**Data Structures**

Linked List

Trees

Heaps

Strings

Hashmap

Arrays

Graphs

Algorithms

Dynamic Programming

**Linked List**

> To find the middle node of a linked list, use 2 pointers, 1 slow one and another faster one. Slow pointer make a single jump at a time while faster pointer two. Start both the pointers in a loop and when faster one reaches the end, return the slower pointer.

> The complete linked list deletion can be done using 2 pointers – cur and next

> Reversal of list iteratively just required use of 1 pointer – (LinkedList\*\* head), deduce cur and next from this.

Node\* rest = reverse(head->next);

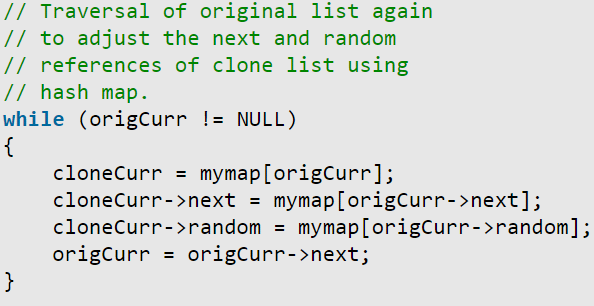
head->next->next = head;

head->next = NULL;

> To check if linked list is palindrome or not, pass head as single and double pointer. Use double pointer to move right after end of each recursive call and compare data of current node in recursive call and node of double pointer.

> To create a copy of list and correct its arbitrary pointers, there are 3 methods –

>> hashing: In this first create the duplicate list and store the corresponding nodes of original and duplicate list in map. You can create a duplicate node and put it alongside the original node in a map



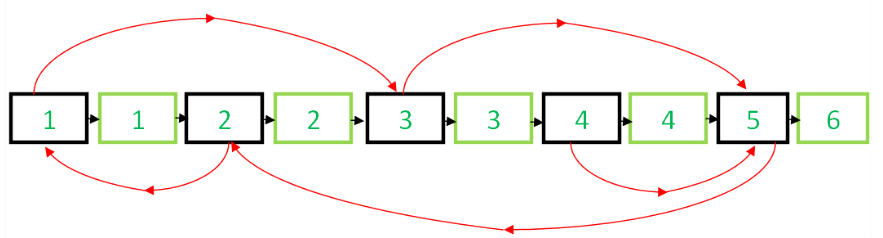
>> Insert ‘copy\_n’ nodes in between ‘original\_n’ and ‘original\_n+1’ node;

original->next->arbitrary = original->arbitrary->next; /\*TRAVERSE TWO NODES\*/

Restore back the nodes:

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

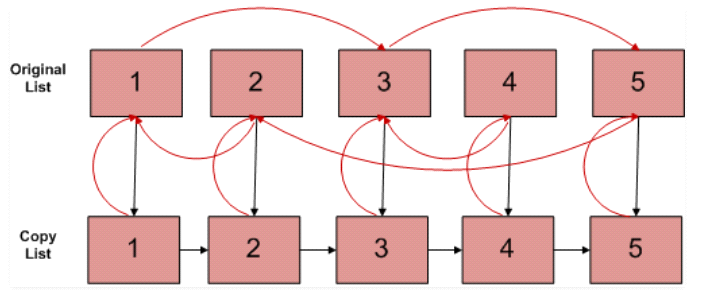
copy->next = copy->next->next;



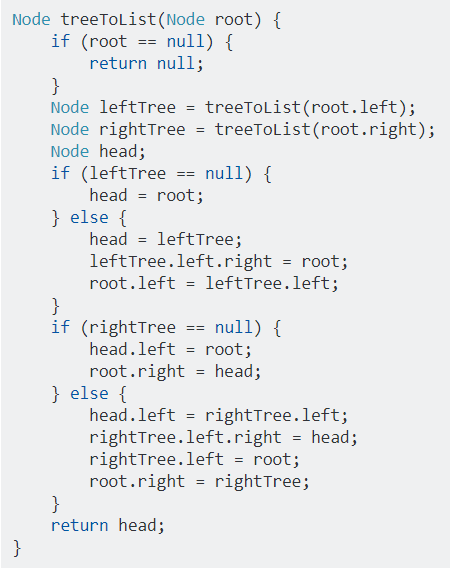
>> Last method is to take backup of next pointers of original list and then points its next to corresponding nodes in duplicate list, pointing arbit pointers of duplicate list pointing to corresponding nodes in original list and then set the arbit pointers of duplicate list as:

copy\_list\_node->arbit = orig\_list\_node->arbit->arbit->next;

copy\_list\_node = copy\_list\_node->next;



> To convert a BT to Circular DLL, first start with simple tree with 3 nodes – 1 root and 2 nodes signifying left and right nodes and then generalizes it for whole tree.



> Intersection point of 2 lists. There are 4 methods:

>> Count number of nodes in 1st and 2nd list. Traverse 1st list for abs(1st\_list\_len – 2nd\_list\_len) and now start comparing pointers. If any of those matches, that’s the intersection point

>> Traverse 1st list and put all addresses in a hashmap. Now, traverse the 2nd list and check if any of the node is in hashmap. If found, you’ve got the intersection point, else both lists don’t intersect.

>> Let the intersection length of 1st list is x1 and of 2nd is x2 and length after intersection is y. So, this method relies on fact that:

len(traverse 1st (x1+y) + intersection length of 2nd (x2)) = len(traverse 2nd (x2+y) + intersection length of 2nd (x2))

1. If curr1 != null curr1 = curr1->next else curr1 = head2

2. If curr2 != null curr2 = curr2->next else curr2 = head1

3. Repeat the above steps while curr1 is not equal to curr2

>> Create loop in 1st list and check for loop in 2nd list. It’s just like finding the loop in a list

> To rotate the linked list clockwise to the right by K places.

1. Change the next of the kth node to NULL.

2. Change the next of the last node to the previous head node.

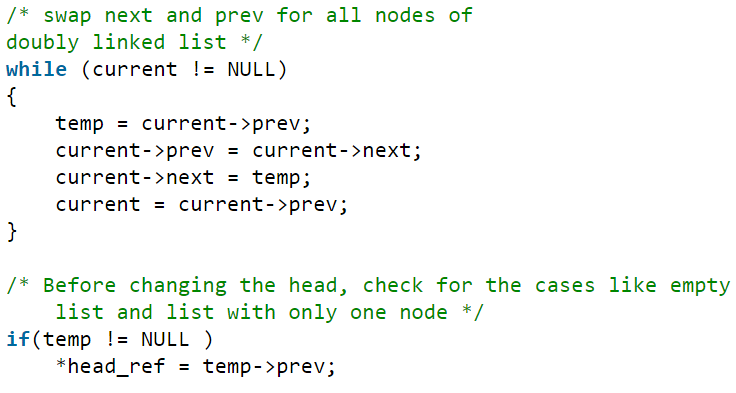
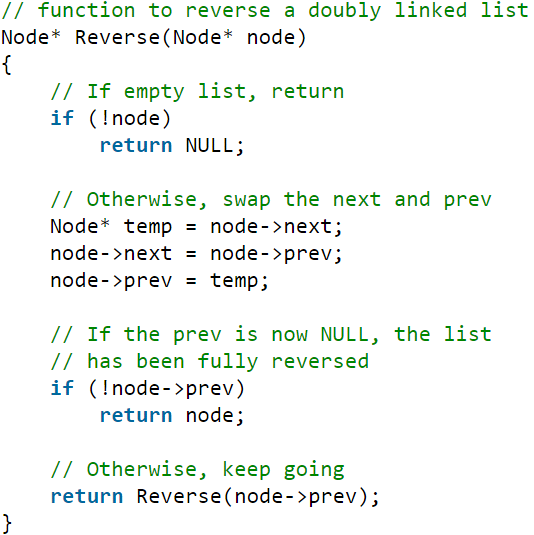
3. Change the head to (k+1)th node.

> To remove duplicates in unsorted linked list

>> Use merge sort and remove duplicates

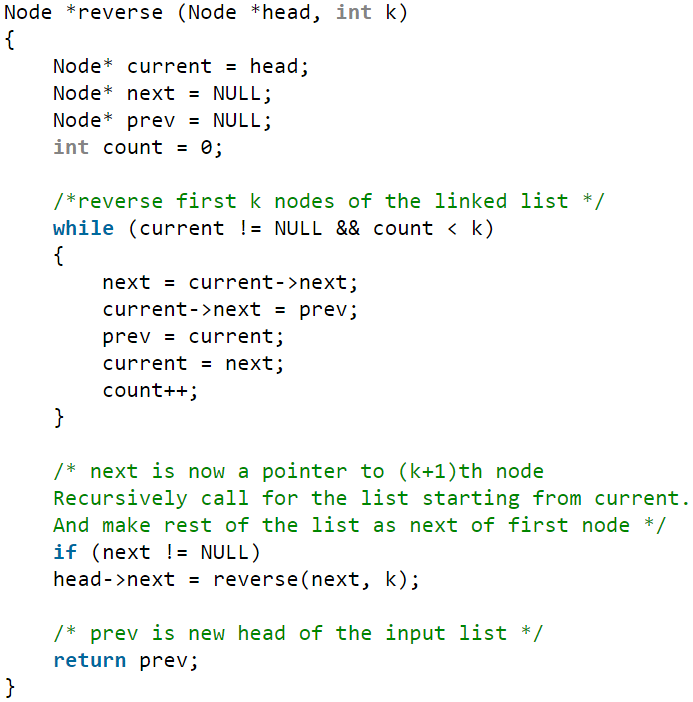
>> Use hashmap. While traversal, if node’s data is in hash, delete that node else add data of that node to hash map.

