Hashing

CSI2103-02 Data Structures

Yonsei University, Spring 2020

Hyung-Chan An

Hash function

$Pr[h(x)=h(y) \mid a_1=1, a_2=3, a_3=7, a_4=8]$ $= \frac{1}{m}$

- Consider any $x \neq y$; assume $x_0 \neq y_0$
- Given $a_1, \ldots, a_k$, what is the cond. prob. that $h(x)=h(y)$?
- $a_0(x_0 - y_0) \equiv -\sum_{i=1}^{k} a_i(x_i - y_i) \pmod{m}$
- $a_0 \equiv -(x_0 - y_0)^{-1} \cdot \sum_{i=1}^{k} a_i(x_i - y_i) \pmod{m}$
- Thus the cond. prob. is $1/m$ and this does not depend on the choice of $a_1, \ldots, a_k$
- The uncond. prob. is therefore also $1/m$

- Universal hashing
  - m: hash table size
  - For $x \neq y$, $Pr[h(x)=h(y)]=1/m$

$Pr[h(x)=h(y) \mid a_1=2, a_2=4, a_3=8, a_4=10]$ $= \frac{1}{m}$

- Choose prime m
- Choose a "piece size" (radix or base) $<m$
- Choose $a_0, \ldots, a_k$ independently and uniformly at random from $\{0, \ldots, m-1\}$

- Decompose the key into $x_0, \ldots, x_k$
- $h(x) = \left(\sum_{i=0}^{k} a_i x_i\right) \bmod m$

$\sum_{n=0}^{k} a_n y_n$

$(x_0 - y_0) \neq 0$ m

h

Dealing with collisions

- *Open addressing*
  - Deletion
    - "deleted" flag
- *Chaining*

Delete 89003

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]		[99]
		90002 *data*	deleted	72003 *data*	73005 *data*			…	

Dealing with collisions

- Load factor
  - #elements / hash table size
  - Performance as a function of load factor

Feedback on the last minute paper

- What are the disadvantages of hash tables?

  - Worst-case running time
  - Space overhead

Advanced Data Structures

CSI2103-02 Data Structures

Yonsei University, Spring 2020

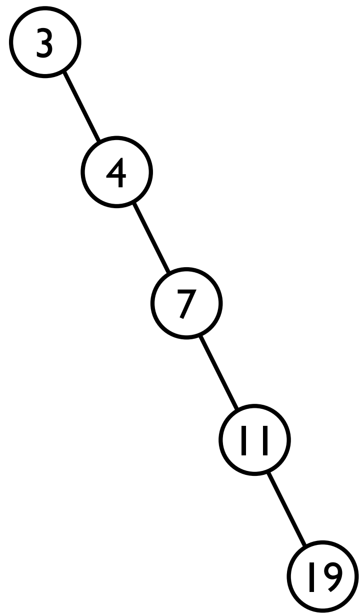
Hyung-Chan An

Binary search tree

- $O(h)$ search
- $O(h)$ addition
- $O(h)$ deletion

Worst case

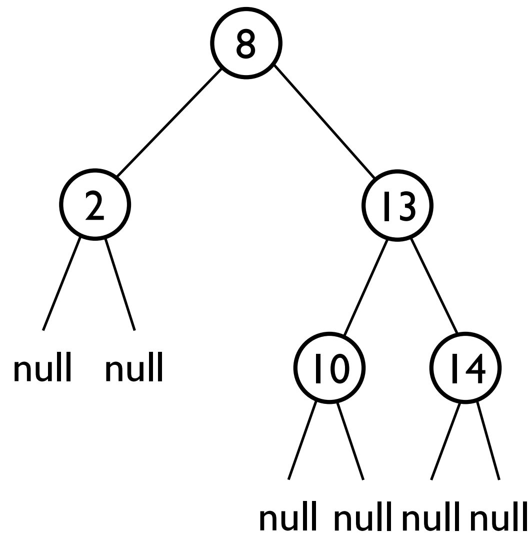
- Insertion in ascending order



- $h=O(n)$
- Balanced tree

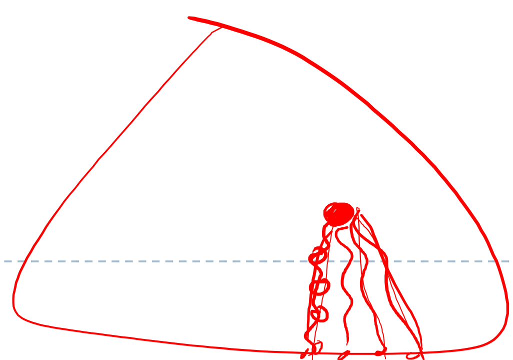
Red-black trees

- For notational simplicity, we will treat the "null pointers" as leaf nodes

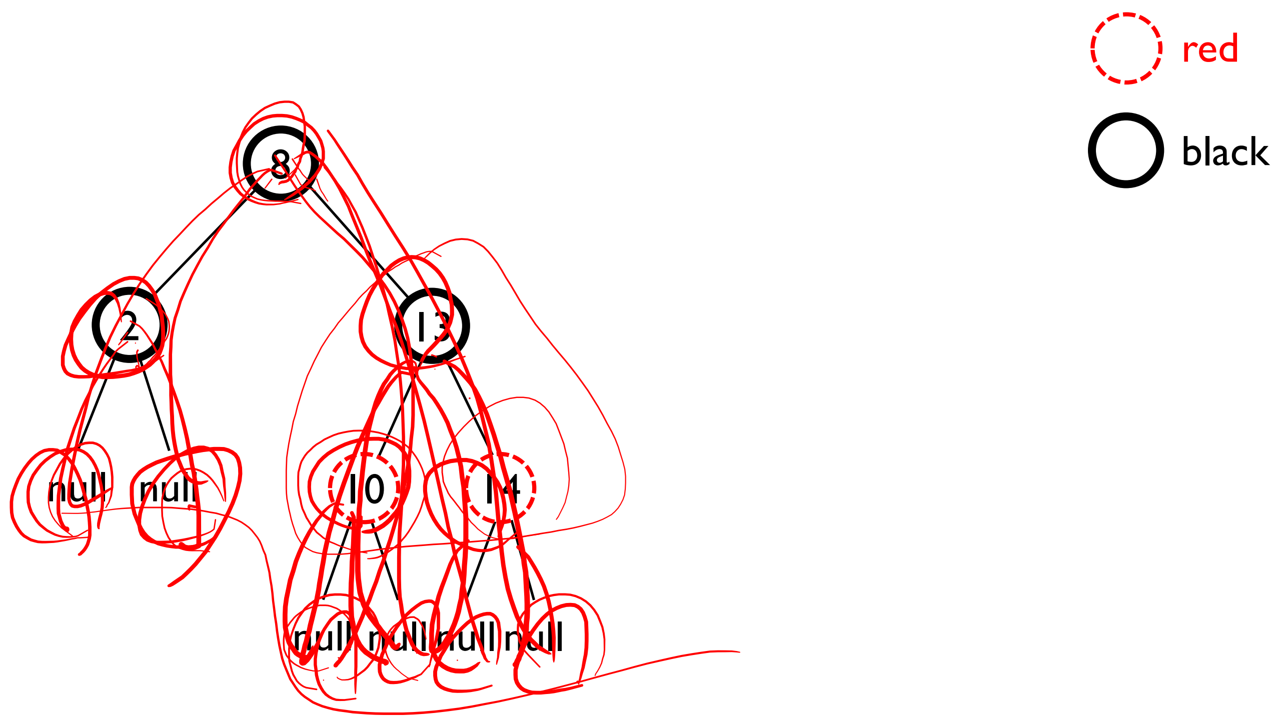


Red-black trees

- For notational simplicity, we will treat the "null pointers" as leaf nodes
- We say a binary search tree is a *red-black tree* if the following five properties are satisfied:
  - (P1) Every node is colored either red or black
  - (P2) The root is black
  - (P3) Every leaf/null pointer is black
  - (P4) If a node is red, both its children are black
  - (P5) Every simple path from a node to a descendant leaf contains the same #black nodes



Red-black trees



- For notational simplicity, we will treat the "null pointers" as leaf nodes
- We say a binary search tree is a *red-black tree* if the following five properties are satisfied:
  - (P1) Every node is colored either red or black
  - (P2) The root is black
  - (P3) Every leaf/null pointer is black
  - (P4) If a node is red, both its children are black
  - (P5) Every simple path from a node to a descendant leaf contains the same #black nodes

Red-black trees

- (P5) Every simple path from a node to a descendant leaf contains the same #black nodes

- We call #black nodes on any path from, but not including, a node *x* to a descendant leaf its *black-height* bh(*x*): well-defined due to (P5)

Red-black trees

- Why are the five properties of red-black trees useful?

Lemma A red-black tree with n (internal) nodes has height $\leq 2\log_2(n+1)-1$

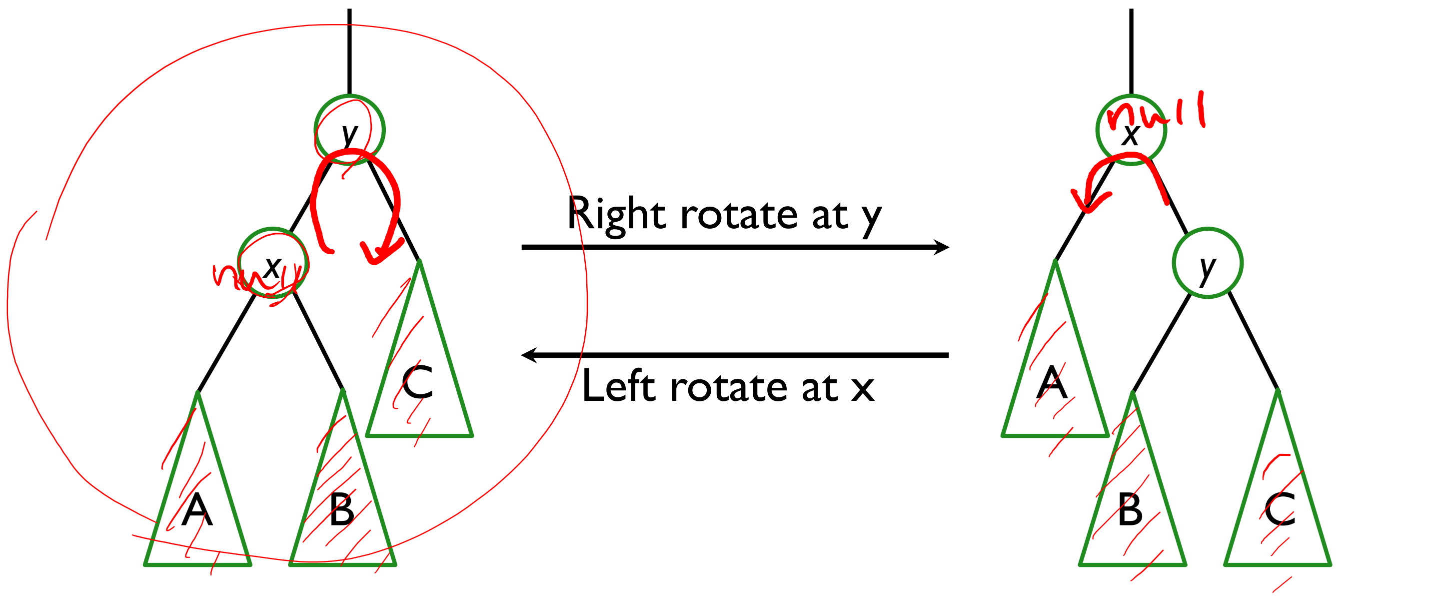
Proof

Claim: (#internal nodes in a subtree rooted at x) $\geq 2^{\mathrm{bh}(x)}-1$.

Proof: Trivial if the "subtree" is empty. Use induction on the *height* of the subtree. If the height is 0, easy. O/w, each child of x has black-height $\geq \mathrm{bh}(x)-1$ and #nodes is therefore $\geq 2\cdot(2^{\mathrm{bh}(x)-1}-1)+1 = 2^{\mathrm{bh}(x)}-1$.

Let h be the height of the tree with n internal nodes. We then have $n \geq 2^{(h+1)/2}-1$ since the black-height of the root is $\geq (h+1)/2$.

