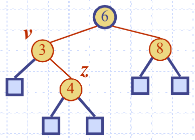
Lecture 9: Red-Black Trees

Perfect Balance and Efficiency



Wholeness Statement

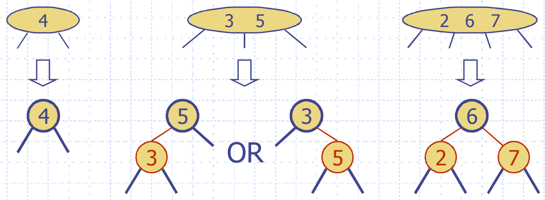
A red-black tree is an implementation of a (2, 4) tree that is optimized for space utilization. The insert and delete operations are also optimized to avoid backtracking; the operations are performed locally yet maintain balance and order in the whole. Nature operates in accord with the law of least action while maintaining balance and order in the whole.

Outline and Reading

- From (2,4) trees to red-black trees (§3.3.3)
- Red-black tree (§ 3.3.3)
  - Definition
  - Height
  - Insertion
    - restructuring
    - recoloring
  - Deletion
    - restructuring
    - recoloring
    - adjustment

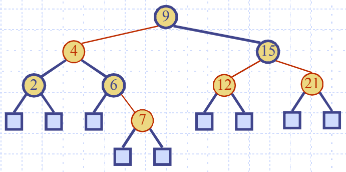
From (2,4) to Red-Black Trees

- A red-black tree is a representation of a (2,4) tree by means of a binary tree whose nodes are colored red or **black**
- In comparison with its associated (2,4) tree, a red-black tree has
  - same logarithmic time performance
  - simpler implementation with a single node type



Red-Black Tree

- A red-black tree can also be defined as a binary search tree that satisfies the following properties:
  - Root Property: the root is black
  - External Property: every leaf is black
  - Internal Property: the children of a red node are black
  - Depth Property: all the leaves have the same black depth



Height of a Red-Black Tree

- Theorem: A red-black tree storing n items has height $O(\log n)$

  Proof:
  - The height of a red-black tree is at most twice the height of its associated (2,4) tree, which is $O(\log n)$
- The search algorithm for a red-black tree is the same as that for a binary search tree
- By the above theorem, searching in a red-black tree takes $O(\log n)$ time

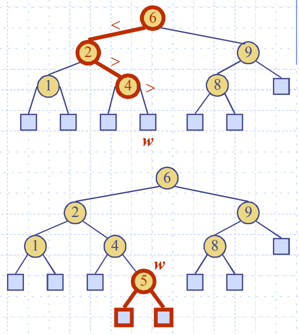
Main Point

1. A red-black tree is an implementation of a (2, 4) tree in which each (2, 4) node is converted into a small binary tree.
   Our individual consciousness is the reflection of pure consciousness in the activity of our physiology.

Red-Black Tree Insertion

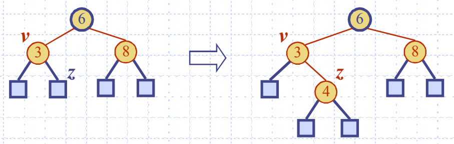
Recall
Binary Tree Insertion (§3.1.4)

- To perform operation insertItem(k, o), search for key k (k should not be in the tree)
- Let w be the leaf reached by the search
- Insert k at node w and expand w into an internal node
- Example: insert 5



Insertion

- To perform operation insertItem(k, o), we execute the insertion algorithm for binary search trees and color the newly inserted node z red unless it is the root. We preserve the root, external, and depth properties.
  - If the parent v of z is black, we preserve the internal property and we are done…
  - …else (v is red) we have a double red (i.e., a violation of the internal property), which requires a reorganization of the tree
- For example, the insertion of 4 causes a double red:

