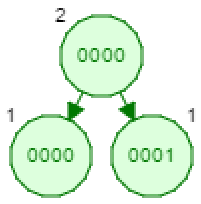
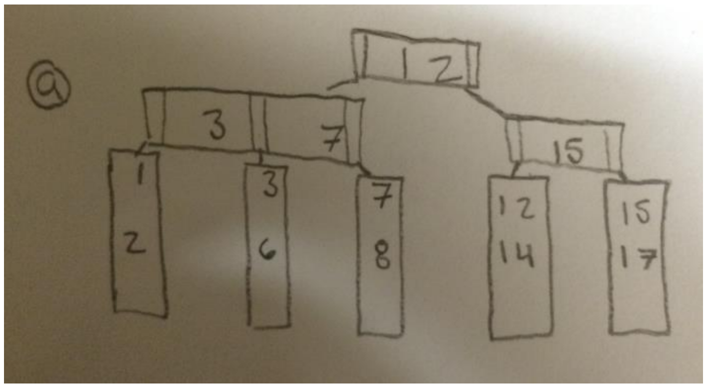
1) Our counter Example will include arbitrary functions f(n) = sin(n) and g(n) = cos(n). These two functions were chosen because they are not monotonic. These two functions have periods of increasing and decreasing. Therefore, at many points the two functions will switch between which is larger. However, by adding a constant, one of these functions can dominate over the other and the theory of Big – O is held.

   If we drop the constant, like the procedure we discussed in class, one of the two functions above will no longer be dominating throughout the entire domain. Big – O is no longer preserved. The constant in such non-monotonic functions is important and cannot be dropped.

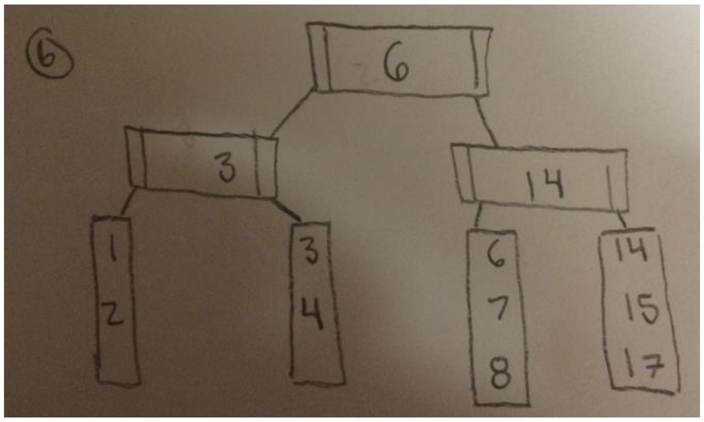
2) If we add value 0 into our tree, then a number greater than 0, and then again zero, then we will obtain the desired AVL tree. The right child of the first node will be the number greater than zero. The left node to this child will be zero. The AVL tree will need to rearrange to balance out and will produce the tree below.



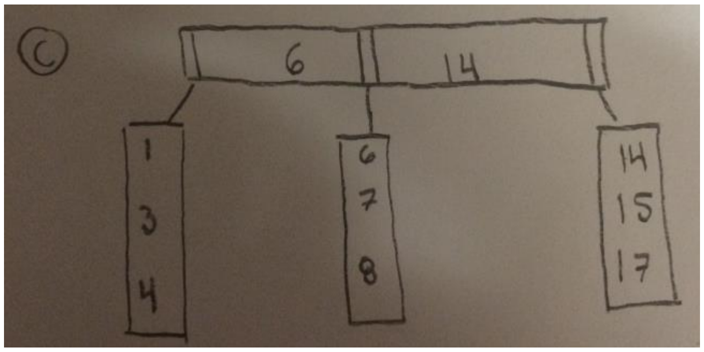
3)



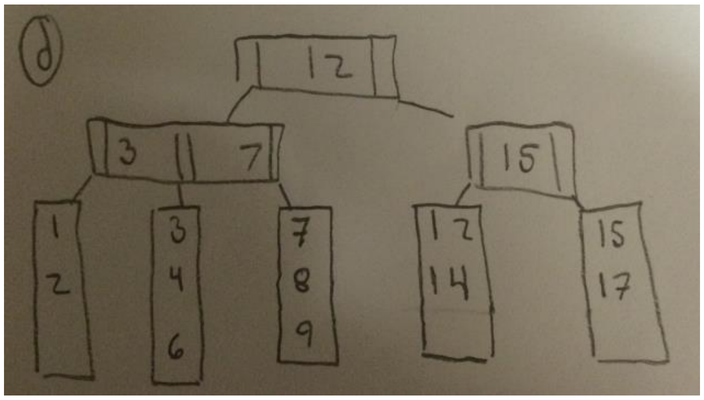
This is a simple problem of removing the number four and pulling from the next consecutive leaf so the under loaded leaf is satisfied. Re arrange the pointers.



