A (Parallel) Algorithm

```c
int findMin(Node current) {
    if (current is a leaf) {
        return current.data;
    }

    return min(current.data, findMin(left), findMin(right));
}
```

To calculate work, we just do our standard analysis. First, we make a recurrence:

\[
T(1) = 1 \\
T(n) = T(0) + T(n-1) + 1
\]

\[\Theta(n) \quad \checkmark\]
A (Parallel) Algorithm

```c
int findMin(Node current) {
  if (current is a leaf) {
    return current.data;
  }
  return min(current.data, findMin(left), findMin(right));
}
```

To calculate span, we assume all calls are in parallel. We look for the longest dependence chain. We make a recurrence:

\[
\text{span}(n) = \begin{cases} 
  1 & \text{if } n = 0 \\
  O(1) & \text{if } n = 1 \\
  \max(\text{span}(0), \text{span}(n-1)) + 1 & \text{otherwise}
\end{cases}
\]

This ends up being the same recurrence as for work. Notice for the degenerate tree work \( n \) = \( \text{span}(n) \). This proves our intuition that we don’t get much of a (any!) speed-up with parallelism for linked lists!
A (Parallel) Algorithm

```java
int findMin(Node current) {
    if (current is a leaf) {
        return current.data;
    }
    return min(current.data, findMin(left), findMin(right));
}
```

To calculate work, we just do our standard analysis. First, we make a recurrence:

\[
T(n) = 2T(n/2) + 1
\]
A (Parallel) Algorithm

```c
int findMin(Node current) {
    if (current is a leaf) {
        return current.data;
    }

    return min(current.data, findMin(left), findMin(right));
}
```

To calculate span, we take the \textbf{max} of the recursive calls. First, we make a recurrence:

\[
    \text{span}(n) = \max(\text{span}(n/2), \text{span}(n/2)) + 1
\]