תירגול נוסף במבנה נתונים

באופן כללי: אין צורך להתעכב על כל המימוש בפרטות אלא להבין את הרעיון הכללי. בחלק שאלות לא העתקי את הmain כי אין לזה משמעות והחוברת מספיק ארוכה....

Find n-th node in Preorder traversal of a Binary Tree

Given a Binary tree and a number N, write a program to find the N-th node in the Preorder traversal of the given Binary tree.

**Prerequisite:**[Tree Traversal](https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/)

**Examples:**

Input: N = 4

11

/ \

21 31

/ \

41 51

Output: 51

**Explanation:** Preorder Traversal of given Binary Tree is 11 21 41 51 31,

so 4th node will be 51.

Input: N = 5

25

/ \

20 30

/ \ / \

18 22 24 32

Output: 30

The idea to solve this problem is to do [preorder traversal](https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/) of the given binary tree and keep track of the count of nodes visited while traversing the tree and print the current node when the count becomes equal to N.

|  |
| --- |
| // C++ program to find n-th node of  // Preorder Traversal of Binary Tree  #include <bits/stdc++.h>  using namespace std;    // Tree node  struct Node {      int data;      Node \*left, \*right;  };    // function to create new node  struct Node\* createNode(int item)  {      Node\* temp = new Node;      temp->data = item;      temp->left = NULL;      temp->right = NULL;        return temp;  }    // function to find the N-th node in the preorder  // traversal of a given binary tree  void NthPreordernode(struct Node\* root, int N)  {      static int flag = 0;        if (root == NULL)          return;        if (flag <= N) {          flag++;            // prints the n-th node of preorder traversal          if (flag == N)              cout << root->data;            // left recursion          NthPreordernode(root->left, N);            // right recursion          NthPreordernode(root->right, N);      }  }    // Driver code  int main()  {      // construction of binary tree      struct Node\* root = createNode(25);      root->left = createNode(20);      root->right = createNode(30);      root->left->left = createNode(18);      root->left->right = createNode(22);      root->right->left = createNode(24);      root->right->right = createNode(32);        // nth node      int N = 6;        // prints n-th found found      NthPreordernode(root, N);        return 0;  } |

**Output:**

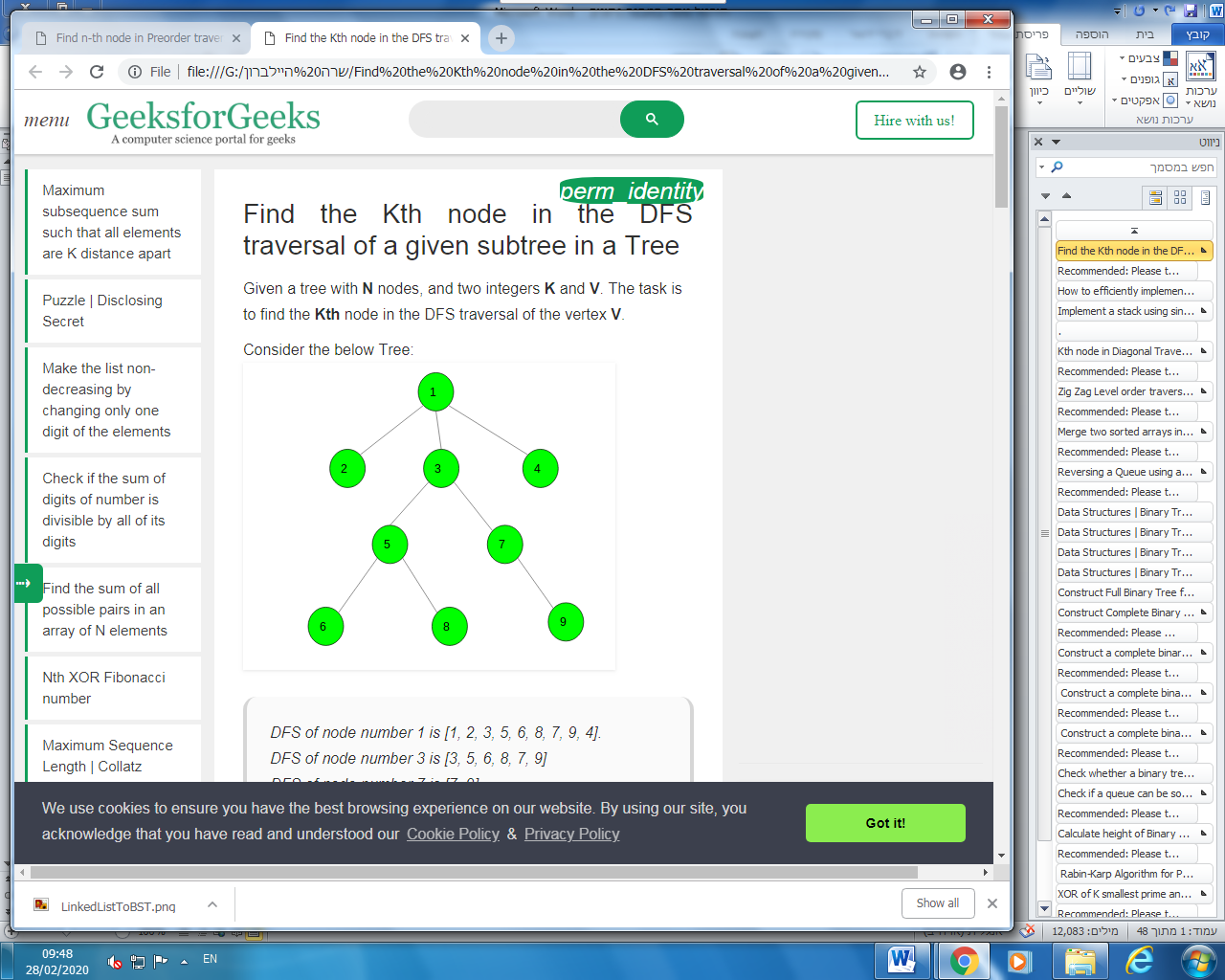
24

**Time Complexity:** O(n), where n is the number of nodes in the given binary tree.  
**Auxiliary Space:** O(1)

Find the Kth node in the DFS traversal of a given subtree in a Tree

Given a tree with **N**nodes, and two integers **K**and **V**. The task is to find the **Kth** node in the DFS traversal of the vertex **V**.

Consider the below Tree:



*DFS of node number 1 is [1, 2, 3, 5, 6, 8, 7, 9, 4].  
DFS of node number 3 is [3, 5, 6, 8, 7, 9]  
DFS of node number 7 is [7, 9]  
DFS of node number 9 is [9].*

Print “-1” if the numbers in the DFS of vertex V are less than K.

**Examples:**

**Input :** Tree: *Shown in above image*, V = 3, K = 4

**Output :** 8

**Input :** Tree: *Shown in above image*, V = 7, K = 3

**Output :** -1

**[Recommended: Please try your approach on](https://ide.geeksforgeeks.org/)*[{IDE}](https://ide.geeksforgeeks.org/)*[first, before moving on to the solution.](https://ide.geeksforgeeks.org/)**

**Approach**: Let’s construct a vector : to store the DFS traversal of the complete tree from vertex 1. Let tinv be the position of the vertex V in the vector p (the size of the vector p in moment we call DFS from the vertex V) and toutv be the position of the first vertex pushed to the vector after leaving the subtree of vertex V (the size of the vector p in moment when we return from DFS from the vertex V). Then it is obvious that the subtree of the vertex V lies in the interval [tinv, toutv).

So, to find the Kth node in the DFS of the subtree of node V, we will have to return the Kth node in the interval [tinv, toutv).

Below is the implementation of the above approach:

|  |
| --- |
| // C++ program to find the Kth node in the  // DFS traversal of the subtree of given  // vertex V in a Tree    #include <bits/stdc++.h>  using namespace std;  #define N 100005    // To store nodes  int n;  vector<int> tree[N];    // To store the current index of vertex in DFS  int currentIdx;    // To store the starting index and ending  // index of vertex in the DFS traversal array  vector<int> startIdx, endIdx;    // To store the DFS of vertex 1  vector<int> p;    // Function to add edge between two nodes  void Add\_edge(int u, int v)  {      tree[u].push\_back(v);      tree[v].push\_back(u);  }    // Initialize the vectors  void intisalise()  {      startIdx.resize(n);      endIdx.resize(n);      p.resize(n);  }    // Function to perform DFS of a vertex  // 1. stores the DFS of the vertex 1 in vector p,  // 2. store the start index of DFS of every vertex  // 3. store the end index of DFS of every vertex  void Dfs(int ch, int par)  {      p[currentIdx] = ch;        // store staring index of node ch      startIdx[ch] = currentIdx++;        for (auto c : tree[ch]) {          if (c != par)              Dfs(c, ch);      }        // store ending index      endIdx[ch] = currentIdx - 1;  }    // Function to find the Kth node in DFS of vertex V  int findNode(int v, int k)  {      k += startIdx[v] - 1;        // check if kth number exits or not      if (k <= endIdx[v])          return p[k];        return -1;  } |

How to efficiently implement k stacks in a single array?

We have discussed [space efficient implementation of 2 stacks in a single array](https://www.geeksforgeeks.org/implement-two-stacks-in-an-array/). In this post, a general solution for k stacks is discussed. Following is the detailed problem statement.

*Create a data structure kStacks that represents k stacks. Implementation of kStacks should use only one array, i.e., k stacks should use the same array for storing elements. Following functions must be supported by kStacks.*

*push(int x, int sn) –> pushes x to stack number ‘sn’ where sn is from 0 to k-1  
pop(int sn) –> pops an element from stack number ‘sn’ where sn is from 0 to k-1*

**Method 1 (Divide the array in slots of size n/k)**  
A simple way to implement k stacks is to divide the array in k slots of size n/k each, and fix the slots for different stacks, i.e., use arr[0] to arr[n/k-1] for first stack, and arr[n/k] to arr[2n/k-1] for stack2 where arr[] is the array to be used to implement two stacks and size of array be n.

