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, Jordi Esteve
ASSIGNATURA: Estructura de la Informació (ESIN)
CURS: Q3
TITULACIONS: Grau en Informàtica
DEPARTAMENT: Ciències de la Computació
ANY: 2022/2023

Vilanova i la Geltrú, 22 de novembre de 2022

Índex

6
Diccionaris

Un diccionari és un univers en ordre alfabètic.

Anatole France (1844-1924)

6.1 Conceptes

Volem modelitzar la classe de les funcions:

$$f : K =\!\!\rightarrow V$$

Dels valors del domini (**K**) en direm **claus**.
Dels valors de l'abast (**V**), **informació** o simplement **valors**.

- **Funció total**: Totes les claus tenen informació associada.
- **Funció parcial**: No totes les claus tenen informació associada.

Una funció parcial pot transformar-se en total associant el valor **indefinit** a totes les claus sense informació associada.

Aquesta classe és coneguda amb el nom de **taula** (anglès: **lookup table**, **symbol table**) o **diccionari**.

També es pot considerar els diccionaris com *conjunts de parells clau i valor*, amb la restricció que no hi hagi dos parells amb la mateixa clau, de manera que la clau identifica unívocament el parell.

Un diccionari proporciona operacions per tal de localitzar un element donada la seva clau (cerca per clau) i obtenir el seu valor associat. Per exemple, un diccionari podria guardar la informació dels estudiants d'una escola i accedir a la informació d'un estudiant en concret donat el seu DNI (si s'ha escollit el DNI com la clau per identificar els estudiants).

Els **conjunts** són un cas particular de diccionari. En un conjunt podem considerar que:

- Els elements del conjunt són les claus.

- El seu valor associat és un booleà: cert pels elements del conjunt i fals pels que no hi són.

En el cas dels conjunts, l'operació que consulta el valor associat a una clau és la típica operació que indica si l'element pertany al conjunt.

6.1.1 Classificació dels diccionaris

Podem classificar les classes diccionaris segons les operacions d'actualització permeses. En tots els casos disposem d'una operació per consultar el valor associat a una clau.

- **Estàtic**: Els n elements que formaran el diccionari són coneguts. No es permeten insercions o eliminacions posteriors. La classe ofereix una operació per crear un diccionari a partir d'un vector o llista d'elements.
- **Semidinàmic**: Es permet la inserció de nous elements i la modificació d'un valor associat a una clau donada, però no les eliminacions.
- **Dinàmic**: Es permet la inserció de nous elements i la modificació i l'eliminació d'elements existents donada la seva clau.

Especificarem el cas més genèric: els **diccionaris dinàmics**.

6.2 Especificació

6.2.1 Especificació bàsica

```
template <typename Clau, typename Valor>
class dicc {
public:
```

```
  Constructora. Crea un diccionari buit.
```

```
  dicc() throw(error);
```

```
  Les tres grans.
```

```
  dicc(const dicc &d) throw(error);
  dicc& operator=(const dicc &d) throw(error);
  ~dicc() throw();
```

```
  Afegeix el parell <k, v> al diccionari si no hi havia cap
  parell amb la clau k; en cas contrari substitueix el valor
  antic per v.
```

```
  void insereix(const Clau &k, const Valor &v)
                          throw(error);
```

```
  Elimina el parell <k, v> si existeix un parell que té com a
  clau k; no fa res en cas contrari.
```

```
  void elimina(const Clau &k) throw();
```

```
  Retorna cert si i només si el diccionari conté un parell amb
  la clau donada.
```

```
  bool existeix(const Clau &k) const throw();
```

```
  Retorna el valor associat a la clau donada en cas que
  existeixi un parell amb la clau k; llança una excepció si
  la clau no existeix.
```

```
  Valor consulta(const Clau &k) const throw(error);

private:
```

```
    ...
};
```

6.2.2 Operacions addicionals

És freqüent que una classe diccionari tingui altres operacions addicionals. Per exemple:

- *Operacions entre dos o més diccionaris*: unió, intersecció i diferència de diccionaris.
- *Operacions específiques quan les claus són strings*: p.e. trobar tots els strings que comencen amb un prefix donat.
- *Operacions específiques quan les claus admeten una relació d'ordre total* (tenim una operació de comparació entre claus):
  - Operacions per examinar els elements per ordre creixent/decreixent similars a les utilitzades per recórrer les llistes (diccionaris recorribles).
  - Operacions per posició: Consultar l'element i-èssim, eliminar l'element i-èssim, ...
  - Eliminar tots els elements que la seva clau estigui entremig de dos claus K1 i K2 donades, determinar quants elements hi ha amb una clau menor a una clau donada, ...

6.3 Diccionaris recorribles

Una família important de diccionari la constitueixen els anomenats diccionaris **recorribles** o **ordenats**.

Si les claus admeten una relació d'ordre total, és a dir, tenim una operació de comparació < entre claus, pot ser útil que la classe ofereixi operacions que permetin examinar els elements (o parells clau-valor). Examinarem els elements per ordre decreixent de les seves claus.

En una primera implementació dels diccionaris recorribles podem usar un punt d'interès per tal de desplaçar-nos pel diccionari.

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

```
  Retorna una llista amb tots els parells (clau, valor) del
  diccionari en ordre ascendent.
```

```
  void llista_dicc(list<pair_cv> &l) const throw(error);
```

```
  Funcions per recórrer les claus en ordre ascendent mitjançant
  un punt d'interès.
```

```
  void principi() throw(error);
  void avanca() throw(error);
  pair_cv actual() const throw(error);
  bool final() const throw();

  ...
};
```

Els recorreguts en un diccionari també es poden realitzar mitjançant iteradors.

```
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 postincrement; avancen l'iterador.

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

Operadors de comparació.

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

Iteradors al principi i al final (sentinella) del diccionari.

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

6.4 Usos dels diccionaris

L'aplicació dels diccionaris o taules en el camp de la programació és diversa. Per exemple:

- Taula de símbols (noms de paraules reservades, variables, constants, accions, funcions) d'un compilador.
- Taula per emmagatzemar els nodes d'un graf.
- Taules i índexs utilitzats per un sistema gestor de bases de dades.
- ...

OBJECTIU DELS DICCIONARIS
Així com les estructures lineals estan orientades a l'accés consecutiu a tots els seus elements, els diccionaris estan orientats a l'accés individual als seus elements. L'objectiu és que les operacions `insereix`, `elimina`, `consulta` i `existeix` tinguin un cost menor de $\Theta(n)$.

6.5 Implementació

- **Vector indexat per les claus**: Si les claus són enteres, el seu nombre no és gaire gran i són consecutives dins d'un rang [ClauMIN .. ClauMAX] pot ser adequat utilitzar un vector indexat per les claus. Cada element del vector emmagatzemarà el valor associat a la clau o el valor indefinit si la clau no s'ha inserit en el diccionari. El cost de totes les operacions, excepte crea, seria constant.

- **Llista enllaçada desordenada**: Cada node de la llista conté un parell clau-valor. El cost de les insercions, eliminacions i consultes és $\Theta(n)$, sent n el número d'elements del diccionari. El cost en espai també és $\Theta(n)$ i tots els algorismes són molt simples. És una solució acceptable si els diccionaris són petits. Si cal suportar operacions de recorregut ordenat no seria una opció recomanable.

- **Llista enllaçada desordenada amb una estratègia d'autoorganització**: El cost en el cas pitjor és el mateix que l'anterior però el cost mig pot millorar molt si hi ha un grau elevat de localitat de referència en les cerques de les claus (si algunes claus es cerquen molt més sovint que d'altres).

- **Llista ordenada seqüencial dins d'un vector**: Només és convenient quan el diccionari és estàtic o les insercions i eliminacions es produeixen molt de tant en tant. Les consultes per clau es poden implementar amb l'algorisme de cerca dicotòmica o binària amb cost $\Theta(log(n))$. També suporta de manera eficient recorreguts ordenats.

- **Llista enllaçada ordenada**: El cost de les insercions, eliminacions i consultes segueix sent $\Theta(n)$ tant en el cas pitjor com en el cas mig. El seu avantatge és que permet implementar fàcilment operacions de recorregut ordenat del diccionari i d'unió, intersecció i diferencia entre diccionaris.

- Altres implementacions que examinarem:

  - **Arbres de cerca** (**BSTs** o Binary Search Trees, **AVLs**).
  - **Taules de dispersió** (**Hashing tables**).
  - **Arbres digitals** (**tries** i variants com els **TSTs** o Ternary Search Trees).

6.6 Arbres binaris de cerca

6.6.1 Definició i exemples

Definició 1: BST

Un **arbre binari de cerca** (**Binary Search Tree** o **BST**) és un arbre binari buit o un arbre binari tal que per a tot node, la clau del node és més gran que qualsevol de les claus del subarbre esquerre i és més petita que qualsevol de les claus del subarbre dret.

- No és necessari que l'arbre binari sigui complet ni ple.
- Quan s'utilitza un BST per implementar un diccionari que guarda parells <clau, valor>, dins de cada node de l'arbre s'emmagatzemarà el valor associat a la clau del node.

Exemples:

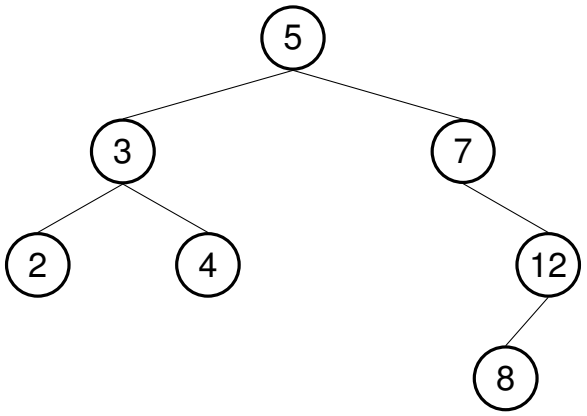


Figura 6.1: Exemple de BST amb claus enteres.

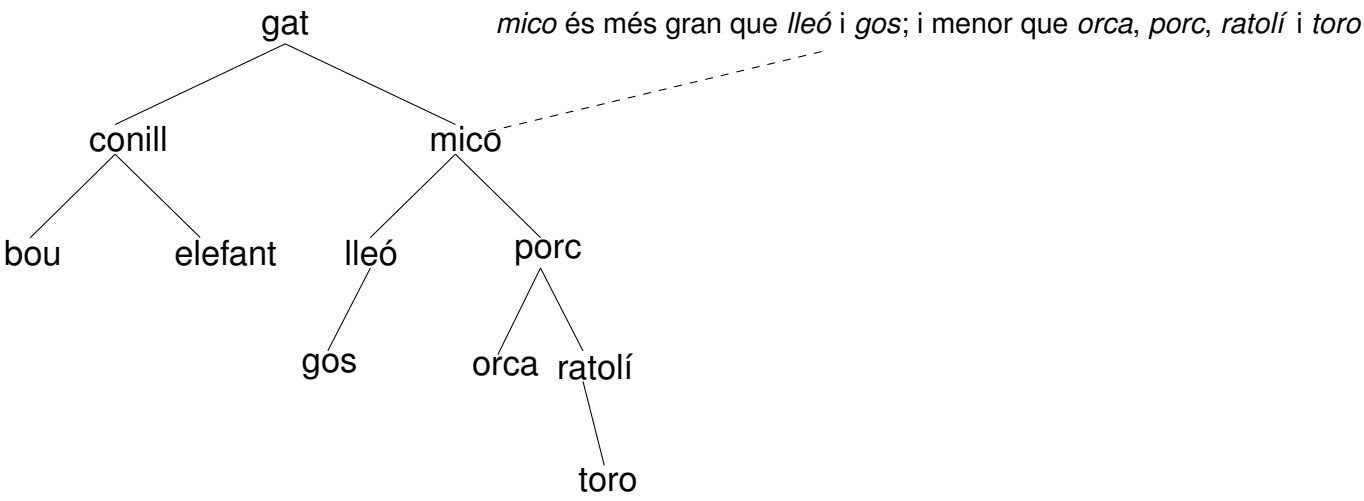


Figura 6.2: Exemple de BST amb claus strings.

6.6.2 Especificació

Un arbre binari de cerca es pot implementar usant qualsevol de les formes que hem vist per a arbre binaris. Encadenada amb punters és la més usual.

```
template <typename Clau, typename Valor>
class dicc {
public:
  void insereix(const Clau &k, const Valor &v)
                throw(error);
  void elimina(const Clau &k) throw();
  void consulta(const Clau &k, bool &hi_es, Valor &v)
                const throw(error);
  ...

private:
  struct node {
    Clau _k;
    Valor _v;
    node* _esq;    // fill esquerre
    node* _dret;   // fill dret
    node(const Clau &k, const Valor &v, node* esq = nullptr,
                 node* dret = nullptr) throw(error);
```

```
  };
  node *_arrel;
```

Mètodes privats.

```
  static node* consulta_bst(node *n, const Clau &k)
                throw();
  static node* insereix_bst(node *n, const Clau &k,
                Valor &v) throw(error);
  static node* elimina_bst(node *n, const Clau &k)
                throw();
  static node* ajunta(node *t1, node* t2) throw();
  static node* elimina_maxim(node *n) throw();
  ...
};
```

6.6.3 Operacions i cost associat

6.6.3.1 a) Consultar una clau k:

Mirem si k coincideix amb la clau que hi ha a l'arrel:

- Si coincideix ja l'hem trobat i la cerca s'acaba.
- Si k < que la clau de l'arrel llavors baixem pel fill esquerre. Segons la definició de BST si hi ha algun element la clau del qual sigui k, llavors aquest element es troba en el subarbre esquerre.
- Si k > que la clau de l'arrel llavors baixem pel fill dret. Anàleg al cas anterior.

El cost és lineal $\Theta(h)$ sent h l'alçada de l'arbre. En el cas pitjor $h = n$ (això succeeix quan el BST s'ha degenerat perquè

s'ha convertit en una llista, per ex. quan tots els nodes són fill dret de l'anterior).

El mètode `consulta` utilitza el mètode `consulta_bst` que a continuació veurem una implementació iterativa i una recursiva.

```
template <typename Clau, typename Valor>
void dicc<Clau, Valor>::consulta(const Clau &k, bool &hi_es,
                                 Valor &v) const throw(error) {
  node *n = consulta_bst(_arrel, k);
  if (n == nullptr) {
    hi_es = false;
  }
  else {
    hi_es = true;
    v = n->_v
  }
}
```

Versió recursiva

Mètode privat que rep un apuntador p al subarbre a partir del qual s'ha de fer la cerca i una clau k. Retorna un apuntador NULL si k no està present en el subarbre, o bé, un apuntador que apunta al node que conté la clau k.

```
template <typename Clau, typename Valor>
typename dicc<Clau, Valor>::node*
dicc<Clau, Valor>::consulta_bst(node *n, const Clau &k)
throw() {
  if (n == nullptr or n->_k == k) {
    return n;
  }
  else {
    if (k < n->_k)
      return consulta_bst(n->_esq, k);
    else           // k > n->_k
      return consulta_bst(n->_dret, k);
  }
}
```

Versió iterativa

Donat que el mètode de cerca recursiu és recursiu final obtenir una versió iterativa és immediat.

Mètode privat que rep un apuntador p al subarbre a partir del qual s'ha de fer la cerca i una clau k. Retorna un apuntador NULL si k no està present en el subarbre, o bé, un apuntador que apunta al node que conté la clau k.

```
template <typename Clau, typename Valor>
typename dicc<Clau, Valor>::node*
dicc<Clau, Valor>::consulta_bst(node *n, const Clau &k)
throw() {
  while (n != nullptr and k != n->_k ) {
    if (k < n->_k)
      n = n->_esq;
    else              // k > n->_k
      n = n->_dret;
  }
  return n;
}
```

6.6.3.2 b) Obtenir la llista ordenada de tots els elements del BST:

Com que fill_esq < node < fill_dret $\Rightarrow$ Cal recórrer l'arbre en inordre:

- Primer els elements del subarbre esquerre en inordre
- Després l'arrel
- Després els elements del subarbre dret en inordre

El cost en aquest cas és lineal $\Theta(n)$, sent n el número de nodes de l'arbre.

6.6.3.3 c) Mínim i Màxim

Mínim és el primer element en inordre $\Rightarrow$ Fill que està més a l'esquerre.

Màxim és l'últim element en inordre $\Rightarrow$ Fill que està més a la dreta.

El cost és $\Theta(h)$.

6.6.3.4 d) Inserir un element

Quan hem d'inserir un nou element, sempre queda incorporat com a fulla de l'arbre. Per saber on és la posició dins de l'arbre hem de baixar des de l'arrel cap a l'esquerra o cap a la dreta depenent del valors que trobem. Se segueix un raonament molt similar al utilitzat per desenvolupar l'algorisme de cerca.

Al final, o bé trobem l'element (llavors no hem d'inserir-lo sinó canviar el valor), o bé trobem el lloc on hem de fer la inserció.

Per exemple, per inserir l'element 13 en el següent arbre on només es mostren les claus es seguirien els següents passos:

1. Es comença per l'arrel, donat que la nova clau és més gran que l'arrel continuarem per l'arbre dret.

2. L'arrel del subarbre dret és més gran que la nova clau. Per tant es continua per l'arbre esquerre.

3. Atès que aquest subarbre ja no té fill esquerre per on continuar això vol dir que ja hem arribat a una fulla i és on afegirem el nou element.

Abans de passar a la implementació de les insercions, veurem la implementació (trivial) de la constructora de la classe node.

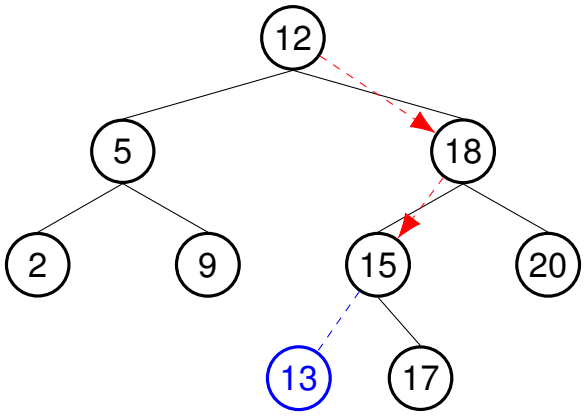


Figura 6.3: Inserció de la clau 13 en un BST.

Constructor de la classe node. Implementem la constructora del node per tal de fer més senzilla la implementació dels altres mètodes.

```
template <typename Clau, typename Valor>
dicc<Clau, Valor>::node::node (const Clau &k,
        const Valor &v, node* esq, node* dret) throw(error) :
        _k(k), _v(v), _esq(esq), _dret(dret) {
}

template <typename Clau, typename Valor>
void dicc<Clau, Valor>::insereix(const Clau &k,
const Valor &v) throw(error) {
  _arrel = insereix_bst(_arrel, k, v);
}
```

Versió recursiva

Mètode privat de classe.

```
template <typename Clau, typename Valor>
typename dicc<Clau, Valor>::node*
dicc<Clau, Valor>::insereix_bst (node *n, const Clau &k, const
  if (n == nullptr) {
```

```
    return new node(k, v);
  }
  else {
    if (k < n->_k) {
      n->_esq = insereix_bst(n->_esq, k, v);
    } else if (k > n->_k) {
      n->_dret = insereix_bst(n->_dret, k, v);
    } else {
      n->_v = v;
    }
    return n;
  }
}
```

Versió iterativa

La versió iterativa és més complexa, ja que a més de localitzar la fulla en la qual s'ha de realitzar la inserció, caldrà mantenir un apuntador `pare` el qual serà el pare del nou node.

