v=R6BQ3gBrfjQ

https://www.youtube.com/watch?v=yV1d-b_NeK8&t=5s

https://www.youtube.com/watch?v=4D_JZxFg9Mk

Interpolation Search

I. Introduce:

Example: For example, we have a problem that does not include N integer elements (characters, ...) that have been sorted in ascending or descending order. We are asked to find an element X in it.

⇨ Solution: With the above problem, we have many ways to do it such as using Linear Search, Binary Search, ... But the above ways will lead to stack overflow with N=1000000000000. But with the Interpolation Search algorithm, it will help us deal with large N (of course it also has a certain limit) because it has a complexity of $O(\log_2(\log_2 N))$. Example is N=1000000, Binary Search has O(log(1000000))=20, Linear Search has O(1000000) and Interpolation Search has $O(\log_2(\log_2 1000000))$=4,3 => The Interpolation Search algorithm is very efficient when N is large.

II. Definition of Interpolation Search algorithm:

- Interpolation Search is an improved variant of Binary Search. For this search algorithm to work correctly, the data set must be sorted. For example, in the case of the phone book, if we want to find the phone number of Craysis for example. In this case, Linear Search and also Binary Search can be slow to perform the search, since we can directly jump to the part of memory space whose name starts with C is stored.

III. Locate in Interpolation Search:

- Starting from the formula to find the middle part of the set according to Binary Search, we have:

$$\boxed{pos = left + (right - left)/2 = (left + right)/2}$$

- We will improve by replacing the value 1/2 with the following expression:

(X – T[left])/(T[right] – T[left]). So:

$$\boxed{pos = left + (X - T[left]) * (right – left) / (T[right] – T[left])}$$

Which:

+ X – T[left] : Number of elements from element X to the left element.

+ right – left : Number of elements in position from left to right.

+ (X- T[left]) / (T[right] – T[left]): In which region is the percentage of element X located.

IV. Implemention:

```
bool Interpolation_Sort(int arr[], int n,int x)
{
     left <- 0;
     right <- n - 1;
    while (left <= right and x >= arr[left] and x <= 
arr[right])
    {
        val1 <- (x - arr[left]) / (arr[right] - arr[left]);
        val2 <- (right - left);
        pos <- left + val1 * val2;

        if (arr[pos] == x)
            return pos;

        if (arr[pos] < x)
            left <- pos + 1;
        else
            right <- pos - 1;
    }
    return -1;
}
```

V. Illustration:

We have an integer array of N elements, sorted in ascending order. Find the element X present in the array.

N=6

X=5

1 5 8 9 10 15

Illustration

* Execution 1:

Left = 0

Right = 5

Search = 0 + $\frac{4}{14}$*5 = 1

a[1]= 5 = X => Stop while loop.

Trie Tree

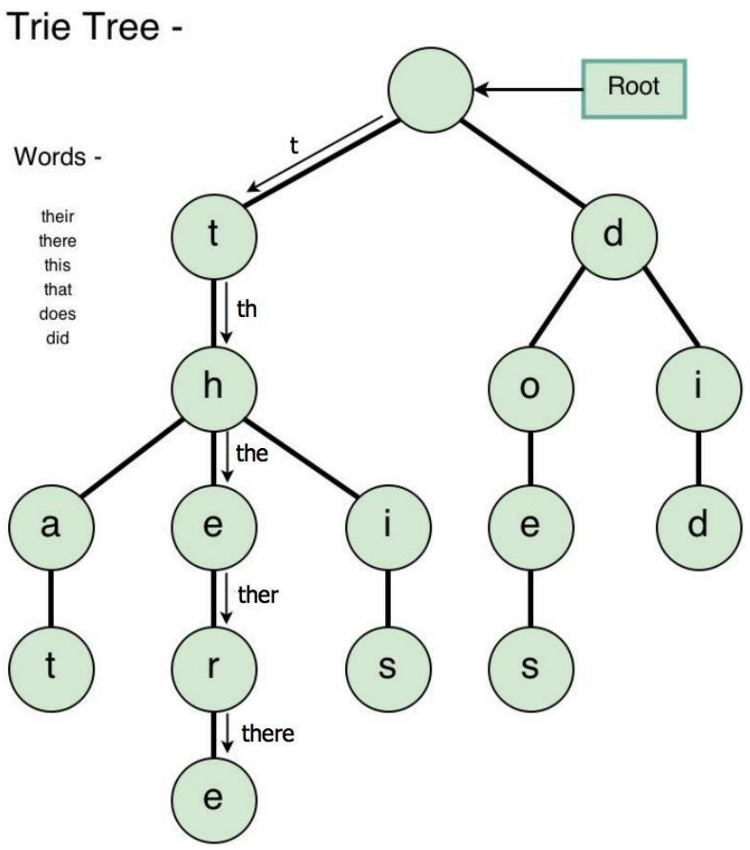
Content:

- **What is Trie tree?**
- **How can we implement the Trie tree?**
    - **Insert**
    - **Search**
    - **Delete**

I. Introduction.

A trie, also called digital tree or prefix tree, is a type of search tree, is an efficient information retrieval data structure.

It used for locating specific keys from within a set. These keys are most often strings.



Using Trie, we can search the key in **O(M)**, where M is maximum string length.

- Every node of Trie consists of mutiltiple branches.
- Each branch represents a possible character of keys.
- We need to mark the last node of every key as end of word node. A Trie node field *isEndOfWord* is used to mark it.

We can represent Trie nodes as following:

```
#define ALPHABET_SIZE 26
struct TrieNode {
     TrieNode* children[ALPHABET_SIZE];
     bool isEndOfWord;
};
```

II. Implement Trie.

1. Inserting

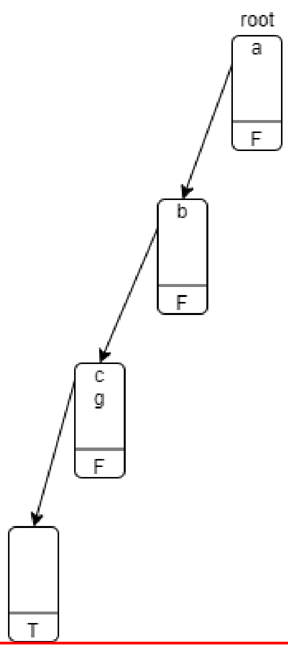
Inserting a key into Trie is very simple.

Every character of the input key is inserted as an individual Trie node. Note that the *children* is an array of pointer to next level trie nodes.

The key character acts as an index into the array children. If the input key is new or an extension of the existing key, we need to construct non-existing nodes of the key, and mark end of the word for the last node.

If the input key is prefix of the existing key in Trie, we simply mark the last node of the key as the end of a word.

The key length determines Trie depth.



The last node is marked True (T)

For example, lets insert the

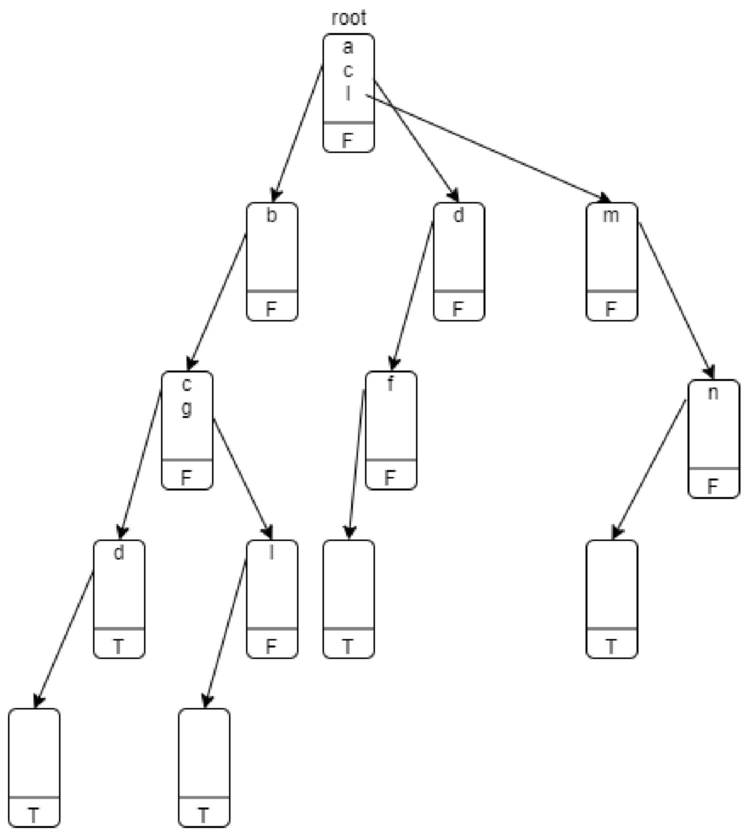
key ‘*abc’* in the structure

Initialize root with NULL

Now we insert one by one character into the Trie, each node contains a character of the word

The last node is empty and is marked the end of the word

Similarly keep inserting ‘abgl’, ‘cdf’, ‘abcd’, ‘lmn’. Note that we always start from root node



After we inserted

‘abgl’

‘cdf’

‘abcd’

‘lmn’

2. Searching

Searching for a key is similar to insert operation, however, we only compare the characters and move down.

The search can terminate due to the end of a string or lack of key in the Trie.

If the *isEndOfWord* field of the last node is true, then the key exists in the Trie.

In the second case, the search terminates without examining all the characters of the key, since the key is not present in the Trie.

The cost for insert and search are **O(key_length)**. The memory requirement of Trie is **O(Alphabet_size * key_length * N)** where N is number of keys in Trie.

This is a program implement insert and search operations on Trie

```
#define ALPHABET_SIZE 26
struct TrieNode {
	TrieNode* children[ALPHABET_SIZE];
	bool isEndOfWord;

	TrieNode() {
		isEndOfWord = false;
		for (int i = 0; i < ALPHABET_SIZE; i++) {
			children[i] = NULL;
		}
	}
};
void insert(TrieNode* root, string key) {
	TrieNode* pCrawl = root;
	for (int i = 0; i < key.length(); i++) {
		int index = key[i] - 'a';
		if (!pCrawl->children[index]) {
			pCrawl->children[index] = new TrieNode();
		}
		pCrawl = pCrawl->children[index];
	}
	// mark last node as leaf
	pCrawl->isEndOfWord = true;
}
bool search(TrieNode* root, string key) {
	TrieNode* pCrawl = root;
	for (int i = 0; i < key.length(); i++) {
		int index = key[i] - 'a';
		if (!pCrawl->children[index]) {
			return false;
		}
		pCrawl = pCrawl->children[index];
	}
	return (pCrawl->isEndOfWord);
}
```

3. Delete

Algorithm how to delete a node from trie, we will delete the key in bottom up maner using recursion.

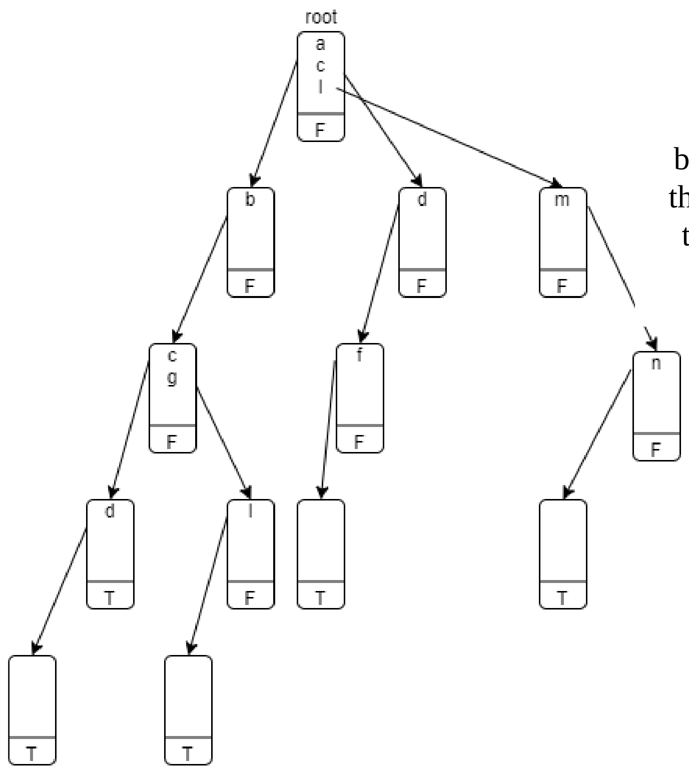
1. Key may not be there in Trie, not modify Trie

2. Key present as unique key (mean that no part of key contains another key (prefix), nor the key itself is prefix of another key in Trie). Delete all nodes.

3. Key is prefix of another long key in Trie. Unmark the leaf node.

4. Key present in Trie, having at least one other key as prefix key. Delete nodes from end of key until first leaf node of longest prefix key.

