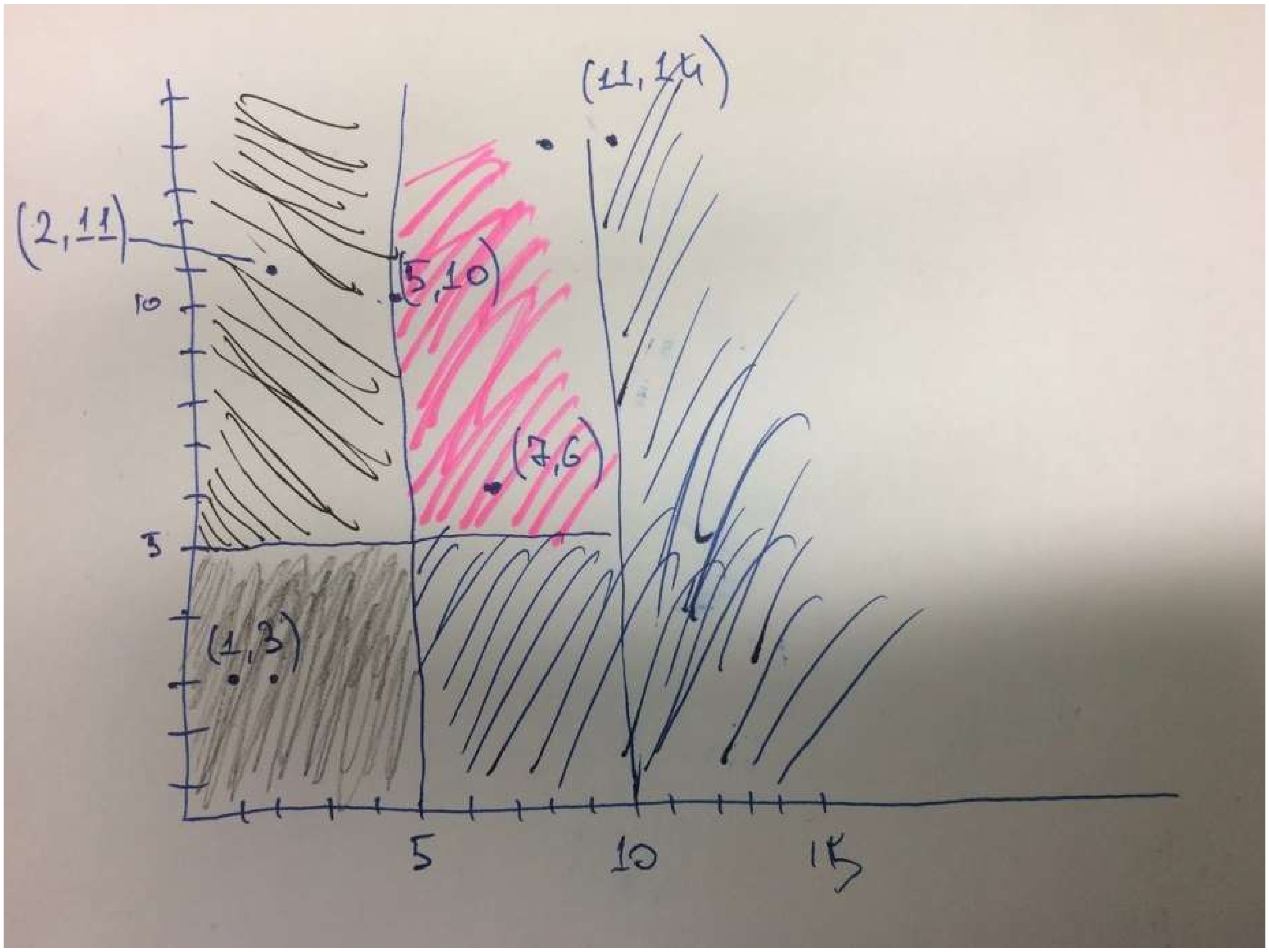
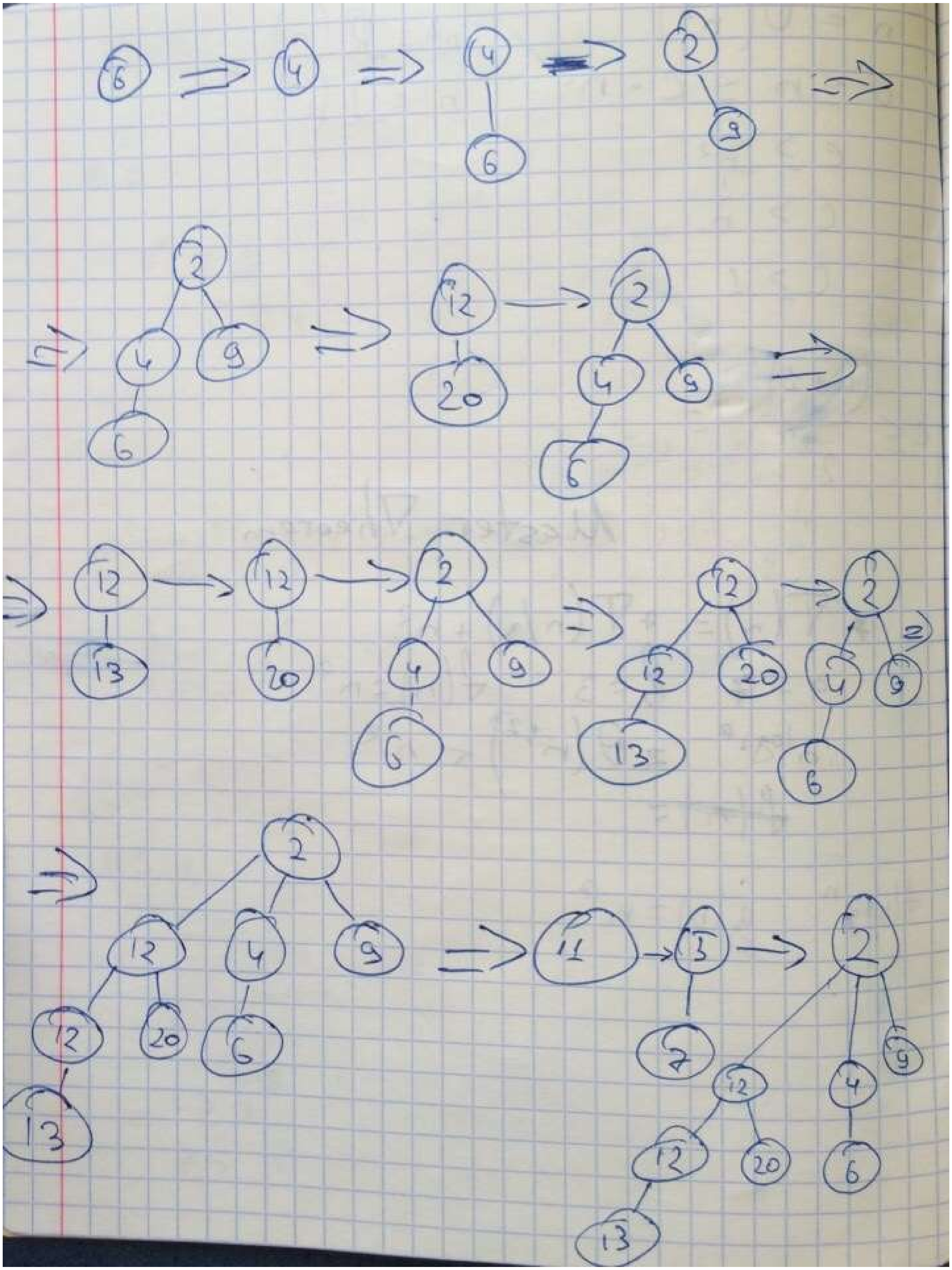
EX1: The main critique is that the node distribution to pages in memory is bad for binary heap as when doing operations on the heap many more pages have to be fetched to main memory than is necessary for the operation that is attempted. That is because if moving vertically through the heap almost every node is on another page, because almost all levels are on different pages. B-heap is a heap implemented to keep subtrees in fixed-length blocks of virtual memory, which leads to "reduced the VM footprint and, consequently, VM page faults". First seven nodes in b-heap are indexed as in a binary heap, then other nodes are grouped in eight, where first level contains i and i + 1, the second — i + 2 and i + 3, the third — i + 4, i + 5, i + 6, i + 7 nodes. Then a new group of eight elements begins.

