Lecture 8
Red/black trees

EECS-214

Note

- If you're like me, then red/black trees will **make you feel stupid**
- You'll think
  - "**I'd never have thought of that**", or
  - "I understand every **individual step** and yet I still **don't feel like I understand the whole thing**"

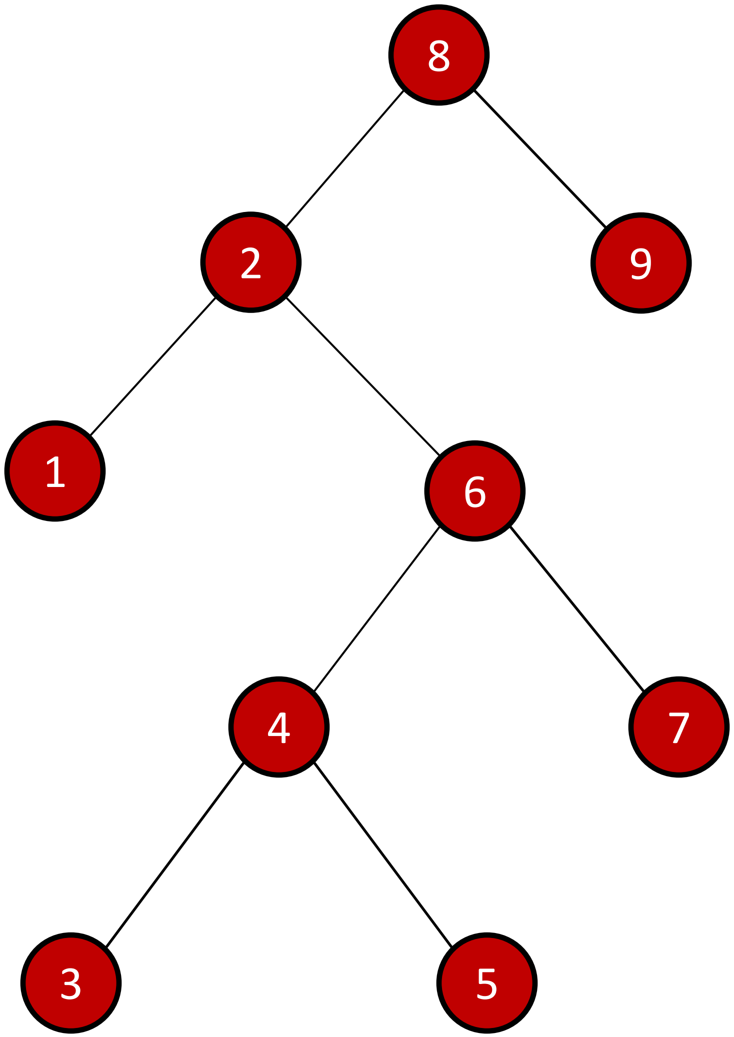
You're not stupid

- People **spent years** figuring this stuff out
  - **AVL trees** (first self-balancing trees)
  - Self-balancing trees with variable fan-out
    - 2-3 trees (1 or 2 keys per node)
      - Some balancing happens by grouping nodes into bigger nodes
    - **2-3-4** trees (up to 3 keys per node)
  - Red/black trees are a kind of **encoding of 2-3-4 trees in binary tree**
  - … which are themselves a form of **B-tree** (the standard data structure used for **databases** like Oracle)

- Each design **makes sense in terms of the previous design**
- Each design is **better** in some way
  - But also **more obscure**

- So it's **hard to get the intuitions** just by jumping into red/black trees
  - But unfortunately, it's **not worth 2 weeks of our time** to study all the precursors

Binary search trees



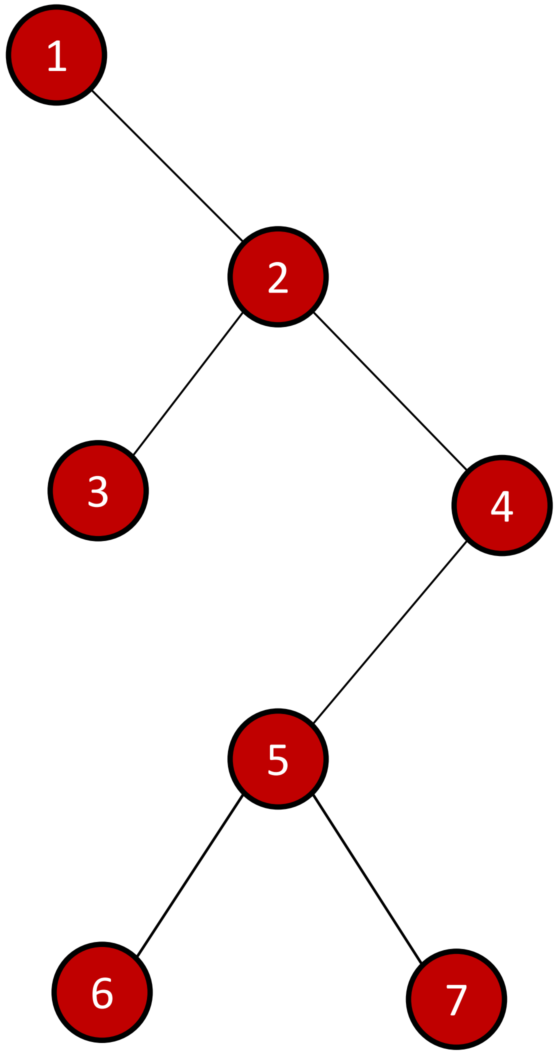
- Binary tree
- Each node **labeled** with a value
  - Number, string, or some other set that has a total order on it
- Has the magic **binary search tree property**
  - For any node
    - All the nodes in the left subtree have labels $\leq$ to its label
    - All the nodes in the right subtree have labels $\geq$ to its label
- Corollary: in-order traversal of tree visits nodes in **sorted order**

Inorder traversal

```
Inorder(node) {
  Inorder(node.leftChild)
  print node
  Inorder(node.rightChild)
}
```

Output:

1 3 2 6 5 7 4

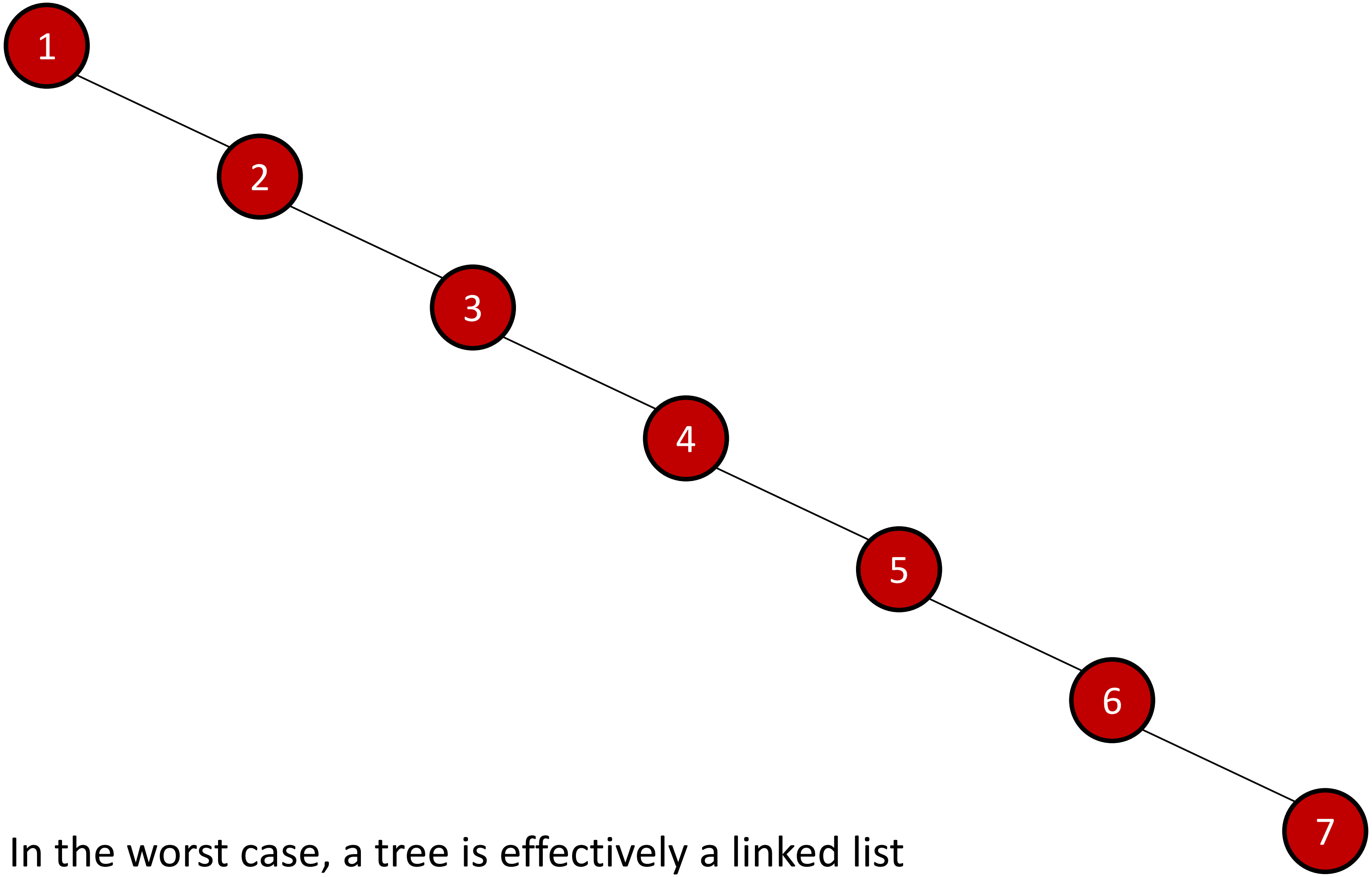


Searching a BST

- Start at root
- Move down
  - Move left or right depending on value of key
- When you find the key return the node
- Or return null if you run off the end of the tree

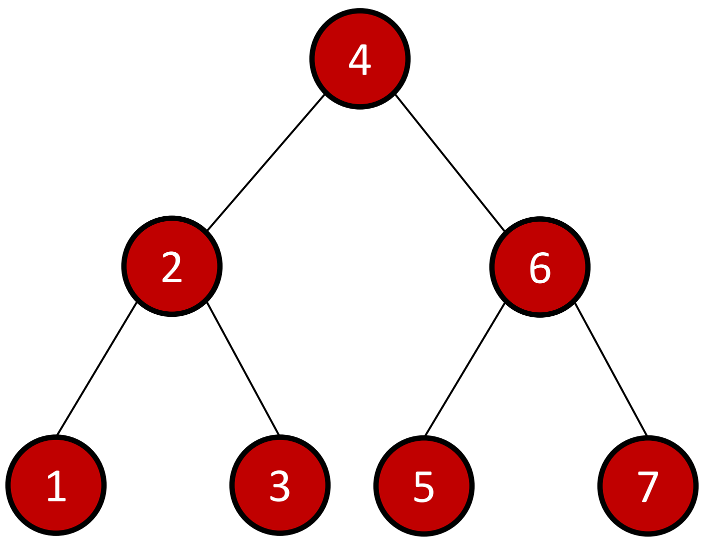
```
Search(node, int k) {
  while (node != null
         && node.key != k)
    if (k < node.key)
      node = node.left;
    else
      node = node.right;
  return node;
}
```

A bad tree to search



In the worst case, a tree is effectively a linked list

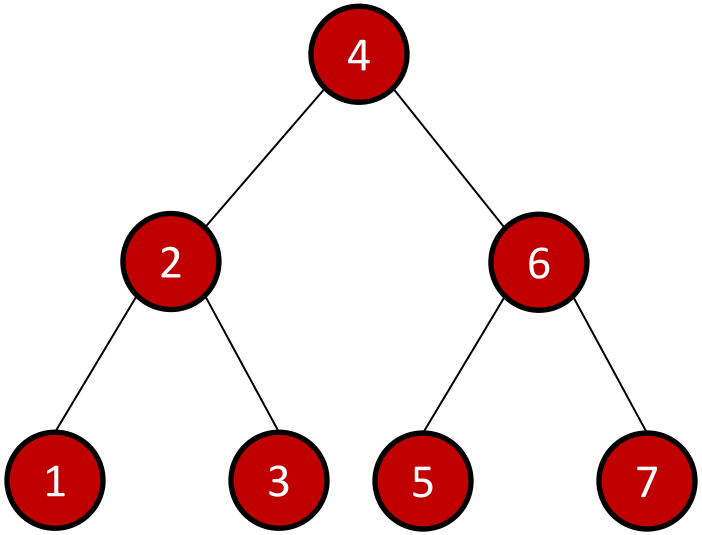
A good tree to search



- There are many different valid binary trees for any given set of keys
- We want to choose **balanced** tree structures
  - Trees whose left and right subtrees have **roughly the same** size and depth
- A balanced search tree has a small height for its given number of nodes
  - $O(\log n)$

Self-balancing trees

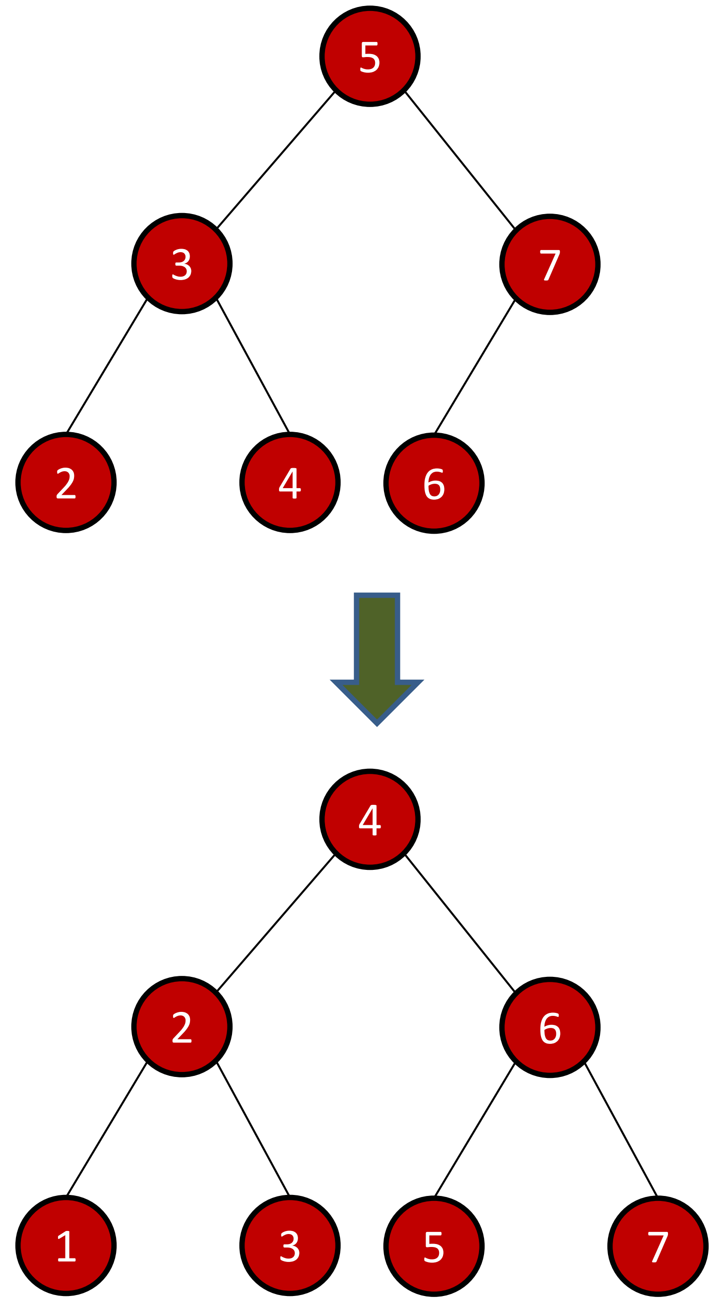
- **Adapt their shapes** as we add/remove elements
- Generally modified version of **normal binary search tree**
  - Same search algorithm
  - Post-processing added to insertion and deletion operations to rebalance tree



