Algorithms

Skip Lists

Administration

- Hand back homework 3
- Hand back exam 1
- Go over exam

Review: Red-Black Trees

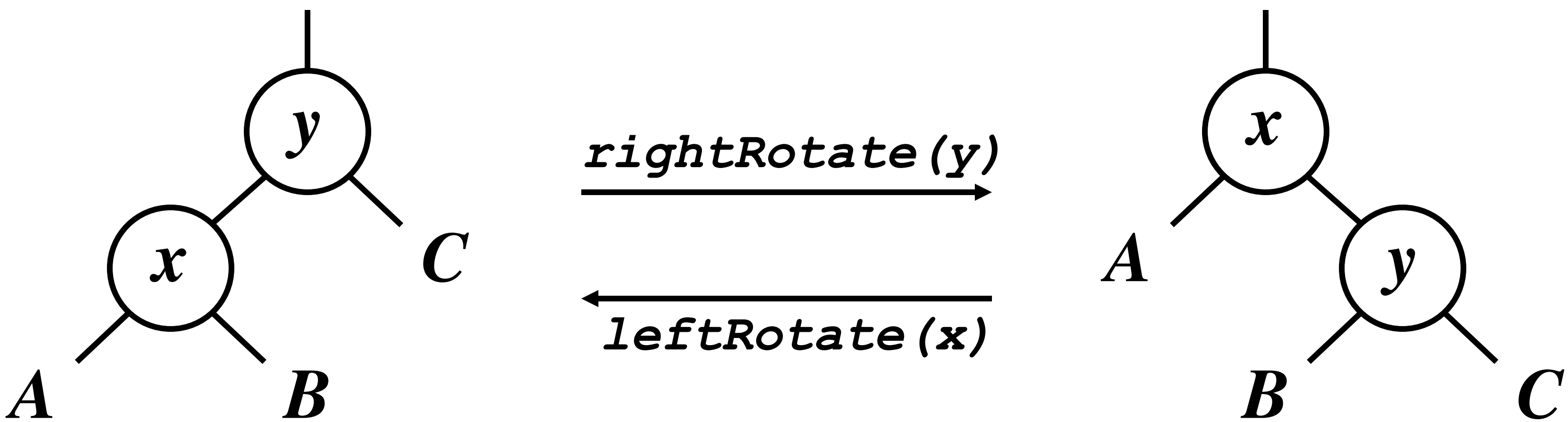
- *Red-black trees*:
  - Binary search trees augmented with node color
  - Operations designed to guarantee that the height $h = O(\lg n)$
- We described the properties of red-black trees
- We proved that these guarantee $h = O(\lg n)$
- Next: describe operations on red-black trees

Review: Red-Black Properties

- The *red-black properties*:
  1. Every node is either red or black
  2. Every leaf (NULL pointer) is black
     - Note: this means every “real” node has 2 children
  3. If a node is red, both children are black
     - Note: can’t have 2 consecutive reds on a path
  4. Every path from node to descendent leaf contains the same number of black nodes
  5. The root is always black

Review: RB Trees: Rotation

- Our basic operation for changing tree structure is called *rotation*:



- Preserves BST key ordering
- O(1) time…just changes some pointers

Review: Red-Black Trees: Insertion

- Insertion: the basic idea
  - Insert x into tree, color x red
  - Only r-b property 3 might be violated (if p[x] red)
    - If so, move violation up tree until a place is found where it can be fixed
  - Total time will be O(lg n)