The problem with this method is inefficient use of array space. A stack push operation may result in stack overflow even if there is space available in arr[]. For example, say the k is 2 and array size (n) is 6 and we push 3 elements to first and do not push anything to second second stack. When we push 4th element to first, there will be overflow even if we have space for 3 more elements in array.

**Method 2 (A space efficient implementation)**  
The idea is to use two extra arrays for efficient implementation of k stacks in an array. This may not make much sense for integer stacks, but stack items can be large for example stacks of employees, students, etc where every item is of hundreds of bytes. For such large stacks, the extra space used is comparatively very less as we use two *integer*arrays as extra space.

Following are the two extra arrays are used:  
***1) top[]:***This is of size k and stores indexes of top elements in all stacks.  
***2) next[]:*** This is of size n and stores indexes of next item for the items in array arr[]. Here arr[] is actual array that stores k stacks.  
Together with k stacks, a stack of free slots in arr[] is also maintained. The top of this stack is stored in a variable ‘free’.

All entries in top[] are initialized as -1 to indicate that all stacks are empty. All entries next[i] are initialized as i+1 because all slots are free initially and pointing to next slot. Top of free stack, ‘free’ is initialized as 0.

|  |
| --- |
| // A C++ program to demonstrate implementation of k stacks in a single  // array in time and space efficient way  #include<bits/stdc++.h>  using namespace std;    // A C++ class to represent k stacks in a single array of size n  class kStacks  {      int \*arr;   // Array of size n to store actual content to be stored in stacks      int \*top;   // Array of size k to store indexes of top elements of stacks      int \*next;  // Array of size n to store next entry in all stacks                  // and free list      int n, k;      int free; // To store beginning index of free list  public:      //constructor to create k stacks in an array of size n      kStacks(int k, int n);        // A utility function to check if there is space available      bool isFull()   {  return (free == -1);  }        // To push an item in stack number 'sn' where sn is from 0 to k-1      void push(int item, int sn);        // To pop an from stack number 'sn' where sn is from 0 to k-1      int pop(int sn);        // To check whether stack number 'sn' is empty or not      bool isEmpty(int sn)  {  return (top[sn] == -1); }  };    //constructor to create k stacks in an array of size n  kStacks::kStacks(int k1, int n1)  {      // Initialize n and k, and allocate memory for all arrays      k = k1, n = n1;      arr = new int[n];      top = new int[k];      next = new int[n];        // Initialize all stacks as empty      for (int i = 0; i < k; i++)          top[i] = -1;        // Initialize all spaces as free      free = 0;      for (int i=0; i<n-1; i++)          next[i] = i+1;      next[n-1] = -1;  // -1 is used to indicate end of free list  }    // To push an item in stack number 'sn' where sn is from 0 to k-1  void kStacks::push(int item, int sn)  {      // Overflow check      if (isFull())      {          cout << "\nStack Overflow\n";          return;      }        int i = free;      // Store index of first free slot        // Update index of free slot to index of next slot in free list      free = next[i];        // Update next of top and then top for stack number 'sn'      next[i] = top[sn];      top[sn] = i;        // Put the item in array      arr[i] = item;  }    // To pop an from stack number 'sn' where sn is from 0 to k-1  int kStacks::pop(int sn)  {      // Underflow check      if (isEmpty(sn))      {           cout << "\nStack Underflow\n";           return INT\_MAX;      }          // Find index of top item in stack number 'sn'      int i = top[sn];        top[sn] = next[i];  // Change top to store next of previous top        // Attach the previous top to the beginning of free list      next[i] = free;      free = i;        // Return the previous top item      return arr[i];  } |

Implement a stack using single queue

We are given queue data structure, the task is to implement stack using only given queue data structure.

We have discussed [a solution that uses two queues](https://www.geeksforgeeks.org/implement-stack-using-queue/). In this article, a new solution is discussed that uses only one queue. This solution assumes that we can find size of queue at any point. The idea is to keep newly inserted element always at rear of queue, keeping order of previous elements same. Below are complete steps.

// x is the element to be pushed and s is stack

**push(s, x)**

1) Let size of q be s.

1) Enqueue x to q

2) One by one Dequeue s items from queue and enqueue them.

// Removes an item from stack

**pop(s)**

1) Dequeue an item from q

[**.**](https://ide.geeksforgeeks.org/)

Below is implementation of the idea.

|  |
| --- |
| // C++ program to implement a stack using  // single queue  #include<bits/stdc++.h>  using namespace std;    // User defined stack that uses a queue  class Stack  {      queue<int>q;  public:      void push(int val);      void pop();      int top();      bool empty();  };    // Push operation  void Stack::push(int val)  {      //  Get previous size of queue      int s = q.size();        // Push current element      q.push(val);        // Pop (or Dequeue) all previous      // elements and put them after current      // element      for (int i=0; i<s; i++)      {          // this will add front element into          // rear of queue          q.push(q.front());            // this will delete front element          q.pop();      }  }    // Removes the top element  void Stack::pop()  {      if (q.empty())          cout << "No elements\n";      else          q.pop();  }    // Returns top of stack  int  Stack::top()  {      return (q.empty())? -1 : q.front();  }    // Returns true if Stack is empty else false  bool Stack::empty()  {      return (q.empty());  }    // Driver code  int main()  {      Stack s;      s.push(10);      s.push(20);      cout << s.top() << endl;      s.pop();      s.push(30);      s.pop();      cout << s.top() << endl;      return 0;  } |

**Output :**

20

10

This article is contributed by **Manu Agrawal**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Kth node in Diagonal Traversal of Binary Tree

Given a binary tree and a value **K**. The task is to print the **k-th**node in the diagonal traversal of the binary tree. If no such node exists then print -1.

**Examples:**

**Input :**

8

/ \

3 10

/ / \

1 6 14

/ \ /

4 7 13

k = 5

**Output :** 6

Diagonal Traversal of the above tree is:

8 10 14

3 6 7 13

1 4

**Input :**

1

/ \

2 3

/ \

4 5

k = 7

**Output :** -1

[**Recommended: Please try your approach on *{IDE}*first, before moving on to the solution.**](https://ide.geeksforgeeks.org/)

**Approach:** The idea is to perform the [diagonal](https://www.geeksforgeeks.org/iterative-diagonal-traversal-binary-tree/) traversal of the binary tree until K nodes are visited in the diagonal traversal. While traversing for each node visited decrement the value of variable **K** and return the current node when the value of K becomes zero. If the diagonal traversal does not contain at least K nodes, return -1.

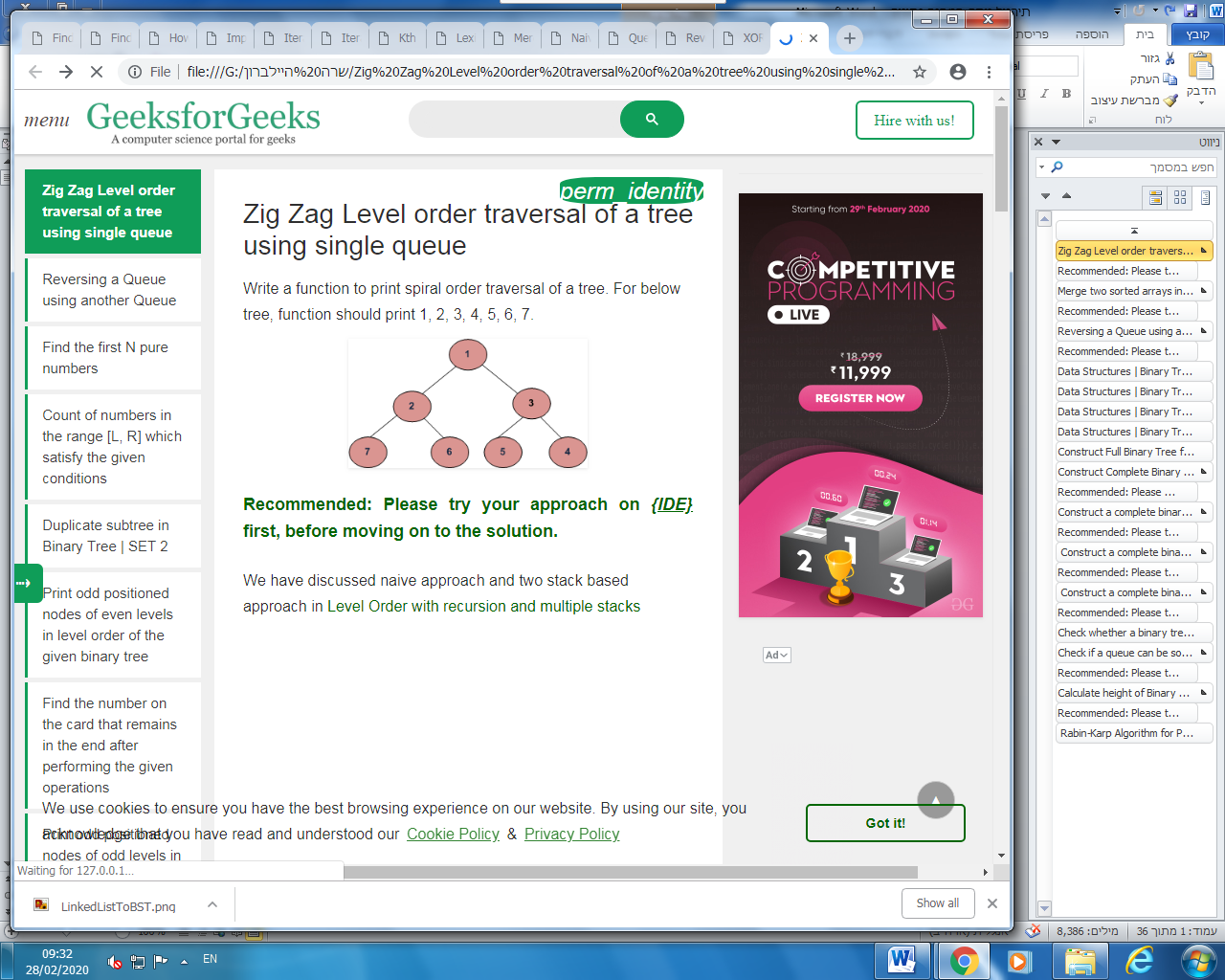
|  |
| --- |
| // C++ program to print kth node  // in the diagonal traversal of a binary tree    #include <bits/stdc++.h>  using namespace std;    // A binary tree node has data, pointer to left  child and a pointer to right child struct Node {      int data;      Node \*left, \*right;  };    // Helper function that allocates a new node  Node\* newNode(int data)  {      Node\* node = (Node\*)malloc(sizeof(Node));      node->data = data;      node->left = node->right = NULL;      return (node);  }    // Iterative function to print kth node  // in diagonal traversal of binary tree  int diagonalPrint(Node\* root, int k)  {      // Base cases      if (root == NULL || k == 0)          return -1;        int ans = -1;      queue<Node\*> q;        // Push root node      q.push(root);        // Push delimiter NULL      q.push(NULL);        while (!q.empty()) {          Node\* temp = q.front();          q.pop();            if (temp == NULL) {              if (q.empty()) {                  // If kth node exists then return                  // the answer                  if (k == 0)                      return ans;                    // If kth node doesnt exists                  // then break from the while loop                  else                      break;              }              q.push(NULL);          }          else {              while (temp) {                  // If the required kth node                  // has been found then return the answer                  if (k == 0)                      return ans;                    k--;                    // Update the value of variable ans                  // each time                  ans = temp->data;                    if (temp->left)                      q.push(temp->left);                    temp = temp->right;              }          }      }        // If kth node doesnt exists then      // return -1      return -1;  }    // Driver Code  int main()  {      Node\* root = newNode(8);      root->left = newNode(3);      root->right = newNode(10);      root->left->left = newNode(1);      root->left->right = newNode(6);      root->right->right = newNode(14);      root->right->right->left = newNode(13);      root->left->right->left = newNode(4);      root->left->right->right = newNode(7);        int k = 9;        cout << diagonalPrint(root, k);        return 0;  } |

