Data Structures and Algorithms for Interviews - 2

Vignesh Narayanan

GESSA

Graduate Engineering & Science Students Association

ASU

Maximum of all sub arrays of size k

- Given an array and an integer k, find the maximum for each and every contiguous subarray of size k.
- Example:
- Input :
  arr[] = {1, 2, 3, 1, 4, 5, 2, 3, 6}
  k = 3
  Output :
  3 3 4 5 5 5 6

```cpp
// A Dequeue (Double ended queue) based method for printing maixmum element of
// all subarrays of size k
void printKMax(int arr[], int n, int k)
{
    // Create a Double Ended Queue, Qi that will store indexes of array elements
    // The queue will store indexes of useful elements in every window and it will
    // maintain decreasing order of values from front to rear in Qi, i.e.,
    // arr[Qi.front[]] to arr[Qi.rear()] are sorted in decreasing order
    std::deque<int>  Qi(k);

    /* Process first k (or first window) elements of array */
    int i;
    for (i = 0; i < k; ++i)
    {
        // For very element, the previous smaller elements are useless so
        // remove them from Qi
        while ( (!Qi.empty()) && arr[i] >= arr[Qi.back()])
            Qi.pop_back();  // Remove from rear

        // Add new element at rear of queue
        Qi.push_back(i);
    }

    // Process rest of the elements, i.e., from arr[k] to arr[n-1]
    for ( ; i < n; ++i)
    {
        // The element at the front of the queue is the largest element of
        // previous window, so print it
        cout << arr[Qi.front()] << " ";

        // Remove the elements which are out of this window
        while ( (!Qi.empty()) && Qi.front() <= i - k)
            Qi.pop_front();  // Remove from front of queue

        // Remove all elements smaller than the currently
        // being added element (remove useless elements)
        while ( (!Qi.empty()) && arr[i] >= arr[Qi.back()])
            Qi.pop_back();

         // Add current element at the rear of Qi
        Qi.push_back(i);
    }

    // Print the maximum element of last window
    cout << arr[Qi.front()];
}
```

Linked Lists

- Arrays vs Linked Lists ?

Find middle node of a given linked list

Method 1:

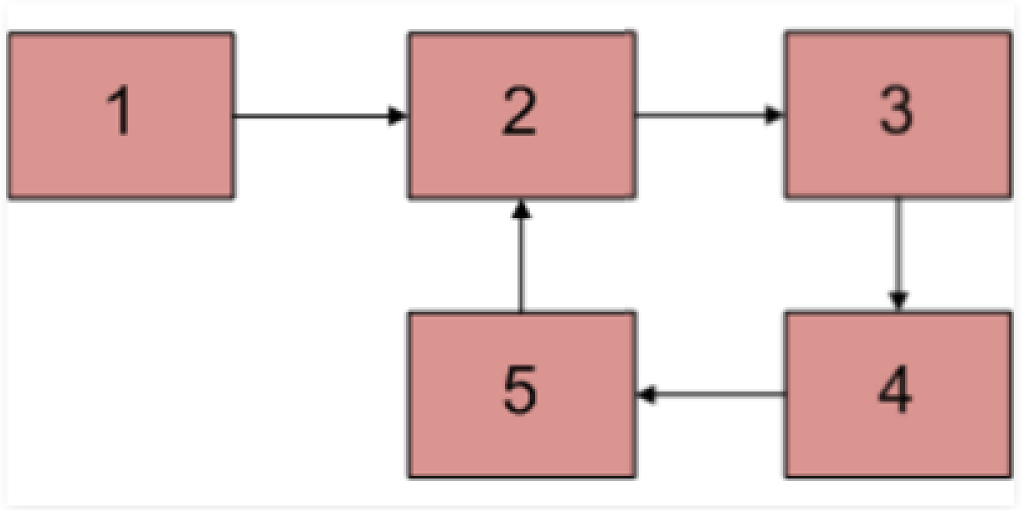
Traverse the whole linked list and count the no. of nodes. Now traverse the list again till count/2 and return the node at count/2.

Method 2:

Traverse linked list using two pointers. Move one pointer by one and other pointer by two. When the fast pointer reaches end slow pointer will reach middle of the linked list.

```c
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to get the middle of the linked list*/
void printMiddle(struct node *head)
{
    struct node *slow_ptr = head;
    struct node *fast_ptr = head;

    if (head!=NULL)
    {
        while (fast_ptr != NULL && fast_ptr->next != NULL)
        {
            fast_ptr = fast_ptr->next->next;
            slow_ptr = slow_ptr->next;
        }
        printf("The middle element is [%d]\n\n", slow_ptr->data);
    }
}
```

Detect loop in a linked list



Use Hashing:

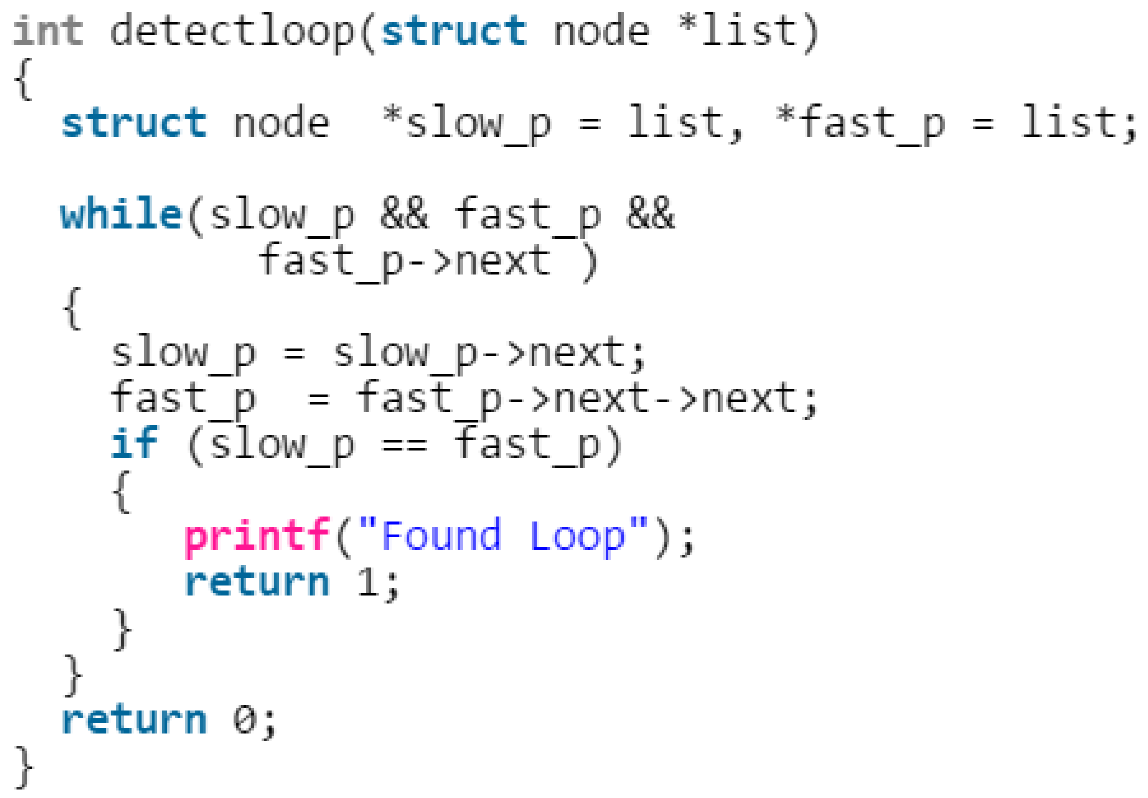
Traverse the list one by one and keep putting the node addresses in a Hash Table. At any point, if NULL is reached then return false and if next of current node points to any of the previously stored nodes in Hash then return true.

