1 Splay tree

Motivation for this data structure is to have binary tree which performs rotations when nodes is accessed. Operations of interest are finding, inserting and deleting key, joining two trees and splitting a tree into at given key into two subtrees.

1.1 Definition

Let T be a binary tree with root $r(T)$; for each node $x \in T$ let $c_l(x), c_r(x), p(x)$ be the pointers to its left, right children and parent, respectively. *Splaying* tree T of size n at node x is sequence of the following *splay steps* until x becomes the root of T:

zig If $p(x)$ is the root, rotate the edge $(x, p(x))$, and finish the sequence.

zig-zig If $p(x)$ is not the root and x and $p(x)$ are both left/right children of their respective parents, rotate the edge $(p(x), p(p(x)))$ and then rotate $(x, p(x))$.

zig-zag If $p(x)$ is not the root, x is left child of $p(x)$ and $p(x)$ is right child of $p(p(x))$, or vice versa (x is right child of $p(x)$ and $p(x)$ is left child of $p(p(x))$), then rotate $(x, p(x))$ and then rotate x with the new $p(x)$ (which was $p(p(x))$ before this step).

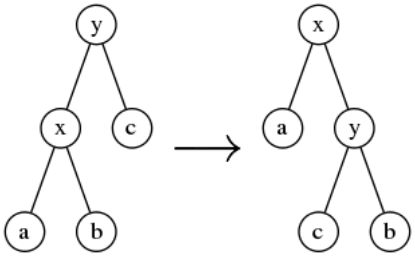

Figure 1: Zig operation on x

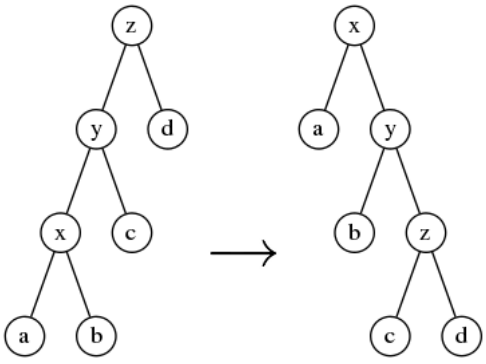

Figure 2: Zig-zig operation on x

Binary search tree T is *splay tree* if for each operation (searching, inserting, deleting) on node x an additional splaying on x is performed.

In the pseudo code, root of a (sub)tree T is denoted with $root(T)$, left and parent children with $child\_l(x)$ and $child\_r(x)$, where $x \in T$. Parent of $x \in T$ is $parent(x)$, key is read by $key(x)$.

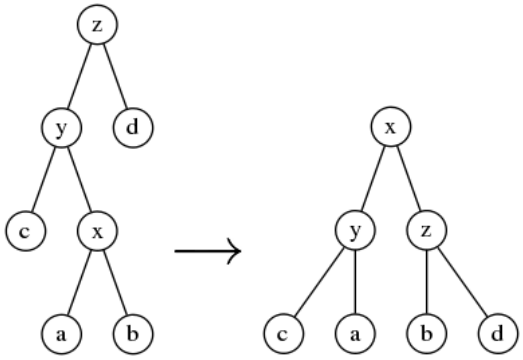

Figure 3: Zig-zag operation on x

1.2 Rotation, linking, assembling

Rotations of nodes are performed as for AVL trees.

If a key K is searched in a tree T, then few subtrees can be recognized. *Left tree* L is a tree containing all nodes from T less than K. *Right tree* R is defined similarly. *Middle tree* is a subtree of T rooted at the current node reached during the search.

If node x is right child of its parent, *left linking* moves x to be the most right child of L (so it becomes $\max(L)$). If node x is left child of its parent, *right linking* moves x to be the most left child of R (so it becomes $\min(R)$).

For a left and right trees L and R, and node x with children $a = c_l(x), b = c_r(x)$, *assembling* creates single tree such that $c_l(x) = L, c_r(x) = R, c_r(L) = a, c_l(R) = b$.

