CSCI 136: Data Structures and Advanced Programming
Lecture 16
Linear structures
Instructor: Kelly Shaw Williams

Topics
- Quicksort
- Linear ADTs
- Stack ADT

Your to-dos
1. Lab 6 (partner lab), due Tuesday 11/1 by 10pm. (two weeks!)
2. Read before Fri: Bailey, Ch 8-8.3.

Announcements
- Colloquium: What I Did Last Summer (Industry), 2:35pm in Wege Auditorium with cookies.
- Midterm: in lab two weeks from now: Wed, October 26 and Thu, October 27 and
- Midterm review: Mon, October 24 in class.
- No class: Fri, October 28.
Quicksort is a sorting algorithm that uses the divide and conquer technique. It works by partitioning the data into two arrays around a pivot (a fixed element, like the first element).

It swaps data so that one array contains elements smaller than the pivot and the other array contains elements larger than the pivot. This ensures that, at each step, the pivot is in the correct position in the array.

Performing this procedure recursively on the left and right subarrays until there is nothing left to partition guarantees a sorted array.
Quicksort

Unlike merge sort, quick sort does not need to combine sub arrays after splitting—the entire array is guaranteed to be sorted upon reaching the base case, and since the sort is done in-place no copying is required.

**Base case** (array of size 1): the pivot is trivially sorted.

**Inductive case:** Assume that the left and right subarrays are sorted. Since the pivot is the middlemost element, then everything to the left is smaller and everything to the right is bigger. Therefore, the entire array is sorted.

Quicksort time proof sketch

In the **worst case**, we repeatedly choose the worst pivot (either the min or max value in the array). This means that the array shrinks by 1 on every call to partition, so we need to do **n partitions**, doing one fewer comparison on each partition. This is the sum of 1 to n. \(O(n^2)\).

In the **best case**, we always happen to choose the **middlemost value** as a pivot. I.e., the two subarrays are the same size. The rest of the proof looks just like the proof for merge sort where we intentionally choose two subarrays of the same size.

If you’re thinking that quicksort’s best case is the same as merge sort’s worst case, remember that quicksort is in-place.

Quicksort

Quicksort takes \(O(n^2)\) time in the **worst case**. This case is improbable, and highly improbable as \(n \to \infty\).

Quicksort takes \(O(n \log n)\) time in the **best case**.

Quicksort takes \(O(n \log n)\) time in the **average case**.

I.e., quicksort is an **in-place sort**. Therefore it needs no auxiliary space. As a result, quicksort is almost always chosen over merge sort in any application where all the data can fit into RAM.

### Sorting Wrapup

<table>
<thead>
<tr>
<th></th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble</td>
<td>Worst: (O(n^2))</td>
<td>(O(n) : n + c)</td>
</tr>
<tr>
<td></td>
<td>Best: (O(n)) - if &quot;optimized&quot;</td>
<td></td>
</tr>
<tr>
<td>Insertion</td>
<td>Worst: (O(n^2))</td>
<td>(O(n) : n + c)</td>
</tr>
<tr>
<td></td>
<td>Best: (O(n))</td>
<td></td>
</tr>
<tr>
<td>Selection</td>
<td>Worst = Best: (O(n^2))</td>
<td>(O(n) : n + c)</td>
</tr>
<tr>
<td>Merge</td>
<td>Worst = Best: (O(n \log n))</td>
<td>(O(n) : 2n + c)</td>
</tr>
<tr>
<td>Quick</td>
<td>Average = Best: (O(n \log n))</td>
<td>(O(n) : n + c)</td>
</tr>
<tr>
<td></td>
<td>Worst: (O(n^2))</td>
<td></td>
</tr>
</tbody>
</table>
Recap & Next Class

Today:

• Sort stability
• Merge sort
• Quick sort

Next class:

• Linear structures

Recall: Abstract Data Type

An abstract data type is a mathematical formulation of a data type. ADTs abstract away accidental properties of data structures (e.g., implementation details, programming language). Instead, ADTs contain only essential properties and are concisely defined by their logical behavior over a set of values and a set of operations.

In an ADT, precisely how data is represented on a computer does not matter.

By contrast: data structure

A data structure is the physical form of a data type, i.e., it is an implementation of an ADT. Generally, data structures are designed to efficiently support the logical operations described by the ADT.

For data structures, precisely how data is represented on a computer matters a lot. Simple data structures are often composed of simple representations, like primitives, while more complex data structures are composed of other data structures.

ADT example: List

A list is a sequential collection of data elements, whose order is not necessarily given by their placement in memory. Elements may store any type of value. A list supports inserting, searching for, and deleting any value in a list, at any location, although not necessarily efficiently.
A linear ADT is one that presents elements in a sequence, even if the elements are not actually stored that way.

In a linear ADT, adding and removing elements is constrained, meaning that the structure can only be inspected and modified according to certain rules.

We will talk about two this week: stack and queue.