EX2: Two intervals A = (a1, b1) and B = (a2, b2) overlap if and only if a2 < b1 and a1 < b2



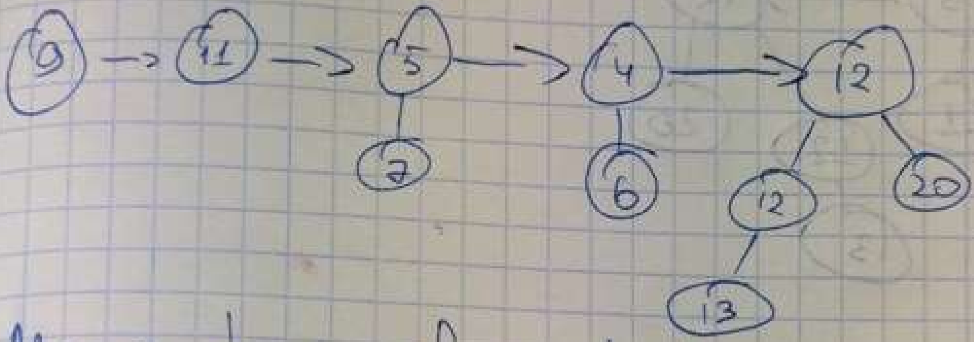
Here are 6 types of relationships: 1.non-overlapping from the left to the reference interval: (1,3). 2.non-overlapping from the right to the reference interval: (11,14). 3.overlapping from the right to the reference interval: (7,6). 4.overlapping from the left to the reference interval: (2,11). 5.in the reference interval. 6.include the reference interval.

