Data structures concepts and programming questions

What is a Data Structure?

A data structure is a way of organizing the data so that the data can be used efficiently. Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. For example, B-trees are particularly well-suited for implementation of databases, while compiler implementations usually use hash tables to look up identifiers.

What are linear and non linear data Structures?

- **Linear:** A data structure is said to be linear if its elements form a sequence or a linear list. Examples: Array. Linked List, Stacks and Queues
- **Non-Linear:** A data structure is said to be non-linear if traversal of nodes is nonlinear in nature. Example: Graph and Trees.

What are the various operations that can be performed on different Data Structures?

- **Insertion** ? Add a new data item in the given collection of data items.
- **Deletion** ? Delete an existing data item from the given collection of data items.
- **Traversal** ? Access each data item exactly once so that it can be processed.
- **Searching** ? Find out the location of the data item if it exists in the given collection of data items.

- **Sorting** ? Arranging the data items in some order i.e. in ascending or descending order in case of numerical data and in dictionary order in case of alphanumeric data.

What is a Linked List?

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers. In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

How is an Array different from Linked List?

- The size of the arrays is fixed, Linked Lists are Dynamic in size.
- Inserting and deleting a new element in an array of elements is expensive, Whereas both insertion and deletion can easily be done in Linked Lists.
- Random access is not allowed in Linked Listed.
- Extra memory space for a pointer is required with each element of the Linked list.
- Arrays have better cache locality that can make a pretty big difference in performance.

https://www.geeksforgeeks.org/linked-list-vs-array/

What is Stack and where it can be used?

Stack is a linear data structure which orders LIFO(Last In First Out) or FILO(First In Last Out) for accessing elements. Basic operations of stack are : Push, Pop , Peek.

Applications of Stack:

1. Infix to Postfix Conversion using Stack
2. Evaluation of Postfix Expression

3. Reverse a String using Stack
4. Implement two stacks in an array
5. Check for balanced parentheses in an expression

Implement stack using linked list

https://www.geeksforgeeks.org/implement-a-stack-using-singly-linked-list/

What is a Queue, how it is different from stack and how is it implemented?

Queue is a linear structure which follows the order is **F**irst **I**n **F**irst **O**ut (FIFO) to access elements. Mainly the following are basic operations on queue: **Enqueue, Dequeue**, **Front, Rear**

The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added. Both Queues and Stacks can be implemented using Arrays and Linked Lists.

Implement queue using linked list

https://www.geeksforgeeks.org/queue-linked-list-implementation/

What are Infix, prefix, Postfix notations?

- **Infix notation:** X **+** Y – Operators are written in-between their operands. This is the usual way we write expressions. An expression such as

  ```
  A * ( B + C ) / D
  ```

- **Postfix notation (also known as "Reverse Polish notation"):** X Y **+** Operators are written after their operands. The infix expression given above is equivalent to

  ```
  A B C + * D/
  ```

- **Prefix notation (also known as “Polish notation”):** + X Y Operators are written before their operands. The expressions given above are equivalent to

```
/ * A + B C D
```

Converting between these notations: Click here

What is a Linked List and What are its types?

A linked list is a linear data structure (like arrays) where each element is a separate object. Each element (that is node) of a list is comprising of two items – the data and a reference to the next node.Types of Linked List :

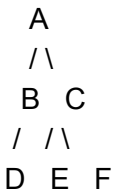
1. **Singly Linked List :** In this type of linked list, every node stores address or reference of next node in list and the last node has next address or reference as NULL. For example 1->2->3->4->NULL
2. **Doubly Linked List :** Here, here are two references associated with each node, One of the reference points to the next node and one to the previous node. Eg. NULL<-1<->2<->3->NULL
3. **Circular Linked List :** Circular linked list is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list. Eg. 1->2->3->1 [The next pointer of last node is pointing to the first]

Difference Between BFS and DFS

Breadth First Search

BFS stands for **Breadth First Search** is a vertex based technique for finding the shortest path in a graph. It uses a Queue data structure which follows first in first out. In BFS, one vertex is selected at a time when it is visited and marked then its adjacent are visited and stored in the queue. It is slower than DFS.

Ex-

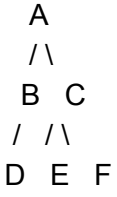


Output is:
A, B, C, D, E, F

Depth First Search

DFS stands for **Depth First Search** is an edge based technique. It uses the Stack data structure, performs two stages, first visited vertices are pushed into stack and second if there are no vertices then visited vertices are popped.

