Programming Assignment B
Due on Saturday, February 9, 2019 at 11:59 pm

The goal of this assignment is to implement 2-3 balanced trees (insertion and lookup, not deletion). What follows is a description of 2-3 trees (in boxes), interspersed with the assignment requirements. We will use these 2-3 trees to replace the simple list dictionary on the previous assignment.[1] First, do not worry about the dictionary part. Just concentrate on building a 2-3 tree.

2-3 trees are (approximately) balanced trees. The values are stored in the internal nodes, as well as the leaves of the tree (much like a standard binary tree). Each node (including the leaves) has either one or two values. If it has one value, then it has two children (a 2-node). One child (the "left") is the subtree of all values less than the node's value. The other (the "right") is the subtree of all of the values greater than the node's value. If it has two values, then it has three children (a 3-node). The left has everything less than the smaller of the two values. The right has everything greater than the larger of the two values. And the "middle" has everything that is between the two values.

Question 1. [5 points]
Define a 2-3 tree datatype called `baltree` (for balanced tree). Name the data constructor for an empty tree `EmptyTree`.

Question 2. [15 points]
Implement a function `find23` of type `('a * 'b -> int) -> 'b baltree -> 'a -> 'b option`.

`find23` is a function that takes a comparison operator, a 2-3 tree, and an item to search for and returns an option (`NONE` if it could not be found and `SOME y` if y is the item that matches). Note that for this assignment, a comparison operator takes two arguments and returns -1 if the first is less, 0 if they are equal, and +1 if the first is greater.[2] This differs from `op<` or other Boolean comparisons. However, it is necessary so that `find23` can search based on an input that isn't exactly the same type of the types stored in the dictionary. Note this means that when `find23` uses the comparison, it must always place the value for the search as the first argument and the value from the tree as the second argument.

Insertion into a 2-3 tree is a little more tricky. It can be thought of like this: First find where the value should belong at a leaf of the tree. Add (or at least consider adding) the value to this node. If the node has one value, this is simple; it just turns this node into a node with two values.

If the leaf into which the new value is to be added already has two values, then we have an "extra value." We select the middle of the three values to be this "extra value" and place the other two values in their own nodes that will become subtrees of this "extra value" (one to the left and one to the right). We tell the *parent* that it needs to insert the extra value (and two associated subtrees) into its node. If this parent node has only 1 value, this is simple. It adds this new value to itself and the associated 2 subtrees join this node (remember, they are replacing the leaf that was there).

If this parent node already has 2 values, then it repeats the process, passing the difficulty to its parent (in a method explained later). If this gets all the way to the root node and the root node is asked to add a value to an already full node, we instead create a new root node (of only the single value to be added) with the two subtrees.

In general then, as we work back up the tree, we will either be replacing a node with another node, or we will have a 3-node that cannot be replaced and we will pass the problem up to our parent. We can view this as an algorithm that works down the tree to find the proper leaf and then back up the tree reporting either a new subtree, or "problem" consisting of a extra value with two subtrees.

If we make it to the root and the result of inserting into the root is an extra value and two subtrees, then we replace the root with a node that has this "extra value" with children of the two subtrees.

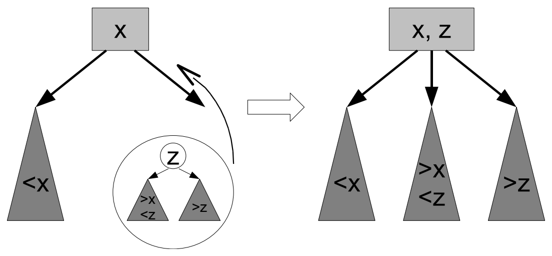
[1] Actually, a faster and more compact dictionary can be used for LZ78 encoding and decoding. We can use the fact that any prefix of a codeword is also a codeword to do linear-time lookup with the proper data structure. However, the point of this assignment is to build a more general dictionary that is also useful for LZ78, so we will not be implementing this faster structure.

[2] There is a datatype already defined for this (order), but I've chosen to use three integers to avoid introducing yet another datatype.

Examples:

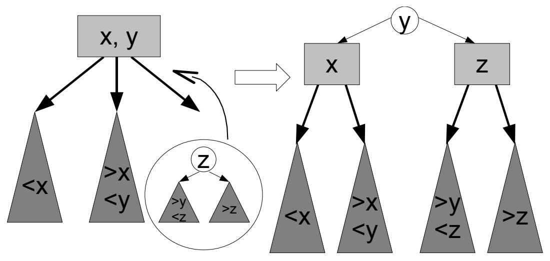
The simplest examples are where a subtree is just replaced with a different subtree. Those are not shown here. Instead, here are two examples that describe when a subtree cannot contain all of the values and so it instead reports an extra value and two subtrees.

