1. Trees

Red-black trees are a form of balanced binary search trees where each node has a colour: red or black, leaves are implicitly black; and the following properties hold:

Invariant 1 No red node has a red child;

Invariant 2 Every path from the root to a leaf has the same number of black nodes.

For these questions, you may assume we are implementing sets, i.e. interested whether an element exists or not, rather than a dictionary which maps keys to values.

1. Write a datatype to represent red-black trees, and a function to test if an element is present in the tree.

[small]

This is very simple, basically out of the lecture notes. Technically you could also use `v=el` and `v<el`, if you use it consistently in `insert` also, the result is the same (until it comes to turning the tree to a list).

```OCaml
type colour = R | B;;
type 'a rbtree = Lf | Br of colour * 'a * 'a rbtree * 'a rbtree;;

let rec member tree el = match tree with
  | Lf -> false
  | Br(_, v, _, _) when el=v -> true
  | Br(_, v, l, _) when el<v -> member l el
  | Br(_, _, _, r)           -> member r el;;
```

It is very useful to create pretty-printing functions for the tree, for debugging. One way to do that is to rotate the normal tree 45 degrees anticlockwise, so it looks more like a file tree. (This just makes the printing code simpler, you can code non-rotated versions too with more difficulty, or code it so it outputs GraphViz plotting code [which is easy to do].)

```OCaml
let string_of_color = function R -> "R" | B -> "B";;
let print_tree string_of_node tree =
  let rec inner tree rprefix lprefix = match tree with
    | Lf -> print_string @@ rprefix ^ "\n"
    | Br(c, v, l, r) ->
      let node = string_of_color c ^ " " ^ string_of_node v in
      let pad = String.make (String.length node) ' 'in
      inner r (rprefix ^ node ^ "-+")
              (lprefix ^ pad ^ " |");
      inner l (lprefix ^ pad ^ " `")
              (lprefix ^ pad ^ "  ")
  in print_newline();
     inner tree "" "";;
let print_int_tree = print_tree string_of_int;;
let print_string_tree = print_tree (fun x -> x);;
#install_printer print_int_tree;;
#install_printer print_string_tree;;
```

This is slightly awkward to read, but is easy to code up. A simple tree with root black 2, and leaves red 1, red 2 prints as follows:

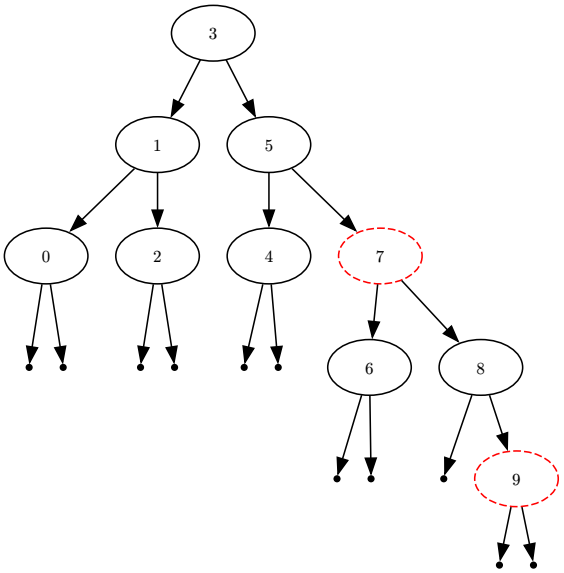
```
Br(B, 2, Br(R, 1, Lf, Lf), Br(R, 3, Lf, Lf));;
```

```
B 2-+R 3-+
    |    `
    `R 1-+
         `
```

And a tree containing ten items, zero through to nine inclusive:

```
List.init 10 (fun n -> n)
|> List.fold_left insert Lf;;
```

```
B 3-+B 5-+R 7-+B 8-+R 9-+
    |    |    |    |    `
    |    |    |    `
    |    |    `B 6-+
    |    |         `
    |    `B 4-+
    |         `
    `B 1-+B 2-+
         |    `
         `B 0-+
              `
```



2. We can implement insertion in two parts, first is finding the place to insert the element to, and inserting it not considering invariants; then fixing the tree so that the invariants hold.

   Hint 1 It is easier to fix up the invariants bottom-up (i.e. leaf first, then proceeding towards the root), because fixing the invariant at the current level might break the invariants at the level above.

   Hint 2 Which invariant we invalidate depends on what kind of node we insert: if we insert a red node then we invalidate invariant 1; if we insert a black node then we invalidate invariant 2. Out of these, inserting a red node is easier to fix up, as invariant 1 is a local property, whereas invariant 2 potentially needs to consider the whole (sub)tree. Code the insert function, you may assume that fixing the invariants is done by a function called `balance`, to be implemented in the next question.

