CS 457, Fall 2016

Drexel University, Department of Computer Science

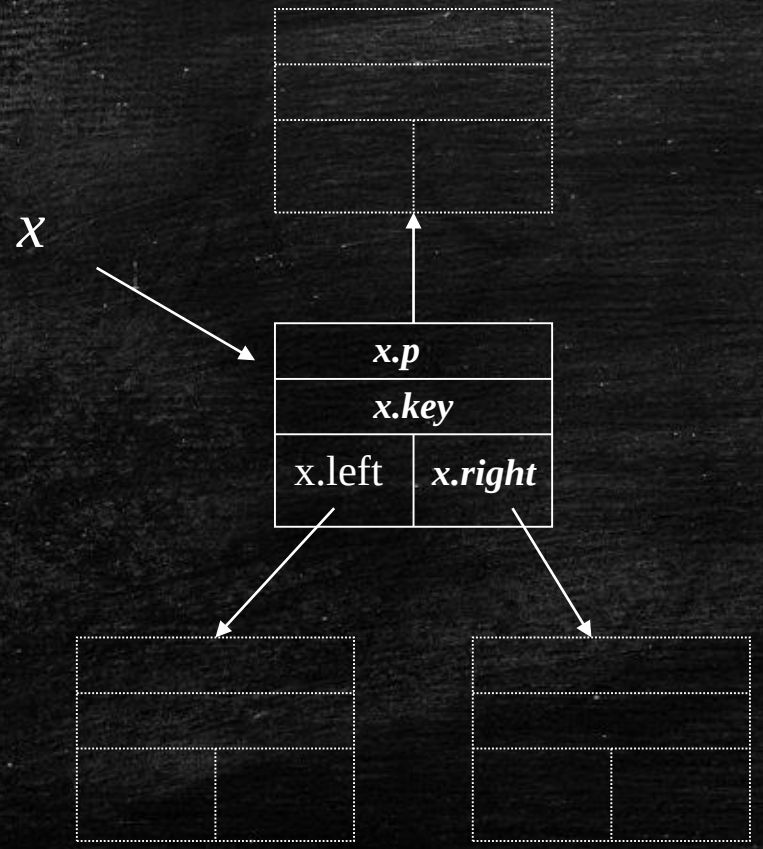
Lecture 9

Today's Lecture

- Binary Search Trees
- Red-Black Trees

Binary Search Trees

- Each node ***x*** in a binary search tree (BST) contains:



- ***x.key*** - The value stored at *x*.
- ***x.left*** - Pointer to left child of ***x.***
- ***x.right*** - Pointer to right child of ***x.***
- ***x.p*** - Pointer to parent of ***x.***

Binary Search Tree Property

- Keys in BST satisfy the following properties:

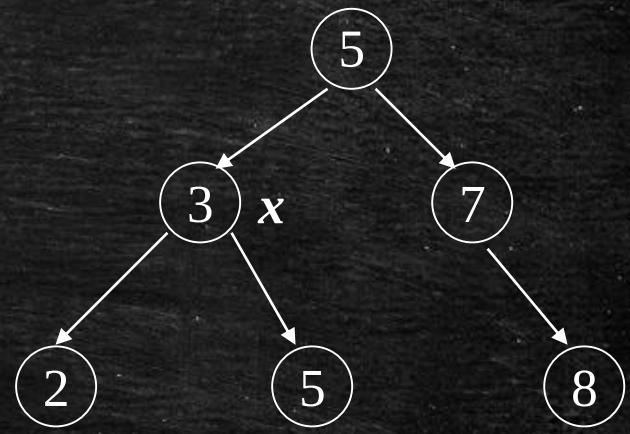
  Let ***x*** be a node in a BST:

  - If ***y*** is in the left subtree of ***x*** then:

    y.*key* <= **x.*key***

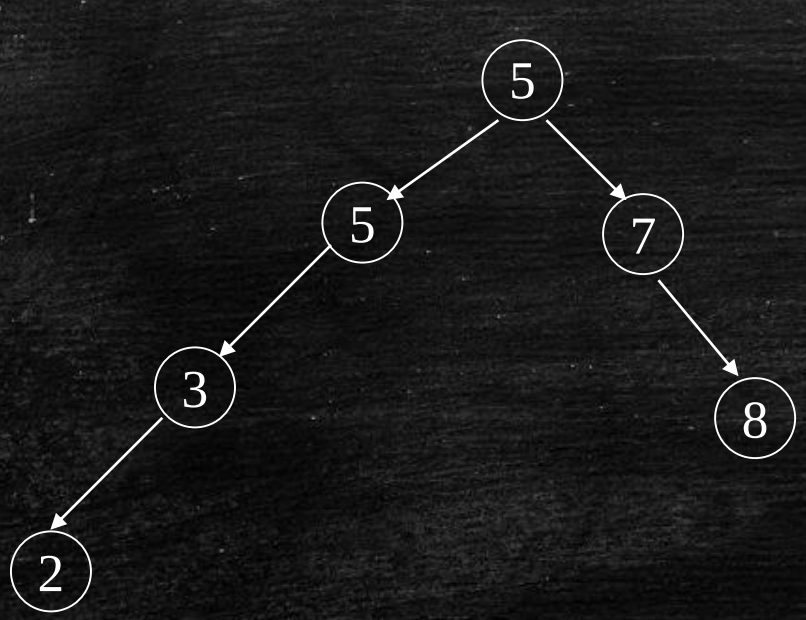
  - If ***y*** is in the right subtree of ***x*** then:

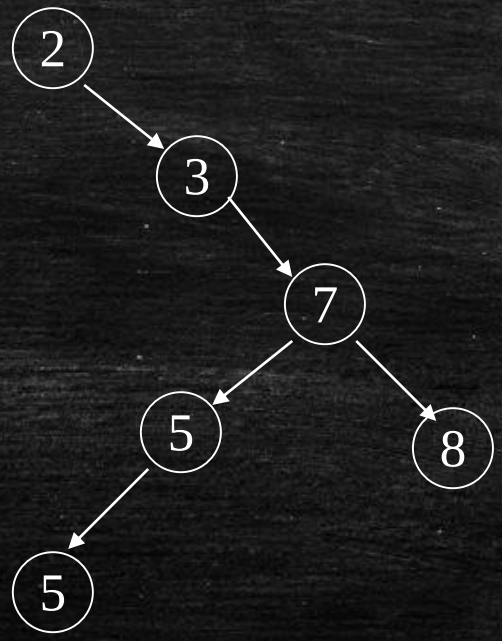
    y.*key* > ***x.key***



Binary Search Tree Examples

- Two valid BST's for the keys: 2,3,5,5,7,8.





