Print all possible combinations of r elements in a given array of size n

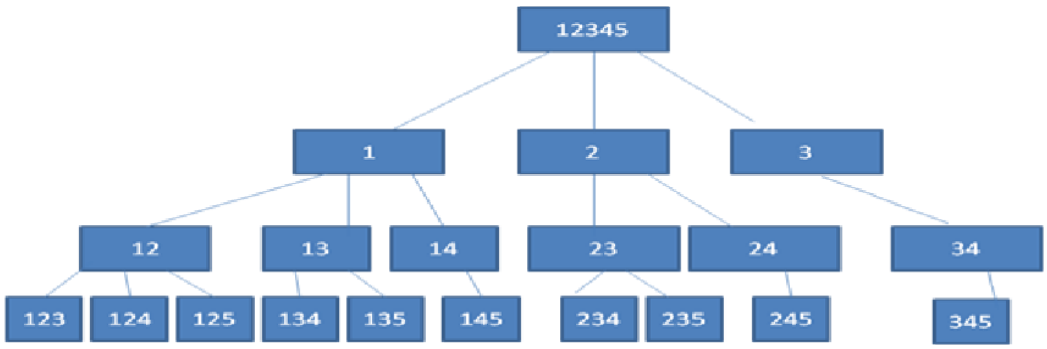
May 30, 2013

Given an array of size n, generate and print all possible combinations of r elements in array. For example, if input array is {1, 2, 3, 4} and r is 2, then output should be {1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4} and {3, 4}.

Following are two methods to do this.

Method 1 (Fix Elements and Recur)
We create a temporary array 'data[]' which stores all outputs one by one. The idea is to start from first index (index = 0) in data[], one by one fix elements at this index and recur for remaining indexes. Let the input array be {1, 2, 3, 4, 5} and r be 3. We first fix 1 at index 0 in data[], then recur for remaining indexes, then we fix 2 at index 0 and recur. Finally, we fix 3 and recur for remaining indexes. When number of elements in data[] becomes equal to r (size of a combination), we print data[].

Following diagram shows recursion tree for same input.



Following is C++ implementation of above approach.

```
// Program to print all combination of size r in an array of size n
#include <stdio.h>
void combinationUtil(int arr[], int data[], int start, int end, int index, int
r);

// The main function that prints all combinations of size r
// in arr[] of size n. This function mainly uses combinationUtil()
void printCombination(int arr[], int n, int r)
{
    // A temporary array to store all combination one by one
    int data[r];

    // Print all combination using temprary array 'data[]'
    combinationUtil(arr, data, 0, n-1, 0, r);
}

/* arr[]  ---> Input Array
   data[] ---> Temporary array to store current combination
```

```
   start & end ---> Staring and Ending indexes in arr[]
   index  ---> Current index in data[]
   r ---> Size of a combination to be printed */
void combinationUtil(int arr[], int data[], int start, int end, int index, int
r)
{
    // Current combination is ready to be printed, print it
    if (index == r)
    {
        for (int j=0; j<r; j++)
            printf("%d ", data[j]);
        printf("\n");
        return;
    }

    // replace index with all possible elements. The condition
    // "end-i+1 >= r-index" makes sure that including one element
    // at index will make a combination with remaining elements
    // at remaining positions
    for (int i=start; i<=end && end-i+1 >= r-index; i++)
    {
        data[index] = arr[i];
        combinationUtil(arr, data, i+1, end, index+1, r);
    }
}

// Driver program to test above functions
int main()
{
    int arr[] = {1, 2, 3, 4, 5};
    int r = 3;
    int n = sizeof(arr)/sizeof(arr[0]);
    printCombination(arr, n, r);
}
```

Output:

```
1 2 3
1 2 4
1 2 5
1 3 4
1 3 5
1 4 5
2 3 4
2 3 5
2 4 5
3 4 5
```

How to handle duplicates?

Note that the above method doesn't handle duplicates. For example, if input array is {1, 2, 1} and r is 2, then the program prints {1, 2} and {2, 1} as two different combinations. We can avoid duplicates by adding following two additional things to above code.

1) Add code to sort the array before calling combinationUtil() in printCombination()

2) Add following lines at the end of for loop in combinationUtil()

```
        // Since the elements are sorted, all occurrences of an element
        // must be together
        while (arr[i] == arr[i+1])
```

```
            i++;
```

See **this** for an implementation that handles duplicates.

Method 2 (Include and Exclude every element)
Like the above method, We create a temporary array data[]. The idea here is similar to Subset Sum Problem. We one by one consider every element of input array, and recur for two cases:

1) The element is included in current combination (We put the element in data[] and increment next available index in data[])
2) The element is excluded in current combination (We do not put the element and do not change index)

When number of elements in data[] become equal to r (size of a combination), we print it.

This method is mainly based on Pascal's Identity, i.e. $\mathbf{n_{cr} = n\text{-}1_{cr} + n\text{-}1_{cr\text{-}1}}$

Following is C++ implementation of method 2.

```
// Program to print all combination of size r in an array of size n
#include<stdio.h>
void combinationUtil(int arr[],int n,int r,int index,int data[],int i);

// The main function that prints all combinations of size r
// in arr[] of size n. This function mainly uses combinationUtil()
void printCombination(int arr[], int n, int r)
{
    // A temporary array to store all combination one by one
    int data[r];

    // Print all combination using temprary array 'data[]'
    combinationUtil(arr, n, r, 0, data, 0);
}

/* arr[]  ---> Input Array
   n      ---> Size of input array
   r      ---> Size of a combination to be printed
   index  ---> Current index in data[]
   data[] ---> Temporary array to store current combination
   i      ---> index of current element in arr[]     */
void combinationUtil(int arr[], int n, int r, int index, int data[], int i)
{
    // Current cobination is ready, print it
    if (index == r)
    {
        for (int j=0; j<r; j++)
            printf("%d ",data[j]);
        printf("\n");
        return;
    }

    // When no more elements are there to put in data[]
    if (i >= n)
        return;
```

```
    // current is included, put next at next location
    data[index] = arr[i];
    combinationUtil(arr, n, r, index+1, data, i+1);

    // current is excluded, replace it with next (Note that
    // i+1 is passed, but index is not changed)
    combinationUtil(arr, n, r, index, data, i+1);
}

// Driver program to test above functions
int main()
{
    int arr[] = {1, 2, 3, 4, 5};
    int r = 3;
    int n = sizeof(arr)/sizeof(arr[0]);
    printCombination(arr, n, r);
    return 0;
}
```

Output:

```
1 2 3
1 2 4
1 2 5
1 3 4
1 3 5
1 4 5
2 3 4
2 3 5
2 4 5
3 4 5
```

How to handle duplicates in method 2?
Like method 1, we can following two things to handle duplicates.
1) Add code to sort the array before calling combinationUtil() in printCombination()
2) Add following lines between two recursive calls of combinationUtil() in combinationUtil()

```
        // Since the elements are sorted, all occurrences of an element
        // must be together
        while (arr[i] == arr[i+1])
             i++;
```

See **this** for an implementation that handles duplicates.

Stock Buy Sell to Maximize Profit

March 31, 2013

The cost of a stock on each day is given in an array, find the max profit that you can make by buying and selling in those days. For example, if the given array is {100, 180, 260, 310, 40, 535, 695}, the maximum profit can earned by buying on day 0, selling on day 3. Again buy on day 4 and sell on day 6. If the given array of prices is sorted in decreasing order, then profit cannot be earned at all.

If we are allowed to buy and sell only once, then we can use following algorithm. Maximum difference between two elements. Here we are allowed to buy and sell multiple times. Following is algorithm for this problem.
1. Find the local minima and store it as starting index. If not exists, return.
2. Find the local maxima. and store it as ending index. If we reach the end, set the end as ending index.
3. Update the solution (Increment count of buy sell pairs)
4. Repeat the above steps if end is not reached.

```
// Program to find best buying and selling days
#include <stdio.h>

// solution structure
struct Interval
{
    int buy;
    int sell;
};

// This function finds the buy sell schedule for maximum profit
void stockBuySell(int price[], int n)
{
    // Prices must be given for at least two days
    if (n == 1)
        return;

    int count = 0; // count of solution pairs

    // solution vector
    Interval sol[n/2 + 1];

    // Traverse through given price array
    int i = 0;
    while (i < n-1)
    {
        // Find Local Minima. Note that the limit is (n-2) as we are
        // comparing present element to the next element.
        while ((i < n-1) && (price[i+1] <= price[i]))
            i++;

        // If we reached the end, break as no further solution possible
        if (i == n-1)
            break;

        // Store the index of minima
        sol[count].buy = i++;

        // Find Local Maxima.  Note that the limit is (n-1) as we are
        // comparing to previous element
        while ((i < n) && (price[i] >= price[i-1]))
```

```
            i++;

        // Store the index of maxima
        sol[count].sell = i-1;

        // Increment count of buy/sell pairs
        count++;
    }

    // print solution
    if (count == 0)
        printf("There is no day when buying the stock will make profit\n");
    else
    {
       for (int i = 0; i < count; i++)
          printf("Buy on day: %d\t Sell on day: %d\n", sol[i].buy,
sol[i].sell);
    }

    return;
}

// Driver program to test above functions
int main()
{
    // stock prices on consecutive days
    int price[] = {100, 180, 260, 310, 40, 535, 695};
    int n = sizeof(price)/sizeof(price[0]);

    // fucntion call
    stockBuySell(price, n);

    return 0;
}
```

Output:

```
Buy on day : 0   Sell on day: 3
Buy on day : 4   Sell on day: 6
```

Time Complexity: The outer loop runs till i becomes n-1. The inner two loops increment value of i in every iteration. So overall time complexity is O(n)

Find the maximum repeating number in O(n) time and O(1) extra space

Given an array of size *n*, the array contains numbers in range from 0 to *k-1* where *k* is a positive integer and *k* <= *n*. Find the maximum repeating number in this array. For example, let *k* be 10 the given array be *arr[]* = {1, 2, 2, 2, 0, 2, 0, 2, 3, 8, 0, 9, 2, 3}, the maximum repeating number would be 2. Expected time complexity is *O(n)* and extra space allowed is *O(1)*. Modifications to array are allowed.

The **naive approach** is to run two loops, the outer loop picks an element one by one, the inner loop counts number of occurrences of the picked element. Finally return the element with maximum count. Time complexity of this approach is *O(n^2)*.

A **better approach** is to create a count array of size k and initialize all elements of *count[]* as 0. Iterate through all elements of input array, and for every element *arr[i]*, increment *count[arr[i]]*. Finally, iterate through *count[]* and return the index with maximum value. This approach takes O(n) time, but requires O(k) space.

Following is the ***O(n)* time and *O(1)* extra space** approach. Let us understand the approach with a simple example where *arr[]* = {2, 3, 3, 5, 3, 4, 1, 7}, *k* = 8, *n* = 8 (number of elements in arr[]).

1) Iterate though input array *arr[]*, for every element *arr[i]*, increment *arr[arr[i]%k]* by *k* (*arr[]* becomes {2, 11, 11, 29, 11, 12, 1, 15 })

2) Find the maximum value in the modified array (maximum value is 29). Index of the maximum value is the maximum repeating element (index of 29 is 3).

3) If we want to get the original array back, we can iterate through the array one more time and do *arr[i] = arr[i] % k* where *i* varies from 0 to *n-1*.

How does the above algorithm work? Since we use *arr[i]%k* as index and add value *k* at the index *arr[i]%k*, the index which is equal to maximum repeating element will have the maximum value in the end. Note that *k* is added maximum number of times at the index equal to maximum repeating element and all array elements are smaller than *k*.

Following is C++ implementation of the above algorithm.

```
#include<iostream>
using namespace std;

// Returns maximum repeating element in arr[0..n-1].
// The array elements are in range from 0 to k-1
int maxRepeating(int* arr, int n, int k)
{
    // Iterate though input array, for every element
    // arr[i], increment arr[arr[i]%k] by k
    for (int i = 0; i< n; i++)
        arr[arr[i]%k] += k;

    // Find index of the maximum repeating element
    int max = arr[0], result = 0;
    for (int i = 1; i < n; i++)
    {
        if (arr[i] > max)
        {
            max = arr[i];
            result = i;
        }
    }

    /* Uncomment this code to get the original array back
       for (int i = 0; i< n; i++)
          arr[i] = arr[i]%k; */
```

```
    // Return index of the maximum element
    return result;
}

// Driver program to test above function
int main()
{
    int arr[] = {2, 3, 3, 5, 3, 4, 1, 7};
    int n = sizeof(arr)/sizeof(arr[0]);
    int k = 8;

    cout << "The maximum repeating number is " <<
         maxRepeating(arr, n, k) << endl;

    return 0;
}
```

Output:

```
The maximum repeating number is 3
```

Exercise:
The above solution prints only one repeating element and doesn't work if we want to print all maximum repeating elements. For example, if the input array is {2, 3, 2, 3}, the above solution will print only 3. What if we need to print both of 2 and 3 as both of them occur maximum number of times. Write a O(n) time and O(1) extra space function that prints all maximum repeating elements. (Hint: We can use maximum quotient arr[i]/n instead of maximum value in step 2).

Note that the above solutions may cause overflow if adding k repeatedly makes the value more than INT_MAX.

Tug of War

Given a set of n integers, divide the set in two subsets of n/2 sizes each such that the difference of the sum of two subsets is as minimum as possible. If n is even, then sizes of two subsets must be strictly n/2 and if n is odd, then size of one subset must be (n-1)/2 and size of other subset must be (n+1)/2.

For example, let given set be {3, 4, 5, -3, 100, 1, 89, 54, 23, 20}, the size of set is 10. Output for this set should be {4, 100, 1, 23, 20} and {3, 5, -3, 89, 54}. Both output subsets are of size 5 and sum of elements in both subsets is same (148 and 148).
Let us consider another example where n is odd. Let given set be {23, 45, -34, 12, 0, 98, -99, 4, 189, -1, 4}. The output subsets should be {45, -34, 12, 98, -1} and {23, 0, -99, 4, 189, 4}. The sums of elements in two subsets are 120 and 121 respectively.

The following solution tries every possible subset of half size. If one subset of half size is formed, the remaining elements form the other subset. We initialize current set as empty and one by one build it. There are two possibilities for every element, either it is part of current set, or it is part of the remaining elements (other subset). We consider both possibilities for

every element. When the size of current set becomes n/2, we check whether this solutions is better than the best solution available so far. If it is, then we update the best solution.

Following is C++ implementation for Tug of War problem. It prints the required arrays.

```
#include <iostream>
#include <stdlib.h>
#include <limits.h>
using namespace std;

// function that tries every possible solution by calling itself
recursively
void TOWUtil(int* arr, int n, bool* curr_elements, int
no_of_selected_elements,
             bool* soln, int* min_diff, int sum, int curr_sum, int
curr_position)
{
    // checks whether the it is going out of bound
    if (curr_position == n)
        return;

    // checks that the numbers of elements left are not less than the
    // number of elements required to form the solution
    if ((n/2 - no_of_selected_elements) > (n - curr_position))
        return;

    // consider the cases when current element is not included in the
solution
    TOWUtil(arr, n, curr_elements, no_of_selected_elements,
              soln, min_diff, sum, curr_sum, curr_position+1);

    // add the current element to the solution
    no_of_selected_elements++;
    curr_sum = curr_sum + arr[curr_position];
    curr_elements[curr_position] = true;

    // checks if a solution is formed
    if (no_of_selected_elements == n/2)
    {
        // checks if the solution formed is better than the best solution
so far
        if (abs(sum/2 - curr_sum) < *min_diff)
        {
            *min_diff = abs(sum/2 - curr_sum);
            for (int i = 0; i<n; i++)
                soln[i] = curr_elements[i];
        }
    }
    else
    {
        // consider the cases where current element is included in the
solution
        TOWUtil(arr, n, curr_elements, no_of_selected_elements, soln,
                  min_diff, sum, curr_sum, curr_position+1);
    }

    // removes current element before returning to the caller of this
function
    curr_elements[curr_position] = false;
```

```
}

// main function that generate an arr
void tugOfWar(int *arr, int n)
{
    // the boolen array that contains the inclusion and exclusion of an
element
    // in current set. The number excluded automatically form the other set
    bool* curr_elements = new bool[n];

    // The inclusion/exclusion array for final solution
    bool* soln = new bool[n];

    int min_diff = INT_MAX;

    int sum = 0;
    for (int i=0; i<n; i++)
    {
        sum += arr[i];
        curr_elements[i] =  soln[i] = false;
    }

    // Find the solution using recursive function TOWUtil()
    TOWUtil(arr, n, curr_elements, 0, soln, &min_diff, sum, 0, 0);

    // Print the solution
    cout << "The first subset is: ";
    for (int i=0; i<n; i++)
    {
        if (soln[i] == true)
            cout << arr[i] << " ";
    }
    cout << "\nThe second subset is: ";
    for (int i=0; i<n; i++)
    {
        if (soln[i] == false)
            cout << arr[i] << " ";
    }
}

// Driver program to test above functions
int main()
{
    int arr[] = {23, 45, -34, 12, 0, 98, -99, 4, 189, -1, 4};
    int n = sizeof(arr)/sizeof(arr[0]);
    tugOfWar(arr, n);
    return 0;
}
```

Output:

```
The first subset is: 45 -34 12 98 -1
The second subset is: 23 0 -99 4 189 4
```

Arrange given numbers to form the biggest number

February 19, 2013

Given an array of numbers, arrange them in a way that yields the largest value. For example, if the given numbers are {54, 546, 548, 60}, the arrangement 6054854654 gives the largest value. And if the given numbers are {1, 34, 3, 98, 9, 76, 45, 4}, then the arrangement 998764543431 gives the largest value.

A simple solution that comes to our mind is to sort all numbers in descending order, but simply sorting doesn't work. For example, 548 is greater than 60, but in output 60 comes before 548. As a second example, 98 is greater than 9, but 9 comes before 98 in output.

So how do we go about it? The idea is to use any comparison based sorting algorithm. In the used sorting algorithm, instead of using the default comparison, write a comparison function myCompare() and use it to sort numbers. Given two numbers X and Y, how should myCompare() decide which number to put first – we compare two numbers XY (Y appended at the end of X) and YX (X appended at the end of Y). If XY is larger, then X should come before Y in output, else Y should come before. For example, let X and Y be 542 and 60. To compare X and Y, we compare 54260 and 60542. Since 60542 is greater than 54260, we put Y first.

Following is C++ implementation of the above approach. To keep the code simple, numbers are considered as strings, and vector is used instead of normal array.

```
// Given an array of numbers, program to arrange the numbers to form the
// largest number
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

// A comparison function which is used by sort() in printLargest()
int myCompare(string X, string Y)
{
    // first append Y at the end of X
    string XY = X.append(Y);

    // then append X at the end of Y
    string YX = Y.append(X);

    // Now see which of the two formed numbers is greater
    return XY.compare(YX) > 0 ? 1: 0;
}

// The main function that prints the arrangement with the largest value.
// The function accepts a vector of strings
void printLargest(vector<string> arr)
{
    // Sort the numbers using library sort funtion. The function uses
    // our comparison function myCompare() to compare two strings.
    // See http://www.cplusplus.com/reference/algorithm/sort/ for details
    sort(arr.begin(), arr.end(), myCompare);

    for (int i=0; i < arr.size() ; i++ )
        cout << arr[i];
```

```
}

// driverr program to test above functions
int main()
{
    vector<string> arr;

    //output should be 6054854654
    arr.push_back("54");
    arr.push_back("546");
    arr.push_back("548");
    arr.push_back("60");
    printLargest(arr);

    // output should be 777776
    /*arr.push_back("7");
    arr.push_back("776");
    arr.push_back("7");
    arr.push_back("7");*/

    //output should be 998764543431
    /*arr.push_back("1");
    arr.push_back("34");
    arr.push_back("3");
    arr.push_back("98");
    arr.push_back("9");
    arr.push_back("76");
    arr.push_back("45");
    arr.push_back("4");
    */

   return 0;
}
```

Output:

```
6054854654
```

Find the first circular tour that visits all petrol pumps

February 11, 2013

Suppose there is a circle. There are n petrol pumps on that circle. You are given two sets of data.

1. The amount of petrol that petrol pump will give.
2. Distance from that petrol pump to the next petrol pump.

Calculate the first point from where a truck will be able to complete the circle (The truck will stop at each petrol pump and it has infinite capacity). Expected time complexity is O(n). Assume for 1 litre petrol, the truck can go 1 unit of distance.

For example, let there be 4 petrol pumps with amount of petrol and distance to next petrol pump value pairs as {4, 6}, {6, 5}, {7, 3} and {4, 5}. The first point from where truck can make a circular tour is 2nd petrol pump. Output should be “start = 1″ (index of 2nd petrol pump).

A **Simple Solution** is to consider every petrol pumps as starting point and see if there is a possible tour. If we find a starting point with feasible solution, we return that starting point. The worst case time complexity of this solution is O(n^2).

We can **use a Queue** to store the current tour. We first enqueue first petrol pump to the queue, we keep enqueueing petrol pumps till we either complete the tour, or current amount of petrol becomes negative. If the amount becomes negative, then we keep dequeueing petrol pumps till the current amount becomes positive or queue becomes empty.

Instead of creating a separate queue, we use the given array itself as queue. We maintain two index variables start and end that represent rear and front of queue.

```
// C program to find circular tour for a truck
#include <stdio.h>

// A petrol pump has petrol and distance to next petrol pump
struct petrolPump
{
  int petrol;
  int distance;
};

// The function returns starting point if there is a possible solution,
// otherwise returns -1
int printTour(struct petrolPump arr[], int n)
{
    // Consider first petrol pump as a starting point
    int start = 0;
    int end =  1;

    int curr_petrol = arr[start].petrol - arr[start].distance;

    /* Run a loop while all petrol pumps are not visited.
      And we have reached first petrol pump again with 0 or more petrol */
    while (end != start || curr_petrol < 0)
    {
        // If curremt amount of petrol in truck becomes less than 0, then
        // remove the starting petrol pump from tour
        while (curr_petrol < 0 && start != end)
        {
            // Remove starting petrol pump. Change start
            curr_petrol -= arr[start].petrol - arr[start].distance;
            start = (start + 1)%n;

            // If 0 is being considered as start again, then there is no
            // possible solution
            if (start == 0)
               return -1;
        }

        // Add a petrol pump to current tour
```

```
        curr_petrol += arr[end].petrol - arr[end].distance;

        end = (end + 1)%n;
    }

    // Return starting point
    return start;
}

// Driver program to test above functions
int main()
{
    struct petrolPump arr[] = {{6, 4}, {3, 6}, {7, 3}};

    int n = sizeof(arr)/sizeof(arr[0]);
    int start = printTour(arr, n);

    (start == -1)? printf("No solution"): printf("Start = %d", start);

    return 0;
}
```

Output:

```
start = 2
```

Time Complexity: Seems to be more than linear at first look. If we consider the items between start and end as part of a circular queue, we can observe that every item is enqueued at most two times to the queue. The total number of operations is proportional to total number of enqueue operations. Therefore the time complexity is O(n).

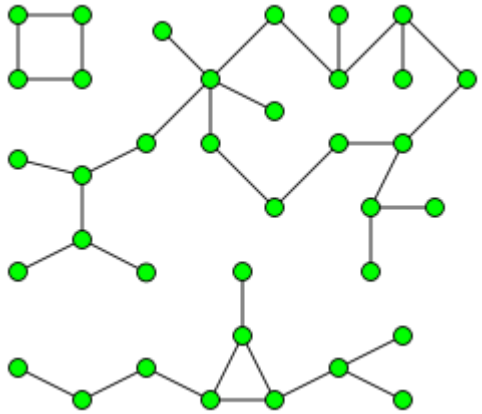
Find the number of islands

October 17, 2012

Given a boolean 2D matrix, find the number of islands.

This is an variation of the standard problem: “Counting number of connected components in a undirected graph”.

Before we go to the problem, let us understand what is a connected component. A connected component of an undirected graph is a subgraph in which every two vertices are connected to each other by a path(s), and which is connected to no other vertices outside the subgraph. For example, the graph shown below has three connected components.



A graph where all vertices are connected with each other, has exactly one connected component, consisting of the whole graph. Such graph with only one connected component is called as Strongly Connected Graph.

The problem can be easily solved by applying DFS() on each component. In each DFS() call, a component or a sub-graph is visited. We will call DFS on the next un-visited component. The number of calls to DFS() gives the number of connected components. BFS can also be used.

What is an island?
A group of connected 1s forms an island. For example, the below matrix contains 5 islands

```
{1, 1, 0, 0, 0},
{0, 1, 0, 0, 1},
{1, 0, 0, 1, 1},
{0, 0, 0, 0, 0},
{1, 0, 1, 0, 1}
```

A cell in 2D matrix can be connected to 8 neighbors. So, unlike standard DFS(), where we recursively call for all adjacent vertices, here we can recursive call for 8 neighbors only. We keep track of the visited 1s so that they are not visited again.

```
// Program to count islands in boolean 2D matrix

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

#define ROW 5
#define COL 5

// A function to check if a given cell (row, col) can be included in DFS
int isSafe(int M[][COL], int row, int col, bool visited[][COL])
{
    return (row >= 0) && (row < ROW) &&     // row number is in range
           (col >= 0) && (col < COL) &&     // column number is in range
           (M[row][col] && !visited[row][col]); // value is 1 and not yet
visited
}

// A utility function to do DFS for a 2D boolean matrix. It only considers
```

```
// the 8 neighbors as adjacent vertices
void DFS(int M[][COL], int row, int col, bool visited[][COL])
{
    // These arrays are used to get row and column numbers of 8 neighbors
    // of a given cell
    static int rowNbr[] = {-1, -1, -1,  0, 0,  1, 1, 1};
    static int colNbr[] = {-1,  0,  1, -1, 1, -1, 0, 1};

    // Mark this cell as visited
    visited[row][col] = true;

    // Recur for all connected neighbours
    for (int k = 0; k < 8; ++k)
        if (isSafe(M, row + rowNbr[k], col + colNbr[k], visited) )
            DFS(M, row + rowNbr[k], col + colNbr[k], visited);
}

// The main function that returns count of islands in a given boolean
// 2D matrix
int countIslands(int M[][COL])
{
    // Make a bool array to mark visited cells.
    // Initially all cells are unvisited
    bool visited[ROW][COL];
    memset(visited, 0, sizeof(visited));

    // Initialize count as 0 and travese through the all cells of
    // given matrix
    int count = 0;
    for (int i = 0; i < ROW; ++i)
        for (int j = 0; j < COL; ++j)
            if (M[i][j] && !visited[i][j]) // If a cell with value 1 is not
            {                              // visited yet, then new island
found
                DFS(M, i, j, visited);     // Visit all cells in this
island.
                ++count;                   // and increment island count
            }

    return count;
}

// Driver program to test above function
int main()
{
    int M[][COL]= {  {1, 1, 0, 0, 0},
        {0, 1, 0, 0, 1},
        {1, 0, 0, 1, 1},
        {0, 0, 0, 0, 0},
        {1, 0, 1, 0, 1}
    };

    printf("Number of islands is: %d\n", countIslands(M));

    return 0;
}
```

Output:

```
Number of islands is: 5
```

Time complexity: O(ROW x COL)

Count the number of possible triangles

October 11, 2012

Given an unsorted array of positive integers. Find the number of triangles that can be formed with three different array elements as three sides of triangles. For a triangle to be possible from 3 values, the sum of any two values (or sides) must be greater than the third value (or third side).
For example, if the input array is {4, 6, 3, 7}, the output should be 3. There are three triangles possible {3, 4, 6}, {4, 6, 7} and {3, 6, 7}. Note that {3, 4, 7} is not a possible triangle.
As another example, consider the array {10, 21, 22, 100, 101, 200, 300}. There can be 6 possible triangles: {10, 21, 22}, {21, 100, 101}, {22, 100, 101}, {10, 100, 101}, {100, 101, 200} and {101, 200, 300}

Method 1 (Brute force)
The brute force method is to run three loops and keep track of the number of triangles possible so far. The three loops select three different values from array, the innermost loop checks for the triangle property (the sum of any two sides must be greater than the value of third side).

Time Complexity: O(N^3) where N is the size of input array.

Method 2 (Tricky and Efficient)
Let a, b and c be three sides. The below condition must hold for a triangle (Sum of two sides is greater than the third side)
i) a + b > c
ii) b + c > a
iii) a + c > b

Following are steps to count triangle.

1. Sort the array in non-decreasing order.

2. Initialize two pointers ‘i’ and ‘j’ to first and second elements respectively, and initialize count of triangles as 0.

3. Fix ‘i’ and ‘j’ and find the rightmost index ‘k’ (or largest ‘arr[k]‘) such that ‘arr[i] + arr[j] > arr[k]‘. The number of triangles that can be formed with ‘arr[i]‘ and ‘arr[j]‘ as two sides is ‘k – j’. Add ‘k – j’ to count of triangles.

Let us consider ‘arr[i]‘ as ‘a’, ‘arr[j]‘ as b and all elements between ‘arr[j+1]‘ and ‘arr[k]‘ as ‘c’. The above mentioned conditions (ii) and (iii) are satisfied because ‘arr[i] < arr[j] < arr[k]'. And we check for condition (i) when we pick 'k'

4. Increment ‘j’ to fix the second element again.

Note that in step 2, we can use the previous value of ‘k’. The reason is simple, if we know that the value of ‘arr[i] + arr[j-1]‘ is greater than ‘arr[k]‘, then we can say ‘arr[i] + arr[j]‘ will also be greater than ‘arr[k]‘, because the array is sorted in increasing order.

5. If ‘j’ has reached end, then increment ‘i’. Initialize ‘j’ as ‘i + 1′, ‘k’ as ‘i+2′ and repeat the steps 3 and 4.

Following is implementation of the above approach.

```
// Program to count number of triangles that can be formed from given array
#include <stdio.h>
#include <stdlib.h>

/* Following function is needed for library function qsort(). Refer
   http://www.cplusplus.com/reference/clibrary/cstdlib/qsort/ */
int comp(const void* a, const void* b)
{  return *(int*)a > *(int*)b ; }

// Function to count all possible triangles with arr[] elements
int findNumberOfTriangles(int arr[], int n)
{
    // Sort the array elements in non-decreasing order
    qsort(arr, n, sizeof( arr[0] ), comp);

    // Initialize count of triangles
    int count = 0;

    // Fix the first element.  We need to run till n-3 as the other two
elements are
    // selected from arr[i+1...n-1]
    for (int i = 0; i < n-2; ++i)
    {
        // Initialize index of the rightmost third element
        int k = i+2;

        // Fix the second element
        for (int j = i+1; j < n; ++j)
        {
            // Find the rightmost element which is smaller than the sum
            // of two fixed elements
            // The important thing to note here is, we use the previous
            // value of k. If value of arr[i] + arr[j-1] was greater than
arr[k],
            // then arr[i] + arr[j] must be greater than k, because the
            // array is sorted.
            while (k < n && arr[i] + arr[j] > arr[k])
               ++k;

            // Total number of possible triangles that can be formed
            // with the two fixed elements is k - j - 1.  The two fixed
```

```
            // elements are arr[i] and arr[j].  All elements between
arr[j+1]
            // to arr[k-1] can form a triangle with arr[i] and arr[j].
            // One is subtracted from k because k is incremented one extra
            // in above while loop.
            // k will always be greater than j. If j becomes equal to k,
then
            // above loop will increment k, because arr[k] + arr[i] is
always
            // greater than arr[k]
            count += k - j - 1;
        }
    }

    return count;
}

// Driver program to test above functionarr[j+1]
int main()
{
    int arr[] =   {10, 21, 22, 100, 101, 200, 300};
    int size = sizeof( arr ) / sizeof( arr[0] );

    printf("Total number of triangles possible is %d ",
           findNumberOfTriangles( arr, size ) );

    return 0;
}
```

Output:

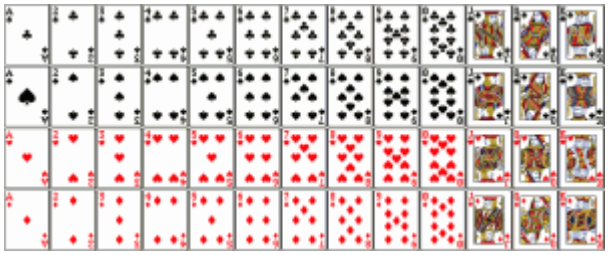
```
Total number of triangles possible is 6
```

Time Complexity: O(n^2). The time complexity looks more because of 3 nested loops. If we take a closer look at the algorithm, we observe that k is initialized only once in the outermost loop. The innermost loop executes at most O(n) time for every iteration of outer most loop, because k starts from i+2 and goes upto n for all values of j. Therefore, the time complexity is O(n^2).