A bad algorithm

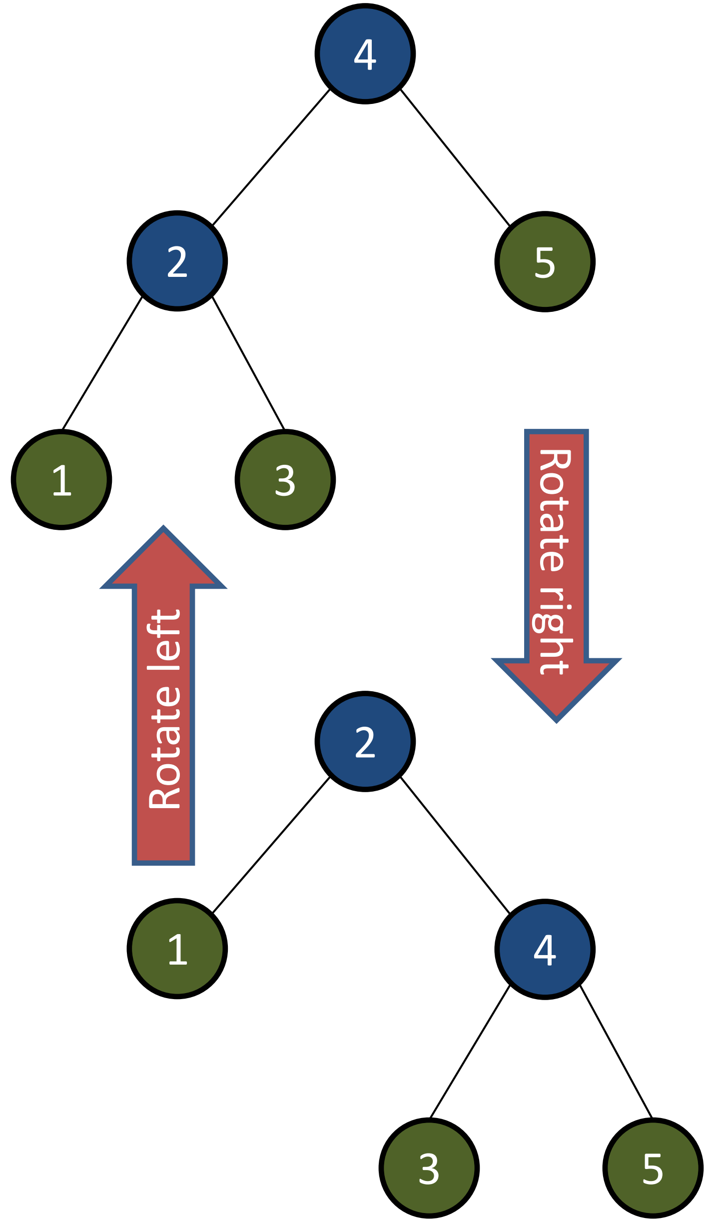
After each modification:

- Walk the tree to count all the nodes
- **Make a new, perfectly balanced tree** with just that number of nodes
  - That turns out to be easier than you might think, but it’s still expensive
- Copy the keys from the old tree to the new tree
- Throw away the old tree



Tree rotation

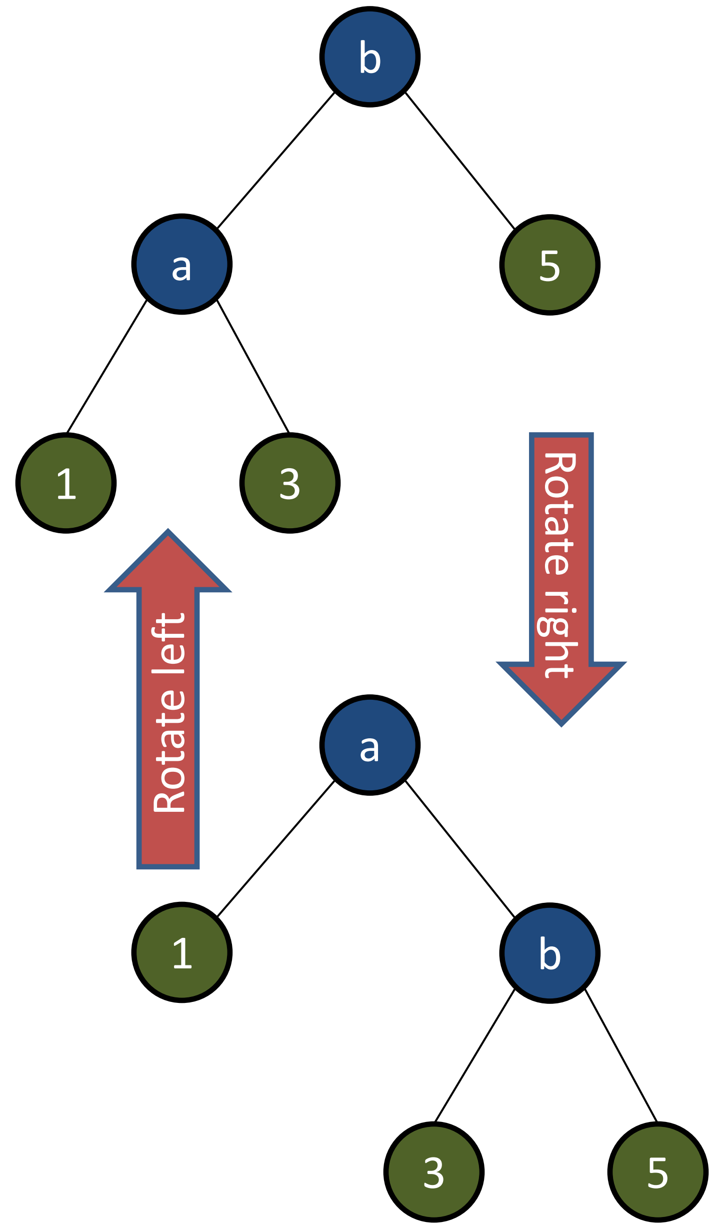
- A tree rotation is an operation that
  - **Changes** the **parent/child relationships** of a group of nodes
  - **Without changing** the **in-order traversal** of the nodes
- In other words, it **preserves the binary-search-tree property**
  - It maintains it as an **invariant**



Right rotation

```
RotateRight(b) {
  a = b.left;
  b.left = a.right;
  a.parent = b.parent
  b.parent = a;
  a.right = b;
}
```

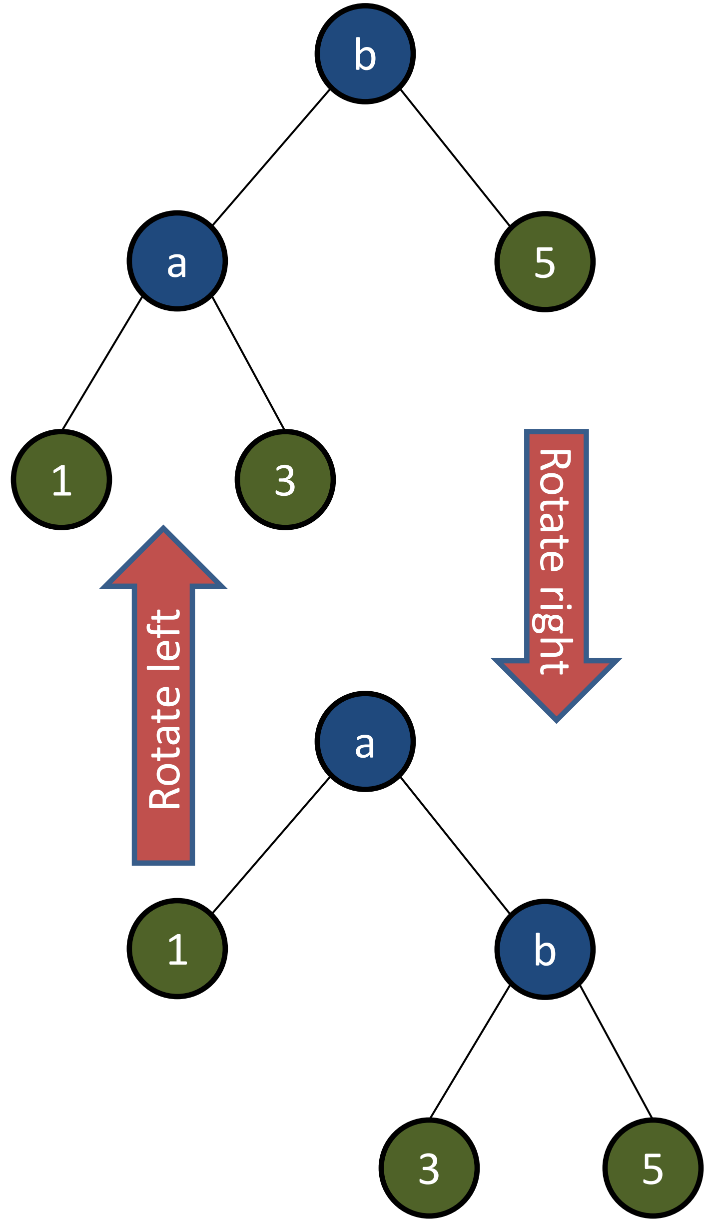
- b here is usually called the **pivot**
- **Demotes pivot, promotes left child**



Left rotation

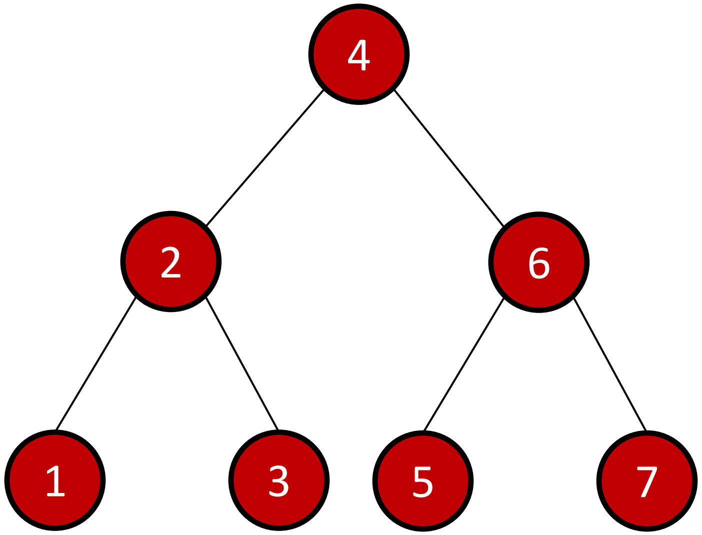
```
RotateLeft(a) {
  b = a.right;
  a.right = b.left;
  b.parent = a.parent
  a.parent = b;
  b.left = a;
}
```

- Node a is the pivot
- Demotes pivot, **promotes right child**



Key idea of self-balancing

- Insertion/deletion is a **local operation**
  - Only affects a small part of the tree
- If **tree starts balanced**
  - We can **rebalance** it using **rotations**
  - In the **modified area**
  - And its **ancestors**
  - Can safely ignore the rest of the tree



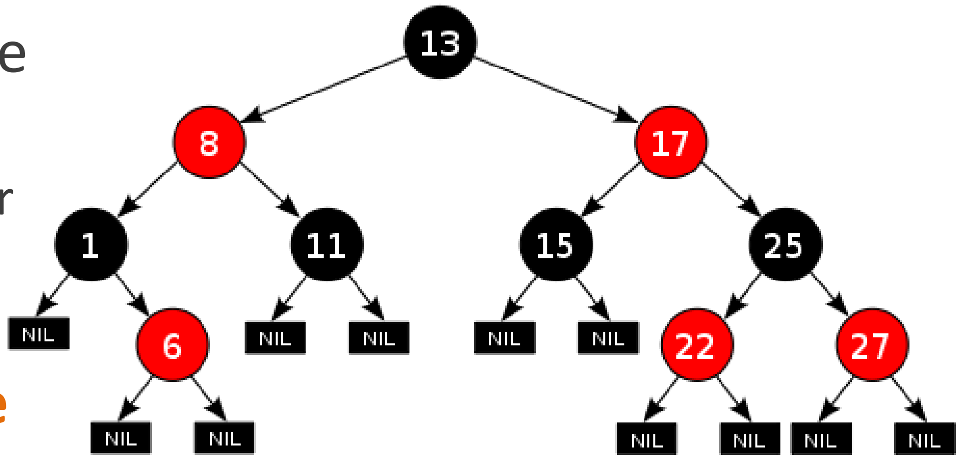
Major issues in designing self-balancing trees

- **How do you tell** if a tree is unbalanced?
  - Don’t want to have to **walk the whole tree** to see how deep it is
  - Need to cache some kind of information in the nodes to help out
- How balanced is **balanced enough**?
  - Takes less work to keep a tree “roughly” balanced than perfectly balanced

Red-black trees

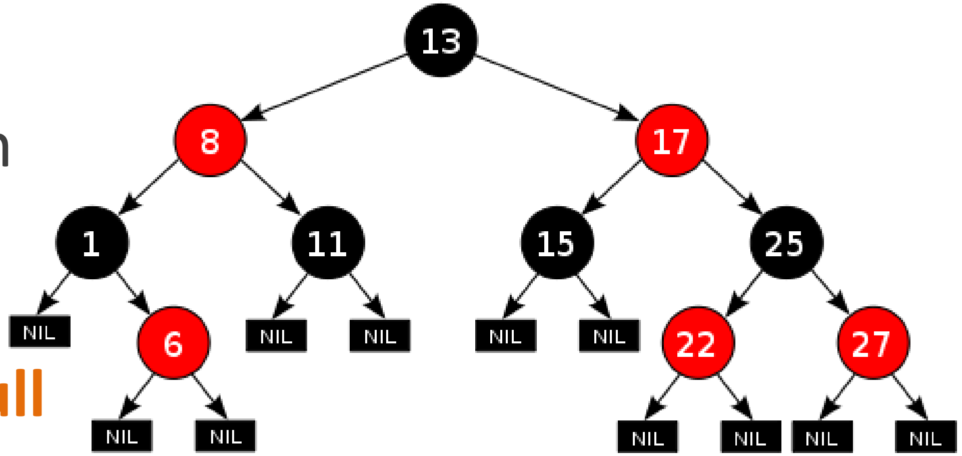
- Color every node of the tree "**red**" or "**black**"
  - Adds only 1 bit of storage per node
- Used as a way of **determining when the tree is getting unbalanced**
  - Assign node colors cleverly (see coming slides)

- Guarantees tree is **no more than twice** as high as the "optimal" tree



Red-black trees

- All **interior** (non-leaf) nodes have two children
- All leaves are marked **null** (or "nil") and are black
  - Can be implemented by treating null pointers as black leaves
- It's common **not to bother drawing** the null leaves in figures



Invariants on red-black trees

- All red-black trees are required to satisfy a set of **red-black tree properties**

- These properties are **invariant**
  - All operations (search, insertion, deletion)
  - Assume they're satisfied before the operation
  - Guarantee they'll be satisfied after the operation

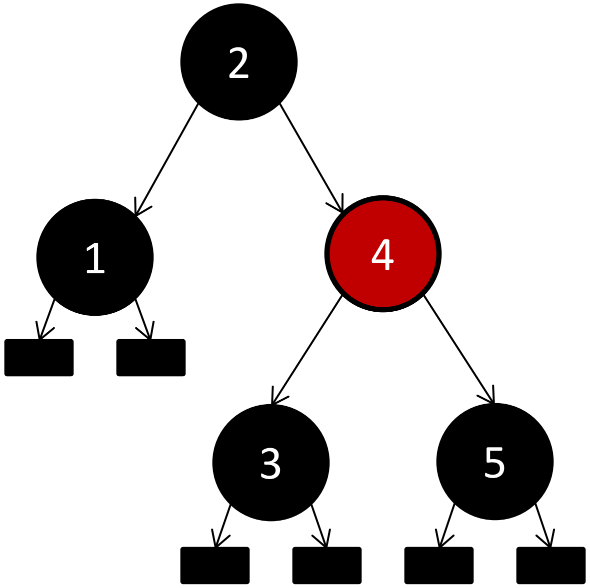
Red-black tree properties

1. All nodes are labeled either **red** or **black**
2. The root node is always **black**
3. All leaves are **black**
   - Remember that "leaves" in a red-black tree are null
4. Both children of any **red** node are **black**
5. Any simple path from a node to one of its descendants has the same number of **black** nodes as the paths to its other descendants

Why the invariants matter: Bounds on path length

- Forcing paths to have the same number of black nodes
  - Forces them to have approximately the same length

- Why?

