Programming in Haskell: Lecture 25

S P Suresh

November 11, 2019

Priority queues

- **Priority queue**: a queue, with each element having a **priority**

Priority queues

- **Priority queue**: a queue, with each element having a **priority**
- Elements exit the queue by priority, not in the order they entered

Priority queues

- **Priority queue**: a queue, with each element having a **priority**
- Elements exit the queue by priority, not in the order they entered
- Each element in a priority queue is a pair `(p,v)`, where `p` is the priority and `v` is the value

Priority queues

- **Priority queue**: a queue, with each element having a **priority**
- Elements exit the queue by priority, not in the order they entered
- Each element in a priority queue is a pair `(p,v)`, where `p` is the priority and `v` is the value
- The priorities are values of some type `a` belonging to the class **`Ord`**

Priority queues

- **Priority queue**: a queue, with each element having a **priority**
- Elements exit the queue by priority, not in the order they entered
- Each element in a priority queue is a pair `(p,v)`, where `p` is the priority and `v` is the value
- The priorities are values of some type `a` belonging to the class `Ord`
- We only show how to maintain priorities, not the values

Priority queues

- **Priority queue**: a queue, with each element having a **priority**
- Elements exit the queue by priority, not in the order they entered
- Each element in a priority queue is a pair `(p,v)`, where `p` is the priority and `v` is the value
- The priorities are values of some type `a` belonging to the class **`Ord`**
- We only show how to maintain priorities, not the values
- We assume priorities are distinct

Priority queues

- **Priority queue**: a queue, with each element having a **priority**
- Elements exit the queue by priority, not in the order they entered
- Each element in a priority queue is a pair `(p,v)`, where `p` is the priority and `v` is the value
- The priorities are values of some type `a` belonging to the class **`Ord`**
- We only show how to maintain priorities, not the values
- We assume priorities are distinct
    - Tie breaker: the time at which element entered the queue

Priority queues

- **Priority queue**: a queue, with each element having a **priority**
- Elements exit the queue by priority, not in the order they entered
- Each element in a priority queue is a pair `(p,v)`, where `p` is the priority and `v` is the value
- The priorities are values of some type `a` belonging to the class **`Ord`**
- We only show how to maintain priorities, not the values
- We assume priorities are distinct
  - Tie breaker: the time at which element entered the queue
- Priority queue operations

Priority queues

- **Priority queue**: a queue, with each element having a **priority**
- Elements exit the queue by priority, not in the order they entered
- Each element in a priority queue is a pair `(p,v)`, where `p` is the priority and `v` is the value
- The priorities are values of some type `a` belonging to the class **`Ord`**
- We only show how to maintain priorities, not the values
- We assume priorities are distinct
  - Tie breaker: the time at which element entered the queue
- Priority queue operations
  - `insert` – insert an element into the queue

Priority queues

- **Priority queue**: a queue, with each element having a **priority**
- Elements exit the queue by priority, not in the order they entered
- Each element in a priority queue is a pair `(p,v)`, where `p` is the priority and `v` is the value
- The priorities are values of some type `a` belonging to the class **`Ord`**
- We only show how to maintain priorities, not the values
- We assume priorities are distinct
  - Tie breaker: the time at which element entered the queue
- Priority queue operations
  - `insert` – insert an element into the queue
  - `deleteMax` – delete the maximum element from the queue

Priority queue – implementations

- Unsorted lists

Priority queue – implementations

- Unsorted lists
  - `insert` – $O(1)$ time

Priority queue – implementations

- Unsorted lists
  - `insert` – $O(1)$ time
  - `deleteMax` – $O(n)$ time

Priority queue – implementations

- Unsorted lists
  - `insert` – $O(1)$ time
  - `deleteMax` – $O(n)$ time
- Sorted lists – descending order of priority

Priority queue – implementations

- Unsorted lists
  - `insert` – $O(1)$ time
  - `deleteMax` – $O(n)$ time
- Sorted lists – descending order of priority
  - `insert` – $O(n)$ time

