Acknowledgments

• The course follows the book “Introduction to Algorithms“, by Cormen, Leiserson, Rivest and Stein, MIT Press [CLRST]. Many examples displayed in these slides are taken from their book.

• These slides are based on those developed by Michael Böhlen for this course.

(See http://www.inf.unibz.it/dis/teaching/DSA/)

• The slides also include a number of additions made by Roberto Sebastiani and Kurt Ranalter when they taught later editions of this course.

(See http://disi.unitn.it/~rseba/DIDATTICA/dsa2011_BZ//)
DSA, Chapter 6: Overview

- Binary Search Trees
  - Tree traversals
  - Searching
  - Insertion
  - Deletion

- Red-Black Trees
  - Properties
  - Rotations
  - Insertion
  - Deletion
DSA, Chapter 6: Overview

- **Binary Search Trees**
  - Tree traversals
  - Searching
  - Insertion
  - Deletion

- **Red-Black Trees**
  - Properties
  - Rotations
  - Insertion
  - Deletion
Dictionaries

A **dictionary** $D$ is a dynamic data structure containing elements with a **Key** and an **Element** field.

A dictionary object allows the operations:

- **Element** `find(Key k)`
  
  *returns (a pointer to) an element $x$ in Element with key = $k$*
  
  *(and otherwise returns null or throws an exception)*

- **void** `insert(Key k, Element x)`

  *adds the data element $x$ with the key $k$*

- **void** `delete(Key k)`

  *removes the element stored with key $k$*
Ordered Dictionaries

A dictionary $D$ may have keys that are comparable (ordered domain)

In addition to the standard dictionary operations, we want to support the operations:

- Key min(), Key max()
- Key floor(), Key ceil()

and

- Key predecessor(Key k)
- Key successor(Key k)
A List-based Implementation

Unordered list
- find, min, max, predecessor, successor: $O(n)$
- insert, delete: $O(1)$

Ordered list
- search, insert: $O(n)$
- min, max, predecessor, successor, delete: $O(1)$

What kind of list is needed to allow for $O(1)$ deletions?
Refresher: Binary Search

Narrow down the search range in stages
- `find(22)`
Run Time of Binary Search

• The range of candidate items to be searched is halved after comparing the key with the middle element
  ➔ binary search on arrays runs in $O(\log n)$ time
• What about insertion and deletion?
  – find: $O(\log n)$
  – min, max, predecessor, successor: $O(1)$
  – insert, delete: $O(n)$
• Challenge: implement insert and delete in $O(\log n)$
• Idea: extended binary search to dynamic data structures
  ➔ binary trees
In what follows we ignore the `Element` field of nodes.
Binary Search Trees

A binary search tree (BST) is a binary tree $T$ with the following properties:

- each internal node stores an item $(k,e)$ of a dictionary
- keys stored at nodes in the left subtree of $x$ are less than or equal to $k$
- keys stored at nodes in the right subtree of $x$ are greater than or equal to $k$

Example BSTs for 2, 3, 5, 5, 7, 8
DSA, Chapter 6: Overview

- Binary Search Trees
  - Tree traversals
  - Searching
  - Insertion
  - Deletion

- Red-Black Trees
  - Properties
  - Rotations
  - Insertion
  - Deletion
Tree Walks

Keys in a BST can be printed using “tree walks”

Option 1: Print the keys of each node between the keys in the left and right subtree

⇒ inorder tree traversal

```python
inorderTreeWalk(Node node)
if node ≠ NULL then
    inorderTreeWalk(node.left)
    print node.key
    inorderTreeWalk(node.right)
```
Tree Walks/2

• inorderTreeWalk is a divide-and-conquer algorithm
• It prints all elements in monotonically increasing order
• Running time $\Theta(n)$
inorderTreeWalk can be thought of as a projection of the BST nodes onto a one-dimensional interval
Other Forms of Tree Walk

A preorder tree walk processes each node before processing its children

```java
preorderTreeWalk(Node node)
    if node ≠ NULL then
        print node.key
        preorderTreeWalk(node.left)
        preorderTreeWalk(node.right)
```

