CSE 326: Data Structures

James Fogarty
Autumn 2007
Lecture 13

Logistics

- Closed Notes
- Closed Book
- Open Mind
- Four Function Calculator Allowed

Material Covered

- Everything we’ve talked/read in class up to and including B-trees

Material Not Covered

- We won’t make you write syntactically correct Java code (pseudocode okay)
- We won’t make you do a super hard proof
- We won’t test you on the details of generics, interfaces, etc. in Java
  - But you should know the basic ideas

Terminology

- Abstract Data Type (ADT)
  - Mathematical description of an object with set of operations on the object. Useful building block.
- Algorithm
  - A high level, language independent, description of a step-by-step process
- Data structure
  - A specific family of algorithms for implementing an abstract data type.
- Implementation of data structure
  - A specific implementation in a specific language

Algorithms vs Programs

- Proving correctness of an algorithm is very important
  - a well designed algorithm is guaranteed to work correctly and its performance can be estimated

- Proving correctness of a program (an implementation) is fraught with weird bugs
  - Abstract Data Types are a way to bridge the gap between mathematical algorithms and programs

First Example: Queue ADT

- FIFO: First In First Out
- Queue operations

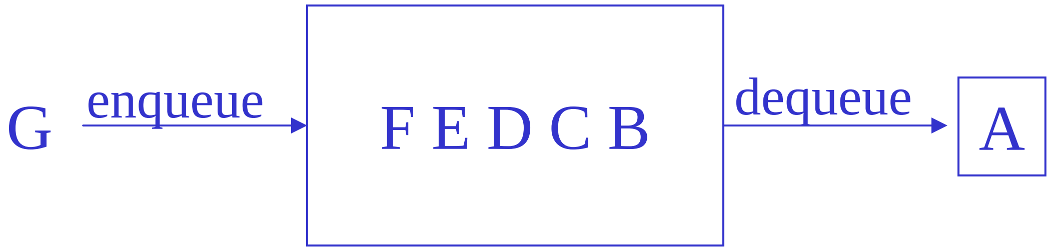
  create

  destroy

  enqueue

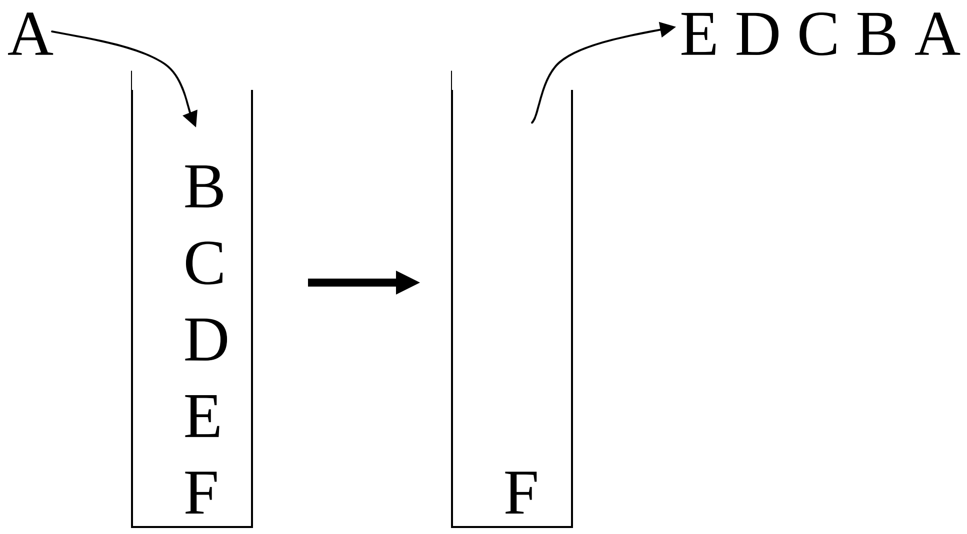
  dequeue

  is_empty



Second Example: Stack ADT

- LIFO: Last In First Out
- Stack operations
  - create
  - destroy
  - push
  - pop
  - top
  - is_empty



Priority Queue ADT

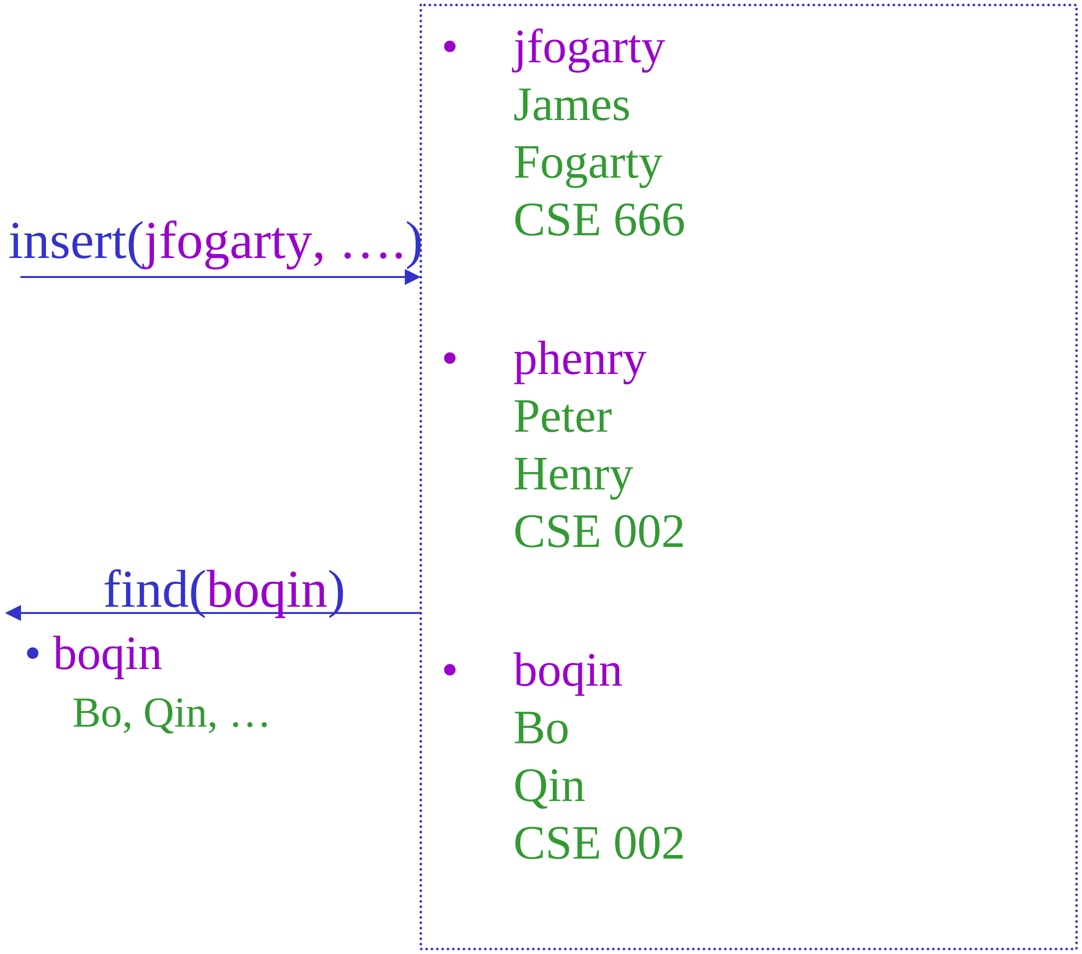
1. **PQueue data** : collection of data with priority

2. **PQueue operations**
   - insert
   - deleteMin

3. **PQueue property**: for two elements in the queue, *x* and *y*, if *x* has a **lower** priority value than *y*, *x* will be deleted before *y*

The Dictionary ADT

- Data:
  - a set of (key, value) pairs
- Operations:
  - Insert (key, value)
  - Find (key)
  - Remove (key)



insert(jfogarty, ….)

- jfogarty
  James
  Fogarty
  CSE 666
- phenry
  Peter
  Henry
  CSE 002
- boqin
  Bo
  Qin
  CSE 002

find(boqin)

- boqin
  Bo, Qin, …

*The Dictionary ADT is also called the "**Map ADT**"*

Proof by Induction

- **Basis Step:** The algorithm is correct for a base case or two by inspection.
- **Inductive Hypothesis (n=k):** Assume that the algorithm works correctly for the first k cases.
- **Inductive Step (n=k+1):** Given the hypothesis above, show that the k+1 case will be calculated correctly.

Recursive algorithm for *sum*

- Write a *recursive* function to find the sum of the first **n** integers stored in array **v**.

```
sum(integer array v, integer n) returns integer
  if n = 0 then
    sum = 0
  else
    sum = nth number + sum of first n-1 numbers
  return sum
```

Program Correctness by Induction

- **Basis Step:**
  `sum(v,0) = 0`. ✓
- **Inductive Hypothesis (n=k):**
  Assume **`sum(v,k)`** correctly returns sum of first k elements of v, i.e. **`v[0]+v[1]+…+v[k-1]+v[k]`**
- **Inductive Step (n=k+1):**
  `sum(v,n)` returns
  `v[k]+sum(v,k-1)= (by inductive hyp.)`
  `v[k]+(v[0]+v[1]+…+v[k-1])=`
  `v[0]+v[1]+…+v[k-1]+v[k]` ✓

Solving Recurrence Relations

1. Determine the recurrence relation. What is/are the base case(s)?

2. “Expand” the original relation to find an equivalent general expression *in terms of the number of expansions*.

3. Find a closed-form expression by setting *the number of expansions* to a value which reduces the problem to a base case

Asymptotic Analysis

- Asymptotic analysis looks at the *order* of the running time of the algorithm
    - A valuable tool when the input gets "large"
    - Ignores the *effects of different machines* or *different implementations* of the same algorithm
    - Intuitively, to find the asymptotic runtime, throw away constants and low-order terms
    - Linear search is $T(n) = 3n + 2 \in \mathbf{O(n)}$
    - Binary search is $T(n) = 4 \log_2 n + 4 \in \mathbf{O(\log n)}$

Meet the Family

- O(f(*n*)) is the set of all functions asymptotically less than or equal to f(*n*)
  - o(f(*n*)) is the set of all functions asymptotically strictly less than f(*n*)
- Ω(f(*n*)) is the set of all functions asymptotically greater than or equal to f(*n*)
  - ω(f(*n*)) is the set of all functions asymptotically strictly greater than f(*n*)
- θ(f(*n*)) is the set of all functions asymptotically equal to f(*n*)

Definition of Order Notation

- Upper bound: $T(n) = O(f(n))$ Big-O

  Exist positive constants *c* and *n'* such that

  $T(n) \leq c\, f(n)$ for all $n \geq n'$

- Lower bound: $T(n) = \Omega(g(n))$ Omega

  Exist positive constants *c* and *n'* such that

  $T(n) \geq c\, g(n)$ for all $n \geq n'$

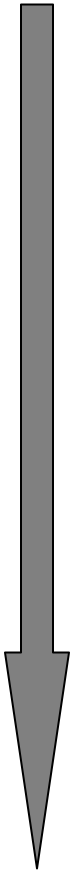
- Tight bound: $T(n) = \theta(f(n))$ Theta

  When both hold:

  $T(n) = O(f(n))$

  $T(n) = \Omega(f(n))$

Big-O: Common Names

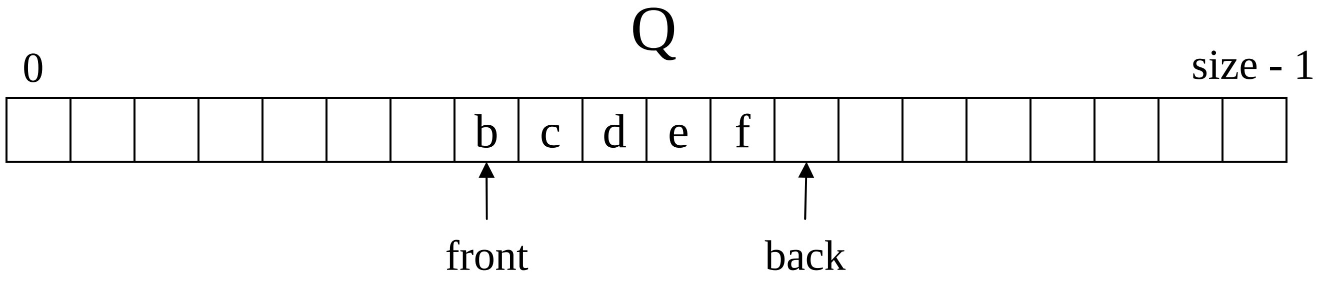


- constant: $O(1)$
- logarithmic: $O(\log n)$ ($\log_k n$, $\log n^2 \in O(\log n)$)
- linear: $O(n)$
- log-linear: $O(n \log n)$
- quadratic: $O(n^2)$
- cubic: $O(n^3)$
- polynomial: $O(n^k)$ (k is a constant)
- exponential: $O(c^n)$ (c is a constant > 1)

Perspective: Kinds of Analysis

- Running time may depend on actual data input, not just length of input
- Distinguish
  - Worst Case
    - Your worst enemy is choosing input
  - Best Case
  - Average Case
    - Assumes some probabilistic distribution of inputs
  - Amortized
    - Average time over many operations

Circular Array Queue Data Structure



```
enqueue(Object x) {
   Q[back] = x ;
   back = (back + 1) % size
   }

dequeue() {
   x = Q[front] ;
   front = (front + 1) % size;
   return x ;
   }
```

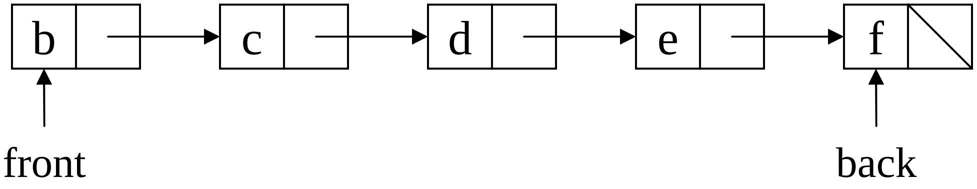
How test for empty list?

