Estructures de la Informació. Control parcial 1

Dept. de Llenguatges i Sistemes Informàtics
E.P.S.E.V.G., 26 d'octubre de 2011, 8:30-10:45

IMPORTANT: Resol els problemes en fulls separats.

1. (2 punts. 20 minuts) **Arbres**

   a) Implementa la funció:

   ```
   nat fulles (const Abin<T> &a)
   ```

   que, donat un arbre binari, retorni el nombre de fulles (subarbres buits) que té. Disposes de les classes `Abin<T>` i `Abin<T>::iterador` amb els mètodes públics vistos a classe i llistats a continuació.

   b) Indica el tipus de recorregut utilitzat i el cost que té en funció del número de nodes de l'arbre binari.

   NOTA: Pots crear els mètodes addicionals que creguis necessaris.

```
template <class T>
class Abin {
 public:
   Abin() throw(error);

   Abin(const Abin<T> &a) throw(error);
   Abin& operator=(const Abin<T> &a) throw(error);
   ~Abin() throw();

   Abin(Abin<T> &fesq, const T &x, Abin<T> &fdret) throw(error);

   bool es_buit() const throw();

   Iterador sobre arbres binaris.
   friend class iterador {
   public:
     friend class Abin;

     iterador() throw();

     Abin<T> arbre() const throw(error);


     Retorna l'element apuntat per l'iterador.
     const T& operator*() const throw(error);

     iterador fesq() const throw(error);
     iterador fdret() const throw(error);

     bool operator==(const iterator &it) const throw();
     bool operator!=(const iterator &it) const throw();

     static const int IteradorInvalid = 410;
   };

   iterador arrel() const throw();
   iterador final() const throw();
  };
```

2. (3,5 punts. 55 minuts) **Estructures Lineals en memòria dinàmica**

a) Implementa el mètode constructor per defecte i el destructor de la següent classe llista:

```
template <class T>
class llista {
 public:
   ...
   Crea una nova llista que conté només els elements que es troben en posicions
   múltiples de k.
   Pre: {k > 0}
    llista compacta_k (nat k) const throw(error);
    ...
 private:
  // Llista simplement enllaçada amb element fantasma.
  struct node {
    T info;
    node *seg;
  };
  node* _head;
  int _size;
};
```

b) Fes el mètode `compacta_k` de l'anterior classe llista. Per exemple, donada la llista d'enters L = {21, 3, 14, 1, 73, 9, 108}:

la crida a `L.compacta_k(2)` retornaria {3, 1, 9}

la crida a `L.compacta_k(3)` retornaria {14, 9}

la crida a `L.compacta_k(4)` retornaria {1}

NOTA 1: Pots crear els mètodes addicionals que creguis necessaris.

NOTA 2: Pots suposar que la classe T disposarà de l'operador assignació.

3. (2,5 punts. 40 minuts) **Estructures Lineals en vector**

Implementa els mètodes `encua(x, n)` i `desencua(n)` que permeten encuar l'element *x* *n* vegades al final de la cua i desencuar *n* elements del principi de la cua respectivament. Codifica'ls com a mètodes de la classe `cua<T>` sabent que aquesta classe està implementada amb un **vector circular**.

```
template <class T>
class cua {
 public:
   ...
   Encua n vegades l'element x al final de la cua. Si la cua té menys de n
   espais lliures, encua tots els que són possibles fins a deixar-la plena.
   void encua(const T &x, nat n) throw(error);

   Desencua n elements del principi de la cua. Si la cua té menys de n
   elements els desencua tots.
   void desencua(nat n) throw();
   ...
   static const nat MAX = 100;

 private:
    T _A[MAX];   // Vector circular
    nat _prim;   // Apunta al primer element
    nat _pl;     // Apunta a la primera posició lliure
    nat _cnt;    // Nombre d'elements
};
```

4. (2 punts. 20 minuts) **Eficiència algorísmica**

Donat el següent algorisme:

```
bool nose(float A[], int i, int j, float x) {
  if (i == j) {
    return A[i]==x;
  }
  int p = i + (j - i + 1)/3;
  int q = p + (j - i + 1)/3;
  if (x < A[p]) {
    return nose(A, i, p-1, x);
  }
  else if (x > A[q]) {
    return nose(A, q+1, j, x);
  }
  else {
    return nose(A, p, q, x);
  }
}
```

a) Quin és el cost de l'algorisme `nose` en funció de la talla *n=j-i+1*?
Dóna l'ordre de creixement asimptòtic d'aquest cost en termes de la notació Θ. Si és necessari, pots utilitzar els teoremes adjunts.

b) Per què creus que serveix aquest algorisme? Consideres que és un bon algorisme (és millor o pitjor que altres algorismes equivalents)?

IMPORTANT: Raona la teva resposta.

