1 AVL tree

If for each node difference between left and right subtree height is less or equal than 1, such binary tree is called *AVL tree*. For n nodes, an AVL tree has height of $1.44 \lg n$. For that reason, finding, inserting or deleting node is of $O(\log n)$ complexity. The presented algorithms assume that all keys stored in the tree are different.

The following conventions for the tree T with n nodes are used: $r_t(T)$ – T's root, $l(x)$ – left child of the node x, $r(x)$ – x's right child, $p(x)$ – x's parent, $b(x)$ – balance factor of the node x (i.e. difference between left and right subtree), $k(x)$ – key stored in node x. If node x has no left and/or right child, then $l(x)$ and/or $r(x)$ is null.

1.1 Finding node

Finding a node with a given key K starts from the root as current node. K is compared with a key of each current node. If it is less than it's value, it continues within left subtree; if it's greater, then proceeds within right subtree.

Complexity: $O(\lg n)$
Input: key K to find
Output: node with key K or null (if K is not in T or tree is empty)

```
find(K)
  x = r_t(T)
  if x = null
    return null
  while x ≠ null
    if K < k(x)
      x = l(x)
    else if K > k(x)
      x = r(x)
    else
      return x
  return null
```

To find a predecessor of a given node x, one should get the most right descendant of x's left child. From this definition, it follows that predecessor has no right child (otherwise, that child would be the predecessor).

Complexity: $O(\lg n)$
Input: node x to find predecessor
Output: predecessor node or null (if x is null or has no predecessor or tree is empty)

```
predecessor(x)
  if x = null or r_t(T) = null or l(x) = null
    return null
  x_l = l(x)
  while x_l ≠ null
    x_l = r(x_l)
  return x_l
```

To find a successor of a given node x, one should get the most left descendant of x's right child.

From this definition, it follows that successor has no left child (otherwise, that child would be the successor).

Complexity: $O(\lg n)$
Input: node x to find successor
Output: successor node or null (if x is null or has no successor or tree is empty)
$successor(x)$
  if $x =$ **null or** $r_t(T) =$ **null or** $r(x) =$ **null**
    return null
  $x_r = r(x)$
  while $x_r \neq$ **null**
    $x_r = l(x_r)$
  return x_r

One can easily check if a node is descendant of an another node.

Complexity: $O(\lg n)$
Input: ancestor a and descendant d
Output: true or false
$is\_left\_descendant(a, d)$
  while $p(d) \neq a$
    $d = p(d)$
  if $d = l(a)$
    return true
  else
    return false

1.2 Rotations

Rotations reconnect nodes as described below. There are no changes on balance factors, they are fixed in the appropriate operations.

Left rotation reconnects nodes, such that rotated node x get it's right child for the parent and left child of $r(x)$ becomes right child of x. Node 2 in the following figure is rotated to the left:

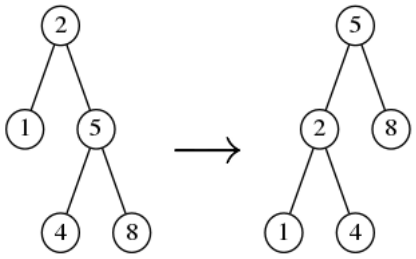

Figure 1: left rotation of node 2

Complexity: $O(1)$
Input: node x to rotate
Output: none
$rotate\_left(x)$
  $x_p = p(x)$

$x_r = r(x)$
$x_{rl} = l(r(x))$
$p(x_r) = x_p$
$p(x) = x_r$
if $x_{rl} \neq$ **null**
  $p(x_{rl}) = x$
$r(x) = x_{rl}$
$l(x_r) = x$
if $x_p \neq$ **null**
  if $x = l(x_p)$
    $l(x_p) = x_r$
  else if $x = r(x_p)$
    $r(x_p) = x_r$
if $r_t(T) = x$
  $r_t(T) = p(x)$

Right rotation reconnects nodes, such that rotated node x get it's left child for the parent and right child of $l(x)$ becomes left child of x. Node 5 in the following figure is rotated to the right:

