

C S Deeraj

CH.SC.U4CSE24106

Week 4

Date – 08/01/2026

Design and Analysis of Algorithms

[23CSE111]

BST Balancing using Rotation Method

C avlrotation.c X

```
C: > Users > savit > OneDrive > Desktop > Sem IV CSE > Ws1 haskell > C avlrotation.c > main()
1 //CH.SC.U4CSE24106
2 #include<stdio.h>
3 #include<stdlib.h>
4 struct node{
5     int key;
6     int height;
7     struct node*left;
8     struct node*right;
9 };
10
11 int max(int a,int b){
12     return a>b?a:b;
13 }
14
15 int height(struct node*n){
16     if(n==NULL)
17         return 0;
18     return n->height;
19 }
20
21 struct node*createnode(int key){
22     struct node*newnode=(struct node*)malloc(sizeof(struct node));
23     newnode->key=key;
24     newnode->left=NULL;
25     newnode->right=NULL;
26     newnode->height=1;
27     return newnode;
```

```
28 }
29 // right rotation
30 struct node*rightrotate(struct node*y){
31     struct node*x=y->left;
32     struct node*t2=x->right;
33
34     x->right=y;
35     y->left=t2;
36
37     y->height=max(height(y->left),height(y->right))+1;
38     x->height=max(height(x->left),height(x->right))+1;
39
40     return x;
41 }
```

```
42 // left rotation
43 struct node*leftrotate(struct node*x){
44     struct node*y=x->right;
45     struct node*t2=y->left;
46
47     y->left=x;
48     x->right=t2;
49
50     x->height=max(height(x->left),height(x->right))+1;
51     y->height=max(height(y->left),height(y->right))+1;
52
53     return y;
54 }
55 int getbalance(struct node*n){
56     if(n==NULL)
57         return 0;
58     return height(n->left)-height(n->right);
59 }
60
61 struct node*insert(struct node*root,int key){
62     if(root==NULL)
63         return createnode(key);
64
65     if(key<root->key)
66         root->left=insert(root->left,key);
67     else if(key>root->key)
68         root->right=insert(root->right,key);
69     else
```

```
C: > Users > savit > OneDrive > Desktop > Sem IV CSE > Ws1 haskell > C avlrotation.c > ↻
61     struct node*insert(struct node*root,int key){
62
63         if(key==root->key)
64             return root;
65
66         if(key<root->key)
67             root->left=insert(root->left,key);
68
69         else
70             root->right=insert(root->right,key);
71
72         root->height=1+max(height(root->left),height(root->right));
73
74         int bal=getbalance(root);
75
76         // ll
77         if(bal>1&&key<root->left->key)
78             return rightrightrotate(root);
79
80         // rr
81         if(bal<-1&&key>root->right->key)
82             return leftleftrotate(root);
83
84         // lr
85         if(bal>1&&key>root->left->key){
86             root->left=leftrotate(root->left);
87             return rightrightrotate(root);
88         }
89
90         // rl
91         if(bal<-1&&key<root->right->key){
92             root->right=rightrightrotate(root->right);
93             return leftleftrotate(root);
94         }
95
96     return root;
}
```

```
96     |     return root;
97 }
98
99 struct node*queue[100];
100 int front=0,rear=0;
101
102 void enqueue(struct node*n){
103     |     queue[rear++]=n;
104 }
105
106 struct node*dequeue(){
107     |     return queue[front++];
108 }
109
110 int isempty(){
111     |     return front==rear;
112 }
113
114 void levelorder(struct node*root){
115     |     if(root==NULL)
116     |         |     return;
117
118     |     enqueue(root);
119
120     |     while(!isempty()){
121     |         |     int count=rear-front;
122
123         |         |     while(count--){
```

```
123     while(count--){  
124         struct node*cur=dequeue();  
125         printf("%d ",cur->key);  
126  
127         if(cur->left!=NULL)  
128             enqueue(cur->left);  
129         if(cur->right!=NULL)  
130             enqueue(cur->right);  
131     }  
132     printf("|\n");  
133 }  
134 }  
135  
136 int main(){  
137     struct node*root=NULL;  
138  
139     int arr[]={122,157,110,117,147,111,112,133,149,123,141,151};  
140     int n=12;  
141  
142     for(int i=0;i<n;i++)  
143         root=insert(root,arr[i]);  
144  
145     printf("AVL Level Order:\n");  
146     levelorder(root);  
147     printf("\nCH.SC.U4CSE24106\n");  
148     return 0;  
149 }
```

