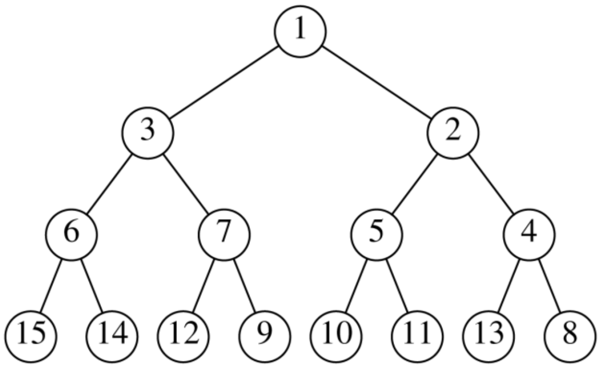
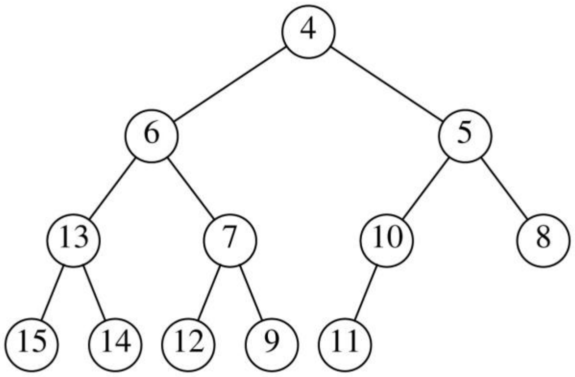
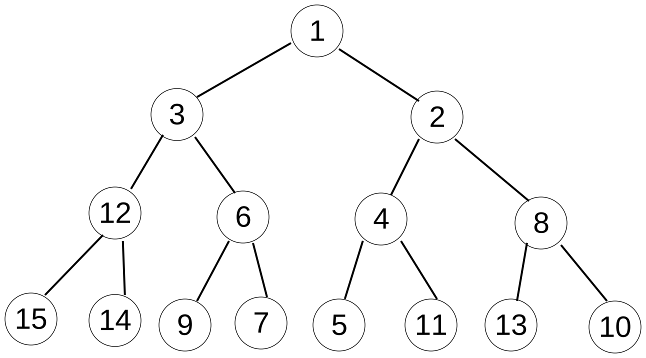
1a.


1
3
2
6
7
5
4
15
14
12
9
10
11
13
8


4
6
5
13
7
10
8
15
14
12
9
11

1b.


1
3
2
12
6
4
8
15
14
9
7
5
11
13
10

8. Hash Tables

a. Given the input:

255 375 254 639 718 101 100 692 538 800 277

a fixed table size of 10, and a hash function of H(x) = x mod 10, show the resulting separate chaining hash table. Why is 10 a bad choice for a table size?

What is the load factor of this table? What is the maximum recommended load factor for a table using separate chaining? What may happen if this load factor is exceeded?

0->100->800
1->101
2->692
3
4->254
5->255->375
6
7->277
8->718->538
9->639
Load Factor 1.1
Table Size should be a prime number
Recommend Load factor is 1

a. Given the input:
375 254 639 718 101 105 692 538
a fixed table size of 10, and a hash function of H(x) = x mod 10, show the resulting linear probing hash table. What is the load factor of this table? What is the maximum recommended load factor for a table using linear probing?
0: 538
1: 101
2: 692
3:
4: 254
5: 375
6: 105
7:
8: 718
9: 639

Load Factor .8
Recommend Load factor is < 0.7. Table gets exponentially slower at load factors above 0.7,

b. Given the input:
255 375 254 639 718 101 100 692 538 800 277
a fixed table size of 10, and a hash function of H(x) = x mod 10, show the resulting quadratic probing hash table.
Indicate if any inputs can't fit in the table. What is the load factor of this table? What is the maximum recommended load factor for a table using quadratic probing? What may happen if this load factor is exceeded?

0: 100
1: 101
2: 692
3: 277
4: 254
5: 255
6: 375
7: 538
8: 718
9: 639

800 does not fit into the table
Load Factor 1.0
Load factor must be < 0.5. Table gets exponentially slower at load factors above 0.5, and above 0.5, there is not guarantee that a spot will be found in the table.

What are the load factor limitations of the following?

a) Separate Chaining?
b) Linear Probing?
c) Quadratic Probing?

Discussed above.

```
// index 0    1    2   3    4    5   6    7    8
      { 16, 21, 45, 8, 11, 53, 3, 26, 49 }
```

9. Trace the execution of the **selection** sort algorithm over the array above. Show each pass of the algorithm and the state of the array after the pass has been performed, until the array is sorted.

```
{ 3, 21, 45, 8, 11, 53, 16, 26, 49 }
{ 3, 8, 45, 21, 11, 53, 16, 26, 49 }
{ 3, 8, 11, 21, 45, 53, 16, 26, 49 }
{ 3, 8, 11, 16, 45, 53, 21, 26, 49 }
{ 3, 8, 11, 16, 21, 53, 45, 26, 49 }
{ 3, 8, 11, 16, 21, 26, 45, 53, 49 }
{ 3, 8, 11, 16, 21, 26, 45, 53, 49 }
{ 3, 8, 11, 16, 21, 26, 45, 49, 53 }
```

```
// index 0    1    2   3    4    5   6    7    8
      { 16, 21, 45, 8, 11, 53, 3, 26, 49 }
```

10. Trace the execution of the **insertion** sort algorithm over the array above. Show each pass of the algorithm and the state of the array after the pass has been performed, until the array is sorted.

```
{ 16, 21, 45, 8, 11, 53, 3, 26, 49 }
{ 16, 21, 45, 8, 11, 53, 3, 26, 49 }
{ 8, 16, 21, 45, 11, 53, 3, 26, 49 }
{ 8, 11, 16, 21, 45, 53, 3, 26, 49 }
{ 8, 11, 16, 21, 45, 53, 3, 26, 49 }
{ 3, 8, 11, 16, 21, 45, 53, 26, 49 }
{ 3, 8, 11, 16, 21, 26, 45, 53, 49 }
{ 3, 8, 11, 16, 21, 26, 45, 29, 53 }
```

11. What are the best, average, and worst Big-Oh descriptions of selection sort and insertion sort?

Selection:

Best: O(N^2)

Average: O(N^2)

Worst: O(N^2)

Insertion:

Best: O(N)

Average: O(N^2)

Worst: O(N^2)

12. Merge Sort: Finish the merge sort code. Assume that you can call a merge function that takes 3 lists and will merge 2 of the lists into the other list.

```
def merge_sort(list1):
    if len(list1) > 1:
        mid = len(list1) // 2
        left_half = list1[:mid]
        merge_sort(left_half)
        right_half = list1[mid:]
        merge_sort(right_half)
        return merge(list1, left_half, right_half)
```