Mark Visited Nodes:

This solution requires modifications to basic linked list data structure. Have a visited flag with each node. Traverse the linked list and keep marking visited nodes. If you see a visited node again then there is a loop. This solution works in O(n) but requires additional information with each node.

A variation of this solution that doesn't require modification to basic data structure can be implemented using hash. Just store the addresses of visited nodes in a hash and if you see an address that already exists in hash then there is a loop.

Floyd’s Cycle-Finding Algorithm



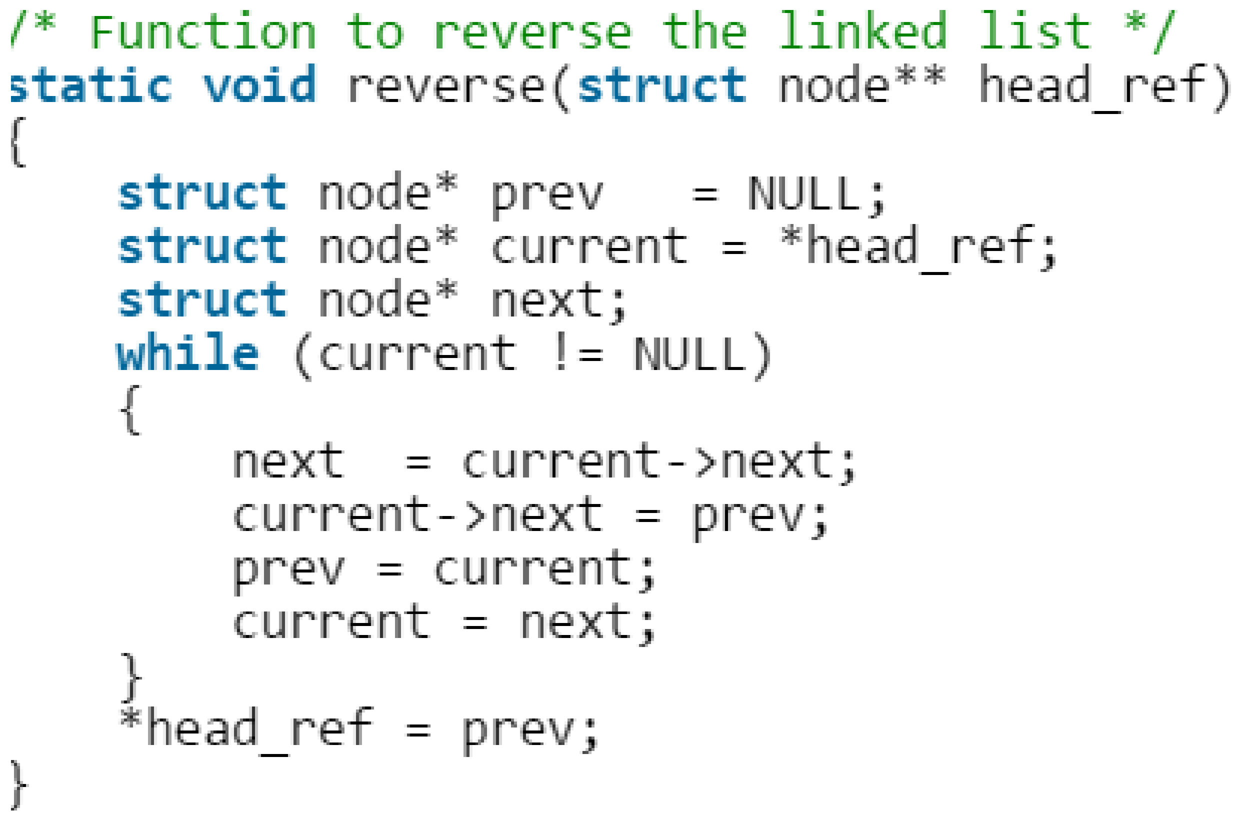
```c
int detectloop(struct node *list)
{
  struct node  *slow_p = list, *fast_p = list;

  while(slow_p && fast_p &&
          fast_p->next )
  {
    slow_p = slow_p->next;
    fast_p  = fast_p->next->next;
    if (slow_p == fast_p)
    {
      printf("Found Loop");
      return 1;
    }
  }
  return 0;
}
```

Remove the detected loop

```
void detectAndRemoveLoop(Node *head)
{
    Node *slow = head;
    Node *fast = head->next;

    // Search for loop using slow and fast pointers
    while (fast && fast->next)
    {
        if (slow == fast)
            break;
        slow = slow->next;
        fast = fast->next->next;
    }

    /* 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 */
    }
}
```

Reverse a linked list

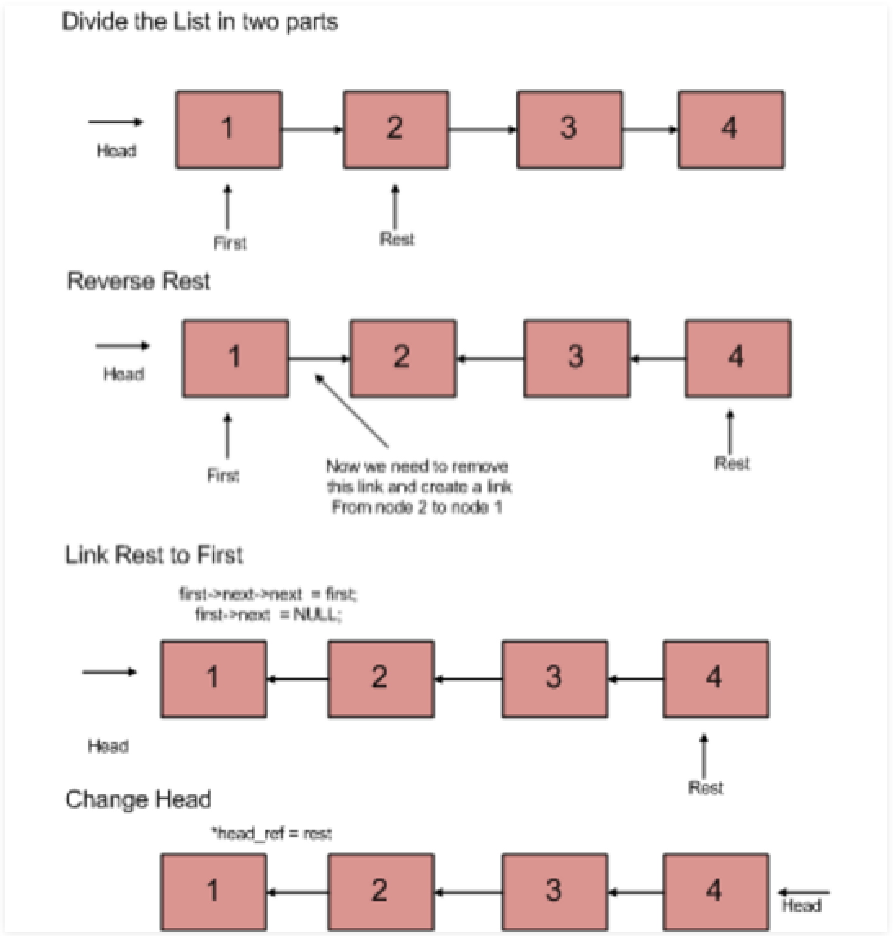
Iterative Method



```c
/* Function to reverse the linked list */
static void reverse(struct node** head_ref)
{
    struct node* prev   = NULL;
    struct node* current = *head_ref;
    struct node* next;
    while (current != NULL)
    {
        next  = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head_ref = prev;
}
```

