UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Politècnica Superior d'Enginyeria
de Vilanova i la Geltrú

PUBLICACIÓ DOCENT

Apunts de teoria
D'ESIN

AUTOR: Bernardino Casas (bcasas@cs.upc.edu)
ASSIGNATURA: Estructura de la Informació (ESIN)
CURS: Segon
TITULACIONS: Grau en Informàtica
DEPARTAMENT: Ciències de la Computació
ANY: 2019/2020

Vilanova i la Geltrú, 30 de desembre de 2019

Índex

6
Diccionaris

Un diccionari és un univers en ordre alfabètic.

Anatole France (1844-1924)

6.8 Quicksort

6.8.1 Introducció

Quicksort és un algorisme d'ordenació que utilitza el principi de divideix i venceràs. A diferència d'altres algorismes similars (per ex. mergesort), no assegura que la divisió es faci en parts de mida similar.

L'algorisme selecciona un element que anomenarem **pivot** i divideix el vector d'elements al voltant del pivot. La tria del pivot es pot fer agafant:

- el primer element
- l'últim element
- un element al·leatori
- la mediana

La tasca clau del quicksort és la **partició**. L'objectiu d'aquesta tasca és col·locar el pivot p en la posició correcta del vector amb els elements més petits que p abans (a l'esquerra del pivot), i els més grans que p després (a la dreta del pivot).

En la versió de l'algorisme que veurem a continuació el pivot és el primer element del vector. L'evolució del vector que es vol ordenar es pot veure a la figura 6.1.

Un cop aconseguim tenir el vector amb els elements més petits abans del pivot i els més grans després del pivot, només caldrà ordenar els trossos a l'esquerra i a la dreta del pivot fent dues crides recursives a quicksort.

La fusió no és necessària, doncs els trossos a ordenar ja queden ben repartits a cada banda del pivot.

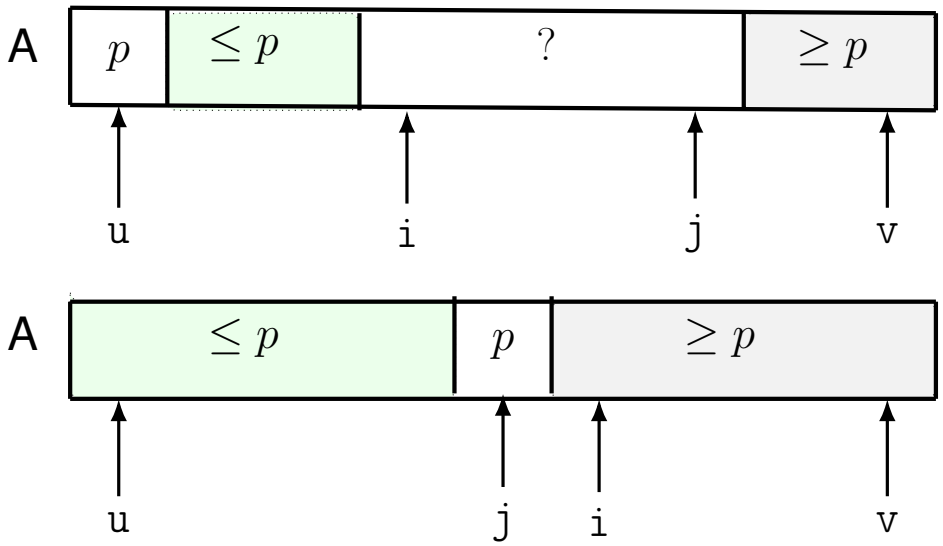


Figura 6.1: Evolució del vector en l'algorisme de quicksort

Mentre que en el mergesort la partició és simple i la feina es realitza durant l'etapa de fusió, en el quicksort és tot el contrari.

6.8.2 Implementació

6.8.2.1 Procediment principal

Acció per ordenar el subvector A[u..v]:

```
template <typename T>
void quicksort(T A[], nat u, nat v) {
  // Pre:  v < nombre d'elements de A
  // Post:  ordena el subvector A[u..v].
  if (v-u+1 <= M) {
    //utilitzar un algorisme d'ordenació simple
  }
  else {
    nat k = particio(A, u, v);
    quicksort(A, u, k-1);
    quicksort(A, k+1, v);
  }
}
```

Com a algorisme d'ordenació simple es pot utilitzar per ex. el d'ordenació per inserció. S'ha estimat que l'elecció òptima pel llindar M és entre 20 i 25.

En lloc d'ordenar per inserció cada tros de M o menys elements dins del mètode quicksort, una bona alternativa és ordenar per inserció tot el vector sencer al final de tot:

```
quicksort(A, 0, n-1);
ordena_insercio(A, 0, n-1);
```

Com que el vector A està quasi ordenat després d'aplicar quicksort, l'ordenació per inserció té un cost $\Theta(n)$, sent n el nombre d'elements a ordenar.

6.8.2.2 Partició

Hi ha moltes maneres de fer la partició. A Bentley & McIlroy (1993)[1] es presenta un procediment de partició molt eficient, fins i tot si hi ha elements repetits. Nosaltres veurem un algorisme bàsic però raonablement eficaç.

L'algorisme és el següent:

1. S'agafa com a pivot p el primer element del subvector (*A[u]*).

2. Es mantenen dos índexs i i j de forma que *A[u+1..i-1]* conté els elements menors o iguals que el pivot i *A[j+1..v]* conté els elements majors o iguals.

[1] JON L. BENTLEY and M. DOUGLAS McILROY. *Engineering a Sort Function*. Practice and Experience, vol. 23(11), 1249–1265. November 1993

3. Els índexs i i j recorren el vector d'esquerra a dreta i de dreta a esquerra respectivament fins que *A[i]>p* i *A[j]<p* o es creuen (i=j+1). En el primer cas (els índexos i i j encara no s'han creuat) s'intercanvien els elements *A[i]* i *A[j]* i es continua.

```
template <typename T>
nat particio(T A[], nat u, nat v) {
  nat i, j;
  T p = A[u];
  i = u+1;
  j = v;
  while (i < j+1) {
    while (i<j+1 and A[i]<=p) {
      ++i;
    }
    while (i<j+1 and A[j]>=p) {
      --j;
    }
    if (i < j+1){
      swap(A[i], A[j]); //intercanvia A[i] ↔ A[j]
    }
  }
  swap(A[u], A[j]); // intercanvia el pivot i l'últim dels menors
  return j;
}
```

6.8.3 Cost

El cost de l'algorisme de quicksort en el cas pitjor és $O(n^2)$. Això passa si en la majoria dels casos un dels trossos té molts pocs elements i l'altre els té gairebé tots. És el cas de si el vector ja estava ordenat creixentment o decreixentment.

Tanmateix, si ordenem un vector desordenat, normalment el pivot quedarà més o menys centrat cap a la meitat del vector.

Si això succeeix en la majoria d'iteracions, tindrem un cas similar al de mergesort: el cost d'una iteració de quicksort (l'acció partició) és lineal i com que es fan dues crides amb trossos més o menys la meitat de l'original resulta un cost total quasi-lineal $\Theta(n \log n)$.

6.10 Arbres digitals

6.10.1 Introducció

Les claus que identifiquen als elements d'un diccionari estan formades per una seqüència de símbols (per ex. caràcters, dígits, bits). La descomposició de les claus en símbols es pot aprofitar per implementar les operacions típiques d'un diccionari de manera notablement eficient.

A més a més, sovint necessitem operacions en el diccionari basades en la descomposició de les claus en símbols.

Exemple:

Donada una col·lecció de paraules C i un prefix p retornar totes les paraules de C que comencen pel prefix p.

Definició 1: Σ

Σ denota els símbols amb els que construeixen les claus. Considerem que els símbols pertanyen a un alfabet finit de $m \geq 2$ elements tal que $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_m\}$.

Exemple:

Si les claus les descomposem en bits, utilitzarem un alfabet que té $m = 2$ elements, és a dir, $\Sigma = \{0, 1\}$.

Definició 2: Σ^*

Σ^* denota el conjunt de les seqüències (cadenes) formades per símbols de l'alfabet (Σ).

Definició 3: Concatenació

Donades 2 seqüències u i v, $u \cdot v$ denota la seqüència resultant de concatenar u i v.

6.10.2 Tries

6.10.2.1 Definició

Un **trie** (anomenat també **arbre digital** o **arbre prefix**) és una estructura de dades de tipus arbre introduïda al 1959 per René de la Briandais. El nom d'aquesta estructura de dades ve de la capacitat de recuperar informació (en anglès reTRIEval).

Definició 4: Trie

A partir d'un conjunt finit de seqüències $X \subset \Sigma^*$ d'idèntica longitud podem construir un trie T tal i com es defineix a continuació. El **trie** T és un arbre m-ari definit recursivament com:

i) Si X és buit llavors T és un arbre buit.

ii) Si X conté un sol element llavors T és un arbre amb un únic node que conté a l'únic element de X.

iii) Si $|X| \geq 2$, sigui T_i el trie corresponent a la fórmula $X_i = \{y \text{ tal que } \sigma_i \cdot y \in X \wedge \sigma_i \in \Sigma\}$. Llavors T és un arbre m-ari constituït per una arrel que té com a fills els m subarbres $T_1, T_2, \ldots, T_m$.

A partir de l'anterior definició es poden extreure el següents lemes:

Lemma 6.1. *Si les arestes del trie T corresponent a un conjunt X s'etiqueten mitjançant els símbols de Σ de forma que l'aresta que uneix l'arrel amb el primer subarbre s'etiqueta amb σ_1, la que uneix l'arrel amb el segon subarbre s'etiqueta σ_2, . . . llavors les etiquetes del camí que ens porta des de l'arrel fins a una fulla no buida que conté a x constitueix el prefix més curt que distingeix unívocament a x (cap altre element de X comença amb el mateix prefix).*

Lemma 6.2. *Sigui p l'etiqueta corresponent a un camí que va des de l'arrel d'un trie T fins a un cert node (intern o fulla) de T. Llavors el subarbre que penja d'aquest node conté tots els elements de X que tenen com a prefix p (i no més elements).*

Lemma 6.3. *Donat un conjunt $X \subset \Sigma^*$ de seqüències d'igual longitud, el seu trie corresponent és únic. En particular T no depèn de l'ordre en que estiguin els elements de X.*

Lemma 6.4. *L'alçada d'un trie T és igual a la longitud mínima del prefix necessari per distingir qualssevol dos elements del conjunt que representa el trie. En particular, si l és la longitud de les seqüències en X, l'alçada de T serà $\leq l$.*

IMPORTANT!!
La definició d'un trie imposa la condició que totes les seqüències que formen el trie han de ser d'igual longitud. **Aquesta condició és molt restrictiva**. Cal trobar alguna manera de superar aquesta condició.

Si no exigim que les seqüències siguin de la mateixa longitud, com podem distingir si donats dos elements un és prefix de d'altre?

Una solució habitual consisteix en ampliar l'alfabet Σ amb un **símbol especial de fi de seqüència** (per ex. #) i marcar cadascuna de les seqüències en X amb aquest símbol. Això garanteix que cap de les seqüències marcades és prefix de les altres. L'inconvenient és que s'ha de treballar amb un alfabet de $m+1$ símbols i, per tant, amb arbres $(m+1)$-aris.

6.10.2.2 Tècniques d'implementació

Les tècniques d'implementació dels tries són les convencionals pels arbres:

- **Vector de punters** per node. En aquesta tècnica els símbols de Σ solen utilitzar-se com índexs (utilitzant una funció ind: $\Sigma \longrightarrow \{1, ..., m\}$).
  Les fulles que contenen els elements de X poden emmagatzemar exclusivament els sufixos restants, ja que el prefix està ja codificat en el camí de l'arrel a la fulla.

Exemple:

A la figura 6.2 es pot veure un exemple gràfic de trie implementat usant un vector de punters per node. Aquest trie s'ha construït a partir de les seqüències de caràcters alfabètics X= {ahora, alto, amigo, amo, asi, bar, barco, bota}. Com que la mida de l'alfabet és $m = 25$ el trie resultant és un arbre 26-ari.

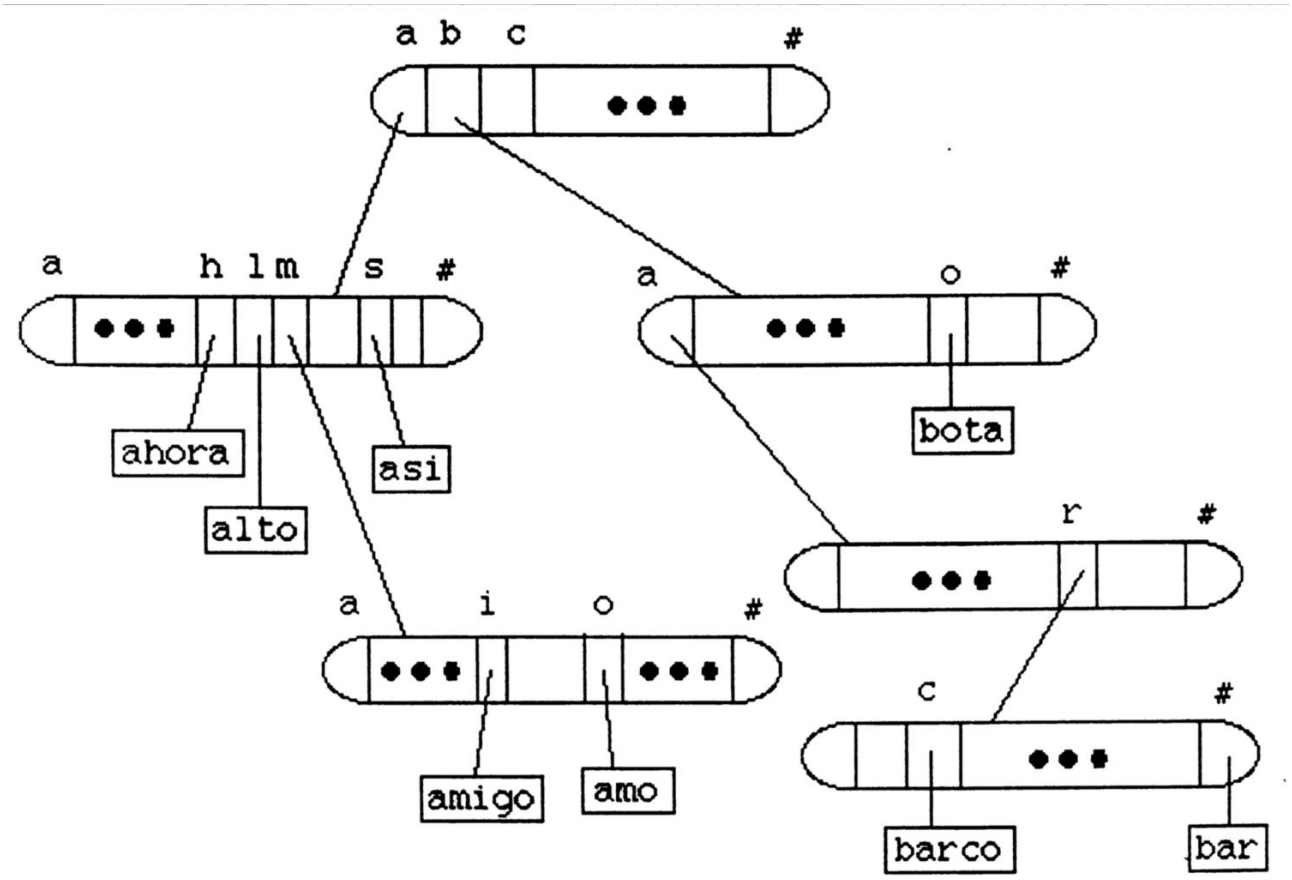


Figura 6.2: Trie implementat amb vectors de punters construït a partir de les seqüències de caràcters alfabètics X= {ahora, alto, amigo, amo, asi, bar, barco, bota}.

- **Primer fill–següent germà**. Cada node guarda un símbol i dos punters, un al primer fill i l'altre al següent germà. Com que acostuma a haver-hi un ordre sobre l'alfabet Σ, la

llista de fills de cada node habitualment s'ordena seguint aquest ordre.

Exemple:

A la figura 6.3 es pot veure un exemple gràfic de trie implementat amb la tècnica de primer fill–següent germà. Aquest trie s'ha construït a partir de les seqüències de caràcters alfabètics X= {ahora, alto, amigo, amo, asi, bar, barco, bota}. Per simplicitat, s'utilitza el mateix tipus de node per guardar els sufixos restants de cada clau. Així evitem usar nodes i apuntadors a nodes de tipus diferent.

6.10.2.3 Implementació primer fill – següent germà

IMPORTANT!!
La classe amb que es vulgui instanciar Clau ha de suportar les següents operacions:

Retorna la longitud >= 0 de la clau.

```
unsigned int size() const throw();
```

Retorna l'i-èssim símbol de la clau. El primer símbol és i==0.

```
Simbol operator[](int i) throw(error);
```

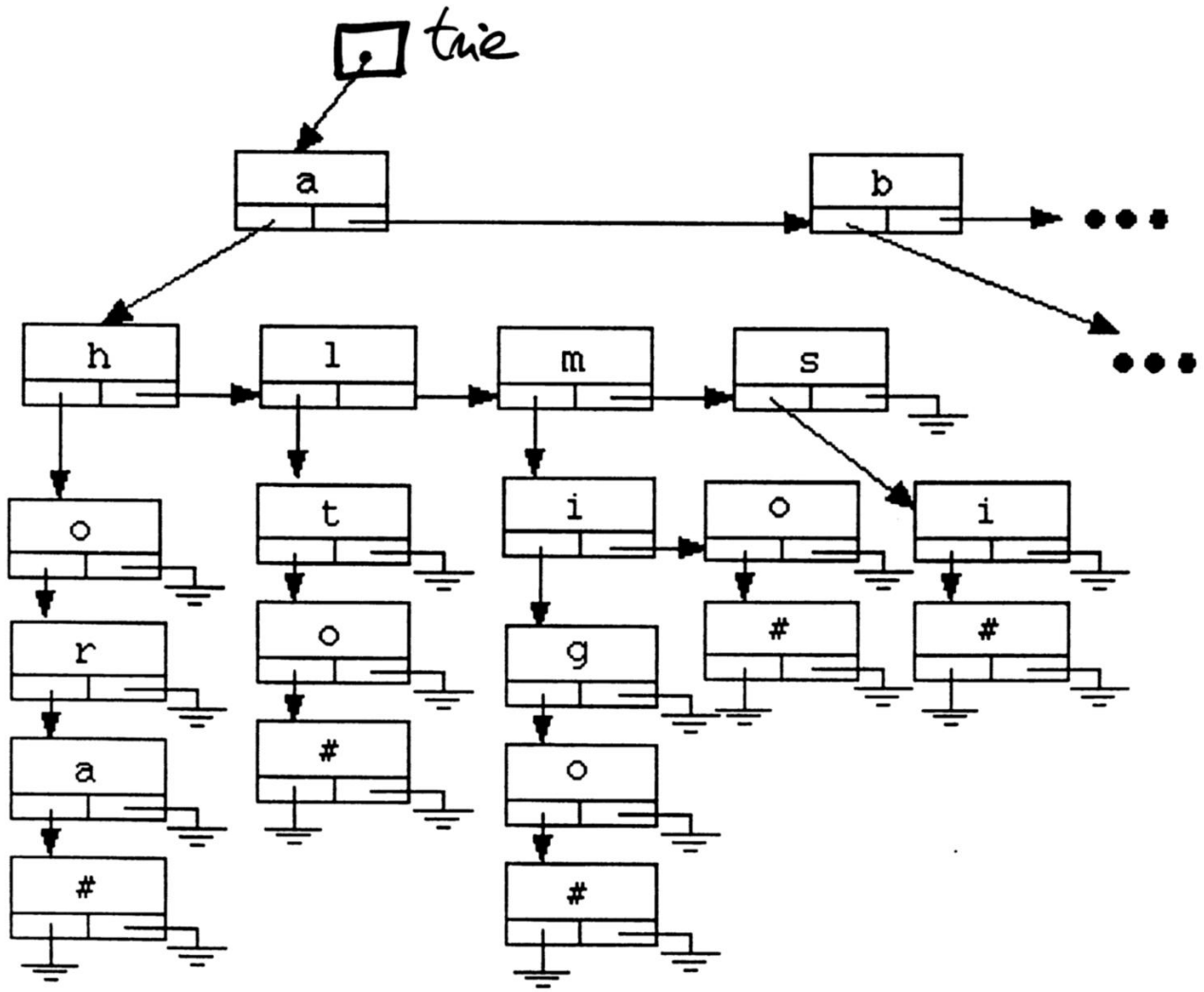


Figura 6.3: Trie implementat amb la tècnica de primer fill–següent germà construït a partir de les seqüències de caràcters alfabètics X= {ahora, alto, amigo, amo, asi, bar, barco, bota}.

A més, caldrà especialitzar la funció `especial` segons el tipus dels símbols de les claus del diccionari.

Per exemple pel tipus string la funció `especial` seria:

Retorna el símbol especial fi de clau.

```
template <>
char especial<string>() {
  return '#';
}
```

La representació d'aquesta classe és la següent:

```
template <class Simbol, class Clau, class Valor>
class diccDigital {
private:
  struct node_trie {
    Simbol _c;
    node_trie* _primfill;        // primer fill
    node_trie* _seggerma;        // següent germà
    Valor _v;
  };
  node_trie *_arrel;

  static node_trie* consulta_node (node_trie *p,
        const Clau &k, nat i) throw();

public:
  void consulta (const Clau &k, bool &hi_es, Valor &v)
         const throw(error);
  ...
};
```

Cost: $\Theta(k.size())$

```
template <class S, class C, class V>
void diccDigital<S, C, V>::consulta (const Clau &k,
        bool &hi_es, Valor &v) const throw() {
  node_trie *n = consulta_node(_arrel, k, 0);
  if (n == NULL) {
    hi_es = false;
  }
  else {
    v = n->_v;
    hi_es = true;
  }
}
```

Mètode privat de classe. Cost: $\Theta(k.size())$

```
template <class S, class C, class V>
typename diccDigital<S, C, V>::node_trie*
diccDigital<S, C, V>::consulta_node (node_trie *n,
        const Clau &k, nat i) throw() {
  node_trie *res = NULL;
  if (n != NULL) {
    if (i == k.size() and n->_c == especial<Clau>()) {
      res = n;
    }
    else if (n->_c > k[i]) {
      res = NULL;
    }
    else if (n->_c < k[i]) {
      res = consulta_node(n->_seggerma, k, i);
    }
    else if (n->_c == k[i]) {
      res = consulta_node(n->_primfill, k, i+1);
    }
  }
  return res;
}
```

6.10.3 Arbre ternari de cerca

6.10.3.1 Definició

Una alternativa que combina eficiència en l'accés als subarbres i estalvi de memòria consisteix en implementar cada node del trie com un ABC. L'estructura resultant s'anomena **arbre ternari de cerca**.

Definició 5: Arbre ternari de cerca

Un **arbre ternari de cerca** (anglès: ternary search tree (TST)) és un tipus d'arbre digital en que els nodes es gestionen de manera similar a un ABC. Cada node conté un únic símbol i tres fills:

- El fill esquerra conté els diferents símbols i-èssims de totes les claus més petites que tenen el mateix prefix format per i-1 símbols.
- El fill dret conté els diferents símbols i-èssims de totes les claus més grans que tenen el mateix prefix format per i-1 símbols.
- El fill central conté els símbols de la següent posició de totes les claus que tenen el mateix prefix.

