Algorytmy funkcjonalne i trwałe struktury danych 2024

Lista zadań nr 1

Na zajęcia 27 lutego 2024

Zamiast SML-a w rozwiązaniach zadań można używać OCaml-a.

Zadanie 1. Oto naiwne implementacje kilku funkcji w SML-u:

```
fun sublist (x::xs) =
    let
       fun addx (ys::yss) = (x::ys) :: addx yss
         | addx nil = nil
       val xss = sublist xs
    in
       xss @ addx xss
    end
  | sublist [] = [[]]

datatype tree = & of int * (tree * tree) | %
infix &
fun flatten (x&(t1,t2)) = flatten t1 @ [x] @ flatten t2
  | flatten % = nil

fun rev (x::xs) = rev xs @ [x]
  | rev nil = nil
```

Wskaż w których miejscach te funkcje tworzą nieużytki. Jak dużo ich jest? Jaki jest czas działania tych funkcji? Zaprogramuj ich efektywne wersje, które nie marnują ani jednej komórki pamięci (cała przydzielona pamięć staje się częścią wyniku).

W poniższych zadaniach funkcja sortująca powinna mieć typ

```
sort : ('a * 'a -> bool) -> 'a list -> 'a list
```

Zadanie 2. Zaimplementuj w SML-u algorytm *Quicksort*. Pokaż, że jego przeciętny czas działania i przeciętna liczba *consingów* przy jednostajnym rozkładzie danych wejściowych wynoszą $O(n \log n)$, a pesymistyczny czas działania i pesymistyczna liczba *consingów* wynoszą $O(n^2)$. Ile jednocześnie potrzeba żywych komórek pamięci? Jaka jest głębokość rekursji?

Zadanie 3. Zaimplementuj w SML-u algorytm *Mergesort*. Pokaż, że jego pesymistyczny czas działania i pesymistyczna liczba *consingów* wynoszą $O(n \log n)$. Ile jednocześnie potrzeba żywych komórek pamięci? Jaka jest głębokość rekursji?

Zadanie 4. Zaprogramuj w Prologu algorytm sortowania list, który działa w czasie $O(n \log n)$ i wykonuje $O(n)$ *consingów* (zatem zmienne logiczne, podobnie jak spamiętywanie, pozwalają na ograniczenie generowania nieużytków).

Zadanie 5. Zaprogramuj w SML-u algorytm sortowania list, który działa w czasie $O(n^2)$ i wykonuje $O(n)$ *consingów*.

Zadanie 6. Udowodnij, że w modelu pamięci SML-a nie istnieje algorytm sortowania, który działa w czasie $O(n \log n)$ i wykonuje $O(n)$ *consingów*.

Zadanie 7. Udowodnij, że czas potrzebny na usunięcie nieużytków wynosi $O(t)$, gdzie t jest czasem działania algorytmu generującego te nieużytki. *Wniosek*: analizując złożoność algorytmów można przyjąć, że komputer dysponuje nieograniczoną pamięcią, a nieużytki nie są usuwane. Takie założenie nie zmienia bowiem klasy złożoności algorytmu.

Zadanie 8. Udowodnij, że ogonowa implementacja funkcji `@` generuje $\Omega(n)$ nieużytków, gdzie n jest długością pierwszej ze spinanych list. (Nieogonowa wersja `@` nie powoduje nieużytków, ale zużywa $\Omega(n)$ pamięci do przechowywania rekordów aktywacji funkcji. Wniosek: nie istnieją małpy ogonowe).

Zadanie 9. Zaprogramuj funkcję

$$\texttt{suffixes}\ [x_1, \ldots, x_n] \quad = \quad [[x_1, \ldots, x_n], [x_2, \ldots, x_n], \ldots, [x_n], []]$$

tak by działała ona w czasie $O(n)$, rezerwowała $O(n)$ komórek pamięci i nie generowała nieużytków.

Zadanie 10. Zaprogramuj standardowe funkcje

```
foldl, foldr: ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

(W OCaml-u i Haskellu mają one różną kolejność argumentów). Która jest ogonowa, a która nie? Przepisz nieogonową tak, żeby nie przepełniała stosu, tj. była ogonowa (ale być może tworzyła nieużytki). Porównaj te implementacje z rozwiązaniami Haskellowymi? Która w Haskellu jest inkrementacyjna, a która monolityczna?

Zadanie 11. Trwała tablica, to w tym zadaniu struktura danych implementująca następujący interfejs:

```
signature PERSISTENT_ARRAY =
sig
   type 'a array
   val empty : 'a array
   val sub : 'a array * int -> 'a option
   val update : 'a array * int * 'a -> 'a array
end
```

(`empty` jest pustą tablicą, zaś `sub` realizuje operację *lookup* i zwraca `NONE`, jeśli element o podanym indeksie nie został zainicjalizowany). Używając trwałych struktur danych dostępnych w SML-u zaprogramuj trwałe tablice, w których operacje `lookup` i `update` działają w czasie $O(\log n)$, gdzie n jest indeksem elementu ($n \geq 0$).

Zadanie 12. Tablice zaimplementowane w bibliotece standardowej SML-a w postaci struktury `Array :> ARRAY` mają stałe czasy dostępu do elementu (`sub`) i modyfikacji (`update`), ale są ulotne. Struktura danych jest *częściowo trwała*, jeżeli operacje *update* mogą być wykonywane jedynie na najnowszej wersji struktury (i — tak jak w trwałych strukturach — nie modyfikują poprzednich wersji). Użyj ulotnych tablic do zaimplementowania częściowo trwałych tablic, w których czas operacji `sub` na pewnej wersji tablicy wynosi $O(k)$, gdzie k jest liczbą wersji tablicy utworzonych na jej podstawie, zaś czas operacji `update` wynosi $O(1)$.

Zadanie 13. Oto naiwna implementacja drzew ze wstawianiem bez powtórzeń:

```
datatype tree = & of int * (tree * tree) | %
infix &
fun member (x,y&(t1,t2)) =
       if x <= y
          then if y <= x
                  then true
                  else member (x,t1)
          else member (x,t2)
  | member (x,%) = false
fun insert (x, t as y&(t1,t2)) =
       if x <= y
          then if y <= x
                  then t
                  else y&(insert (x,t1), t2)
          else y&(t1, insert (x,t2))
  | insert (x,%) = x&(%,%)
```

1. W najgorszym przypadku `member` wykonuje $2h$ porównań, gdzie h jest wysokością drzewa. Napisz wersję `member`, która wykonuje co nawyżej $h + 1$ porównań w najgorszym przypadku.

2. Funkcja `insert` kopiuje ścieżkę w drzewie nawet wówczas, gdy nie dochodzi do wstawienia elementu. Napisz bardziej efektywną wersję, która zwraca oryginalne drzewo, jeśli do wstawienia nie dochodzi.

3. Połącz idee poprzednich dwóch punktów i napisz funkcję, która nie wykonuje zbędnego kopiowania i wykonuje co najwyżej $h + 1$ porównań.

Zadanie 14. Napisz funkcję

```
cbt : int -> int -> tree
```

która buduje pełne drzewo binarne podanej wysokości, którego wszystkie wierzchołki mają tę samą, podaną etykietę. Ile pamięci zużywa ta funkcja? Wyjaśnij jak działa mechanizm współdzielenia w tym przypadku.

Napisz następnie funkcję `bt`, która buduje idealnie wyważone drzewo o podanej liczbie wierzchołków, którego wszystkie wierzchołki mają tę samą, podaną etykietę. Zadbaj o to, by i w tym przypadku maksymalnie wykorzystać współdzielenie.