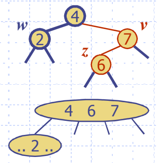


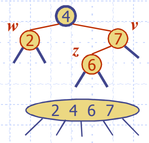
Remedying a Double Red

- Consider a double red with child z and parent v, and let w be the sibling of v

Case 1: z's uncle w is black
- The double red is an incorrect replacement of a 4-node
- Restructuring: we change the 4-node replacement

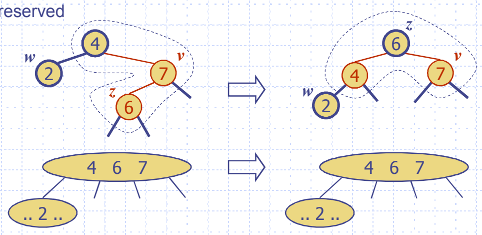
Case 2: z's uncle w is red
- The double red corresponds to an overflow
- Recoloring: we perform the equivalent of a split





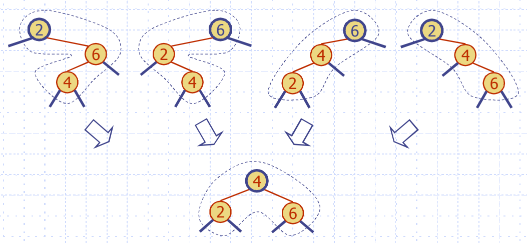
Case 1: Restructuring

- A restructuring remedies a child-parent double red when the parent red node has a black sibling
- It is equivalent to restoring the correct replacement of a 4-node
- The internal property is restored and the other properties are preserved



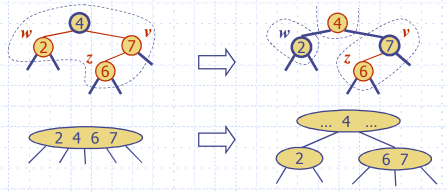
Restructuring (cont.)

- There are four restructuring configurations depending on whether the double red nodes are left or right children



Case 2: Recoloring

- A recoloring remedies a child-parent double red when the parent red node has a red sibling
- The parent *v* and its sibling *w* become black and the grandparent *u* becomes red, unless it is the root
- It is equivalent to performing a split on a 5-node
- The double red violation may propagate to the grandparent *u*



Example:

- Insert the following into an initially empty red-black tree in this order:

  (16, 5, 22, 45, 2, 10, 18, 30, 50, 12, 1)

Analysis of Insertion

```
Algorithm insertItem(k, o)

1. Search for key k to locate the
   insertion node z

2. Add the new item (k, o) at node
   z and color z red

3. while doubleRed(z)
     if isBlack(sibling(parent(z)))
         z ← restructure(z)
         return
     else { sibling(parent(z)) is red }
         z ← recolor(z)
```

- Recall that a red-black tree has $O(\log n)$ height
- Step 1 takes $O($? $)$ time
- Step 2 takes $O($? $)$ time
- Step 3 takes $O($? $)$ time
- Thus, an insertion in a red-black tree takes $O($? $)$ time

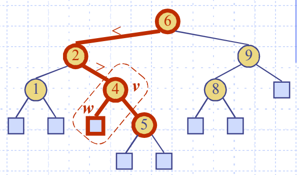
Analysis of Insertion

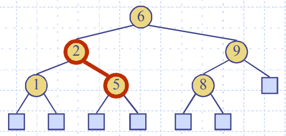
```
Algorithm insertItem(k, o)

1. Search for key k to locate the
   insertion node z

2. Add the new item (k, o) at node
   z and color z red

3. while doubleRed(z)
     if isBlack(sibling(parent(z)))
         z ← restructure(z)
         return
     else { sibling(parent(z)) is red }
         z ← recolor(z)
```

- Recall that a red-black tree has $O(\log n)$ height
- Step 1 takes $O(\log n)$ time because we visit $O(\log n)$ nodes
- Step 2 takes $O(1)$ time
- Step 3 takes $O(\log n)$ time because we perform
  - $O(\log n)$ recolorings, each taking $O(1)$ time, and
  - at most one restructuring taking $O(1)$ time
- Thus, an insertion in a red-black tree takes $O(\log n)$ time