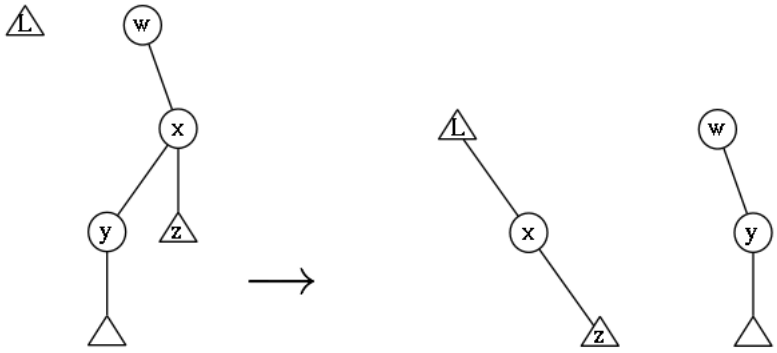

Figure 4: Left linking of node x

Input: node x to link with the left tree L
Output: x left linked to L
Worst Case Complexity: $O(1)$
link_left(L, x)
  $y =$ child_l(x)
  $w =$ parent(x)
  parent$(y) = w$
  child_r$(w) = y$
  parent$(x) =$ root(L)
  child_r(root$(L)) = x$

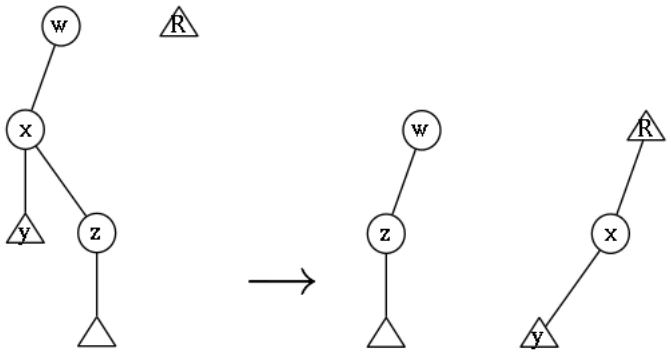

Figure 5: Right linking of node x

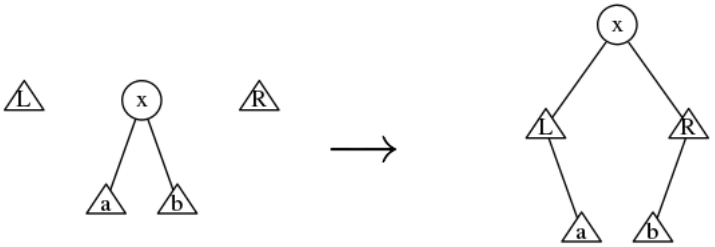

Figure 6: Assembling of node x

Input: node x to link with the right tree R
Output: x right linked to R
Worst Case Complexity: $O(1)$
link_right(R, x)
  $z =$ child_r(x)
  $w =$ parent(x)
  parent$(z) = w$
  child_l$(w) = z$
  parent$(x) =$ root(R)
  child_l(root$(R)) = x$

Input: node x to assemble with the left and right trees L and R
Output: x, L, R assembled
Worst Case Complexity: $O(1)$
assemble(x, L, R)
  $a =$ child_l(x)
  $b =$ child_r(x)
  child_l$(x) =$ root(L)
  child_r$(x) =$ root(R)
  parent(root$(L)) = x$
  parent(root$(R)) = x$
  child_r(root$(L)) = a$
  child_l(root$(R)) = b$
  parent$(a) =$ root(L)
  parent$(b) =$ root(R)

1.3 Splaying

Splaying as defined at the start assumes that it starts from a node x and goes until root is reached. That's bottom-up splaying; it's appropriate when direct access to the node is available.

