Questions of the day

- How can a BST guarantee that it remains balanced, no matter what values are added or removed?
- Could there be anything better than a BST with guaranteed best-case performance?

AVL Trees and hash tables

William Hendrix

Lecture 4

Outline

- Review
  - Lists
  - Stacks and queues
  - BSTs
- Balanced BSTs
- AVL trees
- Hash tables
- Collisions
- Hash functions

Review: lists

- List: ADT that stores and accesses values based on index
  - Implemented as array or linked list

Operation	Array-based (shifting)	Array-based (swapping)	LinkedList (standard)	LinkedList (given pointers)
Get(i)	$\Theta(1)$	$\Theta(1)$	$O(n)$	0
Set(i, x)	$\Theta(1)$	$\Theta(1)$	$O(n)$	$\Theta(1)$
Insert(i, x)	$O(n)$	$\Theta(1)^*$	$O(n)$	$\Theta(1)$
Delete(i)	$O(n)$	$\Theta(1)$	$O(n)$	$\Theta(1)$
Append(x)	$\Theta(1)^*$	$\Theta(1)^*$	$\Theta(1)$	$\Theta(1)$

* Amortized complexity

- Array-based implementations get and set in $\Theta(1)$ time
  - Can insert and delete in $\Theta(1)$ if willing to swap
- LinkedLists usually slower
  - Can insert and delete in $\Theta(1)$ if given pointers
  - Some operations (like concatenation) are faster

Stack/queue review

- Data structures that access newest/oldest element added
  - Array implementations generally preferred
  - Linked list possible

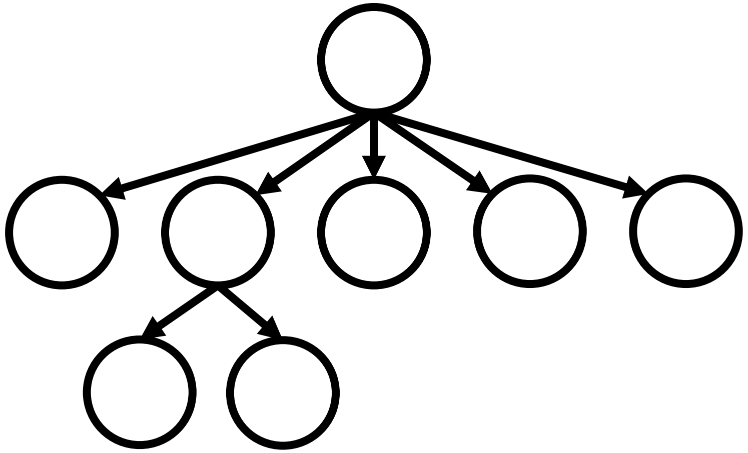
Operation	Stack	Queue	Deque
Push(x) / Enqueue(x)	$\Theta(1)$*	$\Theta(1)$*	$\Theta(1)$*
Pop() / Dequeue()	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Peek()	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

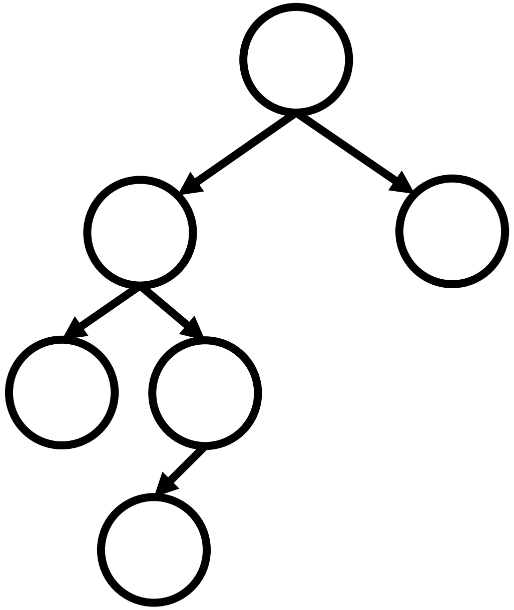
* amortized time

- All operations are constant amortized time
- Do not support random access
- Choice of stack/queue is based on desired access order
  - Stack: Last In, First Out (LIFO)
  - Queue: First In, First Out (FIFO)
  - Deque: Can simulate either

Review: trees

- Nonlinear linked data structure
- From graph theory: any connected acyclic graph
- Most trees in CS are *rooted*
  - Special vertex
  - Other vertices are defined in relation to root
  - Root has children, children have children, etc.
- *Leaf*: vertex with degree 1 (no children)
- *Level*: nodes at same distance from root
- *Height*: max level
- *Siblings*: vertices with same parent
- Binary tree
  - Tree where every node has at most 2 children
    - *left* and *right*
  - *m*-ary tree: every node has at most *m* children
  - *Full* binary tree: all non-leaf nodes have 2 children

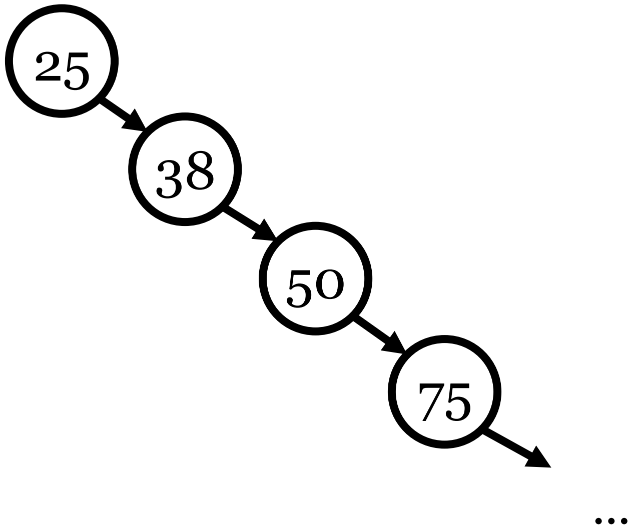




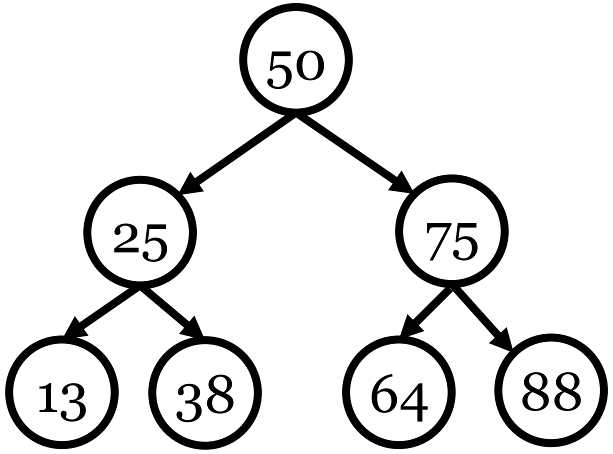
Tree height

- How tall can a BST be?

- **Worst case**
  - Degenerate tree
    - Only left or right children
    - Height = O(n)
  - Can happen if inserting in sorted order

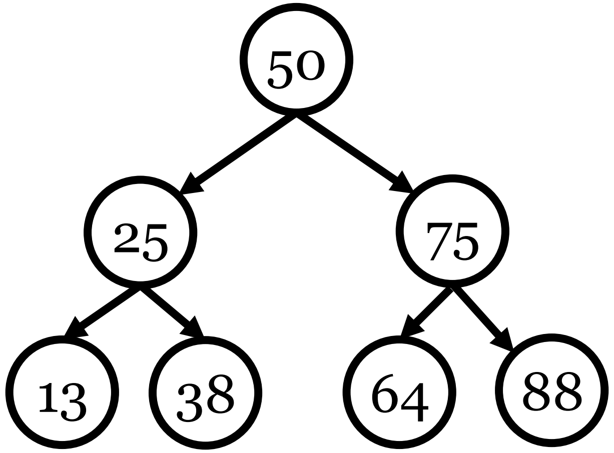


- **Best case**
  - Complete tree



Complete tree height

- **Theorem:** Complete trees have 2^i nodes on level i, for every level (except possibly bottom level)
  - Proof sketch:
    - $1 = 2^0$ on level 0
    - Every subsequent level has twice as many as previous



- # nodes in complete binary tree of height h: $\sum_{i=0}^{h} 2^i$
    - Exponential identity: $\Theta(2^h)$
  - # nodes in complete tree w/ height h is $\Theta(2^h)$
    - Height of tree with n nodes is $\Theta(\lg n)$
- **Theorem:** half of all nodes in complete tree are leaves

BST complexity

	BST (best)	BST (worst)	Unsorted array	Sorted array
Search(x)	$\Omega(\lg n)$	$O(n)$	$O(n)$	$\Theta(\lg n)$
Insert(x)	$\Omega(\lg n)$	$O(n)$	$\Theta(1)$	$O(n)$
Delete(x)	$\Omega(\lg n)$	$O(n)$	$\Theta(1)$*	$O(n)$
Min()	$\Omega(\lg n)$	$O(n)$	$\Theta(n)$	$\Theta(1)$
Max()	$\Omega(\lg n)$	$O(n)$	$\Theta(n)$	$\Theta(1)$

* Swap to end and decrement size

- BST have great best-case complexity
- Worst case is worse than arrays!

