**Data Structures**

[Linked List](#Linked_List)

[Trees](#Trees)

[Heaps](#Heap)

[Strings](#Strings)

[Hashmap](#Hashmap)

**Linked List**

**Sorted Merge:**

**Given two sorted linked lists merge them in place**.

struct node\* result = NULL;

  /\* Base cases \*/

  if (a == NULL)

     return(b);

  else if (b==NULL)

     return(a);

  /\* Pick either a or b, and recur \*/

  if (a->data <= b->data)

  {

     a->next = SortedMerge(a->next, b);

return a;

  }

  else

  {

     b->next = SortedMerge(a, b->next);

return b;

  }

**Floyd cycle detection proof:**  
Let x be the distance from the start of the loop. And y be the distance from loop start to point where 2 pointers meet and z be the remaining measurement of cycle.

Suppose fast pointer has run over ‘m’ cycles before meeting the slow pointer. ‘i’ is the distance travelled by slow pointer and ‘2i’ is covered by fast pointer.

We can write the equations as:

i = x + y

2i = x + m(y+z) + y

This gives us:

2\*(x + y) = x + m(y + z) + y

x = (m-1)(y+z) + z

Hence, if we keep slow pointer at x start moving it, fast pointer would have cycled ‘m-1’ times with meeting slow pointer right at loop beginning. The same can be used to find the point of loop start so as to remove the loop from the linked list.

/\* If loop exists \*/

    if (slow == fast)

    {

        slow = head;

        while (slow != fast->next)

        {

            slow = slow->next;

            fast = fast->next;

        }

        /\* since fast->next is the looping point \*/

        fast->next = NULL; /\* remove loop \*/

    }

To reverse a list after every ‘k’ nodes. Use recursion over complete list and loop for reversal of k nodes.

**Delete nodes which have a greater value on right side**

To delete nodes in linked list which has its larger element on right, reverse the linked list. Take max = head->data and start looping. If cur\_node > max then max = cur\_node else delete the node.

**To create a linked list in which children are 2\*i+1 and 2\*i+2**

Use queue and go for level order traversal while traversing through the list in parallel.

**Create BST from linked list**

Method 1:

* Get the middle of linked list, make it as root

Do

* root->left = createBST(0, mid-1); root->right = createBST(mid+1, n);

**Method 2:**

struct Node\* sortedListToBSTRecur(struct Node \*\*head\_ref, int n)

{

    /\* Base Case \*/

    if (n <= 0)

        return NULL;

    /\* Recursively construct the left subtree \*/

    struct Node \*left = sortedListToBSTRecur(head\_ref, n/2);

    /\* head\_ref now refers to middle node, make middle node as root of BST\*/

    struct Node \*root = \*head\_ref;

    // Set pointer to left subtree

    root->prev = left;

    /\* Change head pointer of Linked List for parent recursive calls \*/

    \*head\_ref = (\*head\_ref)->next;

    /\* Recursively construct the right subtree and link it with root

      The number of nodes in right subtree  is total nodes - nodes in

      left subtree - 1 (for root) \*/

    root->next = sortedListToBSTRecur(head\_ref, n-n/2-1);

    return root;

}

**To find a triplet in 2 list whose sum is equal to a given number:**

Let the lists be a, b and c. Sort ‘b’ in ascending order and ‘c’ in descending order . Start traversing a, while(a != NULL)

If(sum < requisite)

B=b->next

Else

C=c->next

**To flatten linked list having both right and down pointers. Recurse the merge by merging down lists of root and root->right**

Leaf Case: When the two pointer points to last sublist.

// The main function that flattens a given linked list

Node\* flatten (Node\* root)

{

    // Base cases

    if (root == NULL || root->right == NULL)

        return root;

    // Merge this list with the list on right side

    return merge( root, flatten(root->right) );

}

**Flatten a multilevel linked list**

The idea of solution is, we start from first level, process all nodes one by one, if a node has a child, then we append the child at the end of list, otherwise we don’t do anything. After the first level is processed, all next level nodes will be appended after first level. Same process is followed for the appended nodes.

1) Take "cur" pointer, which will point to head of the fist level of the list

2) Take "tail" pointer, which will point to end of the first level of the list

3) Repeat the below procedure while "curr" is not NULL.

