Data Structures

Gabriel Istrate

April 8, 2025

Outline

- Binary Search Trees. Operations. Randomization.
- AVL trees.
- Splay trees
- Red-black trees

Binary Search Trees

- A binary search tree implements of a dynamic set
  - over a totally ordered domain

Binary Search Trees

- A binary search tree implements of a dynamic set
  - over a totally ordered domain
- Interface
  - Tree-Insert(T, k) adds a key k to the dictionary D
  - Tree-Delete(T, k) removes key k from D
  - Tree-Search(T, x) tells whether D contains a key k

Binary Search Trees

- A binary search tree implements of a dynamic set
  - over a totally ordered domain
- Interface
  - Tree-Insert(T, k) adds a key k to the dictionary D
  - Tree-Delete(T, k) removes key k from D
  - Tree-Search(T, x) tells whether D contains a key k
  - tree-walk: Inorder-Tree-Walk(T), etc.

Binary Search Trees

- A binary search tree implements of a dynamic set
  - over a totally ordered domain
- Interface
  - Tree-Insert(T, k) adds a key k to the dictionary D
  - Tree-Delete(T, k) removes key k from D
  - Tree-Search(T, x) tells whether D contains a key k
  - tree-walk: Inorder-Tree-Walk(T), etc.
  - Tree-Minimum(T) finds the smallest element in the tree
  - Tree-Maximum(T) finds the largest element in the tree

Binary Search Trees

- A binary search tree implements of a dynamic set
  - over a totally ordered domain
- Interface
  - Tree-Insert(T, k) adds a key k to the dictionary D
  - Tree-Delete(T, k) removes key k from D
  - Tree-Search(T, x) tells whether D contains a key k
  - tree-walk: Inorder-Tree-Walk(T), etc.
  - Tree-Minimum(T) finds the smallest element in the tree
  - Tree-Maximum(T) finds the largest element in the tree
  - iteration: Tree-Successor(x) and Tree-Predecessor(x) find the successor and predecessor, respectively, of an element x

Binary Search Trees (2)

- Implementation
  - T represents the tree, which consists of a set of nodes

Binary Search Trees (2)

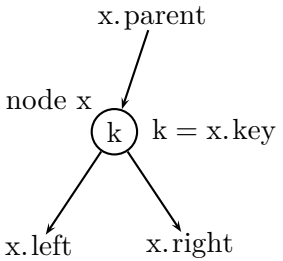
- Implementation
  - T represents the tree, which consists of a set of nodes
  - T.root is the root node of tree T

Binary Search Trees (2)

- Implementation
  - T represents the tree, which consists of a set of nodes
  - T.root is the root node of tree T

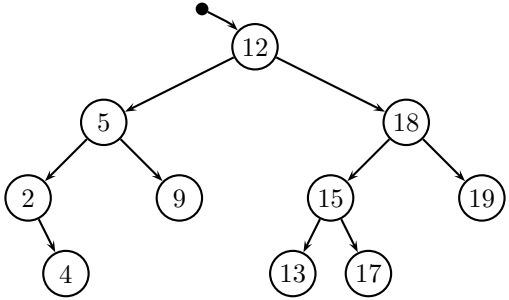
Node x

- x.parent is the parent of node x
- x.key is the key stored in node x
- x.left is the left child of node x
- x.right is the right child of node x

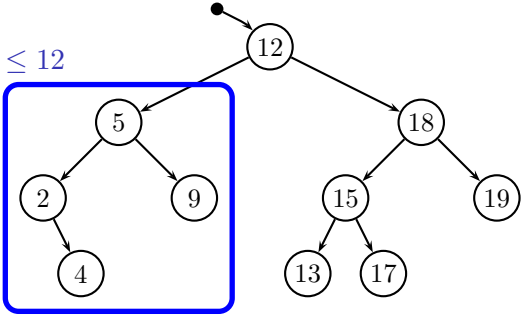


Binary Search Trees (3)

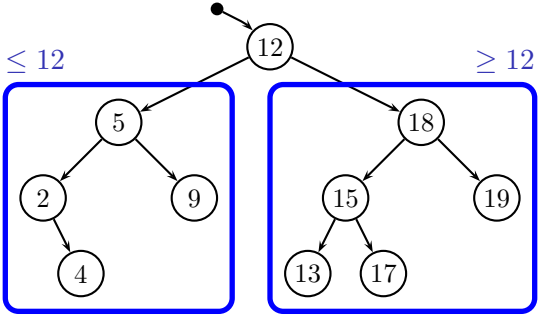
Binary Search Trees (3)



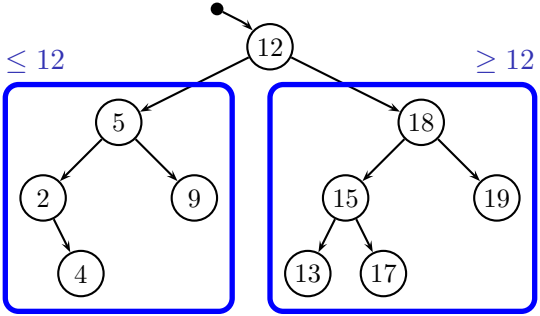
Binary Search Trees (3)



Binary Search Trees (3)



Binary Search Trees (3)

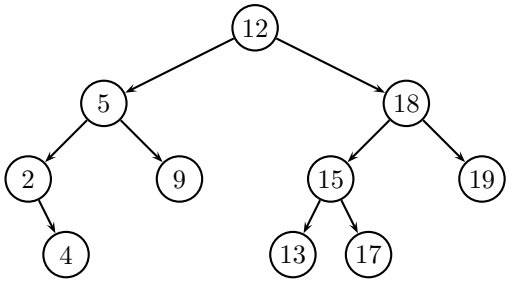


- Binary-search-tree property
  - for all nodes x, y, and z
  - $y \in$ left-subtree(x) $\Rightarrow y.\text{key} \leq x.\text{key}$
  - $z \in$ right-subtree(x) $\Rightarrow z.\text{key} \geq x.\text{key}$

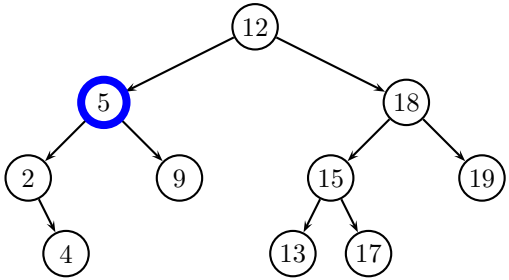
Successor and Predecessor

- Given a node x, find the node containing the next key value

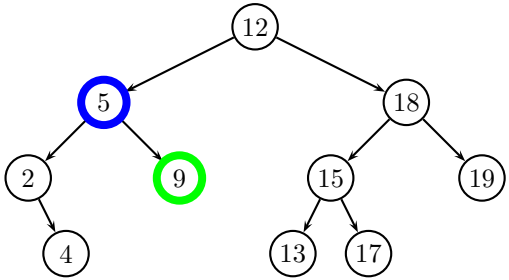
- Given a node x, find the node containing the next key value



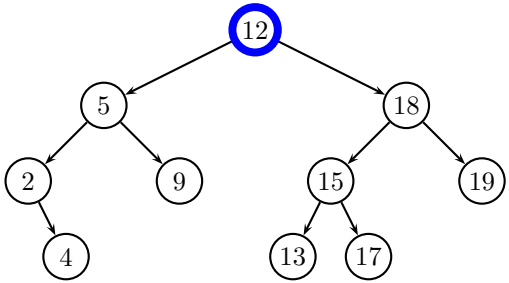
- Given a node x, find the node containing the next key value



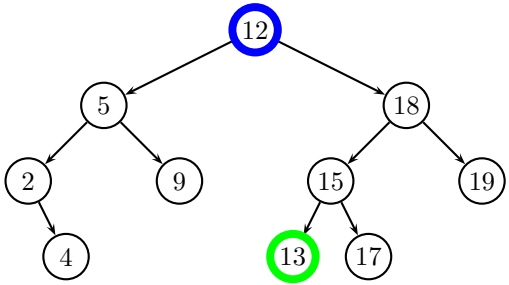
- Given a node x, find the node containing the next key value



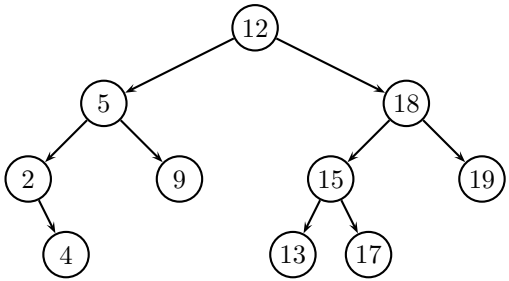
- Given a node x, find the node containing the next key value



- Given a node x, find the node containing the next key value

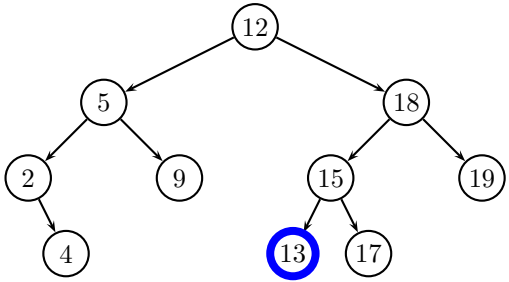


- Given a node x, find the node containing the next key value



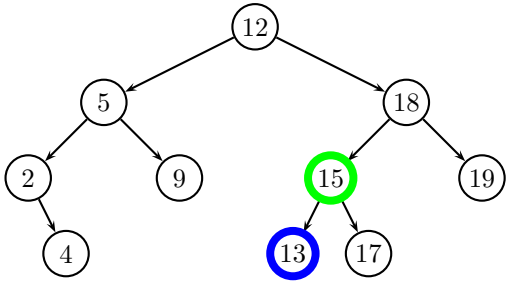
- The successor of x is the minimum of the right subtree of x, if that exists

- Given a node x, find the node containing the next key value



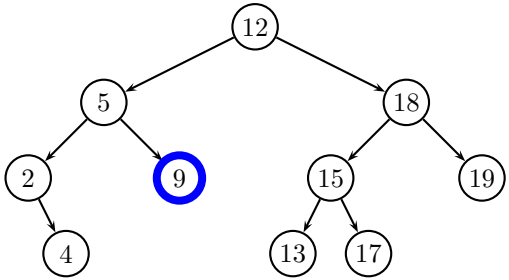
- The successor of x is the minimum of the right subtree of x, if that exists

- Given a node x, find the node containing the next key value



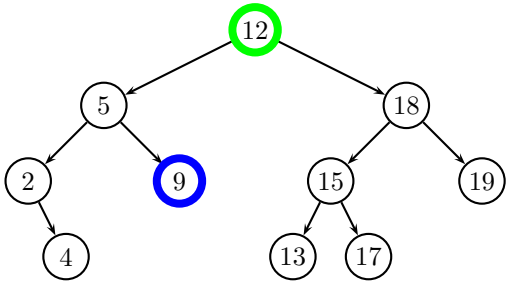
- The successor of x is the minimum of the right subtree of x, if that exists

- Given a node x, find the node containing the next key value



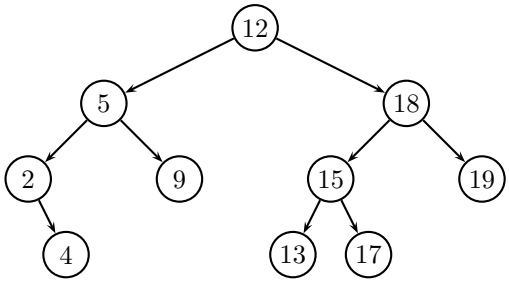
- The successor of x is the minimum of the right subtree of x, if that exists

- Given a node x, find the node containing the next key value



- The successor of x is the minimum of the right subtree of x, if that exists

- Given a node x, find the node containing the next key value



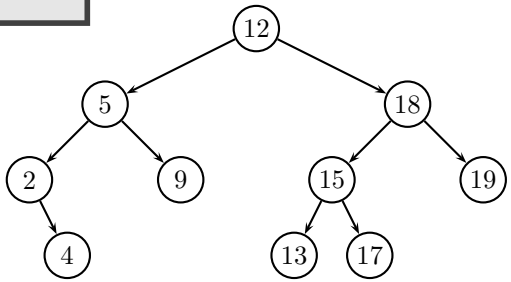
- The successor of x is the minimum of the right subtree of x, if that exists
- Otherwise it is the first ancestor a of x such that x falls in the left subtree of a

Successor and Predecessor(2)

```
Tree-Successor(x)1  if x.right ≠ nil
                2       return Tree-Minimum(x.right)
                3  y = x.parent
                4  while y ≠ nil and x = y.right
                5       x = y
                6       y = y.parent
                7  return y
```

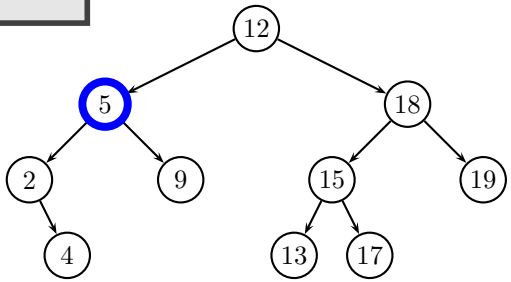
Successor and Predecessor(2)

```
Tree-Successor(x)1   if x.right ≠ nil
                 2       return Tree-Minimum(x.right)
                 3   y = x.parent
                 4   while y ≠ nil and x = y.right
                 5       x = y
                 6       y = y.parent
                 7   return y
```



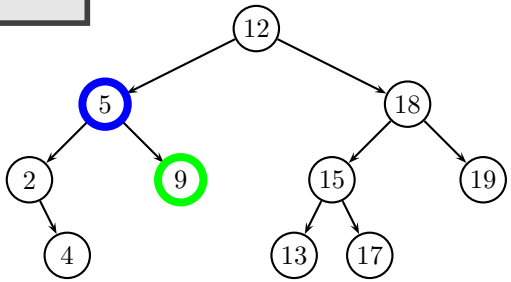
Successor and Predecessor(2)

```
Tree-Successor(x)1   if x.right ≠ nil
                 2       return Tree-Minimum(x.right)
                 3   y = x.parent
                 4   while y ≠ nil and x = y.right
                 5       x = y
                 6       y = y.parent
                 7   return y
```



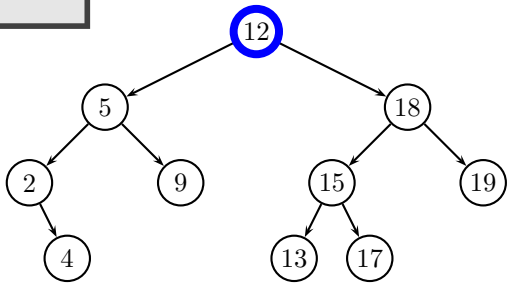
Successor and Predecessor(2)

```
Tree-Successor(x)1   if x.right ≠ nil
                 2       return Tree-Minimum(x.right)
                 3   y = x.parent
                 4   while y ≠ nil and x = y.right
                 5       x = y
                 6       y = y.parent
                 7   return y
```



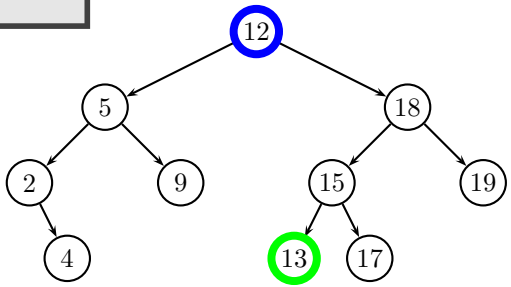
Successor and Predecessor(2)

```
Tree-Successor(x)1   if x.right ≠ nil
                 2       return Tree-Minimum(x.right)
                 3   y = x.parent
                 4   while y ≠ nil and x = y.right
                 5       x = y
                 6       y = y.parent
                 7   return y
```



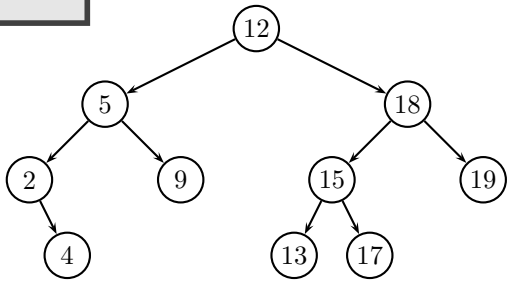
Successor and Predecessor(2)

```
Tree-Successor(x)1   if x.right ≠ nil
                 2       return Tree-Minimum(x.right)
                 3   y = x.parent
                 4   while y ≠ nil and x = y.right
                 5       x = y
                 6       y = y.parent
                 7   return y
```



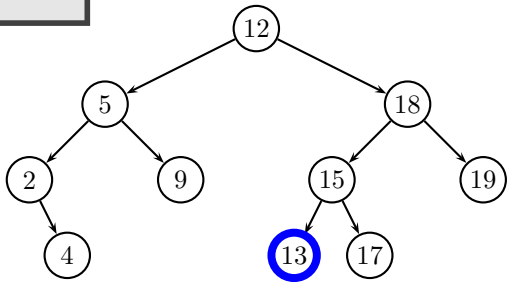
Successor and Predecessor(2)

```
Tree-Successor(x)1   if x.right ≠ nil
                 2       return Tree-Minimum(x.right)
                 3   y = x.parent
                 4   while y ≠ nil and x = y.right
                 5       x = y
                 6       y = y.parent
                 7   return y
```



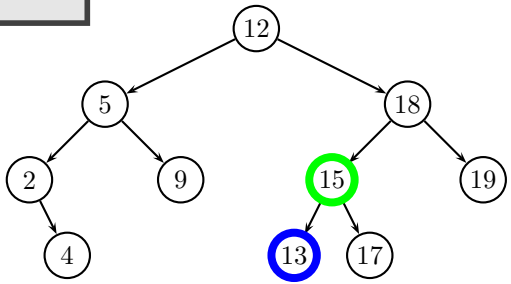
Successor and Predecessor(2)

```
Tree-Successor(x)1   if x.right ≠ nil
                 2       return Tree-Minimum(x.right)
                 3   y = x.parent
                 4   while y ≠ nil and x = y.right
                 5       x = y
                 6       y = y.parent
                 7   return y
```



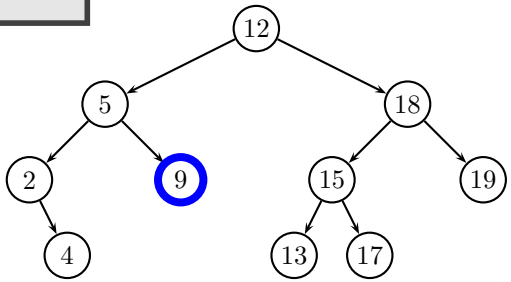
Successor and Predecessor(2)

```
Tree-Successor(x)1   if x.right ≠ nil
                 2       return Tree-Minimum(x.right)
                 3   y = x.parent
                 4   while y ≠ nil and x = y.right
                 5       x = y
                 6       y = y.parent
                 7   return y
```



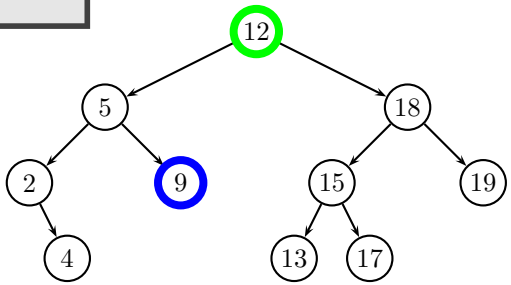
Successor and Predecessor(2)

```
Tree-Successor(x)1   if x.right ≠ nil
                 2       return Tree-Minimum(x.right)
                 3   y = x.parent
                 4   while y ≠ nil and x = y.right
                 5       x = y
                 6       y = y.parent
                 7   return y
```



Successor and Predecessor(2)

```
Tree-Successor(x)1   if x.right ≠ nil
                 2       return Tree-Minimum(x.right)
                 3   y = x.parent
                 4   while y ≠ nil and x = y.right
                 5       x = y
                 6       y = y.parent
                 7   return y
```



Search

- Binary search (thus the name of the tree)

- Binary search (thus the name of the tree)

```
Tree-Search(x, k)1   if x = nil or k = x.key
                 2       return x
                 3   if k < x.key
                 4       return Tree-Search(x.left, k)
                 5   else return Tree-Search(x.right, k)
```

- Binary search (thus the name of the tree)

```
Tree-Search(x, k)1   if x = nil or k = x.key
                 2       return x
                 3   if k < x.key
                 4       return Tree-Search(x.left, k)
                 5   else return Tree-Search(x.right, k)
```

- Is this correct?

- Binary search (thus the name of the tree)

```
Tree-Search(x, k)1  if x = nil or k = x.key
                2       return x
                3  if k < x.key
                4       return Tree-Search(x.left, k)
                5  else return Tree-Search(x.right, k)
```

- Is this correct? Yes, thanks to the binary-search-tree property

- Binary search (thus the name of the tree)

```
Tree-Search(x, k)1   if x = nil or k = x.key
                 2       return x
                 3   if k < x.key
                 4       return Tree-Search(x.left, k)
                 5   else return Tree-Search(x.right, k)
```

- Is this correct? Yes, thanks to the binary-search-tree property
- Complexity?