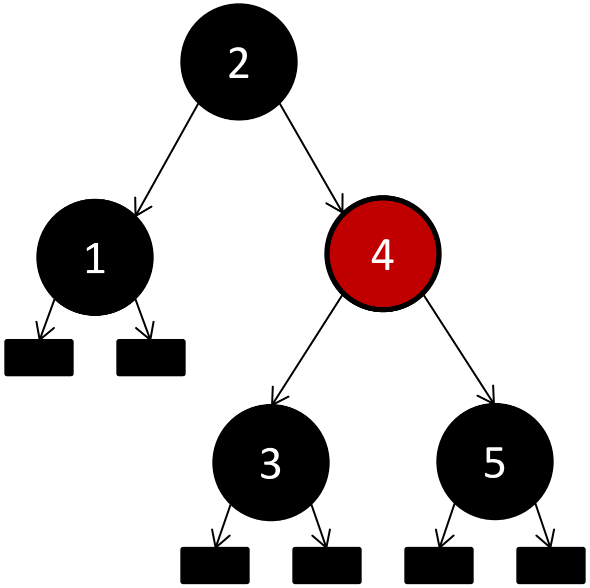


Why the invariants matter: Bounds on path length

Proof:

- Children of red nodes are required to be black
  - So a path can **never have two consecutive red nodes**
  - Therefore # of red nodes ≤ # of black nodes
  - Therefore **path length ≤ 2 times # of black nodes**

- But **all paths have the same # of black nodes**
  - So all paths within a factor of 2 in length

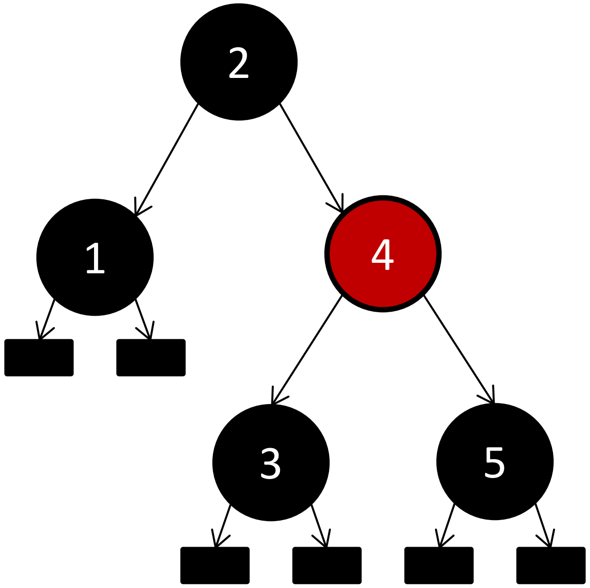


Why the invariants matter: Bounds on path length

Proof:

- So the **heights of two subtrees** can only differ by at most a factor of 2

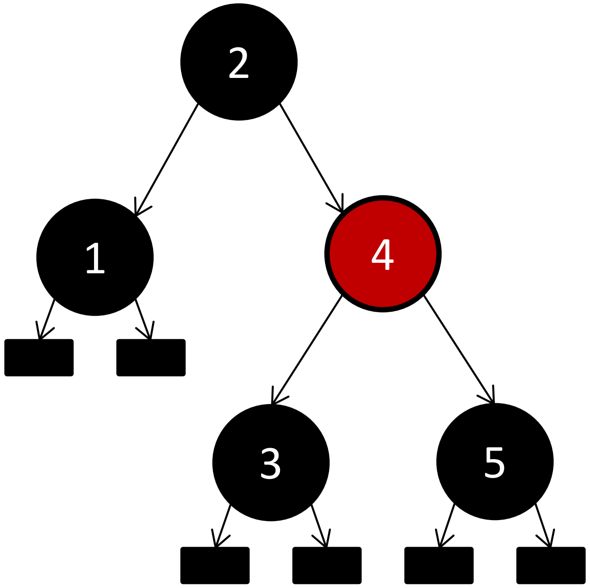
- So the tree is **approximately balanced**



Why the invariants matter: Bounds on path length

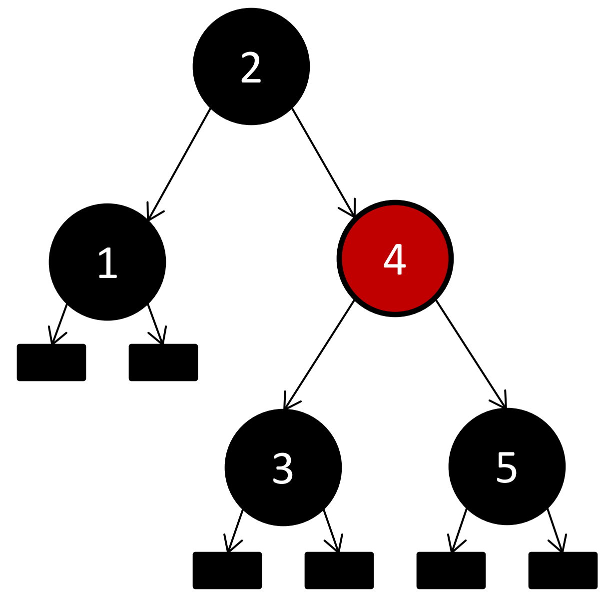
- Okay, but it's **not perfectly balanced**

- **How much difference** does this make?



Why the invariants matter: Bounds on **tree height**

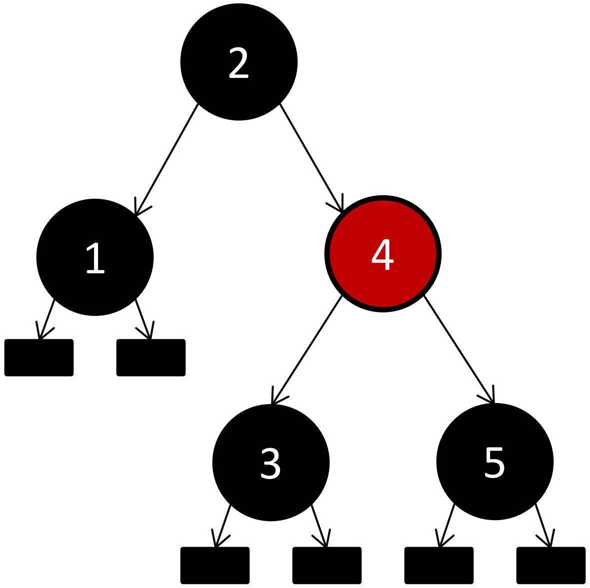
- We want to show that a red/black tree's height is **within a factor of 2** of the optimal case
- We'll start by proving that the **# of black nodes constrains the size and height** of the tree



Why the invariants matter: Bounds on **tree height**

Definition:

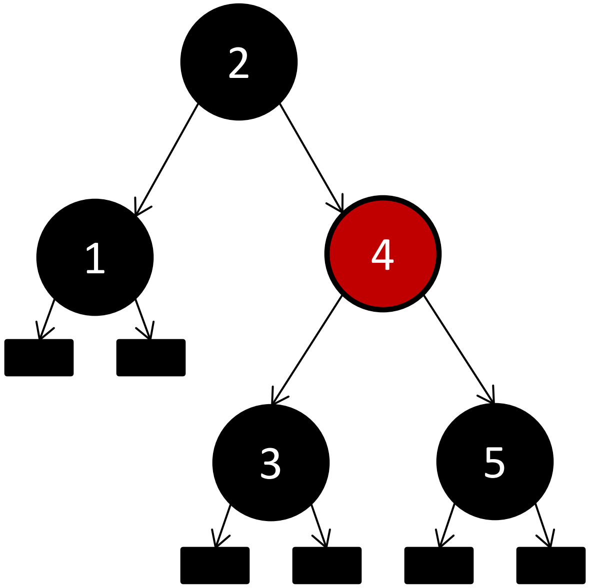
The **black-height**, $bh(v)$, of a node v is the number of black nodes in a path from v to one of its leaf descendants



Why the invariants matter: Bounds on **tree height**

Claim:

A subtree beneath a node v has at least $2^{bh(v)} - 1$ internal nodes



Proof:

by induction on height

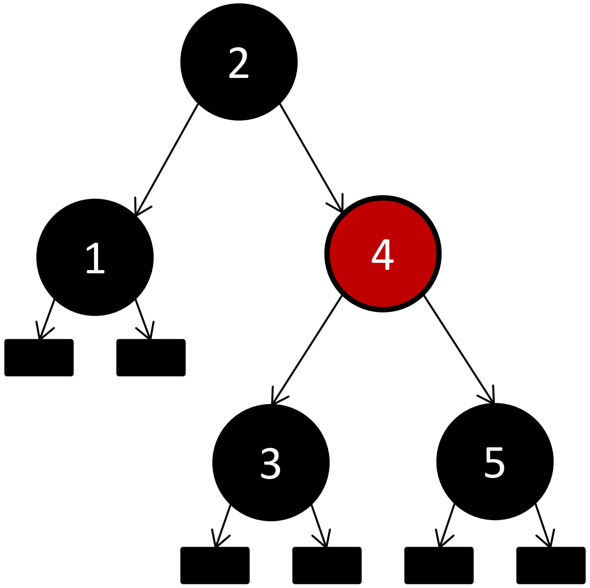
Why the invariants matter: Bounds on **tree height**

Claim:

A subtree beneath a node v has at least $2^{bh(v)} - 1$ internal nodes

Base case:

- **If height is 0**, then v is a leaf
- Subtree has no other nodes
- So subtree has $2^0 - 1 = 0$ internal nodes



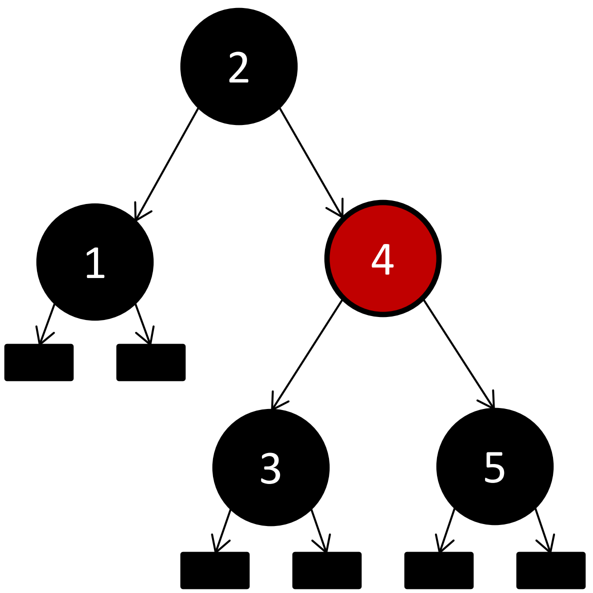
Why the invariants matter: Bounds on **tree height**

Claim:

A subtree beneath a node v has at least $2^{bh(v)} - 1$ internal nodes

Inductive case:

- **Assume** it's true for nodes with **black height** $\boldsymbol{k}$
- Consider a node v for which $\boldsymbol{bh(v) = k + 1}$



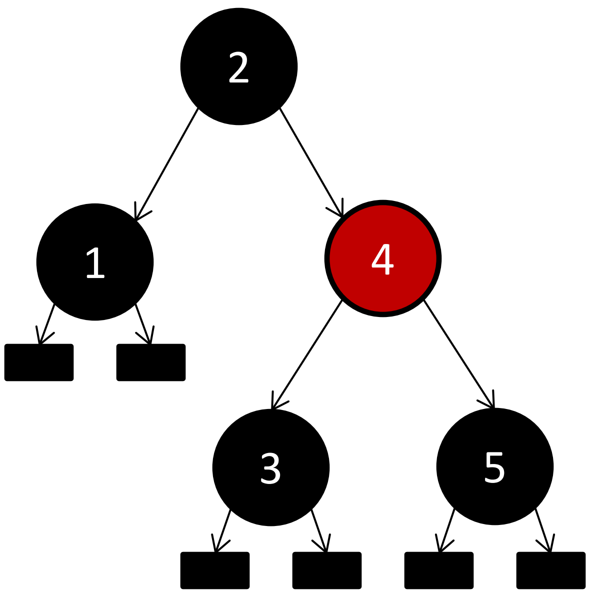
Why the invariants matter: Bounds on **tree height**

Claim:

A subtree beneath a node v has at least $2^{bh(v)} - 1$ internal nodes

Inductive case:

- Since $bh(v) > 0$,
  - It's an internal node
  - And so has **two children**
- The children have black heights of either $bh(v)$ or $bh(v) - 1$
  - So their **black heights are at least k**



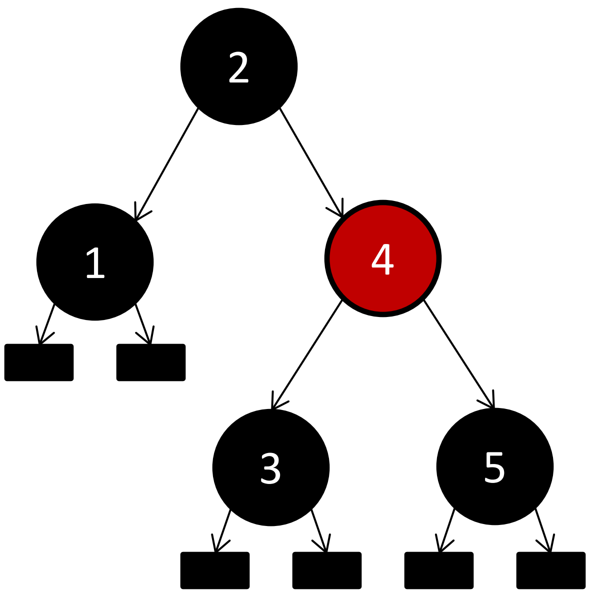
Why the invariants matter: Bounds on **tree height**

Claim:

A subtree beneath a node v has at least $2^{bh(v)} - 1$ internal nodes

Inductive case:

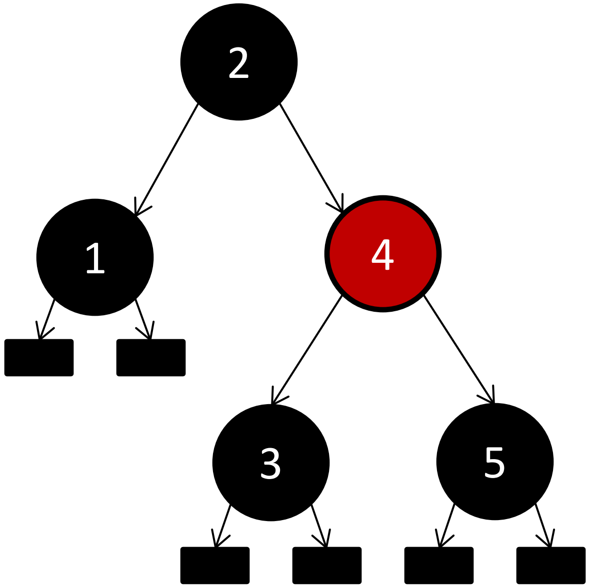
- By assumption, the left- and right-subtrees then have at least $\mathbf{2^k - 1}$ **internal nodes each**
- But v is also an internal node, so the whole thing has at least:
  $$\mathbf{2(2^k - 1) + 1 = 2^{k+1} - 1}$$
  internal nodes. ∎



Why the invariants matter: Bounds on **tree height**

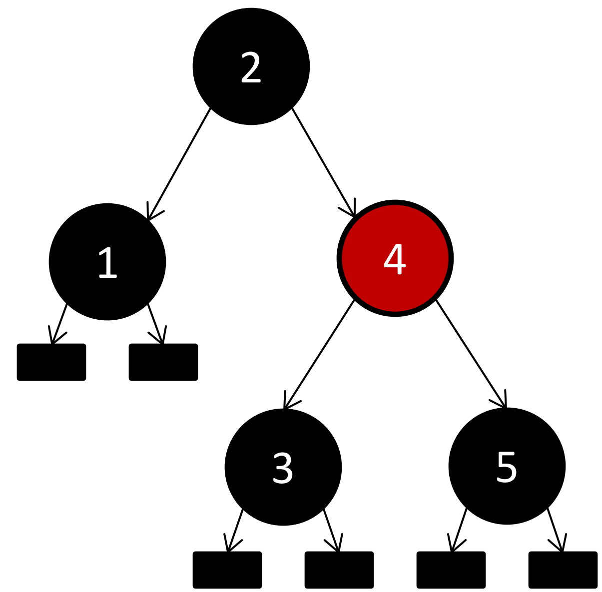
- Red-black trees only store keys in internal nodes – not leaves

- So we want to know how the **height** of the tree **grows** with the **number of internal nodes**, n.



Why the invariants matter: Bounds on **tree height**

- Let:
  - r be the root node of the tree
  - h be the height of the tree
    - Note that $h \leq 2bh(r)$ because at least half the nodes on any path have to be black

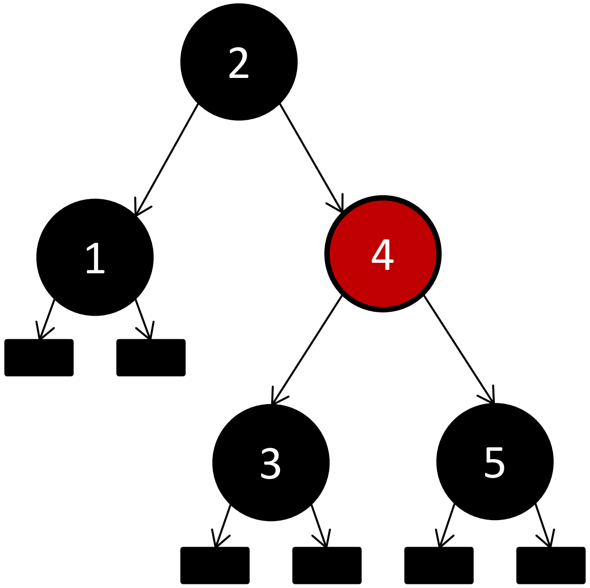


- We have that:
  - $n \geq 2^{bh(r)} - 1 \geq 2^{h/2} - 1$
  - $\log_2(n+1) \geq \frac{h}{2}$

- And thus:
  $\boldsymbol{h \leq 2\, log_2(n+1) = O(log\, n)}$

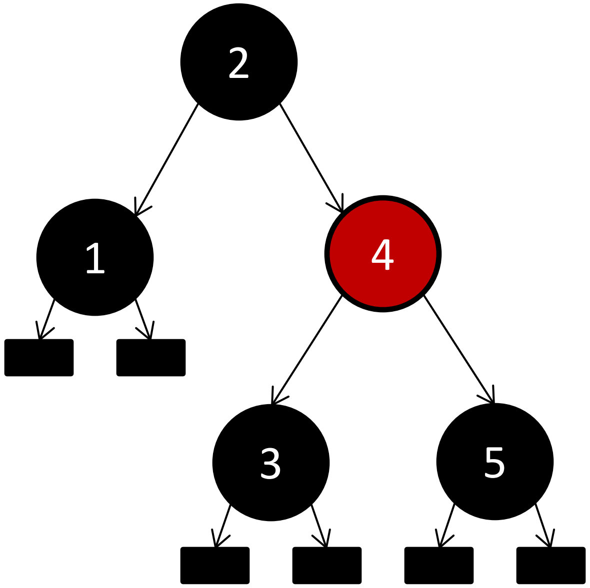
Enforcing the invariants

- So the invariants ensure that
  - The **height** of the tree will be $\boldsymbol{O(\log n)}$
  - And so **searching** will be $\boldsymbol{O(\log n)}$
- And so all we have to do is **ensure** that **insertion and deletion maintain the invariants**
- Oh joy



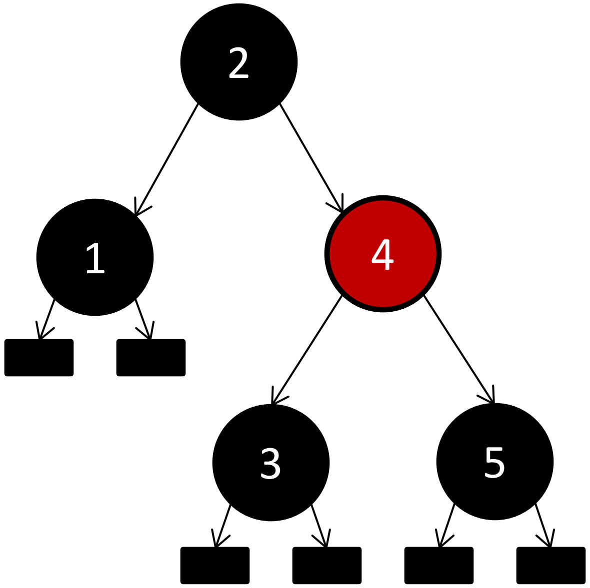
Enforcing the invariants

- Basic idea:
  - Perform the **normal insertion/deletion** algorithm for unbalanced trees
  - "**Fix up**" the tree to rebalance it



Enforcing the invariants: insertion

- Insert node as normal
- Always color node **red**
- Crawl up the tree fixing any violations of invariants
  - The root node is always **black**
  - Both children of any **red** node are **black**
  - Any simple path from a node to one of its descendants has the same number of **black** nodes



Outline of the algorithm

- Insert new node and color it red
- While node not root
  and both node and parent are **red**
  - If uncle (parent’s sibling) is also **red**
    - Flip colors of parent, uncle, and grandparent
    - Node = grandparent (i.e. check grandparent in next iteration)
  - Else (uncle is black)
    - If node is a left child
      - Node = parent
      - Rotate node left
    - Flip colors of parent and grandparent
    - Rotate grandparent right

Outline of the algorithm

- Insert new node and color it red
- While node not root
  and both node and parent are **red**
  - If uncle (parent's sibling) is also **red**
    - Flip colors of parent, uncle, and grandparent
    - Node = grandparent (i.e. check grandparent in next iteration)
  - Else (sibling is **black**)
    - If node is a right child
      - Node = parent
      - Rotate node left
    - Flip colors of parent and grandparent
    - Rotate grandparent right

- We assume all the invariants are satisfied before the insertion

- Want to show that they're satisfied afterward

Outline of the algorithm

- Insert new node and color it red
- While node not root
  and both node and
  parent are **red**
  - If uncle (parent’s sibling) is also **red**
    - Flip colors of parent, uncle, and grandparent
    - Node = grandparent (i.e. check grandparent in next iteration)
  - Else (sibling is black)
    - If node is a right child
      - Node = parent
      - Rotate node left
    - Flip colors of parent and grandparent
    - Rotate grandparent right

Properties in play

- Both children of any **red** node are **black**
  - **Could be violated**
- Any simple path from a node to one of its descendants has the same number of **black** nodes
  - **Unaffected**
  - **Adding a red node doesn’t change any black heights**

Outline of the algorithm

- Insert new node and color it red
- While node not root
  and both node and
  parent are **red**
  - If uncle (parent's sibling) is also **red**
    - Flip colors of parent, uncle, and grandparent
    - Node = grandparent (i.e. check grandparent in next iteration)
  - Else (sibling is black)
    - If node is a right child
      - Node = parent
      - Rotate node left
    - Flip colors of parent and grandparent
    - Rotate grandparent right

Properties in play

- Both children of any **red** node are **black**
  - **Could be violated**
- Any simple path from a node to one of its descendants has the same number of **black** nodes
  - **Unaffected**
  - **Flips grandparent**
  - **But also both of grandparent's children**
  - **So black height consistency is preserved**

Outline of the algorithm

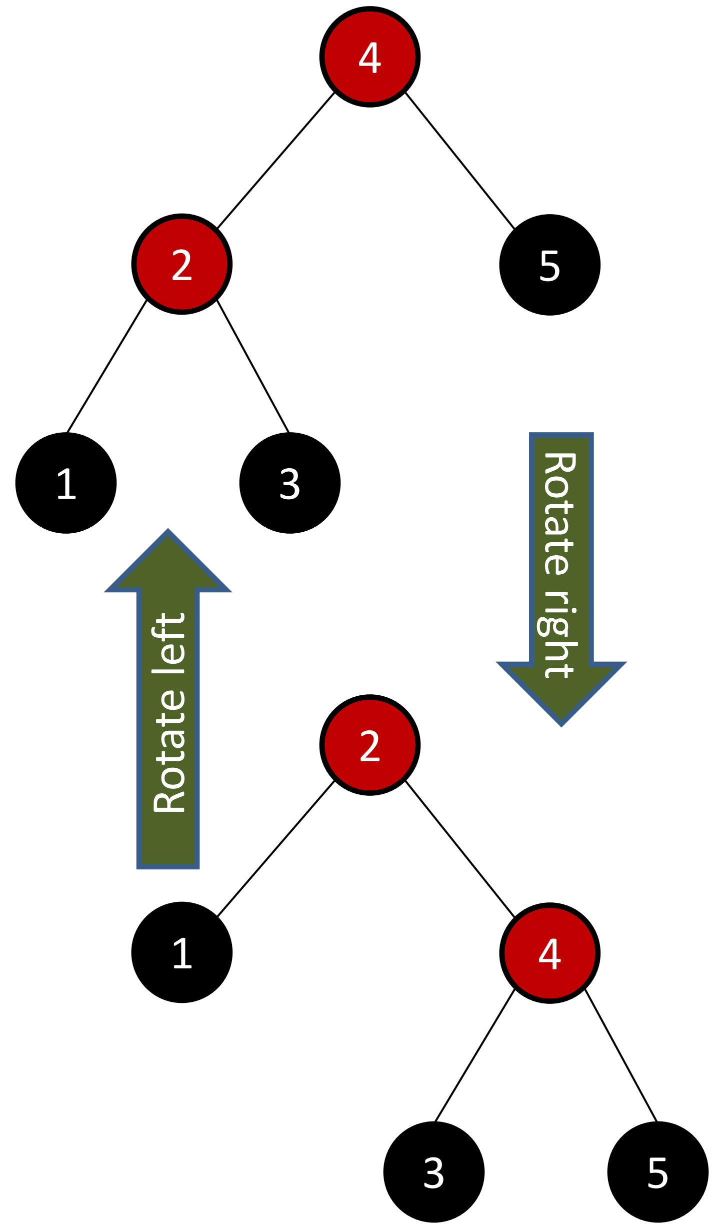
- Insert new node and color it red
- While node not root
  and both node and
  parent are **red**
  - If uncle (parent's sibling) is also **red**
    - Flip colors of parent, uncle, and grandparent
    - Node = grandparent (i.e. check grandparent in next iteration)
  - Else (sibling is black)
    - If node is a right child
      - Node = parent
      - Rotate node left
    - Flip colors of parent and grandparent
    - Rotate grandparent right

Properties in play

- Both children of any **red** node are **black**
  - **Left the same**
- Any simple path from a node to one of its descendants has the same number of **black** nodes
  - **Unaffected**
  - **Rotating two red nodes leaves black heights unaffected**

Tree rotations

- Rotating two red nodes leave black heights unchanged



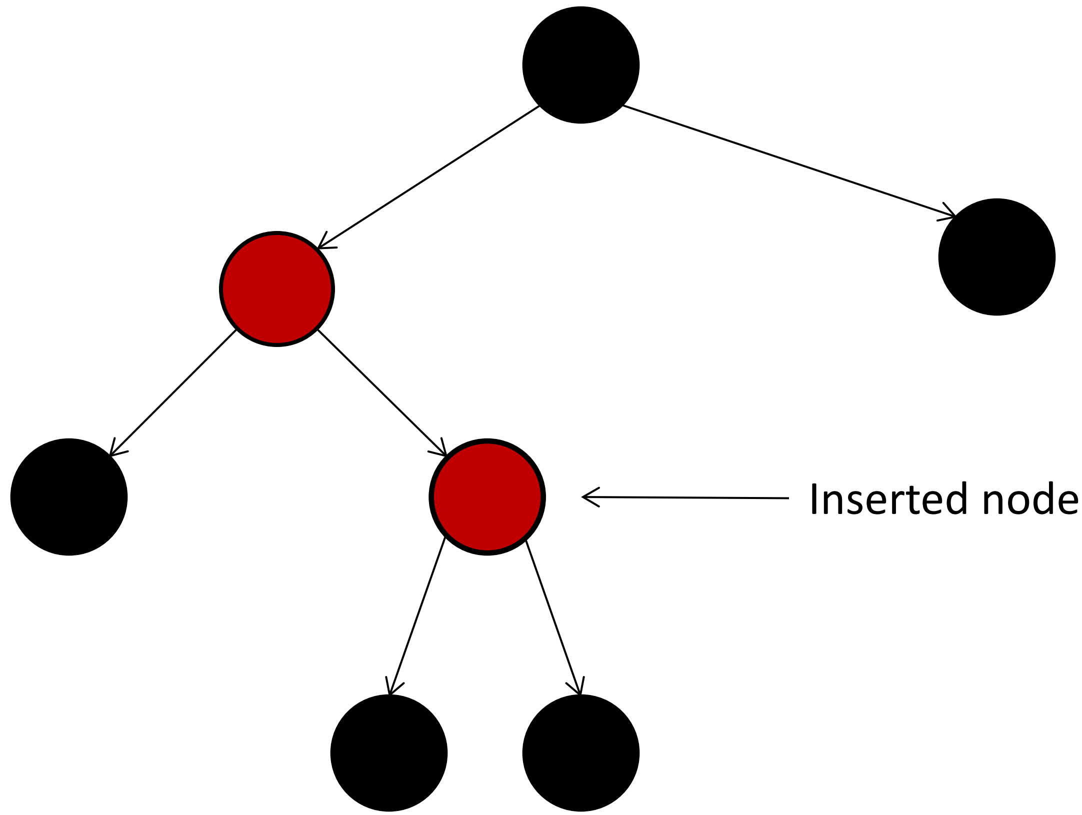
Outline of the algorithm

- Insert new node and color it red
- While node not root
  and both node and
  parent are **red**
  - If uncle (parent's sibling) is also **red**
    - Flip colors of parent, uncle, and grandparent
    - Node = grandparent (i.e. check grandparent in next iteration)
  - Else (sibling is black)
    - If node is a right child
      - Node = parent
      - Rotate node left
    - Flip colors of parent and grandparent
    - Rotate grandparent right