Priority queue – implementations

- Unsorted lists
  - `insert` – $O(1)$ time
  - `deleteMax` – $O(n)$ time
- Sorted lists – descending order of priority
  - `insert` – $O(n)$ time
  - `deleteMax` – $O(1)$ time

Priority queue – implementations

- Unsorted lists
  - `insert` – $O(1)$ time
  - `deleteMax` – $O(n)$ time
- Sorted lists – descending order of priority
  - `insert` – $O(n)$ time
  - `deleteMax` – $O(1)$ time
- AVL trees

Priority queue – implementations

- Unsorted lists
  - `insert` – $O(1)$ time
  - `deleteMax` – $O(n)$ time
- Sorted lists – descending order of priority
  - `insert` – $O(n)$ time
  - `deleteMax` – $O(1)$ time
- AVL trees
  - `insert` – $O(\log n)$ time

Priority queue – implementations

- Unsorted lists
  - `insert` – $O(1)$ time
  - `deleteMax` – $O(n)$ time
- Sorted lists – descending order of priority
  - `insert` – $O(n)$ time
  - `deleteMax` – $O(1)$ time
- AVL trees
  - `insert` – $O(\log n)$ time
  - `deleteMax` – $O(\log n)$ time

Heaps

- A **heap** is another way to implement priority queues

Heaps

- A **heap** is another way to implement priority queues
- To determine the maximum, it is not necessary that elements be sorted

Heaps

- A **heap** is another way to implement priority queues
- To determine the maximum, it is not necessary that elements be sorted
- We need to keep track of the maximum

Heaps

- A **heap** is another way to implement priority queues
- To determine the maximum, it is not necessary that elements be sorted
- We need to keep track of the maximum
- Also the possible second maximum, to be installed as the new maximum after `deleteMax`

Heaps

- A **heap** is another way to implement priority queues
- To determine the maximum, it is not necessary that elements be sorted
- We need to keep track of the maximum
- Also the possible second maximum, to be installed as the new maximum after `deleteMax`
- The next maximum ...

Heaps

- A **heap** is another way to implement priority queues
- To determine the maximum, it is not necessary that elements be sorted
- We need to keep track of the maximum
- Also the possible second maximum, to be installed as the new maximum after `deleteMax`
- The next maximum ...
- We look at **max-heaps** in this lecture, **min-heaps** are analogous

Heaps

- A **heap** is a binary tree satisfying the **heap property**

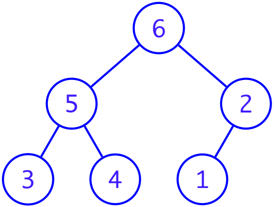
Heaps

- A **heap** is a binary tree satisfying the **heap property**
- **Heap property** – The value at every node is larger than the value at its two children

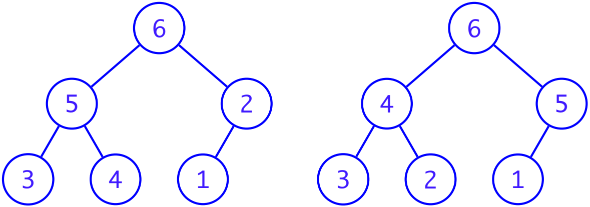
Heaps

- A **heap** is a binary tree satisfying the **heap property**
- **Heap property** – The value at every node is larger than the value at its two children
- In a heap, the largest element is always at the root

