CSE 331: Software Design & Implementation

Homework 5 (due Wednesday, May 1st 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 "HW5 Written" assignment. The following completed files should be directly submitted for the coding portion to the "HW5 Coding" assignment:

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

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

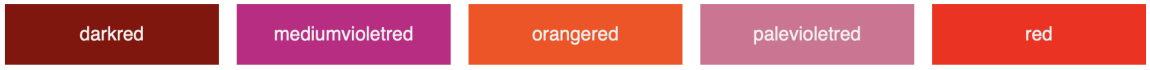
Note: Starting with this assignment, you must write JSDoc specification. We included reminders in the instructions for you, and it is highly reccomended that you write your specifications as you write your code (rather than waiting until the end) so the function behavior is fresh in your mind.

Start by checking out the starter code using the command

```
git clone https://gitlab.cs.washington.edu/cse331-24sp/cse331-24sp-materials/hw-highlight.git
```

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

Try out the application by running `npm run start` and navigating to `localhost:8080`. 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 "HW5 Coding".

The application described on the page prior records information about an individual color in a triple, called `ColorDetails`, 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). This type and the full list of available colors, stored in a `List<ColorDetails>` called `COLORS`, in `colors.ts`.

The file `color_list.ts` exports two functions, `findNameSet` 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. Include proper documentation for the interface and its functions.

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

Specifically, change the two functions, `ShowColorSet` and `ShowHighlights`, to take a `ColorList` called `"colors"` 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. Its constructor should take a `List<ColorDetails>` as an argument and store it in a field.

The class should then implement the methods `findNameSet` and `getColorCss` in the *same way* as the existing functions with those same names except that the methods will use their `List<ColorDetails>` field rather than the `COLORS` constant. After adding these methods to the new class, you can remove the original `findNameSet` and `getColorCss` functions from the file. (However, the helper functions `findNameSetIn` and `getColorCssIn` can stay as they are and can be used in this step.)

Include proper documentation for the class and add an abstraction function of the class (feel free to copy from existing documentation in the file where it applies).

(d) Create a function, called `makeSimpleColorList`, that returns an instance of `SimpleColorList` that uses colors from the `COLORS` list. Write a JSDoc specification for the function.

All instances of `SimpleColorList` created with the same argument (`COLORS`) will behave identically on all operations, so there is no need to create multiple instances. To avoid creating multiple instances, we can use the "singleton" pattern: define a constant in the file that stores a single instance of the class and have `makeSimpleColorList` always return that instance.

Make sure that `ColorList` and `makeSimpleColorList` are the only symbols exported in the file. Clients should not be able to directly access the class, the singleton instance, 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`. (Note that `npm run start` will still not work until we finish the next part.)

(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`.

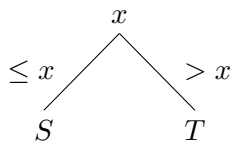
Make sure that the app still works by running `npm run start` and trying it out.

BSTs

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

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

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 y 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.)

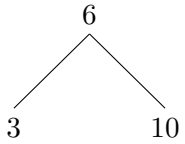


Conditions A and B are *invariants* of the BST node. Every node that is created must have these properties, and we are allowed to use the fact that they hold anywhere in our reasoning.

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

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

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))



`toList(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:

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

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 $m+1$.

For the next problem, you will need the following fact about this function, which you can cite as "**Lemma 1**":

If $m \leq \mathsf{len}(U)$, then $\mathsf{split}(m, U)$ returns a pair (P, S) with the properties that $U = \mathsf{concat}(P, S)$, $\mathsf{len}(P) = m$, and $\mathsf{len}(S) = \mathsf{len}(U) - m$.

This lemma can be proved by induction on m.

2. Not My Cup of Tree (20 points)

The following parts consist entirely of written work. It should be submitted with "HW5 Written".

The function "toList", 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 buildBst(nil) := empty

buildBst(cons(a, L)) := node$(b,$ buildBst$(P),$ buildBst$(R))$ for any $a : \mathbb{Z}$ and L :List

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 list, cons(a, L), into a tree by calling split$(m,$ cons$(a, L))$ to split the list into (P, S), each containing approximately half of cons(a, L). Then, we further take S and call 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$(b,$ buildBst$(P),$ buildBst$(R))$ 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 cons$(a, L) =$ 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(cons(a, L)) is odd, then it cannot be split exactly in half. The definition above says to split after $m = \lfloor$len(cons$(a, L))/2\rfloor$ elements, where the notation $\lfloor\cdot\rfloor$ means to round *down* to the closest integer. In particular, when len(cons(a, L)) is odd, m will be (len(cons$(a, L)) - 1)/2$.