Exemple:

A la figura 6.4 es pot veure l'evolució d'un arbre ternari de cerca buit al qual inserim les paraules DICTAT, DAU i DONA.

A la figura 6.5 es pot veure una representació gràfica d'un arbre ternari de cerca al qual se li han inserit les següents paraules en aquest ordre: X = {DICTAT, DAU, DONA, DEU, DAUS, ABAC, CASA, FAMA, FA, DO, DONS, D}.

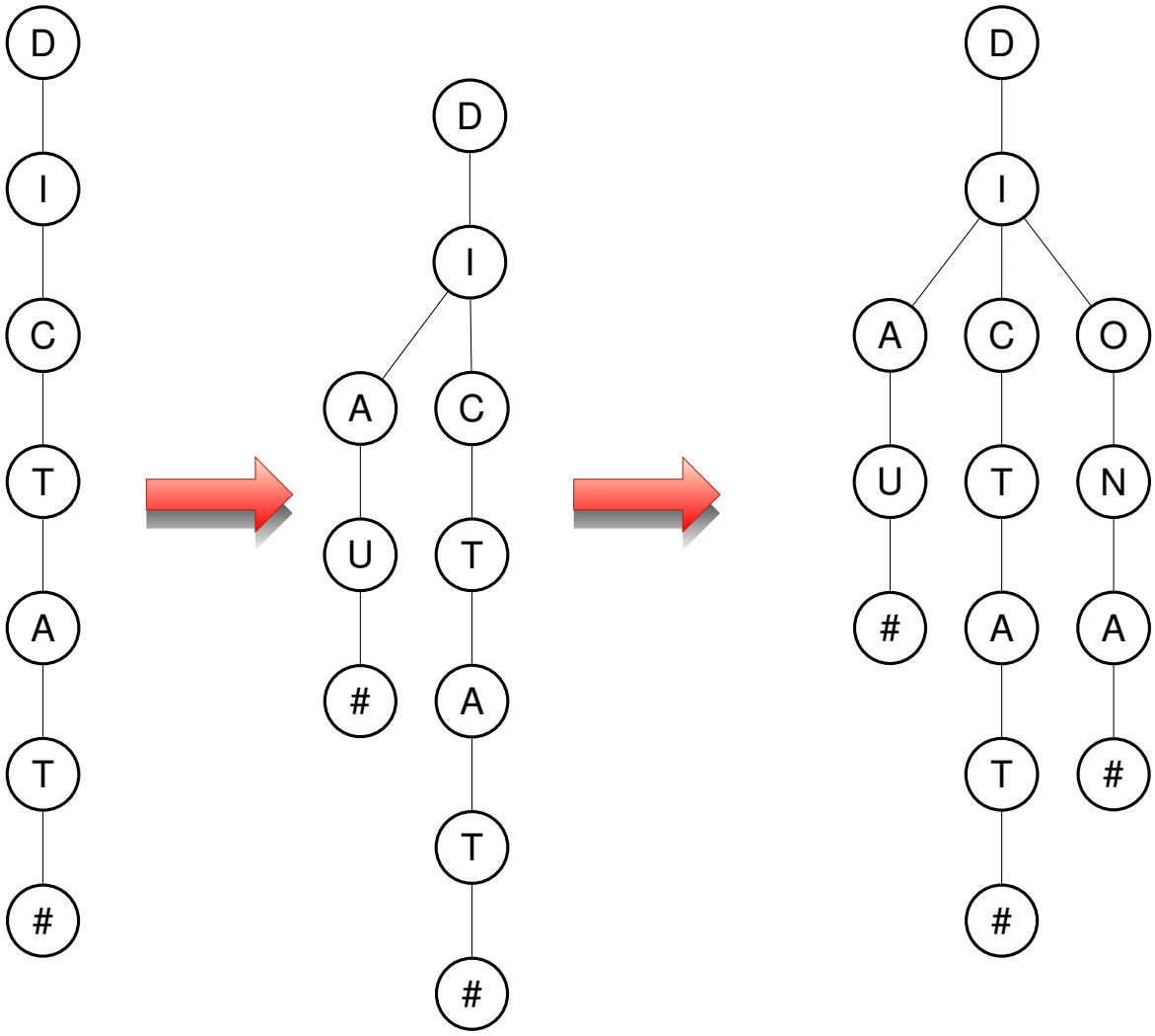


Figura 6.4: Evolució de l'arbre ternari de cerca afegint a l'arbre buit les paraules X = {DICTAT, DAU, DONA} .

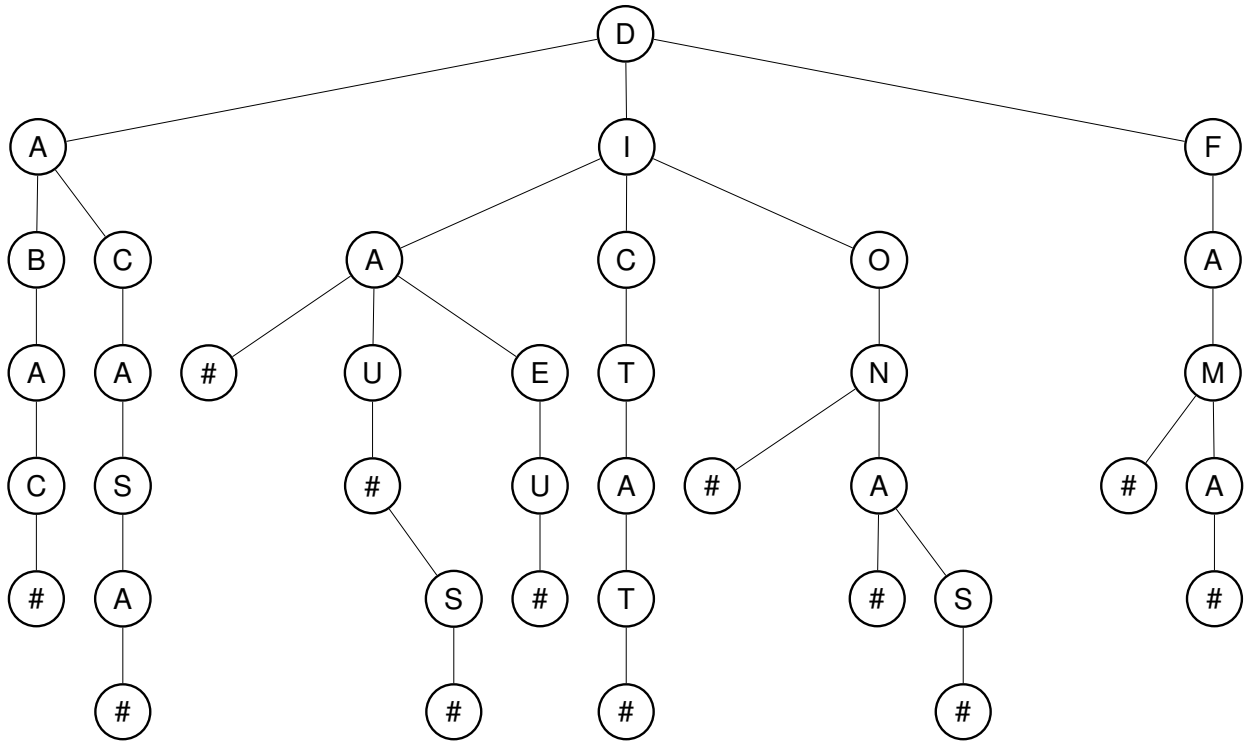


Figura 6.5: Arbre ternari de cerca format per les paraules X = {DICTAT, DAU, DONA, DEU, DAUS, ABAC, CASA, FAMA, FA, DO, DONS, D} .

6.10.3.2 Representació

```
template <class Simbol, class Clau, class Valor>
class diccDigital {
private:
  struct node_tst {
    Simbol _c;
    node_tst* _esq;
    node_tst* _dret;
    node_tst* _cen;
    Valor _v;
  };
  node_tst *_arrel;

  // operacions privades
  static node_tst* rconsulta (node_tst *n, nat i,
        const Clau &k) throw();
  static node_tst* rinsereix (node_tst *n, nat i,
        const Clau &k, const Valor &v) throw(error);

public:
  ...
  void consulta (const Clau &k, bool &hi_es, Valor &v)
        const throw();
  void insereix (const Clau &k, const Valor &v)
        throw(error);
  ...
};
```

6.10.3.3 Implementació

hi_es valdrà `true` i v contindrà el valor associat a k en cas que la clau k es trobi en el diccionari; en cas contrari hi_es valdrà `false` i no importa el valor de v.
Cost : $\Theta(k.size() \cdot \log(\#simbols))$.

```
template <class S, class C, class V>
void diccDigital<S, C, V>::consulta (const C &k,
                bool &hi_es, V &v) const throw() {
```

```
  node_tst *n = rconsulta(_arrel, 0, k);
  if (n == NULL) {
    hi_es = false;
  }
  else {
    v = n->_v;
    hi_es = true;
  }
}
```

Mètode privat de classe.
En cas que la clau k es trobi en el TST torna el node del TST que conté el símbol especial que pertany a la clau k; torna NULL en cas contrari.
Cost: $\Theta(k.size() \cdot \log(\#simbols))$

```
template <class S, class C, class V>
typename diccDigital<S, C, V>::node_tst*
diccDigital<S, C, V>::rconsulta (node_tst *n, nat i,
                 const C &k) throw() {
  node_tst *res = NULL;
  if (n != NULL) {
    if (i == k.size() and n->_c == especial<C>()) {
      res = n;
    }
    else if (n->_c > k[i]) {
      res = rconsulta(n->_esq, i, k);
    }
    else if (n->_c < k[i]) {
      res = rconsulta(n->_dret, i, k);
    }
    else if (n->_c == k[i]) {
      res = rconsulta(n->_cen, i+1, k);
    }
  }
  return res;
}
```

Insereix un parell <Clau, Valor> en el diccionari. Actualitza el valor si la Clau ja era present al diccionari. Cal tenir present que cal afegir un sentinella al final de la clau.
La funció `especial()` torna el símbol usat per indicar el final de la clau. Per ex. si la Clau és de tipus string llavors especial() retorna '#'.
Cost: $\Theta(k.size() \cdot \log(\#simbols))$

```
template <class S, class C, class V>
void diccDigital<S, C, V>::insereix (const C &k,
                const V &v) throw(error) {
  // Afegir el sentinella al final de la clau
  C k2 = k + especial<C>();
  _arrel = rinsereix(_arrel, 0, k2, v);
}
```

Mètode privat de classe.
Recorrerem tots els símbols de la clau. Cal tenir en compte que s'ha afegit el símbol especial al final de la clau.
Cost: $\Theta(k.size() \cdot \log(\#simbols))$

```
template <class S, class C, class V>
typename diccDigital::node*
diccDigital<S, C, V>::rinsereix (node_tst *n, nat i,
                const C &k, const V &v) throw(error) {
  if (n == NULL) {
    n = new node_tst;
    n->_esq = n->_dret = n->_cen = NULL;
    n->_c = k[i];
    try {
      if (i < k.size()-1) {
        n->_cen = rinsereix(n->_cen, i+1, k, v);
      }
      else { // i == k.size()-1; k[i] == Simbol()
        n->_v = v;
      }
    }
    catch (error) {
      delete n;
```

```
      throw;
    }
  }
  else {
    if (n->_c > k[i]) {
      n->_esq = rinsereix(n->_esq, i, k, v);
    }
    else if (n->_c < k[i]) {
      n->_dret = rinsereix(n->_dret, i, k, v);
    }
    else { // (n->_c == k[i])
      n->_cen = rinsereix(n->_cen, i+1, k, v);
    }
  }
  return n;
}
```

6.11 Radix sort

6.11.1 Introducció

La descomposició digital (en símbols o dígits) de les claus també pot utilitzar-se per l'ordenació. Els algorismes basats en aquest principi s'anomenen d'**ordenació digital** (anglès: *radix sort*).

Els algorismes d'ordenació digital es classifiquen en dos grans grups:

- ordenació pel dígit menys significatiu (anglès: *least significant digit (LSD)*)
- ordenació pel dígit més significatiu (anglès: *most significant digit (MSD)*)

6.11.2 Least Significant Digit (LSD)

Una possible implementació d'aquest algorisme utilitza una estructura addicional, en concret una taula de cues. La taula tindrà tantes posicions com la mida de l'alfabet que utilitzen els elements a ordenar.

Exemple: En cas que els elements a ordenar fossin nombres en base 10 la mida de la taula seria 10, ja que l'alfabet és de mida 10.

A partir d'un vector d'elements s'aplica el següent algorisme:

0. Es comença tractant el dígit menys significatiu dels elements del vector (unitats).
1. Cada element es situa a l'estructura de dades auxiliar segons el dígit que s'estigui tractant en aquest moment. En concret s'afegeix l'element a la cua que coincideixi amb el valor del dígit.
2. Un cop s'han tractat tots els elements a ordenar del vector, es recorre l'estructura auxiliar començant per la cua 0 i es copien els elements al vector (mantenint l'ordenació pròpia de cada cua).
3. Es torna a executar el pas 1, tractant els elements segons el següent dígit més significatiu.
4. S'acaba quan ja s'han tractat tots els dígits.

Exemple: Ordenar el següent vector usant *radix sort-LSD*:

93	65	534	742	542	9	554	32	44	23	57

1. Dígit unitats

```
0:
1:
2: 742 542 32
3: 93 23
4: 534 554 44
```

```
5:
6: 65
7: 57
8:
9: 9
```

742	542	32	93	23	534	554	44	65	57	9

2. Dígit desenes

```
0: 9
1:
2: 23
3: 32 534
4: 742 542 44
5: 554 57
6: 65
7:
8:
9: 93
```

9	23	32	534	742	542	44	554	57	65	93

3. Dígit centenes

```
0: 9 23 32 44 57 65 93
1:
2:
3:
4:
5: 534 542 554
6:
```

```
7: 742
8:
9:
```

9	23	32	44	57	65	93	534	542	554	742

6.11.3 Most Significant Digit (MSD)

Estudiarem el cas general en que la clau la descomposem en una seqüència de bits. Considerem que hem d'ordenar un vector de n elements cadascun dels quals és una seqüència de l bits.

6.11.3.1 Descripció de l'algorisme

Si ordenem el vector segons el bit de major pes, després cada bloc resultant l'ordenem segons el bit de següent pes, i així successivament, haurem ordenat tot el vector.

6.11.3.2 Implementació

Acció per ordenar el tros de vector A[u..v] tenint en compte el bit r-èssim.

```
template <typename T>
void radixsort(T A[], nat u, nat v, nat r) {
  if (u<v and r>=0) {
    nat k = particio_radix(A, u, v, r);
    radixsort(A, u, k, r-1);
    radixsort(A, k+1, v, r-1);
  }
}
```

La crida inicial és:

```
radixsort(A, 0, n-1, l);
```

La següent funció mostra una manera de realitzar la partició del tros de vector A[u..v] tenint en compte el bit r-èssim. La manera de procedir és molt similar a la partició del quicksort. Donat un element x, bit(x, r) retorna el bit r-èssim de x.

```
template <typename T>
nat particio_radix(T A[], nat u, nat v, nat r) {
  nat i = u, j = v;
  while (i < j+1) {
    while (i<j+1 and bit(A[i],r)==0) {
      ++i;
    }
    while (i<j+1 and bit(A[j],r)==1) {
      --j;
    }
    if (i < j+1) {
      swap(A[i], A[j]);
    }
  }
  return j;
}
```

6.11.4 Cost

Cadascuna de les etapes de radixsort té un cost lineal. Atès que el número d'etapes és l, el cost de l'algorisme és $\Theta(n \cdot l)$.

Una altra forma de deduir el cost és considerar el cost associat a cada element del vector: un element qualsevol és examinat (i de vegades intercanviat amb un altre element) com a molt l vegades, per tant el cost total és $\Theta(n \cdot l)$.

7

Cues de prioritat

L'altra cua sempre és més ràpida.

5ª formulació de la Llei de Murphy

7.1 Conceptes

Una **cua de prioritat** (anglès: *priority queue*) és una col·lecció d'elements on cada element té associat un valor susceptible d'ordenació denominat prioritat. Una cua de prioritat es caracteritza per admetre:

- insercions de nous elements.
- consulta de l'element de prioritat mínima.
- esborrat l'element de prioritat mínima.

De forma anàloga es poden definir cues de prioritat que admeten la consulta i l'eliminació de l'element de màxima prioritat a la col·lecció. Les cues de prioritat ordenades per segons la prioritat mínima les anomenarem **min-cua** i les ordenades segons la prioritat màxima **max-cua**.

Exemple:
Un exemple de cua de prioritat és una recepció oficial. No importa l'ordre en que arribin els diferents dignataris, aquests sempre sortiran en ordre descendent segons el rang: primer el president convidat, després el del país, després els ministres, etc. Si mentre van sortint, arriba algun endarrerit, tots els de rang inferior li cediran el seu lloc.

7.2 Especificació

A l'especificació següent assumirem que el tipus Prio ofereix una relació d'ordre total $<$. D'altra banda, es pot donar el cas que existeixin diversos elements amb la mateixa prioritat i en aquest cas és irrellevant quin dels elements retorna l'operació `min` o elimina `elim_min`. A vegades s'utilitza una operació `prio_min` que retorna la prioritat mínima.

```
template <typename Elem, typename Prio>
class CuaPrio {
public:
```

Constructora, crea una cua buida.

```
CuaPrio() throw(error);
```

Tres grans.

```
CuaPrio(const CuaPrio &p) throw(error);
CuaPrio& operator=(const CuaPrio &p) throw(error);
~CuaPrio() throw();
```

Afegeix l'element x amb prioritat p a la cua de prioritat.

```
void insereix(const Elem &x, const Prio &p) throw(error);
```

Retorna un element de mínima prioritat en la cua de prioritat. Genera un error si la cua és buida.

```
Elem min() const throw(error);
```

Retorna la mínima prioritat present en la cua de prioritat. Genera un error si la cua és buida.

```
  Prio prio_min() const throw(error);
```

Elimina un element de mínima prioritat de la cua de prioritat. Genera un error si la cua és buida.

```
  void elim_min() throw(error);
```

Retorna cert si i només si la cua és buida; fals en cas contrari.

```
  bool es_buida() const throw();
private:
  ...
};
```

7.3 Usos de les cues de prioritat

Les cues de prioritat tenen múltiples usos:

A) Algorismes voraços

Amb freqüència s'utilitzen per a implementar algorismes voraços. Aquest tipus d'algorismes normalment tenen una iteració principal, i una de les tasques a realitzar a cadascuna d'aquestes iteracions és seleccionar un element que minimitzi o (maximitzi) un cert criteri. El conjunt d'elements entre els quals ha d'efectuar la selecció és freqüentment dinàmic i admet insercions eficients. Alguns d'aquests algorismes són:

- Algorismes de *Kruskal* i *Prim* pel càlcul de l'arbre d'expansió mínim d'un graf etiquetat.
- Algorisme de *Dijkstra* pel càlcul de camins mínims en un graf etiquetat.
- Construcció de *codis de Huffman* (codis binaris de longitud mitja mínima).

B) Ordenació

Una altra tasca en la que es poden utilitzar cues de prioritat és l'ordenació. Utilitzem una cua de prioritat per ordenar la informació de menor a major segons la prioritat.

Exemple:
En aquest exemple tenim dues taules: `info` i `clau`. Es vol que els elements d'aquestes dues taules estiguin ordenats per la clau.

```
template <typename Elem, typename Prio>
void ordenar(Elem info[], Prio clau[], nat n)
        throw(error) {
  CuaPrio<Elem, Prio> c;
  for (i=0; i<n; ++i) {
    c.insereix(info[i], clau[i]);
  }
  for (nat i=0; i<n; ++i) {
    info[i] = c.min();
    clau[i] = c.prio_min();
    c.elim_min();
  }
}
```

C) Element k-èssim

També s'utilitzen cues de prioritat per a trobar l'element k-èssim d'un vector no ordenat. Es col·loquen els k primers elements del vector en una *max-cua* i a continuació es fa un recorregut de la resta del vector, actualitzant la cua de prioritat cada vegada que l'element és menor que el màxim dels elements de la cua, eliminant al màxim i inserint l'element en curs.

```
template <typename T>
T k_essim (T v[], nat k) throw(error) {
  CuaPrio<T, T> c;
  for (nat i=0; i<k; ++i) {
    c.inserir(v[i], v[i]);
  }
  for (nat i=k; i<n; ++i) {
    if (v[i] < c.max()) {
      c.elim_max();
      c.insereix(v[i], v[i]);
    }
  }
  return c.max();
}
```

7.4 Implementació

La majoria de les tècniques utilitzades en la implementació de diccionaris poden ser utilitzades per la implementació de cues de prioritat, a excepció de les taules de dispersió i els tries.

Algunes d'aquestes representacions són:

- Llista ordenada per prioritat
- Arbre de cerca
- Skip Lists
- Taula de llistes
- Monticles

7.4.1 Llista ordenada per prioritat

Tant la consulta com l'eliminació del mínim són trivials i els seu cost és $\Theta(1)$. Però les insercions tenen cost $\Theta(n)$ (on n és el nombre d'elements), tant en cas pitjor com en el promig.

7.4.2 Arbre de cerca

L'arbre de cerca pot ser equilibrat o no.

S'utilitza com a criteri d'ordre dels seus elements les corresponents prioritats. S'ha de modificar lleugerament l'invariant de la representació per a admetre i tractar adequadament les prioritats repetides.

En aquest cas es pot garantir que totes les operacions (insercions, consultes, eliminacions) tenen un cost $\Theta(\log n)$ en el

pitjor cas i en el mig si l'ABC és equilibrat (AVL), i només en el cas mig si no ho és.

7.4.3 Skip lists

Tenen un rendiment idèntic al que ofereixen els ABC's. Malgrat els costos són logarítmics només en el cas mig, aquest cas mig no depèn ni de l'ordre d'inserció ni de l'existència de poques prioritats repetides.

7.4.4 Taula de llistes

Si el conjunt de possibles prioritats és reduït llavors serà convenient utilitzar aquesta estructura. Cada llista correspon a una prioritat o interval reduït de prioritats.

7.4.5 Monticles

En la resta d'aquest capítol estudiarem una tècnica específica per a la implementació de cues de prioritat basada en els denominats monticles.

7.5 Monticles

7.5.1 Definició

Definició 6: Monticle

Un **monticle** (anglès: *heap*) és un arbre binari tal que:

`i)` Totes les fulles (sub-arbres buits) es situen en els dos últims nivells de l'arbre.

`ii)` Al nivell avantpenúltim el nombre de nodes amb un únic fill és com a màxim un. Aquest únic fill serà el seu fill esquerre, i tots els nodes a la seva dreta són nodes interns sense fills.

`iii)` L'element emmagatzemat en un node qualsevol és sempre major o igual (o sempre menor o igual) que els elements emmagatzemats als seus fills esquerre i dret.

Per simplificar els exemples dels monticles considerem que l'element i la prioritat és el mateix.

Un monticle és un arbre binari quasi-complet degut a les propietats `i`–`ii`. La propietat `iii` es denomina ordre del monticle, i parlarem de **max-heaps** si els elements són $\geq$ que els seus fills o **min-heaps** si són $\leq$.