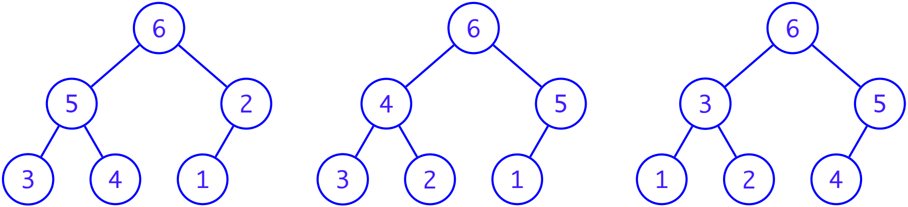
Example heaps



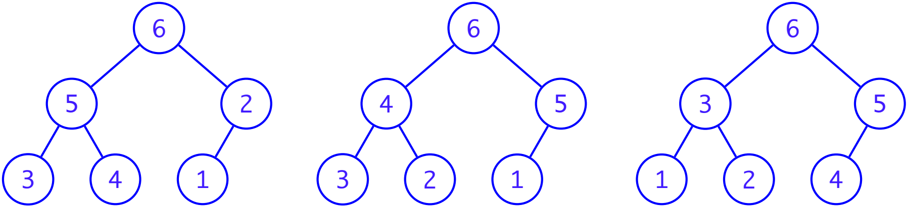
Example heaps



Example heaps



Example heaps



- These three heaps are also **leftist heaps**

Leftist Heaps

- **Leftist heap** – At every node, the size of the left subtree is greater than or equal to the size of the right subtree

Leftist Heaps

- **Leftist heap** – At every node, the size of the left subtree is greater than or equal to the size of the right subtree
- Denote by $\mathrm{lrs}(h)$ the length of the right spine of a heap h

Leftist Heaps

- **Leftist heap** – At every node, the size of the left subtree is greater than or equal to the size of the right subtree
- Denote by $\mathrm{lrs}(h)$ the length of the right spine of a heap h
- Let $\mathrm{size}(h) = n$ and h_1 and h_2 be the left and right subtrees of h

Leftist Heaps

- **Leftist heap** – At every node, the size of the left subtree is greater than or equal to the size of the right subtree
- Denote by $\mathrm{lrs}(h)$ the length of the right spine of a heap h
- Let $\mathrm{size}(h) = n$ and h_1 and h_2 be the left and right subtrees of h
  - $n = \mathrm{size}(h_1) + \mathrm{size}(h_2) + 1$ and $\mathrm{size}(h_1) \geq \mathrm{size}(h_2)$

Leftist Heaps

- **Leftist heap** – At every node, the size of the left subtree is greater than or equal to the size of the right subtree
- Denote by $\mathrm{lrs}(h)$ the length of the right spine of a heap h
- Let $\mathrm{size}(h) = n$ and h_1 and h_2 be the left and right subtrees of h
  - $n = \mathrm{size}(h_1) + \mathrm{size}(h_2) + 1$ and $\mathrm{size}(h_1) \geq \mathrm{size}(h_2)$
  - So $\mathrm{size}(h_2) \leq n/2$

Leftist Heaps

- **Leftist heap** – At every node, the size of the left subtree is greater than or equal to the size of the right subtree
- Denote by $\mathrm{lrs}(h)$ the length of the right spine of a heap h
- Let $\mathrm{size}(h) = n$ and h_1 and h_2 be the left and right subtrees of h
    - $n = \mathrm{size}(h_1) + \mathrm{size}(h_2) + 1$ and $\mathrm{size}(h_1) \geq \mathrm{size}(h_2)$
    - So $\mathrm{size}(h_2) \leq n/2$
- $\mathrm{lrs}(h) = 1 + \mathrm{lrs}(h_2)$

Leftist Heaps

- **Leftist heap** – At every node, the size of the left subtree is greater than or equal to the size of the right subtree
- Denote by $\mathrm{lrs}(h)$ the length of the right spine of a heap h
- Let $\mathrm{size}(h) = n$ and h_1 and h_2 be the left and right subtrees of h
  - $n = \mathrm{size}(h_1) + \mathrm{size}(h_2) + 1$ and $\mathrm{size}(h_1) \geq \mathrm{size}(h_2)$
  - So $\mathrm{size}(h_2) \leq n/2$
- $\mathrm{lrs}(h) = 1 + \mathrm{lrs}(h_2)$
- **Claim**: If $\mathrm{size}(h) = n$, $\mathrm{lrs}(h) \leq \log n + 1$

