Estructures de la Informació. Control parcial

Dept. de Ciències de la Computació

E.P.S.E.V.G., 7 de novembre de 2014, 12:30-14:30

IMPORTANT: Resol els problemes en fulls separats.

1. (4 punts. 60 minuts) **Estructures Lineals**

   Donada una classe llista genèrica que emmagatzema una llista doblement encadenada, no circular i amb element fantasma:

   a) Escriu la representació d'aquesta classe.

   b) Implementa un mètode d'aquesta classe que faci que cada element diferent de la llista aparegui a la llista *n* vegades (on $n \geq 0$). Per exemple,
      L = [2, 2, 1, 3, 2, 5, 1] i n=3, el resultat seria L = [2, 2, 2, 1, 1, 1, 3, 3, 3, 5, 5, 5]
      L = [2, 2, 1, 3, 2, 5, 1] i n=1, el resultat seria L = [2, 1, 3, 5]

      Pots considerar que el tipus genèric del que està format la llista disposa de l'operador d'igualtat, el constructor per còpia i l'operador d'assignació. El resultat no necessàriament ha de mantenir l'ordre original dels elements.

   NOTA: Cal que implementis els mètodes addicionals que utilitzis explícitament.

2. (4 punts. 40 minuts) **Diccionaris**

   Donada aquesta classe `dicc` implementada com un arbre binari de cerca, la representació de la qual és la següent:

```
template <typename C, typename V>
class dicc {
  ...
  private:
    struct node {
      node *fesq;
      node *fdret;
      C clau;
      V valor;
    };
    node* _arrel;
    ...
};
```

   fes un mètode d'aquesta classe que donades dues claus, retorni una llista amb els valors associats a les claus que siguin més grans que la primera clau i més petites que la segona clau. La classe llista és la classe `list` de la biblioteca STL.

   NOTA 1: Suposa que el tipus C disposa dels operadors de comparació i el tipus V el constructor per còpia i operador assignació.

   NOTA 2: Cal que implementis els mètodes addicionals que utilitzis explícitament.

3. (2 punts. 20 minuts) **Eficiència algorísmica**

Donat el següent codi, calcula l'ordre de creixement asimptòtic del cost en el cas pitjor en termes de la notació Θ de manera **raonada**. Si és necessari, pots utilitzar els teoremes adjunts.

```
void f(int i, int j) {
  if (i < j) {
    int m = (i + j) / 2;
    g(j - i + 1);
    f(i, m);
    f(m + 1, j);
  }
}

void g(int n) {
  int *b = new int[n];          // Cost(new[]): Θ(1)
  for (int i = 0; i < n; ++i) {
    b[i] = random(5);
    for (int j = 0; j < n; ++j) {
      for (int k= j-b[i]; k <= j+b[i]; ++k) {
        fun(j, k);
      }
    }
  }
  delete[] b;                   // Cost(delete[]): Θ(1)
}
```

En concret:

a) Quin és el cost en el cas pitjor de la funció `g` respecte de *n*? Assumeix que les crides a la funció `random(a)`, que retorna un número enter a l'atzar entre 0 i *a*, i la funció `fun(a,b)` tenen cost constant.

b) Quin és el cost en el cas pitjor de la crida a `f(1,n)` en funció de *n*?

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 $g(n)=\Theta(n^k)$ amb *k* constant. Llavors,

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

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 $g(n)=\Theta(n^k)$ amb *k* constant. Llavors,

