HANYANG UNIVERSITY
DIVISION OF COMPUTER SCIENCE

Data Structures

박영준 교수님

Lab6: AVL

AVL

- Self-balancing binary search tree.
- For every node in the tree, the heights of its subtrees differ by at most 1

AVL ADT

AVL

- int RetHeight(BTNode *Node);
  - Return height of both sub tree
- int RetDiffInHeightOfSubTree(BTNode *Node);
  - Return difference of both sub tree

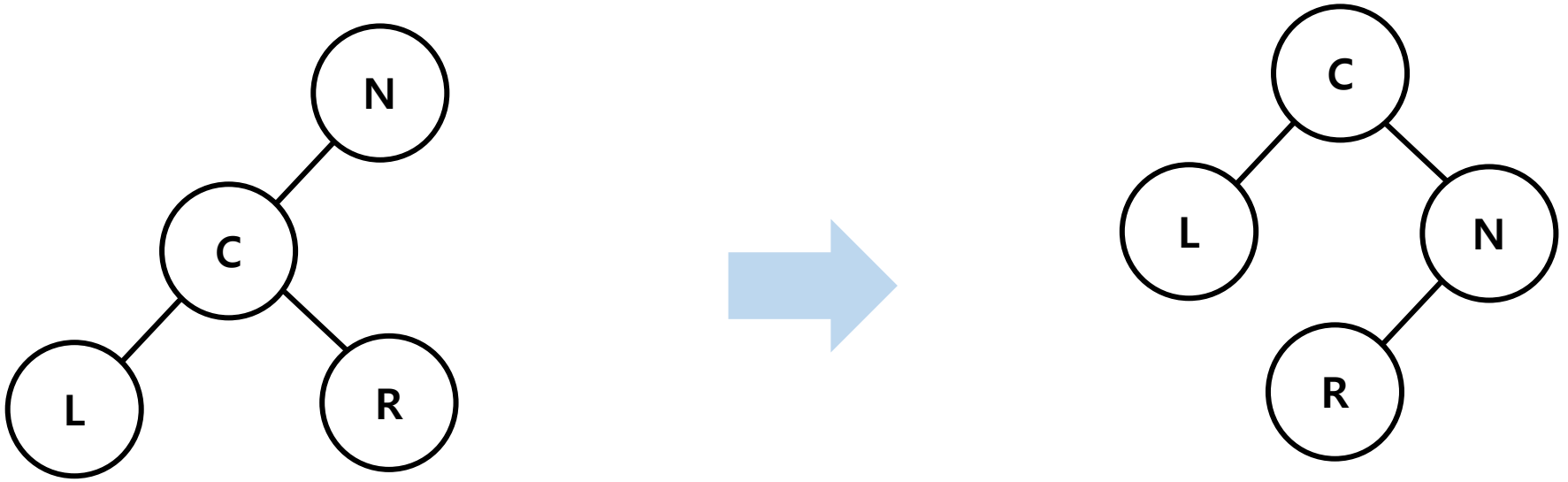
AVL ADT

AVL

- BTNode *RotateLL(BTNode *Node);
  - Balancing LL state of Node
  - Make left child(Child) of node to be parent of Node
  - And Node to be right child of Child
  - If Child had right child, make it to be left child of Node

