# Algorithms & Data Structures

Andrei Krokhin

andrei.krokhin@durham.ac.uk

# Sorting part 2

# This section

- Sorting by counting
- Radixsort

Have seen a good few sorting algorithms so far:

- Selection sort
- Insertion sort
- Merge sort
- Quicksort

Saw that their worst-case running times are somewhere between (asymptotically) $n \log n$ and $n^2$.

Can, in fact prove that that's no coincidence ("that" being that apparently none beats $n \log n$):

**Theorem**

*For any* **comparison-based** *sorting algorithm $\mathcal{A}$ and any $n \in \mathbb{N}$ large enough there exists an input of length $n$ that requires $\mathcal{A}$ to perform $\Omega(n \log n)$ comparisons.*

This is an example of a general *lower bound*: no such algorithm, past, present or future, can consistently beat the stated bound.

We will see a proof of this theorem later.

Two observations:

1. Theorem talks about "comparison-based" algorithms. Vast majority of sorting algorithms are in that class (definitely all that we've seen so far).
2. Theorem says "there exists an input". That means that for most inputs (possibly all but one!) of length $n$ it may actually beat the bound, but that there must be at least one for which it does not.

# BucketSort

# BucketSort

Let's cheat a bit.

Consider the following set-up. You are to sort $n$ numbers, each taking a value in $\{0, \ldots, K-1\}$ (the "range" of possible values is $K$).

If you were to use one of the known algorithms then that would give you a worst-case running time at least $n \log n$ (again, whatever algorithm you choose, at least one input of length $n$ will make it take that many steps).

Specifically, they don't care much about the range of the input items.

# BucketSort

Consider this:

**BucketSort** $(a_1, \ldots, a_n \in \{0, \ldots, K-1\}, n \geq 2)$

```
1: create array C[0, ..., K − 1] and initialise each entry C[i] to
   zero
2: for i = 1 to n do
3:     increment value C[a_i] by one
4: end for
5: for i = 0 to K − 1 do
6:     for j = 1 to C[i] do
7:         print i
8:     end for
9: end for
```

# BucketSort

It's sometimes referred to as "sorting by counting" but more usually as "bucket sort" because it simply chucks elements with key $i$ into the $i$-th bucket, and then empties one bucket after another.

Its running time is $O(n + K)$.

**This means that if $K$ is small, say $\mathbf{o(n \log n)}$, the overall running time of bucket sort is $\mathbf{o(n \log n)}$!!!**

**Apparently beating our lower bound!!!**

# BucketSort

The reason, of course, is that the theorem simply doesn't apply – bucket sort doesn't *do* any comparisons!

Also it's a bit cheating: if $K$ gets really large then the running time is dominated by $\Theta(K)$. If $K = \omega(n \log n)$ then it's worse that that of MergeSort, and if $K = \omega(n^2)$ it's worse than that of *any* other sorting algorithm that we've seen.

# BucketSort

Finally, what if we don't sort numbers but arbitrary things?

Easy: Those things normally come with a numerical key (obtain any way, perhaps by hashing), and you use that for sorting.

Problem is, when we dump the buckets in phase 2, what are we to do? Simple counters don't seem to work any more???

Again, easy: each bucket will be made a queue, and whenever an item's key is, say, $k$, then in phase 1 the item will be appended to the $k$-th queue, and in phase 2 each queue will be dequeued until empty.

# RadixSort

# RadixSort

Obvious drawback: if range of items is large, then we need a large number of buckets (counters or queues).

Improvement: don't look at the item values but "one level below"

Consider again subset of non-negative integers as range, and furthermore consider decimal representation (e.g., decimal digits).

Idea of **RadixSort**:

- have as many buckets as you've got different digits, that is, for base-10 you'll have 10 of them
- repeatedly bucket-sort by given digit
- number of rounds will depend on values (the longer the base-10 representations, the more rounds), but number of buckets only depends on number of different digits

# RadixSort: example

Consider this input:

$$67, 23, 90, 6, 43, 22, 18, 75, 49, 12, 36$$

First pass (right-most digit) gives the following buckets:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 90 | | 12<br>22 | 43<br>23 | | 75 | 36<br>6 | 67 | 18 | 49 |

. . . which, in turn, gives the "new" array

$$90, 22, 12, 23, 43, 75, 6, 36, 67, 18, 49$$

# RadixSort: example

Second round now works on this new array

$$90, 22, 12, 23, 43, 75, 6, 36, 67, 18, 49$$

but now considers the next digit from the right (here: left-most):

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| (0)6 | 18<br>12 | 23<br>22 | 36 | 49<br>43 | | 67 | 75 | | 90 |

Dump this table and get

$$6, 12, 18, 22, 23, 36, 43, 49, 67, 75, 90$$

Voila!

# RadixSort: example

The lot together:

$$67, 23, 90, 6, 43, 22, 18, 75, 49, 12, 36$$

first round (right digit)

$$90, 22, 12, 23, 43, 75, 6, 36, 67, 18, 49$$

second round (left digit)

$$6, 12, 18, 22, 23, 36, 43, 49, 67, 75, 90$$

# RadixSort

. . . works if the BucketSort phases are stable
. . . means, work done in previous rounds isn't being destroyed

Running time: $\Theta(d \cdot n)$ where $d$ is number of rounds (digits per item)

Compare to plain BucketSort for range $\{0, \ldots, K-1\}$:

- RadixSort $d = \log_{10} K$, giving $\Theta(n \cdot \log K)$
- BucketSort $\Theta(n + K)$

Not immediately clear which is better in terms of time (RadixSort clearly better w.r.t space whenever $K$ non-trivial).

Examples:

- if $K = n$ then $\Theta(n \log n)$ vs $\Theta(n) \rightarrow$ BucketSort wins
- if $K = n^2$ then $\Theta(n \log n)$ vs $\Theta(n^2) \rightarrow$ RadixSort wins
- if $K = 2^n$ then $\Theta(n^2)$ vs $\Theta(2^n) \rightarrow$ RadixSort wins

Anyway, if $K$ is constant then both are linear time.

# Searching & Selecting

# Searching

Suppose you're given $n$ numbers in array $A[1 \dots n]$

Further suppose numbers are in sorted order

To make things slightly simpler, assume the $n$ numbers are pairwise distinct, that is, no two are the same

Finally, assume you're also given a number $x$ that's equal to one of the $n$ numbers above, that is,

$$\exists \text{ index } i \in \{1, \dots, n\} \text{ such that } x = A[i]$$

For instance,

$$n = 8, \qquad A[1 \dots 8] = (2, 3, 5, 7, 11, 13, 17, 19), \qquad x = 17$$

Goal: devise algorithm that finds position $p$ of $x$ in $A$
(in example, $p = 7$ because $A[7] = 17 = x$)

# Trivial solution

Scan array left to right, and stop as soon as value $x$ found

## Trivial search

```
int trivial_search (int A[1..n], int x)
{
    p=1
    while (A[p] != x) do
        p = p + 1
    endwhile
    return p
}
```

Notice: absolutely need fact that $x$ is equal to one of the $A[i]$.

Might also ask question: what if this isn't necessarily true, how would you find position $p$ where $x$ **would have to be inserted** in order to maintain overall sorted order?

# Binary search

# Clever(er) solution: (Recursive) Binary Search

Same assumptions as before.

1. Peek right into the middle of the given array, at position $p = \lceil n/2 \rceil$
2. **If A[p] = x** then we're lucky & done, and can return $p$

3 Otherwise, **if x is greater than A[p]**, we may focus our search on stuff **to the right** of $A[p]$ and may completely ignore anything to its left. E.g.,

$$n = 8,\ \lceil n/2 \rceil = 4, \quad x = 17$$

$$A[1 \ldots 8] = (\underbrace{2, 3, 5}_{\text{ignore}}, \boxed{7}, \underbrace{11, 13, \boxed{17}, 19}_{\text{search here}})$$

That's because $x$ is already bigger than $A[p]$ and hence must be **even** bigger than $A[1 \ldots p-1]$
Notice how the problem size got smaller!
Hence, if the "top-level" call was `search(A,1,n,x)` for "search, within array $A$, between the indices of 1 and $n$, for element $x$", we could now recursively call from within this function `search`, using the updated indices, like so:

```
search(A,p+1,n,x)
```

4 Likewise, **if x is smaller than A[p]**, we may focus our search on stuff to the left of $A[p]$ and may completely ignore anything to its right. E.g.,

$$n = 8,\ \lceil n/2 \rceil = 4, \quad x = 3$$

$$A[1 \ldots 8] = (\ \underbrace{2, \boxed{3}, 5}_{\text{search here}}\ , \boxed{7}, \underbrace{11, 13, 17, 19}_{\text{ignore}})$$

That's because $x$ is already smaller than $A[p]$ and hence must be **even** smaller than $A[p+1 \ldots n]$

Notice how the problem size got smaller!

Hence, if the "top-level" call was `search(A,1,n,x)` for "search, within array $A$, between the indices of 1 and $n$, for element $x$", we could now recursively call from within this function `search`, using the updated indices, like so:

```
search(A,1,p-1,x)
```

# Recursive binary search

```
int search (int A[1..n], int left, int right, int x)
{
  if (right == left and A[left] != x)
      handle error; leave function

  p = middle-index between left and right

  if (A[p] == x) then
      return p

  // here come the recursive calls (if x not yet found)
  if (x > A[p]) then
      return search(A,p+1,right,x)  // in right half
  else // x < A[p]
      return search(A,left,p-1,x)  // in left half
}
```

Initial call would be "`search(A,1,n,x)`"

This is called **binary search** because in each (unsuccessful) step it at least halves the search space (we end up with the remaining, interesting part of the array in which $x$ is hiding)

It's an example of the "problem solving paradigm" usually referred to as **divide & conquer**.

A note of interest: the "linear search" requires $\Theta(n)$ comparisons in the worst case, whereas for binary search we have the recurrence

$$T(n) = T(n/2) + O(1) = O(\log n),$$

which, for large $n$, is an awful lot quicker

# Changing the model: Selection

So far, had assumed that

- input is **sorted**, and
- we're looking for the **position** $p$ of an element of given **value** $x$

Now we're changing the set-up:

- input is **unsorted**, and
- we're looking for **value** of the $i$-th smallest element in the input (clearly, if input *were* sorted it'd be trivial: return $i$-th from left)

This problem is called **Selection**. How can we go about solving it?

Easy if we're looking for smallest/largest (how?), 2nd-smallest/largest (how?), $k$-th smallest/largest for some constant $k$ (how?).

What about, say, median ($n/2$-nd smallest)? $\sqrt{n}$-th smallest? Ideas?

Yes, we can sort the input, say in time $O(n \log n)$, and **then** simply return the $i$-th element from the left.

However, is it **clear** that we can't solve the problem quicker, in the worst case?

No, actually it used to be far from clear for many, many years. But let's do one step after another.

# QuickSelect

# QuickSelect

Is to selecting what QuickSort is to sorting. Much the same idea.

- recursive
- `Partition()` function for selecting pivot and partitioning into LOW and HIGH (those smaller/bigger than pivot)
- not two recursive calls to sort but now only one:
  diving into the part (LOW or HIGH) where we know the $i$-th smallest element is to be found
  know that after partitioning, pivot is in correct position w.r.t. overall sortedness, so can simply compare pivot position after partitioning with sought index $i$

## QuickSelect (int A[1...n], int left, int right, int i)

```
 1: if (left == right) then
 2:     return A[left]
 3: else
 4:     // rearrange/partition in place
 5:     // return value "pivot" is index of pivot element
 6:     // in A[] after partitioning
 7:     pivot = Partition (A, left, right)

 8:     // Now:
 9:     // everything in A[left...pivot-1] is smaller than pivot
10:     // everything in A[pivot+1...right] is bigger than pivot
11:     // the pivot is in correct position w.r.t. sortedness

12:     if (i == pivot) then
13:         return A[i]
14:     else if (i < pivot) then
15:         return QuickSelect (A, left, pivot-1, i)
16:     else      // i > pivot
17:         return QuickSelect (A, pivot+1, right, i)
18:     end if
19: end if
```

# What about performance?

Let's start with the bad. If things go wrong, i.e., input is in sorted order and you always pick the right-most element as pivot, then just as slow as QuickSort is when it's slow

$\rightarrow$ the part (LOW/HIGH) that's being recursed into is just one element smaller than the current input is

$\rightarrow$ $T(n) = T(n-1) + O(n)$, which means $T(n) = \Theta(n^2)$

So, **actually** can be rather a lot slower than sorting with e.g. MergeSort

# What about performance?

Can show that if one chooses the pivot **at random** from the current sub-problem, between the indices `left` and `right`, then the **expected** running time of this **randomised QuickSelect** is indeed very good, namely $O(n)$

Clearly, can't hope for better than linear! Why, btw?

But do note that this bound on the expectation is just that: with (certainly extremely small, but positive) probability it may still take $\Theta(n^2)$ many steps

# Median-of-Medians

# Median-of-Medians

Was long unknown whether **worst case linear-time** deterministic selection in **unsorted** data is possible.

Recall: randomised QuickSelect will be fine almost all the time, but the $O(n)$ is only in expectation

Answer: **yes** - the **Median-of-Medians** algorithm

- published by Blum, Floyd, Pratt, Rivest and Tarjan in 1973 (them's some famous names in Computer Science!)
- is not singly but **doubly** recursive!!!
- can be thought of as QuickSelect with a good deterministic pivot-selection strategy
- has **guaranteed** linear time complexity for finding the (any) $i$-th smallest element of a sequence of $n$ unsorted items

# Median-of-Medians: the algorithm

**Select** (int A[1..n], int i)

0 If `length(A)` $\leq 5$ then sort and return $i$-th smallest

1 Divide $n$ elements into $\lfloor n/5 \rfloor$ groups of 5 elements each, plus at most one group containing the remaining $n \bmod 5 < 5$ elements

2 Find median of each of the $\lceil n/5 \rceil$ groups by sorting each one, and then picking median from sorted group elements

3 Call **Select** recursively on set of $\lceil n/5 \rceil$ medians found above, giving median-of-medians, $x$

4 Partition entire input around $x$. Let $k$ be # of elements on low side plus one (simply count after partitioning):

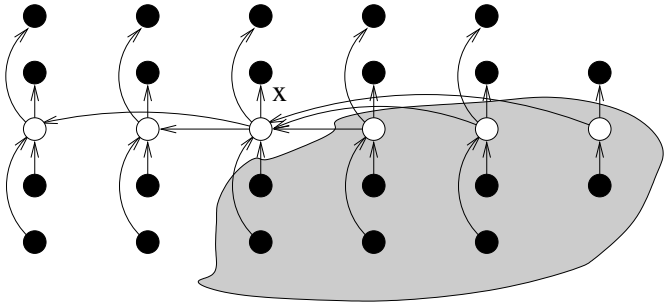
- $x$ is $k$-th smallest element, and
- there are $n - k$ elements on high side of partition

5 If $i = k$, return $x$. Otherwise use **Select** recursively to find $i$-th smallest element of low side if $i < k$, or $(i - k)$-th smallest on high side if $i > k$

Notice: high-level structure exactly the same as QuickSelect

- red: base case
- blue: find pivot
- green: recurse

# Observations




- $n = 5 \cdot 5 + 3 = 28$ elements, depicted as nodes
- groups of 5 are arranged in columns (last one may not be complete)
- white nodes are medians of groups
- $x$ is median of medians
- arrows from greater to smaller elements:
  - three out of every full group to right of $x$ are $> x$, and
  - three out of every group to left of $x$ are $< x$

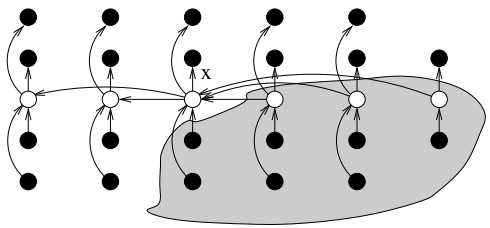
  (because the corresponding medians are greater/smaller than $x$).
- elements on shaded background are guaranteed to be greater than $x$

# Plan of attack

To figure out recurrence, need to

- upper-bound size of LOW side (those $< x$)
- upper-bound size of HIGH side (those $> x$)

Trouble is, that's tricky to do directly.




E.g., those on shaded background are **guaranteed to be bigger**, but that's all we know, and that's a **lower bound** (at least those are bigger, perhaps others as well)

So, we do it indirectly:

- lower bound on HIGH side $\Rightarrow$ upper bound on LOW side
- lower bound on LOW side $\Rightarrow$ upper bound on HIGH side

## We want to lower-bound # elements greater than $x$

Know: at least half of medians found in step 2 are greater than or equal to $x$ ($x$ itself is equal to $x$, and the others are strictly greater)

Thus at least half of $\lceil n/5 \rceil$ groups contribute 3 elements greater than $x$ (except for incomplete group, and group containing $x$ itself)

Disregard these two, and we get

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5}\right\rceil\right\rceil - 2\right) \geq \frac{\mathbf{3n}}{\mathbf{10}} - \mathbf{6}$$

## Same is true for # elements smaller than $x$

- we have at least that many that are strictly greater
- we have at least that many that are strictly smaller

Thus, in worst case, **Select** is called recursively on at most

$$n - \left(\frac{3n}{10} - 6\right) = \frac{7n}{10} + 6$$

elements (step 5): we have just established lower bounds for each partition, and thereby also upper bounds on part to dive into

**Steps 1, 2, and 4**: $O(n)$ time each

**Step 3**: time $T(\lceil n/5 \rceil)$

**Step 5**: time $T(7n/10 + 6)$

**Altogether**:

$$T(n) \leq \begin{cases} \Theta(1) & n < 140 \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) & n \geq 140 \end{cases}$$

Will see the reason for "140" later

Recall:

$$T(n) \leq \begin{cases} \Theta(1) & n < 140 \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) & n \geq 140 \end{cases}$$

Want to show $T(n) \leq cn$ for some constant $c$ and $\forall n > 0$

Clear: When $n < 140$ then $T(n) \leq cn$ for $c$ large enough

Pick constant $a$ s.t. the $O(n)$ term is at most $a \cdot n$ for all $n$ (non-recursive components). Now go forth and substitute:

$$\begin{aligned} & T(n) \\ \leq\ & c\lceil n/5 \rceil + c(7n/10 + 6) + an \leq c(n/5 + 1) + c(7n/10 + 6) + an \\ \leq\ & cn/5 + c + 7cn/10 + 6c + an = 9cn/10 + 7c + an \\ =\ & cn + (-cn/10 + 7c + an) \end{aligned}$$

Now $T(n) \leq cn$ if and only if

$$-cn/10 + 7c + an \ \leq \ 0$$

equivalent to

$$c \geq 10a\frac{n}{n - 70}$$

when $n > 70$

Assumption $n \geq 140$, thus $\frac{n}{n-70} \leq 2$, thus choosing $c \geq 20a$ satisfies inequality

Note: for recursion end, any value greater than 70 would do

OK, that's it: $T(n) \leq cn$ for $c \geq 20a$, thus linear running time

# QuickSelect vs. Median-of-Medians

- QS is singly recursive – so less work in each iteration than MoM
- QS can have more iterations than MoM (if we are unlucky with pivot-selection)

- In practice, the choice of selection algorithm depends on circumstances.
- (Randomized) QuickSelect is often used.
  - Because its bad behaviour is rare, which is often tolerable.
- MoM is used when guaranteed good behaviour is absolutely needed.

