CS 253
Section 02
Fall 2019
Christopher Schmitt
Assignment 4

Problem 1

Table 1: Toy Data (0-9)

	Best Case		Average Case		Worst Case	
Algorithm	Exchanges	Comparisons	Exchanges	Comparisons	Exchanges	Comparisons
Bubble Sort	0	9	20	43	45	45
Selection Sort	9	45	9	45	9	45
Insertion Sort	0	10	20	30	45	55
Merge Sort	24	40	32	56	28	48
Radix Sort	40	0	40	0	40	0
Heap Sort	21	30	18	30	12	30

Table 2: Test Data (0-1999)

	Best Case		Average Case		Worst Case	
Algorithm	Exchanges	Comparisons	Exchanges	Comparisons	Exchanges	Comparisons
Bubble Sort	0	1999	1004333	1997649	1999000	1999000
Selection Sort	1999	1999000	1999	1999000	1999	1999000
Insertion Sort	0	2000	1004333	1006333	1999000	2001000
Merge Sort	12863	23728	21430	40862	13087	24176
Radix Sort	20000	0	20000	0	20000	0
Heap Sort	19301	6000	18190	6000	16709	6000

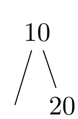
Like merge sort, heap sort runs in $\mathcal{O}(n \log n)$. It is similar to selection sort in the sense that we find the maximal element and place it at the end of the list. We can then repeat this process untill the entire collection is sorted. This is made effecient by the underlying heap, which can be recomputed in $\prime(\log n)$ operations. Since this needs to happen once for each element that needs to be sorted, we get $\mathcal{O}(n \log n)$.

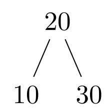
Merge sort exhibits good performance characteristics when the data is in order, or close to in-order. Heap sort shows superior characteristics for both average data and reverse ordered data. It should be noted however, that when dealing with data at this small scale, most of these algorithms will do fine. When dealing with larger sets, merge sort can take the most advantage of multiple threads of execution. We can safely do this since array forms a monoid (with an empty array serving as the identety value). Since monoids obey right-identety, left-identety, and accociativity rules, each thread can operate on a sub-array independently and can be recombined safely at the end of the operation.

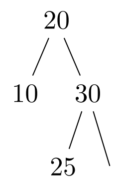
Problem 2

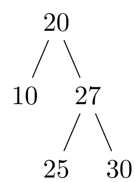
Constructing the AVL tree

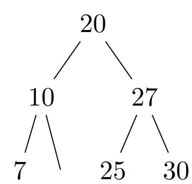
10

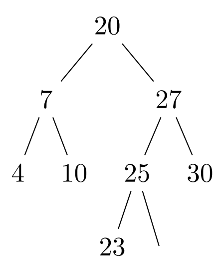


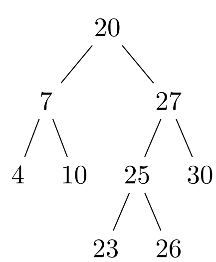

20
10
30

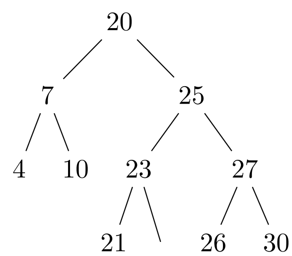

20
10
30
25


20
10
27
25
30

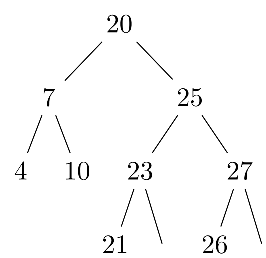

20
10
27
7
25
30

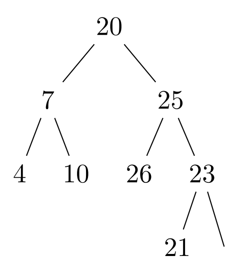

20
7
27
4
10
25
30
23

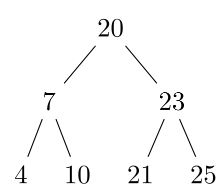

20
7
27
4
10
25
30
23
26



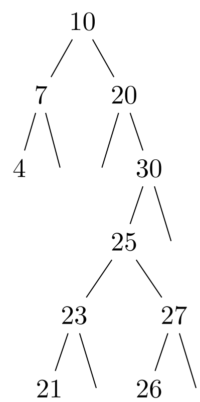
Removing elements from the tree







Binary search tree



This tree is severly unbalanced, and will therefore exhibit worse performance characteristics (in search) to the AVL tree. The ballance of the binary search tree is determined entirely by the order of the inputs. The AVL tree however, has a mechanism to preserve the balance of the tree, ensuring search effecincy near $\mathcal{O} \log n$.