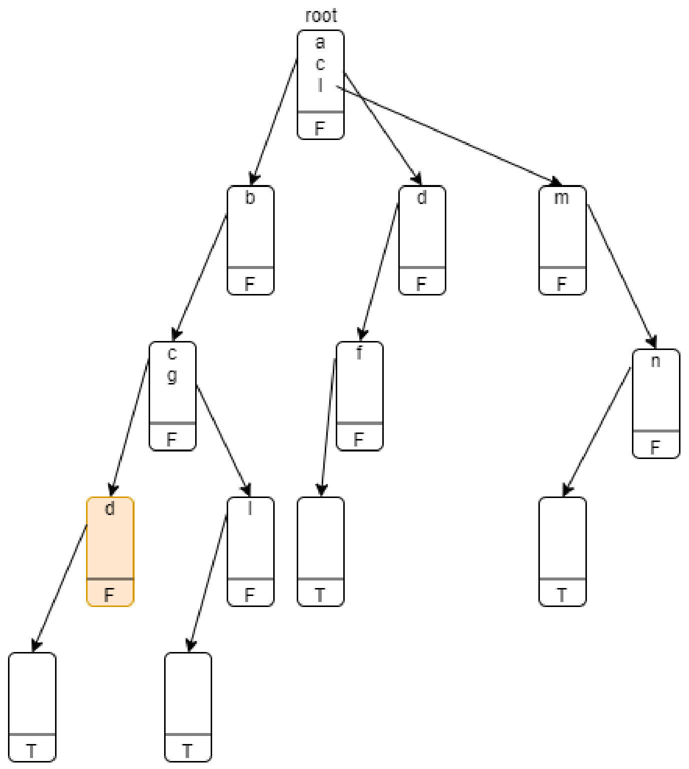
Delete ‘abc’ in the Trie



We do the same thing in Searching,

but when we reach the node that contains the last character of the word, we unmark the leaf node (*isEndOfWord*) of the last character node

After that, we recur bottom up to the root node to check the cases maybe occur in the Trie (algorithm above)



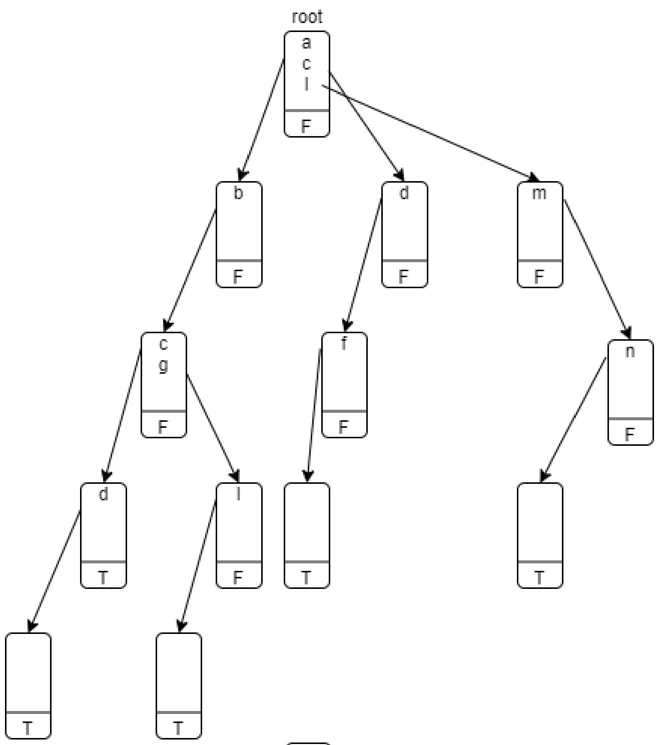
This is the result after we delete ‘abc’ in the Trie

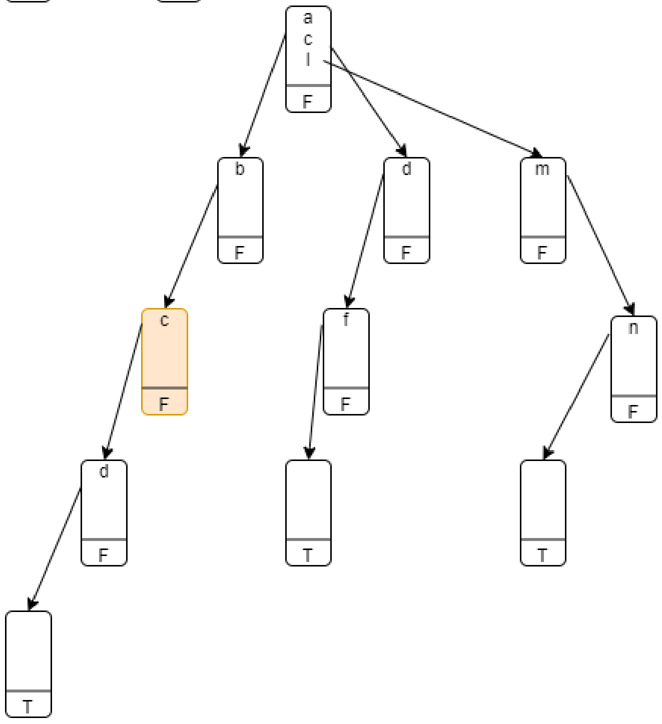
The node contains ‘d’ is unmarked T because it’s a leaf node of ‘c’ node (last character)

We don’t allow to delete ‘a’, ‘b’, and ‘c’ nodes because these nodes are prefixes of ‘abcd’

Go on to delete ‘abgl’ in the Trie

Similar to delete ‘abc’, we going to reach the last node that is the end of the word





This is the result after we delete ‘abgl’ in the Trie

We deleted ‘l’ node straightforward because it is not prefix of another key in Trie.

Inside node ‘c’, we deleted children ‘g’

We shouldn’t to delete ‘a’ and ‘b’ nodes because these nodes are prefixes of another word

This is a program implement delete operation on Trie

```
// Returns true if root has no children, else false
bool isEmpty(TrieNode* root){
	for (int i = 0; i < ALPHABET_SIZE; i++)
		if (root->children[i])
			return false;
	return true;
}
TrieNode* remove(TrieNode* root, string key, int depth = 0) {
	// If tree is empty
	if (!root) {
		return NULL;
	}
	// If last character of key is being processed
	if (depth == key.size()) {
		// 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)) {
			delete root;
			root = NULL;
		}
		return root;
	}
	// If not last character, recur for the child
	// obtaind using ASCII value
	int index = key[depth] - 'a';
	root->children[index] = remove(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) {
		delete root;
		root = NULL;
	}
	return root;
}
```

The time complexity of the deletion operation is **O(n)** where n is the key length.

Segment Tree

I) Definition of Segment Tree

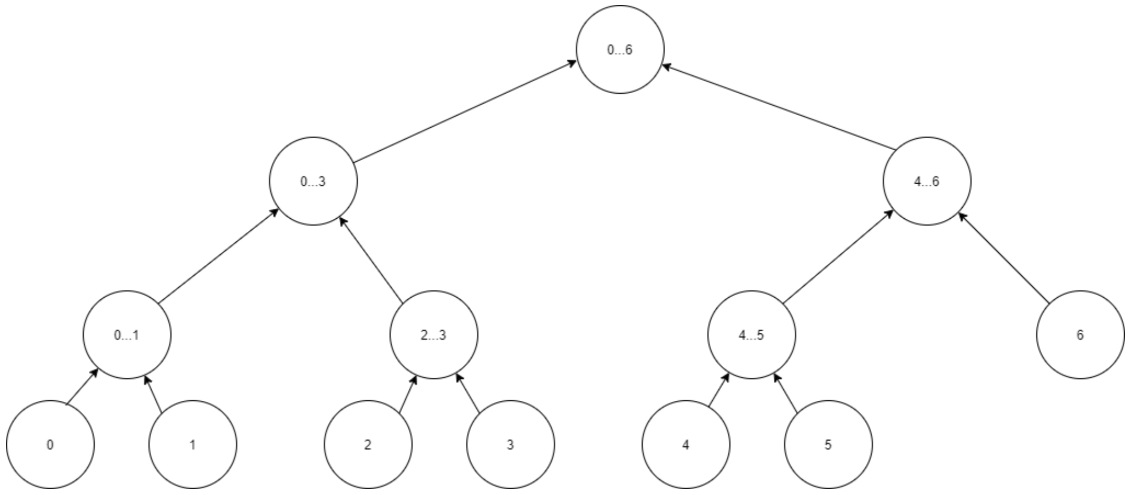
Imagine that you have an array A: A[0], A[1],…, A[n-1] (n is the number elements of the array). You want to do these operations with the array: find sum of the elements which from index l to index r in the array (A[l] + A[l+1] + … + A[r-1] + A[r], $0 \le l \le r \le n-1$), find maximum value, minimum value of elements in the range [l...r] of the array, update value in the rang [l…r] (update one elements, update all elements such as plus a value x to all elements in the range). With all these operations, the time complexity is $O(\log(n))$.
There is one simple solution that you can run some loops with the whole arrays to do these operations. But it's not a effective way because the time complexity is $O(n)$.
So, there is another solution that is more effective than the normal way which uses some loops though the code-implement is more complicated. That is **Segment Tree**. A Segment Tree is a tree data structure. Specifically, this tree's nodes has 2 cases. First, the nodes is the leaf of the tree. Second, each node has 2 children. There is no case that a node has **one** child.
This kind of tree has many applications in competitive coding challenge by dealing with sequence of numbers.

II) Implement Segment Tree

The array **A** has **n** elements. **l** is the left index, **r** is the right index in the array, **l** and **r** form a segment in the array, $0 \le l \le r \le n-1$.

The first node (root) of Segment Tree represents for information of all elements in the array (range: [1…n]). The leaf nodes represent for elements of the input array (A[0], A[1], …, A[n-1]). Each internal node represents for some merging of the leaf nodes. If a node represents for information in the range [l…r], then its children will represent for information in the range [l…(l+r)/2] and [(l+r)/2 + 1 … r].

Example with array A has 7 elements {0,1,2,3,4,5,6}, Segment tree will be implemented:



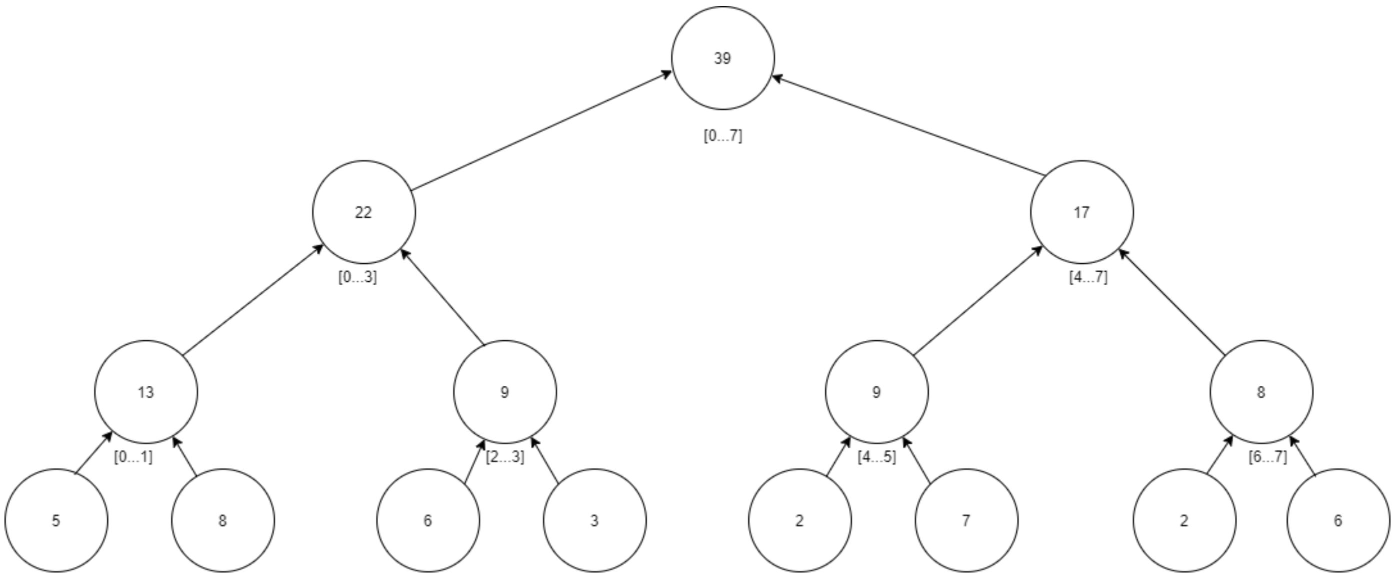
In coding, we can easy implement Segment Tree by using 1D array. The first element A[0] represents for root node. The i element will have 2 children is i*2 (left child) and i*2+1 (right child). The memory used for Segment Tree will be no more than 4*n elements.

III) Use Segment Tree to deal with problems

Example 1: Find sum of the elements in the array A which in range [l…r]

A = {5,8,6,3,2,7,2,6}, l = 2; r = 7.