Rotations

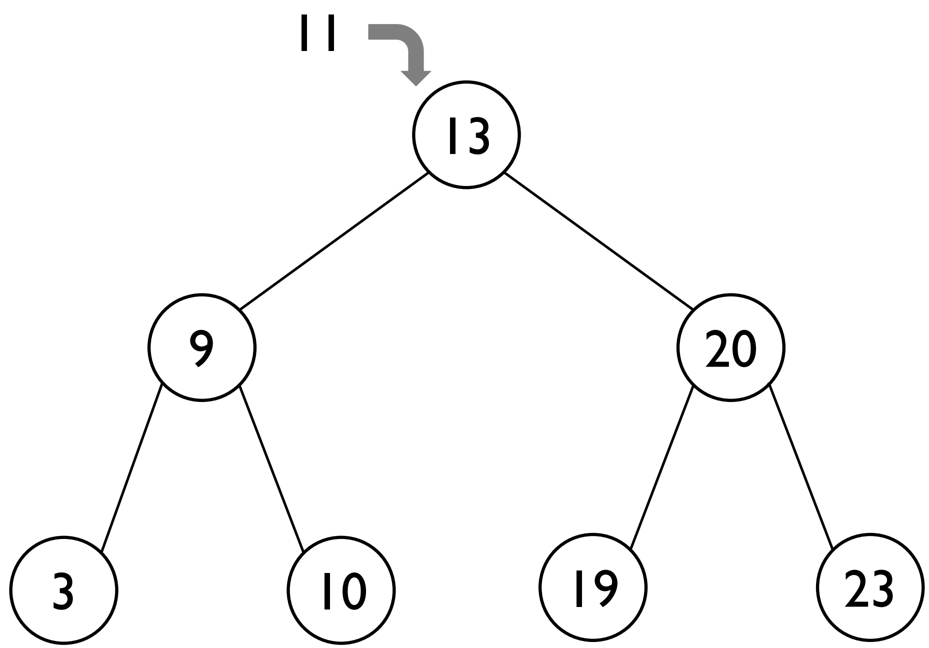


- Binary search tree properties are preserved
- It is important to make sure *x* (and *y*) is an internal node

x&y cannot be null

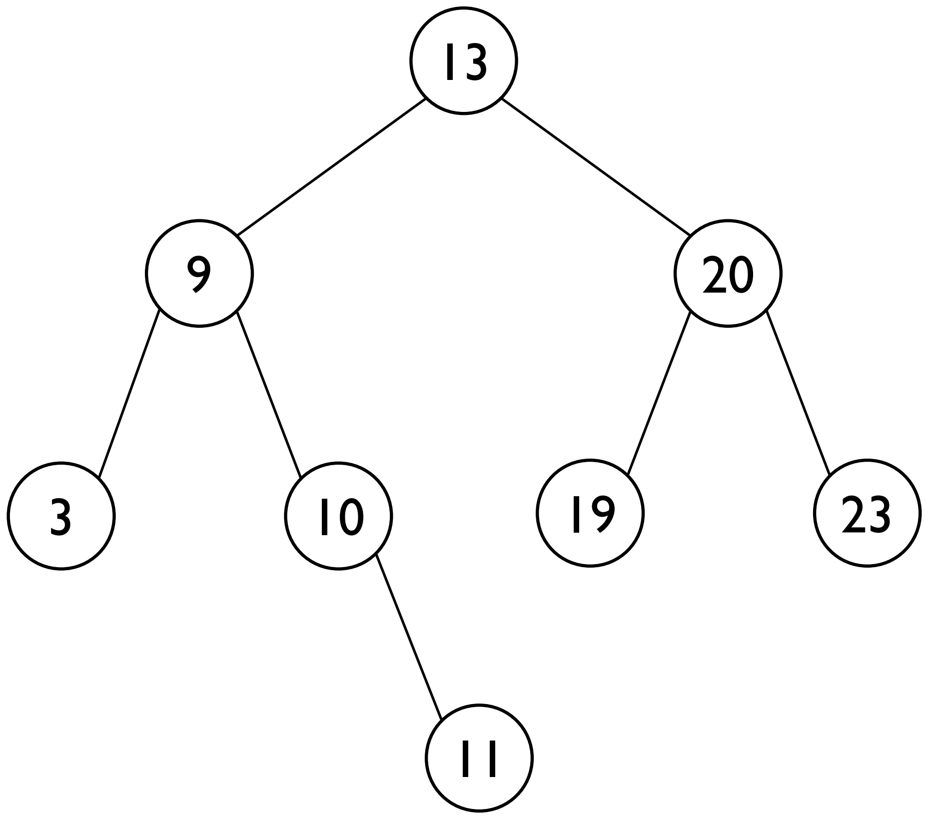
- O(1)-time operation

Insertion into a binary search tree



Insertion into a binary search tree

- $O(h)$ time, where h is the height of the binary search tree

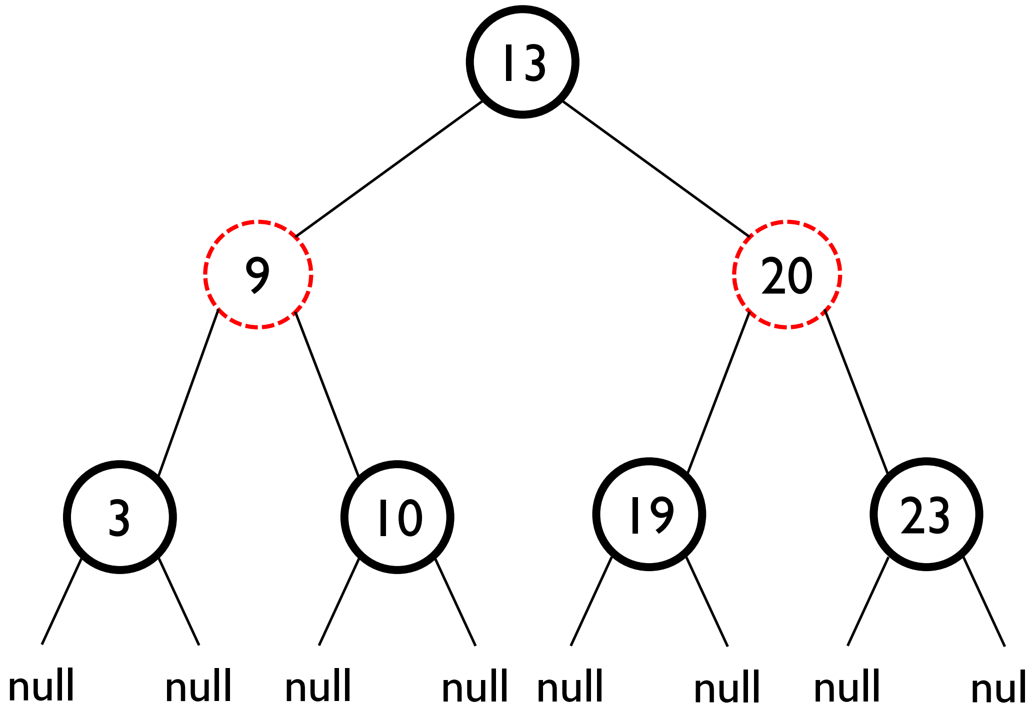


Insertion into a red-black tree

- Insert as usual; color the new node red

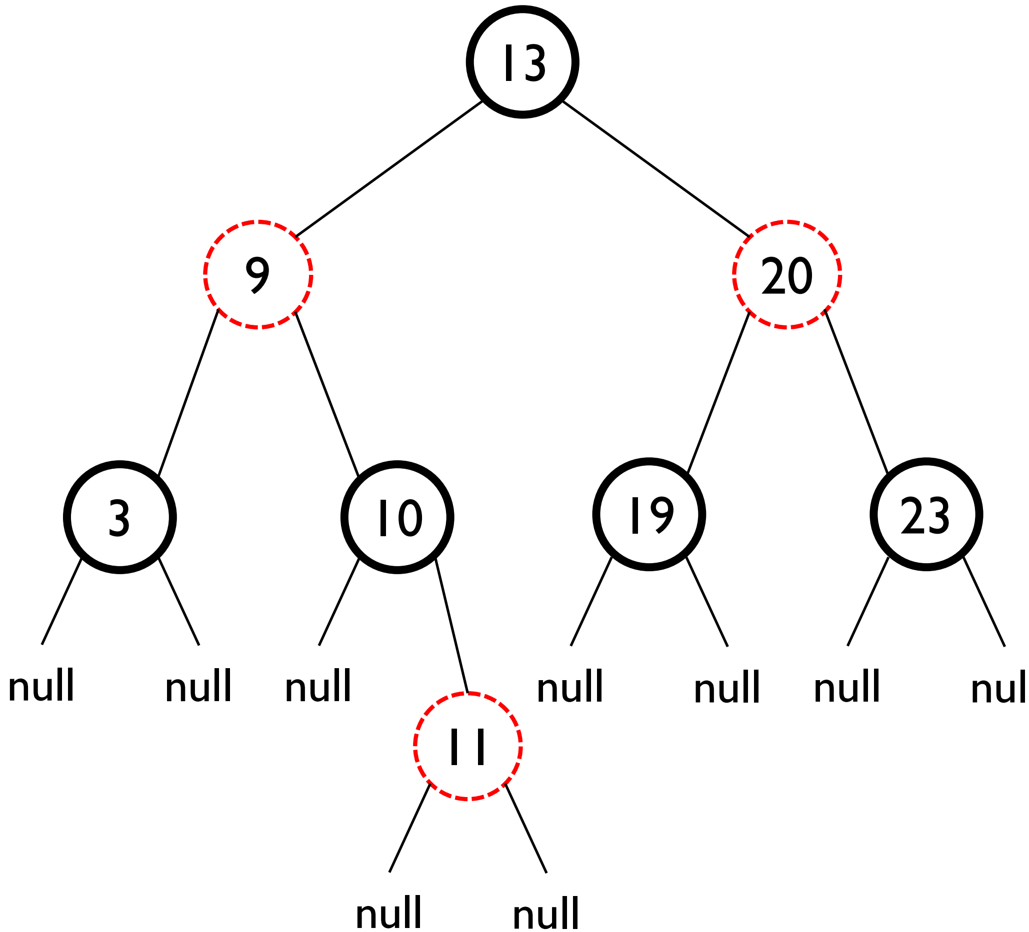
Insertion into a binary search tree

- $O(h)$ time, where h is the height of the binary search tree



Insertion into a binary search tree

- $O(h)$ time, where h is the height of the binary search tree



Insertion into a red-black tree

- Insert as usual; color the new node red
- Only (P2) or (P4) can be violated

(if the parent is red)

(if the new node is the root node)

(P1) Every node is colored either red or black
(P2) The root is black
(P3) Every leaf/null pointer is black
(P4) If a node is red, both its children are black
(P5) Every simple path from a node to a descendant leaf contains the same #black nodes

Insertion into a red-black tree

x ← the newly inserted node
while x is not the root and its parent is red
  (assume the parent is the left child of the grandparent)
  Case 1
  if the "uncle" is red
    color the parent and the uncle black
    color the grandparent red; the grandparent becomes x
  else
  Case 2
    if x is the right child of its parent
      left-rotate at the parent; the former parent becomes x
  Case 3
    color the parent black
    color the grandparent red
    right-rotate at the grandparent
    x ← the root node
color the root black

(P1) Every node is colored either red or black
(P2) ~~The root is black~~
(P3) Every leaf/null pointer is black
(P4) If a node is red, both its children are black
(P5) Every simple path from a node to a descendant leaf contains the same #black nodes

