Geeky Questions

Devesh

2011

BIT Patna

EQUILIBRIUM INDEX OF AN ARRAY

Equilibrium index of an array is an index such that the sum of elements at lower indexes is equal to the sum of elements at higher indexes. For example, in an array A:

A[0] = -7, A[1] = 1, A[2] = 5, A[3] = 2, A[4] = -4, A[5] = 3, A[6]=0

3 is an equilibrium index, because:
A[0] + A[1] + A[2] = A[4] + A[5] + A[6]

6 is also an equilibrium index, because sum of zero elements is zero, i.e., A[0] + A[1] + A[2] + A[3] + A[4] + A[5]=0

7 is not an equilibrium index, because it is not a valid index of array A.

Write a function *int equilibrium(int[] arr, int n)*; that given a sequence arr[] of size n, returns an equilibrium index (if any) or -1 if no equilibrium indexes exist.

Method 1 (Simple but inefficient)
Use two loops. Outer loop iterates through all the element and inner loop finds out whether the current index picked by the outer loop is equilibrium index or not. Time complexity of this solution is O(n^2).

```
#include <stdio.h>

int equilibrium(int arr[], int n)
{
  int i, j;
  int leftsum, rightsum;

  /* Check for indexes one by one until an equilibrium
    index is found */
  for ( i = 0; i < n; ++i)
  {
    leftsum = 0;  // initialize left sum for current index i
    rightsum = 0; // initialize right sum for current index i

    /* get left sum */
    for ( j = 0; j < i; j++)
      leftsum  += arr[j];

    /* get right sum */
    for( j = i+1; j < n; j++)
      rightsum += arr[j];

    /* if leftsum and rightsum are same, then we are done */
    if (leftsum == rightsum)
      return i;
    }
```

```
  /* return -1 if no equilibrium index is found */
  return -1;
}

int main()
{
  int arr[] = {-7, 1, 5, 2, -4, 3, 0};
  int arr_size = sizeof(arr)/sizeof(arr[0]);
  printf("%d\n", equilibrium(arr, arr_size));

  getchar();
  return 0;
}
```

Time Complexity: O(n^2)

Method 2 (Tricky and Efficient)
The idea is to get total sum of array first. Then Iterate through the array and keep updating the left sum which is initialized as zero. In the loop, we can get right sum by subtracting the elements one by one.

```
1) Initialize leftsum  as 0
2) Get the total sum of the array as sum
3) Iterate through the array and for each index i, do following.
    a)  Update sum to get the right sum.
           sum = sum - arr[i]
       // sum is now right sum
    b) If leftsum is equal to sum, then return current index.
    c) leftsum = leftsum + arr[i] // update leftsum for next iteration.
4) return -1 // If we come out of loop without returning then
             // there is no equilibrium index
```

?

```
#include <stdio.h>

int equilibrium(int arr[], int n)
{
   int sum = 0;      // initialize sum of whole array
   int leftsum = 0; // initialize leftsum
   int i;

   /* Find sum of the whole array */
   for (i = 0; i < n; ++i)
        sum += arr[i];

   for( i = 0; i < n; ++i)
   {
      sum -= arr[i]; // sum is now right sum for index i

      if(leftsum == sum)
        return i;
```

```
      leftsum += arr[i];
   }

    /* If no equilibrium index found, then return 0 */
    return -1;
}

int main()
{
  int arr[] = {-7, 1, 5, 2, -4, 3, 0};
  int arr_size = sizeof(arr)/sizeof(arr[0]);
  printf("First equilibrium index is %d\n", equilibrium(arr, arr_size));

  getchar();
  return 0;
}
```

Time Complexity: O(n)

We can remove the return statement and add a print statement to print all equilibrium indexes instead of returning only one.

LINKED LIST VS. ARRAY

Both Arrays and Linked List can be used to store linear data of similar types, but they both have some advantages and disadvantages over each other.

Following are the points in favour of Linked Lists.

(1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage, and in practical uses, upper limit is rarely reached.

(2) Inserting a new element in an array of elements is expensive, because room has to be created for the new elements and to create room existing elements have to shifted.

For example, suppose we maintain a sorted list of IDs in an array id[].

id[] = [1000, 1010, 1050, 2000, 2040, ...].

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).

Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

So Linked list provides following two advantages over arrays
1) Dynamic size
2) Ease of insertion/deletion

Linked lists have following drawbacks:
1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists.
2) Extra memory space for a pointer is required with each element of the list.
3) Arrays have better cache locality that can make a pretty big difference in performance.

References:
http://cslibrary.stanford.edu/103/LinkedListBasics.pdf

TURN AN IMAGE BY 90 DEGREE

Given an image, how will you turn it by 90 degrees? A vague question. Minimize the browser and try your solution before going further.

An image can be treated as 2D matrix which can be stored in a buffer. We are provided with matrix dimensions and it's base address. How can we turn it?

For example see the below picture,

```
* * * ^ * * *
* * * | * * *
* * * | * * *
* * * | * * *
```

After rotating right, it appears (observe arrow direction)

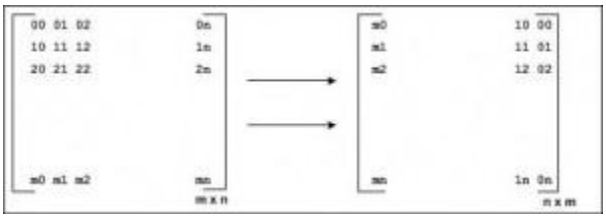
```
* * * *
* * * *
* * * *
-- - - >
* * * *
* * * *
* * * *
```

The idea is simple. Transform each row of source matrix into required column of final image. We will use an auxiliary buffer to transform the image.

From the above picture, we can observe that

```
first row of source ------> last column of destination
second row of source ------> last but-one column of destination
so ... on
last row of source ------> first column of destination
```

In pictorial form, we can represent the above transformations of an (m x n) matrix into (n x m) matrix,



Transformations

If you have not attempted, atleast try your pseudo code now.

It will be easy to write our pseudo code. In C/C++ we will usually traverse matrix on row major order. Each row is transformed into different column of final image. We need to construct columns of final image. See the following algorithm (transformation)

```
for(r = 0; r < m; r++)
{
   for(c = 0; c < n; c++)
   {
      // Hint: Map each source element indices into
      // indices of destination matrix element.
       dest_buffer [ c ] [ m - r - 1 ] = source_buffer [ r ] [ c ];
   }
}
```

Note that there are various ways to implement the algorithm based on traversal of matrix, row major or column major order. We have two matrices and two ways (row and column major) to traverse each matrix. Hence, there can atleast be 4 different ways of transformation of source matrix into final matrix.

Code:

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

void displayMatrix(unsigned int const *p, unsigned int row, unsigned int col);
void rotate(unsigned int *pS, unsigned int *pD, unsigned int row, unsigned int
col);

int main()
{
    // declarations
    unsigned int image[][4] = {{1,2,3,4}, {5,6,7,8}, {9,10,11,12}};
    unsigned int *pSource;
    unsigned int *pDestination;
    unsigned int m, n;

    // setting initial values and memory allocation
    m = 3, n = 4, pSource = (unsigned int *)image;
```

```
    pDestination = (unsigned int *)malloc(sizeof(int)*m*n);

    // process each buffer
    displayMatrix(pSource, m, n);

    rotate(pSource, pDestination, m, n);

    displayMatrix(pDestination, n, m);

    free(pDestination);

    getchar();
    return 0;
}

void displayMatrix(unsigned int const *p, unsigned int r, unsigned int c)
{
    unsigned int row, col;
    printf("\n\n");

    for(row = 0; row < r; row++)
    {
        for(col = 0; col < c; col++)
        {
            printf("%d\t", *(p + row * c + col));
        }
        printf("\n");
    }

    printf("\n\n");
}

void rotate(unsigned int *pS, unsigned int *pD, unsigned int row, unsigned int
col)
{
    unsigned int r, c;
    for(r = 0; r < row; r++)
    {
        for(c = 0; c < col; c++)
        {
            *(pD + c * row + (row - r - 1)) = *(pS + r * col + c);
        }
    }
}
```

SEARCH IN A ROW WISE AND COLUMN WISE SORTED MATRIX

Given an n x n matrix, where every row and column is sorted in increasing order. Given a number x, how to decide whether this x is in the matrix. The designed algorithm should have linear time complexity.

1) Start with top right element
2) Loop: compare this element e with x
….i) if they are equal then return its position
…ii) e < x then move it to down (if out of bound of matrix then break return false)
...iii) e > x then move it to left (if out of bound of matrix then break return false)
3) repeat the i), ii) and iii) till you find element or returned false

Implementation:

```
#include<stdio.h>

/* Searches the element x in mat[][]. If the element is found,
    then prints its position and returns true, otherwise prints
    "not found" and returns false */
int search(int mat[4][4], int n, int x)
{
   int i = 0, j = n-1;  //set indexes for top right element
   while ( i < n && j >= 0 )
   {
      if ( mat[i][j] == x )
      {
         printf("\n Found at %d, %d", i, j);
         return 1;
      }
      if ( mat[i][j] > x )
        j--;
      else //  if mat[i][j] < x
        i++;
   }

   printf("\n Element not found");
   return 0;  // if ( i==n || j== -1 )
}

// driver program to test above function
int main()
{
  int mat[4][4] = { {10, 20, 30, 40},
                    {15, 25, 35, 45},
                    {27, 29, 37, 48},
                    {32, 33, 39, 50},
                  };
  search(mat, 4, 29);
  getchar();
  return 0;
}
```

Time Complexity: O(n)

The above approach will also work for m x n matrix (not only for n x n). Complexity would be O(m + n)

NEXT GREATER ELEMENT

Given an array, print the Next Greater Element (NGE) for every element. The Next greater Element for an element x is the first greater element on the right side of x in array. Elements for which no greater element exist, consider next greater element as -1.

Examples:
a) For any array, rightmost element always has next greater element as -1.
b) For an array which is sorted in decreasing order, all elements have next greater element as -1.
c) For the input array [4, 5, 2, 25}, the next greater elements for each element are as follows.

```
Element       NGE
   4      -->   5
   5      -->   25
   2      -->   25
   25     -->   -1
```

d) For the input array [13, 7, 6, 12}, the next greater elements for each element are as follows.

```
  Element        NGE
   13      -->    -1
   7       -->     12
   6       -->     12
   12      -->    -1
```

Method 1 (Simple)
Use two loops: The outer loop picks all the elements one by one. The inner loop looks for the first greater element for the element picked by outer loop. If a greater element is found then that element is printed as next, otherwise -1 is printed.

Thanks to Sachin for providing following code.

?

```
#include<stdio.h>
/* prints element and NGE pair for all elements of
arr[] of size n */
void printNGE(int arr[], int n)
{
    int next = -1;
    int i = 0;
    int j = 0;
    for (i=0; i<n; i++)
    {
        next = -1;
        for (j = i+1; j<n; j++)
        {
            if (arr[i] < arr[j])
            {
                next = arr[j];
                break;
            }
```

```
        }
        printf("%d -> %d\n", arr[i], next);
    }
}

int main()
{
    int arr[]= {11, 13, 21, 3};
    int n = sizeof(arr)/sizeof(arr[0]);
    printNGE(arr, n);
    getchar();
    return 0;
}
```

Output:

```
11 --> 13
13 --> 21
21 --> -1
3 --> -1
```

Time Complexity: O(n^2). The worst case occurs when all elements are sorted in decreasing order.

Method 2 (Using Stack)
Thanks to pchild for suggesting following approach.
1) Push the first element to stack.
2) Pick rest of the elements one by one and follow following steps in loop.
....a) Mark the current element as *next*.
....b) If stack is not empty, then pop an element from stack and compare it with *next*.
....c) If next is greater than the popped element, then *next* is the next greater element fot the popped element.
....d) Keep poppoing from the stack while the popped element is smaller than *next*. *next* becomes the next greater element for all such popped elements
....g) If *next* is smaller than the popped element, then push the popped element back.
3) After the loop in step 2 is over, pop all the elements from stack and print -1 as next element for them.

?

```
#include<stdio.h>
#include<stdlib.h>
#define STACKSIZE 100

// stack structure
struct stack
{
    int top;
    int items[STACKSIZE];
};
```

```
// Stack Functions to be used by printNGE()
void push(struct stack *ps, int x)
{
    if (ps->top == STACKSIZE-1)
    {
        printf("Error: stack overflow\n");
        getchar();
        exit(0);
    }
    else
    {
        ps->top += 1;
        ps->items[ps->top] = x;
    }
}

bool isEmpty(struct stack *ps)
{
    return (ps->top == -1)? true : false;
}

int pop(struct stack *ps)
{
    int temp;
    if (ps->top == -1)
    {
        printf("Error: stack underflow \n");
        getchar();
        exit(0);
    }
    else
    {
        temp = ps->items[ps->top];
        ps->top -= 1;
        return temp;
    }
}

/* prints element and NGE pair for all elements of
arr[] of size n */
void printNGE(int arr[], int n)
{
    int i = 0;
    struct stack s;
    s.top = -1;
    int element, next;

    /* push the first element to stack */
    push(&s, arr[0]);

    // iterate for rest of the elements
    for (i=1; i<n; i++)
    {
        next = arr[i];
```

```
        if (isEmpty(&s) == false)
        {
            // if stack is not empty, then pop an element from stack
            element = pop(&s);

            /* If the popped element is smaller than next, then
                a) print the pair
                b) keep popping while elements are smaller and
                stack is not empty */
            while (element < next)
            {
                printf("\n %d --> %d", element, next);
                if(isEmpty(&s) == true)
                   break;
                element = pop(&s);
            }

            /* If element is greater than next, then push
               the element back */
            if (element > next)
                push(&s, element);
        }

        /* push next to stack so that we can find
           next greater for it */
        push(&s, next);
    }

    /* After iterating over the loop, the remaining
       elements in stack do not have the next greater
       element, so print -1 for them */
    while(isEmpty(&s) == false)
    {
        element = pop(&s);
        next = -1;
        printf("\n %d --> %d", element, next);
    }
}

/* Driver program to test above functions */
int main()
{
    int arr[]= {11, 13, 21, 3};
    int n = sizeof(arr)/sizeof(arr[0]);
    printNGE(arr, n);
    getchar();
    return 0;
}
```

Output:

```
 11 --> 13
 13 --> 21
 3 --> -1
 21 --> -1
```

Time Complexity: O(n). The worst case occurs when all elements are sorted in decreasing order. If elements are sorted in decreasing order, then every element is processed at most 4 times.
a) Initialy pushed to the stack.
b) Popped from the stack when next element is being processed.
c) Pushed back to the stack because next element is smaller.
d) Popped from the stack in step 3 of algo.

Source:
http://geeksforgeeks.org/forum/topic/next-greater-element#post-60

CHECK IF ARRAY ELEMENTS ARE CONSECUTIVE

Given an unsorted array of numbers, write a function that returns true if array consists of consecutive numbers.

Examples:
a) If array is {5, 2, 3, 1, 4}, then the function should return true because the array has consecutive numbers from 1 to 5.

b) If array is {83, 78, 80, 81, 79, 82}, then the function should return true because the array has consecutive numbers from 78 to 83.

c) If the array is {34, 23, 52, 12, 3 }, then the function should return false because the elements are not consecutive.

d) If the array is {7, 6, 5, 5, 3, 4}, then the function should return false because 5 and 5 are not consecutive.

Method 1 (Use Sorting)
1) Sort all the elements.
2) Do a linear scan of the sorted array. If the difference between current element and next element is anything other than 1, then return false. If all differences are 1, then return true.

Time Complexity: O(nLogn)

Method 2 (Use visited array)
The idea is to check for following two conditions. If following two conditions are true, then return true.
1) $max - min + 1 = n$ where max is the maximum element in array, min is minimum element in array and n is the number of elements in array.
2) All elements are distinct.

To check if all elements are distinct, we can create a visited[] array of size n. We can map the ith element of input array arr[] to visited array by using arr[i] – min as index in visited[].

?

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

/* Helper functions to get minimum and maximum in an array */
int getMin(int arr[], int n);
int getMax(int arr[], int n);

/* The function checks if the array elements are consecutive
  If elements are consecutive, then returns true, else returns
  false */
bool areConsecutive(int arr[], int n)
{
  if ( n <  1 )
    return false;

  /* 1) Get the minimum element in array */
  int min = getMin(arr, n);

  /* 2) Get the maximum element in array */
  int max = getMax(arr, n);

  /* 3) max - min + 1 is equal to n,  then only check all elements */
  if (max - min  + 1 == n)
  {
      /* Create a temp array to hold visited flag of all elements.
         Note that, calloc is used here so that all values are initialized
         as false */
      bool *visited = (bool *)calloc(sizeof(bool), n);
      int i;
      for (i = 0; i < n; i++)
      {
         /* If we see an element again, then return false */
         if ( visited[arr[i] - min] != false )
           return false;

         /* If visited first time, then mark the element as visited */
         visited[arr[i] - min] = true;
      }

      /* If all elements occur once, then return true */
      return true;
  }

  return false; // if (max - min  + 1 != n)
}

/* UTILITY FUNCTIONS */
int getMin(int arr[], int n)
{
  int min = arr[0];
  for (int i = 1; i < n; i++)
   if (arr[i] < min)
     min = arr[i];
  return min;
}
```

```
int getMax(int arr[], int n)
{
  int max = arr[0];
  for (int i = 1; i < n; i++)
   if (arr[i] > max)
     max = arr[i];
  return max;
}

/* Driver program to test above functions */
int main()
{
    int arr[]= {5, 4, 2, 3, 1, 6};
    int n = sizeof(arr)/sizeof(arr[0]);
    if(areConsecutive(arr, n) == true)
        printf(" Array elements are consecutive ");
    else
        printf(" Array elements are not consecutive ");
    getchar();
    return 0;
}
```

Time Complexity: O(n)
Extra Space: O(n)

Method 3 (Mark visited array elements as negative)
This method is O(n) time complexity and O(1) extra space, but it changes the original array and it works only if all numbers are positive. We can get the original array by adding an extra step though. It is an extension of method 2 and it has the same two steps.
1) *max – min + 1 = n* where max is the maximum element in array, min is minimum element in array and n is the number of elements in array.
2) All elements are distinct.

In this method, the implementation of step 2 differs from method 2. Instead of creating a new array, we modify the input array arr[] to keep track of visited elements. The idea is to traverse the array and for each index i (where 0 <= i < n), make arr[arr[i] - min]] as a negative value. If we see a negative value again then there is repetition.

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

/* Helper functions to get minimum and maximum in an array */
int getMin(int arr[], int n);
int getMax(int arr[], int n);

/* The function checks if the array elements are consecutive
  If elements are consecutive, then returns true, else returns
  false */
bool areConsecutive(int arr[], int n)
{
```

```
    if ( n <  1 )
        return false;

    /* 1) Get the minimum element in array */
    int min = getMin(arr, n);

    /* 2) Get the maximum element in array */
    int max = getMax(arr, n);

    /* 3) max - min + 1 is equal to n then only check all elements */
    if (max - min  + 1 == n)
    {
        int i;
        for(i = 0; i < n; i++)
        {
            int j;

            if (arr[i] < 0)
                j = -arr[i] - min;
            else
                j = arr[i] - min;

            // if the value at index j is negative then
            // there is repitition
            if (arr[j] > 0)
                arr[j] = -arr[j];
            else
                return false;
        }

        /* If we do not see a negative value then all elements
           are distinct */
        return true;
    }

    return false; // if (max - min  + 1 != n)
}

/* UTILITY FUNCTIONS */
int getMin(int arr[], int n)
{
    int min = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] < min)
            min = arr[i];
    return min;
}

int getMax(int arr[], int n)
{
    int max = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > max)
            max = arr[i];
```

```
    return max;
}

/* Driver program to test above functions */
int main()
{
    int arr[]= {1, 4, 5, 3, 2, 6};
    int n = sizeof(arr)/sizeof(arr[0]);
    if(areConsecutive(arr, n) == true)
        printf(" Array elements are consecutive ");
    else
        printf(" Array elements are not consecutive ");
    getchar();
    return 0;
}
```

Note that this method might not work for negative numbers. For example, it returns false for {2, 1, 0, -3, -1, -2}.

Time Complexity: O(n)
Extra Space: O(1)

FIND THE FIRST MISSING NUMBER

Given a **sorted** array of n integers where each integer is in the range from 0 to m-1 and m > n. Find the smallest number that is missing from the array.

Examples
Input: {0, 1, 2, 6, 9}, n = 5, m = 10
Output: 3

Input: {4, 5, 10, 11}, n = 4, m = 12
Output: 0

Input: {0, 1, 2, 3}, n = 4, m = 5
Output: 4

Input: {0, 1, 2, 3, 4, 5, 6, 7, 10}, n = 9, m = 11
Output: 8

Method 1 (Use Binary Search)
For i = 0 to m-1, do binary search for i in the array. If i is not present in the array then return i.

Time Complexity: O(m log n)

Method 2 (Linear Search)
Traverse the input array and for each pair of elements a[i] and a[i+1], find the difference between them. if the difference is greater than 1 then a[i]+1 is the missing number.

Time Complexity: O(n)

Method 3 (Use Modified Binary Search)

In the standard Binary Search process, the element to be searched is compared with the middle element and on the basis of comparison result, we decide whether to search is over or to go to left half or right half.
In this method, we modify the standard Binary Search algorithm to compare the middle element with its index and make decision on the basis of this comparison.

…1) If the first element is not same as its index then return first index
…2) Else get the middle index say mid
……a) If arr[mid] greater than mid then the required element lies in left half.
……b) Else the required element lies in right half.

?

```
int findFirstMissing(int array[], int start, int end) {

    if(start   > end)
      return end + 1;

    if (start != array[start])
      return start;

    int mid = (start + end) / 2;

    if (array[mid] > mid)
      return findFirstMissing(array, start, mid);
    else
      return findFirstMissing(array, mid + 1, end);
}

// driver program to test above function
int main()
{
  int arr[] = {0, 1, 2, 3, 4, 5, 6, 7, 10};
  int n = sizeof(arr)/sizeof(arr[0]);
  printf(" First missing element is %d",
           findFirstMissing(arr, 0, n-1));
  getchar();
  return 0;
```

Note: This method doesn't work if there are duplicate elements in the array.

Time Complexity: O(Logn)

COUNT THE NUMBER OF OCCURRENCES IN A SORTED ARRAY

Given a sorted array arr[] and a number x, write a function that counts the occurrences of x in arr[]. Expected time complexity is O(Logn)

Examples:

```
  Input: arr[] = {1, 1, 2, 2, 2, 2, 3,},   x = 2
  Output: 4 // x (or 2) occurs 4 times in arr[]

  Input: arr[] = {1, 1, 2, 2, 2, 2, 3,},   x = 3
  Output: 1

  Input: arr[] = {1, 1, 2, 2, 2, 2, 3,},   x = 1
  Output: 2

  Input: arr[] = {1, 1, 2, 2, 2, 2, 3,},   x = 4
  Output: -1 // 4 doesn't occur in arr[]
```

Method 1 (Linear Search)
Linearly search for x, count the occurrences of x and return the count.

Time Complexity: O(n)

Method 2 (Use Binary Search)
1) Use Binary search to get index of the first occurrence of x in arr[]. Let the index of the first occurrence be i.
2) Use Binary search to get index of the last occurrence of x in arr[]. Let the index of the last occurrence be j.
3) Return (j – i + 1);

```
/* if x is present in arr[] then returns the count of occurrences of x,
   otherwise returns -1. */
int count(int arr[], int x, int n)
{
  int i; // index of first occurrence of x in arr[0..n-1]
  int j; // index of last occurrence of x in arr[0..n-1]

  /* get the index of first occurrence of x */
  i = first(arr, 0, n-1, x, n);

  /* If x doesn't exist in arr[] then return -1 */
  if(i == -1)
    return i;

  /* Else get the index of last occurrence of x. Note that we
      are only looking in the subarray after first occurrence */
  j = last(arr, i, n-1, x, n);

  /* return count */
  return j-i+1;
}

/* if x is present in arr[] then returns the index of FIRST occurrence
```

```
   of x in arr[0..n-1], otherwise returns -1 */
int first(int arr[], int low, int high, int x, int n)
{
  if(high >= low)
  {
    int mid = (low + high)/2;  /*low + (high - low)/2;*/
    if( ( mid == 0 || x > arr[mid-1]) && arr[mid] == x)
      return mid;
    else if(x > arr[mid])
      return first(arr, (mid + 1), high, x, n);
    else
      return first(arr, low, (mid -1), x, n);
  }
  return -1;
}

/* if x is present in arr[] then returns the index of LAST occurrence
   of x in arr[0..n-1], otherwise returns -1 */
int last(int arr[], int low, int high, int x, int n)
{
  if(high >= low)
  {
    int mid = (low + high)/2;  /*low + (high - low)/2;*/
    if( ( mid == n-1 || x < arr[mid+1]) && arr[mid] == x )
      return mid;
    else if(x < arr[mid])
      return last(arr, low, (mid -1), x, n);
    else
      return last(arr, (mid + 1), high, x, n);
  }
  return -1;
}

/* driver program to test above functions */
int main()
{
  int arr[] = {1, 2, 2, 3, 3, 3, 3};
  int x =  3;  // Element to be counted in arr[]
  int n = sizeof(arr)/sizeof(arr[0]);
  int c = count(arr, x, n);
  printf(" %d occurs %d times ", x, c);
  getchar();
  return 0;
}
```

Time Complexity: O(Logn)
Programming Paradigm: Divide & Conquer

GIVEN AN ARRAY ARR[], FIND THE MAXIMUM J – I SUCH THAT ARR[J] > ARR[I]

Given an array arr[], find the maximum j – i such that arr[j] > arr[i].

Examples:

```
  Input: {34, 8, 10, 3, 2, 80, 30, 33, 1}
  Output: 6  (j = 7, i = 1)

  Input: {9, 2, 3, 4, 5, 6, 7, 8, 18, 0}
  Output: 8 ( j = 8, i = 0)

  Input:  {1, 2, 3, 4, 5, 6}
  Output: 5  (j = 5, i = 0)

  Input:  {6, 5, 4, 3, 2, 1}
  Output: -1
```

Method 1 (Simple but Inefficient)
Run two loops. In the outer loop, pick elements one by one from left. In the inner loop, compare the picked element with the elements starting from right side. Stop the inner loop when you see an element greater than the picked element and keep updating the maximum j-i so far.

```
#include <stdio.h>
/* For a given array arr[], returns the maximum j – i such that
    arr[j] > arr[i] */
int maxIndexDiff(int arr[], int n)
{
    int maxDiff = -1;
    int i, j;

    for (i = 0; i < n; ++i)
    {
        for (j = n-1; j > i; --j)
        {
            if(arr[j] > arr[i] && maxDiff < (j - i))
                maxDiff = j - i;
        }
    }

    return maxDiff;
}

int main()
{
    int arr[] = {9, 2, 3, 4, 5, 6, 7, 8, 18, 0};
    int n = sizeof(arr)/sizeof(arr[0]);
    int maxDiff = maxIndexDiff(arr, n);
    printf("\n %d", maxDiff);
    getchar();
    return 0;
}
```

Time Complexity: O(n^2)

Method 2 (Efficient)
To solve this problem, we need to get two optimum indexes of arr[]: left index i and right index j. For an element arr[i], we do not need to consider arr[i] for left index if there is an element smaller than arr[i] on left side of arr[i]. Similarly, if there is a greater element on right side of

arr[j] then we do not need to consider this j for right index. So we construct two auxiliary arrays LMin[] and RMax[] such that LMin[i] holds the smallest element on left side of arr[i] including arr[i], and RMax[j] holds the greatest element on right side of arr[j] including arr[j]. After constructing these two auxiliary arrays, we traverse both of these arrays from left to right. While traversing LMin[] and RMa[] if we see that LMin[i] is greater than RMax[j], then we must move ahead in LMin[] (or do i++) because all elements on left of LMin[i] are greater than or equal to LMin[i]. Otherwise we must move ahead in RMax[j] to look for a greater j – i value.

```
#include <stdio.h>

/* Utility Functions to get max and minimum of two integers */
int max(int x, int y)
{
    return x > y? x : y;
}

int min(int x, int y)
{
    return x < y? x : y;
}

/* For a given array arr[], returns the maximum j – i such that
    arr[j] > arr[i] */
int maxIndexDiff(int arr[], int n)
{
    int maxDiff;
    int i, j;

    int *LMin = (int *)malloc(sizeof(int)*n);
    int *RMax = (int *)malloc(sizeof(int)*n);

   /* Construct LMin[] such that LMin[i] stores the minimum value
       from (arr[0], arr[1], ... arr[i]) */
    LMin[0] = arr[0];
    for (i = 1; i < n; ++i)
        LMin[i] = min(arr[i], LMin[i-1]);

    /* Construct RMax[] such that RMax[j] stores the maximum value
       from (arr[j], arr[j+1], ..arr[n-1]) */
    RMax[n-1] = arr[n-1];
    for (j = n-2; j >= 0; --j)
        RMax[j] = max(arr[j], RMax[j+1]);

    /* Traverse both arrays from left to right to find optimum j - i
        This process is similar to merge() of MergeSort */
    i = 0, j = 0, maxDiff = -1;
    while (j < n && i < n)
    {
        if (LMin[i] < RMax[j])
        {
            maxDiff = max(maxDiff, j-i);
            j = j + 1;
        }
```

```
        else
            i = i+1;
    }

    return maxDiff;
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {9, 2, 3, 4, 5, 6, 7, 8, 18, 0};
    int n = sizeof(arr)/sizeof(arr[0]);
    int maxDiff = maxIndexDiff(arr, n);
    printf("\n %d", maxDiff);
    getchar();
    return 0;
}
```

Time Complexity: O(n)
Auxiliary Space: O(n)

MAXIMUM OF ALL SUBARRAYS OF SIZE K

Given an array and an integer k, find the maximum for each and every contiguous subarray of size k.

Examples:

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

Input :
arr[] = {8, 5, 10, 7, 9, 4, 15, 12, 90, 13}
k = 4
Output :
10 10 10 15 15 90 90

Method 1 (Simple)
Run two loops. In the outer loop, take all subarrays of size k. In the inner loop, get the maximum of the current subarray.

?

```
#include<stdio.h>

void printKMax(int arr[], int n, int k)
{
```

```
    int j, max;

    for (int i = 0; i <= n-k; i++)
    {
        max = arr[i];

        for (j = 1; j < k; j++)
        {
            if (arr[i+j] > max)
               max = arr[i+j];
        }
        printf("%d ", max);
    }
}

int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int n = sizeof(arr)/sizeof(arr[0]);
    int k = 3;
    printKMax(arr, n, k);
    return 0;
}
```

Time Complexity: The outer loop runs n-k+1 times and the inner loop runs k times for every iteration of outer loop. So time complexity is O((n-k+1)*k) which can also be written as O(nk).

Method 2 (Use Self-Balancing BST)
1) Pick first k elements and create a Self-Balancing Binary Search Tree (BST) of size k.
2) Run a loop for i = 0 to n – k
…..a) Get the maximum element from the BST, and print it.
…..b) Search for arr[i] in the BST and delete it from the BST.
…..c) Insert arr[i+k] into the BST.

Time Complexity: Time Complexity of step 1 is O(kLogk). Time Complexity of steps 2(a), 2(b) and 2(c) is O(Logk). Since steps 2(a), 2(b) and 2(c) are in a loop that runs n-k+1 times, time complexity of the complete algorithm is O(kLogk + (n-k+1)*Logk) which can also be written as O(nLogk)

FIND THE MINIMUM DISTANCE BETWEEN TWO NUMBERS

Given an unsorted array arr[] and two numbers x and y, find the minimum distance between x and y in arr[]. The array might also contain duplicates. You may assume that both x and y are different and present in arr[].

Examples:
Input: arr[] = {1, 2}, x = 1, y = 2
Output: Minimum distance between 1 and 2 is 1.

Input: arr[] = {3, 4, 5}, x = 3, y = 5
Output: Minimum distance between 3 and 5 is 2.

Input: arr[] = {3, 5, 4, 2, 6, 5, 6, 6, 5, 4, 8, 3}, x = 3, y = 6
Output: Minimum distance between 3 and 6 is 4.

Input: arr[] = {2, 5, 3, 5, 4, 4, 2, 3}, x = 3, y = 2
Output: Minimum distance between 3 and 2 is 1.

Method 1 (Simple)
Use two loops: The outer loop picks all the elements of arr[] one by one. The inner loop picks all the elements after the element picked by outer loop. If the elements picked by outer and inner loops have same values as x or y then if needed update the minimum distance calculated so far.

?
```
#include <stdio.h>
#include <stdlib.h> // for abs()
#include <limits.h> // for INT_MAX

int minDist(int arr[], int n, int x, int y)
{
   int i, j;
   int min_dist = INT_MAX;
   for (i = 0; i < n; i++)
   {
     for (j = i+1; j < n; j++)
     {
         if( (x == arr[i] && y == arr[j] ||
              y == arr[i] && x == arr[j]) && min_dist > abs(i-j))
         {
              min_dist = abs(i-j);
         }
     }
   }
   return min_dist;
}

/* Driver program to test above fnction */
int main()
{
    int arr[] = {3, 5, 4, 2, 6, 5, 6, 6, 5, 4, 8, 3};
    int n = sizeof(arr)/sizeof(arr[0]);
    int x = 3;
    int y = 6;

    printf("Minimum distance between %d and %d is %d\n", x, y,
              minDist(arr, n, x, y));
    return 0;
}
```

Output: *Minimum distance between 3 and 6 is 4*

Time Complexity: O(n^2)

Method 2 (Tricky)
1) Traverse array from left side and stop if either *x* or *y* are found. Store index of this first occurrrence in a variable say *prev*
2) Now traverse *arr[]* after the index *prev*. If the element at current index *i* matches with either x or y then check if it is different from *arr[prev]*. If it is different then update the minimum distance if needed. If it is same then update *prev* i.e., make *prev = i*.

Thanks to wgpshashank for suggesting this approach.

?

```
#include <stdio.h>
#include <limits.h>  // For INT_MAX

int minDist(int arr[], int n, int x, int y)
{
   int i = 0;
   int min_dist = INT_MAX;
   int prev;

   // Find the first occurence of any of the two numbers (x or y)
   // and store the index of this occurence in prev
   for (i = 0; i < n; i++)
   {
     if (arr[i] == x || arr[i] == y)
     {
       prev = i;
       break;
     }
   }

   // Traverse after the first occurence
   for ( ; i < n; i++)
   {
      if (arr[i] == x || arr[i] == y)
      {
          // If the current element matches with any of the two then
          // check if current element and prev element are different
          // Also check if this value is smaller than minimm distance so far
          if ( arr[prev] != arr[i] && (i - prev) < min_dist )
          {
             min_dist = i - prev;
             prev = i;
          }
          else
             prev = i;
      }
   }

   return min_dist;
}
```

```
/* Driver program to test above fnction */
int main()
{
    int arr[] ={3, 5, 4, 2, 6, 3, 0, 0, 5, 4, 8, 3};
    int n = sizeof(arr)/sizeof(arr[0]);
    int x = 3;
    int y = 6;

    printf("Minimum distance between %d and %d is %d\n", x, y,
              minDist(arr, n, x, y));
    return 0;
}
```

Output: *Minimum distance between 3 and 6 is 4*

Time Complexity: O(n)

PRINT A GIVEN MATRIX IN SPIRAL FORM

Given a 2D array, print it in spiral form. See the following examples.

```
Input:
        1    2   3   4
        5    6   7   8
        9   10  11  12
        13  14  15  16
Output:
1 2 3 4 8 12 16 15 14 13 9 5 6 7 11 10

Input:
        1   2   3   4  5   6
        7   8   9  10  11  12
        13  14  15 16  17  18
Output:
1 2 3 4 5 6 12 18 17 16 15 14 13 7 8 9 10 11
```

Solution:

```
/* This code is adopted from the solution given
   @ http://effprog.blogspot.com/2011/01/spiral-printing-of-two-
dimensional.html */

#include <stdio.h>
#define R 3
#define C 6

void spiralPrint(int m, int n, int a[R][C])
{
    int i, k = 0, l = 0;

    /*  k - starting row index
        m - ending row index
```

```
        l - starting column index
        n - ending column index
        i - iterator
    */

    while (k < m && l < n)
    {
        /* Print the first row from the remaining rows */
        for (i = l; i < n; ++i) {
            printf("%d ", a[k][i]);
        }
        k++;

        /* Print the last column from the remaining columns */
        for (i = k; i < m; ++i) {
            printf("%d ", a[i][n-1]);
        }
        n--;

        /* Print the last row from the remaining rows */
        if ( k < m){
            for (i = n-1; i >= l; --i) {
                printf("%d ", a[m-1][i]);
            }
            m--;
        }

       /* Print the first column from the remaining columns */
        for (i = m-1; i >= k; --i) {
            printf("%d ", a[i][l]);
        }

        l++;
    }
}

/* Driver program to test above functions */
int main()
{
    int a[R][C] = { {1,  2,  3,  4,  5,  6},
        {7,  8,  9,  10, 11, 12},
        {13, 14, 15, 16, 17, 18}
    };

    spiralPrint(R, C, a);
    return 0;
}

/* OUTPUT:
  1 2 3 4 5 6 12 18 17 16 15 14 13 7 8 9 10 11
*/
```

Time Complexity: Time complexity of the above solution is O(mn).

FIND THE REPEATING AND THE MISSING

Given an unsorted array of size n. Array elements are in range from 1 to n. One number from set {1, 2, …n} is missing and one number occurs twice in array. Find these two numbers.

Examples:

```
arr[] = {3, 1, 3}
Output: 2, 3   // 2 is missing and 3 occurs twice

arr[] = {4, 3, 6, 2, 1, 1}
Output: 1, 5  // 5 is missing and 1 occurs twice
```

Method 1 (Use Sorting)
1) Sort the input array.
2) Traverse the array and check for missing and repeating.

Time Complexity: O(nLogn)

Method 2 (Use count array)
1) Create a temp array temp[] of size n with all initial values as 0.
2) Traverse the input array arr[], and do following for each arr[i]
……a) if(temp[arr[i]] == 0) temp[arr[i]] = 1;
……b) if(temp[arr[i]] == 1) output "arr[i]" //repeating
3) Traverse temp[] and output the array element having value as 0 (This is the missing element)

Time Complexity: O(n)
Auxiliary Space: O(n)

Method 3 (Use elements as Index and mark the visited places)
Traverse the array. While traversing, use absolute value of every element as index and make the value at this index as negative to mark it visited. If something is already marked negative then this is the repeating element. To find missing, traverse the array again and look for a positive value.

?

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

void printTwoElements(int arr[], int size)
{
    int i;
    printf("\n The repeating element is");

    for(i = 0; i < size; i++)
    {
        if(arr[abs(arr[i])-1] > 0)
            arr[abs(arr[i])-1] = -arr[abs(arr[i])-1];
        else
```

```
            printf(" %d ", abs(arr[i]));
    }

    printf("\nand the missing element is ");
    for(i=0; i<size; i++)
    {
        if(arr[i]>0)
            printf("%d",i+1);
    }
}

/* Driver program to test above function */
int main()
{
    int arr[] = {7, 3, 4, 5, 5, 6, 2};
    int n = sizeof(arr)/sizeof(arr[0]);
    printTwoElements(arr, n);
    return 0;
}
```

Time Complexity: O(n)

Method 4 (Make two equations)
Let x be the missing and y be the repeating element.

