Priority Queues - Heaps

Advanced Algorithms and Data Structures - Lecture 8

Venanzio Capretta

Thursday 26 November 2020

School of Computer Science, University of Nottingham

Priority Queues

A priority queue or heap is a collection of keys/values that can be accessed in order.

We are not interested in searching the collection.

We just want to extract the minimum element efficiently.

Specification: The type `Heap` must have the following methods:

Priority Queues

A priority queue or heap is a collection of keys/values that can be accessed in order.
We are not interested in searching the collection.
We just want to extract the minimum element efficiently.

Specification: The type `Heap` must have the following methods:

- `empty :: Heap`
  The heap containing no values

Priority Queues

A priority queue or heap is a collection of keys/values that can be accessed in order.
We are not interested in searching the collection.
We just want to extract the minimum element efficiently.

Specification: The type `Heap` must have the following methods:

- `empty :: Heap`
  The heap containing no values
- `isEmpty :: Heap -> Bool`
  Tests if the heap is empty

Priority Queues

A priority queue or heap is a collection of keys/values that can be accessed in order.
We are not interested in searching the collection.
We just want to extract the minimum element efficiently.

Specification: The type `Heap` must have the following methods:

- `empty :: Heap`
  The heap containing no values
- `isEmpty :: Heap -> Bool`
  Tests if the heap is empty
- `insert :: Key -> Heap -> Heap`
  Adds a new element to the heap

Priority Queues

A priority queue or heap is a collection of keys/values that can be accessed in order.
We are not interested in searching the collection.
We just want to extract the minimum element efficiently.

Specification: The type `Heap` must have the following methods:

- `empty :: Heap`
  The heap containing no values
- `isEmpty :: Heap -> Bool`
  Tests if the heap is empty
- `insert :: Key -> Heap -> Heap`
  Adds a new element to the heap
- `minimum :: Heap -> Key`
  Returns the smallest element of the heap

Priority Queues

A priority queue or heap is a collection of keys/values that can be accessed in order.
We are not interested in searching the collection.
We just want to extract the minimum element efficiently.

Specification: The type `Heap` must have the following methods:

- `empty :: Heap`
  The heap containing no values
- `isEmpty :: Heap -> Bool`
  Tests if the heap is empty
- `insert :: Key -> Heap -> Heap`
  Adds a new element to the heap
- `minimum :: Heap -> Key`
  Returns the smallest element of the heap
- `extract :: Heap -> (Key,Heap)`
  Removes the smallest element and returns it together with the rest of the heap

Extra Operations

We may also need a function to merge two heaps into one
(used as auxiliary to other methods, for example extraction):

```
union ::  Heap -> Heap -> Heap
```

Another useful operation consists in decreasing a key in the heap:

```
decreaseKey ::  Heap -> Element -> Key -> Heap
```

Assuming that the new key is smaller than the key of the element

This is useful when using heaps for the queue in Dijktra's algorithm
(the relaxation step)

Inefficient Realizations

We will see several implementations of the heap specification and analyze the complexity of the methods

The first naive instantiations can be:

- (Unordered) Lists
- Ordered Lists
- Binary Search Trees

Inefficient Realizations

We will see several implementations of the heap specification and analyze the complexity of the methods

The first naive instantiations can be:

- (Unordered) Lists
- Ordered Lists
- Binary Search Trees

Implementing heaps as (unordered) lists we have:

- `empty = []`, complexity $\Theta(1)$
- `isEmpty`: test if the list is `[]`, complexity $\Theta(1)$
- `insert x h = x :: h`, complexity $\Theta(1)$
- `minimum`
  Search the list for the least element, complexity $\Theta(n)$
- `extract`
  Find minimum, relink the parts before and after it, complexity $\Theta(n)$

Ordered Lists and Binary Search Trees

Ordered Lists and Binary Search Trees

- Implementing heaps as ordered lists:
  The minimum is now the first element of the list: `minimum` and `extract` can be done in $\Theta(1)$
  But `insert` must put the new element in the right place, the complexity is $\Theta(n)$

Ordered Lists and Binary Search Trees

- Implementing heaps as ordered lists:
  The minimum is now the first element of the list: `minimum` and `extract` can be done in $\Theta(1)$
  But `insert` must put the new element in the right place, the complexity is $\Theta(n)$
- Implementing heaps as (balanced) binary search trees `insert`, `minimum`, and `extract` can be done in $O(\log n)$