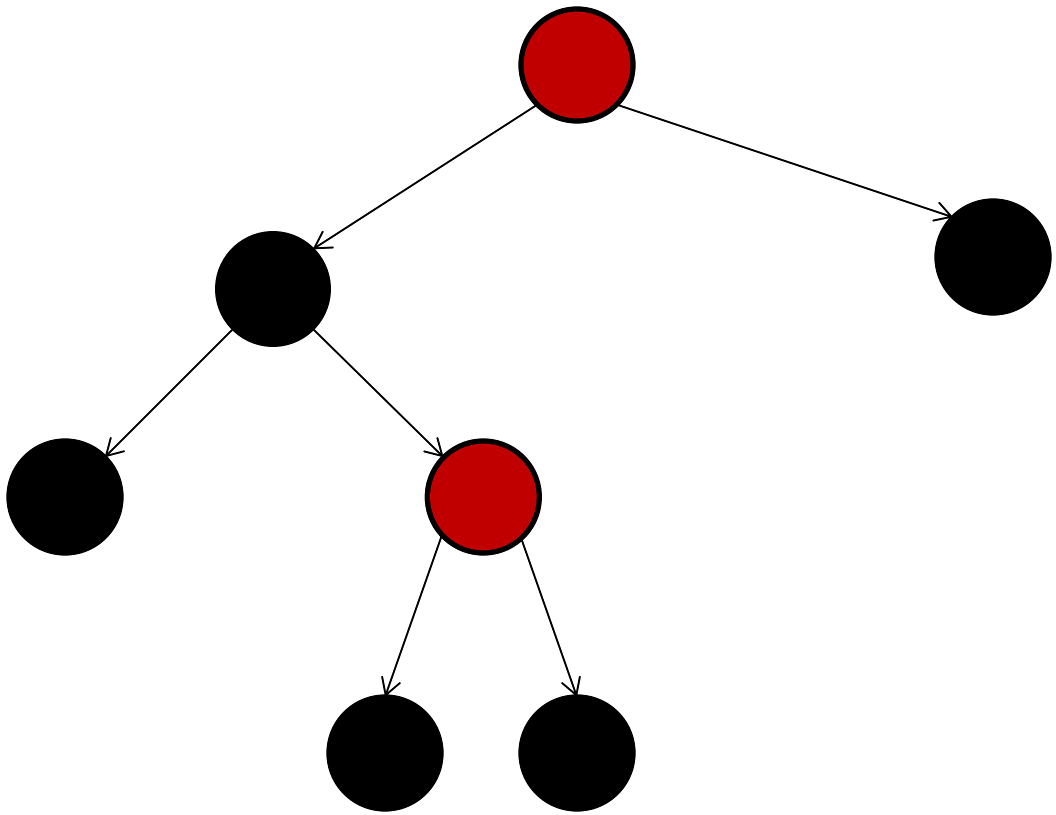
Properties in play

- Both children of any **red** node are **black**
  - **Left the same**
  - **But easier to see in the example coming up**
- Any simple path from a node to one of its descendants has the same number of **black** nodes
  - **Unaffected**
  - **Adding red node, removing red node, and rotating**