Segment Tree of this problem:



The leaf nodes are the elements in the array A (5,8,6,3,2,7,2,6).
The upper nodes are the sum values of the leaf node: 13 = 5 + 6, 9 = 6 + 3, 9 = 2 + 7, 8 = 2 + 6, the next upper nodes and the root node are the same. So, the root node is the sum of all elements in the array.
Now, we need to calculate the sum: A[2] + A[3] + … + A[7]
We can use a recursive function to solve this problem, pseudo-code:

```
int getSum(node, l, r)
    if node information totally in the range [l…r]
        return node value
    else if node information totally out the range [l…r]
        return 0
    else // node information has a small part in the range [l…r]
        return getSum(node->left,l,r) + getSum(node->right,l,r)
```

These steps are explaination of the code above:

- Start function at root node ([0…7])

Because [2…7] range has a small part in [0…7] range, continue to check node->left ([0…3]) and node->right ([4…7])

  - Call function at node [0…3]

Because [2…7] has a small part in [0…3] range, continue to check node->left ([0…1]) and node->right([2…3])

    - Call function at node [0…1]

Because [2…7] out of range [0…1] so return 0.

    - Call function at node [2…3]

Because [2…7] has the range [2…3] so return its value: 9

Now return result of function at node [0…3]: 0 + 9 = 9.

  - Call function at node [4…7]

Because [2…7] has the range [4...7] so return its value: 17

Now return of function at node [0…7]: 9 + 17 = 26.

- Now the function ends, the final result is 26. (6+3+2+7+2+6 = 26)

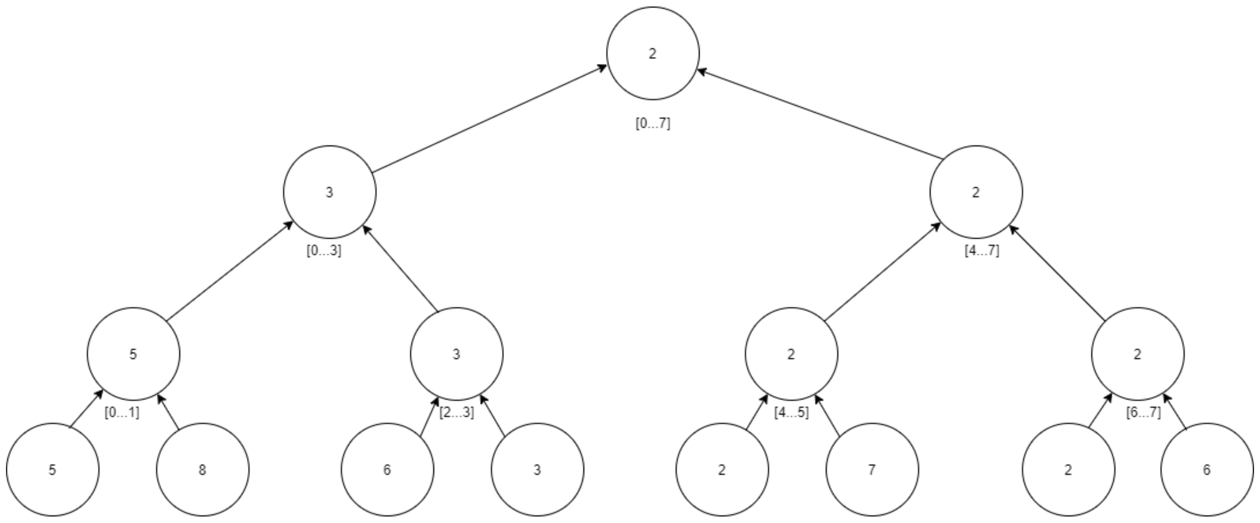
C/C++ code create Segment Tree in this problem:

```
int n;
int tree[4*n];
void createTree(int a[], int v, int l, int r)
{
        if (l == r) tree[v] = a[l]; //stop when at leaf node
        else
        {
                int mid = (l+r)/2;
                createTree(a,v*2+1,l,mid); //left child node
                createTree(a,v*2+2,mid+1,r); //right child node
                tree[v] = tree[v*2+1] + tree[v*2+2]; //node v equals sum of left child and right child
        }
}
```

C/C++ code getSum function:

```
int getSum(int tree[], int v, int leftRange, int rightRange, int l, int r)
{
        if (leftRange <= l && rightRange >= r) return tree[v]; //inside range
        if (l <= leftRange || r >= rightRange) return 0; // outside range
        int mid = (l+r) / 2;
        return getSum(tree, v*2 + 1, leftRange, rightRange, l, mid) + getSum(tree, v*2 + 2, leftRange,
        rightRange, mid, r);
}
```

Example 2: Find minimum value through the elements in the array A which in range [l…r]
A = {5,8,6,3,2,7,2,6}, l = 3; r = 7.



The upper nodes is the smaller node of its children. (5 < 8 => 5, 3 < 6 => 3, ..). Now we need to find the smallest element in range A[3], A[4], .. A[7].

We can use a recursive function to solve this problem, pseudo-code:

```
int getMin(node, l, r)
        if node information totally in the range [l…r]
                return node value
        else if node information totally out the range [l…r]
                return NOTHING
        else // node information has a small part in the range [l…r]
                return Min(getMin(node->left,l,r),  getMin(node->right,l,r))
```

These steps are explaination of the code above:

- Start function at root node ([0…7])

Because [3…7] range has a small part in [0…7] range, continue to check node->left ([0…3]) and node->right ([4…7])

- Call function at node [0…3]

Because [3…7] has a small part in [0…3] range, continue to check node->left ([0…1]) and node->right([2…3])

- Call function at node [0…1]

Because [3…7] out of range [0…1] so return NOTHING.

- Call function at node [2…3]

Because [3…7] has a part in the range [2…3] so continue to check node->left ([2]) and node->right([3])
[2] doesn't in [3…7] so return NOTHING.
[3] is in [3…7] so return its value: 3.
Now return of function at node [0…3] is 3.

- Call function at node [4…7]

Because [3…7] has the range [4...7] so return its value: 2

Now return of function at node [0…7]: 2 < 3 => Min = 2

- Now the function ends, the final result is 2.

C/C++ code create Segment Tree in this problem (Min(a,b) return smaller value between a and b)

```
int n;
int tree[4*n];
void createTree(int a[], int v, int l, int r)
{
        if (l == r) tree[v] = a[l]; //stop when at leaf node
        else
        {
                int mid = (l+r)/2;
                createTree(a,v*2+1,l,mid); //left child node
                createTree(a,v*2+2,mid+1,r); //right child node
                tree[v] = Min(tree[v*2+1],  tree[v*2+2]); //node v equals min value between left-right
                children
        }
}
```

C/C++ code getSum function:

```
int getMin(int tree[], int v, int leftRange, int rightRange, int l, int r)
{
        if (leftRange <= l && rightRange >= r) return tree[v]; //inside range
        if (l <= leftRange || r >= rightRange) return INT_MAX; // outside range, INT_MAX in C++ is the
        largest interger number
        int mid = (l+r) / 2;
        return Min(getMin(tree, v*2 + 1, leftRange, rightRange, l, mid), getMin(tree, v*2 + 2, leftRange,
        rightRange, mid, r));
}
```

Update a element's value by index (pos) in the array A.

Just like other operations with Segment Tree. Updating function is taken by recursive function. The input of the function is an index (pos) and a new value of that element. With Segment Tree, we must go from the root node and update all node that include the index (pos) node.

C/C++ code update an element:

```
void update(int tree[], int v, int l, int r, int pos, int newValue)
{
        If (l == r) tree[v] = newValue;
        Else
        {
                int mid = (l+r)/2;
                if (pos < mid) update(tree,v*2+1,l, mid, pos, newValue);
                else update(tree,v*2+2,mid+1, r, pos, newValue);
                tree[v] =  OPERATION DEALING WITH EACH PROBLEM
        }
}
```

Red line: Depend on the operation that the problem deal with.

Example problem in ex1: Calculate the sum tree[v] = tree[v*2+1] + tree[v*2+2] // Sum of two children

Example problem in ex2: Find minimun value

tree[v] = Min(tree[v*2+1], tree[v*2+2])

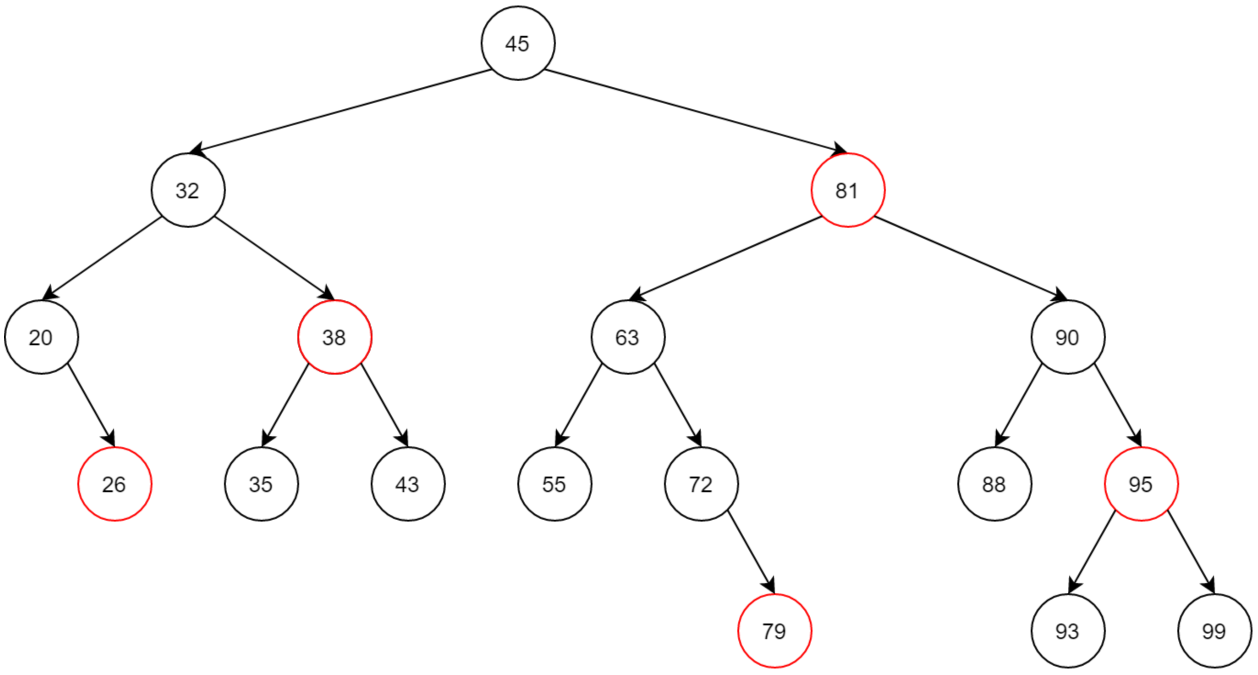
// Find smaller value between two children.

AA tree

Contents:

I. Introduction:

AA tree is a balanced binary search tree, and is a variation of the red-black tree (RB tree). The different is in AA tree, the red nodes can only be added as a right child. Instead of using the color like in RB tree, AA tree actually use the concept of levels for balancing. AA trees are named after their inventor, Arne Andersson. Example for an AA tree:



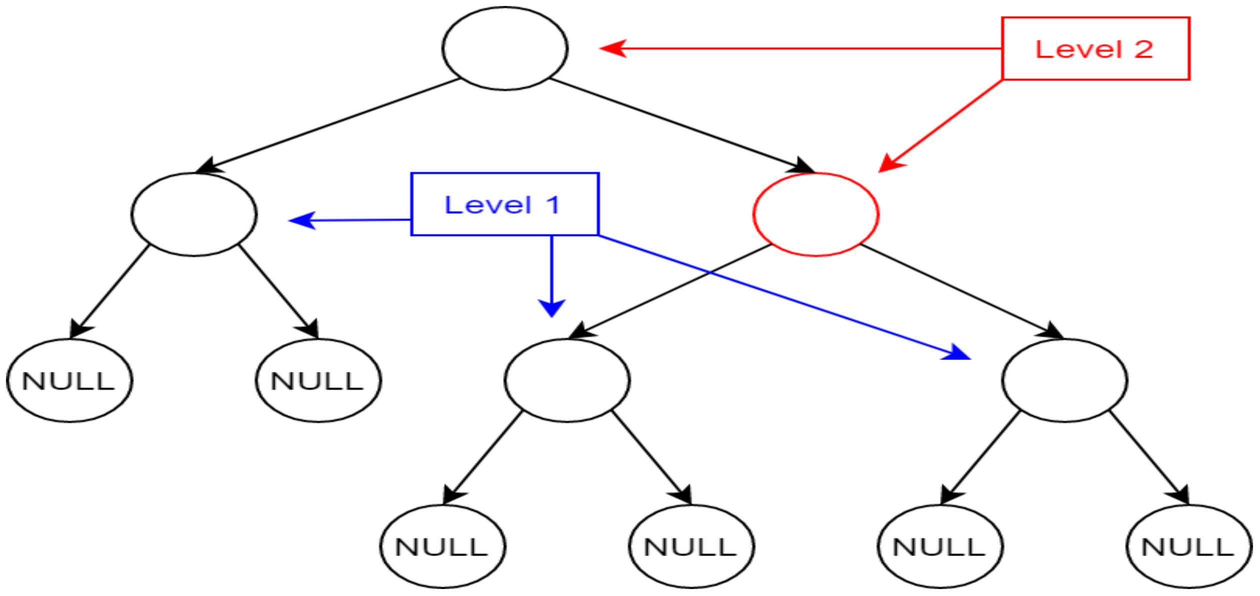
To learn how an AA tree work, we need to know some new definition first.