Exemple: Veure la figura 7.1 per tenir un exemple gràfic de max-cua.

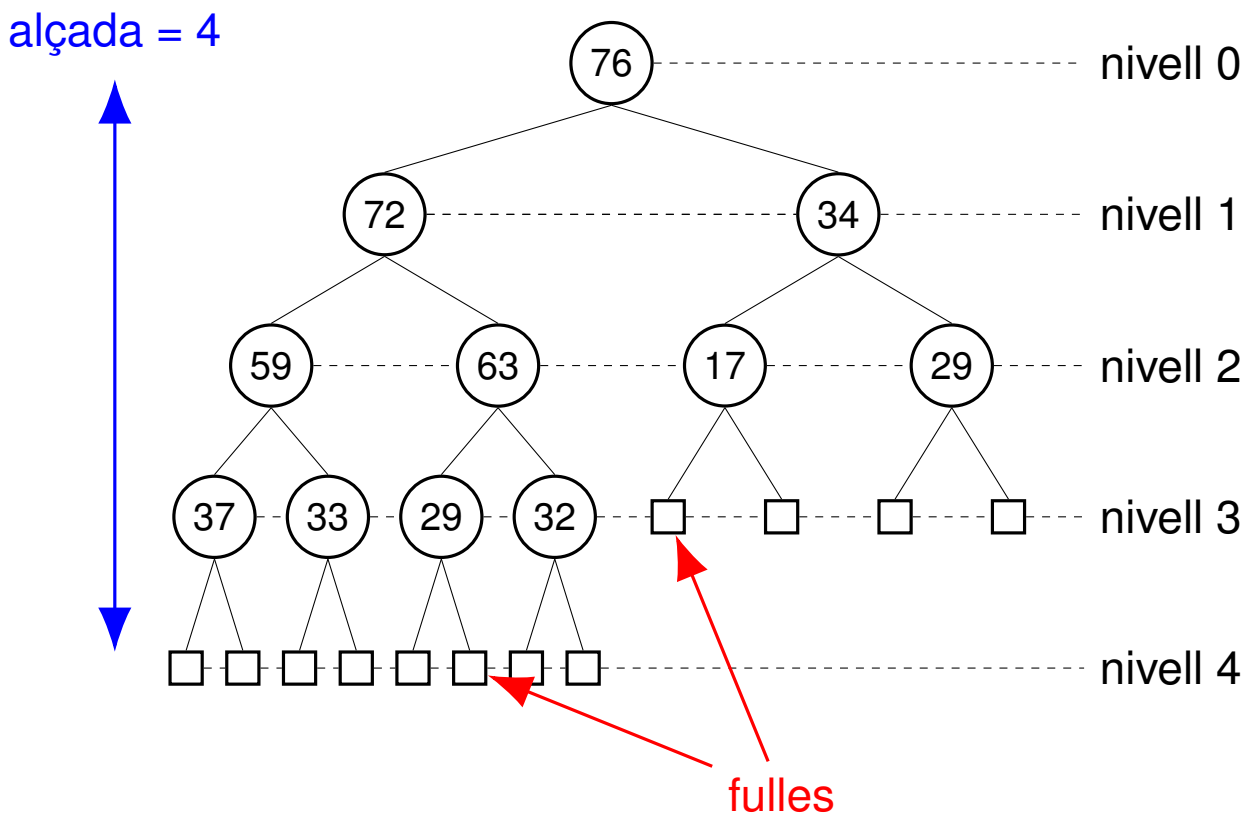


Figura 7.1: Exemple de monticle (max-cua).

De les propietats `i–iii` es desprenen dues conseqüències importants:

1. L'element màxim es troba a l'arrel.

2. Un monticle amb n elements té alçada $h = \lceil \log_2(n+1) \rceil$

7.5.2 Consulta del màxim

La consulta del màxim és senzilla i eficient doncs només cal examinar l'arrel.

7.5.3 Eliminació del màxim

Un procediment que s'utilitza consisteix en substituir l'arrel (el màxim) per l'últim element del monticle (situat a l'últim nivell més a la dreta). Això garanteix que es compleixin les propietats `i` i `ii`, però la propietat `iii` deixa de complir-se. Es restableix la propietat `iii` amb un procediment denominat `enfonsar` que consisteix en:

- intercanviar un node amb el més gran dels seus dos fills sempre i quan el node sigui menor que algun d'ells.
- aquest procés es repeteix amb el node fins que no es pot fer cap canvi.

Exemple: Veure les figures 7.2, 7.3, 7.4 i 7.5 per tenir un exemple del procés d'eliminació del màxim d'un monticle.

7.5.4 Afegir un nou element

Una possibilitat consisteix en col·locar el nou element com a últim element del monticle, justament a la dreta de l'últim o com a primer d'un nou nivell. Per això s'ha de localitzar la primera fulla i substituir-la per un nou node amb l'element a inserir. A continuació s'ha de restablir l'ordre del monticle utilitzant per això un procediment denominat `surar` (treballa a la inversa que l'`enfonsar`):

- el node en curs es compara amb el seu pare i es realitza l'intercanvi si aquest node és més gran que el pare.
- aquest procés es repeteix fins que no es pugui fer cap canvi.

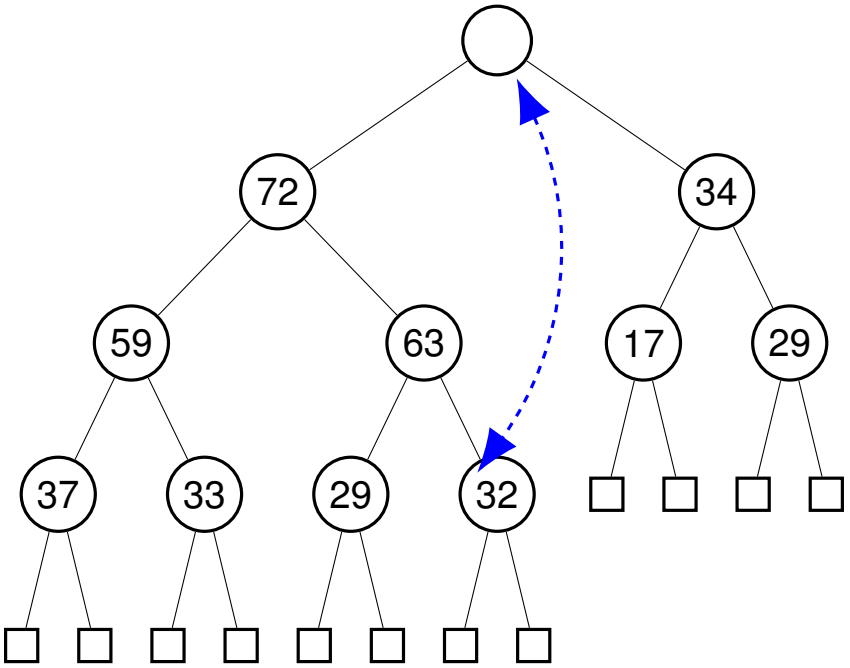


Figura 7.2: Esborrat del màxim del monticle de la figura 7.1 – substitució per l'arrel

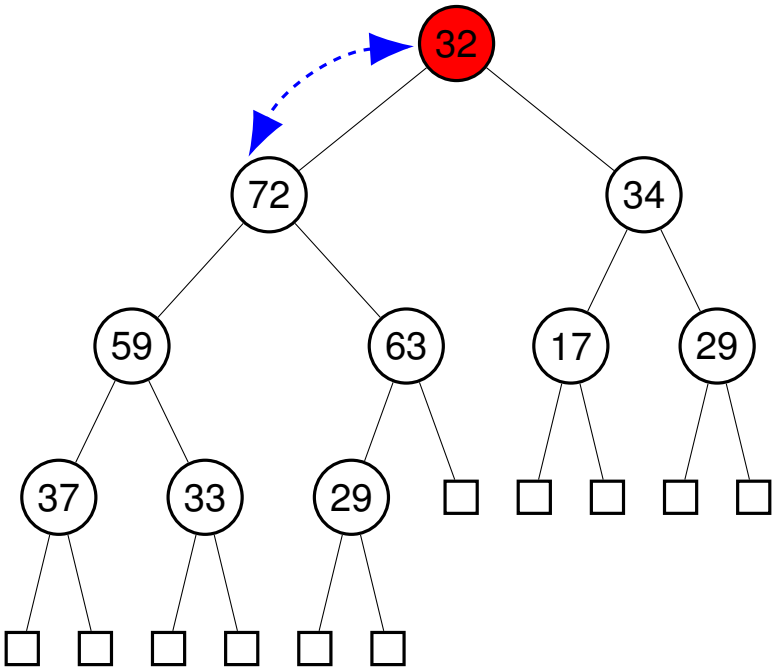


Figura 7.3: Esborrat del màxim del monticle de la figura 7.1 – enfonsat 1

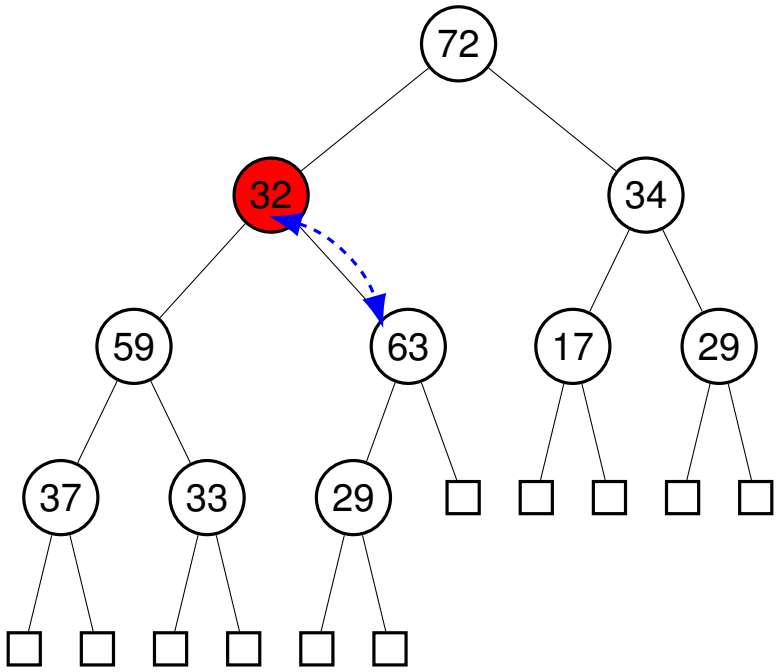


Figura 7.4: Esborrat del màxim del monticle de la figura 7.1 – enfonsat 2

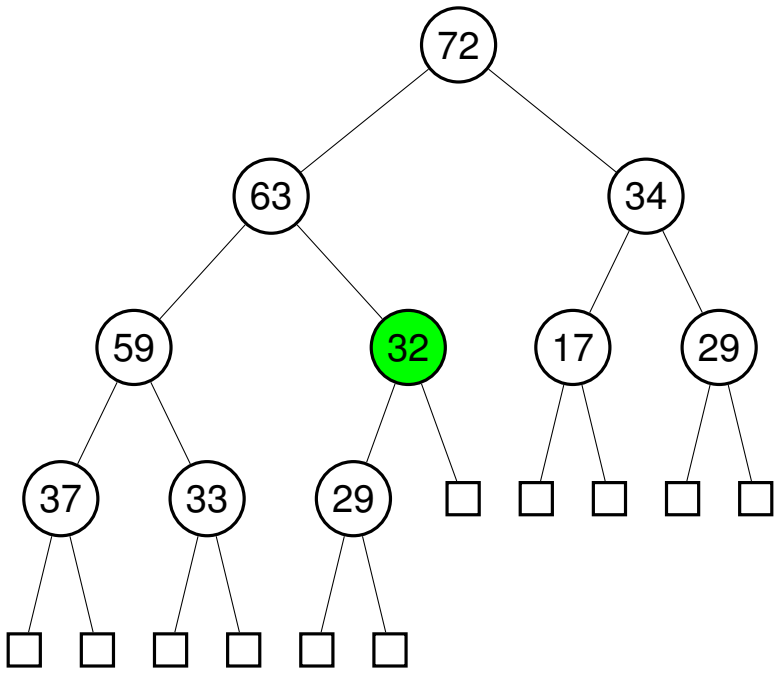


Figura 7.5: Esborrat del màxim del monticle de la figura 7.1 – resultat final

.

7.5.5 Implementació amb memòria dinàmica

Es pot implementar un monticle mitjançant memòria dinàmica. La representació escollida ha d'incloure apuntadors al fill esquerre i al dret i també al pare. A més cal resoldre de manera eficaç la localització de l'últim element i del pare de la primera fulla.

Donat que l'alçada del heap és $\log(n)$, el cost de les insercions i eliminacions és logarítmic.

7.5.6 Implementació amb vector

Una alternativa atractiva és la implementació de monticles mitjançant un vector `A`. No es malbarata massa espai donat que el heap és arbre quasi-complet.

Les regles per a representar els elements del monticle en un vector són simples:

1. `A[1]` conté l'arrel.

2. Si $2i \leq n$ llavors `A[2i]` conté el fill esquerre de `A[i]`

3. Si $2i + 1 \leq n$ llavors `A[2i+1]` conté al fill dret de `A[i]`.

4. Si $i \div 2 \geq 1$ llavors `A[i` $\div$ `2]` conté al pare de `A[i]`.

Les regles anteriors impliquen:

- Els elements del monticle s'ubiquen en posicions consecutives del vector.
- L'arrel es col·loca a la primera posició.
- La resta d'elements segueix un recorregut de l'arbre per nivells d'esquerra a dreta.
- La primera casella del vector (`A[0]`) no s'usarà per guardar elements. Es podria usar per guardar el nombre total d'elements del monticle.

7.5.6.1 Representació

Com a representació usarem una taula estàtica. Dues millores possibles serien:

- usar una taula dinàmica
- usar la primera casella de la taula (`_taula[0]`) per guardar el nombre d'elements del monticle.

```
template <typename Elem, typename Prio>
class CuaPrio {
  ...
  private:
    static const nat MAX_ELEM = 100;
```

Nombre d'elements que conté el monticle.

```
    nat _nelems;
```

Taula de MAX_ELEMS parells <Elem, Prio>.
L'element de la posició 0 de la taula no s'usa.

```
    pair<Elem, Prio> _taula[MAX_ELEM+1];
    void enfonsar (nat p) throw();
    void surar (nat p) throw();
};
```

7.5.6.2 Implementació

Només implementarem els mètodes més destacats de la classe que hem definit a l'apartat 7.2. La implementació de les tres grans i la resta de mètodes és trivial.

```
// Cost: Θ(1)
template <typename Elem, typename Prio>
CuaPrio<Elem, Prio>::CuaPrio() throw (error)
        : _nelems(0) {
}

// Cost: Θ(log2(n))
template <typename Elem, typename Prio>
void CuaPrio<Elem, Prio>::insereix(const Elem &x,
        const Prio &p) throw(error) {
  // En el cas que la taula fos dinàmica si arribem al
  // màxim d'elements caldria ampliar la taula.
  if (_nelems == MAX_ELEM) throw error(CuaPrioPlena);
  ++_nelems;
  _taula[_nelems] = make_pair(x, p);
  surar(_nelems);
}

// Cost: Θ(1)
template <typename Elem, typename Prio>
Elem CuaPrio<Elem, Prio>::min() const throw(error) {
  if (_nelems == 0) throw error(CuaPrioBuida);
  return _taula[1].first;
}
```

```
// Cost: Θ(1)
template <typename Elem, typename Prio>
Prio CuaPrio<Elem, Prio>::prio_min() const throw(error) {
  if (_nelems == 0) throw error(CuaPrioBuida);
  return _taula[1].second;
}

// Cost: Θ(log2(n))
template <typename Elem, typename Prio>
void CuaPrio<Elem, Prio>::elim_min() throw(error) {
  if (_nelems == 0) throw error(CuaPrioBuida);
  swap(_taula[1], _taula[_nelems]);
  --_nelems;
  enfonsar(1);
}
```

Operació privada recursiva.
Enfonsa el node j-èssim fins a restablir l'ordre del monticle a `_taula`; els subarbres del node j són monticles.
Cost : $\Theta(\log_2(\frac{n}{j}))$.

```
template <typename Elem, typename Prio>
void CuaPrio<Elem, Prio>::enfonsar(nat j) throw(error) {
  // si j no té fill esquerre ja hem acabat
  if (2*j <= _nelems) {
    nat hj = 2*j;
    if (hj < _nelems and
        _taula[hj].second > _taula[hj+1].second) {
      ++hj;
    }
    // hj apunta al fill de mínima prioritat de j.
    // Si la prioritat de j és major que la prioritat del
    // seu fill menor cal intercanviar i seguir enfonsant.
    if (_taula[j]).second > _taula[hj].second) {
      swap(_taula[j], _taula[hj]);
      enfonsar(hj);
```

```
    }
  }
}
```

Operació privada (versió iterativa)
Fa surar el node p-èssim fins a restablir l'ordre del monticle; tots els nodes excepte el p-èssim satisfan la propietat `iii`.
Cost : $\Theta(\log_2(p))$

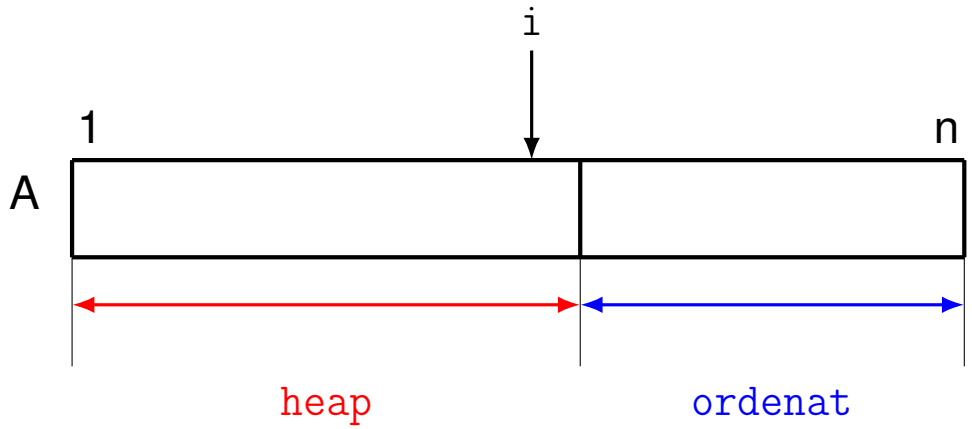
```
template <typename Elem, typename Prio>
void CuaPrio<Elem, Prio>::surar (nat p) throw(error) {
  bool fi = false;
  while (p > 1 and not fi) {
    nat q = p / 2;
    if (_taula[q].second > _taula[p].second) {
      swap(_taula[p], _taula[q]);
      p = q;
    }
    else {
      fi = true;
    }
  }
}
```

7.6 Heapsort

7.6.1 Introducció

Heapsort (Williams, 1964) ordena un vector de n elements construint un monticle amb els n elements i extraient-los un a un del monticle a continuació. El propi vector que emmagatzema els elements s'utilitza per a construir el heap, de manera que heapsort actua in-situ i només requereix un espai auxiliar de memòria constant.

Durant el procés d'ordenació de l'algorisme de heapsort el vector evoluciona com es pot veure a la figura 7.6.



$$A[1] \leq A[i+1] \leq A[i+2] \leq \ldots \leq A[n]$$

$$A[1] = max_{1 \leq k \leq i}\ A[k]$$

Figura 7.6: Evolució del vector en l'algorisme de heapsort.

7.6.2 Funcionament de l'algorisme de heapsort

7.6.2.1 Creació del MAXHEAP

Cal recórrer els elements del vector començant pel final i cada element s'ha d'**enfonsar**. A la pràctica es pot començar a enfonsar a partir de la meitat del vector ja que els elements que estan entre la meitat i el final del vector no són enfonsables.

Veure les figures 7.7, 7.8, 7.9 i 7.10 per tenir una representació de la creació d'una max-cua.

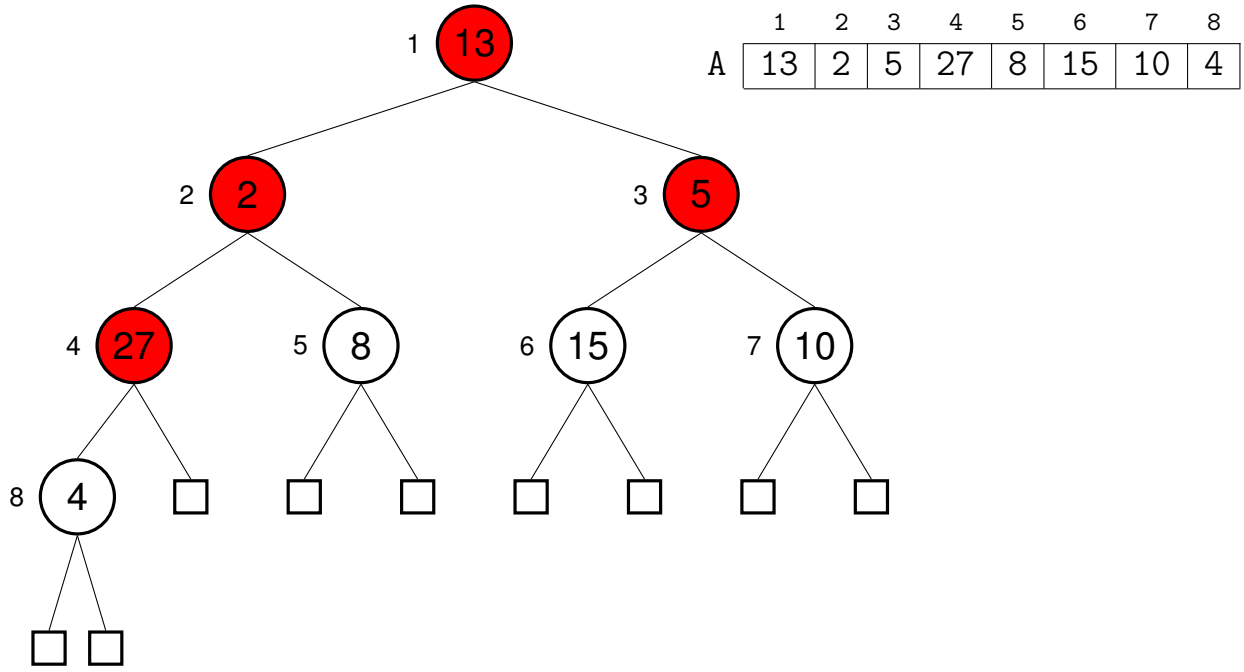


Figura 7.7: Creació del MAXHEAP. Estat inicial. Els nodes blancs ja han estat tractats i els vermells no.

