Contents

Chapter 1

Memory efficient doubly linked list

Asked by Varun Bhatia.

Question:
Write a code for implementation of doubly linked list with use of single pointer in each node.

Solution:
This question is solved and very well explained at http://www.linuxjournal.com/article/6828.

We also recommend to read http://en.wikipedia.org/wiki/XOR_linked_list

Source

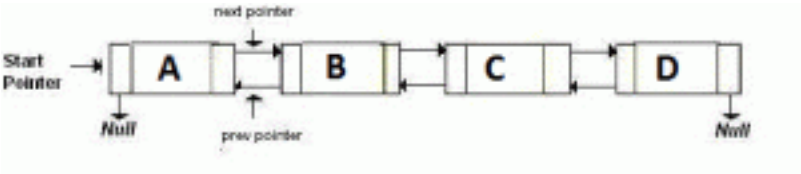
http://www.geeksforgeeks.org/memory-efficient-doubly-linked-list/

Category: Linked Lists

Chapter 2

XOR Linked List - A Memory Efficient Doubly Linked List | Set 1

An ordinary Doubly Linked List requires space for two address fields to store the addresses of previous and next nodes. A memory efficient version of Doubly Linked List can be created using only one space for address field with every node. This memory efficient Doubly Linked List is called XOR Linked List or Memory Efficient as the list uses bitwise XOR operation to save space for one address. In the XOR linked list, instead of storing actual memory addresses, every node stores the XOR of addresses of previous and next nodes.



Consider the above Doubly Linked List. Following are the Ordinary and XOR (or Memory Effiecient) representations of the Doubly Linked List.

Ordinary Representation:
Node A:
prev = NULL, next = add(B) // previous is NULL and next is address of B

Node B:
prev = add(A), next = add(C) // previous is address of A and next is address of C

Node C:
prev = add(B), next = add(D) // previous is address of B and next is address of D

Node D:
prev = add(C), next = NULL // previous is address of C and next is NULL

XOR List Representation:
Let us call the address variable in XOR representation npx (XOR of next and previous)

Node A:
npx = 0 XOR add(B) // bitwise XOR of zero and address of B

Node B:
npx = add(A) XOR add(C) // bitwise XOR of address of A and address of C

Node C:
npx = add(B) XOR add(D) // bitwise XOR of address of B and address of D

Node D:
npx = add(C) XOR 0 // bitwise XOR of address of C and 0

Traversal of XOR Linked List:
We can traverse the XOR list in both forward and reverse direction. While traversing the list we need to remember the address of the previously accessed node in order to calculate the next node's address. For example when we are at node C, we must have address of B. XOR of add(B) and *npx* of C gives us the add(D). The reason is simple: npx(C) is "add(B) XOR add(D)". If we do xor of npx(C) with add(B), we get the result as "add(B) XOR add(D) XOR add(B)" which is "add(D) XOR 0 which is "add(D)". So we have the address of next node. Similarly we can traverse the list in backward direction.

We have covered more on XOR Linked List in the following post.

XOR Linked List – A Memory Efficient Doubly Linked List | Set 2

References:
http://en.wikipedia.org/wiki/XOR_linked_list
http://www.linuxjournal.com/article/6828?page=0,0

Source

http://www.geeksforgeeks.org/xor-linked-list-a-memory-efficient-doubly-linked-list-set-1/

Chapter 3

XOR Linked List – A Memory Efficient Doubly Linked List | Set 2

In the previous post, we discussed how a Doubly Linked can be created using only one space for address field with every node. In this post, we will discuss implementation of memory efficient doubly linked list. We will mainly discuss following two simple functions.

1) A function to insert a new node at the beginning.
2) A function to traverse the list in forward direction.

In the following code, *insert()* function inserts a new node at the beginning. We need to change the head pointer of Linked List, that is why a double pointer is used (See this). Let use first discuss few things again that have been discussed in the previous post. We store XOR of next and previous nodes with every node and we call it npx, which is the only address member we have with every node. When we insert a new node at the beginning, npx of new node will always be XOR of NULL and current head. And npx of current head must be changed to XOR of new node and node next to current head.

printList() traverses the list in forward direction. It prints data values from every node. To traverse the list, we need to get pointer to the next node at every point. We can get the address of next node by keeping track of current node and previous node. If we do XOR of curr->npx and prev, we get the address of next node.

```
/* C/C++ Implementation of Memory efficient Doubly Linked List */
#include <stdio.h>
#include <stdlib.h>

// Node structure of a memory efficient doubly linked list
struct node
{
    int data;
    struct node* npx;  /* XOR of next and previous node */
};

/* returns XORed value of the node addresses */
struct node* XOR (struct node *a, struct node *b)
{
    return (struct node*) ((unsigned int) (a) ^ (unsigned int) (b));
}
```

```
/* Insert a node at the begining of the XORed linked list and makes the
   newly inserted node as head */
void insert(struct node **head_ref, int data)
{
    // Allocate memory for new node
    struct node *new_node  = (struct node *) malloc (sizeof (struct node) );
    new_node->data = data;

    /* Since new node is being inserted at the begining, npx of new node
       will always be XOR of current head and NULL */
    new_node->npx = XOR(*head_ref, NULL);

    /* If linked list is not empty, then npx of current head node will be XOR
       of new node and node next to current head */
    if (*head_ref != NULL)
    {
        // *(head_ref)->npx is XOR of NULL and next. So if we do XOR of
        // it with NULL, we get next
        struct node* next = XOR((*head_ref)->npx,  NULL);
        (*head_ref)->npx = XOR(new_node, next);
    }

    // Change head
    *head_ref = new_node;
}

// prints contents of doubly linked list in forward direction
void printList (struct node *head)
{
    struct node *curr = head;
    struct node *prev = NULL;
    struct node *next;

    printf ("Following are the nodes of Linked List: \n");

    while (curr != NULL)
    {
        // print current node
        printf ("%d ", curr->data);

        // get address of next node: curr->npx is next^prev, so curr->npx^prev
        // will be next^prev^prev which is next
        next = XOR (prev, curr->npx);

        // update prev and curr for next iteration
        prev = curr;
        curr = next;
    }
}

// Driver program to test above functions
int main ()
{
    /* Create following Doubly Linked List
```

```
       head-->40<-->30<-->20<-->10   */
    struct node *head = NULL;
    insert(&head, 10);
    insert(&head, 20);
    insert(&head, 30);
    insert(&head, 40);

    // print the created list
    printList (head);

    return (0);
}
```

Output:

```
Following are the nodes of Linked List:
40 30 20 10
```

Note that XOR of pointers is not defined by C/C++ standard. So the above implementation may not work on all platforms.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/xor-linked-list-a-memory-efficient-doubly-linked-list-set-2/

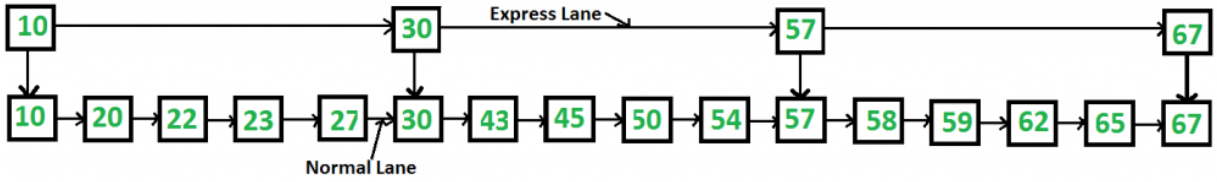
Category: Linked Lists

Chapter 4

Skip List | Set 1 (Introduction)

Can we search in a sorted linked list in better than O(n) time?
The worst case search time for a sorted linked list is O(n) as we can only linearly traverse the list and cannot skip nodes while searching. For a Balanced Binary Search Tree, we skip almost half of the nodes after one comparison with root. For a sorted array, we have random access and we can apply Binary Search on arrays.

Can we augment sorted linked lists to make the search faster? The answer is Skip List. The idea is simple, we create multiple layers so that we can skip some nodes. See the following example list with 16 nodes and two layers. The upper layer works as an "express lane" which connects only main outer stations, and the lower layer works as a "normal lane" which connects every station. Suppose we want to search for 50, we start from first node of "express lane" and keep moving on "express lane" till we find a node whose next is greater than 50. Once we find such a node (30 is the node in following example) on "express lane", we move to "normal lane" using pointer from this node, and linearly search for 50 on "normal lane". In following example, we start from 30 on "normal lane" and with linear search, we find 50.



What is the time complexity with two layers? The worst case time complexity is number of nodes on "express lane" plus number of nodes in a segment (A segment is number of "normal lane" nodes between two "express lane" nodes) of "normal lane". So if we have n nodes on "normal lane", √n (square root of n) nodes on "express lane" and we equally divide the "normal lane", then there will be √n nodes in every segment of "normal lane" . √n is actually optimal division with two layers. With this arrangement, the number of nodes traversed for a search will be O(√n). Therefore, with O(√n) extra space, we are able to reduce the time complexity to O(√n).

Can we do better?
The time complexity of skip lists can be reduced further by adding more layers. In fact, the time complexity of search, insert and delete can become O(Logn) in average case. We will soon be publishing more posts on Skip Lists.

References
MIT Video Lecture on Skip Lists
http://en.wikipedia.org/wiki/Skip_list

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/skip-list/

Chapter 5

Self Organizing List | Set 1 (Introduction)

The worst case search time for a sorted linked list is O(n). With a Balanced Binary Search Tree, we can skip almost half of the nodes after one comparison with root. For a sorted array, we have random access and we can apply Binary Search on arrays.

One idea to make search faster for Linked Lists is Skip List. Another idea (which is discussed in this post) is to *place more frequently accessed items closer to head.*. There can be two possibilities. offline (we know the complete search sequence in advance) and online (we don't know the search sequence).
In case of offline, we can put the nodes according to decreasing frequencies of search (The element having maximum search count is put first). For many practical applications, it may be difficult to obtain search sequence in advance. A Self Organizing list reorders its nodes based on searches which are done. The idea is to use locality of reference (In a typical database, 80% of the access are to 20% of the items). Following are different strategies used by Self Organizing Lists.

1) ***Move-to-Front Method***: Any node searched is moved to the front. This strategy is easy to implement, but it may over-reward infrequently accessed items as it always move the item to front.

2) ***Count Method***: Each node stores count of the number of times it was searched. Nodes are ordered by decreasing count. This strategy requires extra space for storing count.

3) ***Transpose Method***: Any node searched is swapped with the preceding node. Unlike Move-to-front, this method does not adapt quickly to changing access patterns.

Competitive Analysis:
The worst case time complexity of all methods is O(n). In worst case, the searched element is always the last element in list. For average case analysis, we need probability distribution of search sequences which is not available many times.
For online strategies and algorithms like above, we have a totally different way of analyzing them called *competitive analysis* where performance of an online algorithm is compared to the performance of an optimal offline algorithm (that can view the sequence of requests in advance). Competitive analysis is used in many practical algorithms like caching, disk paging, high performance computers. The best thing about competitive analysis is, we don't need to assume anything about probability distribution of input. The Move-to-front method is 4-competitive, means it never does more than a factor of 4 operations than offline algorithm (See the MIT video lecture for proof).

We will soon be discussing implementation and proof of the analysis given in the video lecture.

References:
http://en.wikipedia.org/wiki/Self-organizing_list

MIT Video Lecture

http://www.eecs.yorku.ca/course_archive/2003-04/F/2011/2011A/DatStr_071_SOLists.pdf

http://en.wikipedia.org/wiki/Competitive_analysis_(online_algorithm)

This article is compiled by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/self-organizing-list-set-1-introduction/

Chapter 6

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.

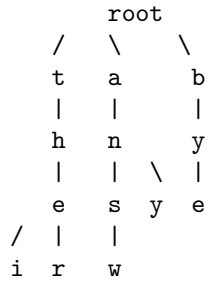
Every node of trie consists of multiple branches. Each branch represents a possible character of keys. We need to mark the last node of every key as leaf node. A trie node field *value* will be used to distinguish the node as leaf node (there are other uses of the *value* field). A simple structure to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



```
|  |   |
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;
}
```

— **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/trie-insert-and-search/

Chapter 7

Trie | (Delete)

In the previous post on trie we have described how to insert and search a node in trie. 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;
}
```

— **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/trie-delete/

Category: Trees Tags: Advance Data Structures

Post navigation

← Understanding "volatile" qualifier in C Find a Fixed Point in a given array →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 8

Longest prefix matching - A Trie based solution in Java

Given a dictionary of words and an input string, find the longest prefix of the string which is also a word in dictionary.

Examples:

```
Let the dictionary contains the following words:
{are, area, base, cat, cater, children, basement}

Below are some input/output examples:
---------------------------------------
Input String            Output
---------------------------------------
caterer                 cater
basemexy                base
child
```

Solution
We build a Trie of all dictionary words. Once the Trie is built, traverse through it using characters of input string. If prefix matches a dictionary word, store current length and look for a longer match. Finally, return the longest match.
Following is Java implementation of the above solution based.

```
import java.util.HashMap;

// Trie Node, which stores a character and the children in a HashMap
class TrieNode {
    public TrieNode(char ch)  {
        value = ch;
        children = new HashMap<>();
        bIsEnd = false;
    }
    public HashMap<Character,TrieNode> getChildren() {   return children;  }
    public char getValue()                           {   return value;     }
```

```
    public void setIsEnd(boolean val)                {   bIsEnd = val;     }
    public boolean isEnd()                           {   return bIsEnd;    }

    private char value;
    private HashMap<Character,TrieNode> children;
    private boolean bIsEnd;
}

// Implements the actual Trie
class Trie {
    // Constructor
    public Trie()   {     root = new TrieNode((char)0);       }

    // Method to insert a new word to Trie
    public void insert(String word)  {

        // Find length of the given word
        int length = word.length();
        TrieNode crawl = root;

        // Traverse through all characters of given word
        for( int level = 0; level < length; level++)
        {
            HashMap<Character,TrieNode> child = crawl.getChildren();
            char ch = word.charAt(level);

            // If there is already a child for current character of given word
            if( child.containsKey(ch))
                crawl = child.get(ch);
            else   // Else create a child
            {
                TrieNode temp = new TrieNode(ch);
                child.put( ch, temp );
                crawl = temp;
            }
        }

        // Set bIsEnd true for last character
        crawl.setIsEnd(true);
    }

    // The main method that finds out the longest string 'input'
    public String getMatchingPrefix(String input)  {
        String result = ""; // Initialize resultant string
        int length = input.length();  // Find length of the input string

        // Initialize reference to traverse through Trie
        TrieNode crawl = root;

        // Iterate through all characters of input string 'str' and traverse
        // down the Trie
        int level, prevMatch = 0;
        for( level = 0 ; level < length; level++ )
        {
```

```
            // Find current character of str
            char ch = input.charAt(level);

            // HashMap of current Trie node to traverse down
            HashMap<Character,TrieNode> child = crawl.getChildren();

            // See if there is a Trie edge for the current character
            if( child.containsKey(ch) )
            {
               result += ch;          //Update result
               crawl = child.get(ch); //Update crawl to move down in Trie

               // If this is end of a word, then update prevMatch
               if( crawl.isEnd() )
                    prevMatch = level + 1;
            }
            else  break;
        }

        // If the last processed character did not match end of a word,
        // return the previously matching prefix
        if( !crawl.isEnd() )
                return result.substring(0, prevMatch);

        else return result;
    }

    private TrieNode root;
}

// Testing class
public class Test {
   public static void main(String[] args) {
        Trie dict = new Trie();
        dict.insert("are");
        dict.insert("area");
        dict.insert("base");
        dict.insert("cat");
        dict.insert("cater");
        dict.insert("basement");

        String input = "caterer";
        System.out.print(input + ":   ");
        System.out.println(dict.getMatchingPrefix(input));

        input = "basement";
        System.out.print(input + ":   ");
        System.out.println(dict.getMatchingPrefix(input));

        input = "are";
        System.out.print(input + ":   ");
        System.out.println(dict.getMatchingPrefix(input));

        input = "arex";
```

```
        System.out.print(input + ":   ");
        System.out.println(dict.getMatchingPrefix(input));

        input = "basemexz";
        System.out.print(input + ":   ");
        System.out.println(dict.getMatchingPrefix(input));

        input = "xyz";
        System.out.print(input + ":   ");
        System.out.println(dict.getMatchingPrefix(input));
    }
}
```

Output:

```
caterer:   cater
basement:   basement
are:   are
arex:   are
basemexz:   base
xyz:
```

Time Complexity: Time complexity of finding the longest prefix is O(n) where n is length of the input string. Refer thisfor time complexity of building the Trie.

This article is compiled by **Ravi Chandra Enaganti**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/longest-prefix-matching-a-trie-based-solution-in-java/

Category: Trees Tags: Advance Data Structures, Java

Post navigation

← QuickSort on Doubly Linked List Dynamic Programming | Set 28 (Minimum insertions to form a palindrome) →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 9

Print unique rows in a given boolean matrix

Given a binary matrix, print all unique rows of the given matrix.

```
Input:
    {0, 1, 0, 0, 1}
        {1, 0, 1, 1, 0}
        {0, 1, 0, 0, 1}
        {1, 1, 1, 0, 0}
Output:
    0 1 0 0 1
    1 0 1 1 0
    1 1 1 0 0
```

Method 1 (Simple)
A simple approach is to check each row with all processed rows. Print the first row. Now, starting from the second row, for each row, compare the row with already processed rows. If the row matches with any of the processed rows, don't print it. If the current row doesn't match with any row, print it.

Time complexity: O(ROW^2 x COL)
Auxiliary Space: O(1)

Method 2 (Use Binary Search Tree)
Find the decimal equivalent of each row and insert it into BST. Each node of the BST will contain two fields, one field for the decimal value, other for row number. Do not insert a node if it is duplicated. Finally, traverse the BST and print the corresponding rows.

Time complexity: O(ROW x COL + ROW x log(ROW))
Auxiliary Space: O(ROW)

This method will lead to Integer Overflow if number of columns is large.

Method 3 (Use Trie data structure)
Since the matrix is boolean, a variant of Trie data structure can be used where each node will be having two children one for 0 and other for 1. Insert each row in the Trie. If the row is already there, don't print the row. If row is not there in Trie, insert it in Trie and print it.

Below is C implementation of method 3.

```
//Given a binary matrix of M X N of integers, you need to return only unique rows of binary array
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define ROW 4
#define COL 5

// A Trie node
typedef struct Node
{
    bool isEndOfCol;
    struct Node *child[2]; // Only two children needed for 0 and 1
} Node;


// A utility function to allocate memory for a new Trie node
Node* newNode()
{
    Node* temp = (Node *)malloc( sizeof( Node ) );
    temp->isEndOfCol = 0;
    temp->child[0] = temp->child[1] = NULL;
    return temp;
}

// Inserts a new matrix row to Trie.  If row is already
// present, then returns 0, otherwise insets the row and
// return 1
bool insert( Node** root, int (*M)[COL], int row, int col )
{
    // base case
    if ( *root == NULL )
        *root = newNode();

    // Recur if there are more entries in this row
    if ( col < COL )
        return insert ( &( (*root)->child[ M[row][col] ] ), M, row, col+1 );

    else // If all entries of this row are processed
    {
        // unique row found, return 1
        if ( !( (*root)->isEndOfCol ) )
            return (*root)->isEndOfCol = 1;

        // duplicate row found, return 0
        return 0;
    }
}

// A utility function to print a row
void printRow( int (*M)[COL], int row )
{
    int i;
```

```
    for( i = 0; i < COL; ++i )
        printf( "%d ", M[row][i] );
    printf("\n");
}

// The main function that prints all unique rows in a
// given matrix.
void findUniqueRows( int (*M)[COL] )
{
    Node* root = NULL; // create an empty Trie
    int i;

    // Iterate through all rows
    for ( i = 0; i < ROW; ++i )
        // insert row to TRIE
        if ( insert(&root, M, i, 0) )
            // unique row found, print it
            printRow( M, i );
}

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

    findUniqueRows( M );

    return 0;
}
```

Time complexity: O(ROW x COL)
Auxiliary Space: O(ROW x COL)

This method has better time complexity. Also, relative order of rows is maintained while printing.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/print-unique-rows/

Category: Arrays Tags: Advance Data Structures, Advanced Data Structures

Post navigation

← Median of two sorted arrays of different sizes Microsoft Interview | Set 8 →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 10

How to Implement Reverse DNS Look Up Cache?

Reverse DNS look up is using an internet IP address to find a domain name. For example, if you type 74.125.200.106 in browser, it automatically redirects to google.in.

How to implement Reverse DNS Look Up cache? Following are the operations needed from cache.
1) Add a IP address to URL Mapping in cache.
2) Find URL for a given IP address.

One solution is to use Hashing.

In this post, a Triebased solution is discussed. One advantage of Trie based solutions is, worst case upper bound is O(1) for Trie, for hashing, the best possible average case time complexity is O(1). Also, with Trie we can implement prefix search (finding all urls for a common prefix of IP addresses).
The general disadvantage of Trie is large amount of memory requirement, this is not a major problem here as the alphabet size is only 11 here. Ten characters are needed for digits from '0' to '9' and one for dot ('.').
The idea is to store IP addresses in Trie nodes and in the last node we store the corresponding domain name.
Following is C style implementation in C++.

```
// C based program to implement reverse DNS lookup
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

// There are atmost 11 different chars in a valid IP address
#define CHARS 11

// Maximum length of a valid IP address
#define MAX 50

// A utility function to find index of child for a given character 'c'
int getIndex(char c) { return (c == '.')? 10: (c - '0'); }

// A utility function to find character for a given child index.
char getCharFromIndex(int i) { return (i== 10)? '.' : ('0' + i); }

// Trie Node.
struct trieNode
```

```
{
    bool isLeaf;
    char *URL;
    struct trieNode *child[CHARS];
};

// Function to create a new trie node.
struct trieNode *newTrieNode(void)
{
    struct trieNode *newNode = new trieNode;
    newNode->isLeaf = false;
    newNode->URL = NULL;
    for (int i=0; i<CHARS; i++)
        newNode->child[i] = NULL;
    return newNode;
}

// This method inserts an ip address and the corresponding
// domain name in the trie. The last node in Trie contains the URL.
void insert(struct trieNode *root, char *ipAdd, char *URL)
{
    // Length of the ip address
    int len = strlen(ipAdd);
    struct trieNode *pCrawl = root;

    // Traversing over the length of the ip address.
    for (int level=0; level<len; level++)
    {
        // Get index of child node from current character
        // in ipAdd[].  Index must be from 0 to 10 where
        // 0 to 9 is used for digits and 10 for dot
        int index = getIndex(ipAdd[level]);

        // Create a new child if not exist already
        if (!pCrawl->child[index])
            pCrawl->child[index] = newTrieNode();

        // Move to the child
        pCrawl = pCrawl->child[index];
    }

    //Below needs to be carried out for the last node.
    //Save the corresponding URL of the ip address in the
    //last node of trie.
    pCrawl->isLeaf = true;
    pCrawl->URL = new char[strlen(URL) + 1];
    strcpy(pCrawl->URL, URL);
}

// This function returns URL if given IP address is present in DNS cache.
// Else returns NULL
char  *searchDNSCache(struct trieNode *root, char *ipAdd)
{
    // Root node of trie.
```

```
    struct trieNode *pCrawl = root;
    int  len = strlen(ipAdd);

    // Traversal over the length of ip address.
    for (int level=0; level<len; level++)
    {
        int index = getIndex(ipAdd[level]);
        if (!pCrawl->child[index])
            return NULL;
        pCrawl = pCrawl->child[index];
    }

    // If we find the last node for a given ip address, print the URL.
    if (pCrawl!=NULL && pCrawl->isLeaf)
        return pCrawl->URL;

    return NULL;
}

//Driver function.
int main()
{
    /* Change third ipAddress for validation */
    char ipAdd[][MAX] = {"107.108.11.123", "107.109.123.255",
                         "74.125.200.106"};
    char URL[][50] = {"www.samsung.com", "www.samsung.net",
                      "www.google.in"};
    int n = sizeof(ipAdd)/sizeof(ipAdd[0]);
    struct trieNode *root = newTrieNode();

    // Inserts all the ip address and their corresponding
    // domain name after ip address validation.
    for (int i=0; i<n; i++)
        insert(root,ipAdd[i],URL[i]);

    // If reverse DNS look up succeeds print the domain
    // name along with DNS resolved.
    char ip[] = "107.108.11.123";
    char *res_url = searchDNSCache(root, ip);
    if (res_url != NULL)
        printf("Reverse DNS look up resolved in cache:\n%s --> %s",
                ip, res_url);
    else
        printf("Reverse DNS look up not resolved in cache ");
    return 0;
}
```

Output:

```
 Reverse DNS look up resolved in cache:
107.108.11.123 --> www.samsung.com
```

Note that the above implementation of Trie assumes that the given IP address does not contain characters

other than {‘0 , ‘1’,..... ‘9’, ‘.’}. What if a user gives an invalid IP address that contains some other characters? This problem can be resolved by validating the input IP address before inserting it into Trie. We can use the approach discussed herefor IP address validation.

This article is contributed by **Kumar Gautam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/implement-reverse-dns-look-cache/

Category: Trees Tags: Advance Data Structures, Advanced Data Structures

Chapter 11

How to Implement Forward DNS Look Up Cache?

We have discussed implementation of Reverse DNS Look Up Cache. Forward DNS look up is getting IP address for a given domain name typed in the web browser.

The cache should do the following operations :
1. Add a mapping from URL to IP address
2. Find IP address for a given URL.

There are a few changes from reverse DNS look up cache that we need to incorporate.
1. Instead of [0-9] and (.) dot we need to take care of [A-Z], [a-z] and (.) dot. As most of the domain name contains only lowercase characters we can assume that there will be [a-z] and (.) 27 children for each trie node.

2. When we type www.google.in and google.in the browser takes us to the same page. So, we need to add a domain name into trie for the words after www(.). Similarly while searching for a domain name corresponding IP address remove the www(.) if the user has provided it.

This is left as an exercise and for simplicity we have taken care of www. also.

One solution is to use Hashing. In this post, a Triebased solution is discussed. One advantage of Trie based solutions is, worst case upper bound is O(1) for Trie, for hashing, the best possible average case time complexity is O(1). Also, with Trie we can implement prefix search (finding all IPs for a common prefix of URLs). The general disadvantage of Trie is large amount of memory requirement.
The idea is to store URLs in Trie nodes and store the corresponding IP address in last or leaf node.

Following is C style implementation in C++.

```
// C based program to implement reverse DNS lookup
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

// There are atmost 27 different chars in a valid URL
// assuming URL consists [a-z] and (.)
#define CHARS 27

// Maximum length of a valid URL
#define MAX 100
```

```
// A utility function to find index of child for a given character 'c'
int getIndex(char c)
{
    return (c == '.') ? 26 : (c - 'a');
}

// A utility function to find character for a given child index.
char getCharFromIndex(int i)
{
    return (i == 26) ? '.' : ('a' + i);
}

// Trie Node.
struct trieNode
{
    bool isLeaf;
    char *ipAdd;
    struct trieNode *child[CHARS];
};

// Function to create a new trie node.
struct trieNode *newTrieNode(void)
{
    struct trieNode *newNode = new trieNode;
    newNode->isLeaf = false;
    newNode->ipAdd = NULL;
    for (int i = 0; i<CHARS; i++)
        newNode->child[i] = NULL;
    return newNode;
}

// This method inserts a URL and corresponding IP address
// in the trie. The last node in Trie contains the ip address.
void insert(struct trieNode *root, char *URL, char *ipAdd)
{
    // Length of the URL
    int len = strlen(URL);
    struct trieNode *pCrawl = root;

    // Traversing over the length of the URL.
    for (int level = 0; level<len; level++)
    {
        // Get index of child node from current character
        // in URL[]  Index must be from 0 to 26 where
        // 0 to 25 is used for alphabets and 26 for dot
        int index = getIndex(URL[level]);

        // Create a new child if not exist already
        if (!pCrawl->child[index])
            pCrawl->child[index] = newTrieNode();

        // Move to the child
        pCrawl = pCrawl->child[index];
```

```
    }

    //Below needs to be carried out for the last node.
    //Save the corresponding ip address of the URL in the
    //last node of trie.
    pCrawl->isLeaf = true;
    pCrawl->ipAdd = new char[strlen(ipAdd) + 1];
    strcpy(pCrawl->ipAdd, ipAdd);
}

// This function returns IP address if given URL is
// present in DNS cache. Else returns NULL
char  *searchDNSCache(struct trieNode *root, char *URL)
{
    // Root node of trie.
    struct trieNode *pCrawl = root;
    int  len = strlen(URL);

    // Traversal over the length of URL.
    for (int level = 0; level<len; level++)
    {
        int index = getIndex(URL[level]);
        if (!pCrawl->child[index])
            return NULL;
        pCrawl = pCrawl->child[index];
    }

    // If we find the last node for a given ip address,
    // print the ip address.
    if (pCrawl != NULL && pCrawl->isLeaf)
        return pCrawl->ipAdd;

    return NULL;
}

// Driver function.
int main()
{
    char URL[][50] = { "www.samsung.com", "www.samsung.net",
                       "www.google.in"
                     };
    char ipAdd[][MAX] = { "107.108.11.123", "107.109.123.255",
                          "74.125.200.106"
                        };
    int n = sizeof(URL) / sizeof(URL[0]);
    struct trieNode *root = newTrieNode();

    // Inserts all the domain name and their corresponding
    // ip address
    for (int i = 0; i<n; i++)
        insert(root, URL[i], ipAdd[i]);

    // If forward DNS look up succeeds print the url along
    // with the resolved ip address.
```

```
    char url[] = "www.samsung.com";
    char *res_ip = searchDNSCache(root, url);
    if (res_ip != NULL)
        printf("Forward DNS look up resolved in cache:\n%s --> %s",
               url, res_ip);
    else
        printf("Forward DNS look up not resolved in cache ");

    return 0;
}
```

Output:

```
Forward DNS look up resolved in cache:
www.samsung.com --> 107.108.11.123
```

This article is contributed by **Kumar Gautam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/implement-forward-dns-look-cache/

Category: Misc

Chapter 12

Suffix Array | Set 1 (Introduction)

We strongly recommend to read following post on suffix trees as a pre-requisite for this post.

Pattern Searching | Set 8 (Suffix Tree Introduction)

A suffix array is a sorted array of all suffixes of a given string. The definition is similar to Suffix Tree which is compressed trie of all suffixes of the given text. Any suffix tree based algorithm can be replaced with an algorithm that uses a suffix array enhanced with additional information and solves the same problem in the same time complexity (Source Wiki).
A suffix array can be constructed from Suffix tree by doing a DFS traversal of the suffix tree. In fact Suffix array and suffix tree both can be constructed from each other in linear time.
Advantages of suffix arrays over suffix trees include improved space requirements, simpler linear time construction algorithms (e.g., compared to Ukkonen's algorithm) and improved cache locality (Source: Wiki)

Example:

```
 Let the given string be "banana".

0 banana                          5 a
1 anana     Sort the Suffixes     3 ana
2 nana      ----------------->    1 anana
3 ana        alphabetically       0 banana
4 na                              4 na
5 a                               2 nana

So the suffix array for "banana" is {5, 3, 1, 0, 4, 2}
```

Naive method to build Suffix Array
A simple method to construct suffix array is to make an array of all suffixes and then sort the array. Following is implementation of simple method.

```
// Naive algorithm for building suffix array of a given text
#include <iostream>
#include <cstring>
#include <algorithm>
using namespace std;

// Structure to store information of a suffix
```

```
struct suffix
{
    int index;
    char *suff;
};

// A comparison function used by sort() to compare two suffixes
int cmp(struct suffix a, struct suffix b)
{
    return strcmp(a.suff, b.suff) < 0? 1 : 0;
}

// This is the main function that takes a string 'txt' of size n as an
// argument, builds and return the suffix array for the given string
int *buildSuffixArray(char *txt, int n)
{
    // A structure to store suffixes and their indexes
    struct suffix suffixes[n];

    // Store suffixes and their indexes in an array of structures.
    // The structure is needed to sort the suffixes alphabatically
    // and maintain their old indexes while sorting
    for (int i = 0; i < n; i++)
    {
        suffixes[i].index = i;
        suffixes[i].suff = (txt+i);
    }

    // Sort the suffixes using the comparison function
    // defined above.
    sort(suffixes, suffixes+n, cmp);

    // Store indexes of all sorted suffixes in the suffix array
    int *suffixArr = new int[n];
    for (int i = 0; i < n; i++)
        suffixArr[i] = suffixes[i].index;

    // Return the suffix array
    return  suffixArr;
}

// A utility function to print an array of given size
void printArr(int arr[], int n)
{
    for(int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver program to test above functions
int main()
{
    char txt[] = "banana";
    int n = strlen(txt);
```

```
    int *suffixArr = buildSuffixArray(txt,  n);
    cout << "Following is suffix array for " << txt << endl;
    printArr(suffixArr, n);
    return 0;
}
```

Output:

```
 Following is suffix array for banana
5 3 1 0 4 2
```

The time complexity of above method to build suffix array is $O(n^2Logn)$ if we consider a O(nLogn) algorithm used for sorting. The sorting step itself takes $O(n^2Logn)$ time as every comparison is a comparison of two strings and the comparison takes O(n) time.
There are many efficient algorithms to build suffix array. We will soon be covering them as separate posts.

Search a pattern using the built Suffix Array
To search a pattern in a text, we preprocess the text and build a suffix array of the text. Since we have a sorted array of all suffixes, Binary Search can be used to search. Following is the search function. Note that the function doesn't report all occurrences of pattern, it only report one of them.

```
// This code only contains search() and main. To make it a complete running
// above code or see http://ideone.com/1Io9eN

// A suffix array based search function to search a given pattern
// 'pat' in given text 'txt' using suffix array suffArr[]
void search(char *pat, char *txt, int *suffArr, int n)
{
    int m = strlen(pat);  // get length of pattern, needed for strncmp()

    // Do simple binary search for the pat in txt using the
    // built suffix array
    int l = 0, r = n-1;  // Initilize left and right indexes
    while (l <= r)
    {
        // See if 'pat' is prefix of middle suffix in suffix array
        int mid = l + (r - l)/2;
        int res = strncmp(pat, txt+suffArr[mid], m);

        // If match found at the middle, print it and return
        if (res == 0)
        {
            cout << "Pattern found at index " << suffArr[mid];
            return;
        }

        // Move to left half if pattern is alphabtically less than
        // the mid suffix
        if (res < 0) r = mid - 1;

        // Otherwise move to right half
        else l = mid + 1;
```

```
    }

    // We reach here if return statement in loop is not executed
    cout << "Pattern not found";
}

// Driver program to test above function
int main()
{
    char txt[] = "banana";  // text
    char pat[] = "nan";   // pattern to be searched in text

    // Build suffix array
    int n = strlen(txt);
    int *suffArr = buildSuffixArray(txt, n);

    // search pat in txt using the built suffix array
    search(pat, txt, suffArr, n);

    return 0;
}
```

Output:

```
Pattern found at index 2
```

The time complexity of the above search function is O(mLogn). There are more efficient algorithms to search pattern once the suffix array is built. In fact there is a O(m) suffix array based algorithm to search a pattern. We will soon be discussing efficient algorithm for search.

Applications of Suffix Array
Suffix array is an extremely useful data structure, it can be used for a wide range of problems. Following are some famous problems where Suffix array can be used.
1) Pattern Searching
2) Finding the longest repeated substring
3) Finding the longest common substring
4) Finding the longest palindrome in a string

See this for more problems where Suffix arrays can be used.

This post is a simple introduction. There is a lot to cover in Suffix arrays. We have discussed a O(nLogn) algorithm for Suffix Array construction here. We will soon be discussing more efficient suffix array algorithms.

References:
http://www.stanford.edu/class/cs97si/suffix-array.pdf
http://en.wikipedia.org/wiki/Suffix_array

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/suffix-array-set-1-introduction/

Chapter 13

Suffix Array | Set 2 (nLogn Algorithm)

A suffix array is a sorted array of all suffixes of a given string. The definition is similar to Suffix Tree which is compressed trie of all suffixes of the given text.

```
Let the given string be "banana".

0 banana                          5 a
1 anana     Sort the Suffixes     3 ana
2 nana      ---------------->     1 anana
3 ana        alphabetically       0 banana
4 na                              4 na
5 a                               2 nana

The suffix array for "banana" is {5, 3, 1, 0, 4, 2}
```

We have discussed **Naive algorithm** for construction of suffix array. The Naive algorithm is to consider all suffixes, sort them using a O(nLogn) sorting algorithm and while sorting, maintain original indexes. Time complexity of the Naive algorithm is $O(n^2Logn)$ where n is the number of characters in the input string.

In this post, a **O(nLogn) algorithm** for suffix array construction is discussed. Let us first discuss a O(n * Logn * Logn) algorithm for simplicity. The idea is to use the fact that strings that are to be sorted are suffixes of a single string.
We first sort all suffixes according to first character, then according to first 2 characters, then first 4 characters and so on while the number of characters to be considered is smaller than 2n. The important point is, if we have sorted suffixes according to first 2^i characters, then we can sort suffixes according to first 2^{i+1} characters in O(nLogn) time using a nLogn sorting algorithm like Merge Sort. This is possible as two suffixes can be compared in O(1) time (we need to compare only two values, see the below example and code).
The sort function is called O(Logn) times (Note that we increase number of characters to be considered in powers of 2). Therefore overall time complexity becomes O(nLognLogn). See http://www.stanford.edu/class/cs97si/suffix-array.pdf for more details.

Let us build suffix array the example string “banana” using above algorithm.

Sort according to first two characters Assign a rank to all suffixes using ASCII value of first character. A simple way to assign rank is to do “str[i] – ‘a’ ” for ith suffix of strp[]

Index	Suffix	Rank
0	banana	1
1	anana	0
2	nana	13
3	ana	0
4	na	13
5	a	0

For every character, we also store rank of next adjacent character, i.e., the rank of character at str[i + 1] (This is needed to sort the suffixes according to first 2 characters). If a character is last character, we store next rank as -1

Index	Suffix	Rank	Next Rank
0	banana	1	0
1	anana	0	13
2	nana	13	0
3	ana	0	13
4	na	13	0
5	a	0	-1

Sort all Suffixes according to rank and adjacent rank. Rank is considered as first digit or MSD, and adjacent rank is considered as second digit.

Index	Suffix	Rank	Next Rank
5	a	0	-1
1	anana	0	13
3	ana	0	13
0	banana	1	0
2	nana	13	0
4	na	13	0

Sort according to first four character
Assign new ranks to all suffixes. To assign new ranks, we consider the sorted suffixes one by one. Assign 0 as new rank to first suffix. For assigning ranks to remaining suffixes, we consider rank pair of suffix just before the current suffix. If previous rank pair of a suffix is same as previous rank of suffix just before it, then assign it same rank. Otherwise assign rank of previous suffix plus one.

Index	Suffix	Rank	
5	a	0	[Assign 0 to first]
1	anana	1	(0, 13) is different from previous
3	ana	1	(0, 13) is same as previous
0	banana	2	(1, 0) is different from previous
2	nana	3	(13, 0) is different from previous
4	na	3	(13, 0) is same as previous

For every suffix str[i], also store rank of next suffix at str[i + 2]. If there is no next suffix at i + 2, we store next rank as -1

```
Index       Suffix          Rank        Next Rank
  5          a               0             -1
  1          anana           1              1
  3          ana             1              0
  0          banana          2              3
  2          nana            3              3
  4          na              3              -1
```

Sort all Suffixes according to rank and next rank.

```
Index       Suffix          Rank        Next Rank
  5          a               0             -1
  3          ana             1              0
  1          anana           1              1
  0          banana          2              3
  4          na              3             -1
  2          nana            3              3
```

```
// C++ program for building suffix array of a given text
#include <iostream>
#include <cstring>
#include <algorithm>
using namespace std;

// Structure to store information of a suffix
struct suffix
{
    int index; // To store original index
    int rank[2]; // To store ranks and next rank pair
};

// A comparison function used by sort() to compare two suffixes
// Compares two pairs, returns 1 if first pair is smaller
int cmp(struct suffix a, struct suffix b)
{
    return (a.rank[0] == b.rank[0])? (a.rank[1] < b.rank[1] ?1: 0):
               (a.rank[0] < b.rank[0] ?1: 0);
}

// This is the main function that takes a string 'txt' of size n as an
// argument, builds and return the suffix array for the given string
int *buildSuffixArray(char *txt, int n)
{
    // A structure to store suffixes and their indexes
    struct suffix suffixes[n];

    // Store suffixes and their indexes in an array of structures.
    // The structure is needed to sort the suffixes alphabatically
    // and maintain their old indexes while sorting
    for (int i = 0; i < n; i++)
```

```
{
    suffixes[i].index = i;
    suffixes[i].rank[0] = txt[i] - 'a';
    suffixes[i].rank[1] = ((i+1) < n)? (txt[i + 1] - 'a'): -1;
}

// Sort the suffixes using the comparison function
// defined above.
sort(suffixes, suffixes+n, cmp);

// At his point, all suffixes are sorted according to first
// 2 characters.  Let us sort suffixes according to first 4
// characters, then first 8 and so on
int ind[n];  // This array is needed to get the index in suffixes[]
             // from original index.  This mapping is needed to get
             // next suffix.
for (int k = 4; k < 2*n; k = k*2)
{
    // Assigning rank and index values to first suffix
    int rank = 0;
    int prev_rank = suffixes[0].rank[0];
    suffixes[0].rank[0] = rank;
    ind[suffixes[0].index] = 0;

    // Assigning rank to suffixes
    for (int i = 1; i < n; i++)
    {
        // If first rank and next ranks are same as that of previous
        // suffix in array, assign the same new rank to this suffix
        if (suffixes[i].rank[0] == prev_rank &&
                suffixes[i].rank[1] == suffixes[i-1].rank[1])
        {
            prev_rank = suffixes[i].rank[0];
            suffixes[i].rank[0] = rank;
        }
        else // Otherwise increment rank and assign
        {
            prev_rank = suffixes[i].rank[0];
            suffixes[i].rank[0] = ++rank;
        }
        ind[suffixes[i].index] = i;
    }

    // Assign next rank to every suffix
    for (int i = 0; i < n; i++)
    {
        int nextindex = suffixes[i].index + k/2;
        suffixes[i].rank[1] = (nextindex < n)?
                              suffixes[ind[nextindex]].rank[0]: -1;
    }

    // Sort the suffixes according to first k characters
    sort(suffixes, suffixes+n, cmp);
}
```

```
    // Store indexes of all sorted suffixes in the suffix array
    int *suffixArr = new int[n];
    for (int i = 0; i < n; i++)
        suffixArr[i] = suffixes[i].index;

    // Return the suffix array
    return  suffixArr;
}

// A utility function to print an array of given size
void printArr(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver program to test above functions
int main()
{
    char txt[] = "banana";
    int n = strlen(txt);
    int *suffixArr = buildSuffixArray(txt,  n);
    cout << "Following is suffix array for " << txt << endl;
    printArr(suffixArr, n);
    return 0;
}
```

Output:

```
 Following is suffix array for banana
5 3 1 0 4 2
```

Note that the above algorithm uses standard sort function and therefore time complexity is O(nLognLogn). We can use Radix Sort here to reduce the time complexity to O(nLogn).

Please note that suffx arrays can be constructed in O(n) time also. We will soon be discussing O(n) algorithms.

References:
http://www.stanford.edu/class/cs97si/suffix-array.pdf
http://www.cbcb.umd.edu/confcour/Fall2012/lec14b.pdf

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/suffix-array-set-2-a-nlognlogn-algorithm/

Chapter 14

Pattern Searching | Set 8 (Suffix Tree Introduction)

Given a text txt[0..n-1] and a pattern pat[0..m-1], write a function search(char pat[], char txt[]) that prints all occurrences of pat[] in txt[]. You may assume that n > m.

Preprocess Pattern or Preoprocess Text?
We have discussed the following algorithms in the previous posts:

KMP Algorithm
Rabin Karp Algorithm
Finite Automata based Algorithm

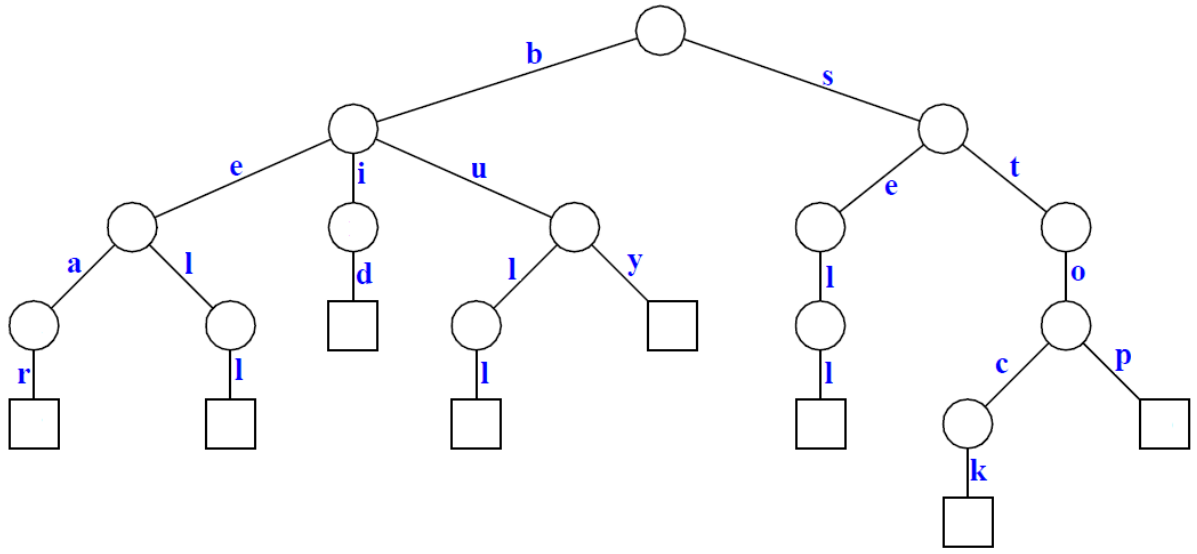
Boyer Moore Algorithm

All of the above algorithms preprocess the pattern to make the pattern searching faster. The best time complexity that we could get by preprocessing pattern is O(n) where n is length of the text. In this post, we will discuss an approach that preprocesses the text. A suffix tree is built of the text. After preprocessing text (building suffix tree of text), we can search any pattern in O(m) time where m is length of the pattern. Imagine you have stored complete work of William Shakespeare and preprocessed it. You can search any string in the complete work in time just proportional to length of the pattern. This is really a great improvement because length of pattern is generally much smaller than text.
Preprocessing of text may become costly if the text changes frequently. It is good for fixed text or less frequently changing text though.

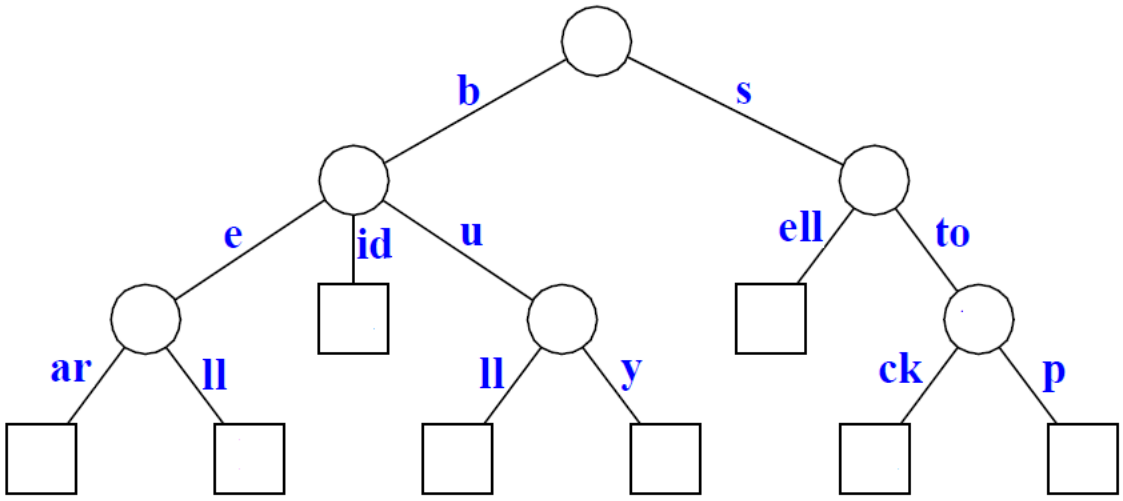
A Suffix Tree for a given text is a compressed trie for all suffixes of the given text. We have discussed Standard Trie. Let us understand **Compressed Trie** with the following array of words.

```
{bear, bell, bid, bull, buy, sell, stock, stop}
```

Following is standard trie for the above input set of words.



Following is the compressed trie. Compress Trie is obtained from standard trie by joining chains of single nodes. The nodes of a compressed trie can be stored by storing index ranges at the nodes.

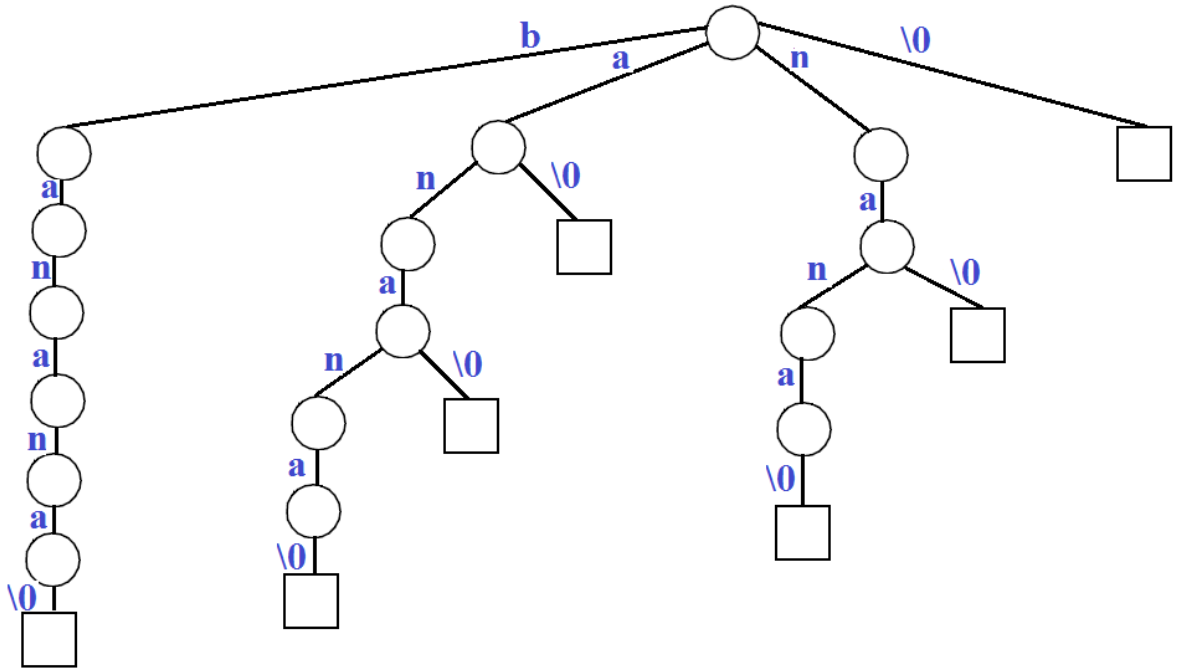


How to build a Suffix Tree for a given text?
As discussed above, Suffix Tree is compressed trie of all suffixes, so following are very abstract steps to build a suffix tree from given text.
1) Generate all suffixes of given text.
2) Consider all suffixes as individual words and build a compressed trie.

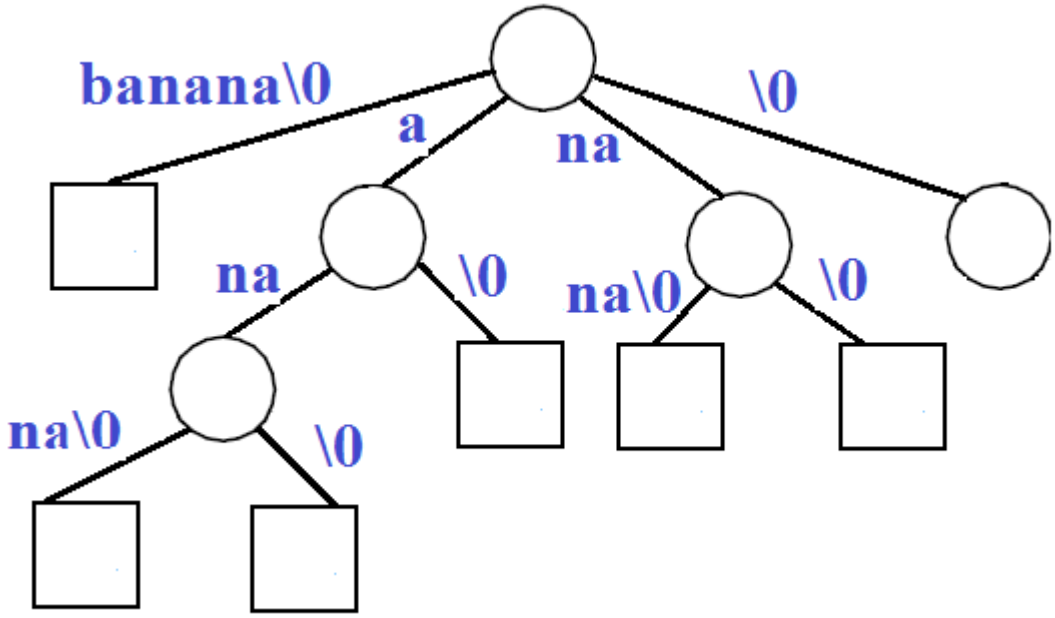
Let us consider an example text "banana\0 where '\0 is string termination character. Following are all suffixes of "banana\0

