Topic 7:
Balanced search trees
Rotate operation
Red-black tree
Augmenting data struct.
Set Operations

- Maximum
- Extract-Max
- Insert
- Increase-key
- Search
- Delete
- Successor
- Predecessor
Motivation

- Using binary search trees,
  - most operations take time $O(h)$ with $h$ being the height of the tree
- Want to keep $h$ small
  - $h = \Theta(\log n)$ the best one can hope for. (why?)
- We can sort the elements, and construct a balanced binary search tree
  - But then how do we maintain it under dynamic operations (insertion / deletion)?
Balanced Search Tree
Un-balanced Search Tree

Diagram of an un-balanced search tree:

- Node 4
  - Node 2
    - Node 3
  - Node 6
  - Node 8
- Node 13
  - Node 9
  - Node 15
  - Node 17
- Node 18
  - Node 20
Goal:

- Maintain a balanced binary search tree with height $\Theta(\lg n)$ such that all operations take $O(\lg n)$
  - Maintain means that the tree is always of height $\Theta(\lg(\#\text{elements}))$ after any operation supported, especially insertion and deletion.

- There are multiple such balanced trees
  - We focus on the so-called Red-black trees
Rotation Operation

- A key operation in maintaining the “balance” of a binary search tree
  - Locally rotate subtree, while maintain binary search tree property
Right Rotation Examples

- Right rotation at 9.

Does it change Binary search tree property?
Left Rotation Examples

- Left rotation at node 3.

What if we then right-rotate at node 6?

Left and right rotations are inverse of each other.
Recall: Transplant

```
function Transplant(T, u, v)
/* Replace subtree rooted at u with subtree rooted at v. */
1  p ← u.parent;
2  if (p = NIL) then T.root ← v;
3  else if (u = p.left) then p.left ← v;
4  else p.right ← v;
5  if (v ≠ NIL) then v.parent ← p;
```
Implementation of Right Rotation

function RightRotate(T, x)
1 y ← x.left;
2 b ← y.right;
3 Transplant(T,x,y);
4 x.left ← b;
5 if (b ≠ NIL) then b.parent ← x;
6 y.right ← x;
7 x.parent ← y;

Running time: \( \Theta(1) \)
Implementation of Left Rotation

function LeftRotate(T, x)
1. y ← x.right;
2. b ← y.left;
3. Transplant(T, x, y);
4. x.right ← b;
5. if (b ≠ NIL) then b.parent ← x;
6. y.left ← x;
7. x.parent ← y;

Running time: \( \Theta(1) \)
Rotation does not change binary search tree property!

- Right rotation
- Left rotation
Rotation to Re-balance

- Apply rotation to node 5

Reduce height to 3!
Rotation to Re-balance

- Apply rotation at node 5

Height not reduced!

Need a double-rotation (first at 16, then at 5)

In general, how and when?
Red-Black Trees

- Compared to an ordinary binary tree, a red-black binary tree:
  - A leaf node is NIL
  - Hence all nodes are either NIL or have exactly two children!
  - Each node will have a color, either red or black
Definition

- A Red-Black tree is a binary tree with the following properties:
  - Every node is either red or black
  - Root is black
  - Every leaf is NIL and is black
  - If a node is red, then both its children are black
  - For each node, all simple paths from this node to its decedent leaves contain same number of black nodes
An Example

- Double nodes are black.

No two consecutive red nodes.
Skipping Leaves
Is This a Red-Black Tree?
Exercise

- Color the following tree to make a valid red-black tree
Given a tree node $x$

- $size(x)$: the total number of internal nodes contained in the subtree rooted at $x$
- $bh(x)$: the number of black nodes on the path from $x$ to leaf (not counting $x$)
Balancing Property of RB-tree

- **Lemma [BN-Bound]:**
  - Let $r$ be the root of tree $T$. Then
  - $size(r) \geq 2^{bh(r)} - 1$.

- **Proof:**
  - Since every root-leaf path has $bh(r)$ black nodes, $T$ contains a complete binary tree of height $bh(r)$ as subtree.
  - For a complete binary tree of height $bh(r)$, its size is $2^{bh(x)} - 1$
  - Hence $size(r) \geq 2^{bh(r)} - 1$
Balancing Property of RB-tree

- **Theorem [RB-Height]**
  - A red-black tree with \( n \) internal nodes has height \( h \leq 2 \log_2(n + 1) \).

