**############################################################**

**1- Occurence of an integer in a Linked List**

Given a singly linked list and a key, count the number of occurrences of given key in the linked list.

**Example 1:**

**Input:**

N = 7

Link List = 1->2->1->2->1->3->1

search\_for = 1

**Output:** 4

**Explanation:**1 appears 4 times.

**Example 2:**

**Input:**

N = 5

Link List = 1->2->1->2->1

search\_for = 3

**Output:** 0

**Explanation:**3 appears 0 times.

**Your Task:**

You dont need to read input or print anything. Complete the function **count**() which takes the head of the link list and search\_for i.e- the key as input parameters and returns the count of occurrences of the given key.

**Expected Time Complexity:**O(N)  
**Expected Auxiliary Space:**O(1)

**Constraints:**  
0 ≤ N ≤ 10^4

int count(struct node\* head, int k)

{ int count = 0;

node \*temp = head;

while(temp != NULL)

{

if(temp->data == k)

count++;

temp = temp->next;

}

return count;

}

**############################################################**

**2- Reverse a linked list**

Given a linked list of **N**nodes. The task is to reverse this list.

**Example 1:**

**Input:**

LinkedList: 1->2->3->4->5->6

**Output:** 6 5 4 3 2 1

**Explanation:** After reversing the list,

elements are 6->5->4->3->2->1.

**Example 2:**

**Input:**

LinkedList: 2->7->8->9->10

**Output:** 10 9 8 7 2

**Explanation:** After reversing the list,

elements are 10->9->8->7->2.

**Your Task:**  
The task is to complete the function **reverseList**() with head reference as the only argument and should return new head after reversing the list.

**Expected Time Complexity:**O(N).  
**Expected Auxiliary Space:**O(1).

**Constraints:**  
1 <= N <= 104

struct Node\* reverseList(struct Node \*head)

{

Node \*a = NULL, \*b = head, \*c = NULL;

while(b != NULL)

{ c = b->next;

b->next = a;

a = b;

b = c;

}

return a;

}

**############################################################**

**3- Pairwise swap elements of a linked list**

Given a singly linked list of size **N**. The task is to swap elements in the linked list pairwise.  
For example, if the input list is 1 2 3 4, the resulting list after swaps will be 2 1 4 3.  
**Note**: You need to swap the nodes, not only the data. If only data is swapped then driver will print -1.

**Example 1:**

**Input:**

LinkedList: 1->2->2->4->5->6->7->8

**Output:** 2 1 4 2 6 5 8 7

**Explanation:** After swapping each pair

considering (1,2), (2, 4), (5, 6).. so

on as pairs, we get 2, 1, 4, 2, 6, 5,

8, 7 as a new linked list.

**Example 1:**

**Input:**

LinkedList: 1->3->4->7->9->10->1

**Output: 3** 1 7 4 10 9 1

**Explanation:** After swapping each pair

considering (1,3), (4, 7), (9, 10).. so

on as pairs, we get 3, 1, 7, 4, 10, 9,

1 as a new linked list.

**Your Task:**  
The task is to complete the function **pairWiseSwap**() which takes the head node as the only argument and returns the head of modified linked list.

**Expected Time Complexity:**O(N).  
**Expected Auxiliary Space:**O(1).

**Constraints:**  
1 ≤ N ≤ 103

Node\* pairWiseSwap(struct Node\* head)

{

if(head->next == NULL)

return head;

Node \*a = head, \*b = head->next, \*x = NULL, \*root = head->next;

while(true)

{

Node \*t = b->next;

a->next = t;

b->next = a;

if(x != NULL)

x->next = b;

if(t == NULL || t->next == NULL)

break;

x = a;

a = t;

b = t->next;

}

return root;

}

**############################################################**

**4- Add two numbers represented by linked lists**

Given two numbers represented by two linked lists of size **N** and **M**. The task is to return a sum list. The sum list is a linked list representation of the addition of two input numbers from the last.

**Example 1:**

**Input:**

N = 2

valueN[] = {4,5}

M = 3

valueM[] = {3,4,5}

**Output:** 3 9 0

**Explanation:** For the given two linked

list (4 5) and (3 4 5), after adding

the two linked list resultant linked

list will be (3 9 0).

**Example 2:**

**Input:**

N = 2

valueN[] = {6,3}

M = 1

valueM[] = {7}

**Output:** 7 0

**Explanation:** For the given two linked

list (6 3) and (7), after adding the

two linked list resultant linked list

will be (7 0).