- Binary search (thus the name of the tree)

```
Tree-Search(x, k)1   if x = nil or k = x.key
                 2       return x
                 3   if k < x.key
                 4       return Tree-Search(x.left, k)
                 5   else return Tree-Search(x.right, k)
```

- Is this correct? Yes, thanks to the binary-search-tree property
- Complexity?

$$\boxed{T(n) = \Theta(\text{depth of the tree})}$$

- Binary search (thus the name of the tree)

```
Tree-Search(x, k)1   if x = nil or k = x.key
                 2       return x
                 3   if k < x.key
                 4       return Tree-Search(x.left, k)
                 5   else return Tree-Search(x.right, k)
```

- Is this correct? Yes, thanks to the binary-search-tree property
- Complexity?

$$\boxed{T(n) = \Theta(\text{depth of the tree})}$$

$$\boxed{T(n) = O(n)}$$

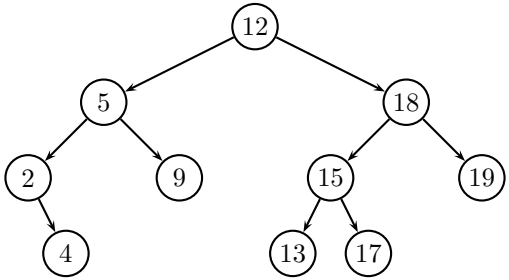
Search (2)

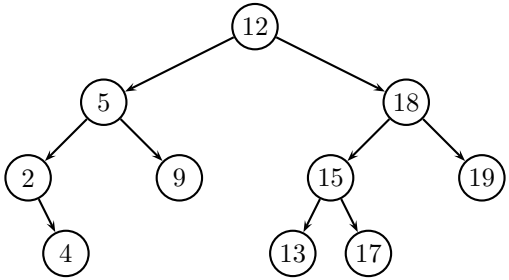
Search (2)

- Iterative binary search

- Iterative binary search

```
Iterative-Tree-Search(T, k)1   x = T.root
                           2   while x ≠ nil ∧ k ≠ x.key
                           3       if k < x.key
                           4           x = x.left
                           5       else x = x.right
                           6   return x
```


12
5
18
2
9
15
19
4
13
17



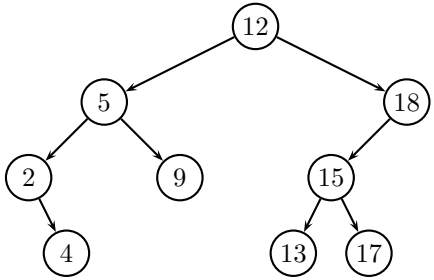
- Idea
  - in order to insert x, we search for x (more precisely x.key)
  - if we don't find it, we add it where the search stopped

Insertion (2)

```
Tree-Insert(T, z)  1   y = nil
                   2   x = T.root
                   3   while x ≠ nil
                   4         y = x
                   5         if z.key < x.key
                   6               x = x.left
                   7         else x = x.right
                   8   z.parent = y
                   9   if y = nil
                   10        T.root = z
                   11  else if z.key < y.key
                   12              y.left = z
                   13        else y.right = z
```

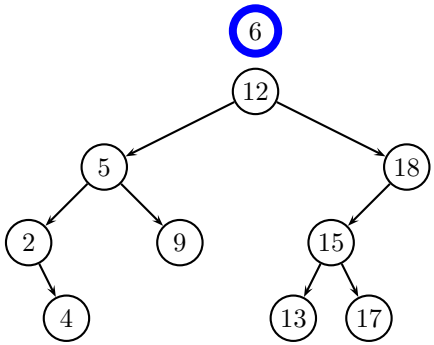
Insertion (2)

```
Tree-Insert(T, z)  1   y = nil
                   2   x = T.root
                   3   while x ≠ nil
                   4         y = x
                   5         if z.key < x.key
                   6               x = x.left
                   7         else x = x.right
                   8   z.parent = y
                   9   if y = nil
                  10         T.root = z
                  11   else if z.key < y.key
                  12               y.left = z
                  13         else y.right = z
```



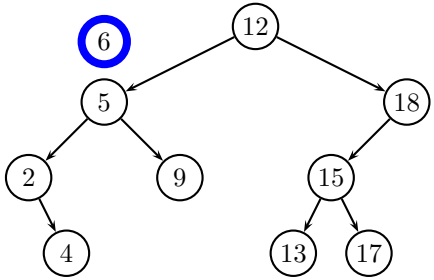
Insertion (2)

```
Tree-Insert(T, z)  1   y = nil
                   2   x = T.root
                   3   while x ≠ nil
                   4         y = x
                   5         if z.key < x.key
                   6               x = x.left
                   7         else x = x.right
                   8   z.parent = y
                   9   if y = nil
                  10         T.root = z
                  11   else if z.key < y.key
                  12               y.left = z
                  13         else y.right = z
```



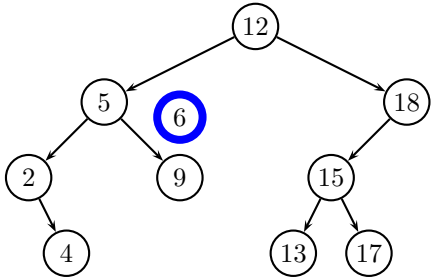
Insertion (2)

```
Tree-Insert(T, z)  1   y = nil
                   2   x = T.root
                   3   while x ≠ nil
                   4         y = x
                   5         if z.key < x.key
                   6               x = x.left
                   7         else x = x.right
                   8   z.parent = y
                   9   if y = nil
                  10         T.root = z
                  11   else if z.key < y.key
                  12               y.left = z
                  13         else y.right = z
```



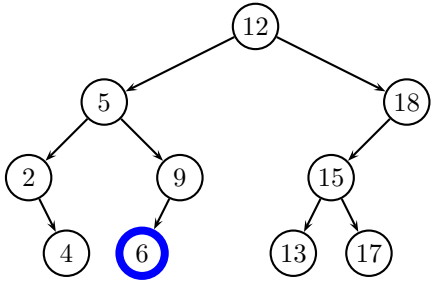
Insertion (2)

```
Tree-Insert(T, z)  1   y = nil
                   2   x = T.root
                   3   while x ≠ nil
                   4         y = x
                   5         if z.key < x.key
                   6               x = x.left
                   7         else x = x.right
                   8   z.parent = y
                   9   if y = nil
                  10         T.root = z
                  11   else if z.key < y.key
                  12               y.left = z
                  13         else y.right = z
```



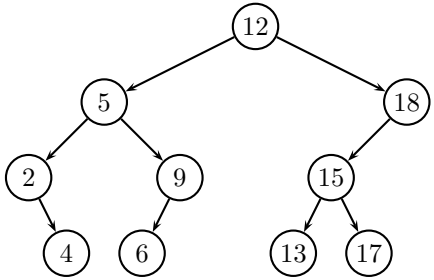
Insertion (2)

```
Tree-Insert(T, z)  1   y = nil
                   2   x = T.root
                   3   while x ≠ nil
                   4         y = x
                   5         if z.key < x.key
                   6               x = x.left
                   7         else x = x.right
                   8   z.parent = y
                   9   if y = nil
                  10         T.root = z
                  11   else if z.key < y.key
                  12               y.left = z
                  13         else y.right = z
```



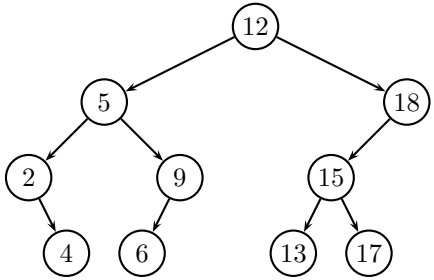
Insertion (2)

```
Tree-Insert(T, z)  1   y = nil
                   2   x = T.root
                   3   while x ≠ nil
                   4         y = x
                   5         if z.key < x.key
                   6               x = x.left
                   7         else x = x.right
                   8   z.parent = y
                   9   if y = nil
                  10         T.root = z
                  11   else if z.key < y.key
                  12               y.left = z
                  13         else y.right = z
```



Insertion (2)

```
Tree-Insert(T, z)  1   y = nil
                   2   x = T.root
                   3   while x ≠ nil
                   4         y = x
                   5         if z.key < x.key
                   6               x = x.left
                   7         else x = x.right
                   8   z.parent = y
                   9   if y = nil
                  10         T.root = z
                  11   else if z.key < y.key
                  12               y.left = z
                  13         else y.right = z
```



$$T(n) = \Theta(h)$$

Observation

- Both insertion and search operations have complexity h, where h is the height of the tree

Observation

- Both insertion and search operations have complexity h, where h is the height of the tree
- $h = O(\log n)$ in the average case
  - i.e., with a random insertion order

Observation

- Both insertion and search operations have complexity h, where h is the height of the tree
- $h = O(\log n)$ in the average case
  - i.e., with a random insertion order
- $h = O(n)$ in some particular cases

Observation

- Both insertion and search operations have complexity h, where h is the height of the tree
- $h = O(\log n)$ in the average case
  - i.e., with a random insertion order
- $h = O(n)$ in some particular cases
  - i.e., with ordered sequences

Observation

- Both insertion and search operations have complexity h, where h is the height of the tree
- $h = O(\log n)$ in the average case
  - i.e., with a random insertion order
- $h = O(n)$ in some particular cases
  - i.e., with ordered sequences
  - the problem is that the “worst” case is not that uncommon

Observation

- Both insertion and search operations have complexity h, where h is the height of the tree
- $h = O(\log n)$ in the average case
  - i.e., with a random insertion order
- $h = O(n)$ in some particular cases
  - i.e., with ordered sequences
  - the problem is that the "worst" case is not that uncommon
- First Idea: use randomization to turn all cases in the average case

Randomized Insertion

- Idea 1: insert every sequence as a random sequence

Randomized Insertion

- Idea 1: insert every sequence as a random sequence
  - i.e., given $A = \langle 1, 2, 3, \ldots, n \rangle$, insert a random permutation of A

Randomized Insertion

- Idea 1: insert every sequence as a random sequence
  - i.e., given $A = \langle 1, 2, 3, \ldots, n \rangle$, insert a random permutation of A
  - problem: A is not necessarily known in advance

Randomized Insertion

- Idea 1: insert every sequence as a random sequence
  - i.e., given $A = \langle 1, 2, 3, \ldots, n \rangle$, insert a random permutation of A
  - problem: A is not necessarily known in advance
- Idea 2: we can obtain a random permutation of the input sequence by randomly alternating two insertion procedures
  - tail insertion: this is what Tree-Insert does

Randomized Insertion

- Idea 1: insert every sequence as a random sequence
  - i.e., given $A = \langle 1, 2, 3, \ldots, n \rangle$, insert a random permutation of A
  - problem: A is not necessarily known in advance
- Idea 2: we can obtain a random permutation of the input sequence by randomly alternating two insertion procedures
  - tail insertion: this is what Tree-Insert does
  - head insertion: for this we need a new procedure Tree-Root-Insert
    - ⋆ inserts n in T as if n was inserted as the first element

Randomized Insertion (2)

```
Tree-Randomized-Insert1(T, z)1   r = uniformly rand. val. from {1, ..., t.size + 1}
                             2   if r = 1
                             3       Tree-Root-Insert(T, z)
                             4   else Tree-Insert(T, z)
```

Randomized Insertion (2)

```
Tree-Randomized-Insert1(T, z)1   r = uniformly rand. val. from {1, ..., t.size + 1}
                             2   if r = 1
                             3       Tree-Root-Insert(T, z)
                             4   else Tree-Insert(T, z)
```

- Does this really simulate a random permutation?
  - i.e., with all permutations being equally likely?

Randomized Insertion (2)

```
Tree-Randomized-Insert1(T, z)1   r = uniformly rand. val. from {1, ..., t.size + 1}
                              2  if r = 1
                              3      Tree-Root-Insert(T, z)
                              4  else Tree-Insert(T, z)
```

- Does this really simulate a random permutation?
  - i.e., with all permutations being equally likely?
  - no, clearly the last element can only go to the top or to the bottom

Randomized Insertion (2)

```
Tree-Randomized-Insert1(T, z)1   r = uniformly rand. val. from {1, ..., t.size + 1}
                              2   if r = 1
                              3       Tree-Root-Insert(T, z)
                              4   else Tree-Insert(T, z)
```

- Does this really simulate a random permutation?
  - i.e., with all permutations being equally likely?
  - no, clearly the last element can only go to the top or to the bottom
- It is true that any node has the same probability of being inserted at the top

Randomized Insertion (2)

```
Tree-Randomized-Insert1(T, z)1   r = uniformly rand. val. from {1, ..., t.size + 1}
                             2   if r = 1
                             3       Tree-Root-Insert(T, z)
                             4   else Tree-Insert(T, z)
```

- Does this really simulate a random permutation?
  - i.e., with all permutations being equally likely?
  - no, clearly the last element can only go to the top or to the bottom
- It is true that any node has the same probability of being inserted at the top
  - this suggests a recursive application of this same procedure

Randomized Insertion (3)

Randomized Insertion (3)

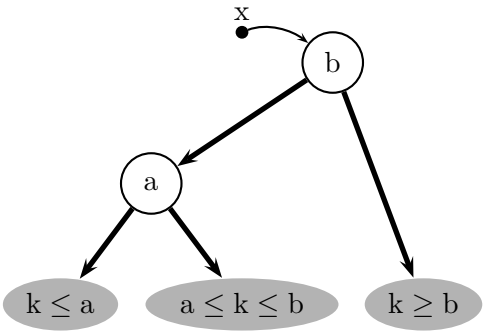
```
Tree-Randomized-Insert(t, z)  1  if t = nil
                              2      return z
                              3  r = uniformly random value from {1, ..., t.size + 1}
                              4  if r = 1                    // Pr[r = 1] = 1/(t.size + 1)
                              5      z.size = t.size + 1
                              6      return Tree-Root-Insert(t, z)
                              7  if z.key < t.key
                              8      t.left = Tree-Randomized-Insert(t.left, z)
                              9  else t.right = Tree-Randomized-Insert(t.right, z)
                             10  t.size = t.size + 1
                             11  return t
```

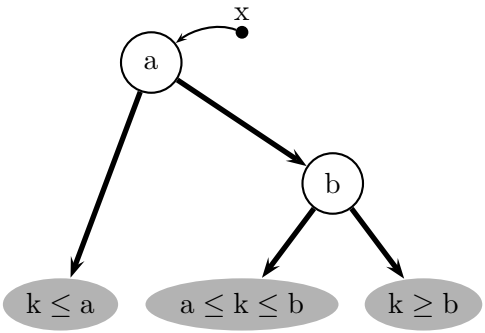
Randomized Insertion (3)

```
Tree-Randomized-Insert(t, z)  1  if t = nil
                              2       return z
                              3  r = uniformly random value from {1, ..., t.size + 1}
                              4  if r = 1                    // Pr[r = 1] = 1/(t.size + 1)
                              5       z.size = t.size + 1
                              6       return Tree-Root-Insert(t, z)
                              7  if z.key < t.key
                              8       t.left = Tree-Randomized-Insert(t.left, z)
                              9  else t.right = Tree-Randomized-Insert(t.right, z)
                             10  t.size = t.size + 1
                             11  return t
```

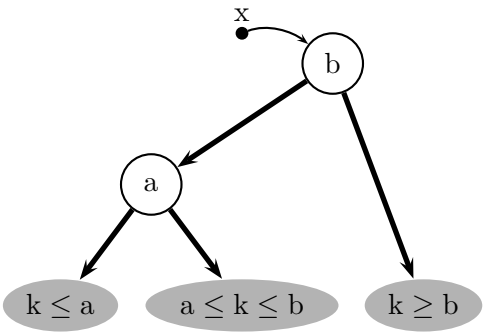
- Looks like this one really simulates a random permutation…

Rotation


x
b
a
k ≤ a
a ≤ k ≤ b
k ≥ b

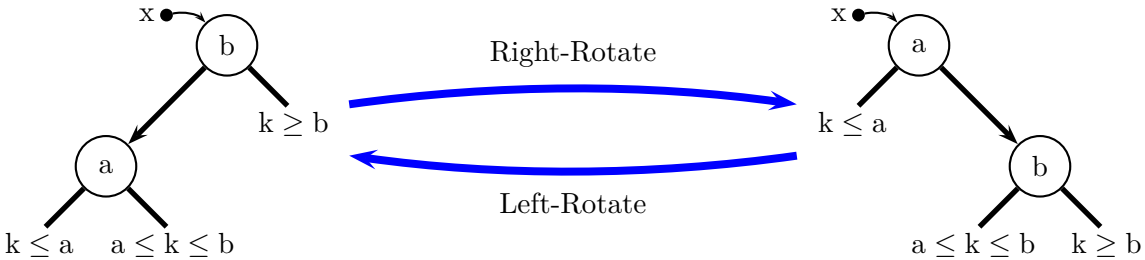


- x = Right-Rotate(x)



- x = Right-Rotate(x)
- x = Left-Rotate(x)

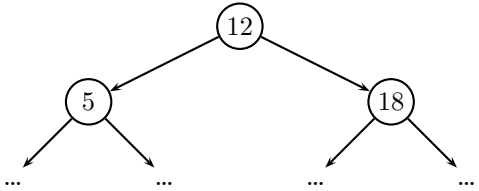
Rotation



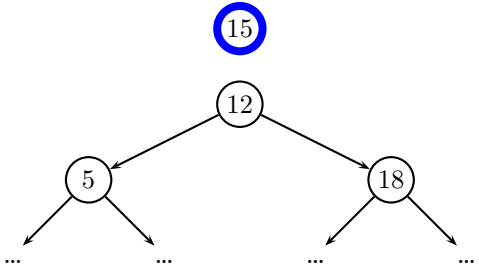

```
Right-Rotate(x)1   l = x.left
               2   x.left = l.right
               3   l.right = x
               4   return l
```

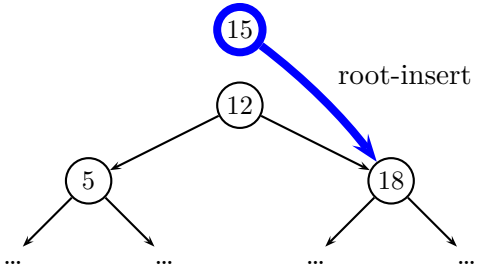
```
Left-Rotate(x)1   r = x.right
              2   x.right = r.left
              3   r.left = x
              4   return r
```

Root Insertion

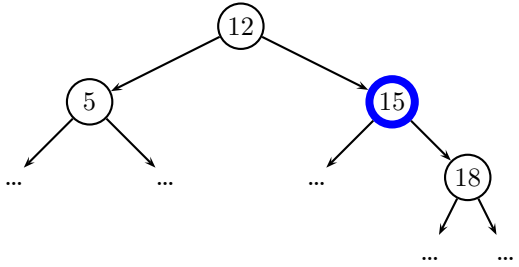


Root Insertion

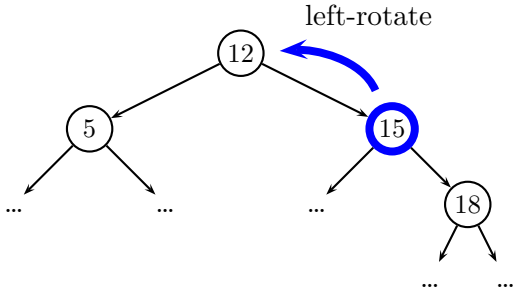




1. Recursively insert z at the root of the appropriate subtree (right)

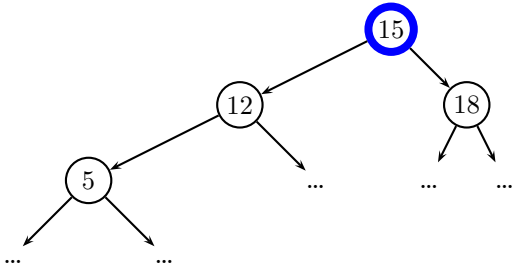


1. Recursively insert z at the root of the appropriate subtree (right)



1. Recursively insert z at the root of the appropriate subtree (right)

2. Rotate x with z (left-rotate)



1. Recursively insert z at the root of the appropriate subtree (right)

2. Rotate x with z (left-rotate)

Root Insertion (2)

```
Tree-Root-Insert(x, z)1  if x = nil
                     2       return z
                     3  if z.key < x.key
                     4       x.left = Tree-Root-Insert(x.left, z)
                     5       return Right-Rotate(x)
                     6  else x.right = Tree-Root-Insert(x.right, z)
                     7       return Left-Rotate(x)
```

Observation

- General strategies to deal with complexity in the worst case

Observation

- General strategies to deal with complexity in the worst case
  - randomization: turns any case into the average case
    - the worst case is still possible, but it is extremely improbable

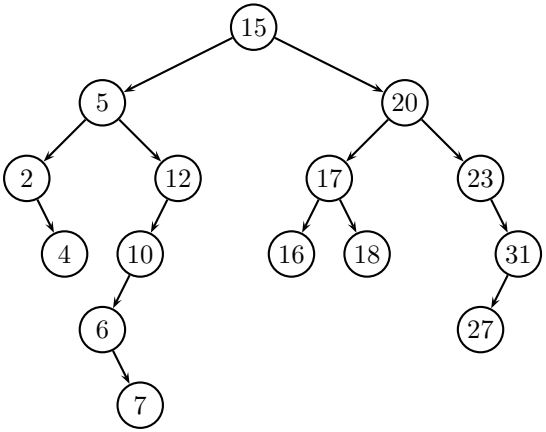
Observation

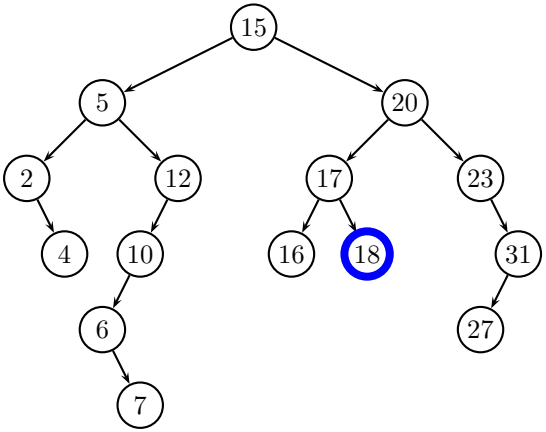
- General strategies to deal with complexity in the worst case
  - randomization: turns any case into the average case
    - ⋆ the worst case is still possible, but it is extremely improbable
  - amortized maintenance: e.g., balancing a BST or resizing a hash table
    - ⋆ relatively expensive but “amortized” operations

Observation

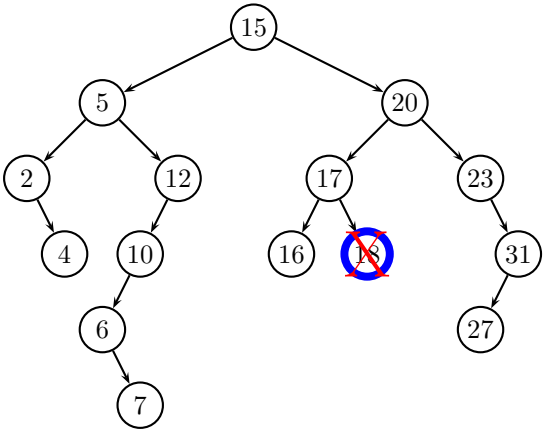
- General strategies to deal with complexity in the worst case
  - randomization: turns any case into the average case
    - ⋆ the worst case is still possible, but it is extremely improbable
  - amortized maintenance: e.g., balancing a BST or resizing a hash table
    - ⋆ relatively expensive but “amortized” operations
  - optimized data structures: a self-balanced data structure
    - ⋆ guaranteed $O(\log n)$ complexity bounds