Ex-



Output is:

A, B, D, C, E, F

S.NO	BFS	DFS
1.	BFS stands for Breadth First Search.	DFS stands for Depth First Search.
2.	BFS(Breadth First Search) uses Queue data structure for finding the shortest path.	DFS(Depth First Search) uses Stack data structure.
3.	BFS can be used to find single source shortest path in an unweighted graph, because in BFS, we reach a vertex with minimum number of edges from a source vertex.	In DFS, we might traverse through more edges to reach a destination vertex from a source.
3.	BFS is more suitable for searching vertices which are closer to the given source.	DFS is more suitable when there are solutions away from source.

4.	BFS considers all neighbors first and therefore not suitable for decision making trees used in games or puzzles.	DFS is more suitable for game or puzzle problems. We make a decision, then explore all paths through this decision. And if this decision leads to win situation, we stop.
5.	The Time complexity of BFS is O(V + E), where V stands for vertices and E stands for edges.	The Time complexity of DFS is also O(V + E), where V stands for vertices and E stands for edges.

Which data structures are used for BFS and DFS of a graph?

- Queue is used for BFS
- Stack is used for DFS. DFS can also be implemented using recursion (Note that recursion also uses function call stack).

Binary Tree

Binary Tree: A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

Binary Tree Representation in C++: A tree is represented by a pointer to the topmost node in a tree. If the tree is empty, then the value of the root is NULL.

A Tree node contains the following parts.

1. Data

2. Pointer to left child

3. Pointer to right child

```
Class node

{

  int data;

   node *left;

   node *right;

};
```

Binary Search Tree

Binary Search Tree, is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.
  There must be no duplicate nodes.

Data Structures complexities

Data Structure	**Time Complexity**								**Space Complexity**
	Average				**Worst**				**Worst**
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)

Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)

AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Array Sorting Algorithms complexities

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)

Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Data structures programming questions

Linked List questions

Add node at the start of linked list

```
void addAtHead(int n){
        Node *temp=new Node();
        temp->next=NULL;
        temp->data=n;
        if(head==NULL){
                head=temp;

        }
        else{
                temp->next=head;
                head=temp;
        }
}
```

Add node at the end of linked list

```
void addAtTail(int n){
        Node *temp=new Node();
        temp->next=NULL;
        temp->data=n;
        if(head==NULL){
                head=temp;

        }
        else{
                Node *tmp=head;
                while(tmp->next!=NULL){
                        tmp=tmp->next;
                }
```

```
                    tmp->next=temp;
                }

            }
```

Delete an element in linked list

```
void removeNode(int n){
                if(head==NULL) cout<<"List is khaali\n";
                else{
                   Node *temp=head->next;
                   Node *prev=head;
                   while(temp!=NULL){
                        if(temp->data==n){
                            prev->next=temp->next;
                            delete temp;
                        }
                        else {
                          prev = temp;
                          temp = temp->next;
                 }
                   }
```

Delete N nodes after M nodes in a linked list till the end of list

```
void skipMdeleteN(Node *head, int M, int N)
{
   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++)
    {
        Node *temp = t;
        t = t->next;
        free(temp);
    }

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

    // Set current pointer for next iteration
    curr = t;
  }
}
```

Delete a linked list node at a given position

```
void deleteNode(Node *node_ptr)
{
    Node *temp = node_ptr->next;
    node_ptr->data = temp->data;
    node_ptr->next = temp->next;
    Delete temp;
}
```

Print Linked list

```
    void showList(){
            Node *temp=head;
            while(temp!=NULL){
                cout<<temp->data<<" ";
                temp=temp->next;
            }
        }
```

Find the middle element of a linked list in one iteration

```
    void findMiddle(){
            Node *slow,*fast;
            slow=fast=head;
            while(fast!=NULL && fast->next!=NULL){
                slow=slow->next;
                fast=fast->next->next;
            }
            cout<<"\nMiddle element is "<<slow->data<<endl;
        }
```

