Lec 17: Sorting Wrapup; Stacks

Sam McCauley

March 20, 2026

Insertion Sort

Insertion Sort

- Similar to Selection Sort
- Still $O(n^2)$, but significantly more efficient in practice (we'll come back to this)
- This time we'll start with why it works, and derive the algorithm

Insertion Sort

- A different approach to sorting
- After the kth loop, the first k items in the array are sorted
  - The first k items may not be the smallest—but they are in sorted order
- How can we guarantee this for $k = 1$?
  - Don't need to do anything
- Let's say it works for k. What does the $k + 1$st loop need to accomplish to maintain the invariant?

  - Needs to insert the $k + 1$st item among the first k items in sorted order.

A Beautiful Way to Accomplish This

- Want to take the new item and move it into sorted position
- Idea: need to move it down until the previous element is smaller
- Inner loop: store element we are trying to insert. Shift elements down while it is smaller.

Insertion Sort Code

```
1  public static void insertionSort(int data[]) {
2      int numSorted = 1; // number of values in place
3      int index; // general index
4      while (numSorted < data.length) {
5          int temp = data[numSorted]; // first unsorted value
6          fo                                              {
7
8
9              } else {
10                 break;
11             }
12         }
13         data[index] = temp; // reinsert value
14         numSorted++;
15     }
16 }
```

Can we get rid of the `break` command in this code?

Insertion Sort Code # 2

```
1  public static void insertionSort(int data[]) {
2      int numSorted = 1; // number of values in place
3      while (numSorted < data.length) {
4          int temp = data[numSorted]; // first unsorted value
5          int index = numSorted;
6          while(index > 0 && temp < data[index - 1]) {
7              data[index] = data[index-1];
8              index--;
9          }
10         data[index] = temp; // reinsert value
11         numSorted++;
12     }
13 }
```

Tradeoff with Selection Sort

- No swap method needed
- Code is a little shorter
- Efficiency?
  - Both take n iterations of the outer loop. What about the inner loop?
  - Selection sort *always* iterates through $n - i$ elements on the ith iteration
  - Insertion sort may stop early! Can lead to better performance in practice (and is never worse)
- To be clear: both are still $O(n^2)$ in terms of worst-case performance. Insertion sort just has better constants, and better best-case performance

Tradeoff with Merge Sort (Just for fun)

- Merge Sort is better than Insertion Sort for large n
- Less clear for small n!
  - Insertion sort: $n^2/2$ worst case; $n^2/4$ “on average”
  - Merge sort: $2n \log_2 n$
  - Tradeoff point n somewhere in the range 32–256
- Bears out in practice: insertion sort is better on small lists
- A well-implemented Merge Sort switches to Insertion Sort as a base case

Stacks

Stack

- Can only add or remove to the *top* of the stack
- (No adding to the middle or removing from the middle.)
- The item we remove is the *most recently added* item

Stack Operations

Stacks have their own vocabulary. (This is fairly universal; you do need to know it.)

- `push()`: Add a new item to the top of the stack
  - Think `addLast()`
- `pop()`: Remove (and return) the top item on the stack
  - Think `removeLast()`
- `peek()`: Return the top item on the stack without removing
  - Think something like `get(size() - 1)`

Stack Operations

There are a few ways to implement a stack: it could be that we're adding and removing the first element!

- `push()`: Add a new item to the top of the stack
  - *Alternative:* `addFirst()`
- `pop()`: Remove (and return) the top item on the stack
  - *Alternative:* `removeFirst()`
- `peek()`: Return the top item on the stack without removing
  - *Alternative:* `get(0)`

Stack

- Only three operations to worry about!
- How can we implement a stack using data structures we already have?
- What are the tradeoffs between some of the options we have?

Implementing a Stack with an Array

- On board: How would this work?
- Downside?
  - Need to specify maximum items in the stack up front.
- Running time for operations?
  - `push()`: $O(1)$
  - `pop()`: $O(1)$
  - `peek()`: $O(1)$

Implementing a Stack with an ArrayList

- On board: How would this work?
- Don't need `int top` anymore; can resize
- Running time for operations?
  - `push()`: $O(n)$ in the worst case. $O(1)$ "on average"!
  - `pop()`: $O(1)$
  - `peek()`: $O(1)$

Implementing a Stack with a Linked List

- In pairs: how can we implement a stack with a Singly Linked List?
- Idea: need to store it backwards! Most recent element is the *head*
- to `push()` we can `addFirst()` which is $O(1)$
- to `pop()` we can `removeFirst()` which is $O(1)$

When to Use Stacks?

- Classic example: “JVM call stack”!
  - JVM: the Java Virtual Machine is what actually runs our code
  - Keeps track of what methods we have called
  - Each time a new method is called, we push it on the top of the stack
  - When the method returns, pop it off the top of the stack
- Useful in implementing backtracking search
- Or any last-in-first-out usage