# Trees

# Trees

Now for something more data-structurely.

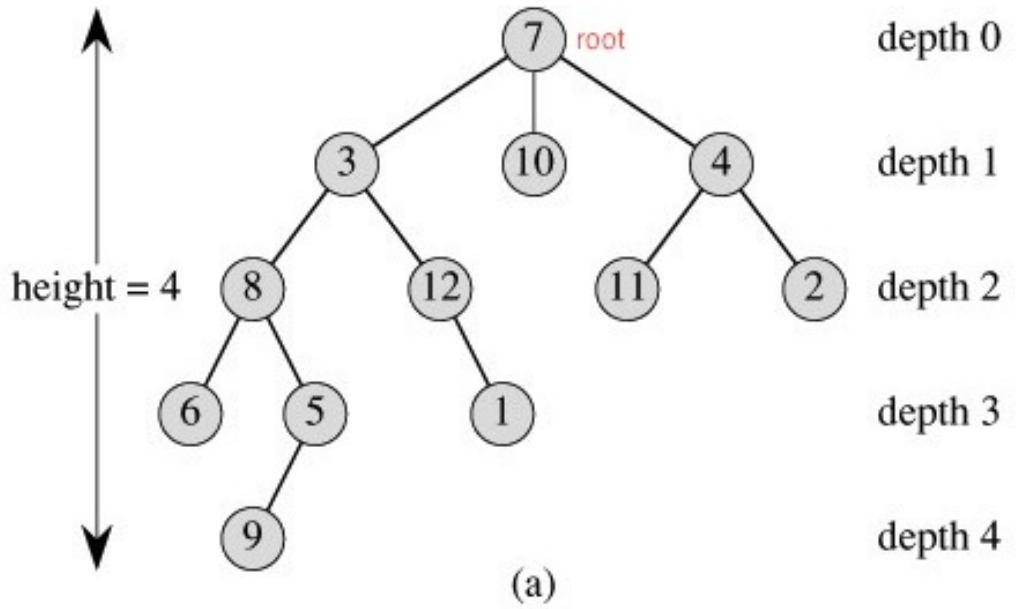
General trees will be studied in Part 4, we'll use only a special type:

Rooted binary trees.

# Rooted trees

.. consist of nodes and parent-child relationship between them.

- there's a special parentless "root node", drawn at the top
- each non-root node has a unique "*parent*" drawn above it,
- each node's "*children*" are drawn right below it (if no more than two children per node then called "*binary tree*")




- Child-less nodes are called "*leaves*"

# Properties of rooted binary trees

## Lemma

*Let $T$ be a rooted binary tree of height $h$. Then*

- *$T$ has at most $2^{h+1}-1$ nodes,*
- *$T$ has at most $2^h$ leaves.*

## Proof.

The max number of nodes is in a complete tree of height $h$ (i.e. all levels are complete): $1+2+2^2+\ldots+2^h = 2^{h+1}-1$.

The second statement follows by induction on $h$. Case $h=0$ is obvious. Consider the left and right subtrees of the root of $T$. Then each of them has height $\leq h-1$ and (by induction hypothesis) $\leq 2^{h-1}$ leaves. Then $T$ has at most $2^{h-1}+2^{h-1} = 2^h$ leaves. □

# We'll be using binary trees to store data

How? Perhaps more importantly, why?

Well, they aren't really all that different from lists

Recall: in a (doubly-linked) list, each element had

- pointer to predecessor, pointer to successor, payload

In a binary tree, it's very similar: each node has

- pointer to parent (or NULL, or possibly to itself, if root)
- pointer to left child (or NULL, or to itself, if there isn't one)
- pointer to right child (or NULL, or to itself, if there isn't one)
- payload

Can then navigate tree much like a list

# Why indeed?

Because they're terribly useful to store data in, giving fast **insert**, **lookup**, and **delete** operations (*dictionary operations*)!

(. . . if done properly)

The idea is simple: suppose you've got a sequence of such "dictionary operations", e.g.

insert(12), insert(27), delete(12), insert(12), lookup(27), . . .

Start with an empty tree

- For each **insert** operation, traverse existing tree according to fixed rules, and insert new element in appropriate place
- For each **lookup** operation, traverse tree according to fixed rules
- For each **delete** operation, first do a lookup, and, if found, delete (and fix tree structure if necessary)

# BST

# The simple binary search tree

May, for some input sequences, not be particularly good

Anyway, basic principle of dictionary data structures could just as well be implemented with list:

- insert(x): append to list
- lookup(x): traverse list and return "TRUE" (plus perhaps pointer to location of element) if found
- delete(x): first lookup, then splice out

BST (binary search tree) not all that different, really

A **binary search tree** (BST) is a tree in which no node has more than two children (not necessarily *exactly* two).

The one additional crucial property of BSTs (this is what we call an *invariant*):

## BST property

You must build and maintain the tree such that it's true for **every node** $v$ of the tree that:

- **all elements in its left sub-tree are "smaller" than** $v$
- **all elements in its right sub-tree are "bigger" than** $v$

Smaller and bigger refer to the payload. The left/right sub-tree refers to the tree rooted in a node's left/right child.

Just saying "*left child smaller and right child bigger*" not sufficient! (Why?)

**Your operations that modify the tree must take care not to destroy this property!**

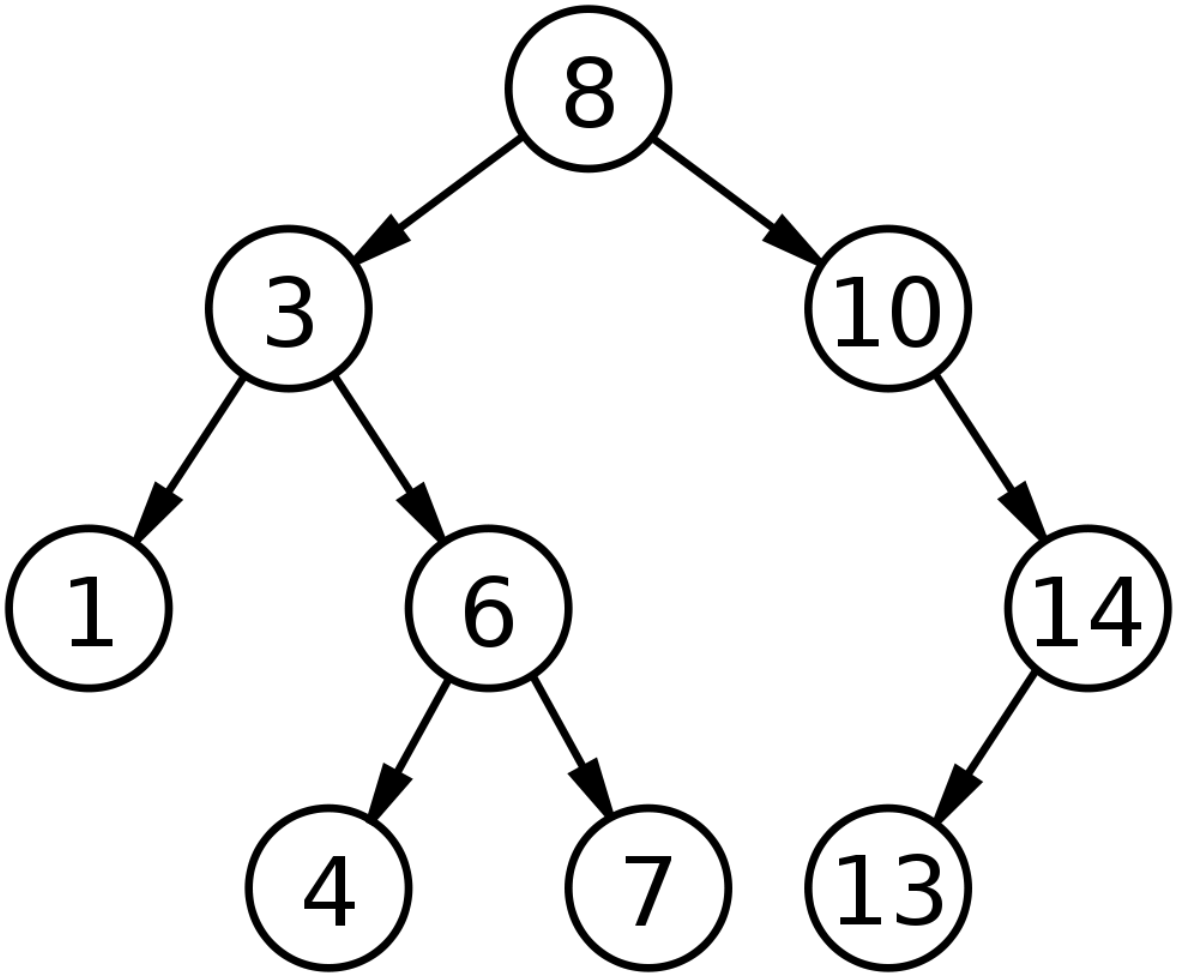
Figures and code in the plain BST part borrowed from

`http://en.wikipedia.org/wiki/Binary_search_tree`

and

`http://www.laurentluce.com/posts/binary-search-tree-library-in-python/`

# Example



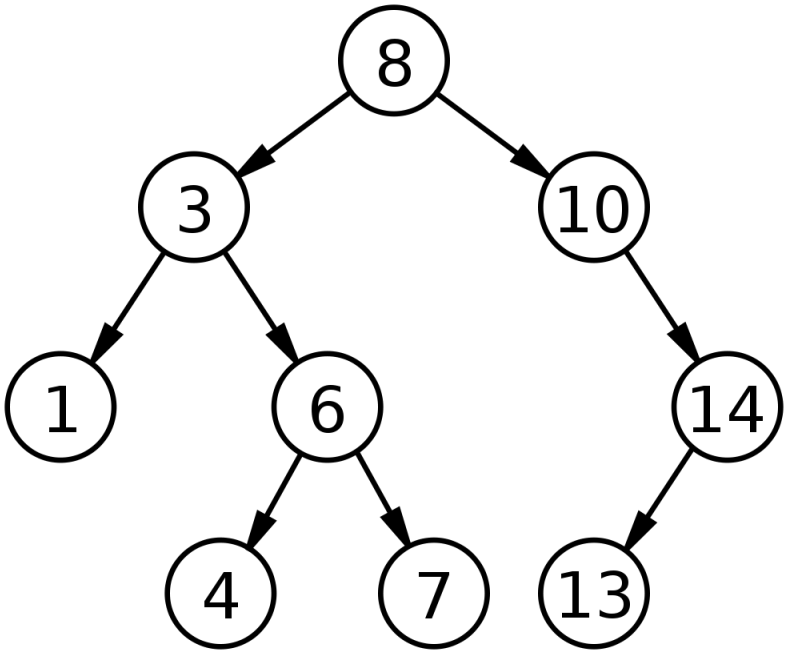
# Traversals

Before we go to BST operations:

May know **tree traversals** from elsewhere. Starting with root:

- **in-order** (for binary trees):
  recurse into left sub-tree, print payload, recurse into right sub-tree
- **pre-order**:
  print payload, recurse into sub-trees
- **post-order**:
  recurse into sub-trees, print payload

# Examples




- **in-order**: 1,3,4,6,7,8,10,13,14
- **pre-order**: 8,3,1,6,4,7,10,14,13
- **post-order**: 1,4,7,6,3,13,14,10,8

Anybody notice anything interesting?

For BSTs, the in-order traversal gives the elements in sorted order!

No coincidence, either!

# In-order traversal of BSTs sorts

By def, in BST

- everything on the left of a node is smaller, and
- everything on the right is bigger.

In-order traversal

- first recurses into left, then
- prints node, then
- recurses into right.

Means:

- first the entire left sub-tree is being printed (all the smaller guys),
- then the node itself,
- then the right sub-tree with the bigger guys.

Next practical: formal proof!

# A class for BSTs

Assume we've got a data structure Node for nodes:

```
class Node:
    """
    Tree node: left and right child + data
               which can be any object
    """
    def __init__(self, data):
        """
        Node constructor
        @param data node data object
        """
        self.left = None
        self.right = None
        self.data = data
```

# Inserting into a BST

```
root = Node(8)
```

creates single node with data (value, payload, key) 8.

# Inserting into a BST

To be called on root of tree.

- If match, return (don't insert again).
- If new key is smaller than that of current node, insert on left.
- Otherwise, insert on right.

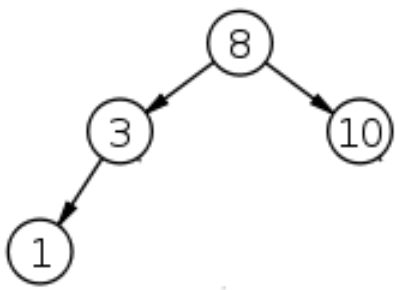
```
class Node:
    ...
    def insert(self, data):
        """
        Insert new node with data
        @param data node data object to insert
        """
        if data < self.data:
            if self.left is None:
                self.left = Node(data)
            else:
                self.left.insert(data)
        else:
            if self.right is None:
                self.right = Node(data)
            else:
                self.right.insert(data)
```

# Inserting into a BST

Consider following sequence of operations after the `root = Node(8)`

```
root.insert(3)
root.insert(10)
root.insert(1)
```

That'll give us

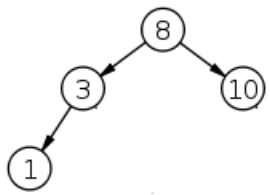


# Inserting into a BST

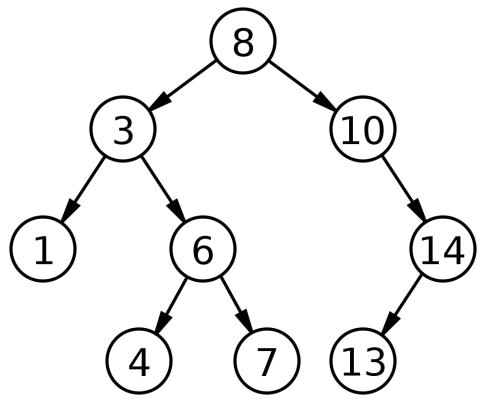
Add this sequence:

```
root.insert(6)
root.insert(4)
root.insert(7)
root.insert(14)
root.insert(13)
```

And



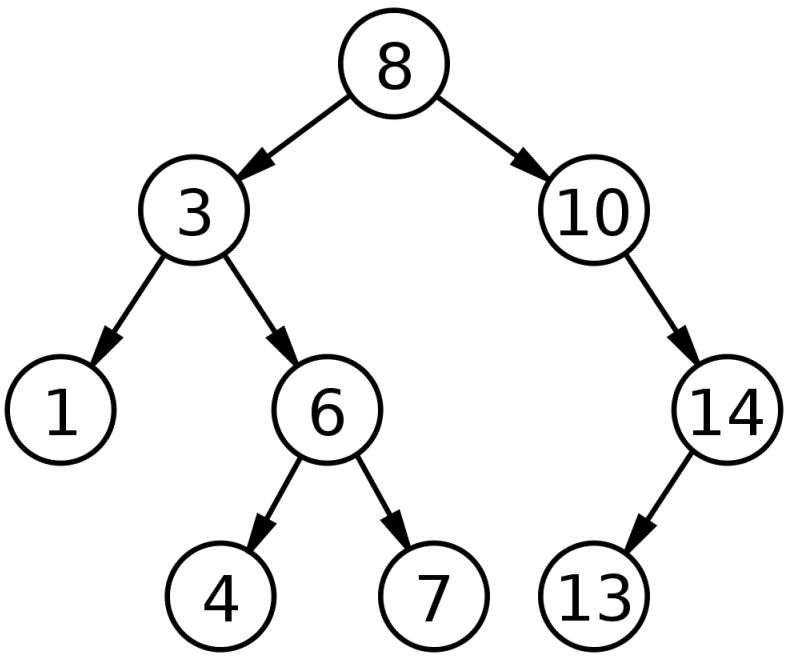

turns into



# Searching in a BST

To be called on the root of the tree.

- If match, return.
- If what we're looking for is smaller, go left (can't be anywhere else).
- Otherwise, go right.



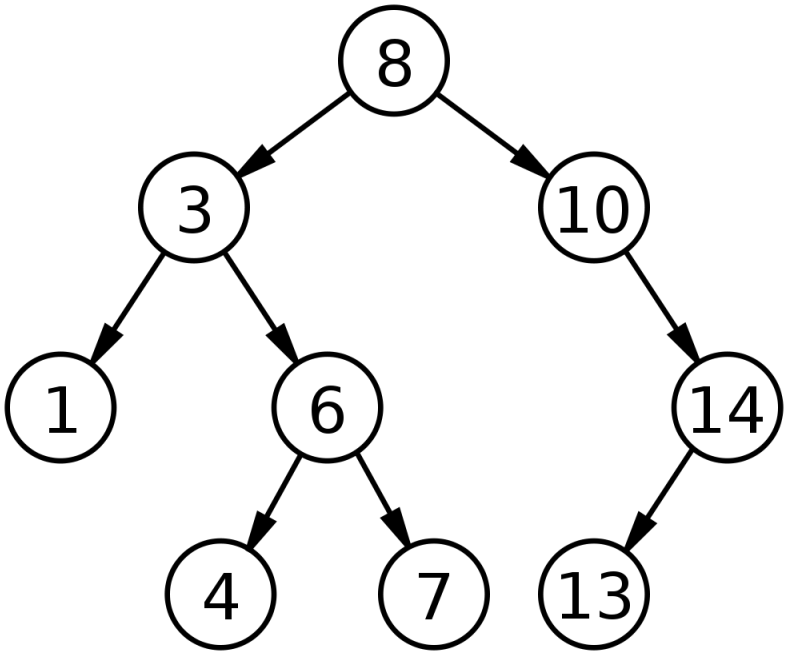
# Searching in a BST

```
class Node:
    ...
    def lookup(self, data, parent=None):
        """
        Lookup node containing data

        @param data node data object to look up
        @param parent node's parent
        @returns node and node's parent if found or None, None
        """
        if data < self.data:
            if self.left is None:
                return None, None
            return self.left.lookup(data, self)
        elif data > self.data:
            if self.right is None:
                return None, None
            return self.right.lookup(data, self)
        else:
            return self, parent
```

This method also returns parent (for later use)

# Searching in a BST

```
node, parent = root.lookup(7)
node, parent = root.lookup(15)
```



# What about deleting?

# Deleting from a BST

Most difficult dictionary operation on BSTs.

We first do a lookup on the element that we wish to remove. If that returns "not found" then we're done (element not in tree).

Otherwise, three cases to be considered:

- If node is a leaf (no child) then simply remove it
- If node has one child (left or right) then remove and replace it with that one child – lift (the one) sub-tree up.
- If node has two children then. . . what?