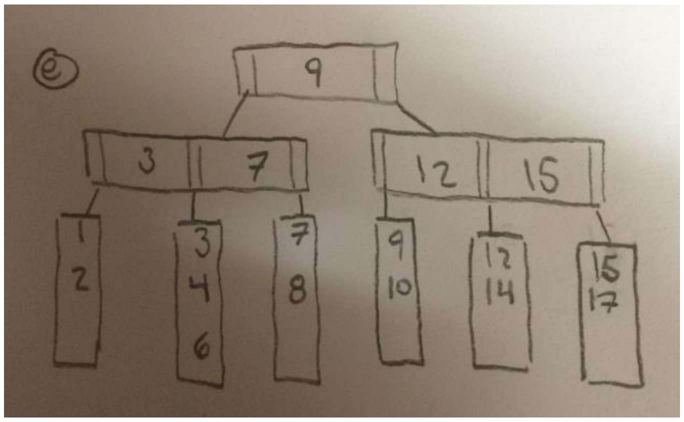
You must first remove the value 12 from the leaf. That leaf becomes under loaded, so you merge 14 into the next leaf. Now the tree is unbalanced so you must pull a leaf over and rearrange the pointers.



By removing 2, the leaf becomes under loaded. The only way to satisfy this is to merge with the next leaf and remove our top pointer which means we must merge our remaining pointers.

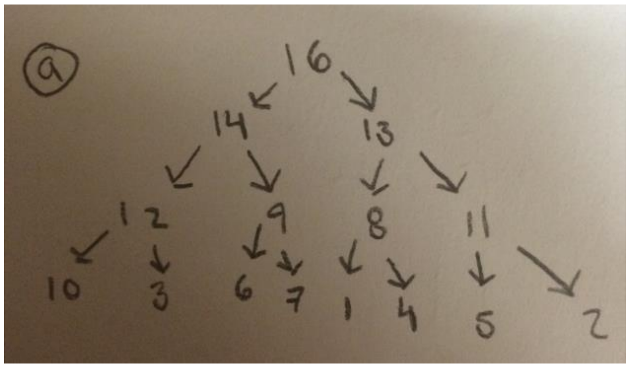


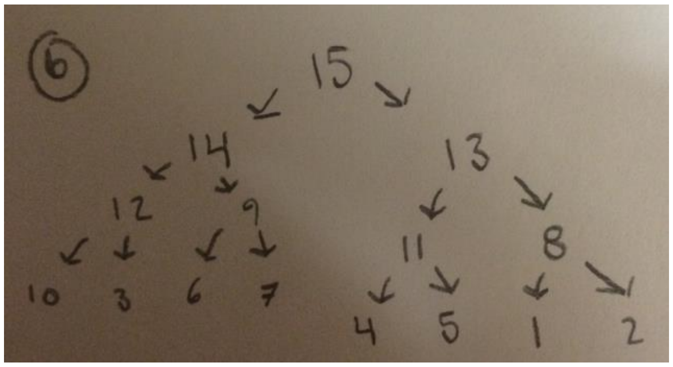
We must first add 9, but the leaf will then be over loaded. So we move the least value to the previous leaf. And then change our pointers.

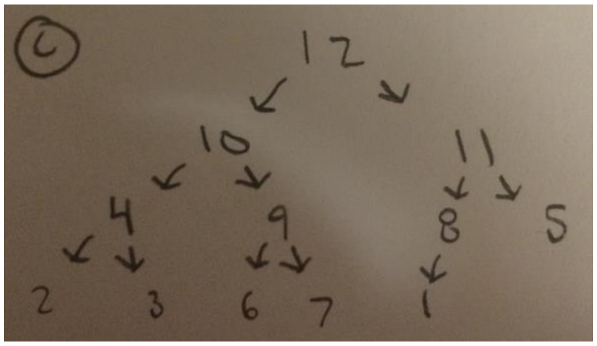


When adding ten, the leaf becomes over loaded, so we must move values. However its adjacent sibling is full. So then it decides to split the overflowing node to the other, not full branch. The pointers must be changed accordingly.