Ex3: Inserting values 6, 4, 2, 9, 12, 20, 12, 13, 7, 5, 11 into a min-priority Binomial heap.

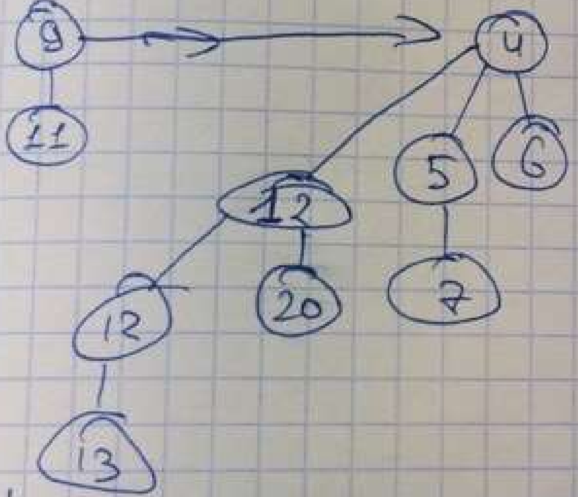


Delete 2

Promoting children

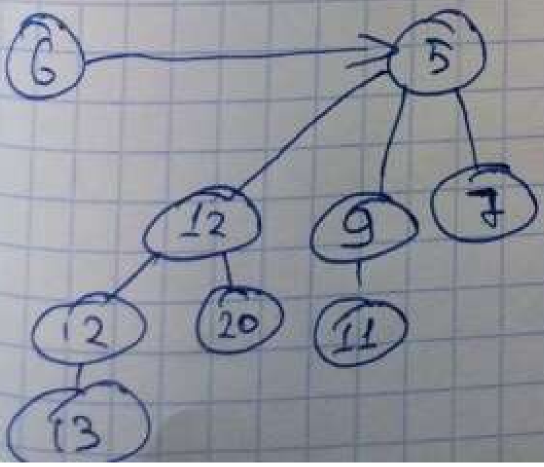


Merge trees of equal degree



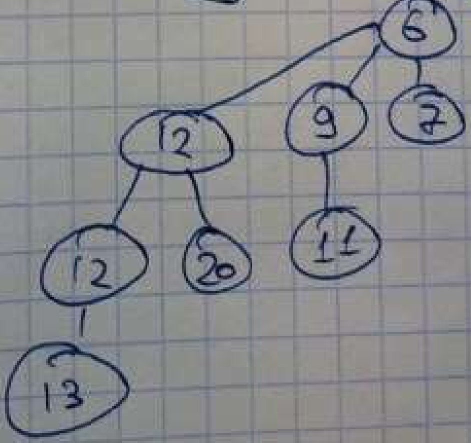
Delete 4

Promoting and merging

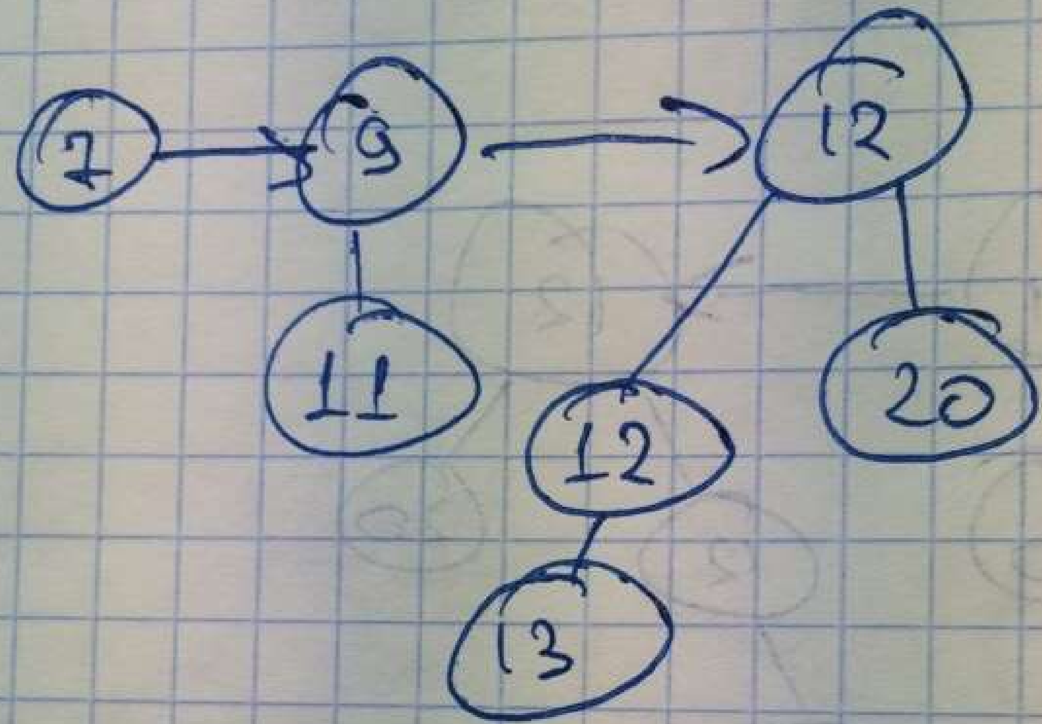


Delete 5

Promoting and merging



Delete 6



EX4:

EX5: The code below was taken by me from wikipedia. It is an implementation of a k-d tree. It is a space-partitioning data structure for organizing points in a k-dimensional space. Given a list of n points, the following algorithm uses a median-finding sort to construct a balanced k-d tree containing those points.

In [71]:
```
from collections import namedtuple
from operator import itemgetter
from pprint import pformat

class Node(namedtuple('Node', 'location left_child right_child')):
    def __repr__(self):
        return pformat(tuple(self))

def kdtree(point_list, depth=0):
    try:
        k = len(point_list[0]) # assumes all points have the same dimension
    except IndexError as e: # if not point_list:
        return None
    # Select axis based on depth so that axis cycles through all valid values
    axis = depth % k

    # Sort point list and choose median as pivot element
    point_list.sort(key=itemgetter(axis))
    median = len(point_list) // 2 # choose median

    # Create node and construct subtrees
    return Node(
        location=point_list[median],
        left_child=kdtree(point_list[:median], depth + 1),
        right_child=kdtree(point_list[median + 1:], depth + 1)
    )

def main():
    """Example usage"""
    point_list = [(69.09, 54.00, 25.41, 53.81), \