Input: node x to splay in bottom-up fashion
Output: tree splayed at x
Worst Case Complexity: $O(\lg n)$

```
splay_up_1(x)
  while parent(x) ≠ null
    if x = child_l(parent(x))
      if parent(parent(x)) = null {zig}
        rotate_right(parent(x))
      else if parent(x) = child_l(parent(parent(x))) {zig-zig}
        rotate_right(parent(parent(x)))
        rotate_right(parent(x))
      else if parent(x) = child_r(parent(parent(x))) {zig-zag}
        rotate_right(parent(x))
        rotate_left(parent(x))
    else if x = child_r(parent(x))
      if parent(parent(x)) = null {zig}
        rotate_left(parent(x))
      else if parent(x) = child_r(parent(parent(x))) {zig-zig}
        rotate_left(parent(parent(x)))
        rotate_left(parent(x))
      else if parent(x) = child_l(parent(parent(x))) {zig-zag}
        rotate_left(parent(x))
        rotate_right(parent(x))
```

All three $x = \text{child\_l}(\text{parent}(x))$ cases perform right rotation. If right rotation is done by default, then the remaining left rotation of zig-zag case can be delayed to the zig step of $x = \text{child\_r}(\text{parent}(x))$ case. So, the function above can be simplified:

Input: node x to splay in bottom-up fashion
Output: tree splayed at x
Worst Case Complexity: $O(\lg n)$

```
splay_up_2(x)
  while parent(x) ≠ null
    if x = child_l(parent(x))
      if parent(x) = child_l(parent(parent(x)))
        rotate_right(parent(parent(x)))
      rotate_right(parent(x))
    else if x = child_r(parent(x))
      if parent(x) = child_r(parent(parent(x)))
        rotate_left(parent(parent(x)))
      rotate_left(parent(x))
```

In some cases, splaying during traversing the tree T can be performed in a more efficient way. Top-down splaying starts from a node x and executes splaying steps until a node t with key K is reached (which is the accessed node). All accessed nodes on the path are classified into left or right subtree; x is moving toward T's bottom by classifying accessed nodes into left and right tree. When x becomes t, middle tree rooted at x is assembled with L and R.

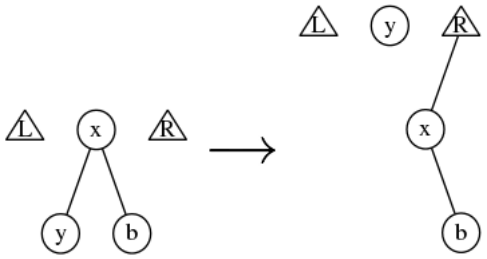

Figure 7: Splay down of x for case R1

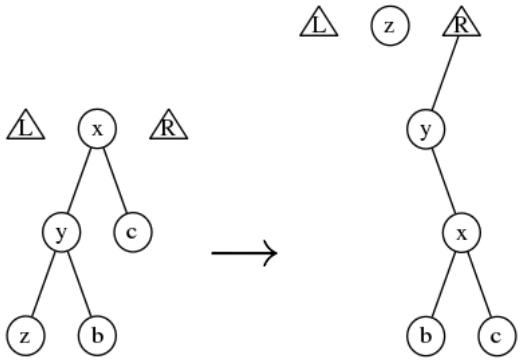

Figure 8: Splay down of x for case R2

Input: node $x \in T$ to start splaying until node with key K is reached
Output: tree splayed at x
Worst Case Complexity: $O(\lg n)$

```
splay_down(K, x)
  L = R = null
  while K ≠ key(x)
    if K < key(x)
      y = child_l(x)
      if K = key(y) {case R1}
        link_right(R, x)
        x = y
      else if K < key(y) {case R2}
        z = child_l(y)
        rotate_right(x)
        link_right(R, y)
        x = z
      else if K > key(y) {case R3}
        z = child_r(y)
        link_right(R, x)
        link_left(L, y)
        x = z
    else
      y = child_r(x)
      if K = key(y) {case L1}
        link_left(L, x)
        x = y
      else if K > key(y) {case L2}
```