Recursive Method

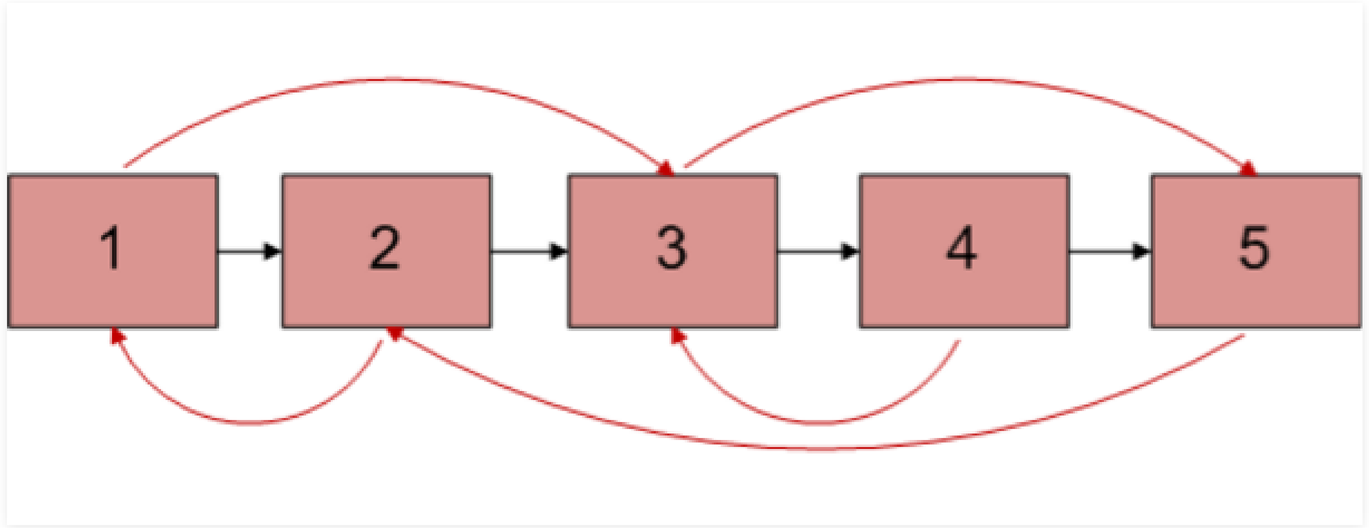
```
1) Divide the list in two parts - first node and rest of the linked list.
2) Call reverse for the rest of the linked list.
3) Link rest to first.
4) Fix head pointer
```



Clone a linked list with next and random pointer

You are given a Double Link List with one pointer of each node pointing to the next node just like in a single link list. The second pointer however CAN point to any node in the list and not just the previous node. Now write a program in **O(n) time** to duplicate this list. That is, write a program which will create a copy of this list.

Let us call the second pointer as arbit pointer as it can point to any arbitrary node in the linked list.



Arbitrary pointers are shown in red and next pointers in black

1) Create all nodes in copy linked list using next pointers.

3) Store the node and its next pointer mappings of original linked list.

3) Change next pointer of all nodes in original linked list to point to the corresponding node in copy linked list.

Following diagram shows status of both Linked Lists after above 3 steps. The red arrow shows arbit pointers and black arrow shows next pointers.

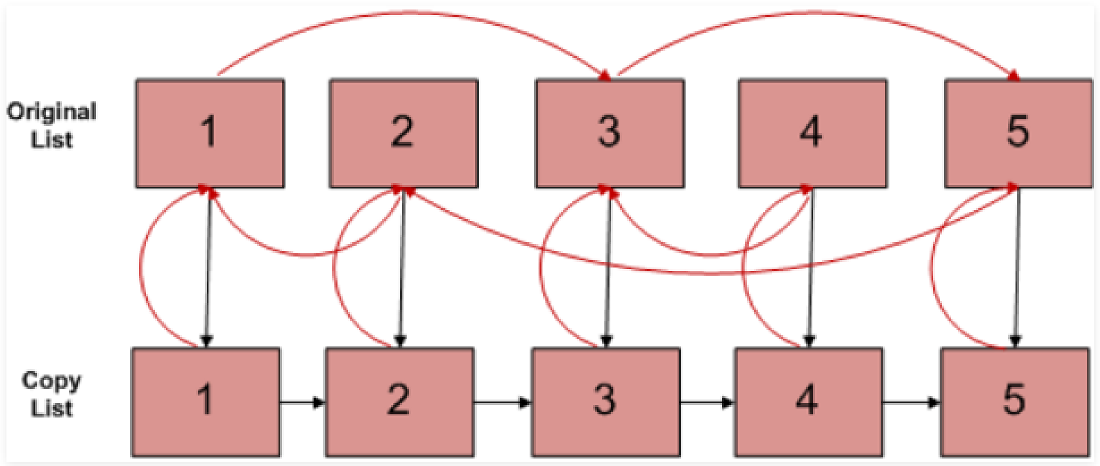


Figure 2

4) Change the arbit pointer of all nodes in copy linked list to point to corresponding node in original linked list.

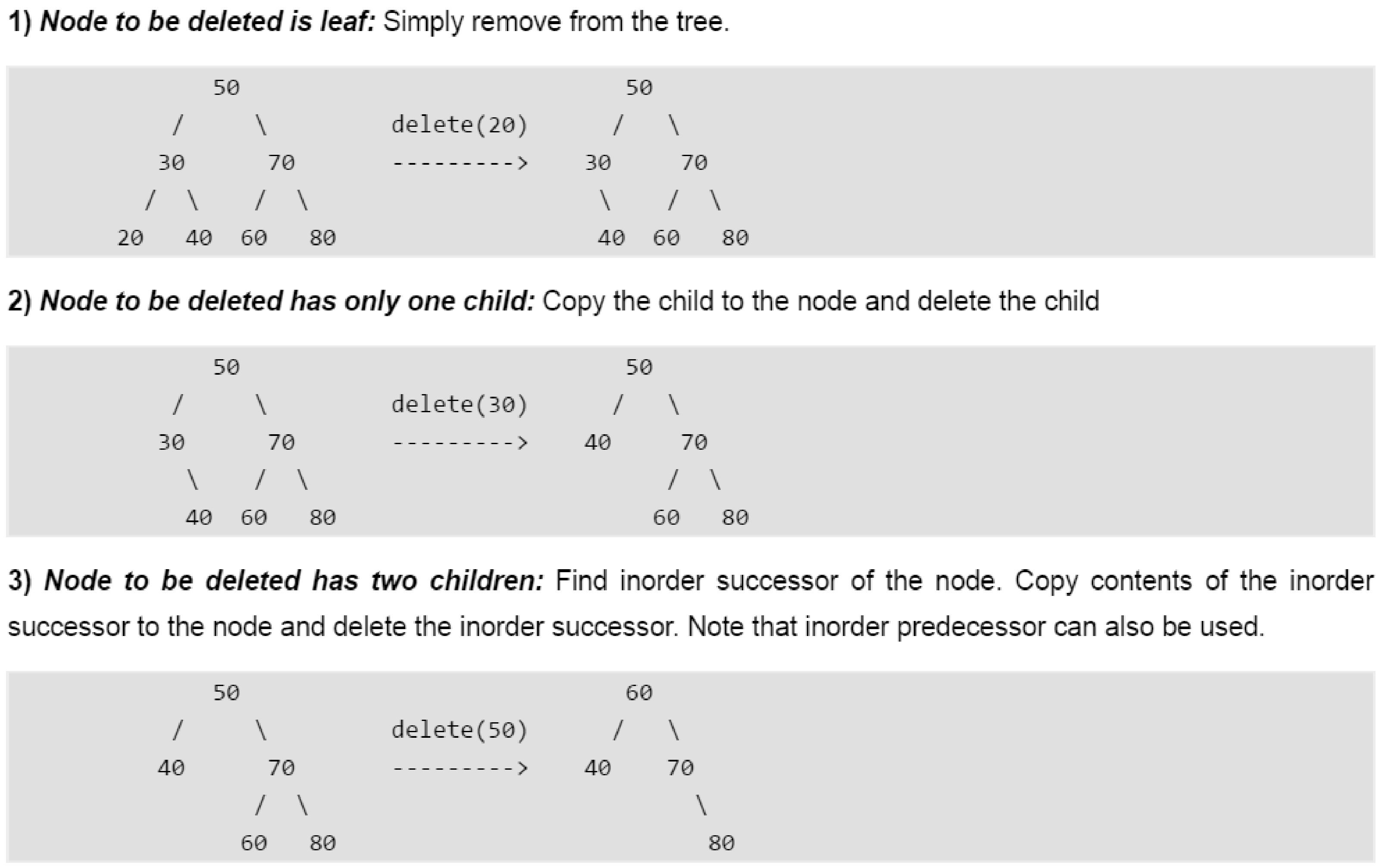
5) Now construct the arbit pointer in copy linked list as below and restore the next pointer of nodes in the original linked list.

```
copy_list_node->arbit =
              copy_list_node->arbit->arbit->next;
copy_list_node = copy_list_node->next;
```