Find and remove loop in linked list

```
    Int findLoop(){
            Node *slow = head, *fast = head;
          while (slow && fast & fast->next) {
              slow = slow->next;
              fast = fast->next->next;
```

```
                if(slow == fast) {
                    removeLoop(slow_p, list);

            /* Return 1 to indicate that loop is found */
            return 1;

                }
            }
            cout<<"\nno loop detected\n";
            return;

          }
/* Function to remove loop.
 loop_node --> Pointer to one of the loop nodes
 head -->  Pointer to the start node of the linked list */
void removeLoop(struct Node* loop_node, struct Node* head)
{
    struct Node* ptr1;
    struct Node* ptr2;

    /* Set a pointer to the beginning of the Linked List and
      move it one by one to find the first node which is
      part of the Linked List */
    ptr1 = head;
    while (1) {
        /* Now start a pointer from loop_node and check if it ever
       reaches ptr2 */
        ptr2 = loop_node;
        while (ptr2->next != loop_node && ptr2->next != ptr1)
            ptr2 = ptr2->next;

        /* If ptr2 reahced ptr1 then there is a loop. So break the
        loop */
        if (ptr2->next == ptr1)
            break;

        /* If ptr2 did't reach ptr1 then try the next node after ptr1 */
        ptr1 = ptr1->next;
    }

    /* After the end of loop ptr2 is the last node of the loop. So
     make next of ptr2 as NULL */
    ptr2->next = NULL;
```

```
}
```

Find the nth element in a linked list from the end

1) Calculate the length of the Linked List. Let the length be len.
2) Print the (len – n + 1)th node from the beginning of the Linked List.

```
void printNthFromLast(struct Node* head, int n)
{
    int len = 0, i;
    struct Node* temp = head;

    // count the number of nodes in Linked List
    while (temp != NULL) {
        temp = temp->next;
        len++;
    }

    // check if value of n is not
    // more than length of the linked list
    if (len < n)
        return;

    temp = head;

    // get the (len-n+1)th node from the beginning
    for (i = 1; i < len - n + 1; i++)
        temp = temp->next;

    cout << temp->data;

    return;
}
```

Recursive approach

```
void printNthFromLast(struct Node* head, int n)
{
    static int i = 0;
    if (head == NULL)
        return;
    printNthFromLast(head->next, n);
```

```
    if (++i == n)
        printf("%d", head->data);
}
```

Find the nth element in a linked list from the end using two pointers

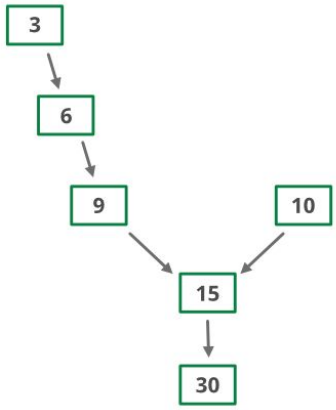
Maintain two pointers – reference pointer and main pointer. Initialize both reference and main pointers to head. First, move the reference pointer to n nodes from head. Now move both pointers one by one until the reference pointer reaches the end. Now the main pointer will point to nth node from the end. Return the main pointer.

```
void printNthFromLast(struct Node *head, int n)
{
  struct Node *main_ptr = head;
  struct Node *ref_ptr = head;

  int count = 0;
  if(head != NULL)
  {
     while( count < n )
     {
        if(ref_ptr == NULL)
        {
           printf("%d is greater than the no. of "
                    "nodes in list", n);
           return;
        }
        ref_ptr = ref_ptr->next;
        count++;
     } /* End of while*/

     while(ref_ptr != NULL)
     {
        main_ptr = main_ptr->next;
        ref_ptr  = ref_ptr->next;
     }
     printf("Node no. %d from last is %d ",
               n, main_ptr->data);
  }
}
```

Find the intersection of 2 singly linked lists in O(n)

- Get count of the nodes in the first list, let count be c1.
- Get count of the nodes in the second list, let count be c2.
- Get the difference of counts **d = abs(c1 – c2)**
- 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.
- 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).

