Challenge 4: Tree Buffer

This challenge has been proposed by Radu Grigore.

The task is to verify a data structure called *tree buffer*. We start with a simple version, and then we will introduce more requirements related to efficiency.

Consider the OCaml interface

```
type 'a buf
val empty : int -> 'a buf
val add : 'a -> 'a buf -> 'a buf
val get : 'a buf -> 'a list
```

with the following implementation

```
type 'a buf = { h : int; xs : 'a list }

let rec take n xs = match n, xs with
  | 0, _ | _, [] -> []
  | n, x :: xs -> x :: take (n-1) xs

let empty h = { h; xs = [] }
let add x { h; xs } = { h; xs = x :: xs }
let get { h; xs } = take h xs
```

When we create a tree buffer, we fix a parameter h, and all invocations to get will return the last h elements added in the tree buffer. Incidentally, this is a tree in the sense that one can write code like the following:

```
let e = empty 3;;   (* e is a root *)
let t1 = add 1 e;;  (* t1 has parent e *)
let t2 = add 2 t1;; (* t2 has parent t1 *)
let t3 = add 3 t1;; (* t3 has parent t1 *)
```

One problem with our current implementation is that it wastes memory. A possible solution is the following caterpillar implementation:

```
type 'a buf = {h : int; xs : 'a list; xs_len : int; ys : 'a list}

let empty h = { h; xs = []; xs_len = 0; ys = [] }
let add x { h; xs; xs_len; ys } =
  if xs_len = h - 1
```

```
  then { h; xs = []; xs_len = 0; ys = x :: xs }
  else { h; xs = x :: xs; xs_len = xs_len + 1; ys }
let get { h; xs; xs_len; ys } = take h (xs @ ys)
```

The length of `xs` is always less that `h` because, whenever it would become `h`, its content is moved into `ys`, and the old content of `ys` is thrown away.

Verification Tasks:

1. Verify that the naive and the caterpillar implementations are functionally equivalent. You are encouraged to translate (a) the naive implementation into your favorite specification language, and (b) the caterpillar implementation into your favorite imperative language. Your caterpillar implementation must use only constant time for add, ignoring time possibly spent in the garbage collector.

2. In this task we want to move to a real-time setting. This means, in particular, that we can no longer ignore the time spent in the garbage collector. But, we still want
   - a constant time add operation, and
   - memory use that is within a constant factor of the caterpillar's live-heap size.

   The idea is to add reference counters to the caterpillar. In addition, we need to control the rate at which objects are deleted by the reference counting scheme. We control the rate by holding some references in the queue to_delete.

```
// g++ -std=c++14

#include <memory>
#include <queue>

struct List { virtual ~List() {} };
typedef std::shared_ptr<List> PL;

struct Nil : List {};
struct Cons : List {
  int head; PL tail;
  Cons(int head, PL tail) : head(head), tail(tail) {}
  virtual ~Cons();
};

std::queue<PL> to_delete;

// IMPORTANT: delay the deletion,
// so that time per operation is constant
Cons::~Cons() { to_delete.push(tail); }

struct Buf {
  int h; PL xs; int xs_len; PL ys;
```

```
  Buf(int h, PL xs, int xs_len, PL ys) : h(h), xs(xs),
        xs_len(xs_len), ys(ys) {}
  virtual ~Buf(); // this is where our manual GC is triggered
  PL get(); // unimplemented here; would use xs and ys
private:
};

void process_queue() {
  if (!to_delete.empty()) to_delete.pop();
  if (!to_delete.empty()) to_delete.pop();
}

// NOTE: This is called automatically by C++.
// In the C implementation, there is a |deactivate|
// function that plays the role of this destructor.
Buf::~Buf() { process_queue(); }

Buf empty(int h) {
  return Buf(h, PL(new Nil()), 0, PL(new Nil()));
}

Buf add(int x, Buf t) {
  process_queue();
  if (t.xs_len + 1 == t.h) {
    return Buf(t.h, PL(new Nil()), 0, PL(new Cons(x, t.xs)));
  } else {
    return Buf(t.h, PL(new Cons(x, t.xs)), 1 + t.xs_len, t.ys);
  }
}

int main() {
  Buf e = empty(3);
  Buf t1 = add(1, e);
  Buf t2 = add(2, t1);
  Buf t3 = add(3, t1);
}
```

Your tasks are:

1. show that the real-time implementation is functionally equivalent to the naive implementation, and
2. prove bounds on the resources used, such as time spent in add, total memory used, and so on.

Warning: The C++ implementation above has not been tested, so it probably has bugs. A C implementation, which has been tested but has the drawback of verbosity, can be found on GitHub. There is also a (inefficient but pretty) javascript implementation in these slides (go to 'real-time…').

You are free to verify whichever implementation you choose: the important part is to satisfy the same, real-time performance characteristics. That does mean that you cannot use a language with garbage collection, unless you can control the garbage collector, and you can verify resource bounds for the system including your program and the garbage collector.