Level of a node :

- Is the number of left links from that node to a NULL node.

- Level of every NULL node is 0.

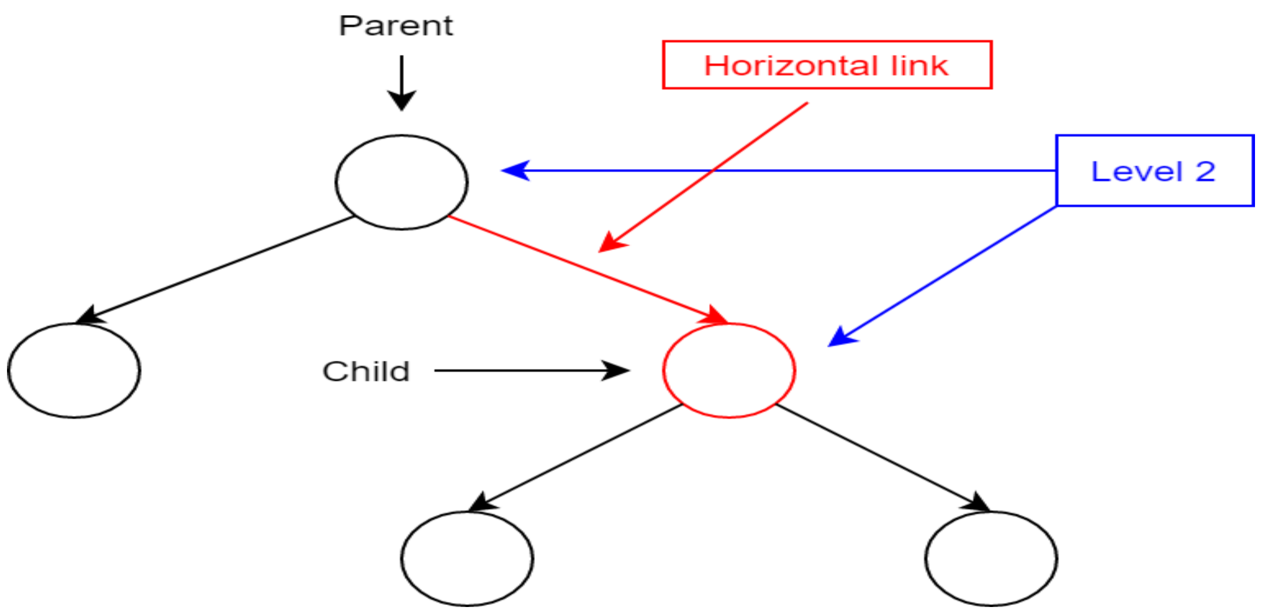
- Level of every leaf node is 1.



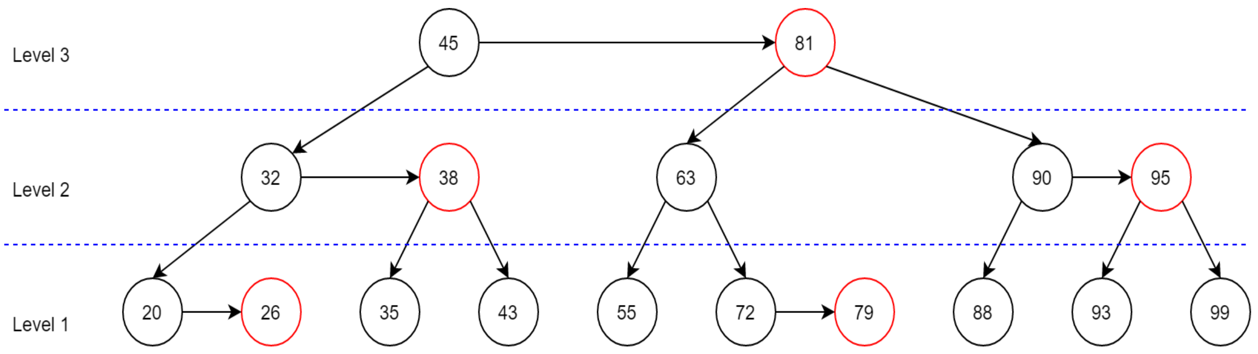
Horizontal link:

- Is a link between a parent node and its child whose levels are the same.

- Red nodes are at the same level as their parent, and the link between them is a horizontal link.



Re-draw the example AA tree with levels and horizontal links, the tree look like this:



The AA tree follow the same rule as RB tree, with some new exclusive properties.

- Each node can be either red or black and root node is always black. In implementing, we don’t even need to use color for AA tree.

- Horizontal link is always oriented to the right. In other words, the red nodes, which connect to their parent by a horizontal link, are always the right child of their parent.

- There are no two adjacent red nodes (or horizontal links).

- Every node with level greater than 1 will have 2 children.

- If a node has no horizontal link, then its children are at the same level.

- Every path from a node (including root) to any of its descendant NULL node has the same number of black nodes. (or black links / normal links).

An AA tree need to satisfy five invariants :

- The level of leaf node is 1.

- The level of left child is exactly one less than of its parent.

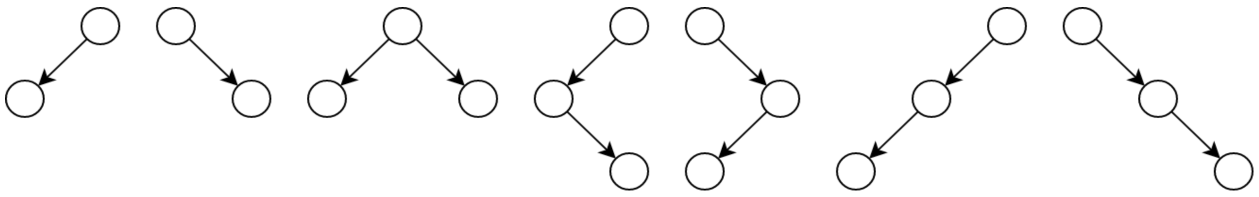
- The level of every right child is equal to or one less than of its parent.

- The level of every right grandchild is strictly less than that of its grandparent.

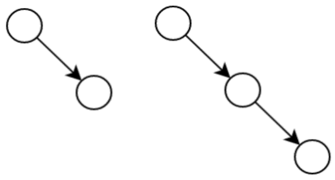
- Every node of level greater than one has two children.

The implementation of the RB tree is very complex with so many cases that need to be considered to properly balance the tree. The AA tree simplifies it by removing many cases, make it easier to code.

RB tree needs to consider seven different shapes when balancing :



The AA tree reduce that number to two :



An AA tree node can be design similar to the RB tree node, without the need of parent node address and color status, and adding the level value.

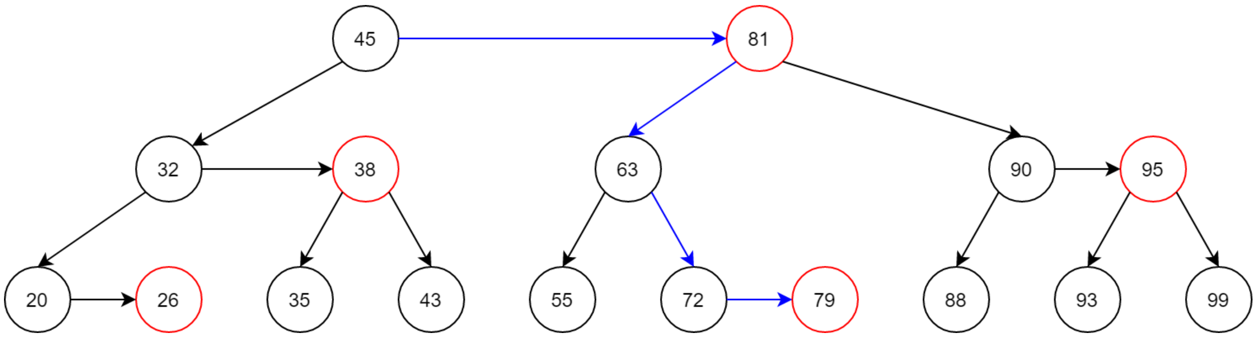
```
struct node
{
	int key;
	node* left;
	node* right;
	int level;
};
```

II. Operations.

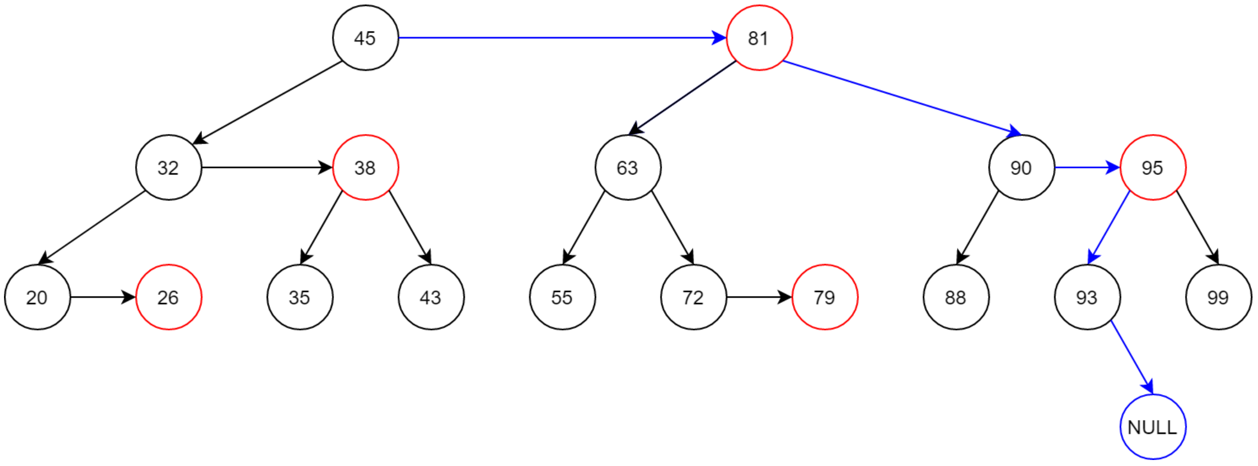
1. Traversing and searching.

Traversing and searching in AA tree is similar with a normal binary search tree.

For example : search for 79



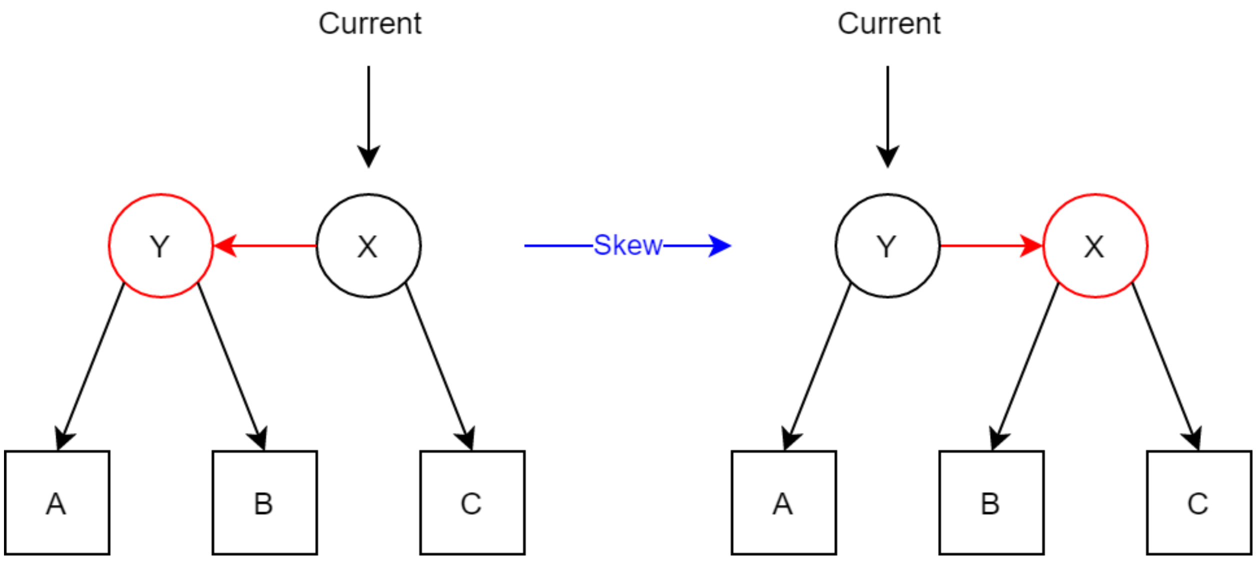
Search for 94 (not found)



2. Skew and split.

Insertions and deletions may make the AA tree become unbalanced (by breaking the rules). Only two operations are needed to help rebanlacing after inserting or deleting, those are skew and split.

