Lista zadań nr 4

Zadanie 1.

Rozważ drzewa binarne z wykładu. Narysuj, jak w pamięci reprezentowane jest drzewo `t` zdefiniowane poniżej:

```
(define t
  (node
    (node (leaf) 2 (leaf))
    5
    (node (node (leaf) 6 (leaf))
          8
          (node (leaf) 9 (leaf)))))
```

Pokaż, jak będzie wyglądał stan pamięci po wykonaniu wstawienia BST wartości 7. Które fragmenty drzewa t są współdzielone między drzewem `t` i `(insert-bst 7 t)`?

Zadanie 2.

Tak jak najprostszy schemat rekursji na listach można wyabstrahować do procedury `foldr`, tak najprostszy schemat rekursji na drzewach z wykładu można wyabstrahować do procedury `fold-tree`. Zdefiniuj tę procedurę i wyraź przy jej użyciu procedury:

- `(tree-sum t)` – suma wszystkich wartości występujących w drzewie,
- `(tree-flip t)` – odwrócenie kolejności: zamiana lewego i prawego poddrzewa wszystkich węzłów w drzewie,
- `(tree-height t)` – wysokość drzewa (liczba węzłów na najdłuższej ścieżce od korzenia do liścia),
- `(tree-span t)` – para złożona z wartości skrajnie prawego i skrajnie lewego węzła w drzewie (czyli najmniejszej i największej wartości w drzewie BST),
- `(flatten t)` – lista wszystkich elementów występujących w drzewie, w kolejności infiksowej.

Zadanie 3.

Zaimplementuj następujące procedury:

- `(bst? t)` – predykat sprawdzający, czy zadane drzewo spełnia własność BST. Zadbaj o to, aby każdy wierzchołek drzewa odwiedzić tylko raz.
- `(sum-paths t)` – produkuje drzewo o tym samym kształcie co `t`, w którym etykietą danego węzła jest suma wartości węzłów na ścieżce od korzenia drzewa do tego węzła. Na przykład:

```
> (sum-paths t)
(node (node (leaf) 7 (leaf))
      5
      (node (node (leaf) 19 (leaf))
            13
            (node (leaf) 22 (leaf)))))
```

Zadanie 4. (2 pkt)

Najprostsza implementacja procedury `flatten` z poprzedniego zadania posiada poważną wadę – tworzy ona duże ilości nieużytków oraz wykonuje nadmiarowe obliczenia. Tę wadę można szczególnie dobrze zaobserwować na przykładzie drzew, które „rosną tylko w lewo":

```
(define (list->left-tree xs)
  (foldl (lambda (x t) (node t x (leaf))) (leaf) xs))
(define test-tree (list->left-tree (build-list 20000 identity)))
```

Napisz inną implementację `flatten`, która nie posiada tej wady. Nie używaj procedury append!

Wskazówka: zaimplementuj najpierw dwuargumentową procedurę `(flat-append t xs)`, której wynikiem jest lista elementów `t` w kolejności infiksowej scalona z listą `xs`. Przykład:

```
> (flat-append t (list 10 11))
'(2 5 6 8 9 10 11)
```

Zadanie 5.

Zmodyfikuj procedurę `insert` z wykładu (wstawiającą element do drzewa BST) tak, aby możliwe było tworzenie drzew BST z duplikatami. Możesz założyć, że elementy równe elementowi w korzeniu drzewa będą trafiać do lewego poddrzewa.

Zaimplementuj procedurę `(treesort xs)`, implementującą algorytm sortowania przy użyciu drzew BST:

- Utwórz drzewo przeszukiwania składające się z elementów listy `xs`.
- Zwróć listę elementów drzewa w kolejności infiksowej.

Zadanie 6.

Zaimplementuj procedurę `delete`, zwracającą drzewo z usuniętym danym kluczem, dla reprezentacji drzew przeszukiwań binarnych z wykładu. Wskazówka: Aby stworzyć drzewo przeszukiwań binarnych z którego usunęliśmy korzeń, najlepiej znaleźć (jeśli istnieje) najmniejszy element większy od tego korzenia.

Zadanie 7. (2 pkt)

Kolejka FIFO *(first in, first out)* to struktura danych, do której można dodać element „na koniec", a także podejrzeć i wyjąć element „na początku". Prosta implementacja kolejki przy pomocy listy może wyglądać tak:

```
(define empty-queue     ; pusta kolejka
    null)
(define (empty? q)      ; czy kolejka jest pusta?
    (null? q))
(define (push-back x q) ; dodaj element na koniec kolejki
    (append q (list x)))
(define (front q)       ; podejrzyj element na poczatku kolejki
    (car q))
(define (pop q)         ; zdejmij element z przodu kolejki
    (cdr q))
```

Ta implementacja kolejki jest bardzo czytelna, ale ma wielką wadę związaną z użyciem procedury `append` w implementacji procedury `push-back`. Lepszą reprezentacją kolejki jest **para list**: pierwsza to prefiks kolejki, a druga to sufiks w odwróconej kolejności. W ten sposób mamy łatwy dostęp do przodu kolejki (bo reprezentowany jest przez przód listy) i do tyłu kolejki (bo też reprezentowany jest przez przód listy). Dokładniej: pierwszy element kolejki to pierwszy element pierwszej listy, a nowe elementy możemy dokładać poprzez dokładanie ich na przód drugiej listy. Dopiero gdy skończą się elementy na pierwszej liście, zastępujemy ją odwróconą drugą listą.