Recall
Binary Tree Deletion (§3.1.5)

- To perform operation removeElement(*k*), first search for key *k*
- Assume key *k* is in the tree, and let *v* be the node storing *k*
- Two cases:
  - Node *v* has a leaf child *w*
  - Node *v* has no leaf child
- If node *v* has a leaf child *w*, we remove *v* and *w* from the tree with operation removeAboveExternal(*w*)
- Example: remove 4

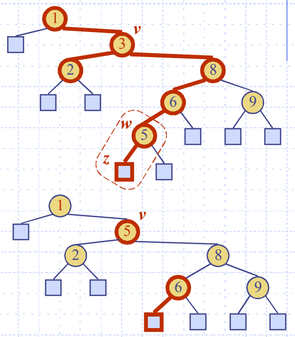




Red-Black Tree Deletion

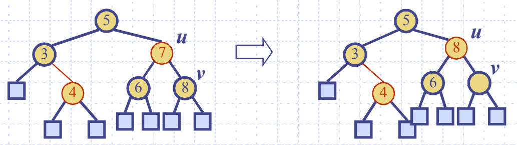
Binary Tree Deletion (cont.)

- Suppose the key k to be removed is stored at a node v whose children are both internal
  - we find the internal node w that follows v in an in-order traversal
  - we copy $key(w)$ into node v
  - we remove node w and its left child z (which must be a leaf) by means of operation removeAboveExternal(z)
- Example: remove 3



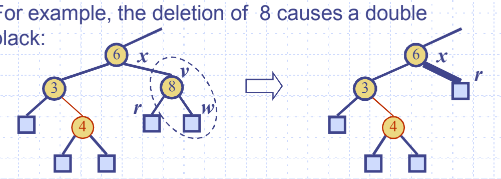
Deletion

- To perform operation remove(k), first execute the deletion algorithm for binary search trees
  - If node to be removed, *u*, does not have an external child, find next internal node by inorder traversal, called *v*, and move key at *v* to *u*, then remove *v*.
  - Thus, every removal occurs at an internal node with an external child.



Deletion

- Let v be the internal node removed, w the external node removed, and r the sibling of w
  - If either v or r was red, we color r black and we are done
  - Else (v and r were both black), so we color r ***double black*** (a fictitious color), which is a violation of the internal property requiring a reorganization of the tree (denotes underflow)
- For example, the deletion of 8 causes a double black:



Remedying a Double Black

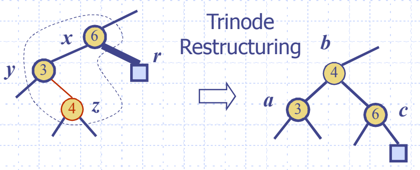
- The algorithm for remedying a double black node r with sibling y considers three cases

Case 1: y is black and has a red child z

Case 2: y is black and its children are both black

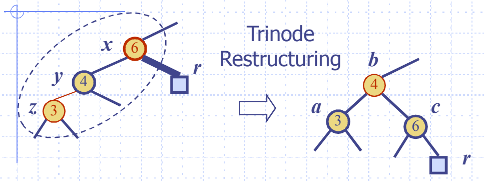
Case 3: y is red

Case 1a – y is black and has a red child z



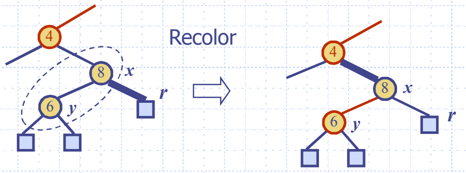
Color a and c black, give b the former color of x, and set r to black