... is used for copying trees
Other Forms of Tree Walk/2

A postorder tree walk processes each node after processing its children

```java
postorderTreeWalk(Node node)
if node != NULL then
    postorderTreeWalk(node.left)
    postorderTreeWalk(node.right)
    print node.key
```

... corresponds to evaluation of trees with values on leaves and operators on inner nodes
**DSA, Chapter 6: Overview**

- **Binary Search Trees**
  - Tree traversals
  - Searching
  - Insertion
  - Deletion

- **Red-Black Trees**
  - Properties
  - Rotations
  - Insertion
  - Deletion
Search Examples

• find(11)
Search Examples/2

- find(6)
Pseudocode for BST Search

Recursive version: divide-and-conquer

```
Node find(int k)
    return find(root, k)

Node find(Node n, int k)
    if n = NULL or n.key = k
        then return n
    if k < n.key
        then return find(n.left, k)
    else return find(n.right, k)
```
Pseudocode for BST Search

Iterative version

Node find(int k)
    return find(root, k)

Node find(Node n, int k)
    curr := n
    while curr ≠ NULL and curr.key ≠ k do
        if k < curr.key
            then curr := curr.left
        else curr := curr.right
    return curr

What is the loop invariant here?
Analysis of Search

• Running time on a tree of height $h$ is $O(h)$
• After the insertion of $n$ keys, the worst-case running time of searching is $O(n)$
Searching a BST

To find an element with key $k$ in the tree rooted at node $n$
- compare $k$ with $n.key$
- if $k < n.key$, search for $k$ in $n.left$
- otherwise, search for $k$ in $n.right$
A call \texttt{find}(k) returns one node with key $k$.

If the tree contains several such nodes, it returns the node at the lowest level (i.e., highest up).

Alternatively, we may want the leftmost node (wrt inorder traversal) with key $k$.

Starting from that node, we can retrieve all nodes with key $k$ by iteratively through the successors wrt inorder traversal (provided we have a method to do so).
Finding the First Node with a Given Key

Idea: Keep the leftmost node with key $k$ found so far as a candidate

```java
Node findFirst(int k)
    return findFirst(root, k, null)

Node findFirst(Node n, int k, Node cand)
    if n = null
        then return cand
    elsif k = n.key
        then return findFirst(n.left, k, n)
    elsif k < n.key
        then return findFirst(n.left, k, cand)
    else return findFirst(n.right, k, cand)
```

Why does this work?
Correctness of `findFirst`

The call

\[ \text{findFirstAux}(\text{Node } n, \text{ int } k, \text{ Node cand}) \]

returns

- the leftmost node with key \( k \) in the subtree rooted at \( n \), if there is such a node
- \( \text{cand} \) otherwise

This follows by induction over the structure of trees ...
Correctness of `findFirst/2`

Induction, base case:
If the tree rooted at n is empty, there is no node with key k. The method has to return `cand`, which it does.

Inductive step:
If the tree rooted at n is non-empty, there are three cases:

- k = n.key
- k < n.key
- k > n.key

In the first case, the call returns the leftmost occurrence of k in the subtree rooted at n.left, if there is one (induction hypothesis), otherwise, it returns n. That is, if there is an occurrence to the left of n, then that is returned, otherwise, n is returned.

In the other cases, a similar argument holds.
DSA, Chapter 6: Overview

- Binary Search Trees
  - Tree traversals
  - Searching
  - Insertion
  - Deletion

- Red-Black Trees
  - Properties
  - Rotations
  - Insertion
  - Deletion
BST Insertion Example

insert(8)
BST Insertion

The basic idea derives from searching:

- **construct** a node \( n \)
  whose left and right children are NULL
  and insert it into the tree

- **find the location** in the tree
  where \( n \) belongs to
  (as if searching for \( n.key \)),

- **add** \( n \) there

Be careful: When searching, remember the previous node, because the current node will end up being NULL

