Analiza Algoritmilor
Tema 2

Dan-George Filimon
322 CA

21 decembrie 2010

1	2	3	4	5	6	7	8	9	Total

Pentru tipul `List` definit prin constructorii:

- $[] :\rightarrow List$
- $[a] : T \rightarrow List$ (îl vom considera un caz particular al lui $a :: []$)
- $a :: x : T \times List \rightarrow List$

1 Definiți funcțiile:

1. $append(l_1, l_2)$ – concatenarea a două liste, l_1 și l_2

```
(define append
  (lambda (l1 l2)
    (if (eq? l1 '()) l2
        (cons (car l1) (append (cdr l1) l2))
        )
    )
  )
```

2. $member(a, l)$ – verifică apartenența elementului a în lista l

```
(define member?
  (lambda (a l)
    (if (eq? l '()) #f
        (if (= a (car l)) #t
            (member? a (cdr l))
            )
        )
    )
  )
```

3. $size(l)$ – dimensiunea listei l

```
(define size
  (lambda (l)
    (if (eq? l '()) 0
        (+ 1 (size (cdr l)))
        )
    )
  )
```

4. $maxelem(l)$, $minelem(l)$ – elementul maxim respectiv minim din lista l

```
(define minelem
  (lambda (l)
    (if (eq? l '()) #f
        (let ( (m (minelem (cdr l))) )
          (if (or (eq? m #f) (< (car l) m)) (car l)
              m
              )
          )
        )
    )
  )

(define maxelem
  (lambda (l)
    (if (eq? l '()) 0
        (let ( (m (maxelem (cdr l))) )
          (if (< m (car l)) (car l)
              m
              )
          )
        )
    )
  )
```

5. $set(l)$ – întoarce lista care conține elementele distincte din l

```
(define set
  (lambda (l)
    (if (eq? l '()) '()
        (let ( (s (set (cdr l))) )
          (if (member? (car l) s) s
              (cons (car l) s)
              )
          )
        )
    )
  )
```

6. $nbrelems(l)$ – numărul de elemente din lista l

```
1 (define nbrelems
2   (lambda (l)
3     (size (set l))
4     )
5   )
```

2 Arătați că pentru orice l de tipul *List*:

$$member(a, l) \rightarrow minelem(l) \leq a \leq maxelem(l)$$

R. Demonstrația pentru cele două părți ale inegalității este analoagă fiindcă funcțiile definite sunt, cu mici variațiuni, la fel. Vom demonstra $member(a, l) \rightarrow minelem(l) \leq a$.

1. Ipoteza inductivă: Pentru orice listă $l' \in Lists$ cu $size(l') < size(l)$ presupunem că ține $member(a, l') \rightarrow minelem(l') \leq a$.
2. Cazul de bază: Deoarece `(member a '()) = #f`, nu avem ce verifica.
3. Pasul de inducție: Fie lista $l = x :: l'$ și un element $a \in l$. Avem de verificat două situații - $a = x$, sau $a \neq x$.
   (a) $a \neq x \Rightarrow a \in l'$ și conform ipotezei inductive, $minelem(l') \leq a$. Dar, cum $l = x :: l' \Rightarrow minelem(l) \leq minelem(l') \leq a$.
   (b) $a = x$, caz în care, pentru $m = minelem(l')$, fie $x < m \Rightarrow minelem(x :: l) = x \leq x = a$ (linia 5), fie $x \geq m \Rightarrow minelem(x :: l) = m \leq x = a$ (linia 6).