I) if current node has a child then

a) append this new child list to the "tail"

tail->next = cur->child

b) find the last node of new child list and update "tail"

tmp = cur->child;

while (tmp->next != NULL)

tmp = tmp->next;

tail = tmp;

II) move to the next node. i.e. cur = cur->next

**Sort list of 0’s, 1’s and 2’s**

**Approach 1:** Count number of 0, 1, and 2 and fill them**.**

**Approach 2 :** Segregate lists of 0, 1 and 2 and then merge them.

**Reverse alternate nodes and append to the end of list**

**Approach 1:** Separate the 2 lists. Reverse the 2nd and append to first.

**Approach 2:** Use a single loop to carry out the task. Below is the solution

while (odd && odd->next)

    {

       // Store the next node in odd list

       struct node \*temp = odd->next->next;

       // Link the next even node at the beginning of even list

       odd->next->next = even;

       even = odd->next;

       // Remove the even node from middle

       odd->next = temp;

       // Move odd to the next odd node

       if (temp != NULL)

         odd = temp;

    }

**To clone a linked list containing the random pointers**

**Approach 1:**

In this, first store the next pointer of the original list in array and point them to the corresponding element of the new linked list. Also, point the random pointer of new list to point to paralleled node of original list. After that do :

clone->arbit = clone->arbit->arbit->next;

And restore the original list after looking into the array.

**Approach 2:**

Add the cloned list in interleaved original list. After that, do:

original->next->arbitrary = original->arbitrary->next;

and restore:

original->next = original->next->next;

copy->next = copy->next->next;

**Approach 3:**

Another approach could be to maintain a hashmap of addresses of original and duplicated list. Traverse through the original list. Search its arbitrary pointer in O(1) time and point the arbitrary pointer of duplicated node to its adjacent one in map.

**Make arbitrary pointer in list to point to next highest node**

Copy the next pointer to arbitrary pointer and run mergesort based on arbitrary pointer.

**To sort a linked list that is sorted alternating ascending and descending orders**

1. Separate two lists.

2. Reverse the one with descending order

3. Merge both lists.

**Rearrange linked list with nth coming after 1st, n-1th coming after 2 and so on:**

Divide the linked list in 2. Reverse the 2nd list and merge into 1st.

**Sort linked list which is already sorted on absolute values**

Given a linked list which is sorted based on absolute values. Sort the list based on actual values.

Input : 1 -> -2 -> -3 -> 4 -> -5

output: -5 -> -3 -> -2 -> 1 -> 4

Traverse through the list while a -ve node is encountered move it to front of list.

**To convert linked list in zigzag order wherein a>b<c>d<e>f..**

Input: 1->2->3->4

Output: 1->3->2->4

Approach 1: Do merge sort and swap alternate nodes.

Approach 2: Traverse through the list and maintain the order while checking either of ‘>’ or ‘<’ using a switch.

**To find decimal from linked list of binary:**

while (head != NULL)

{

// Multiply result by 2 and add

// head's data

res = (res << 1) + head->data;

// Move next

head = head->next;

}

Input : 1->0->0

Output : 4

**Find pair of a given sum in singly linked list:**

Approach 1 : For doubly linked list, it is easy as we maintain 2 pointers one from beginning and another from last while moving them forward and backward depending upon whether the current calculated sum is less or more than the requisite one. For singly linked list, we have to convert the list into XOR linked list so that we can traverse in backward direction too. After that, we can employ the same strategy as we used in case of doubly linked list.

Approach 2 : We can use recursion.

bool printPairs(Node<int>\*\* h1,Node<int>\* h2,int sum){

if(h2!=NULL){

bool ck = printPairs(h1,h2->getNext(),sum);

if(!ck || \*h1==h2)

return false;

while((\*h1)->getData()+h2->getData() <= sum ){

if((\*h1)->getData()+h2->getData()==sum)

cout<<"("<< (\*h1)->getData() <<", "<<h2->getData()<<")"<<" ";

\*h1 = (\*h1)->getNext();

if(\*h1==h2)

return false;

}

}

return true;

}

Subtract two numbers represented by linked list:

* Find the smaller list.
* Pad it with 0’s amounting to diff(size\_large\_list-size\_small\_list)
* Use recursion while keeping the borrow field as flag

**To flatten a linked list depth wise use stack.**

Merge K sorted linked lists

**Find longest length palindrome in linked list:**

Loop over the linked list and one-by-one keep reversing the linked list and after each reverse compare the reversed list with the remaining list. Whole list will be reversed at last, so reverse it again to get the original.

**Trees**

**Full Binary Tree**: A tree with either 0 or 2 children is called a full binary tree.

In FBT, number of leaf nodes = number of internal nodes + 1

**Complete binary tree**: All levels full except the last one

**Perfect Binary tree**: In PBT, all levels are full and each internal node has 2 children

**Degenerate/Pathological Tree**: A tree in which each internal node has 1 child. It generally behaves as a linked list.

**Balanced binary tree**: In BBT, maximum difference between the levels of nodes in left and right

**Threaded Binary Tree:**

Inorder traversal of a Binary tree is either be done using recursion or with the use of a auxiliary stack. The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists). Since the internal node needs to point to right child we can choose the leftmost child of its right subtree which will be its inorder successor.