The running time on a tree of height \( h \) is \( O(h) \)
void insert(int k)
    Node n := new Node(k)
    if root = NULL
        then root := n
    else insertAux(k, n, root, NULL)

void insertAux(int k, Node n, Node curr, Node prev)
    if curr = NULL then
        n.parent := prev
        if k < prev.key
            then prev.left := n
        else prev.right := n
    if k < curr.key
        then insertAux(k, n, curr.left, curr)
    else insertAux(k, n, curr.right, curr)
### BST Insertion: Iterative Version

```java
void insert(int k) {
    Node n := new Node(k)
    if root = null then root := n
    else {
        curr := root
        prev := null
        while curr != null do {
            prev := curr
            if k < curr.key then curr := curr.left := curr.right
            n.parent := prev
            if k < prev.key then prev.left := n
            else prev.right := n
        }
    }
}
```
BST Insertion: Worst Case

In which order must the insertions be made to produce a BST of height $n$?
BST Sorting/2

Sort the numbers

5 10 7 1 3 1 8

- Build a binary search tree

- Call inorderTreeWalk

1 1 3 5 7 8 10
BST Sorting

Sort an array $A$ of $n$ elements using insert and a version of inorderTreeWalk that inserts node keys into an array (instead of printing them)

```java
void treeSort(A)
    T := new Tree() // a new empty tree
    for i := 1 to A.length do
        T.insert(A[i])
    T.inorderTreeWalkPrintToArray(A)
```

We assume a constructor

```java
Tree() that produces an empty tree
```
Printing a Tree onto an Array

Tricky, because we do not know where to print the root ...

```c
void inorderTreeWalkPrintToArray(A)
  ioAux(root,A,1)

int ioAux(Node n, A, int start)
  // starts to print at position start
  // reports where to continue printing
  if n = NULL then
    return start
  else
    nodePos := ioAux(n.left, A, start)
    A[nodePos] := n.key
    return ioAux(n.right, A, nodePos+1)
```
DSA, Chapter 6: Overview

- Binary Search Trees
  - Tree traversals
  - Searching
  - Insertion
  - Deletion

- Red-Black Trees
  - Properties
  - Rotations
  - Insertion
  - Deletion
BST Minimum (Maximum)

Find the node with the minimum key in the tree rooted at node \( x \)

- That is, the leftmost node in the tree, which can be found by walking down along the left child axis as long as possible

```plaintext
minNode(Node n)
while n.left ≠ NULL do
  n := n.left
return n
```

- Maximum: walk down the right child axis, instead
- Running time is \( O(h) \), i.e., proportional to the height of the tree.
Successor

Given node $x$, find the node with the smallest key greater than $x$.key

- We distinguish two cases, depending on the right subtree of $x$

- **Case 1:** The right subtree of $x$ is non-empty (succ($x$) inserted after $x$)
  - successor is the minimal node in the right subtree
  - found by returning minNode($x$.right)
Successor/2

- **Case 2:** the right subtree of $x$ is empty
  (succ($x$), if any, was inserted before $x$)
  - The successor (if any) is the lowest ancestor of $x$
    whose left subtree contains $x$

- Can be found by tracing parent pointers until the current node is the left child of its parent:
  return the parent
Successor Pseudocode

\[
\text{successor}(\text{Node } x) \\
\text{if } x.\text{right} \neq \text{NULL} \text{ then return } \text{minNode}(x.\text{right}) \\
y := x \\
\text{while } y.\text{parent} \neq \text{NULL and } y = y.\text{parent}.\text{right} \\
\hspace{1cm} y := y.\text{parent} \\
\text{return } y.\text{parent}
\]

For a tree of height \( h \), the running time is \( O(h) \)

*Note: no comparison among keys needed, since we have parent pointers!*
Successor with Trailing Pointer

Idea: Introduce yp to avoid derefencing y.parent

```plaintext
successor(Node x)
    if x.right ≠ NULL
        then return minNode(x.right)
    y := x
    yp := y.parent
    while yp ≠ NULL and y = yp.right do
        y := yp
    yp := y.parent
    return yp
```
Deletion

