

CSC 212: Data Structures and Abstractions

Binary Search Trees

Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Fall 2020



Quick notes

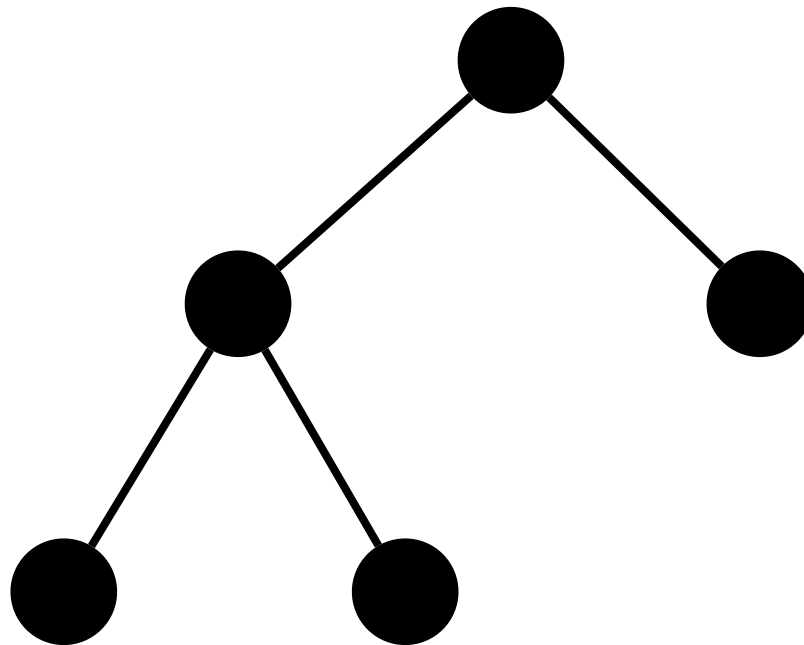
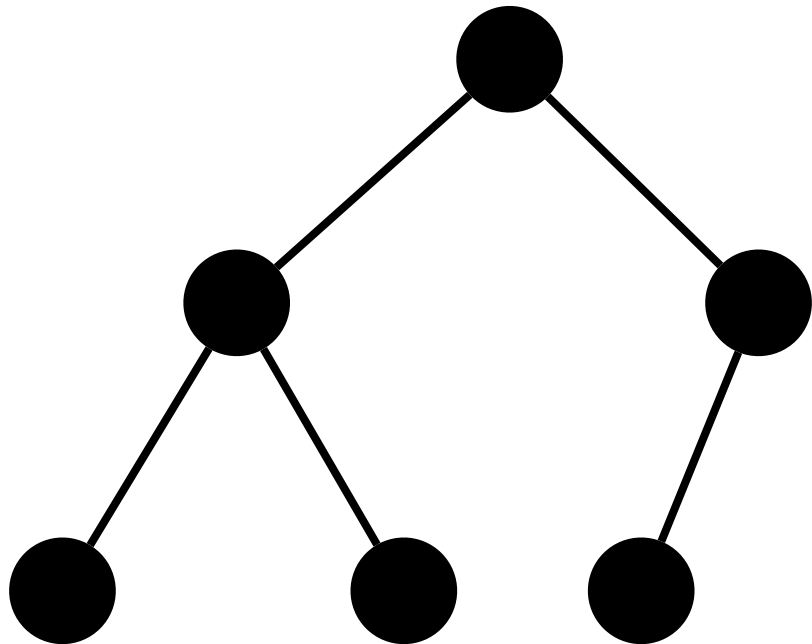
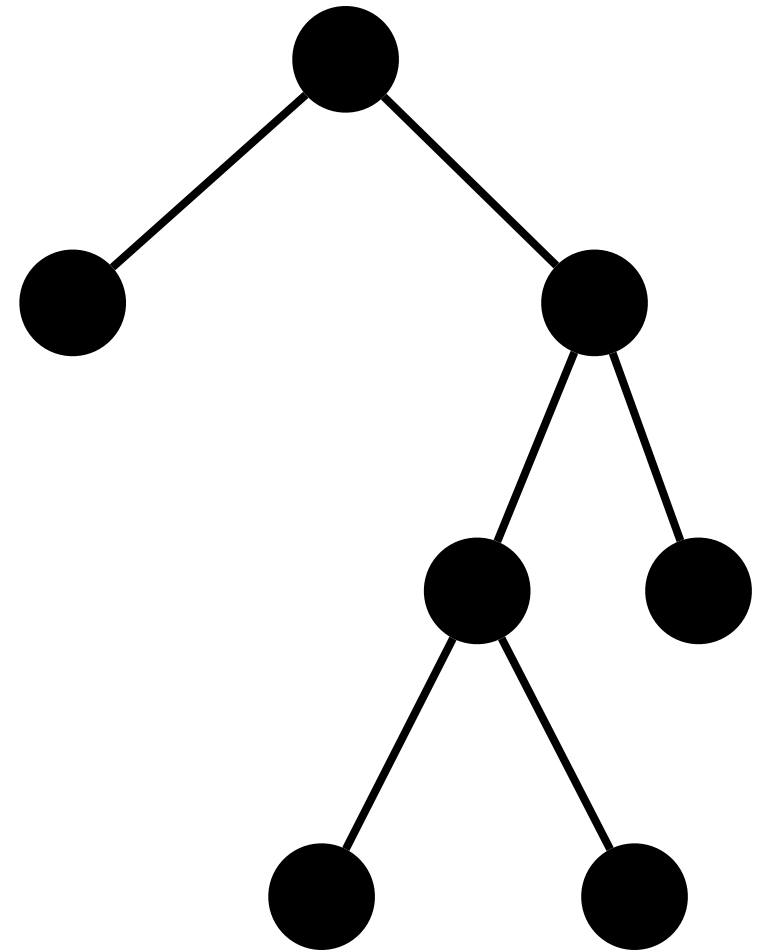
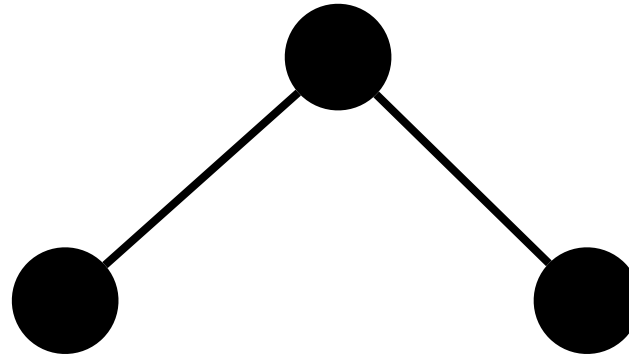
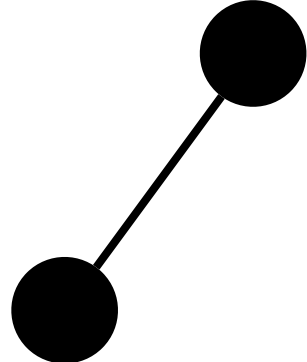
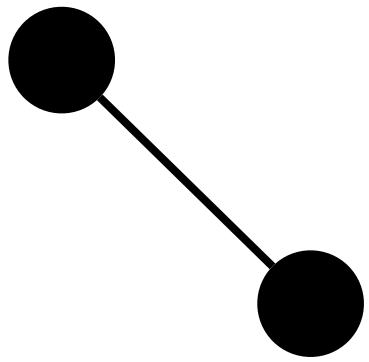
- Final Project (about 5 weeks)
 - ✓ requires planning and long coding hours
 - ✓ there is a lot to learn
- Team Work
 - ✓ motivate each other
 - ✓ all team members must understand the topic and code
 - a presentation to the class will follow by the end of the semester

k-ary Trees

k-ary Trees

- In a **k-ary tree**, every node has between 0 and k children
- In a **full (proper)** k-ary tree, every node has exactly 0 or k children
- In a **complete** k-ary tree, every level is entirely filled, except possibly the deepest, where all nodes are as far left as possible
- In a **perfect** k-ary tree, every leaf has the same depth and the tree is full