Case 1b – y is black and has a red child z



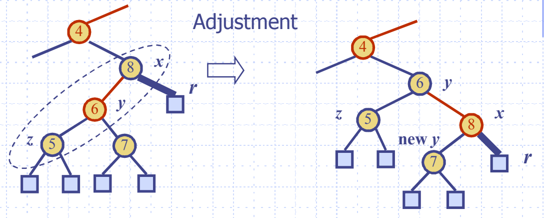
Color a and c black, give b the former color of x, and set r to black

Case 2 – y is black and its children are both black



Color r black and y red;
if x is red, color x black else color x double black

Case 3 – y is red



Execute the trinode restructure operation that makes y the parent of x;
Color y black and x red;

Then apply Case 1 or 2, with new y.

Remedying a Double Black

- The algorithm for remedying a double black node r with sibling y considers three cases
  - Case 1: y is black and has a red child
    - Perform a restructuring, equivalent to a transfer, and we are done
  - Case 2: y is black and its children are both black
    - Perform a recoloring, equivalent to a fusion, which may propagate the double black violation up to parent
  - Case 3: y is red
    - We perform an adjustment, equivalent to choosing a different representation of a 3-node, after which either Case 1 or Case 2 applies
- Deletion in a red-black tree takes $O(\log n)$ time

Main Point

2. Restoring balance after insertion or deletion in a red-black tree only requires a constant number of trinode restructurings (0, 1, or 2) and at most O(log n) recolorings. The algorithm is slightly more complicated than for a (2,4) tree, but the data structure has a major advantage in space requirements.
   The TM technique is a simple, effortless technique for restructuring the physiology to a more balanced state.

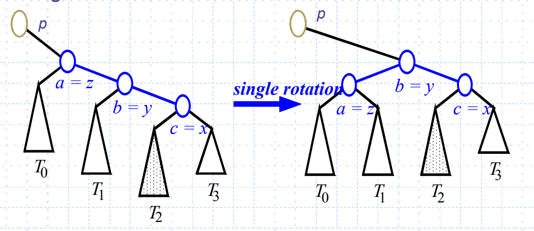
Red-Black Tree Reorganization

Insertion	remedy double red	
Red-black tree action	**(2,4) tree action**	**result**
restructuring	change of 4-node representation	double red removed
recoloring	split	double red removed or propagated up

Deletion	remedy double black	
Red-black tree action	**(2,4) tree action**	**result**
restructuring	transfer	double black removed
recoloring	fusion	double black removed or propagated up
adjustment	change of 3-node representation	restructuring or recoloring follows

Restructuring (as Single Rotations)

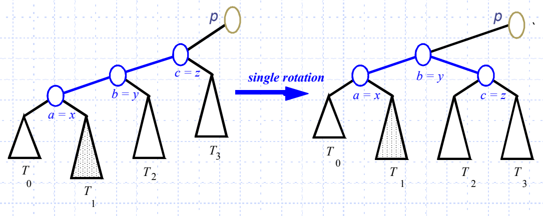
- Single Rotations:



Left Rotation

```
Algorithm rotateLeft(T, y)
    Input Binary Tree T and node y in T
    Output a left rotation around node y is performed

    if T.isRoot(y) then throw InvalidLeftRotation
    z ← T.parent(y)
    p ← T.parent(z)

    T.setRightChild(z, T.leftChild(y))
    T.setParent(T.leftChild(y), z)

    T.setLeftChild(y, z)
    T.setParent(z, y)

    if T.isRoot(z)
        then T.setRoot(y)
        else
            if T.rightChild(p) = z
            then T.setRightChild(p, y)
            else T.setLeftChild(p, y)

            T.setParent(y, p)
```