- **Proof:**
  - Let \( r \) be the root of this red-black tree
  - Since no red node has a red child, \( bh(r) \geq \frac{h}{2} \)
  - By Lemma [BN-Bound], \( n \geq 2^{bh(r)} - 1 \geq 2^{h/2} - 1 \)
  - Thus, \( h \leq 2 \log_2(n + 1) \)
Implication of the Theorem

In other words, if we can maintain a RB-tree under every operation, then the tree always has $\Theta(\lg n)$ height, and all operations thus all take $O(\lg n)$ time.

How to maintain RB-tree under Operations?
Set Operations

- Maximum
  - Extract-Max
  - Insert
- Increase-key
- Search
  - Delete
- Successor
- Predecessor

Only need to consider Insert / delete
Insertion Example

Insert 24?
Insert 2?
Insert 36?
Red-Black Tree Insert

function RBTreeInsert(T, z)
1  y ← RBLocateParent(T, z);
2  if (y = NIL) then  T.root ← z;  /* tree T was empty*/
3  else if (z.key < y.key) then  y.left ← z;
4  else y.right ← z;
5  z.left ← leaf;
6  z.right ← leaf;
7  z.color ← Red;
8  RBInsertFixup(T,z);
Insert Fixup: Case 3 (z is the new node)

If the parent of z is red and its “uncle” is black:
If z is a left child and its parent is a left child:

- Right Rotate on the grandparent of z;
- Color the parent of z Black;
- Color the sibling of z red.
Example: Case 3

Insert 12?
Implementation of Fixup Case 3

function RBInsertFixupC(T, alters z)
1 if (z = T.root) or (z.parent.color = Black) then return;
2 x ← z.parent;
3 w ← x.parent;
4 if (z = x.left) and (x = w.left) then
5     RightRotate(T,w);
6     x.color ← Black;
7     w.color ← Red;
8 else if (z = x.right) and (x = w.right) then
9     Handle same as above with “right” and “left” exchanged
10 ...
Remarks

- Running time of RBInsertFixupC
  - $\Theta(1)$

- If the tree has red-black properties if not counting violation caused by $z$
  - Then after RBInsertFixupC the tree is a red-black tree!
  - No more operations needed.
Insert Fixup: Case 2

If the parent $x$ of $z$ is red and its “uncle” is black:
If $z$ is a right child and its parent $x$ is a left child:

- $z \leftarrow z.p$
- Left Rotate on $x$;
- Apply algorithm for Case III.
Example: Case 2

![Binary search tree diagram]

Insert 14?
function RBInsertFixupB(T, alters z)
1  if \( z = T\.root \) or \( z\.parent\.color = \) Black then return;
2  \( x \leftarrow z\.parent \);
3  \( w \leftarrow x\.parent \);
4  if \( z = x\.right \) and \( x = w\.left \) then
5     \( z \leftarrow x \);
6     LeftRotate\( (T, x) \);
7  else if \( z = x\.left \) and \( x = w\.right \) then
8     Handle same as above with “right” and “left” exchanged
9  ...
Insert Fixup: Case 1: (z is new node)

- The parent and uncle of z are red:
  - Color the parent and uncle of z both black
  - Color the grandparent of z red
  - Continue with the grandparent of z
function Sibling(x)
/* Return sibling of x */
1 if (x.parent = NIL) then error "Root has no siblings."
2 p ← x.parent;
3 if (p.left = x) then return (p.right);
4 else return (p.left);

Takes $\Theta(1)$ time
Implementation of Case 1

function RBInsertFixupA(T, alters z)
1. while (z ≠ T.root) and (z.parent.color ≠ Black) do
   2. y ← Sibling(z.parent);
   3. if (y.color = Black) then return;
   4. z.parent.color ← Black;
   5. y.color ← Black;
   6. z ← z.parent.parent;
   7. z.color ← Red;
8. end

Running time: Θ(h) = Θ(lg n)

When does this procedure terminate?
Red-black Tree Insert Fixup

- Function RBInsertFixup (T, z)
  
  RBInsertFixupA (T, z);
  RBInsertFixupB (T, z);
  RBInsertFixupC (T, z);
  
  T.root.color = Black;

- Note z changes after each call.