[medium, assuming I have given enough hints]

We can actually improve the efficiency by special-casing the left-balance and right-balance, as we only unbalance the tree on the path we insert. But this isn't necessary of course, just an extremely easy optimisation.

OCaml

```
let insert tree el =
  let rec inner tree = match tree with
    | Lf -> Br(R, el, Lf, Lf)
    | Br(_, v, _, _) when el=v -> tree
```

```
5      | Br(c, v, l, r) when el<v -> balanceL (Br(c, v, inner l, r))
6      | Br(c, v, l, r)           -> balanceR (Br(c, v, l, inner r))
7    in match inner tree with
8      | Br(_, v, l, r) -> Br(B, v, l, r)
9      | Lf -> Lf (* Just to silence the warning, never occurs *);;
```

The biggest trick here is ensuring the root of the tree always has a black node, this is to make sure that we can always apply `balance` on the root of the tree – this was missing from the hints, I am expecting people to have tested this on their laptops and see that it doesn't work if they code it up naively.

3. Code the `balance` function.

   Hint 1 When you fix up the current sub-tree, you have a choice of which invariant to invalidate for the parent sub-tree. It is easier if you only invalidate invariant 1 again.

   Hint 2 Related, it is easier to fix up a black node which has a red child which itself has a red child (invariant 1 invalidated), rather than fixing up a red node which has a red child (i.e. considering two nodes deep rather than just one node deep). This is because it allows keeping the number of black nodes the same in the sub-tree.

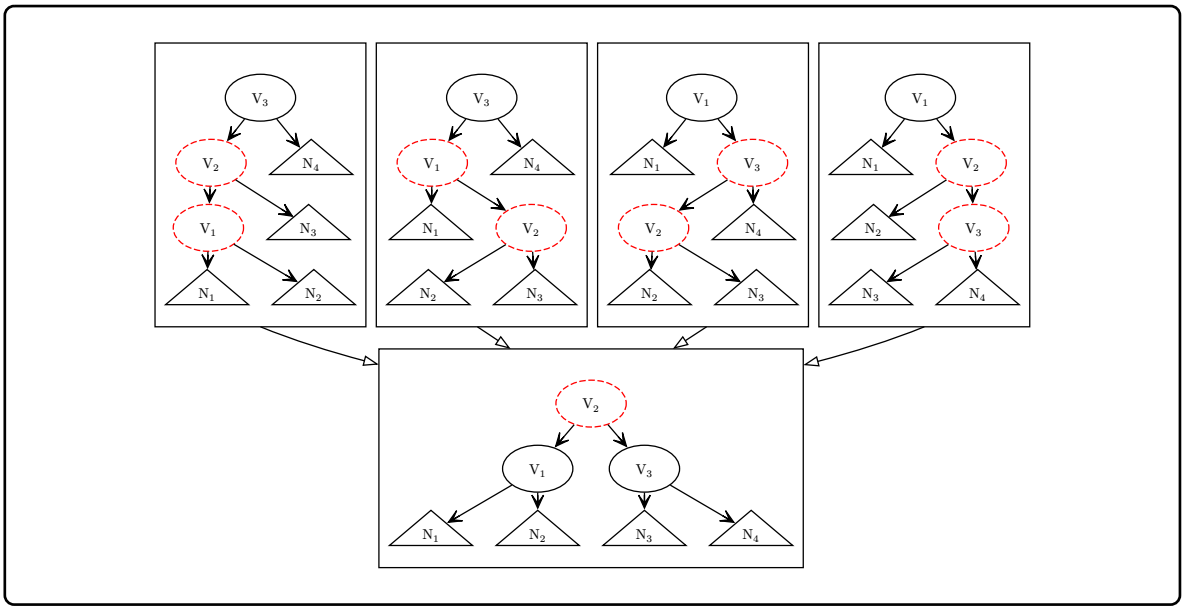
   Hint 3 We are invalidating invariant 1 in one place only, this can reduce the number of cases you need to consider.