AVL

LL Rotation



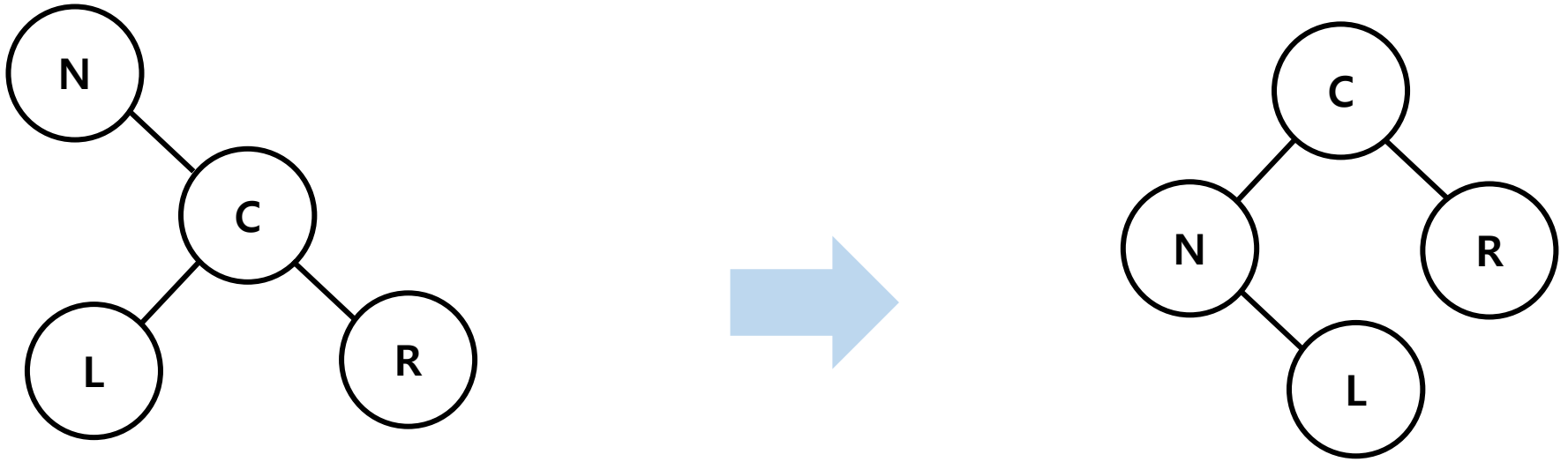
AVL ADT

AVL

- BTNode *RotateRR(BTNode *Node);
  - Balancing RR state of Node
  - Make right child(Child) of node to be parent of Node
  - And Node to be left child of Child
  - If Child had left child, make it to be right child of Node

AVL

RR Rotation



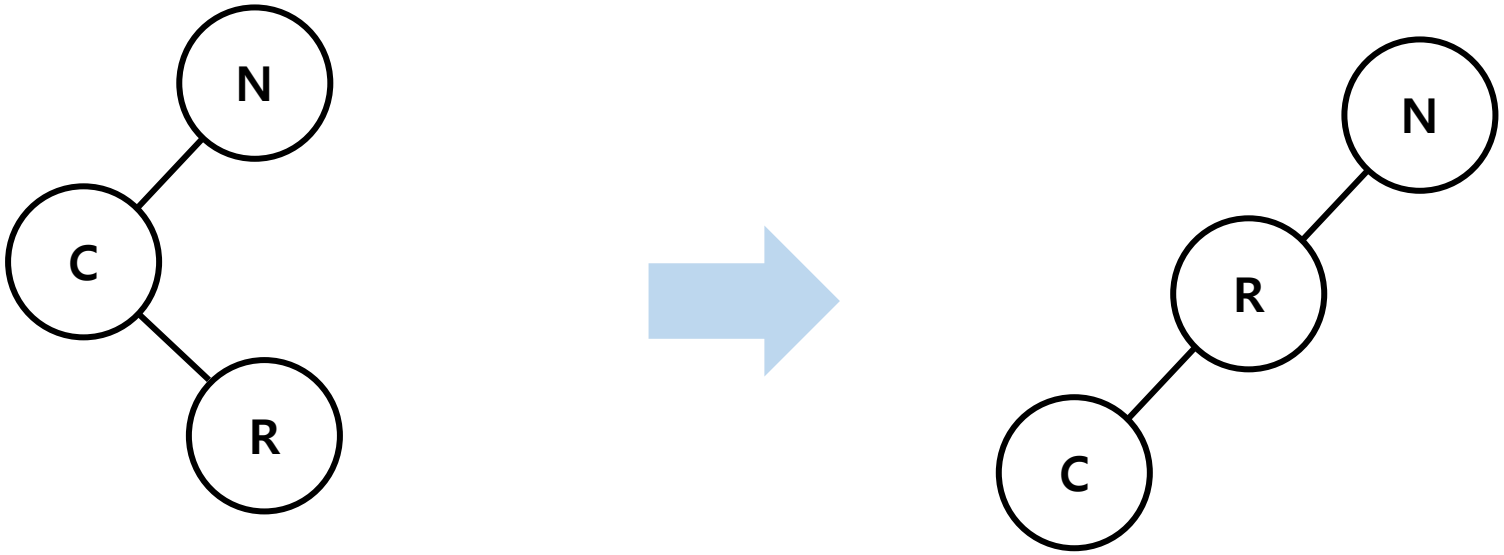
AVL ADT

AVL

- BTNode *RotateLR(BTNode *Node);
  - Balancing LR state of Node
  - First, RR rotate left child(Child) of Node with right child of Child
  - And LL rotate Node

