Lecture 07: Sorting I

928023

COMP526: Efficient Algorithms

Updated: October 24, 2024

Will Rosenbaum
University of Liverpool

Announcements

1. Third Quiz, due Friday
   - Similar format to before
   - Covers fundamental data structures (Lectures 4–6)
   - Quiz is **closed resource**
     - No books, notes, internet, etc.
     - Do not discuss until after submission deadline (Friday night, after midnight)
2. Programming Assignment (Draft) Posted $\longrightarrow$ *course website*
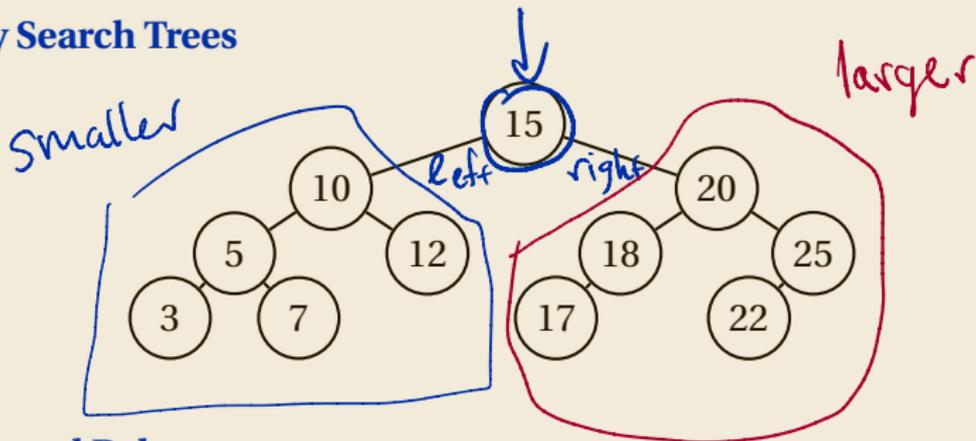   - Due Wednesday, 13 November
3. Attendance Code:

928023

Meeting Goals

- Finish up balanced binary trees
- Discuss the sorting task
- Introduce HEAPSORT
- Discuss Divide and Conquer approaches to sorting
    - MERGESORT
    - QUICKSORT

# AVL Trees

From Last Time

Binary Search Trees

Height and Balance

- **height** of v = max distance to a descendent leaf
- T is **height balanced** if for every v, the heights of v's children differ by at most 1
- Properties of height balanced trees
    - height h satisfies $h \le 2 \log n$
    - CONTAINS, ADD, REMOVE run in $O(\log n)$ time

Question. How can we efficiently maintain height balance for any sequence of operations?
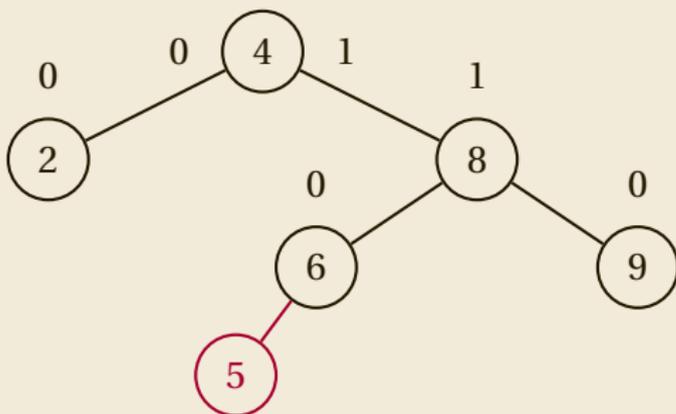
Creating Imbalance

A Minimal Working Example (MWE) **balanced**
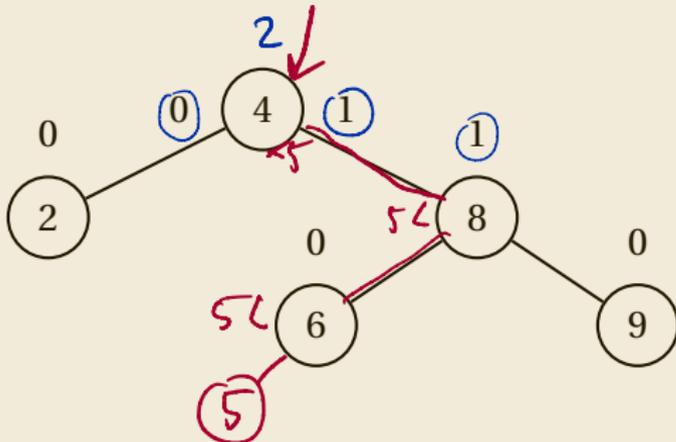
Question. What happens when we
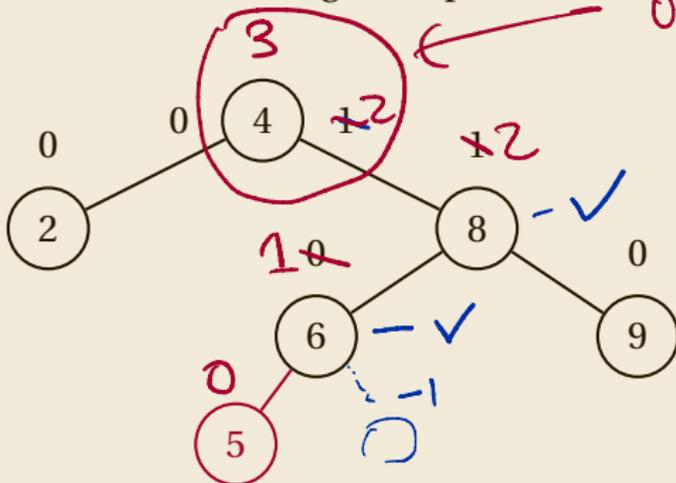ADD(5)?

Creating Imbalance

A Minimal Working Example (MWE) **unbalanced**

Question. What happens when we
ADD(5)?

Creating Imbalance

A Minimal Working Example (MWE)

only unbalanced vertex
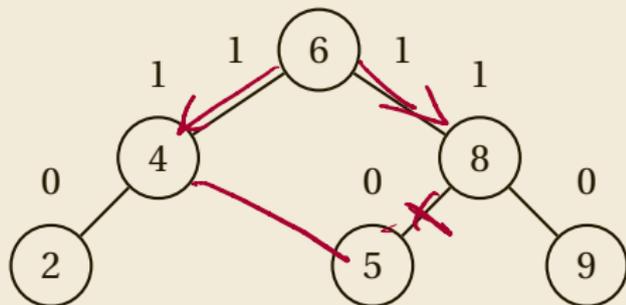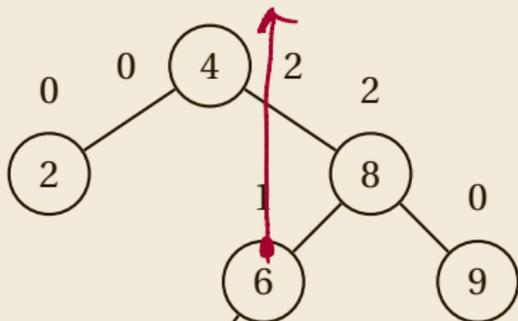
PollEverywhere

Which vertices are unbalanced?

pollev.com/comp526

Question. What happens when we ADD(5)?

Fixing Imbalance

Fix Me!!