How to find K-th element in the queue?

What is complexity of these operations?

Limitations of this structure?

Linked List Queue Data Structure

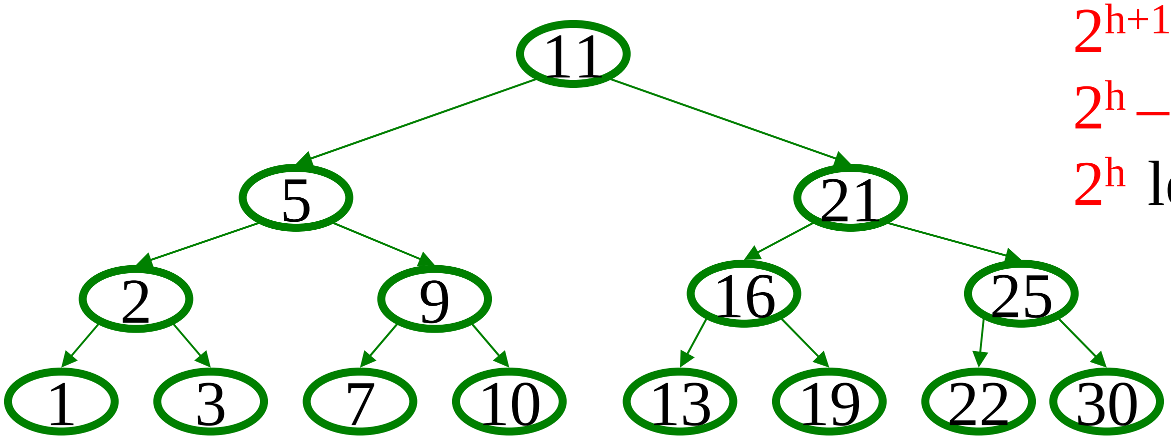


```
void enqueue(Object x) {
   if (is_empty())
        front = back = new Node(x)
   else
        back->next = new Node(x)
        back = back->next
}
bool is_empty() {
   return front == null
}
```

```
Object dequeue() {
   assert(!is_empty)
   return_data = front->data
   temp = front
   front = front->next
   delete temp
   return return_data
}
```

Brief interlude: Some Definitions:

A *Perfect* binary tree – A binary tree with all leaf nodes at the same depth. All internal nodes have 2 children.



height h

$2^{h+1} - 1$ nodes

$2^h - 1$ non-leaves

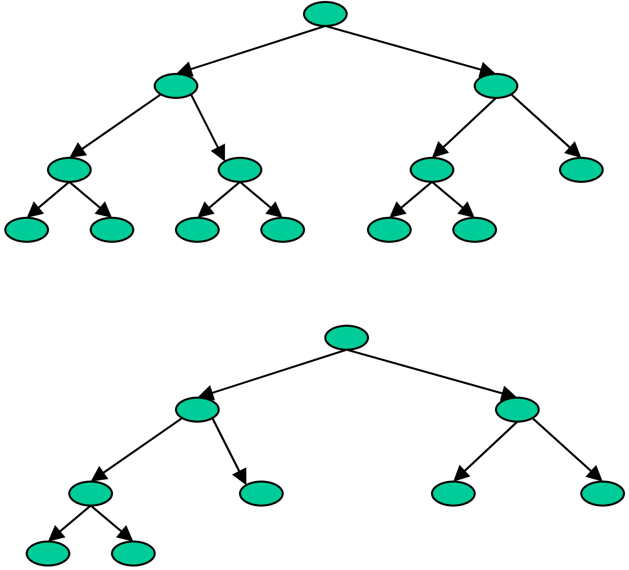
2^h leaves

Heap **Structure** Property

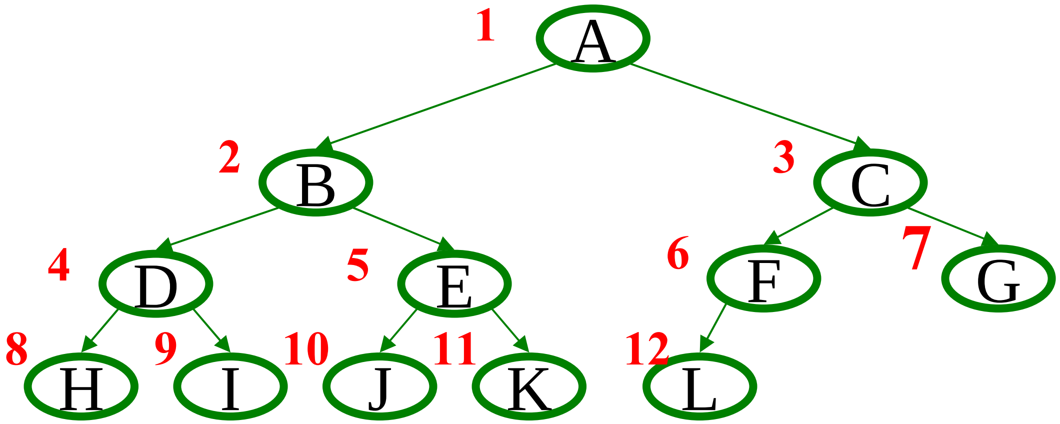
- A binary heap is a ***complete*** binary tree.

Complete binary tree – binary tree that is completely filled, with the possible exception of the bottom level, which is filled left to right.

Examples:



Representing Complete Binary Trees in an Array



From node **i**:

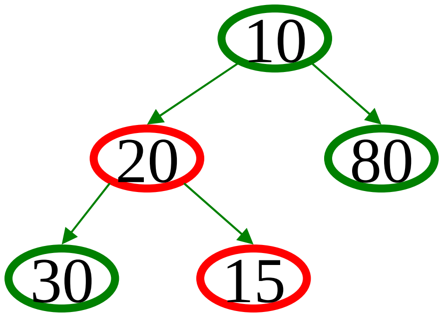
left child:
right child:
parent:

implicit (array) implementation:

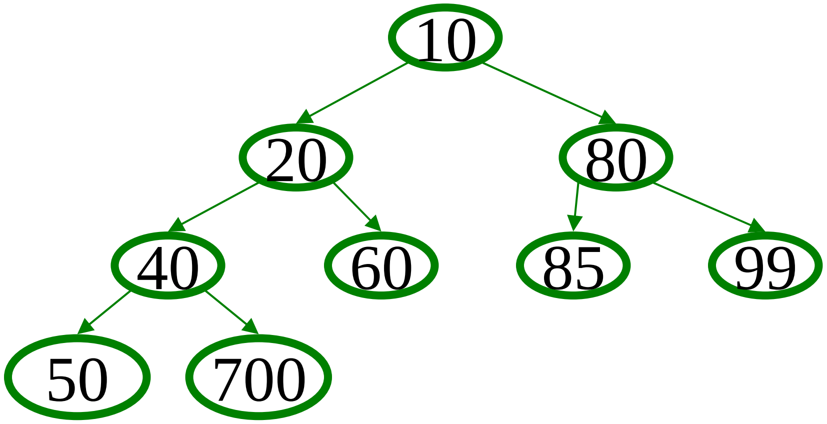
	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Heap **Order** Property

Heap order property: For every non-root node X, the value in the parent of X is less than (or equal to) the value in X.

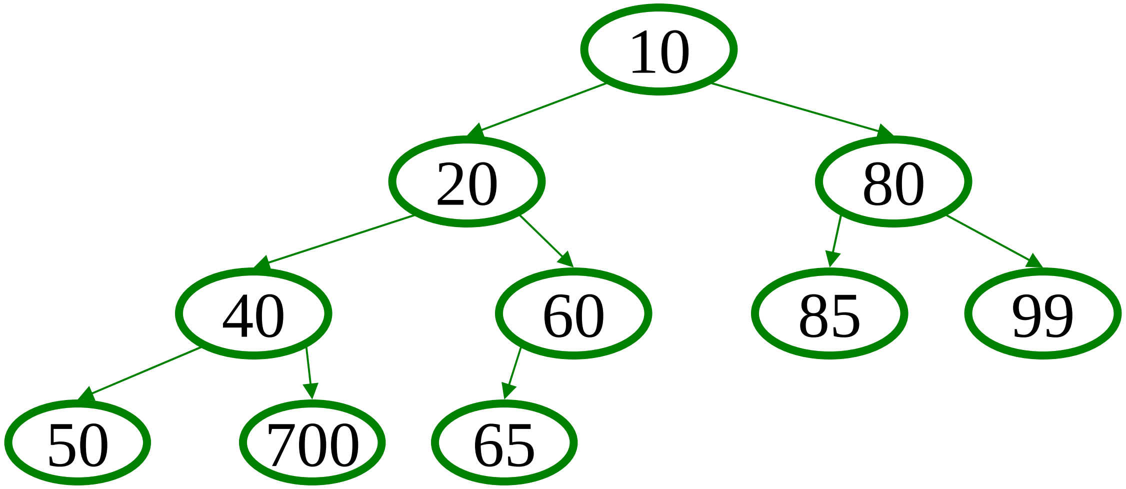


not a heap



Heap Operations

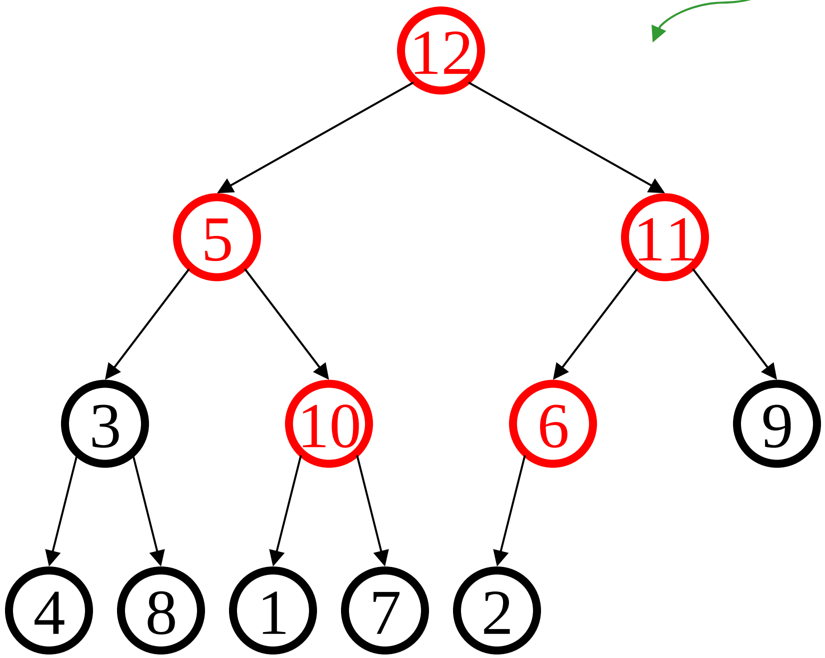
- findMin:
- insert(val): percolate up.
- deleteMin: percolate down.