Skew is a single right rotation used when there is a left horizontal link, or in other words, there is a left red child. Skew can create consecutive right horizontal link in process.

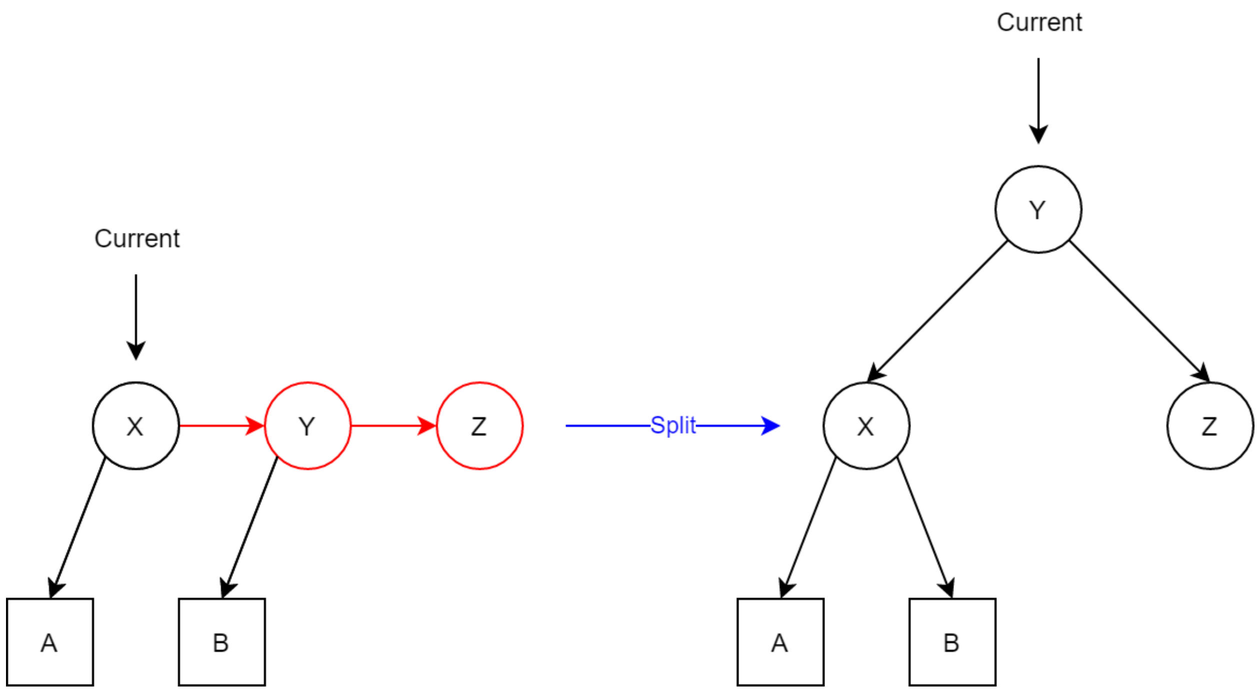


C++ code:

```
node* skew(node* curr)
{
	if (curr == NULL) return curr; // NULL node break

	if (curr->left != NULL) // the left child must exist
	{
		if (curr->left->level == curr->level) // the left child is at the same level as the
parent
		{
			node* term = curr->left;
			curr->left = term->right;
			term->right = curr;

			curr = term;
		}
```

```
        }
        return curr;
}
```

Split is a single left rotation used when there are two consecutive right horizontal links, or in other words, two consective red node. Split also increase level of a node in process, in someway, the AA tree grow up similar to a B-tree.



C++ code:

```
node* split(node* curr) // NULL node break
{
        if (curr == NULL) return curr;

        if (curr->right != NULL) // the right child must exist
        {
                if (curr->right->right != NULL) // the right grandchild must exist
                {
                        // the right grandchild is at the same level as the grandparent
                        if (curr->right->right->level == curr->level)
                        {
                                node* term = curr->right;
                                curr->right = term->left;
                                term->left = curr;
                        term->level++;

                        curr = term;
                    }
                }
            }
            return curr;
}
```

3. Inserting.

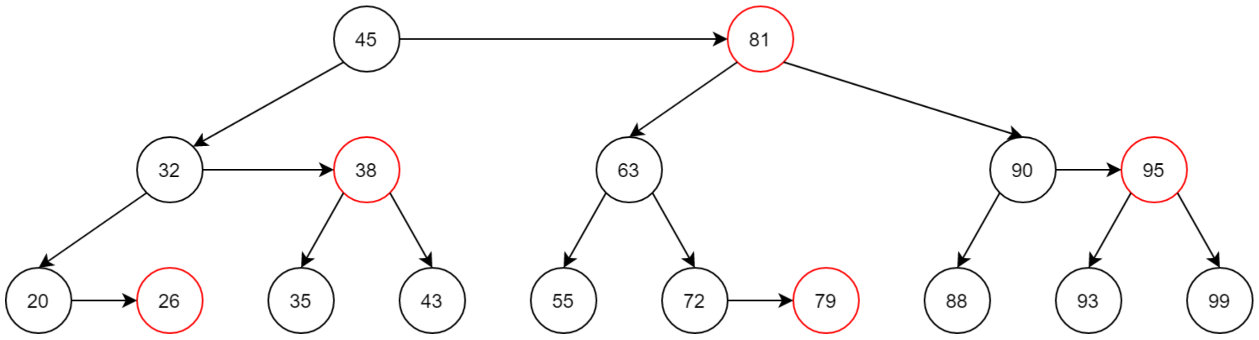
To add a new key to an AA tree, first execute like in a normal BST. The new node is always added at level 1.

- If it is added as a left child, it will create a left horizontal link : use skew operation to fix.

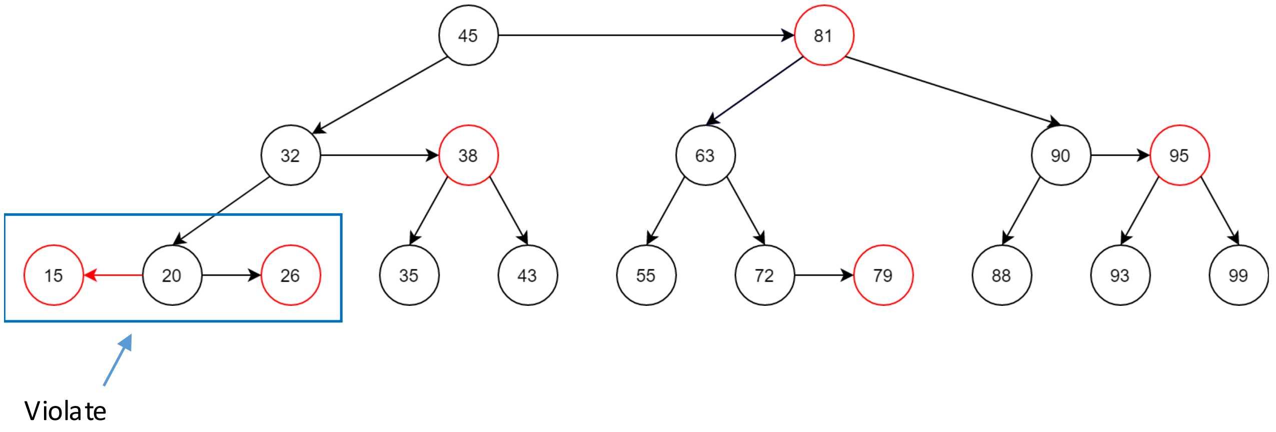
- If it is added as a right child, it will create a right horizontal link : if it cause two consecutive right horizontal link, use split operation to fix.

- Traverse back to root and fix if there is any violate.

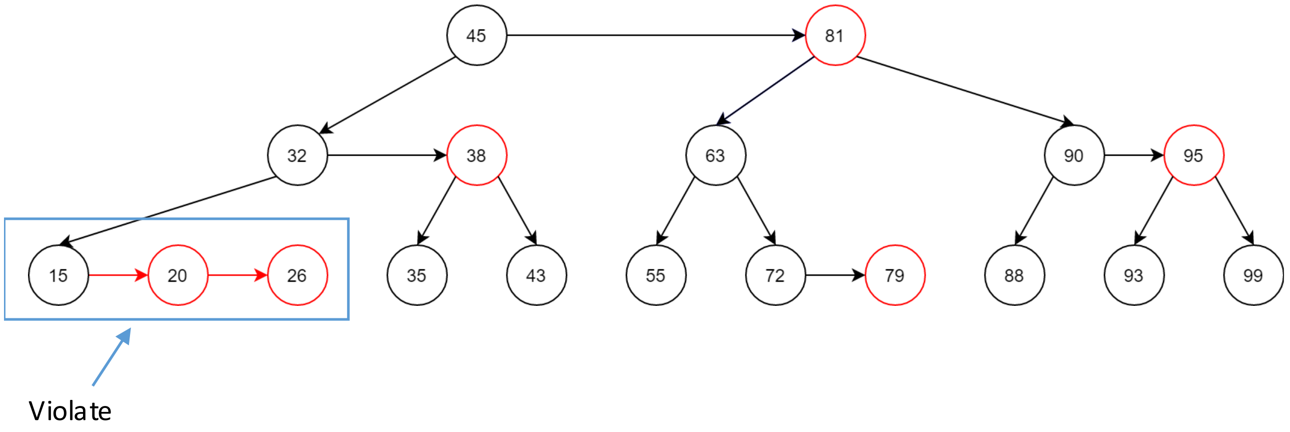
Example : add 15 to this tree



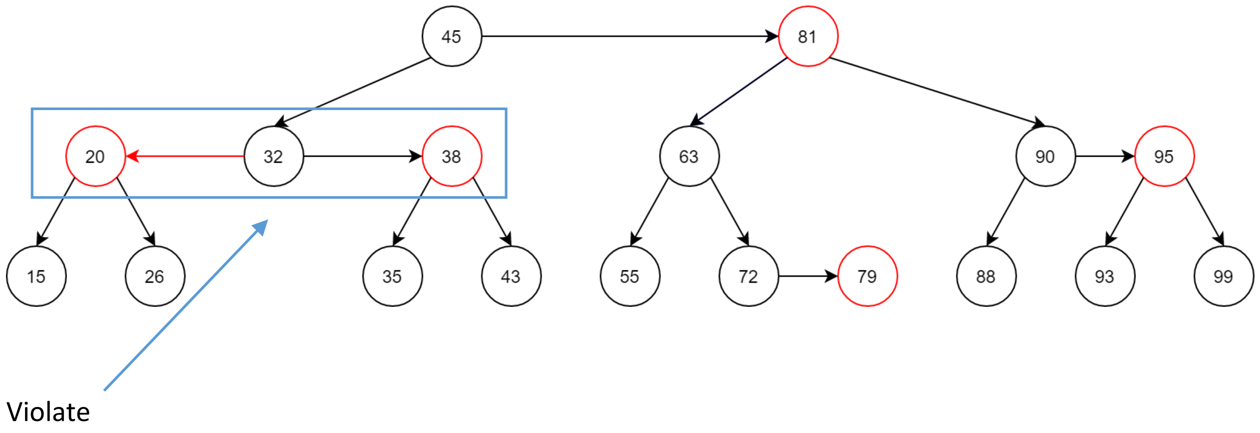
Traverse the tree to reach a NULL node, create a new node and add 15 to it. Level of this new node is 1.



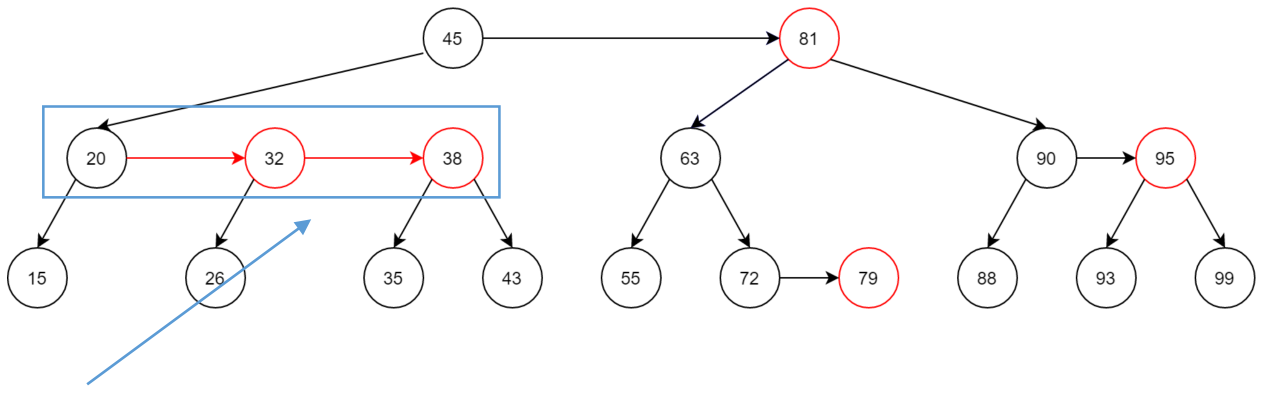
A left horizontal link is created, use skew at 20 to fix.