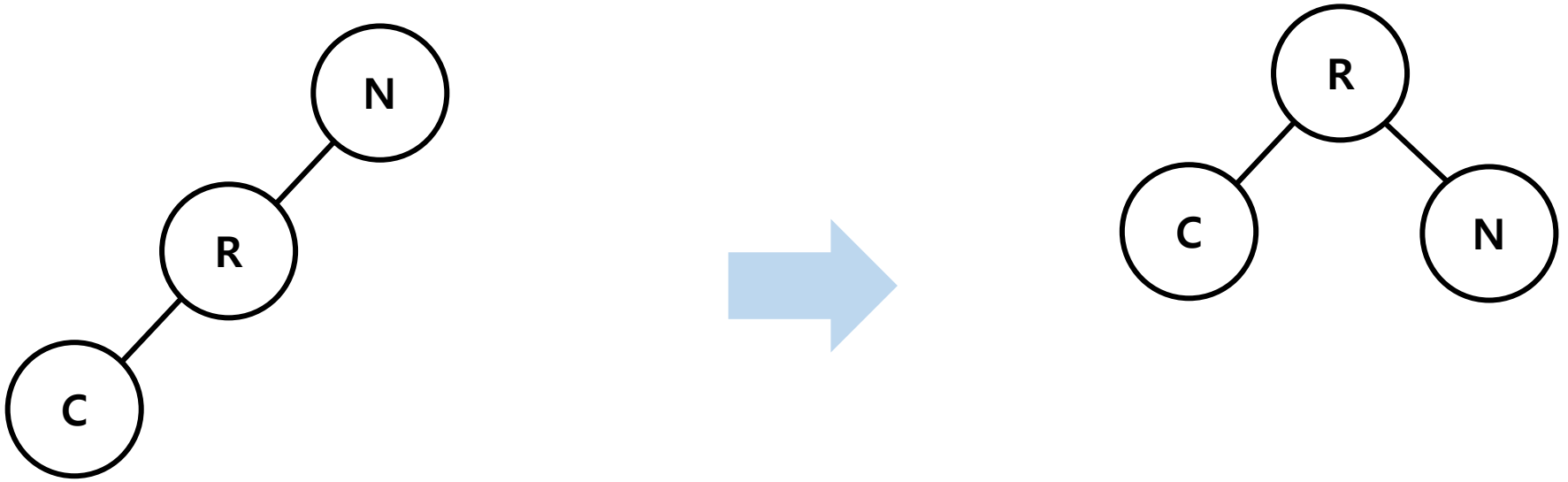
AVL

LR Rotation



AVL

LR Rotation



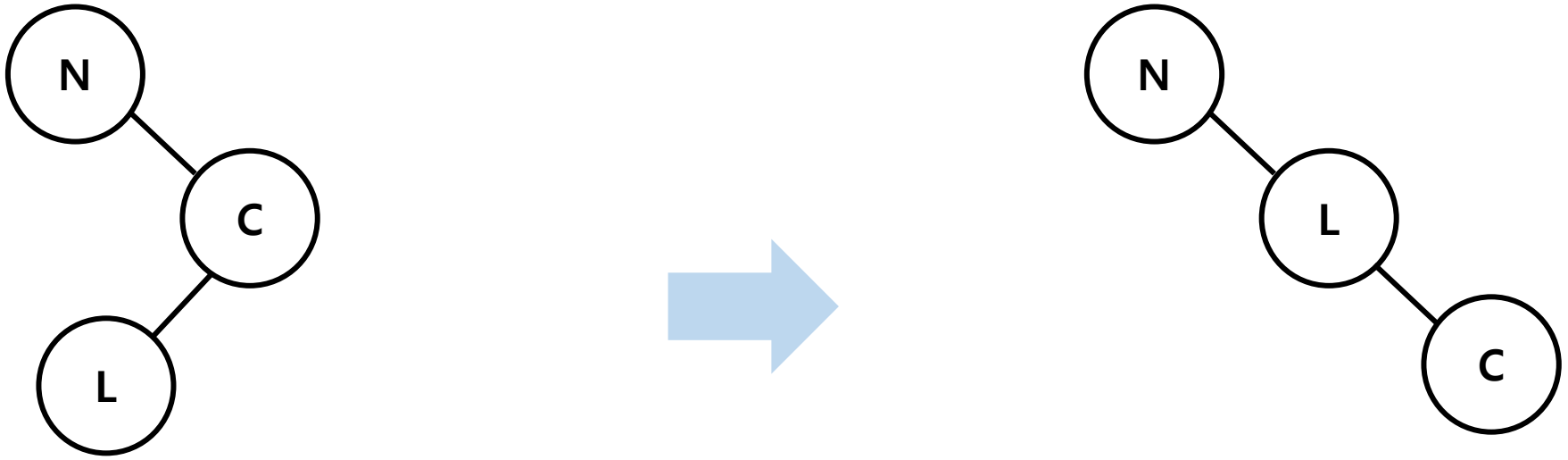
AVL ADT

AVL

- BTNode *RotateRL(BTNode *Node);
  - Balancing RL state of Node
  - First, LL rotate right child(Child) of Node with left child of Child
  - And RR rotate Node

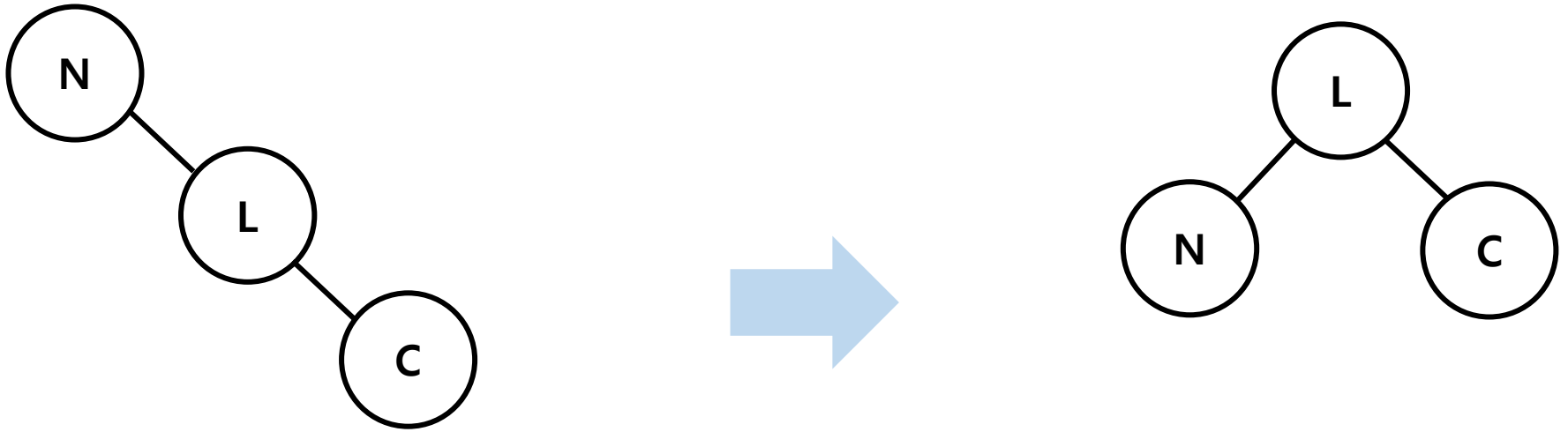
AVL

RL Rotation



AVL

RL Rotation



AVL ADT

AVL

- BTNode *Rebalance(BTNode **Root);
  - Rotate BST if the difference between two subtree is higher than 2
- BTNode *InsertBST(BTNode **Root, DATATYPE Data);
  - Insert Data in Root
  - Rebalancing by comparing heights for each insert

AVL ADT

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

#define COUNT 12

typedef int DATATYPE;

typedef struct BTNode
{
    DATATYPE Data;
    struct BTNode *Left;
    struct BTNode *Right;
} BTNode;

//binary tree
BTNode *MakeBTNode(void);
DATATYPE RetData(BTNode *Node);
void SaveData(BTNode *Node, DATATYPE Data);

BTNode *RetSubTreeLeft(BTNode *Node);
BTNode *RetSubTreeRight(BTNode *Node);

void MakeSubTreeLeft(BTNode *Parent, BTNode *Child);
void MakeSubTreeRight(BTNode *Parent, BTNode *Child);

BTNode *RemoveSubTreeLeft(BTNode *Node);
BTNode *RemoveSubTreeRight(BTNode *Node);

void ChangeSubTreeLeft(BTNode *Parent, BTNode *Child);
void ChangeSubTreeRight(BTNode *Parent, BTNode *Child);