Deletion

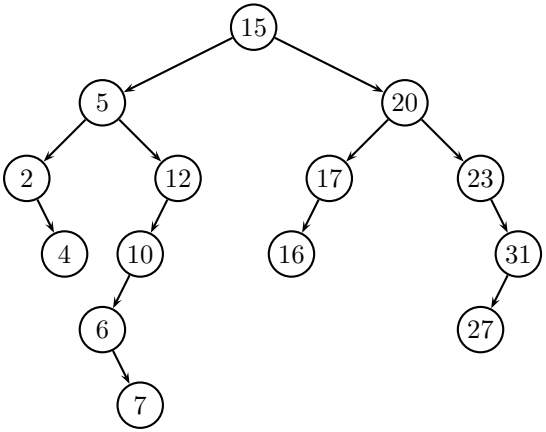

15
5
20
2
12
17
23
4
10
16
18
31
6
27
7



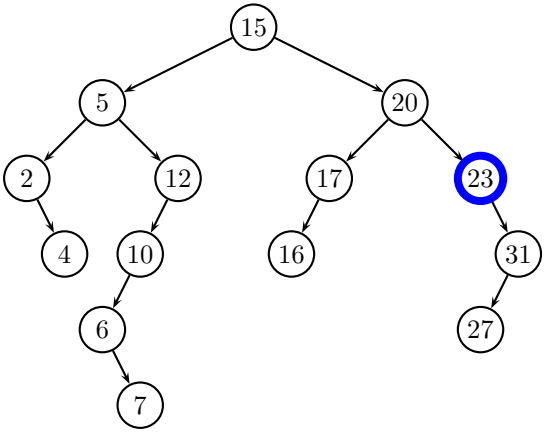
1. z has no children



1. z has no children
   - simply remove z

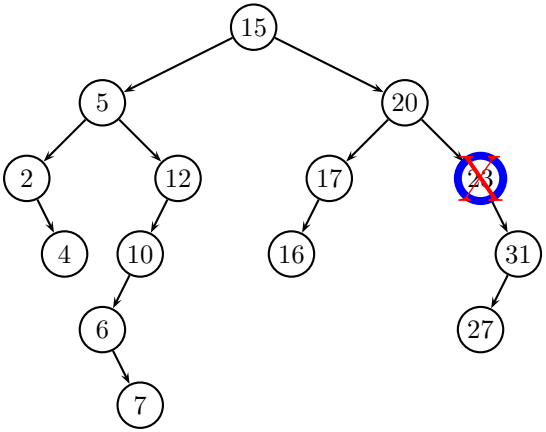


1. z has no children
   - simply remove z



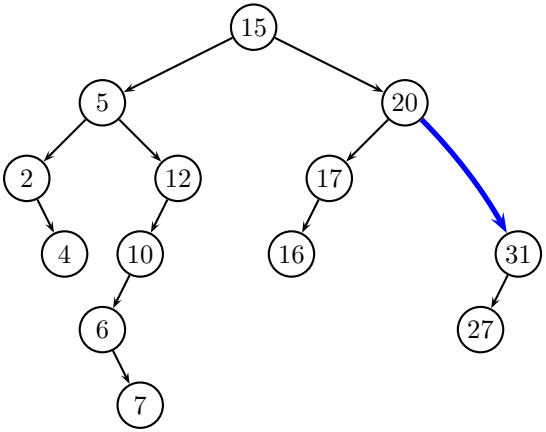
1. z has no children
   - simply remove z
2. z has one child

Deletion



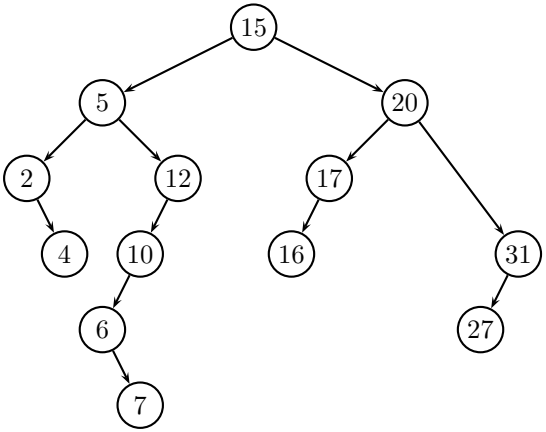
1. z has no children
   - simply remove z
2. z has one child
   - remove z

Deletion



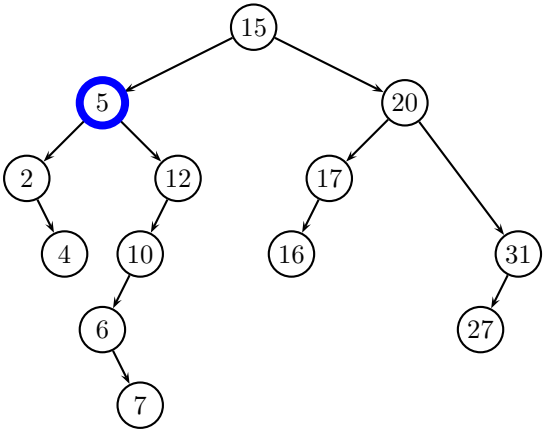
1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect z.parent to z.right

Deletion



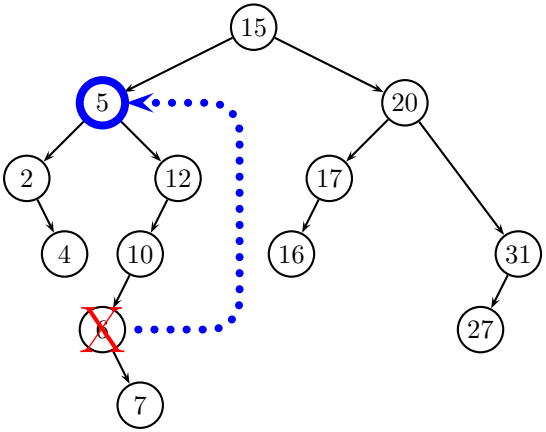
1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect z.parent to z.right

Deletion



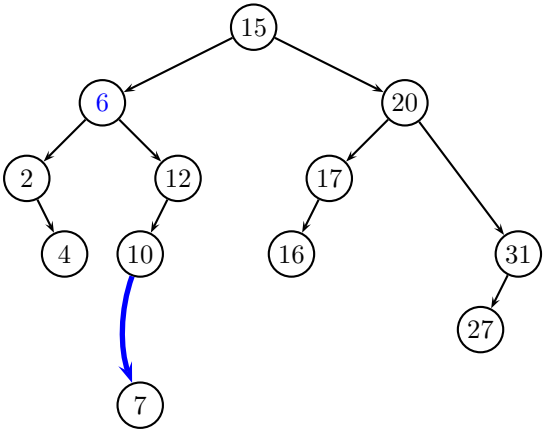
1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect z.parent to z.right
3. z has two children

Deletion



1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect z.parent to z.right
3. z has two children
   - replace z with y = Tree-Successor(z)
   - remove y (1 child!)

Deletion



1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect z.parent to z.right
3. z has two children
   - replace z with y = Tree-Successor(z)
   - remove y (1 child!)
   - connect y.parent to y.right

Deletion (2)

```
Tree-Delete(T, z)  1   if z.left = nil or z.right = nil
                   2       y = z
                   3   else y = Tree-Successor(z)
                   4   if y.left ≠ nil
                   5       x = y.left
                   6   else x = y.right
                   7   if x ≠ nil
                   8       x.parent = y.parent
                   9   if y.parent == nil
                  10       T.root = x
                  11   else if y = y.parent.left
                  12            y.parent.left = x
                  13       else y.parent.right = x
                  14   if y ≠ z
                  15       z.key = y.key
                  16       copy any other data from y into z
```

Where we are

- Want: data structure to support INSERT, DELETE, SEARCH in $O(\log n)$ time.
- Binary search trees: insert, delete, search.
- But complexity bound not met unless trees balanced.
- Can rebalance.

Next: three self-adjusting trees, finally meet the $O(\log n)$ bound: AVL trees, splay trees, red-black-trees.

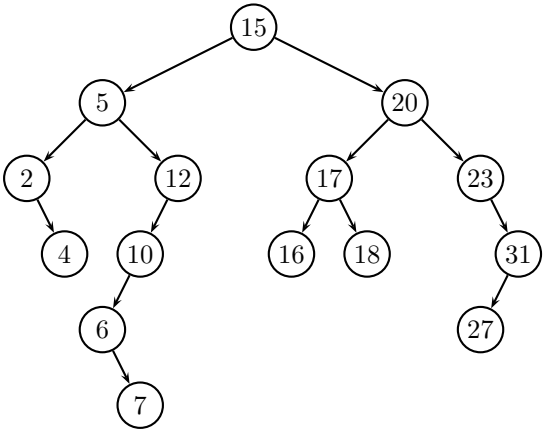
Summary on Binary Search Trees

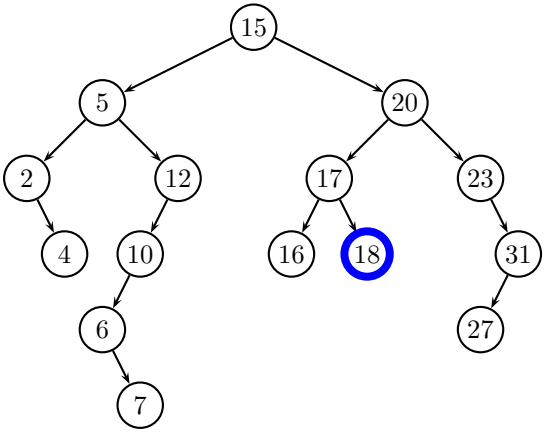
- Binary search trees
  - embody the divide-and-conquer search strategy
  - Search, Insert, Min, and Max are $O(h)$, where h is the height of the tree
  - in general, $h(n) = \Omega(\log n)$ and $h(n) = O(n)$
  - randomization can be used to make the worst-case scenario $h(n) = n$ highly unlikely

Summary on Binary Search Trees

- Binary search trees
  - embody the divide-and-conquer search strategy
  - Search, Insert, Min, and Max are $O(h)$, where h is the height of the tree
  - in general, $h(n) = \Omega(\log n)$ and $h(n) = O(n)$
  - randomization can be used to make the worst-case scenario $h(n) = n$ highly unlikely
- Problem
  - worst-case scenario is unlikely but still possible
  - simply bad cases are even more probable

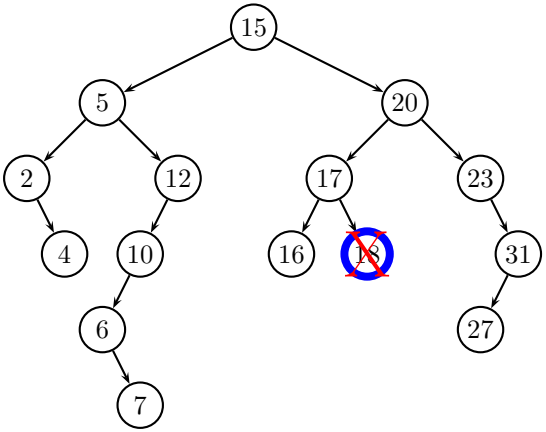
Deletion


15
5
20
2
12
17
23
4
10
16
18
31
6
27
7

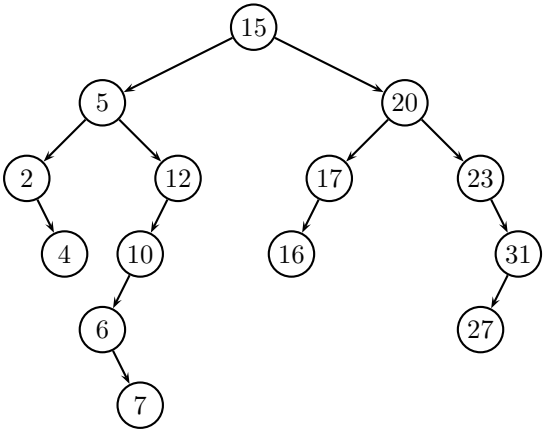


1. z has no children

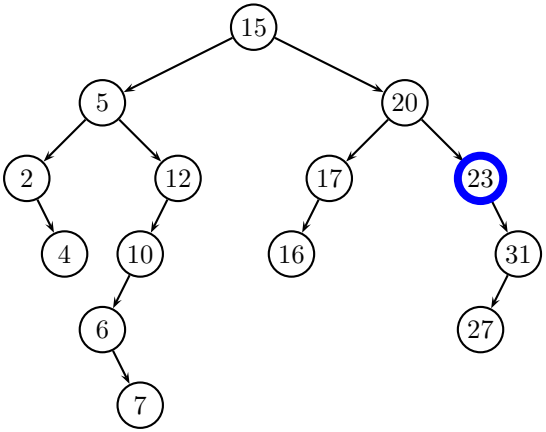
Deletion



1. z has no children
   - simply remove z

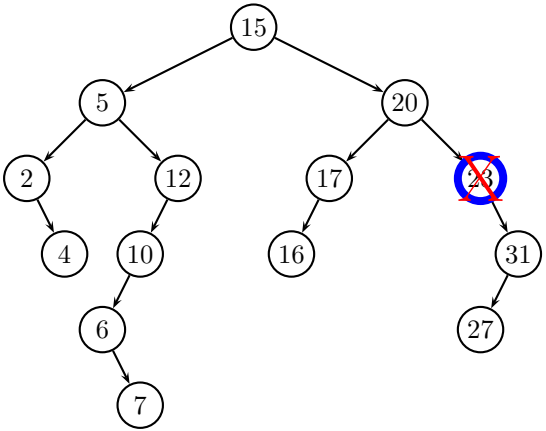


1. z has no children
   - simply remove z



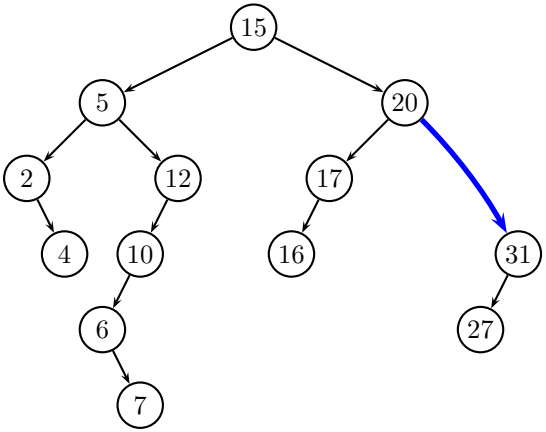
1. z has no children
   - simply remove z
2. z has one child

Deletion



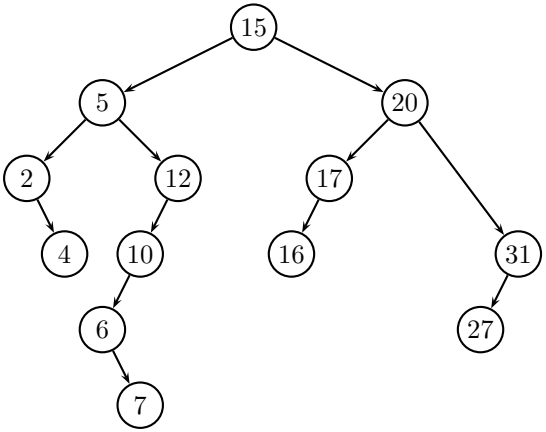
1. z has no children
   - simply remove z
2. z has one child
   - remove z

Deletion



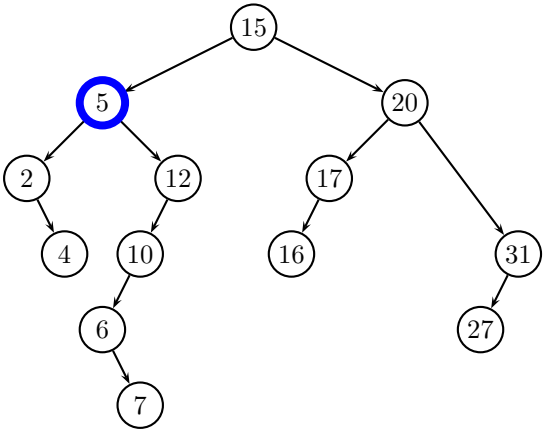
1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect z.parent to z.right

Deletion



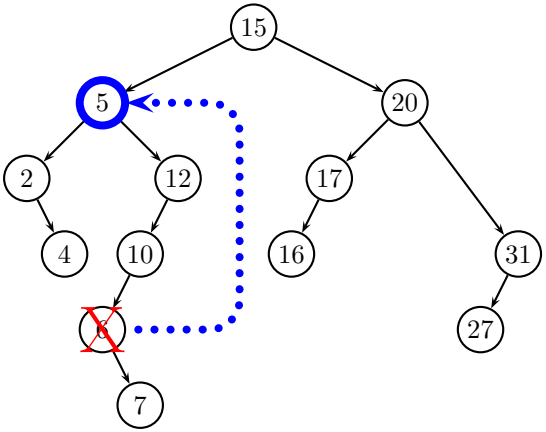
1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect z.parent to z.right

Deletion



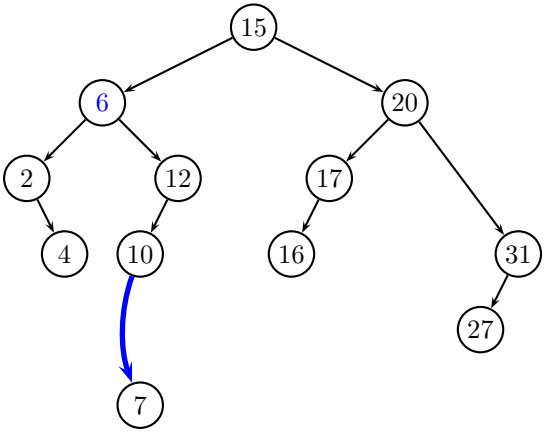
1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect z.parent to z.right
3. z has two children

Deletion



1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect z.parent to z.right
3. z has two children
   - replace z with y = Tree-Successor(z)
   - remove y (1 child!)

Deletion



1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect z.parent to z.right
3. z has two children
   - replace z with y = Tree-Successor(z)
   - remove y (1 child!)
   - connect y.parent to y.right

Deletion (2)

```
Tree-Delete(T, z)  1   if z.left = nil or z.right = nil
                   2       y = z
                   3   else y = Tree-Successor(z)
                   4   if y.left ≠ nil
                   5       x = y.left
                   6   else x = y.right
                   7   if x ≠ nil
                   8       x.parent = y.parent
                   9   if y.parent == nil
                  10       T.root = x
                  11   else if y = y.parent.left
                  12            y.parent.left = x
                  13       else y.parent.right = x
                  14   if y ≠ z
                  15       z.key = y.key
                  16       copy any other data from y into z
```

Step back

- Why study all this stuff ?
- Linked list: search linear.
- Balanced binary trees: search logarithmic.
- For frequent searches it pays.
- Advantage: as long as trees “approximately balanced”.
- But: operations (inserts/deletes) can destroy balance.

Self-balancing trees

If insertion/deletion unbalances the tree, rebalance it.

Why three of them ?

- AVL trees: more strictly balanced than R-B trees. Better for lookup intensive programs.
- For an insert intensive tasks, use a Red-Black tree.
- Simplicity of implementation: splay trees > red-black trees > AVL trees.
- Splay trees: only $O(\log n)$ amortized.
- Splay trees: suitable for cases where there are large number of nodes but only few of them are accessed frequently.
- Splay trees: more memory-efficient than AVL trees, because they do not need to store balance information in the nodes.
- AVL trees: more useful in multithreaded environments with lots of lookups, because lookups in an AVL tree can be done in parallel.
- Benchmarking: AVL trees more than 20% faster than R-B trees in "real-life" benchmarkis

Why three of them ?

- treaps
- T-trees
- tango trees

... many other ! (But this is an introduction)

You can invent your own trees ...

Tango trees: A type of binary search tree proposed by Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Pătraşcu in 2004.

They work by partitioning a BST into a set of preferred paths, which are themselves stored in auxiliary trees (so the tango tree is represented as a tree of trees)

In practice

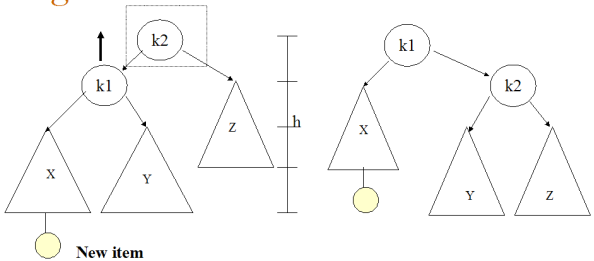
- red-black trees:
  1. Java: java.util.TreeMap , java.util.TreeSet .
  2. C++ STL: map, multimap, multiset.
  3. Linux kernel: completely fair scheduler, linux/rbtree.h
- Splay trees: typically used in the implementation of caches, memory allocators, routers, garbage collectors, data compression, etc.
- Implementations of AVL trees, RB-trees, splay trees: not standardized. STL provides only minimal set of containers.

AVL trees

- Balancedness condition #1: left and right subtrees of the root have the same height. too weak.
- Balancedness condition #2: left and right subtrees of every node have the same height. too strong.

- AVL (Adelson-Velskii and Landis) trees: binary search trees that verify the following balancedness condition: for every node v the left and right subtrees of v have height differing by at most one.
- When a tree violates rule #3 a repair is done.
- The repair is done during insertions, as soon as rule #3 is violated.
- The repair is accomplished via "single" and "double" rotations.

