Artemis 5.4.1

Start exercise

Not yet started.

? ? ? ? ? ? ?

A Heap of Nodes

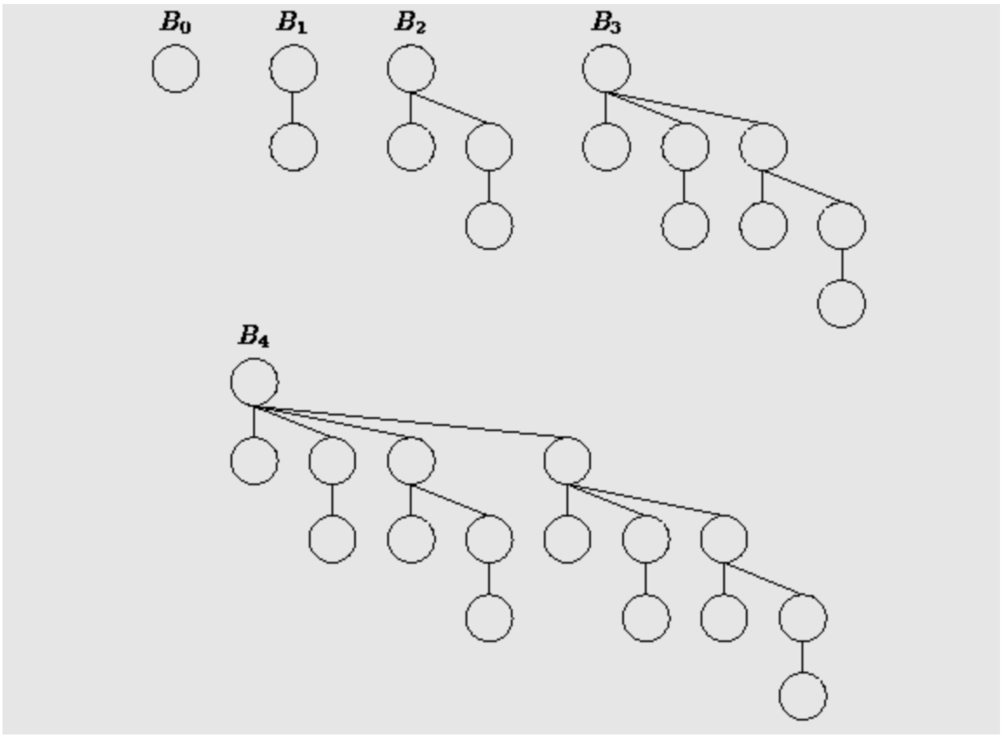
The PUM (Pingu University Munich) gets back to you: "Hello! Hello ? `<Insert Your Name Here>`!! We need your help! Our administration is going down the drain. We have too many things in the pipeline that we have to do and don't know which we should work on first. " As always, you want to help them. You already have an idea. A priority queue should solve the penguins' problems. As a fan of binary numbers, you of course opt for a binomial heap.

Introduction to binomial heaps and invariants that must apply in your implementation

Binomial heaps are priority queues that use several binomial trees to create a heap structure. Binomial trees are structured as follows:
The binomial tree of order$0 \leq k$ with root R is the tree B_k defined as:

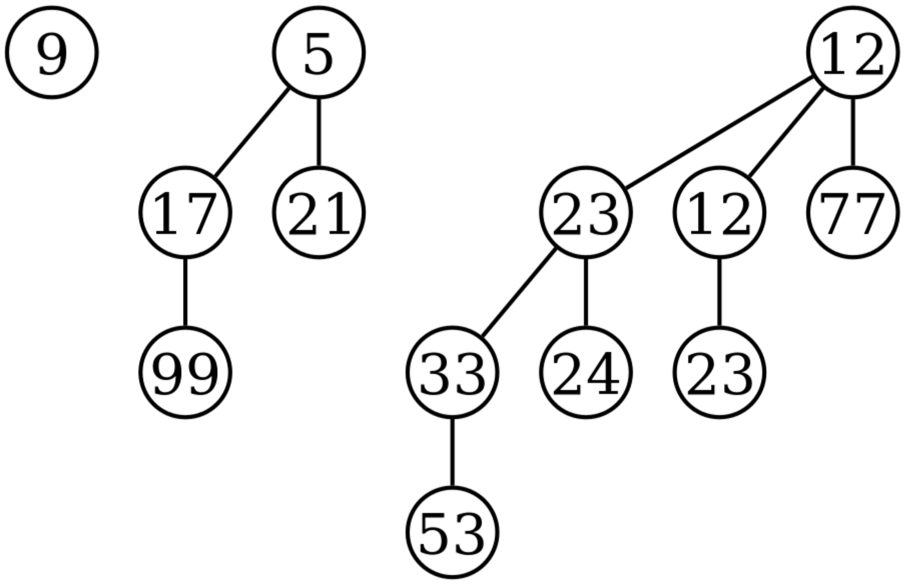
1. When $k = 0$, then $B_k = B_0 = \{R\}$. The binomial tree of order0 consists of a single knot R.
2. When $k > 0$, then $B_k = \{R, B_0, B_1, \ldots, B_{k-1}\}$. The binomial tree of order$k > 0$ consists of the root R and k binomial in Subtrees.

See the following graphic, it visualizes binomial trees B_0 until B_4.



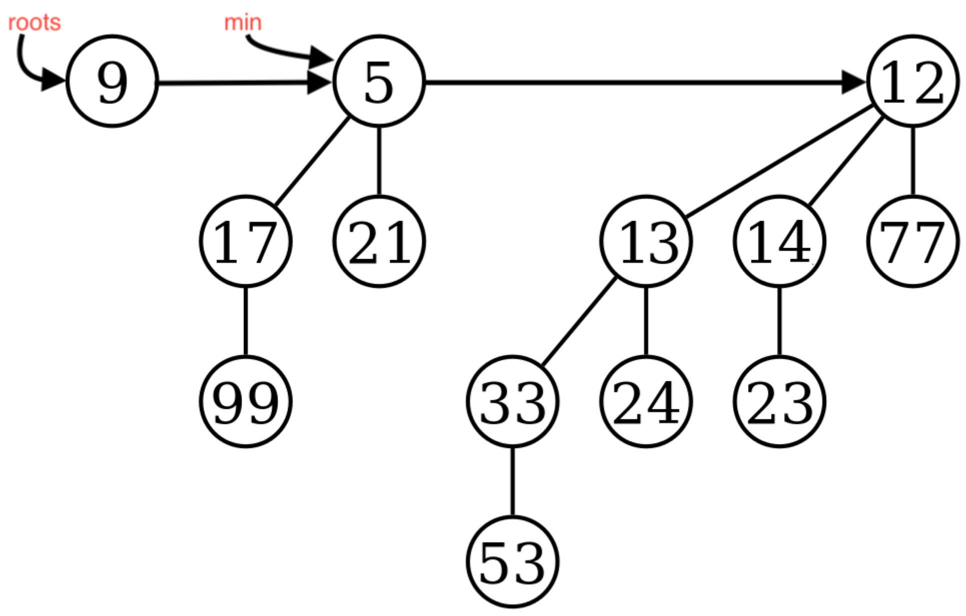
Those

As you can see, a binomial tree of order k + 1 is formed by adding a tree of order k as a child of another tree of order k. *In a (min) heap, the following also applies: the value of the father is always less than or equal to the value of the child. In other words, the further you get towards the root of the tree, the smaller the values become. (Heap invariant)* Here are three trees that each adhere to this invariant:



Those

Since our heap does not always store as many entries as a single binomial tree, our binomial heap stores a list of trees (we use this in your implementation `BinoHeap.roots`). The roots of all binomial trees contained in the binomial heap are stored in this. Any number of entries can be saved. Our implementation is to be effective, therefore, before and after each operation on the data structure (`insert`, `delMin`following invariant, ...) observed: The trees in the `roots`list are sorted in ascending order according to their order and two trees never have the same order. Since we do not know exactly which of the tree roots contains our current minimum, we also keep a reference`BinoHeap.min`With. This should refer to the current minimum after each operation. If the heap is empty, the reference must `null`be. Example:



Source (edited)

Note 1: You can assume that no value will be added to the heap twice. Also, only heaps that do not contain a common value are merged and`decreaseKey`are only used to reduce entries to values that are not yet contained.
Note 2: The signatures are predefined and must not be changed. No separate object / class attributes may be created. `private`- Auxiliary methods are of course allowed and even recommended. The goal is that any implementation of an operation could be used in another developer's implementation without affecting functionality. This is precisely why compliance with the required invariants is so relevant.
*Note 3:*All methods must function properly and also cover edge cases, ie any exceptions that can be thrown count as malfunctions (including null pointer exceptions).
Note 4: A BinoHeap that saves integers (`BinoHeap<Integer>()`) must be able to save at least 1 million entries (with default settings, as long as you have not manipulated your VM, you will not have any problems. If you do not operate your JVM with default settings , it is advisable to switch back to the standard settings temporarily for testing). The model solution manages to process just under 100 million entries, so there is sufficient "overhead" planned.

1. **insert** No results
   This method is supposed to save a new value in the BinoHeap. For this purpose, the transferred value is of the type `T`as a new binomial tree of order0added to the `roots`list. The list now potentially contains two trees with the same order. These two trees now have to be merged into a higher-order binomial tree. You can use the method for this `BinoNode.addChild(BinoNode<T> child)`. It appends the passed child as the last child on the node on which the method is called. This step must be repeated until all of the above-mentioned invariants are observed again.

2. **toString** No results
   In order to be able to visualize our BinoHeap, we would like to `toString()`implement a method next . This should `String`return one in the following form:
   " `min: ␣` <value of the minimum entry of the heap> `\ n`
   `rootlist: \ n`

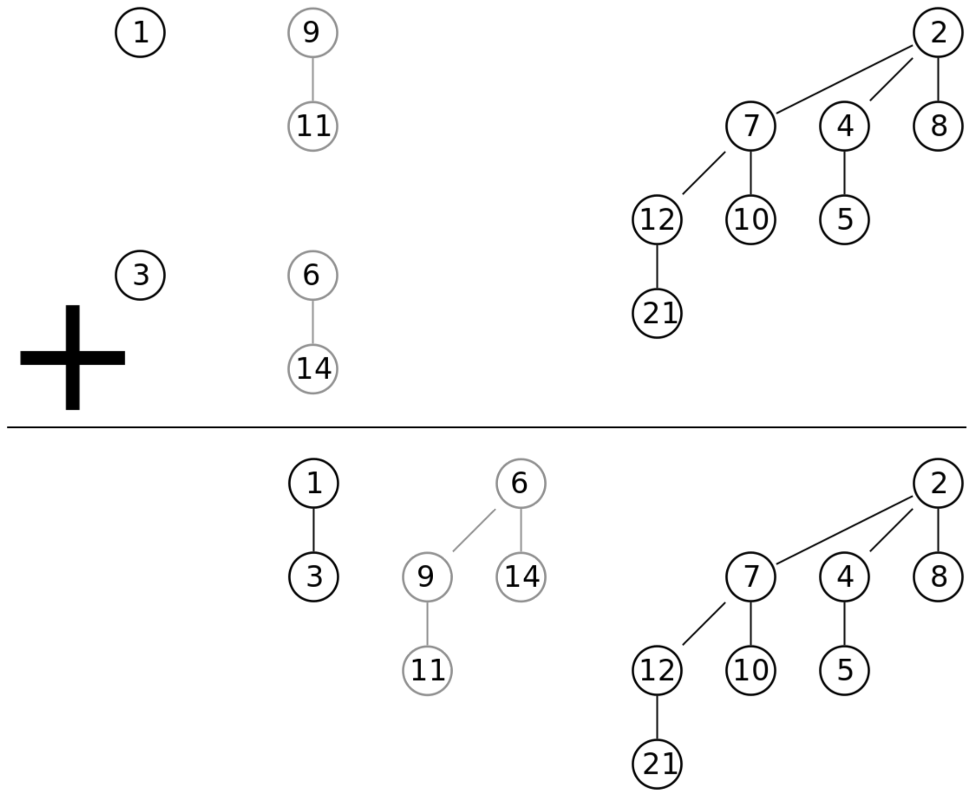
`{\ n`
<string representation of the `0th` tree (smallest order)> `\ n`
<... for each tree the roots list in ascending order ...> `\ n`
<String representation of the i-th tree (highest order)> `\ n`
`}` "
If the heap is empty, no value of the minimum can be displayed, so this is indicated by a `-` (minus) replaced.
Here is an example of an empty and a filled heap:

```
// Empty Heap:
min: -
rootlist:
{
}

 // Heap containing the tree {3} and the tree {1, 2}:
min: 1
rootlist:
{
[3]
[1, [2]]
}
```