Insertion into a red-black tree

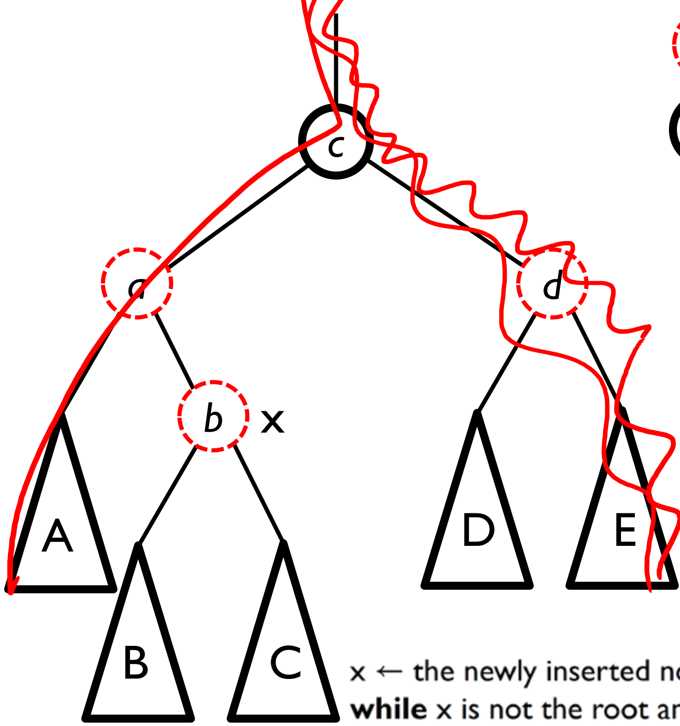
- x always points to a red node (whose parent may be red)
- (P1), (P3) & (P5) are maintained; we work towards (P2) & (P4)
- The grandparent always exists unless the parent is black (assuming the node itself is not the root)

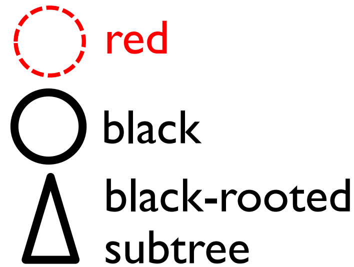
```
x ← the newly inserted node
while x is not the root and its parent is red
   (assume the parent is the left child of the grandparent)
 Case 1 [ if the "uncle" is red
        [    color the parent and the uncle black
        [    color the grandparent red; the grandparent becomes x
   else
 Case 2 [    if x is the right child of its parent
        [       left-rotate at the parent; the former parent becomes x
 Case 3 [    color the parent black
        [    color the grandparent red
        [    right-rotate at the grandparent
        [    x ← the root node
color the root black
```

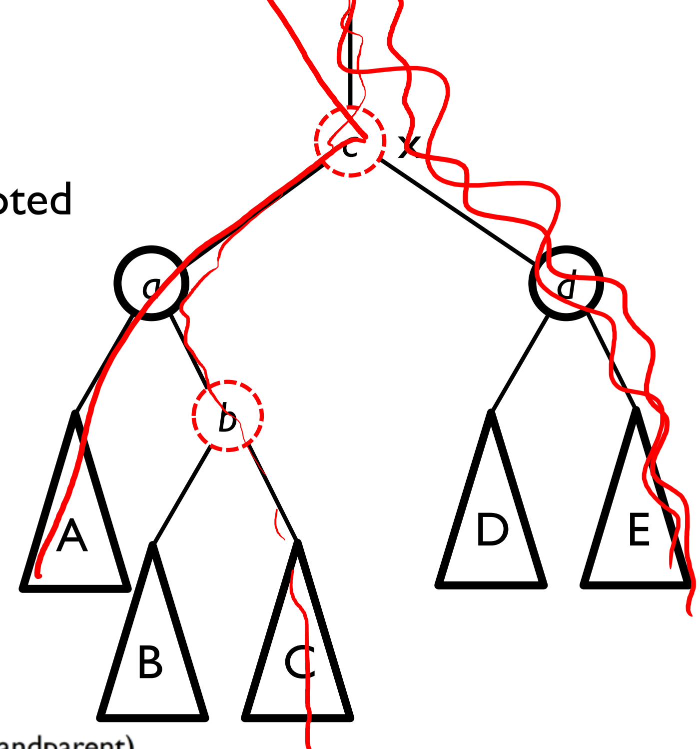
(P1) Every node is colored either red or black
(P2) The root is black
(P3) Every leaf/null pointer is black
(P4) If a node is red, both its children are black
(P5) Every simple path from a node to a descendant leaf contains the same #black nodes

Insertion into a red-black tree

Case 1





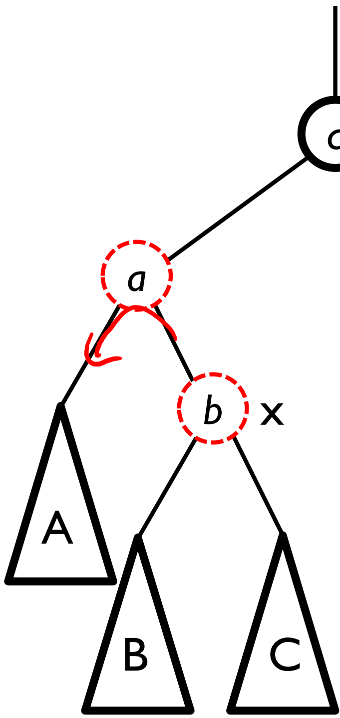


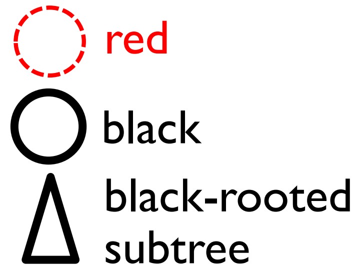
x ← the newly inserted node
while x is not the root and its parent is red
  (assume the parent is the left child of the grandparent)
  Case 1 [**if** the "uncle" is red
    color the parent and the uncle black
    color the grandparent red; the grandparent becomes x
  else
    Case 2 [**if** x is the right child of its parent
      left-rotate at the parent; the former parent becomes x
    Case 3 [color the parent black
    color the grandparent red
    right-rotate at the grandparent
    x ← the root node
color the root black

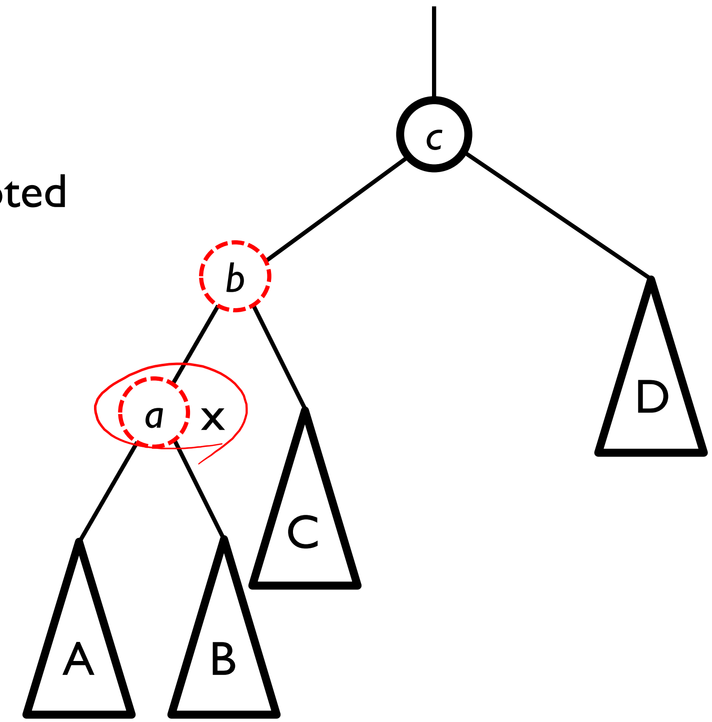
(P1) Every node is colored either red or black
(P2) The root is black
(P3) Every leaf/null pointer is black
(P4) If a node is red, both its children are black
(P5) Every simple path from a node to a descendant leaf contains the same #black nodes

Insertion into a red-black tree

Case 2





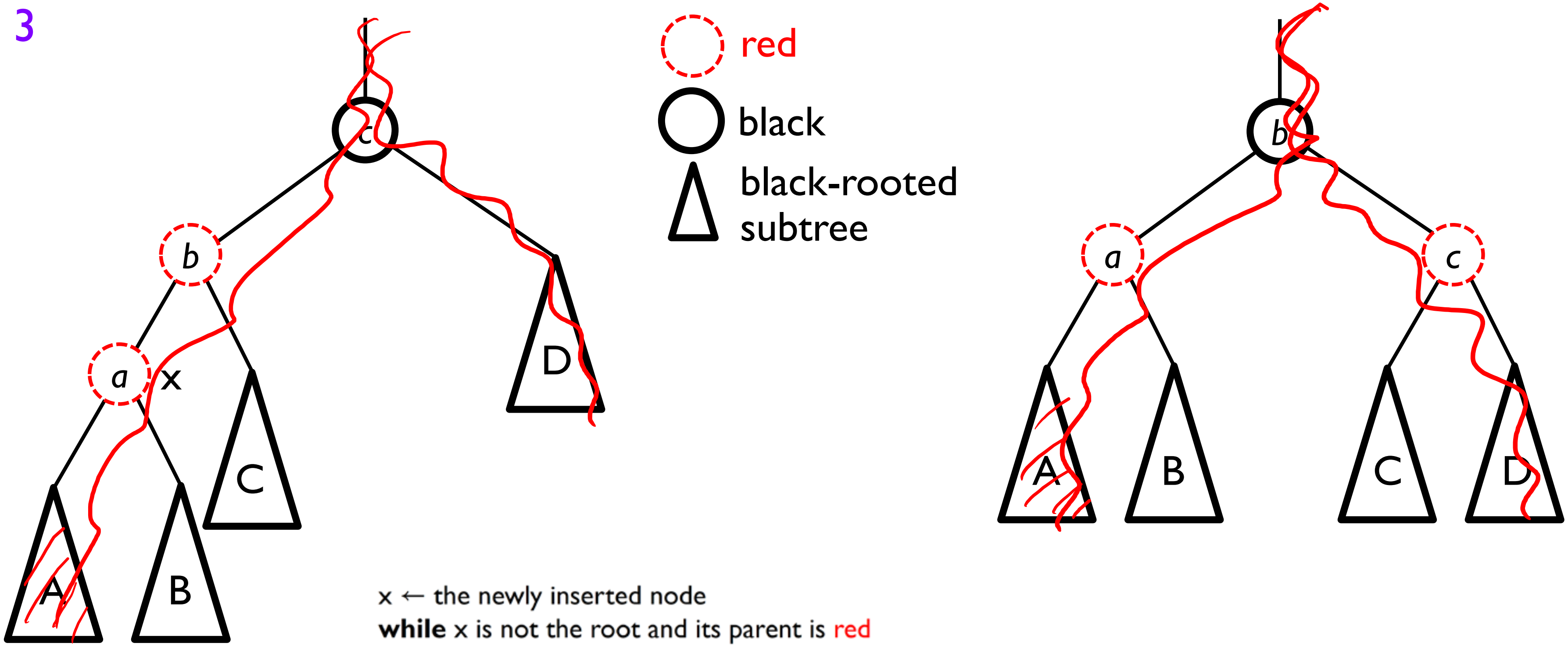


x ← the newly inserted node
while x is not the root and its parent is red
  (assume the parent is the left child of the grandparent)
Case 1
  if the "uncle" is red
    color the parent and the uncle black
    color the grandparent red; the grandparent becomes x
  else
Case 2
    if x is the right child of its parent
      left-rotate at the parent; the former parent becomes x
Case 3
    color the parent black
    color the grandparent red
    right-rotate at the grandparent
    x ← the root node
color the root black

(P1) Every node is colored either red or black
(P2) The root is black
(P3) Every leaf/null pointer is black
(P4) If a node is red, both its children are black
(P5) Every simple path from a node to a descendant leaf contains the same #black nodes

Insertion into a red-black tree

Case 3



x ← the newly inserted node
while x is not the root and its parent is red
  (assume the parent is the left child of the grandparent)
  Case 1 [**if** the "uncle" is red
    color the parent and the uncle black
    color the grandparent red; the grandparent becomes x
  else
  Case 2 [**if** x is the right child of its parent
      left-rotate at the parent; the former parent becomes x
  Case 3 [color the parent black
    color the grandparent red
    right-rotate at the grandparent
    x ← the root node
color the root black

(P1) Every node is colored either red or black
(P2) The root is black
(P3) Every leaf/null pointer is black
(P4) If a node is red, both its children are black
(P5) Every simple path from a node to a descendant leaf contains the same #black nodes

Insertion into a red-black tree

x ← the newly inserted node
while x is not the root and its parent is red
  (assume the parent is the left child of the grandparent)

Case 1
  if the "uncle" is red
    color the parent and the uncle black
    color the grandparent red; the grandparent becomes x
  else

Case 2
    if x is the right child of its parent
      left-rotate at the parent; the former parent becomes x

Case 3
    color the parent black
    color the grandparent red
    right-rotate at the grandparent
    x ← the root node
color the root black