In-Order Tree walk

- Can print keys in BST with in-order tree walk
- Key of each node printed between keys in left and those in right subtrees
- Prints elements in monotonically increasing order

In-Order Traversal

Inorder-Tree-Walk(*x*)

1: If ***x*** ≠ ***NIL*** then

2: Inorder-Tree-Walk(***x.left***)

3: Print(***x.key***)

4: Inorder-Tree-Walk(***x.right***)

- What is the recurrence relation for T(n)?
- What is the running time?

In-Order Traversal (Running Time)

- The running time is $T(n) = T(k) + T(n-k-1) + \Theta(1)$

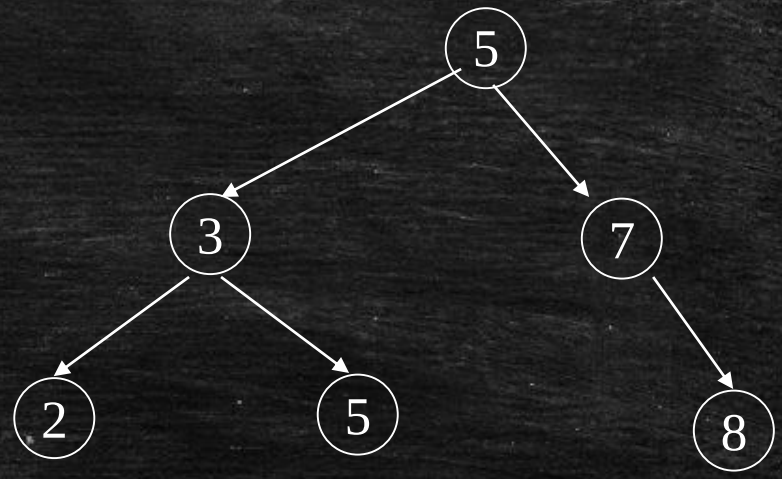
- Guess that $T(n) \in \Theta(n)$
  - We already know that $T(n) \geq n$ since we visit every node
  - It takes constant time on an single-node subtree, so $T(1) = a$
  - Assume that $T(n') \geq cn'$ for some constant c and all $n' < n$.

$$\begin{aligned} T(n) &\leq ck + c(n-k-1) + b \\ &= cn - c + b \\ &\leq cn \end{aligned}$$

For $c \geq b$

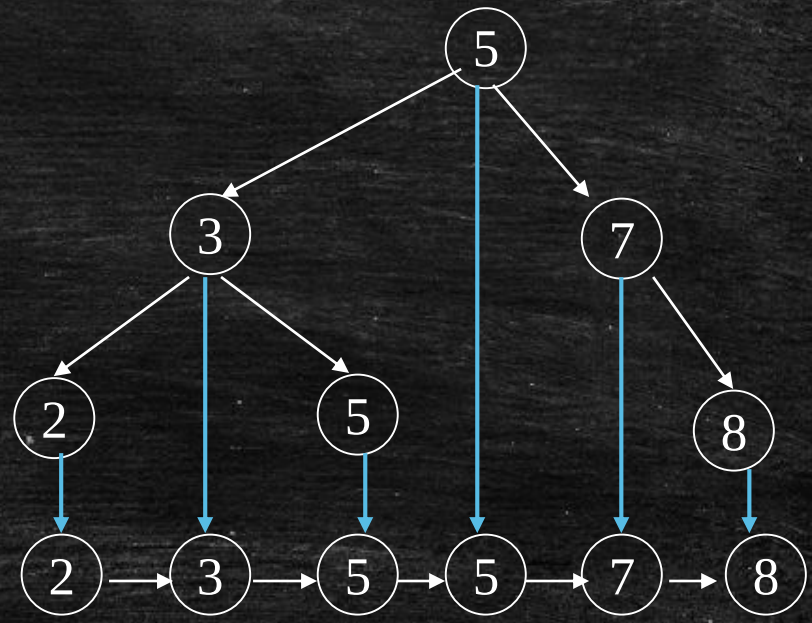
In-Order Traversal

- In-Order traversal can be thought of as a projection of BST nodes on an interval.



In-Order Traversal

- In-Order traversal can be thought of as a projection of BST nodes on an interval.



Other Tree Walks

Preorder-Tree-Walk(x)
1: If $x \neq NIL$ then
2: Print($key[x]$)
3: Preorder-Tree-Walk($left[x]$)
4: Preorder-Tree-Walk($right[x]$)

Postorder-Tree-Walk(x)
1: If $x \neq NIL$ then
2: Postorder-Tree-Walk($left[x]$)
3: Postorder-Tree-Walk($right[x]$)
4: Print($key[x]$)

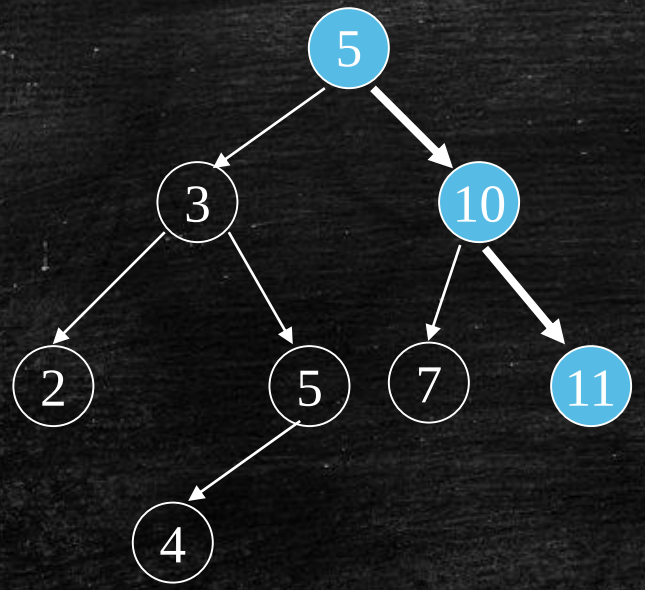
Searching in BST

- To find element with key ***k*** in tree ***T***:
  - Compare ***k*** with the root
  - If ***k***<***key***[**root**[***T***]] search for ***k*** in left subtree
  - Otherwise, search for ***k*** in right subtree

