Parcial 19/04/2006

1. Considereu un TAD diccionari implementat amb un arbre binari de cerca. Com s'ha vist a classe, el cas pitjor de les operacions de `cercar`, `inserir` i `esborrar` un element es dóna quan l'arbre és equivalent a una llista. Una possibilitat d'evitar aquest cas pitjor, suposant que coneixem a priori el conjunt sencer de claus del diccionari, és construir l'arbre binari de cerca com s'explica tot seguit.

   - Posar a l'arrel de l'arbre l'element amb clau igual a la mitjana del conjunt de claus.
   - Construir recursivament, mitjançant el mateix procediment (és clar), els subarbres esquerre i dret.

   Suposant què es disposa d'una funció que torna la mitjana d'un conjunt de n claus en temps $g(n)$, es demana:

   (a) Donar una recurrència que descrigui el cost del procediment anterior quan es construeix un arbre binari de cerca de n elements. Justificar la proposta.

   (b) Donar el cost en el cas pitjor, millor i mitjà de construir un arbre binari de cerca amb aquest procediment quan $g(n) = \Theta(n)$.

   (c) Donar el cost en el cas pitjor de cercar un element en un arbre construït d'aquesta manera. Justificar la resposta.

   (d) Donat un arbre construït d'aquesta manera, descriure detalladament com es podria inserir un nou element.

   (e) Quines avantatges i quins inconvenients té fer servir aquest nou procediment per construir arbres binaris de cerca?

2. Un arbre *quaternari* de cerca és una estructura de dades que permet implementar un diccionari que contingui claus bidimensionals, és a dir, una clau és un parell ordenat de claus (x, y).

   (a) Donar una implementació per aquesta nova estructura sabent que cada node de l'arbre quaternari té associats: una clau bidimensional (x, y), i 4 fills, tots arbres quaternaris, i anomenats *sud-est* `-SE-`, *sud-oest* `-SO-`, *nord-est* `-NE-` i *nord-oest* `-NO-` tals que:

      - un element amb clau (x_1, y_1) es troba al subarbre `SE` si $x_1 > x$ i $y_1 < y$,
      - un element amb clau (x_1, y_1) es troba al subarbre `SO` si $x_1 < x$ i $y_1 < y$,
      - un element amb clau (x_1, y_1) es troba al subarbre `NE` si $x_1 > x$ i $y_1 > y$ i
      - un element amb clau (x_1, y_1) es troba al subarbre `NO` si $x_1 < x$ i $y_1 > y$.

   (b) Explicar detalladament, sense donar codi, com es fa la inserció d'un nou element (x, y) a l'arbre quaternari.

   (c) Codificar en C++ l'operació de `cercar` un element a l'arbre quaternari.

   (d) Volem obtenir una llista de tots els elements presents a l'arbre quaternari. Explicar detalladament, sense donar codi, el procediment adient.

   (e) Modificar la resposta de l'apartat anterior per a que la llista de claus estigui ordenada (Criteri d'ordenació: $p_1 = (x_1, y_1)$ és menor que $p_2 = (x_2, y_2)$ si $x_1 < x_2$ o si ($x_1 = x_2$ i $y_1 < y_2$)).

3. Sigui $G = (V, E)$ un graf no dirigit i connex.

(a) Demostrar que G té, com a mínim, un vèrtex u tal que si esborrem u i totes les arestes que incideixen a u, el graf resultant també és connex.

(b) Dissenyar i codificar en C++ un algorisme que donat un graf no dirigit i connex, retorni un vèrtex que satisfaci les condicions de l'apartat anterior. Es poden fer servir les definicions de grafs vistes a classe:

```
typedef vector< list<int> > graf;
typedef list<int>::iterator arc;

#define forall_adj(uv,L) for (arc uv=(L).begin(); uv!=(L).end(); ++uv)
#define forall_ver(u,G) for (int u=0; u<(G).size(); ++u)
```

(c) Argumentar la correctesa de la solució obtinguda i

(d) Calcular el seu cost per a les dues implementacions (matrius i llistes).

4. Sigui el vector $S = [5, 9, 12, 10, 9, 3, 1, 5, 4]$ indexat de 0 a 8. Construïu un max-heap aplicant el procès de construcció de baix cap a dalt. Cost?

5. Digueu si és certa o falsa i per què la següent afirmació: "Determinar si un graf no dirigit conté cicles o no té cost $\Theta(n)$ en el cas pitjor".

6. *Definició*: Un graf dirigit i acíclic és un arbre arrelat si i només si tot vèrtex, excepte l'arrel, té un únic predecessor i només hi ha un vèrtex, l'arrel, amb grau d'entrada 0. És correcta aquesta definició? Per què?

7. Suposeu que heu de fer una pràctica de programació en la qual necessiteu implementar un algorisme d'ordenació que trigui temps $\Theta(n \log n)$ en cas pitjor. Malauradament, els ordinadors disponibles són molt antics i tenen molt molt poc espai de disc i memòria disponibles. Quin algorisme d'ordenació implementarieu? Per què?