Single Rotation

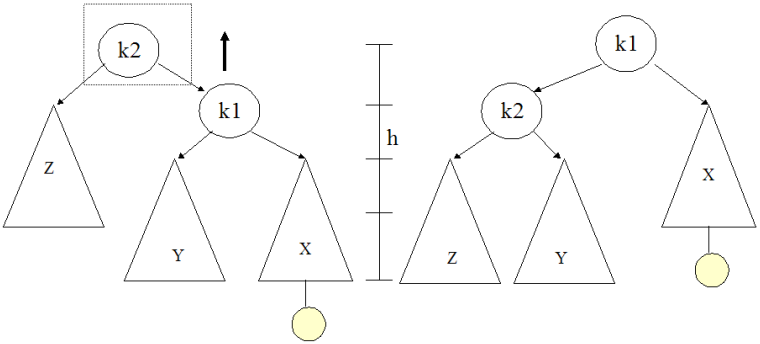


Suppose an item is added at the bottom of subtree X, thus causing an imbalance at k2. Then pull k1 up. Note that after the rotation, the height of the tree is the same as it was before the insertion.

Single rotations

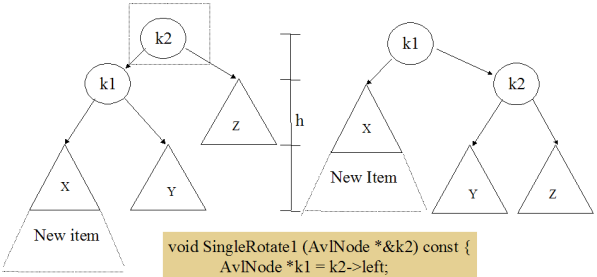
- Balance factor of a node: difference of heights between left and right subtrees.
- AVL trees: each node balance factor 0 or ± 1.
- After single rotations, the new height of the entire subtree is exactly the same as the height of the original subtree prior to the insertion of the new data item that caused X to grow.
- Thus no further updating of heights on the path to the root is needed, and consequently no further rotations are needed.

Single rotations: another example



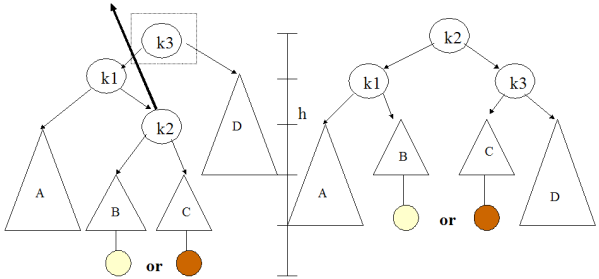
Suppose an item is added at the bottom of subtree X, thus causing an imbalance at k2. Then pull k1 up. Note that after the rotation, the height of the tree is the same as it was before the insertion.

Single rotations: C++




```
void SingleRotate1 (AvlNode *&k2) const {
        AvlNode *k1 = k2->left;
        k2->left = k1->right;
        k1->right = k2;
        k2 = k1;
}
```

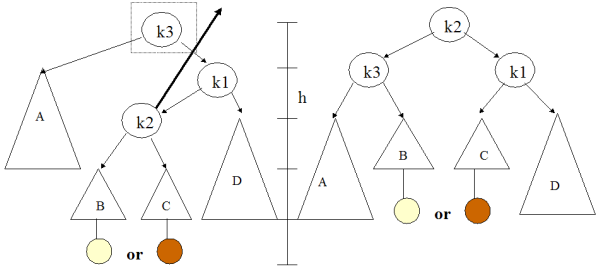
Double Rotation



Suppose an item is added below k2. This causes an imbalance at k3. Then pull k2 up. Note that after the rotation, the height of the tree is the same as it was before the insertion.

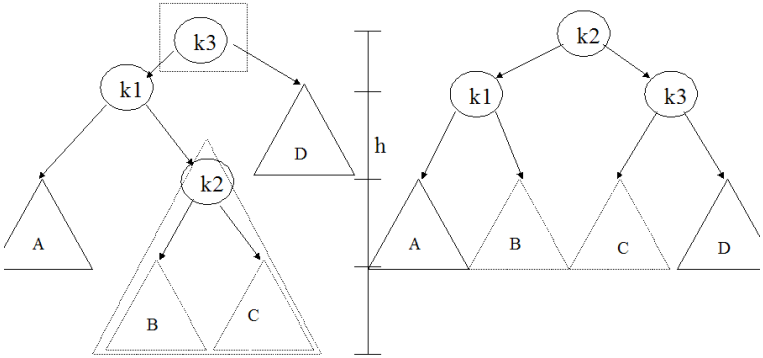
Double rotations (II)

Another Double Rotation



Suppose an item is added below k2. This causes an imbalance at k3. Then pull k2 up. Note that after the rotation, the height of the tree is the same as it was before the insertion.

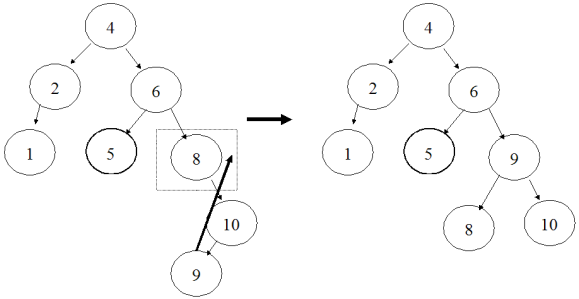
Double rotations: C++




```
void DoubleRotate1 (AvlNode *&k3) const {
        SingleRotate1( k3->left );
        SingleRotate1( k3 );
}
```

Using double rotations in practice

An Example



Imbalance at node 8 solved with double rotation.

Which rotations to use ?

- Recognizing which rotation you have to use is the hardest part.
  1. Find the imbalanced node.
  2. Go down two nodes towards the newly inserted node.
  3. If the path is straight, use single rotation.
  4. If the path zig-zags, use double rotation.

Deleting a node

- Use deleteByCopying() to delete a node. This allows reducing the problem of deleting a node with two descendants to deleting a node with at most one descendant.
- After a node has been deleted, balance factors updated from the parent of the deleted node to the root.
- For each node whose balance becomes ± 2, a single or double rotation has to be performed to restore balance of the tree.
- Deletion: at most $O(\log n)$ rotations.
- Deletion might improve balance factor of its parent.
- It may also worsen the balance factor of its grandparent.

Wrapup

- As with the single rotations, double rotations restore the height of the subtree to what it was before the insertion.
- This guarantees that all rebalancing and height updating is complete.
- AVL trees maintain balance of binary search trees while they are being created via insertions of data.
- An alternative approach is to have trees that readjust themselves when data is accessed, making often accessed data items move to the top of the tree (splay trees).

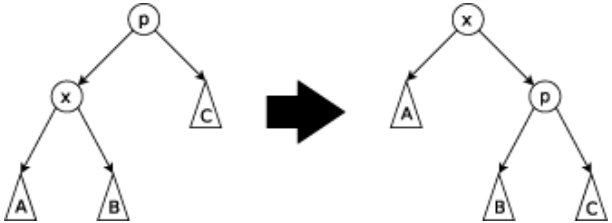
Splay trees

- Invented by Sleator and Tarjan (1985).
- to splay $\sim$ to spread out.
- Self-balancing binary trees, simpler to implement than AVL, red-black trees.
- Additional property: recently accessed elements quick to access.
- Insertion, lookup, removal: $O(\log n)$ amortized time.
- That roughly means that the average price per operation in a long sequence of operations is $O(\log n)$.
- Fundamental operation: splaying. Rearranging the tree such that certain elemen brought at the top of the tree.

Splaying

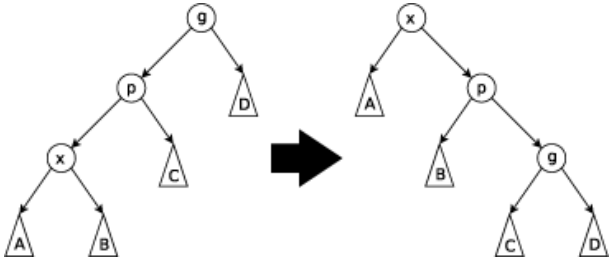
- When a node x is accessed, a splaying operation performed to bring it to the top.
- Composed of a sequence of splaying steps.
- Each splaying step brings x closer to the root.
- Steps depend on:
- Whether z is left or right child of its parent p.
- Whether p is root or not, and
- Whether p is left or right child of its parent g.
- Three types of splaying steps.

First case: p is the root



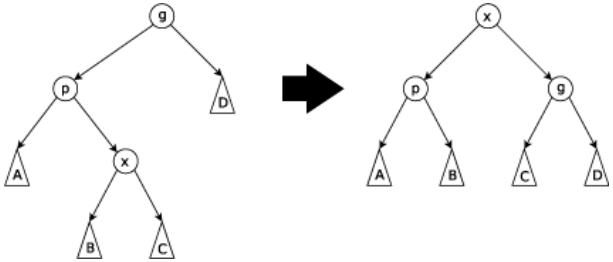
"zig": basically rotation.

Second case: p not the root, x, p both left or both right children



"Zigzig"

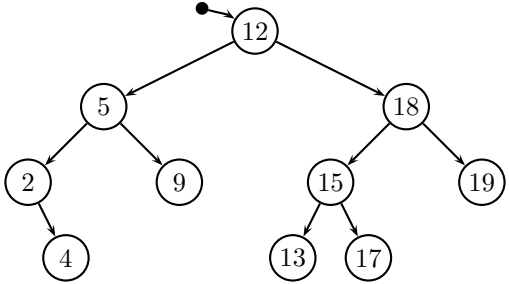
Third case:p, x alternate sides

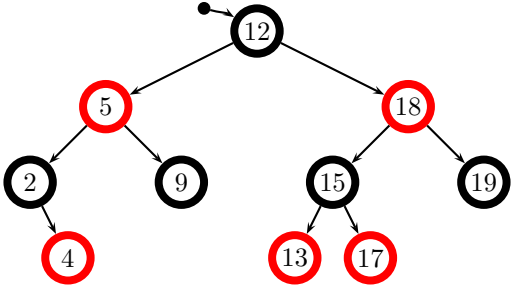


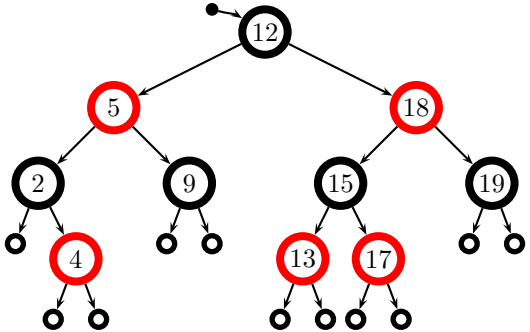
“Zigzag”

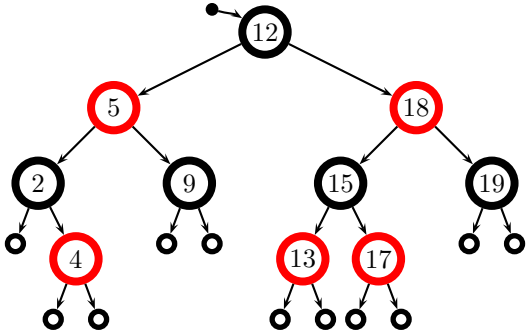
Splaying operations

- First case: rotation.
- All cases: actually two mirror-image cases (only one shown in picture).
- Advantages: more accessed nodes closer to root. Useful for implementing caches, garbage collection.
- Disadvantages: random access worse than for other balanced BST.
- Particularly bad: access elements in sorted order.

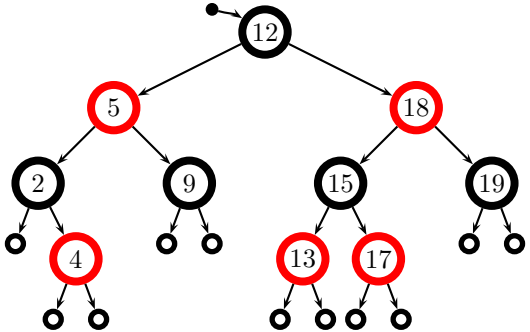

12
5
18
2
9
15
19
4
13
17


12
5
18
2
9
15
19
4
13
17

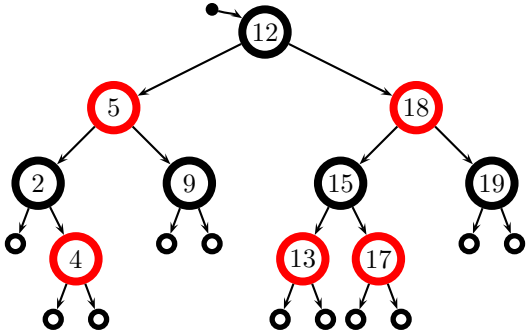

12
5
18
2
9
15
19
4
13
17



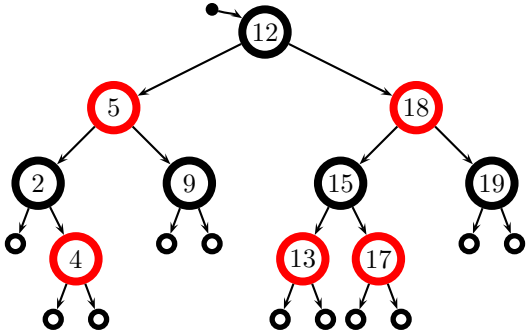
- Red-black-tree property



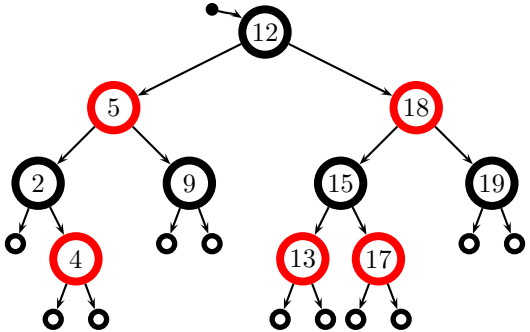
- Red-black-tree property
  1. every node is either red or black



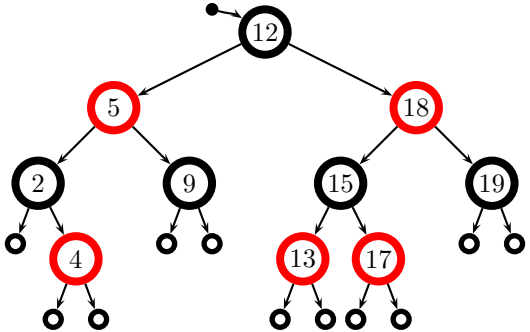
- Red-black-tree property
  1. every node is either red or black
  2. the root is black



- Red-black-tree property
  1. every node is either red or black
  2. the root is black
  3. every (NIL) leaf is black



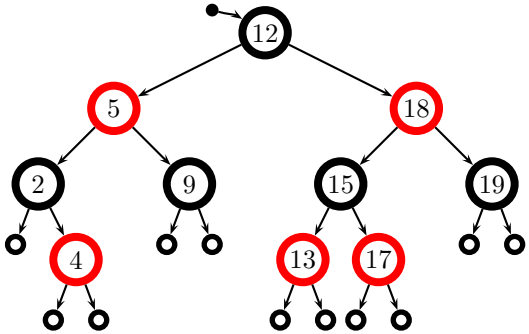
- Red-black-tree property
  1. every node is either red or black
  2. the root is black
  3. every (NIL) leaf is black
  4. if a node is red, then both its children are black



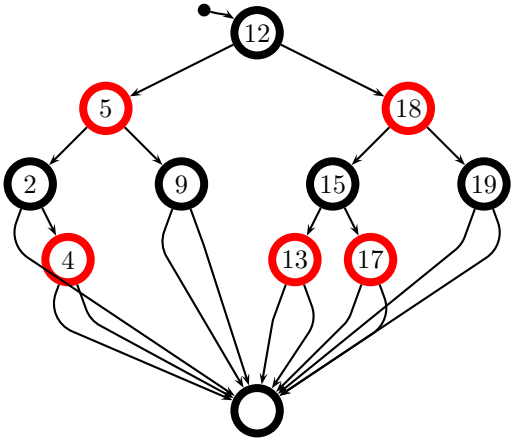
- Red-black-tree property
  1. every node is either red or black
  2. the root is black
  3. every (NIL) leaf is black
  4. if a node is red, then both its children are black
  5. for every node x, each path from x to its descendant leaves has the same number of black nodes bh(x) (the black-height of x)

Red-Black Trees (2)

- Implementation



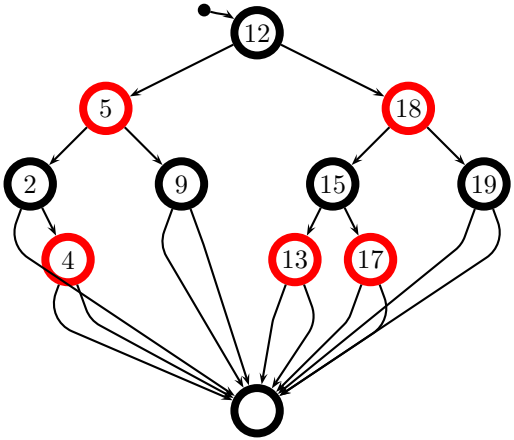
- Implementation



  - we use a common “sentinel” node to represent leaf nodes

Red-Black Trees (2)

- Implementation



  - we use a common “sentinel” node to represent leaf nodes
  - the sentinel is also the parent of the root node

Red-Black Trees (3)

- Implementation
  - T represents the tree, which consists of a set of nodes

Red-Black Trees (3)

- Implementation
  - T represents the tree, which consists of a set of nodes
  - T.root is the root node of tree T

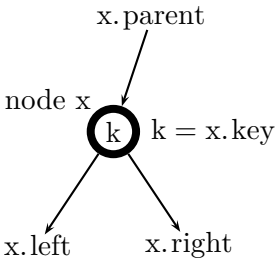
Red-Black Trees (3)

- Implementation
  - T represents the tree, which consists of a set of nodes
  - T.root is the root node of tree T
  - T.nil is the “sentinel” node of tree T

- Implementation
  - T represents the tree, which consists of a set of nodes
  - T.root is the root node of tree T
  - T.nil is the "sentinel" node of tree T

Nodes

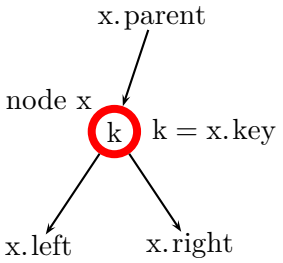
- x.parent is the parent of node x
- x.key is the key stored in node x
- x.left is the left child of node x
- x.right is the right child of node x



- Implementation
  - T represents the tree, which consists of a set of nodes
  - T.root is the root node of tree T
  - T.nil is the "sentinel" node of tree T

Nodes

- x.parent is the parent of node x
- x.key is the key stored in node x
- x.left is the left child of node x
- x.right is the right child of node x
- x.color $\in \{\text{red}, \text{black}\}$ is the color of node x



Height of a Red-Black Tree

Height of a Red-Black Tree

Lemma: the height $h(x)$ of a red-black tree with $n = \text{size}(x)$ internal nodes is at most $2\log(n+1)$.

Height of a Red-Black Tree

Lemma: the height $h(x)$ of a red-black tree with $n = size(x)$ internal nodes is at most $2\log(n+1)$.

Proof:

1. prove that $\forall x : size(x) \geq 2^{bh(x)} - 1$ by induction:
   1. base case: x is a leaf, so $size(x) = 0$ and $bh(x) = 0$

Height of a Red-Black Tree

Lemma: the height $h(x)$ of a red-black tree with $n = \text{size}(x)$ internal nodes is at most $2\log(n+1)$.
Proof:

1. prove that $\forall x : \text{size}(x) \geq 2^{\text{bh}(x)} - 1$ by induction:
   1. base case: x is a leaf, so $\text{size}(x) = 0$ and $\text{bh}(x) = 0$
   2. induction step: consider y_1, y_2, and x such that $y_1.\text{parent} = y_2.\text{parent} = x$, and assume (induction) that $\text{size}(y_1) \geq 2^{\text{bh}(y_1)} - 1$ and $\text{size}(y_2) \geq 2^{\text{bh}(y_2)} - 1$;
      prove that $\text{size}(x) \geq 2^{\text{bh}(x)} - 1$

Height of a Red-Black Tree

Lemma: the height $h(x)$ of a red-black tree with $n = size(x)$ internal nodes is at most $2\log(n+1)$.
Proof:

1. prove that $\forall x : size(x) \geq 2^{bh(x)} - 1$ by induction:
   1. base case: x is a leaf, so $size(x) = 0$ and $bh(x) = 0$
   2. induction step: consider y_1, y_2, and x such that $y_1.parent = y_2.parent = x$, and assume (induction) that $size(y_1) \geq 2^{bh(y_1)} - 1$ and $size(y_2) \geq 2^{bh(y_2)} - 1$;
      prove that $size(x) \geq 2^{bh(x)} - 1$
      proof:
      $size(x) = size(y_1) + size(y_2) + 1 \geq (2^{bh(y_1)} - 1) + (2^{bh(y_2)} - 1) + 1$

Height of a Red-Black Tree

Lemma: the height $h(x)$ of a red-black tree with $n = \text{size}(x)$ internal nodes is at most $2\log(n+1)$.
Proof:

1. prove that $\forall x : \text{size}(x) \geq 2^{\text{bh}(x)} - 1$ by induction:
   1. base case: x is a leaf, so $\text{size}(x) = 0$ and $\text{bh}(x) = 0$
   2. induction step: consider y_1, y_2, and x such that $y_1.\text{parent} = y_2.\text{parent} = x$, and assume (induction) that $\text{size}(y_1) \geq 2^{\text{bh}(y_1)} - 1$ and $\text{size}(y_2) \geq 2^{\text{bh}(y_2)} - 1$;
      prove that $\text{size}(x) \geq 2^{\text{bh}(x)} - 1$
      proof:
      $\text{size}(x) = \text{size}(y_1) + \text{size}(y_2) + 1 \geq (2^{\text{bh}(y_1)} - 1) + (2^{\text{bh}(y_2)} - 1) + 1$
      since

$$\text{bh}(x) = \begin{cases} \text{bh}(y) & \text{if color}(y) = \text{red} \\ \text{bh}(y) + 1 & \text{if color}(y) = \text{black} \end{cases}$$

Height of a Red-Black Tree

Lemma: the height $h(x)$ of a red-black tree with $n = \text{size}(x)$ internal nodes is at most $2\log(n+1)$.
Proof:

1. prove that $\forall x : \text{size}(x) \geq 2^{\text{bh}(x)} - 1$ by induction:
   1. base case: x is a leaf, so $\text{size}(x) = 0$ and $\text{bh}(x) = 0$
   2. induction step: consider y_1, y_2, and x such that $y_1.\text{parent} = y_2.\text{parent} = x$, and assume (induction) that $\text{size}(y_1) \geq 2^{\text{bh}(y_1)} - 1$ and $\text{size}(y_2) \geq 2^{\text{bh}(y_2)} - 1$;
      prove that $\text{size}(x) \geq 2^{\text{bh}(x)} - 1$
      proof:
      $\text{size}(x) = \text{size}(y_1) + \text{size}(y_2) + 1 \geq (2^{\text{bh}(y_1)} - 1) + (2^{\text{bh}(y_2)} - 1) + 1$
      since

$$\text{bh}(x) = \begin{cases} \text{bh}(y) & \text{if color}(y) = \text{red} \\ \text{bh}(y) + 1 & \text{if color}(y) = \text{black} \end{cases}$$

      $\text{size}(x) \geq (2^{\text{bh}(x)-1} - 1) + (2^{\text{bh}(x)-1} - 1) + 1$

Lemma: the height $h(x)$ of a red-black tree with $n = \text{size}(x)$ internal nodes is at most $2\log(n+1)$.