Delete node $x$ from a tree $T$

We distinguish three cases
- $x$ has no child
- $x$ has one child
- $x$ has two children
Deletion Case 1

If $x$ has no children:
   make the parent of $x$ point to $\text{NULL}$
   ($x$ will be removed by the garbage collector)
Deletion Case 2

If $x$ has exactly one child:
make the parent of $x$ point to that child

```
        x
       / \
      A   B
     /
    D
     /
    F
```

```
        D
       /\  \
      B   F
```

Deletion Case 3

• If $x$ has two children:
  – find the largest child $y$ in the left subtree of $x$ (i.e., $y$ is predecessor($x$))
  – recursively remove $y$ (note that $y$ has at most one child), and
  – replace $x$ with $y$.

• “Mirror” version with successor($x$) [CLRS]
The Logic of Deletion

• One node is dropped
  – $n$, if it has at most one child, otherwise, $\text{successor}(n)$
Call the node to be dropped: drop

• One node is (possibly) kept, the child of drop: keep

• Node keep takes on the child role of drop
  – drop’s parent becomes keep’s parent
  – if drop is a left/right child of its parent,
    then keep becomes a left/right child
  – if drop has no parent, it becomes the root

• If $\text{successor}(n)$ is dropped instead of $n$,
  then $\text{successor}(n)$’s content is copied to $n$

• For trees without parent pointers,
  we have to find the parent of drop
BST Deletion Pseudocode

```c
void delete(Node n)
    if n.left = NULL or n.right = NULL
        then drop := n
    else drop := successor(n)
    if drop.left ≠ NULL
        then keep := drop.left
    else keep := drop.right
    if keep ≠ NULL
        then keep.parent := drop.parent
    if drop.parent = NULL
        then root := keep
    else if drop = drop.parent.left
        then drop.parent.left := keep
    else drop.parent.right := keep
    if drop ≠ n
        then n.key := drop.key
    //   n.data := drop.data
```
Avoid Copying

• Instead of copying the content of successor(n) into n, we can replace n with successor(n). After that, we have to restructure the tree.

• There are two cases:
  – successor(n) = n.right, or
  – successor(n) != n.right

Note that always successor(n).left = NULL

• First case:
  – successor(n).left := n.left

• Second case:
  – parent(successor(n)).left := successor(n).right
  – successor(n).right := n.right
BST Deletion Code (Java)

- Java method for class Tree
- Version without “parent” field
- Note the trailing pointer technique

```java
void delete(Node n) {
    front = root; rear = NULL;
    while (front != n) {
        rear := front;
        if (n.key < front.key)
            front := front.left;
        else
            front := front.right;
    }  // rear points to the parent of n (if it exists)
    ...
```

- Java method for class Tree
- Version without “parent” field
- Note the trailing pointer technique
BST Deletion Code (Java)/2

- n has less than 2 children
- fix pointer of parent of n

```java
... 
if (n.right == NULL) {
    if (rear == NULL) root = n.left;
    else if (rear.left == n) rear.left = n.left;
    else rear.right = n.left;
} 
else if (n.left == NULL) {
    if (rear == NULL) root = n.right;
    else if (rear.left == n) rear.left = n.right;
    else rear.right = n.right;
} else {
    ...
    ...
... 
```
BST Deletion Code (Java)/3

• n has 2 children

```java
succ = n.right; srear = n.right;
while (succ.left != NULL)
    { srear:=succ; succ:=succ.left; }
if (rear == NULL) root = succ;
else if (rear.left == n) rear.left = succ;
else rear.right = succ;
succ.left = n.left;
if (srear != succ) { 
    srear.left = succ.right;
    succ.right = n.right;
}
```
Balanced Binary Search Trees

• Problem: execution time for tree operations is \( \Theta(h) \), which in worst case is \( \Theta(n) \)
• Solution: balanced search trees guarantee small height \( h = O(\log n) \)
Suggested Exercises