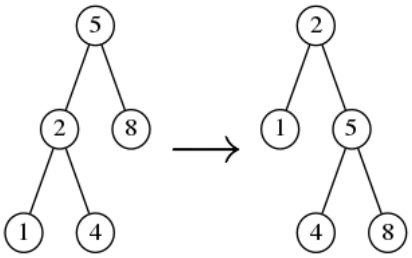

Figure 2: right rotation of node 5

Complexity: $O(1)$
Input: node x to rotate
Output: none
rotate_right(x)
  $x_p = p(x)$
  $x_l = l(x)$
  $x_{lr} = r(l(x))$
  $p(x_l) = x_p$
  $p(x) = x_l$
  if $x_{lr} \neq$ **null**
    $p(x_{lr}) = x$
  $l(x) = x_{lr}$
  $r(x_l) = x$
  if $x_p \neq$ **null**
    if $x = l(x_p)$
      $l(x_p) = x_l$
    else if $x = r(x_p)$
      $r(x_p) = x_l$
  if $r_t(T) = x$
    $r_t(T) = p(x)$

1.3 Inserting node

Inserting node is to put the new key K into tree T by going to the left subtree if K is less than key of the current node, and to the right if K is greater than key of the current node. When a node is inserted, balance factors of some of the traversed nodes can be changed. For that reason, those nodes have to be rebalanced.

While searching correct place to insert new node, last node P with non-zero balance is memorized. If such node does not exist, then no balancing is necessary after inserting the new node. Subtree at P can become corrupted after inserting new node and no other tree except this one can be corrupted. Balances of all nodes from the new one until P are modified. Rotations are made in constant time, so total time for inserting is $O(\lg n)$.

Let A be the last node with non-zero balance, let the new node be inserted into left subtree of A, let be $B = l(A), C = l(B), D = r(B), X = r(A)$ and $h()$ function which returns height of a subtree.

First case is when inserted node is left descendant of B. Then, $b(A) = -2, b(B) = -1$, so $h(D) = h(X), h(C) = h(D)+1 \Rightarrow h(C) = h(X)+1$. If A is right rotated, then B is parent of A, D and X are children of A. It follows that $b(A) = 0$ since $h(D) = h(X)$ and $b(B) = 0$ because $h(C) = h(X) + 1$.

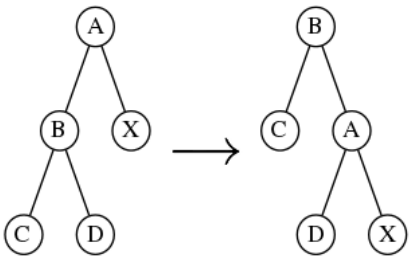

Figure 3: L1 case fixed with $r(A)$

Second, third and fourth case are when inserted node is right descendant of B. Three possibilities are available: $b(A) = -2, b(B) = +1, b(D) = -1$, $b(A) = -2, b(B) = +1, b(D) = +1$ or $b(A) = -2, b(B) = +1, b(D) = 0$, depending of where the node inserted (left or right subtree of D). Denote $D_l = l(D), D_r = r(D)$, and consider the second case $b(D) = -1$. It follows that $b(B) = 1 \Rightarrow h(D) = h(C) + 1, b(D) = -1 \Rightarrow h(D_l) = h(D_r) + 1, h(D) = h(D_l) + 1$, so $h(D_l) = h(C)$. Also, $h(X) = h(B) - 2 = h(D) - 1 = h(D_l) = h(D_r) + 1$. Therefore, $h(D_l) = h(C) \Rightarrow b(B) = 0; h(X) = h(D_r) + 1 \Rightarrow b(A) = +1; h(D_l) = h(X) \Rightarrow b(D) = 0$.

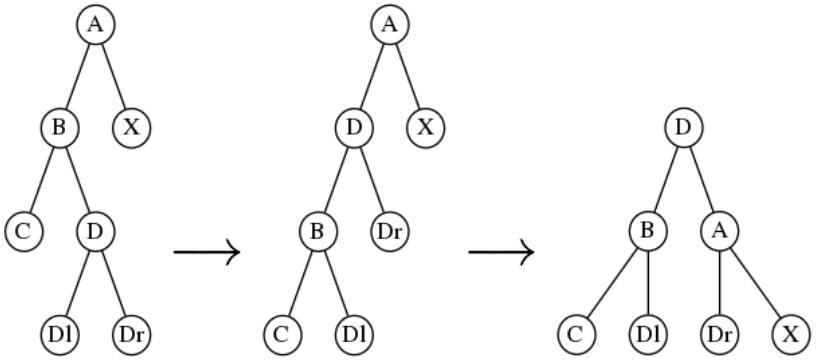

Figure 4: L2, L3, L4 cases fixed with $l(B), r(A)$

For a third case, rotations are the same and calculus is similar: $h(C) = h(D_r) = h(D_l) + 1, h(X) = h(D_r) \Rightarrow b(B) = -1, b(A) = 0, b(D) = 0$. Fourth case is same as this one. Let's write down these cases symbolically:

L1 $b(A) = -2, b(B) = -1, r(A) \Rightarrow b(A) = 0, b(B) = 0$
L2 $b(A) = -2, b(B) = +1, b(D) = -1 \Rightarrow l(B), r(A) \Rightarrow b(A) = +1, b(B) = 0, b(D) = 0$
L3 $b(A) = -2, b(B) = +1, b(D) = +1 \Rightarrow l(B), r(A) \Rightarrow b(A) = 0, b(B) = -1, b(D) = 0$
L4 $b(A) = -2, b(B) = +1, b(D) = 0 \Rightarrow l(B), r(A) \Rightarrow b(A) = 0, b(B) = 0, b(D) = 0$

Symmetric cases come when inserted node is in the right subtree of A.

R1 $b(A) = +2, b(B) = +1 \Rightarrow l(A) \Rightarrow b(A) = 0, b(B) = 0$
R2 $b(A) = +2, b(B) = -1, b(D) = +1 \Rightarrow r(B), l(A) \Rightarrow b(A) = -1, b(B) = 0, b(D) = 0$
R3 $b(A) = +2, b(B) = -1, b(D) = -1 \Rightarrow r(B), l(A) \Rightarrow b(A) = 0, b(B) = +1, b(D) = 0$
R4 $b(A) = +2, b(B) = -1, b(D) = 0 \Rightarrow r(B), l(A) \Rightarrow b(A) = 0, b(B) = 0, b(D) = 0$

The following functions perform cases L1-L3, R1-R3.

Complexity: $O(1)$
Input: subtree's root x to perform case L1
Output: none
$case\_l1(x)$
  $A = x, B = l(A)$
  $rotate\_right(A)$
  $b(A) = 0, b(B) = 0$

Complexity: $O(1)$
Input: subtree's root x to perform case L2
Output: none
$case\_l2(x)$
  $A = x, B = l(A), D = r(B)$
  $rotate\_left(B)$
  $rotate\_right(A)$
  $b(A) = +1, b(B) = 0, b(D) = 0$

Complexity: $O(1)$
Input: subtree's root x to perform case L3
Output: none
$case\_l3(x)$
  $A = x, B = l(A), D = r(B)$
  $rotate\_left(B)$
  $rotate\_right(A)$
  $b(A) = 0, b(B) = -1, b(D) = 0$

Complexity: $O(1)$
Input: subtree's root x to perform case L4
Output: none
$case\_l4(x)$
  $A = x, B = l(A), D = r(B)$
  $rotate\_left(B)$
  $rotate\_right(A)$

$b(A) = 0, b(B) = 0, b(D) = 0$

Complexity: $O(1)$
Input: subtree's root x to perform case R1
Output: none
$case\_r1(x)$
$A = x, B = r(A)$
$rotate\_left(A)$
$b(A) = 0, b(B) = 0$

Complexity: $O(1)$
Input: subtree's root x to perform case R2
Output: none
$case\_r2(x)$
$A = x, B = r(A), D = l(B)$
$rotate\_right(B)$
$rotate\_left(A)$
$b(A) = -1, b(B) = 0, b(D) = 0$

Complexity: $O(1)$
Input: subtree's root x to perform case R3
Output: none
$case\_r3(x)$
$A = x, B = r(A), D = l(B)$
$rotate\_right(B)$
$rotate\_left(A)$
$b(A) = 0, b(B) = +1, b(D) = 0$

Complexity: $O(1)$
Input: subtree's root x to perform case R4
Output: none
$case\_r4(x)$
$A = x, B = r(A), D = l(B)$
$rotate\_right(B)$
$rotate\_left(A)$
$b(A) = 0, b(B) = 0, b(D) = 0$

Inserting searches for an appropriate leaf node to put the key K into one of it's children. Then balance factors of the traversed nodes are fixed.