**Output:**

4

Zig Zag Level order traversal of a tree using single queue

Write a function to print spiral order traversal of a tree. For below tree, function should print 1, 2, 3, 4, 5, 6, 7.



[**Recommended: Please try your approach on *{IDE}*first, before moving on to the solution.**](https://ide.geeksforgeeks.org/)

We have discussed naive approach and two stack based approach in [Level Order with recursion and multiple stacks](https://www.geeksforgeeks.org/level-order-traversal-in-spiral-form/)

The idea behind this approach is first we have to take a queue, a direction flag and a separation flag which is NULL

1. Insert the root element into the queue and again insert NULL into the queue.
2. For every element in the queue insert its child nodes.
3. If a NULL is encountered then check the direction to traverse the particular level is left to right or right to left. If it’s an even level then traverse from left to right otherwise traverse the tree in right to level order i.e., from the front to the previous front i.e., from the current NULL to to the last NULL that has been visited. This continues till the last level then there the loop breaks and we print what is left (that has not printed) by checking the direction to print.

Following is the implementation of the explanation

|  |
| --- |
| // C++ program to print level order traversal  // in spiral form using a single dequeue  #include <bits/stdc++.h>    struct Node {      int data;      struct Node \*left, \*right;  };    // A utility function to create a new node  struct Node\* newNode(int data)  {      struct Node\* node = new struct Node;      node->data = data;      node->left = node->right = NULL;      return (node);  }    // function to print the level order traversal  void levelOrder(struct Node\* root, int n)  {      // We can just take the size as H+N which      // implies the height of the tree with the      // size of the tree      struct Node\* queue[2 \* n];      int top = -1;      int front = 1;      queue[++top] = NULL;      queue[++top] = root;      queue[++top] = NULL;        // struct Node\* t=root;      int prevFront = 0, count = 1;      while (1) {            struct Node\* curr = queue[front];            // A level separator found          if (curr == NULL) {                // If this is the only item in dequeue              if (front == top)                  break;                // Else print contents of previous level              // according to count              else {                  if (count % 2 == 0) {                      for (int i = prevFront + 1; i < front; i++)                          printf("%d ", queue[i]->data);                  }                  else {                      for (int i = front - 1; i > prevFront; i--)                          printf("%d ", queue[i]->data);                  }                    prevFront = front;                  count++;                  front++;                    // Insert a new level separator                  queue[++top] = NULL;                    continue;              }          }            if (curr->left != NULL)              queue[++top] = curr->left;          if (curr->right != NULL)              queue[++top] = curr->right;          front++;      }        if (count % 2 == 0) {          for (int i = prevFront + 1; i < top; i++)              printf("%d ", queue[i]->data);      }      else {          for (int i = top - 1; i > prevFront; i--)              printf("%d ", queue[i]->data);      }  }    // Driver code  int main()  {      struct Node\* root = newNode(1);      root->left = newNode(2);      root->right = newNode(3);      root->left->left = newNode(7);      root->left->right = newNode(6);      root->right->left = newNode(5);      root->right->right = newNode(4);      levelOrder(root, 7);        return 0;  } |

**Output:**

1 2 3 4 5 6 7

**Time Complexity:** O(n)  
**Auxiliary Space :** O(2\*n) = O(n)

Merge two sorted arrays in constant space using Min Heap

Given two sorted arrays, we need to merge them with O(1) extra space into a sorted array, when N is the size of the first array, and M is the size of the second array.

**Example**:

**Input**: arr1[] = {10};

arr2[] = {2, 3};

**Output**: arr1[] = {2}

arr2[] = {3, 10}

**Input**: arr1[] = {1, 5, 9, 10, 15, 20};

arr2[] = {2, 3, 8, 13};

**Output**: arr1[] = {1, 2, 3, 5, 8, 9}

arr2[] = {10, 13, 15, 20}

[**Recommended: Please try your approach on *{IDE}*first, before moving on to the solution.**](https://ide.geeksforgeeks.org/)

We had already discussed two more approaches to solve the above problem in constant space:

* [Merge two sorted arrays with O(1) extra space](https://www.geeksforgeeks.org/merge-two-sorted-arrays-o1-extra-space/).
* [Efficiently merging two sorted arrays with O(1) extra space](https://www.geeksforgeeks.org/efficiently-merging-two-sorted-arrays-with-o1-extra-space/).

In this article, one more approach using the concept of the heap data structure is discussed without taking any extra space to merge the two sorted arrays.

Below is the detailed approach in steps:

1. The idea is to convert the second array into a min-heap first. This can be done in O(M) time complexity.
2. After converting the second array to min-heap:
   * Start traversing the first array and compare the current element for the first array to top of the created min\_heap.
   * If the current element in the first array is greater than heap top, swap the current element of the first array with the root of the heap, and heapify the root of the min\_heap.
   * After performing the above operation for every element of the first array, the first array will now contain first N elements of the sorted merged array.
3. Now, the elements remained in the min\_heap or the second array are the last M elements of the sorted merged array.
4. To arrange them in sorted order, apply in-place heapsort on the second array.

**Note**: We have used built-in STL functions available in C++ to convert array to min\_heap, sorting the heap etc. It is recommended to read – [Heap in C++ STL | make\_heap(), push\_heap(), pop\_heap(), sort\_heap()](https://www.geeksforgeeks.org/heap-using-stl-c/" \t "_blank) before moving on to the program.

Below is the implementation of the above approach:

|  |
| --- |
| // C++ program to merge two sorted arrays in  // constant space    #include <bits/stdc++.h>  using namespace std;    // Function to merge two sorted arrays in  // constant space  void mergeArrays(int\* a, int n, int\* b, int m)  {        // Convert second array into a min\_heap      // using make\_heap() STL function [takes O(m)]      make\_heap(b, b + m, greater<int>());        // Start traversing the first array      for (int i = 0; i < n; i++) {            // If current element is greater than root          // of min\_heap          if (a[i] > b[0]) {                // Pop minimum element from min\_heap using              // pop\_heap() STL function              // The pop\_heap() function removes the minimum element from              // heap and moves it to the end of the container              // converted to heap and reduces heap size by 1              pop\_heap(b, b + m, greater<int>());                // Swapping the elements              int tmp = a[i];              a[i] = b[m - 1];              b[m - 1] = tmp;                // Apply push\_heap() function on the container              // or array to again reorder it in the              // form of min\_heap              push\_heap(b, b + m, greater<int>());          }      }        // Convert the second array again into max\_heap      // because sort\_heap() on min heap sorts the array      // in decreasing order      // This step is [O(m)]      make\_heap(b, b + m); // It's a max\_heap        // Sort the second array using sort\_heap() function      sort\_heap(b, b + m);  }    // Driver Code  int main()  {        int ar1[] = { 1, 5, 9, 10, 15, 20 };      int ar2[] = { 2, 3, 8, 13 };      int m = sizeof(ar1) / sizeof(ar1[0]);      int n = sizeof(ar2) / sizeof(ar2[0]);      mergeArrays(ar1, m, ar2, n);        cout << "After Merging :- \nFirst Array: ";      for (int i = 0; i < m; i++)          cout << ar1[i] << " ";      cout << "\nSecond Array: ";      for (int i = 0; i < n; i++)          cout << ar2[i] << " ";        return 0;  } |

**Output:**

After Merging :-

First Array: 1 2 3 5 8 9

Second Array: 10 13 15 20

**Time Complexity**: O(N\*logM + M\*logN)  
**Auxiliary Space**: O(1)

Reversing a Queue using another Queue

Given a [queue](http://www.geeksforgeeks.org/queue-data-structure/). The task is to reverse the queue using another another empty queue.

