DSA

▼ pre hash table topics:

▼ char* const p vs const char* p

In C, `char* const p` and `const char* p` are both pointers, but they have different meanings and uses.

`char* const p` declares a constant pointer to non-constant data. This means that the pointer `p` points to a memory location that holds a character value that can be modified, but the pointer itself cannot be modified to point to another memory location. This is useful when you want to ensure that the pointer remains pointing to the same memory location throughout the program execution.

On the other hand, `const char* p` declares a pointer to constant data. This means that the pointer `p` can be modified to point to different memory locations, but the data at the memory location pointed to by `p` cannot be modified through `p`. This is useful when you want to ensure that the data pointed to by the pointer cannot be modified, but the pointer itself can be used to point to different data.

▼ dynamic allocation of a 2d array

```
// Allocate memory for an array of pointers to int
int **arr = (int **) malloc(rows * sizeof(int *));

// Allocate memory for each row of the 2D array
for (int i = 0; i < rows; i++) {
    arr[i] = (int *) malloc(cols * sizeof(int));
}
```

▼ recursive functions on ADTs

- recursively delete a linked list
- create a copy of a singly linked list using recursion
- recursively reverse a linked list in place

▼ #ifndef and #endif

In C, `#ifndef` and `#endif` are preprocessor directives that are used to create header file guards.

When a header file is included in a C program, the preprocessor copies the contents of the header file into the program before compilation. If the same header file is included multiple times, this can result in duplicate code and compilation errors.

To avoid this, header file guards are used. The `#ifndef` directive checks whether a symbol has been defined, and if not, it defines the symbol and includes the code between the `#ifndef` and `#endif` directives. If the symbol has already been defined, the code between the `#ifndef` and `#endif` directives is skipped.

For example, suppose we have a header file named `example.h`. We can create a header file guard by using `#ifndef` and `#endif` as follows:

```
#ifndef EXAMPLE_H
#define EXAMPLE_H

/* code for the header file goes here */

#endif
```

In this example, the symbol `EXAMPLE_H` is defined if it has not already been defined. If `EXAMPLE_H` has already been defined, the code between `#ifndef` and `#endif` is skipped. This ensures that the contents of `example.h` are only included once in a given C program.

▼ stack applications

- parenthesis matching
- expression evaluation
- tower of hanoi problem
- printing in level order using stacks
- previous smaller element, next smaller element

▼ wraparound functionality in Queue

To implement wraparound functionality in a queue in C, you can use an array with a fixed size and two pointers to keep track of the front and rear of the queue.

Implementation:

```
#define MAX_SIZE 10  // maximum size of the queueint queue[MAX_SIZE];  // arra
y to hold the queue elements
int front = 0;  // pointer to the front of the queue
int rear = -1;  // pointer to the rear of the queue

void enqueue(int value) {
    if ((rear + 1) % MAX_SIZE == front) {
        printf("Queue is full\n");
        return;
    }
    rear = (rear + 1) % MAX_SIZE;
    queue[rear] = value;
}

int dequeue() {
    if (front == rear + 1) {
        printf("Queue is empty\n");
        return -1;
    }
    int value = queue[front];
    front = (front + 1) % MAX_SIZE;
    return value;
}
```

In this implementation, the `%` operator is used to wrap around the indices of the queue when they reach the end of the array. This allows the queue to behave as if it has a circular structure, and new elements can be added to the front of the queue even when the rear pointer has reached the end of the array.

- tree parameters

  Common parameters used to describe trees:

  1. Root: The topmost node in a tree, which has no parent.
  2. Parent: A node is the parent of another node if it has a direct edge to that node.
  3. Child: A node is a child of another node if it is connected to its parent via a direct edge.
  4. Siblings: Nodes that share the same parent are called siblings.
  5. Leaf: A node that has no children is called a leaf or a terminal node.
  6. Height: The height of a tree is the maximum number of edges from the root node to a leaf node. Alternatively, it is the longest path from the root to a leaf node.
  7. Depth: The depth of a node is the number of edges from the root to that node.
  8. Level: The level of a node is defined as its depth plus one.
  9. Subtree: A subtree is a portion of a tree that includes a node and all of its descendants.
  10. Degree: The degree of a node is the number of children it has.