# Deleting from a BST: counting children

```
class Node:
    ...
    def children_count(self):
        """
        Returns the number of children

        @returns number of children: 0, 1, 2
        """
        if node is None:
            return None
        cnt = 0
        if self.left:
            cnt += 1
        if self.right:
            cnt += 1
        return cnt
```

# Deleting from a BST: setup

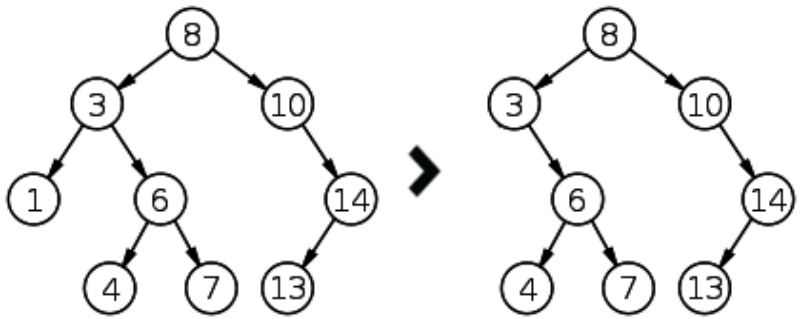
```
class Node:
    ...
    def delete(self, data):
        """
        Delete node containing data

        @param data node's content to delete
        """
        # get node containing data
        node, parent = self.lookup(data)
        if node is not None:
            children_count = node.children_count()
        ...
```

# Deleting from a BST: no child

```
def delete(self, data):
    ...
    if children_count == 0:
        # if node has no children, just remove it
        if parent.left is node:
            parent.left = None
        else:
            parent.right = None
        del node
    ...
```

# Deleting from a BST: no child

```
root.delete(1)
```

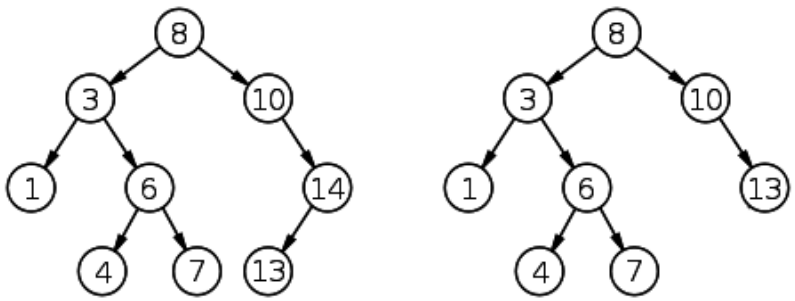


# Deleting from a BST: one child

```
def delete(self, data):
    ...
    elif children_count == 1:
        # if node has 1 child
        # replace node by its child
        if node.left:
                n = node.left
        else:
                n = node.right
        if parent:
            if parent.left is node:
                    parent.left = n
            else:
                    parent.right = n
        del node
    ...
```

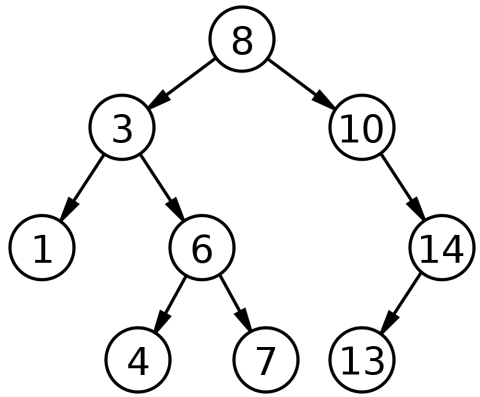
# Deleting from a BST: one child

```
root.delete(14)
```



# Deleting from a BST: two children

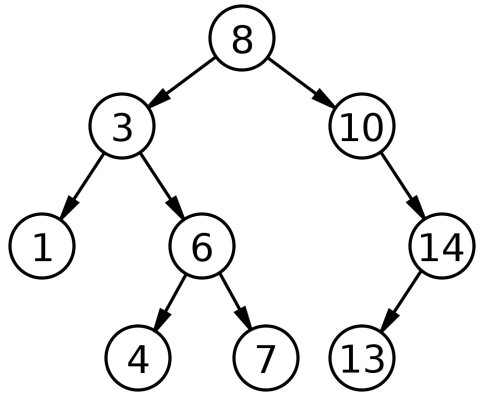
Before we go there: how do you find the smallest element in a given tree?




Answer: starting from root, always go left

# Deleting from a BST: two children

Before we go there: how do you find the smallest element bigger than that in a given node?




Answer: starting from that node, take one step to the right, then always go left

# Deleting from a BST: two children

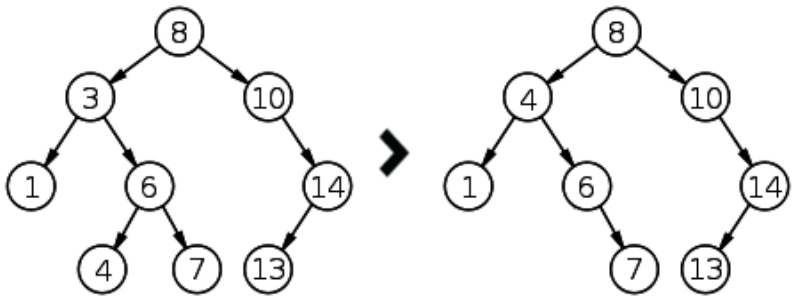
Why is that important, or useful?

Because if we wanted to remove a given node $u$, we could

- find the smallest node $v$ that's bigger
- copy $v$'s data into $u$
- delete $v$

Example:

```
root.delete(3)
```




BST property maintained, everything all right! (Could also have identified largest in left sub-tree and copied that guy's data across!)

# Deleting from a BST: two children

```
def delete(self, data):
    ...
    else:
        # if node has 2 children
        # find its successor
        parent = node
        successor = node.right
        while successor.left:
            parent = successor
            successor = successor.left
        # replace node data by its successor data
        node.data = successor.data
        # fix successor's parent's child
        if parent.left == successor:
            parent.left = successor.right
        else:
            parent.right = successor.right
```

"Successor" here means w.r.t. sorted order (left-most in right subtree)

# Traversing a BST

```
class Node:
    ...
    def print_tree(self):
        """
        Print tree content inorder
        """
        if self.left:
            self.left.print_tree()
        print self.data,
        if self.right:
            self.right.print_tree()
```

# Balanced BSTs

**BST property:** for **all** nodes $v$ with key $k$,

- nodes in left subtree have keys $< k$
- nodes in right subtree have keys $> k$

Have seen simple **BST operations**:

- Lookup, Insert, Delete
- Minimum, Maximum
- Predecessor, Successor

All take $O(h)$ time, $h$ height of BST

**OK** if BST is **balanced**, then $h = O(\log n)$

**Bad** if BST is **degenerated**, then $h = \Omega(n)$
(easy to come up with inputs)

Might be that "thoughtless" insertions & deletions make things bad

How about being somewhat clever, i.e., taking care that BST does **not** degenerate when inserting or deleting?

Preferably without excessive overhead work?

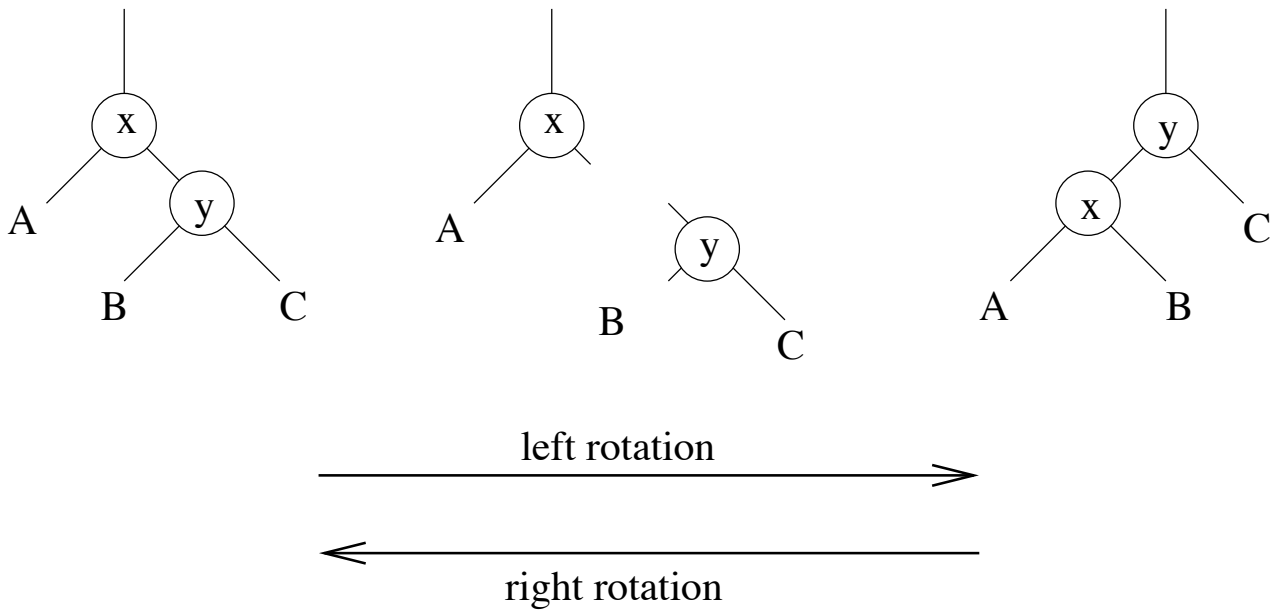
Many (many many...) approaches, we're going to see
**AVL trees** and (very briefly) **red-black trees**

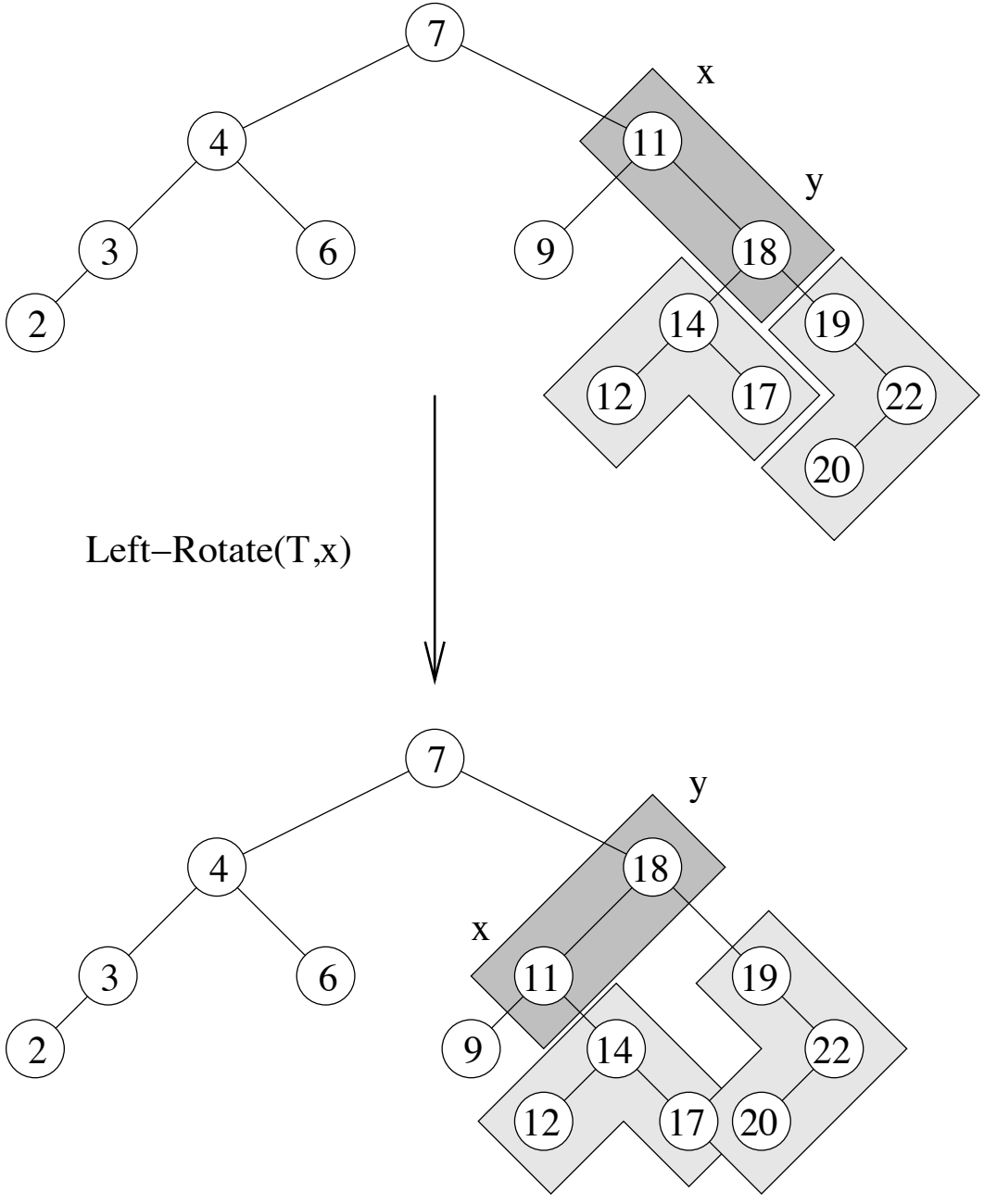
Basic re-balancing operations: **rotations**

Local operation, preserves BST property

**left** and **right** rotations

Assumptions:
left-rotation on $x$: right child not NULL
right-rotation on $y$: left child not NULL




7
4
11
x
y
3
6
9
18
2
14
19
12
17
22
20
Left–Rotate(T,x)
7
y
4
18
x
3
6
11
19
2
9
14
22
12
17
20

The operation is local: it modifies a constant number of parent-child links.

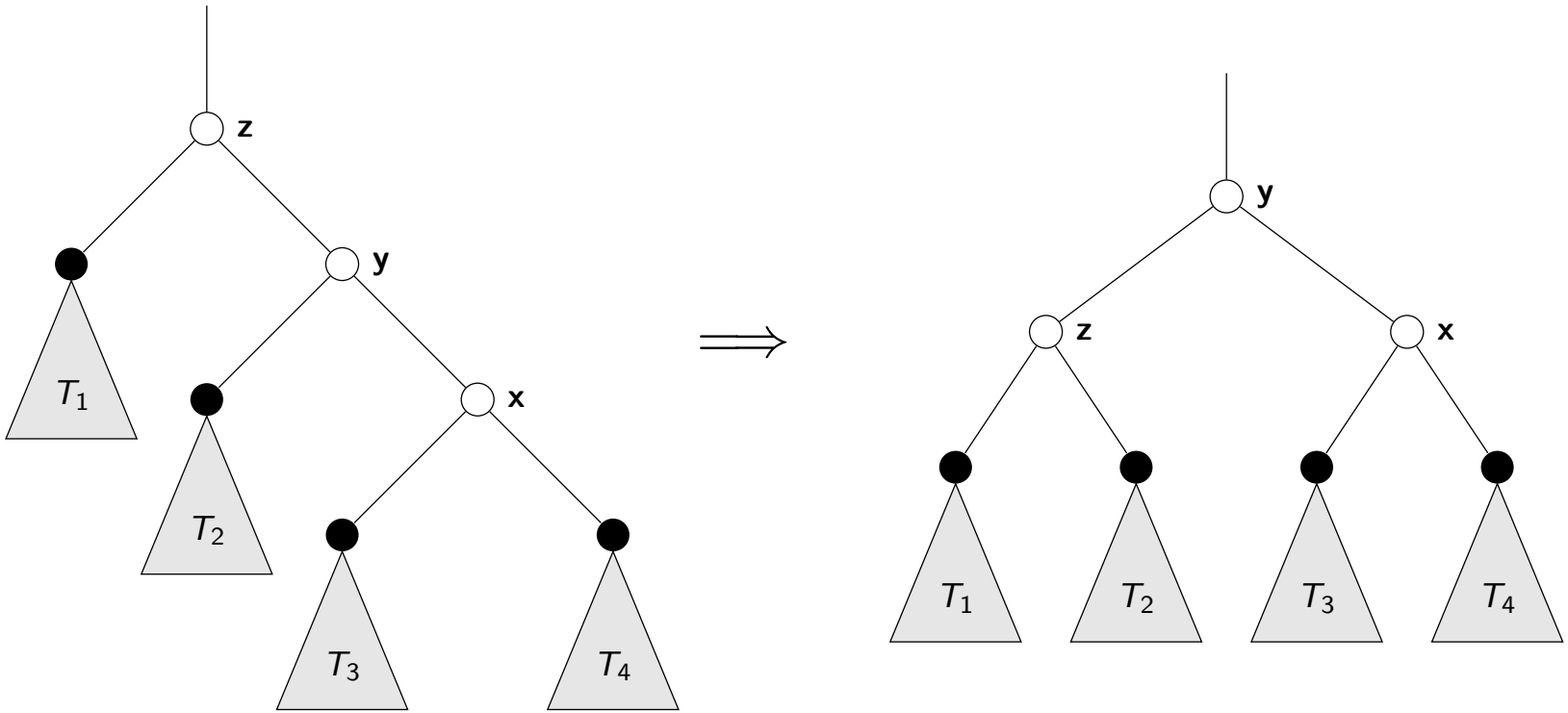
Hence each simple rotation takes $O(1)$ time.

Simple rotations can be combined to provide re-balancing.

We call such a compound operation applied to a triple parent-child-grandchild **trinode restructuring**

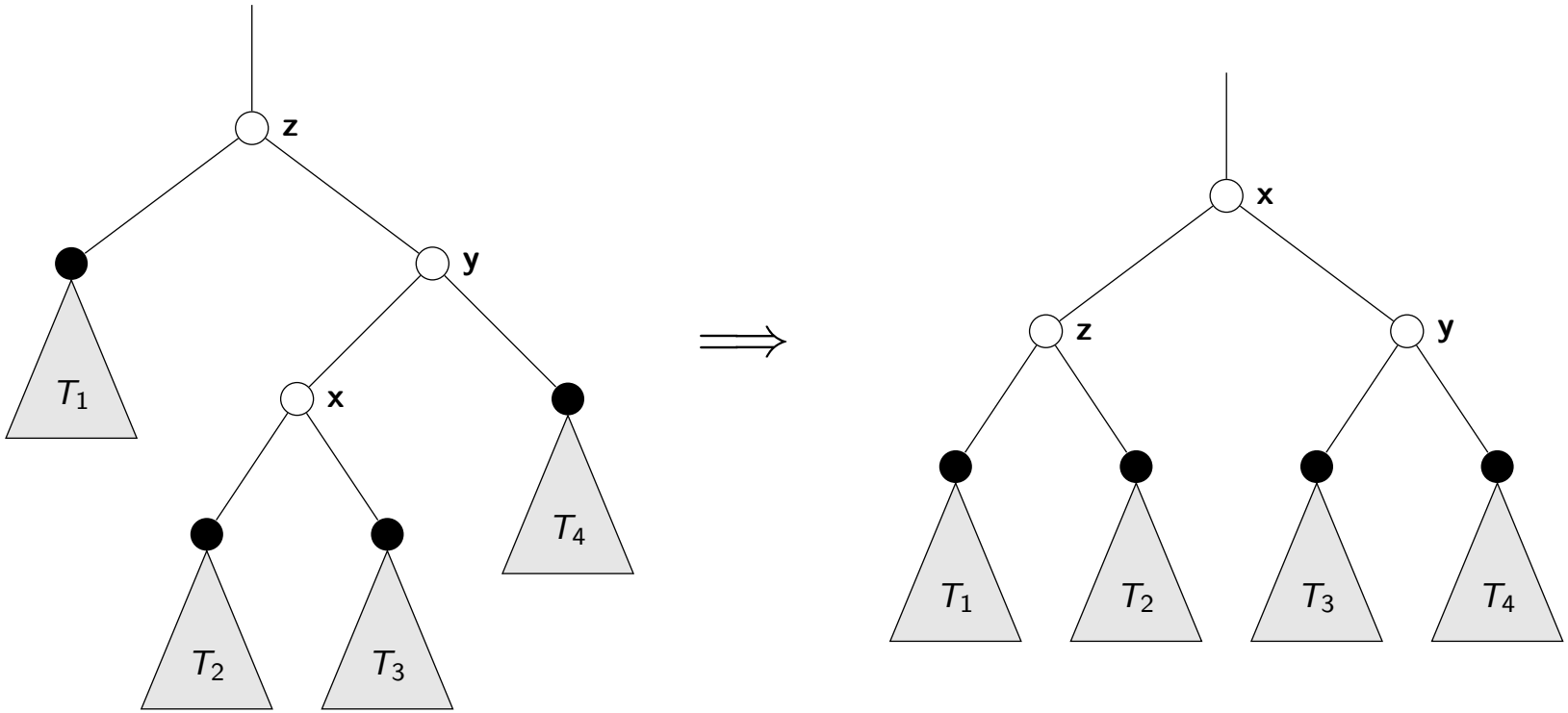
The operation depends on whether child and grandchild are on the same side (i.e. both left or both right) or not.