   [medium]

This is fairly simple, especially if we graph it. I expect most people to just use a single balance function (combination of both), but the two separate can increase the efficiency by not doing unnecessary pattern-matches.

```OCaml
let balanceL = function
  | Br(B, v3, Br(R, v2, Br(R, v1, n1, n2), n3), n4) ->
    Br(R, v2, Br(B, v1, n1, n2), Br(B, v3, n3, n4))
  | Br(B, v3, Br(R, v1, n1, Br(R, v2, n2, n3)), n4) ->
    Br(R, v2, Br(B, v1, n1, n2), Br(B, v3, n3, n4))
  | t -> t
let balanceR = function
  | Br(B, v1, n1, Br(R, v3, Br(R, v2, n2, n3), n4)) ->
    Br(R, v2, Br(B, v1, n1, n2), Br(B, v3, n3, n4))
  | Br(B, v1, n1, Br(R, v2, n2, Br(R, v3, n3, n4))) ->
    Br(R, v2, Br(B, v1, n1, n2), Br(B, v3, n3, n4))
  | t -> t;;
```

Here it is visualised



2. Type-checking and inference

Type-checking and type-inference are very useful concepts, that aren't explicitly covered in the lectures, but they will help you work in OCaml much easier, as they give you an extra tool to check your programs for correctness, and types can even drive the solution to a problem. This section tries to go over some exercises bit-by-bit – do try to solve them without entering it into the computer.

1. What is the type of `let id x = x;;`? (Or more specifically, what is the type of `id` in the expression?)
   Hint The value `id` is a function returning its argument as-is.

   Simple warm-up, `val id : 'a -> 'a`. But it is useful for the introduction of the function abstraction typing rule.

   In natural language it is: "if we assume the argument has type `'a` for some new variable `'a`, and can prove the body has type `'b`; then the whole function has type `'a -> 'b`. And of course, in this case the body `x` has type `'a`. In the second year types course, this is written as

   $$\frac{\Gamma, \text{arg} : \alpha \vdash \text{body} : \beta}{\Gamma \vdash \text{fn arg} \to \text{body} : \alpha \to \beta}\,(\text{fn})$$

   Which reads, if the things above the line are true then the thing below the line is true. And $\Gamma \vdash$ expression : type is just pattern-matching on the syntax $_ \vdash _ : _$ for variables Γ, expression and type.

2. What is the type of `let z = id 5;;`?

   Equally simple, `val z : int`. But this is introducing the concept that in an expression `func arg` if `func` has type `'a -> 'b` and `arg` has type `'a`, then the whole `func arg` has type `'b`.

   Again, in the second year types course's notation, this is:

   $$\frac{\Gamma \vdash \text{func} : \alpha \to \beta \qquad \Gamma \vdash \text{arg} : \alpha}{\Gamma \vdash \text{func arg} : \beta}\,(\text{app})$$

3. What is the type of `let f x y = x y;;`?

Hint It might help to consider `f id 5`, which is a well-typed expression.

This is just gently introducing currying: `val f : ('a -> 'b) -> 'a -> 'b`

4. What is the type of `let f x y z = x y z;;`?

This is really just for comparison with the next expression:

`val f : ('a -> 'b -> 'c) -> 'a -> 'b -> 'c`

5. What is the type of `let f x y z = x y (z y);;`?
Hint Consider `let g y z = z y;;` first

Again, a gentle step-up from the previous one.

`val f : ('a -> 'b -> 'c) -> 'a -> ('a -> 'b) -> 'c`

Just so you see the second-year types course prooftrees in action (which might help put it in context, but if it doesn't make sense then you can ignore it for this year):

$$\dfrac{\dfrac{\dfrac{\dfrac{x:\alpha\in\Gamma}{\Gamma\vdash x:\iota\to\kappa\equiv\alpha}\quad\dfrac{y:\gamma\in\Gamma}{\Gamma\vdash y:\iota\equiv\gamma}}{\Gamma\vdash(x\ y):\theta\to\eta\equiv\kappa}\quad\dfrac{\dfrac{z:\varepsilon\in\Gamma}{\Gamma\vdash z:\lambda\to\theta\equiv\varepsilon}\quad\dfrac{y:\gamma\in\Gamma}{\Gamma\vdash y:\lambda\equiv\gamma}}{\Gamma\vdash(z\ y):\theta}}{(\text{let }\Gamma=\{x:\alpha,y:\gamma,z:\varepsilon\})\vdash(x\ y\ (z\ y)):\eta}}{\{x:\alpha,y:\gamma\}\vdash(\text{fun }z\to x\ y\ (z\ y)):\delta\equiv\varepsilon\to\eta}}{\{x:\alpha\}\vdash(\text{fun }y\to\text{fun }z\to x\ y\ (z\ y)):\beta\equiv\gamma\to\delta}}{\{\}\vdash(\text{fun }x\to\text{fun }y\to\text{fun }z\to x\ y\ (z\ y)):\alpha\to\beta}$$

