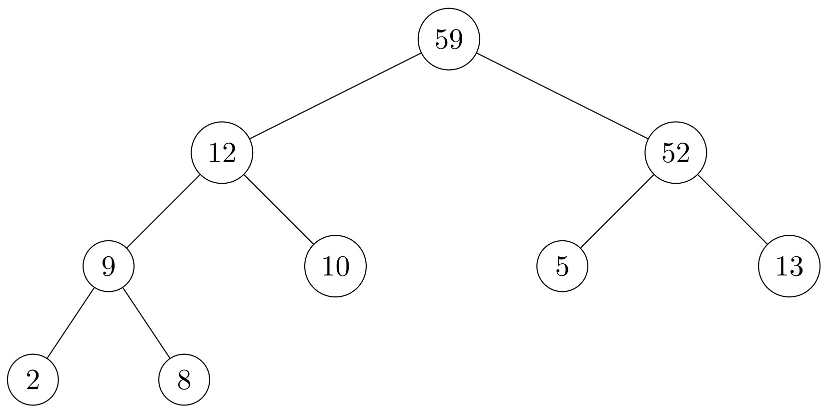
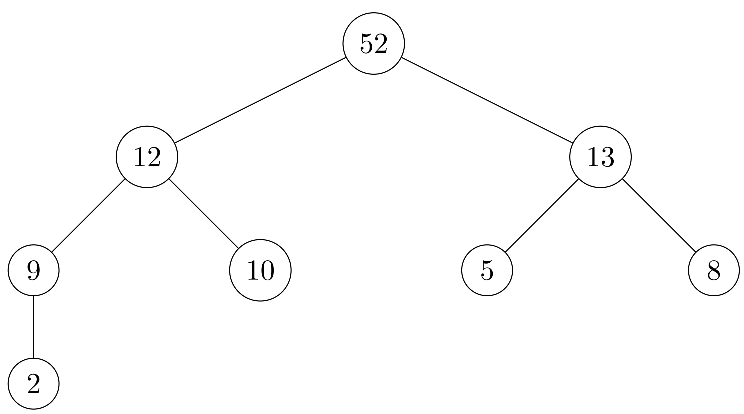
1.

a)



b)

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



2.

a)

$$
\begin{aligned}
T(n) &= cn + T(n/3) + T(2n/3) \\
&= cn + \frac{cn}{3} + \frac{2cn}{3} + T(\frac{n}{9}) + 2T(\frac{2n}{9}) + T(\frac{4n}{9}) \\
&= cn(1 + \frac{1}{3} + \frac{2}{3} + \frac{1}{9} + 2\frac{2}{9} + \frac{4}{9} \dots) \\
&= cn(1 + 1 + 1 \dots) \\
&= O(n \log n)
\end{aligned}
$$

b)

$$
\begin{aligned}
T(n) &= cn + T(\frac{n}{5}) \\
&= cn + \frac{cn}{5} + T(\frac{cn}{25}) \\
&= cn(1 + \frac{1}{5} + \frac{1}{25} \dots) \\
&= \frac{cn}{1 - \frac{1}{5}} \\
&= O(n)
\end{aligned}
$$

c)

$$
\begin{aligned}
T(n) &= n^{\log_5(7)} + 2T(\frac{n}{2}) \\
&= n^{\log_5(7)} + \frac{2n^{\log_5(7)}}{2^{\log_5(7)}} + 2T(\frac{n}{4}) \\
&= n^{\log_5(7)} + \frac{2n^{\log_5(7)}}{2^{\log_5(7)}} + \frac{2n^{\log_5(7)}}{4^{\log_5(7)}} \dots \\
&= n^{\log_5(7)}(1 + \frac{2}{2^{\log_5(7)}} + \frac{2}{4^{\log_5(7)}} \dots) \\
&= n^{\log_5(7)}(1 + \frac{1}{2^{\log_5(7)-1}} + \frac{1}{4^{\log_5(7)}} \dots) \\
&= \frac{n^{\log_5(7)}}{1 - \frac{1}{2}} \\
&= O(n^{\log_5(7)})
\end{aligned}
$$

3.

a)

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

b)

```
n1 = null;
n2 = new Node(A[0]);
for(var i = 1; i < A.length; i++){
  n1 = new Node(A[i]);
  n1.next = n2;
  n2 = n1;
}
n3 = n4 = null;
newLevel = false;
//makes a graph of linked nodes in the shape of a pyramid, nodes on a level are linked one direction through the next pointer
//all nodes that get compared are linked, the larger value gets a clone in the next level with a pointer to it
// will complete after log_2 n levels are formed with the last level, ends when it starts on a level with only 1 element
while(!newLevel || n2.next){
  newLevel = false;
  n1 = n2.next;
  if(n1){
    n2.compare = n1;
    n1.compare = n2;
    if(n1.value > n2.value){
      n3 = new Node(n1.value);
      n3.parent = n1
    }else{
      n3 = new Node(n2.value);
      n3.parent = n2;
    }
    n3.next = n4;
    n4 = n3;
    if(n1.next){
      n2 = n1.next;
    }else{
      n4 = null;
      n2 = n3;
      newLevel = true;
    }
  } else{
    newLevel = true;
    n3 = new Node(n2.value);
    n3.parent = n2;
    n3.next = n4;
    n4 = null;
    n2 = n3;
  }
}
largest = n2.value
n2 = n2.parent;
second = n2.compare.value;
while(n2.parent){
  n2 = n2.parent;
  if(n2.compare && n2.compare.value > second){
    second = n2.compare.value;
  }
}
```

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

4.

```
b = Array(n);
c = Array(k);
smallest = null;
sub = 0;
for(i = 0; i < k; i++){
  c[i] = 0;
}
for(i = 0; i < n; i++){
  smallest = a[c[0]*k];
  sub = 0;
  for(j = 1; j < k; j++){
    if(a[c[j]*k + j] < smallest){
      smallest = a[c[j]*k + j];
      sub = j;
    }
  }
  c[sub]++;
  b[i] = smallest;
}
```

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

5.

```
cur_max = {start: 0, end: 0, sum: 0};
max = {start: 0, end: 0, sum: 0}

for(i = 0; i < a.length; i++){
  if(cur_max.sum + a[i] < a[i]){
    cur_max.start = i;
    cur_max.end = i;
    cur_max.sum = a[i];
  } else{
    cur_max.sum += a[i];
    cur_max.end = i;
  }
  if(max.sum < cur_max.sum){
    max.sum = cur_max.sum;
    max.start = cur_max.start;
    max.end  = cur_max.end;
  }
}
```

at the end of the algorithm "max" contains the max sum of a span of indicies 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.