# Trinode restructuring: single rotation




Single rotation: child and grandchild are on the same side
Here, both are right - do Left-Rotate(T,z)

# Trinode restructuring: double rotation




Double rotation: child and grandchild are not on the same side
Here, right/left - do Right-Rotate(T,y), then Left-Rotate(T,z)

The median of $x, y, z$ always ends up as the root of this subtree

The operation is still local - modifies a constant number of links

Hence, it still takes $O(1)$ time.

The implementation goes through 4 cases:

left/left, right/right, left/right, right/left

# AVL trees

Named after the inventors: Georgy Adelson-Velsky and Evgenii Landis

An **AVL tree** is a BST with the following additional property:

## Height-balance property

**For each node $v$, the heights of $v$'s children differ by at most 1.**

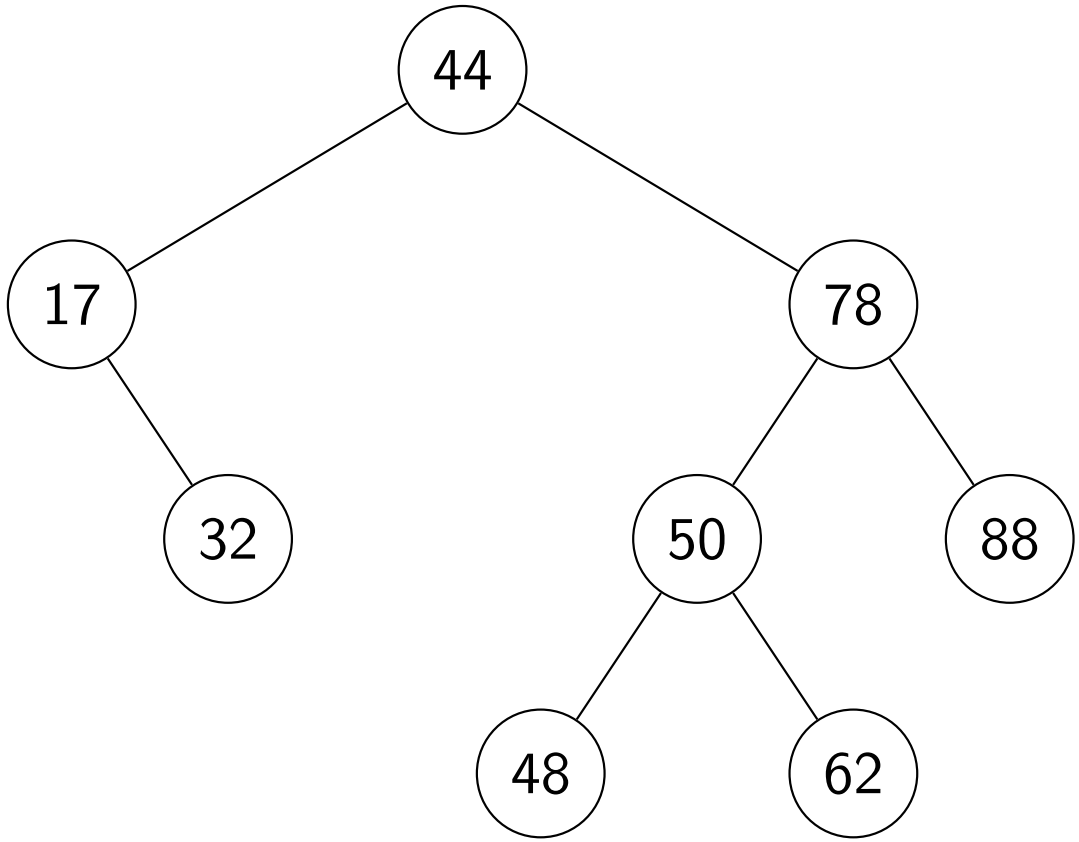
Will use the definition of height of a node:

- height of `Null` is 0 and height of a proper leaf is 1
- height of a parent = max height of a child +1.

Note: height a non-empty tree = height of its root -1.

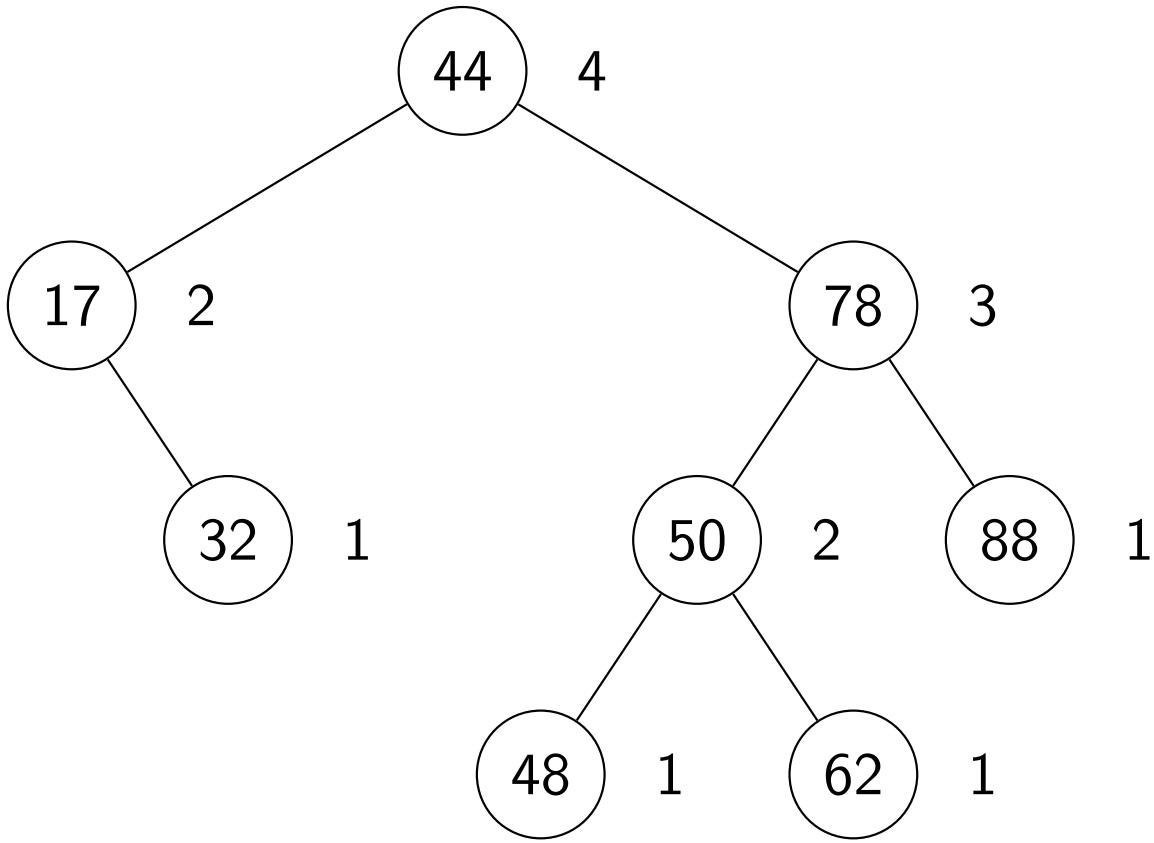
# Example

Is this BST an AVL tree?



# Example

Yes, the height-balance property is satisfied.



# Key property

## Lemma

*The height of an AVL tree with $n$ nodes is at most $2\log_2 n + 2$.*

Proof: Let $n(h)$ be the *smallest* number of nodes in an AVL tree whose root has height $h$ (so the tree has height $h-1$).

What is $n(h)$ for $h = 1, 2, 3, 4$? Answer: $1, 2, 4, 7$.

For $h \geq 3$, we have $n(h) = 1 + n(h-1) + n(h-2)$. Here's why:

Take an AVL tree with $n(h)$ nodes whose root has height $h$. The subtrees rooted at the root's children are AVL trees, and

- one child of the root must have height $h-1$ (so that the height of the root is $h$),
- the other child has height $h-2$, by minimality of $n(h)$
- both subtrees must have the smallest number of elements for their heights, also by the minimality.

# Proof cont'd

Have $n(1) = 1$, $n(2) = 2$, and $n(h) = 1 + n(h-1) + n(h-2)$.

From this formula, have $n(h-1) > n(h-2)$ for $h \geq 3$, so

$$n(h) = 1 + n(h-1) + n(h-2) > 2 \cdot n(h-2).$$

Hence, $n(h) > 2 \cdot n(h-2) > 4 \cdot n(h-4) > \ldots > 2^i \cdot n(h-2i)$.

Taking $i = \lceil \frac{h}{2} \rceil - 1$ (so that $h - 2i$ is 1 or 2), get

$$n(h) > 2^{\lceil \frac{h}{2} \rceil - 1} \cdot n(h - 2\lceil \frac{h}{2} \rceil + 2) \geq 2^{\lceil \frac{h}{2} \rceil - 1} \cdot n(1) \geq 2^{\frac{h}{2} - 1}$$

Taking logs in $n(h) > 2^{\frac{h}{2}-1}$, get $\log_2(n(h)) > \frac{h}{2} - 1$.

Rearrange to get $h < 2\log_2(n(h)) + 2$, and recall that $n(h)$ is the smallest number of nodes for a given height $h$.

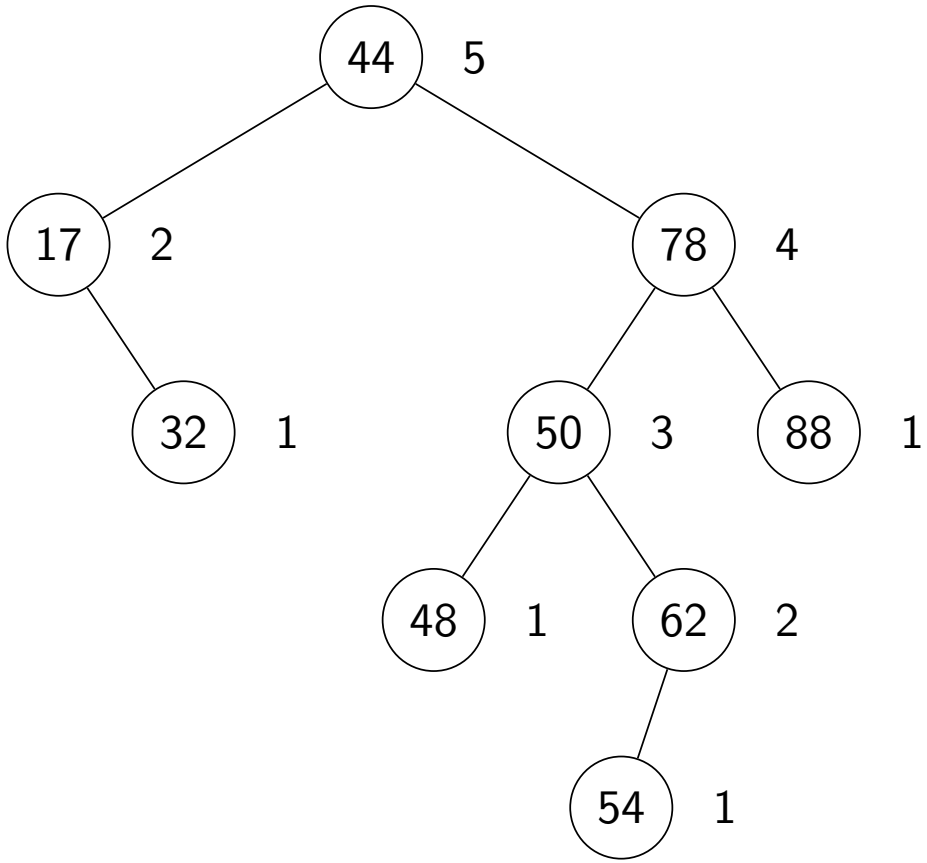
The standard BST operations in AVL trees take $O(\log n)$ time.

Are we done?

- No, insertion and deletion can potentially violate the height-balance property.
  - What happens if we insert 1, 2, 3 into an empty AVL tree?
- So need a way to restore it, preferably in $O(\log n)$ time.
- Trinode restructuring might be useful here.

# Insertion

Take the AVL tree from an earlier example and insert 54:




Height-balance is now violated – how to restore it?

# Post-insertion fix-up

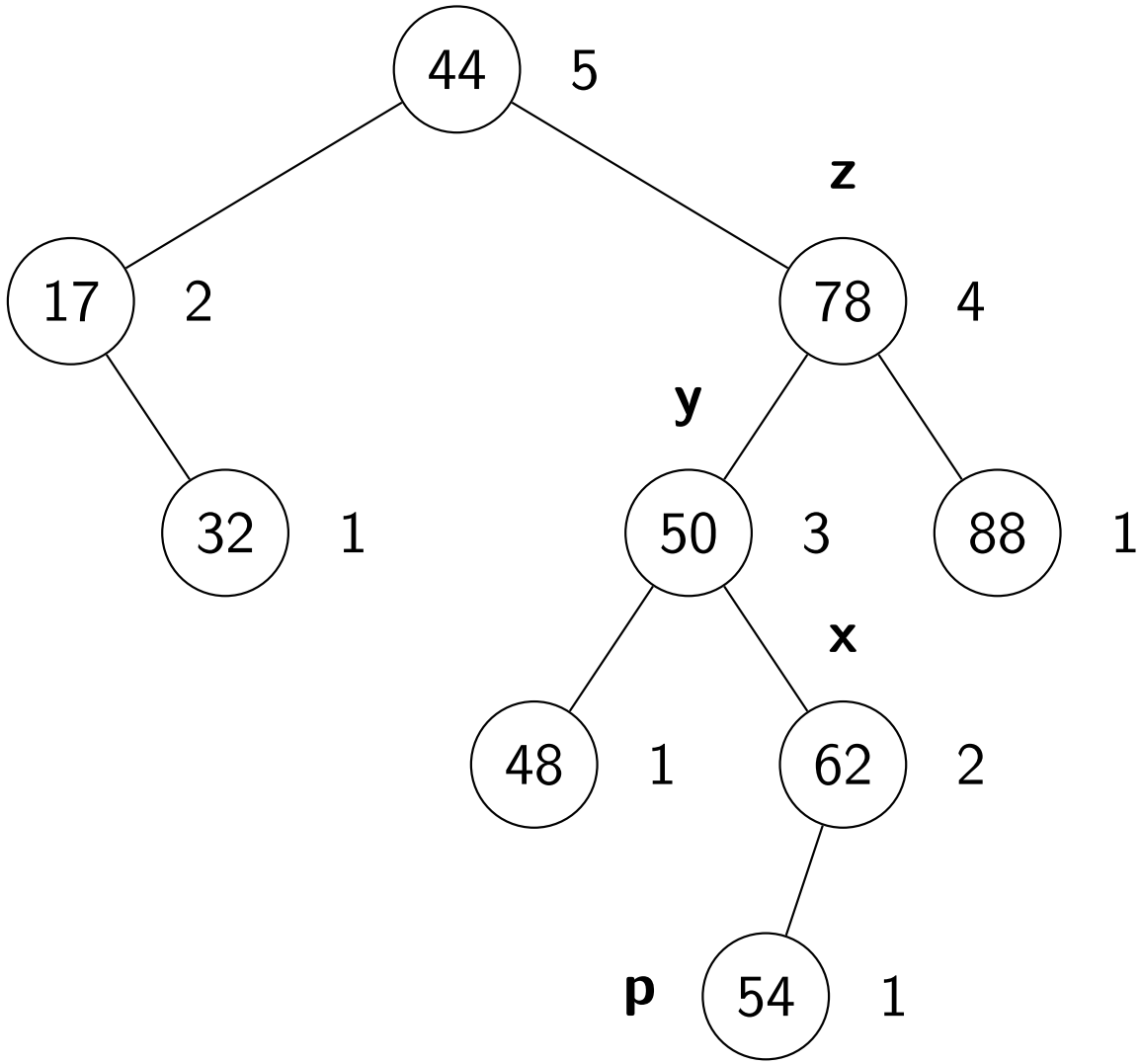
Let $p$ be the inserted node - where are the nodes whose height changed? They can be *only* on the path from $p$ up to the root.