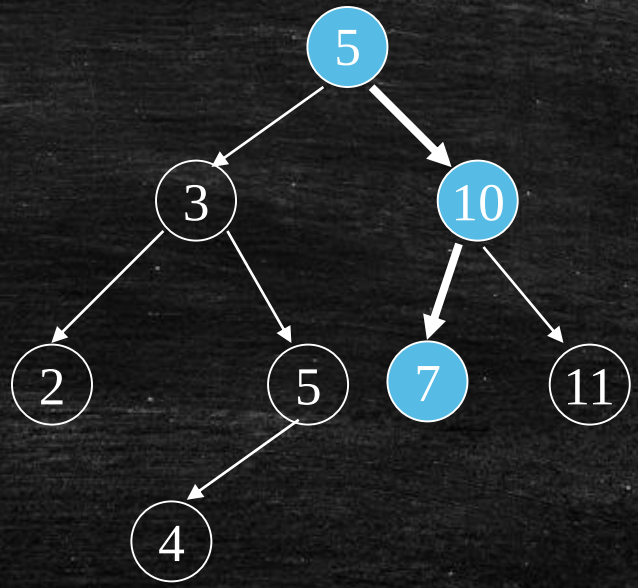
```
Search(T,k)
    x = root[T]
    if x == NIL
        return("not found")
    if k == key[x]
        return("found the key")
    if k < key[x]
        Search(left[x],k)
    else
        Search(right[x],k)
```

Examples:

- Search(*T*,11)

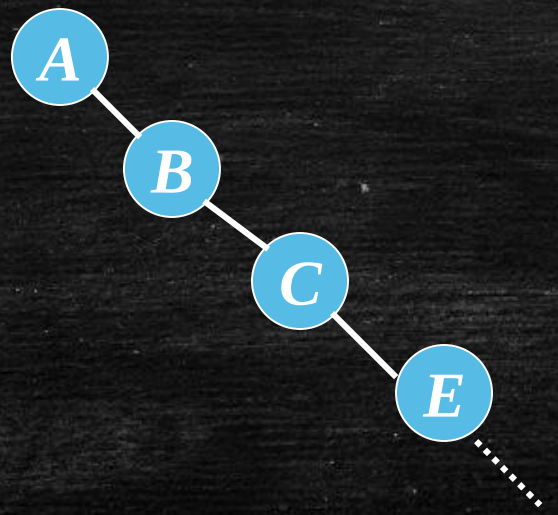


- Search(*T*,6)



Analysis of Search

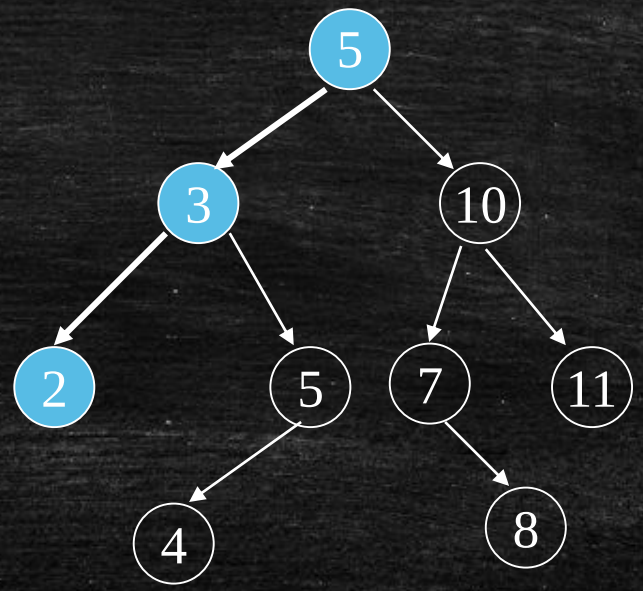
- Running time of search for a tree of height ***h*** is $O(h)$
- After insertion of ***n*** keys, worst case running time of search is $O(n)$



Locating the Minimum

Tree-Minimum(***x***)

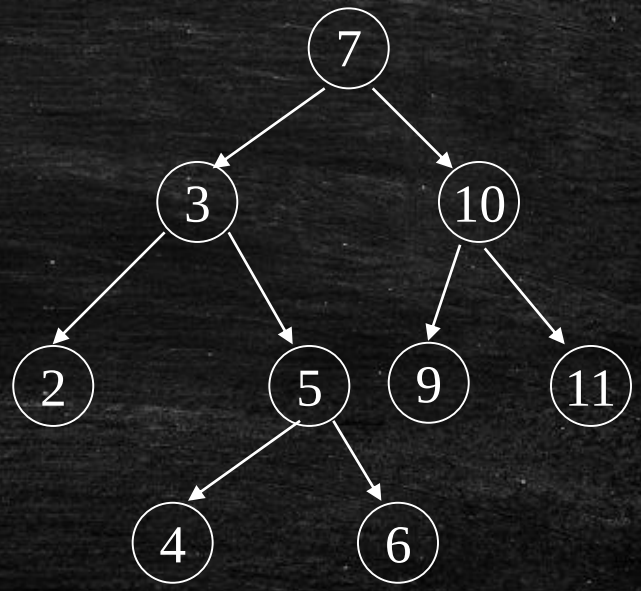
1. while ***left***[***x***]≠***NIL*** do
2. ***x***=***left***[***x***]
3. return ***x***



Successor / Predecessor

Successor of a node *x* : the node with smallest key greater than *x.key*

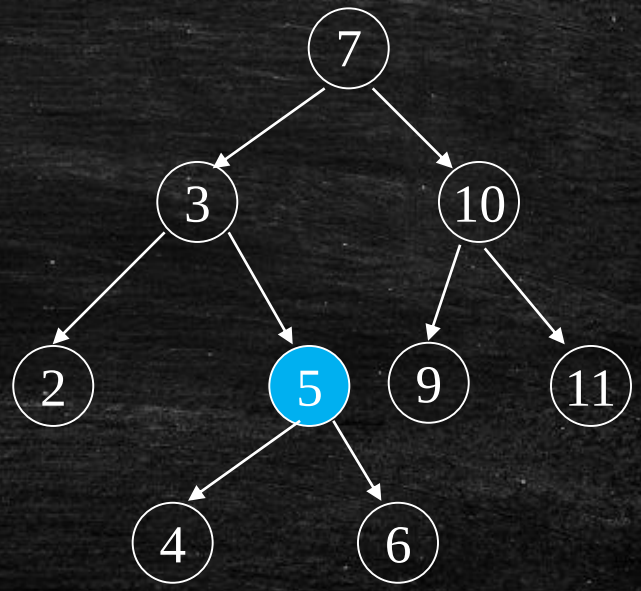
```
Tree-Successor(x)
    if x.right ≠ NIL
        return Tree-Minimum(x.right)
    y=x.p
    while y ≠ NIL and x==y.right
        x=y
        y=y.p
    return y
```