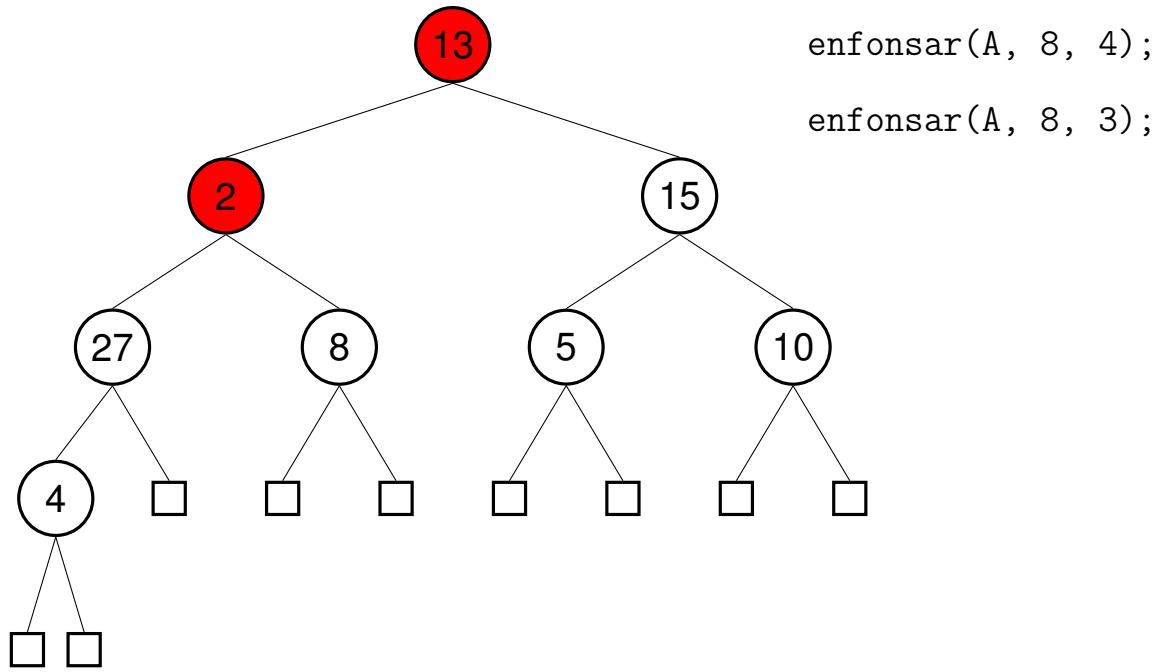


Figura 7.8: Creació del MAXHEAP. Tractament dels elements en les posicions 4 i 3. Els nodes blancs ja han estat tractats i els vermells no.

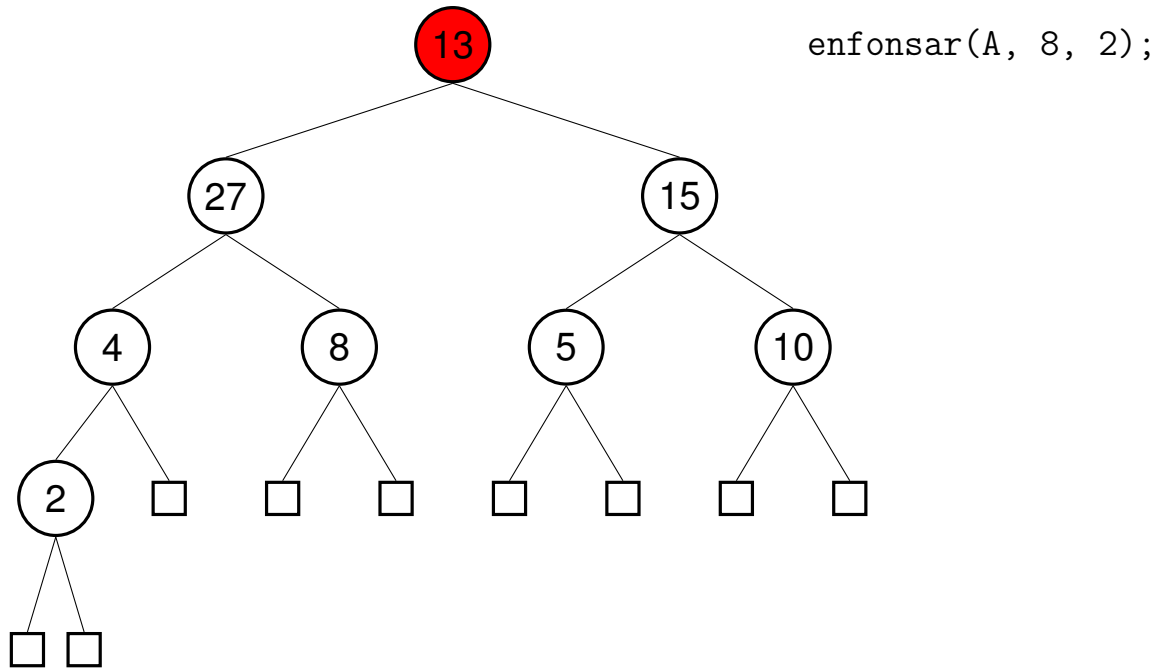


Figura 7.9: Creació del MAXHEAP. Tractament de l'element en la posició 2. Els nodes blancs ja han estat tractats i els vermells no.

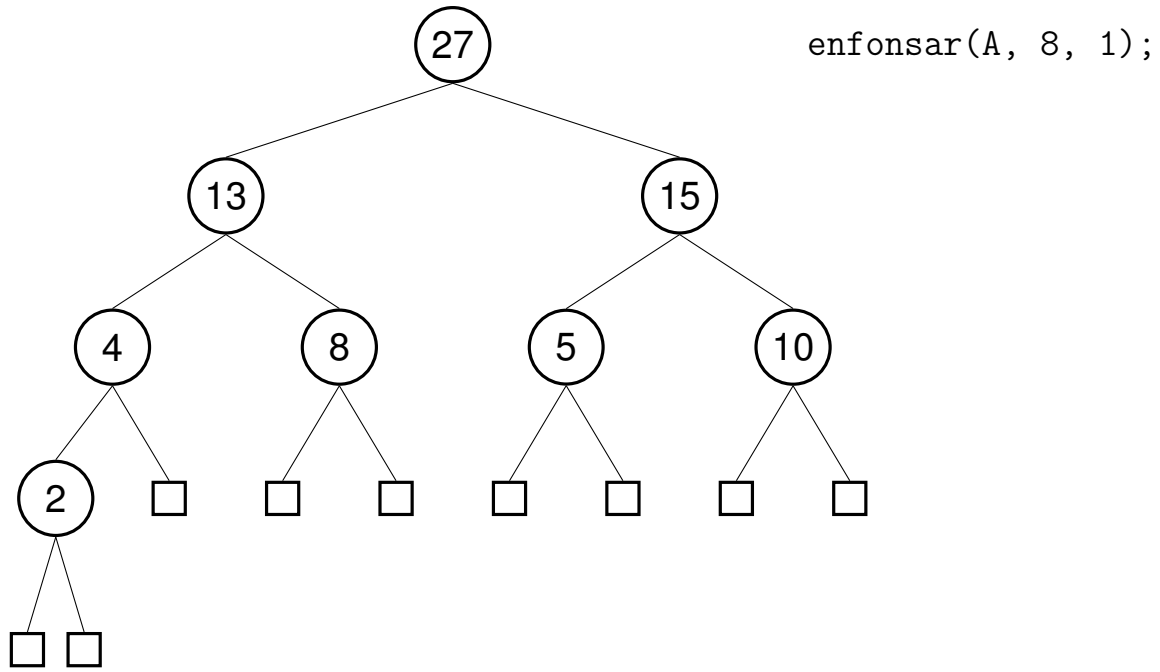


Figura 7.10: Creació del MAXHEAP. Tractament de l'element en la posició 1. Tots els nodes del monticle compleixen les propietats d'un monticle.

7.6.2.2 Eliminació del màxim del MAXHEAP

La següent part de l'algorisme és anar eliminant el màxim del heap.

Les figures 7.11 i 7.12 mostren l'eliminació del màxim en el heap. Es parteix del monticle de la figura 7.10 que es crea en la primera part d'aquest algorisme d'ordenació.

Les figures 7.13 i 7.14 mostren l'eliminació del segon màxim en el heap.

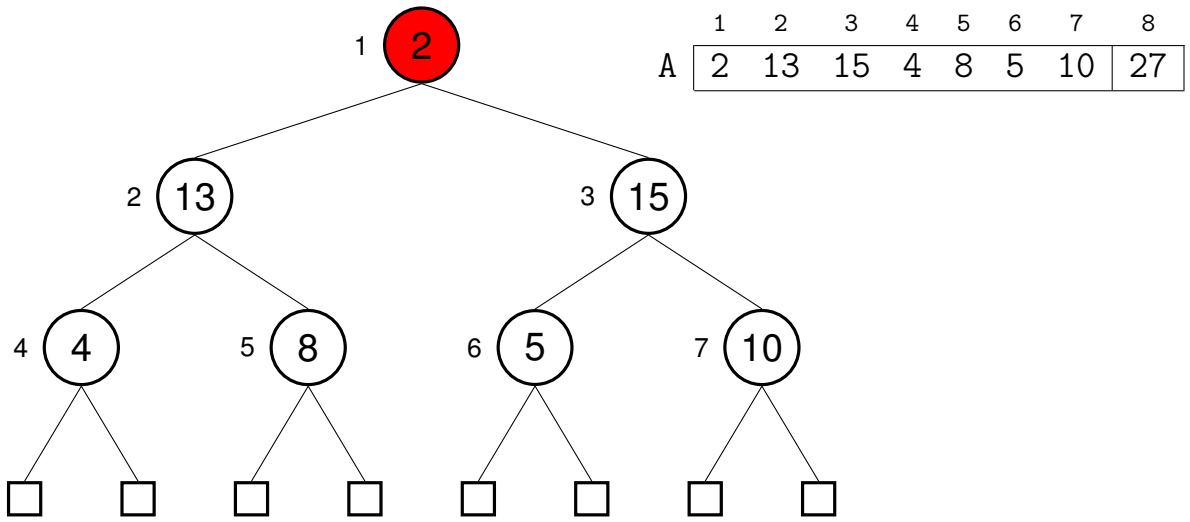


Figura 7.11: Esborrat del màxim del MAXHEAP. Els nodes blancs satisfan la propietat d'un monticle i els vermells no.

Les figures 7.15, 7.16, 7.17 i 7.18 mostren l'evolució de l'algorisme de heapsort centrant-nos en la taula. Es veu únicament la taula d'elements que estem ordenant i quins intercanvis fem per mantenir les propietats del monticle.

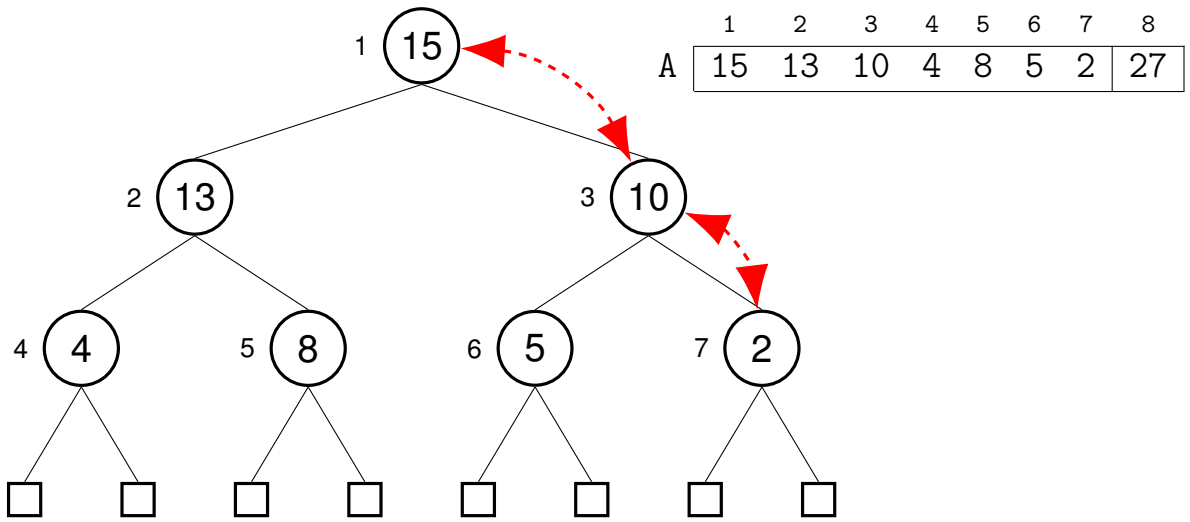


Figura 7.12: Esborrat del màxim del MAXHEAP. Els nodes blancs satisfan la propietat d'un monticle i els vermells no.

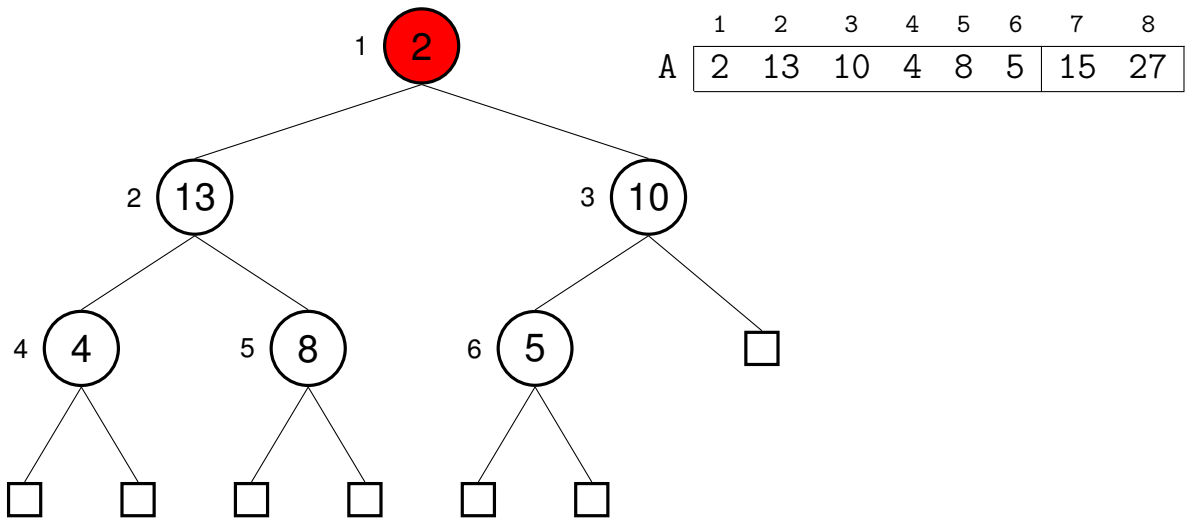


Figura 7.13: Esborrat del màxim del MAXHEAP. Els nodes blancs satisfan la propietat d'un monticle i els vermells no.

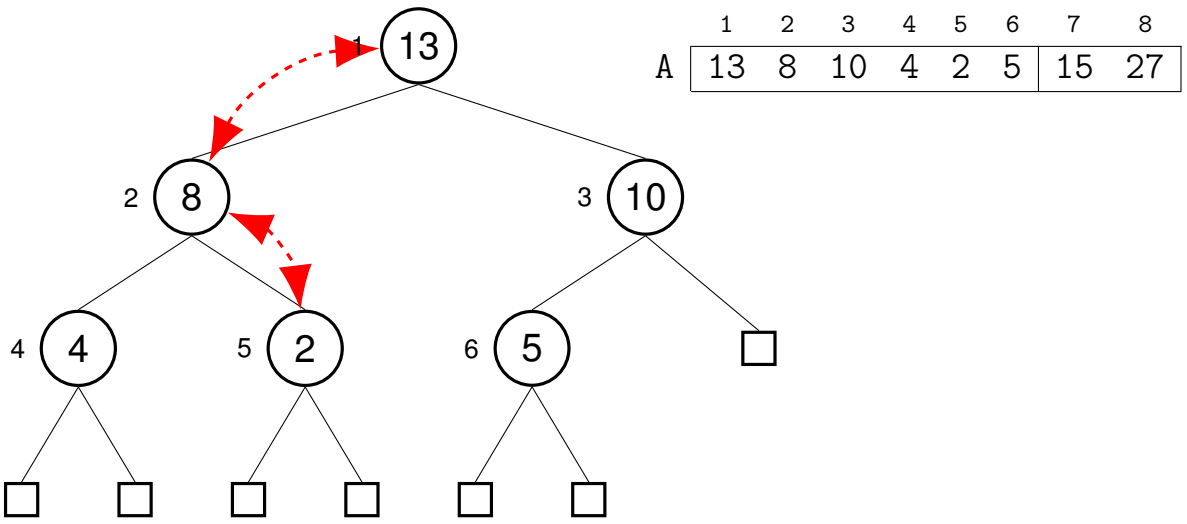


Figura 7.14: Esborrat del màxim del MAXHEAP. Els nodes blancs satisfan la propietat d'un monticle i els vermells no.

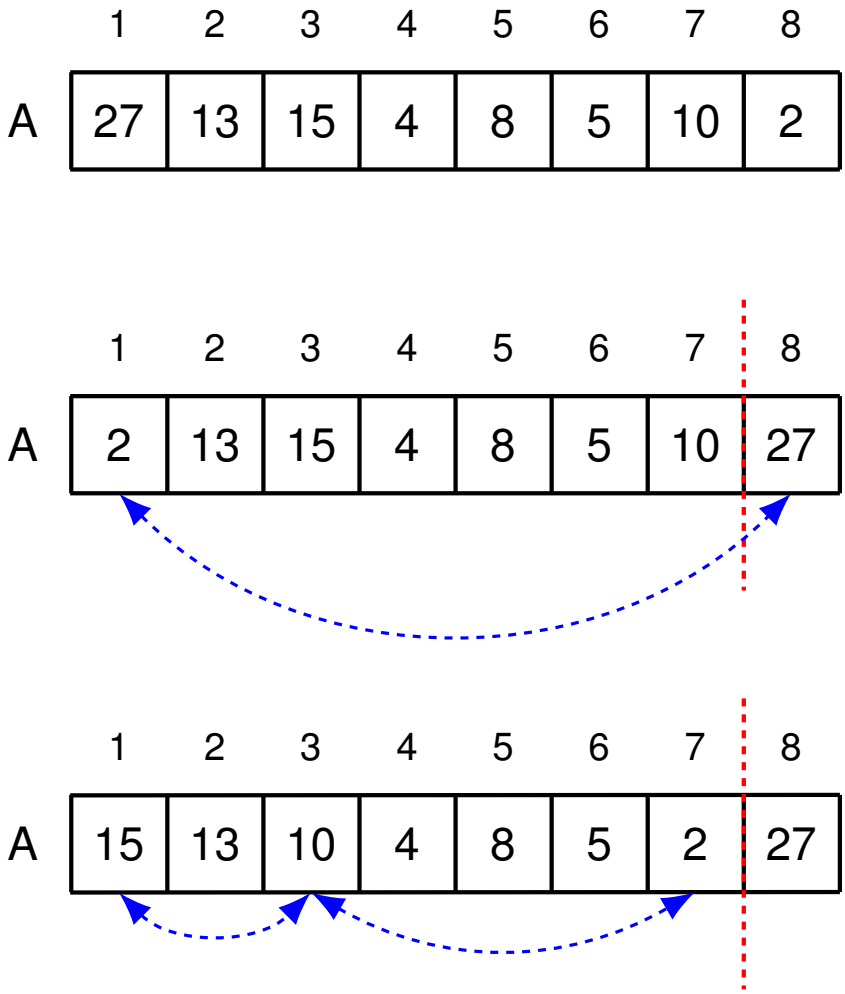


Figura 7.15: Evolució del vector movent el primer màxim.

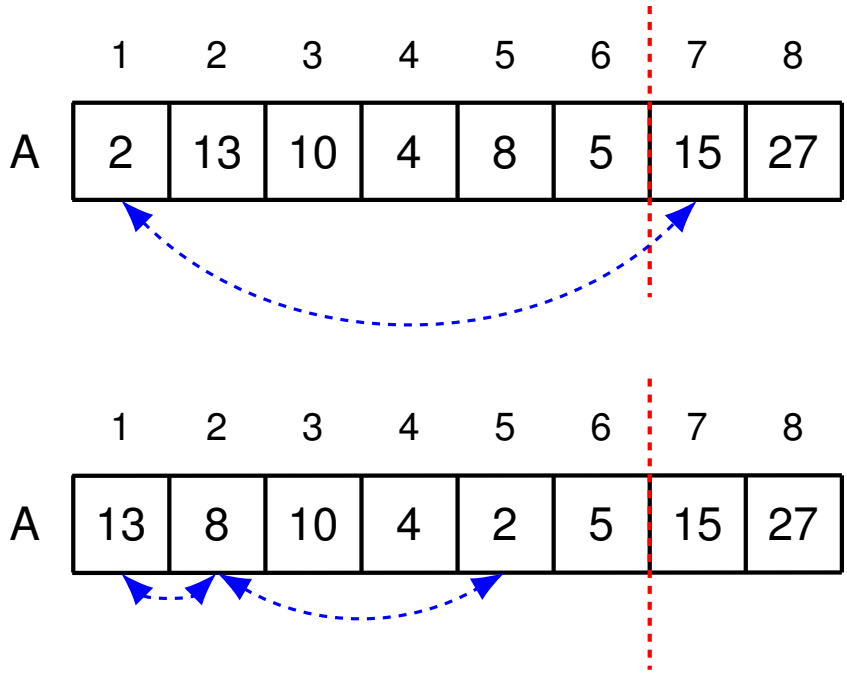


Figura 7.16: Evolució del vector movent el segon màxim.

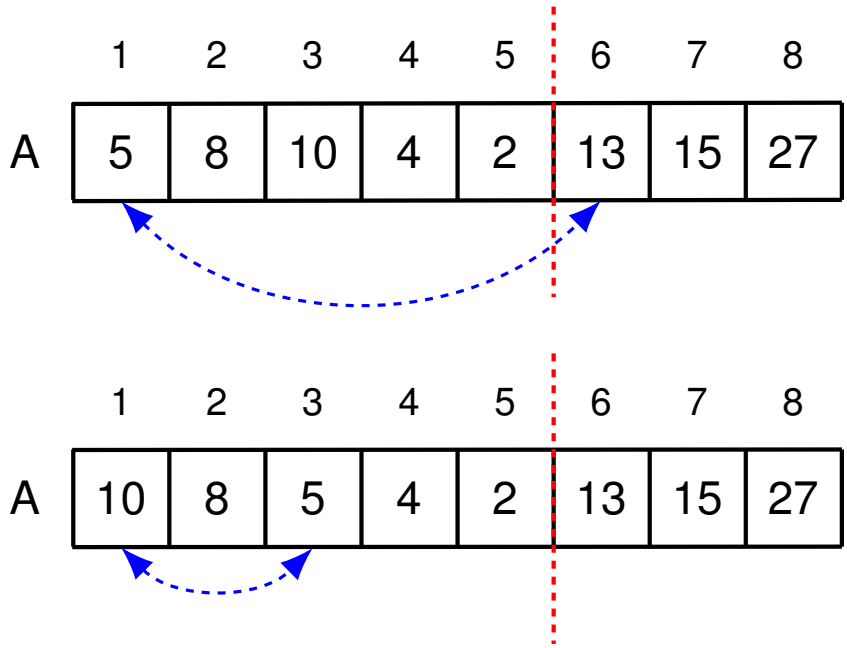


Figura 7.17: Evolució del vector movent el tercer màxim.

