COMP20007 Design of Algorithms

Week 2 Workshop Solutions

Tutorial

1. Arrays

(i) Using a sorted array

- Inserting a new element: $O(n)$
  First, find the right place for the new element with linear or binary search. Then, make a free space here in the array by shifting all larger elements over by one space. Put the new element in this free space.
- Searching for a specified element: $O(\log n)$
  Use binary search.
- Deleting the final element: $O(1)$
  Just remove the element from the end of the array.
- Deleting a specified element: $O(n)$
  Find the element with linear or binary search, remove it, and then fill the free space by shifting all larger elements over by one space.

(ii) Using an unsorted array

- Inserting a new element: $O(1)$
  Since there's no need to maintain sorted order, just put the new element at the end of the array.
- Searching for a specified element: $O(n)$
  Since the array isn't sorted, binary search won't work. We might need to scan the entire array.
- Deleting the final element: $O(1)$
  Just remove the element from the end of the array.
- Deleting a specified element: $O(n)$
  Find the element with linear search (binary search won't work), remove it, and then fill the free space with the element from the end of the array.

2. Linked lists In all of these operations, ensure that the link(s) in the inserted/removed node, the link(s) in the next and previous nodes, and the links to the first and last elements of the list are all updated.

Don't forget the case where you are inserting an element into an empty list, or removing the only element in a list. In these cases both the link to the start of the list and the link to the end of the list need to be updated.

All of these operations on singly-linked and doubly-linked lists are $O(1)$ except for deleting the last element from a singly-linked list. In this case, to find the new last element and update the link to the end of the list, we need to follow the list all the way from the start to the second last element. (This isn't the case in a doubly-linked list, because the last element in a doubly-linked list has a direct link to the second last element.)

3. Stacks In an unsorted array, `push` by adding elements to the end of the array, and `pop` by removing them from the end of the array (both of these operations can be done in constant time).

In a singly-linked list, `push` by adding elements to the start of the list, and `pop` by removing them from the start of the list (it's best to use the start of the list because removing from the end of the list requires $O(n)$ time).

4. Queues In an unsorted array, `enqueue` by adding elements to the end of the array, and `dequeue` by removing them from the start of the array.

Removing from the start of the array would usually take linear time, because we'd need to move all of the remaining elements one space closer to the start to fill the free space we made. We can avoid doing this if we let the starting index of the queue drift as we repeatedly dequeue elements. We can then also let the end of the queue wrap back around to the start of the array, as if the array was a circle. We just need to carefully keep track of where the queue starts and ends as elements are enqueued and dequeued.

In a singly-linked list, `enqueue` by adding elements to the end of the list, and `dequeue` by removing them from the start of the list.

Adding at the end and removing from the start allows both operations to run in constant time, compared with adding at the start and removing from the end. Assuming our list needs to keep track of its last element, removing from the end takes linear time in a singly-linked list.

5. Bonus problem (optional) With access only to stack operations, one way to create queue operations is as follows:

- For `enqueue`, add the new element to the top of a stack using `push`.
- For `dequeue`, we need the item that was enqueued first, which will be at the bottom of the stack. `pop` items from the stack and `push` them into a second stack until `size` of the first stack is 1. `pop` this last element. Before returning it, `pop` the elements off the second stack and `push` them back onto the first stack.

With access only to queue operations, one way to create stack operations is as follows:

- For `push`, add the new elements to the end of a queue using `enqueue`.
- For `pop`, we need the item that was enqueued last, which will be at the end of the queue. `dequeue` the items from the queue and `enqueue` them into a second queue until the first queue has `size` 1. `dequeue` and return this last element. There's no need to put the elements in the second queue back into the first queue; they are already in the right order in the second queue, so we can `push` into the end of the second queue from this point onwards (until we swap again, next time we `pop`).

Note that for both of these approaches insertion (*i.e.,* `enqueue` and `push` respectively) is $O(1)$, while removal (*i.e.,* `dequeue` and `pop`) is an $O(n)$ operation.