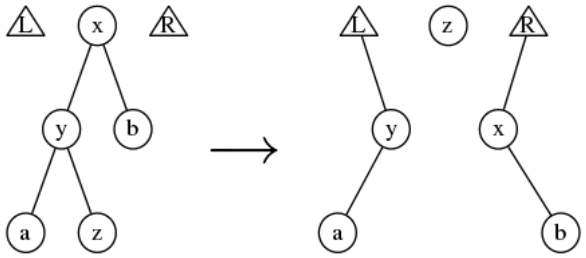

Figure 9: Splay down of x for case R3

```
        z = child_r(y)
        rotate_left(x)
        link_left(L, y)
        x = z
      else if K < key(y) {case L3}
        z = child_l(y)
        link_left(L, x)
        link_right(R, y)
        x = z
  assemble(x, L, R)
```

1.4 Finding key

Finding a key K in subtree T goes by traversing the tree from the root choosing left or right subtree depending of the key in the current node (same as in binary search tree). If such node x is found, then splaying on x is performed. If there is no node x such that $k(x) = K$, then the last (non-null) node on the search path is splayed (in the bottom-up manner). If T is empty, then splaying is not performed.

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

```
find(K)
  x = root(T)
  if x = null
    return null
  while x ≠ null
    y = x {track parent of x}
    if K < key(x)
      x = child_l(x)
    else if K > key(x)
      x = child_r(x)
    else
      break
  if x ≠ null
    splay_up(x)
  else
    splay_up(y)
  return x
```

1.5 Joining trees

Assuming that tree T_1 is less than tree T_2 (i.e. all keys from T_1 are less than each key from T_2), *join* constructs a single tree of all items from T_1 and T_2. Join finds the largest key $m = max(T_1)$ (by taking the rightmost node in T_1). Since finding m splays it (and makes $m = r(T_1)$), the root $r(T_1)$ has null right child. The operation is completed by making T_2 the right subtree of newly created root.

Input: T_1, T_2 trees to join
Output: splay tree consisting of T_1 and T_2
Amortized Complexity: $O(\lg n)$
join(T_1, T_2)
  $m =$ max(T_1)
  splay(m)
  child_r(m) $= T_2$
  parent(root$(T_2)) = m$

1.6 Splitting tree

Splitting tree T at key K returns two subtrees T_1 and T_2 by breaking T at the node which contains K. The operation is accomplished by finding node with K and then returning two trees formed by breaking left or right link of the new root. If K is not in T, then the last non-null node found during the search will be used for splitting.

Input: key K to split on
Output: trees T_1, T_2 obtained by splitting
Amortized Complexity: $O(\lg n)$
split(K)
  find(K) {*moves K to root*}
  $T_1 =$ root(T)
  $T_2 =$ child_r(root$(T))$
  child_r(root$(T)) =$ **null**
  parent(root$(T_2)) =$ **null**
  return $[T_1, T_2]$

1.7 Inserting key

Inserting key K goes by splitting T on K (which returns two subtrees T_1 and T_2), and then replacing T with a new root containing K and left/right subtrees T_1 and T_2. Since K does not exist in T, splitting is performed on the last non-null node found during the search.

Input: key K to insert into tree T
Output: T with key K
Amortized Complexity: $O(\lg n)$
insert(K)
  $[T_1, T_2] =$ split(K)
  key(root$(T)) = K$
  child_l(root$(T)) = T_1$, parent$(T_1) =$ root(T)
  child_r(root$(T)) = T_2$, parent$(T_2) =$ root(T)

1.8 Deleting key

Deleting key K goes by finding K (splaying moves it to root) and then replacing T by joining $c_l(r(T))$ and $c_r(r(T))$.

Input: key K to delete from tree T
Output: tree T without key K
Amortized Complexity: $O(\lg n)$
delete(K)
  find(K)
  join(child_l(root(T)), child_r(root(T)))

1.9 Amortized complexity

For each $x \in T$ of splay tree T, define it's size as $s(x) = \sum_{y \in T_x} w(y)$, where T_x is subtree rooted at x. Then, let's define rank of x as $\lambda(x) = \log s(x)$. Finally, define potential of T as $\Phi(T) = \sum_{x \in T} \lambda(x)$.