Shuffle a given array

October 10, 2012

Given an array, write a program to generate a random permutation of array elements. This question is also asked as “shuffle a deck of cards” or “randomize a given array”.



Let the given array be *arr[]*. A simple solution is to create an auxiliary array *temp[]* which is initially a copy of *arr[]*. Randomly select an element from *temp[]*, copy the randomly selected element to *arr[0]* and remove the selected element from *temp[]*. Repeat the same process n times and keep copying elements to *arr[1], arr[2], ...* . The time complexity of this solution will be O(n^2).

Fisher–Yates shuffle Algorithm works in O(n) time complexity. The assumption here is, we are given a function rand() that generates random number in O(1) time.
The idea is to start from the last element, swap it with a randomly selected element from the whole array (including last). Now consider the array from 0 to n-2 (size reduced by 1), and repeat the process till we hit the first element.

Following is the detailed algorithm

```
To shuffle an array a of n elements (indices 0..n-1):
  for i from n - 1 downto 1 do
       j = random integer with 0 <= j <= i
       exchange a[j] and a[i]
```

Following is C++ implementation of this algorithm.

```
// C Program to shuffle a given array

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

// A utility function to swap to integers
void swap (int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// A utility function to print an array
void printArray (int arr[], int n)
{
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// A function to generate a random permutation of arr[]
void randomize ( int arr[], int n )
{
    // Use a different seed value so that we don't get same
```

```
    // result each time we run this program
    srand ( time(NULL) );

    // Start from the last element and swap one by one. We don't
    // need to run for the first element that's why i > 0
    for (int i = n-1; i > 0; i--)
    {
        // Pick a random index from 0 to i
        int j = rand() % (i+1);

        // Swap arr[i] with the element at random index
        swap(&arr[i], &arr[j]);
    }
}

// Driver program to test above function.
int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8};
    int n = sizeof(arr)/ sizeof(arr[0]);
    randomize (arr, n);
    printArray(arr, n);

    return 0;
}
```

Output:

```
7 8 4 6 3 1 2 5
```

The above function assumes that rand() generates a random number.

Time Complexity: O(n), assuming that the function rand() takes O(1) time.

How does this work?
The probability that ith element (including the last one) goes to last position is 1/n, because we randomly pick an element in first iteration.

The probability that ith element goes to second last position can be proved to be 1/n by dividing it in two cases.
Case 1: i = n-1 (index of last element):
The probability of last element going to second last position is = (probability that last element doesn't stay at its original position) x (probability that the index picked in previous step is picked again so that the last element is swapped)
So the probability = ((n-1)/n) x (1/(n-1)) = 1/n
Case 2: 0 < i < n-1 (index of non-last):
The probability of ith element going to second position = (probability that ith element is not picked in previous iteration) x (probability that ith element is picked in this iteration)
So the probability = ((n-1)/n) x (1/(n-1)) = 1/n

We can easily generalize above proof for any other position.

Find the row with maximum number of 1s

September 23, 2012

Given a boolean 2D array, where each row is sorted. Find the row with the maximum number of 1s.

```
Example
Input matrix
0 1 1 1
0 0 1 1
1 1 1 1  // this row has maximum 1s
0 0 0 0

Output: 2
```

A simple method is to do a row wise traversal of the matrix, count the number of 1s in each row and compare the count with max. Finally, return the index of row with maximum 1s. The time complexity of this method is O(m*n) where m is number of rows and n is number of columns in matrix.

We can do better. Since each row is sorted, we can **use Binary Search** to count of 1s in each row. We find the index of first instance of 1 in each row. The count of 1s will be equal to total number of columns minus the index of first 1.

See the following code for implementation of the above approach.

```
#include <stdio.h>
#define R 4
#define C 4

/* A function to find the index of first index of 1 in a boolean array
arr[] */
int first(bool arr[], int low, int high)
{
  if(high >= low)
  {
    // get the middle index
    int mid = low + (high - low)/2;

    // check if the element at middle index is first 1
    if ( ( mid == 0 || arr[mid-1] == 0) && arr[mid] == 1)
      return mid;

    // if the element is 0, recur for right side
    else if (arr[mid] == 0)
      return first(arr, (mid + 1), high);

    else // If element is not first 1, recur for left side
      return first(arr, low, (mid -1));
  }
  return -1;
}

// The main function that returns index of row with maximum number of 1s.
```

```
int rowWithMax1s(bool mat[R][C])
{
    int max_row_index = 0, max = -1; // Initialize max values

    // Traverse for each row and count number of 1s by finding the index
    // of first 1
    int i, index;
    for (i = 0; i < R; i++)
    {
       index = first (mat[i], 0, C-1);
       if (index != -1 && C-index > max)
       {
           max = C - index;
           max_row_index = i;
       }
    }

    return max_row_index;
}

/* Driver program to test above functions */
int main()
{
    bool mat[R][C] = { {0, 0, 0, 1},
        {0, 1, 1, 1},
        {1, 1, 1, 1},
        {0, 0, 0, 0}
    };

    printf("Index of row with maximum 1s is %d \n", rowWithMax1s(mat));

    return 0;
}
```

Output:

```
Index of row with maximum 1s is 2
```

Time Complexity: O(mLogn) where m is number of rows and n is number of columns in matrix.

The above solution **can be optimized further**. Instead of doing binary search in every row, we first check whether the row has more 1s than max so far. If the row has more 1s, then only count 1s in the row. Also, to count 1s in a row, we don't do binary search in complete row, we do search in before the index of last max.

Following is an optimized version of the above solution.

```
// The main function that returns index of row with maximum number of 1s.
int rowWithMax1s(bool mat[R][C])
{
    int i, index;

    // Initialize max using values from first row.
    int max_row_index = 0;
    int max = C - first(mat[0], 0, C-1);
```

```
    // Traverse for each row and count number of 1s by finding the index
    // of first 1
    for (i = 1; i < R; i++)
    {
        // Count 1s in this row only if this row has more 1s than
        // max so far
        if (mat[i][C-max-1] == 1)
        {
            // Note the optimization here also
            index = first (mat[i], 0, C-max);

            if (index != -1 && C-index > max)
            {
                max = C - index;
                max_row_index = i;
            }
        }
    }
    return max_row_index;
}
```

The worst case time complexity of the above optimized version is also O(mLogn), the will solution work better on average. Thanks to Naveen Kumar Singh for suggesting the above solution.

Sources: this and this

The worst case of the above solution occurs for a matrix like following.
0 0 0 … 0 1
0 0 0 ..0 1 1
0 … 0 1 1 1
….0 1 1 1 1

Following method works in O(m+n) time complexity in worst case.

Step1: Get the index of first (or leftmost) 1 in the first row.

Step2: Do following for every row after the first row
…IF the element on left of previous leftmost 1 is 0, ignore this row.
…ELSE Move left until a 0 is found. Update the leftmost index to this index and max_row_index to be the current row.

The time complexity is O(m+n) because we can possibly go as far left as we came ahead in the first step.

Following is C++ implementation of this method.

```
// The main function that returns index of row with maximum number of 1s.
int rowWithMax1s(bool mat[R][C])
{
    // Initialize first row as row with max 1s
    int max_row_index = 0;
```

```
    // The function first() returns index of first 1 in row 0.
    // Use this index to initialize the index of leftmost 1 seen so far
    int j = first(mat[0], 0, C-1) - 1;
    if (j == -1) // if 1 is not present in first row
      j = C - 1;

    for (int i = 1; i < R; i++)
    {
        // Move left until a 0 is found
        while (j >= 0 && mat[i][j] == 1)
        {
           j = j-1;  // Update the index of leftmost 1 seen so far
           max_row_index = i;  // Update max_row_index
        }
    }
    return max_row_index;
}
```

Thanks to Tylor, Ankan and Palash for their inputs.

Replace every element with the next greatest

August 30, 2012

Given an array of integers, replace every element with the next greatest element (greatest element on the right side) in the array. Since there is no element next to the last element, replace it with -1. For example, if the array is {16, 17, 4, 3, 5, 2}, then it should be modified to {17, 5, 5, 5, 2, -1}.

The question is very similar to this post and solutions are also similar.

A **naive method** is to run two loops. The outer loop will one by one pick array elements from left to right. The inner loop will find the greatest element present after the picked element. Finally the outer loop will replace the picked element with the greatest element found by inner loop. The time complexity of this method will be O(n*n).
A **tricky method** is to replace all elements using one traversal of the array. The idea is to start from the rightmost element, move to the left side one by one, and keep track of the maximum element. Replace every element with the maximum element.

```
#include <stdio.h>

/* Function to replace every element with the
   next greatest element */
void nextGreatest(int arr[], int size)
{
  // Initialize the next greatest element
  int max_from_right =  arr[size-1];

  // The next greatest element for the rightmost element
  // is always -1
  arr[size-1] = -1;
```

```
  // Replace all other elements with the next greatest
  for(int i = size-2; i >= 0; i--)
  {
    // Store the current element (needed later for updating
    // the next greatest element)
    int temp = arr[i];

    // Replace current element with the next greatest
    arr[i] = max_from_right;

    // Update the greatest element, if needed
    if(max_from_right < temp)
       max_from_right = temp;
  }
}

/* A utility Function that prints an array */
void printArray(int arr[], int size)
{
  int i;
  for (i=0; i < size; i++)
    printf("%d ", arr[i]);
  printf("\n");
}

/* Driver program to test above function */
int main()
{
  int arr[] = {16, 17, 4, 3, 5, 2};
  int size = sizeof(arr)/sizeof(arr[0]);
  nextGreatest (arr, size);
  printf ("The modified array is: \n");
  printArray (arr, size);
  return (0);
}
```

Output:

```
The modified array is:
17 5 5 5 2 -1
```

Time Complexity: O(n) where n is the number of elements in array.

Find a pair with the given difference

July 27, 2012

Given an unsorted array and a number n, find if there exists a pair of elements in the array whose difference is n.

```
Examples:
Input: arr[] = {5, 20, 3, 2, 50, 80}, n = 78
Output: Pair Found: (2, 80)
```

```
Input: arr[] = {90, 70, 20, 80, 50}, n = 45
Output: No Such Pair
```

Source: find pair

The simplest method is to run two loops, the outer loop picks the first element (smaller element) and the inner loop looks for the element picked by outer loop plus n. Time complexity of this method is O(n^2).

We can use sorting and Binary Search to improve time complexity to O(nLogn). The first step is to sort the array in ascending order. Once the array is sorted, traverse the array from left to right, and for each element arr[i], binary search for arr[i] + n in arr[i+1..n-1]. If the element is found, return the pair.
Both first and second steps take O(nLogn). So overall complexity is O(nLogn).

The second step of the above algorithm can be improved to O(n). The first step remain same. The idea for second step is take two index variables i and j, initialize them as 0 and 1 respectively. Now run a linear loop. If arr[j] – arr[i] is smaller than n, we need to look for greater arr[j], so increment j. If arr[j] – arr[i] is greater than n, we need to look for greater arr[i], so increment i. Thanks to Aashish Barnwal for suggesting this approach.
The following code is only for the second step of the algorithm, it assumes that the array is already sorted.

```
#include <stdio.h>

// The function assumes that the array is sorted
bool findPair(int arr[], int size, int n)
{
    // Initialize positions of two elements
    int i = 0;
    int j = 1;

    // Search for a pair
    while (i<size && j<size)
    {
        if (i != j && arr[j]-arr[i] == n)
        {
            printf("Pair Found: (%d, %d)", arr[i], arr[j]);
            return true;
        }
        else if (arr[j]-arr[i] < n)
            j++;
        else
            i++;
    }

    printf("No such pair");
    return false;
}

// Driver program to test above function
int main()
{
    int arr[] = {1, 8, 30, 40, 100};
    int size = sizeof(arr)/sizeof(arr[0]);
    int n = 60;
```

```
    findPair(arr, size, n);
    return 0;
}
```

Output:

```
Pair Found: (40, 100)
```

Hashing can also be used to solve this problem. Create an empty hash table HT. Traverse the array, use array elements as hash keys and enter them in HT. Traverse the array again look for value n + arr[i] in HT.

Maximum Product Subarray

July 23, 2012

Given an array that contains both positive and negative integers, find the product of the maximum product subarray. Expected Time complexity is O(n) and only O(1) extra space can be used.

Examples:

```
Input: arr[] = {6, -3, -10, 0, 2}
Output:   180  // The subarray is {6, -3, -10}

Input: arr[] = {-1, -3, -10, 0, 60}
Output:   60  // The subarray is {60}

Input: arr[] = {-2, -3, 0, -2, -40}
Output:   80  // The subarray is {-2, -40}
```

The following solution assumes that the given input array always has a positive ouput. The solution works for all cases mentioned above. It doesn't work for arrays like {0, 0, -20, 0}, {0, 0, 0}.. etc. The solution can be easily modified to handle this case.
It is similar to Largest Sum Contiguous Subarray problem. The only thing to note here is, maximum product can also be obtained by minimum (negative) product ending with the previous element multiplied by this element. For example, in array {12, 2, -3, -5, -6, -2}, when we are at element -2, the maximum product is multiplication of, minimum product ending with -6 and -2.

```
#include <stdio.h>

// Utility functions to get minimum of two integers
int min (int x, int y) {return x < y? x : y; }

// Utility functions to get maximum of two integers
int max (int x, int y) {return x > y? x : y; }

/* Returns the product of max product subarray.  Assumes that the
```

```
   given array always has a subarray with product more than 1 */
int maxSubarrayProduct(int arr[], int n)
{
    // max positive product ending at the current position
    int max_ending_here = 1;

    // min negative product ending at the current position
    int min_ending_here = 1;

    // Initialize overall max product
    int max_so_far = 1;

    /* Traverse throught the array. Following values are maintained after
the ith iteration:
       max_ending_here is always 1 or some positive product ending with
arr[i]
       min_ending_here is always 1 or some negative product ending with
arr[i] */
    for (int i = 0; i < n; i++)
    {
        /* If this element is positive, update max_ending_here. Update
           min_ending_here only if min_ending_here is negative */
        if (arr[i] > 0)
        {
            max_ending_here = max_ending_here*arr[i];
            min_ending_here = min (min_ending_here * arr[i], 1);
        }

        /* If this element is 0, then the maximum product cannot
           end here, make both max_ending_here and min_ending_here 0
           Assumption: Output is alway greater than or equal to 1. */
        else if (arr[i] == 0)
        {
            max_ending_here = 1;
            min_ending_here = 1;
        }

        /* If element is negative. This is tricky
           max_ending_here can either be 1 or positive. min_ending_here can
either be 1
           or negative.
           next min_ending_here will always be prev. max_ending_here *
arr[i]
           next max_ending_here will be 1 if prev min_ending_here is 1,
otherwise
           next max_ending_here will be prev min_ending_here * arr[i] */
        else
        {
            int temp = max_ending_here;
            max_ending_here = max (min_ending_here * arr[i], 1);
            min_ending_here = temp * arr[i];
        }

        // update max_so_far, if needed
        if (max_so_far <  max_ending_here)
          max_so_far  =  max_ending_here;
    }

    return max_so_far;
}
```

```
// Driver Program to test above function
int main()
{
    int arr[] = {1, -2, -3, 0, 7, -8, -2};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Maximum Sub array product is %d", maxSubarrayProduct(arr, n));
    return 0;
}
```

Output:

```
Maximum Sub array product is 112
```

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

Find the maximum element in an array which is first increasing and then decreasing

January 14, 2012

Given an array of integers which is initially increasing and then decreasing, find the maximum value in the array.

```
Input: arr[] = {8, 10, 20, 80, 100, 200, 400, 500, 3, 2, 1}
Output: 500

Input: arr[] = {1, 3, 50, 10, 9, 7, 6}
Output: 50

Corner case (No decreasing part)
Input: arr[] = {10, 20, 30, 40, 50}
Output: 50

Corner case (No increasing part)
Input: arr[] = {120, 100, 80, 20, 0}
Output: 120
```

Method 1 (Linear Search)
We can traverse the array and keep track of maximum and element. And finally return the maximum element.

```
#include <stdio.h>

int findMaximum(int arr[], int low, int high)
{
   int max = arr[low];
   int i;
   for (i = low; i <= high; i++)
   {
       if (arr[i] > max)
```

```
            max = arr[i];
    }
    return max;
}

/* Driver program to check above functions */
int main()
{
    int arr[] = {1, 30, 40, 50, 60, 70, 23, 20};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("The maximum element is %d", findMaximum(arr, 0, n-1));
    getchar();
    return 0;
}
```

Time Complexity: O(n)

Method 2 (Binary Search)
We can modify the standard Binary Search algorithm for the given type of arrays.
i) If the mid element is greater than both of its adjacent elements, then mid is the maximum.
ii) If mid element is greater than its next element and smaller than the previous element then maximum lies on left side of mid. Example array: {3, 50, 10, 9, 7, 6}
iii) If mid element is smaller than its next element and greater than the previous element then maximum lies on right side of mid. Example array: {2, 4, 6, 8, 10, 3, 1}

```
#include <stdio.h>

int findMaximum(int arr[], int low, int high)
{

    /* Base Case: Only one element is present in arr[low..high]*/
    if (low == high)
      return arr[low];

    /* If there are two elements and first is greater then
       the first element is maximum */
    if ((high == low + 1) && arr[low] >= arr[high])
       return arr[low];

    /* If there are two elements and second is greater then
       the second element is maximum */
    if ((high == low + 1) && arr[low] < arr[high])
       return arr[high];

    int mid = (low + high)/2;   /*low + (high - low)/2;*/

    /* If we reach a point where arr[mid] is greater than both of
      its adjacent elements arr[mid-1] and arr[mid+1], then arr[mid]
      is the maximum element*/
    if ( arr[mid] > arr[mid + 1] && arr[mid] > arr[mid - 1])
       return arr[mid];

    /* If arr[mid] is greater than the next element and smaller than the
previous
```

```
    element then maximum lies on left side of mid */
   if (arr[mid] > arr[mid + 1] && arr[mid] < arr[mid - 1])
     return findMaximum(arr, low, mid-1);
   else // when arr[mid] is greater than arr[mid-1] and smaller than
arr[mid+1]
     return findMaximum(arr, mid + 1, high);
}

/* Driver program to check above functions */
int main()
{
   int arr[] = {1, 3, 50, 10, 9, 7, 6};
   int n = sizeof(arr)/sizeof(arr[0]);
   printf("The maximum element is %d", findMaximum(arr, 0, n-1));
   getchar();
   return 0;
}
```

Time Complexity: O(Logn)

This method works only for distinct numbers. For example, it will not work for an array like {0, 1, 1, 2, 2, 2, 2, 2, 3, 4, 4, 5, 3, 3, 2, 2, 1, 1}.

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 */
        if (l < n)
        {
            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;
}
```

Find the minimum distance between two numbers

June 24, 2011

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 1*

Time Complexity: O(n)

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

May 21, 2011

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.

Thanks to celicom for suggesting the algorithm for this method.

```
#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)

Count the number of occurrences in a sorted array

May 3, 2011

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

Find the smallest missing number

April 17, 2011

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

Thanks to Ravichandra for suggesting following two methods.

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)
If arr[0] is not 0, return 0. Otherwise traverse the input array starting from index 1, 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)
Thanks to yasein and Jams for suggesting this method.
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.

```
#include<stdio.h>

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)

Check if array elements are consecutive | Added Method 3

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 (n, sizeof(bool));
      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)

Next Greater Element

March 17, 2011

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;
        int top = ps->top;
        ps->items [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
    {
        int top = ps->top;
        temp = ps->items [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.

Maximum difference between two elements

Given an array arr[] of integers, find out the difference between any two elements **such that larger element appears after the smaller numbe**r in arr[].

Examples: If array is [2, 3, 10, 6, 4, 8, 1] then returned value should be 8 (Diff between 10 and 2). If array is [7, 9, 5, 6, 3, 2] then returned value should be 2 (Diff between 7 and 9)

Method 1 (Simple)
Use two loops. In the outer loop, pick elements one by one and in the inner loop calculate the difference of the picked element with every other element in the array and compare the difference with the maximum difference calculated so far.

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

```
/* The function assumes that there are at least two
   elements in array.
   The function returns a negative value if the array is
   sorted in decreasing order.
   Returns 0 if elements are equal */
int maxDiff(int arr[], int arr_size)
{
  int max_diff = arr[1] - arr[0];
  int i, j;
  for(i = 0; i < arr_size; i++)
  {
    for(j = i+1; j < arr_size; j++)
    {
      if(arr[j] - arr[i] > max_diff)
         max_diff = arr[j] - arr[i];
    }
  }
  return max_diff;
}

/* Driver program to test above function */
int main()
{
  int arr[] = {1, 2, 90, 10, 110};
  printf("Maximum difference is %d",  maxDiff(arr, 5));
  getchar();
  return 0;
}
```

Time Complexity: O(n^2)
Auxiliary Space: O(1)

Method 2 (Tricky and Efficient)
In this method, instead of taking difference of the picked element with every other element, we take the difference with the minimum element found so far. So we need to keep track of 2 things:
1) Maximum difference found so far (max_diff).
2) Minimum number visited so far (min_element).

```
#include<stdio.h>

/* The function assumes that there are at least two
   elements in array.
   The function returns a negative value if the array is
   sorted in decreasing order.
   Returns 0 if elements are equal  */
int maxDiff(int arr[], int arr_size)
{
  int max_diff = arr[1] - arr[0];
  int min_element = arr[0];
  int i;
  for(i = 1; i < arr_size; i++)
  {
    if(arr[i] - min_element > max_diff)
      max_diff = arr[i] - min_element;
    if(arr[i] < min_element)
         min_element = arr[i];
  }
```

```
  return max_diff;
}

/* Driver program to test above function */
int main()
{
  int arr[] = {1, 2, 6, 80, 100};
  int size = sizeof(arr)/sizeof(arr[0]);
  printf("Maximum difference is %d",  maxDiff(arr, size));
  getchar();
  return 0;
}
```

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

Method 3 (Another Tricky Solution)
First find the difference between the adjacent elements of the array and store all differences in an auxiliary array diff[] of size n-1. Now this problems turns into finding the maximum sum subarray of this difference array.
Thanks to Shubham Mittal for suggesting this solution.

```
#include<stdio.h>

int maxDiff(int arr[], int n)
{
    // Create a diff array of size n-1. The array will hold
    //  the difference of adjacent elements
    int diff[n-1];
    for (int i=0; i < n-1; i++)
        diff[i] = arr[i+1] - arr[i];

    // Now find the maximum sum subarray in diff array
    int max_diff = diff[0];
    for (int i=1; i<n-1; i++)
    {
        if (diff[i-1] > 0)
            diff[i] += diff[i-1];
        if (max_diff < diff[i])
            max_diff = diff[i];
    }
    return max_diff;
}

/* Driver program to test above function */
int main()
{
    int arr[] = {80, 2, 6, 3, 100};
    int size = sizeof(arr)/sizeof(arr[0]);
    printf("Maximum difference is %d",  maxDiff(arr, size));
    return 0;
}
```

Output:

```
98
```

This method is also O(n) time complexity solution, but it requires O(n) extra space

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

We can modify the above method to work in O(1) extra space. Instead of creating an auxiliary array, we can calculate diff and max sum in same loop. Following is the space optimized version.

```
int maxDiff (int arr[], int n)
{
    // Initialize diff, current sum and max sum
    int diff = arr[1]-arr[0];
    int curr_sum = diff;
    int max_sum = curr_sum;

    for(int i=1; i<n-1; i++)
    {
        // Calculate current diff
        diff = arr[i+1]-arr[i];

        // Calculate current sum
        if (curr_sum > 0)
           curr_sum += diff;
        else
           curr_sum = diff;

        // Update max sum, if needed
        if (curr_sum > max_sum)
           max_sum = curr_sum;
    }

    return max_sum;
}
```

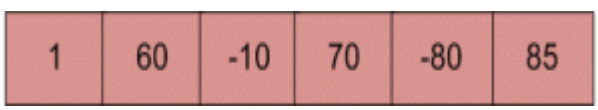
Time Complexity: O(n)
Auxiliary Space: O(1)

Two elements whose sum is closest to zero

January 10, 2010

Question: An Array of integers is given, both +ve and -ve. You need to find the two elements such that their sum is closest to zero.

For the below array, program should print -80 and 85.



METHOD 1 (Simple)
For each element, find the sum of it with every other element in the array and compare sums. Finally, return the minimum sum.

Implementation

```
# include <stdio.h>
# include <stdlib.h> /* for abs() */
# include <math.h>
void minAbsSumPair(int arr[], int arr_size)
{
  int inv_count = 0;
  int l, r, min_sum, sum, min_l, min_r;

  /* Array should have at least two elements*/
  if(arr_size < 2)
  {
    printf("Invalid Input");
    return;
  }

  /* Initialization of values */
  min_l = 0;
  min_r = 1;
  min_sum = arr[0] + arr[1];

  for(l = 0; l < arr_size - 1; l++)
  {
    for(r = l+1; r < arr_size; r++)
    {
      sum = arr[l] + arr[r];
      if(abs(min_sum) > abs(sum))
      {
        min_sum = sum;
        min_l = l;
        min_r = r;
      }
    }
  }

  printf(" The two elements whose sum is minimum are %d and %d",
         arr[min_l], arr[min_r]);
}

/* Driver program to test above function */
int main()
{
  int arr[] = {1, 60, -10, 70, -80, 85};
  minAbsSumPair(arr, 6);
  getchar();
  return 0;
}
```

Time complexity: O(n^2)

METHOD 2 (Use Sorting)

Thanks to baskin for suggesting this approach. We recommend to read this post for background of this approach.

Algorithm
1) Sort all the elements of the input array.
2) Use two index variables l and r to traverse from left and right ends respectively. Initialize l as 0 and r as n-1.
3) sum = a[l] + a[r]
4) If sum is -ve, then l++
5) If sum is +ve, then r–
6) Keep track of abs min sum.
7) Repeat steps 3, 4, 5 and 6 while l < r

Implementation

```
# include <stdio.h>
# include <math.h>
# include <limits.h>

void quickSort(int *, int, int);

/* Function to print pair of elements having minimum sum */
void minAbsSumPair(int arr[], int n)
{
  // Variables to keep track of current sum and minimum sum
  int sum, min_sum = INT_MAX;

  // left and right index variables
  int l = 0, r = n-1;

  // variable to keep track of the left and right pair for min_sum
  int min_l = l, min_r = n-1;

  /* Array should have at least two elements*/
  if(n < 2)
  {
    printf("Invalid Input");
    return;
  }

  /* Sort the elements */
  quickSort(arr, l, r);

  while(l < r)
  {
    sum = arr[l] + arr[r];

    /*If abs(sum) is less then update the result items*/
    if(abs(sum) < abs(min_sum))
    {
      min_sum = sum;
      min_l = l;
      min_r = r;
    }
    if(sum < 0)
      l++;
    else
      r--;
```

```
  }

  printf(" The two elements whose sum is minimum are %d and %d",
         arr[min_l], arr[min_r]);
}

/* Driver program to test above function */
int main()
{
  int arr[] = {1, 60, -10, 70, -80, 85};
  int n = sizeof(arr)/sizeof(arr[0]);
  minAbsSumPair(arr, n);
  getchar();
  return 0;
}

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

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

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

  exchange (&arr[i + 1], &arr[ei]);
  return (i + 1);
}

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

Time Complexity: complexity to sort + complexity of finding the optimum pair = O(nlogn) + O(n) = O(nlogn)

Asked by Vineet

Maximum sum such that no two elements are adjacent

Question: Given an array of positive numbers, find the maximum sum of a subsequence with the constraint that no 2 numbers in the sequence should be adjacent in the array. So 3 2 7 10 should return 13 (sum of 3 and 10) or 3 2 5 10 7 should return 15 (sum of 3, 5 and 7).Answer the question in most efficient way.

Algorithm:
Loop for all elements in arr[] and maintain two sums incl and excl where incl = Max sum including the previous element and excl = Max sum excluding the previous element.

Max sum excluding the current element will be max(incl, excl) and max sum including the current element will be excl + current element (Note that only excl is considered because elements cannot be adjacent).

At the end of the loop return max of incl and excl.

Example:

```
  arr[] = {5,  5, 10, 40, 50, 35}

  inc = 5
  exc = 0

  For i = 1 (current element is 5)
  incl =  (excl + arr[i])  = 5
  excl =  max(5, 0) = 5

  For i = 2 (current element is 10)
  incl =  (excl + arr[i]) = 15
  excl =  max(5, 5) = 5

  For i = 3 (current element is 40)
  incl = (excl + arr[i]) = 45
  excl = max(5, 15) = 15

  For i = 4 (current element is 50)
  incl = (excl + arr[i]) = 65
  excl =  max(45, 15) = 45

  For i = 5 (current element is 35)
  incl =  (excl + arr[i]) = 80
  excl = max(5, 15) = 65

And 35 is the last element. So, answer is max(incl, excl) =  80
```

Thanks to Debanjan for providing code.

Implementation:

```
#include<stdio.h>

/*Function to return max sum such that no two elements
 are adjacent */
int FindMaxSum(int arr[], int n)
{
  int incl = arr[0];
  int excl = 0;
  int excl_new;
  int i;

  for (i = 1; i < n; i++)
  {
     /* current max excluding i */
     excl_new = (incl > excl)? incl: excl;

     /* current max including i */
     incl = excl + arr[i];
     excl = excl_new;
  }

   /* return max of incl and excl */
   return ((incl > excl)? incl : excl);
}

/* Driver program to test above function */
int main()
{
  int arr[] = {5, 5, 10, 100, 10, 5};
  printf("%d \n", FindMaxSum(arr, 6));
  getchar();
  return 0;
}
```

Time Complexity: O(n)

Search an element in a sorted and pivoted array

Question:
An element in a sorted array can be found in O(log n) time via binary search. But suppose I rotate the sorted array at some pivot unknown to you beforehand. So for instance, 1 2 3 4 5 might become 3 4 5 1 2. Devise a way to find an element in the rotated array in O(log n) time.

Solution:
Thanks to Ajay Mishra for initial solution.

