

Week 4 Announcements

- Wattle site “*Announcements*”... all good?
- Survey 1 (weeks 4 and 5):
<https://wattlecourses.anu.edu.au/mod/feedback/view.php?id=2986807>
- Pointing to W3 reminders:
 - Video Assignment #1.
 - Group Project.<https://wattlecourses.anu.edu.au/mod/forum/discuss.php?d=890106>
- UML Exercise.
<https://wattlecourses.anu.edu.au/mod/forum/discuss.php?d=892522>
- Control Flow Graph Exercise.
<https://wattlecourses.anu.edu.au/mod/forum/discuss.php?d=892523>

We'll do some polling! Get ready!

<https://pollev.com/sjrm>



Waiting for sjrm's presentation to begin...

sjrm's presentation is underway. As soon as the activity is active, you'll see it on the screen here. Stay put.

Poll Everywhere helps boost engagement during remote meetings, virtual trainings, and online conferences.

A short reflection...

- **ABC iView** | *Four Corners* = *Shadow State*
- Pay attention to the following excerpt (video): **30:07—31:51**.



In the *Software Construction* process context, what are your main takeaways about Deloitte's claims?



Nobody has responded yet.
Hang tight! Responses are coming in.

COMP2100/6442

Software Design Methodologies / Software Construction

Tree Data Structures



Bernardo Pereira Nunes and
Sergio J. Rodríguez Méndez

Outline

- Data Structures
- Tree Data Structures
 - Binary Search Tree
 - Red-Black Tree
 - AVL Tree
- Exercises



This lecture presents one of the most important topics in Computer Science that is often overlooked!

Niklaus Wirth

Born February 15, 1934, Winterthur, Switzerland, early Promoter of good programming practices; developer of the programming languages Pascal, Modula-2, and Oberon; recipient of the 1984 ACM Turing Award.



Education: undergraduate studies at ETH Zurich, Dept. of Electrical Engineering, 1954-1958; diploma, Electronics Engineer ETH, 1959; MSc, Laval University, Quebec, Canada, 1960; PhD, electrical engineering, University of California, Berkeley, 1963.

Professional Experience: assistant professor of computer science, Stanford University, 1963-1967; assistant professor of computer science, University of Zurich, 1967-1968; professor of computer science, University of Zurich and ETH Zurich, 1968-1972; professor of computer science, ETH Zurich, 1972-present; sabbatical leaves, Xerox Palo Alto Research Center, 1977 and 1985; head of Department of Computer Science, ETH Zurich, 1982-1984 and 1988-1990; head of Institute of Computer Systems, ETH Zurich, 1990-present.

Honors and Awards: ACM Programming Systems and Languages Paper Award for "Towards a Discipline of Real-Time Programming," 1978; honorary doctorate, University of York, England, 1978; honorary doctorate, Ecole Polytechnique Federale, Lausanne, Switzerland, 1978; Computer Design, Hall of Fame Award, 1982; Emanuel R. Piore Award, IEEE, for "achievement in the field of Information Processing contributing to the advancement of science and the betterment of society," 1983; A.M. Turing Award, ACM, "for developing a sequence of innovative computer languages Euler, Algol-W, Pascal, and Modula. Pascal has become pedagogically significant, and has provided a foundation for future computer languages, systems, and architectural research," 1984; ACM-SIGCSE Award, "for

lucid, systematic,
and penetrating
treatment of basic
and dynamic data
structures, sorting,
recursive algorithms,
language structures,
and compiling

NIKLAUS WIRTH

Algorithms +
Data
Structures =
Programs

PRENTICE-HALL
SERIES IN
AUTOMATIC
COMPUTATION

- Niklaus Wirth (1976): "Algorithms + Data Structures = Programs"



What is the first thing (one word) that comes to your mind when you think about *Data Structures*?

Nobody has responded yet.

Hang tight! Responses are coming in.

Data Structures

> No matter how **efficient** the programming language is, if the chosen **data structure is not appropriate!**

Data Structures are universal!

> What is the purpose of data structures?

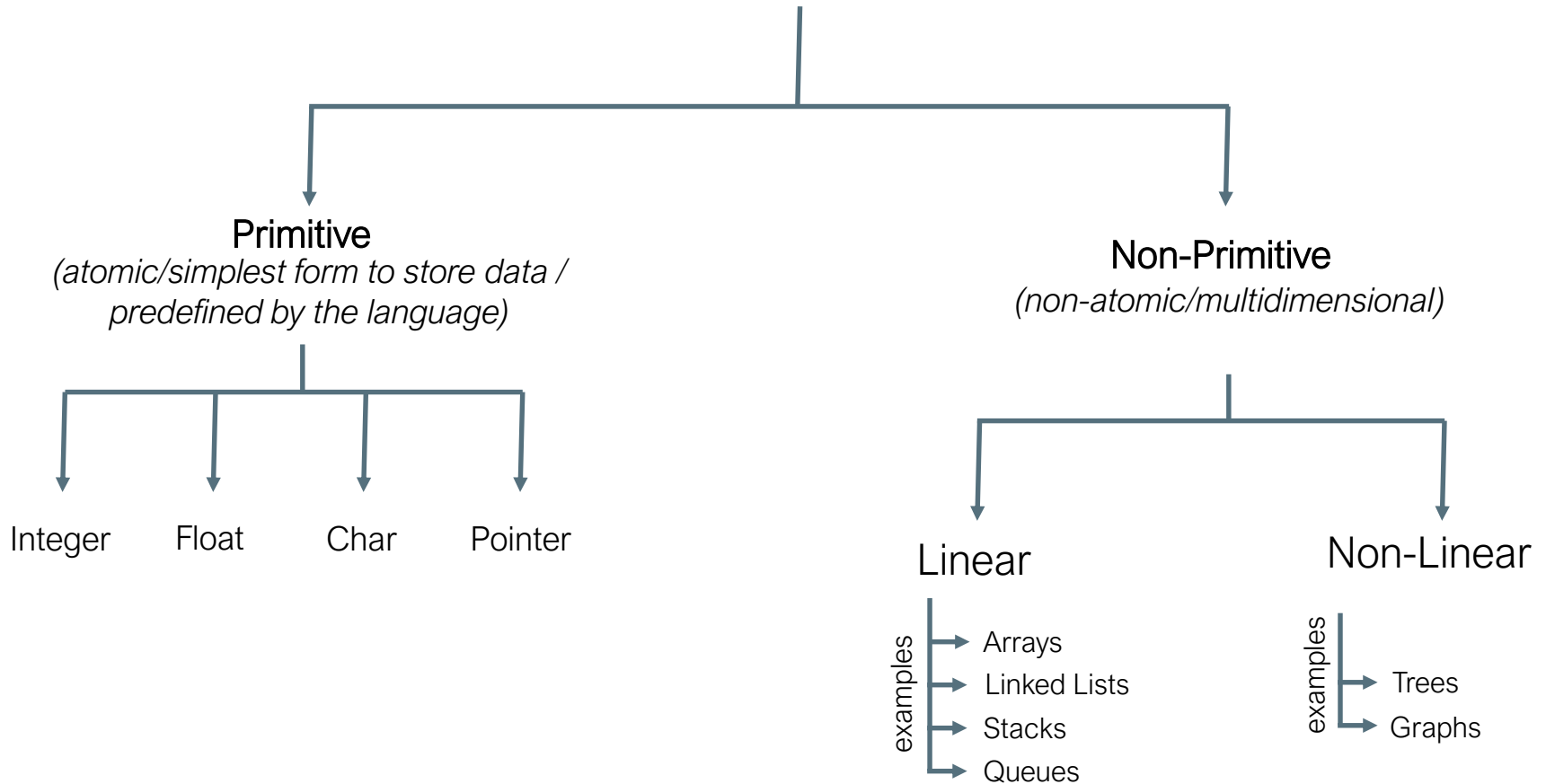
Data structures facilitate **data management and retrieval**

> What is the best data structure?

It can be a linked list or a tree, it **depends on the application**

> Most common data structures in Java are available in the **Java Collections Framework**

Data Structures



Java has Numeric, Textual, Boolean and Null primitives!

Data Structures

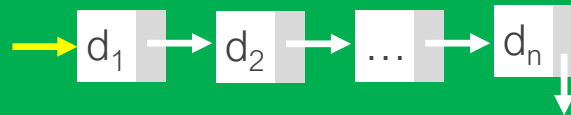
Linear



Array



Linked List



Memory

Arrays need one contiguous memory block.

.....

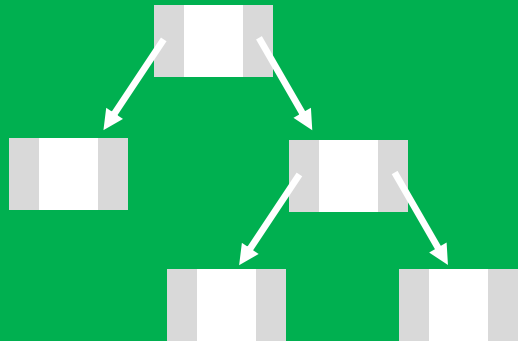
LLs do not need a contiguous memory block to be allocated. It can grow dynamically. It uses a pointer to the next element in the list.

- Elements/nodes are attached adjacently
- Traversing is sequential / Elements/nodes are arranged in some sort of sequence
- Easy to implement
- Use of memory may be ineffective

Example: Music Player; Lift system (one floor after the other [there is a sequence])

Data Structures

Non-Linear | e.g., Trees and Graphs



Memory

It can grow dynamically.
It uses pointers connect to
other elements in a
tree/graph.

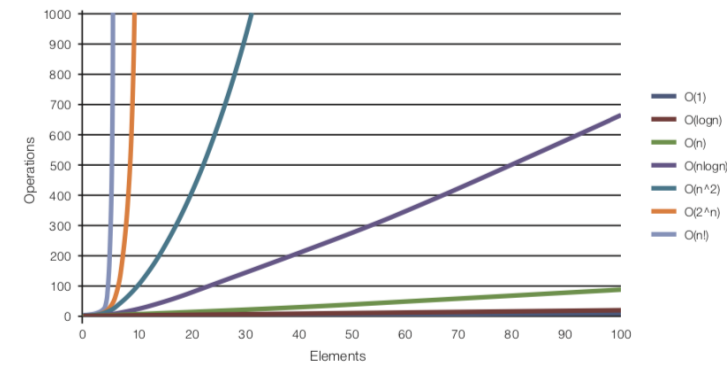
- Elements/nodes are connected to two or more elements/nodes
- Does not organize the elements/nodes in a sequence
- Difficult to implement (*just because we are comparing to the linear ones*)
- Use of memory is more effective

Example: File System (naturally hierarchical); store hierarchical data such as XML, HTML; create indices in DB; ...



Data Structures – Common Operations and Complexity

BIG-O COMPLEXITY CHART



DATA STRUCTURE OPERATIONS

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Stack	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Singly-Linked List	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Doubly-Linked List	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Skip List	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n \log(n))$
Hash Table	–	$O(1)$	$O(1)$	$O(1)$	–	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Binary Search Tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Cartesian Tree	–	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	–	$O(n)$	$O(n)$	$O(n)$	$O(n)$
B-Tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
Red-Black Tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
Splay Tree	–	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	–	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
AVL Tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$

Available on Wattle!

Tree Data Structures



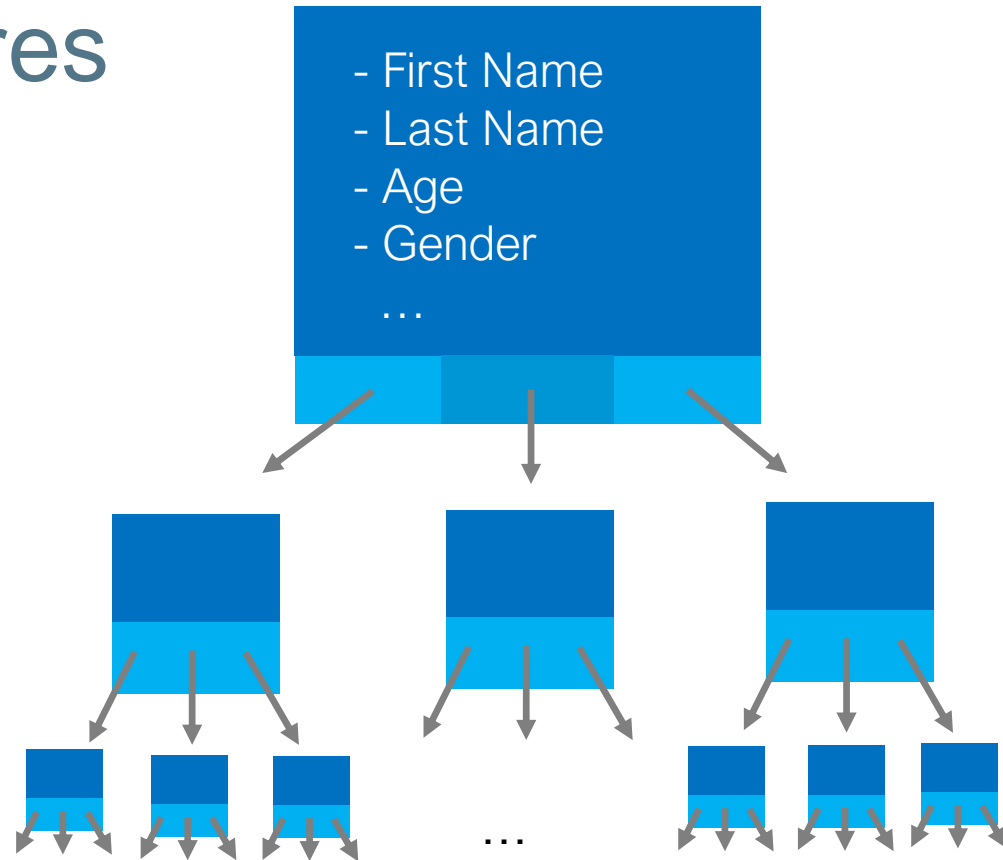
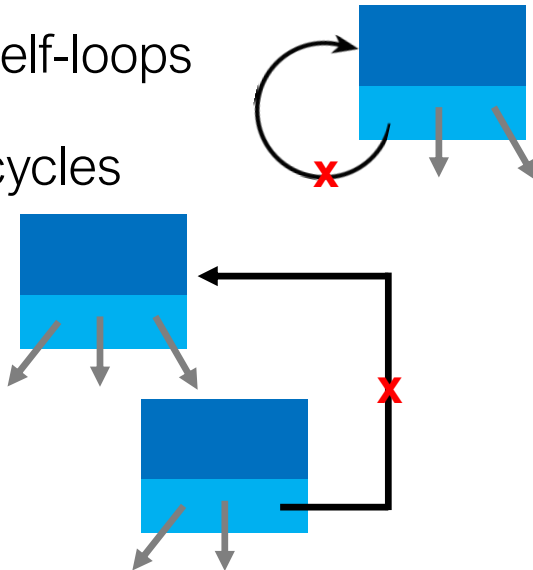
How ordinary people see trees!

How CS people see trees!



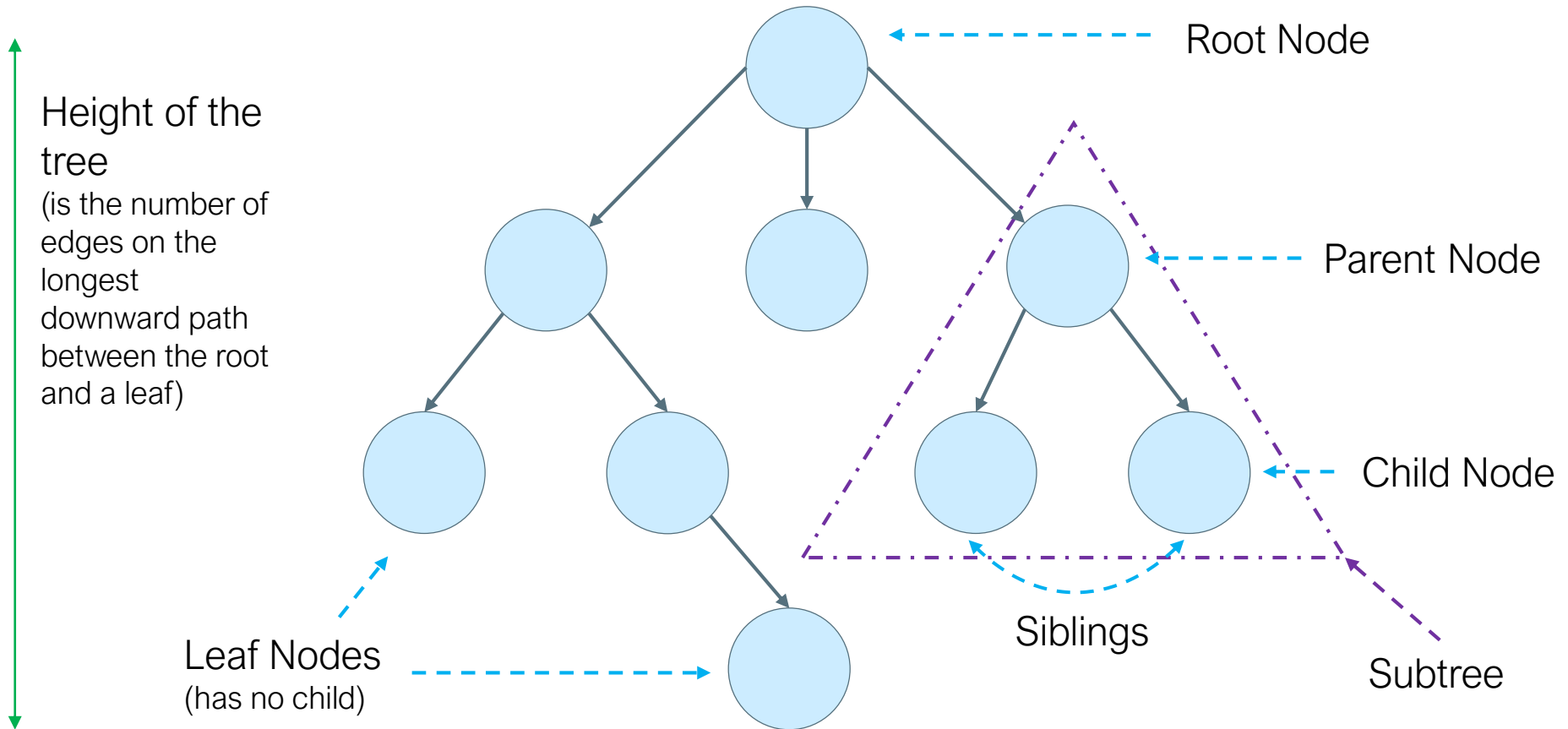
Tree Data Structures

- > Nodes (tuples of attributes)
- > Edges (pointers to other nodes)
- > No self-loops
- > No cycles



- > Natural data structure for hierarchical data

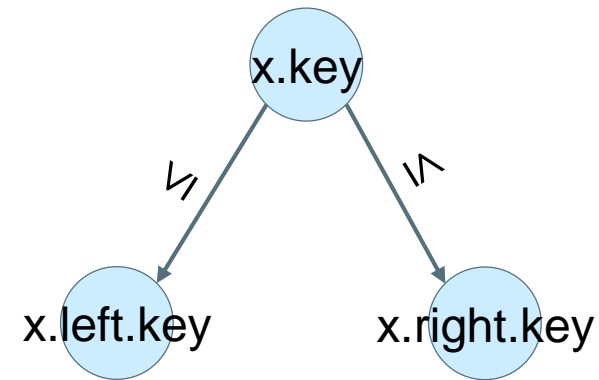
Tree Data Structures



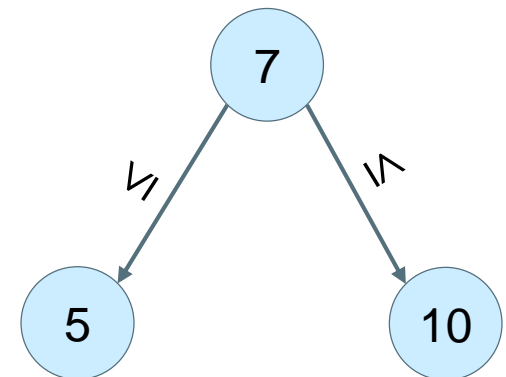
**Another representation using circles*

Binary Search Tree (BST)

- Nodes of (key, value) pairs (e.g. phone book)
- Keys can be sorted (e.g. numbers, strings)
- Edges relate the keys
- BST: tree with at most two children for each node
 - left child node is smaller than its parent node
 - right child node is greater than its parent node
- Data structures that can support **dynamic set operations**
 - Search, Minimum, Maximum, Predecessor, Successor, Insert, and Delete

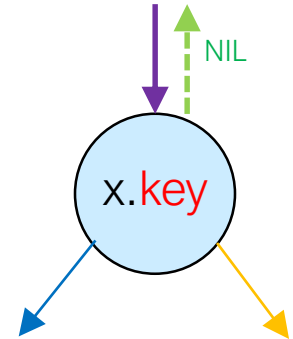


Example



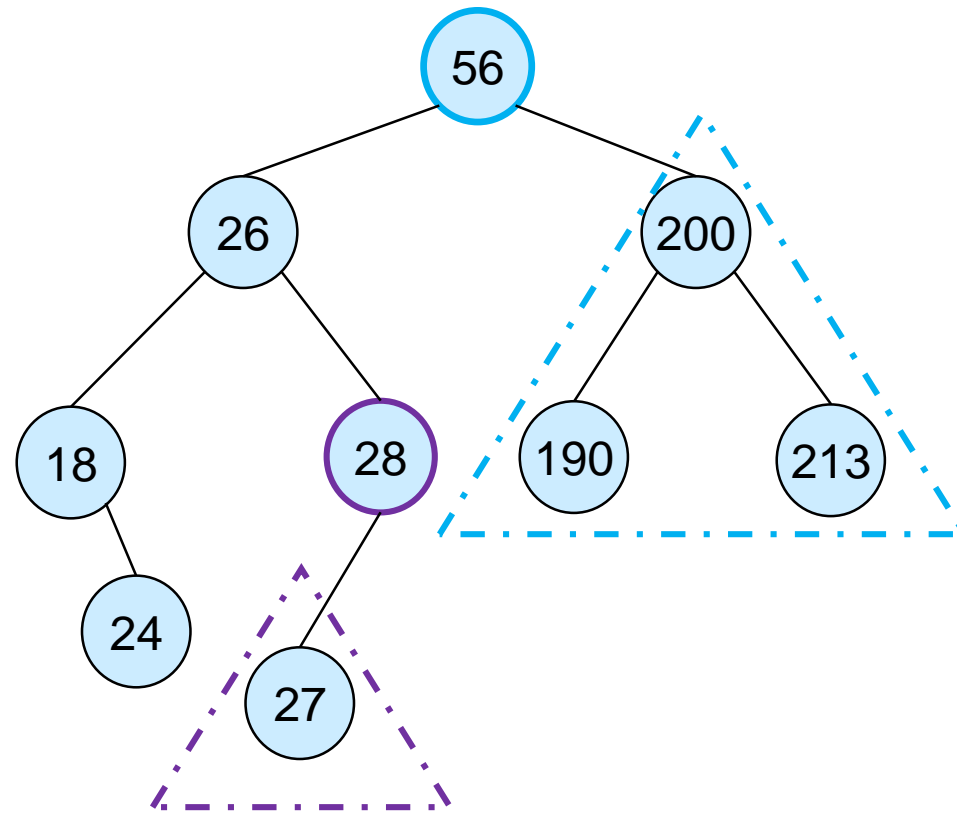
BST – Representation

- Represented by a **linked data structure of nodes**
- **root[T]** points to the root of tree T
- Each node contains fields:
 - **key** (and satellite data)
 - **left** – pointer to left child: root of left subtree
 - **right** – pointer to right child: root of right subtree
 - **p** – pointer to parent. $p[\text{root}[T]] = \text{NIL}$ (*optional*)



Binary Search Tree Property

- Stored keys must satisfy the **binary search tree** property
- Let x be a node in a BST. If y is a node in the left subtree:
 - $\forall y$ in left subtree of x , then $\text{key}[y] \leq \text{key}[x]$
- If y is a node in the right subtree:
 - $\forall y$ in right subtree of x , then $\text{key}[y] \geq \text{key}[x]$



**Keys are integer values for simplification!*

Traversing BST

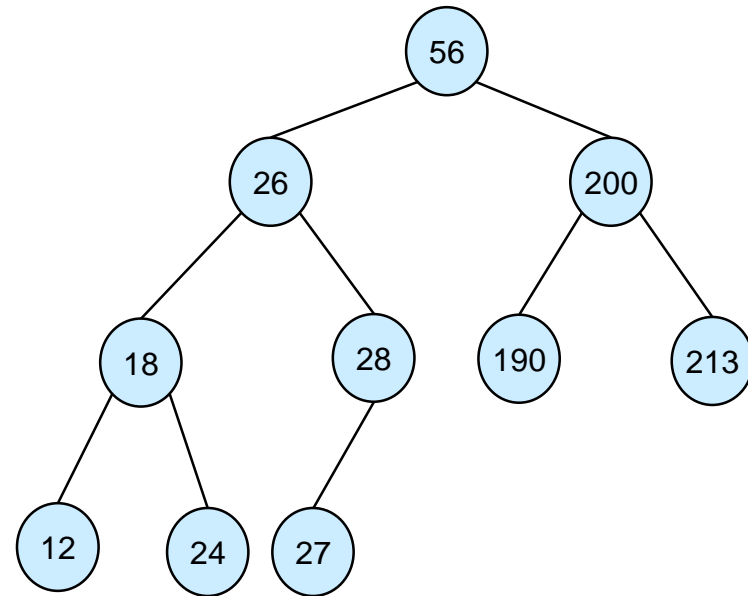
- How to enumerate all the nodes (traverse tree from the root)?
 - It is used to print out the data in a tree in a certain order
 - or apply some operations to each node
- Recursive traverse methods:
 - In-order traversing
 - Pre-order traversing
 - Post-order traversing

In-order Traversal

1. Traverse the left subtree
2. Visit the root
3. Traverse the right subtree

Inorder-Tree-Walk (x)

1. if $x \neq \text{null}$
2. then Inorder-Tree-Walk(left[x])
3. print key[x]
4. Inorder-Tree-Walk(right[x])

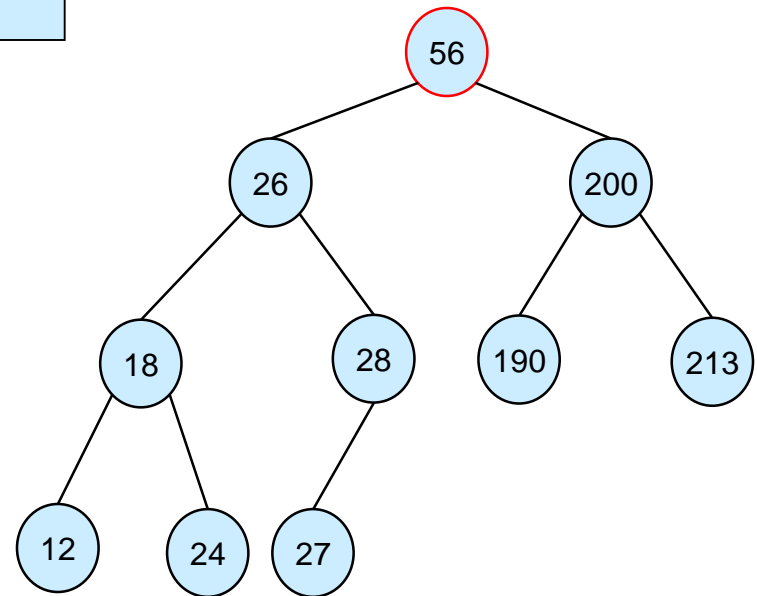


Inorder-Tree-Walk (x) ■

1. if $x \neq \text{null}$ ■
2. then Inorder-Tree-Walk(left[x])
3. print key[x]
4. Inorder-Tree-Walk(right[x])

Running Example

Inorder-Tree-Walk(x); //key:56, $x \neq \text{null}$

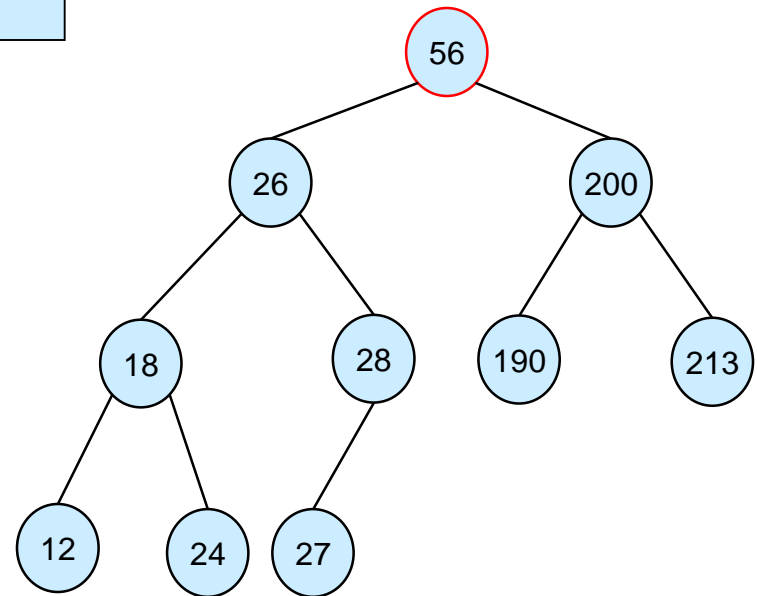


Inorder-Tree-Walk (x) ■

1. if $x \neq \text{null}$
2. then Inorder-Tree-Walk(left[x]) ■
3. print key[x]
4. Inorder-Tree-Walk(right[x])