Proof:

1. prove that $\forall x : \text{size}(x) \geq 2^{\text{bh}(x)} - 1$ by induction:
   1. base case: x is a leaf, so $\text{size}(x) = 0$ and $\text{bh}(x) = 0$
   2. induction step: consider y_1, y_2, and x such that $y_1.\text{parent} = y_2.\text{parent} = x$, and assume (induction) that $\text{size}(y_1) \geq 2^{\text{bh}(y_1)} - 1$ and $\text{size}(y_2) \geq 2^{\text{bh}(y_2)} - 1$;
      prove that $\text{size}(x) \geq 2^{\text{bh}(x)} - 1$
      proof:
      $\text{size}(x) = \text{size}(y_1) + \text{size}(y_2) + 1 \geq (2^{\text{bh}(y_1)} - 1) + (2^{\text{bh}(y_2)} - 1) + 1$
      since

$$\text{bh}(x) = \begin{cases} \text{bh}(y) & \text{if color}(y) = \text{red} \\ \text{bh}(y) + 1 & \text{if color}(y) = \text{black} \end{cases}$$

      $\text{size}(x) \geq (2^{\text{bh}(x)-1} - 1) + (2^{\text{bh}(x)-1} - 1) + 1 = 2^{\text{bh}(x)} - 1$

Height of a Red-Black Tree (2)

1. $\text{size}(x) \geq 2^{\text{bh}(x)} - 1$

Height of a Red-Black Tree (2)

1. $\text{size}(x) \geq 2^{\text{bh}(x)} - 1$
2. Since every red node has black children, in every path from x to a leaf node, at least half the nodes are black

Height of a Red-Black Tree (2)

1. $\text{size}(x) \geq 2^{\text{bh}(x)} - 1$
2. Since every red node has black children, in every path from x to a leaf node, at least half the nodes are black, thus $\text{bh}(x) \geq h(x)/2$

Height of a Red-Black Tree (2)

1. $\text{size}(x) \geq 2^{\text{bh}(x)} - 1$
2. Since every red node has black children, in every path from x to a leaf node, at least half the nodes are black, thus $\text{bh}(x) \geq \text{h}(x)/2$
3. From steps 1 and 2, $n = \text{size}(x) \geq 2^{\text{h}(x)/2} - 1$

Height of a Red-Black Tree (2)

1. $\text{size}(x) \geq 2^{\text{bh}(x)} - 1$
2. Since every red node has black children, in every path from x to a leaf node, at least half the nodes are black, thus $\text{bh}(x) \geq h(x)/2$
3. From steps 1 and 2, $n = \text{size}(x) \geq 2^{h(x)/2} - 1$, therefore

$$h \leq 2 \log(n + 1)$$

Height of a Red-Black Tree (2)

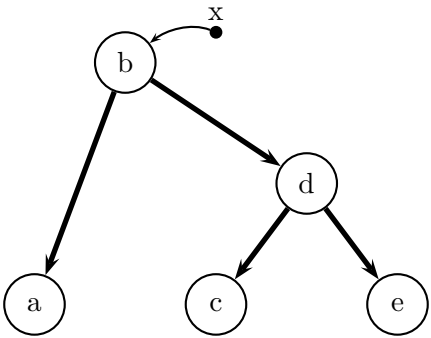
1. $\text{size}(x) \geq 2^{\text{bh}(x)} - 1$
2. Since every red node has black children, in every path from x to a leaf node, at least half the nodes are black, thus $\text{bh}(x) \geq h(x)/2$
3. From steps 1 and 2, $n = \text{size}(x) \geq 2^{h(x)/2} - 1$, therefore

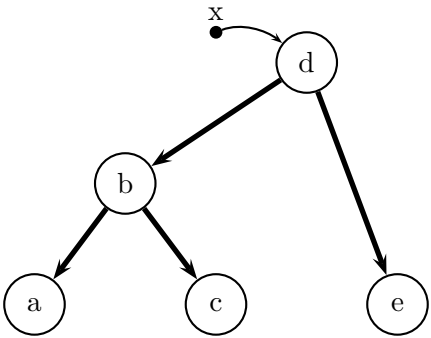
$$h \leq 2 \log(n + 1)$$

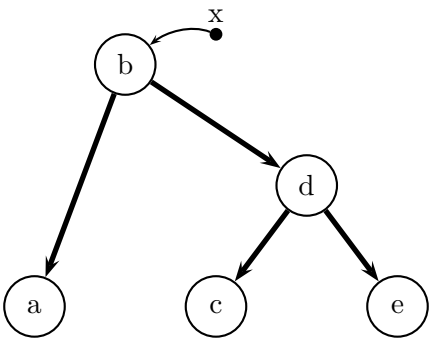
- A red-black tree works as a binary search tree for search, etc.
- So, the complexity of those operations is $T(n) = O(h)$, that is

$$\boxed{T(n) = O(\log n)}$$

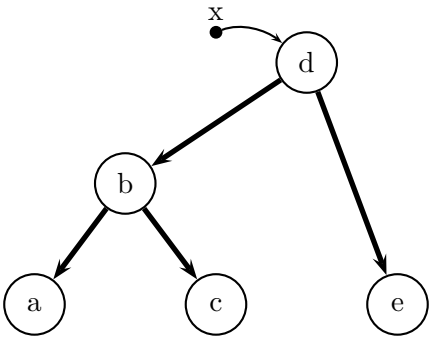
  - which is also the worst-case complexity


x
b
d
a
c
e


x
d
b
a
c
e



- x = Right-Rotate(x)



- x = Right-Rotate(x)
- x = Left-Rotate(x)

- RB-Insert(T, z) works as in a binary search tree

Red-Black Insertion

- RB-Insert(T, z) works as in a binary search tree
- Except that it must preserve the red-black-tree property

Red-Black Insertion

- RB-Insert(T, z) works as in a binary search tree
- Except that it must preserve the red-black-tree property
  1. every node is either red or black
  2. the root is black
  3. every (NIL) leaf is black
  4. if a node is red, then both its children are black
  5. for every node x, each path from x to its descendant leaves has the same number of black nodes bh(x) (the black-height of x)

Red-Black Insertion

- RB-Insert(T, z) works as in a binary search tree
- Except that it must preserve the red-black-tree property
  1. every node is either red or black
  2. the root is black
  3. every (NIL) leaf is black
  4. if a node is red, then both its children are black
  5. for every node x, each path from x to its descendant leaves has the same number of black nodes bh(x) (the black-height of x)
- General strategy

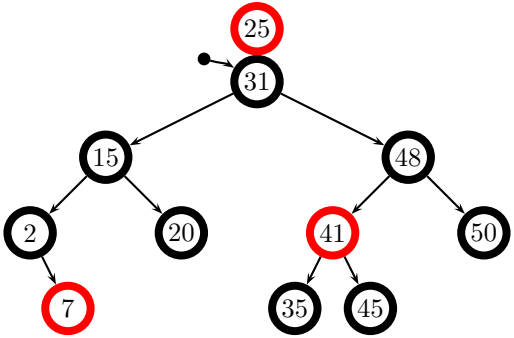
Red-Black Insertion

- RB-Insert(T, z) works as in a binary search tree
- Except that it must preserve the red-black-tree property
  1. every node is either red or black
  2. the root is black
  3. every (NIL) leaf is black
  4. if a node is red, then both its children are black
  5. for every node x, each path from x to its descendant leaves has the same number of black nodes bh(x) (the black-height of x)
- General strategy
  1. insert z as in a binary search tree
  2. color z red so as to preserve property 5
  3. fix the tree to correct possible violations of property 4

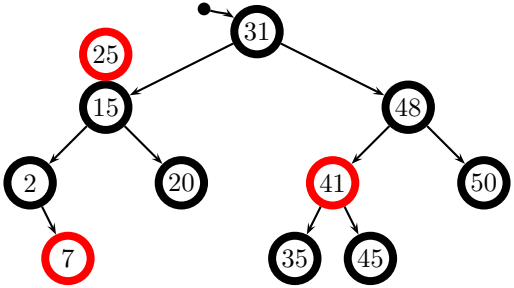
RB-Insert

```
RB-Insert(T, z)  1   y = T.nil
                 2   x = T.root
                 3   while x ≠ T.nil
                 4       y = x
                 5       if z.key < x.key
                 6           x = x.left
                 7       else x = x.right
                 8   z.parent = y
                 9   if y == T.nil
                10       T.root = z
                11   else if z.key < y.key
                12           y.left = z
                13       else y.right = z
                14   z.left = z.right = T.nil
                15   z.color = red
                16   RB-Insert-Fixup(T, z)
```

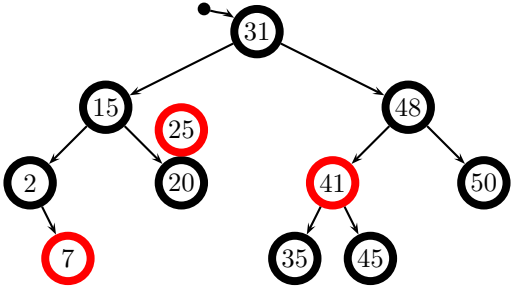
Red-Black Insertion (2)



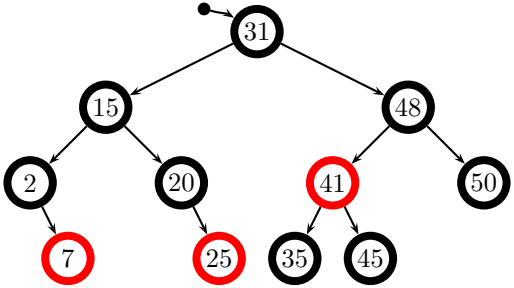
Red-Black Insertion (2)



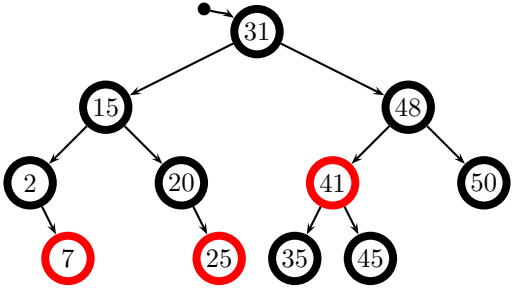
Red-Black Insertion (2)



Red-Black Insertion (2)



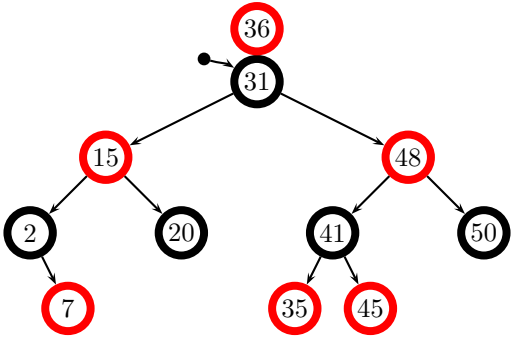
Red-Black Insertion (2)



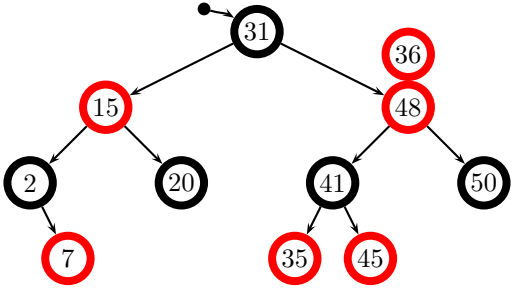
- z's father is black, so no fixup needed

Red-Black Insertion (3)

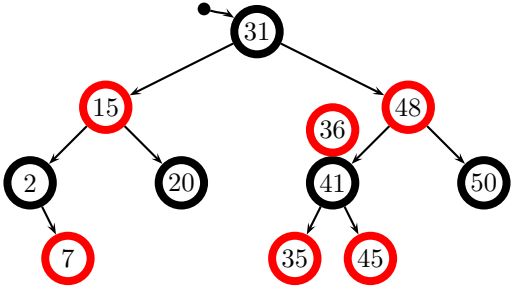
Red-Black Insertion (3)



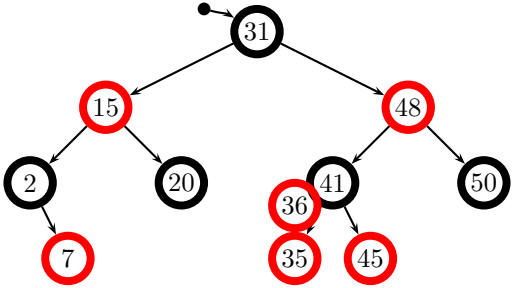
Red-Black Insertion (3)



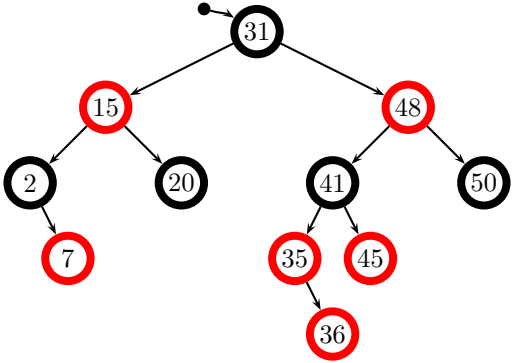
Red-Black Insertion (3)



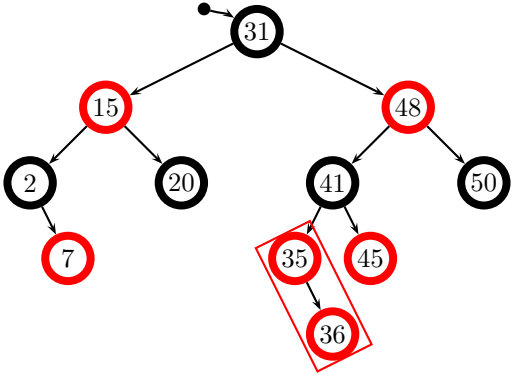
Red-Black Insertion (3)



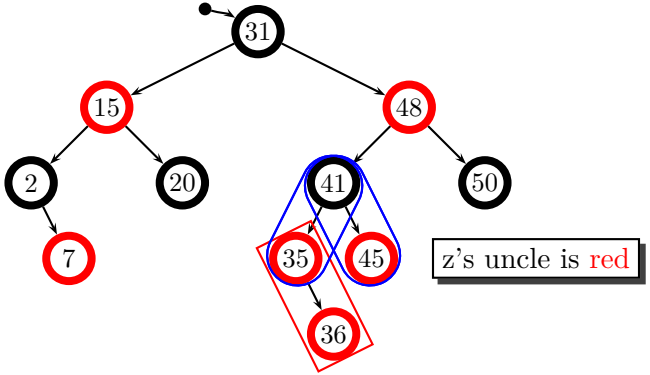
Red-Black Insertion (3)



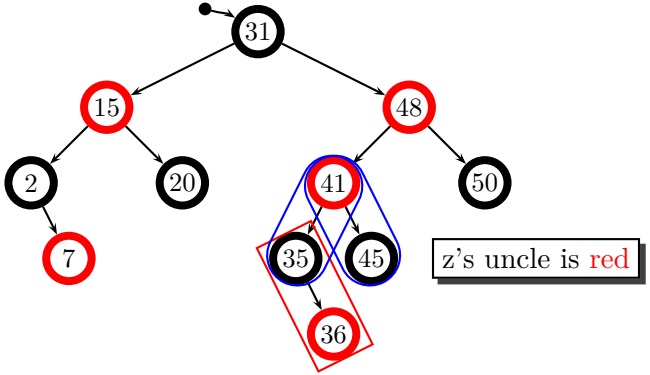
Red-Black Insertion (3)



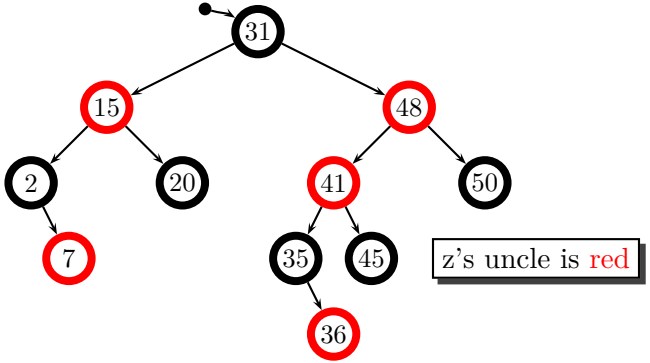
Red-Black Insertion (3)



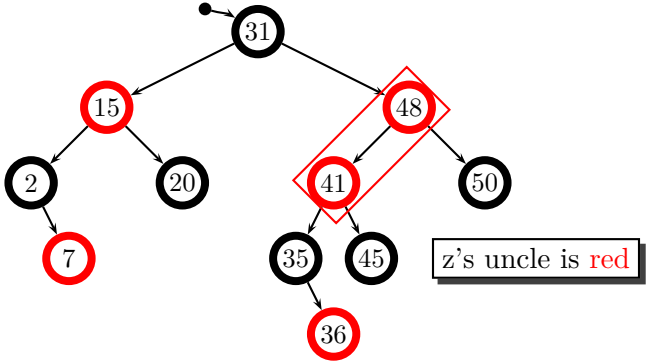
Red-Black Insertion (3)



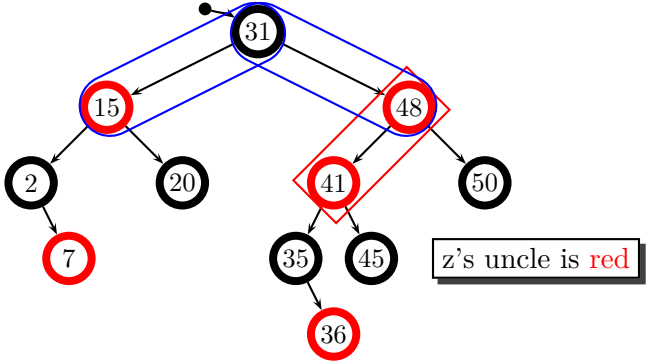
Red-Black Insertion (3)



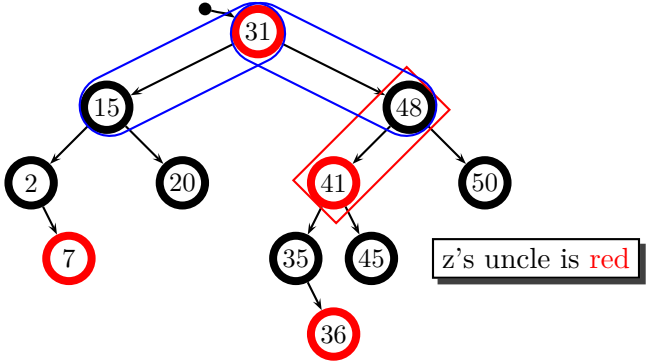
Red-Black Insertion (3)



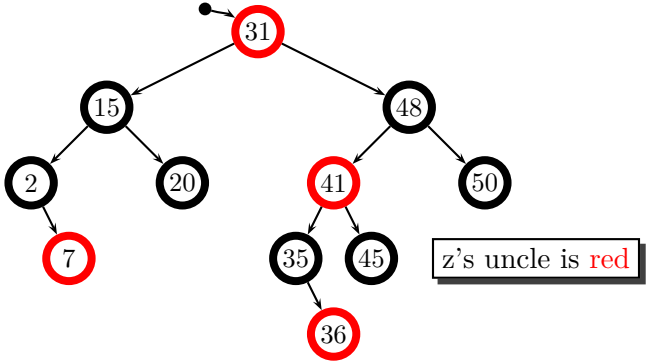
Red-Black Insertion (3)



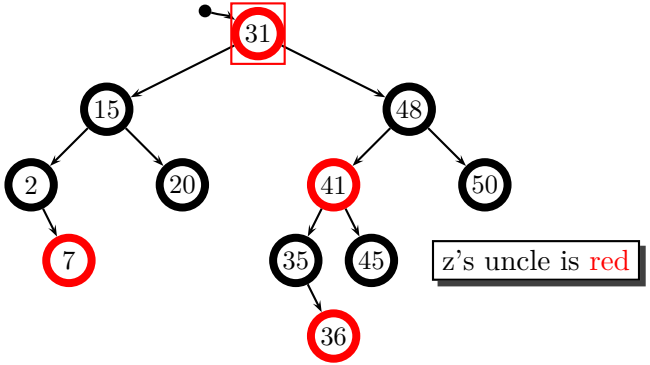
Red-Black Insertion (3)



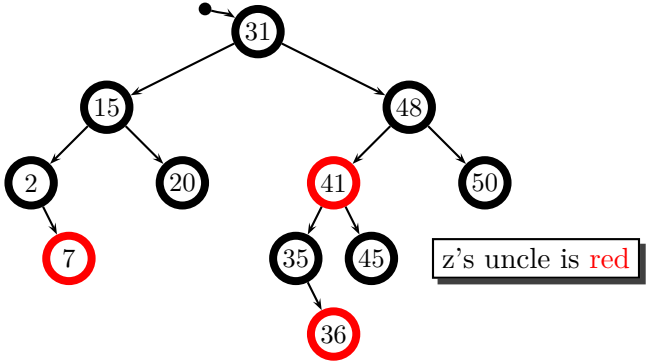
Red-Black Insertion (3)



