Week 07 Studio
Continuation-Passing Style, Searching, Sorting, BST

CS1101S AY21/22 Semester 1
Studio 05E

Yan Xiaozhi (David)
@david_eom
yan_xiaozhi@u.nus.edu

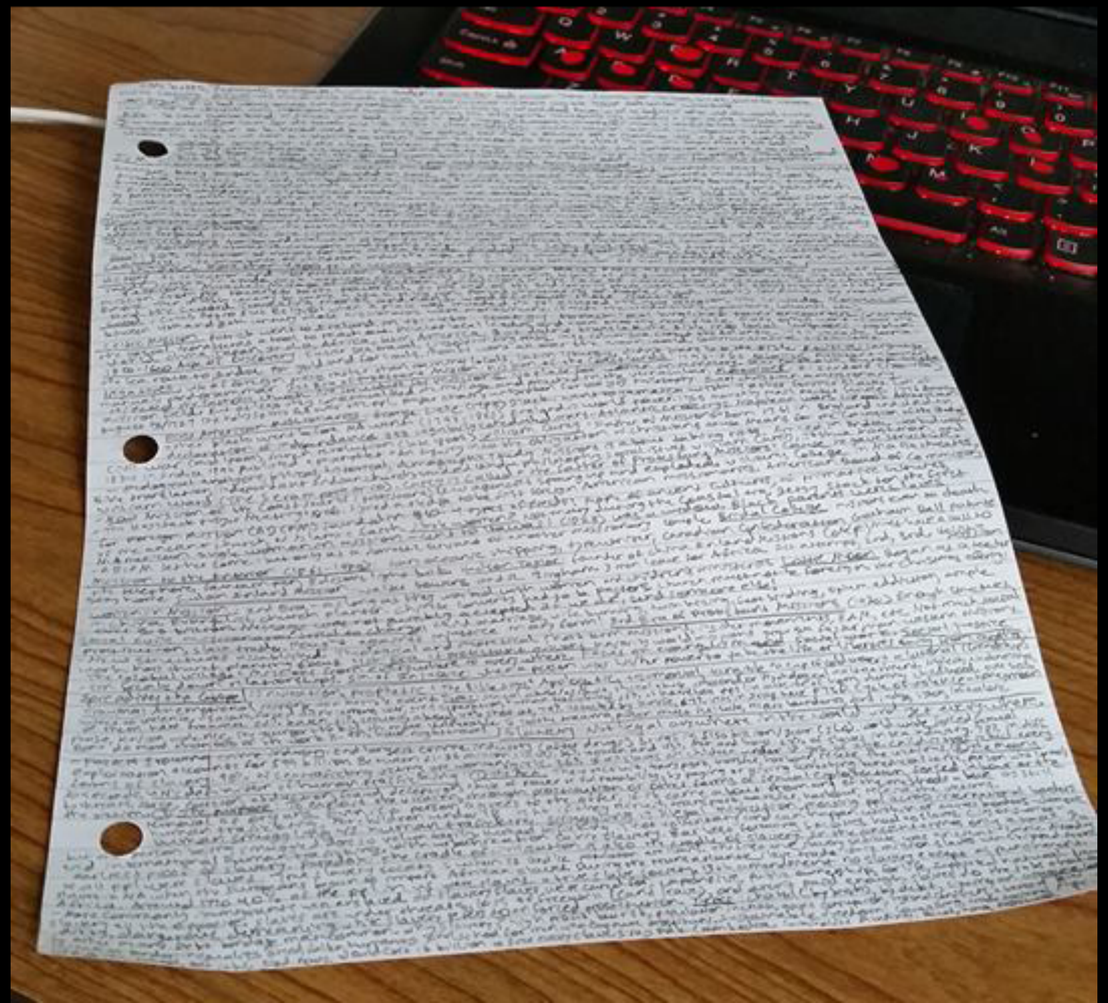
27 Sep 2021

Admin

- Contact tracing (QR code + class photo)
- Midterms this Wednesday!
  - Prepare your cheatsheet (1 page double sided)
  - Calm down, partial credits
- Mastery check progress
  - After Wednesday please! (for your own revision and mine as well :(( )
  - Format of mastery check

Teacher be like

“You can take 1 single sided cheat sheet into the final exam”



Recap

Continuation-Passing Style

What Is It?

- Passing the deferred operation as a function in an extra argument
- Tail-call recursion
- ```
  function app(current_xs, ys, c) {
      return is_null(current_xs)
          ? c(ys)
          : app(tail(current_xs), ys,
              x => c(pair(head(current_xs), x)));
  }
  ```
- ```
  function append_iter(xs, ys) {
      return app(xs, ys, x => x);
  }
  ```

(4) [6 marks]

The pre-declared function `accumulate` can be applied to "fold" a given list from right to left, starting from a given initial value, each time applying a given binary function.

