Data Structures Exam #1

1. See Attached Sheet
2. Breadth First and Depth First
3. Breadth First :

100, 32, 3, 46, 2, 30, 38, 77, 16, 31, 41, 70, 84, 8, 25, 45, 66, 81, 96, 12, 43, 54, 68, 86, 65

Depth First:

100, 32, 3, 2, 30, 16, 8, 12, 25, 31, 46, 38, 41, 45, 43, 77, 70, 66, 54, 65, 68, 84, 81, 96, 86

1. Set 100 as root node

Add 32 as left node

Switch values 32 and 100

Add 46 as 32’s right node

Add 77 as 100’s left node

Swap values 77 and 100

Add 84 as 77’s right node

Add 70 as 46’s left node

Add 66 as 46’s right node

Add 81 as 100’s left node

Swap values 81 and 100

Add 54 as 81’s right node

Swap values 54 and 81

Swap values 54 and 77

Add 38 as 84’s left node

Swap values 38 and 84

Swap values 38 and 54

Add 96 as 54’s right node

Add 3 as 70’s left node

Swap values 3 and 70

Swap values 3 and 46

Swap values 3 and 32

Add 89 as 46’s right node

Add 30 as 66’s left node

Swap values 30 and 66

Swap values 30 and 32

Add 68 as 32’s right node

Add 16 as 100’s left node

Swap values 16 and 100

Swap values 16 and 77

Swap values 16 and 38

Add 25 as 77’s right node

Swap values 25 and 77

Swap values 25 and 38

Add 41 as 81’s left node

Swap values 41 and 81

Add 31 as 41’s right node

Swap values 31 and 41

Add 45 as 84’s left node

Swap values 45 and 84

Swap values 45 and 54

Add 8 as 54’s right node

Swap values 8 and 54

Swap values 8 and 45

Swap values 8 and 16

Add 43 as 96’s left node

Swap values 43 and 96

Add 12 as 43’s right node

Swap values 12 and 43

Swap values 12 and 16

Add 2 as 70’s left node

Swap values 2 and 70

Swap values 2 and 46

Swap values 2 and 30

Swap values 2 and 3

Add 65 as 46’s right node

END ARRAY - - - SEE ATTACHED SHEET FOR TREE

1. Heap sort uses a heap, a tree like data structure with special rules, to sort an array of numbers. Of course first the array has to be entered into the heap following the general rules. To get an ascending sort, you must use a max heap and to get a descending sort, you must use a min heap. The algorithm is simple, take the top most node in the heap and switch with the last node. That node is now sorted so you can consider your array 1 unit smaller so that it doesn’t touch that node anymore. After that, you apply the rules of the heap to the whole heap, moving the node that was moved to the head down to a spot where the heap rules stand. Continue until you have moved all nodes.
2. QuickSort Step-through

Initial Array: 100, 32, 46, 77, 84, 70, 66, 81, 54, 38, 96, 3, 89, 30, 68, 16, 25, 41, 31, 45, 8,

43, 12, 2, 65

Choose 65 as initial pivot

100 stays on right

32 goes to left

46 goes to left

77 stays on right

84 stays on right

70 stays on right

66 stays on right

81 stays on right

54 goes to left

38 goes to left

96 stays on right

3 goes to left

89 stays on right

30 goes to left

68 stays on right

16 goes to left

25 goes to left

41 goes to left

31 goes to left

45 goes to left

8 goes to left

43 goes to left

12 goes to left

2 goes to left

Left Array 1: 32, 46, 54, 38, 3, 30, 16, 25, 41, 31, 45, 12, 2

Switch pivot with first element of right set

Right array 1: 77, 84, 70, 66, 81, 96, 89, 68, 100

Array Now:

32, 46, 54, 38, 3, 30, 16, 25, 41, 31, 45, 12, 2, **65**, 77, 84, 70, 66, 81, 96, 89, 68, 100

Left Array 1: 32, 46, 54, 38, 3, 30, 16, 25, 41, 31, 45, 12, 2

Left Array 1 Pivot is 2

32 stays on right

46 stays on right

54 stays on right

38 stays on right

3 stays on right

30 stays on right

16 stays on right

25 stays on right

41 stays on right

31 stays on right

45 stays on right

12 stays on right

Left Array 2: NULL

Switch pivot with first element of right set

Right Array 2: 46, 54, 38, 3, 30, 16, 25, 41, 31, 45, 12, 32

Left Array 2 Sorted due to null set

Array Now:

**2**, 46, 54, 38, 3, 30, 16, 25, 41, 31, 45, 12, 32

Right Array 2: 46, 54, 38, 3, 30, 16, 25, 41, 31, 45, 12, 32

Right Array 2 Pivot is 32

46 stays on right

54 stays on right

36 stays on right

3 moves to left

30 moves to left

16 moves to left

25 moves to left

41 stays on right

31 moves to left

45 stays on right

12 moves to left

Left Array 3: 3, 30, 16, 25, 31, 12

Switch pivot with first element of right set

Right Array 3: 54, 36, 41, 45, 46

Array Now:

3, 30, 16, 25, 31, 12, **32**, 54, 36, 41, 45, 46

Left Array 3: 3, 30, 16, 25, 31, 12

Left Array 3 Pivot is 12

3 moves to left

30 stays on right

16 stays on right

25 stays on right

31 stays on right

Left Array 4: 3

Switch pivot with first element of right set

Right Array 4: 16, 25, 31, 30

Left Array 4 Sorted due to 1 element

Array Now:

3, **12**, 16, 25, 31, 30

Right Array 4: 16, 25, 31, 30

Right Array 4 Pivot is 30

16 moves to left

25 moves to left

31 stays on right

Left Array 5: 16, 25

Switch pivot with first element of right set

Right Array 5: 31

Right Array 5 Sorted due to 1 element

Array Now:

16, 25, **30**, 31

Left Array 5: 16, 25

Left Array 5 Pivot is 25

16 moves to left

Left Array 6: 16

Pivot stays here due to null right set

Right Array 6: NULL

Left Array 6 Sorted due to one element

Right Array 6 Sorted due to null set

Array Now:

16, **25**

Right Array 3: 54, 36, 41, 45, 46

Right Array 3 Pivot is 46

54 stays on right

36 moves to left

41 moves to left

45 moves to left

Left Array 7: 36, 41, 45

Switch pivot with first element of right set

Right Array 7: 54

Right Array 7 Sorted due to 1 element

Array Now:

36, 41, 45, **46**, 54

Left Array 7: 36, 41, 45

Left Array 7 Pivot is 45

36 moves to left

41 moves to left

Left Array 8: 36, 41

Pivots stays here due to null right set

Right Array 8: NULL

Array Now:

36, 41, **45**

Left Array 8: 36, 41

Left Array 8 Pivot is 41

36 moves to left

Left Array 9: 36

Pivot stays here due to null right set

Right Array 9: NULL

Left Array 9 Sorted due to 1 element

Right Array 9 Sorted due to null set

Array Now:

36, **41**

Right array 1: 77, 84, 70, 66, 81, 96, 89, 68, 100

Right Array 1 Pivot is 100

77 moves to left

84 moves to left

70 moves to left

66 moves to left

81 moves to left

96 moves to left

89 moves to left

68 moves to left

Left Array 10: 77, 84, 70, 66, 81, 96, 89, 68

Pivot stays here due to null right set

Right Array 10: NULL

Right Array 10 Sorted due to null set

Array Now:

77, 84, 70, 66, 81, 96, 89, 68, **100**

Left Array 10: 77, 84, 70, 66, 81, 96, 89, 68

Left Array 10 Pivot is 68

77 stays on right

84 stays on right

70 stays on right

66 moves to left

81 stays on right

96 stays on right

89 stays on right

Left Array 11: 66

Switch pivot with first element of right set

Right Array 11: 84, 70, 81, 96, 89, 77

Left Array 11 Sorted due to one element

Array Now:

66, **68**, 84, 70, 81, 96, 89, 77

Right Array 11: 84, 70, 81, 96, 89, 77

Right Array 11 Pivot is 77

84 stays on right

70 moves to left

81 stays on right

96 stays on right

89 stays on right

Left Array 12: 70

Switch pivot with first element of right set

Right Array 12: 81, 96, 89, 84

Left Array 12 Sorted due to one element

Array Now:

70, **77**, 81, 96, 89, 84

Right Array 12: 81, 96, 89, 84

Right Array 12 Pivot is 84

81 moves to left

96 stays on right

89 stays on right

Left Array 13: 81

Switch pivot with first element of right set

Right Array 13: 89, 96

Left Array 13 Sorted due to one element

Array Now:

81, **84**, 89, 96

Right Array 13: 89, 96

Right Array 13 Pivot is 96

89 moves to left

Left Array 14: 89

Pivot stays here due to null right set

Right Array 15: NULL

Left Array 14 Sorted due to 1 element

Right Array 14 Sorted due to null set

Array Now: 89, **96**

Full sorted list

2, 3, 8, 12, 16, 25, 30, 31, 32, 38, 41, 43, 45, 46, 54, 65, 66, 68, 70, 77, 81, 84, 89, 96, 100

1. A hash function is a complex function that will take a key input and turn that input into an integer which will be that key’s index in a hash table’s array of data. A hashing function needs to be quick so that the search through a hash table is fast as well. A good hash function also has to produce unique indices for nearly every key that it can be put into a hash table; if this does not happen, we have the following occur.
2. Hash collisions are when the hashing function gives two completely different keys the exact same index. This problem comes from a hashing function that doesn’t create a complex enough hash or that the hash table has too small of a range of valid hash values. There are a couple ways that hash tables handle collision that I will explain in the following.
3. Linear probing happens when a hash table finds a collision, either on insertion or lookup. When inserting a key into a hash table and it runs into a collision, with linear probing, the hash table searches till it finds the next empty space in it’s array and inserts that key there. In lookup, the hash table does the same thing, but it will keep searching till the end of the array if it never finds the key.
4. Another way hash tables solve the issue of collision is instead of linear probing and just finding the next spot, with Separate Chaining, the hash table picks a set size and jumps that far into the array. If that spot is taken up, they jump another interval to locate or insert that key.