**Examples:**

**Input:** queue[] = {1, 2, 3, 4, 5}

**Output:** 5 4 3 2 1

**Input:** queue[] = {10, 20, 30, 40}

**Output:** 40 30 20 10

[**Recommended: Please try your approach on *{IDE}*first, before moving on to the solution.**](https://ide.geeksforgeeks.org/)

**Approach:**

* Given a queue and an empty queue.
* The last element of the queue should be the first element of the new queue.
* To get the last element there is a need to pop the queue one by one and add it to the end of the queue, **size – 1** times.
* So after that, we will get the last element in front of the queue. Now pop that element out and add it to the new queue. Repeat the steps **s – 1** times where **s** is the original size of the queue.

Below is the implementation of the approach:

|  |
| --- |
| // C++ implementation of the above approach  #include <bits/stdc++.h>  using namespace std;    // Function to return the reversed queue  queue<int> reverse(queue<int> q)  {      // Size of ueue      int s = q.size();        // Second queue      queue<int> ans;        for (int i = 0; i < s; i++) {            // Get the last element to the          // front of queue          for (int j = 0; j < q.size() - 1; j++) {              int x = q.front();              q.pop();              q.push(x);          }            // Get the last element and          // add it to the new queue          ans.push(q.front());          q.pop();      }      return ans;  }    // Driver Code  int main()  {      queue<int> q;        // Insert elements      q.push(1);      q.push(2);      q.push(3);      q.push(4);      q.push(5);        q = reverse(q);        // Print the queue      while (!q.empty()) {          cout << q.front() << " ";          q.pop();      }        return 0;  } |

**Output:**

5 4 3 2 1

# Data Structures | Binary Trees | Question 15

Consider a node X in a Binary Tree. Given that X has two children, let Y be Inorder successor of X. Which of the following is true about Y?  
**(A)** Y has no right child  
**(B)** Y has no left child  
**(C)** Y has both children  
**(D)** None of the above  
  
  
**Answer:** **(B)**   
  
**Explanation:** Since X has both children, Y must be leftmost node in right child of X.

# Data Structures | Binary Trees | Question 14

Consider the following nested representation of binary trees: (X Y Z) indicates Y and Z are the left and right sub stress, respectively, of node X. Note that Y and Z may be NULL, or further nested. Which of the following represents a valid binary tree?  
**(A)** (1 2 (4 5 6 7))  
**(B)** (1 (2 3 4) 5 6) 7)  
**(C)** (1 (2 3 4)(5 6 7))  
**(D)** (1 (2 3 NULL) (4 5))  
  
  
**Answer:** **(C)**   
  
**Explanation:** C is fine.

(1 (2 3 4)(5 6 7)) represents following binary tree

1

/ \

2 5

/ \ / \

3 4 6 7

A) (1 2 (4 5 6 7)) is not fine as there are 4 elements in one bracket.

B) (1 (2 3 4) 5 6) 7) is not fine as there are 2 opening brackets and 3 closing.

D) (1 (2 3 NULL) (4 5)) is not fine one bracket has only two entries (4 5)

Data Structures | Binary Trees | Question 13

Postorder traversal of a given binary search tree, T produces the following sequence of keys  
10, 9, 23, 22, 27, 25, 15, 50, 95, 60, 40, 29  
Which one of the following sequences of keys can be the result of an in-order traversal of the tree T?   
**(A)** 9, 10, 15, 22, 23, 25, 27, 29, 40, 50, 60, 95  
**(B)** 9, 10, 15, 22, 40, 50, 60, 95, 23, 25, 27, 29  
**(C)** 29, 15, 9, 10, 25, 22, 23, 27, 40, 60, 50, 95  
**(D)** 95, 50, 60, 40, 27, 23, 22, 25, 10, 9, 15, 29  
  
  
**Answer:** **(A)**   
  
**Explanation:** Inorder traversal of a [BST](http://en.wikipedia.org/wiki/Binary_search_tree)always gives elements in increasing order. Among all four options, a) is the only increasing order sequence.

Data Structures | Binary Trees | Question 1

Which of the following is a true about Binary Trees  
**(A)** Every binary tree is either complete or full.  
**(B)** Every complete binary tree is also a full binary tree.  
**(C)** Every full binary tree is also a complete binary tree.  
**(D)** No binary tree is both complete and full.  
**(E)** None of the above  
  
  
**Answer:** **(E)**   
  
**Explanation:** A full binary tree (sometimes proper binary tree or 2-tree or strictly binary tree) is a tree in which every node other than the leaves has two children.

A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

A) is incorrect. For example, the following Binary tree is neither complete nor full

12

/

20

/

30

B) is incorrect. The following binary tree is complete but not full

12

/ \

20 30

/

30

C) is incorrect. Following Binary tree is full, but not complete

12

/ \

20 30

/ \

20 40

D) is incorrect. Following Binary tree is both complete and full

12

/ \

20 30

/ \

10 40

Construct Full Binary Tree from given preorder and postorder traversals

Given two arrays that represent preorder and postorder traversals of a full binary tree, construct the binary tree.

A **Full Binary Tree** is a binary tree where every node has either 0 or 2 children

Following are examples of Full Trees.

1

/ \

2 3

/ \ / \

4 5 6 7

1

/ \

2 3

/ \

4 5

/ \

6 7

1

/ \

2 3

/ \ / \

4 5 6 7

/ \

8 9

It is not possible to construct a general Binary Tree from preorder and postorder traversals (See [this](https://www.geeksforgeeks.org/archives/657)). But if know that the Binary Tree is Full, we can construct the tree without ambiguity. Let us understand this with the help of following example.

Let us consider the two given arrays as pre[] = {1, 2, 4, 8, 9, 5, 3, 6, 7} and post[] = {8, 9, 4, 5, 2, 6, 7, 3, 1};  
In pre[], the leftmost element is root of tree. Since the tree is full and array size is more than 1. The value next to 1 in pre[], must be left child of root. So we know 1 is root and 2 is left child. How to find the all nodes in left subtree? We know 2 is root of all nodes in left subtree. All nodes before 2 in post[] must be in left subtree. Now we know 1 is root, elements {8, 9, 4, 5, 2} are in left subtree, and the elements {6, 7, 3} are in right subtree.

1

/ \

/ \

{8, 9, 4, 5, 2} {6, 7, 3}

We recursively follow the above approach and get the following tree.

1

/ \

2 3

/ \ / \

4 5 6 7

/ \

8 9

|  |
| --- |
| /\* program for construction of full binary tree \*/  #include <bits/stdc++.h>  using namespace std;    /\* A binary tree node has data, pointer to left child  and a pointer to right child \*/  class node  {      public:      int data;      node \*left;      node \*right;  };    // A utility function to create a node  node\* newNode (int data)  {      node\* temp = new node();        temp->data = data;      temp->left = temp->right = NULL;        return temp;  }    // A recursive function to construct Full from pre[] and post[].  // preIndex is used to keep track of index in pre[].  // l is low index and h is high index for the current subarray in post[]  node\* constructTreeUtil (int pre[], int post[], int\* preIndex,                                  int l, int h, int size)  {      // Base case      if (\*preIndex >= size || l > h)          return NULL;        // The first node in preorder traversal is root. So take the node at      // preIndex from preorder and make it root, and increment preIndex      node\* root = newNode ( pre[\*preIndex] );      ++\*preIndex;        // If the current subarry has only one element, no need to recur      if (l == h)          return root;        // Search the next element of pre[] in post[]      int i;      for (i = l; i <= h; ++i)          if (pre[\*preIndex] == post[i])              break;        // Use the index of element found in postorder to divide          // postorder array in two parts. Left subtree and right subtree      if (i <= h)      {          root->left = constructTreeUtil (pre, post, preIndex,                                                  l, i, size);          root->right = constructTreeUtil (pre, post, preIndex,                                                   i + 1, h, size);      }        return root;  }    // The main function to construct Full Binary Tree from given preorder and  // postorder traversals. This function mainly uses constructTreeUtil()  node \*constructTree (int pre[], int post[], int size)  {      int preIndex = 0;      return constructTreeUtil (pre, post, &preIndex, 0, size - 1, size);  }    // A utility function to print inorder traversal of a Binary Tree  void printInorder (node\* node)  {      if (node == NULL)          return;      printInorder(node->left);      cout<<node->data<<" ";      printInorder(node->right);  }    // Driver program to test above functions  int main ()  {      int pre[] = {1, 2, 4, 8, 9, 5, 3, 6, 7};      int post[] = {8, 9, 4, 5, 2, 6, 7, 3, 1};      int size = sizeof( pre ) / sizeof( pre[0] );        node \*root = constructTree(pre, post, size);        cout<<"Inorder traversal of the constructed tree: \n";      printInorder(root);        return 0;  }    //This code is contributed by rathbhupendra |

**Output:**

Inorder traversal of the constructed tree:

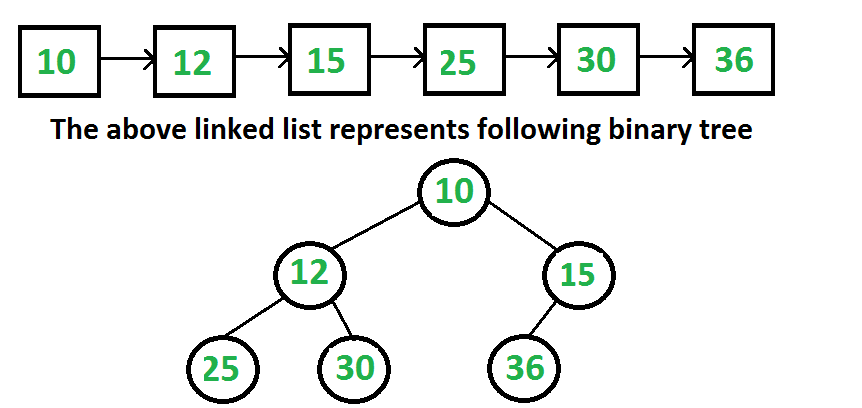
8 4 9 2 5 1 6 3 7

**Improved By :**[shrikanth13](https://auth.geeksforgeeks.org/user/shrikanth13/), [rathbhupendra](https://auth.geeksforgeeks.org/user/rathbhupendra/)

Construct Complete Binary Tree from its Linked List Representation

Given Linked List Representation of Complete Binary Tree, construct the Binary tree. A complete binary tree can be represented in an array in the following approach.