Balanced BSTs

- BST variants that *guarantee* O(lg *n*) height
  - All ops have great complexity
- Three common balanced BSTs
  - AVL trees
    - Nodes keep track of *balance*
    - Balance = height(*right*) - height(*left*)
    - Force balance to be 0, +1, or -1
  - Red-black trees
    - Nodes are assigned color (red or black)
    - Path to every leaf has same # black nodes
    - No two red nodes in a row
  - Splay trees*
    - Nodes move to the top of the tree as they are requested
- Enforce guarantee by adjusting tree as elements added and removed
  - *Main mechanism:* tree rotations

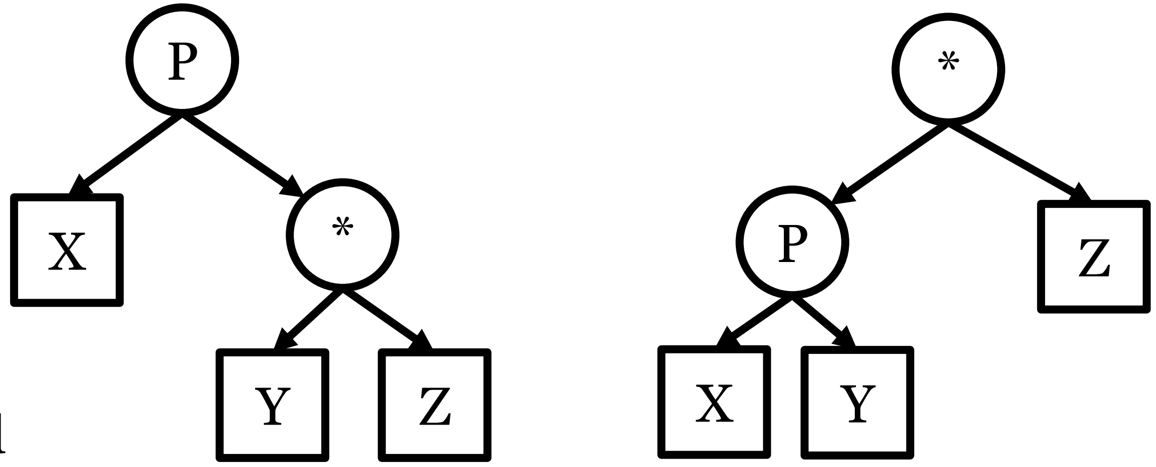
Tree rotations

- Based around a *pivot* node
- Two directions: *left* and *right*

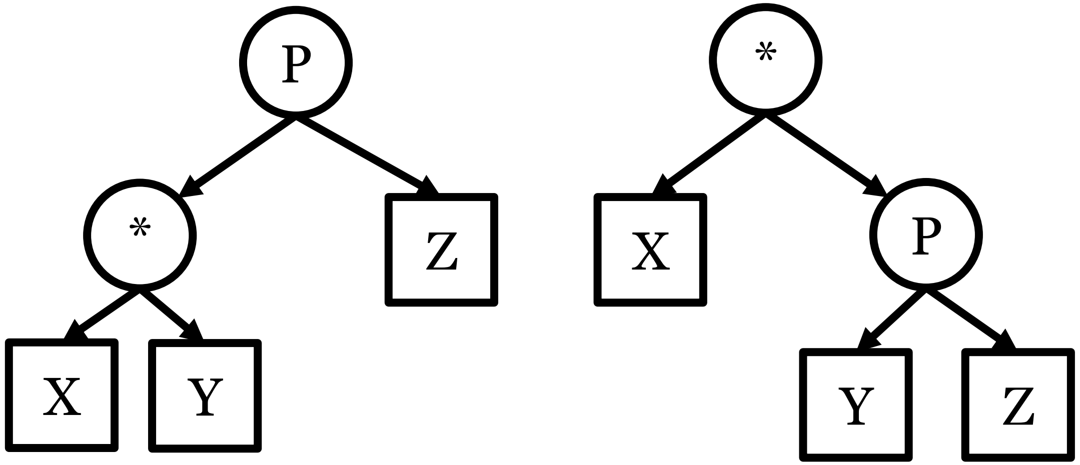
- **Left rotation:**
  - Parent becomes left child
  - Pivot becomes parent
  - Old left child becomes parent right child
  - Pivot is new "root" (fix parent)

- **Right rotation:**
  - Parent becomes right child
  - Pivot becomes parent
  - Old right child becomes parent left child
  - Fix pivot's new parent

Left rotation

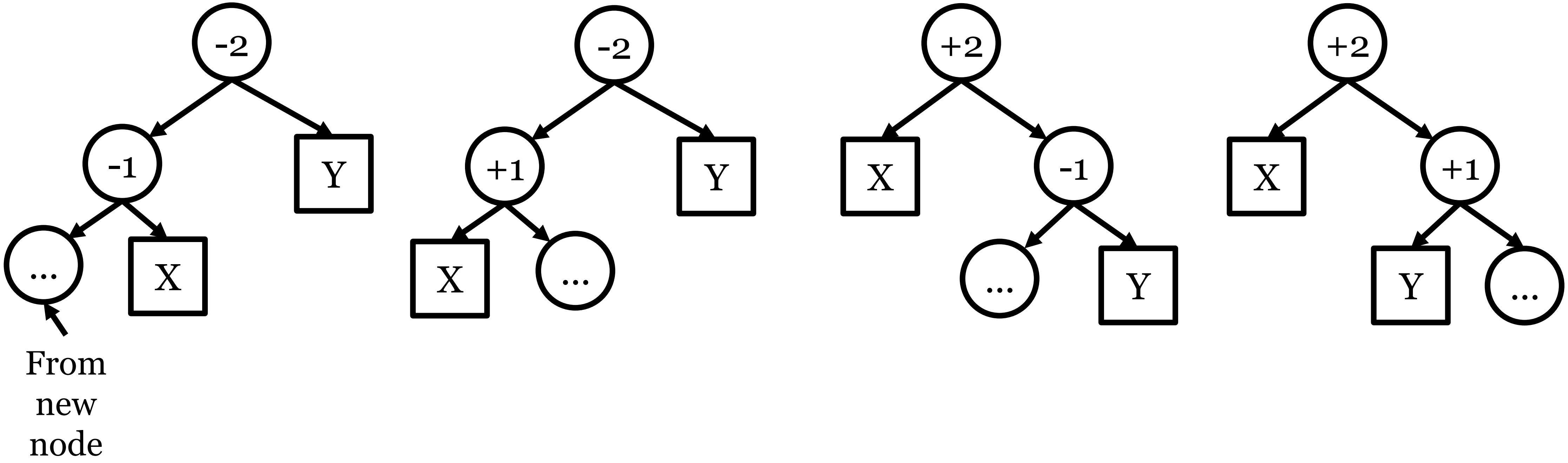


Right rotation



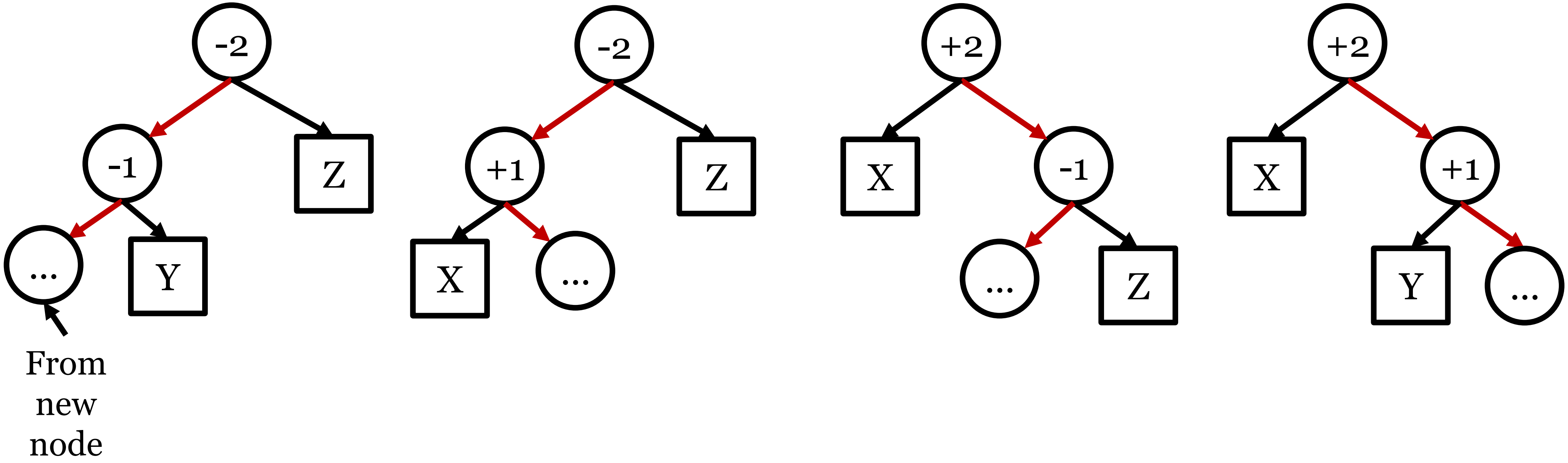
AVL tree insertion

- Leaves have balance 0
- Parent balance: add +1 (right) or -1 (left)
- If parent balance = 0: stop
- If parent balance = ± 1, repeat with parent's parent
- If parent balance = ± 2, *fix the tree!*
  - Four cases:



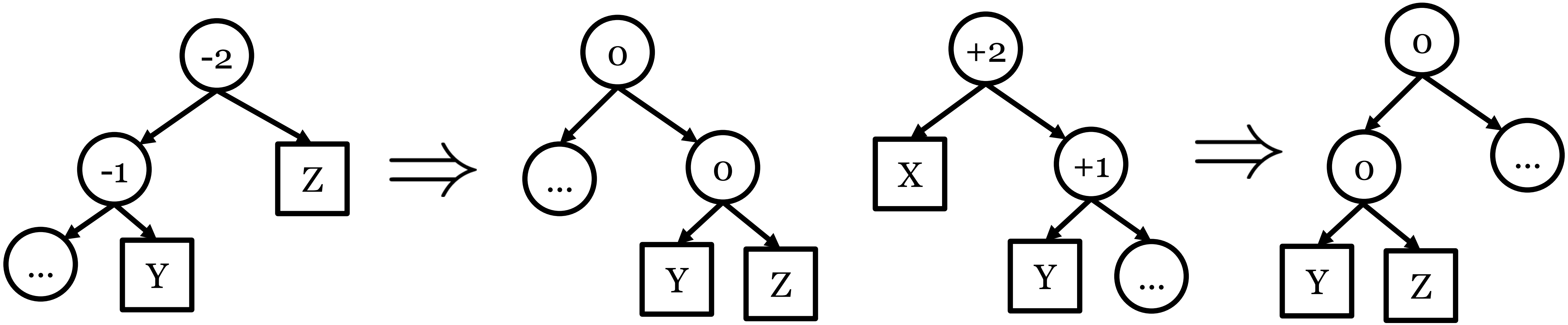
AVL tree insertion

- Leaves have balance 0
- Parent balance: add +1 (right) or -1 (left)
- If parent balance = 0: stop
- If parent balance = ± 1, repeat with parent's parent
- If parent balance = ± 2, *fix the tree!*
  – Four cases:



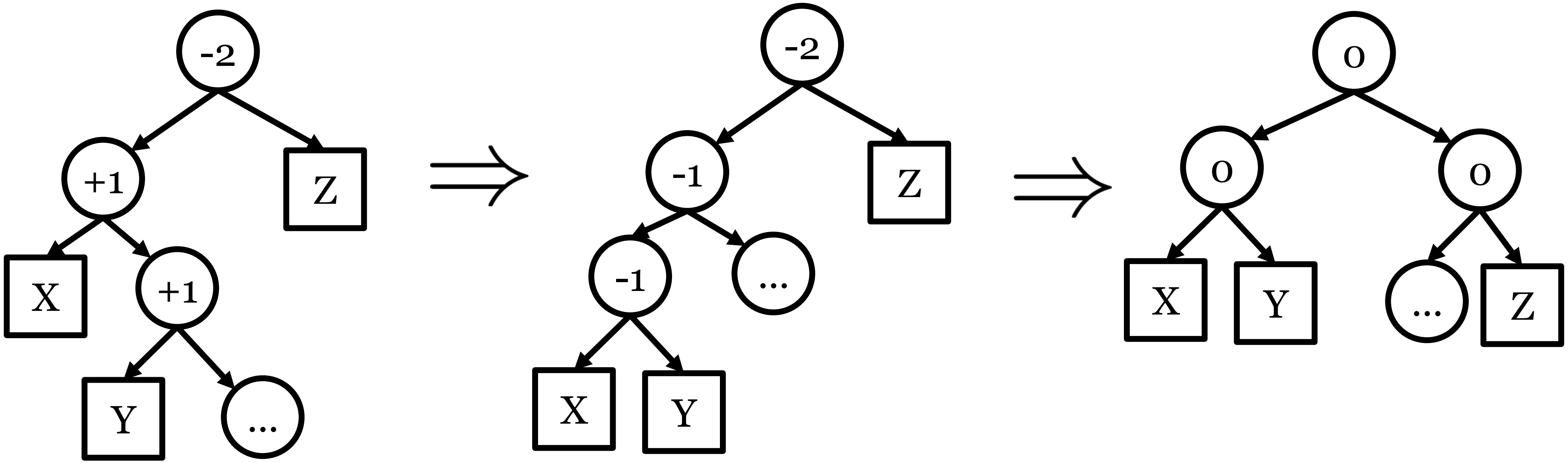
Balancing an AVL tree: left-left and right-right

- If directions match:
  - Rotate current node in opposite direction
  - Current and parent nodes balanced



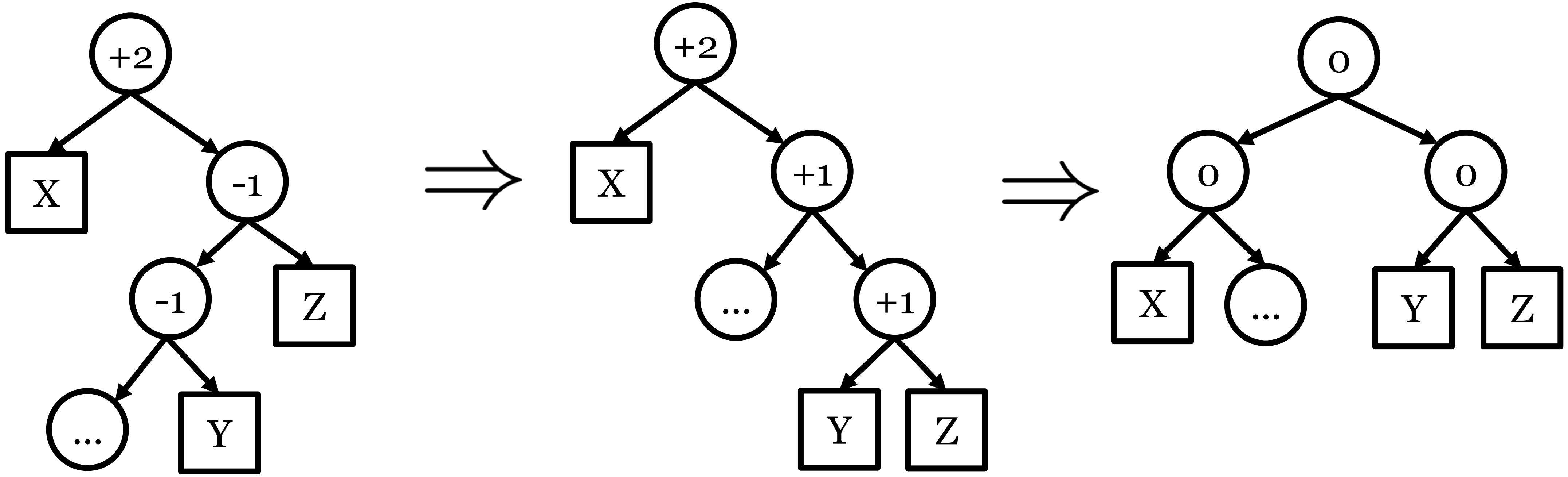
Balancing an AVL tree: left-right and right-left

- If directions are opposite:
  - Rotate child node towards current
  - Rotate that node towards parent
  - All three nodes balanced



Balancing an AVL tree: left-right and right-left

- If directions are opposite:
  - Rotate child node towards current
  - Rotate that node towards parent
  - All three nodes balanced