(P1) Every node is colored either red or black
(P2) The root is black
(P3) Every leaf/null pointer is black
(P4) If a node is red, both its children are black
(P5) Every simple path from a node to a descendant leaf contains the same #black nodes

Insertion into a red-black tree

- Case 2 leads to Case 3
- Case 3 leads to termination
- Case 1 does not change the tree topology and x goes up

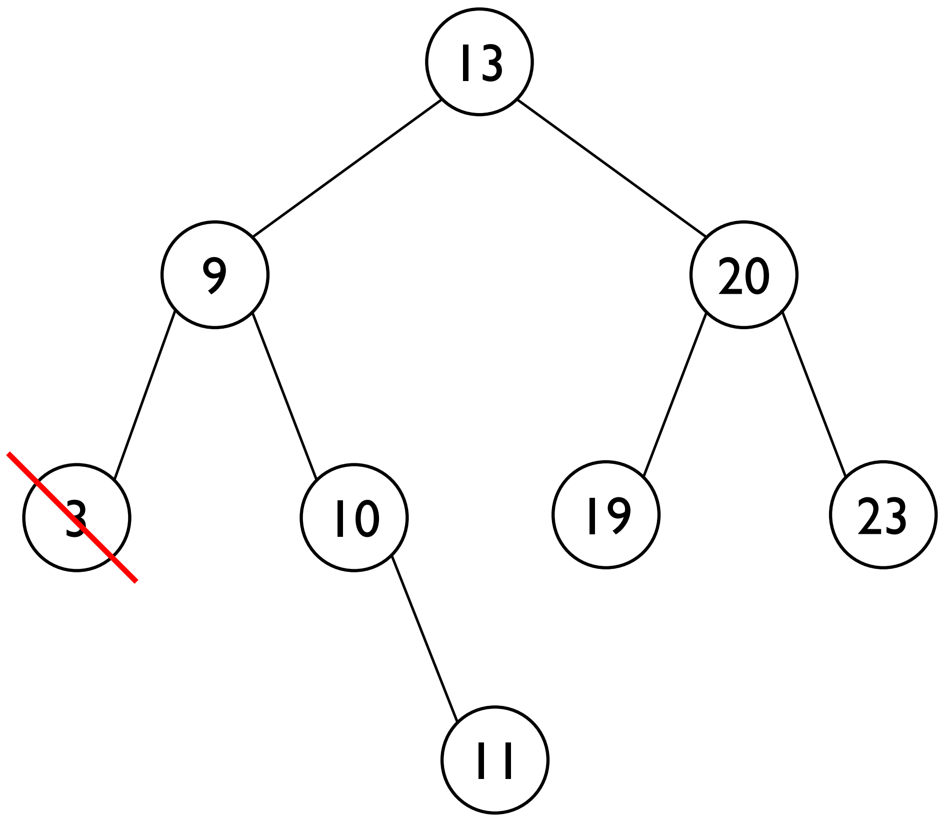
- $O(\log n)$ time

Deletion from a binary search tree

- Deleting a leaf

3:01

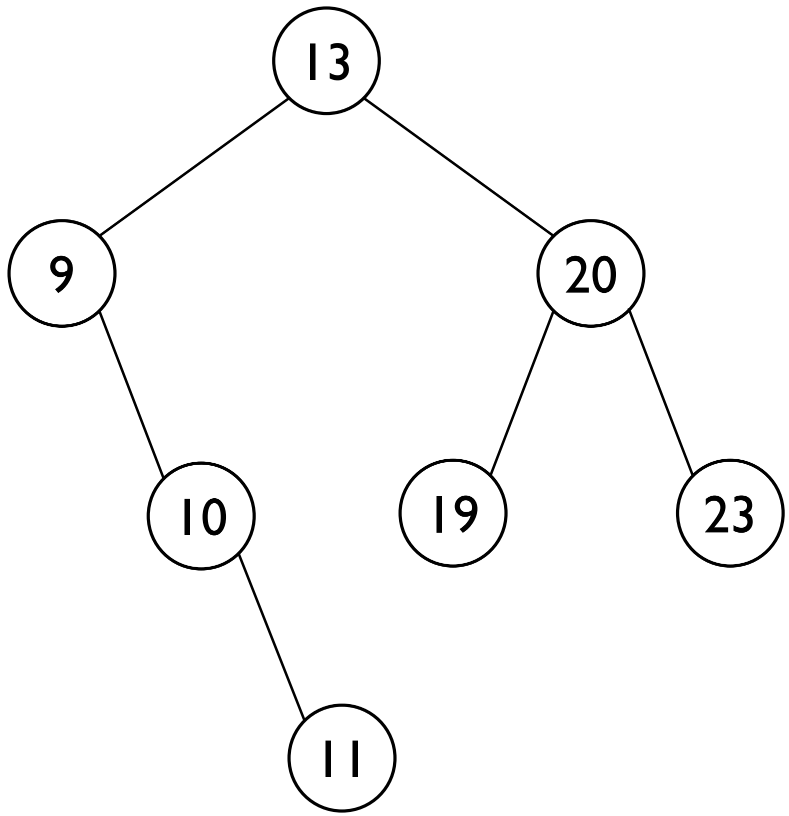
3



Deletion from a binary search tree

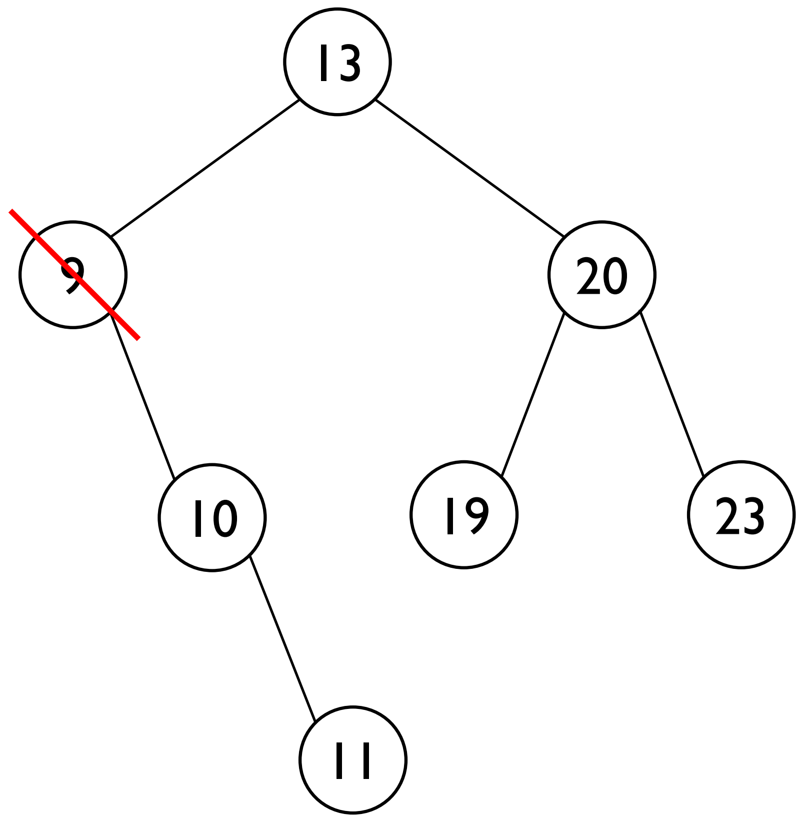
- Deleting a leaf

3



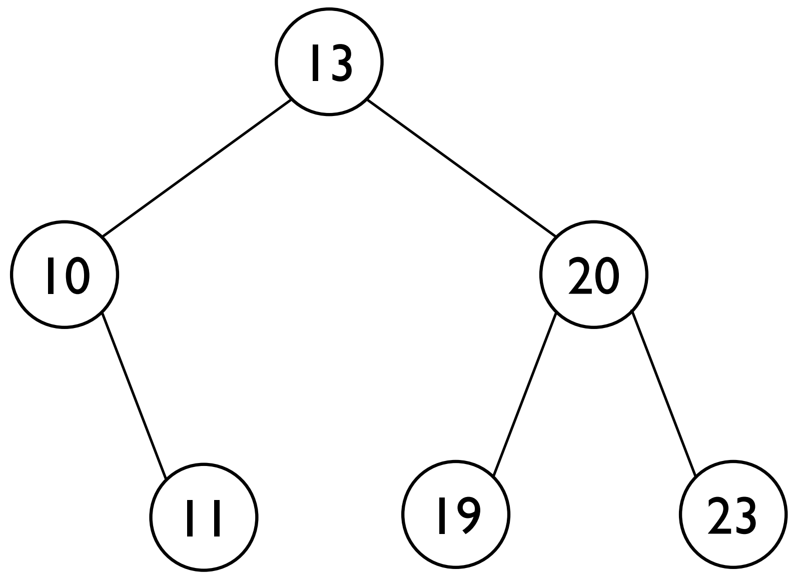
Deletion from a binary search tree

- Deleting a node with a single child



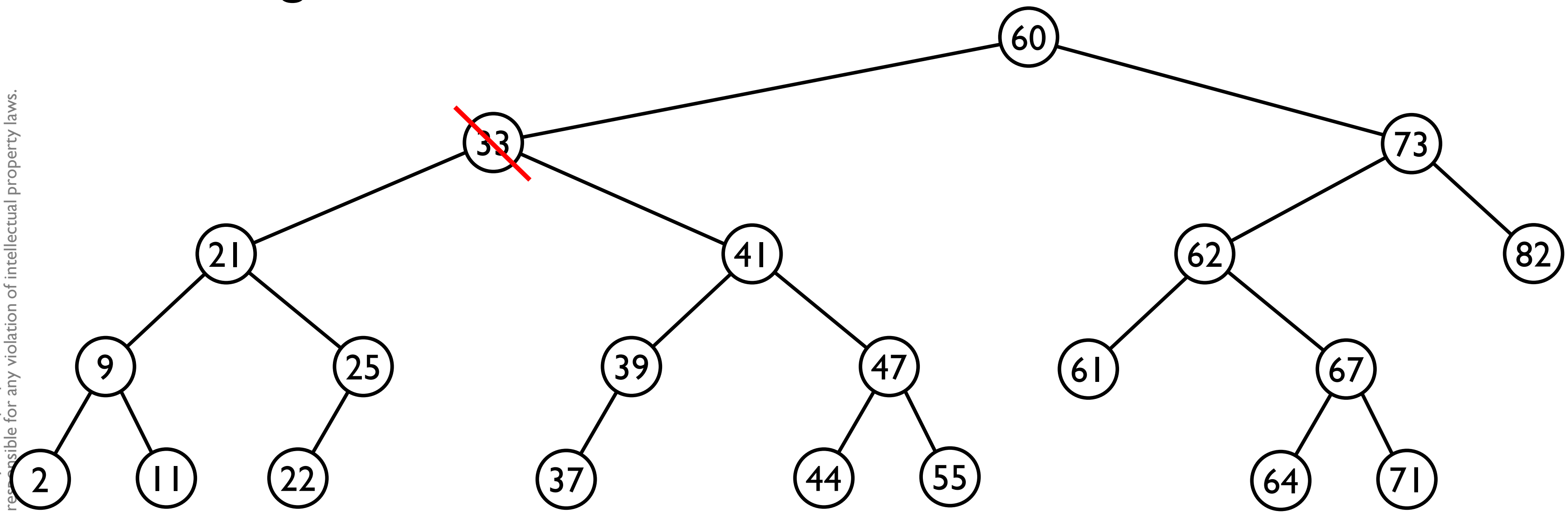
Deletion from a binary search tree

- Deleting a node with a single child



Deletion from a binary search tree

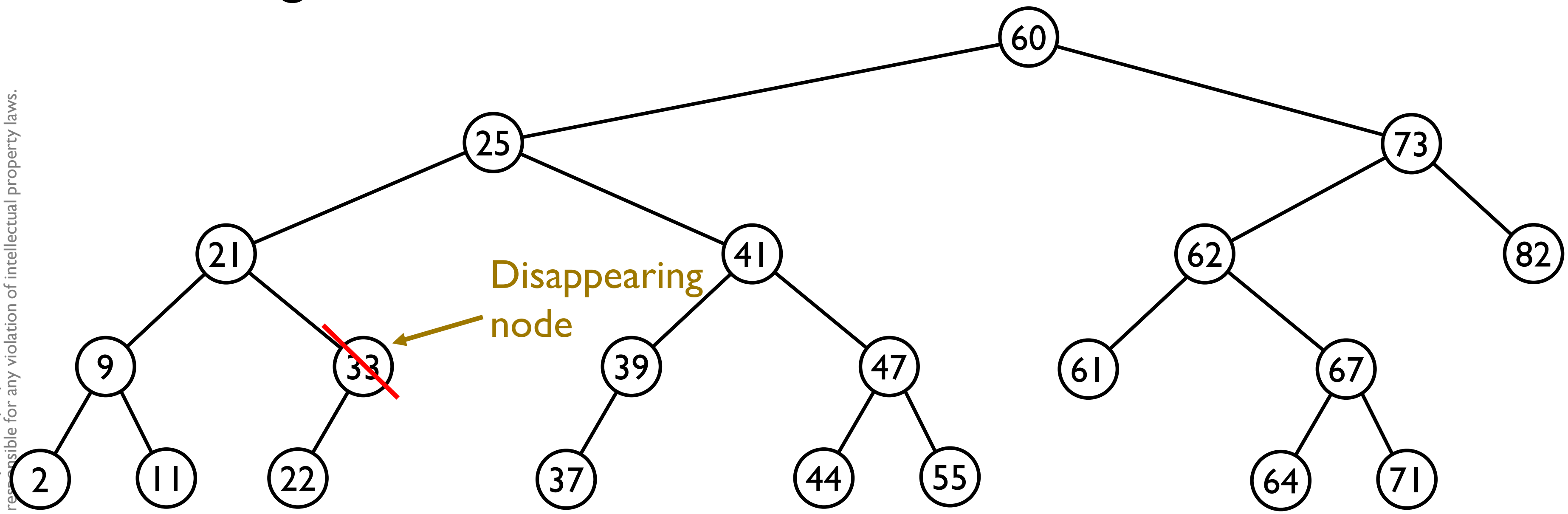
- Deleting a node with two children



- Swap with max of left subtree