```
 banana\0
anana\0
nana\0
ana\0
na\0
a\0
\0
```

If we consider all of the above suffixes as individual words and build a trie, we get following.



If we join chains of single nodes, we get the following compressed trie, which is the Suffix Tree for given text "banana\0

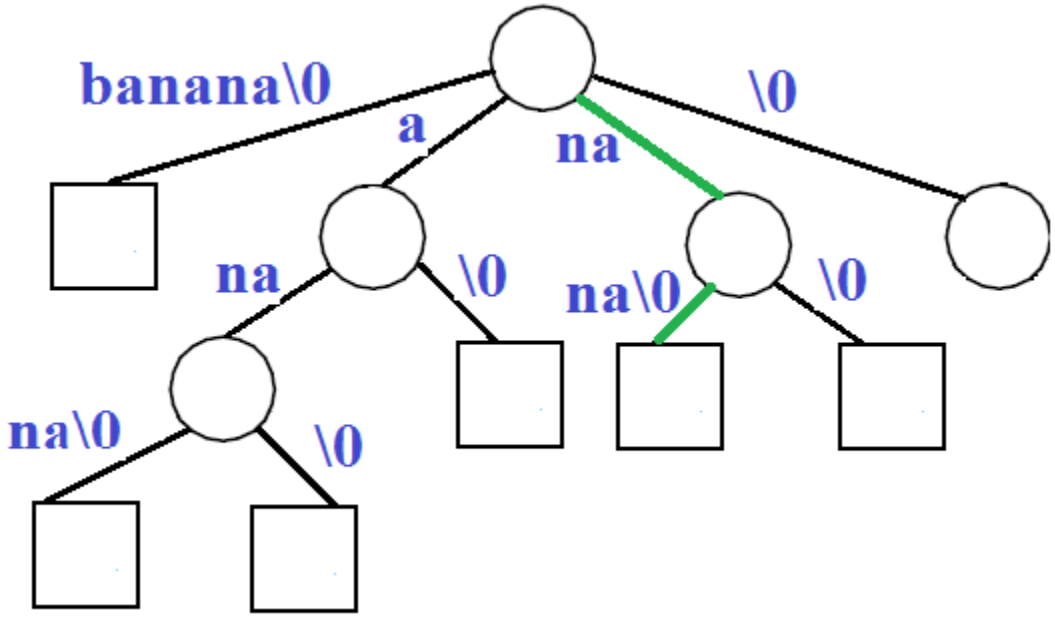


Please note that above steps are just to manually create a Suffix Tree. We will be discussing actual algorithm and implementation in a separate post.

How to search a pattern in the built suffix tree?
We have discussed above how to build a Suffix Tree which is needed as a preprocessing step in pattern searching. Following are abstract steps to search a pattern in the built Suffix Tree.

1) Starting from the first character of the pattern and root of Suffix Tree, do following for every character.
…..**a)** For the current character of pattern, if there is an edge from the current node of suffix tree, follow the edge.
…..**b)** If there is no edge, print "pattern doesn't exist in text" and return.
2) If all characters of pattern have been processed, i.e., there is a path from root for characters of the given pattern, then print "Pattern found".

Let us consider the example pattern as "nan" to see the searching process. Following diagram shows the path followed for searching "nan" or "nana".



How does this work?
Every pattern that is present in text (or we can say every substring of text) must be a prefix of one of all possible suffixes. The statement seems complicated, but it is a simple statement, we just need to take an example to check validity of it.

Applications of Suffix Tree
Suffix tree can be used for a wide range of problems. Following are some famous problems where Suffix Trees provide optimal time complexity solution.
1) Pattern Searching
2) Finding the longest repeated substring
3) Finding the longest common substring
4) Finding the longest palindrome in a string

There are many more applications. See thisfor more details.

Ukkonen's Suffix Tree Construction is discussed in following articles:

Ukkonen's Suffix Tree Construction – Part 1

Ukkonen's Suffix Tree Construction – Part 2

Ukkonen's Suffix Tree Construction – Part 3

Ukkonen's Suffix Tree Construction – Part 4

Ukkonen's Suffix Tree Construction – Part 5

Ukkonen's Suffix Tree Construction – Part 6

References:

http://fbim.fh-regensburg.de/~saj39122/sal/skript/progr/pr45102/Tries.pdf
http://www.cs.ucf.edu/~shzhang/Combio12/lec3.pdf
http://www.allisons.org/ll/AlgDS/Tree/Suffix/

Source

http://www.geeksforgeeks.org/pattern-searching-set-8-suffix-tree-introduction/

Chapter 15

Ukkonen's Suffix Tree Construction - Part 1

Suffix Tree is very useful in numerous string processing and computational biology problems. Many books and e-resources talk about it theoretically and in few places, code implementation is discussed. But still, I felt something is missing and it's not easy to implement code to construct suffix tree and it's usage in many applications. This is an attempt to bridge the gap between theory and complete working code implementation. Here we will discuss Ukkonen's Suffix Tree Construction Algorithm. We will discuss it in step by step detailed way and in multiple parts from theory to implementation. We will start with brute force way and try to understand different concepts, tricks involved in Ukkonen's algorithm and in the last part, code implementation will be discussed.
Note: You may find some portion of the algorithm difficult to understand while 1st or 2nd reading and it's perfectly fine. With few more attempts and thought, you should be able to understand such portions.

Book Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology by **Dan Gusfield** explains the concepts very well.

A suffix tree **T** for a m-character string S is a rooted directed tree with exactly m leaves numbered 1 to **m.** (Given that last string character is unique in string)

- Root can have zero, one or more children.
- Each internal node, other than the root, has at least two children.
- Each edge is labelled with a nonempty substring of S.
- No two edges coming out of same node can have edge-labels beginning with the same character.

Concatenation of the edge-labels on the path from the root to leaf i gives the suffix of S that starts at position i, i.e. S[i…m].

Note: Position starts with 1 (it's not zero indexed, but later, while code implementation, we will used zero indexed position)

For string S = xabxac with m = 6, suffix tree will look like following:

It has one root node and two internal nodes and 6 leaf nodes.

String Depth of red path is 1 and it represents suffix c starting at position 6
String Depth of blue path is 4 and it represents suffix bxca starting at position 3

String Depth of green path is 2 and it represents suffix ac starting at position 5
String Depth of orange path is 6 and it represents suffix xabxac starting at position 1

Edges with labels a (green) and xa (orange) are non-leaf edge (which ends at an internal node). All other edges are leaf edge (ends at a leaf)

If one suffix of S matches a prefix of another suffix of S (when last character in not unique in string), then path for the first suffix would not end at a leaf.

For String S = xabxa, with m = 5, following is the suffix tree:

Here we will have 5 suffixes: xabxa, abxa, bxa, xa and a.
Path for suffixes 'xa' and 'a' do not end at a leaf. A tree like above (Figure 2) is called implicit suffix tree as some suffixes ('xa' and 'a') are not seen explicitly in tree.

To avoid this problem, we add a character which is not present in string already. We normally use $, # etc as termination characters.
Following is the suffix tree for string S = xabxa$ with m = 6 and now all 6 suffixes end at leaf.

A naive algorithm to build a suffix tree
Given a string S of length m, enter a single edge for suffix S[l ..m]$ (the entire string) into the tree, then successively enter suffix S[i..m]$ into the growing tree, for i increasing from 2 to m. Let N_i denote the intermediate tree that encodes all the suffixes from 1 to i.
So N_i+1 is constructed from N_i as follows:

- Start at the root of N_i
- Find the longest path from the root which matches a prefix of S[i+1..m]$
- Match ends either at the node (say w) or in the middle of an edge [say (u, v)].
- If it is in the middle of an edge (u, v), break the edge (u, v) into two edges by inserting a new node w just after the last character on the edge that matched a character in S[i+l..m] and just before the first character on the edge that mismatched. The new edge (u, w) is labelled with the part of the (u, v) label that matched with S[i+1..m], and the new edge (w, v) is labelled with the remaining part of the (u, v) label.
- Create a new edge (w, i+1) from w to a new leaf labelled i+1 and it labels the new edge with the unmatched part of suffix S[i+1..m]

This takes $O(m^2)$ to build the suffix tree for the string S of length m.
Following are few steps to build suffix tree based for string "xabxa$" based on above algorithm:

Implicit suffix tree
While generating suffix tree using Ukkonen's algorithm, we will see implicit suffix tree in intermediate steps few times depending on characters in string S. In implicit suffix trees, there will be no edge with $ (or # or any other termination character) label and no internal node with only one edge going out of it.
To get implicit suffix tree from a suffix tree S$,

- Remove all terminal symbol $ from the edge labels of the tree,
- Remove any edge that has no label
- Remove any node that has only one edge going out of it and merge the edges.

High Level Description of Ukkonen's algorithm
Ukkonen's algorithm constructs an implicit suffix tree T_i for each prefix S[l ..i] of S (of length m).
It first builds T_1 using 1^{st} character, then T_2 using 2^{nd} character, then T_3 using 3^{rd} character, …, T_m using m^{th} character.
Implicit suffix tree T_i+1 is built on top of implicit suffix tree T_i.
The true suffix tree for S is built from T_m by adding $.
At any time, Ukkonen's algorithm builds the suffix tree for the characters seen so far and so it has **on-line** property that may be useful in some situations.
Time taken is O(m).

Ukkonen's algorithm is divided into m phases (one phase for each character in the string with length m)
In phase i+1, tree T_i+1 is built from tree T_i.

Each phase i+1 is further divided into i+1 extensions, one for each of the i+1 suffixes of S[1..i+1]
In extension j of phase i+1, the algorithm first finds the end of the path from the root labelled with substring S[j..i].
It then extends the substring by adding the character S(i+1) to its end (if it is not there already).
In extension 1 of phase i+1, we put string S[1..i+1] in the tree. Here S[1..i] will already be present in tree due to previous phase i. We just need to add S[i+1]th character in tree (if not there already).
In extension 2 of phase i+1, we put string S[2..i+1] in the tree. Here S[2..i] will already be present in tree due to previous phase i. We just need to add S[i+1]th character in tree (if not there already)
In extension 3 of phase i+1, we put string S[3..i+1] in the tree. Here S[3..i] will already be present in tree due to previous phase i. We just need to add S[i+1]th character in tree (if not there already)
.
.
In extension i+1 of phase i+1, we put string S[i+1..i+1] in the tree. This is just one character which may not be in tree (if character is seen first time so far). If so, we just add a new leaf edge with label S[i+1].

High Level Ukkonen's algorithm
Construct tree T_1
For i from 1 to m-1 do
begin {phase i+1}
For j from 1 to i+1
begin {extension j}
Find the end of the path from the root labelled S[j..i] in the current tree.
Extend that path by adding character S[i+l] if it is not there already
end;
end;

Suffix extension is all about adding the next character into the suffix tree built so far.
In extension j of phase i+1, algorithm finds the end of S[j..i] (which is already in the tree due to previous phase i) and then it extends S[j..i] to be sure the suffix S[j..i+1] is in the tree.

There are 3 extension rules:
Rule 1: If the path from the root labelled S[j..i] ends at leaf edge (i.e. S[i] is last character on leaf edge) then character S[i+1] is just added to the end of the label on that leaf edge.

Rule 2: If the path from the root labelled S[j..i] ends at non-leaf edge (i.e. there are more characters after S[i] on path) and next character is not s[i+1], then a new leaf edge with label s{i+1] and number j is created starting from character S[i+1].
A new internal node will also be created if s[1..i] ends inside (in-between) a non-leaf edge.

Rule 3: If the path from the root labelled S[j..i] ends at non-leaf edge (i.e. there are more characters after S[i] on path) and next character is s[i+1] (already in tree), do nothing.

One important point to note here is that from a given node (root or internal), there will be one and only one edge starting from one character. There will not be more than one edges going out of any node, starting with same character.

Following is a step by step suffix tree construction of string xabxac using Ukkonen's algorithm:

In next parts (Part 2, Part 3, Part 4 and Part 5), we will discuss suffix links, active points, few tricks and finally code implementations (Part 6).

References:
http://web.stanford.edu/~mjkay/gusfield.pdf

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-1/

Chapter 16

Ukkonen's Suffix Tree Construction - Part 2

In Ukkonen's Suffix Tree Construction – Part 1, we have seen high level Ukkonen's Algorithm. This 2^{nd} part is continuation of Part 1.
Please go through Part 1, before looking at current article.

In Suffix Tree Construction of string S of length m, there are m phases and for a phase j (1 <= j <= m), we add j^{th} character in tree built so far and this is done through j extensions. All extensions follow one of the three extension rules (discussed in Part 1).

To do j^{th} extension of phase i+1 (adding character S[i+1]), we first need to find end of the path from the root labelled S[j..i] in the current tree. One way is start from root and traverse the edges matching S[j..i] string. This will take $O(m^3)$ time to build the suffix tree. Using few observations and implementation tricks, it can be done in O(m) which we will see now.

Suffix links
For an internal node v with path-label xA, where x denotes a single character and A denotes a (possibly empty) substring, if there is another node s(v) with path-label A, then a pointer from v to s(v) is called a suffix link.
If A is empty string, suffix link from internal node will go to root node.
There will not be any suffix link from root node (As it's not considered as internal node).

In extension j of some phase i, if a new internal node v with path-label xA is added, then in extension j+1 in the same phase i:

- Either the path labelled A already ends at an internal node (or root node if A is empty)
- OR a new internal node at the end of string A will be created

In extension j+1 of same phase i, we will create a suffix link from the internal node created in j^{th} extension to the node with path labelled A.

So in a given phase, any newly created internal node (with path-label xA) will have a suffix link from it (pointing to another node with path-label A) by the end of the next extension.

In any implicit suffix tree T_i after phase i, if internal node v has path-label xA, then there is a node s(v) in T_i with path-label A and node v will point to node s(v) using suffix link.

At any time, all internal nodes in the changing tree will have suffix links from them to another internal node (or root) except for the most recently added internal node, which will receive its suffix link by the end of the next extension.

How suffix links are used to speed up the implementation?
In extension j of phase i+1, we need to find the end of the path from the root labelled S[j..i] in the current tree. One way is start from root and traverse the edges matching S[j..i] string. Suffix links provide a short cut to find end of the path.

So we can see that, to find end of path S[j..i], we need not traverse from root. We can start from the end of path S[j-1..i], walk up one edge to node v (i.e. go to parent node), follow the suffix link to s(v), then walk down the path y (which is abcd here in Figure 17).
This shows the use of suffix link is an improvement over the process.
Note: In the next part 3, we will introduce activePoint which will help to avoid "walk up". We can directly go to node s(v) from node v.

When there is a suffix link from node v to node s(v), then if there is a path labelled with string y from node v to a leaf, then there must be a path labelled with string y from node s(v) to a leaf. In Figure 17, there is a path label "abcd" from node v to a leaf, then there is a path will same label "abcd" from node s(v) to a leaf.
This fact can be used to improve the walk from s(v) to leaf along the path y. This is called "skip/count" trick.

Skip/Count Trick
When walking down from node s(v) to leaf, instead of matching path character by character as we travel, we can directly skip to the next node if number of characters on the edge is less than the number of characters we need to travel. If number of characters on the edge is more than the number of characters we need to travel, we directly skip to the last character on that edge.
If implementation is such a way that number of characters on any edge, character at a given position in string S should be obtained in constant time, then skip/count trick will do the walk down in proportional to the number of nodes on it rather than the number of characters on it.

Using suffix link along with skip/count trick, suffix tree can be built in $O(m^2)$ as there are m phases and each phase takes O(m).

Edge-label compression
So far, path labels are represented as characters in string. Such a suffix tree will take $O(m^2)$ space to store the path labels. To avoid this, we can use two pair of indices (start, end) on each edge for path labels, instead of substring itself. The indices start and end tells the path label start and end position in string S. With this, suffix tree needs O(m) space.

There are two observations about the way extension rules interact in successive extensions and phases. These two observations lead to two more implementation tricks (first trick "skip/count" is seen already while walk down).

Observation 1: Rule 3 is show stopper
In a phase i, there are i extensions (1 to i) to be done.

When rule 3 applies in any extension j of phase i+1 (i.e. path labelled S[j..i] continues with character S[i+1]), then it will also apply in all further extensions of same phase (i.e. extensions j+1 to i+1 in phase i+1). That's because if path labelled S[j..i] continues with character S[i+1], then path labelled S[j+1..i], S[j+2..i], S[j+3..i],..., S[i..i] will also continue with character S[i+1].
Consider Figure 11, Figure12 and Figure 13 in Part 1 where Rule 3 is applied.
In Figure 11, "xab" is added in tree and in Figure 12 (Phase 4), we add next character "x". In this, 3 extensions are done (which adds 3 suffixes). Last suffix "x" is already present in tree.
In Figure 13, we add character "a" in tree (Phase 5). First 3 suffixes are added in tree and last two suffixes "xa" and "a" are already present in tree. This shows that if suffix S[j..i] present in tree, then ALL the remaining suffixes S[j+1..i], S[j+2..i], S[j+3..i],..., S[i..i] will also be there in tree and no work needed to add those remaining suffixes.
So no more work needed to be done in any phase as soon as rule 3 applies in any extension in that phase. If a new internal node v gets created in extension j and rule 3 applies in next extension j+1, then we need to add suffix link from node v to current node (if we are on internal node) or root node. ActiveNode, which will be discussed in part 3, will help while setting suffix links.

Trick 2
Stop the processing of any phase as soon as rule 3 applies. All further extensions are already present in tree implicitly.

Observation 2: Once a leaf, always a leaf
Once a leaf is created and labelled j (for suffix starting at position j in string S), then this leaf will always be a leaf in successive phases and extensions. Once a leaf is labelled as j, extension rule 1 will always apply to extension j in all successive phases.
Consider Figure 9 to Figure 14 in Part 1.
In Figure 10 (Phase 2), Rule 1 is applied on leaf labelled 1. After this, in all successive phases, rule 1 is always applied on this leaf.
In Figure 11 (Phase 3), Rule 1 is applied on leaf labelled 2. After this, in all successive phases, rule 1 is always applied on this leaf.
In Figure 12 (Phase 4), Rule 1 is applied on leaf labelled 3. After this, in all successive phases, rule 1 is always applied on this leaf.

In any phase i, there is an initial sequence of consecutive extensions where rule 1 or rule 2 are applied and then as soon as rule 3 is applied, phase i ends.
Also rule 2 creates a new leaf always (and internal node sometimes).
If J_i represents the last extension in phase i when rule 1 or 2 was applied (i.e after i^{th} phase, there will be J_i leaves labelled 1, 2, 3, ..., J_i) , then $J_i <= J_{i+1}$
J_i will be equal to J_{i+1} when there are no new leaf created in phase i+1 (i.e rule 3 is applied in J_{i+1} extension)
In Figure 11 (Phase 3), Rule 1 is applied in 1st two extensions and Rule 2 is applied in 3rd extension, so here $J_3 = 3$
In Figure 12 (Phase 4), no new leaf created (Rule 1 is applied in 1st 3 extensions and then rule 3 is applied in 4th extension which ends the phase). Here $J_4 = 3 = J_3$
In Figure 13 (Phase 5), no new leaf created (Rule 1 is applied in 1st 3 extensions and then rule 3 is applied in 4th extension which ends the phase). Here $J_5 = 3 = J_4$
J_i will be less than J_{i+1} when few new leaves are created in phase i+1.
In Figure 14 (Phase 6), new leaf created (Rule 1 is applied in 1st 3 extensions and then rule 2 is applied in last 3 extension which ends the phase). Here $J_6 = 6 > J_5$

So we can see that in phase i+1, only rule 1 will apply in extensions 1 to J_i (which really doesn't need much work, can be done in constant time and that's the trick 3), extension J_{i+1} onwards, rule 2 may apply to zero or more extensions and then finally rule 3, which ends the phase.
Now edge labels are represented using two indices (start, end), for any leaf edge, end will always be equal to phase number i.e. for phase i, end = i for leaf edges, for phase i+1, end = i+1 for leaf edges.

Trick 3
In any phase i, leaf edges may look like (p, i), (q, i), (r, i), ... where p, q, r are starting position of different edges and i is end position of all. Then in phase i+1, these leaf edges will look like (p, i+1), (q, i+1), (r,

i+1),…. This way, in each phase, end position has to be incremented in all leaf edges. For this, we need to traverse through all leaf edges and increment end position for them. To do same thing in constant time, maintain a global index e and e will be equal to phase number. So now leaf edges will look like (p, e), (q, e), (r, e).. In any phase, just increment e and extension on all leaf edges will be done. Figure 19 shows this.

So using suffix links and tricks 1, 2 and 3, a suffix tree can be built in linear time.

Tree Tm could be implicit tree if a suffix is prefix of another. So we can add a $ terminal symbol first and then run algorithm to get a true suffix tree (A true suffix tree contains all suffixes explicitly). To label each leaf with corresponding suffix starting position (all leaves are labelled as global index e), a linear time traversal can be done on tree.

At this point, we have gone through most of the things we needed to know to create suffix tree using Ukkonen's algorithm. In next Part 3, we will take string S = "abcabxabcd" as an example and go through all the things step by step and create the tree. While building the tree, we will discuss few more implementation issues which will be addressed by ActivePoints.
We will continue to discuss the algorithm in Part 4 and Part 5. Code implementation will be discussed in Part 6.

References:
http://web.stanford.edu/~mjkay/gusfield.pdf

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-2/

Chapter 17

Ukkonen's Suffix Tree Construction - Part 3

This article is continuation of following two articles:

Ukkonen's Suffix Tree Construction – Part 1

Ukkonen's Suffix Tree Construction – Part 2

Please go through Part 1 and Part 2, before looking at current article, where we have seen few basics on suffix tree, high level ukkonen's algorithm, suffix link and three implementation tricks.

Here we will take string S = "abcabxabcd" as an example and go through all the things step by step and create the tree.
We will add $ (discussed in Part 1 why we do this) so string S would be "abcabxabcd$".

While building suffix tree for string S of length m:

- There will be m phases 1 to m (one phase for each character)
  In our current example, m is 11, so there will be 11 phases.
- First phase will add first character 'a' in the tree, second phase will add second character 'b' in tree, third phase will add third character 'c' in tree, ……, m^{th} phase will add m^{th} character in tree (This makes Ukkonen's algorithm an online algorithm)
- Each phase i will go through at-most i extensions (from 1 to i). If current character being added in tree is not seen so far, all i extensions will be completed (Extension Rule 3 will not apply in this phase). If current character being added in tree is seen before, then phase i will complete early (as soon as Extension Rule 3 applies) without going through all i extensions
- There are three extension rules (1, 2 and 3) and each extension j (from 1 to i) of any phase i will adhere to one of these three rules.
- Rule 1 adds a new character on existing leaf edge
- Rule 2 creates a new leaf edge (And may also create new internal node, if the path label ends in between an edge)
- Rule 3 ends the current phase (when current character is found in current edge being traversed)

- Phase 1 will read first character from the string, will go through 1 extension.
  (In figures, we are showing characters on edge labels just for explanation, while writing code, we will only use start and end indices – The Edge-label compression discussed in Part 2)
  Extension 1 will add suffix "a" in tree. We start from root and traverse path with label 'a'. There is no path from root, going out with label 'a', so create a leaf edge (Rule 2).

  Phase 1 completes with the completion of extension 1 (As a phase i has at most i extensions)
  For any string, Phase 1 will have only one extension and it will always follow Rule 2.

- Phase 2 will read second character, will go through at least 1 and at most 2 extensions.
  In our example, phase 2 will read second character 'b'. Suffixes to be added are "ab" and "b".
  Extension 1 adds suffix "ab" in tree.
  Path for label 'a' ends at leaf edge, so add 'b' at the end of this edge.
  Extension 1 just increments the end index by 1 (from 1 to 2) on first edge (Rule 1).

  Extension 2 adds suffix "b" in tree. There is no path from root, going out with label 'b', so creates a leaf edge (Rule 2).

  Phase 2 completes with the completion of extension 2.
  Phase 2 went through two extensions here. Rule 1 applied in 1st Extension and Rule 2 applied in 2nd Extension.

- Phase 3 will read third character, will go through at least 1 and at most 3 extensions.
  In our example, phase 3 will read third character 'c'. Suffixes to be added are "abc", "bc" and "c".
  Extension 1 adds suffix "abc" in tree.
  Path for label 'ab' ends at leaf edge, so add 'c' at the end of this edge.
  Extension 1 just increments the end index by 1 (from 2 to 3) on this edge (Rule 1).

  Extension 2 adds suffix "bc" in tree.
  Path for label 'b' ends at leaf edge, so add 'c' at the end of this edge.
  Extension 2 just increments the end index by 1 (from 2 to 3) on this edge (Rule 1).

  Extension 3 adds suffix "c" in tree. There is no path from root, going out with label 'c', so creates a leaf edge (Rule 2).

  Phase 3 completes with the completion of extension 3.
  Phase 3 went through three extensions here. Rule 1 applied in first two Extensions and Rule 2 applied in 3rd Extension.

- Phase 4 will read fourth character, will go to at least 1 and at most 4 extensions.
  In our example, phase 4 will read fourth character 'a'. Suffixes to be added are "abca", "bca", "ca" and "a".
  Extension 1 adds suffix "abca" in tree.
  Path for label 'abc' ends at leaf edge, so add 'a' at the end of this edge.
  Extension 1 just increments the end index by 1 (from 3 to 4) on this edge (Rule 1).

  Extension 2 adds suffix "bca" in tree.
  Path for label 'bc' ends at leaf edge, so add 'a' at the end of this edge.
  Extension 2 just increments the end index by 1 (from 3 to 4) on this edge (Rule 1).

  Extension 3 adds suffix "ca" in tree.
  Path for label 'c' ends at leaf edge, so add 'a' at the end of this edge.
  Extension 3 just increments the end index by 1 (from 3 to 4) on this edge (Rule 1).

  Extension 4 adds suffix "a" in tree.
  Path for label 'a' exists in the tree. No more work needed and Phase 4 ends here (Rule 3 and Trick 2).
  This is an example of implicit suffix tree. Here suffix "a" is not seen explicitly (because it doesn't end at a leaf edge) but it is in the tree implicitly. So there is no change in tree structure after extension 4. It will remain as above in Figure 28.
  Phase 4 completes as soon as Rule 3 is applied while Extension 4.
  Phase 4 went through four extensions here. Rule 1 applied in first three Extensions and Rule 3 applied in 4th Extension.

Now we will see few observations and how to implement those.

1. At the end of any phase i, there are at most i leaf edges (if i^{th} character is not seen so far, there will be i leaf edges, else there will be less than i leaf edges).
   e.g. After phases 1, 2 and 3 in our example, there are 1, 2 and 3 leaf edges respectively, but after phase 4, there are 3 leaf edges only (not 4).

2. After completing phase i, "end" indices of all leaf edges are i. How do we implement this in code? Do we need to iterate through all those extensions, find leaf edges by traversing from root to leaf and increment the "end" index? Answer is "NO".
   For this, we will maintain a global variable (say "END") and we will just increment this global variable "END" and all leaf edge end indices will point to this global variable. So this way, if we have j leaf edges after phase i, then in phase i+1, first j extensions (1 to j) will be done by just incrementing variable "END" by 1 (END will be i+1 at the point).
   Here we just implemented the trick 3 – **Once a leaf, always a leaf**. This trick processes all the j leaf edges (i.e. extension 1 to j) using rule 1 in a constant time in any phase. Rule 1 will not apply to subsequent extensions in the same phase. This can be verified in the four phases we discussed above. If at all Rule 1 applies in any phase, it only applies in initial few phases continuously (say 1 to j). Rule 1 never applies later in a given phase once Rule 2 or Rule 3 is applied in that phase.

3. In the example explained so far, in each extension (where trick 3 is not applied) of any phase to add a suffix in tree, we are traversing from root by matching path labels against the suffix being added. If there are j leaf edges after phase i, then in phase i+1, first j extensions will follow Rule 1 and will be done in constant time using trick 3. There are i+1-j extensions yet to be performed. For these

extensions, which node (root or some other internal node) to start from and which path to go? Answer to this depends on how previous phase i is completed.
If previous phase i went through all the i extensions (when i^{th} character is unique so far), then in next phase i+1, trick 3 will take care of first i suffixes (the i leaf edges) and then extension i+1 will start from root node and it will insert just one character [$(i+1)^{th}$] suffix in tree by creating a leaf edge using Rule 2.
If previous phase i completes early (and this will happen if and only if rule 3 applies – when i^{th} character is already seen before), say at j^{th} extension (i.e. rule 3 is applied at j^{th} extension), then there are j-1 leaf edges so far.
We will state few more facts (which may be a repeat, but we want to make sure it's clear to you at this point) here based on discussion so far:

- *Phase 1 starts with Rule 2, all other phases start with Rule 1*
- *Any phase ends with either Rule 2 or Rule 3*
- *Any phase i may go through a series of j extensions (1 <= j <= i). In these j extensions, first p (0 <= p < i) extensions will follow Rule 1, next q (0 <= q <= i-p) extensions will follow Rule 2 and next r (0<= r <= 1) extensions will follow Rule 3. The order in which Rule 1, Rule 2 and Rule 3 apply, is never intermixed in a phase. They apply in order of their number (if at all applied), i.e. in a phase, Rule 1 applies 1st, then Rule 2 and then Rule 3*
- *In a phase i, p + q + r <= i*
- *At the end of any phase i, there will be p+q leaf edges and next phase i+1 will go through Rule 1 for first p+q extensions*

In the next phase i+1, trick 3 (Rule 1) will take care of first j-1 suffixes (the j-1 leaf edges), then extension j will start where we will add j^{th} suffix in tree. For this, we need to find the best possible matching edge and then add new character at the end of that edge. How to find the end of best matching edge? Do we need to traverse from root node and match tree edges against the j^{th} suffix being added character by character? This will take time and overall algorithm will not be linear. activePoint comes to the rescue here.
In previous phase i, while j^{th} extension, path traversal ended at a point (which could be an internal node or some point in the middle of an edge) where i^{th} character being added was found in tree already and Rule 3 applied, j^{th} extension of phase i+1 will start exactly from the same point and we start matching path against $(i+1)^{th}$ character. activePoint helps to avoid unnecessary path traversal from root in any extension based on the knowledge gained in traversals done in previous extension. There is no traversal needed in 1^{st} p extensions where Rule 1 is applied. Traversal is done where Rule 2 or Rule 3 gets applied and that's where activePoint tells the starting point for traversal where we match the path against the current character being added in tree. Implementation is done in such a way that, in any extension where we need a traversal, activePoint is set to right location already (with one exception case **APCFALZ** discussed below) and at the end of current extension, we reset activePoint as apprppriate so that next extension (of same phase or next phase) where a traversal is required, activePoint points to the right place already.

activePoint: This could be root node, any internal node or any point in the middle of an edge. This is the point where traversal starts in any extension. For the 1st extension of phase 1, activePoint is set to root. Other extension will get activePoint set correctly by previous extension (with one exception case **APCFALZ** discussed below) and it is the responsibility of current extension to reset activePoint appropriately at the end, to be used in next extension where Rule 2 or Rule 3 is applied (of same or next phase).
To accomplish this, we need a way to store activePoint. We will store this using three variables: **activeNode**, **activeEdge**, **activeLength**.
activeNode: This could be root node or an internal node.
activeEdge: When we are on root node or internal node and we need to walk down, we need to know which edge to choose. activeEdge will store that information. In case, activeNode itself is the point from where traversal starts, then activeEdge will be set to next character being processed in next phase.

activeLength: This tells how many characters we need to walk down (on the path represented by activeEdge) from activeNode to reach the activePoint where traversal starts. In case, activeNode itself is the point from where traversal starts, then activeLength will be ZERO.
(click on below image to see it clearly)

After phase i, if there are j leaf edges then in phase i+1, first j extensions will be done by trick 3. activePoint will be needed for the extensions from j+1 to i+1 and activePoint may or may not change between two extensions depending on the point where previous extension ends.

activePoint change for extension rule 3 (APCFER3): When rule 3 applies in any phase i, then before we move on to next phase i+1, we increment activeLength by 1. There is no change in activeNode and activeEdge. Why? Because in case of rule 3, the current character from string S is matched on the same path represented by current activePoint, so for next activePoint, activeNode and activeEdge remain the same, only activeLenth is increased by 1 (because of matched character in current phase). This new activePoint (same node, same edge and incremented length) will be used in phase i+1.

activePoint change for walk down (APCFWD): activePoint may change at the end of an extension based on extension rule applied. activePoint may also change during the extension when we do walk down. Let's consider an activePoint is (A, s, 11) in the above activePoint example figure. If this is the activePoint at the start of some extension, then while walk down from activeNode A, other internal nodes will be seen. Anytime if we encounter an internal node while walk down, that node will become activeNode (it will change activeEdge and activeLenght as appropriate so that new activePoint represents the same point as earlier). In this walk down, below is the sequence of changes in activePoint:
(A, s, 11) — >>> (B, w, 7) —- >>> (C, a, 3)
All above three activePoints refer to same point 'c'
Let's take another example.
If activePoint is (D, a, 11) at the start of an extension, then while walk down, below is the sequence of changes in activePoint:
(D, a, 10) — >>> (E, d, 7) — >>> (F, f, 5) — >> (G, j, 1)
All above activePoints refer to same point 'k'.
If activePoints are (A, s, 3), (A, t, 5), (B, w, 1), (D, a, 2) etc when no internal node comes in the way while walk down, then there will be no change in activePoint for APCFWD.
The idea is that, at any time, the closest internal node from the point, where we want to reach, should be the activePoint. Why? This will minimize the length of traversal in the next extension.

activePoint change for Active Length ZERO (APCFALZ): Let's consider an activePoint (A, s, 0) in the above activePoint example figure. And let's say current character being processed from string S is 'x' (or any other character). At the start of extension, when activeLength is ZERO, activeEdge is set to the current character being processed, i.e. 'x', because there is no walk down needed here (as activeLength is ZERO) and so next character we look for is current character being processed.

4. While code implementation, we will loop through all the characters of string S one by one. Each loop for i^{th} character will do processing for phase i. Loop will run one or more time depending on how many extensions are left to be performed (Please note that in a phase i+1, we don't really have to perform all i+1 extensions explicitly, as trick 3 will take care of j extensions for all j leaf edges coming from previous phase i). We will use a variable **remainingSuffixCount**, to track how many extensions are yet to be performed explicitly in any phase (after trick 3 is performed). Also, at the end of any phase, if remainingSuffixCount is ZERO, this tells that all suffixes supposed to be added in tree, are added explicitly and present in tree. If remainingSuffixCount is non-zero at the end of any phase, that tells that suffixes of that many count are not added in tree explicitly (because of rule 3, we stopped early), but they are in tree implicitly though (Such trees are called implicit suffix tree). These implicit suffixes will be added explicitly in subsequent phases when a unique character comes in the way.

We will continue our discussion in Part 4 and Part 5. Code implementation will be discussed in Part 6.

References:
http://web.stanford.edu/~mjkay/gusfield.pdf

Ukkonen's suffix tree algorithm in plain English

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-3/

Category: Strings Tags: Pattern Searching

Post navigation

← Convert a Binary Tree to Threaded binary tree Amazon interview Experience | Set 141 (For SDE1) →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 18

Ukkonen's Suffix Tree Construction - Part 4

This article is continuation of following three articles:

Ukkonen's Suffix Tree Construction – Part 1

Ukkonen's Suffix Tree Construction – Part 2

Ukkonen's Suffix Tree Construction – Part 3

Please go through Part 1, Part 2 and Part 3, before looking at current article, where we have seen few basics on suffix tree, high level ukkonen's algorithm, suffix link and three implementation tricks and some details on activePoint along with an example string "abcabxabcd" where we went through four phases of building suffix tree.

Let's revisit those four phases we have seen already in Part 3, in terms of trick 2, trick 3 and activePoint.

- activePoint is initialized to (root, NULL, 0), i.e. activeNode is root, activeEdge is NULL (for easy understanding, we are giving character value to activeEdge, but in code implemntation, it will be index of the character) and activeLength is ZERO.
- The global variable END and remainingSuffixCount are initialized to ZERO

**********************Phase 1******************************
In Phase 1, we read 1st character (a) from string S

- Set END to 1
- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is 1 extension left to be performed)
- Run a loop remainingSuffixCount times (i.e. one time) as below:
  - If activeLength is ZERO, set activeEdge to the current character (here activeEdge will be 'a'). This is **APCFALZ**.
  - Check if there is an edge going out from activeNode (which is root in this phase 1) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, leaf edge gets created (Rule 2).
  - Once extension is performed, decrement the remainingSuffixCount by 1
  - At this point, activePoint is (root, a, 0)

At the end of phase 1, remainingSuffixCount is ZERO (All suffixes are added explicitly).
Figure 20 in Part 3 is the resulting tree after phase 1.

**********Phase 2**********

In Phase 2, we read 2nd character (b) from string S

Set END to 2 (This will do extension 1)

Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is 1 extension left to be performed)

Run a loop remainingSuffixCount times (i.e. one time) as below:

- If activeLength is ZERO, set activeEdge to the current character (here activeEdge will be 'b'). This is **APCFALZ**.
- Check if there is an edge going out from activeNode (which is root in this phase 2) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, leaf edge gets created.
- Once extension is performed, decrement the remainingSuffixCount by 1
- At this point, activePoint is (root, b, 0)

At the end of phase 2, remainingSuffixCount is ZERO (All suffixes are added explicitly).
Figure 22 in Part 3 is the resulting tree after phase 2.

**********Phase 3**********

In Phase 3, we read 3rd character (c) from string S

Set END to 3 (This will do extensions 1 and 2)

Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is 1 extension left to be performed)

Run a loop remainingSuffixCount times (i.e. one time) as below:

- If activeLength is ZERO, set activeEdge to the current character (here activeEdge will be 'c'). This is **APCFALZ**.
- Check if there is an edge going out from activeNode (which is root in this phase 3) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, leaf edge gets created.
- Once extension is performed, decrement the remainingSuffixCount by 1
- At this point, activePoint is (root, c, 0)

At the end of phase 3, remainingSuffixCount is ZERO (All suffixes are added explicitly).
Figure 25 in Part 3 is the resulting tree after phase 3.

**********Phase 4**********

In Phase 4, we read 4th character (a) from string S

Set END to 4 (This will do extensions 1, 2 and 3)

Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is 1 extension left to be performed)

Run a loop remainingSuffixCount times (i.e. one time) as below:

- If activeLength is ZERO, set activeEdge to the current character (here activeEdge will be 'a'). This is **APCFALZ**.
- Check if there is an edge going out from activeNode (which is root in this phase 3) for the activeEdge. If not, create a leaf edge. If present, walk down (The trick 1 – skip/count). In our example, edge 'a' is present going out of activeNode (i.e. root). No walk down needed as activeLength < edgeLength. We increment activeLength from zero to 1 (**APCFER3**) and stop any further processing (Rule 3).

- At this point, activePoint is (root, a, 1) and remainingSuffixCount remains set to 1 (no change there)

At the end of phase 4, remainingSuffixCount is 1 (One suffix 'a', the last one, is not added explicitly in tree, but it is there in tree implicitly).
Figure 28 in Part 3 is the resulting tree after phase 4.

Revisiting completed for 1st four phases, we will continue building the tree and see how it goes.

**********************Phase 5********************************
In phase 5, we read 5th character (b) from string S

Set END to 5 (This will do extensions 1, 2 and 3). See Figure 29 shown below.

Increment remainingSuffixCount by 1 (remainingSuffixCount will be 2 here, i.e. there are 2 extension left to be performed, which are extensions 4 and 5. Extension 4 is supposed to add suffix "ab" and extension 5 is supposed to add suffix "b" in tree)

Run a loop remainingSuffixCount times (i.e. two times) as below:

- Check if there is an edge going out from activeNode (which is root in this phase 3) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, edge 'a' is present going out of activeNode (i.e. root).
- Do a walk down (The trick 1 – skip/count) if necessary. In current phase 5, no walk down needed as activeLength < edgeLength. Here activePoint is (root, a, 1) for extension 4 (remainingSuffixCount = 2)
- Check if current character of string S (which is 'b') is already present after the activePoint. If yes, no more processing (rule 3). Same is the case in our example, so we increment activeLength from 1 to 2 (**APCFER3**) and we stop here (Rule 3).
- At this point, activePoint is (root, a, 2) and remainingSuffixCount remains set to 2 (no change in remainingSuffixCount)

At the end of phase 5, remainingSuffixCount is 2 (Two suffixes, 'ab' and 'b', the last two, are not added explicitly in tree, but they are in tree implicitly).

**********************Phase 6********************************
In phase 6, we read 6th character (x) from string S

Set END to 6 (This will do extensions 1, 2 and 3)

Increment remainingSuffixCount by 1 (remainingSuffixCount will be 3 here, i.e. there are 3 extension left to be performed, which are extensions 4, 5 and 6 for suffixes "abx", "bx" and "x" respectively)

Run a loop remainingSuffixCount times (i.e. three times) as below:

- While extension 4, the activePoint is (root, a, 2) which points to 'b' on edge starting with 'a'.
- In extension 4, current character 'x' from string S doesn't match with the next character on the edge after activePoint, so this is the case of extension rule 2. So a leaf edge is created here with edge label x. Also here traversal ends in middle of an edge, so a new internal node also gets created at the end of activePoint.
- Decrement the remainingSuffixCount by 1 (from 3 to 2) as suffix "abx" added in tree.

Now activePoint will change after applying rule 2. Three other cases, (**APCFER3**, **APCFWD** and **APCFALZ**) where activePoint changes, are already discussed in Part 3.

activePoint change for extension rule 2 (APCFER2):
Case 1 (APCFER2C1): If activeNode is root and activeLength is greater than ZERO, then decrement the activeLength by 1 and activeEdge will be set "S[i – remainingSuffixCount + 1]" where i is current phase number. Can you see why this change in activePoint? Look at current extension we just discussed above for phase 6 (i=6) again where we added suffix "abx". There activeLength is 2 and activeEdge is 'a'. Now in next extension, we need to add suffix "bx" in the tree, i.e. path label in next extension should start with 'b'. So 'b' (the 5^{th} character in string S) should be active edge for next extension and index of b will be "i – remainingSuffixCount + 1" (6 – 2 + 1 = 5). activeLength is decremented by 1 because activePoint gets closer to root by length 1 after every extension.
What will happen If activeNode is root and activeLength is ZERO? This case is already taken care by **APCFALZ**.

Case 2 (APCFER2C2): If activeNode is not root, then follow the suffix link from current activeNode. The new node (which can be root node or another internal node) pointed by suffix link will be the activeNode for next extension. No change in activeLength and activeEdge. Can you see why this change in activePoint? This is because: If two nodes are connected by a suffix link, then labels on all paths going down from those two nodes, starting with same character, will be exactly same and so for two corresponding similar point on those paths, activeEdge and activeLength will be same and the two nodes will be the activeNode. Look at Figure 18 in Part 2. Let's say in phase i and extension j, suffix 'xAabcdedg' was added in tree. At that point, let's say activePoint was (Node-V, a, 7), i.e. point 'g'. So for next extension j+1, we would add suffix 'Aabcdefg' and for that we need to traverse 2^{nd} path shown in Figure 18. This can be done by following suffix link from current activeNode v. Suffix link takes us to the path to be traversed somewhere in between [Node s(v)] below which the path is exactly same as how it was below the previous activeNode v. As said earlier, "activePoint gets closer to root by length 1 after every extension", this reduction in length will happen above the node s(v) but below s(v), no change at all. So when activeNode is not root in current extension, then for next extension, only activeNode changes (No change in activeEdge and activeLength).

- At this point in extension 4, current activePoint is (root, a, 2) and based on **APCFER2C1**, new activePoint for next extension 5 will be (root, b, 1)
- Next suffix to be added is 'bx' (with remainingSuffixCount 2).
- Current character 'x' from string S doesn't match with the next character on the edge after activePoint, so this is the case of extension rule 2. So a leaf edge is created here with edge label x. Also here traversal ends in middle of an edge, so a new internal node also gets created at the end of activePoint.
  Suffix link is also created from previous internal node (of extension 4) to the new internal node created in current extension 5.
- Decrement the remainingSuffixCount by 1 (from 2 to 1) as suffix "bx" added in tree.

- At this point in extension 5, current activePoint is (root, b, 1) and based on **APCFER2C1** new activePoint for next extension 6 will be (root, x, 0)
- Next suffix to be added is 'x' (with remainingSuffixCount 1).
- In the next extension 6, character x will not match to any existing edge from root, so a new edge with label x will be created from root node. Also suffix link from previous extension's internal node goes to root (as no new internal node created in current extension 6).
- Decrement the remainingSuffixCount by 1 (from 1 to 0) as suffix "x" added in tree

This completes the phase 6.
Note that phase 6 has completed all its 6 extensions (Why? Because the current character c was not seen in string so far, so rule 3, which stops further extensions never got chance to get applied in phase 6) and so the tree generated after phase 6 is a true suffix tree (i.e. not an implicit tree) for the characters 'abcabx' read so far and it has all suffixes explicitly in the tree.

While building the tree above, following facts were noticed:

- A newly created internal node in extension i, points to another internal node or root (if activeNode is root in extension i+1) by the end of extension i+1 via suffix link (Every internal node MUST have a suffix link pointing to another internal node or root)
- Suffix link provides short cut while searching path label end of next suffix
- With proper tracking of activePoints between extensions/phases, unnecessary walkdown from root can be avoided.

We will go through rest of the phases (7 to 11) in Part 5 and build the tree completely and after that, we will see the code for the algorithm in Part 6.

References:
http://web.stanford.edu/~mjkay/gusfield.pdf

Ukkonen's suffix tree algorithm in plain English

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-4/

Category: Strings Tags: Pattern Searching

Post navigation

← K Dimensional Tree | Set 1 (Search and Insert) SAP Labs India | Set 2 (On Campus Interview) →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 19

Ukkonen's Suffix Tree Construction - Part 5

This article is continuation of following four articles:

Ukkonen's Suffix Tree Construction – Part 1

Ukkonen's Suffix Tree Construction – Part 2

Ukkonen's Suffix Tree Construction – Part 3

Ukkonen's Suffix Tree Construction – Part 4

Please go through Part 1, Part 2, Part 3 and Part 4, before looking at current article, where we have seen few basics on suffix tree, high level ukkonen's algorithm, suffix link and three implementation tricks and some details on activePoint along with an example string "abcabxabcd" where we went through six phases of building suffix tree.
Here, we will go through rest of the phases (7 to 11) and build the tree completely.

**********************Phase 7*******************************
In phase 7, we read 7^{th} character (a) from string S

- Set END to 7 (This will do extensions 1, 2, 3, 4, 5 and 6) – because we have 6 leaf edges so far by the end of previous phase 6.

Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is only 1 extension left to be performed, which is extensions 7 for suffix 'a')

Run a loop remainingSuffixCount times (i.e. one time) as below:

- If activeLength is ZERO [activePoint in previous phase was (root, x, 0)], set activeEdge to the current character (here activeEdge will be 'a'). This is **APCFALZ**. Now activePoint becomes (root, 'a', 0).
- Check if there is an edge going out from activeNode (which is root in this phase 7) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, edge 'a' is present going out of activeNode (i.e. root), here we increment activeLength from zero to 1 (**APCFER3**) and stop any further processing.

- At this point, activePoint is (root, a, 1) and remainingSuffixCount remains set to 1 (no change there)

At the end of phase 7, remainingSuffixCount is 1 (One suffix 'a', the last one, is not added explicitly in tree, but it is there in tree implicitly).
Above Figure 33 is the resulting tree after phase 7.

**********************Phase 8********************************
In phase 8, we read 8^{th} character (b) from string S

- Set END to 8 (This will do extensions 1, 2, 3, 4, 5 and 6) – because we have 6 leaf edges so far by the end of previous phase 7 (Figure 34).

Increment remainingSuffixCount by 1 (remainingSuffixCount will be 2 here, i.e. there are two extensions left to be performed, which are extensions 7 and 8 for suffixes 'ab' and 'b' respectively)

Run a loop remainingSuffixCount times (i.e. two times) as below:

- Check if there is an edge going out from activeNode (which is root in this phase 8) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, edge 'a' is present going out of activeNode (i.e. root).
- Do a walk down (The trick 1 – skip/count) if necessary. In current phase 8, no walk down needed as activeLength < edgeLength. Here activePoint is (root, a, 1) for extension 7 (remainingSuffixCount = 2)
- Check if current character of string S (which is 'b') is already present after the activePoint. If yes, no more processing (rule 3). Same is the case in our example, so we increment activeLength from 1 to 2 (**APCFER3**) and we stop here (Rule 3).
- At this point, activePoint is (root, a, 2) and remainingSuffixCount remains set to 2 (no change in remainingSuffixCount)

At the end of phase 8, remainingSuffixCount is 2 (Two suffixes, 'ab' and 'b', the last two, are not added explicitly in tree explicitly, but they are in tree implicitly).

**********************Phase 9********************************
In phase 9, we read 9^{th} character (c) from string S

- Set END to 9 (This will do extensions 1, 2, 3, 4, 5 and 6) – because we have 6 leaf edges so far by the end of previous phase 8.

Increment remainingSuffixCount by 1 (remainingSuffixCount will be 3 here, i.e. there are three extensions left to be performed, which are extensions 7, 8 and 9 for suffixes 'abc', 'bc' and 'c' respectively)

Run a loop remainingSuffixCount times (i.e. three times) as below:

- Check if there is an edge going out from activeNode (which is root in this phase 9) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, edge 'a' is present going out of activeNode (i.e. root).
- Do a walk down (The trick 1 – skip/count) if necessary. In current phase 9,walk down needed as activeLength(2) >= edgeLength(2). While walk down, activePoint changes to (Node A, c, 0) based on **APCFWD** (This is first time **APCFWD** is being applied in our example).

- Check if current character of string S (which is 'c') is already present after the activePoint. If yes, no more processing (rule 3). Same is the case in our example, so we increment activeLength from 0 to 1 (**APCFER3**) and we stop here (Rule 3).
- At this point, activePoint is (Node A, c, 1) and remainingSuffixCount remains set to 3 (no change in remainingSuffixCount)

At the end of phase 9, remainingSuffixCount is 3 (Three suffixes, 'abc', 'bc' and 'c', the last three, are not added explicitly in tree explicitly, but they are in tree implicitly).

**********************Phase 10********************************

In phase 10, we read 10^{th} character (d) from string S

- Set END to 10 (This will do extensions 1, 2, 3, 4, 5 and 6) – because we have 6 leaf edges so far by the end of previous phase 9.

Increment remainingSuffixCount by 1 (remainingSuffixCount will be 4 here, i.e. there are four extensions left to be performed, which are extensions 7, 8, 9 and 10 for suffixes 'abcd', 'bcd', 'cd' and 'd' respectively)

Run a loop remainingSuffixCount times (i.e. four times) as below:

- Check if there is an edge going out from activeNode (Node A) for the activeEdge(c). If not, create a leaf edge. If present, walk down. In our example, edge 'c' is present going out of activeNode (Node A).
- Do a walk down (The trick 1 – skip/count) if necessary. In current Extension 7, no walk down needed as activeLength < edgeLength.
- Check if current character of string S (which is 'd') is already present after the activePoint. If not, rule 2 will apply. In our example, there is no path starting with 'd' going out of activePoint, so we create a leaf edge with label 'd'. Since activePoint ends in the middle of an edge, we will create a new internal node just after the activePoint (Rule 2)

- Decrement the remainingSuffixCount by 1 (from 4 to 3) as suffix "abcd" added in tree.
- Now activePoint will change for next extension 8. Current activeNode is an internal node (Node A), so there must be a suffix link from there and we will follow that to get new activeNode and that's going to be 'Node B'. There is no change in activeEdge and activeLength (This is **APCFER2C2**). So new activePoint is (Node B, c, 1).
- Now in extension 8 (here we will add suffix 'bcd'), while adding character 'd' after the current activePoint, exactly same logic will apply as previous extension 7. In previous extension 7, we added character 'd' at activePoint (Node A, c, 1) and in current extension 8, we are going to add same character 'd' at activePoint (Node B c, 1). So logic will be same and here we a new leaf edge with label 'd' and a new internal node will be created. And the new internal node (C) of previous extension will point to the new node (D) of current extension via suffix link.

- Decrement the remainingSuffixCount by 1 (from 3 to 2) as suffix "bcd" added in tree.
- Now activePoint will change for next extension 9. Current activeNode is an internal node (Node B), so there must be a suffix link from there and we will follow that to get new activeNode and that is 'Root Node'. There is no change in activeEdge and activeLength (This is **APCFER2C2**). So new activePoint is (root, c, 1).

- Now in extension 9 (here we will add suffix 'cd'), while adding character 'd' after the current activePoint, exactly same logic will apply as previous extensions 7 and 8. Note that internal node D created in previous extension 8, now points to internal node E (created in current extension) via suffix link.

- Decrement the remainingSuffixCount by 1 (from 2 to 1) as suffix "cd" added in tree.
- Now activePoint will change for next extension 10. Current activeNode is root and activeLength is 1, based on **APCFER2C1**, activeNode will remain 'root', activeLength will be decremented by 1 (from 1 to ZERO) and activeEdge will be 'd'. So new activePoint is (root, d, 0).
- Now in extension 10 (here we will add suffix 'd'), while adding character 'd' after the current activePoint, there is no edge starting with d going out of activeNode root, so a new leaf edge with label d is created (Rule 2). Note that internal node E created in previous extension 9, now points to root node via suffix link (as no new internal node created in this extension).

- Decrement the remainingSuffixCount by 1 (from 1 to 0) as suffix "d" added in tree. That means no more suffix is there to add and so the phase 10 ends here. Note that this tree is an explicit tree as all suffixes are added in tree explicitly (Why ?? because character d was not seen before in string S so far)
- activePoint for next phase 11 is (root, d, 0).

We see following facts in Phase 10:

- Internal Nodes connected via suffix links have exactly same tree below them, e.g. In above Figure 40, A and B have same tree below them, similarly C, D and E have same tree below them.
- Due to above fact, in any extension, when current activeNode is derived via suffix link from previous extension's activeNode, then exactly same extension logic apply in current extension as previous extension. (In Phase 10, same extension logic is applied in extensions 7, 8 and 9)
- If a new internal node gets created in extension j of any phase i, then this newly created internal node will get it's suffix link set by the end of next extension j+1 of same phase i. e.g. node C got created in extension 7 of phase 10 (Figure 37) and it got it's suffix link set to node D in extension 8 of same phase 10 (Figure 38). Similarly node D got created in extension 8 of phase 10 (Figure 38) and it got its suffix link set to node E in extension 9 of same phase 10 (Figure 39). Similarly node E got created in extension 9 of phase 10 (Figure 39) and it got its suffix link set to root in extension 10 of same phase 10 (Figure 40).
- Based on above fact, every internal node will have a suffix link to some other internal node or root. Root is not an internal node and it will not have suffix link.

**********************Phase 11********************************
In phase 11, we read 11^{th} character ($) from string S

- Set END to 11 (This will do extensions 1 to 10) – because we have 10 leaf edges so far by the end of previous phase 10.

- Increment remainingSuffixCount by 1 (from 0 to 1), i.e. there is only one suffix '$' to be added in tree.
- Since activeLength is ZERO, activeEdge will change to current character '$' of string S being processed (**APCFALZ**).

- There is no edge going out from activeNode root, so a leaf edge with label '$' will be created (Rule 2).

- Decrement the remainingSuffixCount by 1 (from 1 to 0) as suffix "$" added in tree. That means no more suffix is there to add and so the phase 11 ends here. Note that this tree is an explicit tree as all suffixes are added in tree explicitly (Why ?? because character $ was not seen before in string S so far)

Now we have added all suffixes of string 'abcabxabcd$' in suffix tree. There are 11 leaf ends in this tree and labels on the path from root to leaf end represents one suffix. Now the only one thing left is to assign a number (suffix index) to each leaf end and that number would be the suffix starting position in the string S. This can be done by a DFS traversal on tree. While DFS traversal, keep track of label length and when a leaf end is found, set the suffix index as "stringSize – labelSize + 1". Indexed suffix tree will look like below:

In above Figure, suffix indices are shown as character position starting with 1 (It's not zero indexed). In code implementation, suffix index will be set as zero indexed, i.e. where we see suffix index j (1 to m for string of length m) in above figure, in code implementation, it will be j-1 (0 to m-1)
And we are done !!!!

Data Structure to represent suffix tree
How to represent the suffix tree?? There are nodes, edges, labels and suffix links and indices.
Below are some of the operations/query we will be doing while building suffix tree and later on while using the suffix tree in different applications/usages:

- What length of path label on some edge?
- What is the path label on some edge?
- Check if there is an outgoing edge for a given character from a node.
- What is the character value on an edge at some given distance from a node?
- Where an internal node is pointing via suffix link?
- What is the suffix index on a path from root to leaf?
- Check if a given string present in suffix tree (as substring, suffix or prefix)?

We may think of different data structures which can fulfil these requirements.
In the next Part 6, we will discuss the data structure we will use in our code implementation and the code as well.

References:
http://web.stanford.edu/~mjkay/gusfield.pdf

Ukkonen's suffix tree algorithm in plain English

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-5/

Category: Strings Tags: Pattern Searching

Post navigation

← Binomial Heap Ukkonen's Suffix Tree Construction – Part 6 →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 20

Ukkonen's Suffix Tree Construction - Part 6

This article is continuation of following five articles:

Ukkonen's Suffix Tree Construction – Part 1

Ukkonen's Suffix Tree Construction – Part 2

Ukkonen's Suffix Tree Construction – Part 3

Ukkonen's Suffix Tree Construction – Part 4

Ukkonen's Suffix Tree Construction – Part 5

Please go through Part 1, Part 2, Part 3, Part 4 and Part 5, before looking at current article, where we have seen few basics on suffix tree, high level ukkonen's algorithm, suffix link and three implementation tricks and activePoints along with an example string "abcabxabcd" where we went through all phases of building suffix tree.
Here, we will see the data structure used to represent suffix tree and the code implementation.

At that end of Part 5 article, we have discussed some of the operations we will be doing while building suffix tree and later when we use suffix tree in different applications.
There could be different possible data structures we may think of to fulfill the requirements where some data structure may be slow on some operations and some fast. Here we will use following in our implementation:

We will have SuffixTreeNode structure to represent each node in tree. SuffixTreeNode structure will have following members:

- **children** – This will be an array of alphabet size. This will store all the children nodes of current node on different edges starting with different characters.
- **suffixLink** – This will point to other node where current node should point via suffix link.
- **start, end** – These two will store the edge label details from parent node to current node. (start, end) interval specifies the edge, by which the node is connected to its parent node. Each edge will connect two nodes, one parent and one child, and (start, end) interval of a given edge will be stored in the child node. Lets say there are two nods A (parent) and B (Child) connected by an edge with indices (5, 8) then this indices (5, 8) will be stored in node B.

- **suffixIndex** – This will be non-negative for leaves and will give index of suffix for the path from root to this leaf. For non-leaf node, it will be -1 .

This data structure will answer to the required queries quickly as below:

- How to check if a node is root ? — Root is a special node, with no parent and so it's start and end will be -1, for all other nodes, start and end indices will be non-negative.
- How to check if a node is internal or leaf node ? — suffixIndex will help here. It will be -1 for internal node and non-negative for leaf nodes.
- What is the length of path label on some edge? — Each edge will have start and end indices and length of path label will be end-start+1
- What is the path label on some edge ? — If string is S, then path label will be substring of S from start index to end index inclusive, [start, end].
- How to check if there is an outgoing edge for a given character c from a node A ? — If A->children[c] is not NULL, there is a path, if NULL, no path.
- What is the character value on an edge at some given distance d from a node A ? — Character at distance d from node A will be S[A->start + d], where S is the string.
- Where an internal node is pointing via suffix link ? — Node A will point to A->suffixLink
- What is the suffix index on a path from root to leaf ? — If leaf node is A on the path, then suffix index on that path will be A->suffixIndex

Following is C implementation of Ukkonen's Suffix Tree Construction. The code may look a bit lengthy, probably because of a good amount of comments.