Algorithm:
Find the pivot point, divide the array in two sub-arrays and call binary search.
The main idea for finding pivot is – for a sorted (in increasing order) and pivoted array, pivot element is the only only element for which next element to it is smaller than it.
Using above criteria and binary search methodology we can get pivot element in O(logn) time

```
Input arr[] = {3, 4, 5, 1, 2}
Element to Search = 1
  1) Find out pivot point and divide the array in two
      sub-arrays. (pivot = 2) /*Index of 5*/
  2) Now call binary search for one of the two sub-arrays.
      (a) If element is greater than 0th element then
             search in left array
      (b) Else Search in right array
          (1 will go in else as 1 < 0th element(3))
  3) If element is found in selected sub-array then return index
     Else return -1.
```

Implementation:

```
/* Program to search an element in a sorted and pivoted array*/
#include <stdio.h>

int findPivot(int[], int, int);
int binarySearch(int[], int, int, int);

/* Searches an element no in a pivoted sorted array arrp[]
   of size arr_size */
int pivotedBinarySearch(int arr[], int arr_size, int no)
{
   int pivot = findPivot(arr, 0, arr_size-1);

   // If we didn't find a pivot, then array is not rotated at all
   if (pivot == -1)
     return binarySearch(arr, 0, arr_size-1, no);

   // If we found a pivot, then first compare with pivot and then
   // search in two subarrays around pivot
   if (arr[pivot] == no)
     return pivot;
   if (arr[0] <= no)
     return binarySearch(arr, 0, pivot-1, no);
   else
     return binarySearch(arr, pivot+1, arr_size-1, no);
```

```
}

/* Function to get pivot. For array 3, 4, 5, 6, 1, 2 it will
   return 3. If array is not rotated at all, then it returns -1 */
int findPivot(int arr[], int low, int high)
{
   // base cases
   if (high < low)  return -1;
   if (high == low) return low;

   int mid = (low + high)/2;   /*low + (high - low)/2;*/
   if (mid < high && arr[mid] > arr[mid + 1])
     return mid;
   if (mid > low && arr[mid] < arr[mid - 1])
     return (mid-1);
   if (arr[low] >= arr[mid])
     return findPivot(arr, low, mid-1);
   else
     return findPivot(arr, mid + 1, high);
}

/* Standard Binary Search function*/
int binarySearch(int arr[], int low, int high, int no)
{
   if (high < low)
       return -1;
   int mid = (low + high)/2;  /*low + (high - low)/2;*/
   if (no == arr[mid])
     return mid;
   if (no > arr[mid])
     return binarySearch(arr, (mid + 1), high, no);
   else
     return binarySearch(arr, low, (mid -1), no);
}


/* Driver program to check above functions */
int main()
{
   // Let us search 3 in below array
   int arr1[] = {5, 6, 7, 8, 9, 10, 1, 2, 3};
   int arr_size = sizeof(arr1)/sizeof(arr1[0]);
   int no = 3;
   printf("Index of the element is %d\n",  pivotedBinarySearch(arr1,
arr_size, no));

    // Let us search 3 in below array
   int arr2[] = {3, 4, 5, 1, 2};
   arr_size = sizeof(arr2)/sizeof(arr2[0]);
   printf("Index of the element is %d\n",  pivotedBinarySearch(arr2,
arr_size, no));

   // Let us search for 4 in above array
   no = 4;
   printf("Index of the element is %d\n",  pivotedBinarySearch(arr2,
arr_size, no));

   // Let us search 0 in below array
   int arr3[] = {1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1};
   no = 0;
   arr_size = sizeof(arr3)/sizeof(arr3[0]);
```

```
   printf("Index of the element is %d\n",  pivotedBinarySearch(arr3,
arr_size, no));

   // Let us search 3 in below array
   int arr4[] = {2, 3, 0, 2, 2, 2, 2, 2, 2, 2};
   no = 3;
   arr_size = sizeof(arr4)/sizeof(arr4[0]);
   printf("Index of the element is %d\n",  pivotedBinarySearch(arr4,
arr_size, no));

   // Let us search 2 in above array
   no = 2;
   printf("Index of the element is %d\n",  pivotedBinarySearch(arr4,
arr_size, no));

   // Let us search 3 in below array
   int arr5[] = {1, 2, 3, 4};
   no = 3;
   arr_size = sizeof(arr5)/sizeof(arr5[0]);
   printf("Index of the element is %d\n",  pivotedBinarySearch(arr5,
arr_size, no));

   return 0;
}
```

Output:

```
Index of the element is 8
Index of the element is 0
Index of the element is 1
Index of the element is 3
Index of the element is 1
Index of the element is 0
Index of the element is 2
```

Please note that the solution may not work for cases where the input array has duplicates.

Time Complexity O(logn)

Find number of pairs such that x^y > y^x

October 21, 2013

Given two arrays X[] and Y[] of positive integers, find number of pairs such that **x^y > y^x** where x is an element from X[] and y is an element from Y[].

Examples:

```
  Input: X[] = {2, 1, 6}, Y = {1, 5}
  Output: 3
  // There are total 3 pairs where pow(x, y) is greater than pow(y, x)
  // Pairs are (2, 1), (2, 5) and (6, 1)
```

```
Input: X[] = {10, 19, 18}, Y[] = {11, 15, 9};
Output: 2
// There are total 2 pairs where pow(x, y) is greater than pow(y, x)
// Pairs are (10, 11) and (10, 15)
```

The **brute force solution** is to consider each element of X[] and Y[], and check whether the given condition satisfies or not. Time Complexity of this solution is **O(m*n)** where m and n are sizes of given arrays.

Following is C++ code based on brute force solution.

```
int countPairsBruteForce(int X[], int Y[], int m, int n)
{
    int ans = 0;
    for (int i = 0; i < m; i++)
       for (int j = 0; j < n; j++)
          if (pow(X[i], Y[j]) > pow(Y[j], X[i]))
              ans++;
    return ans;
}
```

Efficient Solution:
The problem can be solved in **O(nLogn + mLogn)** time. The trick here is, if y > x then x^y > y^x with some exceptions. Following are simple steps based on this trick.

1) Sort array Y[].
2) For every x in X[], find the index idx of smallest number greater than x (also called ceil of x) in Y[] using binary search or we can use the inbuilt function upper_bound() in algorithm library.
3) All the numbers after idx satisfy the relation so just add (n-idx) to the count.

Base Cases and Exceptions:
Following are exceptions for x from X[] and y from Y[]
If x = 0, then the count of pairs for this x is 0.
If x = 1, then the count of pairs for this x is equal to count of 0s in Y[].
The following cases must be handled separately as they don't follow the general rule that x smaller than y means x^y is greater than y^x.
a) x = 2, y = 3 or 4
b) x = 3, y = 2
Note that the case where x = 4 and y = 2 is not there

Following diagram shows all exceptions in tabular form. The value 1 indicates that the corresponding (x, y) form a valid pair.

Y

X	0	1	2	3	4
0	0	0	0	0	0
1	1	0	0	0	0
2	1	1	0	0	0
3	1	1	1	0	1
4	1	1	0	0	0

Following is C++ implementation. In the following implementation, we pre-process the Y array and count 0, 1, 2, 3 and 4 in it, so that we can handle all exceptions in constant time. The array NoOfY[] is used to store the counts.

```
#include<iostream>
#include<algorithm>
using namespace std;

// This function return count of pairs with x as one element
// of the pair. It mainly looks for all values in Y[] where
// x ^ Y[i] > Y[i] ^ x
int count(int x, int Y[], int n, int NoOfY[])
{
    // If x is 0, then there cannot be any value in Y such that
    // x^Y[i] > Y[i]^x
    if (x == 0) return 0;

    // If x is 1, then the number of pais is equal to number of
    // zeroes in Y[]
    if (x == 1) return NoOfY[0];

    // Find number of elements in Y[] with values greater than x
    // upper_bound() gets address of first greater element in Y[0..n-1]
    int* idx = upper_bound(Y, Y + n, x);
    int ans = (Y + n) - idx;

    // If we have reached here, then x must be greater than 1,
    // increase number of pairs for y=0 and y=1
    ans += (NoOfY[0] + NoOfY[1]);

    // Decrease number of pairs for x=2 and (y=4 or y=3)
    if (x == 2)  ans -= (NoOfY[3] + NoOfY[4]);

    // Increase number of pairs for x=3 and y=2
    if (x == 3)  ans += NoOfY[2];

    return ans;
}

// The main function that returns count of pairs (x, y) such that
// x belongs to X[], y belongs to Y[] and x^y > y^x
```

```
int countPairs(int X[], int Y[], int m, int n)
{
    // To store counts of 0, 1, 2, 3 and 4 in array Y
    int NoOfY[5] = {0};
    for (int i = 0; i < n; i++)
        if (Y[i] < 5)
            NoOfY[Y[i]]++;

    // Sort Y[] so that we can do binary search in it
    sort(Y, Y + n);

    int total_pairs = 0; // Initialize result

    // Take every element of X and count pairs with it
    for (int i=0; i<m; i++)
        total_pairs += count(X[i], Y, n, NoOfY);

    return total_pairs;
}

// Driver program to test above functions
int main()
{
    int X[] = {2, 1, 6};
    int Y[] = {1, 5};

    int m = sizeof(X)/sizeof(X[0]);
    int n = sizeof(Y)/sizeof(Y[0]);

    cout << "Total pairs = " << countPairs(X, Y, m, n);

    return 0;
}
```

Output:

```
Total pairs = 3
```

Time Complexity : Let m and n be the sizes of arrays X[] and Y[] respectively. The sort step takes O(nLogn) time. Then every element of X[] is searched in Y[] using binary search. This step takes O(mLogn) time. Overall time complexity is O(nLogn + mLogn).

Merge k sorted arrays | Set 1

July 29, 2013

Given k sorted arrays of size n each, merge them and print the sorted output.

Example:

Input:
```
k = 3, n =  4
arr[][] = { {1, 3, 5, 7},
            {2, 4, 6, 8},
            {0, 9, 10, 11}} ;
```

Output: 0 1 2 3 4 5 6 7 8 9 10 11

A simple solution is to create an output array of size n*k and one by one copy all arrays to it. Finally, sort the output array using any O(nLogn) sorting algorithm. This approach takes O(nkLognk) time.

We can merge arrays in O(nk*Logk) time using Mean Heap. Following is detailed algorithm.

1. Create an output array of size n*k.
2. Create a min heap of size k and insert 1st element in all the arrays into a the heap
3. Repeat following steps n*k times.
a) Get minimum element from heap (minimum is always at root) and store it in output array.
b) Replace heap root with next element from the array from which the element is extracted. If the array doesn't have any more elements, then replace root with infinite. After replacing the root, heapify the tree.

Following is C++ implementation of the above algorithm.

```
// C++ program to merge k sorted arrays of size n each.
#include<iostream>
#include<limits.h>
using namespace std;

#define n 4

// A min heap node
struct MinHeapNode
{
    int element; // The element to be stored
    int i; // index of the array from which the element is taken
    int j; // index of the next element to be picked from array
};

// Prototype of a utility function to swap two min heap nodes
void swap(MinHeapNode *x, MinHeapNode *y);

// A class for Min Heap
class MinHeap
{
    MinHeapNode *harr; // pointer to array of elements in heap
    int heap_size; // size of min heap
public:
    // Constructor: creates a min heap of given size
    MinHeap(MinHeapNode a[], int size);

    // to heapify a subtree with root at given index
    void MinHeapify(int );

    // to get index of left child of node at index i
```

```
    int left(int i) { return (2*i + 1); }

    // to get index of right child of node at index i
    int right(int i) { return (2*i + 2); }

    // to get the root
    MinHeapNode getMin() { return harr[0]; }

    // to replace root with new node x and heapify() new root
    void replaceMin(MinHeapNode x) { harr[0] = x;  MinHeapify(0); }
};

// This function takes an array of arrays as an argument and
// All arrays are assumed to be sorted. It merges them together
// and prints the final sorted output.
int *mergeKArrays(int arr[][n], int k)
{
    int *output = new int[n*k];  // To store output array

    // Create a min heap with k heap nodes.  Every heap node
    // has first element of an array
    MinHeapNode *harr = new MinHeapNode[k];
    for (int i = 0; i < k; i++)
    {
        harr[i].element = arr[i][0]; // Store the first element
        harr[i].i = i;  // index of array
        harr[i].j = 1;  // Index of next element to be stored from array
    }
    MinHeap hp(harr, k); // Create the heap

    // Now one by one get the minimum element from min
    // heap and replace it with next element of its array
    for (int count = 0; count < n*k; count++)
    {
        // Get the minimum element and store it in output
        MinHeapNode root = hp.getMin();
        output[count] = root.element;

        // Find the next elelement that will replace current
        // root of heap. The next element belongs to same
        // array as the current root.
        if (root.j < n)
        {
            root.element = arr[root.i][root.j];
            root.j += 1;
        }
        // If root was the last element of its array
        else root.element =  INT_MAX; //INT_MAX is for infinite

        // Replace root with next element of array
        hp.replaceMin(root);
    }

    return output;
}

// FOLLOWING ARE IMPLEMENTATIONS OF STANDARD MIN HEAP METHODS
// FROM CORMEN BOOK
// Constructor: Builds a heap from a given array a[] of given size
```

```
MinHeap::MinHeap(MinHeapNode a[], int size)
{
    heap_size = size;
    harr = a;  // store address of array
    int i = (heap_size - 1)/2;
    while (i >= 0)
    {
        MinHeapify(i);
        i--;
    }
}

// A recursive method to heapify a subtree with root at given index
// This method assumes that the subtrees are already heapified
void MinHeap::MinHeapify(int i)
{
    int l = left(i);
    int r = right(i);
    int smallest = i;
    if (l < heap_size && harr[l].element < harr[i].element)
        smallest = l;
    if (r < heap_size && harr[r].element < harr[smallest].element)
        smallest = r;
    if (smallest != i)
    {
        swap(&harr[i], &harr[smallest]);
        MinHeapify(smallest);
    }
}

// A utility function to swap two elements
void swap(MinHeapNode *x, MinHeapNode *y)
{
    MinHeapNode temp = *x;  *x = *y;  *y = temp;
}

// A utility function to print array elements
void printArray(int arr[], int size)
{
   for (int i=0; i < size; i++)
       cout << arr[i] << " ";
}

// Driver program to test above functions
int main()
{
    // Change n at the top to change number of elements
    // in an array
    int arr[][n] =  {{2, 6, 12, 34},
                     {1, 9, 20, 1000},
                     {23, 34, 90, 2000}};
    int k = sizeof(arr)/sizeof(arr[0]);

    int *output = mergeKArrays(arr, k);

    cout << "Merged array is " << endl;
    printArray(output, n*k);

    return 0;
}
```

Output:

```
Merged array is
1 2 6 9 12 20 23 34 34 90 1000 2000
```

Time Complexity: The main step is 3rd step, the loop runs n*k times. In every iteration of loop, we call heapify which takes O(Logk) time. Therefore, the time complexity is O(nk Logk).

There are other interesting methods to merge k sorted arrays in O(nkLogk), we will sonn be discussing them as separate posts.

Find the minimum element in a sorted and rotated array

A sorted array is rotated at some unknown point, find the minimum element in it.

Following solution assumes that all elements are distinct.

Examples

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

Input: {1, 2, 3, 4}
Output: 1

Input: {2, 1}
Output: 1
```

A simple solution is to traverse the complete array and find minimum. This solution requires time.
We can do it in O(Logn) using Binary Search. If we take a closer look at above examples, we can easily figure out following pattern: The minimum element is the only element whose previous element is greater than it. If there is no such element, then there is no rotation and first element is the minimum element. Therefore, we do binary search for an element which is smaller than the previous element. We strongly recommend you to try it yourself before seeing the following C implementation.

```
// C program to find minimum element in a sorted and rotated array
#include <stdio.h>

int findMin(int arr[], int low, int high)
{
    // This condition is needed to handle the case when array is not
    // rotated at all
    if (high < low)  return arr[0];

    // If there is only one element left
    if (high == low) return arr[low];

    // Find mid
    int mid = low + (high - low)/2; /*(low + high)/2;*/
```

```
    // Check if element (mid+1) is minimum element. Consider
    // the cases like {3, 4, 5, 1, 2}
    if (mid < high && arr[mid+1] < arr[mid])
       return arr[mid+1];

    // Check if mid itself is minimum element
    if (mid > low && arr[mid] < arr[mid - 1])
       return arr[mid];

    // Decide whether we need to go to left half or right half
    if (arr[high] > arr[mid])
       return findMin(arr, low, mid-1);
    return findMin(arr, mid+1, high);
}

// Driver program to test above functions
int main()
{
    int arr1[] =  {5, 6, 1, 2, 3, 4};
    int n1 = sizeof(arr1)/sizeof(arr1[0]);
    printf("The minimum element is %d\n", findMin(arr1, 0, n1-1));

    int arr2[] =  {1, 2, 3, 4};
    int n2 = sizeof(arr2)/sizeof(arr2[0]);
    printf("The minimum element is %d\n", findMin(arr2, 0, n2-1));

    int arr3[] =  {1};
    int n3 = sizeof(arr3)/sizeof(arr3[0]);
    printf("The minimum element is %d\n", findMin(arr3, 0, n3-1));

    int arr4[] =  {1, 2};
    int n4 = sizeof(arr4)/sizeof(arr4[0]);
    printf("The minimum element is %d\n", findMin(arr4, 0, n4-1));

    int arr5[] =  {2, 1};
    int n5 = sizeof(arr5)/sizeof(arr5[0]);
    printf("The minimum element is %d\n", findMin(arr5, 0, n5-1));

    int arr6[] =  {5, 6, 7, 1, 2, 3, 4};
    int n6 = sizeof(arr6)/sizeof(arr6[0]);
    printf("The minimum element is %d\n", findMin(arr6, 0, n6-1));

    int arr7[] =  {1, 2, 3, 4, 5, 6, 7};
    int n7 = sizeof(arr7)/sizeof(arr7[0]);
    printf("The minimum element is %d\n", findMin(arr7, 0, n7-1));

    int arr8[] =  {2, 3, 4, 5, 6, 7, 8, 1};
    int n8 = sizeof(arr8)/sizeof(arr8[0]);
    printf("The minimum element is %d\n", findMin(arr8, 0, n8-1));

    int arr9[] =  {3, 4, 5, 1, 2};
    int n9 = sizeof(arr9)/sizeof(arr9[0]);
    printf("The minimum element is %d\n", findMin(arr9, 0, n9-1));

    return 0;
}
```

Output:

```
The minimum element is 1
The minimum element is 1
The minimum element is 1
The minimum element is 1
The minimum element is 1
The minimum element is 1
The minimum element is 1
The minimum element is 1
The minimum element is 1
```

How to handle duplicates?
It turned out that duplicates can't be handled in O(Logn) time in all cases. Thanks to Amit Jain for inputs. The special cases that cause problems are like {2, 2, 2, 2, 2, 2, 2, 2, 0, 1, 1, 2} and {2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 2, 2}. It doesn't look possible to go to left half or right half by doing constant number of comparisons at the middle. Following is an implementation that handles duplicates. It may become O(n) in worst case though.

```
// C program to find minimum element in a sorted and rotated array
#include <stdio.h>

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

// The function that handles duplicates.  It can be O(n) in worst case.
int findMin(int arr[], int low, int high)
{
    // This condition is needed to handle the case when array is not
    // rotated at all
    if (high < low)  return arr[0];

    // If there is only one element left
    if (high == low) return arr[low];

    // Find mid
    int mid = low + (high - low)/2; /*(low + high)/2;*/

    // Check if element (mid+1) is minimum element. Consider
    // the cases like {1, 1, 0, 1}
    if (mid < high && arr[mid+1] < arr[mid])
       return arr[mid+1];

    // This case causes O(n) time
    if (arr[low] == arr[mid] && arr[high] == arr[mid])
        return min(findMin(arr, low, mid-1), findMin(arr, mid+1, high));

    // Check if mid itself is minimum element
    if (mid > low && arr[mid] < arr[mid - 1])
       return arr[mid];

    // Decide whether we need to go to left half or right half
    if (arr[high] > arr[mid])
       return findMin(arr, low, mid-1);
    return findMin(arr, mid+1, high);
}

// Driver program to test above functions
int main()
{
    int arr1[] =  {5, 6, 1, 2, 3, 4};
    int n1 = sizeof(arr1)/sizeof(arr1[0]);
```

```
    printf("The minimum element is %d\n", findMin(arr1, 0, n1-1));

    int arr2[] =  {1, 1, 0, 1};
    int n2 = sizeof(arr2)/sizeof(arr2[0]);
    printf("The minimum element is %d\n", findMin(arr2, 0, n2-1));

    int arr3[] =  {1, 1, 2, 2, 3};
    int n3 = sizeof(arr3)/sizeof(arr3[0]);
    printf("The minimum element is %d\n", findMin(arr3, 0, n3-1));

    int arr4[] =  {3, 3, 3, 4, 4, 4, 4, 5, 3, 3};
    int n4 = sizeof(arr4)/sizeof(arr4[0]);
    printf("The minimum element is %d\n", findMin(arr4, 0, n4-1));

    int arr5[] =  {2, 2, 2, 2, 2, 2, 2, 2, 0, 1, 1, 2};
    int n5 = sizeof(arr5)/sizeof(arr5[0]);
    printf("The minimum element is %d\n", findMin(arr5, 0, n5-1));

    int arr6[] =  {2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1};
    int n6 = sizeof(arr6)/sizeof(arr6[0]);
    printf("The minimum element is %d\n", findMin(arr6, 0, n6-1));

    int arr7[] =  {2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 2, 2};
    int n7 = sizeof(arr7)/sizeof(arr7[0]);
    printf("The minimum element is %d\n", findMin(arr7, 0, n7-1));

    return 0;
}
```

Output:

```
The minimum element is 1
The minimum element is 0
The minimum element is 1
The minimum element is 3
The minimum element is 0
The minimum element is 1
The minimum element is 0
```

Delete alternate nodes of a Linked List

May 20, 2010

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)

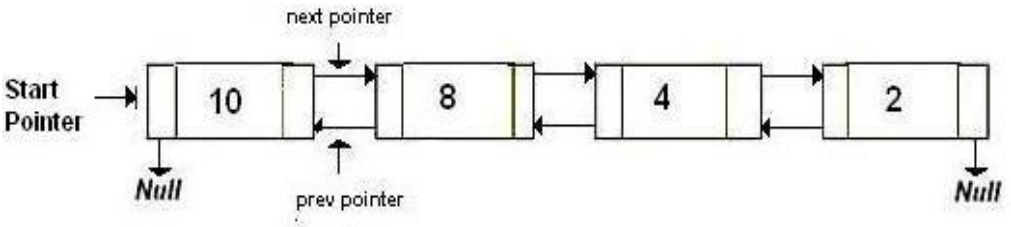
Reverse a Doubly Linked List

March 22, 2010

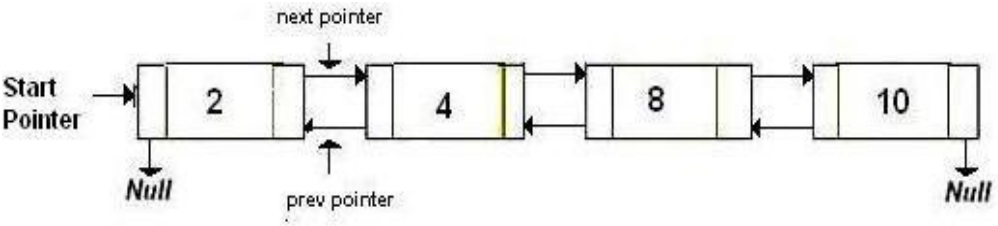
Write a C function to reverse a given Doubly Linked List

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)

Merge Sort for Linked Lists

June 6, 2010

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;

  /* 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);

  /* Sort the above created Linked List */
  MergeSort(&a);

  printf("\n Sorted Linked List is: \n");
  printList(a);

  getchar();
  return 0;
}
```

Time Complexity: O(nLogn)

Union and Intersection of two Linked Lists

April 1, 2012

Given two Linked Lists, create union and intersection lists that contain union and intersection of the elements present in the given lists. Order of elments in output lists doesn't matter.

Example:

```
Input:
   List1: 10->15->4->20
   lsit2:  8->4->2->10
```

```
Output:
   Intersection List: 4->10
   Union List: 2->8->20->4->15->10
```

Method 1 (Simple)
Following are simple algorithms to get union and intersection lists respectively.

Intersection (list1, list2)
Initialize result list as NULL. Traverse list1 and look for its each element in list2, if the element is present in list2, then add the element to result.

Union *(list1, list2):*
Initialize result list as NULL. Traverse list1 and add all of its elements to the result. Traverse list2. If an element of list2 is already present in result then do not insert it to result, otherwise insert.

This method assumes that there are no duplicates in the given lists.

Thanks to Shekhu for suggesting this method. Following is C implementation of this method.

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

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

/* A utility function to insert a node at the begining of a linked list*/
void push (struct node** head_ref, int new_data);

/* A utilty function to chec if given data is present in a list */
bool isPresent (struct node *head, int data);

/* Function to get union of two linked lists head1 and head2 */
struct node *getUnion (struct node *head1, struct node *head2)
{
    struct node *result = NULL;
    struct node *t1 = head1, *t2 = head2;

    // Insert all elements of list1 to the result list
    while (t1 != NULL)
    {
        push(&result, t1->data);
        t1 = t1->next;
    }

    // Insert those elements of list2 which are not present in result list
    while (t2 != NULL)
    {
        if (!isPresent(result, t2->data))
            push(&result, t2->data);
        t2 = t2->next;
    }
```

```
    return result;
}

/* Function to get intersection of two linked lists head1 and head2 */
struct node *getIntersection (struct node *head1, struct node *head2)
{
    struct node *result = NULL;
    struct node *t1 = head1;

    // Traverse list1 and search each element of it in list2. If the
element
    // is present in list 2, then insert the element to result
    while (t1 != NULL)
    {
        if (isPresent(head2, t1->data))
            push (&result, t1->data);
        t1 = t1->next;
    }

    return result;
}

/* A utility function to insert a node at the begining of a 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;
}

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

/* A utilty function that returns true if data is present in linked list
else return false */
bool isPresent (struct node *head, int data)
{
    struct node *t = head;
    while (t != NULL)
    {
        if (t->data == data)
            return 1;
```

```
        t = t->next;
    }
    return 0;
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head1 = NULL;
    struct node* head2 = NULL;
    struct node* intersecn = NULL;
    struct node* unin = NULL;

    /*create a linked lits 10->15->5->20 */
    push (&head1, 20);
    push (&head1, 4);
    push (&head1, 15);
    push (&head1, 10);

    /*create a linked lits 8->4->2->10 */
    push (&head2, 10);
    push (&head2, 2);
    push (&head2, 4);
    push (&head2, 8);

    intersecn = getIntersection (head1, head2);
    unin = getUnion (head1, head2);

    printf ("\n First list is \n");
    printList (head1);

    printf ("\n Second list is \n");
    printList (head2);

    printf ("\n Intersection list is \n");
    printList (intersecn);

    printf ("\n Union list is \n");
    printList (unin);

    return 0;
}
```

Output:

```
 First list is
10 15 4 20
 Second list is
8 4 2 10
 Intersection list is
4 10
 Union list is
2 8 20 4 15 10
```

Time Complexity: O(mn) for both union and intersection operations. Here m is the number of elements in first list and n is the number of elements in second list.

Method 2 (Use Merge Sort)
In this method, algorithms for Union and Intersection are very similar. First we sort the given lists, then we traverse the sorted lists to get union and intersection.
Following are the steps to be followed to get union and intersection lists.

1) Sort the first Linked List using merge sort. This step takes O(mLogm) time. Refer this post for details of this step.
2) Sort the second Linked List using merge sort. This step takes O(nLogn) time. Refer this post for details of this step.
3) Linearly scan both sorted lists to get the union and intersection. This step takes O(m + n) time. This step can be implemented using the same algorithm as sorted arrays algorithm discussed here.

Time complexity of this method is O(mLogm + nLogn) which is better than method 1′s time complexity.

Method 3 (Use Hashing)
Union (list1, list2)
Initialize the result list as NULL and create an empty hash table. Traverse both lists one by one, for each element being visited, look the element in hash table. If the element is not present, then insert the element to result list. If the element is present, then ignore it.

Intersection (list1, list2)
Initialize the result list as NULL and create an empty hash table. Traverse list1. For each element being visited in list1, insert the element in hash table. Traverse list2, for each element being visited in list2, look the element in hash table. If the element is present, then insert the element to result list. If the element is not present, then ignore it.

Both of the above methods assume that there are no duplicates.

Time complexity of this method depends on the hashing technique used and the distribution of elements in input lists. In practical, this approach may turn out to be better than above 2 methods.

In-place conversion of Sorted DLL to Balanced BST

February 3, 2012

Given a Doubly Linked List which has data members sorted in ascending order. Construct a Balanced Binary Search Tree which has same data members as the given Doubly Linked List. The tree must be constructed in-place (No new node should be allocated for tree conversion)

Examples:

```
Input:  Doubly Linked List 1 <--> 2 <--> 3
Output: A Balanced BST
```

```
     2
   /  \
  1    3
```

```
Input: Doubly Linked List 1 <--> 2 <-->3 <--> 4 <-->5 <--> 6 <--> 7
Output: A Balanced BST
        4
      /   \
     2     6
   /  \   / \
  1   3  4   7
```

```
Input: Doubly Linked List 1 <--> 2 <--> 3 <--> 4
Output: A Balanced BST
      3
    /  \
   2    4
 /
1
```

```
Input:  Doubly Linked List 1 <--> 2 <--> 3 <--> 4 <--> 5 <--> 6
Output: A Balanced BST
      4
    /   \
   2     6
 /  \   /
1   3  5
```

The Doubly Linked List conversion is very much similar to this Singly Linked List problem and the method 1 is exactly same as the method 1 of previous post. Method 2 is also almost same. The only difference in method 2 is, instead of allocating new nodes for BST, we reuse same DLL nodes. We use prev pointer as left and next pointer as right.

Method 1 (Simple)
Following is a simple algorithm where we first find the middle node of list and make it root of the tree to be constructed.