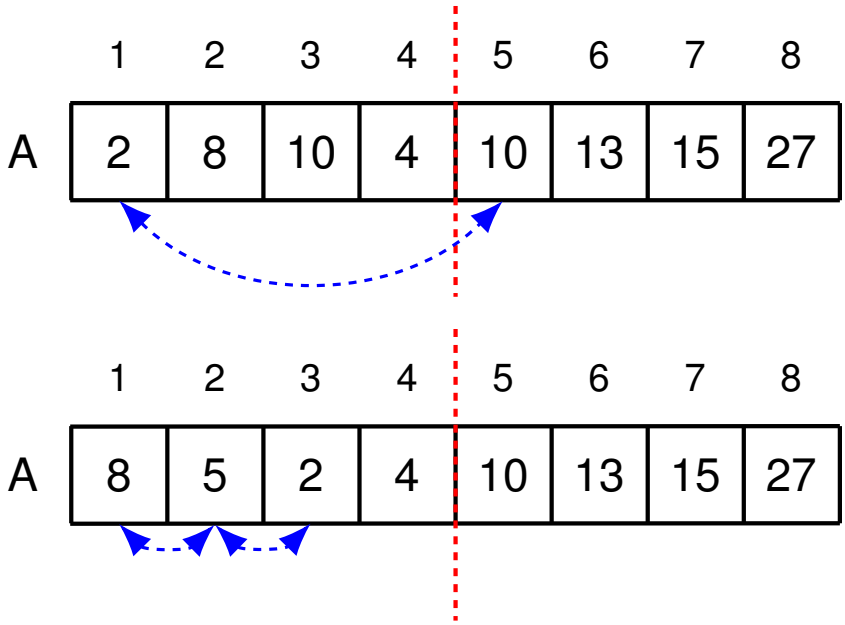


Figura 7.18: Evolució del vector movent el quart màxim.

7.6.3 Implementació

7.6.3.1 Usant la classe CuaPrio

```
template <typename T>
void heap_sort (T A[], nat n) throw(error){
  CuaPrio<T, T> c;
  for (nat i=0; i < n; ++i) { // crea el heap
    c.inserir(A[i], A[i]);
  }
  for (nat i=n-1; i >= 0; --i) {
    A[i] = c.max();
    c.elim_max();
  }
}
```

7.6.3.2 Sense usar la classe CuaPrio

```
template <typename T>
void heapsort (T A[], nat n) {
  // Dóna estructura de max-heap al vector A;
  // aquí cada element s'identifica com la seva prioritat.
  for (nat i=n/2; i>0; --i) {
    enfonsar (A, n, i);
  }
  nat i=n;
  while (i > 0) {
    swap(A[1], A[i]);
    --i;
    // Es crida recursivament enfonsar per enfonsar el
    // nou màxim
    enfonsar(A, i, 1);
  }
}
```

Enfonsa el node j-èssim fins a restablir l'ordre del monticle a `A`; els subarbres del node j són monticles.
Cost : $\Theta(\log_2(\frac{n}{j}))$.

```
template <typename T>
void enfonsar (T A[], nat n, nat j) {
  bool fi = false;
  while ((2*j <= n) and not fi) {
    nat hj = 2*j;
    if (hj < n and A[hj].second > A[hj+1].second) {
      ++hj;
    }
    if (A[j].second > A[hj].second) {
      swap(A[j], A[hj]);
      j = hj;
    }
    else {
      fi = true;
    }
  }
}
```

7.6.4 Cost

El cost d'aquest algorisme és $\Theta(n \log n)$ (fins i tot en el cas pitjor, si tots els elements són diferents).

A la pràctica el seu cost és superior al de *quicksort*, donat que el factor constant multiplicatiu del terme $n \log n$ és més gran.

8
Grafs

Si la depuració és el procés d'eliminar errors, llavors la programació hauria de ser el procés d'introduir-los.

Edsger W. Dijkstra (1930-2002)

8.1 Introducció

- Un **graf** s'utilitza per representar relacions arbitràries entre objectes del mateix tipus, implementant el concepte matemàtic de graf.
- Els objectes reben el nom de **vèrtexs** o nodes i les relacions entre ells s'anomenen **arestes** o arcs.
- Un graf G format pel conjunt finit de vèrtexs V i pel conjunt d'arestes A, es denota pel parell G=(V,A).
- Els grafs es poden classificar en:
  - **dirigits** i **no dirigits** depenent de si les arestes estan orientades o no ho estan.
  - **etiquetats** i **no etiquetats** en funció de si les arestes tenen o no informació associada.

Exemple: Veure la figura 8.1 per obtenir un exemple gràfic de grafs.

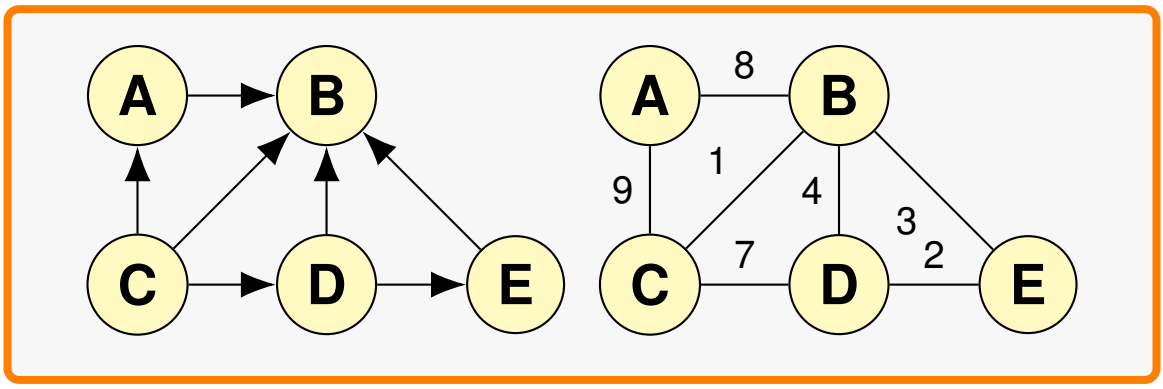


Figura 8.1: Exemples de grafs, *esquerra*: graf dirigit i no etiquetat; *dreta*: graf no dirigit i etiquetat.

8.2 Definicions

Definició 7: Graf

Donat un domini V de vèrtexs i un domini E d'etiquetes, definim un **graf dirigit i etiquetat** G com una funció que associa etiquetes a parells de vèrtexs.

$$G \in \{f : V \times V \to E\}$$

Per a un **graf no etiquetat** la definició és similar; G és una funció que associa un booleà a parells de vèrtexs:

$$G \in \{f : V \times V \to booleà\}$$

8.2.1 Adjacències

8.2.1.1 Adjacències en graf no dirigit

Sigui $G = (V, A)$ un graf NO DIRIGIT. Sigui v un vèrtex de G. Es defineix:

Adjacents de v, $adj(v) = \{v' \in V | (v, v') \in A\}$

Grau de v, $grau(v) = |adj(v)|$

8.2.1.2 Adjacències en graf dirigit

Sigui $G = (V, A)$ un graf DIRIGIT. Sigui v un vèrtex de G. Es defineix:

Successors de v, $succ(v) = \{v' \in V | (v, v') \in A\}$

Predecessors de v, $pred(v) = \{v' \in V | (v', v) \in A\}$

Adjacents de v, $adj(v) = succ(v) \cup pred(v)$

Grau d'entrada de v , $grau\_e(v) = |pred(v)|$

Grau de sortida de v, $grau\_s(v) = |succ(v)|$

Grau de v, $grau(v) = |grau\_s(v) - grau\_e(v)| = ||succ(v)| - |pred(v)||$

8.2.2 Camins

Definició 8: Camí

Un **camí** de longitud $n \geq 0$ en un graf $G = (V, A)$ és una successió $\{v_0, v_1, ..., v_n\}$ tal que:

- Tots els elements de la successió són vèrtexs, és a dir, $\forall i : 0 \leq i \leq n : v_i \in V$
- Existeix aresta entre tot parell de vèrtexs consecutius de la successió, o sigui, $\forall i : 0 \leq i < n : (v_i, v_{i+1}) \in A$.

Donat un camí $\{v_0, v_1, ..., v_n\}$ es diu que:

- Els seus **extrems** són v_0 i v_n i la resta de vèrtexs s'anomenen **intermedis**.
- És **propi** si $n > 0$. Hi ha, com a mínim, dos vèrtexs en la seqüència i, per tant, la seva longitud és ≥ 1.
- És **obert** si $v_0 \neq v_n$.
- És **tancat** si $v_0 = v_n$.

- És **simple** si no es repeteixen arestes.
- És **elemental** si no es repeteixen vèrtexs, excepte potser els extrems. Tot camí elemental és simple (si no es repeteixen els vèrtexs segur que no es repeteixen les arestes).

Definició 9: Cicle elemental

Un **cicle elemental** és un camí tancat, propi i elemental, és a dir, una seqüència de vèrtexs, de longitud major que 0, en la que coincideixen els extrems i no es repeteixen ni arestes ni vèrtexs.

8.2.3 Connectivitat

8.2.3.1 Connectivitat en graf no dirigit

Sigui $G = (V, A)$ un graf NO DIRIGIT. Es diu que:

- És **connex** si existeix camí entre tot parell de vèrtexs.
- És un **bosc** si no conté cicles.
- És un **arbre no dirigit** si és un bosc connex.

Exemple: Veure la figura 8.2 per tenir alguns exemples gràfics de graf connex, bosc i arbre no dirigit.

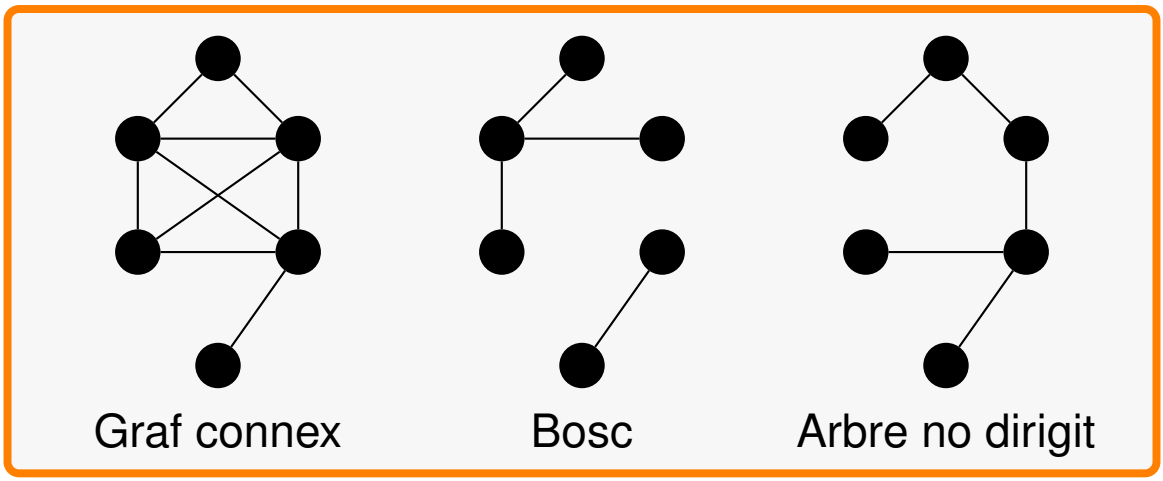


Figura 8.2: Exemple de graf connex, bosc i arbre no dirigit d'un graf no dirigit.

8.2.3.2 Connectivitat en graf dirigit

Sigui $G = (V, A)$ un graf DIRIGIT. Es diu que:

- És **fortament connex** si existeix camí entre tot parell de vèrtexs en ambdós sentits.
- Un **subgraf** del graf $G = (V, A)$, és el graf $H = (W, B)$ tal que $W \subseteq V$ i $B \subseteq A$ i $B \subseteq W \times W$.
- Un **subgraf induït** del graf $G = (V, A)$ és el graf $H = (W, B)$ tal que $W \subseteq V$ i B conté aquelles arestes de A tal que els seus vèrtexs pertanyen a W.
- Un **arbre lliure** del graf $G = (V, A)$ és un subgraf d'ell, $H = (W, B)$, tal que és un arbre no dirigit i conté tots els vèrtexs de G, és a dir, $W = V$.

Exemples: Veure la figura 8.3 per tenir alguns exemples gràfics de subgraf, subgraf induït i arbre lliure.

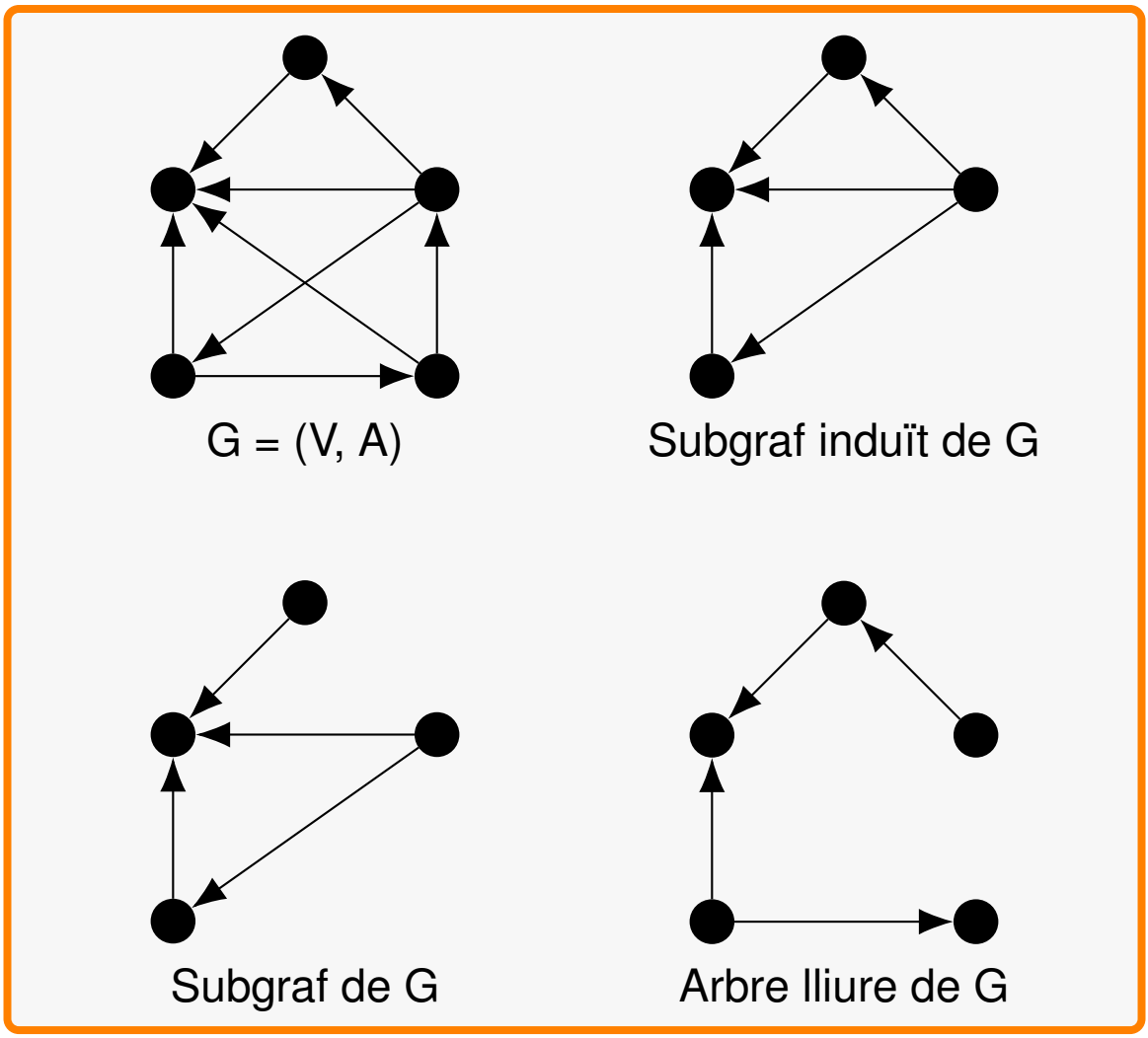


Figura 8.3: Exemples de subgraf, subgraf induït i arbre lliure d'un graf dirigit.

8.2.4 Alguns grafs particulars

8.2.4.1 Graf complet

Definició 10: Graf complet

Un graf $G = (V, A)$ és **complet** si existeix una aresta entre tot parell de vèrtexs de V.

- El nombre d'arestes, $|A|$, d'un *graf complet dirigit* és $|A| =$

$n \cdot (n-1)$; i d'un *graf complet no dirigit* és $|A| = n \cdot \frac{n-1}{2}$.

8.2.4.2 Graf eularià

Definició 11: Graf eulerià

Un graf $G = (V, A)$ és **eulerià** si existeix un camí tancat, de longitud major que zero, simple (no es repeteixen arestes) però no necessàriament elemental, que inclogui totes les arestes de G.

Lemma 8.1. *Un graf no dirigit i connex és eulerià si i només si el grau de tot vèrtex és parell.*

Lemma 8.2. *Un graf dirigit i fortament connex és eulerià si i només si el grau de tot vèrtex és zero.*

Exemples: Veure la figura 8.4 per tenir alguns exemples gràfics de graf connex, bosc i arbre no dirigit.

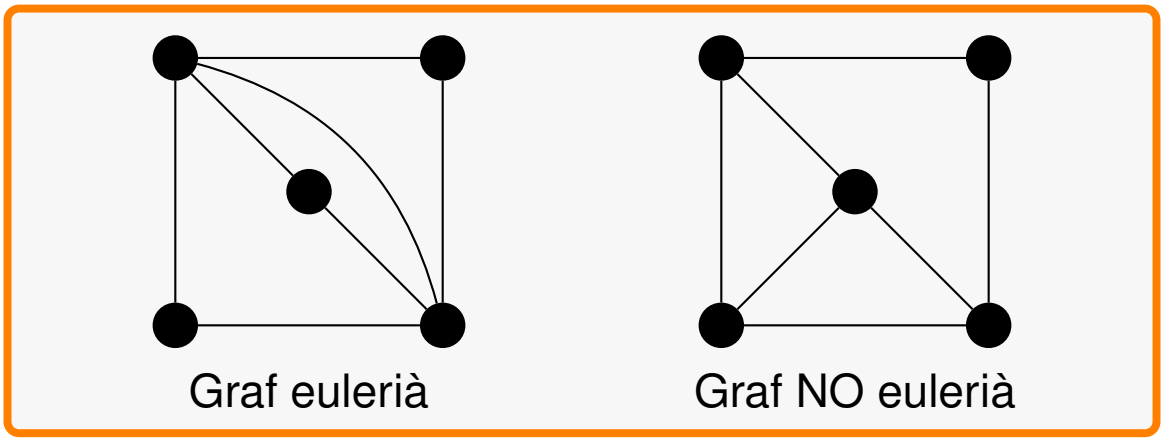


Figura 8.4: Exemples de graf eularià i no eularià.

8.2.4.3 Graf hamiltonià

Definició 12: Graf hamiltonià

Un graf $G = (V, A)$ és **hamiltonià** si existeix un camí tancat i elemental (no es repeteixen vèrtexs) que conté tots els vèrtexs de G. Si existeix, aquest camí es diu **circuit hamiltonià**.

Exemples: Veure la figura 8.5 per tenir alguns exemples gràfics de grafs hamiltonians i no hamiltonians.

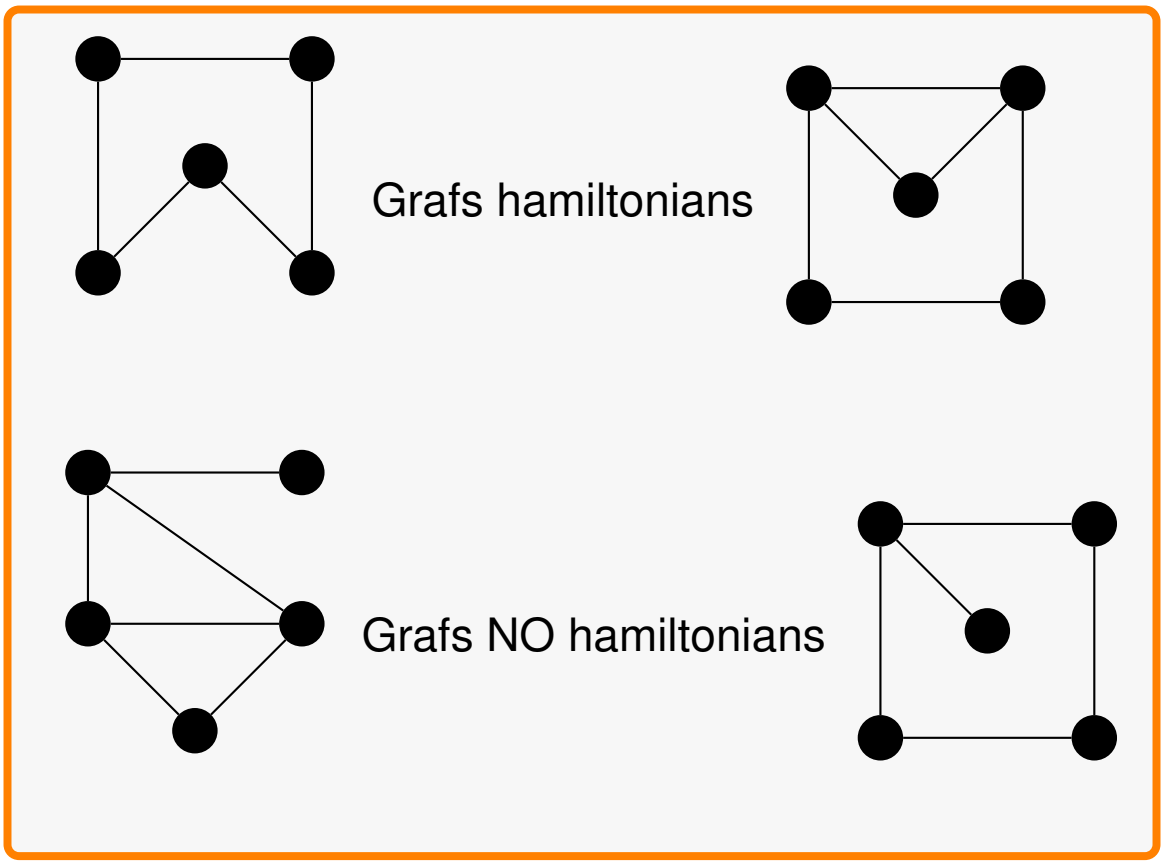


Figura 8.5: Exemples de grafs hamiltonians i no hamiltonians.

8.3 Especificació de la classe graf

Les operacions bàsiques d'un graf són aquelles que ens permeten afegir, consultar i eliminar vèrtexs i arestes.

La classe que es presenta a continuació implementa un **graf dirigit i etiquetat**, en cas que es volgués implementar un altre tipus de graf alguns dels mètodes no serien necessaris o variarien (p. ex.: si el graf fos no dirigit no caldrien els mètodes `successors` o `predecessors`).