Right spine of a leftist heap

- **Claim**: If $\text{size}(h) = n$, $\text{lrs}(h) \leq \log n + 1$

Right spine of a leftist heap

- **Claim**: If $\mathrm{size}(h) = n$, $\mathrm{lrs}(h) \leq \log n + 1$
- **Proof**: If $n = 1$, $\mathrm{lrs}(h) = 1 \leq \log 1 + 1$

Right spine of a leftist heap

- **Claim**: If $\text{size}(h) = n$, $\text{lrs}(h) \leq \log n + 1$
- **Proof**: If $n = 1$, $\text{lrs}(h) = 1 \leq \log 1 + 1$
- If $n > 1$ and h_2 is the right subheap of h,

$$\begin{aligned}\text{lrs}(h) &= 1 + \text{lrs}(h_2) \\ &\leq 1 + (\log n/2 + 1) \\ &\leq 1 + (\log n - 1 + 1) \\ &= \log n + 1\end{aligned}$$

A heap module

- Just as we stored the height at every node of an AVL tree …

A heap module

- Just as we stored the height at every node of an AVL tree ...
- we store the size of the tree at each node of a leftist heap

```
module Heap(Heap, emptyHeap, isEmpty,
              union, insert, findMax, deleteMax,
                      createHeap, toList) where
data Heap a = Nil | Node Int a (Heap a) (Heap a)
emptyHeap :: Heap a
emptyHeap = Nil
isEmpty :: Heap a -> Bool
isEmpty Nil = True
isEmpty _   = False
```

A heap module

```haskell
size Nil              = 0
size (Node s _ _ _) = s
root (Node _ x _ _) = x

isHeap :: Ord a => Heap a -> Bool
isHeap Nil                = True
isHeap (Node s x hl hr) = s == 1 + sl + sr && sl >= sr &&
                          (isEmpty hl || x >= root hl) &&
                          (isEmpty hr || x >= root hr) &&
                          isHeap hl && isHeap hr
    where (sl, sr)        = (size hl, size hr)
```

Union of leftist heaps

- Union of two leftist heaps of size m and n

Union of leftist heaps

- Union of two leftist heaps of size m and n
- The right spines are of length $O(\log m)$ and $O(\log n)$

Union of leftist heaps

- Union of two leftist heaps of size m and n
- The right spines are of length $O(\log m)$ and $O(\log n)$
- Union is implemented by walking down the right spines

Union of leftist heaps

- Union of two leftist heaps of size m and n
- The right spines are of length $O(\log m)$ and $O(\log n)$
- Union is implemented by walking down the right spines
    - Works in $O(\log m + \log n)$ time

Union of leftist heaps

- Union of two leftist heaps of size m and n
- The right spines are of length $O(\log m)$ and $O(\log n)$
- Union is implemented by walking down the right spines
  - Works in $O(\log m + \log n)$ time
- Violation of leftist property at root is handled as follows:

```
realign :: Heap a -> Heap a
realign Nil                = Nil
realign h@(Node s x hl hr)
    | size hl >= size hr = h
    | otherwise          = Node s x hr hl
```

Union of leftist heaps

```
union :: Ord a => Heap a -> Heap a -> Heap a
union h Nil      = h
union Nil h      = h
union h1@(Node s1 x h1l h1r) h2@(Node s2 y h2l h2r)
    | x >= y    = realign
                     (Node (s1+s2) x h1l (union h1r h2))
    | otherwise = realign
                     (Node (s1+s2) y h2l (union h1 h2r))
```

Heap operations

- Important heap operations implemented using `union`

Heap operations

- Important heap operations implemented using `union`
- `insert` and `deleteMax` take $O(\log n)$ time

```
insert :: Ord a => a -> Heap a -> Heap a
findMax :: Heap a -> Maybe a
deleteMax :: Ord a => Heap a -> (Maybe a, Heap a)

insert x h                 = union (Node 1 x Nil Nil) h
findMax h                  = if isEmpty h then Nothing
                             else Just (root h)
deleteMax Nil              = (Nothing, Nil)
deleteMax (Node _ x hl hr) = (Just x, union hl hr)
```