6) Restore the next pointers in original linked list from the stored mappings(in step 2).

Binary Search Tree

- Insertion
- Search
- Deletion



1) ***Node to be deleted is leaf:*** Simply remove from the tree.

```
           50                            50
        /     \         delete(20)      /   \
      30       70       --------->    30     70
     /  \     /  \                     \    /  \
   20   40  60   80                     40  60   80
```

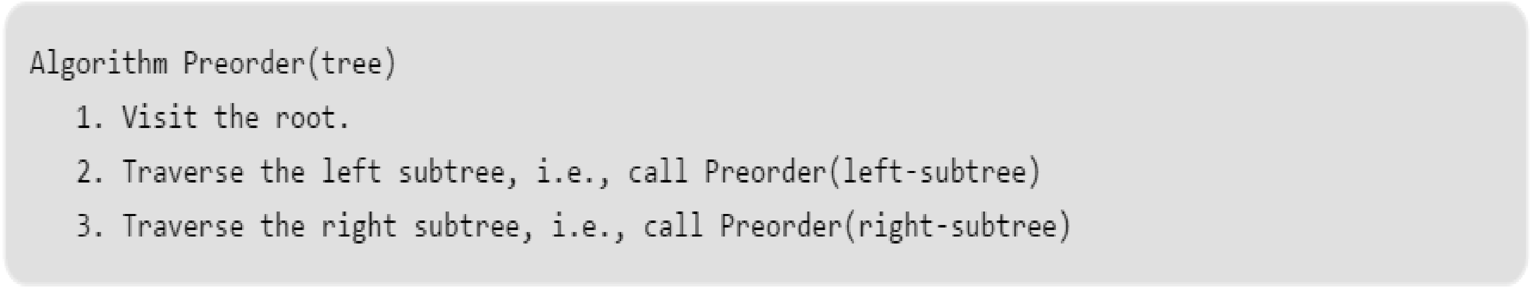
2) ***Node to be deleted has only one child:*** Copy the child to the node and delete the child

```
           50                            50
        /     \         delete(30)      /   \
      30       70       --------->    40     70
        \     /  \                          /  \
        40  60   80                        60   80
```

3) ***Node to be deleted has two children:*** Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.

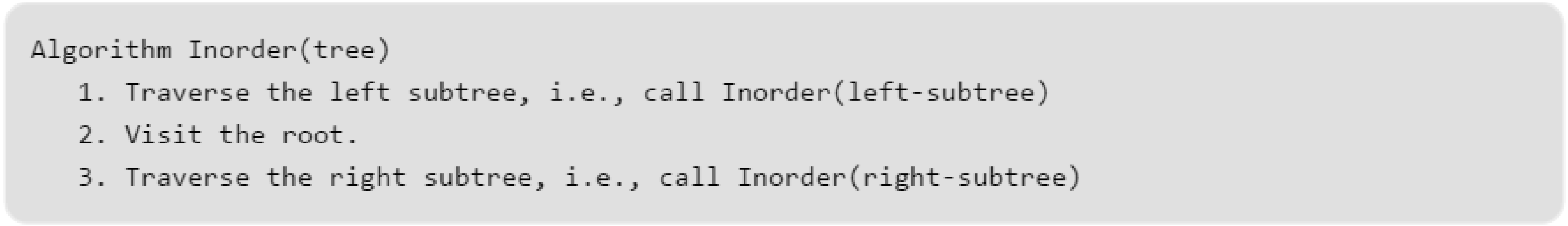
```
           50                            60
        /     \         delete(50)      /   \
      40       70       --------->    40    70
              /  \                            \
            60   80                            80
```

Tree Traversals

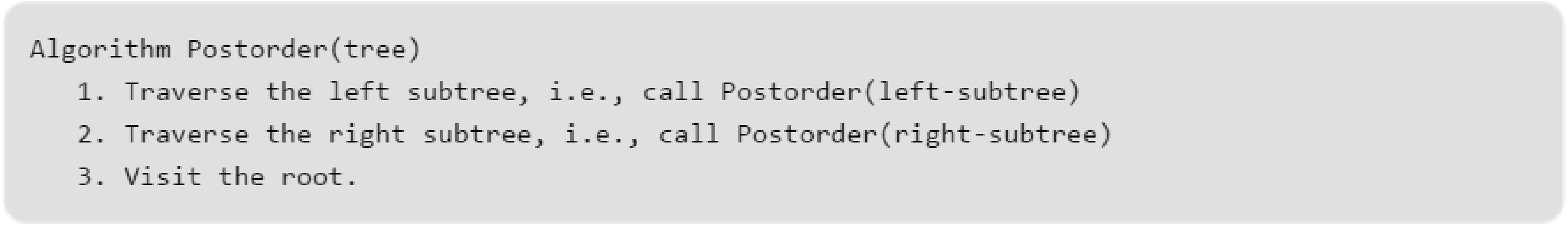


```
Algorithm Preorder(tree)
   1. Visit the root.
   2. Traverse the left subtree, i.e., call Preorder(left-subtree)
   3. Traverse the right subtree, i.e., call Preorder(right-subtree)
```

Inorder Traversal:



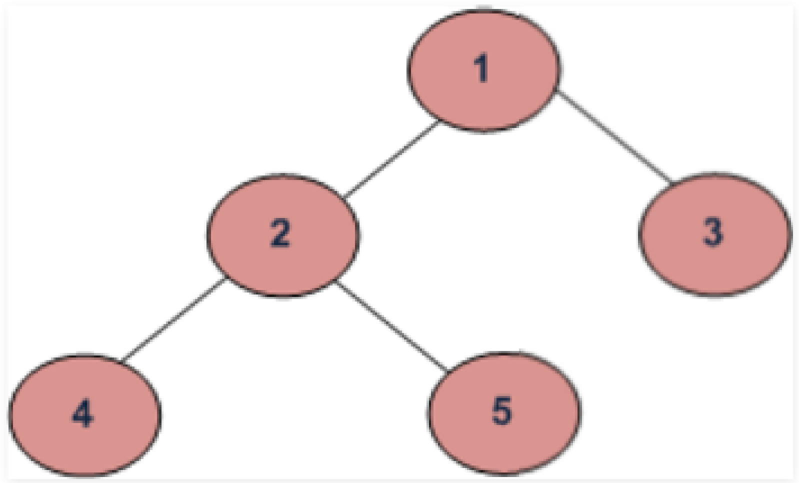
```
Algorithm Inorder(tree)
   1. Traverse the left subtree, i.e., call Inorder(left-subtree)
   2. Visit the root.
   3. Traverse the right subtree, i.e., call Inorder(right-subtree)
```



```
Algorithm Postorder(tree)
   1. Traverse the left subtree, i.e., call Postorder(left-subtree)
   2. Traverse the right subtree, i.e., call Postorder(right-subtree)
   3. Visit the root.
```

Level Order Traversal

Level order traversal of a tree is breadth first traversal for the tree.



