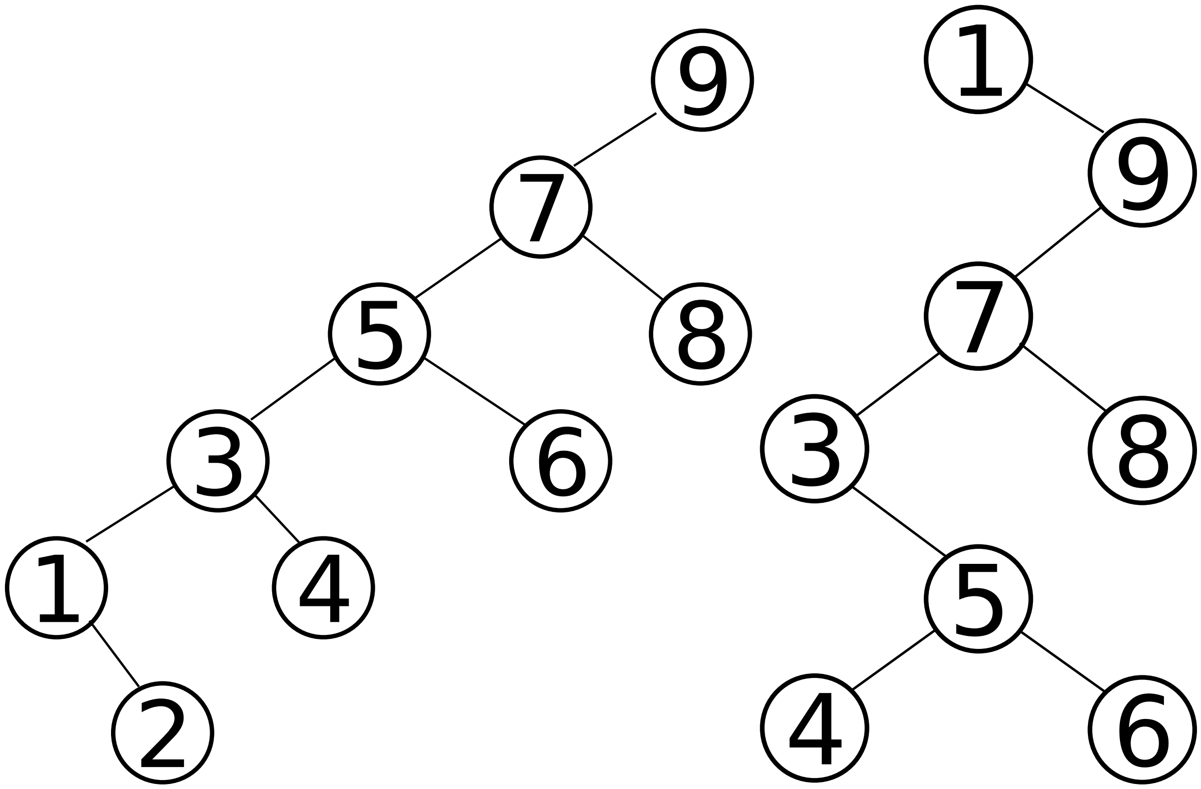
Homework 4

1. We should use a heap (priority queue) to represent our scheduler, sorting by the earliest timestamp. We do this by inserting (timestamp, event) and minimizing (Python tuples are sorted by the first item and then the second). This will ensure that if we extract the minimum item, we get the next item in the list.

2. 

3. We use a variant on selection sort by continuously finding the smallest remaining item.

```
def heap_sort(L):
    heap = Heap(L)
    result = []
    while heap:
        result.append(heap.extract_min())
    return result
```

4. We must look at $n/2$ items: each of the leaves of the heap. Each of the internal nodes is guaranteed to have a bigger child, but the leaves are not guaranteed to be in any sort of order.

5. There are two possible strategies. For numbers, we can insert $-x$ into the heap (which will reverse the order) and then convert back with $-x$ again upon extractions. Alternately, we can design a wrapper

class for the elements whose $<$ operator returns the opposite of their component elements (we wrap the elements when inserting and unwrap when extracting).

6. No, their method does not work. If we insert n items and then remove all but one of them, the next item to be inserted will have a key of 1, whereas the older item (which should be removed first) will have a key of $n-1$, making it be extracted after the new item.

7. We use our variable c to count the number of items that have ever been stored in our makeshift stack (we increment it after every insert). We insert an item with key c and maximize (or key $-c$ and minimize), ensuring that we always receive the most recent item.

8. Closed schemes, like chaining, can support large load factors (though their efficiency deteriorates). Open schemes, such as linear probing, cannot (even at reduced efficiency) because there is no place to insert the new item.

9.

Bucket	Contents
0	35
1	
2	
3	58
4	62, 4
5	88
6	26
7	19
8	12
9	
10	42, 53

10.

Bucket	Contents
0	35
1	53
2	
3	58
4	62
5	88
6	26
7	19
8	4
9	12
10	42

11. We can implement a set but mapping an item to itself (or `None`). The set operations then check for whether the given item is a key in the dictionary.

12. Our `OrderedDict` will use both a regular `dict` and a doubly-linked list. When we insert a key-value pair, we append a list node with the desired key into the linked list. We then insert `k:(n, v)` into the `dict`. When we remove a key from the dictionary we remove node n from the linked list. The addition of the linked list allows us to iterate over the dictionary in FIFO order. The addition of the node as a second value to the dictionary allows use to remove arbitrary keys in constant time.