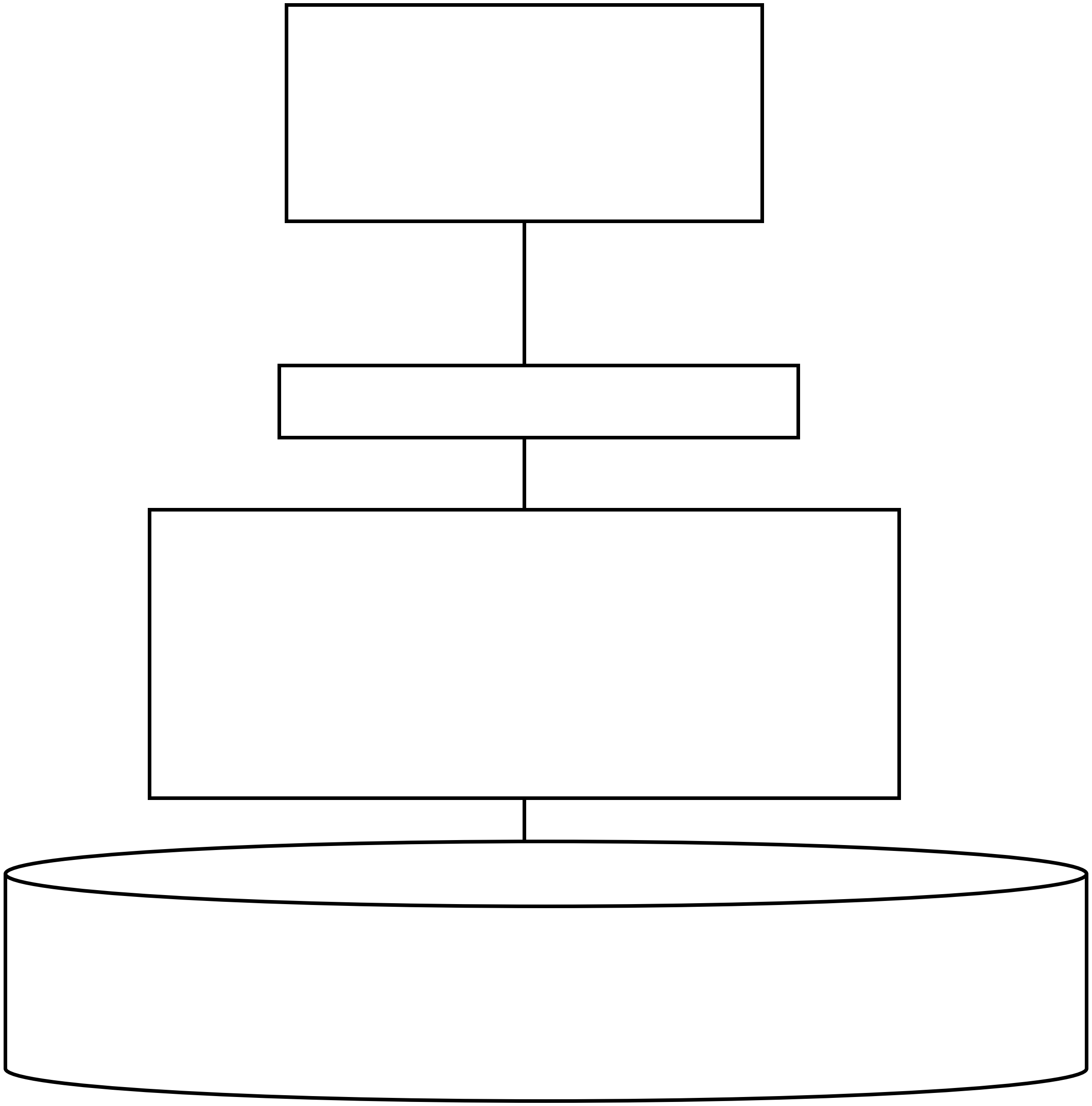


BuildHeap: Floyd’s Method

12	5	11	3	10	6	9	4	8	1	7	2

Add elements arbitrarily to form a complete tree.
Pretend it’s a heap and fix the heap-order property!





Cycles to access:

CPU

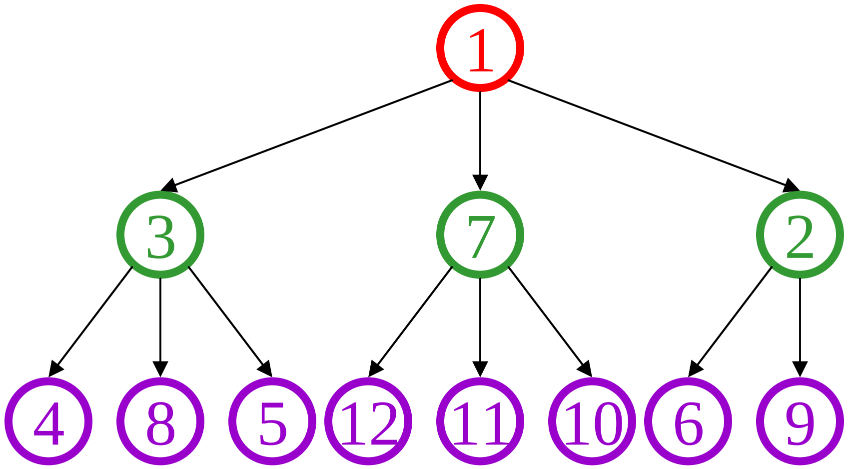
Cache

Memory

Disk

A Solution: *d*-Heaps

- Each node has *d* children
- Still representable by array
- Good choices for *d*:
  - (choose a power of two for efficiency)
  - fit one set of children in a cache line
  - fit one set of children on a memory page/disk block



12	1	3	7	2	4	8	5	12	11	10	6	9

New Heap Operation: Merge

Given two heaps, merge them into one heap

- first attempt: insert each element of the smaller heap into the larger.

  runtime:

- second attempt: concatenate binary heaps' arrays and run buildHeap.

  runtime:

Leftist Heaps

Idea:

Focus all heap maintenance work in one small part of the heap

Leftist heaps:

1. Most nodes are on the left
2. All the merging work is done on the right

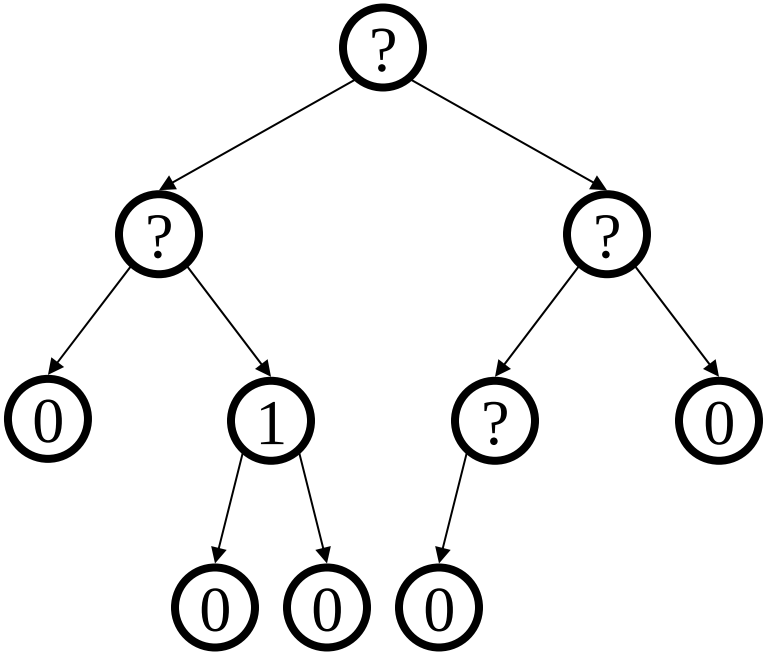
Definition: Null Path Length

null path length (npl) of a node *x* = the number of nodes between *x* and a null in its subtree

OR

npl(x) = min distance to a descendant with 0 or 1 children

- *npl*(null) = -1
- *npl*(leaf) = 0
- *npl*(single-child node) = 0



Equivalent definitions:

1. *npl(x)* is the height of largest complete subtree rooted at *x*
2. *npl(x)* = 1 + min{*npl*(left(x)), *npl*(right(x))}

Leftist Heap Properties

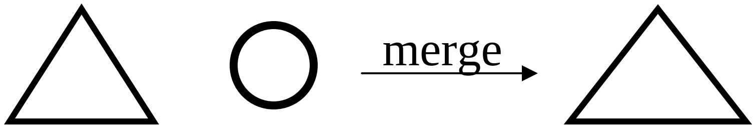
- Heap-order property
  - parent's priority value is $\leq$ to childrens' priority values
  - result: minimum element is at the root

- Leftist property
  - For every node *x*, *npl*(left(*x*)) $\geq$ *npl*(right(*x*))
  - result: tree is at least as "heavy" on the left as the right

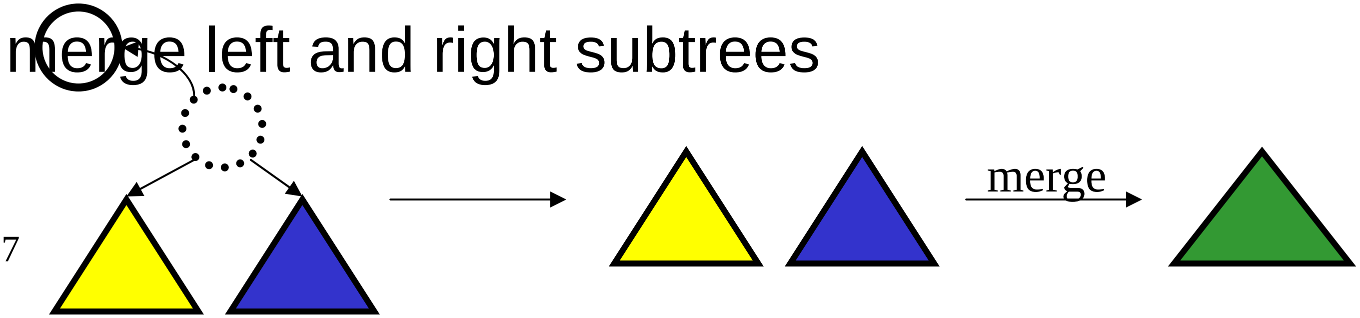
Are leftist trees…
complete?
balanced?

Operations on Leftist Heaps

- merge with two trees of total size n: O(log n)
- insert with heap size n: O(log n)
  - pretend node is a size 1 leftist heap
  - insert by merging original heap with one node heap



- deleteMin with heap size n: O(log n)
  - remove and return root
  - merge left and right subtrees



Skew Heaps

Problems with <u>leftist</u> heaps

- extra storage for npl
- extra complexity/logic to maintain and check npl
- right side is “often” heavy and requires a switch

Solution: <u>skew</u> heaps

- “blindly” adjusting version of leftist heaps
- merge *always* switches children when fixing right path
- <u>amortized time</u> for: merge, insert, deleteMin = O(log *n*)
- however, <u>worst case time</u> for all three = O(*n*)

Runtime Analysis: Worst-case and Amortized

- No worst case guarantee on right path length!
- All operations rely on merge

$\Rightarrow$ worst case complexity of all ops =

- Probably won't get to amortized analysis in this course, but see Chapter 11 if curious.
- Result: *M* merges take time *M* log *n*

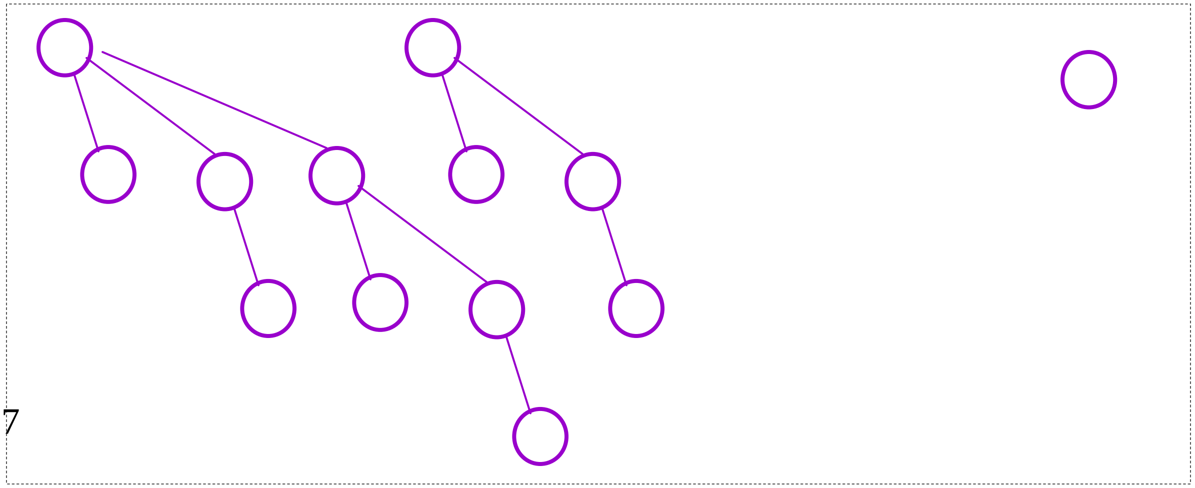
$\Rightarrow$ amortized complexity of all ops =

Binomial Queue with *n* elements

Binomial Q with *n* elements has a *unique* structural representation in terms of binomial trees!

