CSE 331: Software Design & Implementation

Homework 4 (due Wednesday, April 26th at 11:00 PM)

The problems in this assignment contain a mix of written and coding parts. When you have completed them, submit your solutions in Gradescope. The written problems should be clearly labeled and submitted as a pdf to the "HW4 Written" assignment. The following completed files should be directly submitted for the coding portion to the "HW4 Coding" assignment:

```
color_list.ts          color_tree_test.ts     ui_test.tsx
color_list_test.ts     index.tsx              list_test.ts
color_tree.ts          ui.tsx                 parser.ts
```

For complete instructions for how to submit assignments in this course see the Homework Turn-in Guide.

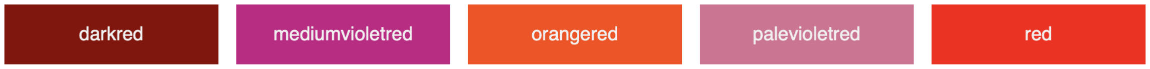
Reminder: Mutation is *not* allowed. Only write straight-line code, conditionals, and recursion. This applies to *all* code written for this assignment, including tests.

Start by checking out the starter code using the command

```
git clone https://gitlab.cs.washington.edu/cse331-23sp-materials/hw-highlight.git
```

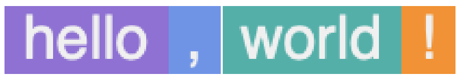
Then, install the modules using `npm install --no-audit`.

Try out the application by running `npm run start`. The app contains a text area which allows the user to search for colors whose names contain a given sub-string. For example, searching for "Red" will show "DarkRed", "PaleVioletRed", etc.



The app contains another text area which allows users to draw text with different colored highlighting. The text area expects each line of input to start with the name of a color followed by the text to be highlighted in that color. For example, the input on the left produces the output on the right:

MediumPurple hello
CornflowerBlue ,
LightSeaGreen world
DarkOrange !



All of the color possibilities are listed in `colors.ts`. More details on the contents of this file and others are to follow, but browse the starter code briefly to see what we're working with.

1. List and Shout (20 points)

The following parts consist entirely of coding work. It should be submitted with "HW4 Coding".

The application described on the page prior records information about an individual color in a triple, called `ColorInfo`, of the form (n, c, w), where n is the name of the color, c is the CSS description of the color, and w is a boolean indicating whether text with that background color should be white rather than black. The full list of available colors is stored in a `List<ColorInfo>` called `COLORS` in `colors.ts`.

The file `color_list.ts` exports two functions, `findMatchingNames` and `getColorCss`, that search for information on colors. Currently, both operations work directly with the list `COLORS`. In this problem, we will replace this with an ADT so that we can easily change the representation used during searches later on.

(a) Create a new interface `ColorList` in `color_list.ts` that has just the two operations mentioned above. Be sure that the interface and its functions have proper documentation.

(b) Change the code in `ui.tsx` to import and use only the interface.

Specifically, change the two functions, `showColors` and `showHighlights`, to take a `ColorList` as part of their `props` argument and then use that `ColorList` to perform the search operations. (You will need to change the helper functions of each of those functions as well.)

(c) Create a class in `color_list.ts` called `SimpleColorList` that implements the `ColorList` interface. It should take a `List<ColorInfo>` to operate on and store it in a field and implement `findingMatchingNames` and `getColorCss` (in the same way as the current implementations of those functions but now by passing in the colors field rather than the `COLORS` constant). After adding these operations to the new class, you can remove the old implementations of `findingMatchingNames` and `getColorCss` in the file.

Be sure to properly document the class and add and abstraction function of the class.

(d) Create a function, called `makeSimpleColorList`, that returns an instance of `SimpleColorList` that uses colors from the `COLORS` list. Be sure to write a complete specification for the function.

Make sure that `ColorList` and `makeSimpleColorList` are the only symbols exported in the file. Clients should not be able to directly access the class, or anything else.

(e) Change the tests in `color_list_test.ts` and `ui_test.tsx` to use `makeSimpleColorList` to create one instance of `ColorList` and use that instance to perform the operations throughout the tests.

Make sure that all the tests still pass by running `npm run test`.

(f) Change the code in `index.tsx` to use `makeSimpleColorList` to create an instance of `ColorList` and pass that to the functions it calls from `ui.tsx`.