```
template <typename Clau, typename Valor>
typename dicc<Clau, Valor>::node*
dicc<Clau, Valor>::insereix_bst(node *p, const Clau &k, const
  node *pare = nullptr;
  node *p_orig = p;

  // El BST estava buit
  if (p == nullptr) {
    p_orig = new node(k, v);
  }
  else {
    // busquem el lloc on inserir el nou element
    while (p != nullptr and p->_k != k) {
      pare = p;
      if (k < p->_k) {
        p = p->_esq;
      } else {
        p = p->_dret;
      }
    }
    // inserim el nou node com a fulla o modifiquem el valor
```

```
    // associat si ja hi havia un node amb la clau donada.
    if (p == nullptr) {
      if (k < pare->_k)
        pare->_esq = new node(k, v);
      } else {
        pare->_dret = new node(k, v);
      }
    } else {
      p->_v = v;                    // La clau ja existia
    }
  }
  return p_orig;
}
```

El cost d'inserir un element és $\Theta(h)$.

6.6.3.5 e) Eliminar un element

És l'operació més complexa sobre arbre binaris de cerca. S'han de distingir diferents situacions:

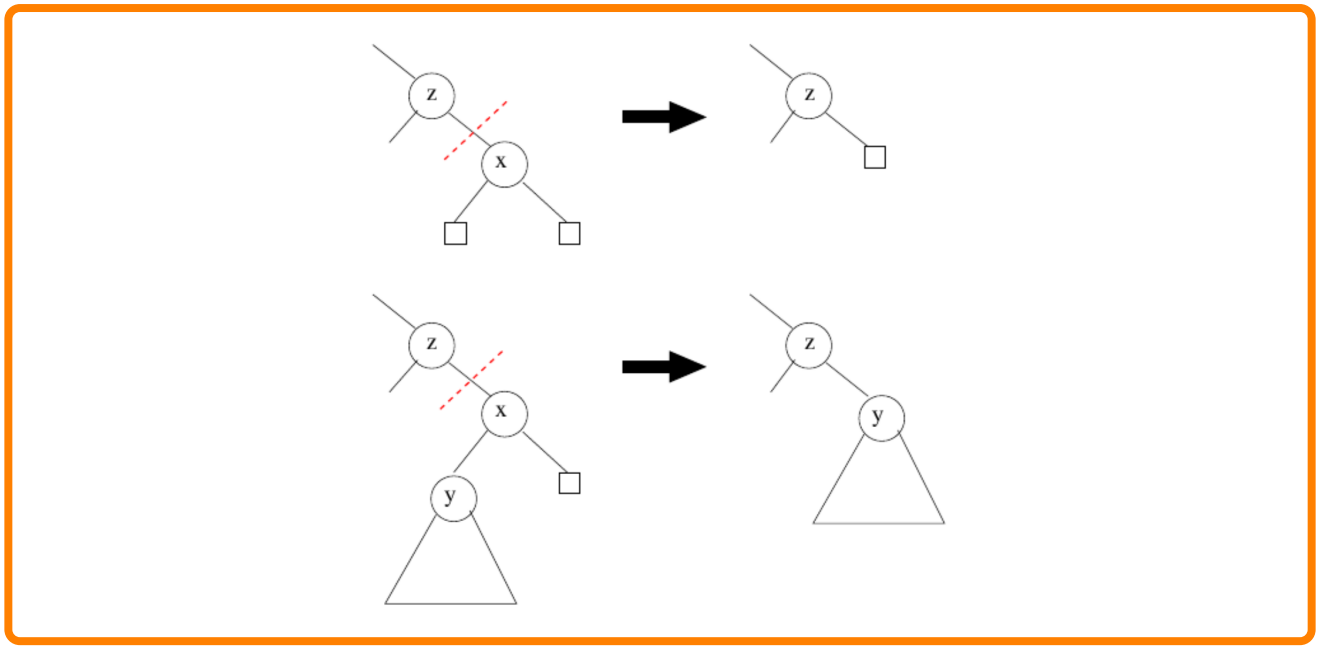


Figura 6.4: Eliminació en un BST, casos 1 i 2.

1. Eliminar una fulla (un node en el que els dos subarbres són buits) $\Rightarrow$ Simplement s'elimina de l'arbre

2. Eliminar un node amb un fill $\Rightarrow$ Enllaçar el pare del node eliminat amb l'únic fill.

3. Eliminar un node amb dos fills $\Rightarrow$ Aquí es poden utilitzar dos mètodes equivalents:

   (a) Amb el successor $\Rightarrow$ Canviar el node a eliminar pel seu successor i eliminar el successor. El successor és el mínim del subarbre dret i, per tant, serà fàcil eliminar-lo utilitzant el mètode 1 o 2.

(b) Amb el predecessor $\Rightarrow$ Canviar el node a eliminar pel seu predecessor i eliminar el predecessor. El predecessor és el màxim del subarbre esquerre i, per tant, serà fàcil eliminar-lo utilitzant el mètode 1 o 2.

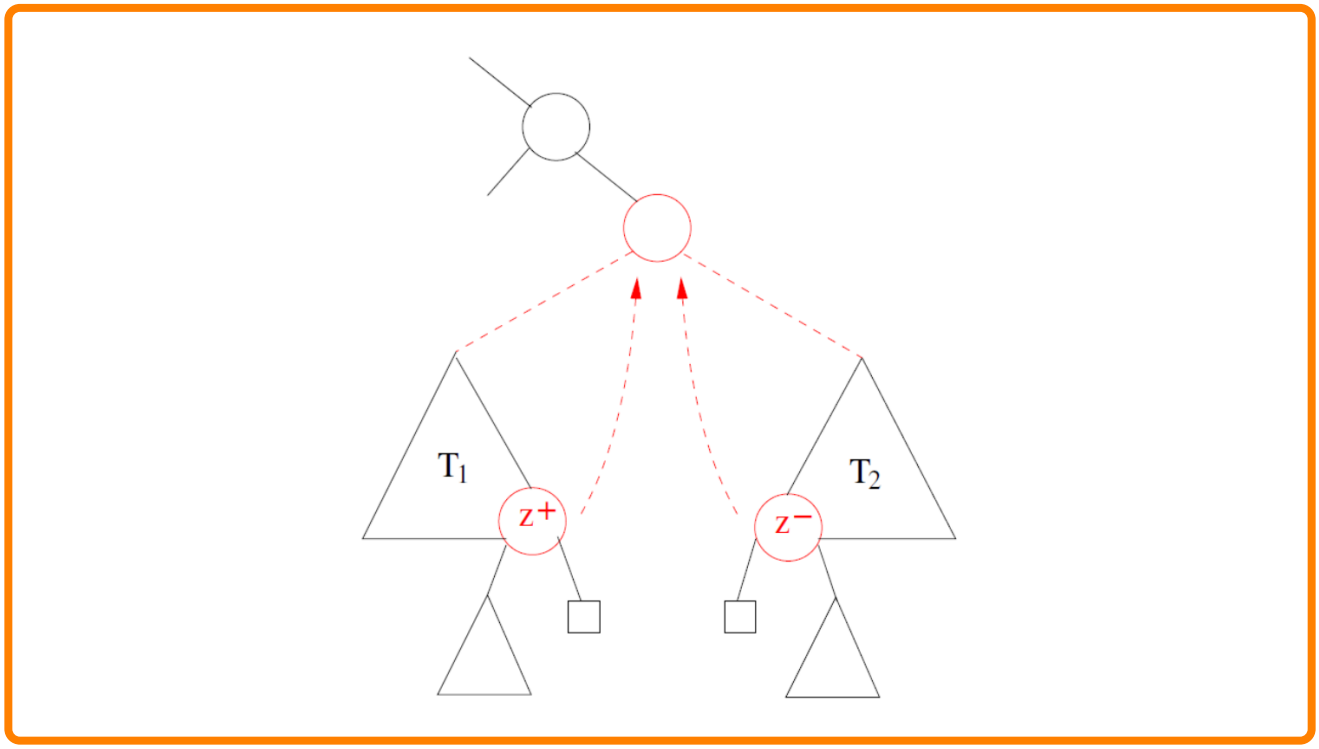


Figura 6.5: Eliminació en un BST, cas3: Dues alternatives.

El cost d'eliminar un element és $\Theta(h)$.

```
template <typename Clau, typename Valor>
void dicc<Clau, Valor>::elimina (const Clau &k) throw() {
  _arrel = elimina_bst(_arrel, k);
}
```

Mètode privat recursiu.

```
template <typename Clau, typename Valor>
typename dicc<Clau, Valor>::node*
dicc<Clau, Valor>::elimina_bst (node *n, const Clau &k) {
  node *p = n;
```

```
  if (n != nullptr) {
    if (k < n->_k) {
      n->_esq = elimina_bst(n->_esq, k);
    }
        else if (k > n->_k) {
      n->_dret = elimina_bst(n->_dret, k);
    }
    else {
      n = ajunta(n->_esq, n->_dret);
      delete(p);
    }
  }
  return n;
}
```

Per ajuntar dos BSTs, si un dels dos és buit el resultat és l'altre. Si els dos arbres t1 i t2 no són buits utilitzarem la tècnica comentada en el punt 3.b: el màxim del subarbre t1 (l'esquerre) el col·locarem com arrel, t1 sense el màxim serà el fill esquerre i t2 serà el fill dret.

```
template <typename Clau, typename Valor>
typename dicc<Clau, Valor>::node*
dicc<Clau, Valor>::ajunta (node *t1, node *t2) throw() {
  if (t1 == nullptr) {
    return t2;
  }
  if (t2 == nullptr)     {
    return t1;
  }
  node* p = elimina_maxim(t1);
  p->_dret = t2;
  return p;
}
```

L'acció `elimina_màxim` elimina el node de clau màxima de l'arbre donat i retorna un punter a aquest node que ha quedat deslligat de l'arbre. Per trobar el màxim de l'arbre cal recórrer

els fills drets començant des de l'arrel.

```
Aquest mètode rep un apuntador a l'arrel d'un BST i ens
retorna l'apuntador al nou BST. L'arrel del nou BST és
l'element màxim del BST antic.  En el nou BST el subarbre
dret és NULL i l'esquerre és el BST que s'obté d'eliminar
l'element màxim.
```

```
template <typename Clau, typename Valor>
typename dicc<Clau, Valor>::node*
dicc<Clau, Valor>::elimina_maxim (node* p) throw() {
  node *p_orig = p, *pare = nullptr;
  while (p->_dret != nullptr) {
    pare = p;
    p = p->_dret;
  }
  if (pare != nullptr) {
    pare->_dret = p->_esq;       // p és fill dret de pare
    p->_esq = p_orig;
  }
  return p;
}
```

Experimentalment s'ha comprovat que convé alternar entre el predecessor i el successor del node a eliminar cada vegada que cal eliminar un node del BST. Ho podem fer mitjançant una decisió aleatòria o amb un booleà que canviï alternativament de cert a fals. S'ha observat que, si ho fem així, el BST té els nodes més ben repartits i, per tant, menor alçada.

6.6.4 Altres algorismes sobre BSTs

Els BSTs permeten vàries operacions, sent els algorismes corresponents simples i amb costos mitjos raonables (típicament lineal respecte l'alçada de l'arbre $\Theta(h)$).

Exemple 1: Donades dues claus $k1$ i $k2$, $k1 < k2$, volem implementar una operació que retorni una llista ordenada de tots els elements que tinguin la seva clau k entremig de les dues donades, o sigui $k1 \leq k \leq k2$. Farem servir la classe `list` de la biblioteca STL per retornar els elements.

```
Retorna una llista ordenada de tots els elements que
tinguin la seva clau k entremig de les dues donades, o sigui
k1 ≤ k ≤ k2.  Suposem que la llista L és buida.
```

```
template <typename Clau, typename Valor>
void dicc<Clau, Valor>::llista_interval(const Clau &k1, const
  rllista_interval(_arrel, k1, k2, L);
}

template <typename Clau, typename Valor>
static void dicc<Clau, Valor>::rllista_interval(node *n,
const Clau &k1, const Clau &k2, list<Valor> &L)
throw(error) {
  if (n != nullptr) {
    if (k1 <= n->_k) {
      rllista_interval(n->_esq, k1, k2, L);
    }
    if (k1 <= n->_k and n->_k <= k2) {
      L.push_back(n->_v);
    }
                if (n->_k <= k2) {
      rllista_interval(n->_dret, k1, k2, L);
                }
  }
}
```

Exemple 2: Consulta o eliminació d'un element del BST per posició (per exemple buscar el 5è element del diccionari).

Cal fer una lleugera modificació de l'estructura de dades del BST: afegir un enter a cada node que guardi la grandària del subarbre que penja d'ell. Per localitzar un element per posició, aquest enter que hem afegit permet decidir recursivament per quina branca s'ha de continuar la cerca.

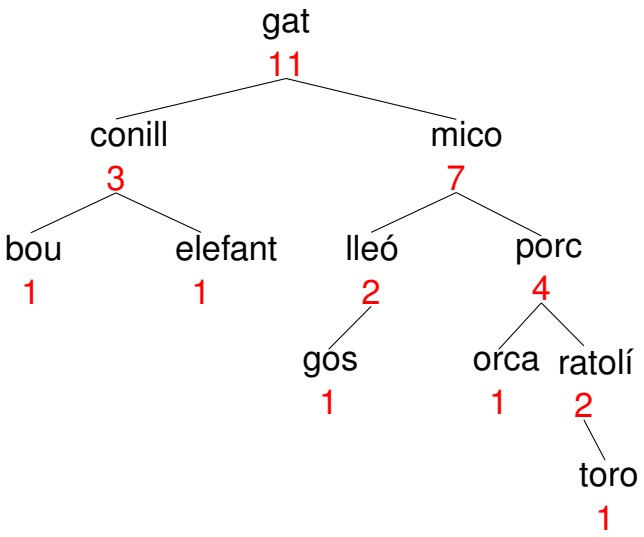


Figura 6.6: BST amb les grandàries dels subarbres.

Si cerquem el 4at element en l'exemple anterior, com que a l'esquerra en hi ha tres (conill conté un 3), el 4at element serà el gat. Si cerquem l'animal amb la posició<4 caldrà baixar pel fill esquerra i si cerquem l'animal amb posició>4 caldrà baixar pel fill dret. Observeu que si baixem pel fill dret canviarà la posició de l'element a cercar:

$$pos = pos\_anterior - \#elements\_subarbre\_esquerre - 1$$

Per exemple, si cerquem el 6è element, baixarem pel fill dret i cercarem el 2on element del subarbre que penja del mico (posició = 6 - 3 - 1 = 2).

Quan inserim/eliminem un element caldrà incrementar/de-

crementar el nombre d'elements per a tots els nodes que formen el camí que s'ha seguit durant la cerca.

6.7 Arbres binaris de cerca equilibrats

6.7.1 Definició i exemples

El cost de la majoria d'operacions dels Arbres Binaris de Cerca són $\Theta(h)$. En el pitjor cas, un Arbre Binari de Cerca de n elements pot tenir una alçada $h = n$. Llavors totes les operacions tenen cost lineal $\Theta(n)$ sent n el número de nodes.

Per a millorar aquest cost tenim dues possibilitats:

1. No fer rés perquè es pot demostrar que si en un Arbre Binari de Cerca introduïm els elements de manera aleatòria l'alçada de l'arbre és $\Theta(log(n))$.

2. Podem forçar que les insercions i supressions dins de l'arbre mantinguin l'alçada en $log(n)$. Es tracta d'aconseguir que els elements de l'arbre estiguin repartits d'una forma més o menys equilibrada. Si ho aconseguim, l'alçada serà $log(n)$ i, per tant, el cost de les operacions serà logarítmic. Aquest és l'objectiu dels Arbres Binaris de Cerca Equilibrats o AVL (nom que prové dels seus inventors: Adelson-Velsky i Landis).

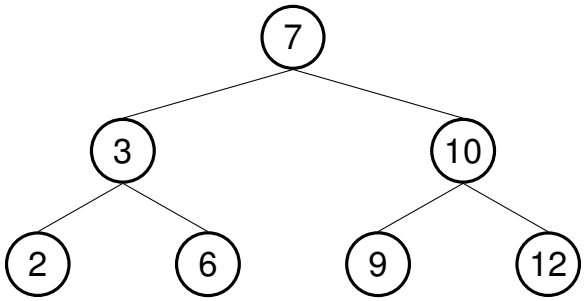
Definició 2: AVL

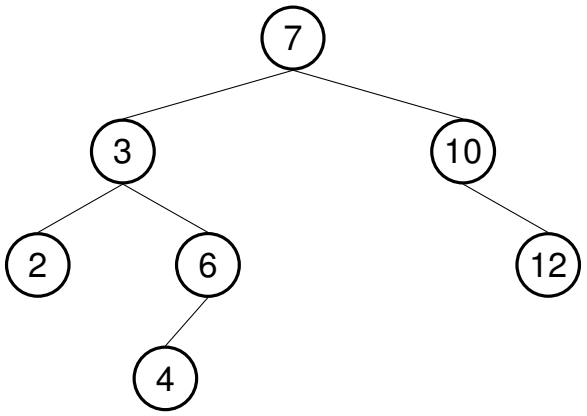
Un **arbre binari de cerca equilibrat** (**AVL**) és un arbre binari buit o un arbre binari de cerca tal que per a tot node es compleix que el factor d'equilibri és ≤ 1.

$$Factor\ d'equilibri\ d'un\ node = |altura(fill\_esquerre) - altura(fill\_dret)|$$

Per tant, en un AVL la diferència màxima entre les altures dels subarbres d'un node és ≤ 1.

Exemples:





En els nodes de l'AVL es guarda l'altura del seu subarbre per calcular fàcilment el factor d'equilibri. Les operacions d'inserir i eliminar hauran d'actualitzar les altures dels nodes que s'han visitat durant la inserció/eliminació.

```
template <typename Clau, typename Valor>
class dicc {
public:
  ...

private:
  struct node {
    Clau _k;
    Valor _v;
```

```
    node* _esq;   // fill esquerre
    node* _dret;  // fill dret
    nat   _altura;
    node(const Clau &k, const Valor &v, node* esq = nullptr,
                 node* dret = nullptr) throw(error);
  };
  node *_arrel;
```

Mètodes privats.

```
  static int altura(node *n);
  static int factor_equilibri(node *n);
  ...
};

template <typename Clau, typename Valor>
dicc<Clau, Valor>::node::node (const Clau &k,
        const Valor &v, node* esq, node* dret) throw(error) :
        _k(k), _v(v), _esq(esq), _dret(dret), _altura(1) {
}

template <typename Clau, typename Valor>
int dicc<Clau, Valor>::altura(node *n)
{
  if (n == nullptr)
    return 0;
  return n->_altura;
}

template <typename Clau, typename Valor>
int dicc<Clau, Valor>::factor_equilibri(node *n)
{
  if (n == nullptr)
    return 0;
  return altura(n->left) - altura(n->right);
}
```

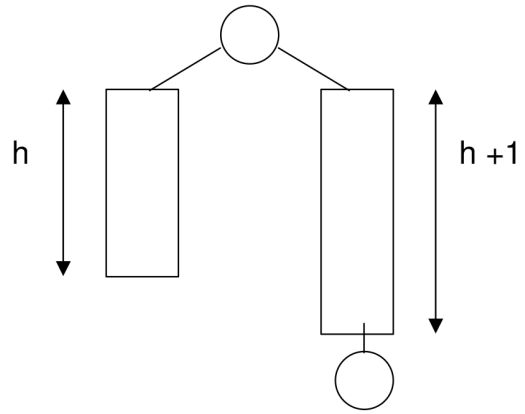
6.7.2 Inserció en un arbre AVL

L'operació d'inserció té dues etapes:

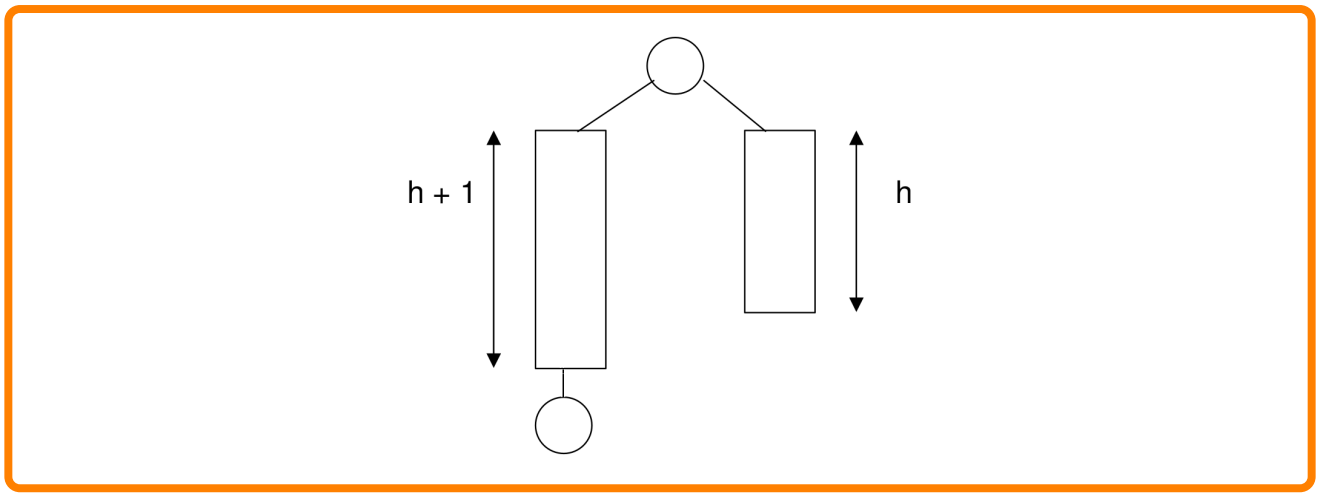
- D'una banda s'ha d'inserir tenint en compte que és un arbre de cerca $\Rightarrow$ si el node és més petit que l'arrel baixar per l'esquerra sinó per la dreta.
- D'altra banda, cal assegurar que l'arbre queda equilibrat, reestructurant-lo si cal.

El desequilibri es produeix quan existeix un subarbre A de l'arbre que es troba en qualsevol del dos casos següents:

1. El subarbre dret de A té una alçada superior en una unitat al subarbre esquerre de A i el node corresponent s'insereix al subarbre dret de A, llavors provoca un increment en 1 de la seva alçada.



2. El subarbre esquerre de A té una alçada superior en una unitat al subarbre dret de A i el node corresponent s'insereix al subarbre esquerre de A, llavors provoca un increment en 1 de la seva alçada.



Notem que ambdues situacions són simètriques i, per això ens centrarem només en la primera. Hi ha dos subcasos:

- Cas DD (Dreta-Dreta)
- Cas DE (Dreta-Esquerre)

6.7.2.1 Cas Dreta-Dreta

El node s'insereix en el subarbre dret del subarbre dret. Podem observar en les figures 6.7 i 6.8 que el recorregut en inordre abans i després de la reestructuració és el mateix: $\alpha A \beta B \delta$

6.7.2.2 Cas Dreta-Esquerre

El node s'insereix en el subarbre esquerre del subarbre dret. La reestructuració es farà en dos passos:

1. Primer s'efectua una rotació cap a la dreta al voltant del node B.
2. Després s'efectua una rotació cap a l'esquerra al voltant del node A.

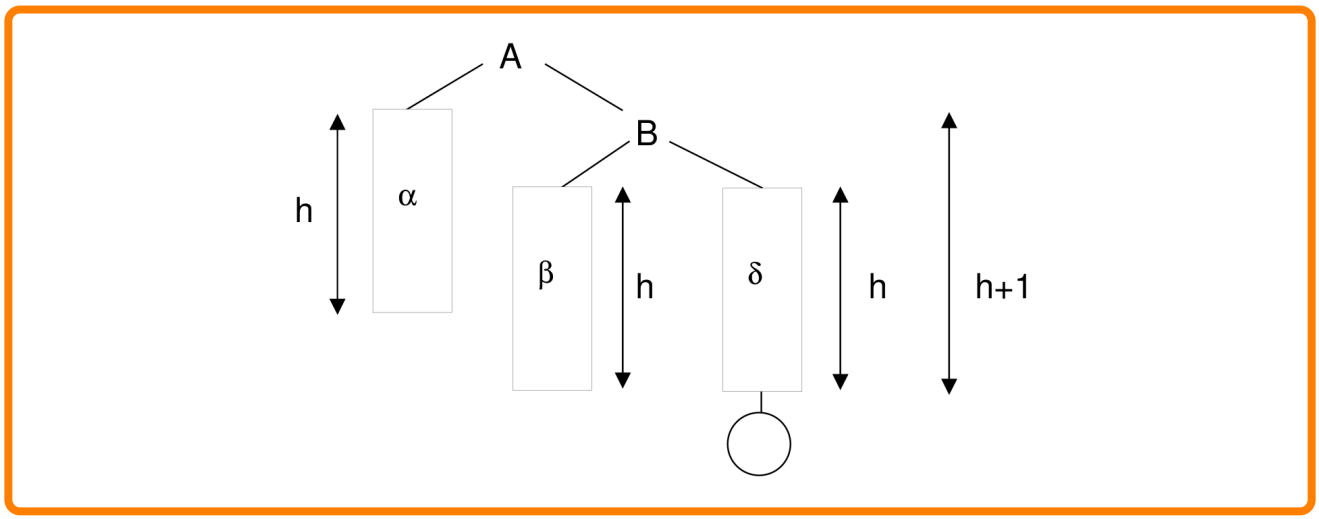


Figura 6.7: Inserir en un AVL cas DD.

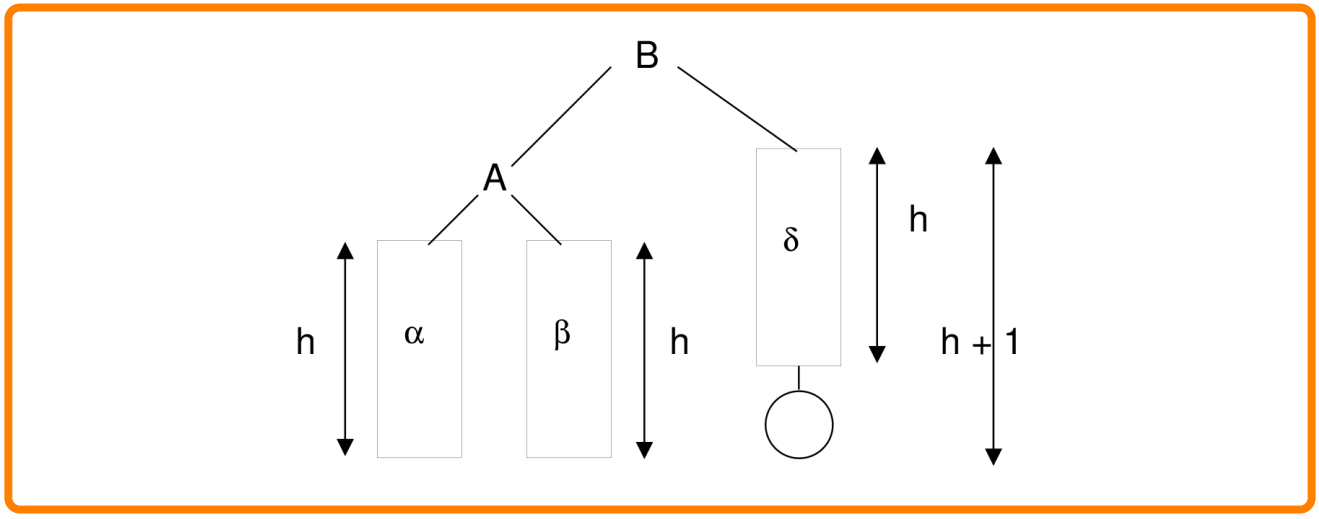


Figura 6.8: Inserir en un AVL cas DD. Reestructuració

Podem observar en les figures 6.9, 6.10 i 6.11 que el recorregut en inordre abans i després de la reestructuració és el mateix: $\alpha A \beta C \delta B \epsilon$

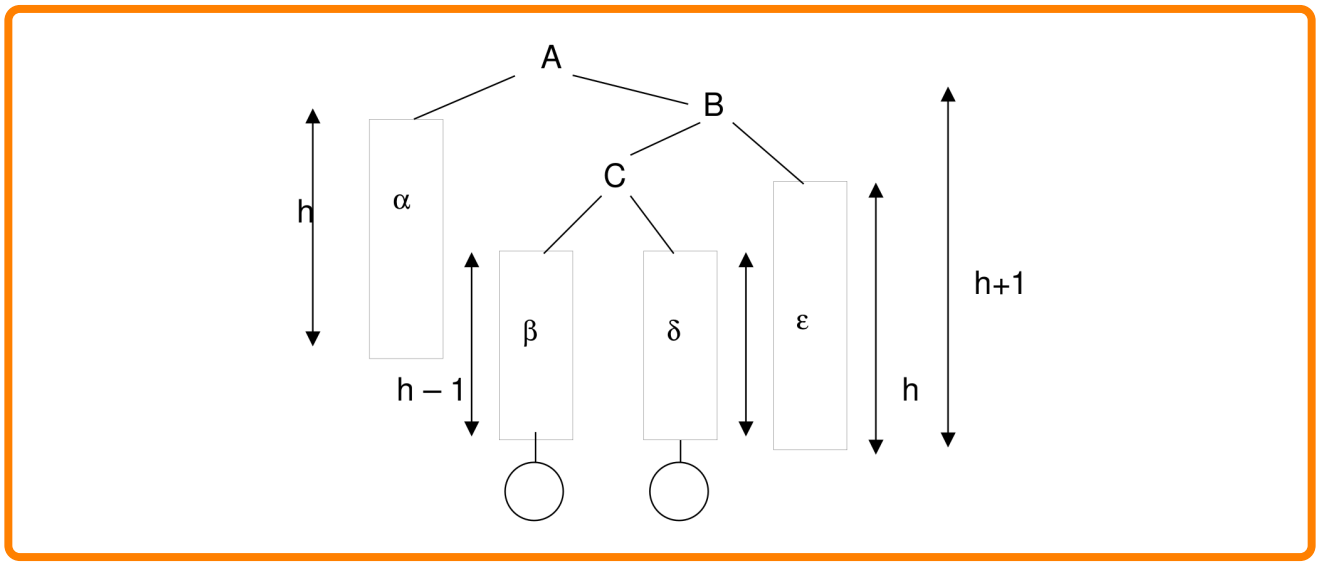


Figura 6.9: Inserir en un AVL cas DE.

Mètode privat per fer rotació dreta al subarbre apuntat per y