3. ? **merge** No results
If this method is called, the current heap and the transferred heap are to be combined. The two `roots`lists are merged for this purpose . The merge order should correspond to the following requirements according to this paragraph. As you can see, it is advisable to implement a helper method. Whether or not you take this advice is up to you.
Note: a naive implementation of inserting each value individually via insert is not accepted.

Merge order:
- The merging always begins with the trees of the smallest order
- If both the "intermediate result" and the two lists to be merged each contain a tree with the same order, the two trees from the lists to be merged are preferably merged.
- Only when only one of the two lists to be merged and the "intermediate result" have a tree of the same order are these trees merged.
  Example: Merge:



Source

4. **findMin** No results
   `findMin()`should return the value of the current minimum. The data structure must not be changed. If there is no minimum, it is `null`returned.

5. **delMin** No results
   This method is used to remove the current minimum from the heap, if it exists. The tree with the minimum as its root is removed from the `roots`list. You will probably notice that the children of the minimum can themselves be viewed as a kind of binomial heap. These children have to be merged back into the heap. The previously described requirements for the merge sequence should again be adhered to.
   Hint: the method `BinoNode.getChildren()`returns an array consisting of the children of the node. The array is sorted in ascending order according to the order of the children.

6. **min** No results
   `min`is a combination of `findMin()`and `delMin()`. It should return the value of the current minimum and remove it from the heap. If such a minimum does not exist, the method `null`returns.

7. **decreaseKey** No results
   `decreaseKey()`can be used to increase the priority of an entry, i.e. to reduce the **value** we save. (Reminder: smaller values have a higher priority.) If the input were to increase the value, the data structure should not be changed. If the heap does not contain the entry to be reduced, the data structure should not be changed either. If the heap contains this value, it must be reduced and then the previously described heap invariants must be restored (ie "the value of the father is always less than or equal to the value of the child"). To do this, you not only have to implement the method `decreaseKey()`in `BinoHeap`, but also the method of the same name in `BinoNode`. As you can see, this requires a recursive approach.

I wish you success!

[Suggested solution] ()
[Tests] ()

FAQ

- *How relevant are the performance tests?*
  The performance tests are public and can therefore be viewed by you from the start. Even if the performance of your implementation is not sufficient, you can still get points (behavior tests). The performance of one subtask will of course not affect the points of another subtask. In order to achieve the full number of points, however, it is necessary to pass the performance tests.

- *Dependencies and consequential errors*
  For some tests (edge cases) you can only receive points if at least one of the public tests of the subtask has been passed. So focus on these first. You shouldn't have any problems with the immediate feedback on Artemis.

- *What happens with zero arguments?*
  If one of the methods is passed one or more null arguments, none should be `NullPointerException`thrown. Furthermore, the heap should not be changed. (Tip: in practice it would of course make sense to throw a suitable `Exception`(e.g. `IllegalArgumentException`) here , but in this task we wanted to focus on the data structure itself and save this part.)

- *toString on an empty heap*
  Please just take a closer look at the example (the first public test by toString also covers exactly this case).

- *Can a heap contain two equivalent elements?*
  The tests only consider cases in which there are no equivalent elements in order to avoid unnecessary confusion. This means that you can assume that no value is inserted into a heap that is already contained, no heaps with common elements are merged, and a key is never reduced to an element that is already contained.

Our dear partners at PUM wish you and your family a Merry Christmas!

You have no results yet