After skew at 20, two consecutive right horizontal links is created, use split at 15 to fix.

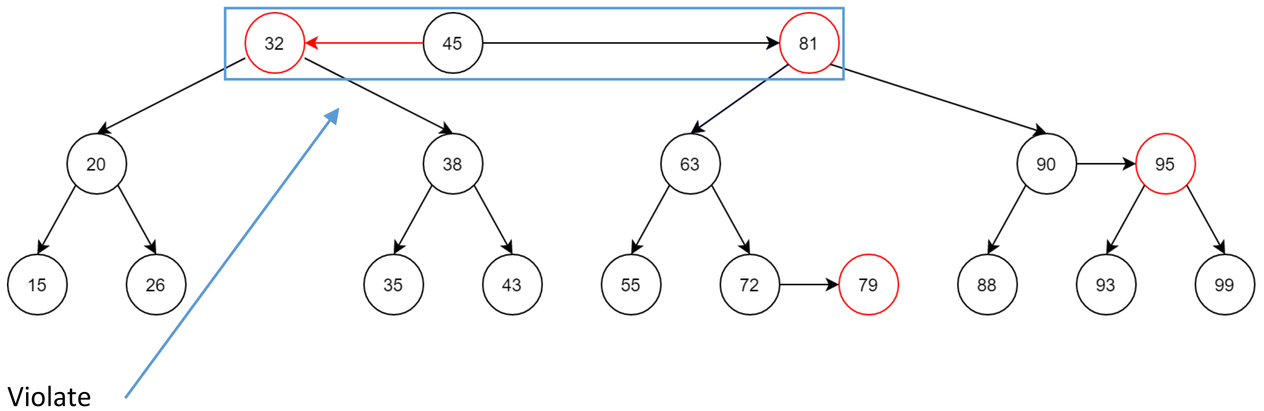


The split at 15 increase level of 20 to 2, putting it at the same level as its parent 32 and create a left horizontal link. Use skew at 32 to fix.



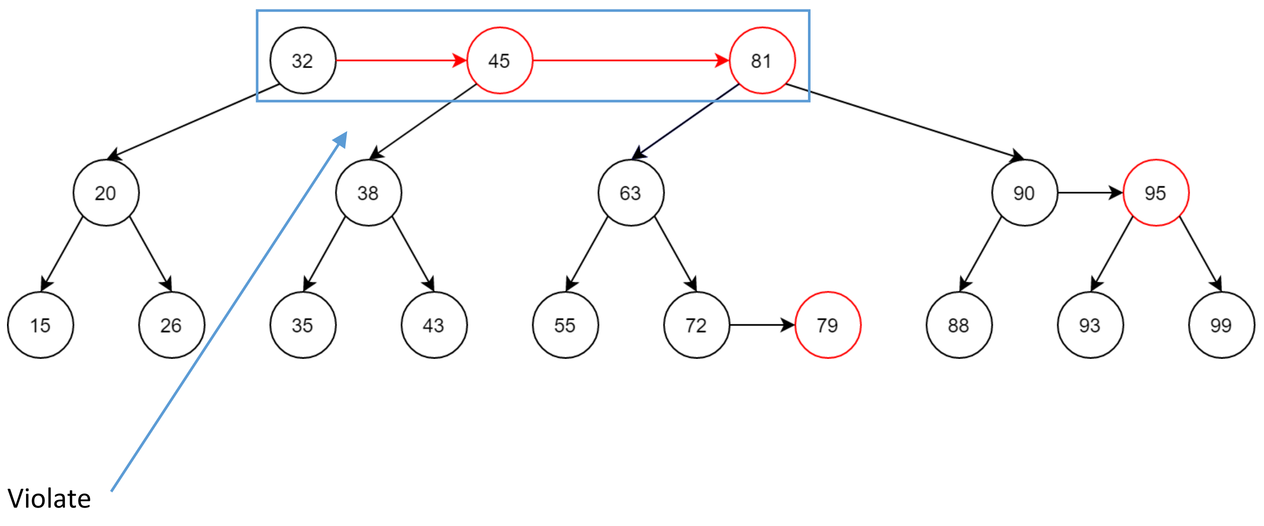
Violate

Again, the skew operation cause two consecutive right horizontal links. Use split at 20 to fix.

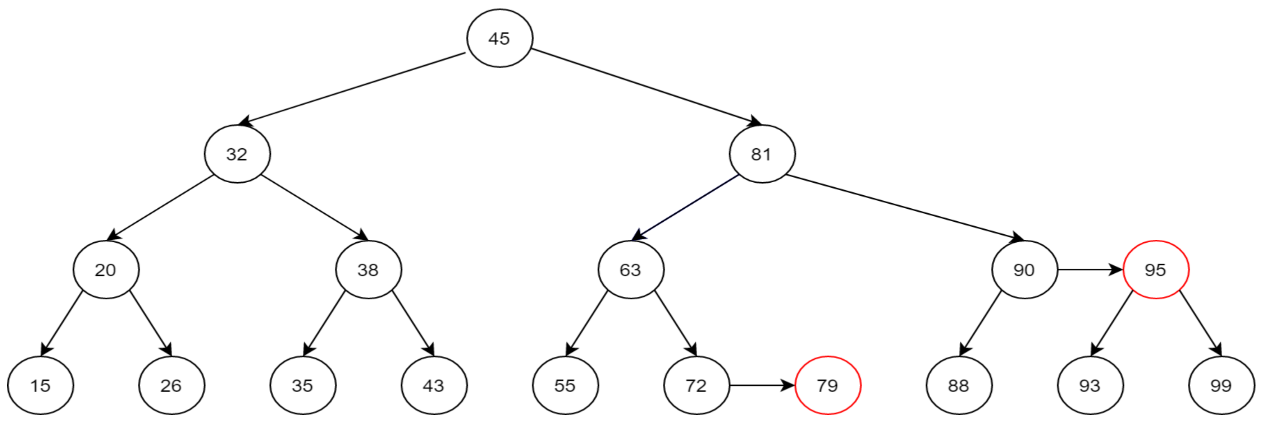


Violate

32 is put at level 3 and create a left horizontal link. Use skew at 45 to fix, 32 is now the new root.

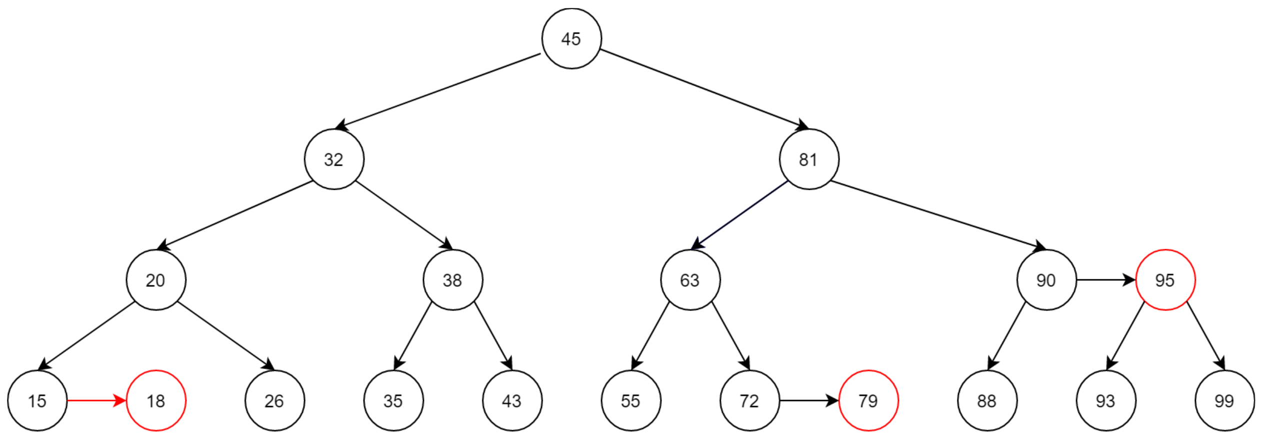


The skew cause two consecutive right horizontal links. Use split at 32 to fix.



45 is now at level 4 and is the new root.

Adding 15 cause so many change to the tree, even create a new level. But now try again with adding 18, there is only 1 right horizontal link created, no adjustment needed.



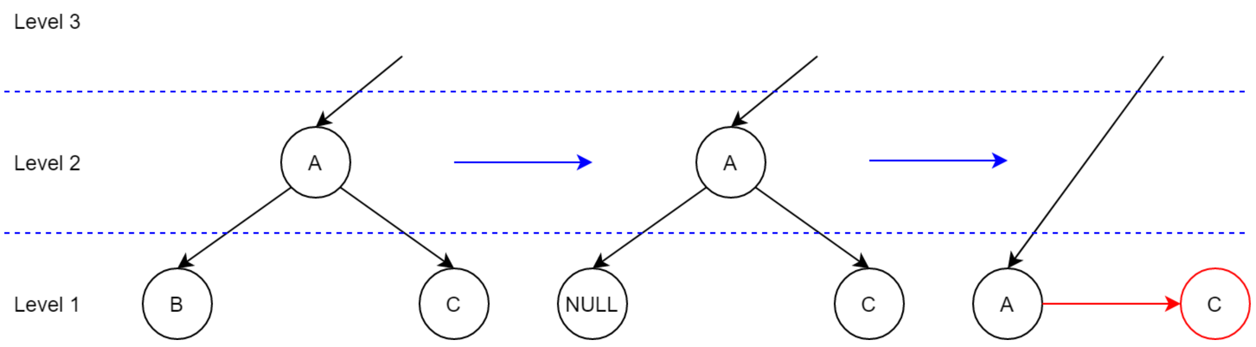
Implement in C++:

```
node* AAinsert(node* curr, int key)
{
        if (curr == NULL)
        {
                curr = new node;
                curr->key = key;
                curr->left = NULL;
                curr->right = NULL;
                curr->level = 1;
        }
        else if (key < curr->key)
        {
                curr->left = AAinsert(curr->left, key);
        }
        else if (key > curr->key)
        {
                curr->right = AAinsert(curr->right, key);
        }
        else return curr;

        curr = skew(curr);
        curr = split(curr);

        return curr;
}
```

4. Deletion.

To delete a key, first we also execute like in a normal BST. The actual deleted node is always a leaf node (always in level 1).

In the process, a node may lose one of its children, so we need to decrease its level. The ideal level of any node in the tree is one more than the minimum level of its children. If a node current level is higher than its ideal level, we need to decrease its to that ideal value. If the node has a right child that is a red node (at the same level and connected horizontally), then the level of the right child is also needs to be decreased.

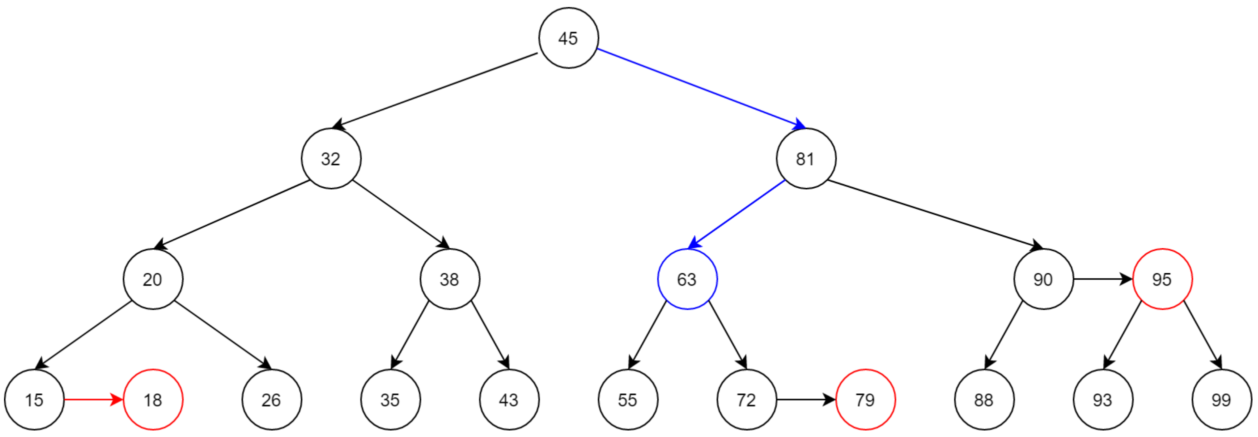


After updating the level of any node, the entire level need to be skewed and split.