Simple "search-and-repair" recipe:

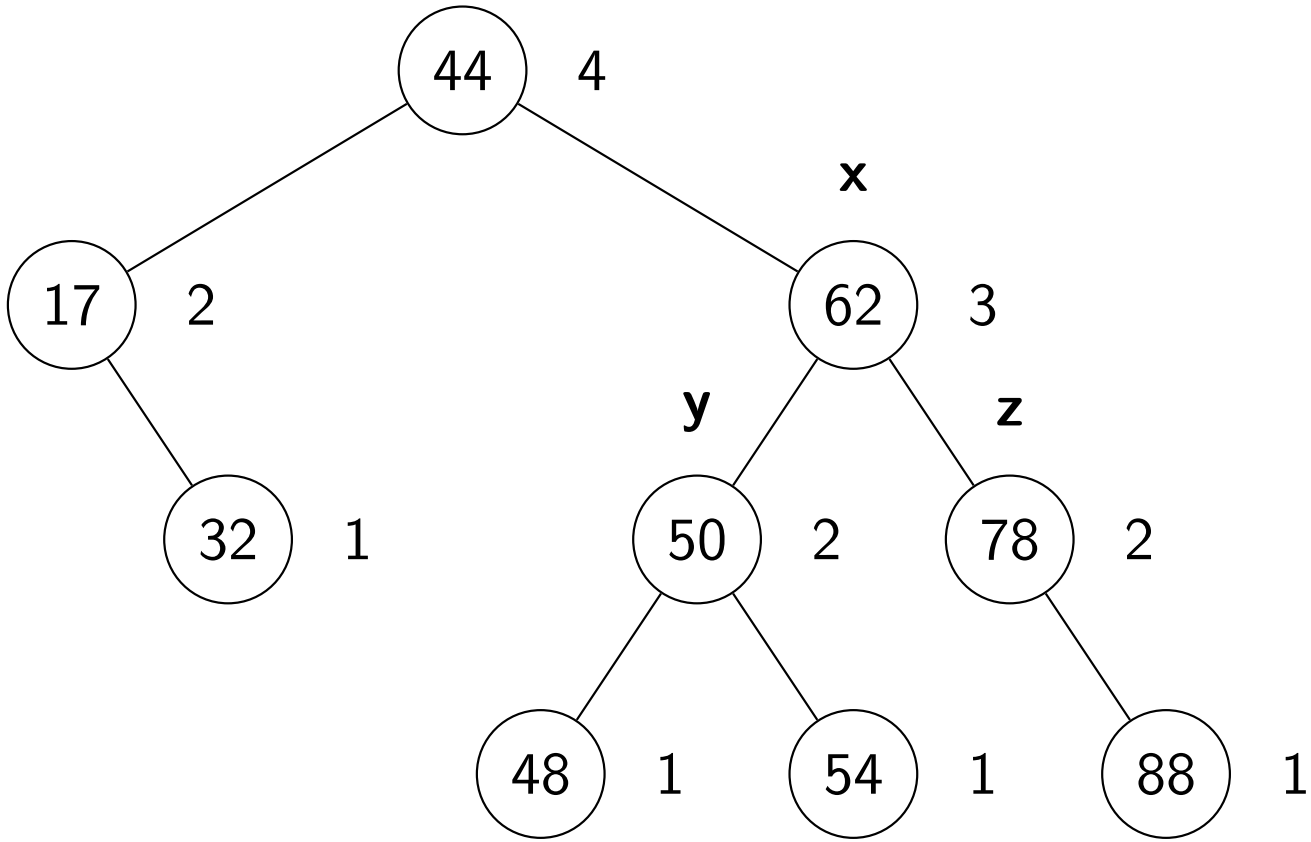
- From the newly inserted node $p$, go up towards the root, adjusting the heights (of nodes on the path)
- Look for the first (from $p$) node $z$ which is *unbalanced*
  - i.e. heights of its children differ by more than 1
- If no such $z$ is found, the height balance is not violated
- If it is found, let $y$ and $x$ be the child and grandchild (respectively) of $z$ on the path to $p$.
  Do trinode restructuring on $z - y - x$

This restores the height-balance property (will see why).

# Example



# After trinode restructuring

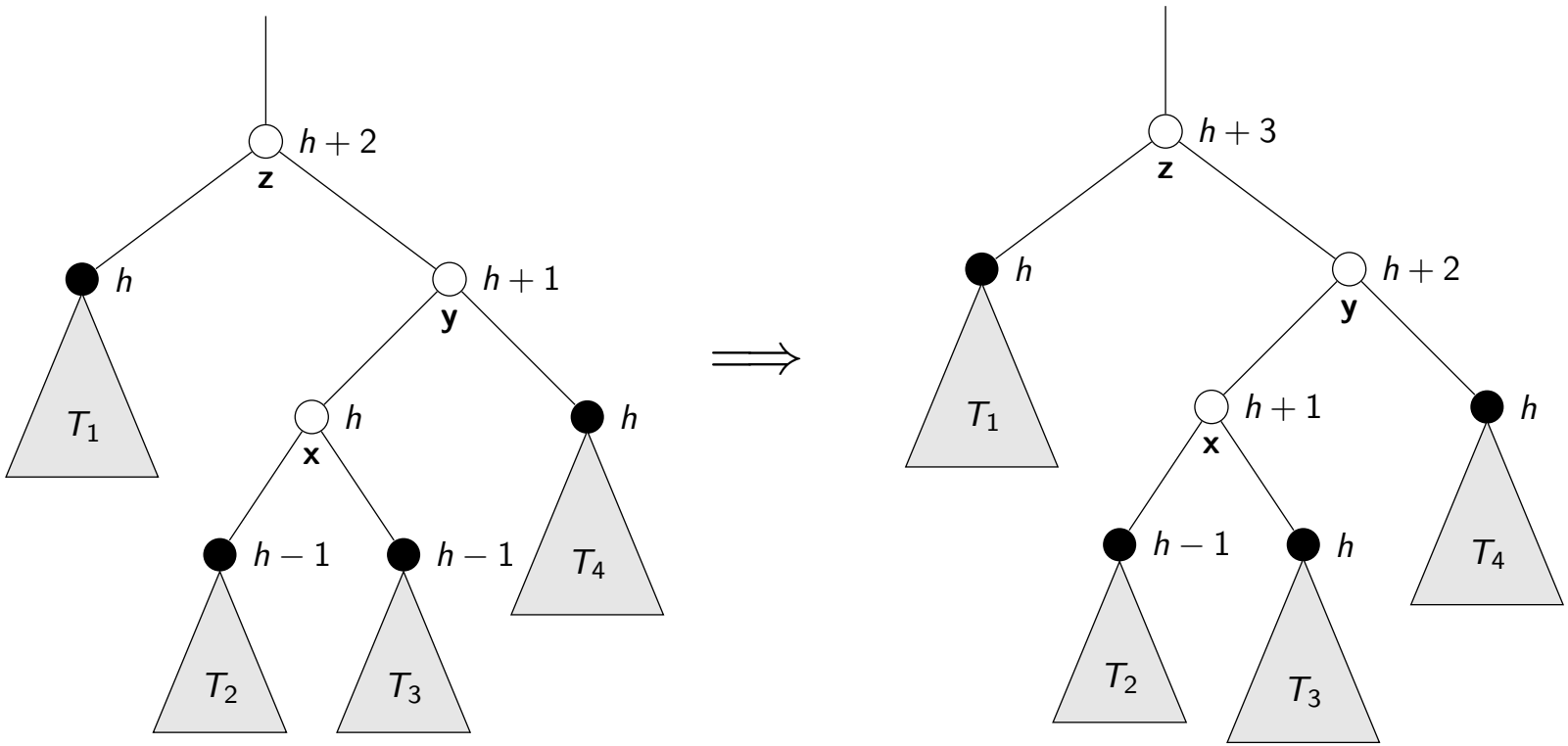



Note: the height-balance property is restored. Let's argue that this will always be the case.

# Proof of re-balancing

Assume $z$ is the first unbalanced node on the path from $p$ up.

The subtree rooted at $z$ before and after insertion (into $T_3$):



# Observations

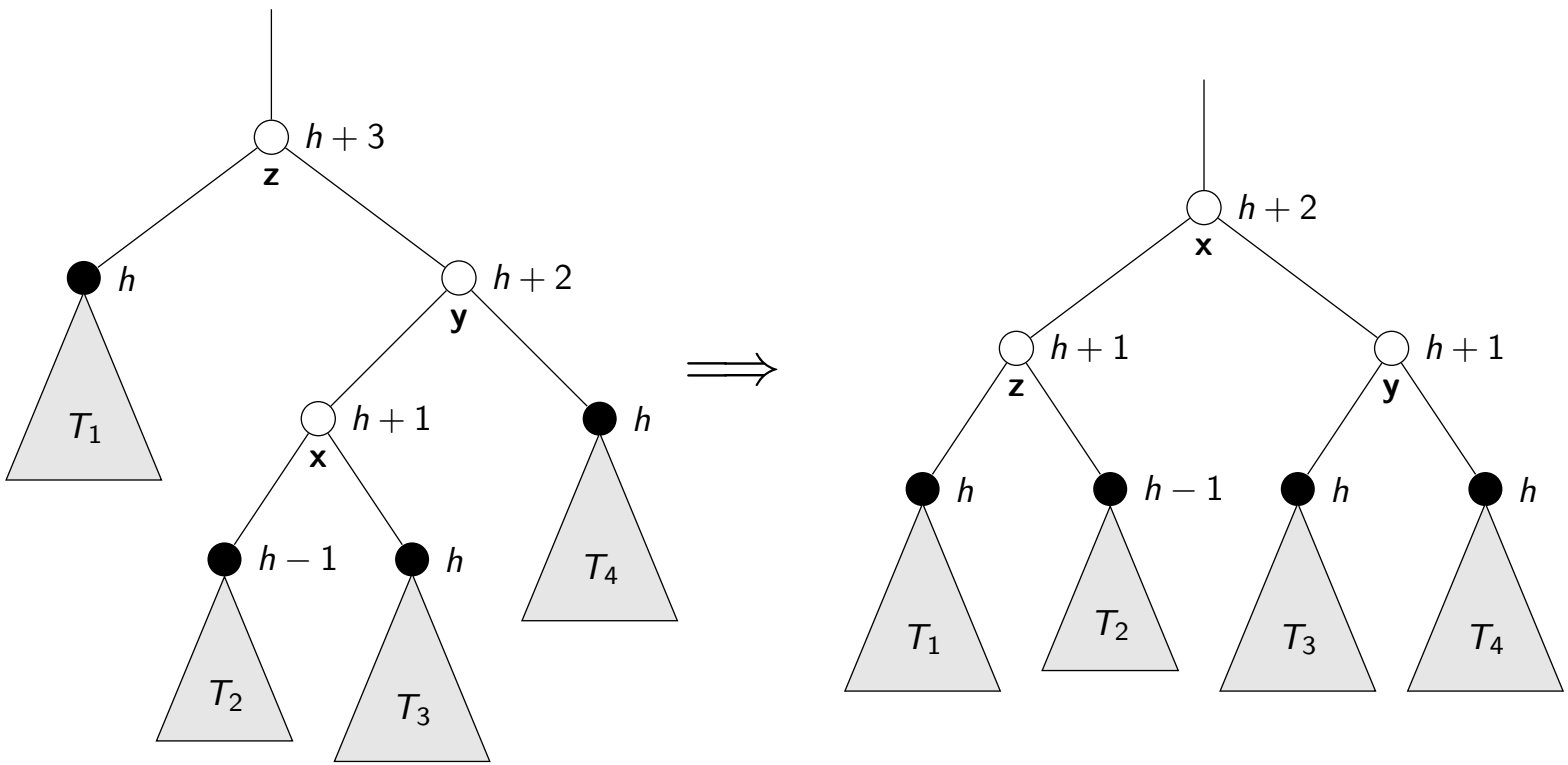
Since $z$ becomes unbalanced, but $y$ doesn't, we must have:

- The height of $z$ goes up by 1
- If $y'$ is the sibling of $y$ then the height of $y$ goes up by 1 and is now the height of $y' + 2$
- The children of $y$ had equal height before the insertion, and the height of $x$ goes up by 1
- Since the height of $x$ goes up by 1, have $x = p$ (and its height goes from 0 to 1) or the children of $x$ had equal height

It follows that if $h \geq 0$ is height of tallest child of $x$ after the insertion, then the heights are as shown on the previous slide.

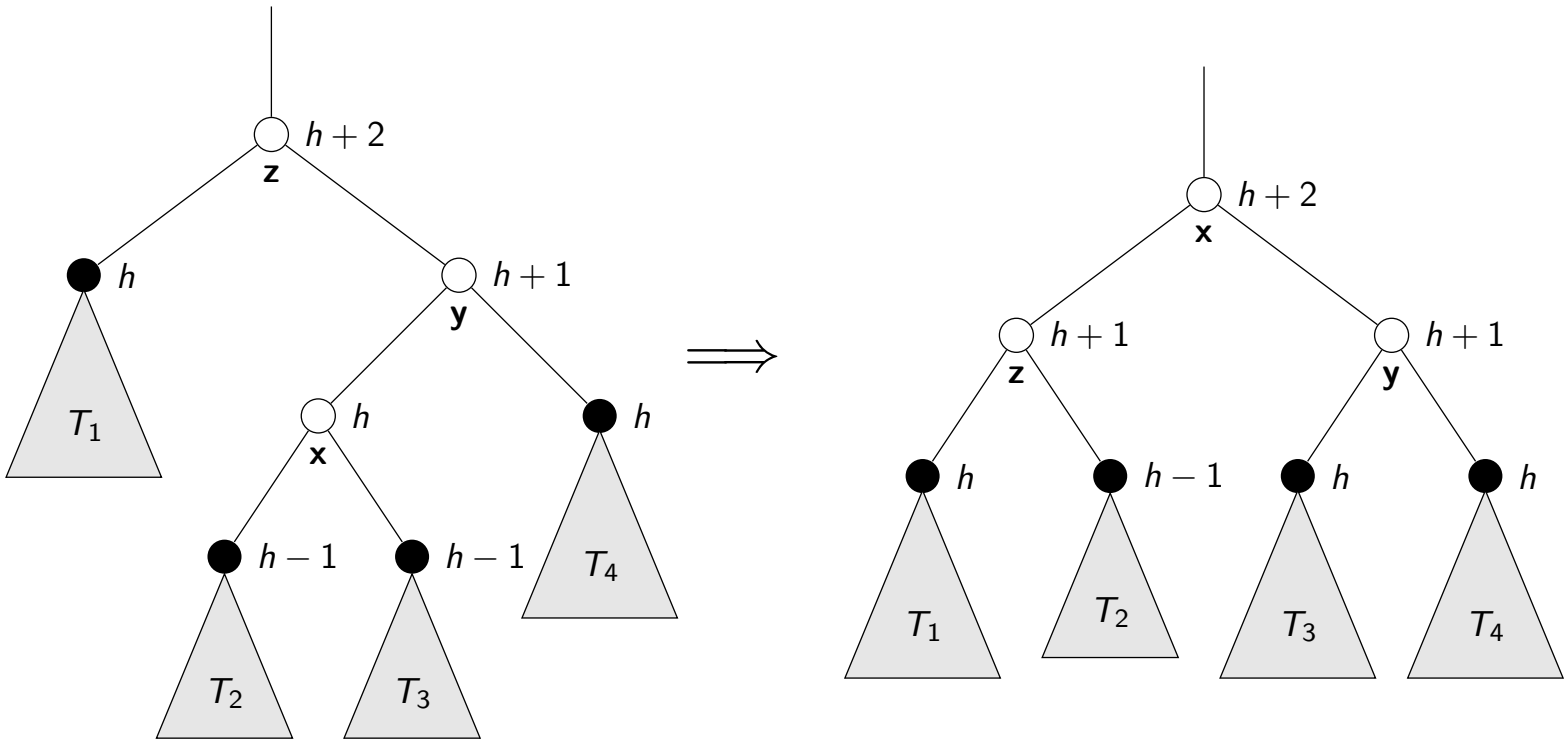
# Proof of re-balancing cont'd

The subtree rooted at $z$ before and after trinode restructuring:



# Proof of re-balancing finished

Insertion and fix-up (trinode restructuring) combined:




This subtree is now balanced and has the same height as before insertion. So, the height-balance property is restored *globally*. (We checked the double rotation case, the other case is similar)

# Complexity of insertion

The standard BST insertion in AVL trees is $O(\log n)$.

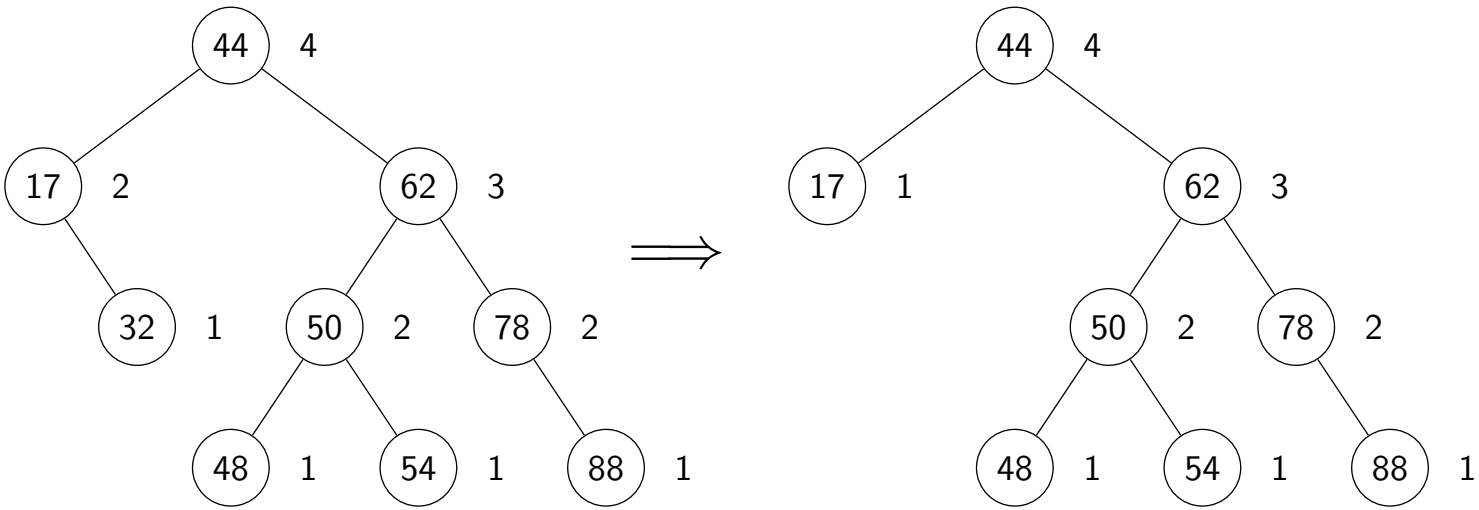
Fix-up procedure:

- travel up the tree from the inserted node (looking for an unbalanced node) — $O(\log n)$ time.
- do trinode restructuring at most once — $O(1)$ time.

All in all, the worst-case running time is $O(\log n)$.

# Deletion

Standard BST deletion can violate the height-balance property.




So again we might need to fix it up after deletion.

# Post-deletion fix-up

If $p$ is the parent of the deleted node (which is not always the node that contained deleted data!), there may be unbalanced nodes on the path from $p$ to the root.

