15-150 Assigment 2

Jonathan Li
jlli
Section S

May 28, 2016

Task 2.2

The work of `part`, $W_{\texttt{part}}(n)$, is $O(n)$, while $S_{\texttt{part}}(n)$ is also $O(n)$.

Task 4.1

Theorem 4.1: ∀ values `t : tree`, `size(t)` ≅ `length(treeToList t)`
Proof: By structural induction on `t`.
Case `Empty`: To show: `size(Empty)` ≅ `length(treeToList(Empty))`
Proof:

`size(Empty))`	≅	`0`	[Definition of `size`]
	≅	`length([])`	[Definition of `length`]
	≅	`length(treeToList(Empty))`	[Definition of `treeToList`]

By extensional equivalence, `size(Empty))` ≅ `length(treeToList(Empty))`.
Case `Node(l, x, r)` for some `l : tree`, `x : int`, `r : tree`.
Inductive Hypothesis: `size(l)` ≅ `length(treeToList(l))`, `size(r)` ≅ `length(treeToList(r))`.
To show: `size(Node(l, x, r))` ≅ `length(treeToList(Node(l, x, r)))`
Proof:

`size(Node(l, x, r))`	≅	`size(l) + 1 + size(r)`	[step, definition of `size`]
	≅	`length(treeToList(l)) + 1 + length(treeToList(r))`	[IH, Referential Transparency]
	≅	`length(treeToList(l)) + length(x::treeToList(r))`	[Lemma 2, Referential Transparency]
	≅	`length(treeToList(l) @ (x::treeToList(r))`	[Lemma 1, Symmetry]
	≅	`length(treeToList(Node(l, x, r)))`	[Definition of `treeToList`]

By extensional equivalence, `size(Node(l, x, r))` ≅ `length(treeToList(Node(l, x, r)))`.
Since the Base Case and the Inductive Step hold, the Theorem 4.1 must be true.

Task 6.4

<u>Work of `swapDown`</u>
Let $d =$ the depth of the input tree.

$$
\begin{aligned}
W_{\texttt{swapDown}}(0) &= k_0 && \text{[Base Case]} \\
W_{\texttt{swapDown}}(d) &= k_1 + W_{\texttt{treecompare}}(d, d-1) + W_{\texttt{treecompare}}(d, d-1) \\
&\quad + W_{\texttt{swapDown}}(d-1) + W_{\texttt{swapDown}}(d-1) && \text{[Two calls to \texttt{treecompare},} \\
& && \text{two recursive calls to \texttt{swapDown}]}
\end{aligned}
$$

To complete this recurrence, we will need to know the work of `treecompare`. However,

$$W_{\texttt{treecompare}}(a, b) \quad = \quad W_{\texttt{Int.compare}}(a, b)$$

Since `treecompare`, in the worst case, makes a single call to `Int.compare`. However, `Int.compare` is constant time, so `treecompare` must also be constant time. In other words, for this recurrence,

$$
\begin{aligned}
W_{\texttt{swapDown}}(d) &= k_2 + W_{\texttt{swapDown}}(d-1) + W_{\texttt{swapDown}}(d-1) && \text{[\texttt{treecompare} is constant time]} \\
&= k_2 + 2 \cdot W_{\texttt{swapDown}}(d-1) && \text{[math]}
\end{aligned}
$$

This gives us the recurrence relation for the work of `swapDown`. To find the closed-form and big-O estimate, we will expand out the recurrence.

$$
\begin{aligned}
W_{\texttt{swapDown}}(d) &= k_2 + 2 \cdot W_{\texttt{swapDown}}(d-1) \\
&= k_2 + 2(k_2 + 2(k_2 + 2(k_2 + \dots && \text{[expansion]} \\
&= k_2 + (2 \cdot k_2) + (4 \cdot k_2) + (8 \cdot k_2) + \cdots + (2^d \cdot k_2) && \text{[d recursive calls are made]}
\end{aligned}
$$

This gives us the closed form for the work of `swapDown`. The largest term in this equation is $2^d k_2$, so it will dominate.k_2 is a constant, so this means that $W_{\texttt{swapDown}}(d)$ is $O(2^d)$. Alternatively, since $d = log\ n$ for $n =$ number of nodes, $W_{\texttt{swapDown}}(n)$ is $O(2^{log n}) = O(n)$, or linear time.

Task 6.4 (cont.)

Span of `swapDown`
Again, let $d =$ the depth of the input tree.

$$\begin{aligned}
S_{\texttt{swapDown}}(0) &= k_0 && \text{[Base Case]} \\
S_{\texttt{swapDown}}(d) &= k_1 + \texttt{max}(S_{\texttt{treecompare}}(d, d-1), S_{\texttt{treecompare}}(d, d-1)) \\
&\quad + \texttt{max}(S_{\texttt{swapDown}}(d-1), S_{\texttt{swapDown}}(d-1)) && \text{[Two calls to } \texttt{treecompare}\text{, two calls to } \texttt{swapDown}\text{]} \\
&= k_1 + S_{\texttt{treecompare}}(d, d-1) + \texttt{max}(S_{\texttt{swapDown}}(d-1), S_{\texttt{swapDown}}(d-1)) && \text{[Each call to } \texttt{treecompare} \text{ has the same span]} \\
&= k_1 + S_{\texttt{treecompare}}(d, d-1) + S_{\texttt{swapDown}}(d-1) && \text{[Each call to } \texttt{swapDown} \text{ has the same span]}
\end{aligned}$$