[Sample tree:](#Sample_Tree)

10

/ \

6 12

/ \ / \

2 9 11 18

\ / \

6 16 20

**To delete a path < sum**

We have to delete the nodes in bottom-up manner and pass on the sum to the next iteration.

deleteSum(root, int sum) {

if(root->left == NULL && root->right == NULL) {

if(sum+root->data < k) {

delete root;

return true;

}

else

return false;

}

if(deleteSum(root->left, root->data+sum) && deleteSum(root->left, root->data+sum)) {

delete root;

return true;

}

return false;

}

Approach 2: Keep decrementing sum by the value == node data. when we reach at the end and sum is still greater than the leaf node, then delete it.

    if (root == NULL) return NULL;

    // Recur for left and right subtrees

    root->left = prune(root->left, sum - root->data);

    root->right = prune(root->right, sum - root->data);

    // If we reach leaf whose data is smaller than sum,

    // we delete the leaf.  An important thing to note

    // is a non-leaf node can become leaf when its

    // chilren are deleted.

    if (root->left==NULL && root->right==NULL)

    {

        if (root->data < sum)

        {

            free(root);

            return NULL;

        }

   }

    return root;

**To find deepest left leaf node in a tree:**

Do simple in-order traversal passing lvl+1 and m.

**Lowest Common Ancestor**

It finds the root which is deepest ancestor of 2 nodes. If both nodes < root, search for ancestor in left, else if > node, search in right.

    // If both n1 and n2 are smaller than root, then LCA lies in left

    if (root->data > n1 && root->data > n2)

        return lca(root->left, n1, n2);

    // If both n1 and n2 are greater than root, then LCA lies in right

    if (root->data < n1 && root->data < n2)

        return lca(root->right, n1, n2);

    return root;

**Approach 2** (If parent pointers are given) : Take first node and store all its ancestors in hashtable. Similarly do for second node. Now take second node ancestors 1-by-1 and check if it is present in parent’s map of first node.

**To print nodes which are at k distance from leaf**

int printKDistantfromLeaf(Node\* node, int k)

{

if(node==NULL)

return -1;

int x=1+printKDistantfromLeaf(node->left,k);

int y=1+printKDistantfromLeaf(node->right,k);

if(x==k || y==k)

printf("%d ",node->key);

}

**To print vertical order traversal**

**Approach 1**: Take a map of number and linked list. Do -1 for left subtree and +1 for right subtree and add (number, node) pair in map. At last, print the map

**Approach 2:**

Find min and max breadth distance from root. Then print

    // If this node is on the given line number

    if (hd == line\_no)

        cout << node->data << " ";

    // Recur for left and right subtrees

    printVerticalLine(node->left, line\_no, hd-1);

    printVerticalLine(node->right, line\_no, hd+1);

called by

for (int line\_no = min; line\_no <= max; line\_no++)

    {

        printVerticalLine(root, line\_no, 0);

}

**Approach 3**(Space Optimized) : Use DLL with each node pertaining to each vertical level.

       llnode.data = llnode.data + tnode.data;

        // Recursively process left subtree

        if (tnode.left != null)

        {

            if (llnode.prev == null)

            {

                llnode.prev = new LLNode(0);

                llnode.prev.next = llnode;

            }

            verticalSumDLLUtil(tnode.left, llnode.prev);

        }

        // Process right subtree

        if (tnode.right != null)

        {

            if (llnode.next == null)

            {

                llnode.next = new LLNode(0);

                llnode.next.prev = llnode;

            }

            verticalSumDLLUtil(tnode.right, llnode.next);

        }

**To print right view of binary tree**

Approach 1: Use level order traversal and print last node

Approach 2: Use recursion, as below. **Note that we are doing level order traversal in reverse order**

    // If this is the last Node of its level

    if (\*max\_level < level)

    {

        printf("%d\t", root->data);

        \*max\_level = level;

    }

    // Recur for right subtree first, then left subtree

    rightViewUtil(root->right, level+1, max\_level);

    rightViewUtil(root->left, level+1, max\_level);

**To print top view of binary tree:**

Do vertical traversal of tree and print the first node of vertical level.

