Novelty Journals

International Journal of Novel Research in Computer Science and Software Engineering
Vol. 1, Issue 1, pp: (20-29), Month: September-December 2014, Available at: www.noveltyjournals.com

Converting BST into Balanced BST using List

[1]Shaik Kareem Basha, [2]Shaik Jaheda Begum, [3]Fouzia bano

[1,2,3] Asst. Professor CSE, Hyderabad Institute of Technology And Management (HITAM), India

***Abstract*: Binary Search Tree (BST) and Balanced Binary Search Trees (BBST) are a category of Binary Trees, which provides efficient retrievals of data items. A BST T may be an empty binary tree. If non empty then it contains finite number of nodes. Though Binary Search Trees in comparision to sequential lists report a better performance of O(log n) time complexity for their insert, delete and retrieval operations, they also posses set backs. There are instances where binary search trees may grow to heights that equal n, the number of elements to be represented as the tree, there by degrading their performance. To eliminate this drawback, it is essential to convert BST into Balanced BST. Trees, whose height in the worst case turns out to be O(log n) are known as BBST. In this paper, five Algorithms ConstructBBST(), DivideList(), RDivideList(), ConstructList(), DisplayBBST() are proposed. We followed the various stages of Software Development Life Cycle to demonstrate the proposed algorithms. Section I will give introduction about proposed algorithms. In section II, proposed Algorithms are Analyzed and Designed. In section III, proposed algorithms are Implemented using C programming Language. In section IV, we will test the implementation of proposed algorithms using different test cases. In section V, we conclude the proposed Algorithms.**

I. INTRODUCTION

Binary Search Trees (BST) and Balanced Binary Search Trees (BBST) are category of binary trees, which provides efficient retrievals. A BST T may be an empty binary tree. If non empty then it contains finite number of nodes. It must satisfy the following norms: 1) All keys of BST must be distinct. 2) All keys in the left sub tree of T are less than the root element. 3) All keys in the right sub tree of T are greater than the root element. 4) The left and right sub trees of T are also binary search trees. The inorder traversal of a binary search tree T arranges all the nodes in ascending order. A binary search tree is commonly represented using linked list representation in the same way as that of a binary tree. Though binary search trees in comparision to sequential lists report a better performance of O(log n) time complexity for their insert, delete and retrieval operations, they also posses drawbacks. There are instances where binary search trees may grow to heights that equal n, number of elements to be represented as a tree, there by degrading their performance. This may occur due to a sequence of insert operation or delete operations. To eliminate this drawback, it is necessary to convert BST into Balanced BST. Trees whose height in the worst case turns out to be O(log n) are known as balanced trees. There are many applications of BBST; some of them are Representation of symbol tables in Compiler Design. In this paper, five Algorithms ConstructBBST(), DivideList(), RDivideList(), ConstructList(), DisplayBBST() are proposed. ConstructBBST() algorithm constructs the Balanced Binary Search Tree from BST using InOrder Traversal List of BST. DivideList() algorithm divides InOrder Traversal List of BST into sub lists and mid elements, which are inserted into BBST. This algorithm uses Queue Data Structure to store indices of left sub list and right sub list obtained after division of list, based on mid index. RDivideList() algorithm is a recursive algorithm, which also divides InOrder Traversal List into left sub lists and right sub lists which is recursively called until all the elements of list are placed into Balanced Binary Search Tree. ConstructList() algorithm is used to construct InOrder Traversal List of given Binary Search Tree (BST). DisplayBBST() algorithm is used to display elements of BBST.

International Journal of Novel Research in Computer Science and Software Engineering
Vol. 1, Issue 1, pp: (20-29), Month: September-December 2014, Available at: www.noveltyjournals.com

II. ANALYSIS AND DESIGN OF PROPOSED ALGORITHMS

In Fig 2.1 For a given Binary Search Tree (BST), InOrder Traversal List is constructed by using ArrayList called as Input. It is divided into sub lists based upon the mid index. Mid element of Input List is inserted into Balanced Binary Search Tree (BBST).