Implement a class of binary search trees with the following methods:

- max, min, successor, predecessor
- find (iterative & recursive), insert
- count (returns number of nodes)
- sum (returns sum of keys)
- minLeafDepth (returns minimal depth of a null leaf)
- maxLeafDepth
- delete (swap with successor and predecessor)
- print, print in reverse order
- treeSort
Suggested Exercises/2

Develop methods that compute the following:

• sum of all keys
• average of all keys
• the maximum/minimum of all keys (provided the tree is nonempty)

For trees without parent pointers, develop methods that compute the parent of a node for the two cases that

• the keys are unique and the tree is a BST
• the tree is not a BST
Suggested Exercises/3

Develop methods that compute the following:

• the deepest node
  (i.e., the node with the longest path from the root)
• the leftmost deepest node, if there are several with the maximal depth

Develop methods that check

• whether a tree is complete (i.e., all levels up to the height of the tree are filled)
• whether a tree is nearly complete (like the heaps in Heapsort)
Suggested Exercises/3

Using paper & pencil:

• Draw the trees after each of the following operations, starting from an empty tree:
  – insert 9, 5, 3, 7, 2, 4, 6, 8, 13, 11, 15, 10, 12, 16, 14
  – delete 16, 15, 5, 7, 9
    (both with successor and predecessor strategies)

• Simulate the following operations after the above:
  – Find the max and minimum
  – Find the successor of 9, 8, 6
DSA, Chapter 6: Overview

– Binary Search Trees
  • Tree traversals
  • Searching
  • Insertion
  • Deletion

– Red-Black Trees
  • Properties
  • Rotations
  • Insertion
  • Deletion
Chapter 6

Binary Search Trees

Java's TreeMap

public class TreeMap<K,V>
extends AbstractMap<K,V>
implements NavigableMap<K,V>, Cloneable, Serializable

A Red-Black tree based NavigableMap implementation. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.

This implementation provides guaranteed log(n) time cost for the containsKey, get, put and remove operations. Algorithms are adaptations of those in Cormen, Leiserson, and Rivest's Introduction to Algorithms.
DSA, Chapter 6: Overview

– Binary Search Trees
  • Tree traversals
  • Searching
  • Insertion
  • Deletion

– Red-Black Trees
  • Properties
  • Rotations
  • Insertion
  • Deletion
Red/Black Trees

A red-black tree is a binary search tree with the following properties:
1. Nodes are colored red or black
2. NULL leaves are black
3. The root is black
4. No two consecutive red nodes on any root-leaf path
5. Same number of black nodes on any root-leaf path (called black height of the tree)
RB-Tree Properties

Some measures

- \( n \) – # of internal nodes
- \( h \) – height
- \( bh \) – black height

- \( 2^{bh} - 1 \leq n \)
- \( h/2 \leq bh \)
- \( 2^{h/2} \leq n + 1 \)
- \( h \leq 2 \log(n + 1) \)

\( \rightarrow \) balanced!
RB-Tree Properties/2

• **Operations** on a binary-search tree (search, insert, delete, ...) can be accomplished in $O(h)$ time

• The **RB-tree** is a binary search tree, whose **height** is bounded by $2 \log(n +1)$, thus the operations run in $O(\log n)$

  Provided that we can **maintain** the red-black tree properties spending no more than $O(h)$ time on each insertion or deletion
DSA, Chapter 6: Overview

- Binary Search Trees
  - Tree traversals
  - Searching
  - Insertion
  - Deletion

- Red-Black Trees
  - Properties
  - Rotations
  - Insertion
  - Deletion
Rotation

right rotation of B

left rotation of A
Right Rotation

RightRotate(Node B)
A := B.left

B.left := A.right
B.left.parent := B

if (B = B.parent.left) then B.parent.left := A
if (B = B.parent.right) then B.parent.right := A
A.parent := B.parent