Can use trinode restructuring to re-balance:

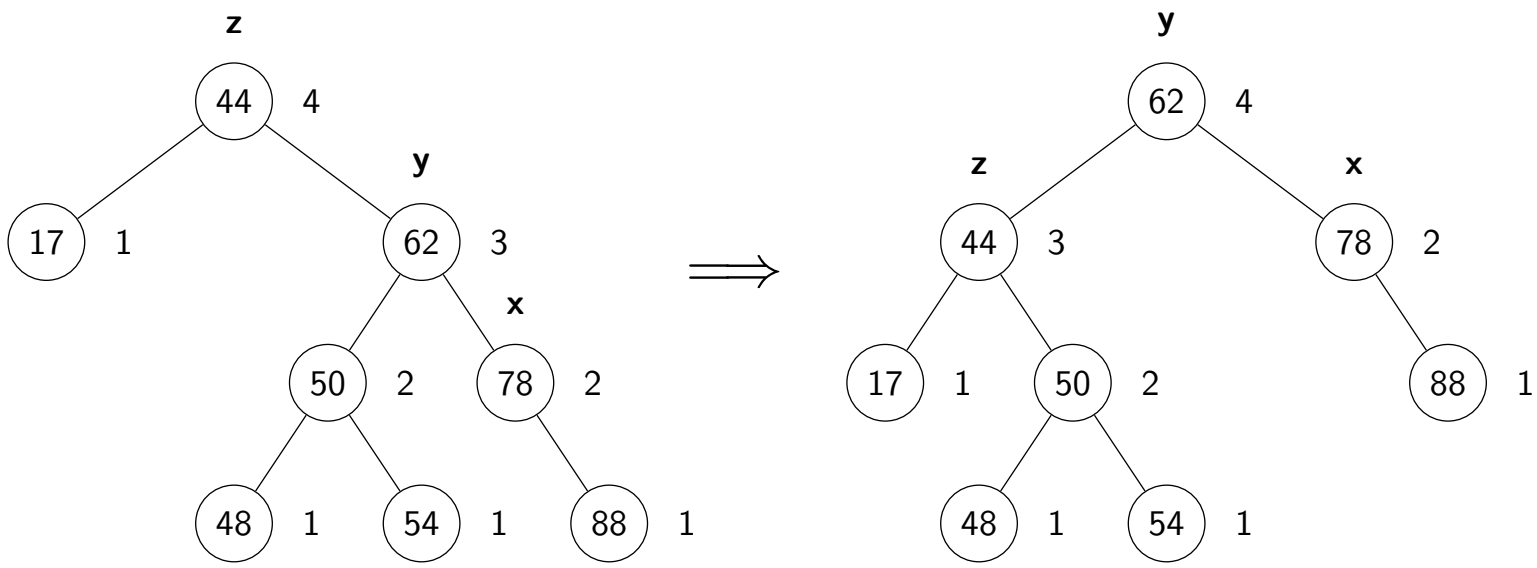
- Let $z$ be the first unbalanced node on the path
- Let $y$ be the taller child of $z$ (i.e. the one not on the path)
- Let $x$ be the taller child of $y$ (or, if children of $y$ have the same height, on the same left/right side as $y$)
- Use trinode restructuring on $z - y - x$.

This will fix up the subtree of $z$, but may *reduce* its height.

Must **keep going up** the tree and do restructuring whenever we meet an unbalanced node.

# Example

## Restoring height balance after deletion



# Complexity of deletion

The standard BST deletion in AVL trees is $O(\log n)$.

The fix-up procedure uses $O(1)$ time per level

Done at most once per level, and the height is $O(\log n)$.

All in all, the worst-case running time is $O(\log n)$.

The post-processing for insertion and deletion can be unified:

- Both trace an upward path from some node
- Both use trinode restructuring when an unbalanced node is found

"Early stop" condition: Can stop the post-processing when

- reach an ancestor whose height didn't change, or
- trinode restructuring of a subtree results in the same height as before insertion/deletion

To detect this, can store the "old" height of each node

# Red-black trees (briefly)

A red-black tree is an ordinary BST with **one extra bit** of storage per node: its **colour**, red or black

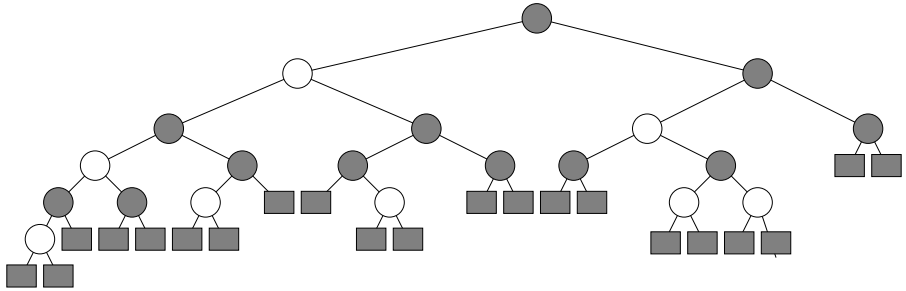
Constraining how nodes can be coloured results in (approximate) balancedness of tree

Assumption: if either parent, left, or right does not exist, pointer to Null

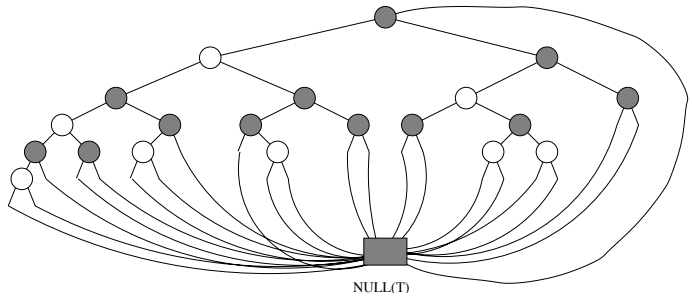
Regard Nulls as external nodes (leaves) of tree, thus normal nodes are internal

A BST is a **red-black tree** if

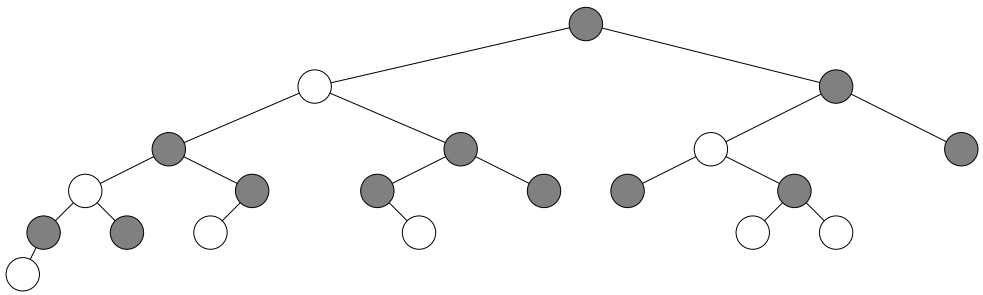
1. every node is either red or black
2. the root is black
3. every leaf (Null) is black
4. red nodes have black children
5. for each node, all paths from the node to descendant leaves contain same number of black nodes



red-black tree, according to definition




replace NULLs with single "sentinel" NULL($T$)



normal drawing style (no NULLleaves or sentinel), but sentinel **is** there

Most important property:

**Lemma**

*A red-black tree with $n$ internal nodes has height at most $2\log(n+1)$.*

Proof omitted (idea: have same number of black nodes on each path from root to any leaf, so it's black-balanced, plus can't have too many red nodes).

As with AVL trees,

- all operations are $O(\log n)$ time
- insertion and deletion require post-processing (fix-up) – can be done with recolouring and rotations

# Insertion

Suppose we insert new node $z$

First, **ordinary BST insertion** of $z$ according to key value (walk down from root, open new leaf)

Then, colour $z$ **red**

May have **violated red-black property** (e.g., $z$'s parent is red)

Call a fix-up procedure to restore red-black property

# What can have gone wrong?

Reminder:

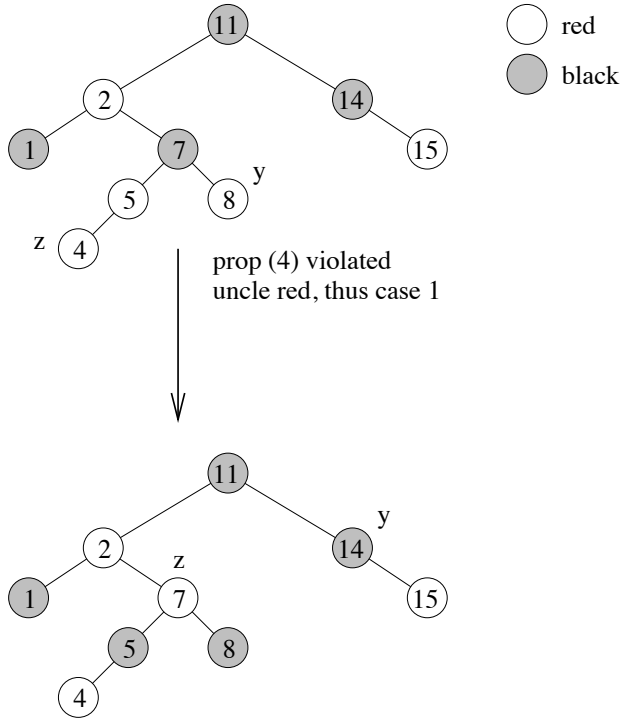
1. every node is either red or black
2. the root is black
3. every leaf (NULL) is black
4. red nodes have black children
5. for all nodes, all paths from node to descendant leaves contain same # black nodes

**(1)** and **(3)** certainly still hold

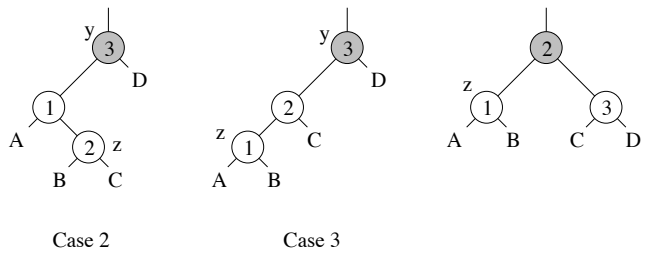
**(5)** as well, since $z$ replaces (black) sentinel, and $z$ is red with sentinel as children

**(2)** and **(4)** may be violated (because $z$ is red);
(2) if $z$ is root, and (4) if $z$'s parent is red

# Several cases, with fix-up procedures that look like so:




or so:



# AVL vs. red-black

Some trade-offs between AVL trees and red-black trees:

To control heights, AVL either use more space than red-black (to store heights in nodes), but still $O(n)$ space, or use extra time to compute heights (if not stored) - how much extra?

AVL trees are better for lookup-intensive tasks - because the balance condition is stricter

Red-black trees are more efficient for insertion/deletion - because the balance condition is more relaxed (easier to fix up).

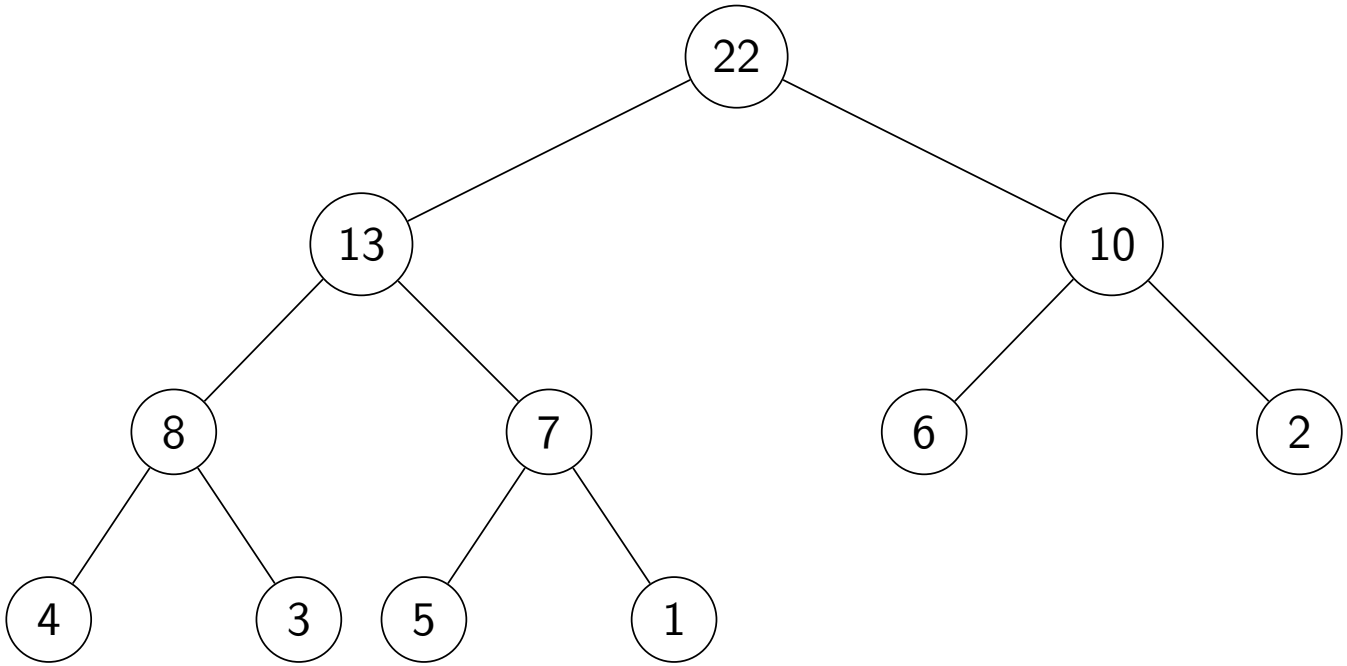
# Heaps

BSTs and family are "proper" tree structures

Heaps are trees as well, but typically assumed to be stored in a flat array

- each tree node corresponds to an element of the array
- the tree is complete except perhaps the lowest level, filled left-to-right
- heap property: for all nodes $v$ in the tree, `v.parent.data >= v.data` (assuming we have a node class like for BSTs)
- This is a max-heap (for min-heaps, `v.parent.data =< v.data`)

# Example




Very much **not** a BST!

Heap represented as an array A has two attributes:

1. Length(A) – the size of the array
2. HeapSize(A) – the size of the heap stored within the array

Clearly, need Length(A) $\geq$ HeapSize(A) at all times

Heap property now: `A[v.parent.index] >= A[v.index]`

# What about them array indices, then?

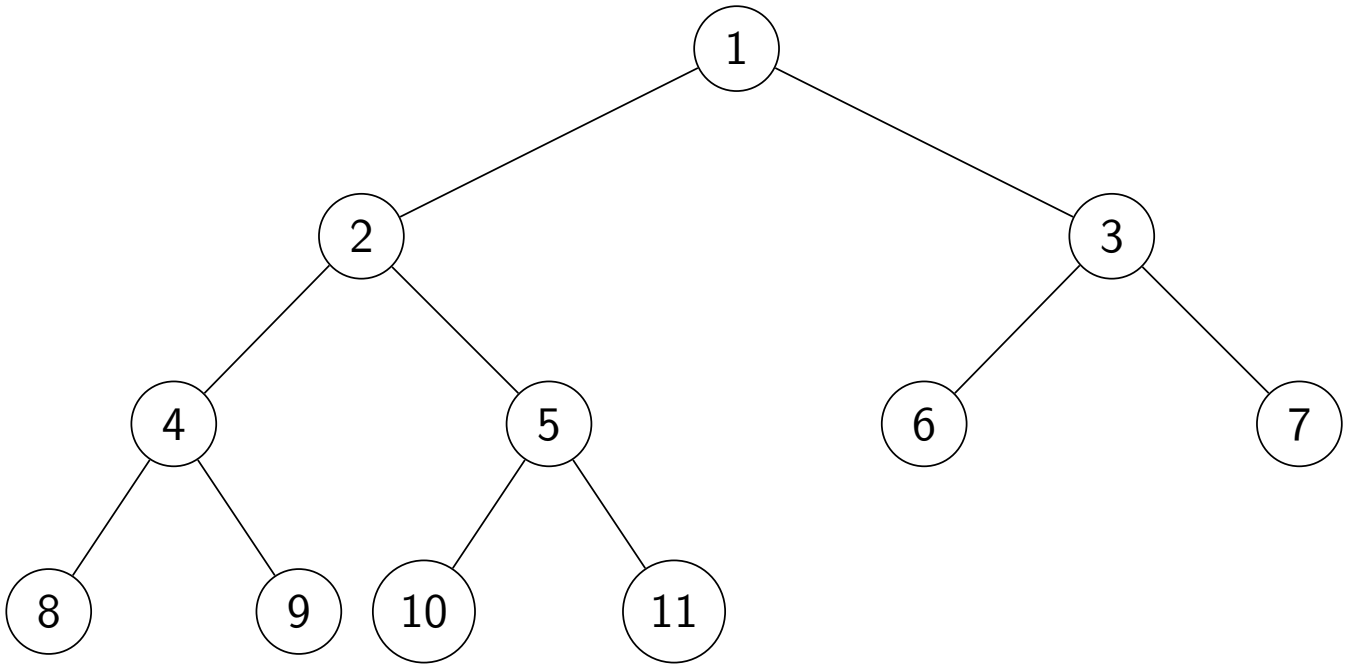
Assume we start counting at position 1

Then:

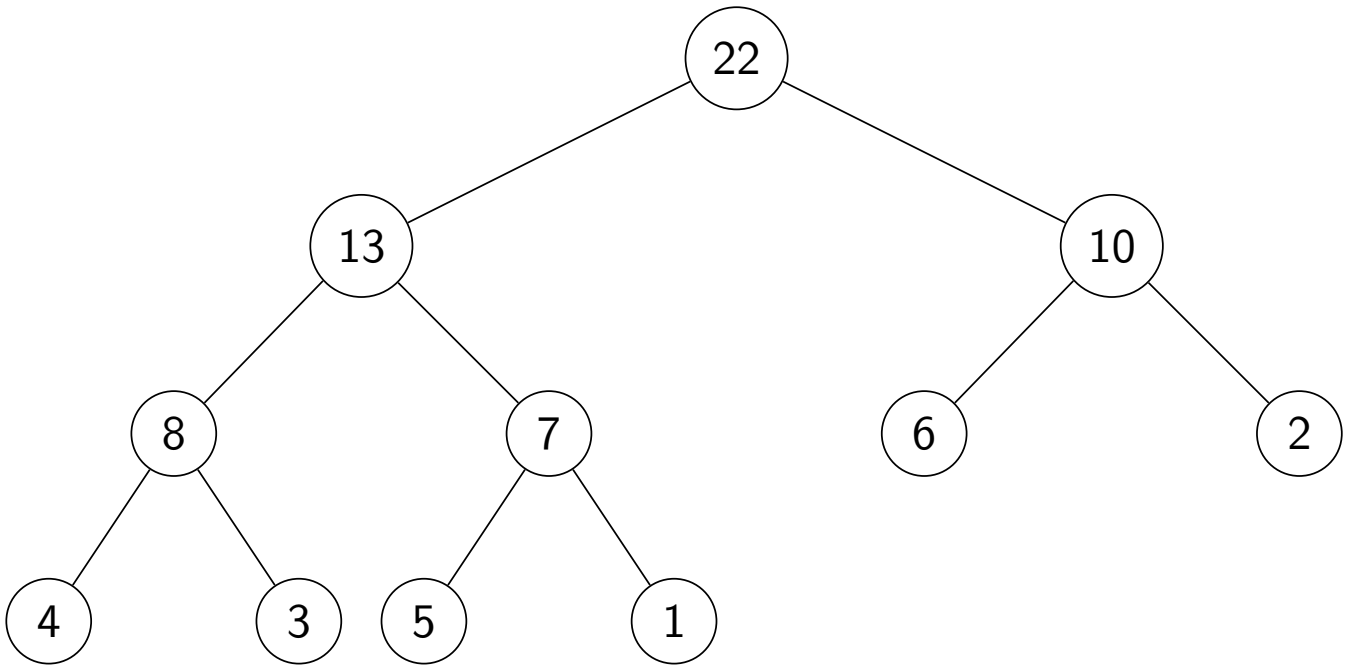
1. the root is in A[1]
2. parent(i) $=$ A[i/2] – integer division, rounds down
3. left(i) $=$ A[2i]
4. right(i) $=$ A[2i+1]

So, `v.left.index = 2*(v.index)`, and so forth

# Example: now with array indices



# Example: back to stored data




The corresponding array: A=[22, 13,10, 8,7,6,2, 4,3,5,1]

# But. . . why???

Very good data structure for **priority queues** and for **sorting**

Given sequence of objects with different priorities, want to deal with highest priority items first