Now we have a bunch of unifications

$$\beta\equiv\gamma\to\delta\quad\delta\equiv\varepsilon\to\eta\quad\theta\to\eta\equiv\kappa\quad\iota\to\kappa\equiv\alpha\quad\iota\equiv\gamma\quad\lambda\to\theta\equiv\varepsilon\quad\lambda\equiv\gamma$$

Which we need to apply to the type

$$\begin{aligned}
&\text{fun }x\to\text{fun }y\to\text{fun }z\to x\ y\ (z\ y):\alpha\to\beta\\
&\text{fun }x\to\text{fun }y\to\text{fun }z\to x\ y\ (z\ y):\alpha\to\gamma\to\delta\\
&\text{fun }x\to\text{fun }y\to\text{fun }z\to x\ y\ (z\ y):\alpha\to\gamma\to\varepsilon\to\eta\\
&\text{fun }x\to\text{fun }y\to\text{fun }z\to x\ y\ (z\ y):(\iota\to\kappa)\to\gamma\to\varepsilon\to\eta\\
&\text{fun }x\to\text{fun }y\to\text{fun }z\to x\ y\ (z\ y):(\gamma\to\kappa)\to\gamma\to\varepsilon\to\eta\\
&\text{fun }x\to\text{fun }y\to\text{fun }z\to x\ y\ (z\ y):(\gamma\to\theta\to\eta)\to\gamma\to\varepsilon\to\eta\\
&\text{fun }x\to\text{fun }y\to\text{fun }z\to x\ y\ (z\ y):(\gamma\to\theta\to\eta)\to\gamma\to(\lambda\to\theta)\to\eta\\
&\text{fun }x\to\text{fun }y\to\text{fun }z\to x\ y\ (z\ y):(\gamma\to\theta\to\eta)\to\gamma\to(\gamma\to\theta)\to\eta
\end{aligned}$$

Which we can rename $\gamma=$ `'a`, $\theta=$ `'b`, and $\eta=$ `'c` to get the result as previously.

`val f : ('a -> 'b -> 'c) -> 'a -> ('a -> 'b) -> 'c`

6. Why might `let f x = x x;;` not be well-typed?

It is not well typed, because we see that `x` is a function (say of type `'a -> 'b`), but also the argument of type `'a`. Hence we have `'a = 'a -> 'b`, which is an infinite expansion: expand

once, we get `'a = ('a -> 'b) -> 'b`, twice we get `(('a -> 'b) -> 'b) -> 'b`, and so on. The unification never ends.

However, we can do

```
# #rectypes;;
# let f x = x x;;
```

```
val f : ('a -> 'b as 'a) -> 'b = <fun>
```

It is not obvious what we can pass in to this function however, and in fact if we pass in the simplest function, the identity function, we get

```
# f (fun x -> x);;
```

```
- : 'a -> 'a as 'a = <fun>
```

```
# f (fun x -> x) (fun x -> x);;
```

```
- : 'a -> 'a as 'a = <fun>
```

```
# f (fun x -> x) 1;;
```

```
Error: This expression has type int but an expression was expected of type
         'a -> 'a as 'a
```

So this function is for now mysterious, and not immediately useful at all.

7. If I define

```
# type 'a ty = Wrapped of ('a ty -> 'a);;
```

what is the type of the following expression?

```
# let unwrap (Wrapped x) = x;;
```

The `'a ty` and `Wrapped` are definitely weird, though if you think back to the previous question, and squint a little, you can see the relation between `'a = 'a -> 'b` and `'a ty = Wrapped of ('a ty -> 'a)`.

In `let unwrap (Wrapped x) = x`, we know `x` has to be `'a ty -> 'a`, and the argument `Wrapped x` is of type `'a ty`, so the function has the type

```
# val unwrap : 'a ty -> 'a ty -> 'a
```

8. What might go in the place of `todo` in the following expression: `let f x = unwrap todo x;;`? What is the type of `f`?
Hint Make sure you have got the type of `unwrap` correct.

Given the previous type of `unwrap`, we know both `todo` and `x` have to have the type `'a ty`, so using `x` in place of `todo` makes sense, and gives us

```
# let f x = unwrap x x;;
# val f : 'a ty -> 'a
```

which is the same type as the argument to the type constructor `Wrapped`.

9. What is the type of `Wrapped f`?

   As noted previously, the type of `f` is the same as the argument to `Wrapped`, so the type of this expression is `'a ty`, the same as the `x` argument to `f` above.