**Max path sum in binary tree**

It is the path which carries the maximum exists in a complete path in a binary tree.

int maxPathSumUtil(struct Node \*root, int &res)

{

    // Base cases

    if (root==NULL) return 0;

    if (!root->left && !root->right) return root->data;

    // Find maximum sum in left and right subtree. Also

    // find maximum root to leaf sums in left and right

    // subtrees and store them in ls and rs

    int ls = maxPathSumUtil(root->left, res);

    int rs = maxPathSumUtil(root->right, res);

    // If both left and right children exist

    if (root->left && root->right)

    {

        // Update result if needed

        res = max(res, ls + rs + root->data);

        // Return maxium possible value for root being

        // on one side

        return max(ls, rs) + root->data;

    }

    // If any of the two children is empty, return

    // root sum for root being on one side

    return (!root->left)? rs + root->data:

                          ls + root->data;

**Check if a subtree exists in a tree**

An inorder and preorder/postorder uniquely identify a binary tree. Hence to check if S is a subtree of a tree T, store inorder and preorder of T in 2 arrays. Do same for S. Search inorder and preorder array of S In corresponding string array of T. If both searches are successful, then return true, else return false

**Convert BT to DLL**

Fix left and right pointers using inorder traversal.

void BinaryTree2DoubleLinkedList(node \*root, node \*\*head)

{

    // Base case

    if (root == NULL) return;

    // Initialize previously visited node as NULL. This is

    // static so that the same value is accessible in all recursive

    // calls

    static node\* prev = NULL;

    // Recursively convert left subtree

    BinaryTree2DoubleLinkedList(root->left, head);

    // Now convert this node

    if (prev == NULL)

        \*head = root;

    else

    {

        root->left = prev;

        prev->right = root;

    }

    prev = root;

    // Finally convert right subtree

    BinaryTree2DoubleLinkedList(root->right, head);

}

**To find the depth of node at odd level:**

// A recursive function to find depth of the deepest odd level leaf

int depthOfOddLeafUtil(Node \*root,int level)

{

    // Base Case

    if (root == NULL)

        return 0;

    // If this node is a leaf and its level is odd, return its level

    if (root->left==NULL && root->right==NULL && level&1)

        return level;

    // If not leaf, return the maximum value from left and right subtrees

    return max(depthOfOddLeafUtil(root->left, level+1),

               depthOfOddLeafUtil(root->right, level+1));

}

**To check if leaves are at same level**

Pass level+1 to next recursive call. Take a reference and set it to 0 initially. After that set it equal to level when a leaf node is encountered. After that compare level of each child with that leafLevel. If not equal, then return false, else return true.

**To print left view of binary tree**

Approach 1: Use level order and print first element.

Approach 2: Use inorder traversal, keeping track of level, Whenever you find a level max than the present max, print it because that will be the first:

    // If this is the first node of its level

    if (\*max\_level < level)

    {

        printf("%d\t", root->data);

        \*max\_level = level;

    }

    // Recur for left and right subtrees

    leftViewUtil(root->left, level+1, max\_level);

    leftViewUtil(root->right, level+1, max\_level);

**Diagonal Traversal/Sum of binary tree:**

Approach 1: using queue

while(!q.empty())

{

local = q.size();

sum = 0;

++level;

for(int i=0; i < local ; i++)

{

dummy = q.front();

ponder = dummy;

q.pop();

sum += dummy->element;

if(dummy->left != NULL) {

q.push(dummy->left);

ponder = dummy->left->right;

while(ponder) {

q.push(ponder);

ponder = ponder->right;

}

}

}

cout<<"Sum at level "<<level<<"# "<<sum<<endl;

}

**Approach 2**:

Use recursion similar to level order traversal, the only difference is that, we will increase the level while going left but keep it same while going right, as:

    // Store all nodes of same line together as a vector

    diagonalPrint[d].push\_back(root->data);

    // Increase the vertical distance if left child

    diagonalPrintUtil(root->left, d + 1, diagonalPrint);

    // Vertical distance remains same for right child

    diagonalPrintUtil(root->right, d, diagonalPrint);

**Bottom view of tree:**

**Approach 1** : Use level order traversal, passing -1 when we go left and +1 when we go right. Add elements to map as (sum, element\_in\_queue). traverse through the map and print last element in the queue of each map entry

**Approach 2**: Use Vertical order, print last node of each traversal.

**Perfect Binary Tree Specific Level Order Traversal**

For [this](#Sample_Tree) tree:

The traversal will be:

10 6 12 2 18 9 11 6 16

Do level order traversal and instead of popping one node, process 2 nodes at a time.

    while (!q.empty())

    {

       // Pop two items from queue

       first = q.front();

       q.pop();

       second = q.front();

       q.pop();

       // Print children of first and second in reverse order

       cout << " " << first->left->data << " " << second->right->data;

       cout << " " << first->right->data << " " << second->left->data;

       // If first and second have grandchildren, enqueue them

       // in reverse order

       if (first->left->left != NULL)

       {

           q.push(first->left);

           q.push(second->right);

           q.push(first->right);

           q.push(second->left);

       }

    }

**Find distance of the closest leaf from ‘k’.**

Do breadth first search and find map entry of ‘k’ it’s nearest will be -1 and +1 of that index.