Example:

```
accumulate( (x, y) => x / y, 2, list(24, 16, 8) )
```

evaluates to

```
24 / (16 / (8 / 2)) = 6
```

The function `accumulate` as given in the lectures gives rise to a recursive process. Write a function `accumulate_iter` that computes the same result as `accumulate`, but that gives rise to an *iterative process*. **Additional requirement: No pairs must be created by `accumulate_iter` when used instead of `accumulate` in the example above.**

```
function accumulate_iter(f, init, xs) {
    /* YOUR SOLUTION */
}
```

(Write the *entire function declaration* of `accumulate_iter` in the space provided below.)

Searching

What Is It?

- Integral part of modern algorithms
- Searching and sorting: interviewers love them!
- Usually sort first then search
  - O(sort + search) > O(linear search)
  - O(sort + m * search) > m * O(linear search)
  - Sort once, search multiple times
  - “Pre-processing”

Searching

Linear Search

- Go through every element in the list, check if it matches
- Precondition: nil (can compare equality of two items)
- Time complexity: O(n)
  - What if the item we are searching for is the last one?
  - What if the item does not exist in the list?
- CS50 analogy:
  - Searching through the yellow pages page by page

Searching

Binary Search

- Reduce the problem size by half every single time
- Precondition: items in the list must be ordered
  - Ascending or descending
- Time complexity: O(log n)
  - Assuming access to item is O(1), maximum log n steps
- CS50 analogy:
  - Tearing of the yellow pages in half


algorithms
algorith

programming pearls

By Jon Bentley

WRITING CORRECT PROGRAMS

The Challenge of Binary Search

Even with the best of designs, every now and then a programmer has to write subtle code. This column is about one problem that requires particularly careful code: binary search. After defining the problem and sketching an algorithm to solve it, we'll use principles of program verification in several stages as we develop the program.

Most programmers think that with the above description in hand, writing the code is easy; they're wrong. The only way you'll believe this is by putting down this column right now, and writing the code yourself. Try it.

I've given this problem as an in-class assignment in courses at Bell Labs and IBM. The professional programmers had one hour (sometimes more) to convert the above description into a program in the language of their choice; a high-level pseudo-code was fine. At the end of the specified time, almost all the programmers reported that they had correct code for the task. We would then take 30 minutes to examine their code, which the programmers did with test cases. In many different classes and with over a hundred programmers, the results varied little: 90 percent of the programmers found bugs in their code (and I wasn't always convinced of the correctness of the code in which no bugs were found).

I found this amazing: only about 10 percent of professional programmers were able to get this small program right. But they aren't the only ones to find this task difficult. In the history in Section 6.2.1 of his *Sorting and Searching*, Knuth points out that while the first binary search was published in 1946, the first published binary search without bugs did not appear until 1962.