To prove that buildBst 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 toList(buildBst$(U)) = U$ holds for any list U.

This is a little trickier than it initially appears, however, because buildBst(cons(a, L)) does not make a recursive call to buildBst(L). Instead, it makes recursive calls to buildBst(P) and buildBst(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 toList(buildBst$(U)) = U$ holds for any list U with len$(U) \leq n$. 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$(U) \leq n =$ len(U) certainly holds.

(a) The base case, $P(0)$, says that toList(buildBst$(U)) = U$ 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 toList(buildBst(nil)) = nil.

(b) The inductive step, $P(n+1)$, says that toList(buildBst$(U)) = U$ for any list U with len$(U) \leq n+1$. Note that len$(U) = n + 1 \geq 1$, 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 toList(buildBst(cons$(a, L))) =$ cons(a, L). Be sure to use the inductive hypothesis which tells us that toList(buildBst$(U)) = U$ for **any** list U with len$(U) \leq n$.

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

(c) **Extra Credit**: The definition of buildBst 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$(L) \geq 1$ to prove that len$(S) \geq 1$, 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 "HW5 Coding".

In an effort to make searchs 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 buildBst from question 2 into a TypeScript function in `color_tree.ts` with the following signature:

```
export const buildBst = (L: List<ColorDetails>): 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`. Note that colors are sorted by their string names, so there's no need to implement any additional ordering, you can use default string ordering with standard operations $>, <, =$.

Write a JSDoc specification for the function.

(b) Write test cases for buildBst 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 finds a node in a binary tree with the given root value:

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

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

```
export const search = (y: string, root: ColorNode): ColorDetails | undefined => {..};
```

Rather than implementing search with numbers, as given in the math definition, we will look up colors in a `ColorNode` BST. When we compare the value `y`, a color, to the current `ColorNode`, we just want to compare the color in the `ColorNode`'s `details`. Then, once we find the color we're looking for, we'll return the `ColorDetails` for that node (which is the value at the root of that node rather than the whole node including its sub-trees). Other than this difference, your implementation should be a direct translation of this math definition.

Write a JSDoc specification for the function. Sometimes we include the formal math definition in the spec, but for this problem just use english in your tags (such as `@returns`) because the formal definition is more information than the client needs and differs some from our implementation.

(d) Write test cases for `search` 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`.

(e) We have everything we need to start using the tree to search 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<ColorDetails>` and store it in a field as well as store the same data in tree form as created by calling `buildBst`.

Properly document the class, which should include both an AF and RI.

There is no efficiency benefit to using the tree to implement `findNameSet`, so instead, you should call the existing function `findNameSetIn` 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 `search` 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. As before, you should use the singleton pattern to avoid creating multiple instances. Write a JSDoc 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 `findNameSet` 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:

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

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$(y, \text{toList}(S)) = \text{true}$, then $y \leq x$, and
- if contains$(y, \text{toList}(T)) = \text{true}$, then $y > x$.

An equivalent way to say this is that, if $y > x$, then y *cannot* appear in S, and if $y \leq x$, 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$(y, \text{toList}(S)) = \text{false}$, and
- if $y \leq x$, then contains$(y, \text{toList}(T)) = \text{false}$.

(These statements as the contrapositives of those above. See 311 for more!)

Another application of contains is that it will allow us to formally state and prove the correctness of search, 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 "HW5 Written".

In this problem, we will prove the correctness of the BST search 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

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

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

$$\text{contains}(a, \text{toList}(U)) = (\text{search}(a, U) \neq \text{undefined})$$

In other words, search returns undefined when $\text{contains}(a, \text{toList}(U)) = \text{false}$, and it returns a node in the tree when $\text{contains}(a, \text{toList}(U)) = \text{true}$.

Hints for the inductive step:

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

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

- 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 $\text{search}(a, \text{node}(b, S, T))$. In the case $a < b$, for example, you should be able to show that the right-hand side simplifies to $\text{contains}(a, \text{toList}(S))$, using the BST invariant, and then show that the simplified expression equals $\text{search}(a, \text{node}(b, S, T))$.

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 "HW5 Written", while parts (a,c,d,e,f) are coding and should be submitted with "HW5 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). For example:

[MediumPurple|hello] [CornflowerBlue|,] [LightSeaGreen|world] [DarkOrange|!]

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 contains the remainder of the values starting with the first c". For example, the calling split-at with inputs $[2, 8, 1, 4, 1]$ and 1 would return $([2, 8], [1, 4, 1])$.

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

$$
\begin{array}{lcll}
\textbf{func}\ \text{split-at}(\text{nil}, c) & := & (\text{nil}, \text{nil}) & \text{for any } c : \mathbb{Z} \\
\quad \text{split-at}(\text{cons}(a, R), c) & := & (\text{nil}, \text{cons}(a, R)) \quad \text{if } a = c & \text{for any } a, c : \mathbb{Z} \text{ and } R : \text{List} \\
\quad \text{split-at}(\text{cons}(a, R), c) & := & (\text{cons}(a, P), S) \quad \text{if } a \neq c & \text{for any } a, c : \mathbb{Z} \text{ and } R : \text{List} \\
 & & \text{where } (P, S) = \text{split-at}(R, c) &
\end{array}
$$

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