Working Solution: Find all ancestors of given node and for each ancestor find closest leaf and also leaf from that root.

**To check if binary tree is complete or not**

A tree is complete if all nodes have either 0 or 2 children except the last level.

**Approach 1**: Do level order traversal, each level should have 2^level nodes. At last level half of the nodes popped from queue should have child while other shouldn’t

**Approach 2**: Count the number of nodes and use the following code:

    // If index assigned to current node is more than

    // number of nodes in tree, then tree is not complete

    if (index >= number\_nodes)

        return (false);

    // Recur for left and right subtrees

    return (isComplete(root->left, 2\*index + 1, number\_nodes) &&

            isComplete(root->right, 2\*index + 2, number\_nodes));

**To create a binary tree from an array who’s each entry represents its parent node**

std::vector<node> bt; // constructed binary tree

void BuildBinaryTree(const std::vector<int>& parent)  
{  
 bt.resize(parent.size());

for (int i = 0; i < parent.size(); ++i)  
 {  
 if (parent[i] == -1)  
 root = &bt[i];  
 else if (bt[parent[i]].left == nullptr)  
 bt[parent[i]].left = &bt[i];  
 else  
 bt[parent[i]].right = &bt[i];

bt[i].val = i;  
 }  
}

**To check if a tree is a mirror/symmetric tree**

**if** (root1 && root2 && root1->key == root2->key)

**return** isMirror(root1->left, root2->right) &&

               isMirror(root1->right, root2->left);

**To find minimum depth of tree**

Do level order traversal and break from queue loop when the first leaf node is encountered.

**Binary Trees**

**Insertion**

/\* If the tree is empty, return a new node \*/

**if** (node == NULL) **return** newNode(key);

    /\* Otherwise, recur down the tree \*/

**if** (key < node->key)

        node->left  = insert(node->left, key);

**else** **if** (key > node->key)

        node->right = insert(node->right, key);

    /\* return the (unchanged) node pointer \*/

**return** node;

**To delete a node (3 cases)**

**When the node is leaf node** - Simply delete the node

**When the node has one child** – Replace the node with its child and delete the child.

**When the node has both left and right child** : Find the inorder successor of the node, replace the node to be deleted with that node and delete that inorder successor.

**if** (root == NULL) **return** root;

    // If the key to be deleted is smaller than the root's key,

    // then it lies in left subtree

**if** (key < root->key)

        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater than the root's key,

    // then it lies in right subtree

**else** **if** (key > root->key)

        root->right = deleteNode(root->right, key);

    // if key is same as root's key, then This is the node

    // to be deleted

**else**

    {

        // node with only one child or no child

**if** (root->left == NULL)

        {

**struct** node \*temp = root->right;

**free**(root);

**return** temp;

        }

**else** **if** (root->right == NULL)

        {

**struct** node \*temp = root->left;

**free**(root);

**return** temp;

        }

        // node with two children: Get the inorder successor (smallest

        // in the right subtree)

**struct** node\* temp = minValueNode(root->right);

        // Copy the inorder successor's content to this node

        root->key = temp->key;

        // Delete the inorder successor

        root->right = deleteNode(root->right, temp->key);

**To Handle Duplicates in Binary Tree:**

Add one for field to the node struct wherein we will store the count (the number of times the node appears in tree). To delete we find it and decrement the count. To Add we check if node is there then increment the count else add afresh.

**To Find closest leaf node to the given node:**

Take current node as root and find the closest leaf node and store its distance. Then start from the root. If that given node is in left subtree of main root then find closest leaf node in right subtree else vice versa and then compare it with the distance obtained above

**Inorder Non-threaded Binary Tree Traversal without Recursion or Stack**

To do this do inorder traversal using loop. You should have one flag too that will tell whether to go left or right. Initially it will be false, so go to left and traverse it and set flag to true. Now come back to parent. since that flag is true, go to right and set flag to false.

**To check if sum of uncovered nodes is same as that of covered nodes**

Find the sum of uncovered nodes by going towards left->left until the only right is available, that will give the sum of left boundary. Do the similar for right boundary. Now do inorder traversal and find the total sum, if it is == 2\*sum\_of\_uncovered\_nodes, the sums are equal.

**To construct a binary tree from postorder traversal:**

Find the postorder traversal, the rightmost number is the root. Make that as root now find the transition part, one part of which is smaller than root and the other half is larger than root. Do the same for those 2 parts.