8. Raoneu sobre la certesa o falsedat del següent enunciat: Per determinar si dos arbres binaris de cerca són idèntics, podríem fer un recorregut en inordre per tots dos i comparar les llistes resultants.

Solucions Parcial 19/04/06

1. (a) Si el diccionari és buit el cost de construir l'arbre binari de cerca corresponent és $\Theta(1)$. En canvi, si el diccionari no és buit amb cost $g(n)$ es busca l'element amb clau mediana i es posa a l'arrel amb cost $\Theta(1)$. Amb cost $\Theta(n)$ se separen els $(n-1)/2$ elements del subarbre esquerre dels del dret i amb cost $T((n-1)/2)$ es construeixen recursivament tots dos subarbres. Per tant, la recurrència és la següent:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 0 \\ 2T(n/2) + \Theta(n) + g(n) & \text{si } n > 0 \end{cases}$$

(b) El cost d'aquest procediment no depèn de la configuració de les dades d'entrada per tant el seu cost en cas millor, pitjor i mitjà només pot variar quan varia el cost $g(n)$ de la funció que cerca la clau mediana, però es suposa què aquesta funció té cost $\Theta(n)$ en tots els casos. Sustituïnt aquest cost dins la recurrència i resolent-la mitjançant el teorema mestre obtenim un cost de $\Theta(n \log n)$.

(c) Un arbre binari de cerca de tamany n construït amb aquest procediment serà un arbre binari balancejat amb alçada $\Theta(\log n)$ que és el cost en el cas pitjor de cercar-hi un element.

(d) El procediment anterior suposa que totes les claus es coneixen a priori. Per tant, per inserir un nou element seria necessari reconstruir l'arbre sencer.

(e) El principal avantatge d'aquest procediment és que produeix arbres binaris de cerca balancejats en els què el cost de cercar elements és óptim (per arbres binaris). L'inconvenient és justament que les claus s'han de conèixer a priori i per tant no és un procediment útil en aplicacions que requereixen un gran nombre d'insercions i esborrats.

2. (a) L'implementació demanada és:

```
template <typename Clau, typename Info>
class ArbreQ {
  struct Node {
    Clau x;
    Clau y;
    Info info;
    Node* saSE;
    Node* saSO;
    Node* saNO;
    Node* saNE;

    Node (Clau& x1, Clau& y1, Info& i, Node* f1, Node* f2, Node* f3, Node* f4)
      : x(x1), y(y1), info(i), saSE(f1), saSO(f2), saNO(f3), saNE(f4){}
  };
  int n;          // nombre d'elements de l'arbreQ
  Node* arrel;    // punter a l'arrel de l'arbreQ
}
```

(b) La inserció és fa com en un arbre binari de cerca (ABC) però tenint en compte que un node té 4 subarbres i que cal comparar la clau que es vol inserir amb la clau que ocupa l'arrel per decidir en quin subarbre s'ha d'inserir la nova clau. Serà un algorisme recursiu que rebrà un punter, *paq*, a l'arrel de l'arbre quaternari i la clau $< x, y >$ a inserir. Si el punter és nul vol dir que la clau no s'hi troba al arbre i, per tant, s'haurà de crear un nou node i incrementar el nombre d'elements de l'arbre. Si el punter no és nul llavors pot passar que l'arrel del arbre coincideixi amb la clau a inserir, i en aquest cas només cal modificar la informació del node, però si la clau de l'arrel no és la que volem inserir, aleshores s'ha d'escollir en quin fill s'ha de fer la inserció. Una possible codificació seria:

```
void insercio(Node*& paq, Clau& x1, Clau& y1, Info i) {
  if (paq) {
    Clau x=paq->x;
    Clau y=paq->y;
    if ((x<x1) && (y<y1)) {insercio(paq->saNE, x1, y1, i);}
    else if ((x<x1) && (y>y1)){insercio(paq->saSE, x1, y1, i);}
    else if ((x>x1) && (y<y1)){insercio(paq->saSO, x1, y1, i);}
```

```
    else if ((x>x1) && (y>y1)){insercio(paq->saNO, x1, y1, i);}
    else { paq->info = i;}
  } else {
    paq = new Node(x1, y1, i, 0, 0, 0, 0 );
    ++n;
  }
}
```

(c) S'ha decidit que la cerca torni un punter al node però es podia haver triat retornar un booleà.

```
static Node* cerca(Node* aq, Clau& x1, Clau& y1) {
  if (aq) {
    Clau x=aq->x;
    Clau y=aq->y;
    if ((x<x1) && (y<y1)) { return cerca(aq->saNE, x1, y1);}
    else if ((x<x1) && (y>y1)) { return cerca(aq->saSE, x1, y1);}
    else if ((x>x1) && (y<y1)) { return cerca(aq->saNO, x1, y1);}
    else if ((x>x1) && (y>y1)) { return cerca(aq->saSO, x1, y1);}
  }
  return aq;
}
```

(d) N'hi ha prou amb fer un recorregut qualsevol de l'arbre quaternari que es rep, per exemple, $\{arrel; recorregut(saSE); recorregut(saSO); recorregut(saNO); recorregut(saNE)\}$. Cada cop que es rep un arbre no nul cal fer 4 crides recursives, una per cada fill, les quals retornaran la llista de les claus emmagatzemades a cadascun dels subarbres. Aquestes 4 llistes s'han de concatenar i a la llista resultant s'ha d'afegir la clau que ocupa l'arrel del arbre. Si l'arbre que es rep és nul s'ha de tornar la llista buida. El cost és $\Theta(n)$.

