**12345780894 -- find where the list circles**

public boolean hasCycle(ListNode head) {

ListNode fast = head, slow = head;

while (fast != null && fast.next != null) {

fast = fast.next.next;

slow = slow.next;

if (fast == slow) return true;

}

return false;

}

**1444123255232555555890 -**

**Write a C program for a input linkedList 1->2->3->4->5->6... o/p should be 2->1->4->3->6->5...**

/\*iterative method:\*/

void swap\_twonodes(struct node\*\* headref)

{

struct node\* current=\*headref;

struct node\* save=NULL;

if(current==NULL) return; /\*empty list\*/

if(current->next!=NULL) /\*set head pointer to correct position\*/

\*headref=current->next;

while( (current!=NULL) && (current->next!=NULL) ) /\*iterate down,to swap the list by two nodes\*/

{

save=current->next->next;

current->next->next=current;

current->next=save;

current=save;

}

}

**Merge a linked list into another linked list at alternate positions**

Given two linked lists, insert nodes of second list into first list at alternate positions of first list.  
For example, if first list is 5->7->17->13->11 and second is 12->10->2->4->6, the first list should become 5->12->7->10->17->2->13->4->11->6 and second list should become empty.

void merge(struct node \*p, struct node \*\*q)

{

     struct node \*p\_curr = p, \*q\_curr = \*q;

     struct node \*p\_next, \*q\_next;

     // While therre are avialable positions in p

     while (p\_curr != NULL && q\_curr != NULL)

     {

         // Save next pointers

         p\_next = p\_curr->next;

         q\_next = q\_curr->next;

         // Make q\_curr as next of p\_curr

         q\_curr->next = p\_next;  // Change next pointer of q\_curr

         p\_curr->next = q\_curr;  // Change next pointer of p\_curr

         // Update current pointers for next iteration

         p\_curr = p\_next;

         q\_curr = q\_next;

    }

    \*q = q\_curr; // Update head pointer of second list

}

**Delete N nodes after M nodes of a linked list**

Input:

M = 2, N = 2

Linked List: 1->2->3->4->5->6->7->8

Output:

Linked List: 1->2->5->6

void skipMdeleteN(struct node  \*head, int M, int N)

{

    struct node \*curr = head, \*t;

    int count;

    // The main loop that traverses through the whole list

    while (curr)

    {

        // Skip M nodes

        for (count = 1; count<M && curr!= NULL; count++)

            curr = curr->next;

        // If we reached end of list, then return

        if (curr == NULL)

            return;

        // Start from next node and delete N nodes

        t = curr->next;

        for (count = 1; count<=N && t!= NULL; count++)

        {

            struct node \*temp = t;

            t = t->next;

            free(temp);

        }

        curr->next = t; // Link the previous list with remaining nodes

        // Set current pointer for next iteration

        curr = t;

    }

}

**Intersection of 2 Loops**

**Method 1(Simply use two loops)**  
Use 2 nested for loops. Outer loop will be for each node of the 1st list and inner loop will be for 2nd list. In the inner loop, check if any of nodes of 2nd list is same as the current node of first linked list. Time complexity of this method will be O(mn) where m and n are the number of nodes in two lists.

**Method 2 (Mark Visited Nodes)**  
This solution requires modifications to basic linked list data structure. Have a visited flag with each node. Traverse the first linked list and keep marking visited nodes. Now traverse second linked list, If you see a visited node again then there is an intersection point, return the intersecting node. This solution works in O(m+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. Traverse the first linked list and store the addresses of visited nodes in a hash. Now traverse the second linked list and if you see an address that already exists in hash then return the intersecting node.

**Method 3(Using difference of node counts)**  
1) Get count of the nodes in first list, let count be c1.  
2) Get count of the nodes in second list, let count be c2.  
3) Get the difference of counts d = abs(c1 – c2)  
4) Now traverse the bigger list from the first node till d nodes so that from here onwards both the lists have equal no of nodes.  
5) Then we can traverse both the lists in parallel till we come across a common node. (Note that getting a common node is done by comparing the address of the nodes)