```
template <typename Clau, typename Valor>
node* dicc<Clau, Valor>::rotacio_dreta(node *y)
{
  node *x = y->_esq;
  node *T2 = x->_dret;

  // Realitzem la rotació
  x->_dret = y;
  y->_esq = T2;

  // Actualitzem les altures
  y->_altura = max(altura(y->_esq), altura(y->_dret)) + 1;
  x->_altura = max(altura(x->_esq), altura(x->_dret)) + 1;
  // Retornem la nova arrel
  return x;
}
```

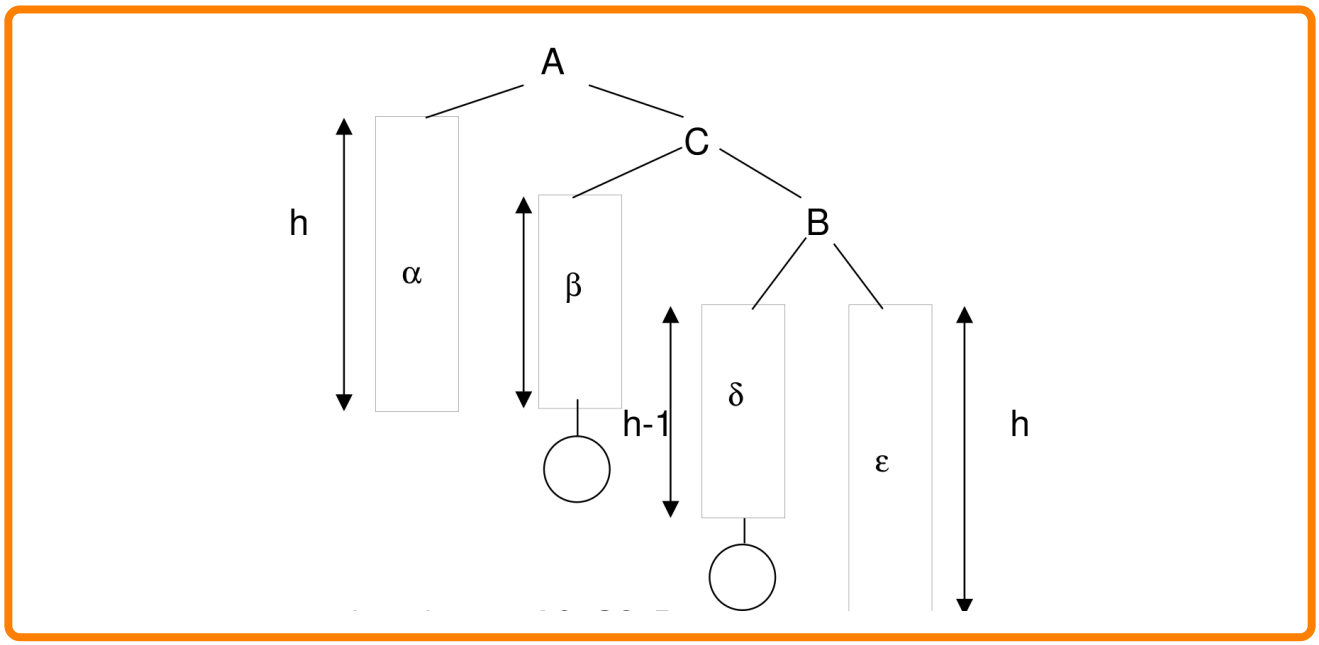


Figura 6.10: Inserir en un AVL cas DE. Reestructuració pas 1

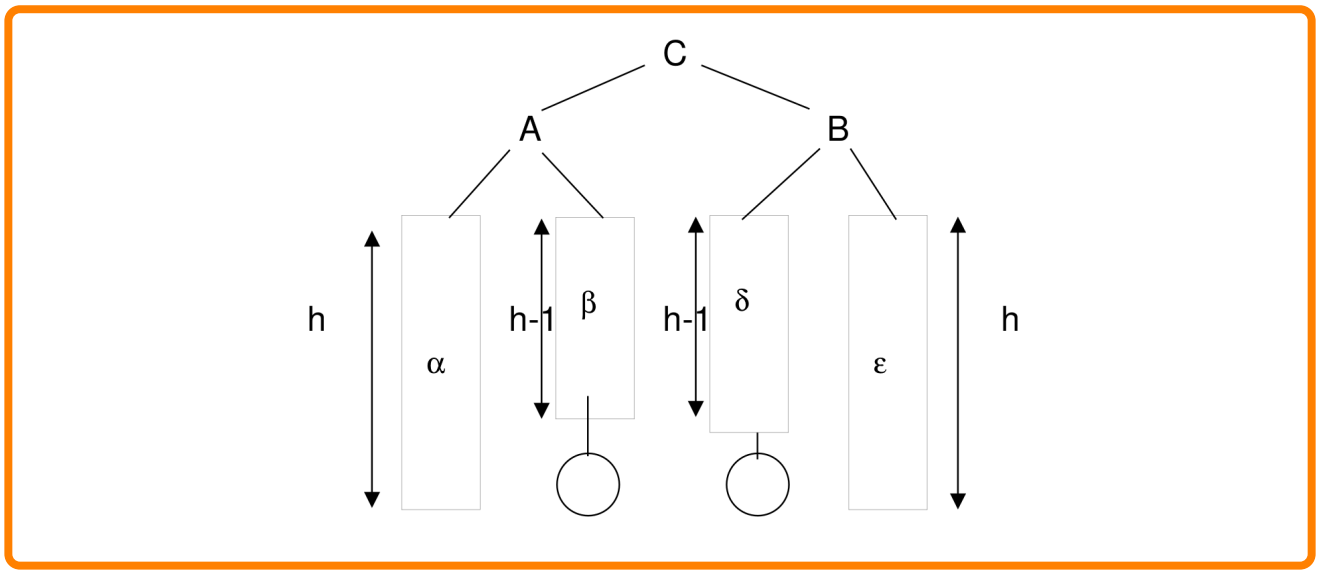


Figura 6.11: Inserir en un AVL cas DE. Reestructuració pas 2

Mètode privat per fer rotació esquerra al subarbre apuntat per x

```
template <typename Clau, typename Valor>
node* dicc<Clau, Valor>::rotacio_esquerra(node *x)
{
  node *y = x->_dret;
  node *T2 = y->_esq;

  // Realitzem la rotació
  y->_esq = x;
  x->_dret = T2;

  // Actualitzem les altures
  x->_altura = max(altura(x->_esq), altura(x->_dret)) + 1;
  y->_altura = max(altura(y->_esq), altura(y->_dret)) + 1;

  // Retornem la nova arrel
  return y;
}

template <typename Clau, typename Valor>
typename dicc<Clau, Valor>::node*
dicc<Clau, Valor>::insereix_bst(node *n, const Clau &k, const
{
  // 1. Fem la inserció a un BST normal
  if (n == nullptr)
    return new node(k, v);
  else if (k < n->_k)
    n->_esq = insereix_bst(n->_esq, k, v);
  else if (k > n->_k)
    n->_dret = insereix_bst(n->_dret, k, v);
  else {
    n->_v = v;
    return n;
  }
```

```
  // 2. Actualitzem l'altura
  n->_altura = max(altura(n->_esq), altura(n->_dret)) + 1;

  // 3. Obtenim el factor d'equilibri per veure si està balancejat
  int fe = factor_equilibri(n);

  // Cas EE
  if (fe > 1 && k < n->_esq->_k)
    return rotacio_dreta(n);

  // Cas DD
  if (fe < -1 && k > n->_dret->_k)
    return rotacio_esquerra(n);

  // Cas ED
  if (fe > 1 && k > n->_esq->_k)
  {
    n->_esq = rotacio_esquerra(n->_esq);
    return rotacio_dreta(n);
  }

  // Cas DE
  if (fe < -1 && key < n->_dret->_k)
  {
    n->_dret = rotacio_dreta(n->_dret);
    return rotacio_esquerra(n);
  }

  return n;
}
```

6.7.3 Supressió en un arbre AVL

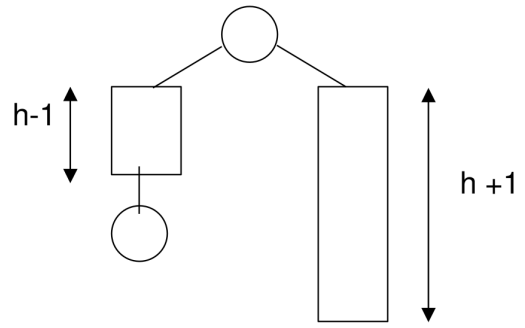
Els dos cassos possibles de desequilibri a la supressió són equivalents al procés d'inserció però ara el desequilibri es produeix perquè l'alçada d'un subarbre disminueix per sota del màxim tolerat. Els casos són:

1. El subarbre dret d'A té una alçada superior en una unitat al subarbre esquerre d'A i el node corresponent s'elimina del subarbre esquerre.

2. El subarbre esquerre d'A té una alçada superior en una unitat al subarbre dret d'A i el node corresponent s'elimina del subarbre dret, llavors provoca un decrement en 1 de la seva alçada.

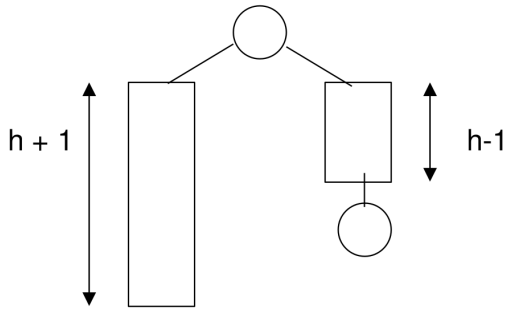
Ara els veurem amb detall.

1. El subarbre dret d'A té una alçada superior en una unitat al subarbre esquerre d'A i el node corresponent s'elimina del subarbre esquerre, llavors provoca un decrement en 1 de la seva alçada.



2. El subarbre esquerre de A té una alçada superior en una unitat al subarbre dret de A i el node corresponent s'elimina del subarbre dret, llavors provoca un decrement en 1 de la seva alçada.

Notem que ambdues situacions són simètriques i, per això ens centrarem només en la primera. Hi ha tres subcasos.



6.7.3.1 Cas Dreta-Dreta amb manteniment de l'altura

Sigui B el subarbre dret de A. Si els dos subarbres de B són de la mateixa alçada ens trobem amb el desequilibri Dreta-Dreta del cas de la inserció, que es resol de la mateixa manera. L'arbre resultant té la mateixa alçada abans i després de la supressió, per la qual cosa n'hi ha prou amb aquesta rotació per restablir l'equilibri. Veure les figures 6.12 i 6.13.

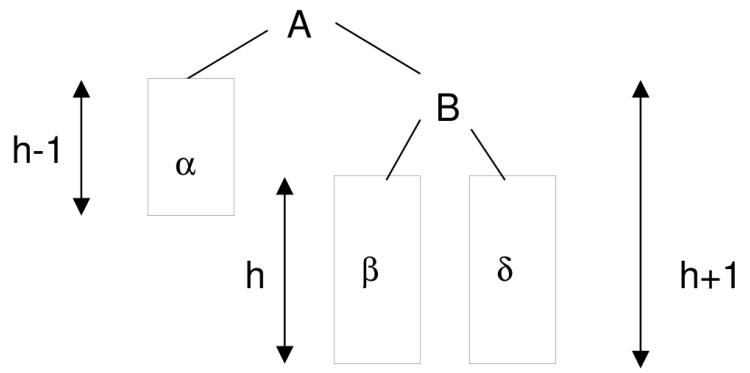


Figura 6.12: Eliminar en un AVL cas DD mateixa altura.

6.7.3.2 Cas Dreta-Dreta amb disminució de l'altura

Sigui B el subarbre dret de A. Si la alçada del subarbre esquerre de B és menor que la alçada del subarbre dret, la rotació és

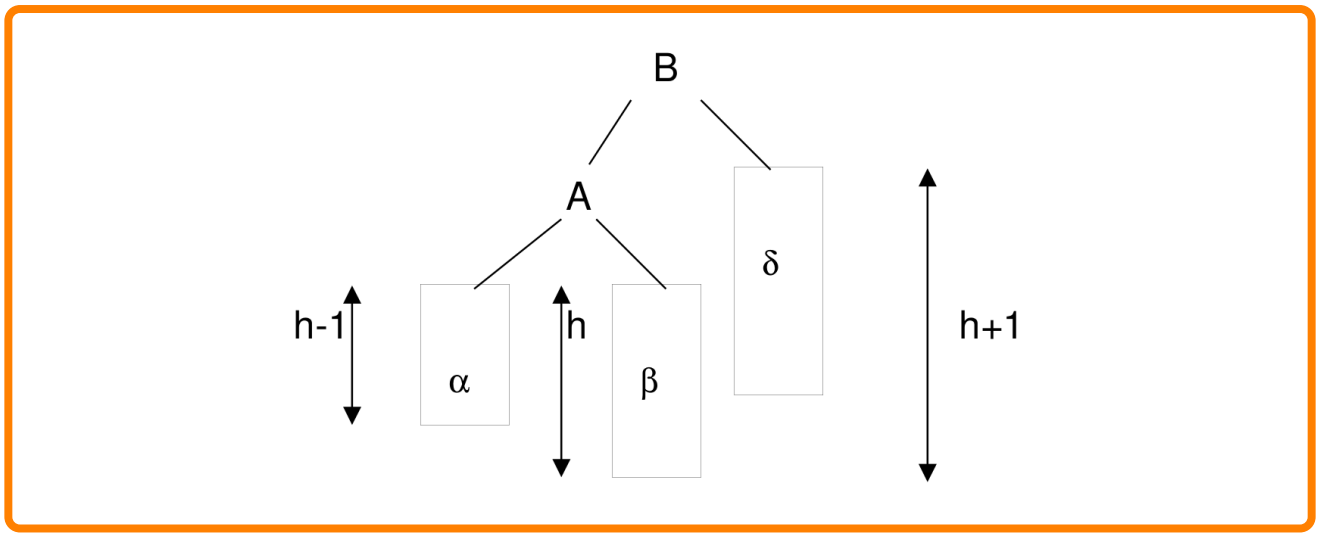


Figura 6.13: Eliminar en un AVL cas DD mateixa altura. Reestructuració

exactament la mateixa que abans. Però l'alçada de l'arbre resultat és una unitat més petita que abans de la supressió. Aquest fet és rellevant, perquè obliga a examinar si algun subarbre que l'engloba també es desequilibra. Veure les figures 6.14 i 6.15.

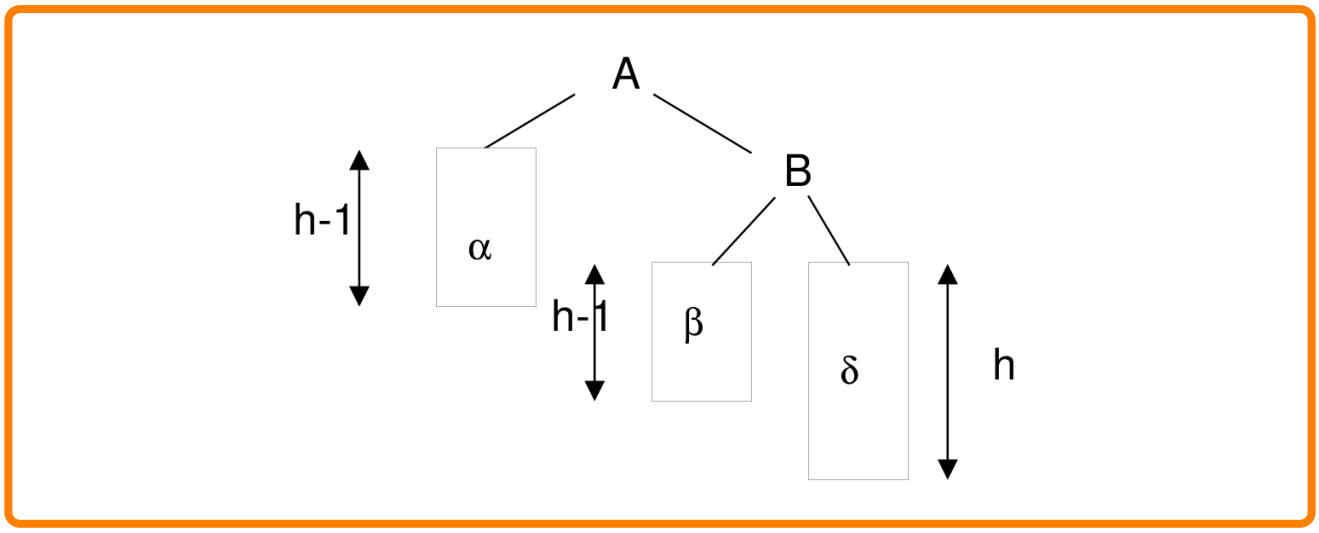


Figura 6.14: Eliminar en un AVL cas DD diferent altura.

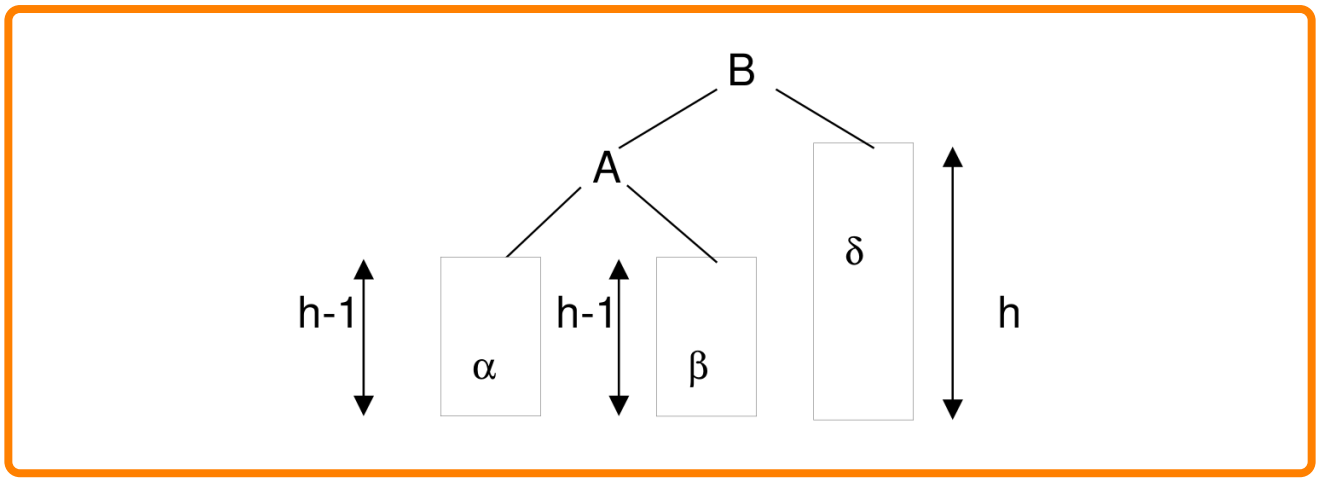


Figura 6.15: Eliminar en un AVL cas DD diferent altura. Reestructuració

6.7.3.3 Cas Dreta-Esquerre

Sigui B el subarbre dret de A. Si la alçada del subarbre esquerre de B és major que la alçada del subarbre dret, la rotació és similar al cas Dreta-Esquerre. També aquí l'alçada de l'arbre resultat és una unitat més petita que abans de la supressió. Aquest fet és rellevant, perquè obliga a examinar si algun subarbre que l'engloba també es desequilibra. Veure les figures 6.16 i 6.17, on la restructuració s'ha fet amb dos passos: primer una rotació dreta en el subarbre amb arrel B i després una rotació esquerra en el subarbre amb arrel A.

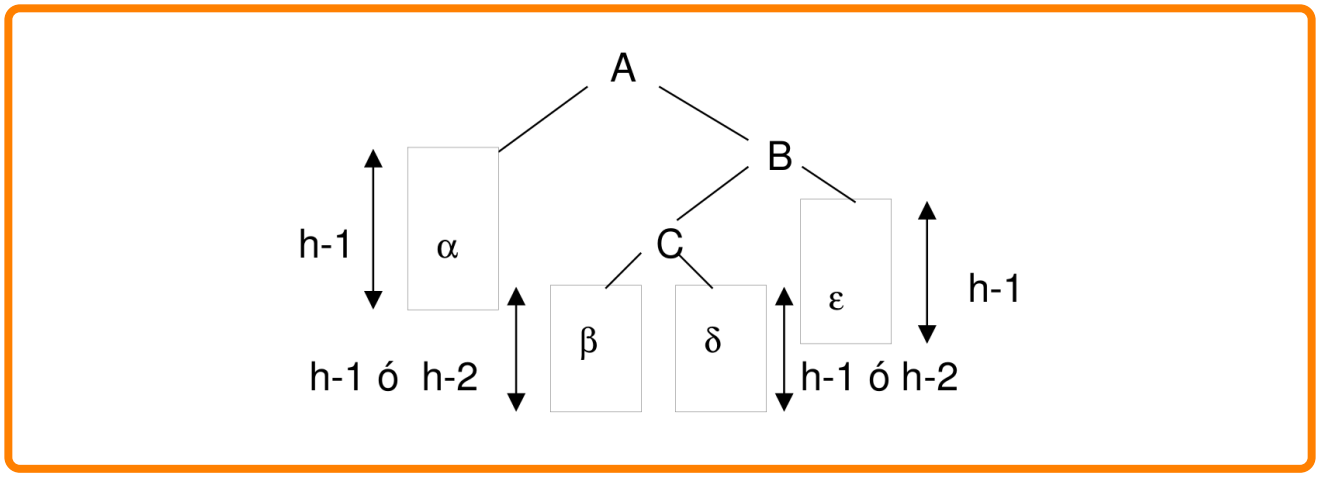


Figura 6.16: Eliminar en un AVL cas DE.

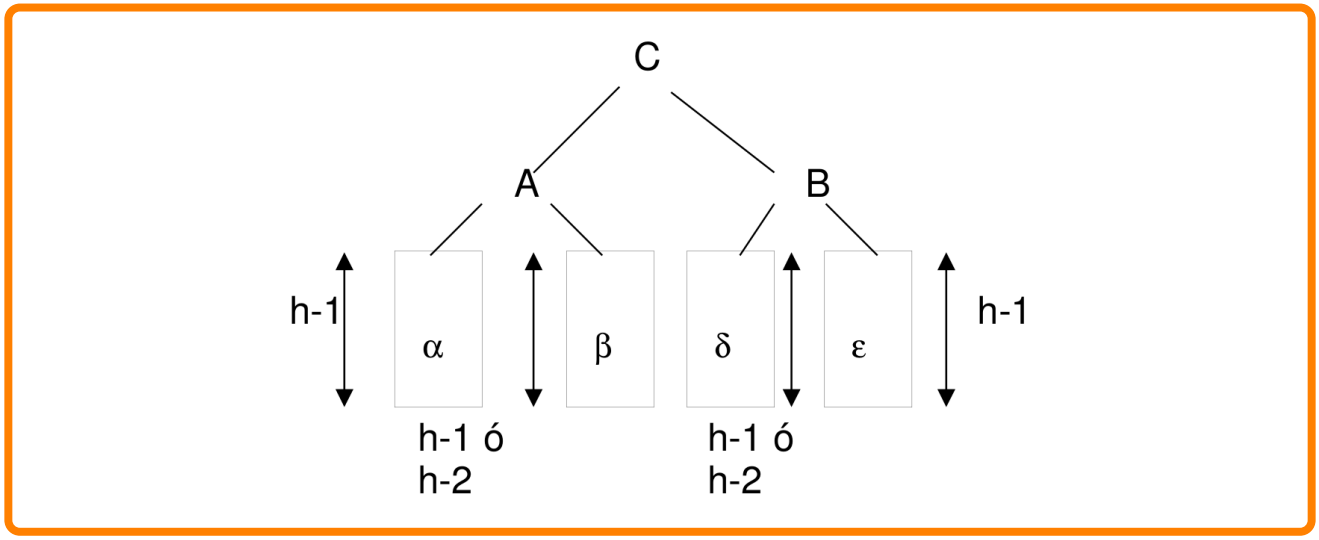


Figura 6.17: Eliminar en un AVL cas DE. Reestructuració

6.8 Algorisme d'ordenació quicksort

6.8.1 Introducció

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

La tasca important del *quicksort* és la partició. Donat un element p (per exemple el primer) anomenat **pivot**, es reorganitzen els elements col·locant a l'esquerra del pivot els més petits i a la dreta els més grans.

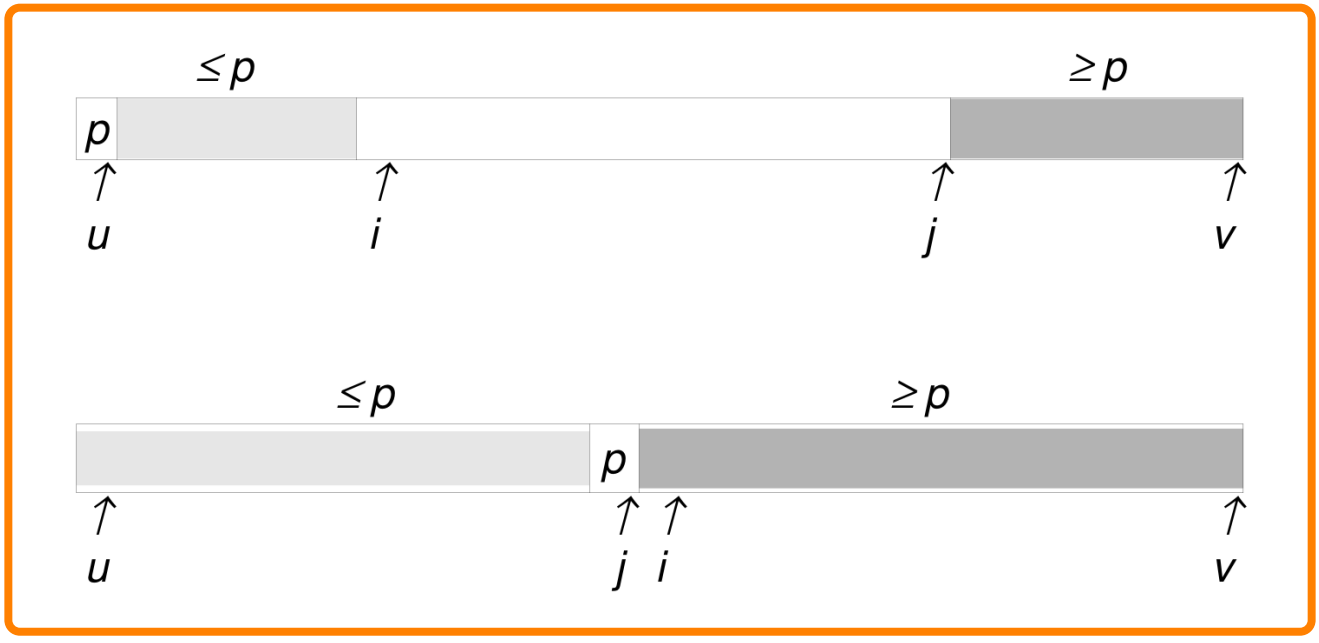


Figura 6.18: Quicksort. Evolució de la partició

Després 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.

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

6.8.2 Implementació

Acció per ordenar el tros de vector $A[u..v]$

