Supervision work
Supervision 13

Tudor Avram
Homerton College, tma33@cam.ac.uk

16 Feb 2016

1 B-Trees

1.1 Exercise 1

(i) The B-Trees are mainly an optimisation of BST trees memory-wise, when they are stored on disk memory. If they are stored in main memory, though, this optimisation is no longer valid. For a minimum degree t, the cost of searching a N-node b-tree is $O(\log_t N)$, while the cost required by a red-black tree would be $O(\log_2 N)$. Although at first sight the b-tree seems to be more efficient, for a large enough N, the logarithm base does not count, so there is no complexity optimisation, either. Therefore, we can deduce that if the tree is stored purely in memory, a b-tree is not a better choice than a red-black tree.

(ii) When the tree is stored in memory, we have the liberty to choose what part of the tree to read from the disk. This way, we only use a part of the main memory at a time. In this case is where the b-trees become really useful. While by applying this strategy for red-black trees would mean reading 1 node at a time (which is very costly), a b-tree node encodes multiple (key, value) pairs. This way, we are required less readings than we would be by using a red-black tree. In conclusion, if the tree is stored on disk, it would be more efficient to use a b-tree rather than a red-black tree.

1.2 Exercise 2

(a) *MINIMUM :* every internal node has 700 children(i.e. 699 keys), except the root which has only 2 (i.e. 1 key).
$min = 1 + 2 \times 699 + 2 \times 699 \times 700 + ... + 2 \times 699 \times 700^{h-2} = 1 + 2 \times 699 \times (1 + 700 + 700^{h-1}) = 1 + 2 \times 699 \times \frac{700^h - 1}{699} = 1 + 2 \times (700^h - 1) = \underline{2 \times 700^h - 1}$ keys.

(b) *MAXIMUM :* every internal node has 1400 children (i.e. 1399 keys). $max = 1399 + 1399 \times 1400 + ... + 1399 \times 1400^h = 1399 \times (1 + 1400 + ... + 1400^h) = 1399 \times \frac{1400^{h+1}-1}{1399} = \underline{1400^{h+1} - 1}$ keys.

2 Hash tables

2.1 Exercise 3

When using chaining to handle collisions in a hash table, we can end up to an average of $\alpha = \frac{n}{N}$ elements on every list, where $n =$ the number of elements inserted in the hash table and $1...N =$ the range of the possible results of the hashing function. The cost needed to access a certain row in the hash table is $O(1)$. Therefore, we can say that the cost of cost of accessing a certain value in a hash table is the cost of getting to the right row $(O(1))$ + the cost of searching through the list (which is an average of $O(\alpha)) = O(1 + \alpha)$.

2.2 Exercise 4

(i) When we use chaining, we just need to delete the entry from its correspondent list.

(ii) When we use open addressing for solving collisions, we do not know for sure on which row our entry is in, so we need to use probing to find its position again. The issue with deleting would be that, if we actually erase the entry from the hash table and we mark a certain element with *null*, our search algorithm may return the fact that a number is not in the hash, although this is not valid. Therefore, we give the entry a special value, *DELETED*, such as when we search the hash, we know that we can proceed to other rows, as the desired value might still be in the table.

3 Priority Queues

3.1 Exercise 6

(a) first() – for both the single and double linked lists, returning the first element requires $\underline{O(1)}$.
(b) extractMin() – for both the single and double linked lists, deleting the first element requires $\underline{O(1)}$, as we just need to replace the first node with the next one.
(c) insert() – In both cases, in order to insert a new entry to the priority queue, we need to search the entire list in order to find its position (i.e. $\underline{O(n)}$ complexity)
(d) delete() – If we want to delete a certain entry, we first need to find it in the priority queue, using the same algorithm for the **insert()** function. Once found, we can delete it simply by swapping references, which has a cost of O(1).