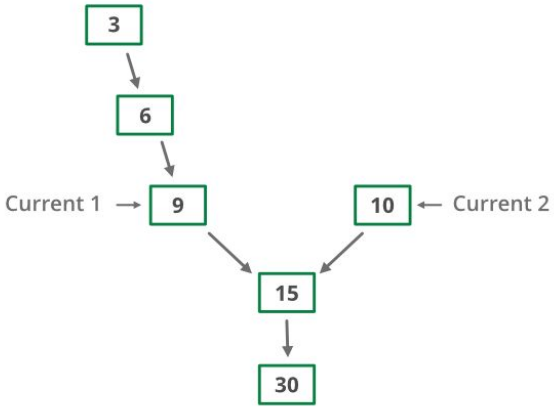
Below image is a dry run of the above approach:

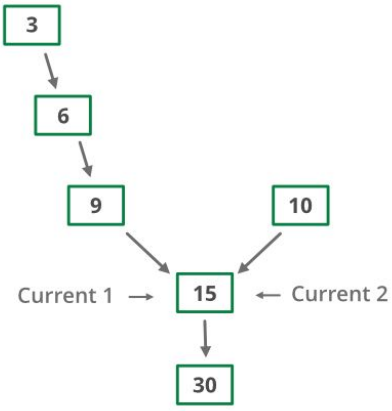


C1 = 5

C2 = 3

Traverse till 3rd node
in the bigger list





Intersection node = 15

```
/* Takes head pointer of the linked list and
returns the count of nodes in the list */
int getCount(Node* head)
{
    Node* current = head;

    // Counter to store count of nodes
    int count = 0;

    // Iterate till NULL
    while (current != NULL) {

        // Increase the counter
        count++;

        // Move the Node ahead
        current = current->next;
    }

    return count;
}

int getIntersectionNode(Node* head1, Node* head2)
{

    // Count the number of nodes in
    // both the linked list
    int c1 = getCount(head1);
    int c2 = getCount(head2);
    int d;

    // If first is greater
    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, Node* head1, Node* head2)
{
    // Stand at the starting of the bigger list
    Node* current1 = head1;
    Node* current2 = head2;

    // Move the pointer forward
    for (int i = 0; i < d; i++) {
        if (current1 == NULL) {
            return -1;
        }
        current1 = current1->next;
    }

    // Move both pointers of both list till they
    // intersect with each other
    while (current1 != NULL && current2 != NULL) {
        if (current1 == current2)
            return current1->data;

        // Move both the pointers forward
        current1 = current1->next;
        current2 = current2->next;
    }

    return -1;
}
```

Reverse a linked list in one iteration

```
 void reverseList(){
          Node *curr=head;
          Node *next=NULL;
          Node *prev=NULL;
          while(curr!=NULL){
              next=curr->next;
              curr->next=prev;
```

```
            prev=curr;
            curr=next;
        }
        head=prev;
    }
```

Reverse a linked list using recursion

In this approach of reversing a linked list by passing a single pointer what we are trying to do is that we are making the previous node of the current node as his next node to reverse the linked list.

```
Node* reverse(Node* node)
    {
        if (node == NULL)
            return NULL;
        if (node->next == NULL) {
            head = node;
            return node;
        }
        Node* node1 = reverse(node->next);
        node1->next = node;
        node->next = NULL;
        return node;
    }
```

Move even node to end of linked list

The idea is to split the linked list into two: one containing all even nodes and another containing all odd nodes. And finally attach the odd node linked list after the even node linked list.
To split the Linked List, traverse the original Linked List and move all odd nodes to a separate Linked List of all odd nodes. At the end of the loop, the original list will have all the even nodes and the odd node list will have all the odd nodes. To keep the ordering of all nodes the same, we must insert all the odd nodes at the end of the odd node list. And to do that in constant time, we must keep track of the last pointer in the odd node list.

```
void segregateEvenOdd(struct Node **head_ref)
{
    // Starting node of list having
    // even values.
    Node *evenStart = NULL;

    // Ending node of even values list.
    Node *evenEnd = NULL;

    // Starting node of odd values list.
    Node *oddStart = NULL;

    // Ending node of odd values list.
    Node *oddEnd = NULL;

    // Node to traverse the list.
    Node *currNode = *head_ref;

    while(currNode != NULL){
        int val = currNode -> data;

        // If current value is even, add
        // it to even values list.
        if(val % 2 == 0) {
            if(evenStart == NULL){
                evenStart = currNode;
                evenEnd = evenStart;
            }

            else{
                evenEnd -> next = currNode;
                evenEnd = evenEnd -> next;
            }
        }

        // If current value is odd, add
        // it to odd values list.
        else{
            if(oddStart == NULL){
                oddStart = currNode;
                oddEnd = oddStart;
            }
            else{
```

```
            oddEnd -> next = currNode;
            oddEnd = oddEnd -> next;
        }
    }

    // Move head pointer one step in
    // forward direction
    currNode = currNode -> next;
  }

  // If either odd list or even list is empty,
  // no change is required as all elements
  // are either even or odd.
  if(oddStart == NULL || evenStart == NULL){
    return;
  }

  // Add odd list after even list.
  evenEnd -> next = oddStart;
  oddEnd -> next = NULL;

  // Modify head pointer to
  // starting of even list.
  *head_ref = evenStart;
```