**To print root to leaf paths without recursion**

Use stack and. Go left->left pushing all entries in stack until NULL is encountered. When null is encountered print path and pop back node from stack, push its right, if !NULL and go left->left pushing again on stack for traversing tree. Stop when stack is empty

**To check if there is a edge whose removal will divide tree in 2 equal parts**

    // Compute sizes of left and right children

    int c = checkRec(root->left, n, res) + 1 +

            checkRec(root->right, n, res);

    // If required property is true for current node

    // set "res" as true

    if (c == n-c)

        res = true;

    // Return size

    return c;

**Clone a binary tree with random pointers:**

The solution is similar to the linked list in which we store the mapping of original node to clone node. After we have created a tree with the standard method of inorder traversal. We can use map to modify random pointers of clone nodes

**Create a tree from inorder and postorder traversal**

The end node of postorder traversal is the root. Hence, make that as root and search for the index in inorder. Recursively pass the start and end index of inorder traversal to the recursion tree and keep picking the last node as root

/\* Pick current node from Preorder traversal using

postIndex and decrement postIndex \*/

Node \*node = newNode(post[\*pIndex]);

(\*pIndex)--;

/\* If this node has no children then return \*/

if (inStrt == inEnd)

return node;

/\* Else find the index of this node in Inorder

traversal \*/

int iIndex = search(in, inStrt, inEnd, node->data);

/\* Using index in Inorder traversal, construct left and

right subtress \*/

node->left = buildUtil(in, post, inStrt, iIndex-1, pIndex);

node->right= buildUtil(in, post, iIndex+1, inEnd, pIndex);

**To print cousins of a node**

**Approach 1**: Use recursion, pass level-1 to each call. When level = 2, it means we are 1 level above. So check if the searched node is either left or its right. If it is, return else print its left and right children

**Approach 2**: Use queue for level order traversal. While traversing before pushing left and right check if any of left or right of given node is that node do not push in queue.

**To print Maximum Consecutive Increasing Path Length**

At each node we need information of its parent node, if current node has value one more than its parent node then it makes a consecutive path, at each node we will compare node’s value with its parent value and update the longest consecutive path accordingly.

For getting the value of parent node, we will pass the (node\_value + 1) as an argument to the recursive method and compare the node value with this argument value, if satisfies, update the current length of consecutive path otherwise reinitialize current path length by 1

// if root data has one more than its parent

    // then increase current length

    if (root->data == expected)

        curLength++;

    else

        curLength = 1;

    //  update the maximum by current length

    res = max(res, curLength);

    // recursively call left and right subtree with

    // expected value 1 more than root data

    longestConsecutiveUtil(root->left, curLength,

                           root->data + 1, res);

    longestConsecutiveUtil(root->right, curLength,

                           root->data + 1, res);

**Find if there is a pair in root to a leaf path with sum equals to root’s data**

**Approach 1 :** Do preorder traversal. if node != NULL, check if root\_sum – node\_value is there in hashmap, if its there then return true else insert that node in map. At the end of recursion remove that node.

if(node == NULL)

return;

if(node!= NULL) {

// check in map for (root\_sum – node\_value) if found then ok else insert it in map

}

recurse(left);

recurse(right);

// remove node from map

**Find inorder successor:**

If node->right != NULL, then successor is the minimum in right subtree. Otherwise start from root and set root as successor when going left

**Find all k sum paths in a tree, wherein the starting node needn’t necessarily be the root node and last node the leaf node**

Do normal preorder traversal pushing each node in vector. After each recursion track back the vector, looping through size -1 -> 0 and print if sum of elements leads to k. At the end of recursion remove the node.

**Segment Tree:**

Segment Trees is a Tree data structure for storing intervals, or segments, It allows querying which of the stored segments contain a given point. It is, in principle, a static structure; that is, its content cannot be modified once the structure is built. It only uses O(N lg(N)) storage.

The leaves in segment tree are elements of input array while the internal node represents a range of value. The problems like : finding the maximum in a given range, finding the minimum in a given range can be solved in O(logn) time using segment tree in which the internal node will store the result.



Segment trees are used to solve range minimum queries problem in O(logn) which otherwise would take O(n) time to compute.

**To find sum of nodes at k-th level represented in the form of string as** “"(0(5(6()())(4()(9()())))(7(1()())(3()())))"

Start from left, when ‘(‘ is encountered increment the level, when ‘)’ is encountered decrement the level. When level is same as needed add that to sum which is initially initialized to 0.