Therefore, the overall complexity of deleting an entry for both the double and singly linked list is $O(n)$.
(e) decreaseKey() – First we need to find the desired entry, algorithm that has a complexity of $O(n)$. In this case, we can use different strategies to get the entry to its new position in the priority queue :

1. *singly-linked list* : Once found, we delete the entry and insert the new entry with its new key on the right position in the list. This operation has a cost of $O(n)$, so the total number of operations needed would be $2 \times n$ or $O(n)$.
2. *double-linked list* : In this case, we have a pointer to both nodes, so we don't need to delete the entry and re-insert it. We can just shift it to the left until we reach the new position. The worst-case complexity of the shifting would be $O(n)$, so the total number of operations would be $2 \times n$ or $O(n)$.

Table 1: Priority queue complexities

Function	**singly-linked list**	**double-linked list**
first()	$O(1)$	$O(1)$
extractMin()	$O(1)$	$O(1)$
insert()	$O(n)$	$O(n)$
delete()	$O(n)$	$O(n)$
decreaseKey()	$O(n)$	$O(n)$

3.2 Exercise 8

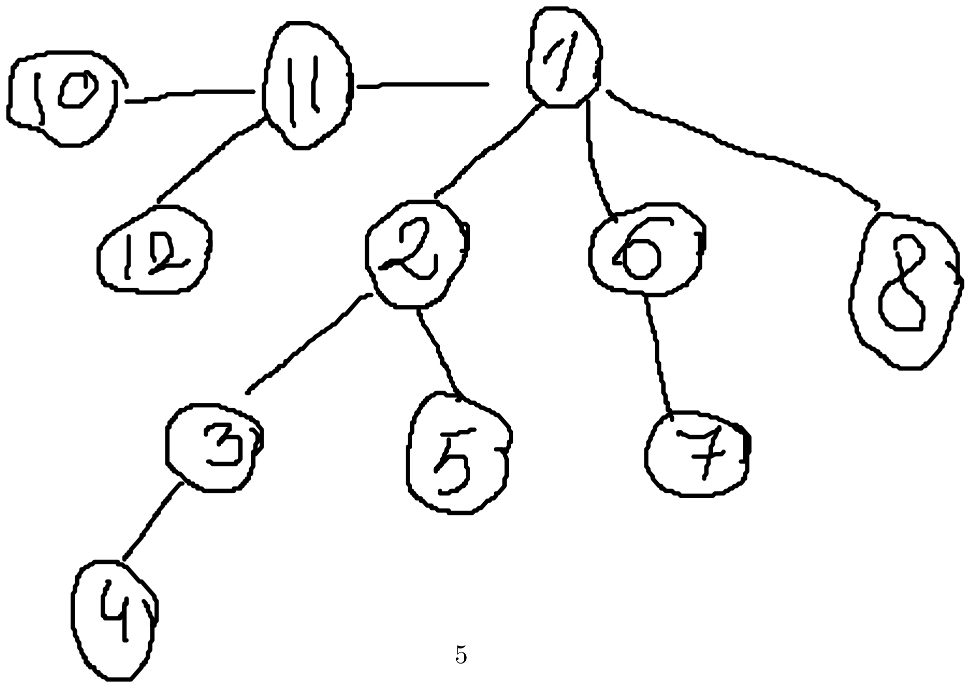
```
package uk.ac.cam.tma33.s13;

public class BinomialHeapNode<K extends Comparable<K>,E>
        implements Comparable<BinomialHeapNode>{

private BinomialTree<K,E> mTree;
private BinomialHeapNode<K,E> mNext;

public BinomialHeapNode() {
        mTree = null;
        mNext = null;
}

public BinomialHeapNode(BinomialTree<K,E> input) {
        mTree = input;
        mNext = null;
}

// Getters

public BinomialHeapNode<K,E> getNext() {
        return mNext;
}

public BinomialTree<K,E> getTree() {
        return mTree;
}

public E getMin() {
        return mTree.getMin();
}

public K getMinKey() {
        return mTree.getMinKey();
}

public int getOrder() {
        return mTree.getOrder();
}

//Setters

public void setNext(BinomialHeapNode<K,E> input) {
        mNext = input;
}

public void setTree(BinomialTree<K,E> input) {
        mTree = input;
}

// Merging 2 trees

public void merge(BinomialHeapNode<K,E> input) {
        int cmp = this.getMinKey().compareTo(input.getMinKey());
        BinomialTree<K,E> newTree = new BinomialTree<K,E>();
        if (cmp > 0) {
                newTree = input.mTree;
                newTree.addChild(this.mTree);
        }
        else {
```

```
                    newTree = this.mTree;
                    newTree.addChild(input.mTree);
            }
            mTree = newTree;
}

@Override
public int compareTo(BinomialHeapNode input) {
        return this.getOrder() - input.getOrder();
}

}
```