//traversal
void PreorderTraversal(BTNode * Node);
void InorderTraversal(BTNode *Node);
void PostorderTraversal(BTNode *Node);

//BST
void MakeBST(BTNode **Node);

BTNode *InsertBST(BTNode **Root, DATATYPE Data);
BTNode *SearchBST(BTNode *Node, DATATYPE Target);
BTNode *RemoveBST(BTNode **Node, DATATYPE Target);

void PrintAllBST(BTNode *Node);
```

AVL ADT

```
46 //AVL
47 BTNode *Rebalance(BTNode **Root);
48
49 BTNode *RotateLL(BTNode *Node);
50 BTNode *RotateRR(BTNode *Node);
51 BTNode *RotateRL(BTNode *Node);
52 BTNode *RotateLR(BTNode *Node);
53
54 int RetHeight(BTNode *Node);
55
56 int RetDiffInHeightOfSubTree(BTNode *Node);
57
58 //util
59 void Print2D(BTNode *root, int space);
60
```

AVL ADT

```
126 BTNode *MakeBTNode(void)
127 {
128     BTNode *Node = (BTNode*)malloc(sizeof(BTNode));
129     Node->Left = NULL;
130     Node->Right = NULL;
131     return Node;
132 }
133
134 DATATYPE RetData(BTNode *Node)
135 {
136     return Node->Data;
137 }
138
139 void SaveData(BTNode *Node, DATATYPE Data)
140 {
141     Node->Data = Data;
142 }
143
144 BTNode *RetSubTreeLeft(BTNode *Node)
145 {
146     return Node->Left;
147 }
148
149 BTNode *RetSubTreeRight(BTNode *Node)
150 {
151     return Node->Right;
152 }
```

```
154 void MakeSubTreeLeft(BTNode *Parent, BTNode *Child)
155 {
156     //if parent has child
157     if(Parent->Left != NULL)
158     {
159         free(Parent->Left);
160     }
161
162     Parent->Left = Child;
163 }
164
165 void MakeSubTreeRight(BTNode *Parent, BTNode *Child)
166 {
167     //if parent has child
168     if(Parent->Right != NULL)
169     {
170         free(Parent->Right);
171     }
172
173     Parent->Right = Child;
174 }
```

AVL ADT

```
176 void PreorderTraversal(BTNode * Node)
177 {
178     if(Node == NULL)
179     {
180         return;
181     }
182
183     printf("%d ", Node->Data);
184     PreorderTraversal(Node->Left);
185     PreorderTraversal(Node->Right);
186 }
187
188 void InorderTraversal(BTNode *Node)
189 {
190     if(Node == NULL)
191     {
192         return;
193     }
194
195     InorderTraversal(Node->Left);
196     printf("%d ", Node->Data);
197     InorderTraversal(Node->Right);
198 }
199
200 void PostorderTraversal(BTNode *Node)
201 {
202     if(Node == NULL)
203     {
204         return;
205     }
206
207     PostorderTraversal(Node->Left);
208     PostorderTraversal(Node->Right);
209     printf("%d ", Node->Data);
210 }
```

```
212 BTNode *RemoveSubTreeLeft(BTNode *Node)
213 {
214     BTNode *Temp;
215
216     if(Node != NULL)
217     {
218         Temp = Node->Left;
219         Node->Left = NULL;
220     }
221
222     return Temp;
223 }
224
225 BTNode *RemoveSubTreeRight(BTNode *Node)
226 {
227     BTNode *Temp;
228
229     if(Node != NULL)
230     {
231         Temp = Node->Right;
232         Node->Right = NULL;
233     }
234
235     return Temp;
236 }
237
238 void ChangeSubTreeLeft(BTNode *Parent, BTNode *Child)
239 {
240     Parent->Left = Child;
241 }
242
243 void ChangeSubTreeRight(BTNode *Parent, BTNode *Child)
244 {
245     Parent->Right = Child;
246 }
```

AVL ADT