Example Tree

Level order traversal of the above tree is 1 2 3 4 5

There are basically two functions in this method. One is to print all nodes at a given level (printGivenLevel), and other is to print level order traversal of the tree (printLevelorder). printLevelorder makes use of printGivenLevel to print nodes at all levels one by one starting from root.

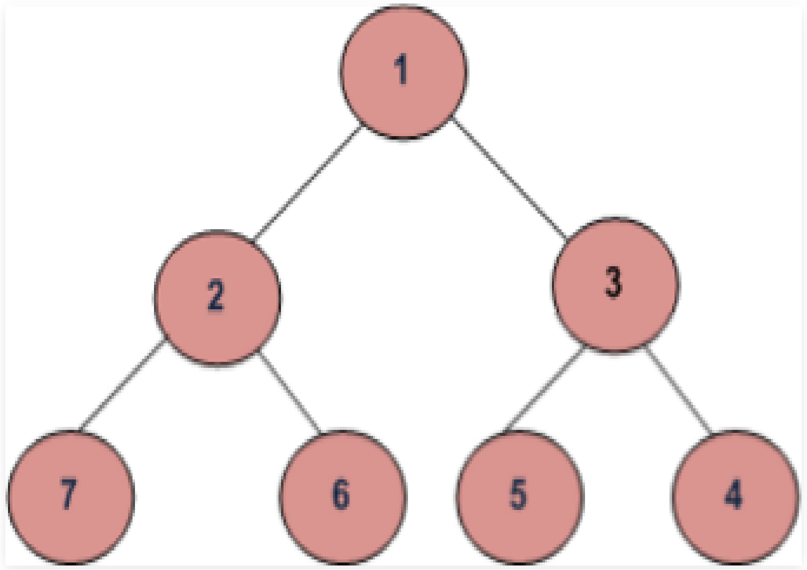
```
/*Function to print level order traversal of tree*/
printLevelorder(tree)
for d = 1 to height(tree)
   printGivenLevel(tree, d);

/*Function to print all nodes at a given level*/
printGivenLevel(tree, level)
if tree is NULL then return;
if level is 1, then
    print(tree->data);
else if level greater than 1, then
    printGivenLevel(tree->left, level-1);
    printGivenLevel(tree->right, level-1);
```

For each node, first the node is visited and then it's child nodes are put in a FIFO queue.

```
printLevelorder(tree)
1) Create an empty queue q
2) temp_node = root /*start from root*/
3) Loop while temp_node is not NULL
    a) print temp_node->data.
    b) Enqueue temp_node's children (first left then right children) to q
    c) Dequeue a node from q and assign it's value to temp_node
```

Level order traversal in spiral form

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



This problem can bee seen as an extension of the level order traversal post.

To print the nodes in spiral order, nodes at different levels should be printed in alternating order. An additional Boolean variable *ltr* is used to change printing order of levels. If *ltr* is 1 then printGivenLevel() prints nodes from left to right else from right to left. Value of *ltr* is flipped in each iteration to change the order.

Function to print level order traversal of tree

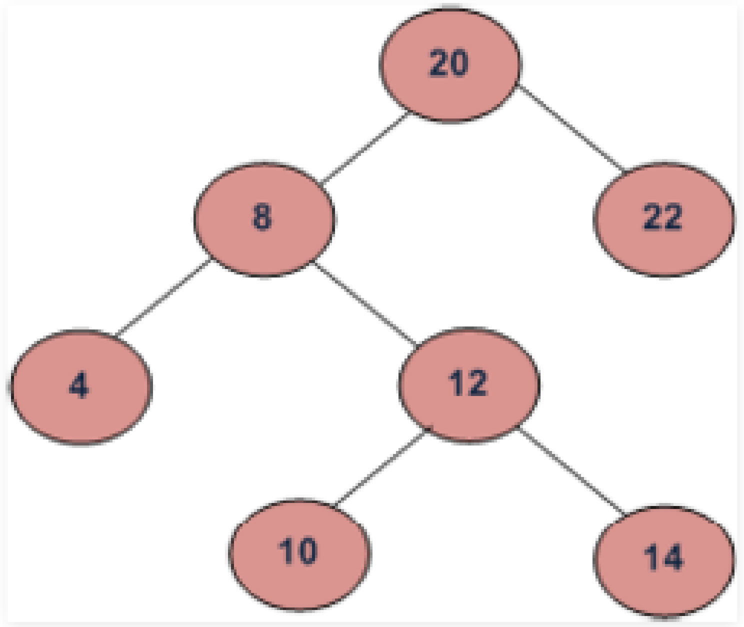
```
printSpiral(tree)
  bool ltr = 0;
  for d = 1 to height(tree)
     printGivenLevel(tree, d, ltr);
     ltr ~= ltr /*flip ltr*/
```

Function to print all nodes at a given level

```
printGivenLevel(tree, level, ltr)
if tree is NULL then return;
if level is 1, then
    print(tree->data);
else if level greater than 1, then
    if(ltr)
        printGivenLevel(tree->left, level-1, ltr);
        printGivenLevel(tree->right, level-1, ltr);
    else
        printGivenLevel(tree->right, level-1, ltr);
        printGivenLevel(tree->left, level-1, ltr);
```

Lowest Common Ancestor in a Binary Search Tree

Let T be a rooted tree. The lowest common ancestor between two nodes n1 and n2 is defined as the lowest node in T that has both n1 and n2 as descendants (where we allow a node to be a descendant of itself).



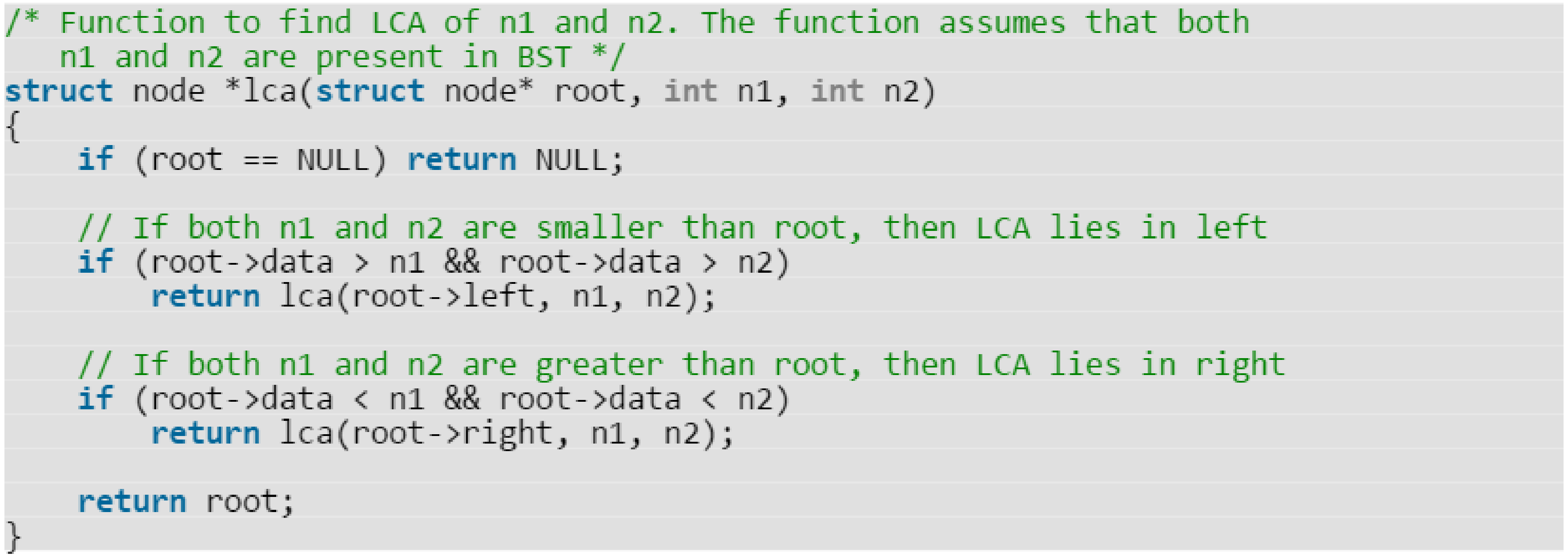
Consider the BST in diagram:

LCA of 10 and 14 is 12

LCA of 8 and 14 is 8.