|  |
| --- |
| #include<stdio.h>  #include<stdlib.h>    /\* Link list node \*/  struct node  {    int data;    struct node\* next;  };    /\* Function to get the counts of node in a linked list \*/  int getCount(struct node\* head);    /\* function to get the intersection point of two linked     lists head1 and head2 where head1 has d more nodes than     head2 \*/  int \_getIntesectionNode(int d, struct node\* head1, struct node\* head2);    /\* function to get the intersection point of two linked     lists head1 and head2 \*/  int getIntesectionNode(struct node\* head1, struct node\* head2)  {    int c1 = getCount(head1);    int c2 = getCount(head2);    int d;      if(c1 > c2)    {      d = c1 - c2;      return \_getIntesectionNode(d, head1, head2);    }    else    {      d = c2 - c1;      return \_getIntesectionNode(d, head2, head1);    }  }    /\* function to get the intersection point of two linked     lists head1 and head2 where head1 has d more nodes than     head2 \*/  int \_getIntesectionNode(int d, struct node\* head1, struct node\* head2)  {    int i;    struct node\* current1 = head1;    struct node\* current2 = head2;      for(i = 0; i < d; i++)    {      if(current1 == NULL)      {  return -1; }      current1 = current1->next;    }      while(current1 !=  NULL && current2 != NULL)    {      if(current1 == current2)        return current1->data;      current1= current1->next;      current2= current2->next;    }      return -1;  }    /\* Takes head pointer of the linked list and     returns the count of nodes in the list \*/  int getCount(struct node\* head)  {    struct node\* current = head;    int count = 0;      while (current != NULL)    {      count++;      current = current->next;    }      return count;  } |

**Trees**

**Min Depth of Tree**

int min\_depth(struct Node\* root, int depth)

{

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

return depth;

int x = (root->left != NULL) ? min\_depth(root->left, depth+1) : depth;

int y = (root->right != NULL) ? min\_depth(root->right, depth+1) : depth;

return (x < y) ? x : y;

}

**Balanced Tree or Not**

public static int maxDepth(TreeNode root) {

2 if (root == null) {

3 return 0;

4 }

5 return 1 + Math.max(maxDepth(root.left), maxDepth(root.right));

6 }

7

8 public static int minDepth(TreeNode root) {

9 if (root == null) {

10 return 0;

11 }

12 return 1 + Math.min(minDepth(root.left), minDepth(root.right));

13 }

14

15 public static boolean isBalanced(TreeNode root){

16 return (maxDepth(root) - minDepth(root) <= 1);

17 }

**Sub tree**

The treeMatch procedure visits each node in the small tree at most once and is called no more than once per node of the large tree.Worst case runtime is at most O(n \* m), where n and m are the sizes of trees T1 and T2, respectively.If k is the number of occurrences of T2’s root in T1, the worst case runtime can be characterized as O(n + k \* m).

bool areIdentical(struct node \* root1, struct node \*root2)

{

    /\* base cases \*/

    if(root1 == NULL && root2 == NULL)

        return true;

    if(root1 == NULL || root2 == NULL)

        return false;

    /\* Check if the data of both roots is same and data of left and right

       subtrees are also same \*/

    return (root1->data == root2->data   &&

            areIdentical(root1->left, root2->left) &&

            areIdentical(root1->right, root2->right) );

}

/\* This function returns true if S is a subtree of T, otherwise false \*/

bool isSubtree(struct node \*T, struct node \*S)

{

    /\* base cases \*/

    if (S == NULL)

        return true;

    if (T == NULL)

        return false;

    /\* Check the tree with root as current node \*/

    if (areIdentical(T, S))

        return true;

    /\* If the tree with root as current node doesn't match then

       try left and right subtrees one by one \*/

    return isSubtree(T->left, S) ||

           isSubtree(T->right, S);

}

**In Order Successor**

**1)** If right subtree of nodeis not NULL, then succlies in right subtree. Do following.  
Go to right subtree and return the node with minimum key value in right subtree.  
**2)**If right sbtree of nodeis NULL, then succis one of the ancestors. Do following.  
Travel up using the parent pointer until you see a node which is left child of it’s parent. The parent of such a node is the succ.

struct node \* inOrderSuccessor(struct node \*root, struct node \*n)

{

  // step 1 of the above algorithm

  if( n->right != NULL )

    return minValue(n->right);

  // step 2 of the above algorithm

  struct node \*p = n->parent;

  while(p != NULL && n == p->right)

  {

     n = p;

     p = p->parent;

  }

  return p;