Write *n* in binary: $n = 1101_{\text{(base 2)}} = 13_{\text{(base 10)}}$

1 B_3 1 B_2 No B_1 1 B_0



Properties of Binomial Queue

- At most one binomial tree of any height

- *n* nodes ⇒ binary representation is of size ?
  ⇒ deepest tree has height ?
  ⇒ number of trees is ?

Define: height(forest F) = $\max_{\text{tree T in F}}$ { height(T) }

Binomial Q with *n* nodes has height Θ(log *n*)

Merging Two Binomial Queues

Essentially like adding two binary numbers!

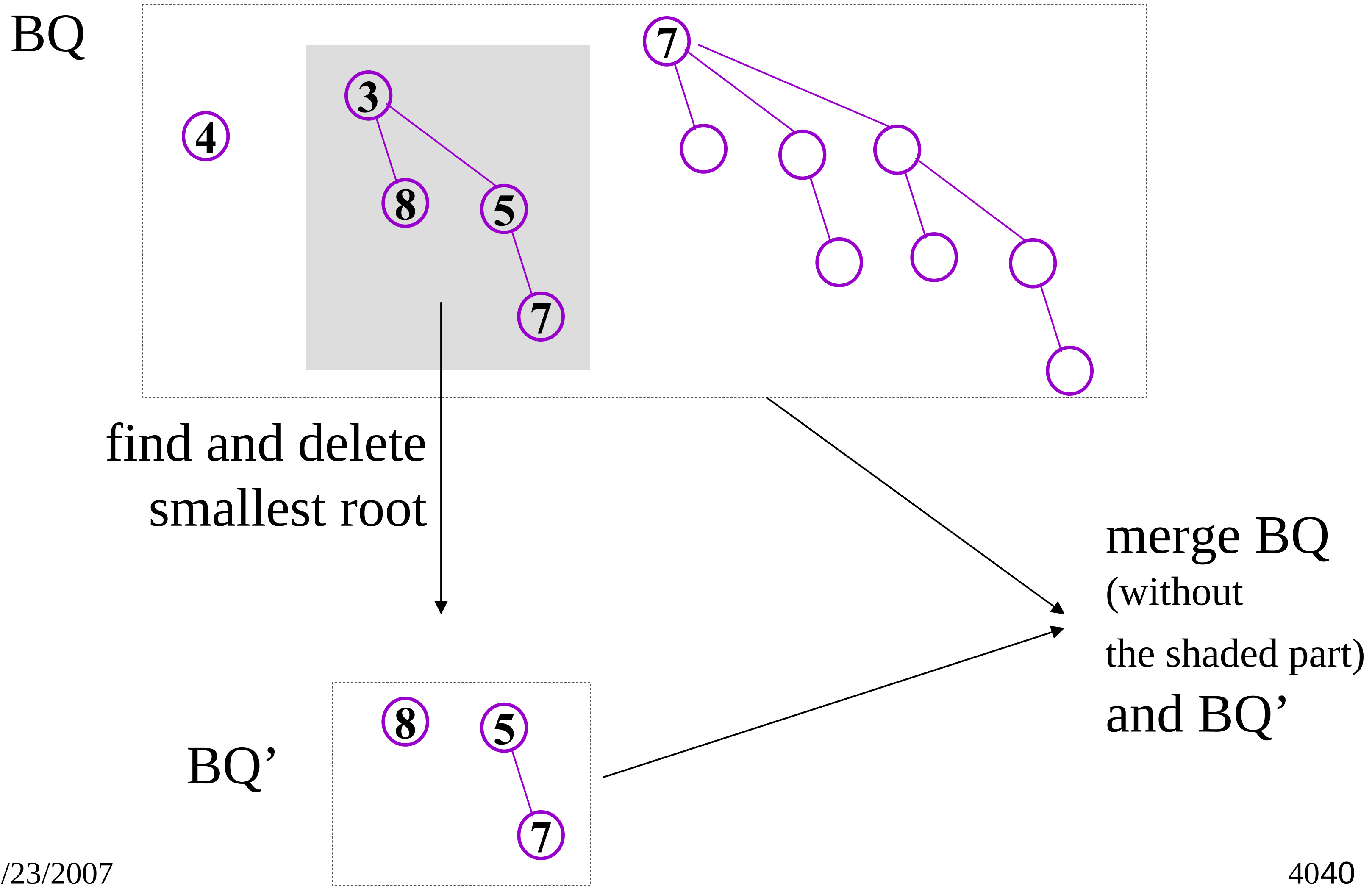
1. Combine the two forests
2. For k from 0 to maxheight {
   a. $m \leftarrow$ total number of B_k's in the two BQs
   b. if m=0: continue;
   c. if m=1: continue;
   d. if m=2: combine the two B_k's to form B_{k+1}
   e. if m=3: retain one B_k and combine the other two to form a B_{k+1}

}

# of 1's
0+0 = 0
1+0 = 1
1+1 = 0+c
1+1+c = 1+c

Claim: When this process ends, the forest has at most one tree of any height

deleteMin: Example



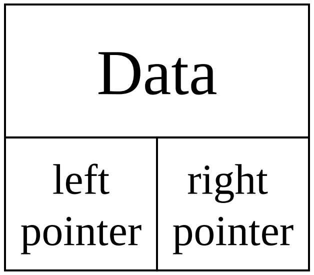
find and delete smallest root

merge BQ (without the shaded part) and BQ’

Binary Trees

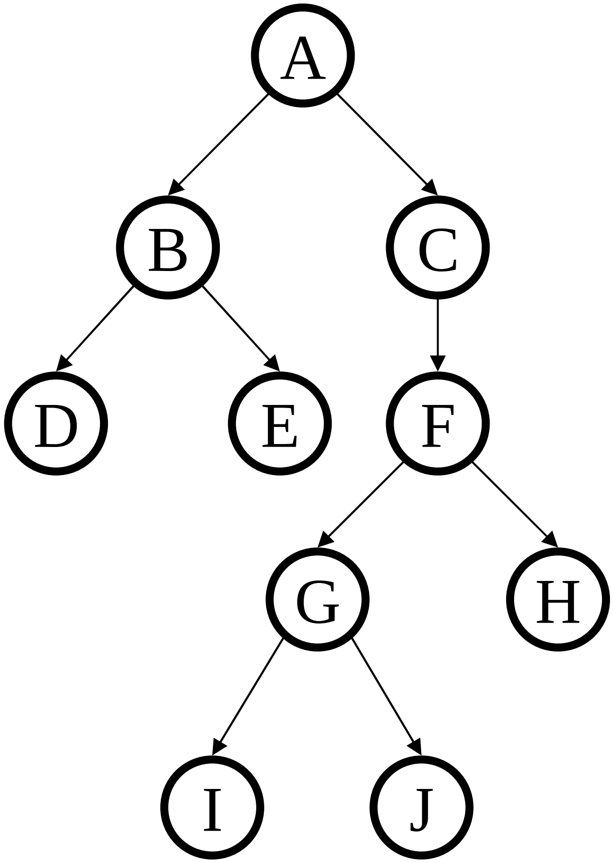
- Binary tree is
  - a root
  - left subtree *(maybe empty)*
  - right subtree *(maybe empty)*

- Representation:

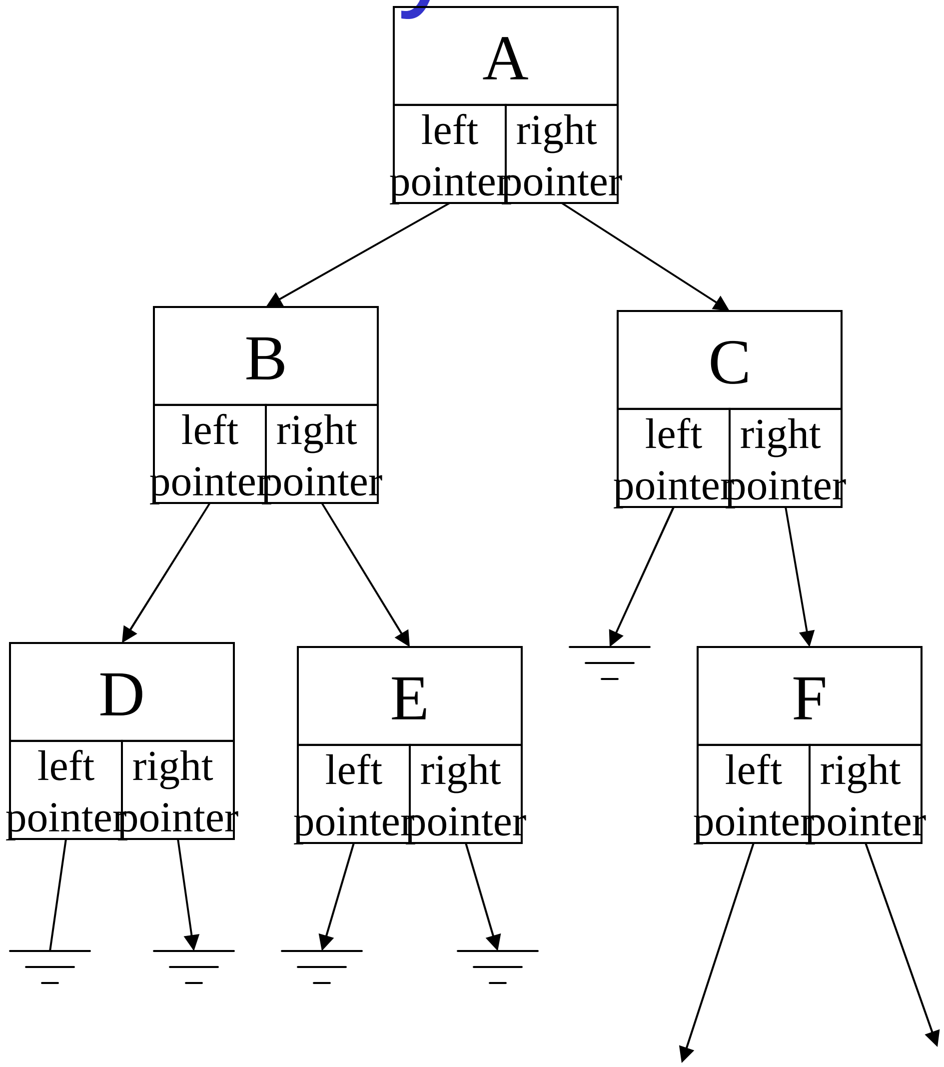


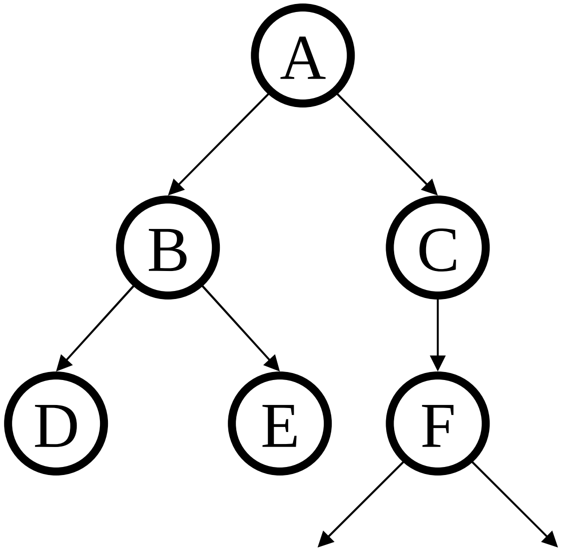
Data

left pointer | right pointer

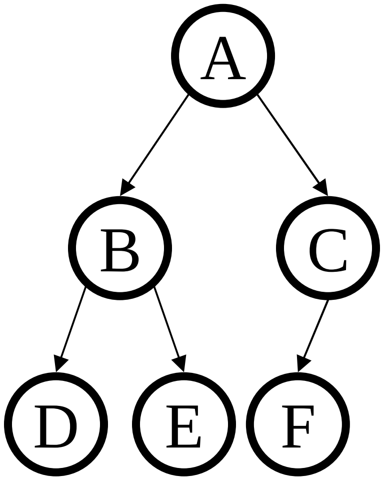


Binary Tree: Representation

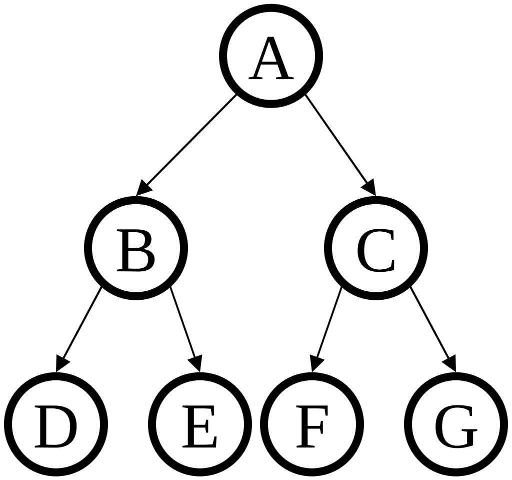




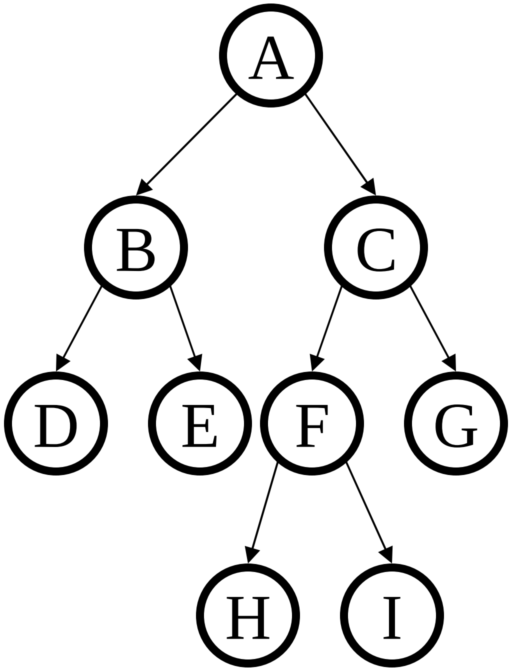
Binary Tree: Special Cases



Complete Tree



Perfect Tree

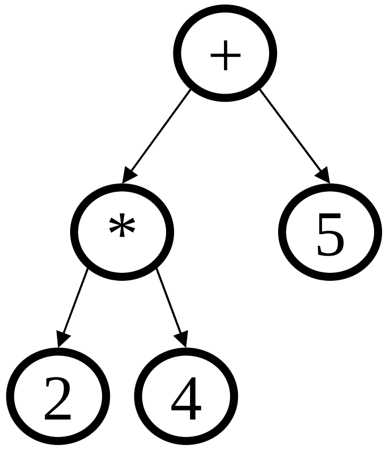


Full Tree

More Recursive Tree Calculations:

Tree Traversals

A *traversal* is an order for visiting all the nodes of a tree



(an expression tree)

Three types:

- Pre-order: Root, left subtree, right subtree
- In-order: Left subtree, root, right subtree

Binary Tree: Some Numbers!

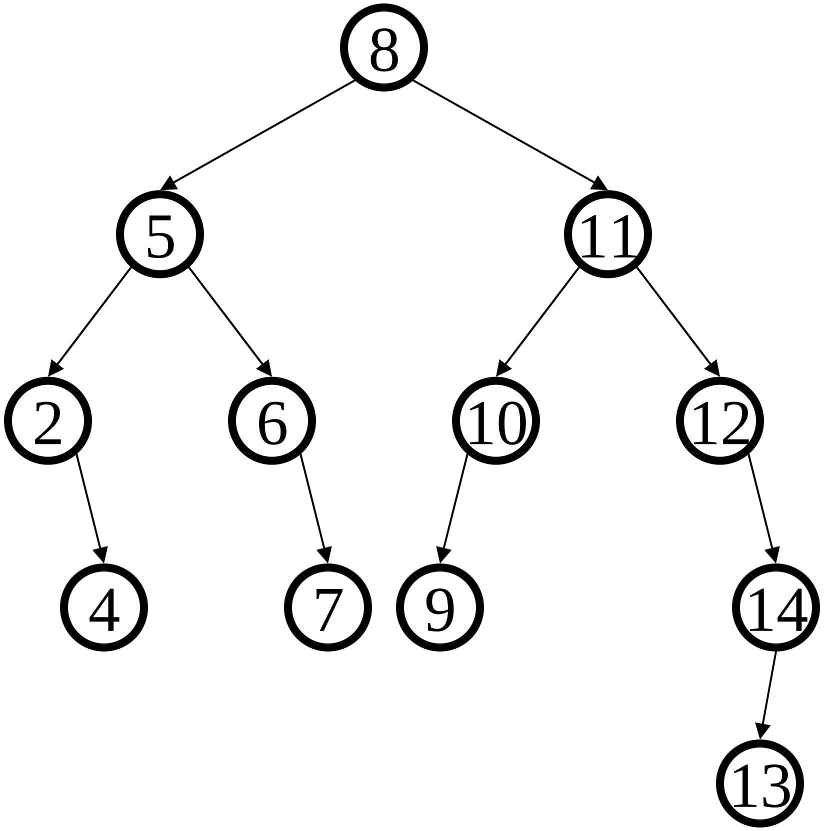
For binary tree of height *h*:

- max # of leaves: 2^h, for perfect tree
- max # of nodes: $2^{h+1} - 1$, for perfect tree
- min # of leaves: 1, for “list” tree
- min # of nodes: $h+1$, for “list” tree

Average Depth for N nodes?

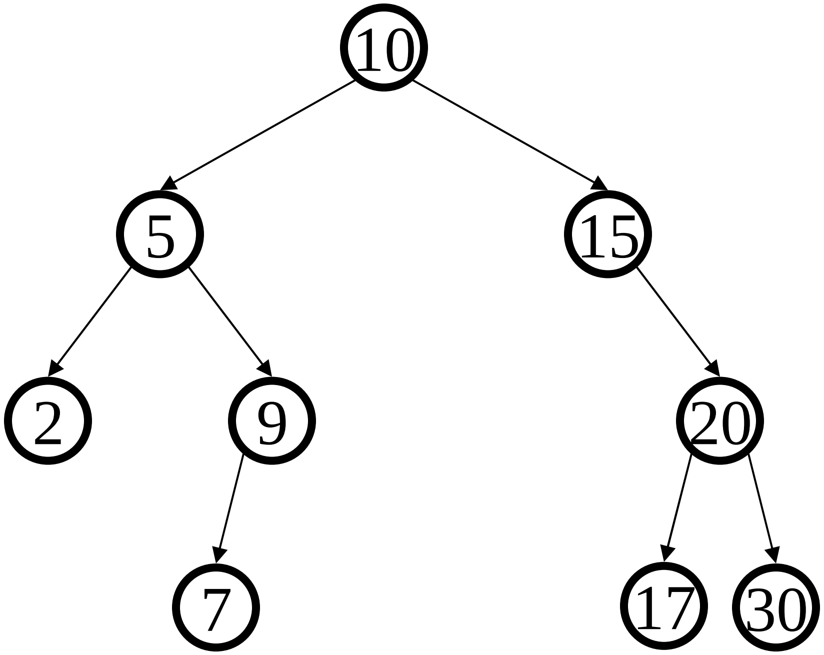
Binary Search Tree Data Structure

- Structural property
  - › each node has ≤ 2 children
  - › result:
    - storage is small
    - operations are simple
    - average depth is small
- Order property
  - › all keys in left subtree smaller than root's key
  - › all keys in right subtree larger than root's key
  - › result: easy to find any given key
- What must I know about what I store?



Comparison, equality testing

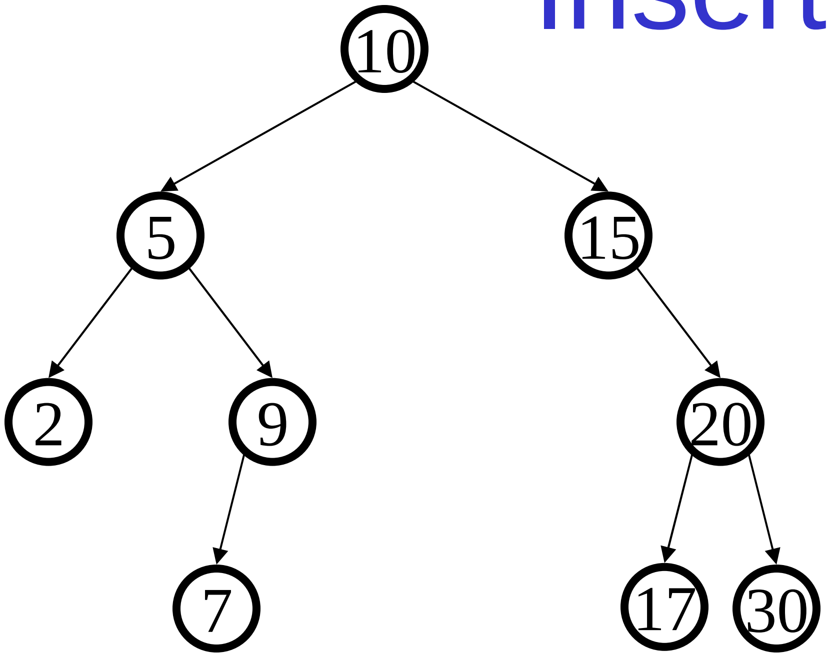
Find in BST, Recursive



```
Node Find(Object key,
            Node root) {
  if (root == NULL)
    return NULL;

  if (key < root.key)
    return Find(key,

  root.left);
  else if (key > root.key)
    return Find(key,

  root.right);
  else
    return root;
}
```

Runtime:

Insert in BST



Insert(13)
Insert(8)
Insert(31)

Insertions happen only at the leaves – easy!

Runtime:

Deletion

- Removing an item disrupts the tree structure.
- Basic idea: find the node that is to be removed. Then “fix” the tree so that it is still a binary search tree.
- Three cases:
  - node has no children (leaf node)
  - node has one child
  - node has two children

Deletion – The Two Child Case

Idea: Replace the deleted node with a value guaranteed to be between the two child subtrees

Options:

- *succ* from right subtree: findMin(t.right)
- *pred* from left subtree : findMax(t.left)

Now delete the original node containing *succ* or *pred*

- Leaf or one child case – easy!

Balanced BST

Observation

- BST: the shallower the better!
- For a BST with *n* nodes
  - › Average height is O(log *n*)
  - › Worst case height is O(*n*)
- Simple cases such as insert(1, 2, 3, ..., n) lead to the worst case scenario

Solution: Require a **Balance Condition** that

1. ensures depth is O(log *n*) – strong enough!
2. is easy to maintain – not too strong!