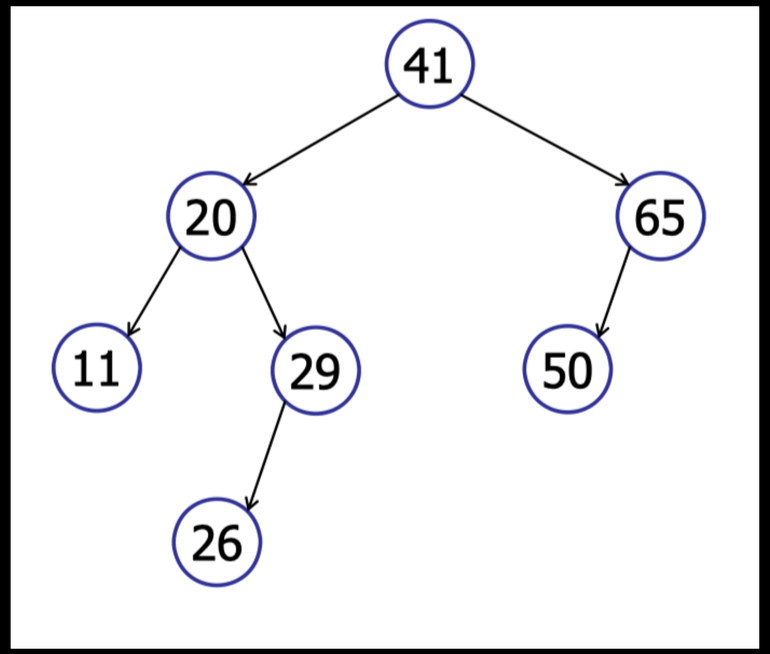
Binary Search Trees

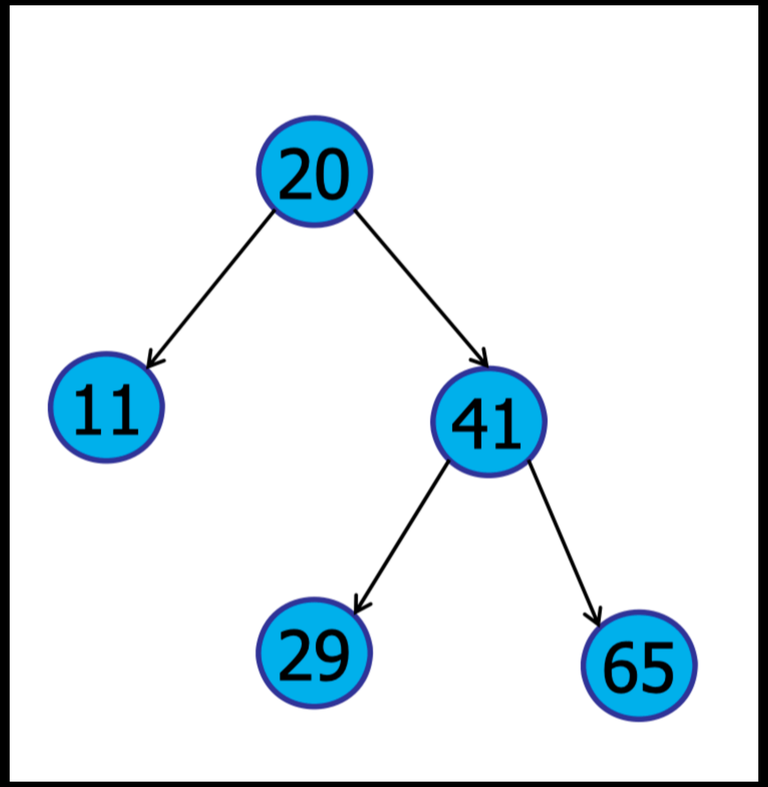
What Is It?

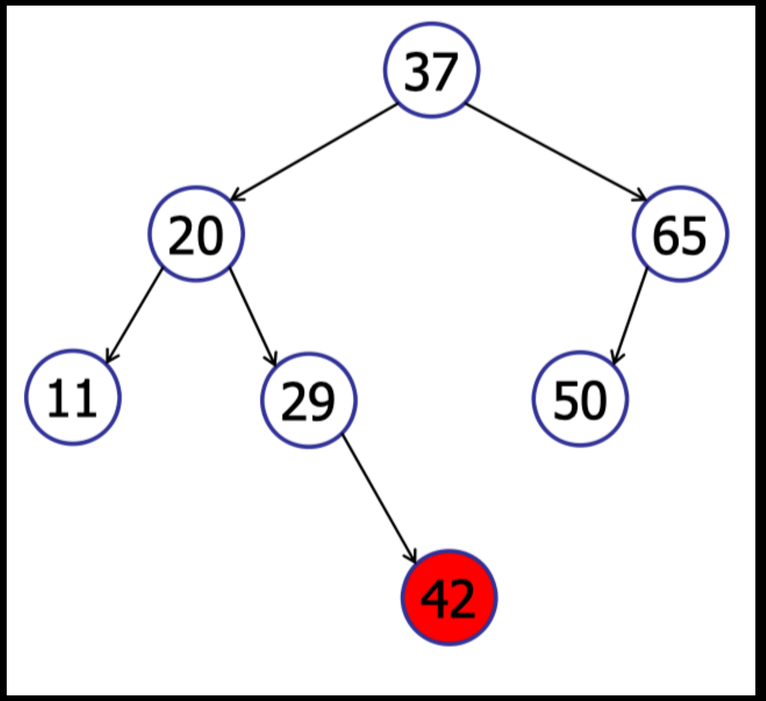
- A binary search tree is the empty tree, or it has an entry, a left branch and a right branch (both also binary search trees).
- Property:
  - All entries in left branch
    < entry
    < all entries in right branch
- Items are all distinct in CS1101S

Binary Search Trees

Are these BSTs?