Find the Second Largest Element in a Linked List

```
first = second = INT_MIN;

  struct Node* temp = head;

  while (temp != NULL) {
    if (temp->data > first) {
      second = first;
      first = temp->data;
    }

    // If current node's data is in between
    // first and second then update second
    else if (temp->data > second && temp->data != first)
      second = temp->data;

    temp = temp->next;
```

```
  }

  if (second == INT_MIN)
     cout << "There is no second largest element\n";
  else
     cout << "The second largest element is " << second;
```

Remove duplicates from an unsorted linked list

```
/* Function to remove duplicates from a
   unsorted linked list */
void removeDuplicates(struct Node *start)
{
    struct Node *ptr1, *ptr2, *dup;
    ptr1 = start;

    /* Pick elements one by one */
    while (ptr1 != NULL && ptr1->next != NULL)
    {
        ptr2 = ptr1;

        /* Compare the picked element with rest
           of the elements */
        while (ptr2->next != NULL)
        {
            /* If duplicate then delete it */
            if (ptr1->data == ptr2->next->data)
            {
                /* sequence of steps is important here */
                dup = ptr2->next;
                ptr2->next = ptr2->next->next;
                delete(dup);
            }
            else /* This is tricky */
                ptr2 = ptr2->next;
        }
        ptr1 = ptr1->next;
    }
}
```

Check whether linked list is palindrome or not

```
/* Function to check if given linked list is
  palindrome or not */
bool isPalindrome(struct Node* head)
{
    struct Node *slow_ptr = head, *fast_ptr = head;
    struct Node *second_half, *prev_of_slow_ptr = head;
    struct Node* midnode = NULL; // To handle odd size list
    bool res = true; // initialize result

    if (head != NULL && head->next != NULL) {
        /* Get the middle of the list. Move slow_ptr by 1
          and fast_ptrr by 2, slow_ptr will have the middle
          node */
        while (fast_ptr != NULL && fast_ptr->next != NULL) {
            fast_ptr = fast_ptr->next->next;

            /*We need previous of the slow_ptr for
             linked lists  with odd elements */
            prev_of_slow_ptr = slow_ptr;
            slow_ptr = slow_ptr->next;
        }

        /* fast_ptr would become NULL when there are even elements in list.
           And not NULL for odd elements. We need to skip the middle node
           for odd case and store it somewhere so that we can restore the
           original list*/
        if (fast_ptr != NULL) {
            midnode = slow_ptr;
            slow_ptr = slow_ptr->next;
        }

        // Now reverse the second half and compare it with first half
        second_half = slow_ptr;
        prev_of_slow_ptr->next = NULL; // NULL terminate first half
        reverse(&second_half); // Reverse the second half
        res = compareLists(head, second_half); // compare

        /* Construct the original list back */
        reverse(&second_half); // Reverse the second half again
```

```
        // If there was a mid node (odd size case) which
        // was not part of either first half or second half.
        if (midnode != NULL) {
            prev_of_slow_ptr->next = midnode;
            midnode->next = second_half;
        }
        else
            prev_of_slow_ptr->next = second_half;
    }
    return res;
}

/* Function to reverse the linked list  Note that this
    function may change the head */
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;
}

/* Function to check if two input lists have same data*/
bool compareLists(struct Node* head1, struct Node* head2)
{
    struct Node* temp1 = head1;
    struct Node* temp2 = head2;

    while (temp1 && temp2) {
        if (temp1->data == temp2->data) {
            temp1 = temp1->next;
            temp2 = temp2->next;
        }
        else
            return 0;
    }
```

```
    /* Both are empty reurn 1*/
    if (temp1 == NULL && temp2 == NULL)
        return 1;

    /* Will reach here when one is NULL
      and other is not */
    return 0;
}
```