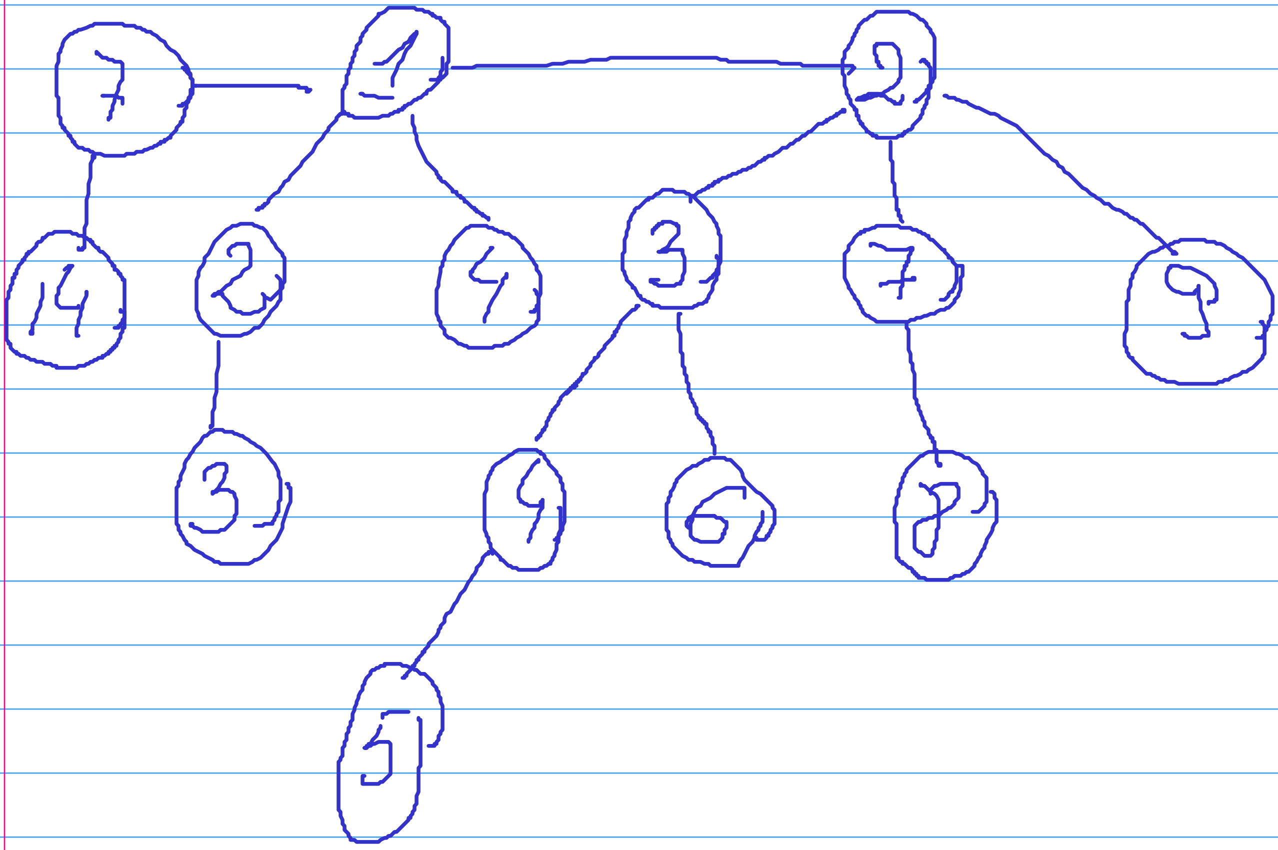
3.3 Exercise 7

(i) Binomial Heaps : I keep the binomial heap as being a list of *BinomialHeapNode*, kept sorted by the order of the binomial trees.