Sformalizuj ten nieformalny opis poprzez lepszą implementację podanego wyżej interfejsu. By zawsze mieć dostęp do pierwszego elementu kolejki, zachowaj nastepujący niezmiennik: pierwsza lista jest pusta tylko wtedy, gdy druga lista jest pusta.

Zadanie 8. (2 pkt)

Kopce lewicowe (znane też jako drzewa lewicowe) to prosta i efektywna struktura danych implementująca kolejkę priorytetową (którą na wykładzie zaimplementowaliśmy używając nieefektywnej struktury listy posortowanej), zaproponowana w 1972

roku przez Clarka Crane'a i uproszczona rok później przez Donalda Knutha. Podobnie jak w przypadku posortowanej listy, chcemy móc znaleźć najmniejszy element w stałym czasie, jednak chcemy żeby pozostałe operacje (wstawianie, usuwanie minimum i scalanie dwóch kolejek) działały szybko — czyli w czasie logarytmicznym. W tym celu, zamiast listy budujemy *drzewo binarne*, w którym wierzchołki zawierają elementy kopca wraz z wagami. Dodatkowym niezmiennikiem struktury danych, który umożliwi efektywną implementację jest to, że każdemu kopcowi przypisujemy *rangę*, którą jest długość „prawego kręgosłupa" (czyli ranga prawego poddrzewa zwiększona o 1 — lub zero w przypadku pustego kopca), i że w każdym poprawnie sformowanym kopcu ranga lewego poddrzewa jest nie mniejsza niż ranga prawego poddrzewa.

Pozwala to nam zdefiniować następującą implementację:

```
(define-struct ord (val priority) #:transparent)

(define-struct hleaf ())
(define-struct hnode (elem rank l r) #:transparent)

(define (make-node elem heap-a heap-b)
  ;;; XXX: fill in the implementation
  ...)

(define (hord? p h)
  (or (hleaf? h)
      (<= p (ord-priority (hnode-elem h)))))

(define (rank h)
  (if (hleaf? h)
      0
      (hnode-rank h)))

(define (heap? h)
  (or (hleaf? h)
      (and (hnode? h)
           (heap? (hnode-l h))
           (heap? (hnode-r h))
           (<= (rank (hnode-r h))
               (rank (hnode-l h)))
           (= (hnode-rank h) (+ 1 (hnode-rank (hnode-r h))))
           (hord? (ord-priority (hnode-elem h))
                  (hnode-l h))
           (hord? (ord-priority (hnode-elem h))
                  (hnode-r h)))))
```

Wierzchołki reprezentujemy przy użyciu struktury `hnode`, którego polami są: element kopca `elem` (będący strukturą `ord`), ranga wierzchołka `rank`, lewe i prawe poddrzewo. Predykat `heap?` sprawdza czy zachowany jest porządek kopca (używając `hord?`) i czy własność rangi opisana powyżej jest spełniona.

Zaimplementuj procedurę („inteligentny konstruktor") `make-node`. Zwróć uwagę, że `make-node` nie przyjmuje rangi tworzonego kopca, ale musi ją wyliczyć. Oznacza też, że musimy stwierdzić w procedurze konstruktora który z kopców powinien zostać prawym, a który lewym poddrzewem (możemy natomiast założyć że porządek kopca zostanie zachowany). Zwróć uwagę na użycie procedury `ord-priority` znajdującej priorytet elementu w kopcu (który powinien być liczbą).

Zaimplementuj następnie procedurę `heap-merge` złączającą dwa kopce. Idea scalania kopców jest następująca: jeśli jeden z kopców jest pusty, scalanie jest trywialne (bierzemy drugi kopiec). Jeśli oba są niepuste, możemy znaleźć najmniejszy element każdego z nich. Mniejszy z tych dwóch elementów powinien znaleźć się w korzeniu wynikowego kopca — łatwo go znaleźć. Mamy zatem cztery obiekty:

- element o najniższym priorytecie (nazwiemy go e),
- lewe poddrzewo kopca z którego korzenia pochodzi $e - h_l$
- prawe poddrzewo kopca z którego korzenia pochodzi $e - h_r$
- drugi kopiec, h, którego korzeń miał priorytet większy niż e.

Aby stworzyć wynikowy kopiec wystarczy teraz scalić h_r i h (rekurencyjnie), a następnie stworzyć wynikowy kopiec z kopca otrzymanego przez rekurencyjne scalanie, kopca h_l i elementu e. Implementując `heap-merge`, wykorzystaj procedurę `make-node`.

Zadanie 9.

Wykorzystaj strukturę danych kopca z poprzedniego zadania, aby zaimplementować interfejs kolejki priorytetowej z wykładu – to znaczy, zaproponuj definicje `empty-pq`, `pq-insert`, `pq-pop`, `pq-min` oraz `pq-empty?` wykorzystujące, zamiast list uporządkowanych, kopce lewicowe.

Używając interfejsu kolejki priorytetowej, zaimplementuj procedurę `(pqsort xs)`, działającą w następujący sposób:

- Utwórz kolejkę priorytetową składającą się z elementów listy `xs` (priorytetem elementu niech będzie on sam).
- Skonstruuj posortowaną listę wynikową przez wyjmowanie kolejnych elementów z kolejki.

Implementacja powinna działać poprawnie dla obu implementacji kolejek priorytetowych (listy uporządkowane i kopce lewicowe).