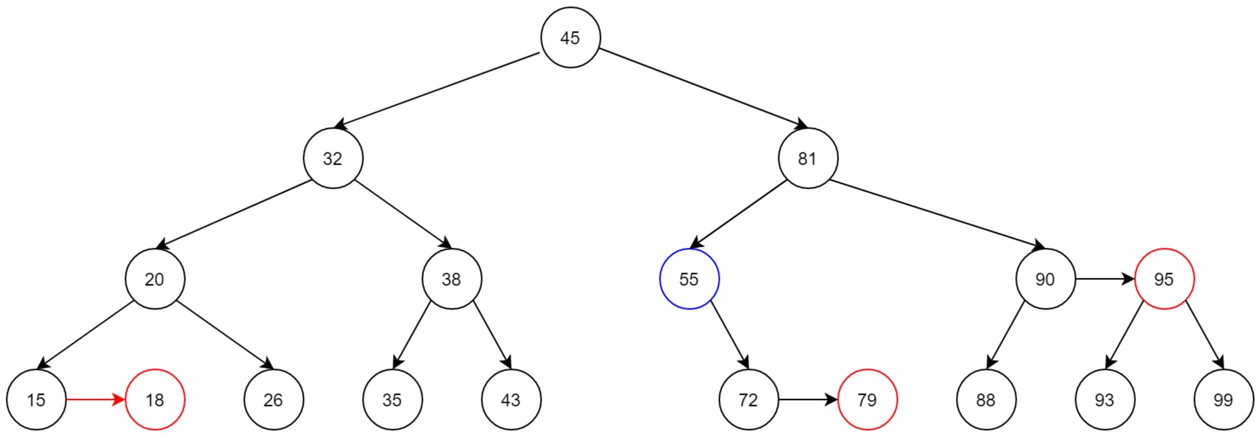
- Perform skew at current node, its right child, and it right grandchild.

- Next, perform a split at current node, which turn it into its right child and turn its old grandchild into its new right child. Then, perform another split at the new right child.

Example : removing 63

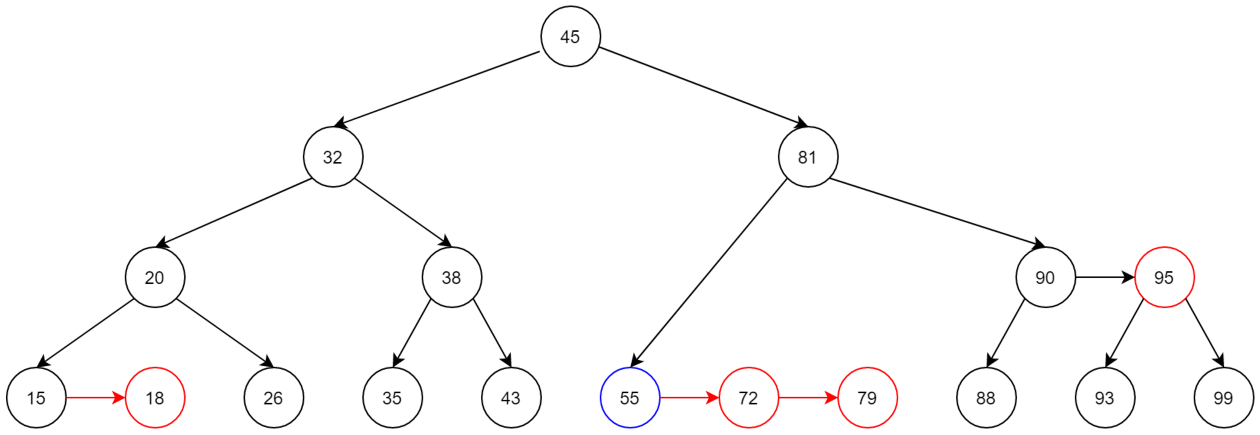


63 has two child, replace 63 with the largest value on its left sub tree, which is 55, and remove 55 from this left tree.



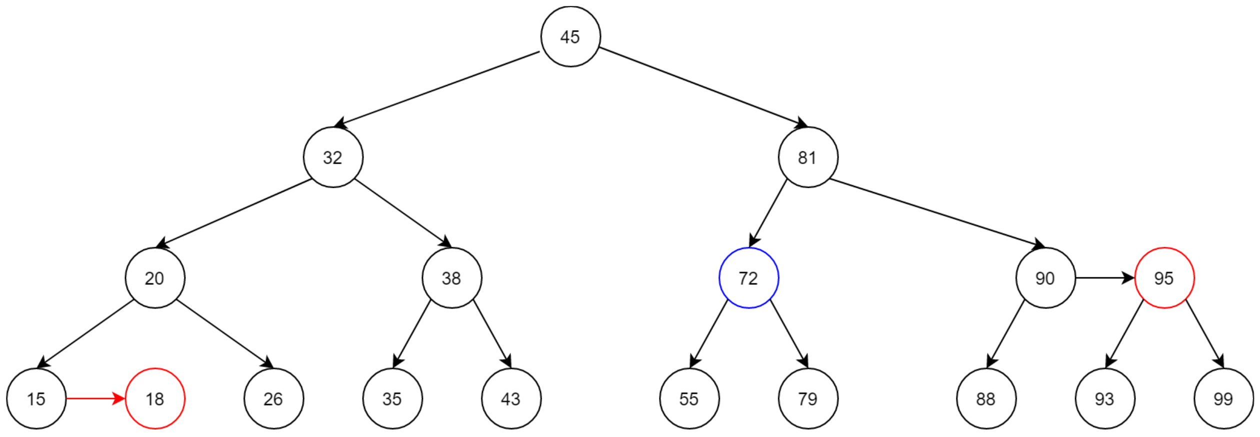
Now we check back up to the root node to see if there is any violation.

The ideal level of the node 55 is [1 + min(0,1)] = 1, but it is at level 2, decrease its level by 1. 55 don't have right horizontal link, so we skip that part.

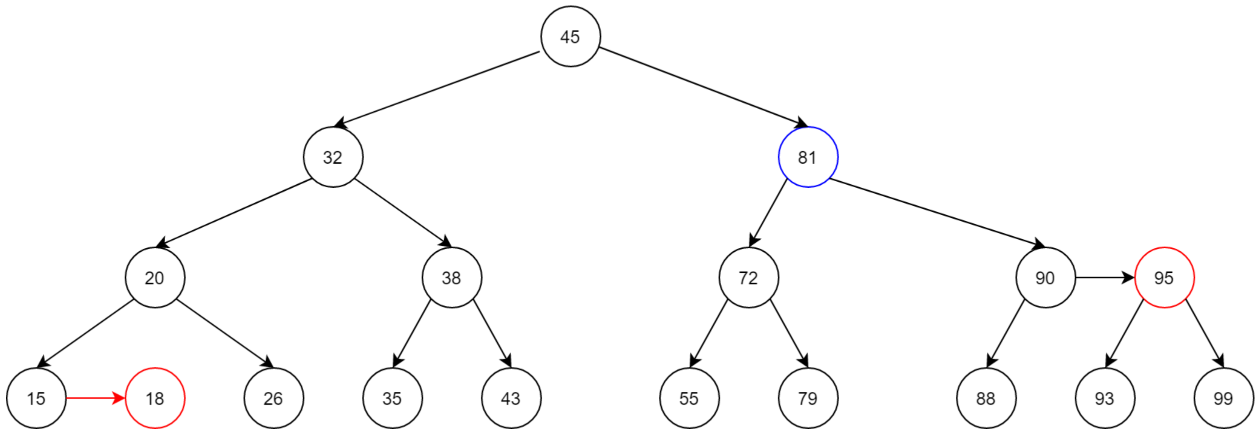


Perform skew at current node 55, its right child 72 and right grandchild 79. None of these node have left horizontal link, so nothing change.

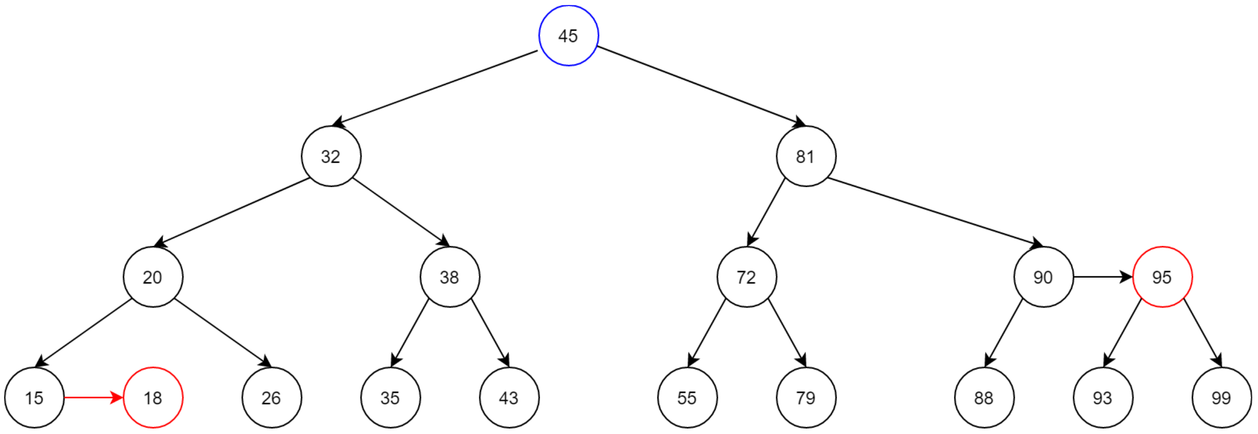
Use split at current node 55.



Then split at the right child 79. This node doens’t have right child so the split doesn’t happen.



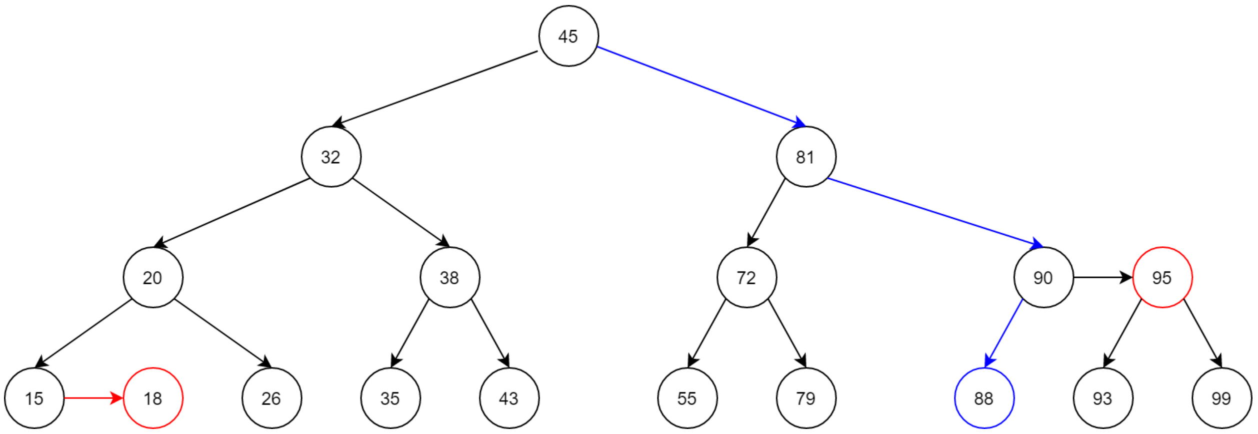
The ideal level of 81 is [1 + min(2,2)] = 3 is equal to its current level, no level updating is done. Skew at 81, 90 and 95, all are not happen. Split at 81 doesn’t happen because the level of its right grandchild 95 is different from its. Split at the right child 90 doesn’t happen because the level of its right grandchild 99 is different from its.



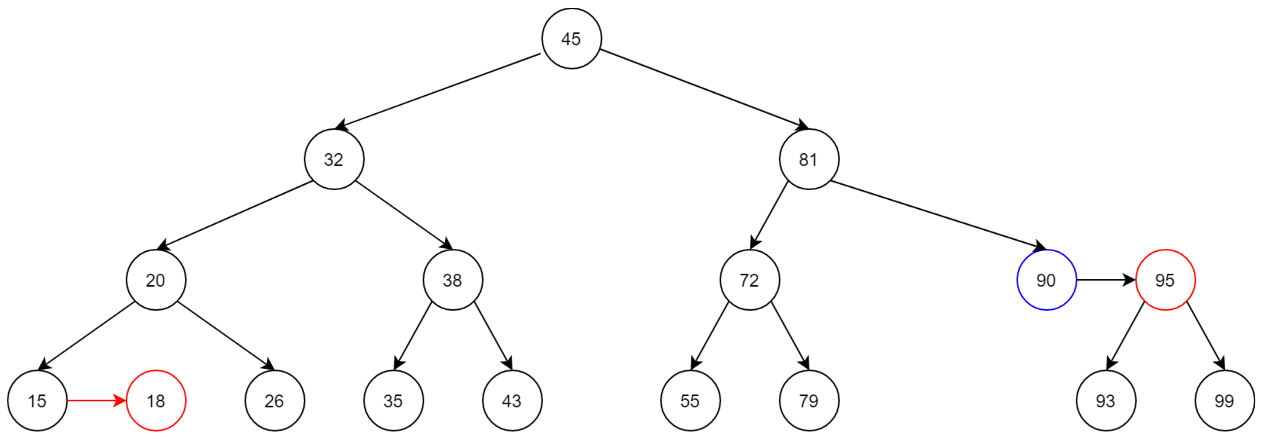
The ideal level of 81 is [1 + min(3,3)] = 4 is equal to its current level, no level updating is done. Skew at 45, 81 and 90, all are not happen. Split at 45 doesn’t happen because the level of its right grandchild 90 is different from its. Split at the right child 81 doesn’t happen because the level of its right grandchild 95 is different from its.