```
248 //BST
249 void MakeBST(BTNode **Node)
250 {
251     *Node = NULL;
252 }
253
254 BTNode *InsertBST(BTNode **Root, DATATYPE Data)
255 {
256     if(*Root == NULL)
257     {
258         *Root = MakeBTNode();
259         SaveData(*Root, Data);
260     }
261     else if(Data < RetData(*Root))
262     {
263         InsertBST(&((*Root)->Left), Data);
264         *Root = Rebalance(Root);
265     }
266     else if(Data > RetData(*Root))
267     {
268         InsertBST(&((*Root)->Right), Data);
269         *Root = Rebalance(Root);
270     }
271     else
272     {
273         //do not allow duplicate data
274         return NULL;
275     }
276
277     return *Root;
278 }
```

```
280 BTNode *SearchBST(BTNode *Node, DATATYPE Target)
281 {
282     BTNode *Current = Node;
283     DATATYPE Data;
284
285     while(Current != NULL)
286     {
287         Data = RetData(Current);
288
289         if(Target == Data)
290         {
291             return Current;
292         }
293         else if(Target < Data)
294         {
295             Current = RetSubTreeLeft(Current);
296         }
297         else
298         {
299             Current = RetSubTreeRight(Current);
300         }
301
302     }
303     return NULL;
304 }
```

AVL ADT

```
306 BTNode *RemoveBST(BTNode **Root, DATATYPE Target)
307 {
308     //create virtual root
309     BTNode *VirtualRoot = MakeBTNode();
310
311     BTNode *Parent = VirtualRoot;
312     BTNode *Current = *Root;
313     BTNode *TargetNode;
314
315     // make root node to be right child of virtual root
316     ChangeSubTreeRight(VirtualRoot, *Root);
317
318     //search target node
319     while(Current != NULL && RetData(Current) != Target)
320     {
321         Parent = Current;
322
323         if(Target < RetData(Current))
324         {
325             Current = RetSubTreeLeft(Current);
326
327         }
328         else
329         {
330             Current = RetSubTreeRight(Current);
331         }
332     }
333
334     //if target not exist
335     if(Current == NULL)
336     {
337         return NULL;
338     }
339
340     TargetNode = Current;
341
342     //if target node is edge node
343     if(RetSubTreeLeft(TargetNode) == NULL && RetSubTreeRight(TargetNode) == NULL)
344     {
345         if(RetSubTreeLeft(Parent) == TargetNode)
346         {
347             RemoveSubTreeLeft(Parent);
348         }
349         else
350         {
351             RemoveSubTreeRight(Parent);
352         }
353     }
354     else if(RetSubTreeLeft(TargetNode) == NULL || RetSubTreeRight(TargetNode) == NULL)
355     {
356         //if target has single child
357         BTNode *ChildofTarget;
358
359         //find target
360         if(RetSubTreeLeft(TargetNode) != NULL)
361         {
362             ChildofTarget = RetSubTreeLeft(TargetNode);
363         }
364         else
365         {
366             ChildofTarget = RetSubTreeRight(TargetNode);
367         }
368
369         //link parent and child of target
370         if(RetSubTreeLeft(Parent) == TargetNode)
371         {
372             ChangeSubTreeLeft(Parent, ChildofTarget);
373         }
374         else
375         {
376             ChangeSubTreeRight(Parent, ChildofTarget);
377         }
378     }
```

AVL ADT

```
379     else
380     {
381         //if target has both child
382         BTNode *MinimumNode = RetSubTreeRight(TargetNode);
383         BTNode *ParentofMinimum = TargetNode;
384 
385         DATATYPE Backup;
386 
387         //find node to replace target node
388         while(RetSubTreeLeft(MinimumNode) != NULL)
389         {
390             ParentofMinimum = MinimumNode;
391             MinimumNode = RetSubTreeLeft(MinimumNode);
392         }
393 
394         //backup target data
395         Backup = RetData(TargetNode);
396         //replace data of target node
397         SaveData(TargetNode, RetData(MinimumNode));
398 
399         //link parent and child of MinimumNode
400         if(RetSubTreeLeft(ParentofMinimum) == MinimumNode)
401         {
402             ChangeSubTreeLeft(ParentofMinimum, RetSubTreeRight(MinimumNode));
403         }
404         else
405         {
406             ChangeSubTreeRight(ParentofMinimum, RetSubTreeRight(MinimumNode));
407         }
408 
409         TargetNode = MinimumNode;
410         SaveData(TargetNode, Backup);
411     }
412 
413     //if target node is root
414     if(RetSubTreeRight(VirtualRoot) != *Root)
415     {
416         *Root = RetSubTreeRight(VirtualRoot);
417     }
418 
419     free(VirtualRoot);
420     return TargetNode;
421 }
```

AVL ADT

