C S Deeraj

CH.SC.U4CSE24106

Week 4

Date – 08/01/2026

Design and Analysis of Algorithms

[23CSE111]

BST Balancing using Rotation Method

avlrotation.c

C: > Users > savit > OneDrive > Desktop > Sem IV CSE > Ws1 haskell > avlrotation.c > main()

```c
//CH.SC.U4CSE24106
#include<stdio.h>
#include<stdlib.h>
struct node{
    int key;
    int height;
    struct node*left;
    struct node*right;
};

int max(int a,int b){
    return a>b?a:b;
}

int height(struct node*n){
    if(n==NULL)
        return 0;
    return n->height;
}

struct node*createnode(int key){
    struct node*newnode=(struct node*)malloc(sizeof(struct node));
    newnode->key=key;
    newnode->left=NULL;
    newnode->right=NULL;
    newnode->height=1;
    return newnode;
```

```c
}
// right rotation
struct node*rightrotate(struct node*y){
    struct node*x=y->left;
    struct node*t2=x->right;

    x->right=y;
    y->left=t2;

    y->height=max(height(y->left),height(y->right))+1;
    x->height=max(height(x->left),height(x->right))+1;

    return x;
}
```

```c
// left rotation
struct node*leftrotate(struct node*x){
    struct node*y=x->right;
    struct node*t2=y->left;

    y->left=x;
    x->right=t2;

    x->height=max(height(x->left),height(x->right))+1;
    y->height=max(height(y->left),height(y->right))+1;

    return y;
}
int getbalance(struct node*n){
    if(n==NULL)
        return 0;
    return height(n->left)-height(n->right);
}

struct node*insert(struct node*root,int key){
    if(root==NULL)
        return createnode(key);

    if(key<root->key)
        root->left=insert(root->left,key);
    else if(key>root->key)
        root->right=insert(root->right,key);
    else
```

C: > Users > savit > OneDrive > Desktop > Sem IV CSE > Ws1 haskell > C avlrotation.c > r

```c
struct node*insert(struct node*root,int key){
    else
        return root;

    root->height=1+max(height(root->left),height(root->right));

    int bal=getbalance(root);

    // ll
    if(bal>1&&key<root->left->key)
        return rightrotate(root);

    // rr
    if(bal<-1&&key>root->right->key)
        return leftrotate(root);

    // lr
    if(bal>1&&key>root->left->key){
        root->left=leftrotate(root->left);
        return rightrotate(root);
    }

    // rl
    if(bal<-1&&key<root->right->key){
        root->right=rightrotate(root->right);
        return leftrotate(root);
    }

    return root;
```

```
    return root;
}

struct node*queue[100];
int front=0,rear=0;

void enqueue(struct node*n){
    queue[rear++]=n;
}

struct node*dequeue(){
    return queue[front++];
}

int isempty(){
    return front==rear;
}

void levelorder(struct node*root){
    if(root==NULL)
        return;

    enqueue(root);

    while(!isempty()){
        int count=rear-front;

        while(count--){
```

```c
        while(count--){
            struct node*cur=dequeue();
            printf("%d ",cur->key);

            if(cur->left!=NULL)
                enqueue(cur->left);
            if(cur->right!=NULL)
                enqueue(cur->right);
        }
        printf("|\n");
    }
}

int main(){
    struct node*root=NULL;

    int arr[]={122,157,110,117,147,111,112,133,149,123,141,151};
    int n=12;

    for(int i=0;i<n;i++)
        root=insert(root,arr[i]);

    printf("AVL Level Order:\n");
    levelorder(root);
    printf("\nCH.SC.U4CSE24106\n");
    return 0;
}
```

Output:

```
Loaded 'C:\Windows\SysWOW64\msvcrt.dll'. Symbols loaded.
AVL Level Order:
122 |
111 147 |
110 117 133 151 |
112 123 141 149 157 |
CH.SC.U4CSE24106
[New Thread 14312.0x2e40]
```

Time Complexity:

The overall time complexity of the program is O(n log n). This is because the AVL tree maintains a balanced height of O(log n) at all times. Each insertion first follows normal BST insertion and then performs height updates and balance checks while returning from recursion, all of which take constant time per node along the path. Since the path length is bounded by the height of the AVL tree, each insertion takes O(log n) time. For n insertions, the total time becomes O(n log n). The level order traversal then visits each node exactly once, performing constant-time queue operations, which takes O(n) time. When combined with insertion, the dominant term remains O(n log n).

Space Complexity:

The space complexity of the program is O(n). This includes memory required to store n nodes of the AVL tree itself. In addition, the recursive insertion process uses a call stack proportional to the height of the tree, which is O(log n) due to AVL balancing. The level order traversal uses a queue that, in the worst case, can store all nodes of the largest level of the tree, which is O(n). Since these are not additive beyond linear growth, the overall space complexity remains O(n).