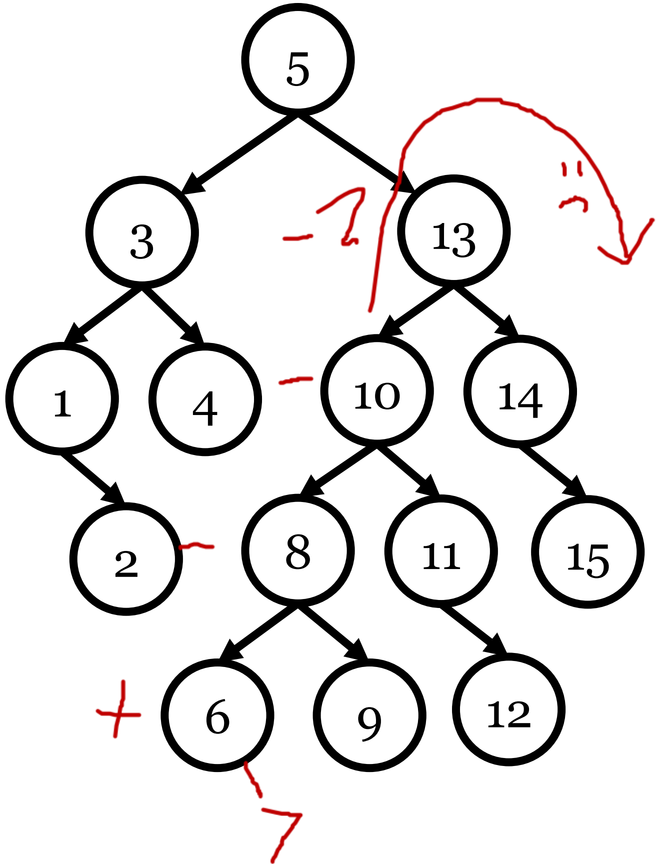
AVL tree insertion

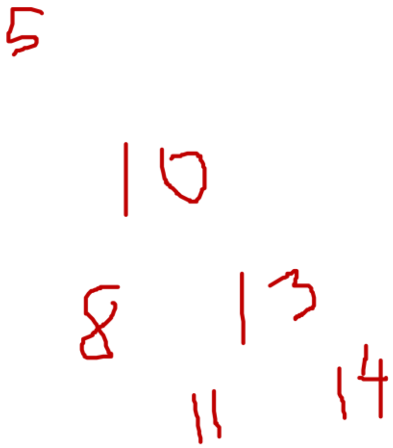
1. Insert new node with normal BST insert (balance = 0)
2. Go to parent of new node
3. If you just came from left child, balance--
4. Otherwise, balance++
5. 3 cases for balance:
   a) If balance = 0: **stop**
   b) If balance = ±1: go to parent and repeat from step 3
   c) If balance = ±2: 4 cases
      a) If balance negative and left child balance negative (<u>left-left</u>): rotate left child right
      b) If balance positive and right child balance positive (<u>right-right</u>): rotate right child left
      c) If balance negative and left child balance positive (<u>left-right</u>): rotate left child's right child left, then right
      d) If balance positive and right child balance negative (<u>right-left</u>): rotate right child's left child right, then left
   d) Stop

<u>Path towards inserted node</u>

AVL insertion exercise

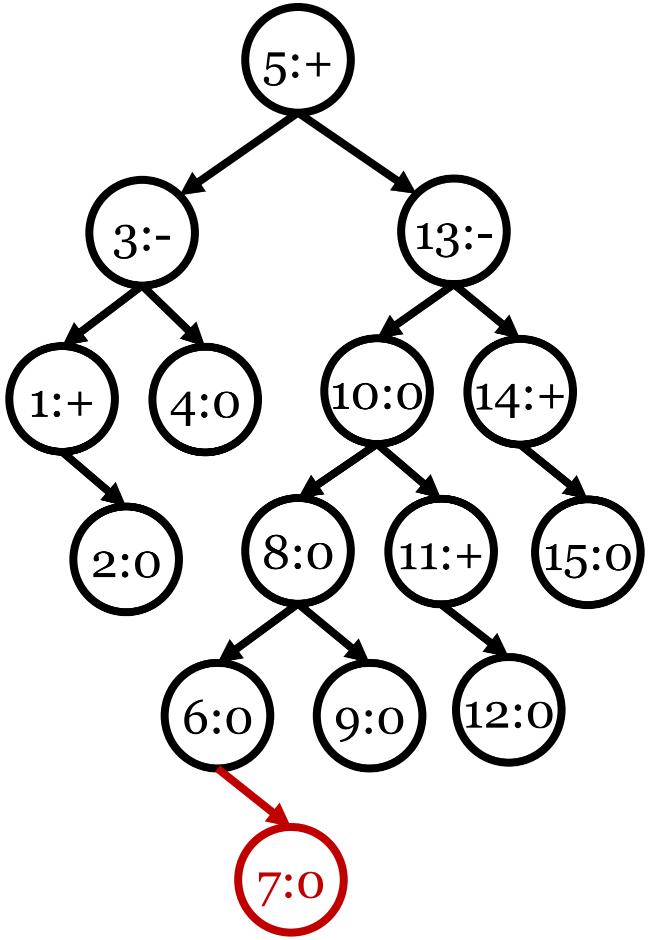
- Insert 7 into the AVL tree below:





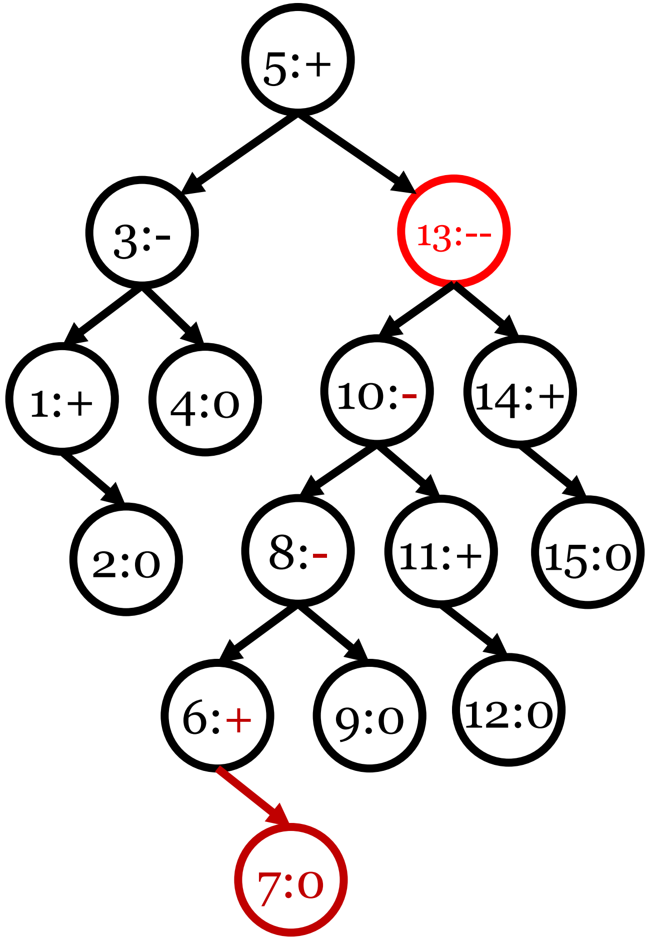
AVL insertion exercise

- Insert 7 into the AVL tree below:



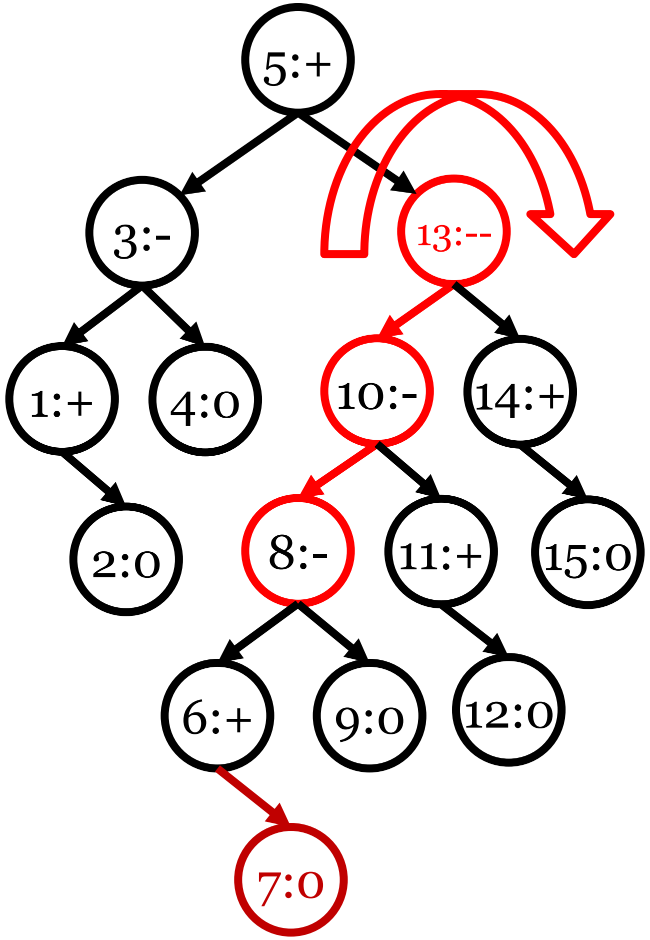
AVL insertion exercise

- Insert 7 into the AVL tree below:



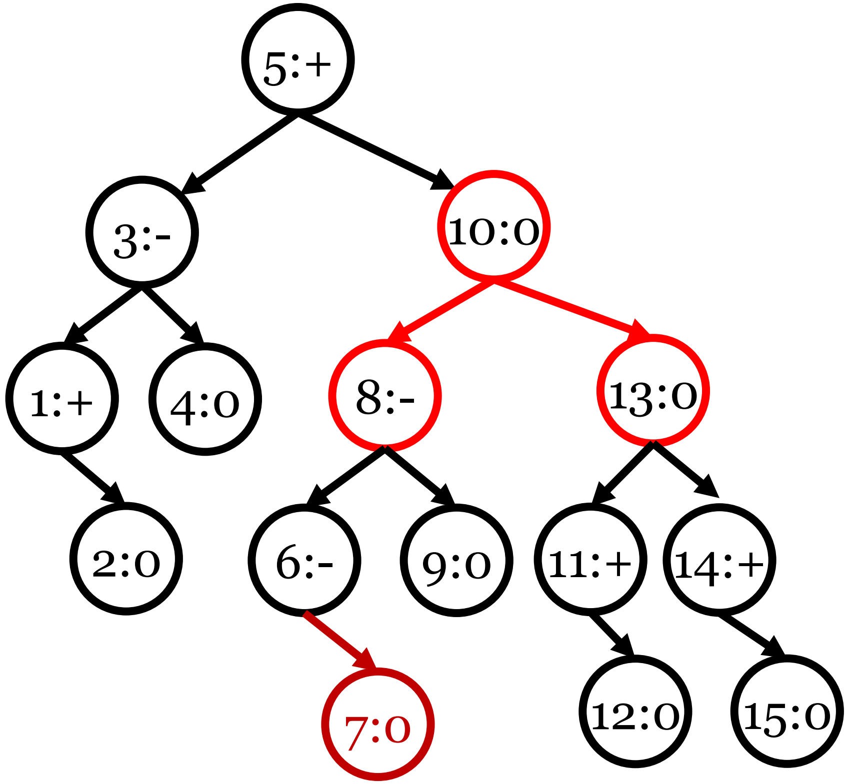
AVL insertion exercise

- Insert 7 into the AVL tree below:



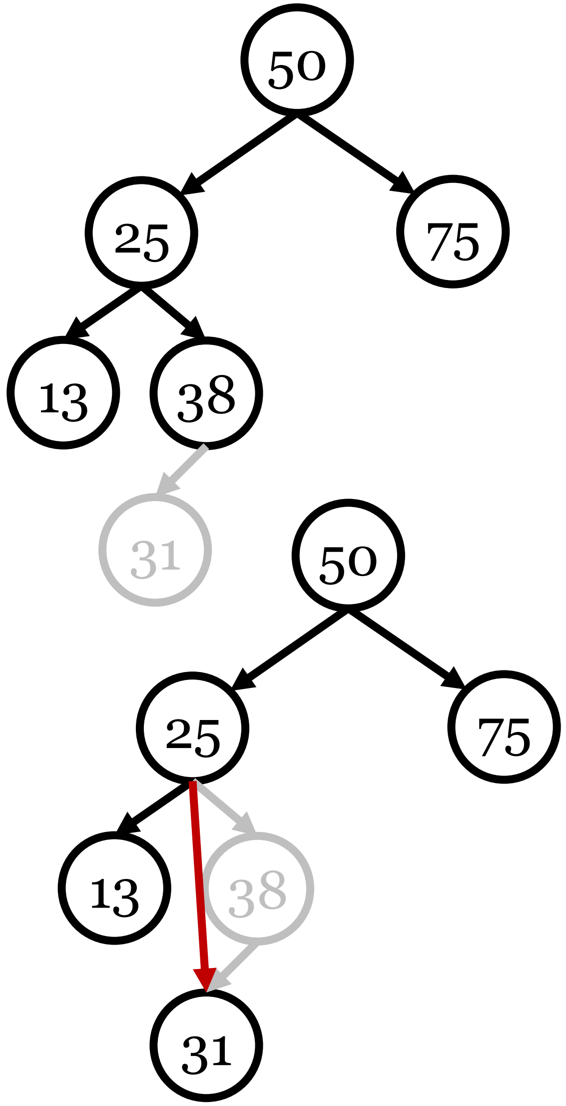
AVL insertion exercise

- Insert 7 into the AVL tree below:



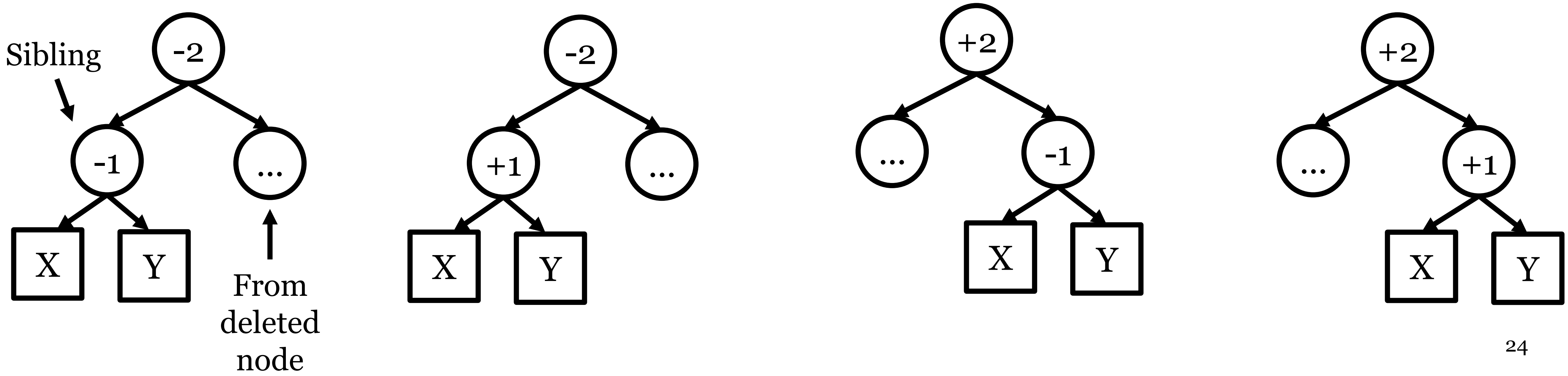
AVL tree deletion

- Similar to insertion
- Perform deletion as normal
- Update parent balance
  - Decrement if *right* child
  - Increment if *left* child
- If balance = ± 1, no further changes
- If balance = 0, continue updating parent
- If balance = ± 2, fix the tree



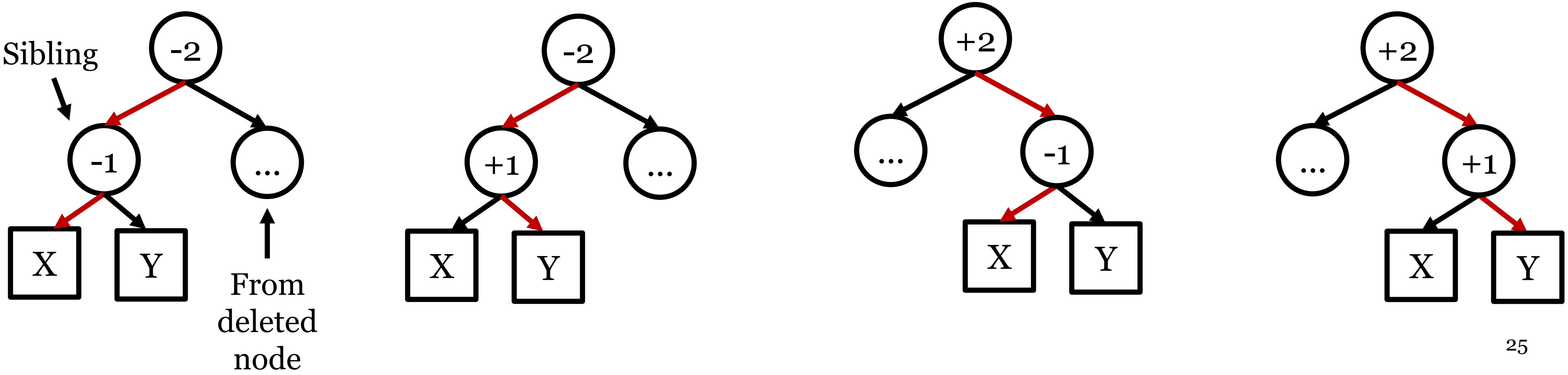
Rebalancing for deletion

- Cases are similar to insertion
  - Based on *sibling* node instead
- Rotate sibling based on its and parent's balance