Successor / Predecessor

Successor of a node *x* : the node with smallest key greater than *x.key*

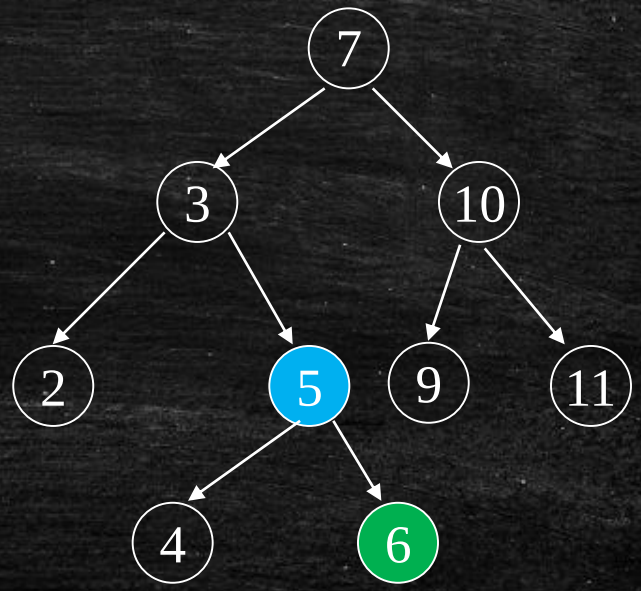
```
Tree-Successor(x)
    if x.right ≠ NIL
        return Tree-Minimum(x.right)
    y=x.p
    while y ≠ NIL and x==y.right
        x=y
        y=y.p
    return y
```



Successor / Predecessor

Successor of a node *x* : the node with smallest key greater than *x.key*

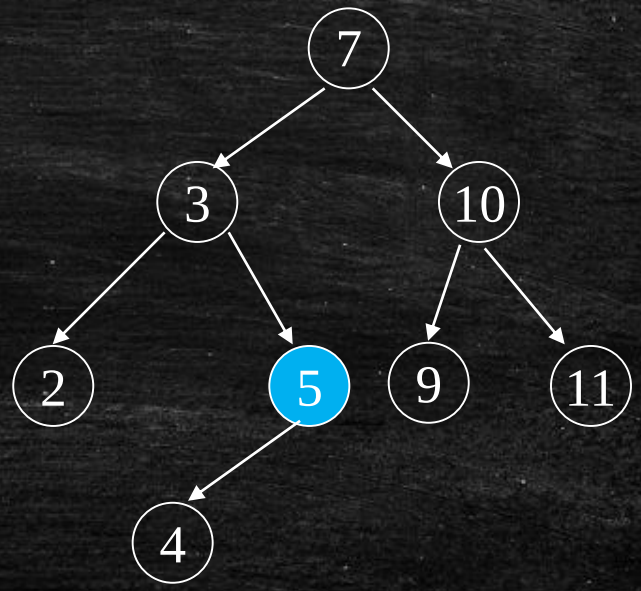
```
Tree-Successor(x)
    if x.right ≠ NIL
        return Tree-Minimum(x.right)
    y=x.p
    while y ≠ NIL and x==y.right
        x=y
        y=y.p
    return y
```



Successor / Predecessor

Successor of a node *x* : the node with smallest key greater than *x.key*

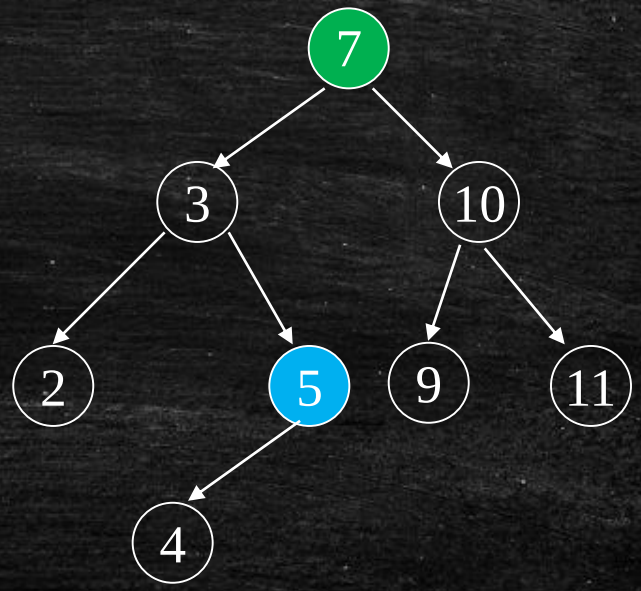
```
Tree-Successor(x)
    if x.right ≠ NIL
        return Tree-Minimum(x.right)
    y=x.p
    while y ≠ NIL and x==y.right
        x=y
        y=y.p
    return y
```



Successor / Predecessor

Successor of a node *x* : the node with smallest key greater than *x.key*

```
Tree-Successor(x)
    if x.right ≠ NIL
        return Tree-Minimum(x.right)
    y=x.p
    while y ≠ NIL and x==y.right
        x=y
        y=y.p
    return y
```



BST Insertion

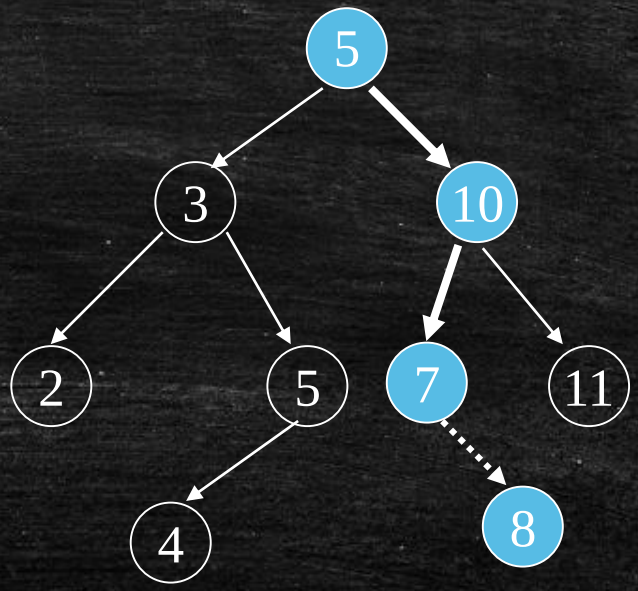
- Basic idea: similar to search.
- BST-Insert:
  - Take an element ***z*** (whose right and left children are NIL) and insert it into ***T***.
  - Find a place where ***z*** belongs, using code similar to that of Search.
  - Add ***z*** there.