> Reversal of DLL. Two methods- Iterative and Recursive. In recursive method we first swap ‘next’ and ‘prev’ pointers of node and pass node->prev to next recursive call, while in iterative method, we use pointers

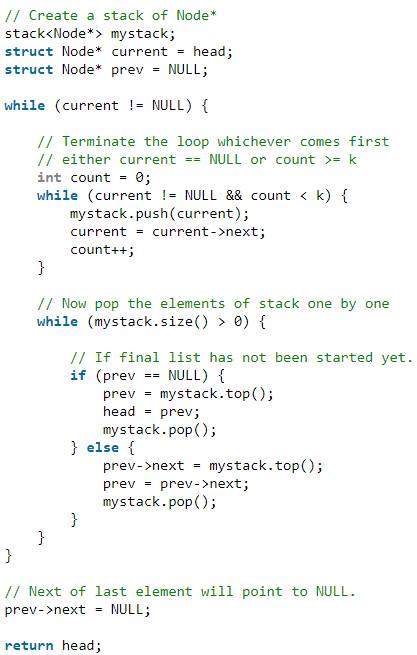


> Reverse every ‘k’ nodes in a linked list:

Use iteration to reverse the linked list and recursion to complete the flow.



> We can use a stack too



> > Another method to reverse the linked list block-wise is to use deque - Insert first 'k'elements (addresses) of linked list in deque. Pick the nodes from back 1-by-1 and link them. But this will consume extra memory.

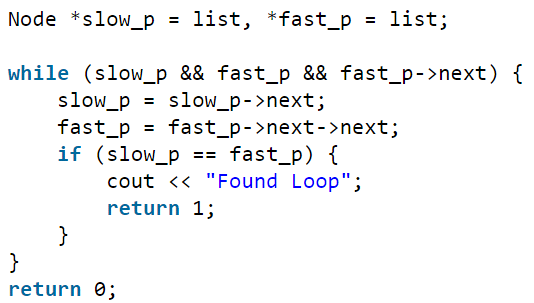
> To create a sorted linked list from a binary tree

>> Go for inorder traversal of tree and use insertion sort.

> For alternate reverse, move forward for ‘k’ elements before making the recursive call. This method is similar to the method in which we reverse the linked list in blocks.

> Detect loop in linked list:

There are two methods – use a hashmap or use floyd’s cycle detection algorithm



**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;**

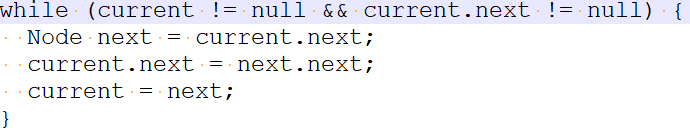
**}**

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

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

**}**

> Alternate split – Split the linked list in 2



> MergeSort on LinkedList

Pseudocode:

recur(&l) {

Split the linked list in 2 using hare and tortoise algorithm. Name those 2 parts, say ‘a’ and ‘b’

recur(&a)

recur(&b)

SortedMerge(&a, &b)

}

> Flatten a linked list with right and down pointers

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;

res = flatten(root->right);

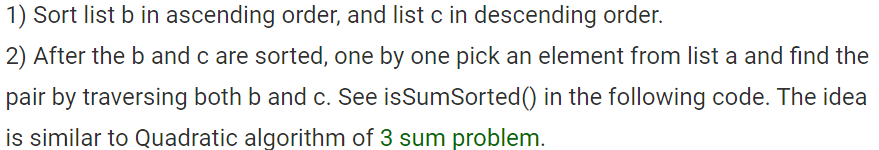
// Merge this list with the list on right side

return merge( root, res);

}

Another method could be to append all down list to the end and then call merge sort on that list.

> Find triplet in 3 linked list so that a node from 1 list (a) is sum of 2 nodes from other 2 lists (b & c):



**Why we need to move fast pointer by factor of 2:**  
Let us suppose the length of the list which does not contain the loop be s, length of the loop be t and the ratio of fast\_pointer\_speed to slow\_pointer\_speed be k.

Let the two pointers meet at a distance j from the start of the loop.

So, the distance slow pointer travels = s + j. Distance the fast pointer travels = s + j + m \* t(where m is the number of times the fast pointer has completed the loop). But, the fast pointer would also have traveled a distance k \* (s + j) (k times the distance of the slow pointer).

Therefore, we get k \* (s + j) = s + j + m \* t.

s + j = (m / k-1)t.

Hence, from the above equation, length the slow pointer travels is an integer multiple of the loop length.

For greatest efficiency , (m / k-1) = 1 (the slow pointer shouldn't have traveled the loop more than once.)

therefore , m = k - 1 => k = m + 1

Since m is the no.of times the fast pointer has completed the loop , m >= 1 . For greatest efficiency , m = 1.

therefore k = 2.

-if we take a value of k > 2 , more the distance the two pointers would have to travel.

> 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;

  }

> **Sorted merge of doubly linked list**

Node\* merge(Node\* first, Node\* second)

{

if (!first) return second;

if (!second) return first;

// Pick the smaller value and adjust the links

if (first->data < second->data) {

first->next = merge(first->next, second);

first->next->prev = first;

first->prev = NULL;

return first;

}

else {

second->next = merge(first, second->next);

second->next->prev = second;

second->prev = NULL;

return second;

}

}

> **Rearrange a linked list in to alternate first and last element**

>> Separate and then reverse second half of list and merge it with 1st half.

>> Use deque, insert complete data in deque.. then update data of linked list picking 1 from front and then 1 from back of deque.

> To alternately split with recursion,

>> /\* Pass first and second node to the recursive function \*/

void moveNode(Node\* a, Node\* b) {

if (b == NULL || a == NULL) return;

if (a->next != NULL) a->next = a->next->next;

if (b->next != NULL) b->next = b->next->next;

moveNode(a->next, b->next);

}

**> Sort the linked list in the order of elements appearing in a given reference array**

>> Use hashtable, store frequencies of each element present in linked list in map. Traverse the reference array, check and fill the data of linked list as per frequency appears in the map. Running time O(N)

>> Traverse the array and for each entry prepend the linked list elements in front of linked list. Running time (Max): O(N\*M)

**> To Extract Leaves of a Binary Tree in a Doubly Linked List**

Have a recursive function returning node pointer. Go for reverse in-order traversal. Perform DLL operation in case left and right of node is NULL and return NULL for them as in:

Node\* extractLeafList(Node \*root, Node \*\*head\_ref)

{

if (root == NULL) return NULL;

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

root->right = \*head\_ref;

// Change left pointer of previous head

if (\*head\_ref != NULL)

(\*head\_ref)->left = root;

// Change head of linked list

\*head\_ref = root;

return root; // Return new root

}

// Recur for right and left subtrees

root->right = extractLeafList(root->right, head\_ref);

root->left = extractLeafList(root->left, head\_ref);

return root;

}

> **First non-repeating character**: 3 approaches

>> Using an array: An array of size int[256], traverse once to fill all entries. Traverse once again to find first with frequency one. 2 traversals – 2 main sequence. Complexity O(n). 1 traversal.

>> To traverse the array once, have a struct which stores the frequency and index of last occurrence. Now traverse the count array and find the character which has least value of first occurrence and frequency value as unity. 2 traversals – 1 main sequence, 1 count array

>> Using queue: Use a hashmap to store <char, address> for a given node. For a new character, add it to queue, for an old character pick that character and put it in back. After all characters are done, front node is the one with least frequency. 1 traversal – 1 main sequence. Complexity O(n). 2 data structures.

> **Construct a Maximum Sum Linked List out of two Sorted Linked Lists having some Common nodes**

The idea is that you have to pick a sublist whose sum is maximum till the common node and assign that sublist to the tail of previous calculated list. So, start with 1st node, since both lists are sorted, traverse both lists, storing the sum, take the list whose sum is max. and then do prev->next = current; where prev is tail pointed of previous list and current is the head of sublist with max sum.

**> To find a single element which is extra in one of the lists. XOR both the linked lists.**

> **To Delete continuous nodes with sum K from a given linked list**

Traverse the given linked list. During traversal store the sum of the node value till that node with the reference of the current node in an unordered\_map.

If there is Node with value (sum – K) present in the unordered\_map then delete all the nodes from the node corresponding to value (sum – K) stored in map to the current node and update the sum as 0. If there is no Node with value (sum – K) present in the unordered\_map, then stored the current sum with node in the map.

**Use the above strategy for any sequence where continuous elements are involved in Qn.**

**> To create a 2D matrix of singly linked lists**

if (i > n - 1 || j > m - 1)

return NULL;

Node\* temp = new Node();

temp->data = arr[i][j];

temp->right = construct(arr, i, j + 1, m, n);

temp->down = construct(arr, i + 1, j, m, n);

return temp;

**or**

// To create DLL from a 2D matrix

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

{

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

{

arr[i][j].up = (i>0) ? arr[i-1][j] : null;

arr[i][j].left = (j>0) ? arr[i][j-1] : null;

arr[i][j].down = (i+1<m) ? arr[i+1][j] : null;

arr[i][j].right = (j+1<n) ? arr[i][j+1] : null;

}

}

**> To find next greater element**

Use stack and traverse the array from last to first

> **Point to next higher value node in a linked list with an arbitrary pointer**

Copy next pointer to arbitrary pointer and sort on arbitrary pointer

> **Sort a linked list that is sorted alternating ascending and descending orders**

Unlink the descending order part and reverse it. After this go for sorted merge of first part and the reversed part.

> **Create a zig-zag list of form 1 < 2 > 4 < 6 > 8 < 10…from list 1 -> 2 -> 6 -> 4 -> 10 -> 8..**

Scan through the list and switch the data based on a flag which determines which order we want to place i.e. ‘>’ or ‘<’ which will be switched for each iteration.

**> Merge two sorted linked lists such that merged list is in reverse order**

Two approaches:

>> Merge the 2 sorted linked lists and reverse the resultant

>> Merge at head in single traversal so that next greater element is added at head.