Creating heaps from lists

- We can form a leftist heap from a list in linear time

Creating heaps from lists

- We can form a leftist heap from a list in linear time
- **Strategy** – Create a size-balanced leftist **tree**

Creating heaps from lists

- We can form a leftist heap from a list in linear time
- **Strategy** – Create a size-balanced leftist **tree**
- Restore heap property

Creating heaps from lists

- We can form a leftist heap from a list in linear time
- **Strategy** – Create a size-balanced leftist **tree**
- Restore heap property
- Creating a leftist tree is just the linear time `createTree`

Creating heaps from lists

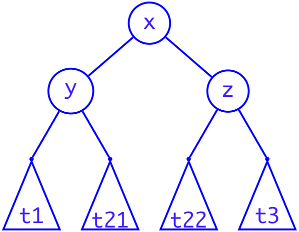
- We can form a leftist heap from a list in linear time
- **Strategy** – Create a size-balanced leftist **tree**
- Restore heap property
- Creating a leftist tree is just the linear time `createTree`
- Since `createTree` produces a size-balanced tree, height is $\log n$

Creating a leftist tree

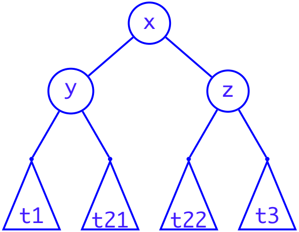
```
leftistTree :: [a] -> Heap a
leftistTree l = fst (go (length l) l)

go :: Int -> [a] -> (Heap a, [a])
go 0 xs = (Nil, xs)
go n xs = (Node s y hl hr, zs)
    where
        m = n `div` 2
        (hl, y:ys) = go m xs
        (hr, zs) = go (n-m-1) ys
        s = 1 + size hl + size hr
```

Repairing heaps

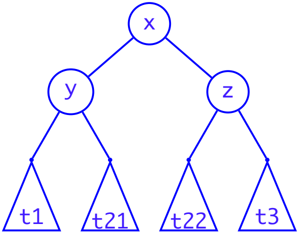


Repairing heaps



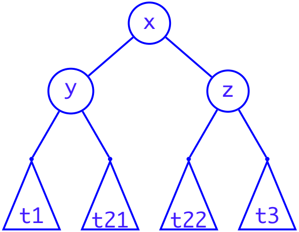
- Assume that subtrees rooted at `y` and `z` are heaps

Repairing heaps



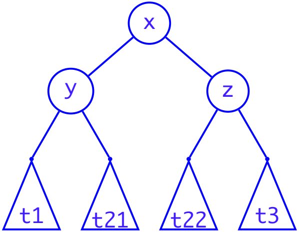
- Assume that subtrees rooted at `y` and `z` are heaps
- How to ensure that tree rooted at `x` is a heap?

Repairing heaps



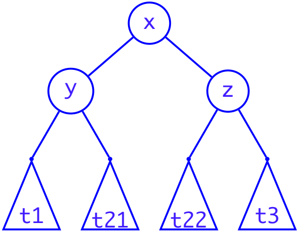
- Assume that subtrees rooted at `y` and `z` are heaps
- How to ensure that tree rooted at `x` is a heap?
- If `x >= max y z` all is okay

Repairing heaps



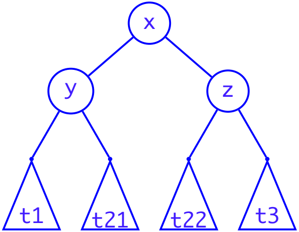
- Assume that subtrees rooted at `y` and `z` are heaps
- How to ensure that tree rooted at `x` is a heap?
- If `x >= max y z` all is okay
- Else swap `x` with `max y z`, say `z`