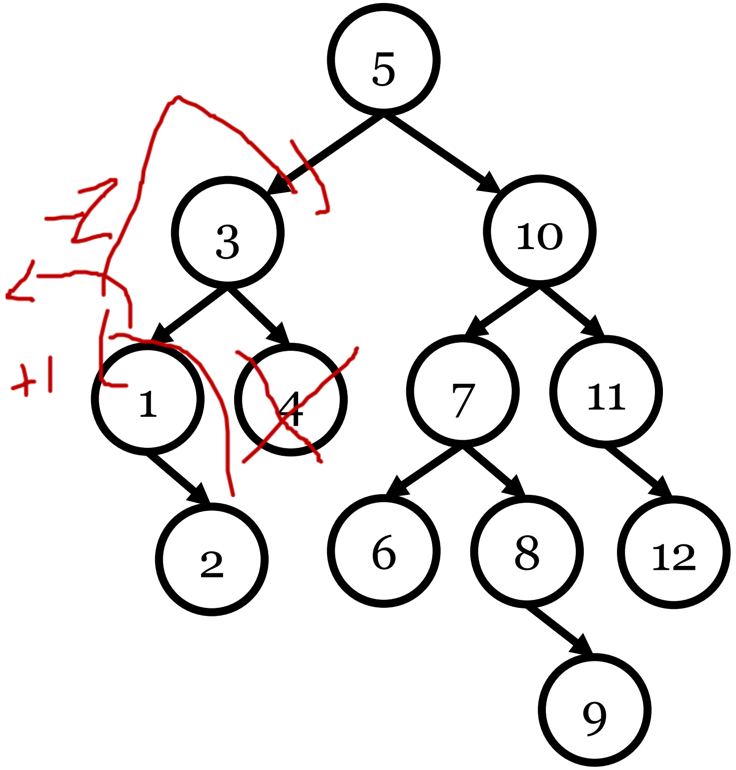
Rebalancing for deletion

- Cases are similar to insertion
  - Based on *sibling* node instead
- Rotate sibling based on its and parent's balance
  - -/-: right rotation
  - -/+: left, then right rotation of grandchild
  - +/-: right, then left rotation of grandchild
  - +/+: left rotation
  - New root balance = 0, so continue upwards



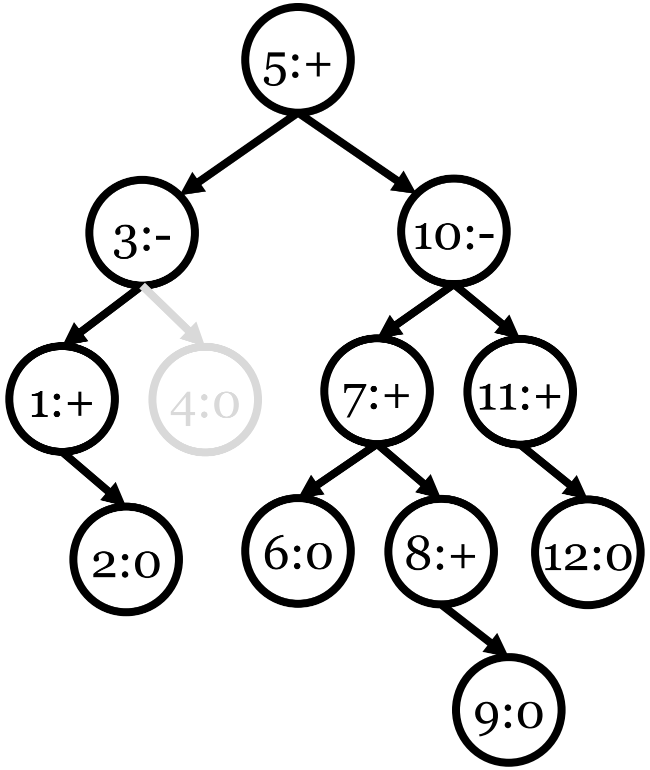
AVL deletion exercise

- Delete the 4 in the AVL tree below:



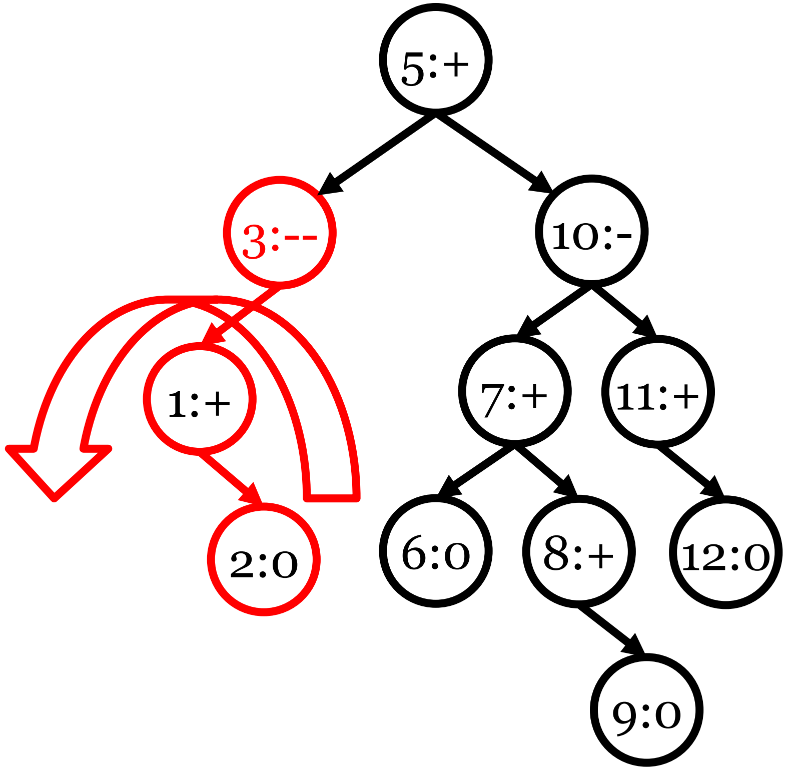
AVL deletion exercise

- Delete the 4 in the AVL tree below:



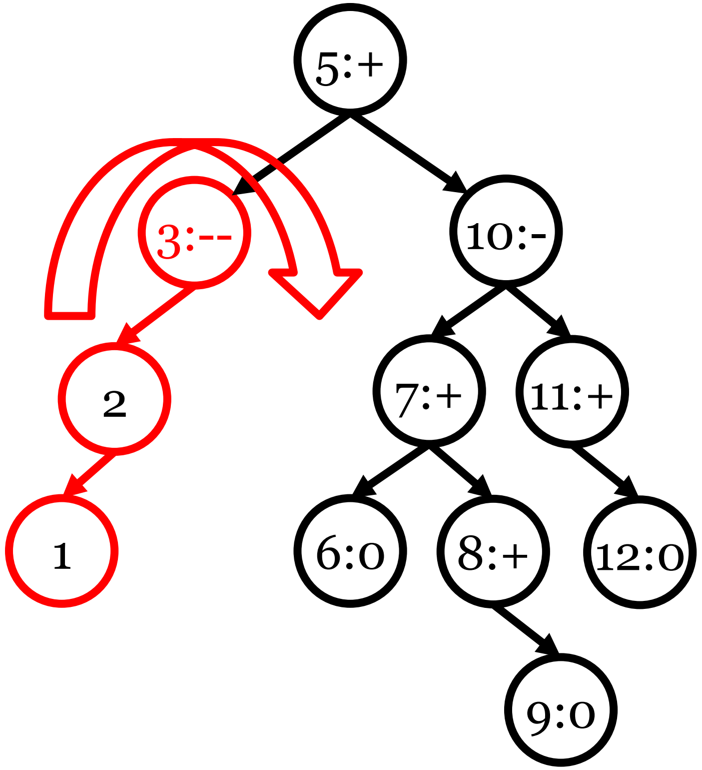
AVL deletion exercise

- Delete the 4 in the AVL tree below:



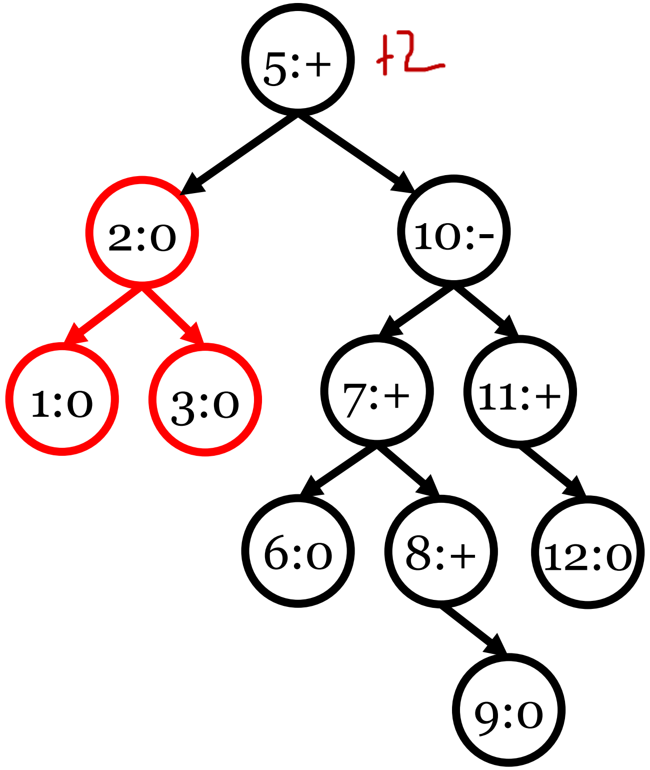
AVL deletion exercise

- Delete the 4 in the AVL tree below:



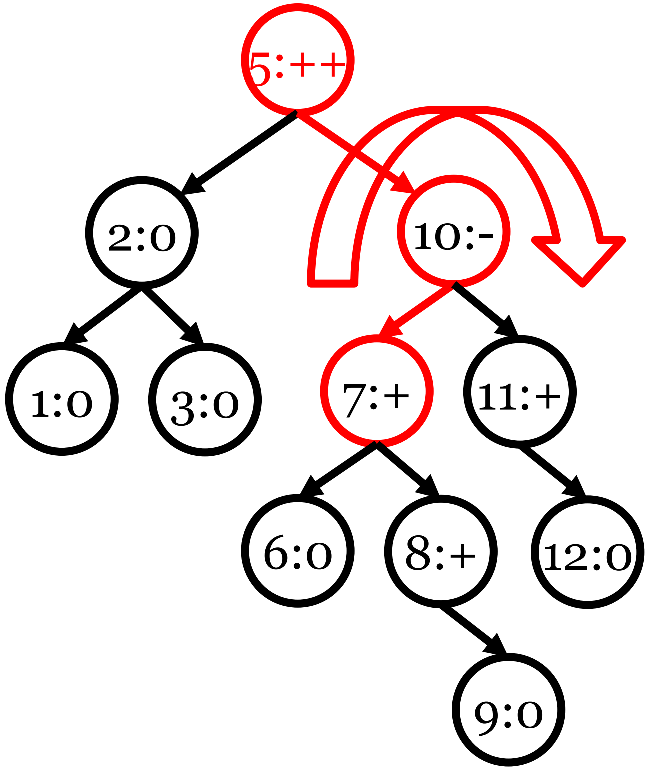
AVL deletion exercise

- Delete the 4 in the AVL tree below:



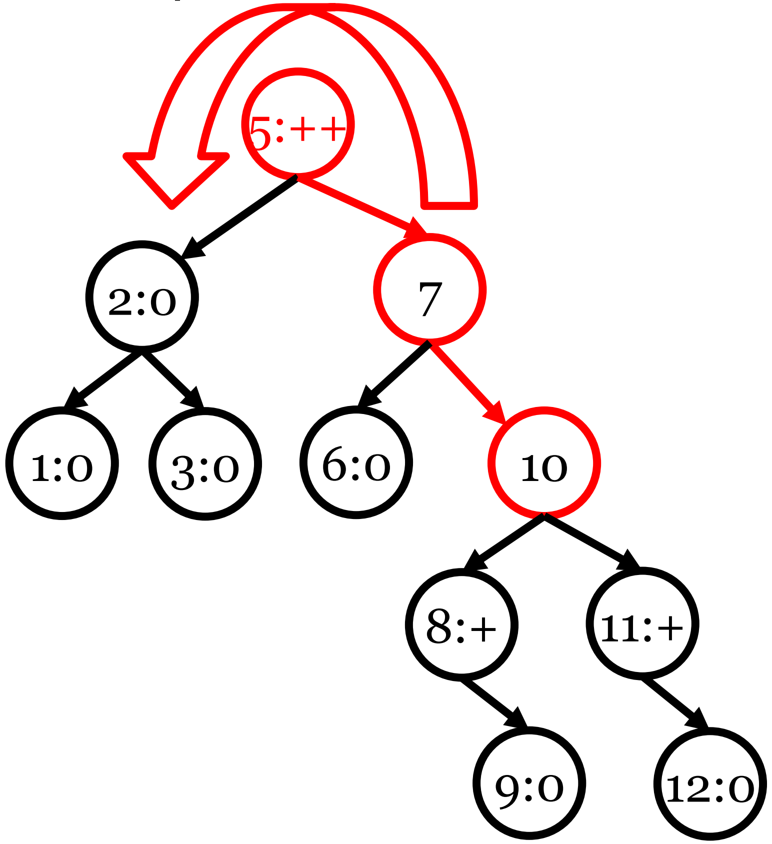
AVL deletion exercise

- Delete the 4 in the AVL tree below:



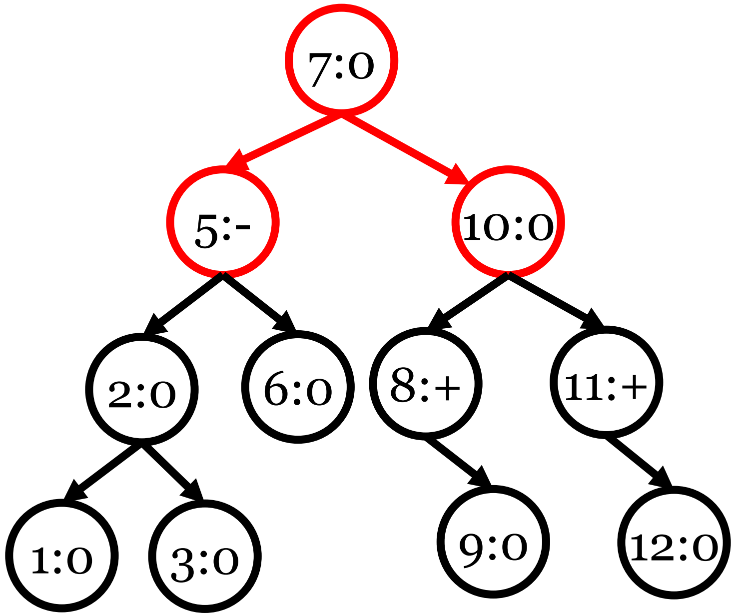
AVL deletion exercise

- Delete the 4 in the AVL tree below:



AVL deletion exercise

- Delete the 4 in the AVL tree below:



Hash table

- Sparse array-based data structure
- Insert elements according to a *hash function*
  - Function that maps elements in domain to integers 0 to size of array minus one (*m*-1)
  - Must take *Θ(1)* time
- **Example hash function**
  - $f : \mathbb{Z} \to [0, m-1]$
  - $f(x) = x \bmod m$
    - Overly simple
    - Most hash functions use modulus to ensure range
- **Example**
  - Size = 10, hash function: mod 10
  - Insert 3, 15, 27, 82, 96, 100

100		82	3		15	96	27		

Collisions

- What do we do when two values map to the same location?

100		82	3		15	96	27		

25

- Insert 25
- Two ways to resolve collisions
  - Separate chaining
  - Open addressing

100		82	3		25	96	27		

15

- Separate chaining
  - Each index in array is the head of a linked list
  - E.g., `node_t* arr;`
- On collision:
  - Node containing new value becomes new head
- To search:
  - Start at index given by hash function
  - Scan list until you find target (or end of list)

Open addressing

- Alternative to separate chaining
- On collision, insert at next available open space
  - Use mod to "wrap around"
- **Example:** insert 25

100		82	3		15	96	27		

25

Open addressing

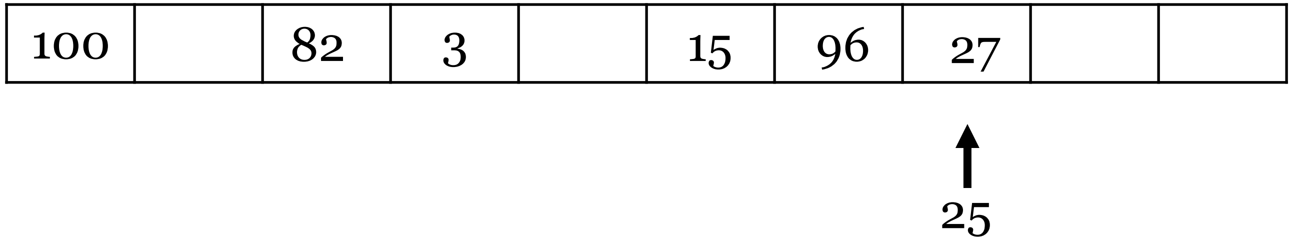
- Alternative to separate chaining
- On collision, insert at next available open space
  - Use mod to "wrap around"
- **Example:** insert 25

100		82	3		15	96	27		

25

Open addressing

- Alternative to separate chaining
- On collision, insert at next available open space
  - Use mod to "wrap around"
- **Example:** insert 25



Open addressing

- Alternative to separate chaining
- On collision, insert at next available open space
  - Use mod to "wrap around"
- **Example:** insert 25

100		82	3		15	96	27	25	

Open addressing

- Alternative to separate chaining
- On collision, insert at next available open space
  - Use mod to "wrap around"
- **Example:** insert 25

100		82	3		15	96	27	25	

- **To search:**
  - Start at hash function index
  - If target not there, scan until you locate element *or* find empty space
- **To delete:** mark element "not present"
- **Congestion**
  - Challenge with open addressing
  - If too many consecutive entries are full, hash table performance degrades
    - More insertions in congested region aggravate problem

Probing strategies

- How to locate "next" open space
  - All use mod to ensure valid range
- **Linear probing**
  - Consider h(x), h(x) + 1, h(x) + 2, h(x) + 3, …
  - Easy
  - Vulnerable to *congestion*
- **Quadratic probing**
  - h(x), h(x) + 1, h(x) + 4, h(x) + 9, …
  - Leaves more "gaps"
  - Not good when array gets nearly full
- **Double hashing**
  - h(x), h(x) + $h_2(x)$, h(x) + $2h_2(x)$, h(x) + $3h_2(x)$, …
  - Usually use secondary hash function
    - Shouldn't return 0
  - Congestion less likely
  - More complex

Hashing complexity

Separate chaining

- **Search**
  - Hash x
  - Scan list
- **Insert**
  - Hash x
  - Prepend
  - **O(1)**
- **Delete**
  - Hash x
  - Scan list and delete
- Complexity depends on length of linked list
  - Worse case: O(n)

Open addressing

- **Search**
  - Hash x
  - Probe until x or empty space
- **Insert**
  - Hash x
  - Probe until empty space
- **Delete**
  - Hash x
  - Probe until x
  - Mark "not present"
- Complexity depends on probing
  - Worst case: O(n)

Why would anyone use a hash table?