IMPORTANT!!
Les classes *V* i *E* amb que s'instancia la classe graf han de tenir definides la constructora per còpia, assignació, la destructora i els operadors de comparació ==, !=.

```
template <class V, class E>
class graf {
  public:
```

Generadores: constructora, afegeix vèrtex i afegeix aresta. En el cas del mètode `afegeix_aresta` genera un error en el cas que algun dels vèrtexs indicat no existeixi o si els dos vèrtexs són iguals (no es permet que una aresta tingui el mateix origen i destí).

```
    graf() throw(error);
    void afegeix_vertex(const V &v) throw(error);
    void afegeix_aresta(const V &u, const V &v, const E &e)
```

```
        throw(error);
```

Tres grans.

```
graf(const graf &g) throw(error);
graf& operator=(const graf &g) throw(error);
~graf() throw();
```

Consultores: valor de l'etiqueta d'una aresta, existeix vèrtex, existeix aresta i vèrtexs successors d'un donat. En el cas dels mètodes `valor`, `existeix_aresta` i `adjacents` generen un error si algun dels vèrtexs indicat no existeix.

```
E valor(const V &u, const V &v) const throw(error);
bool existeix_vertex(const V &v) const throw();
bool existeix_aresta(const V &u, const V &v) const
      throw(error);
void adjacents(const V &v, list<V> &l) const
      throw(error);
```

Consultores: obté una llista amb els vèrtexs successors o predecessors del vèrtex indicat. En el cas que el vèrtex no tingui successors o predecessors la llista serà buida. Aquests mètodes generen un error si el vèrtex indicat no existeix.

```
void successors(const V &v, list<V> &l) const
      throw(error);
void predecessors(const V &v, list<V> &l) const
      throw(error);
```

Operacions modificadores: elimina vèrtex i elimina aresta. En cas que el vèrtex o l'aresta no existeixi no fa res.

```
  void elimina_vertex(const V &v) throw();
  void elimina_aresta(const V &u, const V &v) throw();
```

Gestió d'errors.

```
  const static int VertexNoExisteix = 800;
  const static int MateixOrigeniDesti = 801;

  typedef aresta pair<V, V>;
  typedef vertexs diccRecorrible<V, unsigned int>;

private:
```

Cada objecte de la classe graf ha d'emmagatzemar els vèrtexs i les arestes. Per emmagatzemar els vèrtexs es pot usar un conjunt que es pogués recórrer. Però atès que ens interesa poder identificar amb un enter diferent cada vèrtex (és necessari per guardar les arestes) es farà servir un diccionari recorrible. D'aquesta manera es poden inserir, consultar i eliminar vèrtexs de manera eficient.

```
  diccRecorrible<V, unsigned int> _verts;
  ...
};
```

Per emmagatzemar les relacions entre vèrtexs (les arestes) es poden usar diverses representacions que veurem en la següent secció.

L'especificació de la classe `diccRecorrible` es pot veure a continuació:

```
template <typename Clau, typename Valor>
class diccRecorrible {
  public:
  typedef pair<Clau, Valor> pair_cv;
  ...
```

Iterador del diccionari amb els mètodes habituals (veure l'iterador de la classe llista per més informació).

```
  friend class iterador {
    public:
    friend class diccRecorrible;

    iterador();
    ...
```

Accedeix al parell clau-valor apuntat per l'iterador.

```
    pair_cv operator*() const throw (error);
```

Pre i post-increment; avança l'iterador.

```
    iterador& operator++() throw();
    iterador operator++(int) throw();
```

Operadors de comparació.

```
    bool operator==(const iterador &it) const throw();
    bool operator!=(const iterador &it) const throw();
    ...
  };

  iterador begin() const throw();
  iterador end() const throw();
};
```

8.4 Representacions de grafs

Donat un graf $G = (V, A)$ denotem per:

- n=|V| el nombre de vèrtexs o nodes del graf.
- a=|A| el nombre d'arestes del graf.

8.4.1 Matriu d'adjacència

Per emmagatzemar la informació de les arestes es pot fer servir una matriu.

Si el graf $G = (V, A)$ és no etiquetat s'implementa en una matriu de booleans $M[0..n-1, 0..n-1]$ de forma que ($\forall v, w \in V : M[v, w] = CERT \Leftrightarrow (v, w) \in A$).

Si el graf és etiquetat serà necessària una matriu del tipus de les etiquetes del graf.

> En els exemples gràfics de les representacions de matrius suposem que els vèrtexs s'han inserit en el graf en ordre alfabètic. Per tant la posició 0 li correspon a l'A, la 1 a la B, etc.
> Per tal que es vegi més clarament s'indica en les files el nom del vèrtex que pertany a cada posició.

Exemple: Veure la figura 8.7 per tenir un exemple gràfic de matriu d'adjacència. Aquesta matriu d'adjacència és la que té associat el graf no dirigit i etiquetat de la figura 8.6.

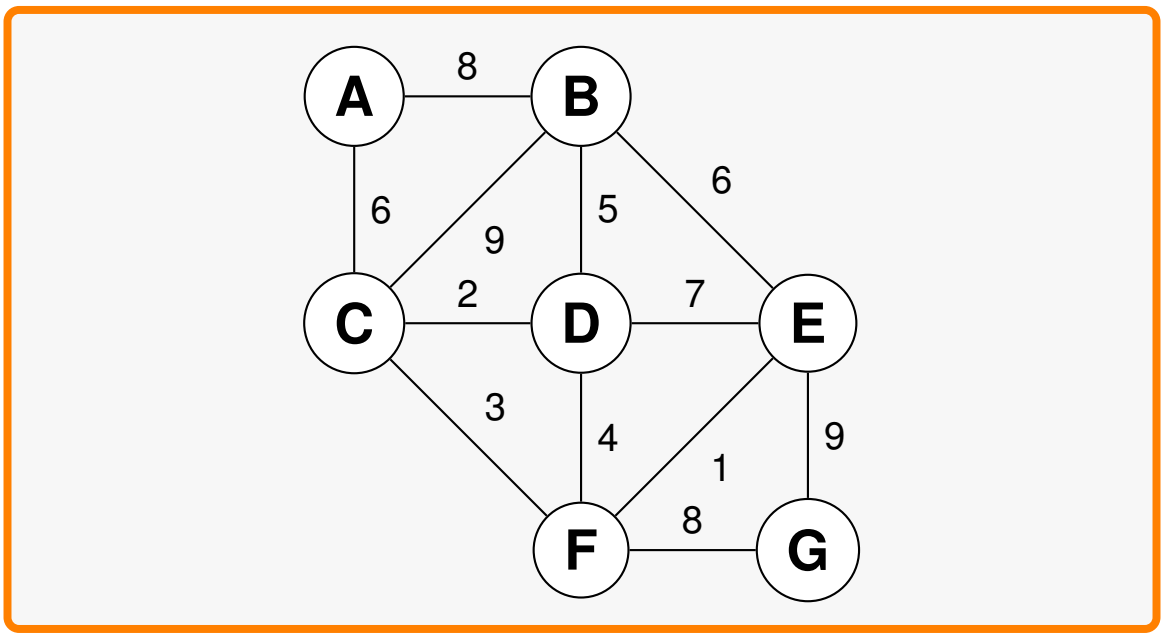


Figura 8.6: Graf no dirigit i etiquetat usat com a exemple de la representació amb una matriu d'adjacències.

		0	1	2	3	4	5	6
A	0		8	6				
B	1			9	5	6		
C	2				2		3	
D	3					7	4	
E	4						1	9
F	5							8
G	6							

Figura 8.7: Representació amb una matriu d'adjacències del graf de la figura 8.6 (només es es mostren les caselles que s'utilitzaran).

8.4.1.1 Cost de la matriu d'adjacència

En general, si el nombre d'arestes del graf és elevat, les matrius d'adjacència tenen bons costos espacials i temporals per a les operacions habituals.

El cost temporal de les operacions bàsiques del graf són:

- Crear el graf (constructora) és $\Theta(n^2)$.
- Afegir aresta (`afegeix_aresta`), existeix aresta (`existeix_aresta`), eliminar aresta (`elimina_aresta`), valor d'una etiqueta (`valor`) són $\Theta(1)$.
- Adjacents a un vèrtex (adjacents) és $\Theta(n)$.
- Per a un graf dirigit el cost de calcular els successors (`successors`) o els predecessores (`predecessors`) d'un vèrtex donat és $\Theta(n)$.

El cost espacial, és a dir, l'espai ocupat per la matriu és de l'ordre de $\Theta(n^2)$:

- Un graf no dirigit només necessita la meitat de l'espai $\frac{n^2-n}{2}$, doncs la matriu d'adjacència és simètrica i tan sols cal guardar la informació de l'etiqueta a la part triangular superior (o inferior).
- Un graf dirigit necessita tot l'espai de la matriu exceptuant la diagonal, és a dir, $n^2 - n$.

En cas que el nombre d'arestes no sigui massa elevat s'estarà desaprofitant força espai, i per tant aquesta representació no seria adequada.

8.4.2 Llista d'adjacència

Aquesta estructura de dades emmagatzema per cada vèrtex del graf una llista amb els vèrtexs adjacents a aquest.

L'estructura que s'utilitza per a implementar un graf $G = (V, A)$ és un vector $L[0..n-1]$ i a cada element del vector $L[i]$, amb $0 \leq i \leq n-1$, hi ha una llista encadenada formada pels vèrtexs que són adjacents (successors, si el graf és dirigit) al vèrtex amb identificador i.

Exemple: Veure la figura 8.9 per tenir un exemple gràfic de llista d'adjacència. Aquesta llista d'adjacència és la que té associat el graf dirigit i etiquetat de la figura 8.8.

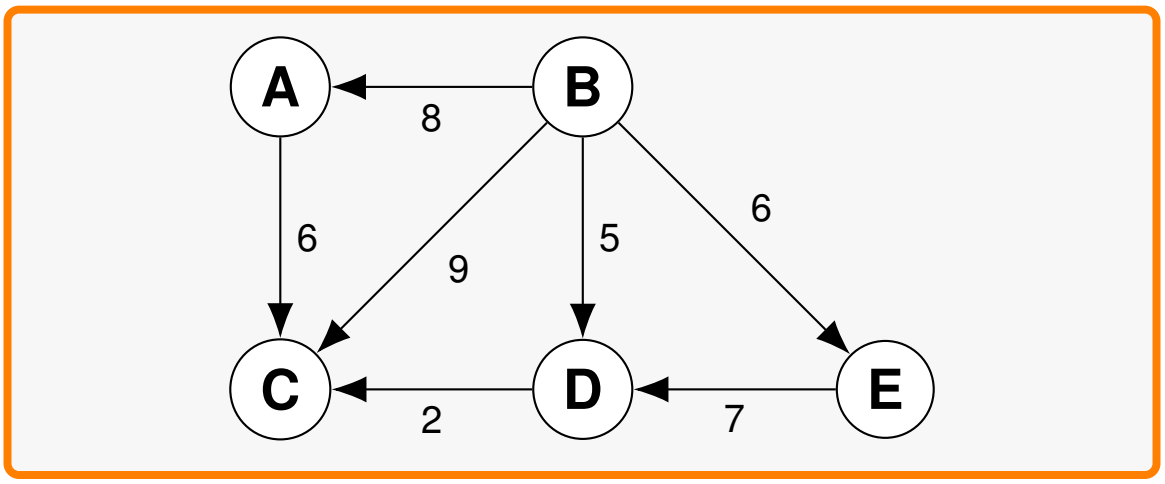


Figura 8.8: Graf dirigit i etiquetat usat com a exemple de la representació amb una llista d'adjacències.

Suposem que els vèrtexs s'han inserit en el graf en ordre alfabètic, per tant la posició 0 li correspon a l'A, la 1 a la B etc. Per tal que es vegi més clarament s'indica el nom del vèrtex que pertany a cada posició.

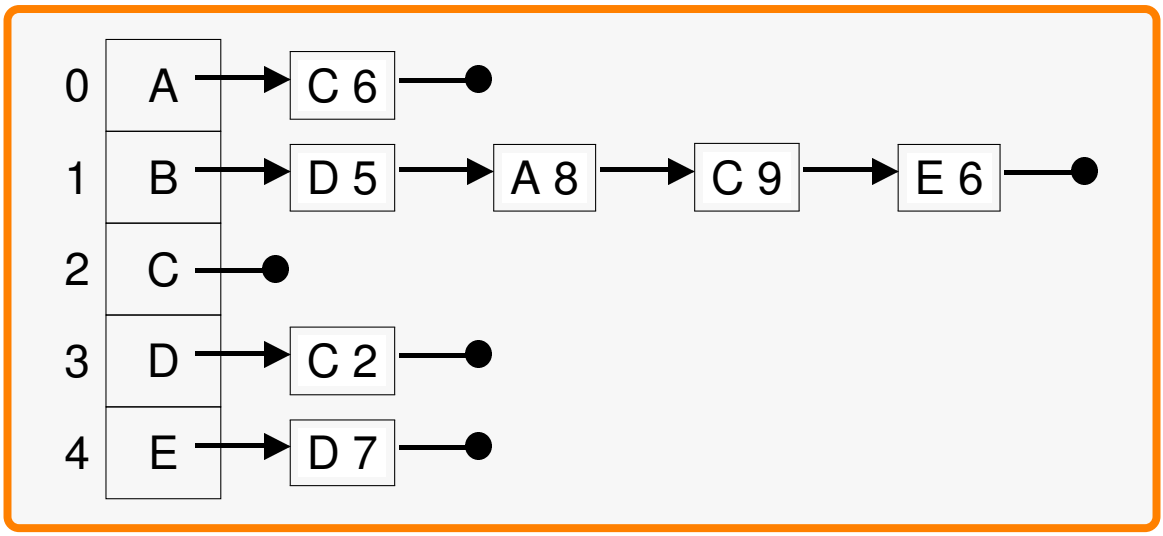


Figura 8.9: Representació amb una llista d'adjacència del graf de la figura 8.8.

8.4.2.1 Cost de la llista d'adjacència

El cost temporal d'algunes operacions bàsiques és el següent:

- Crear l'estructura (constructora) és $\Theta(n)$.
- Afegir aresta (`afegeix_aresta`), dir si una aresta existeix (`existeix_aresta`) i eliminar aresta (`elimina_aresta`) necessiten comprovar si existeix l'aresta. En el pitjor cas requerirà recórrer la llista completa, que pot tenir una grandària màxima de $n-1$ elements. Per tant, tenen un cost $\Theta(n)$. La implementació de la llista d'arestes en un AVL permetria reduir el cost d'aquesta operació a $\Theta(log(n))$.
- El cost de les operacions que calcula els successors (`successors`) i predecessors (`predecessors`) d'un vèrtex donat en un graf DIRIGIT és el següent:
  - Per obtenir els successors d'un vèrtex només cal recórrer tota la seva llista associada, per tant, $\Theta(n)$.

– Per obtenir els predecessors s'ha de recórrer, en el pitjor cas, tota l'estructura per veure en quines llistes apareix el vèrtex del que s'estan buscant els seus predecessors, per tant, $\Theta(n+a)$. Si les llistes de successors d'un vèrtex estan implementades en un AVL llavors el cost de l'operació predecessors és $\Theta(n \cdot log(n))$.

El cost espacial d'aquesta implementació és de l'ordre $\Theta(n+a)$.

8.4.3 Multillista d'adjacència

La implementació d'un graf usant multillistes només té sentit per grafs dirigits. El seu objectiu és millorar el cost de l'operació que obté els predecessores per que passi a tenir cost $\Theta(n)$ en lloc del cost $\Theta(n+a)$. S'aconsegueix, a més a més, que el cost de la resta d'operacions sigui el mateix que el que es tenia utilitzant llistes.

Exemple: Veure la figura 8.10 per tenir un exemple gràfic de multillista d'adjacència. Aquesta multillista d'adjacència és la que té associat el graf dirigit i etiquetat de la figura 8.8.

En general, per implementar un graf és convenient utilitzar llistes d'adjacència quan:

- El graf és poc dens.
- El problema a resoldre ha de recórrer totes les arestes.

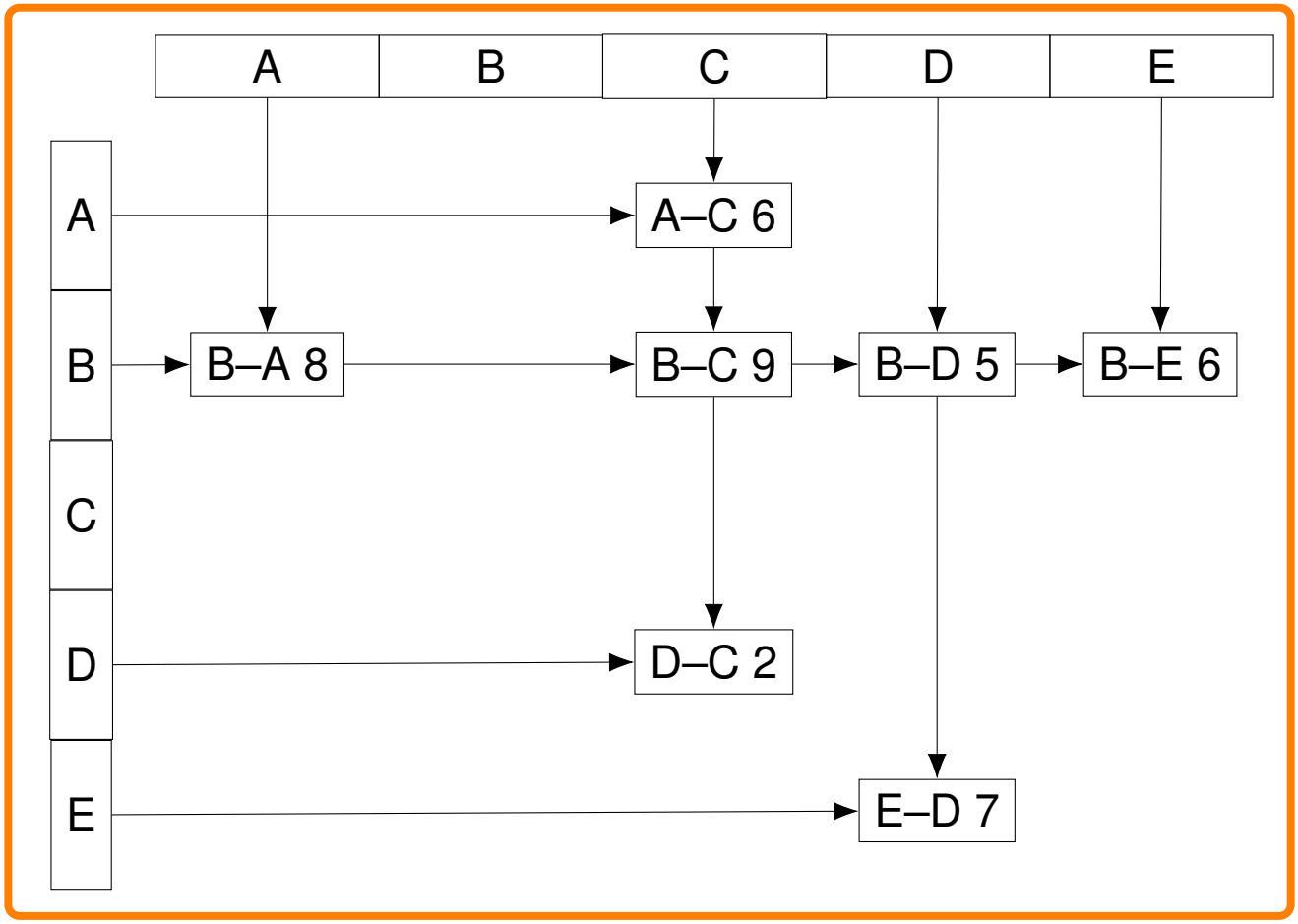


Figura 8.10: Representació amb una multillista d'adjacències del graf de la figura 8.8.

8.5 Recorreguts sobre grafs

8.5.1 Recorregut en profunditat

L'algorisme de **recorregut en profunditat** o profunditat prioritària (anglès: *depth-first search* o *DFS*), es caracteritza perquè permet recórrer completament el graf, tots els vèrtexs i totes les arestes.

- Si el graf és no dirigit, es passa 2 cops per cada aresta (un en cada sentit).
- Si és dirigit es passa un sol cop per cada aresta.

Els vèrtexs es visiten aplicant el criteri **primer en profunditat**. Primer en profunditat significa que donat un vèrtex v que no hagi estat visitat, DFS primer visitarà v i després, recursivament aplicarà DFS sobre cadascun dels adjacents/successors d'aquest vèrtex (depenent de si el graf és no dirigit o dirigit) que encara no hagin estat visitats.

Existeix una clara relació entre el recorregut en **preordre** d'un arbre i el DFS sobre un graf.
Es pot fer un recorregut simètric al DFS que consisteixi en no visitar un node fins que no hem visitat tots els seus descendents. Seria la generalització del recorregut en **postordre** d'un arbre. S'anomena **recorregut en profunditat cap enrere**.

El recorregut s'inicia començant per un vèrtex qualsevol i acaba quan tots els vèrtexs han estat visitats, per tant l'ordre dels vèrtexs que produeix el DFS no és únic.

> En els exercicis de recorregut que fem, per tal de tenir una solució única s'indicarà per quin vèrtex començar el recorregut i l'ordre a visitar dels adjacents/successors serà alfabètic.

Exemple: Veure la figura 8.11 per tenir un exemple de recorregut en profunditat prioritària i recorregut en profunditat cap enrere.

8.5.1.1 Descripció de l'algorisme

- Necessitem una llista amb els vèrtexs sense visitar. Atès que tenim per cada vèrtex un natural podem usar un `vector` de booleans per indicar si un vèrtex s'ha visitat o no. Inicialment tots els valors d'aquest vector han de ser **false**.
- Quan es processa un vèrtex cal indicar que s'ha 'visitat' posant a **false** la casella indicada del vector. Això significa que s'ha arribat a aquest vèrtex des de *u* (el primer vèrtex explorat) pels camins que menen a ell i que es ve d'un vèrtex que ja havia estat visitat.
- Mai més s'arribarà a aquest vèrtex pel mateix camí, encara que es pugui arribar per altres camins diferents.