- Total time complexity:
  
  \( \Theta(\lg n) \)

- Total # rotations:
  
  At most 2
Augmenting Data Structure
Balanced Binary Search Tree

- Maximum
- Extract-Max
- Insert
- Increase-key

Also support Select operation?
Augment Tree Structure

- Select \((T, k)\)
- Goal:
  - Augment the binary search tree data structure so as to support Select \((T, k)\) efficiently

- Ordinary binary search tree
  - \(O(h)\) time for Select\((T, k)\)

- Red-black tree (balanced search tree)
  - \(O(lg n)\) time for Select\((T, k)\)
How To Augment Tree Structure?

- At each node \( x \) of the tree \( T \)
  - store \( x.size = \# \) nodes in the subtree rooted at \( x \)
    - Include \( x \) itself
    - If a node (leaf) is NIL, its size is 0.

- Space of an augmented tree:
  - \( \Theta(n) \)

- Basic property:
  - \( x.size = x.left.size + x.right.size + 1 \)
Example

```
    M/9
   /   \
C/5   P/3
 /   |   |
A/1   F/3   T/2
   /   |   |
D/1   H/1   Q/1
```

CSE 2331/5331
How to Setup Size Information?

- procedure AugmentSize(treenode $x$)
  
  If ($x \neq NIL$) then
  
  $Lsize = \text{AugmentSize}(x\.left);$
  
  $Rsize = \text{AugmentSize}(x\.right);$
  
  $x\.size = Lsize + Rsize + 1;$
  
  Return($x\.size$);

end

Return (0);

Postorder traversal of the tree!
Augmented Binary Search Tree

- Let $T$ be an augmented binary search tree
- $OS-Select(x, k)$:
  - Return the $k$-th smallest element in the subtree rooted at $x$
  - $OS-Select(T.root, k)$ returns the $k$-th smallest elements in the entire tree.

$OS-Select(T.root, 5)$
OS-SELECT(x, i)

\[ r = x\text{.left\_size} + 1 \]
\[ \text{if } i == r \]
\[ \quad \text{return } x \]
\[ \text{elseif } i < r \]
\[ \quad \text{return OS-SELECT(x.left, i)} \]
\[ \text{else return OS-SELECT(x.right, i - r)} \]

- Correctness?
- Running time?
  - O(h)
OS-Rank(\(T, x\))

- Return the rank of the element \(x\) in the linear order determined by an inorder walk of \(T\)
Example

OS-Rank(T, D) ?
OS-RANK(T, x)

\[ r = x.\text{left.size} + 1 \]
\[ y = x \]
\[ \text{while } y \neq T.\text{root} \]
\[ \quad \text{if } y == y.p.\text{right} \]
\[ \quad \quad r = r + y.p.\text{left.size} + 1 \]
\[ y = y.p \]
\[ \text{return } r \]

- Correctness ?
- Time complexity?
  - O(h)
Need to maintain augmented information under dynamic operations

- Insert / delete
  - Extract-Max can be implemented with delete

Are we done?
Example

Insert(J)?
During the downward search, increase the size attribute of each node visited along the path from root to the final insert location.

Time complexity:
- $O(h)$

However, if we have to maintain balanced binary search tree, say Red-black tree
- Also need to adjust size attribute after rotation
- $y.size = x.size$
- $x.size = x.left.size + x.right.size + 1$
- $O(1)$ time per rotation
Right-rotate can be done similarly.

Overall: Two phases:

- Update size for all nodes along the path from root to insertion location
  - $O(h) = O(lg n)$ time
- Update size for the Fixup stage involving $O(1)$ rotations
  - $O(1) + O(lg n) = O(lg n)$ time
- $O(h) = O(lg n)$ time to insert in a Red-Black tree
  - Same asymptotic time complexity as the non-augmented version
Delete

- Two phases:
  - Decrement size in each node on the path from the root to the node to be deleted
    - $O(h) = O(lg n)$ time
  - During Fixup (to maintain balanced binary search tree property), update size for $O(1)$ rotations
    - $O(1) + O(lg n)$ time
- Overall:
  - $O(h) = O(lg n)$ time
Summary

- Simple example of augmenting data structures
- In general, the augmented information can be quite complicated
  - Can be a separate data structure!
- Need to consider how to maintain such information under dynamic changes