Running Example

Inorder-Tree-Walk(x); //key:56, $x \neq \text{null}$



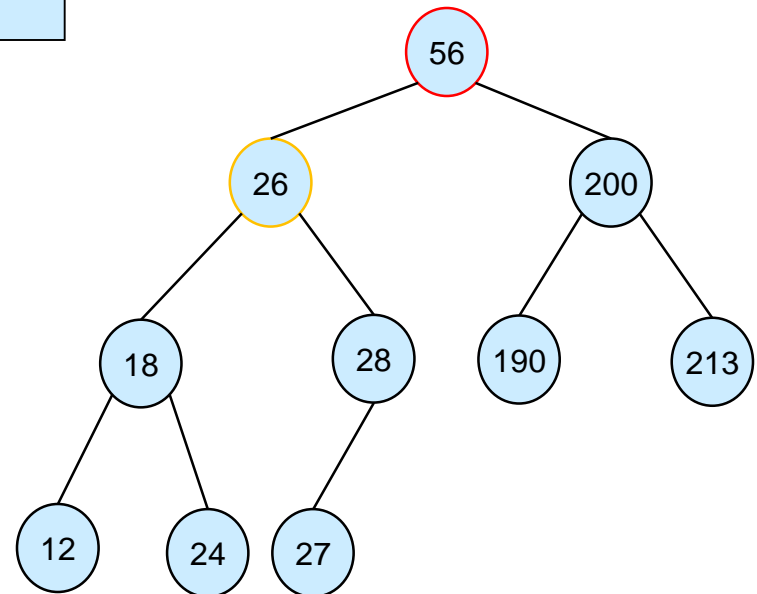
Inorder-Tree-Walk (x) ■

1. if $x \neq \text{null}$
2. then Inorder-Tree-Walk(left[x]) ■■
3. print key[x]
4. Inorder-Tree-Walk(right[x])

Running Example

Inorder-Tree-Walk(x); //key:56, $x \neq \text{null}$

Inorder-Tree-Walk(left(56)); //left: 26



Inorder-Tree-Walk (x) ■■

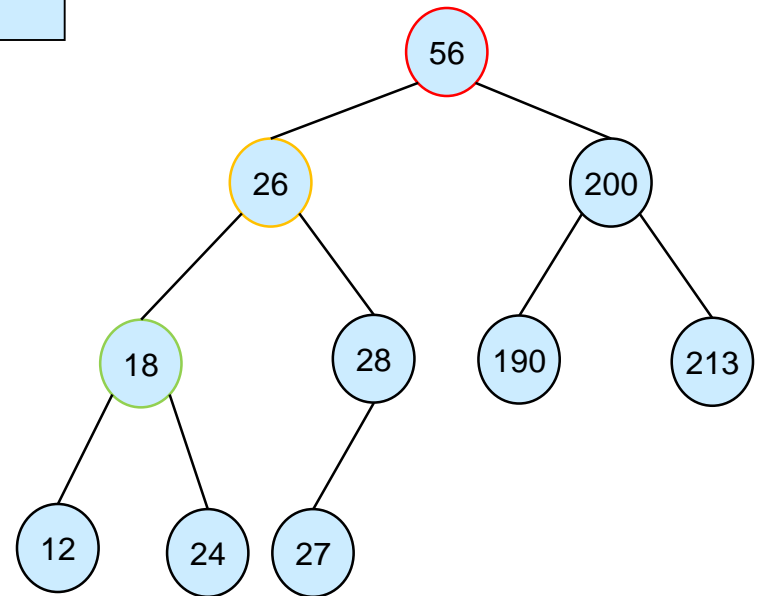
1. if $x \neq \text{null}$ ■
2. then Inorder-Tree-Walk(left[x]) ■■■
3. print key[x]
4. Inorder-Tree-Walk(right[x])

Running Example

Inorder-Tree-Walk(x); //key:56, $x \neq \text{null}$

Inorder-Tree-Walk(left(56)); //left: 26, $x \neq \text{null}$

Inorder-Tree-Walk(left(26)); //left: 18



Inorder-Tree-Walk (x) ■■■

1. if $x \neq \text{null}$ ■
2. then Inorder-Tree-Walk(left[x]) ■■■■
3. print key[x]
4. Inorder-Tree-Walk(right[x])

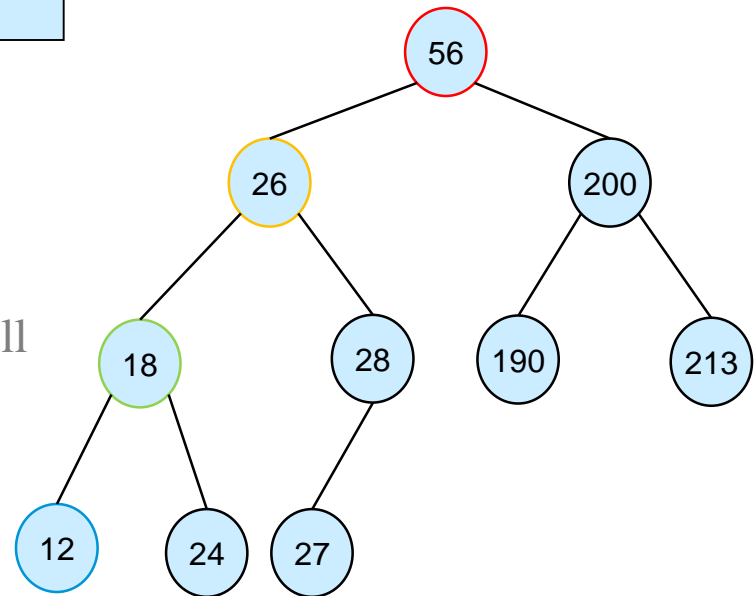
Running Example

Inorder-Tree-Walk(x); //key:56, $x \neq \text{null}$

Inorder-Tree-Walk(left(56)); //left: 26, $x \neq \text{null}$

Inorder-Tree-Walk(left(26)); //left: 18, $x \neq \text{null}$

Inorder-Tree-Walk(left(18)); //left: 12



Inorder-Tree-Walk (x) ■■■■

1. if $x \neq \text{null}$ ■
2. then Inorder-Tree-Walk(left[x]) ■■■■
3. print key[x]
4. Inorder-Tree-Walk(right[x])

Running Example

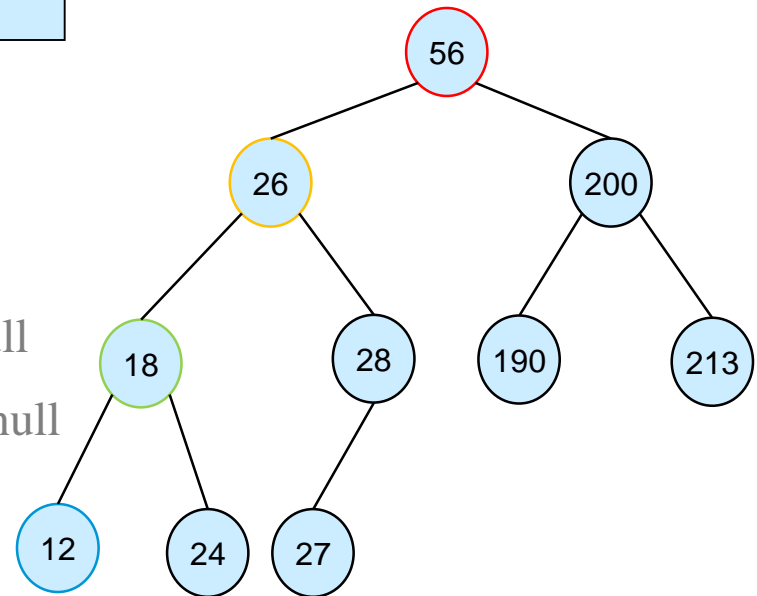
Inorder-Tree-Walk(x); //key:56, $x \neq \text{null}$

Inorder-Tree-Walk(left(56)); //left: 26, $x \neq \text{null}$

Inorder-Tree-Walk(left(26)); //left: 18, $x \neq \text{null}$

Inorder-Tree-Walk(left(18)); //left: 12, $x \neq \text{null}$

Inorder-Tree-Walk(left(12)); //left: null



Inorder-Tree-Walk (x)



1. if $x \neq \text{null}$ ■
2. then Inorder-Tree-Walk(left[x]) ■ ■ ■ ■ ■
3. print key[x]
4. Inorder-Tree-Walk(right[x])

Running Example

Inorder-Tree-Walk(x); //key:56, $x \neq \text{null}$

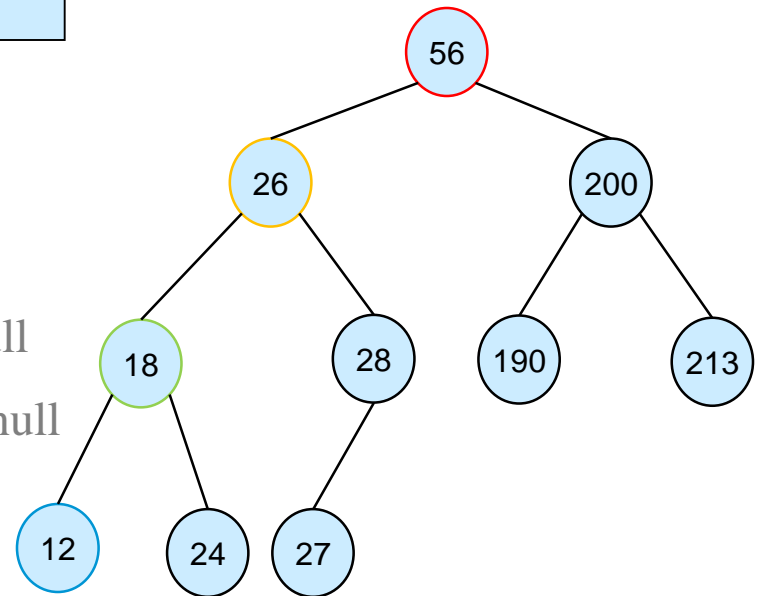
Inorder-Tree-Walk(left(56)); //left: 26, $x \neq \text{null}$

Inorder-Tree-Walk(left(26)); //left: 18, $x \neq \text{null}$

Inorder-Tree-Walk(left(18)); //left: 12, $x \neq \text{null}$

Inorder-Tree-Walk(left(12)); //left: null

//return



Inorder-Tree-Walk (x) ■■■■

1. if $x \neq \text{null}$
2. then Inorder-Tree-Walk(left[x]) ■■■
3. print key[x] ■
4. Inorder-Tree-Walk(right[x])

Output: 12

Running Example

Inorder-Tree-Walk(x); //key:56, $x \neq \text{null}$

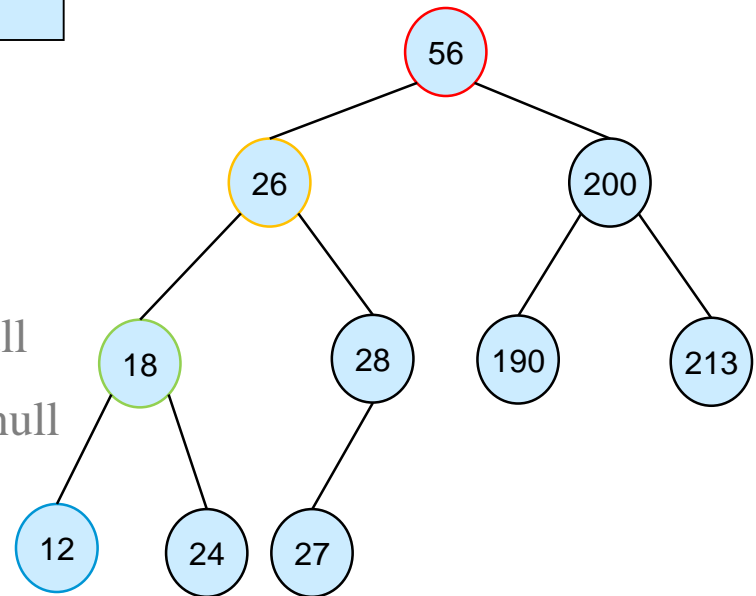
Inorder-Tree-Walk(left(56)); //left: 26, $x \neq \text{null}$

Inorder-Tree-Walk(left(26)); //left: 18, $x \neq \text{null}$

Inorder-Tree-Walk(left(18)); //left: 12, $x \neq \text{null}$



Inorder-Tree-Walk(left(12)); //left: null

print(key(12)); //output



Inorder-Tree-Walk (x)



1. if $x \neq \text{null}$
2. then Inorder-Tree-Walk(left[x]) 
3. print key[x]
4. Inorder-Tree-Walk(right[x]) 

Output: 12

Running Example

Inorder-Tree-Walk(x); //key:56, $x \neq \text{null}$

Inorder-Tree-Walk(left(56)); //left: 26, $x \neq \text{null}$

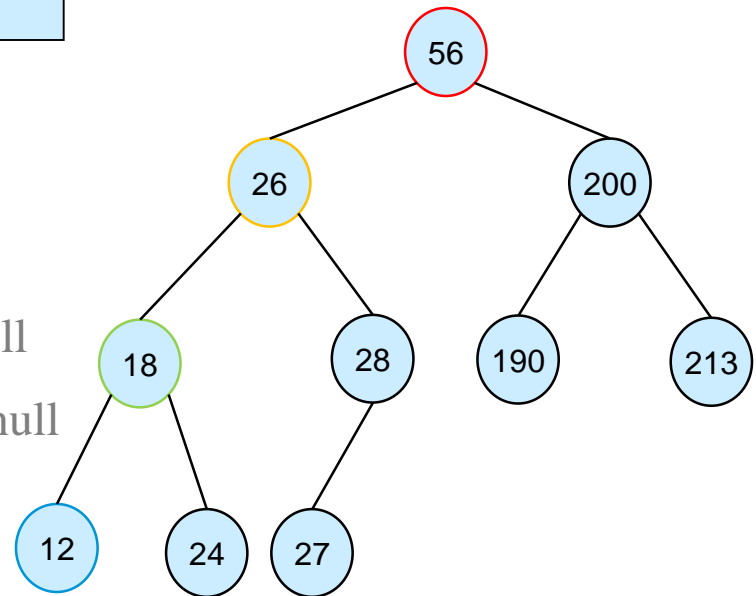
Inorder-Tree-Walk(left(26)); //left: 18, $x \neq \text{null}$

Inorder-Tree-Walk(left(18)); //left: 12, $x \neq \text{null}$

Inorder-Tree-Walk(left(12)); //left: null

print(key(12)); //output

Inorder-Tree-Walk(right(12)); //right: null



Inorder-Tree-Walk (x)



1. if $x \neq \text{null}$ ■
2. then Inorder-Tree-Walk(left[x]) ■ ■ ■
3. print key[x]
4. Inorder-Tree-Walk(right[x]) ■ ■

Output: 12

Running Example

Inorder-Tree-Walk(x); //key:56, $x \neq \text{null}$

Inorder-Tree-Walk(left(56)); //left: 26, $x \neq \text{null}$

Inorder-Tree-Walk(left(26)); //left: 18, $x \neq \text{null}$

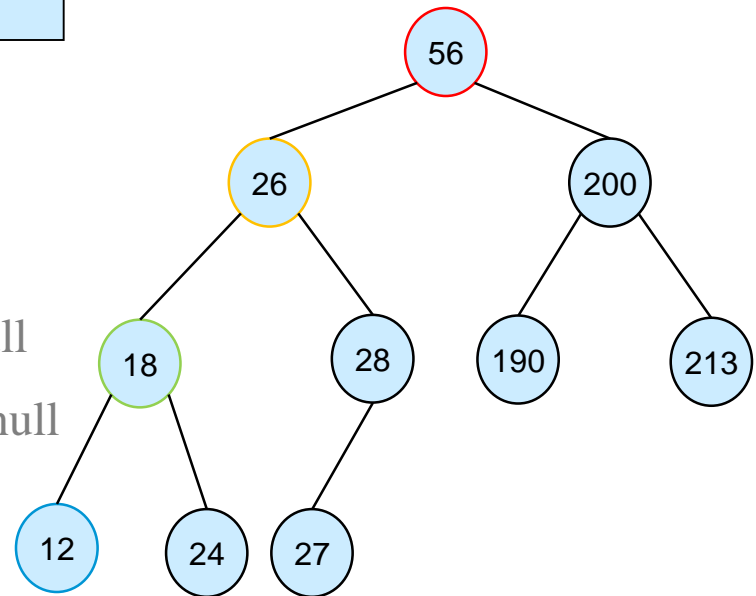
Inorder-Tree-Walk(left(18)); //left: 12, $x \neq \text{null}$

Inorder-Tree-Walk(left(12)); //left: null

print(key(12)); //output

Inorder-Tree-Walk(right(12)); //right: null

//return



Inorder-Tree-Walk (x) ■■■■

1. if $x \neq \text{null}$
2. then Inorder-Tree-Walk(left[x]) ■■■
3. print key[x]
4. Inorder-Tree-Walk(right[x]) ■

Output: 12

Running Example

~~Inorder-Tree-Walk(x); //key:56, $x \neq \text{null}$~~

~~Inorder-Tree-Walk(left(56)); //left: 26, $x \neq \text{null}$~~

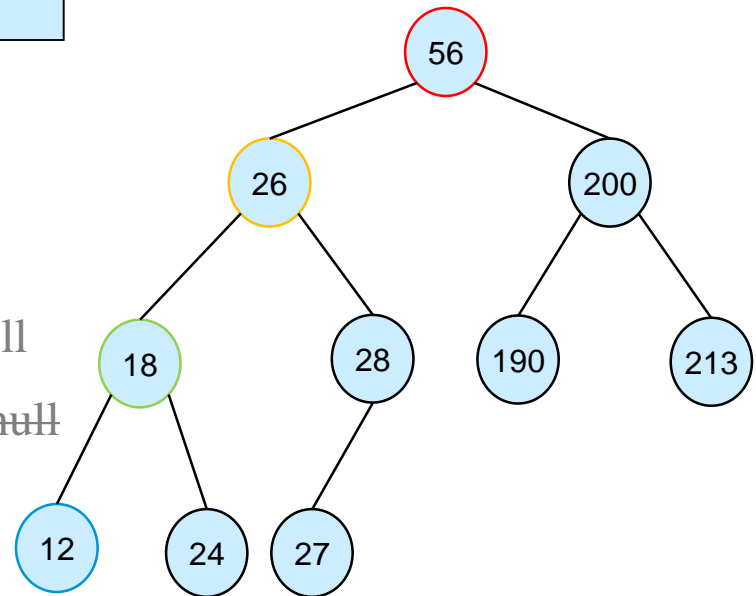
~~Inorder-Tree-Walk(left(26)); //left: 18, $x \neq \text{null}$~~

~~Inorder-Tree-Walk(left(18)); //left: 12, $x \neq \text{null}$~~

~~Inorder-Tree-Walk(left(12)); //left: null~~

~~print(key(12)); //output~~

~~Inorder-Tree-Walk(right(12)); //right: null~~



Inorder-Tree-Walk (x) ■■■

1. if $x \neq \text{null}$
2. then Inorder-Tree-Walk(left[x]) ■■
3. print key[x] ■
4. Inorder-Tree-Walk(right[x])

Output: 12, 18

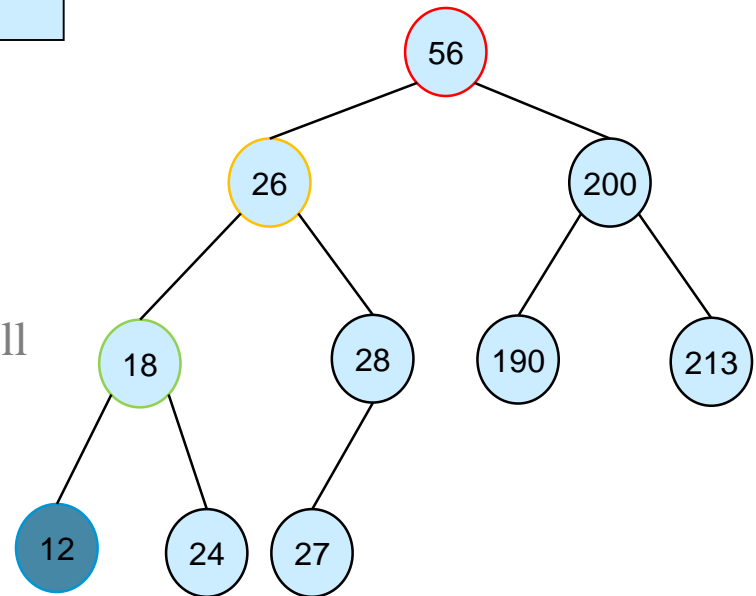
Running Example

Inorder-Tree-Walk(x); //key:56, $x \neq \text{null}$

Inorder-Tree-Walk(left(56)); //left: 26, $x \neq \text{null}$

Inorder-Tree-Walk(left(26)); //left: 18, $x \neq \text{null}$

print(key(18)); //output



Inorder-Tree-Walk (x) ■■■

1. if $x \neq \text{null}$
2. then Inorder-Tree-Walk(left[x]) ■■
3. print key[x]
4. Inorder-Tree-Walk(right[x]) ■

Running Example

Inorder-Tree-Walk(x); //key:56, $x \neq \text{null}$

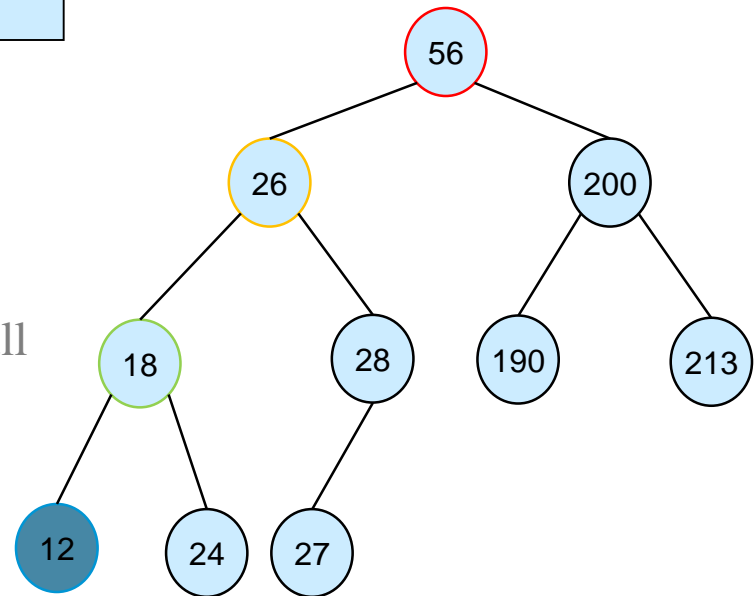
Inorder-Tree-Walk(left(56)); //left: 26, $x \neq \text{null}$

Inorder-Tree-Walk(left(26)); //left: 18, $x \neq \text{null}$

print(key(18)); //output

Continue to the right...

Output: 12, 18, 24, 26, 27, 28, 56, 190, 200, 213



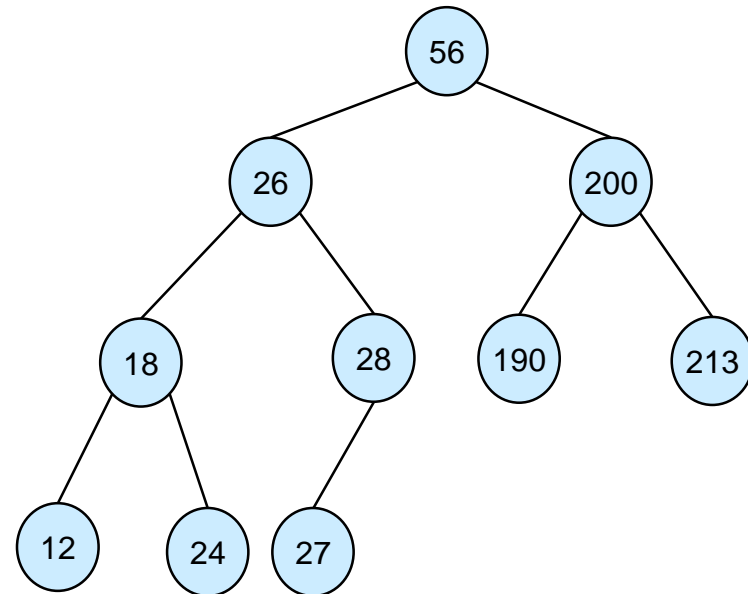
How long does the walk take?

Pre-order Traversal

1. Visit the root
2. Traverse the left subtree
3. Traverse the right subtree

Preorder-Tree-Walk (x)

1. if $x \neq \text{null}$
2. then print $\text{key}[x]$
3. Preorder-Tree-Walk($\text{left}[x]$)
4. Preorder-Tree-Walk($\text{right}[x]$)



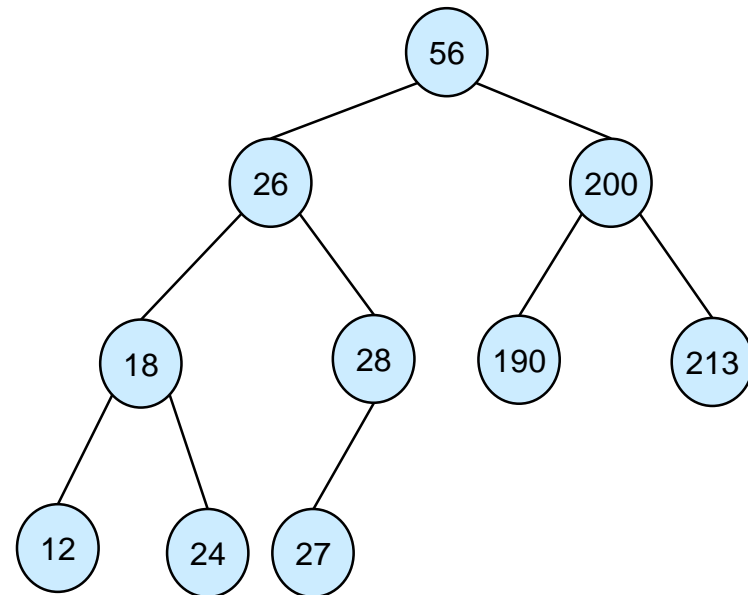
- Output: ?