Repairing heaps



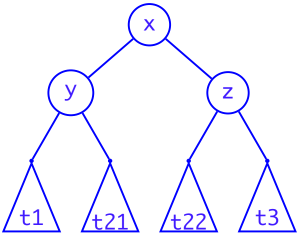
- Assume that subtrees rooted at `y` and `z` are heaps
- How to ensure that tree rooted at `x` is a heap?
- If `x >= max y z` all is okay
- Else swap `x` with `max y z`, say `z`
- The heap property is satisfied at the root

Repairing heaps



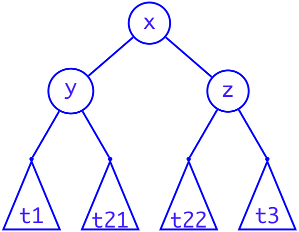
- Assume that subtrees rooted at `y` and `z` are heaps
- How to ensure that tree rooted at `x` is a heap?
- If `x >= max y z` all is okay
- Else swap `x` with `max y z`, say `z`
- The heap property is satisfied at the root
- The left subtree is untouched

Repairing heaps



- Assume that subtrees rooted at `y` and `z` are heaps
- How to ensure that tree rooted at `x` is a heap?
- If `x >= max y z` all is okay
- Else swap `x` with `max y z`, say `z`
- The heap property is satisfied at the root
- The left subtree is untouched
- But the right subtree may no longer be a heap

Repairing heaps



- Assume that subtrees rooted at `y` and `z` are heaps
- How to ensure that tree rooted at `x` is a heap?
- If `x >= max y z` all is okay
- Else swap `x` with `max y z`, say `z`
- The heap property is satisfied at the root
- The left subtree is untouched
- But the right subtree may no longer be a heap
- Recursively repair it – **sifting**

Violation of heap property

- `badness` tells us how the heap property is violated at the root:

```
data Badness = NoBad | LeftBad | RightBad
badness :: Ord a => Heap a -> Badness
badness (Node _ x hl hr)
    | x >= m = NoBad
    | y >= m = LeftBad
    | z >= m = RightBad
    where
        y    = if isEmpty hl then x else root hl
        z    = if isEmpty hr then x else root hr
        m    = maximum [x,y,z]
```

Violation of heap property

- badness tells us how the heap property is violated at the root:

```
data Badness = NoBad | LeftBad | RightBad
badness :: Ord a => Heap a -> Badness
badness (Node _ x hl hr)
    | x >= m = NoBad
    | y >= m = LeftBad
    | z >= m = RightBad
    where
        y    = if isEmpty hl then x else root hl
        z    = if isEmpty hr then x else root hr
        m    = maximum [x,y,z]
```

- Constant time operation

Exchange operations

- To restore the heap property, we need to exchange the root with either the left or right child

Exchange operations

- To restore the heap property, we need to exchange the root with either the left or right child
- Constant time exchange operations:

```
xchngLeft :: Heap a -> Heap a
xchngLeft (Node s x (Node sl y hll hlr) hr)
          = Node s y (Node sl x hll hlr) hr

xchngRight :: Heap a -> Heap a
xchngRight (Node s x hl (Node sr y hrl hrr))
          = Node s y hl (Node sr x hrl hrr)
```

Repairing heaps – `sift`

- Recursively sift the root down the tree till there is no badness

```haskell
sift :: Ord a => Heap a -> Heap a
sift Nil = Nil
sift h   = case badness h of
    NoBad    -> h
    LeftBad  -> let Node s1 x1 hl1 hr1 = xchngLeft h
                in Node s1 x1 (sift hl1) hr1
    RightBad -> let Node s2 x2 hl2 hr2 = xchngRight h
                in Node s2 x2 hl2 (sift hr2)
```

Repairing heaps – `sift`

- Recursively sift the root down the tree till there is no badness

```
sift :: Ord a => Heap a -> Heap a
sift Nil = Nil
sift h   = case badness h of
    NoBad    -> h
    LeftBad  -> let Node s1 x1 hl1 hr1 = xchngLeft h
                in Node s1 x1 (sift hl1) hr1
    RightBad -> let Node s2 x2 hl2 hr2 = xchngRight h
                in Node s2 x2 hl2 (sift hr2)
```

