Binary search tree and Red-Black tree

1 Binary search tree

Note: This section only serves as reference for the next section. For more details on BST refer to your notes, or the textbook.

For insertion, we perform a search in the tree for a place of the new element, then insert it there. This is to make sure that after the insertion the Binary search tree property is preserved. In the algorithm BST-Insert(T, x) below we insert a new element x in to the tree T. We keep track of the supposed parent of x (node z), and the supposed place of x (node y). Thus z is the parent of y, and we move the pair (z, y) down the path that a search for x in T travels.

BST-Insert(T, x)

1. $z \leftarrow NIL$ % parent of x
2. $y \leftarrow T.root$ % place for x
3. while $y \neq NIL$ do
4. $\quad z \leftarrow y$
5. $\quad$ if $x.key < y.key$ do
6. $\quad\quad y \leftarrow y.left$
7. $\quad$ else
8. $\quad\quad y \leftarrow y.right$
9. $\quad$ end if
10. end while
11. % now make x a child of z
12. $x.parent \leftarrow z$
13. if $z = NIL$
14. $\quad T.root \leftarrow x$
15. else if $x.key < z.key$ then
16. $\quad z.left \leftarrow x$
17. else
18. $\quad z.right \leftarrow x$
19. end if

Now for deletion. Suppose that we want to delete a node x from a binary search tree T. As in the case of insertion, we have to make sure that the Binary search tree property is maintained after deletion. We consider the following cases:

1. x has no left child, i.e., $x.left = NIL$: Then we can simply replace x by its right child. (Note that this covers the case where x is a leaf, i.e., both its children are NIL.)
2. x has no right child, i.e., $x.right = NIL$: In this case we replace x by its left child.
3. x has two children. Then we find the successor y of x (using BST-Successor). Note that y is the left-most descendant if x's right child (in particular, $y.left = NIL$). Now there are two subcases.
   (a) y is the right child of x, then simply replace x by y **and make the left child of x the left child of y.**
   (b) y is not the right child of x, then first replace y by its right child, then put y into x's place (keeping x's children and parent as children and parent of y) i.e., copy the key of y (and all its satelite data, if there are any) into x's location.

The "replacement" discussed above is performed by the following subroutine, BST-Transplant(T, u, v), which replaces node u by node v: u's parent becomes v's parent, and v becomes the appropriate child of u's parent. (If u is the root of T then now v is the root of T.)

BST-Transplant(T, u, v)

1. if $u.parent = NIL$ % if u is the root
2. $T.root \leftarrow v$
3. else if $u = u.parent.left$
4. $u.parent.left \leftarrow v$
5. else
6. $u.parent.right \leftarrow v$
7. end if
8. if $v \neq NIL$
9. $v.parent \leftarrow u.parent$
10. end if

The code for deletion is given below:

BST-Delete(T, x)

```
1.  if x.left = NIL    % Case 1
2.      BST-Transplant(T, x, x.right)
3.  else if x.right = NIL    % Case 2
4.      BST-Transplant(T, x, x.left)
5.  else    % Case 3
6.      y ←BST-Successor(T, x)
7.      z ← y.right
8.      if y = x.right    % Case 3(a)
9.          BST-Transplant(T, x, y)
10.         y.left ← x.left
11.     else    % Case 3(b)
12.         BST-Transplant(T, y, y.right)
13.         % now make x's parent/children y's
14.         y.left ← x.left
15.         y.right ← x.right
16.         y.parent ← x.parent
17.         if x.parent = NIL
18.             T.root ← y
19.         else if x = x.parent.left
20.             x.parent.left ← y
21.         else
22.             x.parent.right ← y
23.         end if
24.     end if
25. end if
```

2 Red-Black tree

A binary search tree in general can be very "unbalanced"; its height h can equal $(n-1)$, where n is the number of nodes in the tree. Thus the queries that run in time $\Theta(h)$ are expensive. We want a kind of BSTs that is guaranteed to have small height compared to the number of nodes. Note that in the ideal case, a (perfect) balance binary tree (like a heap), the height of a binary tree having n nodes is at most $\lceil \log_2(n+1) - 1 \rceil$, i.e., roughly $\log_2(n)$. The small height we want here is $c \log_2(n)$, for some constant $c > 1$. In other words, we want to have $\mathcal{O}(\log_2(n))$-height.