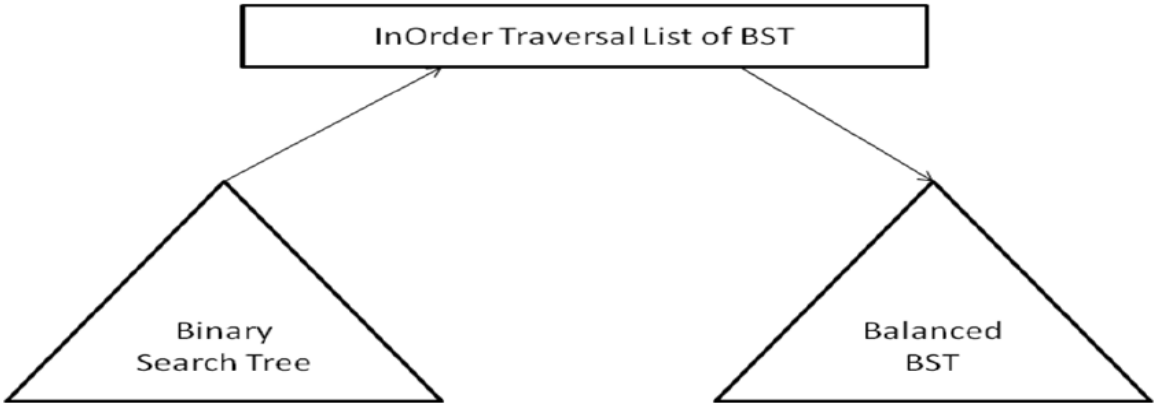

Fig 2.1 Converting BST into BBST using InOrder Traversal List of BST

Algorithm 2.1 constructs BBST using InOrder Traversal list of given BST. It takes bst, root of Binary Search Tree (BST) as input, which is converted into Balanced Binary Search Tree (BBST). It constructs InOrder Traversal List of bst, divides the List into un breakable lists and display the nodes of Balanced Binary Search Tree (BBST) using PreOrder Traversal.

Algorithm ConstructBBST(bst) {

// bst is the root of binary search tree, which is to be converted into BBST

ConstructList(bst); //constructing InOrder Traversal List of bst

DivideList(1,k); (OR) RDivideList(1,k); // Dividing InOrder Traversal List into unbreakable lists

DisplayBBST(bbst); // Displaying the nodes of BBST using PreOrder Traversal

}

Algorithm 2.1 Construction of BBST using In Order Traversal List of BST

The time taken by ConstructBBST() algorithm is as follows:

T(n)=O(n)+O(n)+O(n)

T(n)=O(n)

Algorithm 2.2 is a recursive algorithm, which constructs InOrder Traversal List of given BST. It takes bst, root of Binary Search Tree (BST) as input. It is a recursive algorithm, which initially checks whether the current node of BST is NULL or not. It is not NULL then recursively left sub tree of current node of BST is called, data of current node is placed into Input ArrayList, recursively right sub tree of current node of BST is called. This process continues until all the nodes of BST are visited.

Algorithm ConstructList(bst) {

// bst is the root of Binary Search Tree, whose InOrder Traversal List is constructed

if(bst != NULL) // if BST is not empty{

ConstructList(bst->left); // construct list using left sub tree of BST

k=k+1;

Input[k]=bst->data; // place node value into Input Array List

ConstructList(bst->right); // construct list using right sub tree of BST

}

Algorithm 2.2 Construction of In Order Traversal List of BST

The time taken by Algorithm 2.2 is as follows:

T(n)=1+2T(n/2)

T(n)=1+2(1+2T(n/4))

T(n)=1+2+22T(n/4)

T(n)=1+2+22(1+2T(n/8))

T(n)=1+2+22+23T(n/8)

T(n)=(1+2+22) + 23T(n/23)

T(n)=(1+2+22+----------+2k-1) + 2kT(n/2k)

T(n)=(2k-1)+2kT(n/2k)

Let n=2k , k=log n

T(n)=n-1+nT(1)

T(n)=2n-1

T(n)=O(n)

Algorithm 2.3 divides the InOrder Traversal List of BST into left sub list and right sub list based upon the mid index. Mid element of list is inserted into BBST. The indices of left sub list and right sub list are placed into queue. If queue is not empty then take indices of list from queue and repeat the above process until the queue becomes empty.