Deletion from a binary search tree

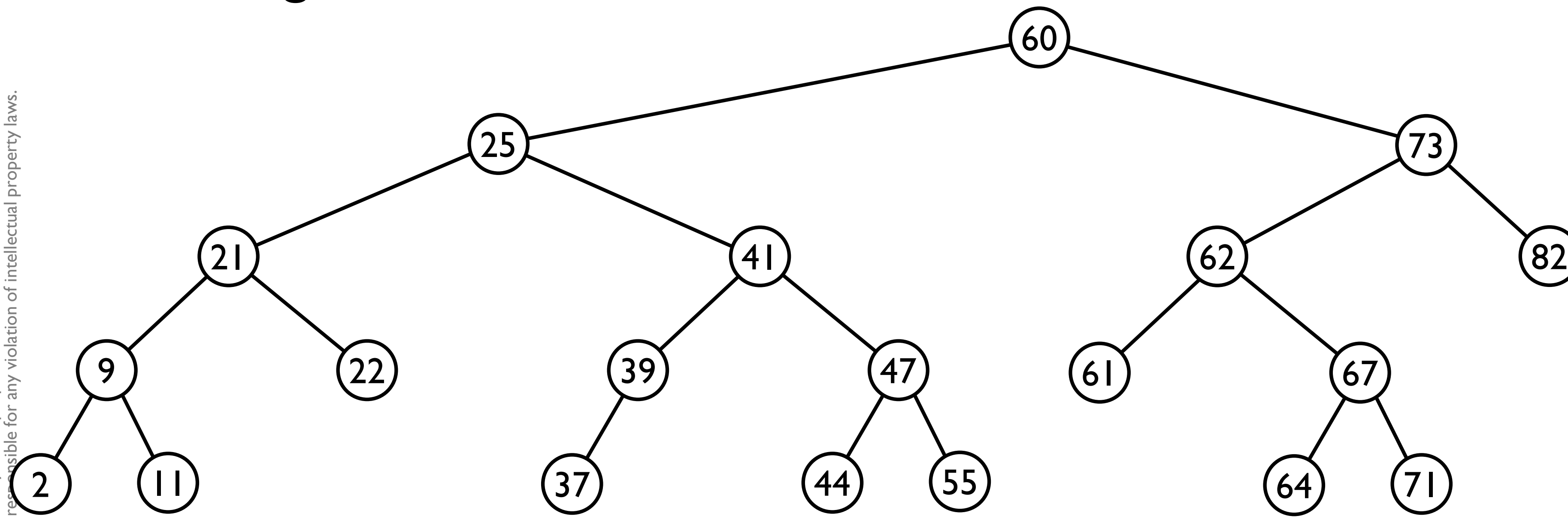
- Deleting a node with two children



- Swap with max of left subtree

Deletion from a binary search tree

- Deleting a node with two children



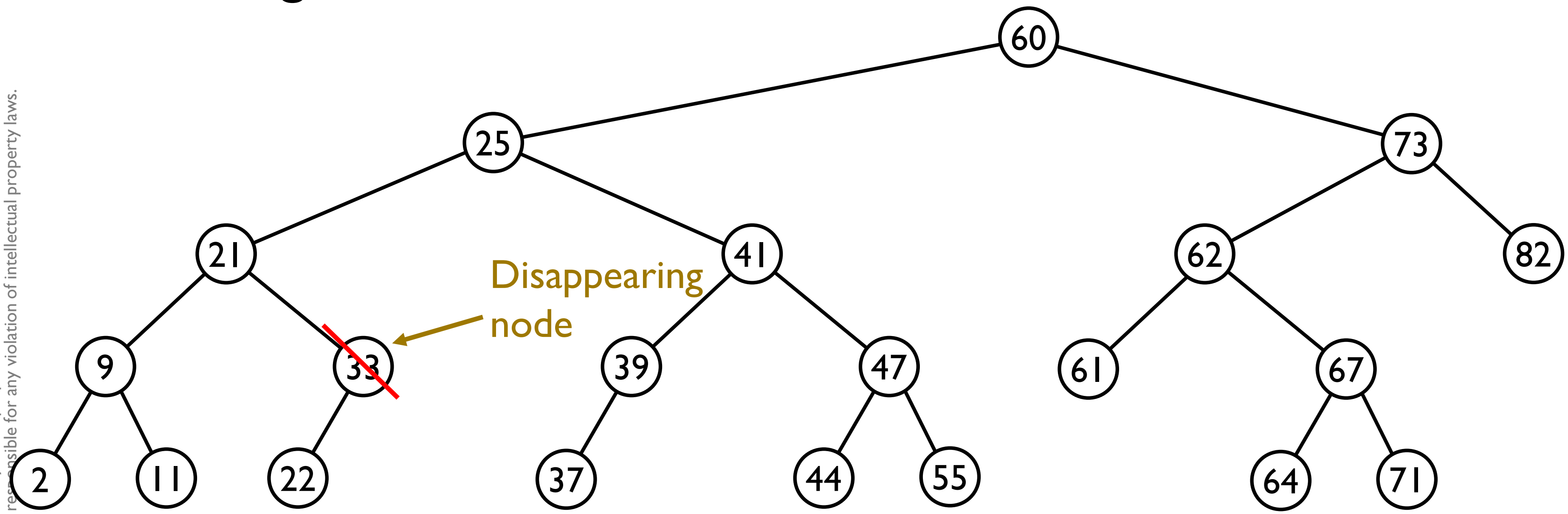
- Swap with max of left subtree

Deletion from a red-black tree

- Let x be the child of the disappearing node replacing it

Deletion from a binary search tree

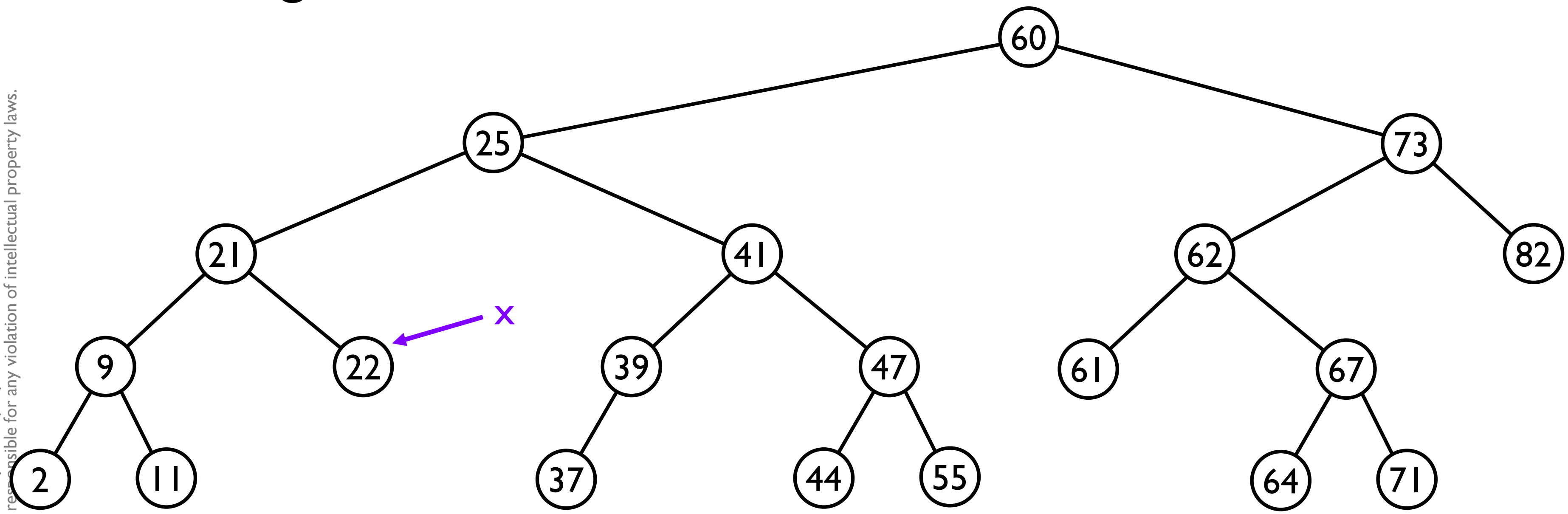
- Deleting a node with two children



- Swap with max of left subtree

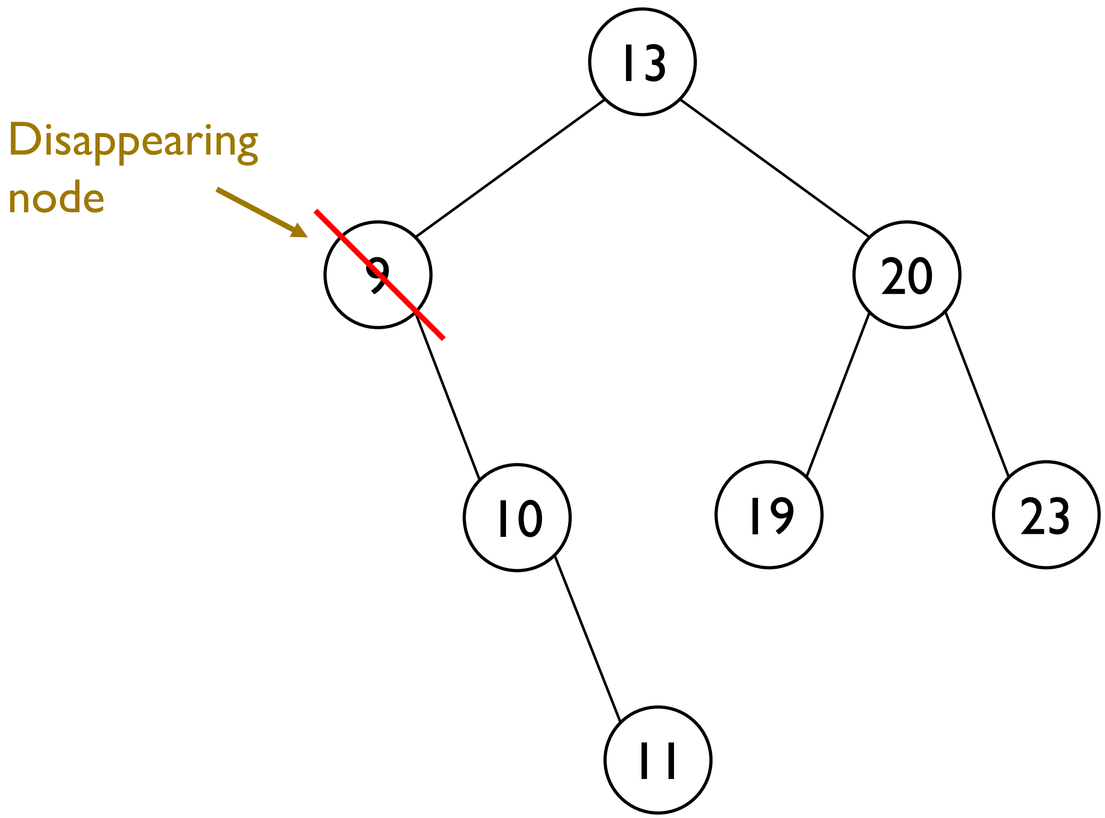
Deletion from a binary search tree

- Deleting a node with two children



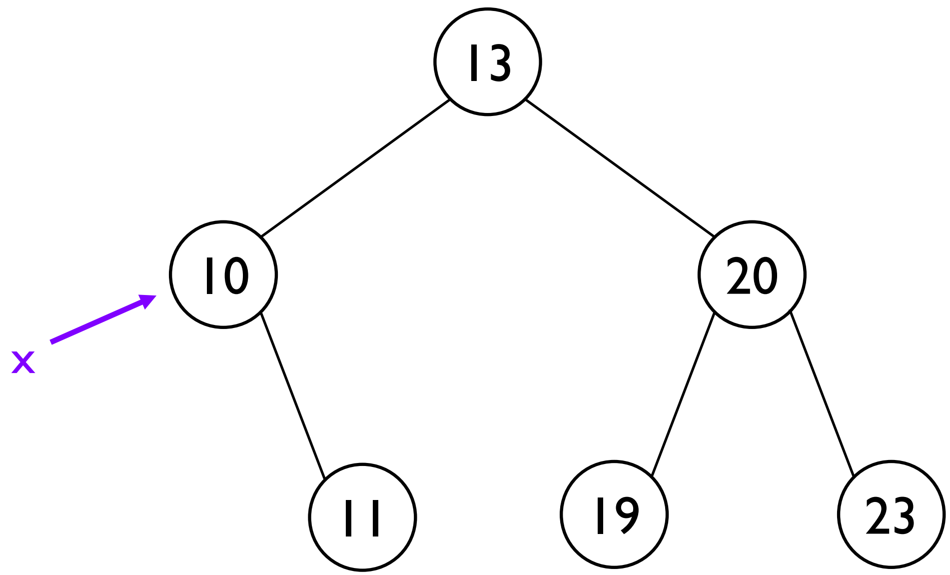
Deletion from a binary search tree

- Deleting a node with a single child



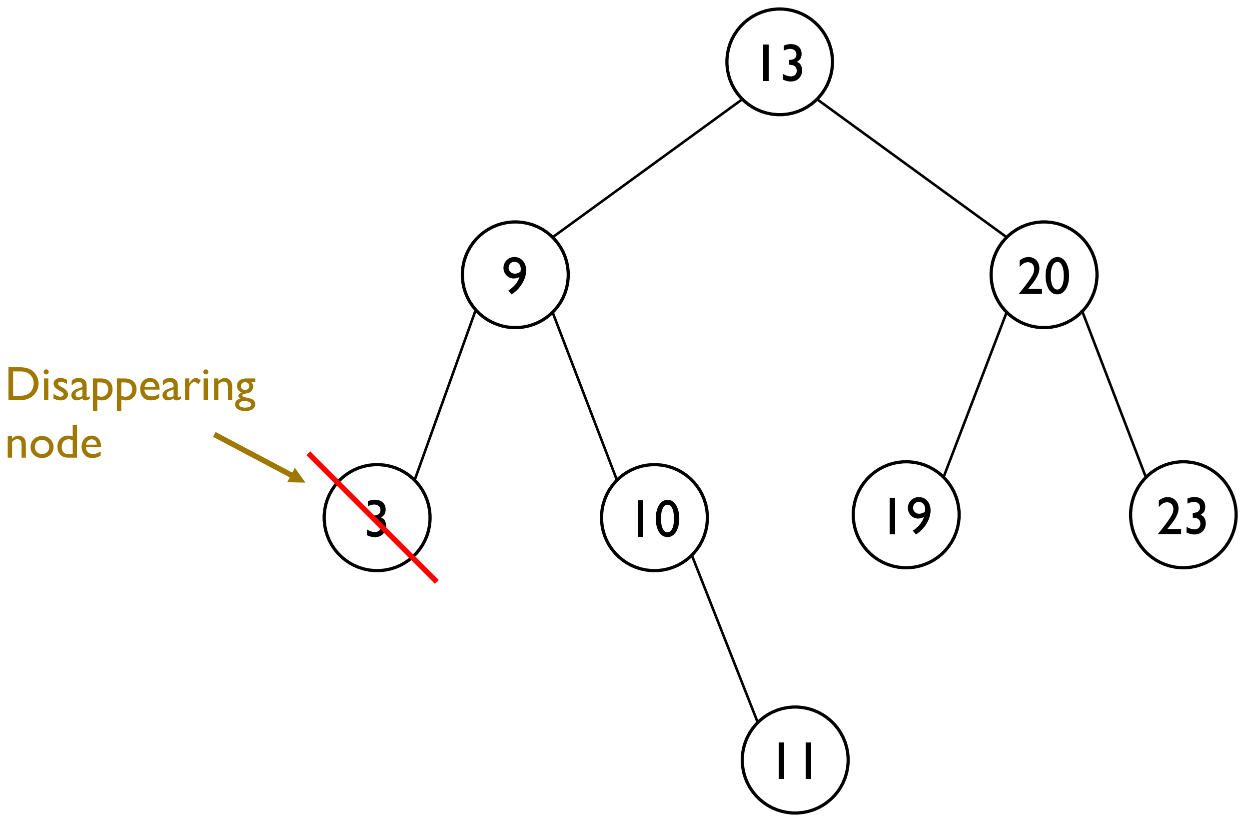
Deletion from a binary search tree

- Deleting a node with a single child



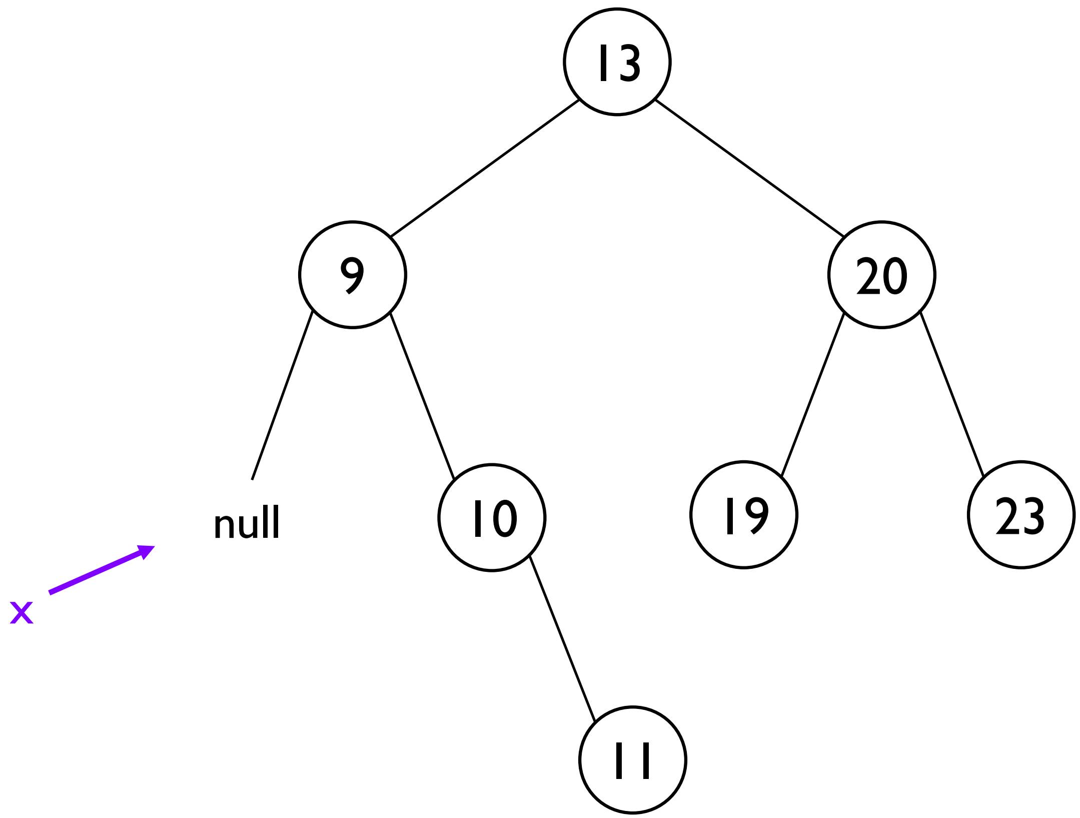
Deletion from a binary search tree

- Deleting a leaf



Deletion from a binary search tree

- Deleting a leaf



