Today

- Quick sort
- Hash tables (intro)

Assignments

- HW7 out (due Tues)
Quick Sort

The basic idea (also a recursive “divide and conquer” approach)

- pick a “pivot” element (e.g., first element in list)
- put values smaller than pivot on left
- put values larger than pivot on right
- put pivot value in the middle
- sort the left and right halves (using quick sort)
The algorithm (psuedocode):

```python
def quick_sort(T array[], int first, int last):  # first, last indexes
    if first < last:
        pivot_val := array[first]
        last_p1 := first  # last index, first part
        for i = first + 1 to last:  # partition
            if array[i] < pivot_val:
                last_p1 := last_p1 + 1
                swap(array[i], array[last_p1])
        swap(array[first], array[last_p1])  # move pivot
        quick_sort(array, first, last_p1 - 1)  # recursive step
        quick_sort(array, last_p1 + 1, last)  # recursive step
```

In the partition step, what is the worst case?

- when pivot is the smallest value
- since we end up with only one partition
- giving one new partition of size $n - 1$

In general, the partition step has $O(n)$ comparisons/swaps

And we do the partition step $n - 1$ times in the worst case!

Therefore, quick sort is $O(n^2)$ in the worst case
However, quick sort is much better in practice …

- each step in the best case (and on average) partitions into equal-sized halves
- so we have approx. \( \log_2 n \) recursive calls (levels) … \( O(\log n) \)
- and at each level we have \( n \) comparisons and swaps (see above)

Therefore, quick sort is \( O(n \log n) \) in the best (and average) case

<table>
<thead>
<tr>
<th>Sorting Alg</th>
<th>Best Case</th>
<th>Avg Case</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Sort</td>
<td>( O(n^2) )</td>
<td>( O(n^2) )</td>
<td>( O(n^2) )</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>( O(n) )</td>
<td>( O(n^2) )</td>
<td>( O(n^2) )</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>( O(n) )</td>
<td>( O(n^2) )</td>
<td>( O(n^2) )</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>( O(n \log n) )</td>
<td>( O(n \log n) )</td>
<td>( O(n \log n) )</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>( O(n \log n) )</td>
<td>( O(n \log n) )</td>
<td>( O(n^2) )</td>
</tr>
<tr>
<td>Tree Sort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap Sort</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Note on Mergesort versus Quicksort:

- quicksort has no "merge" step (which requires extra space)
- each level in quicksort divides list by \((n - 1)/2\) as opposed to \(n/2\) halves
- these make quicksort faster in practice
- also can improve worst case performance by picking a "better" pivot val
  - takes a bit of extra time, but worth the trade off in practice

Quick Sort for HW 6

- Again, implement quick sort by splicing and reattaching nodes

We'll use a helper function:

```c
Node* quick_sort(Node* start, int len)
```

Function does both partitioning and recursive step

**General High-Level Algorithm**

```c
Node* quick_sort(Node* start, int len)
1. if len <= 1 then return start
2. if len == 2 then reorg as needed and return start
3. separate first node from rest of list (pivot node)
4. move nodes into a smaller and larger list (partition)
5. smaller := quick_sort(smaller, smaller_len)
6. larger := quick_sort(larger, larger_len)
7. attach smaller, pivot, and larger together
8. return head node of resulting list
```

```c
void quick_sort()
1. head := quick_sort(head, length)
2. update tail pointer
```
Hash Tables

A hash table is a data structure for $O(1)$ retrieval

- For kv-collections this means
- ... fast find (value)
- ... fast remove
- ... fast add (compared to binary search)

The basic idea:

- Keep an array of elements ("buckets")
- Define a "hash" function $h : \text{Key} \rightarrow \text{Index}$
- Make $h$ fast relative to $n$ (i.e., $h$ should be constant time relative to $n$)
Called a hash “table” since ...

- Keys can refer to more complex values (like in our key-value pairs)
- The notion of a "key" is as identifier for the complex value
- For example, an Employee record as the value:
  - (ssn, first_name, last_name, birth_date, department, age, ...)
  - Q: What would the "key" be?
- The array represents a “table” of such records

Challenges when it comes to defining Hash Tables

- What is a “good” hash function?
- How do we define a good hash function?
- How do we deal with the fixed size nature of an array?

We’ll look at each of these ...