- Running time is $O(\text{height of heap})$

Repairing heaps – `sift`

- Recursively sift the root down the tree till there is no badness

```
sift :: Ord a => Heap a -> Heap a
sift Nil = Nil
sift h   = case badness h of
    NoBad    -> h
    LeftBad  -> let Node s1 x1 hl1 hr1 = xchngLeft h
                in Node s1 x1 (sift hl1) hr1
    RightBad -> let Node s2 x2 hl2 hr2 = xchngRight h
                in Node s2 x2 hl2 (sift hr2)
```

- Running time is $O(\text{height of heap})$
- Applied on a size-balanced tree, it is $O(\log n)$

Repairing heaps – heapify

- heapify transforms a tree into a heap

```
heapify :: Ord a => Heap a -> Heap a
heapify Nil = Nil
heapify (Node s x hl hr)
            = sift (Node s x
                             (heapify hl)
                             (heapify hr))

createHeap :: Ord a => [a] -> Heap a
createHeap = heapify . leftistTree
```

Repairing heaps – `heapify`

- `heapify` transforms a tree into a heap

```haskell
heapify :: Ord a => Heap a -> Heap a
heapify Nil = Nil
heapify (Node s x hl hr)
             = sift (Node s x
                          (heapify hl)
                          (heapify hr))

createHeap :: Ord a => [a] -> Heap a
createHeap = heapify . leftistTree
```

- Applied on a size-balanced tree, `heapify` takes $O(n)$ time

Time complexity of `heapify`

- **Proof**: On size-balanced trees, choose c such that

Time complexity of `heapify`

- **Proof**: On size-balanced trees, choose c such that
  - $T(1) \leq c$

Time complexity of `heapify`

- **Proof**: On size-balanced trees, choose c such that
  - $T(1) \leq c$
  - $T(n) \leq c \log n + 2T(n/2)$

Time complexity of `heapify`

- **Proof**: On size-balanced trees, choose c such that
  - $T(1) \leq c$
  - $T(n) \leq c \log n + 2T(n/2)$
- Letting $n = 2^k$,

$$
\begin{aligned}
T(2^k) &= ck + 2T(2^{k-1}) = ck + 2[c(k-1) + 2T(2^{k-2})] \\
&= ck + 2c(k-1) + 2^2[c(k-2) + 2T(2^{k-3})] \\
&= \cdots \\
&= ck + 2c(k-1) + 2^2c(k-2) + \cdots + 2^{k-1}[c(k-k+1) + 2T(2^{k-k})] \\
&= c[k + 2(k-1) + 2^2(k-2) + \cdots + 2^{k-1}(k-k+1) + 2^k]
\end{aligned}
$$

Time complexity of `heapify`

- **Proof**: On size-balanced trees, choose c such that

Time complexity of `heapify`

- **Proof**: On size-balanced trees, choose c such that
  - $T(1) \leq c$

Time complexity of `heapify`

- **Proof**: On size-balanced trees, choose c such that
  - $T(1) \le c$
  - $T(n) \le c \log n + 2T(n/2)$

Time complexity of `heapify`

- **Proof**: On size-balanced trees, choose c such that
  - $T(1) \leq c$
  - $T(n) \leq c \log n + 2T(n/2)$
- Letting $n = 2^k$,

$$
\begin{aligned}
T(2^k) &= c[k && +2(k-1) && +2^2(k-2) && +\cdots+2^{k-1}(k-k+1) && + && 2^k] \\
2T(2^k) &= c[&& 2(k-0) && +2^2(k-1) && +\cdots+2^{k-1}(k-k+2) && +2^k \cdot 1 && +2^{k+1}] \\
T(2^k) &= c[-k && +2 && +2^2 && +\cdots+2^{k-1} && +2^k && +2^k] \\
&= c[-k && +2^{k+1}-2 && && && && +2^k] \\
&= c[-\log n && +2n-2 && && && && +n] \\
&= O(n)
\end{aligned}
$$