```
1) Get the Middle of the linked list and make it root.
2) Recursively do same for left half and right half.
       a) Get the middle of left half and make it left child of the root
          created in step 1.
       b) Get the middle of right half and make it right child of the
          root created in step 1.
```

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

Method 2 (Tricky)
The method 1 constructs the tree from root to leaves. In this method, we construct from leaves to root. The idea is to insert nodes in BST in the same order as the appear in Doubly Linked List, so that the tree can be constructed in O(n) time complexity. We first count the number of nodes in the given Linked List. Let the count be n. After counting nodes, we take left n/2 nodes and recursively construct the left subtree. After left subtree is constructed, we assign middle node to root and link the left subtree with root. Finally, we recursively construct the right subtree and link it with root.

While constructing the BST, we also keep moving the list head pointer to next so that we have the appropriate pointer in each recursive call.
Following is C implementation of method 2. The main code which creates Balanced BST is highlighted.

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

/* A Doubly Linked List node that will also be used as a tree node */
struct Node
{
    int data;

    // For tree, next pointer can be used as left subtree pointer
    struct Node* next;

    // For tree, prev pointer can be used as right subtree pointer
    struct Node* prev;
};


// A utility function to count nodes in a Linked List
int countNodes(struct Node *head);

struct Node* sortedListToBSTRecur(struct Node **head_ref, int n);

/* This function counts the number of nodes in Linked List and then calls
   sortedListToBSTRecur() to construct BST */
struct Node* sortedListToBST(struct Node *head)
{
    /*Count the number of nodes in Linked List */
    int n = countNodes(head);

    /* Construct BST */
    return sortedListToBSTRecur(&head, n);
}

/* The main function that constructs balanced BST and returns root of it.
       head_ref -->  Pointer to pointer to head node of Doubly linked list
       n  --> No. of nodes in the Doubly Linked List */
struct Node* sortedListToBSTRecur(struct Node **head_ref, int n)
{
    /* Base Case */
    if (n <= 0)
        return NULL;

    /* Recursively construct the left subtree */
    struct Node *left = sortedListToBSTRecur(head_ref, n/2);

    /* head_ref now refers to middle node, make middle node as root of
BST*/
    struct Node *root = *head_ref;

    // Set pointer to left subtree
    root->prev = left;

    /* Change head pointer of Linked List for parent recursive calls */
```

```
    *head_ref = (*head_ref)->next;

    /* Recursively construct the right subtree and link it with root
      The number of nodes in right subtree  is total nodes - nodes in
      left subtree - 1 (for root) */
    root->next = sortedListToBSTRecur(head_ref, n-n/2-1);

    return root;
}

/* UTILITY FUNCTIONS */
/* A utility function that returns count of nodes in a given Linked List */
int countNodes(struct Node *head)
{
    int count = 0;
    struct Node *temp = head;
    while(temp)
    {
        temp = temp->next;
        count++;
    }
    return count;
}

/* 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 linked list */
void printList(struct Node *node)
{
    while(node!=NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}
```

```
/* A utility function to print preorder traversal of BST */
void preOrder(struct Node* node)
{
    if (node == NULL)
        return;
    printf("%d ", node->data);
    preOrder(node->prev);
    preOrder(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 7->6->5->4->3->2->1 */
    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 ");
    printList(head);

    /* Convert List to BST */
    struct Node *root = sortedListToBST(head);
    printf("\n PreOrder Traversal of constructed BST ");
    preOrder(root);

    return 0;
}
```

Time Complexity: O(n)

Sorted Linked List to Balanced BST

January 17, 2012

Given a Singly Linked List which has data members sorted in ascending order. Construct a Balanced Binary Search Tree which has same data members as the given Linked List.

Examples:

```
Input:  Linked List 1->2->3
Output: A Balanced BST
     2
   /  \
  1    3
```

```
Input: Linked List 1->2->3->4->5->6->7
Output: A Balanced BST
         4
       /   \
      2     6
    /  \    / \
   1   3  4   7

Input: Linked List 1->2->3->4
Output: A Balanced BST
       3
     /  \
    2    4
  /
 1

Input:  Linked List 1->2->3->4->5->6
Output: A Balanced BST
       4
     /   \
    2     6
  /  \   /
 1   3  5
```

Method 1 (Simple)
Following is a simple algorithm where we first find the middle node of list and make it root of the tree to be constructed.

```
1) Get the Middle of the linked list and make it root.
2) Recursively do same for left half and right half.
       a) Get the middle of left half and make it left child of the root
          created in step 1.
       b) Get the middle of right half and make it right child of the
          root created in step 1.
```

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

See this forum thread for more details.

Method 2 (Tricky)
The method 1 constructs the tree from root to leaves. In this method, we construct from leaves to root. The idea is to insert nodes in BST in the same order as the appear in Linked List, so that the tree can be constructed in O(n) time complexity. We first count the number of nodes in the given Linked List. Let the count be n. After counting nodes, we take left n/2 nodes and recursively construct the left subtree. After left subtree is constructed, we allocate memory for root and link the left subtree with root. Finally, we recursively construct the right subtree and link it with root.
While constructing the BST, we also keep moving the list head pointer to next so that we have the appropriate pointer in each recursive call.
Following is C implementation of method 2. The main code which creates Balanced BST is highlighted.

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

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

/* A Binary Tree node */
struct TNode
{
    int data;
    struct TNode* left;
    struct TNode* right;
};

struct TNode* newNode(int data);
int countLNodes(struct LNode *head);
struct TNode* sortedListToBSTRecur(struct LNode **head_ref, int n);


/* This function counts the number of nodes in Linked List and then calls
   sortedListToBSTRecur() to construct BST */
struct TNode* sortedListToBST(struct LNode *head)
{
    /*Count the number of nodes in Linked List */
    int n = countLNodes(head);

    /* Construct BST */
    return sortedListToBSTRecur(&head, n);
}

/* The main function that constructs balanced BST and returns root of it.
       head_ref -->  Pointer to pointer to head node of linked list
       n  --> No. of nodes in Linked List */
struct TNode* sortedListToBSTRecur(struct LNode **head_ref, int n)
{
    /* Base Case */
    if (n <= 0)
        return NULL;

    /* Recursively construct the left subtree */
    struct TNode *left = sortedListToBSTRecur(head_ref, n/2);

    /* Allocate memory for root, and link the above constructed left
       subtree with root */
    struct TNode *root = newNode((*head_ref)->data);
    root->left = left;

    /* Change head pointer of Linked List for parent recursive calls */
    *head_ref = (*head_ref)->next;

    /* Recursively construct the right subtree and link it with root
      The number of nodes in right subtree  is total nodes - nodes in
      left subtree - 1 (for root) which is n-n/2-1*/
    root->right = sortedListToBSTRecur(head_ref, n-n/2-1);

    return root;
```

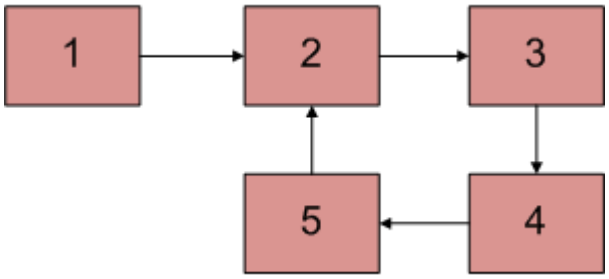
```
}



/* UTILITY FUNCTIONS */

/* A utility function that returns count of nodes in a given Linked List */
int countLNodes(struct LNode *head)
{
    int count = 0;
    struct LNode *temp = head;
    while(temp)
    {
        temp = temp->next;
        count++;
    }
    return count;
}

/* Function to insert a node at the beginging of the linked list */
void push(struct LNode** head_ref, int new_data)
{
    /* allocate node */
    struct LNode* new_node =
        (struct LNode*) malloc(sizeof(struct LNode));
    /* 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 LNode *node)
{
    while(node!=NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct TNode* newNode(int data)
{
    struct TNode* node = (struct TNode*)
                         malloc(sizeof(struct TNode));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return node;
}

/* A utility function to print preorder traversal of BST */
```

```
void preOrder(struct TNode* node)
{
    if (node == NULL)
        return;
    printf("%d ", node->data);
    preOrder(node->left);
    preOrder(node->right);
}

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

    /* Let us create a sorted linked list to test the functions
     Created linked list will be 1->2->3->4->5->6->7 */
    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 ");
    printList(head);

    /* Convert List to BST */
    struct TNode *root = sortedListToBST(head);
    printf("\n PreOrder Traversal of constructed BST ");
    preOrder(root);

    return 0;
}
```

Time Complexity: O(n)

Detect and Remove Loop in a Linked List

Write a function *detectAndRemoveLoop()* that checks whether a given Linked List contains loop and if loop is present then removes the loop and returns true. And if the list doesn't contain loop then returns false. Below diagram shows a linked list with a loop. *detectAndRemoveLoop()* must change the below list to 1->2->3->4->5->NULL.



We recommend to read following post as a prerequisite.

Write a C function to detect loop in a linked list

Before trying to remove the loop, we must detect it. Techniques discussed in the above post can be used to detect loop. To remove loop, all we need to do is to get pointer to the last node of the loop. For example, node with value 5 in the above diagram. Once we have pointer to the last node, we can make the next of this node as NULL and loop is gone.
We can easily use Hashing or Visited node techniques (discussed in the aobve mentioned post) to get the pointer to the last node. Idea is simple: the very first node whose next is already visited (or hashed) is the last node.
We can also use Floyd Cycle Detection algorithm to detect and remove the loop. In the Floyd's algo, the slow and fast pointers meet at a loop node. We can use this loop node to remove cycle. There are following two different ways of removing loop when Floyd's algorithm is used for Loop detection.

Method 1 (Check one by one)
We know that Floyd's Cycle detection algorithm terminates when fast and slow pointers meet at a common point. We also know that this common point is one of the loop nodes (2 or 3 or 4 or 5 in the above diagram). We store the address of this in a pointer variable say ptr2. Then we start from the head of the Linked List and check for nodes one by one if they are reachable from ptr2. When we find a node that is reachable, we know that this node is the starting node of the loop in Linked List and we can get pointer to the previous of this node.

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

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

/* Function to remove loop. Used by detectAndRemoveLoop() */
void removeLoop(struct node *, struct node *);

/* This function detects and removes loop in the list
  If loop was there in the list then it returns 1,
  otherwise returns 0 */
int detectAndRemoveLoop(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 and fast_p meet at some point then there
           is a loop */
        if (slow_p == fast_p)
        {
            removeLoop(slow_p, list);
```

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

    /* Return 0 to indeciate that ther is no loop*/
    return 0;
}

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

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

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

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

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

/* UTILITY FUNCTIONS */
/* Given a reference (pointer to pointer) to the head
  of a list and an int, pushes 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 *node)
{
    while(node != NULL)
    {
        printf("%d  ", node->data);
        node = node->next;
    }
}

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

    push(&head, 10);
    push(&head, 4);
    push(&head, 15);
    push(&head, 20);
    push(&head, 50);

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

    detectAndRemoveLoop(head);

    printf("Linked List after removing loop \n");
    printList(head);

    getchar();
    return 0;
}
```

Method 2 (Efficient Solution)
This method is also dependent on Floyd's Cycle detection algorithm.
1) Detect Loop using Floyd's Cycle detection algo and get the pointer to a loop node.
2) Count the number of nodes in loop. Let the count be k.
3) Fix one pointer to the head and another to kth node from head.
4) Move both pointers at the same pace, they will meet at loop starting node.
5) Get pointer to the last node of loop and make next of it as NULL.

Thanks to WgpShashank for suggesting this method.

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

/* Link list node */
struct node
{
```

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

/* Function to remove loop. */
void removeLoop(struct node *, struct node *);

/* This function detects and removes loop in the list
  If loop was there in the list then it returns 1,
  otherwise returns 0 */
int detectAndRemoveLoop(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 and fast_p meet at some point then there
           is a loop */
        if (slow_p == fast_p)
        {
            removeLoop(slow_p, list);

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

    /* Return 0 to indeciate that ther is no loop*/
    return 0;
}

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

    // Count the number of nodes in loop
    unsigned int k = 1, i;
    while (ptr1->next != ptr2)
    {
        ptr1 = ptr1->next;
        k++;
    }

    // Fix one pointer to head
    ptr1 = head;

    // And the other pointer to k nodes after head
    ptr2 = head;
    for(i = 0; i < k; i++)
      ptr2 = ptr2->next;

    /*  Move both pointers at the same pace,
```

```
      they will meet at loop starting node */
    while(ptr2 != ptr1)
    {
        ptr1 = ptr1->next;
        ptr2 = ptr2->next;
    }

    // Get pointer to the last node
    ptr2 = ptr2->next;
    while(ptr2->next != ptr1)
       ptr2 = ptr2->next;

    /* Set the next node of the loop ending node
      to fix the loop */
    ptr2->next = NULL;
}

/* UTILITY FUNCTIONS */
/* Given a reference (pointer to pointer) to the head
  of a list and an int, pushes 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 *node)
{
    while(node != NULL)
    {
        printf("%d  ", node->data);
        node = node->next;
    }
}

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

    push(&head, 10);
    push(&head, 4);
    push(&head, 15);
    push(&head, 20);
    push(&head, 50);
```

```
    /* Create a loop for testing */
    head->next->next->next->next->next = head->next->next;

    detectAndRemoveLoop(head);

    printf("Linked List after removing loop \n");
    printList(head);

    getchar();
    return 0;
}
```

Reverse alternate K nodes in a Singly Linked List

March 14, 2011

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)

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 a linked list into another linked list at alternate positions

Given two linked lists, insert nodes of second list into first list at alternate positions of first list.
For example, if first list is 5->7->17->13->11 and second is 12->10->2->4->6, the first list should become 5->12->7->10->17->2->13->4->11->6 and second list should become empty. The nodes of second list should only be inserted when there are positions available. For example, if the first list is 1->2->3 and second list is 4->5->6->7->8, then first list should become 1->4->2->5->3->6 and second list to 7->8.

Use of extra space is not allowed (Not allowed to create additional nodes), i.e., insertion must be done in-place. Expected time complexity is O(n) where n is number of nodes in first list.

The idea is to run a loop while there are available positions in first loop and insert nodes of second list by changing pointers. Following is C implementation of this approach.

```
// C implementation of above program.
#include <stdio.h>
#include <stdlib.h>

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

/* Function to insert a node at the beginning */
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 print a singly linked list */
void printList(struct node *head)
{
    struct node *temp = head;
    while (temp != NULL)
```

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

// Main function that inserts nodes of linked list q into p at alternate
// positions. Since head of first list never changes and head of second
list
// may change, we need single pointer for first list and double pointer for
// second list.
void merge(struct node *p, struct node **q)
{
     struct node *p_curr = p, *q_curr = *q;
     struct node *p_next, *q_next;

     // While therre are avialable positions in p
     while (p_curr != NULL && q_curr != NULL)
     {
         // Save next pointers
         p_next = p_curr->next;
         q_next = q_curr->next;

         // Make q_curr as next of p_curr
         q_curr->next = p_next;  // Change next pointer of q_curr
         p_curr->next = q_curr;  // Change next pointer of p_curr

         // Update current pointers for next iteration
         p_curr = p_next;
         q_curr = q_next;
    }

    *q = q_curr; // Update head pointer of second list
}

// Driver program to test above functions
int main()
{
     struct node *p = NULL, *q = NULL;
     push(&p, 3);
     push(&p, 2);
     push(&p, 1);
     printf("First Linked List:\n");
     printList(p);

     push(&q, 8);
     push(&q, 7);
     push(&q, 6);
     push(&q, 5);
     push(&q, 4);
     printf("Second Linked List:\n");
     printList(q);

     merge(p, &q);

     printf("Modified First Linked List:\n");
     printList(p);
```

```
    printf("Modified Second Linked List:\n");
    printList(q);

    getchar();
    return 0;
}
```

Output:

```
First Linked List:
1 2 3
Second Linked List:
4 5 6 7 8
Modified First Linked List:
1 4 2 5 3 6
Modified Second Linked List:
7 8
```

Delete N nodes after M nodes of a linked list

June 11, 2013

Given a linked list and two integers M and N. Traverse the linked list such that you retain M nodes then delete next N nodes, continue the same till end of the linked list.

Difficulty Level: Rookie

Examples:

```
Input:
M = 2, N = 2
Linked List: 1->2->3->4->5->6->7->8
Output:
Linked List: 1->2->5->6

Input:
M = 3, N = 2
Linked List: 1->2->3->4->5->6->7->8->9->10
Output:
Linked List: 1->2->3->6->7->8

Input:
M = 1, N = 1
Linked List: 1->2->3->4->5->6->7->8->9->10
Output:
Linked List: 1->3->5->7->9
```

The main part of the problem is to maintain proper links between nodes, make sure that all corner cases are handled. Following is C implementation of function skipMdeleteN() that skips M nodes and delete N nodes till end of list. It is assumed that M cannot be 0.

```
// C program to delete N nodes after M nodes of a linked list
#include <stdio.h>
```

```
#include <stdlib.h>

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

/* Function to insert a node at the beginning */
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;
    }
    printf("\n");
}

// Function to skip M nodes and then delete N nodes of the linked list.
void skipMdeleteN(struct node  *head, int M, int N)
{
    struct node *curr = head, *t;
    int count;

    // The main loop that traverses through the whole list
    while (curr)
    {
        // Skip M nodes
        for (count = 1; count<M && curr!= NULL; count++)
            curr = curr->next;

        // If we reached end of list, then return
        if (curr == NULL)
            return;

        // Start from next node and delete N nodes
        t = curr->next;
        for (count = 1; count<=N && t!= NULL; count++)
        {
            struct node *temp = t;
```

```
            t = t->next;
            free(temp);
        }
        curr->next = t; // Link the previous list with remaining nodes

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

// Driver program to test above functions
int main()
{
    /* Create following linked list
      1->2->3->4->5->6->7->8->9->10 */
    struct node* head = NULL;
    int M=2, N=3;
    push(&head, 10);
    push(&head, 9);
    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("M = %d, N = %d \nGiven Linked list is :\n", M, N);
    printList(head);

    skipMdeleteN(head, M, N);

    printf("\nLinked list after deletion is :\n");
    printList(head);

    return 0;
}
```

Output:

```
M = 2, N = 3
Given Linked list is :
1 2 3 4 5 6 7 8 9 10

Linked list after deletion is :
1 2 6 7
```

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

Sort a linked list of 0s, 1s and 2s

December 11, 2012

Given a linked list of 0s, 1s and 2s, sort it.

Source: Microsoft Interview | Set 1

Following steps can be used to sort the given linked list.
1) Traverse the list and count the number of 0s, 1s and 2s. Let the counts be n1, n2 and n3 respectively.
2) Traverse the list again, fill the first n1 nodes with 0, then n2 nodes with 1 and finally n3 nodes with 2.

```
// Program to sort a linked list 0s, 1s or 2s
#include<stdio.h>
#include<stdlib.h>

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

// Function to sort a linked list of 0s, 1s and 2s
void sortList(struct node *head)
{
    int count[3] = {0, 0, 0};  // Initialize count of '0', '1' and '2' as 0
    struct node *ptr = head;

    /* count total number of '0', '1' and '2'
     * count[0] will store total number of '0's
     * count[1] will store total number of '1's
     * count[2] will store total number of '2's  */
    while (ptr != NULL)
    {
        count[ptr->data] += 1;
        ptr = ptr->next;
    }

    int i = 0;
    ptr = head;

    /* Let say count[0] = n1, count[1] = n2 and count[2] = n3
     * now start traversing list from head node,
     * 1) fill the list with 0, till n1 > 0
     * 2) fill the list with 1, till n2 > 0
     * 3) fill the list with 2, till n3 > 0  */
    while (ptr != NULL)
    {
        if (count[i] == 0)
            ++i;
        else
        {
            ptr->data = i;
            --count[i];
            ptr = ptr->next;
        }
    }
}
```

```
/* 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;
    }
    printf("\n");
}

/* Drier program to test above function*/
int main(void)
{
    struct node *head = NULL;
    push(&head, 0);
    push(&head, 1);
    push(&head, 0);
    push(&head, 2);
    push(&head, 1);
    push(&head, 1);
    push(&head, 2);
    push(&head, 1);
    push(&head, 2);

    printf("Linked List Before Sorting\n");
    printList(head);

    sortList(head);

    printf("Linked List After Sorting\n");
    printList(head);

    return 0;
}
```

Output:

```
Linked List Before Sorting
2  1  2  1  1  2  0  1  0
Linked List After Sorting
```

```
0  0  1  1  1  1  2  2  2
```

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

Implement LRU Cache

September 28, 2012

How to implement LRU caching scheme? What data structures should be used?

We are given total possible page numbers that can be referred. We are also given cache (or memory) size (Number of page frames that cache can hold at a time). The LRU caching scheme is to remove the least recently used frame when the cache is full and a new page is referenced which is not there in cache. Please see the Galvin book for more details (see the LRU page replacement slide here).

We use two data structures to implement an LRU Cache.

1. A Queue which is implemented using a doubly linked list. The maximum size of the queue will be equal to the total number of frames available (cache size).
The most recently used pages will be near front end and least recently pages will be near rear end.

2. A Hash with page number as key and address of the corresponding queue node as value.

When a page is referenced, the required page may be in the memory. If it is in the memory, we need to detach the node of the list and bring it to the front of the queue.
If the required page is not in the memory, we bring that in memory. In simple words, we add a new node to the front of the queue and update the corresponding node address in the hash.
If the queue is full, i.e. all the frames are full, we remove a node from the rear of queue, and add the new node to the front of queue.

Note: Initially no page is in the memory.

Below is C implementation:

```
// A C program to show implementation of LRU cache
#include <stdio.h>
#include <stdlib.h>

// A Queue Node (Queue is implemented using Doubly Linked List)
typedef struct QNode
{
    struct QNode *prev, *next;
    unsigned pageNumber;  // the page number stored in this QNode
} QNode;

// A Queue (A FIFO collection of Queue Nodes)
typedef struct Queue
```

```
{
    unsigned count;  // Number of filled frames
    unsigned numberOfFrames; // total number of frames
    QNode *front, *rear;
} Queue;

// A hash (Collection of pointers to Queue Nodes)
typedef struct Hash
{
    int capacity; // how many pages can be there
    QNode* *array; // an array of queue nodes
} Hash;

// A utility function to create a new Queue Node. The queue Node
// will store the given 'pageNumber'
QNode* newQNode( unsigned pageNumber )
{
    // Allocate memory and assign 'pageNumber'
    QNode* temp = (QNode *)malloc( sizeof( QNode ) );
    temp->pageNumber = pageNumber;

    // Initialize prev and next as NULL
    temp->prev = temp->next = NULL;

    return temp;
}

// A utility function to create an empty Queue.
// The queue can have at most 'numberOfFrames' nodes
Queue* createQueue( int numberOfFrames )
{
    Queue* queue = (Queue *)malloc( sizeof( Queue ) );

    // The queue is empty
    queue->count = 0;
    queue->front = queue->rear = NULL;

    // Number of frames that can be stored in memory
    queue->numberOfFrames = numberOfFrames;

    return queue;
}

// A utility function to create an empty Hash of given capacity
Hash* createHash( int capacity )
{
    // Allocate memory for hash
    Hash* hash = (Hash *) malloc( sizeof( Hash ) );
    hash->capacity = capacity;

    // Create an array of pointers for refering queue nodes
    hash->array = (QNode **) malloc( hash->capacity * sizeof( QNode* ) );

    // Initialize all hash entries as empty
    int i;
    for( i = 0; i < hash->capacity; ++i )
        hash->array[i] = NULL;
```

```
    return hash;
}

// A function to check if there is slot available in memory
int AreAllFramesFull( Queue* queue )
{
    return queue->count == queue->numberOfFrames;
}

// A utility function to check if queue is empty
int isQueueEmpty( Queue* queue )
{
    return queue->rear == NULL;
}

// A utility function to delete a frame from queue
void deQueue( Queue* queue )
{
    if( isQueueEmpty( queue ) )
        return;

    // If this is the only node in list, then change front
    if (queue->front == queue->rear)
        queue->front = NULL;

    // Change rear and remove the previous rear
    QNode* temp = queue->rear;
    queue->rear = queue->rear->prev;

    if (queue->rear)
        queue->rear->next = NULL;

    free( temp );

    // decrement the number of full frames by 1
    queue->count--;
}

// A function to add a page with given 'pageNumber' to both queue
// and hash
void Enqueue( Queue* queue, Hash* hash, unsigned pageNumber )
{
    // If all frames are full, remove the page at the rear
    if ( AreAllFramesFull ( queue ) )
    {
        // remove page from hash
        hash->array[ queue->rear->pageNumber ] = NULL;
        deQueue( queue );
    }

    // Create a new node with given page number,
    // And add the new node to the front of queue
    QNode* temp = newQNode( pageNumber );
    temp->next = queue->front;

    // If queue is empty, change both front and rear pointers
    if ( isQueueEmpty( queue ) )
        queue->rear = queue->front = temp;
```

```
    else // Else change the front
    {
        queue->front->prev = temp;
        queue->front = temp;
    }

    // Add page entry to hash also
    hash->array[ pageNumber ] = temp;

    // increment number of full frames
    queue->count++;
}

// This function is called when a page with given 'pageNumber' is
referenced
// from cache (or memory). There are two cases:
// 1. Frame is not there in memory, we bring it in memory and add to the
front
//    of queue
// 2. Frame is there in memory, we move the frame to front of queue
void ReferencePage( Queue* queue, Hash* hash, unsigned pageNumber )
{
    QNode* reqPage = hash->array[ pageNumber ];

    // the page is not in cache, bring it
    if ( reqPage == NULL )
        Enqueue( queue, hash, pageNumber );

    // page is there and not at front, change pointer
    else if (reqPage != queue->front)
    {
        // Unlink rquested page from its current location
        // in queue.
        reqPage->prev->next = reqPage->next;
        if (reqPage->next)
           reqPage->next->prev = reqPage->prev;

        // If the requested page is rear, then change rear
        // as this node will be moved to front
        if (reqPage == queue->rear)
        {
           queue->rear = reqPage->prev;
           queue->rear->next = NULL;
        }

        // Put the requested page before current front
        reqPage->next = queue->front;
        reqPage->prev = NULL;

        // Change prev of current front
        reqPage->next->prev = reqPage;

        // Change front to the requested page
        queue->front = reqPage;
    }
}

// Driver program to test above functions
int main()
```

```
{
    // Let cache can hold 4 pages
    Queue* q = createQueue( 4 );

    // Let 10 different pages can be requested (pages to be
    // referenced are numbered from 0 to 9
    Hash* hash = createHash( 10 );

    // Let us refer pages 1, 2, 3, 1, 4, 5
    ReferencePage( q, hash, 1);
    ReferencePage( q, hash, 2);
    ReferencePage( q, hash, 3);
    ReferencePage( q, hash, 1);
    ReferencePage( q, hash, 4);
    ReferencePage( q, hash, 5);

    // Let us print cache frames after the above referenced pages
    printf ("%d ", q->front->pageNumber);
    printf ("%d ", q->front->next->pageNumber);
    printf ("%d ", q->front->next->next->pageNumber);
    printf ("%d ", q->front->next->next->next->pageNumber);

    return 0;
}
```

Output:

```
5 4 1 3
```

Find duplicates in O(n) time and O(1) extra space

Given an array of n elements which contains elements from 0 to n-1, with any of these numbers appearing any number of times. Find these repeating numbers in O(n) and using only constant memory space.

For example, let n be 7 and array be {1, 2, 3, 1, 3, 6, 6}, the answer should be 1, 3 and 6.

This problem is an extended version of following problem.

Find the two repeating elements in a given array

Method 1 and Method 2 of the above link are not applicable as the question says O(n) time complexity and O(1) constant space. Also, Method 3 and Method 4 cannot be applied here because there can be more than 2 repeating elements in this problem. Method 5 can be

extended to work for this problem. Below is the solution that is similar to the Method 5.

Algorithm:

```
traverse the list for i= 0 to n-1 elements
{
  check for sign of A[abs(A[i])] ;
  if positive then
     make it negative by   A[abs(A[i])]=-
A[abs(A[i])];
  else  // i.e., A[abs(A[i])] is negative
     this   element (ith element of list) is a
repetition
}
```

Implementation:

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

void printRepeating(int arr[], int size)
{
  int i;
  printf("The repeating elements are: \n");
  for (i = 0; i < size; i++)
  {
    if (arr[abs(arr[i])] >= 0)
      arr[abs(arr[i])] = -arr[abs(arr[i])];
    else
      printf(" %d ", abs(arr[i]));
  }
}

int main()
{
  int arr[] = {1, 2, 3, 1, 3, 6, 6};
  int arr_size = sizeof(arr)/sizeof(arr[0]);
  printRepeating(arr, arr_size);
```

```
  getchar();
  return 0;
}
```

Note: The above program doesn't handle 0 case (If 0 is present in array). The program can be easily modified to handle that also. It is not handled to keep the code simple.

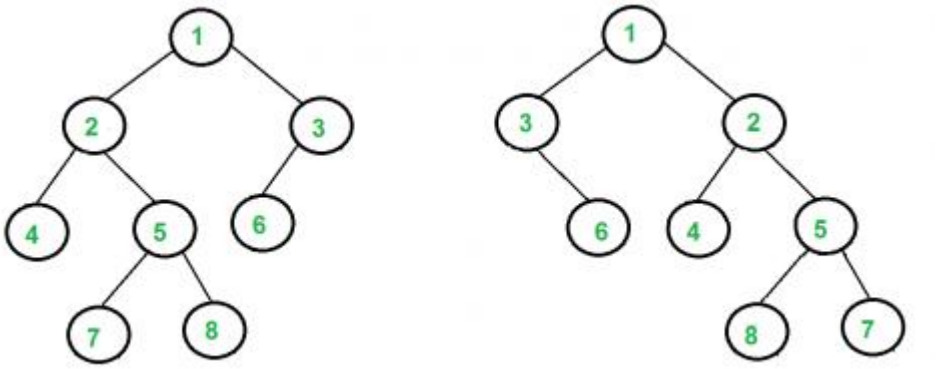
Output:
The repeating elements are:
1 3 6

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

Tree Isomorphism Problem

Write a function to detect if two trees are isomorphic. Two trees are called isomorphic if one of them can be obtained from other by a series of flips, i.e. by swapping left and right children of a number of nodes. Any number of nodes at any level can have their children swapped. Two empty trees are isomorphic.

For example, following two trees are isomorphic with following sub-trees flipped: 2 and 3, NULL and 6, 7 and 8.



We simultaneously traverse both trees. Let the current internal nodes of two trees being traversed be **n1** and **n2** respectively. There are following two conditions for subtrees rooted with n1 and n2 to be isomorphic.
1) Data of n1 and n2 is same.
2) One of the following two is true for children of n1 and n2
……**a)** Left child of n1 is isomorphic to left child of n2 and right child of n1 is isomorphic to right child of n2.

……**b)** Left child of n1 is isomorphic to right child of n2 and right child of n1 is isomorphic to left child of n2.