If root node is stored at index i, its left, and right children are stored at indices 2\*i+1, 2\*i+2 respectively.  
Suppose tree is represented by a linked list in same way, how do we convert this into normal linked representation of binary tree where every node has data, left and right pointers? In the linked list representation, we cannot directly access the children of the current node unless we traverse the list.



**[Recommended: Please solve it on “](https://practice.geeksforgeeks.org/problems/make-binary-tree/1)*[PRACTICE](https://practice.geeksforgeeks.org/problems/make-binary-tree/1)*[” first, before moving on to the solution.](https://practice.geeksforgeeks.org/problems/make-binary-tree/1)**

We are mainly given level order traversal in sequential access form. We know head of linked list is always is root of the tree. We take the first node as root and we also know that the next two nodes are left and right children of root. So we know partial Binary Tree. The idea is to do Level order traversal of the partially built Binary Tree using queue and traverse the linked list at the same time. At every step, we take the parent node from queue, make next two nodes of linked list as children of the parent node, and enqueue the next two nodes to queue.

**1.** Create an empty queue.  
**2.** Make the first node of the list as root, and enqueue it to the queue.  
**3.**Until we reach the end of the list, do the following.  
………**a.** Dequeue one node from the queue. This is the current parent.  
………**b.** Traverse two nodes in the list, add them as children of the current parent.  
………**c.** Enqueue the two nodes into the queue.

Below is the code which implements the same in C++.

|  |
| --- |
| // C++ program to create a Complete Binary tree from its Linked List  // Representation  #include <iostream>  #include <string>  #include <queue>  using namespace std;    // Linked list node  struct ListNode  {      int data;      ListNode\* next;  };    // Binary tree node structure  struct BinaryTreeNode  {      int data;      BinaryTreeNode \*left, \*right;  };    // Function to insert a node at the beginning of the Linked List  void push(struct ListNode\*\* head\_ref, int new\_data)  {      // allocate node and assign data      struct ListNode\* new\_node = new ListNode;      new\_node->data = new\_data;        // link the old list off the new node      new\_node->next = (\*head\_ref);        // move the head to point to the new node      (\*head\_ref)    = new\_node;  }    // method to create a new binary tree node from the given data  BinaryTreeNode\* newBinaryTreeNode(int data)  {      BinaryTreeNode \*temp = new BinaryTreeNode;      temp->data = data;      temp->left = temp->right = NULL;      return temp;  }    // converts a given linked list representing a complete binary tree into the  // linked representation of binary tree.  void convertList2Binary(ListNode \*head, BinaryTreeNode\* &root)  {      // queue to store the parent nodes      queue<BinaryTreeNode \*> q;        // Base Case      if (head == NULL)      {          root = NULL; // Note that root is passed by reference          return;      }        // 1.) The first node is always the root node, and add it to the queue      root = newBinaryTreeNode(head->data);      q.push(root);        // advance the pointer to the next node      head = head->next;        // until the end of linked list is reached, do the following steps      while (head)      {          // 2.a) take the parent node from the q and remove it from q          BinaryTreeNode\* parent = q.front();          q.pop();            // 2.c) take next two nodes from the linked list. We will add          // them as children of the current parent node in step 2.b. Push them          // into the queue so that they will be parents to the future nodes          BinaryTreeNode \*leftChild = NULL, \*rightChild = NULL;          leftChild = newBinaryTreeNode(head->data);          q.push(leftChild);          head = head->next;          if (head)          {              rightChild = newBinaryTreeNode(head->data);              q.push(rightChild);              head = head->next;          }            // 2.b) assign the left and right children of parent          parent->left = leftChild;          parent->right = rightChild;      }  }    // Utility function to traverse the binary tree after conversion  void inorderTraversal(BinaryTreeNode\* root)  {      if (root)      {          inorderTraversal( root->left );          cout << root->data << " ";          inorderTraversal( root->right );      }  }    // Driver program to test above functions  int main()  {      // create a linked list shown in above diagram      struct ListNode\* head = NULL;      push(&head, 36);  /\* Last node of Linked List \*/      push(&head, 30);      push(&head, 25);      push(&head, 15);      push(&head, 12);      push(&head, 10); /\* First node of Linked List \*/        BinaryTreeNode \*root;      convertList2Binary(head, root);        cout << "Inorder Traversal of the constructed Binary Tree is: \n";      inorderTraversal(root);      return 0;  } |

**Output:**

Inorder Traversal of the constructed Binary Tree is:

25 12 30 10 36 15

**Time Complexity:**Time complexity of the above solution is O(n) where n is the number of nodes.

This article is compiled by **Ravi Chandra Enaganti**

Construct a complete binary tree from given array in level order fashion

Given an array of elements, our task is to construct a complete binary tree from this array in level order fashion. That is, elements from left in the array will be filled in the tree level wise starting from level 0.

**Examples:**

Input : arr[] = {1, 2, 3, 4, 5, 6}

Output : Root of the following tree

1

/ \

2 3

/ \ /

4 5 6

Input: arr[] = {1, 2, 3, 4, 5, 6, 6, 6, 6, 6}

Output: Root of the following tree

1

/ \

2 3

/ \ / \

4 5 6 6

/ \ /

6 6 6

**[Recommended: Please try your approach on](https://ide.geeksforgeeks.org/)*[{IDE}](https://ide.geeksforgeeks.org/)*[first, before moving on to the solution.](https://ide.geeksforgeeks.org/)**

If we observe carefully we can see that if parent node is at index i in the array then the left child of that node is at index (2\*i + 1) and right child is at index (2\*i + 2) in the array.  
Using this concept, we can easily insert the left and right nodes by choosing its parent node. We will insert the first element present in the array as the root node at level 0 in the tree and start traversing the array and for every node i we will insert its both childs left and right in the tree.  
Below is the recursive program to do this:

* C++
* Java
* Python3
* C#

*filter\_none*

*edit*

*play\_arrow*

*brightness\_4*

|  |
| --- |
| // CPP program to construct binary  // tree from given array in level  // order fashion Tree Node  #include <bits/stdc++.h>  using namespace std;    /\* A binary tree node has data,  pointer to left child and a  pointer to right child \*/  struct Node  {      int data;      Node\* left, \* right;  };    /\* Helper function that allocates a  new node \*/  Node\* newNode(int data)  {      Node\* node = (Node\*)malloc(sizeof(Node));      node->data = data;      node->left = node->right = NULL;      return (node);  }    // Function to insert nodes in level order  Node\* insertLevelOrder(int arr[], Node\* root,                         int i, int n)  {      // Base case for recursion      if (i < n)      {          Node\* temp = newNode(arr[i]);          root = temp;            // insert left child          root->left = insertLevelOrder(arr,                     root->left, 2 \* i + 1, n);            // insert right child          root->right = insertLevelOrder(arr,                    root->right, 2 \* i + 2, n);      }      return root;  }    // Function to print tree nodes in  // InOrder fashion  void inOrder(Node\* root)  {      if (root != NULL)      {          inOrder(root->left);          cout << root->data <<" ";          inOrder(root->right);      }  }    // Driver program to test above function  int main()  {      int arr[] = { 1, 2, 3, 4, 5, 6, 6, 6, 6 };      int n = sizeof(arr)/sizeof(arr[0]);      Node\* root = insertLevelOrder(arr, root, 0, n);      inOrder(root);  }    // This code is contributed by Chhavi |

**Output:**

6 4 6 2 5 1 6 3 6

**Time Complexity**: O(n), where n is the total number of nodes in the tree.

This article is contributed by **[Haribalaji R](https://www.facebook.com/haribalaji.ravi)**. If you like GeeksforGeeks and would like to contribute, you can also write an article using [contribute.geeksforgeeks.org](http://www.contribute.geeksforgeeks.org/) or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

# Construct a complete binary tree from given array in level order fashion

Given an array of elements, our task is to construct a complete binary tree from this array in level order fashion. That is, elements from left in the array will be filled in the tree level wise starting from level 0.