```
export const 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`.

We have provided you with another helper function `parseNextHighlight` that finds the next highlighted portion of the text, if one exists. It returns either a triple (b, h, a), where b is the text before the next highlight (as a *string*), h is the highlight that was found (a record containing the color name and text), and a is the text after the next highlight (as a *List* of `number` character codes), or undefined if there is no next highlighted portion.

Note that this function parses the *first* highlight found in the text, but not anything after that. In other words, when it returns a triple (b, h, a), while the string b will not contain any text of the form "[color|text]", the list a very well might, so more work is necessary to fully parse the text.

Our goal is to write a function that parses a list of characters into a list of highlight records that describes all the text using the helper function just described. This includes not only the portions of the form "[color|text]", where the user indicated a specific color, but also all the regular text, which should be turned into records with background color "white". For example, the text "my [red|favorite] book" would become the list

$$
\begin{array}{l}
\text{cons}(\{\text{color}: \text{“white”},\ \text{text}: \text{“my ”}\}, \\
\quad \text{cons}(\{\text{color}: \text{“red”},\ \text{text}: \text{“favorite”}\}, \\
\quad\quad \text{cons}(\{\text{color}: \text{“white”},\ \text{text}: \text{“ book”}\}, \text{nil})))
\end{array}
$$

Be careful to ensure that the list you return does not include any extra, empty highlight records (i.e., records where the text field is the empty string) that are not actually described in the input.

For example, if the input text is "[red|one][blue|two]", your result should contain two highlight records. There shouldn't be an extra {color : "white", text : ""} record before, after, or between the two highlights. On the other hand, if the user actually writes "[red|]", then they are explicitly describing a red highlight of the empty string, so you should include that in your answer.

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

Write a formal definition of parse-highlights using calls to `parseNextHighlight`, which we will abbreviate "nh". You are also free to use the "compact" function that turns a list back into a string.

(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 of parse-highlights into a TypeScript function in `parser.ts` with the following signature:

```
export const parseHighlights = (chars: List<number>): List<Highlight> => {..};
```

The type `Highlight` is the record type used above, with string fields called "color" and "text".

(d) Write test cases for `parseHighlights` in the file `parse_test.ts`.

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) The starter code includes a simple wrapper function of `parseHighlights` called `parseText`. It takes input as a string and converts it into a list before calling `parseHighlights`.

Uncomment the code for `parseText` in `parser.ts` and the tests for it in `parser_test.ts`. Confirm these tests pass by running `npm run test`.

(f) Change the code in `index.tsx` to use `parseText` instead of `parseLines`.

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 "HW5 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:

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

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

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

(b) Suppose that $\text{len}(L) \leq 2^{k+1} - 1$ for some integer k. Show that we then have $\text{len}(P) \leq 2^k - 1$, 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 $\text{len}(R) \leq \text{len}(P)$, where $(P, S) = \text{split}(m, \text{cons}(a, L))$, $m = \lfloor \text{len}(\text{cons}(a, L))/2 \rfloor$, and $S = \text{cons}(b, R)$.

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

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

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

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 $\text{search}(y, T)$ before it returns (not just the first recursive call but any recursive calls that makes and so on) is no more than $\text{height}(T)$. (You do not need to write out the proof in full detail.)

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

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