```
// A C program to implement Ukkonen's Suffix Tree Construction
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;

    /*(start, end) interval specifies the edge, by which the
     node is connected to its parent node. Each edge will
     connect two nodes,  one parent and one child, and
     (start, end) interval of a given edge  will be stored
     in the child node. Lets say there are two nods A and B
     connected by an edge with indices (5, 8) then this
     indices (5, 8) will be stored in node B. */
    int start;
    int *end;

    /*for leaf nodes, it stores the index of suffix for
      the path  from root to leaf*/
    int suffixIndex;
};

typedef struct SuffixTreeNode Node;
```

```
char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
  waiting for it's suffix link to be set, which might get
  a new suffix link (other than root) in next extension of
  same phase. lastNewNode will be set to NULL when last
  newly created internal node (if there is any) got it's
  suffix link reset to new internal node created in next
  extension of same phase. */
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represeted as input string character
  index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string

Node *newNode(int start, int *end)
{
    Node *node =(Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
          node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in  current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

    /*suffixIndex will be set to -1 by default and
      actual suffix index will be set later for leaves
      at the end of all phases*/
    node->suffixIndex = -1;
    return node;
}

int edgeLength(Node *n) {
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
```

```
{
    /*activePoint change for walk down (APCFWD) using
     Skip/Count Trick  (Trick 1). If activeLength is greater
     than current edge length, set next  internal node as
     activeNode and adjust activeEdge and activeLength
     accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
    return 0;
}

void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
    leaves created so far in tree*/
    leafEnd = pos;

    /*Increment remainingSuffixCount indicating that a
    new suffix added to the list of suffixes yet to be
    added in tree*/
    remainingSuffixCount++;

    /*set lastNewNode to NULL while starting a new phase,
     indicating there is no internal node waiting for
     it's suffix link reset in current phase*/
    lastNewNode = NULL;

    //Add all suffixes (yet to be added) one by one in tree
    while(remainingSuffixCount > 0) {

        if (activeLength == 0)
            activeEdge = pos; //APCFALZ

        // There is no outgoing edge starting with
        // activeEdge from activeNode
        if (activeNode->children[text[activeEdge]] == NULL)
        {
            //Extension Rule 2 (A new leaf edge gets created)
            activeNode->children[text[activeEdge]] =
                                        newNode(pos, &leafEnd);

            /*A new leaf edge is created in above line starting
             from  an existng node (the current activeNode), and
             if there is any internal node waiting for it's suffix
             link get reset, point the suffix link from that last
             internal node to current activeNode. Then set lastNewNode
             to NULL indicating no more node waiting for suffix link
             reset.*/
            if (lastNewNode != NULL)
```

```
    {
        lastNewNode->suffixLink = activeNode;
        lastNewNode = NULL;
    }
}
// There is an outgoing edge starting with activeEdge
// from activeNode
else
{
    // Get the next node at the end of edge starting
    // with activeEdge
    Node *next = activeNode->children[text[activeEdge]];
    if (walkDown(next))//Do walkdown
    {
        //Start from next node (the new activeNode)
        continue;
    }
    /*Extension Rule 3 (current character being processed
      is already on the edge)*/
    if (text[next->start + activeLength] == text[pos])
    {
        //If a newly created node waiting for it's
        //suffix link to be set, then set suffix link
        //of that waiting node to curent active node
        if(lastNewNode != NULL && activeNode != root)
        {
            lastNewNode->suffixLink = activeNode;
            lastNewNode = NULL;
        }

        //APCFER3
        activeLength++;
        /*STOP all further processing in this phase
        and move on to next phase*/
        break;
    }

    /*We will be here when activePoint is in middle of
      the edge being traversed and current character
      being processed is not  on the edge (we fall off
      the tree). In this case, we add a new internal node
      and a new leaf edge going out of that new node. This
      is Extension Rule 2, where a new leaf edge and a new
    internal node get created*/
    splitEnd = (int*) malloc(sizeof(int));
    *splitEnd = next->start + activeLength - 1;

    //New internal node
    Node *split = newNode(next->start, splitEnd);
    activeNode->children[text[activeEdge]] = split;

    //New leaf coming out of new internal node
    split->children[text[pos]] = newNode(pos, &leafEnd);
    next->start += activeLength;
```

```
            split->children[text[next->start]] = next;

            /*We got a new internal node here. If there is any
              internal node created in last extensions of same
              phase which is still waiting for it's suffix link
              reset, do it now.*/
            if (lastNewNode != NULL)
            {
            /*suffixLink of lastNewNode points to current newly
              created internal node*/
                lastNewNode->suffixLink = split;
            }

            /*Make the current newly created internal node waiting
              for it's suffix link reset (which is pointing to root
              at present). If we come across any other internal node
              (existing or newly created) in next extension of same
              phase, when a new leaf edge gets added (i.e. when
              Extension Rule 2 applies is any of the next extension
              of same phase) at that point, suffixLink of this node
              will point to that internal node.*/
            lastNewNode = split;
        }

        /* One suffix got added in tree, decrement the count of
          suffixes yet to be added.*/
        remainingSuffixCount--;
        if (activeNode == root && activeLength > 0) //APCFER2C1
        {
            activeLength--;
            activeEdge = pos - remainingSuffixCount + 1;
        }
        else if (activeNode != root) //APCFER2C2
        {
            activeNode = activeNode->suffixLink;
        }
    }
}

void print(int i, int j)
{
    int k;
    for (k=i; k<=j; k++)
        printf("%c", text[k]);
}

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL)  return;

    if (n->start != -1) //A non-root node
```

```
    {
        //Print the label on edge from parent to current node
        print(n->start, *(n->end));
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            if (leaf == 1 && n->start != -1)
                printf(" [%d]\n", n->suffixIndex);

            //Current node is not a leaf as it has outgoing
            //edges from it.
            leaf = 0;
            setSuffixIndexByDFS(n->children[i], labelHeight +
                                edgeLength(n->children[i]));
        }
    }
    if (leaf == 1)
    {
        n->suffixIndex = size - labelHeight;
        printf(" [%d]\n", n->suffixIndex);
    }
}

void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;
```

```
    /*Root is a special node with start and end indices as -1,
    as it has no parent from where an edge comes to root*/
    root = newNode(-1, rootEnd);

    activeNode = root; //First activeNode will be root
    for (i=0; i<size; i++)
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);

    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
}

// driver program to test above functions
int main(int argc, char *argv[])
{
//  strcpy(text, "abc"); buildSuffixTree();
//  strcpy(text, "xabxac#");    buildSuffixTree();
//  strcpy(text, "xabxa");  buildSuffixTree();
//  strcpy(text, "xabxa$"); buildSuffixTree();
    strcpy(text, "abcabxabcd$"); buildSuffixTree();
//  strcpy(text, "geeksforgeeks$"); buildSuffixTree();
//  strcpy(text, "THIS IS A TEST TEXT$"); buildSuffixTree();
//  strcpy(text, "AABAACAADAABAAABAA$"); buildSuffixTree();
    return 0;
}
```

Output (Each edge of Tree, along with suffix index of child node on edge, is printed in DFS order. To understand the output better, match it with the last figure no 43 in previous Part 5 article):

```
$ [10]
ab [-1]
c [-1]
abxabcd$ [0]
d$ [6]
xabcd$ [3]
b [-1]
c [-1]
abxabcd$ [1]
d$ [7]
xabcd$ [4]
c [-1]
abxabcd$ [2]
d$ [8]
d$ [9]
xabcd$ [5]
```

Now we are able to build suffix tree in linear time, we can solve many string problem in efficient way:

- Check if a given pattern P is substring of text T (Useful when text is fixed and pattern changes, KMP otherwise

- Find all occurrences of a given pattern P present in text T
- Find longest repeated substring
- Linear Time Suffix Array Creation

The above basic problems can be solved by DFS traversal on suffix tree.
We will soon post articles on above problems and others like below:

And More.

Test you understanding?

1. Draw suffix tree (with proper suffix link, suffix indices) for string "AABAACAADAABAAABAA$" on paper and see if that matches with code output.
2. Every extension must follow one of the three rules: Rule 1, Rule 2 and Rule 3.
   Following are the rules applied on five consecutive extensions in some Phase i (i > 5), which ones are valid:
   A) Rule 1, Rule 2, Rule 2, Rule 3, Rule 3
   B) Rule 1, Rule 2, Rule 2, Rule 3, Rule 2
   C) Rule 2, Rule 1, Rule 1, Rule 3, Rule 3
   D) Rule 1, Rule 1, Rule 1, Rule 1, Rule 1
   E) Rule 2, Rule 2, Rule 2, Rule 2, Rule 2
   F) Rule 3, Rule 3, Rule 3, Rule 3, Rule 3
3. What are the valid sequences in above for Phase 5
4. Every internal node MUST have it's suffix link set to another node (internal or root). Can a newly created node point to already existing internal node or not ? Can it happen that a new node created in extension j, may not get it's right suffix link in next extension j+1 and get the right one in later extensions like j+2, j+3 etc ?
5. Try solving the basic problems discussed above.

We have published following articles on suffix tree applications:

References:
http://web.stanford.edu/~mjkay/gusfield.pdf

Ukkonen's suffix tree algorithm in plain English

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-6/

Category: Strings Tags: Pattern Searching

Post navigation

← Ukkonen's Suffix Tree Construction – Part 5 Nagarro Interview Experience | Set 4 (Off-Campus) →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 21

Generalized Suffix Tree 1

In earlier suffix tree articles, we created suffix tree for one string and then we queried that tree for substring check, searching all patterns, longest repeated substring and built suffix array (All linear time operations).

There are lots of other problems where multiple strings are involved.
e.g. pattern searching in a text file or dictionary, spell checker, phone book, Autocomplete, Longest common substring problem, Longest palindromic substring and More.

For such operations, all the involved strings need to be indexed for faster search and retrieval. One way to do this is using suffix trie or suffix tree. We will discuss suffix tree here.
A suffix tree made of a set of strings is known as Generalized Suffix Tree.
We will discuss a simple way to build Generalized Suffix Tree here for **two strings only**.
Later, we will discuss another approach to build Generalized Suffix Tree for **two or more strings**.

Here we will use the suffix tree implementation for one string discussed already and modify that a bit to build generalized suffix tree.

Lets consider two strings X and Y for which we want to build generalized suffix tree. For this we will make a new string X#Y$ where # and $ both are terminal symbols (must be unique). Then we will build suffix tree for X#Y$ which will be the generalized suffix tree for X and Y. Same logic will apply for more than two strings (i.e. concatenate all strings using unique terminal symbols and then build suffix tree for concatenated string).

Lets say X = xabxa, and Y = babxba, then
X#Y$ = xabxa#babxba$
If we run the code implemented at Ukkonen's Suffix Tree Construction – Part 6 for string xabxa#babxba$, we get following output:

(Click to see it clearly)

We can use this tree to solve some of the problems, but we can refine it a bit by removing unwanted substrings on a path label. A path label should have substring from only one input string, so if there are path labels having substrings from multiple input strings, we can keep only the initial portion corresponding to one string and remove all the later portion. For example, for path labels #babxba$, a#babxba$ and bxa#babxba$, we can remove babxba$ (belongs to 2^{nd} input string) and then new path labels will be #, a# and bxa# respectively. With this change, above diagram will look like below:

(Click to see it clearly)

Below implementation is built on top of original implementation. Here we are removing unwanted characters on path labels. If a path label has "#" character in it, then we are trimming all characters after the "#" in that path label.
Note: This implementation builds generalized suffix tree for only two strings X and Y which are concatenated as X#Y$

```
// A C program to implement Ukkonen's Suffix Tree Construction
// And then build generalized suffix tree
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;

    /*(start, end) interval specifies the edge, by which the
     node is connected to its parent node. Each edge will
     connect two nodes,  one parent and one child, and
     (start, end) interval of a given edge  will be stored
     in the child node. Lets say there are two nods A and B
     connected by an edge with indices (5, 8) then this
     indices (5, 8) will be stored in node B. */
    int start;
    int *end;

    /*for leaf nodes, it stores the index of suffix for
      the path  from root to leaf*/
    int suffixIndex;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
  waiting for it's suffix link to be set, which might get
  a new suffix link (other than root) in next extension of
  same phase. lastNewNode will be set to NULL when last
  newly created internal node (if there is any) got it's
  suffix link reset to new internal node created in next
  extension of same phase. */
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represeted as input string character
  index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;
```

```
// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string

Node *newNode(int start, int *end)
{
    Node *node =(Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
          node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in  current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

    /*suffixIndex will be set to -1 by default and
      actual suffix index will be set later for leaves
      at the end of all phases*/
    node->suffixIndex = -1;
    return node;
}

int edgeLength(Node *n) {
    if(n == root)
        return 0;
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
     Skip/Count Trick  (Trick 1). If activeLength is greater
     than current edge length, set next  internal node as
     activeNode and adjust activeEdge and activeLength
     accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
    return 0;
}
```

```
void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
    leaves created so far in tree*/
    leafEnd = pos;

    /*Increment remainingSuffixCount indicating that a
    new suffix added to the list of suffixes yet to be
    added in tree*/
    remainingSuffixCount++;

    /*set lastNewNode to NULL while starting a new phase,
     indicating there is no internal node waiting for
     it's suffix link reset in current phase*/
    lastNewNode = NULL;

    //Add all suffixes (yet to be added) one by one in tree
    while(remainingSuffixCount > 0) {

        if (activeLength == 0)
            activeEdge = pos; //APCFALZ

        // There is no outgoing edge starting with
        // activeEdge from activeNode
        if (activeNode->children[text[activeEdge]] == NULL)
        {
            //Extension Rule 2 (A new leaf edge gets created)
            activeNode->children[text[activeEdge]] =
                                        newNode(pos, &leafEnd);

            /*A new leaf edge is created in above line starting
             from  an existng node (the current activeNode), and
             if there is any internal node waiting for it's suffix
             link get reset, point the suffix link from that last
             internal node to current activeNode. Then set lastNewNode
             to NULL indicating no more node waiting for suffix link
             reset.*/
            if (lastNewNode != NULL)
            {
                lastNewNode->suffixLink = activeNode;
                lastNewNode = NULL;
            }
        }
        // There is an outgoing edge starting with activeEdge
        // from activeNode
        else
        {
            // Get the next node at the end of edge starting
            // with activeEdge
            Node *next = activeNode->children[text[activeEdge]];
            if (walkDown(next))//Do walkdown
            {
                //Start from next node (the new activeNode)
                continue;
```

```
}
/*Extension Rule 3 (current character being processed
  is already on the edge)*/
if (text[next->start + activeLength] == text[pos])
{
    //If a newly created node waiting for it's
    //suffix link to be set, then set suffix link
    //of that waiting node to curent active node
    if(lastNewNode != NULL && activeNode != root)
    {
        lastNewNode->suffixLink = activeNode;
        lastNewNode = NULL;
    }

    //APCFER3
    activeLength++;
    /*STOP all further processing in this phase
    and move on to next phase*/
    break;
}

/*We will be here when activePoint is in middle of
  the edge being traversed and current character
  being processed is not  on the edge (we fall off
  the tree). In this case, we add a new internal node
  and a new leaf edge going out of that new node. This
  is Extension Rule 2, where a new leaf edge and a new
internal node get created*/
splitEnd = (int*) malloc(sizeof(int));
*splitEnd = next->start + activeLength - 1;

//New internal node
Node *split = newNode(next->start, splitEnd);
activeNode->children[text[activeEdge]] = split;

//New leaf coming out of new internal node
split->children[text[pos]] = newNode(pos, &leafEnd);
next->start += activeLength;
split->children[text[next->start]] = next;

/*We got a new internal node here. If there is any
  internal node created in last extensions of same
  phase which is still waiting for it's suffix link
  reset, do it now.*/
if (lastNewNode != NULL)
{
/*suffixLink of lastNewNode points to current newly
  created internal node*/
    lastNewNode->suffixLink = split;
}

/*Make the current newly created internal node waiting
  for it's suffix link reset (which is pointing to root
  at present). If we come across any other internal node
```

```
              (existing or newly created) in next extension of same
              phase, when a new leaf edge gets added (i.e. when
              Extension Rule 2 applies is any of the next extension
              of same phase) at that point, suffixLink of this node
              will point to that internal node.*/
            lastNewNode = split;
        }

        /* One suffix got added in tree, decrement the count of
          suffixes yet to be added.*/
        remainingSuffixCount--;
        if (activeNode == root && activeLength > 0) //APCFER2C1
        {
            activeLength--;
            activeEdge = pos - remainingSuffixCount + 1;
        }
        else if (activeNode != root) //APCFER2C2
        {
            activeNode = activeNode->suffixLink;
        }
    }
}

void print(int i, int j)
{
    int k;
    for (k=i; k<=j && text[k] != '#'; k++)
        printf("%c", text[k]);
    if(k<=j)
        printf("#");
}

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL)  return;

    if (n->start != -1) //A non-root node
    {
        //Print the label on edge from parent to current node
        print(n->start, *(n->end));
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            if (leaf == 1 && n->start != -1)
                printf(" [%d]\n", n->suffixIndex);

            //Current node is not a leaf as it has outgoing
```

```
            //edges from it.
            leaf = 0;
            setSuffixIndexByDFS(n->children[i], labelHeight +
                                edgeLength(n->children[i]));
        }
    }
    if (leaf == 1)
    {
        for(i= n->start; i<= *(n->end); i++)
        {
            if(text[i] == '#') //Trim unwanted characters
            {
                n->end = (int*) malloc(sizeof(int));
                *(n->end) = i;
            }
        }
        n->suffixIndex = size - labelHeight;
        printf(" [%d]\n", n->suffixIndex);
    }
}

void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;

    /*Root is a special node with start and end indices as -1,
    as it has no parent from where an edge comes to root*/
    root = newNode(-1, rootEnd);

    activeNode = root; //First activeNode will be root
    for (i=0; i<size; i++)
```

```
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);

    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
}

// driver program to test above functions
int main(int argc, char *argv[])
{
//  strcpy(text, "xabxac#abcabxabcd$"); buildSuffixTree();
    strcpy(text, "xabxa#babxba$"); buildSuffixTree();
    return 0;
}
```

Output: (You can see that below output corresponds to the 2nd Figure shown above)

```
# [5]
$ [12]
a [-1]
# [4]
$ [11]
bx [-1]
a# [1]
ba$ [7]
b [-1]
a [-1]
$ [10]
bxba$ [6]
x [-1]
a# [2]
ba$ [8]
x [-1]
a [-1]
# [3]
bxa# [0]
ba$ [9]
```

If two strings are of size M and N, this implementation will take O(M+N) time and space.
If input strings are not concatenated already, then it will take 2(M+N) space in total, M+N space to store the generalized suffix tree and another M+N space to store concatenated string.

Followup:
Extend above implementation for more than two strings (i.e. concatenate all strings using unique terminal symbols and then build suffix tree for concatenated string)

One problem with this approach is the need of unique terminal symbol for each input string. This will work for few strings but if there is too many input strings, we may not be able to find that many unique terminal symbols.
We will discuss another approach to build generalized suffix tree soon where we will need only one unique

terminal symbol and that will resolve the above problem and can be used to build generalized suffix tree for any number of input strings.

We have published following more articles on suffix tree applications:

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/generalized-suffix-tree-1/

Category: Strings Tags: Pattern Searching

Post navigation

← Vertex Cover Problem | Set 1 (Introduction and Approximate Algorithm) Suffix Tree Application 5 – Longest Common Substring →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 22

Suffix Tree Application 4 - Build Linear Time Suffix Array

Given a string, build it's Suffix Array
We have already discussed following two ways of building suffix array:

Please go through these to have the basic understanding.
Here we will see how to build suffix array in linear time using suffix tree.

As a prerequisite, we must know how to build a suffix tree in one or the other way.
Here we will build suffix tree using Ukkonen's Algorithm, discussed already as below:

Ukkonen's Suffix Tree Construction – Part 1

Ukkonen's Suffix Tree Construction – Part 2

Ukkonen's Suffix Tree Construction – Part 3

Ukkonen's Suffix Tree Construction – Part 4

Ukkonen's Suffix Tree Construction – Part 5

Ukkonen's Suffix Tree Construction – Part 6

Lets consider string abcabxabcd.
It's suffix array would be:

```
0 6 3 1 7 4 2 8 9 5
```

Lets look at following figure:

This is suffix tree for String "abcabxabcd$"
If we do a DFS traversal, visiting edges in lexicographic order (we have been doing the same traversal in other Suffix Tree Application articles as well) and print suffix indices on leaves, we will get following:

```
 10 0 6 3 1 7 4 2 8 9 5
```

"$" is lexicographically lesser than [a-zA-Z].
The suffix index 10 corresponds to edge with "$" label.
Except this 1st suffix index, the sequence of all other numbers gives the suffix array of the string.

So if we have a suffix tree of the string, then to get it's suffix array, we just need to do a lexicographic order DFS traversal and store all the suffix indices in resultant suffix array, except the very 1st suffix index.

```
// A C program to implement Ukkonen's Suffix Tree Construction
// And and then create suffix array in linear time
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;

    /*(start, end) interval specifies the edge, by which the
     node is connected to its parent node. Each edge will
     connect two nodes,  one parent and one child, and
     (start, end) interval of a given edge  will be stored
     in the child node. Lets say there are two nods A and B
     connected by an edge with indices (5, 8) then this
     indices (5, 8) will be stored in node B. */
    int start;
    int *end;

    /*for leaf nodes, it stores the index of suffix for
      the path  from root to leaf*/
    int suffixIndex;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
  waiting for it's suffix link to be set, which might get
  a new suffix link (other than root) in next extension of
  same phase. lastNewNode will be set to NULL when last
  newly created internal node (if there is any) got it's
  suffix link reset to new internal node created in next
  extension of same phase. */
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represeted as input string character
```

```
  index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string

Node *newNode(int start, int *end)
{
    Node *node =(Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
          node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in  current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

    /*suffixIndex will be set to -1 by default and
      actual suffix index will be set later for leaves
      at the end of all phases*/
    node->suffixIndex = -1;
    return node;
}

int edgeLength(Node *n) {
    if(n == root)
        return 0;
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
     Skip/Count Trick  (Trick 1). If activeLength is greater
     than current edge length, set next  internal node as
     activeNode and adjust activeEdge and activeLength
     accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
```

```
    return 0;
}

void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
    leaves created so far in tree*/
    leafEnd = pos;

    /*Increment remainingSuffixCount indicating that a
    new suffix added to the list of suffixes yet to be
    added in tree*/
    remainingSuffixCount++;

    /*set lastNewNode to NULL while starting a new phase,
     indicating there is no internal node waiting for
     it's suffix link reset in current phase*/
    lastNewNode = NULL;

    //Add all suffixes (yet to be added) one by one in tree
    while(remainingSuffixCount > 0) {

        if (activeLength == 0)
            activeEdge = pos; //APCFALZ

        // There is no outgoing edge starting with
        // activeEdge from activeNode
        if (activeNode->children[text[activeEdge]] == NULL)
        {
            //Extension Rule 2 (A new leaf edge gets created)
            activeNode->children[text[activeEdge]] =
                                        newNode(pos, &leafEnd);

            /*A new leaf edge is created in above line starting
             from  an existng node (the current activeNode), and
             if there is any internal node waiting for it's suffix
             link get reset, point the suffix link from that last
             internal node to current activeNode. Then set lastNewNode
             to NULL indicating no more node waiting for suffix link
             reset.*/
            if (lastNewNode != NULL)
            {
                lastNewNode->suffixLink = activeNode;
                lastNewNode = NULL;
            }
        }
        // There is an outgoing edge starting with activeEdge
        // from activeNode
        else
        {
            // Get the next node at the end of edge starting
            // with activeEdge
            Node *next = activeNode->children[text[activeEdge]];
            if (walkDown(next))//Do walkdown
```

```
{
    //Start from next node (the new activeNode)
    continue;
}
/*Extension Rule 3 (current character being processed
  is already on the edge)*/
if (text[next->start + activeLength] == text[pos])
{
    //If a newly created node waiting for it's
    //suffix link to be set, then set suffix link
    //of that waiting node to curent active node
    if(lastNewNode != NULL && activeNode != root)
    {
        lastNewNode->suffixLink = activeNode;
        lastNewNode = NULL;
    }

    //APCFER3
    activeLength++;
    /*STOP all further processing in this phase
    and move on to next phase*/
    break;
}

/*We will be here when activePoint is in middle of
  the edge being traversed and current character
  being processed is not  on the edge (we fall off
  the tree). In this case, we add a new internal node
  and a new leaf edge going out of that new node. This
  is Extension Rule 2, where a new leaf edge and a new
internal node get created*/
splitEnd = (int*) malloc(sizeof(int));
*splitEnd = next->start + activeLength - 1;

//New internal node
Node *split = newNode(next->start, splitEnd);
activeNode->children[text[activeEdge]] = split;

//New leaf coming out of new internal node
split->children[text[pos]] = newNode(pos, &leafEnd);
next->start += activeLength;
split->children[text[next->start]] = next;

/*We got a new internal node here. If there is any
  internal node created in last extensions of same
  phase which is still waiting for it's suffix link
  reset, do it now.*/
if (lastNewNode != NULL)
{
/*suffixLink of lastNewNode points to current newly
  created internal node*/
    lastNewNode->suffixLink = split;
}
```

```
            /*Make the current newly created internal node waiting
              for it's suffix link reset (which is pointing to root
              at present). If we come across any other internal node
              (existing or newly created) in next extension of same
              phase, when a new leaf edge gets added (i.e. when
              Extension Rule 2 applies is any of the next extension
              of same phase) at that point, suffixLink of this node
              will point to that internal node.*/
            lastNewNode = split;
        }

        /* One suffix got added in tree, decrement the count of
          suffixes yet to be added.*/
        remainingSuffixCount--;
        if (activeNode == root && activeLength > 0) //APCFER2C1
        {
            activeLength--;
            activeEdge = pos - remainingSuffixCount + 1;
        }
        else if (activeNode != root) //APCFER2C2
        {
            activeNode = activeNode->suffixLink;
        }
    }
}

void print(int i, int j)
{
    int k;
    for (k=i; k<=j; k++)
        printf("%c", text[k]);
}

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL)  return;

    if (n->start != -1) //A non-root node
    {
        //Print the label on edge from parent to current node
        //Uncomment below line to print suffix tree
       // print(n->start, *(n->end));
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            //Uncomment below two lines to print suffix index
           // if (leaf == 1 && n->start != -1)
```

```
             //   printf(" [%d]\n", n->suffixIndex);

            //Current node is not a leaf as it has outgoing
            //edges from it.
            leaf = 0;
            setSuffixIndexByDFS(n->children[i], labelHeight +
                                edgeLength(n->children[i]));
        }
    }
    if (leaf == 1)
    {
        n->suffixIndex = size - labelHeight;
        //Uncomment below line to print suffix index
        //printf(" [%d]\n", n->suffixIndex);
    }
}

void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;

    /*Root is a special node with start and end indices as -1,
    as it has no parent from where an edge comes to root*/
    root = newNode(-1, rootEnd);

    activeNode = root; //First activeNode will be root
    for (i=0; i<size; i++)
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);
}
```

```
void doTraversal(Node *n, int suffixArray[], int *idx)
{
    if(n == NULL)
    {
        return;
    }
    int i=0;
    if(n->suffixIndex == -1) //If it is internal node
    {
        for (i = 0; i < MAX_CHAR; i++)
        {
            if(n->children[i] != NULL)
            {
                doTraversal(n->children[i], suffixArray, idx);
            }
        }
    }
    //If it is Leaf node other than "$" label
    else if(n->suffixIndex > -1 && n->suffixIndex < size)
    {
        suffixArray[(*idx)++] = n->suffixIndex;
    }
}

void buildSuffixArray(int suffixArray[])
{
    int i = 0;
    for(i=0; i< size; i++)
        suffixArray[i] = -1;
    int idx = 0;
    doTraversal(root, suffixArray, &idx);
    printf("Suffix Array for String ");
    for(i=0; i<size; i++)
        printf("%c", text[i]);
    printf(" is: ");
    for(i=0; i<size; i++)
        printf("%d ", suffixArray[i]);
    printf("\n");
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    strcpy(text, "banana$");
    buildSuffixTree();
    size--;
    int *suffixArray =(int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    strcpy(text, "GEEKSFORGEEKS$");
```

```
buildSuffixTree();
size--;
suffixArray =(int*) malloc(sizeof(int) * size);
buildSuffixArray(suffixArray);
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);
free(suffixArray);

strcpy(text, "AAAAAAAAAA$");
buildSuffixTree();
size--;
suffixArray =(int*) malloc(sizeof(int) * size);
buildSuffixArray(suffixArray);
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);
free(suffixArray);

strcpy(text, "ABCDEFG$");
buildSuffixTree();
size--;
suffixArray =(int*) malloc(sizeof(int) * size);
buildSuffixArray(suffixArray);
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);
free(suffixArray);

strcpy(text, "ABABABA$");
buildSuffixTree();
size--;
suffixArray =(int*) malloc(sizeof(int) * size);
buildSuffixArray(suffixArray);
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);
free(suffixArray);

strcpy(text, "abcabxabcd$");
buildSuffixTree();
size--;
suffixArray =(int*) malloc(sizeof(int) * size);
buildSuffixArray(suffixArray);
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);
free(suffixArray);

strcpy(text, "CCAAACCCGATTA$");
buildSuffixTree();
size--;
suffixArray =(int*) malloc(sizeof(int) * size);
buildSuffixArray(suffixArray);
//Free the dynamically allocated memory
freeSuffixTreeByPostOrder(root);
free(suffixArray);

return 0;
```

```
}
```

Output:

```
Suffix Array for String banana is: 5 3 1 0 4 2
Suffix Array for String GEEKSFORGEEKS is: 9 1 10 2 5 8 0 11 3 6 7 12 4
Suffix Array for String AAAAAAAAAA is: 9 8 7 6 5 4 3 2 1 0
Suffix Array for String ABCDEFG is: 0 1 2 3 4 5 6
Suffix Array for String ABABABA is: 6 4 2 0 5 3 1
Suffix Array for String abcabxabcd is: 0 6 3 1 7 4 2 8 9 5
Suffix Array for String CCAAACCCGATTA is: 12 2 3 4 9 1 0 5 6 7 8 11 10
```

Ukkonen's Suffix Tree Construction takes O(N) time and space to build suffix tree for a string of length N and after that, traversal of tree take O(N) to build suffix array.
So overall, it's linear in time and space.
Can you see why traversal is O(N) ?? Because a suffix tree of string of length N will have at most N-1 internal nodes and N leaves. Traversal of these nodes can be done in O(N).

We have published following more articles on suffix tree applications:

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/suffix-tree-application-4-build-linear-time-suffix-array/

Category: Strings Tags: Pattern Searching

Post navigation

← KLA Tencor Interview Experience Wizecommerce On-Campus Interview Experience →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 23

Suffix Tree Application 1 - Substring Check

Given a text string and a pattern string, check if pattern exists in text or not.

Few pattern searching algorithms (KMP, Rabin-Karp, Naive Algorithm, Finite Automata) are already discussed, which can be used for this check.
Here we will discuss suffix tree based algorithm.

As a prerequisite, we must know how to build a suffix tree in one or the other way.
Once we have a suffix tree built for given text, we need to traverse the tree from root to leaf against the characters in pattern. If we do not fall off the tree (i.e. there is a path from root to leaf or somewhere in middle) while traversal, then pattern exists in text as a substring.

Here we will build suffix tree using Ukkonen's Algorithm, discussed already as below:

Ukkonen's Suffix Tree Construction – Part 1

Ukkonen's Suffix Tree Construction – Part 2

Ukkonen's Suffix Tree Construction – Part 3

Ukkonen's Suffix Tree Construction – Part 4

Ukkonen's Suffix Tree Construction – Part 5

Ukkonen's Suffix Tree Construction – Part 6

The core traversal implementation for substring check, can be modified accordingly for suffix trees built by other algorithms.

```
// A C program for substring check using Ukkonen's Suffix Tree Construction
#include <stdio.h>
```

```
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;

    /*(start, end) interval specifies the edge, by which the
     node is connected to its parent node. Each edge will
     connect two nodes,  one parent and one child, and
     (start, end) interval of a given edge  will be stored
     in the child node. Lets say there are two nods A and B
     connected by an edge with indices (5, 8) then this
     indices (5, 8) will be stored in node B. */
    int start;
    int *end;

    /*for leaf nodes, it stores the index of suffix for
      the path  from root to leaf*/
    int suffixIndex;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
  waiting for it's suffix link to be set, which might get
  a new suffix link (other than root) in next extension of
  same phase. lastNewNode will be set to NULL when last
  newly created internal node (if there is any) got it's
  suffix link reset to new internal node created in next
  extension of same phase. */
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represeted as input string character
  index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string

Node *newNode(int start, int *end)
```

```
{
    Node *node =(Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
          node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in  current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

    /*suffixIndex will be set to -1 by default and
      actual suffix index will be set later for leaves
      at the end of all phases*/
    node->suffixIndex = -1;
    return node;
}

int edgeLength(Node *n) {
    if(n == root)
        return 0;
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
     Skip/Count Trick  (Trick 1). If activeLength is greater
     than current edge length, set next  internal node as
     activeNode and adjust activeEdge and activeLength
     accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
    return 0;
}

void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
    leaves created so far in tree*/
    leafEnd = pos;

    /*Increment remainingSuffixCount indicating that a
    new suffix added to the list of suffixes yet to be
    added in tree*/
    remainingSuffixCount++;
```

```
/*set lastNewNode to NULL while starting a new phase,
 indicating there is no internal node waiting for
 it's suffix link reset in current phase*/
lastNewNode = NULL;

//Add all suffixes (yet to be added) one by one in tree
while(remainingSuffixCount > 0) {

    if (activeLength == 0)
        activeEdge = pos; //APCFALZ

    // There is no outgoing edge starting with
    // activeEdge from activeNode
    if (activeNode->children[text[activeEdge]] == NULL)
    {
        //Extension Rule 2 (A new leaf edge gets created)
        activeNode->children[text[activeEdge]] =
                                    newNode(pos, &leafEnd);

        /*A new leaf edge is created in above line starting
         from  an existng node (the current activeNode), and
         if there is any internal node waiting for it's suffix
         link get reset, point the suffix link from that last
         internal node to current activeNode. Then set lastNewNode
         to NULL indicating no more node waiting for suffix link
         reset.*/
        if (lastNewNode != NULL)
        {
            lastNewNode->suffixLink = activeNode;
            lastNewNode = NULL;
        }
    }
    // There is an outgoing edge starting with activeEdge
    // from activeNode
    else
    {
        // Get the next node at the end of edge starting
        // with activeEdge
        Node *next = activeNode->children[text[activeEdge]];
        if (walkDown(next))//Do walkdown
        {
            //Start from next node (the new activeNode)
            continue;
        }
        /*Extension Rule 3 (current character being processed
          is already on the edge)*/
        if (text[next->start + activeLength] == text[pos])
        {
            //If a newly created node waiting for it's
            //suffix link to be set, then set suffix link
            //of that waiting node to curent active node
            if(lastNewNode != NULL && activeNode != root)
            {
```

```
            lastNewNode->suffixLink = activeNode;
            lastNewNode = NULL;
        }

        //APCFER3
        activeLength++;
        /*STOP all further processing in this phase
        and move on to next phase*/
        break;
    }

    /*We will be here when activePoint is in middle of
      the edge being traversed and current character
      being processed is not  on the edge (we fall off
      the tree). In this case, we add a new internal node
      and a new leaf edge going out of that new node. This
      is Extension Rule 2, where a new leaf edge and a new
    internal node get created*/
    splitEnd = (int*) malloc(sizeof(int));
    *splitEnd = next->start + activeLength - 1;

    //New internal node
    Node *split = newNode(next->start, splitEnd);
    activeNode->children[text[activeEdge]] = split;

    //New leaf coming out of new internal node
    split->children[text[pos]] = newNode(pos, &leafEnd);
    next->start += activeLength;
    split->children[text[next->start]] = next;

    /*We got a new internal node here. If there is any
      internal node created in last extensions of same
      phase which is still waiting for it's suffix link
      reset, do it now.*/
    if (lastNewNode != NULL)
    {
    /*suffixLink of lastNewNode points to current newly
      created internal node*/
        lastNewNode->suffixLink = split;
    }

    /*Make the current newly created internal node waiting
      for it's suffix link reset (which is pointing to root
      at present). If we come across any other internal node
      (existing or newly created) in next extension of same
      phase, when a new leaf edge gets added (i.e. when
      Extension Rule 2 applies is any of the next extension
      of same phase) at that point, suffixLink of this node
      will point to that internal node.*/
    lastNewNode = split;
}

/* One suffix got added in tree, decrement the count of
  suffixes yet to be added.*/
```

```
        remainingSuffixCount--;
        if (activeNode == root && activeLength > 0) //APCFER2C1
        {
            activeLength--;
            activeEdge = pos - remainingSuffixCount + 1;
        }
        else if (activeNode != root) //APCFER2C2
        {
            activeNode = activeNode->suffixLink;
        }
    }
}

void print(int i, int j)
{
    int k;
    for (k=i; k<=j; k++)
        printf("%c", text[k]);
}

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL)  return;

    if (n->start != -1) //A non-root node
    {
        //Print the label on edge from parent to current node
        //Uncomment below line to print suffix tree
       // print(n->start, *(n->end));
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            //Uncomment below two lines to print suffix index
           // if (leaf == 1 && n->start != -1)
             //   printf(" [%d]\n", n->suffixIndex);

            //Current node is not a leaf as it has outgoing
            //edges from it.
            leaf = 0;
            setSuffixIndexByDFS(n->children[i], labelHeight +
                                edgeLength(n->children[i]));
        }
    }
    if (leaf == 1)
    {
        n->suffixIndex = size - labelHeight;
        //Uncomment below line to print suffix index
```

```
        //printf(" [%d]\n", n->suffixIndex);
    }
}

void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;

    /*Root is a special node with start and end indices as -1,
    as it has no parent from where an edge comes to root*/
    root = newNode(-1, rootEnd);

    activeNode = root; //First activeNode will be root
    for (i=0; i<size; i++)
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);
}

int traverseEdge(char *str, int idx, int start, int end)
{
    int k = 0;
    //Traverse the edge with character by character matching
    for(k=start; k<=end && str[idx] != '\0'; k++, idx++)
    {
        if(text[k] != str[idx])
            return -1;  // mo match
    }
    if(str[idx] == '\0')
        return 1;  // match
    return 0;  // more characters yet to match
```

```
}

int doTraversal(Node *n, char* str, int idx)
{
    if(n == NULL)
    {
        return -1; // no match
    }
    int res = -1;
    //If node n is not root node, then traverse edge
    //from node n's parent to node n.
    if(n->start != -1)
    {
        res = traverseEdge(str, idx, n->start, *(n->end));
        if(res != 0)
            return res;  // match (res = 1) or no match (res = -1)
    }
    //Get the character index to search
    idx = idx + edgeLength(n);
    //If there is an edge from node n going out
    //with current character str[idx], travrse that edge
    if(n->children[str[idx]] != NULL)
        return doTraversal(n->children[str[idx]], str, idx);
    else
        return -1;  // no match
}

void checkForSubString(char* str)
{
    int res = doTraversal(root, str, 0);
    if(res == 1)
        printf("Pattern <%s> is a Substring\n", str);
    else
        printf("Pattern <%s> is NOT a Substring\n", str);
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    strcpy(text, "THIS IS A TEST TEXT$");
    buildSuffixTree();

    checkForSubString("TEST");
    checkForSubString("A");
    checkForSubString(" ");
    checkForSubString("IS A");
    checkForSubString(" IS A ");
    checkForSubString("TEST1");
    checkForSubString("THIS IS GOOD");
    checkForSubString("TES");
    checkForSubString("TESA");
    checkForSubString("ISB");

    //Free the dynamically allocated memory
```

```
    freeSuffixTreeByPostOrder(root);

    return 0;
}
```

Output:

```
Pattern <TEST> is a Substring
Pattern <A> is a Substring
Pattern < > is a Substring
Pattern <IS A> is a Substring
Pattern < IS A > is a Substring
Pattern <TEST1> is NOT a Substring
Pattern <THIS IS GOOD> is NOT a Substring
Pattern <TES> is a Substring
Pattern <TESA> is NOT a Substring
Pattern <ISB> is NOT a Substring
```

Ukkonen's Suffix Tree Construction takes O(N) time and space to build suffix tree for a string of length N and after that, traversal for substring check takes O(M) for a pattern of length M.

With slight modification in traversal algorithm discussed here, we can answer following:

1. Find all occurrences of a given pattern P present in text T.
2. How to check if a pattern is prefix of a text?
3. How to check if a pattern is suffix of a text?

We have published following more articles on suffix tree applications:

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/suffix-tree-application-1-substring-check/

Category: Strings Tags: Pattern Searching

Post navigation

← Print Nodes in Top View of Binary Tree Algorithm Practice Question for Beginners | Set 1 →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 24

Suffix Tree Application 2 - Searching All Patterns

Given a text string and a pattern string, find all occurrences of the pattern in string.

Few pattern searching algorithms (KMP, Rabin-Karp, Naive Algorithm, Finite Automata) are already discussed, which can be used for this check.
Here we will discuss suffix tree based algorithm.

In the 1st Suffix Tree Application (Substring Check), we saw how to check whether a given pattern is substring of a text or not. It is advised to go through Substring Check 1st.
In this article, we will go a bit further on same problem. If a pattern is substring of a text, then we will find all the positions on pattern in the text.

As a prerequisite, we must know how to build a suffix tree in one or the other way.
Here we will build suffix tree using Ukkonen's Algorithm, discussed already as below:

Ukkonen's Suffix Tree Construction – Part 1

Ukkonen's Suffix Tree Construction – Part 2

Ukkonen's Suffix Tree Construction – Part 3

Ukkonen's Suffix Tree Construction – Part 4

Ukkonen's Suffix Tree Construction – Part 5

Ukkonen's Suffix Tree Construction – Part 6

Lets look at following figure:

This is suffix tree for String "abcabxabcd$", showing suffix indices and edge label indices (start, end). The (sub)string value on edges are shown only for explanatory purpose. We never store path label string in the tree.
Suffix Index of a path tells the index of a substring (starting from root) on that path.

Consider a path "bcd$" in above tree with suffix index 7. It tells that substrings b, bc, bcd, bcd$ are at index 7 in string.
Similarly path "bxabcd$" with suffix index 4 tells that substrings b, bx, bxa, bxab, bxabc, bxabcd, bxabcd$ are at index 4.
Similarly path "bcabxabcd$" with suffix index 1 tells that substrings b, bc, bca, bcab, bcabx, bcabxa, bcabxab, bcabxabc, bcabxabcd, bcabxabcd$ are at index 1.

If we see all the above three paths together, we can see that:

- Substring "b" is at indices 1, 4 and 7
- Substring "bc" is at indices 1 and 7

With above explanation, we should be able to see following:

- Substring "ab" is at indices 0, 3 and 6
- Substring "abc" is at indices 0 and 6
- Substring "c" is at indices 2 and 8
- Substring "xab" is at index 5
- Substring "d" is at index 9
- Substring "cd" is at index 8

Can you see how to find all the occurrences of a pattern in a string ?

1. 1st of all, check if the given pattern really exists in string or not (As we did in Substring Check). For this, traverse the suffix tree against the pattern.
2. If you find pattern in suffix tree (don't fall off the tree), then traverse the subtree below that point and find all suffix indices on leaf nodes. All those suffix indices will be pattern indices in string

```
// A C program to implement Ukkonen's Suffix Tree Construction
// And find all locations of a pattern in string
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;

    /*(start, end) interval specifies the edge, by which the
     node is connected to its parent node. Each edge will
     connect two nodes,  one parent and one child, and
     (start, end) interval of a given edge  will be stored
     in the child node. Lets say there are two nods A and B
     connected by an edge with indices (5, 8) then this
     indices (5, 8) will be stored in node B. */
    int start;
    int *end;
```

```
    /*for leaf nodes, it stores the index of suffix for
      the path  from root to leaf*/
    int suffixIndex;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
  waiting for it's suffix link to be set, which might get
  a new suffix link (other than root) in next extension of
  same phase. lastNewNode will be set to NULL when last
  newly created internal node (if there is any) got it's
  suffix link reset to new internal node created in next
  extension of same phase. */
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represeted as input string character
  index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string

Node *newNode(int start, int *end)
{
    Node *node =(Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
          node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in  current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

    /*suffixIndex will be set to -1 by default and
      actual suffix index will be set later for leaves
      at the end of all phases*/
    node->suffixIndex = -1;
    return node;
```

```
}

int edgeLength(Node *n) {
    if(n == root)
        return 0;
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
     Skip/Count Trick  (Trick 1). If activeLength is greater
     than current edge length, set next  internal node as
     activeNode and adjust activeEdge and activeLength
     accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
    return 0;
}

void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
    leaves created so far in tree*/
    leafEnd = pos;

    /*Increment remainingSuffixCount indicating that a
    new suffix added to the list of suffixes yet to be
    added in tree*/
    remainingSuffixCount++;

    /*set lastNewNode to NULL while starting a new phase,
     indicating there is no internal node waiting for
     it's suffix link reset in current phase*/
    lastNewNode = NULL;

    //Add all suffixes (yet to be added) one by one in tree
    while(remainingSuffixCount > 0) {

        if (activeLength == 0)
            activeEdge = pos; //APCFALZ

        // There is no outgoing edge starting with
        // activeEdge from activeNode
        if (activeNode->children[text[activeEdge]] == NULL)
        {
            //Extension Rule 2 (A new leaf edge gets created)
            activeNode->children[text[activeEdge]] =
                                        newNode(pos, &leafEnd);
```

```
    /*A new leaf edge is created in above line starting
     from  an existng node (the current activeNode), and
     if there is any internal node waiting for it's suffix
     link get reset, point the suffix link from that last
     internal node to current activeNode. Then set lastNewNode
     to NULL indicating no more node waiting for suffix link
     reset.*/
    if (lastNewNode != NULL)
    {
        lastNewNode->suffixLink = activeNode;
        lastNewNode = NULL;
    }
}
// There is an outgoing edge starting with activeEdge
// from activeNode
else
{
    // Get the next node at the end of edge starting
    // with activeEdge
    Node *next = activeNode->children[text[activeEdge]];
    if (walkDown(next))//Do walkdown
    {
        //Start from next node (the new activeNode)
        continue;
    }
    /*Extension Rule 3 (current character being processed
      is already on the edge)*/
    if (text[next->start + activeLength] == text[pos])
    {
        //If a newly created node waiting for it's
        //suffix link to be set, then set suffix link
        //of that waiting node to curent active node
        if(lastNewNode != NULL && activeNode != root)
        {
            lastNewNode->suffixLink = activeNode;
            lastNewNode = NULL;
        }

        //APCFER3
        activeLength++;
        /*STOP all further processing in this phase
        and move on to next phase*/
        break;
    }

    /*We will be here when activePoint is in middle of
      the edge being traversed and current character
      being processed is not  on the edge (we fall off
      the tree). In this case, we add a new internal node
      and a new leaf edge going out of that new node. This
      is Extension Rule 2, where a new leaf edge and a new
    internal node get created*/
    splitEnd = (int*) malloc(sizeof(int));
```

```
            *splitEnd = next->start + activeLength - 1;

            //New internal node
            Node *split = newNode(next->start, splitEnd);
            activeNode->children[text[activeEdge]] = split;

            //New leaf coming out of new internal node
            split->children[text[pos]] = newNode(pos, &leafEnd);
            next->start += activeLength;
            split->children[text[next->start]] = next;

            /*We got a new internal node here. If there is any
              internal node created in last extensions of same
              phase which is still waiting for it's suffix link
              reset, do it now.*/
            if (lastNewNode != NULL)
            {
            /*suffixLink of lastNewNode points to current newly
              created internal node*/
                lastNewNode->suffixLink = split;
            }

            /*Make the current newly created internal node waiting
              for it's suffix link reset (which is pointing to root
              at present). If we come across any other internal node
              (existing or newly created) in next extension of same
              phase, when a new leaf edge gets added (i.e. when
              Extension Rule 2 applies is any of the next extension
              of same phase) at that point, suffixLink of this node
              will point to that internal node.*/
            lastNewNode = split;
        }

        /* One suffix got added in tree, decrement the count of
          suffixes yet to be added.*/
        remainingSuffixCount--;
        if (activeNode == root && activeLength > 0) //APCFER2C1
        {
            activeLength--;
            activeEdge = pos - remainingSuffixCount + 1;
        }
        else if (activeNode != root) //APCFER2C2
        {
            activeNode = activeNode->suffixLink;
        }
    }
}

void print(int i, int j)
{
    int k;
    for (k=i; k<=j; k++)
        printf("%c", text[k]);
}
```

```
//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL)  return;

    if (n->start != -1) //A non-root node
    {
        //Print the label on edge from parent to current node
        //Uncomment below line to print suffix tree
       // print(n->start, *(n->end));
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            //Uncomment below two lines to print suffix index
           // if (leaf == 1 && n->start != -1)
             //   printf(" [%d]\n", n->suffixIndex);

            //Current node is not a leaf as it has outgoing
            //edges from it.
            leaf = 0;
            setSuffixIndexByDFS(n->children[i], labelHeight +
                                edgeLength(n->children[i]));
        }
    }
    if (leaf == 1)
    {
        n->suffixIndex = size - labelHeight;
        //Uncomment below line to print suffix index
        //printf(" [%d]\n", n->suffixIndex);
    }
}

void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
```

```
}

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;

    /*Root is a special node with start and end indices as -1,
    as it has no parent from where an edge comes to root*/
    root = newNode(-1, rootEnd);

    activeNode = root; //First activeNode will be root
    for (i=0; i<size; i++)
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);
}

int traverseEdge(char *str, int idx, int start, int end)
{
    int k = 0;
    //Traverse the edge with character by character matching
    for(k=start; k<=end && str[idx] != '\0'; k++, idx++)
    {
        if(text[k] != str[idx])
            return -1;  // mo match
    }
    if(str[idx] == '\0')
        return 1;  // match
    return 0;  // more characters yet to match
}

int doTraversalToCountLeaf(Node *n)
{
    if(n == NULL)
        return 0;
    if(n->suffixIndex > -1)
    {
        printf("\nFound at position: %d", n->suffixIndex);
        return 1;
    }
    int count = 0;
    int i = 0;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if(n->children[i] != NULL)
        {
            count += doTraversalToCountLeaf(n->children[i]);
        }
```

```
    }
    return count;
}

int countLeaf(Node *n)
{
    if(n == NULL)
        return 0;
    return doTraversalToCountLeaf(n);
}

int doTraversal(Node *n, char* str, int idx)
{
    if(n == NULL)
    {
        return -1; // no match
    }
    int res = -1;
    //If node n is not root node, then traverse edge
    //from node n's parent to node n.
    if(n->start != -1)
    {
        res = traverseEdge(str, idx, n->start, *(n->end));
        if(res == -1)  //no match
            return -1;
        if(res == 1) //match
        {
            if(n->suffixIndex > -1)
                printf("\nsubstring count: 1 and position: %d",
                                n->suffixIndex);
            else
                printf("\nsubstring count: %d", countLeaf(n));
            return 1;
        }
    }
    //Get the character index to search
    idx = idx + edgeLength(n);
    //If there is an edge from node n going out
    //with current character str[idx], travrse that edge
    if(n->children[str[idx]] != NULL)
        return doTraversal(n->children[str[idx]], str, idx);
    else
        return -1;  // no match
}

void checkForSubString(char* str)
{
    int res = doTraversal(root, str, 0);
    if(res == 1)
        printf("\nPattern <%s> is a Substring\n", str);
    else
        printf("\nPattern <%s> is NOT a Substring\n", str);
}
```

```
// driver program to test above functions
int main(int argc, char *argv[])
{
    strcpy(text, "GEEKSFORGEEKS$");
    buildSuffixTree();
    printf("Text: GEEKSFORGEEKS, Pattern to search: GEEKS");
    checkForSubString("GEEKS");
    printf("\n\nText: GEEKSFORGEEKS, Pattern to search: GEEK1");
    checkForSubString("GEEK1");
    printf("\n\nText: GEEKSFORGEEKS, Pattern to search: FOR");
    checkForSubString("FOR");
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "AABAACAADAABAAABAA$");
    buildSuffixTree();
    printf("\n\nText: AABAACAADAABAAABAA, Pattern to search: AABA");
    checkForSubString("AABA");
    printf("\n\nText: AABAACAADAABAAABAA, Pattern to search: AA");
    checkForSubString("AA");
    printf("\n\nText: AABAACAADAABAAABAA, Pattern to search: AAE");
    checkForSubString("AAE");
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "AAAAAAAAA$");
    buildSuffixTree();
    printf("\n\nText: AAAAAAAAA, Pattern to search: AAAA");
    checkForSubString("AAAA");
    printf("\n\nText: AAAAAAAAA, Pattern to search: AA");
    checkForSubString("AA");
    printf("\n\nText: AAAAAAAAA, Pattern to search: A");
    checkForSubString("A");
    printf("\n\nText: AAAAAAAAA, Pattern to search: AB");
    checkForSubString("AB");
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    return 0;
}
```

Output:

```
Text: GEEKSFORGEEKS, Pattern to search: GEEKS
Found at position: 8
Found at position: 0
substring count: 2
Pattern <GEEKS> is a Substring


Text: GEEKSFORGEEKS, Pattern to search: GEEK1
Pattern <GEEK1> is NOT a Substring
```

```
Text: GEEKSFORGEEKS, Pattern to search: FOR
substring count: 1 and position: 5
Pattern <FOR> is a Substring


Text: AABAACAADAABAAABAA, Pattern to search: AABA
Found at position: 13
Found at position: 9
Found at position: 0
substring count: 3
Pattern <AABA> is a Substring


Text: AABAACAADAABAAABAA, Pattern to search: AA
Found at position: 16
Found at position: 12
Found at position: 13
Found at position: 9
Found at position: 0
Found at position: 3
Found at position: 6
substring count: 7
Pattern <AA> is a Substring


Text: AABAACAADAABAAABAA, Pattern to search: AAE
Pattern <AAE> is NOT a Substring


Text: AAAAAAAAA, Pattern to search: AAAA
Found at position: 5
Found at position: 4
Found at position: 3
Found at position: 2
Found at position: 1
Found at position: 0
substring count: 6
Pattern <AAAA> is a Substring


Text: AAAAAAAAA, Pattern to search: AA
Found at position: 7
Found at position: 6
Found at position: 5
Found at position: 4
Found at position: 3
Found at position: 2
Found at position: 1
Found at position: 0
substring count: 8
Pattern <AA> is a Substring
```

```
Text: AAAAAAAAA, Pattern to search: A
Found at position: 8
Found at position: 7
Found at position: 6
Found at position: 5
Found at position: 4
Found at position: 3
Found at position: 2
Found at position: 1
Found at position: 0
substring count: 9
Pattern <A> is a Substring


Text: AAAAAAAAA, Pattern to search: AB
Pattern <AB> is NOT a Substring
```

Ukkonen's Suffix Tree Construction takes O(N) time and space to build suffix tree for a string of length N and after that, traversal for substring check takes O(M) for a pattern of length M and then if there are Z occurrences of the pattern, it will take O(Z) to find indices of all those Z occurrences.
Overall pattern complexity is linear: O(M + Z).

A bit more detailed analysis
How many internal nodes will there in a suffix tree of string of length N ??
Answer: N-1 (Why ??)
There will be N suffixes in a string of length N.
Each suffix will have one leaf.
So a suffix tree of string of length N will have N leaves.
As each internal node has at least 2 children, an N-leaf suffix tree has at most N-1 internal nodes.
If a pattern occurs Z times in string, means it will be part of Z suffixes, so there will be Z leaves below in point (internal node and in between edge) where pattern match ends in tree and so subtree with Z leaves below that point will have Z-1 internal nodes. A tree with Z leaves can be traversed in O(Z) time.
Overall pattern complexity is linear: O(M + Z).
For a given pattern, Z (the number of occurrences) can be atmost N.
So worst case complexity can be: O(M + N) if Z is close/equal to N (A tree traversal with N nodes take O(N) time).

Followup questions:

1. Check if a pattern is prefix of a text?
2. Check if a pattern is suffix of a text?

We have published following more articles on suffix tree applications:

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/suffix-tree-application-2-searching-all-patterns/

Category: Strings Tags: Pattern Searching

Post navigation

← Perfect Binary Tree Specific Level Order Traversal MakeMyTrip Interview Experience | Set 2 (On-Campus) →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 25

Suffix Tree Application 3 - Longest Repeated Substring

Given a text string, find Longest Repeated Substring in the text. If there are more than one Longest Repeated Substrings, get any one of them.

```
Longest Repeated Substring in GEEKSFORGEEKS is: GEEKS
Longest Repeated Substring in AAAAAAAAAA is: AAAAAAAAA
Longest Repeated Substring in ABCDEFG is: No repeated substring
Longest Repeated Substring in ABABABA is: ABABA
Longest Repeated Substring in ATCGATCGA is: ATCGA
Longest Repeated Substring in banana is: ana
Longest Repeated Substring in abcpqrabpqpq is: ab (pq is another LRS here)
```

This problem can be solved by different approaches with varying time and space complexities. Here we will discuss Suffix Tree approach (3rd Suffix Tree Application). Other approaches will be discussed soon.

As a prerequisite, we must know how to build a suffix tree in one or the other way.
Here we will build suffix tree using Ukkonen's Algorithm, discussed already as below:

Ukkonen's Suffix Tree Construction – Part 1

Ukkonen's Suffix Tree Construction – Part 2

Ukkonen's Suffix Tree Construction – Part 3

Ukkonen's Suffix Tree Construction – Part 4

Ukkonen's Suffix Tree Construction – Part 5

Ukkonen's Suffix Tree Construction – Part 6

Lets look at following figure:

This is suffix tree for string “ABABABA$”.
In this string, following substrings are repeated:
A, B, AB, BA, ABA, BAB, ABAB, BABA, ABABA
And Longest Repeated Substring is ABABA.
In a suffix tree, one node can’t have more than one outgoing edge starting with same character, and so if there are repeated substring in the text, they will share on same path and that path in suffix tree will go through one or more internal node(s) down the tree (below the point where substring ends on that path).
In above figure, we can see that

- Path with Substring “A” has three internal nodes down the tree
- Path with Substring “AB” has two internal nodes down the tree
- Path with Substring “ABA” has two internal nodes down the tree
- Path with Substring “ABAB” has one internal node down the tree
- Path with Substring “ABABA” has one internal node down the tree
- Path with Substring “B” has two internal nodes down the tree
- Path with Substring “BA” has two internal nodes down the tree
- Path with Substring “BAB” has one internal node down the tree
- Path with Substring “BABA” has one internal node down the tree

All above substrings are repeated.

Substrings ABABAB, ABABABA, BABAB, BABABA have no internal node down the tree (after the point where substring end on the path), and so these are not repeated.

Can you see how to find longest repeated substring ??
We can see in figure that, longest repeated substring will end at the internal node which is farthest from the root (i.e. deepest node in the tree), because length of substring is the path label length from root to that internal node.

So finding longest repeated substring boils down to finding the deepest node in suffix tree and then get the path label from root to that deepest internal node.

```
// A C program to implement Ukkonen's Suffix Tree Construction
// And then find Longest Repeated Substring
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;

    /*(start, end) interval specifies the edge, by which the
     node is connected to its parent node. Each edge will
     connect two nodes,  one parent and one child, and
     (start, end) interval of a given edge  will be stored
     in the child node. Lets say there are two nods A and B
     connected by an edge with indices (5, 8) then this
     indices (5, 8) will be stored in node B. */
```

```
    int start;
    int *end;

    /*for leaf nodes, it stores the index of suffix for
      the path  from root to leaf*/
    int suffixIndex;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
  waiting for it's suffix link to be set, which might get
  a new suffix link (other than root) in next extension of
  same phase. lastNewNode will be set to NULL when last
  newly created internal node (if there is any) got it's
  suffix link reset to new internal node created in next
  extension of same phase. */
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represeted as input string character
  index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string

Node *newNode(int start, int *end)
{
    Node *node =(Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
          node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in  current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

    /*suffixIndex will be set to -1 by default and
      actual suffix index will be set later for leaves
      at the end of all phases*/
```

```
    node->suffixIndex = -1;
    return node;
}

int edgeLength(Node *n) {
    if(n == root)
        return 0;
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
     Skip/Count Trick  (Trick 1). If activeLength is greater
     than current edge length, set next  internal node as
     activeNode and adjust activeEdge and activeLength
     accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
    return 0;
}

void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
    leaves created so far in tree*/
    leafEnd = pos;

    /*Increment remainingSuffixCount indicating that a
    new suffix added to the list of suffixes yet to be
    added in tree*/
    remainingSuffixCount++;

    /*set lastNewNode to NULL while starting a new phase,
     indicating there is no internal node waiting for
     it's suffix link reset in current phase*/
    lastNewNode = NULL;

    //Add all suffixes (yet to be added) one by one in tree
    while(remainingSuffixCount > 0) {

        if (activeLength == 0)
            activeEdge = pos; //APCFALZ

        // There is no outgoing edge starting with
        // activeEdge from activeNode
        if (activeNode->children[text[activeEdge]] == NULL)
        {
            //Extension Rule 2 (A new leaf edge gets created)
```

```
    activeNode->children[text[activeEdge]] =
                                newNode(pos, &leafEnd);

    /*A new leaf edge is created in above line starting
     from  an existng node (the current activeNode), and
     if there is any internal node waiting for it's suffix
     link get reset, point the suffix link from that last
     internal node to current activeNode. Then set lastNewNode
     to NULL indicating no more node waiting for suffix link
     reset.*/
    if (lastNewNode != NULL)
    {
        lastNewNode->suffixLink = activeNode;
        lastNewNode = NULL;
    }
}
// There is an outgoing edge starting with activeEdge
// from activeNode
else
{
    // Get the next node at the end of edge starting
    // with activeEdge
    Node *next = activeNode->children[text[activeEdge]];
    if (walkDown(next))//Do walkdown
    {
        //Start from next node (the new activeNode)
        continue;
    }
    /*Extension Rule 3 (current character being processed
      is already on the edge)*/
    if (text[next->start + activeLength] == text[pos])
    {
        //If a newly created node waiting for it's
        //suffix link to be set, then set suffix link
        //of that waiting node to curent active node
        if(lastNewNode != NULL && activeNode != root)
        {
            lastNewNode->suffixLink = activeNode;
            lastNewNode = NULL;
        }

        //APCFER3
        activeLength++;
        /*STOP all further processing in this phase
        and move on to next phase*/
        break;
    }

    /*We will be here when activePoint is in middle of
      the edge being traversed and current character
      being processed is not  on the edge (we fall off
      the tree). In this case, we add a new internal node
      and a new leaf edge going out of that new node. This
      is Extension Rule 2, where a new leaf edge and a new
```

```
            internal node get created*/
            splitEnd = (int*) malloc(sizeof(int));
            *splitEnd = next->start + activeLength - 1;

            //New internal node
            Node *split = newNode(next->start, splitEnd);
            activeNode->children[text[activeEdge]] = split;

            //New leaf coming out of new internal node
            split->children[text[pos]] = newNode(pos, &leafEnd);
            next->start += activeLength;
            split->children[text[next->start]] = next;

            /*We got a new internal node here. If there is any
              internal node created in last extensions of same
              phase which is still waiting for it's suffix link
              reset, do it now.*/
            if (lastNewNode != NULL)
            {
            /*suffixLink of lastNewNode points to current newly
              created internal node*/
                lastNewNode->suffixLink = split;
            }

            /*Make the current newly created internal node waiting
              for it's suffix link reset (which is pointing to root
              at present). If we come across any other internal node
              (existing or newly created) in next extension of same
              phase, when a new leaf edge gets added (i.e. when
              Extension Rule 2 applies is any of the next extension
              of same phase) at that point, suffixLink of this node
              will point to that internal node.*/
            lastNewNode = split;
        }

        /* One suffix got added in tree, decrement the count of
          suffixes yet to be added.*/
        remainingSuffixCount--;
        if (activeNode == root && activeLength > 0) //APCFER2C1
        {
            activeLength--;
            activeEdge = pos - remainingSuffixCount + 1;
        }
        else if (activeNode != root) //APCFER2C2
        {
            activeNode = activeNode->suffixLink;
        }
    }
}

void print(int i, int j)
{
    int k;
    for (k=i; k<=j; k++)
```

```
        printf("%c", text[k]);
}

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL)  return;

    if (n->start != -1) //A non-root node
    {
        //Print the label on edge from parent to current node
        //Uncomment below line to print suffix tree
       // print(n->start, *(n->end));
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            //Uncomment below two lines to print suffix index
           // if (leaf == 1 && n->start != -1)
             //   printf(" [%d]\n", n->suffixIndex);

            //Current node is not a leaf as it has outgoing
            //edges from it.
            leaf = 0;
            setSuffixIndexByDFS(n->children[i], labelHeight +
                                edgeLength(n->children[i]));
        }
    }
    if (leaf == 1)
    {
        n->suffixIndex = size - labelHeight;
        //Uncomment below line to print suffix index
        //printf(" [%d]\n", n->suffixIndex);
    }
}

void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
```

```
        free(n->end);
    free(n);
}

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;

    /*Root is a special node with start and end indices as -1,
    as it has no parent from where an edge comes to root*/
    root = newNode(-1, rootEnd);

    activeNode = root; //First activeNode will be root
    for (i=0; i<size; i++)
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);
}

void doTraversal(Node *n, int labelHeight, int* maxHeight,
int* substringStartIndex)
{
    if(n == NULL)
    {
        return;
    }
    int i=0;
    if(n->suffixIndex == -1) //If it is internal node
    {
        for (i = 0; i < MAX_CHAR; i++)
        {
            if(n->children[i] != NULL)
            {
                doTraversal(n->children[i], labelHeight +
                                edgeLength(n->children[i]), maxHeight,
                                 substringStartIndex);
            }
        }
    }
    else if(n->suffixIndex > -1 &&
                (*maxHeight < labelHeight - edgeLength(n)))
    {
        *maxHeight = labelHeight - edgeLength(n);
        *substringStartIndex = n->suffixIndex;
    }
}

void getLongestRepeatedSubstring()
```

```
{
    int maxHeight = 0;
    int substringStartIndex = 0;
    doTraversal(root, 0, &maxHeight, &substringStartIndex);
//  printf("maxHeight %d, substringStartIndex %d\n", maxHeight,
//           substringStartIndex);
    printf("Longest Repeated Substring in %s is: ", text);
    int k;
    for (k=0; k<maxHeight; k++)
        printf("%c", text[k + substringStartIndex]);
    if(k == 0)
        printf("No repeated substring");
    printf("\n");
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    strcpy(text, "GEEKSFORGEEKS$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "AAAAAAAAAA$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "ABCDEFG$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "ABABABA$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "ATCGATCGA$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "banana$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
```

```
    strcpy(text, "abcpqrabpqpq$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "pqrpqpqabab$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    return 0;
}
```

Output:

```
Longest Repeated Substring in GEEKSFORGEEKS$ is: GEEKS
Longest Repeated Substring in AAAAAAAAAA$ is: AAAAAAAAA
Longest Repeated Substring in ABCDEFG$ is: No repeated substring
Longest Repeated Substring in ABABABA$ is: ABABA
Longest Repeated Substring in ATCGATCGA$ is: ATCGA
Longest Repeated Substring in banana$ is: ana
Longest Repeated Substring in abcpqrabpqpq$ is: ab
Longest Repeated Substring in pqrpqpqabab$ is: ab
```

In case of multiple LRS (As we see in last two test cases), this implementation prints the LRS which comes 1st lexicographically.

Ukkonen's Suffix Tree Construction takes O(N) time and space to build suffix tree for a string of length N and after that finding deepest node will take O(N).
So it is linear in time and space.

Followup questions:

1. Find all repeated substrings in given text
2. Find all unique substrings in given text
3. Find all repeated substrings of a given length
4. Find all unique substrings of a given length
5. In case of multiple LRS in text, find the one which occurs most number of times

All these problems can be solved in linear time with few changes in above implementation.

We have published following more articles on suffix tree applications:

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/suffix-tree-application-3-longest-repeated-substring/

Category: Strings Tags: Pattern Searching

Post navigation

← One97 Interview Experience | Set 2 KLA Tencor Interview Experience →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 26

Suffix Tree Application 6 - Longest Palindromic Substring

Given a string, find the longest substring which is palindrome.

We have already discussed Naïve [$O(n^3)$], quadratic [$O(n^2)$] and linear [O(n)] approaches in Set 1, Set 2 and Manacher's Algorithm.
In this article, we will discuss another linear time approach based on suffix tree.
If given string is S, then approach is following:

- Reverse the string S (say reversed string is R)
- Get Longest Common Substring of S and R **given that LCS in S and R must be from same position in S**

Can you see why we say that **LCS in R and S must be from same position in S** ?

Let's look at following examples:

- For S = *xababayz* and R = *zyababax*, LCS and LPS both are ababa (SAME)
- For S = *abacdfgdcaba* and R = *abacdgfdcaba*, LCS is *abacd* and LPS is *aba* (DIFFERENT)
- For S = *pqrqpabcdfgdcba* and R = *abcdgfdcbapqrqp*, LCS and LPS both are *pqrqp* (SAME)
- For S = *pqqpabcdfghfdcba* and R = *abcdfhgfdcbapqqp*, LCS is *abcdf* and LPS is *pqqp* (DIFFERENT)

We can see that LCS and LPS are not same always. When they are different ?
When S has a reversed copy of a non-palindromic substring in it which is of same or longer length than LPS in S, then LCS and LPS will be different.
In 2^{nd} example above (S = *abacdfgdcaba*), for substring *abacd*, there exists a reverse copy *dcaba* in S, which is of longer length than LPS *aba* and so LPS and LCS are different here. Same is the scenario in 4^{th} example.

To handle this scenario we say that LPS in S is same as LCS in S and R **given that LCS in R and S must be from same position in S**.
If we look at 2^{nd} example again, substring *aba* in R comes from exactly same position in S as substring *aba* in S which is ZERO (0^{th} index) and so this is LPS.

The Position Constraint:

(Click to see it clearly)

We will refer string S index as forward index (S_i) and string R index as reverse index (R_i).
Based on above figure, a character with index i (forward index) in a string S of length N, will be at index N-1-i (reverse index) in it's reversed string R.
If we take a substring of length L in string S with starting index i and ending index j (j = i+L-1), then in it's reversed string R, the reversed substring of the same will start at index N-1-j and will end at index N-1-i.
If there is a common substring of length L at indices S_i (forward index) and R_i (reverse index) in S and R, then these will come from same position in S if $\mathbf{R_i = (N - 1) - (S_i + L - 1)}$ where N is string length.

So to find LPS of string S, we find longest common string of S and R where both substrings satisfy above constraint, i.e. if substring in S is at index S_i, then same substring should be in R at index (N – 1) – (S_i + L – 1). If this is not the case, then this substring is not LPS candidate.

Naive [$O(N*M^2)$] and Dynamic Programming [O(N*M)] approaches to find LCS of two strings are already discussed here which can be extended to add position constraint to give LPS of a given string.

Now we will discuss suffix tree approach which is nothing but an extension to Suffix Tree LCS approach where we will add the position constraint.

While finding LCS of two strings X and Y, we just take deepest node marked as XY (i.e. the node which has suffixes from both strings as it's children).
While finding LPS of string S, we will again find LCS of S and R with a condition that the common substring should satisfy the position constraint (the common substring should come from same position in S). To verify position constraint, we need to know all forward and reverse indices on each internal node (i.e. the suffix indices of all leaf children below the internal nodes).

In Generalized Suffix Tree of *S#R$*, a substring on the path from root to an internal node is a common substring if the internal node has suffixes from both strings S and R. The index of the common substring in S and R can be found by looking at suffix index at respective leaf node.
If string *S#* is of length N then:

- If suffix index of a leaf is less than N, then that suffix belongs to S and same suffix index will become forward index of all ancestor nodes
- If suffix index of a leaf is greater than N, then that suffix belongs to R and reverse index for all ancestor nodes will be **N − suffix index**

Let's take string S = *cabbaabb*. The figure below is Generalized Suffix Tree for *cabbaabb#bbaabbac$* where we have shown forward and reverse indices of all children suffixes on all internal nodes (except root).
Forward indices are in Parentheses () and reverse indices are in square bracket [].

(Click to see it clearly)

In above figure, all leaf nodes will have one forward or reverse index depending on which string (S or R) they belong to. Then children's forward or reverse indices propagate to the parent.

Look at the figure to understand what would be the forward or reverse index on a leaf with a given suffix index. At the bottom of figure, it is shown that leaves with suffix indices from 0 to 8 will get same values (0 to 8) as their forward index in S and leaves with suffix indices 9 to 17 will get reverse index in R from 0 to 8.

For example, the highlighted internal node has two children with suffix indices 2 and 9. Leaf with suffix index 2 is from position 2 in S and so it's forward index is 2 and shown in (). Leaf with suffix index 9 is from position 0 in R and so it's reverse index is 0 and shown in []. These indices propagate to parent and the parent has one leaf with suffix index 14 for which reverse index is 4. So on this parent node forward index is (2) and reverse index is [0,4]. And in same way, we should be able to understand the how forward and reverse indices are calculated on all nodes.

In above figure, all internal nodes have suffixes from both strings S and R, i.e. all of them represent a common substring on the path from root to themselves. Now we need to find deepest node satisfying

position constraint. For this, we need to check if there is a forward index S_i on a node, then there must be a reverse index R_i with value (N – 2) – (S_i + L – 1) where N is length of string *S#* and L is node depth (or substring length). If yes, then consider this node as a LPS candidate, else ignore it. In above figure, deepest node is highlighted which represents LPS as bbaabb.

We have not shown forward and reverse indices on root node in figure. Because root node itself doesn't represent any common substring (In code implementation also, forward and reverse indices will not be calculated on root node)

How to implement this apprach to find LPS? Here are the things that we need:

- We need to know forward and reverse indices on each node.
- For a given forward index S_i on an internal node, we need know if reverse index $\mathbf{R_i = (N - 2) - (S_i + L - 1)}$ also present on same node.
- Keep track of deepest internal node satisfying above condition.

One way to do above is:
While DFS on suffix tree, we can store forward and reverse indices on each node in some way (storage will help to avoid repeated traversals on tree when we need to know forward and reverse indices on a node). Later on, we can do another DFS to look for nodes satisfying position constraint. For position constraint check, we need to search in list of indices.
What data structure is suitable here to do all these in quickest way ?

- If we store indices in array, it will require linear search which will make overall approach non-linear in time.
- If we store indices in tree (set in C++, TreeSet in Java), we may use binary search but still overall approach will be non-linear in time.
- If we store indices in hash function based set (unordered_set in C++, HashSet in Java), it will provide a constant search on average and this will make overall approach linear in time. *A hash function based set may take more space depending on values being stored.*

We will use two unordered_set (one for forward and other from reverse indices) in our implementation, added as a member variable in SuffixTreeNode structure.