**Your Task:**  
The task is to complete the function **addTwoLists**() which has node reference of both the linked lists and returns the head of the new list.

**Expected Time Complexity:**O(N+M)  
**Expected Auxiliary Space:**O(Max(N,M))

**Constraints:**  
1 <= N, M <= 5000

struct Node\* reverselist(struct Node \*head)

{

Node \*a = NULL, \*b = head, \*c = NULL;

while(b != NULL)

{ c = b->next;

b->next = a;

a = b;

b = c;

}

return a;

}

struct Node\* addTwoLists(struct Node\* first, struct Node\* second)

{

int c = 0;

Node \*head = NULL;

first = reverselist(first);

second = reverselist(second);

while(first != NULL && second != NULL)

{

int n = first->data + second->data + c;

Node \*temp = new Node(n%10);

temp->next = head;

head = temp;

c = n / 10;

first = first->next;

second = second->next;

}

while(first != NULL)

{

int n = first->data + c;

Node \*temp = new Node(n%10);

temp->next = head;

head = temp;

c = n / 10;

first = first->next;

}

while(second != NULL)

{

int n = second->data + c;

Node \*temp = new Node(n%10);

temp->next = head;

head = temp;

c = n / 10;

second = second->next;

}

if(c != 0)

{

Node \*temp = new Node(c);

temp->next = head;

head = temp;

}

return head;

}

**############################################################**

**5- Sorted insert for circular linked list**

Given a sorted circular linked list, the task is to insert a new node in this circular list so that it remains a sorted circular linked list.

**Example 1:**

**Input:**

LinkedList = 1->2->4

(the first and last node is connected,

i.e. 4 --> 1)

data = 2

**Output:** 1 2 2 4

**Example 2:**

**Input:**

LinkedList = 1->4->7->9

(the first and last node is connected,

i.e. 9 --> 1)

data = 5

**Output:** 1 4 5 7 9

**Your Task:**  
The task is to complete the function **sortedInsert**() which should insert the new node into the given circular linked list and return the head of the linkedlist.

**Expected Time Complexity:** O(N)  
**Expected Auxiliary Space:** O(1)

**Constraints:**  
0 <= N <= 105

Node \*sortedInsert(Node\* head, int data)

{

Node \*temp = head;

if(data < temp->data)

{

Node \*nd = new Node(data);

Node \*tr = head;

while(tr->next != head)

tr = tr->next;

tr->next = nd;

nd->next = head;

return nd;

}

while(temp->next != head)

{

if(data < temp->next->data)

{

Node \*nd = new Node(data);

nd->next = temp->next;

temp->next = nd;

return head;

}

temp = temp->next;

}

Node \*nd = new Node(data);

nd->next = temp->next;

temp->next = nd;

return head;

}

**############################################################**

**6- Split a Circular Linked List into two halves**

Given a **C**irular **L**inked **L**ist of size **N,** split it into two halves circular lists. If there are odd number of nodes in the given circular linked list then out of the resulting two halved lists, first list should have one node more than the second list. The resultant lists should also be circular lists and not linear lists.

**Example 1:**

**Input:**

Circular LinkedList: 1->5->7

**Output:**

1 5

7

**Example 2:**

**Input:**

Circular LinkedList: 2->6->1->5

**Output:**

2 6

1 5

**Your Task:**  
Your task is to complete the given function **splitList**(), which takes 3 input parameters: The address of the head of the linked list, addresses of the head of the first and second halved resultant lists and Set the **head1\_ref**and **head2\_ref** to the first resultant list and second resultant list respectively.

**Expected Time Complexity**: O(N)  
**Expected Auxilliary Space**: O(1)

**Constraints:**  
1 <= N <= 100

void splitList(Node \*head, Node \*\*head1, Node \*\*head2)

{

Node \*slow = head, \*fast = head;

while(fast->next != head && fast->next->next != head)

{ fast = fast->next->next;

slow = slow->next;

}

\*head1 = head;

\*head2 = slow->next;

slow->next = head;

if(fast->next == head)

fast->next = \*head2;

else

fast->next->next = \*head2;

}

**############################################################**

**7- Detect Loop in linked list**

Given a linked list of **N** nodes. The task is to check if the linked list has a loop. Linked list can contain self loop.