- general tree implementation
- printing directory: play around with indentation, order.

```
// Function to print a file directory with proper indentation
void printDirectory(struct node *root, int depth) {
    // Base case: if the root is null, return
    if (root == NULL) {
        return;
    }

    // Print the name of the current node with proper indentation
    for (int i = 0; i < depth; i++) {
        printf("\t");
    }
    printf("%s\n", root->name);

    // Recursively print the children and siblings of the current node
    printDirectory(root->child, depth + 1);
    printDirectory(root->sibling, depth);
}
```

- BST traversal: in-order, pre-order, post-order and their properties.
- making tree from given traversals.
- time-complexities of various operations that can happen on a tree.
- deleting a node in BST and deleting a whole tree.

▼ hash tables:

▼ hash functions for int, string.

▼ integers

The most commonly used hash functions for integer data types:

1. `Modular hashing:` This method involves taking the integer value and computing the remainder when divided by a prime number. The prime number should be chosen carefully to minimize collisions.
2. `Folding hashing:` This method involves dividing the integer into smaller parts and adding them together. For example, if the integer is 123456, it could be divided into 12, 34, and 56 and then added together (12 + 34 + 56 = 102).

▼ strings

The most commonly used hash functions for string data types:

1. `ASCII sum hashing:` This method involves summing the ASCII values of each character in the string. The resulting sum can then be used as the hash value.
2. `Polynomial rolling hashing:` This method involves treating the string as a polynomial with each character represented as a coefficient. The hash value is then computed using a rolling algorithm that takes into account the previous hash value and the new character.

▼ insertion and search in hash table

- separate chaining

- linear probing
- quadratic probing

▼ clustering

The problem of clustering in a hash table occurs when a large number of elements are hashed to the same bucket or nearby buckets, resulting in a longer search time to access those elements. This happens because hash functions are not perfect and can produce collisions, where multiple keys are hashed to the same bucket.

Clustering can be avoided by using a good hash function that distributes the keys as evenly as possible across the buckets of the hash table. A good hash function minimizes the number of collisions by mapping keys that are likely to be used together to different buckets. The hash function should also be fast to compute, so that the time to hash a key is not a bottleneck.

One technique to reduce clustering is to use a technique called "open addressing," where collisions are resolved by finding the next available slot in the table instead of using a separate data structure, such as a linked list, to store the collided items. Different probing methods, such as linear probing or quadratic probing, can be used to determine the next available slot. By using open addressing, the search for a key can be done quickly, since it does not require following pointers to a separate linked list.

Another technique to avoid clustering is to use a resizing policy that increases the size of the hash table when the load factor exceeds a certain threshold. By increasing the number of buckets in the hash table, the likelihood of collisions and clustering can be reduced.

▼ load factor concepts, collision:

`Load factor` in a hash table refers to the ratio of the number of elements stored in the hash table to the number of slots in the hash table. In other words, it represents the amount of "occupied" space in the hash table.

Load factor is an important factor in the performance of a hash table because it affects the number of collisions that occur during hash table operations. Collisions occur when two or more keys hash to the same index in the table. A high load factor means that there are many elements in the hash table and fewer empty slots, which increases the likelihood of collisions.

A good rule of thumb is to keep the load factor below a certain threshold, typically 0.7 or 0.8. If the load factor exceeds this threshold, the hash table may need to be resized to accommodate more elements, which can be a costly operation in terms of time and memory usage.

▼ horner's rule

Horner's rule: in which a polynomial

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

is written in *nested form:*

$$f(x) = (((\cdots((a_n x + a_{n-1})x + a_{n-2})\cdots)x + a_1)x + a_0)$$

This allows the evaluation of a polynomial of degree *n* with only *n* multiplications and *n* additions. This is optimal, since there are polynomials of degree *n* that cannot be evaluated with fewer arithmetic operations.

▼ re-hashing

`Rehashing` is the process of increasing the size of a hash table and redistributing its contents to new buckets. This is typically done when the load factor (the ratio of the number of elements in the hash table to the size of the underlying array) exceeds a certain threshold, such as 0.7. Rehashing ensures that the hash table remains efficient by reducing the number of collisions and maintaining a low load factor.

For a `linear probed hash table`, rehashing involves creating a new array with twice the number of buckets as the original and copying all the elements from the old array to the new array. Each element is then inserted into the new array using the linear probing algorithm.

For a `quadratic probed hash table`, rehashing also involves creating a new array with twice the number of buckets as the original and copying all the elements from the old array to the new array. However, the elements are then inserted into the new array using the quadratic probing algorithm, which takes into account the quadratic function of the current probe index to ensure that collisions are avoided.