Algorithm DivideList(low,high) {

// low is the starting index of InOrder Traversal Array List of BST

//high is the ending index of InOrder Traversal Array List of BST

while(true) {

if(low<=high) // if List is Breakable{

mid=(low+high)/2; // Find the mid index of List

insert(&bbst,input[mid]); // insert mid element of List into BBST

rear=rear+1;

Queue[rear].low=low; // store starting index of left sub list into queue

Queue[rear].high=mid-1; // store ending index of left sub list into queue

rear=rear+1;

Queue[rear].low=mid+1; // store starting index of right sub list into queue

Queue[rear].high=high; // store ending index of right sub list into queue

}

If(front<rear) // if queue is not empty{

front=front+1;

low=queue[front].low; // assign starting index of list to low

high=queue[front].high; // assign ending index of list to high

}

else break; // if queue becomes empty

} }

Algorithm 2.3 Dividing In Order Traversal List of BST using Queue

The time taken by Algorithm 2.3 is as follows:

T(n)=1+2T(n/2)

T(n)=1+2(1+2T(n/4))

T(n)=1+2+22T(n/4)

T(n)=1+2+22(1+2T(n/8))

T(n)=1+2+22+23T(n/8)

T(n)=(1+2+22) + 23T(n/23)

T(n)=(1+2+22+----------+2k-1) + 2kT(n/2k)

T(n)=(2k-1)+2kT(n/2k)

Let n=2k , k=log n

T(n)=n-1+nT(1)

T(n)=2n-1

T(n)=O(n)

Algorithm 2.4 is a recursive algorithm, which divides the Inorder Traversal List of BST into left sub list and right sub list based upon the mid index. Mid element is inserted into BBST, recursively left sub list and right sub lists are called and the above process is repeated until all the elements of list are inserted into BBST.

Algorithm RDivideList(low,high) {

// low is the starting index of InOrder Traversal List of BST

//high is the ending index of InOrder Traversal List of BST

if(low<=high) // if list is breakable {

mid=(low+high)/2; // find mid index of list

Insert(&bbst,input[mid]); // insert mid element of list into BBST

RDivideList(low,mid-1); // Recursive call of Left sub list

RDivideList(mid+1,high); // Recursive call of Right sub list

} }

Algorithm 2.4 Dividing In Order Traversal List of BST using Recursion

The time taken by Algorithm 2.4 is as follows:

T(n)=1+2T(n/2)

T(n)=1+2(1+2T(n/4))

T(n)=1+2+22T(n/4)

T(n)=1+2+22(1+2T(n/8))

T(n)=1+2+22+23T(n/8)

T(n)=(1+2+22) + 23T(n/23)

T(n)=(1+2+22+----------+2k-1) + 2kT(n/2k)

T(n)=(2k-1)+2kT(n/2k)

Let n=2k , k=log n

T(n)=n-1+nT(1)

T(n)=2n-1

T(n)=O(n)

Algorithm 2.5 is a recursive algorithm, which inserts elements into BBST. If BBST is empty, then root node is created and element is placed into it, otherwise if element is smaller than data of current node, then recursively left sub list is called, to insert element otherwise if element is greater than data of current node, then recursively right sub list is called, to insert. The process of recursive calling insert algorithm, continues until suitable empty node is found within BBST.

Algorithm Insert(node, ele) {

// node is the starting node of BBST

//ele is the element to be placed into BBST

if(node==NULL) // if node is empty {

node = new Node; // create new node

node->data=ele; // place ele into data field of new node

node->left=NULL; //

node->right=NULL;

} else if(ele<node->data) // if element is smaller than data value of current node

Insert(node->left,ele); // recursive call of left sub tree to insert ele

else if(ele>node->data) // if element is greater than data value of current node

Insert(node->right,ele); // recursive call of right sub tree to insert ele

}

Algorithm 2.5 Inserting elements into BBST using Recursion

The time taken by Algorithm 2.5 is as follows:

T(n)=k+k+--------+k=kn

T(n)=O(n)

Algorithm 2.6 is a recursive algorithm, which displays the elements of BBST using PreOrder Traversal.

Algorithm DisplayBBST(bbst) {

// bbst is the starting node of Balanced Binary Search Tree

if(bbst != NULL) if current node is not empty{

print bbst->data; // display data value of parent node

DisplayBBST(bbst->left); // recursive call of left sub tree to display node values

DisplayBBST(bbst->right);// recursive call of right sub tree to display node values

} }

Algorithm 2.6 Displaying elements of BBST using Recursion

The time taken by Algorithm 2.6 is as follows:

T(n)=1+2T(n/2)

T(n)=1+2(1+2T(n/4))

T(n)=1+2+22T(n/4)

T(n)=1+2+22(1+2T(n/8))