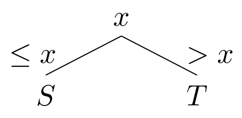
Make sure that the app still works by running `npm run start` and trying it out.

BSTs

The next problem makes use of the following inductive type, representing a binary search tree

$$\begin{array}{rll}
\textbf{type}\ \mathsf{BST} := & \mathsf{empty} & \\
\mid & \mathsf{node}(x:\mathbb{Z},\ S:\mathsf{BST},\ T:\mathsf{BST}) & \text{with conditions A and B}
\end{array}$$

where "A" is the condition that, for every $y : \mathbb{Z}$ with $y > x$, the value y cannot appear in any node of S, and "B" is the condition that, for every $y : \mathbb{Z}$ with $y \leq x$, the value x cannot appear in any node of T. (We will define "appear in" more precisely below, but hopefully this is clear: numbers in the nodes of S must be less than or equal to x and numbers in the nodes of T must be greater than x.)

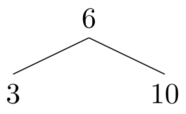


Conditions A and B are *invariants* of the BST node. Every node that is created must have these properties, and we are allowed to use the fact that they hold anywhere in our reasoning.

We can define the list of numbers appearing in the tree as follows:

$$\begin{array}{lll}
\textbf{func}\ \mathsf{values}(\mathsf{empty}) & := & \mathsf{nil} \\
\quad \mathsf{values}(\mathsf{node}(x, S, T)) & := & \mathsf{concat}(\mathsf{values}(S), \mathsf{cons}(x, \mathsf{values}(T))) \quad \text{for any } x : \mathbb{Z} \text{ and } S, T : \mathsf{BST}
\end{array}$$

In words, the list of numbers in $\mathsf{node}(x, S, T)$ is the list of numbers appearing in S followed by x followed by the list of numbers appearing in T. (This is the natural way to order the list because it is then sorted.)
For example:
`B: BST` = node(x: 6, S: node(x: 3, S: empty, T: empty), T: node(x: 10, S: empty, T: empty))



`values(B)` = cons(3, cons(6, cons(10, nil)))

Lists

We will also need the prefix and suffix functions from HW3, but we will combine them into a single function, called split, which is defined as follows:

$$\begin{array}{llll}
\textbf{func}\ \mathsf{split}(0, L) & := & (\mathsf{nil}, L) & \text{for any } L : \mathsf{List} \\
\quad \mathsf{split}(m+1, \mathsf{nil}) & := & \mathsf{undefined} & \text{for any } m : \mathbb{N} \\
\quad \mathsf{split}(m+1, \mathsf{cons}(a, L)) & := & (\mathsf{cons}(a, P), S) & \text{for any } m : \mathbb{N},\ a : \mathbb{Z}, \text{ and } L : \mathsf{List} \\
 & & \quad \text{where } (P, S) := \mathsf{split}(m, L) &
\end{array}$$

In the third case, to split a non-empty list $\mathsf{cons}(a, L)$ into its first $m + 1$ elements and the rest, we recursively split L into its first m elements and the rest and then prepend a to the prefix, giving it length $m + 1$.

It is not hard to prove, using techniques we saw in HW3, that, if $m \leq \mathsf{len}(L)$, then $\mathsf{split}(m, L)$ returns a pair (P, S) with the properties that $L = \mathsf{concat}(P, S)$, $\mathsf{len}(P) = m$, and $\mathsf{len}(S) = \mathsf{len}(L) - m$. We will need those facts in the next problem. (You can call this "Lemma 1".)

2. Not My Cup of Tree (20 points)

The following parts consist entirely of written work. It should be submitted with "HW4 Written".

The function "values", defined on the previous page, turns a BST into a List, but we will also need a way to turn a List into a BST. Here is one way to do so:

```
func makeBst(nil)          := empty
     makeBst(cons($a, L$))  := node($b$, makeBst($P$), makeBst($R$))
                                  where $(P, S)$ = split($m$, cons($a, L$)), $m = \lfloor$len(cons($a, L$))$/2\rfloor$
                                    and $S$ = cons($b, R$)
```

This definition turns a non-empty list L into a tree by calling split(m, L) to split L into (P, S), each containing approximately half of L. Then, we further split S into its first element, b, and the rest, R. Finally, we return the tree with b at its root and P and R, each made recursively into trees, as its left and right subtree.