▼ sorting:

Insertion Sort

- `Introduction:` Insertion sort is a sorting algorithm that repeatedly selects an element and inserts it into the correct position within a sorted portion of the list.
- `Working:` Insertion sort works by maintaining a sorted portion of the list and gradually expanding it. It starts with the second element and compares it with the elements before it, inserting it into the correct position within the sorted portion. This process continues for each subsequent element until the entire list is sorted.
- `Time Complexity:` The time complexity of insertion sort is $O(n^2)$ in the worst and average case scenarios, where n is the number of elements in the list. This is because, on average, each element may need to be compared with roughly half of the elements in the sorted portion before finding its correct position
- `Algorithm:`
    - Iterate through the array from left to right.
    - Examine each element and compare it to the elements on the left of it.
    - If the element being examined is smaller than the left element, we keep swapping it with the left element until it reaches it’s correct sorted position, i.e. swap until it is bigger than its left element.

```
for x = 1 to len(A)-1
    y = x
    while y>0 and A[y-1]>A[y]
        swap A[y] and A[y-1]
        j=j-1
```

Bubble Sort

- `Introduction:` Bubble sort is a sorting algorithm that repeatedly compares adjacent elements and swaps them if they are in the wrong order, gradually moving the largest elements to the end of the list.
- `Working:` Bubble sort works by repeatedly comparing adjacent elements in the list and swapping them if they are in the wrong order. In each iteration, the largest element in the unsorted portion of the list "bubbles

up" to its correct position. This process continues until the entire list is sorted, with the largest elements eventually "bubbling" to the end of the list.

- `Time Complexity:` The time complexity of bubble sort is $O(n^2)$ in the worst and average case scenarios, where n is the number of elements in the list. This is because, in each iteration, bubble sort compares and swaps adjacent elements, generally requiring multiple passes through the entire list to fully sort it.

▼ `Algorithm:`

  - Compare each pair of adjacent elements from the beginning of the list. If the elements are in the wrong order, swap them.
  - Move to the next pair of elements and continue the comparisons until reaching the end of the unsorted part of the array.
  - After each pass through the list, the largest element will reach to the end of the unsorted part of the array.
  - Repeat the above steps until the list is sorted. Remember, the size of the unsorted part of the array decreases by one with each subsequent pass.

```
for x = 1 to N
  for j = 0 to N-1
    if A[j] > A[j+1]
      swap A[j] and A[j+1]
```

▼ Selection Sort

- `Introduction:` Selection sort is a sorting algorithm that repeatedly selects the smallest element from the unsorted portion of the list and swaps it with the element in the current position, gradually building a sorted portion of the list from left to right.
- `Working:` Selection sort works by dividing the input list into two portions: a sorted portion and an unsorted portion. In each pass, the algorithm finds the smallest element in the unsorted portion and swaps it with the element in the current position. This effectively grows the sorted portion by one element The process continues, with the sorted portion expanding from left to right, until the entire list is sorted.

- `Note:` Selection Sort minimizes the number of swaps by only performing a single swap per iteration.
- `Time Complexity:` The time complexity of selection sort is $O(n^2)$ in the worst and average case scenarios, where n is the number of elements in the list.

▼ `Algorithm:`

  - During each pass through the array, we'll find the smallest element in the unsorted portion of the array and move it to the sorted portion of the array.

```
for j = 0 to N-2
  int min_index = j
  for i = j+1 to N-1
      if A[i]<A[min_index]
          min_index = i
  if(min_index!=j)
    swap A[j] and A[min_index]
```

▼ Merge Sort

:) ikyk

▼ Quick Sort

- `Introduction:` Quick sort is a recursive sorting algorithm. Quick sort is a divide-and-conquer sorting algorithm that recursively divides the list into smaller sub-lists based on a chosen pivot element and then sorts these sub-lists independently.
- `Working:` Quick sort works by selecting a pivot element from the list and partitioning the other elements into two sub-lists, according to whether they are less than or greater than the pivot. This step is known as the partitioning process.
- The pivot is then placed in its final sorted position, with all elements to its left being smaller and all elements to its right being greater. This ensures that the pivot is in its correct position in the final sorted list.
- The partitioning process is recursively applied to the sub-lists on the left and right of the pivot until the entire list is sorted. This divide-and-conquer approach allows quick sort to efficiently sort large lists.