T(n)=1+2+22+23T(n/8)

T(n)=(1+2+22) + 23T(n/23)

T(n)=(1+2+22+----------+2k-1) + 2kT(n/2k)

T(n)=(2k-1)+2kT(n/2k)

Let n=2k , k=log n

T(n)=n-1+nT(1)

T(n)=2n-1

T(n)=O(n)

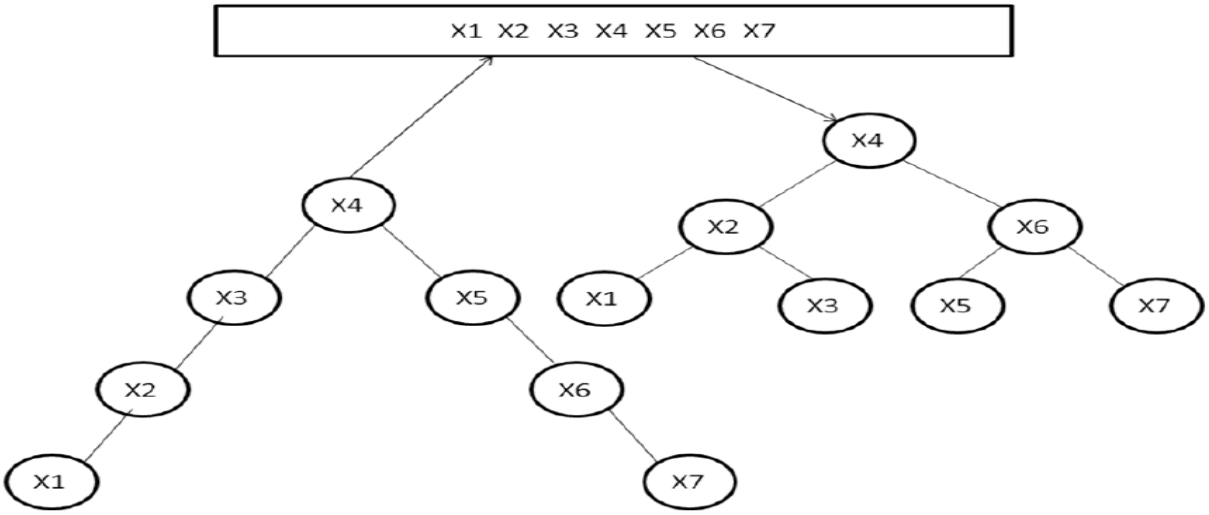

Fig 2.2 BST to BBST using List of nodes

Fig 2.2 shows the relationship of BST, InOrder Traversal List of BST and BBST. In the proposed Algorithms, Binary Search Tree (BST) is taken as input, In Order Traversal List is constructed for the given BST, and it is divided into sub lists based upon the mid index, mid elements are inserted into BBST. In Fig 2.2,BST contains nodes {X4,X3,X5,X2,X6,X1,X7}. For the given BST, InOrder Traversal List {X1,X2,X3,X4,X5,X6,X7} is constructed. This list is divided into left sub list {X1,X2,X3} and right sub list {X5,X6,X7}, mid element X4 is inserted into BBST. Again left sub list is divided into two sub lists {X1} and {X3}, mid element X2 is inserted into BBST. This process continues until all the elements of InOrder Traversal List are inserted into BBST.

International Journal of Novel Research in Computer Science and Software Engineering
Vol. 1, Issue 1, pp: (20-29), Month: September-December 2014, Available at: www.noveltyjournals.com