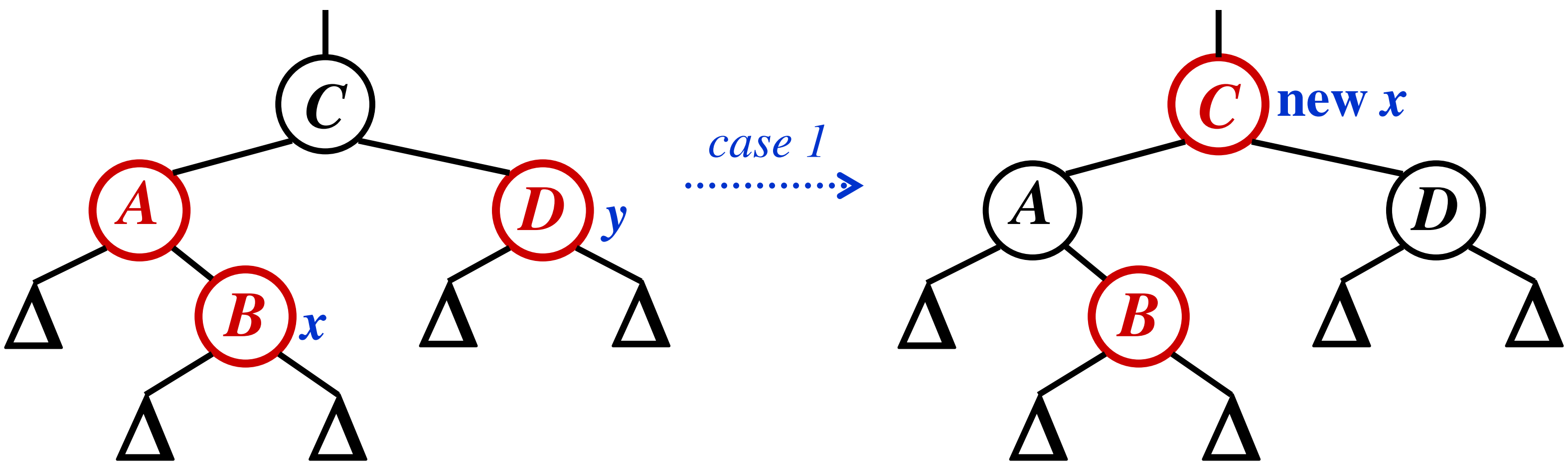
```
rbInsert(x)
  treeInsert(x);
  x->color = RED;
  // Move violation of #3 up tree, maintaining #4 as invariant:
  while (x!=root && x->p->color == RED)
  if (x->p == x->p->p->left)
      y = x->p->p->right;
      if (y->color == RED)
          x->p->color = BLACK;
          y->color = BLACK;                     Case 1
          x->p->p->color = RED;
          x = x->p->p;
      else   // y->color == BLACK
          if (x == x->p->right)
              x = x->p;                         Case 2
              leftRotate(x);
          x->p->color = BLACK;
          x->p->p->color = RED;                 Case 3
          rightRotate(x->p->p);
  else    // x->p == x->p->p->right
      (same as above, but with
       "right" & "left" exchanged)
```

```
rbInsert(x)
  treeInsert(x);
  x->color = RED;
  // Move violation of #3 up tree, maintaining #4 as invariant:
  while (x!=root && x->p->color == RED)
  if (x->p == x->p->p->left)
      y = x->p->p->right;
      if (y->color == RED)
          x->p->color = BLACK;
          y->color = BLACK;                  Case 1: uncle is RED
          x->p->p->color = RED;
          x = x->p->p;
      else    // y->color == BLACK
          if (x == x->p->right)
              x = x->p;                      Case 2
              leftRotate(x);
          x->p->color = BLACK;
          x->p->p->color = RED;              Case 3
          rightRotate(x->p->p);
  else    // x->p == x->p->p->right
      (same as above, but with
       "right" & "left" exchanged)
```

Review: RB Insert: Case 1

```
if (y->color == RED)
    x->p->color = BLACK;
    y->color = BLACK;
    x->p->p->color = RED;
    x = x->p->p;
```

- Case 1: "uncle" is red
- In figures below, all Δ's are equal-black-height subtrees

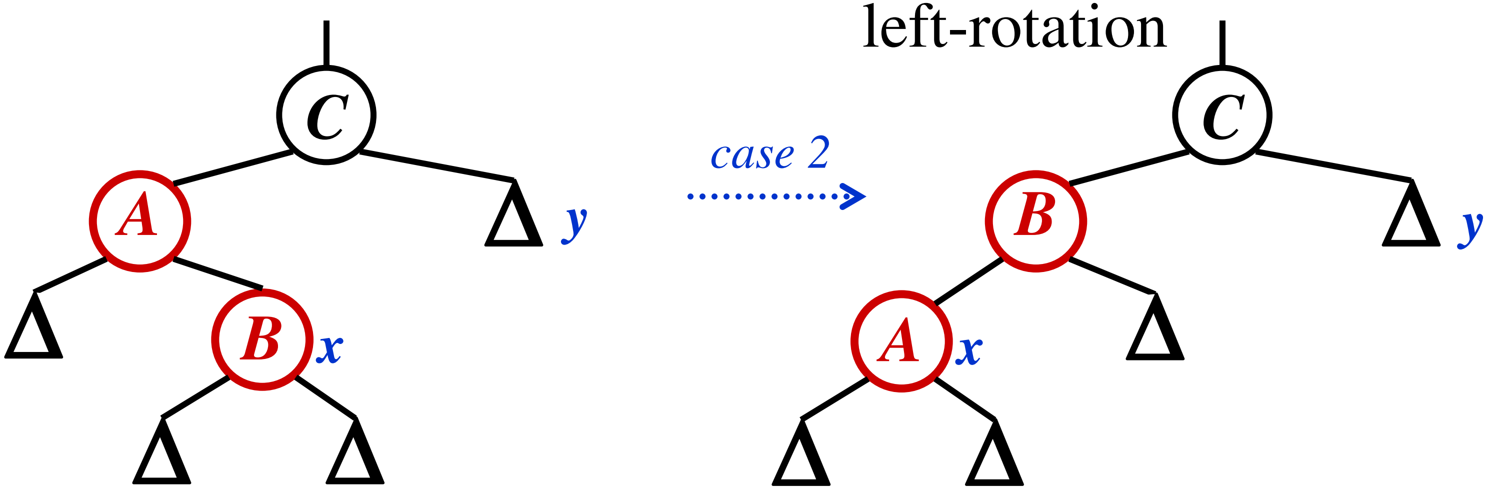


Change colors of some nodes, preserving #4: all downward paths have equal b.h.
The while loop now continues with x's grandparent as the new x