Pre-order Traversal

1. Visit the root
2. Traverse the left subtree
3. Traverse the right subtree

Preorder-Tree-Walk (x)

1. if $x \neq \text{null}$
2. then print $\text{key}[x]$
3. Preorder-Tree-Walk($\text{left}[x]$)
4. Preorder-Tree-Walk($\text{right}[x]$)



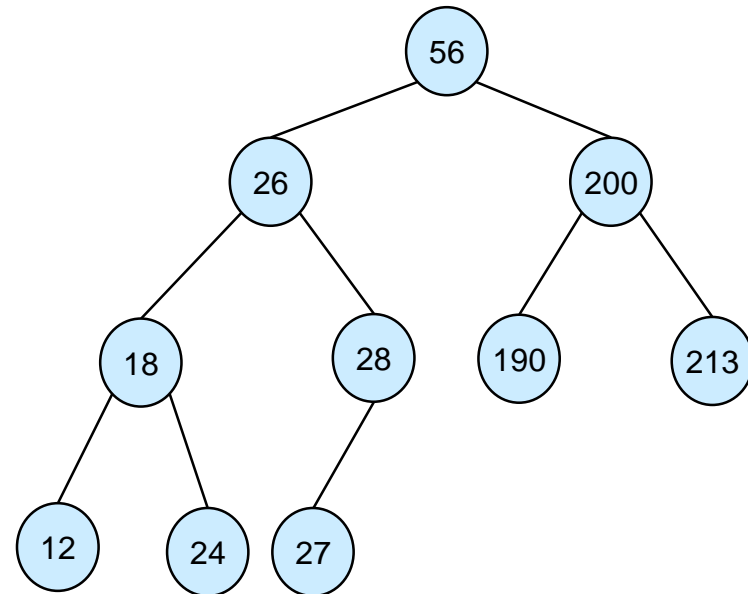
- Output: 56, 26, 18, 12, 24, 28, 27, 200, 190, 213

Post-order Traversal

1. Traverse the left subtree
2. Traverse the right subtree
3. Visit the root

Postorder-Tree-Walk (x)

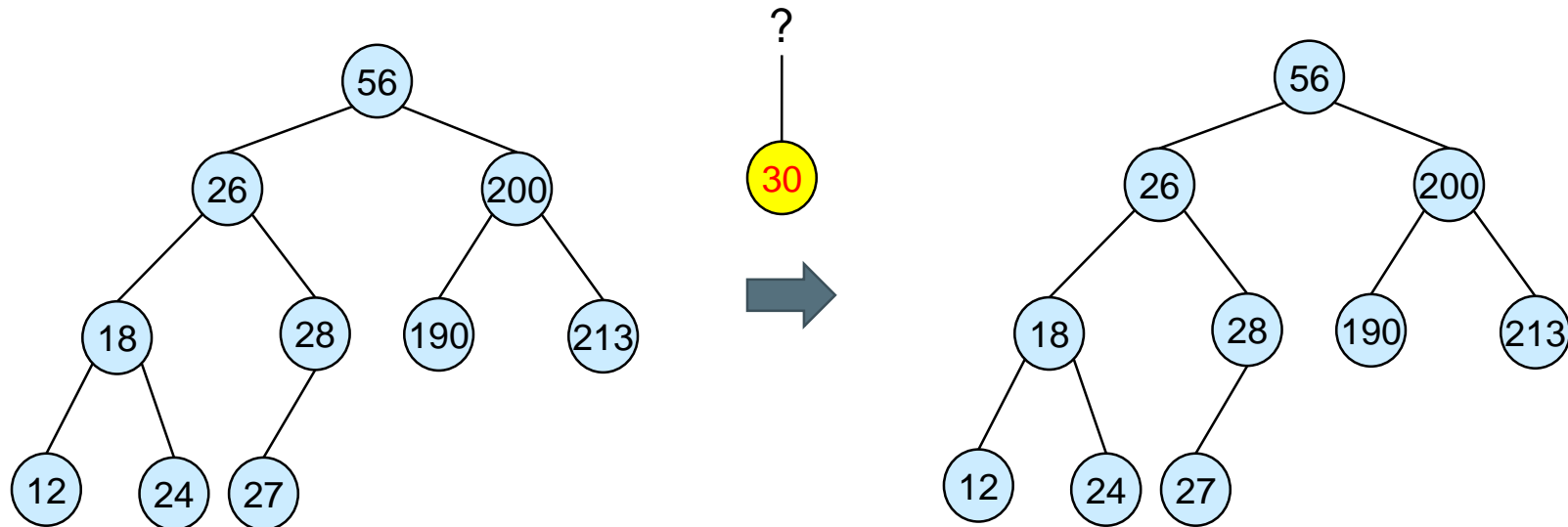
1. if $x \neq \text{null}$
2. then Postorder-Tree-Walk(left[x])
3. Postorder-Tree-Walk(right[x])
4. print key[x]



- Output: 12, 24, 18, 27, 28, 26, 190, 213, 200, 56

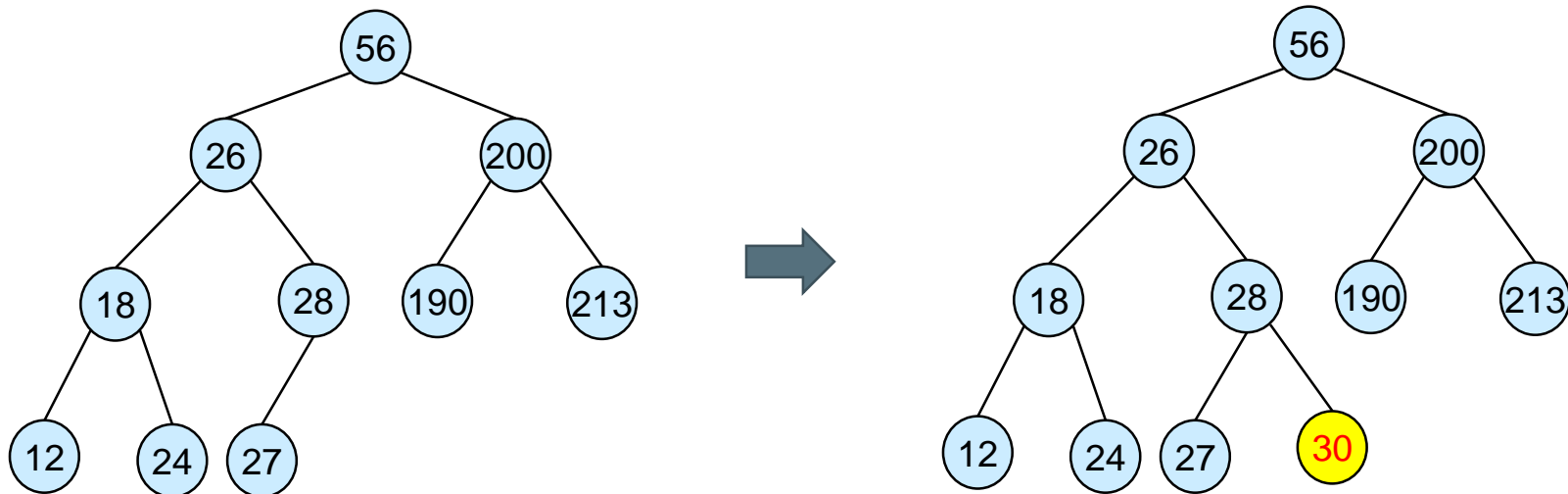
BST Insertion

- Ensure the **binary-search-tree** property holds after insertion
- A new key is always inserted at the leaf node

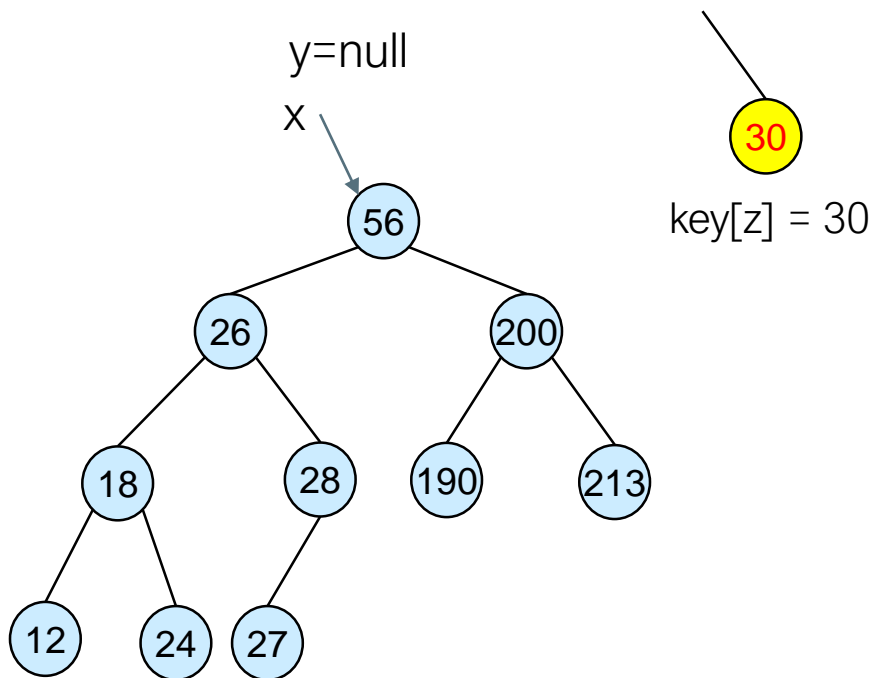


BST Insertion

- Ensure the **binary-search-tree** property holds after insertion
- A new key is always inserted at the leaf node



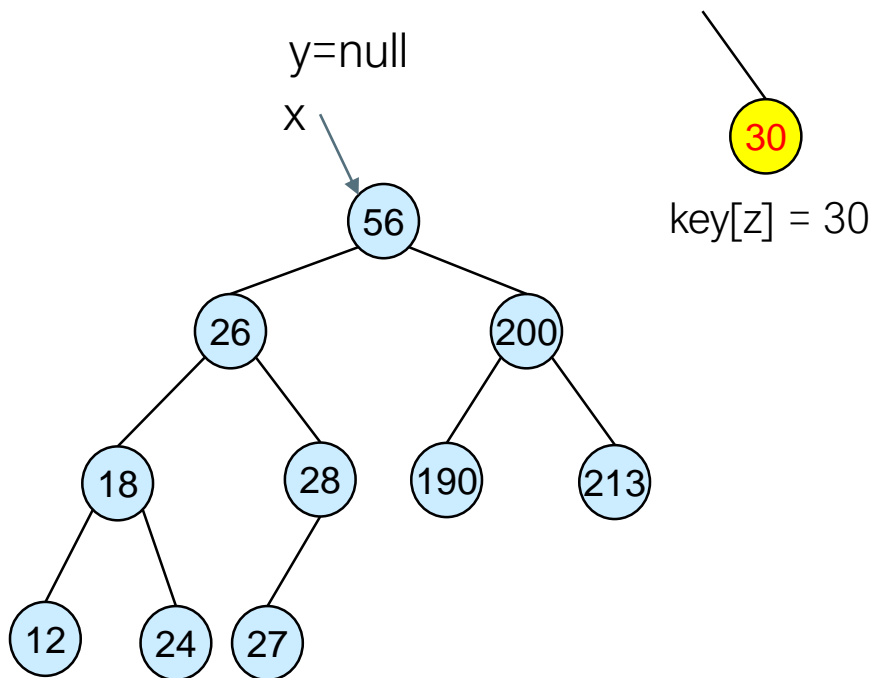
BST Insertion



Tree-Insert(T, z)

1. $y \leftarrow null$
2. $x \leftarrow root[T]$
3. **while** $x \neq null$
4. **do** $y \leftarrow x$
5. **if** $key[z] < key[x]$
6. **then** $x \leftarrow left[x]$
7. **else** $x \leftarrow right[x]$
8. $p[z] \leftarrow y$
9. **if** $y = null$
10. **then** $root[T] \leftarrow z$
11. **else if** $key[z] < key[y]$
12. **then** $left[y] \leftarrow z$
13. **else** $right[y] \leftarrow z$

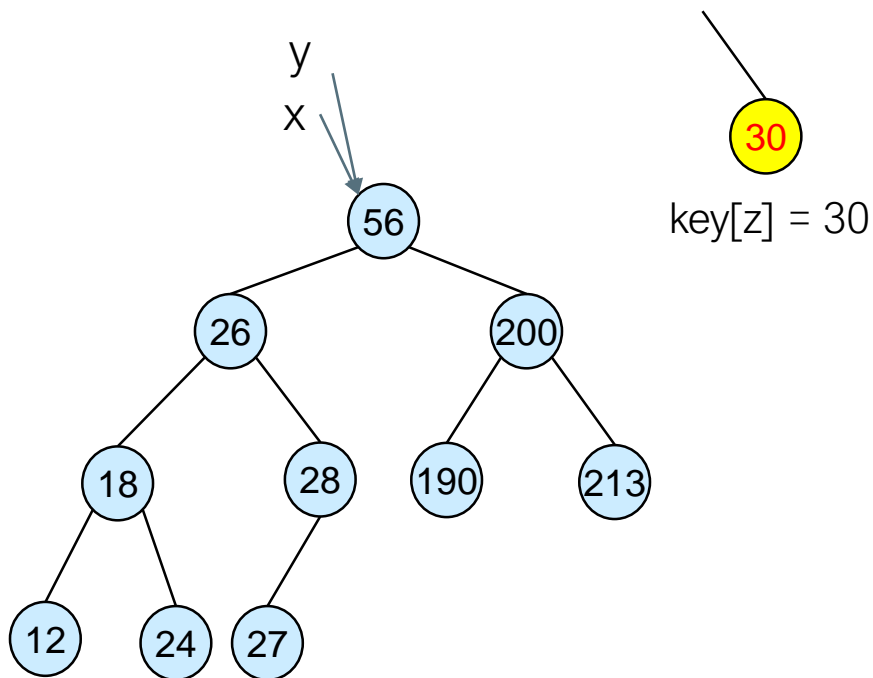
BST Insertion



Tree-Insert(T, z)

1. $y \leftarrow null$
2. $x \leftarrow root[T]$
3. **while** $x \neq null$
4. **do** $y \leftarrow x$
5. **if** $key[z] < key[x]$
6. **then** $x \leftarrow left[x]$
7. **else** $x \leftarrow right[x]$
8. $p[z] \leftarrow y$
9. **if** $y = null$
10. **then** $root[T] \leftarrow z$
11. **else if** $key[z] < key[y]$
12. **then** $left[y] \leftarrow z$
13. **else** $right[y] \leftarrow z$

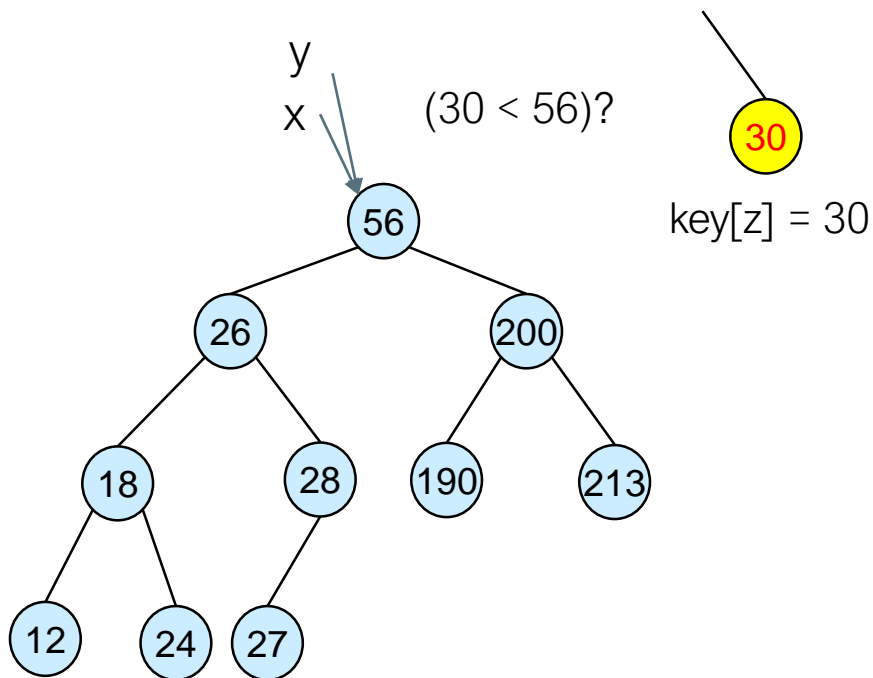
BST Insertion



Tree-Insert(T, z)

1. $y \leftarrow \text{null}$
2. $x \leftarrow \text{root}[T]$
3. **while** $x \neq \text{null}$
4. **do** $y \leftarrow x$
5. **if** $\text{key}[z] < \text{key}[x]$
6. **then** $x \leftarrow \text{left}[x]$
7. **else** $x \leftarrow \text{right}[x]$
8. $p[z] \leftarrow y$
9. **if** $y = \text{null}$
10. **then** $\text{root}[T] \leftarrow z$
11. **else if** $\text{key}[z] < \text{key}[y]$
12. **then** $\text{left}[y] \leftarrow z$
13. **else** $\text{right}[y] \leftarrow z$

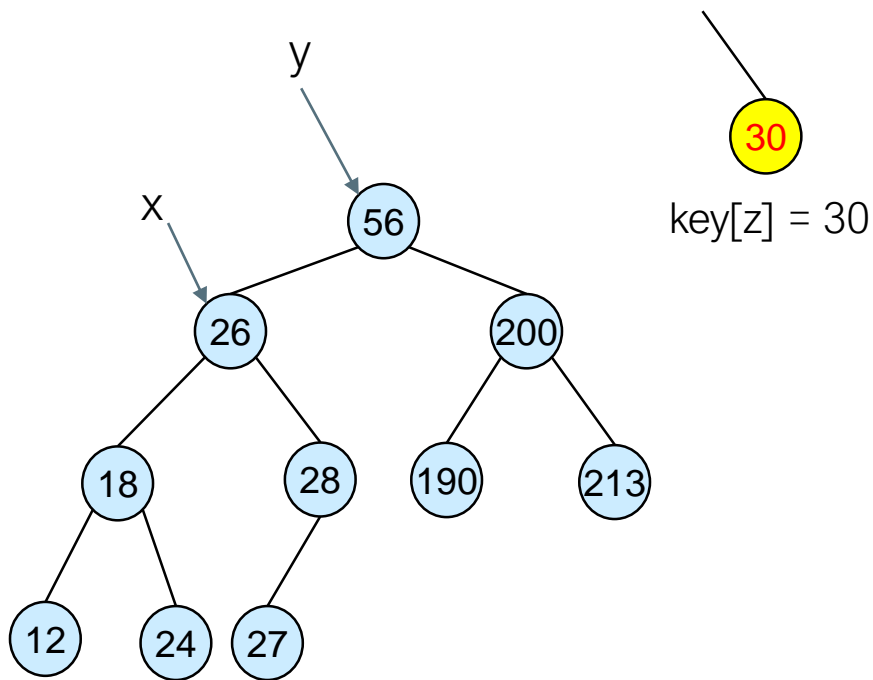
BST Insertion



Tree-Insert(T, z)

1. $y \leftarrow \text{null}$
2. $x \leftarrow \text{root}[T]$
3. **while** $x \neq \text{null}$
4. **do** $y \leftarrow x$
5. **if** $\text{key}[z] < \text{key}[x]$
6. **then** $x \leftarrow \text{left}[x]$
7. **else** $x \leftarrow \text{right}[x]$
8. $p[z] \leftarrow y$
9. **if** $y = \text{null}$
10. **then** $\text{root}[T] \leftarrow z$
11. **else if** $\text{key}[z] < \text{key}[y]$
12. **then** $\text{left}[y] \leftarrow z$
13. **else** $\text{right}[y] \leftarrow z$

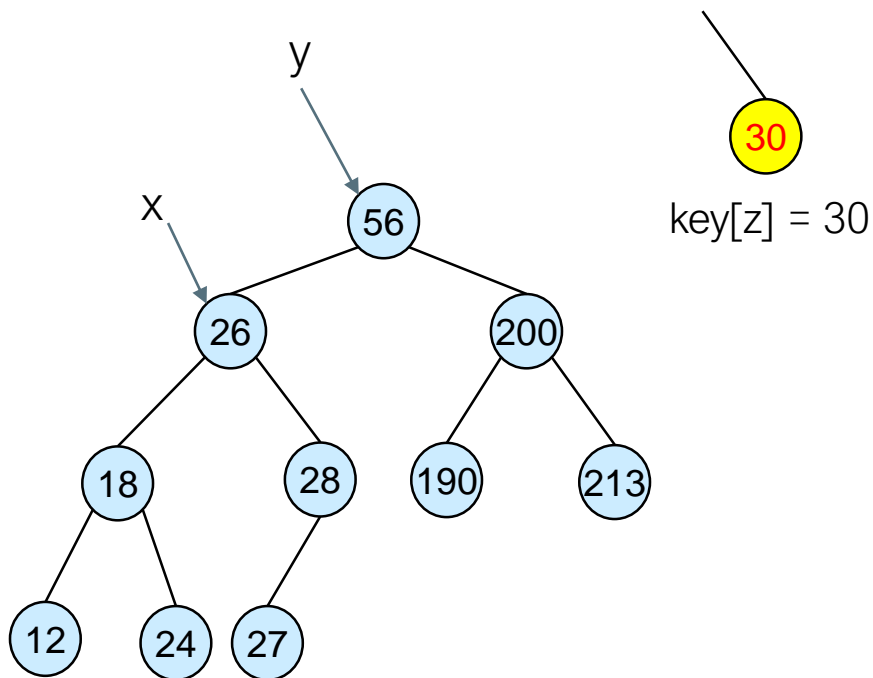
BST Insertion



Tree-Insert(T, z)

1. $y \leftarrow \text{null}$
2. $x \leftarrow \text{root}[T]$
3. **while** $x \neq \text{null}$
4. **do** $y \leftarrow x$
5. **if** $\text{key}[z] < \text{key}[x]$
6. **then** $x \leftarrow \text{left}[x]$
7. **else** $x \leftarrow \text{right}[x]$
8. $p[z] \leftarrow y$
9. **if** $y = \text{null}$
10. **then** $\text{root}[T] \leftarrow z$
11. **else if** $\text{key}[z] < \text{key}[y]$
12. **then** $\text{left}[y] \leftarrow z$
13. **else** $\text{right}[y] \leftarrow z$

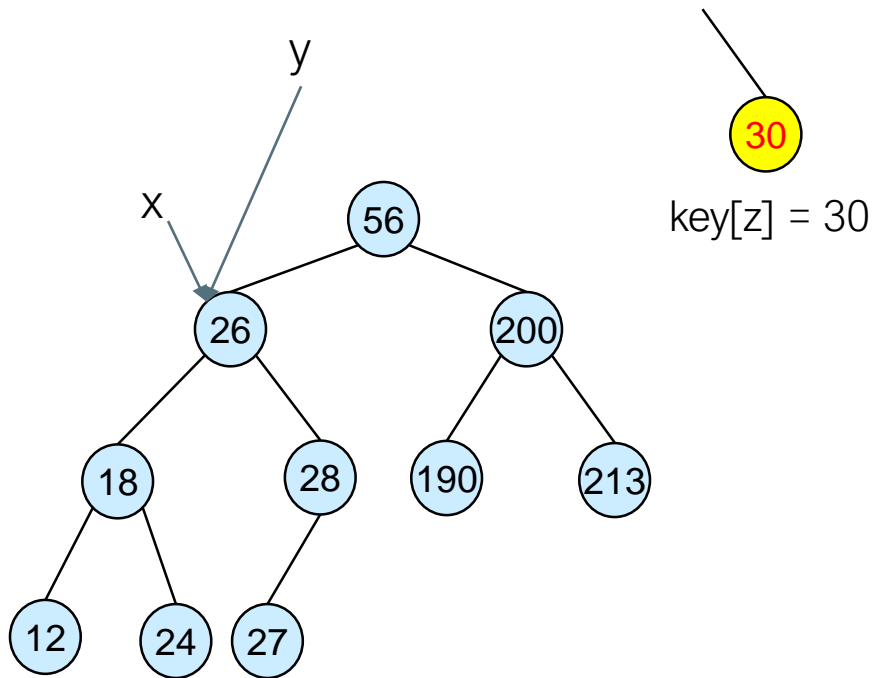
BST Insertion



Tree-Insert(T, z)

1. $y \leftarrow \text{null}$
2. $x \leftarrow \text{root}[T]$
3. **while** $x \neq \text{null}$
4. **do** $y \leftarrow x$
5. **if** $\text{key}[z] < \text{key}[x]$
6. **then** $x \leftarrow \text{left}[x]$
7. **else** $x \leftarrow \text{right}[x]$
8. $p[z] \leftarrow y$
9. **if** $y = \text{null}$
10. **then** $\text{root}[T] \leftarrow z$
11. **else if** $\text{key}[z] < \text{key}[y]$
12. **then** $\text{left}[y] \leftarrow z$
13. **else** $\text{right}[y] \leftarrow z$

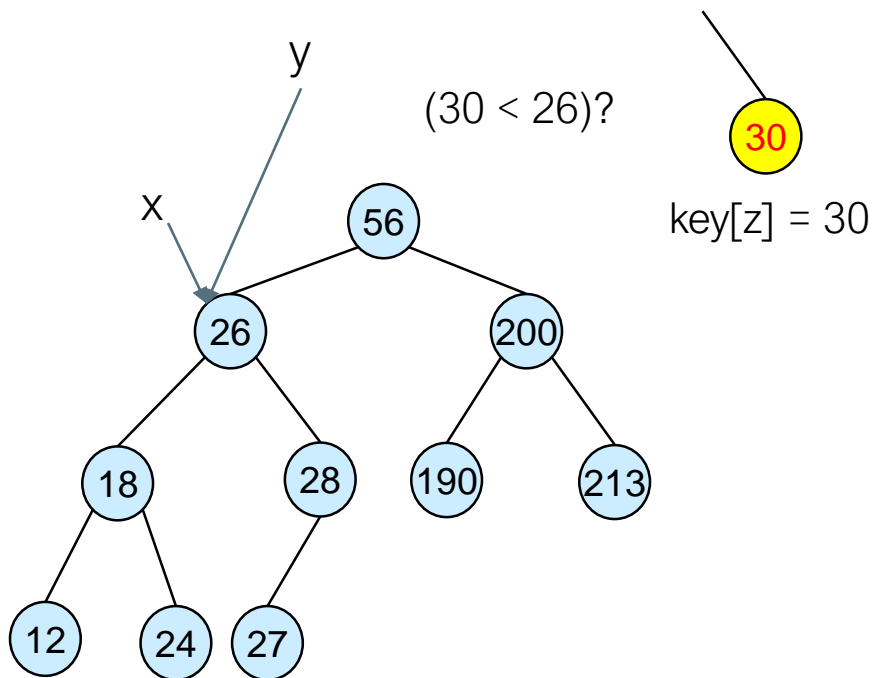
BST Insertion



Tree-Insert(T, z)

1. $y \leftarrow \text{null}$
2. $x \leftarrow \text{root}[T]$
3. **while** $x \neq \text{null}$
4. **do** $y \leftarrow x$
5. **if** $\text{key}[z] < \text{key}[x]$
6. **then** $x \leftarrow \text{left}[x]$
7. **else** $x \leftarrow \text{right}[x]$
8. $p[z] \leftarrow y$
9. **if** $y = \text{null}$
10. **then** $\text{root}[T] \leftarrow z$
11. **else if** $\text{key}[z] < \text{key}[y]$
12. **then** $\text{left}[y] \leftarrow z$
13. **else** $\text{right}[y] \leftarrow z$

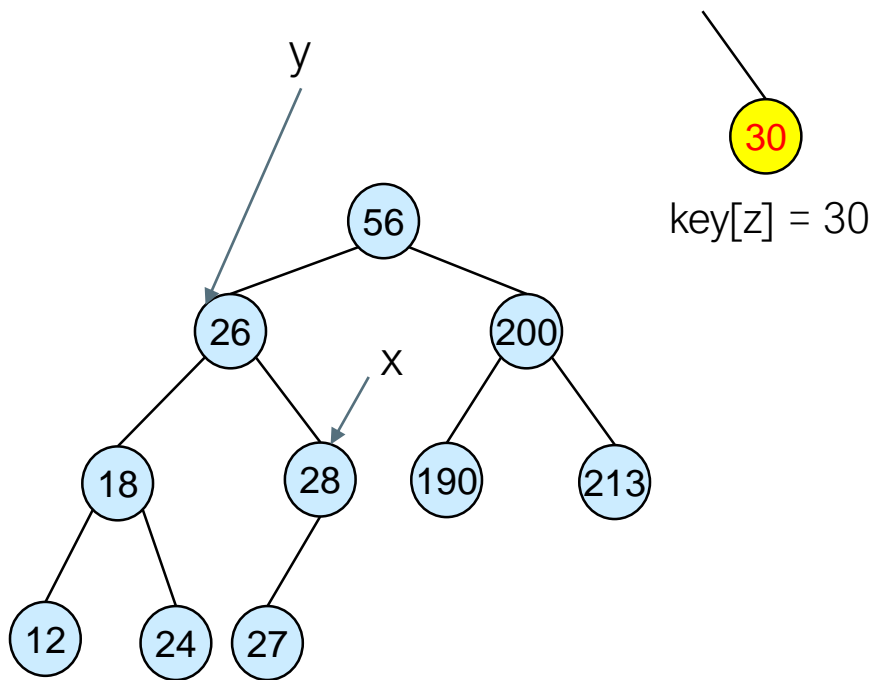
BST Insertion



Tree-Insert(T, z)

1. $y \leftarrow \text{null}$
2. $x \leftarrow \text{root}[T]$
3. **while** $x \neq \text{null}$
4. **do** $y \leftarrow x$
5. **if** $\text{key}[z] < \text{key}[x]$
6. **then** $x \leftarrow \text{left}[x]$
7. **else** $x \leftarrow \text{right}[x]$
8. $p[z] \leftarrow y$
9. **if** $y = \text{null}$
10. **then** $\text{root}[T] \leftarrow z$
11. **else if** $\text{key}[z] < \text{key}[y]$
12. **then** $\text{left}[y] \leftarrow z$
13. **else** $\text{right}[y] \leftarrow z$

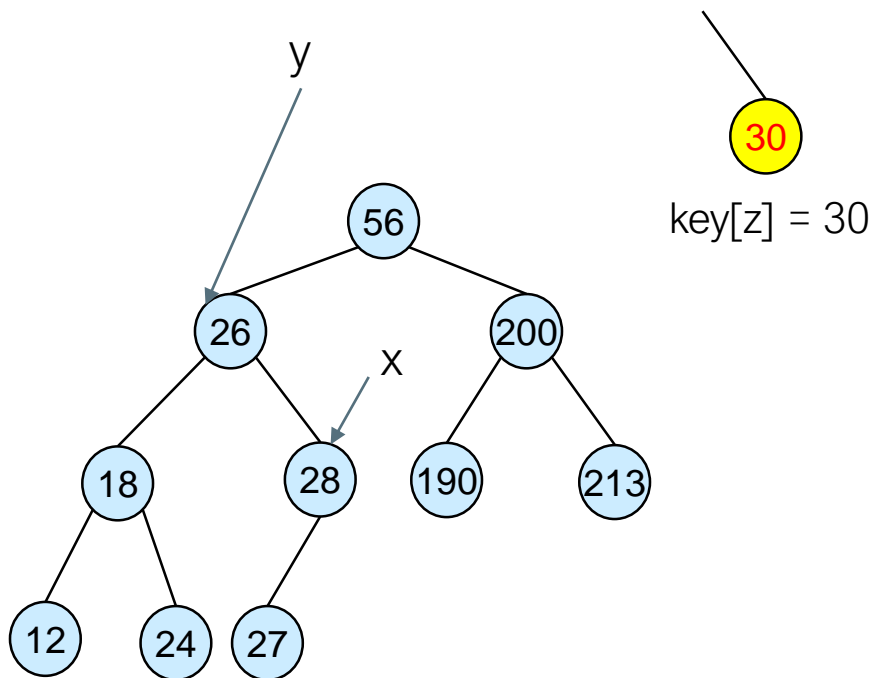
BST Insertion



Tree-Insert(T, z)

1. $y \leftarrow \text{null}$
2. $x \leftarrow \text{root}[T]$
3. **while** $x \neq \text{null}$
4. **do** $y \leftarrow x$
5. **if** $\text{key}[z] < \text{key}[x]$
6. **then** $x \leftarrow \text{left}[x]$
7. **else** $x \leftarrow \text{right}[x]$
8. $p[z] \leftarrow y$
9. **if** $y = \text{null}$
10. **then** $\text{root}[T] \leftarrow z$
11. **else if** $\text{key}[z] < \text{key}[y]$
12. **then** $\text{left}[y] \leftarrow z$
13. **else** $\text{right}[y] \leftarrow z$