```
// A C++ program to implement Ukkonen's Suffix Tree Construction
// Here we build generalized suffix tree for given string S
// and it's reverse R, then we find
// longest palindromic substring of given string S
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <iostream>
#include <unordered_set>
#define MAX_CHAR 256
using namespace std;

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;

    /*(start, end) interval specifies the edge, by which the
```

```
     node is connected to its parent node. Each edge will
     connect two nodes,  one parent and one child, and
     (start, end) interval of a given edge  will be stored
     in the child node. Lets say there are two nods A and B
     connected by an edge with indices (5, 8) then this
     indices (5, 8) will be stored in node B. */
    int start;
    int *end;

    /*for leaf nodes, it stores the index of suffix for
      the path  from root to leaf*/
    int suffixIndex;

    //To store indices of children suffixes in given string
    unordered_set<int> *forwardIndices;

    //To store indices of children suffixes in reversed string
    unordered_set<int> *reverseIndices;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
  waiting for it's suffix link to be set, which might get
  a new suffix link (other than root) in next extension of
  same phase. lastNewNode will be set to NULL when last
  newly created internal node (if there is any) got it's
  suffix link reset to new internal node created in next
  extension of same phase. */
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represeted as input string character
  index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string
int size1 = 0; //Size of 1st string
int reverseIndex; //Index of a suffix in reversed string
unordered_set<int>::iterator forwardIndex;

Node *newNode(int start, int *end)
{
    Node *node =(Node*) malloc(sizeof(Node));
```

```
    int i;
    for (i = 0; i < MAX_CHAR; i++)
          node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in  current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

    /*suffixIndex will be set to -1 by default and
      actual suffix index will be set later for leaves
      at the end of all phases*/
    node->suffixIndex = -1;
    node->forwardIndices = new unordered_set<int>;
    node->reverseIndices = new unordered_set<int>;
    return node;
}

int edgeLength(Node *n) {
    if(n == root)
        return 0;
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
     Skip/Count Trick  (Trick 1). If activeLength is greater
     than current edge length, set next  internal node as
     activeNode and adjust activeEdge and activeLength
     accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
    return 0;
}

void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
    leaves created so far in tree*/
    leafEnd = pos;

    /*Increment remainingSuffixCount indicating that a
    new suffix added to the list of suffixes yet to be
    added in tree*/
    remainingSuffixCount++;
```

```
/*set lastNewNode to NULL while starting a new phase,
 indicating there is no internal node waiting for
 it's suffix link reset in current phase*/
lastNewNode = NULL;

//Add all suffixes (yet to be added) one by one in tree
while(remainingSuffixCount > 0) {

    if (activeLength == 0)
        activeEdge = pos; //APCFALZ

    // There is no outgoing edge starting with
    // activeEdge from activeNode
    if (activeNode->children[text[activeEdge]]  == NULL)
    {
        //Extension Rule 2 (A new leaf edge gets created)
        activeNode->children[text[activeEdge]]  =
                                    newNode(pos, &leafEnd);

        /*A new leaf edge is created in above line starting
         from  an existng node (the current activeNode), and
         if there is any internal node waiting for it's suffix
         link get reset, point the suffix link from that last
         internal node to current activeNode. Then set lastNewNode
         to NULL indicating no more node waiting for suffix link
         reset.*/
        if (lastNewNode != NULL)
        {
            lastNewNode->suffixLink = activeNode;
            lastNewNode = NULL;
        }
    }
    // There is an outgoing edge starting with activeEdge
    // from activeNode
    else
    {
        // Get the next node at the end of edge starting
        // with activeEdge
        Node *next = activeNode->children[text[activeEdge]] ;
        if (walkDown(next))//Do walkdown
        {
            //Start from next node (the new activeNode)
            continue;
        }
        /*Extension Rule 3 (current character being processed
          is already on the edge)*/
        if (text[next->start + activeLength] == text[pos])
        {
            //APCFER3
            activeLength++;
            /*STOP all further processing in this phase
            and move on to next phase*/
            break;
```

```
    }

    /*We will be here when activePoint is in middle of
      the edge being traversed and current character
      being processed is not  on the edge (we fall off
      the tree). In this case, we add a new internal node
      and a new leaf edge going out of that new node. This
      is Extension Rule 2, where a new leaf edge and a new
    internal node get created*/
    splitEnd = (int*) malloc(sizeof(int));
    *splitEnd = next->start + activeLength - 1;

    //New internal node
    Node *split = newNode(next->start, splitEnd);
    activeNode->children[text[activeEdge]]  = split;

    //New leaf coming out of new internal node
    split->children[text[pos]] = newNode(pos, &leafEnd);
    next->start += activeLength;
    split->children[text[next->start]]  = next;

    /*We got a new internal node here. If there is any
      internal node created in last extensions of same
      phase which is still waiting for it's suffix link
      reset, do it now.*/
    if (lastNewNode != NULL)
    {
    /*suffixLink of lastNewNode points to current newly
      created internal node*/
        lastNewNode->suffixLink = split;
    }

    /*Make the current newly created internal node waiting
      for it's suffix link reset (which is pointing to root
      at present). If we come across any other internal node
      (existing or newly created) in next extension of same
      phase, when a new leaf edge gets added (i.e. when
      Extension Rule 2 applies is any of the next extension
      of same phase) at that point, suffixLink of this node
      will point to that internal node.*/
    lastNewNode = split;
}

/* One suffix got added in tree, decrement the count of
  suffixes yet to be added.*/
remainingSuffixCount--;
if (activeNode == root && activeLength > 0) //APCFER2C1
{
    activeLength--;
    activeEdge = pos - remainingSuffixCount + 1;
}
else if (activeNode != root) //APCFER2C2
{
    activeNode = activeNode->suffixLink;
```

```
        }
    }
}

void print(int i, int j)
{
    int k;
    for (k=i; k<=j && text[k] != '#'; k++)
        printf("%c", text[k]);
    if(k<=j)
        printf("#");
}

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL)  return;

    if (n->start != -1) //A non-root node
    {
        //Print the label on edge from parent to current node
        //Uncomment below line to print suffix tree
        //print(n->start, *(n->end));
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            //Uncomment below two lines to print suffix index
         //   if (leaf == 1 && n->start != -1)
           //     printf(" [%d]\n", n->suffixIndex);

            //Current node is not a leaf as it has outgoing
            //edges from it.
            leaf = 0;
            setSuffixIndexByDFS(n->children[i], labelHeight +
                                edgeLength(n->children[i]));
            if(n != root)
            {
                //Add chldren's suffix indices in parent
                n->forwardIndices->insert(
                    n->children[i]->forwardIndices->begin(),
                    n->children[i]->forwardIndices->end());
                n->reverseIndices->insert(
                    n->children[i]->reverseIndices->begin(),
                    n->children[i]->reverseIndices->end());
            }
        }
    }
    if (leaf == 1)
```

```
    {
        for(i= n->start; i<= *(n->end); i++)
        {
            if(text[i] == '#')
            {
                n->end = (int*) malloc(sizeof(int));
                *(n->end) = i;
            }
        }
        n->suffixIndex = size - labelHeight;

        if(n->suffixIndex < size1) //Suffix of Given String
            n->forwardIndices->insert(n->suffixIndex);
        else //Suffix of Reversed String
            n->reverseIndices->insert(n->suffixIndex - size1);

        //Uncomment below line to print suffix index
       // printf(" [%d]\n", n->suffixIndex);
    }
}

void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;

    /*Root is a special node with start and end indices as -1,
    as it has no parent from where an edge comes to root*/
    root = newNode(-1, rootEnd);

    activeNode = root; //First activeNode will be root
    for (i=0; i<size; i++)
```

```
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);
}

void doTraversal(Node *n, int labelHeight, int* maxHeight,
int* substringStartIndex)
{
    if(n == NULL)
    {
        return;
    }
    int i=0;
    int ret = -1;
    if(n->suffixIndex < 0) //If it is internal node
    {
        for (i = 0; i < MAX_CHAR; i++)
        {
            if(n->children[i] != NULL)
            {
                doTraversal(n->children[i], labelHeight +
                    edgeLength(n->children[i]),
                    maxHeight, substringStartIndex);

                if(*maxHeight < labelHeight
                    && n->forwardIndices->size() > 0 &&
                    n->reverseIndices->size() > 0)
                {
                    for (forwardIndex=n->forwardIndices->begin();
                            forwardIndex!=n->forwardIndices->end();
                            ++forwardIndex)
                    {
                        reverseIndex = (size1 - 2) -
                            (*forwardIndex + labelHeight - 1);
                        //If reverse suffix comes from
                        //SAME position in given string
                        //Keep track of deepest node
                        if(n->reverseIndices->find(reverseIndex) !=
                            n->reverseIndices->end())
                        {
                            *maxHeight = labelHeight;
                            *substringStartIndex = *(n->end) -
                                labelHeight + 1;
                            break;
                        }
                    }
                }
            }
        }
    }
}

void getLongestPalindromicSubstring()
{
```

```
    int maxHeight = 0;
    int substringStartIndex = 0;
    doTraversal(root, 0, &maxHeight, &substringStartIndex);

    int k;
    for (k=0; k<maxHeight; k++)
        printf("%c", text[k + substringStartIndex]);
    if(k == 0)
        printf("No palindromic substring");
    else
        printf(", of length: %d",maxHeight);
    printf("\n");
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    size1 = 9;
    printf("Longest Palindromic Substring in cabbaabb is: ");
    strcpy(text, "cabbaabb#bbaabbac$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 17;
    printf("Longest Palindromic Substring in forgeeksskeegfor is: ");
    strcpy(text, "forgeeksskeegfor#rofgeeksskeegrof$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 6;
    printf("Longest Palindromic Substring in abcde is: ");
    strcpy(text, "abcde#edcba$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 7;
    printf("Longest Palindromic Substring in abcdae is: ");
    strcpy(text, "abcdae#eadcba$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 6;
    printf("Longest Palindromic Substring in abacd is: ");
    strcpy(text, "abacd#dcaba$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 6;
    printf("Longest Palindromic Substring in abcdc is: ");
```

```
    strcpy(text, "abcdc#cdcba$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 13;
    printf("Longest Palindromic Substring in abacdfgdcaba is: ");
    strcpy(text, "abacdfgdcaba#abacdgfdcaba$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 15;
    printf("Longest Palindromic Substring in xyabacdfgdcaba is: ");
    strcpy(text, "xyabacdfgdcaba#abacdgfdcabayx$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 9;
    printf("Longest Palindromic Substring in xababayz is: ");
    strcpy(text, "xababayz#zyababax$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 6;
    printf("Longest Palindromic Substring in xabax is: ");
    strcpy(text, "xabax#xabax$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    return 0;
}
```

Output:

```
Longest Palindromic Substring in cabbaabb is: bbaabb, of length: 6
Longest Palindromic Substring in forgeeksskeegfor is: geeksskeeg, of length: 10
Longest Palindromic Substring in abcde is: a, of length: 1
Longest Palindromic Substring in abcdae is: a, of length: 1
Longest Palindromic Substring in abacd is: aba, of length: 3
Longest Palindromic Substring in abcdc is: cdc, of length: 3
Longest Palindromic Substring in abacdfgdcaba is: aba, of length: 3
Longest Palindromic Substring in xyabacdfgdcaba is: aba, of length: 3
Longest Palindromic Substring in xababayz is: ababa, of length: 5
Longest Palindromic Substring in xabax is: xabax, of length: 5
```

Followup:

Detect ALL palindromes in a given string.
e.g. For string abcddcbefgf, all possible palindromes are a, b, c, d, e, f, g, dd, fgf, cddc, bcddcb.

We have published following more articles on suffix tree applications:

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/suffix-tree-application-6-longest-palindromic-substring/

Category: Strings

Post navigation

← Given a binary string, count number of substrings that start and end with 1. Zoho Interview | Set 4 →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 27

AVL Tree | Set 1 (Insertion)

AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

Why AVL Trees?
Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that height of the tree remains O(Logn) after every insertion and deletion, then we can guarantee an upper bound of O(Logn) for all these operations. The height of an AVL tree is always O(Logn) where n is the number of nodes in the tree (See thisvideo lecture for proof).

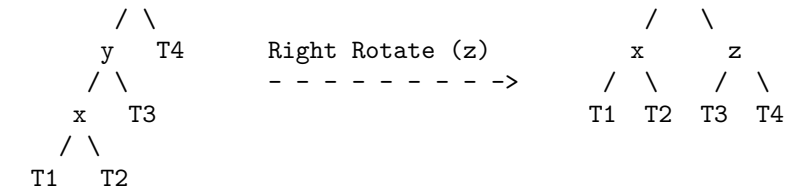
Insertion
To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys(left) T1, T2 and T3 are subtrees of the tree rooted with y (on left side) or x (on right side) y x / \ Right Rotation / \ x T3 – – – – – – – > T1 y / \

Steps to follow for insertion
Let the newly inserted node be w
1) Perform standard BST insert for w.
2) Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z.
3) Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:
a) y is left child of z and x is left child of y (Left Left Case)
b) y is left child of z and x is right child of y (Left Right Case)
c) y is right child of z and x is right child of y (Right Right Case)
d) y is right child of z and x is left child of y (Right Left Case)

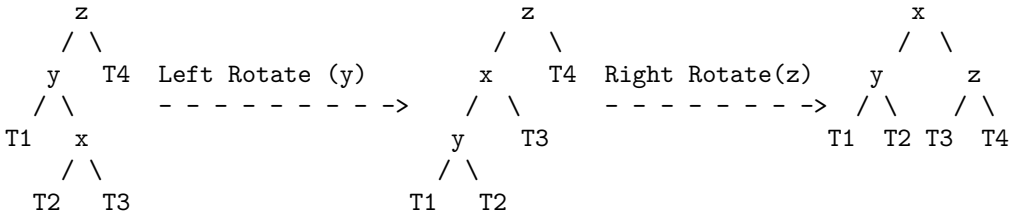
Following are the operations to be performed in above mentioned 4 cases. In all of the cases, we only need to re-balance the subtree rooted with z and the complete tree becomes balanced as the height of subtree (After appropriate rotations) rooted with z becomes same as it was before insertion. (See thisvideo lecture for proof)

a) Left Left Case

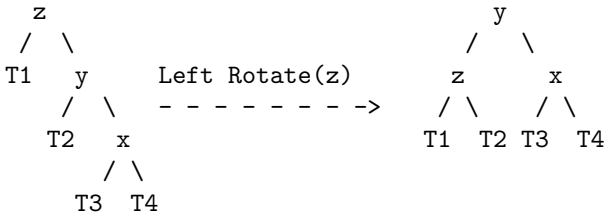
```
T1, T2, T3 and T4 are subtrees.
         z                                      y
```



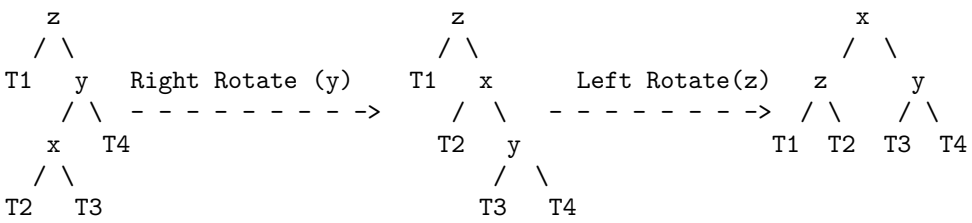
b) Left Right Case



c) Right Right Case



d) Right Left Case



C implementation

Following is the C implementation for AVL Tree Insertion. The following C implementation uses the recursive BST insert to insert a new node. In the recursive BST insert, after insertion, we get pointers to all ancestors one by one in bottom up manner. So we don't need parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of the newly inserted node.

1) Perform the normal BST insertion.

2) The current node must be one of the ancestors of the newly inserted node. Update the height of the current node.

3) Get the balance factor (left subtree height – right subtree height) of the current node.
4) If balance factor is greater than 1, then the current node is unbalanced and we are either in Left Left case or left Right case. To check whether it is left left case or not, compare the newly inserted key with the key in left subtree root.
5) If balance factor is less than -1, then the current node is unbalanced and we are either in Right Right case or Right Left case. To check whether it is Right Right case or not, compare the newly inserted key with the key in right subtree root.

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

// An AVL tree node
struct node
{
    int key;
    struct node *left;
    struct node *right;
    int height;
};

// A utility function to get maximum of two integers
int max(int a, int b);

// A utility function to get height of the tree
int height(struct node *N)
{
    if (N == NULL)
        return 0;
    return N->height;
}

// A utility function to get maximum of two integers
int max(int a, int b)
{
    return (a > b)? a : b;
}

/* 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;
    node->height = 1;  // new node is initially added at leaf
    return(node);
}

// A utility function to right rotate subtree rooted with y
// See the diagram given above.
struct node *rightRotate(struct node *y)
```

```
{
    struct node *x = y->left;
    struct node *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left), height(y->right))+1;
    x->height = max(height(x->left), height(x->right))+1;

    // Return new root
    return x;
}

// A utility function to left rotate subtree rooted with x
// See the diagram given above.
struct node *leftRotate(struct node *x)
{
    struct node *y = x->right;
    struct node *T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    //  Update heights
    x->height = max(height(x->left), height(x->right))+1;
    y->height = max(height(y->left), height(y->right))+1;

    // Return new root
    return y;
}

// Get Balance factor of node N
int getBalance(struct node *N)
{
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}

struct node* insert(struct node* node, int key)
{
    /* 1.  Perform the normal BST rotation */
    if (node == NULL)
        return(newNode(key));

    if (key < node->key)
        node->left  = insert(node->left, key);
    else
        node->right = insert(node->right, key);
```

```
    /* 2. Update height of this ancestor node */
    node->height = max(height(node->left), height(node->right)) + 1;

    /* 3. Get the balance factor of this ancestor node to check whether
       this node became unbalanced */
    int balance = getBalance(node);

    // If this node becomes unbalanced, then there are 4 cases

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node->left->key)
    {
        node->left =  leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key)
    {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    /* return the (unchanged) node pointer */
    return node;
}

// A utility function to print preorder traversal of the tree.
// The function also prints height of every node
void preOrder(struct node *root)
{
    if(root != NULL)
    {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

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

  /* Constructing tree given in the above figure */
  root = insert(root, 10);
```

```
  root = insert(root, 20);
  root = insert(root, 30);
  root = insert(root, 40);
  root = insert(root, 50);
  root = insert(root, 25);

  /* The constructed AVL Tree would be
            30
           /  \
         20   40
        /  \     \
       10  25    50
  */

  printf("Pre order traversal of the constructed AVL tree is \n");
  preOrder(root);

  return 0;
}
```

Output:

```
  Pre order traversal of the constructed AVL tree is
  30 20 10 25 40 50
```

Time Complexity: The rotation operations (left and right rotate) take constant time as only few pointers are being changed there. Updating the height and getting the balance factor also take constant time. So the time complexity of AVL insert remains same as BST insert which is O(h) where h is height of the tree. Since AVL tree is balanced, the height is O(Logn). So time complexity of AVL insert is O(Logn).

The AVL tree and other self balancing search trees like Red Black are useful to get all basic operations done in O(Logn) time. The AVL trees are more balanced compared to Red Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves many frequent insertions and deletions, then Red Black trees should be preferred. And if the insertions and deletions are less frequent and search is more frequent operation, then AVL tree should be preferred over Red Black Tree.

Following is the post for delete.
AVL Tree | Set 2 (Deletion)

Following are some previous posts that have used self-balancing search trees.

Median in a stream of integers (running integers)
Maximum of all subarrays of size k
Count smaller elements on right side

References:
IITD Video Lecture on AVL Tree Introduction
IITD Video Lecture on AVL Tree Insertion and Deletion

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/avl-tree-set-1-insertion/

Chapter 28

AVL Tree | Set 2 (Deletion)

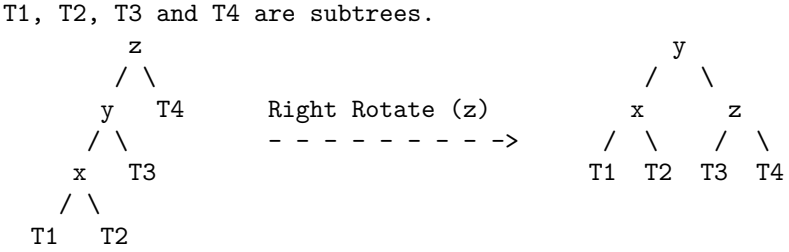
We have discussed AVL insertion in the previous post. In this post, we will follow a similar approach for deletion.

Steps to follow for deletion.
To make sure that the given tree remains AVL after every deletion, we must augment the standard BST delete operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys(left) T1, T2 and T3 are subtrees of the tree rooted with y (on left side) or x (on right side) y x / \ Right Rotation / \ x T3 – – – – – – – > T1 y / \

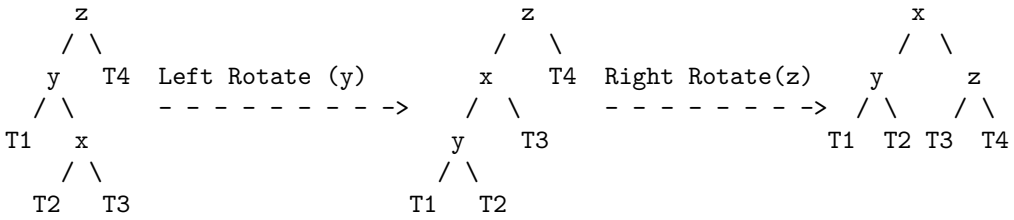
Let w be the node to be deleted
1) Perform standard BST delete for w.
2) Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the larger height child of z, and x be the larger height child of y. Note that the definitions of x and y are different from insertionhere.
3) Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:
a) y is left child of z and x is left child of y (Left Left Case)
b) y is left child of z and x is right child of y (Left Right Case)
c) y is right child of z and x is right child of y (Right Right Case)
d) y is right child of z and x is left child of y (Right Left Case)

Like insertion, following are the operations to be performed in above mentioned 4 cases. Note that, unlike insertion, fixing the node z won't fix the complete AVL tree. After fixing z, we may have to fix ancestors of z as well (See this video lecture for proof)

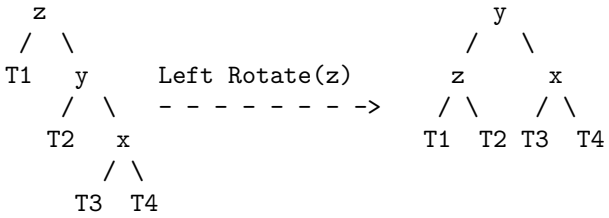
a) Left Left Case



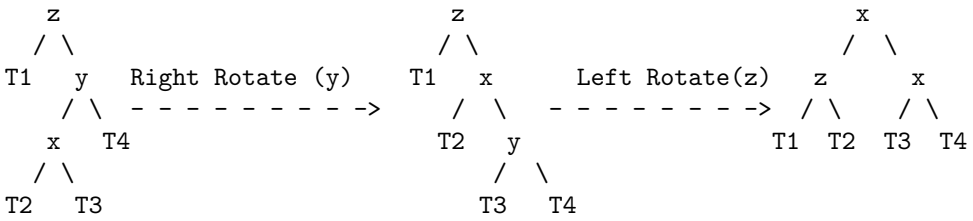
b) Left Right Case



c) Right Right Case



d) Right Left Case



Unlike insertion, in deletion, after we perform a rotation at z, we may have to perform a rotation at ancestors of z. Thus, we must continue to trace the path until we reach the root.

C implementation

Following is the C implementation for AVL Tree Deletion. The following C implementation uses the recursive BST delete as basis. In the recursive BST delete, after deletion, we get pointers to all ancestors one by one in bottom up manner. So we don't need parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of the deleted node.

1) Perform the normal BST deletion.

2) The current node must be one of the ancestors of the deleted node. Update the height of the current node.

3) Get the balance factor (left subtree height – right subtree height) of the current node.

4) If balance factor is greater than 1, then the current node is unbalanced and we are either in Left Left case or Left Right case. To check whether it is Left Left case or Left Right case, get the balance factor of left subtree. If balance factor of the left subtree is greater than or equal to 0, then it is Left Left case, else Left Right case.

5) If balance factor is less than -1, then the current node is unbalanced and we are either in Right Right case or Right Left case. To check whether it is Right Right case or Right Left case, get the balance factor of right subtree. If the balance factor of the right subtree is smaller than or equal to 0, then it is Right Right case, else Right Left case.

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

// An AVL tree node
struct node
{
    int key;
    struct node *left;
    struct node *right;
    int height;
};

// A utility function to get maximum of two integers
int max(int a, int b);

// A utility function to get height of the tree
int height(struct node *N)
{
    if (N == NULL)
        return 0;
    return N->height;
}

// A utility function to get maximum of two integers
int max(int a, int b)
{
    return (a > b)? a : b;
}

/* 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;
    node->height = 1;  // new node is initially added at leaf
    return(node);
}

// A utility function to right rotate subtree rooted with y
// See the diagram given above.
struct node *rightRotate(struct node *y)
{
    struct node *x = y->left;
    struct node *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;
```

```
    // Update heights
    y->height = max(height(y->left), height(y->right))+1;
    x->height = max(height(x->left), height(x->right))+1;

    // Return new root
    return x;
}

// A utility function to left rotate subtree rooted with x
// See the diagram given above.
struct node *leftRotate(struct node *x)
{
    struct node *y = x->right;
    struct node *T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    //  Update heights
    x->height = max(height(x->left), height(x->right))+1;
    y->height = max(height(y->left), height(y->right))+1;

    // Return new root
    return y;
}

// Get Balance factor of node N
int getBalance(struct node *N)
{
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}

struct node* insert(struct node* node, int key)
{
    /* 1.  Perform the normal BST rotation */
    if (node == NULL)
        return(newNode(key));

    if (key < node->key)
        node->left  = insert(node->left, key);
    else
        node->right = insert(node->right, key);

    /* 2. Update height of this ancestor node */
    node->height = max(height(node->left), height(node->right)) + 1;

    /* 3. Get the balance factor of this ancestor node to check whether
       this node became unbalanced */
    int balance = getBalance(node);

    // If this node becomes unbalanced, then there are 4 cases
```

```
    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node->left->key)
    {
        node->left =  leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key)
    {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    /* return the (unchanged) node pointer */
    return node;
}

/* Given a non-empty binary search tree, return the node with minimum
   key value found in that tree. Note that the entire tree does not
   need to be searched. */
struct node * minValueNode(struct node* node)
{
    struct node* current = node;

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

    return current;
}

struct node* deleteNode(struct node* root, int key)
{
    // STEP 1: PERFORM STANDARD BST DELETE

    if (root == NULL)
        return root;

    // If the key to be deleted is smaller than the root's key,
    // then it lies in left subtree
    if ( key < root->key )
        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater than the root's key,
```

```
// then it lies in right subtree
else if( key > root->key )
    root->right = deleteNode(root->right, key);

// if key is same as root's key, then This is the node
// to be deleted
else
{
    // node with only one child or no child
    if( (root->left == NULL) || (root->right == NULL) )
    {
        struct node *temp = root->left ? root->left : root->right;

        // No child case
        if(temp == NULL)
        {
            temp = root;
            root = NULL;
        }
        else // One child case
         *root = *temp; // Copy the contents of the non-empty child

        free(temp);
    }
    else
    {
        // node with two children: Get the inorder successor (smallest
        // in the right subtree)
        struct node* temp = minValueNode(root->right);

        // Copy the inorder successor's data to this node
        root->key = temp->key;

        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->key);
    }
}

// If the tree had only one node then return
if (root == NULL)
  return root;

// STEP 2: UPDATE HEIGHT OF THE CURRENT NODE
root->height = max(height(root->left), height(root->right)) + 1;

// STEP 3: GET THE BALANCE FACTOR OF THIS NODE (to check whether
//  this node became unbalanced)
int balance = getBalance(root);

// If this node becomes unbalanced, then there are 4 cases

// Left Left Case
if (balance > 1 && getBalance(root->left) >= 0)
    return rightRotate(root);
```

```
    // Left Right Case
    if (balance > 1 && getBalance(root->left) < 0)
    {
        root->left =  leftRotate(root->left);
        return rightRotate(root);
    }

    // Right Right Case
    if (balance < -1 && getBalance(root->right) <= 0)
        return leftRotate(root);

    // Right Left Case
    if (balance < -1 && getBalance(root->right) > 0)
    {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}

// A utility function to print preorder traversal of the tree.
// The function also prints height of every node
void preOrder(struct node *root)
{
    if(root != NULL)
    {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

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

  /* Constructing tree given in the above figure */
    root = insert(root, 9);
    root = insert(root, 5);
    root = insert(root, 10);
    root = insert(root, 0);
    root = insert(root, 6);
    root = insert(root, 11);
    root = insert(root, -1);
    root = insert(root, 1);
    root = insert(root, 2);

    /* The constructed AVL Tree would be
            9
           /  \
          1    10
```

```
        /  \     \
       0    5     11
      /    /  \
     -1   2    6
    */

    printf("Pre order traversal of the constructed AVL tree is \n");
    preOrder(root);

    root = deleteNode(root, 10);

    /* The AVL Tree after deletion of 10
            1
           /  \
          0    9
         /     /  \
        -1    5     11
            /  \
           2    6
    */

    printf("\nPre order traversal after deletion of 10 \n");
    preOrder(root);

    return 0;
}
```

Output:

```
Pre order traversal of the constructed AVL tree is
9 1 0 -1 5 2 6 10 11
Pre order traversal after deletion of 10
1 0 -1 9 5 2 6 11
```

Time Complexity: The rotation operations (left and right rotate) take constant time as only few pointers are being changed there. Updating the height and getting the balance factor also take constant time. So the time complexity of AVL delete remains same as BST delete which is O(h) where h is height of the tree. Since AVL tree is balanced, the height is O(Logn). So time complexity of AVL delete is O(Logn).

References:
IITD Video Lecture on AVL Tree Insertion and Deletion

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/avl-tree-set-2-deletion/

Category: Trees Tags: Advance Data Structures

Post navigation

← Vertical Sum in a given Binary Tree Operating Systems | Set 5 →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 29

Splay Tree | Set 1 (Search)

The worst case time complexity of Binary Search Tree (BST) operations like search, delete, insert is O(n). The worst case occurs when the tree is skewed. We can get the worst case time complexity as O(Logn) with AVL and Red-Black Trees.

Can we do better than AVL or Red-Black trees in practical situations?
Like AVLand Red-Black Trees, Splay tree is also self-balancing BST. The main idea of splay tree is to bring the recently accessed item to root of the tree, this makes the recently searched item to be accessible in O(1) time if accessed again. The idea is to use locality of reference (In a typical application, 80% of the access are to 20% of the items). Imagine a situation where we have millions or billions of keys and only few of them are accessed frequently, which is very likely in many practical applications.

All splay tree operations run in O(log n) time on average, where n is the number of entries in the tree. Any single operation can take Theta(n) time in the worst case.

Search Operation
The search operation in Splay tree does the standard BST search, in addition to search, it also splays (move a node to the root). If the search is successful, then the node that is found is splayed and becomes the new root. Else the last node accessed prior to reaching the NULL is splayed and becomes the new root.

There are following cases for the node being accessed.

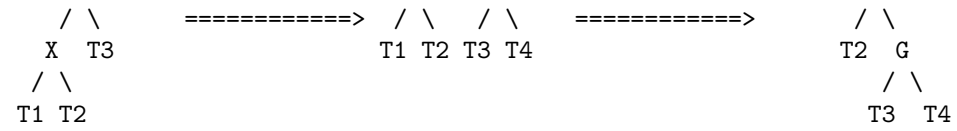
1) Node is root We simply return the root, don't do anything else as the accessed node is already root.

2) Zig: ***Node is child of root*** (the node has no grandparent). Node is either a left child of root (we do a right rotation) or node is a right child of its parent (we do a left rotation).
T1, T2 and T3 are subtrees of the tree rooted with y (on left side) or x (on right side)

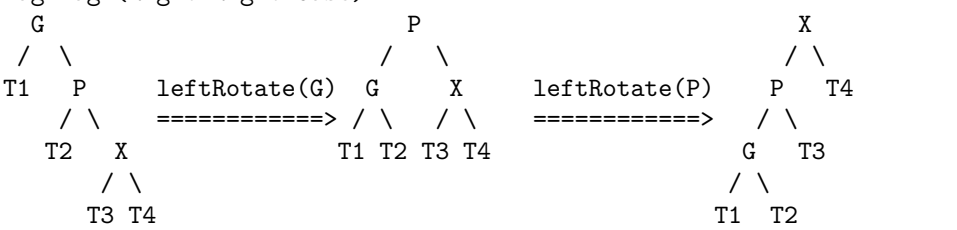
```
             y                                x
            / \     Zig (Right Rotation)     /  \
           x   T3   - - - - - - - - - ->    T1   y
          / \
```

```
3) Node has both parent and grandparent.  There can be following subcases.

........3.a) Zig-Zig and Zag-Zag Node is left child of parent and parent is also left child of grand parent (T
```

```
Zig-Zig (Left Left Case):
       G                        P                          X
      / \                      /   \                      / \
     P  T4   rightRotate(G)   X     G    rightRotate(P)  T1   P
```

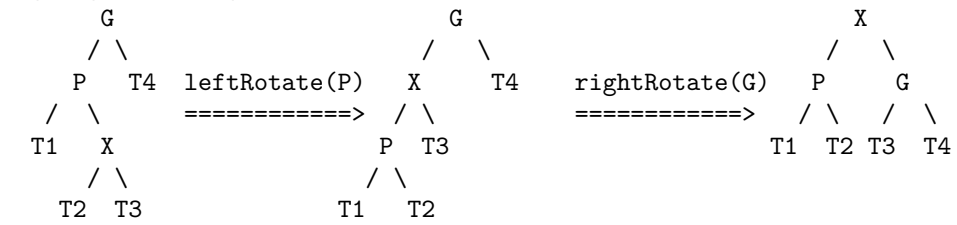


Zag-Zag (Right Right Case):

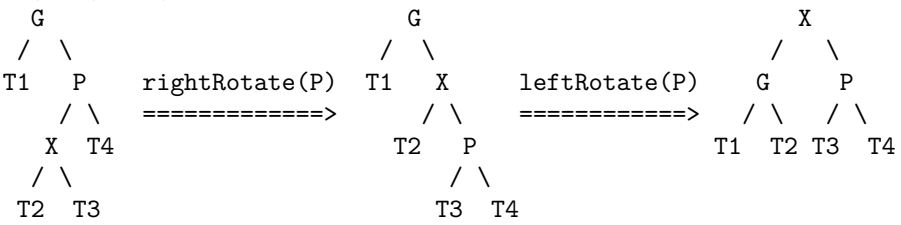


……..**3.b) Zig-Zag and Zag-Zig** Node is left child of parent and parent is right child of grand parent (Left Rotation followed by right rotation) OR node is right child of its parent and parent is left child of grand parent (Right Rotation followed by left rotation).

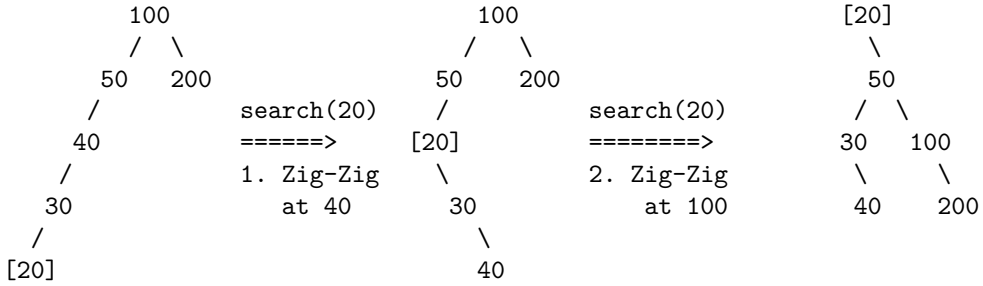
Zig-Zag (Left Right Case):



Zag-Zig (Right Left Case):



Example:



The important thing to note is, the search or splay operation not only brings the searched key to root, but also balances the BST. For example in above case, height of BST is reduced by 1.

Implementation:

```
// The code is adopted from http://goo.gl/SDH9hH
#include<stdio.h>
#include<stdlib.h>

// An AVL tree node
struct node
{
    int key;
    struct node *left, *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  = node->right  = NULL;
    return (node);
}

// A utility function to right rotate subtree rooted with y
// See the diagram given above.
struct node *rightRotate(struct node *x)
{
    struct node *y = x->left;
    x->left = y->right;
    y->right = x;
    return y;
}

// A utility function to left rotate subtree rooted with x
// See the diagram given above.
struct node *leftRotate(struct node *x)
{
    struct node *y = x->right;
    x->right = y->left;
    y->left = x;
    return y;
}

// This function brings the key at root if key is present in tree.
// If key is not present, then it brings the last accessed item at
// root.  This function modifies the tree and returns the new root
struct node *splay(struct node *root, int key)
{
    // Base cases: root is NULL or key is present at root
    if (root == NULL || root->key == key)
        return root;

    // Key lies in left subtree
    if (root->key > key)
    {
```

```
        // Key is not in tree, we are done
        if (root->left == NULL) return root;

        // Zig-Zig (Left Left)
        if (root->left->key > key)
        {
            // First recursively bring the key as root of left-left
            root->left->left = splay(root->left->left, key);

            // Do first rotation for root, second rotation is done after else
            root = rightRotate(root);
        }
        else if (root->left->key < key) // Zig-Zag (Left Right)
        {
            // First recursively bring the key as root of left-right
            root->left->right = splay(root->left->right, key);

            // Do first rotation for root->left
            if (root->left->right != NULL)
                root->left = leftRotate(root->left);
        }

        // Do second rotation for root
        return (root->left == NULL)? root: rightRotate(root);
    }
    else // Key lies in right subtree
    {
        // Key is not in tree, we are done
        if (root->right == NULL) return root;

        // Zag-Zig (Right Left)
        if (root->right->key > key)
        {
            // Bring the key as root of right-left
            root->right->left = splay(root->right->left, key);

            // Do first rotation for root->right
            if (root->right->left != NULL)
                root->right = rightRotate(root->right);
        }
        else if (root->right->key < key)// Zag-Zag (Right Right)
        {
            // Bring the key as root of right-right and do first rotation
            root->right->right = splay(root->right->right, key);
            root = leftRotate(root);
        }

        // Do second rotation for root
        return (root->right == NULL)? root: leftRotate(root);
    }
}

// The search function for Splay tree.  Note that this function
// returns the new root of Splay Tree.  If key is present in tree
```

```
// then, it is moved to root.
struct node *search(struct node *root, int key)
{
    return splay(root, key);
}

// A utility function to print preorder traversal of the tree.
// The function also prints height of every node
void preOrder(struct node *root)
{
    if (root != NULL)
    {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

/* Drier program to test above function*/
int main()
{
    struct node *root = newNode(100);
    root->left = newNode(50);
    root->right = newNode(200);
    root->left->left = newNode(40);
    root->left->left->left = newNode(30);
    root->left->left->left->left = newNode(20);

    root = search(root, 20);
    printf("Preorder traversal of the modified Splay tree is \n");
    preOrder(root);
    return 0;
}
```

Output:

```
 Preorder traversal of the modified Splay tree is
20 50 30 40 100 200
```

Summary

1) Splay trees have excellent locality properties. Frequently accessed items are easy to find. Infrequent items are out of way.

2) All splay tree operations take O(Logn) time on average. Splay trees can be rigorously shown to run in O(log n) average time per operation, over any sequence of operations (assuming we start from an empty tree)

3) Splay trees are simpler compared to AVLand Red-Black Trees as no extra field is required in every tree node.

4) Unlike AVL tree, a splay tree can change even with read-only operations like search.

Applications of Splay Trees

Splay trees have become the most widely used basic data structure invented in the last 30 years, because

they're the fastest type of balanced search tree for many applications.
Splay trees are used in Windows NT (in the virtual memory, networking, and file system code), the gcc compiler and GNU C++ library, the sed string editor, Fore Systems network routers, the most popular implementation of Unix malloc, Linux loadable kernel modules, and in much other software (Source: http://www.cs.berkeley.edu/~jrs/61b/lec/36)

See Splay Tree | Set 2 (Insert) for splay tree insertion.

References:
http://www.cs.berkeley.edu/~jrs/61b/lec/36

http://www.cs.cornell.edu/courses/cs3110/2009fa/recitations/rec-splay.html

http://courses.cs.washington.edu/courses/cse326/01au/lectures/SplayTrees.ppt

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/splay-tree-set-1-insert/

Chapter 30

Splay Tree | Set 2 (Insert)

It is recommended to refer following post as prerequisite of this post.

Splay Tree | Set 1 (Search)

As discussed in the previous post, Splay tree is a self-balancing data structure where the last accessed key is always at root. The insert operation is similar to Binary Search Tree insert with additional steps to make sure that the newly inserted key becomes the new root.

Following are different cases to insert a key k in splay tree.

1) Root is NULL: We simply allocate a new node and return it as root.

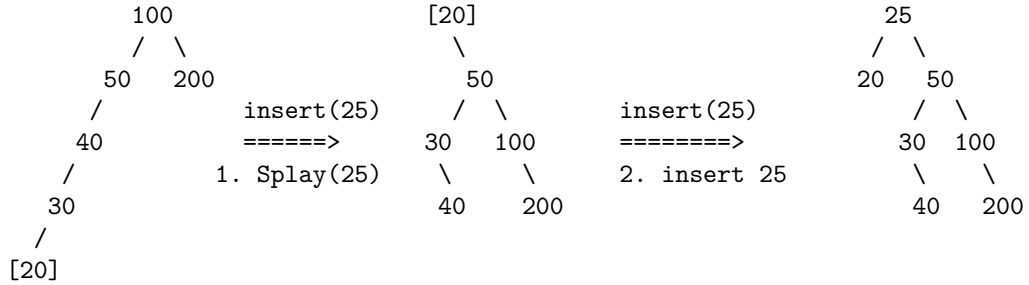
2) Splaythe given key k. If k is already present, then it becomes the new root. If not present, then last accessed leaf node becomes the new root.

3) If new root's key is same as k, don't do anything as k is already present.

4) Else allocate memory for new node and compare root's key with k.
……**4.a)** If k is smaller than root's key, make root as right child of new node, copy left child of root as left child of new node and make left child of root as NULL.
……**4.b)** If k is greater than root's key, make root as left child of new node, copy right child of root as right child of new node and make right child of root as NULL.

5) Return new node as new root of tree.

Example:




```
// This code is adopted from http://algs4.cs.princeton.edu/33balanced/SplayBST.java.html
#include<stdio.h>
#include<stdlib.h>
```

```
// An AVL tree node
struct node
{
    int key;
    struct node *left, *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  = node->right  = NULL;
    return (node);
}

// A utility function to right rotate subtree rooted with y
// See the diagram given above.
struct node *rightRotate(struct node *x)
{
    struct node *y = x->left;
    x->left = y->right;
    y->right = x;
    return y;
}

// A utility function to left rotate subtree rooted with x
// See the diagram given above.
struct node *leftRotate(struct node *x)
{
    struct node *y = x->right;
    x->right = y->left;
    y->left = x;
    return y;
}

// This function brings the key at root if key is present in tree.
// If key is not present, then it brings the last accessed item at
// root.  This function modifies the tree and returns the new root
struct node *splay(struct node *root, int key)
{
    // Base cases: root is NULL or key is present at root
    if (root == NULL || root->key == key)
        return root;

    // Key lies in left subtree
    if (root->key > key)
    {
        // Key is not in tree, we are done
        if (root->left == NULL) return root;

        // Zig-Zig (Left Left)
```

```
        if (root->left->key > key)
        {
            // First recursively bring the key as root of left-left
            root->left->left = splay(root->left->left, key);

            // Do first rotation for root, second rotation is done after else
            root = rightRotate(root);
        }
        else if (root->left->key < key) // Zig-Zag (Left Right)
        {
            // First recursively bring the key as root of left-right
            root->left->right = splay(root->left->right, key);

            // Do first rotation for root->left
            if (root->left->right != NULL)
                root->left = leftRotate(root->left);
        }

        // Do second rotation for root
        return (root->left == NULL)? root: rightRotate(root);
    }
    else // Key lies in right subtree
    {
        // Key is not in tree, we are done
        if (root->right == NULL) return root;

        // Zig-Zag (Right Left)
        if (root->right->key > key)
        {
            // Bring the key as root of right-left
            root->right->left = splay(root->right->left, key);

            // Do first rotation for root->right
            if (root->right->left != NULL)
                root->right = rightRotate(root->right);
        }
        else if (root->right->key < key)// Zag-Zag (Right Right)
        {
            // Bring the key as root of right-right and do first rotation
            root->right->right = splay(root->right->right, key);
            root = leftRotate(root);
        }

        // Do second rotation for root
        return (root->right == NULL)? root: leftRotate(root);
    }
}

// Function to insert a new key k in splay tree with given root
struct node *insert(struct node *root, int k)
{
    // Simple Case: If tree is empty
    if (root == NULL) return newNode(k);
```

```
    // Bring the closest leaf node to root
    root = splay(root, k);

    // If key is already present, then return
    if (root->key == k) return root;

    // Otherwise allocate memory for new node
    struct node *newnode  = newNode(k);

    // If root's key is greater, make root as right child
    // of newnode and copy the left child of root to newnode
    if (root->key > k)
    {
        newnode->right = root;
        newnode->left = root->left;
        root->left = NULL;
    }

    // If root's key is smaller, make root as left child
    // of newnode and copy the right child of root to newnode
    else
    {
        newnode->left = root;
        newnode->right = root->right;
        root->right = NULL;
    }

    return newnode; // newnode becomes new root
}

// A utility function to print preorder traversal of the tree.
// The function also prints height of every node
void preOrder(struct node *root)
{
    if (root != NULL)
    {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

/* Drier program to test above function*/
int main()
{
    struct node *root = newNode(100);
    root->left = newNode(50);
    root->right = newNode(200);
    root->left->left = newNode(40);
    root->left->left->left = newNode(30);
    root->left->left->left->left = newNode(20);
    root = insert(root, 25);
    printf("Preorder traversal of the modified Splay tree is \n");
    preOrder(root);
```

```
    return 0;
}
```

Output:

```
 Preorder traversal of the modified Splay tree is
25 20 50 30 40 100 200
```

This article is compiled by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/splay-tree-set-2-insert-delete/

Category: Trees Tags: Advance Data Structures, Advanced Data Structures

Post navigation

← Splay Tree | Set 1 (Search) Cisco Interview | Set 6 →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

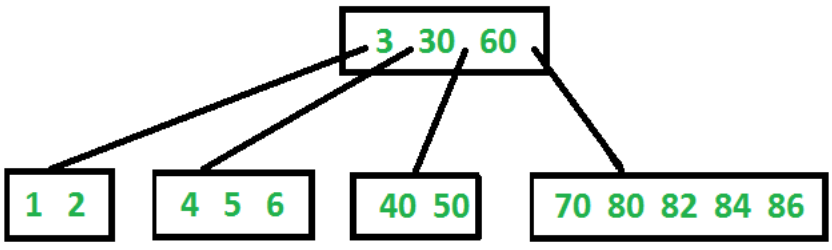
Chapter 31

B-Tree | Set 1 (Introduction)

B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like AVL and Red Black Trees), it is assumed that everything is in main memory. To understand use of B-Trees, we must think of huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min, ..etc) require O(h) disk accesses where h is height of the tree. B-tree is a fat tree. Height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, a B-Tree node size is kept equal to the disk block size. Since h is low for B-Tree, total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red Black Tree, ..etc.

Properties of B-Tree
1) All leaves are at same level.
2) A B-Tree is defined by the term *minimum degree* 't'. The value of t depends upon disk block size.
3) Every node except root must contain at least t-1 keys. Root may contain minimum 1 key.
4) All nodes (including root) may contain at most 2t – 1 keys.
5) Number of children of a node is equal to the number of keys in it plus 1.
6) All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in range from k1 and k2.
7) B-Tree grows and shrinks from root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
8) Like other balanced Binary Search Trees, time complexity to search, insert and delete is O(Logn).

Following is an example B-Tree of minimum degree 3. Note that in practical B-Trees, the value of minimum degree is much more than 3.



Search
Search is similar to search in Binary Search Tree. Let the key to be searched be k. We start from root and

recursively traverse down. For every visited non-leaf node, if the node has key, we simply return the node. Otherwise we recur down to the appropriate child (The child which is just before the first greater key) of the node. If we reach a leaf node and don't find k in the leaf node, we return NULL.

Traverse
Traversal is also similar to Inorder traversal of Binary Tree. We start from the leftmost child, recursively print the leftmost child, then repeat the same process for remaining children and keys. In the end, recursively print the rightmost child.

```
// C++ implemntation of search() and traverse() methods
#include<iostream>
using namespace std;

// A BTree node
class BTreeNode
{
    int *keys;  // An array of keys
    int t;      // Minimum degree (defines the range for number of keys)
    BTreeNode **C; // An array of child pointers
    int n;     // Current number of keys
    bool leaf; // Is true when node is leaf. Otherwise false
public:
    BTreeNode(int _t, bool _leaf);   // Constructor

    // A function to traverse all nodes in a subtree rooted with this node
    void traverse();

    // A function to search a key in subtree rooted with this node.
    BTreeNode *search(int k);   // returns NULL if k is not present.

// Make BTree friend of this so that we can access private members of this
// class in BTree functions
friend class BTree;
};

// A BTree
class BTree
{
    BTreeNode *root; // Pointer to root node
    int t;  // Minimum degree
public:
    // Constructor (Initializes tree as empty)
    BTree(int _t)
    {  root = NULL;  t = _t; }

    // function to traverse the tree
    void traverse()
    {  if (root != NULL) root->traverse(); }

    // function to search a key in this tree
    BTreeNode* search(int k)
    {  return (root == NULL)? NULL : root->search(k); }
};
```

```
// Constructor for BTreeNode class
BTreeNode::BTreeNode(int _t, bool _leaf)
{
    // Copy the given minimum degree and leaf property
    t = _t;
    leaf = _leaf;

    // Allocate memory for maximum number of possible keys
    // and child pointers
    keys = new int[2*t-1];
    C = new BTreeNode *[2*t];

    // Initialize the number of keys as 0
    n = 0;
}

// Function to traverse all nodes in a subtree rooted with this node
void BTreeNode::traverse()
{
    // There are n keys and n+1 children, travers through n keys
    // and first n children
    int i;
    for (i = 0; i < n; i++)
    {
        // If this is not leaf, then before printing key[i],
        // traverse the subtree rooted with child C[i].
        if (leaf == false)
            C[i]->traverse();
        cout << " " << keys[i];
    }

    // Print the subtree rooted with last child
    if (leaf == false)
        C[i]->traverse();
}

// Function to search key k in subtree rooted with this node
BTreeNode *BTreeNode::search(int k)
{
    // Find the first key greater than or equal to k
    int i = 0;
    while (i < n && k > keys[i])
        i++;

    // If the found key is equal to k, return this node
    if (keys[i] == k)
        return this;

    // If key is not found here and this is a leaf node
    if (leaf == true)
        return NULL;

    // Go to the appropriate child
    return C[i]->search(k);
```

```
}
```

The above code doesn't contain driver program. We will be covering the complete program in our next post on B-Tree Insertion.

There are two conventions to define a B-Tree, one is to define by minimum degree (followed in Cormen book), second is define by order. We have followed the minimum degree convention and will be following same in coming posts on B-Tree. The variable names used in the above program are also kept same as Cormen book for better readability.

Insertion and Deletion
B-Tree Insertion
B-Tree Deletion

References:

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

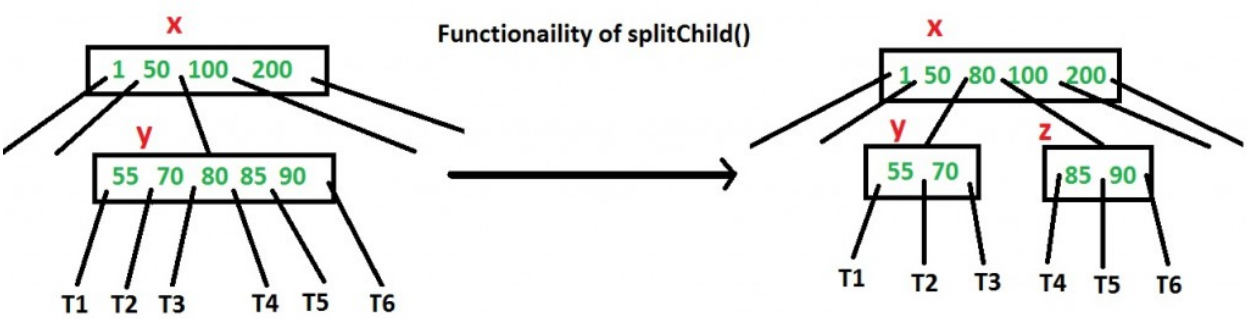
Source

http://www.geeksforgeeks.org/b-tree-set-1-introduction-2/

Chapter 32

B-Tree | Set 2 (Insert)

In the previous post, we introduced B-Tree. We also discussed search() and traverse() functions.
In this post, insert() operation is discussed. A new key is always inserted at leaf node. Let the key to be inserted be k. Like BST, we start from root and traverse down till we reach a leaf node. Once we reach a leaf node, we insert the key in that leaf node. Unlike BSTs, we have a predefined range on number of keys that a node can contain. So before inserting a key to node, we make sure that the node has extra space.
How to make sure that a node has space available for key before the key is inserted? We use an operation called splitChild() that is used to split a child of a node. See the following diagram to understand split. In the following diagram, child y of x is being split into two nodes y and z. Note that the splitChild operation moves a key up and this is the reason B-Trees grow up unlike BSTs which grow down.



As discussed above, to insert a new key, we go down from root to leaf. Before traversing down to a node, we first check if the node is full. If the node is full, we split it to create space. Following is complete algorithm.

Insertion
1) Initialize x as root.
2) While x is not leaf, do following
..**a)** Find the child of x that is going to to be traversed next. Let the child be y.
..**b)** If y is not full, change x to point to y.
..**c)** If y is full, split it and change x to point to one of the two parts of y. If k is smaller than mid key in y, then set x as first part of y. Else second part of y. When we split y, we move a key from y to its parent x.
3) The loop in step 2 stops when x is leaf. x must have space for 1 extra key as we have been splitting all nodes in advance. So simply insert k to x.

Note that the algorithm follows the Cormen book. It is actually a proactive insertion algorithm where before going down to a node, we split it if it is full. The advantage of splitting before is, we never traverse a node twice. If we don't split a node before going down to it and split it only if new key is inserted (reactive), we

may end up traversing all nodes again from leaf to root. This happens in cases when all nodes on the path from root to leaf are full. So when we come to the leaf node, we split it and move a key up. Moving a key up will cause a split in parent node (because parent was already full). This cascading effect never happens in this proactive insertion algorithm. There is a disadvantage of this proactive insertion though, we may do unnecessary splits.

Let us understand the algorithm with an example tree of minimum degree 't' as 3 and a sequence of integers 10, 20, 30, 40, 50, 60, 70, 80 and 90 in an initially empty B-Tree.

Initially root is NULL. Let us first insert 10.

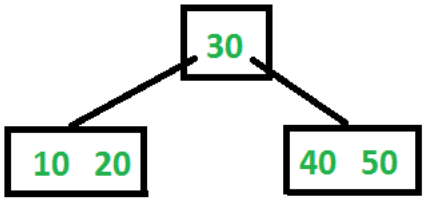
Insert 10

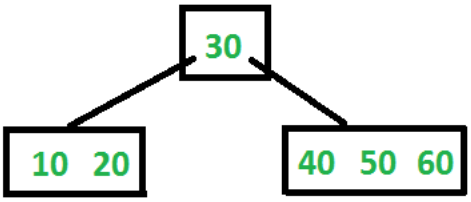
Let us now insert 20, 30, 40 and 50. They all will be inserted in root because maximum number of keys a node can accommodate is 2*t – 1 which is 5.

Insert 20, 30, 40 and 50

Let us now insert 60. Since root node is full, it will first split into two, then 60 will be inserted into the appropriate child.

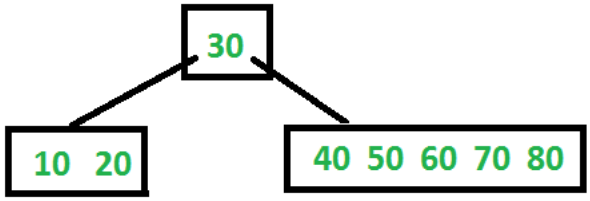
Insert 60



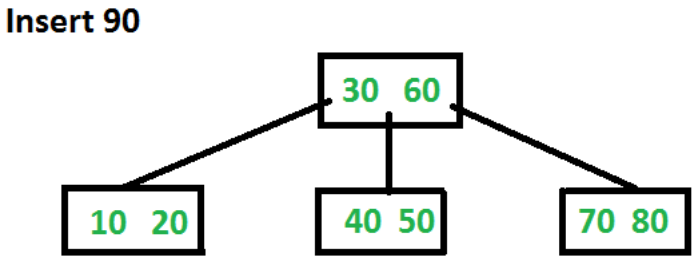


Let us now insert 70 and 80. These new keys will be inserted into the appropriate leaf without any split.

Insert 70 and 80



Let us now insert 90. This insertion will cause a split. The middle key will go up to the parent.



See thisfor more examples.

Following is C++ implementation of the above proactive algorithm.

