1 Algorithms and Data Structures 2 - Exercises 1

1.1 Problem 1

1. **Give the important differences between a normal priority queue and an addressable priority queue.**

   Normal priority queues maintain a set M of elements offering the following operations:
   $build(\{e_1, ..., e_n\}) : M := \{e_1, ..., e_n\}$
   $insert(e) : M := M \cup \{e\}$
   $min : return\ min\ M$
   $deleteMin : e := min\ M; M := M \backslash \{e\}; return\ e$

   Addressable priority queues additionally support operations on arbitrary elements addressed by an element handle h:
   $insert$: As before but return a handle to the element inserted.
   $remove(h)$: Remove the element specified by handle h.
   $decreaseKey(h : Handle, k : Key)$: Decrease the key of the element specified by handle h to key k.
   $merge(Q_2) : Q_1 := Q_1 \cup Q_2; Q_2 := \emptyset$.

2. **Compare the running time of a merge operation for pairing heaps and binary heaps.**

   The *Pairing Heap* can merge another *Pairing Heap* in constant time $O(1)$ as only the minPtr is eventually updated and the addressable priority queue to be merged is simply attached to the forest.

   The *Binary Heap* has a complexity of $O(n + m)$ as it throws the n elements of the first and the m elements of the second *Binary Heap* together and then calls $build((\{e_1, ..., e_{n+m}\}))$ again.

1.2 Problem 2

1. **Prove the general lower bound for addressable priority queues of Ω(log n) for deleteMin under the condition that insert runs in constant time.**

2. **Why does this bound not have to hold if insert is allowed to consume more time?**

1.3 Problem 3 (4 points)

For a very big festival, you are responsible for the bar. For this task, you designed a bar robot (aka Bender) that mixes excellent cocktails. The ingredients are stored in big boxes. But exactly here is the problem: you have to make sure that none of the boxes runs empty. On the other hand, you also want to enjoy the evening and not constantly check all of the boxes. To deal

with this problem, there is only one solution: a refill display! Unfortunately, the only displays available are the ones that can display one line only. It is clear that this line should display the most urgently need ingredient. The goal is to design an algorithm that keeps the display up to date.

1. **As a base for you algorithm, think about a data structure to efficiently implement your algorithm. Assume that the number of ingredients for a specific cocktail is significantly smaller than the number of ingredients totally available.**

   To model the refill display, that can only show the most urgent ingredient, an addressable priority queue will be used. An addressable one is needed, as the ingredients decrease in each box throughout the evening, which can be modeled only with the *decreaseKey(h: Handlke, k: Key)* function of addressable priority queues. The priority is represented by how much each box is filled. The one that is filled the least will be shown on the display, it is the *minimum* element in the priority queue. With *deleteMin()* an empty box can be removed and a new box can be added by *insert(e)*. The *remove(h: Handle)* function is not needed as one can anyways only look at one box at most by the restrictions of the display (the one that is filled the least by using *min()*). As the number of ingredients for a specific cocktail is significantly smaller than the number of ingredients totally available, a lot of cache misses might occur in the application. To counter this a *Binary Heap* implementation will be chosen that reduces the amount of cache misses in practice as it uses addressable arrays/vectors internally instead of pointers like a *Pairing Heap* or *Fibonacci Heap* implementation. Thus this leads to better performance in practice even if the *decreaseKey(h: Handlke, k: Key)*, *deleteMin()* and *insert(e)* function only run in *O(log n)* in theory. On implementation level, a vector-like data structure will be used that is organized by an implicit tree representation maintaining the heap property/invariant.

2. **Design a function MixDrink(recipe), that operates on your data structure. Give Pseudocode. Your data structure has to be updated! Other functions of the robot don't have to be "programmed".**

```
  public Cocktail MixDrink( recipe ) {
    Cocktail cocktail = new Cocktail();
    for ingredient in recipe {
      unsigned int amount = ingredient.amount;
      IngredientBox box = this.fetchIngredientBox(ingredient.name)
;
      if(box.fill >= amount) {
        cocktail.add(box.remove(amount));
        # decreaseKey calls siftUp() internally
        displayPQ.decreaseKey(ingredient.name, box.fill);
      } else {
        amount -= box.fill
        cocktail.add(box.remove(box.fill));
        displayPQ.decreaseKey(ingredient.name, 0);
```

```

          this.throwToTrash(box);
          box = this.getNewIngredientBox(ingredient.name);
          # deleteMin calls siftDown() internally
          displayPQ.deleteMin();
          # insert calls siftUp() internally
          displayPQ.insert(ingredient.name, box.fill);

          cocktail.add(box.remove(amount));
          displayPQ.decreaseKey(ingredient.name, box.fill);
        }
      }
      return cocktail;
    }