**Example 1:**

**Input:**

N = 3

value[] = {1,3,4}

x = 2

**Output:** True

**Explanation:** In above test case N = 3.

The linked list with nodes N = 3 is

given. Then value of x=2 is given which

means last node is connected with xth

node of linked list. Therefore, there

exists a loop.

**Example 2:**

**Input:**

N = 4

value[] = {1,8,3,4}

x = 0

**Output:** False

**Explanation:** For N = 4 ,x = 0 means

then lastNode->next = NULL, then

the Linked list does not contains

any loop.

**Your Task:**  
The task is to complete the function **detectloop**() which contains reference to the head as only argument. This function should return 1 if linked list contains loop, else return 0.

**Expected Time Complexity:** O(N)  
**Expected Auxiliary Space:** O(1)

**Constraints:**  
1 <= N <= 104  
1 <= Data on Node <= 103

bool detectLoop(Node\* head)

{

Node \*slow = head;

Node \*fast = head;

while(slow != NULL && fast != NULL && fast->next != NULL)

{

slow = slow->next;

fast = fast->next->next;

if(slow == fast)

return true;

}

return false;

}

**############################################################**

**8- Delete Middle of Linked List**

Given a singly linked list, delete **middle**of the linked list. For example, if given linked list is 1->2->**3**->4->5 then linked list should be modified to 1->2->4->5.  
If there are **even** nodes, then there would be **two middle**nodes, we need to delete the second middle element. For example, if given linked list is 1->2->3->4->5->6 then it should be modified to 1->2->3->5->6.  
If the input linked list is NULL or has 1 node, then it should return NULL

**Example 1:**

**Input:**

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

**Output:** 1 2 4 5

**Example 2:**

**Input:**

LinkedList: 2->4->6->7->5->1

**Output:** 2 4 6 5 1

**Your Task:**  
The task is to complete the function **deleteMid**() which should delete the middle element from the linked list and return the head of the modified linked list. If the linked list is empty then it should return NULL.

**Expected Time Complexity:**O(N).  
**Expected Auxiliary Space:**O(1).

**Constraints:**  
1 <= N <= 1000  
1 <= value <= 1000

Node\* deleteMid(Node\* head)

{

Node \*slow = head, \*fast = head;

if(head->next == NULL)

return NULL;

while(fast->next->next != NULL && fast->next->next->next != NULL)

{ slow = slow->next;

fast = fast->next->next;

}

Node \*t = slow->next->next;

free(slow->next);

slow->next = t;

return head;

}

**############################################################**

**9- Circular Linked List Delete at Position**

Given a linked list of size **n**, you have to**delete the node at position pos**of the linked list and return the new head. The position of initial node is 1.  
The tail of the circular linked list is connected to the head using next pointer.

**Example 1:**

**Input:**

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

(the first and last node are connected,

i.e. 5 --> 1)

position: 4

**Output:** 1 2 3 5

**Example 2:**

**Input:**

LinkedList: 2->5->7->8->99->100

(the first and last node are connected,

i.e. 5 --> 1)

position: 6

**Output:** 2 5 7 8 99

**Your Task:**  
The task is to complete the function **deleteAtPosition()** which takes**head reference** and **pos**as argument**and returns** reference to the **new head**node, which is then used to display the list. The **printing**is done **automatically**by the **driver code**.

**Expected Time Complexity:**O(n).  
**Expected Auxiliary Space:**O(1).

**Constraints:**  
2 <= number of nodes <= 103  
1 <= pos <= n

Node \* deleteAtPosition(Node \*head,int pos)

{

Node \*cur = head, \*rem;

if(pos == 1)

{ Node \*temp = head;

while(temp->next != head)

temp = temp->next;

temp->next = temp->next->next;

free(head);

return temp->next;

}

pos -= 2;

while(pos--)

cur = cur->next;

rem = cur->next;

cur->next = rem->next;

free(rem);

return head;

}

**############################################################**

**10- Delete without head pointer**

You are given a pointer/ reference to the node which is to be deleted from the linked list of **N**nodes. The task is to delete the node. Pointer/ reference to head node is not given.   
**Note:** No head reference is given to you. It is guaranteed that the node to be deleted isnot a tail nodein the linked list.

**Example 1:**

**Input:**

N = 2

value[] = {1,2}

node = 1

**Output:** 2

**Explanation:** After deleting 1 from the

linked list, we have remaining nodes

as 2.