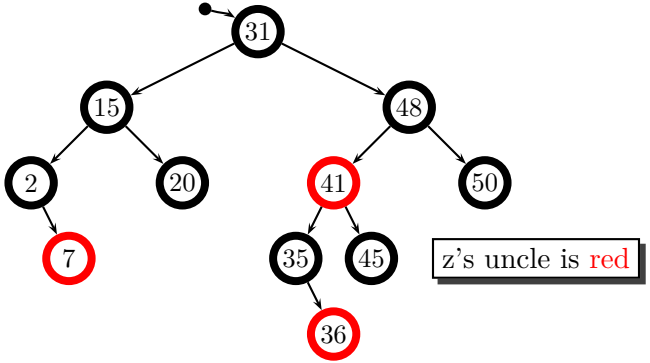
Red-Black Insertion (3)



Red-Black Insertion (3)

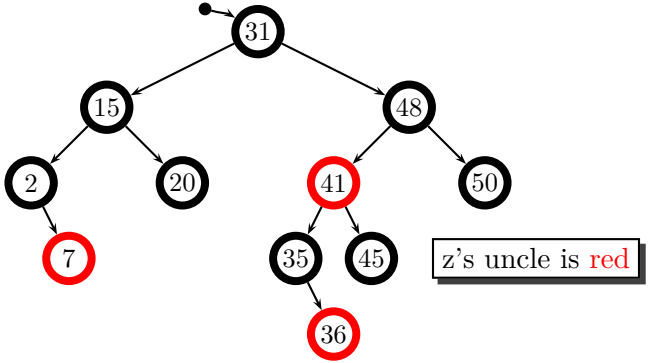


Red-Black Insertion (3)



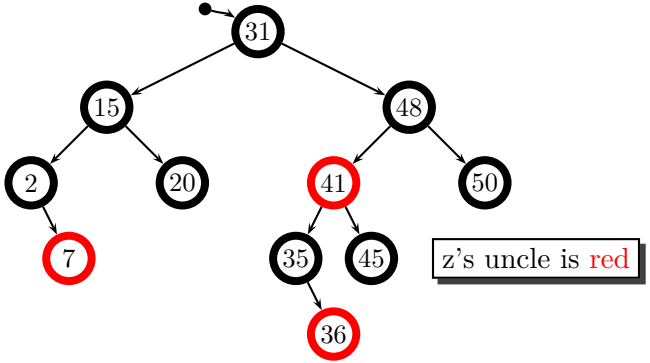
- A black node can become red and transfer its black color to its two children

Red-Black Insertion (3)



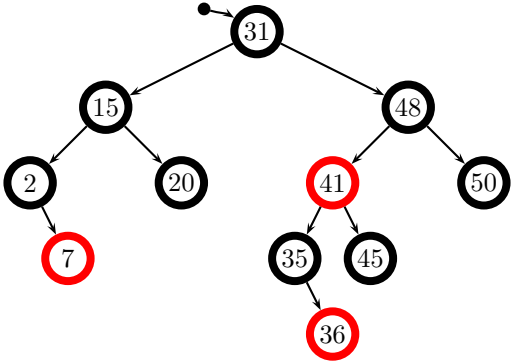
- A black node can become red and transfer its black color to its two children
- This may cause other red–red conflicts, so we iterate…

Red-Black Insertion (3)

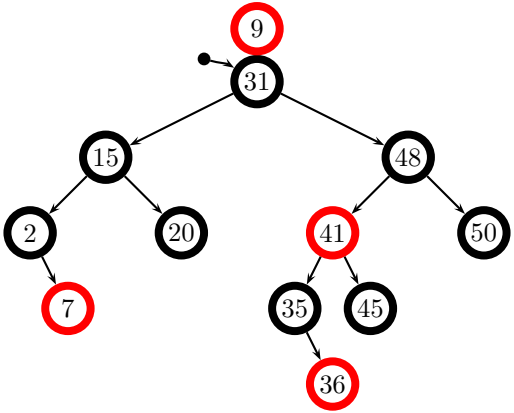


- A black node can become red and transfer its black color to its two children
- This may cause other red–red conflicts, so we iterate…
- The root can change to black without causing conflicts

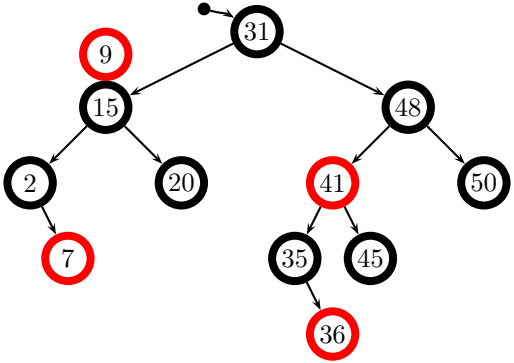
Red-Black Insertion (4)



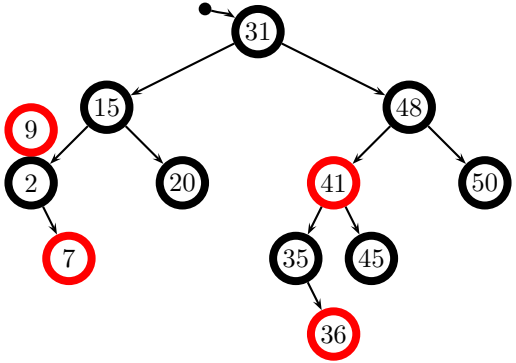
Red-Black Insertion (4)



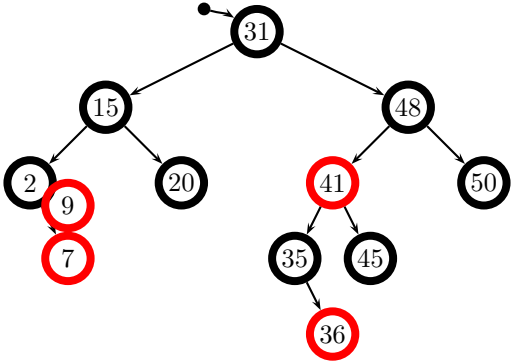
Red-Black Insertion (4)



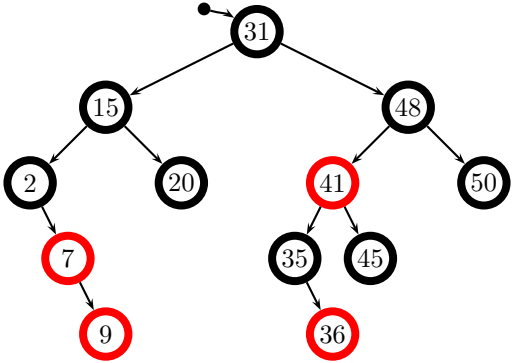
Red-Black Insertion (4)



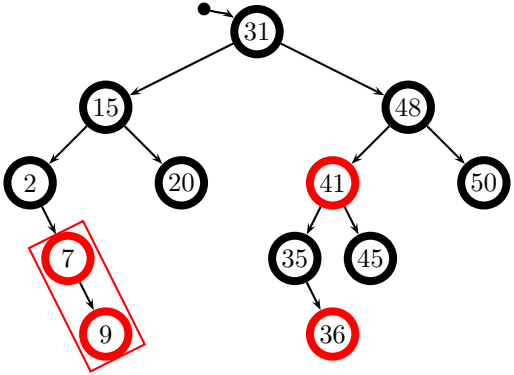
Red-Black Insertion (4)



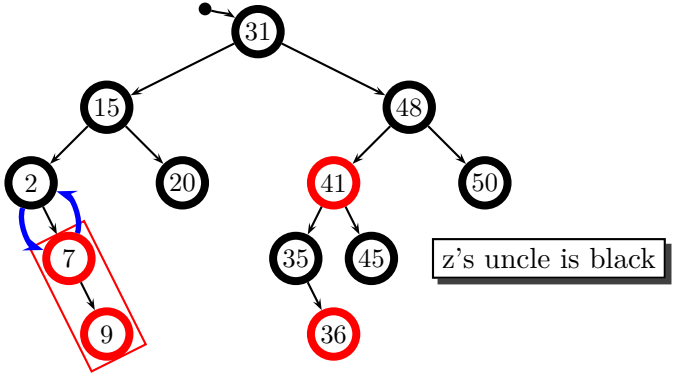
Red-Black Insertion (4)



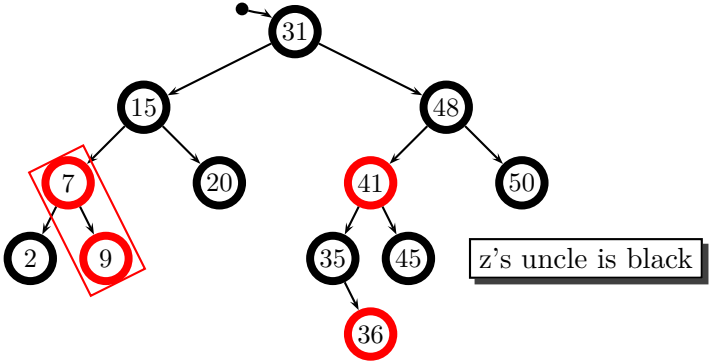
Red-Black Insertion (4)



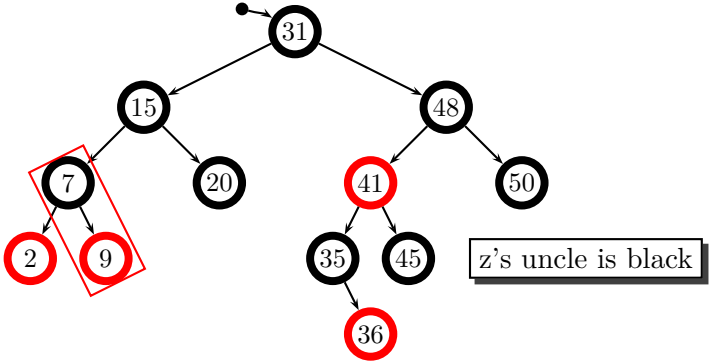
Red-Black Insertion (4)



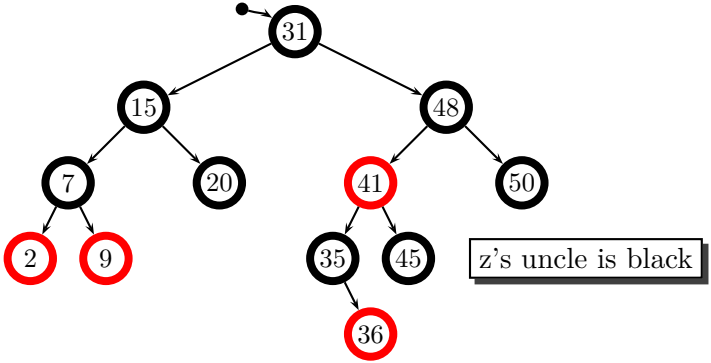
Red-Black Insertion (4)



Red-Black Insertion (4)

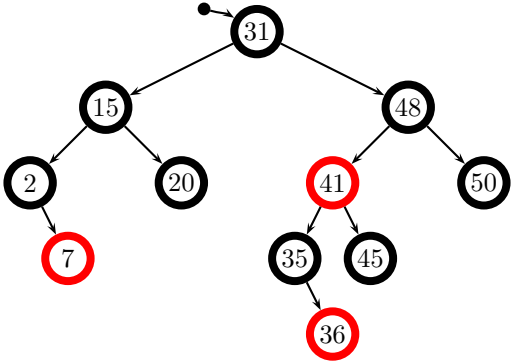


Red-Black Insertion (4)

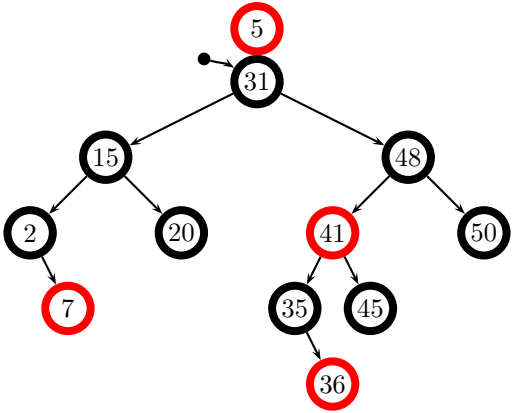


- An in-line red–red conflicts can be resolved with a rotation plus a color switch

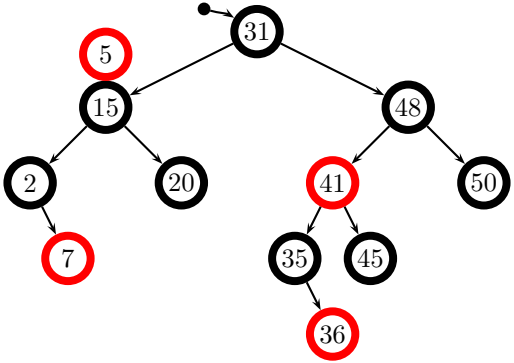
Red-Black Insertion (5)



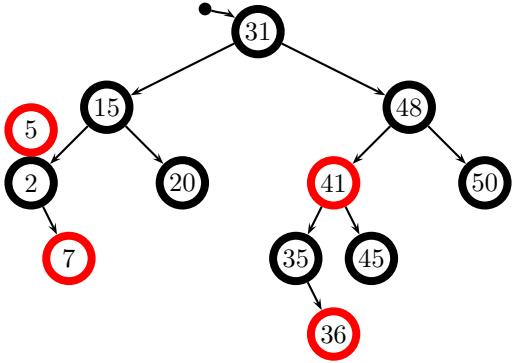
Red-Black Insertion (5)



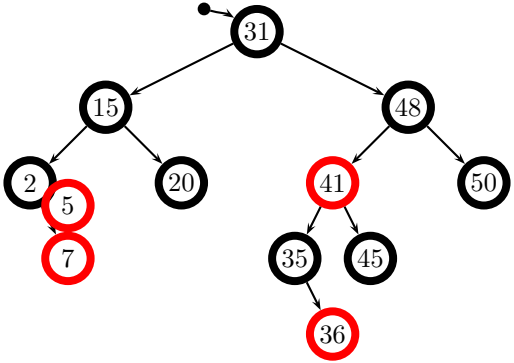
Red-Black Insertion (5)



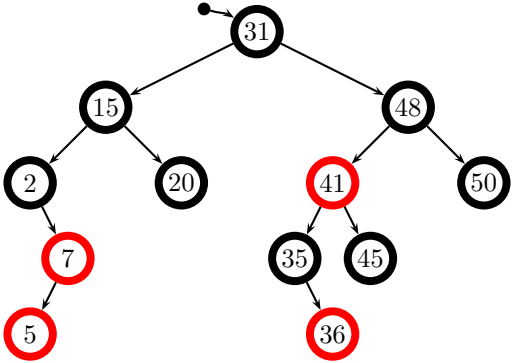
Red-Black Insertion (5)



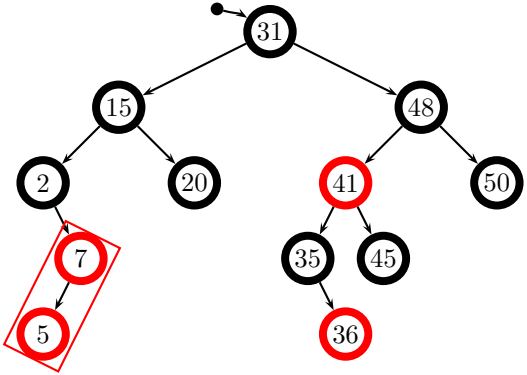
Red-Black Insertion (5)



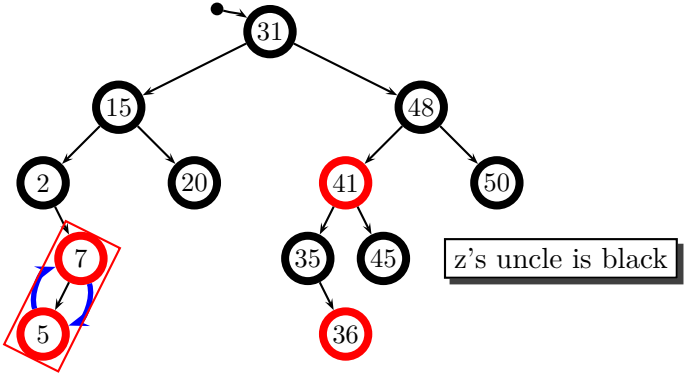
Red-Black Insertion (5)



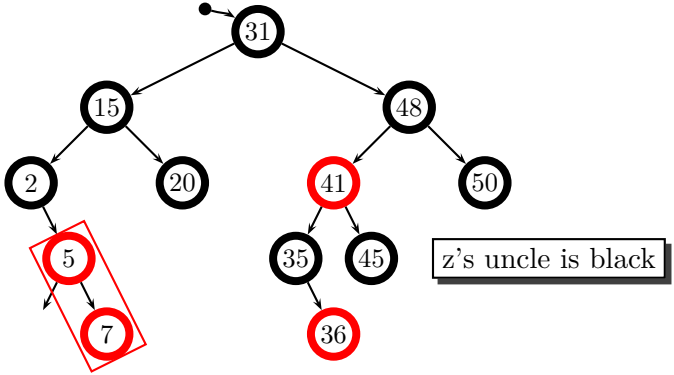
Red-Black Insertion (5)



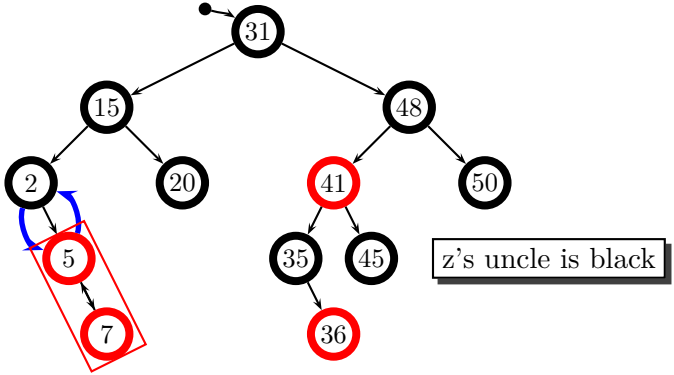
Red-Black Insertion (5)



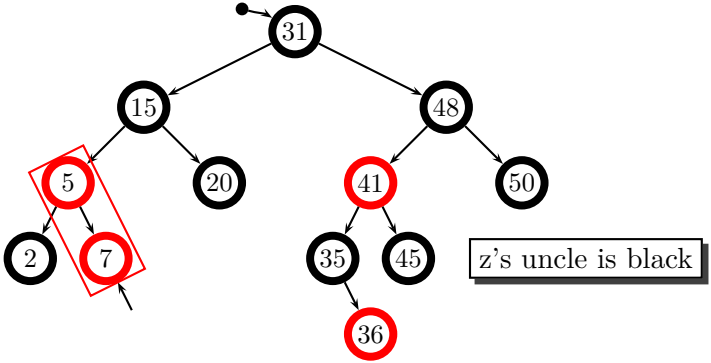
Red-Black Insertion (5)



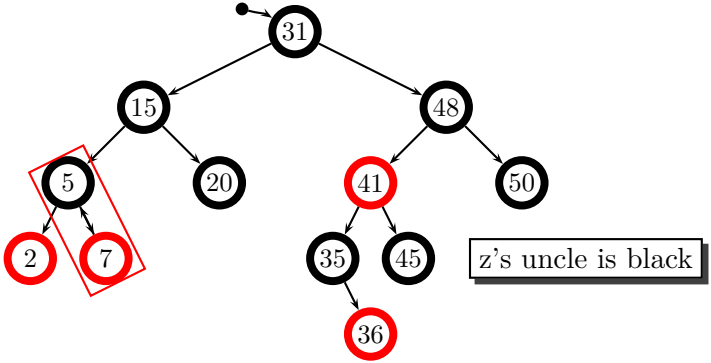
Red-Black Insertion (5)



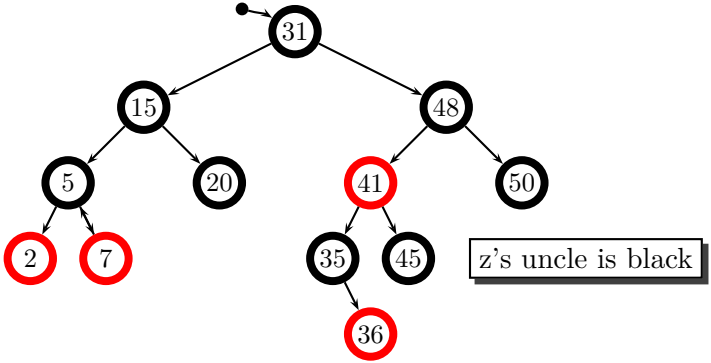
Red-Black Insertion (5)



Red-Black Insertion (5)



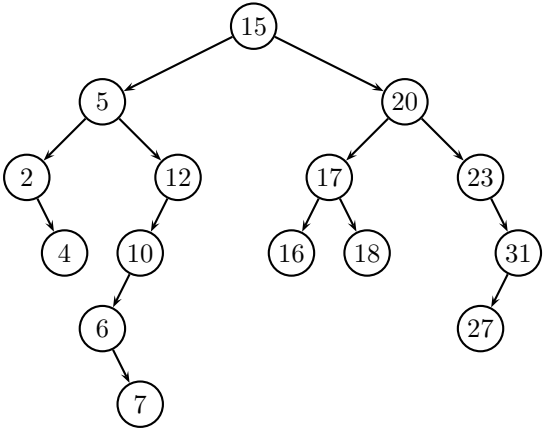
Red-Black Insertion (5)



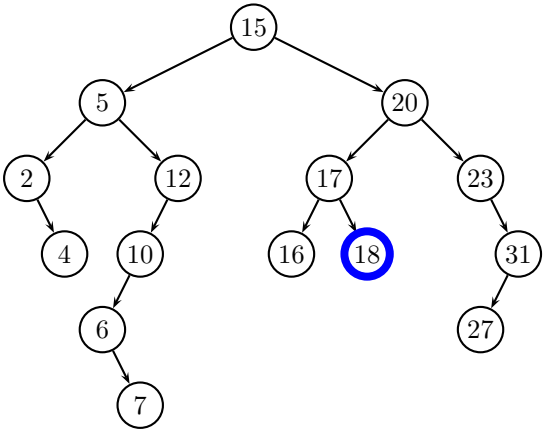
- A zig-zag red–red conflicts can be resolved with a rotation to turn it into an in-line conflict, and then a rotation plus a color switch