Complexity: $O(\lg n)$
Input: key K to insert
Output: none
$insert(K)$
new z
$k(z) = K$
{*empty tree is trivial case*}
if $r_t(T) =$ **null**
$r_t(T) = z$
return
{*not empty tree*}

```
c = r_t(T)
P = null {last ancestor with non-zero balance}
{insert z into an empty place}
while true
   if b(c) ≠ 0
      P = c
   if K < k(c)
      if l(c) = null
         l(c) = z
         p(z) = c
         break
      c = l(c)
   else
      if r(c) = null
         r(c) = z
         p(z) = c
         break
      c = r(c)
if P = null or b(P) = 0 {just modify balances}
   c = z
    do
      if c = l(p(c))
         b(p(c)) = b(p(c)) − 1
      else
         b(p(c)) = b(p(c)) + 1
      c = p(c)
   while c ≠ r_t(T)
   return
{modify balances from z to P}
c = z
 do
   if c = l(p(c))
      b(p(c)) = b(p(c)) − 1
   else
      b(p(c)) = b(p(c)) + 1
   c = p(c)
while c ≠ P
{fix balance factors}
if is_left_descendant(P, c) = true
   {node inserted to the left}
   A = P, B = l(P), D = r(B)
   if b(A) = −2 and b(B) = −1
      case_l1(A)
   else if b(A) = −2 and b(B) = +1 and b(D) = −1
      case_l2(A)
   else if b(A) = −2 and b(B) = +1 and b(D) = +1
      case_l3(A)
   else if b(A) = −2 and b(B) = +1 and b(D) = 0
      case_l4(A)
else
   {node inserted to the right}
   A = P, B = r(P), D = l(B)
```

if $b(A) = +2$ **and** $b(B) = +1$
  $case\_r1(A)$
else if $b(A) = +2$ **and** $b(B) = -1$ **and** $b(D) = +1$
  $case\_r2(A)$
else if $b(A) = +2$ **and** $b(B) = -1$ **and** $b(D) = -1$
  $case\_r3(A)$
else if $b(A) = +2$ **and** $b(B) = -1$ **and** $b(D) = 0$
  $case\_r4(A)$

1.4 Deleting node

When a node is deleted, heights of subtrees containing that node may be changed. For that reason, rebalancing has to be performed of all nodes from a deleted one until the root. Deleting Z if both children are null is removing it and checking all parents for balances. If some of the Z's children isn't null, then deleting it is replacing it with predecessor or successor node (call it N). N's parent A takes N's single child B as a new child instead of N, Z is replaced with N. The procedure is shown on the figure 5.

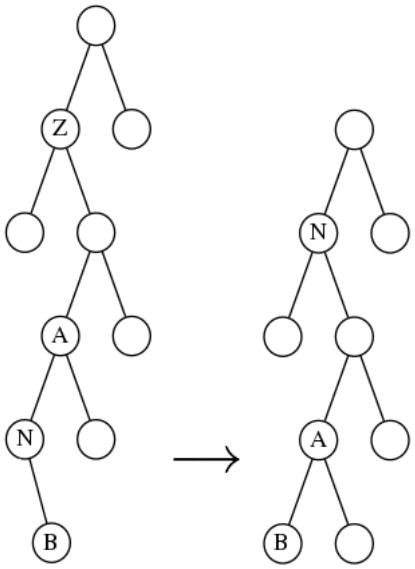

Figure 5: Deleting node Z by replacing it with successor N

Nodes starting from A should be checked for balances and rotated if necessary. If A's height has not changed (balance is 0), the deleting procedure ends; otherwise, A becomes A's parent and procedure is repeated. There are three cases on deleting:

D1 $b(A) = 0$, after deleting $b(A) = \pm 1$ and height of A-tree is not changed, so the deleting procedure is ended.

D2 $b(A) = \pm 1$, after deleting $b(A) = 0$, so there's no need for rotations; but height of trees containing A is changed, so procedure of balancing continues on parent of A.

D3 $b(A) = \pm 1$, after deleting $b(A) = \pm 2$, so rotation are made; height of trees containing A is changed, so procedure of balancing continues on parent of A. L1-L5 and R1-R5 cases are possible here.