1) Get sum of all numbers.
Sum of array computed S = n(n+1)/2 – x + y
2) Get product of all numbers.
Product of array computed P = 1*2*3*…*n * y / x
3) The above two steps give us two equations, we can solve the equations and get the values of x and y.

Time Complexity: O(n)

This method can cause arithmetic overflow as we calculate product and sum of all array elements.

Method 5 (Use XOR)
Let x and y be the desired output elements.
Calculate XOR of all the array elements.

```
xor1 = arr[0]^arr[1]^arr[2].....arr[n-1]
```

XOR the result with all numbers from 1 to n

```
xor1 = xor1^1^2^.....^n
```

In the result *xor1*, all elements would nullify each other except x and y. All the bits that are set in *xor1* will be set in either x or y. So if we take any set bit (We have chosen the rightmost set bit in code) of *xor1* and divide the elements of the array in two sets – one set of elements with same bit

set and other set with same bit not set. By doing so, we will get x in one set and y in another set. Now if we do XOR of all the elements in first set, we will get x, and by doing same in other set we will get y.

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

/* The output of this function is stored at *x and *y */
void getTwoElements(int arr[], int n, int *x, int *y)
{
  int xor1;   /* Will hold xor of all elements and numbers from 1 to n */
  int set_bit_no;  /* Will have only single set bit of xor1 */
  int i;
  *x = 0;
  *y = 0;

  xor1 = arr[0];

  /* Get the xor of all array elements elements */
  for(i = 1; i < n; i++)
     xor1 = xor1^arr[i];

  /* XOR the previous result with numbers from 1 to n*/
  for(i = 1; i <= n; i++)
     xor1 = xor1^i;

  /* Get the rightmost set bit in set_bit_no */
  set_bit_no = xor1 & ~(xor1-1);

  /* Now divide elements in two sets by comparing rightmost set
   bit of xor1 with bit at same position in each element. Also, get XORs
   of two sets. The two XORs are the output elements.
   The following two for loops serve the purpose */
  for(i = 0; i < n; i++)
  {
    if(arr[i] & set_bit_no)
     *x = *x ^ arr[i]; /* arr[i] belongs to first set */
    else
     *y = *y ^ arr[i]; /* arr[i] belongs to second set*/
  }
  for(i = 1; i <= n; i++)
  {
    if(i & set_bit_no)
     *x = *x ^ i; /* i belongs to first set */
    else
     *y = *y ^ i; /* i belongs to second set*/
  }

  /* Now *x and *y hold the desired output elements */
}

/* Driver program to test above function */
int main()
{
```

```
  int arr[] = {1, 3, 4, 5, 5, 6, 2};
  int *x = (int *)malloc(sizeof(int));
  int *y = (int *)malloc(sizeof(int));
  int n = sizeof(arr)/sizeof(arr[0]);
  getTwoElements(arr, n, x, y);
  printf(" The two elements are %d and %d", *x, *y);
  getchar();
}
```

Time Complexity: O(n)

This method doesn't cause overflow, but it doesn't tell which one occurs twice and which one is missing.

RETURN MAXIMUM OCCURRING CHARACTER IN THE INPUT STRING

Write an efficient C function to return maximum occurring character in the input string e.g., if input string is “test string” then function should return ‘t’.

Algorithm:

```
Input string = “test”
1: Construct character count array from the input string.
  count['e'] = 1
  count['s'] = 1
  count['t'] = 2

2: Return the index of maximum value in count array (returns ‘t’).
```

Implementation:

```
#include <stdio.h>
#include <stdlib.h>
#define NO_OF_CHARS 256

int *getCharCountArray(char *);
char getIndexOfMax(int *, int);

/* Returns the maximum occurring character in
 the input string */
char getMaxOccuringChar(char *str)
{
  int *count = getCharCountArray(str);
  return getIndexOfMax(count, NO_OF_CHARS);
}

/* Returns an array of size 256 containg count
   of characters in the passed char array */
int *getCharCountArray(char *str)
{
  int *count = (int *)calloc(sizeof(int), NO_OF_CHARS);
  int i;

  for (i = 0; *(str+i);  i++)
    count[*(str+i)]++;

  return count;
}

char getIndexOfMax(int ar[], int ar_size)
{
  int i;
  int max_index = 0;
```

```
  for(i = 1; i < ar_size; i++)
    if(ar[i] > ar[max_index])
      max_index = i;

  /* free memory allocated to count */
  free(ar);
  ar = NULL;

  return max_index;
}

int main()
{
  char str[] = "sample string";
  printf("%c", getMaxOccuringChar(str));

  getchar();
  return 0;
}
```

Time Complexity: O(n)

Notes:
If more than one character have the same and maximum count then function returns only the first one. For example if input string is “test sample” then function will return only ‘t’.

REMOVE ALL DUPLICATES FROM THE INPUT STRING.

Below are the different methods to remove duplicates in a string.

METHOD 1 (Use Sorting)

Algorithm:

```
  1) Sort the elements.
  2) Now in a loop, remove duplicates by comparing the
      current character with previous character.
  3)  Remove extra characters at the end of the resultant string.
```

Example:

```
Input string:  geeksforgeeks
1) Sort the characters
   eeeefggkkosss
2) Remove duplicates
    efgkosgkkosss
3) Remove extra characters
     efgkos
```

Note that, this method doesn't keep the original order of the input string. For example, if we are to remove duplicates for geeksforgeeks and keep the order of characters same, then output should be geksfor, but above function returns efgkos. We can modify this method by storing the original order. METHOD 2 keeps the order same.

Implementation:

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

/* Function to remove duplicates in a sorted array */
char *removeDupsSorted(char *str);

/* Utitlity function to sort array A[] */
void quickSort(char A[], int si, int ei);

/* Function removes duplicate characters from the string
   This function work in-place and fills null characters
   in the extra space left */
char *removeDups(char *str)
{
  int len = strlen(str);
  quickSort(str, 0, len-1);
  return removeDupsSorted(str);
}

/* Function to remove duplicates in a sorted array */
char *removeDupsSorted(char *str)
{
  int res_ind = 1, ip_ind = 1;

  /* In place removal of duplicate characters*/
  while(*(str + ip_ind))
  {
    if(*(str + ip_ind) != *(str + ip_ind - 1))
    {
      *(str + res_ind) = *(str + ip_ind);
      res_ind++;
    }
    ip_ind++;
  }

  /* After above step string is stringiittg.
     Removing extra iittg after string*/
  *(str + res_ind) = '\0';

  return str;
}

/* Driver program to test removeDups */
int main()
{
```

```
  char str[] = "eeeefggkkosss";
  printf("%s", removeDups(str));
  getchar();
  return 0;
}

/* FOLLOWING FUNCTIONS ARE ONLY FOR SORTING
    PURPOSE */
void exchange(char *a, char *b)
{
  char temp;
  temp = *a;
  *a   = *b;
  *b   = temp;
}

int partition(char A[], int si, int ei)
{
  char x = A[ei];
  int i = (si - 1);
  int j;

  for (j = si; j <= ei - 1; j++)
  {
    if(A[j] <= x)
    {
      i++;
      exchange(&A[i], &A[j]);
    }
  }
  exchange (&A[i + 1], &A[ei]);
  return (i + 1);
}

/* Implementation of Quick Sort
A[] --> Array to be sorted
si  --> Starting index
ei  --> Ending index
*/
void quickSort(char A[], int si, int ei)
{
  int pi;    /* Partitioning index */
  if(si < ei)
  {
    pi = partition(A, si, ei);
    quickSort(A, si, pi - 1);
    quickSort(A, pi + 1, ei);
  }
}
```

Time Complexity: O(nlogn) If we use some nlogn sorting algorithm instead of quicksort.

METHOD 2 (Use Hashing)

Algorithm:

```
1: Initialize:
    str  =  "test string" /* input string */
    ip_ind =  0          /* index to  keep track of location of next
                            character in input string */
    res_ind  =  0         /* index to  keep track of location of
                             next character in the resultant string */
    bin_hash[0..255] = {0,0, ….} /* Binary hash to see if character is
                                    already processed or not */
2: Do following for each character *(str + ip_ind) in input string:
              (a) if bin_hash is not set for *(str + ip_ind) then
                   // if program sees the character *(str + ip_ind) first
time
                          (i)  Set bin_hash for *(str + ip_ind)
                          (ii)  Move *(str  + ip_ind) to the resultant string.
                               This is done in-place.
                          (iii) res_ind++
              (b) ip_ind++
  /* String obtained after this step is "te sringng" */
3: Remove extra characters at the end of the resultant string.
  /*  String obtained after this step is "te sring" */
```

Implementation:

```
?
# include <stdio.h>
# include <stdlib.h>
# define NO_OF_CHARS 256
# define bool int

/* Function removes duplicate characters from the string
   This function work in-place and fills null characters
   in the extra space left */
char *removeDups(char *str)
{
  bool bin_hash[NO_OF_CHARS] = {0};
  int ip_ind = 0, res_ind = 0;
  char temp;

  /* In place removal of duplicate characters*/
  while(*(str + ip_ind))
  {
    temp = *(str + ip_ind);
    if(bin_hash[temp] == 0)
    {
        bin_hash[temp] = 1;
        *(str + res_ind) = *(str + ip_ind);
        res_ind++;
    }
    ip_ind++;
  }

  /* After above step string is stringiittg.
     Removing extra iittg after string*/
```

```
  *(str+res_ind) = '\0';

  return str;
}

/* Driver program to test removeDups */
int main()
{
    char str[] = "geeksforgeeks";
    printf("%s", removeDups(str));
    getchar();
    return 0;
}
```

Time Complexity: O(n)

NOTES:
* It is assumed that number of possible characters in input string are 256. NO_OF_CHARS should be changed accordingly.
* calloc is used instead of malloc for memory allocations of counting array (count) to initialize allocated memory to '\0'. malloc() followed by memset() could also be used.
* Above algorithm also works for an integer array inputs if range of the integers in array is given. Example problem is to find maximum occurring number in an input array given that the input array contain integers only between 1000 to 1100

PRINT ALL THE DUPLICATES IN THE INPUT STRING.

Write an efficient C program to print all the duplicates and their counts in the input string

Algorithm: Let input string be "geeksforgeeks"
1: Construct character count array from the input string.

count['e'] = 4
count['g'] = 2
count['k'] = 2
……

2: Print all the indexes from the constructed array which have value greater than 0.

Solution

```
# include <stdio.h>
# include <stdlib.h>
# define NO_OF_CHARS 256

/* Returns an array of size 256 containg count
   of characters in the passed char array */
int *getCharCountArray(char *str)
{
```

```
    int *count = (int *)calloc(sizeof(int), NO_OF_CHARS);
    int i;

    for (i = 0; *(str+i);  i++)
       count[*(str+i)]++;

    return count;
}

/* Print duplicates present in the passed string */
void printDups(char *str)
{
  int *count = getCharCountArray(str);
  int i;
  char temp;

  for (i = 0; i < NO_OF_CHARS; i++)
    if(count[i] > 1)
        printf("%c,  count = %d \n", i,  count[i]);
}

/* Driver program to test to pront printDups*/
int main()
{
    char str[] = "test string";
    printDups(str);
    getchar();
    return 0;
}
```

Time Complexity: O(n)

REMOVE CHARACTERS FROM THE FIRST STRING WHICH ARE PRESENT IN THE SECOND STRING

Write an efficient C function that takes two strings as arguments and removes the characters from first string which are present in second string (mask string).

Algorithm: Let first input string be"test string" and the string which has characters to be removed from first string be "mask"
1: Initialize:
res_ind = 0 /* index to keep track of processing of each character in i/p string */
ip_ind = 0 /* index to keep track of processing of each character in the resultant string */

2: Construct count array from mask_str. Count array would be:
(We can use Boolean array here instead of int count array because we don't need count, we need to know only if character is present in mask string)
count['a'] = 1
count['k'] = 1

count['m'] = 1
count['s'] = 1

3: Process each character of the input string and if count of that character is 0 then only add the character to the resultant string.
str = “tet tringng” // ’s’ has been removed because ’s’ was present in mask_str but we we have got two extra characters “ng”
ip_ind = 11
res_ind = 9

4: Put a ‘\0′ at the end of the string?

Implementation:

```
#include <stdio.h>
#include <stdlib.h>
#define NO_OF_CHARS 256

/* Returns an array of size 256 containg count
   of characters in the passed char array */
int *getCharCountArray(char *str)
{
   int *count = (int *)calloc(sizeof(int), NO_OF_CHARS);
   int i;
   for (i = 0; *(str+i);  i++)
      count[*(str+i)]++;
   return count;
}

/* removeDirtyChars takes two string as arguments: First
string (str)  is the one from where function removes dirty
characters. Second  string is the string which contain all
dirty characters which need to be removed  from first string */
char *removeDirtyChars(char *str, char *mask_str)
{
  int *count  = getCharCountArray(mask_str);
  int ip_ind  = 0, res_ind = 0;
  char temp;
  while(*(str + ip_ind))
  {
    temp = *(str + ip_ind);
    if(count[temp] == 0)
    {
      *(str + res_ind) = *(str + ip_ind);
      res_ind++;
    }
    ip_ind++;
  }

  /* After above step string is ngring.
    Removing extra "iittg" after string*/
  *(str+res_ind) = '\0';
```

```
  return str;
}

/* Driver program to test getCharCountArray*/
int main()
{
    char mask_str[]  = "mask";
    char str[]       = "geeksforgeeks";
    printf("%s", removeDirtyChars(str, mask_str));
    getchar();
    return 0;
}
```

Time Complexity: O(m+n) Where m is the length of mask string and n is the length of the input string.

A PROGRAM TO CHECK IF STRINGS ARE ROTATIONS OF EACH OTHER OR NOT

Given a string s1 and a string s2, write a snippet to say whether s2 is a rotation of s1 using only one call to strstr routine?
(eg given s1 = ABCD and s2 = CDAB, return true, given s1 = ABCD, and s2 = ACBD , return false)

Algorithm: areRotations(str1, str2)

```
    1. Create a temp string and store concatenation of str1 to
       str1 in temp.
                          temp = str1.str1
    2. If str2 is a substring of temp then str1 and str2 are
       rotations of each other.

    Example:
                     str1 = "ABACD"
                     str2 = "CDABA"

     temp = str1.str1 = "ABACDABACD"
     Since str2 is a substring of temp, str1 and str2 are
     rotations of each other.
```

Implementation:

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

/* Function checks if passed strings (str1 and str2)
   are rotations of each other */
int areRotations(char *str1, char *str2)
{
  int size1   = strlen(str1);
  int size2   = strlen(str2);
```

```
  char *temp;
  void *ptr;

  /* Check if sizes of two strings are same */
  if(size1 != size2)
     return 0;

  /* Create a temp string with value str1.str1 */
  temp   = (char *)malloc(sizeof(char)*size1*2 + 1);
  temp[0] = '\0';
  strcat(temp, str1);
  strcat(temp, str1);

  /* Now check if str2 is a substring of temp */
  ptr = strstr(temp, str2);

  /* strstr returns NULL if the second string is NOT a
    substring of first string */
  if(ptr != NULL)
    return 1;
  else
    return 0;
}

/* Driver program to test areRotations */
int main()
{
    char *str1 = "ABCD";
    char *str2 = "ABCDA";

    if(areRotations(str1, str2))
       printf("Strings are rotations of each other");
    else
       printf("Strings are not rotations of each other");

    getchar();
    return 0;
}
```

Library Functions Used:
strstr:
strstr finds a sub-string within a string.
Prototype: char * strstr(const char *s1, const char *s2);
See
http://www.lix.polytechnique.fr/Labo/Leo.Liberti/public/computing/prog/c/C/MAN/strstr.htm
for more details

strcat:
strcat concatenate two strings
Prototype: char *strcat(char *dest, const char *src);
See

http://www.lix.polytechnique.fr/Labo/Leo.Liberti/public/computing/prog/c/C/MAN/strcat.htm for more details

Time Complexity: Time complexity of this problem depends on the implementation of strstr function.
If implementation of strstr is done using KMP matcher then complexity of the above program is (-)(n1 + n2) where n1 and n2 are lengths of strings. KMP matcher takes (-)(n) time to find a substrng in a string of length n where length of substring is assumed to be smaller than the string

PRINT REVERSE OF A STRING USING RECURSION

Write a recursive C function to print reverse of a given string.

Program:

```
# include <stdio.h>

/* Function to print reverse of the passed string */
void reverse(char *str)
{
   if(*str)
   {
       reverse(str+1);
       printf("%c", *str);
   }
}

/* Driver program to test above function */
int main()
{
   char a[] = "Geeks for Geeks";
   reverse(a);
   getchar();
   return 0;
}
```

Explanation: Recursive function (reverse) takes string pointer (str) as input and calls itself with next location to passed pointer (str+1). Recursion continues this way, when pointer reaches '\0', all functions accumulated in stack print char at passed location (str) and return one by one.

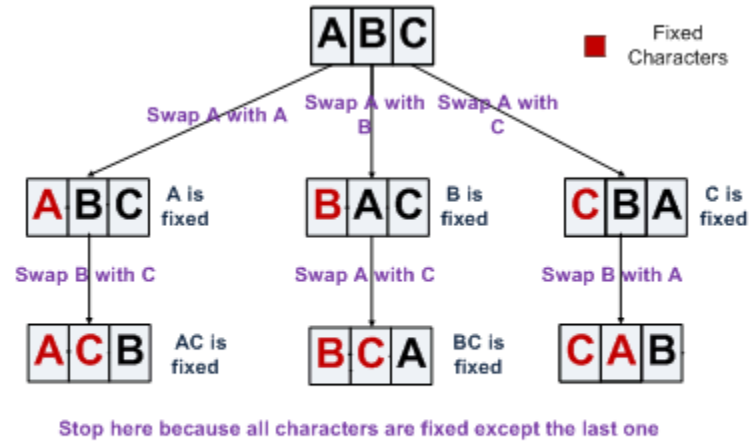
Time Complexity: O(n)

WRITE A C PROGRAM TO PRINT ALL PERMUTATIONS OF A GIVEN STRING

A permutation, also called an "arrangement number" or "order," is a rearrangement of the elements of an ordered list S into a one-to-one correspondence with S itself. A string of length n has n! permutation.
Source: Mathword(http://mathworld.wolfram.com/Permutation.html)

Below are the permutations of string ABC.
ABC, ACB, BAC, BCA, CAB, CBA

Here is a solution using backtracking.



Recursion Tree for Permutations of String "ABC"

```
# include <stdio.h>
# include <conio.h>

/* Function to swap values at two pointers */
void swap (char *x, char *y)
{
    char temp;
    temp = *x;
    *x = *y;
    *y = temp;
}

/* Function to print permutations of string
   This function takes three parameters:
   1. String
   2. Starting index of the string
   3. Ending index of the string. */
void permute(char *a, int i, int n)
{
   int j;
   if (i == n)
     printf("%s\n", a);
   else
   {
        for (j = i; j <= n; j++)
       {
          swap((a+i), (a+j));
          permute(a, i+1, n);
```

```
            swap((a+i), (a+j)); //backtrack
        }
    }
}

/* Driver program to test above functions */
int main()
{
   char a[] = "ABC";
   permute(a, 0, 2);
   getchar();
   return 0;
}
```

Algorithm Paradigm: Backtracking
Time Complexity: O(n!)

DIVIDE A STRING IN N EQUAL PARTS

Question:
Write a program to print N equal parts of a given string.

Solution:
1) Get the size of the string using string function strlen() (present in string.h)
2) Get size of a part.

```
part_size = string_length/n
```

3) Loop through the input string. In loop, if index becomes multiple of part_size then put a part separator("\n")

Implementation:

?

```
#include<stdio.h>
#include<string.h>

/* Function to print n equal parts of str*/
void divideString(char *str, int n)
{
   int str_size = strlen(str);
   int i;
   int part_size;

   /*Check if string can be divided in n equal parts */
   if(str_size%n != 0)
   {
     printf("Invalid Input: String size is not divisible by n");
     return;
   }
```

```
   /* Calculate the size of parts to find the division points*/
   part_size = str_size/n;
   for(i = 0; i< str_size; i++)
   {
     if(i%part_size == 0)
        printf("\n"); /* newline separator for different parts */
     printf("%c", str[i]);
   }
}

int main()
{
    /*length od string is 28*/
    char *str = "a_simple_divide_string_quest";

    /*Print 4 equal parts of the string */
    divideString(str, 4);

    getchar();
    return 0;
}
```

In above solution, we are simply printing the N equal parts of the string. If we want individual parts to be stored then we need to allocate part_size + 1 memory for all N parts (1 extra for string termination character '\0′), and store the addresses of the parts in an array of character pointers.

GIVEN A STRING, FIND ITS FIRST NON-REPEATING CHARACTER

Algorithm:

```
1) Scan the string from left to right and construct the count array.
2) Again, scan the string from left to right and check for count of each
 character, if you find an element who's count is 1, return it.
```

Example:

```
Input string: str = geeksforgeeks
1: Construct character count array from the input string.
   ....
  count['e'] = 4
  count['f'] = 1
  count['g'] = 2
  count['k'] = 2
  ……
2: Get the first character who's count is 1 ('f').
```

Implementation:

```
#include<stdlib.h>
#include<stdio.h>
#define NO_OF_CHARS 256
```

```
/* Returns an array of size 256 containg count
   of characters in the passed char array */
int *getCharCountArray(char *str)
{
   int *count = (int *)calloc(sizeof(int), NO_OF_CHARS);
   int i;
   for (i = 0; *(str+i);  i++)
      count[*(str+i)]++;
   return count;
}

/* The function returns index of first non-repeating
   character in a string. If all characters are repeating
   then reurns -1 */
int firstNonRepeating(char *str)
{
  int *count = getCharCountArray(str);
  int index = -1, i;

  for (i = 0; *(str+i);  i++)
  {
    if(count[*(str+i)] == 1)
    {
      index = i;
      break;
    }
  }
  return index;
}

/* Driver program to test above function */
int main()
{
  char str[] = "geeksforgeeks";
  int index =  firstNonRepeating(str);
  if(index == -1)
    printf("Either all characters are repeating or string is empty");
  else
   printf("First non-repeating character is %c", str[index]);
  getchar();
  return 0;
}
```

Time Complexity: O(n)

PRINT LIST ITEMS CONTAINING ALL CHARACTERS OF A GIVEN WORD

There is a list of items. Given a specific word, e.g., “sun”, print out all the items in list which contain all the characters of “sum”

For example if the given word is “sun” and the items are “sunday”, “geeksforgeeks”, “utensils”, “”just” and “sss”, then the program should print “sunday” and “utensils”.

Algorithm: Thanks to geek4u for suggesting this algorithm.

```
1) Initialize a binary map:
        map[256] = {0, 0, ..}
2) Set values in map[] for the given word "sun"
        map['s'] = 1,  map['u'] = 1,  map['n'] = 1
3) Store length of the word "sun":
        len = 3 for "sun"
4) Pick words (or items)one by one from the list
    a) set count = 0;
    b) For each character ch of the picked word
         if(map['ch'] is set)
            increment count and unset map['ch']
    c) If count becomes equal to len (3 for "sun"),
           print the currently picked word.
    d) Set values in map[] for next list item
          map['s'] = 1,  map['u'] = 1,  map['n'] = 1
```

?

```
# include <stdio.h>
# include <stdlib.h>
# include <string.h>
# define NO_OF_CHARS 256

/* prints list items having all caharacters of word */
void print(char *list[], char *word, int list_size)
{
  /*Since calloc is used, map[] is initialized as 0 */
  int *map = (int *)calloc(sizeof(int), NO_OF_CHARS);
  int i, j, count, word_size;

  /*Set the values in map */
  for (i = 0; *(word+i); i++)
      map[*(word + i)] = 1;

  /* Get the length of given word */
  word_size = strlen(word);

  /* Check each item of list if has all characters
     of word*/
  for (i = 0; i < list_size; i++)
  {
    for(j = 0, count = 0; *(list[i] + j); j++)
    {
      if(map[*(list[i] + j)])
      {
         count++;

          /* unset the bit so that strings like
            sss not printed*/
         map[*(list[i] + j)] = 0;
      }
```

```
    }
    if(count == word_size)
      printf("\n %s", list[i]);

    /*Set the values in map for next item*/
    for (j = 0; *(word+j); j++)
      map[*(word + j)] = 1;
  }
}

/* Driver program to test to pront printDups*/
int main()
{
  char str[] = "sun";
  char *list[] = {"geeksforgeeks", "unsorted", "sunday", "just", "sss" };
  print(list, str, 5);
  getchar();
  return 0;
}
```

Time Complexity: O(n + m) where n is total number of characters in the list of items. And m = (number of items in list) * (number of characters in the given word)

Please write comments if you find any bug in above code/algorithm, or find other ways to solve the same problem

REVERSE WORDS IN A GIVEN STRING

Example: Let the input string be “i like this program very much”. The function should change the string to “much very program this like i”

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;
  }
}

/* Driver function to test above functions */
int main()
{
  char s[] = "i like this program very much";
  char *temp = s;
  reverseWords(s);
  printf("%s", s);
  getchar();
  return 0;
}
```

Time Complexity: O(n)

RUN LENGTH ENCODING

Given an input string, write a function that returns the Run Length Encoded string for the input string.

For example, if the input string is “wwwwaaadexxxxxx”, then the function should return “w4a3d1e1x6″.

Algorithm:
a) Pick the first character from source string.
b) Append the picked character to the destination string.
c) Count the number of subsequent occurrences of the picked character and append the count to destination string.
d) Pick the next character and repeat steps b) c) and d) if end of string is NOT reached.

?

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#define MAX_RLEN 50

/* Returns the Run Length Encoded string for the
   source string src */
char *encode(char *src)
{
  int rLen;
  char count[MAX_RLEN];
  int len = strlen(src);

  /* If all characters in the source string are different,
    then size of destination string would be twice of input string.
    For example if the src is "abcd", then dest would be "a1b1c1"
    For other inputs, size would be less than twice.  */
  char *dest = (char *)malloc(sizeof(char)*(len*2 + 1));

  int i, j = 0, k;

  /* traverse the input string one by one */
  for(i = 0; i < len; i++)
  {

    /* Copy the first occurrence of the new character */
    dest[j++] = src[i];

    /* Count the number of occurrences of the new character */
    rLen = 1;
    while(i + 1 < len && src[i] == src[i+1])
    {
      rLen++;
      i++;
    }

    /* Store rLen in a character array count[] */
    sprintf(count, "%d", rLen);

    /* Copy the count[] to destination */
    for(k = 0; *(count+k); k++, j++)
    {
      dest[j] = count[k];
    }
  }
```

```
  /*terminate the destination string */
  dest[j] = '\0';
  return dest;
}

/*driver program to test above function */
int main()
{
  char str[] = "geeksforgeeks";
  char *res = encode(str);
  printf("%s", res);
  getchar();
}
```

Time Complexity: O(n)

References:
http://en.wikipedia.org/wiki/Run-length_encoding

FIND THE SMALLEST WINDOW IN A STRING CONTAINING ALL CHARACTERS OF ANOTHER STRING

Given two strings string1 and string2, find the smallest substring in string1 containing all characters of string2 efficiently.

For Example:

Input string1: “this is a test string”
Input string2: “tist”
Output string: “t stri”

Method 1 (Brute force solution)
a) Generate all substrings of string1 (“this is a test string”)
b) For each substring, check whether the substring contains all characters of string2 (“tist”)
c) Finally print the smallest substring containing all characters of string2.

Method 2 (Efficient Solution)
1) Build a boolean count array count[] of string 2
count['i'] = 1
count['t'] = 2
count['s'] = 1

2) Scan the string1 from left to right until we find all the characters of string2. To check if all the characters are there, use count[] built in step 1. So we have substring “this is a t” containing all characters of string2. Note that the first and last characters of the substring must be present in string2. Store the length of this substring as min_len.

3) Now move forward in string1 and keep adding characters to the substring “this is a t”. Whenever a character is added, check if the added character matches the left most character of substring. If matches, then add the new character to the right side of substring and remove the leftmost character and all other extra characters after left most character. After removing the extra characters, get the length of this substring and compare with min_len and update min_len accordingly.

Basically we add ‘e’ to the substring “this is a t”, then add ‘s’ and then’t’. ‘t’ matches the left most character, so remove ‘t’ and ‘h’ from the left side of the substring. So our current substring becomes “is a test”. Compare length of it with min_len and update min_len.
Again add characters to current substring “is a test”. So our string becomes “is a test str”. When we add ‘i’, we remove leftmost extra characters, so current substring becomes “t stri”. Again, compare length of it with min_len and update min_len. Finally add ‘n’ and ‘g’. Adding these characters doesn’t decrease min_len, so the smallest window remains “t stri”.

4) Return min_len.

Please write comments if you find the above algorithms incorrect, or find other ways to solve the same problem.

Source: http://geeksforgeeks.org/forum/topic/find-smallest-substring-containing-all-characters-of-a-given-word

WRITE A FUNCTION TO GET NTH NODE IN A LINKED LIST

Write a GetNth() function that takes a linked list and an integer index and returns the data value stored in the node at that index position.

Algorithm:

```
1. Initialize count = 0
2. Loop through the link list
     a. if count is equal to the passed index then return current
         node
     b. Increment count
     c. change current to point to next of the current.
```

Implementation:

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

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

/* Given a reference (pointer to pointer) to the head
    of a list and an int, push a new node on the front
    of the list. */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
             (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Takes head pointer of the linked list and index
    as arguments and return data at index*/
int GetNth(struct node* head, int index)
{
```

```
    struct node* current = head;
    int count = 0; /* the index of the node we're currently
                  looking at */
    while (current != NULL)
    {
       if (count == index)
          return(current->data);
       count++;
       current = current->next;
    }

    /* if we get to this line, the caller was asking
       for a non-existent element so we assert fail */
    assert(0);
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

    /* Use push() to construct below list
     1->12->1->4->1  */
    push(&head, 1);
    push(&head, 4);
    push(&head, 1);
    push(&head, 12);
    push(&head, 1);

    /* Check the count function */
    printf("Element at index 3 is %d", GetNth(head, 3));
    getchar();
}
```

Time Complexity: O(n)

GIVEN ONLY A POINTER TO A NODE TO BE DELETED IN A SINGLY LINKED LIST, HOW DO YOU DELETE IT?

A **simple solution** is to traverse the linked list until you find the node you want to delete. But this solution requires pointer to the head node which contradicts the problem statement.

Fast solution is to copy the data from the next node to the node to be deleted and delete the next node. Something like following.

```
    struct node *temp  = node_ptr->next;
    node_ptr->data  = temp->data;
    node_ptr->next  = temp->next;
    free(temp);
```

Program:

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

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

/* Given a reference (pointer to pointer) to the head
    of a list and an int, push a new node on the front
    of the list. */
void push(struct node** head_ref, int new_data)
{
   /* allocate node */
   struct node* new_node =
             (struct node*) malloc(sizeof(struct node));

   /* put in the data  */
   new_node->data  = new_data;

   /* link the old list off the new node */
   new_node->next = (*head_ref);

   /* move the head to point to the new node */
   (*head_ref)    = new_node;
}

void printList(struct node *head)
{
   struct node *temp = head;
   while(temp != NULL)
   {
      printf("%d  ", temp->data);
      temp = temp->next;
   }
}

void deleteNode(struct node *node_ptr)
{
   struct node *temp = node_ptr->next;
   node_ptr->data    = temp->data;
   node_ptr->next    = temp->next;
   free(temp);
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;
```

```
    /* Use push() to construct below list
    1->12->1->4->1  */
    push(&head, 1);
    push(&head, 4);
    push(&head, 1);
    push(&head, 12);
    push(&head, 1);

    printf("\n Before deleting \n");
    printList(head);

    /* I m deleting the head itself.
        You can check for more cases */
    deleteNode(head);

    printf("\n After deleting \n");
    printList(head);
    getchar();
}
```

This solution doesn't work if the node to be deleted is the last node of the list. To make this solution work we can mark the end node as a dummy node. But the programs/functions that are using this function should also be modified.

WRITE A C FUNCTION TO PRINT THE MIDDLE 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.

Implementation of Method 2:

```
?
#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->next)!=NULL &&
                (fast_ptr->next->next)!=NULL)
       {
          fast_ptr = fast_ptr->next->next;
          slow_ptr = slow_ptr->next;
       }
       printf("The middle element is [%d]",slow_ptr->data);
  }
}

void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

     push(&head, 20);
     push(&head, 4);
     push(&head, 15);

     printMiddle(head);
     getchar();
}
```

Time Complexity: O(n)

NTH NODE FROM THE END OF A LINKED LIST

Algorithm
Maintain two pointers – reference pointer and main pointer. Initialize both reference and main

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

Implementation:

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

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

/* Function to get the nth node from the last of a linked list*/
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);
  }
}

void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
           (struct node*) malloc(sizeof(struct node));
```

```
  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

/* Drier program to test above function*/
int main()
{
  /* Start with the empty list */
  struct node* head = NULL;
  push(&head, 20);
  push(&head, 4);
  push(&head, 15);

  printNthFromLast(head, 3);
  getchar();
}
```

Time Complexity: O(n)

WRITE A FUNCTION TO DELETE A LINKED LIST

Algorithm: Iterate through the linked list and delete all the nodes one by one. Main point here is not to access next of the current pointer if current pointer is deleted.

Implementation:

```
?
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>

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

/* Function to delete the entire linked list */
void deleteList(struct node** head_ref)
{
   /* deref head_ref to get the real head */
   struct node* current = *head_ref;
   struct node* next;

   while (current != NULL)
```

```
    {
        next = current->next;
        free(current);
        current = next;
    }

    /* deref head_ref to affect the real head back
       in the caller. */
    *head_ref = NULL;
}

/* Given a reference (pointer to pointer) to the head
  of a list and an int, push a new node on the front
  of the list. */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Drier program to test count function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

    /* Use push() to construct below list
     1->12->1->4->1  */
    push(&head, 1);
    push(&head, 4);
    push(&head, 1);
    push(&head, 12);
    push(&head, 1);

    printf("\n Deleting linked list");
    deleteList(&head);

    printf("\n Linked list deleted");
    getchar();
}
```

Time Complexity: O(n)
Space Complexity: O(1)

WRITE A FUNCTION THAT COUNTS THE NUMBER OF TIMES A GIVEN INT OCCURS IN A LINKED LIST

Here is a solution.

Algorithm:

```
1. Initialize count as zero.
2. Loop through each element of linked list:
     a) If element data is equal to the passed number then
        increment the count.
3. Return count.
```

Implementation:

?

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

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

/* Given a reference (pointer to pointer) to the head
  of a list and an int, push a new node on the front
  of the list. */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Counts the no. of occurences of a node
   (search_for) in a linked list (head)*/
int count(struct node* head, int search_for)
{
    struct node* current = head;
    int count = 0;
```

```
    while (current != NULL)
    {
        if (current->data == search_for)
           count++;
        current = current->next;
    }
    return count;
}

/* Drier program to test count function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

    /* Use push() to construct below list
     1->2->1->3->1  */
    push(&head, 1);
    push(&head, 3);
    push(&head, 1);
    push(&head, 2);
    push(&head, 1);

    /* Check the count function */
    printf("count of 1 is %d", count(head, 1));
    getchar();
}
```

Time Complexity: O(n)
Space Complexity: O(1)

WRITE A FUNCTION TO REVERSE A LINKED LIST

Iterative Method
Iterate trough the linked list. In loop, change next to prev, prev to current and current to next.

Implementation of Iterative Method

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

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

/* 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;
}

/* Function to push a node */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print linked list */
void printList(struct node *head)
{
    struct node *temp = head;
    while(temp != NULL)
    {
        printf("%d  ", temp->data);
        temp = temp->next;
    }
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

     push(&head, 20);
     push(&head, 4);
     push(&head, 15);
     push(&head, 85);

     printList(head);
     reverse(&head);
     printf("\n Reversed Linked list \n");
```

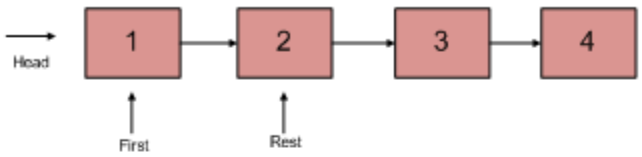
```
    printList(head);
    getchar();
}
```

Time Complexity: O(n)
Space Complexity: O(1)

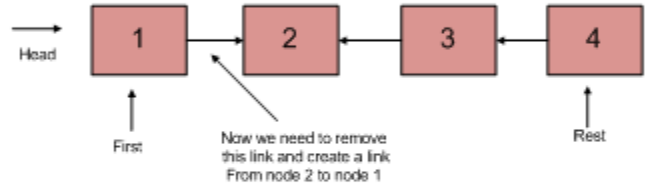
Recursive Method:

```
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
```

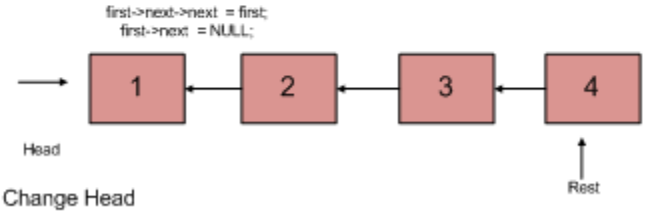
Divide the List in two parts



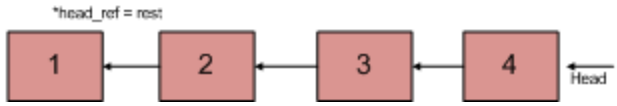
Reverse Rest



Link Rest to First



Change Head



[?]

```
void recursiveReverse(struct node** head_ref)
{
    struct node* first;
    struct node* rest;

    /* empty list */
    if (*head_ref == NULL)
       return;

    /* suppose first = {1, 2, 3}, rest = {2, 3} */
```

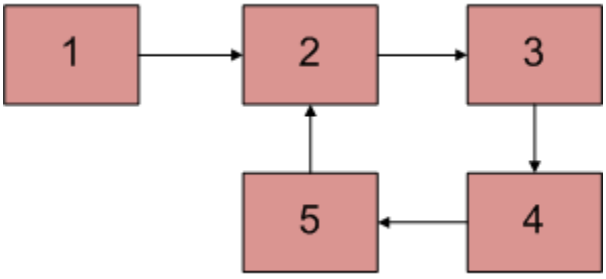
```
    first = *head_ref;
    rest  = first->next;

    /* List has only one node */
    if (rest == NULL)
       return;

    /* put the first element on the end of the list */
    recursiveReverse(&rest);
    first->next->next  = first;

    /* tricky step -- see the diagram */
    first->next  = NULL;

    /* fix the head pointer */
    *head_ref = rest;
}
```

Time Complexity: O(n)
Space Complexity: O(1)

References:
http://cslibrary.stanford.edu/105/LinkedListProblems.pdf

WRITE A C FUNCTION TO DETECT LOOP IN A LINKED LIST

Below diagram shows a linked list with a loop



Following are different ways of doing this
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:
This is the fastest method. Traverse linked list using two pointers. Move one pointer by one and other pointer by two. If these pointers meet at some node then there is a loop. If pointers do not meet then linked list doesn't have loop.