```
// C++ program for B-Tree insertion
#include<iostream>
using namespace std;

// A BTree node
class BTreeNode
{
    int *keys;  // An array of keys
    int t;      // Minimum degree (defines the range for number of keys)
    BTreeNode **C; // An array of child pointers
    int n;     // Current number of keys
    bool leaf; // Is true when node is leaf. Otherwise false
public:
    BTreeNode(int _t, bool _leaf);   // Constructor

    // A utility function to insert a new key in the subtree rooted with
    // this node. The assumption is, the node must be non-full when this
    // function is called
    void insertNonFull(int k);

    // A utility function to split the child y of this node. i is index of y in
    // child array C[].  The Child y must be full when this function is called
    void splitChild(int i, BTreeNode *y);

    // A function to traverse all nodes in a subtree rooted with this node
    void traverse();

    // A function to search a key in subtree rooted with this node.
    BTreeNode *search(int k);   // returns NULL if k is not present.

// Make BTree friend of this so that we can access private members of this
// class in BTree functions
friend class BTree;
};

// A BTree
class BTree
```

```
{
    BTreeNode *root; // Pointer to root node
    int t;  // Minimum degree
public:
    // Constructor (Initializes tree as empty)
    BTree(int _t)
    {  root = NULL;  t = _t; }

    // function to traverse the tree
    void traverse()
    {  if (root != NULL) root->traverse(); }

    // function to search a key in this tree
    BTreeNode* search(int k)
    {  return (root == NULL)? NULL : root->search(k); }

    // The main function that inserts a new key in this B-Tree
    void insert(int k);
};

// Constructor for BTreeNode class
BTreeNode::BTreeNode(int t1, bool leaf1)
{
    // Copy the given minimum degree and leaf property
    t = t1;
    leaf = leaf1;

    // Allocate memory for maximum number of possible keys
    // and child pointers
    keys = new int[2*t-1];
    C = new BTreeNode *[2*t];

    // Initialize the number of keys as 0
    n = 0;
}

// Function to traverse all nodes in a subtree rooted with this node
void BTreeNode::traverse()
{
    // There are n keys and n+1 children, travers through n keys
    // and first n children
    int i;
    for (i = 0; i < n; i++)
    {
        // If this is not leaf, then before printing key[i],
        // traverse the subtree rooted with child C[i].
        if (leaf == false)
            C[i]->traverse();
        cout << " " << keys[i];
    }

    // Print the subtree rooted with last child
    if (leaf == false)
        C[i]->traverse();
```

```
}

// Function to search key k in subtree rooted with this node
BTreeNode *BTreeNode::search(int k)
{
    // Find the first key greater than or equal to k
    int i = 0;
    while (i < n && k > keys[i])
        i++;

    // If the found key is equal to k, return this node
    if (keys[i] == k)
        return this;

    // If key is not found here and this is a leaf node
    if (leaf == true)
        return NULL;

    // Go to the appropriate child
    return C[i]->search(k);
}

// The main function that inserts a new key in this B-Tree
void BTree::insert(int k)
{
    // If tree is empty
    if (root == NULL)
    {
        // Allocate memory for root
        root = new BTreeNode(t, true);
        root->keys[0] = k;  // Insert key
        root->n = 1;  // Update number of keys in root
    }
    else // If tree is not empty
    {
        // If root is full, then tree grows in height
        if (root->n == 2*t-1)
        {
            // Allocate memory for new root
            BTreeNode *s = new BTreeNode(t, false);

            // Make old root as child of new root
            s->C[0] = root;

            // Split the old root and move 1 key to the new root
            s->splitChild(0, root);

            // New root has two children now.  Decide which of the
            // two children is going to have new key
            int i = 0;
            if (s->keys[0] < k)
                i++;
            s->C[i]->insertNonFull(k);
```

```
            // Change root
            root = s;
        }
        else  // If root is not full, call insertNonFull for root
            root->insertNonFull(k);
    }
}

// A utility function to insert a new key in this node
// The assumption is, the node must be non-full when this
// function is called
void BTreeNode::insertNonFull(int k)
{
    // Initialize index as index of rightmost element
    int i = n-1;

    // If this is a leaf node
    if (leaf == true)
    {
        // The following loop does two things
        // a) Finds the location of new key to be inserted
        // b) Moves all greater keys to one place ahead
        while (i >= 0 && keys[i] > k)
        {
            keys[i+1] = keys[i];
            i--;
        }

        // Insert the new key at found location
        keys[i+1] = k;
        n = n+1;
    }
    else // If this node is not leaf
    {
        // Find the child which is going to have the new key
        while (i >= 0 && keys[i] > k)
            i--;

        // See if the found child is full
        if (C[i+1]->n == 2*t-1)
        {
            // If the child is full, then split it
            splitChild(i+1, C[i+1]);

            // After split, the middle key of C[i] goes up and
            // C[i] is splitted into two.  See which of the two
            // is going to have the new key
            if (keys[i+1] < k)
                i++;
        }
        C[i+1]->insertNonFull(k);
    }
}
```

```
// A utility function to split the child y of this node
// Note that y must be full when this function is called
void BTreeNode::splitChild(int i, BTreeNode *y)
{
    // Create a new node which is going to store (t-1) keys
    // of y
    BTreeNode *z = new BTreeNode(y->t, y->leaf);
    z->n = t - 1;

    // Copy the last (t-1) keys of y to z
    for (int j = 0; j < t-1; j++)
        z->keys[j] = y->keys[j+t];

    // Copy the last t children of y to z
    if (y->leaf == false)
    {
        for (int j = 0; j < t; j++)
            z->C[j] = y->C[j+t];
    }

    // Reduce the number of keys in y
    y->n = t - 1;

    // Since this node is going to have a new child,
    // create space of new child
    for (int j = n; j >= i+1; j--)
        C[j+1] = C[j];

    // Link the new child to this node
    C[i+1] = z;

    // A key of y will move to this node. Find location of
    // new key and move all greater keys one space ahead
    for (int j = n-1; j >= i; j--)
        keys[j+1] = keys[j];

    // Copy the middle key of y to this node
    keys[i] = y->keys[t-1];

    // Increment count of keys in this node
    n = n + 1;
}

// Driver program to test above functions
int main()
{
    BTree t(3); // A B-Tree with minium degree 3
    t.insert(10);
    t.insert(20);
    t.insert(5);
    t.insert(6);
    t.insert(12);
    t.insert(30);
    t.insert(7);
```

```
    t.insert(17);

    cout << "Traversal of the constucted tree is ";
    t.traverse();

    int k = 6;
    (t.search(k) != NULL)? cout << "\nPresent" : cout << "\nNot Present";

    k = 15;
    (t.search(k) != NULL)? cout << "\nPresent" : cout << "\nNot Present";

    return 0;
}
```

Output:

```
 Traversal of the constucted tree is  5 6 7 10 12 17 20 30
Present
Not Present
```

References:

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

http://www.cs.utexas.edu/users/djimenez/utsa/cs3343/lecture17.html

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/b-tree-set-1-insert-2/

Chapter 33

B-Tree | Set 3 (Delete)

It is recommended to refer following posts as prerequisite of this post.

B-Tree | Set 1 (Introduction)
B-Tree | Set 2 (Insert)

B-Tree is a type of a multi-way search tree. So, if you are not familiar with multi-way search trees in general, it is better to take a look at this video lecture from IIT-Delhi, before proceeding further. Once you get the basics of a multi-way search tree clear, B-Tree operations will be easier to understand.

Source of the following explanation and algorithm is Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

Deletion process:
Deletion from a B-tree is more complicated than insertion, because we can delete a key from any node-not just a leaf—and when we delete a key from an internal node, we will have to rearrange the node's children.

As in insertion, we must make sure the deletion doesn't violate the B-tree properties. Just as we had to ensure that a node didn't get too big due to insertion, we must ensure that a node doesn't get too small during deletion (except that the root is allowed to have fewer than the minimum number t-1 of keys). Just as a simple insertion algorithm might have to back up if a node on the path to where the key was to be inserted was full, a simple approach to deletion might have to back up if a node (other than the root) along the path to where the key is to be deleted has the minimum number of keys.

The deletion procedure deletes the key k from the subtree rooted at x. This procedure guarantees that whenever it calls itself recursively on a node x, the number of keys in x is at least the minimum degree t . Note that this condition requires one more key than the minimum required by the usual B-tree conditions, so that sometimes a key may have to be moved into a child node before recursion descends to that child. This strengthened condition allows us to delete a key from the tree in one downward pass without having to "back up" (with one exception, which we'll explain). You should interpret the following specification for deletion from a B-tree with the understanding that if the root node x ever becomes an internal node having no keys (this situation can occur in cases 2c and 3b then we delete x, and x's only child x.c1 becomes the new root of the tree, decreasing the height of the tree by one and preserving the property that the root of the tree contains at least one key (unless the tree is empty).

We sketch how deletion works with various cases of deleting keys from a B-tree.

1. If the key k is in node x and x is a leaf, delete the key k from x.

2. If the key k is in node x and x is an internal node, do the following.

a) If the child y that precedes k in node x has at least t keys, then find the predecessor k0 of k in the sub-tree rooted at y. Recursively delete k0, and replace k by k0 in x. (We can find k0 and delete it in a single downward pass.)

b) If y has fewer than t keys, then, symmetrically, examine the child z that follows k in node x. If z has at least t keys, then find the successor k0 of k in the subtree rooted at z. Recursively delete k0, and replace k by k0 in x. (We can find k0 and delete it in a single downward pass.)

c) Otherwise, if both y and z have only t-1 keys, merge k and all of z into y, so that x loses both k and the pointer to z, and y now contains 2t-1 keys. Then free z and recursively delete k from y.

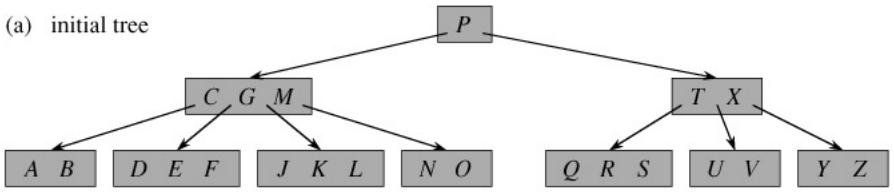
3. If the key k is not present in internal node x, determine the root x.c(i) of the appropriate subtree that must contain k, if k is in the tree at all. If x.c(i) has only t-1 keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then finish by recursing on the appropriate child of x.

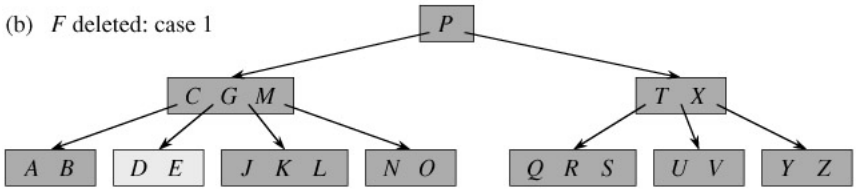
a) If x.c(i) has only t-1 keys but has an immediate sibling with at least t keys, give x.c(i) an extra key by moving a key from x down into x.c(i), moving a key from x.c(i) 's immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into x.c(i).

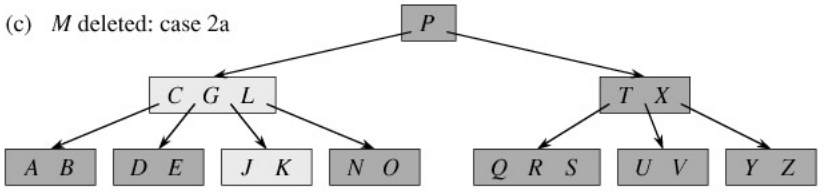
b) If x.c(i) and both of x.c(i)'s immediate siblings have t-1 keys, merge x.c(i) with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

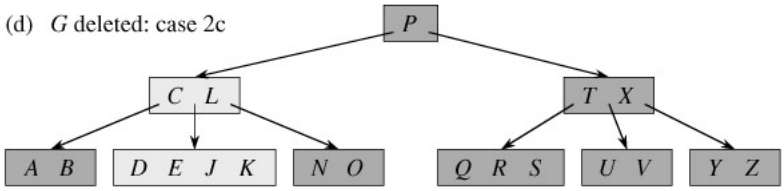
Since most of the keys in a B-tree are in the leaves, deletion operations are most often used to delete keys from leaves. The recursive delete procedure then acts in one downward pass through the tree, without having to back up. When deleting a key in an internal node, however, the procedure makes a downward pass through the tree but may have to return to the node from which the key was deleted to replace the key with its predecessor or successor (cases 2a and 2b).

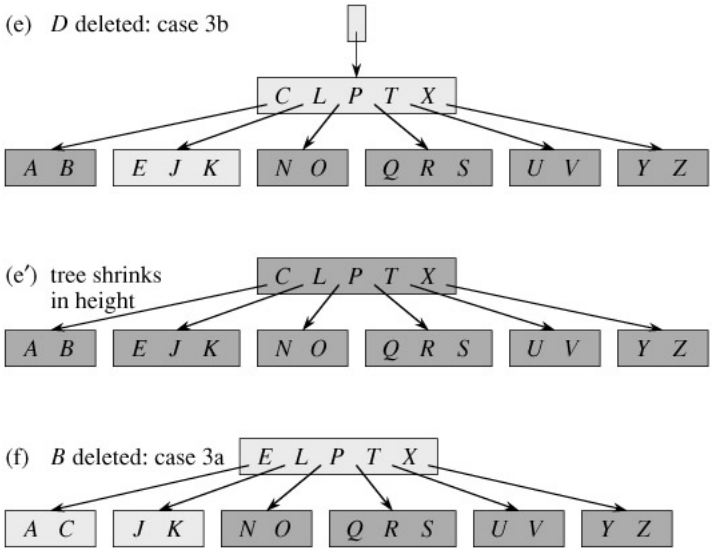
The following figures from CLRS book explain the deletion porcess.











Implementation:
Following is C++ implementation of deletion process.

```
/* The following program performs deletion on a B-Tree. It contains functions
   specific for deletion along with all the other functions provided in the
  previous articles on B-Trees. See http://www.geeksforgeeks.org/b-tree-set-1-introduction-2/
   for previous article.

   The deletion function has been compartmentalized into 8 functions for ease
   of understanding and clarity

   The following functions are exclusive for deletion
   In class BTreeNode:
    1) remove
    2) removeFromLeaf
    3) removeFromNonLeaf
    4) getPred
    5) getSucc
    6) borrowFromPrev
    7) borrowFromNext
    8) merge
    9) findKey

   In class BTree:
     1) remove

  The removal of a key from a B-Tree is a fairly complicated process. The program handles
  all the 6 different cases that might arise while removing a key.

  Testing: The code has been tested using the B-Tree provided in the CLRS book( included
  in the main function ) along with other cases.

  Reference: CLRS3 - Chapter 18 - (499-502)
  It is advised to read the material in CLRS before taking a look at the code. */
```

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

// A BTree node
class BTreeNode
{
    int *keys;  // An array of keys
    int t;      // Minimum degree (defines the range for number of keys)
    BTreeNode **C; // An array of child pointers
    int n;     // Current number of keys
    bool leaf; // Is true when node is leaf. Otherwise false

public:

    BTreeNode(int _t, bool _leaf);   // Constructor

    // A function to traverse all nodes in a subtree rooted with this node
    void traverse();

    // A function to search a key in subtree rooted with this node.
    BTreeNode *search(int k);   // returns NULL if k is not present.

    // A function that returns the index of the first key that is greater
    // or equal to k
    int findKey(int k);

    // A utility function to insert a new key in the subtree rooted with
    // this node. The assumption is, the node must be non-full when this
    // function is called
    void insertNonFull(int k);

    // A utility function to split the child y of this node. i is index
    // of y in child array C[].  The Child y must be full when this
    // function is called
    void splitChild(int i, BTreeNode *y);

    // A wrapper function to remove the key k in subtree rooted with
    // this node.
    void remove(int k);

    // A function to remove the key present in idx-th position in
    // this node which is a leaf
    void removeFromLeaf(int idx);

    // A function to remove the key present in idx-th position in
    // this node which is a non-leaf node
    void removeFromNonLeaf(int idx);

    // A function to get the predecessor of the key- where the key
    // is present in the idx-th position in the node
    int getPred(int idx);

    // A function to get the successor of the key- where the key
```

```
    // is present in the idx-th position in the node
    int getSucc(int idx);

    // A function to fill up the child node present in the idx-th
    // position in the C[] array if that child has less than t-1 keys
    void fill(int idx);

    // A function to borrow a key from the C[idx-1]-th node and place
    // it in C[idx]th node
    void borrowFromPrev(int idx);

    // A function to borrow a key from the C[idx+1]-th node and place it
    // in C[idx]th node
    void borrowFromNext(int idx);

    // A function to merge idx-th child of the node with (idx+1)th child of
    // the node
    void merge(int idx);

    // Make BTree friend of this so that we can access private members of
    // this class in BTree functions
    friend class BTree;
};

class BTree
{
    BTreeNode *root; // Pointer to root node
    int t;  // Minimum degree
public:

    // Constructor (Initializes tree as empty)
    BTree(int _t)
    {
        root = NULL;
        t = _t;
    }

    void traverse()
    {
        if (root != NULL) root->traverse();
    }

    // function to search a key in this tree
    BTreeNode* search(int k)
    {
        return (root == NULL)? NULL : root->search(k);
    }

    // The main function that inserts a new key in this B-Tree
    void insert(int k);

    // The main function that removes a new key in thie B-Tree
    void remove(int k);
```

```
};

BTreeNode::BTreeNode(int t1, bool leaf1)
{
    // Copy the given minimum degree and leaf property
    t = t1;
    leaf = leaf1;

    // Allocate memory for maximum number of possible keys
    // and child pointers
    keys = new int[2*t-1];
    C = new BTreeNode *[2*t];

    // Initialize the number of keys as 0
    n = 0;
}

// A utility function that returns the index of the first key that is
// greater than or equal to k
int BTreeNode::findKey(int k)
{
    int idx=0;
    while (idx<n && keys[idx] < k)
        ++idx;
    return idx;
}

// A function to remove the key k from the sub-tree rooted with this node
void BTreeNode::remove(int k)
{
    int idx = findKey(k);

    // The key to be removed is present in this node
    if (idx < n && keys[idx] == k)
    {

        // If the node is a leaf node - removeFromLeaf is called
        // Otherwise, removeFromNonLeaf function is called
        if (leaf)
            removeFromLeaf(idx);
        else
            removeFromNonLeaf(idx);
    }
    else
    {

        // If this node is a leaf node, then the key is not present in tree
        if (leaf)
        {
            cout << "The key "<< k <<" is does not exist in the tree\n";
            return;
        }

        // The key to be removed is present in the sub-tree rooted with this node
```

```
        // The flag indicates whether the key is present in the sub-tree rooted
        // with the last child of this node
        bool flag = ( (idx==n)? true : false );

        // If the child where the key is supposed to exist has less that t keys,
        // we fill that child
        if (C[idx]->n < t)
            fill(idx);

        // If the last child has been merged, it must have merged with the previous
        // child and so we recurse on the (idx-1)th child. Else, we recurse on the
        // (idx)th child which now has atleast t keys
        if (flag && idx > n)
            C[idx-1]->remove(k);
        else
            C[idx]->remove(k);
    }
    return;
}

// A function to remove the idx-th key from this node - which is a leaf node
void BTreeNode::removeFromLeaf (int idx)
{

    // Move all the keys after the idx-th pos one place backward
    for (int i=idx+1; i<n; ++i)
        keys[i-1] = keys[i];

    // Reduce the count of keys
    n--;

    return;
}

// A function to remove the idx-th key from this node - which is a non-leaf node
void BTreeNode::removeFromNonLeaf(int idx)
{

    int k = keys[idx];

    // If the child that precedes k (C[idx]) has atleast t keys,
    // find the predecessor 'pred' of k in the subtree rooted at
    // C[idx]. Replace k by pred. Recursively delete pred
    // in C[idx]
    if (C[idx]->n >= t)
    {
        int pred = getPred(idx);
        keys[idx] = pred;
        C[idx]->remove(pred);
    }

    // If the child C[idx] has less that t keys, examine C[idx+1].
    // If C[idx+1] has atleast t keys, find the successor 'succ' of k in
    // the subtree rooted at C[idx+1]
```

```
    // Replace k by succ
    // Recursively delete succ in C[idx+1]
    else if  (C[idx+1]->n >= t)
    {
        int succ = getSucc(idx);
        keys[idx] = succ;
        C[idx+1]->remove(succ);
    }

    // If both C[idx] and C[idx+1] has less that t keys,merge k and all of C[idx+1]
    // into C[idx]
    // Now C[idx] contains 2t-1 keys
    // Free C[idx+1] and recursively delete k from C[idx]
    else
    {
        merge(idx);
        C[idx]->remove(k);
    }
    return;
}

// A function to get predecessor of keys[idx]
int BTreeNode::getPred(int idx)
{
    // Keep moving to the right most node until we reach a leaf
    BTreeNode *cur=C[idx];
    while (!cur->leaf)
        cur = cur->C[cur->n];

    // Return the last key of the leaf
    return cur->keys[cur->n-1];
}

int BTreeNode::getSucc(int idx)
{

    // Keep moving the left most node starting from C[idx+1] until we reach a leaf
    BTreeNode *cur = C[idx+1];
    while (!cur->leaf)
        cur = cur->C[0];

    // Return the first key of the leaf
    return cur->keys[0];
}

// A function to fill child C[idx] which has less than t-1 keys
void BTreeNode::fill(int idx)
{

    // If the previous child(C[idx-1]) has more than t-1 keys, borrow a key
    // from that child
    if (idx!=0 && C[idx-1]->n>=t)
        borrowFromPrev(idx);
```

```
    // If the next child(C[idx+1]) has more than t-1 keys, borrow a key
    // from that child
    else if (idx!=n && C[idx+1]->n>=t)
        borrowFromNext(idx);

    // Merge C[idx] with its sibling
    // If C[idx] is the last child, merge it with with its previous sibling
    // Otherwise merge it with its next sibling
    else
    {
        if (idx != n)
            merge(idx);
        else
            merge(idx-1);
    }
    return;
}

// A function to borrow a key from C[idx-1] and insert it
// into C[idx]
void BTreeNode::borrowFromPrev(int idx)
{

    BTreeNode *child=C[idx];
    BTreeNode *sibling=C[idx-1];

    // The last key from C[idx-1] goes up to the parent and key[idx-1]
    // from parent is inserted as the first key in C[idx]. Thus, the  loses
    // sibling one key and child gains one key

    // Moving all key in C[idx] one step ahead
    for (int i=child->n-1; i>=0; --i)
        child->keys[i+1] = child->keys[i];

    // If C[idx] is not a leaf, move all its child pointers one step ahead
    if (!child->leaf)
    {
        for(int i=child->n; i>=0; --i)
            child->C[i+1] = child->C[i];
    }

    // Setting child's first key equal to keys[idx-1] from the current node
    child->keys[0] = keys[idx-1];

    // Moving sibling's last child as C[idx]'s first child
    if (!leaf)
        child->C[0] = sibling->C[sibling->n];

    // Moving the key from the sibling to the parent
    // This reduces the number of keys in the sibling
    keys[idx-1] = sibling->keys[sibling->n-1];

    child->n += 1;
    sibling->n -= 1;
```

```
    return;
}

// A function to borrow a key from the C[idx+1] and place
// it in C[idx]
void BTreeNode::borrowFromNext(int idx)
{

    BTreeNode *child=C[idx];
    BTreeNode *sibling=C[idx+1];

    // keys[idx] is inserted as the last key in C[idx]
    child->keys[(child->n)] = keys[idx];

    // Sibling's first child is inserted as the last child
    // into C[idx]
    if (!(child->leaf))
        child->C[(child->n)+1] = sibling->C[0];

    //The first key from sibling is inserted into keys[idx]
    keys[idx] = sibling->keys[0];

    // Moving all keys in sibling one step behind
    for (int i=1; i<sibling->n; ++i)
        sibling->keys[i-1] = sibling->keys[i];

    // Moving the child pointers one step behind
    if (!sibling->leaf)
    {
        for(int i=1; i<=sibling->n; ++i)
            sibling->C[i-1] = sibling->C[i];
    }

    // Increasing and decreasing the key count of C[idx] and C[idx+1]
    // respectively
    child->n += 1;
    sibling->n -= 1;

    return;
}

// A function to merge C[idx] with C[idx+1]
// C[idx+1] is freed after merging
void BTreeNode::merge(int idx)
{
    BTreeNode *child = C[idx];
    BTreeNode *sibling = C[idx+1];

    // Pulling a key from the current node and inserting it into (t-1)th
    // position of C[idx]
    child->keys[t-1] = keys[idx];

    // Copying the keys from C[idx+1] to C[idx] at the end
```

```
    for (int i=0; i<sibling->n; ++i)
        child->keys[i+t] = sibling->keys[i];

    // Copying the child pointers from C[idx+1] to C[idx]
    if (!child->leaf)
    {
        for(int i=0; i<=sibling->n; ++i)
            child->C[i+t] = sibling->C[i];
    }

    // Moving all keys after idx in the current node one step before -
    // to fill the gap created by moving keys[idx] to C[idx]
    for (int i=idx+1; i<n; ++i)
        keys[i-1] = keys[i];

    // Moving the child pointers after (idx+1) in the current node one
    // step before
    for (int i=idx+2; i<=n; ++i)
        C[i-1] = C[i];

    // Updating the key count of child and the current node
    child->n += sibling->n+1;
    n--;

    // Freeing the memory occupied by sibling
    delete(sibling);
    return;
}

// The main function that inserts a new key in this B-Tree
void BTree::insert(int k)
{
    // If tree is empty
    if (root == NULL)
    {
        // Allocate memory for root
        root = new BTreeNode(t, true);
        root->keys[0] = k;  // Insert key
        root->n = 1;  // Update number of keys in root
    }
    else // If tree is not empty
    {
        // If root is full, then tree grows in height
        if (root->n == 2*t-1)
        {
            // Allocate memory for new root
            BTreeNode *s = new BTreeNode(t, false);

            // Make old root as child of new root
            s->C[0] = root;

            // Split the old root and move 1 key to the new root
            s->splitChild(0, root);
```

```
            // New root has two children now.  Decide which of the
            // two children is going to have new key
            int i = 0;
            if (s->keys[0] < k)
                i++;
            s->C[i]->insertNonFull(k);

            // Change root
            root = s;
        }
        else  // If root is not full, call insertNonFull for root
            root->insertNonFull(k);
    }
}

// A utility function to insert a new key in this node
// The assumption is, the node must be non-full when this
// function is called
void BTreeNode::insertNonFull(int k)
{
    // Initialize index as index of rightmost element
    int i = n-1;

    // If this is a leaf node
    if (leaf == true)
    {
        // The following loop does two things
        // a) Finds the location of new key to be inserted
        // b) Moves all greater keys to one place ahead
        while (i >= 0 && keys[i] > k)
        {
            keys[i+1] = keys[i];
            i--;
        }

        // Insert the new key at found location
        keys[i+1] = k;
        n = n+1;
    }
    else // If this node is not leaf
    {
        // Find the child which is going to have the new key
        while (i >= 0 && keys[i] > k)
            i--;

        // See if the found child is full
        if (C[i+1]->n == 2*t-1)
        {
            // If the child is full, then split it
            splitChild(i+1, C[i+1]);

            // After split, the middle key of C[i] goes up and
            // C[i] is splitted into two.  See which of the two
            // is going to have the new key
```

```
            if (keys[i+1] < k)
                i++;
        }
        C[i+1]->insertNonFull(k);
    }
}

// A utility function to split the child y of this node
// Note that y must be full when this function is called
void BTreeNode::splitChild(int i, BTreeNode *y)
{
    // Create a new node which is going to store (t-1) keys
    // of y
    BTreeNode *z = new BTreeNode(y->t, y->leaf);
    z->n = t - 1;

    // Copy the last (t-1) keys of y to z
    for (int j = 0; j < t-1; j++)
        z->keys[j] = y->keys[j+t];

    // Copy the last t children of y to z
    if (y->leaf == false)
    {
        for (int j = 0; j < t; j++)
            z->C[j] = y->C[j+t];
    }

    // Reduce the number of keys in y
    y->n = t - 1;

    // Since this node is going to have a new child,
    // create space of new child
    for (int j = n; j >= i+1; j--)
        C[j+1] = C[j];

    // Link the new child to this node
    C[i+1] = z;

    // A key of y will move to this node. Find location of
    // new key and move all greater keys one space ahead
    for (int j = n-1; j >= i; j--)
        keys[j+1] = keys[j];

    // Copy the middle key of y to this node
    keys[i] = y->keys[t-1];

    // Increment count of keys in this node
    n = n + 1;
}

// Function to traverse all nodes in a subtree rooted with this node
void BTreeNode::traverse()
{
    // There are n keys and n+1 children, travers through n keys
```

```
    // and first n children
    int i;
    for (i = 0; i < n; i++)
    {
        // If this is not leaf, then before printing key[i],
        // traverse the subtree rooted with child C[i].
        if (leaf == false)
            C[i]->traverse();
        cout << " " << keys[i];
    }

    // Print the subtree rooted with last child
    if (leaf == false)
        C[i]->traverse();
}

// Function to search key k in subtree rooted with this node
BTreeNode *BTreeNode::search(int k)
{
    // Find the first key greater than or equal to k
    int i = 0;
    while (i < n && k > keys[i])
        i++;

    // If the found key is equal to k, return this node
    if (keys[i] == k)
        return this;

    // If key is not found here and this is a leaf node
    if (leaf == true)
        return NULL;

    // Go to the appropriate child
    return C[i]->search(k);
}

void BTree::remove(int k)
{
    if (!root)
    {
        cout << "The tree is empty\n";
        return;
    }

    // Call the remove function for root
    root->remove(k);

    // If the root node has 0 keys, make its first child as the new root
    //  if it has a child, otherwise set root as NULL
    if (root->n==0)
    {
        BTreeNode *tmp = root;
        if (root->leaf)
            root = NULL;
```

```
        else
            root = root->C[0];

        // Free the old root
        delete tmp;
    }
    return;
}

// Driver program to test above functions
int main()
{
    BTree t(3); // A B-Tree with minium degree 3

    t.insert(1);
    t.insert(3);
    t.insert(7);
    t.insert(10);
    t.insert(11);
    t.insert(13);
    t.insert(14);
    t.insert(15);
    t.insert(18);
    t.insert(16);
    t.insert(19);
    t.insert(24);
    t.insert(25);
    t.insert(26);
    t.insert(21);
    t.insert(4);
    t.insert(5);
    t.insert(20);
    t.insert(22);
    t.insert(2);
    t.insert(17);
    t.insert(12);
    t.insert(6);

    cout << "Traversal of tree constructed is\n";
    t.traverse();
    cout << endl;

    t.remove(6);
    cout << "Traversal of tree after removing 6\n";
    t.traverse();
    cout << endl;

    t.remove(13);
    cout << "Traversal of tree after removing 13\n";
    t.traverse();
    cout << endl;

    t.remove(7);
    cout << "Traversal of tree after removing 7\n";
```

```
    t.traverse();
    cout << endl;

    t.remove(4);
    cout << "Traversal of tree after removing 4\n";
    t.traverse();
    cout << endl;

    t.remove(2);
    cout << "Traversal of tree after removing 2\n";
    t.traverse();
    cout << endl;

    t.remove(16);
    cout << "Traversal of tree after removing 16\n";
    t.traverse();
    cout << endl;

    return 0;
}
```

Output:

```
 Traversal of tree constructed is
 1 2 3 4 5 6 7 10 11 12 13 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 6
 1 2 3 4 5 7 10 11 12 13 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 13
 1 2 3 4 5 7 10 11 12 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 7
 1 2 3 4 5 10 11 12 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 4
 1 2 3 5 10 11 12 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 2
 1 3 5 10 11 12 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 16
 1 3 5 10 11 12 14 15 17 18 19 20 21 22 24 25 26
```

This article is contributed by **Balasubramanian.N**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/b-tree-set-3delete/

Category: Trees Tags: Advance Data Structures, Advanced Data Structures

Post navigation

← Radix Sort Dynamic Programming | Set 36 (Maximum Product Cutting) →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 34

Segment Tree | Set 1 (Sum of given range)

Let us consider the following problem to understand Segment Trees.

We have an array arr[0 . . . n-1]. We should be able to
1 Find the sum of elements from index l to r where 0 2 Change value of a specified element of the array arr[i] = x where 0

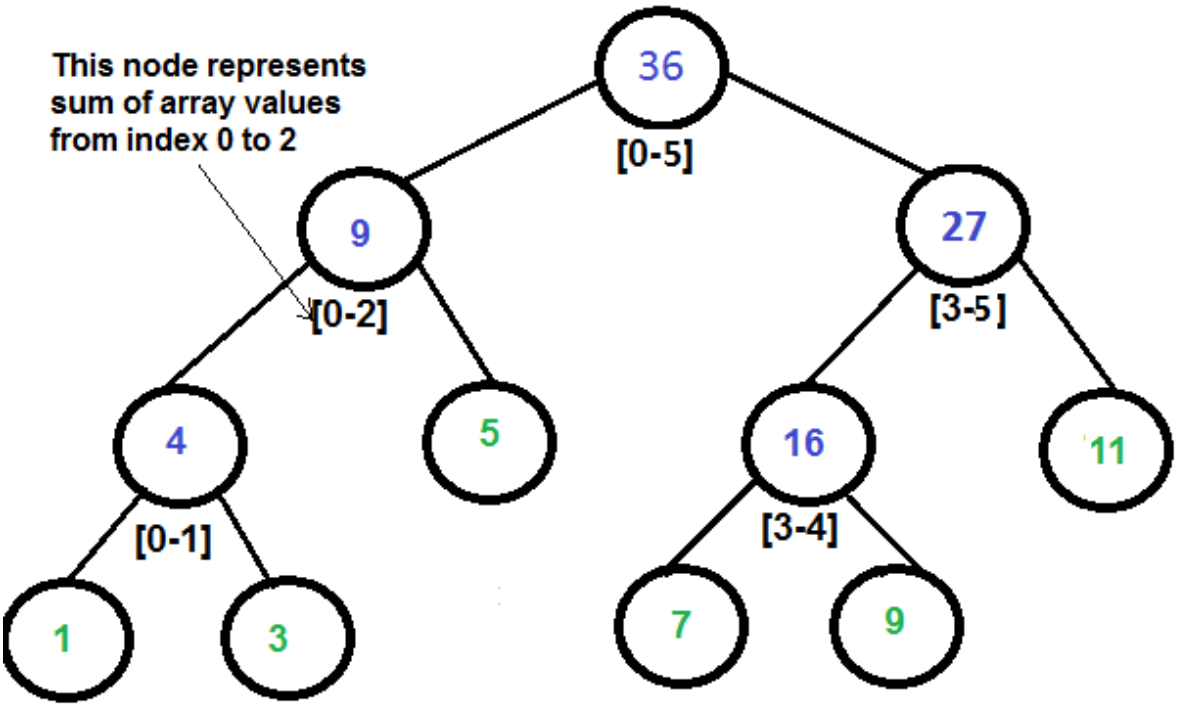
A **simple solution** is to run a loop from l to r and calculate sum of elements in given range. To update a value, simply do arr[i] = x. The first operation takes O(n) time and second operation takes O(1) time.

Another solution is to create another array and store sum from start to i at the ith index in this array. Sum of a given range can now be calculated in O(1) time, but update operation takes O(n) time now. This works well if the number of query operations are large and very few updates.

What if the number of query and updates are equal? **Can we perform both the operations in O(log n) time once given the array?** We can use a Segment Tree to do both operations in O(Logn) time.

Representation of Segment trees
1. Leaf Nodes are the elements of the input array.
2. Each internal node represents some merging of the leaf nodes. The merging may be different for different problems. For this problem, merging is sum of leaves under a node.

An array representation of tree is used to represent Segment Trees. For each node at index i, the left child is at index 2*i+1, right child at 2*i+2 and the parent is at $\lfloor (i-1)/2 \rfloor$.



Segment Tree for input array {1, 3, 5, 7, 9, 11}

Construction of Segment Tree from given array

We start with a segment arr[0 . . . n-1]. and every time we divide the current segment into two halves(if it has not yet become a segment of length 1), and then call the same procedure on both halves, and for each such segment we store the sum in corresponding node.

All levels of the constructed segment tree will be completely filled except the last level. Also, the tree will be a Full Binary Tree because we always divide segments in two halves at every level. Since the constructed tree is always full binary tree with n leaves, there will be n-1 internal nodes. So total number of nodes will be 2*n – 1.

Height of the segment tree will be $\lceil \log_2 n \rceil$. Since the tree is represented using array and relation between parent and child indexes must be maintained, size of memory allocated for segment tree will be $2 * 2^{\lceil \log_2 n \rceil} - 1$.

Query for Sum of given range

Once the tree is constructed, how to get the sum using the constructed segment tree. Following is algorithm to get the sum of elements.

```
int getSum(node, l, r)
{
   if range of node is within l and r
        return value in node
   else if range of node is completely outside l and r
        return 0
   else
    return getSum(node's left child, l, r) +
```

```
           getSum(node's right child, l, r)
}
```

Update a value
Like tree construction and query operations, update can also be done recursively. We are given an index which needs to updated. Let *diff* be the value to be added. We start from root of the segment tree, and add *diff* to all nodes which have given index in their range. If a node doesn't have given index in its range, we don't make any changes to that node.

Implementation:
Following is implementation of segment tree. The program implements construction of segment tree for any given array. It also implements query and update operations.

C

```
// C program to show segment tree operations like construction, query
// and update
#include <stdio.h>
#include <math.h>

// A utility function to get the middle index from corner indexes.
int getMid(int s, int e) {  return s + (e -s)/2;  }

/*  A recursive function to get the sum of values in given range
    of the array. The following are parameters for this function.

    st    --> Pointer to segment tree
    si    --> Index of current node in the segment tree. Initially
              0 is passed as root is always at index 0
    ss & se  --> Starting and ending indexes of the segment represented
                 by current node, i.e., st[si]
    qs & qe  --> Starting and ending indexes of query range */
int getSumUtil(int *st, int ss, int se, int qs, int qe, int si)
{
    // If segment of this node is a part of given range, then return
    // the sum of the segment
    if (qs <= ss && qe >= se)
        return st[si];

    // If segment of this node is outside the given range
    if (se < qs || ss > qe)
        return 0;

    // If a part of this segment overlaps with the given range
    int mid = getMid(ss, se);
    return getSumUtil(st, ss, mid, qs, qe, 2*si+1) +
           getSumUtil(st, mid+1, se, qs, qe, 2*si+2);
}

/* A recursive function to update the nodes which have the given
   index in their range. The following are parameters
    st, si, ss and se are same as getSumUtil()
    i    --> index of the element to be updated. This index is
             in input array.
```

```
   diff --> Value to be added to all nodes which have i in range */
void updateValueUtil(int *st, int ss, int se, int i, int diff, int si)
{
    // Base Case: If the input index lies outside the range of
    // this segment
    if (i < ss || i > se)
        return;

    // If the input index is in range of this node, then update
    // the value of the node and its children
    st[si] = st[si] + diff;
    if (se != ss)
    {
        int mid = getMid(ss, se);
        updateValueUtil(st, ss, mid, i, diff, 2*si + 1);
        updateValueUtil(st, mid+1, se, i, diff, 2*si + 2);
    }
}

// The function to update a value in input array and segment tree.
// It uses updateValueUtil() to update the value in segment tree
void updateValue(int arr[], int *st, int n, int i, int new_val)
{
    // Check for erroneous input index
    if (i < 0 || i > n-1)
    {
        printf("Invalid Input");
        return;
    }

    // Get the difference between new value and old value
    int diff = new_val - arr[i];

    // Update the value in array
    arr[i] = new_val;

    // Update the values of nodes in segment tree
    updateValueUtil(st, 0, n-1, i, diff, 0);
}

// Return sum of elements in range from index qs (quey start)
// to qe (query end).  It mainly uses getSumUtil()
int getSum(int *st, int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n-1 || qs > qe)
    {
        printf("Invalid Input");
        return -1;
    }

    return getSumUtil(st, 0, n-1, qs, qe, 0);
}
```

```
// A recursive function that constructs Segment Tree for array[ss..se].
// si is index of current node in segment tree st
int constructSTUtil(int arr[], int ss, int se, int *st, int si)
{
    // If there is one element in array, store it in current node of
    // segment tree and return
    if (ss == se)
    {
        st[si] = arr[ss];
        return arr[ss];
    }

    // If there are more than one elements, then recur for left and
    // right subtrees and store the sum of values in this node
    int mid = getMid(ss, se);
    st[si] =  constructSTUtil(arr, ss, mid, st, si*2+1) +
              constructSTUtil(arr, mid+1, se, st, si*2+2);
    return st[si];
}

/* Function to construct segment tree from given array. This function
   allocates memory for segment tree and calls constructSTUtil() to
   fill the allocated memory */
int *constructST(int arr[], int n)
{
    // Allocate memory for segment tree

    //Height of segment tree
    int x = (int)(ceil(log2(n)));

    //Maximum size of segment tree
    int max_size = 2*(int)pow(2, x) - 1;

    // Allocate memory
    int *st = new int[max_size];

    // Fill the allocated memory st
    constructSTUtil(arr, 0, n-1, st, 0);

    // Return the constructed segment tree
    return st;
}

// Driver program to test above functions
int main()
{
    int arr[] = {1, 3, 5, 7, 9, 11};
    int n = sizeof(arr)/sizeof(arr[0]);

    // Build segment tree from given array
    int *st = constructST(arr, n);

    // Print sum of values in array from index 1 to 3
    printf("Sum of values in given range = %d\n",
```

```
            getSum(st, n, 1, 3));

    // Update: set arr[1] = 10 and update corresponding
    // segment tree nodes
    updateValue(arr, st, n, 1, 10);

    // Find sum after the value is updated
    printf("Updated sum of values in given range = %d\n",
             getSum(st, n, 1, 3));
    return 0;
}
```

Java

```
// Java Program to show segment tree operations like construction,
// query and update
class SegmentTree
{
    int st[]; // The array that stores segment tree nodes

    /* Constructor to construct segment tree from given array. This
       constructor  allocates memory for segment tree and calls
       constructSTUtil() to  fill the allocated memory */
    SegmentTree(int arr[], int n)
    {
        // Allocate memory for segment tree
        //Height of segment tree
        int x = (int) (Math.ceil(Math.log(n) / Math.log(2)));

        //Maximum size of segment tree
        int max_size = 2 * (int) Math.pow(2, x) - 1;

        st = new int[max_size]; // Memory allocation

        constructSTUtil(arr, 0, n - 1, 0);
    }

    // A utility function to get the middle index from corner indexes.
    int getMid(int s, int e) {
        return s + (e - s) / 2;
    }

    /*  A recursive function to get the sum of values in given range
        of the array.  The following are parameters for this function.

      st    --> Pointer to segment tree
      si    --> Index of current node in the segment tree. Initially
                0 is passed as root is always at index 0
      ss & se  --> Starting and ending indexes of the segment represented
                    by current node, i.e., st[si]
      qs & qe  --> Starting and ending indexes of query range */
    int getSumUtil(int ss, int se, int qs, int qe, int si)
```

```
{
    // If segment of this node is a part of given range, then return
    // the sum of the segment
    if (qs <= ss && qe >= se)
        return st[si];

    // If segment of this node is outside the given range
    if (se < qs || ss > qe)
        return 0;

    // If a part of this segment overlaps with the given range
    int mid = getMid(ss, se);
    return getSumUtil(ss, mid, qs, qe, 2 * si + 1) +
            getSumUtil(mid + 1, se, qs, qe, 2 * si + 2);
}

/* A recursive function to update the nodes which have the given
   index in their range. The following are parameters
    st, si, ss and se are same as getSumUtil()
    i    --> index of the element to be updated. This index is in
             input array.
   diff --> Value to be added to all nodes which have i in range */
void updateValueUtil(int ss, int se, int i, int diff, int si)
{
    // Base Case: If the input index lies outside the range of
    // this segment
    if (i < ss || i > se)
        return;

    // If the input index is in range of this node, then update the
    // value of the node and its children
    st[si] = st[si] + diff;
    if (se != ss) {
        int mid = getMid(ss, se);
        updateValueUtil(ss, mid, i, diff, 2 * si + 1);
        updateValueUtil(mid + 1, se, i, diff, 2 * si + 2);
    }
}

// The function to update a value in input array and segment tree.
// It uses updateValueUtil() to update the value in segment tree
void updateValue(int arr[], int n, int i, int new_val)
{
    // Check for erroneous input index
    if (i < 0 || i > n - 1) {
        System.out.println("Invalid Input");
        return;
    }

    // Get the difference between new value and old value
    int diff = new_val - arr[i];

    // Update the value in array
    arr[i] = new_val;
```

```
    // Update the values of nodes in segment tree
    updateValueUtil(0, n - 1, i, diff, 0);
}

// Return sum of elements in range from index qs (quey start) to
// qe (query end).  It mainly uses getSumUtil()
int getSum(int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n - 1 || qs > qe) {
        System.out.println("Invalid Input");
        return -1;
    }
    return getSumUtil(0, n - 1, qs, qe, 0);
}

// A recursive function that constructs Segment Tree for array[ss..se].
// si is index of current node in segment tree st
int constructSTUtil(int arr[], int ss, int se, int si)
{
    // If there is one element in array, store it in current node of
    // segment tree and return
    if (ss == se) {
        st[si] = arr[ss];
        return arr[ss];
    }

    // If there are more than one elements, then recur for left and
    // right subtrees and store the sum of values in this node
    int mid = getMid(ss, se);
    st[si] = constructSTUtil(arr, ss, mid, si * 2 + 1) +
             constructSTUtil(arr, mid + 1, se, si * 2 + 2);
    return st[si];
}

// Driver program to test above functions
public static void main(String args[])
{
    int arr[] = {1, 3, 5, 7, 9, 11};
    int n = arr.length;
    SegmentTree  tree = new SegmentTree(arr, n);

    // Build segment tree from given array

    // Print sum of values in array from index 1 to 3
    System.out.println("Sum of values in given range = " +
                       tree.getSum(n, 1, 3));

    // Update: set arr[1] = 10 and update corresponding segment
    // tree nodes
    tree.updateValue(arr, n, 1, 10);

    // Find sum after the value is updated
```

```
        System.out.println("Updated sum of values in given range = " +
                tree.getSum(n, 1, 3));
    }
}
//This code is contributed by Ankur Narain Verma
```

Output:

```
Sum of values in given range = 15
Updated sum of values in given range = 22
```

Time Complexity:
Time Complexity for tree construction is O(n). There are total 2n-1 nodes, and value of every node is calculated only once in tree construction.

Time complexity to query is O(Logn). To query a sum, we process at most four nodes at every level and number of levels is O(Logn).

The time complexity of update is also O(Logn). To update a leaf value, we process one node at every level and number of levels is O(Logn).

Segment Tree | Set 2 (Range Minimum Query)

References:
http://www.cse.iitk.ac.in/users/aca/lop12/slides/06.pdf

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/segment-tree-set-1-sum-of-given-range/

Chapter 35

Segment Tree | Set 2 (Range Minimum Query)

We have introduced segment tree with a simple example in the previous post. In this post, Range Minimum Query problem is discussed as another example where Segment Tree can be used. Following is problem statement.

We have an array arr[0 . . . n-1]. We should be able to efficiently find the minimum value from index *qs* (query start) to *qe* (query end) where *0 . The array is static (elements are not deleted and inserted during the series of queries).*

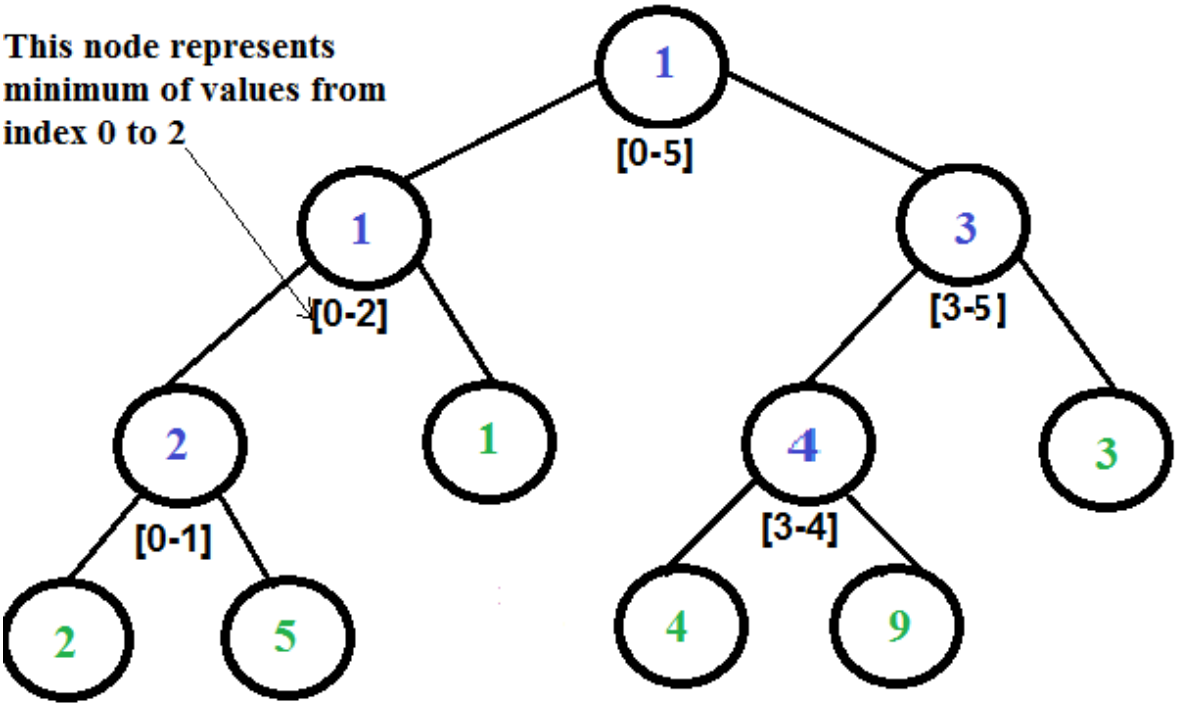
A **simple solution** is to run a loop from *qs* to *qe* and find minimum element in given range. This solution takes O(n) time in worst case.

Another solution is to create a 2D array where an entry [i, j] stores the minimum value in range arr[i..j]. Minimum of a given range can now be calculated in O(1) time, but preprocessing takes O(n^2) time. Also, this approach needs O(n^2) extra space which may become huge for large input arrays.

Segment tree can be used to do preprocessing and query in moderate time. With segment tree, preprocessing time is O(n) and time to for range minimum query is O(Logn). The extra space required is O(n) to store the segment tree.

Representation of Segment trees
1. Leaf Nodes are the elements of the input array.
2. Each internal node represents minimum of all leaves under it.

An array representation of tree is used to represent Segment Trees. For each node at index i, the left child is at index 2*i+1, right child at 2*i+2 and the parent is at $\lfloor (i-1)/2 \rfloor$.



Segment Tree for input array {2, 5, 1, 4, 9, 3 }

Construction of Segment Tree from given array

We start with a segment arr[0 . . . n-1]. and every time we divide the current segment into two halves(if it has not yet become a segment of length 1), and then call the same procedure on both halves, and for each such segment, we store the minimum value in a segment tree node.

All levels of the constructed segment tree will be completely filled except the last level. Also, the tree will be a Full Binary Tree because we always divide segments in two halves at every level. Since the constructed tree is always full binary tree with n leaves, there will be n-1 internal nodes. So total number of nodes will be 2*n – 1.

Height of the segment tree will be $\lceil \log_2 n \rceil$. Since the tree is represented using array and relation between parent and child indexes must be maintained, size of memory allocated for segment tree will be $2 * 2^{\lceil \log_2 n \rceil} - 1$.

Query for minimum value of given range

Once the tree is constructed, how to do range minimum query using the constructed segment tree. Following is algorithm to get the minimum.

```
// qs --> query start index, qe --> query end index
int RMQ(node, qs, qe)
{
   if range of node is within qs and qe
        return value in node
   else if range of node is completely outside qs and qe
        return INFINITE
   else
```

```
    return min( RMQ(node's left child, qs, qe), RMQ(node's right child, qs, qe) )
}
```

Implementation:

C

```
// C program for range minimum query using segment tree
#include <stdio.h>
#include <math.h>
#include <limits.h>

// A utility function to get minimum of two numbers
int minVal(int x, int y) { return (x < y)? x: y; }

// A utility function to get the middle index from corner indexes.
int getMid(int s, int e) {  return s + (e -s)/2;  }

/*  A recursive function to get the minimum value in a given range
     of array indexes. The following are parameters for this function.

    st    --> Pointer to segment tree
    index --> Index of current node in the segment tree. Initially
              0 is passed as root is always at index 0
    ss & se  --> Starting and ending indexes of the segment represented
                 by current node, i.e., st[index]
    qs & qe  --> Starting and ending indexes of query range */
int RMQUtil(int *st, int ss, int se, int qs, int qe, int index)
{
    // If segment of this node is a part of given range, then return
    //  the min of the segment
    if (qs <= ss && qe >= se)
        return st[index];

    // If segment of this node is outside the given range
    if (se < qs || ss > qe)
        return INT_MAX;

    // If a part of this segment overlaps with the given range
    int mid = getMid(ss, se);
    return minVal(RMQUtil(st, ss, mid, qs, qe, 2*index+1),
                  RMQUtil(st, mid+1, se, qs, qe, 2*index+2));
}

// Return minimum of elements in range from index qs (quey start) to
// qe (query end).  It mainly uses RMQUtil()
int RMQ(int *st, int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n-1 || qs > qe)
    {
        printf("Invalid Input");
        return -1;
```

```
    }

    return RMQUtil(st, 0, n-1, qs, qe, 0);
}

// A recursive function that constructs Segment Tree for array[ss..se].
// si is index of current node in segment tree st
int constructSTUtil(int arr[], int ss, int se, int *st, int si)
{
    // If there is one element in array, store it in current node of
    // segment tree and return
    if (ss == se)
    {
        st[si] = arr[ss];
        return arr[ss];
    }

    // If there are more than one elements, then recur for left and
    // right subtrees and store the minimum of two values in this node
    int mid = getMid(ss, se);
    st[si] =  minVal(constructSTUtil(arr, ss, mid, st, si*2+1),
                     constructSTUtil(arr, mid+1, se, st, si*2+2));
    return st[si];
}

/* Function to construct segment tree from given array. This function
   allocates memory for segment tree and calls constructSTUtil() to
   fill the allocated memory */
int *constructST(int arr[], int n)
{
    // Allocate memory for segment tree

    //Height of segment tree
    int x = (int)(ceil(log2(n)));

    // Maximum size of segment tree
    int max_size = 2*(int)pow(2, x) - 1;

    int *st = new int[max_size];

    // Fill the allocated memory st
    constructSTUtil(arr, 0, n-1, st, 0);

    // Return the constructed segment tree
    return st;
}

// Driver program to test above functions
int main()
{
    int arr[] = {1, 3, 2, 7, 9, 11};
    int n = sizeof(arr)/sizeof(arr[0]);

    // Build segment tree from given array
```

```
    int *st = constructST(arr, n);

    int qs = 1;  // Starting index of query range
    int qe = 5;  // Ending index of query range

    // Print minimum value in arr[qs..qe]
    printf("Minimum of values in range [%d, %d] is = %d\n",
                          qs, qe, RMQ(st, n, qs, qe));

    return 0;
}
```

Java

```
// Program for range minimum query using segment tree
class SegmentTreeRMQ
{
    int st[]; //array to store segment tree

    // A utility function to get minimum of two numbers
    int minVal(int x, int y) {
        return (x < y) ? x : y;
    }

    // A utility function to get the middle index from corner
    // indexes.
    int getMid(int s, int e) {
        return s + (e - s) / 2;
    }

    /*  A recursive function to get the minimum value in a given
        range of array indexes. The following are parameters for
        this function.

        st    --> Pointer to segment tree
        index --> Index of current node in the segment tree. Initially
                  0 is passed as root is always at index 0
        ss & se  --> Starting and ending indexes of the segment
                     represented by current node, i.e., st[index]
        qs & qe  --> Starting and ending indexes of query range */
    int RMQUtil(int ss, int se, int qs, int qe, int index)
    {
        // If segment of this node is a part of given range, then
        // return the min of the segment
        if (qs <= ss && qe >= se)
            return st[index];

        // If segment of this node is outside the given range
        if (se < qs || ss > qe)
            return Integer.MAX_VALUE;

        // If a part of this segment overlaps with the given range
```

```
    int mid = getMid(ss, se);
    return minVal(RMQUtil(ss, mid, qs, qe, 2 * index + 1),
            RMQUtil(mid + 1, se, qs, qe, 2 * index + 2));
}

// Return minimum of elements in range from index qs (quey
// start) to qe (query end).  It mainly uses RMQUtil()
int RMQ(int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n - 1 || qs > qe) {
        System.out.println("Invalid Input");
        return -1;
    }

    return RMQUtil(0, n - 1, qs, qe, 0);
}

// A recursive function that constructs Segment Tree for
// array[ss..se]. si is index of current node in segment tree st
int constructSTUtil(int arr[], int ss, int se, int si)
{
    // If there is one element in array, store it in current
    //  node of segment tree and return
    if (ss == se) {
        st[si] = arr[ss];
        return arr[ss];
    }

    // If there are more than one elements, then recur for left and
    // right subtrees and store the minimum of two values in this node
    int mid = getMid(ss, se);
    st[si] = minVal(constructSTUtil(arr, ss, mid, si * 2 + 1),
            constructSTUtil(arr, mid + 1, se, si * 2 + 2));
    return st[si];
}

/* Function to construct segment tree from given array. This function
   allocates memory for segment tree and calls constructSTUtil() to
   fill the allocated memory */
void constructST(int arr[], int n)
{
    // Allocate memory for segment tree

    //Height of segment tree
    int x = (int) (Math.ceil(Math.log(n) / Math.log(2)));

    //Maximum size of segment tree
    int max_size = 2 * (int) Math.pow(2, x) - 1;
    st = new int[max_size]; // allocate memory

    // Fill the allocated memory st
    constructSTUtil(arr, 0, n - 1, 0);
}
```

```
    // Driver program to test above functions
    public static void main(String args[])
    {
        int arr[] = {1, 3, 2, 7, 9, 11};
        int n = arr.length;
        SegmentTreeRMQ tree = new SegmentTreeRMQ();

        // Build segment tree from given array
        tree.constructST(arr, n);

        int qs = 1;  // Starting index of query range
        int qe = 5;  // Ending index of query range

        // Print minimum value in arr[qs..qe]
        System.out.println("Minimum of values in range [" + qs + ", "
                           + qe + "] is = " + tree.RMQ(n, qs, qe));
    }
}
// This code is contributed by Ankur Narain Verma
```

Output:

```
Minimum of values in range [1, 5] is = 2
```

Time Complexity:
Time Complexity for tree construction is O(n). There are total 2n-1 nodes, and value of every node is calculated only once in tree construction.