```
template <typename T>
void quicksort(T A[], nat u, nat 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. Una bona alternativa és, enlloc d'ordenar per inserció cada tros de M o menys elements dins del mètode *quicksort*, ordenar per inserció tot el vector sencer al final de tot:

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

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. S'ha estimat que l'elecció òptima pel llindar M és entre 20 i 25.

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

- S'agafa com a pivot p el primer element $A[u]$.
- 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.
- 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 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){
      T aux = A[j];      // intercanvi
      A[j] = A[i];
      A[i] = aux;
    }
  }
  T aux = A[u]; // intercanvi pivot i últim dels menors
  A[u] = A[j];
  A[j] = aux;
  return j;
}
```

6.8.3 Cost

El cost de *quicksort* en cas pitjor és $\Theta(n^2)$. Això passa si en la majoria de 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 $\Theta(n \cdot logn)$.

6.9 Taules de dispersió

6.9.1 Definició

Definició 3: Taula de dispersió

Una **taula de dispersió** (anglès: **hash table** o **taula d'adreçament calculat**) emmagatzema un conjunt d'elements identificats amb una clau:

- dins d'una taula $D[0 \,..\, M-1]$
- mitjançant una **funció de dispersió** (hash function) h que va del conjunt de claus K fins al conjunt de posicions de la taula D:

$$\begin{array}{rrcl} & claus & \rightarrow & valors\ de\ dispersio\ (posicions\ taula) \\ h: & K & \rightarrow & 0 \,..\, M-1 \end{array}$$

Idealment la funció de dispersió h hauria de ser injectiva (a cada clau li correspondria una posició de la taula diferent). Normalment això no serà així i a claus diferents els hi pot correspondre la mateixa posició de la taula.

Donades dues claus x i y diferents, es diu que x i y són **sinònims** o que han produït una **col·lisió** si $h(x) = h(y)$.

Si la funció de dispersió dispersa eficaçment hi hauran poques col·lisions (la probabilitat de que moltes claus tinguin el mateix valor de dispersió serà baixa) i la idea de les taules de dispersió seguirà sent bona.

Per implementar eficaçment les taules de dispersió cal resoldre dues qüestions:

1. Dissenyar funcions de dispersió h que dispersin eficaçment i siguin relativament ràpides de calcular.

2. Definir estratègies per resoldre les col·lisions que apareguin.

A continuació veurem com resoldre aquests dos punts.

El **factor de càrrega** ($\alpha = \#claus/M$) (anglès: **load factor**) és una mesura de l'ocupació de la taula de dispersió. Quan se supera aquest llindar caldria aplicar una redispersió en la taula per tal d'incrementar la seva mida (per més informació veure l'apartat 6.9.5).

El nombre esperat de claus de la taula de dispersió i el factor de càrrega haurien de tenir-se en compte per inicialitzar la capacitat inicial de la taula per minimitzar el nombre de redispersions. Si la capacitat inicial és més gran que el nombre de claus dividit pel factor de càrrega, no seran necessàries operacions de redispersió.

6.9.2 Especificació

Atès que es desconeix el tipus de la clau cal encapsular la funció de dispersió dins d'una classe `Hash` (que s'ha d'implementar per cada tipus de Clau). La classe `Hash<T>` defineix l'`operator()` de manera que si `h` és un objecte de la classe `Hash<T>` i `x` és un objecte de la classe `T`, podem aplicar `h(x)`. Aquesta crida ens retornarà un nombre enter, el valor de dispersió.

```
template <typename T>
class Hash {
public:
  int operator()(const T &x) const throw();
};

template <typename Clau, typename Valor,
          typename HashFunct = Hash>
class dicc {
public:
  ...
  void consulta(const Clau &k, bool &hi_es, Valor &v)
       const throw(error);
  void insereix(const Clau &k, const Valor &v)
       throw(error);
  void elimina(const Clau &k) throw();
  ...

private:
  struct node_hash {
    Clau _k;
    Valor _v;
    ...
  };

  node_hash *_taula; // taula amb les parelles clau-valor
  //node_hash **_taula; // o taula de punters a node

  nat _M;         // mida de la taula
  nat _quants;    // nº d'elements guardats al diccionari

  static int hash(const Clau &k) throw() {
    HashFunct<Clau> h;
    return h(k) % _M;
  }
};
```

6.9.3 Funcions de dispersió

La funció de dispersió ha de complir les següents propietats:

- **Distribució uniforme**: Volem que tots els valors de dispersió tinguin el mateix nombre de claus associades.
- **Independència de l'aparença de la clau**: Ens interessa que claus similars no tinguin valors de dispersió iguals.
- **Exhaustivitat**: Volem que tot valor de dispersió tingui com a mínim una clau associada. Així cap posició de la taula de dispersió queda desaprofitada a priori.
- **Rapidesa de càlcul**.

La construcció de bones funcions de dispersió no és fàcil i requereix coneixements sòlids de vàries especialitats de les matemàtiques. Està molt relacionada amb la construcció de generadors de números pseudoaleatoris.

Una manera senzilla de construir funcions de dispersió i que dóna bons resultats és:

1. Calcular un número enter positiu a partir de la representació binaria de la clau (per ex. sumant el bytes que la composen).
2. Al resultat anterior fer mòdul M, sent M la grandària de la taula. Així obtenim una natural dins del rang $[0 \,..\, M-1]$. Es recomana que M sigui un número primer.

L'operació privada `hash` de la classe `dicc` calcularà el mòdul $h(x)\%M$ de manera que obtindrem un índex vàlid de la taula, entre 0 i $M-1$.

Per tal que la funció de dispersió h dispersi millor es pot sofisticar una mica més utilitzant alguna de les següents estratègies:

- **Suma ponderada de tots els bytes/caràcters/dígits**:

  $s = s + codi\_ascii(s[i]) * b^i$

  sent b la base utilitzada

  Això evita que el valor d'un byte sigui el mateix independent de la posició on aparegui dins la clau. Però és més ineficient doncs es necessita calcular una potència i un producte. Una bona opció és agafar $b = 2$.

- **Desplegament**: Es parteix la clau K en m parts de la mateixa longitud i es combinen aquestes parts de determinada manera (sumes, OR lògic, XOR lògic). Després es fa mòdul M. Exemple:

  $K = 123456 \;\; m = 3 \;\; hash(K) = (12 + 34 + 56)\ mod\ M$

- **Quadrat**: S'eleva al quadrat el número representat pels m bits/bytes centrals de la clau K. Finalment es fa mòdul M. Aquest és un exemple d'una transformació no lineal.

  Si M és una potència de 2, es pot agafar directament els $log_2(M)$ bits centrals de K^2. Exemple:

  $K = 12 \;\; K^2 = 144 = 10010000$

  si $M = 32$ agafarem els 5 bits centrals: 01000

Caldria tenir definit una especialització de la classe Hash per la nostra classe Clau. Per exemple si la classe Clau fos igual a un string o un enter unes possibles especialitzacions serien:

Especialització del template Hash per T = string: Suma ponderada dels caràcters.

```
template <>
class Hash<string> {
public:
  int operator()(const string &x) const throw() {
    nat n = 0;
    for (nat i=0; i < x.length(); ++i) {
      n = n + x[i]*(i+1);        // n acumula el codi ascii
    }
    return n;
  }
};
```

Especialització del template Hash per T = int: Bits centrals del quadrat del número multiplicat per PI.

```
template <>
class Hash<int> {
static long const MULT = 31415926;
public:
  int operator()(const int &x) const throw() {
    long y = ((x * x * MULT) << 20) >> 4;
    return y;
  }
};
```

6.9.4 Estratègies de resolució de col·lisions

Existeixen dues grans famílies per organitzar una taula de dispersió que resolgui les col·lisions. Per cadascuna d'elles estudiarem una estratègia en concret:

- **Dispersió oberta** (anglès: open hashing)

  - **Sinònims encadenats** (anglès: separate chaining) indirectes i directes: Els sinònims es guarden formant llistes encadenades.

- **Direccionament obert** (anglès: open addressing)

  - **Sondeig lineal** (anglès: linear probing): Si una component de la taula ja està ocupada es busca una altra que estigui lliure. El sondeig lineal és la tècnica més senzilla: busca els llocs lliures de la taula de forma correlativa.

6.9.4.1 Sinònims encadenats indirectes

Cada entrada a la taula apunta a una llista encadenada de sinònims. S'anomena indirecte perquè la taula no guarda els elements en si, sinó un enllaç (punter) al primer element de la llista de sinònims.

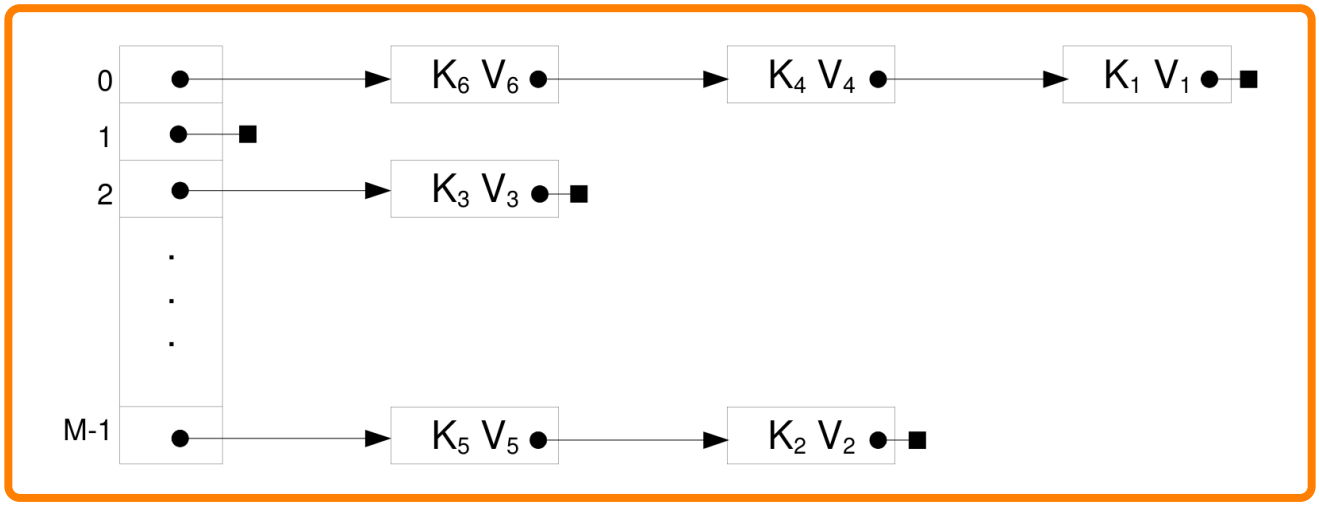


Figura 6.19: Taula de dispersió. Encadenaments indirectes

Nota: Els índexs i de K_i V_i indiquen un possible ordre d'inserció.

La representació d'aquesta classe és la següent:

```
template <typename Clau, typename Valor>
class dicc {
  ...
private:
  struct node_hash {
    Clau _k;
    Valor _v;
    node_hash* _seg;
    node_hash(const Clau &k, const Valor &v,
              node_hash* seg = nullptr) throw(error);
  };
  node_hash **_taula;  // taula amb punters a les llistes
  nat _M;              // mida de la taula
  nat _quants;         // nº d'elements guardats al diccionari
  ...
};
```

Una possible implementació d'aquesta classe es pot veure a continuació.

```
template <typename Clau, typename Valor, typename HashFunct>
dicc<Clau, Valor, HashFunct>::dicc() throw(error) : _quants(0)
  _M = 53;
  _taula = new node_hash*[_M];
  for (int i=0; i < _M; ++i) {
    _taula[i] = nullptr;
  }
}
```

Constructora del node.

```
template <typename Clau, typename Valor, typename HasFunct>
dicc<Clau, Valor, HashFunct>::node_hash::node_hash (const Clau
        : _k(k), _v(v), _seg(seg) {
}
```

La consulta d'un element és ben simple: s'accedeix a la llista apropiada mitjançant la funció de hash, i es realitza un recorregut seqüencial de la llista fins que es troba un node amb la clau donada o s'ha examinat tota la llista.

```
template <typename Clau, typename Valor, typename HasFunct>
void dicc<Clau, Valor, HasFunct>::consulta (const Clau &k, boo
  int i = hash(k);
  node_hash* p = _taula[i];
  hi_es = false;
  while (p != nullptr and not hi_es) {
    if (p->_k == k) {
      hi_es = true;
      v = p->_v;
    }
    else {
      p = p->_seg;
    }
  }
}
```

Per fer la inserció s'accedeix a la llista corresponent mitjançant la funció de hash, i es recorre per determinar si ja existia o no un element amb la clau donada. En el primer cas, es modifica el valor associat; i en el segon s'afegeix un nou node a la llista.Donat que les llistes de sinònims contenen generalment pocs elements el més efectiu és efectuar les insercions a l'inici.

```
template <typename Clau, typename Valor, typename HasFunct>
void dicc<Clau, Valor>::insereix (const Clau &k,
const Valor &v) throw(error) {
  int i = hash(k);
  node_hash *p = _taula[i];
  bool trobat = false;
  while (p != nullptr and not trobat) {
    if (p->_k == k) {
      trobat = true;
```

```
    }
    else {
      p = p->_seg;
    }
  }
  if (trobat) {
    // Només canviem el valor associat
    p->_v = v;
  }
  else {
    // Cal crear un nou node i l'afegim al principi
    _taula[i] = new node_hash(k, v, _taula[i]);
    ++_quants;
  }
}
```

En l'eliminació cal controlar si l'element esborrat era el primer de la llista per actualitzar correctament la taula.