So as to support priority queues we need to extract maximum element from collection, quickly:

## HeapExtractMax(A)

```
1  ret = A[1] // biggest element (highest priority)
2  A[1] = A[HeapSize(A)]
3  HeapSize(A) = HeapSize(A)-1
4  Heapify(A,1,HeapSize(A))
5  return ret
```

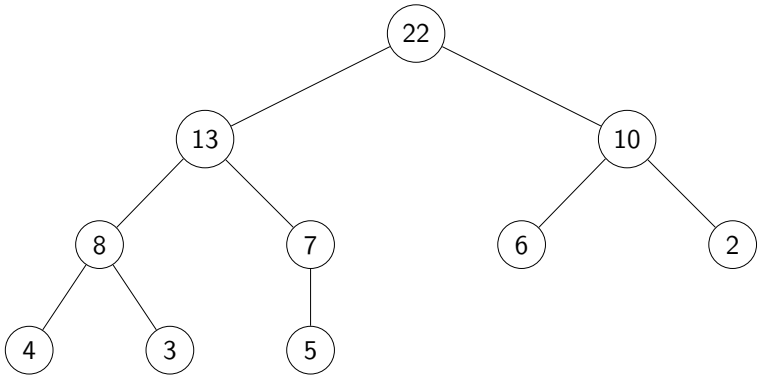
Running time: $O(1)$ plus time for heapification

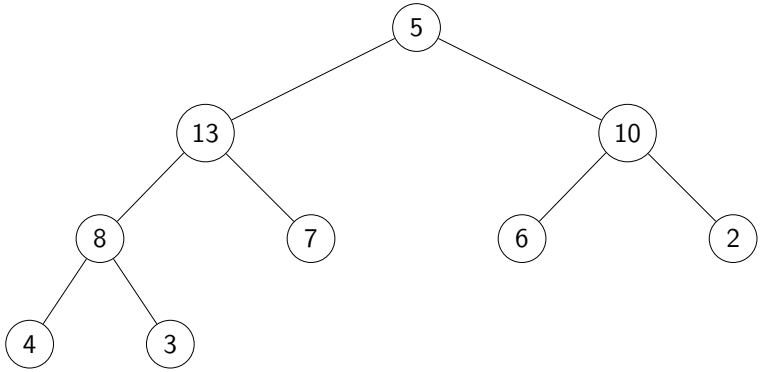
# Example

## HeapExtractMax(A)

```
1  ret = A[1]
2  A[1] = A[HeapSize(A)]
3  HeapSize(A) = HeapSize(A)-1
4  Heapify(A,1,HeapSize(A))
5  return ret
```

After extraction of maximum, but before heapification



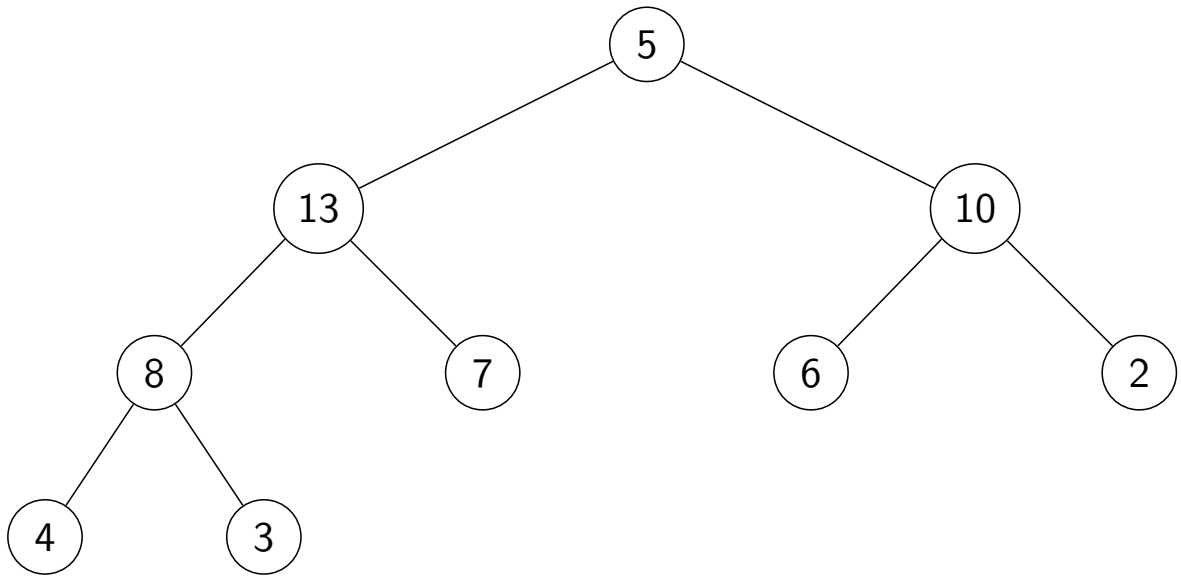



# What's to come?

1. Heapify: maintaining heap property
2. BuildHeap: initially building a heap
3. HeapSort: sorting using a heap

# Heapify

Given something like this, how to turn it into a proper heap again?



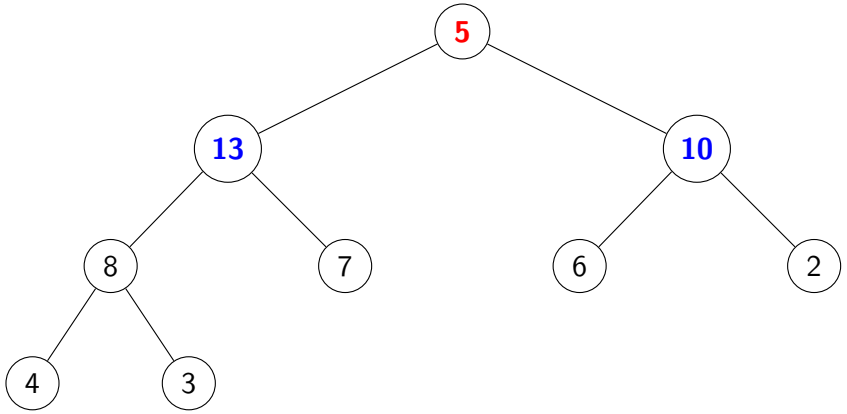

Any ideas?

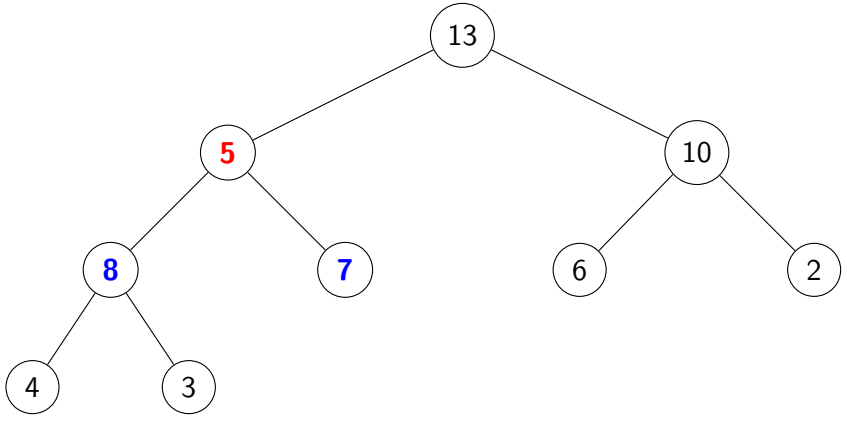
Idea:

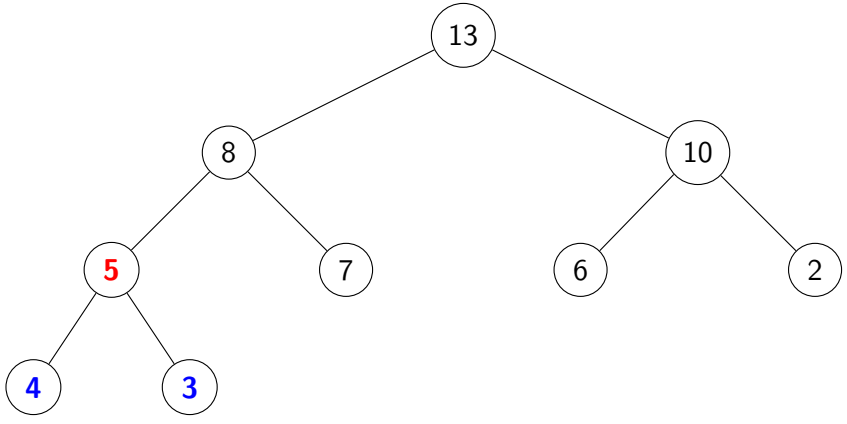
1. starting at the root, identify largest of current node $v$ and its children
2. suppose largest element is in $w$
3. if $w \neq v$
   1. swap A[w] and A[v]
   2. recurse into w (contains now what root contained previously)

# Heapify (A, v, n)

```
// n is heap size
// find largest among v and 2v (left child)
largest = v
if 2v <= n and A[2v]>A[v] then largest = 2v

// find largest among winner above and
// 2v+1 (right child)
if 2v+1 <= n and A[2v+1]>A[largest] then
  largest = 2v+1

if largest != v then
  swap A[v], A[largest]
  Heapify (A, largest, n)
```



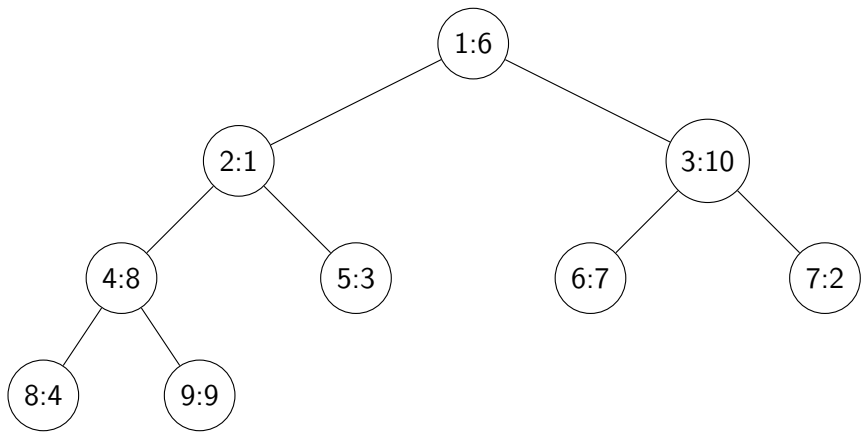




Running time: linear in height of tree, $O(\log n)$

# BuildHeap

Task: given array $A$ with $n$ arbitrary numbers stored in it, convert $A$ into a heap

Example: A=[6,1,10,8,3,7,2,4,9] gives



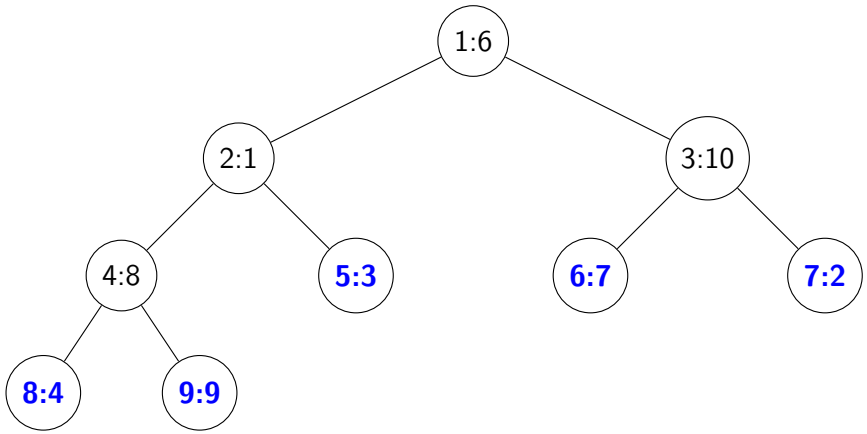

Clearly a lot of work to be done here!

**Note**: leaves are fine!

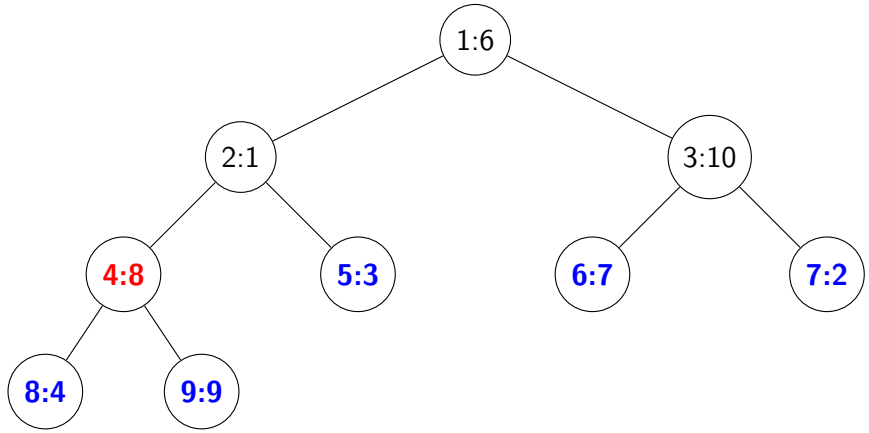
**Idea:** start from leaves and build up from there

## BuildHeap(A,n)

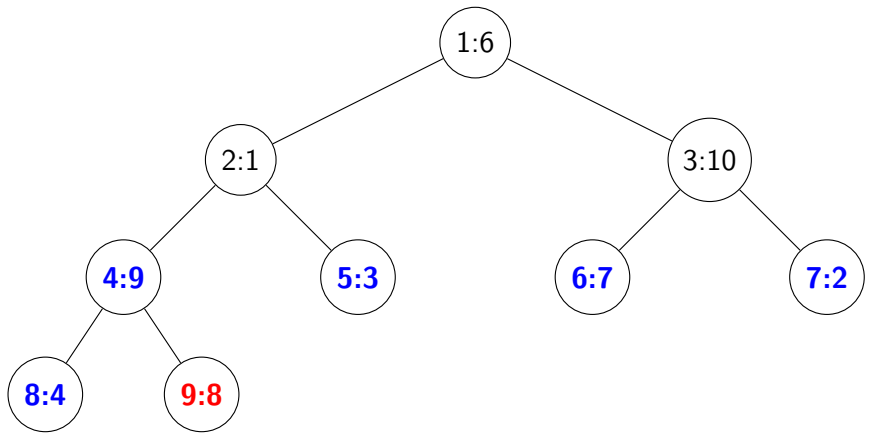
```
for i=n downto 1 do
  Heapify(A,i,n)
endfor
```



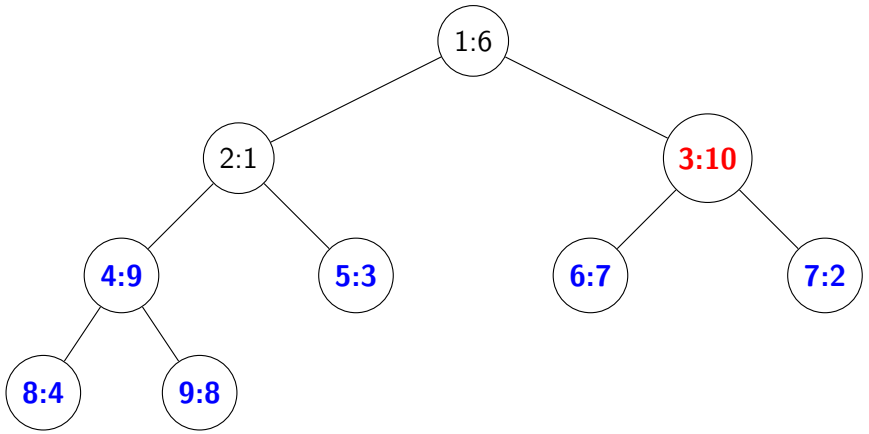

Heapify(A,9,9) exits – leaf
Heapify(A,8,9) exits – leaf
Heapify(A,7,9) exits – leaf
Heapify(A,6,9) exits – leaf
Heapify(A,5,9) exits – leaf



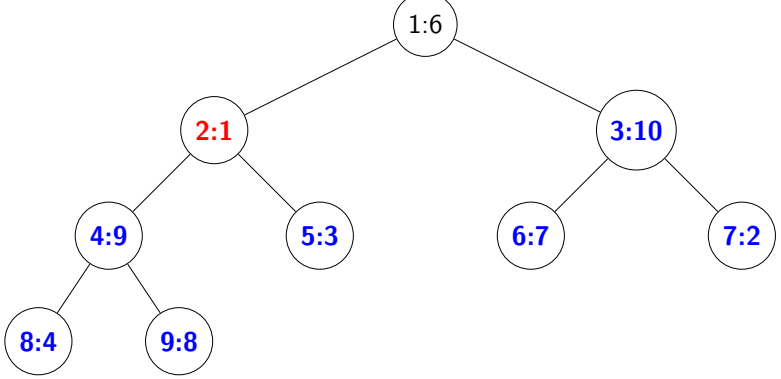

Heapify(A,4,9) puts largest of 8,4,9 into A[4]



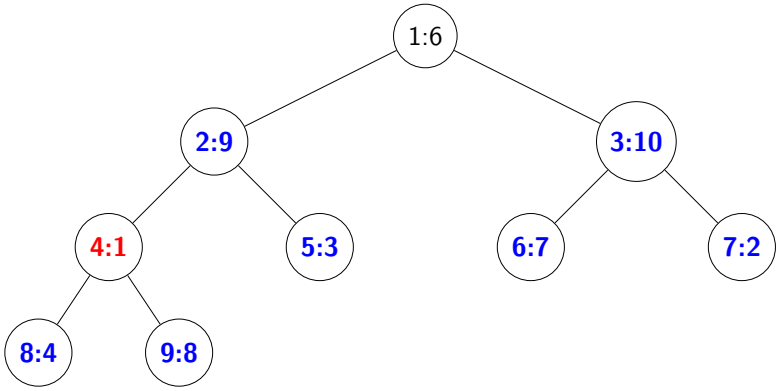

and recursively calls Heapify on A[9] (here: exits)



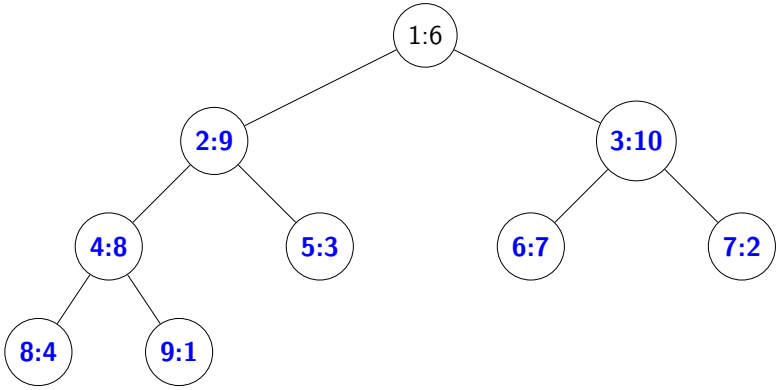

Heapify(A,3,9) puts largest of 10,7,2 into A[3] (here: exits)