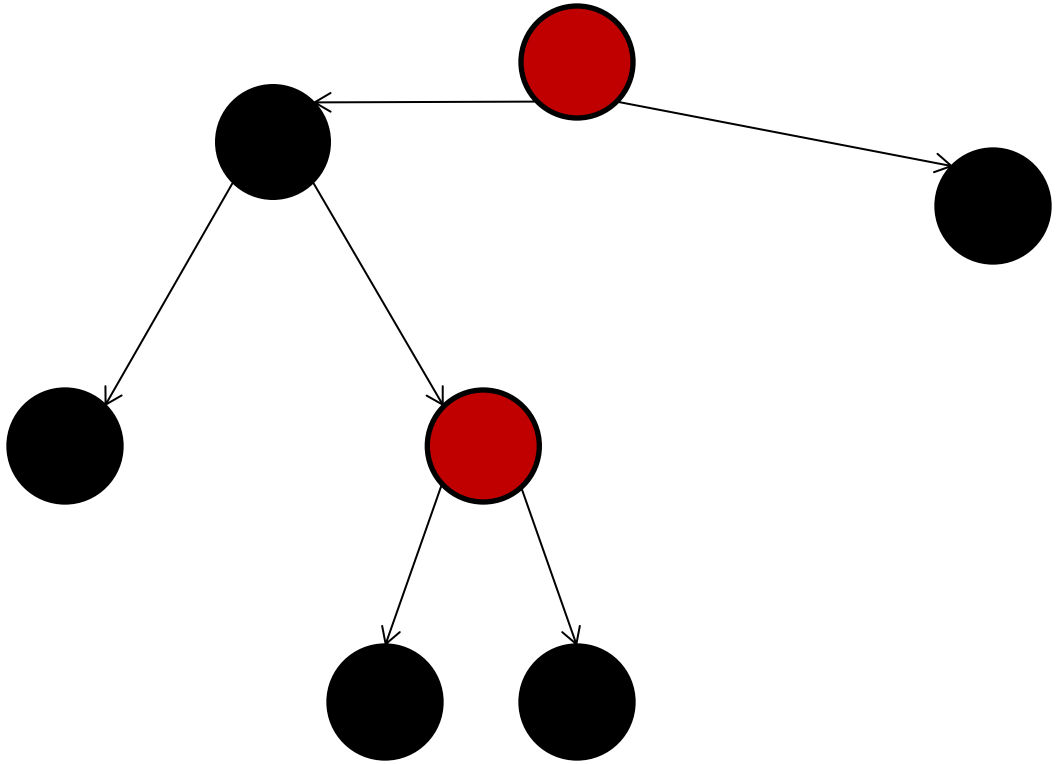
Starting configuration



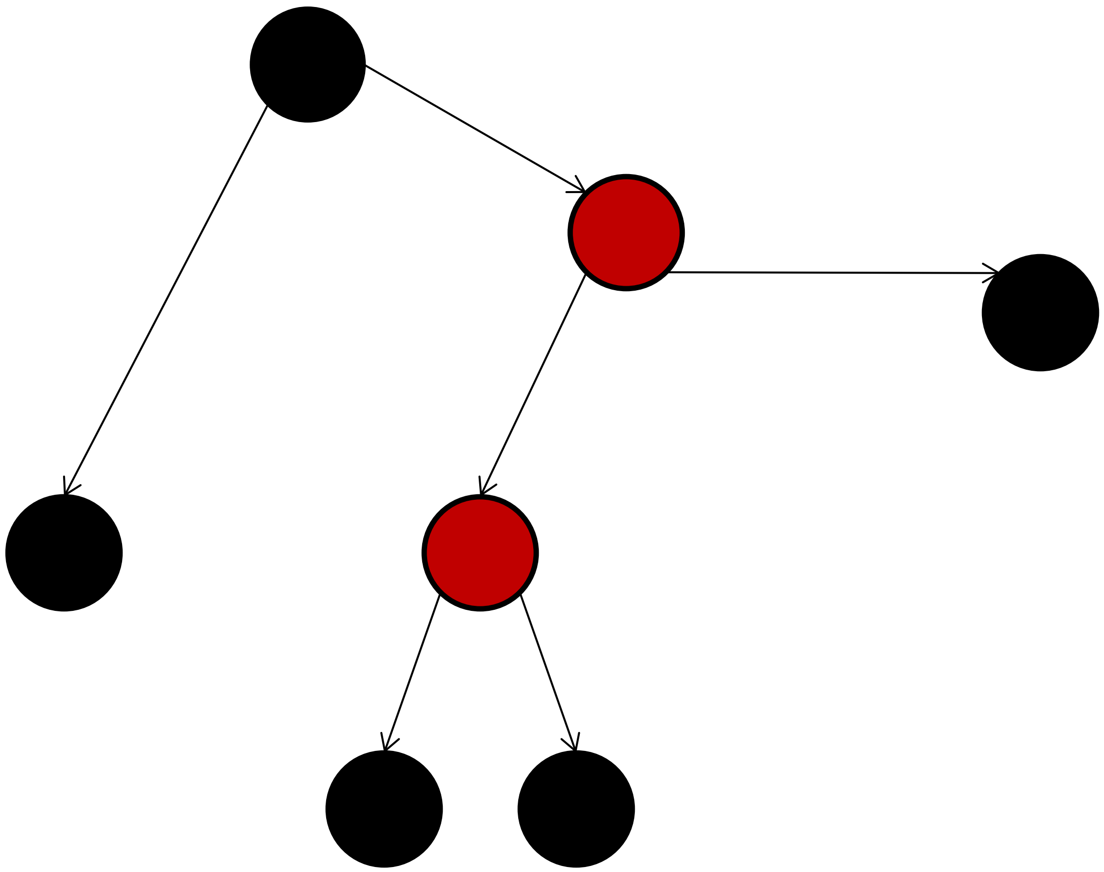
Flip colors
of parent & grandparent



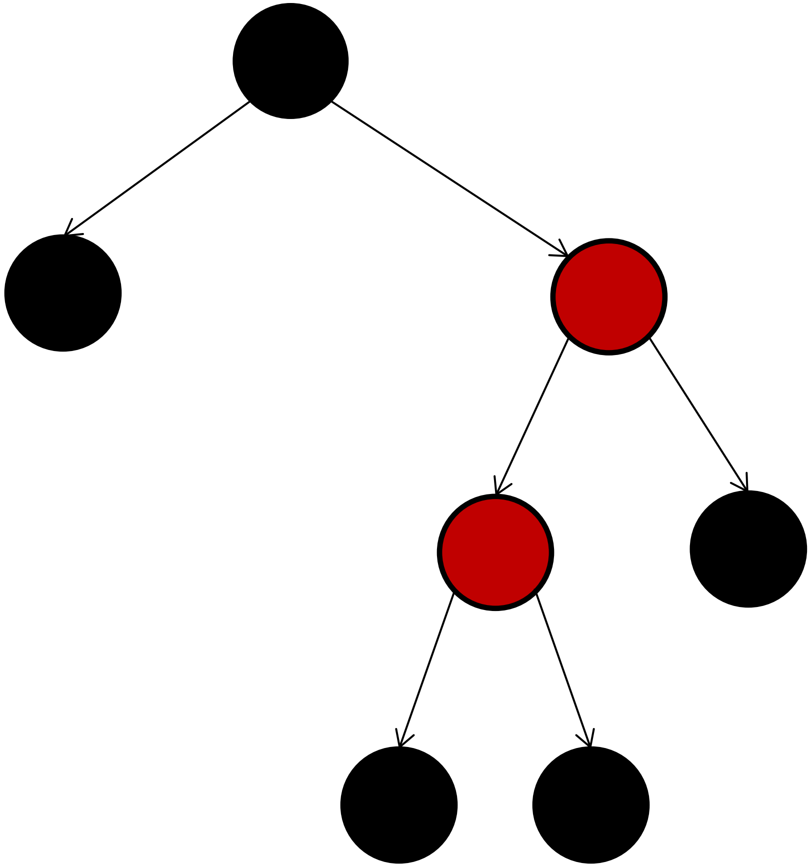
Flip colors
of parent & grandparent



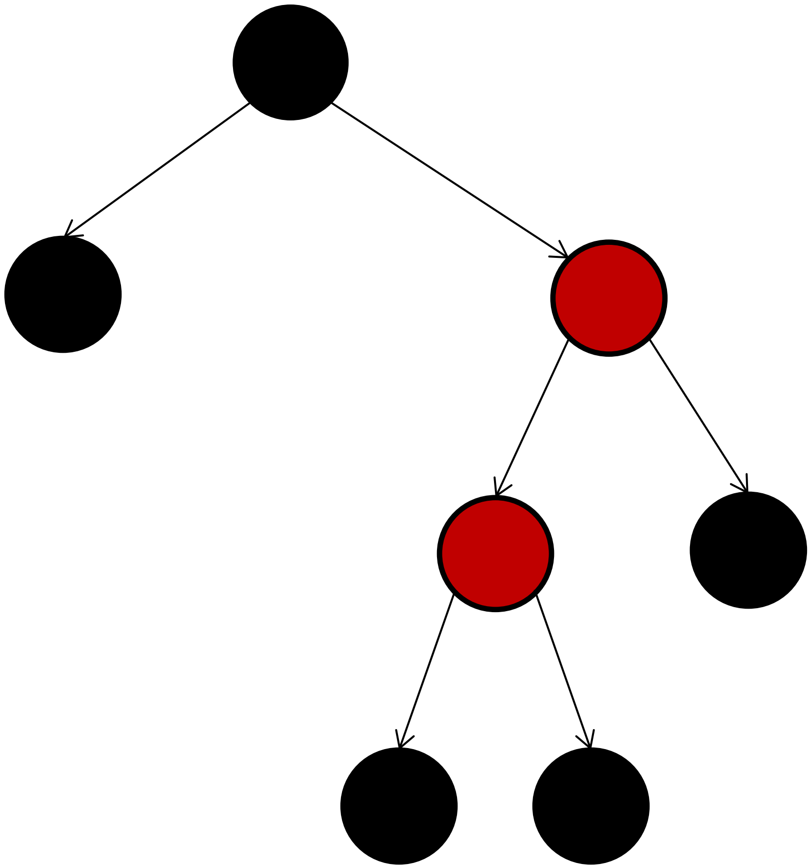
Rotate right



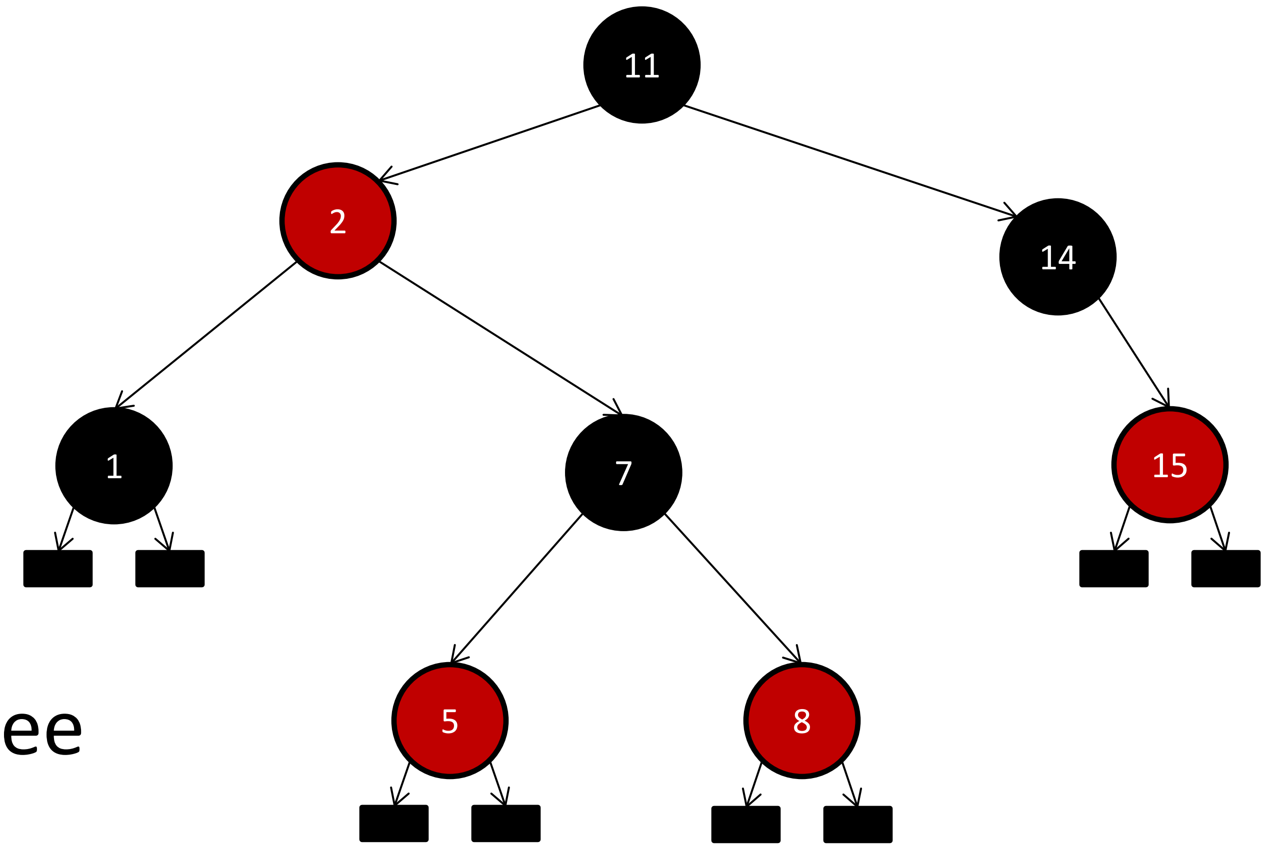
(fix layout ...)



(fix layout ...)

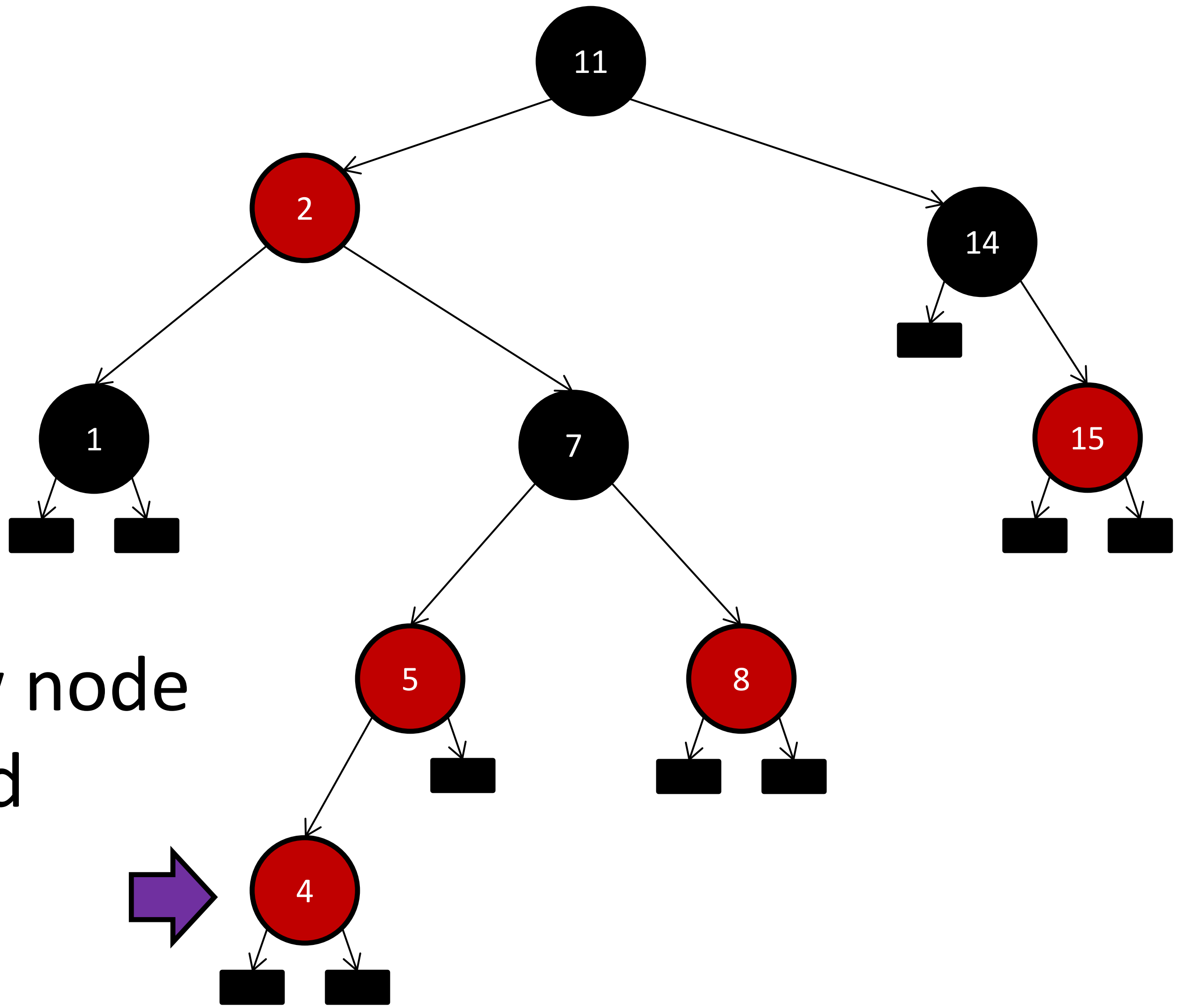


Insertion example (from CLR)



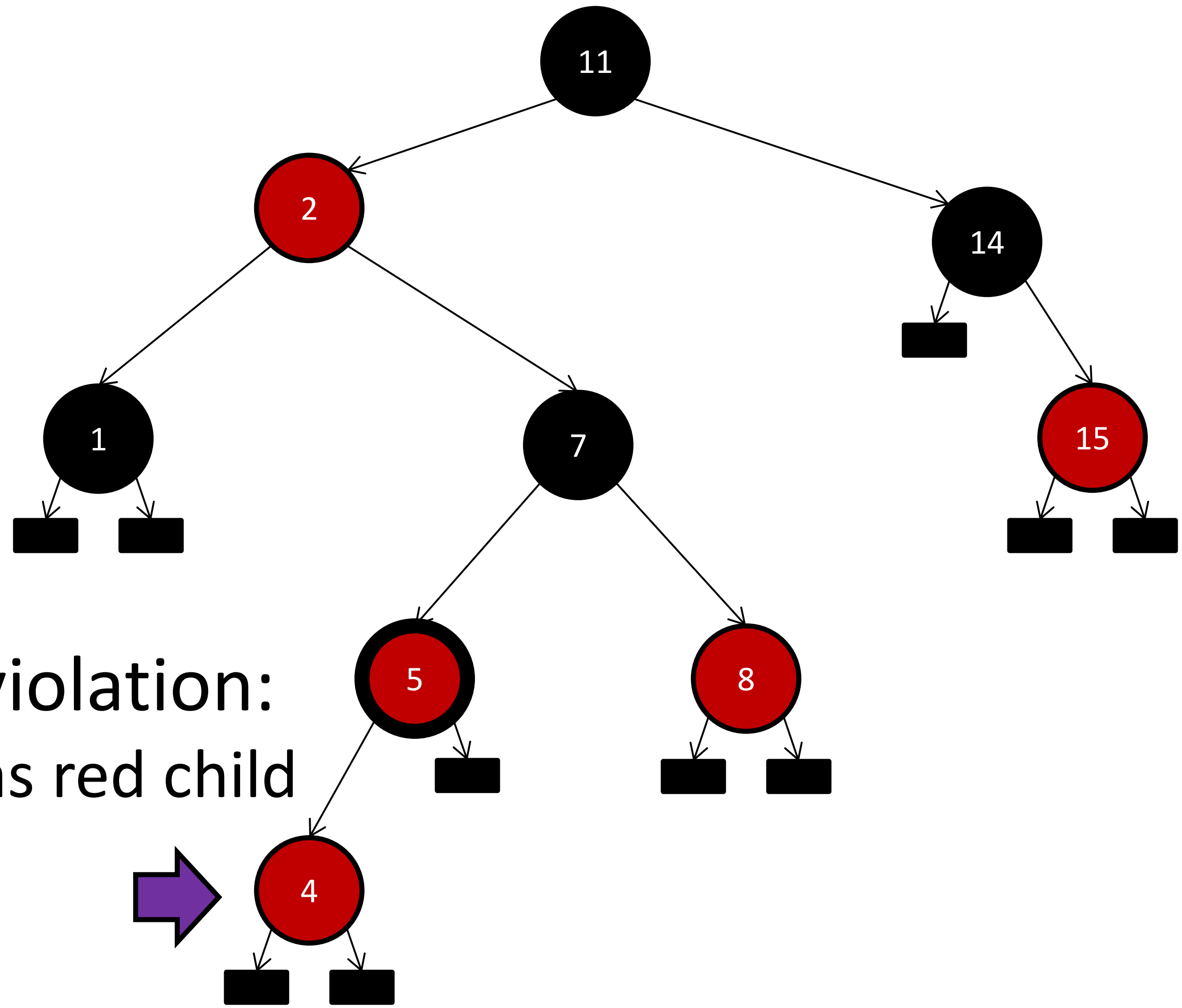
Starting tree

Insertion example (from CLR)



Insert new node
Color it red

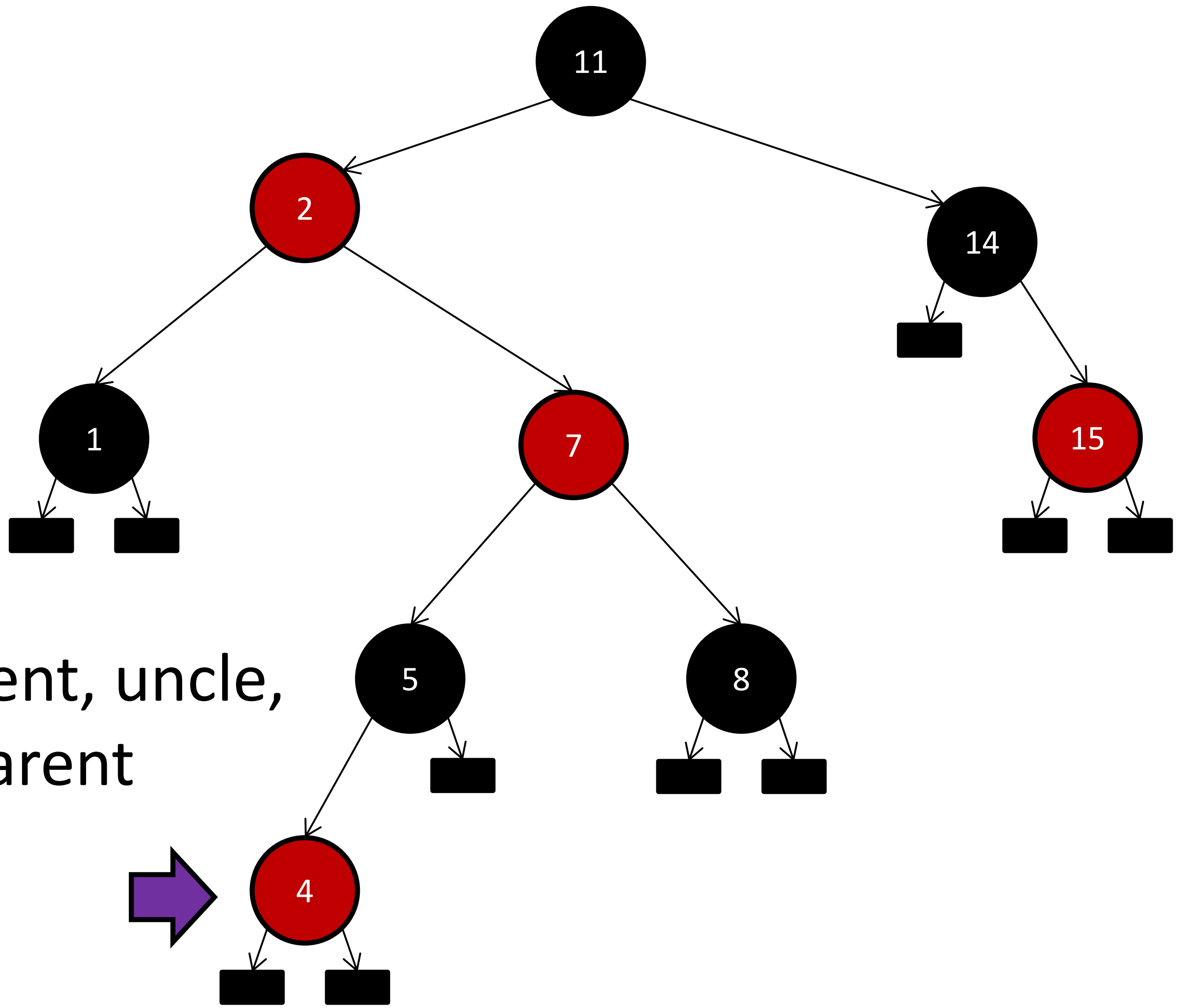
Insertion example (from CLR)



Invariant violation:

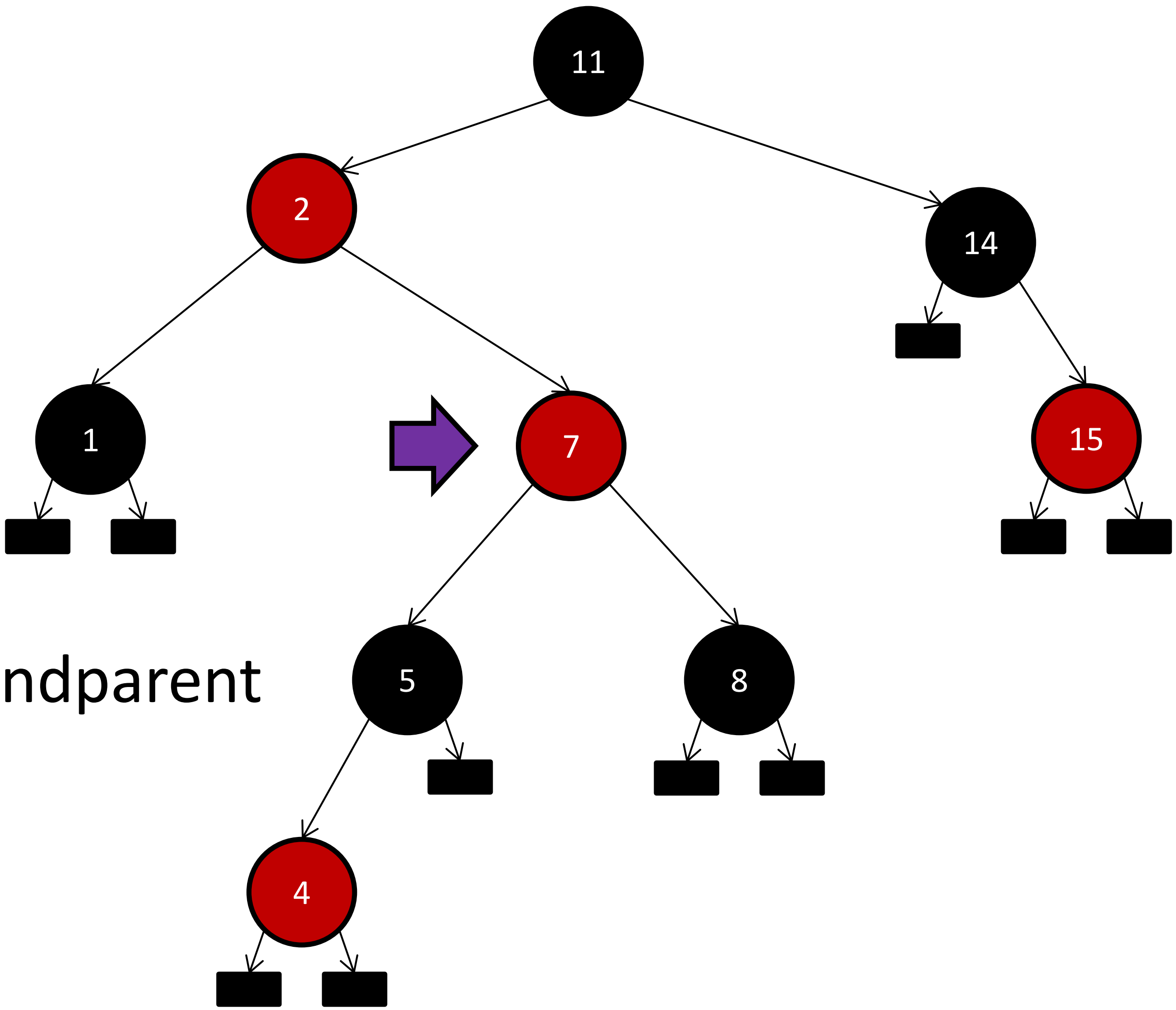
red node has red child

Insertion example (from CLR)



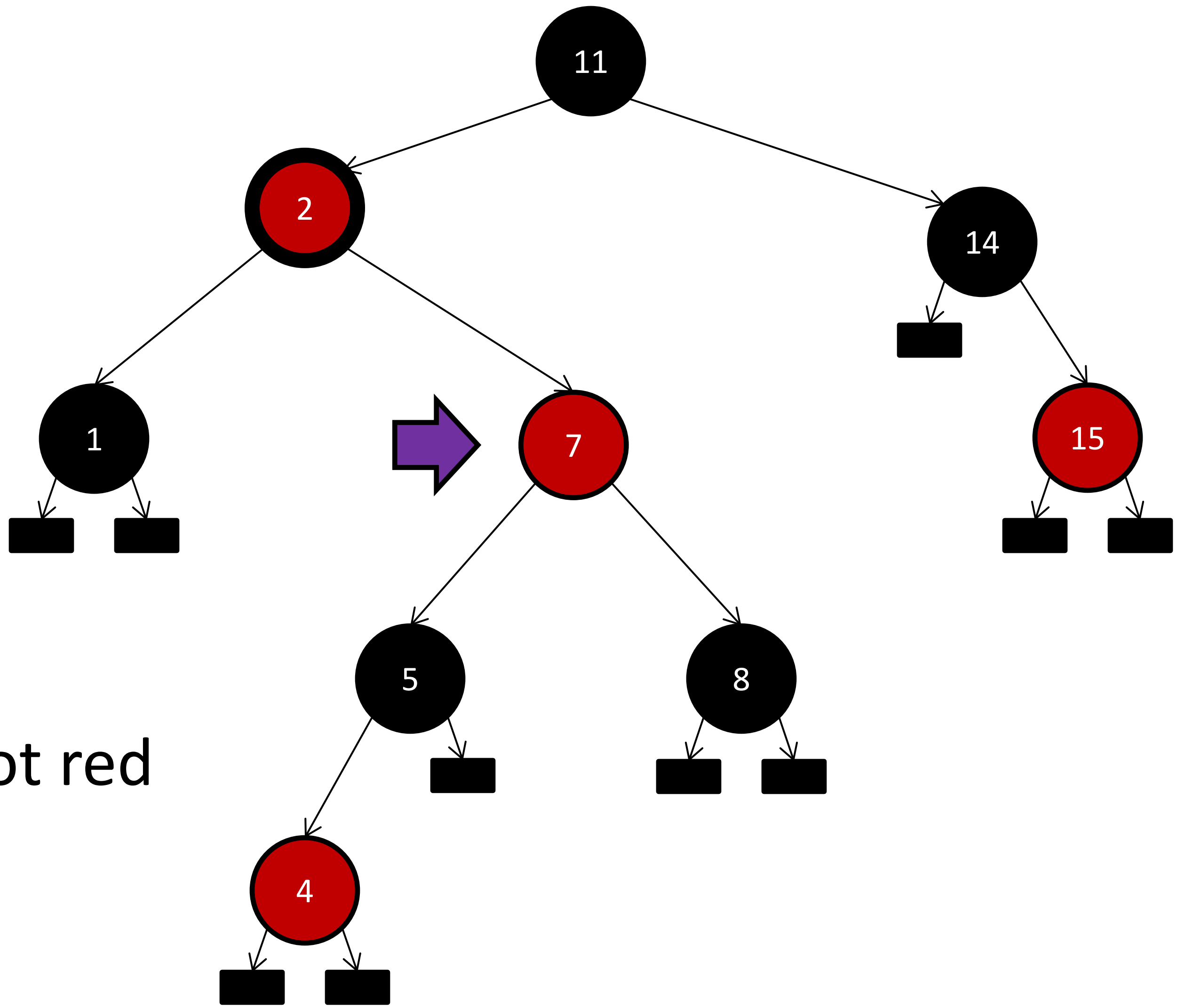
Recolor parent, uncle, and grandparent

Insertion example (from CLR)



Move to grandparent

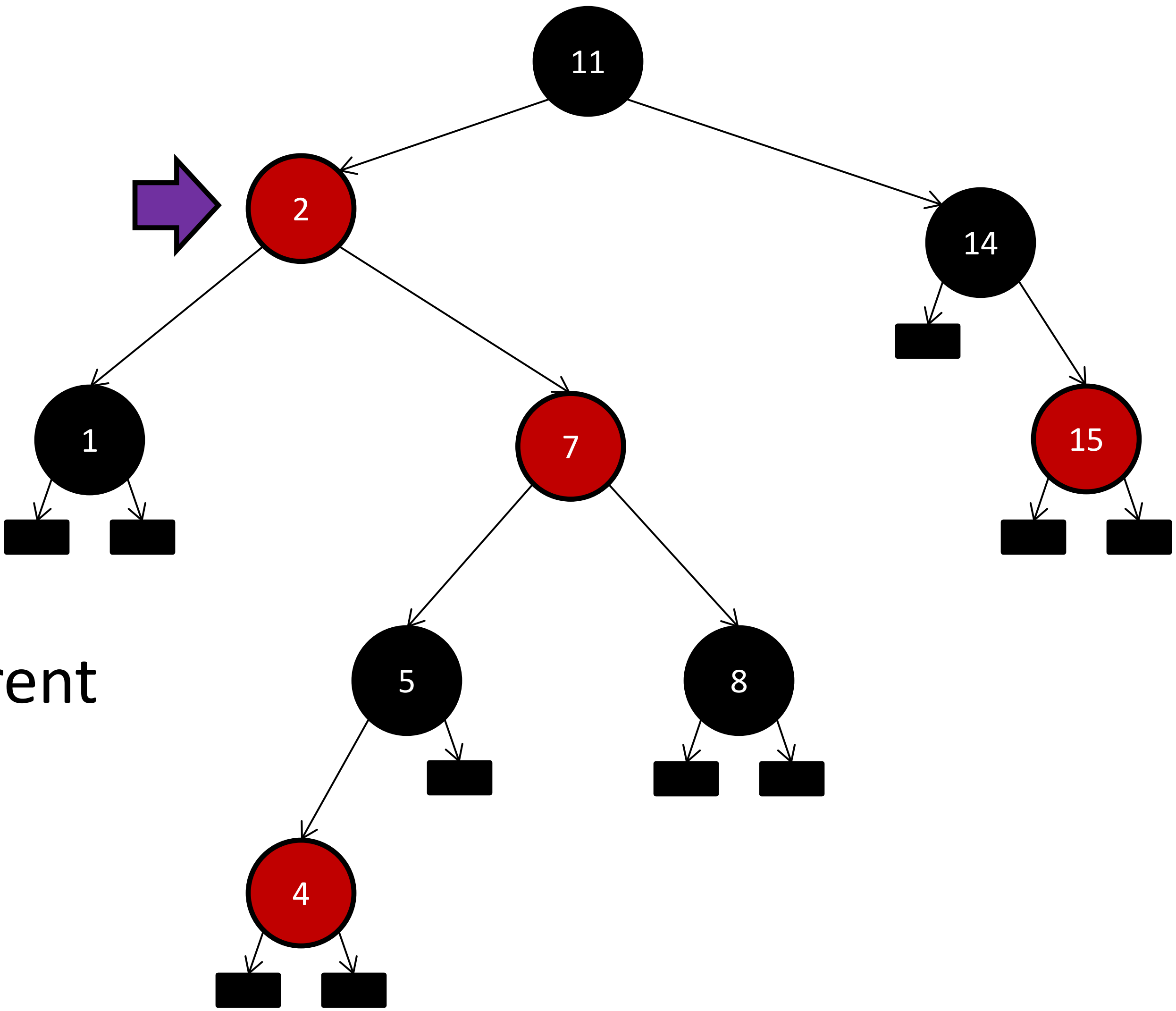
Insertion example (from CLR)



Violation

But uncle not red

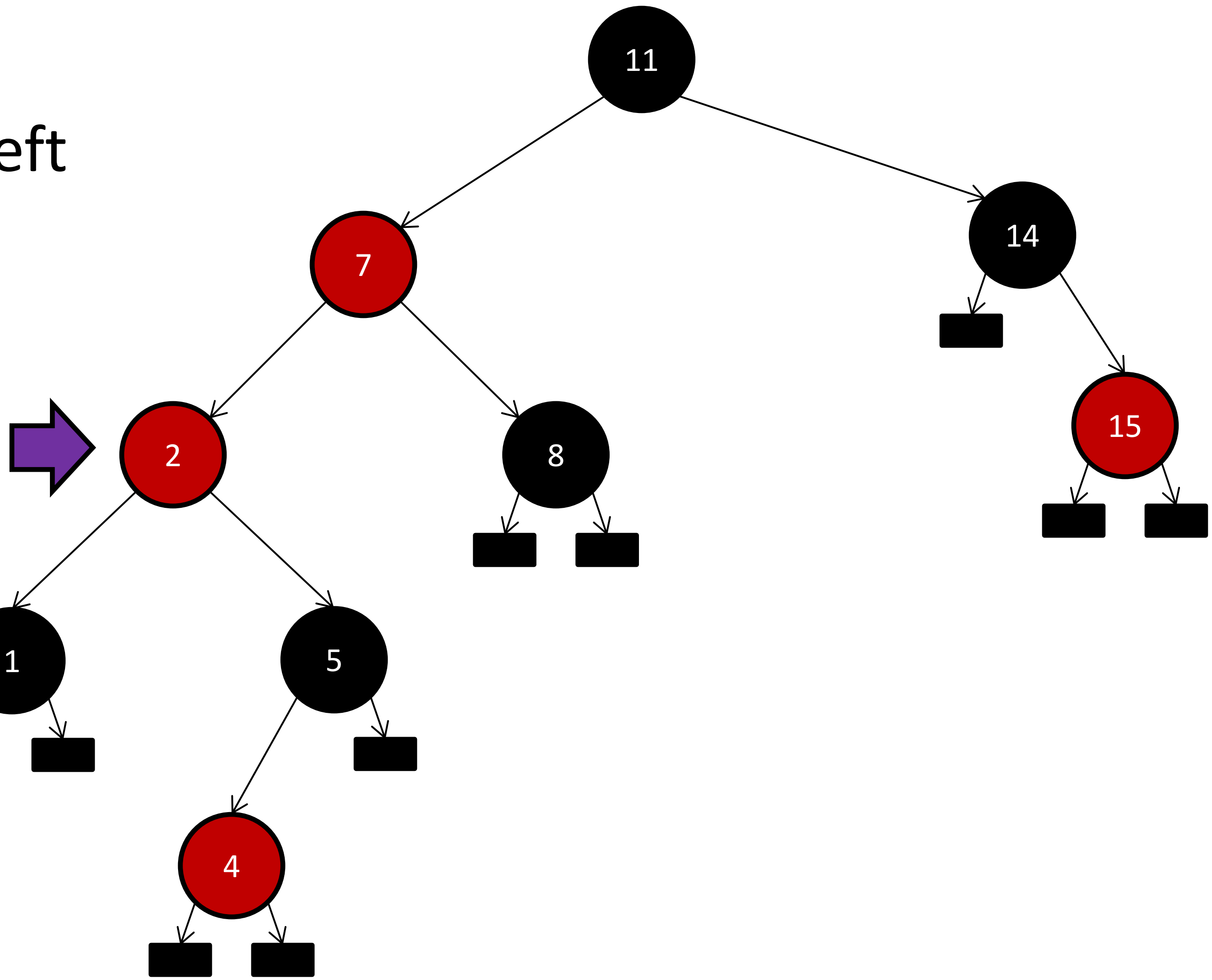
Insertion example (from CLR)