2. BST balancing using Red black Method

```c
//CH.SC.U4CSE24106
#include<stdio.h>
#include<stdlib.h>

#define red 1
#define black 0

struct node{
    int key;
    int color;
    struct node*left;
    struct node*right;
    struct node*parent;
};
/* create A new red node */
```

```c
/* create A new red node */
struct node*createnode(int key){
    struct node*n=(struct node*)malloc(sizeof(struct node));
    n->key=key;
    n->color=red;
    n->left=NULL;
    n->right=NULL;
    n->parent=NULL;
    return n;
}
/* left rotation */
void leftrotate(struct node**root,struct node*x){
    struct node*y=x->right;
    x->right=y->left;

    if(y->left!=NULL)
        y->left->parent=x;

    y->parent=x->parent;

    if(x->parent==NULL)
        *root=y;
```

```c
    if(x->parent==NULL)
        *root=y;
    else if(x==x->parent->left)
        x->parent->left=y;
    else
        x->parent->right=y;

    y->left=x;
    x->parent=y;
}
/* right rotation */
void rightrotate(struct node**root,struct node*y){
    struct node*x=y->left;
    y->left=x->right;

    if(x->right!=NULL)
        x->right->parent=y;

    x->parent=y->parent;

    if(y->parent==NULL)
        *root=x;
    else if(y==y->parent->left)
        y->parent->left=x;
    else
        y->parent->right=x;

    x->right=y;
    y->parent=x;
}
/* fix red black violations */
void fixinsert(struct node**root,struct node*z){
    while(z!=*root&&z->parent->color==red){
        if(z->parent==z->parent->parent->left){
            struct node*u=z->parent->parent->right;

            if(u!=NULL&&u->color==red){
                z->parent->color=black;
                u->color=black;
                z->parent->parent->color=red;
                z=z->parent->parent;
            }
```

```
            else{
                if(z==z->parent->right){
                    z=z->parent;
                    leftrotate(root,z);
                }
                z->parent->color=black;
                z->parent->parent->color=red;
                rightrotate(root,z->parent->parent);
            }
        }
        else{
            struct node*u=z->parent->parent->left;

            if(u!=NULL&&u->color==red){
                z->parent->color=black;
                u->color=black;
                z->parent->parent->color=red;
                z=z->parent->parent;
            }
```

```
            else{
                if(z==z->parent->left){
                    z=z->parent;
                    rightrotate(root,z);
                }
                z->parent->color=black;
                z->parent->parent->color=red;
                leftrotate(root,z->parent->parent);
            }
        }
    }
    (*root)->color=black;
}

/* bst insert + fix */
void insert(struct node**root,int key){
    struct node*z=createnode(key);
    struct node*y=NULL;
    struct node*x=*root;
```

```c

    while(x!=NULL){
        y=x;
        if(z->key<x->key)
            x=x->left;
        else
            x=x->right;
    }

    z->parent=y;

    if(y==NULL)
        *root=z;
    else if(z->key<y->key)
        y->left=z;
    else
        y->right=z;

    fixinsert(root,z);
}

```

```c
struct node*queue[100];
int front=0,rear=0;
void enqueue(struct node*n){
    queue[rear++]=n;
}
struct node*dequeue(){
    return queue[front++];
}
int isempty(){
    return front==rear;
}
void levelorder(struct node*root){
    if(root==NULL)
        return;

    enqueue(root);

    while(!isempty()){
        int count=rear-front;

        while(count--){
            struct node*cur=dequeue();
```

```c
        while(count--){
            struct node*cur=dequeue();
            printf("%d ",cur->key);

            if(cur->left!=NULL)
                enqueue(cur->left);
            if(cur->right!=NULL)
                enqueue(cur->right);
        }
        printf("|\n");
    }
}

int main(){
    struct node*root=NULL;

    int arr[]={149,110,157,147,117,112,111,133,122,123,151,141};
    int n=12;

    for(int i=0;i<n;i++)
```

```c
    for(int i=0;i<n;i++)
        insert(&root,arr[i]);

    printf("Red Black Tree Level Order:\n");
    levelorder(root);

    return 0;
}
```

Output:

```
Loaded 'C:\Windows\SysWOW64\msvcrt.dll'. Symbols loaded.
Red Black Tree Level Order:
133 |
117 149 |
111 122 147 157 |
110 112 123 141 151 |
The program 'C:\Users\savit\OneDrive\Desktop\Sem IV CSE\Ws1
```

Time Complexity:

The overall time complexity of the Red–Black Tree program is O(n log n). A red–black tree maintains a height of O(log n) by enforcing color properties and performing rotations and recoloring after each insertion. Each insertion first behaves like a normal BST insertion, which takes time proportional to the height of the tree, i.e., O(log n). The fixing process after insertion involves a constant number of recoloring operations and at most a few rotations, all of which take O(1) time. Therefore, for n insertions, the total time complexity becomes O(n log n). The level order traversal visits each node exactly once and performs constant-time queue operations, resulting in O(n) time, which does not change the overall complexity.

Space Complexity:

The space complexity of the Red–Black Tree program is O(n). This is due to the memory required to store n nodes in the tree, where each node contains key, color, and pointer information. The insertion process is iterative, so it does not require additional recursion stack space beyond a constant amount. During level order traversal, a queue is used that can store up to all the nodes of the largest level of the tree, which in the worst case is O(n). Since all extra memory used grows linearly with the number of nodes, the overall space complexity remains O(n).