Ordered Lists and Binary Search Trees

- Implementing heaps as ordered lists:
  The minimum is now the first element of the list: `minimum` and `extract` can be done in $\Theta(1)$
  But `insert` must put the new element in the right place, the complexity is $\Theta(n)$
- Implementing heaps as (balanced) binary search trees `insert`, `minimum`, and `extract` can be done in $O(\log n)$

Since we're not interested in searching the heap, but only in finding the minimum, binary search trees are an overkill. We will see:

- (Leftist) Heaps:
  `minimum` in $\Theta(1)$; `insert` and `extract` in $\Theta(\log n)$
- Fibonacci Heaps:
  `minimum`, `insert`, `union`, `decrease` in $O(1)$ amortized complexity; `extract` in $O(\log n)$

Heap Sort

Using Heaps for Sorting:

There is a correspondence between realizations of the heap data structure and sorting algorithms

$$
\begin{tikzcd}
\text{Lists} & \text{Heaps} \\
l \arrow[dd, dotted, "\texttt{sort}"'] \arrow[dr, bend left, "\texttt{listHeap}"] & \\
& h \arrow[dl, bend left, "\texttt{heapList}"] \\
l' &
\end{tikzcd}
$$

Conversion Between Lists and Heaps

```
listHeap :: [Key] → Heap
listHeap [] = empty
listHeap (x:xs) = insert x (listHeap xs)
```

```
heapList :: Heap → [Key]
heapList h = if (isEmpty h)
             then []
             else let (x,h‘) = extract h
                  in (x:heapList h‘)
```

These are generic conversion functions

For specific heap implementations there may be more efficient algorithms

```
sort :: [Key] → [Key]
sort = heapList ∘ listHeap
```

For some implementations of heaps there may be ad hoc versions of `listHeap` and `heapList` that are more efficient

- For unordered lists, `listHeap` is just the identity
- For ordered lists, `heapList` is just the identity

For some implementations of heaps there may be ad hoc versions of `listHeap` and `heapList` that are more efficient

- For unordered lists, `listHeap` is just the identity
- For ordered lists, `heapList` is just the identity

Different heap realizations correspond to different sorting algorithms

- With unordered lists, we get selection sort
- With ordered lists, we get insertion sort

Binary Heaps

We implement heaps as binary trees

The minimum element will always be at the root of the tree

Binary Heaps

We implement heaps as binary trees

The minimum element will always be at the root of the tree

- Heap Property: The key in every node is smaller or equal to all the keys in its children

Binary Heaps

We implement heaps as binary trees

The minimum element will always be at the root of the tree

- Heap Property: The key in every node is smaller or equal to all the keys in its children
- Balance: The number of elements in the left and right children differ by at most one

Binary Heaps

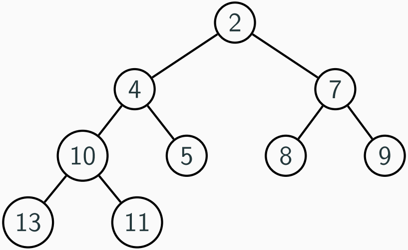
We implement heaps as binary trees

The minimum element will always be at the root of the tree

- Heap Property: The key in every node is smaller or equal to all the keys in its children
- Balance: The number of elements in the left and right children differ by at most one

Stronger condition in AI - the tree is complete: every lever, except the last, is full, and elements in the last level are as far left as possible

A complete heap:

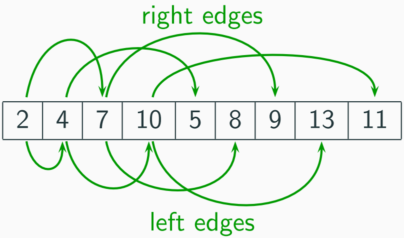


Implementation as an Array

Complete Binary Heaps can be easily implemented as arrays, with the parent-child link implicit (by indexing), instead of using pointers

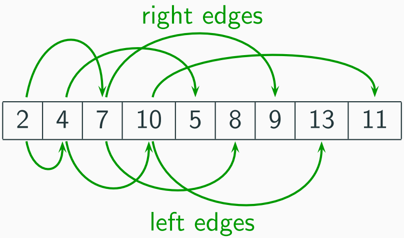
Implementation as an Array

Complete Binary Heaps can be easily implemented as arrays, with the parent-child link implicit (by indexing), instead of using pointers



Implementation as an Array

Complete Binary Heaps can be easily implemented as arrays, with the parent-child link implicit (by indexing), instead of using pointers



This is an example of implicit data structure

It makes it easy to add an element *"at the end"*, which is needed for insertion, and *"get the last element"*, which is needed for elimination.