- `Performance and Time Complexity:` The choice of the pivot can affect the performance of quick sort. The four common pivots used are:
  1. Random element
  2. First/Last element
  3. Middle element
  4. Median of three strategy (median of the first, last and the middle element)
- Overall, quick sort has an average time complexity of $O(nlogn)$, making it one of the fastest sorting algorithms. However, in the worst case scenario, when the pivot selection is unbalanced, the time complexity can degrade to $O(n^2)$.

▼ `Algorithm:`

1. Choose a pivot element from the list.
2. Partition the list, rearranging the elements so that all elements less than the pivot are placed before it, and all elements greater than the pivot are placed after it. The pivot is now in its final sorted position.
3. Recursively apply steps 1 and 2 to the sub-lists on the left and right of the pivot.
4. Continue this process until each sub-list contains zero or one element, as these sub-lists are already considered sorted.
5. Combine the sorted sub-lists to obtain the fully sorted list.

```
quicksort(array, low, high)
    if low < high
        piv = partition (array, low, high)
        quicksort(array, low, piv - 1)
        quicksort(array, piv + 1, high)


partition(array, low, high)
    piv_value = array[high]
    i = low - 1;
    for j = low to high - 1
        if array[j] < piv_value
            i++
            swap array[i] and array[j]
    swap array[i+1] and array[high]
    return i+1;
```

▼ In-Situ Permutation Sort/Indirect Sorting

▼ Shell Sort

- `Introduction:` Shell sort is an in-place comparison-based sorting algorithm that divides the list into smaller sub-lists and performs insertion sort on them with decreasing gap sizes, eventually sorting the entire list.
- `Working:` Shell sort works by dividing the list into multiple sub-lists and applying an insertion sort-like algorithm to each sub-list.
- The sub-lists are created by selecting elements that are a certain gap distance apart. Initially, a large gap is used, and the elements are compared and swapped within the sub-lists.
- After each iteration, the gap size is reduced, and the process is repeated with the new gap size. This continues until the gap size becomes 1, at which point the algorithm performs a final insertion sort on the entire list.
- The idea behind Shell sort is that by initially sorting the elements that are far apart, it can quickly reduce the total number of inversions or out-of-place elements in the list. As the gap size decreases, the algorithm gradually improves the order of the elements until the list becomes fully sorted.
- Shell sort provides a balance between the simplicity of insertion sort and the efficiency of more complex sorting algorithms.
- `Time Complexity:` The time complexity of Shell sort is influenced by both the size of the input list and the chosen gap sequence. It is often considered an improvement over simple quadratic sorting algorithms like bubble sort or insertion sort but is generally not as efficient as more advanced algorithms like merge sort or quicksort.
- `Note:` On an average, Shell Sort does less number of comparisons than merge sort.

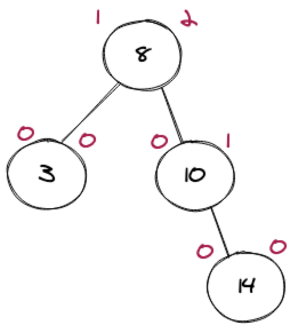
▼ `Algorithm:`

1. Start with an unsorted list of elements. Choose a gap size.
2. Repeat the following steps until the gap size becomes 1:
    - Divide the list into sub-lists of elements that are `gap` distance apart.

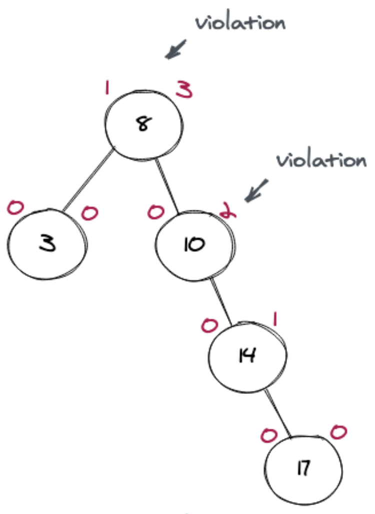
    - Perform an insertion sort on each sub-list, comparing and swapping elements within the sub-list.
    - Reduce the gap size using the chosen gap sequence.
3. Finally, perform a final insertion sort on the entire list with a gap size of 1. The list is now sorted.