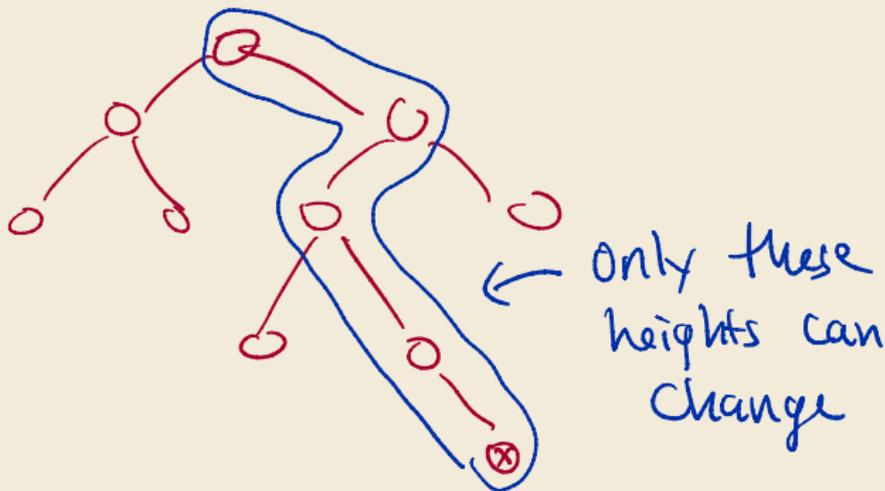**General Strategy.** Find the "lowest" unbalanced vertex, and "pull up" its lower child.
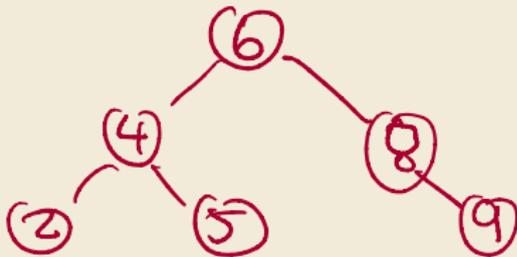
Unbalanced Observations

Suppose T was balanced before ADD(x) and unbalanced after ADD(x). Then:

1. ADD(x) can only change the height/balance of x's **ancestors**.
2. The height of any vertex can can only increase by one as the result of ADD(x).

← only these heights can change

Unbalanced Observations

Suppose T was balanced before ADD(x) and unbalanced after ADD(x). Then:

1. ADD(x) can only change the height/balance of x's **ancestors**.
2. The height of any vertex can can only increase by one as the result of ADD(x).

This means:

- We only need to check x's ancestors for imbalance after ADD(x).
- We only need to correct an imbalance of 2 to restore balance in the tree after ADD(x).

Rotations

$\Longrightarrow$ right rotation at y $\Longrightarrow$
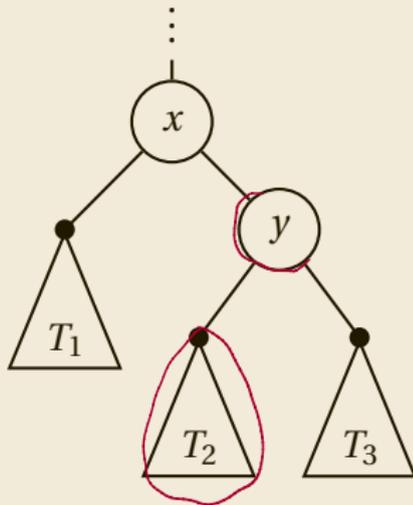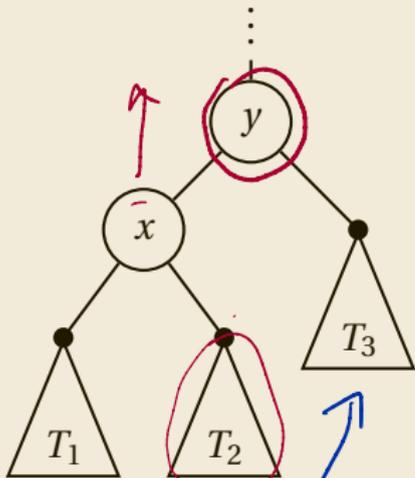
$\Longleftarrow$ left rotation at x $\Longleftarrow$

subtrees

Rotations

$\Longrightarrow$ right rotation at $y \Longrightarrow$

$\Longleftarrow$ left rotation at $x \Longleftarrow$

Note: given x or y rotation can be implemented in $\Theta(1)$ time
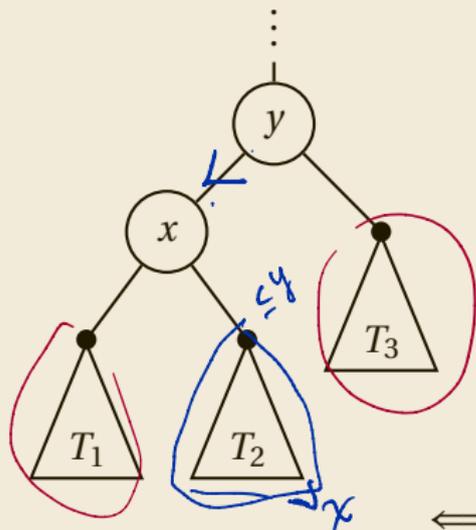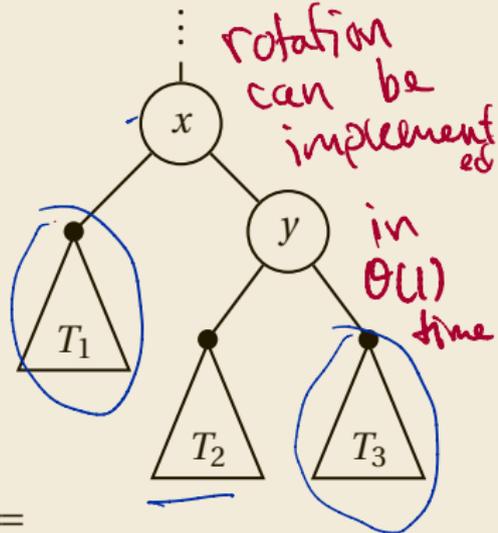
Main Observation. If T is a BST, then it remains a BST after any rotation.

Restoring Balance After Add

Suppose T was balanced before $\text{ADD}(w)$ and is unbalanced after the operation. Then define

- z is w's closest unbalanced **ancestor**
- y is z's child towards w
- x is y's child towards w
  - Why do these vertices exist?
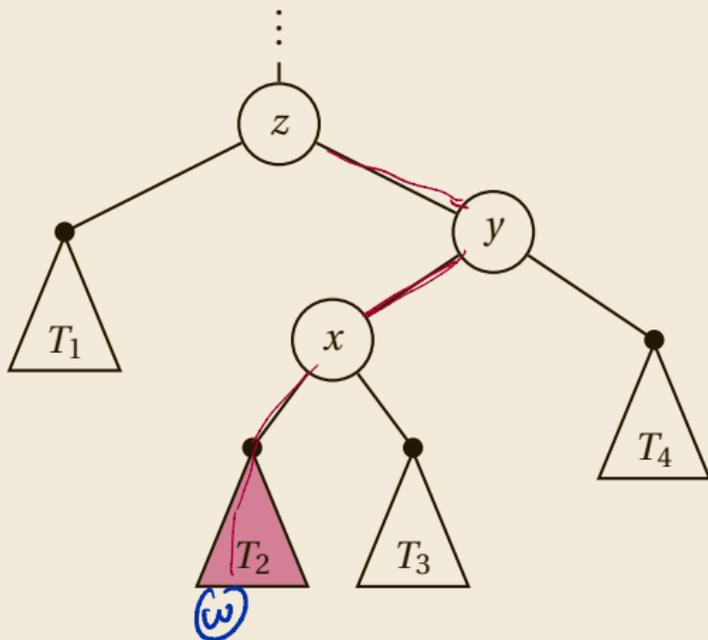
PollEverywhere