>>> res = NULL.

>>> while (a != NULL and b != NULL)

>>> Find the smaller of two (Current 'a' and 'b')

>>> Insert the smaller value node at the front of result.

>>> Move ahead in the list of smaller node.

> **Calculate decimal from binary number represented by linked list.**

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

**> To find max length palindrome**  
>> Use approach of iterative reversal of linked list

int maxPalindrome(Node \*head)

{

int result = 0;

Node \*prev = NULL, \*curr = head;

// loop till the end of the linked list

while (curr)

{

// The sublist from head to current

// reversed.

Node \*next = curr->next;

curr->next = prev;

// check for odd length palindrome

// by finding longest common list elements

// beginning from prev and from next (We

// exclude curr)

result = max(result,

2\*countCommon(prev, next)+1);

// check for even length palindrome

// by finding longest common list elements

// beginning from curr and from next

result = max(result,

2\*countCommon(curr, next));

// update prev and curr for next iteration

prev = curr;

curr = next;

}

return result;

}

**To subtract one number from another, represented by linked list**

>> Steps:

1. Find length of lists and append 0s to the front of the list with lesser length.
2. Find larger list using normal element-by-element comparison.
3. Now, recurse go to the end, do d1

Do, dif = (d1 – d2). If dif < 0, set borrow = true and do (d1+10). Thereafter, create a node with data (d1 – d2) and do current->next = prev; where prev is the recent node returned by recursion tree.

**To move all keys (specific items to end of list)**

>> Use 2 pointers pCrawl and pKey. Both will move next, with pKey not moving in case of data ==key

During traversal, if pCrawl != pKey and pCrawl data is also != pKey it means we have found the key so swap the data and do pkey = pkey->next.

>> Create another list of keys and move them to the back

>> Move these keys one-by-one to the tail by having a tail pointer.

**To unfold a linked list**

**A linked list L0 -> L1 -> L2 -> ….. -> LN can be folded as L0 -> LN -> L1 -> LN – 1 -> L2 -> …..**

**Given a folded linked list, the task is to unfold and print the original linked list**

>> Pluck out 2nd half, reversing it and append it to 1st half.

>> Use below code (**recursive**)

private void unfold(Node node) {

if (node == null) return;

// the number of nodes is odd

if (node.next == null) {

head = tail = node;

return;

}

// the number of nodes is even

else if (node.next.next == null) {

head = node;

tail = node.next;

return;

}

// Storing next node in temp pointer

Node temp = node.next;

unfold(node.next.next);

// Connecting first node to head and mark it as a new head

node.next = head;

head = node;

// Connecting tail to second node (temp) and mark it as a new tail

tail.next = temp;

tail = temp;

tail.next = null;

}

**To find intersection of 2 lists**

>> Sort both the lists.

>> Finding the intersection is easy. Just do linear scan incrementing ‘a’ if a->data < b->data or vice-versa and add to intersection list if a->data == b->data

>> Finding the union is easy too. Just do a linear scan of 2, insert entries of both ‘a’ and ‘b’ are different in case they are equal, add single entry and increment both.

**To partition list around a given number**

Three approaches:

>> Modifying link:

>>> Maintain 3 pointers, cur, head and tail. Initially point all 3 to head. Whenever an element < partition element is encountered, add that to head. If greater add to tail. Store next = cur->next; before making the changes. Then do cur = next;

Node \*curr = head;

while (curr != NULL) {

struct Node \*next = curr->next;

if (curr->data < x) {

/\* Insert node at head. \*/

curr->next = head;

head = curr;

}

else // Append to the list of greater values {

/\* Insert node at tail. \*/

tail->next = curr;

tail = curr;

}

curr = next;

}

**Above method won’t preserve the relative order. Use below method for stable sort:**

>>> Divide and create 3 lists. One will contain elements smaller than the partition element, second will have elements equal to that numbers and last one will contain element greater than the partition element.

**> Add two polynomials**

Think of them as sorted list and traverse them parallel storing the result in a new list

**> To multiply 2 lists:**

Extract the integers from 2 lists, multiply them like numbers and return the multiplication result.

**> To multiply 2 polynomials represented by linked lists:**

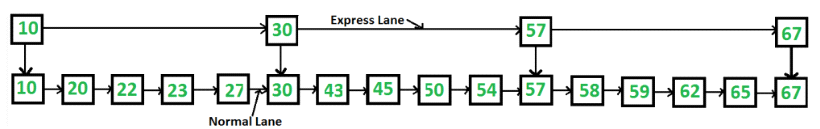
In nested loop, multiply two nodes and create a new node for resultant list storing multiplied coefficients and power in it. Keep track of head and tail. Head will point to node with least coefficient and tail to node with highest coefficient. At last, remove the duplicate nodes (nodes with the same power)

**> Skip lists**

**>** In general, it takes O(n) to search for a node in linked list. Can we do it in a faster way similar to what we can do with a binary search and tree and a sorted array in case list is sorted. The answer is yes, through skip lists.

In this we maintain 2 lanes, express lane and normal lane. 2 nodes in express lane traverse sqrt(n)

number of nodes if we divide the nodes uniformly. Overall, it takes O(sqrt(n)) complexity to search a node in skip list.



**> Given K sorted linked lists merge them in place with total elements as N**

**>> Approach 1** : Naïve method

Pick first 2 lists merge them. Pick the merge list and merge it with 3rd sorted list and so on.

T(N) = O (k^2\*N)

**>> Approach 2** : Using heap.

Create a heap with picking first element from each list. Recursively, do this till the heap is empty

>>> Remove the top of heap and add it to resultant list

>>> Add to heap the next node of the node which is removed from heap.

Time complexity - O(nklog k)

**>> Approach 3** : Using map

Iterate over all lists and store their data in map with count. If count > 1, add to new list and –count.

T(N) = Insertion – O(NlogN) + Retrieval = O(NlogN) = O(NklogNk) S(N) = O(N)

**>> Approach 4** : Divide and conquer.

Solve the problem for first k/2 and last k/2 list. Then you have 2 sorted lists. Then simply merge the lists.

Time complexity - O(nklog k)

**The advantage of this solution over heap based solution is that no extra space is required.**

**Divide and Conquer Solution:**

// The main function that takes an array of lists

// arr[0..last] and generates the sorted output

Node\* mergeKLists(Node\* arr[], int last)

{

    // repeat until only one list is left

    while (last != 0)

    {

        int i = 0, j = last;

        // (i, j) forms a pair

        while (i < j)

        {

            // merge List i with List j and store

            // merged list in List i.SortedMerge Method is defined above

            arr[i] = SortedMerge(arr[i], arr[j]);

            // consider next pair

            i++, j--;

            // If all pairs are merged, update last

            if (i >= j)

                last = j;

        }

    }

    return arr[0];

}

**> To find first non-repeating character in a stream**

Take a Queue and a char array of size 26 to store frequencies. For each stream element, increase its frequency in char array and add it to Queue. In a loop which is providing stream of characters, for each entry, have a loop that pops the element in front which are of frequency > 1 and print the first entry with freq\_count=1. if no element is left, it means there is no non-repeating element.

> **To sort a 'k' sorted DLL**

Use a min-heap of size 'k'

**> To find balanced node (whose left\_sum and right\_sum are equal) in linked list.**

Find the sum of all elements of linked list. Now traverse the linked list again, summing the list elements as you go along. At any time, if that partial sum is total\_sum/2 that node is balanced node.

**> To find nearest square number before and after a number:**

Before: floor(sqrt(number))^2, After: ceil(sqrt(number))^2

e.g. for 8, we have sqrt floor(2.812)^2 = 4 and square greater than 8 is ceil(2.82)^2 = 9

**> To add a digit to a number represented by a linked list**:

Don't reverse the linked list. Just find the last number in linked list which is < 9, let us call it lastptr. Add digit to last node, if it results in

carry start from lastptr uptill the last node and change the value (node->data+1)%10

> **Kadane's algorithm for linked list**

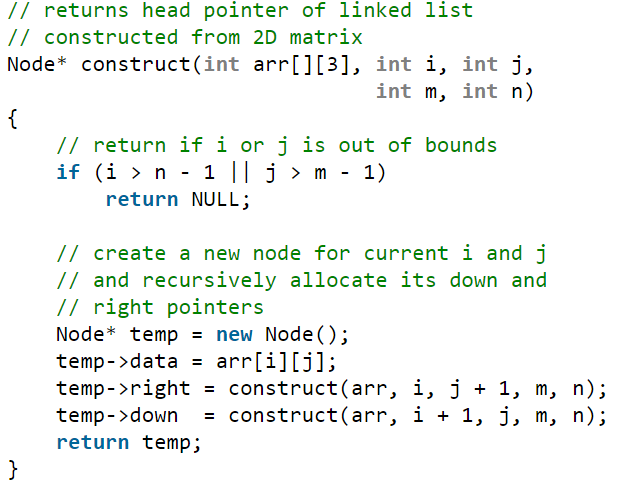
max\_ending\_here = max(head->data, max\_ending\_here + head->data);

// Update the maximum sum so far

max\_so\_far = max(max\_ending\_here, max\_so\_far);

head = head->next;

**> To recursively create a linked list matrix from a simple linked list**



> To delete duplicates from sorted list using recursion

struct Node\* removeDuplicates(struct Node\* head) {

if (head == NULL)

return NULL;

head->next = removeDuplicates(head->next);

// Check if head itself is duplicate

if (head->next != NULL &&

head->next->data == head->data) {

Node\* res = head->next;

delete head;

return res;

}

return head;

}

**Longest Common Prefix using Linked List. Given a set of strings, find the longest common prefix. e.g. - “geeksforgeeks”, “geeks”, “geek”, “geezer”; Output : "gee"**

> Create the SLL using the 1st string and then traverse over rest of the strings removing rest of the characters after the first unmatched character. The leftover is the actual prefix.