```
template <typename Clau, typename Valor, typename HasFunct>
void dicc<Clau, Valor, HashFunct>::elimina (const Clau &k) thr
  nat i = hash(k);
  node_hash *p = _taula[i], *ant=nullptr;
  bool trobat = false;
  while (p != nullptr and not trobat) {
    if (p->_k == k) {
      trobat = true;
    }
    else {
      ant = p;
      p = p->_seg;
    }
  }
  if (trobat) {
    if (ant == nullptr) {
      _taula[i] = p->seg;        // Era el primer
    }
    else        {
```

```
        ant->seg = p->seg;
      }
      delete(p);
      --_quants;
    }
}
```

Algunes consideracions generals:

- Les llistes estan simplement encadenades i no fa falta utilitzar fantasmes ni tancar-les circularment. De fet aquestes llistes en general tindran pocs elements.
- Es pot considerar la possibilitat d'ordenar els sinònims de cada llista per clau. Això permet reduir el cost de les cerques sense èxit.
- Si el factor de càrrega $\alpha = \_quants/\_M$ és un valor raonablement petit podem considerar que el cost mitjà de totes les operacions del diccionari (excepte la constructora) és $\Theta(1)$.
- L'inconvenient de les taules de dispersió encadenades indirectes és que l'accés al primer element de la llista de sinònims no és immediat i que ocupa força espai degut als encadenaments.

6.9.4.2 Sinònims encadenats directes

La taula està dividida en dues zones:

- La zona principal de M elements: La posició i o bé està buida o bé conté un parell $< k, v >$ tal que $hash(k) = i$.

- La zona d'excedents que ocupa la resta de la taula i guarda les claus sinònimes. Quan es produeix una col·lisió, el nou sinònim passa a aquesta zona. Tots els sinònims d'un mateix valor de dispersió estan encadenats (ara els encadenaments són enters).

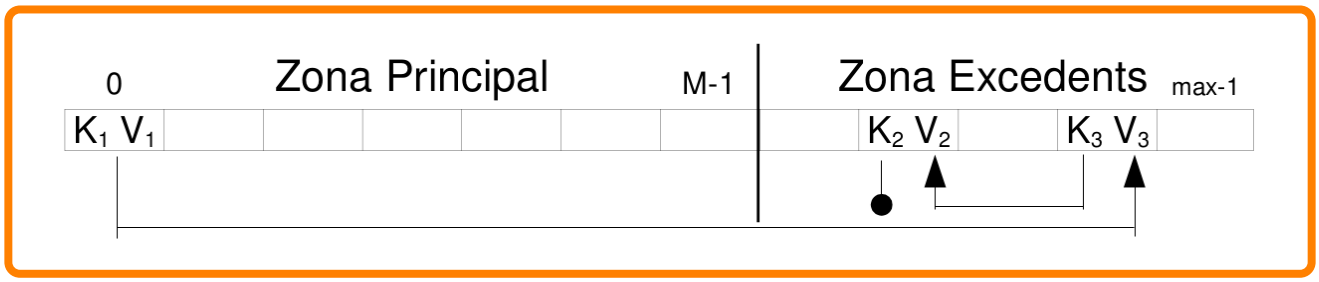


Figura 6.20: Taula de dispersió. Encadenaments directes

S'anomena directe perquè la taula guarda realment els elements de manera que ens estalviem un accés respecte a les indirectes. Així l'accés al primer element de la llista de sinònims és immediat.

La codificació de les operacions és una mica més complicat que en el cas de taules encadenades indirectes doncs cal distingir si una posició és lliure o no i també cal fer la gestió de les posicions lliures de la zona d'excedents (per ex. amb una llista encadenada dels llocs lliures).

Cal anar amb molt de compte amb l'eliminació amb els sinònims encadenats directament:

- Quan eliminem un element de la zona d'excedents l'eliminem de la llista de sinònims incorporant-lo a la llista de llocs lliures.

- Quan eliminem un element de la zona principal cal evitar de perdre l'encadenament amb els sinònims. Es podem utilitzar dues estratègies:

1. **Traslladar el primer element sinònim** des de la zona d'excedents a la zona principal i eliminar aquest. No convé si el tipus clau+valor ocupa molt d'espai.
2. **Marcar l'element com a esborrat** (s'interpreta com ocupat per les consultes i eliminacions i lliure per les insercions). L'inconvenient és que es perd l'avantatge de l'accés directe.

Per obtenir bons resultats la zona d'excedents no hauria de passar del 14% de la dimensió total de la taula.

6.9.4.3 Direccionament obert: Sondeig lineal

En les estratègies de direccionament obert els sinònims es guarden dins de la mateixa taula de dispersió.

Per a cada clau K es defineix una seqüència de posicions $i_0 = h(K),\ i_1,\ i_2,\ \ldots$ que determinen on pot estar (consulta) o on anirà a parar (inserció) la clau K.

Hi ha diferents estratègies segons la seqüència de posicions sondejades. La més senzilla de totes és el sondeig lineal:

$$i_0 = h(K),\ i_1 = (i_0 + 1)\ mod\ M,\ i_2 = (i_1 + 1)\ mod\ M,\ \ldots$$

```
template <typename Clau, typename Valor>
class dicc {
public:
  ...
private:
  enum Estat {lliure, esborrat, ocupat};
  struct node_hash {
    Clau _k;
    Valor _v;
    Estat _est;
  };

  node_hash *_taula;     // taula amb parells <k,v>
```

```
  nat _quants;   // nº d'elements guardats al diccionari
  nat _M;                    // mida de la taula

  int busca_node(const Clau &k) const throw();
};
```

En el cas de la inserció cal assegurar-se que almenys hi ha un lloc no ocupat dins de la taula abans d'inserir, `_quants < _M`. En cas contrari es pot generar un error o fer més gran la taula de dispersió (per més informació veure l'apartat 6.9.5).

Retorna la posició on es troba l'element amb la clau indicada o, en cas que no es trobi la clau, la primera posició no ocupada.

```
template <typename Clau, typename Valor>
int dicc<Clau, Valor>::busca_node(const Clau &k) const throw()
  nat i = hash(k);

  // prilliure és la primera posició esborrada que
  // trobem, val -1 si no trobem cap posició esborrada.

  int prilliure = -1;
  // comptem el nombre d'elements que visitem per només
  // fer una passada.
  nat cont = 0;
  while (((_taula[i]._est == ocupat and _taula[i]._k != k)
         or _taula[i]._est == esborrat) and cont < _M) {
    ++cont;
    if (_taula[i]._est == esborrat and prilliure == -1) {
      prilliure = i;
    }
    i = (i+1) % _M;
  }
  if (_taula[i]._est == lliure and prilliure != -1)
      i = prilliure;

  return i;
```

```
}

template <typename Clau, typename Valor>
void dicc<Clau, Valor>::insereix(const Clau &k, const Valor &v
  nat i = busca_node(k);
  if (_taula[i]._est == ocupat and _taula[i]._k != k) {
    // redispersió
  }
  if (_taula[i]._est != ocupat) {
    ++_quants;
  }
  _taula[i]._k = k;
  _taula[i]._v = v;
  _taula[i]._est = ocupat;
}

template <typename Clau, typename Valor>
void dicc<Clau, Valor>::consulta(const Clau &k, bool &hi_es, V
  nat i = busca_node(k);
  if (_taula[i]._est == ocupat and _taula[i]._k == k) {
    v = _taula[i]._v;
    hi_es = true;
  }
  else {
    hi_es = false;
  }
}
```

L'eliminació d'elements en taules de direccionament obert és complicada. Si una clau K col·lisiona i després de fer diferents sondeigs es diposita en una posició determinada, és perquè les posicions anteriors estaven ocupades. Si eliminem algun element de les posicions anteriors no podem marcar la posició com a LLIURE, doncs una cerca posterior de la clau K fracassaria.

Hem d'afegir un tercer estat i marcar la posició eliminada com ESBORRADA.

Una posició marcada com ESBORRADA:

- En les cerques es comporta com si fos ocupada.
- En les insercions es pot aprofitar per col·locar nous elements.

```
template <typename Clau, typename Valor>
void dicc<Clau, Valor>::elimina(const Clau &k) const throw() {
  nat i = busca_node(k);
  if (_taula[i]._est == ocupat and _taula[i]._k == k) {
    _taula[i]._est = esborrat;
  }
}
```

Les eliminacions degraden notablement el rendiment de les taules de direccionament obert ja que les posicions ESBORRADES compten igual que les realment ocupades quan es busquen elements.

L'avantatge de les taules de direccionament obert és que la seva representació és molt compacte (no ocupa espai de memòria addicional amb encadenaments). L'inconvenient és que es degeneren si es realitzen insercions i eliminacions alternativament o si el factor de càrrega és gran (α proper a 1).

Si $\alpha < 1$ el cost de les cerques amb èxit (i les modificacions) serà proporcional a

$$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$$

i el cost de les cerques sense èxit (i les insercions) serà proporcional a la fórmula

$$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$$

Un fenomen indesitjable que s'accentua quan α és proper a 1 és l'**apinyament** (anglès: **clustering**). Es produeix quan molts elements no poden ocupar la seva posició "preferida"a l'estar ocupada per un altre element que no té perquè ser un sinònim (són els ïnvasors"). Grups de sinònims més o menys dispersos acaben fonent-se en grans apinyaments. Llavors, quan busquem un clau, tenim que examinar no només els seus sinònims sinó altres claus que no tenen cap relació amb ella.

6.9.5 Redispersió

Si el nostre llenguatge de programació permet definir la grandària d'una taula en temps d'execució podem utilitzar la tècnica de **redispersió** (anglès: **rehash**).

Si el *factor de càrrega* $\alpha = \_quants/\_M$ és molt alt superant un cert llindar:

1. Es reclama a la memòria dinàmica una taula de grandària el doble de l'actual.

2. Es reinserta tota la informació que contenia la taula actual a la nova taula. Caldrà recórrer seqüencialment la taula actual i cadascun dels elements presents s'insereix a la nova taula utilitzant una nova funció de dispersió. Això té un cost proporcional a la grandària de la taula ($\Theta(n)$) però es fa molt de tant en tant.

La redispersió permet que el diccionari creixi sense límits prefixats, garantint un bon rendiment de totes les operacions i

sense malgastar massa memòria, doncs la mateixa tècnica es pot aplicar a la inversa, per evitar que el *factor de càrrega* sigui excessivament baix.

Pot demostrar-se que, encara que una operació individual d'inserció o eliminació en un diccionari pugui tenir cost $\Theta(n)$ degut a la redispersió, una seqüència de n operacions d'inserció o eliminació tindrà cost $\Theta(n)$ i, per tant, en terme mig cada operació té cost $\Theta(1)$.

6.10 Arbres digitals (Tries)

6.10.1 Definició i exemples

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. Per exemple, donada una col·lecció de paraules C i un prefix p retornar totes les paraules de C que comencen amb el prefix p.

Considerem que els símbols pertanyen a un **alfabet** finit de $m \geq 2$ elements $\Sigma = \{\delta_1, \delta_2, \ldots, \delta_m\}$.

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

- $\Sigma*$ denota el conjunt de les seqüències (cadenes) formades per símbols de Σ.

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

A partir d'un conjunt finit de seqüències $X \subset \Sigma*$ de idèntica longitud podem construir un **trie** T tal i com indica aquesta definició:

Definició 4: Trie

El **trie** T és un arbre m-ari definit recursivament com:

1. Si X és buit llavors T és un arbre buit.
2. Si X conté un sol element llavors T és un arbre amb un únic node que conté a l'únic element de X.
3. Si $|X| \geq 2$, sigui T_i el trie corresponent a $X_i = y/\delta_i\dot{y} \in X \wedge \delta_i \in \Sigma$. Llavors T és un arbre m-ari constituït per una arrel que té com a fills els m subarbres $T_1, T_2, ..., T_m$.

Exemple: Trie construït a partir d'un conjunt X de 10 seqüències, totes elles formades per 6 símbols binaris. Com que $m = 2$ el trie resultant és un arbre binari.

$X_1 = 000101$ $X_2 = 010001$ $X_3 = 101000$ $X_4 = 010101$
$X_5 = 110101$ $X_6 = 100111$ $X_7 = 110001$ $X_8 = 011111$
$X_9 = 001110$ $X_{10} = 100001$

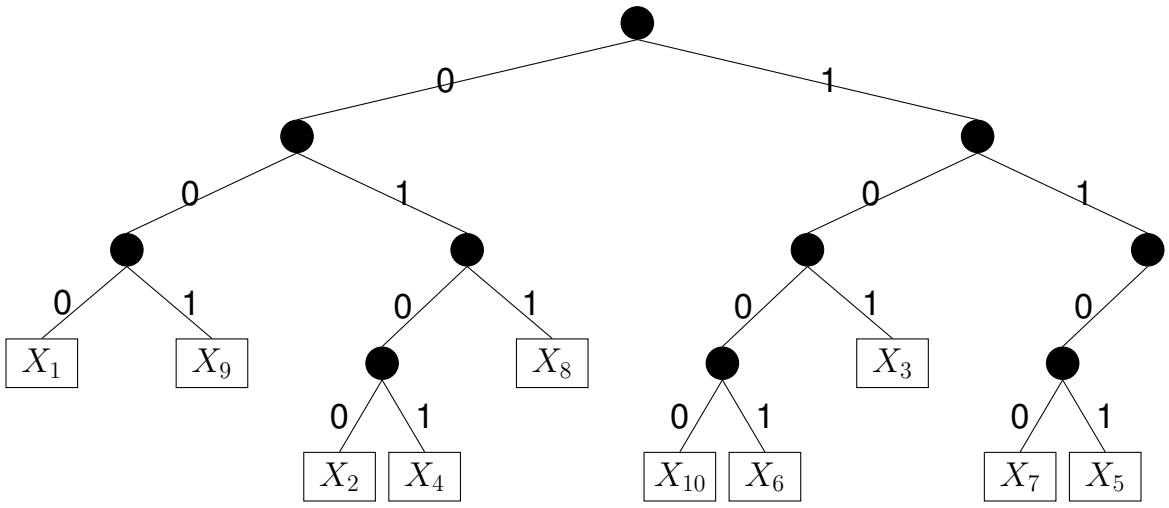


Figura 6.21: Exemple de Trie amb un alfabet binari.

Lema 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 δ_1, la que uneix l'arrel amb el segon subarbre s'etiqueta δ_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).

Lema 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).

Lema 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.

Lema 4: L'alçada d'un trie T és igual a la longitud mínima del prefix necessari per distingir qualsevol 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 $\leq l$.

La definició d'un trie imposa que totes les seqüències siguin d'igual longitud, el qual és molt restrictiu.

Si no exigim aquesta condició, com podem distingir dos elements x i y si x és prefix de y?

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.

Exemple: Trie construït a partir del conjunt de seqüències

X = {DIA, ARC, MEL, DEU, DO, DE, MAL, DOL, JO, MA} totes elles formades per símbols caràcters. Com que m=25 el trie resultant és un arbre 26-ari.

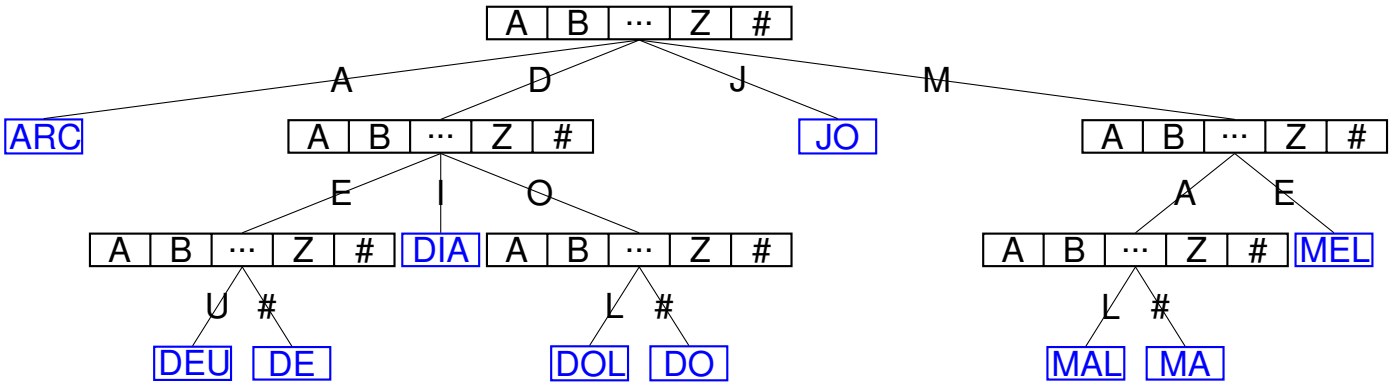


Figura 6.22: Exemple de Trie amb símbols caràcters i caràcter especial #.

6.10.2 Tècniques d'implementació

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

- Si s'utilitza un **vector de punters per node**, els símbols de Σ solen utilitzar-se com índexs (utilitzant una funció $index : \Sigma \rightarrow 1, \ldots, m$). Les fulles que contenen els elements de X poden emmagatzemar exclusivament els sufixes restants, ja que el prefix està ja codificat en el camí de l'arrel a la fulla (veure figura 6.23).

- Si s'utilitza la representació **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 acostuma a ordenar-se seguint aquest ordre.

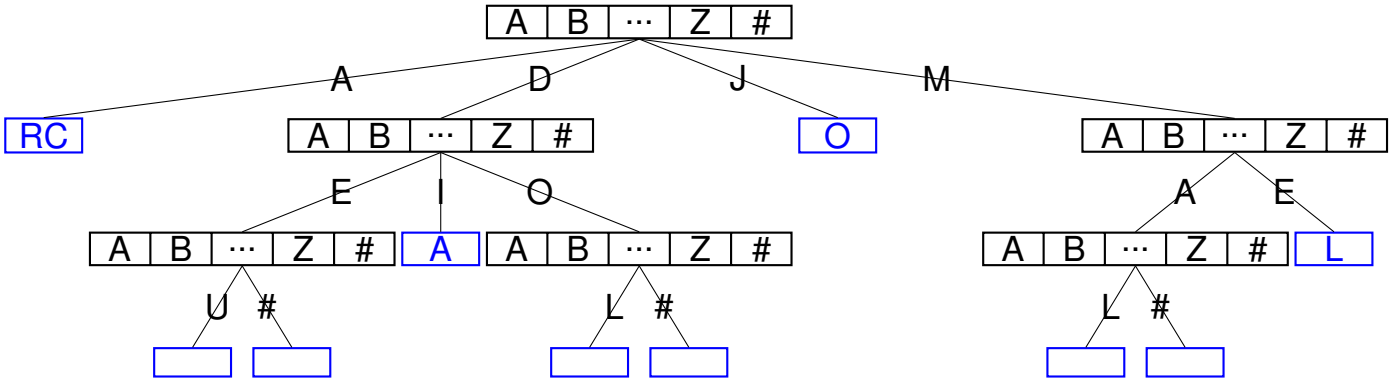


Figura 6.23: Exemple de Trie amb vectors de punters a node.

La figura 6.24 mostra un exemple de representació primer fill-següent germà en que, per simplicitat, s'utilitza el mateix tipus de node per guardar els sufixes restants de cada clau. Així evitem usar nodes i punters a nodes de diferent tipus.

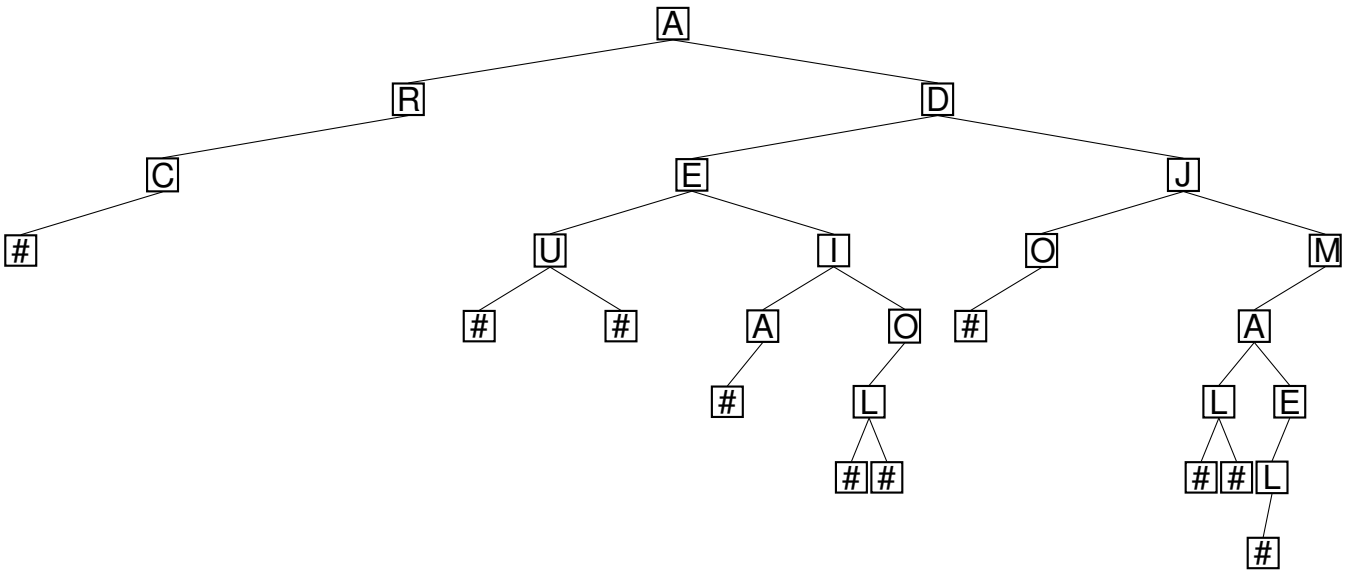


Figura 6.24: Exemple de Trie amb la representació primer fill-següent germà.

6.10.3 Implementació primer fill - següent germà

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

```
Retorna la longitud >= 0 de la clau.
```

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

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

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

Caldrà especialitzar la funció `especial` segons el tipus dels símbols de les claus del diccionari. Per exemple pel tipus `string` seria:

```
Retorna el símbol especial fi de clau.
```

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

IMPORTANT!!
Per tal de fer més llegible el codi dels diccionaris digitals en les implementacions dels mètodes s'han eliminat tots els paràmetres del `template`.

```
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: $\Theta(k.length())$

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

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

```
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
  node_trie *res = nullptr;
  if (n != nullptr) {
    if (i == k.length() and n->_c == especial<Clau>()) {
      res = n;
    }
    else if (n->_c > k[i]) {
      res = nullptr;
    }
    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;
}
```

6.10.4 Arbre ternari de cerca

Una alternativa que combina eficiència en l'accés als subarbres i estalvi de memòria consisteix en implementar cada node del trie com un BST. L'estructura resultant s'anomena **arbre ternari de cerca** (anglès: **ternary search tree** o **TST**) ja que cada node conté tres apuntadors:

- Dos punters al fill esquerra i fill dret del BST que conté els diferents símbols i-èssims de totes les claus que tenen el mateix prefix format per i-1 elements.
- Un punter (l'anomenem central) a l'arrel del subarbre que

conté, formant un BST, els símbols de la següent posició de totes les claus que tenen el mateix prefix.