The root node is reached, stop balancing.

Now let’s remove 88.

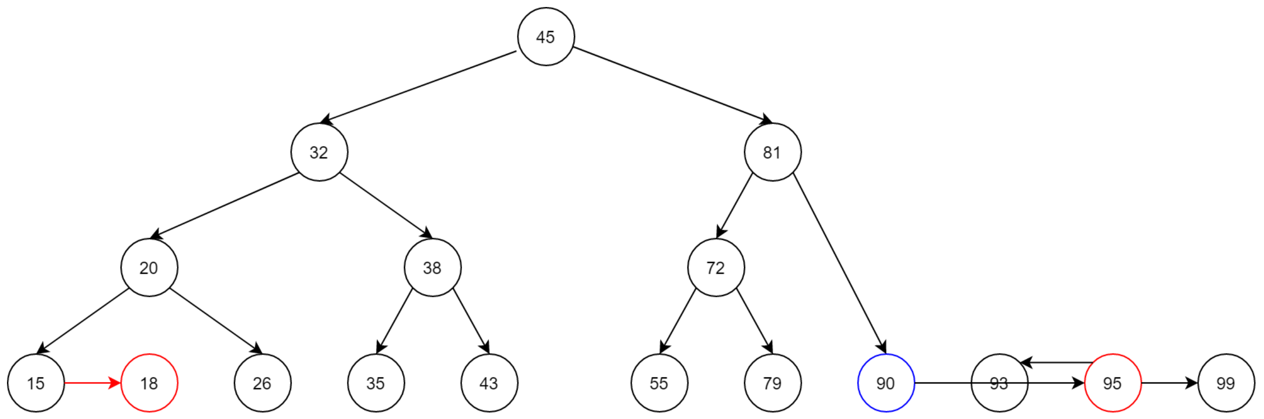


88 doesn’t have any child, just remove it.

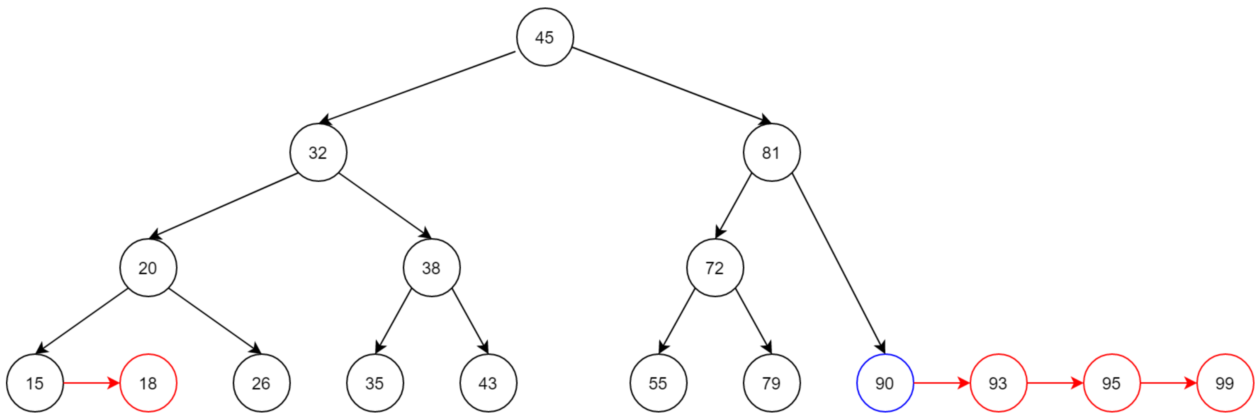


Begin rebalancing.

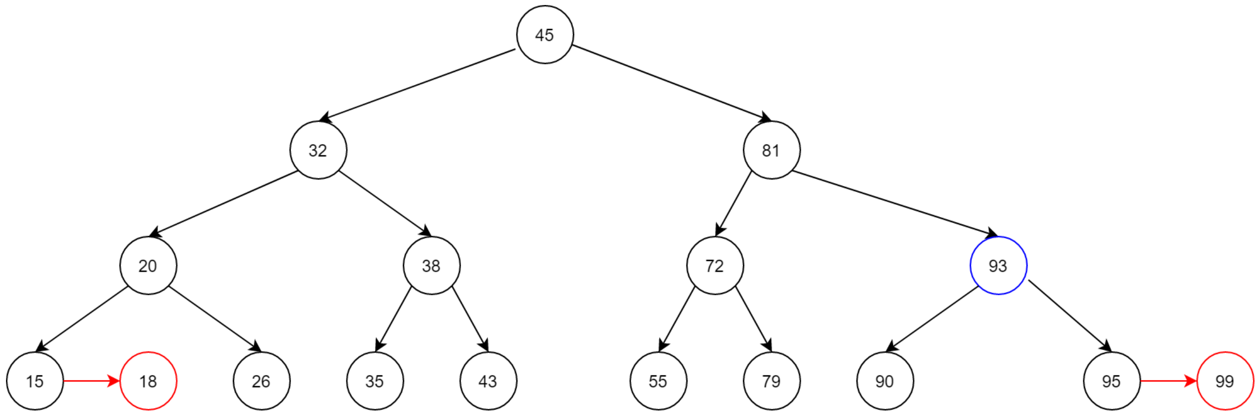
The ideal level of 90 is [1 + min(0,1)] = 1 is lower than its current level (2). Set 90 level to 1, and 90 has a horizontally connected right child 95, whose level will also be set to 1.



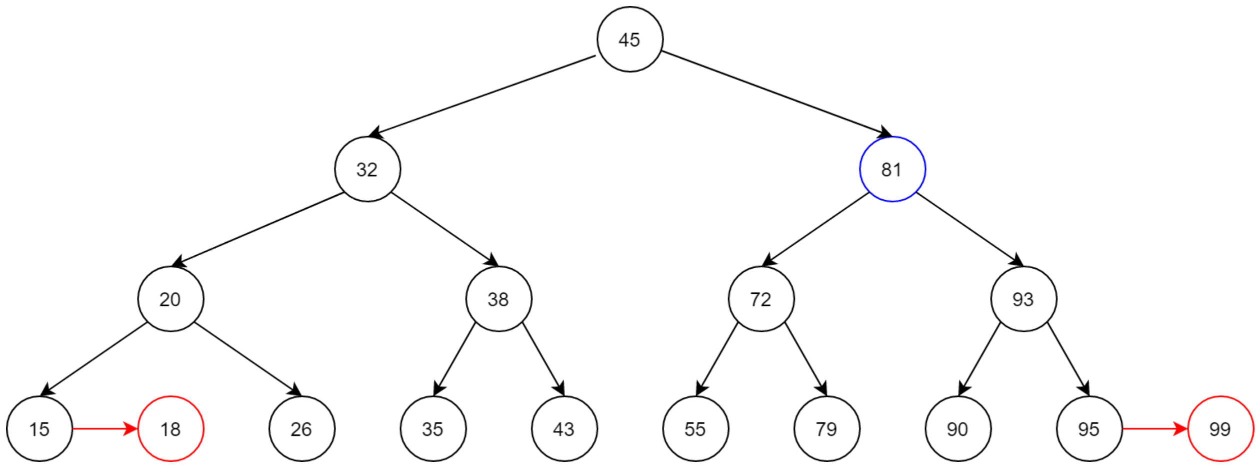
Perform skew at current node 90 doesn’t happen, at its right child 95 which happen and turn 95 into it new right grandchild. The skew at this new grandchild 95 doesn’t happen.



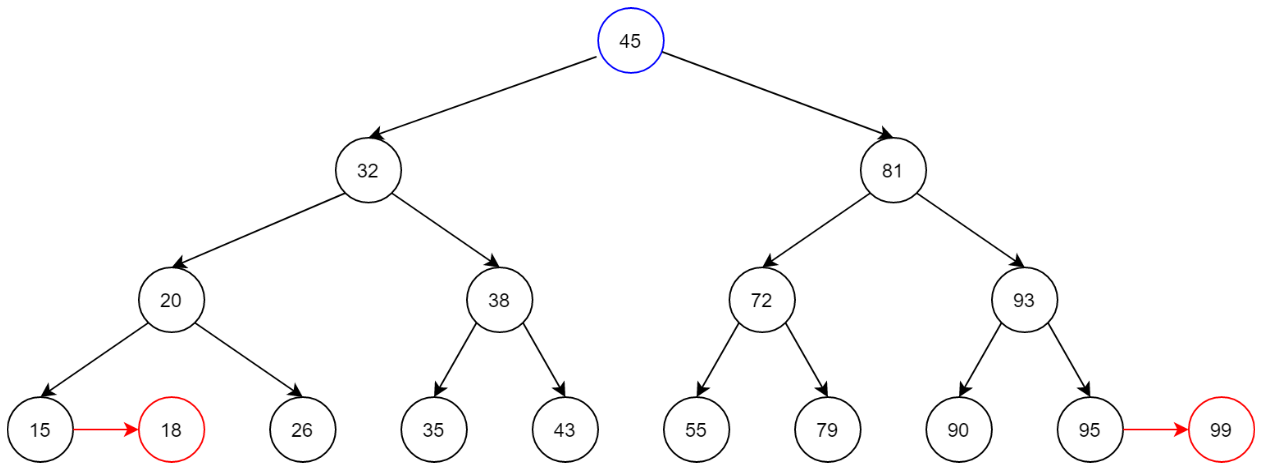
Perform split at 90.



The split at 95 doesn’t happen because its right grandchild doesn’t exist.



The ideal level of 81 is [1 + min(2,2)] = 3 is equal to its current level, no level updating is done. Skew at 81, 93 and 95, all are not happen. Split at 81 doesn’t happen because the level of its right grandchild 95 is different from its. Split at the right child 93 doesn't happen because the level of its right grandchild 99 is different from its.



The ideal level of 45 is [1 + min(3,3)] = 4 is equal to its current level, no level updating is done. Skew at 45, 81 and 93, all are not happen. Split at 45 doesn't happen because the level of its right grandchild 93 is different from its. Split at the right child 81 doesn't happen because the level of its right grandchild 95 is different from its.

The root node is reached, stop balancing.

C++ code:

```
int minChildLevel(node* curr)
{
        // return the minimum level of current node's children
        int l, r;
        if (curr->left == NULL) l = 0;
        else l = curr->left->level;

        if (curr->right == NULL) r = 0;
        else r = curr->right->level;

        return (l < r) ? l : r;
}

void updateLevel(node* curr)
{
	int idealLevel = minChildLevel(curr) + 1;
	if (curr->level > idealLevel) curr->level = idealLevel;
	if ((curr->right != NULL)&&(curr->right->level > idealLevel)) curr->right->level = 
idealLevel;
}

void replaceLeft(node* curr,node* ori)
{
	// replace current key with largest key in left sub tree
	if (curr->right != NULL) return replaceLeft(curr->right, ori);

	ori->key = curr->key;
}

void replaceRight(node* curr, node* ori)
{
	// replace current key with smallest key in right sub tree
	if (curr->left != NULL) return replaceRight(curr->left, ori);

	ori->key = curr->key;
}

node* AArebalance(node* curr)
{
	updateLevel(curr);

	curr = skew(curr);
	curr->right = skew(curr->right);
	curr->right->right = skew(curr->right->right);

	curr = split(curr);
	curr->right = split(curr->right);

	return curr;
}

node* AAdelete(node* curr, int key)
{
	if (curr == NULL) return NULL;

	if (key < curr->key) curr->left = AAdelete(curr->left, key);
	else if (key > curr->key) curr->right = AAdelete(curr->right, key);
	else
	{
		if (curr->left != NULL)
		{
```

```
                        replaceLeft(curr->left, curr);
                        curr->left = AAdelete(curr->left, curr->key);
                }
                else if (curr->right != NULL)
                {
                        replaceRight(curr->right, curr);
                        curr->right = AAdelete(curr->right, curr->key);
                }
                else
                {
                        delete curr;
                        return NULL;
                }
        }

        return AArebalance(curr);
}
```