Following the same logic as above, since the work of `treecompare` is constant time, the span of `treecompare` must also be constant time, so:

$$S_{\texttt{swapDown}}(d) = k_2 + S_{\texttt{swapDown}}(d-1) \qquad \text{[}\texttt{treecompare} \text{ is constant time]}$$

This gives us the recurrence relation for the span of `swapDown`. To find the closed-form and big-O estimate, we will expand out the recurrence.

$$\begin{aligned}
S_{\texttt{swapDown}}(d) &= k_2 + S_{\texttt{swapDown}}(d-1) \\
&= k_2 + (k_2 + (k_2 + (k_2 + \dots && \text{[expansion]} \\
&= d \cdot k_2 && \text{[}d\text{ recursive calls are made, associativity of addition]}
\end{aligned}$$

This gives us the closed form for the span of `swapDown`. Since k_2 is a constant, this means that $S_{\texttt{swapDown}}(d)$ is $O(d)$. Alternatively, since $d = log\ n$ for $n =$ number of nodes, $S_{\texttt{swapDown}}(n)$ is $O(log\ n)$, or logarithmic time.

Task 6.4 (cont.)

Work of `heapify`
Same as before. Let $d =$ the depth of the input tree.

$$\begin{aligned}
W_{\texttt{heapify}}(0) &= k_0 && \text{[Base Case]}\\
W_{\texttt{heapify}}(d) &= k_1 + W_{\texttt{heapify}}(d-1) + W_{\texttt{heapify}}(d-1) + W_{\texttt{swapDown}}(d) && \text{[Two recursive calls to } \texttt{heapify}, \\
&&& \text{one call to } \texttt{swapDown}]\\
&= k_1 + (2 \cdot W_{\texttt{heapify}}(d-1)) + W_{\texttt{swapDown}}(d) && \text{[math]}\\
&= k_1 + 2^d + (2 \cdot W_{\texttt{heapify}}(d-1)) && [W_{\texttt{swapDown}}(d) \text{ is } O(2^d),\\
&&& \text{commutativity of addition]}
\end{aligned}$$

This gives us a recurrence relation for the work of `heapify`.To find the closed-form and big-O estimate, we will expand out the recurrence.

$$\begin{aligned}
W_{\texttt{heapify}}(d) &= k_1 + 2^d + (2 \cdot W_{\texttt{heapify}}(d-1))\\
&= k_1 + 2^d + 2(k_1 + 2^{d-1} + 2(k_1 + 2^{d-2} + 2(k_1 + \dots && \text{[epxansion]}\\
&= k_1 + 2^d + (2 \cdot k_1) + (2 \cdot 2^{d-1}) + (2 \cdot 2 \cdot k_1) + (2 \cdot 2 \cdot 2^{d-2}) + \dots && \text{[math]}\\
&= (k_1 + 2^d) + ((2 \cdot k_1) + 2^d) + ((4 \cdot k_1) + 2^d) + \dots && \text{[math]}\\
&= (k_1 + (2 \cdot k_1) + (4 \cdot k_1) + \dots + (2^d \cdot k_1)) + (2^d + 2^d + 2^d + \dots) && \text{[Associativity of addition,}\\
&&& d \text{ recursive calls are made]}\\
&= (k_1 + (2 \cdot k_1) + (4 \cdot k_1) + \dots + (2^d \cdot k_1)) + (d \cdot 2^d) && [d \text{ recursive calls are made]}
\end{aligned}$$

This gives us the closed form for the work of `heapify`. The largest term in this expression is $d \cdot 2^d$, so $W_{\texttt{heapify}}(d)$ must be $O(d2^d)$, or $O(log\ n \cdot 2^{log\ n}) = O(log\ n \cdot n) = O(n \cdot log\ n)$.

Task 6.4 (cont.)

Span of `heapify`
You know the drill. Let $d =$ the depth of the input tree.

$$
\begin{aligned}
S_{\texttt{heapify}}(0) =&\quad k_0 && \text{[Base Case]}\\
S_{\texttt{heapify}}(d) =&\quad k_1 + \texttt{max}(S_{\texttt{heapify}}(d-1), S_{\texttt{heapify}}(d-1)) + S_{\texttt{swapDown}}(d) && \text{[Two recursive calls to } \texttt{heapify}\text{, one call to } \texttt{swapDown}\text{]}\\
=&\quad k_1 + S_{\texttt{heapify}}(d-1)) + S_{\texttt{swapDown}}(d) && \text{[Each call to } \texttt{heapify} \text{ has the same span]}\\
=&\quad k_1 + S_{\texttt{heapify}}(d-1) + d && [S_{\texttt{swapDown}}(d) \text{ is } O(d)]\\
=&\quad k_1 + d + S_{\texttt{heapify}}(d-1) && \text{[Commutativity of addition]}
\end{aligned}
$$

This gives us a recurrence relation for the span of `heapify`.To find the closed-form and big-O estimate, we will expand out the recurrence.

$$
\begin{aligned}
S_{\texttt{heapify}}(d) =&\quad k_1 + d + S_{\texttt{heapify}}(d-1)\\
=&\quad k_1 + d + (k_1 + d + (k_1 + d + (k_1 + d + \ldots && \text{[expansion]}\\
=&\quad (d \cdot k_1) + (d \cdot d) && [d \text{ recursive calls are made, associativity of addition]}\\
=&\quad (d \cdot k_1) + d^2 && \text{[math]}
\end{aligned}
$$

This gives us the closed form for the span of `heapify`. The largest term in this expression is d^2, so $S_{\texttt{heapify}}(d)$ must be $O(d^2)$, or $O((log\ n)^2)$.