```
// A C++ program to check if two given trees are isomorphic
#include <iostream>
using namespace std;

/* A binary tree node has data, pointer to left and right children */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Given a binary tree, print its nodes in reverse level order */
bool isIsomorphic(node* n1, node *n2)
{
 // Both roots are NULL, trees isomorphic by definition
 if (n1 == NULL && n2 == NULL)
    return true;

 // Exactly one of the n1 and n2 is NULL, trees not isomorphic
 if (n1 == NULL || n2 == NULL)
    return false;

 if (n1->data != n2->data)
    return false;

 // There are two possible cases for n1 and n2 to be isomorphic
 // Case 1: The subtrees rooted at these nodes have NOT been "Flipped".
 // Both of these subtrees have to be isomorphic, hence the &&
 // Case 2: The subtrees rooted at these nodes have been "Flipped"
 return
 (isIsomorphic(n1->left,n2->left) && isIsomorphic(n1->right,n2->right))||
 (isIsomorphic(n1->left,n2->right) && isIsomorphic(n1->right,n2->left));
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
node* newNode(int data)
{
    node* temp = new node;
    temp->data = data;
    temp->left = NULL;
    temp->right = NULL;

    return (temp);
}

/* Driver program to test above functions*/
int main()
{
    // Let us create trees shown in above diagram
    struct node *n1 = newNode(1);
    n1->left        = newNode(2);
    n1->right       = newNode(3);
    n1->left->left  = newNode(4);
    n1->left->right = newNode(5);
```

```
    n1->right->left  = newNode(6);
    n1->left->right->left = newNode(7);
    n1->left->right->right = newNode(8);

    struct node *n2 = newNode(1);
    n2->left         = newNode(3);
    n2->right        = newNode(2);
    n2->right->left   = newNode(4);
    n2->right->right   = newNode(5);
    n2->left->right   = newNode(6);
    n2->right->right->left = newNode(8);
    n2->right->right->right = newNode(7);

    if (isIsomorphic(n1, n2) == true)
       cout << "Yes";
    else
      cout << "No";

    return 0;
}
```

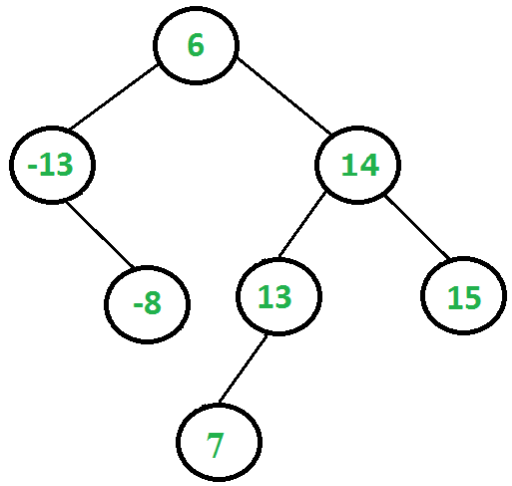
Output:

```
Yes
```

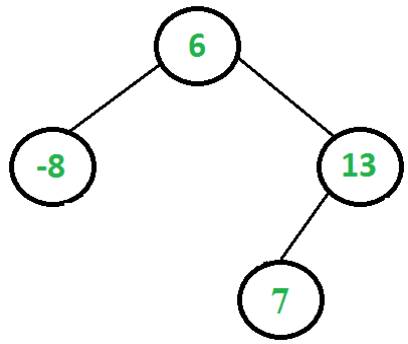
Time Complexity: The above solution does a traversal of both trees. So time complexity is O(m + n) where m and n are number of nodes in given trees.

Remove BST keys outside the given range

Given a Binary Search Tree (BST) and a range [min, max], remove all keys which are outside the given range. The modified tree should also be BST. For example, consider the following BST and range [-10, 13].



The given tree should be changed to following. Note that all keys outside the range [-10, 13] are removed and modified tree is BST.



There are two possible cases for every node.
1) Node's key is outside the given range. This case has two sub-cases.
…….**a)** Node's key is smaller than the min value.
…….**b)** Node's key is greater that the max value.
2) Node's key is in range.

We don't need to do anything for case 2. In case 1, we need to remove the node and change root of sub-tree rooted with this node.
The idea is to fix the tree in Postorder fashion. When we visit a node, we make sure that its left and right sub-trees are already fixed. In case 1.a), we simply remove root and return right sub-tree as new root. In case 1.b), we remove root and return left sub-tree as new root.

Following is C++ implementation of the above approach.

```
// A C++ program to remove BST keys outside the given range
#include<stdio.h>
#include <iostream>

using namespace std;

// A BST node has key, and left and right pointers
struct node
{
    int key;
    struct node *left;
    struct node *right;
};

// Resmoves all nodes having value outside the given range and returns the
root
// of modified tree
node* removeOutsideRange(node *root, int min, int max)
{
   // Base Case
   if (root == NULL)
      return NULL;

   // First fix the left and right subtrees of root
   root->left =  removeOutsideRange(root->left, min, max);
   root->right =  removeOutsideRange(root->right, min, max);

   // Now fix the root.  There are 2 possible cases for toot
```

```
    // 1.a) Root's key is smaller than min value (root is not in range)
    if (root->key < min)
    {
        node *rChild = root->right;
        delete root;
        return rChild;
    }
    // 1.b) Root's key is greater than max value (root is not in range)
    if (root->key > max)
    {
        node *lChild = root->left;
        delete root;
        return lChild;
    }
    // 2. Root is in range
    return root;
}

// A utility function to create a new BST node with key as given num
node* newNode(int num)
{
    node* temp = new node;
    temp->key = num;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to insert a given key to BST
node* insert(node* root, int key)
{
    if (root == NULL)
       return newNode(key);
    if (root->key > key)
       root->left = insert(root->left, key);
    else
       root->right = insert(root->right, key);
    return root;
}

// Utility function to traverse the binary tree after conversion
void inorderTraversal(node* root)
{
    if (root)
    {
        inorderTraversal( root->left );
        cout << root->key << " ";
        inorderTraversal( root->right );
    }
}

// Driver program to test above functions
int main()
{
    node* root = NULL;
    root = insert(root, 6);
    root = insert(root, -13);
    root = insert(root, 14);
    root = insert(root, -8);
    root = insert(root, 15);
    root = insert(root, 13);
```

```
    root = insert(root, 7);

    cout << "Inorder traversal of the given tree is: ";
    inorderTraversal(root);

    root = removeOutsideRange(root, -10, 13);

    cout << "\nInorder traversal of the modified tree is: ";
    inorderTraversal(root);

    return 0;
}
```

Output:

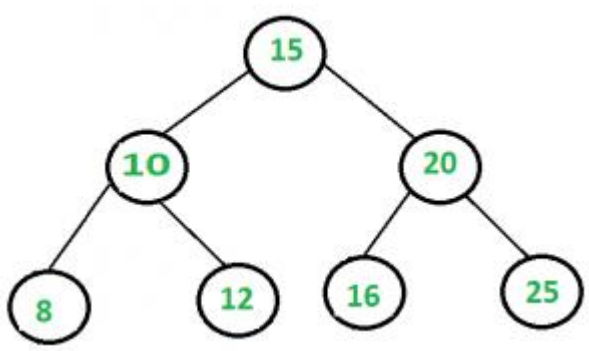
```
Inorder traversal of the given tree is: -13 -8 6 7 13 14 15
Inorder traversal of the modified tree is: -8 6 7 13
```

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

Find a pair with given sum in a Balanced BST

March 10, 2013

Given a Balanced Binary Search Tree and a target sum, write a function that returns true if there is a pair with sum equals to target sum, otherwise return false. Expected time complexity is O(n) and only O(Logn) extra space can be used. Any modification to Binary Search Tree is not allowed. Note that height of a Balanced BST is always O(Logn).



This problem is mainly extension of the previous post. Here we are not allowed to modify the BST.

The **Brute Force Solution** is to consider each pair in BST and check whether the sum equals to X. The time complexity of this solution will be O(n^2).

A **Better Solution** is to create an auxiliary array and store Inorder traversal of BST in the array. The array will be sorted as Inorder traversal of BST always produces sorted data. Once

we have the Inorder traversal, we can pair in O(n) time (See this for details). This solution works in O(n) time, but requires O(n) auxiliary space.

A **space optimized solution** is discussed in previous post. The idea was to first in-place convert BST to Doubly Linked List (DLL), then find pair in sorted DLL in O(n) time. This solution takes O(n) time and O(Logn) extra space, but it modifies the given BST.

The **solution discussed below takes O(n) time, O(Logn) space and doesn't modify BST**. The idea is same as finding the pair in sorted array (See method 1 of this for details). We traverse BST in Normal Inorder and Reverse Inorder simultaneously. In reverse inorder, we start from the rightmost node which is the maximum value node. In normal inorder, we start from the left most node which is minimum value node. We add sum of current nodes in both traversals and compare this sum with given target sum. If the sum is same as target sum, we return true. If the sum is more than target sum, we move to next node in reverse inorder traversal, otherwise we move to next node in normal inorder traversal. If any of the traversals is finished without finding a pair, we return false. Following is C++ implementation of this approach.

```
/* In a balanced binary search tree isPairPresent two element which sums to
   a given value time O(n) space O(logn) */
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 100

// A BST node
struct node
{
    int val;
    struct node *left, *right;
};

// Stack type
struct Stack
{
    int size;
    int top;
    struct node* *array;
};

// A utility function to create a stack of given size
struct Stack* createStack(int size)
{
    struct Stack* stack =
        (struct Stack*) malloc(sizeof(struct Stack));
    stack->size = size;
    stack->top = -1;
    stack->array =
        (struct node**) malloc(stack->size * sizeof(struct node*));
    return stack;
}

// BASIC OPERATIONS OF STACK
int isFull(struct Stack* stack)
{   return stack->top - 1 == stack->size;  }

int isEmpty(struct Stack* stack)
```

```
{    return stack->top == -1;    }

void push(struct Stack* stack, struct node* node)
{
    if (isFull(stack))
        return;
    stack->array[++stack->top] = node;
}

struct node* pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return NULL;
    return stack->array[stack->top--];
}

// Returns true if a pair with target sum exists in BST, otherwise false
bool isPairPresent(struct node *root, int target)
{
    // Create two stacks. s1 is used for normal inorder traversal
    // and s2 is used for reverse inorder traversal
    struct Stack* s1 = createStack(MAX_SIZE);
    struct Stack* s2 = createStack(MAX_SIZE);

    // Note the sizes of stacks is MAX_SIZE, we can find the tree size and
    // fix stack size as O(Logn) for balanced trees like AVL and Red Black
    // tree. We have used MAX_SIZE to keep the code simple

    // done1, val1 and curr1 are used for normal inorder traversal using s1
    // done2, val2 and curr2 are used for reverse inorder traversal using
s2
    bool done1 = false, done2 = false;
    int val1 = 0, val2 = 0;
    struct node *curr1 = root, *curr2 = root;

    // The loop will break when we either find a pair or one of the two
    // traversals is complete
    while (1)
    {
        // Find next node in normal Inorder traversal. See following post
        // http://www.geeksforgeeks.org/inorder-tree-traversal-without-
recursion/
        while (done1 == false)
        {
            if (curr1 != NULL)
            {
                push(s1, curr1);
                curr1 = curr1->left;
            }
            else
            {
                if (isEmpty(s1))
                    done1 = 1;
                else
                {
                    curr1 = pop(s1);
                    val1 = curr1->val;
                    curr1 = curr1->right;
                    done1 = 1;
                }
```

```
            }
        }

        // Find next node in REVERSE Inorder traversal. The only
        // difference between above and below loop is, in below loop
        // right subtree is traversed before left subtree
        while (done2 == false)
        {
            if (curr2 != NULL)
            {
                push(s2, curr2);
                curr2 = curr2->right;
            }
            else
            {
                if (isEmpty(s2))
                    done2 = 1;
                else
                {
                    curr2 = pop(s2);
                    val2 = curr2->val;
                    curr2 = curr2->left;
                    done2 = 1;
                }
            }
        }

        // If we find a pair, then print the pair and return. The first
        // condition makes sure that two same values are not added
        if ((val1 != val2) && (val1 + val2) == target)
        {
            printf("\n Pair Found: %d + %d = %d\n", val1, val2, target);
            return true;
        }

        // If sum of current values is smaller, then move to next node in
        // normal inorder traversal
        else if ((val1 + val2) < target)
            done1 = false;

        // If sum of current values is greater, then move to next node in
        // reverse inorder traversal
        else if ((val1 + val2) > target)
            done2 = false;

        // If any of the inorder traversals is over, then there is no pair
        // so return false
        if (val1 >= val2)
            return false;
    }
}

// A utility function to create BST node
struct node * NewNode(int val)
{
    struct node *tmp = (struct node *)malloc(sizeof(struct node));
    tmp->val = val;
    tmp->right = tmp->left =NULL;
    return tmp;
}
```

```
// Driver program to test above functions
int main()
{
    /*
                    15
                  /    \
                10      20
               / \     /  \
              8  12   16  25    */
    struct node *root =  NewNode(15);
    root->left = NewNode(10);
    root->right = NewNode(20);
    root->left->left = NewNode(8);
    root->left->right = NewNode(12);
    root->right->left = NewNode(16);
    root->right->right = NewNode(25);

    int target = 33;
    if (isPairPresent(root, target) == false)
        printf("\n No such values are found\n");

    getchar();
    return 0;
}
```

Output:

```
 Pair Found: 8 + 25 = 33
```

Iterative Postorder Traversal | Set 1 (Using Two Stacks)

February 23, 2013

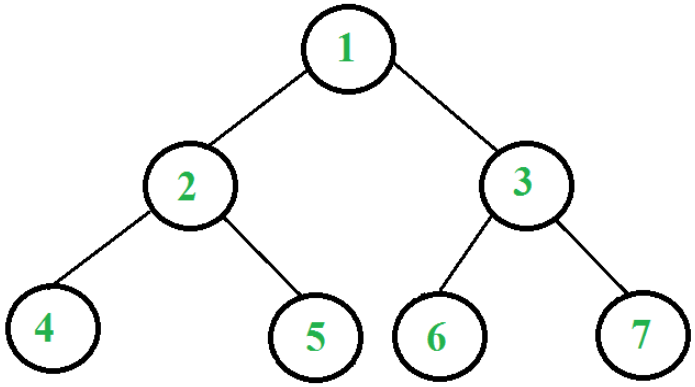
We have discussed iterative inorder and iterative preorder traversals. In this post, iterative postorder traversal is discussed which is more complex than the other two traversals (due to its nature of non-tail recursion, there is an extra statement after the final recursive call to itself). The postorder traversal can easily be done using two stacks though. The idea is to push reverse postorder traversal to a stack. Once we have reverse postorder traversal in a stack, we can just pop all items one by one from the stack and print them, this order of printing will be in postorder because of LIFO property of stacks. Now the question is, how to get reverse post order elements in a stack – the other stack is used for this purpose. For example, in the following tree, we need to get 1, 3, 7, 6, 2, 5, 4 in a stack. If take a closer look at this sequence, we can observe that this sequence is very similar to preorder traversal. The only difference is right child is visited before left child and therefore sequence is "root right left" instead of "root left right". So we can do something like iterative preorder traversal with following differences.
a) Instead of printing an item, we push it to a stack.
b) We push left subtree before right subtree.

Following is the complete algorithm. After step 2, we get reverse postorder traversal in second stack. We use first stack to get this order.

```
1. Push root to first stack.
```

```
2. Loop while first stack is not empty
   2.1 Pop a node from first stack and push it to second stack
   2.2 Push left and right children of the popped node to first stack
3. Print contents of second stack
```

Let us consider the following tree



Following are the steps to print postorder traversal of the above tree using two stacks.

```
1. Push 1 to first stack.
      First stack: 1
      Second stack: Empty

2. Pop 1 from first stack and push it to second stack.
   Push left and right children of 1 to first stack
      First stack: 2, 3
      Second stack: 1

3. Pop 3 from first stack and push it to second stack.
   Push left and right children of 3 to first stack
      First stack: 2, 6, 7
      Second stack:1, 3

4. Pop 7 from first stack and push it to second stack.
      First stack: 2, 6
      Second stack:1, 3, 7

5. Pop 6 from first stack and push it to second stack.
      First stack: 2
      Second stack:1, 3, 7, 6

6. Pop 2 from first stack and push it to second stack.
   Push left and right children of 2 to first stack
      First stack: 4, 5
      Second stack:1, 3, 7, 6, 2

7. Pop 5 from first stack and push it to second stack.
      First stack: 4
      Second stack: 1, 3, 7, 6, 2, 5

8. Pop 4 from first stack and push it to second stack.
      First stack: Empty
      Second stack: 1, 3, 7, 6, 2, 5, 4

The algorithm stops since there is no more item in first stack.
Observe that content of second stack is in postorder fashion. Print them.
```

Following is C implementation of iterative postorder traversal using two stacks.

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

// Maximum stack size
#define MAX_SIZE 100

// A tree node
struct Node
{
    int data;
    struct Node *left, *right;
};

// Stack type
struct Stack
{
    int size;
    int top;
    struct Node* *array;
};

// A utility function to create a new tree node
struct Node* newNode(int data)
{
    struct Node* node = (struct Node*) malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// A utility function to create a stack of given size
struct Stack* createStack(int size)
{
    struct Stack* stack =
            (struct Stack*) malloc(sizeof(struct Stack));
    stack->size = size;
    stack->top = -1;
    stack->array =
           (struct Node**) malloc(stack->size * sizeof(struct Node*));
    return stack;
}

// BASIC OPERATIONS OF STACK
int isFull(struct Stack* stack)
{  return stack->top - 1 == stack->size; }

int isEmpty(struct Stack* stack)
{  return stack->top == -1;  }

void push(struct Stack* stack, struct Node* node)
{
    if (isFull(stack))
        return;
    stack->array[++stack->top] = node;
}

struct Node* pop(struct Stack* stack)
```

```
{
    if (isEmpty(stack))
        return NULL;
    return stack->array[stack->top--];
}

// An iterative function to do post order traversal of a given binary tree
void postOrderIterative(struct Node* root)
{
    // Create two stacks
    struct Stack* s1 = createStack(MAX_SIZE);
    struct Stack* s2 = createStack(MAX_SIZE);

    // push root to first stack
    push(s1, root);
    struct Node* node;

    // Run while first stack is not empty
    while (!isEmpty(s1))
    {
        // Pop an item from s1 and push it to s2
        node = pop(s1);
        push(s2, node);

        // Push left and right children of removed item to s1
        if (node->left)
            push(s1, node->left);
        if (node->right)
            push(s1, node->right);
    }

    // Print all elements of second stack
    while (!isEmpty(s2))
    {
        node = pop(s2);
        printf("%d ", node->data);
    }
}

// Driver program to test above functions
int main()
{
    // Let us construct the tree shown in above figure
    struct Node* root = NULL;
    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->right->right = newNode(7);

    postOrderIterative(root);

    return 0;
}
```

Output:

```
4 5 2 6 7 3 1
```

Iterative Postorder Traversal | Set 2 (Using One Stack)

February 25, 2013

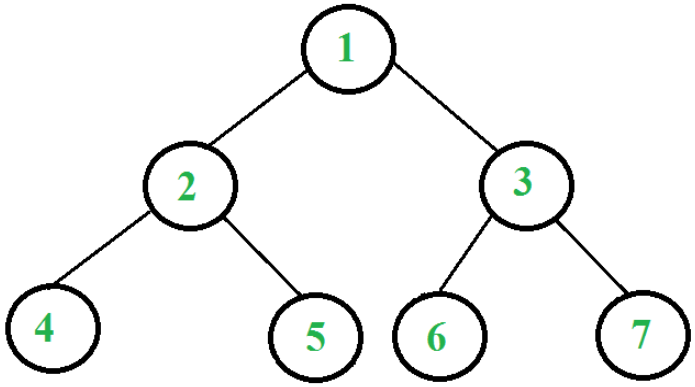
We have discussed a simple iterative postorder traversal using two stacks in the previous post. In this post, an approach with only one stack is discussed.

The idea is to move down to leftmost node using left pointer. While moving down, push root and root's right child to stack. Once we reach leftmost node, print it if it doesn't have a right child. If it has a right child, then change root so that the right child is processed before.

Following is detailed algorithm.

```
1.1 Create an empty stack
2.1 Do following while root is not NULL
    a) Push root's right child and then root to stack.
    b) Set root as root's left child.
2.2 Pop an item from stack and set it as root.
    a) If the popped item has a right child and the right child
       is at top of stack, then remove the right child from stack,
       push the root back and set root as root's right child.
    b) Else print root's data and set root as NULL.
2.3 Repeat steps 2.1 and 2.2 while stack is not empty.
```

Let us consider the following tree



Following are the steps to print postorder traversal of the above tree using one stack.

```
1. Right child of 1 exists.
   Push 3 to stack. Push 1 to stack. Move to left child.
        Stack: 3, 1

2. Right child of 2 exists.
   Push 5 to stack. Push 2 to stack. Move to left child.
        Stack: 3, 1, 5, 2

3. Right child of 4 doesn't exist. '
   Push 4 to stack. Move to left child.
```

```
        Stack: 3, 1, 5, 2, 4

4. Current node is NULL.
   Pop 4 from stack. Right child of 4 doesn't exist.
   Print 4. Set current node to NULL.
        Stack: 3, 1, 5, 2

5. Current node is NULL.
    Pop 2 from stack. Since right child of 2 equals stack top element,
    pop 5 from stack. Now push 2 to stack.
    Move current node to right child of 2 i.e. 5
        Stack: 3, 1, 2

6. Right child of 5 doesn't exist. Push 5 to stack. Move to left child.
        Stack: 3, 1, 2, 5

7. Current node is NULL. Pop 5 from stack. Right child of 5 doesn't exist.
   Print 5. Set current node to NULL.
        Stack: 3, 1, 2

8. Current node is NULL. Pop 2 from stack.
   Right child of 2 is not equal to stack top element.
   Print 2. Set current node to NULL.
        Stack: 3, 1

9. Current node is NULL. Pop 1 from stack.
   Since right child of 1 equals stack top element, pop 3 from stack.
   Now push 1 to stack. Move current node to right child of 1 i.e. 3
        Stack: 1

10. Repeat the same as above steps and Print 6, 7 and 3.
    Pop 1 and Print 1.
// C program for iterative postorder traversal using one stack
#include <stdio.h>
#include <stdlib.h>

// Maximum stack size
#define MAX_SIZE 100

// A tree node
struct Node
{
    int data;
    struct Node *left, *right;
};

// Stack type
struct Stack
{
    int size;
    int top;
    struct Node* *array;
};

// A utility function to create a new tree node
struct Node* newNode(int data)
{
    struct Node* node = (struct Node*) malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
```

```
    return node;
}

// A utility function to create a stack of given size
struct Stack* createStack(int size)
{
    struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));
    stack->size = size;
    stack->top = -1;
    stack->array = (struct Node**) malloc(stack->size * sizeof(struct
Node*));
    return stack;
}

// BASIC OPERATIONS OF STACK
int isFull(struct Stack* stack)
{  return stack->top - 1 == stack->size; }

int isEmpty(struct Stack* stack)
{  return stack->top == -1; }

void push(struct Stack* stack, struct Node* node)
{
    if (isFull(stack))
        return;
    stack->array[++stack->top] = node;
}

struct Node* pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return NULL;
    return stack->array[stack->top--];
}

struct Node* peek(struct Stack* stack)
{
    if (isEmpty(stack))
        return NULL;
    return stack->array[stack->top];
}

// An iterative function to do postorder traversal of a given binary tree
void postOrderIterative(struct Node* root)
{
    // Check for empty tree
    if (root == NULL)
        return;

    struct Stack* stack = createStack(MAX_SIZE);
    do
    {
        // Move to leftmost node
        while (root)
        {
            // Push root's right child and then root to stack.
            if (root->right)
                push(stack, root->right);
            push(stack, root);
```

```
            // Set root as root's left child
            root = root->left;
        }

        // Pop an item from stack and set it as root
        root = pop(stack);

        // If the popped item has a right child and the right child is not
        // processed yet, then make sure right child is processed before
root
        if (root->right && peek(stack) == root->right)
        {
            pop(stack);  // remove right child from stack
            push(stack, root);  // push root back to stack
            root = root->right; // change root so that the right
                                // child is processed next
        }
        else // Else print root's data and set root as NULL
        {
            printf("%d ", root->data);
            root = NULL;
        }
    } while (!isEmpty(stack));
}

// Driver program to test above functions
int main()
{
    // Let us construct the tree shown in above figure
    struct Node* root = NULL;
    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->right->right = newNode(7);

    postOrderIterative(root);

    return 0;
}
```

Output:

```
4 5 2 6 7 3 1
```

Floor and Ceil from a BST

October 17, 2012

There are numerous applications we need to find floor (ceil) value of a key in a binary search tree or sorted array. For example, consider designing memory management system in which free nodes are arranged in BST. Find best fit for the input request.

Ceil Value Node: Node with smallest data larger than or equal to key value.

Imagine we are moving down the tree, and assume we are root node. The comparison yields three possibilities,

A) Root data is equal to key. We are done, root data is ceil value.

B) Root data < key value, certainly the ceil value can't be in left subtree. Proceed to search on right subtree as reduced problem instance.

C) Root data > key value, the ceil value *may be* in left subtree. We may find a node with is larger data than key value in left subtree, if not the root itself will be ceil node.

Here is code in C for ceil value.

```
// Program to find ceil of a given value in BST
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has key, left child and right child */
struct node
{
    int key;
    struct node* left;
    struct node* right;
};

/* Helper function that allocates a new node with the given key and
   NULL left and right  pointers.*/
struct node* newNode(int key)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    return(node);
}

// Function to find ceil of a given input in BST. If input is more
// than the max key in BST, return -1
int Ceil(node *root, int input)
{
    // Base case
    if( root == NULL )
        return -1;

    // We found equal key
    if( root->key == input )
        return root->key;

    // If root's key is smaller, ceil must be in right subtree
    if( root->key < input )
```

```
        return Ceil(root->right, input);

    // Else, either left subtree or root has the ceil value
    int ceil = Ceil(root->left, input);
    return (ceil >= input) ? ceil : root->key;
}

// Driver program to test above function
int main()
{
    node *root = newNode(8);

    root->left = newNode(4);
    root->right = newNode(12);

    root->left->left = newNode(2);
    root->left->right = newNode(6);

    root->right->left = newNode(10);
    root->right->right = newNode(14);

    for(int i = 0; i < 16; i++)
        printf("%d  %d\n", i, Ceil(root, i));

    return 0;
}
```

Output:

```
0  2
1  2
2  2
3  4
4  4
5  6
6  6
7  8
8  8
9  10
10  10
11  12
12  12
13  14
14  14
15  -1
```

Two nodes of a BST are swapped, correct the BST

Two of the nodes of a Binary Search Tree (BST) are swapped. Fix (or correct) the BST.

```
Input Tree:
         10
        /  \
       5    8
```

```
      / \
     2   20

In the above tree, nodes 20 and 8 must be swapped to fix the tree.
Following is the output tree
         10
        /  \
       5    20
      / \
     2   8
```

The inorder traversal of a BST produces a sorted array. So a **simple method** is to store inorder traversal of the input tree in an auxiliary array. Sort the auxiliary array. Finally, insert the auxiilary array elements back to the BST, keeping the structure of the BST same. Time complexity of this method is O(nLogn) and auxiliary space needed is O(n).

We can solve this in O(n) time and with a single traversal of the given BST. Since inorder traversal of BST is always a sorted array, the problem can be reduced to a problem where two elements of a sorted array are swapped. There are two cases that we need to handle:

1. The swapped nodes are not adjacent in the inorder traversal of the BST.

```
For example, Nodes 5 and 25 are swapped in {3 5 7 8 10 15 20 25}.
The inorder traversal of the given tree is 3 25 7 8 10 15 20 5
```

If we observe carefully, during inorder traversal, we find node 7 is smaller than the previous visited node 25. Here save the context of node 25 (previous node). Again, we find that node 5 is smaller than the previous node 20. This time, we save the context of node 5 (current node). Finally swap the two node's values.

2. The swapped nodes are adjacent in the inorder traversal of BST.

```
For example, Nodes 7 and 8 are swapped in {3 5 7 8 10 15 20 25}.
The inorder traversal of the given tree is 3 5 8 7 10 15 20 25
```

Unlike case #1, here only one point exists where a node value is smaller than previous node value. e.g. node 7 is smaller than node 8.

How to Solve? *We will maintain three pointers, first, middle and last. When we find the first point where current node value is smaller than previous node value, we update the first with the previous node & middle with the current node. When we find the second point where current node value is smaller than previous node value, we update the last with the current node. In case #2, we will never find the second point. So, last pointer will not be updated. After processing, if the last node value is null, then two swapped nodes of BST are adjacent.*

Following is C implementation of the given code.

```
// Two nodes in the BST's swapped, correct the BST.
#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, *right;
};

// A utility function to swap two integers
void swap( int* a, int* b )
{
    int t = *a;
    *a = *b;
    *b = t;
}

/* 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 does inorder traversal to find out the two swapped nodes.
// It sets three pointers, first, middle and last.  If the swapped nodes
are
// adjacent to each other, then first and middle contain the resultant
nodes
// Else, first and last contain the resultant nodes
void correctBSTUtil( struct node* root, struct node** first,
                     struct node** middle, struct node** last,
                     struct node** prev )
{
    if( root )
    {
        // Recur for the left subtree
        correctBSTUtil( root->left, first, middle, last, prev );

        // If this node is smaller than the previous node, it's violating
        // the BST rule.
        if (*prev && root->data < (*prev)->data)
        {
            // If this is first violation, mark these two nodes as
            // 'first' and 'middle'
            if ( !*first )
            {
                *first = *prev;
                *middle = root;
            }

            // If this is second violation, mark this node as last
            else
                *last = root;
        }

        // Mark this node as previous
        *prev = root;
```

```
        // Recur for the right subtree
        correctBSTUtil( root->right, first, middle, last, prev );
    }
}

// A function to fix a given BST where two nodes are swapped.  This
// function uses correctBSTUtil() to find out two nodes and swaps the
// nodes to fix the BST
void correctBST( struct node* root )
{
    // Initialize pointers needed for correctBSTUtil()
    struct node *first, *middle, *last, *prev;
    first = middle = last = prev = NULL;

    // Set the poiters to find out two nodes
    correctBSTUtil( root, &first, &middle, &last, &prev );

    // Fix (or correct) the tree
    if( first && last )
        swap( &(first->data), &(last->data) );
    else if( first && middle ) // Adjacent nodes swapped
        swap( &(first->data), &(middle->data) );

    // else nodes have not been swapped, passed tree is really BST.
}

/* A utility function to print Inoder traversal */
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()
{
    /*   6
        /  \
       10    2
      / \   / \
     1   3 7  12
     10 and 2 are swapped
    */

    struct node *root = newNode(6);
    root->left        = newNode(10);
    root->right       = newNode(2);
    root->left->left  = newNode(1);
    root->left->right = newNode(3);
    root->right->right = newNode(12);
    root->right->left = newNode(7);

    printf("Inorder Traversal of the original tree \n");
    printInorder(root);

    correctBST(root);
```

```
    printf("\nInorder Traversal of the fixed tree \n");
    printInorder(root);

    return 0;
}
```

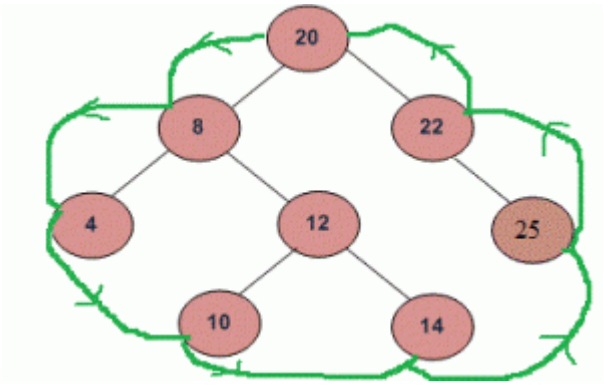
Output:

```
Inorder Traversal of the original tree
1 10 3 6 7 2 12
Inorder Traversal of the fixed tree
1 2 3 6 7 10 12
```

Time Complexity: O(n)

Boundary Traversal of binary tree

Given a binary tree, print boundary nodes of the binary tree Anti-Clockwise starting from the root. For example, boundary traversal of the following tree is “20 8 4 10 14 25 22″



We break the problem in 3 parts:
1. Print the left boundary in top-down manner.
2. Print all leaf nodes from left to right, which can again be sub-divided into two sub-parts:
…..**2.1** Print all leaf nodes of left sub-tree from left to right.
…..**2.2** Print all leaf nodes of right subtree from left to right.
3. Print the right boundary in bottom-up manner.

We need to take care of one thing that nodes are not printed again. e.g. The left most node is also the leaf node of the tree.