                  (20.94, 1.42, 81.92, 22.04),
                  (37.50, 17.07, 67.45, 47.57),
                  (81.43, 12.07, 34.50, 43.36),
                  (45.47, 48.67, 20.09, 71.35),
                  (61.15, 37.69, 66.32, 82.65),
                  (10.36, 40.16, 73.01, 28.76),
                  (81.32, 48.76, 92.31, 66.00),
                  (25.58, 97.98, 33.35, 86.06),
                  (41.63, 95.85, 29.15, 97.37),
                  (12.32, 1.54, 32.08, 38.43),
                  (15.24, 79.05, 83.79, 48.47),
                  (65.94, 44.86, 35.05, 65.39),
                  (66.48, 75.14, 39.06, 63.50),
                  (90.28, 34.17, 97.13, 92.62),
                  (60.91, 38.70, 26.60, 81.90),
                  (22.75, 26.87, 51.45, 89.81),
                  (72.22, 23.59, 7.51, 53.80),
                  (77.12, 73.64, 94.43, 44.94)]
    tree = kdtree(point_list)
    print(tree)

if __name__ == '__main__':
    main()
```

```
((60.91, 38.7, 26.6, 81.9),
 ((10.36, 40.16, 73.01, 28.76),
 ((37.5, 17.07, 67.45, 47.57),
 ((22.75, 26.87, 51.45, 89.81), ((12.32, 1.54, 32.08, 38.43), None, None), No
ne),
 ((20.94, 1.42, 81.92, 22.04), None, None)),
 ((25.58, 97.98, 33.35, 86.06),
 ((41.63, 95.85, 29.15, 97.37), ((45.47, 48.67, 20.09, 71.35), None, None), N
one),
 ((15.24, 79.05, 83.79, 48.47), None, None))),
 ((65.94, 44.86, 35.05, 65.39),
 ((61.15, 37.69, 66.32, 82.65),
 ((72.22, 23.59, 7.51, 53.8), ((81.43, 12.07, 34.5, 43.36), None, None), Non
e),
 ((90.28, 34.17, 97.13, 92.62), None, None)),
 ((81.32, 48.76, 92.31, 66.0),
 ((66.48, 75.14, 39.06, 63.5), ((69.09, 54.0, 25.41, 53.81), None, None), Non
e),
 ((77.12, 73.64, 94.43, 44.94), None, None))))
```

The output is a random hyperplane direction.