**Examples:**

Input : arr[] = {1, 2, 3, 4, 5, 6}

Output : Root of the following tree

1

/ \

2 3

/ \ /

4 5 6

Input: arr[] = {1, 2, 3, 4, 5, 6, 6, 6, 6, 6}

Output: Root of the following tree

1

/ \

2 3

/ \ / \

4 5 6 6

/ \ /

6 6 6

**[Recommended: Please try your approach on](https://ide.geeksforgeeks.org/)*[{IDE}](https://ide.geeksforgeeks.org/)*[first, before moving on to the solution.](https://ide.geeksforgeeks.org/)**

If we observe carefully we can see that if parent node is at index i in the array then the left child of that node is at index (2\*i + 1) and right child is at index (2\*i + 2) in the array.  
Using this concept, we can easily insert the left and right nodes by choosing its parent node. We will insert the first element present in the array as the root node at level 0 in the tree and start traversing the array and for every node i we will insert its both childs left and right in the tree.  
Below is the recursive program to do this:

* C++
* Java
* Python3
* C#

*filter\_none*

*edit*

*play\_arrow*

*brightness\_4*

|  |
| --- |
| // CPP program to construct binary  // tree from given array in level  // order fashion Tree Node  #include <bits/stdc++.h>  using namespace std;    /\* A binary tree node has data,  pointer to left child and a  pointer to right child \*/  struct Node  {      int data;      Node\* left, \* right;  };    /\* Helper function that allocates a  new node \*/  Node\* newNode(int data)  {      Node\* node = (Node\*)malloc(sizeof(Node));      node->data = data;      node->left = node->right = NULL;      return (node);  }    // Function to insert nodes in level order  Node\* insertLevelOrder(int arr[], Node\* root,                         int i, int n)  {      // Base case for recursion      if (i < n)      {          Node\* temp = newNode(arr[i]);          root = temp;            // insert left child          root->left = insertLevelOrder(arr,                     root->left, 2 \* i + 1, n);            // insert right child          root->right = insertLevelOrder(arr,                    root->right, 2 \* i + 2, n);      }      return root;  }    // Function to print tree nodes in  // InOrder fashion  void inOrder(Node\* root)  {      if (root != NULL)      {          inOrder(root->left);          cout << root->data <<" ";          inOrder(root->right);      }  }    // Driver program to test above function  int main()  {      int arr[] = { 1, 2, 3, 4, 5, 6, 6, 6, 6 };      int n = sizeof(arr)/sizeof(arr[0]);      Node\* root = insertLevelOrder(arr, root, 0, n);      inOrder(root);  }    // This code is contributed by Chhavi |

**Output:**

6 4 6 2 5 1 6 3 6

**Time Complexity**: O(n), where n is the total number of nodes in the tree.

This article is contributed by **[Haribalaji R](https://www.facebook.com/haribalaji.ravi)**. If you like GeeksforGeeks and would like to contribute, you can also write an article using [contribute.geeksforgeeks.org](http://www.contribute.geeksforgeeks.org/) or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

# Construct a complete binary tree from given array in level order fashion

Given an array of elements, our task is to construct a complete binary tree from this array in level order fashion. That is, elements from left in the array will be filled in the tree level wise starting from level 0.

**Examples:**

Input : arr[] = {1, 2, 3, 4, 5, 6}

Output : Root of the following tree

1

/ \

2 3

/ \ /

4 5 6

Input: arr[] = {1, 2, 3, 4, 5, 6, 6, 6, 6, 6}

Output: Root of the following tree

1

/ \

2 3

/ \ / \

4 5 6 6

/ \ /

6 6 6

**[Recommended: Please try your approach on](https://ide.geeksforgeeks.org/)*[{IDE}](https://ide.geeksforgeeks.org/)*[first, before moving on to the solution.](https://ide.geeksforgeeks.org/)**

If we observe carefully we can see that if parent node is at index i in the array then the left child of that node is at index (2\*i + 1) and right child is at index (2\*i + 2) in the array.  
Using this concept, we can easily insert the left and right nodes by choosing its parent node. We will insert the first element present in the array as the root node at level 0 in the tree and start traversing the array and for every node i we will insert its both childs left and right in the tree.  
Below is the recursive program to do this:

|  |
| --- |
| // CPP program to construct binary  // tree from given array in level  // order fashion Tree Node  #include <bits/stdc++.h>  using namespace std;    /\* A binary tree node has data,  pointer to left child and a  pointer to right child \*/  struct Node  {      int data;      Node\* left, \* right;  };    /\* Helper function that allocates a  new node \*/  Node\* newNode(int data)  {      Node\* node = (Node\*)malloc(sizeof(Node));      node->data = data;      node->left = node->right = NULL;      return (node);  }    // Function to insert nodes in level order  Node\* insertLevelOrder(int arr[], Node\* root,                         int i, int n)  {      // Base case for recursion      if (i < n)      {          Node\* temp = newNode(arr[i]);          root = temp;            // insert left child          root->left = insertLevelOrder(arr,                     root->left, 2 \* i + 1, n);            // insert right child          root->right = insertLevelOrder(arr,                    root->right, 2 \* i + 2, n);      }      return root;  }    // Function to print tree nodes in  // InOrder fashion  void inOrder(Node\* root)  {      if (root != NULL)      {          inOrder(root->left);          cout << root->data <<" ";          inOrder(root->right);      }  }    // Driver program to test above function  int main()  {      int arr[] = { 1, 2, 3, 4, 5, 6, 6, 6, 6 };      int n = sizeof(arr)/sizeof(arr[0]);      Node\* root = insertLevelOrder(arr, root, 0, n);      inOrder(root);  }    // This code is contributed by Chhavi |

**Output:**

6 4 6 2 5 1 6 3 6

**Time Complexity**: O(n), where n is the total number of nodes in the tree.

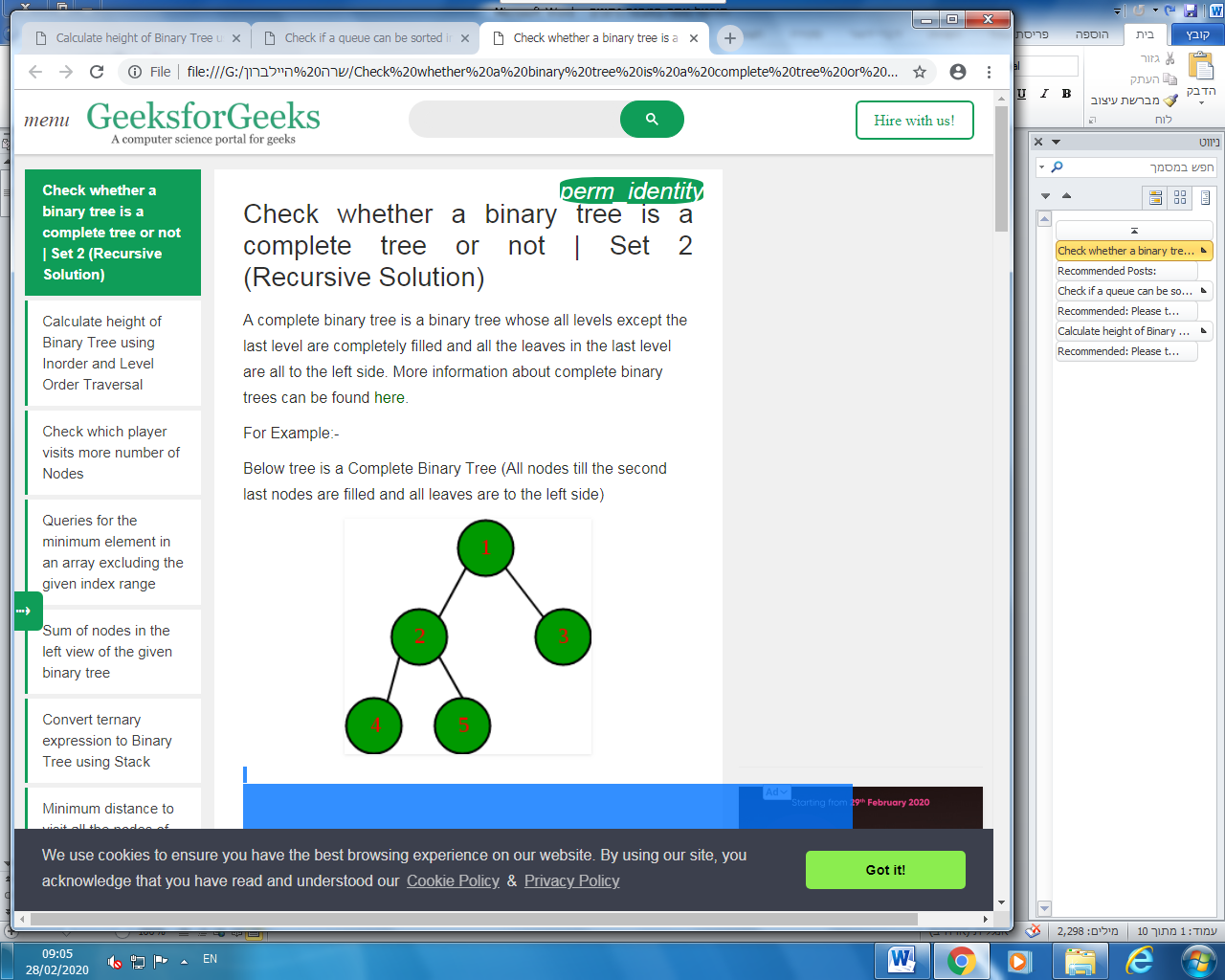
This article is contributed by **[Haribalaji R](https://www.facebook.com/haribalaji.ravi)**.

Check whether a binary tree is a complete tree or not | Set 2 (Recursive Solution)

A complete binary tree is a binary tree whose all levels except the last level are completely filled and all the leaves in the last level are all to the left side. More information about complete binary trees can be found [here](http://courses.cs.vt.edu/~cs3114/Fall09/wmcquain/Notes/T03a.BinaryTreeTheorems.pdf).

For Example:-

Below tree is a Complete Binary Tree (All nodes till the second last nodes are filled and all leaves are to the left side)



An iterative solution for this problem is discussed in below post.  
[Check whether a given Binary Tree is Complete or not | Set 1 (Using Level Order Traversal)](https://www.geeksforgeeks.org/check-if-a-given-binary-tree-is-complete-tree-or-not/)

In this post a recursive solution is discussed.

In the array representation of a binary tree, if the parent node is assigned an index of ‘i’ and left child gets assigned an index of ‘2\*i + 1’ while the right child is assigned an index of ‘2\*i + 2’. If we represent the above binary tree as an array with the respective indices assigned to the different nodes of the tree above from top to down and left to right.