3 Arătați că pentru orice l de tipul *List*:

1. $size(set(l)) \leq size(l)$
   R.
   (a) Ipoteza inductivă: Pentru orice $l' \in Lists$ cu $size(l') < size(l)$, $size(set(l')) < size(l')$.
   (b) Cazul de bază: $size(set([])) = size([]) = 0$ (linia 3 din `set`, linia 3 din `size`).
   (c) Pasul de inducție: Pentru $l = x :: l'$, conform definiției lui `set`, $set(x :: l') = x :: set(l')$ dacă $x \notin l'$ (linia 6 din `set`), respectiv $set(l')$ dacă $x \in l'$ (apartenența este echivalentă cu `(member a l) = #t`):
      i. $x \in l' \Rightarrow set(x :: l') = set(l')$, deci $size(set(x :: l')) = size(set(l')) \leq size(l')$ conform ipotezei inductive. Dar, cum $size(x :: l') = 1 + size(l')$ (linia 4 din `set`) rezultă că $size(l') \leq size(l) \Rightarrow size(set(x :: l')) \leq size(x :: l')$.
      ii. $x \notin l' \Rightarrow set(x :: l') = x :: set(l') \Rightarrow size(set(x :: l')) = size(x :: set(l')) = 1 + size(set(l')) \leq 1 + size(l') = size(x :: l')$ (linia 6 din `set`, linia 4 din `size`, ipoteza inductivă). Rezultă că $size(set(x :: l')) \leq size(x :: l')$.

2. $nbrelems(l) \leq size(l)$
   R. Conform definiției lui `nbrelems` (linia 3), $nbrelems(l) = size(set(l))$, și cum am demonstrat la punctul 1. că $size(set(l)) \leq size(l) \Rightarrow nbrelems(l) \leq size(l)$.

4 Demonstrați că pentru orice $l \in Lists$:

$$size(set(l)) = size(l) \rightarrow set(l) = l$$

R.

1. Ipoteza inductivă: Pentru orice listă $l' \in Lists$ care verifică $size(set(l')) = size(l')$ și pentru care $size(l') < size(l)$, este adevărat că $set(l') = l'$.

2. Cazul de bază: Cum $size(set([])) = size([])$ este adevărat (am justificat la problema 3, punctul 1) și deoarece $set([]) = []$ (linia 3), cazul de bază se verifică.

3. Pasul de inducție: Fie lista $l = x :: l'$ care respectă $size(set(x :: l')) = size(x :: l')$ și pentru care l' verifică ipoteza inductivă. Să presupunem prin reducere la absurd că $set(x :: l') \neq x :: l'$. Atunci, deoarece știm din ipoteza inductivă că $set(l') = l'$, cu siguranță $set(x :: l') = set(l')$ (linia 5 din definiția lui `set`), altfel $set(x :: l') = x :: l'$. Deci, rezultă că $size(set(x :: l')) = size(set(l')) = size(l')$. Dar, din ipoteză știm că $size(set(x :: l')) = size(x :: l') = 1 + size(l')$ (definiția lui `size`, linia 4). Înseamnă că $size(l') = 1 + size(l') \Rightarrow 0 = 1$. Contradicție. Rezultă că proprietate este adevărată.

5 Definiți funcțiile:

1. $remove(a, l)$ – întoarce lista rezultată prin eliminarea tuturor elementelor egale cu a din l

```
1  (define remove
2    (lambda (a l)
3      (if (eq? l '()) '()
4          (let ( (lp (remove a (cdr l))) )
5            (if (= (car l) a) lp
6                (cons (car l) lp)
7                )
8            )
9          )
10     )
11   )
```

2. $double(a, l)$ – întoarce lista rezultată prin dublarea fiecărui element egal cu a din lista l ($a \to aa$)

```
1  (define double
2    (lambda (a l)
3      (if (eq? l '()) '()
4          (let (
5                (lp ( cons (car l) (double a (cdr l)) ))
6                )
7            (if (= (car l) a) (cons a lp)
8                lp
9                )
10           )
11         )
12     )
13   )
```

Să se demonstreze că $member(a, l) \to size(remove(a, l)) \neq size(double(a, l))$.
R. Vom demonstra că $size(remove(a, l)) < size(double(a, l))$, ceea ce implică proprietatea din enunț.

1. Ipoteza inductivă: Presupunem că pentru orice listă $l' \in Lists$ care în plus respectă $size(l') < size(l)$, dacă $member(a, l)$ (`(member? a l) = #t`) atunci $size(remove(a, l)) < size(double(a, l))$.