Implementation of Floyd's Cycle-Finding Algorithm:

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

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

void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
  (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

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;
}

/* Drier program to test above function*/
int main()
{
  /* Start with the empty list */
  struct node* head = NULL;

  push(&head, 20);
  push(&head, 4);
  push(&head, 15);
  push(&head, 10);

  /* Create a loop for testing */
  head->next->next->next->next = head;
  detectloop(head);

  getchar();
}
```

Time Complexity: O(n)
Space Complexity: O(1)

References:
http://en.wikipedia.org/wiki/Cycle_detection
http://ostermiller.org/find_loop_singly_linked_list.html

FUNCTION TO CHECK IF A SINGLY LINKED LIST IS PALINDROME

METHOD 1 (By reversing the list)

1. Get the middle of the linked list.
2. Reverse the second half of the linked list.
3. Compare the first half and second half.
4. Construct the original linked list by reversing the second half again and attaching it back to the first half

Implementation:

```
/* Program to check if a linked list is palindrome */
#include<stdio.h>
#include<stdlib.h>
#define bool int

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

void reverse(struct node**);
bool compareLists(struct node*, struct node *);

/* Function to check if given linked list is
  palindrome or not */
bool isPalindrome(struct node *head)
{
   struct node *slow_ptr = head;
   struct node *fast_ptr = head;
   struct node *second_half;
   struct node *prev_of_slow_ptr = head;
   char res;

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

       /* Case where we have even no of elements */
       if(fast_ptr->next != NULL)
       {
         second_half = slow_ptr->next;
         reverse(&second_half);
         slow_ptr->next = NULL;
         res = compareLists(head, second_half);

         /*construct the original list back*/
         reverse(&second_half);
         slow_ptr->next = second_half;
```

```
        }

        /* Case where we have odd no. of elements. Neither first
           nor second list should have the middle element */
        else
        {
           second_half = slow_ptr->next;
           prev_of_slow_ptr->next = NULL;
           reverse(&second_half);
           res = compareLists(head, second_half);

          /*construct the original list back*/
          reverse(&second_half);
          prev_of_slow_ptr->next = slow_ptr;
          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*/
int 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;
}

/* Push a node to linked list. Note that this function
  changes the head */
void push(struct node** head_ref, char new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to pochar to the new node */
    (*head_ref)    = new_node;
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

     push(&head, 'p');
     push(&head, 'e');
     push(&head, 'e');
     push(&head, 'p');

     /* p->e->e->p */
     if(isPalindrome(head) == 1)
       printf("Linked list is Palindrome");
     else
       printf("Linked list is not Palindrome");

     getchar();
     return 0;
}
```

Time Complexity O(n)
Auxiliary Space: O(1)

METHOD 2 (Using Recursion)
Use two pointers left and right. Move right and left using recursion and check for following in

each recursive call.
1) Sub-list is palindrome.
2) Value at current left and right are matching.

If both above conditions are true then return true.

```
#define bool int
#include<stdio.h>
#include<stdlib.h>

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

bool isPalindrome(struct node **left, struct node *right)
{
   /* stop recursion here */
   if (!right)
      return true;

   /* If sub-list is not palindrome then no need to
       check for current left and right, return false */
   bool isp = isPalindrome(left, right->next);
   if (isp == false)
      return false;

   /* Check values at current left and right */
   bool isp1 = (right->data == (*left)->data);

   /* Move left to next node */
   *left = (*left)->next; /* save next pointer */

   return isp1;
}

/* UTILITY FUNCTIONS */
/* Push a node to linked list. Note that this function
  changes the head */
void push(struct node** head_ref, char new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);
```

```
    /* move the head to pochar to the new node */
    (*head_ref)    = new_node;
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

     push(&head, 'r');
     push(&head, 'a');
     push(&head, 'd');
     push(&head, 'a');
     push(&head, 'r');

     /* r->a->d->a->r*/
     if(isPalindrome(&head, head) == 1)
       printf("Linked list is Palindrome");
     else
       printf("Linked list is not Palindrome");

     getchar();
     return 0;
}
```

Time Complexity: O(n)
Auxiliary Space: O(n) if Function Call Stack size is considered, otherwise O(1).

COPY A LINKED LIST WITH NEXT AND ARBIT POINTER

Question:
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.

1. Solution for Restricted Version of the Question:
Let us first solve the restricted version of the original question. The restriction here is that a node will be pointed by only one arbit pointer in a linked list. In below diagram, we have obeyed the restriction.

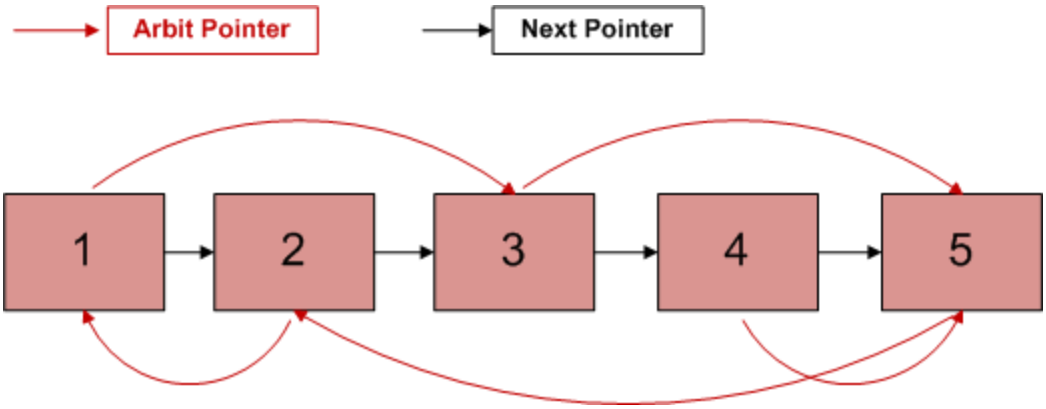


Figure 1

Algorithm:
1) Create all nodes in copy linked list using next pointers.
2) Change next of original linked list to the corresponding node in copy linked list.
3) Change the arbit pointer of copy linked list to point corresponding node in original linked list.

See below diagram after above three steps.

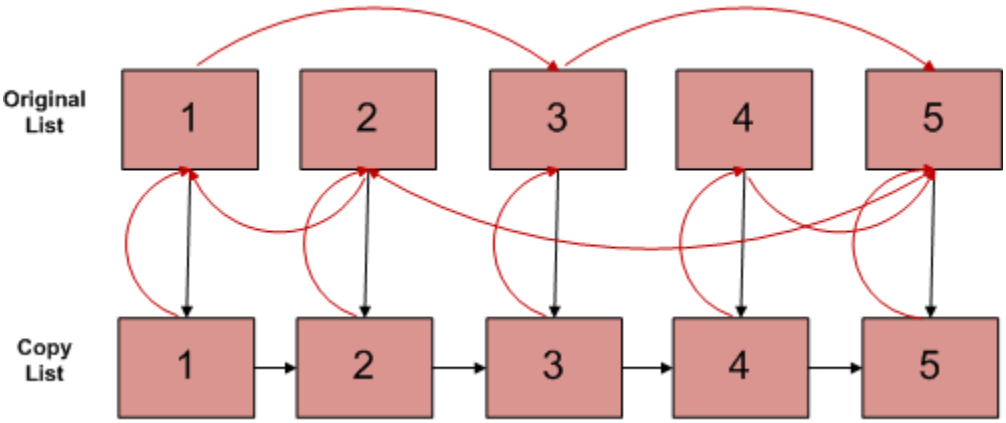


Figure 2

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

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

/* This can not be done if we remove the restriction*/
orig_list_node->next =
                orig_list_node->next->next->arbit
```

Time Complexity: O(n)
Space Complexity: O(1)

2. Solution for the Original Question:
If we remove the restriction given in above solution then we CANNOT restore the next pointer of the original linked in step 4 of above solution. We have to store the node and it's next pointer mapping in original linked list. Below diagram doesn't obey the restriction as node '3′ is pointed by arbit of '1′ and '4′.

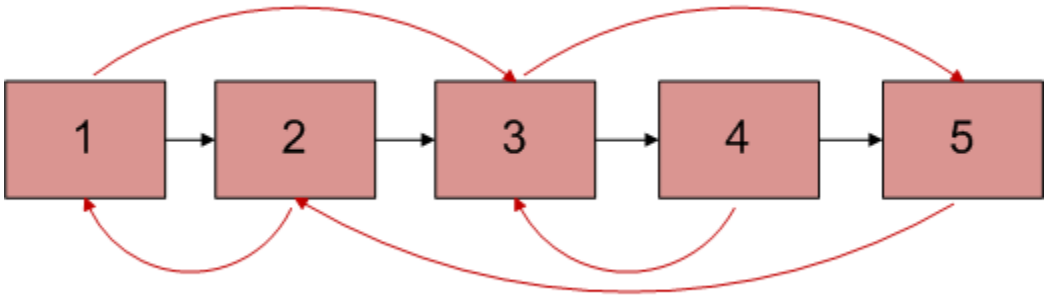


Figure 3

Algorithm:
1) Create all nodes in copy linked list using next pointers.
3) Store the node and it's next pointer mappings of original linked list.
3) Change next of original linked list to the corresponding node in copy linked list.

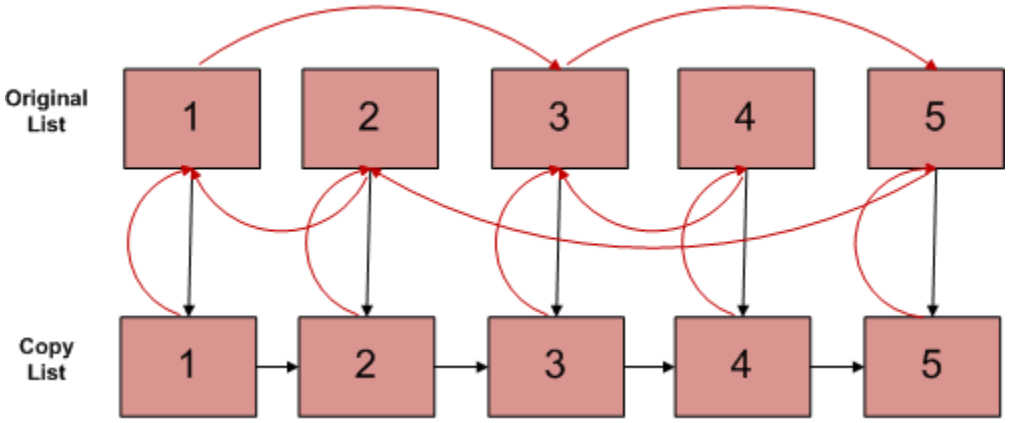


Figure 4

4) Change the arbit pointer of copy linked list to point corresponding node in original linked list.
5) Now construct the arbit pointer in copy linked list as below and restore the next pointer of the original linked list.

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

6) Restore the node and it's next pointer of original linked list from the stored mappings(in step 2).

Time Complexity: O(n)
Space Complexity: O(n)

3. A Constant Space and O(n) Time Solution for the Original Question:
This is the best solution among all three as it works in constant space and without any restriction on original linked list.

1) Create the copy of 1 and insert it between 1 & 2, create the copy of 2 and insert it between 2 & 3.. Continue in this fashion, add the copy of N to Nth node

2) Now copy the arbitrary link in this fashion

```
     original->next->arbitrary = original->arbitrary->next;  /*TRAVERSE TWO
NODES*/
```

This works because original->next is nothing but copy of original and Original->arbitrary->next is nothing but copy of arbitrary.

3) Now restore the original and copy linked lists in this fashion in a single loop.

```
     original->next = original->next->next;
     copy->next = copy->next->next;
```

4) Make sure that last element of original->next is NULL.

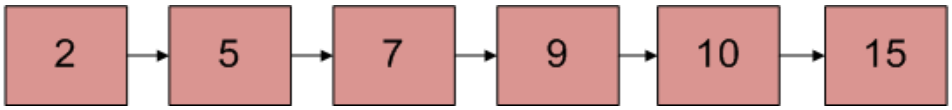
Time Complexity: O(n)
Space Complexity: O(1)

GIVEN A LINKED LIST WHICH IS SORTED, HOW WILL YOU INSERT IN SORTED WAY

Algorithm:
Let input linked list is sorted in increasing order.

```
1) If Linked list is empty then make the node as head and return it.
2) If value of the node to be inserted is smaller than value of head node
    then insert the node at start and make it head.
3) In a loop, find the appropriate node after which the input node (let 9) is
    to be inserted. To find the appropriate node start from head, keep moving
    until you reach a node GN (10 in the below diagram) who's value is
    greater than the input node. The node just before GN is the appropriate
    node (7).
4) Insert the node (9) after the appropriate node (7) found in step 3.
```

Linked List after insertion of 9



Implementation:

?

```
/* Program to insert in a sorted list */
#include<stdio.h>
#include<stdlib.h>

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

/* function to insert a new_node in a list. Note that this
  function expects a pointer to head_ref as this can modify the
  head of the input linked list (similar to push())*/
void sortedInsert(struct node** head_ref, struct node* new_node)
{
  struct node* current;
  /* Special case for the head end */
  if (*head_ref == NULL || (*head_ref)->data >= new_node->data)
  {
    new_node->next = *head_ref;
    *head_ref = new_node;
  }
  else
  {
    /* Locate the node before the point of insertion */
    current = *head_ref;
    while (current->next!=NULL && current->next->data < new_node->data)
    {
      current = current->next;
    }
    new_node->next = current->next;
    current->next = new_node;
  }
}

/* BELOW FUNCTIONS ARE JUST UTILITY TO TEST sortedInsert */

/* Given a reference (pointer to pointer) to the head
```

```
  of a list and an int, push a new node on the front
  of the list. */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print linked list */
void printList(struct node *head)
{
    struct node *temp = head;
    while(temp != NULL)
    {
        printf("%d  ", temp->data);
        temp = temp->next;
    }
}

/* Drier program to test count function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;
    int value_to_insert;

    /* Use push() to construct below list
     2->5->7->10->15  */
    push(&head, 15);
    push(&head, 10);
    push(&head, 7);
    push(&head, 5);
    push(&head, 2);

    /* Let us try inserting 9 */
    value_to_insert = 9;
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));
    new_node->data  = value_to_insert;

    printf("\n List before insertion of %d \n", value_to_insert);
    printList(head);

    sortedInsert(&head, new_node);
```

```
    printf("\n List after insertion of %d \n", value_to_insert);
    printList(head);

    getchar();
    return 1;
}
```

Shorter Implementation using double pointers
The code uses double pointer to keep track of the next pointer of the previous node (after which new node is being inserted).

Note that below line in code changes *current* to have address of next pointer in a node.

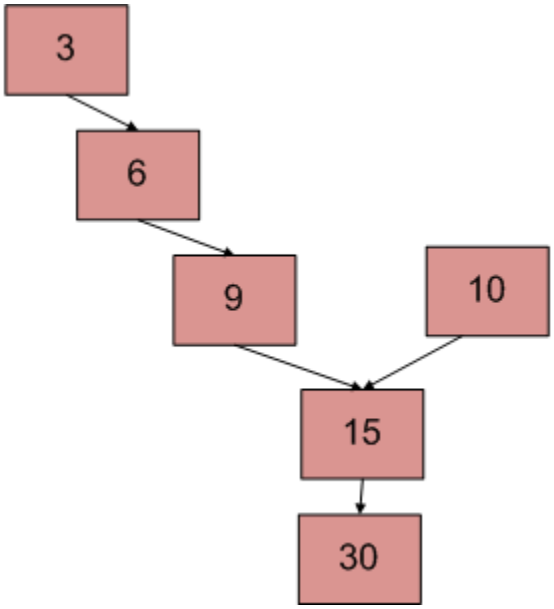
```
current = &((*current)->next);
```

Also, note below comments.

```
new_node->next = *current; /* Copies the value-at-address current to
new_node's next pointer*/

*current = new_node;  /* Fix next pointer of the node (using it's address)
after which new_node is being inserted */
```

WRITE A FUNCTION TO GET THE INTERSECTION POINT OF TWO LINKED LISTS.

There are two singly linked lists in a system. By some programming error the end node of one of the linked list got linked into the second list, forming a inverted Y shaped list. Write a program to get the point where two linked list merge.



Above diagram shows an example with two linked list having 15 as intersection point.

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;
```

```
}

/* IGNORE THE BELOW LINES OF CODE. THESE LINES
   ARE JUST TO QUICKLY TEST THE ABOVE FUNCTION */
int main()
{
  /*
    Create two linked lists

    1st 3->6->9->15->30
    2nd 10->15->30

    15 is the intersection point
  */

  struct node* newNode;
  struct node* head1 =
            (struct node*) malloc(sizeof(struct node));
  head1->data  = 10;

  struct node* head2 =
            (struct node*) malloc(sizeof(struct node));
  head2->data  = 3;

  newNode = (struct node*) malloc (sizeof(struct node));
  newNode->data = 6;
  head2->next = newNode;

  newNode = (struct node*) malloc (sizeof(struct node));
  newNode->data = 9;
  head2->next->next = newNode;

  newNode = (struct node*) malloc (sizeof(struct node));
  newNode->data = 15;
  head1->next = newNode;
  head2->next->next->next  = newNode;

  newNode = (struct node*) malloc (sizeof(struct node));
  newNode->data = 30;
  head1->next->next= newNode;

  head1->next->next->next = NULL;

  printf("\n The node of intersection is %d \n",
          getIntesectionNode(head1, head2));

  getchar();
}
```

Time Complexity: O(m+n)
Auxiliary Space: O(1)

Method 4(Make circle in first list)
Thanks to Saravanan Man for providing below solution.
1. Traverse the first linked list(count the elements) and make a circular linked list. (Remember last node so that we can break the circle later on).
2. Now view the problem as find the loop in the second linked list. So the problem is solved.
3. Since we already know the length of the loop(size of first linked list) we can traverse those many number of nodes in second list, and then start another pointer from the beginning of second list. we have to traverse until they are equal, and that is the required intersection point.
4. remove the circle from the linked list.

Time Complexity: O(m+n)
Auxiliary Space: O(1)

Method 5 (Reverse the first list and make equations)
Thanks to Saravanan Mani for providing this method.

```
1) Let X be the length of the first linked list until intersection point.
   Let Y be the length of the second linked list until the intersection
point.
   Let Z be the length of the linked list from intersection point to End of
   the linked list including the intersection node.
   We Have
           X + Z = C1;
           Y + Z = C2;
2) Reverse first linked list.
3) Traverse Second linked list. Let C3 be the length of second list - 1.
     Now we have
        X + Y = C3
     We have 3 linear equations. By solving them, we get
       X = (C1 + C3 - C2)/2;
       Y = (C2 + C3 - C1)/2;
       Z = (C1 + C2 - C3)/2;
      WE GOT THE INTERSECTION POINT.
4)  Reverse first linked list.
```

Advantage: No Comparison of pointers.
Disadvantage : Modifying linked list(Reversing list).

Time complexity: O(m+n)
Auxiliary Space: O(1)

Method 6 (Traverse both lists and compare addresses of last nodes) This method is only to detect if there is an intersection point or not.

```
1) Traverse the list 1, store the last node address
2) Traverse the list 2, store the last node address.
3) If nodes stored in 1 and 2 are same then they are intersecting.
```

Time complexity of this method is O(m+n) and used Auxiliary space is O(1)

WRITE A RECURSIVE FUNCTION TO PRINT REVERSE OF A LINKED LIST

Note that the question is only about printing the reverse. To reverse the list itself see this

Difficulty Level: Rookie

Algorithm

```
printReverse(head)
  1. call print reverse for hed->next
  2. print head->data
```

Implementation:

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

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

/* Function to reverse the linked list */
void printReverse(struct node* head)
{
  if(head == NULL)
    return;

  printReverse(head->next);
  printf("%d  ", head->data);
}

/*UTILITY FUNCTIONS*/
/* Push a node to linked list. Note that this function
  changes the head */
void push(struct node** head_ref, char new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to pochar to the new node */
    (*head_ref)    = new_node;
}
```

```
/* Drier program to test above function*/
int main()
{

  struct node* head = NULL;

  push(&head, 1);
  push(&head, 2);
  push(&head, 3);
  push(&head, 4);

  printReverse(head);
  getchar();
}
```

Time Complexity: O(n)

REMOVE DUPLICATES FROM A SORTED LINKED LIST

Write a removeDuplicates() function which takes a list sorted in non-decreasing order and deletes any duplicate nodes from the list. The list should only be traversed once.

For example if the linked list is 11->11->11->21->43->43->60 then removeDuplicates() should convert the list to 11->21->43->60.

Algorithm:
Traverse the list from the head (or start) node. While traversing, compare each node with its next node. If data of next node is same as current node then delete the next node. Before we delete a node, we need to store next pointer of the node

Implementation:
Functions other than removeDuplicates() are just to create a linked linked list and test removeDuplicates().

?

```
/*Program to remove duplicates from a sorted linked list */
#include<stdio.h>
#include<stlib.h>

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

/* The function removes duplicates from a sorted list */
void removeDuplicates(struct node* head)
{
```

```
  /* Pointer to traverse the linked list */
  struct node* current = head;

  /* Pointer to store the next pointer of a node to be deleted*/
  struct node* next_next;

  /* do nothing if the list is empty */
  if(current == NULL)
     return;

  /* Traverse the list till last node */
  while(current->next != NULL)
  {
   /* Compare current node with next node */
    if(current->data == current->next->data)
    {
       /*The sequence of steps is important*/
      next_next = current->next->next;
      free(current->next);
      current->next = next_next;
    }
    else /* This is tricky: only advance if no deletion */
    {
      current = current->next;
    }
  }
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginging of the linked list */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while(node!=NULL)
  {
   printf("%d ", node->data);
   node = node->next;
  }
}
```

```
/* Drier program to test above functions*/
int main()
{
  /* Start with the empty list */
  struct node* head = NULL;

  /* Let us create a sorted linked list to test the functions
   Created linked list will be 11->11->11->13->13->20 */
  push(&head, 20);
  push(&head, 13);
  push(&head, 13);
  push(&head, 11);
  push(&head, 11);
  push(&head, 11);

  printf("\n Linked list before duplicate removal  ");
  printList(head);

  /* Remove duplicates from linked list */
  removeDuplicates(head);

  printf("\n Linked list after duplicate removal ");
  printList(head);

  getchar();
}
```

Time Complexity: O(n) where n is number of nodes in the given linked list.

REMOVE DUPLICATES FROM AN UNSORTED LINKED LIST

Write a removeDuplicates() function which takes a list and deletes any duplicate nodes from the list. The list is not sorted.

For example if the linked list is 12->11->12->21->41->43->21 then removeDuplicates() should convert the list to 12->11->21->41->43.

METHOD 1 (Using two loops)
This is the simple way where two loops are used. Outer loop is used to pick the elements one by one and inner loop compares the picked element with rest of the elements.

```
/* Program to remove duplicates in an unsorted array */

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

/* A linked list node */
struct node
{
```

```
 int data;
 struct node *next;
};

/* 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;
          free(dup);
       }
       else /* This is tricky */
       {
          ptr2 = ptr2->next;
       }
     }
     ptr1 = ptr1->next;
  }
}

/* UTILITY FUNCTIONS */
/* Function to push a node */
void push(struct node** head_ref, int new_data);

/* Function to print nodes in a given linked list */
void printList(struct node *node);

/* Driver program to test above function */
int main()
{
  struct node *start = NULL;

  /* The constructed linked list is:
   10->12->11->11->12->11->10*/
  push(&start, 10);
  push(&start, 11);
  push(&start, 12);
  push(&start, 11);
  push(&start, 11);
  push(&start, 12);
```

```
  push(&start, 10);

  printf("\n Linked list before removing duplicates ");
  printList(start);

  removeDuplicates(start);

  printf("\n Linked list after removing duplicates ");
  printList(start);

  getchar();
}

/* Function to push a node */
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while(node != NULL)
  {
    printf("%d ", node->data);
    node = node->next;
  }
}
```

Time Complexity: O(n^2)

METHOD 2 (Use Sorting)
In general, Merge Sort is the best suited sorting algorithm for sorting linked lists efficiently.
1) Sort the elements using Merge Sort. We will soon be writing a post about sorting a linked list. O(nLogn)
2) Remove duplicates in linear time using the algorithm for removing duplicates in sorted Linked List. O(n)

Time Complexity: O(nLogn)

METHOD 3 (Use Hashing)
We traverse the link list from head to end. For the newly encountered element, we check whether it is in the hash table: if yes, we remove it; otherwise we put it in the hash table.

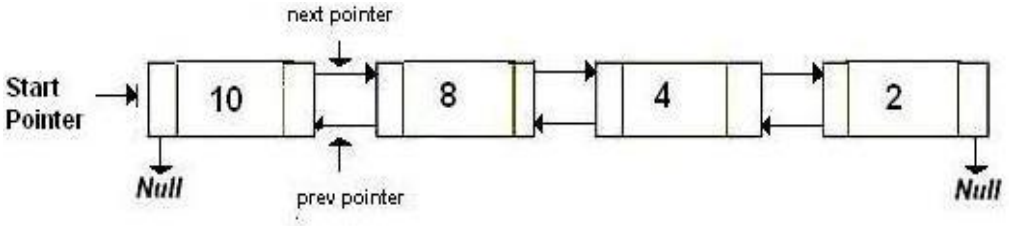
Time Complexity: O(n) on average (assuming that hash table access time is O(1) on average).

REVERSE A DOUBLY LINKED LIST

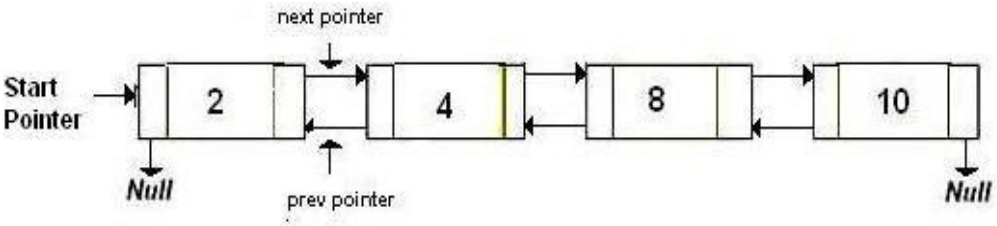
Write a C function to reverse a given Doubly Linked List

See below diagrams for example.

(a) Original Doubly Linked List



(b) Reversed Doubly Linked List



Here is a simple method for reversing a Doubly Linked List. All we need to do is swap prev and next pointers for all nodes, change prev of the head (or start) and change the head pointer in the end.

```
/* Program to reverse a doubly linked list */
#include <stdio.h>
#include <stdlib.h>

/* a node of the doubly linked list */
struct node
{
  int data;
  struct node *next;
  struct node *prev;
};
```

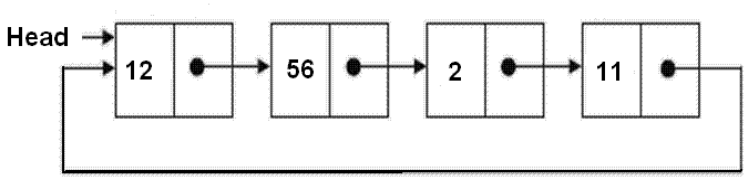
```
/* Function to reverse a Doubly Linked List */
void reverse(struct node **head_ref)
{
     struct node *temp = NULL;
     struct node *current = *head_ref;

     /* swap next and prev for all nodes of
       doubly linked list */
     while (current !=  NULL)
     {
       temp = current->prev;
       current->prev = current->next;
       current->next = temp;
       current = current->prev;
     }

     /* Before changing head, check for the cases like empty
        list and list with only one node */
     if(temp != NULL )
        *head_ref = temp->prev;
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginging of the Doubly Linked List */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* since we are adding at the begining,
      prev is always NULL */
    new_node->prev = NULL;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* change prev of head node to new node */
    if((*head_ref) !=  NULL)
      (*head_ref)->prev = new_node ;

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print nodes in a given doubly linked list
   This function is same as printList() of singly linked lsit */
void printList(struct node *node)
{
  while(node!=NULL)
  {
```

```
    printf("%d ", node->data);
    node = node->next;
   }
}

/* Drier program to test above functions*/
int main()
{
  /* Start with the empty list */
  struct node* head = NULL;

  /* Let us create a sorted linked list to test the functions
   Created linked list will be 10->8->4->2 */
  push(&head, 2);
  push(&head, 4);
  push(&head, 8);
  push(&head, 10);

  printf("\n Original Linked list ");
  printList(head);

  /* Reverse doubly linked list */
  reverse(&head);

  printf("\n Reversed Linked list ");
  printList(head);

  getchar();
}
```

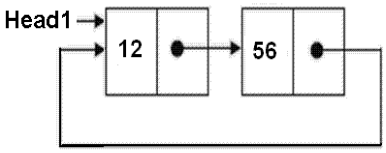
Time Complexity: O(n)

We can also swap data instead of pointers to reverse the Doubly Linked List. Method used for reversing array can be used to swap data. Swapping data can be costly compared to pointers if size of data item(s) is more.

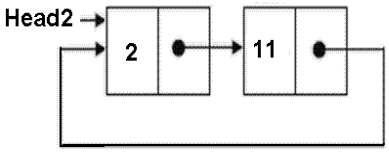
SPLIT A CIRCULAR LINKED LIST INTO TWO HALVES



Original Linked List



Result Linked List 1



Result Linked List 2

1) Store the mid and last pointers of the circular linked list using tortoise and hare algorithm.
2) Make the second half circular.
3) Make the first half circular.
4) Set head (or start) pointers of the two linked lists.

In the below implementation, if there are odd nodes in the given circular linked list then the first result list has 1 more node than the second result list.

```
/* Program to split a circular linked list into two halves */
#include<stdio.h>
#include<stdlib.h>

/* structure for a node */
struct node
{
  int data;
  struct node *next;
};

/* Function to split a list (starting with head) into two lists.
   head1_ref and head2_ref are references to head nodes of
    the two resultant linked lists */
void splitList(struct node *head, struct node **head1_ref,
                                                 struct node **head2_ref)
{
  struct node *slow_ptr = head;
  struct node *fast_ptr = head;

  if(head == NULL)
    return;

  /* If there are odd nodes in the circular list then
     fast_ptr->next becomes head and for even nodes
```

```
     fast_ptr->next->next becomes head */
  while(fast_ptr->next != head &&
        fast_ptr->next->next != head)
  {
     fast_ptr = fast_ptr->next->next;
     slow_ptr = slow_ptr->next;
  }

 /* If there are even elements in list then move fast_ptr */
  if(fast_ptr->next->next == head)
    fast_ptr = fast_ptr->next;

  /* Set the head pointer of first half */
  *head1_ref = head;

  /* Set the head pointer of second half */
  if(head->next != head)
    *head2_ref = slow_ptr->next;

  /* Make second half circular */
  fast_ptr->next = slow_ptr->next;

  /* Make first half circular */
  slow_ptr->next = head;
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the begining of a Circular
   linked lsit */
void push(struct node **head_ref, int data)
{
  struct node *ptr1 = (struct node *)malloc(sizeof(struct node));
  struct node *temp = *head_ref;
  ptr1->data = data;
  ptr1->next = *head_ref;

  /* If linked list is not NULL then set the next of
    last node */
  if(*head_ref != NULL)
  {
    while(temp->next != *head_ref)
      temp = temp->next;
    temp->next = ptr1;
  }
  else
     ptr1->next = ptr1; /*For the first node */

  *head_ref = ptr1;
}

/* Function to print nodes in a given Circular linked list */
void printList(struct node *head)
{
  struct node *temp = head;
  if(head != NULL)
```

```
  {
    printf("\n");
    do {
      printf("%d ", temp->data);
      temp = temp->next;
    } while(temp != head);
  }
}

/* Driver program to test above functions */
int main()
{
  int list_size, i;

  /* Initialize lists as empty */
  struct node *head = NULL;
  struct node *head1 = NULL;
  struct node *head2 = NULL;

  /* Created linked list will be 12->56->2->11 */
  push(&head, 12);
  push(&head, 56);
  push(&head, 2);
  push(&head, 11);

  printf("Original Circular Linked List");
  printList(head);

  /* Split the list */
  splitList(head, &head1, &head2);

  printf("\nFirst Circular Linked List");
  printList(head1);

  printf("\nSecond Circular Linked List");
  printList(head2);

  getchar();
  return 0;
}
```

Time Complexity: O(n)

PRACTICE QUESTIONS FOR LINKED LIST AND RECURSION

Assume the structure of a Linked List node is as follows.

```
struct node
{
  int data;
  struct node *next;
```

```
};
```

Explain the functionality of following C functions.

1. What does the following function do for a given Linked List?

```
?
void fun1(struct node* head)
{
  if(head == NULL)
    return;

  fun1(head->next);
  printf("%d  ", head->data);
}
```

fun1() prints the given Linked List in reverse manner. For Linked List 1->2->3->4->5, fun1() prints 5->4->3->2->1.

2. What does the following function do for a given Linked List ?

```
?
void fun2(struct node* head)
{
  if(head== NULL)
    return;
  printf("%d  ", head->data);

  if(head->next != NULL )
    fun2(head->next->next);
  printf("%d  ", head->data);
}
```

fun2() prints alternate nodes of the given Linked List, first from head to end, and then from end to head. If Linked List has even number of nodes, then fun2() skips the last node. For Linked List 1->2->3->4->5, fun2() prints 1 3 5 5 3 1. For Linked List 1->2->3->4->5->6, fun2() prints 1 3 5 5 3 1.

Below is a complete running program to test above functions.

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

/* A linked list node */
struct node
{
  int data;
  struct node *next;
};
```

```
/* Prints a linked list in reverse manner */
void fun1(struct node* head)
{
  if(head == NULL)
    return;

  fun1(head->next);
  printf("%d  ", head->data);
}

/* prints alternate nodes of a Linked List, first
  from head to end, and then from end to head. */
void fun2(struct node* start)
{
  if(start == NULL)
    return;
  printf("%d  ", start->data);

  if(start->next != NULL )
    fun2(start->next->next);
  printf("%d  ", start->data);
}

/* UTILITY FUNCTIONS TO TEST fun1() and fun2() */
/* Given a reference (pointer to pointer) to the head
  of a list and an int, push a new node on the front
  of the list. */
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
          (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

/* Drier program to test above functions */
int main()
{
  /* Start with the empty list */
  struct node* head = NULL;

  /* Using push() to construct below list
    1->2->3->4->5  */
  push(&head, 5);
  push(&head, 4);
  push(&head, 3);
```

```
  push(&head, 2);
  push(&head, 1);

  printf("\n Output of fun1() for list 1->2->3->4->5 \n");
  fun1(head);

  printf("\n Output of fun2() for list 1->2->3->4->5 \n");
  fun2(head);

  getchar();
  return 0;
}
```

MOVE LAST ELEMENT TO FRONT OF A GIVEN LINKED LIST

Write a C function that moves last element to front in a given Singly Linked List. For example, if the given Linked List is 1->2->3->4->5, then the function should change the list to 5->1->2->3->4.

Algorithm:
Traverse the list till last node. Use two pointers: one to store the address of last node and other for address of second last node. After the end of loop do following operations.
i) Make second last as last (secLast->next = NULL).
ii) Set next of last as head (last->next = *head_ref).
iii) Make last as head (*head_ref = last)

```
/* Program to move last element to front in a given linked list */
#include<stdio.h>
#include<stdlib.h>

/* A linked list node */
struct node
{
 int data;
 struct node *next;
};

/* We are using a double pointer head_ref here because we change
   head of the linked list inside this function.*/
void moveToFront(struct node **head_ref)
{
  /* If linked list is empty, or it contains only one node,
    then nothing needs to be done, simply return */
  if(*head_ref == NULL || (*head_ref)->next == NULL)
    return;

  /* Initialize second last and last pointers */
  struct node *secLast = NULL;
  struct node *last = *head_ref;

  /*After this loop secLast contains address of second last
```

```
  node and last contains address of last node in Linked List */
  while(last->next != NULL)
  {
    secLast = last;
    last = last->next;
  }

  /* Set the next of second last as NULL */
  secLast->next = NULL;

  /* Set next of last as head node */
  last->next = *head_ref;

  /* Change the head pointer to point to last node now */
  *head_ref = last;
}

/* UTILITY FUNCTIONS */
/* Function to add a node at the begining of Linked List */
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while(node != NULL)
  {
    printf("%d ", node->data);
    node = node->next;
  }
}

/* Druver program to test above function */
int main()
{
  struct node *start = NULL;

  /* The constructed linked list is:
   1->2->3->4->5 */
  push(&start, 5);
  push(&start, 4);
  push(&start, 3);
```

```
  push(&start, 2);
  push(&start, 1);

  printf("\n Linked list before moving last to front ");
  printList(start);

  moveToFront(&start);

  printf("\n Linked list after removing last to front ");
  printList(start);

  getchar();
}
```

Time Complexity: O(n) where n is the number of nodes in the given Linked List.

PAIRWISE SWAP ELEMENTS OF A GIVEN LINKED LIST

Given a singly linked list, write a function to swap elements pairwise. For example, if the linked list is 1->2->3->4->5 then the function should change it to 2->1->4->3->5, and if the linked list is 1->2->3->4->5->6 then the function should change it to 2->1->4->3->6->5.

METHOD 1 (Iterative)
Start from the head node and traverse the list. While traversing swap data of each node with its next node's data.