**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. At last, reverse the linked list.

**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); root->right = createBST(mid->next, n);
* In the first part, do remember that ‘mid’ should be excluded.

**Method 2:**

**In this method tree is created from leaf to root**

BinaryTree\* sortedListToBSTUtil(ListNode \*& list, int start, int end) {

if (start > end) return NULL;

// same as (start+end)/2, avoids overflow

int mid = start + (end - start) / 2;

BinaryTree \*leftChild = sortedListToBSTUtil (list, start, mid-1);

BinaryTree \*parent = new BinaryTree(list->data);

parent->left = leftChild;

list = list->next;

parent->right = sortedListToBSTUtil (list, mid+1, end);

return parent;

}

BinaryTree\* sortedListToBST(ListNode \*head, int n) {

return sortedListToBSTUtil(head, 0, n-1);

}

**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 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 sorted 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.**

**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.

**To check if linked list is palindrome**

Use a function same as printing the reverse of list. When you reach the end, start the static pointer first = first->next and at each step compare the data at left node with the data at end node.

**Self-organizing list**

Self-organizing list is linked list with condition attached that the most accessed node is near to the front.

3 ways that linked list re-organized itself:

**> Move to front**: Whenever an accessed is made, element is moved to front.

Pros: Dynamic, consumes less memory and is fast.

Cons: An element is moved even if it accessed less number of times. So it may prioritize infrequently accessed nodes

**> Maintain a count variable which maintains the number of times a node is accessed**.

Pros: It's more realistic in representing the actual access pattern.

Cons: Doesn't well adapts to real organization of data. If A's count is 100 and now B is being accessed continually. Then it should be accessed

60 more times for it to become eligible to lead the linked list. Also, it's not space efficient.

**> Transpose Method**: This technique involves swapping an accessed node with its predecessor. Therefore, if any node is accessed, it is swapped with the node in front unless it is the head node, thereby increasing its priority. This algorithm is again easy to implement and space efficient and is more likely to keep frequently accessed nodes at the front of the list. However, the transpose method is more cautious. i.e. it will take many accesses to move the element to the head of the list. This method also does not allow for rapid response to changes in the query distributions on the nodes in the list.

**Notes**:  
> Two ways of updating in loop: If we want no extra argument in function think returning pointers/variables from the recursive calls otherwise you have to maintain an additional argument which you need to update e.g. – head pointer reference is passed, if needed to be updated or we can sent back new node and assign that to node to append to as \*->next = recurCall (node\*) rather than – recurCall (node\*, head)

> If it seems that you need 2 recursive calls, try 1 loop and recursive calls inside it

> When you have 2 pointers for a given node in list, consider swapping the pointers and use it for respective purpose.

> While creating list dynamically, try creating list by adding node in front…the best and simple way.

> Unrolled linked list is list which contains > 1 elements in each node. It has benefits of both cache access & fast insertion/deletion.

> To perform operations on singly linked list with O(1) space and which could have been done easily with recursion otherwise, use XOR linked list.

> An efficient way to reverse a DLL is to maintain 2 pointers: one at start and one at end and move them in opposite directions, swapping them.

> For most part of solving linked list questions. Do next = cur->next; as first statement in loop

>

**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:**

In-order traversal of a Binary tree is either be done using recursion or with the use of an auxiliary stack. The idea of threaded binary trees is to make in-order 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 in-order 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 in-order successor.

Sample tree:

10

/ \

6 12

/ \ / \

2 9 11 18

\ / \

12 16 20

**To Check if a tree is a BST or not**

Just checking whether max\_left<present\_node && present\_node<min\_right is wrong since there can be faulty element that can make is non-BST so for that we need to check min and max value from a given depth as well.

if (node==NULL) return 1;

if (node->data < min || node->data > max)

return 0;

// tighten the min or max constraint

return isBSTUtil(node->left, min, node->data-1) &&

isBSTUtil(node->right, node->data+1, max);

For edge-cases

if (root == NULL)

return true;

if (l != NULL and root->data <= l->data)

return false;

if (r != NULL and root->data >= r->data)

return false;

return isBST(root->left, l, root) && isBST(root->right, root, r);

or we can use in-order traversal, checking is cur is > prev

if (!isBST(root->left)) return false;

if (prev != NULL && root->data <= prev->data) return false;

prev = root;

return isBST(root->right);

Total number of binary trees with 'n' nodes  
Total number of possible Binary Search Trees with n different keys (countBST(n)) = Catalan number Cn = (2n)! / ((n + 1)! \* n!)

**To delete a path < sum**

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

bool 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->right, 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 pass directional flag to true while going left.

// Recur for left and right subtrees

deepestLeftLeafUtil(root->left, lvl+1, maxlvl, true, resPtr);

deepestLeftLeafUtil(root->right, lvl+1, maxlvl, false, resPtr);

**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 (root == NULL) return NULL;

if (root->key == n1 || root->key == n2)

return root;

Node \*left\_lca = findLCA(root->left, n1, n2);

Node \*right\_lca = findLCA(root->right, n1, n2);

// if both keys are present this node is the LCA

if (left\_lca && right\_lca) return root;

// Otherwise check if left subtree or right subtree is LCA

return (left\_lca != NULL)? left\_lca: right\_lca;

**Approach 2** (If parent pointers are given) : Take first node and store all its ancestors in hash-table. 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 root**

**(To print from leaf, use array to store complete path, didn’t find any other solution)**

int printKDistantfromLeaf(Node\* node, int k)

if(k == 0) {

printf("%d", root->data );

return;

}

else {

printKDistant( root->left, k-1 ) ;

printKDistant( root->right, k-1 ) ;

}

**To print vertical order traversal**

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

Map is implemented in the form of RB-tree. Hence, time complexity would be O(nlogn). To make time complexity O(n), use unordered\_map<int, vector<int>> and for traversing you have to maintain 2 variables min and max while doing level order traversal.

**Approach 2**: Find min and max width of tree. Then traverse the tree and print the nodes in level order fashion from min to max.

**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 (level > \*max\_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 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

// below condition also takes into account if the max path sum

// doesn’t pass through root node.

        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));

}

**For finding floor and ceil in a tree**

if(root.data == key) {

value.floor = root.data;

value.ceil = root.data;

}

else {

if(root.data < key) {

value.floor = root.data;

floorCeil(root.right, value, key);

}

else {

value.ceil = root.data;

floorCeil(root.left, value, key);

}

**To check if a tree with given preorder traversal has only 1 child at each node**

For this to happen go from len-1 to 0 and for each access, we should have min, max maintained which must be updated and if, for all indices, we have min and max both either greater or less than the given node. We are good to go. Do update min and max after visiting this node if this node’s value is not in between min and max.

**To create a tree recursively from preorder traversal**

// preIndex is used to keep track of index in pre[].

node\* constructTreeUtil(int pre[], int\* preIndex, int low,

int high, int size)

{

// Base case

if (\*preIndex >= size || low > high) return NULL;

// The first node in preorder traversal is root. So take

node\* root = newNode(pre[\*preIndex]);

\*preIndex = \*preIndex + 1;

// If the current subarray has only one element, no need

if (low == high) return root;

// Search for the 1st element > root

for (int i = low; i <= high; ++i)

if (pre[i] > root->data)

break;

// Use the index of element found in preorder to divide

// preorder array in two parts. Left subtree and right

// subtree

root->left = constructTreeUtil(pre, preIndex, \*preIndex,

i - 1, size);

root->right = constructTreeUtil(pre, preIndex, i, high, size);

return root;

}

> **To create min-heap in place using a BST**

Convert that tree to SLL and then use queue to take front and make next 2 nodes as left and right child recursively..push those 2 nodes in queue and loop over..perform operations.

> **To convert tree in min-heap such that elements on LHS are smaller than element on RHS**.

Take an array.. store in-order traversal of tree in that array and then do.. pre-order traversal. while doing so, copy the values from array

**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.

**The distance between two nodes can be obtained in terms of lowest common ancestor. Following is the formula.**

Dist(n1, n2) = Dist(root, n1) + Dist(root, n2) - 2\*Dist(root, lca)

**To print complete boundary of tree**

1. Print the left boundary in top-down manner.

2. Print all leaf nodes from left to right, which can again be sub-divided into two sub-parts:

…..2.1 Print all leaf nodes of left sub-tree from left to right.

…..2.2 Print all leaf nodes of right subtree from left to right.

1. Print the right boundary in bottom-up manner.

We need to take care of one thing that nodes are not printed again. e.g. The left most node is also the leaf node of the tree.

**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 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.

Another 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 using queue, while pushing left/right set flag to true in case of empty child. When we go for other child right/left and it is present while flag was true return false.

Otherwise enqueuer.

**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 in-order successor of the node, replace the node to be deleted with that node and delete that in-order 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.

**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 it’s present 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 size of each subtree recursively

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 Postorder 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**: First find the level of that node using recursion. Then do another recursion and 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 node check if any of left or right node is the node for which we want to print cousin. If yes, then break the loop. The remaining elements in the queue will be the cousins of that required nodes.