Binary Search Tree questions

Search an element in BST

```
/ /C++ function to search a given key in a given BST
node* search(node* root, int key)
{
        // Base Cases: root is null or key is present at root
        if (root == NULL || root->key == key)
        return root;

        // Key is greater than root's key
        if (root->key < key)
        return search(root->right, key);

        // Key is smaller than root's key
        return search(root->left, key);
}
```

Insert an element in BST

```
BST* Insert(BST *root, int value)
{
    if(!root)
    {
        // Insert the first node, if root is NULL.
        return new BST(value);
    }
```

```
    // Insert data.
    if(value > root->data)
    {
        // Insert right node data, if the 'value'
        // to be inserted is greater than 'root' node data.

        // Process right nodes.
        root->right = Insert(root->right, value);
    }
    else
    {
        // Insert left node data, if the 'value'
        // to be inserted is greater than 'root' node data.

        // Process left nodes.
        root->left = Insert(root->left, value);
    }

    // Return 'root' node, after insertion.
    return root;
}
```

A program to check if a binary tree is BST or not

```
bool isBST(Node* root, Node* l=NULL, Node* r=NULL)
{
    // Base condition
    if (root == NULL)
        return true;

    // if left node exist then check it has
    // correct data or not i.e. left node's data
    // should be less than root's data
    if (l != NULL and root->data <= l->data)
        return false;

    // if right node exist then check it has
    // correct data or not i.e. right node's data
    // should be greater than root's data
    if (r != NULL and root->data >= r->data)
        return false;
```

```
    // check recursively for every node.
    return isBST(root->left, l, root) and
           isBST(root->right, root, r);
}
```

Check if two binary trees are identical or not

The idea is to traverse both trees and compare value at their root node. If the value matches, we recursively check if the left subtree of the first tree is identical to the left subtree of the second tree and the right subtree of the first tree is identical to the right subtree of the second tree. If the value at their root node differs, the trees violate data property. If at any point in the recursion, the first tree is empty & second tree is non-empty or second tree is empty & first tree is non-empty, the trees violate structural property and they cannot be identical.

```
// Recursive function to check if two given binary trees are identical or not
int isIdentical(Node* x, Node* y)
{
        // if both trees are empty, return true
        if (x == nullptr && y == nullptr)
                return 1;

        // if both trees are non-empty and value of their root node matches,
        // recur for their left and right sub-tree
        return (x && y) && (x->key == y->key) &&
                        isIdentical(x->left, y->left) &&
                        isIdentical(x->right, y->right);
}
```

Find the node with minimum value in a Binary Search Tree

```
int minValue(struct node* node)
{
struct node* current = node;

/* loop down to find the leftmost leaf */
while (current->left != NULL)
{
    current = current->left;
```

```
}
return(current->data);
}
```

Write a program to Calculate Size of a tree using recursion

```
int size(node* node)
{
    if (node == NULL)
        return 0;
    else
        return(size(node->left) + 1 + size(node->right));
}
```

Write a Program to Find the Maximum Depth or Height of a Tree

```
int maxDepth(node* node)
{
    if (node == NULL)
        return 0;
    else
    {
        /* compute the depth of each subtree */
        int lDepth = maxDepth(node->left);
        int rDepth = maxDepth(node->right);

        /* use the larger one */
        if (lDepth > rDepth)
            return(lDepth + 1);
        else return(rDepth + 1);
    }
}
```

Program to count leaf nodes in a binary tree

```
int getLeafCount(struct node* node)
{
    if(node == NULL)
        return 0;
    if(node->left == NULL && node->right == NULL)
        return 1;
```

```
    else
        return getLeafCount(node->left)+
            getLeafCount(node->right);
}
```

A program to print path having ‘k’ length

```
void printKPathUtil(Node *root, vector<int>& path,
                                            int k)
{
    // empty node
    if (!root)
        return;

    // add current node to the path
    path.push_back(root->data);

    // check if there's any k sum path
    // in the left sub-tree.
    printKPathUtil(root->left, path, k);

    // check if there's any k sum path
    // in the right subtree.
    printKPathUtil(root->right, path, k);

    // check if there's any k sum path that
    // terminates at this node
    // Traverse the entire path as
    // there can be negative elements too
    int f = 0;
    for (int j=path.size()-1; j>=0; j--)
    {
        f += path[j];

        // If path sum is k, print the path
        if (f == k)
            printVector(path, j);
          break;
    }

    // Remove the current element from the path
    path.pop_back();
}
```