This turns out be guaranteed by making sure that the lengths of the branches in the trees are within a constant factor of each other. For example, it is shown in class that if the lengths of the branches are within a factor of 2 of each other, then the height of the tree is at most $2\log_2(n+1)$, i.e., $\mathcal{O}(\log_2(n))$.

A Red-Black tree is a binary search tree that satisfies the above condition by maintaining the following conditions on a coloring of its vertices by two colors red and black:

1. Every node in the tree is colored either black or red,
2. The root is black,
3. If a node is red, then its children (if there are any) are black,
4. The branches of the tree have the same number of black node,
5. If a node x has only one child y, then y is a leaf, and y is red.

Notes that if (1), (3), (4) and (5) are already satisfied then 2) can also be satisfied simply by coloring the root black. Property (5) ensures that the binary tree is almost full, i.e., except for the leaves and for the parents of the leaves, every node has exactly two children.

Now we see how insertion and deletion work for Red-black trees. Having a balance tree, every time we insert a new node into it or delete a node from it we have to make sure that the tree remains balance. In other words, we have to make sure that after each insertion and deletion, the properties 1–5 above are maintained.

We will need to following subroutines: Left-rotate and Right-rotate. The intuition behind these (see pictures drawn in class, or in the text book) is that they elevate a subtree when it becomes "deeper" as a result of inserting a new node in it (or as a result of removing from another subtree).

The subroutine Left-rotate(T, x) rotates the link between x and its right child y to the left, making y x's parent, and x y's left child.

Left-rotate(T, x):

1. if $x \neq NIL$ and $x.right \neq NIL$
2. % moving the subtrees
3. $y \leftarrow x.right$
4. $x.right \leftarrow y.left$
5. $y.left \leftarrow x$
6. % now make x's parent y's parent, and fixing the pointer from y's parent to y

7. $y.parent \leftarrow x.parent$

8. if $x.parent = NIL$ % x is the root

9. $T.root \leftarrow y$

10. else

11. if $x = x.parent.left$

12. $x.parent.left \leftarrow y$

13. else

14. $x.parent.right \leftarrow y$

15. end if

16. end if

17. $x.parent \leftarrow y$

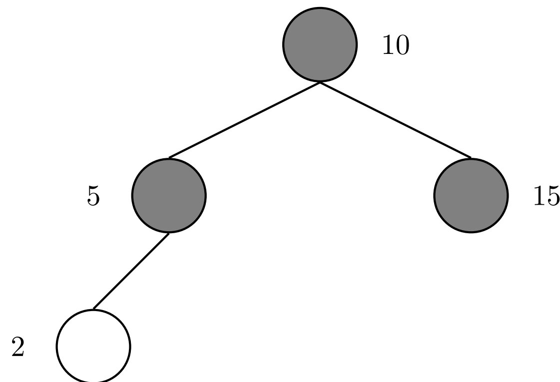
18. end if

The procedure Right-rotate(T, x) works in the opposite direction. It rotates the link between x and its left child y to the right, making x a right child of y. Details are left as an exercise.

2.1 Insertion

To insert a new node x into a tree T, first we do as insertion for an ordinary binary search tree. Now there are two choice for coloring the new node: we can color it either black or red. Of course some properties will be violated so we need to fix them up.

Coloring the new node black means that there is a branch which has more more black node than all other branches, and this violates condition (4) and possibly (5). For example, consider inserting a node with key 7 into the following Red-black tree. If we color the new node black then to fix the violation of (4) we have to look at all the tree.



The other choice, i.e., coloring the new node red, results in only at most a single (local) violation of property (3): there is at most a pair of vertices $(x, x.parent)$ that are both red. (Note that if $x.parent$ is red, then its other children, if it exists, is black.) We can fix this violation by pushing the pair up the tree. At the root (i.e., when $x.parent$ is the root) it can be fixed simply by coloring the root black. This is exactly what the procedure RB-Insert-Fix(T, x) does.

Now we give the details for RB-Insert. As mentioned above, we first perform the regular insertion for BST, with additional task of coloring, then we call RB-Insert-Fix.