A.right := B
B.parent := A
The Effect of a Rotation

- **Maintains** inorder key ordering
  
  For all $a \in \alpha$, $b \in \beta$, $c \in \gamma$

  rotation maintains the invariant (for the keys)

  $a \leq A \leq b \leq B \leq c$

- **After right rotation**
  - $\text{depth}(\alpha)$ decreases by 1
  - $\text{depth}(\beta)$ stays the same
  - $\text{depth}(\gamma)$ increases by 1

- **Left rotation**: symmetric

- Rotation takes $O(1)$ time
DSA, Chapter 6: Overview

– Binary Search Trees
  • Tree traversals
  • Searching
  • Insertion
  • Deletion

– Red-Black Trees
  • Properties
  • Rotations
  • Insertion
  • Deletion
**Insertion in the RB-Trees**

\[ \text{rBInsert(RBTree } t, \text{ RBNode } n) \]

*Insert } n \text{ into } t \text{ using the binary search tree insertion procedure*  

\[ n.\text{left} := \text{NULL} \]
\[ n.\text{right} := \text{NULL} \]
\[ n.\text{color} := \text{red} \]

\[ \text{rBInsertFixup}(n) \]
Fixing Up a Node: Intuition

Case 0: parent is black
    ➔ ok

Case 1: both parent and uncle are red
    ➔ change colour of parent/uncle to black
    ➔ change colour of grandparent to red
    ➔ fix up the grandparent

Exception: grandparent is root ➔ then keep it black

Case 2: parent is red and uncle is black, and
    node and parent are in a straight line
    ➔ rotate at grandparent

Case 3: parent is red and uncle is black, and
    node and parent are not in a straight line
    ➔ rotate at parent (leads to Case 2)
Insertion

Let

\[ n = \text{the new node} \]
\[ p = n.\text{parent} \]
\[ g = p.\text{parent} \]

In the following assume

\[ p = g.\text{left} \]
Insertion: Case 0

Case 0: \texttt{p.color} = black

– No properties of the tree are violated
– We are done
Insertion: Case 1

Case 1: n’s uncle u is red

- Action
  p.color := black
  u.color := black
  g.color := red
  n := g

- Note: the tree rooted at g is balanced enough (black depth of all descendants remains unchanged)
**Insertion: Case 2**

**Case 2:** *n’s uncle u is black* and *n* is a left child

- **Action**
  - $p.\text{color} := \text{black}$
  - $g.\text{color} := \text{red}$
  - RightRotate($g$)

- **Note:** the tree rooted at $g$ is balanced enough (black depth of all descendents remains unchanged).
Insertion: Case 3

Case 3: \( n \)'s uncle \( u \) is black and \( n \) is a right child

- Action
  
  \[
  \text{LeftRotate}(p) \\
  n := p
  \]

- Note: The result is a Case 2
Insertion: Mirror Cases

• All three cases are handled analogously if $p$ is a right child

• Exchange *left* and *right* in all three cases
Insertion: Case 2 and 3 Mirrored

**Case 2m:** n’s uncle u is black and n is a right child

- Action

  ```
  p.color := black
  g.color := red
  LeftRotate(g)
  ```

**Case 3m:** n’s uncle u is black and n is a left child

- Action

  ```
  RightRotate(p)
  n := p
  ```
Insertion Summary

• If two red nodes are adjacent, we perform either
  – a restructuring (with one or two rotations) and stop (cases 2 and 3), or
  – recursively propagate red upward (case 1)
• A restructuring takes constant time and is performed at most once; it reorganizes an off-balanced section of the tree
• Propagations may continue up the tree and are executed $O(\log n)$ times (height of the tree)
• The running time of an insertion is $O(\log n)$
An Insertion Example

Insert "REDSOX" into an empty tree

Now, let us insert "CUBS"
Insert C (Case 0)
Insert U (Case 3, Mirror)
Insert U/2
Insert B (Case 2)
Insert B/2

Before:

```
  E
 /   \
D     R
 /     /
C      O
 /     /
B      U
```

After:

```
  E
 /   \
C     R
 /     /
B      D
 /     /
S      O
 /     /
X      U
```