(a) Binomial Trees 0, 1, 3

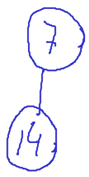


(b) Binomial Trees 1,2,3

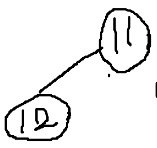
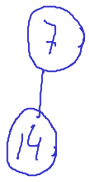


(ii) Merging : I use the classic merging algorithm of two sorted arrays/lists, slightly modified in order to be able to merge the binomial trees, when needed.

I compare 2 nodes at a time, and decide which one goes into the new binomial heap, which I keep on heap (a).

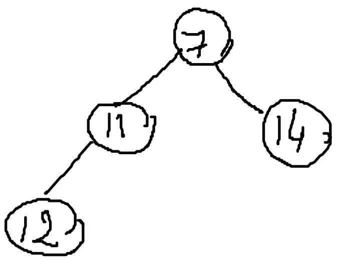
STEP 1 : Nodes and 

The first node has a smaller order than the second one, so there's no need for merging them. We just keep 10 in the heap (a).

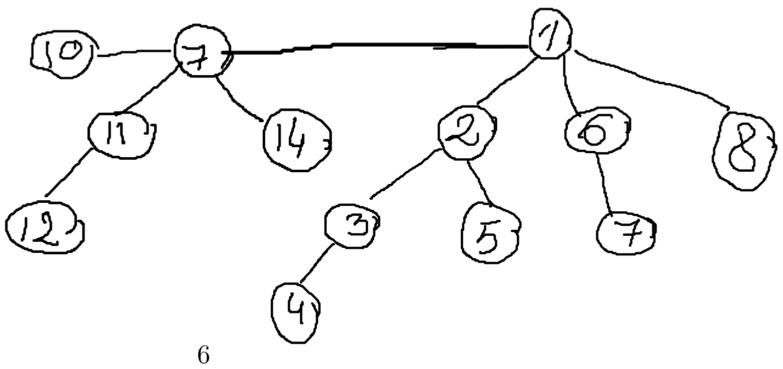
STEP2 : Nodes  and 

Both nodes have the same oder, so this time we can merge them.

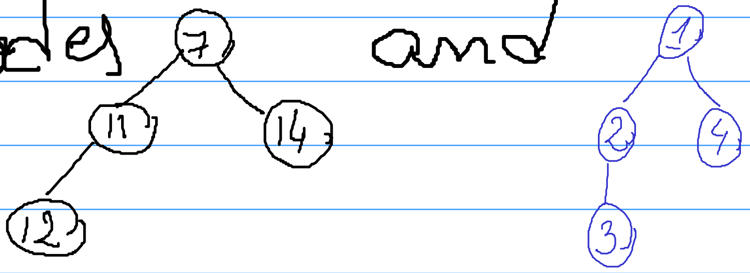
$\Downarrow$



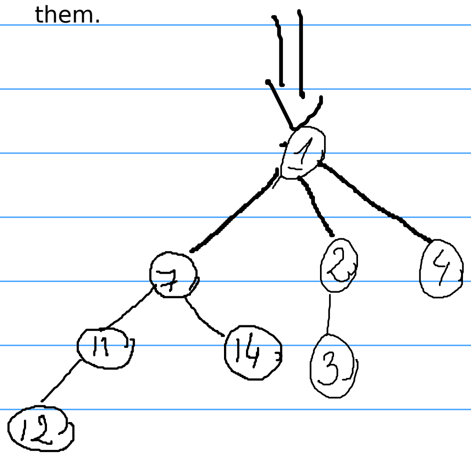
Therefore, up until this step, our merged binomial heap looks like this :



