Estructura de la Informació. Control parcial

Dept. de Ciències de la Computació

E.P.S.E.V.G., 28 d'octubre de 2016, 12:30-14:30

IMPORTANT: Resol els problemes en fulls separats.

1. (2 punts. 20 minuts) **Eficiència algorísmica**

   Donat el següent codi:

```
node* func(node* n,  int r) {
  // Pre: n!=NULL i r>0
  node *ret;
  if (r > 1) {
    node *aux = new node;
    aux->info = n->info;
    aux->seg = n->seg;
    ret = func(aux, r-1);
    n->seg = aux;
  }
  else ret = n;
  return ret;
}

void proc(node *n, int a, int p, int r) {
  // Pre: a > 0, p > 0 i r > 0
  if (n != NULL) {
    node *n2 = n;
    if (a % p == 0) n2 = func(n, r);
    proc(n2->seg, a+1, p, r);
  }
}
```

   Calcula el cost asimptòtic de l'acció `proc` en funció del nombre d'elements de L, tenint en compte que la crida inicial és `proc(L, 1, 2, 3)`. 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,

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

2. (4 punts. 55 minuts) **Estructures Lineals**

Donada una classe llista d'enters implementada amb nodes doblement encadenats amb element fantasma i no circular. Fes el següent:

a) Escriu la representació d'aquesta classe.

b) Implementa el mètode `encripta` de la classe llista, que donada una cadena de caràcters i un enter *k* guarda en la llista la següent informació:

1. El node fantasma guarda el valor de *k*.
2. En el primer node de la llista es guarda dues vegades la codificació ASCII del primer caràcter de l'string per *k*: $ascii_0 * 2 * k$
3. En el node i-èssim es guarda la codificació ASCII del caràcter *i* sumat a la codificació ASCII del caràcter *i-1* multiplicat per *k*: $(ascii_i + ascii_{i-1}) * k$

Per exemple, si la cadena és «Lorem ipsum» i *k* val 5 la llista resultant seria:

L = [760, 935, 1125, 1075, 1050, 705, 685, 1085, 1135, 1160, 1130]

c) Implementa el mètode `desencripta` de la classe llista, que retorna el missatge que té emmagatzemat la llista fent el procés invers descrit a l'apartat anterior. La llista es buida un cop el missatge s'ha desencriptat.

NOTA 1: Tingues en compte que la llista inicialment pot tenir elements.

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

3. (4 punts. 45 minuts) **Arbres i diccionaris**

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

```
template <typename T>
class abin {
  private:
    struct node {
      node *fesq;
      node *fdret;
      T info;
    };
    node* _arrel;
    int _sz;
    ...
};
```

a) Fes un mètode d'aquesta classe que indiqui quina és la mediana de tot l'arbre (si ordenem les dades és aquell valor tal que té igual nombre de dades per sota i per sobre).

b) Indica quin és el cost asimptòtic del teu algorisme. Hi ha alguna manera de millorar aquest cost? Indica raonadament com ho faries.

NOTA 1: Suposa que el tipus T disposa dels operadors de comparació.

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

Solució problema 1

Per calcular el cost asimptòtic de l'acció `proc` primer cal calcular el cost de `func`

Càlcul del cost temporal de la funció `func()`:

Atès que el nombre de crides que es fan de `func` no depèn del nombre d'elements de la llista (sinó que està relacionat amb un altre paràmetre), el cost d'aquesta funció és constant.
En la crida exemple on r=3, es farien dues crides de `func` que és independent del nombre d'elements que tingui la llista.

Càlcul del cost temporal de l'acció `proc()`:

`proc` és una acció recursiva l'equació de la recurrència de la qual és la següent:

$$T(n)=\begin{cases} ctt & si\ 0\leqslant n<1 \\ a\cdot T(n-c)+\Theta(n^k) & si\ 1\leqslant n \end{cases}$$

En aquesta equació *a*=1 ja que hi ha una crida recursives dins la mateixa acció; *c*=1 ja que la disminució de les dades és en un; *k*=0 ja que els cost de les altres operacions que hi ha dins d'aquesta acció són constants.