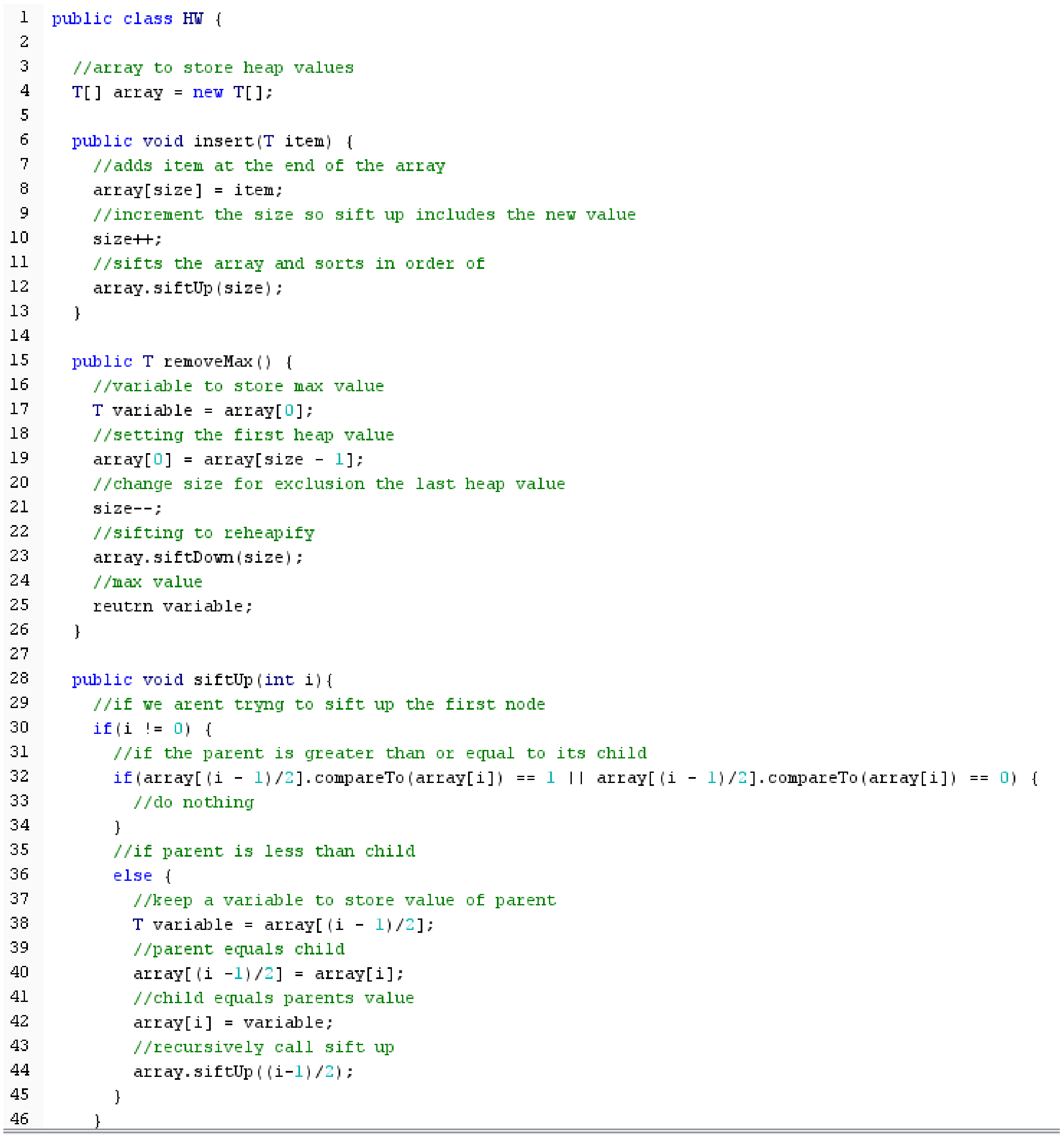
4)