To convert Sorted DLL to Tree

Approach 1 O(nlogn) : Use array like approach, take the middle, make it root and recur for its left and right

Approach 2 O(n):

/\* The main function that constructs balanced BST and returns root of it.

head\_ref --> Pointer to pointer to head node of Doubly linked list

n --> No. of nodes in the Doubly Linked List \*/

struct Node\* sortedListToBSTRecur(struct Node \*\*head\_ref, int n)

{

/\* Base Case \*/

if (n <= 0)

return NULL;

/\* Recursively construct the left subtree \*/

struct Node \*left = sortedListToBSTRecur(head\_ref, n/2);

/\* head\_ref now refers to middle node, make middle node as root of BST\*/

struct Node \*root = \*head\_ref;

// Set pointer to left subtree

root->prev = left;

/\* Change head pointer of Linked List for parent recursive calls \*/

\*head\_ref = (\*head\_ref)->next;

/\* Recursively construct the right subtree and link it with root

The number of nodes in right subtree is total nodes - nodes in

left subtree - 1 (for root) \*/

root->next = sortedListToBSTRecur(head\_ref, n-n/2-1);

return root;

}

**Red-Black Trees**

RB are the balanced binary trees which guarantees height of O(logn)

Invariants:

* Each node is either red or black.
* Root is black.
* No 2 reds in a row.
* Every root->NULL path has same number of black nodes.

A chain of 3 cannot be a red-black tree.

Every red-black tree has height of < 2\*log(n+1)

To make the tree balanced, we do left and right rotations.

**Insertion in RB-tree**

We insert in tree just like a normal BST. If an invariant is destroyed then do recoloring and/or rotations.

**Add new node as red** because making it black will destroy the equal-black-nodes in root->leaf path.

In recoloring, if 2 child are of red color while their parent is black. Recolor that parent to red and convert two red’s to black.

Do check: splay trees

**HEAP**

Heap is a data structure which is represented similar to a binary tree with each node having either 0, 1 or 2 children. We can have either min-heap or max-heap. In min-heap, we have minimum element at the top while in max-heap, we have maximum element at top. Mostly, Heap is implemented using Array as the data container.

Supported Operations:

* Extract Min or Max - O(logn)
* Insertion - O(logn)
* Delete - O(logn)
* Creation of Heap - O(nlogn)

**To Extract Min from min-heap**

> Move last leaf to new root

> Delete last leaf

> Heapify

**Median Maintenance:**

Use two heaps - low and high

Low heap will contain small half of integers while high heap will have large half integers. Median of the added integers will be either average of max of low heap and min of high heap in case number of elements are equal in both heaps or it will be max of low heap or min of high heap otherwise.

**Hash Table**

A kind of table that can be used data that employ association.

Some applications:

> De-duplication of data – Either read from file or from stream.

> 2-Sum problem – To find couple whose sum is equal to a given sum. For each x, look for sum – x in hash table.

> In symbol table.

> To block network traffic - By looking into a hash table of black list ip’s

> In general, Use hash table to avoid exploring any configuration more than once. One example is configuration of chess board.

Hash function takes as input key and spit out a position in the array.

or A|h(x)| where h(x) is a hash function.

**Collision**: When 2 entries map to the same value, collision results.

**Resolution**:

**Chaining**: We link colliding entries and put them in same bucket.

**Open Addressing**: No multiple entries. When we find a collision, probe the table for next free entry and insert the element in that.

**Double chaining**: Use both independently. If one hash function generates collision, use second hash function.

Open Hashing (Separate Chaining): In open hashing, keys are stored in linked lists attached to cells of a hash table.

Closed Hashing (Open Addressing): In closed hashing, all keys are stored in the hash table itself without the use of linked lists.

**Cuckoo Hashing**:

A hashing technique in which we maintain 2 tables instead of 1 so that when conflict arises then key from table 1 can be shifted to table 2. For the same we use 2 hash functions, if we find a conflict in

**2-Sum problem when we have to find sums which lie in a given range:**  
> Store the elements in a set.

> for each element, find lower and upper bound in both left and right.

> calculate upper-lower for both left and right and sum them henceforth.

**Strings**

**Find duplicates in a string**

**Check if 1 string is rotation of another**

**Remove all chars from str1 which are in str2**

Create an array of char as char a[256], set a[<c>] for all observed in string1 and process the same.

**Find the minimum windows in string which matches a given string**

C Maintain 2 hashmaps, one for main\_string and another for pattern. Also, take a variable count = 0 which will maintain the number of matches. Then traverse over the pattern and increment and fill its respective histogram. After that traverse over the main\_string filling its histogram and incrementing the count at character char when

main\_string[char\_count] < pattern[char\_count]

When count reaches the pattern.length(), then start traversing the string from left with condition

while (!String.containsKey(char) || String.get(sc) > pattern.get(sc))

**Permutations of a string**

void permute(char \*a, int l, int r)

{

int i;

if (l == r)

printf("%s\n", a);

else

{

for (i = l; i <= r; i++)

{

swap((a+l), (a+i));

permute(a, l+1, r);

swap((a+l), (a+i)); //backtrack

}

}

}

**Lexicographic rank of String**

Use permutation, find count of all by considering the starting point as chars < first char. Then fix the first char and repeat the process.

e.g for STRING, it will be 4\*5! + 4\*4! + 3\*3! + 1\*2! + 1\*1! + 0\*0! = 597

597 + 1 = 598

**Longest Consecutive Subsequence**

To find longest consecutive subsequence wherein the elements in subsequence are dispersed.