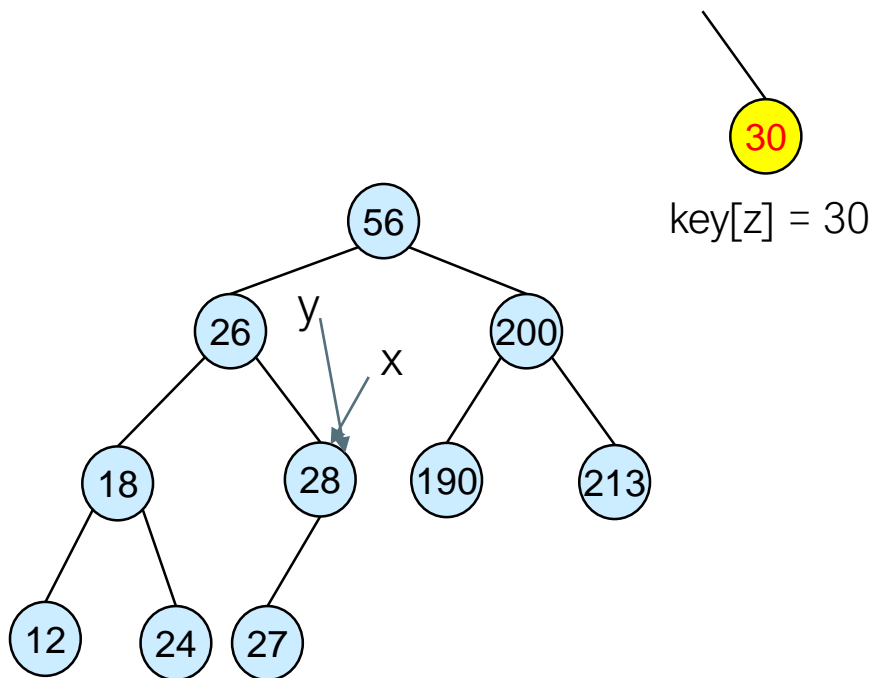
BST Insertion



Tree-Insert(T, z)

1. $y \leftarrow \text{null}$
2. $x \leftarrow \text{root}[T]$
3. **while** $x \neq \text{null}$
4. **do** $y \leftarrow x$
5. **if** $\text{key}[z] < \text{key}[x]$
6. **then** $x \leftarrow \text{left}[x]$
7. **else** $x \leftarrow \text{right}[x]$
8. $p[z] \leftarrow y$
9. **if** $y = \text{null}$
10. **then** $\text{root}[T] \leftarrow z$
11. **else if** $\text{key}[z] < \text{key}[y]$
12. **then** $\text{left}[y] \leftarrow z$
13. **else** $\text{right}[y] \leftarrow z$

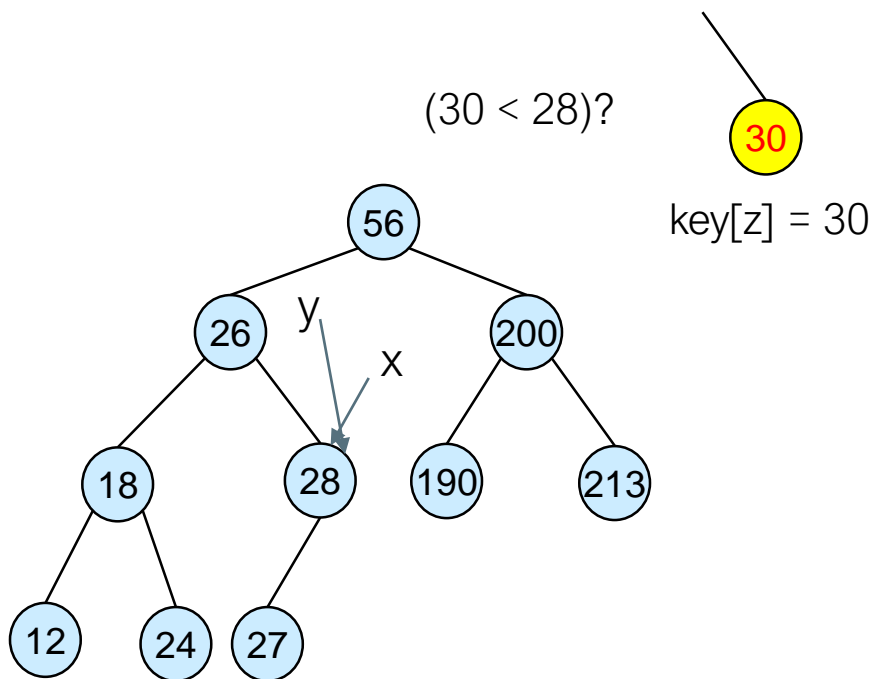
BST Insertion



Tree-Insert(T, z)

1. $y \leftarrow \text{null}$
2. $x \leftarrow \text{root}[T]$
3. **while** $x \neq \text{null}$
4. **do** $y \leftarrow x$
5. **if** $\text{key}[z] < \text{key}[x]$
6. **then** $x \leftarrow \text{left}[x]$
7. **else** $x \leftarrow \text{right}[x]$
8. $p[z] \leftarrow y$
9. **if** $y = \text{null}$
10. **then** $\text{root}[T] \leftarrow z$
11. **else if** $\text{key}[z] < \text{key}[y]$
12. **then** $\text{left}[y] \leftarrow z$
13. **else** $\text{right}[y] \leftarrow z$

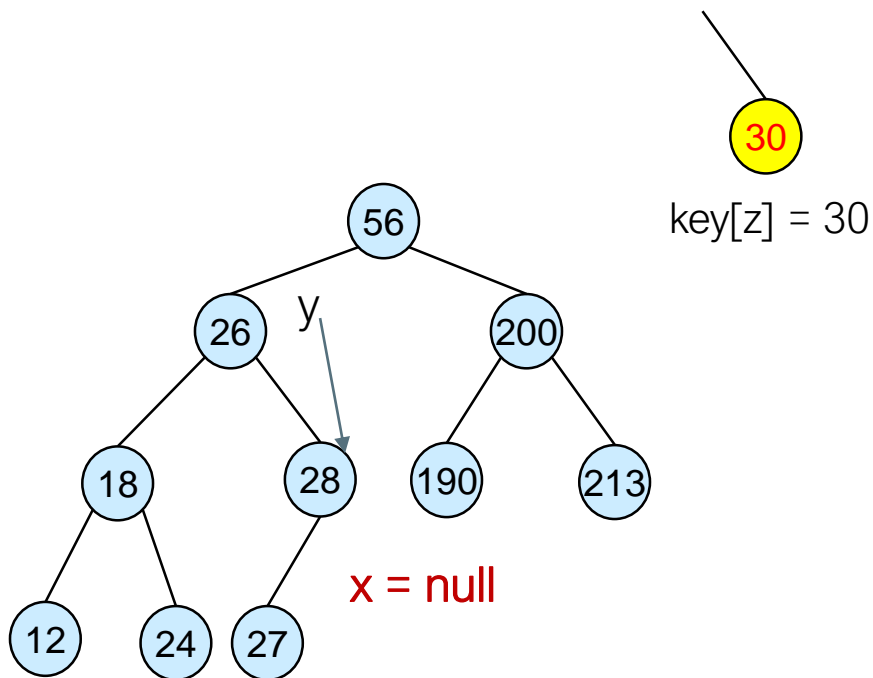
BST Insertion



Tree-Insert(T, z)

1. $y \leftarrow \text{null}$
2. $x \leftarrow \text{root}[T]$
3. **while** $x \neq \text{null}$
4. **do** $y \leftarrow x$
5. **if** $\text{key}[z] < \text{key}[x]$
6. **then** $x \leftarrow \text{left}[x]$
7. **else** $x \leftarrow \text{right}[x]$
8. $p[z] \leftarrow y$
9. **if** $y = \text{null}$
10. **then** $\text{root}[T] \leftarrow z$
11. **else if** $\text{key}[z] < \text{key}[y]$
12. **then** $\text{left}[y] \leftarrow z$
13. **else** $\text{right}[y] \leftarrow z$

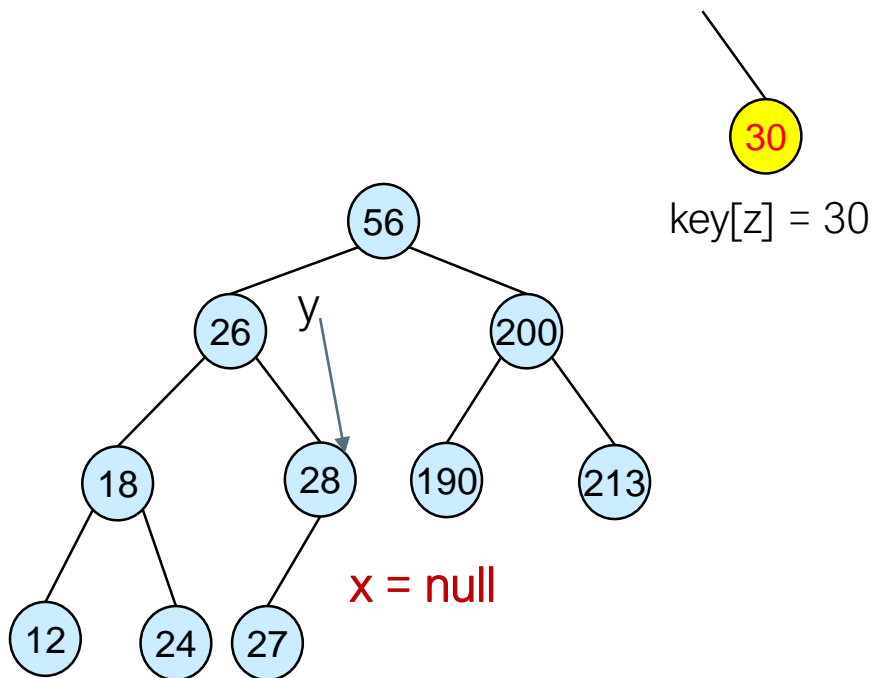
BST Insertion



Tree-Insert(T, z)

1. $y \leftarrow \text{null}$
2. $x \leftarrow \text{root}[T]$
3. **while** $x \neq \text{null}$
4. **do** $y \leftarrow x$
5. **if** $\text{key}[z] < \text{key}[x]$
6. **then** $x \leftarrow \text{left}[x]$
7. **else** $x \leftarrow \text{right}[x]$
8. $p[z] \leftarrow y$
9. **if** $y = \text{null}$
10. **then** $\text{root}[T] \leftarrow z$
11. **else if** $\text{key}[z] < \text{key}[y]$
12. **then** $\text{left}[y] \leftarrow z$
13. **else** $\text{right}[y] \leftarrow z$

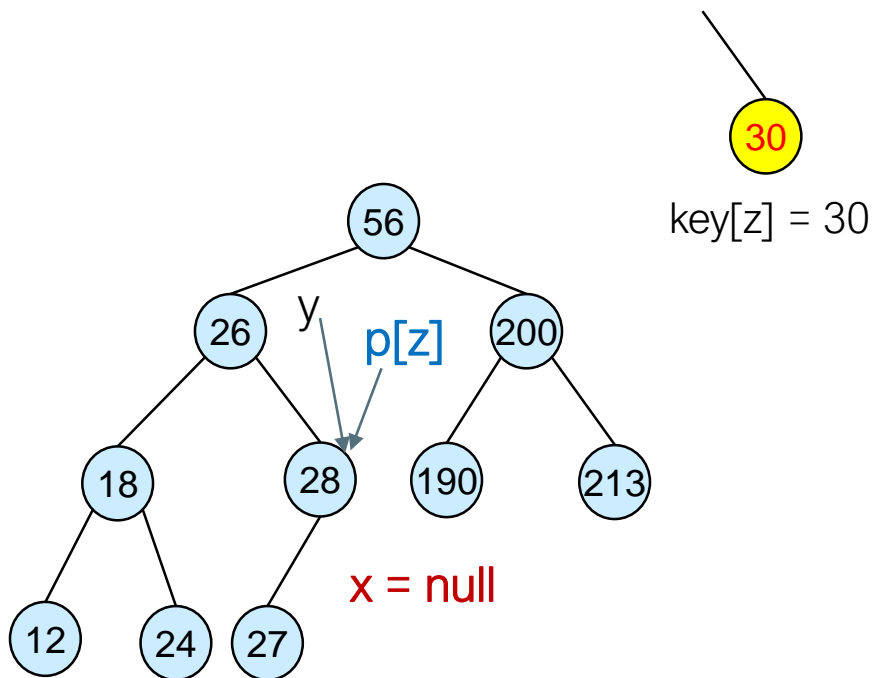
BST Insertion



Tree-Insert(T, z)

1. $y \leftarrow \text{null}$
2. $x \leftarrow \text{root}[T]$
3. **while** $x \neq \text{null}$
4. **do** $y \leftarrow x$
5. **if** $\text{key}[z] < \text{key}[x]$
6. **then** $x \leftarrow \text{left}[x]$
7. **else** $x \leftarrow \text{right}[x]$
8. $p[z] \leftarrow y$
9. **if** $y = \text{null}$
10. **then** $\text{root}[T] \leftarrow z$
11. **else if** $\text{key}[z] < \text{key}[y]$
12. **then** $\text{left}[y] \leftarrow z$
13. **else** $\text{right}[y] \leftarrow z$

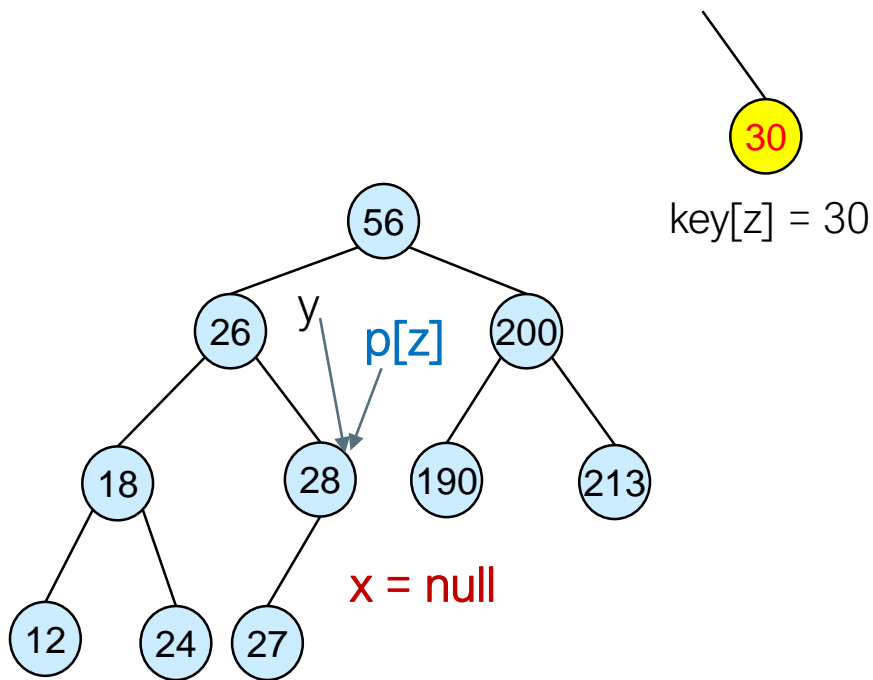
BST Insertion



Tree-Insert(T, z)

1. $y \leftarrow \text{null}$
2. $x \leftarrow \text{root}[T]$
3. **while** $x \neq \text{null}$
4. **do** $y \leftarrow x$
5. **if** $\text{key}[z] < \text{key}[x]$
6. **then** $x \leftarrow \text{left}[x]$
7. **else** $x \leftarrow \text{right}[x]$
8. $p[z] \leftarrow y$ (optional)
9. **if** $y = \text{null}$
10. **then** $\text{root}[T] \leftarrow z$
11. **else if** $\text{key}[z] < \text{key}[y]$
12. **then** $\text{left}[y] \leftarrow z$
13. **else** $\text{right}[y] \leftarrow z$

BST Insertion

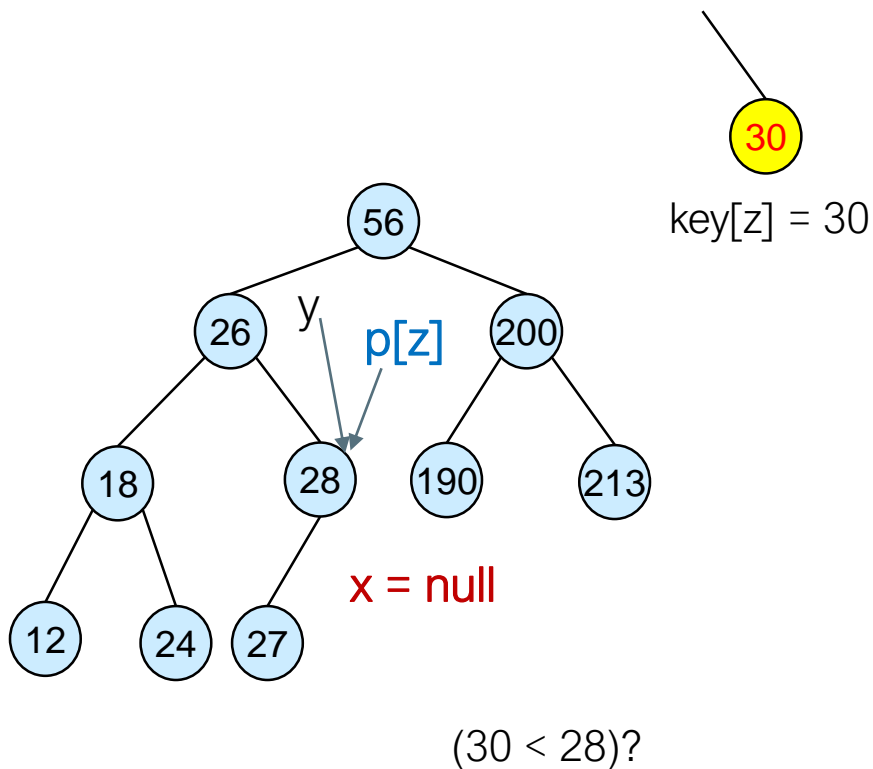


(is the tree T empty)?

Tree-Insert(T, z)

1. $y \leftarrow \text{null}$
2. $x \leftarrow \text{root}[T]$
3. **while** $x \neq \text{null}$
4. **do** $y \leftarrow x$
5. **if** $\text{key}[z] < \text{key}[x]$
6. **then** $x \leftarrow \text{left}[x]$
7. **else** $x \leftarrow \text{right}[x]$
8. $p[z] \leftarrow y$
9. **if** $y = \text{null}$
10. **then** $\text{root}[T] \leftarrow z$
11. **else if** $\text{key}[z] < \text{key}[y]$
12. **then** $\text{left}[y] \leftarrow z$
13. **else** $\text{right}[y] \leftarrow z$

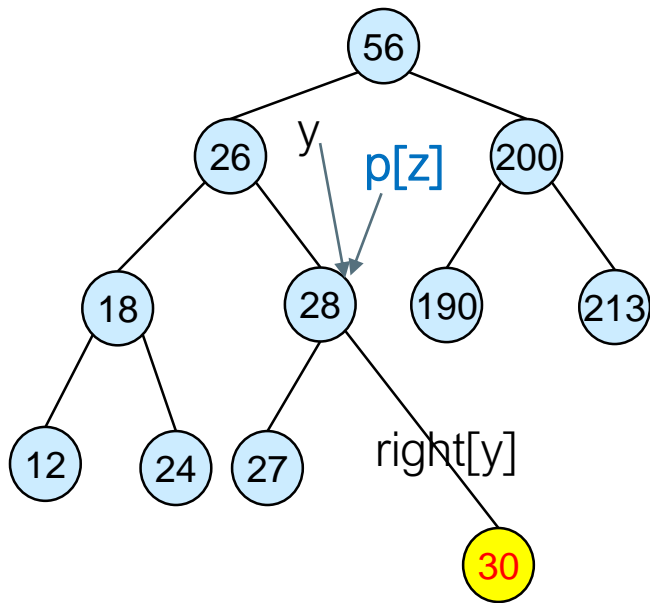
BST Insertion



Tree-Insert(T, z)

1. $y \leftarrow \text{null}$
2. $x \leftarrow \text{root}[T]$
3. **while** $x \neq \text{null}$
4. **do** $y \leftarrow x$
5. **if** $\text{key}[z] < \text{key}[x]$
6. **then** $x \leftarrow \text{left}[x]$
7. **else** $x \leftarrow \text{right}[x]$
8. $p[z] \leftarrow y$
9. **if** $y = \text{null}$
10. **then** $\text{root}[T] \leftarrow z$
11. **else if** $\text{key}[z] < \text{key}[y]$
12. **then** $\text{left}[y] \leftarrow z$
13. **else** $\text{right}[y] \leftarrow z$

BST Insertion



x is used to find the position where we wish to place z

Note that $p[z]$ is optional

Tree-Insert(T, z)

1. $y \leftarrow \text{null}$
2. $x \leftarrow \text{root}[T]$
3. **while** $x \neq \text{null}$
4. **do** $y \leftarrow x$
5. **if** $\text{key}[z] < \text{key}[x]$
6. **then** $x \leftarrow \text{left}[x]$
7. **else** $x \leftarrow \text{right}[x]$
8. $p[z] \leftarrow y$
9. **if** $y = \text{null}$
10. **then** $\text{root}[T] \leftarrow z$
11. **else if** $\text{key}[z] < \text{key}[y]$
12. **then** $\text{left}[y] \leftarrow z$
13. **else** $\text{right}[y] \leftarrow z$

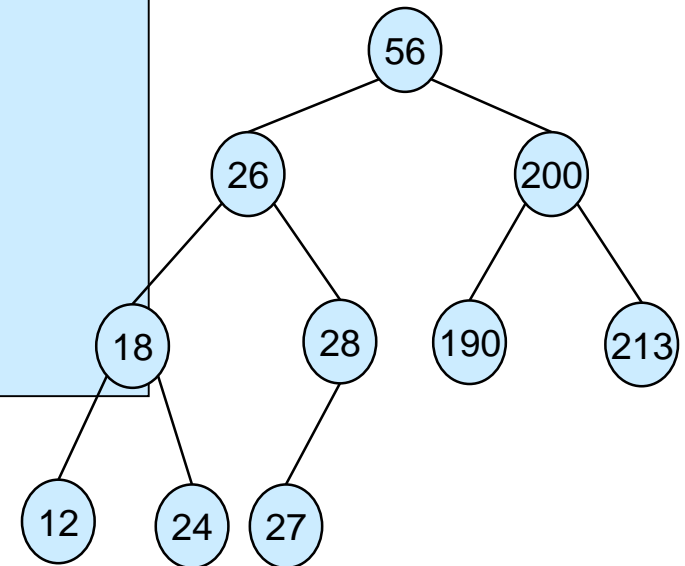
Predecessor and Successor

- Successor of node x is the node y such that $\text{key}[y]$ is the smallest key greater than $\text{key}[x]$
- The successor of the largest key is NIL
- Search consists of two cases:
 - If node x has a non-empty right subtree, then x 's successor is the minimum in the right subtree of x
 - If node x has an empty right subtree, then:
 - As long as we move to the left up the tree (move up through right children), we are visiting smaller keys
 - x 's successor y is the node that x is the predecessor of (x is the maximum in y 's left subtree)
 - In other words, x 's successor y , is the lowest ancestor of x whose left child is also an ancestor of x

Pseudo-code for Successor

Tree-Successor(*x*)

1. **if** $right[x] \neq \text{null}$
2. **then** return Tree-Minimum($right[x]$)
3. $y \leftarrow p[x]$
4. **while** $y \neq \text{null}$ **and** $x = right[y]$
5. **do** $x \leftarrow y$
6. $y \leftarrow p[y]$
7. **return** y

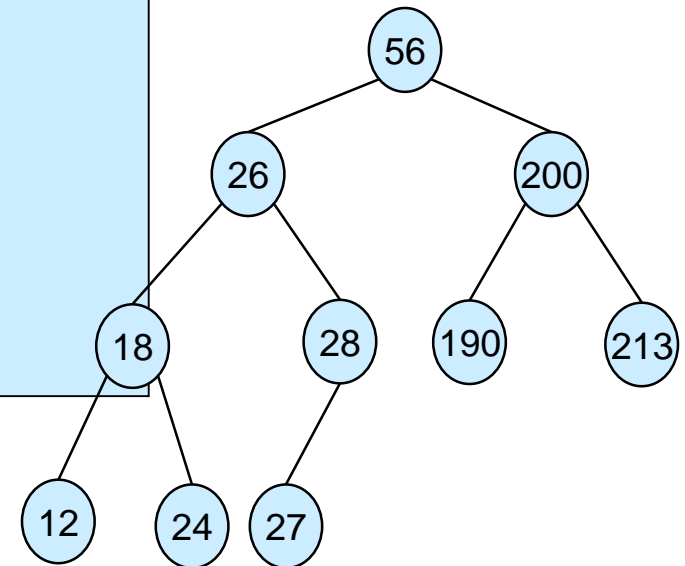


Code for **predecessor** is symmetric.

Pseudo-code for Successor

Tree-Successor(x)

1. **if** $right[x] \neq \text{null}$
2. **then** return Tree-Minimum($right[x]$)
3. $y \leftarrow p[x]$
4. **while** $y \neq \text{null}$ **and** $x = right[y]$
5. **do** $x \leftarrow y$
6. $y \leftarrow p[y]$
7. **return** y



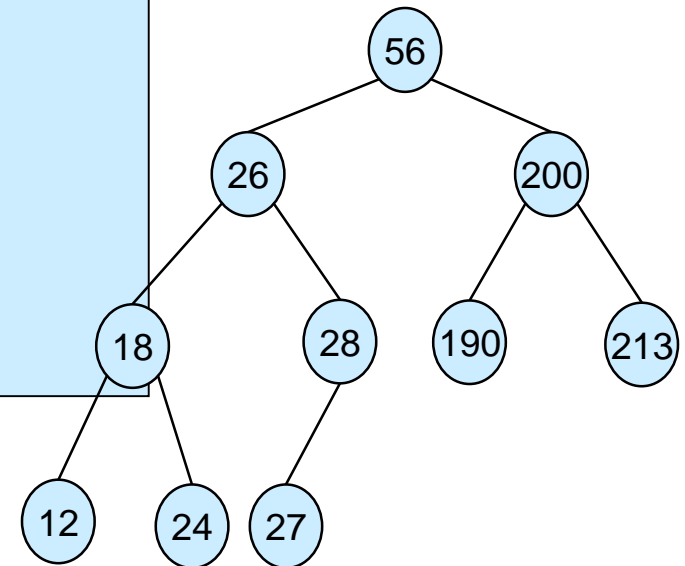
What's the successor
of 56?

Pseudo-code for Successor

Tree-Successor(*x*)

1. **if** *right*[*x*] \neq null
2. **then** return Tree-Minimum(*right*[*x*])
3. *y* \leftarrow *p*[*x*]
4. **while** *y* \neq null **and** *x* = *right*[*y*]
5. **do** *x* \leftarrow *y*
6. *y* \leftarrow *p*[*y*]
7. **return** *y*

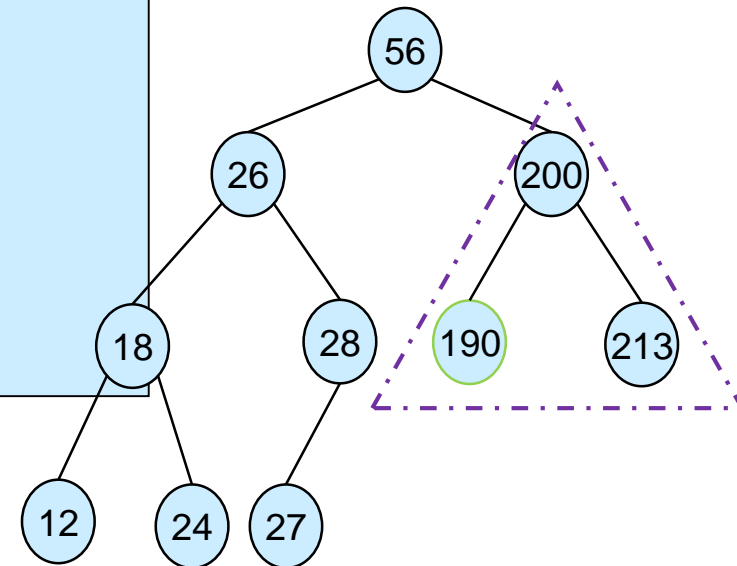
What's the successor
of 56?



Pseudo-code for Successor

Tree-Successor(x)

1. **if** $right[x] \neq \text{null}$
2. **then** **return** Tree-Minimum($right[x]$)
3. $y \leftarrow p[x]$
4. **while** $y \neq \text{null}$ **and** $x = right[y]$
5. **do** $x \leftarrow y$
6. $y \leftarrow p[y]$
7. **return** y



What is the successor
of 56?

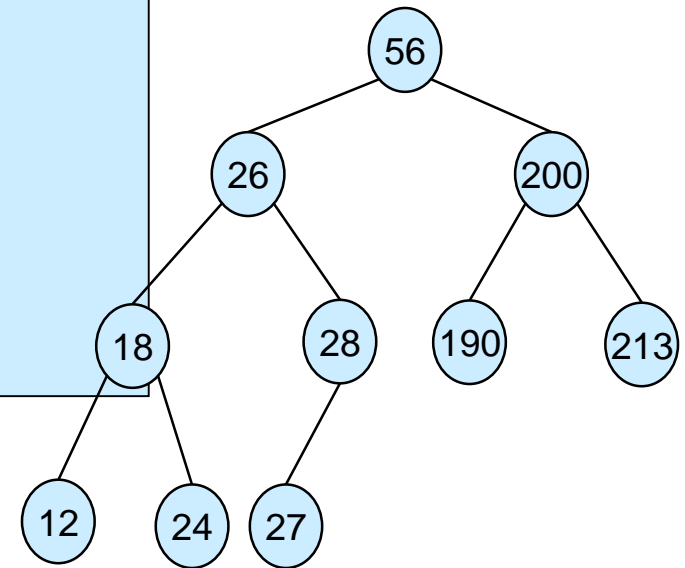
It is the leftmost node in the right subtree!

Pseudo-code for Successor

Tree-Successor(*x*)

1. **if** $right[x] \neq \text{null}$
2. **then** return Tree-Minimum($right[x]$)
3. $y \leftarrow p[x]$
4. **while** $y \neq \text{null}$ **and** $x = right[y]$
5. **do** $x \leftarrow y$
6. $y \leftarrow p[y]$
7. **return** y

What is the successor of 28?