```
423 void PrintAllBST(BTNode *Node)
424 {
425     //PreorderTraversal(Node);
426 //  InorderTraversal(Node);
427     //PostorderTraversal(Node);
428     Print2D(Node, 0);
429 }
430
431 void Print2D(BTNode *root, int space)
432 {
433     if(root == NULL)
434     {
435         return;
436     }
437
438     space += COUNT;
439
440     Print2D(root->Right, space);
441
442     printf("\n");
443     for(int i = COUNT; i < space; i++)
444     {
445         printf(" ");
446     }
447     printf("%d\n", root->Data);
448
449     Print2D(root->Left, space);
450 }
451
```

AVL ADT

```
452 //AVL
453 BTNode *Rebalance(BTNode **Root)
454 {
455     int Diff = RetDiffInHeightOfSubTree(*Root);
456
457     if(Diff > 1)
458     {
459         //if left subtree is higher than 2
460         if(RetDiffInHeightOfSubTree(RetSubTreeLeft(*Root)) > 0)
461         {
462             printf("Rotate LL\n");
463             *Root = RotateLL(*Root);
464         }
465         else
466         {
467             printf("Rotate LR\n");
468             *Root = RotateLR(*Root);
469         }
470     }
471
472     if(Diff < -1)
473     {
474         //if right subtree is higher than 2
475         if(RetDiffInHeightOfSubTree(RetSubTreeRight(*Root)) < 0)
476         {
477             printf("Rotate RR\n");
478             *Root = RotateRR(*Root);
479         }
480         else
481         {
482             printf("Rotate RL\n");
483             *Root = RotateRL(*Root);
484         }
485     }
486
487     return *Root;
488 }
```

```
490 BTNode *RotateLL(BTNode *Node)
491 {
492     BTNode *Parent;
493     BTNode *Child;
494
495     Parent = Node;
496     Child = RetSubTreeLeft(Parent);
497
498     ChangeSubTreeLeft(Parent, RetSubTreeRight(Child));
499     ChangeSubTreeRight(Child, Parent);
500
501     return Child;
502 }
503
504 BTNode *RotateRR(BTNode *Node)
505 {
506     BTNode *Parent;
507     BTNode *Child;
508
509     Parent = Node;
510     Child = RetSubTreeRight(Parent);
511
512     ChangeSubTreeRight(Parent, RetSubTreeLeft(Child));
513     ChangeSubTreeLeft(Child, Parent);
514
515     return Child;
516 }
```

AVL ADT

```
518 BTNode *RotateRL(BTNode *Node)
519 {
520     //note, rotate RL
521
522 }
523
524 BTNode *RotateLR(BTNode *Node)
525 {
526     BTNode *Parent;
527     BTNode *Child;
528
529     Parent = Node;
530     Child = RetSubTreeLeft(Parent);
531
532     ChangeSubTreeLeft(Parent, RotateRR(Child));
533
534     return RotateLL(Parent);
535 }
```

```
537 int RetHeight(BTNode *Node)
538 {
539     int HeightOfLeft;
540     int HeightOfRight;
541
542     if(Node == NULL)
543     {
544         return 0;
545     }
546
547     //calculate height of left subtree
548     HeightOfLeft = RetHeight(RetSubTreeLeft(Node));
549
550     //calculate height of right subtree
551     HeightOfRight = RetHeight(RetSubTreeRight(Node));
552
553     if(HeightOfLeft > HeightOfRight)
554     {
555         return HeightOfLeft + 1;
556     }
557     else
558     {
559         return HeightOfRight + 1;
560     }
561 }
562
563 int RetDiffInHeightOfSubTree(BTNode *Node)
564 {
565     int HeightOfLeft;
566     int HeightOfRight;
567
568     if(Node == NULL)
569     {
570         return 0;
571     }
572
573     HeightOfLeft = RetHeight(RetSubTreeLeft(Node));
574     HeightOfRight = RetHeight(RetSubTreeRight(Node));
575
576     return HeightOfLeft - HeightOfRight;
577 }
```

Lab6:AVL

- Submit on GitLab
- Complete Rotate RL function
- Create Lab6 directory on your own GitLab project
- Submit file : source_code(c only, run on linux)
- Filename : StudentID_lab6.c
- Input file : no