Move to parent

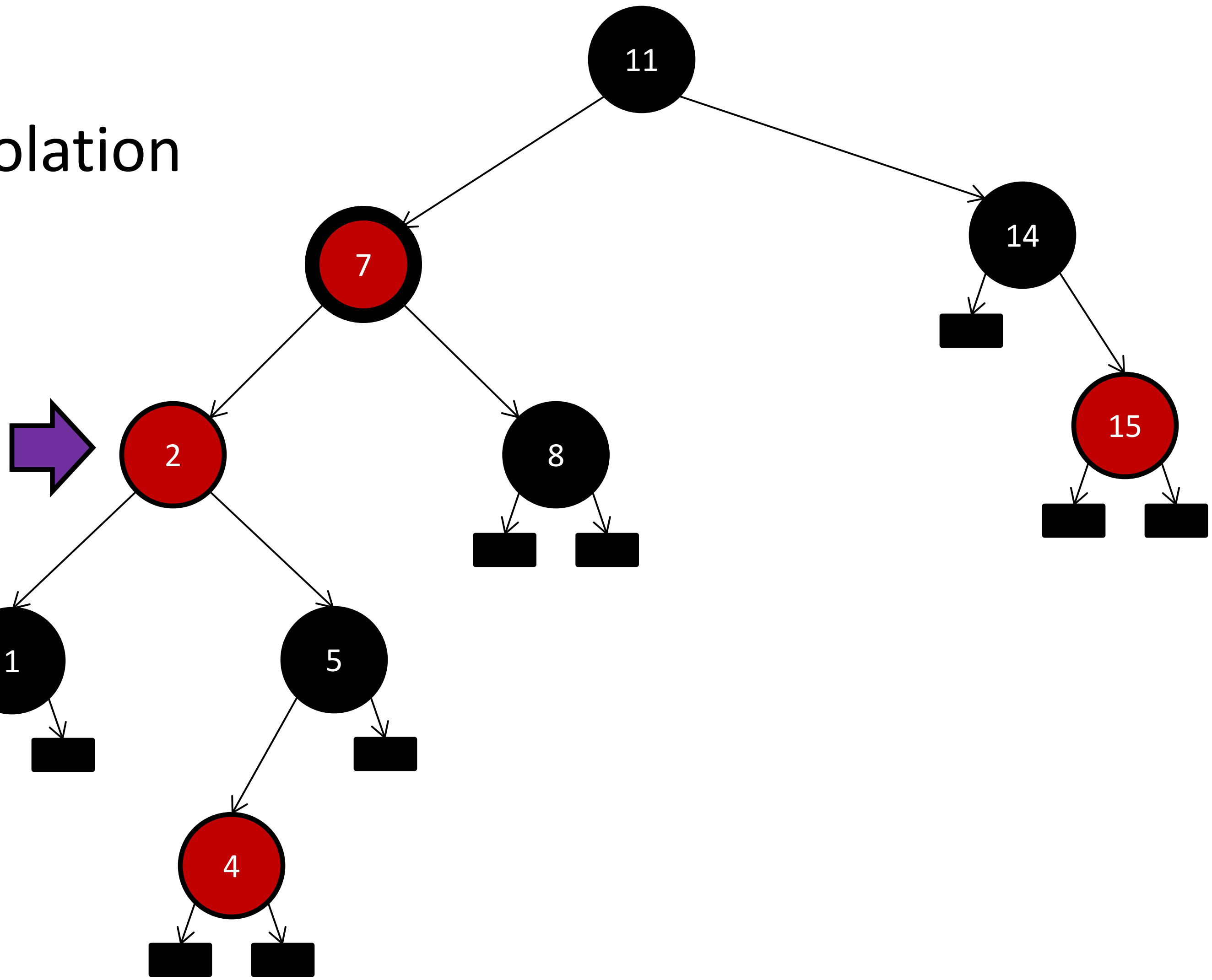
Insertion example (from CLR)

Rotate left



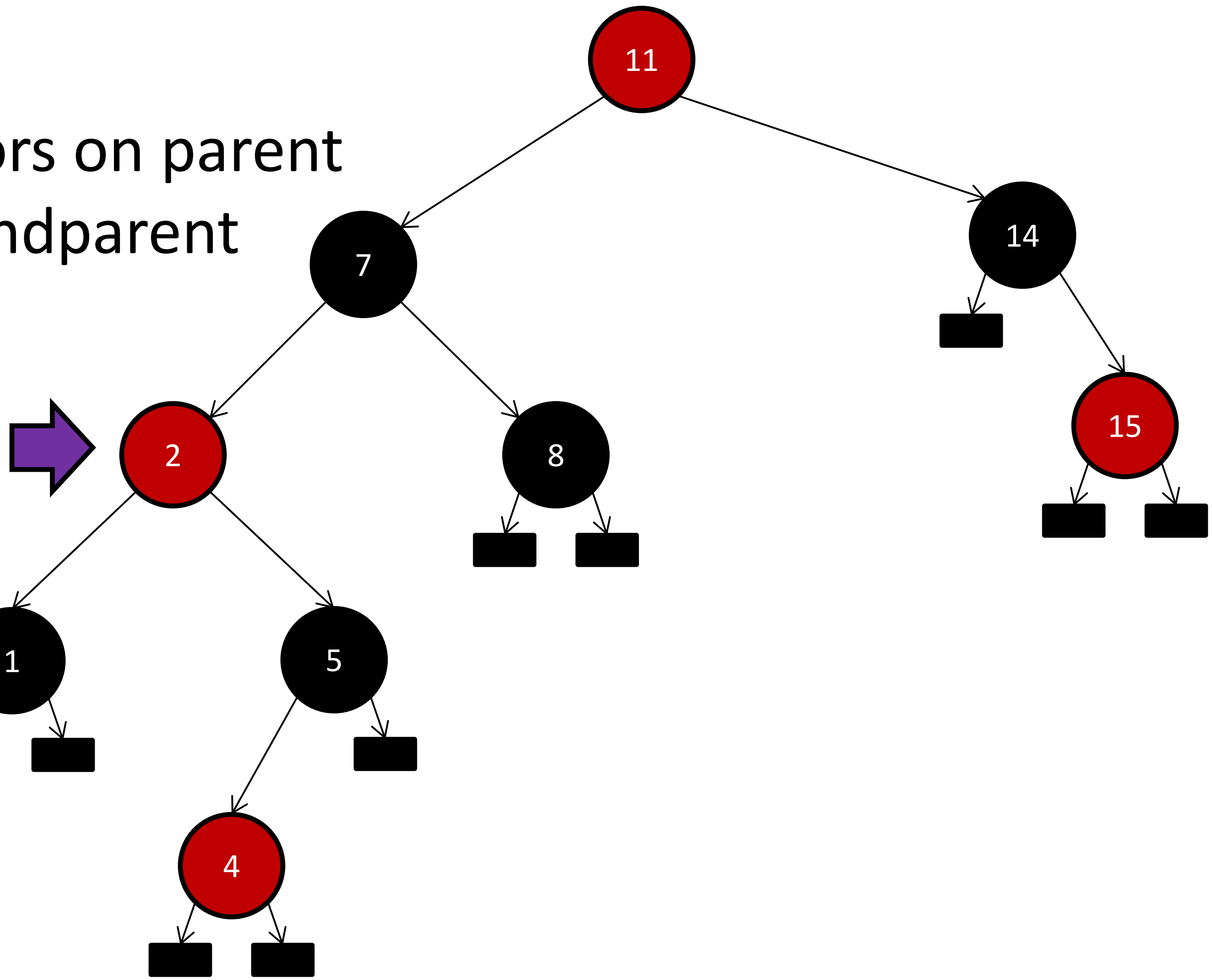
Insertion example (from CLR)

Still a violation



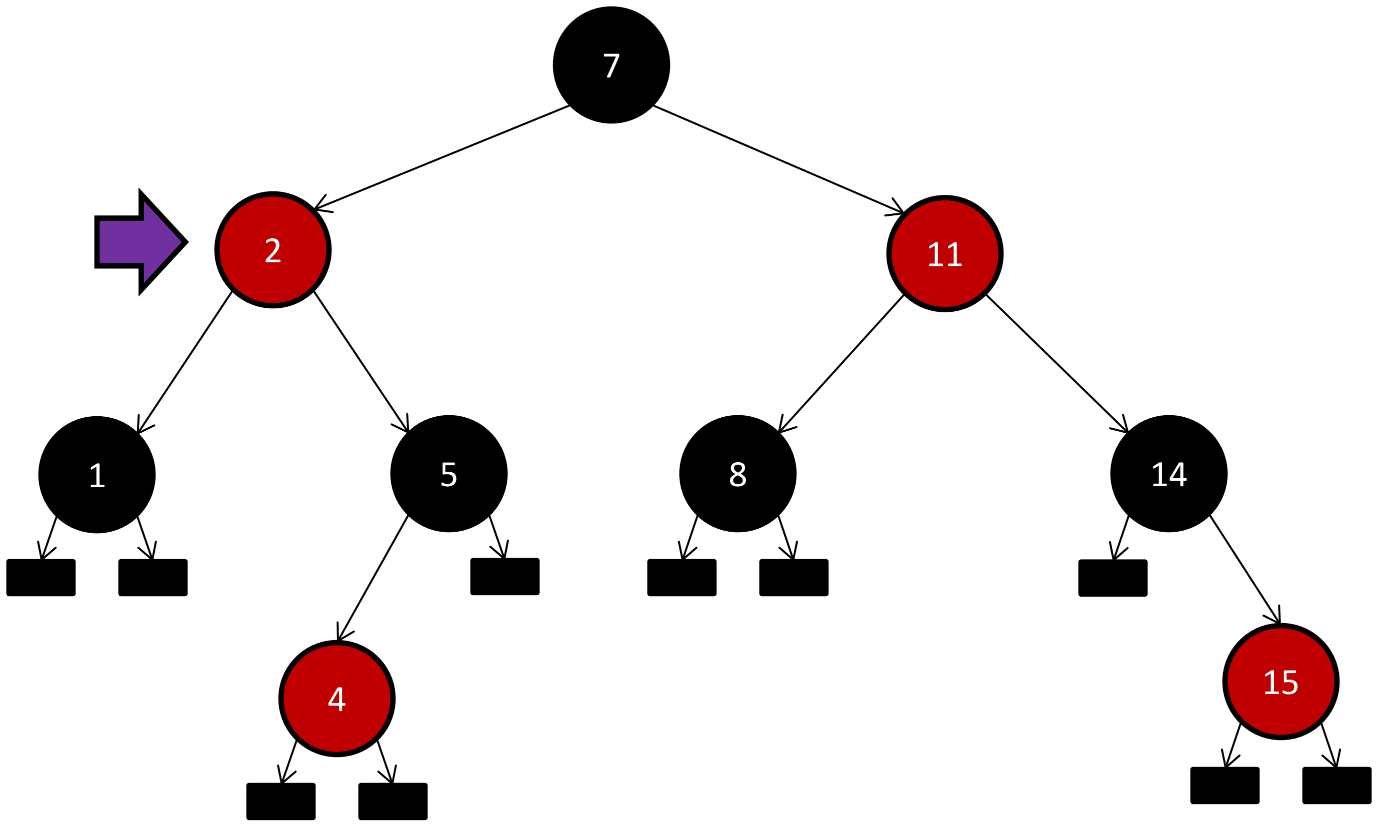
Insertion example (from CLR)

Flip colors on parent
and grandparent



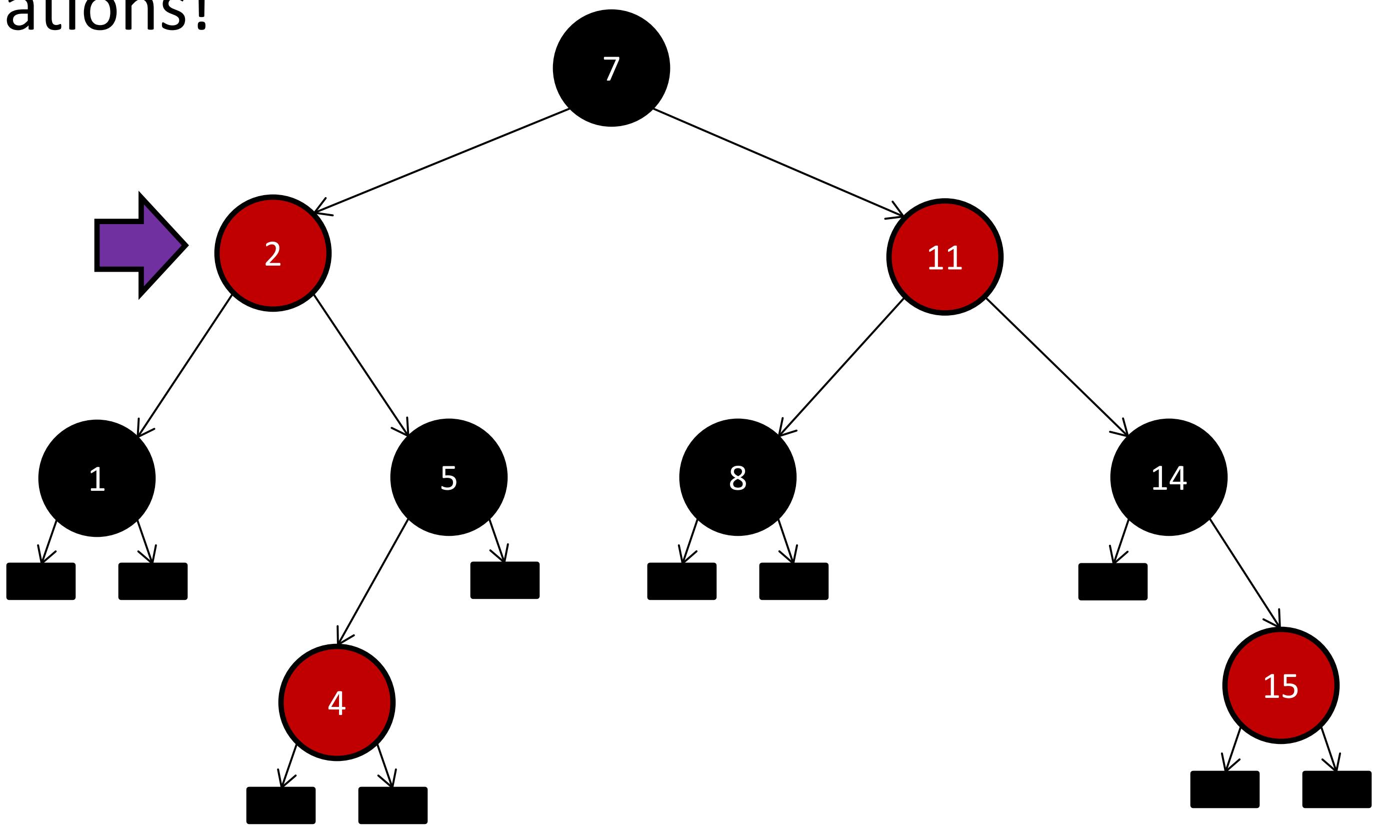
Insertion example (from CLR)

Rotate grandparent right



Insertion example (from CLR)

No violations!



Complexity analysis

- Insert new node and color it red
- While node not root
  and both node and parent are **red**
  - If uncle (parent's sibling) is also **red**
    - Flip colors of parent, uncle, and grandparent
    - Node = grandparent (i.e. check grandparent in next iteration)
  - Else (sibling is black)
    - If node is a right child
      - Node = parent
      - Rotate node left
    - Flip colors of parent and grandparent
    - Rotate grandparent right

Complexity analysis

- Insert new node and color it red $O(h)$
- While node not root
  and both node and parent are **red**
  - If uncle (parent's sibling) is also **red**
    - Flip colors of parent, uncle, and grandparent
    - Node = grandparent (i.e. check grandparent in next iteration)
  - Else (sibling is black)
    - If node is a right child
      - Node = parent
      - Rotate node left
    - Flip colors of parent and grandparent
    - Rotate grandparent right

Complexity analysis

- Insert new node and color it red $O(h)$
- While node not root
  and both node and parent are **red**
  - If uncle (parent's sibling) is also **red**
    - Flip colors of parent, uncle, and grandparent
    - Node = grandparent (i.e. check grandparent in next iteration)
  - Else (sibling is black)
    - If node is a right child
      - Node = parent
      - Rotate node left
    - Flip colors of parent and grandparent
    - Rotate grandparent right

$O(1)$

Complexity analysis

- Running time is:
  - $O(h)$ +
  - number of iterations $\times\, O(1)$

- How many iterations?
  - We start at the bottom
  - Move up for case 1 (red uncle)
    - Can happen at most h times
  - For cases 2 and 3 we end with a non-red parent
    - So no further iteration
- So the number of iterations is **bounded by $\boldsymbol{h}$**

Complexity analysis

- Running time is:

$$
\begin{aligned}
&O(h) + h \times O(1) \\
&= O(h) \\
&= O(\log n)
\end{aligned}
$$

Deletion

- Same basic idea
  - Run the standard deletion algorithm
  - Fix things up afterward through rotation and recoloring
  - However more subcases to worry about

Reading

- Read CLR chapter 13