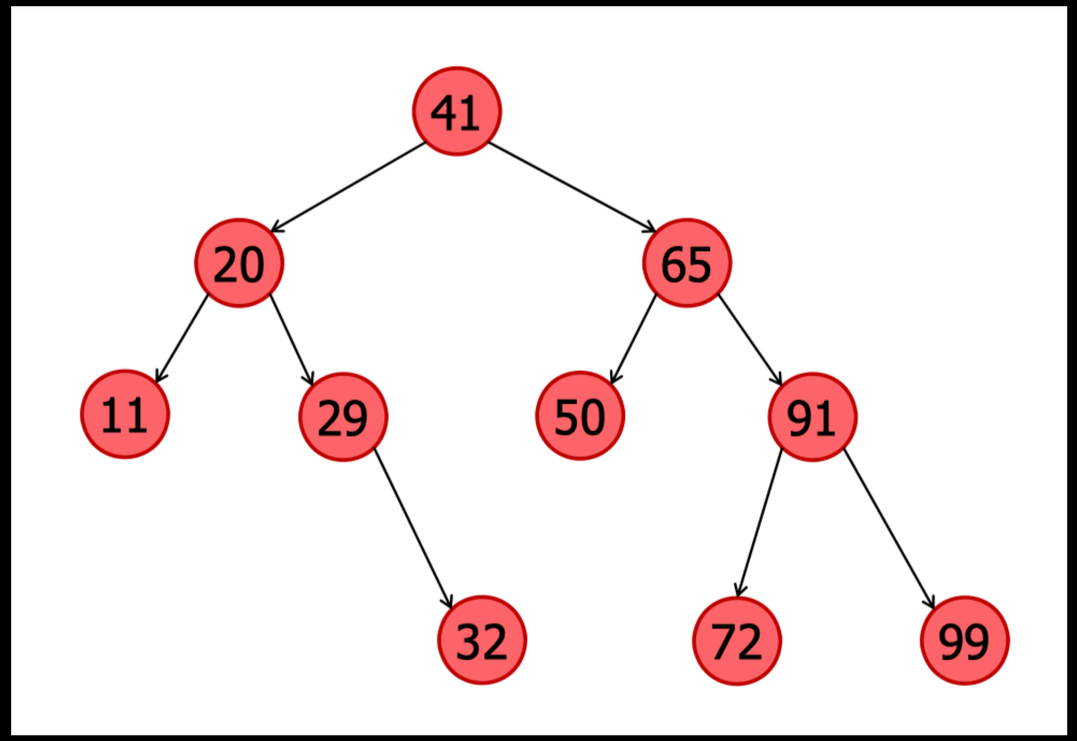




Binary Search Trees

Traversal

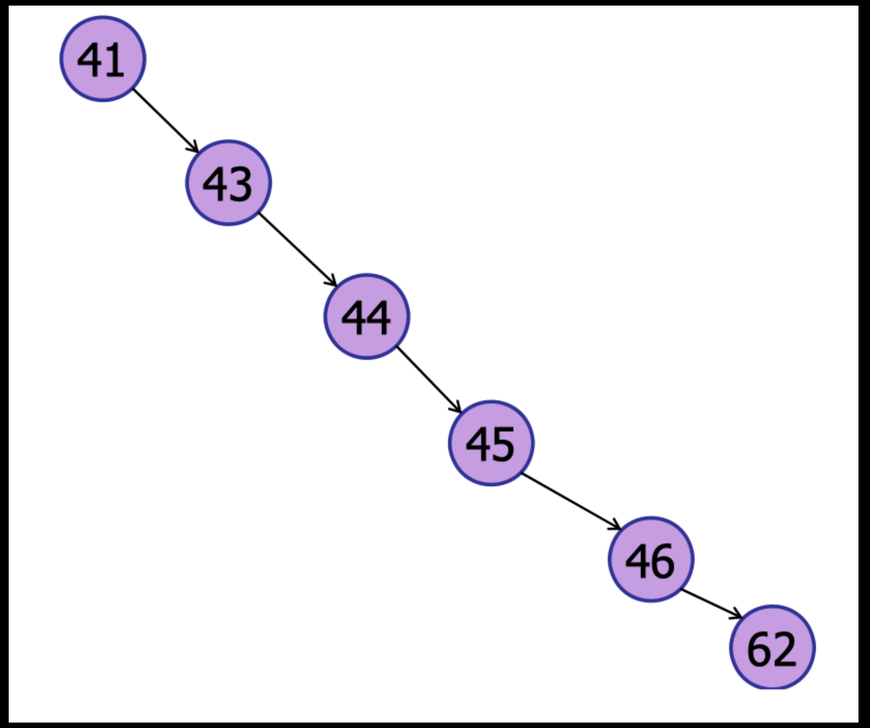
- Pre-order
  - Visit root, traverse left sub-tree, traverse right sub-tree
- In-order (items visited in sorted order)
  - Traverse left sub-tree, visit root, traverse right sub-tree
- Post-order
  - Traverse left sub-tree, traverse right sub-tree, visit root
- Gonna be covered in CS1231S!