The example below shows the easy case where a child has reported it cannot hold all of the values, but the extra value can be held in the current node.



In this picture, triangles represent subtrees. The inequalities are just statements about what we know must be true about the subtrees. In this case, we just replace our current 2-node with a 3-node. We don't bother our parent with any problems (just that this node has changed) and the extra value (and associated subtrees) does not continue to propagate.

The more complex example is shown below. This is the case where the child (in this case the right child) reports an extra value and two subtrees (instead of a simple change). Because the current node already has 2 values (it is a 3-node), we cannot keep all four values here, and thus must pass an extra value to our parent:



Thus we build two subtrees, but cannot assign either one of them to the current node. Instead we ask our parent to insert them into the correct place (so this whole structre on the right side is passed to its parent, just like the item in a circle was passed to this node). Note that if the child had not been the right child, then the result would be structurally the same, but with different values used as the extra value and the heads of the two subtrees. In general, if the child of a 3-node reports an extra value and two subtrees, we end up with 3 values and 4 subtrees. We place the middle value as the extra value to report to our parent and order the subtrees accordingly.

Question 3. [35 points]
Implement a function `insert23` of type

```
('a * 'a -> int) -> 'a baltree -> 'a -> 'a baltree
```

It takes in a comparison operator (like before, but now because the value is to be inserted, ML will insist both arguments to the comparison be of the same type), a 2-3 tree, and an object to insert and returns a new tree that is the result of inserting the object into the 2-3 tree. You may assume that the item to be inserted does not already exist in the tree.

I would recommend implementing this using a helper recursive function. The helper function can search for the insertion spot on the way "down" the recursion, and can edit the tree on the way back "up." Note that the recursive function will have to report (return) either a new tree, or an extra value and two subtrees so that its parent can figure out what to do. Defining a datatype for this return value would probably help. (Note that datatypes can also be locally scoped inside of `let` expressions.) This function has many cases (Is the current node a 2-node or a 3-node? Does the value fall into the left, middle, or right branch? Does my child report a simple change or an extra value and two subtrees?), but each case is pretty straight-forward once you understand the algorithm.

Question 4. [15 points]
Using your `find23` and `insert23`, you should now be able to define `treedictfind` and `treedictadd`. They should be of types

```
treedictfind: ('a * 'b -> int) -> ('b * 'c) baltree -> 'a -> 'c option
treedictadd: ('a * 'a -> int) -> ('a * 'b) baltree -> 'a * 'b -> ('a * 'b) baltree
```

For `treedictfind`, `'a` is the type of the value to be searched for. `'b` is the type of the key (which is actually probably the same as `'a`). `'c` is the type of value retrieved. For `treedictadd`, `'a` is the key type and `'b` is the type of the value retrieved. Again, the comparison functions should return -1, 0, or $+1$.

You can now define LZ78 encoding and decoding functions that use 2-3 trees:

```
fun lz78te cmp l = lz78e (EmptyTree,(treedictfind cmp),(treedictadd cmp)) l;
fun lz78td cmp l = lz78d (EmptyTree,(treedictfind cmp),(treedictadd cmp)) l;
```

The only difference is that they now require a comparison operator for the type to be encoded:

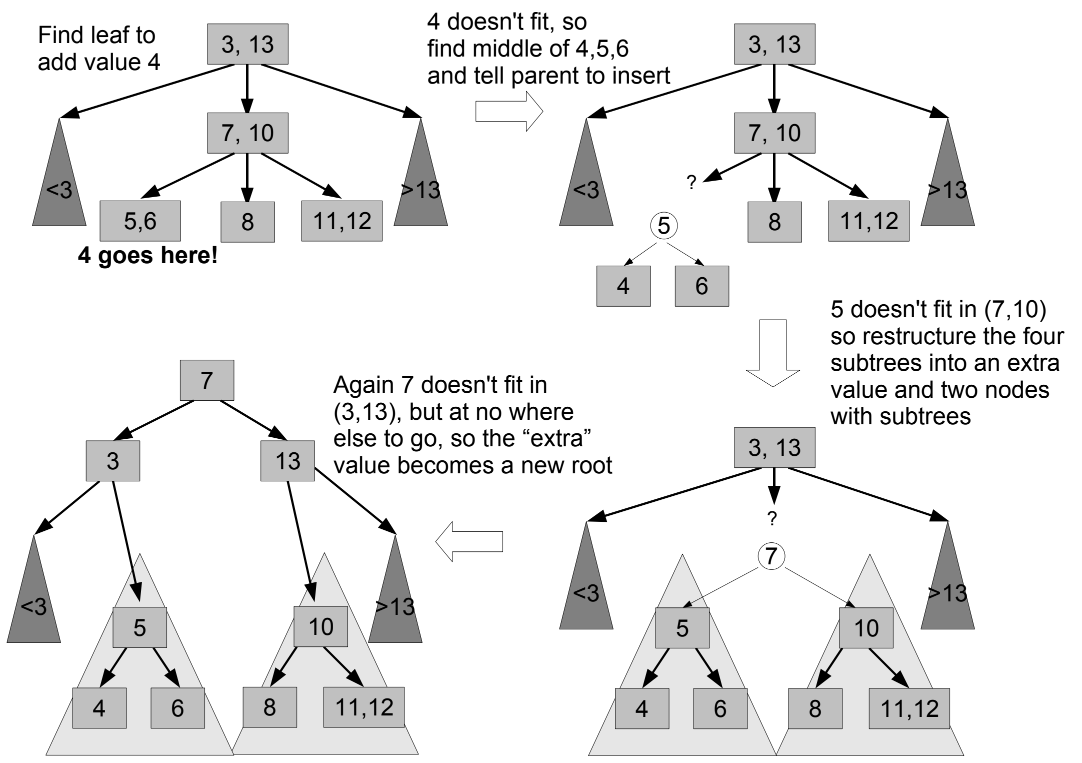
```
- fun intcmp (s1:int,s2:int) =
  if s1<s2 then ~1 else if s2<s1 then 1 else 0;
= val intcmp = fn : int * int -> int

- fun charcmp (s1:char,s2:char) =
  if s1<s2 then ~1 else if s2<s1 then 1 else 0;
= val charcmp = fn : char * char -> int

- fun listcmp _ ([],[]) = 0
- | listcmp _ (_,[]) = 1
- | listcmp _ ([],_) = ~1
- | listcmp cmp (a::ta,b::tb) =
- let val c = cmp(a,b) in if c=0 then listcmp cmp (ta,tb) else c end;
= val listcmp = fn : ('a * 'b -> int) -> 'a list * 'b list -> int

- lz78te (listcmp intcmp) [1,1,1,2,1,1,1,2,2,1,2];
val it = [(0,1),(1,1),(0,2),(2,1),(3,2),(1,2)] : (int * int) list
- lz78td intcmp it;
val it = [1,1,1,2,1,1,1,2,2,1,2] : int list

- lz78te (listcmp charcmp) (explode "aabbababab");
val it = [(0,#"a"),(1,#"b"),(0,#"b"),(2,#"a"),(3,#"a"),(0,#"b")] : (int * char) list
- implode (lz78td intcmp it);
val it = "aabbababab" : string
```

Submission Instructions:
Submit the datatype `baltree` and all functions that you wrote (`find23`, `insert23`, `treedictadd`, and `treedictfind`, as well as the supplied functions (`lz78te`, and `lz78td`) and any auxiliary functions you wrote in a single `.sml` file. *Your functions must have the names given in this assignment and the empty tree data constructor must be named* `EmptyTree`.

Please comment your code and do not submit any other files (any additional information should be placed in comments in the .sml file).

Submit the single `.sml` file via gradescope.com.

Here's one final big example. Usually an addition does not propagate all the way to the root node. However, if there are only 3-nodes on the path to the leaf, then it can happen. Here's an example of inserting a value (4) into a 2-3 tree where it results in a change to the root node.


Find leaf to add value 4
3, 13
7, 10
<3
>13
5,6
8
11,12
4 goes here!
4 doesn't fit, so find middle of 4,5,6 and tell parent to insert
?
5
4
6
5 doesn't fit in (7,10) so restructure the four subtrees into an extra value and two nodes with subtrees
7
10
Again 7 doesn't fit in (3,13), but at no where else to go, so the "extra" value becomes a new root
3
13

Note that these two trees (light triangles) are preserved. The only reorganization at any step involves the values in the current 3-node and the "extra" value passed up.

At this very last step, the algorithm constructed an extra value of 7 with two subtrees (the one with a root of 3 and the one with a root of 13). However, it could not pass this up to its parent, because there is no parent. So, this extra value became the value in a 2-node that is the new root.

Now that we inserted 4, most other changes do not result in large restructings. If we wanted to insert 9, it would only change the 2-node 8 into a 3-node of (8,9). If we wanted to add 11.5, it would involve reworking the 3-node of (11,12). However, the 2-node of 10 could add the extra value without needing to propagate the change any further.