▼ AVL trees:

- self balancing binary search trees (stricter form of BST).
- height of AVL tree = $ceil(log_2(n))$ always, where n is the number of nodes.
- benefits of using an AVL: all the three operations; insertion, search and deletion can be performed in $O(log_2(n))$ time.
- we define balance factor as $BF = \text{height of left subtree} - \text{height of right subtree}$.
- height is defined as: height of a node is the number of edges between the node and the deepest leaf node.
- property of AVL tree: $|BF| \leq 1$ for all nodes. violation will void the property of AVL.
- example to demonstrate the property of AVL tree:



AVL Tree



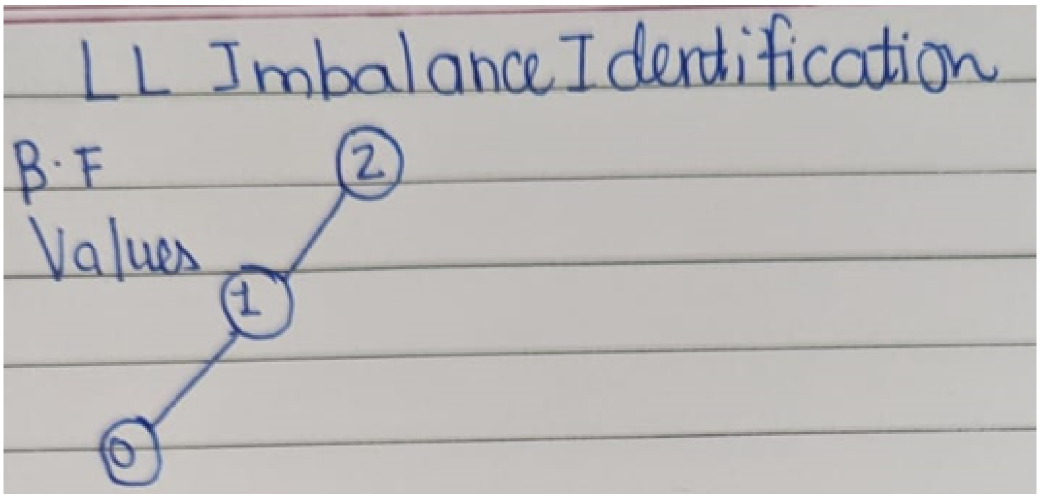
Not an AVL Tree

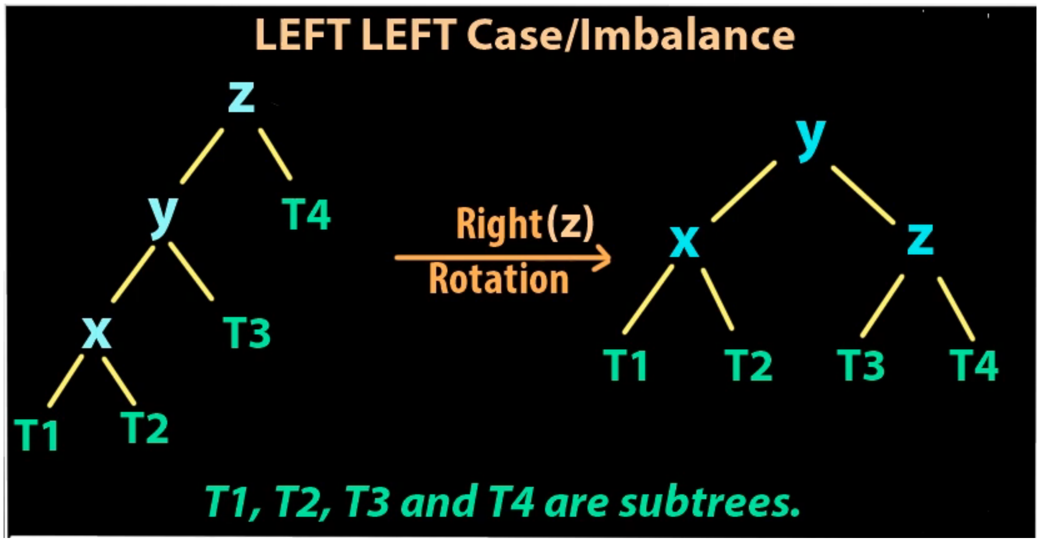
▼ four types of imbalances in a AVL tree:

- remember: T_1, T_2, T_3, T_4 are subtrees of the same height.
- we return the new root node in each function
- rotation routine is just $O(1)$ since balancing an unbalanced AVL tree is simply pointer manipulation.
- we use the value of BF to identify if a node is violating AVL property or not, and if it is, then what type of imbalance it is causing.