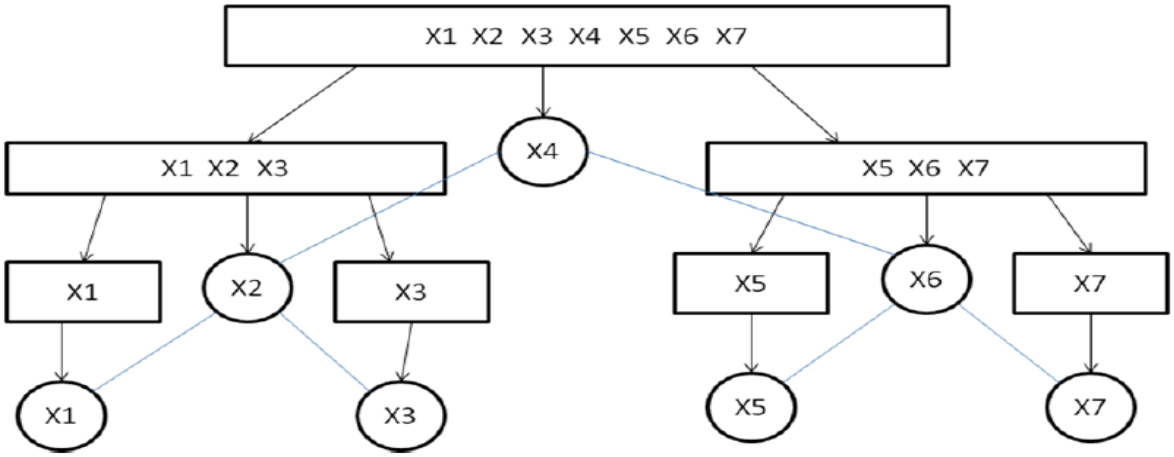

Fig 2.3 Dividing InOrder Traversal List of BST to construct BBST

Fig 2.3 shows the Division of InOrder Traversl List of BST into left sub lists and right sub lists, insertion of mid elements into BBST.

III. MPLEMENTATION OF PROPOSED ALGORITHM USING C

The Proposed Algorithms are implemented by using C Programming Language, which is a robust, structure oriented programming language, which provides different concepts like functions, pointers, structures, arrays and so on to solve complex problems.

Program 3.1

```
/* Implementation of Balanced BST using InOrder Traversal List of BST*/
# include <stdio.h>
# include <conio.h>
struct node // node structure of BBST and BST{
int data; // data field of node
struct node *left,*right; // left and right address fields
};
struct queue // queue structure to store indices of sub lists{
int low,high; // starting and ending indices of list
};
struct queue q[30]; // queue to store indices of sub lists
int rear=0,front=0; // indices to insert into queue and remove from queue
int input[100],k=0; // inorder traversal list of BST
struct node *bbst=NULL; // BBST is empty
void insert(struct node **p,int ele) // to insert elements into BBST {
if((*p)==NULL) // if current node is not empty{
(*p)=(struct node *)malloc(sizeof(struct node)); // create new node
(*p)->data=ele; // place ele into data field of new node
(*p)->left=NULL; // left sub tree of new node is empty
(*p)->right=NULL; // right sub tree of new node is empty
} else if(ele<(*p)->data) // if element is smaller than data of current node
```

```
insert(&((*p)->left),ele); // recursive call of left sub tree of current node to insert ele
else
insert(&((*p)->right),ele); // recursive call of right sub tree of current node to insert ele
}
void ConstructList(struct node *p) // constructing InOrder traversal List of BST {
if(p!=NULL) // if current node is not empty{
ConstructList(p->left); // recursive call of left sub list of current node
input[++k]=p->data; // placing data of current node into Input List
ConstructList(p->right); // recursive call of right sub list of current node
}
}
void DivideList(int low,int high) // Dividing InOrder Traversal List of BST {
int mid;
while(1) {
if(low<=high) // if list is breakable{
mid=(low+high)/2; // find the mid index of list
insert(&bbst,input[mid]); // insert mid element of list into BBST
rear++;
q[rear].low=low; // store starting index of left sub list into queue
q[rear].high=mid-1; // store ending index of left sub list into queue
rear++;
q[rear].low=mid+1; // store starting index of right sub list into queue
q[rear].high=high; // store ending index of right sub list into queue
}
if(front<rear) // if queue is not empty {
front++;
low=q[front].low; //assign starting index of list to low
high=q[front].high; // assign ending index of list to high
}
else break; // if queue becomes empty
}
}
void DisplayBBST(struct node *p) // display the nodes of BBST {
if(p!=NULL) // if current node is not empty{
printf("%d ",p->data); // print data of current node
DisplayBBST(p->left); // recursive call of left sub tree of current node of BBST
DisplayBBST(p->right); // recursive call of right sub tree of current node of BBST
}
}
void ConstructBBST(struct node *bst) // construction of BBST using BST{
```

```
ConstructList(bst); //construct InOrder Traversal list of BST
DivideList(1,k); // divide InOrder Traversal list of BST to construct BBST
printf("\nPreOrder Traversal of BBST:\n");
DisplayBBST(bbst); // displaying nodes of BBST
}
void main() {
int n,ele,i;
struct node *bst=NULL;
clrscr();
printf("\nEnter number of nodes of tree:\n");
scanf("%d",&n);
printf("\nEnter %d nodes of tree:\n",n);
for(i=1;i<=n;i++)
{
scanf("%d",&ele);
insert(&bst,ele);
}
ConstructBBST(bst);
getch(); }
```