Insert Key

BST-Insert(*T*,*z*)

```
y = NIL
x = T.root
while x ≠ NIL
    y = x;
    if key[z]<key[x]
        x = left[x]
    else  x = right[x]
z.p = y
if y == NIL
    T.root = z
else if key[z]<key[y]
    left[y] = z
else  right[y] = z
```

Insert(*T*,8)



Application: Sorting

We can use BST-Insert and Inorder-Tree-Walk to sort a list of ***n*** numbers

BST-Sort
1: root[***T***]=NIL
2: for ***i***=1to ***n*** do
3: BST-Insert(***T***,***A***[***i***])
4: Inorder-Tree-Walk(T)

Sort Input: 5, 10, 3, 5, 7, 5, 4, 8

Inorder Walk: 3, 4, 5, 5, 7, 8, 10

Analysis

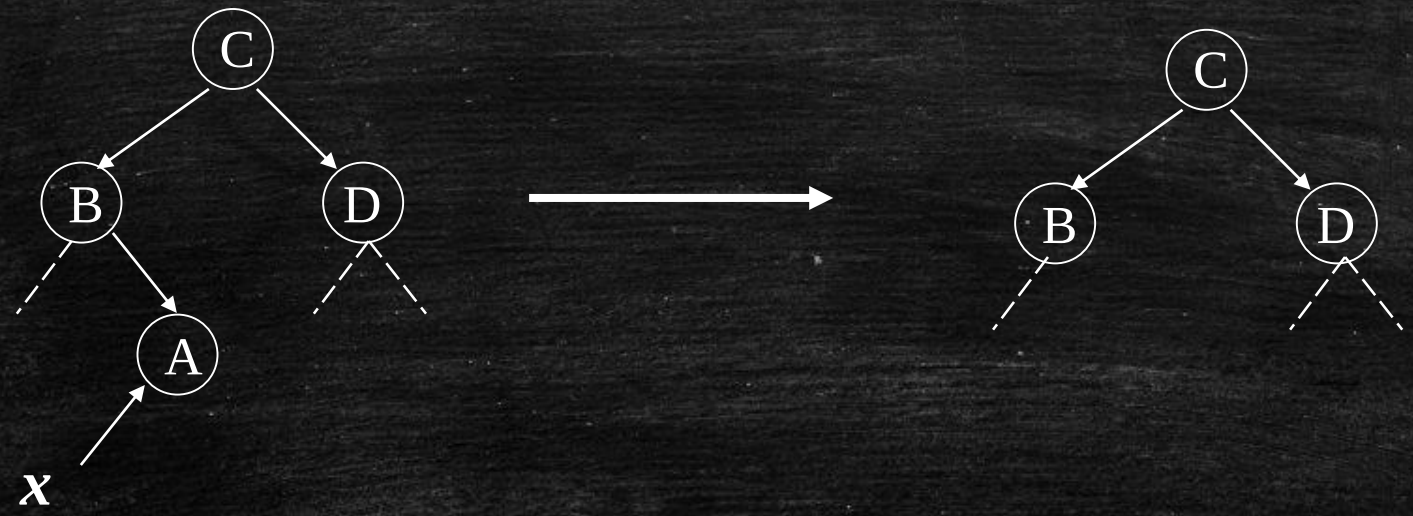
- The running time depends on the height of the tree (the Insert time).
- Average case analysis is like quick sort (which element becomes the root).
- Therefore the expected running time is $O(n \log n)$.
- Average BST height is $O(\log n)$.

Deletion

- Delete a node ***x*** from tree ***T***
- Has three cases:
  - **Case 1**: ***x*** has no child.
  - **Case 2**: ***x*** has one child.
  - **Case 3**: ***x*** has two children.

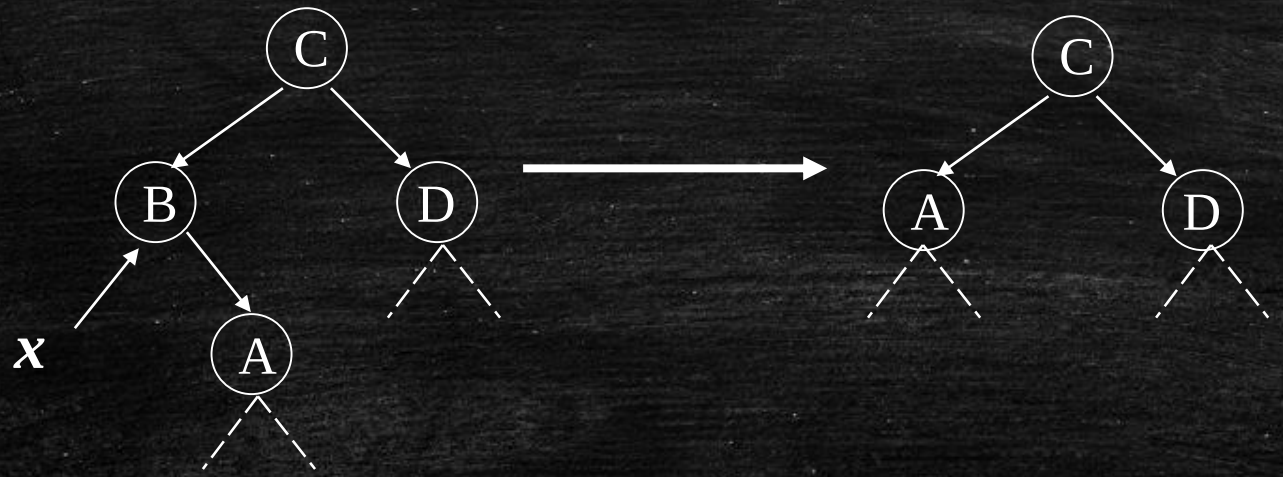
Deletion

- **Case 1**: ***x*** has no children.