```

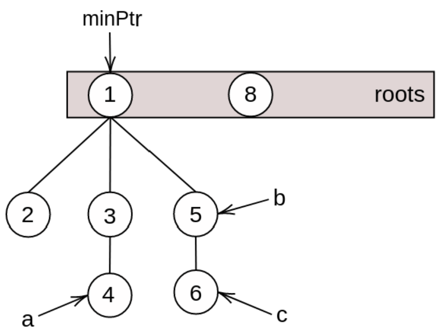
Listing 1: The function MixDrink(recipe)

3. **When a box is exchanged, your data structure has to be updated as well. Describe what consequences the exchange has on your data structure.**

   There are actually two possibilities. Either one calls *deleteMin()* and *insert(e)* right afterwards which leads to recursive *siftDown()* and *siftUp()* operations or one supports an *increaseKey(h: Handle, k: Key)* function, that leads to only one recursive *siftDown()* call. Nevertheless, *O(log n)* swaps need to be performed.

1.4 Problem 4

The definition of a pairing heap does not state which elements are neighbors. For the solution of the next tasks we assume a sorted list of roots (by insertion order) and neighborhood induced by this. Given is a Pairing Heap in the following state.

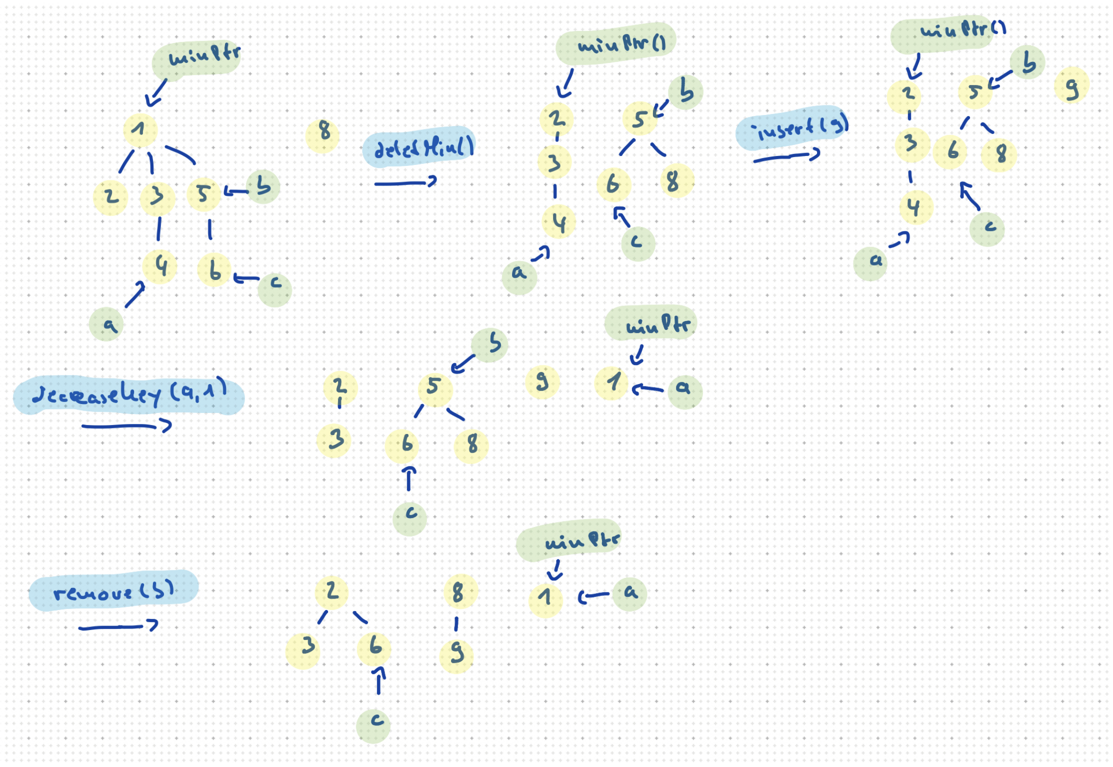


1. **Give a very short sequence of operations that creates this state.**

   insert(0)
   insert(1)
   insert(2)

insert(3)
insert(4)
insert(5)
insert(6)
deleteMin()
insert(0)
deleteMin()
insert(0)
deleteMin()
insert(8)

2. **Execute the following operations step by step on the given heap and draw the intermediate state of the heap after each operation: deleteMin(), insert(9), decreaseKey(a,1), remove(b).**


minPtr
deleteMin()
minPtr()
insert(9)
minPtr()
decreaseKey(a,1)
minPtr
remove(b)
minPtr