Animations

- BST: http://babbage.clarku.edu/~achou/cs160/examples/bst_animation/BST-Example.html
- Rotations: http://babbage.clarku.edu/~achou/cs160/examples/bst_animation/index2.html
- RB-Trees: http://babbage.clarku.edu/~achou/cs160/examples/bst_animation/RedBlackTree-Example.html

Binary Search Trees

- TreeSearch(x, k) // pg 257

```
// Search for key k in tree rooted at x
      if ((x = NIL) or (k = key[x]))
            return x
      if (k < key[x])
            return TreeSearch(left[x], k)
      else
            return TreeSearch(right[x], k)
```

Binary Search Trees

- <u>TreeInsert</u> (T,z) // pg 261

```
// Insert node z in tree T
        y = NIL
        x = root[T]                   // y follows x down the tree
                                      // when x is NIL, y points to a leaf
        while (x ≠ NIL) do
                y = x
                if (key[z] < key[x])
                        x = left[x]
                        x = right[x]
        p[z] = y
        if (y == NIL)
                root[T] = z
        else if (key[z] < key[y])
                left[y] = z
        else right[y] = z
```

Binary Search Trees

- <u>TreeDelete</u>(T,z) //page 262

```
// delete node z in tree T
      if (left[z] == NIL) or (right[z] == NIL) then
            y = z
      else  y = TreeSuccessor(z)           // y has at most 1 child
      if (left[y] ≠ NIL) then
            x = left[y]
      else  x = right[y]                   // x points to a child of y
      if (x ≠ NIL) then
            p[x] = p[y]
      if (p[y] == NIL) then
            root[T] = x
      else  if (y == left[p[y]]) then
                  left[p[y]] = x
            else  right[p[y]] = x
      if (y ≠ z) then
            key[z] = key[y]
            copy y's data into z
      return y
```

Red-Black Trees

- RB-Insert (T,z) // pg 280

```
// Insert node z in tree T
// First BST-Insert
y = NIL
x = root[T]
while (x ≠ NIL) do
        y = x
        if (key[z] < key[x])
                x = left[x]
                x = right[x]
p[z] = y
if (y == NIL)
        root[T] = z
else if (key[z] < key[y])
        left[y] = z
else right[y] = z
left[z] = NIL[T]
right[z] = NIL[T]
color[z] = RED
RB-Insert-Fixup (T,z)
```

RB-Insert-Fixup (T,z) //page 281

```
while (color[p[z]] == RED) do
        if (p[z] = left[p[p[z]]]) then
                y = right[p[p[z]]]
                if (color[y] == RED) then        // C-1
                        color[p[z]] = BLACK
                        color[y] = BLACK
                        z = p[p[z]]
                else    if (z == right[p[z]]) then // C-2
                                z = p[z]
                                LeftRotate(T,z)
                        color[p[z]] = BLACK       // C-3
                        color[p[p[z]]] = RED
                        RightRotate(T,p[p[z]])
        else
                // Symmetric code: "right" ↔ "left"
                . . .
color[root[T]] = BLACK
```

Rotations

- LeftRotate(T,x) // pg 278

```
// right child of x becomes x's parent.
// Subtrees need to be readjusted.
y = right[x]
right[x] = left[y]     // y's left subtree becomes x's right
p[left[y]] = x
p[y] = p[x]
if (p[x] == NIL[T]) then
        root[T] = y
else if (x == left[p[x]]) then
        left[p[x]] = y
else right[p[x]] = y
left[y] = x
p[x] = y
```

Augmented Data Structures

- Why is it needed?
  - Because basic data structures not enough for all operations
  - storing extra information helps execute special operations more efficiently.
- Can any data structure be augmented?
  - Yes. Any data structure can be augmented.
- Can a data structure be augmented with any additional information?
  - Theoretically, yes.
- How to choose which additional information to store.
  - Only if we can maintain the additional information efficiently under all operations. That means, with additional information, we need to perform old and new operations efficiently maintain the additional information efficiently.

New Operations on RB-Trees

- Basic operations
  - RB-Search, RB-Insert, RB-Delete
- New Operations
  - Rank(T,x)
  - Select(T,k)
  - NO EFFICIENT WAY TO IMPLEMENT THEM!
  - Unless more information is stored in each node!
- What information to be added in each node?
  - Rank information
    - Very useful but hard to maintain under Insert/Delete
  - Size information
    - Useful and easy to maintain under Insert/Delete

How to augment data structures

1. choose an underlying data structure
2. determine additional information to be maintained in the underlying data structure,
3. develop new operations,
4. verify that the additional information can be maintained for the modifying operations on the underlying data structure.

RB-Tree Augmentation

- Augment x with **Size(x)**, where
  - Size(x) = size of subtree rooted at x
  - Size(NIL) = 0

OS-Select

```
OS-SELECT(x,i) //page 304
// Select the node with rank i
// in the subtree rooted at x
1.  r ← size[left[x]]+1
2.  if i = r then
3.          return x
4.  elseif  i < r then
5.          return OS-SELECT (left[x], i)
6.  else    return OS-SELECT (right[x], i-r)
```

OS-Rank

```
OS-RANK(x,y)
// Different from text (recursive version)
// Find the rank of y in the subtree rooted at x
  r = size[left[y]] + 1
  if x = y then return r
  else if ( key[x] < key[y] ) then
       return OS-RANK(x,left[y])
  else return r + OS-RANK(x,right[y] )
```

Time Complexity O(log n)

Augmenting RB-Trees

Theorem 14.1, page 309

Let f be a field that augments a red-black tree T with n nodes, and f(x) can be computed using only the information in nodes x, left[x], and right[x], including f[left[x]] and f[right[x]].

Then, we can maintain f(x) during insertion and deletion without asymptotically affecting the O(lgn) performance of these operations.

For example,

size[x] = size[left[x]] + size[right[x]] + 1

rank[x] = ?

Examples of augmenting information for RB-Trees

- Parent
- Height
- Any associative function on all previous values or all succeeding values.
- Next
- Previous