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

```
1 //CH.SC.U4CSE24106
2 ↘ #include<stdio.h>
3 #include<stdlib.h>
4
5 #define red 1
6 #define black 0
7
8 ↘ struct node{
9     int key;
10    int color;
11    struct node*left;
12    struct node*right;
13    struct node*parent;
14 };
15 /* create A new red node */
```

```
15  /* create A new red node */
16  struct node*createnode(int key){
17      struct node*n=(struct node*)malloc(sizeof(struct node));
18      n->key=key;
19      n->color=red;
20      n->left=NULL;
21      n->right=NULL;
22      n->parent=NULL;
23      return n;
24  }
25  /* left rotation */
26  void leftrotate(struct node**root,struct node*x){
27      struct node*y=x->right;
28      x->right=y->left;
29
30      if(y->left!=NULL)
31          y->left->parent=x;
32
33      y->parent=x->parent;
34
35      if(x->parent==NULL)
36          *root=y;
```

```
55     if(x->parent==NULL)
56     |     *root=y;
57     |     else if(x==x->parent->left)
58     |         x->parent->left=y;
59     |     else
60     |         x->parent->right=y;
61
62     y->left=x;
63     x->parent=y;
64 }
65 /* right rotation */
66 void rightrotate(struct node**root,struct node*y){
67     struct node*x=y->left;
68     y->left=x->right;
69
70     if(x->right!=NULL)
71         x->right->parent=y;
72
73     x->parent=y->parent;
74
75     if(y->parent==NULL)
```

```
56         *root=x;
57     else if(y==y->parent->left)
58         y->parent->left=x;
59     else
60         y->parent->right=x;
61
62     x->right=y;
63     y->parent=x;
64 }
65 /* fix red black violations */
66 void fixinsert(struct node**root,struct node*z){
67     while(z!=*root&&z->parent->color==red){
68         if(z->parent==z->parent->parent->left){
69             struct node*u=z->parent->parent->right;
70
71             if(u!=NULL&&u->color==red){
72                 z->parent->color=black;
73                 u->color=black;
74                 z->parent->parent->color=red;
75                 z=z->parent->parent;
76             }
```

```
77         else{
78             if(z==z->parent->right){
79                 z=z->parent;
80                 leftrotate(root,z);
81             }
82             z->parent->color=black;
83             z->parent->parent->color=red;
84             rightrotate(root,z->parent->parent);
85         }
86     }
87     else{
88         struct node*u=z->parent->parent->left;
89
90         if(u!=NULL&&u->color==red){
91             z->parent->color=black;
92             u->color=black;
93             z->parent->parent->color=red;
94             z=z->parent->parent;
95         }

```

```
96     else{
97         if(z==z->parent->left){
98             z=z->parent;
99             rightrotate(root,z);
100        }
101        z->parent->color=black;
102        z->parent->parent->color=red;
103        leftrotate(root,z->parent->parent);
104    }
105}
106}
107(*root)->color=black;
108}
109
110 /* bst insert + fix */
111 void insert(struct node**root,int key){
112     struct node*z=createnode(key);
113     struct node*y=NULL;
114     struct node*x=*root;
```

```
115
116     while(x!=NULL){
117         y=x;
118         if(z->key<x->key)
119             x=x->left;
120         else
121             x=x->right;
122     }
123
124     z->parent=y;
125
126     if(y==NULL)
127         *root=z;
128     else if(z->key<y->key)
129         y->left=z;
130     else
131         y->right=z;
132
133     fixinsert(root,z);
134 }
135
```

```
● 137     struct node*queue[100];
138     int front=0,rear=0;
139     void enqueue(struct node*n){
140         queue[rear++]=n;
141     }
142     struct node*dequeue(){
143         return queue[front++];
144     }
145     int isempty(){
146         return front==rear;
147     }
148     void levelorder(struct node*root){
149         if(root==NULL)
150             return;
151         enqueue(root);
152         while(!isempty()){
153             int count=rear-front;
154             while(count--){
155                 struct node*cur=dequeue();
```

```
157     while(count--){  
158         struct node*cur=dequeue();  
159         printf("%d ",cur->key);  
160  
161         if(cur->left!=NULL)  
162             enqueue(cur->left);  
163         if(cur->right!=NULL)  
164             enqueue(cur->right);  
165     }  
166     printf("|\n");  
167 }  
168  
169  
170 int main(){  
171     struct node*root=NULL;  
172  
173     int arr[]={149,110,157,147,117,112,111,133,122,123,151,141};  
174     int n=12;  
175  
176     for(int i=0;i<n;i++)  
177         insert(&root,arr[i]);  
178  
179     printf("Red Black Tree Level Order:\n");  
180     levelorder(root);  
181  
182     return 0;  
183 }  
184
```

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 $O(n)$.