Exemple: TST construït a partir del conjunt de seqüències X = {DIA, ARC, MEL, DEU, DO, DE, MAL, DOL, JO, MA} inserides en aquest mateix ordre. La forma del TST depèn de l'ordre en que s'han inserit les claus, igual que succeix amb els BSTs.

Per exemple es pot observar en la figura 6.25 com el primer símbol de les claus inserides (D, A, M, J) formen un BST utilitzant els dos punters al fill esquerra i fill dret de cada node. I el punter fill central avança al següent símbol de la clau.

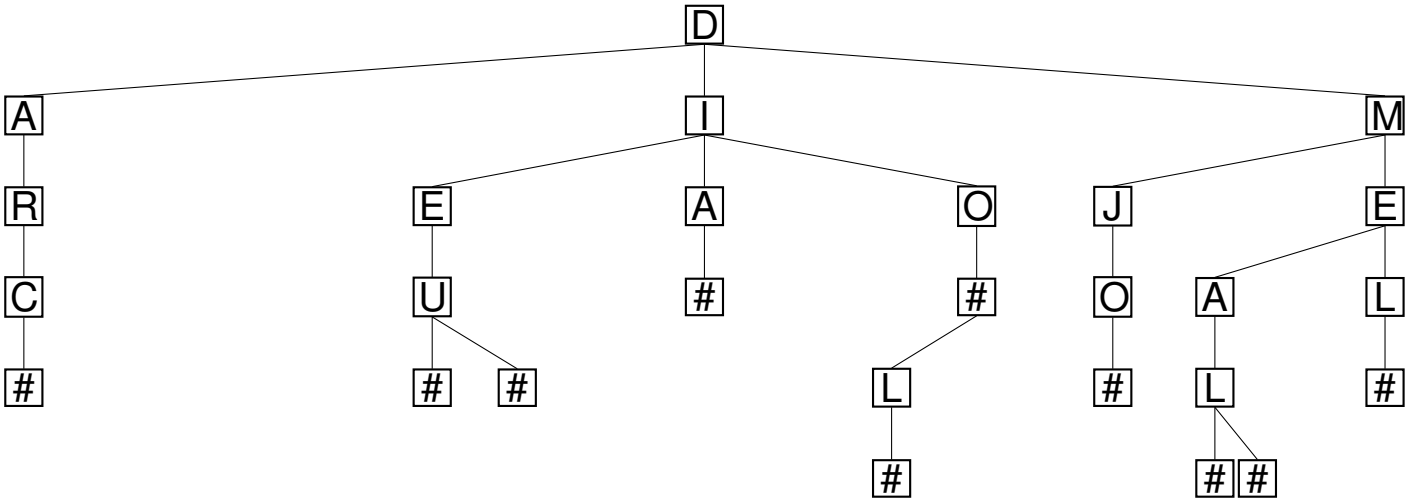


Figura 6.25: Exemple d'arbre ternari de cerca (TST).

```
template <class Simbol, class Clau, class Valor>
class diccDigital {
private:
  struct node_tst {      // tst = ternary search tree
    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);
  ...
};
```

Cost: $\Theta(k.length() \cdot log(\#simbols))$

```
template <class Simbol, class Clau, class Valor>
void diccDigital::consulta (const Clau &k, bool &hi_es, Valor
  node_tst *n = rconsulta(_arrel, 0, k);
  if (n == nullptr) {
    hi_es = false;
  }
  else {
    v = n->_v;
    hi_es = true;
  }
}
```

Mètode privat de classe. Cost: $\Theta(k.length() \cdot log(\#simbols))$

```
template <class Simbol, class Clau, class Valor>
typename diccDigital::node_tst*
diccDigital::rconsulta (node_tst *n, nat i, const Clau &k) thr
  node_tst *res = nullptr;
  if (n != nullptr) {
    if (i == k.length() and n->_c == especial<Clau>()) {
      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() retorna el símbol usat per indicar el final de la clau, per ex. si la Clau és string llavors especial() retorna '#'. Cost: $\Theta(k.length() \cdot log(\#simbols))$

```
template <class Simbol, class Clau, class Valor>
void diccDigital::insereix (const Clau &k, const Valor &v) thr
  // Afegir el sentinella al final de la clau
  Clau k2 = k + especial<Clau>();
  _arrel = rinsereix(_arrel, 0, k2, v);
}
```

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

```
template <class Simbol, class Clau, class Valor>
typename diccDigital::node*
diccDigital::rinsereix (node_tst *n, nat i, const Clau &k, con
  if (n == nullptr) {
    n = new node_tst;
    n->_esq = n->_dret = n->_cen = nullptr;
    n->_c = k[i];
    try {
      if (i < k.length()-1) {
        n->_cen = rinsereix(n->_cen, i+1, k, v);
      }
      else { // i == k.length()-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 de cues seria 10, ja que l'alfabet conté 10 símbols, els dígits del 0 al 9.

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

0. Es comença tractant el digít menys significatiu dels elements del vector (les unitats).

1. Cada element es mou a l'estructura de dades auxiliar segons el dígit que s'estigui tractant en aquest moment. En concret es mou a la cua que coincideixi amb el valor del dígit.

2. Un cop s'han tractat tots els elements a ordenar, es recorre l'estructura auxiliar començant per la primera cua i es mouen els elements al vector (mantenint l'ordenació pròpia de la 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

```
template <typename T>
void radixsort(T A[], nat u, nat v, nat r) {
/* Acció per ordenar el tros de vector A[u..v] tenint en compt
   el bit r-èssim.*/
  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-1);
```

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;
}
```

6.11.3.3 Cost

Cadascuna de les etapes de *radix sort* 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 l vegades (una vegada per cadascun dels seus l bits), 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.
- consultar l'element de prioritat mínima.
- esborrar 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 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 5: 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 la implementació 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.

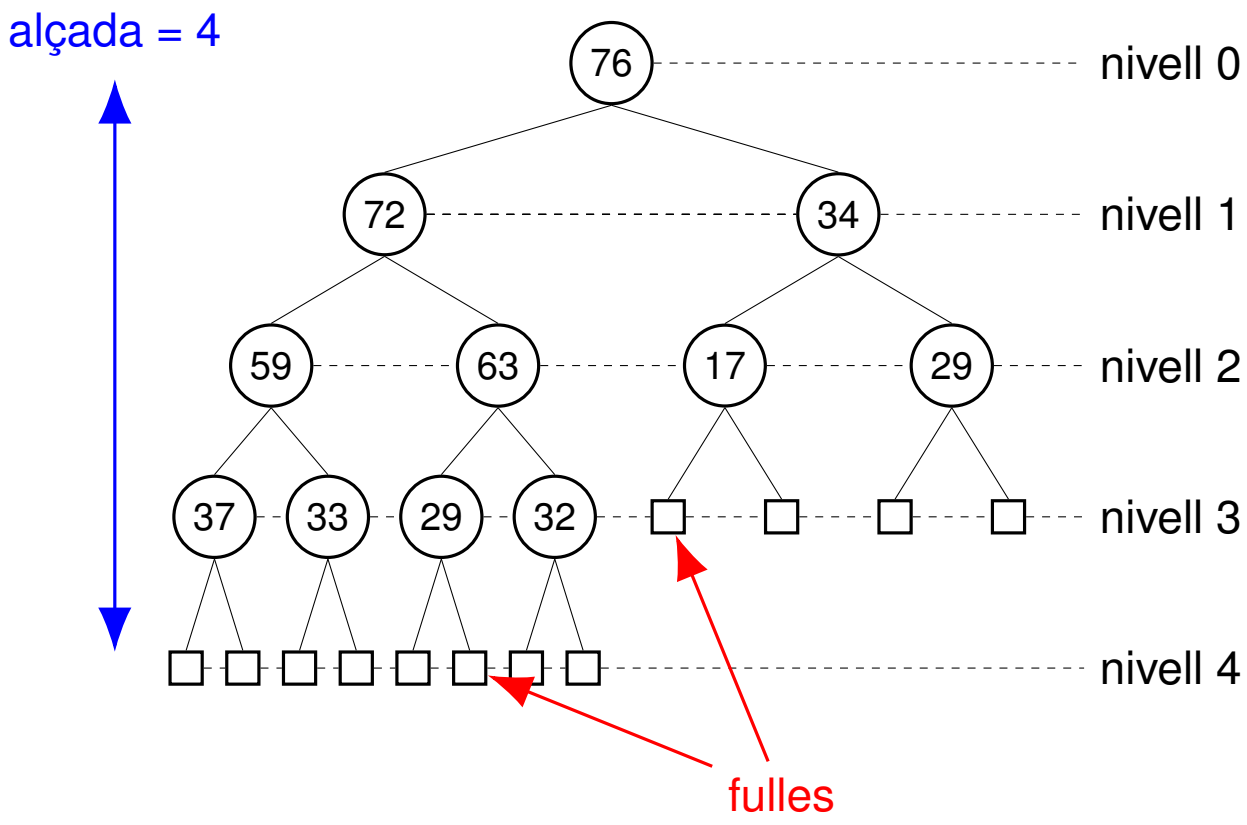


Figura 7.1: Exemple de monticle (max-cua).

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 es pugui fer cap canvi.

.

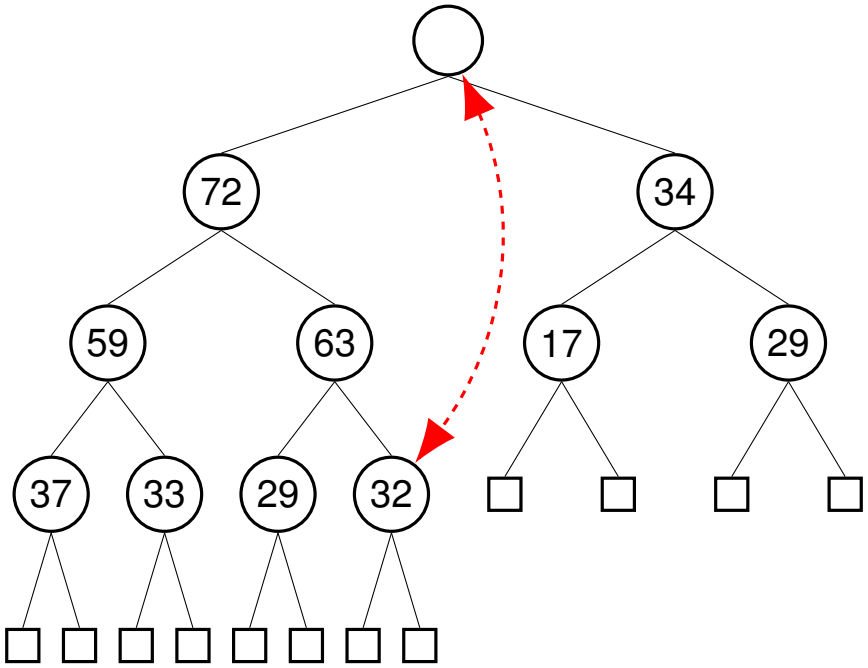


Figura 7.2: Esborrat del màxim del monticle de la figura 7.1 – substitució per l'arrel

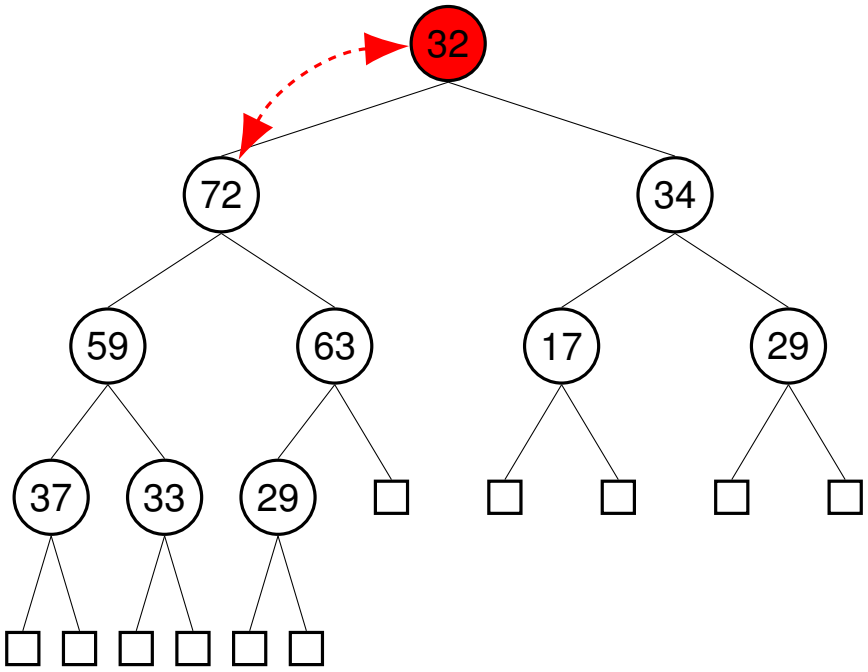


Figura 7.3: Esborrat del màxim del monticle de la figura 7.1 – enfonsat 1

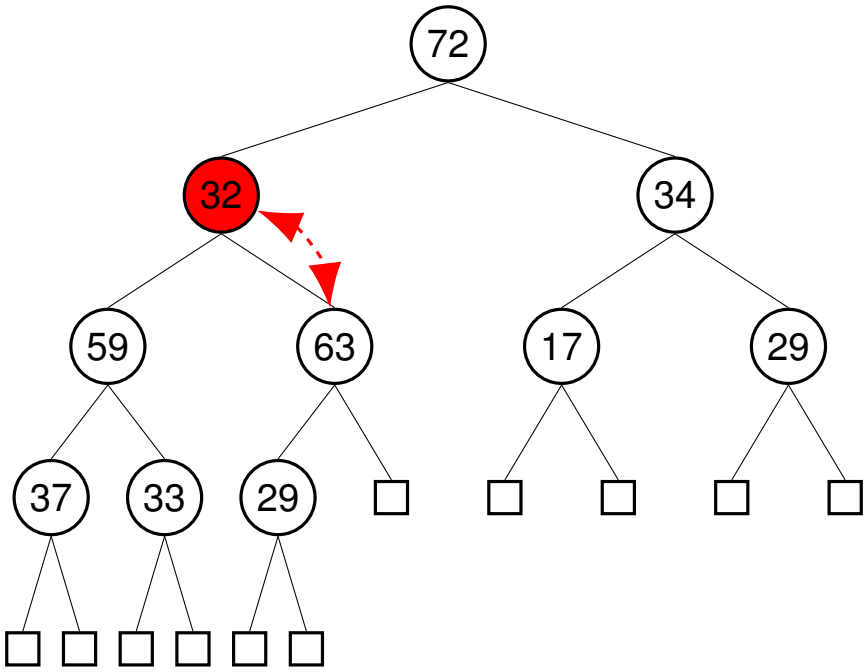


Figura 7.4: Esborrat del màxim del monticle de la figura 7.1 – enfonsat 2

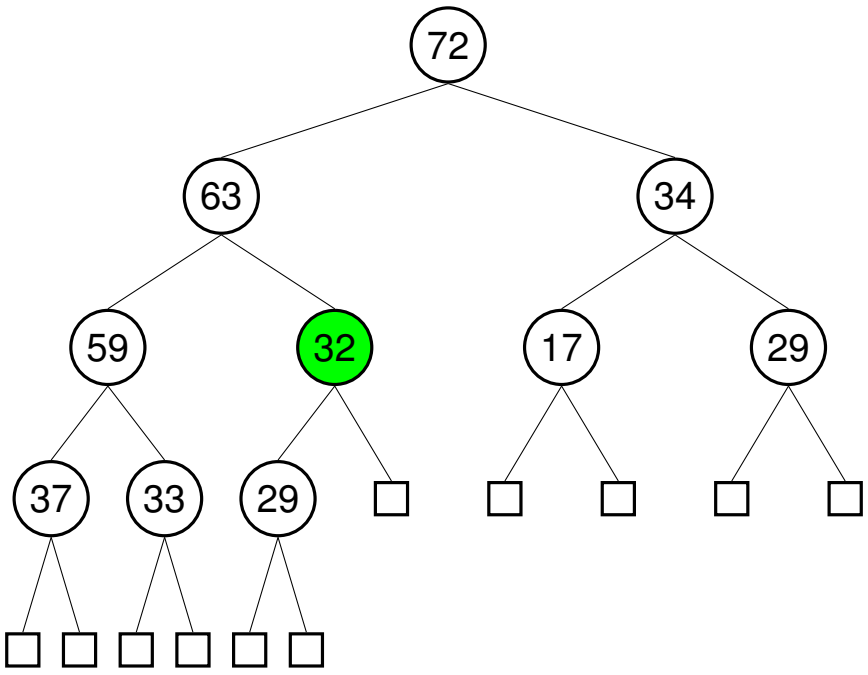


Figura 7.5: Esborrat del màxim del monticle de la figura 7.1 – resultat final

7.5.5 Implementació amb vector

Donat que l'alçada del heap és $\log(n)$, el cost de les insercions i eliminacions és logarítmic. 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, i resoldre de manera eficaç la localització de l'últim element i del pare de la primera fulla.

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

7.5.5.1 Representació

Com a representació usarem una taula estàtica. Dues millores possibles serien usar una taula dinàmica i 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();
};
```

7.5.5.2 Implementació

Només implementarem aquells mètodes més destacats de la classe. La implementació de les tres grans i la resta de mètodes és trivial.

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

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

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

```
template <typename Elem, typename Prio>
void CuaPrio<Elem, Prio>::enfonsar (nat i) {
  bool fi = false;
  while ((2*i <= _nelems) and not fi) {
    nat f = 2*i;
    if (f < _nelems and
        _taula[f].second < _taula[f+1].second) {
```

```
      ++f;
    }
    if (_taula[i].second < _taula[f].second) {
      swap(_taula[i], _taula[f]);
      i = f;
    }
    else {
      fi = true;
    }
  }
}
```

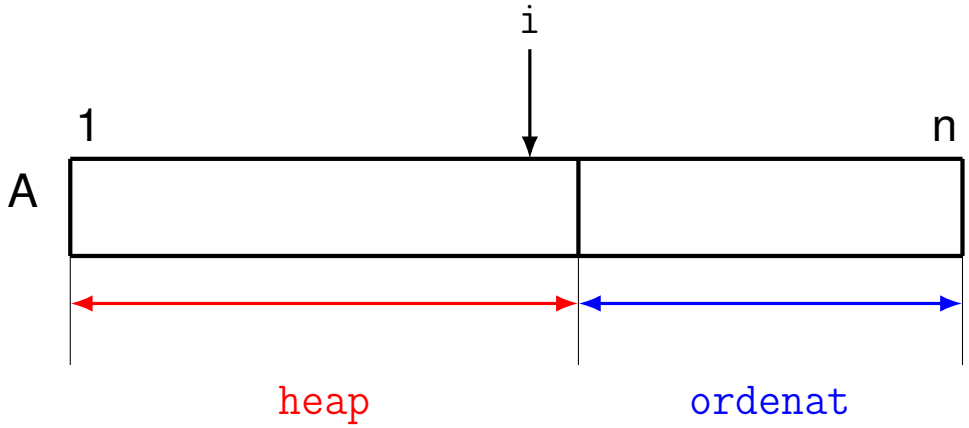
Operació privada (versió iterativa)
Fa surar el node i-èssim fins a restablir l'ordre del monticle; tots els nodes excepte el i-èssim satisfan la propietat iii.
Cost : $\Theta(\log_2(i))$

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



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

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

7.6.2 Funcionament de l'algorisme de heapsort

7.6.2.1 Creació del MAXHEAP

Veure les figures 7.6, 7.7, 7.8 i 7.9 per tenir una representació de la creació d'una max-cua.

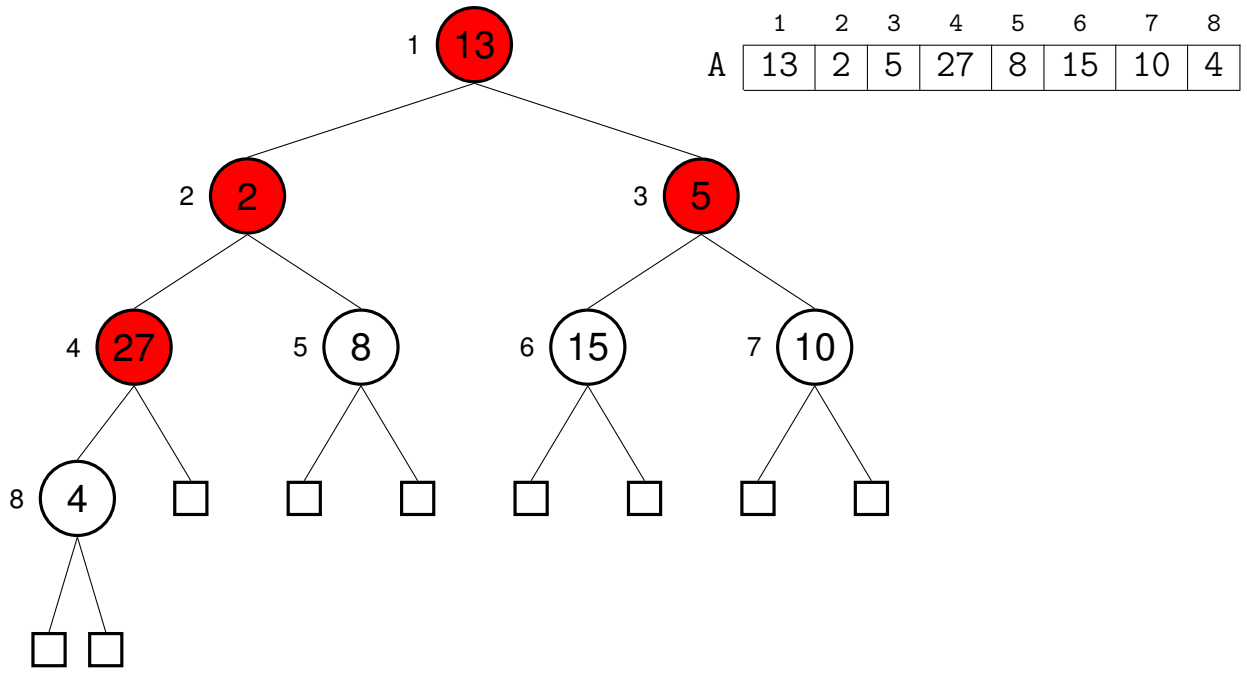


Figura 7.6: Creació del MAXHEAP. Els nodes blancs satisfan la propietat d'un monticle i els vermells no.