Note that, in order for node$(b,$ makeBst(P), makeBst$(R))$ to satisfy the BST invariant, it must be the case that everything in P is less than or equal to b and everything in R is greater than b. That will always be the case if we start with a list that is **sorted**: the input list $L =$ concat$(P,$ cons$(b, R))$: is only sorted if everything in P is smaller than b and everything in R is larger than b.

We said "approximately half" above because, if len(L) is odd, then it cannot be split exactly in half. The definition above says to split after $m = \lfloor$len$(L)/2\rfloor$ elements, where the notation $\lfloor\cdot\rfloor$ means to round *down* to the closest integer. In particular, when len(L) is odd, m will be (len$(L) - 1)/2$.

To prove that makeBst is correct, we want to show that it produces a tree containing the same elements as in the list it was given, i.e., that values(makeBst$(U)) = U$ holds for any list U.

This is a little trickier than it initially appears, however, because makeBst(cons(a, L)) does not make a recursive call to makeBst(L). Instead, it makes recursive calls to makeBst(P) and makeBst(R), and all we know about the lists P and R is that they are shorter than cons(a, L). In order to apply the inductive hypothesis to P and R, we will need to phrase the claim a little differently.[1]

We will prove by induction on $n : \mathbb{N}$ that values(makeBst$(U)) = U$ holds for any list U with len$(U) \le n$. This means the equation holds for any list U since we can apply the theorem with $n =$ len(U), as we know that the condition len$(U) \le n =$ len(U) certainly holds.

(a) The base case, $P(0)$, says that values(makeBst$(U)) = U$ for any list U with len$(U) = 0$. The only list satisfying this claim is nil (i.e., len$(U) = 0$ implies that $U =$ nil).

Prove by calculation that values(makeBst(nil)) = nil.

(b) The inductive step, $P(n+1)$, says that values(makeBst$(U)) = U$ for any list U with len$(U) \le n + 1$. Note that len$(U) = n + 1 \ge 1$, in which case, $U =$ cons(a, L) for some some $a : \mathbb{Z}$ and L : List, and it is only necessary to prove the claim for a U of this form.

Prove by calculation that values(makeBst(cons(a, L))) = cons(a, L). Be sure to use the inductive hypothesis which tells us that values(makeBst$(U)) = U$ for **any** list U with len$(U) \le n$.

That completes the induction argument. We have shown that values(makeBst$(U)) = U$ holds for any list U.

(c) **Extra Credit**: The definition of makeBst only makes sense if S (the second half of the list) is non-empty. Otherwise, there would be no element "b" at the front of S to put at the root of the tree!

Use the properties of split mentioned on the previous page and the fact that len$(L) \ge 1$ to prove that len$(S) \ge 1$, where $(P, S) =$ split(m, L) and $m = \lfloor$len$(L)/2\rfloor$. That means S has a first element.

[1]This approach is sometimes called "strong" induction.

3. You Haven't Heard the Last of Tree (20 points)

The following parts consist entirely of coding work. It should be submitted with "HW4 Coding".

In an effort to make lookups more efficient, we decide to make a tree implementation of `ColorList`. Some portions have already been created or defined, so it's your job to put it all together.

(a) Translate the mathematical definition of makeBst from question 2 into a TypeScript function in `color_tree.ts` with the following signature:

```
export function makeBst(L: List<ColorInfo>): ColorNode
```

`ColorNode` is a BST type for colors (rather than integers) provided in `color_node.ts`. `node` is a function to create a `ColorNode` provided in the same file. The function `split` is provided in `list.ts`.

Be sure to write a specification for the function.

(b) Write test cases for makeBst in the file `color_tree_test.ts`. There is one example test given. Follow the rules taught in lecture for choosing appropriate test cases.

Include brief comments in your test suite justifying the test cases you chose.

Confirm that all your tests pass by running `npm run test`.

Next, we consider the following function, which looks up a value in a binary tree:

func lookup(y, empty)	:=	false		for any $y : \mathbb{Z}$
lookup$(y, \text{node}(x, S, T))$	:=	true	if $x = y$	for any $x, y : \mathbb{Z}$ and S, T : BST
lookup$(y, \text{node}(x, S, T))$	:=	lookup(y, T)	if $x < y$	for any $x, y : \mathbb{Z}$ and S, T : BST
lookup$(y, \text{node}(x, S, T))$	:=	lookup(y, S)	if $y < x$	for any $x, y : \mathbb{Z}$ and S, T : BST

(c) Translate the definition of lookup into a TypeScript function in the file `color_tree.ts` with the following signature:

```
export function lookup(y: string, root: ColorNode): ColorNode
```

Instead of returning a boolean like in the mathematical definition, if a node is found containing that color name, return that `ColorNode`, and if none is found, return `empty`.