If z had height h before ADD(w), what are the heights of z, T_1, y, and x afterward?

pollev.com/comp526

height of root of T_1 (= z's child)
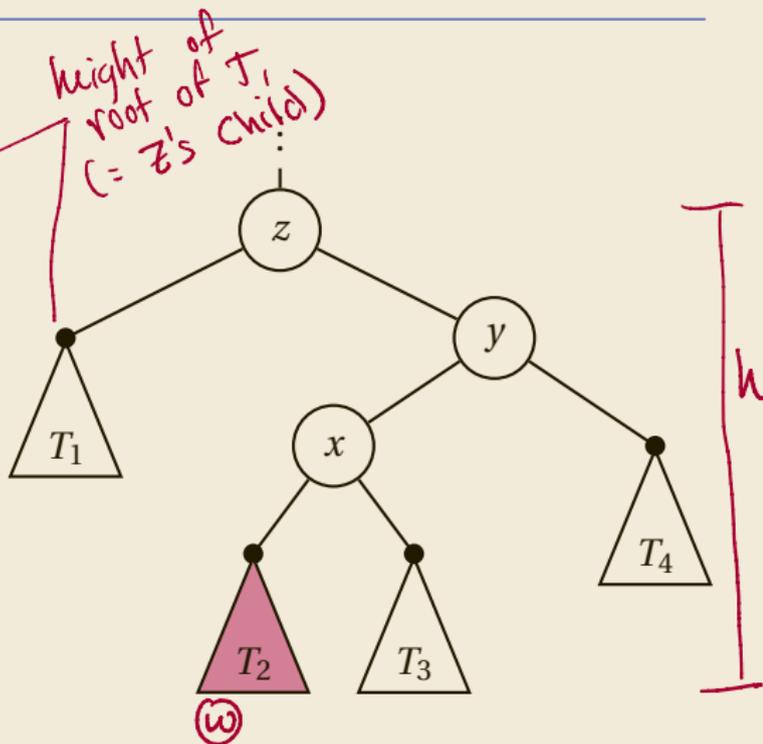
Clue: height of z changed increased by 1

Heights after $\text{ADD}(w)$
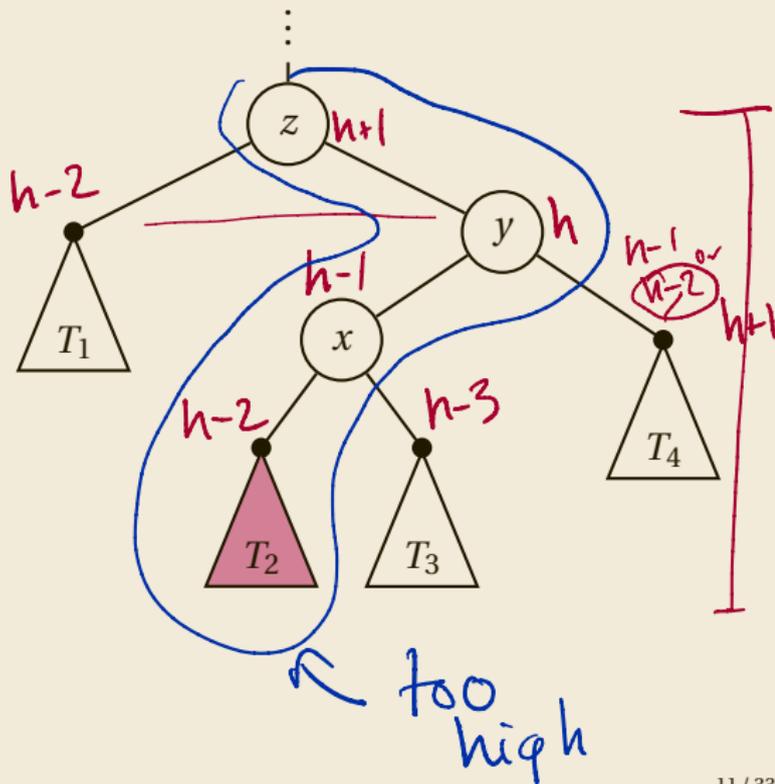
- $z: h+1$
- $y: h$
- $x: h-1$
- $T_1: h-2$
- $T_2: h-2$
  - why not $h-3$?
- $T_3: h-3$
  - why not $h-4$?
- $T_4: h-2$

h-2

h-1

z $h+1$

y h

x

h-1 or $\boxed{h-2}$ $h+1$

h-2

h-3

T_1

T_2

T_3

T_4

← too high

Heights After Add

Heights after $\text{ADD}(w)$

- $z : h+1$
- $y : h$
- $x : h-1$
- $T_1 : h-2$
- $T_2 : h-2$
  - why not $h-3$?
- $T_3 : h-3$
  - why not $h-4$?
- $T_4 : h-2$
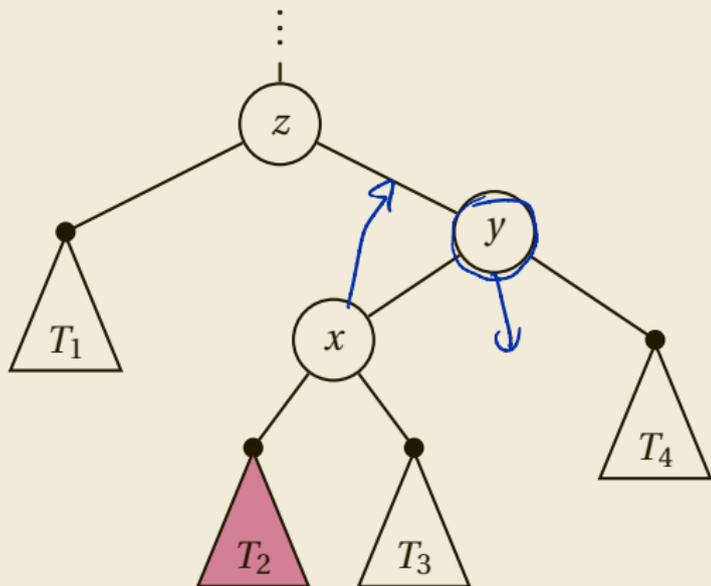
Question. How to "pull" T_2 up?