Time complexity to query is O(Logn). To query a range minimum, we process at most two nodes at every level and number of levels is O(Logn).

Please refer following links for more solutions to range minimum query problem.

http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor#Range_Minimum_Query_(RMQ)
http://wcipeg.com/wiki/Range_minimum_query

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

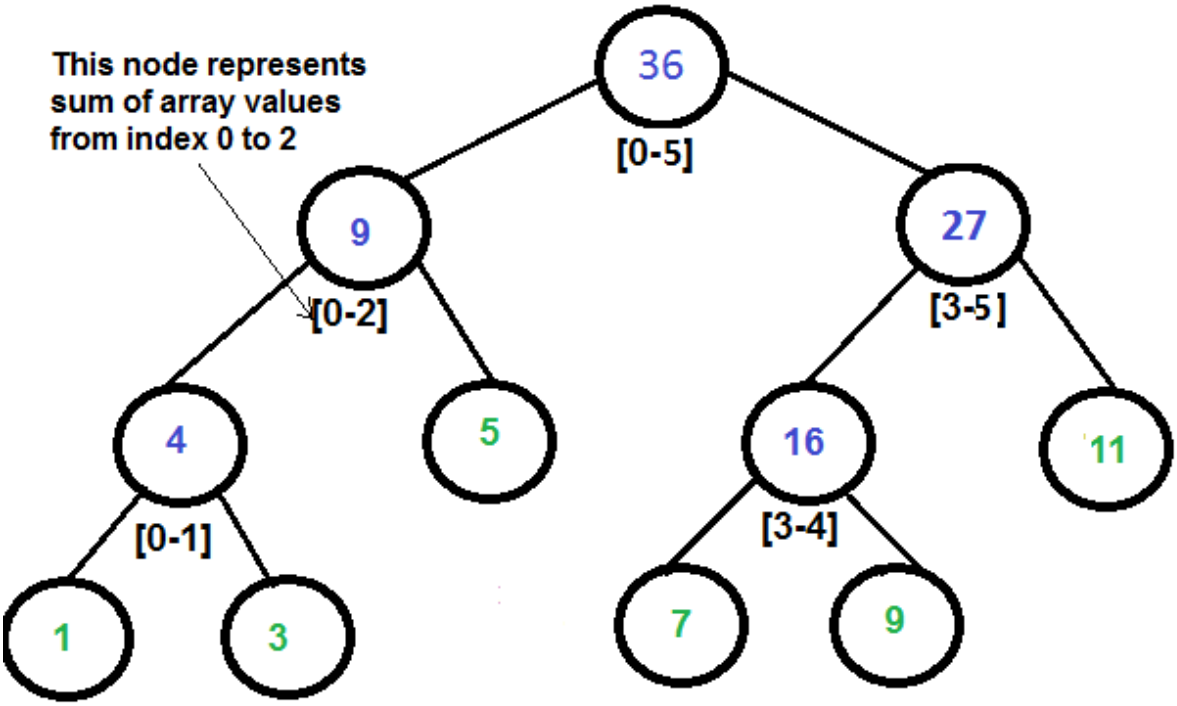
Source

http://www.geeksforgeeks.org/segment-tree-set-1-range-minimum-query/

Chapter 36

Lazy Propagation in Segment Tree

Segment tree is introduced in previous post with an example of range sum problem. We have used the same "Sum of given Range" problem to explain Lazy propagation



Segment Tree for input array {1, 3, 5, 7, 9, 11}

How does update work in Simple Segment Tree?
In the previous post, update function was called to update only a single value in array. Please note that a single value update in array may cause multiple updates in Segment Tree as there may be many segment tree nodes that have a single array element in their ranges.

Below is simple logic used in previous post.
1) Start with root of segment tree.

2) If array index to be updated is not in current node's range, then return
3) Else update current node and recur for children.

Below is code taken from previous post.

```
/* A recursive function to update the nodes which have the given
   index in their range. The following are parameters
    tree[] --> segment tree
    si     -->  index of current node in segment tree.
                Initial value is passed as 0.
    ss and se --> Starting and ending indexes of array elements
                  covered under this node of segment tree.
                  Initial values passed as 0 and n-1.
    i    --> index of the element to be updated. This index
             is in input array.
   diff --> Value to be added to all nodes which have array
             index i in range */
void updateValueUtil(int tree[], int ss, int se, int i,
                     int diff, int si)
{
    // Base Case: If the input index lies outside the range
    // of this segment
    if (i < ss || i > se)
        return;

    // If the input index is in range of this node, then
    // update the value of the node and its children
    st[si] = st[si] + diff;
    if (se != ss)
    {
        int mid = getMid(ss, se);
        updateValueUtil(st, ss, mid, i, diff, 2*si + 1);
        updateValueUtil(st, mid+1, se, i, diff, 2*si + 2);
    }
}
```

What if there are updates on a range of array indexes?
For example add 10 to all values at indexes from 2 to 7 in array. The above update has to be called for every index from 2 to 7. We can avoid multiple calls by writing a function updateRange() that updates nodes accordingly.

```
/* Function to update segment tree for range update in input
   array.
    si -> index of current node in segment tree
    ss and se -> Starting and ending indexes of elements for
                 which current nodes stores sum.
    us and eu -> starting and ending indexes of update query
    ue  -> ending index of update query
    diff -> which we need to add in the range us to ue */
void updateRangeUtil(int si, int ss, int se, int us,
                     int ue, int diff)
{
```

```
    // out of range
    if (ss>se || ss>ue || se<us)
        return ;

    // Current node is a leaf node
    if (ss==se)
    {
        // Add the difference to current node
        tree[si] += diff;
        return;
    }

    // If not a leaf node, recur for children.
    int mid = (ss+se)/2;
    updateRangeUtil(si*2+1, ss, mid, us, ue, diff);
    updateRangeUtil(si*2+2, mid+1, se, us, ue, diff);

    // Use the result of children calls to update this
    // node
    tree[si] = tree[si*2+1] + tree[si*2+2];
}
```

Lazy Propagation – An optimization to make range updates faster

When there are many updates and updates are done on a range, we can postpone some updates (avoid recursive calls in update) and do those updates only when required.

Please remember that a node in segment tree stores or represents result of a query for a range of indexes. And if this node's range lies within the update operation range, then all descendants of the node must also be updated. For example consider the node with value 27 in above diagram, this node stores sum of values at indexes from 3 to 5. If our update query is for range 2 to 5, then we need to update this node and all descendants of this node. With Lazy propagation, we update only node with value 27 and postpone updates to its children by storing this update information in separate nodes called lazy nodes or values. We create an array lazy[] which represents lazy node. Size of lazy[] is same as array that represents segment tree, which is tree[] in below code.

The idea is to initialize all elements of lazy[] as 0. A value 0 in lazy[i] indicates that there are no pending updates on node i in segment tree. A non-zero value of lazy[i] means that this amount needs to be added to node i in segment tree before making any query to the node.

Below is modified update method.

```
// To update segment tree for change in array
// values at array indexes from us to ue.
updateRange(us, ue)
1) If current segment tree node has any pending
   update, then first add that pending update to
   current node.
2) If current node's range lies completely in
   update query range.
....a) Update current node
....b) Postpone updates to children by setting
       lazy value for children nodes.
3) If current node's range overlaps with update
   range, follow the same approach as above simple
```

```
   update.
...a) Recur for left and right children.
...b) Update current node using results of left
      and right calls.
```

Is there any change in Query Function also?
Since we have changed update to postpone its operations, there may be problems if a query is made to a node that is yet to be updated. So we need to update our query method also which is getSumUtil in previous post. The getSumUtil() now first checks if there is a pending update and if there is, then updates the node. Once it makes sure that pending update is done, it works same as the previous getSumUtil().

Below are programs to demonstrate working of Lazy Propagation.

C/C++

```
// Program to show segment tree to demonstrate lazy
// propagation
#include <stdio.h>
#include <math.h>
#define MAX 1000

// Ideally, we should not use global variables and large
// constant-sized arrays, we have done it here for simplicity.
int tree[MAX] = {0};  // To store segment tree
int lazy[MAX] = {0};  // To store pending updates

/*  si -> index of current node in segment tree
    ss and se -> Starting and ending indexes of elements for
                 which current nodes stores sum.
    us and eu -> starting and ending indexes of update query
    ue  -> ending index of update query
    diff -> which we need to add in the range us to ue */
void updateRangeUtil(int si, int ss, int se, int us,
                     int ue, int diff)
{
    // If lazy value is non-zero for current node of segment
    // tree, then there are some pending updates. So we need
    // to make sure that the pending updates are done before
    // making new updates. Because this value may be used by
    // parent after recursive calls (See last line of this
    // function)
    if (lazy[si] != 0)
    {
        // Make pending updates using value stored in lazy
        // nodes
        tree[si] += (se-ss+1)*lazy[si];

        // checking if it is not leaf node because if
        // it is leaf node then we cannot go further
        if (ss != se)
        {
            // We can postpone updating children we don't
            // need their new values now.
            // Since we are not yet updating children of si,
```

```
            // we need to set lazy flags for the children
            lazy[si*2 + 1]   += lazy[si];
            lazy[si*2 + 2]   += lazy[si];
        }

        // Set the lazy value for current node as 0 as it
        // has been updated
        lazy[si] = 0;
    }

    // out of range
    if (ss>se || ss>ue || se<us)
        return ;

    // Current segment is fully in range
    if (ss>=us && se<=ue)
    {
        // Add the difference to current node
        tree[si] += (se-ss+1)*diff;

        // same logic for checking leaf node or not
        if (ss != se)
        {
            // This is where we store values in lazy nodes,
            // rather than updating the segment tree itelf
            // Since we don't need these updated values now
            // we postpone updates by storing values in lazy[]
            lazy[si*2 + 1]   += diff;
            lazy[si*2 + 2]   += diff;
        }
        return;
    }

    // If not completely in rang, but overlaps, recur for
    // children,
    int mid = (ss+se)/2;
    updateRangeUtil(si*2+1, ss, mid, us, ue, diff);
    updateRangeUtil(si*2+2, mid+1, se, us, ue, diff);

    // And use the result of children calls to update this
    // node
    tree[si] = tree[si*2+1] + tree[si*2+2];
}

// Function to update a range of values in segment
// tree
/*  us and eu -> starting and ending indexes of update query
    ue  -> ending index of update query
    diff -> which we need to add in the range us to ue */
void updateRange(int n, int us, int ue, int diff)
{
   updateRangeUtil(0, 0, n-1, us, ue, diff);
}
```

```
/*  A recursive function to get the sum of values in given
    range of the array. The following are parameters for
    this function.
    si --> Index of current node in the segment tree.
           Initially 0 is passed as root is always at'
           index 0
    ss & se  --> Starting and ending indexes of the
                 segment represented by current node,
                 i.e., tree[si]
    qs & qe  --> Starting and ending indexes of query
                 range */
int getSumUtil(int ss, int se, int qs, int qe, int si)
{
    // If lazy flag is set for current node of segment tree,
    // then there are some pending updates. So we need to
    // make sure that the pending updates are done before
    // processing the sub sum query
    if (lazy[si] != 0)
    {
        // Make pending updates to this node. Note that this
        // node represents sum of elements in arr[ss..se] and
        // all these elements must be increased by lazy[si]
        tree[si] += (se-ss+1)*lazy[si];

        // checking if it is not leaf node because if
        // it is leaf node then we cannot go further
        if (ss != se)
        {
            // Since we are not yet updating children os si,
            // we need to set lazy values for the children
            lazy[si*2+1] += lazy[si];
            lazy[si*2+2] += lazy[si];
        }

        // unset the lazy value for current node as it has
        // been updated
        lazy[si] = 0;
    }

    // Out of range
    if (ss>se || ss>qe || se<qs)
        return 0;

    // At this point we are sure that pending lazy updates
    // are done for current node. So we can return value
    // (same as it was for query in our previous post)

    // If this segment lies in range
    if (ss>=qs && se<=qe)
        return tree[si];

    // If a part of this segment overlaps with the given
    // range
```

```
    int mid = (ss + se)/2;
    return getSumUtil(ss, mid, qs, qe, 2*si+1) +
           getSumUtil(mid+1, se, qs, qe, 2*si+2);
}

// Return sum of elements in range from index qs (quey
// start) to qe (query end).  It mainly uses getSumUtil()
int getSum(int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n-1 || qs > qe)
    {
        printf("Invalid Input");
        return -1;
    }

    return getSumUtil(0, n-1, qs, qe, 0);
}

// A recursive function that constructs Segment Tree for
//  array[ss..se]. si is index of current node in segment
// tree st.
void constructSTUtil(int arr[], int ss, int se, int si)
{
    // out of range as ss can never be greater than se
    if (ss > se)
        return ;

    // If there is one element in array, store it in
    // current node of segment tree and return
    if (ss == se)
    {
        tree[si] = arr[ss];
        return;
    }

    // If there are more than one elements, then recur
    // for left and right subtrees and store the sum
    // of values in this node
    int mid = (ss + se)/2;
    constructSTUtil(arr, ss, mid, si*2+1);
    constructSTUtil(arr, mid+1, se, si*2+2);

    tree[si] = tree[si*2 + 1] + tree[si*2 + 2];
}

/* Function to construct segment tree from given array.
   This function allocates memory for segment tree and
   calls constructSTUtil() to fill the allocated memory */
void constructST(int arr[], int n)
{
    // Fill the allocated memory st
    constructSTUtil(arr, 0, n-1, 0);
}
```

```
// Driver program to test above functions
int main()
{
    int arr[] = {1, 3, 5, 7, 9, 11};
    int n = sizeof(arr)/sizeof(arr[0]);

    // Build segment tree from given array
    constructST(arr, n);

    // Print sum of values in array from index 1 to 3
    printf("Sum of values in given range = %d\n",
           getSum(n, 1, 3));

    // Add 10 to all nodes at indexes from 1 to 5.
    updateRange(n, 1, 5, 10);

    // Find sum after the value is updated
    printf("Updated sum of values in given range = %d\n",
            getSum( n, 1, 3));

    return 0;
}
```

Java

```
// Java program to demonstrate lazy propagation in segment tree
class LazySegmentTree
{
    final int MAX = 1000;        // Max tree size
    int tree[] = new int[MAX];  // To store segment tree
    int lazy[] = new int[MAX];  // To store pending updates

    /*  si -> index of current node in segment tree
        ss and se -> Starting and ending indexes of elements for
                     which current nodes stores sum.
        us and eu -> starting and ending indexes of update query
        ue  -> ending index of update query
        diff -> which we need to add in the range us to ue */
    void updateRangeUtil(int si, int ss, int se, int us,
                         int ue, int diff)
    {
        // If lazy value is non-zero for current node of segment
        // tree, then there are some pending updates. So we need
        // to make sure that the pending updates are done before
        // making new updates. Because this value may be used by
        // parent after recursive calls (See last line of this
        // function)
        if (lazy[si] != 0)
        {
            // Make pending updates using value stored in lazy
```

```
        // nodes
        tree[si] += (se - ss + 1) * lazy[si];

        // checking if it is not leaf node because if
        // it is leaf node then we cannot go further
        if (ss != se)
        {
            // We can postpone updating children we don't
            // need their new values now.
            // Since we are not yet updating children of si,
            // we need to set lazy flags for the children
            lazy[si * 2 + 1] += lazy[si];
            lazy[si * 2 + 2] += lazy[si];
        }

        // Set the lazy value for current node as 0 as it
        // has been updated
        lazy[si] = 0;
    }

    // out of range
    if (ss > se || ss > ue || se < us)
        return;

    // Current segment is fully in range
    if (ss >= us && se <= ue)
    {
        // Add the difference to current node
        tree[si] += (se - ss + 1) * diff;

        // same logic for checking leaf node or not
        if (ss != se)
        {
            // This is where we store values in lazy nodes,
            // rather than updating the segment tree itelf
            // Since we don't need these updated values now
            // we postpone updates by storing values in lazy[]
            lazy[si * 2 + 1] += diff;
            lazy[si * 2 + 2] += diff;
        }
        return;
    }

    // If not completely in rang, but overlaps, recur for
    // children,
    int mid = (ss + se) / 2;
    updateRangeUtil(si * 2 + 1, ss, mid, us, ue, diff);
    updateRangeUtil(si * 2 + 2, mid + 1, se, us, ue, diff);

    // And use the result of children calls to update this
    // node
    tree[si] = tree[si * 2 + 1] + tree[si * 2 + 2];
}
```

```
// Function to update a range of values in segment
// tree
/*  us and eu -> starting and ending indexes of update query
    ue  -> ending index of update query
    diff -> which we need to add in the range us to ue */
void updateRange(int n, int us, int ue, int diff)  {
    updateRangeUtil(0, 0, n - 1, us, ue, diff);
}

/*  A recursive function to get the sum of values in given
    range of the array. The following are parameters for
    this function.
    si --> Index of current node in the segment tree.
           Initially 0 is passed as root is always at'
           index 0
    ss & se  --> Starting and ending indexes of the
                 segment represented by current node,
                 i.e., tree[si]
    qs & qe  --> Starting and ending indexes of query
                 range */
int getSumUtil(int ss, int se, int qs, int qe, int si)
{
    // If lazy flag is set for current node of segment tree,
    // then there are some pending updates. So we need to
    // make sure that the pending updates are done before
    // processing the sub sum query
    if (lazy[si] != 0)
    {
        // Make pending updates to this node. Note that this
        // node represents sum of elements in arr[ss..se] and
        // all these elements must be increased by lazy[si]
        tree[si] += (se - ss + 1) * lazy[si];

        // checking if it is not leaf node because if
        // it is leaf node then we cannot go further
        if (ss != se)
        {
            // Since we are not yet updating children os si,
            // we need to set lazy values for the children
            lazy[si * 2 + 1] += lazy[si];
            lazy[si * 2 + 2] += lazy[si];
        }

        // unset the lazy value for current node as it has
        // been updated
        lazy[si] = 0;
    }

    // Out of range
    if (ss > se || ss > qe || se < qs)
        return 0;

    // At this point sure, pending lazy updates are done
    // for current node. So we can return value (same as
```

```
    // was for query in our previous post)

    // If this segment lies in range
    if (ss >= qs && se <= qe)
        return tree[si];

    // If a part of this segment overlaps with the given
    // range
    int mid = (ss + se) / 2;
    return getSumUtil(ss, mid, qs, qe, 2 * si + 1) +
           getSumUtil(mid + 1, se, qs, qe, 2 * si + 2);
}

// Return sum of elements in range from index qs (query
// start) to qe (query end).  It mainly uses getSumUtil()
int getSum(int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n - 1 || qs > qe)
    {
        System.out.println("Invalid Input");
        return -1;
    }

    return getSumUtil(0, n - 1, qs, qe, 0);
}

/* A recursive function that constructs Segment Tree for
  array[ss..se]. si is index of current node in segment
  tree st. */
void constructSTUtil(int arr[], int ss, int se, int si)
{
    // out of range as ss can never be greater than se
    if (ss > se)
        return;

    /* If there is one element in array, store it in
     current node of segment tree and return */
    if (ss == se)
    {
        tree[si] = arr[ss];
        return;
    }

    /* If there are more than one elements, then recur
       for left and right subtrees and store the sum
       of values in this node */
    int mid = (ss + se) / 2;
    constructSTUtil(arr, ss, mid, si * 2 + 1);
    constructSTUtil(arr, mid + 1, se, si * 2 + 2);

    tree[si] = tree[si * 2 + 1] + tree[si * 2 + 2];
}
```

```
    /* Function to construct segment tree from given array.
       This function allocates memory for segment tree and
       calls constructSTUtil() to fill the allocated memory */
    void constructST(int arr[], int n)
    {
        // Fill the allocated memory st
        constructSTUtil(arr, 0, n - 1, 0);
    }


    // Driver program to test above functions
    public static void main(String args[])
    {
        int arr[] = {1, 3, 5, 7, 9, 11};
        int n = arr.length;
        LazySegmentTree tree = new LazySegmentTree();

        // Build segment tree from given array
        tree.constructST(arr, n);

        // Print sum of values in array from index 1 to 3
        System.out.println("Sum of values in given range = " +
                           tree.getSum(n, 1, 3));

        // Add 10 to all nodes at indexes from 1 to 5.
        tree.updateRange(n, 1, 5, 10);

        // Find sum after the value is updated
        System.out.println("Updated sum of values in given range = " +
                           tree.getSum(n, 1, 3));
    }
}
// This Code is contributed by Ankur Narain Verma
```

Output:

```
 Sum of values in given range = 15
Updated sum of values in given range = 45
```

This article is contributed by **Ankit Mittal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

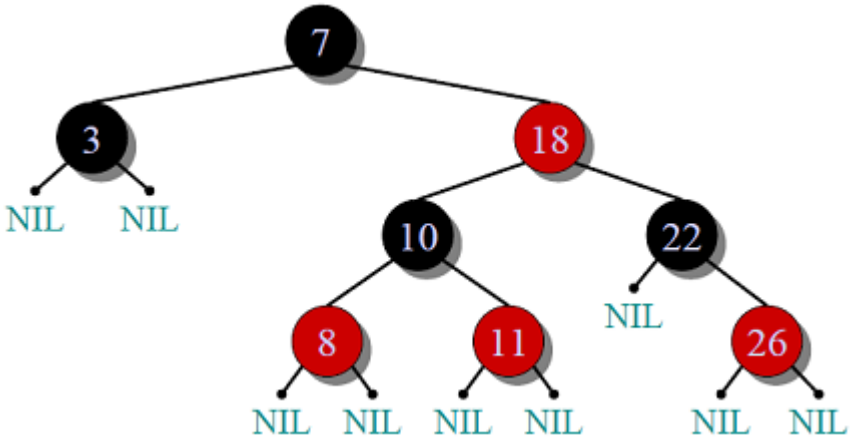
Source

http://www.geeksforgeeks.org/lazy-propagation-in-segment-tree/

Chapter 37

Red-Black Tree | Set 1 (Introduction)

Red-Black Tree is a self-balancing Binary Search Tree (BST) where every node follows following rules.



1) Every node has a color either red or black.

2) Root of tree is always black.

3) There are no two adjacent red nodes (A red node cannot have a red parent or red child).

4) Every path from root to a NULL node has same number of black nodes.

Why Red-Black Trees?
Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that height of the tree remains O(Logn) after every insertion and deletion, then we can guarantee an upper bound of O(Logn) for all these operations. The height of a Red Black tree is always O(Logn) where n is the number of nodes in the tree.

Comparison withAVL Tree
The AVL trees are more balanced compared to Red Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves many frequent insertions and deletions, then Red Black trees should be preferred. And if the insertions and deletions are less frequent and search is more frequent operation, then AVL tree should be preferred over Red Black Tree.

How does a Red-Black Tree ensure balance?
A simple example to understand balancing is, a chain of 3 nodes is not possible in red black tree. We can try any combination of colors and see all of them violate Red-Black tree property.

```
 A chain of 3 nodes is nodes is not possible in Red-Black Trees.
Following are NOT Red-Black Trees
        30              30               30
       / \             /  \             /  \
     20  NIL         20   NIL         20   NIL
    / \             / \              /  \
  10  NIL          10  NIL          10  NIL
Violates          Violates        Violates
Property 4.      Property 4       Property 3

Following are different possible Red-Black Trees with above 3 keys
         20                            20
       /   \                         /   \
     10     30                     10     30
    /  \   /  \                   /  \    /  \
 NIL  NIL NIL NIL              NIL  NIL NIL NIL
```

From the above examples, we get some idea how Red-Black trees ensure balance. Following is an important fact about balancing in Red-Black Trees.

Every Red Black Tree with n nodes has height <= $2\text{Log}_2(n+1)$

This can be proved using following facts:
1) For a general Binary Tree, let **k** be the minimum number of nodes on all root to NULL paths, then n >= $2^k - 1$ (Ex. If k is 3, then n is atleast 7). This expression can also be written as k <= $2\text{Log}_2(n+1)$

2) From property 4 of Red-Black trees and above claim, we can say in a Red-Black Tree with n nodes, there is a root to leaf path with at-most $\text{Log}_2(n+1)$ black nodes.

3) From property 3 of Red-Black trees, we can claim that the number black nodes in a Red-Black tree is at least $n/2$ where n is total number of nodes.

From above 2 points, we can conclude the fact that Red Black Tree with **n** nodes has height <= $2\text{Log}_2(n+1)$

In this post, we introduced Red-Black trees and discussed how balance is ensured. The hard part is to maintain balance when keys are added and removed. We will soon be discussing insertion and deletion operations in coming posts on Red-Black tree.

Exercise:
1) Is it possible to have all black nodes in a Red-Black tree?
2) Draw a Red-Black Tree that is not an AVL tree structure wise?

Insertion and Deletion
Red Black Tree Insertion

Red-Black Tree Deletion

References:

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest
http://en.wikipedia.org/wiki/Red%E2%80%93black_tree
Video Lecture on Red-Black Tree by Tim Roughgarden

MIT Video Lecture on Red-Black Tree

MIT Lecture Notes on Red Black Tree

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/

Chapter 38

Red Black Tree Insertion.

In the previous post, we discussed introduction to Red-Black Trees. In this post, insertion is discussed.

InAVL tree insertion, we used rotation as a tool to do balancing after insertion caused imbalance. In Red-Black tree, we use two tools to do balancing.

1) Recoloring
2) Rotation

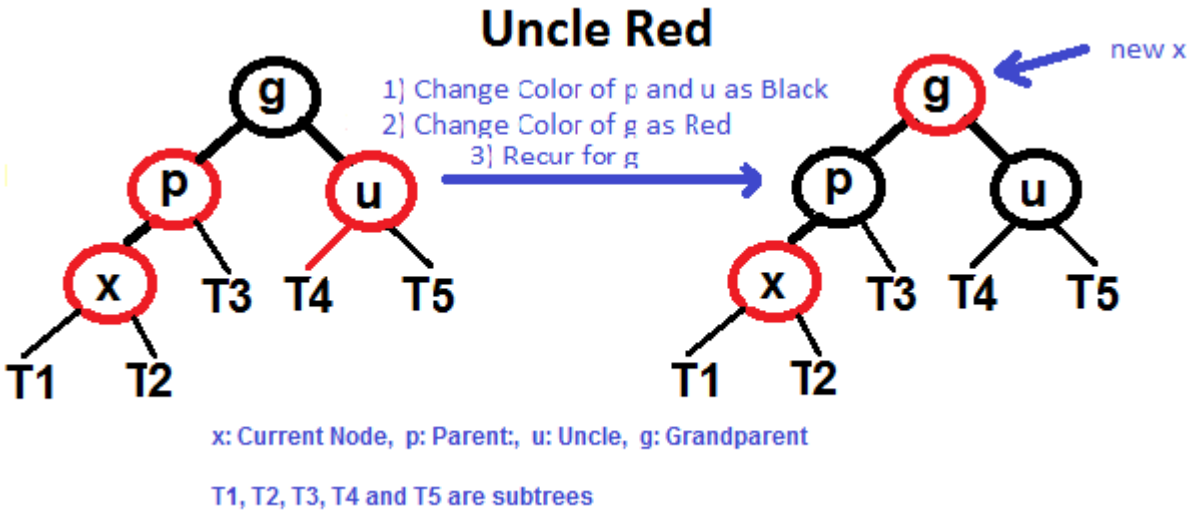
We try recoloring first, if recoloring doesn't work, then we go for rotation. Following is detailed algorithm. The algorithms has mainly two cases depending upon the color of uncle. If uncle is red, we do recoloring. If uncle is black, we do rotations and/or recoloring.

Color of a NULL node is considered as BLACK.

Let x be the newly inserted node.
1) Perform standard BST insertion and make the color of newly inserted nodes as RED.

2) If x is root, change color of x as BLACK (Black height of complete tree increases by 1).

3) Do following if color of x's parent is not BLACK or x is not root.
....**a) If x's uncle is RED** (Grand parent must have been black from property 4)
.....**(i)** Change color of parent and uncle as BLACK.
.....**(ii)** color of grand parent as RED.
.....**(iii)** Change x = x's grandparent, repeat steps 2 and 3 for new x.



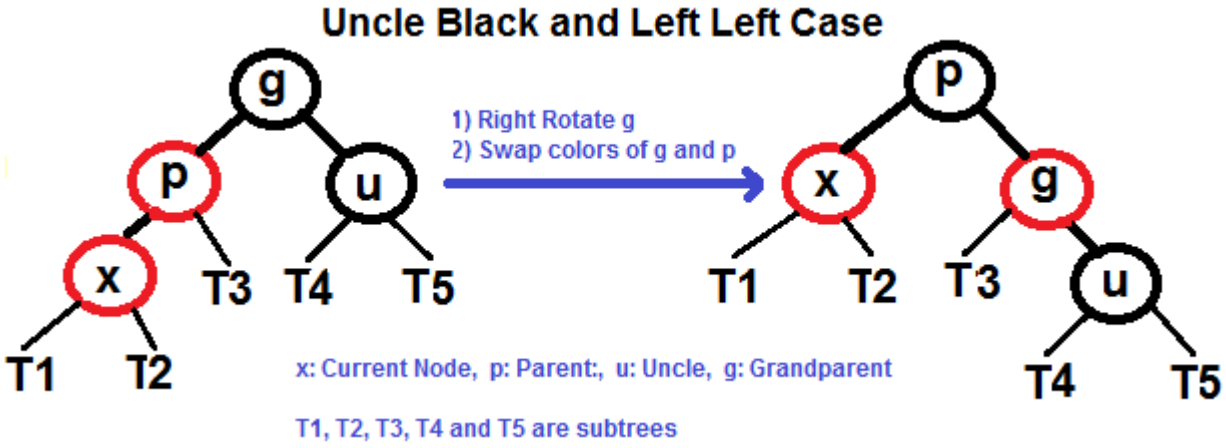
….**b) If x's uncle is BLACK**, then there can be four configurations for x, x's parent (**p**) and x's grandparent (**g**) (This is similar toAVL Tree)
……..**i)** Left Left Case (p is left child of g and x is left child of p)
……..**ii)** Left Right Case (p is left child of g and x is right child of p)
……..**iii)** Right Right Case (Mirror of case a)
……..**iv)** Right Left Case (Mirror of case c)

Following are operations to be performed in four subcases when uncle is BLACK.

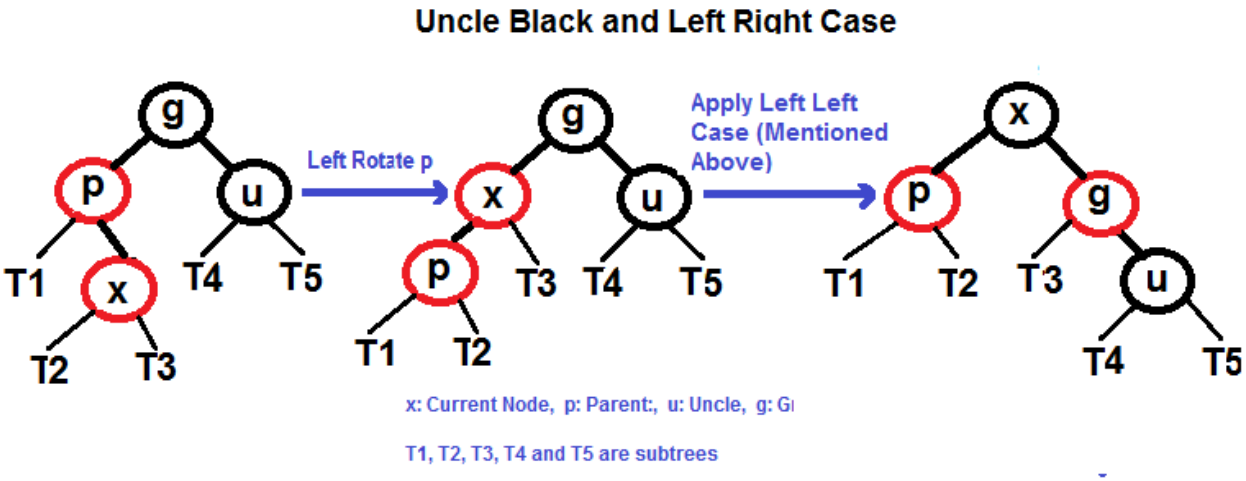
Chapter 39

All four cases when Uncle is BLACK

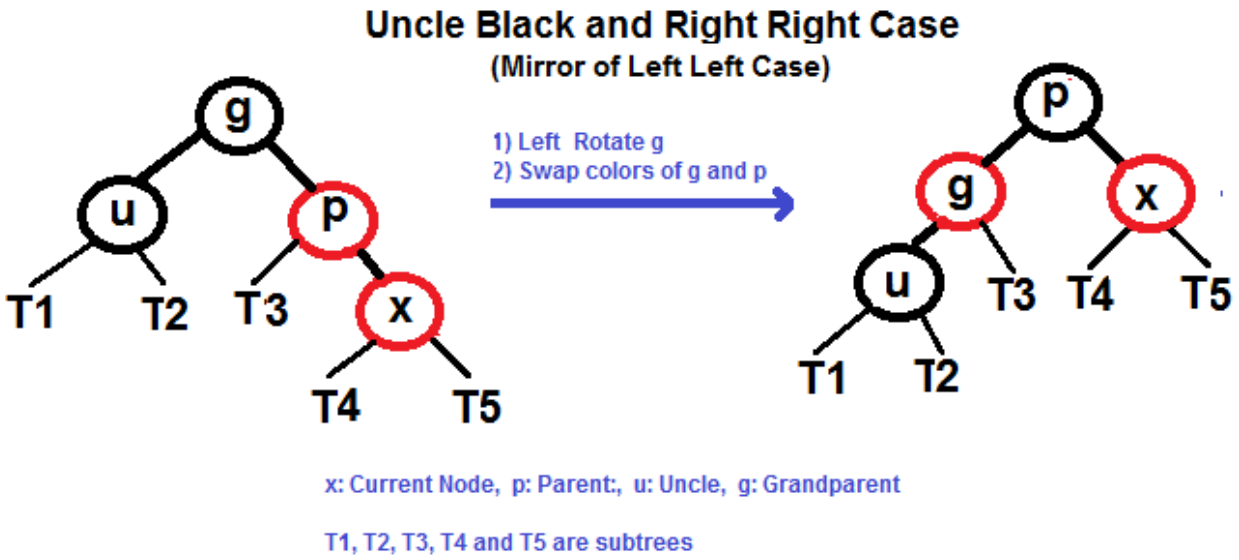
Left Left Case (See g, p and x)



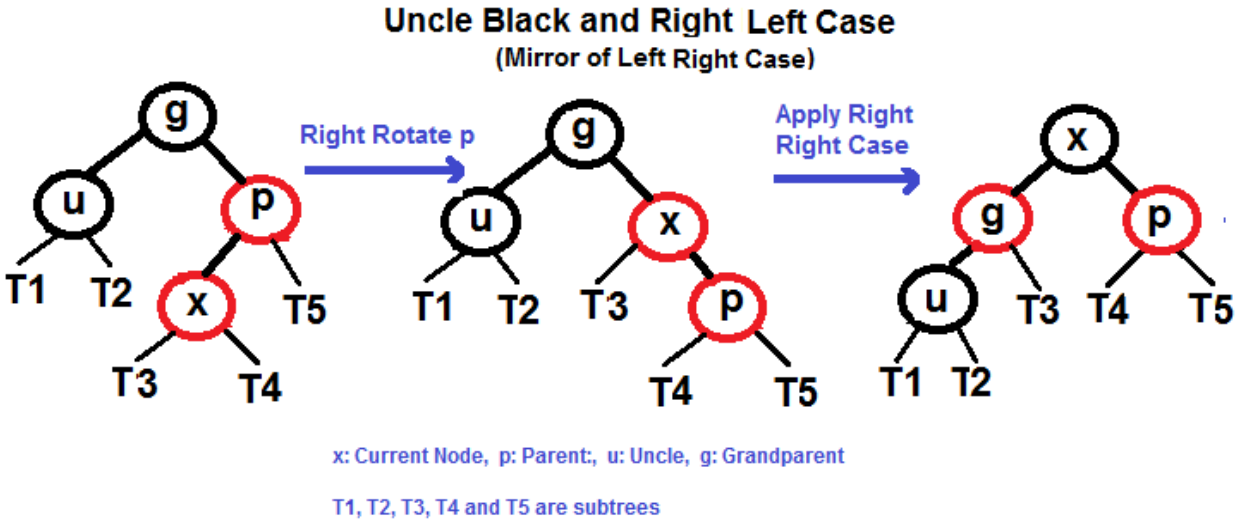
Left Right Case (See g, p and x)



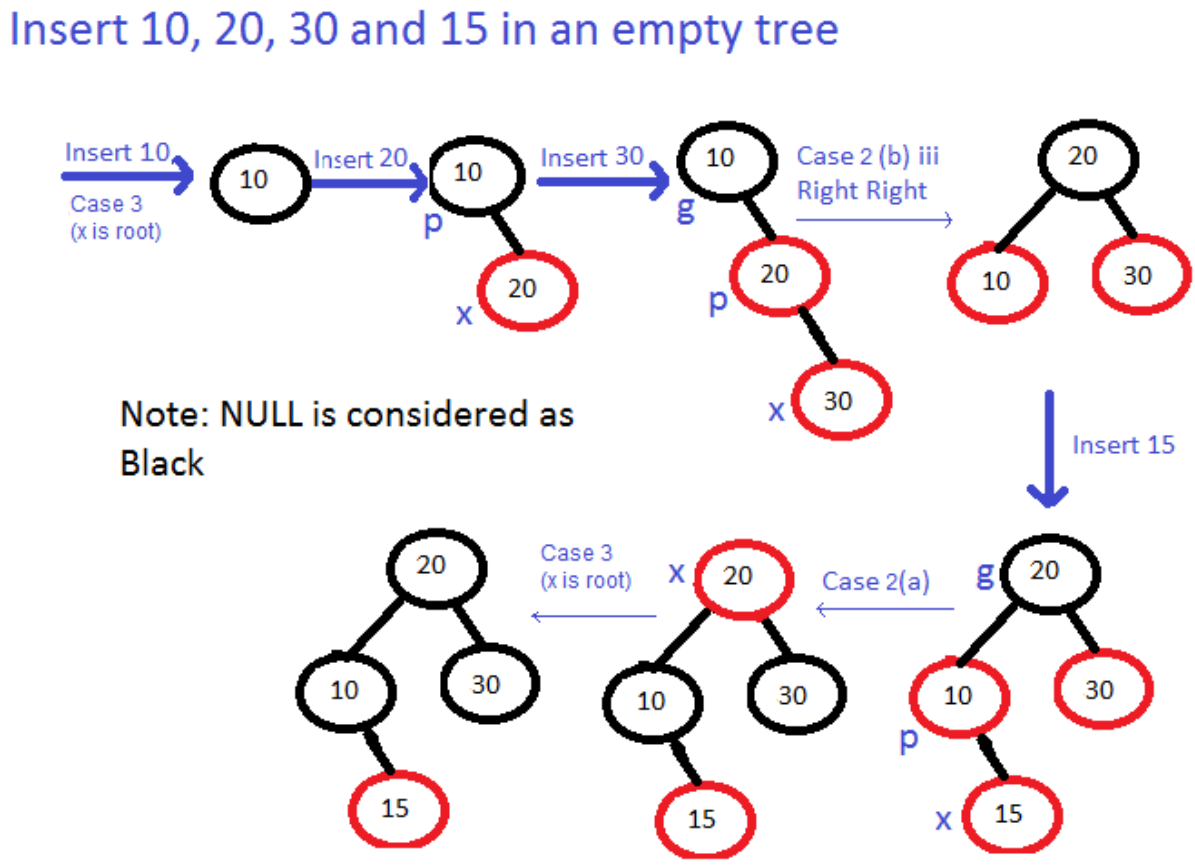
Right Right Case (See g, p and x)



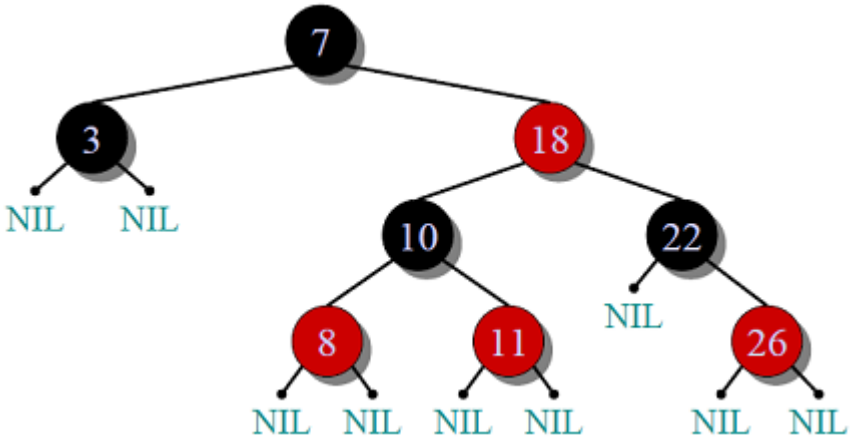
Right Left Case (See g, p and x)



Examples of Insertion



Exercise:
Insert 2, 7 and 13 in below tree. Insertion of 13 is going to be really interesting, try it to check if you have understood insertion well for exams.



Please refer C Program for Red Black Tree Insertion for complete implementation of above algorithm.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/red-black-tree-set-2-insert/

Chapter 40

Red-Black Tree | Set 3 (Delete)

We have discussed following topics on Red-Black tree in previous posts. We strongly recommend to refer following post as prerequisite of this post.

Red-Black Tree Introduction
Red Black Tree Insert

Insertion Vs Deletion:
Like Insertion, recoloring and rotations are used to maintain the Red-Black properties.

In insert operation, we check color of uncle to decide the appropriate case. In delete operation, ***we check color of sibling*** to decide the appropriate case.

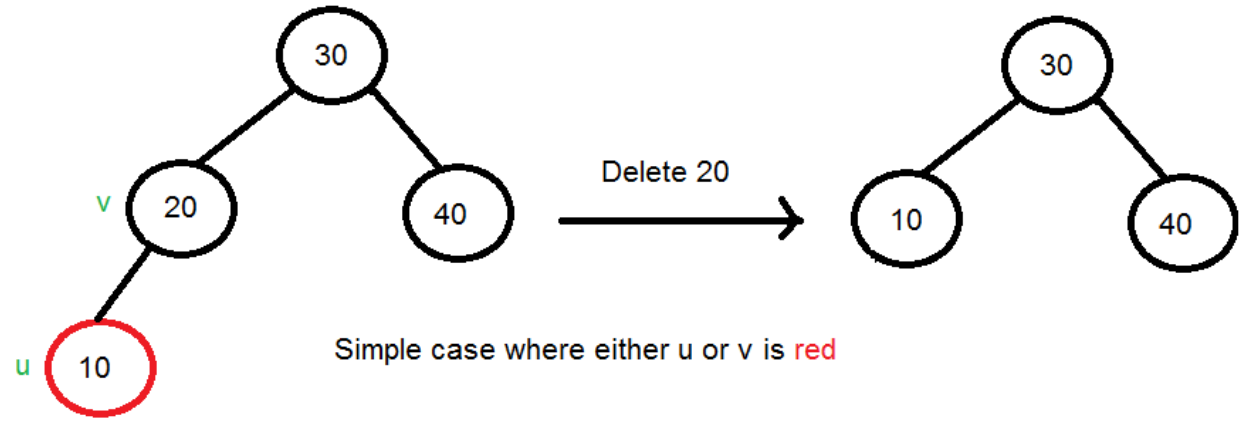
The main property that violates after insertion is two consecutive reds. In delete, the main violated property is, change of black height in subtrees as deletion of a black node may cause reduced black height in one root to leaf path.

Deletion is fairly complex process. To understand deletion, notion of double black is used. When a black node is deleted and replaced by a black child, the child is marked as ***double black***. The main task now becomes to convert this double black to single black.

Deletion Steps
Following are detailed steps for deletion.

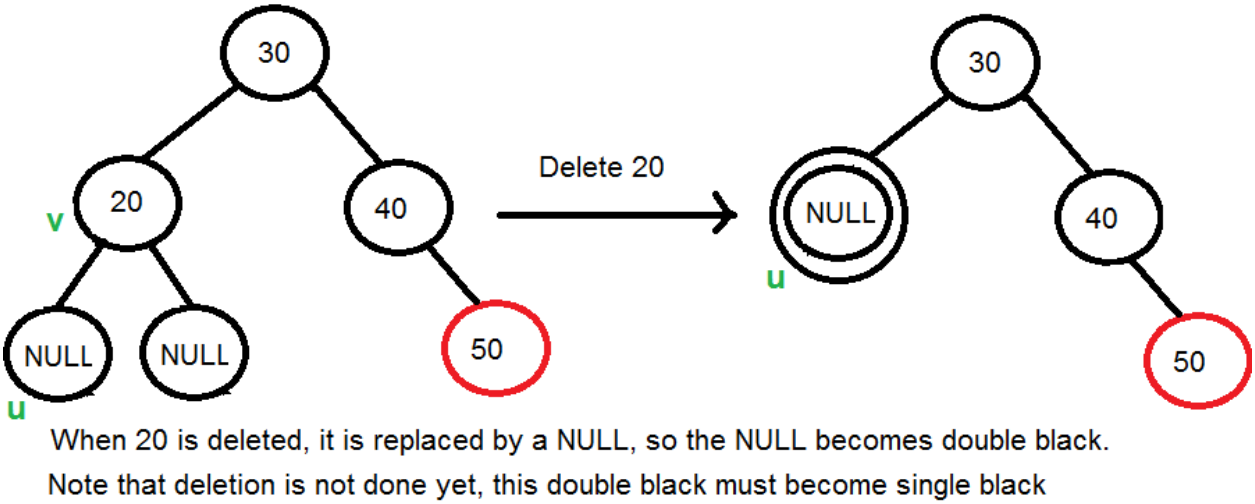
1) Perform standard BST delete. When we perform standard delete operation in BST, we always end up deleting a node which is either leaf or has only one child (For an internal node, we copy the successor and then recursively call delete for successor, successor is always a leaf node or a node with one child). So we only need to handle cases where a node is leaf or has one child. Let v be the node to be deleted and u be the child that replaces v (Note that u is NULL when v is a leaf and color of NULL is considered as Black).

2) Simple Case: If either u or v is red, we mark the replaced child as black (No change in black height). Note that both u and v cannot be red as v is parent of u and two consecutive reds are not allowed in red-black tree.



3) If Both u and v are Black.

3.1) Color u as double black. Now our task reduces to convert this double black to single black. Note that If v is leaf, then u is NULL and color of NULL is considered as black. So the deletion of a black leaf also causes a double black.

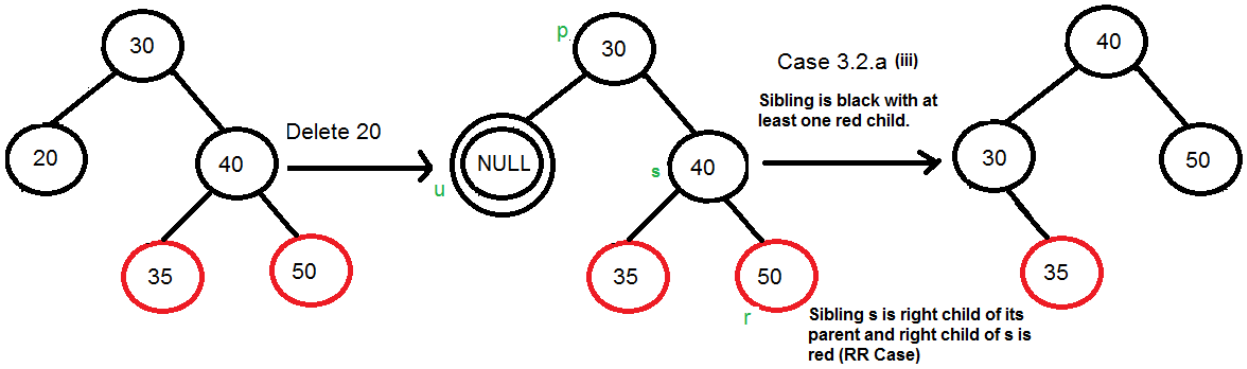


3.2) Do following while the current node u is double black or it is not root. Let sibling of node be **s**.
….**(a): If sibling s is black and at least one of sibling's children is red**, perform rotation(s). Let the red child of s be **r**. This case can be divided in four subcases depending upon positions of s and r.

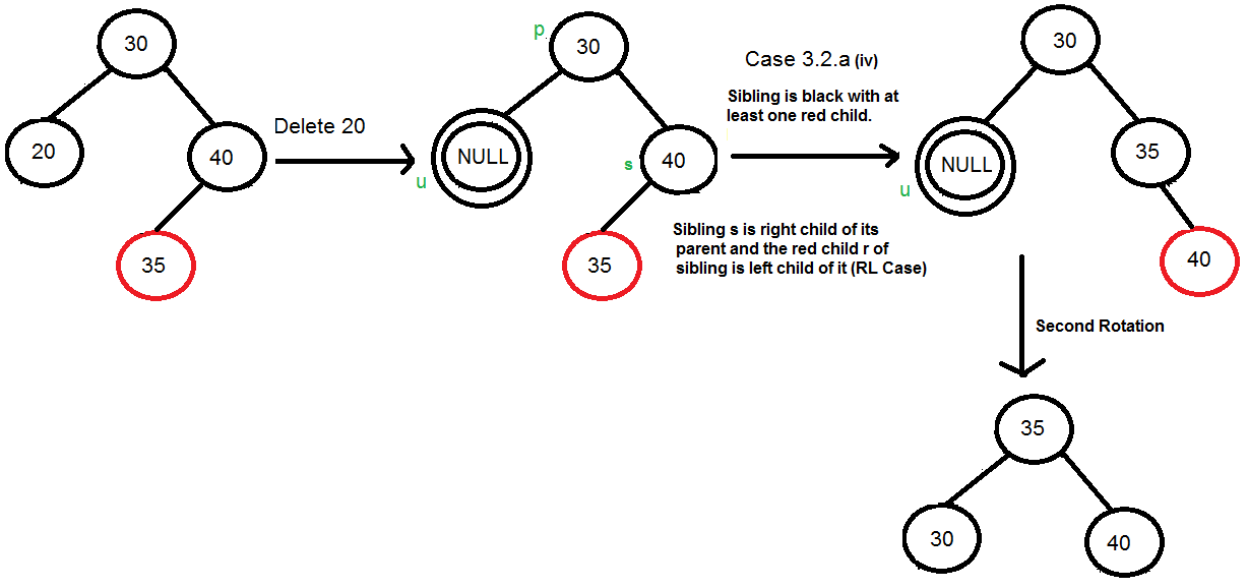
……..**(i)** Left Left Case (s is left child of its parent and r is left child of s or both children of s are red). This is mirror of right right case shown in below diagram.

……..**(ii)** Left Right Case (s is left child of its parent and r is right child). This is mirror of right left case shown in below diagram.

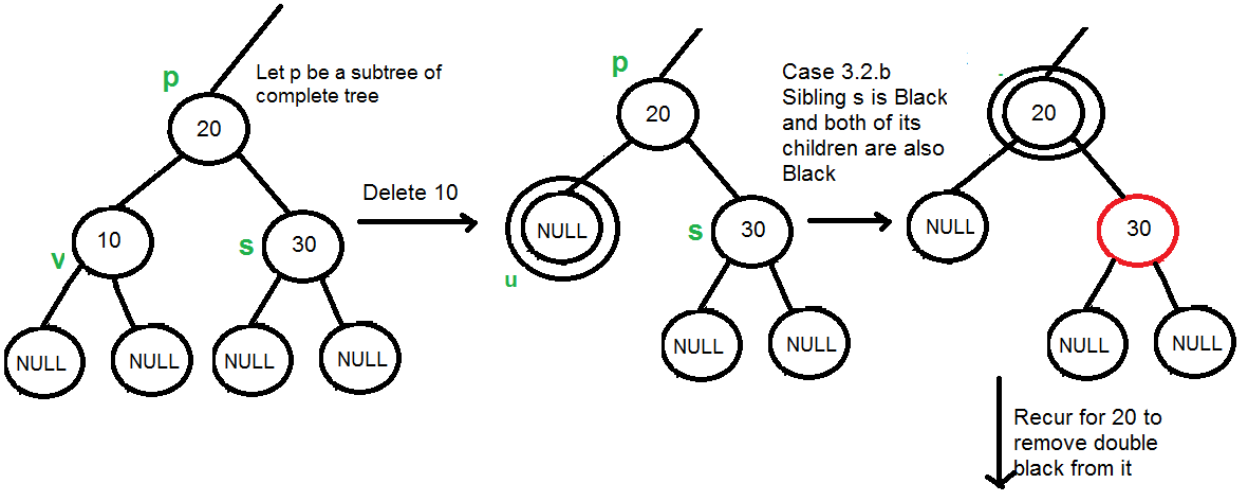
……..**(iii)** Right Right Case (s is right child of its parent and r is right child of s or both children of s are red)



……..**(iv)** Right Left Case (s is right child of its parent and r is left child of s)

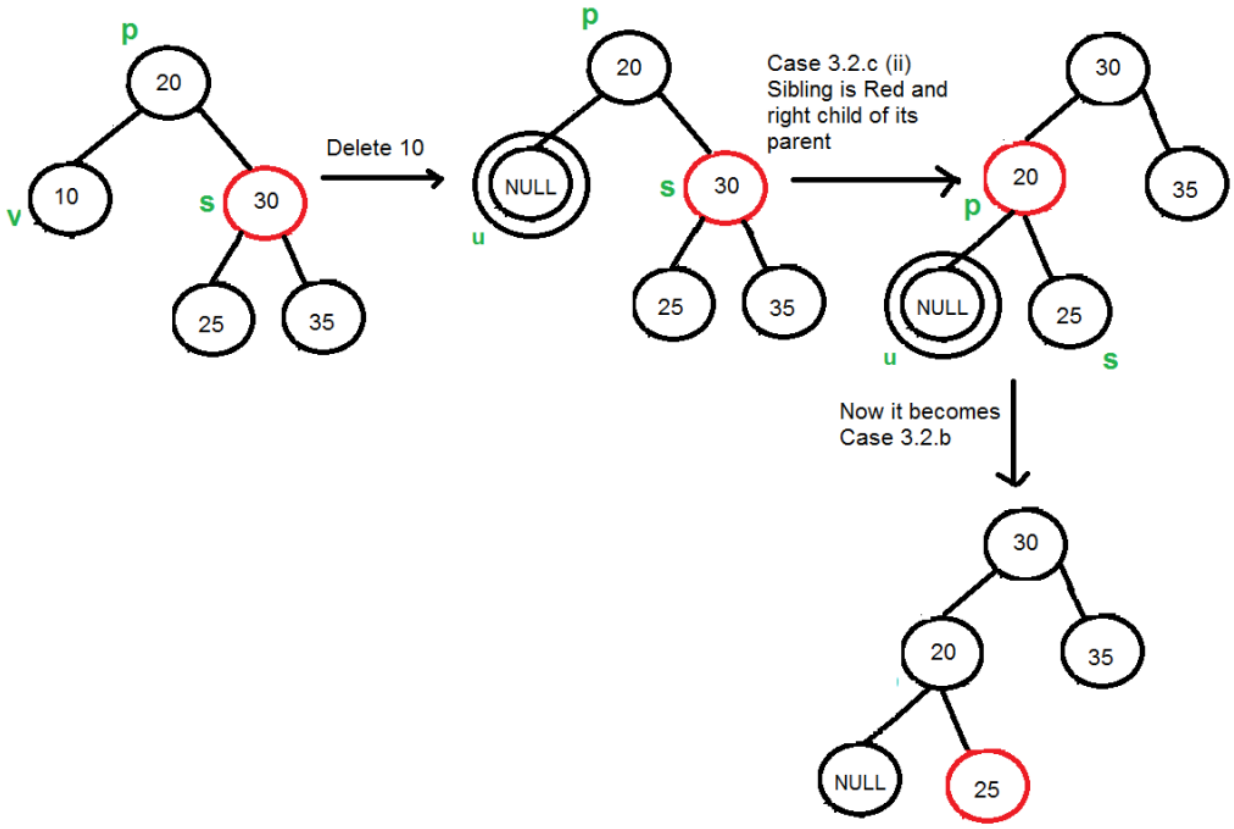


…..**(b): If sibling is black and its both children are black**, perform recoloring, and recur for the parent if parent is black.



In this case, if parent was red, then we didn't need to recur for prent, we can simply make it black (red + double black = single black)

…..**(c): If sibling is red**, perform a rotation to move old sibling up, recolor the old sibling and parent. The new sibling is always black (See the below diagram). This mainly converts the tree to black sibling case (by rotation) and leads to case (a) or (b). This case can be divided in two subcases.
……..**(i)** Left Case (s is left child of its parent). This is mirror of right right case shown in below diagram. We right rotate the parent p.
……..**(iii)** Right Case (s is right child of its parent). We left rotate the parent p.



3.3) If u is root, make it single black and return (Black height of complete tree reduces by 1).

References:
https://www.cs.purdue.edu/homes/ayg/CS251/slides/chap13c.pdf

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/red-black-tree-set-3-delete-2/

Chapter 41

K Dimensional Tree | Set 1 (Search and Insert)

A K-D Tree(also called as K-Dimensional Tree) is a binary search tree where data in each node is a K-Dimensional point in space. In short, it is a space partitioning(details below) data structure for organizing points in a K-Dimensional space.

A non-leaf node in K-D tree divides the space into two parts, called as half-spaces.

Points to the left of this space are represented by the left subtree of that node and points to the right of the space are represented by the right subtree. We will soon be explaining the concept on how the space is divided and tree is formed.

For the sake of simplicity, let us understand a 2-D Tree with an example.

The root would have an x-aligned plane, the root's children would both have y-aligned planes, the root's grandchildren would all have x-aligned planes, and the root's great-grandchildren would all have y-aligned planes and so on.

Generalization:
Let us number the planes as 0, 1, 2, …(K – 1). From the above example, it is quite clear that a point (node) at depth D will have A aligned plane where A is calculated as:

A = D mod K

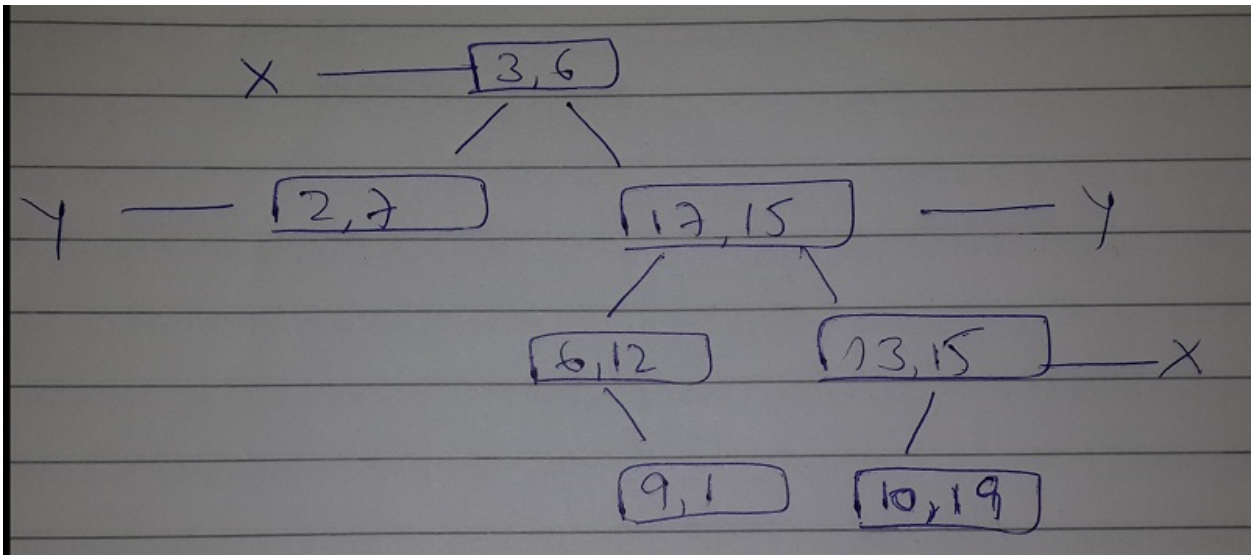
How to determine if a point will lie in the left subtree or in right subtree?