2. Cazul de bază: $member(a, []) = false$, $\forall a \in T$, deci nu avem ce verifica.

3. Pasul de inducție: Fie $l = x :: l'$, unde l' este o listă care respectă ipoteza de inducție. Pentru a demonstra proprietatea, să considerăm cazurile:

   (a) $a = x \Rightarrow size(remove(a, x :: l')) = size(remove(a, l'))$ (linia 5 din `remove`). De asemenea $size(double(a, x :: l')) = size(a :: x :: double(a, l')) = 2+size(double(a, l'))$ (liniile 5 și 7 din `double`). Deoarece, conform ipoteze inductive, $size(remove(a, l')) < size(double(a, l')) \Rightarrow size(remove(a, l')) = size(remove(a, x :: l')) < size(double(a, l'))+ 2 = size(double(a, x :: l'))$.

   (b) $a \neq x \Rightarrow size(remove(a, x :: l') = size(x :: remove(a, l'))$ (linia 6 din `remove`). În plus, $size(double(a, x :: l')) = size(x :: double(a, l'))$ (liniile 5 și 8 din `double`). Conform ipotezei inductive, avem că $size(remove(a, l')) = size(remove(a, x :: l')) < size(double(a, l')) = size(double(a, x :: l'))$.

   Rezultă că proprietatea pe care am definit-o este adevărată pentru orice element $a \in T$ și orice listă l Din ea urmează imediat și proprietatea cerută.

6 Să se demonstreze că:

$$remove(a, append(l_1, l_2)) = append(remove(a, l_1), remove(a, l_2))$$

R Deoarece proprietatea de demonstrat este simetrică în l_1 și l_2, iar $a \in T$ este ales oarecare, vom fixa a și l_2 și vom trata diversele cazuri pentru l_1.

1. Ipoteza de inducție: Pentru orice $l_1' \in Lists$ cu $size(l_1') < size(l_1)$, ține proprietatea $remove(a, append(l_1', l_2)) = append(remove(a, l_1'), remove(a, l_2))$.

2. Cazul de bază: Fie $l_1 = []$. Atunci, $remove(a, append([], l_2)) = remove(a, l_2)$ (linia 3 din `append`) și $append(remove(a, []), remove(a, l_2)) = append([], remove(a, l_2) = remove(a, l_2)$ (linia 3 din `remove` și linia 3 din `append`). Deci, cazul de bază este adevărat.

3. Pasul de inducție: Fie $l_1 = x :: l_1'$, unde l_1' verifică ipoteza inductivă. Distingem cazurile:

   (a) $a \neq x$: În stânga, $remove(a, append(x :: l_1', l2)) = remove(a, x :: append(l_1', l2)) = x :: remove(a, append(l_1', l2))$ (linia 4 din `append`, liniile 4 și 6 din `remove`).
   În dreapta, $append(remove(a, x :: l_1'), remove(a, l_2)) = append(x :: remove(a, l_1'), remove(a, l_2)) = x :: append(remove(a, l_1'), remove(a, l_2))$ (liniile 4 și 6 din `remove`, linia 4 din `append`).
   Cum cele două părți sunt egale, pentru $a \neq x$, proprietatea ține.

   (b) $a = x$: În stânga, $remove(a, append(x :: l_1', l2)) = remove(a, x :: append(l_1', l2)) = remove(a, append(l_1', l2))$ (linia 4 din `append`, liniile 4 și 5 din `remove`).
   În dreapta, $append(remove(a, x :: l_1'), remove(a, l_2)) = append(remove(a, l_1'), remove(a, l2))$. (liniile 4 și 5 din `remove`).
   Astfel, în baza ipotezei de inducție, cele două părți sunt egale și deci proprietatea ține pentru $a = x$.

   Astfel, am demonstrat că pentru orice listă l_1, proprietatea cerută este adevărată.

7 Se consideră o stivă implementată folosind un vector. Operațiile definite pentru stivă sunt *push* și *pop*. Astfel, când se adaugă în stivă un element printr-un *push*, și vectorul este plin, trebuie să se aloce un nou vector și să se copieze elementele vectorului vechi în cel nou, urmând ca cel nou să se folosească pe post de stivă.

