Red-Black Trees

Red-Black Trees

each node: data, left, right, parent, color

Properties:

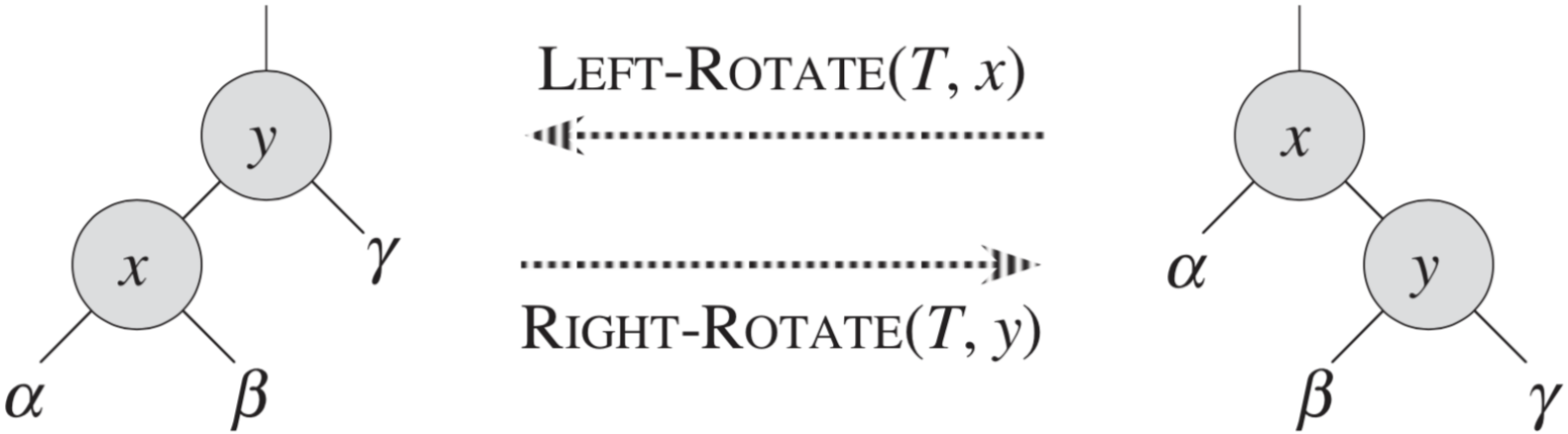
- every node is either red or black
- the root is black
- nil is black
- both children of a red node are black
- from any node, all the paths to leaf *nil*s have the same number of black nodes

changes to `find` and `next`/`prev`?

Rotation

useful subroutines:

- ROTATE-RIGHT
- ROTATE-LEFT



Insert

INSERT(n)

```
 1: n.parent ← FINDPARENT(n, root, nil)
 2: if n.parent is nil then
 3:     root ← n
 4: else
 5:     if n < n.parent then
 6:         n.parent.left ← n
 7:     else
 8:         n.parent.right ← n
 9: n's children ← nil
10: color n red
11: FIX-INSERT(n)
```

Fixing Insertion

Properties:

- every node is either red or black
- the root is black
- nil is black
- both children of a red node are black
- from any node, all the paths to leaf *nil*s have the same number of black nodes

Cases:

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

Fixing Insertion

Properties:

- every node is either red or black
- **the root is black**
- nil is black
- **both children of a red node are black**
- from any node, all the paths to leaf *nil*s have the same number of black nodes

Cases:

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

FIX-INSERT(z)

```
 1: while z's parent is red do
 2:     if z's parent is a left child then
 3:         y ← z's grandparent's right child
 4:         if y is red then
 5:             color z's parent black
 6:             color y black
 7:             color z's grandparent red
 8:             z ← z's grandparent
 9:         else
10:             if z is a right child then
11:                 z ← z's parent
12:                 rotate-left(z)
13:             color z's parent black
14:             color z's grandparent red
15:             rotate-right(z's grandparent)
16:     else 3 symmetric cases (switch left↔right)
17: color root black
```

Fixing Invariant

Cases:

1. root is red (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, one property is violated at z
   a) if property 2: z is root and red
   b) if property 4: z and parent are both red

Invariant initialization:

1. we colored z red
2. we didn't touch z's parent, and root is black
3. just shown

Fixing Invariant

Cases:

1. root is red (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, one property is violated at z
   a) if property 2: z is root and red
   b) if property 4: z and parent are both red

Invariant initialization:

1. we colored z red
2. we didn't touch z's parent, and root is black
3. just shown

Invariant Termination

Assuming other properties are maintained, do we have a red-black tree now?

Examine invariant:

1. irrelevant
2. irrelevant
3. only 2 xor 4 can be violated in loop
   a) if 2: root colored black at line 17
   b) if 4: z's parent is black (by 5 and 13), so 4 is not violated

What about maintaining the other red-black properties?

Invariant Maintenance

central problem: z and parent are red

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

1. z's uncle y is also red
2. z's uncle y is black and z is a right child
3. z's uncle y is black and z is a left child

Proof plan:

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

Invariant Maintenance

central problem: z and parent are red
3 cases (+3 more by symmetry of z's parent being left/right):

1. z's uncle y is also red
2. z's uncle y is black and z is a right child
3. z's uncle y is black and z is a left child

Proof plan:

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

Radix Trees

Searching

What if we are searching for long keys?

Can we detect a miss without examining the entire key?

Tries

trie: test each digit of key, branch on digit value

- some nodes do not hold values
- trie depth = key length
- canonical representation

retrieval

CLRS: ‘trie’ = ‘radix tree’
Wikipedia: ‘trie’ ≠ ‘radix tree’, ‘radix tree’ = ‘radix trie’ = ‘patricia trie’

duplicate keys?
what is their weakness?