Functional Realization

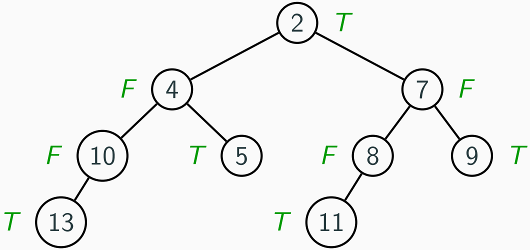
In functional programming, where tree structures are more natural, we use balanced trees.

Each node has a Boolean Flag:

- true if left and right children have the same number of elements
- false if the left child has one more element than the right

```
data BinTree = Empty
             | Node Bool Key BinTree BinTree
```

Functional Example



Formally:

`Node True 2 (Node False 4 ...)  (Node False 7 ...)`

(Leaves are not shown)

It's slightly different from the previous (complete) version:

To keep the balance, 11 is on the right side

Insertion

To insert a new element: (Similar for imperative version)

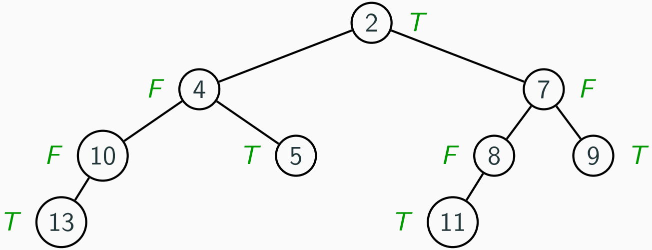
- First place it at the bottom, preserving the balance
- Then move it up, swapping it with higher elements that are bigger

Insertion

To insert a new element: (Similar for imperative version)

- First place it at the bottom, preserving the balance
- Then move it up, swapping it with higher elements that are bigger

`insert 3`

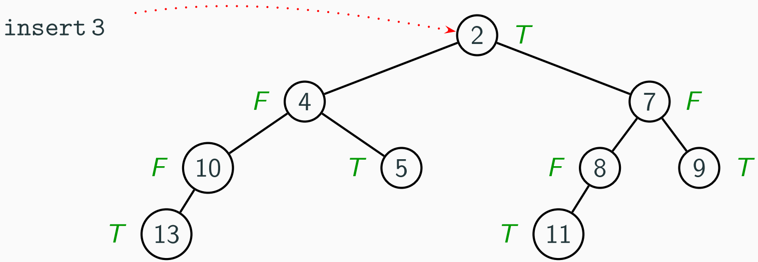


- If the node is true, go left
- If the node is false, go right
- Flip the Boolean flag

Insertion

To insert a new element: (Similar for imperative version)

- First place it at the bottom, preserving the balance
- Then move it up, swapping it with higher elements that are bigger

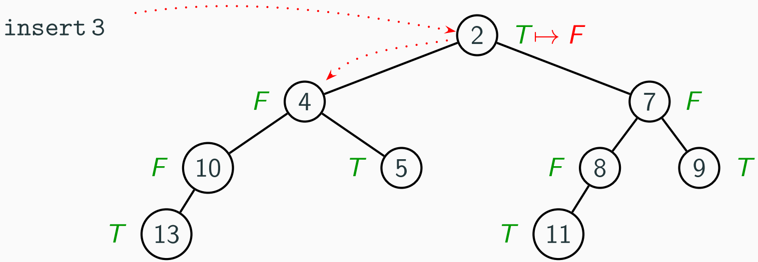


- If the node is true, go left
- If the node is false, go right
- Flip the Boolean flag

Insertion

To insert a new element: (Similar for imperative version)

- First place it at the bottom, preserving the balance
- Then move it up, swapping it with higher elements that are bigger

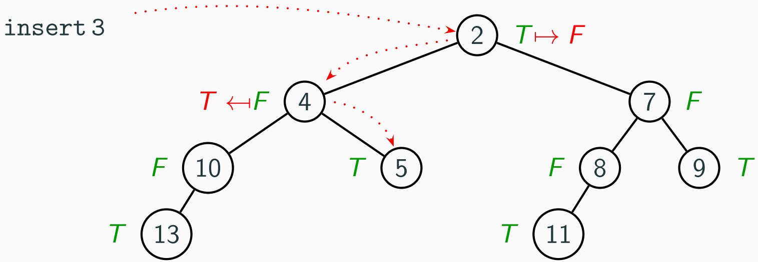


- If the node is true, go left
- If the node is false, go right
- Flip the Boolean flag