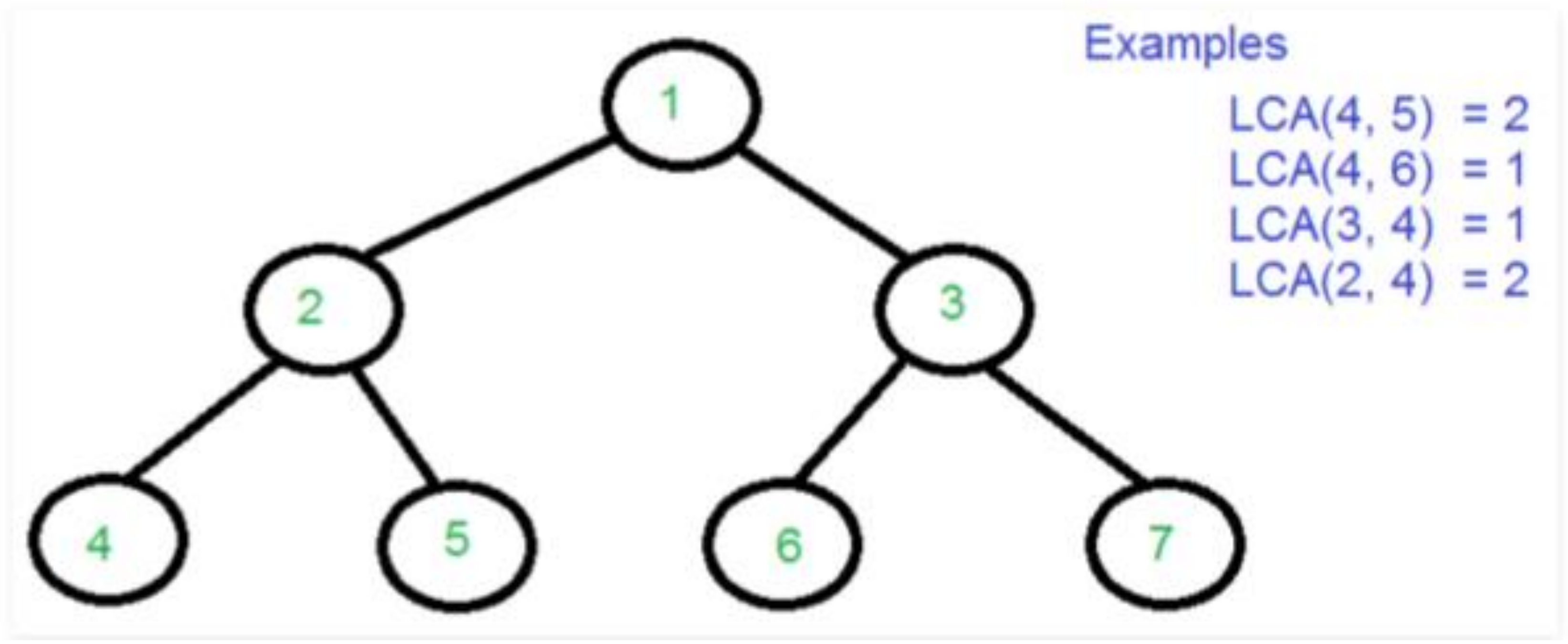
We can **recursively traverse** the BST from root.

The main idea of the solution is, while traversing from top to bottom, the first node n we encounter with value between n1 and n2, i.e., n1 < n < n2 or same as one of the n1 or n2, is LCA of n1 and n2 (assuming that n1 < n2). So just recursively traverse the BST in, if node's value is greater than both n1 and n2 then our LCA lies in left side of the node, if it's is smaller than both n1 and n2, then LCA lies on right side. Otherwise root is LCA (assuming that both n1 and n2 are present in BST)



```
/* Function to find LCA of n1 and n2. The function assumes that both
   n1 and n2 are present in BST */
struct node *lca(struct node* root, int n1, int n2)
{
    if (root == NULL) return NULL;

    // If both n1 and n2 are smaller than root, then LCA lies in left
    if (root->data > n1 && root->data > n2)
        return lca(root->left, n1, n2);

    // If both n1 and n2 are greater than root, then LCA lies in right
    if (root->data < n1 && root->data < n2)
        return lca(root->right, n1, n2);

    return root;
}
```

Lowest Common Ancestor in a Binary Tree



Method 1 (By Storing root to n1 and root to n2 paths):

Following is simple O(n) algorithm to find LCA of n1 and n2.

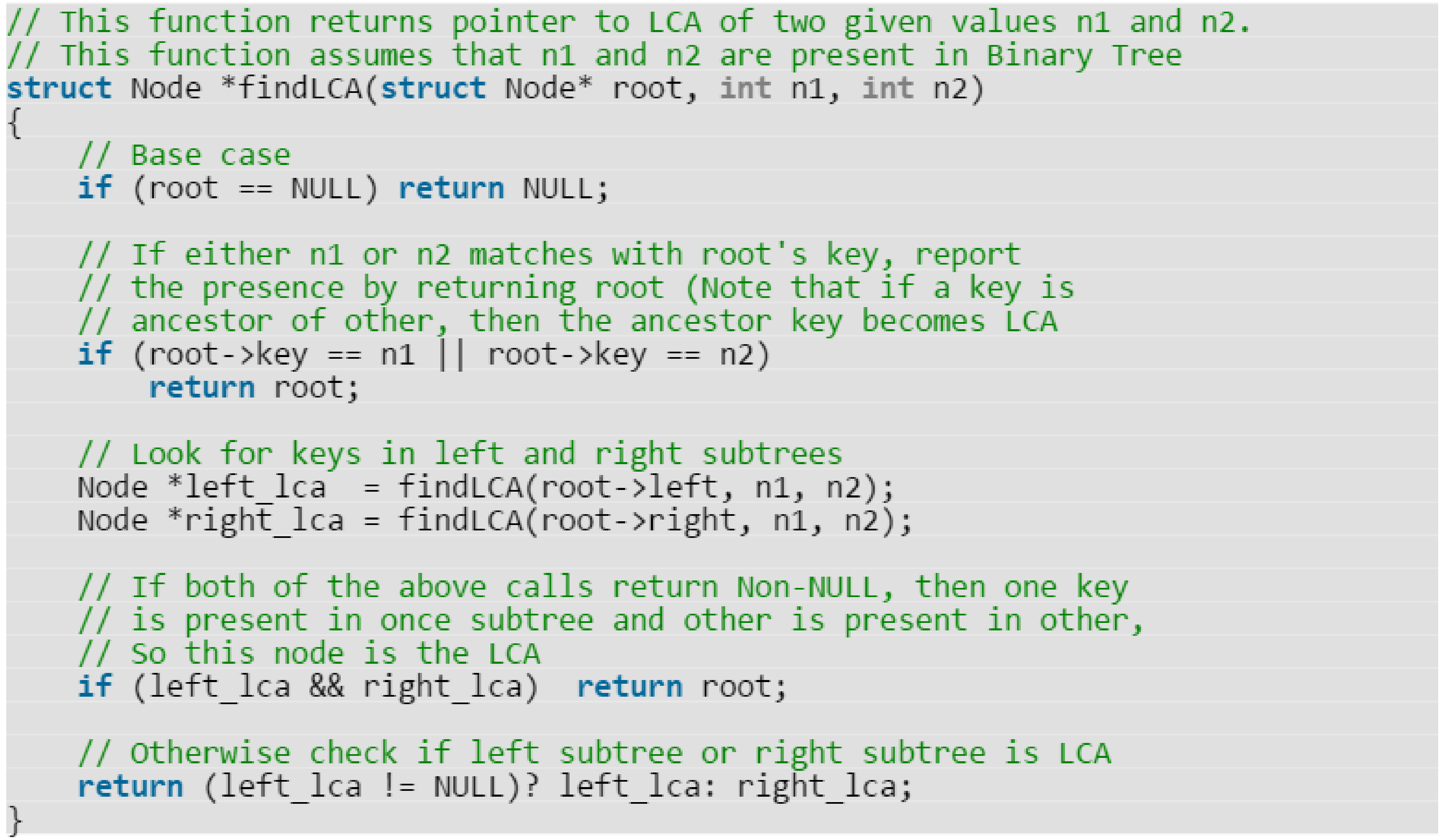
1) Find path from root to n1 and store it in a vector or array.