RB-Insert(T, x): % insert a new node x in to the red-black tree T

1. $z \leftarrow NIL$ % parent of x
2. $y \leftarrow T.root$ % place for x
3. while $y \neq NIL$ do
4. $\quad z \leftarrow y$
5. $\quad$ if $x.key < y.key$ do
6. $\qquad y \leftarrow y.left$
7. $\quad$ else
8. $\qquad y \leftarrow y.right$
9. $\quad$ end if
10. end while
11. % now make x a child of z
12. $x.parent \leftarrow z$
13. if $z = NIL$
14. $\quad T.root \leftarrow x$
15. else if $x.key < z.key$ then
16. $\quad z.left \leftarrow x$
17. else
18. $\quad z.right \leftarrow x$
19. end if
20. $x.color \leftarrow red$
21. RB-Insert-Fix(T, x)

Now we discuss the fixing-up procedure RB-Insert-Fix(T, x). REFER TO YOUR LECTURE NOTES, OR THE TEXTBOOK FOR THE ILLUSTRATIONS, OR DRAW YOUR OWN.

If x is at the root, we simply color it black (as required by property (2)). If $x.parent$ is already black, there is nothing to fix. So we work under the assumption that both x and $x.parent$ are red (this is the condition for the while-loop below).

If $x.parent$ is already at the root, we just need to color it black and will be done. Otherwise it has a parent, which must be black, and we consider two cases, depending on whether it's a left child (Case Left) or a right child (Case Right) of its parent. The two cases are symmetric, so we focus on Case Left: the case where $x.parent$ is the left child of its parent.

This case consists of subcases, depending on the color of $x.parent$'s sibling which we call y. Below we consider the case where y is not NIL. The case where y is NIL (i.e., $x.parent$ was a leaf) is handled in the same way as case IIb.

- Case I: y is red. Then $x.parent.parent$ must be black. We move the pair $(x, x.parent)$ one level up, by recoloring both $x.parent$ and y black, and $x.parent.parent$ red. The new value of x is $x.parent$.

- Case II: y is black. In order to move the pair $(x, x.parent)$ one level up (closer to the root) we will do a right rotation at $x.parent.parent$. The rotation results in $x.parent$'s right child being $x.parent.parent$'s left child. So we want to make sure that x is not the right child of $x.parent$ before rotating. We consider two subcases:
  - Case IIa: x is the right child of $x.parent$. We will reduce this case to the next case (IIb: x is the left child of $x.parent$) by a left rotation at $x.parent$.
  - Case IIb: x is the left child of $x.parent$. We elevate x up by a right rotation at $x.parent.parent$. Now the branches going through x has one less black node. We fix this problem by recoloring $x.parent$ black, and recoloring $x.parent.parent$ (which by this time is a right child of $x.parent$) red. (The algorithm terminates at this point.)

RB-Insert-Fix(T, x)

1. while $x.parent \neq NIL$ and $x.parent.color = Red$ do
2.   if $x.parent = T.root$ do $x.parent.color \leftarrow Black$
3.   else
4.   if $x.parent = x.parent.parent.left$ % Case Left
5. $y \leftarrow x.parent.parent.right$
6.   if $y = NIL$ % similar to case IIb below
7. $x.parent.parent.color \leftarrow Red$
8. $x.parent.color \leftarrow Black$
9.   Right-rotate$(T, x.parent.parent)$
10.   else if $y.color = Red$ % Case I

```
11.               x.parent.color ← Black
12.               y.color ← Black
13.               x.parent.parent.color ← Red
14.               x ← x.parent.parent
15.            else    % Case II: now y's color is Black
16.               if x = x.parent.right    % Case IIa
17.                  Left-rotate(T, x.parent)
18.                  x ← x.parent
19.               end if
20.               % now case IIb
21.               Right-rotate(T, x.parent.parent)
22.               x.parent.parent.color ← Red
23.               x.parent.color ← Black
24.            end if
25.         else   % Case R
26.            CODE FOR CASE R HERE
27.         end if
28.      end if
29. end while
```

2.2 Deletion

For deletion, we perform the usual BST deletion, then as for insertion we need to take care of possible violations of the Red-black tree properties. Recall the cases for BST deletion on page 2. We will follow these cases and point out how to fix violations that arises in each case.

Suppose we are deleting node x. In each case we will take note of the "lost color", i.e., the color of the node that is removed from the tree (this is the color of either x or its successor y). If the lost color is black, then we need to do a fixing up. Our fixing-up algorithm will start at a vertex z with the assumption that

- if $z \neq NIL$ then all branches going through z have one less black node than other branches (this violates condition 4);
- if $z = NIL$ then the parent p of z violates condition 5 (because after the deletion it p has a single child which is not a red leaf).