Based on the above cases, below is the implementation:

```
/* program for boundary traversal of a binary 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, *right;
};

// A simple function to print leaf nodes of a binary tree
void printLeaves(struct node* root)
{
    if ( root )
    {
        printLeaves(root->left);

        // Print it if it is a leaf node
        if ( !(root->left)  &&  !(root->right) )
            printf("%d ", root->data);

        printLeaves(root->right);
    }
}

// A function to print all left boundry nodes, except a leaf node.
// Print the nodes in TOP DOWN manner
void printBoundaryLeft(struct node* root)
{
    if (root)
    {
        if (root->left)
        {
            // to ensure top down order, print the node
            // before calling itself for left subtree
            printf("%d ", root->data);
            printBoundaryLeft(root->left);
        }
        else if( root->right )
        {
            printf("%d ", root->data);
            printBoundaryLeft(root->right);
        }
        // do nothing if it is a leaf node, this way we avoid
        // duplicates in output
    }
}

// A function to print all right boundry nodes, except a leaf node
// Print the nodes in BOTTOM UP manner
void printBoundaryRight(struct node* root)
{
    if (root)
    {
        if ( root->right )
        {
            // to ensure bottom up order, first call for right
            //  subtree, then print this node
            printBoundaryRight(root->right);
            printf("%d ", root->data);
        }
        else if ( root->left )
        {
            printBoundaryRight(root->left);
```

```
            printf("%d ", root->data);
        }
       // do nothing if it is a leaf node, this way we avoid
       // duplicates in output
    }
}


// A function to do boundary traversal of a given binary tree
void printBoundary (struct node* root)
{
    if (root)
    {
        printf("%d ",root->data);

        // Print the left boundary in top-down manner.
        printBoundaryLeft(root->left);

        // Print all leaf nodes
        printLeaves(root->left);
        printLeaves(root->right);

        // Print the right boundary in bottom-up manner
        printBoundaryRight(root->right);
    }
}

// A utility function to create a node
struct node* newNode( int data )
{
    struct node* temp = (struct node *) malloc( sizeof(struct node) );

    temp->data = data;
    temp->left = temp->right = NULL;

    return temp;
}

// Driver program to test above functions
int main()
{
    // Let us construct the tree given in the above diagram
    struct node *root         = newNode(20);
    root->left                 = newNode(8);
    root->left->left           = newNode(4);
    root->left->right          = newNode(12);
    root->left->right->left    = newNode(10);
    root->left->right->right   = newNode(14);
    root->right                = newNode(22);
    root->right->right         = newNode(25);

    printBoundary( root );

    return 0;
}
```

Output:

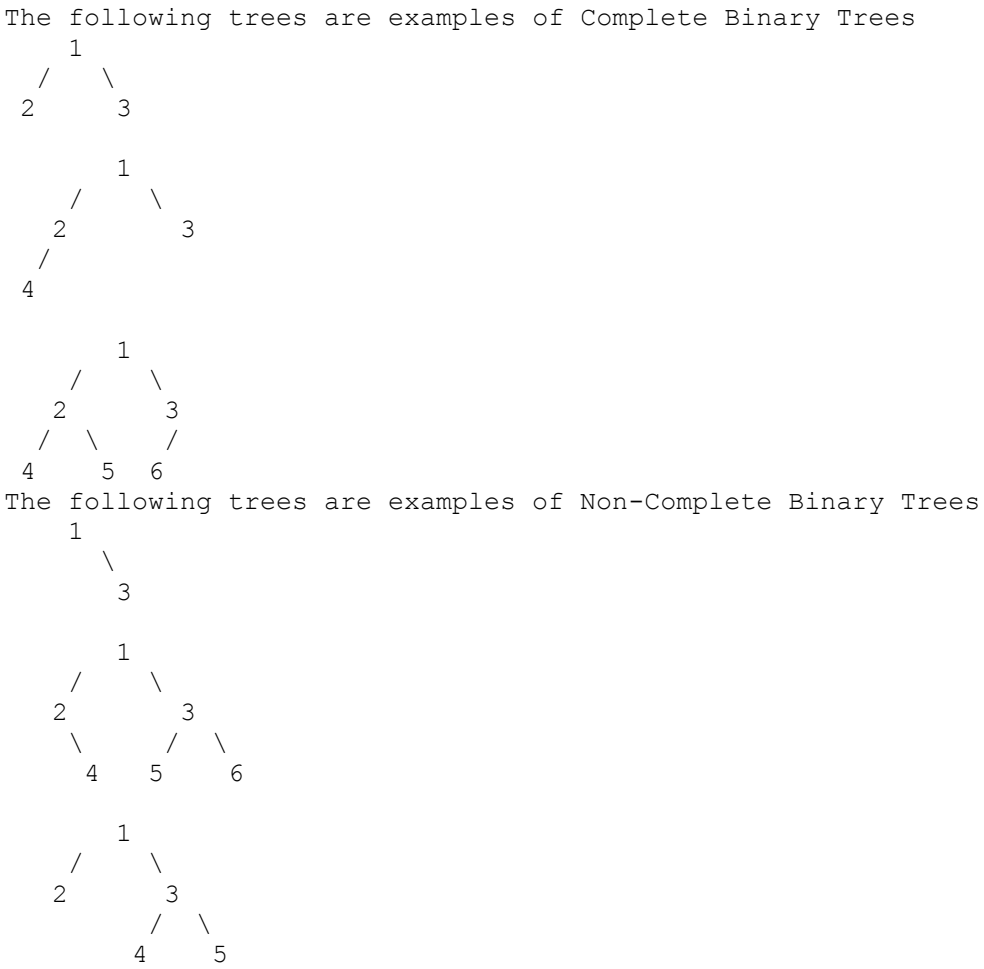
```
20 8 4 10 14 25 22
```

Time Complexity: O(n) where n is the number of nodes in binary tree.

Check whether a given Binary Tree is Complete or not

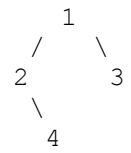
Given a Binary Tree, write a function to check whether the given Binary Tree is Complete Binary Tree or not.

A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible. See following examples.



Source: Write an algorithm to check if a tree is complete binary tree or not

The method 2 of level order traversal post can be easily modified to check whether a tree is Complete or not. To understand the approach, let us first define a term 'Full Node'. A node is 'Full Node' if both left and right children are not empty (or not NULL).
The approach is to do a level order traversal starting from root. In the traversal, once a node is found which is NOT a Full Node, all the following nodes must be leaf nodes.
Also, one more thing needs to be checked to handle the below case: If a node has empty left child, then the right child must be empty.



Thanks to Guddu Sharma for suggesting this simple and efficient approach.

```
// A program to check if a given binary tree is complete or not
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.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 for functions needed for Queue data
   structure. A queue is needed for level order tarversal */
struct node** createQueue(int *, int *);
void enQueue(struct node **, int *, struct node *);
struct node *deQueue(struct node **, int *);
bool isQueueEmpty(int *front, int *rear);

/* Given a binary tree, return true if the tree is complete
   else false */
bool isCompleteBT(struct node* root)
{
  // Base Case: An empty tree is complete Binary Tree
  if (root == NULL)
    return true;

  // Create an empty queue
  int rear, front;
  struct node **queue = createQueue(&front, &rear);

  // Create a flag variable which will be set true
  // when a non full node is seen
  bool flag = false;

  // Do level order traversal using queue.
  enQueue(queue, &rear, root);
  while(!isQueueEmpty(&front, &rear))
```

```
  {
    struct node *temp_node = deQueue(queue, &front);

    /* Ceck if left child is present*/
    if(temp_node->left)
    {
       // If we have seen a non full node, and we see a node
       // with non-empty left child, then the given tree is not
       // a complete Binary Tree
       if (flag == true)
         return false;

       enQueue(queue, &rear, temp_node->left);  // Enqueue Left Child
    }
    else // If this a non-full node, set the flag as true
       flag = true;

    /* Ceck if right child is present*/
    if(temp_node->right)
    {
       // If we have seen a non full node, and we see a node
       // with non-empty left child, then the given tree is not
       // a complete Binary Tree
       if(flag == true)
         return false;

       enQueue(queue, &rear, temp_node->right);  // Enqueue Right Child
    }
    else // If this a non-full node, set the flag as true
       flag = true;
  }

  // If we reach here, then the tree is complete Bianry Tree
  return true;
}



/*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];
}
```

```
bool isQueueEmpty(int *front, int *rear)
{
   return (*rear == *front);
}

/* 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()
{
   /* Let us construct the following Binary Tree which
      is not a complete Binary Tree
            1
          /   \
         2     3
        / \     \
       4   5     6
    */

  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(6);

  if ( isCompleteBT(root) == true )
      printf ("Complete Binary Tree");
  else
      printf ("NOT Complete Binary Tree");

  return 0;
}
```

Output:

```
NOT Complete Binary Tree
```

Time Complexity: O(n) where n is the number of nodes in given Binary Tree

Auxiliary Space: O(n) for queue.

Check if each internal node of a BST has exactly one child

August 5, 2012

Given Preorder traversal of a BST, check if each non-leaf node has only one child. Assume that the BST contains unique entries.

Examples

```
Input: pre[] = {20, 10, 11, 13, 12}
Output: Yes
The give array represents following BST. In the following BST, every
internal
node has exactly 1 child. Therefor, the output is true.
        20
       /
      10
       \
        11
          \
           13
           /
          12
```

In Preorder traversal, descendants (or Preorder successors) of every node appear after the node. In the above example, 20 is the first node in preorder and all descendants of 20 appear after it. All descendants of 20 are smaller than it. For 10, all descendants are greater than it. In general, we can say, if all internal nodes have only one child in a BST, then all the descendants of every node are either smaller or larger than the node. The reason is simple, since the tree is BST and every node has only one child, all descendants of a node will either be on left side or right side, means all descendants will either be smaller or greater.

Approach 1 (Naive)
This approach simply follows the above idea that all values on right side are either smaller or larger. Use two loops, the outer loop picks an element one by one, starting from the leftmost element. The inner loop checks if all elements on the right side of the picked element are either smaller or greater. The time complexity of this method will be O(n^2).

Approach 2
Since all the descendants of a node must either be larger or smaller than the node. We can do following for every node in a loop.
1. Find the next preorder successor (or descendant) of the node.
2. Find the last preorder successor (last element in pre[]) of the node.
3. If both successors are less than the current node, or both successors are greater than the current node, then continue. Else, return false.

```
#include <stdio.h>

bool hasOnlyOneChild(int pre[], int size)
{
    int nextDiff, lastDiff;

    for(int i=0; i<size-1; i++)
```

```
    {
        nextDiff = pre[i] - pre[i+1];
        lastDiff = pre[i] - pre[size-1];
        if(nextDiff*lastDiff < 0)
            return false;;
    }
    return true;
}

// driver program to test above function
int main()
{
    int pre[] = {8, 3, 5, 7, 6};
    int size = sizeof(pre)/sizeof(pre[0]);
    if (hasOnlyOneChild(pre, size) == true )
        printf("Yes");
    else
        printf("No");
    return 0;
}
```

Output:

```
 Yes
```

Approach 3

1. Scan the last two nodes of preorder & mark them as min & max.
2. Scan every node down the preorder array. Each node must be either smaller than the min node or larger than the max node. Update min & max accordingly.

```
#include <stdio.h>

int hasOnlyOneChild(int pre[], int size)
{
    // Initialize min and max using last two elements
    int min, max;
    if (pre[size-1] > pre[size-2])
    {
        max = pre[size-1];
        min = pre[size-2]):
    else
    {
        max = pre[size-2];
        min = pre[size-1];
    }


    // Every element must be either smaller than min or
    // greater than max
    for(int i=size-3; i>=0; i--)
    {
        if (pre[i] < min)
            min = pre[i];
        else if (pre[i] > max)
            max = pre[i];
        else
            return false;
    }
```

```
    return true;
}

// Driver program to test above function
int main()
{
    int pre[] = {8, 3, 5, 7, 6};
    int size = sizeof(pre)/sizeof(pre[0]);
    if (hasOnlyOneChild(pre,size))
        printf("Yes");
    else
        printf("No");
    return 0;
}
```

Output:

```
Yes
```

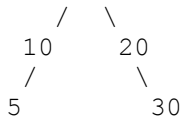
Binary Tree to Binary Search Tree Conversion

June 15, 2012

Given a Binary Tree, convert it to a Binary Search Tree. The conversion must be done in such a way that keeps the original structure of Binary Tree.

Examples.

```
Example 1
Input:
          10
         /  \
        2    7
       / \
      8   4
Output:
          8
         /  \
        4    10
       / \
      2   7


Example 2
Input:
          10
         /  \
        30   15
       /      \
      20       5
Output:
          15
```



Solution
Following is a 3 step solution for converting Binary tree to Binary Search Tree.
1) Create a temp array arr[] that stores inorder traversal of the tree. This step takes O(n) time.
2) Sort the temp array arr[]. Time complexity of this step depends upon the sorting algorithm. In the following implementation, Quick Sort is used which takes (n^2) time. This can be done in O(nLogn) time using Heap Sort or Merge Sort.
3) Again do inorder traversal of tree and copy array elements to tree nodes one by one. This step takes O(n) time.

Following is C implementation of the above approach. The main function to convert is highlighted in the following code.

```
/* A program to convert Binary Tree to Binary Search Tree */
#include<stdio.h>
#include<stdlib.h>

/* A binary tree node structure */
struct node
{
    int data;
    struct node *left;
    struct node *right;
};

/* A helper function that stores inorder traversal of a tree rooted
  with node */
void storeInorder (struct node* node, int inorder[], int *index_ptr)
{
    // Base Case
    if (node == NULL)
        return;

    /* first store the left subtree */
    storeInorder (node->left, inorder, index_ptr);

    /* Copy the root's data */
    inorder[*index_ptr] = node->data;
    (*index_ptr)++;  // increase index for next entry

    /* finally store the right subtree */
    storeInorder (node->right, inorder, index_ptr);
}

/* A helper function to count nodes in a Binary Tree */
int countNodes (struct node* root)
{
    if (root == NULL)
     return 0;
    return countNodes (root->left) +
           countNodes (root->right) + 1;
}
```

```
// Following function is needed for library function qsort()
int compare (const void * a, const void * b)
{
    return ( *(int*)a - *(int*)b );
}

/* A helper function that copies contents of arr[] to Binary Tree.
   This functon basically does Inorder traversal of Binary Tree and
   one by one copy arr[] elements to Binary Tree nodes */
void arrayToBST (int *arr, struct node* root, int *index_ptr)
{
    // Base Case
    if (root == NULL)
      return;

    /* first update the left subtree */
    arrayToBST (arr, root->left, index_ptr);

    /* Now update root's data and increment index */
    root->data = arr[*index_ptr];
    (*index_ptr)++;

    /* finally update the right subtree */
    arrayToBST (arr, root->right, index_ptr);
}

// This function converts a given Binary Tree to BST
void binaryTreeToBST (struct node *root)
{
    // base case: tree is empty
    if(root == NULL)
       return;

    /* Count the number of nodes in Binary Tree so that
       we know the size of temporary array to be created */
    int n = countNodes (root);

    // Create a temp array arr[] and store inorder traversal of tree in
arr[]
    int *arr = new int[n];
    int i = 0;
    storeInorder (root, arr, &i);

    // Sort the array using library function for quick sort
    qsort (arr, n, sizeof(arr[0]), compare);

    // Copy array elements back to Binary Tree
    i = 0;
    arrayToBST (arr, root, &i);

    // delete dynamically allocated memory to avoid meory leak
    delete [] arr;
}

/* 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;
}

/* Utility function to print inorder traversal of Binary Tree */
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);
}

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

    /* Constructing tree given in the above figure
          10
         /  \
        30   15
       /      \
      20       5   */
    root = newNode(10);
    root->left = newNode(30);
    root->right = newNode(15);
    root->left->left = newNode(20);
    root->right->right = newNode(5);

    // convert Binary Tree to BST
    binaryTreeToBST (root);

    printf("Following is Inorder Traversal of the converted BST: \n");
    printInorder (root);

    return 0;
}
```

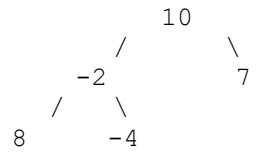
Output:

```
Following is Inorder Traversal of the converted BST:
5 10 15 20 30
```

Find the maximum sum leaf to root path in a Binary Tree

April 6, 2012

Given a Binary Tree, find the maximum sum path from a leaf to root. For example, in the following tree, there are three leaf to root paths 8->-2->10, -4->-2->10 and 7->10. The sums of these three paths are 16, 4 and 17 respectively. The maximum of them is 17 and the path for maximum is 7->10.



Solution
1) First find the leaf node that is on the maximum sum path. In the following code getTargetLeaf() does this by assigning the result to *target_leaf_ref.
2) Once we have the target leaf node, we can print the maximum sum path by traversing the tree. In the following code, printPath() does this.

The main function is maxSumPath() that uses above two functions to get the complete solution.

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

/* A tree node structure */
struct node
{
    int data;
    struct node *left;
    struct node *right;
};

// A utility function that prints all nodes on the path from root to
target_leaf
bool printPath (struct node *root, struct node *target_leaf)
{
    // base case
    if (root == NULL)
        return false;

    // return true if this node is the target_leaf or target leaf is
present in
    // one of its descendants
    if (root == target_leaf || printPath(root->left, target_leaf) ||
            printPath(root->right, target_leaf) )
    {
        printf("%d ", root->data);
        return true;
    }

    return false;
}

// This function Sets the target_leaf_ref to refer the leaf node of the
maximum
```

```
// path sum.  Also, returns the max_sum using max_sum_ref
void getTargetLeaf (struct node *node, int *max_sum_ref, int curr_sum,
                    struct node **target_leaf_ref)
{
    if (node == NULL)
        return;

    // Update current sum to hold sum of nodes on path from root to this
node
    curr_sum = curr_sum + node->data;

    // If this is a leaf node and path to this node has maximum sum so far,
    // then make this node target_leaf
    if (node->left == NULL && node->right == NULL)
    {
        if (curr_sum > *max_sum_ref)
        {
            *max_sum_ref = curr_sum;
            *target_leaf_ref = node;
        }
    }

    // If this is not a leaf node, then recur down to find the target_leaf
    getTargetLeaf (node->left, max_sum_ref, curr_sum, target_leaf_ref);
    getTargetLeaf (node->right, max_sum_ref, curr_sum, target_leaf_ref);
}

// Returns the maximum sum and prints the nodes on max sum path
int maxSumPath (struct node *node)
{
    // base case
    if (node == NULL)
        return 0;

    struct node *target_leaf;
    int max_sum = INT_MIN;

    // find the target leaf and maximum sum
    getTargetLeaf (node, &max_sum, 0, &target_leaf);

    // print the path from root to the target leaf
    printPath (node, target_leaf);

    return max_sum;  // return maximum sum
}

/* 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 = NULL;

    /* Constructing tree given in the above figure */
    root = newNode(10);
    root->left = newNode(-2);
    root->right = newNode(7);
    root->left->left = newNode(8);
    root->left->right = newNode(-4);

    printf ("Following are the nodes on the maximum sum path \n");
    int sum = maxSumPath(root);
    printf ("\nSum of the nodes is %d ", sum);

    getchar();
    return 0;
}
```

Output:

```
Following are the nodes on the maximum sum path
7 10
Sum of the nodes is 17
```

Vertical Sum in a given Binary Tree

February 29, 2012

Given a Binary Tree, find vertical sum of the nodes that are in same vertical line. Print all sums through different vertical lines.

Examples:

```
      1
    /   \
  2       3
 / \     / \
4   5   6   7
```

The tree has 5 vertical lines

Vertical-Line-1 has only one node 4 => vertical sum is 4
Vertical-Line-2: has only one node 2=> vertical sum is 2
Vertical-Line-3: has three nodes: 1,5,6 => vertical sum is 1+5+6 = 12
Vertical-Line-4: has only one node 3 => vertical sum is 3
Vertical-Line-5: has only one node 7 => vertical sum is 7

So expected output is 4, 2, 12, 3 and 7

Solution:
We need to check the Horizontal Distances from root for all nodes. If two nodes have the same Horizontal Distance (HD), then they are on same vertical line. The idea of HD is

simple. HD for root is 0, a right edge (edge connecting to right subtree) is considered as +1 horizontal distance and a left edge is considered as -1 horizontal distance. For example, in the above tree, HD for Node 4 is at -2, HD for Node 2 is -1, HD for 5 and 6 is 0 and HD for node 7 is +2.
We can do inorder traversal of the given Binary Tree. While traversing the tree, we can recursively calculate HDs. We initially pass the horizontal distance as 0 for root. For left subtree, we pass the Horizontal Distance as Horizontal distance of root minus 1. For right subtree, we pass the Horizontal Distance as Horizontal Distance of root plus 1.

Following is Java implementation for the same. HashMap is used to store the vertical sums for different horizontal distances. Thanks to Nages for suggesting this method.

```
import java.util.HashMap;

// Class for a tree node
class TreeNode {

    // data members
    private int key;
    private TreeNode left;
    private TreeNode right;

    // Accessor methods
    public int key()         { return key; }
    public TreeNode left()   { return left; }
    public TreeNode right()  { return right; }

    // Constructor
    public TreeNode(int key) { this.key = key; left = null; right = null; }

    // Methods to set left and right subtrees
    public void setLeft(TreeNode left)   { this.left = left; }
    public void setRight(TreeNode right) { this.right = right; }
}

// Class for a Binary Tree
class Tree {

    private TreeNode root;

    // Constructors
    public Tree() { root = null; }
    public Tree(TreeNode n) { root = n; }

    // Method to be called by the consumer classes like Main class
    public void VerticalSumMain() { VerticalSum(root); }

    // A wrapper over VerticalSumUtil()
    private void VerticalSum(TreeNode root) {

        // base case
        if (root == null) { return; }

        // Creates an empty hashMap hM
        HashMap<Integer, Integer> hM = new HashMap<Integer, Integer>();
```

```
        // Calls the VerticalSumUtil() to store the vertical sum values in
hM
        VerticalSumUtil(root, 0, hM);

        // Prints the values stored by VerticalSumUtil()
        if (hM != null) {
            System.out.println(hM.entrySet());
        }
    }

    // Traverses the tree in Inoorder form and builds a hashMap hM that
    // contains the vertical sum
    private void VerticalSumUtil(TreeNode root, int hD,
                                         HashMap<Integer, Integer> hM) {

        // base case
        if (root == null) {  return; }

        // Store the values in hM for left subtree
        VerticalSumUtil(root.left(), hD - 1, hM);

        // Update vertical sum for hD of this node
        int prevSum = (hM.get(hD) == null) ? 0 : hM.get(hD);
        hM.put(hD, prevSum + root.key());

        // Store the values in hM for right subtree
        VerticalSumUtil(root.right(), hD + 1, hM);
    }
}

// Driver class to test the verticalSum methods
public class Main {

    public static void main(String[] args) {
        /* Create following Binary Tree
              1
            /    \
          2        3
         / \      / \
        4   5    6   7

        */
        TreeNode root = new TreeNode(1);
        root.setLeft(new TreeNode(2));
        root.setRight(new TreeNode(3));
        root.left().setLeft(new TreeNode(4));
        root.left().setRight(new TreeNode(5));
        root.right().setLeft(new TreeNode(6));
        root.right().setRight(new TreeNode(7));
        Tree t = new Tree(root);

        System.out.println("Following are the values of vertical sums with
"
                + "the positions of the columns with respect to root ");
        t.VerticalSumMain();
    }
}
```

See this for a sample run.

Time Complexity: O(n)

Find the largest BST subtree in a given Binary Tree

February 17, 2012

Given a Binary Tree, write a function that returns the size of the largest subtree which is also a Binary Search Tree (BST). If the complete Binary Tree is BST, then return the size of whole tree.

Examples:

```
Input:
       5
     /  \
    2    4
  /  \
 1    3

Output: 3
The following subtree is the maximum size BST subtree
    2
  /  \
 1    3


Input:
        50
      /    \
   30       60
  /  \     /  \
 5   20   45    70
              /  \
            65    80
Output: 5
The following subtree is the maximum size BST subtree
       60
      /  \
    45    70
         /  \
       65    80
```

Method 1 (Simple but inefficient)
Start from root and do an inorder traversal of the tree. For each node N, check whether the subtree rooted with N is BST or not. If BST, then return size of the subtree rooted with N. Else, recur down the left and right subtrees and return the maximum of values returned by left and right subtrees.

```
/*
  See http://www.geeksforgeeks.org/archives/632 for implementation of
size()
```

```
  See Method 3 of http://www.geeksforgeeks.org/archives/3042 for
  implementation of isBST()

  max() returns maximum of two integers
*/
int largestBST(struct node *root)
{
   if (isBST(root))
     return size(root);
   else
    return max(largestBST(root->left), largestBST(root->right));
}
```

Time Complexity: The worst case time complexity of this method will be O(n^2). Consider a skewed tree for worst case analysis.

Method 2 (Tricky and Efficient)
In method 1, we traverse the tree in top down manner and do BST test for every node. If we traverse the tree in bottom up manner, then we can pass information about subtrees to the parent. The passed information can be used by the parent to do BST test (for parent node) only in constant time (or O(1) time). A left subtree need to tell the parent whether it is BST or not and also need to pass maximum value in it. So that we can compare the maximum value with the parent's data to check the BST property. Similarly, the right subtree need to pass the minimum value up the tree. The subtrees need to pass the following information up the tree for the finding the largest BST.
1) Whether the subtree itself is BST or not (In the following code, is_bst_ref is used for this purpose)
2) If the subtree is left subtree of its parent, then maximum value in it. And if it is right subtree then minimum value in it.
3) Size of this subtree if this subtree is BST (In the following code, return value of largestBSTtil() is used for this purpose)

max_ref is used for passing the maximum value up the tree and min_ptr is used for passing minimum value up the tree.

```
#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;
};

/* 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 largestBSTUtil(struct node* node, int *min_ref, int *max_ref,
                               int *max_size_ref, bool *is_bst_ref);

/* Returns size of the largest BST subtree in a Binary Tree
  (efficient version). */
int largestBST(struct node* node)
{
  // Set the initial values for calling largestBSTUtil()
  int min = INT_MAX;  // For minimum value in right subtree
  int max = INT_MIN;  // For maximum value in left subtree

  int max_size = 0;  // For size of the largest BST
  bool is_bst = 0;

  largestBSTUtil(node, &min, &max, &max_size, &is_bst);

  return max_size;
}

/* largestBSTUtil() updates *max_size_ref for the size of the largest BST
   subtree.   Also, if the tree rooted with node is non-empty and a BST,
   then returns size of the tree. Otherwise returns 0.*/
int largestBSTUtil(struct node* node, int *min_ref, int *max_ref,
                               int *max_size_ref, bool *is_bst_ref)
{

  /* Base Case */
  if (node == NULL)
  {
     *is_bst_ref = 1; // An empty tree is BST
     return 0;    // Size of the BST is 0
  }

  int min = INT_MAX;

  /* A flag variable for left subtree property
     i.e., max(root->left) < root->data */
  bool left_flag = false;

  /* A flag variable for right subtree property
     i.e., min(root->right) > root->data */
  bool right_flag = false;

  int ls, rs; // To store sizes of left and right subtrees

  /* Following tasks are done by recursive call for left subtree
    a) Get the maximum value in left subtree (Stored in *max_ref)
    b) Check whether Left Subtree is BST or not (Stored in *is_bst_ref)
    c) Get the size of maximum size BST in left subtree (updates *max_size)
*/
```

```
  *max_ref = INT_MIN;
  ls = largestBSTUtil(node->left, min_ref, max_ref, max_size_ref,
is_bst_ref);
  if (*is_bst_ref == 1 && node->data > *max_ref)
     left_flag = true;

  /* Before updating *min_ref, store the min value in left subtree. So that
we
     have the correct minimum value for this subtree */
  min = *min_ref;

  /* The following recursive call does similar (similar to left subtree)
    task for right subtree */
  *min_ref =  INT_MAX;
  rs = largestBSTUtil(node->right, min_ref, max_ref, max_size_ref,
is_bst_ref);
  if (*is_bst_ref == 1 && node->data < *min_ref)
     right_flag = true;

  // Update min and max values for the parent recursive calls
  if (min < *min_ref)
     *min_ref = min;
  if (node->data < *min_ref) // For leaf nodes
     *min_ref = node->data;
  if (node->data > *max_ref)
     *max_ref = node->data;

  /* If both left and right subtrees are BST. And left and right
     subtree properties hold for this node, then this tree is BST.
     So return the size of this tree */
  if(left_flag && right_flag)
  {
     if (ls + rs + 1 > *max_size_ref)
         *max_size_ref = ls + rs + 1;
     return ls + rs + 1;
  }
  else
  {
    //Since this subtree is not BST, set is_bst flag for parent calls
     *is_bst_ref = 0;
     return 0;
  }
}

/* Driver program to test above functions*/
int main()
{
    /* Let us construct the following Tree
          50
       /      \
     10        60
    /  \       /  \
   5   20    55    70
            /     /  \
          45     65    80
  */

  struct node *root = newNode(50);
  root->left        = newNode(10);
  root->right       = newNode(60);
```

```
  root->left->left  = newNode(5);
  root->left->right = newNode(20);
  root->right->left  = newNode(55);
  root->right->left->left  = newNode(45);
  root->right->right = newNode(70);
  root->right->right->left = newNode(65);
  root->right->right->right = newNode(80);

  /* The complete tree is not BST as 45 is in right subtree of 50.
     The following subtree is the largest BST
        60
      /  \
    55    70
   /     /  \
  5     65    80
  */
  printf(" Size of the largest BST is %d", largestBST(root));

  getchar();
  return 0;
}
```

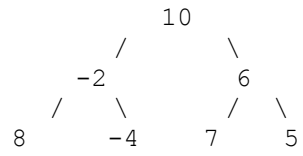
Time Complexity: O(n) where n is the number of nodes in the given Binary Tree.

Convert a given tree to its Sum Tree

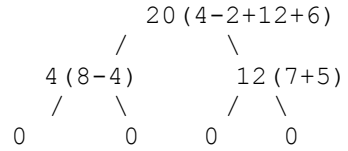
February 12, 2012

Given a Binary Tree where each node has positive and negative values. Convert this to a tree where each node contains the sum of the left and right sub trees in the original tree. The values of leaf nodes are changed to 0.

For example, the following tree



should be changed to



Solution:
Do a traversal of the given tree. In the traversal, store the old value of the current node, recursively call for left and right subtrees and change the value of current node as sum of the

values returned by the recursive calls. Finally return the sum of new value and value (which is sum of values in the subtree rooted with this node).