```
/* Program to pairwise swap elements in a given linked list */
#include<stdio.h>
#include<stdlib.h>

/* A linked list node */
struct node
{
  int data;
  struct node *next;
};

/*Function to swap two integers at addresses a and b */
void swap(int *a, int *b);

/* Function to pairwise swap elements of a linked list */
void pairWiseSwap(struct node *head)
{
  struct node *temp = head;

   /* Traverse further only if there are at-least two nodes left */
  while(temp != NULL && temp->next != NULL)
  {
    /* Swap data of node with its next node's data */
    swap(&temp->data, &temp->next->data);
```

```
    /* Move temp by 2 for the next pair */
    temp = temp->next->next;
  }
}

/* UTILITY FUNCTIONS */
/* Function to swap two integers */
void swap(int *a, int *b)
{
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}

/* Function to add a node at the begining of Linked List */
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while(node != NULL)
  {
    printf("%d ", node->data);
    node = node->next;
  }
}

/* Druver program to test above function */
int main()
{
  struct node *start = NULL;

  /* The constructed linked list is:
   1->2->3->4->5 */
  push(&start, 5);
  push(&start, 4);
  push(&start, 3);
  push(&start, 2);
  push(&start, 1);
```

```
  printf("\n Linked list before calling  pairWiseSwap() ");
  printList(start);

  pairWiseSwap(start);

  printf("\n Linked list after calling  pairWiseSwap() ");
  printList(start);

  getchar();
  return 0;
}
```

Time complexity: O(n)

METHOD 2 (Recursive)
If there are 2 or more than 2 nodes in Linked List then swap the first two nodes and recursively call for rest of the list.

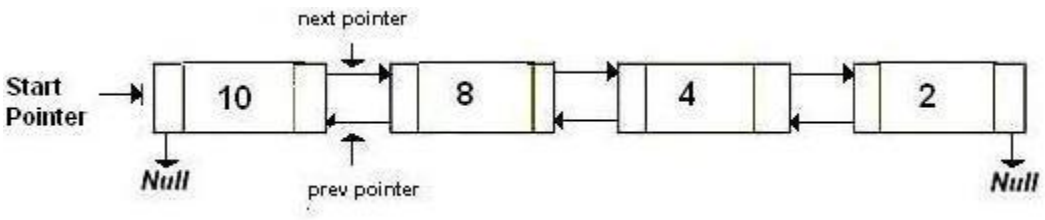
```
/* Recursive function to pairwise swap elements of a linked list */
void pairWiseSwap(struct node *head)
{
  /* There must be at-least two nodes in the list */
  if(head != NULL && head->next != NULL)
  {
    /* Swap the node's data with data of next node */
    swap(&head->data, &head->next->data);

    /* Call pairWiseSwap() for rest of the list */
    pairWiseSwap(head->next->next);
  }
}
```

Time complexity: O(n)

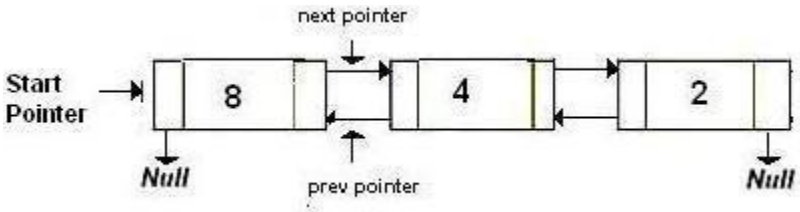
DELETE A NODE IN A DOUBLY LINKED LIST

Write a function to delete a given node in a doubly linked list.

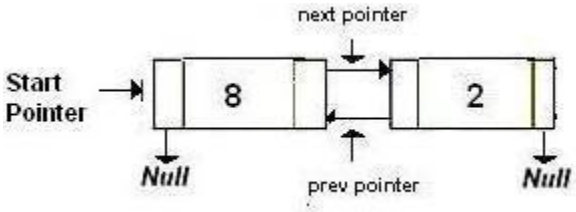
(a) Original Doubly Linked List



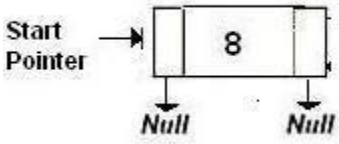
(a) After deletion of head node



(a) After deletion of middle node



(a) After deletion of last node



Algorithm
Let the node to be deleted is *del*.
1) If node to be deleted is head node, then change the head pointer to next current head.
2) Set *next* of previous to *del*, if previous to *del* exixts.
3) Set *prev* of next to *del*, if next to *del* exixts.

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

/* a node of the doubly linked list */
struct node
{
  int data;
  struct node *next;
  struct node *prev;
};

/* Function to delete a node in a Doubly Linked List.
   head_ref --> pointer to head node pointer.
   del  -->  pointer to node to be deleted. */
void deleteNode(struct node **head_ref, struct node *del)
{
```

```
  /* base case */
  if(*head_ref == NULL || del == NULL)
    return;

  /* If node to be deleted is head node */
  if(*head_ref == del)
    *head_ref = del->next;

  /* Change next only if node to be deleted is NOT the last node */
  if(del->next != NULL)
    del->next->prev = del->prev;

  /* Change prev only if node to be deleted is NOT the first node */
  if(del->prev != NULL)
    del->prev->next = del->next;

  /* Finally, free the memory occupied by del*/
  free(del);
  return;
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginning of the Doubly Linked List */
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
       (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;

  /* since we are adding at the begining,
    prev is always NULL */
  new_node->prev = NULL;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* change prev of head node to new node */
  if((*head_ref) !=  NULL)
   (*head_ref)->prev = new_node ;

  /* move the head to point to the new node */
   (*head_ref)    = new_node;
}

/* Function to print nodes in a given doubly linked list
   This function is same as printList() of singly linked lsit */
void printList(struct node *node)
{
  while(node!=NULL)
  {
   printf("%d ", node->data);
```

```
   node = node->next;
  }
}

/* Drier program to test above functions*/
int main()
{
  /* Start with the empty list */
  struct node* head = NULL;

  /* Let us create the doubly linked list 10<->8<->4<->2 */
  push(&head, 2);
  push(&head, 4);
  push(&head, 8);
  push(&head, 10);

  printf("\n Original Linked list ");
  printList(head);

  /* delete nodes from the doubly linked list */
  deleteNode(&head, head);  /*delete first node*/
  deleteNode(&head, head->next);  /*delete middle node*/
  deleteNode(&head, head->next);  /*delete last node*/

  /* Modified linked list will be NULL<-8->NULL */
  printf("\n Modified Linked list ");
  printList(head);

  getchar();
}
```

Time Complexity: O(1)
Time Complexity: O(1)

INTERSECTION OF TWO SORTED LINKED LISTS

Given two lists sorted in increasing order, create and return a new list representing the intersection of the two lists. The new list should be made with its own memory — the original lists should not be changed.

For example, let the first linked list be 1->2->3->4->6 and second linked list be 2->4->6->8, then your function should create and return a third list as 2->4->6.

Method 1 (Using Dummy Node)
The strategy here uses a temporary dummy node as the start of the result list. The pointer tail always points to the last node in the result list, so appending new nodes is easy. The dummy node gives tail something to point to initially when the result list is empty. This dummy node is efficient, since it is only temporary, and it is allocated in the stack. The loop proceeds, removing one node from either ‘a’ or ‘b’, and adding it to tail. When we are done, the result is in dummy.next.

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

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

void push(struct node** head_ref, int new_data);

/*This solution uses the temporary dummy to build up the result list */
struct node* sortedIntersect(struct node* a, struct node* b)
{
  struct node dummy;
  struct node* tail = &dummy;
  dummy.next = NULL;

  /* Once one or the other list runs out -- we're done */
  while (a != NULL && b != NULL)
  {
    if(a->data == b->data)
    {
       push((&tail->next), a->data);
       tail = tail->next;
       a = a->next;
       b = b->next;
    }
    else if (a->data < b->data)
    {
       /* advance the smaller list */
       a = a->next;
    }
    else
    {
       b = b->next;
    }
  }
  return(dummy.next);
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginging of the linked list */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
```

```
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while(node != NULL)
  {
   printf("%d ", node->data);
   node = node->next;
  }
}

/* Drier program to test above functions*/
int main()
{
  /* Start with the empty lists */
  struct node* a = NULL;
  struct node* b = NULL;
  struct node *intersect = NULL;

  /* Let us create the first sorted linked list to test the functions
   Created linked list will be 1->2->3->4->5->6 */
  push(&a, 6);
  push(&a, 5);
  push(&a, 4);
  push(&a, 3);
  push(&a, 2);
  push(&a, 1);

  /* Let us create the second sorted linked list
   Created linked list will be 2->4->6->8 */
  push(&b, 8);
  push(&b, 6);
  push(&b, 4);
  push(&b, 2);

  /* Find the intersection two linked lists */
  intersect = sortedIntersect(a, b);

  printf("\n Linked list containing common items of a & b \n ");
  printList(intersect);

  getchar();
}
```

Time Complexity: O(n) where n is the number of nodes in the smaller list.

Method 2 (Using Local References)
This solution is structurally very similar to the above, but it avoids using a dummy node Instead, it maintains a struct node** pointer, lastPtrRef, that always points to the last pointer of the result list. This solves the same case that the dummy node did — dealing with the result list when it is empty. If you are trying to build up a list at its tail, either the dummy node or the struct node** "reference" strategy can be used

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

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

void push(struct node** head_ref, int new_data);

/* This solution uses the local reference */
struct node* sortedIntersect(struct node* a, struct node* b)
{
  struct node* result = NULL;
  struct node** lastPtrRef = &result;

  /* Advance comparing the first nodes in both lists.
    When one or the other list runs out, we're done. */
  while (a!=NULL && b!=NULL)
  {
    if(a->data == b->data)
    {
      /* found a node for the intersection */
      push(lastPtrRef, a->data);
      lastPtrRef = &((*lastPtrRef)->next);
      a = a->next;
      b = b->next;
    }
    else if (a->data < b->data)
    {
      /* advance the smaller list */
      a=a->next;
    }
    else
    {
      b=b->next;
    }
  }
  return(result);
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginging of the linked list */
void push(struct node** head_ref, int new_data)
```

```
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while(node != NULL)
  {
   printf("%d ", node->data);
   node = node->next;
  }
}

/* Drier program to test above functions*/
int main()
{
  /* Start with the empty lists */
  struct node* a = NULL;
  struct node* b = NULL;
  struct node *intersect = NULL;

  /* Let us create the first sorted linked list to test the functions
   Created linked list will be 1->2->3->4->5->6 */
  push(&a, 6);
  push(&a, 5);
  push(&a, 4);
  push(&a, 3);
  push(&a, 2);
  push(&a, 1);

  /* Let us create the second sorted linked list
   Created linked list will be 2->4->6->8 */
  push(&b, 8);
  push(&b, 6);
  push(&b, 4);
  push(&b, 2);

  /* Find the intersection two linked lists */
  intersect = sortedIntersect(a, b);

  printf("\n Linked list containing common items of a & b \n ");
  printList(intersect);
```

```
  getchar();
}
```

Time Complexity: O(n) where n is the number of nodes in the smaller list.

Method 3 (Recursive)
Below is the recursive implementation of sortedIntersect(). The method expects three parameters. The initial value passed as third parameter must be NULL.

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

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

struct node *sortedIntersect(struct node *a, struct node *b,
                                        struct node *result)
{
  /* base case */
  if(a == NULL || b == NULL)
  {
    return NULL;
  }

  /* If both lists are non-empty */

  /* advance the smaller list and call recursively */
  if(a->data < b->data)
  {
    return sortedIntersect(a->next, b, result);
  }
  else if(a->data > b->data)
  {
    return sortedIntersect(a, b->next, result);
  }
  else if(a->data == b->data)
  {
    /* If same data is found then allocate memory */
    struct node *temp = (struct node *)malloc(sizeof(struct node));
    temp->data = a->data;

    /* If the first node is being added to resultant list */
    if(result == NULL)
    {
      result = temp;
```

```
    }

    /* Else change the next of result and move result to next */
    else
    {
      result->next = temp;
      result = temp;
    }

    /* advance both lists and call recursively */
    result->next = sortedIntersect(a->next, b->next, result);
  }

  return result;
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginging of the linked list */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while(node != NULL)
  {
   printf("%d ", node->data);
   node = node->next;
  }
}

/* Drier program to test above functions*/
int main()
{
  /* Start with the empty lists */
  struct node* a = NULL;
  struct node* b = NULL;
  struct node *intersect = NULL;

  /* Let us create the first sorted linked list to test the functions
   Created linked list will be 1->2->3->4->5->6 */
  push(&a, 6);
```

```
  push(&a, 5);
  push(&a, 4);
  push(&a, 3);
  push(&a, 2);
  push(&a, 1);

  /* Let us create the second sorted linked list
   Created linked list will be 2->4->6->8 */
  push(&b, 8);
  push(&b, 6);
  push(&b, 4);
  push(&b, 2);

  /* Find the intersection two linked lists */
  intersect = sortedIntersect(a, b, NULL);

  printf("\n Linked list containing common items of a & b \n ");
  printList(intersect);

  getchar();
}
```

Time Complexity: O(n) where n is the number of nodes in the smaller list.

DELETE ALTERNATE NODES OF A LINKED LIST

Given a Singly Linked List, starting from the second node delete all alternate nodes of it. For example, if the given linked list is 1->2->3->4->5 then your function should convert it to 1->3->5, and if the given linked list is 1->2->3->4 then convert it to 1->3.

Method 1 (Iterative)
Keep track of previous of the node to be deleted. First change the next link of previous node and then free the memory allocated for the node.

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

/* A linked list node */
struct node
{
  int data;
  struct node *next;
};

/* deletes alternate nodes of a list starting with head */
void deleteAlt(struct node *head)
{
  if (head == NULL)
    return;
```

```
   /* Initialize prev and node to be deleted */
  struct node *prev = head;
  struct node *node = head->next;

  while (prev != NULL && node != NULL)
  {
     /* Change next link of previous node */
     prev->next = node->next;

     /* Free memory */
     free(node);

     /* Update prev and node */
     prev = prev->next;
     if(prev != NULL)
        node = prev->next;
  }
}

/* UTILITY FUNCTIONS TO TEST fun1() and fun2() */
/* Given a reference (pointer to pointer) to the head
  of a list and an int, push a new node on the front
  of the list. */
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
          (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while(node != NULL)
  {
    printf("%d ", node->data);
    node = node->next;
  }
}

/* Drier program to test above functions */
int main()
{
  int arr[100];

  /* Start with the empty list */
```

```
  struct node* head = NULL;

  /* Using push() to construct below list
    1->2->3->4->5  */
  push(&head, 5);
  push(&head, 4);
  push(&head, 3);
  push(&head, 2);
  push(&head, 1);

  printf("\n List before calling deleteAlt() ");
  printList(head);

  deleteAlt(head);

  printf("\n List after calling deleteAlt() ");
  printList(head);

  getchar();
  return 0;
}
```

Time Complexity: O(n) where n is the number of nodes in the given Linked List.

Method 2 (Recursive)
Recursive code uses the same approach as method 1. The recursive coed is simple and short, but causes O(n) recursive function calls for a linked list of size n.

?

```
/* deletes alternate nodes of a list starting with head */
void deleteAlt(struct node *head)
{
  if (head == NULL)
    return;

  struct node *node = head->next;

  if (node == NULL)
    return;

  /* Change the next link of head */
  head->next = node->next;

  /* free memory allocated for node */
  free(node);

  /* Recursively call for the new next of head */
  deleteAlt(head->next);
}
```

Time Complexity: O(n)

ALTERNATING SPLIT OF A GIVEN SINGLY LINKED LIST

Write a function AlternatingSplit() that takes one list and divides up its nodes to make two smaller lists ‘a’ and ‘b’. The sublists should be made from alternating elements in the original list. So if the original list is 0->1->0->1->0->1 then one sublist should be 0->0->0 and the other should be 1->1->1.

Method 1(Simple)
The simplest approach iterates over the source list and pull nodes off the source and alternately put them at the front (or beginning) of ‘a’ and b’. The only strange part is that the nodes will be in the reverse order that they occurred in the source list. Method 2 inserts the node at the end by keeping track of last node in sublists.

?
```
/*Program to alternatively split a linked list into two halves */
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>

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

/* pull off the front node of the source and put it in dest */
void MoveNode(struct node** destRef, struct node** sourceRef) ;

/* Given the source list, split its nodes into two shorter lists.
  If we number the elements 0, 1, 2, ... then all the even elements
  should go in the first list, and all the odd elements in the second.
  The elements in the new lists may be in any order. */
void AlternatingSplit(struct node* source, struct node** aRef,
                            struct node** bRef)
{
  /* split the nodes of source to these 'a' and 'b' lists */
  struct node* a = NULL;
  struct node* b = NULL;

  struct node* current = source;
  while (current != NULL)
  {
    MoveNode(&a, &current); /* Move a node to list 'a' */
    if (current != NULL)
    {
       MoveNode(&b, &current); /* Move a node to list 'b' */
    }
  }
  *aRef = a;
  *bRef = b;
}
```

```
/* Take the node from the front of the source, and move it to the front of
the dest.
   It is an error to call this with the source list empty.

   Before calling MoveNode():
   source == {1, 2, 3}
   dest == {1, 2, 3}

   Affter calling MoveNode():
   source == {2, 3}
   dest == {1, 1, 2, 3}
*/
void MoveNode(struct node** destRef, struct node** sourceRef)
{
  /* the front source node  */
  struct node* newNode = *sourceRef;
  assert(newNode != NULL);

  /* Advance the source pointer */
  *sourceRef = newNode->next;

  /* Link the old dest off the new node */
  newNode->next = *destRef;

  /* Move dest to point to the new node */
  *destRef = newNode;
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginging of the linked list */
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while(node!=NULL)
  {
   printf("%d ", node->data);
   node = node->next;
```

```
  }
}

/* Drier program to test above functions*/
int main()
{
  /* Start with the empty list */
  struct node* head = NULL;
  struct node* a = NULL;
  struct node* b = NULL;

  /* Let us create a sorted linked list to test the functions
   Created linked list will be 0->1->2->3->4->5 */
  push(&head, 5);
  push(&head, 4);
  push(&head, 3);
  push(&head, 2);
  push(&head, 1);
  push(&head, 0);

  printf("\n Original linked List:  ");
  printList(head);

  /* Remove duplicates from linked list */
  AlternatingSplit(head, &a, &b);

  printf("\n Resultant Linked List 'a' ");
  printList(a);

  printf("\n Resultant Linked List 'b' ");
  printList(b);

  getchar();
  return 0;
}
```

Time Complexity: O(n) where n is number of node in the given linked list.

Method 2(Using Dummy Nodes)
Here is an alternative approach which builds the sub-lists in the same order as the source list. The code uses a temporary dummy header nodes for the ‘a’ and ‘b’ lists as they are being built. Each sublist has a “tail” pointer which points to its current last node — that way new nodes can be appended to the end of each list easily. The dummy nodes give the tail pointers something to point to initially. The dummy nodes are efficient in this case because they are temporary and allocated in the stack. Alternately, local “reference pointers” (which always points to the last pointer in the list instead of to the last node) could be used to avoid Dummy nodes.

?

```
void AlternatingSplit(struct node* source, struct node** aRef,
                      struct node** bRef)
{
```

```
  struct node aDummy;
  struct node* aTail = &aDummy; /* points to the last node in 'a' */
  struct node bDummy;
  struct node* bTail = &bDummy; /* points to the last node in 'b' */
  struct node* current = source;
  aDummy.next = NULL;
  bDummy.next = NULL;
  while (current != NULL)
  {
    MoveNode(&(aTail->next), &current); /* add at 'a' tail */
    aTail = aTail->next; /* advance the 'a' tail */
    if (current != NULL)
    {
      MoveNode(&(bTail->next), &current);
      bTail = bTail->next;
    }
  }
  *aRef = aDummy.next;
  *bRef = bDummy.next;
}
```

Time Complexity: O(n) where n is number of node in the given linked list.

MERGE TWO SORTED LINKED LISTS

Write a SortedMerge() function that takes two lists, each of which is sorted in increasing order, and merges the two together into one list which is in increasing order. SortedMerge() should return the new list. The new list should be made by splicing
together the nodes of the first two lists.

For example if the first linked list a is 5->10->15 and the other linked list b is 2->3->20, then SortedMerge() should return a pointer to the head node of the merged list 2->3->5->10->15->20.

There are many cases to deal with: either ‘a’ or ‘b’ may be empty, during processing either ‘a’ or ‘b’ may run out first, and finally there’s the problem of starting the result list empty, and building it up while going through ‘a’ and ‘b’.

Method 1 (Using Dummy Nodes)
The strategy here uses a temporary dummy node as the start of the result list. The pointer Tail always points to the last node in the result list, so appending new nodes is easy.
The dummy node gives tail something to point to initially when the result list is empty. This dummy node is efficient, since it is only temporary, and it is allocated in the stack. The loop proceeds, removing one node from either ‘a’ or ‘b’, and adding it to tail. When
we are done, the result is in dummy.next.

```
/*Program to alternatively split a linked list into two halves */
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>
```

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

/* pull off the front node of the source and put it in dest */
void MoveNode(struct node** destRef, struct node** sourceRef);

/* Takes two lists sorted in increasing order, and splices their nodes
together to make one big sorted list which is returned.  */
struct node* SortedMerge(struct node* a, struct node* b)
{
   /* a dummy first node to hang the result on */
   struct node dummy;

   /* tail points to the last result node  */
   struct node* tail = &dummy;

   /* so tail->next is the place to add new nodes
     to the result. */
   dummy.next = NULL;
   while(1)
   {
      if(a == NULL)
      {
         /* if either list runs out, use the other list */
         tail->next = b;
         break;
      }
      else if (b == NULL)
      {
         tail->next = a;
         break;
      }
      if (a->data <= b->data)
      {
         MoveNode(&(tail->next), &a);
      }
      else
     {
        MoveNode(&(tail->next), &b);
     }
     tail = tail->next;
  }
  return(dummy.next);
}

/* UTILITY FUNCTIONS */
/*MoveNode() function takes the node from the front of the source, and move
it to the front of the dest.
   It is an error to call this with the source list empty.

   Before calling MoveNode():
```

```
   source == {1, 2, 3}
   dest == {1, 2, 3}

   Affter calling MoveNode():
   source == {2, 3}
   dest == {1, 1, 2, 3}
*/
void MoveNode(struct node** destRef, struct node** sourceRef)
{
  /* the front source node  */
  struct node* newNode = *sourceRef;
  assert(newNode != NULL);

  /* Advance the source pointer */
  *sourceRef = newNode->next;

  /* Link the old dest off the new node */
  newNode->next = *destRef;

  /* Move dest to point to the new node */
  *destRef = newNode;
}

/* Function to insert a node at the beginging of the linked list */
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while(node!=NULL)
  {
   printf("%d ", node->data);
   node = node->next;
  }
}

/* Drier program to test above functions*/
int main()
{
  /* Start with the empty list */
  struct node* res = NULL;
```

```
  struct node* a = NULL;
  struct node* b = NULL;

  /* Let us create two sorted linked lists to test the functions
   Created lists shall be a: 5->10->15,  b: 2->3->20 */
  push(&a, 15);
  push(&a, 10);
  push(&a, 5);

  push(&b, 20);
  push(&b, 3);
  push(&b, 2);

  /* Remove duplicates from linked list */
  res = SortedMerge(a, b);

  printf("\n Merged Linked List is: \n");
  printList(res);

  getchar();
  return 0;
}
```

Method 2 (Using Local References)
This solution is structurally very similar to the above, but it avoids using a dummy node. Instead, it maintains a struct node** pointer, lastPtrRef, that always points to the last pointer of the result list. This solves the same case that the dummy node did — dealing with the result list when it is empty. If you are trying to build up a list at its tail, either the dummy node or the struct node** "reference" strategy can be used (see Section 1 for details).

```
struct node* SortedMerge(struct node* a, struct node* b)
{
  struct node* result = NULL;

  /* point to the last result pointer */
  struct node** lastPtrRef = &result;

  while(1)
  {
    if (a == NULL)
    {
      *lastPtrRef = b;
       break;
    }
    else if (b==NULL)
    {
       *lastPtrRef = a;
       break;
    }
    if(a->data <= b->data)
    {
```

```
      MoveNode(lastPtrRef, &a);
    }
    else
    {
      MoveNode(lastPtrRef, &b);
    }

    /* tricky: advance to point to the next ".next" field */
    lastPtrRef = &((*lastPtrRef)->next);
  }
  return(result);
}
```

Method 3 (Using Recursion)
Merge is one of those nice recursive problems where the recursive solution code is much cleaner than the iterative code. You probably wouldn't want to use the recursive version for production code however, because it will use stack space which is proportional to the length of the lists.

```
struct node* SortedMerge(struct node* a, struct node* b)
{
  struct node* result = NULL;

  /* Base cases */
  if (a == NULL)
     return(b);
  else if (b==NULL)
     return(a);

  /* Pick either a or b, and recur */
  if (a->data <= b->data)
  {
     result = a;
     result->next = SortedMerge(a->next, b);
  }
  else
  {
     result = b;
     result->next = SortedMerge(a, b->next);
  }
  return(result);
}
```

IDENTICAL LINKED LISTS

Two Linked Lists are identical when they have same data and arrangement of data is also same. For example Linked lists a (1->2->3) and b(1->2->3) are identical. . Write a function to check if the given two linked lists are identical.

Method 1 (Iterative)
To identify if two lists are identical, we need to traverse both lists simultaneously, and while traversing we need to compare data.

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

/* Structure for a linked list node */
struct node
{
  int data;
  struct node *next;
};

/* returns 1 if linked lists a and b are identical, otherwise 0 */
bool areIdentical(struct node *a, struct node *b)
{
  while(1)
  {
    /* base case */
    if(a == NULL && b == NULL)
    {  return 1; }
    if(a == NULL && b != NULL)
    {  return 0; }
    if(a != NULL && b == NULL)
    {  return 0; }
    if(a->data != b->data)
    {  return 0; }

    /* If we reach here, then a and b are not NULL and their
       data is same, so move to next nodes in both lists */
    a = a->next;
    b = b->next;
  }
}

/* UTILITY FUNCTIONS TO TEST fun1() and fun2() */
/* Given a reference (pointer to pointer) to the head
  of a list and an int, push a new node on the front
  of the list. */
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
          (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
```

```
  (*head_ref)    = new_node;
}

/* Druver program to test above function */
int main()
{
  struct node *a = NULL;
  struct node *b = NULL;

  /* The constructed linked lists are :
   a: 3->2->1
   b: 3->2->1 */
  push(&a, 1);
  push(&a, 2);
  push(&a, 3);

  push(&b, 1);
  push(&b, 2);
  push(&b, 3);

  if(areIdentical(a, b) == 1)
    printf(" Linked Lists are identical ");
  else
    printf(" Linked Lists are not identical ");

  getchar();
  return 0;
}
```

Method 2 (Recursive)

Recursive solution code is much cleaner than the iterative code. You probably wouldn't want to use the recursive version for production code however, because it will use stack space which is proportional to the length of the lists

```
bool areIdentical(struct node *a, struct node *b)
{
  if (a == NULL && b == NULL)
  {  return 1;  }
  if (a == NULL && b != NULL)
  {  return 0;  }
  if (a != NULL && b == NULL)
  {  return 0;  }
  if (a->data != b->data)
  {  return 0;  }

  /* If we reach here, then a and b are not NULL and their
       data is same, so move to next nodes in both lists */
  return areIdentical(a->next, b->next);
}
```

Time Complexity: O(n) for both iterative and recursive versions. n is the length of the smaller list among a and b.

REVERSE ALTERNATE K NODES IN A SINGLY LINKED LIST

Given a linked list, write a function to reverse every alternate k nodes (where k is an input to the function) in an efficient way. Give the complexity of your algorithm.

```
Example:
Inputs:   1->2->3->4->5->6->7->8->9->NULL and k = 3
Output:   3->2->1->4->5->6->9->8->7->NULL.
```

Method 1 (Process 2k nodes and recursively call for rest of the list)
This method is basically an extension of the method discussed in this post.

```
kAltReverse(struct node *head, int k)
  1)  Reverse first k nodes.
  2)  In the modified list head points to the kth node.  So change next
       of head to (k+1)th node
  3)  Move the current pointer to skip next k nodes.
  4)  Call the kAltReverse() recursively for rest of the n - 2k nodes.
  5)  Return new head of the list.
```

?

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

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

/* Reverses alternate k nodes and
   returns the pointer to the new head node */
struct node *kAltReverse(struct node *head, int k)
{
    struct node* current = head;
    struct node* next;
    struct node* prev = NULL;
    int count = 0;

    /*1) reverse first k nodes of the linked list */
    while (current != NULL && count < k)
    {
       next  = current->next;
       current->next = prev;
       prev = current;
       current = next;
       count++;
    }
```

```
    /* 2) Now head points to the kth node.  So change next
       of head to (k+1)th node*/
    if(head != NULL)
      head->next = current;

    /* 3) We do not want to reverse next k nodes. So move the current
        pointer to skip next k nodes */
    count = 0;
    while(count < k-1 && current != NULL )
    {
      current = current->next;
      count++;
    }

    /* 4) Recursively call for the list starting from current->next.
       And make rest of the list as next of first node */
    if(current !=  NULL)
       current->next = kAltReverse(current->next, k);

    /* 5) prev is new head of the input list */
    return prev;
}

/* UTILITY FUNCTIONS */
/* Function to push a node */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print linked list */
void printList(struct node *node)
{
    int count = 0;
    while(node != NULL)
    {
        printf("%d  ", node->data);
        node = node->next;
        count++;
    }
}

/* Drier program to test above function*/
int main(void)
```

```
{
    /* Start with the empty list */
    struct node* head = NULL;

    // create a list 1->2->3->4->5...... ->20
    for(int i = 20; i > 0; i--)
      push(&head, i);

     printf("\n Given linked list \n");
     printList(head);
     head = kAltReverse(head, 3);

     printf("\n Modified Linked list \n");
     printList(head);

     getchar();
     return(0);
}
```

Output:
Given linked list
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Modified Linked list
3 2 1 4 5 6 9 8 7 10 11 12 15 14 13 16 17 18 20 19

Time Complexity: O(n)

Method 2 (Process k nodes and recursively call for rest of the list)
The method 1 reverses the first k node and then moves the pointer to k nodes ahead. So method 1 uses two while loops and processes 2k nodes in one recursive call.
This method processes only k nodes in a recursive call. It uses a third bool parameter b which decides whether to reverse the k elements or simply move the pointer.

```
_kAltReverse(struct node *head, int k, bool b)
  1)  If b is true, then reverse first k nodes.
  2)  If b is false, then move the pointer k nodes ahead.
  3)  Call the kAltReverse() recursively for rest of the n - k nodes and link
       rest of the modified list with end of first k nodes.
  4)  Return new head of the list.
```

?

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

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

/* Helper function for kAltReverse() */
```

```
struct node * _kAltReverse(struct node *node, int k, bool b);

/* Alternatively reverses the given linked list in groups of
   given size k. */
struct node *kAltReverse(struct node *head, int k)
{
  return _kAltReverse(head, k, true);
}

/*  Helper function for kAltReverse().  It reverses k nodes of the list only
if
    the third parameter b is passed as true, otherwise moves the pointer k
    nodes ahead and recursively calls iteself  */
struct node * _kAltReverse(struct node *node, int k, bool b)
{
   if(node == NULL)
       return NULL;

   int count = 1;
   struct node *prev = NULL;
   struct node  *current = node;
   struct node *next;

   /* The loop serves two purposes
      1) If b is true, then it reverses the k nodes
      2) If b is false, then it moves the current pointer */
   while(current != NULL && count <= k)
   {
       next = current->next;

       /* Reverse the nodes only if b is true*/
       if(b == true)
          current->next = prev;

       prev = current;
       current = next;
       count++;
   }

   /* 3) If b is true, then node is the kth node.
       So attach rest of the list after node.
     4) After attaching, return the new head */
   if(b == true)
   {
        node->next = _kAltReverse(current,k,!b);
        return prev;
   }

   /* If b is not true, then attach rest of the list after prev.
     So attach rest of the list after prev */
   else
   {
        prev->next = _kAltReverse(current, k, !b);
        return node;
   }
```

```
}

/* UTILITY FUNCTIONS */
/* Function to push a node */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print linked list */
void printList(struct node *node)
{
    int count = 0;
    while(node != NULL)
    {
        printf("%d  ", node->data);
        node = node->next;
        count++;
    }
}

/* Drier program to test above function*/
int main(void)
{
    /* Start with the empty list */
    struct node* head = NULL;
    int i;

    // create a list 1->2->3->4->5...... ->20
    for(i = 20; i > 0; i--)
      push(&head, i);

    printf("\n Given linked list \n");
    printList(head);
    head = kAltReverse(head, 3);

    printf("\n Modified Linked list \n");
    printList(head);

    getchar();
    return(0);
}
```

Output:
Given linked list
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Modified Linked list
3 2 1 4 5 6 9 8 7 10 11 12 15 14 13 16 17 18 20 19

Time Complexity: O(n)

DELETE NODES WHICH HAVE A GREATER VALUE ON RIGHT SIDE

Given a singly linked list, remove all the nodes which have a greater value on right side.

Examples:
a) The list 12->15->10->11->5->6->2->3->NULL should be changed to 15->11->6->3->NULL. Note that 12, 10, 5 and 2 have been deleted because there is a greater value on the right side.

When we examine 12, we see that after 12 there is one node with value greater than 12 (i.e. 15), so we delete 12.
When we examine 15, we find no node after 15 that has value greater than 15 so we keep this node.
When we go like this, we get 15->6->3

b) The list 10->20->30->40->50->60->NULL should be changed to 60->NULL. Note that 10, 20, 30, 40 and 50 have been deleted because they all have a greater value on the right side.

c) The list 60->50->40->30->20->10->NULL should not be changed.

Method 1 (Simple)
Use two loops. In the outer loop, pick nodes of the linked list one by one. In the inner loop, check if there exist a node whose value is greater than the picked node. If there exists a node whose value is greater, then delete the picked node.
Time Complexity: O(n^2)

Method 2 (Use Reverse)
Thanks to Paras for providing the below algorithm.
1. Reverse the list.
2. Traverse the reversed list. Keep max till now. If next node < max, then delete the next node, otherwise max = next node.
3. Reverse the list again to retain the original order.

Time Complexity: O(n)

Thanks to R.Srinivasan for providing below code.

?
```
#include <stdio.h>
```

```
#include <stdlib.h>

/* structure of a linked list node */
struct node
{
     int data;
     struct node *next;
};

/* prototype for utility functions */
void reverseList(struct node **headref);
void _delLesserNodes(struct node *head);

/* Deletes nodes which have a node with greater value node
  on left side */
void delLesserNodes(struct node **head_ref)
{
    /* 1) Reverse the linked list */
    reverseList(head_ref);

    /* 2) In the reversed list, delete nodes which have a node
       with greater value node on left side. Note that head
       node is never deleted because it is the leftmost node.*/
    _delLesserNodes(*head_ref);

    /* 3) Reverse the linked list again to retain the
       original order */
    reverseList(head_ref);
}

/* Deletes nodes which have greater value node(s) on left side */
void _delLesserNodes(struct node *head)
{
     struct node *current = head;

     /* Initialize max */
     struct node *maxnode = head;
     struct node *temp;

     while (current != NULL && current->next != NULL)
     {
         /* If current is smaller than max, then delete current */
         if(current->next->data < maxnode->data)
         {
             temp = current->next;
             current->next = temp->next;
             free(temp);
         }

         /* If current is greater than max, then update max and
            move current */
         else
         {
             current = current->next;
             maxnode = current;
```

```
          }

     }
}

/* Utility function to insert a node at the begining */
void push(struct node **head_ref, int new_data)
{
     struct node *new_node =
               (struct node *)malloc(sizeof(struct node));
     new_node->data = new_data;
     new_node->next = *head_ref;
     *head_ref = new_node;
}

/* Utility function to reverse a linked list */
void reverseList(struct node **headref)
{
     struct node *current = *headref;
     struct node *prev = NULL;
     struct node *next;
     while(current != NULL)
     {
          next = current->next;
          current->next = prev;
          prev = current;
          current = next;
     }
     *headref = prev;
}

/* Utility function to print a linked list */
void printList(struct node *head)
{
     while(head!=NULL)
     {
        printf("%d ",head->data);
        head=head->next;
     }
     printf("\n");
}

/* Driver program to test above functions */
int main()
{
    struct node *head = NULL;

    /* Create following linked list
      12->15->10->11->5->6->2->3 */
    push(&head,3);
    push(&head,2);
    push(&head,6);
    push(&head,5);
    push(&head,11);
    push(&head,10);
```

```
    push(&head,15);
    push(&head,12);

    printf("Given Linked List: ");
    printList(head);

    delLesserNodes(&head);

    printf("\nModified Linked List: ");
    printList(head);

    getchar();
    return 0;
}
```

Output:

```
Given Linked List: 12 15 10 11 5 6 2 3
Modified Linked List: 15 11 6 3
```

SEGREGATE EVEN AND ODD NODES IN A LINKED LIST

Given a Linked List of integers, write a function to modify the linked list such that all even numbers appear before all the odd numbers in the modified linked list. Also, keep the order of even and odd numbers same.