Binary Search Trees

Searching in BST

- Worst-case time complexity of searching in BST?
  - O(n)!!! (one straight line bopes)
- O(log n) only in balanced trees
  - e.g. AVL tree
  - Implementation / insertion / deletion to be covered in CS2040S



Sorting

What Is It?

- Preprocessing for searching
- Common types:
  - Bogo sort
  - Insertion sort, selection sort, bubble sort
  - Merge sort, quick sort
- Implementations are important, but ideas are more important!
  - Can always put implementations in cheatsheet

Sorting

Bogo Sort

- Best sort, average performance: O(n ·n!)
- Actually not the worst! (Ingrassia-Kurtz Sort)
    1. Generate all permutation of the input
    2. Sort permutations based on number of inversion required using Bogo Sort
    3. Return the first element in the sorted list of permutations, minimum number of inversions, possibly sorted
    - O((n!)!)

Sorting

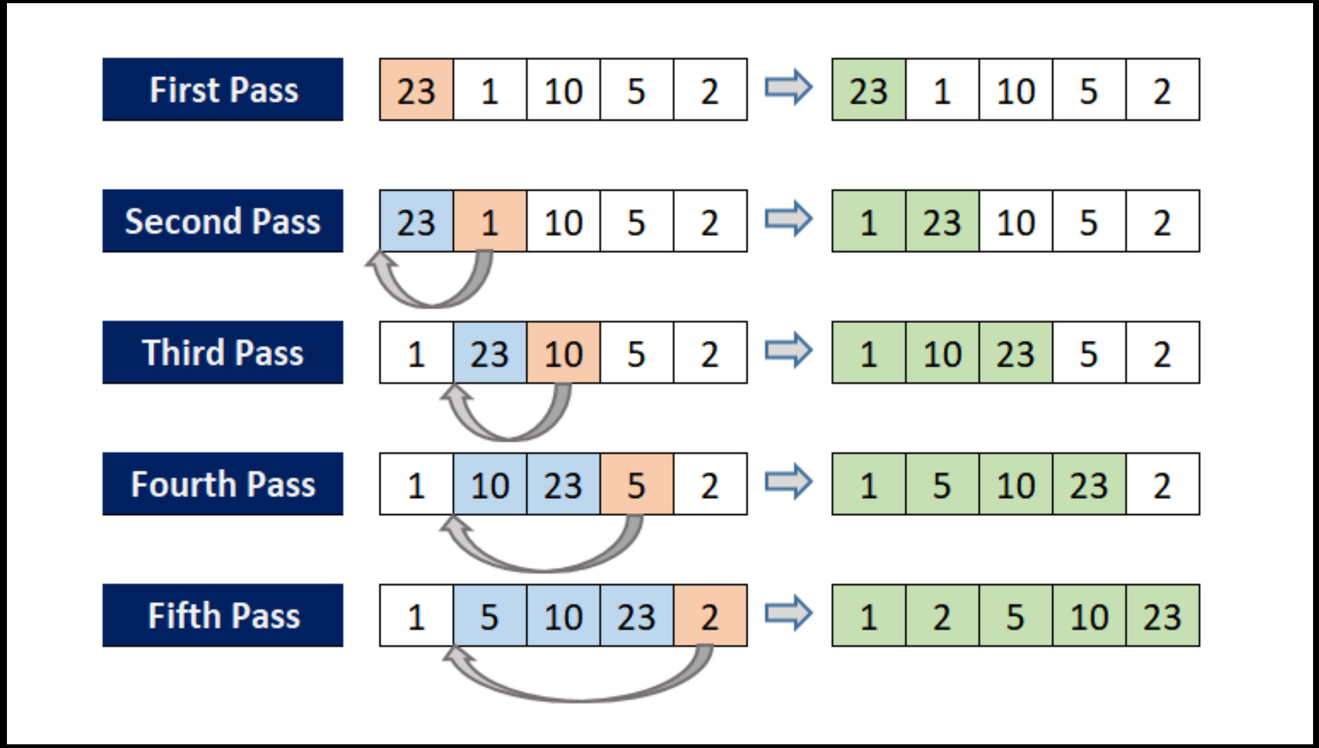
Insertion Sort

- Take the head of the list, insert it into the right place
- Use wishful thinking
- Invariant: after the n^{th} evaluation of `insertion_sort` the last `n` items in the list are sorted
- Time complexity: best-case $\Omega(n)$, average and worst-case $O(n^2)$
- Space complexity: $O(n)$