Heights After Add

Heights after ADD(w)

- z: $h+1$
- y: h
- x: $h-1$
- T_1: $h-2$
- T_2: $h-2$
  - why not $h-3$?
- T_3: $h-3$
  - why not $h-4$?
- T_4: $h-2$
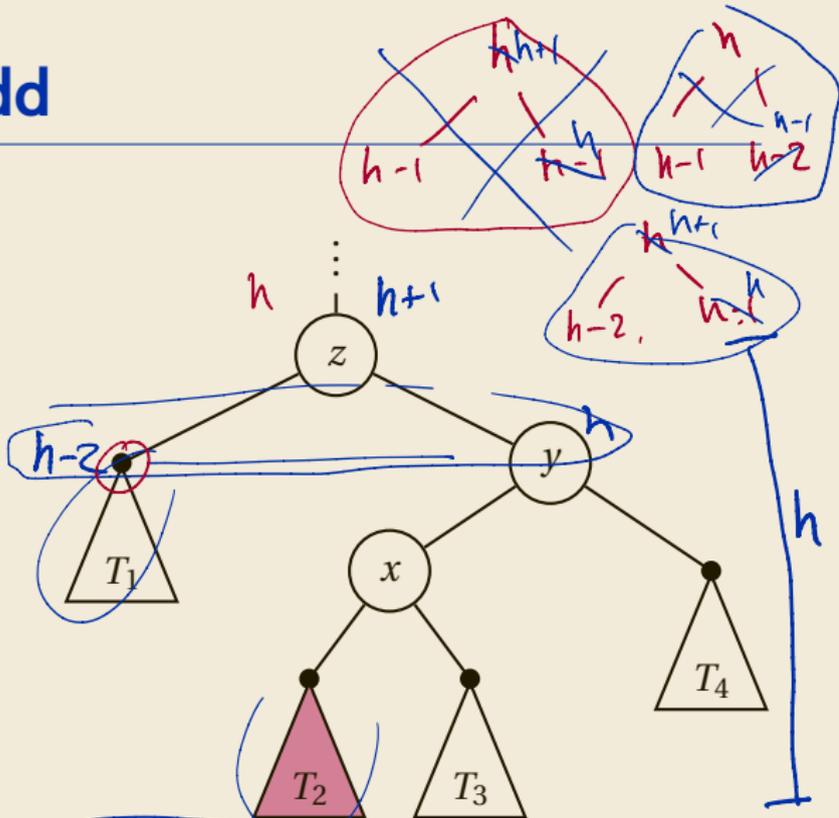
Question. How to "pull" T_2 up?
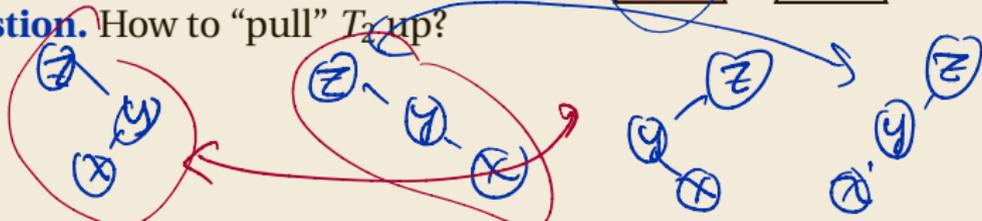
Heights After Right Rotation at y

PollEverywhere

What is the new height of z's right child?

pollev.com/comp526
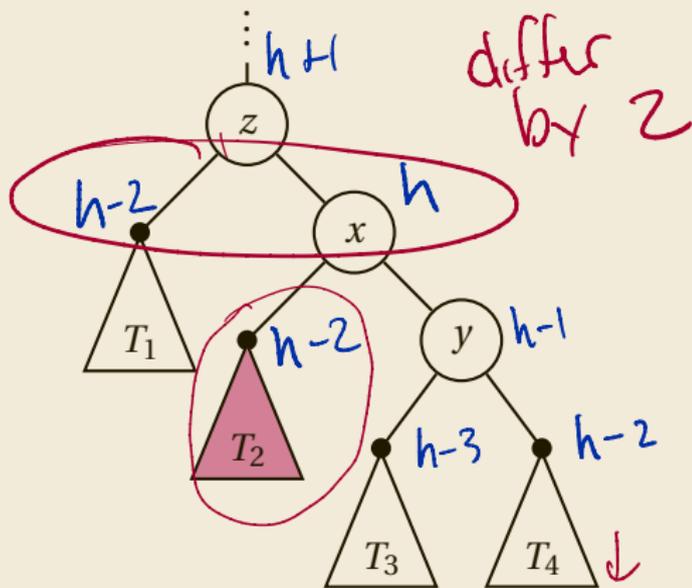
Heights After Right Rotation at y

Heights after Right Rotation at y

- $z : h+1$
- $y : h-1$
- $x : h$
- $T_1 : h-2$
- $T_2 : h-2$
- $T_3 : h-3$
- $T_4 : h-2$
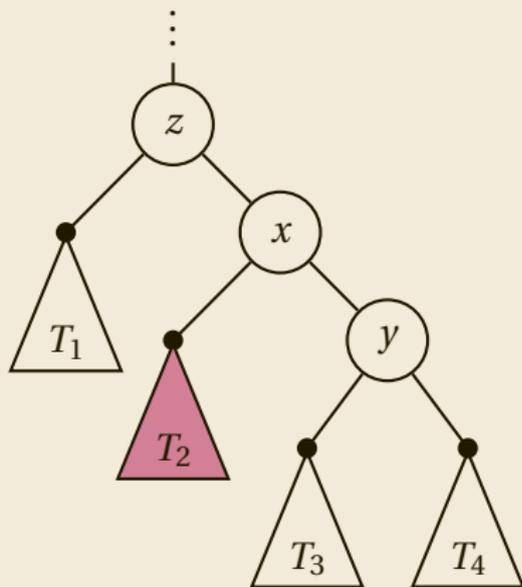
Heights After Right Rotation at y

Heights after Right Rotation at y

- $z: h+1$
- $y: h-1$
- $x: h$
- $T_1: h-2$
- $T_2: h-2$
- $T_3: h-3$
- $T_4: h-2$
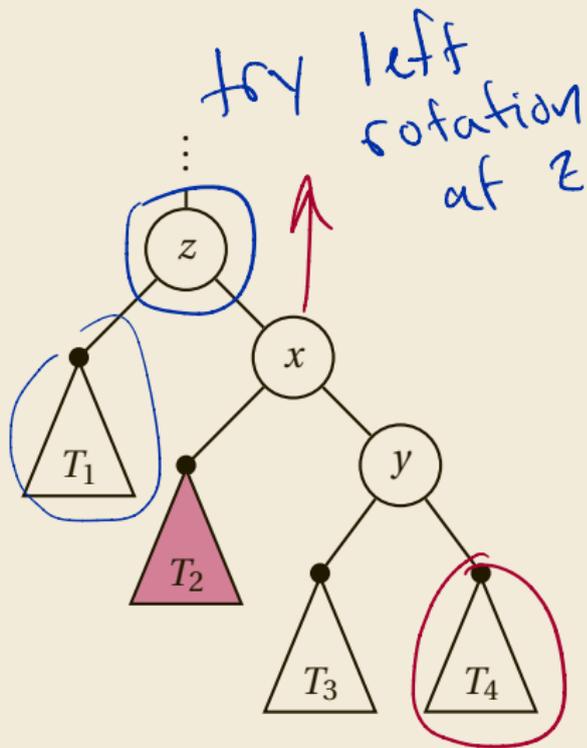
Damn! What if we try again?