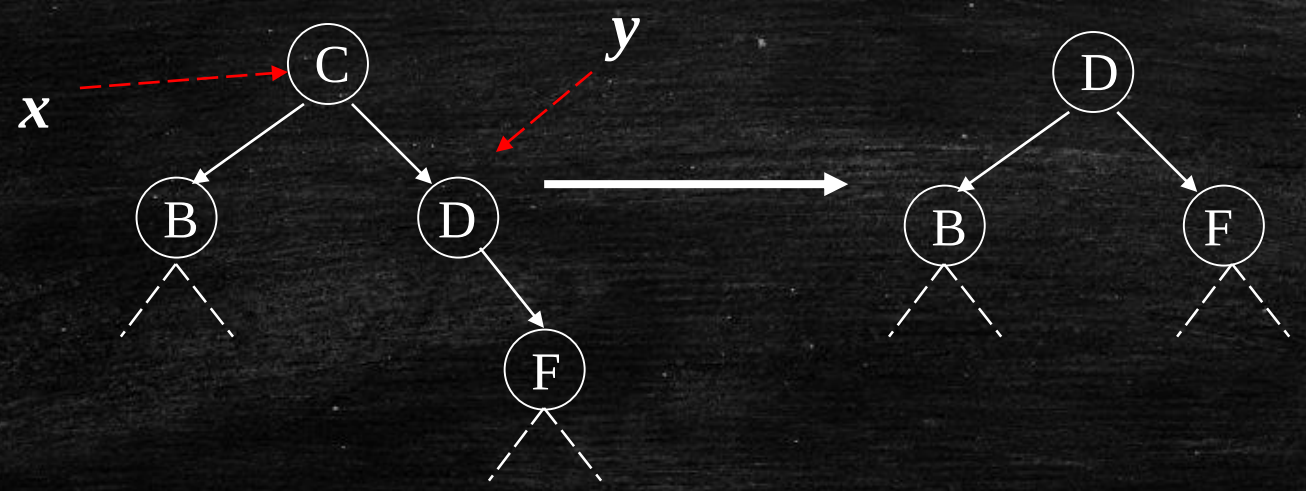
Deletion

- **Case 2**: ***x*** has one child (call it ***y***).
  - Elevate **y** to take the position of **x**



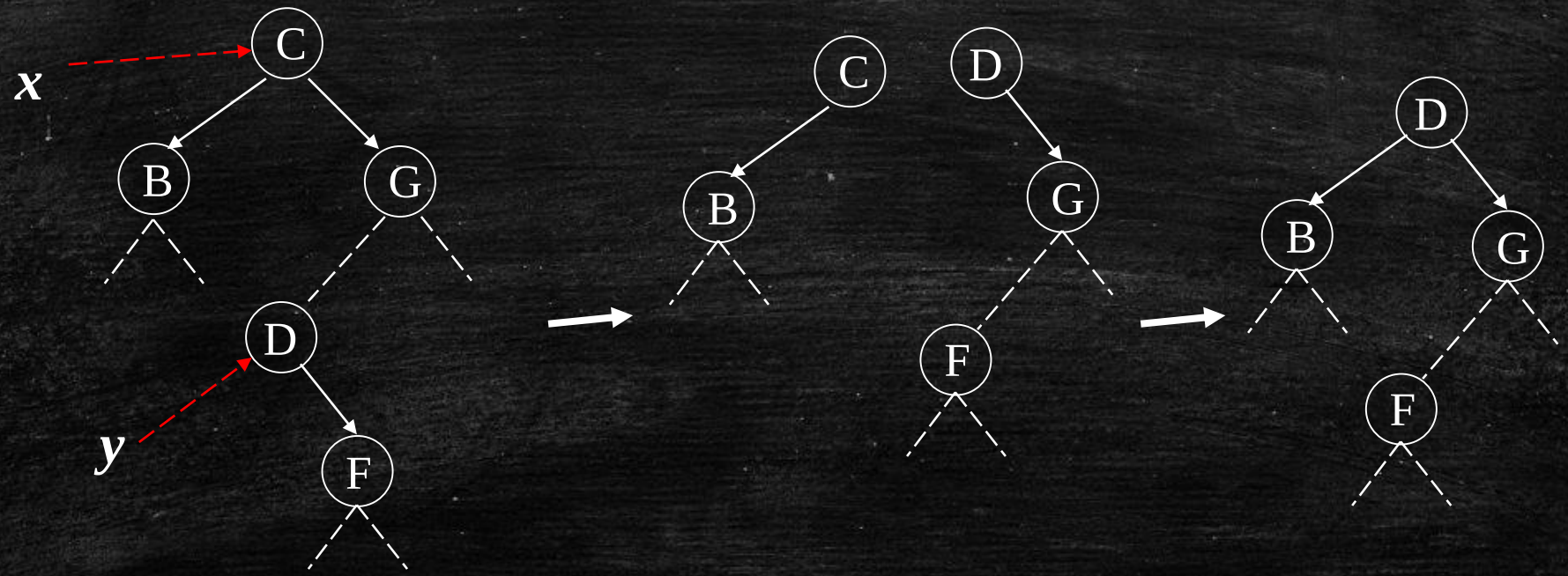
Deletion

- **Case 3**: ***x*** has two children:
  - Find its successor ***y*** in right subtree.
    - y is an immediate child of x
    - y is a descendant of x
  - Remove ***y.*** (Note ***y*** has at most one child, why?)
  - Replace ***x*** by ***y***.



Deletion

- **Case 3**: ***x*** has two children:
  - Find its successor ***y*** in right subtree.
    - y is an immediate child of x
    - y is a descendant of x



Second Problem Set

1. (20 pts) Solve problem 4-3 on page 108 of your textbook.

2. (15 pts) You are given a set S of n integers, as well as one more integer v. Design an algorithm that determines whether or not there exist two distinct elements $x, y \in S$ such that $x + y = v$. Your algorithm should run in time $O(n \log n)$, and it should return (x, y) if such elements exist and (NIL, NIL) otherwise. Prove the worst case running time bound and the correctness of the algorithm.

3. (15 pts) Prove tight **worst-case** asymptotic upper bounds for the following recurrence equations that satisfy $T(n) = 1$ for $n \leq 2$, and depend on a variable $q \in [0, n/4]$:

   (a) $T(n) = T(n - 2q - 1) + T(3q/2) + T(q/2) + \Theta(1)$
   (b) $T(n) = T(n - q - 1) + T(n/2 - q) + \Theta(n)$
   (c) $T(n) = T(n - q - 1) + T(3q) + \Theta(n)$

4. (10 pts) Consider the following silly randomized variant of binary search. You are given a sorted array A of n integers and the integer v that you are searching for is chosen uniformly at random from A. Then, instead of comparing v with the value in the middle of the array, the randomized binary search variant chooses a random number r from 1 to n and it compares v with $A[r]$. Depending on whether v is larger or smaller, this process is repeated recursively on the left sub-array or the right sub-array, until the location of v is found. Prove a tight bound on the expected running time of this algorithm.

5. (10 pts) Can the master method be applied to the recurrence $T(n) = 4T(n/2) + n^2 \log n$? Why or why not? Give an asymptotic upper bound for this recurrence.

Second Problem Set

6. (10 pts) Use a recursion tree to give an asymptotically tight solution to the recurrence:

   (a) $T(n) = T(n-a) + T(a) + cn$, where $a \geq 1$ and $c > 0$ are constants.

   (b) $T(n) = T(an) + T((1-a)n) + cn$, where $a \in (0,1)$ and $c > 0$ are constants.