**To find diameter if binary tree**

/\* Function to find height of a tree \*/

int height(Node\* root, int& ans)

{

if (root == NULL)

return 0;

int left\_height = height(root->left, ans);

int right\_height = height(root->right, ans);

// update the answer, because diameter of a

// tree is nothing but maximum value of

// (left\_height + right\_height + 1) for each node

ans = max(ans, 1 + left\_height + right\_height);

return 1 + max(left\_height, right\_height);

}

**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

int maxPathLenUtil(Node \*root, int prev\_val, int prev\_len) {

if (!root)

return prev\_len;

// The value of the current node will be

// prev Node for its left and right children

int cur\_val = root->val;

// If current node has to be a part of the

// consecutive path then it should be 1 greater

// than the value of the previous node

if (cur\_val == prev\_val+1)

{

// a) Find the length of the Left Path

// b) Find the length of the Right Path

// Return the maximum of Left path and Right path

return max(maxPathLenUtil(root->left, cur\_val, prev\_len+1),

maxPathLenUtil(root->right, cur\_val, prev\_len+1));

}

// Find length of the maximum path under subtree rooted with this

// node (The path may or may not include this node)

int newPathLen = max(maxPathLenUtil(root->left, cur\_val, 1),

maxPathLenUtil(root->right, cur\_val, 1));

// Take the maximum previous path and path under subtree rooted

// with this node.

return max(prev\_len, newPathLen);

}

**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 it’s 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

}

recur(left);

recur(right);

// remove node from map

**To merge 2 BSTs:**

> Take nodes of 1 tree 1-by-1 and insert into another

> Store inorder traversals of both in an array and at last create another after merging both those arrays. Then create another BST from that array

> Convert both the BSTs to DLLs, merge those DLLs and create BST from that

**To print 2 sorted trees in sorted fashion with max space being height of tree:**

Do inorder traversal of both trees simultaneously using stack. Picking smaller of the element which is not NULL first.

**> To do postorder traversal using a single stack:**

In this case you have to put both root's right child and then root onto the stack. Code below

void postOrderIterative(struct Node\* root)

{

if (root == NULL)

return;

struct Stack\* stack = createStack(MAX\_SIZE);

do {

// Move to leftmost node

while (root)

{

// Push root's right child and then root to stack.

if (root->right) push(stack, root->right);

push(stack, root);

// Set root as root's left child

root = root->left;

}

// Pop an item from stack and set it as root

root = pop(stack);

// If the popped item has a right child and the right child is not

// processed yet, then make sure right child is processed before root

if (root->right && peek(stack) == root->right)

{

pop(stack); // remove right child from stack

push(stack, root); // push root back to stack

root = root->right; // change root so that the right

// child is processed next

}

else // Else print root's data and set root as NULL

{

printf("%d ", root->data);

root = NULL;

}

} while (!isEmpty(stack));

}

**Morris inorder traversal without stack**

Find predecessor. If predecessor->right == NULL, attach to current and go left, else if predecessor->right == current, break the link, predecessor->right = NULL

if current->left == NULL, print the node and current = current->left;

Largest element smaller than current element on left for every element in Array

Put all the elements in a balanced BT (set<T> in C++) and then find the element using std::lower\_bound() API

> **To find the largest/smallest on the LHS or RHS**

Insert the array element 1-by-1 and for each access use the predecessor of current element. Since, elements in BST are in sorted order, so you will get the largest/smallest element on LHS/RHS. Use lower\_bound function for this.

**> To check if a given BT has balanced BST of a given length.**

Take a struct at the beginning of each function call which keeps the following parameters:

bool isBST , bool balanced , int size , int hight , int min , int max

With the tree traversal going in post-order fashion, we need to return this struct from each side of the tree.. check the min/max values for BST conformance and difference of heights to check if tree is balance, if yes make/keep the flag 'isBST' as TRUE or FALSE accordingly. Add a following check at bottom too, so that you can exit when you are done. For this, check the value of ans after completion of each left/right iteration. If true, return the temp variable created at entry.

// Condition to check whether the size of Balanced BST is K or not

if (temp.balanced == true && temp.size == k)

ans = true;

**> To find kth smallest element, use divide-by-2 approach. normal inorder will take O(n) complexity**

To count the number of nodes smaller than a given node on RHS in an array. Take a tree with attribute size that number of nodes beneath that tree. Insert the element 1 by 1 in a tree and match it with root if root is < given key that 'size' attribute of root->left will be the answer for that given node.

**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.

**> To find inorder successor**

if (n->right != NULL)

return minValue(n->right);

struct node\* succ = NULL;

while (root != NULL) {

if (n->data < root->data) {

succ = root;

root = root->left;

}

else if (n->data > root->data) root = root->right;

else break;

}

return succ;

**Construct a BST from linked list**

In this, We’ll create the list bottom up. First we will create left node, then we’ll create root. Linking left to root and then attach right to root->right.

Following is the solution:

struct TNode\* sortedListToBSTRecur(struct LNode \*\*head\_ref, int n)

{

    /\* Base Case \*/

    if (n <= 0)

        return NULL;

    /\* Recursively construct the left subtree \*/

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

    /\* Allocate memory for root, and link the above constructed left

       subtree with root \*/

    struct TNode \*root = newNode((\*head\_ref)->data);

    root->left = 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) which is n-n/2-1\*/

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

    return root;

}

**Check if a tree is height balanced**

bool isBalanced(struct node \*root, int\* height)

{

/\* lh --> Height of left subtree

rh --> Height of right subtree \*/

int lh = 0, rh = 0;

/\* l will be true if left subtree is balanced

and r will be true if right subtree is balanced \*/

int l = 0, r = 0;

if(root == NULL)

{

\*height = 0;

return 1;

}

/\* Get the heights of left and right subtrees in lh and rh

And store the returned values in l and r \*/

l = isBalanced(root->left, &lh);

r = isBalanced(root->right,&rh);

/\* Height of current node is max of heights of left and

right subtrees plus 1\*/

\*height = (lh > rh? lh: rh) + 1;

/\* If difference between heights of left and right

subtrees is more than 2 then this node is not balanced

so return 0 \*/

if((lh - rh >= 2) || (rh - lh >= 2))

return 0;

/\* If this node is balanced and left and right subtrees

are balanced then return true \*/

else return l&&r;

}

In case, you want to send back huge data in recursion, rather than sending pointer or variable, try returning a structure.

**To find largest BST in BT**

// Returns Information about subtree. The

// Information also includes size of largest

// subtree which is a BST.

Info largestBSTBT(Node\* root)

{

    // Base cases : When tree is empty or it has

    // one child.

    if (root == NULL)

        return {0, INT\_MIN, INT\_MAX, 0, true};

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

        return {1, root->data, root->data, 1, true};

    // Recur for left subtree and right subtrees

    Info l = largestBSTBT(root->left);

    Info r = largestBSTBT(root->right);

    // Create a return variable and initialize its

    // size.

    Info ret;

    ret.sz = (1 + l.sz + r.sz);

    // If whole tree rooted under current root is

    // BST.

    if (l.isBST && r.isBST && l.max < root->data &&

            r.min > root->data)

    {

        ret.min = min(l.min, min(r.min, root->data));

        ret.max = max(r.max, max(l.max, root->data));

        // Update answer for tree rooted under

        // current 'root'

        ret.ans = ret.sz;

        ret.isBST = true;

        return ret;

    }

    // If whole tree is not BST, return maximum

    // of left and right subtrees

    ret.ans = max(l.ans, r.ans);

    ret.isBST = false;

    return ret;

}

To check if each internal node of BST has exactly one child.

Solution: The next successor will be either smaller of larger and that too will be the same property of last node w.r.t the current node.

bool hasOnlyOneChild(int pre[], int size)

{

    int nextDiff, lastDiff;

    for (int i=0; i<size-1; i++)

    {

        nextDiff = pre[i] - pre[i+1];

        lastDiff = pre[i] - pre[size-1];

        if (nextDiff\*lastDiff < 0)

            return false;;

    }

    return true;

}

To correct 2 swapped nodes:

Solution: 2 situations: Either 2 nodes are adjacent or are far apart.

// Recur for the left subtree

correctBSTUtil( root->left, first, middle, last, prev );

// If this node is smaller than the previous node, it's violating

// the BST rule.

if (\*prev && root->data < (\*prev)->data)

{

// If this is first violation, mark these two nodes as

// 'first' and 'middle'

if ( !\*first )

{

\*first = \*prev;

\*middle = root;

}

// If this is second violation, mark this node as last

else

\*last = root;

}

// Mark this node as previous

prev = root;

// Recur for the right subtree

correctBSTUtil( root->right, first, middle, last, prev );

**Convert a BST to a Binary Tree such that sum of all greater keys is added to every key**

Do reverse in-order traversal and keeping summing up old nodes while adding to current node the sum.

**To find triplet in a tree whose sum is 0**

**2 approaches:**

**Approach 1**: Store the tree in array after inorder traversal. Find the triplet by fixing 1 element and doing l++/r—in array

**Approach 2**: Convert tree to DLL. Now, use the same method. But here, caveat is that you don’t need array and hence no extra space.