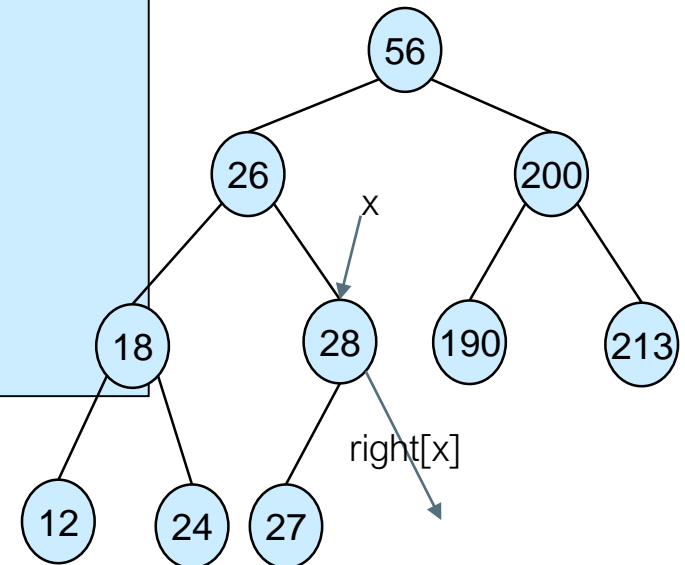


Pseudo-code for Successor

Tree-Successor(x)

1. **if** $right[x] \neq \text{null}$
2. **then** return Tree-Minimum($right[x]$)
3. $y \leftarrow p[x]$
4. **while** $y \neq \text{null}$ **and** $x = right[y]$
5. **do** $x \leftarrow y$
6. $y \leftarrow p[y]$
7. **return** y

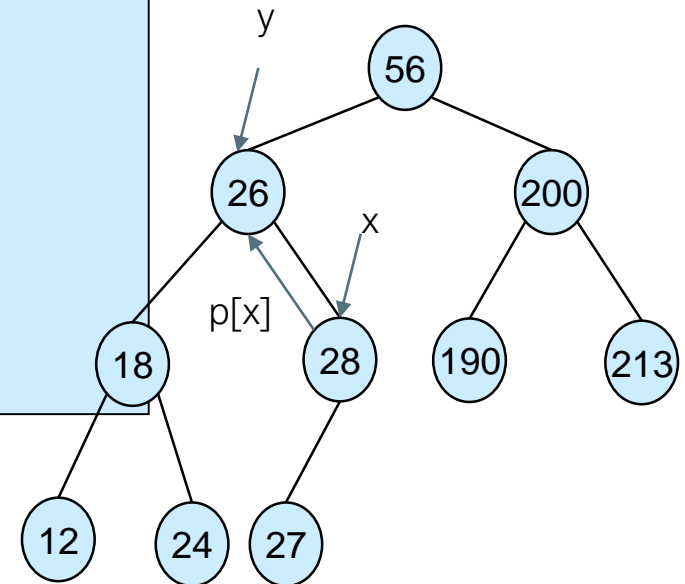
What is the successor of 28?



Pseudo-code for Successor

Tree-Successor(x)

1. **if** $right[x] \neq \text{null}$
2. **then** return Tree-Minimum($right[x]$)
3. $y \leftarrow p[x]$
4. **while** $y \neq \text{null}$ **and** $x = right[y]$
5. **do** $x \leftarrow y$
6. $y \leftarrow p[y]$
7. **return** y

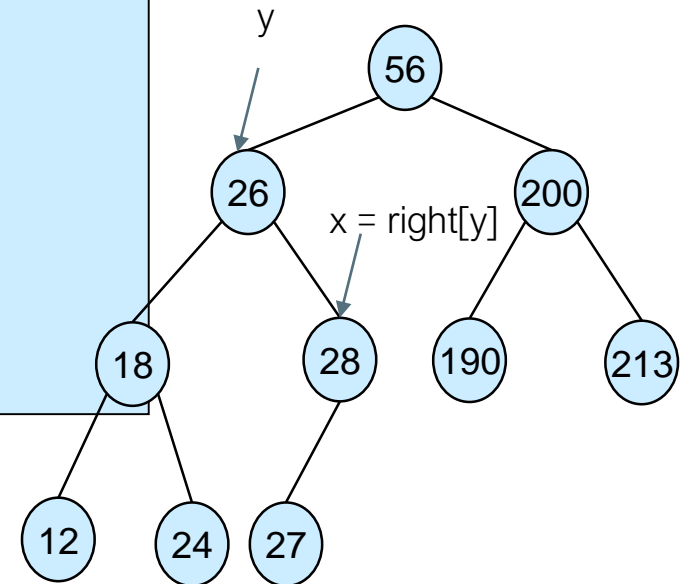


What is the successor of 28?

Pseudo-code for Successor

Tree-Successor(*x*)

1. **if** $right[x] \neq \text{null}$
2. **then** return Tree-Minimum($right[x]$)
3. $y \leftarrow p[x]$
4. **while** $y \neq \text{null}$ **and** $x = right[y]$
5. **do** $x \leftarrow y$
6. $y \leftarrow p[y]$
7. **return** y

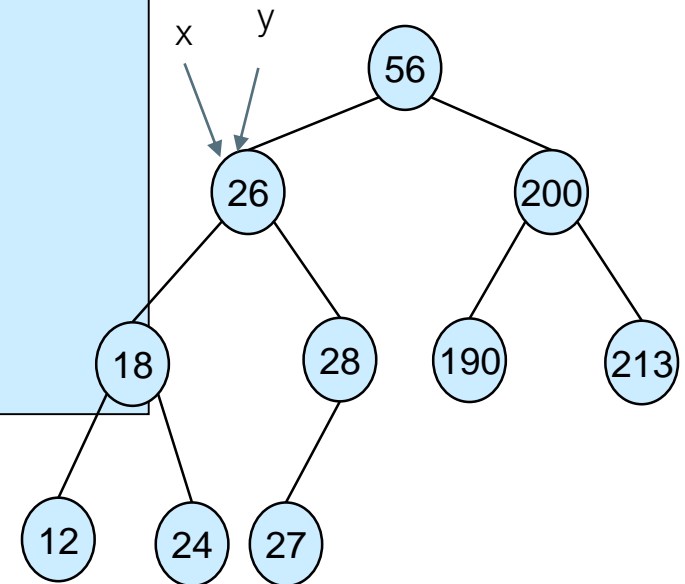


What is the successor of 28?

Pseudo-code for Successor

Tree-Successor(*x*)

1. **if** $right[x] \neq \text{null}$
2. **then** return Tree-Minimum($right[x]$)
3. $y \leftarrow p[x]$
4. **while** $y \neq \text{null}$ **and** $x = right[y]$
5. **do** $x \leftarrow y$
6. $y \leftarrow p[y]$
7. **return** y

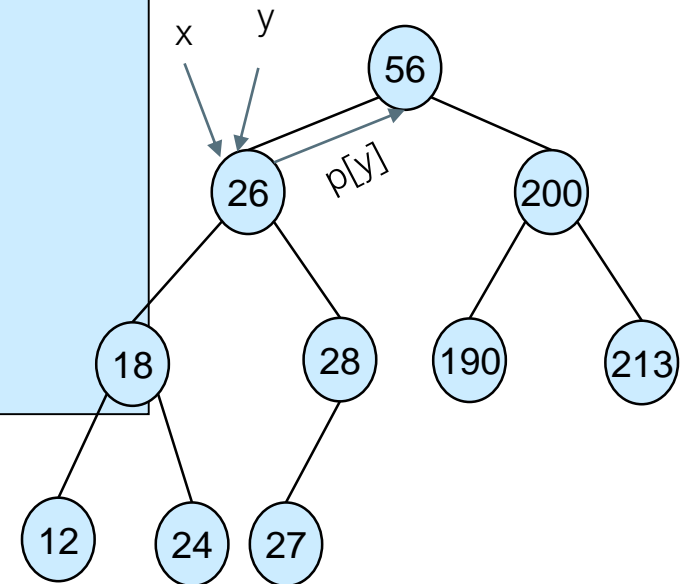


What is the successor of 28?

Pseudo-code for Successor

Tree-Successor(*x*)

1. **if** $right[x] \neq \text{null}$
2. **then** return Tree-Minimum($right[x]$)
3. $y \leftarrow p[x]$
4. **while** $y \neq \text{null}$ **and** $x = right[y]$
5. **do** $x \leftarrow y$
6. $y \leftarrow p[y]$
7. **return** y

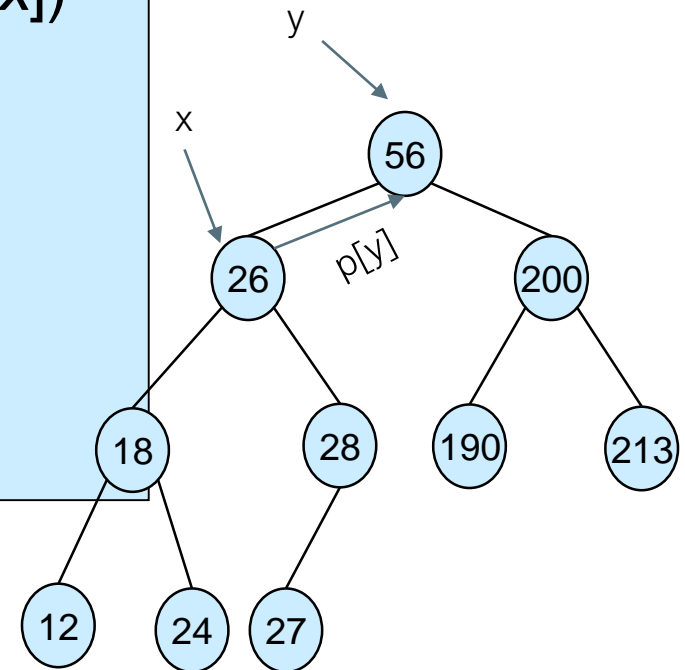


What is the successor of 28?

Pseudo-code for Successor

Tree-Successor(*x*)

1. **if** $right[x] \neq \text{null}$
2. **then** return Tree-Minimum($right[x]$)
3. $y \leftarrow p[x]$
4. **while** $y \neq \text{null}$ **and** $x = right[y]$
5. **do** $x \leftarrow y$
6. $y \leftarrow p[y]$
7. **return** y



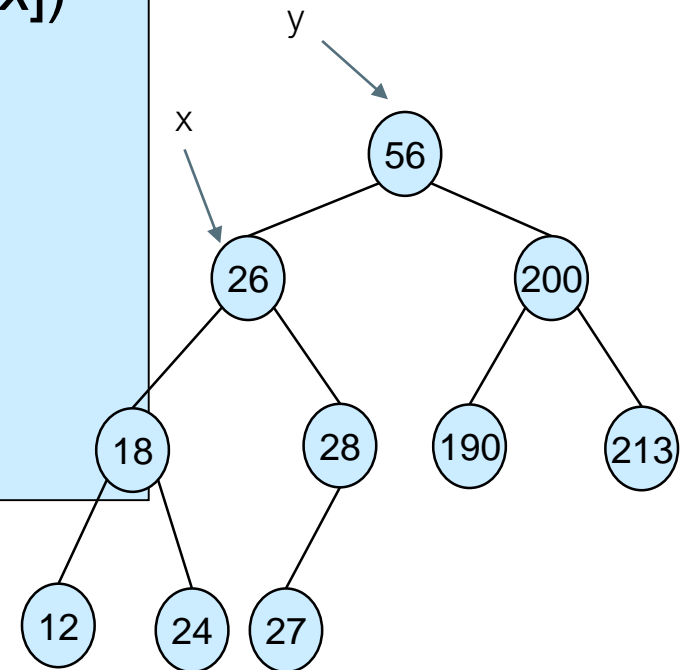
What is the successor of 28?

Pseudo-code for Successor

Tree-Successor(*x*)

1. if $right[x] \neq \text{null}$
2. **then** return Tree-Minimum($right[x]$)
3. $y \leftarrow p[x]$
4. **while** $y \neq \text{null}$ **and** $x = right[y]$
5. **do** $x \leftarrow y$
6. $y \leftarrow p[y]$
7. **return** y

What is the successor of 28?



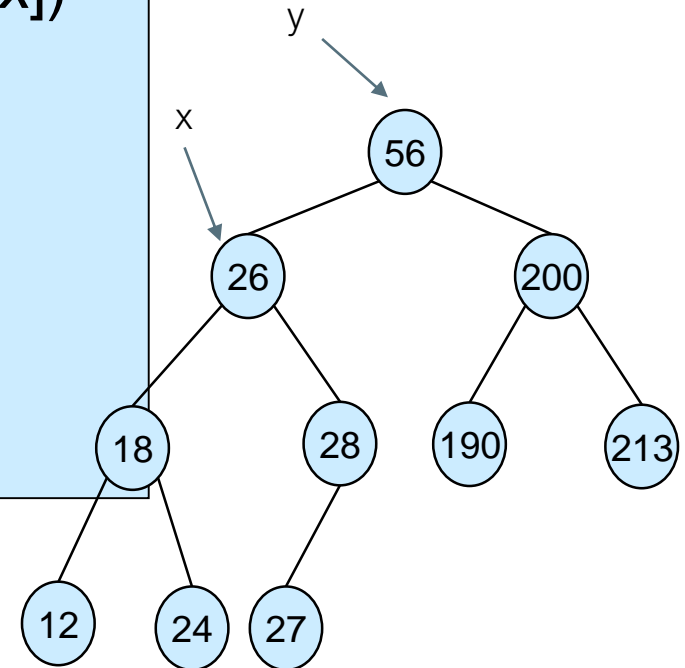
Pseudo-code for Successor

Tree-Successor(*x*)

1. **if** $right[x] \neq \text{null}$
2. **then** return Tree-Minimum($right[x]$)
3. $y \leftarrow p[x]$
4. **while** $y \neq \text{null}$ **and** $x = right[y]$
5. **do** $x \leftarrow y$
6. $y \leftarrow p[y]$
7. **return** y

What is the successor of 28?

56 is the successor of 28!



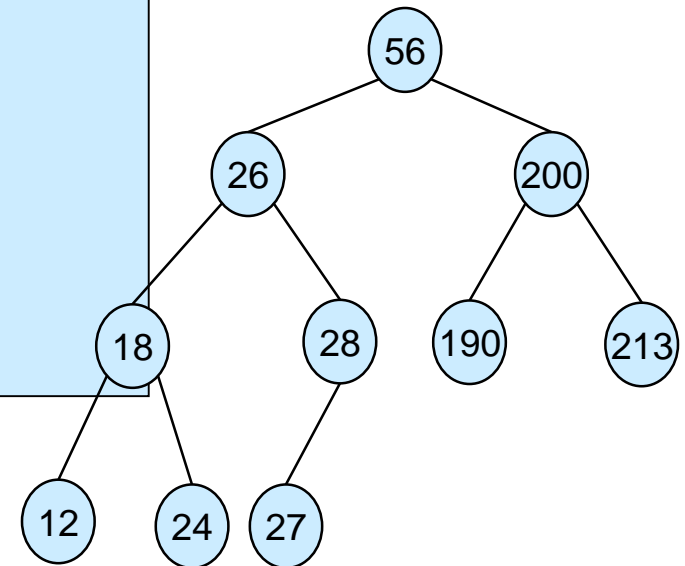
Pseudo-code for Successor

Tree-Successor(*x*)

1. **if** $right[x] \neq \text{null}$
2. **then** return Tree-Minimum($right[x]$)
3. $y \leftarrow p[x]$
4. **while** $y \neq \text{null}$ **and** $x = right[y]$
5. **do** $x \leftarrow y$
6. $y \leftarrow p[y]$
7. **return** y

Code for **predecessor** is symmetric.

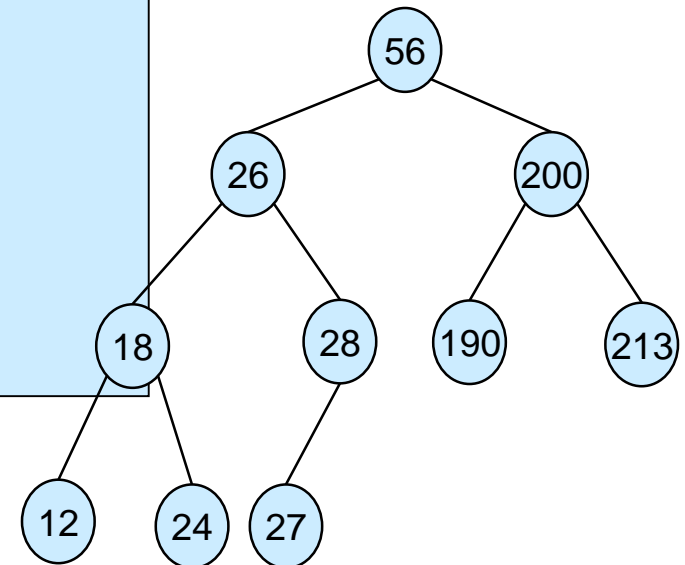
What must be changed to find the predecessor?



Pseudo-code for Predecessor

Tree-Predecessor(x)

1. if $left[x] \neq \text{null}$
2. then return **Tree-Maximum**($left[x]$)
3. $y \leftarrow p[x]$
4. while $y \neq \text{null}$ and $x = left[y]$
5. do $x \leftarrow y$
6. $y \leftarrow p[y]$
7. return y

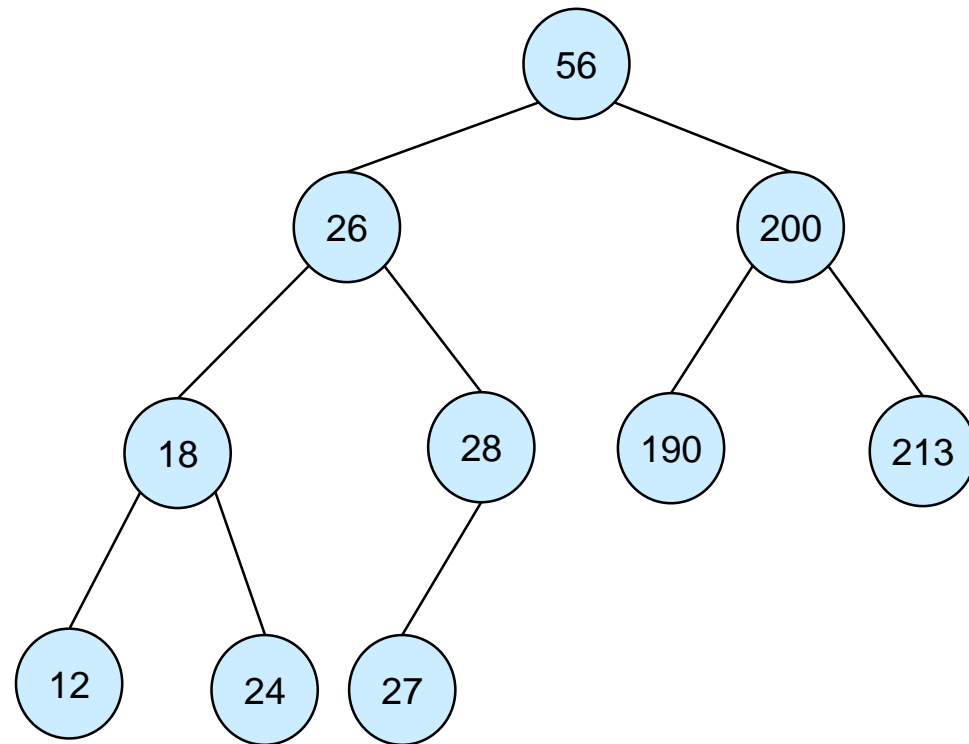


successor and predecessor are symmetric.

Tree-Delete (T, x)

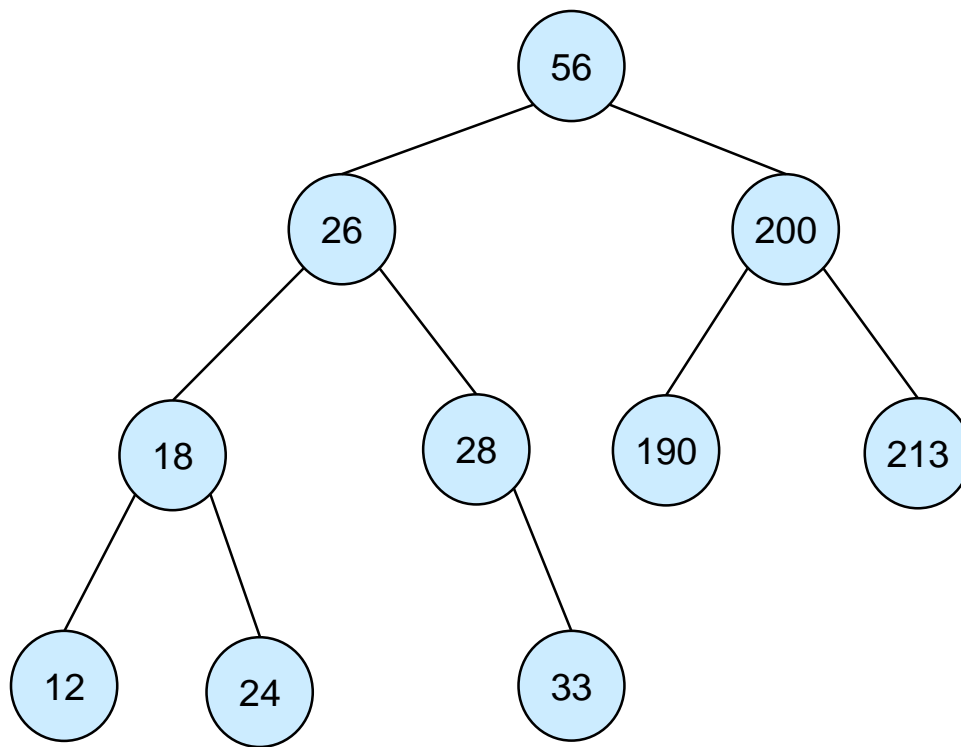
1. If x has no children ◆ case 0
2. If x has one child ◆ case 1
3. If x has two children (subtrees) ◆ case 2

Delete: Examples



- Case 0
 - Remove 12
- Case 1
 - Remove 28
 - $p[27]=26$
- Case 2
 - Remove 26
 - Replace 26 with 27
 - Remove 27 from subtree

Delete: Examples 2



- Case 2
 - Remove 26
 - Replace 26 with 28
 - Remove 28 from subtree (case 1)

Tree-Delete (T, x)

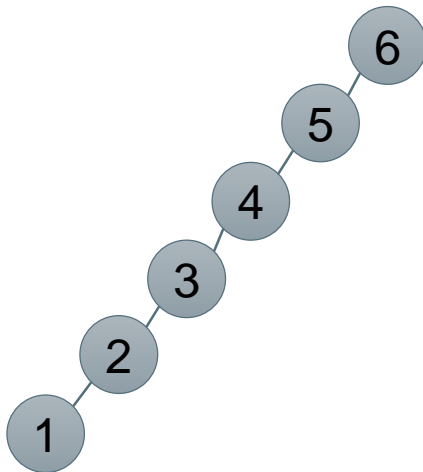
1. If x has no children ♦ case 0
 - then remove x
2. If x has one child ♦ case 1
 - then make $p[x.child]$ point to $p[x]$
3. If x has two children (subtrees) ♦ case 2
 - then replace x with its successor
 - Delete successor in subtree // case 0 or 1

Red-Black Tree

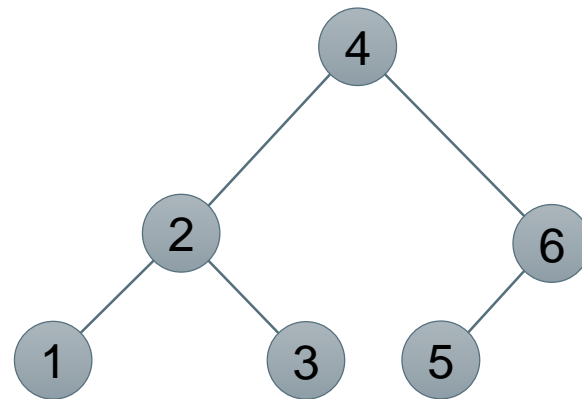
Close to Balanced Tree

Worse case scenario

- Insert 6, 5, 4, 3, 2, 1 in an empty tree in order
- Depth of tree linear increases



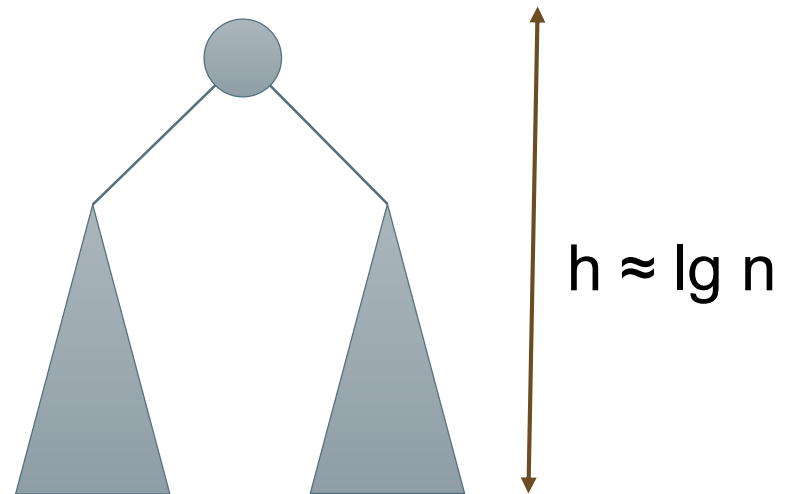
Unbalanced tree



Balanced tree

Balanced Tree

- Balanced search tree
 - Belong to binary search tree
 - But with a height of $O(\lg n)$ guaranteed for n items
 - Height = maximum # edges from root to node
- Examples
 - AVL trees
 - Red-black trees



Red-Black Tree

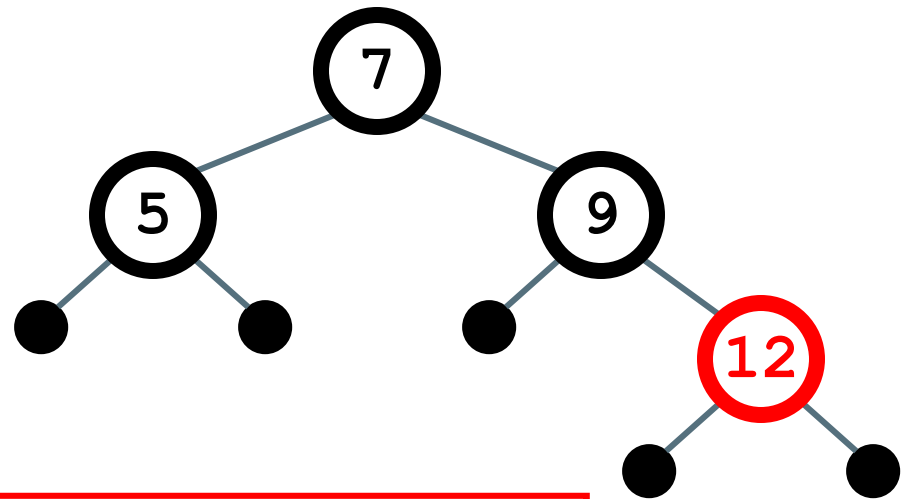
- Close to **balanced tree**
- Tree structure requires an extra **one-bit color field** in each node: either **red** or **black**
- **Color** is used to maintain balance
- Node:
 - Key
 - Color
 - Left
 - Right
 - Parent
 - (Data)

Red-Black Properties

- The **red-black properties**:
 1. Every node is either **red** or **black**
 2. (a) Root and (b) leaves (NULL node) are **black**
 - Note: this means every “real” node has 2 children
 3. If a node is **red**, both children are **black**
 - Note: can’t have 2 consecutive **reds** on a path
 4. Every path from node to descendent leaf contains the same number of **black** nodes

Red-Black Trees: An Example

- Valid example:

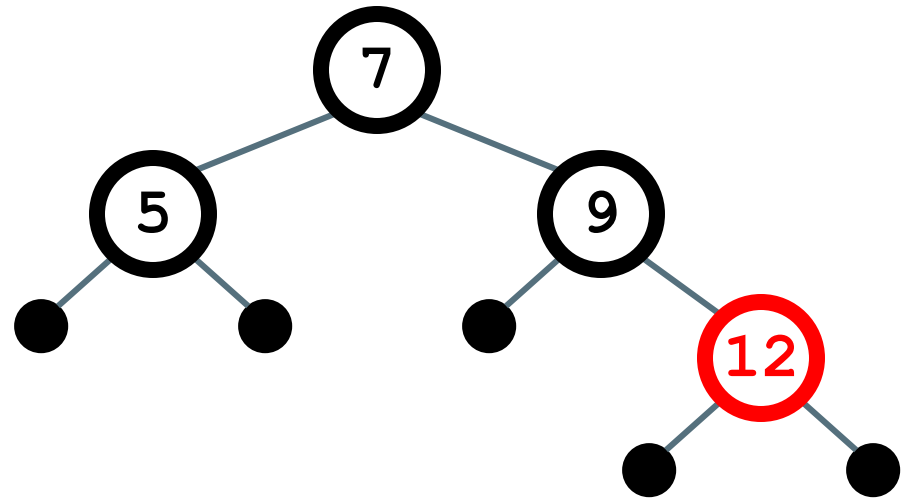


Red-black properties:

1. Every node is either red or black
2. Root and every leaf (NULL pointer) are black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes

The Problem With Insertion

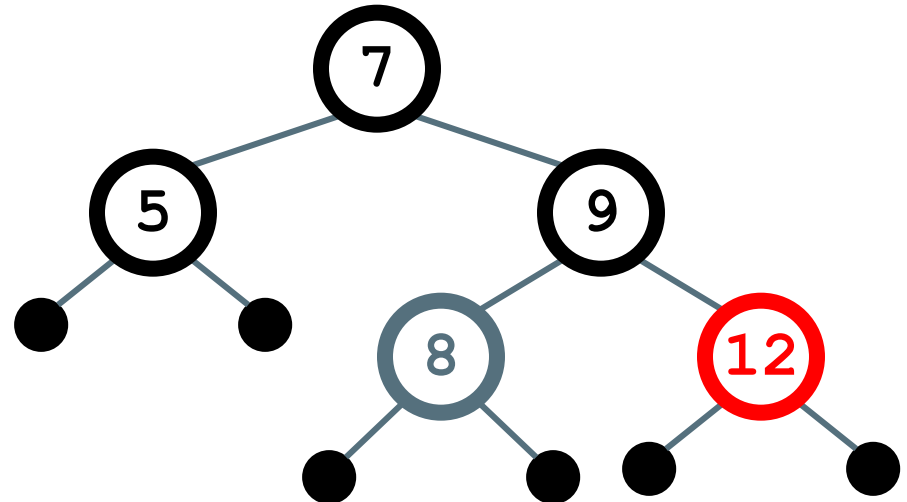
- Insert 8
 - Where does it go?