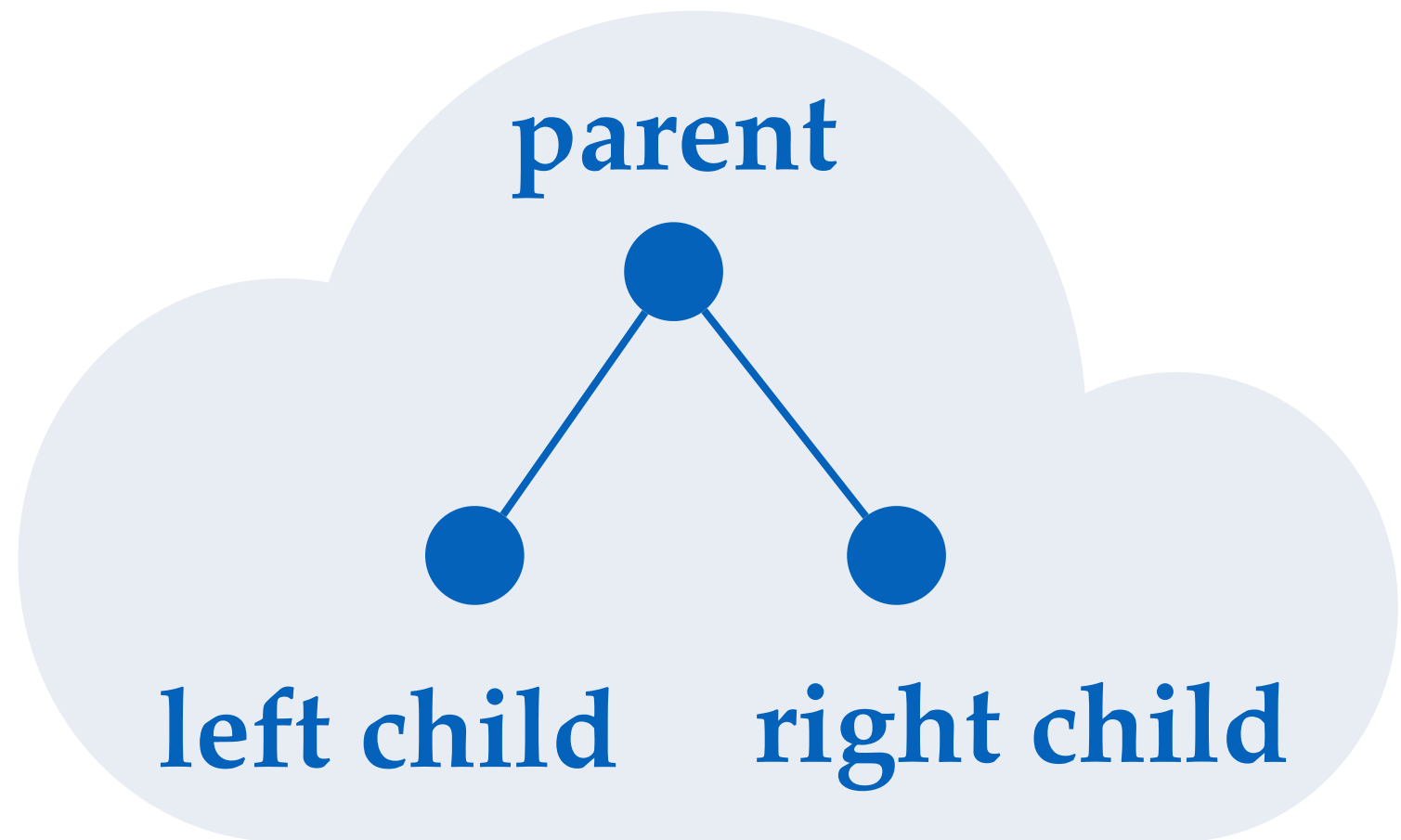
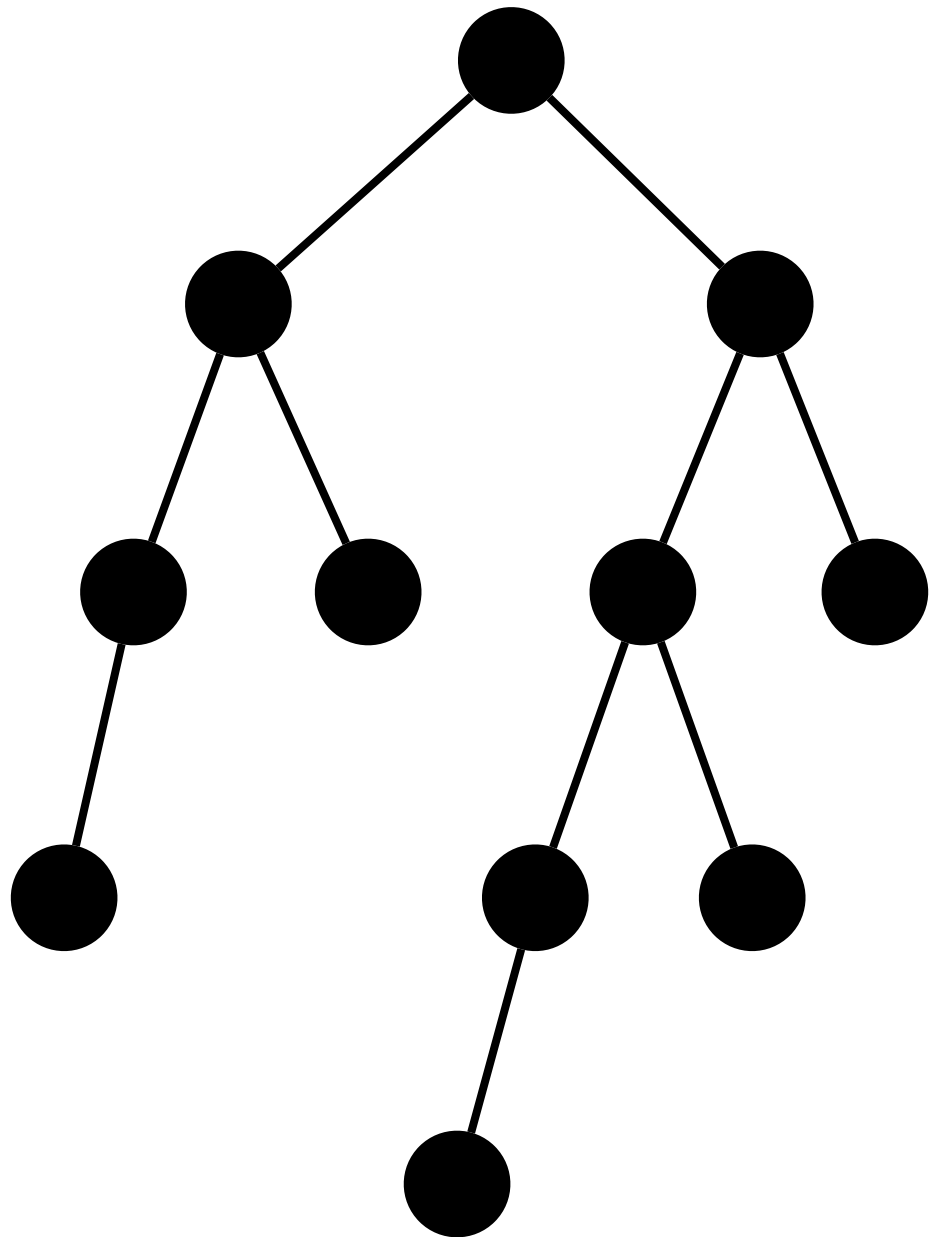
Quiz ($k = 2$)



Full? Complete? Perfect?