(e) En el nostre cas només es pot produir la situació en la que $x1 < x2$ ja que no hi ha valors repetits. Això vol dir que el valor de les y no ens importa gens i que només fa falta tenir en compte el valor de les x de cada parella. També veiem que fent el recorregut de l'apartat anterior no sabem com *ordenar* entre ells els dos fills menors d'un node. Per tant, una solució simple per aquest apartat és agafar la llista obtinguda a l'apartat anterior i ordenarla creixentment pel valor de la x.

3. (a) Sigui $G = (V, E)$ un graf dirigit i connex, i sigui T un arbre generador de G (és a dir, un subgraf connex i acíclic de G que conté tots els seus vèrtexos) qualsevol. Sigui $x \in V$ una fulla qualsevol de T. Aleshores, $T - x$ és un arbre i per tant, $G - x$ és connex.

(b)
```
list<int> darrer (graph& G) { // int
  list<int> L; // int x;
  stack<int> S;
  vector<bool> vis(G.size(),false);
  forall_ver(u,G) {
    S.push(u);
```

```
    while (not S.empty()) {
      int v = S.top(); S.pop();
      if (not vis[v]) {
        vis[v] = true;
        L.push_back(v); // x = v;
        forall_adj(vw,G[v]) {
          S.push(*vw);
  } } } }
  return L.back(); // return x;
}
```

(c) Aquest algorisme fa un recorregut en profunditat del graf G i retorna el darrer vèrtex x visitat durant el recorregut. Donat que el recorregut en profunditat indueix un arbre generador del graf, aquest vèrtex x compleix les propietats de la demostració anterior i per tant, $G - x$ és connex.

(d) Cada vèrtex de G s'empila i desempila només una vegada i cada aresta de G només es travessa una vegada. Per tant, suposant una representació del graf amb llistes d'adjacència, el cost espacial i temporal de l'algorisme és $O(n + m)$, on n és l'ordre i m és la talla de G.

4. Primer cal crear un nou vector, VH, al qual traspassarem el contingut del conjunt S de manera que $VH[1] = S[0]$, $VH[2] = S[1]$, etc. Com que la primera fulla del heap es troba a la posició 5, només hem de fer 4 operacions d'enfonçar.

VH	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***
Estat inicial		5	9	12	10	9	3	1	5	4
enfonçar(4)		5	9	12	10	9	3	1	5	4
enfonçar(3)		5	9	12	10	9	3	1	5	4
enfonçar(2)		5	10	12	9	9	3	1	5	4
enfoncar(1)		12	10	5	9	9	3	1	5	4

5. Depèn. L'afirmació és certa si el graf està implementat amb llistes d'adjacència. En el pitjor dels casos, quan es processi l'aresta n-èsima, el recorregut en profunditat s'adonarà que hi ha cicle. I si no hi ha cicles, el nombre d'arestes recorregut serà més petit o igual a $n - 1$. Per tant, en tots dos casos, té cost $\Theta(n)$. Ara bé, si el graf està implementat amb matrius d'adjacència,i tingui o no cicles, cal un recorregut de tota la matriu i llavors té cost $\Theta(n^2)$.

6. La definició és correcte. Perquè un graf sigui un arbre arrelat cal que només hi hagi un vèrtex amb grau d'entrada 0, l'arrel, i també que existeixi un únic camí des de l'arrel a la resta de vèrtexs del graf. A més, entre dos vèrtexs qualsevol de l'arbre, u i v, només hi ha camí entre ells si pertanyen a la mateixa branca de l'arbre i com que no hi ha cicles només hi ha camí en un sentit (des de u a v o des de v a u). Les dues últimes condicions imposen que el grau d'entrada de tots els vèrtexs, excepte l'arrel, ha de ser necessàriament 1.

7. Entre heapsort i mergesort, ens quedaríem amb el heapsort perquè no necessita espai addicional.

8. És fals. Amb l'inordre es perd part de la informació sobre la posició exacta. Per exemple, considereu els dos arbres de cerca que formen els elements $\{1, 2\}$ (un amb arrel l'1 i fill dret el 2, i l'altre amb arrel el 2 i fill esquerre l'1). Els arbres són diferents, però l'inordre dóna el mateix.