Lemma 1.1. The amortized time to splay tree with root $r(T)$ at node x is

$$\hat{c}_x = 3(\lambda(r(T)) - \lambda(x)) + 1 = O\left(\log \frac{s(r(T))}{s(x)}\right)$$

Proof See [4] for the proof. **QED**

For a sequence of m accesses on n-node splay tree, the potential decrease is $\Phi_m - \Phi_0$ where

$$\Phi_0 = \sum_{i=1}^{n} \lambda(i) = \sum_{i=1}^{n} \log s(i) = \sum_{i=1}^{n} \log \left(\sum_{j \in T_i} w(j) \right) \leq \sum_{i=1}^{n} \log \left(\sum_{j=1}^{n} w(j) \right)$$

$$\Phi_m = \sum_{i=1}^{n} \log \left(\sum_{j \in T_i^m} w(j) \right) \geq \sum_{i=1}^{n} \log w(i)$$

with T_i as subtree at i-th node before splaying and T_i^m as subtree at i-th node after m-th splay operation. Thus,

$$\Phi_m - \Phi_0 \leq \sum_{i=1}^{n} \log W - \sum_{i=1}^{n} \log w(i) = \sum_{i=1}^{n} \log \frac{W}{w(i)}$$

where $W = \sum_{i=1}^{n} w(i)$.

Let's apply the obtained results to the tree operations. For $x \in T$ denote with x_p and x_s predecessor and successor of x.

Theorem 1.2. The amortized time of search operation for $x = find(K)$ is

$$\begin{cases} 3 \log \frac{W}{w(x)} + 1, x \in T \\ 3 \log \frac{W}{min\{w(x_p), w(x_s)\}} + 1, x \notin T \end{cases}$$

Proof If $x \in T$, by lemma 1.1, the splaying time for node x is $3 \log \frac{s(r(T))}{s(x)} + 1 \leq 3 \log \frac{W}{w(x)} + 1$, where $W = s(r(T)), s(x) \geq w(x)$. If $x \notin T$, then either x_p or x_s is in T, so $s(x) \geq \min\{w(x_p), w(x_s)\}$ and thus splaying time is $3 \log \frac{W}{\min\{w(x_p), w(x_s)\}} + 1$. **QED**

Theorem 1.3. The amortized time of join operation is

$$3 \log \frac{W}{w(x)} + O(1)$$

where $x = \max T_1$.

Proof The bound on join is immediate from the bound on find - the splaying time is at most $3 \log \frac{s(T_1)}{w(x)} + 1$. The increase in potential caused by linking T_1 and T_2 is $\log \frac{s(T_1)+s(T_2)}{s(T_1)} \leq 3 \log \frac{W}{s(T_1)}$ (because $W = s(T_1) + s(T_2)$). **QED**

Theorem 1.4. The amortized time of split operation is

$$\begin{cases} 3 \log \frac{W}{w(x)} + O(1), x \in T \\ 3 \log \frac{W}{min\{w(x_p), w(x_s)\}} + O(1), x \notin T \end{cases}$$

where x is node such that $k(x) = K$.

Proof Since the searching operation is the only non-constant time operation, then amortized time of split is same as of find. **QED**

Theorem 1.5. The amortized time of insert operation is

$$3 \log \frac{W - w(x)}{\min \{w(x_p), w(x_s)\}} + \log \frac{W}{w(x)} + O(1)$$

where x is such that $k(x) = K$.

Proof Follows from the complexity of finding a key. **QED**

Theorem 1.6. The amortized time od delete operation is

$$3 \log \frac{W}{w(x)} + 3 \log \frac{W - w(x)}{w(x_p)} + O(1)$$

Proof Result follows from the bounds on find and join operations. **QED**

Theorem 1.7. The amortized time of search operation for $x = find(K)$ is $O(\lg n)$, where $n = |T|$ is number of nodes in T.

Proof Assigning $w(x) = 1/n$ in theorem 1.2 the proof follows. **QED**

1.10 Use cases

- Most Recently Used cache

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

[4] Daniel Dominic Sleator, Robert Endre Tarjan: Self-Adjusting Binary Search Trees