**Delete the value outside range**

// Removes all nodes having value outside the given range and returns the root

// of modified tree

node\* removeOutsideRange(node \*root, int min, int max)

{

   // Base Case

   if (root == NULL)

      return NULL;

   // First fix the left and right subtrees of root

   root->left =  removeOutsideRange(root->left, min, max);

   root->right =  removeOutsideRange(root->right, min, max);

   // Now fix the root.  There are 2 possible cases for toot

   // 1.a) Root's key is smaller than min value (root is not in range)

   if (root->key < min)

   {

       node \*rChild = root->right;

       delete root;

       return rChild;

   }

   // 1.b) Root's key is greater than max value (root is not in range)

   if (root->key > max)

   {

       node \*lChild = root->left;

       delete root;

       return lChild;

   }

   // 2. Root is in range

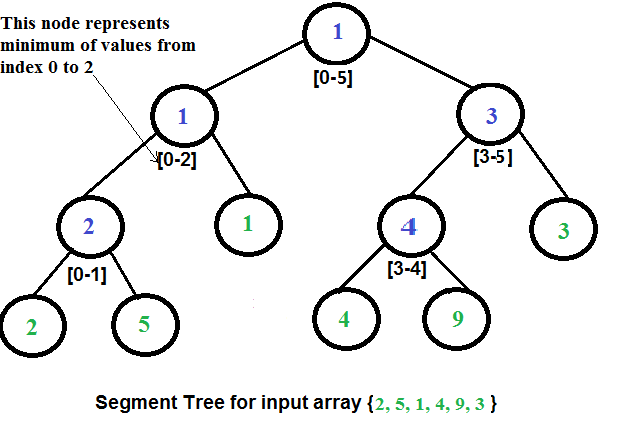
   return root;

}

**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;

}

**To store all root->leaf paths resulting in a given sum:**

void findListWithGivenSum(TreeNode \*ptr, int count, int sum, vector<int>& tmp\_vec)

{

if(ptr == NULL)

return;

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

if(count+ptr->val == sum) {

//cout<<"now pushing back\n";

tmp\_vec.push\_back(ptr->val);

result.push\_back(tmp\_vec); //vector< vector<int> > result; tmp\_vec.pop\_back();

return;

}

}

count += ptr->val;

//cout<<"After addition, count: "<<count<<endl;

tmp\_vec.push\_back(ptr->val);

findListWithGivenSum(ptr->left, count, sum, tmp\_vec);

findListWithGivenSum(ptr->right, count, sum, tmp\_vec);

count -= ptr->val;

tmp\_vec.pop\_back();

}

**Notes:**

> To handle the case in which we need to check if one child is present and other doesn’t, do it this way (2nd check will find that)

If(!tree->right && !tree->left)

// if both left and right child are absent

If(tree->left == NULL || tree->right == NULL)

// Now only one of them is present..

> Function to check 2 intervals:

bool doOVerlap(Interval i1, Interval i2)

{

    if (i1.low <= i2.high && i2.low <= i1.high)

        return true;

    return false;

}

**Notes**

> Consider converting tree to DLL for some complicated problems of binary tree

> Two arrays represent the same BST if, for every element x, the elements in left and right subtrees of x appear after it in both arrays. And same is true for roots of left and right subtrees

> Total number of possible Binary Search Trees with n different keys (countBST(n)) = Catalan number Cn = (2n)! / ((n + 1)! \* n!)

**AVL trees**

AVL are self balancing BSTs in which height of left subtree and right subtree is not > 1. To keep it in shape, if we insert an element which makes is unbalanced, we have to make left-right operations in some order to make it balanced. Steps:

1) Perform the normal BST insertion.

2) The current node must be one of the ancestors of the newly inserted node. Update the height of the current node.

3) Get the balance factor (left subtree height – right subtree height) of the current node.

4) If balance factor is greater than 1, then the current node is unbalanced and we are either in Left Left case or left Right case. To check whether it is left left case or not, compare the newly inserted key with the key in left subtree root.

5) If balance factor is less than -1, then the current node is unbalanced and we are either in Right Right case or Right-Left case. To check whether it is Right Right case or not, compare the newly inserted key with the key in right subtree root.

Code for insertion:

// An AVL tree node

class Node

{

public:

int key;

Node \*left;

Node \*right;

int height;

};

int height(Node \*N)

{

if (N == NULL)

return 0;

return N->height;

}

// A utility function to get maximum

// of two integers

int max(int a, int b)

{

return (a > b)? a : b;

}

Node \*rightRotate(Node \*y)

{

Node \*x = y->left;

Node \*T2 = x->right;

// Perform rotation

x->right = y;

y->left = T2;

// Update heights

y->height = max(height(y->left),

height(y->right)) + 1;

x->height = max(height(x->left),

height(x->right)) + 1;

// Return new root

return x;

}

Node \*leftRotate(Node \*x)

{

Node \*y = x->right;

Node \*T2 = y->left;

// Perform rotation

y->left = x;

x->right = T2;

// Update heights

x->height = max(height(x->left),

height(x->right)) + 1;

y->height = max(height(y->left),

height(y->right)) + 1;

// Return new root

return y;

}

// Get Balance factor of node N

int getBalance(Node \*N)

{

if (N == NULL)

return 0;

return height(N->left) - height(N->right);

}

Node\* insert(Node\* node, int key)

{

/\* 1. Perform the normal BST insertion \*/

if (node == NULL)

return(newNode(key));

if (key < node->key)

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

else if (key > node->key)

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

else // Equal keys are not allowed in BST

return node;

/\* 2. Update height of this ancestor node \*/

node->height = 1 + max(height(node->left),

height(node->right));

/\* 3. Get the balance factor of this ancestor

node to check whether this node became

unbalanced \*/

int balance = getBalance(node);

// If this node becomes unbalanced, then

// there are 4 cases

// Left Left Case

if (balance > 1 && key < node->left->key)

return rightRotate(node);

// Right Right Case

if (balance < -1 && key > node->right->key)

return leftRotate(node);

// Left Right Case

if (balance > 1 && key > node->left->key)

{

node->left = leftRotate(node->left);

return rightRotate(node);

}

// Right Left Case

if (balance < -1 && key < node->right->key)

{

node->right = rightRotate(node->right);

return leftRotate(node);

}

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

return node;

}

**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 tree

STACK:

Clone a stack without using extra space:

> Using array: Pop all the the elements from source stack and push them to destination stack. Keep a counter to check how much elements were copied. Now, save the last element pushed element. In first iteration, pop all element from destination and push them to source stack. In further iterations, pop and push and revert (last\_popped\_element\_count-1)

> Using stack: It's just about reversing the list and copying the data over

**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(1)
* Insertion - O(logn)
* Delete - O(logn)
* Creation of Heap - O(n) (Because, for a given node, heapify operation time depends on the position of that node in that heap)

**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.

Maintain min-heap for upper half of numbers and max-heap for lower half of the numbers.

**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.

**Cuckoo Hashing**:

A hashing technique in which we maintain 2 tables instead of 1 so that when conflict arises then we look for empty location in table 2. If there is collision in both tables then one of the older key is kicked out and re-positioned, which in turns kicks-out other keys in recursion.

**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.

**To find k-most frequent elements in input stream.**

1. Maintain a Linked List with decreasing order of frequency. Each node of linked list will have the number and its frequency.   
2. Maintain a hash map with key as number and value as the pointer to the node in the linked list.   
3. If a new number comes, add it to the tail of the linked list, maintaining the hash map.   
  
The cost of maintaining the list for every number is O(1).   
Reason:   
1. When the list is empty - add a node O(1)   
2. When node is not in list - add a node O(1)   
3. when frequency of a node changes, it moves before the previous node - O(1)   
  
The cost of returning the top occurred (k) numbers - O(k) - returning the first k numbers from the linked list. Similar method is used in implementing cache.

**To find k-largest/minimum elements in a stream**

Maintain a min-heap/max-heap respectively

**To find palindrome in running stream**

Use rolling hash method of Robin-Karp algorithm: Store hash of 1st half and hash of 2nd half and compare after each insertion. If they match, match character-by-character of 2 substrings to check if the two halves are of an actual palindrome.

**4-Sum in array**

Problem: Find quadruplets in array – a, b, c, d such that sum of those four integers is equal to a given target

Solution:

> Sort the array.

> Use 2 loops. i=0 to size-3 and j = i+1 to size-2

> Fix initial 2 values as [i] and [j] and loop over remaining ones while increasing if result is getting smaller and decreasing the higher index in case result is larger than expected.

**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.

Let the given string be “STRING”. In the input string, ‘S’ is the first character. There are total 6 characters and 4 of them are smaller than ‘S’. So there can be 4 \* 5! smaller strings where first character is smaller than ‘S’, like following

R X X X X X  
I X X X X X  
N X X X X X  
G X X X X X

Now let us Fix S’ and find the smaller strings staring with ‘S’.

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.

**Approach 3**: Dynamic programming. Single table and O(n^2)

**Longest Common substring:**

Solution: Use dynamic programming using a 2-D array as int LCS[M+1][N+1]

Code below:

for (int i=0; i<=m; i++) {

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

if (i == 0 || j == 0) {

LCSuff[i][j] = 0;

continue;

} else if (X[i-1] == Y[j-1]) {

LCSuff[i][j] = LCSuff[i-1][j-1] + 1;

result = max(result, LCSuff[i][j]);

} else {

LCSuff[i][j] = 0;

}

}

}

**To make the string palindromic:**

Solution: Check the number of characters which are repeated odd number of times. Then, requisite number will be the count-1 since one extra character is allowed flanked by palindromic sub-string

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**

**Longest Increasing Sub-sequence:**

Problem: Find the sub-sequence (not necessarily contiguous) which is increasing and contains maximum elements

Approach: Use DP, using 2 loops i=0->n; j=0->i we can compare each a[j] with a[i] and update the value in lis[i], another array used to store result.

**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 sub-array with maximum number of 0’s and 1’s**

Consider 0 as -1 and find max size subarray such that sum of that subarray is zero.

**Find first non-repeating element**

> To find it in O(n), use normal hashmap or specifically char array of size [256].

> To find it in O(1) time, you have to use DLL and 2 arrays – repeated[i] (stores how many times an item has been repeated..0 means not yet encountered, 1 means once, 2 means it’s repeated) and DLLPtrs[256] which stores pointer of DLLs. If an item is found for which repeated[item] >=2, that’s ignored. If it was 1, make it 2 and delete corresponding entry from DLL.

**Find first repeating element**

To find it in O(1), use previous approach, else normal hashmap would work.