5



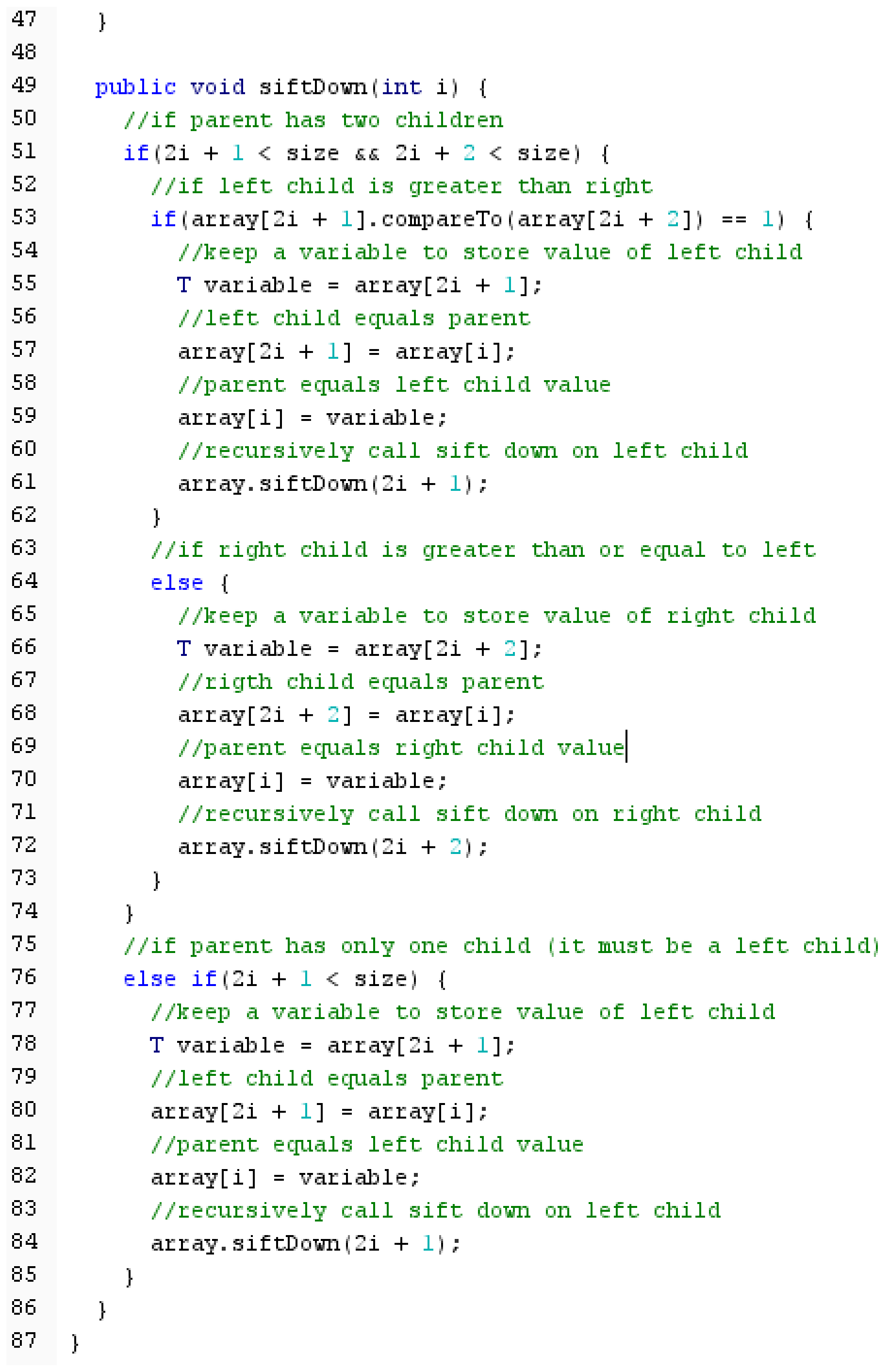
```java
public class HW {

  //array to store heap values
  T[] array = new T[];

  public void insert(T item) {
    //adds item at the end of the array
    array[size] = item;
    //increment the size so sift up includes the new value
    size++;
    //sifts the array and sorts in order of
    array.siftUp(size);
  }

  public T removeMax() {
    //variable to store max value
    T variable = array[0];
    //setting the first heap value
    array[0] = array[size - 1];
    //change size for exclusion the last heap value
    size--;
    //sifting to reheapify
    array.siftDown(size);
    //max value
    reutrn variable;
  }

  public void siftUp(int i){
    //if we arent tryng to sift up the first node
    if(i != 0) {
      //if the parent is greater than or equal to its child
      if(array[(i - 1)/2].compareTo(array[i]) == 1 || array[(i - 1)/2].compareTo(array[i]) == 0) {
        //do nothing
      }
      //if parent is less than child
      else {
        //keep a variable to store value of parent
        T variable = array[(i - 1)/2];
        //parent equals child
        array[(i -1)/2] = array[i];
        //child equals parents value
        array[i] = variable;
        //recursively call sift up
        array.siftUp((i-1)/2);
      }
    }
```



```
  }

  public void siftDown(int i) {
    //if parent has two children
    if(2i + 1 < size && 2i + 2 < size) {
      //if left child is greater than right
      if(array[2i + 1].compareTo(array[2i + 2]) == 1) {
        //keep a variable to store value of left child
        T variable = array[2i + 1];
        //left child equals parent
        array[2i + 1] = array[i];
        //parent equals left child value
        array[i] = variable;
        //recursively call sift down on left child
        array.siftDown(2i + 1);
      }
      //if right child is greater than or equal to left
      else {
        //keep a variable to store value of right child
        T variable = array[2i + 2];
        //rigth child equals parent
        array[2i + 2] = array[i];
        //parent equals right child value
        array[i] = variable;
        //recursively call sift down on right child
        array.siftDown(2i + 2);
      }
    }
    //if parent has only one child (it must be a left child)
    else if(2i + 1 < size) {
      //keep a variable to store value of left child
      T variable = array[2i + 1];
      //left child equals parent
      array[2i + 1] = array[i];
      //parent equals left child value
      array[i] = variable;
      //recursively call sift down on left child
      array.siftDown(2i + 1);
    }
  }
}
```