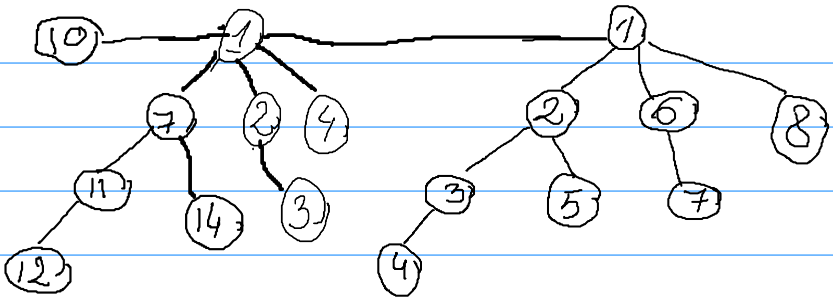
STEP 3 : Nodes



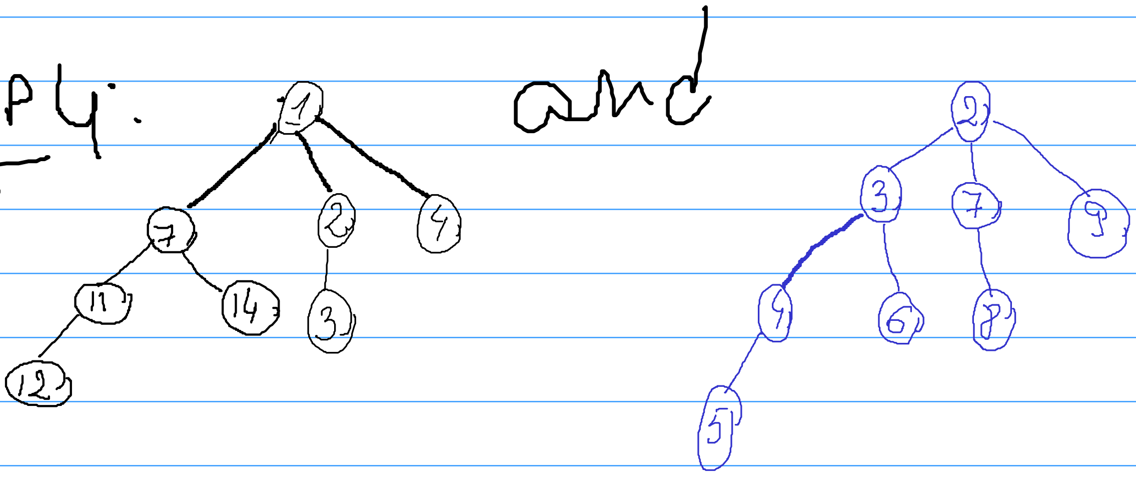
Both nodes have the same oder, so this time we can merge them.



Therefore, up until this step, our merged binomial heap looks like this :

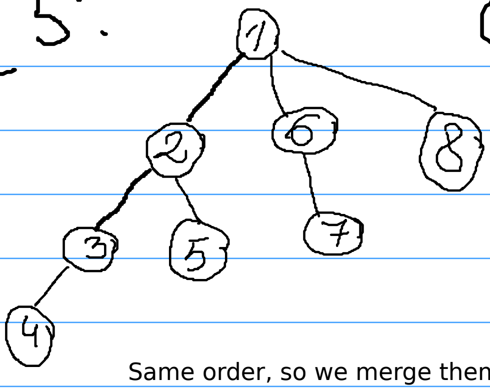


STEP 4 :

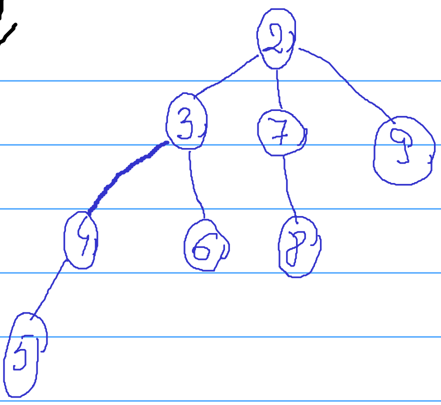


We observe that the nodes have the same order again, but we will not merge them togheter at this stage, because the next tree in heap (a) also has an order of 3, so if we would merge them at this stage, we woud violate the property of keeping the trees sorted by their order. So we move on to the next step.