Review: RB Insert: Case 2

```
if (x == x->p->right)
    x = x->p;
    leftRotate(x);
// continue with case 3 code
```

- Case 2:
  - "Uncle" is black
  - Node *x* is a right child
- Transform to case 3 via a left-rotation

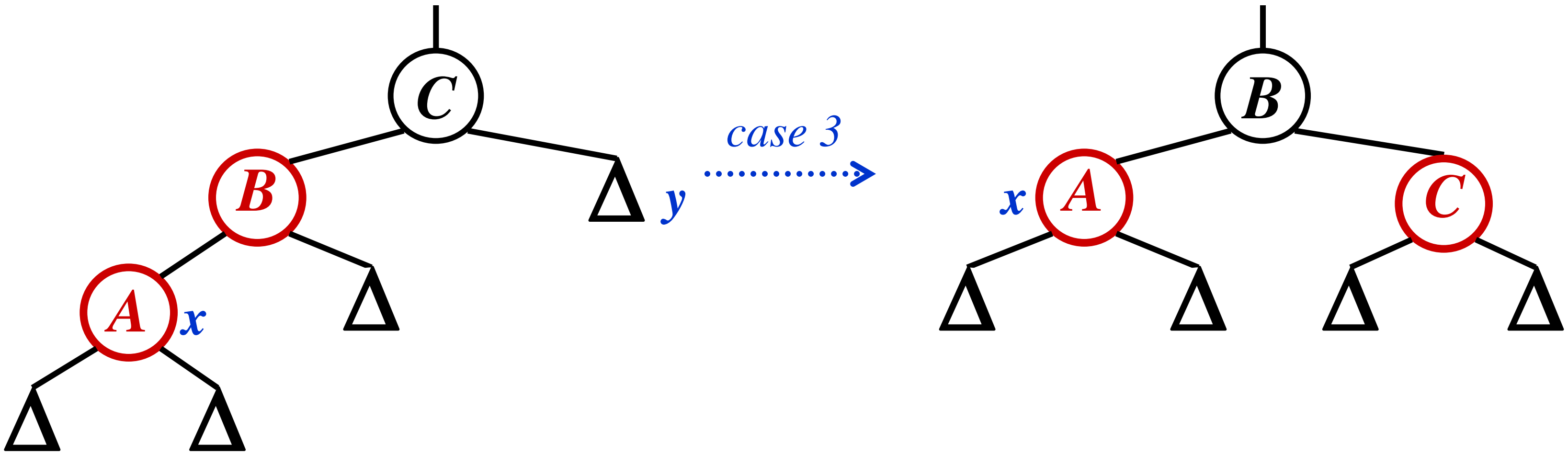


Transform case 2 into case 3 (x is left child) with a left rotation
This preserves property 4: all downward paths contain same number of black nodes

Review: RB Insert: Case 3

```
x->p->color = BLACK;
x->p->p->color = RED;
rightRotate(x->p->p);
```

- Case 3:
  - "Uncle" is black
  - Node *x* is a left child
- Change colors; rotate right



Perform some color changes and do a right rotation
Again, preserves property 4: all downward paths contain same number of black nodes

Red-Black Trees

- Red-black trees do what they do very well
- *What do you think is the worst thing about red-black trees?*
- A: coding them up

Skip Lists

- A relatively recent data structure
  - "A probabilistic alternative to balanced trees"
  - A randomized algorithm with benefits of r-b trees
    - O(lg n) expected time for Search, Insert
    - O(1) time for Min, Max, Succ, Pred
  - *Much* easier to code than r-b trees
  - Fast!

Linked Lists

- Think about a linked list as a structure for dynamic sets. What is the running time of:
  - **`Min()`** *and* **`Max()`** *?*
  - **`Successor()`** *?*
  - **`Delete()`** *?*
    - *How can we make this O(1)?*
  - **`Predecessor()`** *?*
  - **`Search()`** *?*
  - **`Insert()`** *?*

So these all take O(1) time in a linked list. Can you think of a way to do these in O(1) time in a red-black tree?

Goal: make these O(lg n) time in a linked-list setting