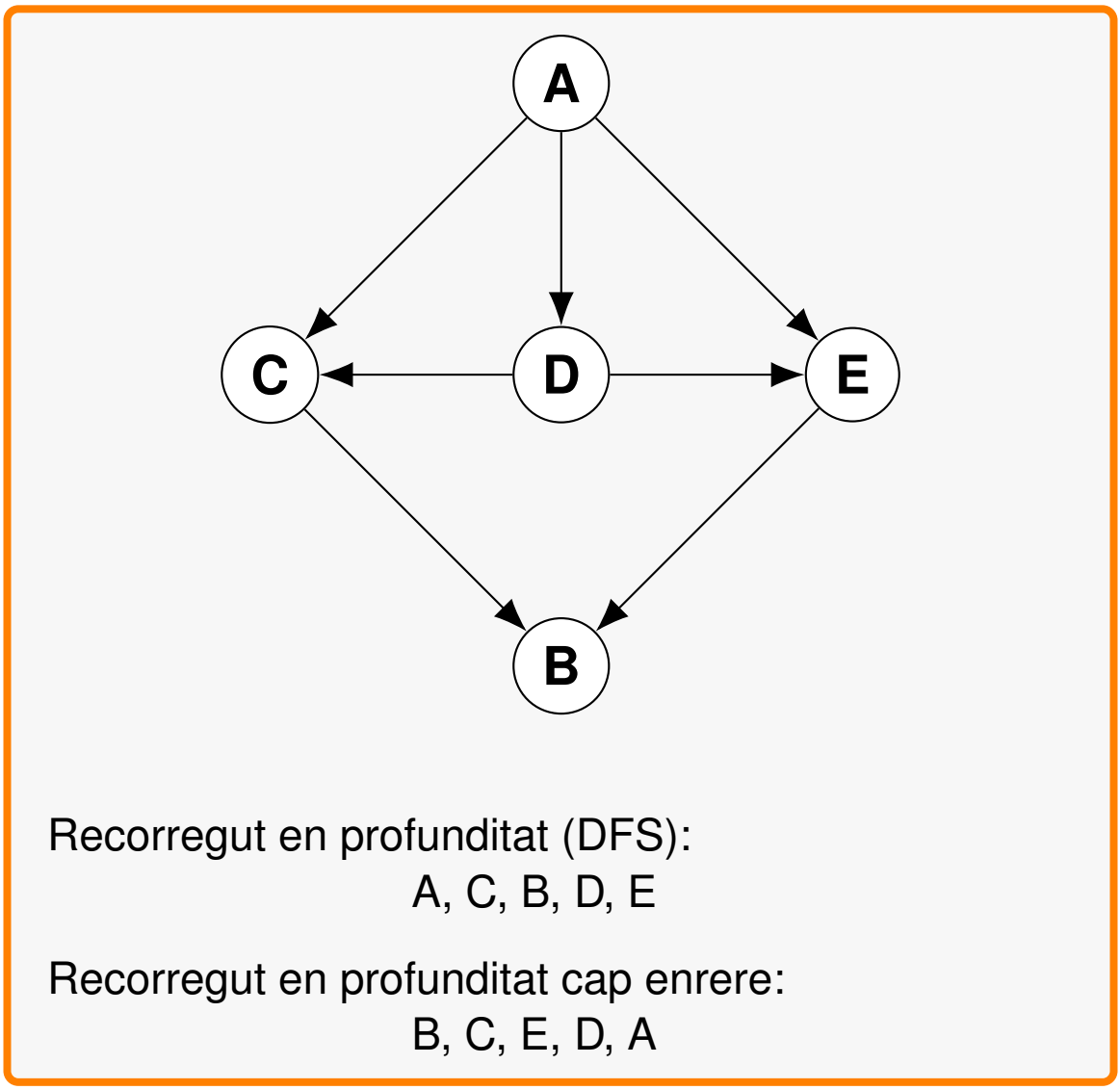


Figura 8.11: Exemple de recorregut en profunditat i en profunditat cap enrere partint del vèrtex A.

8.5.1.2 Implementació

```
template <typename V, typename E>
void graf<V, E>::rec_prof () const throw() {
  vector<bool> visit (_ultim_vertex, false);
  for (vertexs::iterador it = _verts.begin();
        it != _verts.end(); ++it) {
    unsigned int i = (*it).second;
    if (not visit[i]) {
      rec_prof((*it).first, visit);
    }
  }
}

template <typename V, typename E>
void graf<V, E>::rec_prof (const V &u,
        vector<bool> &visit) const throw() {
  unsigned int i = _verts.consulta(u);
  visit[i] = true;

  TRACTAR( u ); /* PREWORK */

  list<V> l;
  adjacents(u, l);
  for (list<V>::iterator v=l.begin(); v!=l.end(); ++v) {
    unsigned int j = _verts.consulta(v);
    if (not visit[j]) {
      rec_prof (*v, visit);
    }
    TRACTAR( u, v ); /* POSTWORK */
  }
```

Hem marcat a visitat tots els vèrtexs del graf accessibles des de '*u*' per camins formats exclusivament per vèrtexs que no havien estat visitats.

```
}
```

8.5.1.3 Cost de l'algorisme

Suposem que *PREWORK* i *POSTWORK* tenen cost constant. Depenent de quina sigui la representació del graf tindrem els següents costos:

- matriu d'adjacència: $\Theta(n^2)$
- llistes d'adjacència: $\Theta(n+a)$ degut a que es recorren totes les arestes dues vegades, una en cada sentit, i a que el bucle exterior recorre tots els vèrtexs.

8.5.2 Recorregut en amplada

L'algorisme de **recorregut en amplada** o amplada prioritària (anglès: *breadth-first search* o *BFS*) generalitza el concepte de **recorregut per nivells** d'un arbre:

- Després de visitar un vèrtex es visiten els successors, després els successors dels successors, i així reiteradament.
- Si després de visitar tots els descendents del primer node encara en queden per visitar, es repeteix el procés començant per un dels nodes no visitats.

8.5.2.1 Descripció de l'algorisme

L'algorisme utilitza una cua per a poder aplicar l'estratègia exposada. Cal marcar el vèrtex com a visitat quan l'encuem per no encuar-lo més d'una vegada. L'acció principal és idèntica a la del DFS.

8.5.2.2 Implementació

```
template <typename V, typename E>
void graf<V, E>::rec_amplada () const throw() {
  vector<bool> visit (_ultim_vertex, false);
  for (vertexs::iterador it = _verts.begin();
    it != _verts.end(); ++it) {
    unsigned int i = (*it).second;
    if (not visit[i]) {
      rec_amplada((*it).first, visit);
    }
  }
}

template <typename V, typename E>
void graf<V, E>::rec_amplada (const V &u,
        vector<bool> &visit) const throw() {
  cua<V> c;
  c.encua(u);
  unsigned int i = _verts.consulta(u)
  visit[i] = true;
  while (not c.es_buida()) {
    V v = c.cap();
    c.desencua();
    TRACTAR(v);
    list<V> l;
    adjacents(v, l);
    for (list<V>::iterator w= l.begin();
        w != l.end(); ++w) {
      unsigned int j = _verts.consulta(w)
      if (not visit[j]) {
        c.encua(w);
        visit[j] = true;
      }
    }
  }
}
```

8.5.2.3 Cost de l'algorisme

Aquest algorisme té el mateix cost que l'algorisme de recorregut en profunditat prioritària:

- matriu d'adjacència: $\Theta(n^2)$
- llistes d'adjacència: $\Theta(n + a)$

8.5.3 Recorregut en ordenació topològica

L'**ordenació topològica** és un recorregut només aplicable a **grafs dirigits acíclics**.

Un vèrtex només es visita si han estat visitats tots els seus predecessors dins del graf. Per tant, en aquest recorregut les arestes defineixen una restricció més forta sobre l'ordre de visita dels vèrtexs.

Exemple: A la figura 8.12 es pot veure el pla d'estudis amb prerequisits entre assignatures.

Quan un alumne es matricula ha de tenir en compte que:

- En començar només pot cursar les assignatures que no tinguin cap prerrequisit (cap predecessor).
- Només pot cursar una assignatura si han estat cursats prèviament tots els seus prerequisits (tots els seus predecessors).

8.5.3.1 Descripció de l'algorisme

- El funcionament de l'algorisme consisteix en anar escollint els vèrtexs que no tenen predecessors (que són els que es poden visitar) i, a mesura que s'escullen, s'esborren les

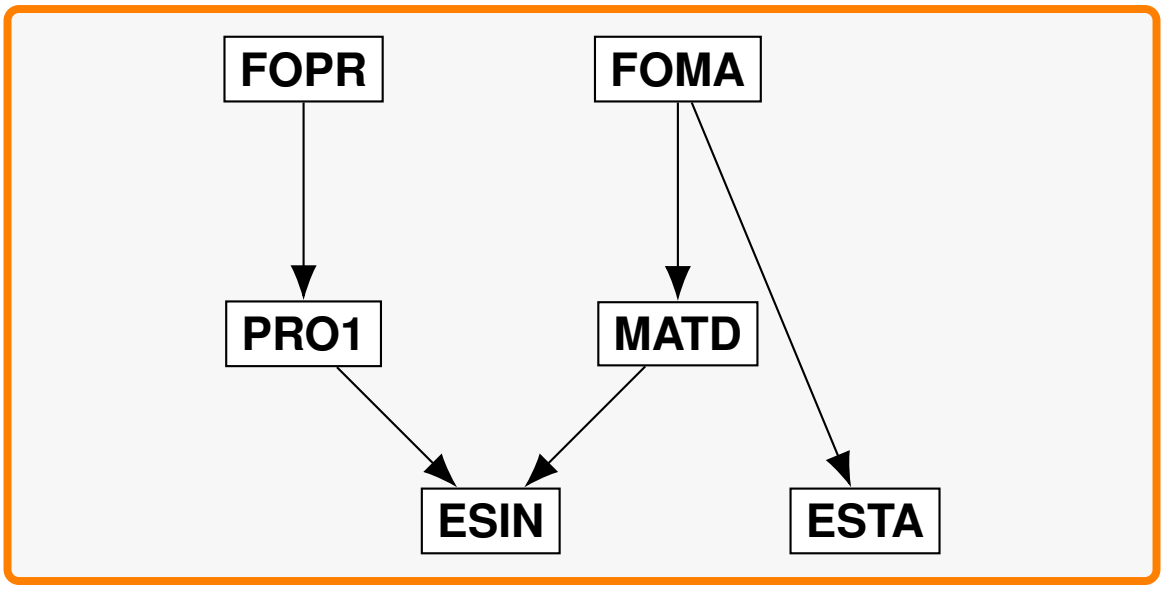


Figura 8.12: Graf que conté els prerrequisits entre assignatures del pla d'estudis d'informàtica.

arestes que en surten. La implementació directa d'això és costosa ja que cal cercar els vèrtexs sense predecessors.

- Una alternativa consisteix en calcular prèviament quants predecessors té cada vèrtex. Per tant cal:
  - un `vector` per emmagatzemar el nombre de predecessors de cada vèrtex. Aquest vector s'ha d'actualitzar sempre que s'incorpori un nou vèrtex a la solució. El vector guarda el nombre de predecessors de cada vèrtex que no són a la solució.
  - una llista (o conjunt) amb els vèrtexs que tinguin 0 predecessors que seran els vèrtexs que anirem processant.

8.5.3.2 Implementació

```
template <typename V, typename E>
void graf::rec_ordenacio_topologica () const throw() {
  // Pre:  el graf és dirigit i acíclic.

  conjunt<V> zeros; // crea el conjunt buit
  nat num_pred = new nat[_verts.size()];

  // emmagatzamem els predecessors de cada vèrtex
  for (vertexs::iterador v = _verts.begin();
        v != _verts.end(); ++v) {
    list<V> lst;
    predecessors(v, l);
    nat i = (*v).second;
    num_pred[i] = lst.size();
    if (lst.size() == 0) {}
      zeros.insereix(v); // afegeix els vèrtexs sense preds.
    }
  }
  while (not zeros.es_buit()) {
    V v = *(zeros.begin());
    zeros.elimina(v);
    TRACTAR( v );
    list<V> lst;
    successors(v, lst);
    for (list<V>::iterator w= lst.begin();
                  w != lst.end(); ++w) {
      nat j = _verts.consulta(*w);
      --num_pred[j];
      if (num_pred[j]==0)
        zeros.insereix(*w);
      }
    }
  }
  delete[] num_pred;
}
```

8.5.3.3 Cost de l'algorisme

El cost temporal és el mateix que el de l'algorisme DFS:

- matriu d'adjacència: $\Theta(n^2)$
- llista d'adjacència: $\Theta(n + a)$

8.6 Connectivitat i ciclicitat

8.6.1 Connectivitat

8.6.1.1 Problema

Determinar si un graf no dirigit és connex o no.

Propietat

Si un graf és connex és possible arribar a tots els vèrtexs del graf començant un recorregut per qualsevol d'ells.

Es necessita un algorisme que recorri tot el graf i que permeti esbrinar a quins vèrtexs es pot arribar a partir d'un donat. Per això utilitzarem un DFS.

8.6.1.2 Descripció de l'algorisme

La solució proposada per aquest problema determina el nombre de components connexes del graf. Si el nombre de components és major que 1 llavors G no és connex.

A més a més, tots els vèrtexs d'una component connexa s'etiqueten amb un natural que és diferent per cadascuna de les components connexes que té el graf inicial.

8.6.1.3 Implementació

```
template <typename V, typename E>
nat graf<V, E>::compo_connexes () const throw() {
  vector<pair<bool, nat> > visit(_ultim_vertex);
    // el boolèa indica si el vèrtex ha estat visitat
    // el natural indica la nombre de la component connexa a
    //     la que pertany el vèrtex

  for (nat i=0; i < _ultim_vertex; ++i) {
    visit[i].first = false;
    visit[i].second = 0;
  }

  nat nc = 0;
  for (vertexs::iterador it = _verts.begin();
        it != _verts.end(); ++it) {
    nat i = (*it).second;
    if (not visit[i]) {
      ++nc;
      connex((*it).first, visit, nc);
    }
  }
  return nc;
}

template <typename V, typename E>
void graf<V, E>::connex (const V &u,
        vector<bool, nat> &lst, nat numc) const throw() {
  /* PREWORK: Anotar el número de component connexa a
        la que pertany el vèrtex. */
  nat i = _verts.consulta(u);
  visit[i].first = true;
  visit[i].second = numc;
  list<V> l;
  adjacents(u, l);
  // v avaluarà a false quan arribem a l'últim vèrtex
  for (list<V>::iterator v = l.begin();
```

```
        v != l.end(); ++v) {
    nat j = _verts.consulta(v);
    if (not visit[j]) {
      connex (*v, visit, numc);
    }
  }
```

Hem visitat tots els vèrtexs del graf accessibles des d'u per camins formats exclusivament per vèrtexs que no estaven visitats, segons l'ordre de visita del recorregut en profunditat. D'aquests vèrtexs a més a més s'ha afegit la informació que indica a quina component connexa pertanyen.

```
}
```

8.6.1.4 Cost de l'algorisme

El cost d'aquest algorisme és idèntic al de recorregut en profunditat prioritària.

8.6.2 Test de ciclicitat

8.6.2.1 Problema

Donat un graf dirigit, determinar si té cicles o no.

Propietat

En un graf amb cicles, en algun moment durant el recorregut s'arriba a un vèrtex vist anteriorment per una aresta que no estava visitada.

8.6.2.2 Descripció de l'algorisme

El mètode que segueix l'algorisme proposat és mantenir, pel camí que va des de l'arrel al vèrtex actual, quins vèrtexs en

formen part. Si un vèrtex apareix més d'una vegada en aquest camí és que hi ha cicle.

8.6.2.3 Implementació

```
template <typename V, typename E>
bool graf<V, E>::cicles () const throw() {
  vector<pair<bool, bool> > visit(_ultim_vertex);
  for (nat i = 0; i < _ultim_vertex; ++i) {
    visit[i].first = false;
    visit[i].second = false; // no hi ha cap vèrtex en el
  } // camí de l'arrel al vèrtex actual

  bool trobat = false; // inicialment no hi ha cicles
  vertexs::iterador it = _verts.begin();
  while (it != _verts.end() and not trobat) {
    nat i = (*it),second;
    if (not visit[i].first) {
      trobat = cicles((*it).first, visit);
    }
    else {
      ++it;
    }
  }
  return trobat;
}

template <typename V, typename E>
bool graf<V, E>::cicles (const V &u,
        vector<bool, bool> &visit) const throw() {
  nat i = _verts.consulta(u);
  visit[i].first = true;

  // PREWORK: Anotem que s'ha visitat u i que aquest es
  // troba en el camí de l'arrel a ell mateix.
  visit[i].second = true;

  bool trobat = false;
```

```
  list<V> l;
  successors(u, l);
  list<V>::iterator v = l.begin();
  while (v != l.end() and not trobat) {
    nat j = _verts.consulta(v);
    if (visit[j].first) {
      if (visit[j].second) {
        // Hem trobat un cicle!  Es recorre l'aresta (u,v)
        // però ja existia camí de vèrtex inicial a u
        trobat = true;
      }
    }
    else {
      trobat = cicles(*v, visit);
    }
    ++v;
  }
  // POSTWORK: Ja s'ha recorregut tota la descendència
  // d'u i, per tant, s'abandona el camí actual
  // des de l'arrel (es va desmuntant el camí).
  visit[i].second = false;

  return trobat;
```

Hem visitat tots els vèrtexs accessibles des d'u per camins formats exclusivament per vèrtexs que no estaven visitats. b indicarà si en el conjunt de camins que tenen en comú des de l'arrel fins u i completat amb la descendència d'u hi ha cicles.

```
}
```

8.6.2.4 Cost de l'algorisme

Aquesta versió té el mateix cost que l'algorisme recorregut en profunditat prioritària.

8.7 Arbres d'expansió mínima

8.7.1 Introducció

Definició 13: Arbre d'expansió mínima

Donat un graf $G = (V, A)$ que és un graf no dirigit, etiquetat amb valors naturals i connex; es defineix un **arbre d'expansió mínima** (anglès: *minimun spaning tree* (MST)) de G com un subgraf de G, $T = (V, B)$ amb $B \subseteq A$, connex i sense cicles, tal que la suma dels pesos de les arestes de T sigui mínima.

Per crear un arbre d'expansió mínima d'un graf es pot usar l'*algorisme de Kruskal* o l'*algorisme de Prim*.

8.7.2 Algorisme de Kruskal

8.7.2.1 Descripció de l'algorisme

L'**algorisme de Kruskal** construeix el MST seguint els següents passos:

1. Es parteix d'un subgraf de G amb tots els vèrtexs de G i cap aresta, és a dir, $T = (V, \emptyset)$.

2. A cada pas s'afegeix una aresta de G a T. S'escull aquella aresta de valor mínim, d'entre totes les que no s'han processat, tal que connecti dos vèrtexs que formen part de dos components connexes diferents (de dos grups diferents de vèrtexs de T).

3. Es repeteix el pas anterior fins que T només té una única component connexa. Llavors el subgraf $T = (V, B)$ és el MST desitjat.

Aquest procés dóna garanties que no es formen cicles i que T sempre és un bosc format per arbres lliures.

Per reduir el cost del procés de selecció de l'aresta de valor mínim, es pot ordenar les arestes de G per ordre creixent del valor de la seva etiqueta.

Exemple: Sigui G el graf de la figura 8.13.

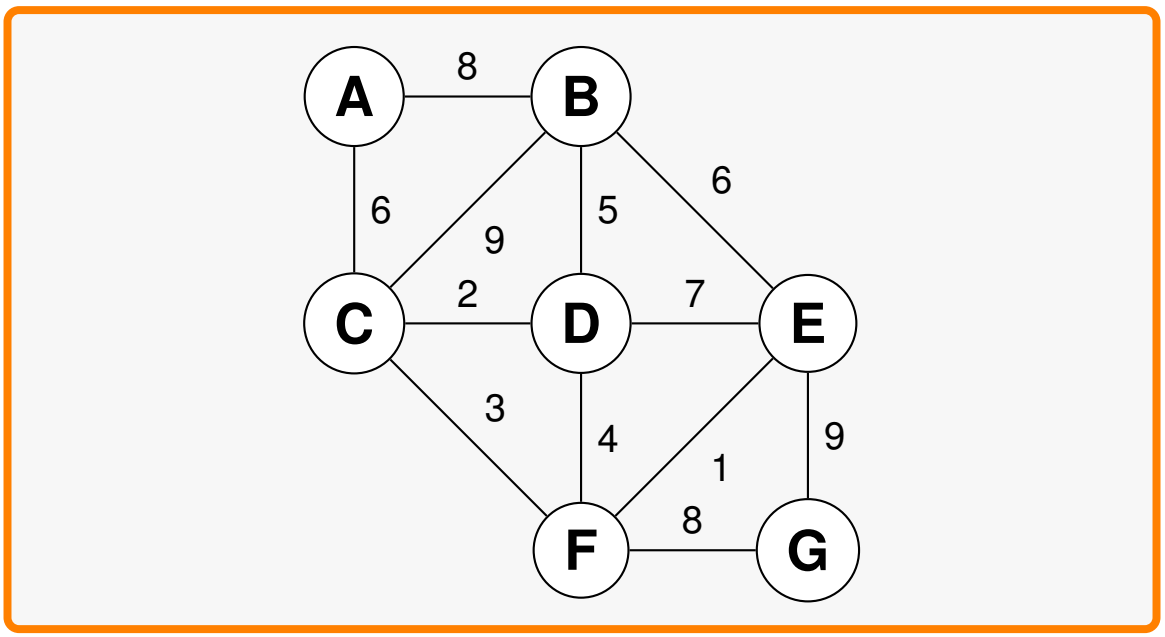


Figura 8.13: Graf no dirigit, etiquetat i connex.

La llista ordenada de les arestes de G és la següent:

1	2	3	4	5	6	6	7	8	8	9	9
E–F	C–D	C–F	D–F	B–D	A–C	B–E	D–E	A–B	F–G	B–C	E–G

La taula 8.1 mostra el procés que es realitza en aplicar l'algorisme de Kruskal sobre el graf G. Inicialment cada vèrtex pertany a un arbre diferent i al final tots formen part del mateix

arbre, el MST. Les arestes examinades i no refusades són les arestes que formen el MST.

Etapa	Aresta examinada	Grups de vèrtexs
inicial	- - -	{A}, {B}, {C}, {D}, {E}, {F}, {G}
1	E–F etiqueta 1	{A}, {B}, {C}, {D}, {E, F}, {G}
2	C–D etiqueta 2	{A}, {B}, {C, D}, {E, F}, {G}
3	C–F etiqueta 3	{A}, {B}, {C, D, E, F}, {G}
4	D–F etiqueta 4	Aresta refusada (forma cicle)
5	B–D etiqueta 5	{A}, {B, C, D, E, F}, {G}
6	A–C etiqueta 6	{A, B, C, D, E, F}, {G}
7	B–E etiqueta 6	Aresta refusada (forma cicle)
8	D–E etiqueta 7	Aresta refusada (forma cicle)
9	A–B etiqueta 8	Aresta refusada (forma cicle)
10	F–G etiqueta 8	{A, B, C, D, E, F, G}

Taula 8.1: Passos en aplicar l'algorisme de Kruskal en el graf de la figura 8.13.

L'arbre d'expansió mínima resultant es pot veure a la figura 8.14.

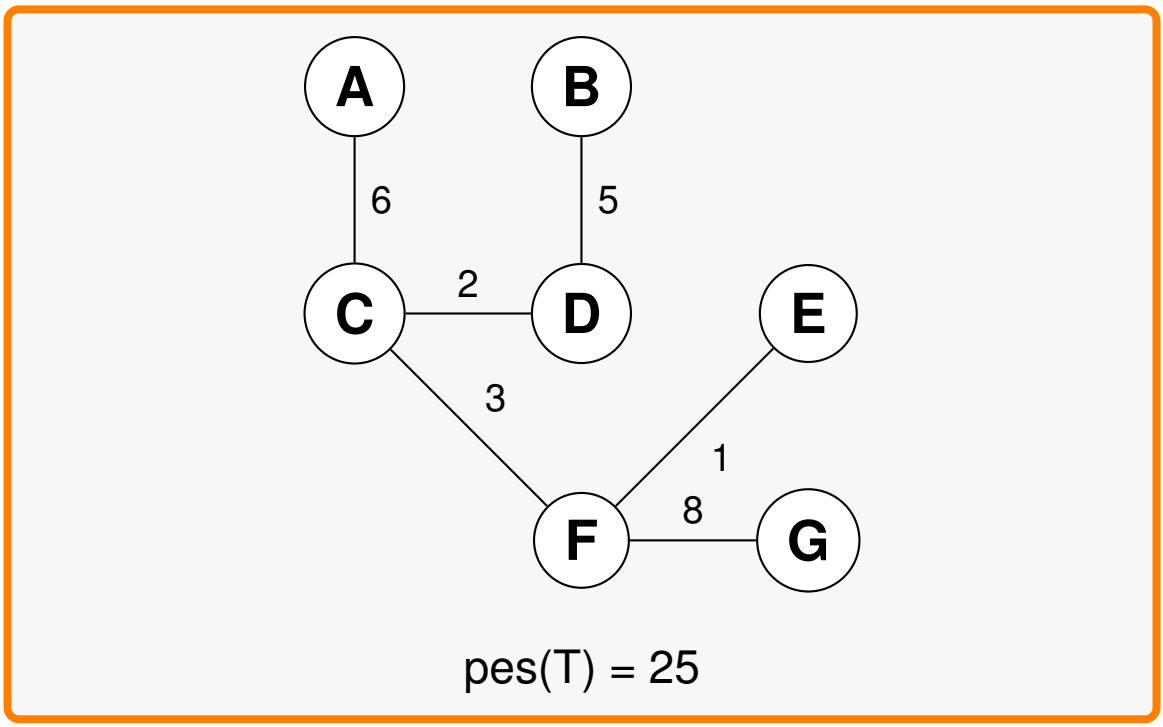


Figura 8.14: Arbre d'expansió mínima resultat d'aplicar l'algorisme de Kruskal sobre el graf 8.13.