**Find largest sub-array with contiguous elements**

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

**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 adding the 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;

**Dutch national flag problem**

**We are given an array of integers of values 0, 1 and 2. The task is to arrange them in increasing order of 0, 1 and then 2. Complexity should be O(n)**

The program will use 3 elements i0, i and i2. i0 will point to the last 0 discovered. ‘i’ will be the main index used for traversal and left of which will contain 0’s and 1’s.’ i2’ will be set to extreme right initially and decremented as we find 2’s

while( i < i2 ) {

if(arr[i] == 0) {

swap(arr[i0+1], arr[i]);

i0++; i++;

}

else if(arr[i] == 2 ) {

swap(arr[i2-1], arr[i]); //Don't increment ‘i’ because we don't know what’s the current element after swap

--i2;

}

else

i++;

}

**To find median in 2 sorted arrays**

if (m1 < m2)

return getMedian(ar1 + n/2, ar2, n - n/2 +1);

else if(m1 == m2) {

return m1;

}

else {

return getMedian(arr1, arr2+n/2, n - n/2 +1);

**> Activity selection problem: Use greedy approach.**

Sort the task by finishing time and then go on picking for those whose start time is greater than last's finishing time.

**> For rotating the matrix clockwise or anticlockwise:**

Divide the matrix into squares or cycles. First cycle will be outer square (1st row, last column, last row, first column). Now, for each cycle, swap elements in group. temp=(0)(0), (0)(0) = (0)(3), (0)(3)=(3)(3), (3)(3)=(3)(0), (3)(0)=temp.

**> Linear search improve tips:**

Use multicore processor

Add that element to end and check its presence. It’ll reduce complexity from O(2n) to O(n)

Move to front

**>Trapping rainwater puzzle**

>>Trapping rain water in O(n) space - min(lmax, rmax)-a[i] and using O(1) space:

while (lo <= hi) {

if (arr[lo] < arr[hi]) {

if (arr[lo] > left\_max) {

left\_max = arr[lo];

else

result += left\_max - arr[lo]; // water on curr element = max - curr

lo++;

}

else {

if (arr[hi] > right\_max)

right\_max = arr[hi];

else

result += right\_max - arr[hi];

hi--;

}

}

> **To arrange a given array so that arr[i] becomes arr[arr[i]]**

loop 0 to n: arr[i] += (arr[arr[i]] % n) \* n;

loop 0 to n: arr[i] = arr[i]/n;

>**Given an array arr[], find the maximum j – i such that arr[j] > arr[i]**

Create another array max[] that will store, for a given element a[i], maximum element on its right, including itself. Now, we have monotonically decreasing array and we can find extreme right maximum for that array using binary searched performed on array max[]

**> Binary search in infinite array**

For binary search in infinite array, use unbounded search wherein you will first find range: i to i\*2 wherein element 'x' resides and then perform binary search in that interval

**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.

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;

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.

**To find the maximum occurring element in array**

Iterate though input array arr[], for every element arr[i], increment arr[arr[i]%k] by k, where k is the number of elements in array

**Largest Sum Contiguous Subarray - Find the sum of contiguous subarray within an array of numbers which has the largest sum (Solved using Kadane’s Algorithm)**

int maxSubArraySum(int a[], int size)

{

int max\_so\_far = INT\_MIN, max\_ending\_here = 0;

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

{

max\_ending\_here = max\_ending\_here + a[i];

if (max\_so\_far < max\_ending\_here)

max\_so\_far = max\_ending\_here;

if (max\_ending\_here < 0)

max\_ending\_here = 0;

}

return max\_so\_far;

}

**Find the missing number in an array of dimension ‘n’ which contains the numbers from 0 to n**

Do XOR of numbers 1 to N and then XOR of given array. After that, XOR of both the results will give the result.

**Find subarray with given sum**

Start from i=0 and cur\_sum=a[0]; Now keep adding cur\_sum += a[i]; in map with pair<cur\_sum, index>. After each addition check if (cur\_sum-sum) exists in map. If it is then (cur\_index-searched\_index) will give us the range.

**Equilibrium point in array:**

Calculate sum of array in variable ‘sum’

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

{

sum -= arr[i]; // sum is now right sum for index i

if(leftsum == sum)

return i;

leftsum += arr[i];

}

**Maximum Sum Increasing Subsequence**

Let the input array be A. We have to find the subsequence which is in ascending order and has the maximum sum. Take another array MSIS with same size as that of A and initialize all its elements as the contents of A. Take 2 loops and for each position, check if sum of MSIS[j] + arr[i] > MSIS[i] and A[i] > A[j]. At last find the max in MSIS array

Below is the main loop:

for ( i = 1; i < n; i++ )

for ( j = 0; j < i; j++ )

if ( arr[i] > arr[j] && msis[i] < msis[j] + arr[i])

msis[i] = msis[j] + arr[i];

**Note: Use a map <index\_cur, index\_picked\_max\_from> to get the results. That’ll be <I, j>**

**Minimum number of platforms needed for trains:**

**Solution:** Sort arrays of both arrival and departure time by storing them both in another array. Now scan through the array. Take counter = 0 and max = 0.

For each arrival, counter++ and for each departure counter--. For each step, check if counter > max then, max = counter

**Chocolates distribution – array of size ‘n’ Each entry represents number of chocolates in packet and it has to be distributed among ‘m’ students with the condition that difference between max and min is minimum.**

**Solution:** Sort the array (of n integers). We have to find the minimum of (max\_in\_window – min\_in\_window) as i = 0 to i+m and then ++i

Find the min. difference in this window starting from I = 0 to i+m-1

**Stock Buy Sell to Maximize Profit**

**Solution:** The cost of a stock on each day is given in an array, find the max profit that you can make by buying and selling in those days.

Solution: We have to find max difference here. So start from i=0 and find minimum by traversing till arr[i+1]<arr[i] which will give use the local minima. Let new state be t = i. Then starting from t move while(arr[t+1]>arr[t]) this will give us maxima. Store the both and set i = t+1 to move over again.

**To find the kth smallest or largest element**

**Solution:** You can use either min-max-heap or quickselect algorithm.

**To find non-repeating element in a sorted array in O(logn)**

**Solution:** All numbers before that unique element have their first occurrence at even indices (0, 2, 4 …) and after that unique number at odd indices (1, 3, 5, …) because of disruption in sequence by that number. Using this property we can find the number such as if our index is even, then check if next is same number as the present one. If yes, that unique number exists on RHS of that index otherwise it is on LHS of that index.

**To find Pythagoras triplet in an array:**

* Do square of each element O(n)
* Sort all elements (O(nlogn)
* Use 2 loops to find 2 numbers whose sum is a number currently under consideration

**Find trapped rain water**

Given is an array which specifies the tower size. Find the volume of water that can be stored in vacant area.

Take two arrays. Store left\_max to store max. tower on left and array right\_max to store the size of maximum tower.

For each index location, do

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

water += min(left[i],right[i]) - arr[i];

**Search in a row wise and column wise sorted matrix**

Start from top right. In case number is < e[i] go left else if number is > current number, go down else return the number.

**LCS(Longest Common Subsequence)**

/\* Returns length of LCS for X[0..m-1], Y[0..n-1] \*/

int lcs( char \*X, char \*Y, int m, int n )

{

if (m == 0 || n == 0)

return 0;

if (X[m-1] == Y[n-1])

return 1 + lcs(X, Y, m-1, n-1);

else

return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));

}

**3-Sum zero**

Find all triplets in an array so that sum of those 3 is 0.

Approach:

> Sort the array

> Use 2 loops, fix ‘I’ and move ‘j’ and ‘k’. if sum < 0, ++j else –k;

> Add all such triplets to vector<vector<int>>

**Minimize absolute difference among 3 sorted array**

Solution:

Find max and min among 3 elements of each array. Then, depending upon whose value is min, increment the index.

**Notes:**

> When questions demands using 2 pointers and starting 1 pointer from left and 1 from right is not obvious choice, try starting from beginning.

**Stacks**

Design a stack that supports pop(), push() and getMin() in O(1) time and O(1) space

Solution:

**Push(x)** – if stack is empty, insert x at the top of stack

If stack is not empty, compare x with minElement. If x is greater than or equal to minElement, simply insert x. If x is less than minElement, insert (2\*x – minElement) into the stack and make minElement equal to x. For example, let previous minElement was 3. Now we want to insert 2. We update minElement as 2 and insert 2\*2 – 3 = 1 into the stack.

**Pop()** : Removes an element from top of stack.

Remove element from top. Let the removed element be y. Two cases arise:

If y is greater than or equal to minElement, the minimum element in the stack is still minElement.

If y is less than minElement, the minimum element now becomes (2\* minElement – y), so update (minElement = 2\* minElement – y). This is where we retrieve previous minimum from current minimum and its value in stack. For example, let the element to be removed be 1 and minElement be 2. We remove 1 and update minElement as 2\*2 – 1 = 3.

**Graphs**

A graph is a data structure which represents a mash of interconnected components. The abstract components are termed as vertices while the connection between them are called edges.

Mathematically, a graph is represented as G(V, E) where V denotes the set of vertices while E denotes the set of edges, each edge being represented in the form of (u, v) where ‘u’ and ‘v’ are starting and ending vertex respectively.

**Representation:**

A general graph can be represented either in the form of adjacency matrix or in the form of adjacency list. Adjacency matrix consumes a lot of space in case graph is sparse, but in case graph is dense, it is efficient.

There are two basic operations that are performed on graphs viz.

Breadth-first search and

Depth-first search

For breadth-first, we use queue while for depth-first search, we use stack.

**Detecting cycle in graph:**

A Graph has cycle if there is a back-edge. When we are traversing the graph through DFS, we keep on inserting graph nodes in stack. While traversing, if we visit a vertex which is already in stack, then there exists a cycle.

We’ll use an array recStack[] which will record the items currently inside the stack. We will do recStack[i] = true; for each function entry of a particular vertex ‘i’.While traversing, child of A particular node if we encounter recStack[i] = true;, then there exists a cycle.