**Example 2:**

**Input:**

N = 4

value[] = {10,20,4,30}

node = 20

**Output:** 10 4 30

**Explanation:** After deleting 20 from

the linked list, we have remaining

nodes as 10, 4 and 30.

**Your Task:**  
You only need to complete the **function deleteNode**that takes **reference**to the node that needs to be **deleted**. The **printing**is done **automatically**by the**driver code**.

**Expected Time Complexity** : O(1)  
**Expected Auxilliary Space** : O(1)

**Constraints:**  
1 <= N <= 103

void deleteNode(Node \*del)

{

Node \*t = del->next;

del->data = t->data;

del->next = t->next;

free(t);

}

**############################################################**

**11- Reverse a Linked List in groups of given size.**

Given a linked list of size **N**. The task is to reverse every **k** nodes (where k is an input to the function) in the linked list.

**Example 1:**

**Input:**

LinkedList: 1->2->2->4->5->6->7->8

K = 4

**Output:** 4 2 2 1 8 7 6 5

**Explanation:**

The first 4 elements 1,2,2,4 are reversed first

and then the next 4 elements 5,6,7,8. Hence, the

resultant linked list is 4->2->2->1->8->7->6->5.

**Example 2:**

**Input:**

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

K = 3

**Output:** 3 2 1 5 4

**Explanation:**

The first 3 elements are 1,2,3 are reversed

first and then elements 4,5 are reversed.Hence,

the resultant linked list is 3->2->1->5->4.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **reverse**() which should reverse the linked list in group of size **k**and return the head of the modified linked list.

**Expected Time Complexity**: O(N)  
**Expected Auxilliary Space**: O(1)

**Constraints:**

1 <= N <= 104  
1 <= k <= N

struct node \*reverse (struct node \*head, int k)

{

if(head == NULL || k == 1)

return head;

node \*ref = new node(0);

ref->next = head;

node \*a = ref, \*b = ref, \*c = ref, \*t = head;

int n = 0;

while(t->next != NULL)

{ t = t->next;

n++;

}

while(n > 0)

{ b = a->next;

c = b->next;

int i;

if(n >= k)

i = 1;

else

i = k - n;

while(i < k)

{

b->next = c->next;

c->next = a->next;

a->next = c;

c = b->next;

i++;

}

a = b;

n -= k;

}

return ref->next;

}

**############################################################**

**12- Intersection Point in Y Shapped Linked Lists**

Given two singly linked lists of size **N** and **M,**write a program to get the point where two linked lists intersect each other.

**Example 1:**

**Input:**

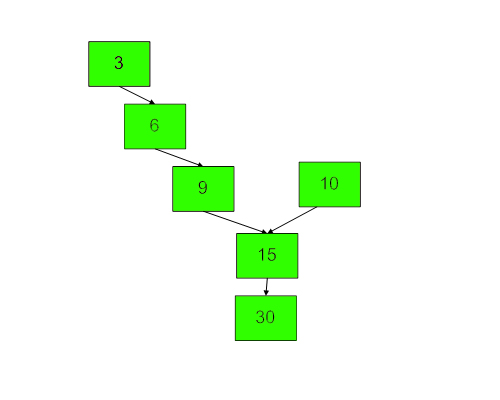
LinkList1 = 3->6->9->common

LinkList2 = 10->common

common = 15->30->NULL

**Output: 1**5

**Explanation:**



**Example 2:**

**Input:**

Linked List 1 = 4->1->common

Linked List 2 = 5->6->1->common

common = 8->4->5->NULL

**Output:** 8

**Explanation:**

**4 5**

**| |**

**1 6**

**\ /**

**8 ----- 1**

**|**

**4**

**|**

**5**

**|**

**NULL**

**Your Task:**  
You don't need to read input or print anything. The task is to complete the function **intersetPoint**() which takes the pointer to the head of linklist1(**head1**) and linklist2(**head2**) as input parameters and returns data value of a node where two linked lists intersect. If linked list do not merge at any point, then it should return **-1**.  
**Challenge**: Try to solve the problem without using any extra space.