The tree is balanced after the insertion of B/2.
Insert S (Case 1)
Insert S/2 (Case 2 Mirror)
DSA, Chapter 6: Overview

- **Binary Search Trees**
  - Tree traversals
  - Searching
  - Insertion
  - Deletion

- **Red-Black Trees**
  - Properties
  - Rotations
  - Insertion
  - Deletion
Deletion

We first apply binary search tree deletion
• We can easily delete a node with at least one NULL child
• If the key to be deleted is stored at a node u with two children, we replace its content with the content of the largest node v of the left subtree (the predecessor of u) and delete v instead

![Binary Search Tree Graph]

Deletion Process:
1. Identify the node u with two children.
2. Find the largest node v (predecessor of u) in the left subtree of u.
3. Replace the content of u with the content of v.
4. Delete node v instead of u.
Deletion Algorithm

1. Remove \( u \)

2. If \( u.\text{color} = \text{red} \) we are done;
   else, assume that \( v \) (the predecessor of \( u \))
   gets an \textit{additional black color}:
   
   - if \( v.\text{color} = \text{red} \) then \( v.\text{color} = \text{black} \)
     and we are done!
   
   - else \( v \)'s color is “double black”
Deletion Algorithm/2

How to eliminate double black edges?

- The intuitive idea is to perform a color compensation
  Find a red node nearby, and change the pair (red, double black) into (black, black)

- Two cases: restructuring and recoloring
  - Restructuring resolves the problem locally, while recoloring may propagate it upward.

Hereafter we assume v is a left child (swap right and left otherwise)
Deletion Case 1

Case 1: v’s sibling s is black and both children of s are black

- Action: recoloring

  s.color := red
  v.color := black
  p.color := p.color + black

- Note: We reduce the black depth of both subtrees of p by 1; parent p becomes more black
Deletion: Case 1

If parent $p$ becomes **double black**, continue upward
Deletion: Case 2

**Case 2:** \( v \)'s sibling \( s \) is **black** and \( s \)'s right child is **red**

- **Action**
  
  \[
  \begin{align*}
  s.\text{color} &= p.\text{color} \\
  p.\text{color} &= \text{black} \\
  s.\text{right}.\text{color} &= \text{black} \\
  \text{LeftRotate}(p)
  \end{align*}
  \]

- **Idea:** Compensate the extra black ring of \( v \) by the red of \( r \)

- **Note:** Terminates after restructuring
Deletion: Case 3

Case 3: v’s sibling s is black, s’s left child is red, and s’s right child is black

- Idea: Reduce to Case 2
- Action
  
  s.left.color = black  
  s.color = red  
  RightRotation(s)  
  s = p.right  

- Note: This is now Case 2
Deletion: Case 4

Case 4: \( v \)'s sibling \( s \) is red

- Idea: give \( v \) a black sibling
- Action

\[
\begin{align*}
  s.\text{color} &= \text{black} \\
  p.\text{color} &= \text{red} \\
  \text{LeftRotation}(p) \\
  s &= p.\text{right}
\end{align*}
\]

- Note: This is now a Case 1, 2, or 3
Delete 9
Delete 9/2

• Case 2 (sibling is black with black children) – recoloring
Delete 8
Delete 7: Restructuring
How Long Does it Take?

Deletion in a RB-tree takes $O(\log n)$

Maximum:

- three rotations and
- $O(\log n)$ recolorings
Suggested Exercises

• Add left-rotate and right-rotate to the implementation of your binary trees

• Implement a class of red-black search trees with the following methods:
  – (...), insert, delete,
Suggested Exercises/2

Using paper and pencil:

• Draw the RB-trees after each of the following operations, starting from an empty tree:
  1. Insert 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
  2. Delete 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1

• Try insertions and deletions at random
Other Balanced Trees

- Red-Black trees are related to 2-3-4 trees (non-binary)

- AVL-trees have simpler algorithms, but may perform a lot of rotations
Next Part

• Hashing