7. (10 pts) For the following recurrences, find the bound implied by the master theorem. Then, try to prove the same bound using the substitution method. If your initial approach fails, show how it fails, and try subtracting off a lower-order term to make it work:

   (a) $T(n) = 4T(n/3) + n$

   (b) $T(n) = 4T(n/2) + n$

8. (5 pts) Solve the recurrence $T(n) = 3T(\sqrt{n}) + \log n$ by making a change of variable. Your solution should be asymptotically tight. Do not worry about whether values are integral.

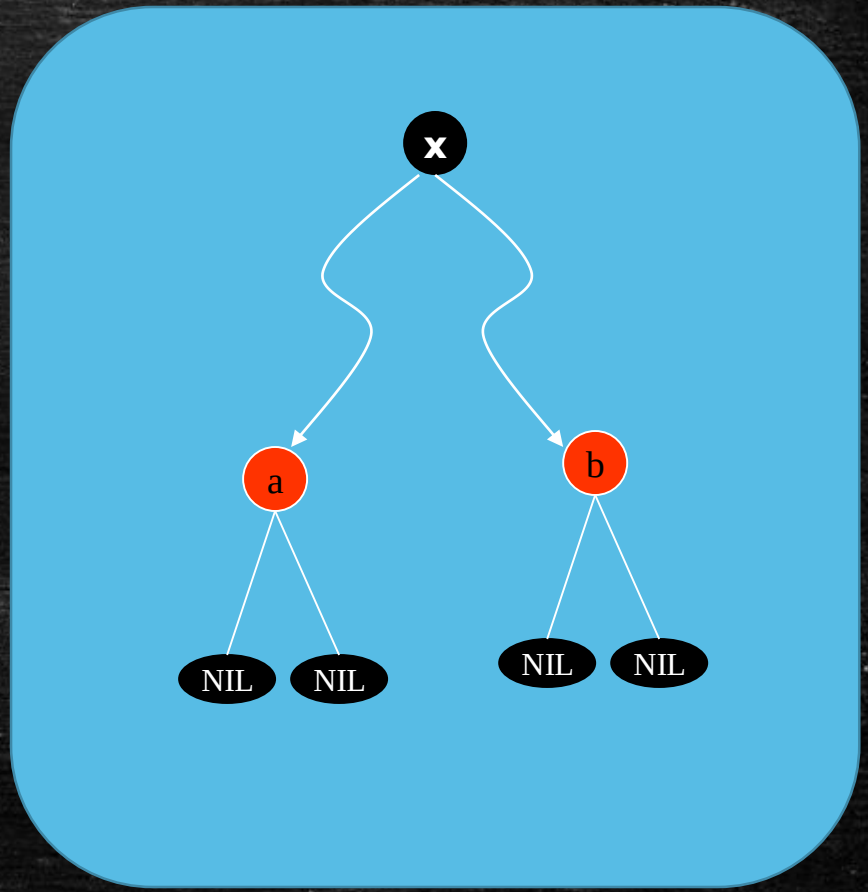
9. (5 pts) Using proof by induction, show that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in any n-element heap.

Red-black Trees

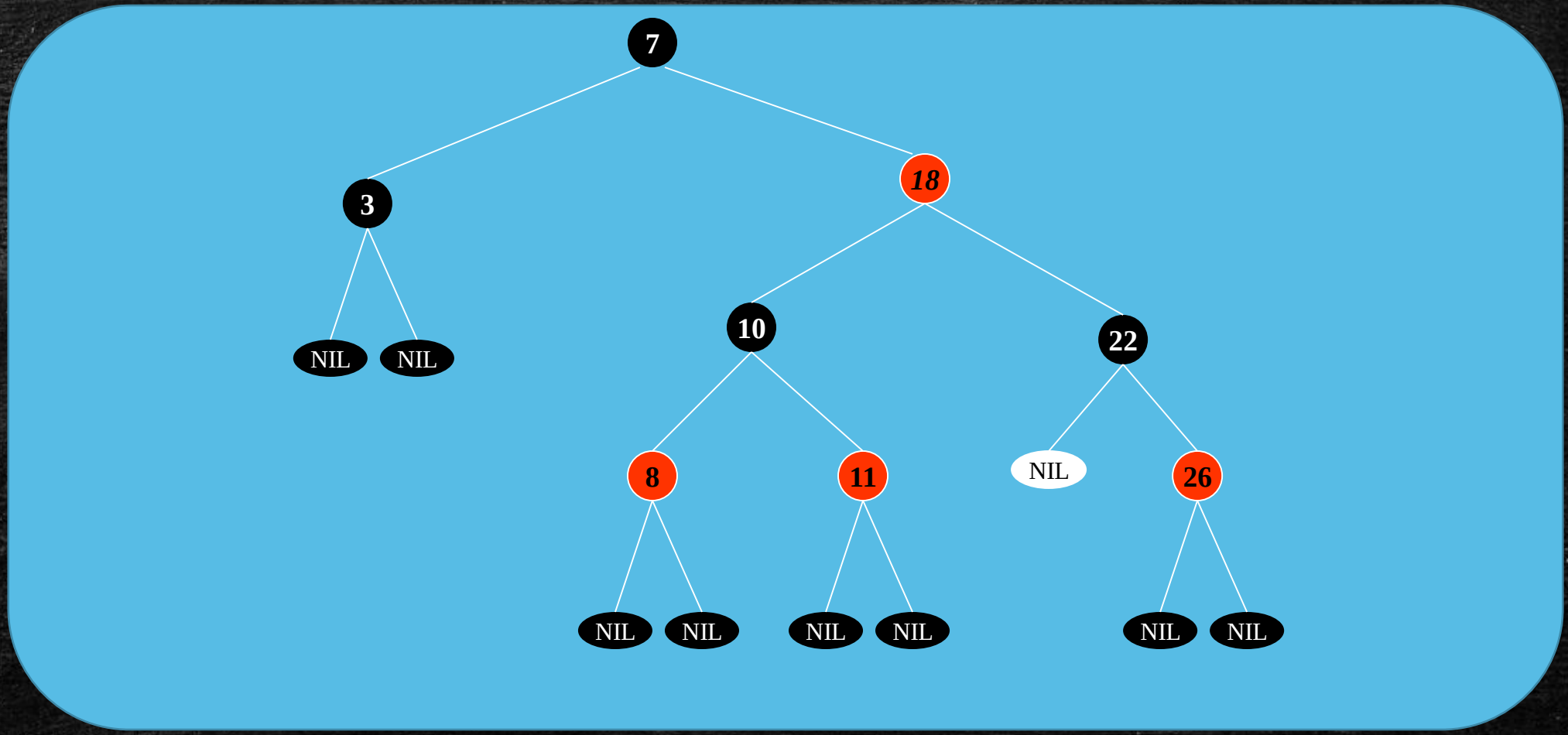
- They are balanced search trees (their height is $O(\log n)$)
- Most of the search and update operations on these trees take $O(\log n)$ time
- The structure is well balanced, i.e., each subtree is a balanced search tree.