Exercise: Do Right Rotation

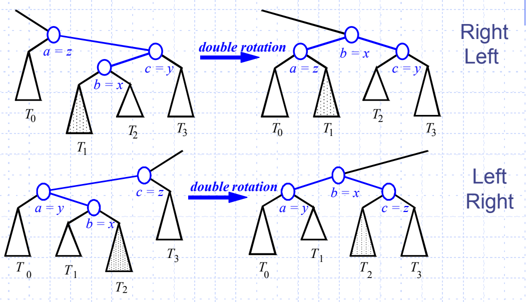


Right Rotation

```
Algorithm rotateRight(T, y)
    Input Binary Tree T and node y in T
    Output a right rotation around node y is performed

    if T.isRoot(y) then throw InvalidRightRotation
    z ← T.parent(y)
    p ← T.parent(z)

    T.setLeftChild(z, T.rightChild(y))
    T.setParent(T.rightChild(y), z)

    T.setRightChild(y, z)
    T.setParent(z, y)

    if T.isRoot(z)
        then T.setRoot(y)
        else
            if T.rightChild(p) = z
            then T.setRightChild(p, y)
            else T.setLeftChild(p, y)

            T.setParent(y, p)
```

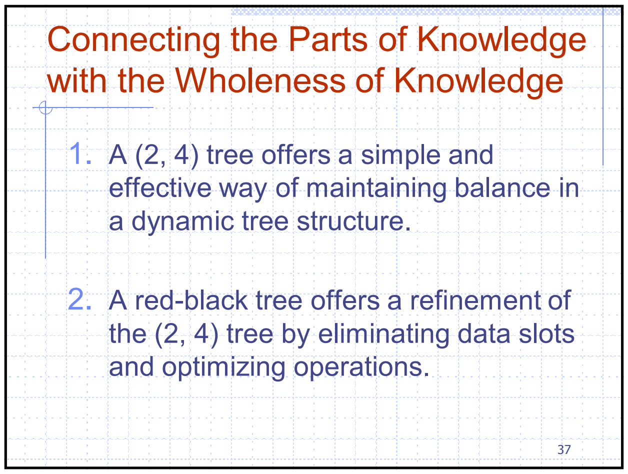
Restructuring (as Double Rotations)

- double rotations:



Main Point

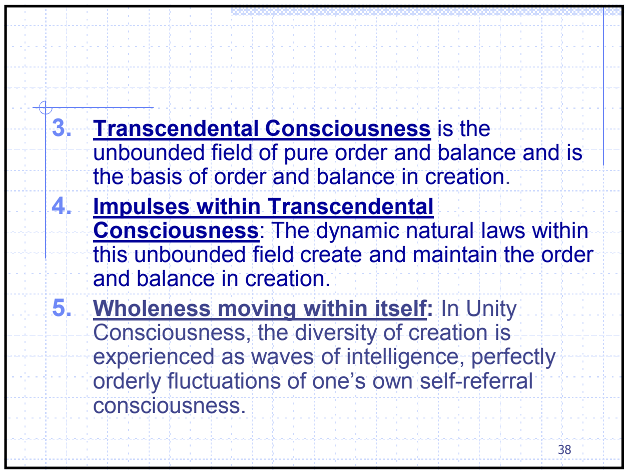
3. A red-black tree is an efficient way to implement an ordered dictionary ADT because it achieves logarithmic worst-case running times for both searching and updating.
   The TM technique is a very simple, effortless way to facilitate contact with the field of total knowledge, where the fulfillment of intellectual study is achieved, i.e., one feels at home with everything and everyone.



Connecting the Parts of Knowledge with the Wholeness of Knowledge

1. A (2, 4) tree offers a simple and effective way of maintaining balance in a dynamic tree structure.

2. A red-black tree offers a refinement of the (2, 4) tree by eliminating data slots and optimizing operations.



3. **Transcendental Consciousness** is the unbounded field of pure order and balance and is the basis of order and balance in creation.
4. **Impulses within Transcendental Consciousness**: The dynamic natural laws within this unbounded field create and maintain the order and balance in creation.
5. **Wholeness moving within itself:** In Unity Consciousness, the diversity of creation is experienced as waves of intelligence, perfectly orderly fluctuations of one's own self-referral consciousness.