42.Trie | (Insert and Search)

Trie is an efficient information re***trie***val data structure. Using trie, search complexities can be brought to optimal limit (key length). If we store keys in binary search tree, a well balanced BST will need time proportional to **M * log N**, where M is maximum string length and N is number of keys in tree. Using trie, we can search the key in O(M) time. However the penalty is on trie storage requirements.

Every node of trie consists of multiple branches. Each branch represents a possible character of keys. We need to mark the last node of every key as leaf node. A trie node field *value* will be used to distinguish the node as leaf node (there are other uses of the *value* field). A simple structure to represent nodes of English alphabet can be as following,

```
struct trie_node
{
    int value; /* Used to mark leaf nodes */
    trie_node_t *children[ALPHABET_SIZE];
};
```

Inserting a key into trie is simple approach. Every character of input key is inserted as an individual trie node. Note that the *children* is an array of pointers to next level trie nodes. The key character acts as an index into the array *children*. If the input key is new or an extension of existing key, we need to construct non-existing nodes of the key, and mark leaf node. If the input key is prefix of existing key in trie, we simply mark the last node of key as leaf. The key length determines trie depth.

Searching for a key is similar to insert operation, however we only compare the characters and move down. The search can terminate due to end of string or lack of key in trie. In the former case, if the*value* field of last node is non-zero then the key exists in trie. In the second case, the search terminates without examining all the characters of key, since the key is not present in trie.

The following picture explains construction of trie using keys given in the example below,

```
                       root
                    /   \    \
                    t   a     b
                    |   |     |
                    h   n     y
                    |   |  \  |
                    e   s  y  e
                 /  |   |
                 i  r   w
                 |  |   |
                        r   e    e
                                 |
                                 r
```

In the picture, every character is of type *trie_node_t*. For example, the *root* is of type trie_node_t, and it's children *a*, *b* and *t* are filled, all other nodes of root will be NULL. Similarly, "a" at the next level is having only one child ("n"), all other children are NULL. The leaf nodes are in blue.

Insert and search costs O(key_length), however the memory requirements of trie is O(ALPHABET_SIZE * key_length * N) where N is number of keys in trie. There are efficient representation of trie nodes (e.g. compressed trie, ternary search tree, etc.) to minimize memory requirements of trie.

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

#define ARRAY_SIZE(a) sizeof(a)/sizeof(a[0])

// Alphabet size (# of symbols)
#define ALPHABET_SIZE (26)
// Converts key current character into index
// use only 'a' through 'z' and lower case
#define CHAR_TO_INDEX(c) ((int)c - (int)'a')

// trie node
typedef struct trie_node trie_node_t;
struct trie_node
{
   int value;
   trie_node_t *children[ALPHABET_SIZE];
};

// trie ADT
typedef struct trie trie_t;
struct trie
{
   trie_node_t *root;
   int count;
};

// Returns new trie node (initialized to NULLs)
trie_node_t *getNode(void)
{
   trie_node_t *pNode = NULL;

   pNode = (trie_node_t *)malloc(sizeof(trie_node_t));

   if( pNode )
```

```
    {
        int i;

        pNode->value = 0;

        for(i = 0; i < ALPHABET_SIZE; i++)
        {
            pNode->children[i] = NULL;
        }
    }

    return pNode;
}

// Initializes trie (root is dummy node)
void initialize(trie_t *pTrie)
{
    pTrie->root = getNode();
    pTrie->count = 0;
}

// If not present, inserts key into trie
// If the key is prefix of trie node, just marks leaf node
void insert(trie_t *pTrie, char key[])
{
    int level;
    int length = strlen(key);
    int index;
    trie_node_t *pCrawl;

    pTrie->count++;
    pCrawl = pTrie->root;

    for( level = 0; level < length; level++ )
    {
        index = CHAR_TO_INDEX(key[level]);
        if( !pCrawl->children[index] )
        {
            pCrawl->children[index] = getNode();
        }

        pCrawl = pCrawl->children[index];
    }

    // mark last node as leaf
    pCrawl->value = pTrie->count;
}

// Returns non zero, if key presents in trie
int search(trie_t *pTrie, char key[])
{
    int level;
    int length = strlen(key);
    int index;
    trie_node_t *pCrawl;

    pCrawl = pTrie->root;

    for( level = 0; level < length; level++ )
    {
        index = CHAR_TO_INDEX(key[level]);

if( !pCrawl->children[index] )
        {
            return 0;
        }

        pCrawl = pCrawl->children[index];
    }

    return (0 != pCrawl && pCrawl->value);
}

// Driver
int main()
{
    // Input keys (use only 'a' through 'z' and lower case)
    char keys[][8] = {"the", "a", "there", "answer", "any", "by", "bye", "their"};
    trie_t trie;

    char output[][32] = {"Not present in trie", "Present in trie"};

    initialize(&trie);

    // Construct trie
    for(int i = 0; i < ARRAY_SIZE(keys); i++)
    {
        insert(&trie, keys[i]);
    }