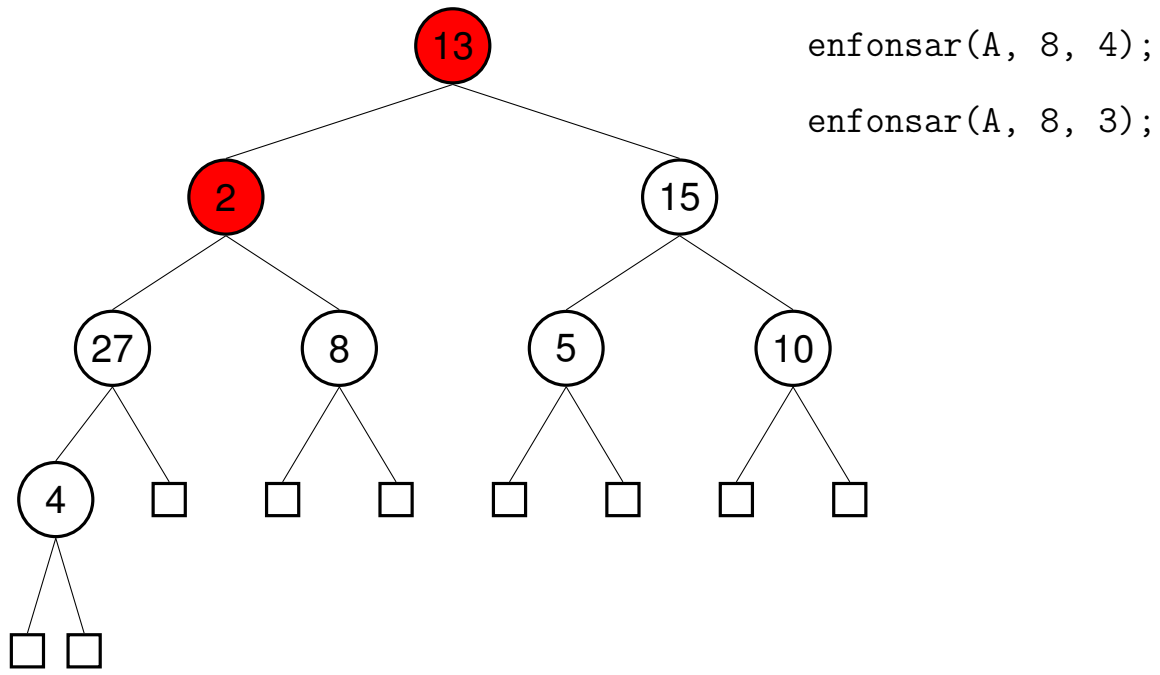


Figura 7.7: Creació del MAXHEAP. Els nodes blancs satisfan la propietat d'un monticle i els vermells no.

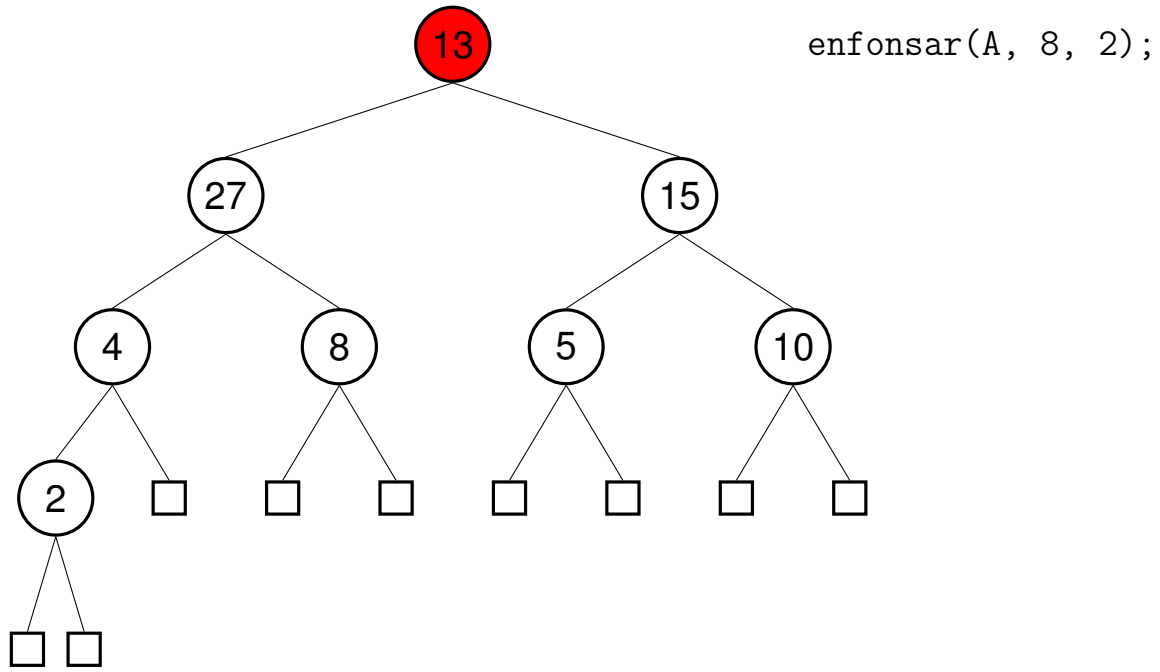


Figura 7.8: Creació del MAXHEAP. Els nodes blancs satisfan la propietat d'un monticle i els vermells no.

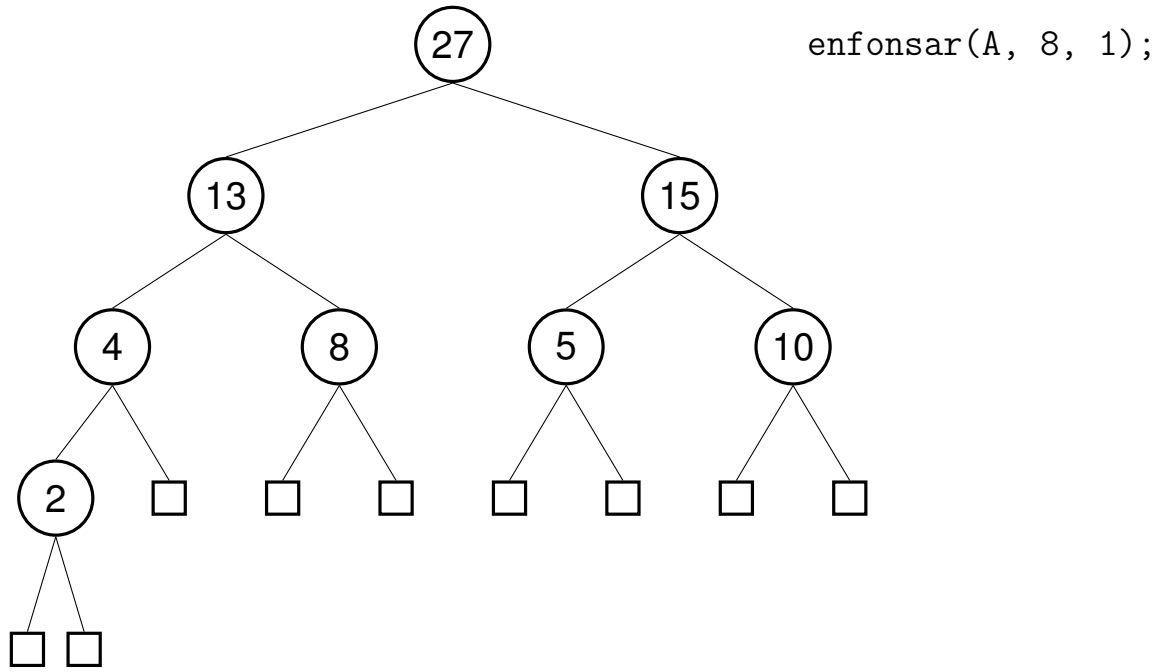
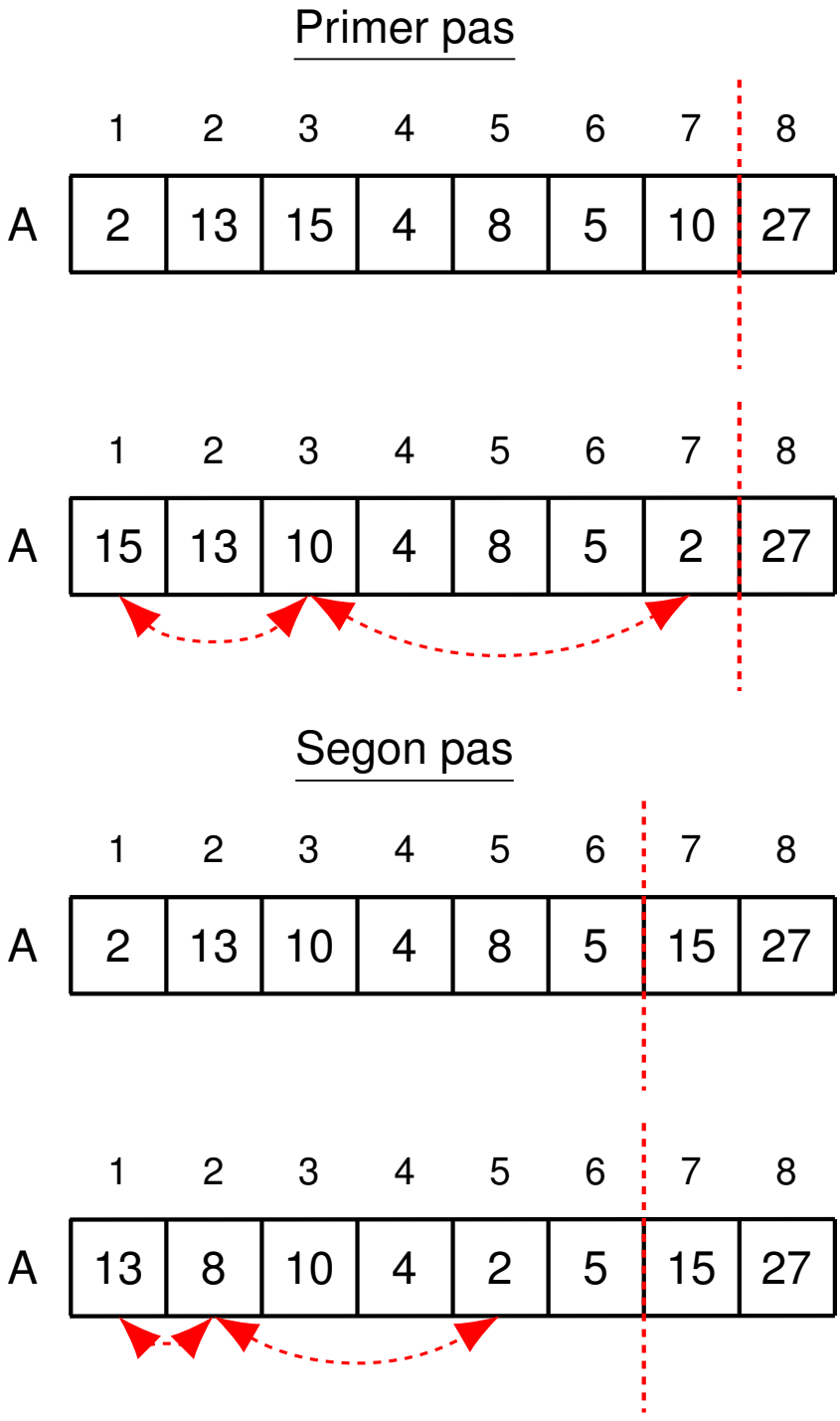


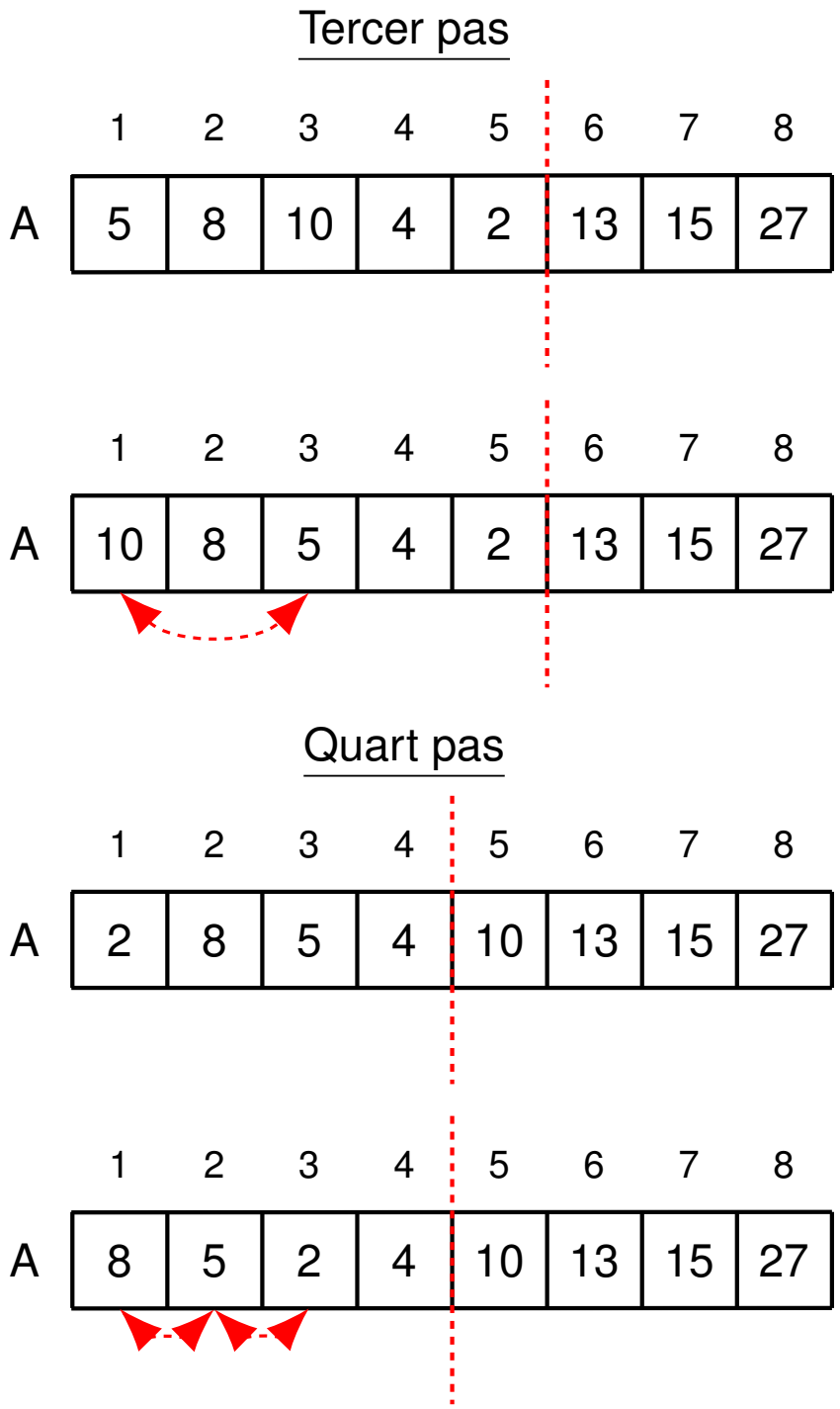
Figura 7.9: Creació del MAXHEAP, on tots els nodes del monticle compleixen les propietats d'un monticle.

7.6.2.2 Eliminació del màxim del MAXHEAP

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

Les figures 7.10, 7.11, 7.12 i 7.13 mostren els primers passos de l'algorisme de *heapsort*. Es parteix del monticle de la figura 7.9 que es crea en la primera part d'aquest algorisme d'ordenació.


Primer pas
1 2 3 4 5 6 7 8
A 2 13 15 4 8 5 10 27
1 2 3 4 5 6 7 8
A 15 13 10 4 8 5 2 27
Segon pas
1 2 3 4 5 6 7 8
A 2 13 10 4 8 5 15 27
1 2 3 4 5 6 7 8
A 13 8 10 4 2 5 15 27


Tercer pas
1 2 3 4 5 6 7 8
A 5 8 10 4 2 13 15 27
1 2 3 4 5 6 7 8
A 10 8 5 4 2 13 15 27
Quart pas
1 2 3 4 5 6 7 8
A 2 8 5 4 10 13 15 27
1 2 3 4 5 6 7 8
A 8 5 2 4 10 13 15 27

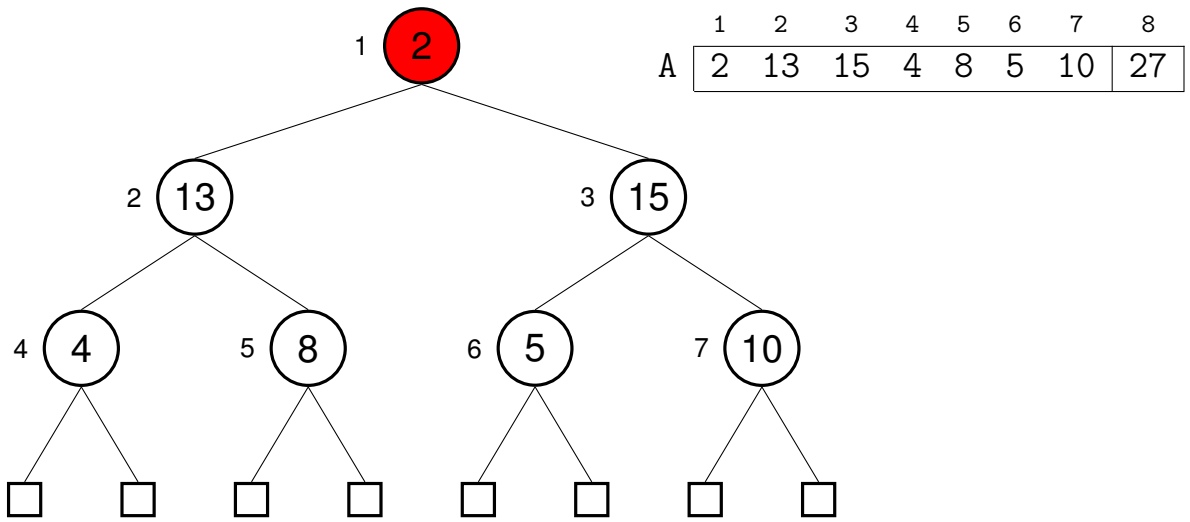


Figura 7.10: Esborrat del màxim del MAXHEAP. Els nodes blancs satisfan la propietat d'un monticle i els vermells no.

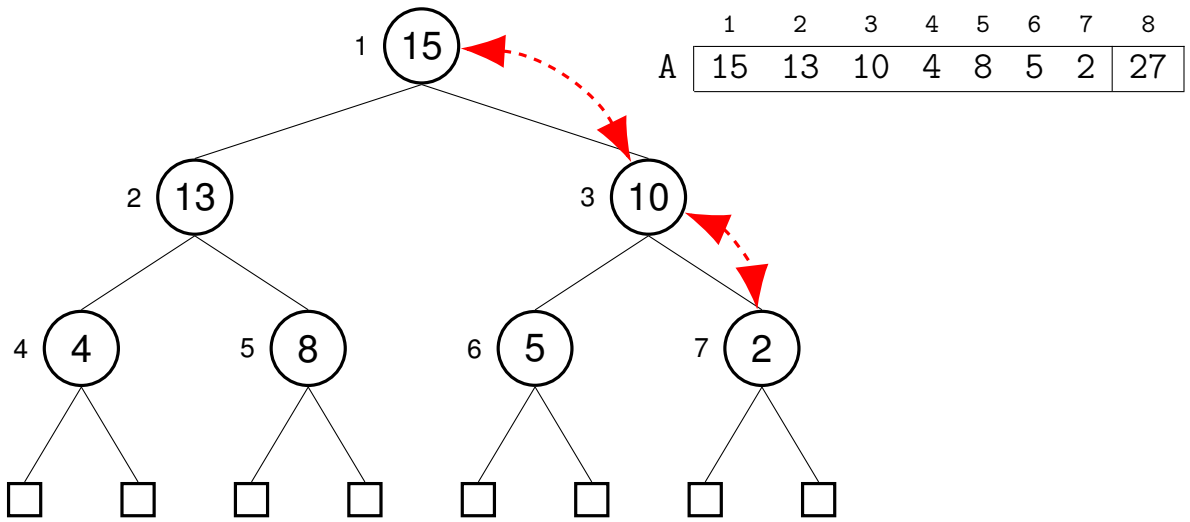


Figura 7.11: Esborrat del màxim del MAXHEAP. Els nodes blancs satisfan la propietat d'un monticle i els vermells no.

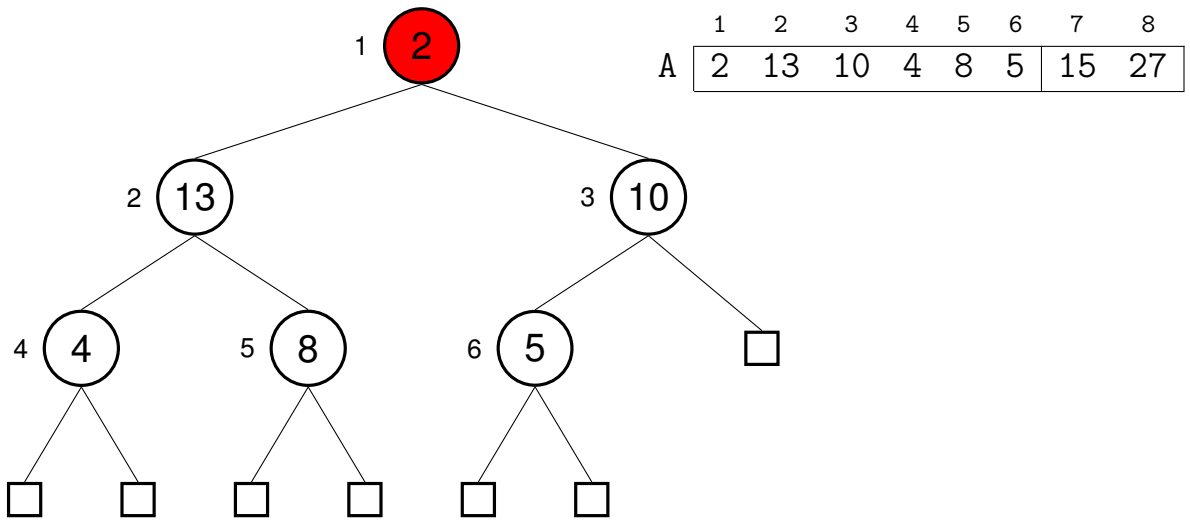


Figura 7.12: Esborrat del màxim del MAXHEAP. Els nodes blancs satisfan la propietat d'un monticle i els vermells no.