2) Find path from root to n2 and store it in another vector or array.

3) Traverse both paths till the values in arrays are same. Return the common element just before the mismatch.

```cpp
// Returns LCA if node n1, n2 are present in the given binary tree,
// otherwise return -1
int findLCA(Node *root, int n1, int n2)
{
  // to store paths to n1 and n2 from the root
  vector<int> path1, path2;

  // Find paths from root to n1 and root to n1. If either n1 or n2
  // is not present, return -1
  if ( !findPath(root, path1, n1) || !findPath(root, path2, n2))
     return -1;

  /* Compare the paths to get the first different value */
  int i;
  for (i = 0; i < path1.size() && i < path2.size() ; i++)
    if (path1[i] != path2[i])
      break;
  return path1[i-1];
}
```

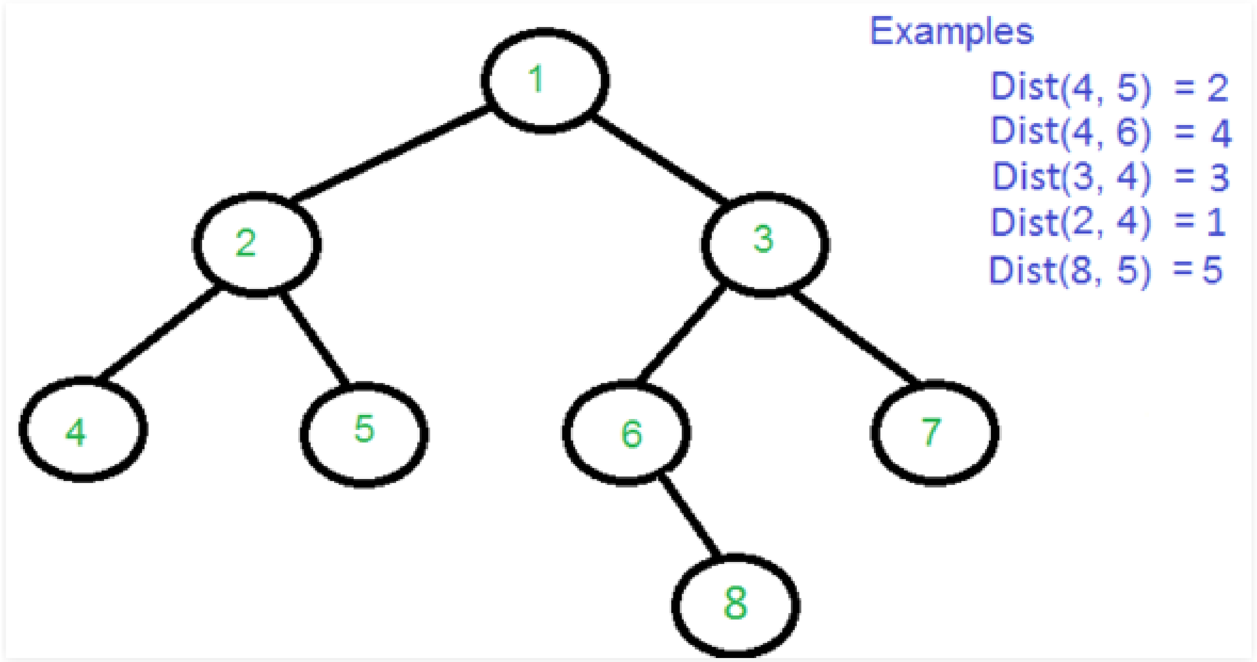
The idea is to traverse the tree starting from root. If any of the given keys (n1 and n2) matches with root, then root is LCA (assuming that both keys are present). If root doesn’t match with any of the keys, we recur for left and right subtree. The node which has one key present in its left subtree and the other key present in right subtree is the LCA. If both keys lie in left subtree, then left subtree has LCA also, otherwise LCA lies in right subtree.



```
// This function returns pointer to LCA of two given values n1 and n2.
// This function assumes that n1 and n2 are present in Binary Tree
struct Node *findLCA(struct Node* root, int n1, int n2)
{
    // Base case
    if (root == NULL) return NULL;

    // If either n1 or n2 matches with root's key, report
    // the presence by returning root (Note that if a key is
    // ancestor of other, then the ancestor key becomes LCA
    if (root->key == n1 || root->key == n2)
        return root;

    // Look for keys in left and right subtrees
    Node *left_lca  = findLCA(root->left, n1, n2);
    Node *right_lca = findLCA(root->right, n1, n2);

    // If both of the above calls return Non-NULL, then one key
    // is present in once subtree and other is present in other,
    // So 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;
}
```