Sorting

Insertion Sort

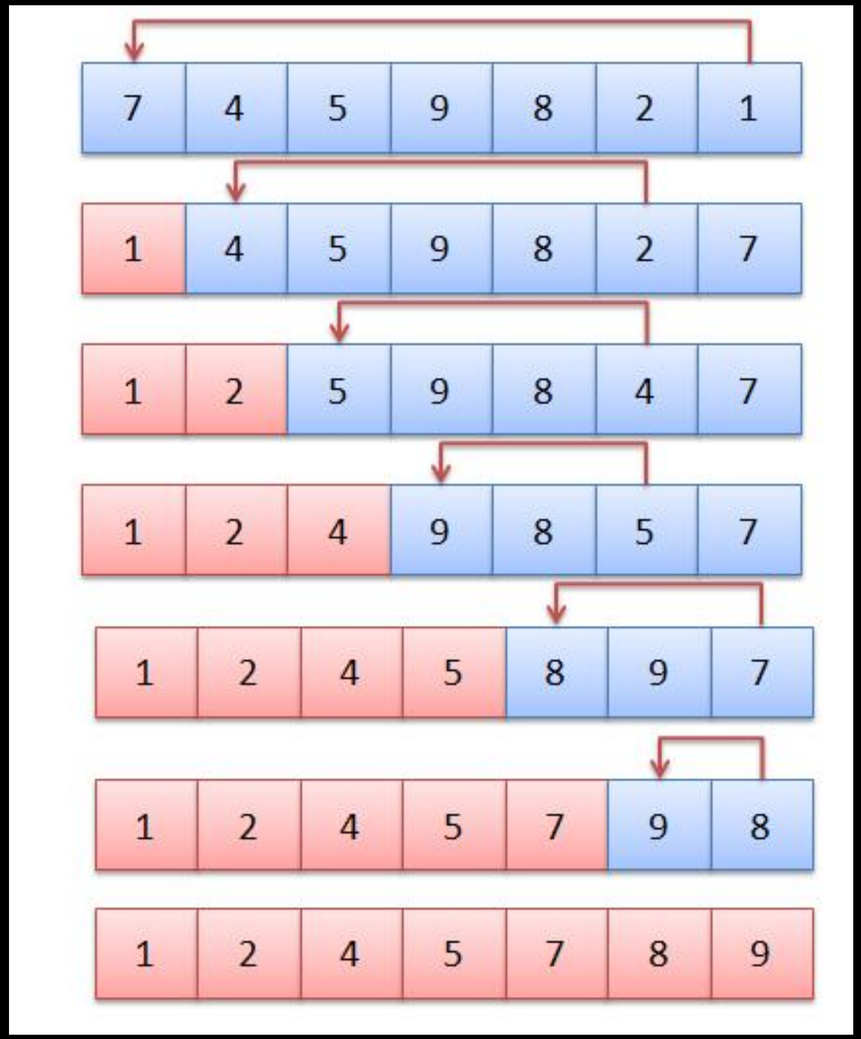
- The implementation in Source is NOT THE SAME as below!



Sorting

Selection Sort

- Select the smallest/largest element in the list, insert it at the start/end of the list
- Invariant: after the `n`th evaluation of `selection_sort` the smallest/biggest `n` items in the list are in their correct position
- Time complexity: $\Theta(n^2)$
- Space complexity: $O(n)$