Deletion from a red-black tree

- Let x be the child of the disappearing node replacing it
- If the disappearing node is red, done
- O/w if we could place an "extra black" on x we would be done

- Delete in the usual way
- A **while** loop where x is the node with extra black

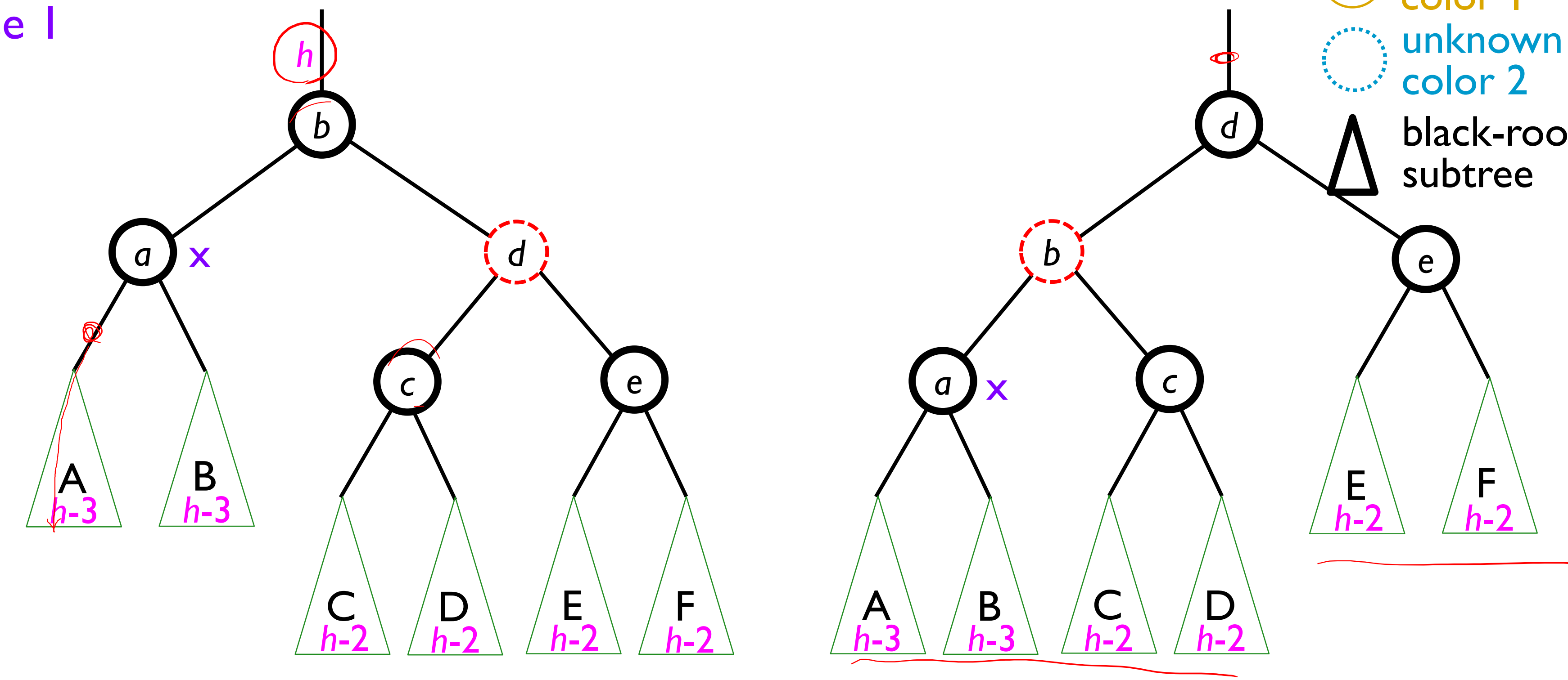
Deletion from a red-black tree

x ← the node with an "extra black"
while x is not the root and its color is black
  (assume that x is the left child of its parent)
  if the sibling is red
    Case 1:
    color the sibling black
    color the parent red
    left-rotate at the parent
  else
    if both "nephews" are black
      Case 2:
      color the sibling red
      the parent becomes x
    else
      if only the "right nephew" is black
        Case 3:
        color the "left nephew" black
        color the sibling red
        right-rotate at the sibling
      Case 4:
      swap the colors of the sibling and the parent
      color the "right nephew" black
      left-rotate at the parent
      the root becomes x
color x black