**Expected Time Complexity:** O(N+M)  
**Expected Auxiliary Space:** O(1)

**Constraints:**  
1 ≤ N + M ≤ 2\*105  
-1000 ≤ value ≤ 1000

int intersectPoint(Node\* head1, Node\* head2)

{ /\*

if(head1 == NULL || head2 == NULL)

return -1;

unordered\_map<Node\*, int> mp;

Node \*t = head1;

while(t != NULL)

{ mp[t]++;

t = t->next;

}

t = head2;

while(t != NULL)

{ mp[t]++;

if(mp[t] > 1)

return t->data;

t = t->next;

}

return -1;

//Execution Time:1.71

\*/

if(head1 == NULL || head2 == NULL)

return -1;

Node \*a = head1, \*b = head2;

while(a != b)

{

if(a == NULL)

a = head2;

else

a = a->next;

if(b == NULL)

b = head1;

else

b = b->next;

}

return a->data;

//Execution Time:0.61

}

**############################################################**

**13- Quick Sort on Linked List**

Sort the given **L**inked **L**ist using quicksort. which takes **O(n^2)** time in worst case and **O(nLogn)** in average and best cases, otherwise you may get TLE.

**Input:**  
In this problem, method takes 1 argument: address of the **head** of the linked list. The function should not read any input from stdin/console.  
The struct Node has a data part which stores the **data** and a next pointer which points to the **next** element of the linked list.  
There are multiple test cases. For each test case, this method will be called individually.

**Output:**  
Set **\*headRef** to head of resultant linked list.

**User Task:**  
The task is to complete the function **quickSort**() which should set the \*headRef to head of the resultant linked list.

**Constraints:**  
1<=**T**<=100  
1<=**N**<=200

**Note:**If you use "Test" or "Expected Output Button" use below example format  
  
**Example:  
Input:**  
2  
3  
1 6 2  
4  
1 9 3 8

**Output:**  
1 2 6  
1 3 8 9

**Explanation:  
Testcase 1:** After sorting the nodes, we have 1, 2 and 6.  
**Testcase 2:** After sorting the nodes, we have 1, 3, 8 and 9.

struct node\* gett(struct node \*t)

{

while(t != NULL && t->next != NULL)

t = t->next;

return t;

}

struct node\* split(struct node \*l, struct node \*r, struct node \*\*nl, struct node \*\*nr)

{

struct node \*piv = r, \*a = NULL, \*b = l, \*tail = r;

while(b != piv)

{

if(b->data < piv->data)

{

if(\*nl == NULL)

\*nl = b;

a = b;

b = b->next;

}

else

{

if(a != NULL)

a->next = b->next;

struct node \*temp = b->next;

b->next = NULL;

tail->next = b;

tail = b;

b = temp;

}

}

if(\*nl == NULL)

\*nl = piv;

\*nr = tail;

return piv;

}

struct node\* quick(struct node \*l, struct node \*r)

{

if(!l || l == r)

return l;

node \*nl = NULL, \*nr = NULL;

struct node \*piv = split(l, r, &nl, &nr);

if(nl != piv)

{ struct node \*t = nl;

while(t->next != piv)

t = t->next;

t->next = NULL;

nl = quick(nl, t);

t = gett(nl);

t->next = piv;

}

piv->next = quick(piv->next, nr);

return nl;

}

void quickSort(struct node \*\*headRef)

{

\*headRef = quick(\*headRef, gett(\*headRef));

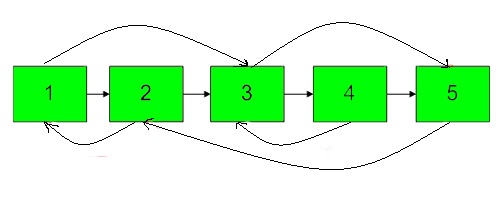
return;

}

**############################################################**

**14- Clone a linked list with next and random pointer**

You are given a special linked list with **N**nodes where each node has a next pointer pointing to its next node. You are also given **M** random pointers , where you will be given **M**number of pairs denoting two nodes **a** and **b**  **i.e.**a->arb = b**.**



**Example 1:**

**Input:**

N = 4, M = 2

value = {1,2,3,4}

pairs = {{1,2},{2,4}}

**Output:** 1

**Explanation:** In this test case, there

re 4 nodes in linked list.  Among these

4 nodes,  2 nodes have arbit pointer

set, rest two nodes have arbit pointer

as NULL. Second line tells us the value

of four nodes. The third line gives the

information about arbitrary pointers.

The first node arbit pointer is set to

node 2.  The second node arbit pointer

is set to node 4.