Binary Tree

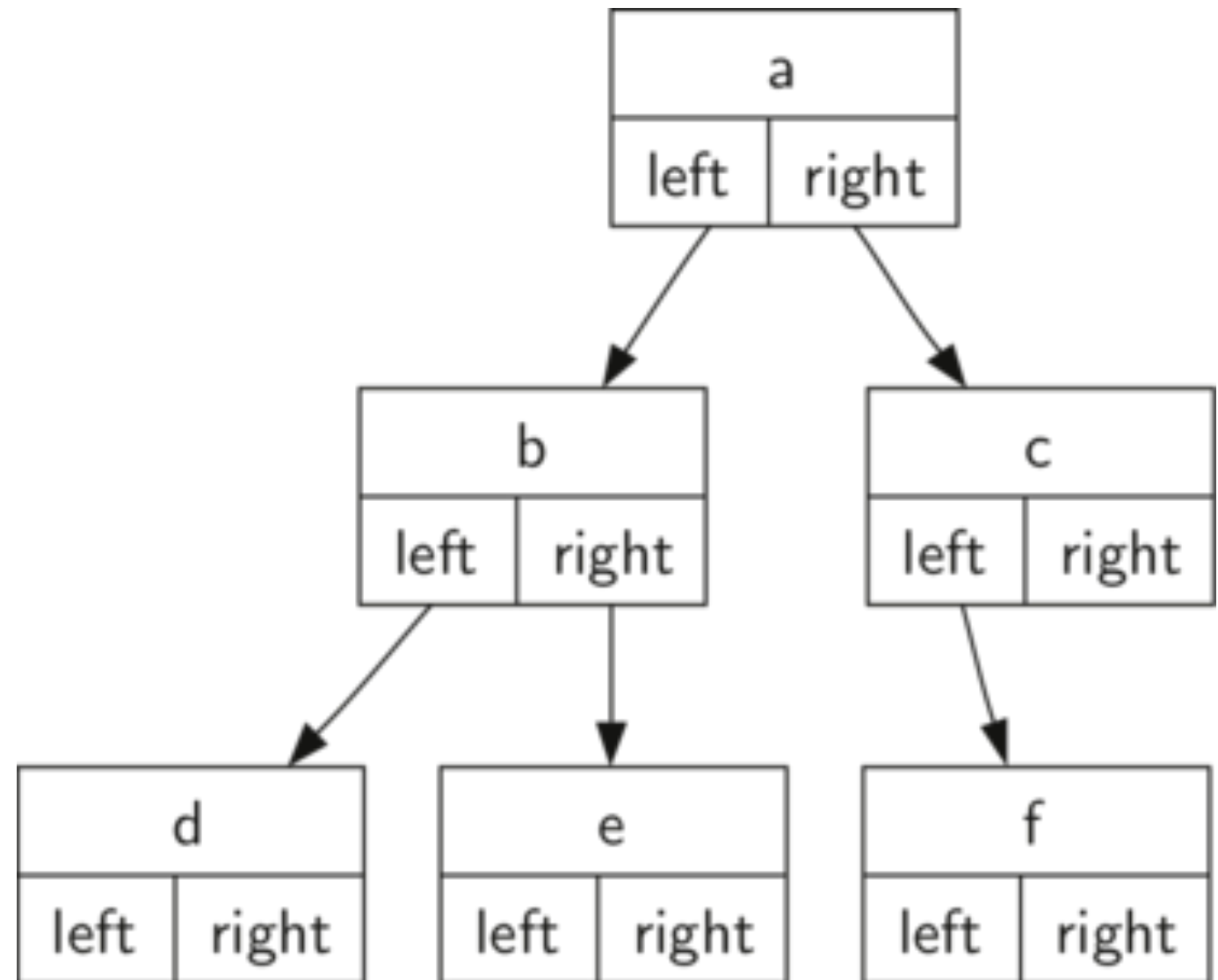
A k-ary tree where **k = 2**



How to implement binary trees?

Node:

data
left child
right child

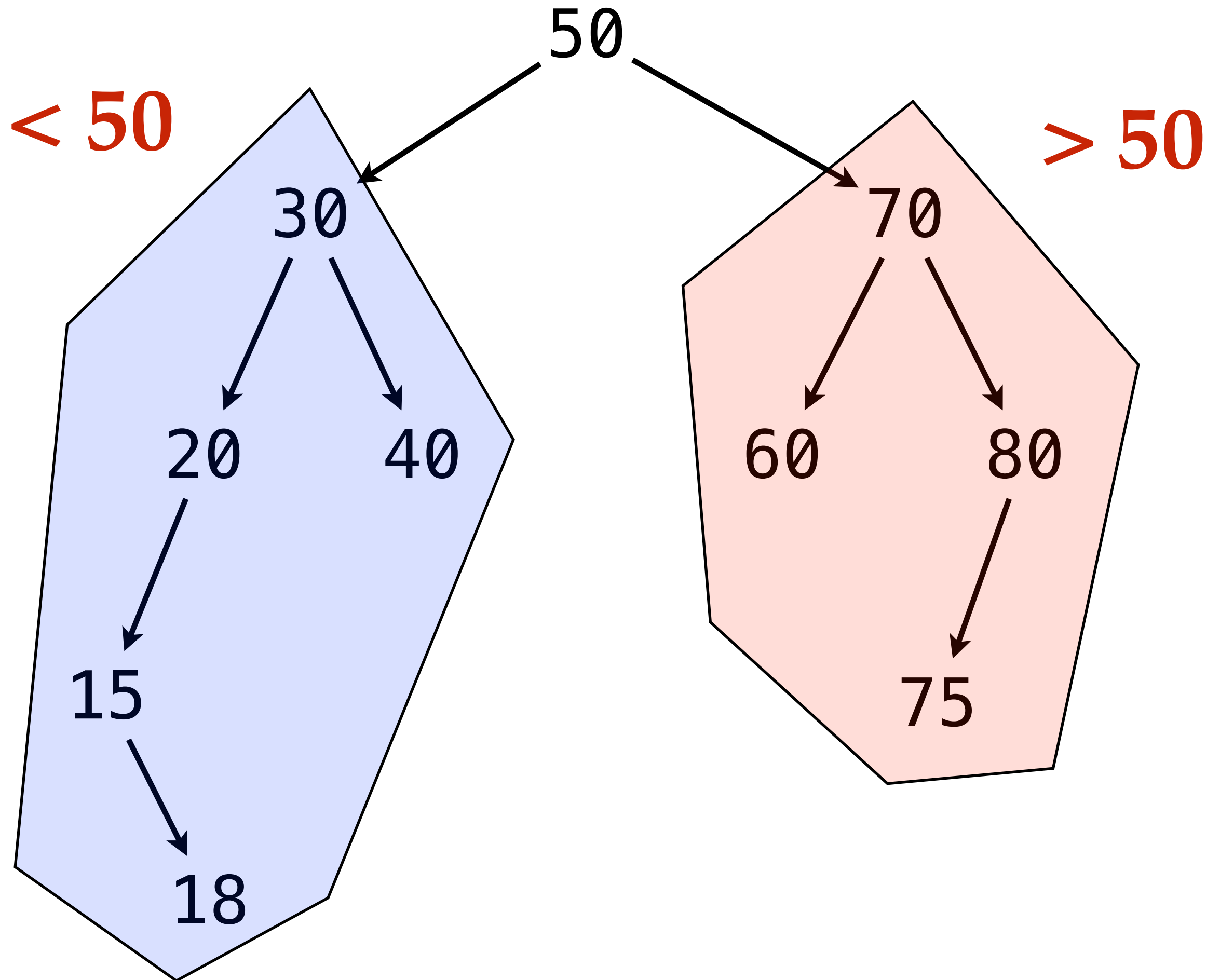


Binary Search Trees

Binary Search Tree

- A BST is a **binary tree**
- A BST has **symmetric order**
 - ✓ each node x in a BST has a key $\text{key}(x)$
 - ✓ for all nodes y in the left subtree of x , $\text{key}(y) < \text{key}(x)$ **
 - ✓ for all nodes y in the right subtree of x , $\text{key}(y) > \text{key}(x)$ **

(**) assume that the keys of a BST are pairwise distinct



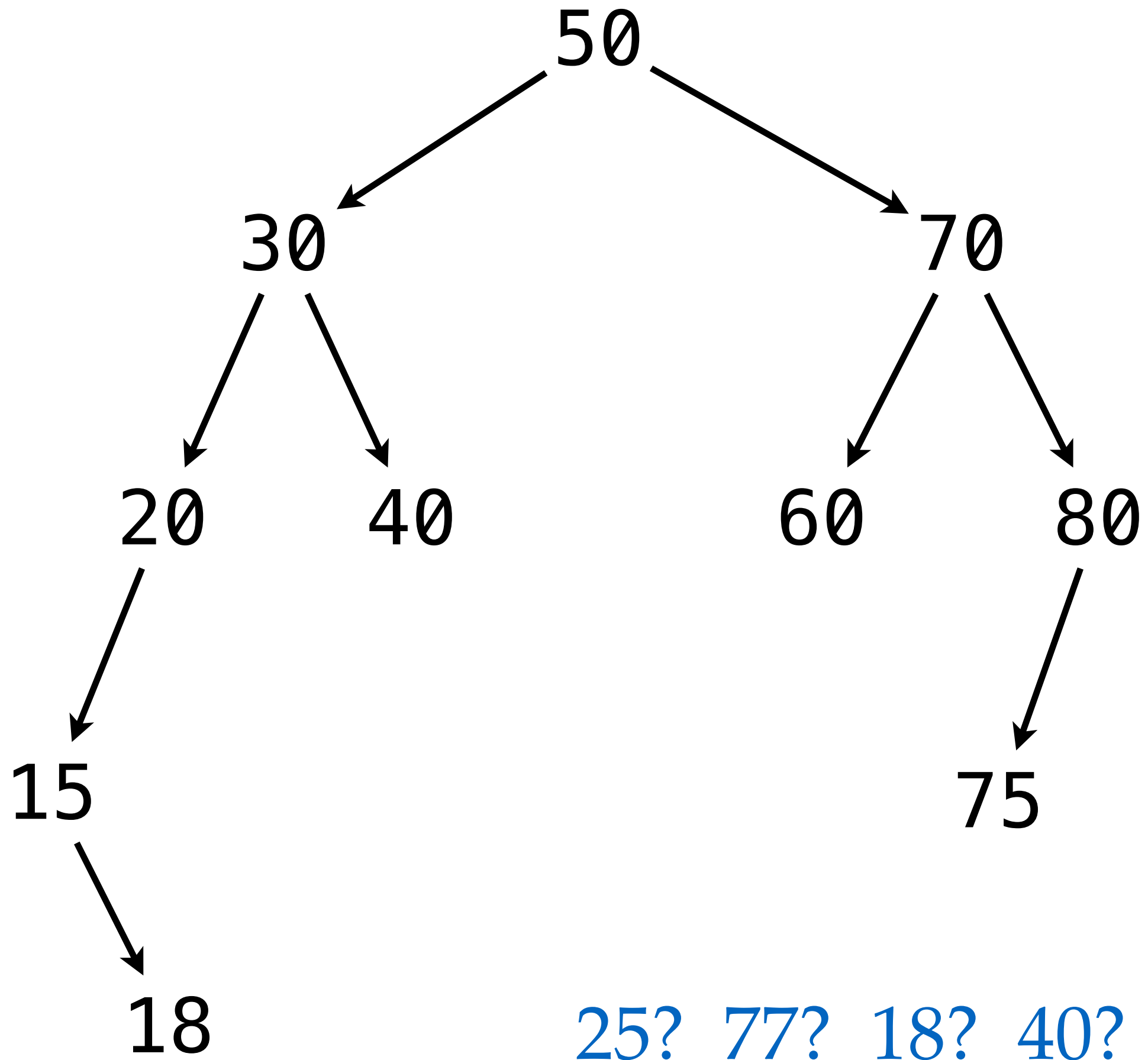
```
class BSTNode {  
  
    private:  
        int data;  
        BSTNode *left;  
        BSTNode *right;  
  
    public:  
        BSTNode(int d);  
        ~BSTNode();  
  
    friend class BSTree;  
};
```

```
class BSTree {  
    private:  
        BSTNode *root;  
        void destroy(BSTNode *p);  
  
    public:  
        BSTree();  
        ~BSTree();  
  
        void insert(int d);  
        void remove(int d);  
        BSTNode *search(int d);  
  
};
```