10. What might go in the place of `todo` in `let g = todo (Wrapped f)`? Here `f` is the function defined previously. What is the type of `g`?
    Hint The value `todo` must be a function, given it is applied to a value `Wrapped f`. Look over your previous types.

    As noted previously, the expression `Wrapped f` has type `'a ty`, which is also the argument type to `f`, so if we use `f` then we have

    ```
    # let g = f (Wrapped f)
    ```

    And this has type `'a`, by looking at the type of `f`, which is a very unusual type as it is unconstrained, compared to something like the type of `id` in the first point, which has type `'b -> 'b` (renamed for clarity), so we know `'b` has to be the type of the argument.

11. What does the type of that expression tell us about the expression?

    Given it is unconstrained, it means we could in theory use it in any expression, for example as an int: `1 + g`, as a string: `"hello, " ^ g`, and this doesn't make sense – how could an expression be of any type? More specifically, how could the function `f` return any type?

    The answer is only by never returning. Indeed if you enter all of these into your OCaml interpreter, you get the OCaml interpreter never giving you the next prompt, and if you look at the system resource use of the OCaml interpreter, you see it is using 100% of the CPU.

12. Can you think of another way of creating an expression of such type?

    We can create infinite loops by writing

    ```
    # let rec loop x = loop x;;
    ```

    ```
    val loop : 'a -> 'b = <fun>
    ```

    ```
    # loop ();;
    ```

[each is tiny/small, but overall perhaps medium/big]

3. Full enumeration

Write a function that when given a list of colours, and a list of vertices, gives each possible assignment of colours to vertices. Colours and vertices need not be strings/integers, but for example:

```OCaml
all_pairings ["red"; "green"] [1; 2] =
[[("red", 1); ("red", 2)]; [("red", 1); ("green", 2)];
 [("green", 1); ("red", 2)]; [("green", 1); ("green", 2)]]
```

Note, the order of the outer list doesn't matter.

[medium]

The easy way to do this is just using `map`/`concat_map`, though this does end up doing a lot of appending.

```OCaml
(* let map = List.map *)
let rec map f = function
  | [] -> []
  | l::ls -> (f l)::(map f ls)
(* let concat_map f l = List.concat (List.map f l) *)
let rec concat_map f = function
  | [] -> []
  | l::ls -> (f l)@(concat_map f ls)
let rec all_pairings colours = function
  | [] -> [[]]
  | v::vertices ->
      concat_map
        (fun c ->
          map
            (fun vs -> (c, v)::vs)
            (all_pairings colours vertices))
        colours
```

This is a little complex to think about, I find it helps if I consider some examples:

"**`all_pairings colours []`**" returns `[[]]`, which makes sense. Conceptually `[]` might also look like a good base case, but the issue there is it doesn't then work with recursive calls. Also, `[]` is saying there are zero pairings, whereas `[[]]` is saying there is one pairing of colours to vertices, the empty list.

"**`all_pairings colours [1]`**" will match `v=1` and `vertices=[]`. We know from before that the recursive call `all_pairings colours []` will return `[[]]`, so it is the same as

```
concat_map
  (fun c ->
    map (fun vs -> (c, 1)::vs) [[]])
  colours
```

where the inner `map` will result in

```
concat_map
  (fun c ->
    [(c, 1)]]
  colours
```

and the outer `concat_map` will replace `c` in `[(c, 1)]` with each of the values of `colours` as expected. Let's say it returns `[(c1, 1)]; ...; [(cn, 1)]` for the next step.

"**`all_pairings colours [0, 1]`**" is by the same logic as the previous step, the same as

```
concat_map
  (fun c ->
```

```
      map (fun vs -> (c, 0)::vs) [[(c1, 1)]; ...; [(cn, 1)]])
  colours
```

where the inner `map` will result in

```
concat_map
  (fun c ->
    [[(c1, 0); (c1, 1)]; ..., [(cn, 0); (c1, 1)]; ...;
     [(c1, 0); (cn, 1)]; ..., [(cn, 0); (cn, 1)]]
  colours
```

This is what we are after.

Note, we can get rid of the appends by defining a double-map which does the concatenation also.

```OCaml
let concat_map2 f xs ys =
  let rec inner ixs iys = match (ixs, iys) with
    | ([], _) -> []
    | (ix::ixs, []) -> inner ixs ys
    | (ix::ixs, iy::iys) -> (f ix iy)::inner (ix::ixs) iys
  in inner xs ys;;

let rec all_pairings cols = function
  | [] -> [[]]
  | v::vs ->
    concat_map2
      (fun c vs -> (c, v)::vs)
      cols
      (all_pairings cols vs);;
```