Sorting

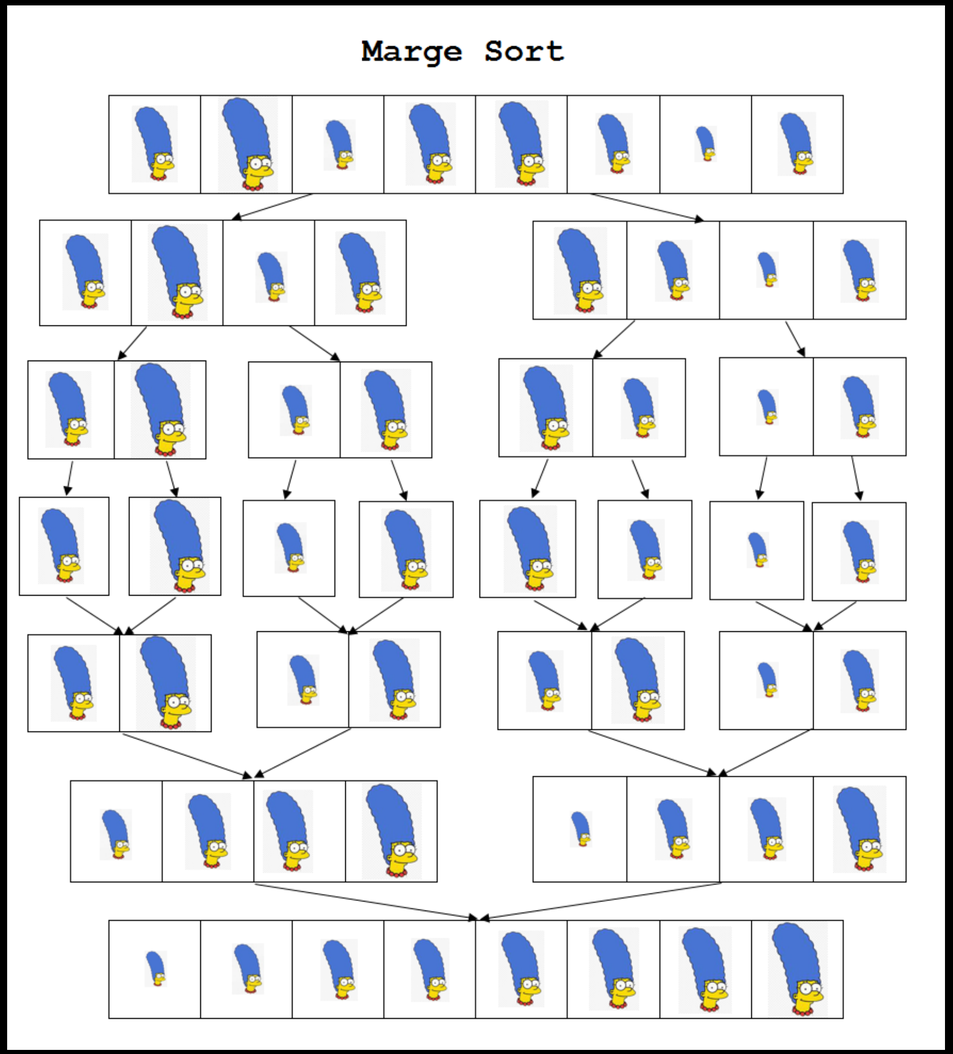
Bubble Sort (Not Covered)

- Repeatedly pass through the whole list
- Swapping the adjacent elements if they are in wrong order
- Finish sorting when 0 swap is required for one passing
- Invariant: after the n^{th} iteration the biggest n items in the list are in their correct position
- Time complexity: best-case $\Omega(n)$, average and worst-case $O(n^2)$
- Space complexity: $O(n)$

Sorting

Merge Sort

- Split the list into two halves
- Merge sort the two lists separately
  - If list is length 1 it is already sorted
- Merge the two lists (divide and conquer)
- Time complexity: O(n log n) (why?)
- Space complexity: extra O(n)



Sorting

Quick Sort

- Invented by C.A.R. Hoare in 1960
- Used for machine translation (English/Russian)
- Java’s default sorting algo: dual pivot quick sort
- Take a pivot (first item or random)
- Partition into two sublists, quick sort the sublists
- Join (smaller than + pivot + larger than)
  - What if pivot appears more than once?

Sorting

Quick Sort

- Time complexity: depends on the choice of pivot!
  - Best-case and average case: O(n log n)
    - The pivot nicely partitioned the list into 2 equal halves
  - Worst-case: $O(n^2)$
    - The pivot suay suay is the smallest/largest in the list
- Space complexity: O(n)

Sorting

Why Do We Need Quick Sort?

- We already have merge sort what?
- Merge sort worst case better than quick sort worst case
- Space complexity
  - Merge sort: extra O(n)
  - Quick sort: extra O(1), in-place algorithm
- Good caching performance
- Good parallelisation