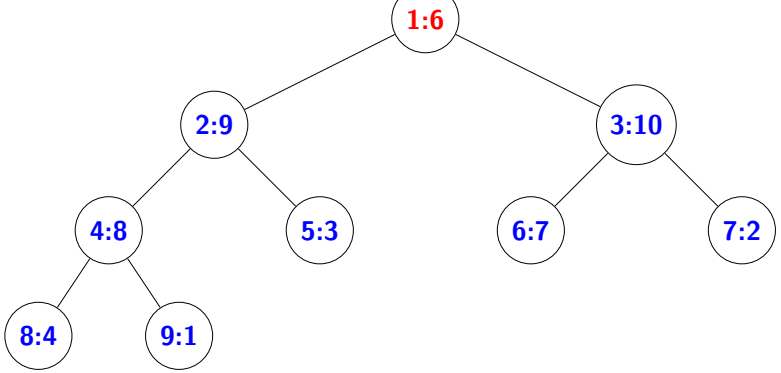



Heapify(A,2,9) puts largest of 1,9,3 into A[2]

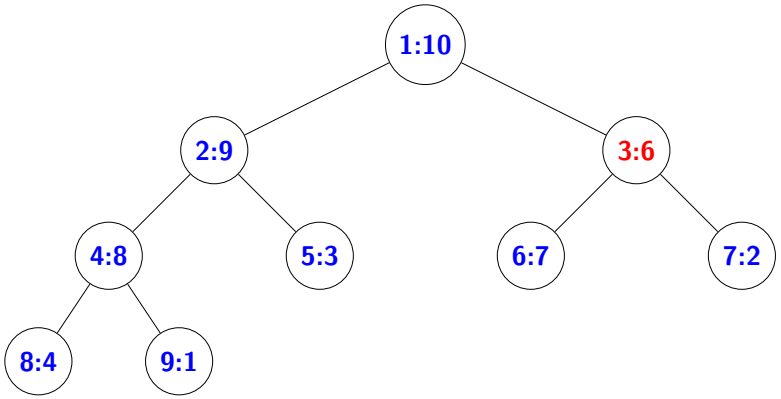



and recurses into A[4] (here: exits)

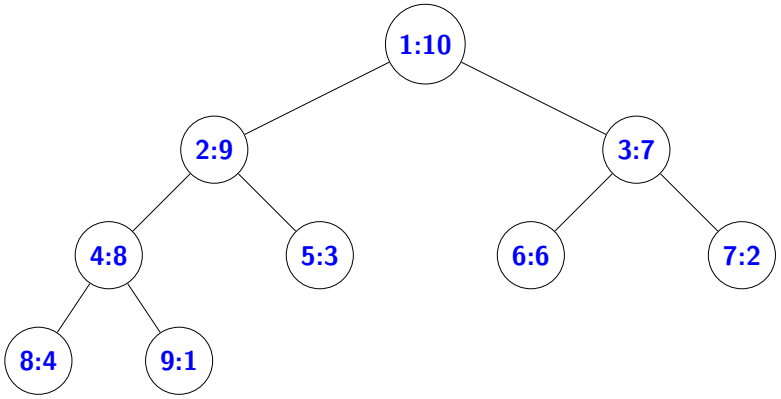





Heapify(A,1,9) puts largest of 6,9,10 into A[1]




and recurses into A[3]



# Running time BuildHeap

Loop with $n$ iterations, each call to Heapify takes $O(\log n)$ time

Implies overall bound $O(n \log n)$

Certainly true, but so is $O(2^{2^n})$

Better:

- if *height* of a node is counting upwards from lowest leaves with height of leaf on lowest level $=1$ and height(root)$=\log n$ then in $n$-element heap there are $\leq \frac{n}{2^h}$ nodes of height $h$
- time for Heapify when called on node of height $h$ is $O(h)$
- cost:

$$T(n) = \sum_{h=1}^{\log n} (\#\text{ nodes at height } h) \cdot O(h)$$

$$\leq \sum_{h=1}^{\log n} \frac{n}{2^h} \cdot O(h) \;=\; O\left(\sum_{h=1}^{\log n} \frac{n}{2^h} \cdot h\right) = O\left(n \cdot \sum_{h=1}^{\log n} \frac{h}{2^h}\right)$$

Well known: for $x \in (0,1)$,

$$\sum_{i=0}^{\infty} i \cdot x^i = \frac{x}{(1-x)^2}$$

Use this ($x = 1/2$):

$$\sum_{h=1}^{\log n} \frac{h}{2^h} \leq \sum_{h=0}^{\infty} h \cdot \left(\frac{1}{2}\right)^h$$

$$= \frac{1/2}{(1-1/2)^2} = \frac{1/2}{1/4} = 2$$

and hence

$$T(n) = O\left(n \cdot \sum_{h=1}^{\log n} \frac{h}{2^h}\right) = O(n)$$

That is, can turn any old array into heap in time $O(n)$

# Finally: HeapSort

Trivial:

- call BuildHeap on unsorted data, and
- repeatedly call HeapExtractMin until empty

Running time: $O(n) + n \cdot O(\log n) = O(n \log n)$

This version is incorrect. Why? – Can't find min in a max-heap quickly enough

# Proper HeapSort

## HeapSort(A)

```
BuildHeap(A, Length(A))
for i = Length(A) downto 2 do
  swap A[i] and A[1]
  HeapSize(A) = HeapSize(A)-1
  Heapify(A, 1, HeapSize(A))
endfor
```

Running time: $O(n) + n \cdot O(\log n) = O(n \log n)$

# Sorting Algorithms

We've seen

| Algorithm | Running Time |
|---|---|
| SelectionSort | $O(n^2)$ |
| InsertionSort | $O(n^2)$ |
| MergeSort | $O(n \log n)$ |
| QuickSort | $O(n^2)$ |
| BucketSort | $O(n + K)$ |
| RadixSort | $O(n \log K)$ |
| HeapSort | $O(n \log n)$ |

Trade-offs involving, e.g., efficiency, memory usage, stability

Most programming languages have a built-in sorting function: highly optimized combination of various algorithms

# Lower bounds

Plan:

Lower bounds for sorting – decision tree argument

Lower bounds for selection – adversary argument

Lower bounds for merging - the input changing argument

# Sorting and decision trees

# Lower bound for sorting

We've seen a few **comparison-based** sorting algorithms, e.g.:

**MergeSort, HeapSort**: upper bound $O(n \log n)$

**SelectionSort, QuickSort:** upper bound $O(n^2)$

We discussed earlier:

**Theorem**

*Any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons in the worst case.*

# Decision trees

A decision tree is a **full binary tree** (every node has either zero or two children)

It represents comparisons between elements performed by particular algorithm run on particular (size of) input

Only **comparisons** are relevant, everything else is ignored

**Internal nodes** are labelled $i : j$ for $1 \leq i, j \leq n$, meaning elements $i$ and $j$ are compared (indices w.r.t. original order)

**Downward-edges** are labelled "$\leq$" or $>$, depending on the outcome of the comparisons

**Leaves** are labelled with some permutation of $\{1, \ldots, n\}$

A **branch** from root to leaf describes sequence of comparisons (nodes and edges) and ends in some resulting sequence (leaf)

We have at least $n!$ leaves – at least one for each outcome

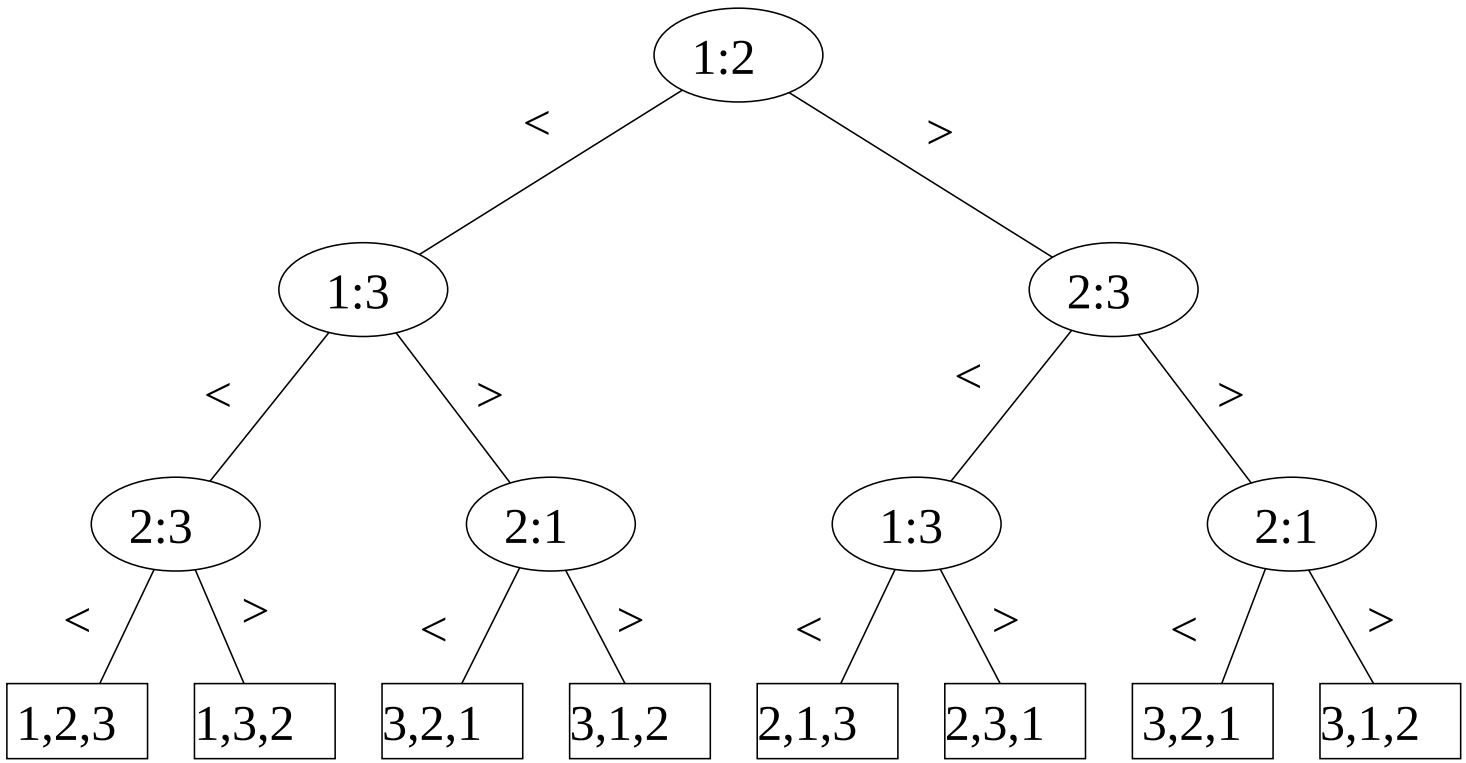
**Example:** SelectionSort

Reminder:

**SelectionSort** $(a_1, \ldots, a_n \in \mathbb{R}, n \geq 2)$

```
1:  for i = 1 to n − 1 do
2:      elem = a_i
3:      pos = i
4:      for j = i + 1 to n do
5:          if a_j < elem then
6:              elem = a_j
7:              pos = j
8:          end if
9:      end for
10:     swap a_i and a_pos
11: end for
```

# Decision tree for SelectionSort on 3-element input (hopefully. . . :-)



Any correct sorting algorithm must be able to produce each permutation of input (or: must sort any permutation of e.g. $\{1, \ldots, n\}$)

$\Longrightarrow$ necessary condition is that each of the $n!$ permutations must appear as leaf of decision tree

**Lower bound for worst case**

Length of longest path from root of decision tree to any leaf (height) represents worst case number of comparisons for given value of $n$

For given $n$, lower bound on heights of **all** decision trees where each permutation appears as leaf is thus lower bound on running time of **any** comparison based sort algorithm

## Theorem

*Any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons in the worst case.*

## Proof.

- Sufficient to determine minimum height of a decision tree in which each permutation appears as leaf
- Consider decision tree of height $h$ with $\ell$ leaves corresponding to a comparison sort on $n$ elements
- Each of the $n!$ permutations of input appears as some leaf: $\ell \geq n!$
- Binary tree of height $h$ has at most $2^h$ leaves: $\ell \leq 2^h$
- Together: $n! \leq \ell \leq 2^h$ and therefore $2^h \geq n!$
- Take logs: $h \geq \log(n!) = \Omega(n \log n)$

□

# Selection and adversaries

# Lower bounds via adversaries

Adversary:

- is a second algorithm intercepting access to input
- gives answers so that there's always a consistent input

  i.e. adversary can never be caught cheating
- ensures that original algorithm has **not enough info** to make a decision, for **as long as possible**

  alternaive view: dynamically constructs a bad input for it
- doesn't know what original algorithm will do in the future

  i.e. must work for **any original algorithm**

To get a good lower bound, design a good adversary

# Finding max

Goal: find max element in an unsorted array

In an array of size $n$, can do this with $n-1$ comparisons

Same set-up as before: Only **comparisons** are relevant

## Theorem

*Any comparison-based algorithm for this problem requires at least $n-1$ comparisons in the worst case.*

## Proof.

After $\leq n-2$ comparisons, $\geq 2$ elements never lost (a comp)
$\Rightarrow$ Adv can make any of them max and be consistent
$\Rightarrow$ not enough info for Alg to make a decision.
Hence Alg needs to make at least $n-1$ comparisons. $\square$

# Designing Adversary's strategy

The only relevant info for Algorithm is

*how many elements lost* $\geq 1$ *comparison.*

As soon as $n-1$ elements lost (at least once), the algorithm can make a decision - max is the one that has never lost.

Assume each index $i$ has a status, N (never lost) or L (lost), initially all indices have status N.

After each comparison, there can be status changes N $\rightarrow$ L

Adversary's goal: delay changes N $\rightarrow$ L as much as possible

# Designing Adversary's strategy

Adversary's strategy can be represented as a status update table (below), with the number of bits of new relevant info.

When Algorithm asks to compare $i : j$, reply as follows:

| Status for $i, j$ | Adv Reply | New Status | New Info |
|---|---|---|---|
| N;N | $a_i > a_j$ | N;L | 1 |
| N;L | $a_i > a_j$ | No Change | 0 |
| L,N | $a_i < a_j$ | No Change | 0 |
| L;L | Consistent | No Change | 0 |

One comparsion gives $\leq 1$ bit of new relevant info (i.e. new L).

In total, Algorithm needs $n - 1$ bits of info to make a decision.

Hence, $\geq n - 1$ comparisons are necessary.

# Finding 2nd largest

Goal: find the 2nd largest element in an unsorted array

**Theorem**

*Any comparison-based algorithm for this problem requires at least* $n + \lceil \log_2 n \rceil - 2$ *comparisons in the worst case.*

Observations:

1. Any correct algorithm must also determine the max.
   $n-1$ elements must lose $\geq 1$ comparisons. (What if not?)
2. The 2nd largest must lose to max directly.
3. If max had direct comparisons with $m$ elements, then $m-1$ of these elements must lose $\geq 2$ comparisons.
4. Hence, at least $n-1+m-1 = n+m-2$ comparisons are required.

# Adversary's strategy

Enough to prove the following:

**Lemma**

*Adversary can force* $\lceil \log_2 n \rceil$ *comparisons involving max.*

Proof: Adversary assigns weight $w_i$ to each input element $a_i$.

Initially all $w_i = 1$, then update using these rules:

| Weights | Adv Reply | Update |
|---|---|---|
| $w_i > w_j$ | $a_i > a_j$ | $w_i = w_i + w_j, w_j = 0$ |
| $w_i < w_j$ | $a_i < a_j$ | $w_j = w_i + w_j, w_i = 0$ |
| $w_i = w_j > 0$ | $a_i > a_j$ | $w_i = w_i + w_j, w_j = 0$ |
| $w_i = w_j = 0$ | Consistent | No change |

# Analysis

- $w_i = 0 \Leftrightarrow i$ lost $\geq 1$ comparison
- Weight of an element at most doubles after a comparison
- If max is involved in $m$ comparisons, its weight is $\leq 2^m$
- In the end, max accumulates all the weight, so $n \leq 2^m$.
- Taking logs, $\log_2 n \leq m$, as required.

So, Adversary can force $\lceil \log_2 n \rceil$ comparions with max,

which proves the lemma and hence the theorem.

# Finding 2nd largest: Algorithm

**Theorem**

*There is a comparison-based algorithm for finding 2nd largest that requires $n + \lceil \log_2 n \rceil - 2$ comparisons in the worst case.*

Here's an algorithm that matches our lower bound.

Consider a knock-out tournament: players = array elements

Think a balanced binary tree, leaves = array elements.
With an array of size $n$, the height of the tree is $\lceil \log_2 n \rceil$.

Play the tournament to find max: this takes $n - 1$ comparisons

Consider the $\lceil \log_2 n \rceil$ elements who lost directly to max.
2nd largest overall = largest among those.
Find it with $\lceil \log_2 n \rceil - 1$ comparisons.

Altogether, this uses $n + \lceil \log_2 n \rceil - 2$ comparsions.

# Finding $k$th largest

Let $V(n,k)$ be the number of comparisons necessary and sufficient to find $k$th largest in any array of size $n$.

The exact value of $V(n,k)$ is known for

$$k = 1, 2, n-1, n.$$

For other values of $k$, only some bounds are known, e.g. :

**Theorem**

*For all* $1 \leq k \leq n$, *it holds that* $V(n,k) \geq n - k + \lceil \log_2 \binom{n}{k-1} \rceil$.

# Merging

# Merging sorted arrays

Consider the following **merging** problem: given two sorted arrays with $n$ elements each, merge them into one sorted array.

Still use the comparison model: only comparisons are counted.

**Theorem**

- *There is a comparison-based algorithm solving the merging problem in $2n-1$ comparisons in the worst case.*
- *Any correct comparison-based algorithm for the merging problem makes at least $2n-1$ comparisons in the worst case.*

Proof: For the upper bound (1st item), merge as in MergeSort.

# Proof of the lower bound

- For contradiction, let *Alg* be an algorithm correctly solving our problem with $\leq 2n-2$ comparisons in the worst case.
- Consider a specific input, the following arrays $X$ and $Y$:

| | | | | | |
|---|---|---|---|---|---|
| $x_1 = 1$ | $x_2 = 3$ | $\cdots$ | $x_i = 2i-1$ | $\ldots$ | $x_n = 2n-1$ |
| $y_1 = 2$ | $y_2 = 4$ | $\ldots$ | $y_i = 2i$ | $\ldots$ | $y_n = 2n$ |

- Consider the $(2n-1)$ comparisons $x_i : y_i$ for $i = 1, \ldots, n$ and the comparisons $x_{i+1} : y_i$ for $i = 1, \ldots, n-1$.
- When *Alg* runs on the above input, it does not make at least one of these comparisons, say $x_j : y_j$ for some $j$.
- Now change the input: choose one $j$ as above and swap the values of $x_j$ and $y_j$, i.e. set $x_j = 2j$ and $y_j = 2j-1$.
- If we run *Alg* on the modified input, it will not notice the change - from its point of view, all comparisons come out the same, so it will not compare $x_j : y_j$.
- So *Alg* will output the same sequence of $x$'s and $y$'s as for the original input, which would now be incorrect.