MATH182 DISCUSSION 4 WORKSHEET

1. Heaps

Exercise 1. In this exercise we find the best-case running time of heapsort. Let A be an array of length n, with all elements distinct, on which we will run HeapSort.

(a) After making a max-heap, HeapSort swaps $A[1]$ and $A[n] =: \alpha$. Suppose that, at some later time during the running of HeapSort, we observe that α is on level l of the heap. What can we deduce about the number of swaps which have occurred involving α?
(b) Show that in any max-heap, for any $m \leq n$, at least $\lceil m/2 \rceil$ of the m smallest elements occur at leaves.
(c) Suppose that HeapSort has pulled off $\lceil n/2 \rceil$ elements into the sorted portion, so the heap has $\lfloor n/2 \rfloor$ elements. Use (b) to show that at least $n/4$ elements have been swapped into the root of the heap, and are still in the heap.
(d) How many of those $n/4$ elements can be in the top $\frac{\lg n}{2}$ levels of the heap?
(e) Use (a) to deduce that HeapSort has $\Omega(n \lg n)$ running time even in the best case.
(f) If we don't assume that the elements of the array are distinct, what is the best-case running time?

Exercise 2. When we wrote Build-Max-Heap, we started by putting all of the desired elements into the heap, then continually modified it until it had the max-heap property. We can instead add one element at a time:

```
Build-Max-Heap-2($A$)
1  $A.heap\text{-}size = 1$
2  for $i = 2$ to $A.length$
3        Max-Heap-Insert($A, A[i]$)
```

(a) Without doing any examples or careful analysis, do you expect Build-Max-Heap-2 to create the same max-heap as Build-Max-Heap when run on the same input array?
(b) Now try to verify your guess, by a proof or counterexample. (*Hint:* There is no reason that a counterexample should be large or complicated. If searching for a counterexample, you probably don't need to try anything with more than 5 elements.)
(c) What is the worst-case running time of Build-Max-Heap-2?

Exercise 3. Write an algorithm which takes as input two max-heaps, and outputs a max-heap with the same elements as the input. What is its running time? Do you think that this running time is the best possible, or could we do better by implementing heaps differently?

2. Lists

Recall that a *stack* is a list in which you can add or remove elements from the front, and a *queue* is a list in which you can add elements to the back and remove elements from the front.

Exercise 4. (a) Use two queues to implement a stack. Analyze the running times of the stack operations (they may be slow).
(b) Use two stacks to implement a queue. Analyze the running times of the queue operations.

A *(singly) linked list* is a data structure in which elements are associated only by pointers: each element contains a value, and a pointer to the next element. The last element points to a special terminator instead. In a *doubly linked list*, each element contains a pointer to the previous element as well as the next. As a data structure, a linked list contains a pointer to the first node in the list.

Exercise 5. How long do the following operations take in the worst case? Answer for both singly and doubly linked lists You do not need to prove your answers.

(1) Determining the length of a list.
(2) Determining the position of a given element in the list.
(3) Determining the predecessor a given element.
(4) Inserting a new element after a given element.
(5) Inserting a new element before a given element.
(6) Inserting a new element at the beginning of the list.
(7) Inserting a new element at the end of the list.
(8) Deleting a given element.
(9) Deleting the next element.
(10) Concatenating two lists.

3. Trees

Recall that a *(rooted) tree* consists of a (finite) collection of *nodes*, or *vertices*, one of which is the *root*, such that

(1) Each node x other than the root has a unique *parent* $\text{parent}(x)$.
(2) For any node x, $\text{parent}^n(x)$ is the root for some n, where parent^n denotes n applications of the parent map.

If node x is the parent of node y, then y is called a *child* of x. A *leaf* is a node with no children.

The above definition is very clean mathematically, but often not especially useful. There are various other ways to define trees. One which provides a useful perspective for algorithmic purposes is the recursive definition: a tree consists of a root and a (possibly empty) list of children, which are the roots of disjoint trees.

Exercise 6. Prove that the two definitions above are equivalent.

The *depth* of a node in a tree is its distance from the root, i.e. the number of times parent must be applied before reaching the root. The *height* of a tree is the maximum depth of any of its nodes. The height of a node is the height of the subtree rooted at that node.

Very often, trees are also *ordered*: for every node, the children of that node have a definite ordering.

A tree is a *binary* tree if nodes have at most two children. A *binary search tree* is an ordered binary tree such that the value of any given node is greater than all nodes in the subtree rooted at its left child, and less than all nodes in the subtree rooted at its right child.

Exercise 7. Describe algorithms which perform the following operations on a binary search tree, and analyze their running times in terms of the size and height of the tree.

(1) Binary search: given a value, find its location in the tree.
(2) In-order traversal: prints the values of the nodes of a tree, in sorted order. (*Challenge:* do this using only $O(1)$ extra memory.)
(3) Delete a node.
(4) Insert a node with a given value.