A stack is an abstract data type that stores a collection of any type of element. A stack restricts which elements are accessible: elements may only be added and removed from the "top" of the collection. The "push" operation places an element onto the top of the stack while a "pop" operation removes an element from the top.

Also sometimes referred to as a LIFO: “last in, first out.”

We also sometimes include a "peek" operation that lets us look at an element on the top of a stack without removing it, and "size" and "isEmpty" operations that let us check how many elements are stored and whether a stack stores zero elements, respectively.
Stack ADT

Interesting history: first appeared in print in a paper by Alan Turing (1946).

Unclear if he actually invented it.

push = bury, pop = unbury.

Application: Arithmetic

A computer can perform arithmetic using a stack.

E.g., $1 + 2 * 3 = 7$

Small problem: order of operations in infix arithmetic depends on the operations themselves.

In postfix arithmetic, order is always the same: left to right

E.g., $1 2 3 * +$

Once in this form, processing is easy. (Example)

Activity: Arithmetic

Convert infix to postfix: $x*y+z*w$

1. Add parens to preserve order of operations:
   $$((x*y)+(z*w))$$

2. Move all operators to the end of each parenthesized expression:
   $$((x*y)(z*w)) +$$

3. Remove parens:
   $$x*y*z*w*+$$

Evaluate these using a stack:

1. $4 + 1 * 8$
2. $5 * (6 + 2) - 12 / 4$
Cool application: function evaluation

```
class Meowww {
    public static String wwow(int n) {
        if (n == 0) {
            return "wow";
        }
        if (n == 1) {
            return "w";
        }
        return wwow(n-1) + wwow(n-2);
    }
    public static String meowww(int n) {
        return "meo" + wwow(n);
    }
    public static void main(String[] args) {
        int n = Integer.valueOf(args[0]);
        System.out.println(meowww(n));
    }
}
```

Cool application: backtracking search

Search strategy: straight, left, right
Decision point: ● Dead end: ●

```
```

Search strategy: straight, left, right
Decision point: ● Dead end: ●

Turn stack
Search strategy: straight, left, right
Decision point: ● Dead end: ●

Turn stack

straight
straight

Turn stack

straight

straight

left

straight

left

straight

Turn stack
Search strategy: straight, left, right
Decision point:  ●  Dead end:  ●

Turn stack

Search strategy: straight, left, right
Decision point:  ●  Dead end:  ●

Turn stack

Search strategy: straight, left, right
Decision point:  ●  Dead end:  ●

Turn stack

Search strategy: straight, left, right
Decision point:  ●  Dead end:  ●

Turn stack
Search strategy: straight, left, right
Decision point:  ❄️  Dead end:  ⚫

Turn stack

straight
left
right
straight

straight
left
right
straight

straight
left
right
straight

left
straight
right
left
straight
Search strategy: straight, left, right
Decision point: ● Dead end: ●

Turn stack

Search strategy: straight, left, right
Decision point: ● Dead end: ●

Turn stack

Search strategy: straight, left, right
Decision point: ● Dead end: ●

Turn stack

Search strategy: straight, left, right
Decision point: ● Dead end: ●

Turn stack
Search strategy: straight, left, right
Decision point:  ●  Dead end:  ●

Turn stack

Search strategy: straight, left, right
Decision point:  ●  Dead end:  ●

Turn stack

Search strategy: straight, left, right
Decision point:  ●  Dead end:  ●

Turn stack

Search strategy: straight, left, right
Decision point:  ●  Dead end:  ●

Turn stack
The saga continues…
Stack data structures

StackArray

A StackArray is a stack implemented using an array for element storage.

Pros: push and pop are $O(1)$ operations.

Cons: data structure has a maximum capacity.

Stack data structures

StackVector

A StackVector is a stack implemented using a Vector for element storage.

Pros: push and pop are amortized $O(1)$ operations. There is no maximum capacity.

Cons: Most of the time, ops take $O(1)$ time, but occasionally (when the underlying array needs to grow) an $O(n)$ cost is incurred. This may be fine for most applications, but if the application cannot tolerate wide variation in time, this is a bad choice.

Also, unless the underlying array is completely full, Vectors waste some space.

Stack data structures

StackList

A StackList is a stack implemented using a List (usu. SLL) for element storage.

Pros: push and pop are $O(1)$ operations. There is no maximum capacity, and no wasted space. push and pop costs are predictable (always the same), unlike StackVector.

Cons: because of the way computer hardware is implemented, a StackList's constant-time cost is likely to be much higher than a StackVector's. So a StackList's performance may be more predictable than a StackVector, but it will likely be slower on average.
Let's look at **StackList**

Uses an SLL for storage.

Adding an element puts it at the front of the list.
Uses an SLL for storage.

Adding an element puts it at the front of the list.

Wait! What about push?

Push just calls add.

Pop just calls remove.
Recap & Next Class

**Today:**

Linear ADTs
Stack

**Next class:**

Queue, etc.