Red-black Trees

1. Every node is either ***red*** or ***black***.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. All paths from a node ***x*** to a leaf have same number of black nodes (Black-Height(***x***))

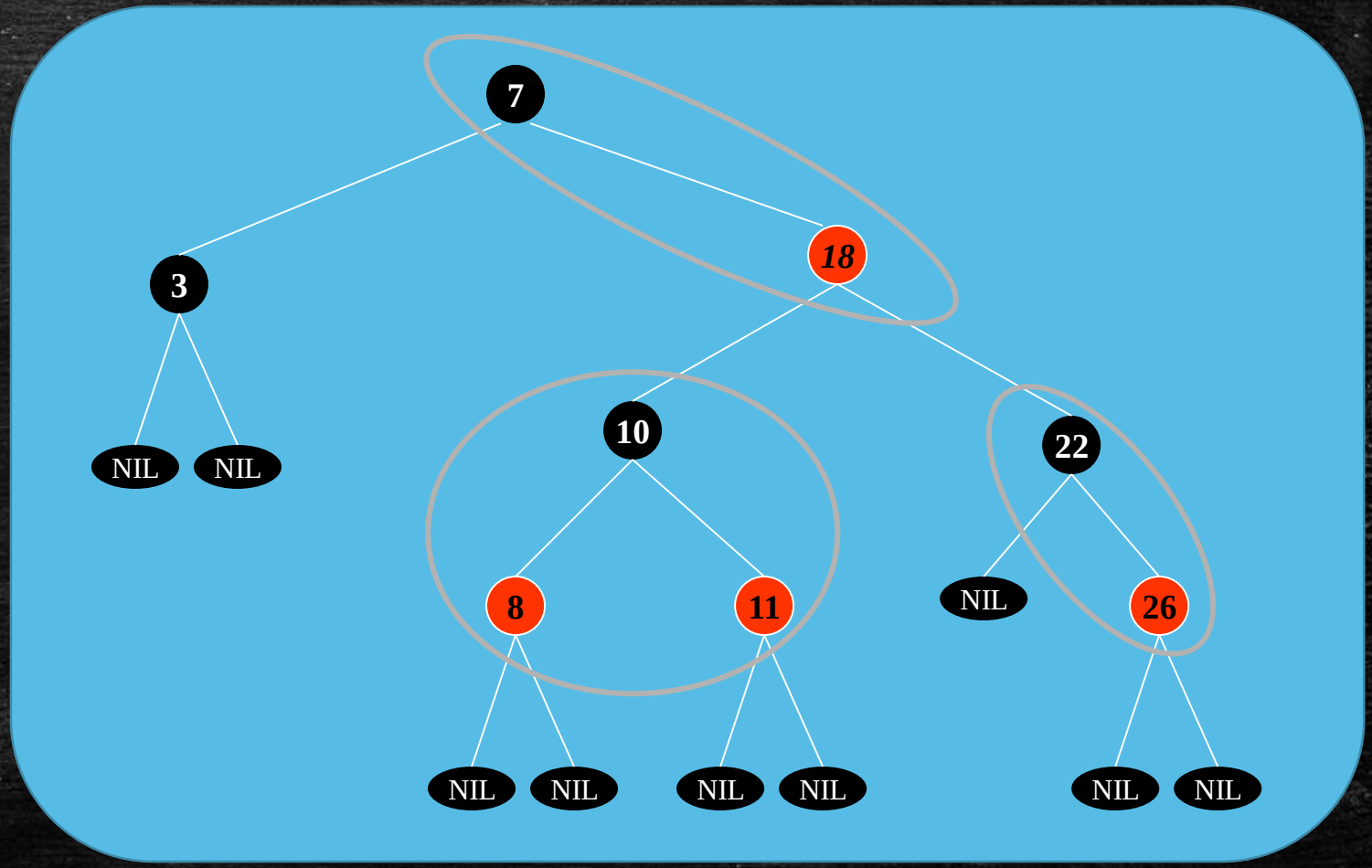


Example



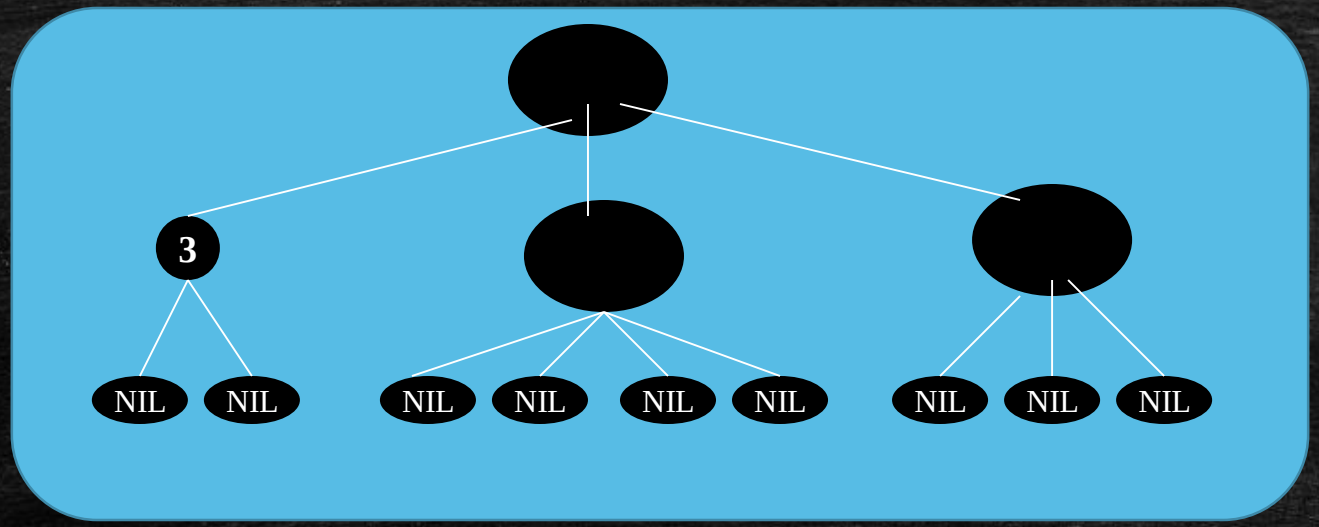
Height

- A red-black tree with n keys has height at most $2\lg(n+1)$.
- Proof (Intuition): Merge the red nodes into their parents



Proof

- Produces a tree with nodes having 2,3, or 4 nodes



- Height ***h***' of new tree is black height of original tree