▼ LL imbalance:

- imbalance is due to the left child of the left subtree of Z.



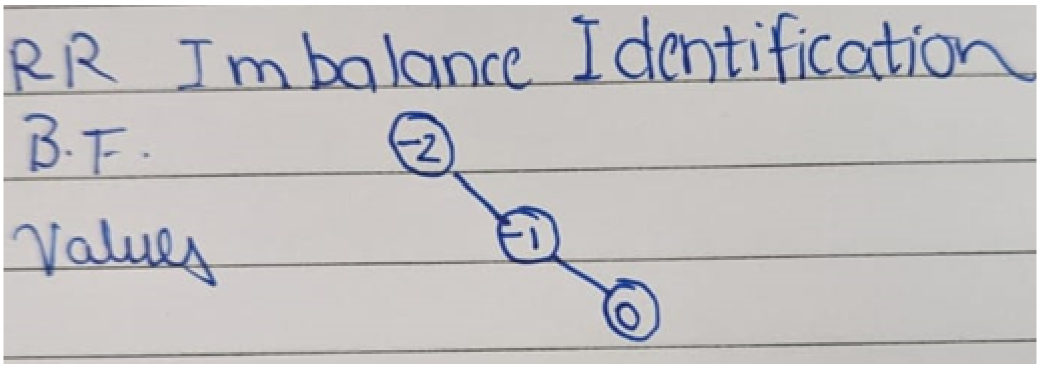


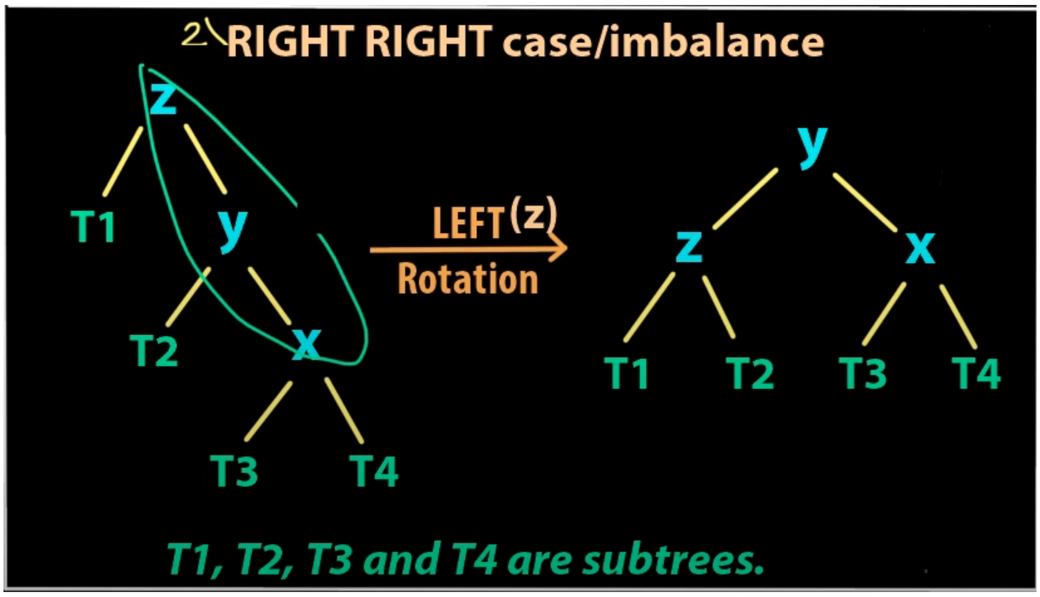
```
TN LLRotate(TN root)
{
    TN newRoot = root->left;
    TN T3 = newRoot->right;
    newRoot->right = root;
    newRoot->right->left = T3;

    return newRoot;
}
```

▼ RR imbalance

- imbalance is due to the left child of the left subtree of Z.





