`http://www.cs.unh.edu/~ruml/cs758`

Red-Black Trees

Searching

Structure	Find	Insert	Delete
List			
Heap			
Hash table			
Binary tree			
Binary tree (balanced)			

Balanced Trees

1. AVL Trees (1962)
2. 2-3 Trees
3. red-black trees (1972, popularized 1978)
4. AA trees (1992)
5. left-leaning red-black trees (2008)

probabilistically balanced

1. treaps
2. skip lists

Red-Black Trees

node: data, left, right, parent, color

1. every node is either red or black
2. the root is black
3. (consider nil to be black)
4. both children of a red node are black
5. from any node, all paths to leaves have the same 'black height'

search and traversal are unchanged

Rotation

useful subroutines:

- rotate-right
- rotate-left

Insert(z)

1. z's parent $\leftarrow$ find-parent(z, root, nil)
2. if parent is nil
3.   root $\leftarrow z$
4. else
5.   if z should be before parent
6.   parent's left child $\leftarrow z$
7.   else
8.   parent's right child $\leftarrow z$
9. z's children $\leftarrow$ nil

Insert(z)

1. z's parent $\leftarrow$ find-parent(z, root, nil)
2. if parent is nil
3.   root $\leftarrow z$
4. else
5.   if z should be before parent
6.   parent's left child $\leftarrow z$
7.   else
8.   parent's right child $\leftarrow z$
9. z's children $\leftarrow$ nil
10. color z red
11. fix-insert(z)

Fixing Insertion

Recall properties:

1. every node is either red or black
2. the root is black
3. (consider nil to be black)
4. both children of a red node are black
5. from any node, all paths to leaves have the same 'black height'

Recall properties:

1. every node is either red or black
2. the root is black
3. (consider nil to be black)
4. both children of a red node are black
5. from any node, all paths to leaves have the same ‘black height’

Fixing Insertion

Recall properties:

1. every node is either red or black
2. the root is black
3. (consider nil to be black)
4. both children of a red node are black
5. from any node, all paths to leaves have the same 'black height'

Fixing Insertion

Recall properties:

1. every node is either red or black
2. the root is black
3. (consider nil to be black)
4. both children of a red node are black
5. from any node, all paths to leaves have the same 'black height'

Cases:

1. red root (property 2)
2. two red in a row (property 4)

Fixup Invariant

Cases:

1. red root (property 2)
2. two red in a row (property 4)

During fixup:

1. z is red
2. if z's parent is the root, it is black
3. at most, property 2 xor 4 is violated at z
   (a) if 2: because z is root and red
   (b) if 4: because z and parent are red

Fixup Invariant

Cases:

1. red root (property 2)
2. two red in a row (property 4)

During fixup:

1. z is red
2. if z's parent is the root, it is black
3. at most, property 2 xor 4 is violated at z
   (a) if 2: because z is root and red
   (b) if 4: because z and parent are red

Initialization:

1. we colored z red
2. we didn't touch z's parent, and roots are black
3. just saw this

```
while z's parent is red
    if z's parent is a left child
        y ← z's uncle (a right child)
        if y is red
            color z's parent black              case 1
            color z's uncle y black
            color z's grandparent red
            z ← z's grandparent
        else if z is a right child
                z ← z's parent                  case 2
                rotate-left(z)
            color z's parent black              case 3
            color z's grandparent red
            rotate-right(z's grandparent)
    else, 3 symmetric cases (left↔right)
color root black
```

First Look: Termination

Assuming other properties are maintained, are we red-black now?

Leverage the invariant:

1. irrelevant
2. irrelevant
3. only 2 xor 4 can be violated in loop
   (a) if 2: root colored black at end, so 2 not violated
   (b) if 4: z's parent now black, so 4 not violated

First Look: Termination

Assuming other properties are maintained, are we red-black now?

Leverage the invariant:

1. irrelevant
2. irrelevant
3. only 2 xor 4 can be violated in loop
   (a) if 2: root colored black at end, so 2 not violated
   (b) if 4: z's parent now black, so 4 not violated

How to make progress around loop while maintaining invariant?

Break

- asst 4
- Steve office hours survey

Red-Black Trees

Maintenance

central problem: prop 4 violated: z and parent are red

note z has an uncle because the root is black

3 cases ($+$ 3 more by symmetry of z's parent being left/right):

1. z's uncle y is also red (we have a red layer)
2. z's uncle y is black and z is right child
3. z's uncle y is black and z is left child

Maintenance

central problem: prop 4 violated: z and parent are red

note z has an uncle because the root is black

3 cases ($+$ 3 more by symmetry of z's parent being left/right):

1. z's uncle y is also red (we have a red layer)
2. z's uncle y is black and z is right child
3. z's uncle y is black and z is left child

Plan:

1. fix case 1, possibly introducing case 2.
2. reduce case 2 to case 3.
3. fix case 3.

Case 1

case 1: z's uncle y is also red

solution: move redness up

1. color z's parent and uncle black
2. color grandparent red and recur

fixup loop invariants:

1. z is red
2. if z's parent is the root, it is black (unchanged)
3. at most, property 2 xor 4 is violated at new z. Note previous violations at old z are fixed.
   (a) if 2: because z is root and red
   (b) if 4: because z and parent are red

if new z is root, will be colored black, increasing all heights

Case 2

case 2: z's uncle y is black and z is right child

reduce to case 3: z's uncle y is black and z is left child

rotation doesn't affect any properties

Case 3

case 3: z's uncle y is black and z is left child

fix prop 4 at z: pull blackness down to z's parent and rotate grandparent under it.

fixup loop invariants:

1. z is red
2. if z's parent is the root, it is black
3. at most, property 2 xor 4 is violated at z.
   (a) can't be prop 2 (root black)
   (b) if 4: fixed because z's parent is now black
   (c) note black-height is preserved!

We are done and loop will exit

Complexity

finding place is

Complexity

finding place is $O(\lg n)$

one fixup iteration is constant time

fixup loops only when moving up, so is

Complexity

finding place is $O(\lg n)$

one fixup iteration is constant time

fixup loops only when moving up, so is $O(\lg n)$

how many rotations are performed?

EOLQs

For example:

- What's still confusing?
- What question didn't you get to ask today?
- What would you like to hear more about?

Please write down your most pressing question about algorithms and put it in the box on your way out.
Thanks!