Search into BSTs

Search

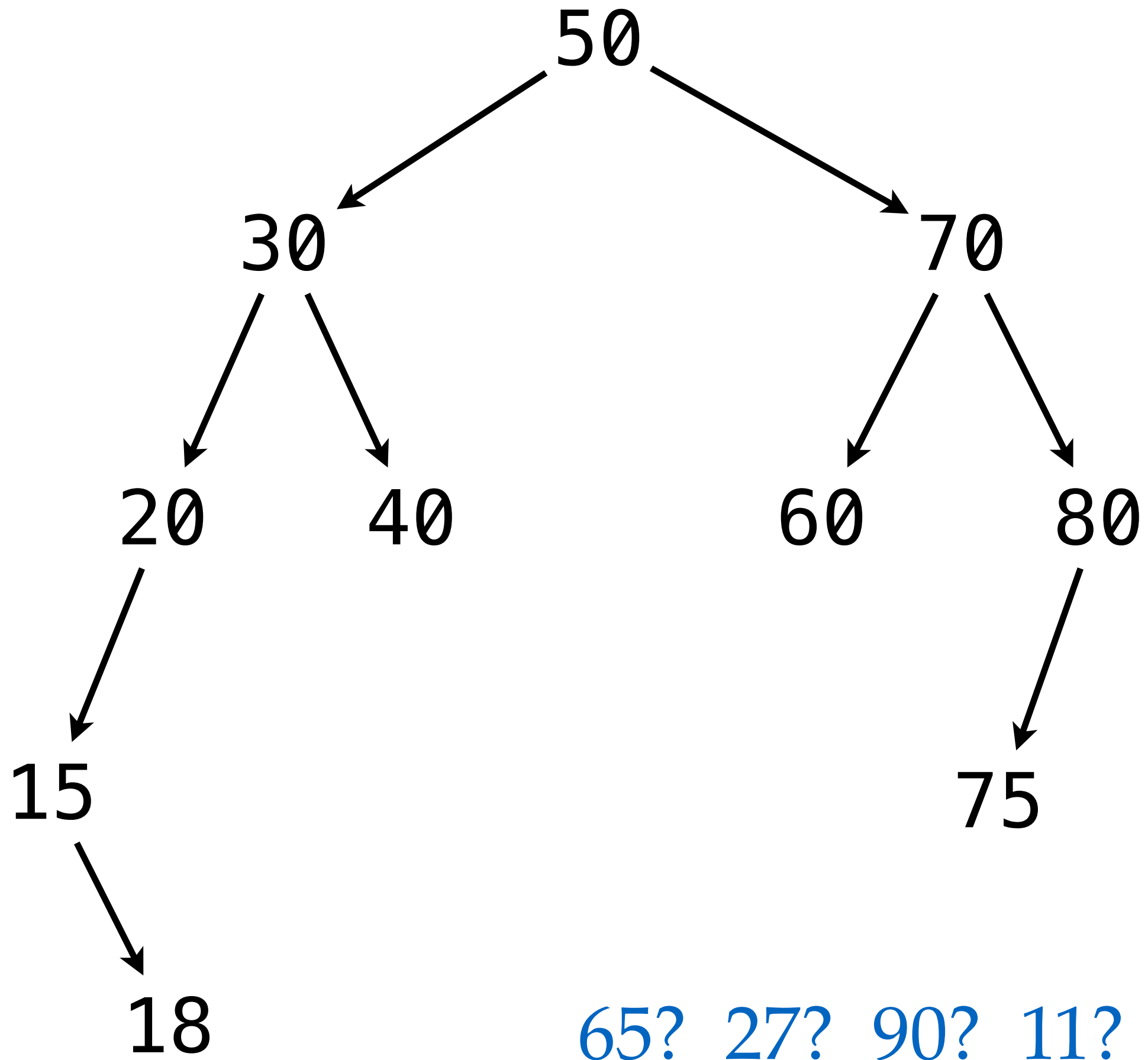
- › Start at root node
- › If the search key matches the current node's key then **found**
- › If search key is greater than current node's key
 - ✓ search on right child
- › If search key is less than current node's
 - ✓ search on left child
- › Stop when current node is NULL (**not found**)



Insert into BSTs

Insert

- Perform a Search operation
- If **found**, no need to insert (may increase counter)
- If **not found**, insert node where Search stopped



65? 27? 90? 11? 51?

Remove from BSTs

Remove

- **Case 1: node is a leaf**

- ✓ trivial, delete node and set parent's pointer to NULL

- **Case 2: node has 1 child**

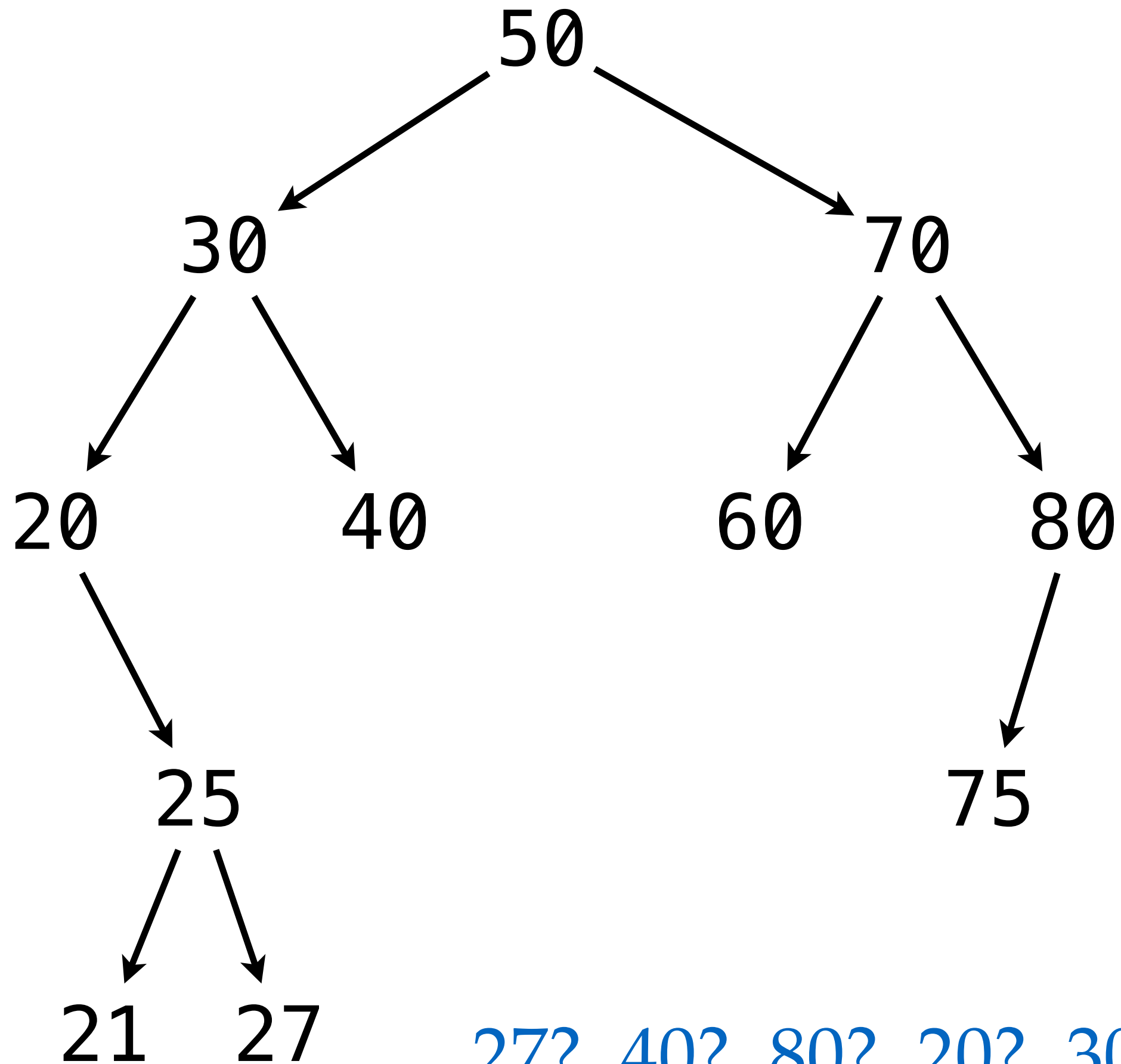
- ✓ trivial, set parent's pointer to the only child and delete node