Hence we proceed in the following manner in order to check if the binary tree is complete binary tree.

1. Calculate the number of nodes (count) in the binary tree.
2. Start recursion of the binary tree from the root node of the binary tree with index (i) being set as 0 and the number of nodes in the binary (count).
3. If the current node under examination is NULL, then the tree is a complete binary tree. Return true.
4. If index (i) of the current node is greater than or equal to the number of nodes in the binary tree (count) i.e. (i>= count), then the tree is not a complete binary. Return false.
5. Recursively check the left and right sub-trees of the binary tree for same condition. For the left sub-tree use the index as (2\*i + 1) while for the right sub-tree use the index as (2\*i + 2).

The time complexity of the above algorithm is O(n). Following is the code for checking if a binary tree is a complete binary tree.

|  |
| --- |
| /\* C++ program to checks if a binary tree complete ot not \*/  #include<bits/stdc++.h>  #include<stdbool.h>  using namespace std;    /\* Tree node structure \*/  class Node  {      public:      int key;      Node \*left, \*right;        Node \*newNode(char k)      {          Node \*node = ( Node\*)malloc(sizeof( Node));          node->key = k;          node->right = node->left = NULL;          return node;      }    };    /\* Helper function that allocates a new node with the  given key and NULL left and right pointer. \*/      /\* This function counts the number of nodes  in a binary tree \*/  unsigned int countNodes(Node\* root)  {      if (root == NULL)          return (0);      return (1 + countNodes(root->left) +              countNodes(root->right));  }    /\* This function checks if the binary tree  is complete or not \*/  bool isComplete ( Node\* root, unsigned int index,                      unsigned int number\_nodes)  {      // An empty tree is complete      if (root == NULL)          return (true);        // 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));  }    // Driver code  int main()  {      Node n1;        // Let us create tree in the last diagram above      Node\* root = NULL;      root = n1.newNode(1);      root->left = n1.newNode(2);      root->right = n1.newNode(3);      root->left->left = n1.newNode(4);      root->left->right = n1.newNode(5);      root->right->right = n1.newNode(6);        unsigned int node\_count = countNodes(root);      unsigned int index = 0;        if (isComplete(root, index, node\_count))          cout << "The Binary Tree is complete\n";      else          cout << "The Binary Tree is not complete\n";      return (0);  }    // This code is contributed by SoumikMondal |

**Output:**

The Binary Tree is not complete

This article is contributed by **Gaurav Gupta**.

Check if a queue can be sorted into another queue using a stack

Given a Queue consisting of first **n** natural numbers (in random order). The task is to check whether the given Queue elements can be arranged in increasing order in another Queue using a stack. The operation allowed are:  
1. Push and pop elements from the stack  
2. Pop (Or enqueue) from the given Queue.  
3. Push (Or Dequeue) in the another Queue.

**Examples :**

*Input : Queue[] = { 5, 1, 2, 3, 4 }  
Output : Yes  
Pop the first element of the given Queue i.e 5.  
Push 5 into the stack.  
Now, pop all the elements of the given Queue and push them to  
second Queue.  
Now, pop element 5 in the stack and push it to the second Queue.  
   
Input : Queue[] = { 5, 1, 2, 6, 3, 4 }  
Output : No  
Push 5 to stack.  
Pop 1, 2 from given Queue and push it to another Queue.  
Pop 6 from given Queue and push to stack.  
Pop 3, 4 from given Queue and push to second Queue.  
Now, from using any of above operation, we cannot push 5  
into the second Queue because it is below the 6 in the stack.*

**[Recommended: Please try your approach on](https://ide.geeksforgeeks.org/)*[{IDE}](https://ide.geeksforgeeks.org/)*[first, before moving on to the solution.](https://ide.geeksforgeeks.org/)**

Observe, second Queue (which will contain the sorted element) takes inputs (or enqueue elements) either from given Queue or Stack. So, next expected (which will initially be 1) element must be present as a front element of given Queue or top element of the Stack. So, simply simulate the process for the second Queue by initializing the expected element as 1. And check if we can get expected element from the front of the given Queue or from the top of the Stack. If we cannot take it from the either of them then pop the front element of given Queue and push it in the Stack.  
Also, observe, that the stack must also be sorted at each instance i.e the element at the top of the stack must be smallest in the stack. For eg. let x > y, then x will always be expected before y. So, x cannot be pushed before y in the stack. Therefore, we cannot push element with the higher value on the top of the element having lesser value.

Algorithm:  
1. Initialize the expected\_element = 1  
2. Check if either front element of given Queue or top element of the stack have expected\_element  
….a) If yes, increment expected\_element by 1, repeat step 2.  
….b) Else, pop front of Queue and push it to the stack. If the popped element is greater than top of the Stack, return “No”.

Below is the implementation of this approach:

|  |
| --- |
| // CPP Program to check if a queue of first  // n natural number can be sorted using a stack  #include <bits/stdc++.h>  using namespace std;    // Function to check if given queue element  // can be sorted into another queue using a  // stack.  bool checkSorted(int n, queue<int>& q)  {      stack<int> st;      int expected = 1;      int fnt;        // while given Queue is not empty.      while (!q.empty()) {          fnt = q.front();          q.pop();            // if front element is the expected element          if (fnt == expected)              expected++;            else {              // if stack is empty, push the element              if (st.empty()) {                  st.push(fnt);              }                // if top element is less than element which              // need to be pushed, then return fasle.              else if (!st.empty() && st.top() < fnt) {                  return false;              }                // else push into the stack.              else                  st.push(fnt);          }            // while expected element are coming from          // stack, pop them out.          while (!st.empty() && st.top() == expected) {              st.pop();              expected++;          }      }        // if the final expected element value is equal      // to initial Queue size and the stack is empty.      if (expected - 1 == n && st.empty())          return true;        return false;  }    // Driven Program  int main()  {      queue<int> q;      q.push(5);      q.push(1);      q.push(2);      q.push(3);      q.push(4);        int n = q.size();        (checkSorted(n, q) ? (cout << "Yes") :                           (cout << "No"));        return 0;  } |

**Output :**

Yes

Video Contributed by [Parul Shandilya](https://auth.geeksforgeeks.org/user/ParulShandilya/practice/)

Calculate height of Binary Tree using Inorder and Level Order Traversal

Given Inorder traversal and Level Order traversal of a Binary Tree. The task is to calculate the height of the tree without constructing it.

**Example:**

**Input :** Input: Two arrays that represent Inorder

and level order traversals of a

Binary Tree

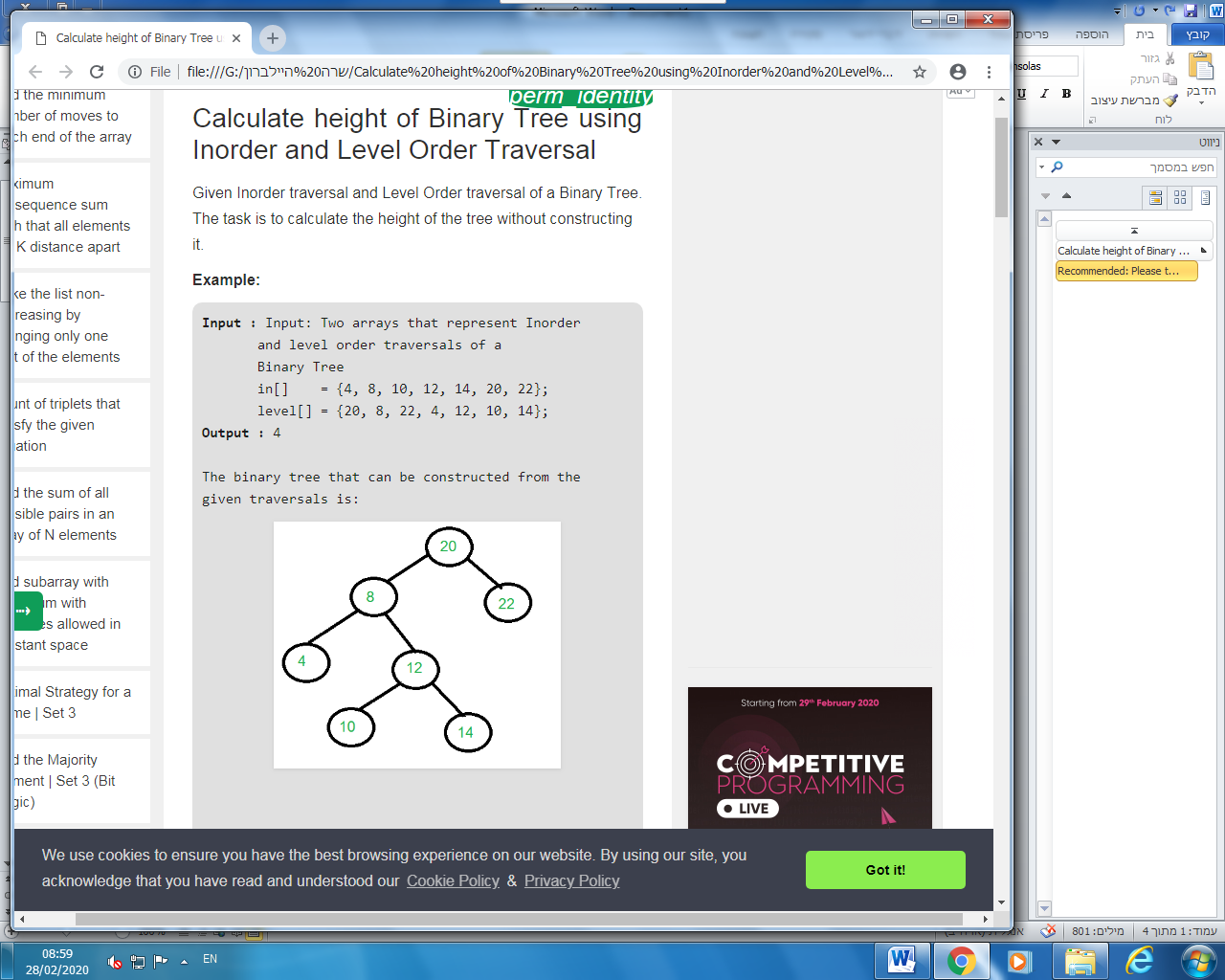
in[] = {4, 8, 10, 12, 14, 20, 22};

level[] = {20, 8, 22, 4, 12, 10, 14};

**Output :** 4

The binary tree that can be constructed from the

given traversals is:



We can clearly see in the above image that the

height of the tree is 4.