}

**In Order Predecessor**

**Node \* find\_maximum(Node \*root){**

**if(!root)**  
**return NULL;**  
 **while(root->right){**  
**root = root->right;**  
 **}**  
**return root;**  
**}**

**Node \*inorder\_preced(Node \*root, int K){**  
 **Node \* successor = NULL;**  
**Node \*current  = root;**  
**if(!root)**  
**return NULL;**  
 **while(current && current->value != K){**  
**if(current->value >K){**  
**current= current->left;**  
**}**  
**else{**  
**successor = current;**  
**current = current->right;**  
**}**  
**}**  
**if(current && current->left){**  
**successor = find\_maximum(current->left);**  
**}**  
 **return successor;**  
 **}**

**Binary tree to link list**

node\* bintree2listUtil(node\* root)

{

    // Base case

    if (root == NULL)

        return root;

    // Convert the left subtree and link to root

    if (root->left != NULL)

    {

        // Convert the left subtree

        node\* left = bintree2listUtil(root->left);

        // Find inorder predecessor. After this loop, left

        // will point to the inorder predecessor

        for (; left->right!=NULL; left=left->right);

        // Make root as next of the predecessor

        left->right = root;

        // Make predecssor as previous of root

        root->left = left;

    }

    // Convert the right subtree and link to root

    if (root->right!=NULL)

    {

        // Convert the right subtree

        node\* right = bintree2listUtil(root->right);

        // Find inorder successor. After this loop, right

        // will point to the inorder successor

        for (; right->left!=NULL; right = right->left);

        // Make root as previous of successor

        right->left = root;

        // Make successor as next of root

        root->right = right;

    }

    return root;

}

// The main function that first calls bintree2listUtil(), then follows step 3

//  of the above algorithm

node\* bintree2list(node \*root)

{

    // Base case

    if (root == NULL)

        return root;

    // Convert to DLL using bintree2listUtil()

    root = bintree2listUtil(root);

    // bintree2listUtil() returns root node of the converted

    // DLL.  We need pointer to the leftmost node which is

    // head of the constructed DLL, so move to the leftmost node

    while (root->left != NULL)

        root = root->left;

    return (root);

}

**Max Sum Path**

// A utility function that prints all nodes on the path from root to target\_leaf

bool printPath (struct node \*root, struct node \*target\_leaf)

{

    // base case

    if (root == NULL)

        return false;

    // return true if this node is the target\_leaf or target leaf is present in

    // one of its descendants

    if (root == target\_leaf || printPath(root->left, target\_leaf) ||

            printPath(root->right, target\_leaf) )

    {

        printf("%d ", root->data);

        return true;

    }

    return false;

}

// This function Sets the target\_leaf\_ref to refer the leaf node of the maximum

// path sum.  Also, returns the max\_sum using max\_sum\_ref

void getTargetLeaf (struct node \*node, int \*max\_sum\_ref, int curr\_sum,

                   struct node \*\*target\_leaf\_ref)

{

    if (node == NULL)

        return;

    // Update current sum to hold sum of nodes on path from root to this node

    curr\_sum = curr\_sum + node->data;

    // If this is a leaf node and path to this node has maximum sum so far,

    // then make this node target\_leaf

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

    {

        if (curr\_sum > \*max\_sum\_ref)

        {

            \*max\_sum\_ref = curr\_sum;

            \*target\_leaf\_ref = node;

        }

    }

    // If this is not a leaf node, then recur down to find the target\_leaf

    getTargetLeaf (node->left, max\_sum\_ref, curr\_sum, target\_leaf\_ref);

    getTargetLeaf (node->right, max\_sum\_ref, curr\_sum, target\_leaf\_ref);

}

// Returns the maximum sum and prints the nodes on max sum path

int maxSumPath (struct node \*node)

{

    // base case

    if (node == NULL)

        return 0;

    struct node \*target\_leaf;

    int max\_sum = INT\_MIN;

    // find the target leaf and maximum sum

    getTargetLeaf (node, &max\_sum, 0, &target\_leaf);

    // print the path from root to the target leaf

    printPath (node, target\_leaf);

    return max\_sum;  // return maximum sum

}

**Strings**

**Anagram**

**Method 2 (Count characters)**  
This method assumes that the set of possible characters in both strings is small. In the following implementation, it is assumed that the characters are stored using 8 bit and there can be 256 possible characters.  
1) Create count arrays of size 256 for both strings. Initialize all values in count arrays as 0.  
2) Iterate through every character of both strings and increment the count of character in the corresponding count arrays.  
3) Compare count arrays. If both count arrays are same, then return true.