1. Care este complexitatea operațiilor *push* și *pop* în cazul cel mai defavorabil?
   R. Considerând o stivă cu k elemente, în cazul cel mai defavorabil, operația *push* va determina o realocare a vectorului. Astfel, vor trebui mutate toate cele k elemente, ajungând rezultând o complex. Admițând că mutarea respectiv atribuirea unui element sunt operații elementare, pentru k elemente, în afară de mutarea elementelor, trebuie atribuit 1 element, pentru un total de $k+1$ operații și obținând $k+1$ elemente în stivă. Pentru o secvență de n operații, după pasul k având k elemente în stivă, numărul total de operații elementare va fi cel mult:

$$\sum_{k=1}^{n} cost_{push}(k) = \sum_{k=1}^{n} O(k) = O\left(\frac{n(n+1)}{2}\right) = O(n^2)$$

   Pentru o operație *pop*, presupunând că vectorul cu care se implementează stiva nu se redimensionează, $cost_{pop}(k) = \Theta(1)$, deoarece se poate decrementa dimensiunea efectivă a stivei (presupunând că este independentă de capacitatea vectorului).

2. Dacă dimensiunea vectorului se incrementează cu 1 când este necesar un vector mai mare, care este costul amortizat al unei operații pe stivă folosind metoda agregării?
R. Incrementarea dimensiunii cu 1 corespunde cazului în care fiecare inserare are costul exact k și astfel pentru o secvență de n operații *push*:

$$\sum_{k=1}^{n} cost_{push}(k) = \sum_{k=1}^{n} k = \frac{n(n+1)}{2} = \Theta(n^2)$$

Rezultă deci că $cost_{push} = \frac{\Theta(n^2)}{n} = \Theta(n)$, în timp ce pentru *pop*, costul este tot constant (ca mai sus), $cost_{pop} = \frac{n\Theta(1)}{n} = \Theta(1)$.

3. Dacă dimensiunea vectorului de dublează atunci când este necesar un vector mai mare, care este costul total pentru n operații și care este costul mediu al unei operații pe stivă folosind metoda agregării?
R. Considerând că începem cu un vector de dimensiune 1 și după pasul k vom avea k elemente în vector, observăm că trebuie dublat la pași de forma $k = 2^i + 1$, deci pentru pașii 2, 3, 5, 9, 16, ... Costul total la un pas k, va fi costul atribuirii în vector adunat cu costul mutării (dacă există). Atunci, pentru o secvență de n operații *push*:

$$\begin{aligned} C = \sum_{k=1}^{n} cost_{push}(k) = \sum_{k=1}^{n} (cost_{atribuire}(k) + cost_{mutare}(k)) = n + \sum_{i=0}^{\lfloor \log(n-1) \rfloor} 2^i \\ = n + 2^{\lfloor \log(n-1) \rfloor + 1} - 1 \leq n + 2^{\lfloor \log(n-1) \rfloor + 1} = n + 2(n-1) - 1 = 3n - 3 \\ C = \Theta(n) \end{aligned}$$

Atunci $\widehat{cost}_{push} = \frac{\Theta(n)}{n} = \Theta(1)$. Pentru o operație *pop*, dacă vectorul nu trebuie redimensionat când este prea gol, la fel ca anterior, $\widehat{cost}_{pop} = cost_{pop} = \frac{\Theta(n)}{n} = \Theta(1)$. Dacă vectorul ar trebui redimensionat, pentru o secvență de n *pop*-uri pentru un vector de dimensiune n, costul amortizal al operației *pop* ar fi tot $\Theta(1)$ prin analogie cu *push*.