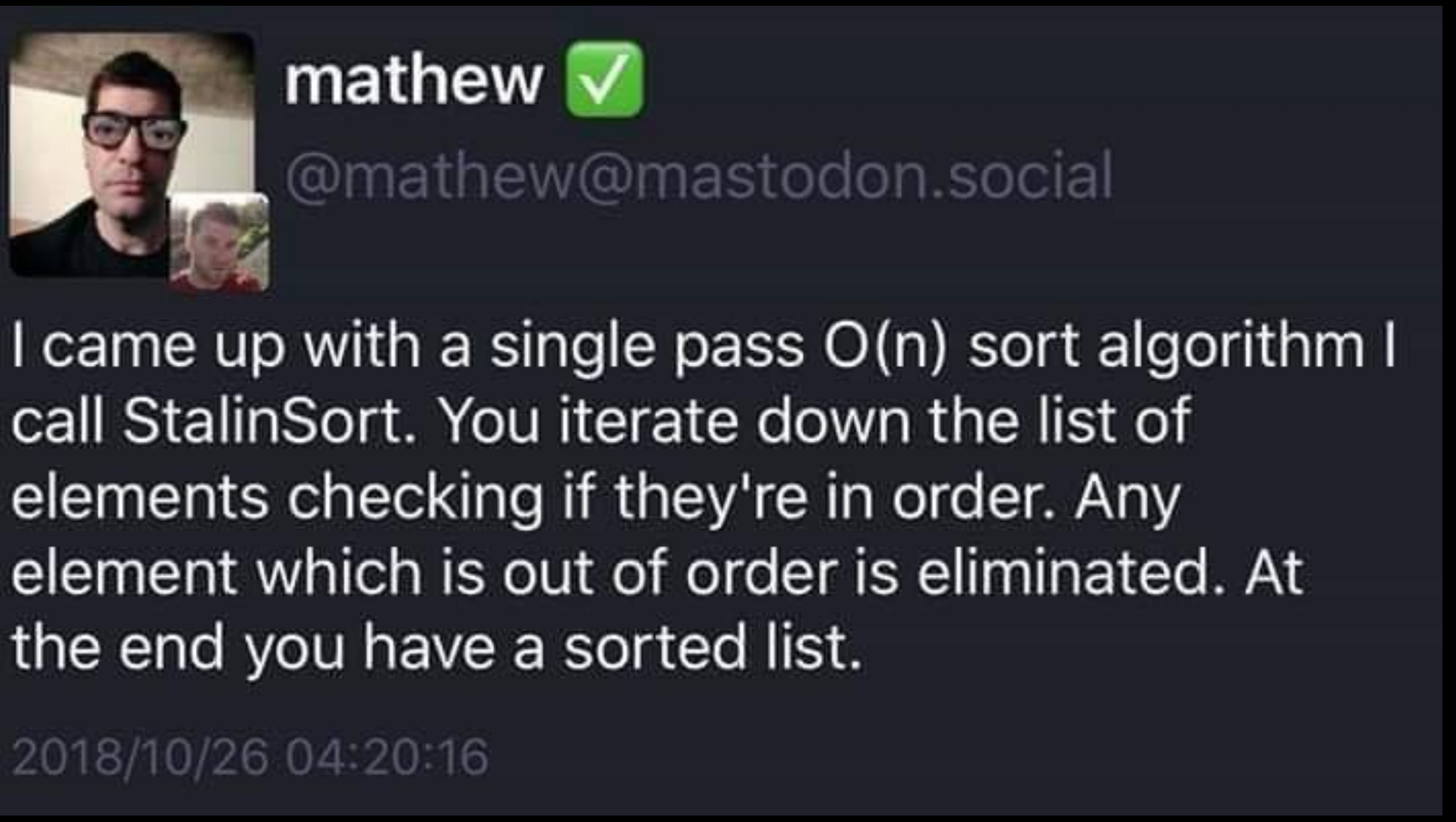
Sorting

Complexities

- Best average-case time complexity: O(n log n)
- Can we do better?
- “Any deterministic comparison-based sorting algorithm must take Ω(n log n) time to sort an array of n elements in the worst case”
- At least O(log n) comparisons are made

Sorting

Complexities



readme.md

What is Stalin Sort? ❓

Introduction

Stalin Sort is an efficient sorting algorithm, serving as a systematic method for placing the elements of a random access file or an array in order. Stalin Sort is also know as the best sorting algorithm of all times because of its AMAZING capacity of always ordering an array with an O(n) performance.

How it works?

It's simple, all you need to do is iterate through the array, checking if its elements are in order. Any element that isn't in order you pull out, in other words, you send it to Gulag.

Step-by-step example

1. (1 2 5 3 5 7) -> (**1** 2 5 3 5 7) Here the algorithm stores the first of element of the array
2. (**1** 2 5 3 5 7) -> (1 **2** 5 3 5 7) Now it will compare the stored element with the second one, if this is bigger than the stored, it replaces the stored element by this
3. (1 **2** 5 3 5 7) -> (1 2 **5** 3 5 7) Repeats step 2
4. (1 2 **5** 3 5 7) -> (1 2 **5** 5 7) Since the 4th element is smaller than the 3rd one, the 4th element will be eliminated.
5. (1 2 **5** 5 7) -> (1 2 5 **5** 7) Equal elements are preserved
6. **(1 2 5 5 7)** Ordered array!

Quantum Bogo Sort

QuantumBogoSort a quantum sorting algorithm which can sort any list in O(1), using the "many worlds" interpretation of quantum mechanics.

It works as follows:

1. Quantumly randomise the list, such that there is no way of knowing what order the list is in until it is observed. This will divide the universe into O(n!) universes; however, the division has no cost, as it happens constantly anyway.

2. If the list is not sorted, destroy the universe. (This operation is left as an exercise to the reader.)

3. All remaining universes contain lists which are sorted.

Any Questions?