Insertion

To insert a new element: (Similar for imperative version)

- First place it at the bottom, preserving the balance
- Then move it up, swapping it with higher elements that are bigger

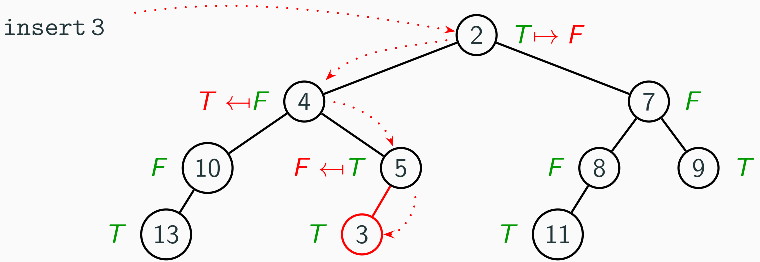


- If the node is true, go left
- If the node is false, go right
- Flip the Boolean flag

Insertion

To insert a new element: (Similar for imperative version)

- First place it at the bottom, preserving the balance
- Then move it up, swapping it with higher elements that are bigger

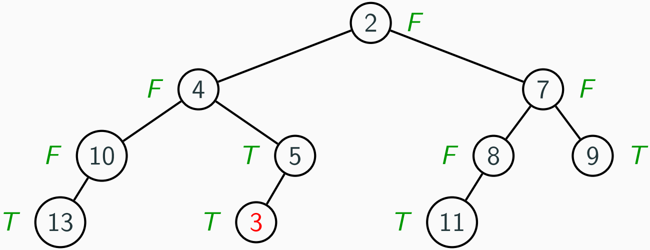


- If the node is true, go left
- If the node is false, go right
- Flip the Boolean flag

Swapping

We must now fix the Heap Property:

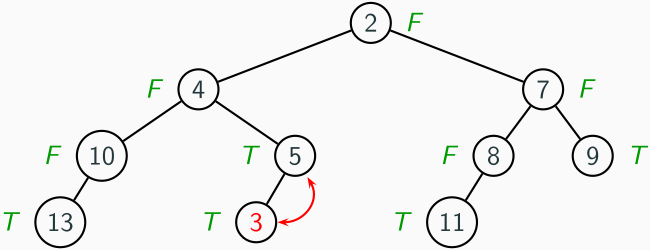
Move the new element upward until it gets to the right place



Swapping

We must now fix the Heap Property:

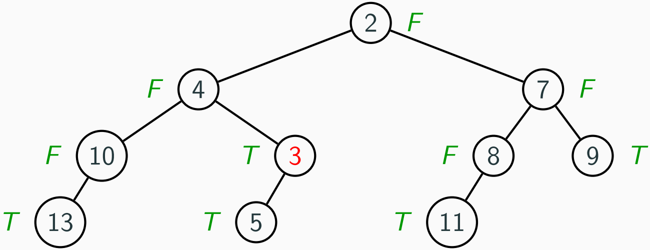
Move the new element upward until it gets to the right place



Swapping

We must now fix the Heap Property:

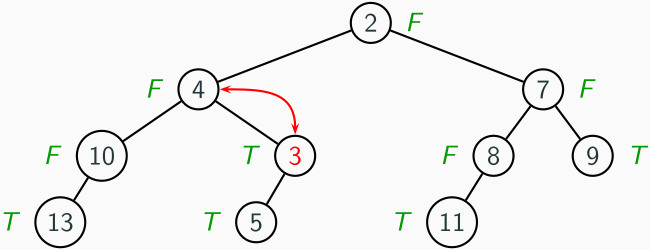
Move the new element upward until it gets to the right place



Swapping

We must now fix the Heap Property:

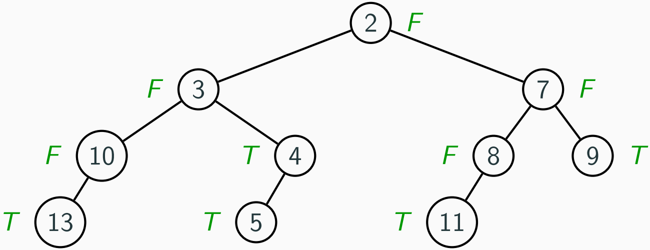
Move the new element upward until it gets to the right place