Additional cases on deleting are:

L5 $b(A) = -2, b(B) = 0 \Rightarrow r(A) \Rightarrow b(A) = -1, b(B) = +1$
R5 $b(A) = +2, b(B) = 0 \Rightarrow l(A) \Rightarrow b(A) = +1, b(B) = -1$

Complexity: $O(1)$
Input: subtree's root x to perform case L5
Output: none
$case\_l5(x)$
  $A = x, B = l(A)$
  $rotate\_right(A)$
  $b(A) = -1, b(B) = +1$

Complexity: $O(1)$
Input: subtree's root x to perform case R5
Output: none
$case\_r5(x)$
  $A = x, B = r(A)$
  $rotate\_left(A)$
  $b(A) = +1, b(B) = -1$

Rotations are made in $O(\lg n)$ time, so as finding node to delete, so total time for deleting is $O(\lg n)$.

Complexity: $O(\lg n)$ where n is number of elements in the tree
Input: key K to delete
Output: none
$remove(K)$
  if $r_t(T) =$ **null**
    return
  $Z = find(K)$
  if $Z =$ **null**
    return
  if $Z = r_t(T)$
    delete $r_t(T)$
    return
  $N_s = successor(Z), N_p = predecessor(Z)$
  $A =$ **null** {*parent of* N_s *or* N_p }
  if $N_s =$ **null and** $N_p =$ **null**
    $A = p(Z)$
    if $Z = l(A)$
      $b(A) = b(A) + 1$
      $l(A) =$ **null**
    else if $Z = r(A)$
      $b(A) = b(A) - 1$
      $r(A) =$ **null**
    delete Z
  else if $N_s \neq$ **null**
    $k(Z) = k(N_s)$
    $A = p(N_s)$ {*could be also* $A = p(Z)$ }
    if $N_s = l(A)$ {*successor is not sibling of* Z}
      if $r(N_s) \neq$ **null** {*connect* A *with single child (if exists)*}
        $l(A) = r(N_s)$
        $p(r(N_s)) = A$
      else
        delete $l(A)$

```
      b(A) = b(A) + 1
    else if N_s = r(A) {successor is sibling of Z}
      if r(N_s) ≠ null
        r(A) = r(N_s)
        p(r(N_s)) = A
      else
        delete r(A)
      b(A) = b(A) − 1
    delete N_s
  else if N_p ≠ null
    k(Z) = k(N_p)
    A = p(N_p) {could be also A = p(Z) }
    if N_p = r(A) {successor is not sibling of Z}
      if l(N_p) ≠ null {connect A with single child (if exists)}
        r(A) = l(N_p)
        p(l(N_p)) = A
      else
        delete r(A)
      b(A) = b(A) − 1
    else if N_p = l(A) {successor is sibling of Z}
      if l(N_p) ≠ null
        l(A) = l(N_p)
        p(l(N_p)) = A
      else
        delete l(A)
      b(A) = b(A) + 1
    delete N_p
  {correct balances along the tree starting from parent}
  while A ≠ null
    if b(A) = ±1 {case D1}
      break
    else if b(A) = 0 {case D2}
      if p(A) ≠ null
        if A = l(p(A))
          b(p(A)) = b(p(A)) + 1
        else if A = r(p(A))
          b(p(A)) = b(p(A)) − 1
    else if b(A) = +2 {cases R1 - R5}
      B = r(A)
      if b(B) = +1
        case_r1(A)
      else if b(B) = −1
        D = l(B)
        if b(D) = +1
          case_r2(A)
        else if b(D) = −1
          case_r3(A)
        else if b(D) = 0
          case_r4(A)
      else if b(B) = 0
          case_r5(A)
    else if b(A) = −2 {cases L1 - L5}
```

```
    B = l(A)
    if b(B) = −1
      case_l1(A)
    else if b(B) = +1
      D = r(B)
      if b(D) = −1
        case_l2(A)
      else if b(D) = +1
        case_l3(A)
      else if b(D) = 0
        case_l4(A)
    else if b(B) = 0
        case_l5(A)
A = p(A)
```

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: Introduction to Algorithms; Second Edition

[2] Miodrag Zivkovic: Algoritmi

[3] Robert Sedgewick: Algorithms