1. Every node is either red or black
2. Root and every leaf (NULL pointer) are black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes

The Problem With Insertion

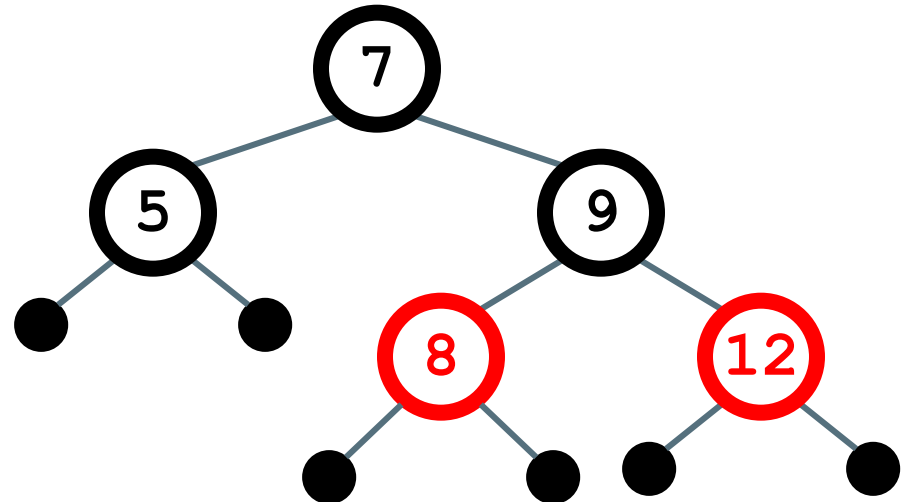
- Insert 8
 - Where does it go?
 - What color should it be?



1. Every node is either red or black
2. Root and every leaf (NULL pointer) are black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes

The Problem With Insertion

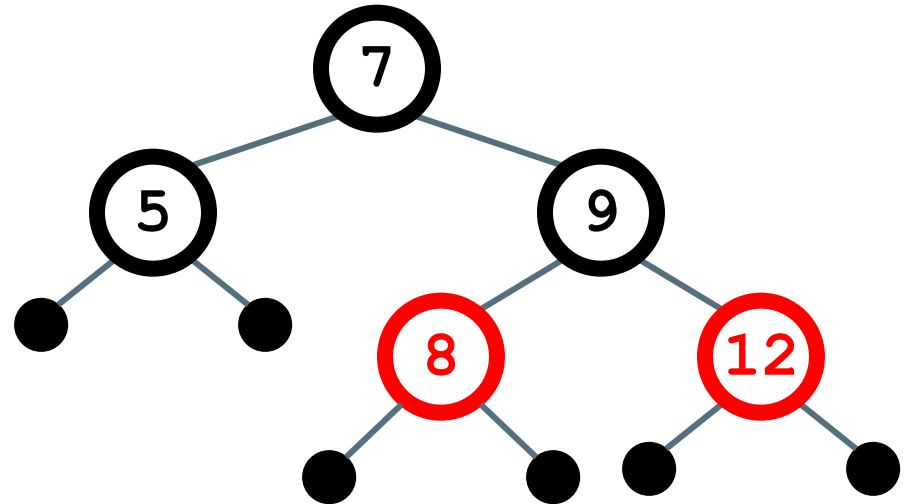
- Insert 8
 - Where does it go?
 - What color should it be?



1. Every node is either red or black
2. Root and every leaf (NULL pointer) are black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes

The Problem With Insertion

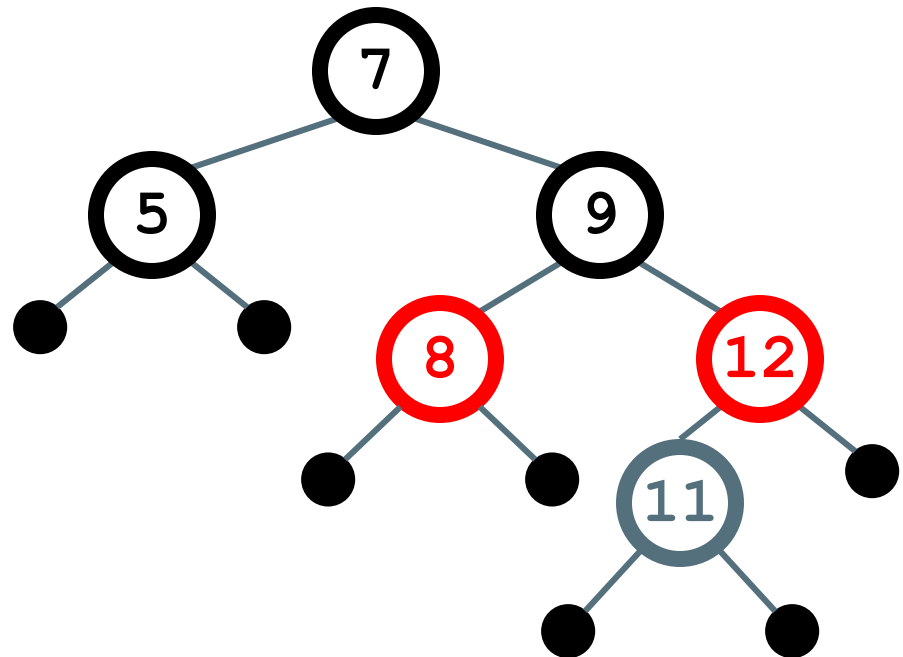
- Insert 11
 - Where does it go?



1. Every node is either red or black
2. Root and every leaf (NULL pointer) are black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes

The Problem With Insertion

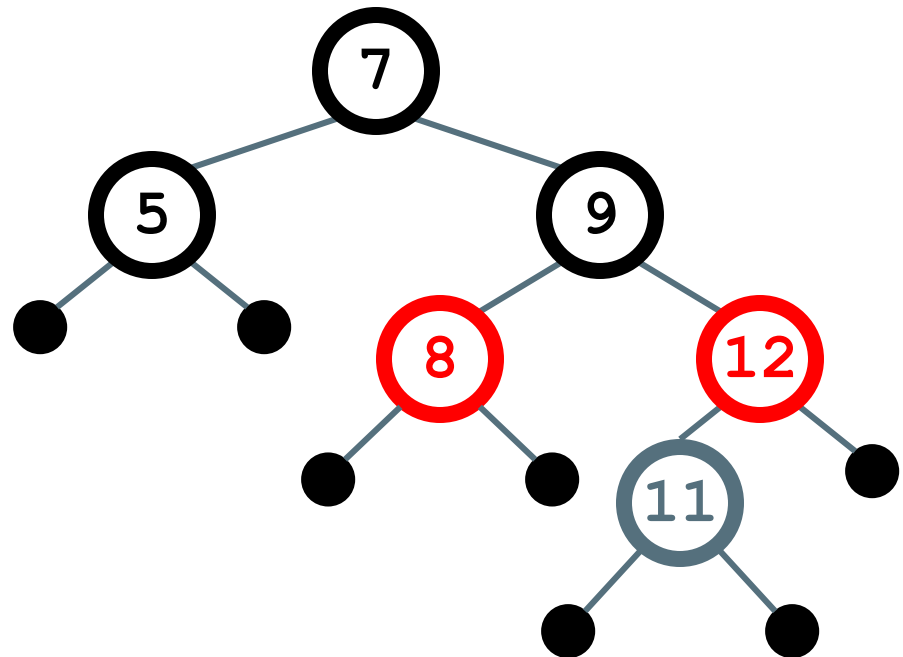
- Insert 11
 - Where does it go?
 - What color?



1. Every node is either red or black
2. Root and every leaf (NULL pointer) are black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes

The Problem With Insertion

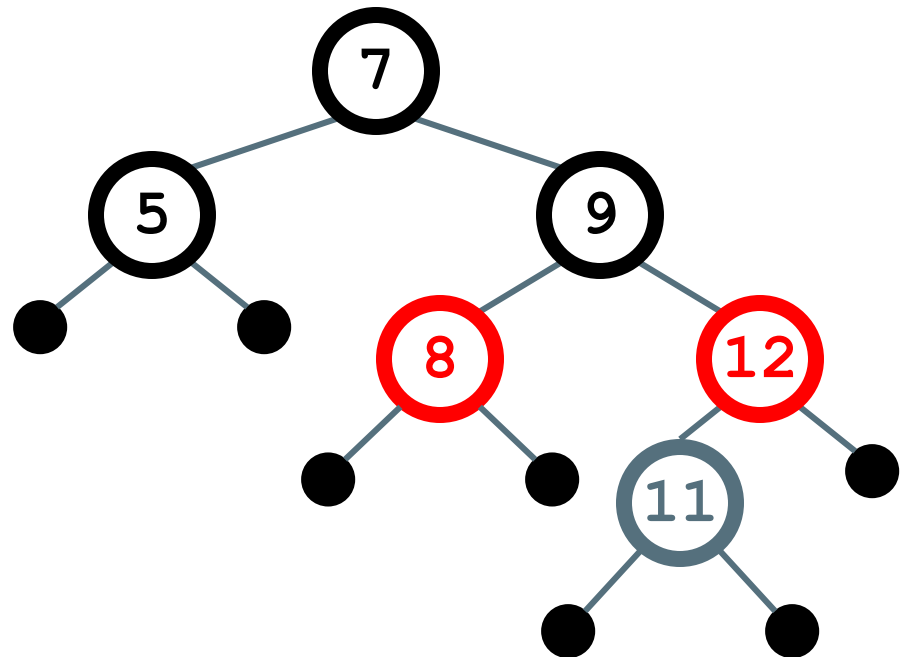
- Insert 11
 - Where does it go?
 - What color?
 - Can't be red! (#3)



1. Every node is either red or black
2. Root and every leaf (NULL pointer) are black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes

The Problem With Insertion

- Insert 11
 - Where does it go?
 - What color?
 - Can't be red! (#3)
 - Can't be black! (#4)



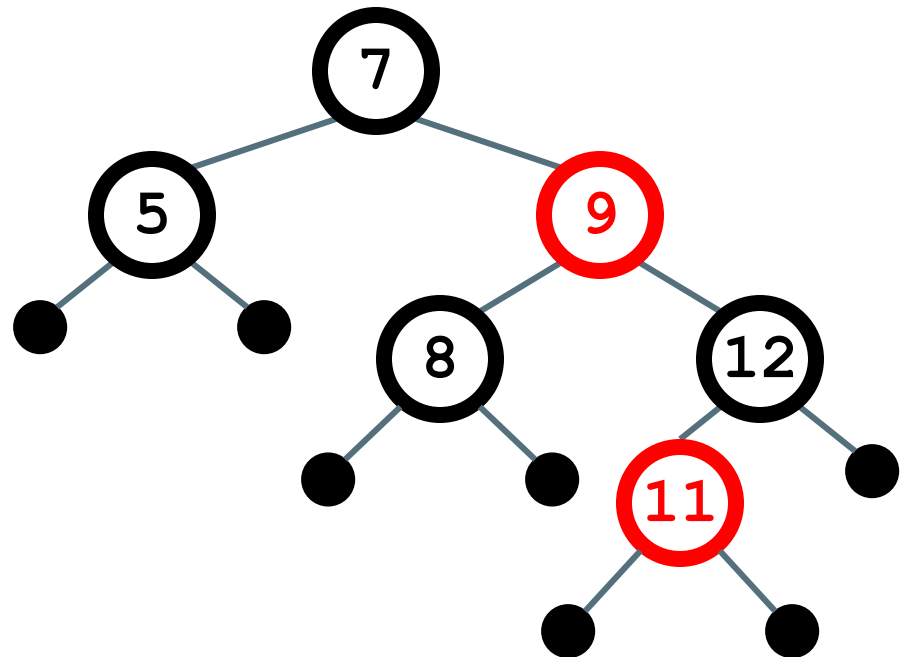
1. Every node is either red or black
2. Root and every leaf (NULL pointer) are black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes

Red-Black Trees: Insertion

- Insertion: the basic idea
 - Insert x into tree, color x red
 - Only r-b property 3 might be violated
 - It will be violated if $p[x]$ is red
 - If so, move violation up tree until a place is found where it can be fixed
 - Total time will be $O(\lg n)$

The Problem With Insertion

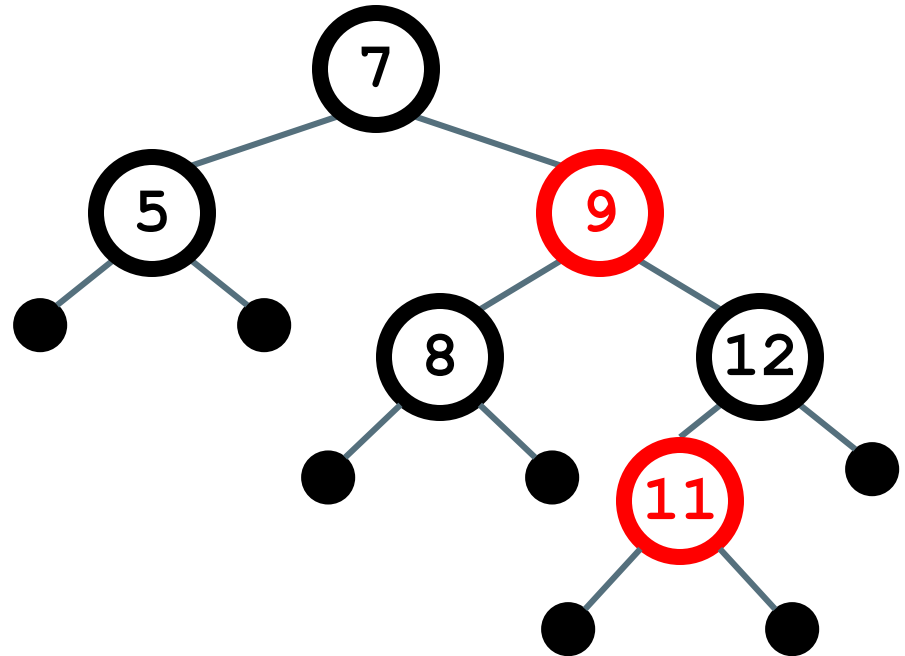
- Insert 11
 - Where does it go?
 - What color?
 - Solution:
recolor the tree



1. Every node is either red or black
2. Root and every leaf (NULL pointer) are black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes

The Problem With Insertion

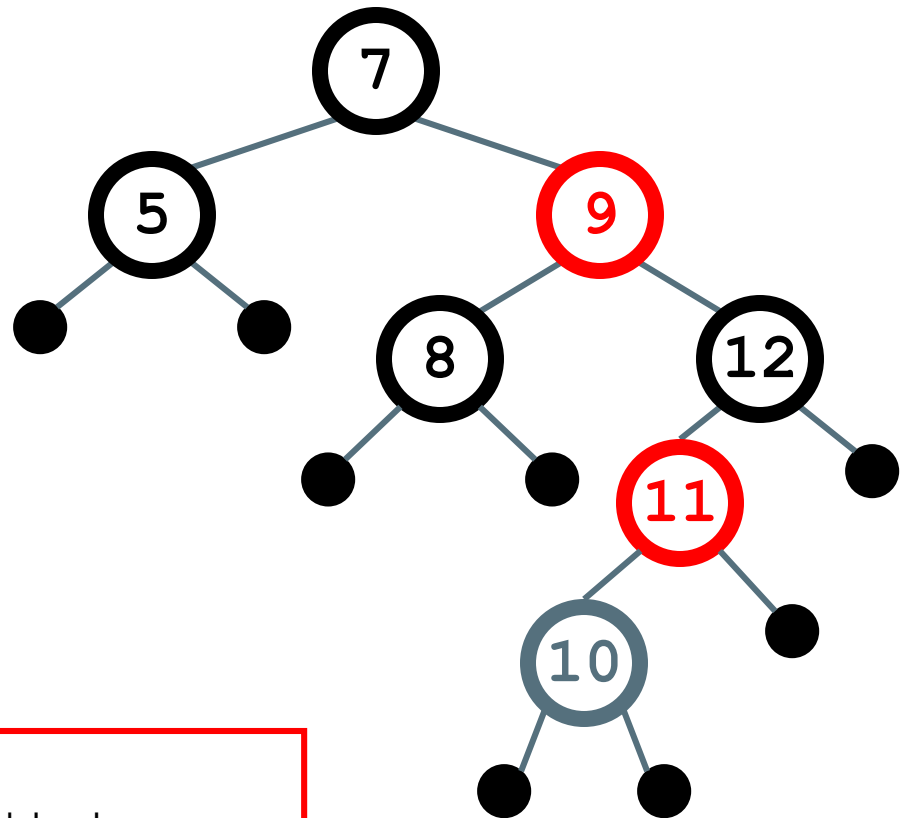
- Insert 10
 - Where does it go?



1. Every node is either red or black
2. Root and every leaf (NULL pointer) are black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes

The Problem With Insertion

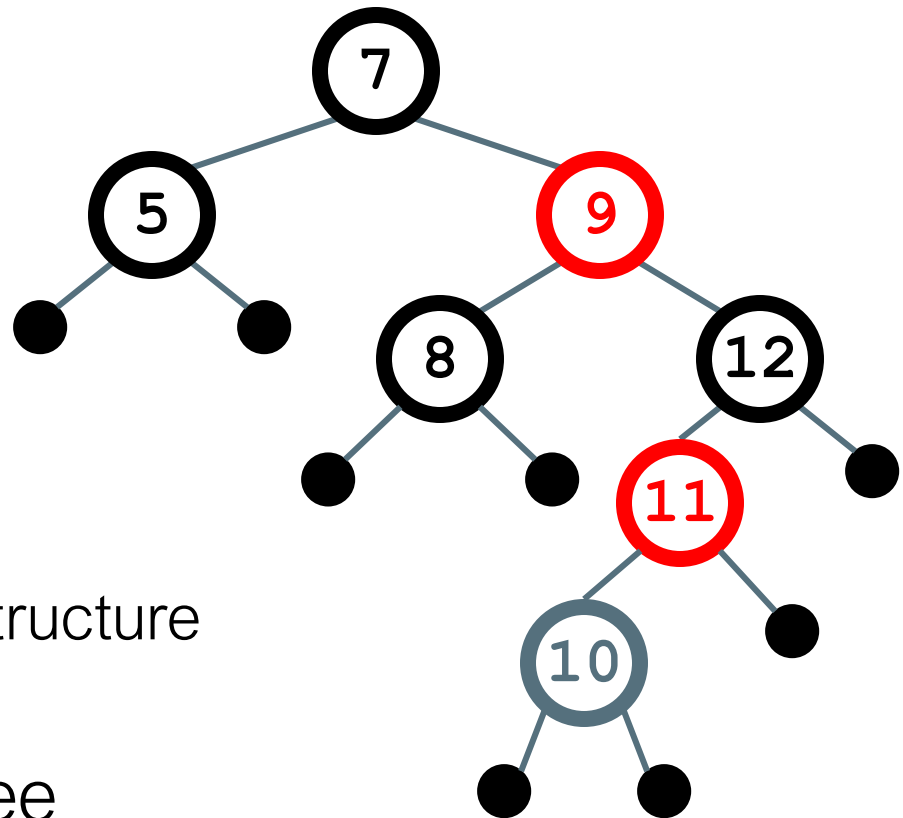
- Insert 10
 - Where does it go?
 - What color?



1. Every node is either red or black
2. Root and every leaf (NULL pointer) are black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes

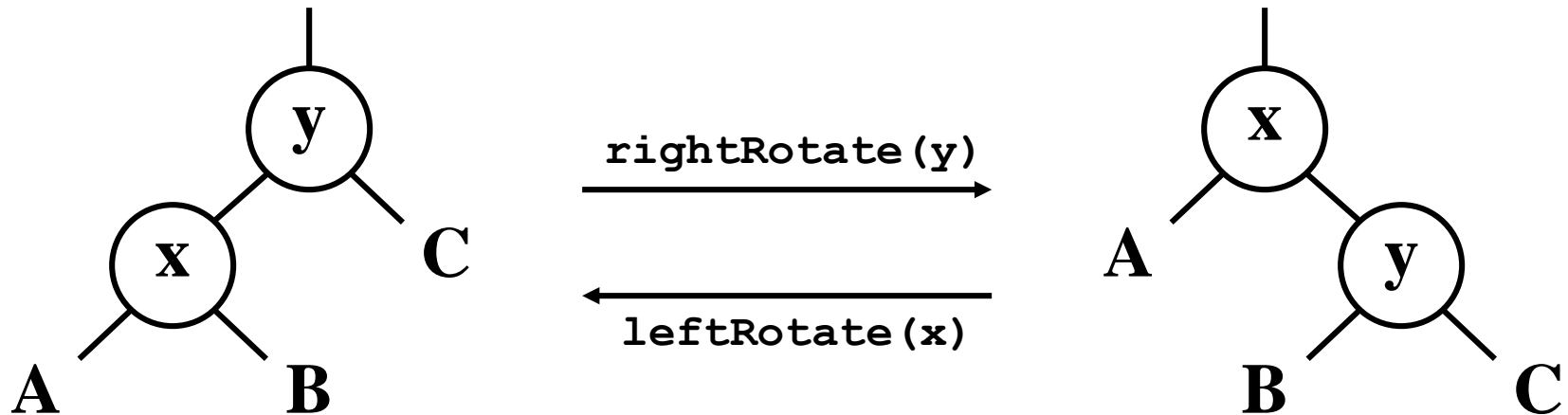
The Problem With Insertion

- Insert 10
 - Where does it go?
 - What color?
 - A: no color! Tree is too imbalanced
 - Must change tree structure to allow recoloring
 - Goal: restructure tree



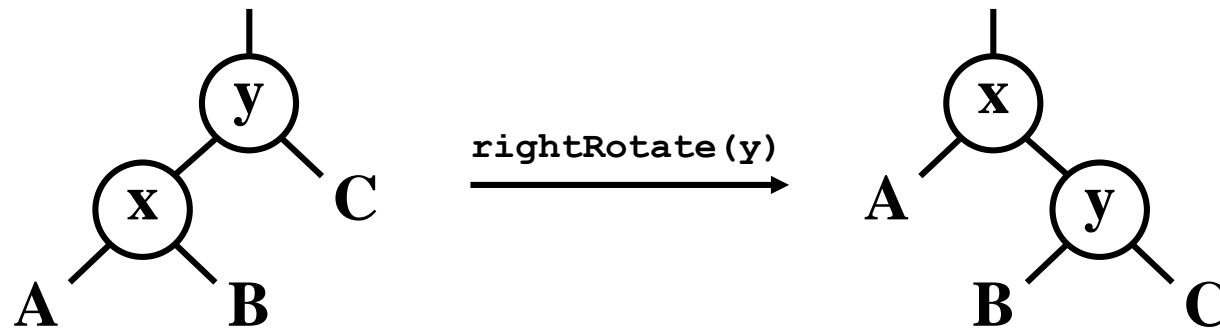
RB Trees: Rotation

- Our basic operation for changing tree structure is called **rotation**:



- Does rotation preserve inorder key ordering?
- What would the code for `rightRotate()` actually do?

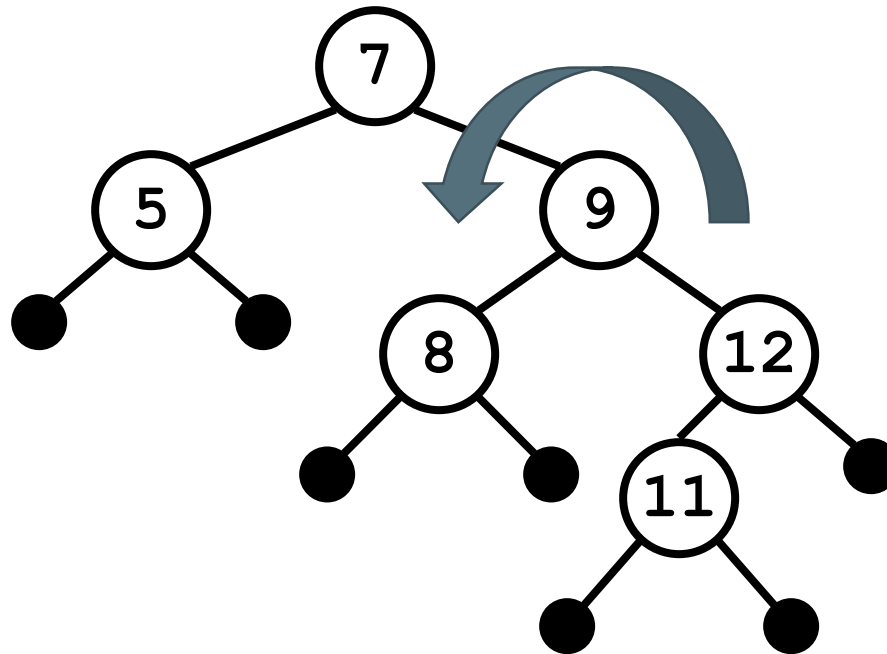
RB Trees: Rotation



- Answer: A lot of pointer manipulation
 - x keeps its left child
 - y keeps its right child
 - x's right child becomes y's left child
 - x's and y's parents change
- What is the running time?