    // Search for different keys
    printf("%s --- %s\n", "the", output[search(&trie, "the")] );
    printf("%s --- %s\n", "these", output[search(&trie, "these")] );
    printf("%s --- %s\n", "their", output[search(&trie, "their")] );
    printf("%s --- %s\n", "thaw", output[search(&trie, "thaw")] );

    return 0;
}
```

43.Trie | (Delete)

Here is an algorithm how to delete a node from trie.

During delete operation we delete the key in bottom up manner using recursion. The following are possible conditions when deleting key from trie,

1. Key may not be there in trie. Delete operation should not modify trie.
2. Key present as unique key (no part of key contains another key (prefix), nor the key itself is prefix of another key in trie). Delete all the nodes.
3. Key is prefix key of another long key in trie. Unmark the leaf node.
4. Key present in trie, having atleast one other key as prefix key. Delete nodes from end of key until first leaf node of longest prefix key.

The highlighted code presents algorithm to implement above conditions. (One may be in dilemma how a pointer passed to delete helper is reflecting changes from deleteHelper to deleteKey. Note that we are holding trie as an ADT in trie_t node, which is passed by reference or pointer).

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

#define ARRAY_SIZE(a) sizeof(a)/sizeof(a[0])

// Alphabet size (# of symbols)

#define ALPHABET_SIZE (26)
#define INDEX(c) ((int)c - (int)'a')

#define FREE(p) \
   free(p);    \
   p = NULL;

// forward declration
typedef struct trie_node trie_node_t;

// trie node
struct trie_node
{
   int value; // non zero if leaf
   trie_node_t *children[ALPHABET_SIZE];
};

// trie ADT
typedef struct trie trie_t;

struct trie
{
   trie_node_t *root;
   int count;
};

trie_node_t *getNode(void)
```

```
{
   trie_node_t *pNode = NULL;

   pNode = (trie_node_t *)malloc(sizeof(trie_node_t));

   if( pNode )
   {
      int i;

      pNode->value   = 0;

      for(i = 0; i < ALPHABET_SIZE; i++)
      {
         pNode->children[i] = NULL;
      }
   }

   return pNode;
}

void initialize(trie_t *pTrie)
{
   pTrie->root = getNode();
   pTrie->count = 0;
}

void insert(trie_t *pTrie, char key[])
{
   int level;
   int length = strlen(key);
   int index;
   trie_node_t *pCrawl;

   pTrie->count++;
   pCrawl = pTrie->root;

   for( level = 0; level < length; level++ )
   {
      index = INDEX(key[level]);

      if( pCrawl->children[index] )
      {
         // Skip current node
         pCrawl = pCrawl->children[index];
      }
      else
      {
         // Add new node
         pCrawl->children[index] = getNode();
         pCrawl = pCrawl->children[index];
      }
   }
```

```
    // mark last node as leaf (non zero)
    pCrawl->value = pTrie->count;
}

int search(trie_t *pTrie, char key[])
{
    int level;
    int length = strlen(key);
    int index;
    trie_node_t *pCrawl;

    pCrawl = pTrie->root;

    for( level = 0; level < length; level++ )
    {
        index = INDEX(key[level]);

        if( !pCrawl->children[index] )
        {
            return 0;
        }

        pCrawl = pCrawl->children[index];
    }

    return (0 != pCrawl && pCrawl->value);
}

int leafNode(trie_node_t *pNode)
{
    return (pNode->value != 0);
}

int isItFreeNode(trie_node_t *pNode)
{
    int i;
    for(i = 0; i < ALPHABET_SIZE; i++)
    {
        if( pNode->children[i] )
            return 0;
    }

    return 1;
}

bool deleteHelper(trie_node_t *pNode, char key[], int level, int len)
{
    if( pNode )
    {
        // Base case
        if( level == len )
        {
            if( pNode->value )
            {
                // Unmark leaf node
                pNode->value = 0;

                // If empty, node to be deleted
                if( isItFreeNode(pNode) )
                {
                    return true;
                }

                return false;
            }
        }
        else // Recursive case
        {
            int index = INDEX(key[level]);

            if( deleteHelper(pNode->children[index], key, level+1, len) )
            {
                // last node marked, delete it
                FREE(pNode->children[index]);

                // recursively climb up, and delete eligible nodes
                return ( !leafNode(pNode) && isItFreeNode(pNode) );
            }
        }
    }

    return false;
}

void deleteKey(trie_t *pTrie, char key[])
{
    int len = strlen(key);

    if( len > 0 )
    {
        deleteHelper(pTrie->root, key, 0, len);
    }
}

int main()
{
    char keys[][8] = {"she", "sells", "sea", "shore", "the", "by", "sheer"};
    trie_t trie;
    initialize(&trie);
    for(int i = 0; i < ARRAY_SIZE(keys); i++)
    {
        insert(&trie, keys[i]);
    }

    deleteKey(&trie, keys[0]);
    printf("%s %s\n", "she", search(&trie, "she") ? "Present in trie" : "Not present in trie");

    return 0;
}
```

44.Connect nodes at same level

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;
}
```

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.

```
            1
          /    \
        2         3
       / \       /  \
      4   5     6     7
     / \              / \
    8   9         10    11
```

45.Connect nodes at same level using constant extra space

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).

```
            1            -------------- Level 0
          /    \
        2        3       -------------- Level 1
       / \      /  \
      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
```

46.Sorted Linked List to Balanced BST

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.

```c
#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 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 TNode *root = sortedListToBST(head);
    printf("\n PreOrder Traversal of constructed BST ");
    preOrder(root);

    return 0;
}
```

Time Complexity: O(n)