Examples:
Input: 17->15->8->12->10->5->4->1->7->6->NULL
Output: 8->12->10->4->6->17->15->5->1->7->NULL

Input: 8->12->10->5->4->1->6->NULL
Output: 8->12->10->4->6->5->1->NULL

// If all numbers are even then do not change the list
Input: 8->12->10->NULL
Output: 8->12->10->NULL

// If all numbers are odd then do not change the list
Input: 1->3->5->7->NULL
Output: 1->3->5->7->NULL

Method 1
The idea is to get pointer to the last node of list. And then traverse the list starting from the head node and move the odd valued nodes from their current position to end of the list.

Thanks to blunderboy for suggesting this method.

Algorithm:
…1) Get pointer to the last node.
…2) Move all the odd nodes to the end.
……..a) Consider all odd nodes before the first even node and move them to end.
……..b) Change the head pointer to point to the first even node.
……..b) Consider all odd nodes after the first even node and move them to the end.

?

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

/* a node of the singly linked list */
struct node
{
  int data;
  struct node *next;
};

void segregateEvenOdd(struct node **head_ref)
{
  struct node *end = *head_ref;

  struct node *prev = NULL;
  struct node *curr = *head_ref;

  /* Get pointer to the last node */
  while(end->next != NULL)
       end = end->next;

  struct node *new_end = end;

  /* Consider all odd nodes before the first even node
     and move then after end */
  while(curr->data %2 != 0 && curr != end)
  {
    new_end->next = curr;
    curr = curr->next;
    new_end->next->next = NULL;
    new_end = new_end->next;
  }

  // 10->8->17->17->15
  /* Do following steps only if there is any even node */
  if (curr->data%2 == 0)
  {
    /* Change the head pointer to point to first even node */
    *head_ref = curr;

    /* now current points to the first even node */
    while(curr != end)
    {
      if ( (curr->data)%2 == 0 )
      {
```

```
            prev = curr;
            curr = curr->next;
        }
        else
        {
            /* break the link between prev and current */
            prev->next = curr->next;

             /* Make next of curr as NULL  */
            curr->next = NULL;

            /* Move curr to end */
            new_end->next = curr;

            /* make curr as new end of list */
            new_end = curr;

            /* Update current pointer to next of the moved node */
            curr = prev->next;
        }
      }
    }

    /* We must have prev set before executing lines following this
       statement */
    else
      prev = curr;

    /* If the end of the original list is odd then move this node to
      end to maintain same order of odd numbers in modified list */
    if((end->data)%2 != 0)
    {
        prev->next = end->next;
        end->next = NULL;
        new_end->next = end;
    }
    return;
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginning of the Doubly Linked List */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
         (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
     (*head_ref)    = new_node;
```

```
}

/* Function to print nodes in a given doubly linked list
   This function is same as printList() of singly linked lsit */
void printList(struct node *node)
{
  while(node!=NULL)
  {
   printf("%d ", node->data);
   node = node->next;
  }
}

/* Drier program to test above functions*/
int main()
{
  /* Start with the empty list */
  struct node* head = NULL;

  /* Let us create a sample linked list as following
    17->15->8->12->10->5->4->1->7->6 */
  push(&head, 6);
  push(&head, 7);
  push(&head, 1);
  push(&head, 4);
  push(&head, 5);
  push(&head, 10);
  push(&head, 12);
  push(&head, 8);
  push(&head, 15);
  push(&head, 17);

  printf("\n Original Linked list ");
  printList(head);

  segregateEvenOdd(&head);

  printf("\n Modified Linked list ");
  printList(head);

  getchar();
  return 0;
}
```

Output:

```
 Original Linked list 17 15 8 12 10 5 4 1 7 6
 Modified Linked list 8 12 10 4 6 17 15 5 1 7
```

Time complexity: O(n)

Method 2

The idea is to split the linked list into two: one containing all even nodes and other 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 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 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 last pointer in the odd node list.

Time complexity: O(n)

REVERSE A LINKED LIST IN GROUPS OF GIVEN SIZE

Given a linked list, write a function to reverse every k nodes (where k is an input to the function).

```
Example:
Inputs:  1->2->3->4->5->6->7->8->NULL and k = 3
Output:  3->2->1->6->5->4->8->7->NULL.

Inputs:   1->2->3->4->5->6->7->80->NULL and k = 5
Output:  5->4->3->2->1->8->7->6->NULL.
```

Algorithm: *reverse(head, k)*
1) Reverse the first sub-list of size k. While reversing keep track of the next node and previous node. Let the pointer to the next node be *next* and pointer to the previous node be *prev*. See this post for reversing a linked list.
2) *head->next = reverse(next, k)* /* Recursively call for rest of the list and link the two sub-lists */
3) return *prev* /* *prev* becomes the new head of the list (see the diagrams of iterative method of this post) */

?

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

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

/* Reverses the linked list in groups of size k and returns the pointer to
the new head node */
struct node *reverse (struct node *head, int k)
{
    struct node* current = head;
    struct node* next;
    struct node* prev = NULL;
    int count = 0;
```

```
    /*reverse first k nodes of the linked list */
    while (current != NULL && count < k)
    {
       next  = current->next;
       current->next = prev;
       prev = current;
       current = next;
       count++;
    }

    /* next is now a pointer to (k+1)th node
       Recursively call for the list starting from current.
       And make rest of the list as next of first node */
    if(next !=  NULL)
    {  head->next = reverse(next, k); }

    /* prev is new head of the input list */
    return prev;
}

/* UTILITY FUNCTIONS */
/* Function to push a node */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print linked list */
void printList(struct node *node)
{
    while(node != NULL)
    {
        printf("%d  ", node->data);
        node = node->next;
    }
}

/* Drier program to test above function*/
int main(void)
{
    /* Start with the empty list */
    struct node* head = NULL;

     /* Created Linked list is 1->2->3->4->5->6->7->8 */
```

```
    push(&head, 8);
    push(&head, 7);
    push(&head, 6);
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);

    printf("\n Given linked list \n");
    printList(head);
    head = reverse(head, 3);

    printf("\n Reversed Linked list \n");
    printList(head);

    getchar();
    return(0);
}
```

Time Complexity: O(n) where n is the number of nodes in the given list.

MERGE SORT FOR LINKED LISTS

Merge Sort is often preferred for sorting a linked list. The slow random-access performance of a linked list makes some other algorithms (such as quicksort) perform poorly, and others (such as heapsort) completely impossible.

Let head be the first node of the linked list to be sorted and headRef be the pointer to head. Note that we need a reference to head in MergeSort() as the below implementation changes next links to sort the linked lists (not data at the nodes), so head node has to be changed if the data at original head is not the smallest value in linked list.

```
MergeSort(headRef)
1) If head is NULL or there is only one element in the Linked List
    then return.
2) Else divide the linked list into two halves.
      FrontBackSplit(head, &a, &b); /* a and b are two halves */
3) Sort the two halves a and b.
      MergeSort(a);
      MergeSort(b);
4) Merge the sorted a and b (using SortedMerge() discussed here)
   and update the head pointer using headRef.
     *headRef = SortedMerge(a, b);
```

?

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

/* Link list node */
struct node
{
    int data;
```

```
    struct node* next;
};

/* function prototypes */
struct node* SortedMerge(struct node* a, struct node* b);
void FrontBackSplit(struct node* source,
          struct node** frontRef, struct node** backRef);

/* sorts the linked list by changing next pointers (not data) */
void MergeSort(struct node** headRef)
{
  struct node* head = *headRef;
  struct node* a;
  struct node* b;

  /* Base case -- length 0 or 1 */
  if ((head == NULL) || (head->next == NULL))
  {
    return;
  }

  /* Split head into 'a' and 'b' sublists */
  FrontBackSplit(head, &a, &b);

  /* Recursively sort the sublists */
  MergeSort(&a);
  MergeSort(&b);

  /* answer = merge the two sorted lists together */
  *headRef = SortedMerge(a, b);
}

/* See http://geeksforgeeks.org/?p=3622 for details of this
   function */
struct node* SortedMerge(struct node* a, struct node* b)
{
  struct node* result = NULL;

  /* Base cases */
  if (a == NULL)
     return(b);
  else if (b==NULL)
     return(a);

  /* Pick either a or b, and recur */
  if (a->data <= b->data)
  {
     result = a;
     result->next = SortedMerge(a->next, b);
  }
  else
  {
     result = b;
     result->next = SortedMerge(a, b->next);
  }
```

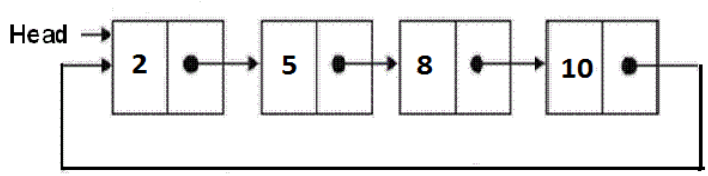
```
  return(result);
}

/* UTILITY FUNCTIONS */
/* Split the nodes of the given list into front and back halves,
     and return the two lists using the reference parameters.
     If the length is odd, the extra node should go in the front list.
     Uses the fast/slow pointer strategy.  */
void FrontBackSplit(struct node* source,
          struct node** frontRef, struct node** backRef)
{
  struct node* fast;
  struct node* slow;
  if (source==NULL || source->next==NULL)
  {
    /* length < 2 cases */
    *frontRef = source;
    *backRef = NULL;
  }
  else
  {
    slow = source;
    fast = source->next;

    /* Advance 'fast' two nodes, and advance 'slow' one node */
    while (fast != NULL)
    {
      fast = fast->next;
      if (fast != NULL)
      {
        slow = slow->next;
        fast = fast->next;
      }
    }

    /* 'slow' is before the midpoint in the list, so split it in two
      at that point. */
    *frontRef = source;
    *backRef = slow->next;
    slow->next = NULL;
  }
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while(node!=NULL)
  {
   printf("%d ", node->data);
   node = node->next;
  }
}

/* Function to insert a node at the beginging of the linked list */
void push(struct node** head_ref, int new_data)
{
```

```
  /* allocate node */
  struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

/* Drier program to test above functions*/
int main()
{
  /* Start with the empty list */
  struct node* res = NULL;
  struct node* a = NULL;
  struct node* b = NULL;

  /* Let us create a unsorted linked lists to test the functions
   Created lists shall be a: 2->3->20->5->10->15 */
  push(&a, 15);
  push(&a, 10);
  push(&a, 5);
  push(&a, 20);
  push(&a, 3);
  push(&a, 2);

  /* Remove duplicates from linked list */
  MergeSort(&a);

  printf("\n Sorted Linked List is: \n");
  printList(a);

  getchar();
  return 0;
}
```

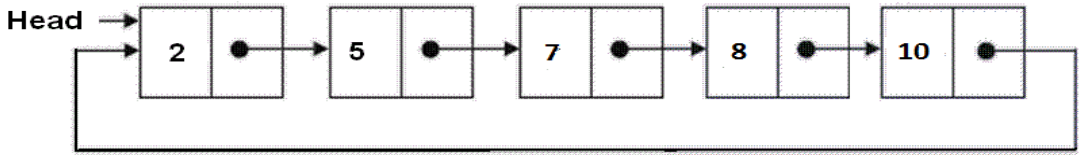
Time Complexity: O(nLogn)

SORTED INSERT FOR CIRCULAR LINKED LIST

Write a C function to insert a new value in a sorted Circular Linked List (CLL). For example, if the input CLL is following.



After insertion of **7**, the above CLL should be changed to following



Algorithm:
Allocate memory for the newly inserted node and put data in the newly allocated node. Let the pointer to the new node be new_node. After memory allocation, following are the three cases that need to be handled.

```
1) Linked List is empty:
    a)  since new_node is the only node in CLL, make a self loop.
          new_node->next = new_node;
    b) change the head pointer to point to new node.
          *head_ref = new_node;
2) New node is to be inserted just before the head node:
  (a) Find out the last node using a loop.
         while(current->next != *head_ref)
            current = current->next;
  (b) Change the next of last node.
         current->next = new_node;
  (c) Change next of new node to point to head.
         new_node->next = *head_ref;
  (d) change the head pointer to point to new node.
         *head_ref = new_node;
3) New node is to be  inserted somewhere after the head:
   (a) Locate the node after which new node is to be inserted.
         while ( current->next!= *head_ref &&
             current->next->data < new_node->data)
         {   current = current->next;   }
   (b) Make next of new_node as next of the located pointer
         new_node->next = current->next;
   (c) Change the next of the located pointer
         current->next = new_node;
```

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

/* structure for a node */
struct node
{
  int data;
  struct node *next;
```

```
};

/* function to insert a new_node in a list in sorted way.
   Note that this function expects a pointer to head node
   as this can modify the head of the input linked list */
void sortedInsert(struct node** head_ref, struct node* new_node)
{
  struct node* current = *head_ref;

  // Case 1 of the above algo
  if (current == NULL)
  {
     new_node->next = new_node;
     *head_ref = new_node;
  }

  // Case 2 of the above algo
  else if (current->data >= new_node->data)
  {
    /* If value is smaller than head's value then
      we need to change next of last node */
    while(current->next != *head_ref)
        current = current->next;
    current->next = new_node;
    new_node->next = *head_ref;
    *head_ref = new_node;
  }

  // Case 3 of the above algo
  else
  {
    /* Locate the node before the point of insertion */
    while (current->next!= *head_ref && current->next->data < new_node->data)
      current = current->next;

    new_node->next = current->next;
    current->next = new_node;
  }
}

/* Function to print nodes in a given linked list */
void printList(struct node *start)
{
  struct node *temp;

  if(start != NULL)
  {
    temp = start;
    printf("\n");
    do {
      printf("%d ", temp->data);
      temp = temp->next;
    } while(temp != start);
  }
}
```

```
/* Driver program to test above functions */
int main()
{
  int arr[] = {12, 56, 2, 11, 1, 90};
  int list_size, i;

  /* start with empty linked list */
  struct node *start = NULL;
  struct node *temp;

  /* Create linked list from the array arr[].
    Created linked list will be 1->2->11->56->12 */
  for(i = 0; i< 6; i++)
  {
    temp = (struct node *)malloc(sizeof(struct node));
    temp->data = arr[i];
    sortedInsert(&start, temp);
  }

  printList(start);
  getchar();
  return 0;
}
```

Output:
1 2 11 12 56 90

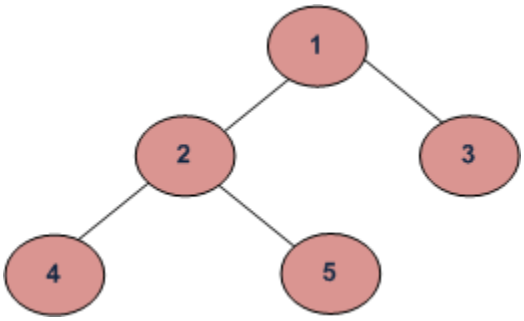
Time Complexity: O(n) where n is the number of nodes in the given linked list.

Case 2 of the above algorithm/code can be optimized. To implement the suggested change we need to modify the case 2 to following.

```
// Case 2 of the above algo
else if (current->data >= new_node->data)
{
  // swap the data part of head node and new node
  swap(&(current->data), &(new_node->data));  // assuming that we have a
function swap(int *, int *)

  new_node->next = (*head_ref)->next;
  (*head_ref)->next = new_node;
}
```

TREE TRAVERSALS

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.



Example Tree

Depth First Traversals:
(a) Inorder
(b) Preorder
(c) Postorder

Breadth First or Level Order Traversal
Please see this post for Breadth First Traversal.

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)
```

Uses of Inorder
In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder itraversal s reversed, can be used.
Example: Inorder traversal for the above given figure is 4 2 5 1 3.

Preorder Traversal:

```
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)
```

Uses of Preorder
Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expression on of an expression tree. Please see http://en.wikipedia.org/wiki/Polish_notation to know why prefix expressions are useful.
Example: Preorder traversal for the above given figure is 1 2 4 5 3.

Postorder Traversal:

```
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.
```

Uses of Postorder
Postorder traversal is used to delete the tree. Please see the question for deletion of tree for details. Postorder traversal is also useful to get the postfix expression of an expression tree. Please see http://en.wikipedia.org/wiki/Reverse_Polish_notation to for the usage of postfix expression.

Example: Postorder traversal for the above given figure is 4 5 2 3 1.

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

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
     int data;
     struct node* left;
     struct node* right;
};

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
     struct node* node = (struct node*)
                                  malloc(sizeof(struct node));
     node->data = data;
     node->left = NULL;
     node->right = NULL;

     return(node);
}

/* Given a binary tree, print its nodes according to the
  "bottom-up" postorder traversal. */
void printPostorder(struct node* node)
{
     if (node == NULL)
        return;
```

```
     // first recur on left subtree
     printPostorder(node->left);

     // then recur on right subtree
     printPostorder(node->right);

     // now deal with the node
     printf("%d ", node->data);
}

/* Given a binary tree, print its nodes in inorder*/
void printInorder(struct node* node)
{
     if (node == NULL)
          return;

     /* first recur on left child */
     printInorder(node->left);

     /* then print the data of node */
     printf("%d ", node->data);

     /* now recur on right child */
     printInorder(node->right);
}

/* Given a binary tree, print its nodes in inorder*/
void printPreorder(struct node* node)
{
     if (node == NULL)
          return;

     /* first print data of node */
     printf("%d ", node->data);

     /* then recur on left sutree */
     printPreorder(node->left);

     /* now recur on right subtree */
     printPreorder(node->right);
}

/* Driver program to test above functions*/
int main()
{
     struct node *root  = newNode(1);
     root->left             = newNode(2);
     root->right           = newNode(3);
     root->left->left     = newNode(4);
     root->left->right   = newNode(5);

     printf("\n Preorder traversal of binary tree is \n");
     printPreorder(root);
```

```
    printf("\n Inorder traversal of binary tree is \n");
    printInorder(root);

    printf("\n Postorder traversal of binary tree is \n");
    printPostorder(root);

    getchar();
    return 0;
}
```

Time Complexity: O(n)
Let us prove it:

Complexity function T(n) — for all problem where tree traversal is involved — can be defined as:

$T(n) = T(k) + T(n – k – 1) + c$

Where k is the number of nodes on one side of root and n-k-1 on the other side.

Let's do analysis of boundary conditions

Case 1: Skewed tree (One of the subtrees is empty and other subtree is non-empty)

k is 0 in this case.
$T(n) = T(0) + T(n-1) + c$
$T(n) = 2T(0) + T(n-2) + 2c$
$T(n) = 3T(0) + T(n-3) + 3c$
$T(n) = 4T(0) + T(n-4) + 4c$

……
……
$T(n) = (n-1)T(0) + T(1) + (n-1)c$
$T(n) = nT(0) + (n)c$

Value of T(0) will be some constant say d. (traversing a empty tree will take some constants time)

$T(n) = n(c+d)$
$T(n) = (-)(n)$ (Theta of n)

Case 2: Both left and right subtrees have equal number of nodes.

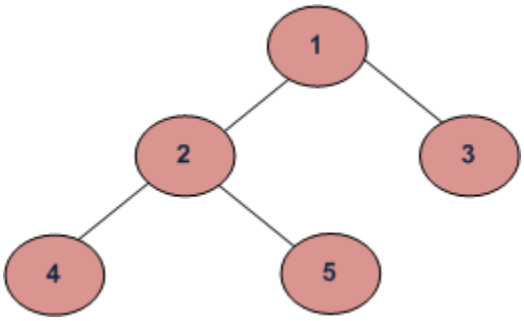
$T(n) = 2T(|\_n/2\_|) + c$

This recursive function is in the standard form (T(n) = aT(n/b) + (-)(n)) for master method http://en.wikipedia.org/wiki/Master_theorem. If we solve it by master method we get (-)(n)

Auxiliary Space : If we don't consider size of stack for function calls then O(1) otherwise O(n).

WRITE A C PROGRAM TO CALCULATE SIZE OF A TREE

Size of a tree is the number of elements present in the tree. Size of the below tree is 5.



Example Tree

Size() function recursively calculates the size of a tree. It works as follows:

Size of a tree = Size of left subtree + 1 + Size of right subtree

Algorithm:

```
size(tree)
1. If tree is empty then return 0
2. Else
     (a) Get the size of left subtree recursively  i.e., call
          size( tree->left-subtree)
     (a) Get the size of right subtree recursively  i.e., call
          size( tree->right-subtree)
     (c) Calculate size of the tree as following:
            tree_size  =  size(left-subtree) + size(right-
                                 subtree) + 1
     (d) Return tree_size
```

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

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
```

```
};

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
  struct node* node = (struct node*)
                       malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;

  return(node);
}

/* Computes the number of nodes in a tree. */
int size(struct node* node)
{
  if (node==NULL)
    return 0;
  else
    return(size(node->left) + 1 + size(node->right));
}

/* Driver program to test size function*/
int main()
{
  struct node *root = newNode(1);
  root->left        = newNode(2);
  root->right       = newNode(3);
  root->left->left  = newNode(4);
  root->left->right = newNode(5);

  printf("Size of the tree is %d", size(root));
  getchar();
  return 0;
}
```

Time & Space Complexities: Since this program is similar to traversal of tree, time and space complexities will be same as Tree traversal (Please see our Tree Traversal post for details)

WRITE C CODE TO DETERMINE IF TWO TREES ARE IDENTICAL

Two trees are identical when they have same data and arrangement of data is also same.

To identify if two trees are identical, we need to traverse both trees simultaneously, and while traversing we need to compare data and children of the trees.

Algorithm:

```
sameTree(tree1, tree2)
```

```
1. If both trees are empty then return 1.
2. Else If both trees are non -empty
     (a) Check data of the root nodes (tree1->data ==  tree2->data)
     (b) Check left subtrees recursively  i.e., call sameTree(
          tree1->left_subtree, tree2->left_subtree)
     (c) Check right subtrees recursively  i.e., call sameTree(
          tree1->right_subtree, tree2->right_subtree)
     (d) If a,b and c are true then return 1.
3  Else return 0 (one is empty and other is not)
```

?

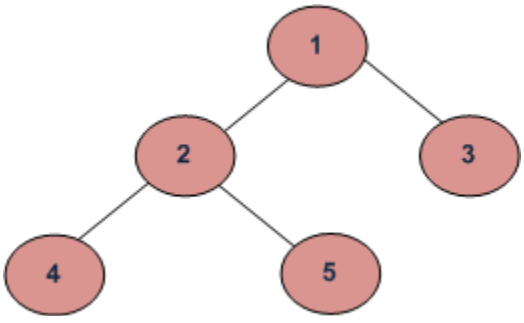
```
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                               malloc(sizeof(struct node));
    node->data  = data;
    node->left  = NULL;
    node->right = NULL;

    return(node);
}

/* Given two trees, return true if they are
 structurally identical */
int identicalTrees(struct node* a, struct node* b)
{
    /*1. both empty */
    if (a==NULL && b==NULL)
        return 1;

    /* 2. both non-empty -> compare them */
    else if (a!=NULL && b!=NULL)
   {
        return
        (
            a->data == b->data &&
            identicalTrees(a->left, b->left) &&
            identicalTrees(a->right, b->right)
        );
    }
    /* 3. one empty, one not -> false */
    else return 0;
```

```
}

/* Driver program to test identicalTrees function*/
int main()
{
    struct node *root1 = newNode(1);
    struct node *root2 = newNode(1);
    root1->left = newNode(2);
    root1->right = newNode(3);
    root1->left->left  = newNode(4);
    root1->left->right = newNode(5);

    root2->left = newNode(2);
    root2->right = newNode(3);
    root2->left->left = newNode(4);
    root2->left->right = newNode(5);

    if(identicalTrees(root1, root2))
        printf("Both tree are identical.");
    else
        printf("Trees are not identical.");

    getchar();
  return 0;
}
```

Time Complexity:
Complexity of the identicalTree() will be according to the tree with lesser number of nodes. Let number of nodes in two trees be m and n then complexity of sameTree() is O(m) where m < n.

WRITE A C PROGRAM TO FIND THE MAXIMUM DEPTH OR HEIGHT OF A TREE

Maximum depth or height of the below tree is 3.



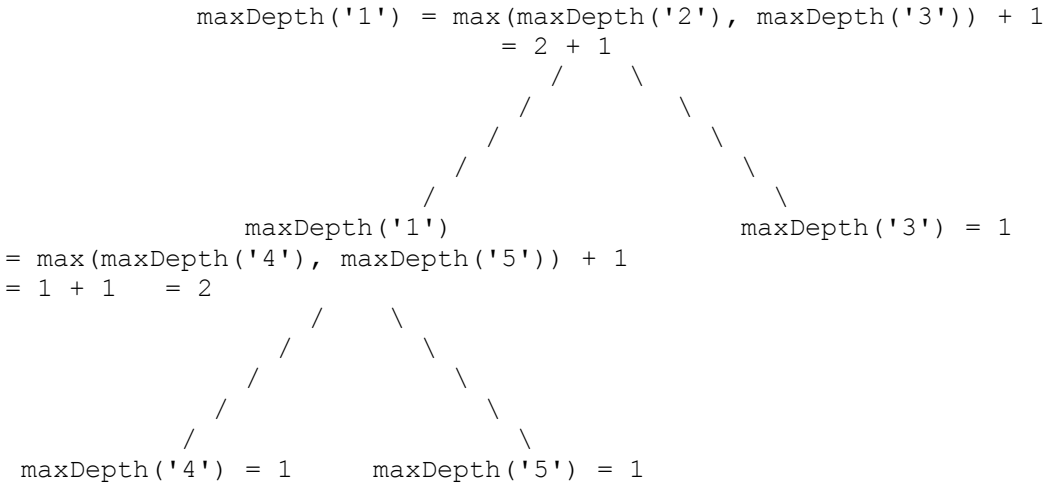
Example Tree

Recursively calculate height of left and right subtrees of a node and assign height to the node as max of the heights of two children plus 1. See below pseudo code and program for details.

Algorithm:

```
 maxDepth()
1. If tree is empty then return 0
2. Else
     (a) Get the max depth of left subtree recursively  i.e.,
          call maxDepth( tree->left-subtree)
     (a) Get the max depth of right subtree recursively  i.e.,
          call maxDepth( tree->right-subtree)
     (c) Get the max of max depths of left and right
          subtrees and add 1 to it for the current node.
         max_depth = max(max dept of left subtree,
                         max depth of right subtree)
                         + 1
     (d) Return max_depth
```

See the below diagram for more clarity about execution of the recursive function maxDepth() for above example tree.



Implementation:

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

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Compute the "maxDepth" of a tree -- the number of
    nodes along the longest path from the root node
    down to the farthest leaf node.*/
```

```
int maxDepth(struct 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);
   }
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                                malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

int main()
{
    struct node *root = newNode(1);

    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    printf("Height of tree is %d", maxDepth(root));

    getchar();
    return 0;
}
```

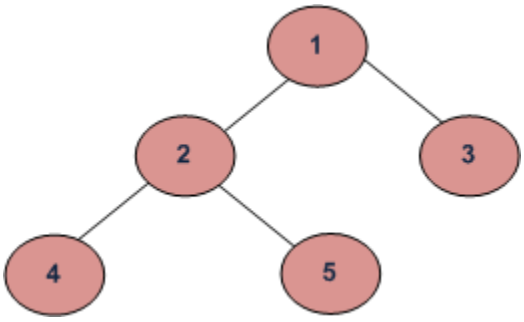
Time Complexity: O(n)

WRITE A C PROGRAM TO DELETE A TREE.

To delete a tree we must traverse all the nodes of the tree and delete them one by one. So which traversal we should use – Inorder or Preorder or Postorder. Answer is simple – Postorder, because before deleting the parent node we should delete its children nodes first

We can delete tree with other traversals also with extra space complexity but why should we go for other traversals if we have Postorder available which does the work without storing anything in same time complexity.

For the following tree nodes are deleted in order – 4, 5, 2, 3, 1



Example Tree

Program

?

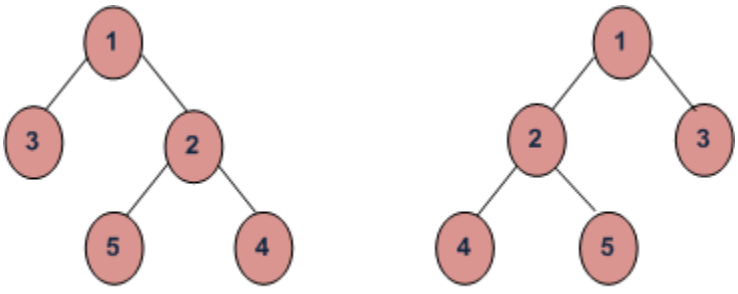
```
#include<stdio.h>
#include<stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */

struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                        malloc(sizeof(struct node));

    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return(node);
}
```

```
/*  This function traverses tree in post order to
    to delete each and every node of the tree */
void deleteTree(struct node* node)
{
    if (node == NULL) return;

    /* first delete both subtrees */
    deleteTree(node->left);
    deleteTree(node->right);

    /* then delete the node */
    printf("\n Deleting node: %d", node->data);
    free(node);
}

/* Driver program to test deleteTree function*/
int main()
{
    struct node *root = newNode(1);
    root->left            = newNode(2);
    root->right          = newNode(3);
    root->left->left     = newNode(4);
    root->left->right   = newNode(5);

    deleteTree(root);
    printf("\n Tree deleted ");

    getchar();
    return 0;
}
```

Time Complexity: O(n)
Space Complexity: If we don't consider size of stack for function calls then O(1) otherwise O(n)

WRITE AN EFFICIENT C FUNCTION TO CONVERT A BINARY TREE INTO ITS MIRROR TREE

Mirror of a Tree: Mirror of a Binary Tree T is another Binary Tree M(T) with left and right children of all non-leaf nodes interchanged.



Mirror Trees

Trees in the below figure are mirror of each other

Algorithm - Mirror(tree):

```
(1)  Call Mirror for left-subtree    i.e., Mirror(left-subtree)
(2)  Call Mirror for right-subtree  i.e., Mirror(left-subtree)
(3)  Swap left and right subtrees.
          temp = left-subtree
          left-subtree = right-subtree
          right-subtree = temp
```

Program:

?

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

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)

{
  struct node* node = (struct node*)
                       malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;

  return(node);
}

/* Change a tree so that the roles of the  left and
    right pointers are swapped at every node.

 So the tree...
       4
      / \
     2   5
    / \
   1   3

 is changed to...
       4
      / \
     5   2
        / \
```

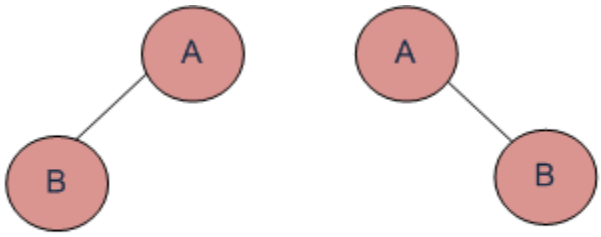
```
       3   1
*/
void mirror(struct node* node)
{
  if (node==NULL)
    return;
  else
  {
    struct node* temp;

    /* do the subtrees */
    mirror(node->left);
    mirror(node->right);

    /* swap the pointers in this node */
    temp        = node->left;
    node->left  = node->right;
    node->right = temp;
  }
}

/* Helper function to test mirror(). Given a binary
   search tree, print out its data elements in
   increasing sorted order.*/
void inOrder(struct node* node)
{
  if (node == NULL)
    return;

  inOrder(node->left);
  printf("%d ", node->data);

  inOrder(node->right);
}

/* Driver program to test mirror() */
int main()
{
  struct node *root = newNode(1);
  root->left        = newNode(2);
  root->right       = newNode(3);
  root->left->left  = newNode(4);
  root->left->right = newNode(5);

  /* Print inorder traversal of the input tree */
  printf("\n Inorder traversal of the constructed tree is \n");
  inOrder(root);

  /* Convert tree to its mirror */
  mirror(root);

  /* Print inorder traversal of the mirror tree */
  printf("\n Inorder traversal of the mirror tree is \n");
  inOrder(root);
```

```
  getchar();
  return 0;
}
```

Time & Space Complexities: This program is similar to traversal of tree space and time complexities will be same as Tree traversal (Please see our Tree Traversal post for details)

IF YOU ARE GIVEN TWO TRAVERSAL SEQUENCES, CAN YOU CONSTRUCT THE BINARY TREE?

It depends on what traversals are given. If one of the traversal methods is Inorder then the tree can be constructed, otherwise not.



Trees having Preorder, Postorder and Level-Order and traversals

Therefore, following combination can uniquely identify a tree.

Inorder and Preorder.
Inorder and Postorder.
Inorder and Level-order.

And following do not.
Postorder and Preorder.
Preorder and Level-order.
Postorder and Level-order.

For example, Preorder, Level-order and Postorder traversals are same for the trees given in above diagram.

Preorder Traversal = AB
Postorder Traversal = BA
Level-Order Traversal = AB

So, even if three of them (Pre, Post and Level) are given, the tree can not be constructed.

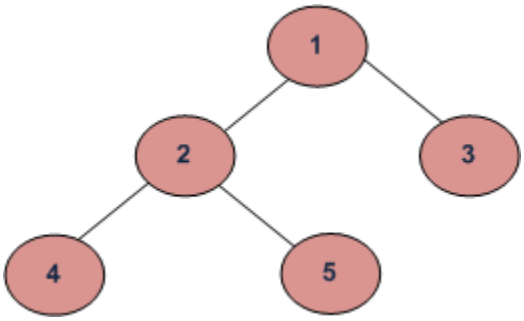
GIVEN A BINARY TREE, PRINT OUT ALL OF ITS ROOT-TO-LEAF PATHS ONE PER LINE.

Here is the solution.

Algorithm:

```
initialize: pathlen = 0, path[1000]
/*1000 is some max limit for paths, it can change*/

/*printPathsRecur traverses nodes of tree in preorder */
printPathsRecur(tree, path[], pathlen)
   1) If node is not NULL then
         a) push data to path array:
                path[pathlen] = node->data.
         b) increment pathlen
                pathlen++
   2) If node is a leaf node then print the path array.
   3) Else
        a) Call printPathsRecur for left subtree
                 printPathsRecur(node->left, path, pathLen)
        b) Call printPathsRecur for right subtree.
                printPathsRecur(node->right, path, pathLen)
```

Example:



Example Tree

Output for the above example will be

```
1 2 4
1 2 5
1 3
```

Implementation:

```
/*program to print all of its root-to-leaf paths for a tree*/
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
```

```
    int data;
    struct node* left;
    struct node* right;
};

void printArray(int [], int);
void printPathsRecur(struct node*, int [], int);
struct node* newNode(int );
void printPaths(struct node*);

/* Given a binary tree, print out all of its root-to-leaf
   paths, one per line. Uses a recursive helper to do the work.*/
void printPaths(struct node* node)
{
  int path[1000];
  printPathsRecur(node, path, 0);
}

/* Recursive helper function -- given a node, and an array containing
 the path from the root node up to but not including this node,
 print out all the root-leaf paths. */
void printPathsRecur(struct node* node, int path[], int pathLen)
{
  if (node==NULL) return;

  /* append this node to the path array */
  path[pathLen] = node->data;
  pathLen++;

  /* it's a leaf, so print the path that led to here */
  if (node->left==NULL && node->right==NULL)
  {
    printArray(path, pathLen);
  }
  else
  {
  /* otherwise try both subtrees */
    printPathsRecur(node->left, path, pathLen);
    printPathsRecur(node->right, path, pathLen);
  }
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
  struct node* node = (struct node*)
                      malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;

  return(node);
}
```

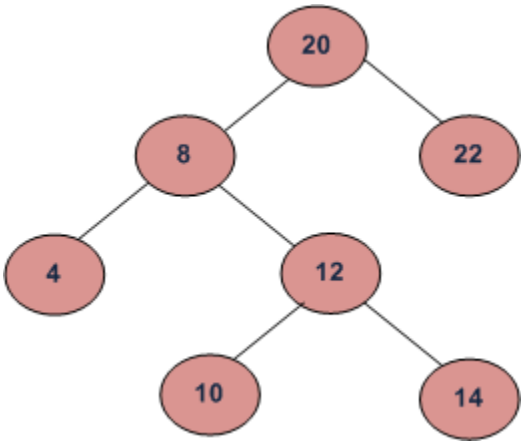
```
/* Utility that prints out an array on a line */
void printArray(int ints[], int len)
{
  int i;
  for (i=0; i<len; i++) {
    printf("%d ", ints[i]);
  }
  printf("\n");
}

/* Driver program to test mirror() */
int main()
{
  struct node *root = newNode(1);
  root->left        = newNode(2);
  root->right       = newNode(3);
  root->left->left  = newNode(4);
  root->left->right = newNode(5);

  /* Print all root-to-leaf paths of the input tree */
  printPaths(root);

  getchar();
  return 0;
}
```

LOWEST COMMON ANCESTOR IN A BINARY SEARCH TREE.

Given the values of two nodes in a *binary search tree*, write a c program to find the lowest common ancestor. You may assume that both values already exist in the tree.

The function prototype is as follows:

```
int FindLowestCommonAncestor(node* root, int value1, int value)
```




```
I/P : 4 and 14
O/P : 8
(Here the common ancestors of 4 and 14, are {8,20}.
```

```
  Of {8,20}, the lowest one is 8).
```

Here is the solution

Algorithm:
The main idea of the solution is — While traversing Binary Search Tree from top to bottom, the first node n we encounter with value between n1 and n2, i.e., n1 < n < n2 is the Lowest or Least Common Ancestor(LCA) of n1 and n2 (where n1 < n2). So just traverse the BST in pre-order, if you find a node with value in between n1 and n2 then n is the LCA, if it's value is greater than both n1 and n2 then our LCA lies on left side of the node, if it's value is smaller than both n1 and n2 then LCA lies on right side.

Implementation:

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

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

struct node* newNode(int );

/* Function to find least comman ancestor of n1 and n2 */
int leastCommanAncestor(struct node* root, int n1, int n2)
{
  /* If we have reached a leaf node then LCA doesn't exist
     If root->data is equal to any of the inputs then input is
     not valid. For example 20, 22 in the given figure */
  if(root == NULL || root->data == n1 || root->data == n2)
    return -1;

  /* If any of the input nodes is child of the current node
     we have reached the LCA. For example, in the above figure
     if we want to calculate LCA of 12 and 14, recursion should
     terminate when we reach 8*/
  if((root->right != NULL) &&
    (root->right->data == n1 || root->right->data == n2))
    return root->data;
  if((root->left != NULL) &&
    (root->left->data == n1 || root->left->data == n2))
    return root->data;

  if(root->data > n1 && root->data < n2)
    return root->data;
  if(root->data > n1 && root->data > n2)
```

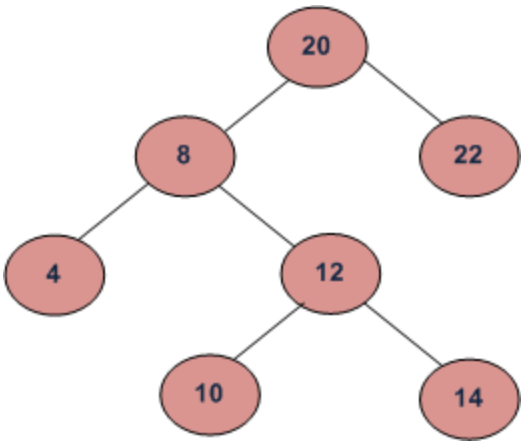
```
    return leastCommanAncestor(root->left, n1, n2);
  if(root->data < n1 && root->data < n2)
    return leastCommanAncestor(root->right, n1, n2);
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
  struct node* node = (struct node*)
                      malloc(sizeof(struct node));
  node->data  = data;
  node->left  = NULL;
  node->right = NULL;

  return(node);
}

/* Driver program to test mirror() */
int main()
{
  struct node *root  = newNode(2);
  root->left         = newNode(1);
  root->right        = newNode(4);
  root->right->left  = newNode(3);
  root->right->right = newNode(5);

/* Constructed binary search tree is
            2
          / \
        1   4
            / \
          3   5
*/
  printf("\n The Least Common Ancestor is \n");
  printf("%d", leastCommanAncestor(root, 3, 5));

  getchar();
  return 0;
}
```

Note that above function assumes that n1 is smaller than n2.