The AVL Tree Data Structure

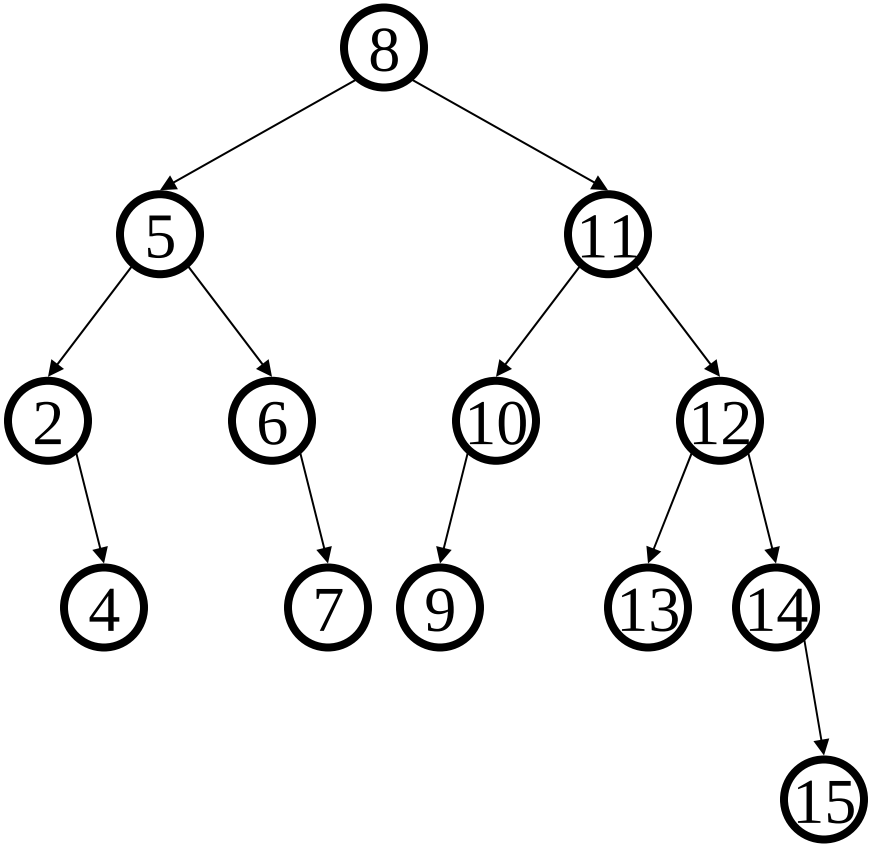
Structural properties

1. Binary tree property (0,1, or 2 children)
2. Heights of left and right subtrees of *every node* **differ by at most 1**

Result:

Worst case depth of any node is: O(log *n*)

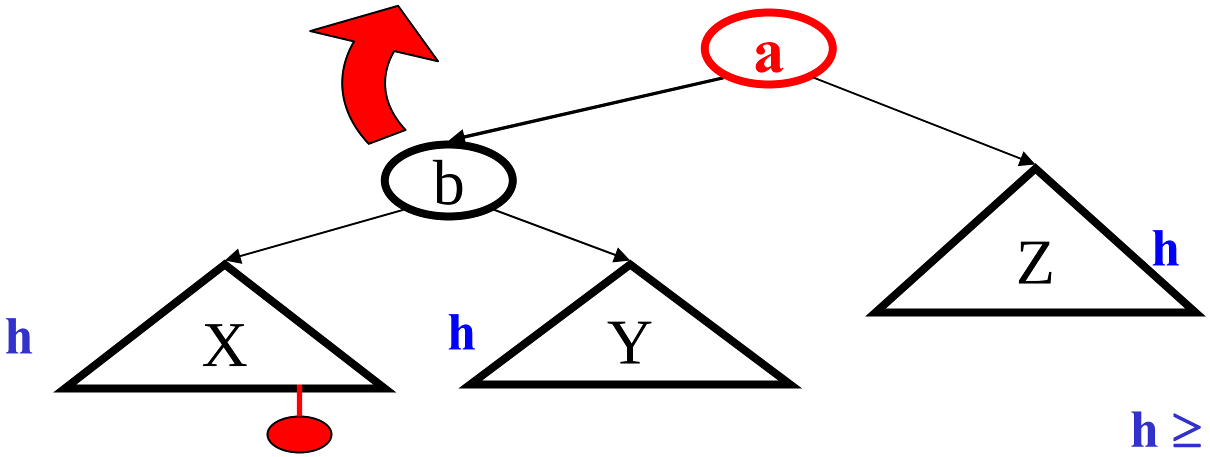
This is an AVL tree



AVL trees: find, insert

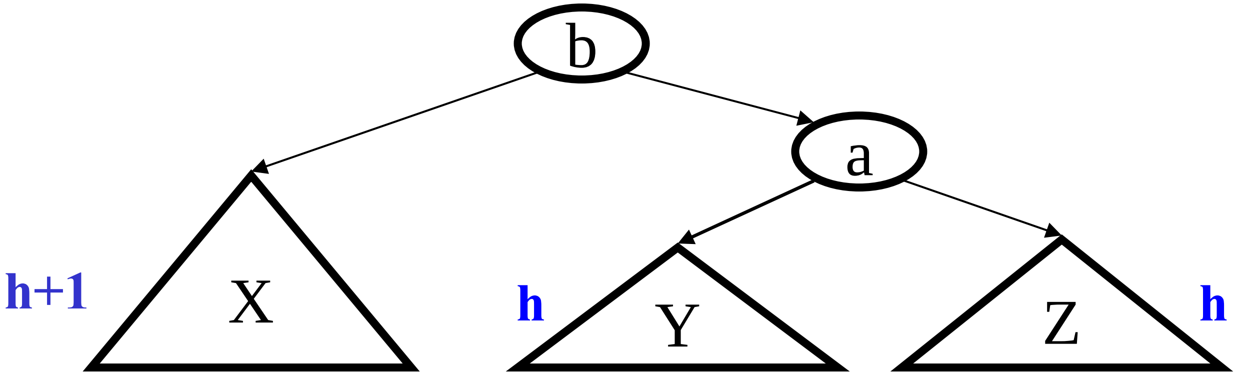
- **AVL find**:
  - same as BST find.
- **AVL insert**:
  - same as BST insert, *except* may need to “fix” the AVL tree after inserting new value.

Single rotation in general



h ≥ -1

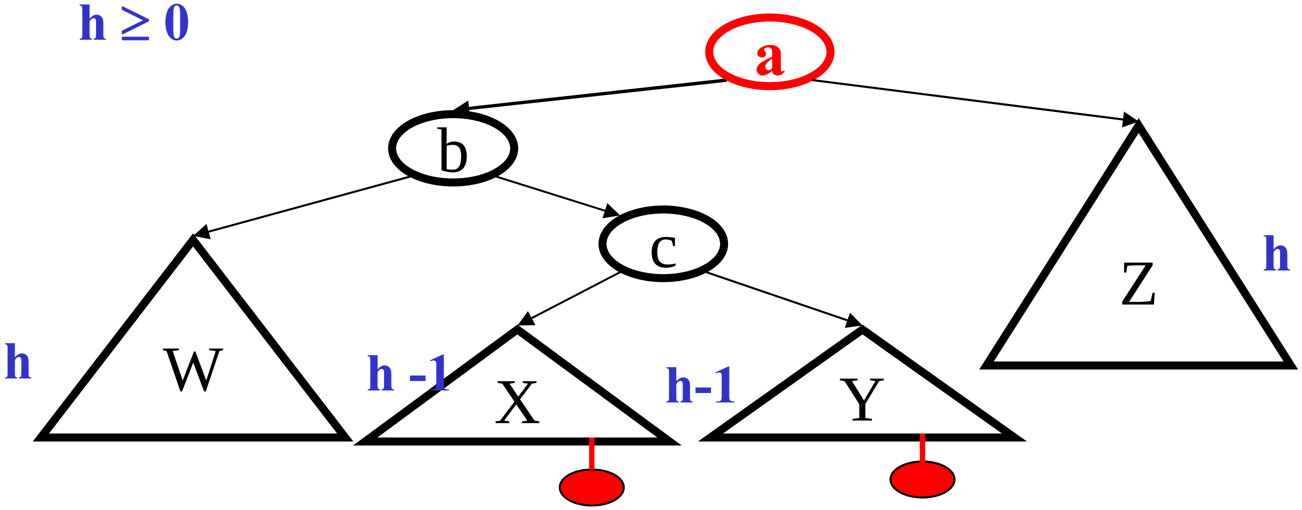
X < b < Y < a < Z



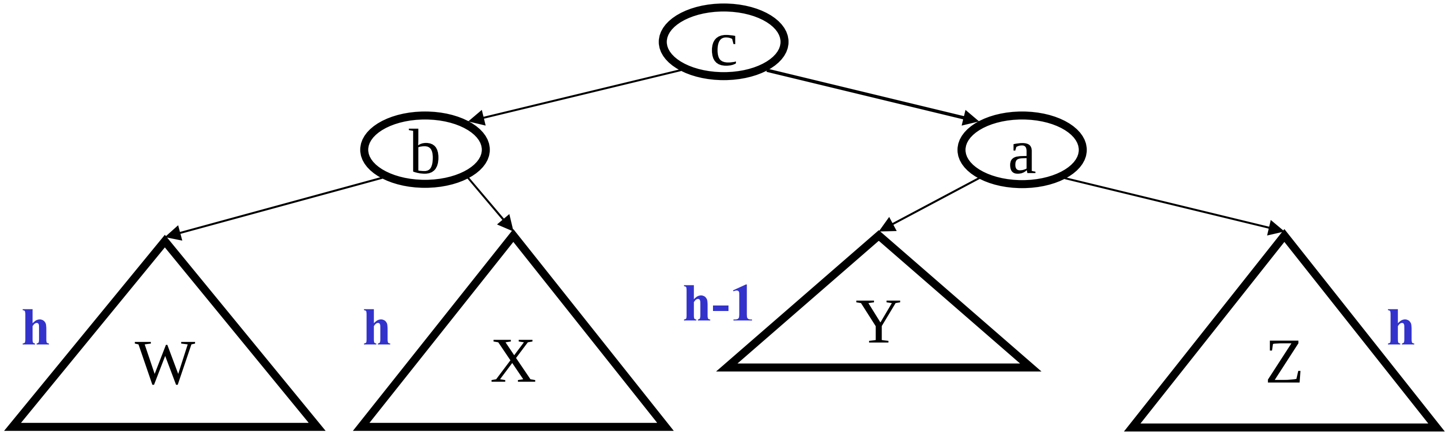
Height of tree before? Height of tree after? Effect on Ancestors?

Double rotation in general

h ≥ 0



W < b <X < c < Y < a < Z

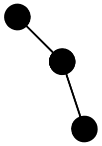


Height of tree before? Height of tree after? Effect on Ancestors?

Insertion into AVL tree

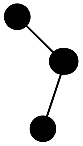
1. Find spot for new key
2. Hang new node there with this key
3. Search back up the path for imbalance
4. If there is an imbalance:

Zig-zig



case #1: Perform single rotation and exit

Zig-zag



case #2: Perform double rotation and exit

Both rotations keep the subtree height unchanged.
Hence only one rotation is sufficient!

Splay Trees

- Blind adjusting version of AVL trees
  - Why worry about balances? Just rotate anyway!
- *Amortized* time per operations is O(log *n*)
- Worst case time per operation is O(*n*)
  - But guaranteed to happen rarely

Insert/Find always rotate node *to the root*!

SAT/GRE Analogy question:

AVL is to Splay trees as ___________ is to __________

Leftish heap : Skew heap

Recall: Amortized Complexity

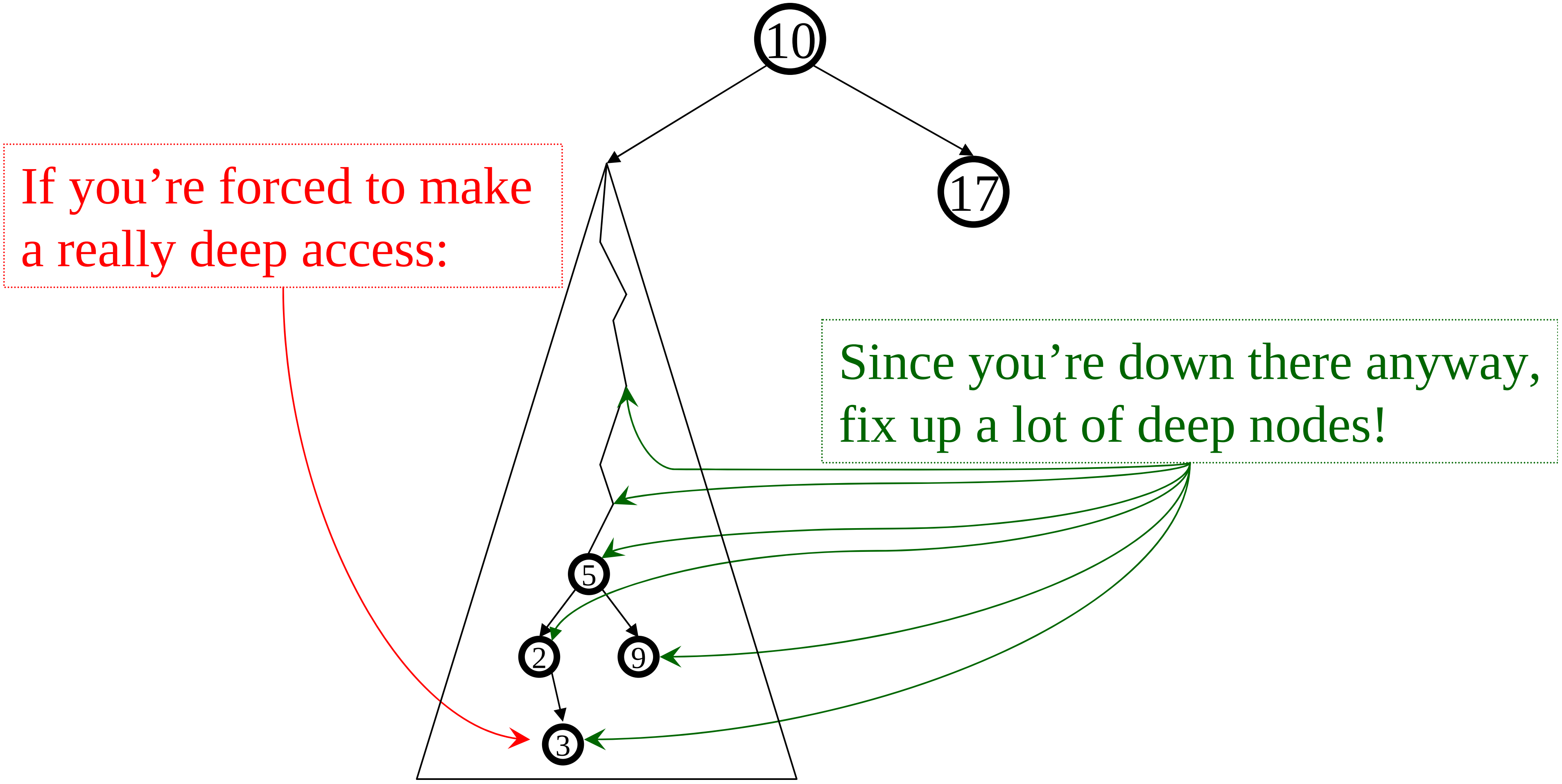
If a sequence of M operations takes O(M f(*n*)) time, we say the amortized runtime is O(f(*n*)).