In the first scenario, of course if z is red and is not the only child of its parent, the violation can be resolved simply by coloring it black. So we will assume that z is black. By rotating we will push this violation up to the root, and at that point it is no longer a violation, because there are no other branches.

Because z can be NIL, we need to keep track of its parent p as well.

1. x has no left child, i.e., $x.left = NIL$. Then we replace x by $x.right$, so $x.right$'s parent is now $x.parent$. If x's color is black then we call fixing-up procedure for $z = x.righ$, with $p = x.parent$.

2. x has a left child but it has no right child, i.e., $x.left \neq NIL$ and $x.right = NIL$: Then by condition 5 the left child must be a red leaf, and hence by condition 3 x must be a black node. We simply replace x by its left child and color its left child black.

3. x has two children. Then we find the successor y of x (using BST-Successor). Note that y is the left-most descendant if x's right child (in particular, $y.left = NIL$). Now there are two subcases.

   (a) y is the right child of x, then simply replace x by y and make the left child of x the left child of y. Also, we give y the color of x. So if y is black, we loose a black node on the branches going through y's right child. In this case we have to call the fixing-up procedure for $z = y.right$ and $p = y$.

   (b) y is not the right child of x: first replace y by its right child, then put y into x's place (keeping x's children and parent as children and parent of y) i.e., copy the key of y (and all its satelite data, if there are any) into x's location. We maintain the color of x, and call the fixing up procedure for $z = y.right$ and $p = y.parent$.

RB-Delete(T, x)

1. lost-color $\leftarrow x.color$
2. $z \leftarrow x$
3. $p \leftarrow x.parent$
4. if $x.left = NIL$ % Case 1
5. $\quad z \leftarrow x.right$
6. $\quad$ BST-Transplant$(T, x, x.right)$
7. else if $x.right = NIL$ % Case 2
8. $\quad z \leftarrow x.left$
9. $\quad$ BST-Transplant$(T, x, x.left)$
10. else % Case 3
11. $\quad y \leftarrow$BST-Successor(T, x)

12. $z \leftarrow y.right$

13. lost-color $\leftarrow y.color$

14. if $y = x.right$ % Case 3(a)

15. BST-Transplant(T, x, y)

16. $y.left \leftarrow x.left$

17. $p \leftarrow y$

18. else % Case 3(b)

19. $p \leftarrow y.parent$

20. BST-Transplant$(T, y, y.right)$

21. % now make x's parent/children y's

22. $y.left \leftarrow x.left$

23. $y.right \leftarrow x.right$

24. $y.parent \leftarrow x.parent$

25. if $x.parent = NIL$

26. $T.root \leftarrow y$

27. else if $x = x.parent.left$

28. $x.parent.left \leftarrow y$

29. else

30. $x.parent.right \leftarrow y$

31. end if

32. end if

33. end if

34. if lost-color = black

35. RB-Delete-Fixup(T, z, p)

36. end if

The procedure for fixing up: RB-Delete-Fixup(T, z, p). If z is already the root (i.e., $p = NIL$) then simply make sure that z is black. Otherwise we consider the case where $z = NIL$ and $z \neq NIL$.

First, suppose that $z \neq NIL$. If z is red then color it black and we are done. So suppose that z is black. Note that all branches going through z have one less black node than other branches. There are two subcases:

- Case L: z is the left child of p,
- Case R: z is the right child of p.

The two cases are symmetric, so we discuss only Case L. This case is further divided into subcases depending on the color of w, z's sibling. (Note that z must have a sibling in this case; furthermore, this sibling must have two children (Proof?).)

- Case I: w is red. Then by condition 3 p must be black. It can be deduced that both of w's children must be black. We do a left rotate at p, recolor w to black and p to red. This bring us to case II below (with new w).
- Case II: w is black.
  - Case IIa: w has two black children. We color w red. Then consider:
    * Case IIa-1: p is black. Then make p the new value of z (and recursively go up the path).
    * Case IIa-2: p is red. Then color p black and we are done.
  - Case IIb: $w.right$ is black (and hence $w.left$ is red). Then it can be shown that $w.left$ have two children who are both black. We reduce this to case IIc below (with a new value of w), by rotating right at w, and recoloring w red and $w.left$ (which now becomes w's parent) black.
  - Case IIc: $w.right$ is red. First we perform a left rotate at p. We let w have p's color, and color p black. Also color $w.right$ black. (Verify that this preserves the number of black nodes going through $w.right$ as well as $w.left$ and increases the number of black nodes on branches going through z by 1.)