- Bad worst-case complexity but great *expected-case* complexity
- Expected-case assumptions
  - Hash function produces $\Theta(1)$ collisions
    - Each inserted value has $\Theta(1)$ duplicates
  - $m = \Theta(n)$

- Search(x)
  - Hashing and scanning take $\Theta(1)$ time
- Insert(x)
  - Hashing and scanning take $\Theta(1)$ time
- Delete(x)
  - Hashing and scanning take $\Theta(1)$ time
  - Reinsertion takes $\Theta(1)$ time (open addressing)

Rehashing

- As elements get added to hash table, performance gets worse
  - Lists get longer
  - Table gets more clogged
- Rehashing
  - Allocating larger table
    - Double capacity
  - Reinsert everything in old table into new one
  - Doesn't increase average complexity
- Rehashing triggered by *load factor*
  - `size / capacity` ratio
  - Rehash at some threshold
- Open addressing does *very* poorly with high load factor
  - Definitely ≤ 0.5
- Separate chaining tolerates higher load factor
  - Between 0.5 and 1

Hash functions

- Good HT performance depends on having good hash function
- Many hash functions use mod at end
  - Ensures range 0..capacity-1
- Good hash functions should:
  - Be fast to compute
  - Produce every index equally often
  - Spread nearby values apart
    - Especially for open addressing
    - Reduces chance of congestion
- **Example:**
  - Capacity = 10; h(x) = 3x % 10
  - Open addressing + linear probing
  - Insert 1, 1, 2, 3, 3

3			1	1		2			3
0	1	2	3	4	5	6	7	8	9

  - Much worse if h(x) = x % 10

Linear hash functions

- Linear functions can be dangerous
- **Example:**
  - Capacity = 10
  - h(x) = 4x % 10
  - Insert 1-10:

5		8		6		9		7	
0		3		1		4		2	
0	1	2	3	4	5	6	7	8	9

- Important that multiplier and capacity not have common factors
  - Prime capacity addresses this issue
- *Alternative:* rotations
  - Bitwise operation
  - Rotate 8 bits left by 3: (x << 3) | (x >> 5)

```
01001101   x = 77

01101000   x << 3
00000010   x >> 5
__________
01101010   rot(x, 3) = 106
```

Multi-byte inputs

- Hash functions may also need to process strings, arrays, or objects
- Treat input as string of bytes
  - Combine bytes (or ints) together
  - Hash resulting int
- **Example**

Input: str: string to be hashed
Input: n: length of str
Input: m: capacity of hash table
Output: hash index for str in the range $[0, m)$
Algorithm: BasicStringHash

$idx = 0$
for $i = 1$ to n **do**
    $idx = idx + str[i]$
end
return $idx \bmod m$

- Addition and bitwise xor commonly used to combine

Permutations

- Potential issue for multi-byte hash functions
  - opts, post, pots, stop, tops
  - Affects strings and arrays
- Clever multi-byte hash functions use progressive hashing
  - Multiply or rotate previous hash by a constant
  - Add or xor next char or int
- **Example**

Input: *str*: string to be hashed
Input: n: length of *str*
Input: m: capacity of hash table
Output: hash index for *str* in the range $[0, m)$

Algorithm: BetterStringHash
$idx = 0$
for $i = 1$ to n **do**
  $idx = 57 \cdot idx + str[i]$
end
return $idx \bmod m$

<u>Multiplying by 2 and adding</u>

1	0	2

1 -> 2 + 0 = 2 -> 4 + 2 = 6

1	1	0

1 -> 2 + 1 = 3 -> 6 + 0 = 6

- Small multipliers can still produce collisions
- Overflow not an issue; just "wraps around"

Sets or dictionaries

- Abstract data type for storing and retrieving values
  - Multiple valid implementations

- **Primary operations**
  - *Search(x)*: returns the location of *x* in the set, or NIL if not contained
  - *Insert(x)*: adds *x* to the set
  - *Delete(x)*: removes *x* from the set

- **Additional operations**
  - *Build(data):* construct set from unsorted array
  - *Max(), Min()*: return the location of the largest/smallest element
  - *Successor(x), Predecessor(x)*: return the next largest/smallest element than *x*

Set complexity

Operation	Balanced BST (worst case)	Hash table (expected)	Separate chaining (worst case)
Search(x)	$O(\lg n)$	$\Theta(1)$	$O(n)$
Delete(x)	$O(\lg n)$	$\Theta(1)$	$O(n)$
Insert(x)	$O(\lg n)$	$\Theta(1)$*	$\Theta(1)$*

- Open addressing: same as separate chaining except that worst case insertion is $O(n)$
- Hash table expected complexity better than balanced BST
  - Worst case is <u>much</u> worse
- Hash tables better "most of the time"
- Balanced BST better if you need to access values in sorted order
- Or if worried about worst case
  - Not sure about hash function or input data
  - Lots of duplicates

Maps

- Abstraction of a function
- Main operations
  - **Insert(x, y):** declares that $f(x) = y$
  - **Delete(x):** declares that $f(x)$ does not have a value
  - **Search(x):** returns y such that $f(x) = y$, or NIL if $f(x)$ does not have a value
- **Example:** letter frequencies
  - Problem: count how many times a letter appears in a given text
    - Used in cryptography
  - Sample output

E	T	A	O	I	N	S	R	H	…
12	9	8	7	7	6	6	6	6	…

  - Need to associate count with letter
  - f(E) = 12, etc.

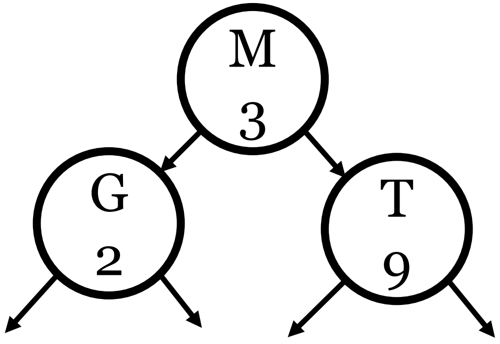
Map implementations

- Two main implementations
- Array-based map
  - Array of all possible *x* values
  - Stores *f*(*x*) in arr[*x*] (NIL if not initialized)

8	2	3	4	12	2	2	…
A	B	C	D	E	F	G	…

  - All main operations are constant time
  - Only useful when input domain is small

- Set-based map
  - A.k.a., hash map
  - Set of ordered (x, y) pairs
  - Pairs added/searched according to *x* value
  - **Search** returns associated *y* value
  - Time complexity determined by hash table or BBST



Map complexity

Operation	Array-based map	Set-based (hash table)	Set-based (BBST)
Add(x, y)	$\Theta(1)$	$\Theta(1)$*	$O(lg\ n)$
Delete(x)	$\Theta(1)$	$\Theta(1)$*	$O(lg\ n)$
Insert(x)	$\Theta(1)$	$\Theta(1)$*	$O(lg\ n)$
Build()	$\Theta(D)$	$\Theta(n)$	$O(n\ lg\ n)$

D: size of domain (*x* values)
* Expected complexity for hash table

The power of maps

- Maps are very useful for storing values that we compute repeatedly
  - Especially when we can use direct maps
- **Example:** Discrete Fourier Transform
  - Given array *x* compute transformed array *c* such that

$$c_k = \sum_{j=1}^{n} x_j \boxed{e}^{jk(-2\pi i/n)}$$

Store values in lookup table

- Can also improve best-case performance for <u>any</u> algorithm

1. Build a map that contains problem instances and solutions
2. Before running another algorithm, test whether input is in map
3. If so, return the answer

- Best case typically constant or linear time
- Best case analysis not useful to compare algorithm quality

Coming up

- Hash tables
- Sets and maps
- Priority queues
- Heaps
- Union-Find

- **Recommended readings:** Sections 4.1, 4.2, 6.1, 6.2, and 6.3
  - Consider reading section 6.4
  - We won't cover cuckoo hashing, but it is a powerful idea
  - *Practice problems:* R-4.1, R-4.6, C-4.5, C-4.7, A-4.1, A-4.2, R-6.1, R-6.2, R-6.4 to R-6.7, C-6.6, A-6.1, A-6.6