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

Solució problema 1

a)

```
struct node {
  T info;
  node *seg;
  node *ant;
};

node *_head; // apunta al fantasma
```

b)

```
template <typename T>
void llista<T>::repeteix_diferents(int num) throw(error) {
  node *n = _head->seg;
  while (n != NULL) {
    repeteix(n, num);
    n = n->seg;
  }
}


template <typename T>
void llista<T>::repeteix(node *n, int num) throw(error) {
  node *ant = n->ant;
  node *p = n;
  int k = 0;
  while (p != NULL) {
    if (p->info == n->info) {
      ++k;

      // desencadenem el node de la llista
      node *aux = p;
      p = p->seg;
      aux->ant->seg = aux->seg;
      if (aux->seg != NULL) {
        aux->seg->ant = aux->ant;
      }

      if (k > num) {
        // esborrem el node
        delete aux;
      }
      else {
        // movem el node al principi
        inserir_node(aux, ant);
      }
    }
    else {
      p = p->seg;
    }
  } // end while
```

```
  // repetim els nodes que falten per arribar a num
  while (k < num) {
    node *nou = new node(n->info);
    inserir_node(nou, ant);
    ++k;
  }
}


template <typename T>
void llista<T>::inserir_node(node *qui, node *on) throw() {
  qui->seg = on->seg;
  if (on->seg != NULL) {
    on->seg->ant = qui;
  }
  qui->ant = on;
  on->seg = qui;
}


template <typename T>
llista<T>::node::node(const T &valor) throw(error)
      : info(valor), seg(NULL), ant(NULL) {
}
```

Solució problema 2

```
template <typename C, typename V>
void dicc<Clau, Valor>::cerca_rang(const C &a, const C &b,
                                  list<V> &lst) throw(error) {
  cerca_rang(_arrel, a, b, lst);
}
```

Per tal d'implementar `cerca_rang` es proposen dues opcions. La segona opció és la millor.

```
// Opció poc eficient i que no aprofita el fet que la classe estigui
// implementada com un abc
template <typename C, typename V>
void dicc<Clau, Valor>::cerca_rang(node* n, const C &a, const C &b,
                                  list<V> &lst) throw(error) {
  if (n != NULL) {
    if (n->clau > a and n->clau < b) {
      lst.push_back(n->valor);
    }
    cerca_rang(n->fesq, a, b, lst);
    cerca_rang(n->dret, a, b, lst);
  }
}


// Opció molt més bona
template <typename C, typename V>
void dicc<Clau, Valor>::cerca_rang(node* n, const C &a, const C &b,
                                  list<V> &lst) throw(error) {
  if (n != NULL) {
    if (n->clau > a and n->clau < b) {
      lst.push_back(n->valor);
    }
    if (n->clau > a) {
      cerca_rang(n->fesq, a, b, lst);
    }
    if (n->clau < b) {
      cerca_rang(n->dret, a, b, lst);
    }
  }
}
```

Solució problema 3

a)
<u>Càlcul del cost temporal de l'acció `g()`:</u>

```
for (int k= i-b[i]; k <= i+b[i]; ++j)
  fun(i, j);
```

Té cost constant $\Theta(1)$ doncs, com que b[i] conté un enter entres 0 i 5, s'executa com a molt 11 vegades un codi de cost constant: $11 \cdot \Theta(1) \in \Theta(1)$

```
for (int j = 0; j < n; ++j)
  for (int k= i-b[i]; k <= i+b[i]; ++k)
    fun(i, j);
```

Té cost lineal $\Theta(n)$ doncs s'executa *n* vegades un codi de cost constant.

```
for (int i = 0; i < n; ++i)
  b[i] = random(5);
  for (int j = 0; j < n; ++j)
    for (int k= j-b[i]; k <= j+b[i]; ++k)
      fun(j, k);
```

Té cost lineal $\Theta(n^2)$ doncs s'executa *n* vegades un doble bucle que té cost *n*.
La instrucció `b[i]=random(5);` no afecta atès que té cost constant.

Per tant, l'acció `g()` té cost lineal $\Theta(n^2)$.

b)
<u>Càlcul del cost temporal de l'acció `f()`:</u>

Per resoldre el cost de la recursivitat de l'acció `f()` 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=2, b=2, k=2.

Llavors es compleix que $a<b^k$ i, per tant, el cost temporal de `f()` és $T_f(n) = \Theta(n^2)$