Be sure to write a specification for the function. Write your formal specification with tags such as `@returns`, rather than rewriting the above formal definition (which is more information than the client wants or needs).

(d) Write test cases for lookup in the file `color_tree_test.ts`. Follow the rules taught in lecture for choosing appropriate test cases.

There is one example test given, however in the originally released starter code there was a typo in the test. The first parameter to the test should be: `lookup('Yellow', node(['Yellow', '#FFFF00', false], empty, empty)).info` –the test will not pass without **.info** at the end, so add it if it is missing.

The return value of lookup can evaluate to empty, which .info would not work on, so you will likely get a warning. The warning can be ignored if you are sure lookup will not return empty for that test case.

Include brief comments in your test suite justifying the test cases you chose.

Confirm that all your tests pass by running `npm run test`.

(e) We have everything we need to start using the tree to lookup color names. In order to use it in the rest of the code, however, we need to implement the `ColorList` interface that we created earlier.

Create a class in `color_tree.ts` called `ColorTree` that implements the `ColorList` interface. It should take a `List<ColorInfo>` and store it in a field as well as store the same data in tree form as created by calling `makeBst`.

Be sure to properly document the class, which should now have both an AF and RI.

There is no efficiency benefit to using the tree to implement `findMatchingNames`, so instead, you should call the existing function `findMatchingNamesIn` from `color_list.ts` (which you must now edit to export), passing it the list of colors instead of the tree.

Implement the function `getColorCss` by calling `lookup` on the tree. (See the `getColorCssIn` function in `color_list.ts` for an example of what to return if the color is found and what to throw if it is not found). Also, note that you may need to explicitly include the return type on the function you write, as shown here, in order to please the type checker:

```
getColorCss = (name: string): readonly [string, string] => { ... }
```

(f) Create a factory function, called `makeColorTree`, that returns an instance of `ColorTree`, passing it the colors from the `COLORS` list. Be sure to write a specification for the function.

(g) We also need tests for the new code. However, since we are using the same interface as before, we can simply copy the existing tests for `findMatchingNames` and `getColorCss` from `color_list_test.ts` into `color_tree_test.ts`. The only change that should be needed is to call the new factory function, `makeColorTree`, instead of the old one.

Confirm that all the tests pass by running `npm run test`.

(h) Change the code in `index.tsx` to use `makeColorTree` instead of `makeSimpleColorList`.

Make sure that the app still works by running `npm run start` and trying it out.

More Lists

Earlier, we defined the list of numbers that appear in a tree. We also talked about whether a number appears in a list, but we never defined that precisely. We can do so as follows:

$$\begin{array}{lll} \textbf{func}\ \mathsf{contains}(a, \mathsf{nil}) & := \mathsf{false} & \text{for any } a : \mathbb{Z} \\ \quad\ \mathsf{contains}(a, \mathsf{cons}(b, L)) & := (a = b) \text{ or } \mathsf{contains}(a, L) & \text{for any } a, b : \mathbb{Z} \text{ and } L : \mathsf{List} \end{array}$$

More BSTs

Earlier, we gave an informal description of what must be true in order to create node(x, S, T): every value in S must be less than or equal to x and every value in T must be greater than x. Formally, the BST invariant requires that the following must hold for any integer y:

- if contains$(y,$ values$(S))$ = true, then $y \le x$, and
- if contains$(y,$ values$(T))$ = true, then $y > x$.

An equivalent way to say this is that, if $y > x$, then y *cannot* appear in S, and if $y \le x$, then y *cannot* appear in T. Formally, then, the BST invariant equivalently says that the following must hold for any integer y:

- if $y > x$, then contains$(y,$ values$(S))$ = false, and
- if $y \le x$, then contains$(y,$ values$(T))$ = false.

(Students who have taken 311 should recognize these statements as the contrapositives of those above.)

Another application of contains is that it will allow us to formally state and prove the correctness of lookup, which we will do in the next problem.