8 Considerând un min-heap binar cu operațiile *insert* și *deleteMin*, se cere:

1. Care este complexitatea celor două operații în cazul cel mai defavorabil?
R. Cel mai folosit mod de a implementa un heap este un vector deoarece, deși s-ar putea face la fel și cu un arbore binar, s-ar pierde prea mult spațiu pentru cei doi pointeri pentru fiecare nod. Vom considera așadar că lucrăm cu un vector de dimensiune suficient de mare ca să nu se pună problemă dacă la o operație *insert* elementul încape în heap.
Implementările operațiilor folosesc două funcții pe care le vom numi *sink* și *swim* care împing în heap un nod cât mai jos, comparându-l cu succesorii săi din heap respectiv ridică un nod în heap, comparânu-l cu predecesorul său. Pentru a ilustra concret funcțiile, dăm următoarea implementare:

```
1  def insert(H, k):
2      H.append(k)
3      swim(H, len(H) - 1)
4
5  def deleteMin(H):
6      m = H[0]
7      H[0] = H[len(H) - 1]
8      del H[len(H) - 1]
9      sink(H, 0)
10     return m
11
12 def swim(H, i):
13     pred = (i - 1) / 2
14     if pred < 0:
15         return
16     if H[i] < H[pred]:
17         H[i], H[pred] = H[pred], H[i]
18         swim(H, pred)
19
20 def sink(H, i):
21     left = 2 * i + 1
22     right = 2 * i + 2
23     if right < len(H):
24         if H[i] > min(H[left], H[right]):
25             if H[left] < H[right]:
26                 H[i], H[left] = H[left], H[i]
27                 sink(H, left)
28             else:
29                 H[i], H[right] = H[right], H[i]
30                 sink(H, right)
31     elif left < len(H):
32         if H[i] > H[left]:
33             H[i], H[left] = H[left], H[i]
34             sink(H, left)
35
36 def makeHeap(H):
37     for i in xrange(len(H) / 2 - 1, -1, -1):
38         sink(H, i)
```

Pentru analiză admitem că `len(H)`, `H.append(k)`, `del H[len(H) - 1]`, atribuirile și comparațiile se fac în timp constant, $\Theta(1)$. Putem face aceste presupunere deoarce în general, listele Python folosesc array-uri redimensionabile [1] și deci adăugarea respectiv ștergerea unui element la final se fac în timp amortizat constant.
Să observăm că atât `insert(H, k)` cât și `deleteMin(H, k)` execută o serie constantă de operații (linia 2 și liniile 6-9 din cod) după care apelează `swim` respectiv `sink`. Vom analiza deci comportamentul acestor două funcții în continuare considerând că n este dimensiunea (numărul de elemente al) heap-ului și elementele sunt indexate de la $0 \to n-1$.

(a) *swim*: Corpul procedurii `swim` este, cu excepția apelului de funcție de la linia 18 constant, trecând de la i la $\frac{i-1}{2}$, cât timp i rămâne pozitiv. Deoarece la fiecare pas dimensiunea lui i se înjumătățește, putem considera că în cazul cel mai defavorabil $T(n) = T(\frac{n}{2}) + \Theta(1)$, de unde rezultă că $T(n) = \Theta(\log(n))$

(b) *sink* : Pentru a simplifica analiza, să presupunem că mereu coborâm spre stânga (sau, mereu $H[left] < H[right] < H[i]$). Atunci trecem de la i la $2i + 1$ cât timp $i < n$. Fiindcă începem cu $i = 0$, prin inducție este clar că $i = 2^q - 1$, $2(2^q - 1 + 1) - 1 = 2^{q+1} - 1$, iar ultimul i va fi de forma $2^{\lfloor \log n \rfloor} - 1$. Prin urmare și *sink* necesită în cazul cel mai defavorabil $\Theta(\log n)$ pași.

Astfel, în cazul cel mai rău, ambele operații au complexitatea $\Theta(\log n)$.

2. Definiți o funcție potențial pentru min-heap astfel încât costurile amortizate ale celor două operații să fie cât mai mici (iar $deleteMin$ să fie $O(1)$).
R. Dacă vrem un cost amortizat pentru $deleteMin$ de O(1) și funcția de potențial o notăm cu $\Phi(D_i) = \Phi(n)$, unde D_i e starea heap-ului la pasul i iar n este numărul de elemente din heap (e clar că funcția de potențial trebuie să depindă de n cumva), trebuie ca $\hat{c}_{deleteMin} = c_{deleteMin} + \Phi(n-1) - \Phi(n) = O(1)$. Dar știm că $c_{deleteMin} = \log n$ (pentru simplitate fără notația asimptotică) și atunci să definim $\Phi(n) = \sum_{k=1}^{n} \log k$ și să considerăm în plus că $\Phi(0) = 0$. Am ales această funcție doarece pentru un heap cu n noduri, înălțimea sa va fi $\log n$, și astfel funcția $\Phi(n)$ ar ține informații despre toate stările anterioare ale structurii.
Considerăm că starea structurii la pasul i, D_i este dată exclusiv de numărul de elemente din heap, deci $\Phi(D_i) = \Phi(n)$. Restricțiile care trebuie satisfăcute pentru ca Φ să fie funcție de potențial sunt:

(a) $\Phi(D_0) = 0$: D_0 este starea inițială a heap-ului – atunci când acesta este gol. Deoarece am stabilit că prin convenție $\Phi(0) = 0$, proprietatea este satisfăcută.

