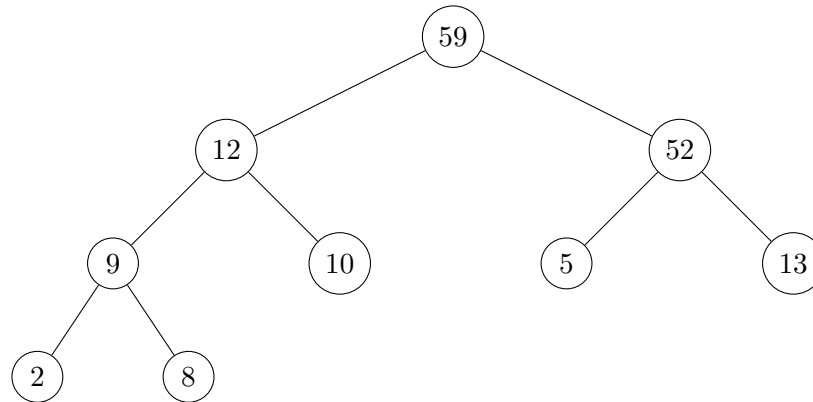


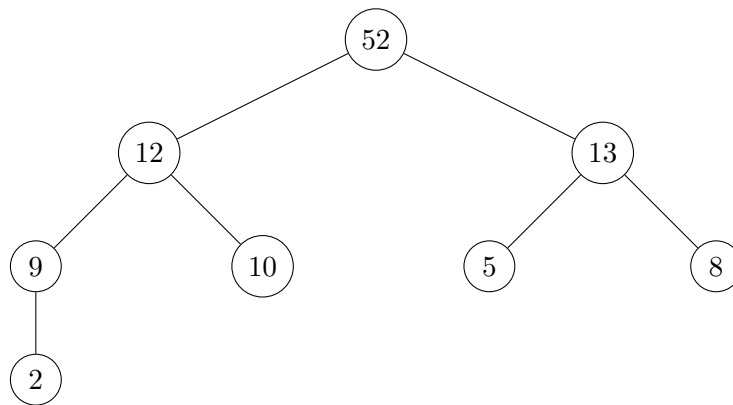
1.

a)



b)

you would remove 59 and replace it with 8 then heapify down until 8 settles into a spot.



2.

3.

a)

$$2(n - 2) + 1 = 2n - 3$$

b)

```
1  n1 = null;
2  n2 = new Node(A[0]);
3  for(var i = 1; i < A.length; i++){
4      n1 = new Node(A[i]);
5      n1.next = n2;
6      n2 = n1;
7  }
8  n3 = n4 = null;
9  newLevel = false;
10 //makes a graph of linked nodes in the shape of a pyramid, nodes on
    a level are linked one direction through the next pointer
11 //all nodes that get compared are linked, the larger value gets a
    clone in the next level with a pointer to it
12 // will complete after log_2 n levels are formed with the last
    level, ends when it starts on a level with only 1 element
13 while(!newLevel || n2.next){
14     newLevel = false;
15     n1 = n2.next;
16     if(n1){
17         n2.compare = n1;
18         n1.compare = n2;
19         if(n1.value > n2.value){
20             n3 = new Node(n1.value);
21             n3.parent = n1
22         }else{
23             n3 = new Node(n2.value);
24             n3.parent = n2;
25         }
26         n3.next = n4;
27         n4 = n3;
28         if(n1.next){
29             n2 = n1.next;
30         }else{
31             n4 = null;
32             n2 = n3;
33             newLevel = true;
34         }
35     } else{
36         newLevel = true;
37         n3 = new Node(n2.value);
38         n3.parent = n2;
39         n3.next = n4;
40         n4 = null;
41         n2 = n3;
42     }
43 }
44 largest = n2.value
45 n2 = n2.parent;
46 second = n2.compare.value;
47 while(n2.parent){
48     n2 = n2.parent;
49     if(n2.compare && n2.compare.value > second){
50         second = n2.compare.value;
51     }
52 }
```

the number of comparisons for the first while loop which finds the largest element is  $n - 1$ , and the number of comparisons to go back down the graph and find the second largest is  $\log_2(n - 1)$ , so the overall number of comparisons is  $n - 1 + \log_2(n - 1)$

4.

```

1      b = Array(n);
2      c = Array(k);
3      smallest = null;
4      sub = 0;
5      for(i = 0; i < k; i++){
6          c[i] = 0;
7      }
8      for(i = 0; i < n; i++){
9          smallest = a[c[0]*k];
10         sub = 0;
11         for(j = 1; j < k; j++){
12             if(a[c[j]*k + j] < smallest){
13                 smallest = a[c[j]*k + j];
14                 sub = j;
15             }
16         }
17         c[sub]++;
18         b[i] = smallest;
19     }

```

This runs in  $n * k$  time and it is essentially mimicking the second half of the merge sort algorithm since we start with  $k$  sorted sub arrays.

5.

```

1      cur_max = {start: 0, end: 0, sum: 0};
2      max = {start: 0, end: 0, sum: 0}
3
4      for(i = 0; i < a.length; i++){
5          if(cur_max.sum + a[i] < a[i]){
6              cur_max.start = i;
7              cur_max.end = i;
8              cur_max.sum = a[i];
9          } else{
10             cur_max.sum += a[i];
11             cur_max.end = i;
12         }
13         if(max.sum < cur_max.sum){
14             max.sum = cur_max.sum;
15             max.start = cur_max.start;
16             max.end = cur_max.end;
17         }
18     }

```

at the end of the algorithm “max” contains the max sum of a span of indices in  $a$ , as well as the starting and ending index of the span. the algorithm runs in  $O(n)$  since it loops through the array once and has constant time computation in the loop.