[**Recommended: Please try your approach on *{IDE}*first, before moving on to the solution.**](https://ide.geeksforgeeks.org/)

The approach to calculating height is similar to the approach discussed in the post [Constructing Tree from Inorder and Level Order Traversals](https://www.geeksforgeeks.org/construct-tree-inorder-level-order-traversals/).

Let us consider the above example.

in[] = {4, 8, 10, 12, 14, 20, 22};  
level[] = {20, 8, 22, 4, 12, 10, 14};

In a Levelorder sequence, the first element is the root of the tree. So we know ’20’ is root for given sequences. By searching ’20’ in Inorder sequence, we can find out all elements on the left side of ‘20’ are in left subtree and elements on right are in the right subtree. So we know below structure now.

20

/ \

/ \

{4, 8, 10, 12, 14} {22}

Let us call {4, 8, 10, 12, 14} as left subarray in Inorder traversal and {22} as right subarray in Inorder traversal.  
In level order traversal, keys of left and right subtrees are not consecutive. So we extract all nodes from level order traversal which are in left subarray of Inorder traversal. To calculate the height of the left subtree of the root, we recur for the extracted elements from level order traversal and left subarray of inorder traversal. In the above example, we recur for the following two arrays.

// Recur for following arrays to

// calculate the height of the left subtree

In[] = {4, 8, 10, 12, 14}

level[] = {8, 4, 12, 10, 14}

Similarly, we recur for following two arrays and calculate the height of the right subtree.

// Recur for following arrays to calculate

// height of the right subtree

In[] = {22}

level[] = {22}

Below is the implementation of the above approach:

|  |
| --- |
| // C++ program to caulate height of Binary Tree  // from InOrder and LevelOrder Traversals  #include <iostream>  using namespace std;    /\* Function to find index of value     in the InOrder Traversal array \*/  int search(int arr[], int strt, int end, int value)  {      for (int i = strt; i <= end; i++)          if (arr[i] == value)              return i;      return -1;  }    // Function to calculate the height  // of the Binary Tree  int getHeight(int in[], int level[], int start,                int end, int& height, int n)  {        // Base Case      if (start > end)          return 0;        // Get index of current root in InOrder Traversal      int getIndex = search(in, start, end, level[0]);        if (getIndex == -1)          return 0;        // Count elements in Left Subtree      int leftCount = getIndex - start;        // Count elements in right Subtree      int rightCount = end - getIndex;        // Declare two arrays for left and      // right subtrees      int\* newLeftLevel = new int[leftCount];      int\* newRightLevel = new int[rightCount];        int lheight = 0, rheight = 0;      int k = 0;        // Extract values from level order traversal array      // for current left subtree      for (int i = 0; i < n; i++) {          for (int j = start; j < getIndex; j++) {              if (level[i] == in[j]) {                  newLeftLevel[k] = level[i];                  k++;                  break;              }          }      }        k = 0;        // Extract values from level order traversal array      // for current right subtree      for (int i = 0; i < n; i++) {          for (int j = getIndex + 1; j <= end; j++) {              if (level[i] == in[j]) {                  newRightLevel[k] = level[i];                  k++;                  break;              }          }      }        // Recursively call to calculate height of left Subtree      if (leftCount > 0)          lheight = getHeight(in, newLeftLevel, start,                              getIndex - 1, height, leftCount);        // Recursively call to calculate height of right Subtree      if (rightCount > 0)          rheight = getHeight(in, newRightLevel,                              getIndex + 1, end, height, rightCount);        // Current height      height = max(lheight + 1, rheight + 1);        // Delete Auxiliary arrays      delete[] newRightLevel;      delete[] newLeftLevel;        // return height      return height;  }    // Driver program to test above functions  int main()  {      int in[] = { 4, 8, 10, 12, 14, 20, 22 };      int level[] = { 20, 8, 22, 4, 12, 10, 14 };      int n = sizeof(in) / sizeof(in[0]);        int h = 0;        cout << getHeight(in, level, 0, n - 1, h, n);        return 0;  } |

**Output**:

4

# 

Reversing a queue using recursion

Given a queue, write a recursive function to reverse it.  
**Standard operations allowed :**  
enqueue(x) : Add an item x to rear of queue.  
dequeue() : Remove an item from front of queue.  
empty() : Checks if a queue is empty or not.

**Examples :**

Input : Q = [5, 24, 9, 6, 8, 4, 1, 8, 3, 6]

Output : Q = [6, 3, 8, 1, 4, 8, 6, 9, 24, 5]

**Explanation :** Output queue is the reverse of the input queue.

Input : Q = [8, 7, 2, 5, 1]

Output : Q = [1, 5, 2, 7, 8]

**Recursive Algorithm :**

1. The pop element from the queue if the queue has elements otherwise return empty queue.
2. Call reverseQueue function for the remaining queue.
3. Push the popped element in the resultant reversed queue.

**Pseudo Code :**

queue reverseFunction(queue)

{

if (queue is empty)

return queue;

else {

data = queue.front()

queue.pop()

queue = reverseFunction(queue);

q.push(data);

return queue;

}

}

**Time Complexity :** O(n).

Iterative Preorder Traversal of an N-ary Tree

Given a K-ary Tree. The task is to write an iterative program to perform the [preorder traversal](https://www.geeksforgeeks.org/iterative-preorder-traversal/) of the given n-ary tree.

**Examples:**

**Input:** 3-Array Tree

1

/ | \

/ | \

2 3 4

/ \ / | \

5 6 7 8 9

/ / | \

10 11 12 13

**Output:** 1 2 5 10 6 11 12 13 3 4 7 8 9

**Input:** 3-Array Tree

1

/ | \

/ | \

2 3 4

/ \ / | \

5 6 7 8 9

**Output:** 1 2 5 6 3 4 7 8 9

[**Recommended: Please try your approach on *{IDE}*first, before moving on to the solution.**](https://ide.geeksforgeeks.org/)

Preorder Traversal of an N-ary Tree is similar to the preorder traversal of Binary Search Tree or Binary Tree with the only difference that is, all the child nodes of a parent are traversed from left to right in a sequence.

[Iterative Preorder Traversal of Binary Tree](https://www.geeksforgeeks.org/iterative-preorder-traversal/).

**Cases to handle during traversal:** Two Cases have been taken care of in this Iterative Preorder Traversal Algorithm:

1. If Top of the stack is a leaf node then remove it from the stack
2. If Top of the stack is Parent with children:
   * As soon as an unvisited child is found(left to right sequence), Push it to Stack and Store it in  
     Auxillary List and mark the following child as visited.Then, start again from Case-1, to explore this newly visited child.
   * If all Child nodes from left to right of a Parent has been visited then remove the Parent from  
     the stack.

**Note**: In the below python implementation, a “dequeue” is used to implement the stack instead of a list because of its efficient append and pop operations.

|  |
| --- |
| from collections import deque    # Node Structure of K-ary Tree  class NewNode():        def \_\_init\_\_(self, val):          self.key = val          # all children are stored in a list          self.child =[]      # Utility function to print the  # preorder of the given K-Ary Tree  def preorderTraversal(root):        Stack = deque([])      # 'Preorder'-> contains all the      # visited nodes.      Preorder =[]      Preorder.append(root.key)      Stack.append(root)      while len(Stack)>0:          # 'Flag' checks whether all the child          # nodes have been visited.          flag = 0          # CASE 1- If Top of the stack is a leaf          # node then remove it from the stack:          if len((Stack[len(Stack)-1]).child)== 0:              X = Stack.pop()              # CASE 2- If Top of the stack is              # Parent with children:          else:              Par = Stack[len(Stack)-1]          # a)As soon as an unvisited child is          # found(left to right sequence),          # Push it to Stack and Store it in          # Auxillary List(Marked Visited)          # Start Again from Case-1, to explore          # this newly visited child          for i in range(0, len(Par.child)):              if Par.child[i].key not in Preorder:                  flag = 1                  Stack.append(Par.child[i])                  Preorder.append(Par.child[i].key)                  break;                  # b)If all Child nodes from left to right                  # of a Parent have been visited                  # then remove the parent from the stack.          if flag == 0:              Stack.pop()      print(Preorder)    # Execution Start From here  if \_\_name\_\_=='\_\_main\_\_':  # input nodes                  '''                  1                  / | \                  / | \              2 3 4              / \ / | \              5 6 7 8 9          / / | \          10 11 12 13                      '''    root = NewNode(1)  root.child.append(NewNode(2))  root.child.append(NewNode(3))  root.child.append(NewNode(4))  root.child[0].child.append(NewNode(5))  root.child[0].child[0].child.append(NewNode(10))  root.child[0].child.append(NewNode(6))  root.child[0].child[1].child.append(NewNode(11))  root.child[0].child[1].child.append(NewNode(12))  root.child[0].child[1].child.append(NewNode(13))  root.child[2].child.append(NewNode(7))  root.child[2].child.append(NewNode(8))  root.child[2].child.append(NewNode(9))    preorderTraversal(root) |

**Output:**

[1, 2, 5, 10, 6, 11, 12, 13, 3, 4, 7, 8, 9]