**Approach 1 O(nlogn)**: Sort the array and find the same.

**Approach 2 O(n)**: Use hashing – Insert all elements in hash. Re-traverse the array and look for the element first in sequence then search for subsequent elements in hash.

for (int i=0; i<n; i++)

{

// if current element is the starting

// element of a sequence

if (S.find(arr[i]-1) == S.end())

{

// Then check for next elements in the

// sequence

int j = arr[i];

while (S.find(j) != S.end())

j++;

// update optimal length if this length

// is more

ans = max(ans, j - arr[i]);

}

}

**Check for Palindrome after every character replacement**

Traverse through the string and store unequal indices in set/map. For every query, if both incoming indices are not equal then simply update the string but if both incoming indices match then remove them from set and count the number of elements in set.

**Arrays**

Find whether an array is subset of another array

**Approach 1**: Sort both arrays and compare like merge sort.

**Approach 2**: Use hashing, Store array 1 in hash and compare array 2 members with hash elements.

**Find largest subarray with maximum number of 0’s and 1’s**

Create another temp array which will store the difference of number of 1’s and 0’s at a given step. Now, scan from left to right and calculate and store the same. After complete pass, we just need to find the two elements in temp array which are far apart. Max range could be –n to n. so you can make a 2n+1 element table and store in it the indices of the first and last time each number appears. From there, it's easy to find the longest range. Overall, this needs O(n) space and everything can be populated and searched in O(n) time.

Find if there is subarray with 0 sum

The approach will be the same as that of above. Create a hashtable and store in it the intermediate sum calculated at an index while we traverse the array from left to right. If a partial sum is already present in hashtable then answer is YES, else add that sum to hashtable.

Find first repeating element

Use hash

**Find largest subarray with contiguous elements**

**Approach 1 O(n^2)** - A subarray has contiguous elements if and only if the difference between maximum and minimum elements in subarray is equal to the difference between last and first indexes of subarray. So the idea is to keep track of minimum and maximum element in every subarray.

**Approach 2 O(nlogn)**: Sort the array and check the sequence.

**Count distinct elements in every window of size k**

Use a hashmap, insert first k nodes in it. Find the distinct while inserting. After that slide the window 1-by-1 removing the left element and add/increment count right element. If the inserted element is new one, it is distinct.

**Find if an array can be divided in pairs whose sum is divisible by a given element k**

Traverse through array and store (remainder, frequency) in map. Now, traverse again and for each element. If its remainder is x such that 2\*x = k, then there should be even number of such elements. or if remainder is 0 then also even no. of elements, else number of occurrences of remainder must be equal to number of occurrences of k – remainder.

map<int, int> freq;

// Count occurrences of all remainders

for (int i = 0; i < n; i++)

freq[arr[i] % k]++;

// Traverse input array and use freq[] to decide

// if given array can be divided in pairs

for (int i = 0; i < n; i++)

{

// Remainder of current element

int rem = arr[i] % k;

// If remainder with current element divides

// k into two halves.

if (2\*rem == k)

{

// Then there must be even occurrences of

// such remainder

if (freq[rem] % 2 != 0)

return false;

}

// If remainder is 0, then there must be two

// elements with 0 remainder

else if (rem == 0)

{

if (freq[rem] & 1)

return false;

}

// Else number of occurrences of remainder

// must be equal to number of occurrences of

// k - remainder

else if (freq[rem] != freq[k - rem])

return false;

}

return true;

**Find the missing numbers in an unsorted array of size 'n' in which numbers are ranged from 1 to n**

To find the same use that same array.

Decrement count of each element by 1 and then for every encountered number add 'n' to that element's original position after modulus. Modulus is used so that we get correct number if that number has already gone > n (Number already encountered) . At last divide that position by n.

void printfrequency(int arr[],int n)

{

for(int j =0;j<n;j++)

{

arr[j] = arr[j]-1;

}

for(int i = 0;i<n;i++ )

{

arr[arr[i]%n] = arr[arr[i]%n] + n;

}

for(int i =0; i<n;i++)

{

cout << i + 1<<" -> "<<arr[i]/n<<endl;

}

}nd pairs with given sum such that elements of pair are in different rows

# Create a hash with value as key and its position as index. Start from left top and search for the element - ( sum – current\_value ) if element is there in the hash, compare rows of both, if equal then skip else continue;

# **To find the smallest range in k sorted list, with each of size n**

# Use heap of k elements, find the min and find the difference and now insert the element from the list whose minimum was extracted. Find the difference and continue.