Time complexity: Time complexity is O(Logn) for a balanced BST and O(n) for a skewed BST.

FIND THE NODE WITH MINIMUM VALUE IN A BINARY SEARCH TREE

This is quite simple. Just traverse the node from root to left recursively until left is NULL. The node whose left is NULL is the node with minimum value.



For the above tree, we start with 20, then we move left 8, we keep on moving to left until we see NULL. Since left of 4 is NULL, 4 is the node with minimum value.

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

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Helper function that allocates a new node
with the given data and NULL left and right
pointers. */
struct node* newNode(int data)
{
  struct node* node = (struct node*)
                       malloc(sizeof(struct node));
  node->data  = data;
  node->left  = NULL;
  node->right = NULL;

  return(node);
}

/* Give a binary search tree and a number,
inserts a new node with the given number in
the correct place in the tree. Returns the new
root pointer which the caller should then use
(the standard trick to avoid using reference
parameters). */
struct node* insert(struct node* node, int data)
{
  /* 1. If the tree is empty, return a new,
```

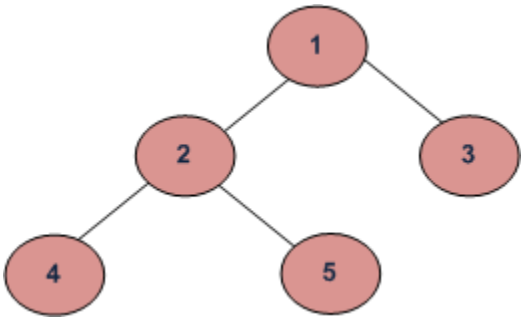
```
      single node */
  if (node == NULL)
    return(newNode(data));
  else
  {
    /* 2. Otherwise, recur down the tree */
    if (data <= node->data)
        node->left  = insert(node->left, data);
    else
        node->right = insert(node->right, data);

    /* return the (unchanged) node pointer */
    return node;
  }
}

/* Given a non-empty binary search tree,
return the minimum data value found in that
tree. Note that the entire tree does not need
to be searched. */
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);
}

/* Driver program to test sameTree function*/
int main()
{
  struct node* root = NULL;
  root = insert(root, 4);
  insert(root, 2);
  insert(root, 1);
  insert(root, 3);
  insert(root, 6);
  insert(root, 5);

  printf("\n Minimum value in BST is %d", minValue(root));
  getchar();
  return 0;
}
```

Time Complexity: O(n) Worst case happens for left skewed trees.

Similarly we can get the maximum value by recursively traversing the right node of a binary search tree.

LEVEL ORDER TREE 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

METHOD 1 (Use function to print a given level)

Algorithm:
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);
```

Implementation:

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

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
```

```
    int data;
    struct node* left;
    struct node* right;
};

/*Function protoypes*/
void printGivenLevel(struct node* root, int level);
int height(struct node* node);
struct node* newNode(int data);

/* Function to print level order traversal a tree*/
void printLevelOrder(struct node* root)
{
  int h = height(root);
  int i;
  for(i=1; i<=h; i++)
    printGivenLevel(root, i);
}

/* Print nodes at a given level */
void printGivenLevel(struct node* root, int level)
{
  if(root == NULL)
    return;
  if(level == 1)
    printf("%d ", root->data);
  else if (level > 1)
  {
    printGivenLevel(root->left, level-1);
    printGivenLevel(root->right, level-1);
  }
}

/* Compute the "height" of a tree -- the number of
    nodes along the longest path from the root node
    down to the farthest leaf node.*/
int height(struct node* node)
{
   if (node==NULL)
       return 0;
   else
   {
     /* compute the height of each subtree */
     int lheight = height(node->left);
     int rheight = height(node->right);

     /* use the larger one */
     if (lheight > rheight)
         return(lheight+1);
     else return(rheight+1);
   }
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
```

```
struct node* newNode(int data)
{
  struct node* node = (struct node*)
                       malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;

  return(node);
}

/* Driver program to test above functions*/
int main()
{
  struct node *root = newNode(1);
  root->left        = newNode(2);
  root->right       = newNode(3);
  root->left->left  = newNode(4);
  root->left->right = newNode(5);

  printf("Level Order traversal of binary tree is \n");
  printLevelOrder(root);

  getchar();
  return 0;
}
```

Time Complexity: O(n^2) in worst case. For a skewed tree, printGivenLevel() takes O(n) time where n is the number of nodes in the skewed tree. So time complexity of printLevelOrder() is O(n) + O(n-1) + O(n-2) + .. + O(1) which is O(n^2).

METHOD 2 (Use Queue)

Algorithm:
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
```

Implementation:
Here is a simple implementation of the above algorithm. Queue is implemented using an array with maximum size of 500. We can implement queue as linked list also.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_Q_SIZE 500

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* frunction prototypes */
struct node** createQueue(int *, int *);
void enQueue(struct node **, int *, struct node *);
struct node *deQueue(struct node **, int *);

/* Given a binary tree, print its nodes in level order
   using array for implementing queue */
void printLevelOrder(struct node* root)
{
  int rear, front;
  struct node **queue = createQueue(&front, &rear);
  struct node *temp_node = root;

  while(temp_node)
  {
    printf("%d ", temp_node->data);

    /*Enqueue left child */
    if(temp_node->left)
      enQueue(queue, &rear, temp_node->left);

    /*Enqueue right child */
    if(temp_node->right)
      enQueue(queue, &rear, temp_node->right);

    /*Dequeue node and make it temp_node*/
    temp_node = deQueue(queue, &front);
  }
}

/*UTILITY FUNCTIONS*/
struct node** createQueue(int *front, int *rear)
{
  struct node **queue =
   (struct node **)malloc(sizeof(struct node*)*MAX_Q_SIZE);

  *front = *rear = 0;
  return queue;
}

void enQueue(struct node **queue, int *rear, struct node *new_node)
{
```

```
  queue[*rear] = new_node;
  (*rear)++;
}

struct node *deQueue(struct node **queue, int *front)
{
  (*front)++;
  return queue[*front - 1];
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
  struct node* node = (struct node*)
                      malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;

  return(node);
}

/* Driver program to test above functions*/
int main()
{
  struct node *root = newNode(1);
  root->left        = newNode(2);
  root->right       = newNode(3);
  root->left->left  = newNode(4);
  root->left->right = newNode(5);

  printf("Level Order traversal of binary tree is \n");
  printLevelOrder(root);

  getchar();
  return 0;
}
```

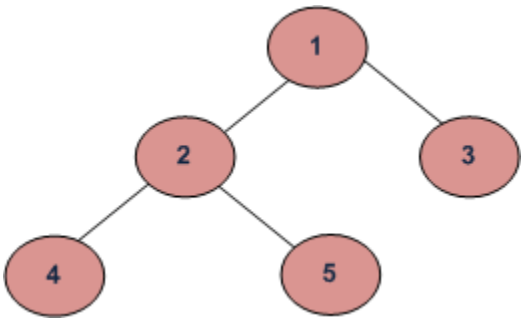
Time Complexity: O(n) where n is number of nodes in the binary tree

PROGRAM TO COUNT LEAF NODES IN A BINARY TREE

A node is a leaf node if both left and right child nodes of it are NULL.

Here is an algorithm to get the leaf node count.

```
getLeafCount(node)
1) If node is NULL then return 0.
2) Else If left and right child nodes are NULL return 1.
3) Else recursively calculate leaf count of the tree using below formula.
    Leaf count of a tree = Leaf count of left subtree +
                               Leaf count of right subtree
```



Example Tree

Leaf count for the above tree is 3.

Implementation:

?

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

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Function to get the count of leaf nodes in a binary tree*/
unsigned 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);
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
  struct node* node = (struct node*)
                      malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;
```

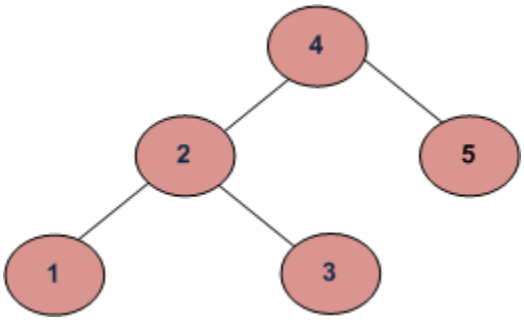
```
  return(node);
}

/*Driver program to test above functions*/
int main()
{
  /*create a tree*/
  struct node *root = newNode(1);
  root->left        = newNode(2);
  root->right       = newNode(3);
  root->left->left  = newNode(4);
  root->left->right = newNode(5);

  /*get leaf count of the above created tree*/
  printf("Leaf count of the tree is %d", getLeafCount(root));

  getchar();
  return 0;
}
```

Time & Space Complexities: Since this program is similar to traversal of tree, time and space complexities will be same as Tree traversal

A PROGRAM TO CHECK IF A BINARY TREE IS BST OR NOT

A binary search tree (BST) is a node based binary tree data structure which has the following properties.
• The left subtree of a node contains only nodes with keys less than the node's key.
• The right subtree of a node contains only nodes with keys greater than the node's key.
• Both the left and right subtrees must also be binary search trees.

From the above properties it naturally follows that:
• Each node (item in the tree) has a distinct key.

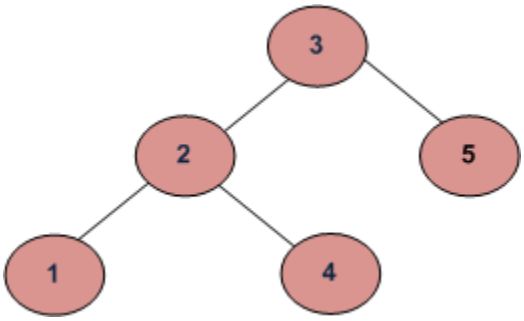


METHOD 1 (Simple but Wrong)
Following is a simple program. For each node, check if left node of it is smaller than the node and right node of it is greater than the node.

?

```
int isBST(struct node* node)
{
  if (node == NULL)
    return 1;

  /* false if left is > than node */
  if (node->left != NULL && node->left->data > node->data)
    return 0;

  /* false if right is < than node */
  if (node->right != NULL && node->right->data <= node->data)
    return 0;

  /* false if, recursively, the left or right is not a BST */
  if (!isBST(node->left) || !isBST(node->right))
    return 0;

  /* passing all that, it's a BST */
  return 1;
}
```

This approach is wrong as this will return true for below binary tree (and below tree is not a BST because 4 is in left subtree of 3)



METHOD 2 (Correct but not efficient)
For each node, check if max value in left subtree is smaller than the node and min value in right subtree greater than the node.

```
/* Returns true if a binary tree is a binary search tree */
int isBST(struct node* node)
{
  if (node == NULL)
    return(true);

  /* false if the max of the left is > than us */
  if (node->left!=NULL && maxValue(node->left) > node->data)
```

```
    return(false);

  /* false if the min of the right is <= than us */
  if (node->right!=NULL && minValue(node->right) <= node->data)
    return(false);

  /* false if, recursively, the left or right is not a BST */
  if (!isBST(node->left) || !isBST(node->right))
    return(false);

  /* passing all that, it's a BST */
  return(true);
}
```

It is assumed that you have helper functions minValue() and maxValue() that return the min or max int value from a non-empty tree

METHOD 3 (Correct and Efficient)
Method 2 above runs slowly since it traverses over some parts of the tree many times. A better solution looks at each node only once. The trick is to write a utility helper function isBSTUtil(struct node* node, int min, int max) that traverses down the tree keeping track of the narrowing min and max allowed values as it goes, looking at each node only once. The initial values for min and max should be INT_MIN and INT_MAX — they narrow from there.

```
/* Returns true if the given tree is a binary search tree
 (efficient version). */
int isBST(struct node* node)
{
  return(isBSTUtil(node, INT_MIN, INT_MAX));
}

/* Returns true if the given tree is a BST and its
 values are >= min and <= max. */
int isBSTUtil(struct node* node, int min, int max)
```

Implementation:

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

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};
```

```
int isBSTUtil(struct node* node, int min, int max);

/* Returns true if the given tree is a binary search tree
 (efficient version). */
int isBST(struct node* node)
{
  return(isBSTUtil(node, INT_MIN, INT_MAX));
}

/* Returns true if the given tree is a BST and its
   values are >= min and <= max. */
int isBSTUtil(struct node* node, int min, int max)
{

  /* an empty tree is BST */
  if (node==NULL)
     return 1;

  /* false if this node violates the min/max constraint */
  if (node->data < min || node->data > max)
     return 0;

  /* otherwise check the subtrees recursively,
   tightening the min or max constraint */
  return
    isBSTUtil(node->left, min, node->data) &&
    isBSTUtil(node->right, node->data+1, max);
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
  struct node* node = (struct node*)
                       malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;

  return(node);
}

/* Driver program to test above functions*/
int main()
{
  struct node *root = newNode(4);
  root->left        = newNode(2);
  root->right       = newNode(5);
  root->left->left  = newNode(1);
  root->left->right = newNode(3);

  if(isBST(root))
    printf("Is BST");
  else
    printf("Not a BST");
```

```
  getchar();
  return 0;
}
```

Time Complexity: O(n)
Auxiliary Space : O(1) if Function Call Stack size is not considered, otherwise O(n)

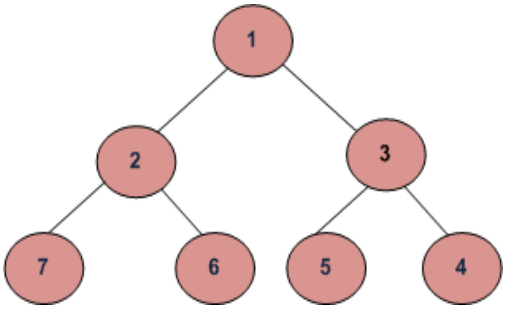
METHOD 4(Using In-Order Traversal)
1) Do In-Order Traversal of the given tree and store the result in a temp array.
3) Check if the temp array is sorted in ascending order, if it is, then the tree is BST.

Time Complexity: O(n)
Auxiliary Space: O(n)

Sources:
http://en.wikipedia.org/wiki/Binary_search_tree
http://cslibrary.stanford.edu/110/BinaryTrees.html

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.



Algorithm:
This problem is 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

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

Implementation:

?

```
#include <stdio.h>
#include <stdlib.h>
#define bool int

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
  int data;
  struct node* left;
  struct node* right;
};

/*Function protoypes*/
void printGivenLevel(struct node* root, int level, int ltr);
int height(struct node* node);
struct node* newNode(int data);

/* Function to print spiral traversal of a tree*/
void printLevelOrder(struct node* root)
{
  int h = height(root);
  int i;

  /*ltr -> Left to Right. If this variable is set,
    then the given level is traverseed from left to right. */
  bool ltr = 0;
  for(i=1; i<=h; i++)
  {
    printGivenLevel(root, i, ltr);

    /*Revert ltr to traverse next level in oppposite order*/
    ltr = ~ltr;
  }
}
```

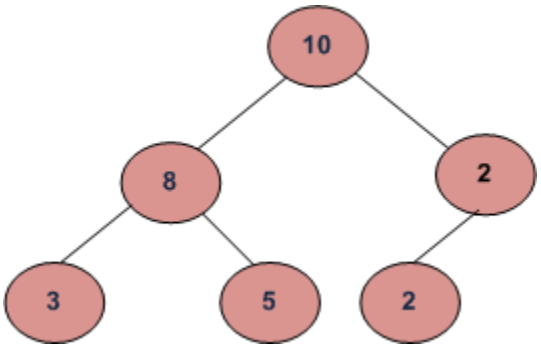
```
/* Print nodes at a given level */
void printGivenLevel(struct node* root, int level, int ltr)
{
  if(root == NULL)
    return;
  if(level == 1)
    printf("%d ", root->data);
  else if (level > 1)
  {
    if(ltr)
    {
      printGivenLevel(root->left, level-1, ltr);
      printGivenLevel(root->right, level-1, ltr);
    }
    else
    {
      printGivenLevel(root->right, level-1, ltr);
      printGivenLevel(root->left, level-1, ltr);
    }
  }
}

/* Compute the "height" of a tree -- the number of
    nodes along the longest path from the root node
    down to the farthest leaf node.*/
int height(struct node* node)
{
   if (node==NULL)
       return 0;
   else
   {
     /* compute the height of each subtree */
     int lheight = height(node->left);
     int rheight = height(node->right);

     /* use the larger one */
     if (lheight > rheight)
         return(lheight+1);
     else return(rheight+1);
   }
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
  struct node* node = (struct node*)
                       malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;

  return(node);
}
```

```
/* Driver program to test above functions*/
int main()
{
  struct node *root = newNode(1);
  root->left        = newNode(2);
  root->right       = newNode(3);
  root->left->left  = newNode(7);
  root->left->right = newNode(6);
  root->right->left  = newNode(5);
  root->right->right = newNode(4);
  printf("Level Order traversal of binary tree is \n");
  printLevelOrder(root);

  getchar();
  return 0;
}
```

CHECK FOR CHILDREN SUM PROPERTY IN A BINARY TREE.

Given a binary tree, write a function that returns true if the tree satisfies below property.

For every node, data value must be equal to sum of data values in left and right children. Consider data value as 0 for NULL children. Below tree is an example



Algorithm:
Traverse the given binary tree. For each node check (recursively) if the node and both its children satisfy the Children Sum Property, if so then return true else return false.

Implementation:

[?]
```
/* Program to check children sum property */
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, left child and right child */
```

```
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* returns 1 if children sum property holds for the given
    node and both of its children*/
int isSumProperty(struct node* node)
{
  /* left_data is left child data and right_data is for right child data*/
  int left_data = 0,  right_data = 0;

  /* If node is NULL or it's a leaf node then
     return true */
  if(node == NULL ||
     (node->left == NULL && node->right == NULL))
    return 1;
  else
  {
    /* If left child is not present then 0 is used
       as data of left child */
    if(node->left != NULL)
      left_data = node->left->data;

    /* If right child is not present then 0 is used
      as data of right child */
    if(node->right != NULL)
      right_data = node->right->data;

    /* if the node and both of its children satisfy the
       property return 1 else 0*/
    if((node->data == left_data + right_data)&&
        isSumProperty(node->left) &&
        isSumProperty(node->right))
      return 1;
    else
      return 0;
  }
}

/*
 Helper function that allocates a new node
 with the given data and NULL left and right
 pointers.
*/
struct node* newNode(int data)
{
  struct node* node =
      (struct node*)malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;
  return(node);
}
```

```
/* Driver program to test above function */
int main()
{
  struct node *root  = newNode(10);
  root->left         = newNode(8);
  root->right        = newNode(2);
  root->left->left   = newNode(3);
  root->left->right  = newNode(5);
  root->right->right = newNode(2);
  if(isSumProperty(root))
    printf("The given tree satisfies the children sum property ");
  else
    printf("The given tree does not satisfy the children sum property ");

  getchar();
  return 0;
}
```

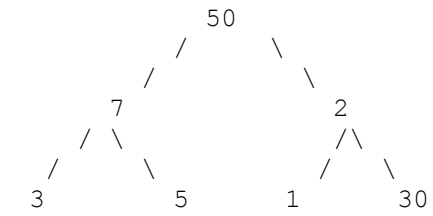
Time Complexity: O(n), we are doing a complete traversal of the tree.

Now extend the above question for an n-ary tree.

CONVERT AN ARBITRARY BINARY TREE TO A TREE THAT HOLDS CHILDREN SUM PROPERTY

Question: Given an arbitrary binary tree, convert it to a binary tree that holds Children Sum Property. You can only increment data values in any node.

For example, the below tree doesn't hold the children sum property, convert it to a tree that holds the property.



Algorithm:
Traverse given tree in post order to convert it, i.e., first change left and right children to hold the children sum property then change the parent node.

Let difference between node's data and children sum be diff.

```
diff = node's children sum - node's data
```

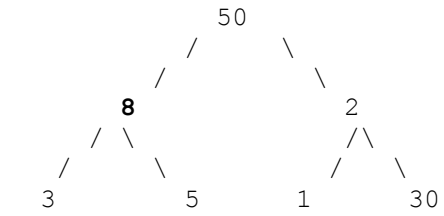
If diff is 0 then nothing needs to be done.

If diff > 0 (node's data is smaller than node's children sum) increment the node's data by diff.

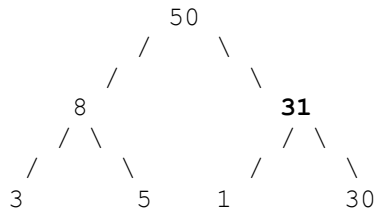
If diff < 0 (node's data is smaller than the node's children sum) then increment one child's data. We can choose to increment either left or right child. Let us always increment the left child. Incrementing a child changes the subtree's children sum property so we need to change left subtree also. Left subtree is fixed by incrementing all the children in left subtree by diff, we have a function increment() for this purpose (see below C code).

Let us run the algorithm for the given example.

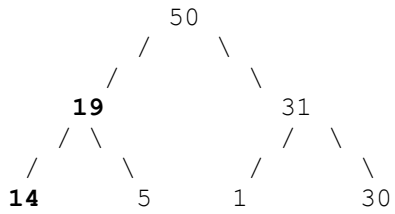
First convert the left subtree (increment 7 to 8).



Then convert the right subtree (increment 2 to 31)



Now convert the root, we have to increment left subtree for converting the root.



Please note the last step – we have incremented 8 to 19, and to fix the subtree we have incremented 3 to 14.

Implementation:

```
?
/* Program to convert an aribitary binary tree to
   a tree that holds children sum property */

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

struct node
```

```
{
  int data;
  struct node* left;
  struct node* right;
};

/* This function is used to increment left subtree */
void increment(struct node* node, int diff);

/* Helper function that allocates a new node
 with the given data and NULL left and right
 pointers. */
struct node* newNode(int data);

/* This function changes a tree to to hold children sum
   property */
void convertTree(struct node* node)
{
  int left_data = 0,  right_data = 0, diff;

  /* If tree is empty or it's a leaf node then
     return true */
  if(node == NULL ||
     (node->left == NULL && node->right == NULL))
    return;
  else
  {
    /* convert left and right subtrees  */
    convertTree(node->left);
    convertTree(node->right);

    /* If left child is not present then 0 is used
       as data of left child */
    if(node->left != NULL)
      left_data = node->left->data;

    /* If right child is not present then 0 is used
      as data of right child */
    if(node->right != NULL)
      right_data = node->right->data;

    /* get the diff of node's data and children sum */
    diff = left_data + right_data - node->data;

    /* If node's data is smaller than children sum,
       then increment node's data by diff */
    if(diff > 0)
       node->data = node->data + diff;

    /* THIS IS TRICKY --> If node's data is greater than children sum,
      then increment left subtree by diff */
    if(diff < 0)
      increment(node, -diff);
  }
}
```

```
/* This function is used to increment left subtree */
void increment(struct node* node, int diff)
{
  /* This if is for the case where left child is NULL */
  if(node->left == NULL)
  {
    node->left = newNode(diff);
    return;
  }

  /* Go in depth, and fix all left children */
  while(node->left != NULL)
  {
    node->left->data = node->left->data + diff;
    node = node->left;
  }
}

/* Given a binary tree, printInorder() prints out its
   inorder traversal*/
void printInorder(struct node* node)
{
  if (node == NULL)
    return;

  /* first recur on left child */
  printInorder(node->left);

  /* then print the data of node */
  printf("%d ", node->data);

  /* now recur on right child */
  printInorder(node->right);
}

/* Helper function that allocates a new node
 with the given data and NULL left and right
 pointers. */
struct node* newNode(int data)
{
  struct node* node =
      (struct node*)malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;
  return(node);
}

/* Driver program to test above functions */
int main()
{
  struct node *root = newNode(50);
  root->left        = newNode(7);
  root->right       = newNode(2);
```

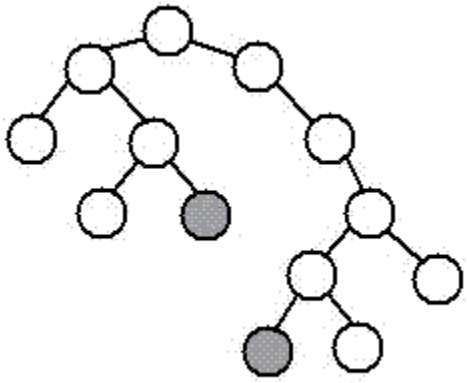
```
  root->left->left  = newNode(3);
  root->left->right = newNode(5);
  root->right->left  = newNode(1);
  root->right->right = newNode(30);

  printf("\n Inorder traversal before conversion ");
  printInorder(root);

  convertTree(root);

  printf("\n Inorder traversal after conversion ");
  printInorder(root);

  getchar();
  return 0;
}
```

Time Complexity: O(n^2), Worst case complexity is for a skewed tree such that nodes are in decreasing order from root to leaf.

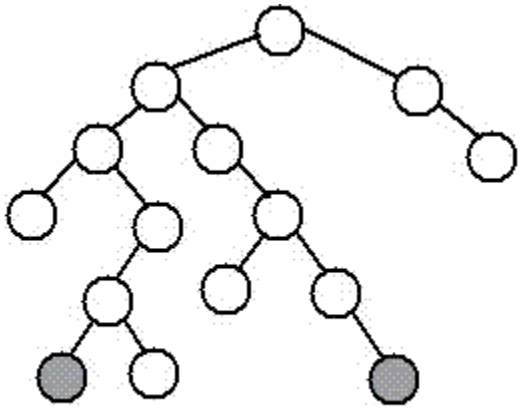
Now extend the above question for an n-ary tree.

DIAMETER OF A BINARY TREE

The diameter of a tree (sometimes called the width) is the number of nodes on the longest path between two leaves in the tree. The diagram below shows two trees each with diameter nine, the leaves that form the ends of a longest path are shaded (note that there is more than one path in each tree of length nine, but no path longer than nine nodes).



diameter, 9 nodes, through root



diameter, 9 nodes, NOT through root

The diameter of a tree T is the largest of the following quantities:

* the diameter of T's left subtree
* the diameter of T's right subtree

* the longest path between leaves that goes through the root of T (this can be computed from the heights of the subtrees of T)

Implementation:

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

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* function to create a new node of tree and returns pointer */
struct node* newNode(int data);

/* returns max of two integers */
int max(int a, int b);

/* Function to get diameter of a binary tree */
int diameter(struct node * tree)
{
   /* base case where tree is empty */
   if (tree == 0)
     return 0;

  /* get the height of left and right sub-trees */
  int lheight = height(tree->left);
  int rheight = height(tree->right);

  /* get the diameter of left and right sub-trees */
  int ldiameter = diameter(tree->left);
  int rdiameter = diameter(tree->right);

  /* Return max of following three
   1) Diameter of left subtree
   2) Diameter of right subtree
   3) Height of left subtree + height of right subtree + 1 */
  return max(lheight + rheight + 1, max(ldiameter, rdiameter));
}

/* UTILITY FUNCTIONS TO TEST diameter() FUNCTION */

/*  The function Compute the "height" of a tree. Height is the
    number f nodes along the longest path from the root node
    down to the farthest leaf node.*/
int height(struct node* node)
{
   /* base case tree is empty */
   if(node == NULL)
```

```
        return 0;

    /* If tree is not empty then height = 1 + max of left
       height and right heights */
    return 1 + max(height(node->left), height(node->right));
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
  struct node* node = (struct node*)
                       malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;

  return(node);
}

/* returns maximum of two integers */
int max(int a, int b)
{
  return (a >= b)? a: b;
}

/* Driver program to test above functions*/
int main()
{

  /* Constructed binary tree is
            1
          /   \
        2      3
      /  \
    4     5
  */
  struct node *root = newNode(1);
  root->left        = newNode(2);
  root->right       = newNode(3);
  root->left->left  = newNode(4);
  root->left->right = newNode(5);

  printf("Diameter of the given binary tree is %d\n", diameter(root));

  getchar();
  return 0;
}
```

Time Complexity: O(n^2)

Optimized implementation: The above implementation can be optimized by calculating the

height in the same recursion rather than calling a height() separately. This optimization reduces time complexity to O(n).

```
/*The second parameter is to store the height of tree.
   Initially, we need to pass a pointer to a location with value
   as 0. So, function should be used as follows:

   int height = 0;
   struct node *root = SomeFunctionToMakeTree();
   int diameter = diameterOpt(root, &height); */
int diameterOpt(struct node *root, int* height)
{
  /* lh --> Height of left subtree
      rh --> Height of right subtree */
  int lh = 0, rh = 0;

  /* ldiameter  --> diameter of left subtree
      rdiameter  --> Diameter of right subtree */
  int ldiameter = 0, rdiameter = 0;

  if(root == NULL)
  {
    *height = 0;
     return 0; /* diameter is also 0 */
  }

  /* Get the heights of left and right subtrees in lh and rh
    And store the returned values in ldiameter and ldiameter */
  ldiameter = diameterOpt(root->left, &lh);
  rdiameter = diameterOpt(root->right, &rh);

  /* Height of current node is max of heights of left and
     right subtrees plus 1*/
  *height = max(lh, rh) + 1;

  return max(lh + rh + 1, max(ldiameter, rdiameter));
}
```

Time Complexity: O(n)

References:
http://www.cs.duke.edu/courses/spring00/cps100/assign/trees/diameter.html

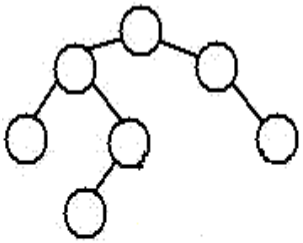
HOW TO DETERMINE IF A BINARY TREE IS HEIGHT-BALANCED?

A tree where no leaf is much farther away from the root than any other leaf. Different balancing schemes allow different definitions of “much farther” and different amounts of work to keep them balanced.

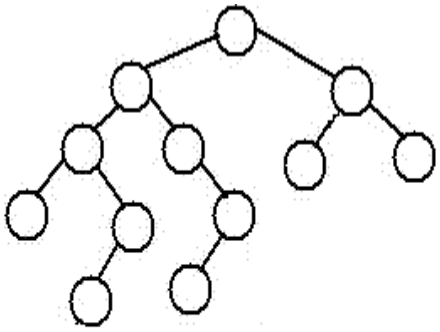
Source: http://www.itl.nist.gov/div897/sqg/dads/HTML/balancedtree.html

Consider a height-balancing scheme where following conditions should be checked to determine if a binary tree is balanced.
An empty tree is height-balanced. A non-empty binary tree T is balanced if:
1) Left subtree of T is balanced
2) Right subtree of T is balanced
3) The difference between heights of left subtree and right subtree is not more than 1.

The above height-balancing scheme is used in AVL trees. The diagram below shows two trees, one of them is height-balanced and other is not. The second tree is not height-balanced because height of left subtree is 2 more than height of right subtree.



A height-balanced Tree



Not a height-balanced tree

To check if a tree is height-balanced, get the height of left and right subtrees. Return true if difference between heights is not more than 1 and left and right subtrees are balanced, otherwise return false.

?

```
/* program to check if a tree is height-balanced or not */
#include<stdio.h>
#include<stdlib.h>
#define bool int

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Returns the height of a binary tree */
int height(struct node* node);

/* Returns true if binary tree with root as root is height-balanced */
bool isBalanced(struct node *root)
{
```

```
   int lh; /* for height of left subtree */
   int rh; /* for height of right subtree */

   /* If tree is empty then return true */
   if(root == NULL)
    return 1;

   /* Get the height of left and right sub trees */
   lh = height(root->left);
   rh = height(root->right);

   if( abs(lh-rh) <= 1 &&
       isBalanced(root->left) &&
       isBalanced(root->right))
     return 1;

   /* If we reach here then tree is not height-balanced */
   return 0;
}

/* UTILITY FUNCTIONS TO TEST isBalanced() FUNCTION */

/* returns maximum of two integers */
int max(int a, int b)
{
  return (a >= b)? a: b;
}

/*  The function Compute the "height" of a tree. Height is the
    number of nodes along the longest path from the root node
    down to the farthest leaf node.*/
int height(struct node* node)
{
   /* base case tree is empty */
   if(node == NULL)
       return 0;

   /* If tree is not empty then height = 1 + max of left
      height and right heights */
   return 1 + max(height(node->left), height(node->right));
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                                malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}
```

```
int main()
{
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->left->left->left = newNode(8);

    if(isBalanced(root))
      printf("Tree is balanced");
    else
      printf("Tree is not balanced");

    getchar();
    return 0;
}
```

Time Complexity: O(n^2) Worst case occurs in case of skewed tree.

Optimized implementation: Above implementation can be optimized by calculating the height in the same recursion rather than calling a height() function separately. Thanks to Amar for suggesting this optimized version. This optimization reduces time complexity to O(n).

?

```
/* program to check if a tree is height-balanced or not */
#include<stdio.h>
#include<stdlib.h>
#define bool int

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
  int data;
  struct node* left;
  struct node* right;
};

/* The function returns true if root is balanced else false
   The second parameter is to store the height of tree.
   Initially, we need to pass a pointer to a location with value
   as 0. We can also write a wrapper over this function */
bool isBalanced(struct node *root, int* height)
{
  /* lh --> Height of left subtree
     rh --> Height of right subtree */
  int lh = 0, rh = 0;

  /* l will be true if left subtree is balanced
    and r will be true if right subtree is balanced */
```

```
  int l = 0, r = 0;

  if(root == NULL)
  {
    *height = 0;
     return 1;
  }

  /* Get the heights of left and right subtrees in lh and rh
    And store the returned values in l and r */
  l = isBalanced(root->left, &lh);
  r = isBalanced(root->right,&rh);

  /* Height of current node is max of heights of left and
     right subtrees plus 1*/
  *height = (lh > rh? lh: rh) + 1;

  /* If difference between heights of left and right
     subtrees is more than 2 then this node is not balanced
     so return 0 */
  if((lh - rh >= 2) || (rh - lh >= 2))
    return 0;

  /* If this node is balanced and left and right subtrees
    are balanced then return true */
  else return l&&r;
}

/* UTILITY FUNCTIONS TO TEST isBalanced() FUNCTION */

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                                malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

int main()
{
  int height = 0;

  /* Constructed binary tree is
             1
           /   \
         2      3
       /  \    /
     4     5  6
    /
   7
```

```
  */
  struct node *root = newNode(1);
  root->left = newNode(2);
  root->right = newNode(3);
  root->left->left = newNode(4);
  root->left->right = newNode(5);
  root->right->left = newNode(6);
  root->left->left->left = newNode(7);

  if(isBalanced(root, &height))
    printf("Tree is balanced");
  else
    printf("Tree is not balanced");

  getchar();
  return 0;
}
```

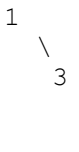
Time Complexity: O(n)

INORDER TREE TRAVERSAL WITHOUT RECURSION

Using Stack is the obvious way to traverse tree without recursion. Below is an algorithm for traversing binary tree using stack. See this for step wise step execution of the algorithm.

```
1) Create an empty stack S.
2) Initialize current node as root
3) Push the current node to S and set current = current->left until current
is NULL
4) If current is NULL and stack is not empty then
     a) Pop the top item from stack.
     b) Print the popped item, set current = current->right
     c) Go to step 3.
5) If current is NULL and stack is empty then we are done.
```

Let us consider the below tree for example




```
Step 1 Creates an empty stack: S = NULL

Step 2 sets current as address of root: current -> 1

Step 3 Pushes the current node and set current = current->left until current
is NULL
     current -> 1
     push 1: Stack S -> 1
     current -> 2
     push 2: Stack S -> 2, 1
```

```
     current -> 4
     push 4: Stack S -> 4, 2, 1
     current = NULL

Step 4 pops from S
     a) Pop 4: Stack S -> 2, 1
     b) print "4"
     c) current = NULL /*right of 4 */ and go to step 3
Since current is NULL step 3 doesn't do anything.

Step 4 pops again.
     a) Pop 2: Stack S -> 1
     b) print "2"
     c) current -> 5/*right of 2 */ and go to step 3

Step 3 pushes 5 to stack and makes current NULL
     Stack S -> 5, 1
     current = NULL

Step 4 pops from S
     a) Pop 5: Stack S -> 1
     b) print "5"
     c) current = NULL /*right of 5 */ and go to step 3
Since current is NULL step 3 doesn't do anything

Step 4 pops again.
     a) Pop 1: Stack S -> NULL
     b) print "1"
     c) current -> 3 /*right of 5 */

Step 3 pushes 3 to stack and makes current NULL
     Stack S -> 3
     current = NULL

Step 4 pops from S
     a) Pop 3: Stack S -> NULL
     b) print "3"
     c) current = NULL /*right of 3 */

Traversal is done now as stack S is empty and current is NULL.
```

Implementation:

```
?
#include<stdio.h>
#include<stdlib.h>
#define bool int

/* A binary tree tNode has data, pointer to left child
   and a pointer to right child */
struct tNode
{
   int data;
   struct tNode* left;
   struct tNode* right;
};
```

```
/* Structure of a stack node. Linked List implementation is used for
   stack. A stack node contains a pointer to tree node and a pointer to
   next stack node */
struct sNode
{
  struct tNode *t;
  struct sNode *next;
};

/* Stack related functions */
void push(struct sNode** top_ref, struct tNode *t);
struct tNode *pop(struct sNode** top_ref);
bool isEmpty(struct sNode *top);

/* Iterative function for inorder tree traversal */
void inOrder(struct tNode *root)
{
  /* set current to root of binary tree */
  struct tNode *current = root;
  struct sNode *s = NULL;  /* Initialize stack s */
  bool done = 0;

  while (!done)
  {
    /* Reach the left most tNode of the current tNode */
    if(current !=  NULL)
    {
      /* place pointer to a tree node on the stack before traversing
        the node's left subtree */
      push(&s, current);
      current = current->left;
    }

    /* backtrack from the empty subtree and visit the tNode
       at the top of the stack; however, if the stack is empty,
      you are done */
    else
    {
      if (!isEmpty(s))
      {
        current = pop(&s);
        printf("%d ", current->data);

        /* we have visited the node and its left subtree.
          Now, it's right subtree's turn */
        current = current->right;
      }
      else
        done = 1;
    }
  } /* end of while */
}

/* UTILITY FUNCTIONS */
```

```
/* Function to push an item to sNode*/
void push(struct sNode** top_ref, struct tNode *t)
{
  /* allocate tNode */
  struct sNode* new_tNode =
            (struct sNode*) malloc(sizeof(struct sNode));

  if(new_tNode == NULL)
  {
     printf("Stack Overflow \n");
     getchar();
     exit(0);
  }

  /* put in the data  */
  new_tNode->t  = t;

  /* link the old list off the new tNode */
  new_tNode->next = (*top_ref);

  /* move the head to point to the new tNode */
  (*top_ref)    = new_tNode;
}

/* The function returns true if stack is empty, otherwise false */
bool isEmpty(struct sNode *top)
{
   return (top == NULL)? 1 : 0;
}

/* Function to pop an item from stack*/
struct tNode *pop(struct sNode** top_ref)
{
  struct tNode *res;
  struct sNode *top;

  /*If sNode is empty then error */
  if(isEmpty(*top_ref))
  {
     printf("Stack Underflow \n");
     getchar();
     exit(0);
  }
  else
  {
     top = *top_ref;
     res = top->t;
     *top_ref = top->next;
     free(top);
     return res;
  }
}

/* Helper function that allocates a new tNode with the
   given data and NULL left and right pointers. */
```

```
struct tNode* newtNode(int data)
{
  struct tNode* tNode = (struct tNode*)
                        malloc(sizeof(struct tNode));
  tNode->data = data;
  tNode->left = NULL;
  tNode->right = NULL;

  return(tNode);
}

/* Driver program to test above functions*/
int main()
{

  /* Constructed binary tree is
            1
          /   \
        2      3
      /  \
    4     5
  */
  struct tNode *root = newtNode(1);
  root->left        = newtNode(2);
  root->right       = newtNode(3);
  root->left->left  = newtNode(4);
  root->left->right = newtNode(5);

  inOrder(root);

  getchar();
  return 0;
}
```

Time Complexity: O(n)

References:
http://web.cs.wpi.edu/~cs2005/common/iterative.inorder
http://neural.cs.nthu.edu.tw/jang/courses/cs2351/slide/animation/Iterative%20Inorder%20Traversal.pps

See this post for another approach of Inorder Tree Traversal without recursion and without stack!