// This function is a variation of DFSUytil() in

bool Graph::isCyclicUtil(int v, bool visited[], bool \*recStack)

{

if(visited[v] == false)

{

// Mark the current node as visited and part of recursion stack

visited[v] = true;

recStack[v] = true;

// Recur for all the vertices adjacent to this vertex

list<int>::iterator i;

for(i = adj[v].begin(); i != adj[v].end(); ++i)

{

if ( !visited[\*i] && isCyclicUtil(\*i, visited, recStack) )

return true;

else if (recStack[\*i])

return true;

}

}

recStack[v] = false; // remove the vertex from recursion stack

return false;

}

**Topological sort of a graph:**

Topological sort of graph is order of vertices of a graph such that for every edge (u, v) in a graph, ‘u’ will always come before ‘v’

To find TS of a graph, we use two data structures – a stack and a set. Stack is used to store the result while set is used to maintain set of visited vertices.

The core of topological sort of graph is depth-first traversal. First, we pick any node and call a DFS recursive method to traverse over the children of that node which in turn make recursive call to their children. After we have traversed all children of a given node, we will put that node in stack.

// call DFS for each vertex and push the node in stack after all its // children are visited

void Graph::dfsWithStack(int vertex) {

marked[vertex] = true;

vector<int> vect = graphVector[vertex];

for(int w : vect) {

if(!marked[w]) {

dfsWithStack(w);

}

}

reverse.push(vertex);

}

void Graph::topSort() {

// Call DFS for all vertices…

for(int i = 0; i < vertexCount; ++i) {

if(!marked[i]) {

dfsWithStack(i);

}

}

// At last print the stack..

while(!reverse.empty()){

cout<<", "<<reverse.top()<<", ";

reverse.pop();

}

}

**Minimum Spanning tree:**

Minimum spanning tree or, in short, MST, is a subset of graph which contains all the vertices and in which the sum of weights of all edges is minimum. There are 2 famous algorithms to solve this problem:

**Kruskal Algorithm** : In kruskal algorithm, we use disjoint sets. One set containing all vertices which are visited in MST and other set contains yet to be visited vertices.

Steps:

> Create V sets, V being the number of vertices

> Sort all edges in increasing order of weights by using custom comparator. This step takes O(ElogE) time.

> Traverse all the edges

> for each edge, check if two extremes of that edge are present in same disjoint sets, if they are, ignore them, else, merge those 2 disjoint sets.

> Repeat the above step, till only 1 set remains.

**Prim’s Algorithm:** Prim’s algorithm is another such algorithm to find minimum spanning tree just like Kruskal algorithm. This algorithm is similar to the Dijkstra algorithm. In this algorithm, we maintain 2 sets, 1st set contains the vertices which are included in the MST and 2nd set contains the vertices which are yet to be included in MST.

In this algorithm, at each step, we pick an edge (u, v) whose weight is minimum among all the edges directed from M->N, where M represents the set containing all vertices included in MST and N represents the set containing the vertices yet to be included in MST. To pick this minimum, we will use binary heap. The binary heap will contains entries all the vertices not yet included in MST where each entry is represented as a pair (vertex, value) where ‘value’ is the current minimum weight of vertex incident on that vertex.

Steps

> Put all vertices in binary heap with their value as as infinity. Set the value of source vertex as 0.

> Now, do the following till binary heap is empty

> Pop-out the vertex with the minimum value (extract-min) operation) from binary heap and add that to the set M and update the values of all vertices in binary heap which are adjacent to popped up vertex.

**Dijkstra algorithm** – It is a single source shortest path algorithm which determines the minimum distance of all vertices from a given source. The logic is similar to what is used in Prim’s algorithm

We maintain the following data structures:

> A heap (To sort vertices based on their weights) Each heap entry is a kind of map <vertex\_id, distance>

> A map <node, parent\_of\_node> to store path to a given node.

> A map <node, distance> which represents the visited set, or the nodes which have been included in the solution.

Initially, we will put all vertices with weights set to INT\_MAX in binary heap. We will set weight of source vertex to 0 in heap and extract that out.

Now, do the following till heap is empty

* Do extract-min and pick the node with lowest weight, add that to visited set.
* Update all its adjacent vertices which are present in heap (Don’t touch the ones present in visited set) depending upon if the calculated weight is lesser than the current vertex weight.

**Difference between Prim and Dijkstra:**

The only difference I see is that Prim's algorithm stores a minimum cost edge whereas Dijkstra's algorithm stores the total cost from a source vertex to the current vertex.

So, if you want to deploy a train to connect several cities, you would use Prim's algorithm But if you want to go from one city to other saving as much time as possible, you'd use Dijkstra's algorithm.

**Shortest path in a Binary Maze**: In this, we have to find the shortest path from a given cell of a maze to another cell. A maze is represented in the form of matrix of 0’s and 1’s and cell is denoted by the coordinate which contains 1.

Solution:

We can solve this problem using Lee’s algorithm which deploys Breadth-first search to find the result. Here in, a queue of nodes is used in which each node is represented by endpoints and a variable ‘dst’ that will store the distance of that cell from the source vertex.

Shortest path in snake and ladder game: <Similar to above algorithm>

Minimum Cost Path with Left, Right, Bottom and Up moves allowed

Here also, we can use Lee’s algorithm wherein BFS is used. We will another 2-D array dist[M][N] which will store the minimum distance between that node and start node.

Solution:

> Push start node in queue.

> Do the following till queue is not empty

> Pop the node and update its adjacent nodes by checking the distance so as to pick the minimum

Code as below:

while (!st.empty())

{

// get the cell with minimum distance and delete

// it from the set

cell k = \*st.begin();

st.erase(st.begin());

// looping through all neighbours

for (int i = 0; i < 4; i++)

{

int x = k.x + dx[i];

int y = k.y + dy[i];

// if not inside boundry, ignore them

if (!isInsideGrid(x, y))

continue;

// If distance from current cell is smaller, then

// update distance of neighbour cell

if (dis[x][y] > dis[k.x][k.y] + grid[x][y])

{

// If cell is already there in set, then

// remove its previous entry

if (dis[x][y] != INT\_MAX)

st.erase(st.find(cell(x, y, dis[x][y])));

// update the distance and insert new updated

// cell in set

dis[x][y] = dis[k.x][k.y] + grid[x][y];

st.insert(cell(x, y, dis[x][y]));

}

}

}

Strongly Connected Component: SCC of a graph refers to the subgraph/tree of the given graph such that each vertex ‘v’ in subgraph is reachable from remaining vertices of subgraph.

Solution: SCC can be extracted by using Kosaraju’s algorithm.

> Do DFS of graph and store the vertices of graph in stack in order of their finishing times. Put stack.push() after

for(auto a:children)

> Reverse(Transpose) the graph

> Now, again do DFS by parsing through the stack entries and running them over the transposed graph.

Given a sorted dictionary of an alien language, find order of characters.

Solution: The main idea is to create a graph using the characters of string and do topological sort.

> Create a graph g with number of vertices equal to the size of alphabet in the given alien language. For example, if the alphabet size is 5, then there can be 5 characters in words. Initially there are no edges in graph.

> Do following for every pair of adjacent words in given sorted array.

…..a) Let the current pair of words be word1 and word2. One by one compare characters of both words and find the first mismatching characters.

…..b) Create an edge in g from mismatching character of word1 to that of word2.

> Print topological sorting of the above created graph.

**Check if array of strings can be chained to form circle**

A string X can be put before another string Y in circle if the last character of X is same as first character of Y.

Solution:

> Treat start and end character of string as vertex of graph which are connected via an edge.

> Create a graph by connecting two end points of string via an edge.

> For solution to be valid, the graph must contains the loop which can be depicted only if graph is SCC (i.e. all vertices are reachable from a given vertex in graph) and in/out-degrees of all vertices is same.

Find if we can reach from one word to another

> Use BFS:

Take a queue:

For each character of word, modify it add to queue if it is present in dictionary.

**Notes:**

BFS can only be used to find shortest distance in an unweighted graph like a mesh or maze. For a weighed graph you may need Dijkstra's algorithm or Bellmann-Ford's algorithm.

Consider a graph like this:

A------(3)-----------B

| |

\--(1)----C-------(1)/

The shortest path from A to B is via C (with a total weight of 2). A normal BFS will take the path directly from A to B, marking B as seen, and A to C, marking C as seen.

> We can use BFS to find the shortest path in weighted graph, but we need to replace each weighted nodes with nodes at unit distance.

> To find the minimum cost from source to destination in M X N grid, use dijkstra algorithm.

> A vertex in an undirected connected graph is an articulation point (or cut vertex) if removing it (and edges through it) disconnects the graph

**Dynamic Programming**

Given an array of non-negative integers, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Determine if you are able to reach the last index.

Solution: This can be done using dynamic programming starting from the end. Maintain the minimum index from where we can reach to the destination.

for (int i = n - 2; i >= 0; i--) {

bool isPossibleFromThisIndex = false;

if (i + A[i] >= minIndexPossible) {

isPossibleFromThisIndex = true;

minIndexPossible = i;

}

if (i == 0) return isPossibleFromThisIndex;

}

Find max non-contiguous sequence of numbers

If size == 1 return first element

Else if size == 2, return max of first 2 elements

Else if size == 3, return max (e[2]+e[0], e[1])

Else res[i] += max(res[i-2], res[i-3]);

Find path with minimum sum in matrix wherein we can go down and right

Either create a new 2-D matrix or keep the current one and traverse over the whole matrix with each position = minimum of [i-1][j] and [i][j-1] keeping into account the boundary conditions. At last, return array[m-1][n-1]

Longest increasing and decreasing sequence

Maintain 2 arrays inc[n] and dec[n] find common point where inc[i]+dec[i]-1 is mzximum.

Given a string, find if there is any sub-sequence that repeats itself.

iFor this problem, you only need to find a subsequence of 2 chars which is repeated in that string. For that, you can use unordered\_map<string, pair<int, int> >.

For each pair, if it doesn’t exist, insert else check its <I,j> if it’s different, you are good to go.

Notes:  
> Not necessarily you need to maintain a result array. You can play with indices.