- **Case 3: node has 2 children**

- ✓ find **successor**

can also use predecessor

- ✓ copy successor's data to node
- ✓ delete successor



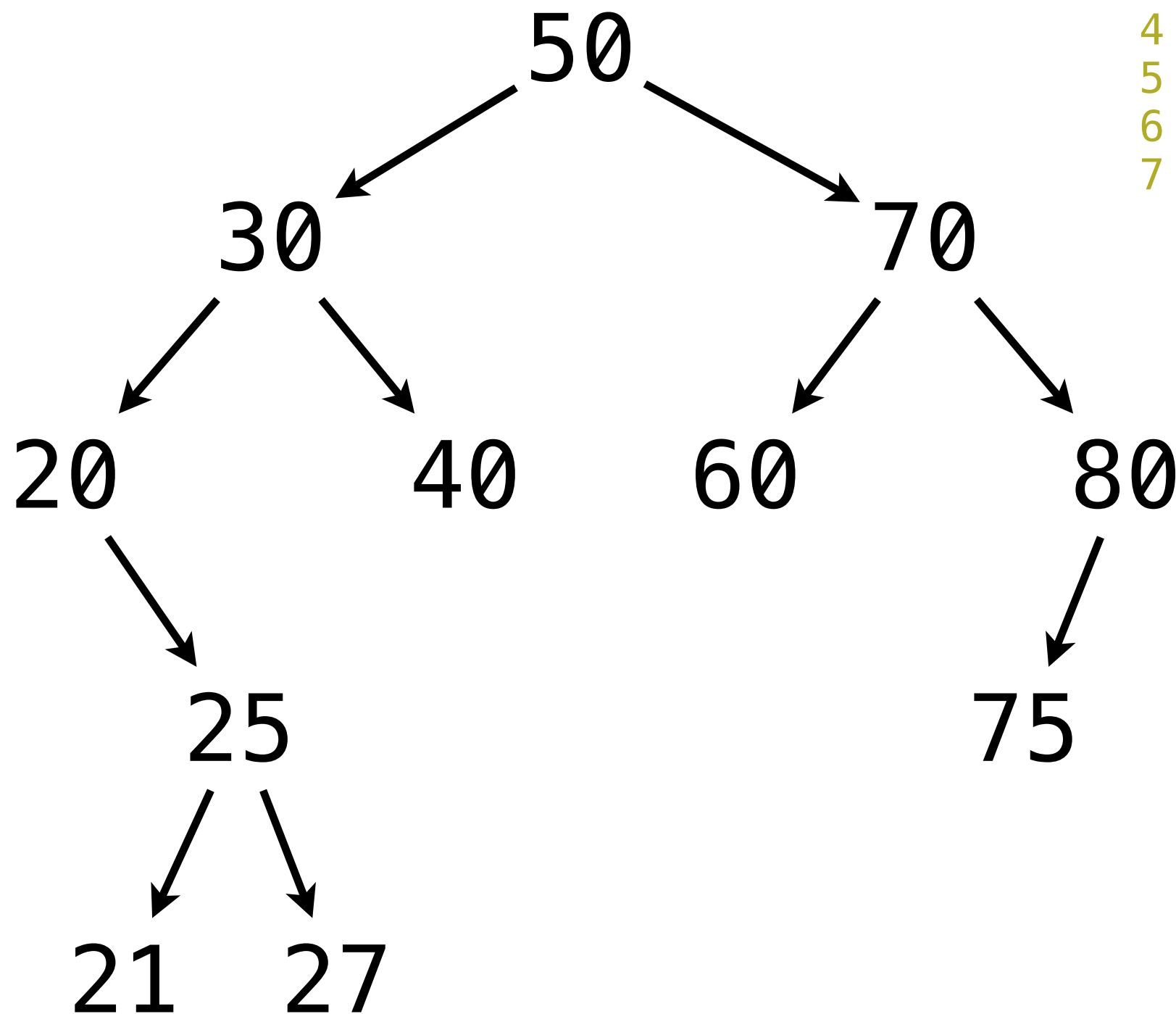
BST Traversals

Traversals

- Preorder traversal
- Inorder traversal
- Postorder traversal

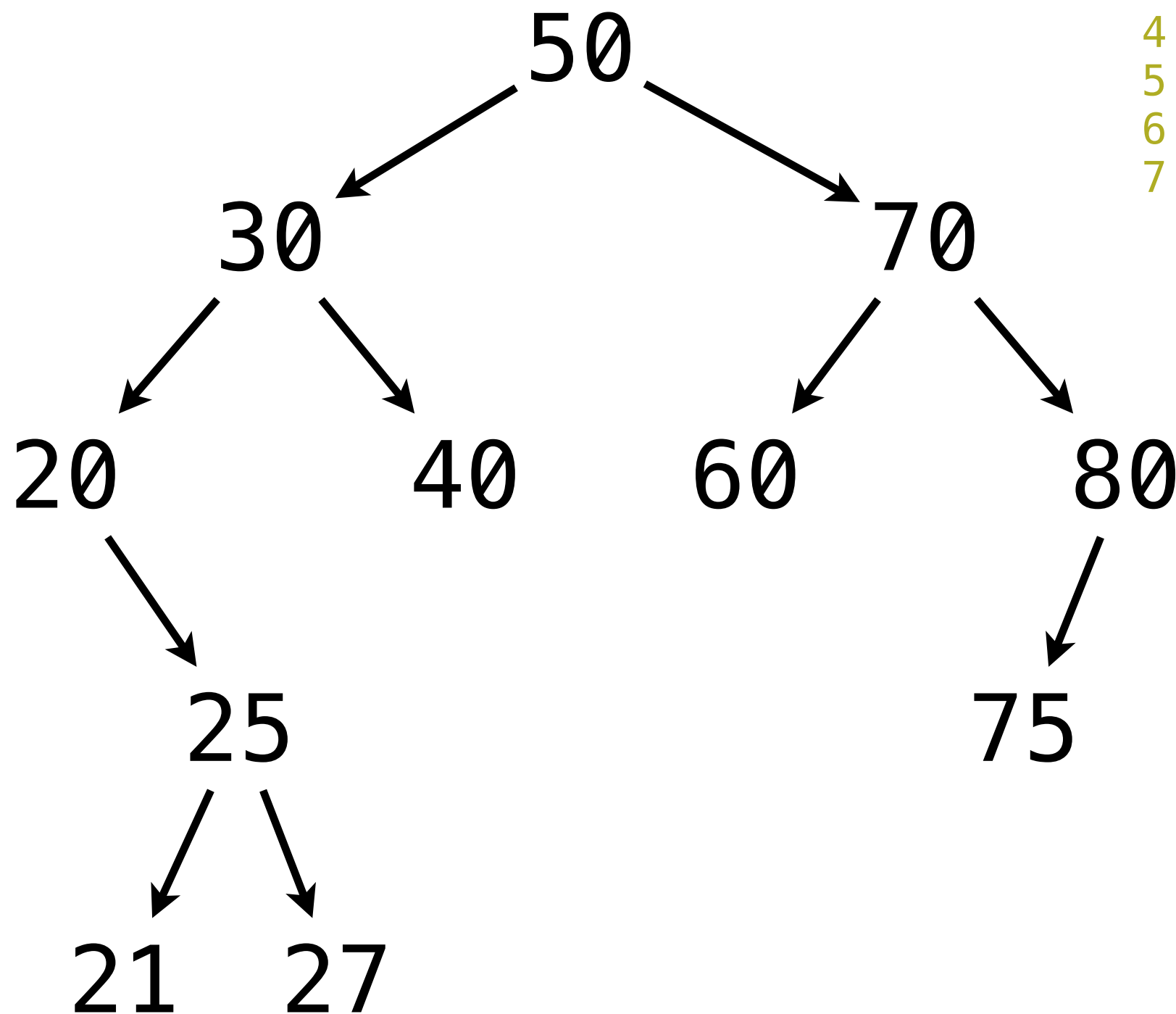

$$\Theta(n)$$

Preorder traversal?