(P1) Every node is colored either red or black
(P2) The root is black
(P3) Every leaf/null pointer is black
(P4) If a node is red, both its children are black
(P5) Every simple path from a node to a descendant leaf contains the same #black nodes

Deletion from a red-black tree

Case 1

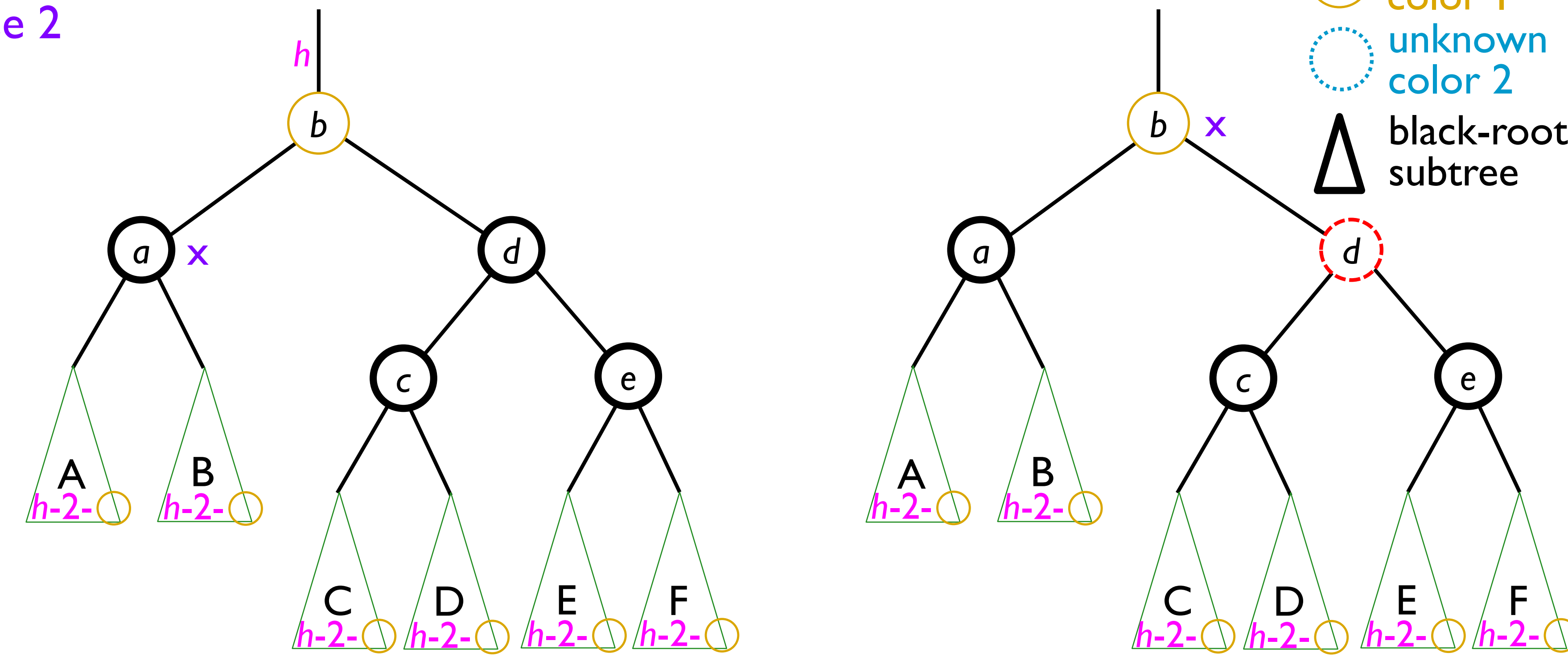



```
x ← the node with an "extra black"
while x is not the root and its color is black
  (assume that x is the left child of its parent)
  if the sibling is red
Case 1 {  color the sibling black
          color the parent red
          left-rotate at the parent
  else
    if both "nephews" are black
Case 2 {    color the sibling red
            the parent becomes x
    else
      if only the "right nephew" is black
Case 3 {      color the "left nephew" black
              color the sibling red
              right-rotate at the sibling
Case 4 {    swap the colors of the sibling and the parent
            color the "right nephew" black
            left-rotate at the parent
            the root becomes x
color x black
```

(P1) Every node is colored either red or black
(P2) The root is black
(P3) Every leaf/null pointer is black
(P4) If a node is red, both its children are black
(P5) Every simple path from a node to a descendant leaf contains the same #black node

Deletion from a red-black tree

Case 2

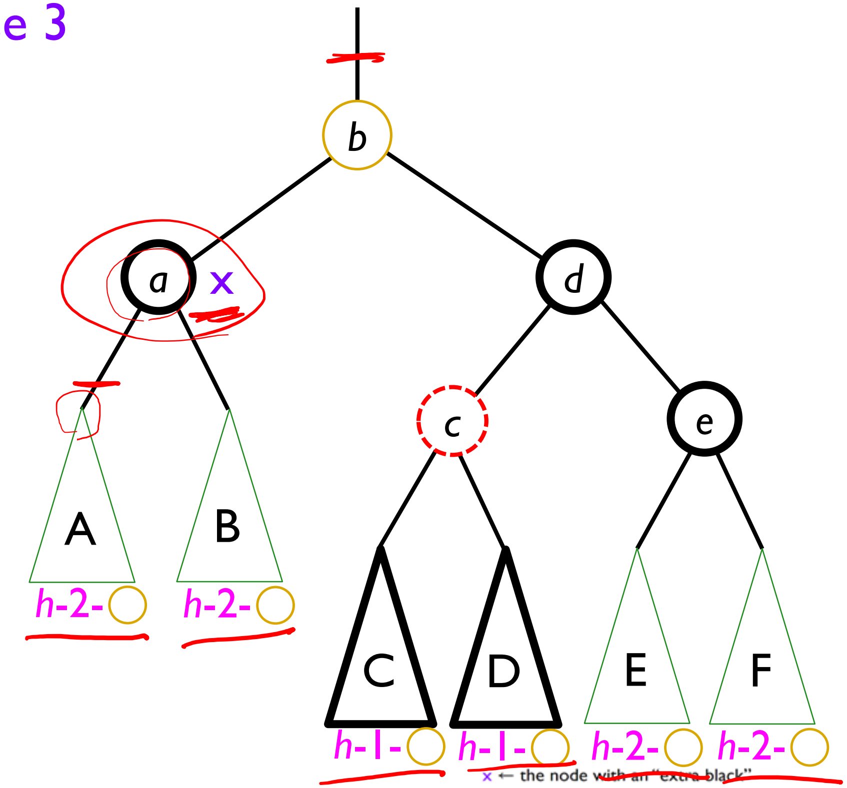


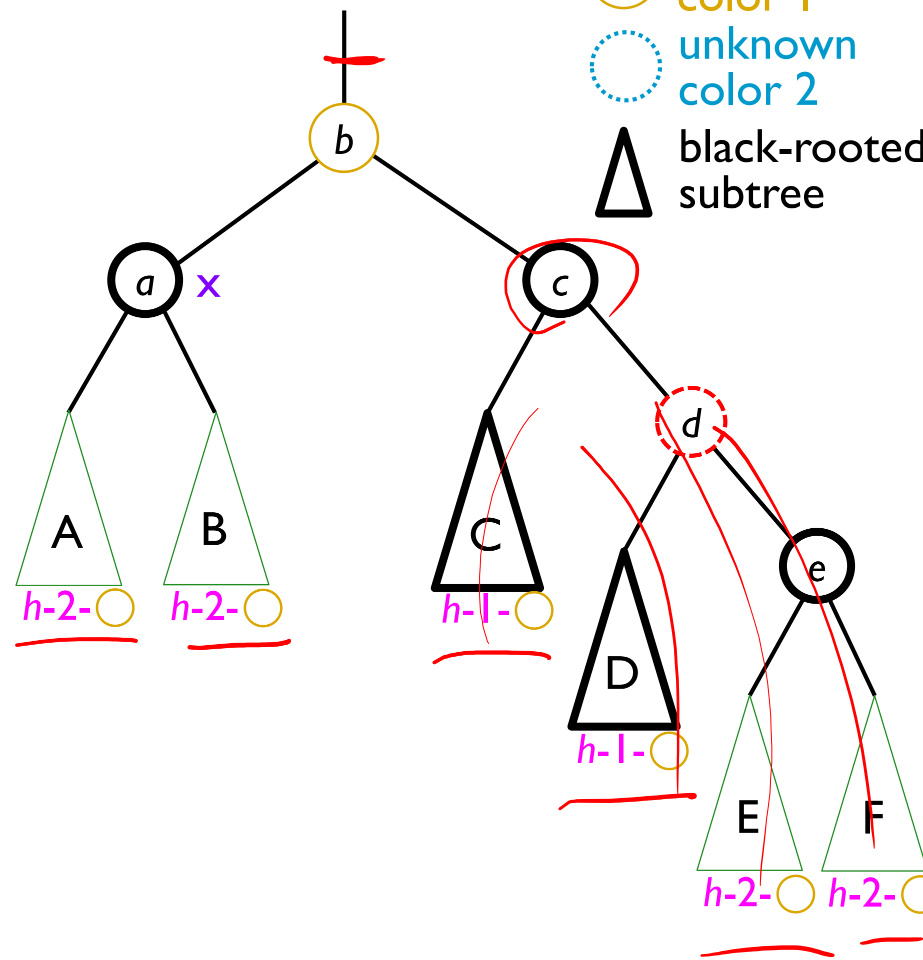
x ← the node with an "extra black"
while x is not the root and its color is black
  (assume that x is the left child of its parent)
  if the sibling is red
    Case 1: color the sibling black
    color the parent red
    left-rotate at the parent
  else
    if both "nephews" are black
      Case 2: color the sibling red
      the parent becomes x
    else
      if only the "right nephew" is black
        Case 3: color the "left nephew" black
        color the sibling red
        right-rotate at the sibling
      Case 4: swap the colors of the sibling and the parent
      color the "right nephew" black
      left-rotate at the parent
      the root becomes x
color x black

(P1) Every node is colored either red or black
(P2) The root is black
(P3) Every leaf/null pointer is black
(P4) If a node is red, both its children are black
(P5) Every simple path from a node to a descendant leaf contains the same #black node