Per tant, l'equació de la recurrència és:

$$T(n)=\begin{cases} ctt & si\ 0\leqslant n<1 \\ T(n-1)+\Theta(1) & si\ 1\leqslant n \end{cases}$$

Per resoldre el cost de `proc()` es pot usar el teorema mestre de decreixement aritmètic (ja que l'equació de recurrència s'adapta correctament a aquest teorema mestre).

Llavors atès que *a=1* i aplicant el teorema mestre el cost temporal de `proc()` és:

$$\Theta(n^{k+1})=\Theta(n^{0+1})=\Theta(n)$$

Solució problema 2

a)

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

node *_head;   // apunta a l'element fantasma de la llista
int _sz;
```

b)

```
void llista::encripta(const string &s, int k) throw(error) {
  node *inici = new node;
  inici->seg = NULL;
  inici->ant = NULL;
  if (s.size() > 0) {
    try {
      node *p = new node;
    }
    catch (error) {
      delete inici;
    }
    p->info = int(s[0])*int(s[0])*k;
    p->seg = NULL;
    p->ant = inici;
    inici->seg = p;
    for (unsigned int i = 1; i < i.size(); ++i) {
      try {
        node *q = new node;
      }
      catch (error) {
        esborra(inici);
        throw;
      }
      q->info = int(s[i])*int(s[i-1])*k;
      q->seg = NULL;
      q->ant = inici;
      p->seg = q;
      p = q;
    }
  }
  esborra(_head);
  _head = inici;
  inici = NULL;
  _sz = s.size();
  _head->info = n;
}

// mètode privat
void llista::esborra(node *n) throw() {
  if (n != NULL) {
    esborra(n->seg);
    delete n;
  }
}
```

c)

```
void llista::desencripta(string &s) throw(error) {
  s = "";
  if (_head->seg != NULL) {
    node *n = _head->seg;
    int k = _head->info;

    // tractament primer element
    int elem = n->info/(k*2);
    s += char(elem);
    n = n->seg;
    delete n->ant;
    while (n != NULL) {
      elem = n->info/k-elem;
      s += char(elem);
      n = n->seg;
      delete n->ant;
    }
  }
  _head->seg = NULL;
}
```

Solució problema 3

a)

```
template <typename T>
T abin<T>::mediana () const throw(error) {
  // Si l'abin és buit cal generar un error ja que no es pot calcular la mediana
  if (_arrel == NULL) throw error(ArbreBuit);
  return mediana(_arrel)->info;
}


template <typename T>
typename abin<T>::node* abin<T>::mediana (node *n) const throw() {
  node* res = NULL;
  if (n != NULL) {
    int men = menors(n->info, _arrel);
    int maj = majors(n->info, _arrel);
    if ((_sz%2 != 0 and men == maj) or (_sz%2 == 0 and men == maj-1))
      res = n;
    else {
      res = mediana(n->fesq);
      if (res != NULL) res = mediana(n->fdret);
    }
  }
  return res;
}


template <typename T>
int abin<T>::menors (const T &t, node *on) const throw() {
  int res = 0;
  if (on != NULL) {
    if (on->info < t) {
      res = menors(t, on->fesq) + menors(t, on->fdret) + 1;
    }
    else {
      res = menors(t, on->fesq);
    }
  }
  return res;
}


template <typename T>
int abin<T>::majors (const T &t, node *on) const throw() {
  int res = 0;
  if (on != NULL) {
    if (on->info > t) {
      res = majors(t, on->fesq) + majors(t, on->fdret) + 1;
    }
    else {
      res = majors(t, on->fdret);
    }
  }
  return res;
}
```

b)

El cost d'aquest algorisme és $O(n^2)$ en el cas pitjor, ja que per cada node hem de visitar gairebé tot l'arbre.

Es podria millorar el cost si en cada node tinguéssim precalculat el nombre de nodes que hi pengen (mida del subarbre). D'aquesta manera per saber si és el node conté la mediana només caldria mirar la mida de fills esquerra i dret, i en cas contrari visitar el fill esquerre o dret. Això ens permetria no haver de calcular el nombre d'elements menors i majors.