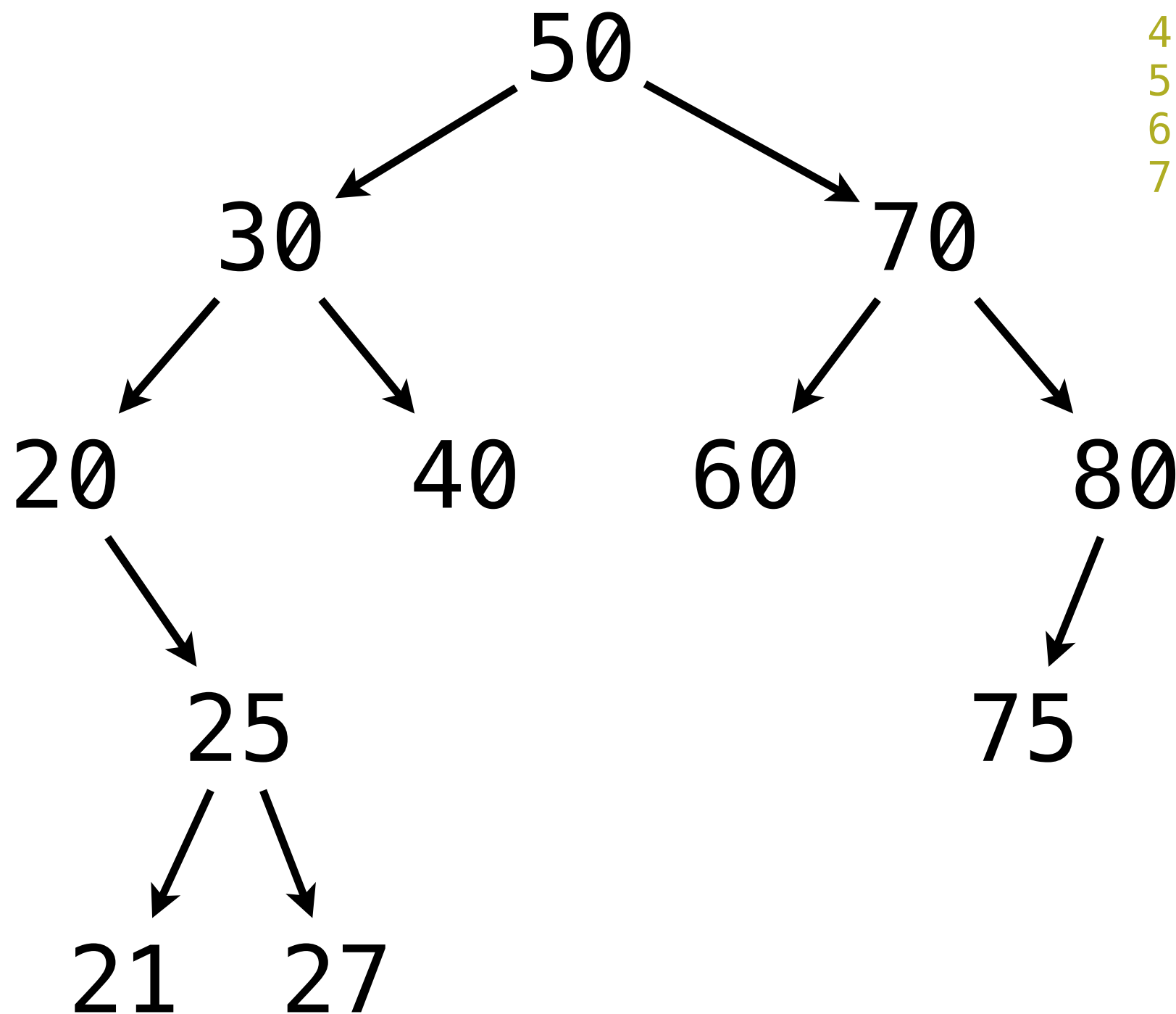
```
1 algorithm preorder(p) {  
2     if (p) {  
3         visit(p)  
4         preorder(p->left)  
5         preorder(p->right)  
6     }  
7 }
```


Postorder traversal?



```
1 algorithm postorder(p) {  
2     if (p) {  
3         postorder(p->left)  
4         postorder(p->right)  
5         visit(p)  
6     }  
7 }
```

Inorder traversal?



```
1 algorithm inorder(p) {  
2     if (p) {  
3         inorder(p->left)  
4         visit(p)  
5         inorder(p->right)  
6     }  
7 }
```

How to destroy a binary tree?

How to print all elements in
increasing order?

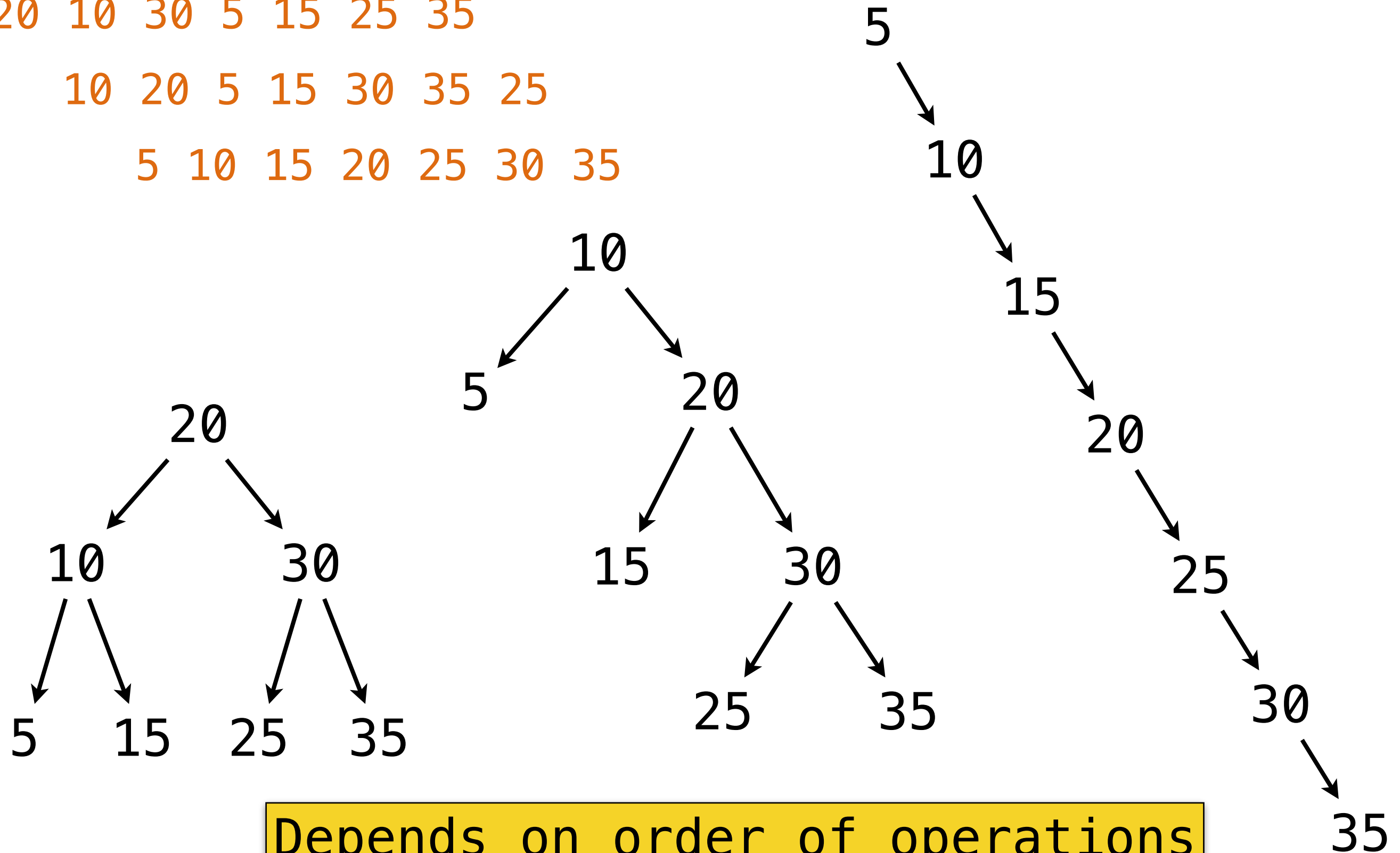
Analysis

Tree Shape?