```
#include<stdio.h>

/* A tree node structure */
struct node
{
  int data;
  struct node *left;
  struct node *right;
};

// Convert a given tree to a tree where every node contains sum of values
of
// nodes in left and right subtrees in the original tree
int toSumTree(struct node *node)
{
    // Base case
    if(node == NULL)
      return 0;

    // Store the old value
    int old_val = node->data;

    // Recursively call for left and right subtrees and store the sum as
    // new value of this node
    node->data = toSumTree(node->left) + toSumTree(node->right);

    // Return the sum of values of nodes in left and right subtrees and
    // old_value of this node
    return node->data + old_val;
}

// A utility function to print inorder traversal of a Binary Tree
void printInorder(struct node* node)
{
     if (node == NULL)
          return;
     printInorder(node->left);
     printf("%d ", node->data);
     printInorder(node->right);
}

/* 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 = NULL;
  int x;

  /* Constructing tree given in the above figure */
  root = newNode(10);
  root->left = newNode(-2);
  root->right = newNode(6);
  root->left->left = newNode(8);
  root->left->right = newNode(-4);
  root->right->left = newNode(7);
  root->right->right = newNode(5);

  toSumTree(root);

  // Print inorder traversal of the converted tree to test result of
toSumTree()
  printf("Inorder Traversal of the resultant tree is: \n");
  printInorder(root);

  getchar();
  return 0;
}
```

Output:

```
Inorder Traversal of the resultant tree is:
0 4 0 20 0 12 0
```

Populate Inorder Successor for all nodes

January 26, 2012

Given a Binary Tree where each node has following structure, write a function to populate next pointer for all nodes. The next pointer for every node should be set to point to inorder successor.

```
struct node
{
  int data;
  struct node* left;
  struct node* right;
  struct node* next;
}
```

Initially, all next pointers have NULL values. Your function should fill these next pointers so that they point to inorder successor.

Solution (Use Reverse Inorder Traversal)
Traverse the given tree in reverse inorder traversal and keep track of previously visited node. When a node is being visited, assign previously visited node as next.

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

struct node
{
    int data;
    struct node *left;
    struct node *right;
    struct node *next;
};

/* Set next of p and all descendents of p by traversing them in reverse
Inorder */
void populateNext(struct node* p)
{
    // The first visited node will be the rightmost node
    // next of the rightmost node will be NULL
    static struct node *next = NULL;

    if (p)
    {
        // First set the next pointer in right subtree
        populateNext(p->right);

        // Set the next as previously visited node in reverse Inorder
        p->next = next;

        // Change the prev for subsequent node
        next = p;

        // Finally, set the next pointer in right subtree
        populateNext(p->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;
    node->next = NULL;

    return(node);
}

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

    /* Constructed binary tree is
              10
            /   \
          8      12
        /
```

```
      3
    */
    struct node *root = newnode(10);
    root->left        = newnode(8);
    root->right       = newnode(12);
    root->left->left  = newnode(3);

    // Populates nextRight pointer in all nodes
    populateNext(root);

    // Let us see the populated values
    struct node *ptr = root->left->left;
    while(ptr)
    {
        // -1 is printed if there is no successor
        printf("Next of %d is %d \n", ptr->data, ptr->next? ptr->next-
>data: -1);
        ptr = ptr->next;
    }

    return 0;
}
```

We can avoid the use of static variable by passing reference to next as paramater.

```
// An implementation that doesn't use static variable

// A wrapper over populateNextRecur
void populateNext(struct node *root)
{
    // The first visited node will be the rightmost node
    // next of the rightmost node will be NULL
    struct node *next = NULL;

    populateNextRecur(root, &next);
}

/* Set next of all descendents of p by traversing them in reverse Inorder
*/
void populateNextRecur(struct node* p, struct node **next_ref)
{
    if (p)
    {
        // First set the next pointer in right subtree
        populateNextRecur(p->right, next_ref);

        // Set the next as previously visited node in reverse Inorder
        p->next = *next_ref;

        // Change the prev for subsequent node
        *next_ref = p;

        // Finally, set the next pointer in right subtree
        populateNextRecur(p->left, next_ref);
    }
}
```

Time Complexity: O(n)

Connect nodes at same level using constant extra space

January 12, 2012

Write a function to connect all the adjacent nodes at the same level in a binary tree. Structure of the given Binary Tree node is like following.

```
struct node {
  int data;
  struct node* left;
  struct node* right;
  struct node* nextRight;
}
```

Initially, all the nextRight pointers point to garbage values. Your function should set these pointers to point next right for each node. You can use only constant extra space.

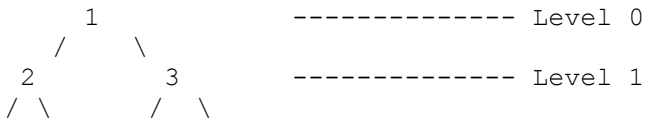
Example

```
Input Tree
       A
      / \
     B   C
    / \   \
   D   E   F

Output Tree
       A--->NULL
      / \
     B-->C-->NULL
    / \   \
   D-->E-->F-->NULL
```

We discussed two different approaches to do it in the previous post. The auxiliary space required in both of those approaches is not constant. Also, the method 2 discussed there only works for complete Binary Tree.

In this post, we will first modify the method 2 to make it work for all kind of trees. After that, we will remove recursion from this method so that the extra space becomes constant.

A Recursive Solution
In the method 2 of previous post, we traversed the nodes in pre order fashion. Instead of traversing in Pre Order fashion (root, left, right), if we traverse the nextRight node before the left and right children (root, nextRight, left), then we can make sure that all nodes at level i have the nextRight set, before the level i+1 nodes. Let us consider the following example (same example as previous post). The method 2 fails for right child of node 4. In this method, we make sure that all nodes at the 4′s level (level 2) have nextRight set, before we try to set the nextRight of 9. So when we set the nextRight of 9, we search for a nonleaf node on right side of node 4 (getNextRight() does this for us).



```
      4   5    6    7    -------------- Level 2
     / \           / \
    8   9        10   11 -------------- Level 3
void connectRecur(struct node* p);
struct node *getNextRight(struct node *p);

// Sets the nextRight of root and calls connectRecur() for other nodes
void connect (struct node *p)
{
    // Set the nextRight for root
    p->nextRight = NULL;

    // Set the next right for rest of the nodes (other than root)
    connectRecur(p);
}

/* Set next right of all descendents of p. This function makes sure that
nextRight of nodes ar level i is set before level i+1 nodes. */
void connectRecur(struct node* p)
{
    // Base case
    if (!p)
       return;

    /* Before setting nextRight of left and right children, set nextRight
    of children of other nodes at same level (because we can access
    children of other nodes using p's nextRight only) */
    if (p->nextRight != NULL)
       connectRecur(p->nextRight);

    /* Set the nextRight pointer for p's left child */
    if (p->left)
    {
       if (p->right)
       {
           p->left->nextRight = p->right;
           p->right->nextRight = getNextRight(p);
       }
       else
           p->left->nextRight = getNextRight(p);

       /* Recursively call for next level nodes.  Note that we call only
       for left child. The call for left child will call for right child */
       connectRecur(p->left);
    }

    /* If left child is NULL then first node of next level will either be
      p->right or getNextRight(p) */
    else if (p->right)
    {
        p->right->nextRight = getNextRight(p);
        connectRecur(p->right);
    }
    else
       connectRecur(getNextRight(p));
}

/* This function returns the leftmost child of nodes at the same level as
p.
```

```
   This function is used to getNExt right of p's right child
   If right child of p is NULL then this can also be used for the left
child */
struct node *getNextRight(struct node *p)
{
    struct node *temp = p->nextRight;

    /* Traverse nodes at p's level and find and return
       the first node's first child */
    while(temp != NULL)
    {
        if(temp->left != NULL)
            return temp->left;
        if(temp->right != NULL)
            return temp->right;
        temp = temp->nextRight;
    }

    // If all the nodes at p's level are leaf nodes then return NULL
    return NULL;
}
```

An Iterative Solution

The recursive approach discussed above can be easily converted to iterative. In the iterative version, we use nested loop. The outer loop, goes through all the levels and the inner loop goes through all the nodes at every level. This solution uses constant space.

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

struct node
{
    int data;
    struct node *left;
    struct node *right;
    struct node *nextRight;
};

/* This function returns the leftmost child of nodes at the same level as
p.
   This function is used to getNExt right of p's right child
   If right child of is NULL then this can also be sued for the left child
*/
struct node *getNextRight(struct node *p)
{
    struct node *temp = p->nextRight;

    /* Traverse nodes at p's level and find and return
       the first node's first child */
    while (temp != NULL)
    {
        if (temp->left != NULL)
            return temp->left;
        if (temp->right != NULL)
            return temp->right;
        temp = temp->nextRight;
```

```
    }

    // If all the nodes at p's level are leaf nodes then return NULL
    return NULL;
}

/* Sets nextRight of all nodes of a tree with root as p */
void connect(struct node* p)
{
    struct node *temp;

    if (!p)
      return;

    // Set nextRight for root
    p->nextRight = NULL;

    // set nextRight of all levels one by one
    while (p != NULL)
    {
        struct node *q = p;

        /* Connect all childrem nodes of p and children nodes of all other
nodes
          at same level as p */
        while (q != NULL)
        {
            // Set the nextRight pointer for p's left child
            if (q->left)
            {
                // If q has right child, then right child is nextRight of
                // p and we also need to set nextRight of right child
                if (q->right)
                    q->left->nextRight = q->right;
                else
                    q->left->nextRight = getNextRight(q);
            }

            if (q->right)
                q->right->nextRight = getNextRight(q);

            // Set nextRight for other nodes in pre order fashion
            q = q->nextRight;
        }

        // start from the first node of next level
        if (p->left)
           p = p->left;
        else if (p->right)
           p = p->right;
        else
           p = getNextRight(p);
    }
}

/* 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;
    node->nextRight = NULL;

    return(node);
}

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

    /* Constructed binary tree is
              10
            /   \
          8      2
        /          \
      3             90
    */
    struct node *root = newnode(10);
    root->left        = newnode(8);
    root->right       = newnode(2);
    root->left->left  = newnode(3);
    root->right->right       = newnode(90);

    // Populates nextRight pointer in all nodes
    connect(root);

    // Let us check the values of nextRight pointers
    printf("Following are populated nextRight pointers in the tree "
           "(-1 is printed if there is no nextRight) \n");
    printf("nextRight of %d is %d \n", root->data,
           root->nextRight? root->nextRight->data: -1);
    printf("nextRight of %d is %d \n", root->left->data,
           root->left->nextRight? root->left->nextRight->data: -1);
    printf("nextRight of %d is %d \n", root->right->data,
           root->right->nextRight? root->right->nextRight->data: -1);
    printf("nextRight of %d is %d \n", root->left->left->data,
           root->left->left->nextRight? root->left->left->nextRight->data:
-1);
    printf("nextRight of %d is %d \n", root->right->right->data,
           root->right->right->nextRight? root->right->right->nextRight-
>data: -1);

    getchar();
    return 0;
}
```

Output:

```
Following are populated nextRight pointers in the tree (-1 is printed if
there is no nextRight)
nextRight of 10 is -1
nextRight of 8 is 2
nextRight of 2 is -1
nextRight of 3 is 90
```

```
nextRight of 90 is -1
```

Connect nodes at same level

January 11, 2012

Write a function to connect all the adjacent nodes at the same level in a binary tree. Structure of the given Binary Tree node is like following.

```
struct node{
  int data;
  struct node* left;
  struct node* right;
  struct node* nextRight;
}
```

Initially, all the nextRight pointers point to garbage values. Your function should set these pointers to point next right for each node.

Example

```
Input Tree
       A
      / \
     B   C
    / \   \
   D   E   F


Output Tree
       A--->NULL
      / \
     B-->C-->NULL
    / \   \
   D-->E-->F-->NULL
```

Method 1 (Extend Level Order Traversal or BFS)
Consider the method 2 of Level Order Traversal. The method 2 can easily be extended to connect nodes of same level. We can augment queue entries to contain level of nodes also which is 0 for root, 1 for root's children and so on. So a queue node will now contain a pointer to a tree node and an integer level. When we enqueue a node, we make sure that correct level value for node is being set in queue. To set nextRight, for every node N, we dequeue the next node from queue, if the level number of next node is same, we set the nextRight of N as address of the dequeued node, otherwise we set nextRight of N as NULL.

Time Complexity: O(n)

Method 2 (Extend Pre Order Traversal)
This approach works only for Complete Binary Trees. In this method we set nextRight in Pre Order fashion to make sure that the nextRight of parent is set before its children. When we are at node p, we set the nextRight of its left and right children. Since the tree is complete tree, nextRight of p's left child (p->left->nextRight) will always be p's right child, and nextRight of p's right child (p->right->nextRight) will always be left child of p's nextRight (if p is not the rightmost node at its level). If p is the rightmost node, then nextRight of p's right child will be NULL.

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

struct node
{
  int data;
  struct node *left;
  struct node *right;
  struct node *nextRight;
};

void connectRecur(struct node* p);

// Sets the nextRight of root and calls connectRecur() for other nodes
void connect (struct node *p)
{
    // Set the nextRight for root
    p->nextRight = NULL;

    // Set the next right for rest of the nodes (other than root)
    connectRecur(p);
}

/* Set next right of all descendents of p.
   Assumption:  p is a compete binary tree */
void connectRecur(struct node* p)
{
  // Base case
  if (!p)
    return;

  // Set the nextRight pointer for p's left child
  if (p->left)
    p->left->nextRight = p->right;

  // Set the nextRight pointer for p's right child
  // p->nextRight will be NULL if p is the right most child at its level
  if (p->right)
    p->right->nextRight = (p->nextRight)? p->nextRight->left: NULL;

  // Set nextRight for other nodes in pre order fashion
  connectRecur(p->left);
  connectRecur(p->right);
}

/* UTILITY FUNCTIONS */
/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
```

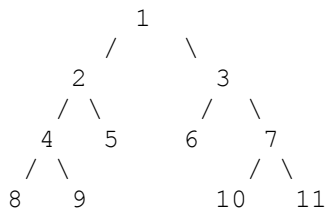
```
struct node* newnode(int data)
{
  struct node* node = (struct node*)
                       malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;
  node->nextRight = NULL;

  return(node);
}

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

  /* Constructed binary tree is
            10
          /   \
        8      2
      /
    3
  */
  struct node *root = newnode(10);
  root->left        = newnode(8);
  root->right       = newnode(2);
  root->left->left  = newnode(3);

  // Populates nextRight pointer in all nodes
  connect(root);

  // Let us check the values of nextRight pointers
  printf("Following are populated nextRight pointers in the tree "
         "(-1 is printed if there is no nextRight) \n");
  printf("nextRight of %d is %d \n", root->data,
         root->nextRight? root->nextRight->data: -1);
  printf("nextRight of %d is %d \n", root->left->data,
        root->left->nextRight? root->left->nextRight->data: -1);
  printf("nextRight of %d is %d \n", root->right->data,
        root->right->nextRight? root->right->nextRight->data: -1);
  printf("nextRight of %d is %d \n", root->left->left->data,
        root->left->left->nextRight? root->left->left->nextRight->data: -
1);

  getchar();
  return 0;
}
```

Thanks to Dhanya for suggesting this approach.

Time Complexity: O(n)

Why doesn't method 2 work for trees which are not Complete Binary Trees?
Let us consider following tree as an example. In Method 2, we set the nextRight pointer in pre order fashion. When we are at node 4, we set the nextRight of its children which are 8 and 9 (the nextRight of 4 is already set as node 5). nextRight of 8 will simply be set as 9, but nextRight of 9 will be set as NULL which is incorrect. We can't set the correct nextRight,

because when we set nextRight of 9, we only have nextRight of node 4 and ancestors of node 4, we don’t have nextRight of nodes in right subtree of root.

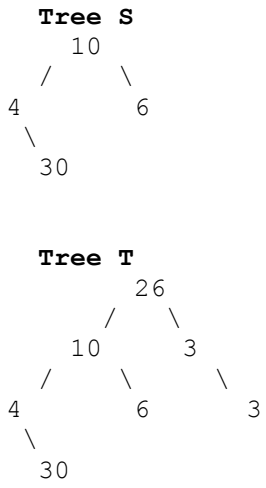


Check if a binary tree is subtree of another binary tree

August 15, 2011

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 preorder 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;
}
```

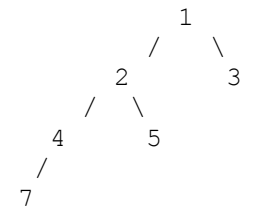
Output:
Tree S is subtree of tree T

Print Ancestors of a given node in Binary Tree

March 1, 2011

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;
}
```

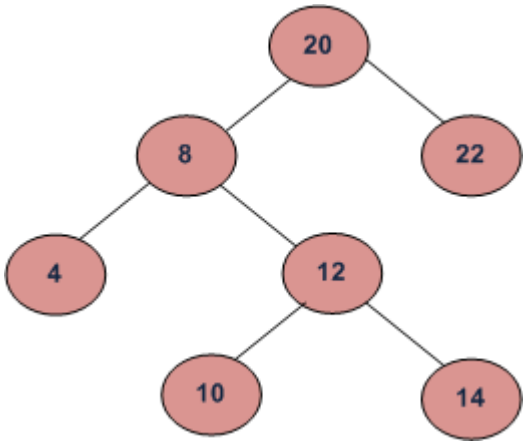
Output:
4 2 1

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

Inorder Successor in Binary Search Tree

February 11, 2011

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**.

Method 1 (Uses Parent Pointer)
In this method, we assume that every node has parent pointer.

The 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);
  else
    printf("\n Inorder Successor doesn't exit");

  getchar();
  return 0;
}
```

Output of the above program:
Inorder Successor of 14 is 20

Time Complexity: O(h) where h is height of tree.

Method 2 (Search from root)
Parent pointer is NOT needed in this algorithm. The 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 start from root and us search like technique. Do following.

Travel down the tree, if a node's data is greater than root's data then go right side, otherwise go to left side.

```
struct node * inOrderSuccessor(struct node *root, struct node *n)
{
    // step 1 of the above algorithm
    if( n->right != NULL )
        return minValue(n->right);

    struct node *succ = NULL;

    // Start from root and search for successor down the tree
    while (root != NULL)
    {
        if (n->data < root->data)
        {
            succ = root;
            root = root->left;
        }
        else if (n->data > root->data)
            root = root->right;
        else
           break;
    }

    return succ;
}
```

Thanks to R.Srinivasan for suggesting this method.

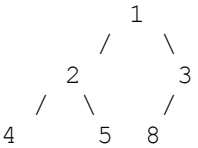
Time Complexity: O(h) where h is height of tree.

Print nodes at k distance from root

September 16, 2010

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.

Total number of possible Binary Search Trees with n keys

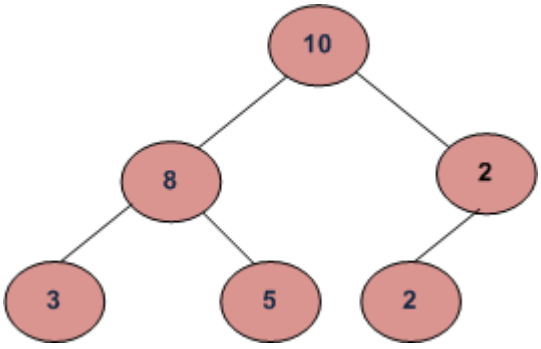
May 26, 2010

Total number of possible Binary Search Trees with n different keys = Catalan number Cn = (2n)!/(n+1)!*n!

Given a binary tree, print all root-to-leaf paths

April 18, 2010

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)

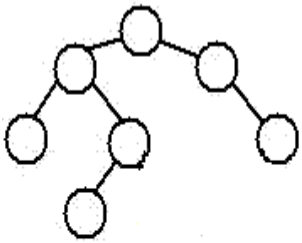
How to determine if a binary tree is height-balanced?

March 20, 2010

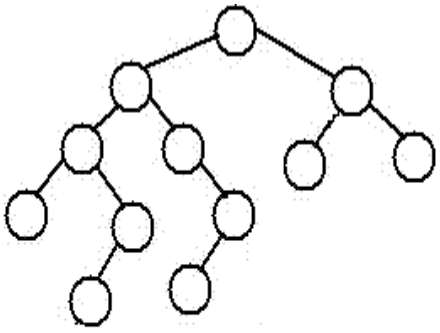
A tree where no leaf is much farther away from the root than any other leaf. Different balancing schemes allow different definitions of “much farther” and different amounts of work to keep them balanced.

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
```

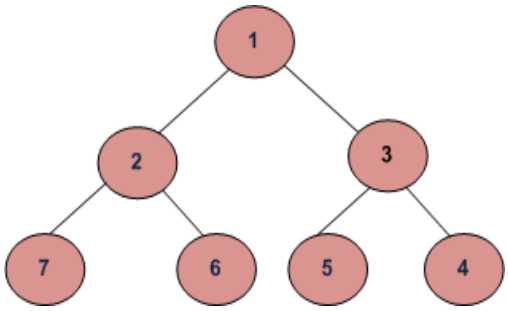
```
    are balanced then return true */
  else return l&&r;
}


/* 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)

Level order traversal in spiral form

December 20, 2009

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



Method 1 (Recursive)
This problem can bee seen as an extension of the level order traversal post.
To print the nodes in spiral order, nodes at different levels should be printed in alternating order. An additional Boolean variable *ltr* is used to change printing order of levels. If *ltr* is 1 then printGivenLevel() prints nodes from left to right else from right to left. Value of *ltr* is flipped in each iteration to change the order.

Function to print level order traversal of tree

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

Function to print all nodes at a given level

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

Following is C implementation of above algorithm.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.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 ltr);
int height(struct node* node);
struct node* newNode(int data);

/* Function to print spiral traversal of a tree*/
void printSpiral(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 = false;
    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("Spiral Order traversal of binary tree is \n");
    printSpiral(root);

    return 0;
}
```

Output:

```
Spiral Order traversal of binary tree is
1 2 3 4 5 6 7
```

Time Complexity: Worst case time complexity of the above method is **O(n^2)**. Worst case occurs in case of skewed trees.

Method 2 (Iterative)

We can print spiral order traversal in **O(n) time** and O(n) extra space. The idea is to use two stacks. We can use one stack for printing from left to right and other stack for printing from right to left. In every iteration, we have nodes of one level in one of the stacks. We print the nodes, and push nodes of next level in other stack.

```
// C++ implementation of a O(n) time method for spiral order traversal
#include <iostream>
#include <stack>
using namespace std;

// Binary Tree node
struct node
{
    int data;
    struct node *left, *right;
};

void printSpiral(struct node *root)
{
    if (root == NULL)  return;   // NULL check

    // Create two stacks to store alternate levels
    stack<struct node*> s1;  // For levels to be printed from right to left
    stack<struct node*> s2;  // For levels to be printed from left to right

    // Push first level to first stack 's1'
    s1.push(root);

    // Keep ptinting while any of the stacks has some nodes
    while (!s1.empty() || !s2.empty())
    {
        // Print nodes of current level from s1 and push nodes of
        // next level to s2
        while (!s1.empty())
        {
            struct node *temp = s1.top();
            s1.pop();
            cout << temp->data << " ";

            // Note that is right is pushed before left
            if (temp->right)
                s2.push(temp->right);
            if (temp->left)
                s2.push(temp->left);
        }

        // Print nodes of current level from s2 and push nodes of
        // next level to s1
        while (!s2.empty())
        {
            struct node *temp = s2.top();
            s2.pop();
            cout << temp->data << " ";

            // Note that is left is pushed before right
            if (temp->left)
                s1.push(temp->left);
            if (temp->right)
                s1.push(temp->right);
        }
    }
}

// A utility functiont to create a new node
```

```
struct node* newNode(int data)
{
    struct node* node = new 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(7);
    root->left->right = newNode(6);
    root->right->left  = newNode(5);
    root->right->right = newNode(4);
    cout << "Spiral Order traversal of binary tree is \n";
    printSpiral(root);

    return 0;
}
```

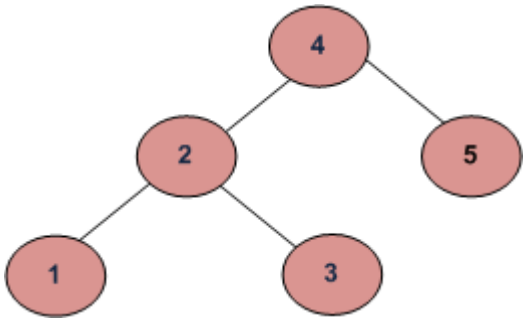
Output:

```
Spiral Order traversal of binary tree is
1 2 3 4 5 6 7
```

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.

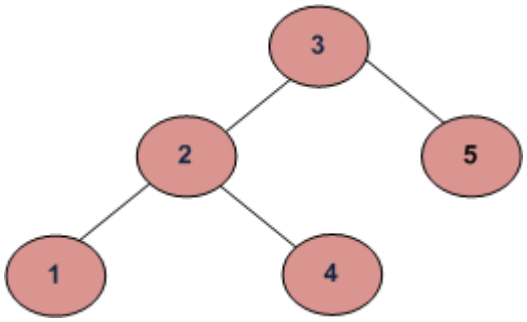
From the above properties it naturally follows that:
• Each node (item in the tree) has a distinct key.



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-1) &&  // Allow only distinct
values
    isBSTUtil(node->right, node->data+1, max);  // Allow only distinct
values
}

/* 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)
Thanks to LJW489 for suggesting this method.
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)

We can avoid the use of Auxiliary Array. While doing In-Order traversal, we can keep track of previously visited node. If the value of the currently visited node is less than the previous value, then tree is not BST. Thanks to ygos for this space optimization.

```
bool isBST(struct node* root)
{
    static struct node *prev = NULL;

    // traverse the tree in inorder fashion and keep track of prev node
    if (root)
    {
        if (!isBST(root->left))
          return false;

        // Allows only distinct valued nodes
        if (prev != NULL && root->data <= prev->data)
          return false;

        prev = root;

        return isBST(root->right);
    }

    return true;
}
```

The use of static variable can also be avoided by using reference to prev node as a parameter (Similar to this post).

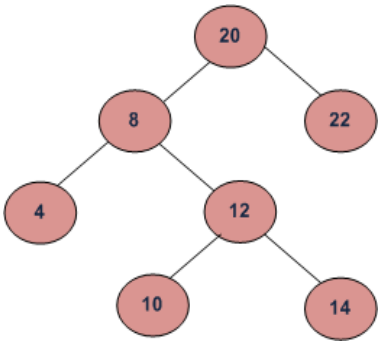
Lowest Common Ancestor in a Binary Search Tree.

August 9, 2009

Given values of two nodes in a Binary Search Tree, write a c program to find the **L**owest **C**ommon **A**ncestor (LCA). You may assume that both the values exist in the tree.

The function prototype should be as follows:

```
struct node *lca(node* root, int n1, int n2)
n1 and n2 are two given values in the tree with given root.
```



For example, consider the BST in diagram, LCA of 10 and 14 is 12 and LCA of 8 and 14 is 8.

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

The LCA of n1 and n2 in T is the shared ancestor of n1 and n2 that is located farthest from the root. Computation of lowest common ancestors may be useful, for instance, as part of a procedure for determining the distance between pairs of nodes in a tree: the distance from n1 to n2 can be computed as the distance from the root to n1, plus the distance from the root to n2, minus twice the distance from the root to their lowest common ancestor. (Source Wiki)

Solutions:
If we are given a BST where every node has **parent pointer**, then LCA can be easily determined by traversing up using parent pointer and printing the first intersecting node.

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

```
// A recursive C program to find LCA of two nodes n1 and n2.
#include <stdio.h>
#include <stdlib.h>

struct node
{
    int data;
    struct node* left, *right;
};

/* Function to find LCA of n1 and n2. The function assumes that both
   n1 and n2 are present in BST */
```

```
struct node *lca(struct node* root, int n1, int n2)
{
    if (root == NULL) return NULL;

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

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

    return root;
}

/* Helper function that allocates a new node with the given data.*/
struct node* newNode(int data)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->data  = data;
    node->left  = node->right = NULL;
    return(node);
}

/* Driver program to test mirror() */
int main()
{
    // Let us construct the BST shown in the above figure
    struct node *root        = newNode(20);
    root->left               = newNode(8);
    root->right              = newNode(22);
    root->left->left         = newNode(4);
    root->left->right        = newNode(12);
    root->left->right->left  = newNode(10);
    root->left->right->right = newNode(14);

    int n1 = 10, n2 = 14;
    struct node *t = lca(root, n1, n2);
    printf("LCA of %d and %d is %d \n", n1, n2, t->data);

    n1 = 14, n2 = 8;
    t = lca(root, n1, n2);
    printf("LCA of %d and %d is %d \n", n1, n2, t->data);

    n1 = 10, n2 = 22;
    t = lca(root, n1, n2);
    printf("LCA of %d and %d is %d \n", n1, n2, t->data);

    getchar();
    return 0;
}
```

Output:

```
LCA of 10 and 14 is 12
LCA of 14 and 8 is 8
LCA of 10 and 22 is 20
```

Time complexity of above solution is O(h) where h is height of tree. Also, the above solution requires O(h) extra space in function call stack for recursive function calls. We can avoid extra space using **iterative solution**.

```
/* Function to find LCA of n1 and n2. The function assumes that both
   n1 and n2 are present in BST */
struct node *lca(struct node* root, int n1, int n2)
{
    while (root != NULL)
    {
         // If both n1 and n2 are greater than root, then LCA lies in left
        if (root->data > n1 && root->data > n2)
            root = root->left;

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

        else break;
    }
    return root;
}
```

See this for complete program.

Extract Leaves of a Binary Tree in a Doubly Linked List

Given a Binary Tree, extract all leaves of it in a **D**oubly **L**inked **L**ist (DLL). Note that the DLL need to be created in-place. Assume that the node structure of DLL and Binary Tree is same, only the meaning of left and right pointers are different. In DLL, left means previous pointer and right means next pointer.

```
Let the following be input binary tree
         1
      /     \
     2       3
    / \       \
   4   5       6
  / \         / \
 7   8       9   10


Output:
Doubly Linked List
7<->8<->5<->9<->10

Modified Tree:
         1
      /     \
     2       3
    /         \
```

4 6

We strongly recommend you to minimize the browser and try this yourself first.

We need to traverse all leaves and connect them by changing their left and right pointers. We also need to remove them from Binary Tree by changing left or right pointers in parent nodes. There can be many ways to solve this. In the following implementation, we add leaves at the beginning of current linked list and update head of the list using pointer to head pointer. Since we insert at the beginning, we need to process leaves in reverse order. For reverse order, we first traverse the right subtree then the left subtree. We use return values to update left or right pointers in parent nodes.

```
// C program to extract leaves of a Binary Tree in a Doubly Linked List
#include <stdio.h>
#include <stdlib.h>

// Structure for tree and linked list
struct Node
{
    int data;
    struct Node *left, *right;
};

// Main function which extracts all leaves from given Binary Tree.
// The function returns new root of Binary Tree (Note that root may change
// if Binary Tree has only one node).  The function also sets *head_ref as
// head of doubly linked list.  left pointer of tree is used as prev in DLL
// and right pointer is used as next
struct Node* extractLeafList(struct Node *root, struct Node **head_ref)
{
   // Base cases
   if (root == NULL)  return NULL;

   if (root->left == NULL && root->right == NULL)
   {
       // This node is going to be added to doubly linked list
       // of leaves, set right pointer of this node as previous
       // head of DLL. We don't need to set left pointer as left
       // is already NULL
       root->right = *head_ref;

       // Change left pointer of previous head
       if (*head_ref != NULL) (*head_ref)->left = root;

       // Change head of linked list
       *head_ref = root;

       return NULL;  // Return new root
   }

   // Recur for right and left subtrees
   root->right = extractLeafList(root->right, head_ref);
   root->left  = extractLeafList(root->left, head_ref);

   return root;
}
```

```
// Utility function for allocating node for Binary Tree.
struct Node* newNode(int data)
{
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// Utility function for printing tree in In-Order.
void print(struct Node *root)
{
    if (root != NULL)
    {
         print(root->left);
         printf("%d ",root->data);
         print(root->right);
    }
}

// Utility function for printing double linked list.
void printList(struct Node *head)
{
     while (head)
     {
         printf("%d ", head->data);
         head = head->right;
     }
}

// Driver program to test above function
int main()
{
     struct Node *head = NULL;
     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(6);
     root->left->left->left = newNode(7);
     root->left->left->right = newNode(8);
     root->right->right->left = newNode(9);
     root->right->right->right = newNode(10);

     printf("Inorder Trvaersal of given Tree is:\n");
     print(root);

     root = extractLeafList(root, &head);

     printf("\nExtracted Double Linked list is:\n");
     printList(head);

     printf("\nInorder traversal of modified tree is:\n");
     print(root);
     return 0;
}
```

Output:

```
Inorder Trvaersal of given Tree is:
7 4 8 2 5 1 3 9 6 10
Extracted Double Linked list is:
7 8 5 9 10
Inorder traversal of modified tree is:
4 2 1 3 6
```

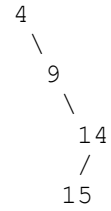
Time Complexity: O(n), the solution does a single traversal of given Binary Tree.