If the root node is aligned in planeA, then the left subtree will contain all points whose coordinates in that plane are smaller than that of root node. Similarly, the right subtree will contain all points whose coordinates in that plane are greater-equal to that of root node.

Creation of a 2-D Tree:
Consider following points in a 2-D plane:
(3, 6), (17, 15), (13, 15), (6, 12), (9, 1), (2, 7), (10, 19)

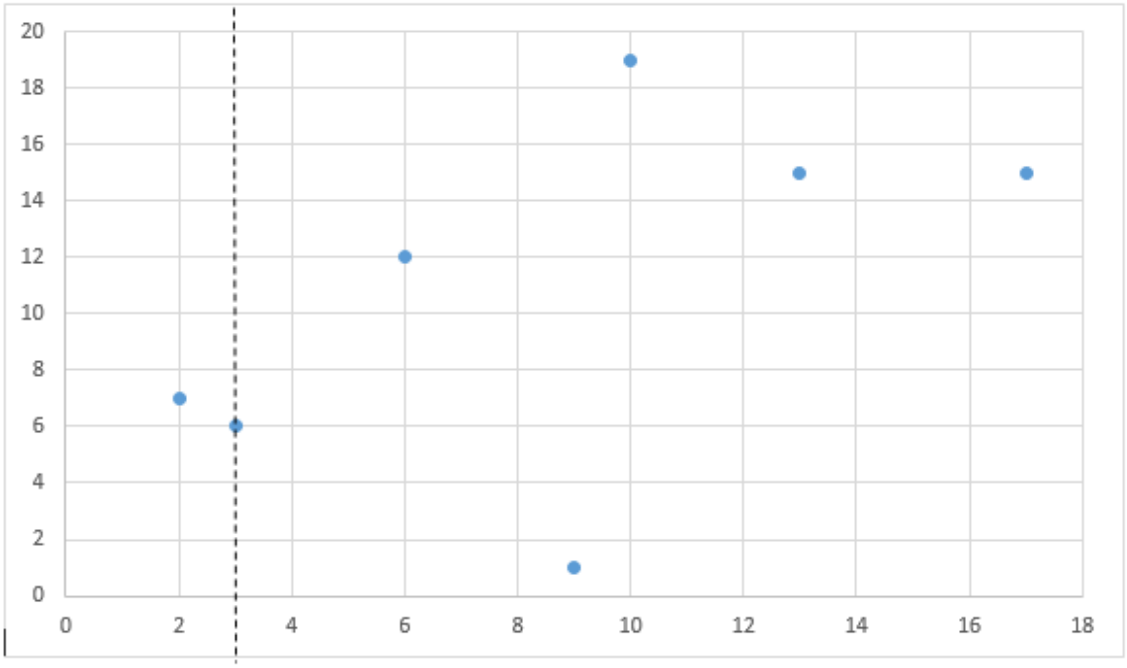
1. Insert (3, 6): Since tree is empty, make it the root node.
2. Insert (17, 15): Compare it with root node point. Since root node is X-aligned, the X-coordinate value will be compared to determine if it lies in the rightsubtree or in the right subtree. This point will be Y-aligned.
3. Insert (13, 15): X-value of this point is greater than X-value of point in root node. So, this will lie in the right subtree of (3, 6). Again Compare Y-value of this point with the Y-value of point (17, 15) (Why?). Since, they are equal, this point will lie in the right subtree of (17, 15). This point will be X-aligned.

4. Insert (6, 12): X-value of this point is greater than X-value of point in root node. So, this will lie in the right subtree of (3, 6). Again Compare Y-value of this point with the Y-value of point (17, 15) (Why?). Since, 12 < 15, this point will lie in the left subtree of (17, 15). This point will be X-aligned.
5. Insert (9, 1):Similarly, this point will lie in the right of (6, 12).
6. Insert (2, 7):Similarly, this point will lie in the left of (3, 6).
7. Insert (10, 19): Similarly, this point will lie in the left of (13, 15).



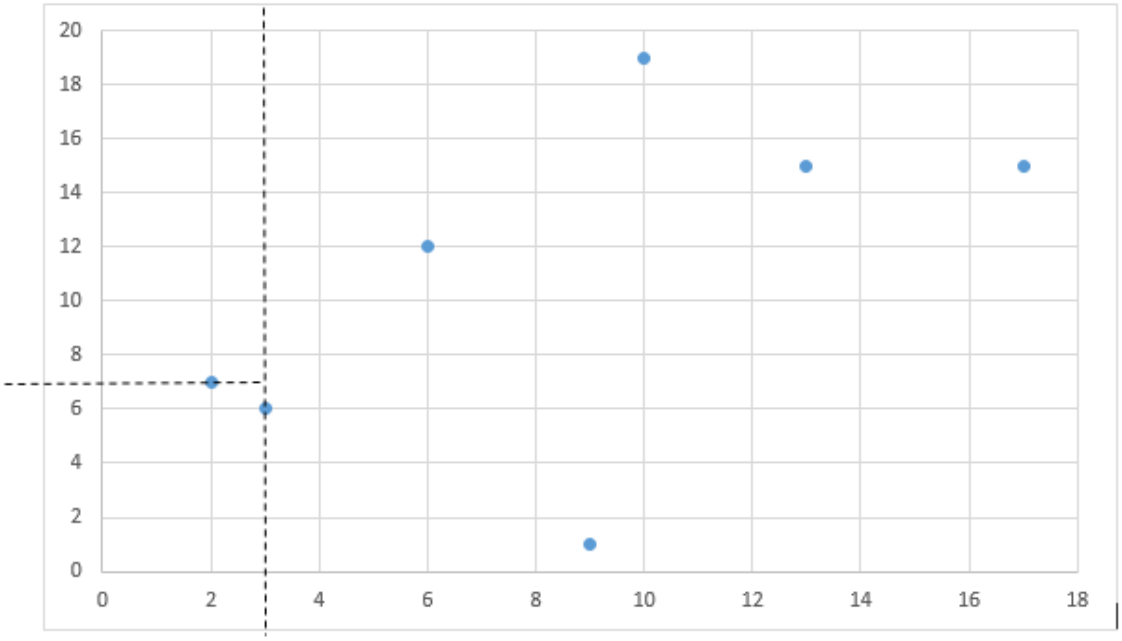
How is space partitioned?
All 7 points will be plotted in the X-Y plane as follows:

1. Point (3, 6) will divide the space into two parts: Draw line X = 3.

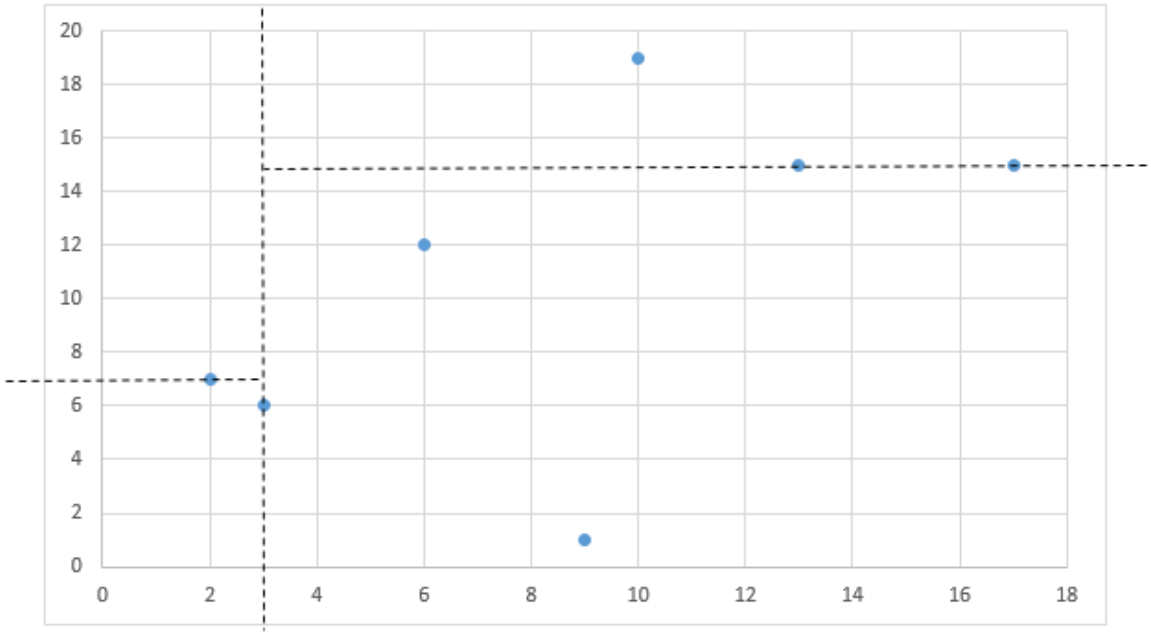


2. Point (2, 7) will divide the space to the left of line X = 3 into two parts horizontally.

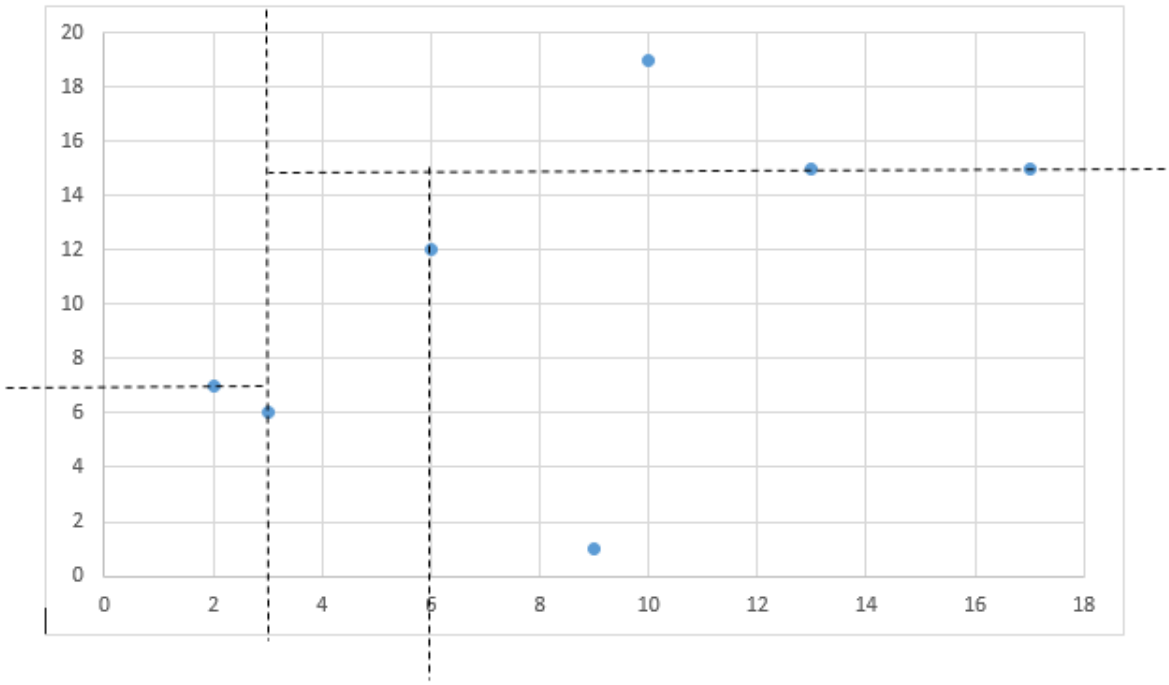
Draw line Y = 7 to the left of line X = 3.



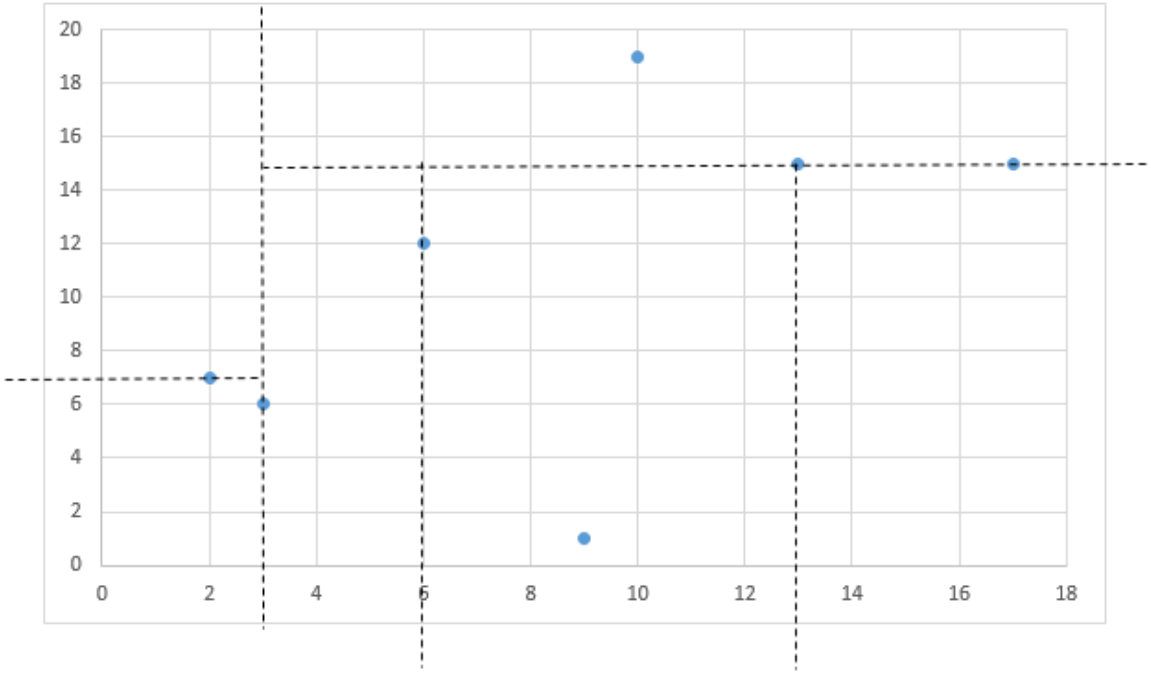
3. Point (17, 15) will divide the space to the right of line X = 3 into two parts horizontally. Draw line Y = 15 to the right of line X = 3.



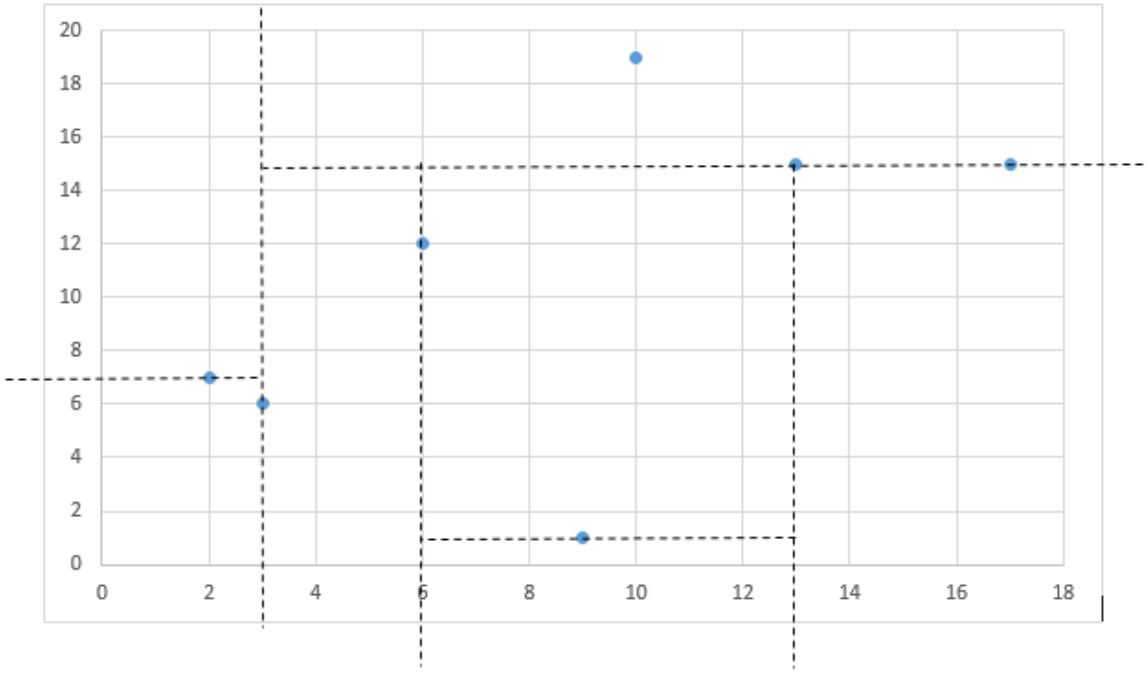
- Point (6, 12) will divide the space below line Y = 15 and to the right of line X = 3 into two parts. Draw line X = 6 to the right of line X = 3 and below line Y = 15.



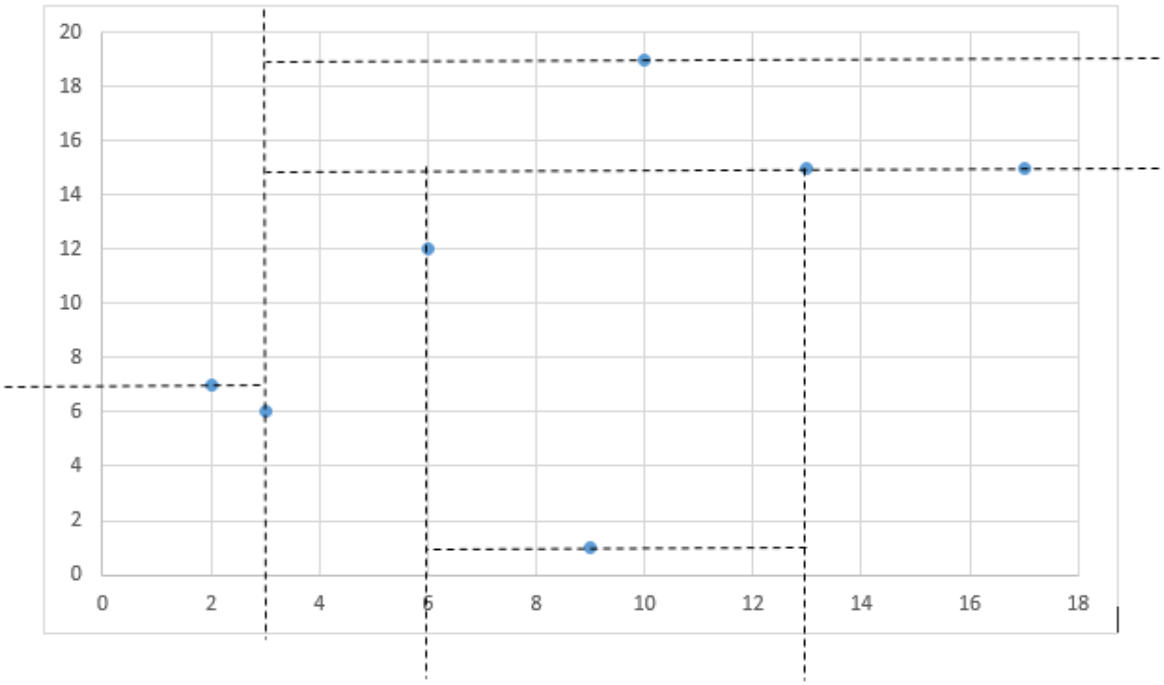
- Point (13, 15) will divide the space below line Y = 15 and to the right of line X = 6 into two parts. Draw line X = 13 to the right of line X = 6 and below line Y = 15.



- Point (9, 1) will divide the space between lines X = 3, X = 6 and Y = 15 into two parts. Draw line Y = 1 between lines X = 3 and X = 6.



- Point (10, 19) will divide the space to the right of line X = 3 and above line Y = 15 into two parts. Draw line Y = 19 to the right of line X = 3 and above line Y = 15.



Following is C++ implementation of KD Tree basic operations like search, insert and delete.

```
// A C++ program to demonstrate operations of KD tree
#include<bits/stdc++.h>
```

```
using namespace std;

const int k = 2;

// A structure to represent node of kd tree
struct Node
{
    int point[k]; // To store k dimensional point
    Node *left, *right;
};

// A method to create a node of K D tree
struct Node* newNode(int arr[])
{
    struct Node* temp = new Node;

    for (int i=0; i<k; i++)
       temp->point[i] = arr[i];

    temp->left = temp->right = NULL;
    return temp;
}

// Inserts a new node and returns root of modified tree
// The parameter depth is used to decide axis of comparison
Node *insertRec(Node *root, int point[], unsigned depth)
{
    // Tree is empty?
    if (root == NULL)
       return newNode(point);

    // Calculate current dimension (cd) of comparison
    unsigned cd = depth % k;

    // Compare the new point with root on current dimension 'cd'
    // and decide the left or right subtree
    if (point[cd] < (root->point[cd]))
        root->left  = insertRec(root->left, point, depth + 1);
    else
        root->right = insertRec(root->right, point, depth + 1);

    return root;
}

// Function to insert a new point with given point in
// KD Tree and return new root. It mainly uses above recursive
// function "insertRec()"
Node* insert(Node *root, int point[])
{
    return insertRec(root, point, 0);
}

// A utility method to determine if two Points are same
// in K Dimensional space
```

```
bool arePointsSame(int point1[], int point2[])
{
    // Compare individual pointinate values
    for (int i = 0; i < k; ++i)
        if (point1[i] != point2[i])
            return false;

    return true;
}

// Searches a Point represented by "point[]" in the K D tree.
// The parameter depth is used to determine current axis.
bool searchRec(Node* root, int point[], unsigned depth)
{
    // Base cases
    if (root == NULL)
        return false;
    if (arePointsSame(root->point, point))
        return true;

    // Current dimension is computed using current depth and total
    // dimensions (k)
    unsigned cd = depth % k;

    // Compare point with root with respect to cd (Current dimension)
    if (point[cd] < root->point[cd])
        return searchRec(root->left, point, depth + 1);

    return searchRec(root->right, point, depth + 1);
}

// Searches a Point in the K D tree. It mainly uses
// searchRec()
bool search(Node* root, int point[])
{
    // Pass current depth as 0
    return searchRec(root, point, 0);
}

// Driver program to test above functions
int main()
{
    struct Node *root = NULL;
    int points[][k] = {{3, 6}, {17, 15}, {13, 15}, {6, 12},
                       {9, 1}, {2, 7}, {10, 19}};

    int n = sizeof(points)/sizeof(points[0]);

    for (int i=0; i<n; i++)
       root = insert(root, points[i]);

    int point1[] = {10, 19};
    (search(root, point1))? cout << "Found\n": cout << "Not Found\n";
```

```
    int point2[] = {12, 19};
    (search(root, point2))? cout << "Found\n": cout << "Not Found\n";

    return 0;
}
```

Output:

```
 Found
Not Found
```

This article is compiled by **Aashish Barnwal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/k-dimensional-tree/

Category: Trees Tags: Advance Data Structures, Advanced Data Structures

Post navigation

← Amazon interview Experience | Set 141 (For SDE1) Ukkonen's Suffix Tree Construction – Part 4 →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 42

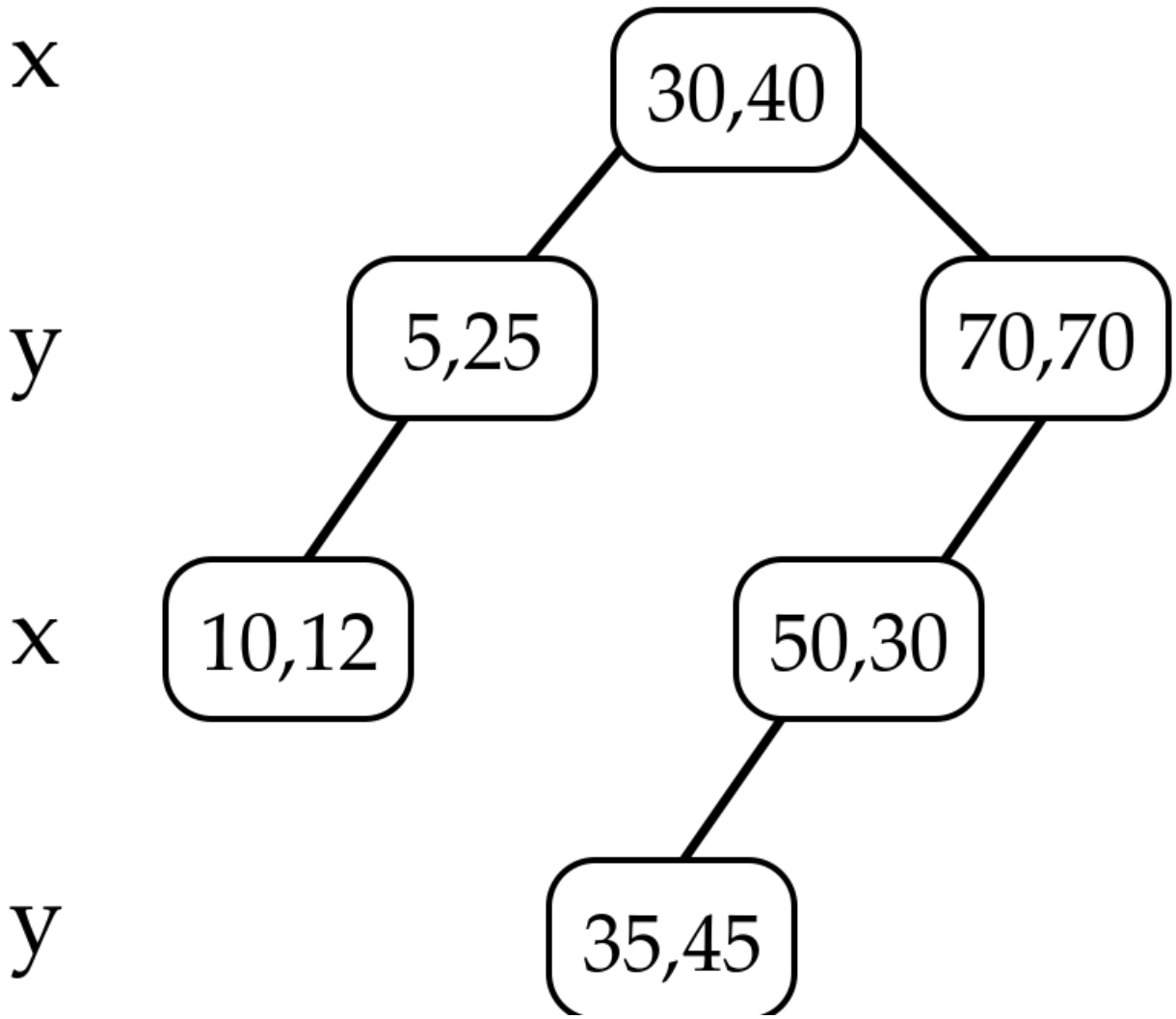
K Dimensional Tree | Set 2 (Find Minimum)

We strongly recommend to refer below post as a prerequisite of this.

K Dimensional Tree | Set 1 (Search and Insert)

In this post find minimum is discussed. The operation is to find minimum in the given dimension. This is especially needed in delete operation.

For example, consider below KD Tree, if given dimension is x, then output should be 5 and if given dimensions is t, then output should be 12. Below image is taken from this source.



In KD tree, points are divided dimension by dimension. For example, root divides keys by dimension 0, level next to root divides by dimension 1, next level by dimension 2 if k is more then 2 (else by dimension 0), and so on.

To find minimum we traverse nodes starting from root. ***If dimension of current level is same as given dimension, then required minimum lies on left side if there is left child***. This is same as Binary Search Tree Minimum.
Above is simple, what to do when current level's dimension is different. ***When dimension of current level is different, minimum may be either in left subtree or right subtree or current node may also be minimum***. So we take minimum of three and return. This is different from Binary Search tree.

Below is C++ implementation of find minimum operation.

```
// A C++ program to demonstrate find minimum on KD tree
#include<bits/stdc++.h>
using namespace std;

const int k = 2;

// A structure to represent node of kd tree
```

```
struct Node
{
    int point[k]; // To store k dimensional point
    Node *left, *right;
};

// A method to create a node of K D tree
struct Node* newNode(int arr[])
{
    struct Node* temp = new Node;

    for (int i=0; i<k; i++)
        temp->point[i] = arr[i];

    temp->left = temp->right = NULL;
    return temp;
}

// Inserts a new node and returns root of modified tree
// The parameter depth is used to decide axis of comparison
Node *insertRec(Node *root, int point[], unsigned depth)
{
    // Tree is empty?
    if (root == NULL)
        return newNode(point);

    // Calculate current dimension (cd) of comparison
    unsigned cd = depth % k;

    // Compare the new point with root on current dimension 'cd'
    // and decide the left or right subtree
    if (point[cd] < (root->point[cd]))
        root->left = insertRec(root->left, point, depth + 1);
    else
        root->right = insertRec(root->right, point, depth + 1);

    return root;
}

// Function to insert a new point with given point in
// KD Tree and return new root. It mainly uses above recursive
// function "insertRec()"
Node* insert(Node *root, int point[])
{
    return insertRec(root, point, 0);
}

// A utility function to find minimum of three integers
int min(int x, int y, int z)
{
    return min(x, min(y, z));
}

// Recursively finds minimum of d'th dimension in KD tree
```

```
// The parameter depth is used to determine current axis.
int findMinRec(Node* root, int d, unsigned depth)
{
    // Base cases
    if (root == NULL)
        return INT_MAX;

    // Current dimension is computed using current depth and total
    // dimensions (k)
    unsigned cd = depth % k;

    // Compare point with root with respect to cd (Current dimension)
    if (cd == d)
    {
        if (root->left == NULL)
            return root->point[d];
        return findMinRec(root->left, d, depth+1);
    }

    // If current dimension is different then minimum can be anywhere
    // in this subtree
    return min(root->point[d],
               findMinRec(root->left, d, depth+1),
               findMinRec(root->right, d, depth+1));
}

// A wrapper over findMinRec(). Returns minimum of d'th dimension
int findMin(Node* root, int d)
{
    // Pass current level or depth as 0
    return findMinRec(root, d, 0);
}

// Driver program to test above functions
int main()
{
    struct Node *root = NULL;
    int points[][k] = {{30, 40}, {5, 25}, {70, 70},
                       {10, 12}, {50, 30}, {35, 45}};

    int n = sizeof(points)/sizeof(points[0]);

    for (int i=0; i<n; i++)
        root = insert(root, points[i]);

    cout << "Minimum of 0'th dimension is " << findMin(root, 0) << endl;
    cout << "Minimum of 1'th dimension is " << findMin(root, 1) << endl;

    return 0;
}
```

Output:

```
 Minimum of 0'th dimension is 5
Minimum of 1'th dimension is 12
```

Source:
https://www.cs.umd.edu/class/spring2008/cmsc420/L19.kd-trees.pdf

This article is contributed by **Ashish Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/k-dimensional-tree-set-2-find-minimum/

Category: Trees Tags: Advanced Data Structures

Post navigation

← Must use JavaScript Array Functions – Part 1 Sum of bit differences among all pairs →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 43

K Dimensional Tree | Set 3 (Delete)

We strongly recommend to refer below posts as a prerequisite of this.

K Dimensional Tree | Set 1 (Search and Insert)

K Dimensional Tree | Set 2 (Find Minimum)

In this post delete is discussed. The operation is to delete a given point from K D Tree.

Like Binary Search Tree Delete, we recursively traverse down and search for the point to be deleted. Below are steps are followed for every node visited.

1) If current node contains the point to be deleted

1. If node to be deleted is a leaf node, simply delete it (Same as BST Delete)
2. If node to be deleted has right child as not NULL (Different from BST)
   (a) Find minimum of current node's dimension in right subtree.
   (b) Replace the node with above found minimum and recursively delete minimum in right subtree.
3. Else If node to be deleted has left child as not NULL (Different from BST)
   (a) Find minimum of current node's dimension in left subtree.
   (b) Replace the node with above found minimum and recursively delete minimum in left subtree.
   (c) Make new left subtree as right child of current node.

2) If current doesn't contain the point to be deleted

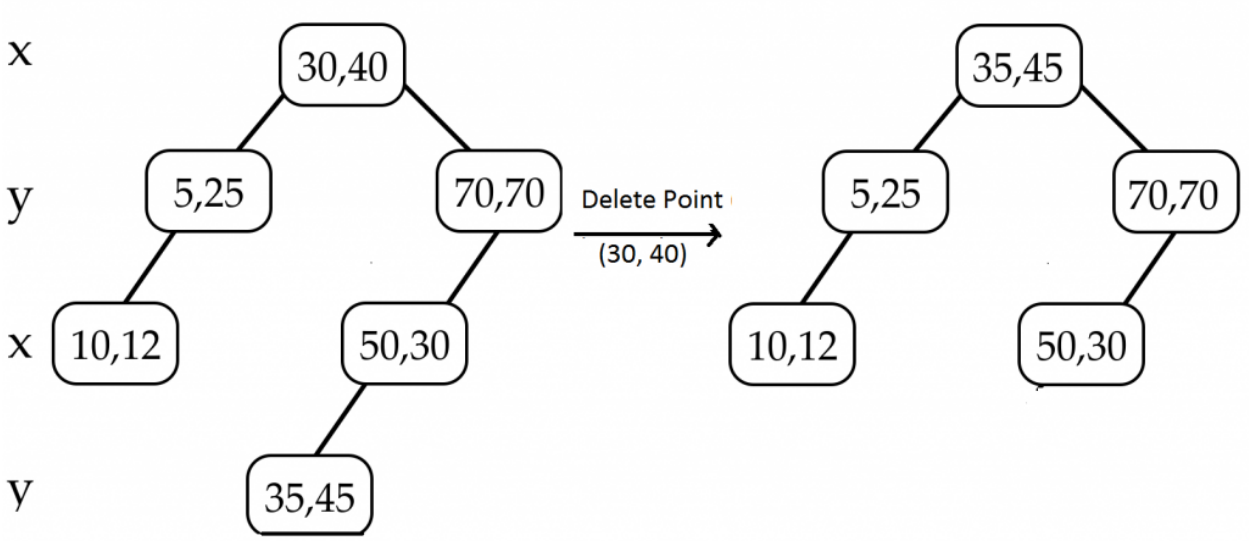
1. If node to be deleted is smaller than current node on current dimension, recur for left subtree.
2. Else recur for right subtree.

Why 1.b and 1.c are different from BST?
In BST delete, if a node's left child is empty and right is not empty, we replace the node with right child. In K D Tree, doing this would violate the KD tree property as dimension of right child of node is different from node's dimension. For example, if node divides point by x axis values. then its children divide by y axis, so we can't simply replace node with right child. Same is true for the case when right child is not empty and left child is empty.

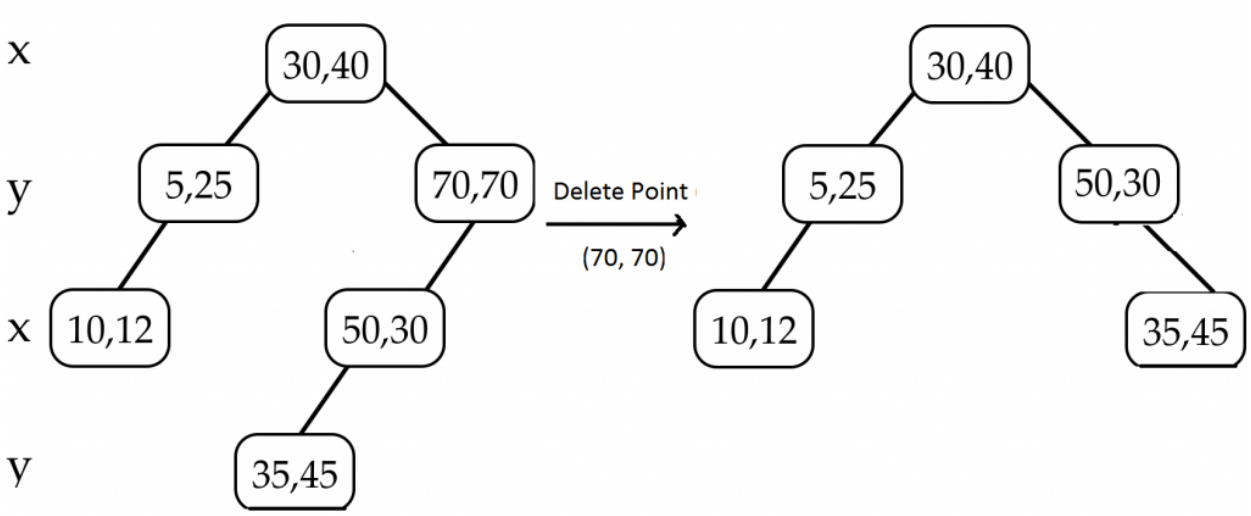
Why 1.c doesn't find max in left subtree and recur for max like 1.b?
Doing this violates the property that all equal values are in right subtree. For example, if we delete (!0, 10) in below subtree and replace if with

```
Wrong Way (Equal key in left subtree after deletion)
            (5, 6)                              (4, 10)
             /              Delete(5, 6)          /
        (4, 10)            ------------>     (4, 20)
             \
          (4, 20)

Right way (Equal key in right subtree after deletion)
             (5, 6)                            (4, 10)
             /              Delete(5, 6)           \
         (4, 10)            ------------>         (4, 20)
              \
             (4, 20)
```

Example of Delete:
Delete (30, 40): Since right child is not NULL and dimension of node is x, we find the node with minimum x value in right child. The node is (35, 45), we replace (30, 40) with (35, 45) and delete (35, 45).



Delete (70, 70): Dimension of node is y. Since right child is NULL, we find the node with minimum y value in left child. The node is (50, 30), we replace (70, 70) with (50, 30) and recursively delete (50, 30) in left subtree. Finally we make the modified left subtree as right subtree of (50, 30).



Below is C++ implementation of K D Tree delete.

```
// A C++ program to demonstrate delete in K D tree
#include<bits/stdc++.h>
using namespace std;

const int k = 2;

// A structure to represent node of kd tree
struct Node
{
    int point[k]; // To store k dimensional point
    Node *left, *right;
};

// A method to create a node of K D tree
struct Node* newNode(int arr[])
{
    struct Node* temp = new Node;

    for (int i=0; i<k; i++)
        temp->point[i] = arr[i];

    temp->left = temp->right = NULL;
    return temp;
}

// Inserts a new node and returns root of modified tree
// The parameter depth is used to decide axis of comparison
Node *insertRec(Node *root, int point[], unsigned depth)
{
    // Tree is empty?
    if (root == NULL)
        return newNode(point);

    // Calculate current dimension (cd) of comparison
```

```
    unsigned cd = depth % k;

    // Compare the new point with root on current dimension 'cd'
    // and decide the left or right subtree
    if (point[cd] < (root->point[cd]))
        root->left = insertRec(root->left, point, depth + 1);
    else
        root->right = insertRec(root->right, point, depth + 1);

    return root;
}

// Function to insert a new point with given point in
// KD Tree and return new root. It mainly uses above recursive
// function "insertRec()"
Node* insert(Node *root, int point[])
{
    return insertRec(root, point, 0);
}

// A utility function to find minimum of three integers
Node *minNode(Node *x, Node *y, Node *z, int d)
{
    Node *res = x;
    if (y != NULL && y->point[d] < res->point[d])
       res = y;
    if (z != NULL && z->point[d] < res->point[d])
       res = z;
    return res;
}

// Recursively finds minimum of d'th dimension in KD tree
// The parameter depth is used to determine current axis.
Node *findMinRec(Node* root, int d, unsigned depth)
{
    // Base cases
    if (root == NULL)
        return NULL;

    // Current dimension is computed using current depth and total
    // dimensions (k)
    unsigned cd = depth % k;

    // Compare point with root with respect to cd (Current dimension)
    if (cd == d)
    {
        if (root->left == NULL)
            return root;
        return findMinRec(root->left, d, depth+1);
    }

    // If current dimension is different then minimum can be anywhere
    // in this subtree
    return minNode(root,
```

```
               findMinRec(root->left, d, depth+1),
               findMinRec(root->right, d, depth+1), d);
}

// A wrapper over findMinRec(). Returns minimum of d'th dimension
Node *findMin(Node* root, int d)
{
    // Pass current level or depth as 0
    return findMinRec(root, d, 0);
}

// A utility method to determine if two Points are same
// in K Dimensional space
bool arePointsSame(int point1[], int point2[])
{
    // Compare individual pointinate values
    for (int i = 0; i < k; ++i)
        if (point1[i] != point2[i])
            return false;

    return true;
}

// Copies point p2 to p1
void copyPoint(int p1[], int p2[])
{
   for (int i=0; i<k; i++)
       p1[i] = p2[i];
}

// Function to delete a given point 'point[]' from tree with root
// as 'root'.  depth is current depth and passed as 0 initially.
// Returns root of the modified tree.
Node *deleteNodeRec(Node *root, int point[], int depth)
{
    // Given point is not present
    if (root == NULL)
        return NULL;

    // Find dimension of current node
    int cd = depth % k;

    // If the point to be deleted is present at root
    if (arePointsSame(root->point, point))
    {
        // 2.b) If right child is not NULL
        if (root->right != NULL)
        {
            // Find minimum of root's dimension in right subtree
            Node *min = findMin(root->right, cd);

            // Copy the minimum to root
            copyPoint(root->point, min->point);
```

```
            // Recursively delete the minimum
            root->right = deleteNodeRec(root->right, min->point, depth+1);
        }
        else if (root->left != NULL) // same as above
        {
            Node *min = findMin(root->left, cd);
            copyPoint(root->point, min->point);
            root->right = deleteNodeRec(root->left, min->point, depth+1);
        }
        else // If node to be deleted is leaf node
        {
            delete root;
            return NULL;
        }
        return root;
    }

    // 2) If current node doesn't contain point, search downward
    if (point[cd] < root->point[cd])
        root->left = deleteNodeRec(root->left, point, depth+1);
    else
        root->right = deleteNodeRec(root->right, point, depth+1);
    return root;
}

// Function to delete a given point from K D Tree with 'root'
 Node* deleteNode(Node *root, int point[])
{
   // Pass depth as 0
   return deleteNodeRec(root, point, 0);
}

// Driver program to test above functions
int main()
{
    struct Node *root = NULL;
    int points[][k] = {{30, 40}, {5, 25}, {70, 70},
                       {10, 12}, {50, 30}, {35, 45}};

    int n = sizeof(points)/sizeof(points[0]);

    for (int i=0; i<n; i++)
        root = insert(root, points[i]);

    // Delet (30, 40);
    root = deleteNode(root, points[0]);

    cout << "Root after deletion of (30, 40)\n";
    cout << root->point[0] << ", " << root->point[1] << endl;

    return 0;
}
```

Output:

```
 Root after deletion of (30, 40)
35, 45
```

Source:
https://www.cs.umd.edu/class/spring2008/cmsc420/L19.kd-trees.pdf

This article is contributed by **Ashish Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/k-dimensional-tree-set-3-delete/

Category: Trees Tags: Advanced Data Structures

Post navigation

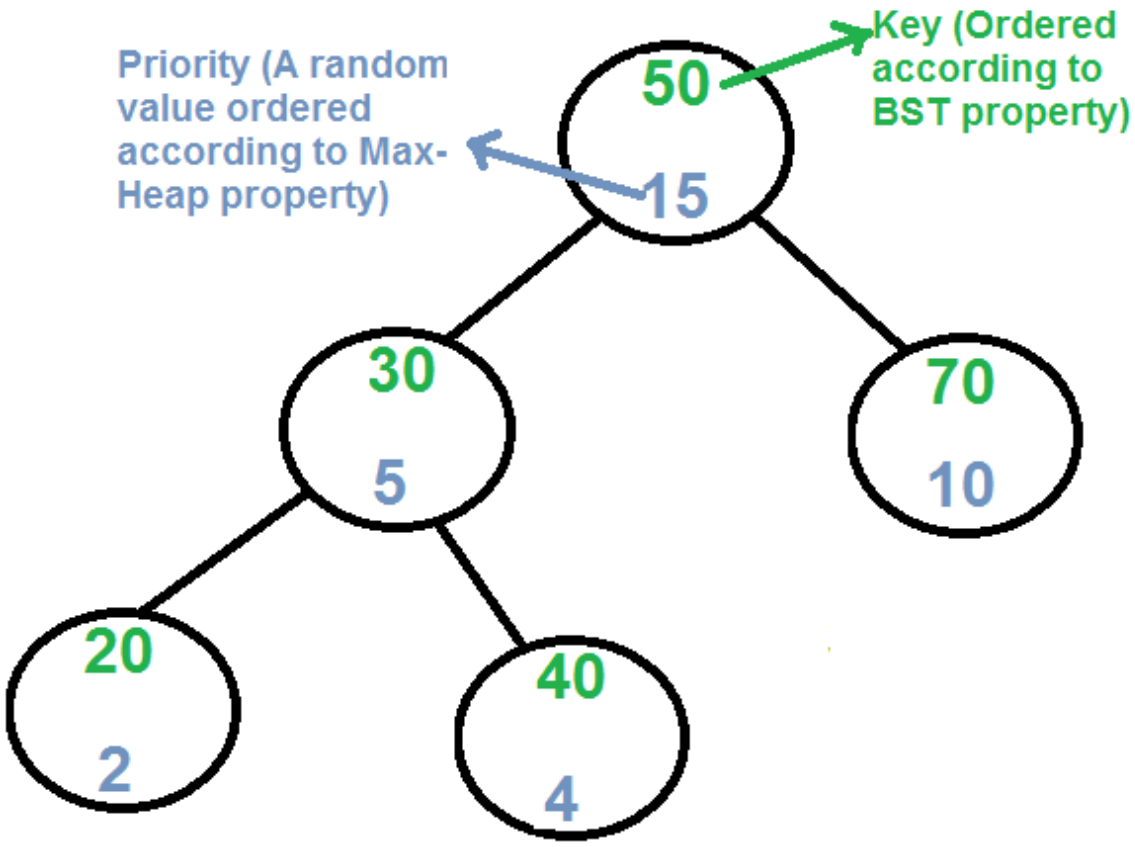
← Sum of bit differences among all pairs SAP Labs Interview Experience | Set 12 (On-Campus) →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 44

Treap (A Randomized Binary Search Tree)

Like Red-Black and AVL Trees, Treap is a Balanced Binary Search Tree, but not guaranteed to have height as O(Log n). The idea is to use Randomization and Binary Heap property to maintain balance with high probability. The expected time complexity of search, insert and delete is O(Log n).



Every node of Treap maintains two values.

1) **Key** Follows standard BST ordering (left is smaller and right is greater)
2) **Priority** Randomly assigned value that follows Max-Heap property.

Basic Operation on Treap:
Like other self-balancing Binary Search Trees, Treap uses rotations to maintain Max-Heap property during insertion and deletion.

```
T1, T2 and T3 are subtrees of the tree rooted with y (on left side)
or x (on right side)
               y                               x
              / \     Right Rotation          /  \
             x   T3   - - - - - - - >        T1   y
            / \
search(x)

Perform standard BST Search to find x.
Insert(x):

1) Create new node with key equals to x and value equals to a random value.

2) Perform standard BST insert.

3) Use rotations to make sure that inserted node's priority follows max heap property.

Delete(x):

1) If node to be deleted is a leaf, delete it.

2) Else replace node's priority with minus infinite ( -INF ), and do appropriate rotations to bring the node d

Refer Implementation of Treap Search, Insert and Delete for more details.


References:

https://en.wikipedia.org/wiki/Treap

https://courses.cs.washington.edu/courses/cse326/00wi/handouts/lecture19/sld017.htm
Please write comments if you find anything incorrect, or you want to share more information about the topic di
```

Source

http://www.geeksforgeeks.org/treap-a-randomized-binary-search-tree/

Category: Trees Tags: Advanced Data Structures

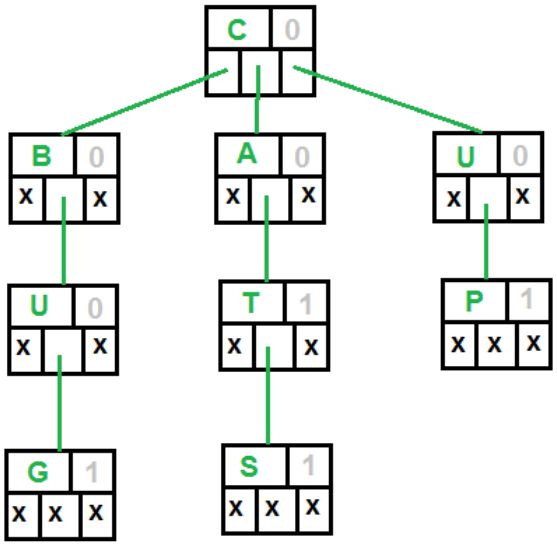
Chapter 45

Ternary Search Tree

A ternary search tree is a special trie data structure where the child nodes of a standard trie are ordered as a binary search tree.

Representation of ternary search trees:
Unlike trie(standard) data structure where each node contains 26 pointers for its children, each node in a ternary search tree contains only 3 pointers:
1. The left pointer points to the node whose value is less than the value in the current node.
2. The equal pointer points to the node whose value is equal to the value in the current node.
3. The right pointer points to the node whose value is greater than the value in the current node.

Apart from above three pointers, each node has a field to indicate data(character in case of dictionary) and another field to mark end of a string.
So, more or less it is similar to BST which stores data based on some order. However, data in a ternary search tree is distributed over the nodes. e.g. It needs 4 nodes to store the word "Geek".
Below figure shows how exactly the words in a ternary search tree are stored?



Ternary Search Tree for CAT, BUG, CATS, UP

One of the advantage of using ternary search trees over tries is that ternary search trees are a more space

efficient (involve only three pointers per node as compared to 26 in standard tries). Further, ternary search trees can be used any time a hashtable would be used to store strings.

Tries are suitable when there is a proper distribution of words over the alphabets so that spaces are utilized most efficiently. Otherwise ternary search trees are better. Ternary search trees are efficient to use(in terms of space) when the strings to be stored share a common prefix.

Applications of ternary search trees:
1. Ternary search trees are efficient for queries like "Given a word, find the next word in dictionary(near-neighbor lookups)" or "Find all telephone numbers starting with 9342 or "typing few starting characters in a web browser displays all website names with this prefix"(Auto complete feature)".

2. Used in spell checks: Ternary search trees can be used as a dictionary to store all the words. Once the word is typed in an editor, the word can be parallely searched in the ternary search tree to check for correct spelling.

Implementation:
Following is C implementation of ternary search tree. The operations implemented are, search, insert and traversal.

```
// C program to demonstrate Ternary Search Tree (TST) insert, travese
// and search operations
#include <stdio.h>
#include <stdlib.h>
#define MAX 50

// A node of ternary search tree
struct Node
{
    char data;

    // True if this character is last character of one of the words
    unsigned isEndOfString: 1;

    struct Node *left, *eq, *right;
};

// A utility function to create a new ternary search tree node
struct Node* newNode(char data)
{
    struct Node* temp = (struct Node*) malloc(sizeof( struct Node ));
    temp->data = data;
    temp->isEndOfString = 0;
    temp->left = temp->eq = temp->right = NULL;
    return temp;
}

// Function to insert a new word in a Ternary Search Tree
void insert(struct Node** root, char *word)
{
    // Base Case: Tree is empty
    if (!(*root))
        *root = newNode(*word);

    // If current character of word is smaller than root's character,
```

```
    // then insert this word in left subtree of root
    if ((*word) < (*root)->data)
        insert(&( (*root)->left ), word);

    // If current character of word is greate than root's character,
    // then insert this word in right subtree of root
    else if ((*word) > (*root)->data)
        insert(&( (*root)->right ), word);

    // If current character of word is same as root's character,
    else
    {
        if (*(word+1))
            insert(&( (*root)->eq ), word+1);

        // the last character of the word
        else
            (*root)->isEndOfString = 1;
    }
}

// A recursive function to traverse Ternary Search Tree
void traverseTSTUtil(struct Node* root, char* buffer, int depth)
{
    if (root)
    {
        // First traverse the left subtree
        traverseTSTUtil(root->left, buffer, depth);

        // Store the character of this node
        buffer[depth] = root->data;
        if (root->isEndOfString)
        {
            buffer[depth+1] = '\0';
            printf( "%s\n", buffer);
        }

        // Traverse the subtree using equal pointer (middle subtree)
        traverseTSTUtil(root->eq, buffer, depth + 1);

        // Finally Traverse the right subtree
        traverseTSTUtil(root->right, buffer, depth);
    }
}

// The main function to traverse a Ternary Search Tree.
// It mainly uses traverseTSTUtil()
void traverseTST(struct Node* root)
{
    char buffer[MAX];
    traverseTSTUtil(root, buffer, 0);
}

// Function to search a given word in TST
```

```
int searchTST(struct Node *root, char *word)
{
    if (!root)
        return 0;

    if (*word < (root)->data)
        return searchTST(root->left, word);

    else if (*word > (root)->data)
        return searchTST(root->right, word);

    else
    {
        if (*(word+1) == '\0')
            return root->isEndOfString;

        return searchTST(root->eq, word+1);
    }
}

// Driver program to test above functions
int main()
{
    struct Node *root = NULL;

    insert(&root, "cat");
    insert(&root, "cats");
    insert(&root, "up");
    insert(&root, "bug");

    printf("Following is traversal of ternary search tree\n");
    traverseTST(root);

    printf("\nFollowing are search results for cats, bu and cat respectively\n");
    searchTST(root, "cats")? printf("Found\n"): printf("Not Found\n");
    searchTST(root, "bu")?   printf("Found\n"): printf("Not Found\n");
    searchTST(root, "cat")?  printf("Found\n"): printf("Not Found\n");

    return 0;
}
```

Output:

```
 Following is traversal of ternary search tree
bug
cat
cats
up

Following are search results for cats, bu and cat respectively
Found
Not Found
Found
```

Time Complexity: The time complexity of the ternary search tree operations is similar to that of binary search tree. i.e. the insertion, deletion and search operations take time proportional to the height of the ternary search tree. The space is proportional to the length of the string to be stored.

Reference:
http://en.wikipedia.org/wiki/Ternary_search_tree

This article is compiled by **Aashish Barnwal**and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/ternary-search-tree/

Category: Trees Tags: Advance Data Structures, Advanced Data Structures

Post navigation

← [TopTalent.in] Exclusive Interview with Ravi Kiran from BITS, Pilani who got placed in Google, Microsoft and Facebook Amazon Interview | Set 17 →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 46

Interval Tree

Consider a situation where we have a set of intervals and we need following operations to be implemented efficiently.
1) Add an interval
2) Remove an interval
3) Given an interval x, find if x overlaps with any of the existing intervals.

Interval Tree: The idea is to augment a self-balancing Binary Search Tree (BST) like Red Black Tree, AVL Tree, etc to maintain set of intervals so that all operations can be done in O(Logn) time.

Every node of Interval Tree stores following information.
a) **i**: An interval which is represented as a pair *[low, high]*
b) **max**: Maximum *high* value in subtree rooted with this node.

The low value of an interval is used as key to maintain order in BST. The insert and delete operations are same as insert and delete in self-balancing BST used.

The main operation is to search for an overlapping interval. Following is algorithm for searching an overlapping interval x in an Interval tree rooted with *root.*

```
Interval overlappingIntervalSearch(root, x)
1) If x overlaps with root's interval, return the root's interval.

2) If left child of root is not empty and the max  in left child
is greater than x's low value, recur for left child

3) Else recur for right child.
```

How does the above algorithm work?
Let the interval to be searched be x. We need to prove this in for following two cases.

Case 1: *When we go to right subtree, one of the following must be true.*
a) There is an overlap in right subtree: This is fine as we need to return one overlapping interval.
b) There is no overlap in either subtree: We go to right subtree only when either left is NULL or maximum value in left is smaller than *x.low*. So the interval cannot be present in left subtree.

Case 2: *When we go to left subtree, one of the following must be true.*
a) There is an overlap in left subtree: This is fine as we need to return one overlapping interval.
b) There is no overlap in either subtree: This is the most important part. We need to consider following

facts.
… We went to left subtree because *x.low in left subtree*
…. *max in left subtree is a high of one of the intervals let us say* [a, max] *in left subtree.*
…. *Since* x *doesn't overlap with any node in left subtree* x.low *must be smaller than* 'a'.
…. *All nodes in BST are ordered by low value, so all nodes in right subtree must have low value greater than* 'a'.
…. *From above two facts, we can say all intervals in right subtree have low value greater than* x.low. *So* x *cannot overlap with any interval in right subtree.*

Implementation of Interval Tree:
Following is C++ implementation of Interval Tree. The implementation uses basic insert operation of BST to keep things simple. Ideally it should be insertion of AVL Tree or insertion of Red-Black Tree. Deletion from BST is left as an exercise.

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

// Structure to represent an interval
struct Interval
{
    int low, high;
};

// Structure to represent a node in Interval Search Tree
struct ITNode
{
    Interval *i;  // 'i' could also be a normal variable
    int max;
    ITNode *left, *right;
};

// A utility function to create a new Interval Search Tree Node
ITNode * newNode(Interval i)
{
    ITNode *temp = new ITNode;
    temp->i = new Interval(i);
    temp->max = i.high;
    temp->left = temp->right = NULL;
};

// A utility function to insert a new Interval Search Tree Node
// This is similar to BST Insert.  Here the low value of interval
// is used tomaintain BST property
ITNode *insert(ITNode *root, Interval i)
{
    // Base case: Tree is empty, new node becomes root
    if (root == NULL)
        return newNode(i);

    // Get low value of interval at root
    int l = root->i->low;

    // If root's low value is smaller, then new interval goes to
    // left subtree
```

```
    if (i.low < l)
        root->left = insert(root->left, i);

    // Else, new node goes to right subtree.
    else
        root->right = insert(root->right, i);

    // Update the max value of this ancestor if needed
    if (root->max < i.high)
        root->max = i.high;

    return root;
}

// A utility function to check if given two intervals overlap
bool doOVerlap(Interval i1, Interval i2)
{
    if (i1.low <= i2.high && i2.low <= i1.high)
        return true;
    return false;
}

// The main function that searches a given interval i in a given
// Interval Tree.
Interval *overlapSearch(ITNode *root, Interval i)
{
    // Base Case, tree is empty
    if (root == NULL) return NULL;

    // If given interval overlaps with root
    if (doOVerlap(*(root->i), i))
        return root->i;

    // If left child of root is present and max of left child is
    // greater than or equal to given interval, then i may
    // overlap with an interval is left subtree
    if (root->left != NULL && root->left->max >= i.low)
        return overlapSearch(root->left, i);

    // Else interval can only overlap with right subtree
    return overlapSearch(root->right, i);
}

void inorder(ITNode *root)
{
    if (root == NULL) return;

    inorder(root->left);

    cout << "[" << root->i->low << ", " << root->i->high << "]"
         << " max = " << root->max << endl;

    inorder(root->right);
}
```

```
// Driver program to test above functions
int main()
{
    // Let us create interval tree shown in above figure
    Interval ints[] = {{15, 20}, {10, 30}, {17, 19},
        {5, 20}, {12, 15}, {30, 40}
    };
    int n = sizeof(ints)/sizeof(ints[0]);
    ITNode *root = NULL;
    for (int i = 0; i < n; i++)
        root = insert(root, ints[i]);

    cout << "Inorder traversal of constructed Interval Tree is\n";
    inorder(root);

    Interval x = {6, 7};

    cout << "\nSearching for interval [" << x.low << "," << x.high << "]";
    Interval *res = overlapSearch(root, x);
    if (res == NULL)
        cout << "\nNo Overlapping Interval";
    else
        cout << "\nOverlaps with [" << res->low << ", " << res->high << "]";
    return 0;
}
```

Output:

```
 Inorder traversal of constructed Interval Tree is
[5, 20] max = 20
[10, 30] max = 30
[12, 15] max = 15
[15, 20] max = 40
[17, 19] max = 40
[30, 40] max = 40

Searching for interval [6,7]
Overlaps with [5, 20]
```

Applications of Interval Tree:
Interval tree is mainly a geometric data structure and often used for windowing queries, for instance, to find all roads on a computerized map inside a rectangular viewport, or to find all visible elements inside a three-dimensional scene (Source Wiki).

Interval Tree vs Segment Tree
Both segment and interval trees store intervals. Segment tree is mainly optimized for queries for a given point, and interval trees are mainly optimized for overlapping queries for a given interval.

Exercise:
1) Implement delete operation for interval tree.
2) Extend the intervalSearch() to print all overlapping intervals instead of just one.

http://en.wikipedia.org/wiki/Interval_tree
http://www.cse.unr.edu/~mgunes/cs302/IntervalTrees.pptx

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest
https://www.youtube.com/watch?v=dQF0zyaym8A

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/interval-tree/

Category: Trees Tags: Advance Data Structures, Advanced Data Structures

Post navigation

← Adobe Interview | Set 8 (Off-Campus) Print a Binary Tree in Vertical Order | Set 1 →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 47

Implement LRU Cache

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
```

This article is compiled by Aashish Barnwaland reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/implement-lru-cache/

Category: Linked Lists Tags: Advance Data Structures, Advanced Data Structures, Queue

Post navigation

← Microsoft Interview | Set 7 Median of two sorted arrays of different sizes →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 48

Sort numbers stored on different machines

Given N machines. Each machine contains some numbers in sorted form. But the amount of numbers, each machine has is not fixed. Output the numbers from all the machine in sorted non-decreasing form.

```
Example:
       Machine M1 contains 3 numbers: {30, 40, 50}
       Machine M2 contains 2 numbers: {35, 45}
       Machine M3 contains 5 numbers: {10, 60, 70, 80, 100}

       Output: {10, 30, 35, 40, 45, 50, 60, 70, 80, 100}
```

Representation of stream of numbers on each machine is considered as linked list. A Min Heap can be used to print all numbers in sorted order.