20 10 30 5 15 25 35

10 20 5 15 30 35 25

5 10 15 20 25 30 35



Depends on order of operations

Implications

Cost of basic Operations?

- ✓ Search
- ✓ Insert
- ✓ Remove

Worst-case?

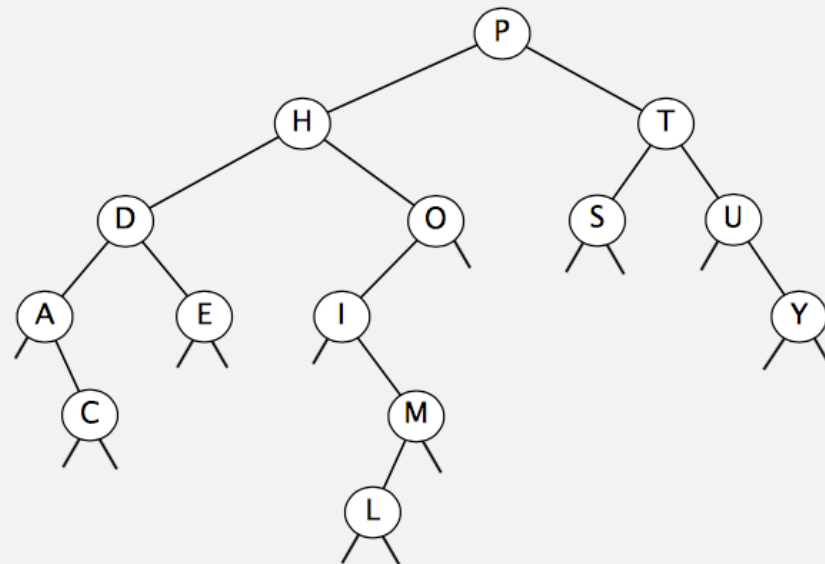
Best-case?

Average-case?

Average-case analysis

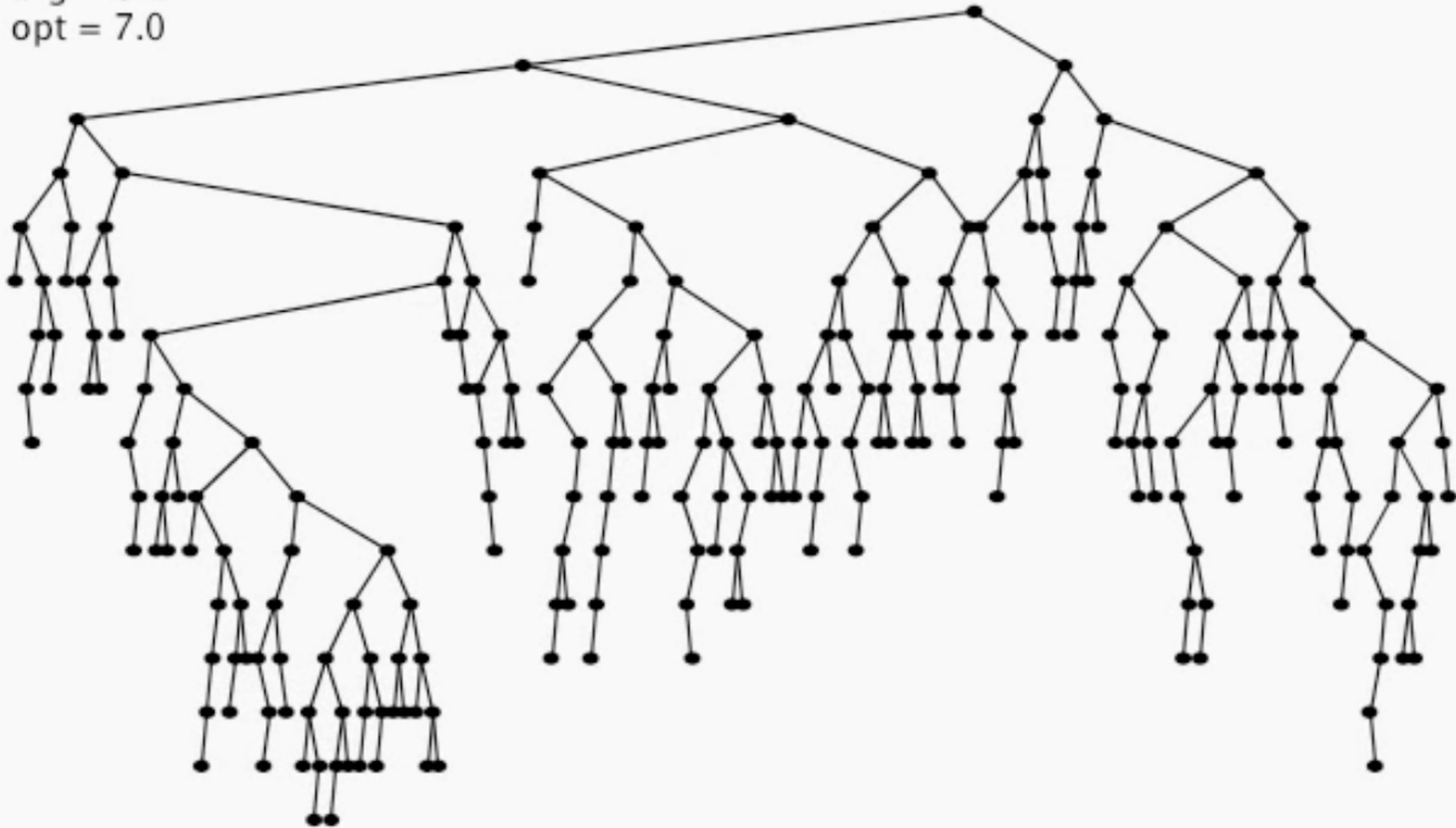
- If **n distinct keys** are inserted into a BST in random order, expected number of compares for basic operations is **$\sim 2 \ln n \sim 1.39 \log n$**
 - ✓ **proof:** 1-1 correspondence with quick-sort partitioning

0	1	2	3	4	5	6	7	8	9	10	11	12	13
P	S	E	U	D	O	M	Y	T	H	I	C	A	L
P	S	E	U	D	O	M	Y	T	H	I	C	A	L
H	L	E	A	D	O	M	C	I	P	T	Y	U	S
D	C	E	A	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	I	M	L	O	P	T	Y	U	S
A	C	D	E	H	I	M	L	O	P	T	Y	U	S
A	C	D	E	H	I	L	M	O	P	T	Y	U	S
A	C	D	E	H	I	L	M	O	P	S	T	U	Y
A	C	D	E	H	I	L	M	O	P	S	T	U	Y
A	C	D	E	H	I	L	M	O	P	S	T	U	Y
A	C	D	E	H	I	L	M	O	P	S	T	U	Y



$$h = O(\log n)$$

N = 255
max = 16
avg = 9.1
opt = 7.0



Collections / Dictionaries

	What?	Sequential (unordered)	Sequential (ordered)	BST
search	search for a key	$O(n)$	$O(\log n)$	$O(h)$
insert	insert a key	$O(n)$	$O(n)$	$O(h)$
delete	delete a key	$O(n)$	$O(n)$	$O(h)$
min/max	smallest/largest key	$O(n)$	$O(1)$	$O(h)$
floor/ ceiling	predecessor/ successor	$O(n)$	$O(\log n)$	$O(h)$
rank	number of keys less than key	$O(n)$	$O(\log n)$	$O(h)^{**}$

(**) requires the use of 'size' at every node