8.7.2.2 Implementació

- En la implementació de l'algorisme de Kruskal, s'utilitza la classe MFSet per mantenir els diferents grups de vèrtexs (blocs de la partició).

  Aquesta classe està definida de la següent forma:

```
class MFSet {
  public:
  typedef unsigned int nat;
```

Crea un MFSet format per n grups que etiquetarem del 0 a $n-1$.

```
  explicit MFSet(nat n) throw(error);
```

Tres grans.

```
  MFSet(const MFSet &mf) throw(error);
  MFSet& operator=(const MFSet &mf) throw(error);
  ~MFSet() throw();
```

Uneix els dos grups als quals pertanyen x i y. Genera un error si x o y és més gran o igual que el nombre inicial de grups.

```
  void unir(nat x, nat y) throw(error);
```

Retorna el representant del grup al qual pertany x. El representant d'un grup és l'element més petit del grup. Genera un error si x és més gran o igual que el nombre inicial de grups.

```
  nat find(nat x) throw(error);
```

Gestió d'errors.

```
  const unsigned int ElementMesGranMax = 900;
};
```

- La classe graf cal que tingui un mètode anomenat arestes que retorni totes les arestes del graf.
- Suposem que $G = (V, A)$ és un graf no dirigit, connex i etiquetat amb valors naturals.

```
template <typename V, typename E>
void graf<V, E>::kruskal (conjunt<aresta> &T)
        const throw() {
  // Pre:  el graf és no dirigit, connex i etiquetat amb valors
  //     naturals.

  MFSet MF(_verts.size());

  // Una aresta està definida com un pair de vèrtexs.
  list<aresta> AO = arestes();
  ordenar_creixentment(AO);

  list<aresta>::iterator it = begin();
  while (T.size() < _verts.size()-1) {
    aresta p = *it; // p=(u,v) és l'aresta de valor mínim
    nat u = _verts.consulta(p.first());
    nat v = _verts.consulta(p.second());
    nat repr_u = MF.find(u);
    nat repr_v = MF.find(v);
    if (repr_u != repr_v) {
      MF.union(repr_u, repr_v);
      T.insereix(p);
    }
    ++it;
  }
}
```

8.7.2.3 Cost de l'algorisme

El cost d'ordenar el conjunt d'arestes és $\Theta(a \cdot \log a)$ i el d'inicialitzar el MFSet és $\Theta(n)$.

Cada iteració realitza dues operacions `find` i, alguns cops (en total $n-1$ vegades) una operació `unir`. El bucle, en el pitjor dels casos, realitza $2 \cdot a$ operacions `find` i $n-1$ operacions `unir`. Sigui $m = 2a+n-1$, llavors el bucle costa $O(m \cdot \log n)$ si el MFSet s'implementa amb alguna tècnica d'unió per rang o per pes i de compressió de camins. Aproximadament és $O(a \cdot \log n)$. Com que el cost del bucle és menor que el cost de les inicialitzacions, el cost de l'algorisme és $\Theta(a \cdot \log a)$.

8.7.3 Algorisme de Prim

L'**algorisme de Prim** serveix per trobar un arbre d'expansió mínima d'un graf connex, no dirigit i etiquetat.

En altres paraules, l'algorisme troba el subconjunt d'arestes que formen l'arbre amb tots els vèrtexs en què el pes total de les arestes de l'arbre és el mínim possible.

8.7.3.1 Descripció de l'algorisme

L'algorisme funciona de la següent forma:

1. Comença amb l'arbre buit i inicialment s'afegeix un vèrtex qualsevol del graf.

2. De les possibles arestes que connecten els vèrtexs de l'arbre als altres vèrtexs que encara no estan l'arbre es tria la que té menor etiqueta (distància mínima). S'afegeix aquesta nou vèrtex i aresta a l'arbre.

3. Es repeteix el pas 2 fins que tots els vèrtexs del graf estiguin a l'arbre.

8.7.3.2 Diferències entre Kruskal i Prim

La diferència principal entre Prim i Kruskal és que el primer va afegint vèrtexs amb menor distància, i el segon treballa amb arestes. Prim afegeix vèrtexs sabent que arribem a ells amb menor cost i Kruskal ordena les arestes per seleccionar en cada moment la de menor cost.

La complexitat de cada algorisme per un graf $G = (V, A)$ on $a = |A|$ i $n = |V|$, és:

- Prim: $O(n^2)$
- Kruskal: $O(a \log a)$

Definició 14: Densitat d'un graf

La **densitat** (D) d'un graf $G = (V, A)$ es defineix per la següent fórmula:

$$D = \frac{2 \cdot |A|}{|V| \cdot (|V| - 1)}$$

Tenint en compte la densitat es poden classificar els grafs en:

Dens si D és aproximadament 1.

Dispers si D és aproximadament 0.

En els casos en els que el graf sigui dens serà necessari usar l'algorisme de Prim. En els casos en els que tinguem un graf dispers, serà adequat usar l'algorisme de Kruskal.

8.8 Algorismes de camins mínims

8.8.1 Introducció

Existeix una col·lecció de problemes, denominats problemes de camins mínims (anglès: *shortest-paths problems*), basats en el concepte de camí mínim.

Definició 15: Pes d'un camí

Sigui $G = (V, A)$ un graf dirigit i etiquetat amb valors naturals. Es defineix el **pes del camí** p, $p =< v_0, v_1, v_2, ..., v_k >$, com la suma dels valors de les arestes que el composen:

$$pes(p) = \sum_{i=1}^{k} valor(G, v_{i-1}, v_i)$$

Definició 16: Camí mínim

El **camí mínim** del vèrtex *u* al *v* en $G = (V, A)$, amb $u, v \in V$, és el camí de menor pes d'entre tots els existents entre u i v.

Si existeixen diferents camins amb idèntic menor pes qualsevol d'ells és un camí mínim.

Si no hi ha camí entre u i v el pes del camí mínim és ∞.

La col·lecció de problemes de camins mínims està composada per quatre variants:

1. **Single source shortest paths problem**: S'ha de trobar el camí més curt entre un determinat vèrtex, *source*, i tota la resta de vèrtexs del graf. Aquest problema es resol

eficientment utilitzant l'algorisme de Dijkstra que és un algorisme voraç.

2. **Single destination shortest paths problem**: S'ha de trobar el camí més curt des de tots els vèrtexs a un vèrtex determinat, *destination*. Es resol aplicant Dijkstra al graf de partida però canviant el sentit de totes les arestes.

3. **Single pair shortest paths problem**: Donats dos vèrtexs determinats del graf, *source* i *destination*, trobar el camí més curt entre ells. En el cas pitjor no hi ha un algorisme millor que el propi Dijkstra.

4. **All pairs shortest paths problem**: S'ha de trobar el camí més curt entre els vèrtexs *u* i *v*, per tot parell de vèrtexs del graf. Es resol aplicant Dijkstra a tots els vèrtexs del graf. També es pot utilitzar l'algorisme de Floyd que és més elegant però no més eficient i que segueix l'esquema de Programació Dinàmica.

8.8.2 Algorisme de Dijkstra

L'**algorisme de Dijkstra** (1959) resol el problema de trobar el camí més curt entre un vèrtex donat, anomenat inicial, i tots la resta de vèrtexs del graf.

8.8.2.1 Descripció de l'algorisme

Per desenvolupar aquest algorisme necessitem tenir distribuïts el conjunt de vèrtexs del graf en dos conjunts disjunts:

- En el conjunt `VISTOS` estan els vèrtexs pels que ja es coneix quina és la longitud del camí més curt entre el vèrtex

inicial i ell mateix.

- En el conjunt `NO_VISTOS` estan la resta de vèrtexs i per cadascun d'ells es guarda la longitud del camí més curt des del vèrtex inicial fins a ell que no surt de `VISTOS`. Suposarem que aquestes longituds es guarden en el vector `D`.

L'algorisme de Dijkstra funciona de la següent manera:

- A cada iteració s'escull el vèrtex v de `NO_VISTOS` amb `D[v]` mínima de manera que v deixa el conjunt `NO_VISTOS`, passa a `VISTOS` i la longitud del camí més curt entre el vèrtex inicial i v és, precisament, `D[v]`.
- Tots aquells vèrtexs u de `NO_VISTOS` que siguin successors de v actualitzen convenientment el seu `D[u]`.
- L'algorisme acaba quan tots els vèrtexs estan en el conjunt `VISTOS`.

$G = (V, A)$ és un graf etiquetat i pot ser dirigit o no dirigit. En aquest cas és dirigit. Per facilitar la lectura de l'algorisme se suposa que el graf està implementat en una matriu i que `_matriu[u][v]` conté el valor de l'aresta que va del vèrtex u al v i, si no hi ha aresta, conté el valor infinit.

Exemple:

Agafant com a punt de partida el graf de la figura 8.15, observem com funciona l'algorisme si triem com a vèrtex inicial el vèrtex A.

La taula 8.2 mostra en quin ordre van entrant els vèrtexs en el conjunt `VISTOS` i com es va modificant el vector `D` que conté les distàncies mínimes.

Suposem que la distància d'un vèrtex a si mateix és zero.

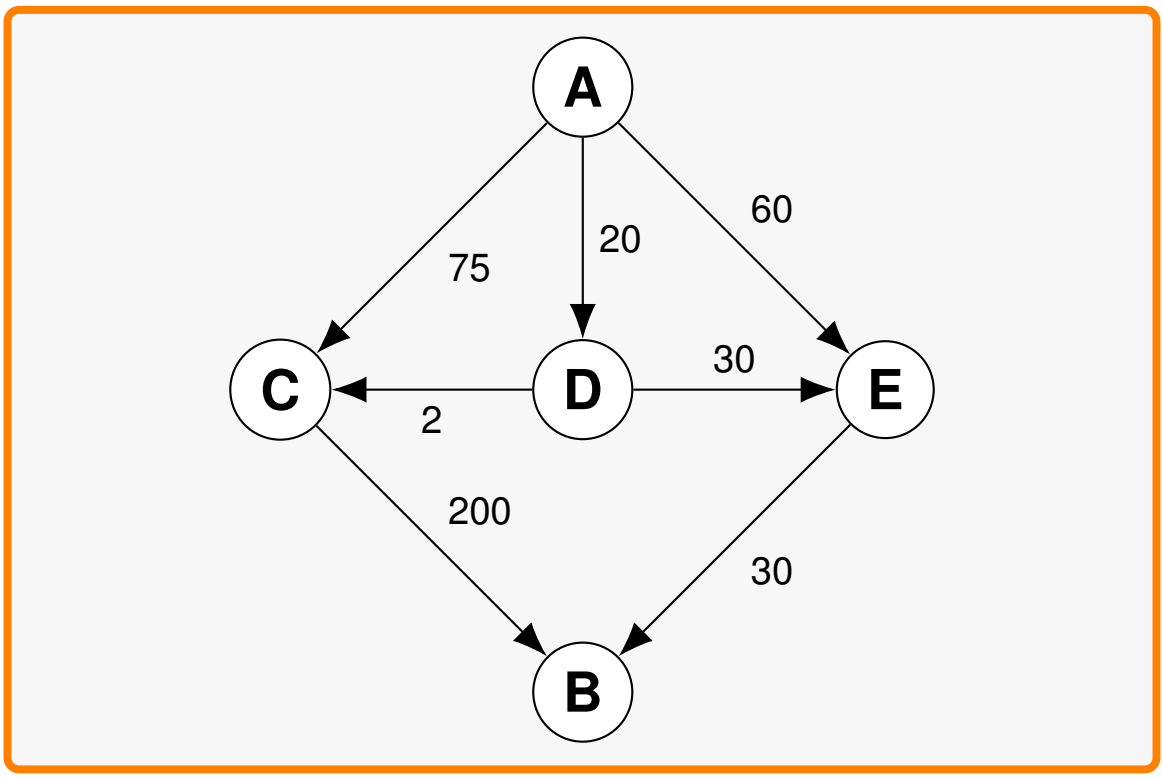


Figura 8.15: Graf dirigit i etiquetat

D					VISTOS
A	B	C	D	E	
0	∞	75	20	60	{A}
0	∞	22	**20**	50	{A, D}
0	222	**22**	**20**	50	{A, D, C}
0	80	**22**	**20**	**50**	{A, D, C, E}
0	**80**	**22**	**20**	**50**	{A, D, C, E, B}

Taula 8.2: Estat de les estructures de dades en aplicar l'algorisme de Dijkstra en el graf 8.15. Els valors en negreta de la taula corresponen a valors definitius i, per tant, contenen la distància mínima entre el vèrtex A i el que indica la columna.

8.8.2.2 Implementació

Calcula la distància mínima entre el vèrtex indicat i tota la resta. Retorna un vector d'n etiquetes.

```
template <typename V, typename E>
E* graf<V, E>::dijkstra (const V &v_ini)
        const throw(error){
  nat n = _verts.size();
  E *D = new E[n];
  conjunt<V> VISTOS;

  try {
    nat x = _verts.consulta(v_ini);
  }
  catch (error) {
    delete[] *D;
    throw error(VertexNoExisteix);
  }

  for (vertexs::iterador v = _verts.begin();
       v != _verts.end(); ++v) {
    nat j = (*v).second;
    D[j] = _matriu[x][j];
  }

  D[i] = 0;
  VISTOS.insereix(v_ini);

  while (VISTOS.size() < n) {
    // S'obté el vèrtex u tal que no pertany a VISTOS i
    // té D mínima
    V u = minim_no_vistos(D, VISTOS);
    VISTOS.afegeix(u);
    nat i = _verts.consulta(u);
    list<V> l;
    successors(u, l);
    for (list<V>::iterator v = l.begin();
         v != l.end(); ++v) {
```

```
        nat j = _verts.consulta(v);
        if ((not VISTOS.conte(*v)) and
            (D[j] > D[i] + _matriu[i][j])) {
          D[j] = D[i] + _matriu[i][j];
        }
      }
    }
```

$\forall u : u \in V$: D[u] conté la longitud del camí més curt des de v_ini a u que no surt de VISTOS, i com VISTOS ja conté tots els vèrtexs, a D hi ha les distàncies mínimes definitives.

```
  return D;
}
```

8.8.2.3 Implementació utilitzant una cua de prioritat

Calcula la distància mínima entre el vèrtex indicat i tota la resta. Retorna un vector d'n etiquetes.

```
template <typename V, typename E>
E* graf<V,E>::dijkstra (const V &v_ini)
        const throw(error) {
  E *D = new E[_verts.size()];
  cuaprioritat<V, nat> C;
  for (vertexs::iterador v = _verts.begin();
       v != _verts.end(); ++v) {
    D[(*v).second] = HUGE_VAL; // ∞
  }

  try {
    nat x = _verts.consulta(v_ini);
  }
  catch (error) {
    delete[] *D;
    throw error(VertexNoExisteix);
  }

  D[x] = 0;
```

```
  C.insereix(v_ini, D[x]);
  while (not C.es_buida()) {
    V u = C.minim();
    nat i = _verts.consulta(u);
    C.elimina_minim();
    // Ajustar el vector D
    list<V> l;
    successors(u, l);
    for (list<V>::iterador v = l.begin();
         v != l.end(); v++) {
      nat j = _verts.consulta(*v);
      if (D[j] > D[i]+_matriu[i][j]) {
        D[j] = D[i]+_matriu[i][j];
        if (C.conte(v)) {
          C.substitueix(*v, D[j]);
        }
        else {
          C.insereix(*v, D[j]);
        }
      }
    }
  }
  return D;
}
```

8.8.2.4 Cost de l'algorisme

El bucle que inicialitza el vector `D` costa $\Theta(n)$ i el bucle principal, que calcula els camins mínims, fa $n-1$ iteracions. En el cost de cada iteració intervenen dos factors:

- **obtenció del mínim**: cost $\Theta(n)$ perquè s'ha de recórrer `D`,
- **ajust de** `D`: depenent de com estigui implementat el graf tindrem:
  - Si el graf està implementat usant una matriu d'adjacència, el seu cost també és $\Theta(n)$ i, per tant, el cost total de l'algorisme és $\Theta(n^2)$.

– Si el graf està implementat en llistes d'adjacència i s'utilitza una cua de prioritat (implementada en un minHeap) per accelerar l'elecció del mínim, llavors seleccionar el mínim costa $\Theta(1)$, la construcció del Heap $\Theta(n)$ i cada operació d'inserció o eliminació del mínim requereix $\Theta(\log n)$.

  Com que en total s'efectuen, en el pitjor dels casos, a operacions d'inserció i n operacions d'eliminació del mínim tenim que l'algorisme triga $\Theta((n+a)\cdot\log n)$ que resulta millor que el de les matrius quan el graf és poc dens.

8.8.3 Algorisme de Floyd

L'**algorisme de Floyd** resol el problema de trobar el camí més curt entre tot parell de vèrtexs del graf.

Aquest problema també es podria resoldre aplicant repetidament l'algorisme de Dijkstra variant el node inicial.

L'algorisme de Floyd proporciona una solució més compacta i elegant pensada especialment per a aquesta situació.

8.8.3.1 Descripció de l'algorisme

L'algorisme de Floyd funciona de la següent manera:

- Utilitza una matriu quadrada `D` indexada per parells de vèrtexs per guardar la distància mínima entre cada parell de vèrtexs.

- El bucle principal tracta un vèrtex, anomenat pivot, cada vegada.

- Quan u és el pivot, es compleix que `D[v][w]` és la longitud del camí més curt entre v i w format íntegrament per pivots de passos anteriors.
- L'actualització de `D` consisteix a comprovar, per a tot parell de nodes v i w, si la distància entre ells es pot reduir tot passant pel pivot u mitjançant el càlcul de la fórmula:

$$D[v][w] = min(D[v][w], D[v][u] + D[u][w])$$

Exemple:
Donat el graf de la figura 8.16 volem trobar els camins mínims entre tots els vèrtexs usant l'algorisme de Floyd.

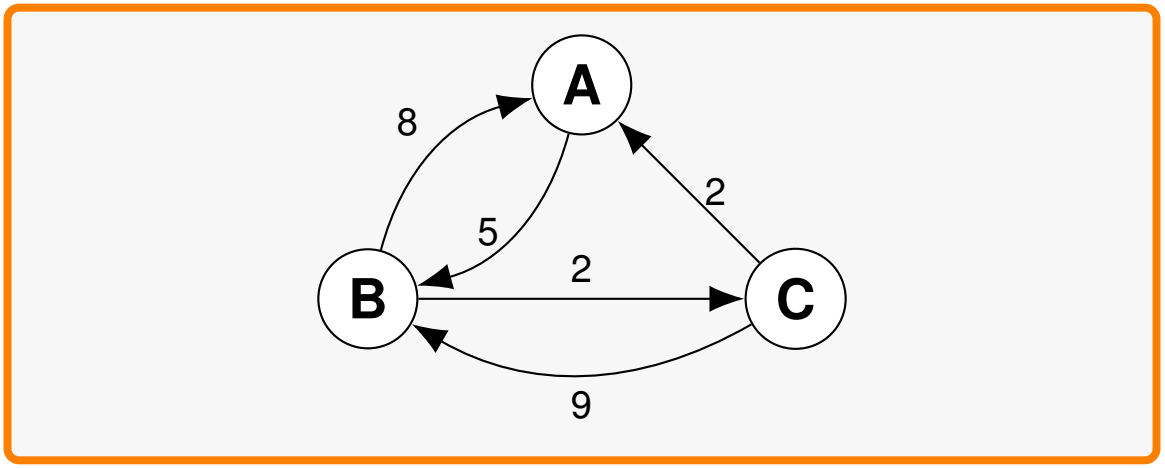


Figura 8.16: Graf dirigit i etiquetat

El resultat d'aplicar aquest algorisme es pot veure a la taula 8.3. Aquest algorisme tindrà tantes iteracions com vèrtexs tingui el graf, car cadascun dels vèrtexs ha de ser en algun moment el pívot.

Estat inicial

	A	B	C
A	0	5	∞
B	8	0	2
C	2	9	0

Pivot = A

	A	B	C
A	0	5	∞
B	8	0	2
C	2	**7**	0

Pivot = B

	A	B	C
A	0	5	**7**
B	8	0	2
C	2	7	0

Pivot = C

	A	B	C
A	0	5	7
B	**4**	0	2
C	2	7	0

Taula 8.3: Esquema de l'aplicació de l'algorisme de Floyd en el graf 8.16.

8.8.3.2 Implementació

Mètode privat. Crea una matriu de n x n.
Cost: $\Theta(n^2)$

```
template <typename V, typename E>
E** graf<V, E>::crear_matriu(unsigned int n)
        const throw(error) {
  // Pre:  E té constructor per defecte
  E** m = new (E*)[n];
  for (nat i = 0; i < n; ++i) {
    m[i] = new E[n];
  }
  return m;
}
```

```
template <typename V, typename E>
E** graf<V, E>::floyd() const throw(error) {
  E** D = crear_matriu(_verts.size());
  for (vertexs::iterador v = _verts.begin();
       v != _verts.end(); ++v) {
    unsigned int i = (*v).second;
    for (vertexs::iterador w = _verts.begin();
              w != _verts.end(); ++w) {
      unsigned int j = (*w).second;
      D[i][j] = _matriu[i][j];
    }
    D[i][i] = 0;
  }

  for (vertexs::iterador u = _verts.begin();
       u != _verts.end(); ++u) {
    unsigned int i = (*u).second;
    for (vertexs::iterador v = _verts.begin();
              v != _verts.end(); ++v) {
      unsigned int j = (*v).second;
      for (vertexs::iterador w = _verts.begin();
              w != _verts.end(); ++w) {
        unsigned int k = (*w).second;
        if (D[j][k] > D[j][i] + D[i][k]) {
          D[j][k] = D[j][i] + D[i][k];
        }
      }
    }
  }
  return D;
}
```

8.8.3.3 Diferències entre els algorismes de Floyd i Dijkstra

- Els vèrtexs es tracten en un ordre que no té res a veure amb les distàncies.

- No es pot assegurar que cap distància mínima sigui definitiva fins que acaba l'algorisme.

8.8.3.4 Cost de l'algorisme

L'eficiència temporal és $\Theta(n^3)$. L'aplicació reiterada de Dijkstra té el mateix cos però la constant multiplicativa de l'algorisme de Floyd és menor que la de Dijkstra degut a la simplicitat de cada iteració.

Índex alfabètic