Following is the detailed process

1. Store the head pointers of the linked lists in a minHeap of size N where N is number of machines.

2. Extract the minimum item from the minHeap. Update the minHeap by replacing the head of the minHeap with the next number from the linked list or by replacing the head of the minHeap with the last number in the minHeap followed by decreasing the size of heap by 1.

3. Repeat the above step 2 until heap is not empty.

Below is C++ implementation of the above approach.

```
// A program to take numbers from different machines and print them in sorted order
#include <stdio.h>

// A Linked List node
struct ListNode
{
    int data;
    struct ListNode* next;
};
```

```
// A Min Heap Node
struct MinHeapNode
{
    ListNode* head;
};

// A Min Heao (Collection of Min Heap nodes)
struct MinHeap
{
    int count;
    int capacity;
    MinHeapNode* array;
};

// A function to create a Min Heap of given capacity
MinHeap* createMinHeap( int capacity )
{
    MinHeap* minHeap = new MinHeap;
    minHeap->capacity = capacity;
    minHeap->count = 0;
    minHeap->array = new MinHeapNode [minHeap->capacity];
    return minHeap;
}

/* A utility function to insert a new node at the begining
   of linked list */
void push (ListNode** head_ref, int new_data)
{
    /* allocate node */
    ListNode* new_node = new ListNode;

    /* 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 swap two min heap nodes. This function
// is needed in minHeapify
void swap( MinHeapNode* a, MinHeapNode* b )
{
    MinHeapNode temp = *a;
    *a = *b;
    *b = temp;
}

// The standard minHeapify function.
void minHeapify( MinHeap* minHeap, int idx )
{
```

```
    int left, right, smallest;
    left = 2 * idx + 1;
    right = 2 * idx + 2;
    smallest = idx;

    if ( left < minHeap->count &&
         minHeap->array[left].head->data <
         minHeap->array[smallest].head->data
       )
        smallest = left;

    if ( right < minHeap->count &&
         minHeap->array[right].head->data <
         minHeap->array[smallest].head->data
       )
        smallest = right;

    if( smallest != idx )
    {
        swap( &minHeap->array[smallest], &minHeap->array[idx] );
        minHeapify( minHeap, smallest );
    }
}

// A utility function to check whether a Min Heap is empty or not
int isEmpty( MinHeap* minHeap )
{
    return (minHeap->count == 0);
}

// A standard function to build a heap
void buildMinHeap( MinHeap* minHeap )
{
    int i, n;
    n = minHeap->count  - 1;
    for( i = (n - 1) / 2; i >= 0; --i )
        minHeapify( minHeap, i );
}

// This function inserts array elements to heap and then calls
// buildHeap for heap property among nodes
void populateMinHeap( MinHeap* minHeap, ListNode* *array, int n )
{
    for( int i = 0; i < n; ++i )
        minHeap->array[ minHeap->count++ ].head = array[i];

    buildMinHeap( minHeap );
}

// Return minimum element from all linked lists
ListNode* extractMin( MinHeap* minHeap )
{
    if( isEmpty( minHeap ) )
         return NULL;
```

```
    // The root of heap will have minimum value
    MinHeapNode temp = minHeap->array[0];

    // Replace root either with next node of the same list.
    if( temp.head->next )
        minHeap->array[0].head = temp.head->next;
    else // If list empty, then reduce heap size
    {
        minHeap->array[0] = minHeap->array[ minHeap->count - 1 ];
        --minHeap->count;
    }

    minHeapify( minHeap, 0 );
    return temp.head;
}

// The main function that takes an array of lists from N machines
// and generates the sorted output
void externalSort( ListNode *array[], int N )
{
    // Create a min heap of size equal to number of machines
    MinHeap* minHeap = createMinHeap( N );

    // populate first item from all machines
    populateMinHeap( minHeap, array, N );

    while ( !isEmpty( minHeap ) )
    {
        ListNode* temp = extractMin( minHeap );
        printf( "%d ",temp->data );
    }
}

// Driver program to test above functions
int main()
{
    int N = 3; // Number of machines

    // an array of pointers storing the head nodes of the linked lists
    ListNode *array[N];

    // Create a Linked List 30->40->50 for first machine
    array[0] = NULL;
    push (&array[0], 50);
    push (&array[0], 40);
    push (&array[0], 30);

    // Create a Linked List 35->45 for second machine
    array[1] = NULL;
    push (&array[1], 45);
    push (&array[1], 35);

    // Create Linked List 10->60->70->80 for third machine
```

```
    array[2] = NULL;
    push (&array[2], 100);
    push (&array[2], 80);
    push (&array[2], 70);
    push (&array[2], 60);
    push (&array[2], 10);

    // Sort all elements
    externalSort( array, N );

    return 0;
}
```

Output:

```
 10 30 35 40 45 50 60 70 80 100
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/sort-numbers-stored-on-different-machines/

Category: Misc Tags: Advance Data Structures, Advanced Data Structures

Post navigation

← Find the k most frequent words from a file Microsoft Interview | Set 7 →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 49

Find the k most frequent words from a file

Given a book of words. Assume you have enough main memory to accommodate all words. design a data structure to find top K maximum occurring words. The data structure should be dynamic so that new words can be added.

A simple solution is to **use Hashing**. Hash all words one by one in a hash table. If a word is already present, then increment its count. Finally, traverse through the hash table and return the k words with maximum counts.

We can **use Trie and Min Heap** to get the k most frequent words efficiently. The idea is to use Trie for searching existing words adding new words efficiently. Trie also stores count of occurrences of words. A Min Heap of size k is used to keep track of k most frequent words at any point of time(Use of Min Heap is same as we used it to find k largest elements in this post).
Trie and Min Heap are linked with each other by storing an additional field in Trie 'indexMinHeap' and a pointer 'trNode' in Min Heap. The value of 'indexMinHeap' is maintained as -1 for the words which are currently not in Min Heap (or currently not among the top k frequent words). For the words which are present in Min Heap, 'indexMinHeap' contains, index of the word in Min Heap. The pointer 'trNode' in Min Heap points to the leaf node corresponding to the word in Trie.

Following is the complete process to print k most frequent words from a file.

Read all words one by one. For every word, insert it into Trie. Increase the counter of the word, if already exists. Now, we need to insert this word in min heap also. For insertion in min heap, 3 cases arise:

1. The word is already present. We just increase the corresponding frequency value in min heap and call minHeapify() for the index obtained by "indexMinHeap" field in Trie. When the min heap nodes are being swapped, we change the corresponding minHeapIndex in the Trie. Remember each node of the min heap is also having pointer to Trie leaf node.

2. The minHeap is not full. we will insert the new word into min heap & update the root node in the min heap node & min heap index in Trie leaf node. Now, call buildMinHeap().

3. The min heap is full. Two sub-cases arise.
....**3.1** The frequency of the new word inserted is less than the frequency of the word stored in the head of min heap. Do nothing.

....**3.2** The frequency of the new word inserted is greater than the frequency of the word stored in the head of min heap. Replace & update the fields. Make sure to update the corresponding min heap index of the "word to be replaced" in Trie with -1 as the word is no longer in min heap.

4. Finally, Min Heap will have the k most frequent words of all words present in given file. So we just need to print all words present in Min Heap.

```
// A program to find k most frequent words in a file
#include <stdio.h>
#include <string.h>
#include <ctype.h>

# define MAX_CHARS 26
# define MAX_WORD_SIZE 30

// A Trie node
struct TrieNode
{
    bool isEnd; // indicates end of word
    unsigned frequency;  // the number of occurrences of a word
    int indexMinHeap; // the index of the word in minHeap
    TrieNode* child[MAX_CHARS]; // represents 26 slots each for 'a' to 'z'.
};

// A Min Heap node
struct MinHeapNode
{
    TrieNode* root; // indicates the leaf node of TRIE
    unsigned frequency; //  number of occurrences
    char* word; // the actual word stored
};

// A Min Heap
struct MinHeap
{
    unsigned capacity; // the total size a min heap
    int count; // indicates the number of slots filled.
    MinHeapNode* array; //  represents the collection of minHeapNodes
};

// A utility function to create a new Trie node
TrieNode* newTrieNode()
{
    // Allocate memory for Trie Node
    TrieNode* trieNode = new TrieNode;

    // Initialize values for new node
    trieNode->isEnd = 0;
    trieNode->frequency = 0;
    trieNode->indexMinHeap = -1;
    for( int i = 0; i < MAX_CHARS; ++i )
        trieNode->child[i] = NULL;

    return trieNode;
}

// A utility function to create a Min Heap of given capacity
MinHeap* createMinHeap( int capacity )
{
    MinHeap* minHeap = new MinHeap;
```

```
    minHeap->capacity = capacity;
    minHeap->count  = 0;

    // Allocate memory for array of min heap nodes
    minHeap->array = new MinHeapNode [ minHeap->capacity ];

    return minHeap;
}

// A utility function to swap two min heap nodes. This function
// is needed in minHeapify
void swapMinHeapNodes ( MinHeapNode* a, MinHeapNode* b )
{
    MinHeapNode temp = *a;
    *a = *b;
    *b = temp;
}

// This is the standard minHeapify function. It does one thing extra.
// It updates the minHapIndex in Trie when two nodes are swapped in
// in min heap
void minHeapify( MinHeap* minHeap, int idx )
{
    int left, right, smallest;

    left = 2 * idx + 1;
    right = 2 * idx + 2;
    smallest = idx;
    if ( left < minHeap->count &&
         minHeap->array[ left ]. frequency <
         minHeap->array[ smallest ]. frequency
       )
        smallest = left;

    if ( right < minHeap->count &&
         minHeap->array[ right ]. frequency <
         minHeap->array[ smallest ]. frequency
       )
        smallest = right;

    if( smallest != idx )
    {
        // Update the corresponding index in Trie node.
        minHeap->array[ smallest ]. root->indexMinHeap = idx;
        minHeap->array[ idx ]. root->indexMinHeap = smallest;

        // Swap nodes in min heap
        swapMinHeapNodes (&minHeap->array[ smallest ], &minHeap->array[ idx ]);

        minHeapify( minHeap, smallest );
    }
}
```

```
// A standard function to build a heap
void buildMinHeap( MinHeap* minHeap )
{
    int n, i;
    n = minHeap->count - 1;

    for( i = ( n - 1 ) / 2; i >= 0; --i )
        minHeapify( minHeap, i );
}

// Inserts a word to heap, the function handles the 3 cases explained above
void insertInMinHeap( MinHeap* minHeap, TrieNode** root, const char* word )
{
    // Case 1: the word is already present in minHeap
    if( (*root)->indexMinHeap != -1 )
    {
        ++( minHeap->array[ (*root)->indexMinHeap ]. frequency );

        // percolate down
        minHeapify( minHeap, (*root)->indexMinHeap );
    }

    // Case 2: Word is not present and heap is not full
    else if( minHeap->count < minHeap->capacity )
    {
        int count = minHeap->count;
        minHeap->array[ count ]. frequency = (*root)->frequency;
        minHeap->array[ count ]. word = new char [strlen( word ) + 1];
        strcpy( minHeap->array[ count ]. word, word );

        minHeap->array[ count ]. root = *root;
        (*root)->indexMinHeap = minHeap->count;

        ++( minHeap->count );
        buildMinHeap( minHeap );
    }

    // Case 3: Word is not present and heap is full. And frequency of word
    // is more than root. The root is the least frequent word in heap,
    // replace root with new word
    else if ( (*root)->frequency > minHeap->array[0]. frequency )
    {

        minHeap->array[ 0 ]. root->indexMinHeap = -1;
        minHeap->array[ 0 ]. root = *root;
        minHeap->array[ 0 ]. root->indexMinHeap = 0;
        minHeap->array[ 0 ]. frequency = (*root)->frequency;

        // delete previously allocated memoory and
        delete [] minHeap->array[ 0 ]. word;
        minHeap->array[ 0 ]. word = new char [strlen( word ) + 1];
        strcpy( minHeap->array[ 0 ]. word, word );

        minHeapify ( minHeap, 0 );
```

```
    }
}

// Inserts a new word to both Trie and Heap
void insertUtil ( TrieNode** root, MinHeap* minHeap,
                        const char* word, const char* dupWord )
{
    // Base Case
    if ( *root == NULL )
        *root = newTrieNode();

    //  There are still more characters in word
    if ( *word != '\0' )
        insertUtil ( &((*root)->child[ tolower( *word ) - 97 ]),
                         minHeap, word + 1, dupWord );
    else // The complete word is processed
    {
        // word is already present, increase the frequency
        if ( (*root)->isEnd )
            ++( (*root)->frequency );
        else
        {
            (*root)->isEnd = 1;
            (*root)->frequency = 1;
        }

        // Insert in min heap also
        insertInMinHeap( minHeap, root, dupWord );
    }
}


// add a word to Trie & min heap.  A wrapper over the insertUtil
void insertTrieAndHeap(const char *word, TrieNode** root, MinHeap* minHeap)
{
    insertUtil( root, minHeap, word, word );
}

// A utility function to show results, The min heap
// contains k most frequent words so far, at any time
void displayMinHeap( MinHeap* minHeap )
{
    int i;

    // print top K word with frequency
    for( i = 0; i < minHeap->count; ++i )
    {
        printf( "%s : %d\n", minHeap->array[i].word,
                             minHeap->array[i].frequency );
    }
}

// The main funtion that takes a file as input, add words to heap
// and Trie, finally shows result from heap
```

```
void printKMostFreq( FILE* fp, int k )
{
    // Create a Min Heap of Size k
    MinHeap* minHeap = createMinHeap( k );

    // Create an empty Trie
    TrieNode* root = NULL;

    // A buffer to store one word at a time
    char buffer[MAX_WORD_SIZE];

    // Read words one by one from file.  Insert the word in Trie and Min Heap
    while( fscanf( fp, "%s", buffer ) != EOF )
        insertTrieAndHeap(buffer, &root, minHeap);

    // The Min Heap will have the k most frequent words, so print Min Heap nodes
    displayMinHeap( minHeap );
}

// Driver program to test above functions
int main()
{
    int k = 5;
    FILE *fp = fopen ("file.txt", "r");
    if (fp == NULL)
        printf ("File doesn't exist ");
    else
        printKMostFreq (fp, k);
    return 0;
}
```

Output:

```
 your : 3
well : 3
and : 4
to : 4
Geeks : 6
```

The above output is for a file with following content.

```
 Welcome to the world of Geeks
This portal has been created to provide well written well thought and well explained
solutions for selected questions If you like Geeks for Geeks and would like to contribute
here is your chance You can write article and mail your article to contribute at
geeksforgeeks org See your article appearing on the Geeks for Geeks main page and help
thousands of other Geeks
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/find-the-k-most-frequent-words-from-a-file/

Category: Misc Tags: Advance Data Structures

Post navigation

← Dynamic Programming | Set 22 (Box Stacking Problem) Sort numbers stored on different machines →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 50

Given a sequence of words, print all anagrams together | Set 2

Given an array of words, print all anagrams together. For example, if the given array is {“cat”, “dog”, “tac”, “god”, “act”}, then output may be “cat tac act dog god”.

We have discussed two different methods in the previous post. In this post, a more efficient solution is discussed.

Trie data structure can be used for a more efficient solution. Insert the sorted order of each word in the trie. Since all the anagrams will end at the same leaf node. We can start a linked list at the leaf nodes where each node represents the index of the original array of words. Finally, traverse the Trie. While traversing the Trie, traverse each linked list one line at a time. Following are the detailed steps.

1) Create an empty Trie
2) One by one take all words of input sequence. Do following for each word
…**a)** Copy the word to a buffer.
…**b)** Sort the buffer
…**c)** Insert the sorted buffer and index of this word to Trie. Each leaf node of Trie is head of a Index list. The Index list stores index of words in original sequence. If sorted buffe is already present, we insert index of this word to the index list.
3) Traverse Trie. While traversing, if you reach a leaf node, traverse the index list. And print all words using the index obtained from Index list.

```
// An efficient program to print all anagrams together
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define NO_OF_CHARS 26

// Structure to represent list node for indexes of words in
// the given sequence. The list nodes are used to connect
// anagrams at leaf nodes of Trie
struct IndexNode
{
    int index;
```

```
    struct IndexNode* next;
};

// Structure to represent a Trie Node
struct TrieNode
{
    bool isEnd;  // indicates end of word
    struct TrieNode* child[NO_OF_CHARS]; // 26 slots each for 'a' to 'z'
    struct IndexNode* head; // head of the index list
};


// A utility function to create a new Trie node
struct TrieNode* newTrieNode()
{
    struct TrieNode* temp = new TrieNode;
    temp->isEnd = 0;
    temp->head = NULL;
    for (int i = 0; i < NO_OF_CHARS; ++i)
        temp->child[i] = NULL;
    return temp;
}

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

/* A utility function to create a new linked list node */
struct IndexNode* newIndexNode(int index)
{
    struct IndexNode* temp = new IndexNode;
    temp->index = index;
    temp->next = NULL;
    return temp;
}

// A utility function to insert a word to Trie
void insert(struct TrieNode** root, char* word, int index)
{
    // Base case
    if (*root == NULL)
        *root = newTrieNode();

    if (*word != '\0')
        insert( &( (*root)->child[tolower(*word) - 'a'] ), word+1, index );
    else  // If end of the word reached
    {
        // Insert index of this word to end of index linked list
        if ((*root)->isEnd)
        {
            IndexNode* pCrawl = (*root)->head;
            while( pCrawl->next )
                pCrawl = pCrawl->next;
```

```
            pCrawl->next = newIndexNode(index);
        }
        else  // If Index list is empty
        {
            (*root)->isEnd = 1;
            (*root)->head = newIndexNode(index);
        }
    }
}

// This function traverses the built trie. When a leaf node is reached,
// all words connected at that leaf node are anagrams. So it traverses
// the list at leaf node and uses stored index to print original words
void printAnagramsUtil(struct TrieNode* root, char *wordArr[])
{
    if (root == NULL)
        return;

    // If a lead node is reached, print all anagrams using the indexes
    // stored in index linked list
    if (root->isEnd)
    {
        // traverse the list
        IndexNode* pCrawl = root->head;
        while (pCrawl != NULL)
        {
            printf( "%s \n", wordArr[ pCrawl->index ] );
            pCrawl = pCrawl->next;
        }
    }

    for (int i = 0; i < NO_OF_CHARS; ++i)
        printAnagramsUtil(root->child[i], wordArr);
}

// The main function that prints all anagrams together. wordArr[] is input
// sequence of words.
void printAnagramsTogether(char* wordArr[], int size)
{
    // Create an empty Trie
    struct TrieNode* root = NULL;

    // Iterate through all input words
    for (int i = 0; i < size; ++i)
    {
        // Create a buffer for this word and copy the word to buffer
        int len = strlen(wordArr[i]);
        char *buffer = new char[len+1];
        strcpy(buffer, wordArr[i]);

        // Sort the buffer
        qsort( (void*)buffer, strlen(buffer), sizeof(char), compare );

        // Insert the sorted buffer and its original index to Trie
```

```
        insert(&root,  buffer, i);
    }

    // Traverse the built Trie and print all anagrms together
    printAnagramsUtil(root, wordArr);
}


// Driver program to test above functions
int main()
{
    char* wordArr[] = {"cat", "dog", "tac", "god", "act", "gdo"};
    int size = sizeof(wordArr) / sizeof(wordArr[0]);
    printAnagramsTogether(wordArr, size);
    return 0;
}
```

Output:

```
 cat
tac
act
dog
god
gdo
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/given-a-sequence-of-words-print-all-anagrams-together-set-2/

Category: Strings Tags: Advance Data Structures

Post navigation

← Given a sequence of words, print all anagrams together | Set 1 [TopTalent.in] Interview with Arun Dobriyal who landed a job at Facebook, Palo Alto →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 51

Tournament Tree (Winner Tree) and Binary Heap

Given a team of N players. How many minimum games are required to find second best player?

We can use adversary arguments based on tournament tree (Binary Heap).

Tournament tree is a form of min (max) heap which is a complete binary tree. Every external node represents a player and internal node represents winner. In a tournament tree every internal node contains winner and every leaf node contains one player.

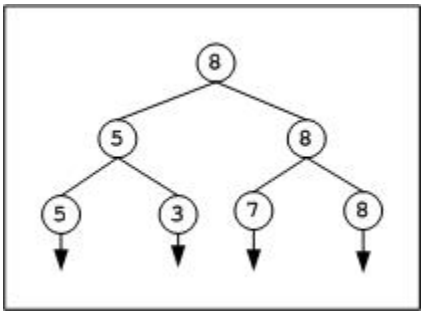
There will be N – 1 internal nodes in a binary tree with N leaf (external) nodes. For details see this post (put n = 2 in equation given in the post).

It is obvious that to select the best player among N players, (N – 1) players to be eliminated, i.e. we need minimum of (N – 1) games (comparisons). Mathematically we can prove it. In a binary tree I = E – 1, where I is number of internal nodes and E is number of external nodes. It means to find maximum or minimum element of an array, we need N – 1 (internal nodes) comparisons.

Second Best Player

The information explored during best player selection can be used to minimize the number of comparisons in tracing the next best players. For example, we can pick second best player in $\mathbf{(N + \log_2 N - 2)}$ comparisons. For details read this comment.

The following diagram displays a tournament tree (*winner tree*) as a max heap. Note that the concept of *loser tree* is different.



The above tree contains 4 leaf nodes that represent players and have 3 levels 0, 1 and 2. Initially 2 games are conducted at level 2, one between 5 and 3 and another one between 7 and 8. In the next move, one more game is conducted between 5 and 8 to conclude the final winner. Overall we need 3 comparisons. For second best player we need to trace the candidates participated with final winner, that leads to 7 as second best.

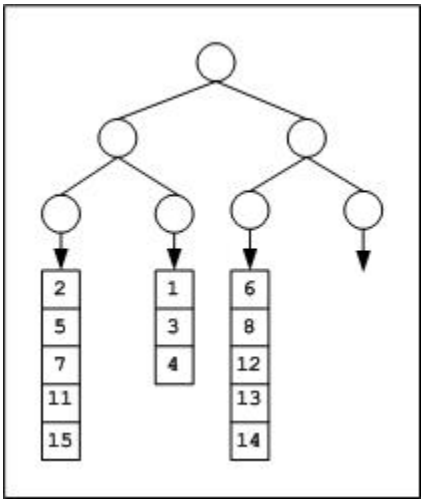
Median of Sorted Arrays

Tournament tree can effectively be used to find median of sorted arrays. Assume, given M sorted arrays of equal size L (for simplicity). We can attach all these sorted arrays to the tournament tree, one array per leaf. We need a tree of height **CEIL ($\log_2$M)** to have atleast M external nodes.

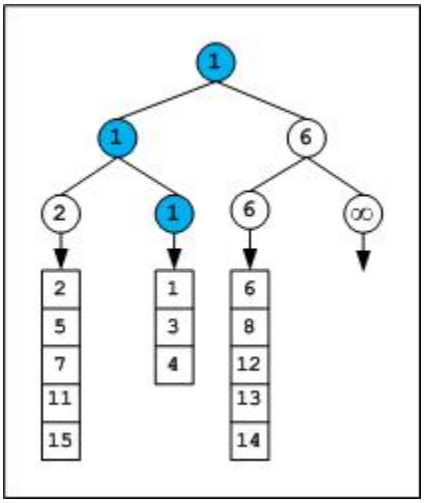
Consider an example. Given 3 (M = 3) sorted integer arrays of maximum size 5 elements.

```
 { 2, 5, 7, 11, 15 } ---- Array1
{1, 3, 4} ---- Array2
{6, 8, 12, 13, 14} ---- Array3
```

What should be the height of tournament tree? We need to construct a tournament tree of height $\log_2 3 .= 1.585 = 2$ rounded to next integer. A binary tree of height 2 will have 4 leaves to which we can attach the arrays as shown in the below figure.



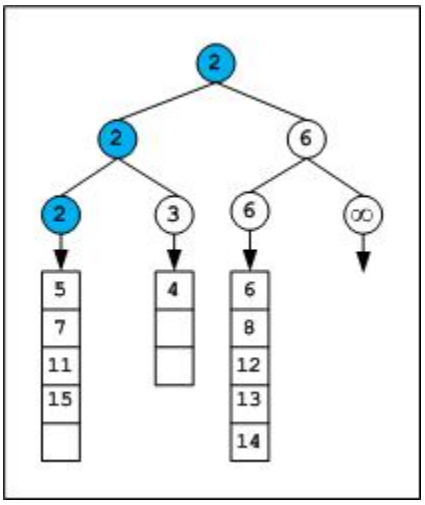
After the first tournament, the tree appears as below,



We can observe that the winner is from Array2. Hence the next element from Array2 will dive-in and games will be played along the winner path of previous tournament.

Note that infinity is used as sentinel element. Based on data being hold in nodes we can select the sentinel character. For example we usually store the pointers in nodes rather than keys, so NULL can serve as sentinel. If any of the array exhausts we will fill the corresponding leaf and upcoming internal nodes with sentinel.

After the second tournament, the tree appears as below,



The next winner is from Array1, so next element of Array1 array which is 5 will dive-in to the next round, and next tournament played along the path of 2.

The tournaments can be continued till we get median element which is (5+3+5)/2 = 7th element. Note that there are even better algorithms for finding median of union of sorted arrays, for details see the related links given below.

In general with M sorted lists of size L_1, L_2 … L_m requires time complexity of $O((L_1 + L_2 + \ldots + L_m) * logM)$ to merge all the arrays, and $O(m*logM)$ time to find median, where m is median position.

Select smallest one million elements from one billion unsorted elements:

As a simple solution, we can sort the billion numbers and select first one million.

On a limited memory system sorting billion elements and picking the first one million seems to be impractical. We can use tournament tree approach. At any time only elements of tree to be in memory.

Split the large array (perhaps stored on disk) into smaller size arrays of size one million each (or even smaller that can be sorted by the machine). Sort these 1000 small size arrays and store them on disk as individual files. Construct a tournament tree which can have atleast 1000 leaf nodes (tree to be of height 10 since $2^9 < 1000 < 2^{10}$, if the individual file size is even smaller we will need more leaf nodes). Every leaf node will have an engine that picks next element from the sorted file stored on disk. We can play the tournament tree game to extract first one million elements.

Total cost = sorting 1000 lists of one million each + tree construction + tournaments

Implementation

We need to build the tree (heap) in bottom-up manner. All the leaf nodes filled first. Start at the left extreme of tree and fill along the breadth (i.e. from 2^{k-1} to $2^k - 1$ where k is depth of tree) and play the game. After practicing with few examples it will be easy to write code. We will have code in an upcoming article.

Related Posts

Link 1, Link 2, Link 3, Link 4, Link 5, Link 6, Link 7.

— by **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/tournament-tree-and-binary-heap/

Chapter 52

Decision Trees - Fake (Counterfeit) Coin Puzzle (12 Coin Puzzle)

Let us solve the classic "fake coin" puzzle using decision trees. There are the two different variants of the puzzle given below. I am providing description of both the puzzles below, try to solve on your own, assume N = 8.

Easy: *Given* a two pan fair balance and *N identically looking coins, out of which only one coin is* ***lighter (or heavier)****.* To figure out the odd coin, *how many minimum number of weighing are required* in the worst case*?*

Difficult: *Given a two pan fair balance and N identically looking coins out of which only one coin* ***may be*** *defective. How can we trace which coin, if any, is odd one and also determine whether it is lighter or heavier in minimum number of trials in the worst case?*

Let us start with relatively simple examples. After reading every problem try to solve on your own.

Problem 1: (Easy)

Given 5 coins out of which one coin is ***lighter****. In the worst case, how many minimum number of weighing are required to figure out the odd coin?*

Name the coins as 1, 2, 3, 4 and 5. We know that one coin is lighter. Considering best out come of balance, we can group the coins in two different ways, [(1, 2), (3, 4) and (5)], or [(12), (34) and (5)]. We can easily rule out groups like [(123) and (45)], as we will get obvious answer. Any other combination will fall into one of these two groups, like [(2)(45) and (13)], etc.

Consider the first group, pairs (1, 2) and (3, 4). We can check (1, 2), if they are equal we go ahead with (3, 4). We need two weighing in worst case. The same analogy can be applied when the coin in heavier.

With the second group, weigh (12) and (34). If they balance (5) is defective one, otherwise pick the lighter pair, and we need one more weighing to find odd one.

Both the combinations need two weighing in case of 5 coins with prior information of one coin is lighter.

Analysis: In general, if we know that the coin is heavy or light, we can trace the coin in $\log_3(N)$ trials (rounded to next integer). If we represent the outcome of balance as ternary tree, every leaf represent an outcome. Since any coin among N coins can be defective, we need to get a 3-ary tree having minimum of N leaves. A 3-ary tree at k-th level will have 3^k leaves and hence we need $3^k >= N$.

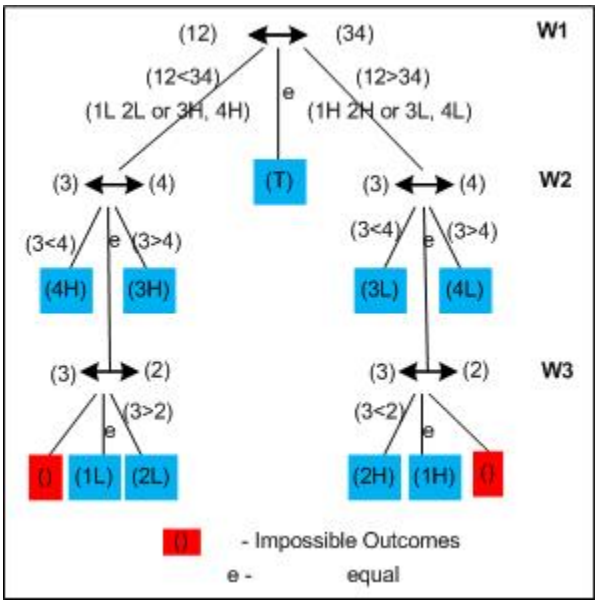
In other-words, in $\boldsymbol{k}$ trials we can examine upto $\boldsymbol{3^k}$ coins, if we know whether the defective coin is heavier or lighter. Given that a coin is heavier, verify that 3 trials are sufficient to find the odd coin among 12 coins, because $3^2 < 12 < 3^3$.

Problem 2: (Difficult)

We are given 4 coins, out of which only one coin ***may be*** *defective. We don't know, whether all coins are genuine or any defective one is present. How many number of weighing are required in worst case to figure out the odd coin, if present? We also need to tell whether it is heavier or lighter.*

From the above analysis we may think that k = 2 trials are sufficient, since a two level 3-ary tree yields 9 leaves which is greater than N = 4 (read the problem once again). Note that it is impossible to solve above 4 coins problem in two weighing. The decision tree confirms the fact (try to draw).

We can group the coins in two different ways, [(12, 34)] or [(1, 2) and (3, 4)]. Let us consider the combination (12, 34), the corresponding decision tree is given below. Blue leaves are valid outcomes, and red leaves are impossible cases. We arrived at impossible cases due to the assumptions made earlier on the path.



The outcome can be (12) < (34) i.e. we go on to left subtree or (12) > (34) i.e. we go on to right subtree.

The left subtree is possible in two ways,

- A) Either 1 or 2 can be lighter OR
- B) Either 3 or 4 can be heavier.

Further on the left subtree, as second trial, we weigh (1, 2) or (3, 4). Let us consider (3, 4) as the analogy for (1, 2) is similar. The outcome of second trail can be three ways

- A) (3) < (4) yielding 4 as defective heavier coin, OR
- B) (3) > (4) yielding 3 as defective heavier coin OR
- C) (3) = (4), yielding ambiguity. Here we need one more weighing to check a genuine coin against 1 or 2. In the figure I took (3, 2) where 3 is confirmed as genuine. We can get (3) > (2) in which 2 is lighter, or (3) = (2) in which 1 is lighter. Note that it impossible to get (3) < (2), it contradicts our assumption leaned to left side.

Similarly we can analyze the right subtree. We need two more weighings on right subtree as well.

Overall we need 3 weighings to trace the odd coin. Note that we are unable to utilize two outcomes of 3-ary trees. Also, the tree is not full tree, middle branch terminated after first weighing. Infact, we can get 27 leaves of 3 level full 3-ary tree, but only we got 11 leaves including impossible cases.

Analysis: Given N coins, all may be genuine or only one coin is defective. We need a decision tree with atleast (2N + 1) leaves correspond to the outputs. Because there can be N leaves to be lighter, or N leaves to be heavier or one genuine case, on total (2N + 1) leaves.

As explained earlier ternary tree at level k, can have utmost 3^k leaves and we need a tree with leaves of $3^k > (2N + 1)$.

In other words, we need atleast $k > log_3(2N + 1)$ weighing to find the defective one.

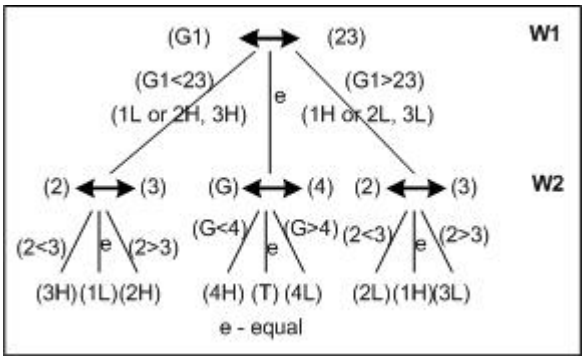
Observe the above figure that not all the branches are generating leaves, i.e. we are missing valid outputs under some branches that leading to more number of trials. When possible, we should group the coins in such a way that every branch is going to yield valid output (in simple terms generate full 3-ary tree). Problem 4 describes this approach of 12 coins.

Problem 3: (Special case of two pan balance)

*We are given 5 coins, a group of 4 coins out of which one coin is defective (we **don't know** whether it is heavier or lighter), and one coin is genuine. How many weighing are required in worst case to figure out the odd coin whether it is heavier or lighter?*

Label the coins as 1, 2, 3, 4 and G (genuine). We now have some information on coin purity. We need to make use that in the groupings.

We can best group them as [(G1, 23) and (4)]. Any other group can't generate full 3-ary tree, try yourself. The following diagram explains the procedure.



The middle case (G1) = (23) is self explanatory, i.e. 1, 2, 3 are genuine and 4th coin can be figured out lighter or heavier in one more trial.

The left side of tree corresponds to the case (G1) < (23). This is possible in two ways, either 1 should be lighter or either of (2, 3) should be heavier. The former instance is obvious when next weighing (2, 3) is balanced, yielding 1 as lighter. The later instance could be (2) < (3) yielding 3 as heavier or (2) > (3) yielding 2 as heavier. The leaf nodes on left branch are named to reflect these outcomes.

The right side of tree corresponds to the case (G1) > (23). This is possible in two ways, either 1 is heavier or either of (2, 3) should be lighter. The former instance is obvious when the next weighing (2, 3) is balanced, yielding 1 as heavier. The later case could be (2) < (3) yielding 2 as lighter coin, or (2) > (3) yielding 3 as lighter.

In the above problem, under any possibility we need only two weighing. We are able to use all outcomes of two level full 3-ary tree. We started with (N + 1) = 5 coins where N = 4, we end up with (2N + 1) = 9 leaves. ***Infact we should have 11 outcomes since we stared with 5 coins, where are other 2 outcomes? These two outcomes can be declared at the root of tree itself (prior to first weighing), can you figure these two out comes?***

If we observe the figure, after the first weighing the problem reduced to "we know three coins, either one can be lighter (heavier) or one among other two can be heavier (lighter)". This can be solved in one weighing (read Problem 1).

Analysis: Given (N + 1) coins, one is genuine and the rest N can be genuine or only one coin is defective. The required decision tree should result in minimum of (2N + 1) leaves. Since the total possible outcomes are (2(N + 1) + 1), number of weighing (trials) are given by the height of ternary tree, k >= $\log_3$[2(N + 1) + 1]. *Note the equality sign.*

Rearranging k and N, *we can weigh maximum of N <= (3^k – 3)/2 coins in k trials.*

Problem 4: (The classic 12 coin puzzle)

You are given two pan fair balance. You have 12 identically looking coins out of which one coin may be lighter or heavier. How can you find odd coin, if any, in minimum trials, also determine whether defective coin is lighter or heavier, in the worst case?

How do you want to group them? Bi-set or tri-set? Clearly we can discard the option of dividing into two equal groups. It can't lead to best tree. *From the above two examples, we can ensure that the decision tree can be used in optimal way if we can reveal atleaset one genuine coin.* Remember to group coins such that the first weighing reveals atleast one genuine coin.

Let us name the coins as 1, 2, … 8, A, B, C and D. We can combine the coins into 3 groups, namely (1234), (5678) and (ABCD). Weigh (1234) and (5678). You are encouraged to draw decision tree while reading the procedure. The outcome can be three ways,

1. (1234) = (5678), both groups are equal. Defective coin may be in (ABCD) group.
2. (1234) < (5678), i.e. first group is less in weight than second group.
3. (1234) > (5678), i.e. first group is more in weight than second group.

The output (1) can be solved in two more weighing as special case of two pan balance given in Problem 3. We know that groups (1234) and (5678) are genuine and defective coin may be in (ABCD). Pick one genuine coin from any of weighed groups, and proceed with (ABCD) as explained in Problem 3.

Outcomes (2) and (3) are special. In both the cases, we know that (ABCD) is genuine. And also, we know a set of coins being lighter and a set of coins being heavier. We need to shuffle the weighed two groups in such a way that we end up with smaller height decision tree.

Consider the second outcome where (1234) < (5678). It is possible when any coin among (1, 2, 3, 4) is lighter or any coin among (5, 6, 7, 8) is heavier. We revealed lighter or heavier possibility after first weighing. If we proceed as in Problem 1, we will not generate best decision tree. Let us shuffle coins as (1235) and (4BCD) as new groups (there are different shuffles possible, they also lead to minimum weighing, can you try?). If we weigh these two groups again the outcome can be three ways, i) (1235) < (4BCD) yielding one among 1, 2, 3 is lighter which is similar to Problem 1 explained above, we need one more weighing, ii) (1235) = (4BCD) yielding one among 6, 7, 8 is heavier which is similar to Problem 1 explained above, we need one more weighing iii) (1235) > (4BCD) yielding either 5 as heavier coin or 4 as lighter coin, at the expense of one more weighing.

Similar way we can also solve the right subtree (third outcome where (1234) > (5678)) in two more weighing.

We are able to solve the 12 coin puzzle in 3 weighing in the worst case.

Few Interesting Puzzles:

1. Solve Problem 4 with N = 8 and N = 13, How many minimum trials are required in each case?
2. Given a function *int weigh(A[], B[])* where A and B are arrays (need not be equal size). The function returns -1, 0 or 1. It returns 0 if sum of all elements in A and B are equal, -1 if A < B and 1 if A > B. Given an array of 12 elements, all elements are equal except one. The odd element can be as that of others, smaller or greater than others. Write a program to find the odd element (if any) using *weigh()* minimum number of times.
3. You might have seen 3-pan balance in science labs during school days. Given a 3-pan balance (4 outcomes) and N coins, how many minimum trials are needed to figure out odd coin?

References:

Similar problem was provided in one of the exercises of the book "Introduction to Algorithms by Levitin". Specifically read section 5.5 and section 11.2 including exercises.

– – – by **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

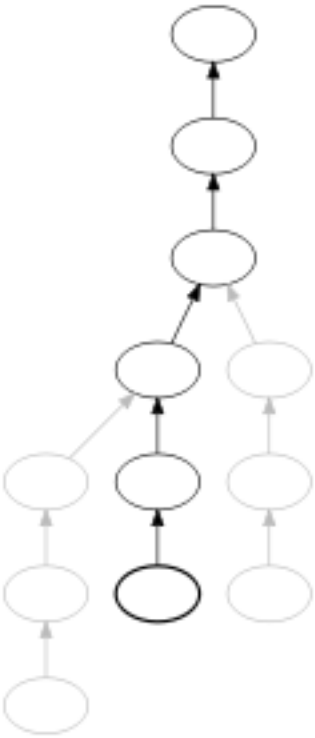
Source

http://www.geeksforgeeks.org/decision-trees-fake-coin-puzzle/

Chapter 53

Spaghetti Stack

Spaghetti stack
A spaghetti stack is an N-ary tree data structure in which child nodes have pointers to the parent nodes (but not vice-versa)



Spaghetti stack structure is used in situations when records are dynamically pushed and popped onto a stack as execution progresses, but references to the popped records remain in use. Following are some applications of Spaghetti Stack.

Compilers for languages such as C create a spaghetti stack as it opens and closes symbol tables representing block scopes. When a new block scope is opened, a symbol table is pushed onto a stack. When the closing curly brace is encountered, the scope is closed and the symbol table is popped. But that symbol table is remembered, rather than destroyed. And of course it remembers its higher level "parent" symbol table and so on.

Spaghetti Stacks are also used to implement Disjoint-set data structure.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Sources:
http://en.wikipedia.org/wiki/Spaghetti_stack

Source

http://www.geeksforgeeks.org/g-fact-87/

Category: Misc Tags: Advanced Data Structures

Chapter 54

Data Structure for Dictionary and Spell Checker?

Which data structure can be used for efficiently building a word dictionary and Spell Checker?

The answer depends upon the functionalists required in Spell Checker and availability of memory. For example following are few possibilities.

Hashingis one simple option for this. We can put all words in a hash table. Refer thispaper which compares hashing with self-balancing Binary Search Trees and Skip List, and shows that hashing performs better.

Hashing doesn't support operations like prefix search. Prefix search is something where a user types a prefix and your dictionary shows all words starting with that prefix. Hashing also doesn't support efficient printing of all words in dictionary in alphabetical order and nearest neighbor search.

If we want both operations, look up and prefix search, **Trie** is suited. With Trie, we can support all operations like insert, search, delete in O(n) time where n is length of the word to be processed. Another advantage of Trie is, we can print all words in alphabetical order which is not possible with hashing.

The disadvantage of Trie is, it requires lots of space. If space is concern, then **Ternary Search Tree** can be preferred. In Ternary Search Tree, time complexity of search operation is O(h) where h is height of the tree. Ternary Search Trees also supports other operations supported by Trie like prefix search, alphabetical order printing and nearest neighbor search.

If we want to support suggestions, like google shows "*did you mean ...*", then we need to find the closest word in dictionary. The closest word can be defined as the word that can be obtained with minimum number of character transformations (add, delete, replace). A Naive way is to take the given word and generate all words which are 1 distance (1 edit or 1 delete or 1 replace) away and one by one look them in dictionary. If nothing found, then look for all words which are 2 distant and so on. There are many complex algorithms for this. As perthe wiki page, The most successful algorithm to date is Andrew Golding and Dan Roth's Window-based spelling correction algorithm.

See thisfor a simple spell checker implementation.

This article is compiled by **Piyush**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/data-structure-dictionary-spell-checker/

Chapter 55

Binary Indexed Tree or Fenwick tree

Let us consider the following problem to understand Binary Indexed Tree.

We have an array arr[0 . . . n-1]. We should be able to
1 Find the sum of first i elements.
2 Change value of a specified element of the array arr[i] = x where 0 <= i <= n-1.

A **simple solution** is to run a loop from 0 to i-1 and calculate sum of elements. To update a value, simply do arr[i] = x. The first operation takes O(n) time and second operation takes O(1) time. Another simple solution is to create another array and store sum from start to i at the i'th index in this array. Sum of a given range can now be calculated in O(1) time, but update operation takes O(n) time now. This works well if the number of query operations are large and very few updates.

Can we perform both the operations in O(log n) time once given the array?
One Efficient Solution is to use Segment Tree that does both operations in O(Logn) time.

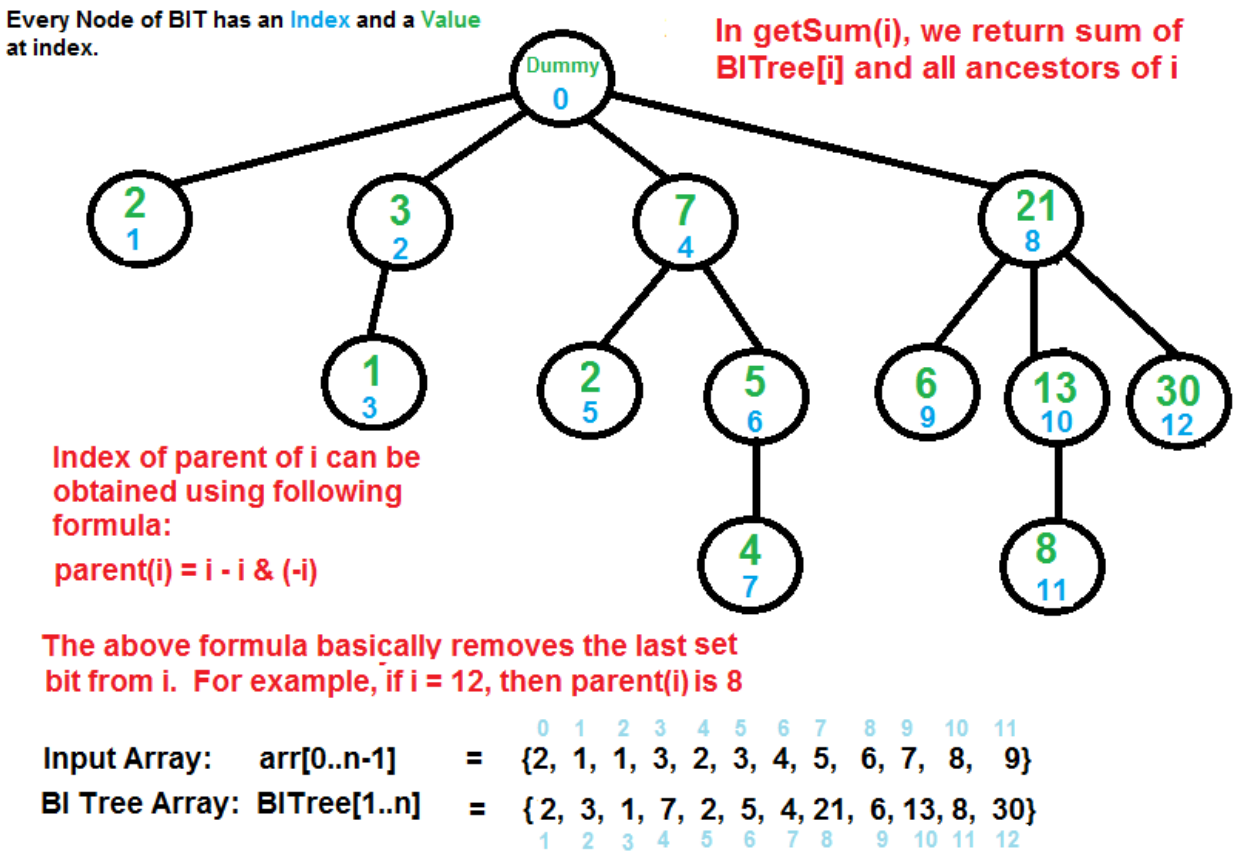
Using Binary Indexed Tree, we can do both tasks in O(Logn) time. The advantages of Binary Indexed Tree over Segment are, requires less space and very easy to implement..

Representation
Binary Indexed Tree is represented as an array. Let the array be BITree[]. Each node of Binary Indexed Tree stores sum of some elements of given array. Size of Binary Indexed Tree is equal to n where n is size of input array. In the below code, we have used size as n+1 for ease of implementation.

Construction
We construct the Binary Indexed Tree by first initializing all values in BITree[] as 0. Then we call update() operation for all indexes to store actual sums, update is discussed below.

Operations

```
getSum(index): Returns sum of arr[0..index]
// Returns sum of arr[0..index] using BITree[0..n].  It assumes that
// BITree[] is constructed for given array arr[0..n-1]
1) Initialize sum as 0 and index as index+1.
2) Do following while index is greater than 0.
...a) Add BITree[index] to sum
...b) Go to parent of BITree[index].  Parent can be obtained by removing
     the last set bit from index, i.e., index = index - (index & (-index))
3) Return sum.
```



View of Binary Indexed Tree to unserstand getSum() operation

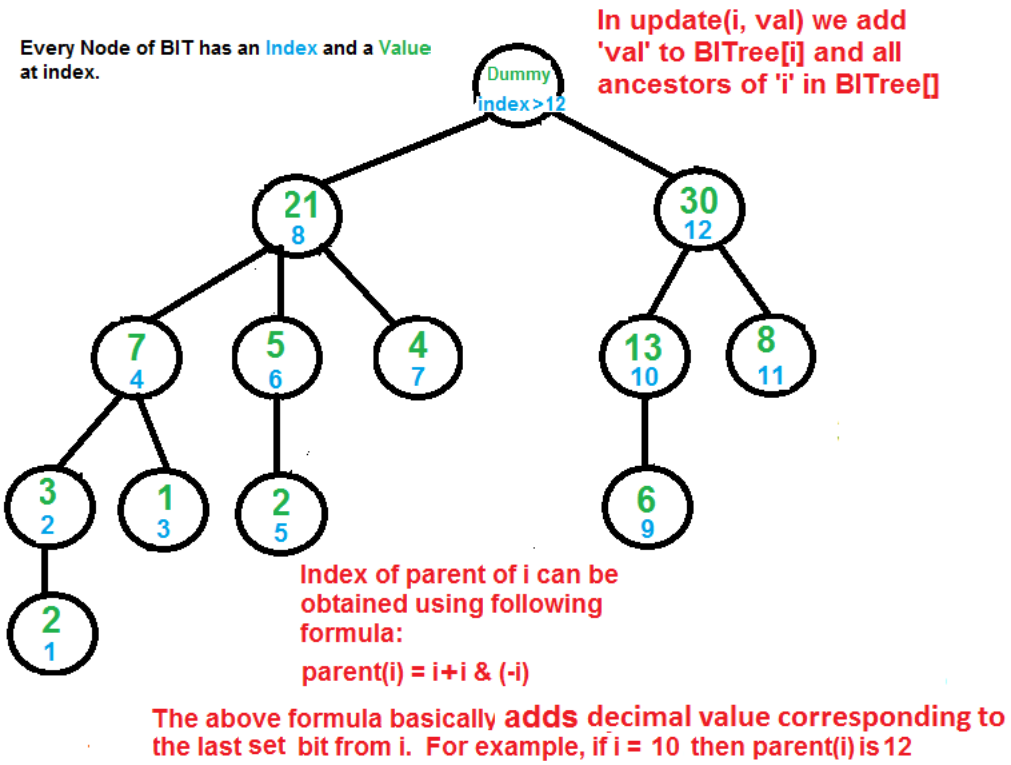
The above diagram demonstrates working of getSum(). Following are some important observations.

Node at index 0 is a dummy node.

A node at index y is parent of a node at index x, iff y can be obtained by removing last set bit from binary representation of x.

A child x of a node y stores sum of elements from of y(exclusive y) and of x(inclusive x).

```
update(index, val): Updates BIT for operation arr[index] += val
// Note that arr[] is not changed here.  It changes
// only BI Tree for the already made change in arr[].
1) Initialize index as index+1.
2) Do following while index is smaller than or equal to n.
...a) Add value to BITree[index]
...b) Go to parent of BITree[index].  Parent can be obtained by removing
     the last set bit from index, i.e., index = index - (index & (-index))
```



Contents of arr[] and BITree[] are same as above diagram for getSum()

View of Binary Indexed Tree to understand update() operation

The update process needs to make sure that all BITree nodes that have arr[i] as part of the section they cover must be updated. We get all such nodes of BITree by repeatedly adding the decimal number corresponding to the last set bit.

How does Binary Indexed Tree work?
The idea is based on the fact that all positive integers can be represented as sum of powers of 2. For example 19 can be represented as 16 + 2 + 1. Every node of BI Tree stores sum of n elements where n is a power of 2. For example, in the above first diagram for getSum(), sum of first 12 elements can be obtained by sum of last 4 elements (from 9 to 12) plus sum of 8 elements (from 1 to 8). The number of set bits in binary representation of a number n is O(Logn). Therefore, we traverse at-most O(Logn) nodes in both getSum() and update() operations. Time complexity of construction is O(nLogn) as it calls update() for all n elements.

Implementation:
Following is C++ implementation of Binary Indexed Tree.

```
// C++ code to demonstrate operations of Binary Index Tree
#include <iostream>
using namespace std;

/*            n  --> No. of elements present in input array.
    BITree[0..n] --> Array that represents Binary Indexed Tree.
    arr[0..n-1]  --> Input array for whic prefix sum is evaluated. */
```

```
// Returns sum of arr[0..index]. This function assumes
// that the array is preprocessed and partial sums of
// array elements are stored in BITree[].
int getSum(int BITree[], int n, int index)
{
    int sum = 0; // Iniialize result

    // index in BITree[] is 1 more than the index in arr[]
    index = index + 1;

    // Traverse ancestors of BITree[index]
    while (index>0)
    {
        // Add current element of BITree to sum
        sum += BITree[index];

        // Move index to parent node
        index -= index & (-index);
    }
    return sum;
}

// Updates a node in Binary Index Tree (BITree) at given index
// in BITree.  The given value 'val' is added to BITree[i] and
// all of its ancestors in tree.
void updateBIT(int *BITree, int n, int index, int val)
{
    // index in BITree[] is 1 more than the index in arr[]
    index = index + 1;

    // Traverse all ancestors and add 'val'
    while (index <= n)
    {
       // Add 'val' to current node of BI Tree
       BITree[index] += val;

       // Update index to that of parent
       index += index & (-index);
    }
}

// Constructs and returns a Binary Indexed Tree for given
// array of size n.
int *constructBITree(int arr[], int n)
{
    // Create and initialize BITree[] as 0
    int *BITree = new int[n+1];
    for (int i=1; i<=n; i++)
        BITree[i] = 0;

    // Store the actual values in BITree[] using update()
    for (int i=0; i<n; i++)
        updateBIT(BITree, n, i, arr[i]);
```

```
    // Uncomment below lines to see contents of BITree[]
    //for (int i=1; i<=n; i++)
    //      cout << BITree[i] << " ";

    return BITree;
}


// Driver program to test above functions
int main()
{
    int freq[] = {2, 1, 1, 3, 2, 3, 4, 5, 6, 7, 8, 9};
    int n = sizeof(freq)/sizeof(freq[0]);
    int *BITree = constructBITree(freq, n);
    cout << "Sum of elements in arr[0..5] is "
         << getSum(BITree, n, 5);

    // Let use test the update operation
    freq[3] += 6;
    updateBIT(BITree, n, 3, 6); //Update BIT for above change in arr[]

    cout << "\nSum of elements in arr[0..5] after update is "
         << getSum(BITree, n, 5);

    return 0;
}
```

Output:

```
 Sum of elements in arr[0..5] is 12
Sum of elements in arr[0..5] after update is 18
```

Can we extend the Binary Indexed Tree for range Sum in Logn time?
This is simple to answer. The rangeSum(l, r) can be obtained as getSum(r) – getSum(l-1).

Applications:
Used to implement the arithmetic coding algorithm. Development of operations it supports were primarily motivated by use in that case. See thisfor more details.

References:
http://en.wikipedia.org/wiki/Fenwick_tree

http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=binaryIndexedTrees

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

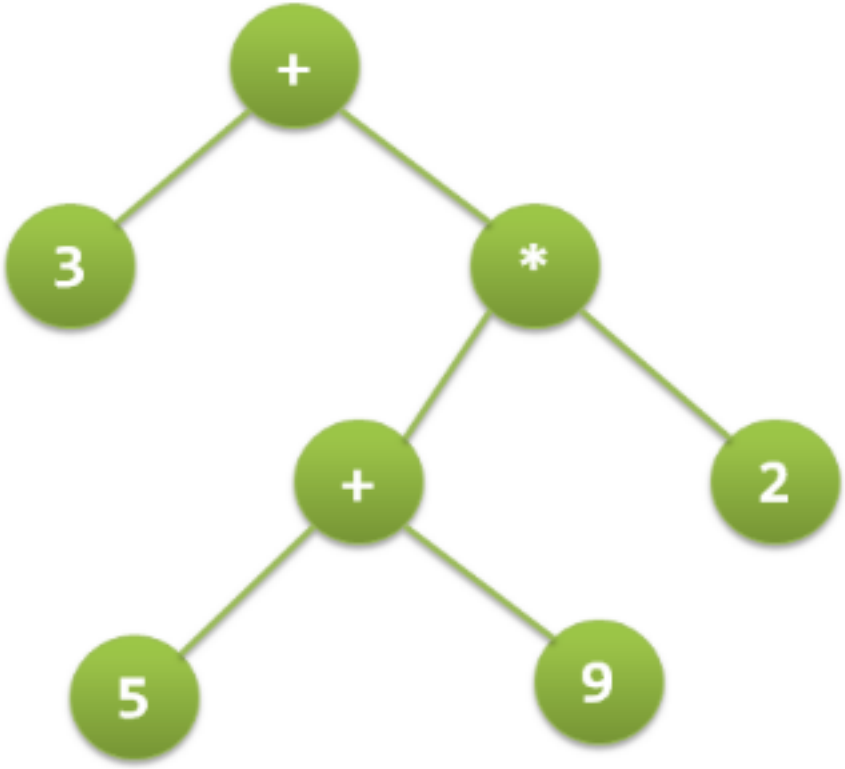
Source

http://www.geeksforgeeks.org/binary-indexed-tree-or-fenwick-tree-2/

Chapter 56

Expression Tree

Expression tree is a binary tree in which each internal node corresponds to operator and each leaf node corresponds to operand so for example expression tree for 3 + ((5+9)*2) would be:



Inorder traversal of expression tree produces infix version of given postfix expression (same with preorder traversal it gives prefix expression)

Evaluating the expression represented by expression tree:

```
Let t be the expression tree
If  t is not null then
      If t.value is operand then
                Return  t.value
```

```
    A = solve(t.left)
    B = solve(t.right)

    // calculate applies operator 't.value'
    // on A and B, and returns value
    Return calculate(A, B, t.value)
```

Construction of Expression Tree:
Now For constructing expression tree we use a stack. We loop through input expression and do following for every character.
1) If character is operand push that into stack
2) If character is operator pop two values from stack make them its child and push current node again.
At the end only element of stack will be root of expression tree.

Below is C++ implementation t

```
// C++ program for expression tree
#include<bits/stdc++.h>
using namespace std;

// An expression tree node
struct et
{
    char value;
    et* left, *right;
};

// A utility function to check if 'c'
// is an operator
bool isOperator(char c)
{
    if (c == '+' || c == '-' ||
            c == '*' || c == '/' ||
            c == '^')
        return true;
    return false;
}

// Utility function to do inorder traversal
void inorder(et *t)
{
    if(t)
    {
        inorder(t->left);
        printf("%c ", t->value);
        inorder(t->right);
    }
}

// A utility function to create a new node
et* newNode(int v)
{
```

```
    et *temp = new et;
    temp->left = temp->right = NULL;
    temp->value = v;
    return temp;
};

// Returns root of constructed tree for given
// postfix expression
et* constructTree(char postfix[])
{
    stack<et *> st;
    et *t, *t1, *t2;

    // Traverse through every character of
    // input expression
    for (int i=0; i<strlen(postfix); i++)
    {
        // If operand, simply push into stack
        if (!isOperator(postfix[i]))
        {
            t = newNode(postfix[i]);
            st.push(t);
        }
        else // operator
        {
            t = newNode(postfix[i]);

            // Pop two top nodes
            t1 = st.top(); // Store top
            st.pop();      // Remove top
            t2 = st.top();
            st.pop();

            //  make them children
            t->right = t1;
            t->left = t2;

            // Add this subexpression to stack
            st.push(t);
        }
    }

    //  only element will be root of expression
    // tree
    t = st.top();
    st.pop();

    return t;
}

// Driver program to test above
int main()
{
    char postfix[] = "ab+ef*g*-";
```

```
    et* r = constructTree(postfix);
    printf("infix expression is \n");
    inorder(r);
    return 0;
}
```

Output:

```
 infix expression is
a + b - e * f * g
```

This article is contributed by **Utkarsh Trivedi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/expression-tree/

Category: Trees