INORDER TREE TRAVERSAL WITHOUT RECURSION AND WITHOUT STACK!

Using Morris Traversal, we can traverse the tree without using stack and recursion. The idea of Morris Traversal is based on Threaded Binary Tree. In this traversal, we first create links to Inorder successor and print the data using these links, and finally revert the changes to restore original tree.

```
1. Initialize current as root
```

```
2. While current is not NULL
   If current does not have left child
      a) Print current's data
      b) Go to the right, i.e., current = current->right
   Else
      a) Make current as right child of the rightmost node in current's left
subtree
      b) Go to this left child, i.e., current = current->left
```

Although the tree is modified through the traversal, it is reverted back to its original shape after the completion. Unlike Stack based traversal, no extra space is required for this traversal.

?

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

/* A binary tree tNode has data, pointer to left child
   and a pointer to right child */
struct tNode
{
   int data;
   struct tNode* left;
   struct tNode* right;
};

/* Function to traverse binary tree without recursion and
   without stack */
void MorrisTraversal(struct tNode *root)
{
  struct tNode *current,*pre;

  if(root == NULL)
     return;

  current = root;
  while(current != NULL)
  {
    if(current->left == NULL)
    {
      printf(" %d ", current->data);
      current = current->right;
    }
    else
    {
      /* Find the inorder predecessor of current */
      pre = current->left;
      while(pre->right != NULL && pre->right != current)
        pre = pre->right;

      /* Make current as right child of its inorder predecessor */
      if(pre->right == NULL)
      {
        pre->right = current;
        current = current->left;
```

```
      }

      /* Revert the changes made in if part to restore the original
        tree i.e., fix the right child of predecssor */
      else
      {
        pre->right = NULL;
        printf(" %d ",current->data);
        current = current->right;
      } /* End of if condition pre->right == NULL */
    } /* End of if condition current->left == NULL*/
  } /* End of while */
}

/* UTILITY FUNCTIONS */
/* Helper function that allocates a new tNode with the
   given data and NULL left and right pointers. */
struct tNode* newtNode(int data)
{
  struct tNode* tNode = (struct tNode*)
                        malloc(sizeof(struct tNode));
  tNode->data = data;
  tNode->left = NULL;
  tNode->right = NULL;

  return(tNode);
}

/* Driver program to test above functions*/
int main()
{

  /* Constructed binary tree is
            1
          /   \
        2      3
      /  \
    4     5
  */
  struct tNode *root = newtNode(1);
  root->left        = newtNode(2);
  root->right       = newtNode(3);
  root->left->left  = newtNode(4);
  root->left->right = newtNode(5);

  MorrisTraversal(root);

  getchar();
  return 0;
}
```

References:

www.liacs.nl/~deutz/DS/september28.pdf

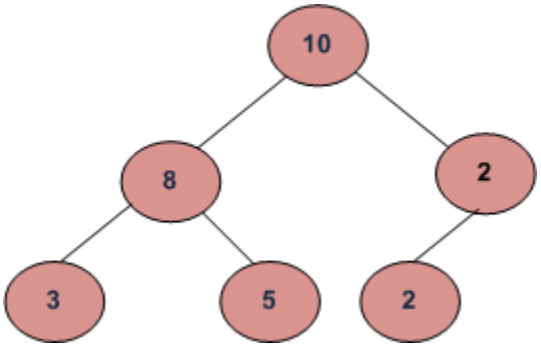
http://comsci.liu.edu/~murali/algo/Morris.htm

www.scss.tcd.ie/disciplines/software_systems/…/HughGibbonsSlides.pdf
http://www.mec.ac.in/resources/notes/notes/ds/thread.htm

Please write comments if you find any bug in above code/algorithm, or want to share more information about stack Morris Inorder Tree Traversal.

ROOT TO LEAF PATH SUM EQUAL TO A GIVEN NUMBER

Given a binary tree and a number, return true if the tree has a root-to-leaf path such that adding up all the values along the path equals the given number. Return false if no such path can be found.



For example, in the above tree root to leaf paths exist with following sums.

21 –> 10 – 8 – 3
23 –> 10 – 8 – 5
14 –> 10 – 2 – 2

So the returned value should be true only for numbers 21, 23 and 14. For any other number, returned value should be false.

Algorithm:
Recursively check if left or right child has path sum equal to (number – value at current node)

Implementation:

?

```
#include<stdio.h>
#include<stdlib.h>
#define bool int

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
   int data;
   struct node* left;
```

```
   struct node* right;
};

/*
 Given a tree and a sum, return true if there is a path from the root
 down to a leaf, such that adding up all the values along the path
 equals the given sum.

 Strategy: subtract the node value from the sum when recurring down,
 and check to see if the sum is 0 when you run out of tree.
*/
bool hasPathSum(struct node* node, int sum)
{
  /* return true if we run out of tree and sum==0 */
  if (node == NULL)
  {
    return(sum == 0);
  }
  else
  {
    /* otherwise check both subtrees */
    int subSum = sum - node->data;
    return(hasPathSum(node->left, subSum) ||
           hasPathSum(node->right, subSum));
  }
}

/* UTILITY FUNCTIONS */
/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newnode(int data)
{
  struct node* node = (struct node*)
                      malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;

  return(node);
}

/* Driver program to test above functions*/
int main()
{

  int sum = 21;

  /* Constructed binary tree is
            10
          /   \
        8      2
      /  \    /
    3     5  2
  */
  struct node *root = newnode(10);
```

```
  root->left        = newnode(8);
  root->right       = newnode(2);
  root->left->left  = newnode(3);
  root->left->right = newnode(5);
  root->right->left = newnode(2);

  if(hasPathSum(root, sum))
    printf("There is a root-to-leaf path with sum %d", sum);
  else
    printf("There is no root-to-leaf path with sum %d", sum);

  getchar();
  return 0;
}
```

Time Complexity: O(n)

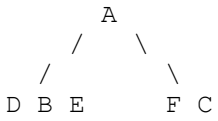
References:
http://cslibrary.stanford.edu/110/BinaryTrees.html

CONSTRUCT TREE FROM GIVEN INORDER AND PREORDER TRAVERSALS

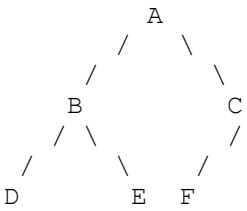
Let us consider the below traversals:

Inorder sequence: D B E A F C
Preorder sequence: A B D E C F

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



We recursively follow above steps and get the following tree.



Algorithm: buildTree()
1) Pick an element from Preorder. Increment a Preorder Index Variable (preIndex in below code) to pick next element in next recursive call.
2) Create a new tree node tNode with the data as picked element.

3) Find the picked element's index in Inorder. Let the index be inIndex.
4) Call buildTree for elements before inIndex and make the built tree as left subtree of tNode.
5) Call buildTree for elements after inIndex and make the built tree as right subtree of tNode.
6) return tNode.

Thanks to Rohini and Tushar for suggesting the code.

?

```
/* program to construct tree using inorder and preorder traversals */
#include<stdio.h>
#include<stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
  char data;
  struct node* left;
  struct node* right;
};

/* Prototypes for utility functions */
int search(char arr[], int strt, int end, char value);
struct node* newNode(char data);

/* Recursive function to construct binary of size len from
   Inorder traversal in[] and Preorder traversal pre[].  Initial values
   of inStrt and inEnd should be 0 and len -1.  The function doesn't
   do any error checking for cases where inorder and preorder
   do not form a tree */
struct node* buildTree(char in[], char pre[], int inStrt, int inEnd)
{
  static int preIndex = 0;

  if(inStrt > inEnd)
     return NULL;

  /* Pick current node from Preorder traversal using preIndex
    and increment preIndex */
  struct node *tNode = newNode(pre[preIndex++]);

  /* If this node has no children then return */
  if(inStrt == inEnd)
    return tNode;

  /* Else find the index of this node in Inorder traversal */
  int inIndex = search(in, inStrt, inEnd, tNode->data);

  /* Using index in Inorder traversal, construct left and
     right subtress */
  tNode->left = buildTree(in, pre, inStrt, inIndex-1);
  tNode->right = buildTree(in, pre, inIndex+1, inEnd);
```

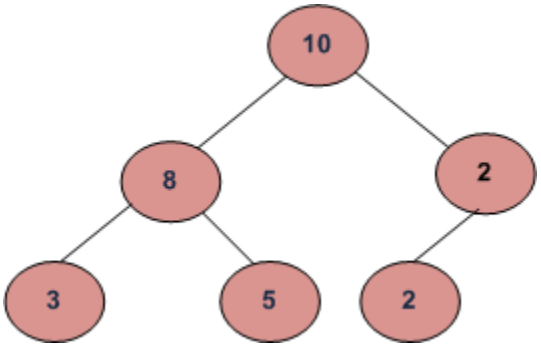
```
  return tNode;
}

/* UTILITY FUNCTIONS */
/* Function to find index of value in arr[start...end]
   The function assumes that value is present in in[] */
int search(char arr[], int strt, int end, char value)
{
  int i;
  for(i = strt; i <= end; i++)
  {
    if(arr[i] == value)
      return i;
  }
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(char data)
{
  struct node* node = (struct node*)malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;

  return(node);
}

/* This funtcion is here just to test buildTree() */
void printInorder(struct node* node)
{
  if (node == NULL)
     return;

  /* first recur on left child */
  printInorder(node->left);

  /* then print the data of node */
  printf("%c ", node->data);

  /* now recur on right child */
  printInorder(node->right);
}

/* Driver program to test above functions */
int main()
{
  char in[] = {'D', 'B', 'E', 'A', 'F', 'C'};
  char pre[] = {'A', 'B', 'D', 'E', 'C', 'F'};
  int len = sizeof(in)/sizeof(in[0]);
  struct node *root = buildTree(in, pre, 0, len - 1);

  /* Let us test the built tree by printing Insorder traversal */
  printf("\n Inorder traversal of the constructed tree is \n");
  printInorder(root);
```

```
  getchar();
}
```

Time Complexity: O(n^2). Worst case occurs when tree is left skewed. Example Preorder and Inorder traversals for worst case are {A, B, C, D} and {D, C, B, A}.

GIVEN A BINARY TREE, PRINT ALL ROOT-TO-LEAF PATHS

For the below example tree, all root-to-leaf paths are:
10 –> 8 –> 3
10 –> 8 –> 5
10 –> 2 –> 2



Algorithm:
Use a path array path[] to store current root to leaf path. Traverse from root to all leaves in top-down fashion. While traversing, store data of all nodes in current path in array path[]. When we reach a leaf node, print the path array.

?

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

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
   int data;
   struct node* left;
   struct node* right;
};

/* Prototypes for funtions needed in printPaths() */
void printPathsRecur(struct node* node, int path[], int pathLen);
void printArray(int ints[], int len);

/*Given a binary tree, print out all of its root-to-leaf
 paths, one per line. Uses a recursive helper to do the work.*/
void printPaths(struct node* node)
{
```

```
  int path[1000];
  printPathsRecur(node, path, 0);
}

/* Recursive helper function -- given a node, and an array containing
 the path from the root node up to but not including this node,
 print out all the root-leaf paths.*/
void printPathsRecur(struct node* node, int path[], int pathLen)
{
  if (node==NULL)
    return;

  /* append this node to the path array */
  path[pathLen] = node->data;
  pathLen++;

  /* it's a leaf, so print the path that led to here  */
  if (node->left==NULL && node->right==NULL)
  {
    printArray(path, pathLen);
  }
  else
  {
    /* otherwise try both subtrees */
    printPathsRecur(node->left, path, pathLen);
    printPathsRecur(node->right, path, pathLen);
  }
}

/* UTILITY FUNCTIONS */
/* Utility that prints out an array on a line. */
void printArray(int ints[], int len)
{
  int i;
  for (i=0; i<len; i++)
  {
    printf("%d ", ints[i]);
  }
  printf("\n");
}

/* utility that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newnode(int data)
{
  struct node* node = (struct node*)
                      malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;

  return(node);
}

/* Driver program to test above functions*/
```

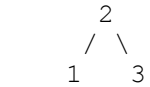
```
int main()
{

  /* Constructed binary tree is
            10
          /   \
        8      2
      /  \    /
    3     5  2
  */
  struct node *root = newnode(10);
  root->left        = newnode(8);
  root->right       = newnode(2);
  root->left->left  = newnode(3);
  root->left->right = newnode(5);
  root->right->left = newnode(2);

  printPaths(root);

  getchar();
  return 0;
}
```

Time Complexity: O(n)

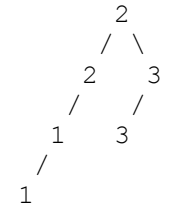
DOUBLE TREE

Write a program that converts a given tree to its Double tree. To create Double tree of the given tree, create a new duplicate for each node, and insert the duplicate as the left child of the original node.

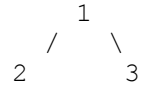
So the tree…

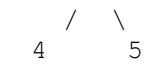


is changed to…

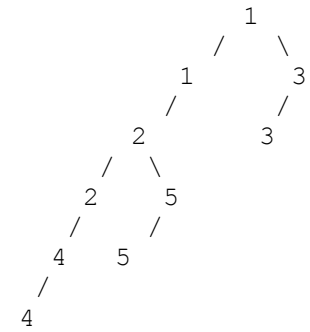


And the tree





is changed to



Algorithm:
Recursively convert the tree to double tree in postorder fashion. For each node, first convert the left subtree of the node, then right subtree, finally create a duplicate node of the node and fix the left child of the node and left child of left child.

Implementation:

?

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

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* function to create a new node of tree and returns pointer */
struct node* newNode(int data);

/* Function to convert a tree to double tree */
void doubleTree(struct node* node)
{
  struct node* oldLeft;

  if (node==NULL) return;

  /* do the subtrees */
  doubleTree(node->left);
  doubleTree(node->right);

  /* duplicate this node to its left */
  oldLeft = node->left;
```

```
  node->left = newNode(node->data);
  node->left->left = oldLeft;
}

/* UTILITY FUNCTIONS TO TEST doubleTree() FUNCTION */
 /* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
  struct node* node = (struct node*)
                      malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;

  return(node);
}

/* Given a binary tree, print its nodes in inorder*/
void printInorder(struct node* node)
{
  if (node == NULL)
    return;
  printInorder(node->left);
  printf("%d ", node->data);
  printInorder(node->right);
}

/* Driver program to test above functions*/
int main()
{

  /* Constructed binary tree is
            1
          /   \
        2      3
      /  \
    4     5
  */
  struct node *root = newNode(1);
  root->left        = newNode(2);
  root->right       = newNode(3);
  root->left->left  = newNode(4);
  root->left->right = newNode(5);

  printf("Inorder traversal of the original tree is \n");
  printInorder(root);

  doubleTree(root);

  printf("\n Inorder traversal of the double tree is \n");
  printInorder(root);

  getchar();
  return 0;
```

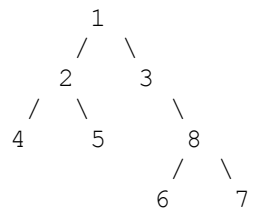
```
}
```

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

MAXIMUM WIDTH OF A BINARY TREE

Given a binary tree, write a function to get the maximum width of the given tree. Width of a tree is maximum of widths of all levels.

Let us consider the below example tree.



For the above tree,
width of level 1 is 1,
width of level 2 is 2,
width of level 3 is 3
width of level 4 is 2.

So the maximum width of the tree is 3.

Algortihm:
There are basically two functions. One is to count nodes at a given level (getWidth), and other is to get the maximum width of the tree(getMaxWidth). getMaxWidth() makes use of getWidth() to get the width of all levels starting from root.

```
/*Function to print level order traversal of tree*/
getMaxWidth(tree)
maxWdth = 0
for i = 1 to height(tree)
  width =   getWidth(tree, i);
  if(width > maxWdth)
      maxWdth  = width
return width
/*Function to get width of a given level */
getWidth(tree, level)
if tree is NULL then return 0;
if level is 1, then return 1;
else if level greater than 1, then
    return getWidth(tree->left, level-1) +
    getWidth(tree->right, level-1);
```

Implementation:

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

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/*Function protoypes*/
int getWidth(struct node* root, int level);
int height(struct node* node);
struct node* newNode(int data);

/* Function to get the maximum width of a binary tree*/
int getMaxWidth(struct node* root)
{
  int maxWidth = 0;
  int width;
  int h = height(root);
  int i;

  /* Get width of each level and compare
     the width with maximum width so far */
  for(i=1; i<=h; i++)
  {
    width = getWidth(root, i);
    if(width > maxWidth)
      maxWidth = width;
  }

  return maxWidth;
}

/* Get width of a given level */
int getWidth(struct node* root, int level)
{

  if(root == NULL)
    return 0;

  if(level == 1)
    return 1;

  else if (level > 1)
    return getWidth(root->left, level-1) +
              getWidth(root->right, level-1);
```

```
}

/* UTILITY FUNCTIONS */
/* Compute the "height" of a tree -- the number of
    nodes along the longest path from the root node
    down to the farthest leaf node.*/
int height(struct node* node)
{
   if (node==NULL)
     return 0;
   else
   {
     /* compute the height of each subtree */
     int lHeight = height(node->left);
     int rHeight = height(node->right);
     /* use the larger one */

     return (lHeight > rHeight)? (lHeight+1): (rHeight+1);
   }
}
/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
  struct node* node = (struct node*)
                       malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;
  return(node);
}
/* Driver program to test above functions*/
int main()
{
  struct node *root = newNode(1);
  root->left        = newNode(2);
  root->right       = newNode(3);
  root->left->left  = newNode(4);
  root->left->right = newNode(5);
  root->right->right = newNode(8);
  root->right->right->left  = newNode(6);
  root->right->right->right  = newNode(7);

  /*
   Constructed bunary tree is:
          1
        /  \
       2    3
     /  \     \
    4   5     8
              /  \
             6   7
  */
  printf("Maximum width is %d \n", getMaxWidth(root));
  getchar();
  return 0;
```

```
}
```

FOLDABLE BINARY TREES

Question: Given a binary tree, find out if the tree can be folded or not.

A tree can be folded if left and right subtrees of the tree are structure wise mirror image of each other. An empty tree is considered as foldable.

```
Consider the below trees:
(a) and (b) can be folded.
(c) and (d) cannot be folded.

(a)
       10
     /    \
    7      15
     \    /
      9  11

(b)
        10
       /  \
      7    15
     /      \
    9       11

(c)
        10
       /  \
      7   15
     /    /
    5   11

(d)

         10
       /   \
      7     15
    /  \    /
   9   10  12
```

Method 1 (Change Left subtree to its Mirror and compare it with Right subtree)
Algorithm: isFoldable(root)

```
1) If tree is empty, then return true.
2) Convert the left subtree to its mirror image
    mirror(root->left); /* See this post */
3) Check if the structure of left subtree and right subtree is same
   and store the result.
    res = isStructSame(root->left, root->right); /*isStructSame()
        recursively compares structures of two subtrees and returns
        true if structures are same */
4) Revert the changes made in step (2) to get the original tree.
```

```
    mirror(root->left);
5) Return result res stored in step 2.
```

Thanks to ajaym for suggesting this approach.

?

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

/* You would want to remove below 3 lines if your compiler
   supports bool, true and false */
#define bool int
#define true 1
#define false 0

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
  int data;
  struct node* left;
  struct node* right;
};

/* converts a tree to its mrror image */
void mirror(struct node* node);

/* returns true if structure of two trees a and b is same
   Only structure is considered for comparison, not data! */
bool isStructSame(struct node *a, struct node *b);

/* Returns true if the given tree is foldable */
bool isFoldable(struct node *root)
{
  bool res;

  /* base case */
  if(root == NULL)
    return true;

  /* convert left subtree to its mirror */
  mirror(root->left);

  /* Compare the structures of the right subtree and mirrored
    left subtree */
  res = isStructSame(root->left, root->right);

  /* Get the originial tree back */
  mirror(root->left);

  return res;
}
```

```
bool isStructSame(struct node *a, struct node *b)
{
  if (a == NULL && b == NULL)
  {  return true; }
  if ( a != NULL && b != NULL &&
       isStructSame(a->left, b->left) &&
       isStructSame(a->right, b->right)
     )
  {  return true; }

  return false;
}

/* UTILITY FUNCTIONS */
/* Change a tree so that the roles of the  left and
    right pointers are swapped at every node.
    See http://geeksforgeeks.org/?p=662 for details */
void mirror(struct node* node)
{
  if (node==NULL)
    return;
  else
  {
    struct node* temp;

    /* do the subtrees */
    mirror(node->left);
    mirror(node->right);

    /* swap the pointers in this node */
    temp        = node->left;
    node->left  = node->right;
    node->right = temp;
  }
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
  struct node* node = (struct node*)
                      malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;

  return(node);
}

/* Driver program to test mirror() */
int main(void)
{
  /* The constructed binary tree is
         1
       /   \
```

```
      2     3
      \    /
       4  5
  */
  struct node *root = newNode(1);
  root->left        = newNode(2);
  root->right       = newNode(3);
  root->right->left  = newNode(4);
  root->left->right = newNode(5);

  if(isFoldable(root) == 1)
  { printf("\n tree is foldable"); }
  else
  { printf("\n tree is not foldable"); }

  getchar();
  return 0;
}
```

Time complexity: O(n)

Method 2 (Check if Left and Right subtrees are Mirror)
There are mainly two functions:

// Checks if tree can be folded or not

```
IsFoldable(root)
1) If tree is empty then return true
2) Else check if left and right subtrees are structure wise mirrors of
    each other. Use utility function IsFoldableUtil(root->left,
    root->right) for this.
```

// Checks if n1 and n2 are mirror of each other.

```
IsFoldableUtil(n1, n2)
1) If both trees are empty then return true.
2) If one of them is empty and other is not then return false.
3) Return true if following conditions are met
   a) n1->left is mirror of n2->right
   b) n1->right is mirror of n2->left

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

/* You would want to remove below 3 lines if your compiler
   supports bool, true and false */
#define bool int
#define true 1
#define false 0

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
```

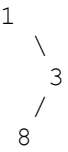
```
{
  int data;
  struct node* left;
  struct node* right;
};

/* A utility function that checks if trees with roots as n1 and n2
  are mirror of each other */
bool IsFoldableUtil(struct node *n1, struct node *n2);

/* Returns true if the given tree can be folded */
bool IsFoldable(struct node *root)
{
     if (root == NULL)
     {  return true;  }

     return IsFoldableUtil(root->left, root->right);
}

/* A utility function that checks if trees with roots as n1 and n2
  are mirror of each other */
bool IsFoldableUtil(struct node *n1, struct node *n2)
{
    /* If both left and right subtrees are NULL,
      then return true */
    if (n1 == NULL && n2 == NULL)
    {  return true;  }

    /* If one of the trees is NULL and other is not,
      then return false */
    if (n1 == NULL || n2 == NULL)
    {  return false; }

    /* Otherwise check if left and right subtrees are mirrors of
       their counterparts */
    return IsFoldableUtil(n1->left, n2->right) &&
           IsFoldableUtil(n1->right, n2->left);
}

/*UTILITY FUNCTIONS */
/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
  struct node* node = (struct node*)
                      malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;

  return(node);
}

/* Driver program to test mirror() */
int main(void)
```

```
{
  /* The constructed binary tree is
         1
       /   \
      2     3
      \    /
       4  5
  */
  struct node *root = newNode(1);
  root->left        = newNode(2);
  root->right       = newNode(3);
  root->left->right  = newNode(4);
  root->right->left = newNode(5);

  if(IsFoldable(root) == true)
  { printf("\n tree is foldable"); }
  else
  { printf("\n tree is not foldable"); }

  getchar();
  return 0;
}
```

PRINT NODES AT K DISTANCE FROM ROOT

Given a root of a tree, and an integer k. Print all the nodes which are at k distance from root.

For example, in the below tree, 4, 5 & 8 are at distance 2 from root.



The problem can be solved using recursion. Thanks to eldho for suggesting the solution.

?

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

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
   int data;
   struct node* left;
   struct node* right;
};

void printKDistant(node *root , int k)
{
   if(root == NULL)
```

```
      return;
   if( k == 0 )
   {
      printf( "%d ", root->data );
      return ;
   }
   else
   {
      printKDistant( root->left, k-1 ) ;
      printKDistant( root->right, k-1 ) ;
   }
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
  struct node* node = (struct node*)
                      malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;

  return(node);
}

/* Driver program to test above functions*/
int main()
{

  /* Constructed binary tree is
            1
          /   \
        2      3
      /  \    /
    4     5  8
  */
  struct node *root = newNode(1);
  root->left        = newNode(2);
  root->right       = newNode(3);
  root->left->left  = newNode(4);
  root->left->right = newNode(5);
  root->right->left = newNode(8);

  printKDistant(root, 2);

  getchar();
  return 0;
}
```

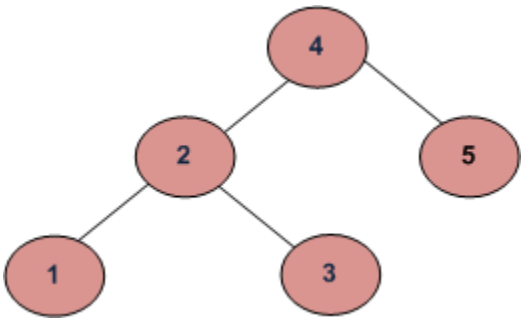
The above program prints 4, 5 and 8.

Time Complexity: O(n) where n is number of nodes in the given binary tree.

SORTED ORDER PRINTING OF A GIVEN ARRAY THAT REPRESENTS A BST

Given an array that stores a complete Binary Search Tree, write a function that efficiently prints the given array in ascending order.

For example, given an array [4, 2, 5, 1, 3], the function should print 1, 2, 3, 4, 5



Solution:
Inorder traversal of BST prints it in ascending order. The only trick is to modify recursion termination condition in standard Inorder Tree Traversal.

Implementation:

?

```
#include<stdio.h>

void printSorted(int arr[], int start, int end)
{
  if(start > end)
    return;

  // print left subtree
  printSorted(arr, start*2 + 1, end);

  // print root
  printf("%d  ", arr[start]);

  // print right subtree
  printSorted(arr, start*2 + 2, end);
}

int main()
{
  int arr[] = {4, 2, 5, 1, 3};
  int arr_size = sizeof(arr)/sizeof(int);
  printSorted(arr, 0, arr_size-1);
  getchar();
  return 0;
}
```

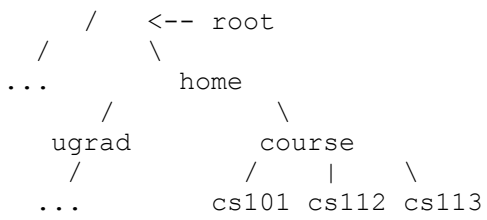
Time Complexity: O(n)

APPLICATIONS OF TREE DATA STRUCTURE

Why Tree?
Unlike Array and Linked List, which are linear data structures, tree is hierarchical (or non-linear) data structure.

1) One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:

file system
———



2) If we organize keys in form of a tree (with some ordering e.g., BST), we can search for a given key in moderate time (quicker than Linked List and slower than arrays). Self-balancing search trees like AVL and Red-Black trees guarantee an upper bound of O(Logn) for search.

3) We can insert/delete keys in moderate time (quicker than Arrays and slower than Unordered Linked Lists). Self-balancing search trees like AVL and Red-Black trees guarantee an upper bound of O(Logn) for insertion/deletion.

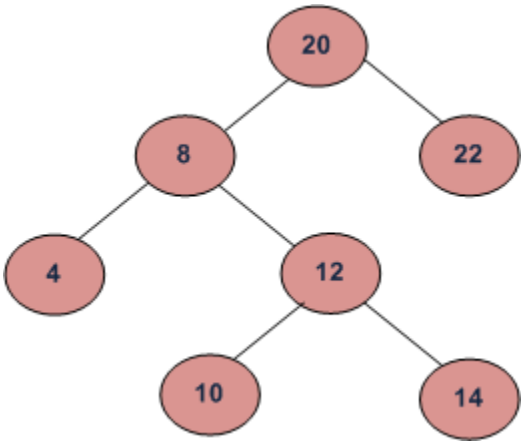
4) Like Linked Lists and unlike Arrays, Pointer implementation of trees don't have an upper limit on number of nodes as nodes are linked using pointers.

As per Wikipedia, following are the common uses of tree.
1. Manipulate hierarchical data.
2. Make information easy to search (see tree traversal).
3. Manipulate sorted lists of data.
4. As a workflow for compositing digital images for visual effects.
5. Router algorithms

INORDER SUCCESSOR IN BINARY SEARCH TREE

In Binary Tree, Inorder successor of a node is the next node in Inorder traversal of the Binary Tree. Inorder Successor is NULL for the last node in Inoorder traversal.
In Binary Search Tree, Inorder Successor of an input node can also be defined as the node with

the smallest key greater than the key of input node. So, it is sometimes important to find next node in sorted order.



In the above diagram, inorder successor of **8** is **10**, inorder successor of **10** is **12** and inorder successor of **14** is **20**.

Algorithm to find Inorder Successor
Algorithm is divided into two cases on the basis of right subtree of the input node being empty or not.

Input: *node, root* // *node* is the node whose Inorder successor is needed.
output: *succ* // *succ* is Inorder successor of *node*.

1) If right subtree of *node* is not *NULL*, then *succ* lies 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 *node* is NULL, then *succ* is 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*.

Implementation
Note that the function to find InOrder Successor is highlighted (with gray background) in below code.