|  |
| --- |
| # include <stdio.h>  # define NO\_OF\_CHARS 256    /\* function to check whether two strings are anagram of each other \*/  bool areAnagram(char \*str1, char \*str2)  {      // Create two count arrays and initialize all values as 0      int count1[NO\_OF\_CHARS] = {0};      int count2[NO\_OF\_CHARS] = {0};      int i;        // For each character in input strings, increment count in      // the corresponding count array      for (i = 0; str1[i] && str2[i];  i++)      {          count1[str1[i]]++;          count2[str2[i]]++;      }        // If both strings are of different length. Removing this condition      // will make the program fail for strings like "aaca" and "aca"      if (str1[i] || str2[i])        return false;        // Compare count arrays      for (i = 0; i < NO\_OF\_CHARS; i++)          if (count1[i] != count2[i])              return false;        return true; |

**Reverse a string**

void reverse(char \*str) {

2 char \* end = str;

3 char tmp;

4 if (str) {

5 while (\*end) {

6 ++end;

7 }

8 --end;

9 while (str < end) {

10 tmp = \*str;

11 \*str++ = \*end;

12 \*end-- = tmp;

13 }

14 }

15 }

**Reverse a sentence**

void reverse(char \*src, char \*end)

{

while(src < end) {

char temp = \*src;

\*src++ = \*end;

\*end-- = temp;

}

}

void reverse\_sentence(char \*src, size\_t len)

{

if(len < 3) {

return;

}

reverse(src, src + len - 2);

char \*end = src + len - 1;

while(src != end) {

char \*start = src;

for(; start < end && \*start != ' '; ++start)

;

reverse(src, start - 1);

for(;start < end && \*start == ' '; ++start)

;

src = start;

}

}

**Reverse words in a given string**

Algorithm:

1) Reverse the individual words, we get the below string.

"i ekil siht margorp yrev hcum"

2) Reverse the whole string from start to end and you get the desired output.

"much very program this like i"

|  |
| --- |
| #include<stdio.h>    /\* function prototype for utility function to    reverse a string from begin to end  \*/  void reverse(char \*begin, char \*end);    /\*Function to reverse words\*/  void reverseWords(char \*s)  {    char \*word\_begin = s;    char \*temp = s; /\* temp is for word boundry \*/      /\*STEP 1 of the above algorithm \*/    while( \*temp )    {      temp++;      if (\*temp == '\0')      {        reverse(word\_begin, temp-1);      }      else if(\*temp == ' ')      {        reverse(word\_begin, temp-1);        word\_begin = temp+1;      }    } /\* End of while \*/       /\*STEP 2 of the above algorithm \*/    reverse(s, temp-1);  }    /\* UTILITY FUNCTIONS \*/  /\*Function to reverse any sequence starting with pointer    begin and ending with pointer end  \*/  void reverse(char \*begin, char \*end)  {    char temp;    while (begin < end)    {      temp = \*begin;      \*begin++ = \*end;      \*end-- = temp;    }  } |

**Wrong Push**

**WrongPush()**

Unfortunately Push() written in C suffers from a basic problem: what should be the

parameters to Push()? This is, unfortunately, a sticky area in C. There's a nice, obvious

way to write Push() which looks right but is wrong. Seeing exactly how it doesn't work

will provide an excuse for more practice with memory drawings, motivate the correct

solution, and just generally make you a better programmer....

void WrongPush(struct node\* head, int data) {

struct node\* newNode = malloc(sizeof(struct node));

newNode->data = data;

newNode->next = head;

head = newNode; // NO this line does not work!

}

void WrongPushTest() {

List head = BuildTwoThree();

WrongPush(head, 1); // try to push a 1 on front -- doesn't work

}

WrongPush() is very close to being correct. It takes the correct 3-Step Link In and puts it

an almost correct context. The problem is all in the very last line where the 3-Step Link

In dictates that we change the head pointer to refer to the new node. What does the line

head = newNode; do in WrongPush()? It sets a head pointer, but not the right one. It

sets the variable named head local to WrongPush(). It does not in any way change the

variable named head we really cared about which is back in the caller WrontPushTest().