Heights After Left Rotation at z

Heights after Right Rotation at y

- z:
- y:
- x:
- $T_1 : h - 2$
- $T_2 : h - 2$
- $T_3 : h - 3$
- $T_4 : h - 2$
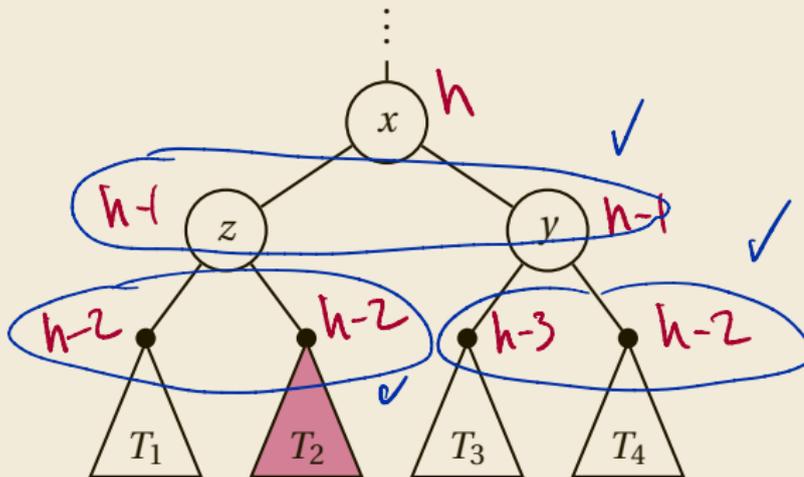
Balance restored for entire tree

Heights After Left Rotation at z

Heights after Right Rotation at y

- z:
- y:
- x:
- $T_1 : h - 2$
- $T_2 : h - 2$
- $T_3 : h - 3$
- $T_4 : h - 2$
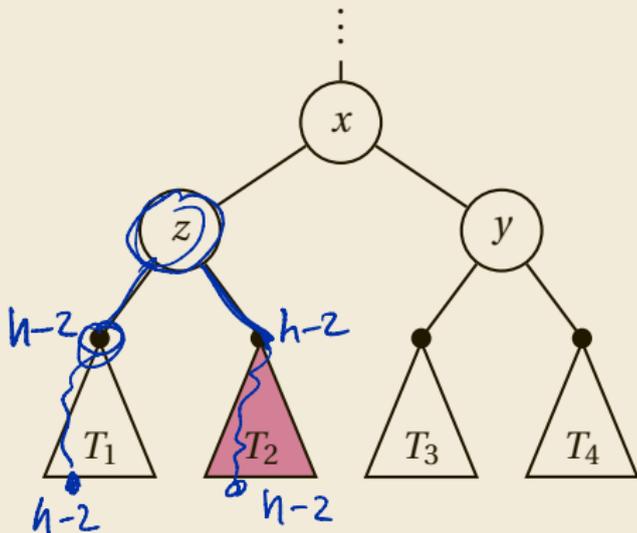
Hooray! We restored balance!!

- … Not just at in our subtree, but on the whole tree?

Other Cases

Example we considered
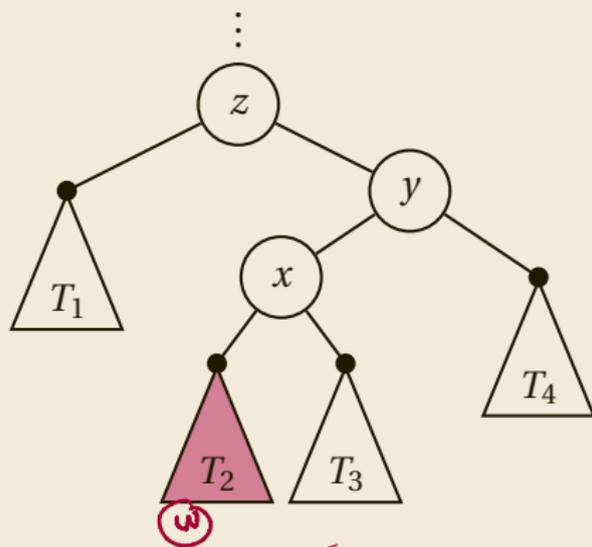
Another Possibility
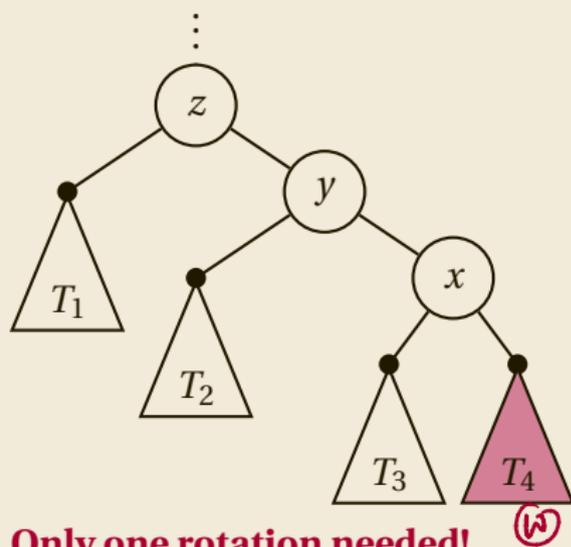
Only one rotation needed!

Also to consider: mirror images.

- These are the only 4 possibilities for z, y, and x.

Implementation Details

Unfortunately to pull this off, we need more overhead.

- More storage:
    - maintain **height** of each vertex (in addition to references to children, parent)
- More **work** on each ADD/REMOVE:
    - update the heights of vertices
    - check for imbalance
    - restore balance as above

Implementation Details

Unfortunately to pull this off, we need more overhead.

- More storage:
  - maintain **height** of each vertex (in addition to references to children, parent)
- More **work** on each ADD/REMOVE:
  - update the heights of vertices
  - check for imbalance
  - restore balance as above

PollEverywhere

What is the add'l cost of checking/restoring balance for ADD?

1. $\Theta(1)$
2. $\Theta(\log n)$
3. $\Theta(\sqrt{n})$
4. $\Theta(n)$

`pollev.com/comp526`

Implementation Details

$h = \Theta(\log n)$ b/c tree is balanced

Unfortunately to pull this off, we need more overhead.

- More storage:
  - maintain **height** of each vertex (in addition to references to children, parent)
- More **work** on each ADD/REMOVE:
  - update the heights of vertices
    - **Only need to update ancestors of added vertex**
  - check for imbalance
    - **Only need to check ancestors of added vertex**
  - restore balance as above
    - **Only takes $O(1)$ time!**

$O(\log n)$ vertices

$\Theta(\log n)$ vertices

They Payoff

This scheme for balancing BST is called **AVL trees**

- Named for **A**delson-**V**elsky and **L**andis (1962)

Similar re-balancing technique also works for REMOVE method

- Re-balancing removal also takes worst case $\Theta(\log n)$ time.

Big Deal: We can now implement ORDEREDSETs and MAPs where **all** operations are performed in worst case $O(\log n)$ time!

They Payoff

This scheme for balancing BST is called **AVL trees**

- Named for **A**delson-**V**elsky and **L**andis (1962)

Similar re-balancing technique also works for REMOVE method

- Re-balancing removal also takes worst case $\Theta(\log n)$ time.

Big Deal: We can now implement ORDEREDSETs and MAPs where **all** operations are performed in worst case $O(\log n)$ time!

Other balanced (binary) tree implementations also exist:

- Red-Black trees
- Scapegoat trees
- 2-3 trees
- …

All have similar *worst case, asymptotic* running time

- different implementations suited for different applications

ADT & Data Structure Recap

Simple ADTs

- STACK
- QUEUE
- DEQUE

Efficient implementation with
linear data structures:

- arrays
- linked lists

All operations performed in
(amortized) $\Theta(1)$ time.

ADT & Data Structure Recap

Simple ADTs

- STACK
- QUEUE
- DEQUE

Efficient implementation with linear data structures:

- arrays
- linked lists

All operations performed in (amortized) $\Theta(1)$ time.