4. Many More Fish in the Tree (20 points)

The following parts consist entirely of written work. It should be submitted with "HW4 Written".

In this problem, we will prove the correctness of the BST lookup function we defined earlier. To get there, we first need to prove another fact.

(a) Let a be any integer and S be any list. Prove by induction on L that

$$\mathsf{contains}(a, \mathsf{concat}(L, S)) = \mathsf{contains}(a, L) \text{ or } \mathsf{contains}(a, S)$$

Hint: It may be useful to calculate both left-to-right and right-to-left until you get values that are equal.

(b) Let a be any integer. Prove by induction that, for any U : BST, we must have

$$\mathsf{contains}(a, \mathsf{values}(U)) = \mathsf{lookup}(a, U)$$

In other words, lookup always returns the correct answer.

Hints for the inductive step:

- Start by proving, by calculation, and using part (a), that

$$\mathsf{contains}(a, \mathsf{values}(\mathsf{node}(b, S, T))) = \mathsf{contains}(a, \mathsf{values}(S)) \text{ or } (a = b) \text{ or } \mathsf{contains}(a, \mathsf{values}(T))$$

- Then, continue the argument by cases over whether $a = b$, $a < b$, or $a > b$. In each case, you should be able to prove that the right-hand side is lookup$(a,$ node$(b, S, T))$. In the case $a < b$, for example, you should be able to show that the right-hand side simplifies to contains$(a,$ values$(S))$, using the BST invariant, and then show that the simplified expression equals lookup$(a,$ node$(b, S, T))$.

5. Chomping at the Split (20 points)

The following parts consist of a mix of written and coding work: part (b) is written and should be submitted with "HW4 Written", while parts (a,c,d,e,f) are coding and should be submitted with "HW4 Coding".

In this problem we will change the format in which users write the text that they want highlighted. Instead of asking them to split the text into lines, with each line having text in a single color, we will allow them to write all the text in one line and indicate where they want highlighting using the syntax "[color|text]", where "color" is the name of the color and "text" is everything they want shown in that color. All text not written inside "[..]" will be written without a highlight (i.e., black text on a white background).

In order to do that, we will first need a helper function, split-at, that takes a list L and a value c as arguments and "returns a pair of lists (P, S), with $L = \text{concat}(P, S)$, where P contains all the values before the first c and S is either empty or starts with a c". For example, the calling split-at with inputs $[1, 2, 3, 4, 5]$ and 3 would return $([1, 2], [3, 4, 5])$.

The following is a formal definition of split-at using recursion:

```
func split-at(nil, $c$)              :=  nil                                     for any $c : \mathbb{Z}$
     split-at(cons($a, R$), $c$)     :=  (nil, cons($a, R$))     if $a = c$      for any $a, c : \mathbb{Z}$ and $R$ : List
     split-at(cons($a, R$), $c$)     :=  (cons($a, P$), $S$)     if $a \neq c$   for any $a, c : \mathbb{Z}$ and $R$ : List
                                            where $(P, S) = \text{split-at}(R, c)$
```

A translation of this function in to Typescript is given in the `list.ts` file with the following signature:

```
export function split_at<A>(L: List<A>, c: A): readonly [List<A>, List<A>]
```

(a) Write test cases for `split_at` in `list_test.ts`. Follow the rules taught in lecture for choosing appropriate test cases.

Include brief comments in your test suite justifying the test cases you chose.

Confirm that all your tests pass by running `npm run test`.

Next, we will write a function that parses the text in the format described above. It should take a list of characters as an argument and return a list of pairs, each of the form (c, T), where c is a color name and T is a list of characters (text) in that color. If the user includes "[" that is not followed by "|" or both of those that are not followed by "]", then the text is not in the proper format and we will return undefined.

(b) Above, we were given an English definition of the problem, so our first step is to formalize it.

Write a formal definition of parse using calls to split-at and recursion. Use the constants "LB" , "MB", and "RB" to refer to the character codes of "[", "|", and "]", respectively.

(Reminder: doing this on paper — trying to get all the cases right without testing it out on a computer — is good practice for interviews, where you will not have computer help.)