Program 3.1 Implementation of Balanced BST using InOrder List of BST

IV. TESTING OF PROPOSED ALGORITHM

Different Test cases are considered to test the result of Program 3.1. Consider the following Test Case in which a BST of 12 nodes {1 2 3 4 5 6 7 8 9 10 11 12 } is considered as input. The program displays the PreOrder Traversal list of BBST, which consists of 12 nodes {6 3 1 2 4 5 9 7 8 11 10 12}

Fig 4.1 screen shot of Converting BST to BBST

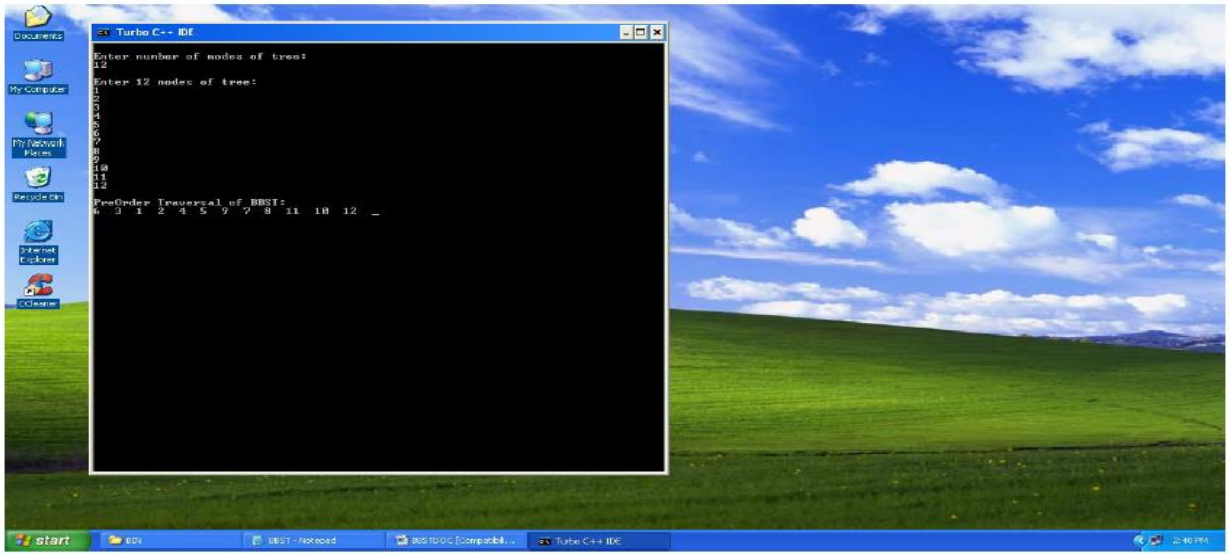

Fig 4.1 screen shot of Converting BST to BBST

International Journal of Novel Research in Computer Science and Software Engineering
Vol. 1, Issue 1, pp: (20-29), Month: September-December 2014, Available at: www.noveltyjournals.com

V. CONCLUSION

We conclude that in the proposed Algorithms, we are considering Binary Search Tree (BST) with finite number of nodes as Input, for the given BST, we are constructing InOrder Traversal List, which is divided into left sub list and right sub list based on the mid index. Mid element is inserted into BBST. Two algorithms are proposed to divide the list into sub lists. In DivideList() algorithm, we are using queue to store indices of left sub list and right sub list, after division of list. This division of lists, insertion of mid elements into BBST continues until queue becomes empty. In RDivideList(), we are using recursion concept to call left sub list and right sub list, mid element is placed into BBST.

REFERENCES

[1] Aho, Alfred V. and Jeffrey D. Ullman [1983]. Data Structures and Algorithms. Addison Wesley, Reading, Massachusetts.

[2] Cormen, Thomas H., Charles E. Leiserson and Ronald L. Rivest [1990]. Introduction to

[3] Algorithms. McGraw-Hill, New York.

[4] Knuth, Donald. E. [1998]. The Art of Computer Programming, Volume 3, Sorting and

[5] Searching. Addison-Wesley, Reading, Massachusetts.