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

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
    struct node* parent;
```

```
};

struct node * minValue(struct node* node);

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;
}

/* Given a non-empty binary search tree, return the minimum data
    value found in that tree. Note that the entire tree does not need
    to be searched. */
struct node * minValue(struct node* node) {
  struct node* current = node;

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

/* Helper function that allocates a new node with the given data and
    NULL left and right pointers. */
struct node* newNode(int data)
{
  struct node* node = (struct node*)
                      malloc(sizeof(struct node));
  node->data   = data;
  node->left   = NULL;
  node->right  = NULL;
  node->parent = NULL;

  return(node);
}

/* Give a binary search tree and a number, inserts a new node with
    the given number in the correct place in the tree. Returns the new
    root pointer which the caller should then use (the standard trick to
    avoid using reference parameters). */
struct node* insert(struct node* node, int data)
{
  /* 1. If the tree is empty, return a new,
      single node */
```

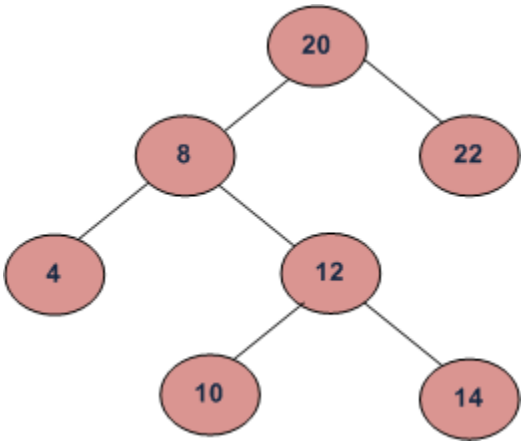
```
  if (node == NULL)
    return(newNode(data));
  else
  {
    struct node *temp;

    /* 2. Otherwise, recur down the tree */
    if (data <= node->data)
    {
         temp = insert(node->left, data);
         node->left  = temp;
         temp->parent= node;
    }
    else
    {
        temp = insert(node->right, data);
        node->right = temp;
        temp->parent = node;
    }

    /* return the (unchanged) node pointer */
    return node;
  }
}

/* Driver program to test above functions*/
int main()
{
  struct node* root = NULL, *temp, *succ, *min;

  //creating the tree given in the above diagram
  root = insert(root, 20);
  root = insert(root, 8);
  root = insert(root, 22);
  root = insert(root, 4);
  root = insert(root, 12);
  root = insert(root, 10);
  root = insert(root, 14);
  temp = root->left->right->right;

  succ =  inOrderSuccessor(root, temp);
  if(succ !=  NULL)
    printf("\n Inorder Successor of %d is %d ", temp->data, succ->data);
  getchar();
  return 0;
}
```

Output of the above program:
Inorder Successor of 14 is 20

FIND K-TH SMALLEST ELEMENT IN BST (ORDER STATISTICS IN BST)

Given root of binary search tree and K as input, find K-th smallest element in BST. Related Post

For example, in the following BST, if k = 3, then output should be 10, and if k = 5, then output should be 14.



Method 1: Using Inorder Traversal.

Inorder traversal of BST retrieves elements of tree in the sorted order. The inorder traversal uses stack to store to be explored nodes of tree (threaded tree avoids stack and recursion for traversal, see this post). The idea is to keep track of popped elements which participate in the order statics. Hypothetical algorithm is provided below,

Time complexity: O(n) where n is total nodes in tree..

Algorithm:

```
/* initialization */
pCrawl = root
set initial stack element as NULL (sentinal)

/* traverse upto left extreme */
while(pCrawl is valid )
   stack.push(pCrawl)
   pCrawl = pCrawl.left

/* process other nodes */
while( pCrawl = stack.pop() is valid )
   stop if sufficient number of elements are popped.
   if( pCrawl.right is valid )
      pCrawl = pCrawl.right
      while( pCrawl is valid )
         stack.push(pCrawl)
         pCrawl = pCrawl.left
```

Implementation:

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

```
#define ARRAY_SIZE(arr) sizeof(arr)/sizeof(arr[0])

/* just add elements to test */
/* NOTE: A sorted array results in skewed tree */
int ele[] = { 20, 8, 22, 4, 12, 10, 14 };

/* same alias */
typedef struct node_t node_t;

/* Binary tree node */
struct node_t
{
    int data;

    node_t* left;
    node_t* right;
};

/* simple stack that stores node addresses */
typedef struct stack_t stack_t;

/* initial element always NULL, uses as sentinal */
struct stack_t
{
    node_t*  base[ARRAY_SIZE(ele) + 1];
    int   stackIndex;
};

/* pop operation of stack */
node_t *pop(stack_t *st)
{
    node_t *ret = NULL;

    if( st && st->stackIndex > 0 )
    {
        ret = st->base[st->stackIndex];
        st->stackIndex--;
    }

    return ret;
}

/* push operation of stack */
void push(stack_t *st, node_t *node)
{
    if( st )
    {
        st->stackIndex++;
        st->base[st->stackIndex] = node;
    }
}

/* Iterative insertion
   Recursion is least preferred unless we gain something
```

```
*/
node_t *insert_node(node_t *root, node_t* node)
{
    /* A crawling pointer */
    node_t *pTraverse = root;
    node_t *currentParent = root;

    // Traverse till appropriate node
    while(pTraverse)
    {
        currentParent = pTraverse;

        if( node->data < pTraverse->data )
        {
            /* left subtree */
            pTraverse = pTraverse->left;
        }
        else
        {
            /* right subtree */
            pTraverse = pTraverse->right;
        }
    }

    /* If the tree is empty, make it as root node */
    if( !root )
    {
        root = node;
    }
    else if( node->data < currentParent->data )
    {
        /* Insert on left side */
        currentParent->left = node;
    }
    else
    {
        /* Insert on right side */
        currentParent->right = node;
    }

    return root;
}

/* Elements are in an array. The function builds binary tree */
node_t* binary_search_tree(node_t *root, int keys[], int const size)
{
    int iterator;
    node_t *new_node = NULL;

    for(iterator = 0; iterator < size; iterator++)
    {
        new_node = (node_t *)malloc( sizeof(node_t) );

        /* initialize */
        new_node->data   = keys[iterator];
```

```
        new_node->left   = NULL;
        new_node->right  = NULL;

        /* insert into BST */
        root = insert_node(root, new_node);
    }

    return root;
}

node_t *k_smallest_element_inorder(stack_t *stack, node_t *root, int k)
{
    stack_t *st = stack;
    node_t *pCrawl = root;

    /* move to left extremen (minimum) */
    while( pCrawl )
    {
        push(st, pCrawl);
        pCrawl = pCrawl->left;
    }

    /* pop off stack and process each node */
    while( pCrawl = pop(st) )
    {
        /* each pop operation emits one element
           in the order
        */
        if( !--k )
        {
            /* loop testing */
            st->stackIndex = 0;
            break;
        }

        /* there is right subtree */
        if( pCrawl->right )
        {
            /* push the left subtree of right subtree */
            pCrawl = pCrawl->right;
            while( pCrawl )
            {
                push(st, pCrawl);
                pCrawl = pCrawl->left;
            }

            /* pop off stack and repeat */
        }
    }

    /* node having k-th element or NULL node */
    return pCrawl;
}

/* Driver program to test above functions */
```

```
int main(void)
{
    node_t* root = NULL;
    stack_t stack = { {0}, 0 };
    node_t *kNode = NULL;

    int k = 5;

    /* Creating the tree given in the above diagram */
    root = binary_search_tree(root, ele, ARRAY_SIZE(ele));

    kNode = k_smallest_element_inorder(&stack, root, k);

    if( kNode )
    {
        printf("kth smallest elment for k = %d is %d", k, kNode->data);
    }
    else
    {
        printf("There is no such element");
    }

    getchar();
    return 0;
}
```

Method 2: Augmented Tree Data Structure.

The idea is to maintain rank of each node. We can keep track of elements in a subtree of any node while building the tree. Since we need K-th smallest element, we can maintain number of elements of left subtree in every node.

Assume that the root is having N nodes in its left subtree. If K = N + 1, root is K-th node. If K < N, we will continue our search (recursion) for the Kth smallest element in the left subtree of root. If K > N + 1, we continue our search in the right subtree for the (K – N – 1)-th smallest element. Note that we need the count of elements in left subtree only.

Time complexity: O(n) where n is total nodes in tree.

Algorithm:

```
start:
if K = root.leftElement + 1
   root node is the K th node.
   goto stop
else if K > root.leftElements
   K = K - (root.leftElements + 1)
   root = root.right
   goto start
else
   root = root.left
   goto srart
```

```
stop:
```

Implementation:

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

#define ARRAY_SIZE(arr) sizeof(arr)/sizeof(arr[0])

typedef struct node_t node_t;

/* Binary tree node */
struct node_t
{
    int data;
    int lCount;

    node_t* left;
    node_t* right;
};

/* Iterative insertion
   Recursion is least preferred unless we gain something
*/
node_t *insert_node(node_t *root, node_t* node)
{
    /* A crawling pointer */
    node_t *pTraverse = root;
    node_t *currentParent = root;

    // Traverse till appropriate node
    while(pTraverse)
    {
        currentParent = pTraverse;

        if( node->data < pTraverse->data )
        {
            /* We are branching to left subtree
               increment node count */
            pTraverse->lCount++;
            /* left subtree */
            pTraverse = pTraverse->left;
        }
        else
        {
            /* right subtree */
            pTraverse = pTraverse->right;
        }
    }

    /* If the tree is empty, make it as root node */
    if( !root )
    {
```

```
        root = node;
    }
    else if( node->data < currentParent->data )
    {
        /* Insert on left side */
        currentParent->left = node;
    }
    else
    {
        /* Insert on right side */
        currentParent->right = node;
    }

    return root;
}

/* Elements are in an array. The function builds binary tree */
node_t* binary_search_tree(node_t *root, int keys[], int const size)
{
    int iterator;
    node_t *new_node = NULL;

    for(iterator = 0; iterator < size; iterator++)
    {
        new_node = (node_t *)malloc( sizeof(node_t) );

        /* initialize */
        new_node->data   = keys[iterator];
        new_node->lCount = 0;
        new_node->left   = NULL;
        new_node->right  = NULL;

        /* insert into BST */
        root = insert_node(root, new_node);
    }

    return root;
}

int k_smallest_element(node_t *root, int k)
{
    int ret = -1;

    if( root )
    {
        /* A crawling pointer */
        node_t *pTraverse = root;

        /* Go to k-th smallest */
        while(pTraverse)
        {
            if( (pTraverse->lCount + 1) == k )
            {
                ret = pTraverse->data;
                break;
```

```
            }
            else if( k > pTraverse->lCount )
            {
                /*  There are less nodes on left subtree
                    Go to right subtree */
                k = k - (pTraverse->lCount + 1);
                pTraverse = pTraverse->right;
            }
            else
            {
                /* The node is on left subtree */
                pTraverse = pTraverse->left;
            }
        }
    }

    return ret;
}

/* Driver program to test above functions */
int main(void)
{
    /* just add elements to test */
    /* NOTE: A sorted array results in skewed tree */
    int ele[] = { 20, 8, 22, 4, 12, 10, 14 };
    int i;
    node_t* root = NULL;

    /* Creating the tree given in the above diagram */
    root = binary_search_tree(root, ele, ARRAY_SIZE(ele));

    /*  It should print the sorted array */
    for(i = 1; i <= ARRAY_SIZE(ele); i++)
    {
        printf("\n kth smallest elment for k = %d is %d",
                 i, k_smallest_element(root, i));
    }

    getchar();
    return 0;
}
```

CHECK IF A GIVEN BINARY TREE IS SUMTREE

Write a function that returns true if the given Binary Tree is SumTree else false. A SumTree is a Binary Tree where the value of a node is equal to sum of the nodes present in its left subtree and right subtree. An empty tree is SumTree and sum of an empty tree can be considered as 0. A leaf node is also considered as SumTree.

Following is an example of SumTree.

```
          26
        /   \
```

```
      10     3
    /    \     \
  4      6      3
```

Method 1 (Simple)
Get the sum of nodes in left subtree and right subtree. Check if the sum calculated is equal to root's data. Also, recursively check if the left and right subtrees are SumTrees.

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

/* A binary tree node has data, left child and right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* A utility function to get the sum of values in tree with root
  as root */
int sum(struct node *root)
{
   if(root == NULL)
     return 0;
   return sum(root->left) + root->data + sum(root->right);
}

/* returns 1 if sum property holds for the given
    node and both of its children */
int isSumTree(struct node* node)
{
    int ls, rs;

    /* If node is NULL or it's a leaf node then
       return true */
    if(node == NULL ||
            (node->left == NULL && node->right == NULL))
        return 1;

   /* Get sum of nodes in left and right subtrees */
   ls = sum(node->left);
   rs = sum(node->right);

   /* if the node and both of its children satisfy the
       property return 1 else 0*/
    if((node->data == ls + rs)&&
            isSumTree(node->left) &&
            isSumTree(node->right))
        return 1;

   return 0;
}

/*
 Helper function that allocates a new node
```

```
 with the given data and NULL left and right
 pointers.
*/
struct node* newNode(int data)
{
    struct node* node =
        (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return(node);
}

/* Driver program to test above function */
int main()
{
    struct node *root  = newNode(26);
    root->left         = newNode(10);
    root->right        = newNode(3);
    root->left->left   = newNode(4);
    root->left->right  = newNode(6);
    root->right->right = newNode(3);
    if(isSumTree(root))
        printf("The given tree is a SumTree ");
    else
        printf("The given tree is not a SumTree ");

    getchar();
    return 0;
}
```

Time Complexity: O(n^2) in worst case. Worst case occurs for a skewed tree.

Method 2 (Tricky)
The Method 1 uses sum() to get the sum of nodes in left and right subtrees. The method 2 uses following rules to get the sum directly.
1) If the node is a leaf node then sum of subtree rooted with this node is equal to value of this node.
2) If the node is not a leaf node then sum of subtree rooted with this node is twice the value of this node (Assuming that the tree rooted with this node is SumTree).

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

/* A binary tree node has data, left child and right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Utillity function to check if the given node is leaf or not */
int isLeaf(struct node *node)
```

```
{
    if(node == NULL)
        return 0;
    if(node->left == NULL && node->right == NULL)
        return 1;
    return 0;
}

/* returns 1 if SumTree property holds for the given
    tree */
int isSumTree(struct node* node)
{
    int ls; // for sum of nodes in left subtree
    int rs; // for sum of nodes in right subtree

    /* If node is NULL or it's a leaf node then
       return true */
    if(node == NULL || isLeaf(node))
        return 1;

    if( isSumTree(node->left) && isSumTree(node->right))
    {
        // Get the sum of nodes in left subtree
        if(node->left == NULL)
            ls = 0;
        else if(isLeaf(node->left))
            ls = node->left->data;
        else
            ls = 2*(node->left->data);

        // Get the sum of nodes in right subtree
        if(node->right == NULL)
            rs = 0;
        else if(isLeaf(node->right))
            rs = node->right->data;
        else
            rs = 2*(node->right->data);

        /* If root's data is equal to sum of nodes in left
           and right subtrees then return 1 else return 0*/
        return(node->data == ls + rs);
    }

    return 0;
}

/* Helper function that allocates a new node
 with the given data and NULL left and right
 pointers.
*/
struct node* newNode(int data)
{
    struct node* node =
        (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
```

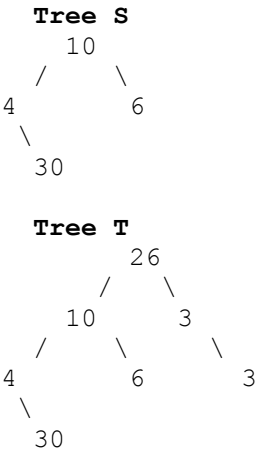
```
    return(node);
}

/* Driver program to test above function */
int main()
{
    struct node *root  = newNode(26);
    root->left         = newNode(10);
    root->right        = newNode(3);
    root->left->left   = newNode(4);
    root->left->right  = newNode(6);
    root->right->right = newNode(3);
    if(isSumTree(root))
        printf("The given tree is a SumTree ");
    else
        printf("The given tree is not a SumTree ");

    getchar();
    return 0;
}
```

Time Complexity: O(n)

CHECK IF A BINARY TREE IS SUBTREE OF ANOTHER BINARY TREE

Given two binary trees, check if the first tree is subtree of the second one. A subtree of a tree T is a tree S consisting of a node in T and all of its descendants in T. The subtree corresponding to the root node is the entire tree; the subtree corresponding to any other node is called a proper subtree.

For example, in the following case, tree S is a subtree of tree T.



Solution: Traverse the tree T in inorder fashion. For every visited node in the traversal, see if the subtree rooted with this node is identical to S.

Following is C implementation for this.

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

/* A binary tree node has data, left child and right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* A utility function to check whether trees with roots as root1 and
   root2 are identical or not */
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);
}

/* Helper function that allocates a new node with the given data
   and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node =
        (struct node*)malloc(sizeof(struct node));
    node->data  = data;
    node->left  = NULL;
```

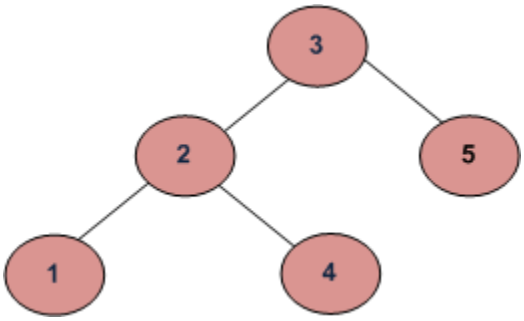
```
    node->right = NULL;
    return(node);
}

/* Driver program to test above function */
int main()
{
    /* Construct the following tree
              26
            /   \
          10     3
        /    \     \
      4      6      3
       \
        30
    */
    struct node *T        = newNode(26);
    T->right              = newNode(3);
    T->right->right       = newNode(3);
    T->left               = newNode(10);
    T->left->left         = newNode(4);
    T->left->left->right  = newNode(30);
    T->left->right        = newNode(6);

    /* Construct the following tree
          10
        /    \
      4      6
       \
        30
    */
    struct node *S    = newNode(10);
    S->right          = newNode(6);
    S->left           = newNode(4);
    S->left->right    = newNode(30);

    if( isSubtree(T, S) )
        printf("Tree S is subtree of tree T");
    else
        printf("Tree S is not a subtree of tree T");

    getchar();
    return 0;
}
```

Output:
Tree S is subtree of tree T

GET LEVEL OF A NODE IN A BINARY TREE

Given a Binary Tree and a key, write a function that returns level of the key.

For example, consider the following tree. If the input key is 3, then your function should return 1. If the input key is 4, then your function should return 3. And for key which is not present in key, then your function should return 0.



Thanks to prandeey for suggesting the following solution.

The idea is to start from the root and level as 1. If the key matches with root's data, return level. Else recursively call for left and right subtrees with level as level + 1.

?

```
#include<stdio.h>

/* A tree node structure */
struct node
{
  int data;
  struct node *left;
  struct node *right;
};

/*
   Helper function for getLevel().  It returns level of the data if data is
   present in tree, otherwise returns 0.
*/
int getLevelUtil(struct node *node, int data, int level)
{
  if ( node == NULL )
    return 0;

  if ( node->data == data )
    return level;

  return getLevelUtil ( node->left, data, level+1) |
         getLevelUtil ( node->right, data, level+1);
}

/* Returns level of given data value */
int getLevel(struct node *node, int data)
{
  return getLevelUtil(node,data,1);
}
```

```
/* Utility function to create a new Binary Tree node */
struct node* newNode(int data)
{
  struct node *temp = new struct node;
  temp->data = data;
  temp->left = NULL;
  temp->right = NULL;

  return temp;
}

/* Driver function to test above functions */
int main()
{
  struct node *root = new struct node;
  int x;

  /* Constructing tree given in the above figure */
  root = newNode(3);
  root->left = newNode(2);
  root->right = newNode(5);
  root->left->left = newNode(1);
  root->left->right = newNode(4);

  x = 3;
  printf(" Level of %d is %d", x, getLevel(root, x));

  x = 6;
  printf("\n Level of %d is %d", x, getLevel(root, x));

  getchar();
  return 0;
}
```

Output:
Level of 3 is 1
Level of 6 is 0

Time Complexity: O(n) where n is the number of nodes in the given Binary Tree.

TOURNAMENT TREE (WINNER TREE) AND BINARY HEAP

Given a team of N players. How many minimum games are required to find second best player?

We can use adversary arguments based on tournament tree (Binary Heap).

Tournament tree is a form of min (max) heap which is a complete binary tree. Every external node represents a player and internal node represents winner. In a tournament tree every internal node contains winner and every leaf node contains one player.

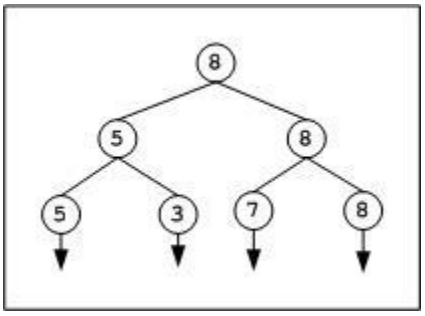
There will be N – 1 internal nodes in a binary tree with N leaf (external) nodes. For details see this post (put n = 2 in equation given in the post).

It is obvious that to select the best player among N players, (N – 1) players to be eliminated, i.e. we need minimum of (N – 1) games (comparisons). Mathematically we can prove it. In a binary tree I = E – 1, where I is number of internal nodes and E is number of external nodes. It means to find maximum or minimum element of an array, we need N – 1 (internal nodes) comparisons.

Second Best Player

The information explored during best player selection can be used to minimize the number of comparisons in tracing the next best players. For example, we can pick second best player in **(N + $\log_2$N – 2)** comparisons. For details read this comment.

The following diagram displays a tournament tree (*winner tree*) as a max heap. Note that the concept of *loser tree* is different.



The above tree contains 4 leaf nodes that represent players and have 3 levels 0, 1 and 2. Initially 2 games are conducted at level 2, one between 5 and 3 and another one between 7 and 8. In the next move, one more game is conducted between 5 and 8 to conclude the final winner. Overall we need 3 comparisons. For second best player we need to trace the candidates participated with final winner, that leads to 7 as second best.

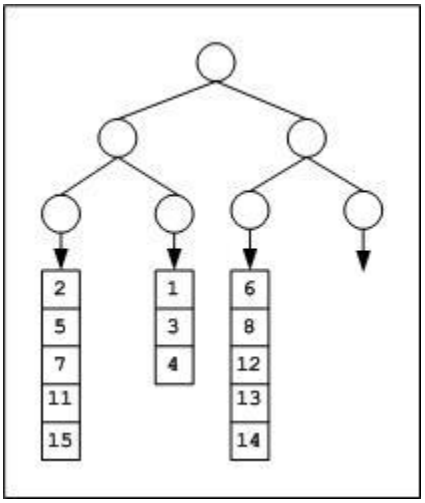
Median of Sorted Arrays

Tournament tree can effectively be used to find median of sorted arrays. Assume, given M sorted arrays of equal size L (for simplicity). We can attach all these sorted arrays to the tournament tree, one array per leaf. We need a tree of height **CEIL ($\log_2$M)** to have atleast M external nodes.

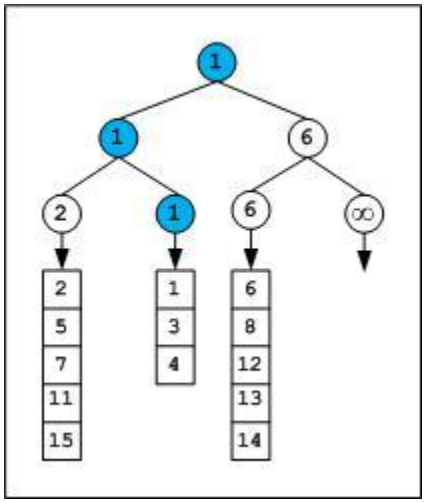
Consider an example. Given 3 (M = 3) sorted integer arrays of maximum size 5 elements.

```
{ 2, 5, 7, 11, 15 } ---- Array1
{1, 3, 4} ---- Array2
{6, 8, 12, 13, 14} ---- Array3
```

What should be the height of tournament tree? We need to construct a tournament tree of height $\log_2 3 .= 1.585 = 2$ rounded to next integer. A binary tree of height 2 will have 4 leaves to which we can attach the arrays as shown in the below figure.



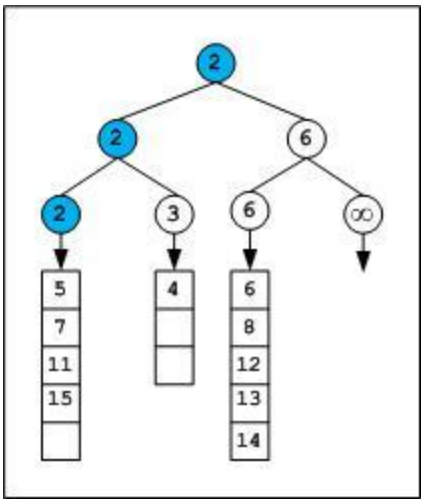
After the first tournament, the tree appears as below,



We can observe that the winner is from Array2. Hence the next element from Array2 will dive-in and games will be played along the winner path of previous tournament.

Note that infinity is used as sentinel element. Based on data being hold in nodes we can select the sentinel character. For example we usually store the pointers in nodes rather than keys, so NULL can serve as sentinel. If any of the array exhausts we will fill the corresponding leaf and upcoming internal nodes with sentinel.

After the second tournament, the tree appears as below,



The next winner is from Array1, so next element of Array1 array which is 5 will dive-in to the next round, and next tournament played along the path of 2.

The tournaments can be continued till we get median element which is (5+3+5)/2 = 7th element. Note that there are even better algorithms for finding median of union of sorted arrays, for details see the related links given below.

In general with M sorted lists of size L_1, L_2 … L_m requires time complexity of $O((L_1 + L_2 + \ldots + L_m) * logM)$ to merge all the arrays, and $O(m*logM)$ time to find median, where m is median position.

Select smallest one million elements from one billion unsorted elements: Read the Source.

As a simple solution, we can sort the billion numbers and select first one million.

On a limited memory system sorting billion elements and picking the first one million seems to be impractical. We can use tournament tree approach. At any time only elements of tree to be in memory.

Split the large array (perhaps stored on disk) into smaller size arrays of size one million each (or even smaller that can be sorted by the machine). Sort these 1000 small size arrays and store them on disk as individual files. Construct a tournament tree which can have atleast 1000 leaf nodes (tree to be of height 10 since $2^9 < 1000 < 2^{10}$, if the individual file size is even smaller we will need more leaf nodes). Every leaf node will have an engine that picks next element from the sorted file stored on disk. We can play the tournament tree game to extract first one million elements.

Total cost = sorting 1000 lists of one million each + tree construction + tournaments

Implementation

We need to build the tree (heap) in bottom-up manner. All the leaf nodes filled first. Start at the left extreme of tree and fill along the breadth (i.e. from 2^{k-1} to $2^k - 1$ where k is depth of tree) and play the game. After practicing with few examples it will be easy to write code. We will have code in an upcoming article.

DECISION TREES – FAKE (COUNTERFEIT) COIN PUZZLE

Let us solve the "fake coin" puzzle using decision trees. There are two variants of the puzzle. One such variant is given below (try to solve on your own, assume N = 8), another variant is given as next problem.

*We are provided a two pan fair balance and N identically looking coins, out of which only one coin **may be** defective. How can we trace which coin, if any, is odd one, and also determine whether it is lighter or heavier in minimum number of trials in the worst case?*

Let us start with relatively simple examples. After reading every problem try to solve on your own.

Problem 1: (Easy)

*Given 5 coins out of which one coin is **lighter**. In the worst case, how many minimum number of weighing are required to figure out the odd coin?*

Name the coins as 1, 2, 3, 4 and 5. We know that one coin is lighter. Considering best out come of balance, we can group the coins in two different ways, [(1, 2), (3, 4) and (5)], or [(12), (34) and (5)]. We can easily rule out groups like [(123) and (45)], as we will get obvious answer. Any other combination will fall into one of these two groups, like [(2)(45) and (13)], etc.

Consider the first group, pairs (1, 2) and (3, 4). We can check (1, 2), if they are equal we go ahead with (3, 4). We need two weighing in worst case. The same analogy can be applied when the coin in heavier.

With the second group, weigh (12) and (34). If they balance (5) is defective one, otherwise pick the lighter pair, and we need one more weighing to find odd one.

Both the combinations need two weighing in case of 5 coins with prior information of one coin is lighter.

Analysis: In general, if we know that the coin is heavy or light, we can trace the coin in $\log_3(N)$ trials (rounded to next integer). If we represent the outcome of balance as ternary tree, every leaf represent an outcome. Since any coin among N coins can be defective, we need to get a 3-ary tree having minimum of N leaves. A 3-ary tree at k-th level will have 3^k leaves and hence we need $3^k >= N$.

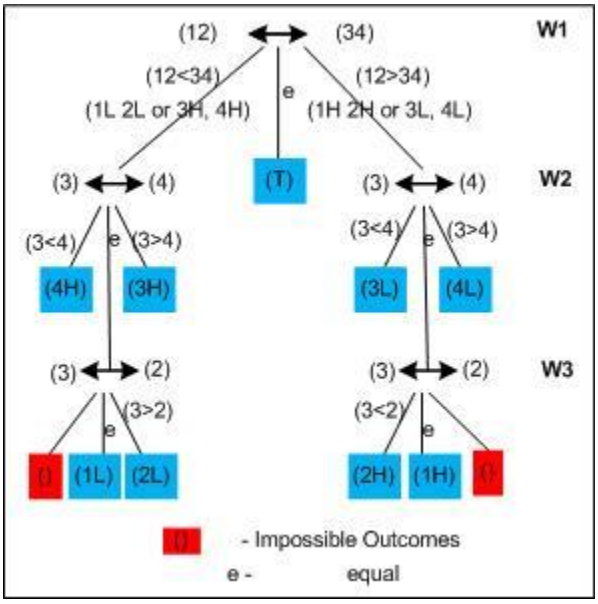
In other-words, in ***k*** trials we can examine upto 3^k coins, if we know whether the defective coin is heavier or lighter. Given that a coin is heavier, verify that 3 trials are sufficient to find the odd coin among 12 coins, because $3^2 < 12 < 3^3$.

Problem 2: (Difficult)

*We are given 4 coins, out of which only one coin **may be** defective. We don't know, whether all coins are genuine or any defective one is present. How many number of weighing are required in worst case to figure out the odd coin, if present? We also need to tell whether it is heavier or lighter.*

From the above analysis we may think that k = 2 trials are sufficient, since a two level 3-ary tree yields 9 leaves which is greater than N = 4 (read the problem once again). Note that it is impossible to solve above 4 coins problem in two weighing. The decision tree confirms the fact (try to draw).

We can group the coins in two different ways, [(12, 34)] or [(1, 2) and (3, 4)]. Let us consider the combination (12, 34), the corresponding decision tree is given below. Blue leaves are valid outcomes, and red leaves are impossible cases. We arrived at impossible cases due to the assumptions made earlier on the path.



The outcome can be (12) < (34) i.e. we go on to left subtree or (12) > (34) i.e. we go on to right subtree.

The left subtree is possible in two ways,

- A) Either 1 or 2 can be lighter OR
- B) Either 3 or 4 can be heavier.

Further on the left subtree, as second trial, we weigh (1, 2) or (3, 4). Let us consider (3, 4) as the analogy for (1, 2) is similar. The outcome of second trail can be three ways

- A) (3) < (4) yielding 4 as defective heavier coin, OR
- B) (3) > (4) yielding 3 as defective heavier coin OR
- C) (3) = (4), yielding ambiguity. Here we need one more weighing to check a genuine coin against 1 or 2. In the figure I took (3, 2) where 3 is confirmed as genuine. We can get (3) > (2) in which 2 is lighter, or (3) = (2) in which 1 is lighter. Note that it impossible to get (3) < (2), it contradicts our assumption leaned to left side.

Similarly we can analyze the right subtree. We need two more weighings on right subtree as well.

Overall we need 3 weighings to trace the odd coin. Note that we are unable to utilize two outcomes of 3-ary trees. Also, the tree is not full tree, middle branch terminated after first weighing. Infact, we can get 27 leaves of 3 level full 3-ary tree, but only we got 11 leaves including impossible cases.

Analysis: Given N coins, all may be genuine or only one coin is defective. We need a decision tree with atleast (2N + 1) leaves correspond to the outputs. Because there can be N leaves to be lighter, or N leaves to be heavier or one genuine case, on total (2N + 1) leaves.

As explained earlier ternary tree at level k, can have utmost 3^k leaves and we need a tree with leaves of $3^k > (2N + 1)$.

In other words, we need atleast $k > log_3(2N + 1)$ weighing to find the defective one.

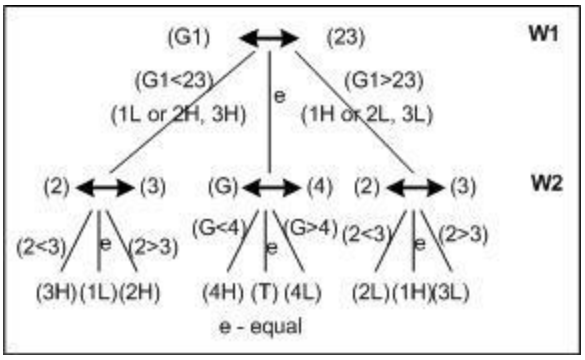
Observe the above figure that not all the branches are generating leaves, i.e. we are missing valid outputs under some branches that leading to more number of trials. When possible, we should group the coins in such a way that every branch is yielding valid output (in simple terms generate full 3-ary tree). Problem 4 describes this approach of 12 coins.

Problem 3: (Special case of two pan balance)

*We are given 5 coins, a group of 4 coins out of which one coin is defective (we **don't know** whether it is heavier or lighter), and one coin is genuine. How many weighing are required in worst case to figure out the odd coin whether it is heavier or lighter?*

Label the coins as 1, 2, 3, 4 and G (genuine). We now have some information on coin purity. We need to make use that in the groupings.

We can best group them as [(G1, 23) and (4)]. Any other group can't generate full 3-ary tree, try yourself. The following diagram explains the procedure.



The middle case (G1) = (23) is self explanatory, i.e. 1, 2, 3 are genuine and 4th coin can be figured out lighter or heavier in one more trial.

The left side of tree corresponds to the case (G1) < (23). This is possible in two ways, either 1 should be lighter or either of (2, 3) should be heavier. The former instance is obvious when next weighing (2, 3) is balanced, yielding 1 as lighter. The later instance can be (2) < (3) yielding 3 as heavier or (2) > (3) yielding 2 as heavier. The leaf nodes on left branch are named to reflect these outcomes.

The right side of tree corresponds to the case (G1) > (23). This is possible in two ways, either 1 is heavier or either of (2, 3) should be lighter. The former instance is obvious when the next weighing (2, 3) is balanced, yielding 1 as heavier. The later case can be (2) < (3) yielding 2 as lighter coin, or (2) > (3) yielding 3 as lighter.

In the above problem, under any possibility we need only two weighing. We are able to use all outcomes of two level full 3-ary tree. We started with (N + 1) = 5 coins where N = 4, we end up with (2N + 1) = 9 leaves. *Infact we should have 11 outcomes since we stared with 5 coins, where are other 2 outcomes? These two outcomes can be declared at the root of tree itself (prior to first weighing), can you figure out?*

If we observe the figure, after the first weighing the problem reduced to "we know three coins, either one can be lighter (heavier) or one among other two can be heavier (lighter)". This can be solved in one weighing (read Problem 1).

Analysis: Given (N + 1) coins, one is genuine and the rest N can be genuine or only one coin is defective. The required decision tree should result in minimum of (2N + 1) leaves. Since the total possible outcomes are (2(N + 1) + 1), number of weighing (trials) are given by the height of ternary tree, $k >= \log_3[2(N + 1) + 1]$. *Note the equality sign.*

Rearranging k and N, *we can weigh maximum of* $N <= (3^k - 3)/2$ *coins in k trials.*

Problem 4: (The classic 12 coin puzzle)

You are given two pan fair balance. You have 12 identically looking coins out of which one coin may be lighter or heavier. How can you find odd coin, if any, in minimum trials, also determine whether defective coin is lighter or heavier, in the worst case?

How do you want to group them? Bi-set or tri-set? Clearly we can discard the option of dividing into two equal groups. It can't lead to best tree. Remember to group coins such that the first weighing reveals atleast one genuine coin.

Let us name the coins as 1, 2, … 8, A, B, C and D. We can combine the coins into 3 groups, namely (1234), (5678) and (ABCD). Weigh (1234) and (5678). You are encouraged to draw decision tree while reading the procedure. The outcome can be three ways,

1. (1234) = (5678), both groups are equal. Defective coin may be in (ABCD) group.
2. (1234) < (5678), i.e. first group is less in weight than second group.
3. (1234) > (5678), i.e. first group is more in weight than second group.

The output (1) can be solved in two more weighing as special case of two pan balance given in Problem 3. We know that groups (1234) and (5678) are genuine and defective coin may be in (ABCD). Pick one genuine coin from any of weighed groups, and proceed with (ABCD) as explained in Problem 3.

Outcomes (2) and (3) are special. In both the cases, we know that (ABCD) is genuine. And also, we know a set of coins being lighter and a set of coins being heavier. We need to shuffle the weighed two groups in such a way that we end up with smaller height decision tree.

Consider the second outcome where (1234) < (5678). It is possible when any coin among (1, 2, 3, 4) is lighter or any coin among (5, 6, 7, 8) is heavier. We revealed lighter or heavier possibility after first weighing. If we proceed as in Problem 1, we will not generate best decision tree. Let us shuffle coins as (1235) and (4BCD) as new groups (there are different shuffles possible, they also lead to minimum weighing, can you try?). If we weigh these two groups again the outcome can be three ways, i) (1235) < (4BCD) yielding one among 1, 2, 3 is lighter which is similar to Problem 1 explained above, we need one more weighing, ii) (1235) = (4BCD) yielding one among 6, 7, 8 is heavier which is similar to Problem 1 explained above, we need one more weighing iii) (1235) > (4BCD) yielding either 5 as heavier coin or 4 as lighter coin, at the expense of one more weighing.

Similar way we can also solve the right subtree (third outcome where (1234) > (5678)) in two more weighing.

We are able to solve the 12 coin puzzle in 3 weighing worst case.

Few Interesting Puzzles:

1. Solve Problem 4 with N = 8 and N = 13, How many minimum trials are required in each case?
2. Given a function *int weigh(A[], B[])* where A and B are arrays (need not be equal size). The function returns -1, 0 or 1. It returns 0 if sum of all elements in A and B are equal, -1 if A < B and 1 if A > B. Given an array of 12 elements, all elements are equal except one. The odd element can be as that of others, smaller or greater than other. Write a program to find the odd element (if any) using *weigh()* minimum times.

3. You might have seen 3-pan balance in science labs during school days. Given a 3-pan balance (4 outcomes) and N coins, how many minimum trials are need to figure out odd coin?

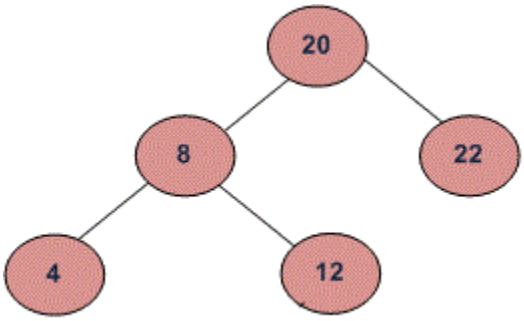
References:

Similar problem was provided in one of the exercises of the book ”Introduction to Algorithms by Levitin”. Specifically read section 5.5 and section 11.2 including exercises

PRINT BST KEYS IN THE GIVEN RANGE

Given two values k1 and k2 (where k1 < k2) and a root pointer to a Binary Search Tree. Print all the keys of tree in range k1 to k2. i.e. print all x such that k1<=x<=k2 and x is a key of given BST. Print all the keys in increasing order.

For example, if k1 = 10 and k2 = 22, then your function should print 12, 20 and 22.



Thanks to bhasker for suggesting the following solution.

Algorithm:
1) If value of root’s key is greater than k1, then recursively call in left subtree.
2) If value of root’s key is in range, then print the root’s key.
3) If value of root’s key is smaller than k2, then recursively call in right subtree.

Implementation:

?

```
#include<stdio.h>

/* A tree node structure */
struct node
{
  int data;
  struct node *left;
  struct node *right;
};
```

```
/* The functions prints all the keys which in the given range [k1..k2].
    The function assumes than k1 < k2 */
void Print(struct node *root, int k1, int k2)
{
   /* base case */
   if ( NULL == root )
      return;

   /* Since the desired o/p is sorted, recurse for left subtree first
      If root->data is greater than k1, then only we can get o/p keys
      in left subtree */
   if ( k1 < root->data )
     Print(root->left, k1, k2);

   /* if root's data lies in range, then prints root's data */
   if ( k1 <= root->data && k2 >= root->data )
     printf("%d ", root->data );

  /* If root->data is smaller than k2, then only we can get o/p keys
      in right subtree */
   if ( k2 > root->data )
     Print(root->right, k1, k2);
}

/* Utility function to create a new Binary Tree node */
struct node* newNode(int data)
{
  struct node *temp = new struct node;
  temp->data = data;
  temp->left = NULL;
  temp->right = NULL;

  return temp;
}

/* Driver function to test above functions */
int main()
{
  struct node *root = new struct node;
  int k1 = 10, k2 = 25;

  /* Constructing tree given in the above figure */
  root = newNode(20);
  root->left = newNode(8);
  root->right = newNode(22);
  root->left->left = newNode(4);
  root->left->right = newNode(12);

  Print(root, k1, k2);

  getchar();
  return 0;
}
```

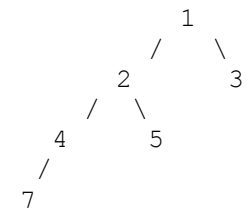
Output:
12 20 22

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

PRINT ANCESTORS OF A GIVEN NODE IN BINARY TREE

Given a Binary Tree and a key, write a function that prints all the ancestors of the key in the given binary tree.

For example, if the given tree is following Binary Tree and key is 7, then your function should print 4, 2 and 1.



Thanks to Mike , Sambasiva and wgpshashank for their contribution.

?

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

using namespace std;

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
   int data;
   struct node* left;
   struct node* right;
};

/* If target is present in tree, then prints the ancestors
   and returns true, otherwise returns false. */
bool printAncestors(struct node *root, int target)
{
  /* base cases */
  if ( root == NULL )
     return false;

  if ( root->data == target )
     return true;
```

```
  /* If target is present in either left or right subtree of this node,
     then print this node */
  if ( printAncestors(root->left, target) ||
       printAncestors(root->right, target) )
  {
    cout<<root->data<<" ";
    return true;
  }

  /* Else return false */
  return false;
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newnode(int data)
{
  struct node* node = (struct node*)
                      malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;

  return(node);
}

/* Driver program to test above functions*/
int main()
{

  /* Construct the following binary tree
              1
            /   \
          2      3
        /  \
      4     5
     /
    7
  */
  struct node *root = newnode(1);
  root->left        = newnode(2);
  root->right       = newnode(3);
  root->left->left  = newnode(4);
  root->left->right = newnode(5);
  root->left->left->left  = newnode(7);

  printAncestors(root, 7);

  getchar();
  return 0;
}
```

Output:

4 2 1 Time Complexity: O(n) where n is the number of nodes in the given Binary Tree.