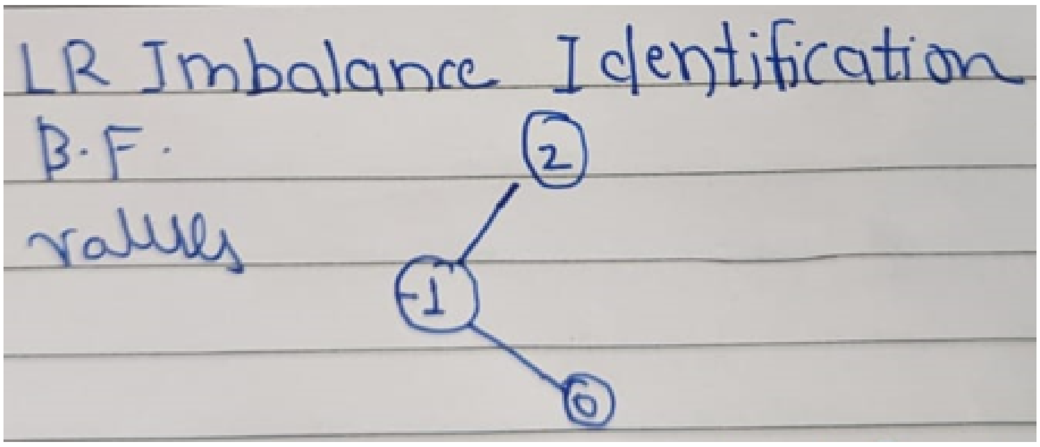
2\ RIGHT RIGHT case/imbalance
z
T1
y
T2
x
T3
T4
LEFT (z)
Rotation
y
z
x
T1
T2
T3
T4
T1, T2, T3 and T4 are subtrees.

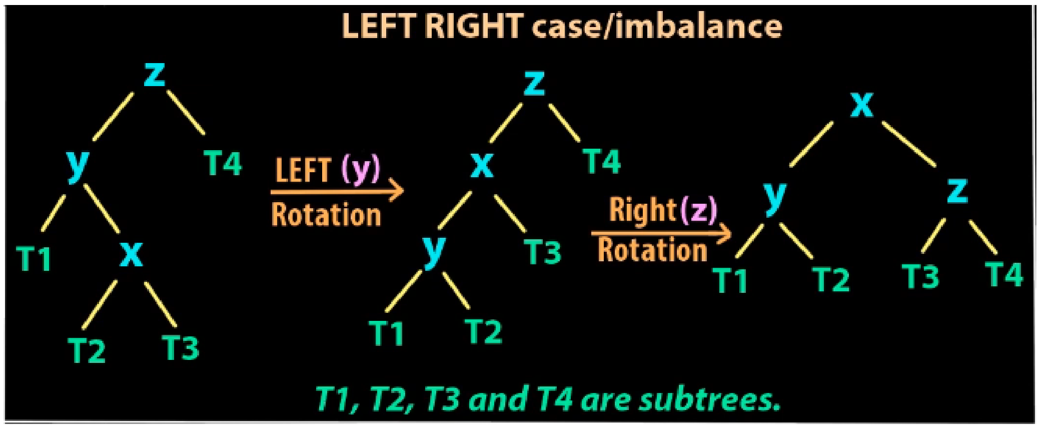
```
TN RRRotate(TN root)
{
    TN newRoot = root->right;
    TN T2 = newRoot->right->left;
    newRoot->left = root;
    newRoot->left->right = T2;

    return newRoot;
}
```

▼ LR imbalance

imbalance is due to greater height of the right node of the left subtree of Z.