- Worst case time *per operation* can still be large, say O(*n*)
- Worst case time for *<u>any</u> sequence* of M operations is O(M f(*n*))

Average time *per operation* for *any* sequence is O(f(*n*))

Amortized complexity is *worst-case* guarantee over *sequences* of operations.

The Splay Tree Idea



If you’re forced to make a really deep access:

Since you’re down there anyway, fix up a lot of deep nodes!

Find/Insert in Splay Trees

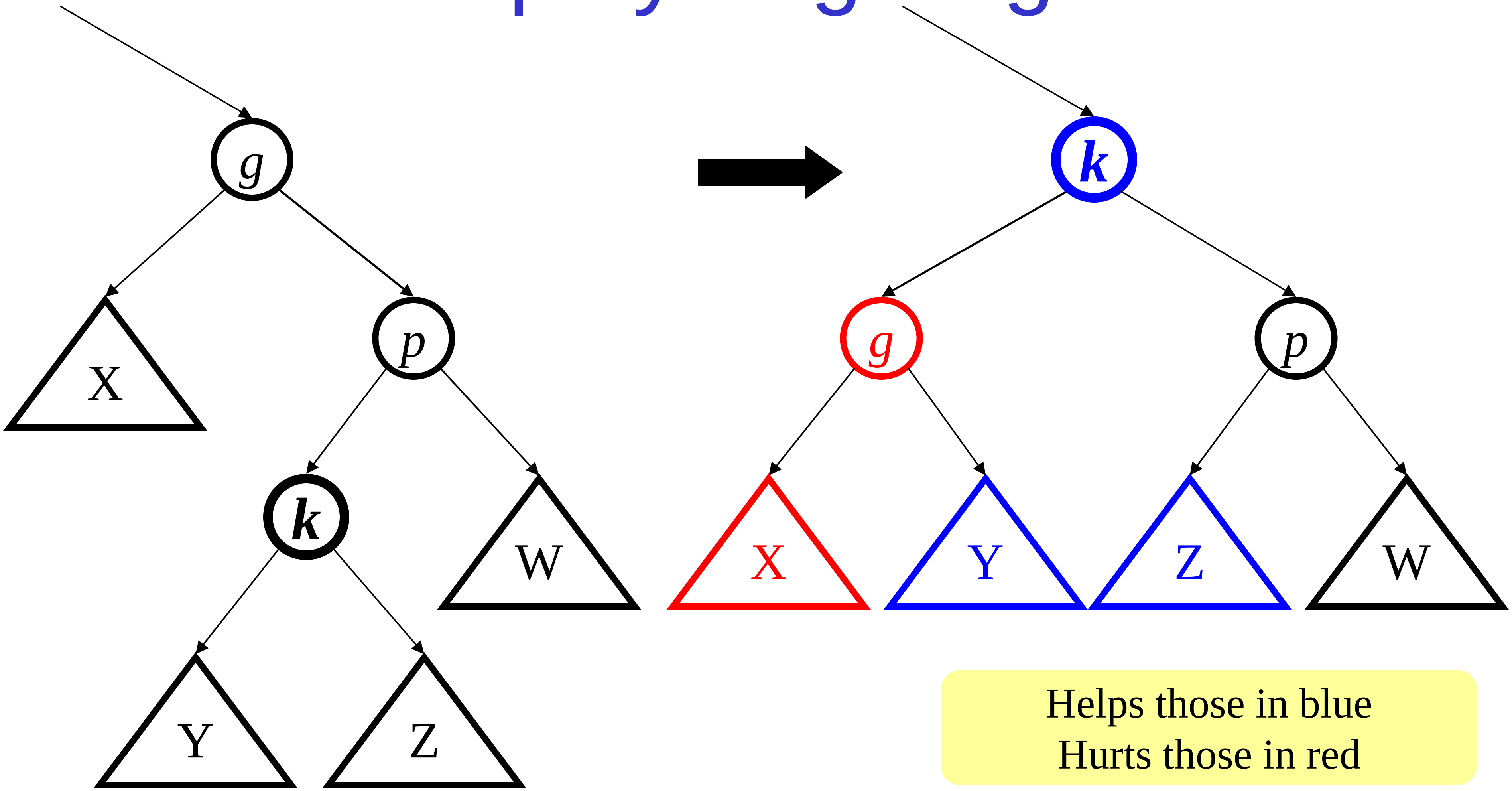
1. Find or insert a node ***k***
2. **Splay *k* to the root using**:

   zig-zag, zig-zig, or plain old zig rotation

Why could this be good??

1. Helps the new root, ***k***
   - *Great if k is accessed again*
2. And helps many others!
   - *Great if many others on the path are accessed*

Splay: Zig-Zag*



Helps those in blue
Hurts those in red

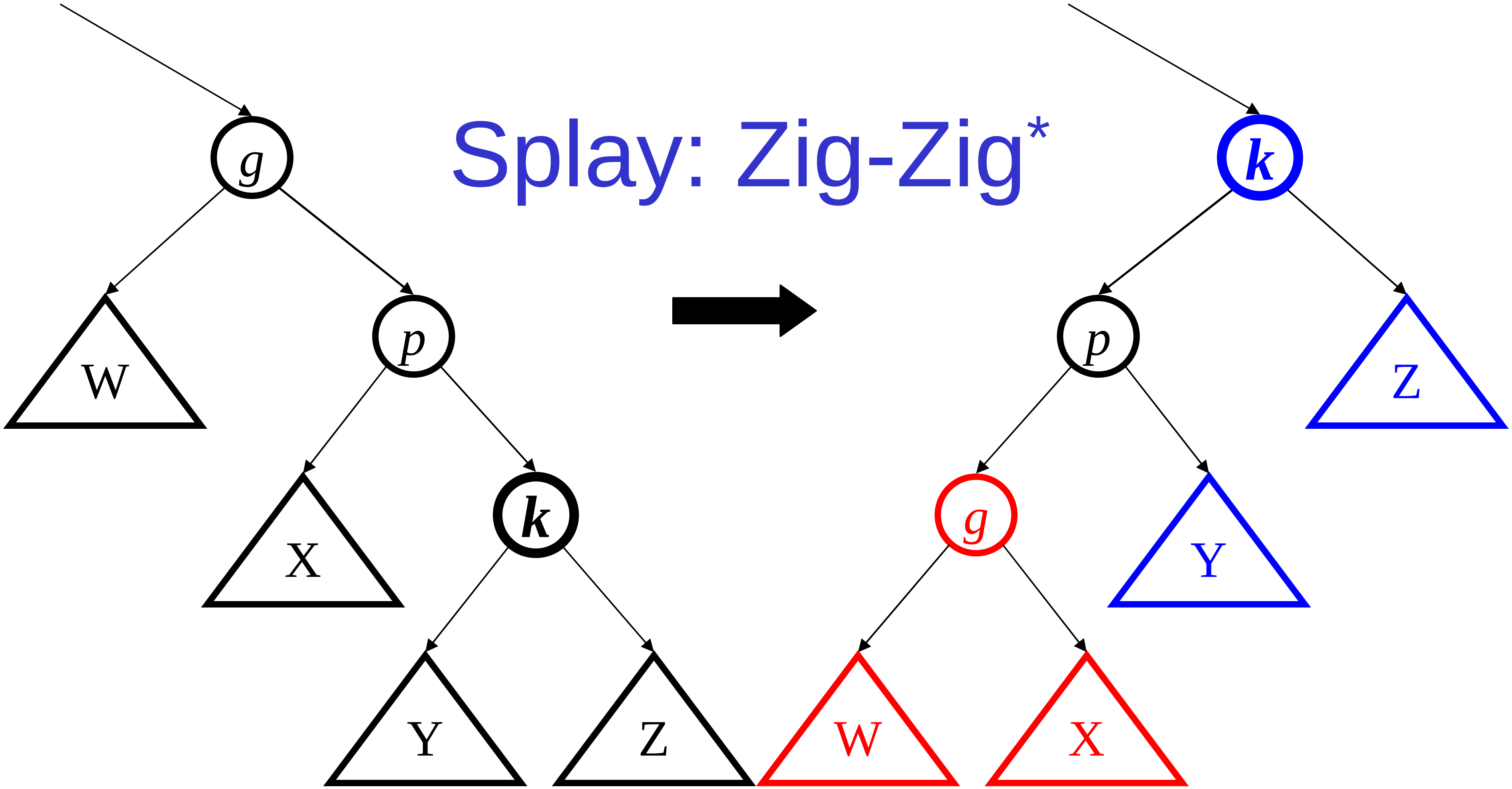
*Just like an…

AVL double rotation

Which nodes improve depth?

k **and** its original children

Splay: Zig-Zig*



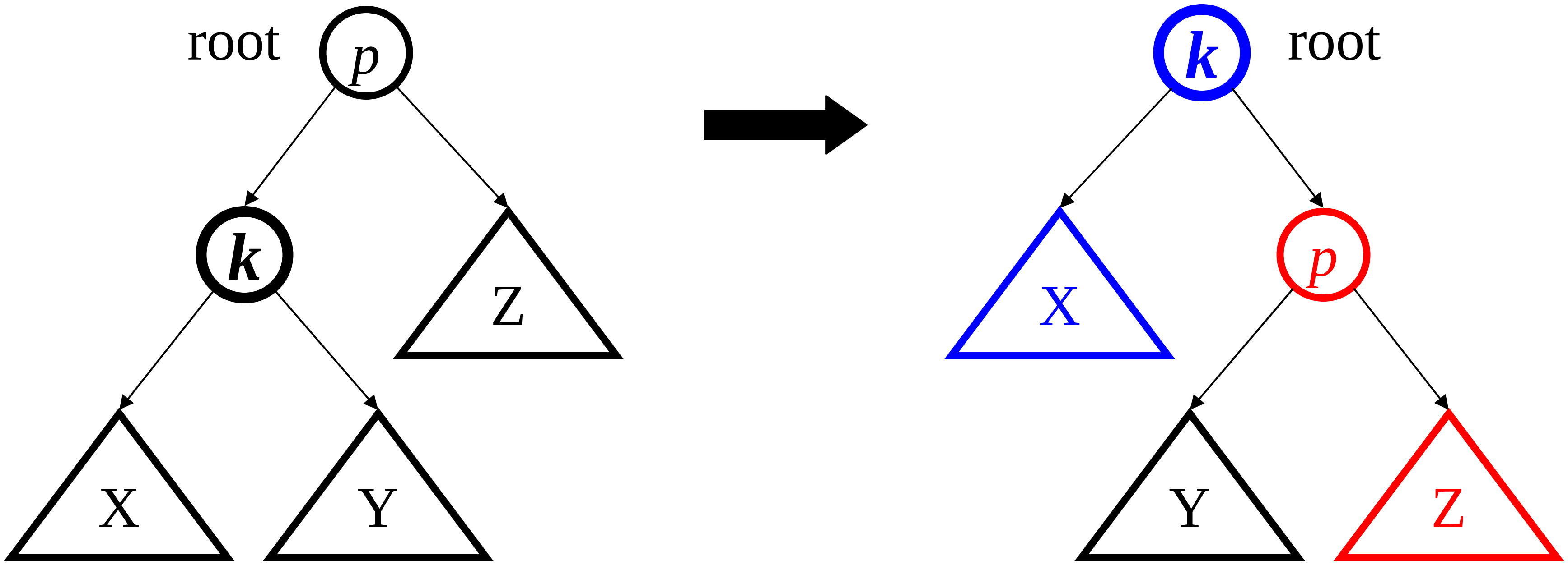
*Is this just two AVL single rotations in a row?