(c) Translate your definition into a TypeScript function in `parser.ts` with the following signature:

```
function findHighlights(chars: List<number>): List<Highlight>
```

Note that, in the case of incorrect user input ('[' with no closing ']', missing '|', etc.), rather than returning undefined, we want to make the user experience good by *not* throwing errors which could confuse the user. Instead, in these cases, accept the text and print it out with a "white" background (no highlight). This behavior is expected even if they partially define the highlight: "hello [red|there" should return "hello [red|there" exactly with a white background. There are some tests that cover these cases; in step (e), when you run them, they should pass.

Also, rather that returning pairs, the function should return a list of `Highlight` (a record type defined in the code). The "`text`" field of the `Highlight` is a `string` rather than a list of characters, so you will need to call `compact` to convert each list into a string.

(d) `findHighlights` should not be exported. Instead, create an exported function with the following signature:

```
export function parseHighlightText(text: string): List<Highlight>
```

which calls `findHighlights` with `text` converted into a list of characters by calling `explode`.

Be sure to write a specification for `parseHighlightText`, but this time, let's prefer a precise, English description of what is returned rather than spelling out the mathematical details in your definition. (The client shouldn't need to follow those details, which are pretty complicated.)

(e) There are tests given for this function in `parser_test.ts`. Confirm that your code works as expected and the tests pass by uncommenting them and running `npm run test`.

(f) Change the code in `index.tsx` to use `parseHighlightText` instead of `parseHighlightLines`.

Make sure that the app now supports the new syntax by running `npm run start` and trying it out.

Congratulations! You've made the app both faster and more useful. Great work!

6. Extra Credit: Live Tree or Die (0 points)

The following parts consist of entirely written work. It should be submitted with "HW4 Written".
On the previous page, we claimed that we have made the app faster by replacing the list search with a BST search, but we haven't actually proven that. We will do that in this extra credit problem.

In order to do so, we need the following definition:

$$\begin{aligned} &\textbf{func}\ \mathsf{height}(\mathsf{empty}) && := && -1 \\ &\phantom{\textbf{func}}\ \mathsf{height}(\mathsf{node}(x, S, T)) && := && 1 + \max(\mathsf{height}(S), \mathsf{height}(T)) \quad \text{for any } x : \mathbb{Z} \text{ and } S, T : \mathsf{BST} \end{aligned}$$

With that in hand, we can now describe the running time of lookup and show that it is faster than linear.

(a) Let L be any list. Show that $\mathsf{len}(P) \le \mathsf{len}(S)$, where $(P, S) = \mathsf{split}(m, L)$ and $m = \lfloor \mathsf{len}(L)/2 \rfloor$.

(b) Suppose that $\mathsf{len}(L) \le 2^{k+1} - 1$ for some integer k. Show that we then have $\mathsf{len}(P) \le 2^k - 1$, where P is defined as in part (a).(You may need to use part (a) to show this.)

(c) Let L be any non-empty list, and let a any integer. Show that we have $\mathsf{len}(R) \le \mathsf{len}(P)$, where $(P, S) = \mathsf{split}(m, \mathsf{cons}(a, L))$, $m = \lfloor \mathsf{len}(\mathsf{cons}(a, L))/2 \rfloor$, and $S = \mathsf{cons}(b, R)$.

It may be easiest to argue by cases based on whether $n = \mathsf{len}(\mathsf{cons}(a, L))$ is even or odd.

(d) Let $a, b, c : \mathbb{Z}$ with $a \le c$ and $b \le c$. Prove by cases that $\max(a, b) \le c$.

(e) Prove by induction that, for any list L, the inequality $\mathsf{height}(\mathsf{makeBst}(L)) \le k$ holds for any integer k satisfying $\mathsf{len}(L) \le 2^{k+1} - 1$.

You may need to use parts (b-d).

(f) Explain how to prove that, for any non-empty tree T and integer y, the number of recursive calls made by $\mathsf{lookup}(y, T)$ before it returns (not just the first recursive call but any recursive calls that makes and so on) is no more than $\mathsf{height}(T)$. (You do not need to write out the proof in full detail.)

(g) Let L be a list and $n := \mathsf{len}(L)$. The integer $k := \lceil \log_2(n+1) \rceil - 1$, where $\lceil \cdot \rceil$ means rounding up to the closest larger integer, satisfies $n \le 2^{k+1} - 1$.

Explain why we know that $\mathsf{lookup}(y, T)$, for any integer y, runs in $O(\log n)$ time. (This is exponentially faster than linear search, which runs in $\Theta(n)$ time.)