Now we consider the case where $z = NIL$ (and the lost-color is black). Recall that this happens in the following cases:

- x is a black leaf (and z is $x.right$, see case 1 of BST-Delete). In this case p is $x.parent$, and z can be either left or right child of p. (If $p \neq NIL$, then it must have another child.)
- x has two children, and its successor y is a black leaf (and $z = y.right$, see case 3 of BST-Delete). In this case p is y's parent, and we know that p must have a right child.

We will consider the case LL where $p.left$ is NIL (so p has a right child). Let w denote $p.right$. The case RR where $p.right = NIL$ is symmetric, and is left as an exercise.

We consider the following subcases of case LL:

- Case A: w is red. In this case p must be black, and w must have two children both of them are black. As in case I above, we do a left rotate at p, recolor w to black and p to red. This bring us to case B below (with new w).
- Case B: w is black. In this case, w cannot have a proper black descendant. (Prove this.) This case is similar to but simpler than case II above. We consider the following subcases:
  - Case B1: w is is a leaf. (So condition 5 is violated.) We color w red. Now if p is red we color it black, and we are done. Otherwise, p is black, and call the fixing up procedure recursively at p.

- Case B2: w has at least one child (which must be red leaf(s)).
  * Case B2a: If $w.right$ is NIL (then $w.left \neq NIL$) then right rotate at w, and recolor w red, recolor $w.left$ (which is now w's new parent) black. Let the new value of w be $p.right$. This takes us to case B2b below.
  * Case B2b: $w.right \neq NIL$. This is similar to case IIc above. First, do a left rotate at p. Then give w p's color, and color p black. Also color $w.right$ black, and we are done.

The pseudo-code for the fixing-up procedure is as follows.
RB-Delete-Fixup(T, z, p)

1. while $p \neq NIL$
2. if $z \neq NIL$ and $z = p.left$ % Case L
3. $w \leftarrow p.right$
4. if $w.color = Red$ % Case I
5. Left-rotate(T, p)
6. $p.color \leftarrow Red$
7. $w.color \leftarrow Black$
8. $w \leftarrow p.right$
9. end if % Case I is now reduced to Case II
10. if $w.left.color = Black$ and $w.right.color = Black$ % Case IIa
11. $w.color \leftarrow Red$
12. if $p.color = Black$ % Case IIa-1
13. $z \leftarrow p$
14. $p \leftarrow z.parent$
15. else % Case IIa-2
16. $p.color \leftarrow Black$
17. $p \leftarrow NIL$ % indicate that the while loop should end now
18. end if
19. else % Case IIb and IIc
20. if $w.right.color = Black$ % Case IIb
21. $w.color \leftarrow Red$

```
22.             w.right.color ← Black
23.             Right-rotate(T, w)
24.             w ← p.right
25.         end if   % Now we are in case IIc
26.         w.color ← p.color
27.         p.color ← Black
28.         w.right.color ← Black
29.         Left-rotate(T, p)
30.         p ← NIL   % exiting the while loop
31.     end if
32. else if z ≠ NIL and z = p.right    % Case R is symmetric and left as an exercise
33.     DO CASE R HERE
34. else if z = NIL and p.right ≠ NIL   % Case LL
35.     w ← p.right
36.     if w.color = Red    % Case A
37.         Left-rotate(T, p)
38.         p.color ← Red
39.         w.color ← Black
40.         w ← p.right
41.     end if   % Now case A is reduced to case B
42.     if w.left = NIL and w.right = NIL   % Case B1
43.         w.color ← Red
44.         if p.color = Red
45.             p.color ← Black
46.             p ← NIL   % indicate that the while loop stop now
47.         else
48.             z ← p
49.             p ← z.parent
```

```
50.             end if
51.         else   % Case B2
52.             if w.right = NIL    % Case B2a
53.                 w.left.color ← Black
54.                 w.color ← Red
55.                 Right-rotate(T, w)
56.                 w ← p.right
57.             end if   % Now we are in case B2b
58.             w.color ← p.color
59.             p.color ← Black
60.             w.right.color ← Black
61.             Left-rotate(T, p)
62.             p ← NIL   % exiting the while loop
63.         end if
64.     else if z = NIL and p.left ≠ NIL   Similar as above
65.         DO CASE RR HERE
66.     end if
67. end while
68. z.color ← Black
```