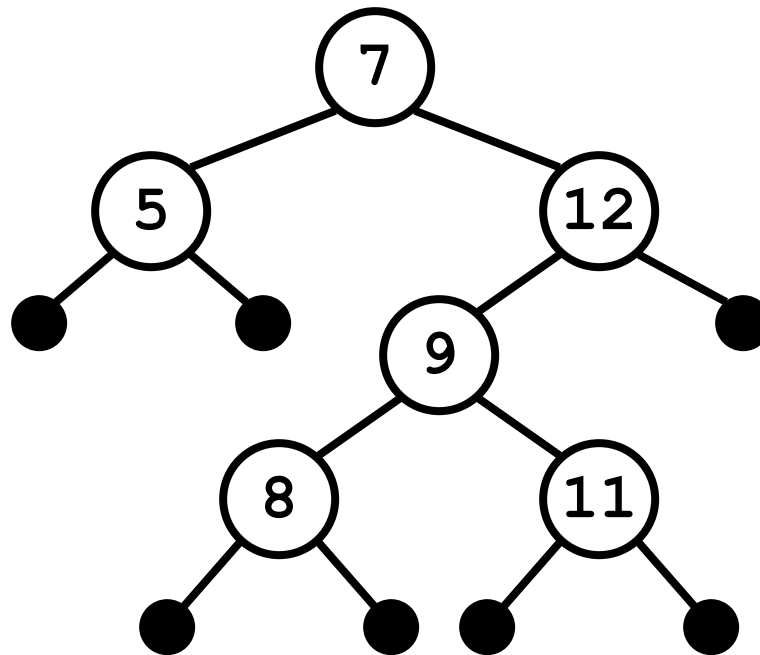
Rotation Example

- Rotate left about 9:



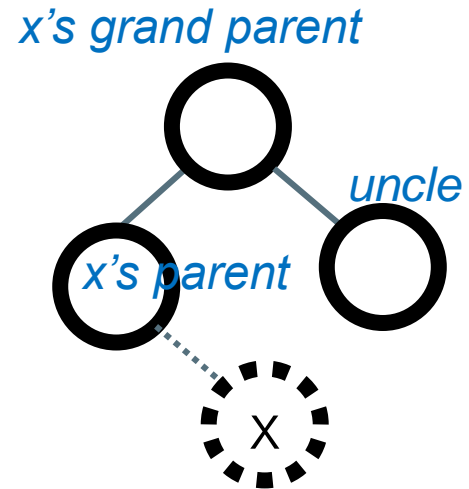
Rotation Example

- Rotate left about 9:



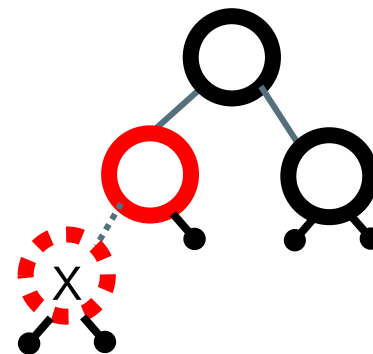
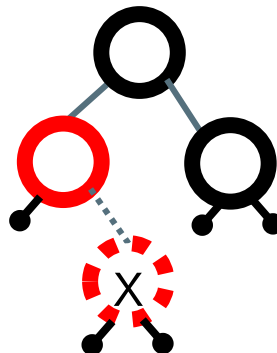
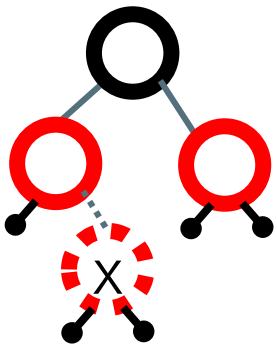
Three possible cases

- Suppose that the parent of the inserted node is left child of grand parent
- “Uncle” is the other child of grand parent



Cases:

- Inserted node's “uncle” is **red**
- Inserted node is **right** child and its “uncle” is **black**
- Inserted node is **left** child and its “uncle” is **black**

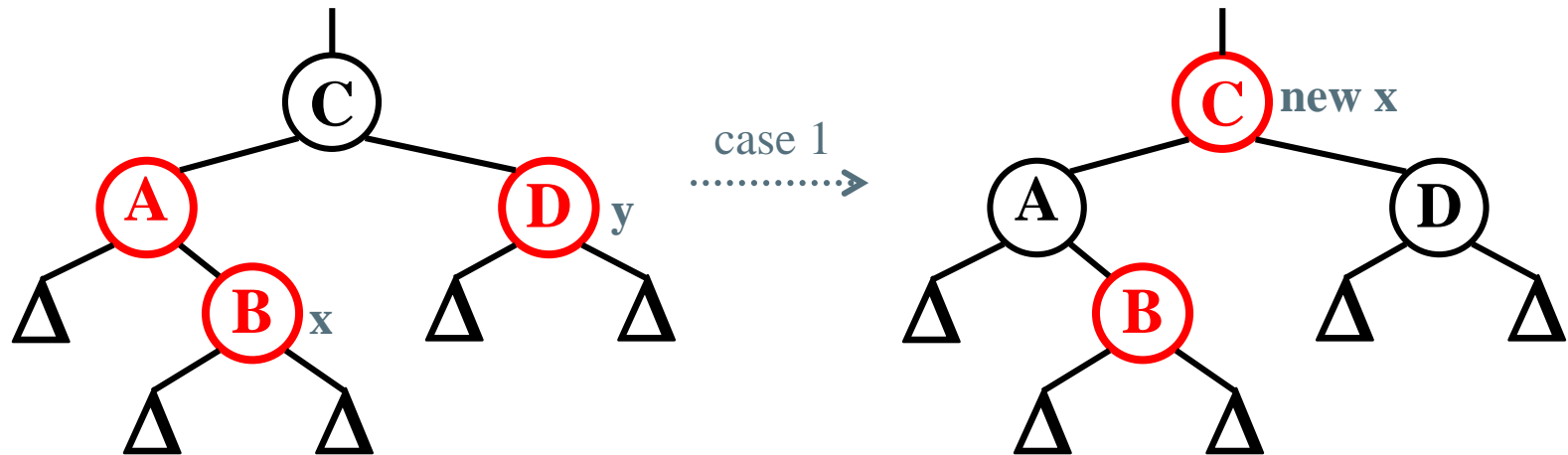


RB Insert: Case 1

```
if (y->color == RED)
    x->p->color = BLACK;
    y->color = BLACK;
    x->p->p->color = RED;
    x = x->p->p;
```

Case 1: “uncle” is **red**

In figures below, all Δ 's are equal-black-height subtrees



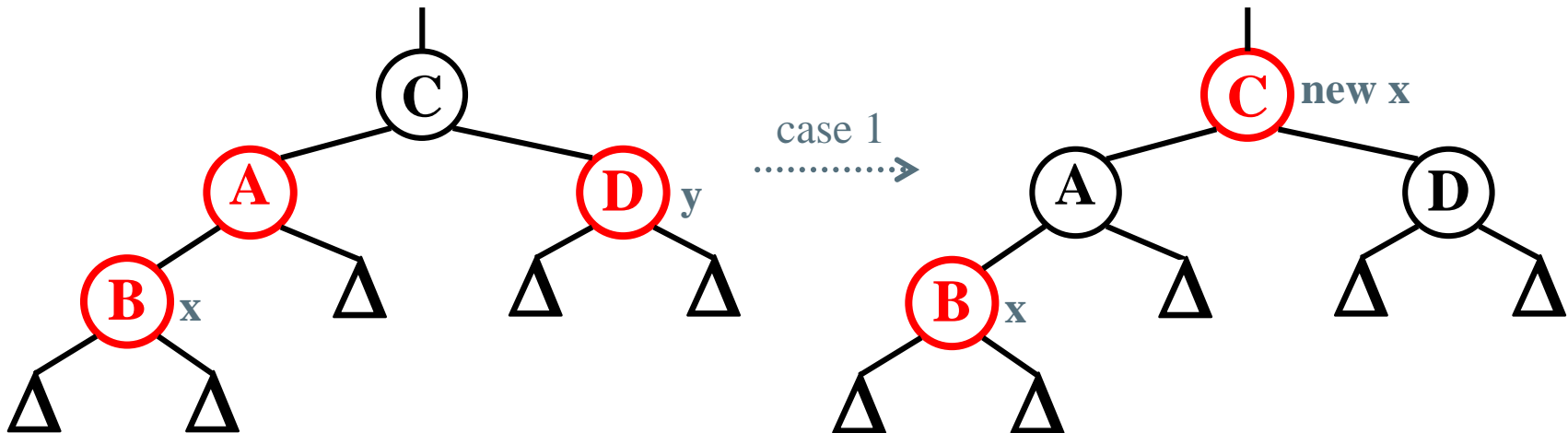
Note: if C is root, then reset the color to be black at the end of insertion

RB Insert: Case 1

```
if (y->color == RED)
    x->p->color = BLACK;
    y->color = BLACK;
    x->p->p->color = RED;
    x = x->p->p;
```

Case 1: “uncle” is **red**

In figures below, all Δ 's are equal-black-height subtrees



Same action whether x is a left or a right child

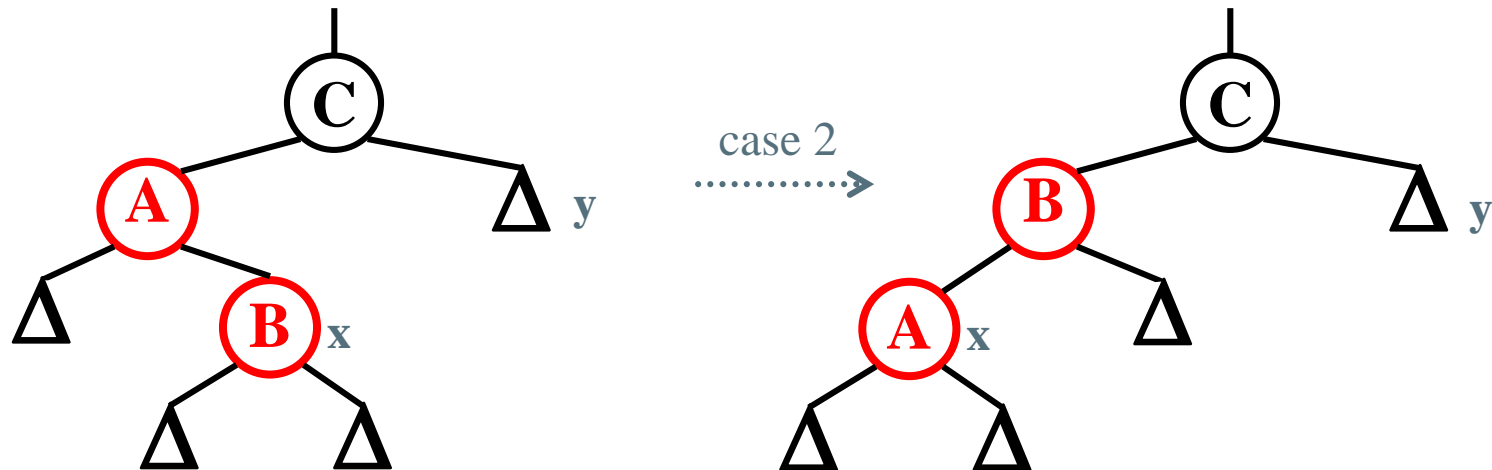
RB Insert: Case 2

```
if (x == x->p->right)
    x = x->p;
    leftRotate(x);
// continue with case 3 code
```

Case 2:

- “Uncle” is black
- Node x is a right child

Transform to case 3 via a left-rotation



Transform case 2 into case 3 (x is left child) with a left rotation
This preserves property 4: all downward paths contain same number of black nodes

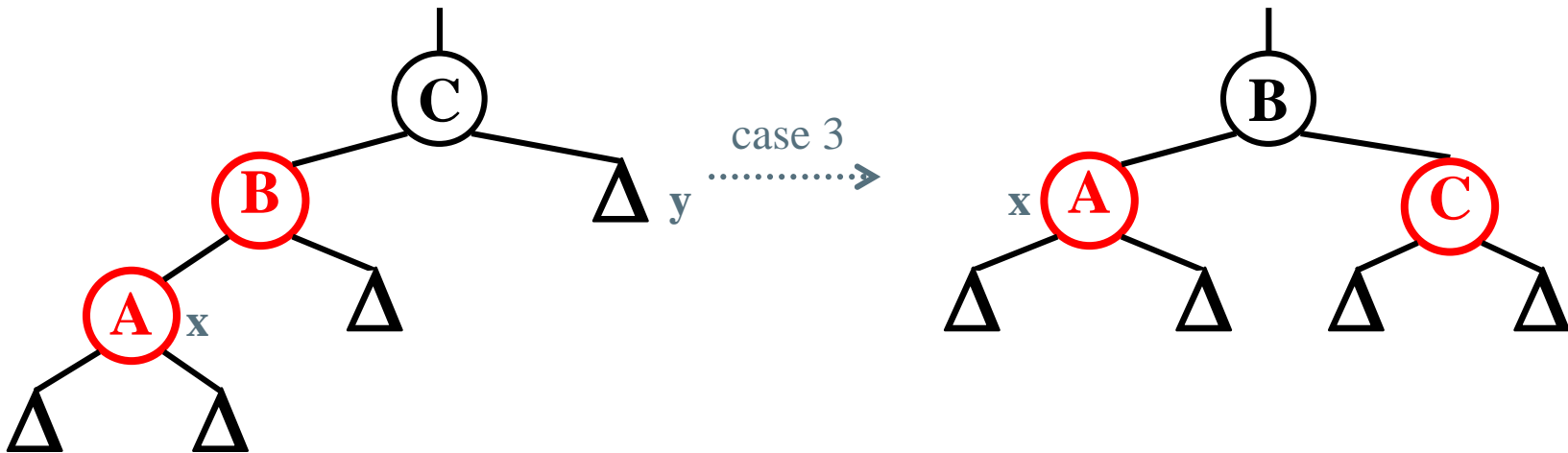
RB Insert: Case 3

```
x->p->color = BLACK;  
x->p->p->color = RED;  
rightRotate(x->p->p);
```

Case 3:

- “Uncle” is black
- Node x is a left child

Change colors; rotate right



Perform some color changes and do a right rotation

Again, preserves property 4: all downward paths contain same number of black nodes

RB Insert: Cases 4-6

- Cases 1-3 hold if x 's parent is a left child
- If x 's parent is a right child, cases 4-6 are symmetric (swap left for right)

Red-Black Trees: Deletion

- And you thought insertion was tricky...
- We will not cover RB delete in class
- Read Chapter 14 of Introduction to Algorithms (by Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest)

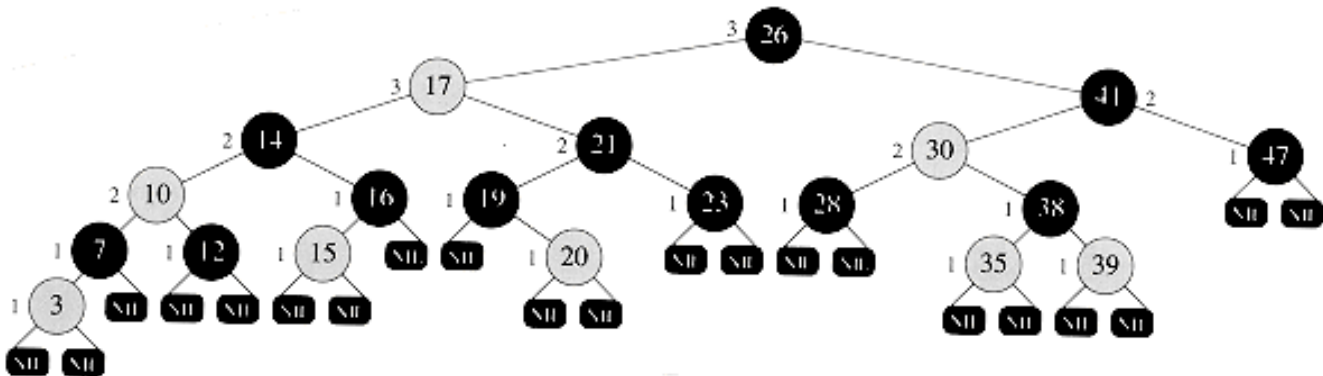
Black-height

> The black height ($bh(x)$) of a node is defined as the number of black nodes on any path from, but not including, a node x down to a leaf (null leaf is counted).

What is the minimum black-height of a node with height h ?

A: a height- h node has black-height $\geq h/2$

Theorem: A red-black tree with n internal nodes has height $h \leq 2 \lg(n + 1)$ > Proved by (what else?) induction



RB Trees: Worst-Case Time

- A **red-black** tree has $O(\lg n)$ height
- Corollary: These operations take $O(\lg n)$ time:
 - Minimum(), Maximum()
 - Successor(), Predecessor()
 - Search()
- Insert() and Delete():
 - will also take $O(\lg n)$ time
 - but will need special care since they modify tree

Visualizations

<https://www.cs.usfca.edu/~galles/visualization/RedBlack.html>

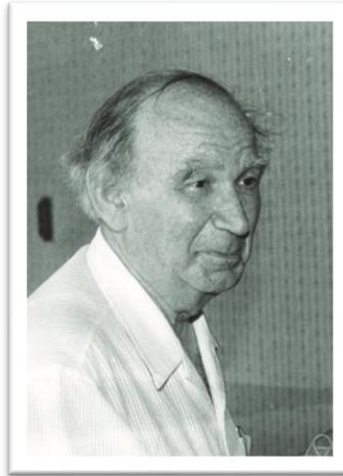
AVL Tree

Close to Balanced Tree

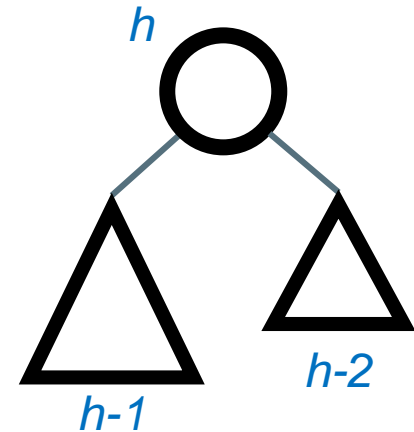
Adelson-Velskii & Landis



1922-2014



1921-1997



AVL is a Binary Search Tree built in such a way that in all its nodes, **the height of its right subtree differs from the height of the left subtree by at most 1.**

AVL is known as the first self-balanced binary search tree.

Balance Factor

The balance factor of a BST is given by the following formula:

$$\mathbf{bf(n) = h(n_{left}) - h(n_{right})}$$

The balance factor for any node in AVL is:

$$\mathbf{bf(n) \in \{1, 0, -1\}}$$

What happens if the balance factor of a node in an AVL Tree is $\mathbf{bf(n) \notin \{1, 0, -1\}}$?

The AVL property is violated!

Balance Factor

How the AVL property is violated?

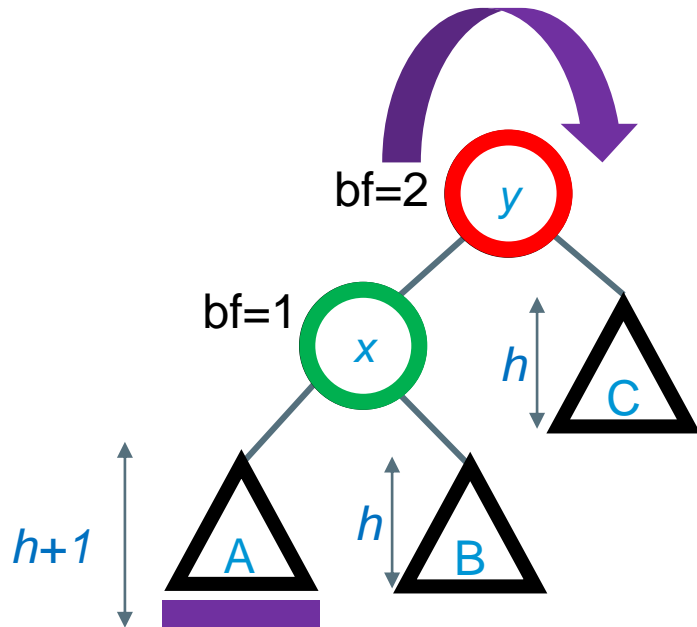
Inserting or deleting a node in an AVL tree can increase/decrease the height of a subtree by 1. That is, $bf(n)$ can become +2 or -2.

The three must be rebalanced. How?

Single or Double rotations must be applied to the node n where the property was violated. The rotations will restore the AVL property.

Single Right Rotation

New node is inserted
in the subtree A

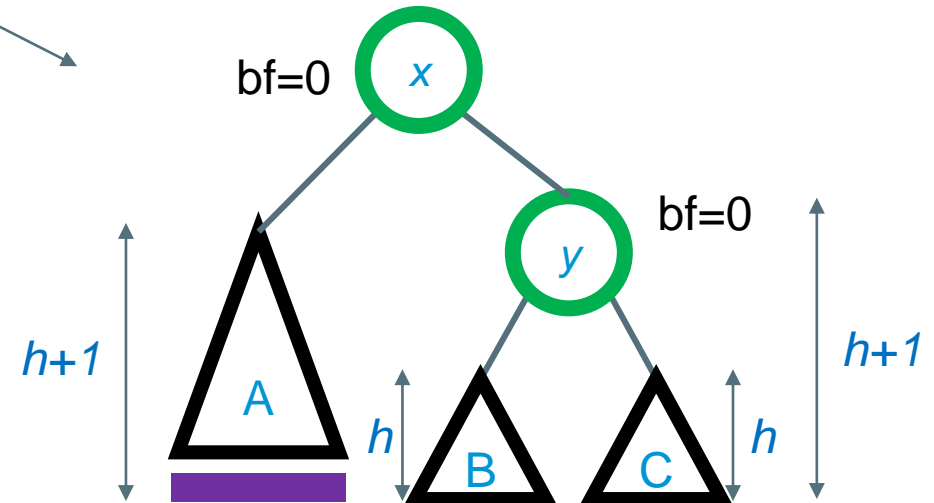


The AVL Tree is balanced again!

What are the balance factors for x and y?

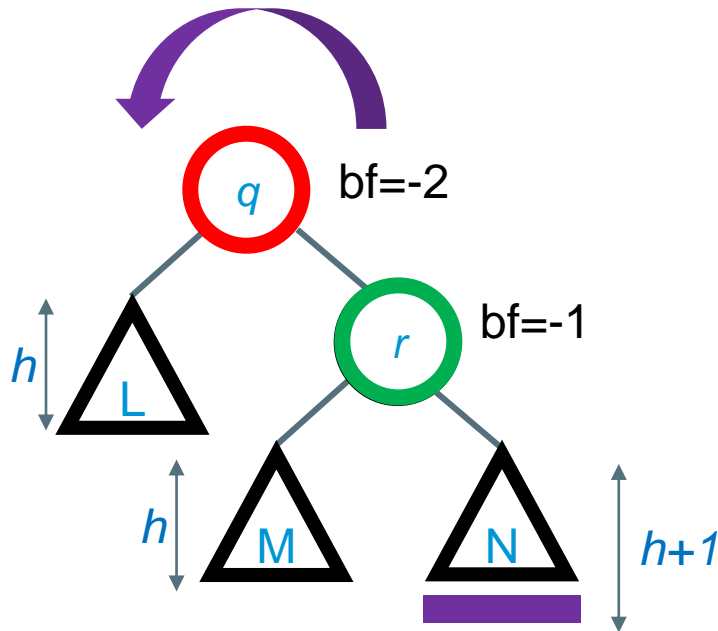
AVL property is violated at y!!!

How can we fix the AVL tree?



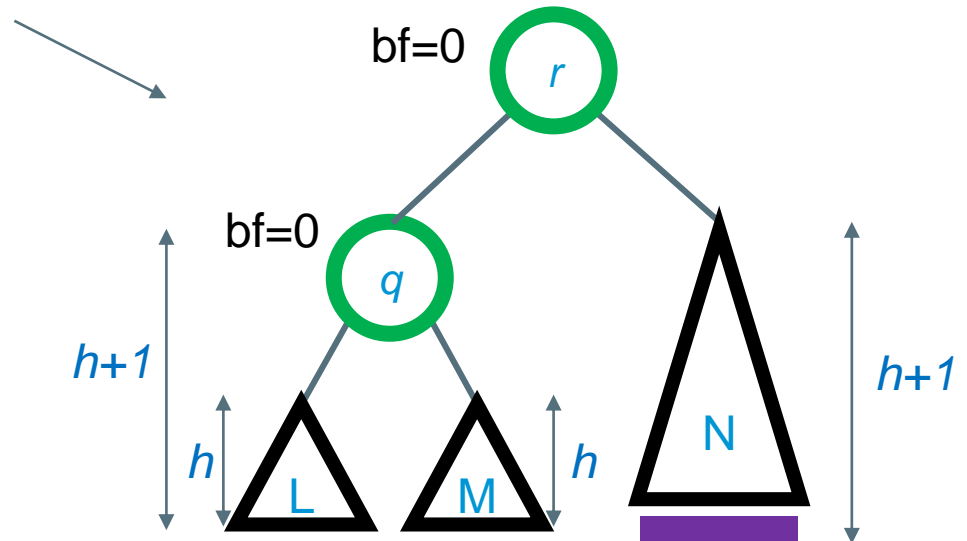
Single Left Rotation

New node is inserted in the subtree N



The AVL Tree is balanced again!

Right and Left rotations are symmetric
Single rotation takes $O(1)$ time
(only pointer manipulation)



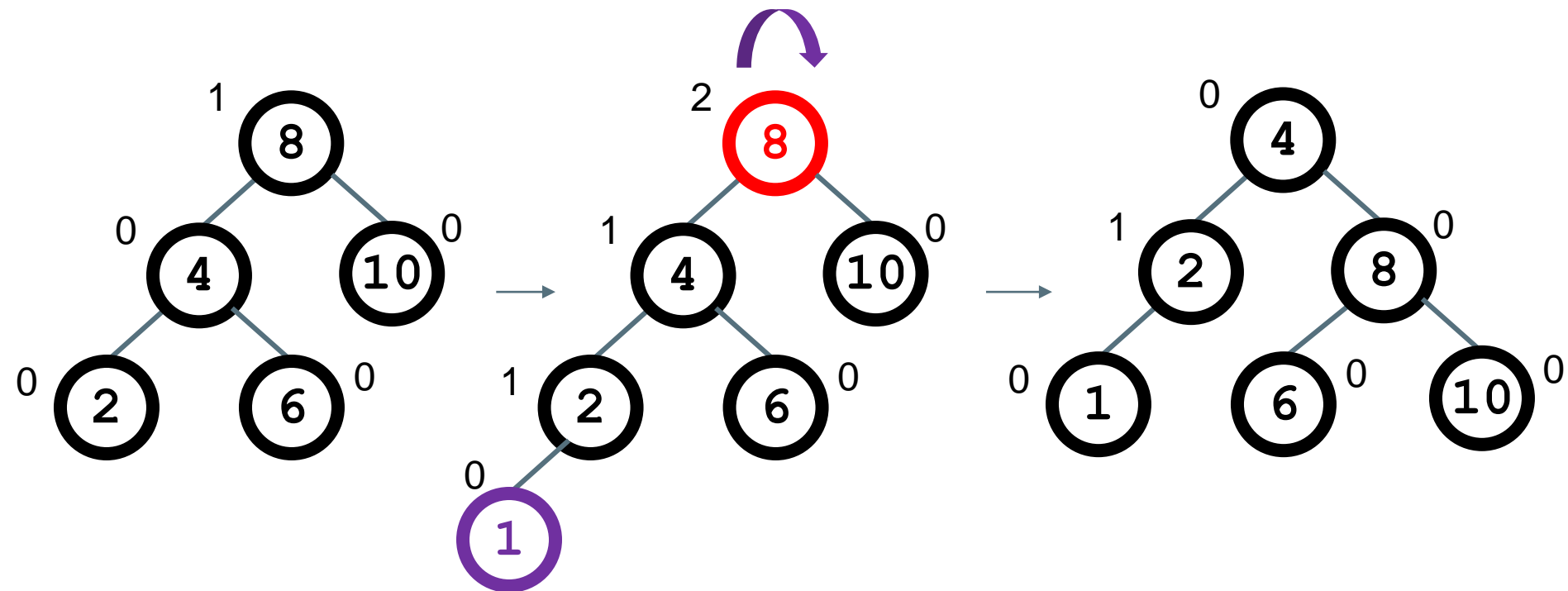
Example 1 – Right Rotation



* New inserted node

Left rotation is symmetric

Example 2 – Right Rotation



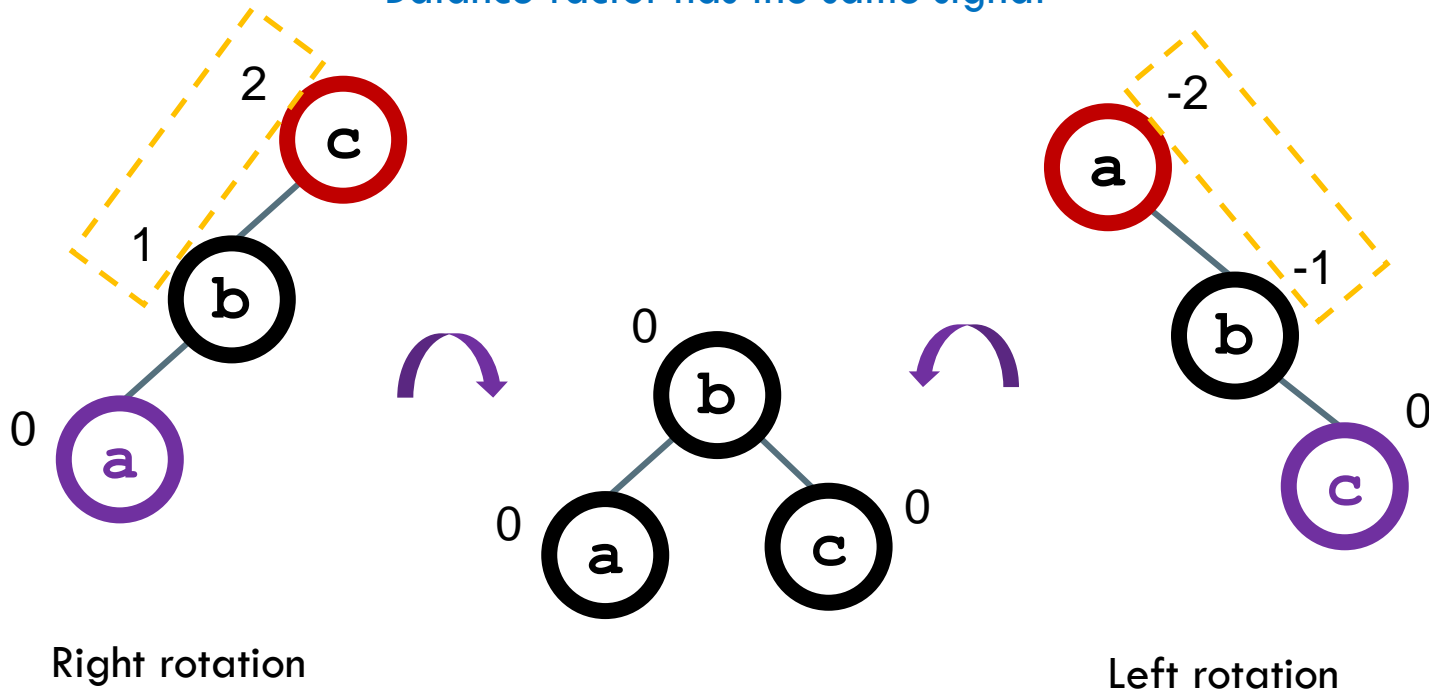
Insert node: 1

Left rotation is symmetric

Example 3 –Right vs Left Rotation

Do you see any pattern?

Balance factor has the same signal

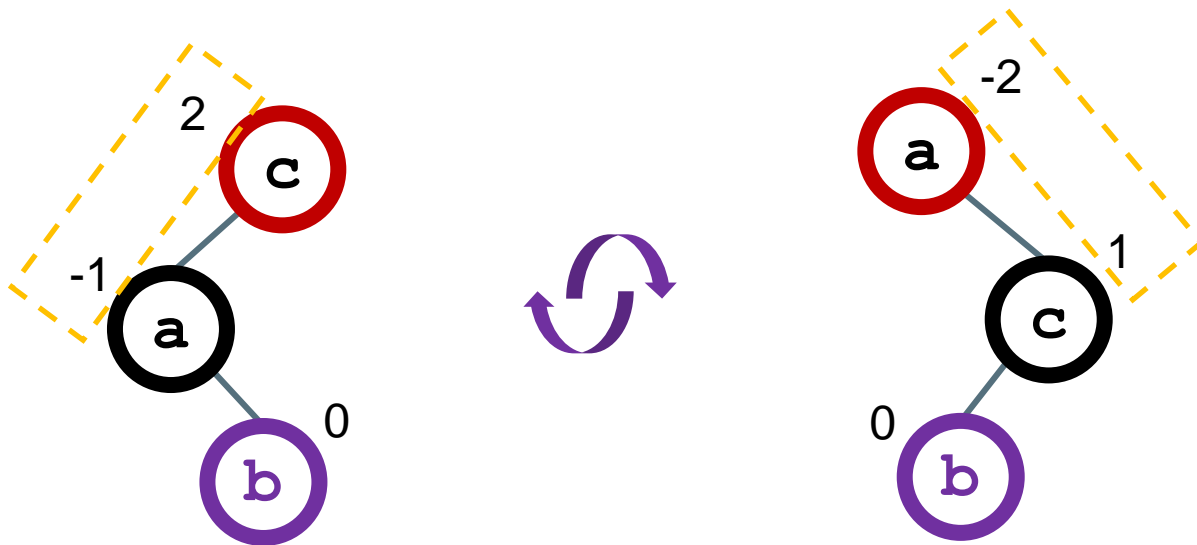


* New inserted node

Example 4 – What about these cases?