**Example 2:**

**Input:**

N = 4, M = 2

value[] = {1,3,5,9}

pairs[] = {{1,1},{3,4}}

**Output:** 1

**Explanation:** In the given testcase ,

applying the method as stated in the

above example, the output will be 1.

**Your Task:**  
The task is to complete the function **copyList**() which takes one argument the head of the linked list to be cloned and should **return** the head of the cloned linked list.  
**NOTE :**If their is any node whose arbitrary pointer is not given then its by default null.

**Expected Time Complexity** : O(n)  
**Expected Auxilliary Space**: O(1)

**Constraints:**  
1 <= N <= 100  
1 <= M <= N  
1 <= a, b <= 100

Node \*copyList(Node \*head)

{

Node \*temp = head;

while(temp != NULL)

{ Node \*nxt = temp->next;

temp->next = new Node(temp->data);

temp->next->next = nxt;

temp = nxt;

}

temp = head;

while(temp != NULL)

{

if(temp->next != NULL)

temp->next->arb = temp->arb? temp->arb->next: temp->arb;

temp = temp->next? temp->next->next: temp->next;

}

temp = head;

Node \*newhead = head->next;

Node \*temp2 = newhead;

while(temp != NULL && temp2 != NULL)

{

temp->next = temp->next? temp->next->next: temp->next;

temp2->next = temp2->next? temp2->next->next: temp2->next;

temp = temp->next;

temp2 = temp2->next;

}

return newhead;

}

**############################################################**

**14- Merge Sort on Doubly Linked List**

Given Pointer/Reference to the head of a doubly linked list of N nodes, the task is to **Sort the given doubly linked list using Merge Sort**in both **non-decreasing** and **non-increasing** order.

**Example 1:**

**Input:**

N = 8

value[] = {7,3,5,2,6,4,1,8}

**Output:**

1 2 3 4 5 6 7 8

8 7 6 5 4 3 2 1

**Explanation:** After sorting the given

linked list in both ways, resultant

matrix will be as given in the first

two line of output, where first line

is the output for non-decreasing

order and next line is for non-

increasing order.

**Example 2:**

**Input:**

N = 5

value[] = {9,15,0,-1,0}

**Output:**

-1 0 0 9 15

15 9 0 0 -1

**Explanation:** After sorting the given

linked list in both ways, the

resultant list will be -1 0 0 9 15

in non-decreasing order and

15 9 0 0 -1 in non-increasing order.

**Your Task:**  
The task is to complete the function **sortDoubly**() which sorts the doubly linked list. The **printing**is done **automatically**by the**driver code**.

**Constraints:**  
1 <= N <= 105

Node\* split(Node \*head)

{

Node \*slow = head, \*fast = head;

while(fast->next != NULL && fast->next->next)

{ slow = slow->next;

fast = fast->next->next;

}

Node \*temp = slow->next;

slow->next = NULL;

return temp;

}

Node\* merge(Node \*left, Node \*right)

{

if(left == NULL)

return right;

if(right == NULL)

return left;

if(left->data < right->data)

{

left->next = merge(left->next, right);

left->next->prev = left;

left->prev = NULL;

return left;

}

else

{

right->next = merge(left, right->next);

right->next->prev = right;

right->prev = NULL;

return right;

}

}

struct Node \*sortDoubly(struct Node \*left)

{

if(left == NULL || left->next == NULL)

return left;

Node \*right = split(left);

left = sortDoubly(left);

right = sortDoubly(right);

left = merge(left, right);

return left;

}

**############################################################**

**15- QuickSort on Doubly Linked List**

Sort the given doubly linked list of size **N** using quicksort. Just complete the partition function using the quicksort technique.

**Example 1:**

**Input:**

LinkedList: 4->2->9

**Output:**

2 4 9

**Example 2:**

**Input:**

LinkedList: 1->4->9->2

**Output:**

1 2 4 9

**Your Task:**  
Your task is to complete the given function **partition**(), which accepts the first and last node of the given linked list as input parameters and returns the pivot's address.

**Expected Time Complexity**: O(NlogN)  
**Expected Auxilliary Space**: O(1)

**Constraints:**  
1 <= N <= 200

Node\* partition(Node \*l, Node \*r)

{

Node \*j = l;

for (Node \*i = l; i != r; i = i->next)

{

if (i->data < h->data)

{

swap(&i->data, &j->data);

j = j->next;

}

}

swap(&r->data, &j->data);

return j;

}

**############################################################**