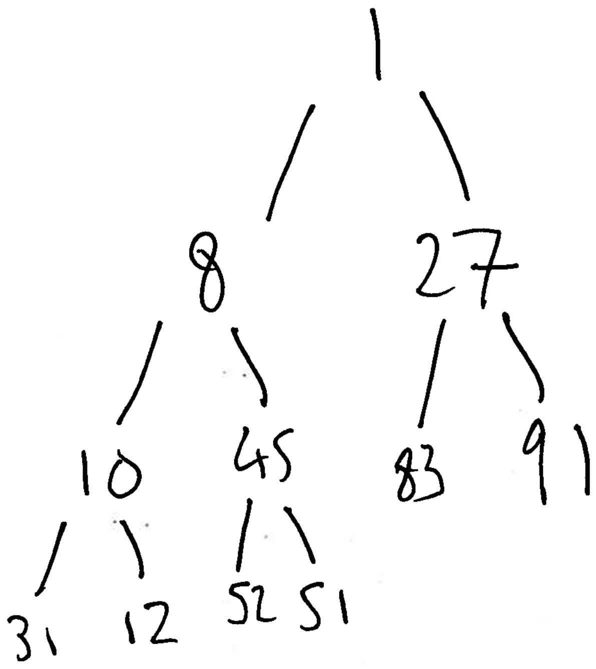
1. Each operation on S takes $O(\log n)$ time. There are $O(n)$ ~~of~~ 4 operations in total, so total time is:

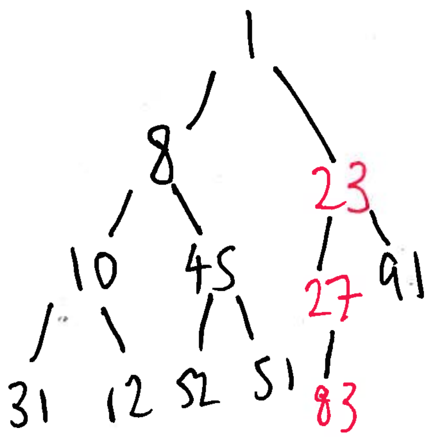
$$O(n \log n)$$

2. B is a heap.

a)



b) I've drawn the moved bits in red:



3. For a G:

Simply sort the array (using e.g. mergesort) and add the last K elements of the array.

For a VG:

```
h = new heap
for each x in array
    h.insert(x)
    if h.size > K then h.deleteMin()
sum = 0
while h not empty do
    sum = sum + h.findMin()
    h.deleteMin()
```

The idea is to loop through the array, and h contains the k greatest elements we've seen so far. Whenever h contains k+1 elements we remove the smallest one so it only contains the k greatest elements. Afterwards we sum up the whole heap.

4. For a G:

Use an AVL tree.

- new: new AVL tree
- inset: AVL insertion
- member: BST lookup

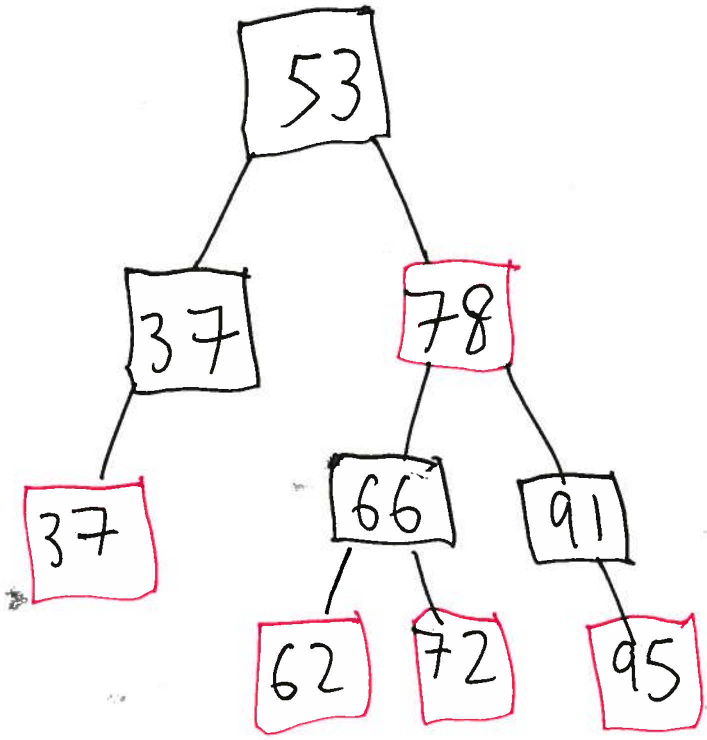
For increaseBy(x), add x to the value of each node in the tree (this doesn't change the relative order of any nodes so the BST + AVL invariants still hold afterward.