Deletion from a red-black tree

Case 3



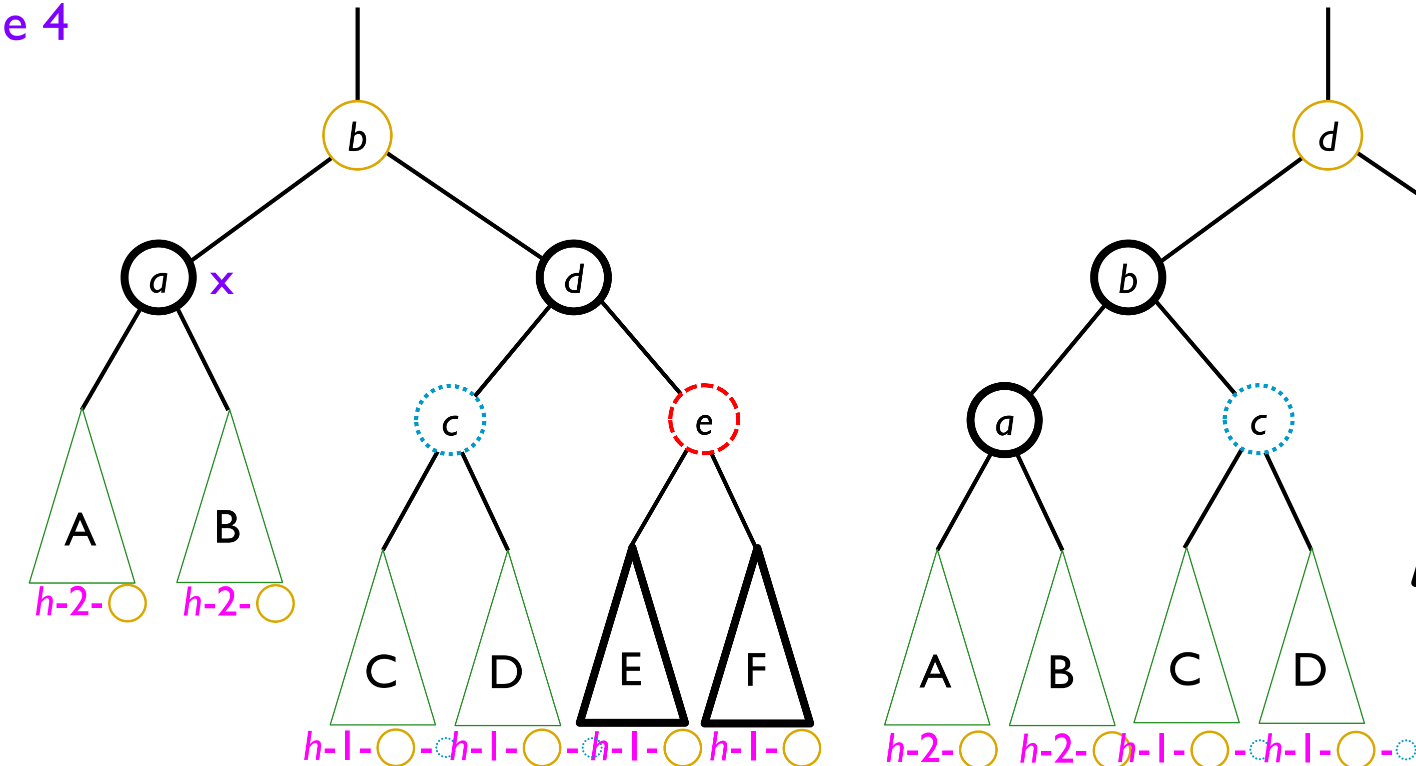


x ← the node with an "extra black"
while x is not the root and its color is black
  (assume that x is the left child of its parent)
  if the sibling is red
    Case 1 { color the sibling black
    color the parent red
    left-rotate at the parent }
  else
    if both "nephews" are black
      Case 2 { color the sibling red
      the parent becomes x }
    else
      if only the "right nephew" is black
        Case 3 { color the "left nephew" black
        color the sibling red
        right-rotate at the sibling }
      Case 4 { swap the colors of the sibling and the parent
      color the "right nephew" black
      left-rotate at the parent
      the root becomes x }
color x black

(P1) Every node is colored either red or black
(P2) The root is black
(P3) Every leaf/null pointer is black
(P4) If a node is red, both its children are black
(P5) Every simple path from a node to a descendant leaf contains the same #black node

Deletion from a red-black tree

Case 4



x ← the node with an "extra black"
while x is not the root and its color is black
(assume that x is the left child of its parent)
if the sibling is red
Case 1: color the sibling black
color the parent red
left-rotate at the parent
else
if both "nephews" are black
Case 2: color the sibling red
the parent becomes x
else
if only the "right nephew" is black
Case 3: color the "left nephew" black
color the sibling red
right-rotate at the sibling
Case 4: swap the colors of the sibling and the parent
color the "right nephew" black
left-rotate at the parent
the root becomes x
color x black

(P1) Every node is colored either red or black
(P2) The root is black
(P3) Every leaf/null pointer is black
(P4) If a node is red, both its children are black
(P5) Every simple path from a node to a descendant leaf contains the same #black node

Deletion from a red-black tree

- Case 3 leads to Case 4
- Case 4 leads to termination
- Case 2 does not change the tree topology and x goes up
  - If the parent is red, leads to termination
- Case 1 results in red parent

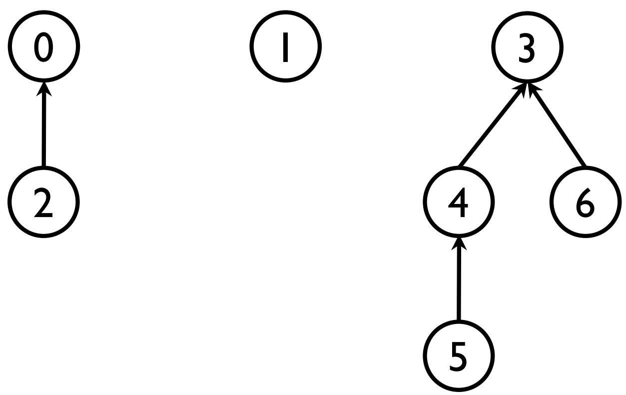
- $O(\log n)$ time

Data structures for disjoint sets

- Union-Find data structure
  - Maintains a family of disjoint subsets of {0, …, n-1}
  - Supports the following operations:
    - Initialization with *n* singletons
    - Union(i, j): unites S_i and S_j
    - Find(x): returns the (name of the) set that contains x

Tree representation

- {0, 2}, {1}, {3, 4, 5, 6}

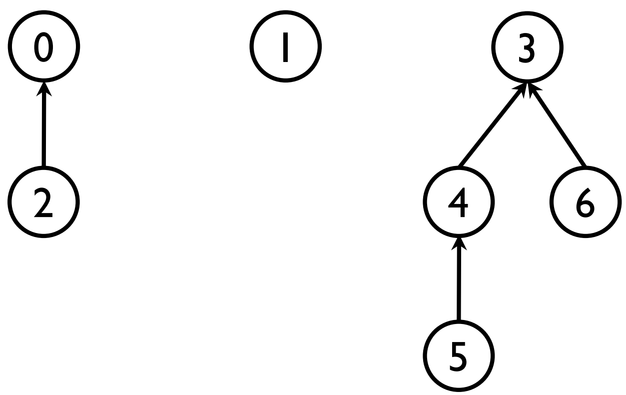


An improvement

- Weighted union
  - Make the smaller-cardinality tree a subtree

Tree representation

- {0, 2}, {1}, {3, 4, 5, 6}



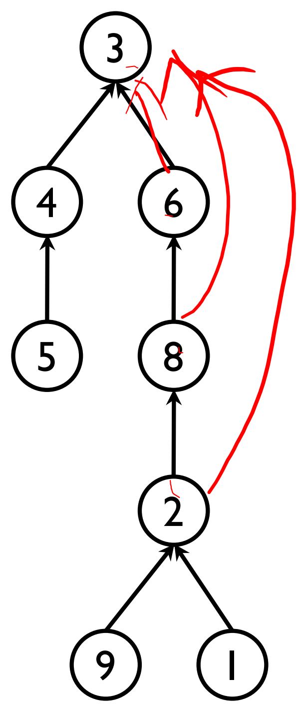
- Union
  - Make one tree a subtree of the other's root
  - $O(1)$ time*
- Find
  - Find the root
  - $O(\log n)$ time

parent[0]	parent[1]	parent[2]	parent[3]	parent[4]	parent[5]	parent[6]
-1	-1	0	-1	3	4	3
card[0]	card[1]	card[2]	card[3]	card[4]	card[5]	card[6]
2	1	1	4	2	1	1

Another improvement

- Path compression

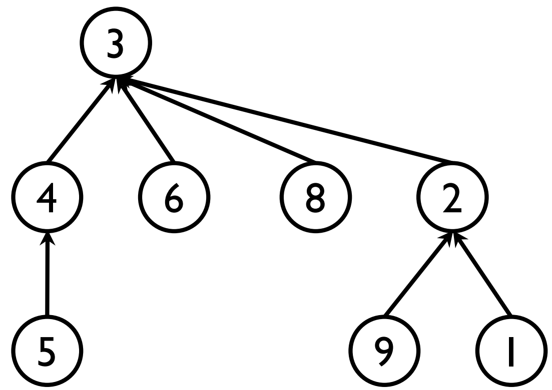
Find(2)



Another improvement

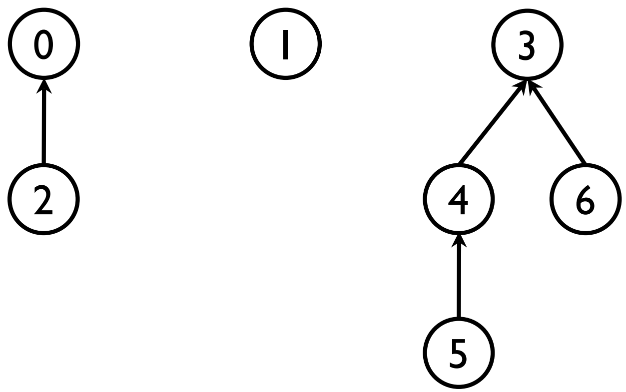
- Path compression

Find(2)



Tree representation

- {0, 2}, {1}, {3, 4, 5, 6}

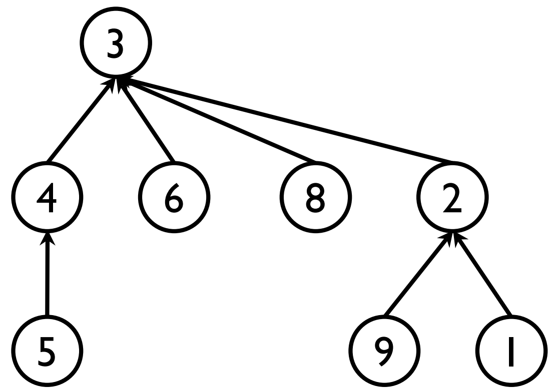


- Union
  - Make one tree a subtree of the other's root
  - $O(1)$ time*
- Find
  - Find the root
  - $O(\log n)$ time

parent[0]	parent[1]	parent[2]	parent[3]	parent[4]	parent[5]	parent[6]
-1	-1	0	-1	3	4	3
card[0]	card[1]	card[2]	card[3]	card[4]	card[5]	card[6]
2	1	1	4	2	1	1

Amortized analysis

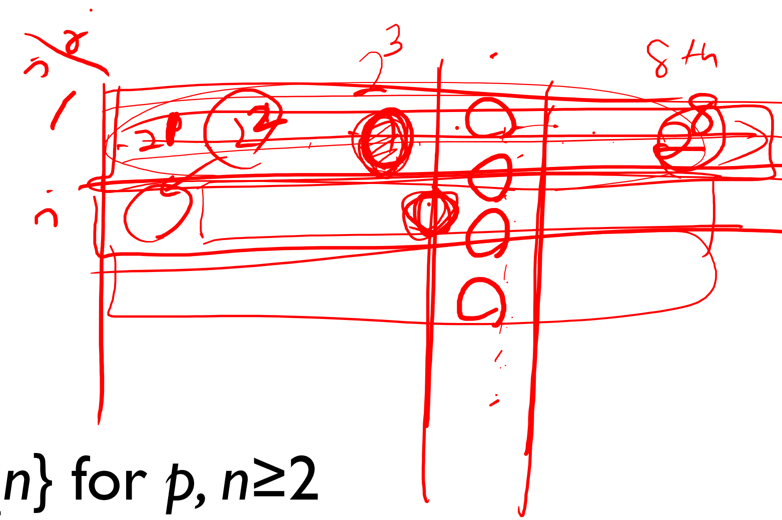
- Not all find operations can take that much time…



- Amortized analysis
  - Analyze the "average time" required to perform an operation on a given data structure
  - But we do not know the exact sequence of operations in advance…

Amortized analysis

- Any mixed sequence of f finds and u unions ($u \geq n/2$) takes $O(n+f\alpha(f+n, n))$ time, where α is the inverse Ackermann's function



- Ackermann's function
  - $A(1, j) = 2^j$ for $j \geq 1$
  - $A(i, 1) = A(i-1, 2)$ for $i \geq 2$
  - $A(i, j) = A(i-1, A(i, j-1))$ for $i, j \geq 2$
  - $\alpha(p, n) = \min\{z \geq 1 \mid A(z, \lfloor p/n \rfloor) > \log_2 n\}$ for $p, n \geq 2$

  - $\alpha(f+n, n)$ is therefore ≤ 4 for all practical purposes