Decreixement aritmètic

Teorema. *Sigui T(n) el cost d'un algorisme recursiu descrit per la recurrència*

$$T(n)=\begin{cases} f(n) & si\ 0\leqslant n<n_0 \\ a\cdot T(n-c)+g(n) & si\ n_0\leqslant n \end{cases}$$

on n_0 és una constant, $c\geqslant 1$, $f(n)$ és una funció arbitrària i $g(n)=\Theta\left(n^k\right)$ amb *k* constant. Llavors,

$$T(n)=\begin{cases} \Theta\left(n^k\right) & si\ a<1 \\ \Theta\left(n^{k+1}\right) & si\ a=1 \\ \Theta\left(a^{n/c}\right) & si\ a>1 \end{cases}$$

Decreixement geomètric

Teorema. *Sigui T(n) el cost d'un algorisme recursiu descrit per la recurrència*

$$T(n)=\begin{cases} f(n) & si\ 0\leqslant n<n_0 \\ a\cdot T(n/b)+g(n) & si\ n_0\leqslant n \end{cases}$$

on n_0 és una constant, $b>1$, $f(n)$ és una funció arbitrària i $g(n)=\Theta\left(n^k\right)$ amb *k* constant. Llavors,

$$T(n)=\begin{cases} \Theta\left(n^k\right) & si\ a<b^k \\ \Theta\left(n^k\log n\right) & si\ a=b^k \\ \Theta\left(n^{\log_b a}\right) & si\ a>b^k \end{cases}$$

Solució problema 1

a)

```
template <typename T>
nat fulles (const Abin<T> &a) {
  return fulles(a.arrel(), a.final());
}

template <typename T>
nat fulles (Abin<T>::iterator it, Abin<T>::iterator end) {
  nat f;
  if (it == end) { // Subarbre buit
    f = 1;
  }
  else {
    f = fulles(it.fesq(), end) + fulles(it.fdret(), end);
  }
  return f;
}
```

Una altra manera de fer-ho és tenin en compte que en un arbre binari es compleix que:

#fulles = #nodes + 1

```
template <typename T>
nat fulles (const Abin<T> &a) {
  return nodes(a.arrel(), a.final()) + 1;
}

template <typename T>
nat nodes (Abin<T>::iterator it, Abin<T>::iterator end) {
  nat n = 0;
  if (it != end) {
    n = 1 + nodes(it.fesq(), end) + nodes(it.fdret(), end);
  }
  return n;
}
```

b) Ambdós casos hem fet un recorregut en preordre (primer visitar l'arrel i després els fills recursivament. El cost és lineal doncs visitem tots els elements de l'arbre una vegada.

Solució problema 2

a)

```
template <class T>
llista<T>::llista() throw(error) {
  _head = new node;
  _head->seg = NULL;
  _size = 0;
}

template <class T>
llista<T>::~llista() throw() {
  while (_head != NULL) {
    node* aux = _head;
    _head = _head->seg;
    delete aux;
  }
}
```

b)

```
// Mètode privat de classe.
// Caldria declarar-lo com a static dins de la representació de la classe.
template <class T>
typename llista<T>::node* llista<T>::crea_node(node* n) throw(error) {
  node* n2 = new node;
  try {
    n2->info = n->info;
  }
  catch (error) {
    delete n2;
    throw();
  }
  n2->seg = NULL;
  return n2;
}


template <class T>
llista<T> llista<T>::compacta_k(nat k) const throw(error) {
  llista<T> l;
  node* ant = l._head;
  node* n = _head->seg;
  nat i = 1;
  while (n != NULL) {
    if (i % k == 0) {
      ant->seg = crea_node(n);
      ant = ant->seg;
      ++l._size;
    }
    n = n->seg;
    i = i+1;
  }
  return l;
}
```

Solució problema 3

```
template <class T>
void cua<T>::encua(const T &x, nat n) throw(error) {
  for (nat i=0; i<n and _cnt<MAX; i++) {
    _A[_pl] = x;
    _pl = (_pl+1) % MAX;
    ++_cnt;
  }
}

template <class T>
void cua<T>::desencua(nat n) throw() {
  for (nat i=0; i<n and _cnt>0; i++) {
    _prim = (_prim+1) % MAX;
    --_cnt;
  }
}
```

Solució problema 4

a) Càlcul del cost temporal de la funció `nose()`:

Per resoldre el cost de la recursivitat de l'acció `nose()` utilitzarem les equacions de recurrència pel cas de decreixement geomètric.

$$T(n)=\begin{cases} ctt & si\ 0\leqslant n<2 \\ a\cdot T(n/b)+\Theta(n^k) & si\ 2\leqslant n \end{cases}$$

on a=1, b=3, k=0.

Llavors es compleix que $a=b^k$ i, per tant, el cost temporal de `nose()` és
$T_{nose}(n) = \Theta(n^k \log n) = \Theta(\log n)$

b) L'algorisme pot servir per cercar un element *x* en el vector *A* si els seus elements estan ordenats de forma ascendent. És una variació de la cerca dicotòmica, en aquest cas parlaríem de cerca tricotòmica. El cost és el mateix (logarítmic) que el de la cerca dicotòmica i menor que el de la cerca seqüencial, per tant és un bon algorisme de cerca de vectors ordenats.