For a VG:

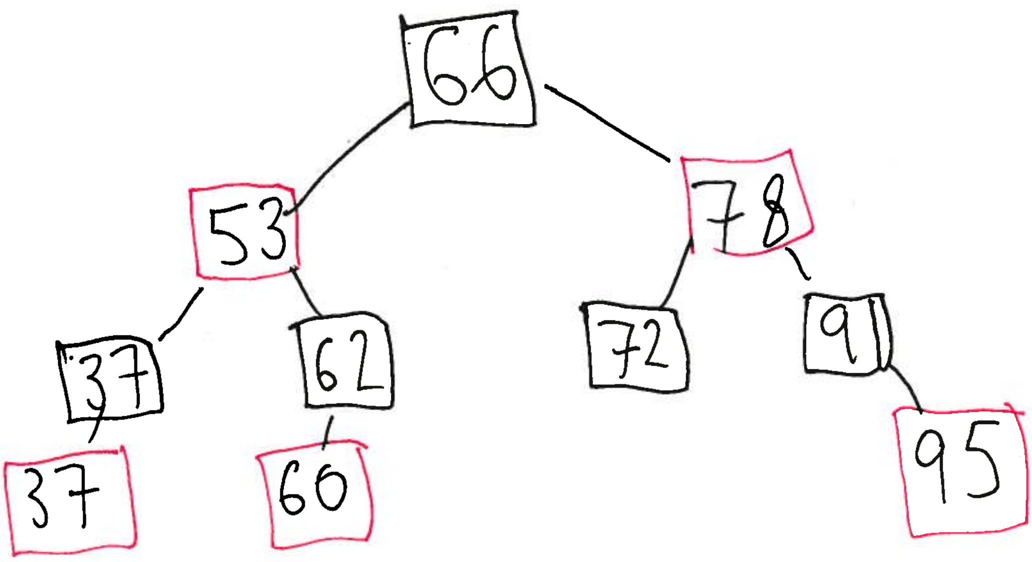
Use an AVL tree plus an integer variable "extra" which stores the total of all calls to increaseBy.

- new: new AVL tree, set extra to 0.
- increaseBy(x): extra = extra + x
- member(x): look up x - extra in tree
- insert(x): insert x - extra in tree

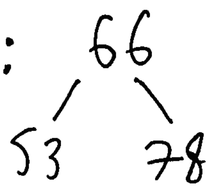
5a)



b) 60 goes as the left child of 62, there is a colour flip, then 53-78-66 is the right-left case ("inside grandchild"): so 66 ends up at the root:



The easiest way is to start with:



and fill in the rest from there.

6a)
```
greatest (Node x l Nil) = x
greatest (Node x l r) = greatest r
```

b)
```
delete x Nil = Nil
delete x (Node y l r)
  | x < y  = Node y (delete x l) r
  | x > y  = Node y l (delete x r)
delete x (Node y l Nil)
  | x == y = l
delete x (Node y l r)
  | x == y = Node g l (delete g r)
  where
    g = greatest r
```

The last case above corresponds to deleting a node with two children.