Recap on Deletion in Binary Trees

Recap on Deletion in Binary Trees

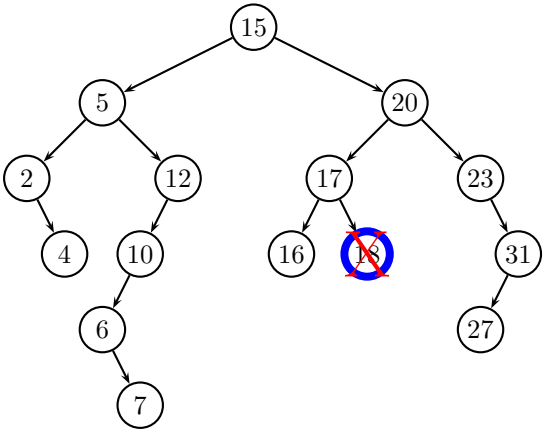


Recap on Deletion in Binary Trees



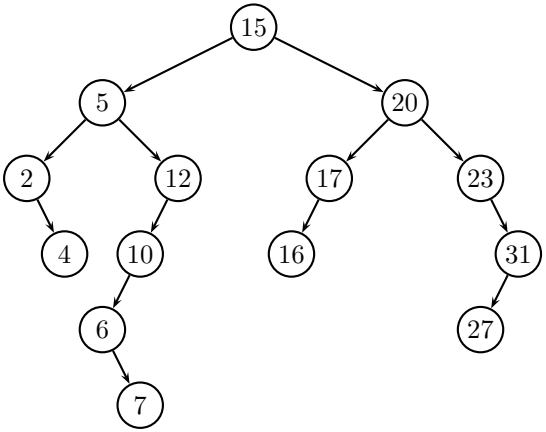
1. z has no children

Recap on Deletion in Binary Trees



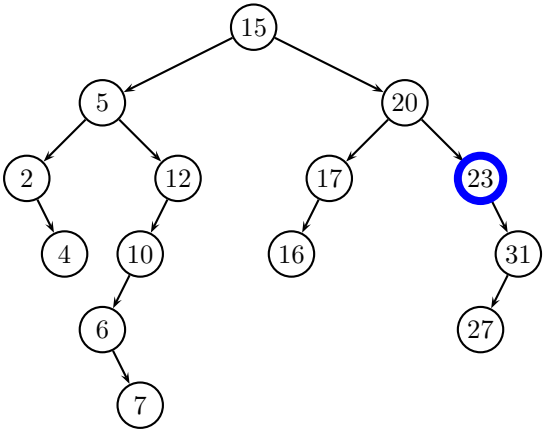
1. z has no children
   - simply remove z

Recap on Deletion in Binary Trees



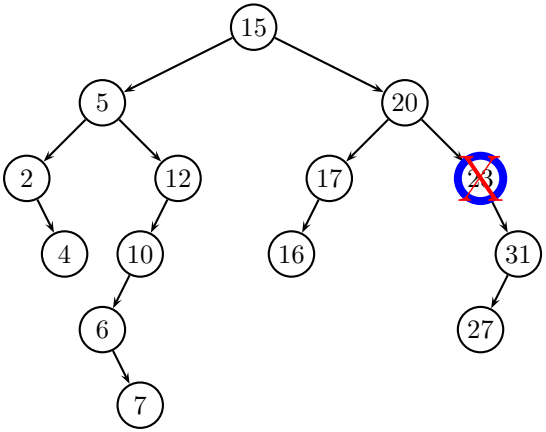
1. z has no children
   - simply remove z

Recap on Deletion in Binary Trees



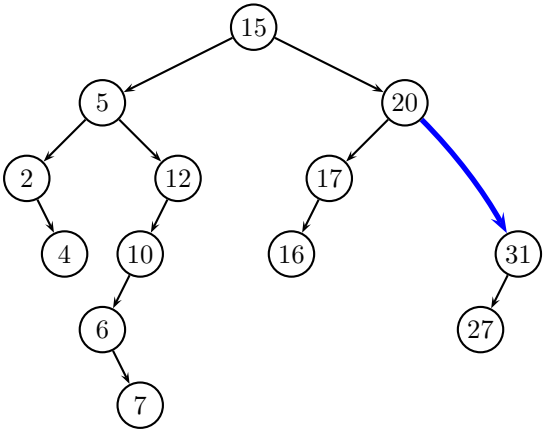
1. z has no children
   - simply remove z
2. z has one child

Recap on Deletion in Binary Trees



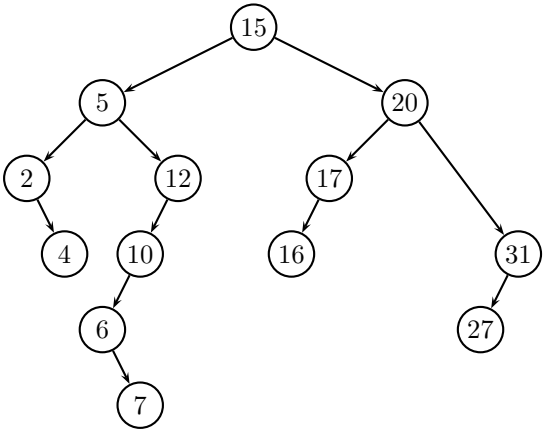
1. z has no children
   - simply remove z
2. z has one child
   - remove z

Recap on Deletion in Binary Trees



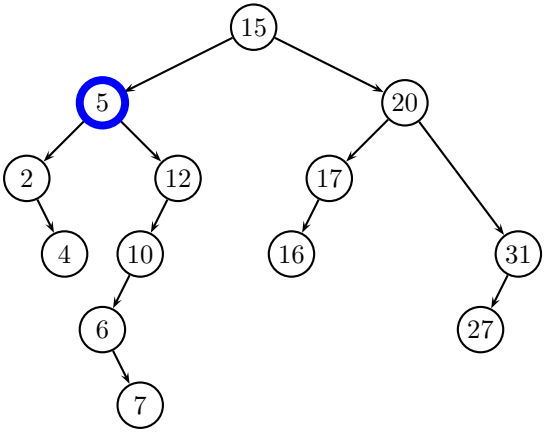
1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect z.parent to z.right

Recap on Deletion in Binary Trees



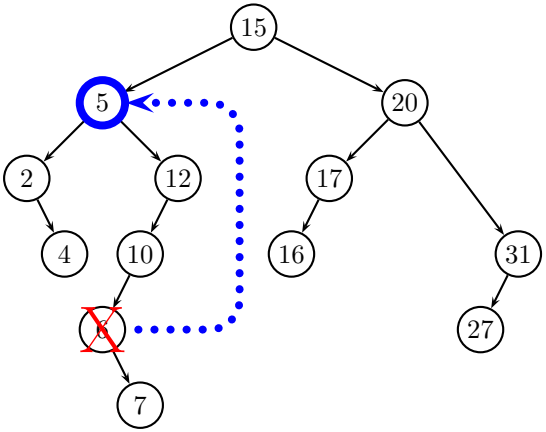
1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect z.parent to z.right

Recap on Deletion in Binary Trees



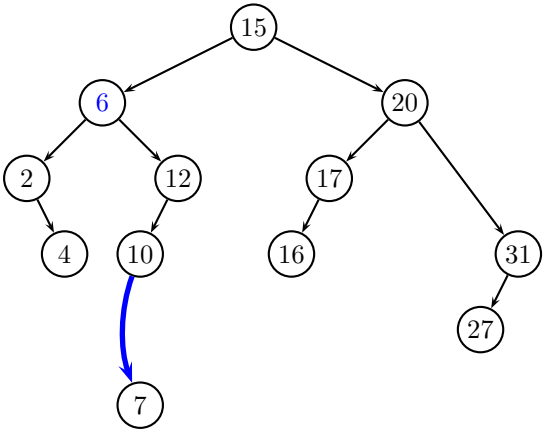
1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect z.parent to z.right
3. z has two children

Recap on Deletion in Binary Trees



1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect z.parent to z.right
3. z has two children
   - replace z with y = Tree-Successor(z)
   - remove y (1 child!)

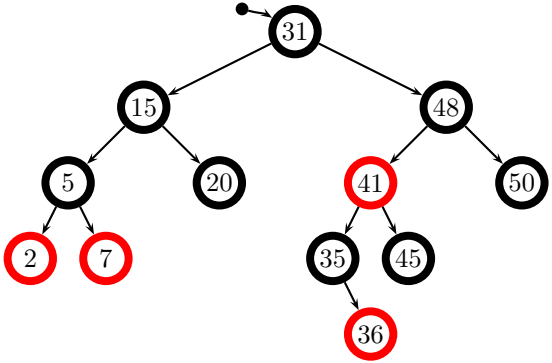
Recap on Deletion in Binary Trees



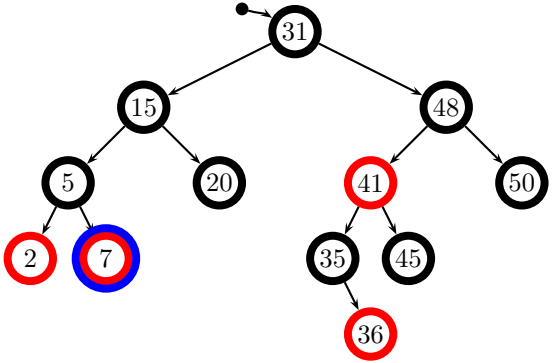
1. z has no children
   - simply remove z
2. z has one child
   - remove z
   - connect z.parent to z.right
3. z has two children
   - replace z with y = Tree-Successor(z)
   - remove y (1 child!)
   - connect y.parent to y.right

Red-Black Deletion

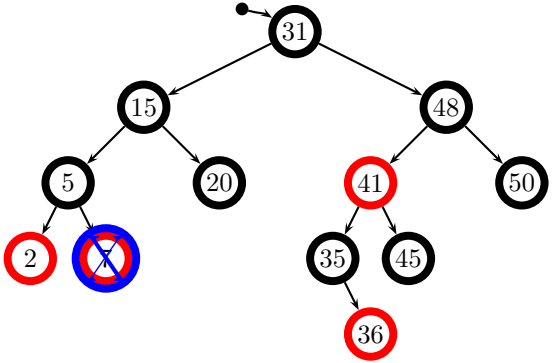
Red-Black Deletion

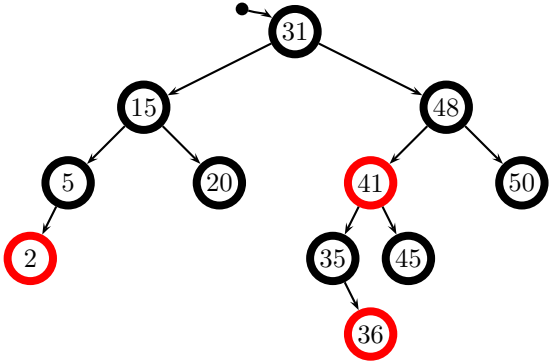


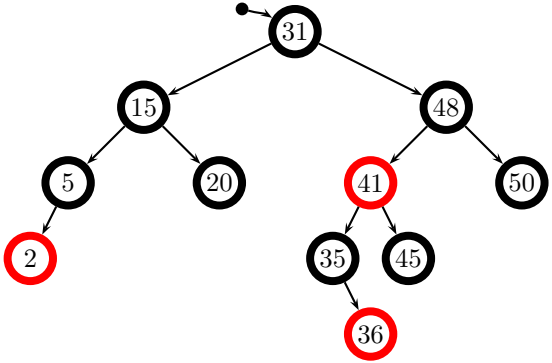
Red-Black Deletion



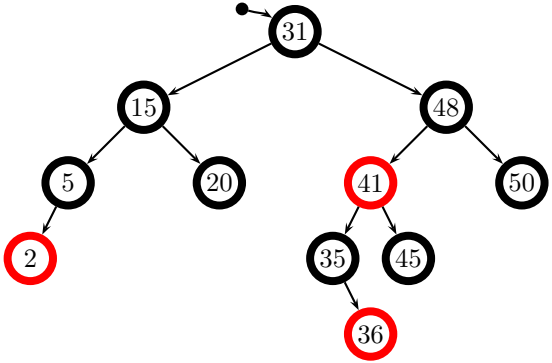
Red-Black Deletion



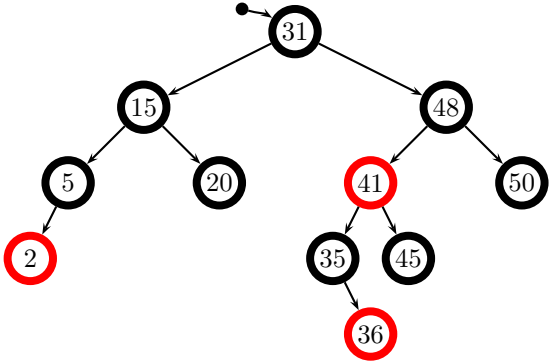

31
15
48
5
20
41
50
2
35
45
36



- A deleting a red leaf does not require any adjustment

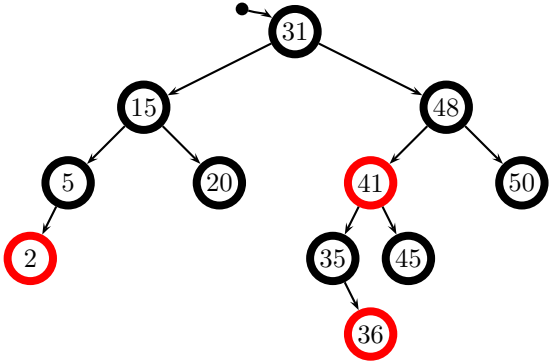


- A deleting a red leaf does not require any adjustment
  - the deletion does not affect the black height (property 5)

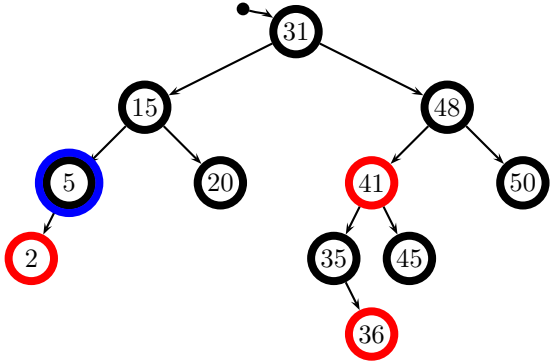


- A deleting a red leaf does not require any adjustment
  - the deletion does not affect the black height (property 5)
  - no two red nodes become adjacent (property 4)

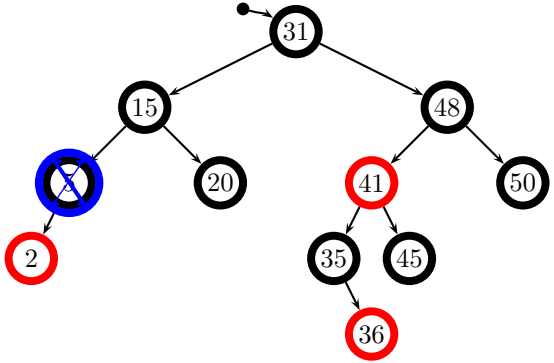
Red-Black Deletion (2)



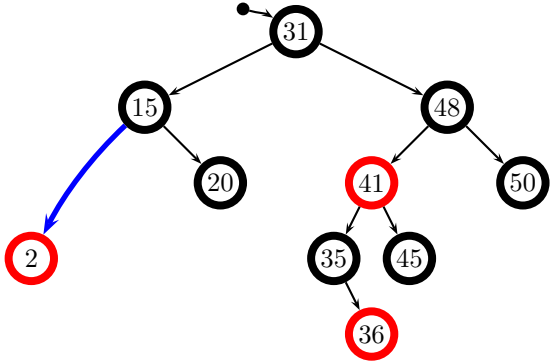
Red-Black Deletion (2)



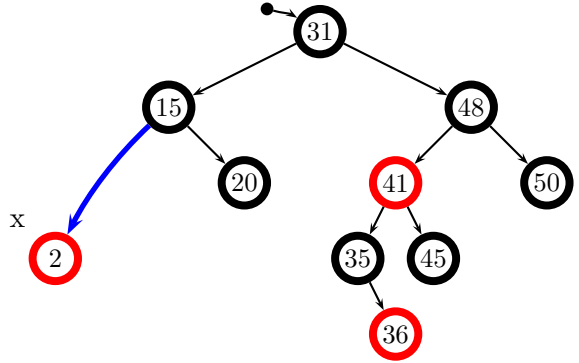
Red-Black Deletion (2)



Red-Black Deletion (2)

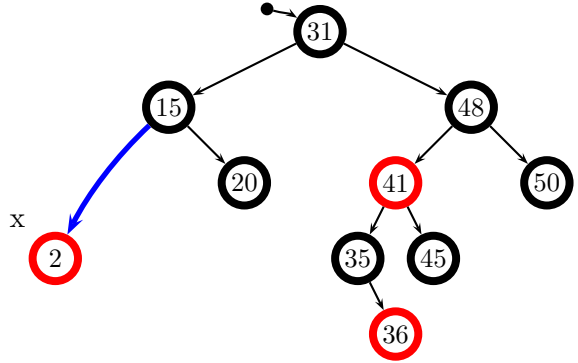


Red-Black Deletion (2)



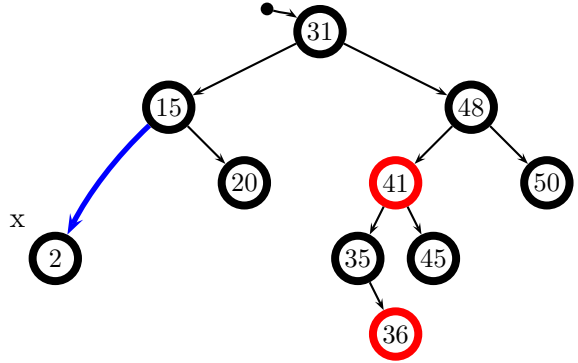
- Deleting a black node changes the balance of black-height in a subtree x

Red-Black Deletion (2)



- Deleting a black node changes the balance of black-height in a subtree x
  - RB-Delete-Fixup(T, x) fixes the tree after a deletion

Red-Black Deletion (2)



- Deleting a black node changes the balance of black-height in a subtree x
  - RB-Delete-Fixup(T, x) fixes the tree after a deletion
  - in this simple case: x.color = black

Fixup Conditions

Fixup Conditions

- y is the spliced node (y = z if z has zero or one child)
  - if y is red, then no fixup is necessary
  - so, here we assume that y is black

Fixup Conditions

- y is the spliced node (y = z if z has zero or one child)
  - if y is red, then no fixup is necessary
  - so, here we assume that y is black
- x is either y's only child or T.nil
  - y was spliced out, so y can not have two children
  - x = T.nil iff y has no (key-bearing) children

Fixup Conditions

- y is the spliced node (y = z if z has zero or one child)
  - if y is red, then no fixup is necessary
  - so, here we assume that y is black
- x is either y's only child or T.nil
  - y was spliced out, so y can not have two children
  - x = T.nil iff y has no (key-bearing) children
- Problem 1: y = T.root and x is red
  - violates red-black property ?? (root must be black)

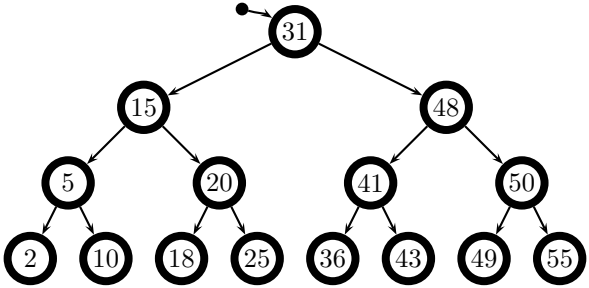
Fixup Conditions

- y is the spliced node (y = z if z has zero or one child)
  - if y is red, then no fixup is necessary
  - so, here we assume that y is black
- x is either y's only child or T.nil
  - y was spliced out, so y can not have two children
  - x = T.nil iff y has no (key-bearing) children
- Problem 1: y = T.root and x is red
  - violates red-black property ?? (root must be black)
- Problem 2: both x and y.parent are red
  - violates red-black property 4 (no adjacent red nodes)

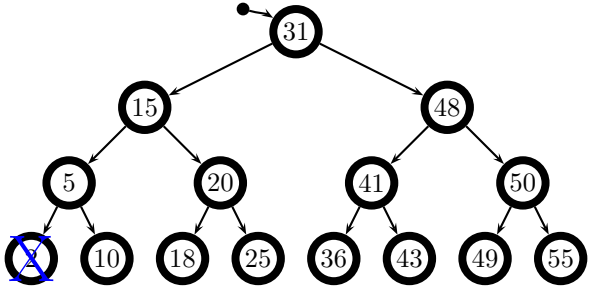
Fixup Conditions

- y is the spliced node (y = z if z has zero or one child)
  - if y is red, then no fixup is necessary
  - so, here we assume that y is black
- x is either y's only child or T.nil
  - y was spliced out, so y can not have two children
  - x = T.nil iff y has no (key-bearing) children
- Problem 1: y = T.root and x is red
  - violates red-black property ?? (root must be black)
- Problem 2: both x and y.parent are red
  - violates red-black property 4 (no adjacent red nodes)
- Problem 3: we are removing y, which is black
  - violates red-black property 5 (same black height for all paths)

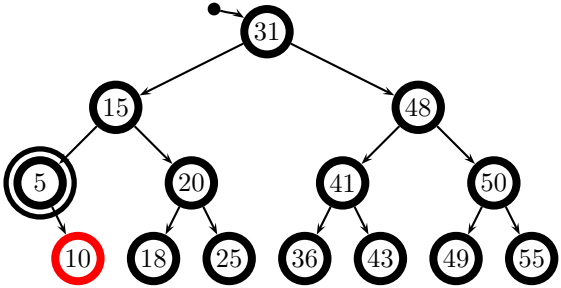
Red-Black Deletion (3)



Red-Black Deletion (3)