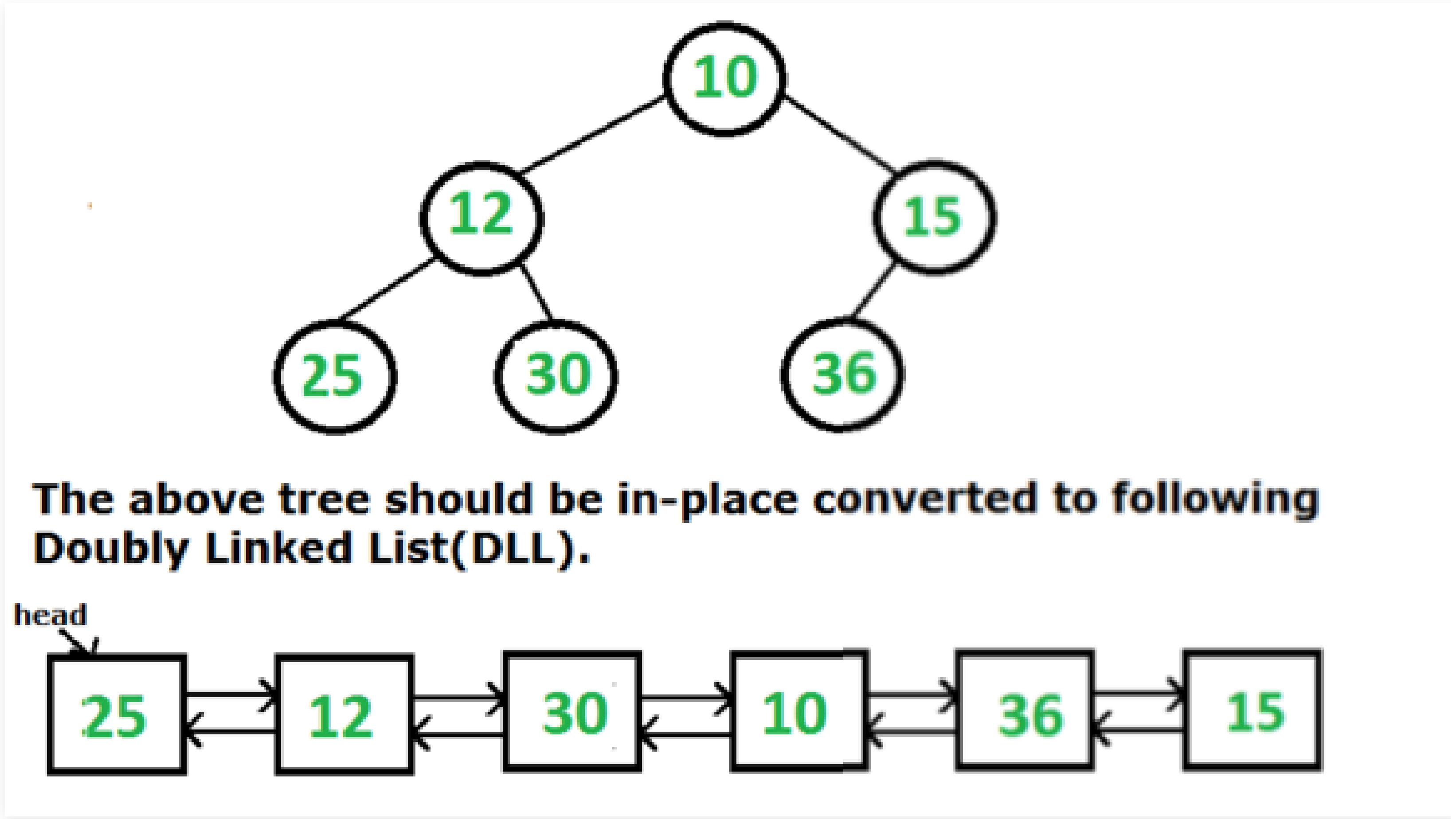
Find distance between two given keys of a Binary Tree

Distance between two nodes is the minimum number of edges to be traversed to reach one node from other.



```
Dist(n1, n2) = Dist(root, n1) + Dist(root, n2) - 2*Dist(root, lca)
'n1' and 'n2' are the two given keys
'root' is root of given Binary Tree.
'lca' is lowest common ancestor of n1 and n2
Dist(n1, n2) is the distance between n1 and n2.
```

Convert a given Binary Tree to Doubly Linked List



The above tree should be in-place converted to following Doubly Linked List(DLL).

1. If left subtree exists, process the left subtree

…..**1.a)** Recursively convert the left subtree to DLL.

…..**1.b)** Then find inorder predecessor of root in left subtree (inorder predecessor is rightmost node in left subtree).

…..**1.c)** Make inorder predecessor as previous of root and root as next of inorder predecessor.

2. If right subtree exists, process the right subtree (Below 3 steps are similar to left subtree).

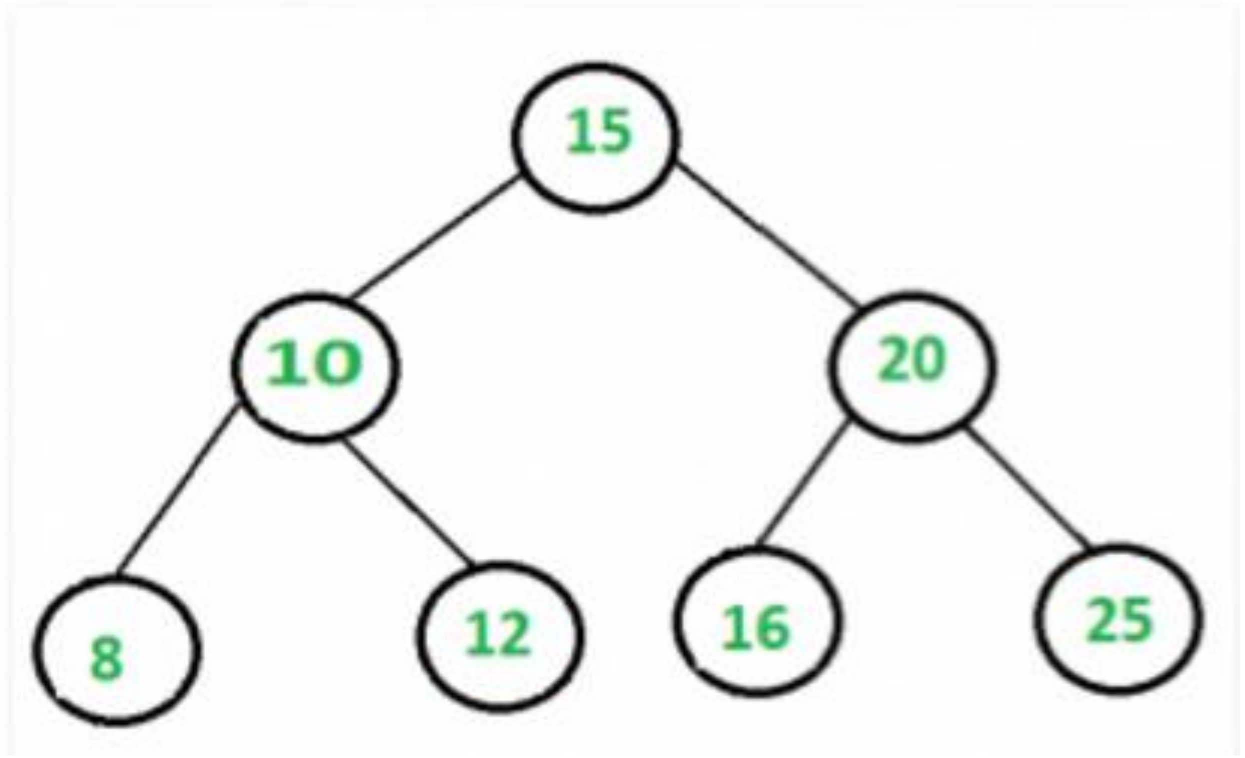
…..**2.a)** Recursively convert the right subtree to DLL.

…..**2.b)** Then find inorder successor of root in right subtree (inorder successor is leftmost node in right subtree).

…..**2.c)** Make inorder successor as next of root and root as previous of inorder successor.

3. Find the leftmost node and return it (the leftmost node is always head of converted DLL).

Find a pair with given sum in a Balanced BST



- The **Brute Force Solution** is to consider each pair in BST and check whether the sum equals to X. The time complexity of this solution will be O(n^2).
- A **Better Solution** is to create an auxiliary array and store Inorder traversal of BST in the array. The array will be sorted as Inorder traversal of BST always produces sorted data. Once we have the Inorder traversal, we can pair in O(n) time. This solution works in O(n) time, but requires O(n) auxiliary space.

Space Optimized Solution

The idea is:

- First in-place convert BST to Doubly Linked List (DLL)
- Then find pair in sorted DLL in O(n) time.
- This solution takes O(n) time and O(Logn) extra space.
- See to that it modifies the given BST.
- Again convert the DLL back to BST (if needed).