Right Rotation or Left Rotation?

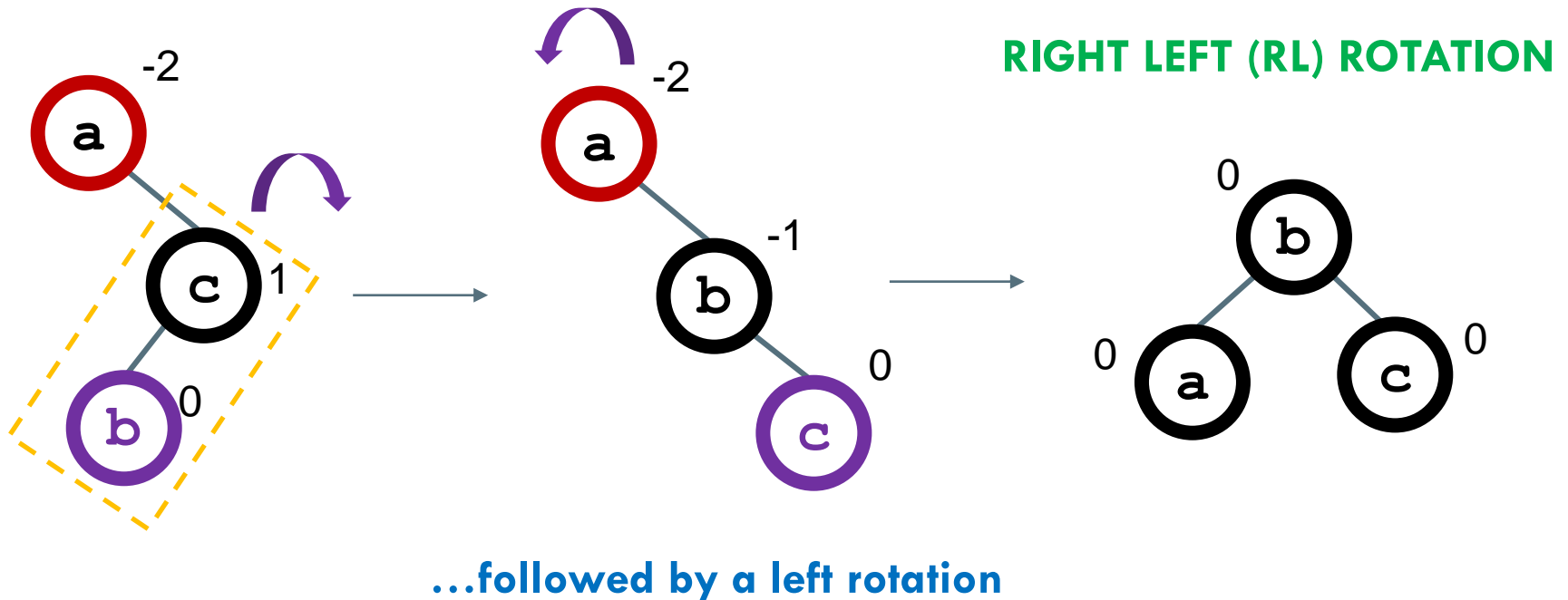
It does not work. We need to double rotate.



* New inserted node

Double Rotation

A right rotation in the right sub-tree...

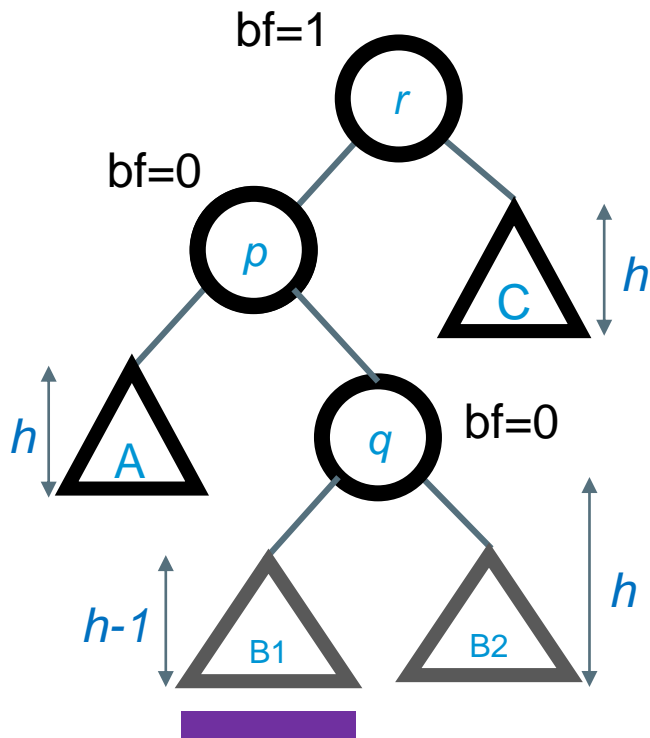


* New inserted node

* Left-Right rotation is symmetric

Left-Right Rotation

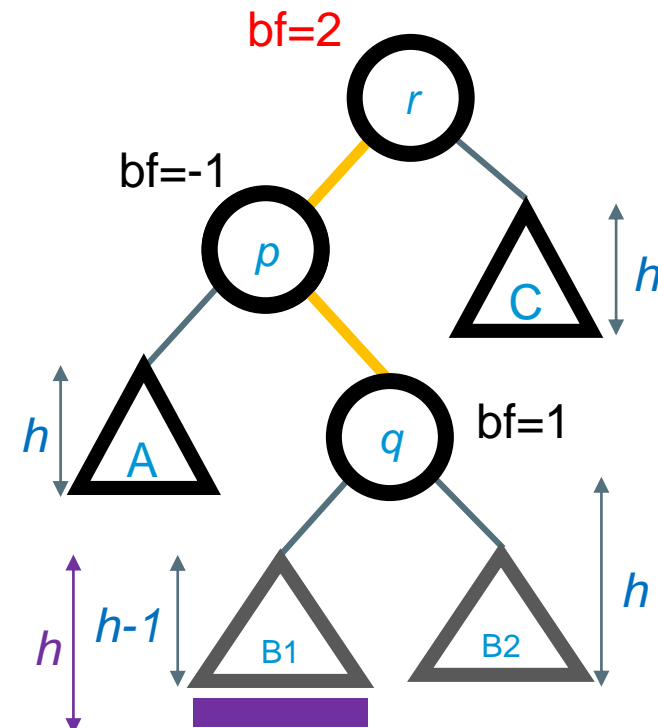
New node is inserted
in the subtree B1



New node is inserted in the sub-tree **B1**
(It could be B1 or B2)

AVL property is violated at r !!!

Zig-Zag shape (different signals)

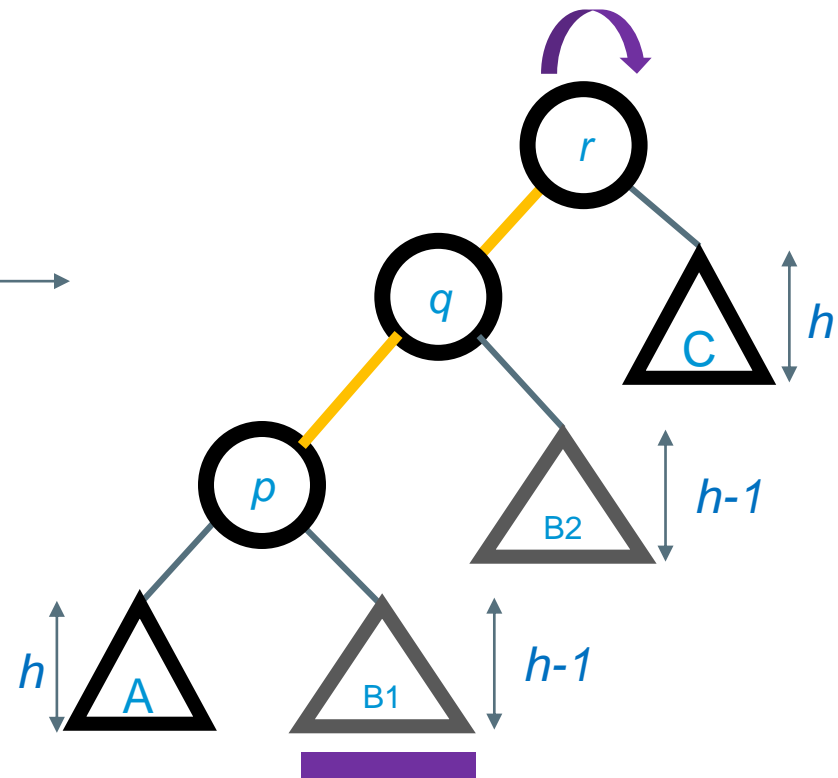
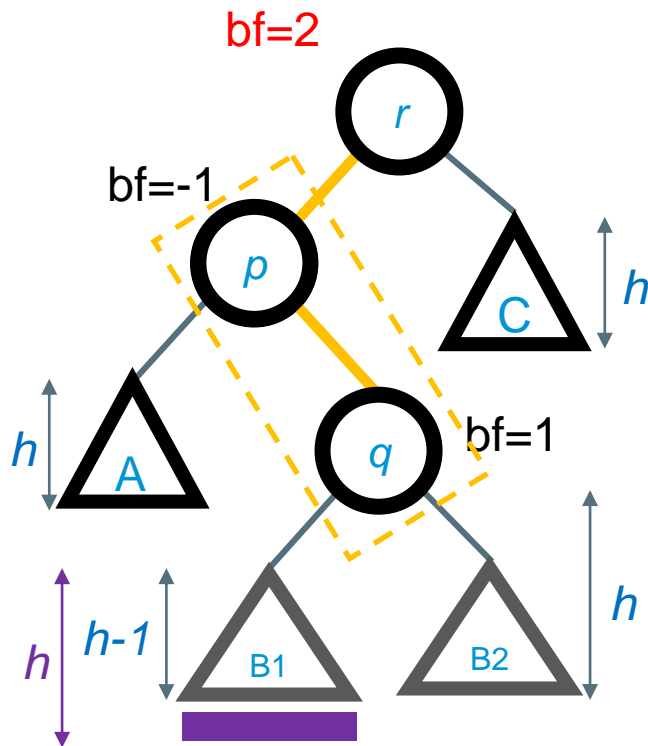


Left-Right Rotation

New node is inserted
in the subtree B1

A left rotation in the left sub-tree...

Now, we need a right rotation...



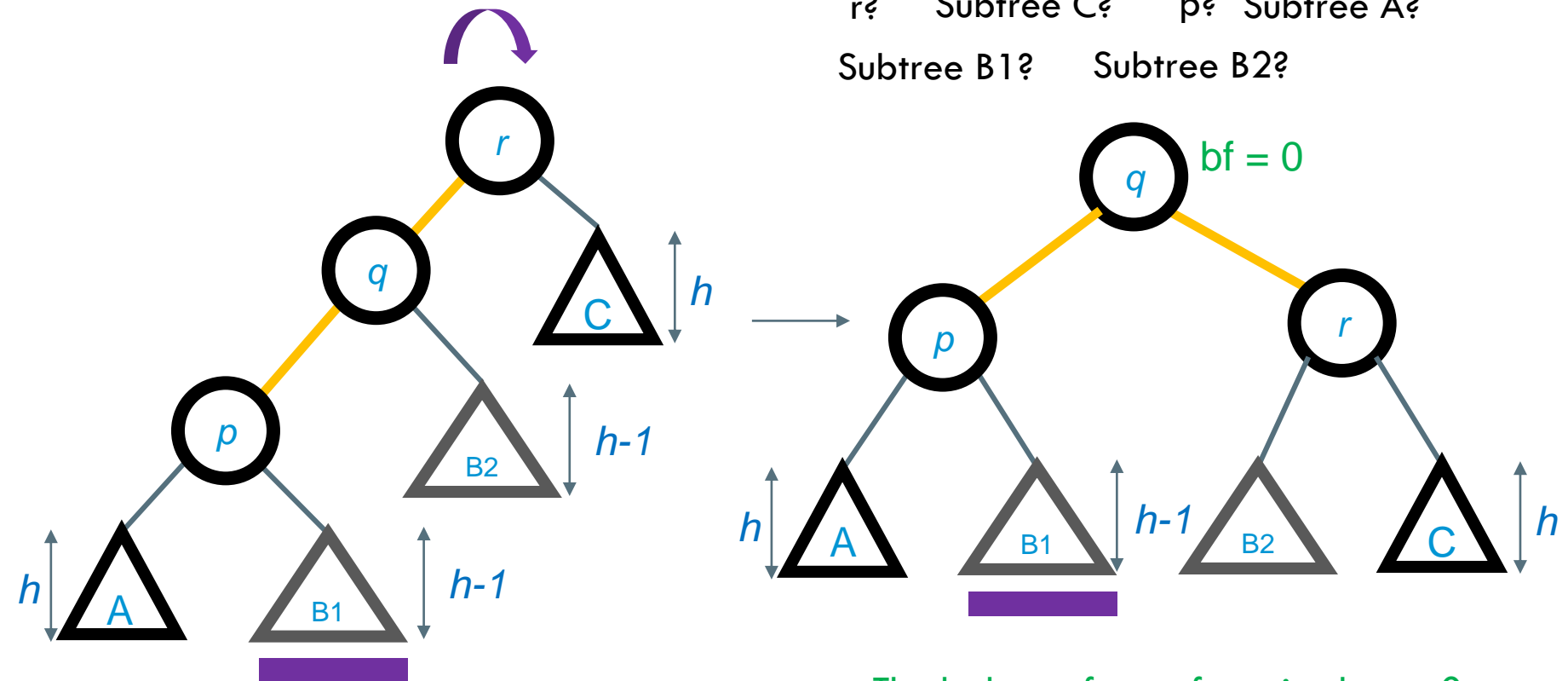
Left-Right Rotation

New node is inserted
in the subtree B1

q is the new parent/root

r? Subtree C? p? Subtree A?

Subtree B1? Subtree B2?

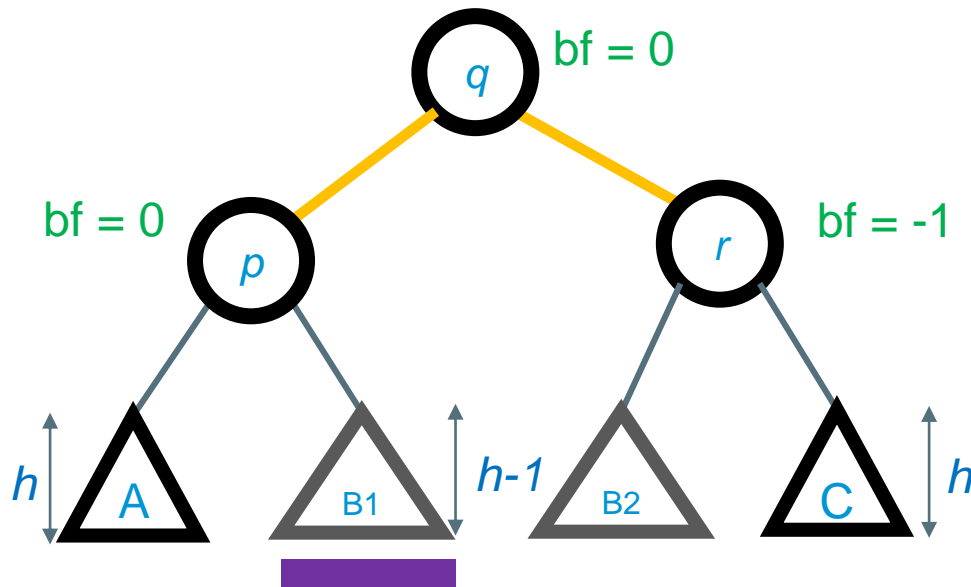


The balance factor for q is always 0

What about the balance factor for p and r ?

Left-Right Rotation

```
p->bf = (q->bf == -1) ? 1 : 0;
r->bf = (q->bf == 1) ? -1 : 0;
q->bf = 0; //always 0
```



Left Right and Right Left rotations are symmetric

AVL Insertion

The Insertion algorithm for AVL is similar to the BST.

- > Traverse the tree (left and right) until a null child is reached
- > Insert the **new node** (the new node is inserted as a leaf)

[Generic statement] After the **new node** is inserted, check if any of the subtrees requires to be rebalanced (i.e., the **new node** inserted may violate the balance factor). Note that once a subtree has been rebalanced, the rest of the tree is guaranteed to be balanced as well.

- > (Recursive approach) Starting from the inserted **node**, we can traverse upwards to the root, checking/updating the balance factor of each node along the way and rebalancing it if necessary (if $bf(n) > 1$ or $bf(n) < -1$ then rebalance n).

We need to consider **4 cases** for rebalancing:

Single Rotations

Case 1: Insertion into left subtree of left child of the node violated > **rightRotate(n)**

Case 2: Insertion into right subtree of right child of the node violated > **leftRotate(n)**

Double Rotations

Case 3: Insertion into right subtree of left child of the node violated > **leftRotate(n.left)** and then **rightRotate(n)**

Case 4: Insertion into left subtree of right child of the node violated > **rightRotate(n.right)** and then **leftRotate(n)**

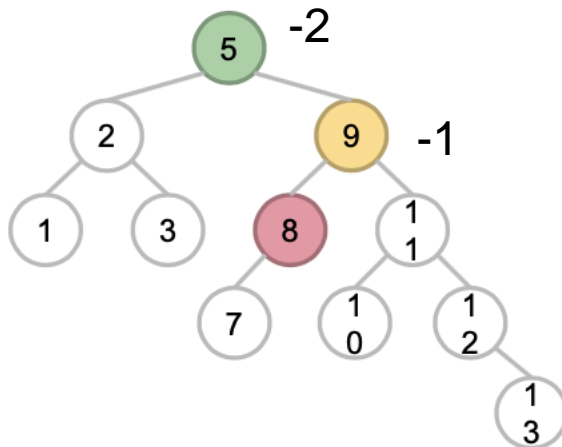
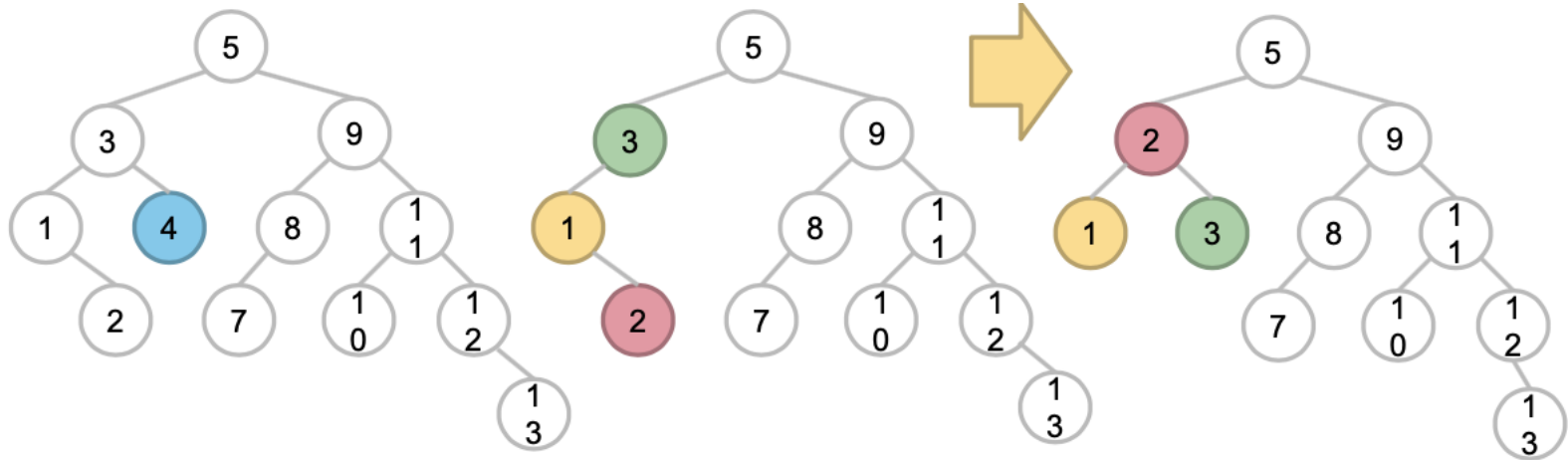
AVL Deletion

The Deletion algorithm for AVL is similar to the BST.

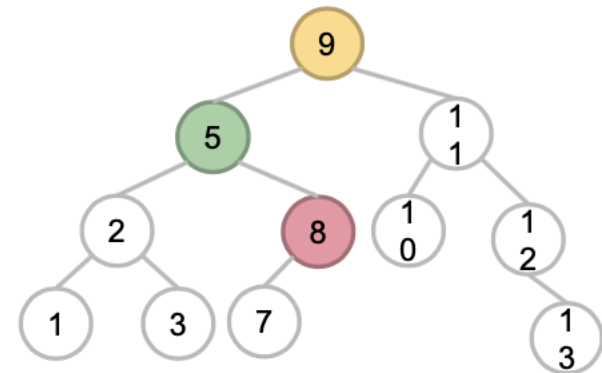
- > Recursively traverse the tree (left and right) until the **searched node** is found
- > Delete the **searched node**
 - >> Case 1: removed node is a leaf
 - >> Case 2: removed node has only one child
 - >> Case 3: removed node has two children (subtrees)
- > After the **searched node** is deleted, check/update the balance factor of the nodes all the way up to the root making the required changes/rotations when the balance factor of a node violates the AVL property.
- >> Note that after a deletion a single or double rotation may not rebalance the tree. Rotations may cascade back up the path to the root of the tree. We may need $O(\log n)$ rotations to rebalance the tree.

Example Deletion (4) – multiple rotations required

Double rotation LR

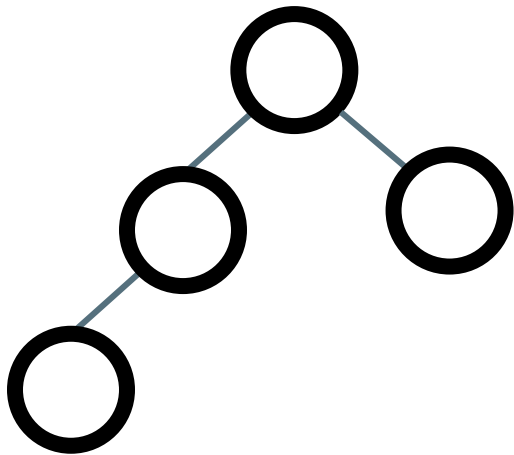


Left Rotation

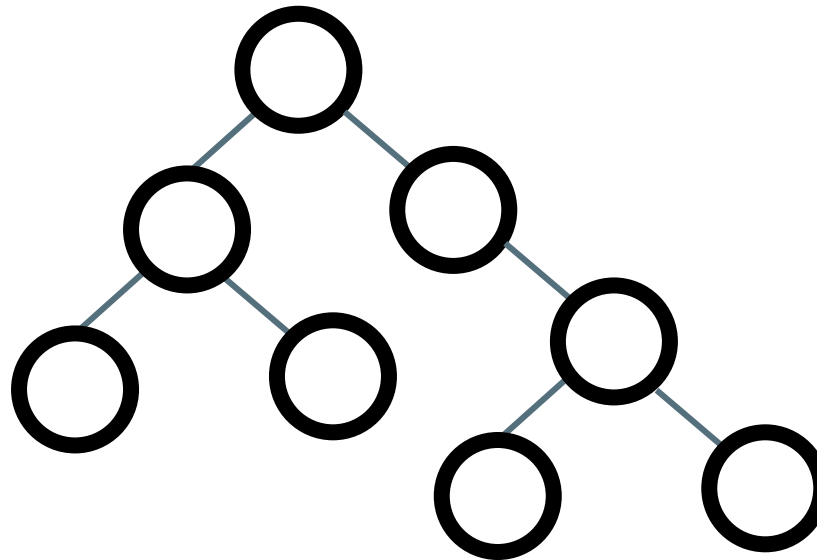


Which tree is not AVL? Why?

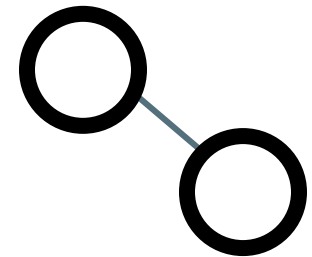
AVL Tree



¬AVL Tree



AVL Tree



AVL Height

Let $N(h)$ be the minimum number of nodes of an AVL of height h .

So, the height of one of the subtrees is at least $h-1$ and the height of the other is at least $h-2$.

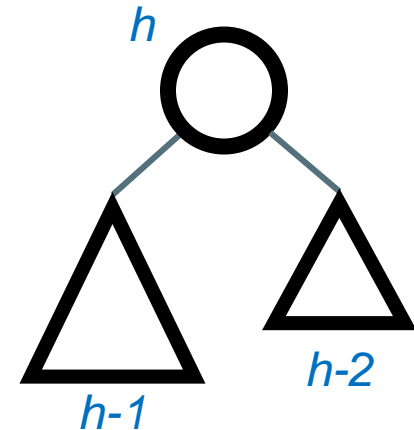
$$N(h) = N(h-1) + N(h-2) + 1$$

$$[N(h) + 1] = [N(h-1) + 1] + [N(h-2) + 1]$$

$$[N(h) + 1] = \text{Fibonacci numbers}$$

$$N(h) + 1 \approx \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} \right)^{h+3}$$

$$h \approx 1.44 \log(n)$$



The height of an AVL tree is bounded by roughly $1.44 * \log_2(n)$, while the height of a red-black tree may be up to $2 * \log_2(n)$. Thus, lookup is slightly faster on the average in AVL.

Visualizations

<https://www.cs.usfca.edu/~galles/visualization/AVLtree.html>

Exercise

Which one of the following properties is INCORRECT in a red-black tree?

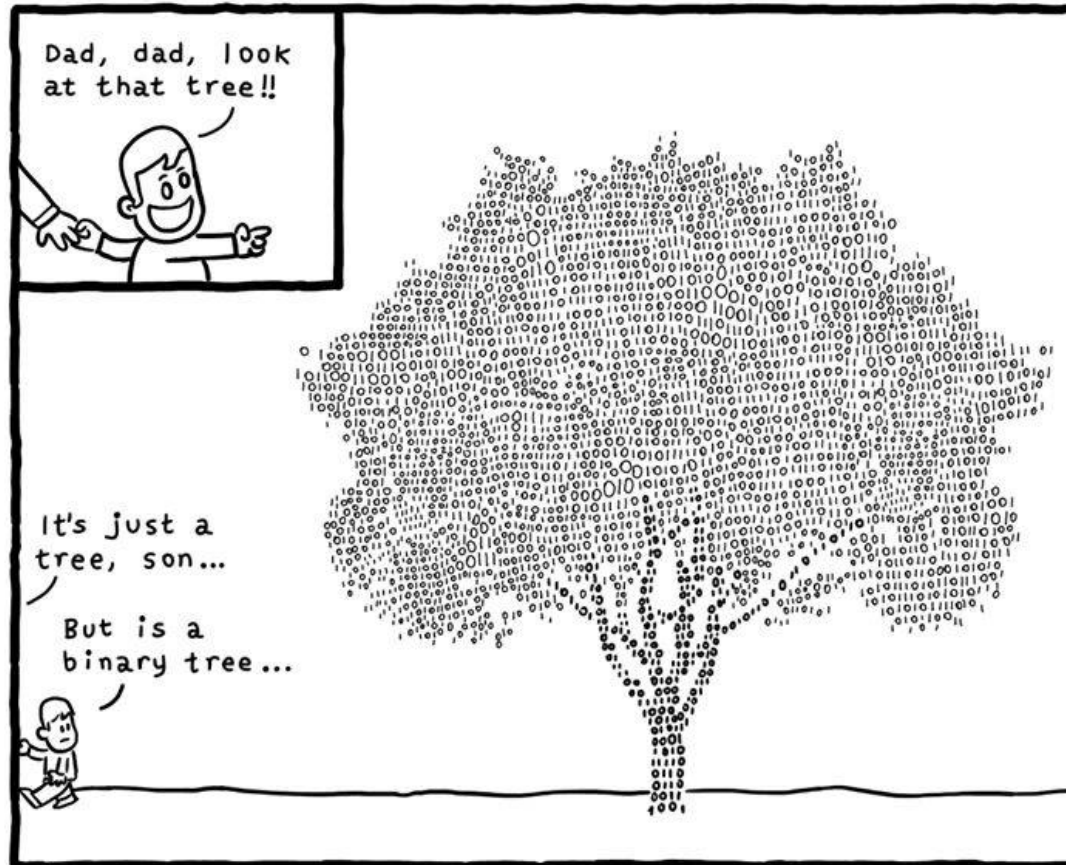
1. Every node must be either red or black.
2. If a parent is red, then its children can be either red or black.
3. All simple paths from any node x to a descendant leaf have the same number of black nodes.
4. The root and leaves should be black.

Exercise

Fill in the blanks in the following statement:

- 1) A binary search tree is _____ .
 - 2) A Red-Black and AVL trees are _____ trees.
 - 3) A Red-Black tree is a _____ with additional properties.
-
1. balanced, unbalanced, binary search tree
 2. unbalanced, balanced, binary search tree
 3. unbalanced, close to balanced, binary search tree
 4. unbalanced, unbalanced, binary tree

Meme for today's lecture! Keep practicing!



Daniel Stori {turnoff.us}

Should it be upside down?

References

- Chapter 14 of Introduction to Algorithms (by Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest)
- Chapter 10 of Lee K.D., Hubbard S. (2015) Balanced Binary Search Trees. In: Data Structures and Algorithms with Python. Undergraduate Topics in Computer Science. Springer, Cham.