Not quite – we rotate g and p, then p and k

Why does this help?

Same number of nodes helped as hurt. **But** *later* rotations help the whole subtree.

Special Case for Root: Zig



Relative depth of *p*, Y, Z?

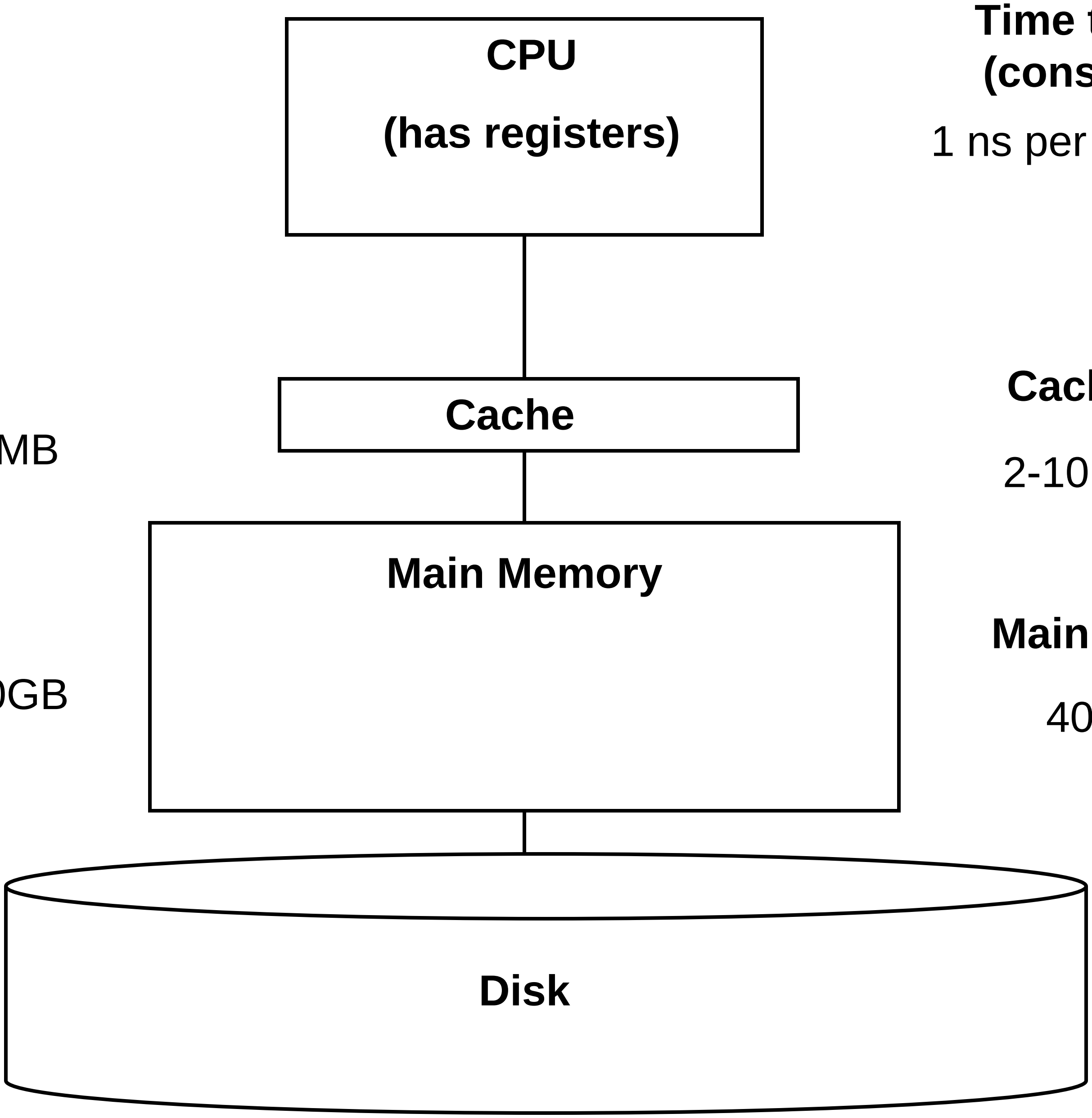
Relative depth of everyone else?

Down 1 level

Much better

Why not drop zig-zig and just zig all the way?

Zig only helps one child!



CPU

(has registers)

Time to access (conservative)

1 ns per instruction

SRAM

8KB - 4MB

Cache

Cache

2-10 ns

Main Memory

DRAM

up to 10GB

Main Memory

40-100 ns

Disk

many GB

Disk

a few *milli*seconds

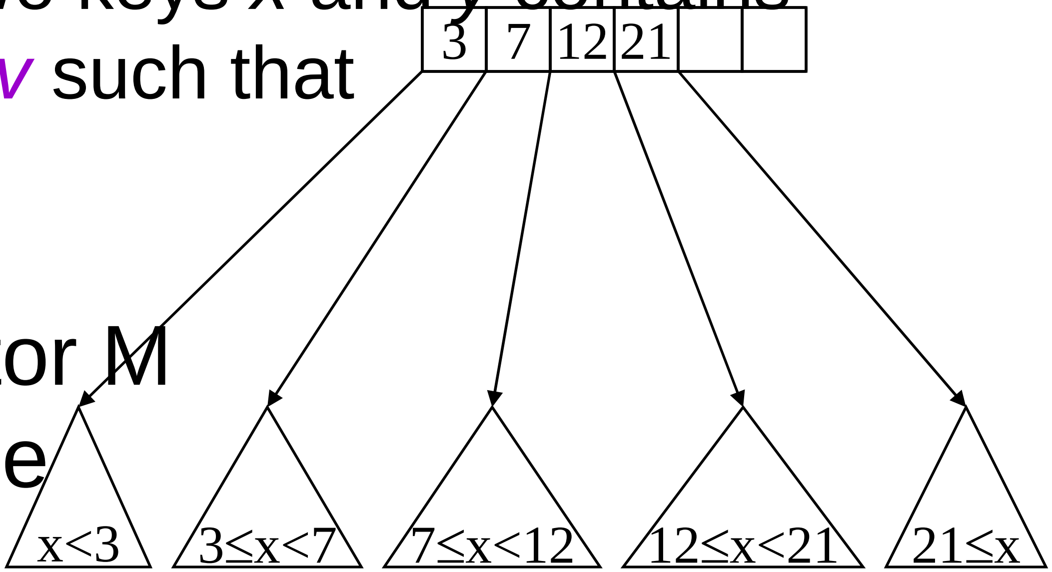
(5-10 *Million* ns)

Solution: B-Trees

- specialized *M*-ary search trees

- Each **node** has (up to) M-1 keys:
  - subtree between two keys *x* and *y* contains leaves with *values* *v* such that $x \leq v < y$

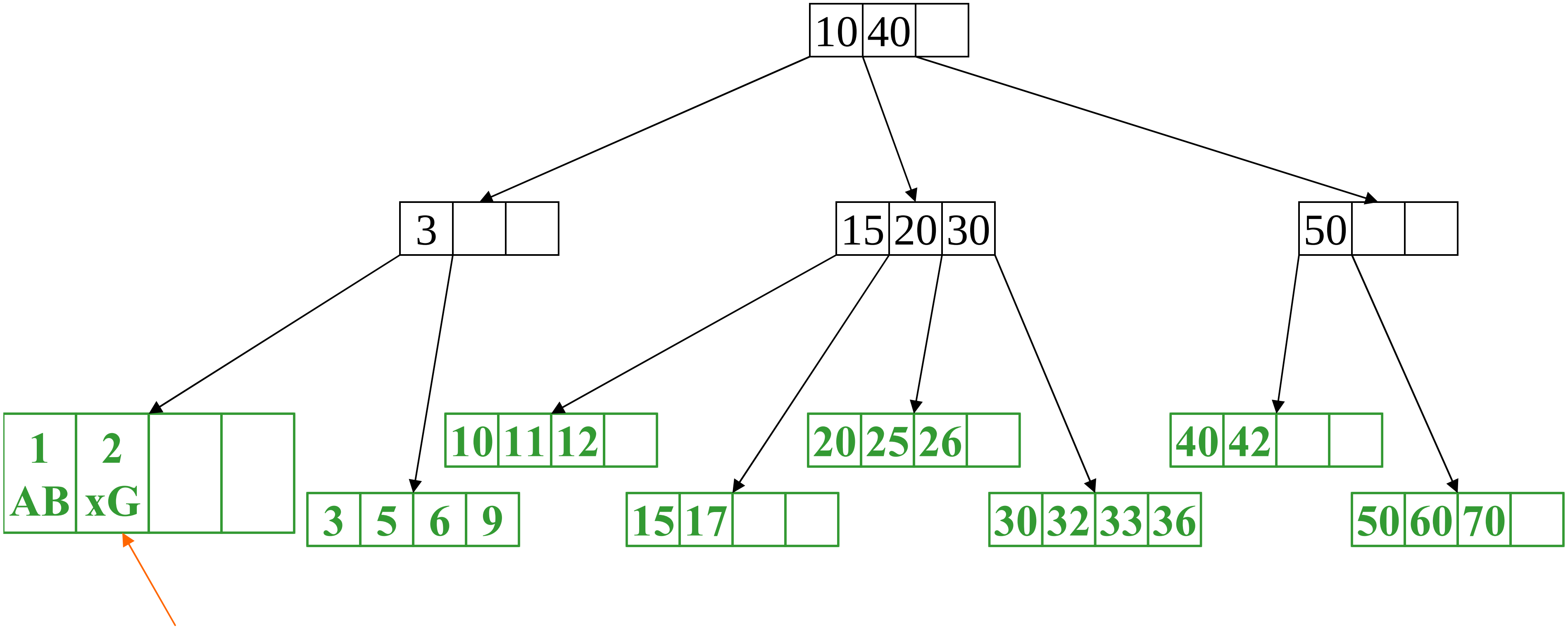
- Pick branching factor M such that each node takes one full {*page*, *block*} of memory



B-Tree: Example

B-Tree with ***M* = 4** (# pointers in internal node)

and ***L* = 4** (# data items in leaf)



Data objects, that I’ll ignore in slides

Note: All leaves at the same depth!

B-Tree Properties [‡]

- Data is stored at the leaves
- All leaves are at the same depth and contains between $\lceil L/2 \rceil$ and L data items
- Internal nodes store up to M-1 keys
- Internal nodes have between $\lceil M/2 \rceil$ and M children
- Root (special case) has between 2 and $\boldsymbol{M}$ children (or root could be a leaf)

[‡] These are technically B^+-Trees

Insertion Algorithm

1. Insert the key in its leaf
2. If the leaf ends up with L+1 items, **overflow**!
   - Split the leaf into two nodes:
     - original with $\lceil (L+1)/2 \rceil$ items
     - new one with $\lfloor (L+1)/2 \rfloor$ items
   - Add the new child to the parent
   - If the parent ends up with $M+1$ items, **overflow**!
3. If an internal node ends up with M+1 items, **overflow**!
   - Split the node into two nodes:
     - original with $\lceil (M+1)/2 \rceil$ items
     - new one with $\lfloor (M+1)/2 \rfloor$ items
   - Add the new child to the parent
   - If the parent ends up with $M+1$ items, **overflow**!
4. Split an overflowed root in two and hang the new nodes under a new root

This makes the tree deeper!

Deletion Algorithm

1. Remove the key from its leaf

2. If the leaf ends up with fewer than $\lceil L/2 \rceil$ items, **underflow**!
    - Adopt data from a sibling; update the parent
    - If adopting won't work, delete node and merge with neighbor
    - If the parent ends up with fewer than $\lceil M/2 \rceil$ items, **underflow**!

Deletion Slide Two

3. If an internal node ends up with fewer than $\lceil M/2 \rceil$ items, **underflow**!
   - Adopt from a neighbor; update the parent
   - If adoption won't work, merge with neighbor
   - If the parent ends up with fewer than $\lceil M/2 \rceil$ items, **underflow**!

4. If the root ends up with only one child, make the child the new root of the tree

This reduces the height of the tree!