(b) $\Phi(D_i) \geq 0$, $\forall i$: este echivalent cu $\Phi(n) \geq 0$, pentru orice n, ceea ce este sigur fiindcă $\log n > 0$, $\forall n \in \mathbb{N}^*$ și $\Phi(n)$ este o sumă de cantități pozitive, deci este și ea pozitivă.

3. Demonstrați prin metoda potențialului care sunt costurile amortizate ale celor două operații.
R. Pentru $insert$, dimensiunea heap-ului crește cu 1, deci trecem de la $\Phi(n) \to \Phi(n+1)$, în timp ce pentru $deleteMin$, dimensiunea heap-ului scade cu 1, $\Phi(n) \to \Phi(n-1)$.

(a) $deleteMin$: $\hat{c}_{deleteMin} = c_{deleteMin} + \Phi(D_i) - \Phi(D_{i-1}) = \log n + \Phi(n-1) - \Phi(n) = \log n + \sum_{k=1}^{n-1} \log k - \sum_{k=1}^{n} \log k = \log n - \log n = 0 = O(1)$

(b) $insert$: $\hat{c}_{insert} = c_{insert} + \Phi(D_i) - \Phi(D_{i-1}) = \log n + \Phi(n) - \Phi(n-1) = \log n + \sum_{k=1}^{n} \log k - \sum_{k=1}^{n-1} \log k = \log n + \log n = 2 \log n = O(\log n)$

9 Se consideră o secvență de n operații ce se execută pe o structură de date. Operația i are costul i dacă $i = 2^q$, respectiv 1 altfel. Aplicați metoda agregării și metoda potențialului pentru a determina costul amortizat.
R. Notăm costul unei operații i, c_i iar costul ei amortizat $\hat{c}_i$.

1. Metoda agregării: Pentru o secvență de n operații, costul total va fi:

$$C = \sum_{k=1}^{n} c_i = \sum_{k=1}^{n} 1 + \sum_{i=0}^{\lfloor \log n \rfloor} (2^i - 1) = n + 2^{\lfloor \log n \rfloor + 1} - 1 + \lfloor \log n \rfloor + 1 = 3n + \lfloor \log n \rfloor \leq 3n$$

$$\Rightarrow \hat{c}_i = \frac{3n}{n} = 3 = \Theta(1)$$

2. Metoda potențialului: Alegem funcția de potențial $\Phi(D_i) = 2i - 2^{\lfloor \log i \rfloor + 1}$. Considerăm prin convenție că $\log 0 = 0$ și astfel $\Phi(D_0) = 0$. În plus, deoarece $\lfloor \log i \rfloor \leq \log i \Rightarrow 2^{\lfloor \log i \rfloor} \leq 2^{\log i} = i$ și deci $-2^{\lfloor \log i \rfloor + 1} \geq -2i \Rightarrow \Phi(D_i) \geq 0$. Cum costul operației i depinde de i:

   (a) $i = 2^q \Rightarrow \lfloor \log 2^q \rfloor = q$ și $\lfloor \log(2^q - 1) \rfloor = q - 1$, deci $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = i + 2i - 2^{q+1} - 2(i-1) + 2^{q-1+1} = i + 2i - 2i - 2i + 2 + i = 2 = \Theta(1)$

   (b) $i \neq 2^q \Rightarrow \lfloor \log 2^q \rfloor = \lfloor \log(2^q - 1) \rfloor$ și deci $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 2i - 2(i-1) = 3 = \Theta(1)$

În concluzie, costul amortizat al unei operații c_i este $\Theta(1)$, pentru orice i.

Bibliografie

[1] *Performance Notes*, `http://effbot.org/zone/python-list.htm`