Sophisticated ADTs

- PRIORITYQUEUE
- MAP (associative array, dictionary, symbol table)
- Ordered Set

Efficient implementation with tree-like data structures

- heaps
- (balanced) binary search trees

All operations in (amortized) $O(\log n)$ time.
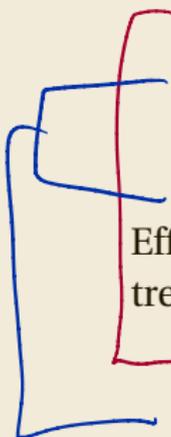
Sorting

The Sorting Task

Fundamental Task: sorting a list of elements from smallest to largest
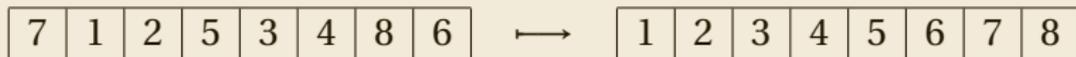
| 7 | 1 | 2 | 5 | 3 | 4 | 8 | 6 | $\longmapsto$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

Typical basic (unit cost) operations:

- **compare** two elements to see which is larger
- **swap** two elements in the array

The Sorting Task

Fundamental Task: sorting a list of elements from smallest to largest
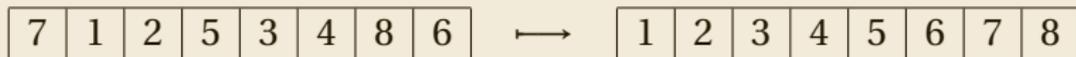
| 7 | 1 | 2 | 5 | 3 | 4 | 8 | 6 | $\longmapsto$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

Typical basic (unit cost) operations:

- **compare** two elements to see which is larger
- **swap** two elements in the array

(Perhaps) surprisingly sorting is still an active area of study/research!

- practical and theoretical improvements still being found
- algorithms for different contexts
  - e.g., non-standard sorting models

Elementary Sorting

Iterative sorting:

- Sort in **phases** where each phase accomplishes some global task.

Three Basic Strategies

1. SELECTIONSORT
   - Each phase i finds the smallest element in $a[i \dots n-1]$ and swaps it into position i
   - Uses (asymptotically) fewest SWAPs possible

Elementary Sorting

Iterative sorting:

- Sort in **phases** where each phase accomplishes some global task.

Three Basic Strategies

1. SELECTIONSORT
   - Each phase i finds the smallest element in $a[i \ldots n-1]$ and swaps it into position i
   - Uses (asymptotically) fewest SWAPS possible

2. BUBBLESORT
   - Each phase iterates over adjacent *pairs* and swaps those which are out of order
     - after phase i, $a[n-i-1 \ldots n-1]$ contains the i largest elements sorted
   - Used mostly for illustrative purposes.

largest element
here

second largest
here after phase 2.

Elementary Sorting

Iterative sorting:

- Sort in **phases** where each phase accomplishes some global task.

Three Basic Strategies

1. SELECTIONSORT
   - Each phase i finds the smallest element in $a[i \ldots n-1]$ and swaps it into position i
   - Uses (asymptotically) fewest SWAPs possible

2. BUBBLESORT
   - Each phase iterates over adjacent *pairs* and swaps those which are out of order
     - after phase i, $a[n-i-1 \ldots n-1]$ contains the i largest elements sorted
   - Used mostly for illustrative purposes.

3. INSERTIONSORT
   - Each phase i inserts $x = a[i]$ into sorted order in $a[0 \ldots i]$
   - Typically fast for *small* sequences and "almost sorted" sequences

InsertionSort in Detail

Phases $i = 1, 2, \ldots, n-1$:

- Phase i moves $x = a[i]$ into sorted position in $a[0 \ldots i]$.

- Performed via adjacent comparisons:

  - if x is smaller than left neighbor, swap x with left neighbor

1: **procedure** INSERTIONSORT(a, n)
2:     **for** $i = 1, 2, \ldots, n-1$ **do** i = phase
3:         $j \leftarrow i$
4:         **while** $j > 0$ and $a[j] < a[j-1]$ **do**
5:             SWAP($a, j, j-1$)
6:             $j \leftarrow j-1$
7:         **end while**
8:     **end for**
9: **end procedure**

Phase i

sorted

InsertionSort in Detail

PollEverywhere

What is the *worst case* running time of INSERTIONSORT?

1. $\Theta(n)$
2. $\Theta(n \log n)$
3. $\Theta(n^2)$
4. $\Theta(2^n)$

pollev.com/comp526

```
1: procedure INSERTIONSORT(a, n)
2:     for i = 1, 2, …, n−1 do
3:         j ← i          — Θ(1)
4:         while j > 0 and a[j] < a[j−1] do
5:             SWAP(a, j, j−1)
6:             j ← j−1
7:         end while
8:     end for
9: end procedure
```

$$1 + 2 \cdots \frac{n}{2} + \frac{n}{2} + 1 \cdots n-1$$

$$\Theta(n)$$

$$\Theta(n^2) \geq \frac{n^2}{4} = \frac{n}{2} \times \frac{n}{2}$$

$$n-1$$

$\leq i$ iterations
could be
as many
as i

Summing up ops:

$$1 + 2 + 3 + 4 + \cdots + n-1$$

$$=$$

InsertionSort in Detail

State after each phase:

	0	1	2	3	4
1	4	3	5	1	2
2	3	4	5	1	2
3	3	4	5	1	2
4	1	3	4	5	2
5	1	2	3	4	5

1: **procedure** INSERTIONSORT(a, n)
2:     **for** $i = 1, 2, \ldots, n-1$ **do**
3:         $j \leftarrow i$
4:         **while** $j > 0$ and $a[j] < a[j-1]$ **do**
5:             SWAP($a, j, j-1$)
6:             $j \leftarrow j-1$
7:         **end while**
8:     **end for**
9: **end procedure**

Phase 3

	0	1	2	3	4
0	3	4	5	1	2
1	3	4	1	5	2
2	3	1	4	5	2
3	1	3	4	5	2

Sorting Using Heaps

Recall the (array backed) heap data structure:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	3	13	10	6	66	39	42	17	96	70	89	95	98	63

Heap Operations in $O(\log n)$ time:

- INSERT(x)
- REMOVEMIN().

Question. How to use heaps to sort *efficiently* ($o(n^2)$ time)?

Sorting Using Heaps

Recall the (array backed) heap data structure:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	3	13	10	6	66	39	42	17	96	70	89	95	98	63

Heap Operations in $O(\log n)$ time:

- INSERT(x)
- REMOVEMIN().

Question. How to use heaps to sort *efficiently* ($o(n^2)$ time)?

- Add all elements to a heap. $\quad n \cdot \log n$
  $\uparrow$ ops $\quad \nwarrow$ time per op
- Repeatedly REMOVEMIN and add elements back to sorted array

What is the running time of this procedure?

$$\bigcirc(n \log n). \quad \ll \quad n^2$$

Sorting Using Heaps

Recall the (array backed) heap data structure:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	3	13	10	6	66	39	42	17	96	70	89	95	98	63

Heap Operations in $O(\log n)$ time:

- INSERT(x)
- REMOVEMIN().

Question. How to use heaps to sort *efficiently* ($o(n^2)$ time)?

- Add all elements to a heap.
- Repeatedly REMOVEMIN and add elements back to sorted array

What is the running time of this procedure?

- $\Theta(n \log n)$ This is much better than $\Theta(n^2)$!