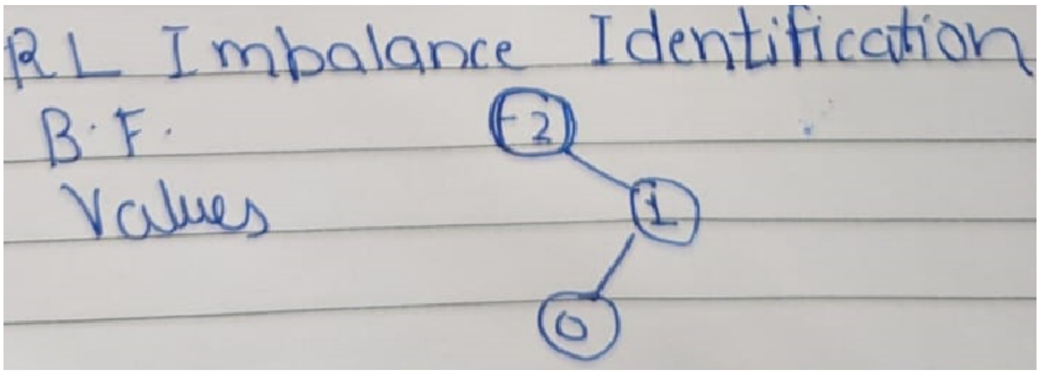


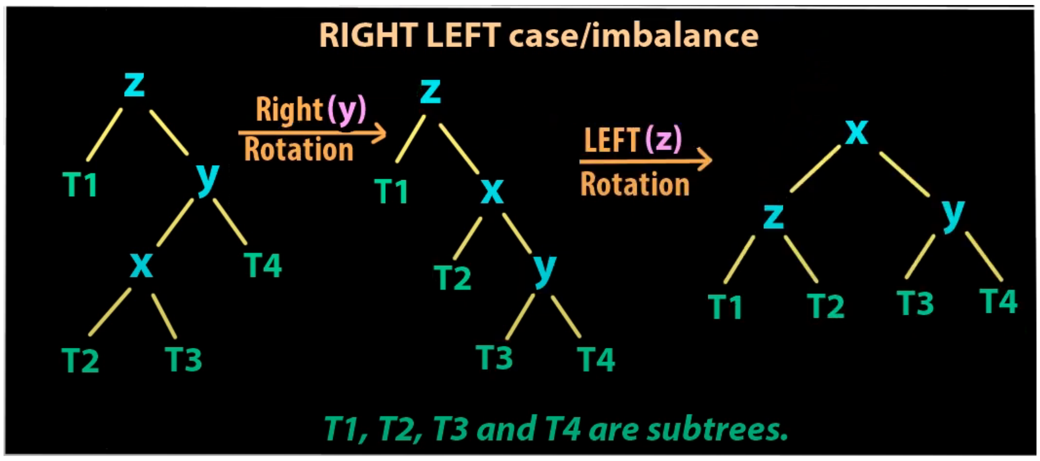
```
TN LRRotate(TN root)
{
    TN newRoot = root->left->right;
    TN T2 = newRoot->left;
    TN T3 = newRoot->right;
    newRoot->right = root;
    newRoot->left = root->left;
    newRoot->left->right = T2;
    newRoot->left->left = T3;

    return newRoot;
}
```

▼ RL imbalance

imbalance is due to greater height of the left node of the right subtree of Z.





```
TN RLRotate(TN root)
{
    TN newRoot = root->right->left;
    TN T2 = newRoot->left;
    TN T3 = newRoot->right;
    newRoot->left = root;
    newRoot->right = root->right;
    newRoot->left->right = T2;
    newRoot->right->left = T3;

    return newRoot;
}
```

▼ minimum spanning tree:

- **minimum weight spanning tree:** it is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.

- if there are n vertices in the graph, then each spanning tree has $n - 1$ edges.
- if each edge has a distinct weight then there will be only one, unique minimum spanning tree.
- Cycle Property: For any cycle C in the graph, if the weight of an edge e of C is larger than any of the individual weights of all other edges of C, then this edge cannot belong to an MST.
- Cut property: For any proper non-empty subset X of the vertices the lightest edge with exactly one end points in X belongs to MST.
  $\rightarrow$what is a cut? set of edges which, when all of them are removed, divides the graph into two disjoint graph.
- two algorithms to solve:

▼ Kruskal's Algorithm

  - In Kruskal's algorithm, sort all edges of the given graph in non-decreasing order.
  - Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If the cycle is not formed, include this edge. Else, discard it.
  - Repeat the procedure until there are $V - 1$ edges in the spanning tree.
  - We detect cycles using Union-Find algorithm.
  - Overall Time Complexity: $O(E \cdot log_2(E))$ or $O(E \cdot log_2(V))$

▼ Prim's Algorithm

Prim's algorithm starts with the single node and explore all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

> **Method:**
>
> 1. Initialize a tree with a single vertex, chosen arbitrarily from the graph.
> 2. Grow the tree by one edge: Of the edges that connect the tree to vertices not yet in the tree, find the minimum-

> weight edge, and transfer it to the tree.
> 3. Repeat step 2 until all vertices are in the tree.

Time Complexity:

Data Structure	Time Complexity
Binary Heap	$O((E+V)logV))$
Fibonacci Heap	$O(E+VlogV)$
Array	$O(V^2)$
Adjacency List	$O((E+V)logV)$