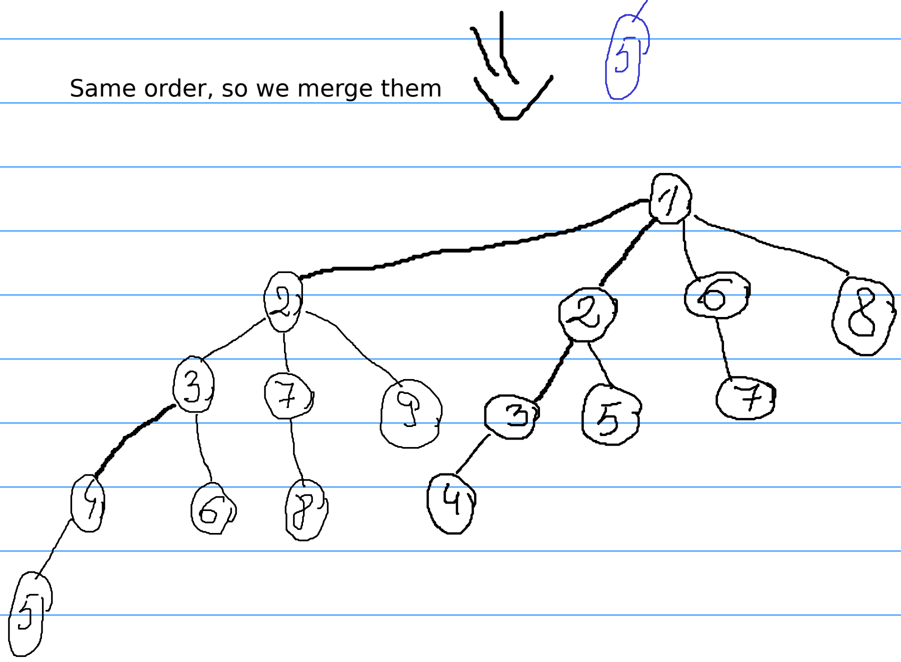
STEP 5:



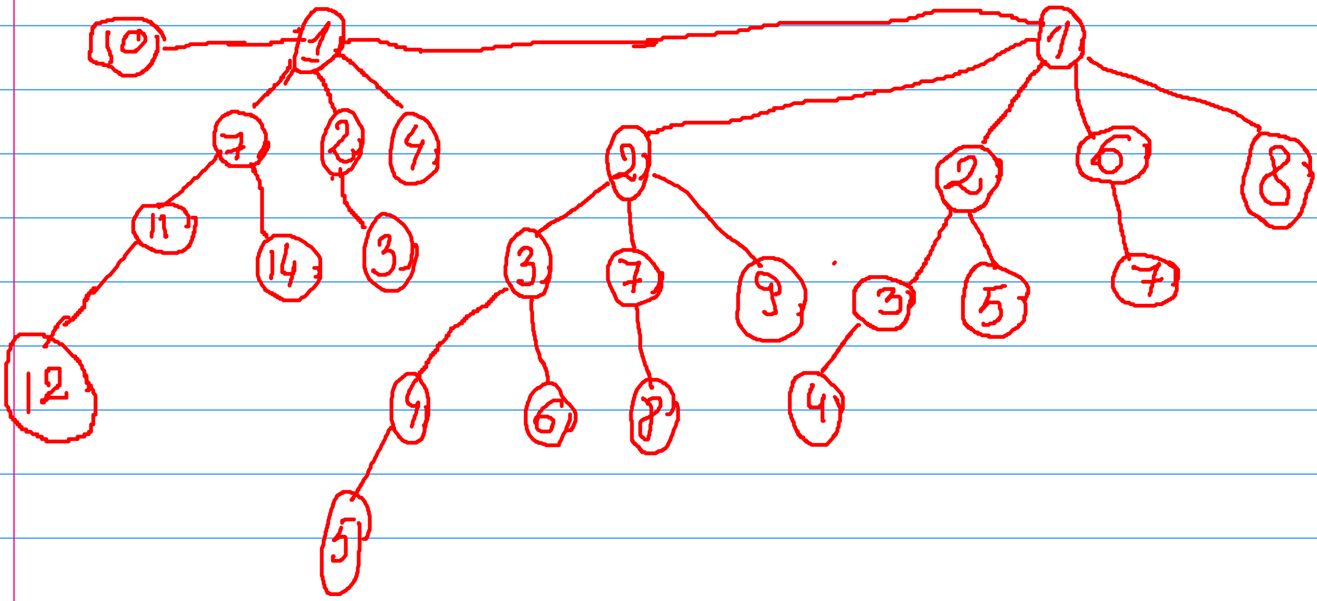
and



Same order, so we merge them

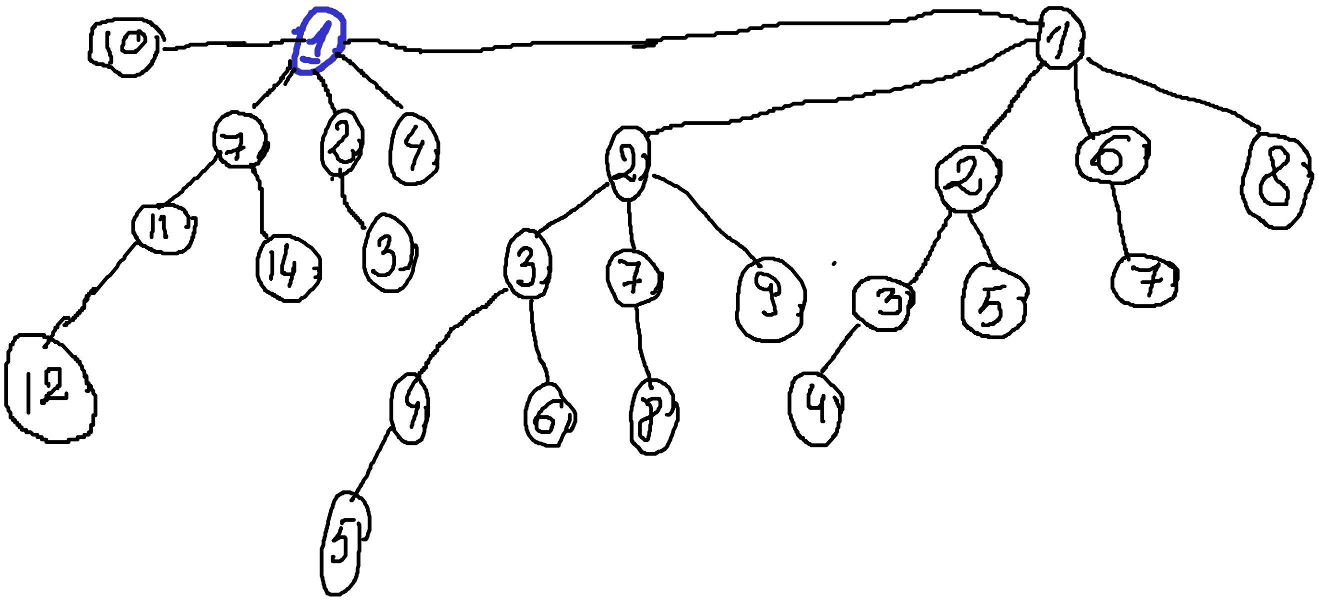


At this stage, we finished the merging, and the resulting binomial heap will look like this :