Remove all nodes which don’t lie in any path with sum>= k

Given a binary tree, a complete path is defined as a path from root to a leaf. The sum of all nodes on that path is defined as the sum of that path. Given a number K, you have to remove (prune the tree) all nodes which don’t lie in any path with sum>=k.

Note: A node can be part of multiple paths. So we have to delete it only in case when all paths from it have sum less than K.

```
Consider the following Binary Tree
          1
       /      \
      2        3
    /   \     /  \
   4     5   6    7
  / \    /       /
 8   9  12      10
    / \           \
   13  14         11
       /
      15

For input k = 20, the tree should be changed to following
(Nodes with values 6 and 8 are deleted)
          1
       /      \
      2        3
    /   \        \
   4     5        7
    \    /       /
     9  12      10
    / \           \
   13  14         11
       /
      15

For input k = 45, the tree should be changed to following.
       1
     /
    2
   /
```



We strongly recommend you to minimize the browser and try this yourself first.

The idea is to traverse the tree and delete nodes in bottom up manner. While traversing the tree, recursively calculate the sum of nodes from root to leaf node of each path. For each visited node, checks the total calculated sum against given sum “k”. If sum is less than k, then free(delete) that node (leaf node) and return the sum back to the previous node. Since the path is from root to leaf and nodes are deleted in bottom up manner, a node is deleted only when all of its descendants are deleted. Therefore, when a node is deleted, it must be a leaf in the current Binary Tree.

Following is C implementation of the above approach.

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

// A utility function to get maximum of two integers
int max(int l, int r) { return (l > r ? l : r); }

// A Binary Tree Node
struct Node
{
    int data;
    struct Node *left, *right;
};

// A utility function to create a new Binary Tree node with given data
struct Node* newNode(int data)
{
    struct Node* node = (struct Node*) malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// print the tree in LVR (Inorder traversal) way.
void print(struct Node *root)
{
    if (root != NULL)
    {
        print(root->left);
        printf("%d ",root->data);
        print(root->right);
    }
}

/* Main function which truncates the binary tree. */
struct Node *pruneUtil(struct Node *root, int k, int *sum)
```

```
{
    // Base Case
    if (root == NULL)  return NULL;

    // Initialize left and right sums as sum from root to
    // this node (including this node)
    int lsum = *sum + (root->data);
    int rsum = lsum;

    // Recursively prune left and right subtrees
    root->left = pruneUtil(root->left, k, &lsum);
    root->right = pruneUtil(root->right, k, &rsum);

    // Get the maximum of left and right sums
    *sum = max(lsum, rsum);

    // If maximum is smaller than k, then this node
    // must be deleted
    if (*sum < k)
    {
        free(root);
        root = NULL;
    }

    return root;
}

// A wrapper over pruneUtil()
struct Node *prune(struct Node *root, int k)
{
    int sum = 0;
    return pruneUtil(root, k, &sum);
}

// Driver program to test above function
int main()
{
    int k = 45;
    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->right->right = newNode(7);
    root->left->left->left = newNode(8);
    root->left->left->right = newNode(9);
    root->left->right->left = newNode(12);
    root->right->right->left = newNode(10);
    root->right->right->left->right = newNode(11);
    root->left->left->right->left = newNode(13);
    root->left->left->right->right = newNode(14);
    root->left->left->right->right->left = newNode(15);

    printf("Tree before truncation\n");
    print(root);

    root = prune(root, k); // k is 45
```

```
    printf("\n\nTree after truncation\n");
    print(root);

    return 0;
}
```

Output:

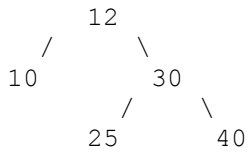
```
Tree before truncation
8 4 13 9 15 14 2 12 5 1 6 3 10 11 7

Tree after truncation
4 9 15 14 2 1
```

Time Complexity: O(n), the solution does a single traversal of given Binary Tree.

Print Left View of a Binary Tree

August 30, 2013

Given a Binary Tree, print left view of it. Left view of a Binary Tree is set of nodes visible when tree is visited from left side. Left view of following tree is 12, 10, 25.



The left view contains all nodes that are first nodes in their levels. A simple solution is to **do level order traversal** and print the first node in every level.

The problem can also be solved **using simple recursive traversal**. We can keep track of level of a node by passing a parameter to all recursive calls. The idea is to keep track of maximum level also. Whenever we see a node whose level is more than maximum level so far, we print the node because this is the first node in its level (Note that we traverse the left subtree before right subtree). Following is C implementation of this approach.

```
// C program to print left view of Binary Tree
#include<stdio.h>
#include<stdlib.h>

struct node
{
    int data;
    struct node *left, *right;
};

// A utility function to create a new Binary Tree node
```

```
struct node *newNode(int item)
{
    struct node *temp =  (struct node *)malloc(sizeof(struct node));
    temp->data = item;
    temp->left = temp->right = NULL;
    return temp;
}

// Recursive function to print left view of a binary tree.
void leftViewUtil(struct node *root, int level, int *max_level)
{
    // Base Case
    if (root==NULL)  return;

    // If this is the first node of its level
    if (*max_level < level)
    {
        printf("%d\t", root->data);
        *max_level = level;
    }

    // Recur for left and right subtrees
    leftViewUtil(root->left, level+1, max_level);
    leftViewUtil(root->right, level+1, max_level);
}

// A wrapper over leftViewUtil()
void leftView(struct node *root)
{
    int max_level = 0;
    leftViewUtil(root, 1, &max_level);
}

// Driver Program to test above functions
int main()
{
    struct node *root = newNode(12);
    root->left = newNode(10);
    root->right = newNode(30);
    root->right->left = newNode(25);
    root->right->right = newNode(40);

    leftView(root);

    return 0;
}
```

Output:

```
12      10      25
```

Print Postorder traversal from given Inorder and Preorder traversals

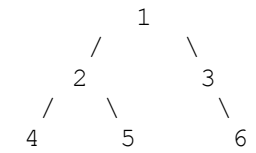
August 22, 2013

Given Inorder and Preorder traversals of a binary tree, print Postorder traversal.

Example:

```
Input:
Inorder traversal in[] = {4, 2, 5, 1, 3, 6}
Preorder traversal pre[] = {1, 2, 4, 5, 3, 6}

Output:
Postorder traversal is {4, 5, 2, 6, 3, 1}
```

Trversals in the above example represents following tree



A **naive method** is to first construct the tree, then use simple recursive method to print postorder traversal of the constructed tree.

We can print postorder traversal without constructing the tree. The idea is, root is always the first item in preorder traversal and it must be the last item in postorder traversal. We first recursively print left subtree, then recursively print right subtree. Finally, print root. To find boundaries of left and right subtrees in pre[] and in[], we search root in in[], all elements before root in in[] are elements of left subtree and all elements after root are elements of right subtree. In pre[], all elements after index of root in in[] are elements of right subtree. And elements before index (including the element at index and excluding the first element) are elements of left subtree.

```
// C++ program to print postorder traversal from preorder and inorder
traversals
#include <iostream>
using namespace std;

// A utility function to search x in arr[] of size n
int search(int arr[], int x, int n)
{
    for (int i = 0; i < n; i++)
      if (arr[i] == x)
         return i;
    return -1;
}

// Prints postorder traversal from given inorder and preorder traversals
void printPostOrder(int in[], int pre[], int n)
{
   // The first element in pre[] is always root, search it
   // in in[] to find left and right subtrees
   int root = search(in, pre[0], n);

   // If left subtree is not empty, print left subtree
   if (root != 0)
```

```
      printPostOrder(in, pre+1, root);

   // If right subtree is not empty, print right subtree
   if (root != n-1)
      printPostOrder(in+root+1, pre+root+1, n-root-1);

   // Print root
   cout << pre[0] << " ";
}

// Driver program to test above functions
int main()
{
   int in[] = {4, 2, 5, 1, 3, 6};
   int pre[] =  {1, 2, 4, 5, 3, 6};
   int n = sizeof(in)/sizeof(in[0]);
   cout << "Postorder traversal " << endl;
   printPostOrder(in, pre, n);
   return 0;
}
```

Output

```
Postorder traversal
4 5 2 6 3 1
```

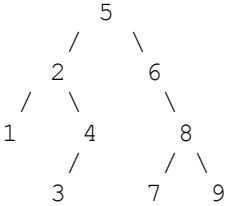
Time Complexity: The above function visits every node in array. For every visit, it calls search which takes O(n) time. Therefore, overall time complexity of the function is $O(n^2)$

Difference between sums of odd level and even level nodes of a Binary Tree

August 19, 2013

Given a a Binary Tree, find the difference between the sum of nodes at odd level and the sum of nodes at even level. Consider root as level 1, left and right children of root as level 2 and so on.

For example, in the following tree, sum of nodes at odd level is (5 + 1 + 4 + 8) which is 18. And sum of nodes at even level is (2 + 6 + 3 + 7 + 9) which is 27. The output for following tree should be 18 – 27 which is -9.



A straightforward method is to **use level order traversal**. In the traversal, check level of current node, if it is odd, increment odd sum by data of current node, otherwise increment even sum. Finally return difference between odd sum and even sum. See following for implementation of this approach.

C implementation of level order traversal based approach to find the difference.

The problem can also be solved **using simple recursive traversal**. We can recursively calculate the required difference as, value of root's data subtracted by the difference for subtree under left child and the difference for subtree under right child. Following is C implementation of this approach.

```
// A recursive program to find difference between sum of nodes at
// odd level and sum at even level
#include <stdio.h>
#include <stdlib.h>

// Binary Tree node
struct node
{
    int data;
    struct node* left, *right;
};

// A utility function to allocate a new tree node with given data
struct node* newNode(int data)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left =  node->right = NULL;
    return (node);
}

// The main function that return difference between odd and even level
// nodes
int getLevelDiff(struct node *root)
{
   // Base case
   if (root == NULL)
         return 0;

   // Difference for root is root's data - difference for left subtree
   // - difference for right subtree
   return root->data - getLevelDiff(root->left) - getLevelDiff(root-
>right);
}

// Driver program to test above functions
int main()
{
    struct node *root = newNode(5);
    root->left = newNode(2);
    root->right = newNode(6);
    root->left->left  = newNode(1);
    root->left->right = newNode(4);
    root->left->right->left = newNode(3);
    root->right->right = newNode(8);
```

```
    root->right->right->right = newNode(9);
    root->right->right->left = newNode(7);
    printf("%d is the required difference\n", getLevelDiff(root));
    getchar();
    return 0;
}
```

Output:

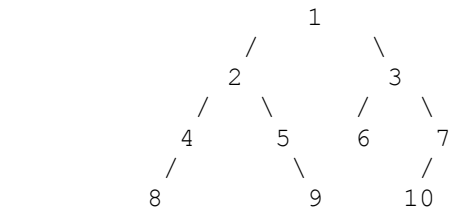
```
-9 is the required difference
```

Time complexity of both methods is O(n), but the second method is simple and easy to implement.

Print ancestors of a given binary tree node without recursion

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, consider the following Binary Tree



Following are different input keys and their ancestors in the above tree

```
Input Key    List of Ancestors
------------------------
 1
 2             1
 3             1
 4             2 1
 5             2 1
 6             3 1
 7             3 1
 8             4 2 1
 9             5 2 1
10             7 3 1
```

Recursive solution for this problem is discussed here.
It is clear that we need to use a stack based iterative traversal of the Binary Tree. The idea is to have all ancestors in stack when we reach the node with given key. Once we reach the key, all we have to do is, print contents of stack.
How to get all ancestors in stack when we reach the given node? We can traverse all nodes in Postorder way. If we take a closer look at the recursive postorder traversal, we can easily

observe that, when recursive function is called for a node, the recursion call stack contains ancestors of the node. So idea is do iterative Postorder traversal and stop the traversal when we reach the desired node.

Following is C implementation of the above approach.

```
// C program to print all ancestors of a given key
#include <stdio.h>
#include <stdlib.h>

// Maximum stack size
#define MAX_SIZE 100

// Structure for a tree node
struct Node
{
    int data;
    struct Node *left, *right;
};

// Structure for Stack
struct Stack
{
    int size;
    int top;
    struct Node* *array;
};

// A utility function to create a new tree node
struct Node* newNode(int data)
{
    struct Node* node = (struct Node*) malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// A utility function to create a stack of given size
struct Stack* createStack(int size)
{
    struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));
    stack->size = size;
    stack->top = -1;
    stack->array = (struct Node**) malloc(stack->size * sizeof(struct
Node*));
    return stack;
}

// BASIC OPERATIONS OF STACK
int isFull(struct Stack* stack)
{
    return stack->top - 1 == stack->size;
}
int isEmpty(struct Stack* stack)
{
    return stack->top == -1;
}
void push(struct Stack* stack, struct Node* node)
```

```
{
    if (isFull(stack))
        return;
    stack->array[++stack->top] = node;
}
struct Node* pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return NULL;
    return stack->array[stack->top--];
}
struct Node* peek(struct Stack* stack)
{
    if (isEmpty(stack))
        return NULL;
    return stack->array[stack->top];
}

// Iterative Function to print all ancestors of a given key
void printAncestors(struct Node *root, int key)
{
    if (root == NULL) return;

    // Create a stack to hold ancestors
    struct Stack* stack = createStack(MAX_SIZE);

    // Traverse the complete tree in postorder way till we find the key
    while (1)
    {
        // Traverse the left side. While traversing, push the nodes into
        // the stack so that their right subtrees can be traversed later
        while (root && root->data != key)
        {
            push(stack, root);   // push current node
            root = root->left;  // move to next node
        }

        // If the node whose ancestors are to be printed is found,
        // then break the while loop.
        if (root && root->data == key)
            break;

        // Check if right sub-tree exists for the node at top
        // If not then pop that node because we don't need this
        // node any more.
        if (peek(stack)->right == NULL)
        {
            root = pop(stack);

            // If the popped node is right child of top, then remove the
top
            // as well. Left child of the top must have processed before.
            // Consider the following tree for example and key = 3.  If we
            // remove the following loop, the program will go in an
            // infinite loop after reaching 5.
            //          1
            //        /   \
            //       2     3
            //         \
            //           4
```

```
            //             \
            //              5
            while (!isEmpty(stack) && peek(stack)->right == root)
               root = pop(stack);
        }

        // if stack is not empty then simply set the root as right child
        // of top and start traversing right sub-tree.
        root = isEmpty(stack)? NULL: peek(stack)->right;
    }

    // If stack is not empty, print contents of stack
    // Here assumption is that the key is there in tree
    while (!isEmpty(stack))
        printf("%d ", pop(stack)->data);
}

// Driver program to test above functions
int main()
{
    // Let us construct a binary tree
    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->right->right = newNode(7);
    root->left->left->left = newNode(8);
    root->left->right->right = newNode(9);
    root->right->right->left = newNode(10);

    printf("Following are all keys and their ancestors\n");
    for (int key = 1; key <= 10; key++)
    {
        printf("%d: ", key);
        printAncestors(root, key);
        printf("\n");
    }

    getchar();
    return 0;
}
```

Output:

```
Following are all keys and their ancestors
1:
2: 1
3: 1
4: 2 1
5: 2 1
6: 3 1
7: 3 1
8: 4 2 1
9: 5 2 1
10: 7 3 1
```

Check for Identical BSTs without building the trees

June 23, 2013

Given two arrays which represent a sequence of keys. Imagine we make a Binary Search Tree (BST) from each array. We need to tell whether two BSTs will be identical or not without actually constructing the tree.

Examples
For example, the input arrays are {2, 4, 3, 1} and {2, 1, 4, 3} will construct the same tree

```
 Let the input arrays be a[] and b[]

Example 1:
a[] = {2, 4, 1, 3} will construct following tree.
   2
 /  \
1    4
    /
   3
b[] = {2, 4, 3, 1} will also also construct the same tree.
   2
 /  \
1    4
    /
   3
So the output is "True"

Example 2:
a[] = {8, 3, 6, 1, 4, 7, 10, 14, 13}
b[] = {8, 10, 14, 3, 6, 4, 1, 7, 13}

They both construct the same following BST, so output is "True"
            8
          /    \
        3       10
      /  \         \
    1     6        14
        /   \     /
       4     7   13
```

Solution:
According to BST property, elements of left subtree must be smaller and elements of right subtree must be greater than root.
Two arrays represent sane BST if for every element x, the elements in left and right subtrees of x appear after it in both arrays. And same is true for roots of left and right subtrees.
The idea is to check of if next smaller and greater elements are same in both arrays. Same properties are recursively checked for left and right subtrees. The idea looks simple, but implementation requires checking all conditions for all elements. Following is an interesting recursive implementation of the idea.

```
// A C program to check for Identical BSTs without building the trees
#include<stdio.h>
#include<limits.h>
#include<stdbool.h>
```

```
/* The main function that checks if two arrays a[] and b[] of size n
construct
  same BST. The two values 'min' and 'max' decide whether the call is made
for
  left subtree or right subtree of a parent element. The indexes i1 and i2
are
  the indexes in (a[] and b[]) after which we search the left or right
child.
  Initially, the call is made for INT_MIN and INT_MAX as 'min' and 'max'
  respectively, because root has no parent.
  i1 and i2 are just after the indexes of the parent element in a[] and
b[]. */
bool isSameBSTUtil(int a[], int b[], int n, int i1, int i2, int min, int max)
{
   int j, k;

   /* Search for a value satisfying the constraints of min and max in a[]
and
      b[]. If the parent element is a leaf node then there must be some
      elements in a[] and b[] satisfying constraint. */
   for (j=i1; j<n; j++)
       if (a[j]>min && a[j]<max)
           break;
   for (k=i2; k<n; k++)
       if (b[k]>min && b[k]<max)
           break;

   /* If the parent element is leaf in both arrays */
   if (j==n && k==n)
       return true;

   /* Return false if any of the following is true
      a) If the parent element is leaf in one array, but non-leaf in other.
      b) The elements satisfying constraints are not same. We either search
         for left child or right child of the parent element (decinded by
min
         and max values). The child found must be same in both arrays */
   if (((j==n)^(k==n)) || a[j]!=b[k])
       return false;

   /* Make the current child as parent and recursively check for left and
right
      subtrees of it. Note that we can also pass a[k] in place of a[j] as
they
      are both are same */
   return isSameBSTUtil(a, b, n, j+1, k+1, a[j], max) &&  // Right Subtree
          isSameBSTUtil(a, b, n, j+1, k+1, min, a[j]);    // Left Subtree
}

// A wrapper over isSameBSTUtil()
bool isSameBST(int a[], int b[], int n)
{
   return isSameBSTUtil(a, b, n, 0, 0, INT_MIN, INT_MAX);
}

// Driver program to test above functions
int main()
{
   int a[] = {8, 3, 6, 1, 4, 7, 10, 14, 13};
```

```
    int b[] = {8, 10, 14, 3, 6, 4, 1, 7, 13};
    int n=sizeof(a)/sizeof(a[0]);
    printf("%s\n", isSameBST(a, b, n)?
               "BSTs are same":"BSTs not same");
    return 0;
}
```

Output:

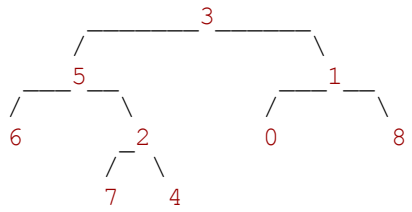
```
BSTs are same
```

Lowest Common Ancestor of a Binary Tree Part II

July 21, 2011 in binary tree

Given a binary tree, find the lowest common ancestor of two given nodes in the tree. Each node contains a parent pointer which links to its parent.

Note:
This is Part II of Lowest Common Ancestor of a Binary Tree. If you need to find the lowest common ancestor without parent pointers, please read Lowest Common Ancestor of a Binary Tree Part I.



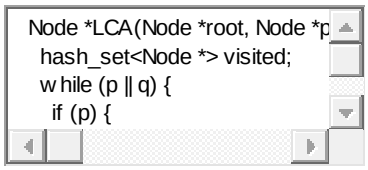
If you are not so sure about the definition of lowest common ancestor (LCA), please refer to my previous post: Lowest Common Ancestor of a Binary Search Tree (BST) or the definition of LCA here. Using the tree above as an example, the LCA of nodes 5 and 1 is 3. Please note that LCA for nodes 5 and 4 is 5.

In my last post: Lowest Common Ancestor of a Binary Tree Part I, we have devised a recursive solution which finds the LCA in O(*n*) time. If each node has a pointer that link to its parent, could we devise a better solution?

Hint:
No recursion is needed. There is an easy solution which uses extra space. Could you eliminate the need of extra space?

An easy solution:
As we trace the two paths from both nodes up to the root, eventually it will merge into one single path. The LCA is the exact first intersection node where both paths merged into a single path. An easy solution is to use a hash table which records visited nodes as we trace

both paths up to the root. Once we reached the first node which is already marked as visited, we immediately return that node as the LCA.

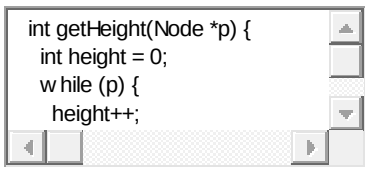



```
1  Node *LCA(Node *root, Node *p, Node *q) {
2    hash_set<Node *> visited;
3    while (p || q) {
4      if (p) {
5        if (!visited.insert(p).second)
6          return p; // insert p failed (p exists in the table)
7        p = p->parent;
8      }
9      if (q) {
10       if (!visited.insert(q).second)
11         return q; // insert q failed (q exists in the table)
12       q = q->parent;
13     }
14   }
15   return NULL;
16 }
```

The run time complexity of this approach is O(*h*), where *h* is the tree's height. The space complexity is also O(*h*), since it can mark at most 2*h* nodes.

The best solution:
A little creativity is needed here. Since we have the parent pointer, we could easily get the distance (height) of both nodes from the root. Once we knew both heights, we could subtract from one another and get the height's difference (*dh*). If you observe carefully from the previous solution, the node which is closer to the root is always *dh* steps ahead of the deeper node. We could eliminate the need of marking visited nodes altogether. Why?

The reason is simple, if we advance the deeper node *dh* steps above, both nodes would be at the same depth. Then, we advance both nodes one level at a time. They would then eventually intersect at one node, which is the LCA of both nodes. If not, one of the node would eventually reach NULL (root's parent), which we conclude that both nodes are not in the same tree. However, that part of code shouldn't be reached, since the problem statement assumed that both nodes are in the same tree.



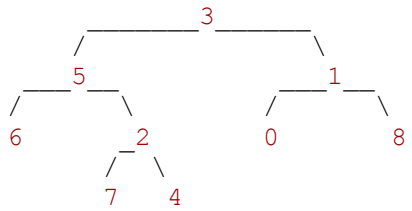

```
1  int getHeight(Node *p) {
2    int height = 0;
```

```
3     while (p) {
4       height++;
5       p = p->parent;
6     }
7     return height;
8   }
9
10 // As root->parent is NULL, we don't need to pass root in.
11 Node *LCA(Node *p, Node *q) {
12   int h1 = getHeight(p);
13   int h2 = getHeight(q);
14   // swap both nodes in case p is deeper than q.
15   if (h1 > h2) {
16     swap(h1, h2);
17     swap(p, q);
18   }
19   // invariant: h1 <= h2.
20   int dh = h2 - h1;
21   for (int h = 0; h < dh; h++)
22     q = q->parent;
23   while (p && q) {
24     if (p == q) return p;
25     p = p->parent;
26     q = q->parent;
27   }
28   return NULL;  // p and q are not in the same tree
29 }
```

Lowest Common Ancestor of a Binary Tree Part I

July 18, 2011 in binary tree

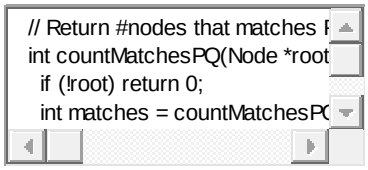
Given a binary tree, find the lowest common ancestor of two given nodes in the tree.



If you are not so sure about the definition of lowest common ancestor (LCA), please refer to my previous post: Lowest Common Ancestor of a Binary Search Tree (BST) or the definition of LCA here. Using the tree above as an example, the LCA of nodes **5** and **1** is **3**. Please note that LCA for nodes **5** and **4** is **5**.

Hint:
Top-down or bottom-up? Consider both approaches and see which one is more efficient.

A Top-Down Approach (Worst case $O(n^2)$):
Let's try the top-down approach where we traverse the nodes from the top to the bottom. First, if the current node is one of the two nodes, it must be the LCA of the two nodes. If not, we count the number of nodes that matches either p or q in the left subtree (which we call *totalMatches*). If *totalMatches* equals 1, then we know the right subtree will contain the other node. Therefore, the current node must be the LCA. If *totalMatches* equals 2, we know that both nodes are contained in the left subtree, so we traverse to its left child. Similar with the case where *totalMatches* equals 0 where we traverse to its right child.




```
1   // Return #nodes that matches P or Q in the subtree.
2   int countMatchesPQ(Node *root, Node *p, Node *q) {
3     if (!root) return 0;
4     int matches = countMatchesPQ(root->left, p, q) + countMatchesPQ(root->right, p, q);
5     if (root == p || root == q)
6       return 1 + matches;
7     else
8       return matches;
9   }
10
11 Node *LCA(Node *root, Node *p, Node *q) {
12    if (!root || !p || !q) return NULL;
13    if (root == p || root == q) return root;
14    int totalMatches = countMatchesPQ(root->left, p, q);
15    if (totalMatches == 1)
16      return root;
17    else if (totalMatches == 2)
18      return LCA(root->left, p, q);
19    else /* totalMatches == 0 */
20      return LCA(root->right, p, q);
21  }
```

What is the run time complexity of this top-down approach?

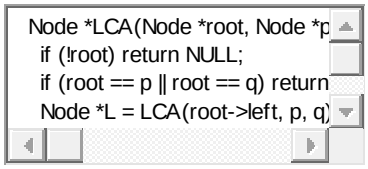
First, just for fun, we assume that the tree contains *n* nodes and is balanced (with its height equals to log(*n*)). In this case, the run time complexity would be O(*n*). Most people would guess a higher ordered complexity than O(*n*) due to the function *countMatchesPQ*() traverses the same nodes over and over again. Notice that the tree is balanced, you cut off half of the nodes you need to traverse in each recursive call of *LCA*() function. The proof that the complexity is indeed O(*n*) is left as an exercise to the reader.

What if the tree is not necessarily balanced? Then in the worst case the complexity could go up to O(n^2). Why? Could you come up with such case? (Hint: The tree might be a degenerate tree).

A Bottom-up Approach (Worst case O(*n*)):
Using a bottom-up approach, we can improve over the top-down approach by avoiding traversing the same nodes over and over again.

We traverse from the bottom, and once we reach a node which matches one of the two nodes, we pass it up to its parent. The parent would then test its left and right subtree if each contain one of the two nodes. If yes, then the parent must be the LCA and we pass its parent up to the root. If not, we pass the lower node which contains either one of the two nodes (if the left or right subtree contains either p or q), or NULL (if both the left and right subtree does not contain either p or q) up.

Sounds complicated? Surprisingly the code appears to be much simpler than the top-down one.



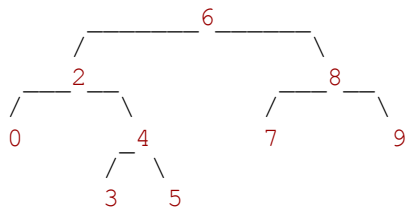

```
1 Node *LCA(Node *root, Node *p, Node *q) {
2   if (!root) return NULL;
3   if (root == p || root == q) return root;
4   Node *L = LCA(root->left, p, q);
5   Node *R = LCA(root->right, p, q);
6   if (L && R) return root;  // if p and q are on both sides
7   return L ? L : R;  // either one of p,q is on one side OR p,q is not in L&R subtrees
8 }
```

Notes:
The LCA problem had been studied extensively by many computer scientists. There exists efficient algorithms for finding LCA in constant time after initial processing of the tree in linear time. For the adventurous reader, please read this article for more details: Range Minimum Query and Lowest Common Ancestor in Topcoder.

Lowest Common Ancestor of a Binary Search Tree (BST)

July 17, 2011 in binary tree

Given a binary search tree (BST), find the lowest common ancestor of two given nodes in the BST.



Using the above tree as an example, the lowest common ancestor (LCA) of nodes 2 and 8 is 6. But how about LCA of nodes 2 and 4? Should it be 6 or 2?

According to the definition of LCA on Wikipedia: "The lowest common ancestor is defined between two nodes *v* and *w* as the lowest node in T that has both *v* and *w* as descendants (where we allow a node to be a descendant of itself)." Since a node can be a descendant of itself, the LCA of 2 and 4 should be 2, according to this definition.

Hint:
A top-down walk from the root of the tree is enough.

Solution:
There's only three cases you need to consider.

1. Both nodes are to the left of the tree.
2. Both nodes are to the right of the tree.
3. One node is on the left while the other is on the right.

For case 1), the LCA must be in its left subtree. Similar with case 2), LCA must be in its right subtree. For case 3), the current node must be the LCA.

Therefore, using a top-down approach, we traverse to the left/right subtree depends on the case of 1) or 2), until we reach case 3), which we concludes that we have found the LCA.

A careful reader might notice that we forgot to handle one extra case. What if the node itself is equal to one of the two nodes? This is the exact case from our earlier example! (The LCA of 2 and 4 should be 2). Therefore, we add case number 4):

4. When the current node equals to either of the two nodes, this node must be the LCA too.

The run time complexity is O(*h*), where *h* is the height of the BST. Translating this idea to code is straightforward (Note that we handle case 3) and 4) in the else statement to save us some extra typing ):

```
Node *LCA(Node *root, Node *p
  if (!root || !p || !q) return NULL;
  if (max(p->data, q->data) < ro
    return LCA(root->left, p, q);
```

```
1 Node *LCA(Node *root, Node *p, Node *q) {
2   if (!root || !p || !q) return NULL;
3   if (max(p->data, q->data) < root->data)
4     return LCA(root->left, p, q);
5   else if (min(p->data, q->data) > root->data)
6     return LCA(root->right, p, q);
7   else
8     return root;
9 }
```

You should realize quickly that the above code contains tail recursion. Converting the above function to an iterative one is left as an exercise to the reader.