Another Question. Do we need a separate heap?

Sorting In-Place

no extra storage for values

Heap Modification: **MaxHeap**

- Same as MinHeap, but all inequalities reversed
    - **Largest** value at root
    - Children store **smaller** values

HEAPSORT outline:

1. Make array a MaxHeap
    - HEAPIFY by calling BUBBLEUP at each index
2. Sort from right side of array
    - swap $a[0]$ with $a[n-i-1]$
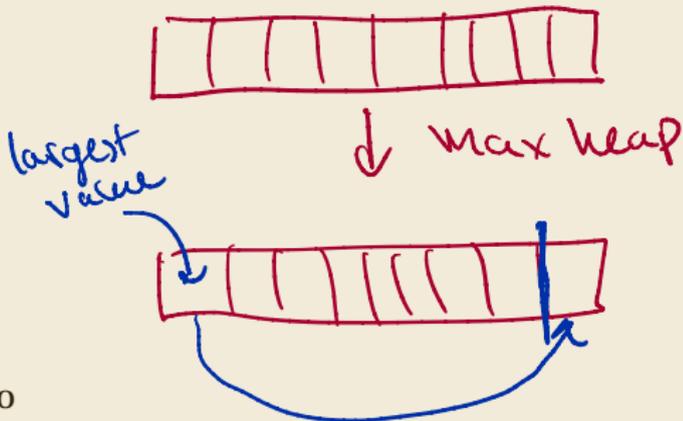   - TRICKLEDOWN from $a[0]$ to $a[n-i-1]$

Max heap

largest value

Sorting In-Place

Heap Modification: **MaxHeap**

- Same as MinHeap, but all inequalities reversed
  - **Largest** value at root
  - Children store **smaller** values

HEAPSORT outline:

1. Make array a MaxHeap
   - HEAPIFY by calling BUBBLEUP at each index
2. Sort from right side of array
   - swap $a[0]$ with $a[n-i-1]$
   - TRICKLEDOWN from $a[0]$ to $a[n-i-1]$

max heap property

```
1:  procedure HEAPSORT(a, n)
2:      for i = 1, 2, …, n − 1 do
3:          BUBBLEUP(a, i)
4:              ▷ Start from index i
5:      end for
6:      for i = n − 1, n − 2, …, 1 do
7:          SWAP(a, 0, i)
8:          TRICKLEDOWN(a, i − 1)
9:              ▷ Stop at index i − 1
10:     end for
11: end procedure
```

Sorting In-Place

Heap Modification: **MaxHeap**

- Same as MinHeap, but all inequalities reversed
  - **Largest** value at root
  - Children store **smaller** values

HEAPSORT outline:

1. Make array a MaxHeap
   - HEAPIFY by calling BUBBLEUP at each index
2. Sort from right side of array
   - swap $a[0]$ with $a[n-i-1]$
   - TRICKLEDOWN from $a[0]$ to $a[n-i-1]$

Question. What is the running time of HEAPSORT?

Heapify

1: **procedure** HEAPSORT(a, n)
2:   **for** $i = 1, 2, \ldots, n-1$ **do**
3:     BUBBLEUP(a, i) $O(\log n)$
4:          ▷ Start from index i
5:   **end for**
6:   **for** $i = n-1, n-2, \ldots, 1$ **do**
7:     SWAP($a, 0, i$) $O(\log n)$
8:     TRICKLEDOWN($a, i-1$)
9:          ▷ Stop at index $i-1$
10: **end for**
11: **end procedure**
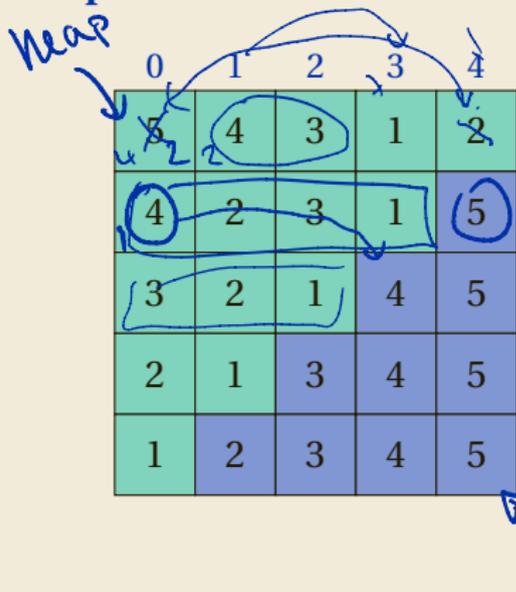
$O(n \log n)$

HeapSort Example

Step 1: HEAPIFY!

0	1	2	3	4
4	3	5	1	2
4	3	5	1	2
5	3	4	1	2
5	3	4	1	2
5	3	4	1	2

un processed

1: **procedure** HEAPSORT(a, n)
2:    **for** $i = 1, 2, \ldots, n - 1$ **do**
3:       BUBBLEUP(a, i)
4:          ▷ Start from index i
5:    **end for**
6:    **for** $i = n - 1, n - 2, \ldots, 1$ **do**
7:       SWAP($a, 0, i$)
8:       TRICKLEDOWN($a, i - 1$)
9:          ▷ Stop at index $i - 1$
10:    **end for**
11: **end procedure**

HeapSort Example

Step 2: Remove maximum values!

1: **procedure** HEAPSORT(a, n)
2:     **for** $i = 1, 2, \ldots, n-1$ **do**
3:         BUBBLEUP(a, i)
4:             ▷ Start from index i
5:     **end for**
6:     **for** $i = n-1, n-2, \ldots, 1$ **do**
7:         SWAP($a, 0, i$)
8:         TRICKLEDOWN($a, i-1$)
9:             ▷ Stop at index $i-1$
10:     **end for**
11: **end procedure**

Worst case running time is $\Theta(\log n)$, but HEAPSORT doesn't perform great in practice (for large arrays) $\Theta(n \log n)$ overall
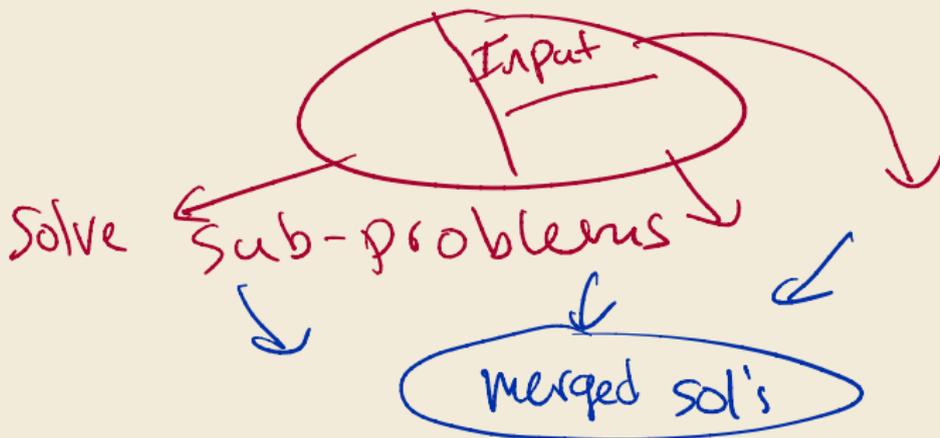
- poor locality of reference

Sorting by Divide & Conquer

The Divide & Conquer Strategy

Generic Strategy

Given an algorithmic task:

1. Break the input into smaller instances of the task
2. Solve the smaller instances
   - this is typically recursive!
3. Combine smaller solutions to a solution to the whole task

The Divide & Conquer Strategy

Generic Strategy

Given an algorithmic task:

1. Break the input into smaller instances of the task
2. Solve the smaller instances
   - this is typically recursive!
3. Combine smaller solutions to a solution to the whole task

Divide & Conquer Sorting

MERGESORT: Divide by *index*

- divide array into left and right halves
- recursively sort halves
- merge halves

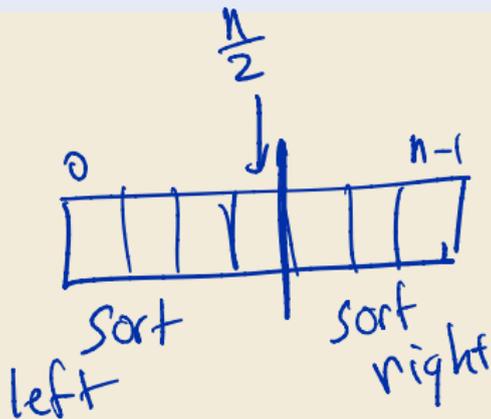# The Divide & Conquer Strategy

Generic Strategy

Given an algorithmic task:

1. Break the input into smaller instances of the task
2. Solve the smaller instances
   - this is typically recursive!
3. Combine smaller solutions to a solution to the whole task

Divide & Conquer Sorting

MERGESORT: Divide by *index*

- divide array into left and right halves
- recursively sort halves
- merge halves

QUICKSORT: Divide by *value*

- pick a *pivot value p*
- split array according to *p*
  - ≤ *p* on left, > *p* on right
- recursively sort sub-arrays

Merging Sorted Arrays

Think about for next time

Question

Suppose we are given two **sorted arrays**, a and b. How can we merge them into a single sorted array that contains all the values from both arrays?

0	1	2	3	4
2	3	6	7	8

0	1	2	3
1	4	5	9

0	1	2	3	4	5	6	7	8

Merging Code

Merging *sorted* arrays *a* (size *m*) and *b* (size *n*) into array *c* starting at index *s*

```
1: procedure MERGE(a, b, c, s, m, n)        ▷
      Merge arrays a and b into array c
      starting at index s. a has size m and b
      has size n
2:    i, j ← 0, k ← s
3:    while k < s + m + n do
4:        if j = n or a[i] < b[j] then
5:            c[k] ← a[i]
6:            i ← i + 1
7:        else
8:            c[k] ← b[j]
9:            j ← j + 1
10:       end if
11:       k ← k + 1
12:   end while
13: end procedure
```

Merging Code

```
1: procedure MERGE(a, b, c, s, m, n)        ▷
       Merge arrays a and b into array c
       starting at index s. a has size m and b
       has size n
2:     i, j ← 0, k ← s
3:     while k < s + m + n do
4:         if j = n or a[i] < b[j] then
5:             c[k] ← a[i]
6:             i ← i + 1
7:         else
8:             c[k] ← b[j]
9:             j ← j + 1
10:        end if
11:        k ← k + 1
12:    end while
13: end procedure
```

Sorting by Merging

MERGESORTStrategy:

- To sort $a[i\ldots k]$:
    - If $i = k$, then we're done
    - Othewise split (sub)interval in half
    - Recursively sort halves
    - Merge sorted halves
        - copy values to new arrays for this

Sorting by Merging

MERGESORTStrategy:

- To sort $a[i \ldots k]$:
    - If $i = k$, then we're done
    - Otherwise split (sub)interval in half
    - Recursively sort halves
    - Merge sorted halves
        - copy values to new arrays for this

1: **procedure** MERGESORT(a, i, k)
2:     **if** $i < k$ **then**
3:         $j \leftarrow \lfloor (i + k)/2 \rfloor$
4:         MERGESORT(a, i, j)
5:         MERGESORT($a, j + 1, k$)
6:         $b \leftarrow$ COPY(a, i, j)
7:         $c \leftarrow$ COPY($a, j + 1, k$)
8:         MERGE(b, c, a, i)
9:     **end if**
10: **end procedure**

Sorting by Merging

```
 1: procedure MERGESORT(a, i, k)
 2:     if i < k then
 3:         j ← ⌊(i + k)/2⌋
 4:         MERGESORT(a, i, j)
 5:         MERGESORT(a, j + 1, k)
 6:         b ← COPY(a, i, j)
 7:         c ← COPY(a, j + 1, k)
 8:         MERGE(b, c, a, i)
 9:     end if
10: end procedure
```

Sorting by Merging

Tracing the Recursive Calls
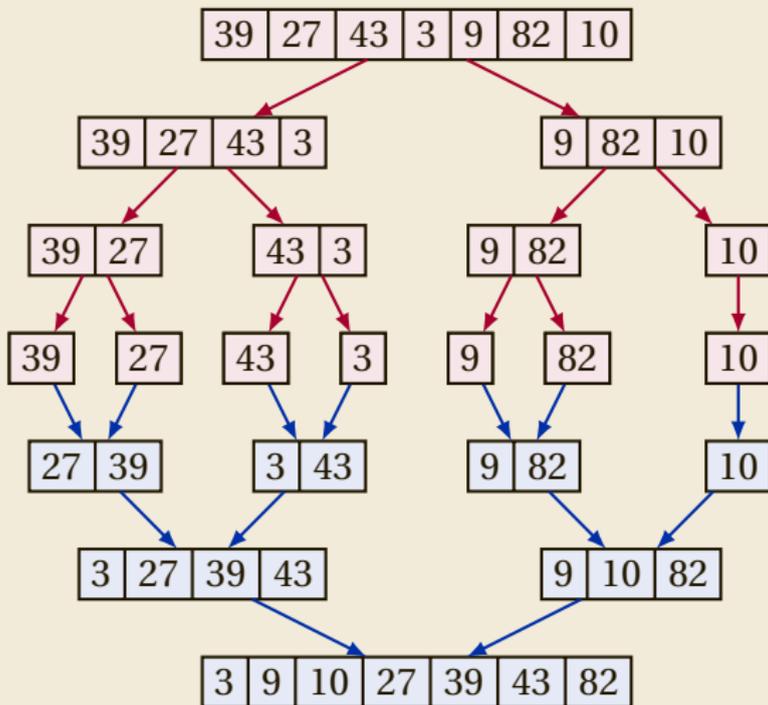
1: **procedure** MERGESORT(a, i, k)
2:     **if** $i < k$ **then**
3:         $j \leftarrow \lfloor (i+k)/2 \rfloor$
4:         MERGESORT(a, i, j)
5:         MERGESORT($a, j+1, k$)
6:         $b \leftarrow$ COPY(a, i, j)
7:         $c \leftarrow$ COPY($a, j+1, k$)
8:         MERGE(b, c, a, i)
9:     **end if**
10: **end procedure**

A Larger Example

tikz code courtesy of SebGlav on `tex.stackexchange.com`

MergeSort Analysis

Question. What is the running time of MERGESORT?

- How do we analyzing the running time of a recursive function?

MergeSort Analysis

Question. What is the running time of MERGESORT?

- How do we analyzing the running time of a recursive function?

Think about this for next time.

Next Time: More Sorting

- MERGESORT analysis
- QUICKSORT
- Lower Bounds
- Non-comparison Based Methods
- More Sorting Algorithms?

Scratch Notes