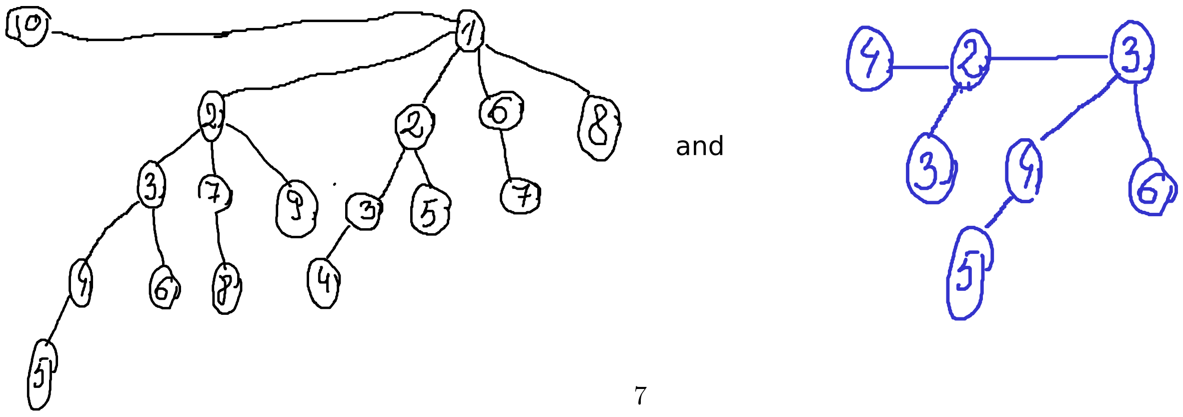


(ii) Extracting the minimum : First, I need to find the minimum entry, from the sub-trees. After I do this, I have to delete it, so I extract the node and create a new binomial heap using its children. From this stage, I just proceed with a standard merging operation between the two binomial heaps.

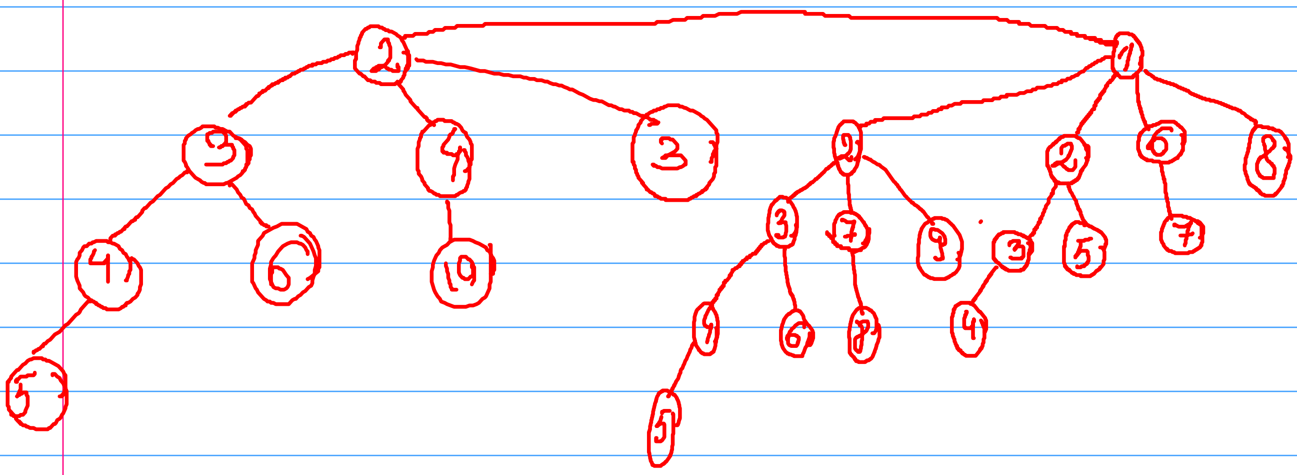
We found the minimum. No we have to extract it from the heap.



The resulting binomial heaps are :



So, after re-merging the 2 heaps, the final binomial heap will look like this :


2
3
4
3
4
6
10
5
1
2
2
6
8
3
7
9
3
5
7
4
6
8
4
5