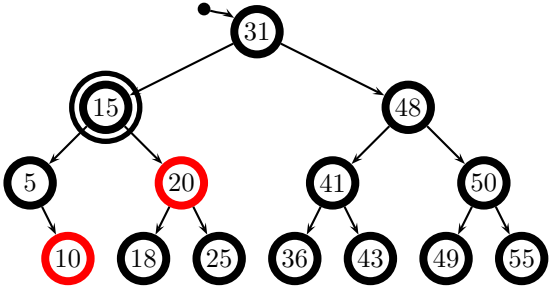


Red-Black Deletion (3)



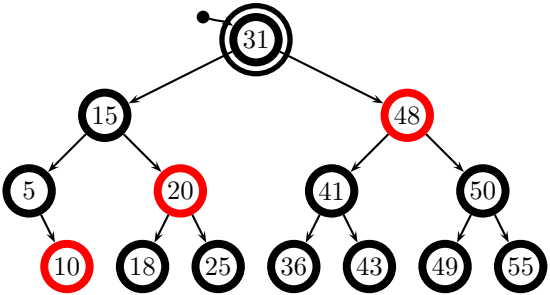
- x carries an additional black weight
  - the fixup algorithm pushes it up towards to root

Red-Black Deletion (3)



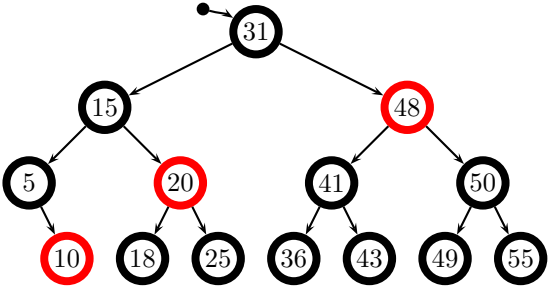
- x carries an additional black weight
  - the fixup algorithm pushes it up towards to root

Red-Black Deletion (3)



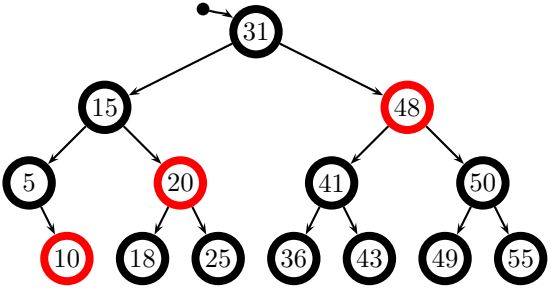
- x carries an additional black weight
  - the fixup algorithm pushes it up towards to root

Red-Black Deletion (3)



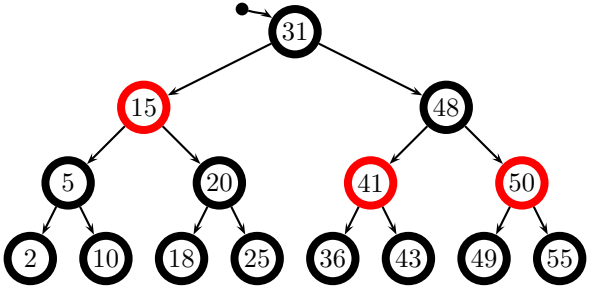
- x carries an additional black weight
  - the fixup algorithm pushes it up towards to root

Red-Black Deletion (3)

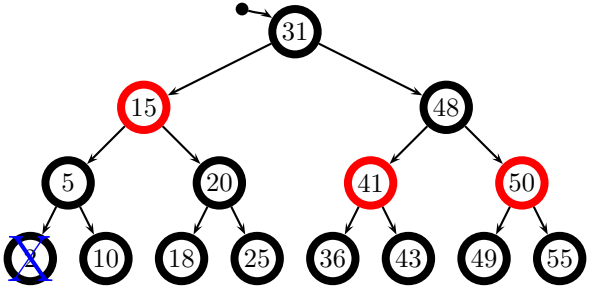


- x carries an additional black weight
  - the fixup algorithm pushes it up towards to root
- The additional black weight can be discarded if it reaches the root, otherwise...

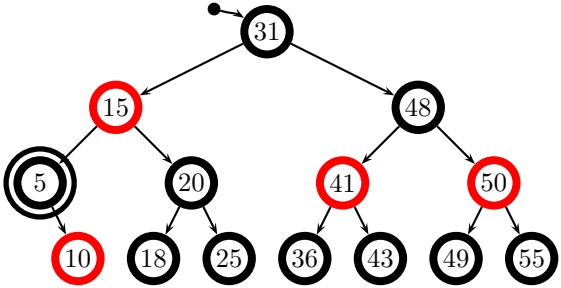
Red-Black Deletion (4)



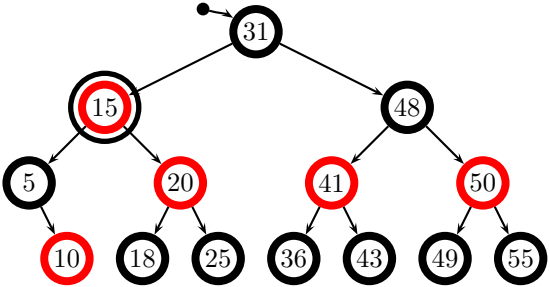
Red-Black Deletion (4)



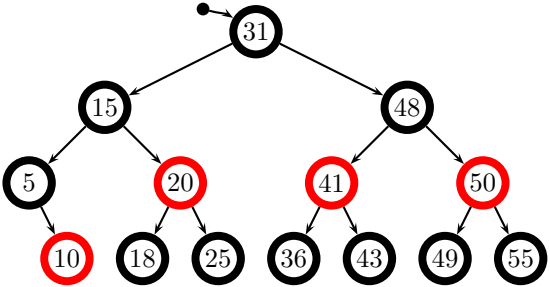
Red-Black Deletion (4)



Red-Black Deletion (4)

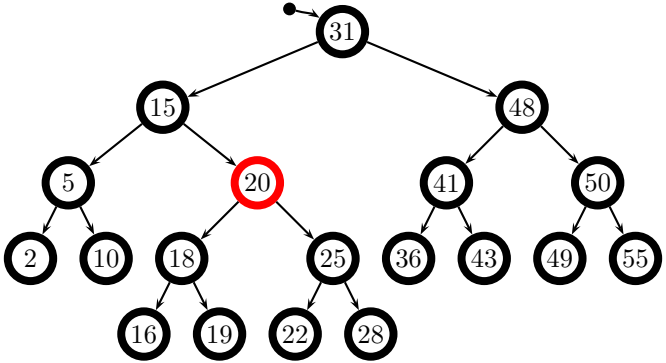


Red-Black Deletion (4)

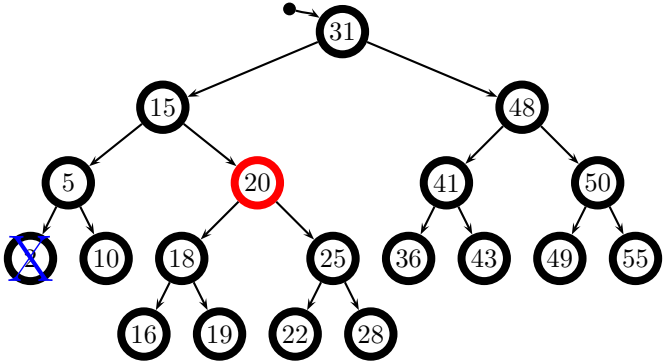


- The additional black weight can also stop as soon as it reaches a red node, which will absorb the extra black color

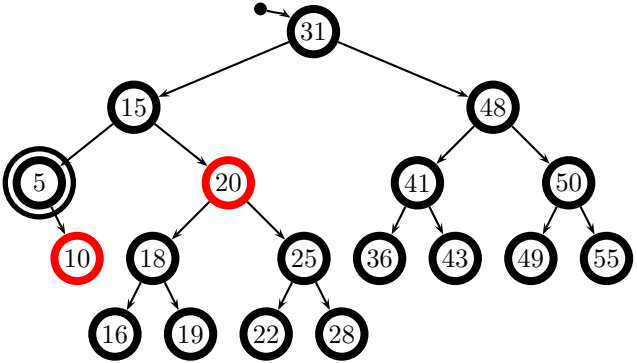
Red-Black Deletion (5)



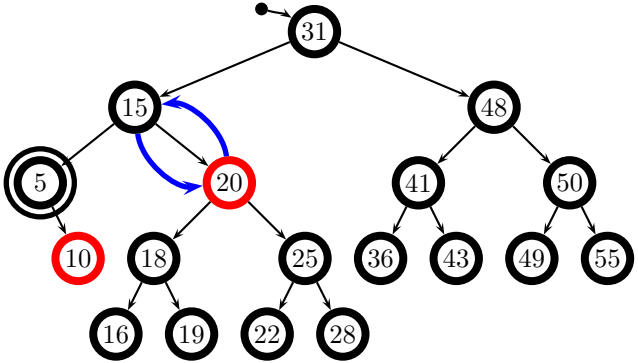
Red-Black Deletion (5)



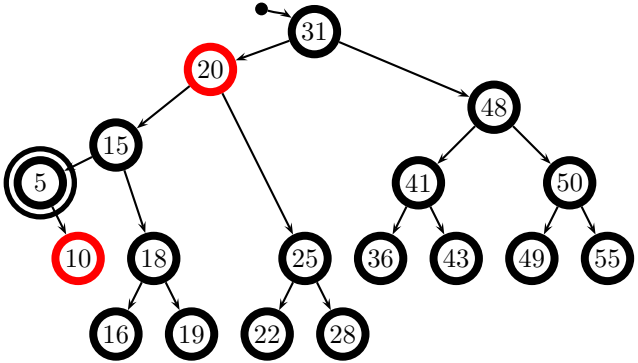
Red-Black Deletion (5)



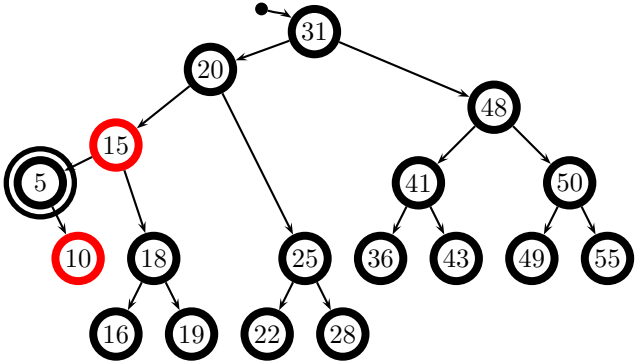
Red-Black Deletion (5)



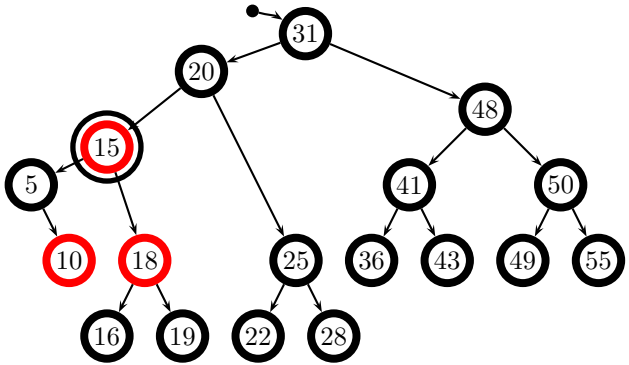
Red-Black Deletion (5)



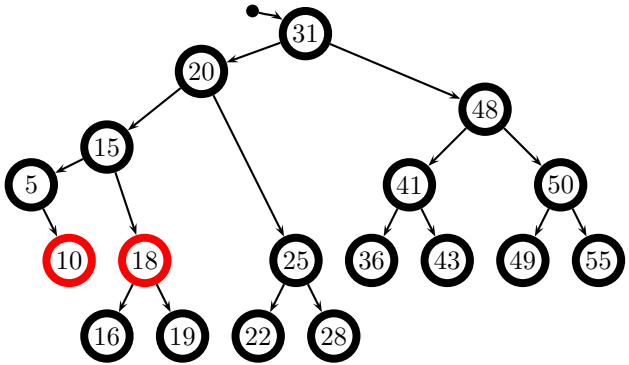
Red-Black Deletion (5)



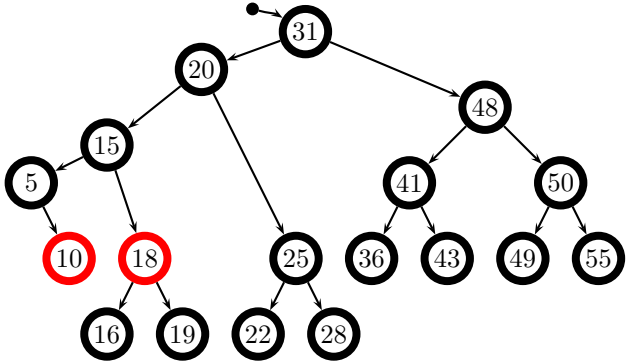
Red-Black Deletion (5)



Red-Black Deletion (5)



Red-Black Deletion (5)



- In other cases where we can not push the additional black color up, we can apply appropriate rotations and color transfers that preserve all other red-black properties

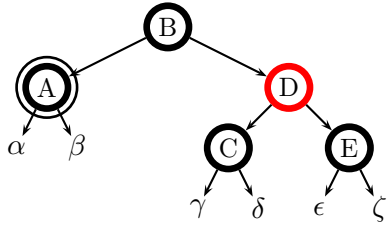
Basic Fixup Iteration (1)

Basic Fixup Iteration (1)

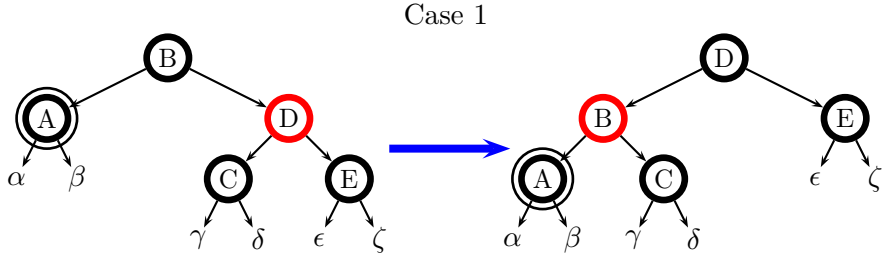
Case 1

Basic Fixup Iteration (1)

Case 1

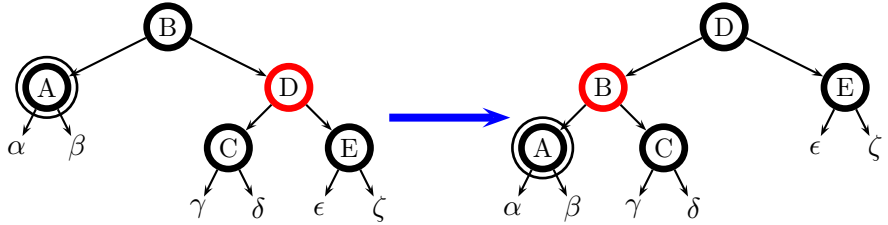


Basic Fixup Iteration (1)



Basic Fixup Iteration (1)

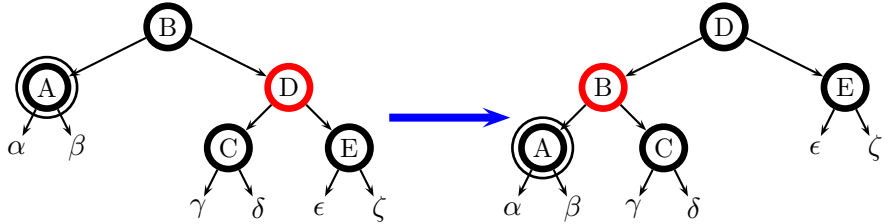
Case 1



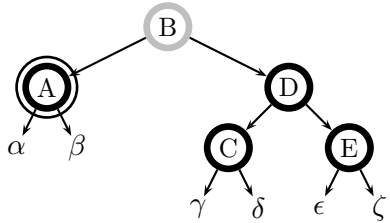
Case 2

Basic Fixup Iteration (1)

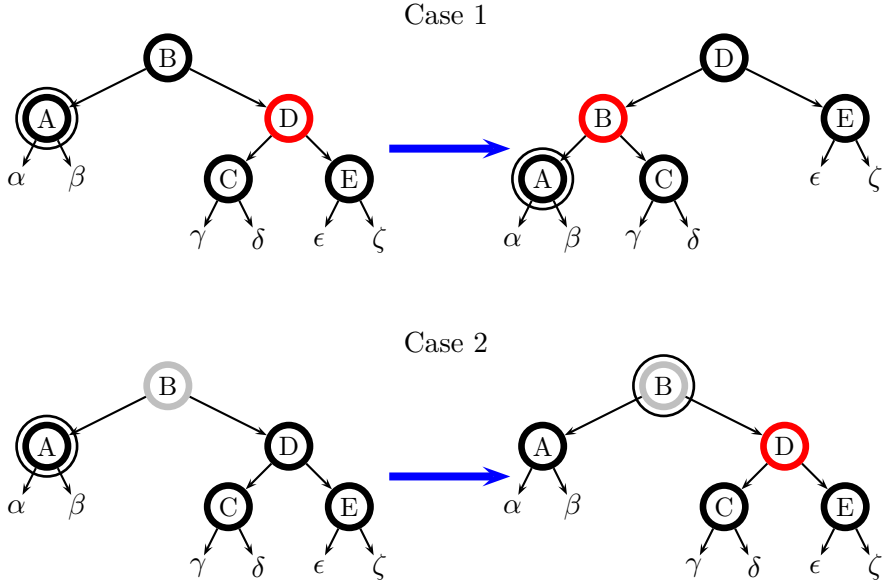
Case 1



Case 2



Basic Fixup Iteration (1)

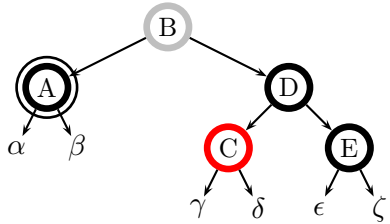


Basic Fixup Iteration (2)

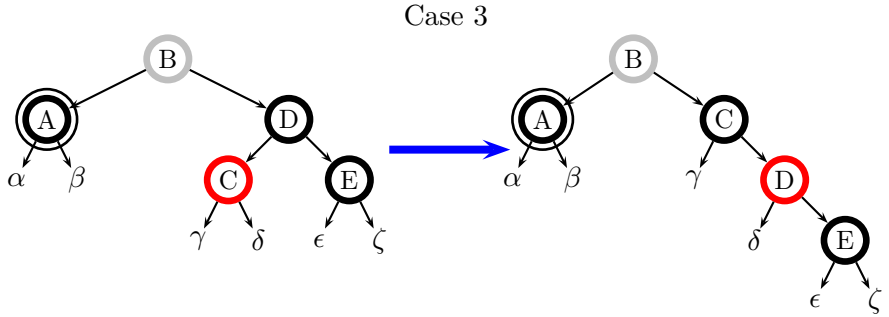
Case 3

Basic Fixup Iteration (2)

Case 3

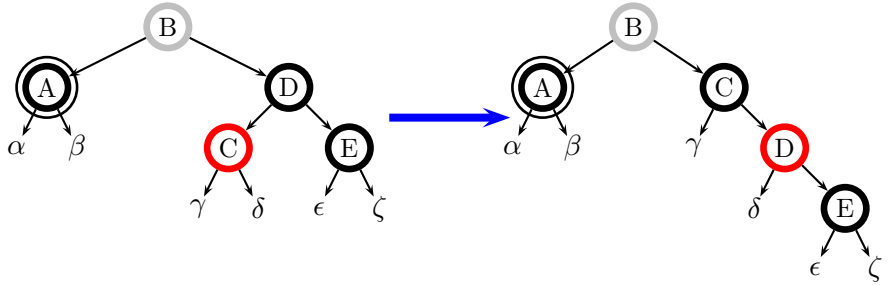


Basic Fixup Iteration (2)

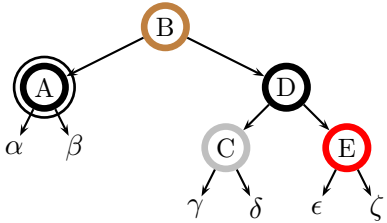


Basic Fixup Iteration (2)

Case 3

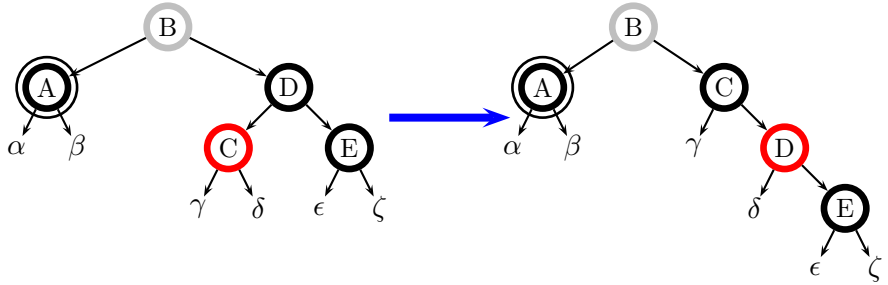


Case 4

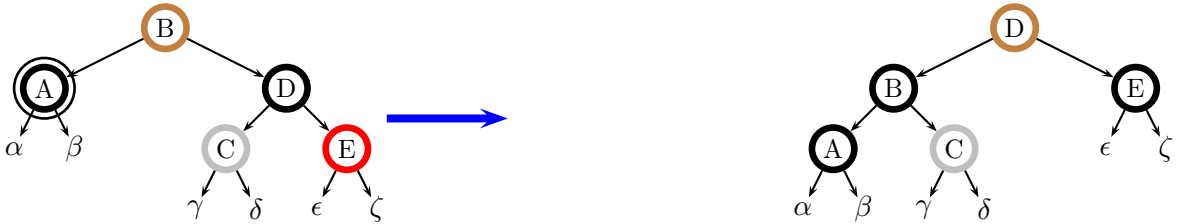


Basic Fixup Iteration (2)

Case 3



Case 4



Red-Black Delete Fixup

```
RB-Delete-Fixup(T, x)  1   while x ≠ T.root ∧ x.color = black
                       2       if x == x.parent.left
                       3           w = x.parent.right
                       4           if w.color == red
                       5               case 1...
                       6           if w.left.color == black ∧ w.right.color = black
                       7               w.color = red                    // case 2
                       8               x = x.parent
                       9           else if w.right.color == black
                      10                   case 3...
                      11               case 4...
                      12       else same as above, exchanging right and left
                      13   x.color = black
```