Swapping

We must now fix the Heap Property:

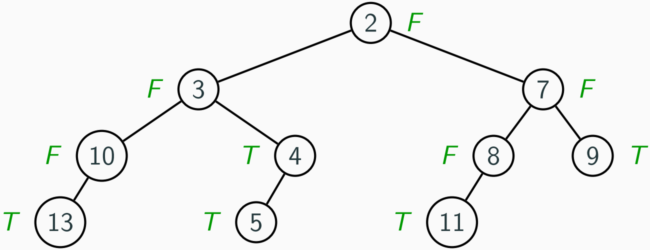
Move the new element upward until it gets to the right place



Swapping

We must now fix the Heap Property:

Move the new element upward until it gets to the right place

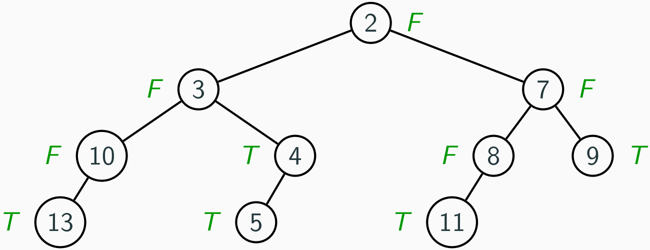


In functional programming we can optimize the two phases:

Do only one pass of the tree, swap as you move down

Swapping

We must now fix the Heap Property:
Move the new element upward until it gets to the right place



In functional programming we can optimize the two phases:
Do only one pass of the tree, swap as you move down

Complexity: The depth of the tree is $O(\log n)$
So the complexity of `insert` is $O(\log n)$

Haskell Version

The previous version is good for the array implementation

Placing the element *"at the bottom"* is easy:

Just add it at the end of the array

In functional programming, we can do the swapping as we descend the tree:

```
insert :: Key → BinHeap → BinHeap
insert x Empty = Node True x Empty Empty
insert x (Node True y h1 h2) = Node False (min x y)
                                    (insert (max x y) h1)
                                    h2
insert x (Node False y h1 h2) = Node True (min x y)
                                     h1
                                     (insert (max x y) h2)
```

`minimum` is just the root of the tree

Extraction

`extract` is more complicated:

- We extract the minimum, which is the root
- We are left with the two children; We need to merge them

Extraction

`extract` is more complicated:

- We extract the minimum, which is the root
- We are left with the two children; We need to merge them

Idea:

- Recursively extract the minimum from one of the children (choose which so the balance is preserved)
- Use it as the new root
- Swap it down inside the other child if necessary

Extraction

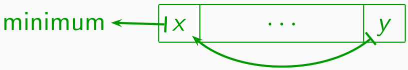
`extract` is more complicated:

- We extract the minimum, which is the root
- We are left with the two children; We need to merge them

Idea:

- Recursively extract the minimum from one of the children (choose which so the balance is preserved)
- Use it as the new root
- Swap it down inside the other child if necessary

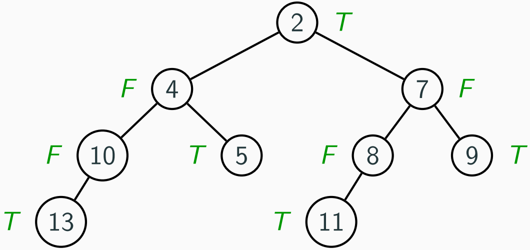
(In the array representation, choose the last element as the new root:



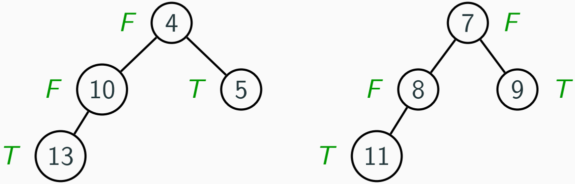
Then move it down if necessary)

Example Extraction

Extract the minimum from this tree:

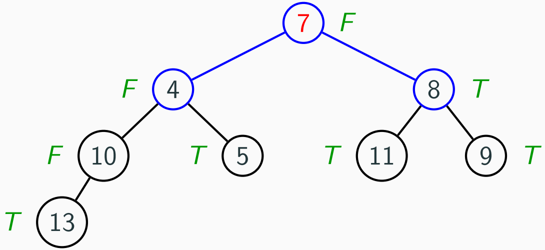


The minimum is 2; we are left with the children



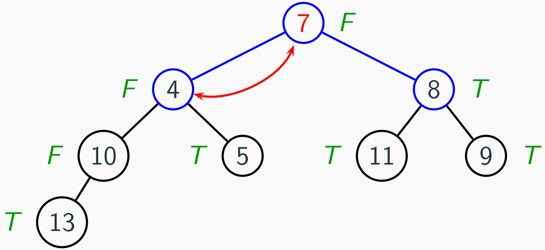
Merging

Since the two trees have the same number of elements
(the Boolean value at the root was true)
we recursively extract from the right tree
and use its minimum as the new root:



Merging

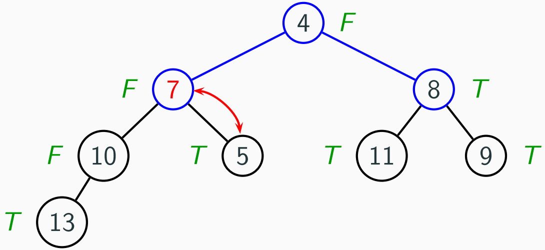
Since the two trees have the same number of elements
(the Boolean value at the root was true)
we recursively extract from the right tree
and use its minimum as the new root:



Then we *swap down* the new root to move it to the right place

Merging

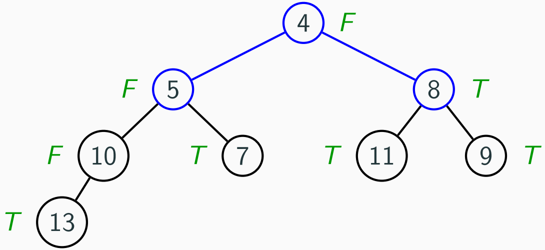
Since the two trees have the same number of elements
(the Boolean value at the root was true)
we recursively extract from the right tree
and use its minimum as the new root:



Then we *swap down* the new root to move it to the right place

Merging

Since the two trees have the same number of elements
(the Boolean value at the root was true)
we recursively extract from the right tree
and use its minimum as the new root:



Then we *swap down* the new root to move it to the right place

Swapping Function

Auxiliary function to move an element down to the right position:

```
siftDown :: Key → BinHeap → (Key,BinHeap)
siftDown x Empty = (x, Empty)
siftDown x h@(Node b y h1 h2) =
  if x>y then (y,downHeap (Node b x h1 h2))
         else (x,h)

downHeap :: BinHeap → BinHeap
downHeap Empty = Empty
downHeap (Node b x h1 h2) =
  if h1 ≤ h2
  then let (x‘,h1‘) = siftDown x h1
       in (Node b x‘ h1‘ h2)
  else let (x‘,h2‘) = siftDown x h2
       in (Node b x‘ h1 h2‘)
```

The order relation $\leq$ on trees is true if the root of `h1` is smaller than the root of `h2` or `h2` is empty (if `h1` empty, then `h2` empty, by balance)

Extraction in Haskell

Finally we can implement extraction:

```
extract :: BinTree → (Key,BinTree)
extract (Node b x Empty Empty) = (x,Empty)
extract (Node True x h1 h2) =
  let (y,h2‘) = extract h2
      (z,h1‘) = siftDown y h1
  in  (x, Node False z h1‘ h2‘)
extract (Node False x h1 h2) =
   ...      (similar to previous case)
```

Complexity

The complexity of `siftDown` and `downHeap` is the same:
`siftDown` just calls `downHeap` after making a constant-time operation

$$T_0(n) = T_0(n/2) + c_0$$

- The recursive call to `siftDown` is on either `h1` or `h2` which have half of the elements (plus or minus 1)
- At each call we do a constant number of extra steps
- Therefore $T_0(n) = \Theta(\log n)$

Complexity

The complexity of `siftDown` and `downHeap` is the same:
`siftDown` just calls `downHeap` after making a constant-time operation

$$T_0(n) = T_0(n/2) + c_0$$

- The recursive call to `siftDown` is on either `h1` or `h2` which have half of the elements (plus or minus 1)
- At each call we do a constant number of extra steps
- Therefore $T_0(n) = \Theta(\log n)$

Complexity of `extract`:

$$T_1 = T_1(n/2) + T_0(n/2) + c_1 = T_1(n/2) + \Theta(\log n)$$

- The call to `siftDown` gives the term $T_0(n/2)$
- Therefore $T_1(n) = \Theta(\log n)$