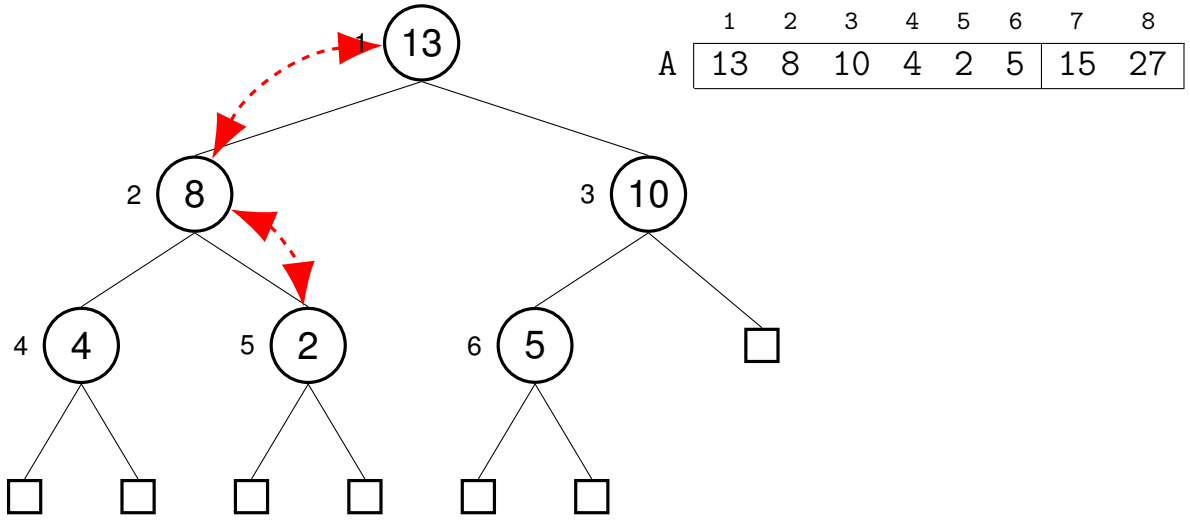


Figura 7.13: Esborrat del màxim del MAXHEAP. Els nodes blancs satisfan la propietat d'un monticle i els vermells no.

7.6.3 Implementació

7.6.3.1 Ordenació 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 Ordenació sense usar la classe CuaPrio

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

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
Particions

Hi ha tres grups de persones: els que fan que les coses passin; els que miren les coses que passen i els que es pregunten què va passar.

Nicholas Murray Butler (1862-1947), Premi Nobel del la Pau de 1931

8.1 INTRODUCCIÓ

Definició 6: Partició

Una **partició** P d'un conjunt d'elements no buit A és una col·lecció de subconjunts no buits $P = \{A_1, A_2, \ldots, A_n\}$ tal que

1. $A_i \cap A_j = \emptyset$ si i no és igual a j
2. $A_1 \cup A_2 \cup \cdots \cup A_n = A$

A cadascun dels subconjunts A_i de la partició P s'anomena **bloc**.

Les particions i les **relacions d'equivalència** (representades amb el símbol $\equiv$) estan molt lligades. $\equiv$ és una relació d'equivalència sobre els elements de A si i només si

1. $\equiv$ és reflexiva: $a \equiv a$ per tot $a \in A$.
2. $\equiv$ és simètrica: $a \equiv b$ si i només si $b \equiv a$ per qualsevol $a, b \in A$.
3. $\equiv$ és transitiva: Si $a \equiv b$ i $b \equiv c$, llavors $a \equiv c$ per qualsevol $a, b, c \in A$.

Donada una partició P del conjunt A, aquesta indueix una relació d'equivalència $\equiv$ definida per

$a \equiv b$ si i només si a i b pertanyen a un mateix bloc A_i de P.

I a la inversa, donada una relació d'equivalència $\equiv$ en A, aquesta indueix una partició P on cada bloc A_i conté tots els elements equivalents a un cert element $a \in A$.

Moltes vegades es distingeix un element qualsevol de cada bloc. Aquest element s'anomena el **representant** del bloc.

Hi ha molts algorismes, sobretot els algorismes de grafs, pels quals és molt important representar particions d'un conjunt finit d'elements de forma eficient.

Com en els diccionaris, podem distingir entre particions:

- **Estàtiques**: No es permeten insercions o eliminacions d'elements.
- **Semidinàmiques**: Es permet la inserció de nous elements però no les eliminacions.
- **Dinàmiques**: Es permet la inserció i eliminació d'elements.

Per facilitar l'estudi de les particions treballarem amb particions estàtiques definides sobre el conjunt d'elements {0, 1, 2, ..., n-1}. Si tenim particions dinàmiques (o semidinàmiques) o el conjunt d'elements és diferent, normalment s'aplica alguna de les tècniques de diccionaris vistes anteriorment de manera que a cada element del conjunt se li associa un enter entre 0 i n-1.

Les operacions bàsiques d'una PARTICIÓ són

1. Donat dos elements i i j, determinar si pertanyen al mateix bloc.
2. Donat dos elements i i j, fusionar els blocs als que pertanyen (si són diferents) en un sol bloc, retornant la partició resultant.

Com que normalment distingim un element representant de cada bloc de la partició, és habitual disposar d'una operació

$find$ que, donat un element i, retorna el representant del bloc al que pertany i.

Si dos elements i i j tenen el mateix representant vol dir que pertanyen al mateix bloc.

A les particions se les anomena **union-find sets** o **MFSets** (Merge-Find Sets) degut a que les seves operacions bàsiques són

1. **find**: Per conèixer el representant d'un element.
2. **merge** o **union**: Per fusionar dos blocs.

Normalment, abans de fer una union dels blocs que contenen dos elements i, j, s'ha comprovar si i, j ja estan o no en el mateix bloc. Per tant mirarem si `find(i) == find(j)`. Per aquest motiu és habitual que l'operació $union$ només actuï sobre els elements representants de cada bloc.

```
...
u = find(i);
v = find(j);
if (u != v) {
  union(u, v);
  ...
}
...
```

8.2 IMPLEMENTACIÓ

Si treballem amb particions estàtiques definides sobre el conjunt d'elements {0, 1, 2, ..., n-1} podem utilitzar una implementació sobre vector. L'operació constructora del $mfset$ pot demanar en temps d'execució que el vector tingui n elements.

Si tenim particions dinàmiques (o semidinàmiques) o el conjunt d'elements és diferent, caldrà adaptar alguna de les tècniques de diccionaris vistes anteriorment.

Hi ha dues estratègies diferents per implementar una partició:

- **Quick-find**: L'operació $find$ serà molt ràpida a costa de fer més costosa l'operació $union$.
- **Quick-union**: Es prioritza que l'operació $union$ sigui ràpida alentint l'operació $find$. Utilitzant estratègies addicionals podem millorar el cost de $find$ convertint a aquesta alternativa en la millor.

8.2.1 Quick-find

La idea de **quick-find** és guardar per a cada element quin és el seu representant. Si s'utilitza un vector P per representar la partició, la component i del vector P contindrà el representant del bloc al que pertany i.

- $find$ tindrà un cost constant, ja que només caldrà examinar la component `P[i]`.

- $union$ tindrà cost $\Theta(n)$, ja que si unim els blocs que contenen els elements i i j, caldrà canviar tots els elements del vector P que tenen com a representant $find(i)$ pel nou representant $find(j)$ (o a la inversa).

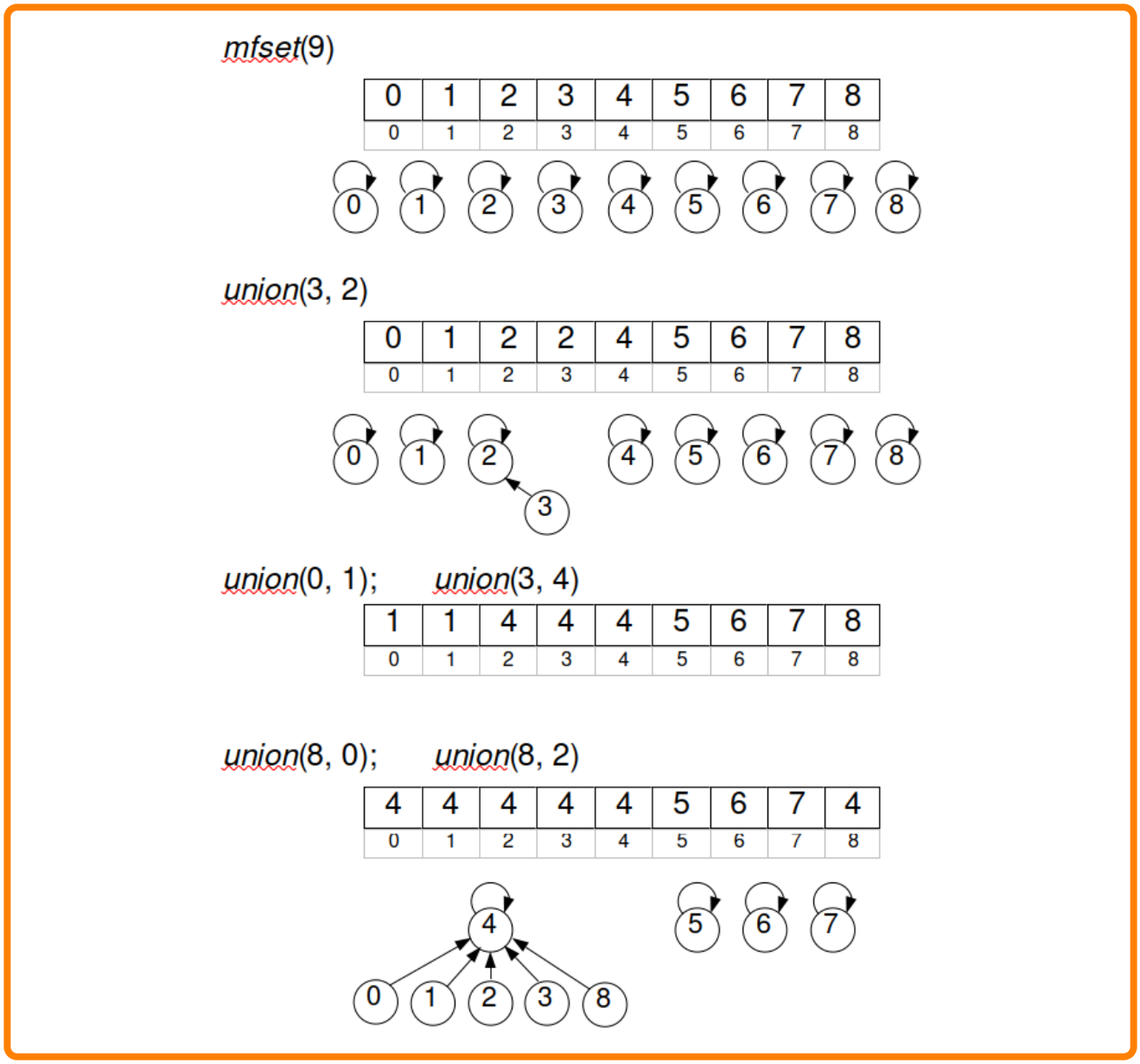


Figura 8.1: MFset amb l'estratègia **quick-find**.

Es pot veure la representació **quick-find** com un bosc d'arbres. Cada bloc de la partició correspon a un arbre. Els ele-

ments d'aquests arbres apunten al pare. L'arrel de l'arbre és el representant del bloc que s'apunta a si mateix.

Tots els arbres tenen alçada 1 (si només contenen un element) o alçada 2 (tots els elements del bloc excepte el representant estan en el nivell 2 i apunten a l'arrel). Per això l'operació $find$ és molt ràpida.

8.2.2 Quick-union

Per tal de poder fer les unions més ràpid evitant recórrer tot el vector, l'estratègia **quick-union** simplement posa el representant d'un bloc com a fill del representant de l'altre bloc.

```
class mfset {
 private:
  nat _num;// Nombre d'elements de la partició
  nat* _P; // Vector de n naturals (n es fixa en temps d'execució)

 public:
  mfset(nat n);
  void union(nat i, nat j);
  nat find(nat i);
};

mfset::mfset(nat n) {
  nat i;
  _P = new nat[n]; // Demanem n elements de tipus nat
  _num = n;
  for (i=0; i < n; i++) {
    _P[i] = i;    // Cada element és el seu propi representant
  }
}

void mfset::union(nat i, nat j) {
  nat u = find(i);
  nat v = find(j);
  _P[u] = v;
```

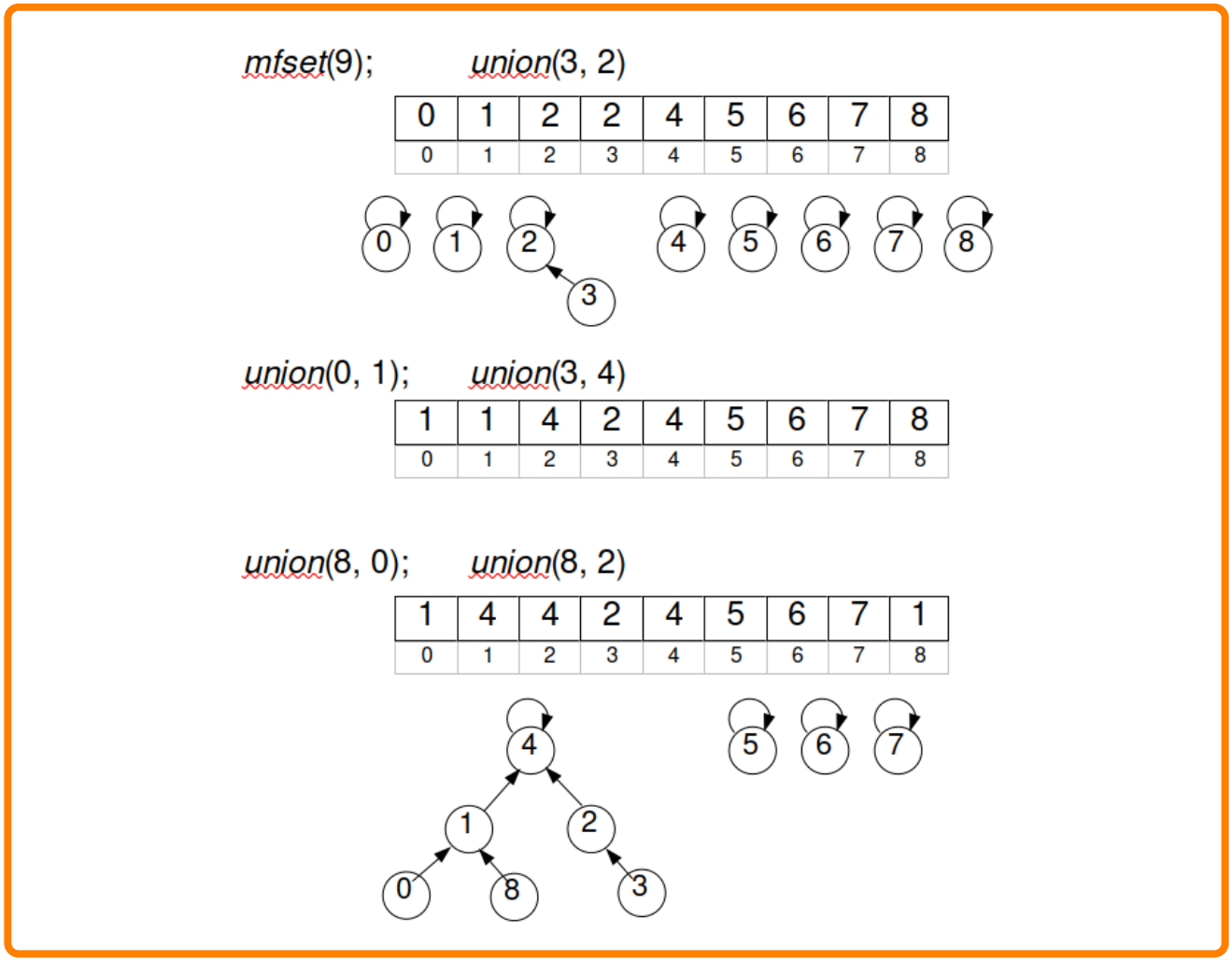


Figura 8.2: MFset amb l'estratègia **quick-union**.

```
}

nat mfset::find(nat i) {
  if (i < 0 or i > _num-1)
    throw error(ElementEquivocat);
  while (_P[i] != i) {
    i = _P[i];
  }
  return i;
}
```

Generalment els arbres resultants d'una seqüència d'unions seran relativament equilibrats i el cost de les operacions serà petit. En el cas pitjor podem crear arbres poc equilibrats i, per tant, l'operació $find(i)$ (i indirectament l'operació $union$) tingui un cost proporcional al nombre d'elements que hi ha des de i fins a l'arrel.

Si, per exemple, fem la seqüència d'operacions:

```
mfset(5);  union(1, 2);  union(2, 3);
union(3, 4);  union(4, 5);
```

obtindrem un arbre completament desequilibrat doncs s'ha convertit en una llista.

8.2.3 Tècniques per millorar quick-union

1) **Unió per pes**: L'arbre amb menys elements és el que es posa com a fill del que té més elements.

Cal guardar informació auxiliar sobre la grandària de cada arbre. Ho podem guardar junt amb l'arrel. De fet la informació que conté l'arrel és inútil ja que simplement s'apunta a si mateix per indicar que és l'element representant. Només necessitem un booleà (un bit) per indicar si és l'arrel o no. Si utilitzem un

vector d'enters podem fer servir el conveni de que si $P[i] < 0$ llavors i és una arrel i $-P[i]$ és la grandària de l'arbre.

```
void mfset::union(nat i, nat j) {
  int u = find(i);
  int v = find(j);
  if (u != v) {
    if (-_P[u] > -_P[v]) swap(u, v);
    _P[v] = _P[v] + _P[u];
    _P[u] = v;
  }
}

int mfset::find(nat i) {
  if (i < 0 or i > _num-1) throw error(ElementEquivocat);
  while (_P[i] >= 0) {
    i = _P[i];
  }
  return i;
}
```

2) **Unió per rang**: L'arbre amb menor alçada és el que es posa com a fill del de major alçada.

Podem utilitzar la mateixa tècnica anterior per guardar l'alçada d'un arbre dins de la seva arrel: Si $P[i] < 0$ llavors i és una arrel i $-P[i]$ és l'alçada de l'arbre. L'operació $find$ és idèntica a l'anterior.

```
void mfset::union(nat i, nat j) {
  int u = find(i);
  int v = find(j);
  if (u != v) {
    if (-_P[u] > -_P[v]) swap(u, v);
    _P[v] = -max(-_P[v], -_P[u]+1);
    _P[u] = v;
  }
}
```

Es pot demostrar que, en la unió per pes i en la unió per rang, l'alçada màxima d'un arbre no pot ser superior a $log(n)$, sent n el

nombre d'elements de la partició. Per tant les operacions $find$ i $union$ tenen un cost $O(log(n))$ ($log(n)$ és una cota superior del cost).

3) **Compressió de camins**

Aquest mètode heurístic tracta de reduir la distància a l'arrel dels elements que trobem en el camí des de i fins a l'arrel durant una operació $find(m, i)$. Així que anem pujant des de i fins a l'arrel disminuïm l'alçada de l'arbre. Posteriors operacions $find$ que s'apliquin sobre i o algun dels elements superiors seran més ràpids.

La compressió de camins va escurçant els camins de forma que, a poc a poc, els arbres adopten la forma que tindrien amb **quick-find**, però evitant que ho faci l'operació $union$ i evitant de compactar tot un arbre d'un sol cop.

S'ha demostrat que una seqüència de m $unions$ i n $finds$ té un cost $O(m + n)$. Encara que el cost d'una única crida a l'operació $union$ o $find$ no és constant, el cost amortitzat si que ho és.

Compressió per meitats: Es modifica l'apuntador de cada element des de i fins a l'arrel per que, enlloc d'apuntar al pare, passi a apuntar a l'avi (l'arrel i el fill de l'arrel no canvien doncs no tenen avi).

```
nat mfset::find(nat i) {
  nat j;
  if (i < 0 or i > _num-1) throw error(ElementEquivocat);
  j = _P[i];
  while (_P[j] != j) {
    _P[i] = _P[j];
    i = j;
    j = _P[j];
  }
  return j;
}
```

Compressió total: Es modifica l'apuntador de cada element des de i fins a l'arrel per que apunti a l'arrel.

```
nat mfset::find(nat i) {
  nat j, k, arrel;
  if (i < 0 or i > _num-1) throw error(ElementEquivocat);
  j = i;
  while (_P[j] != j) {
    j = _P[j];
  }
  arrel = j;
  j = i;
  while (_P[j] != j) {
    k = _P[j];
    _P[j] = arrel;
    j = k;
  }
  return j;
}
```

9
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)

9.1 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 9.1 diferents exemples de grafs.

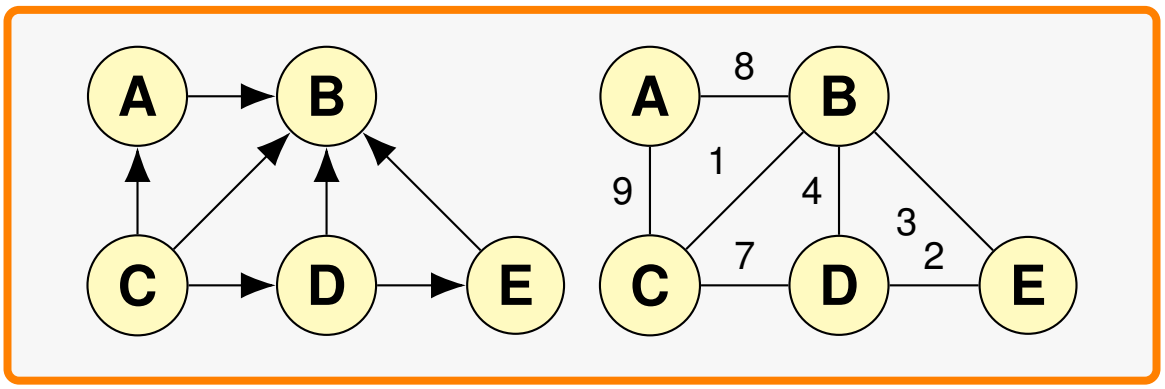


Figura 9.1: Exemples de grafs, *esquerra*: graf dirigit i no etiquetat; *dreta*: graf no dirigit i etiquetat.

9.2 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.

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

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

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

9.2.1 Adjacències

9.2.1.1 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, $grau(v) = |adj(v)|$

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

9.2.2 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.

9.2.3 Connectivitat

9.2.3.1 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 9.2 per tenir alguns exemples gràfics de graf connex, bosc i arbre no dirigit.

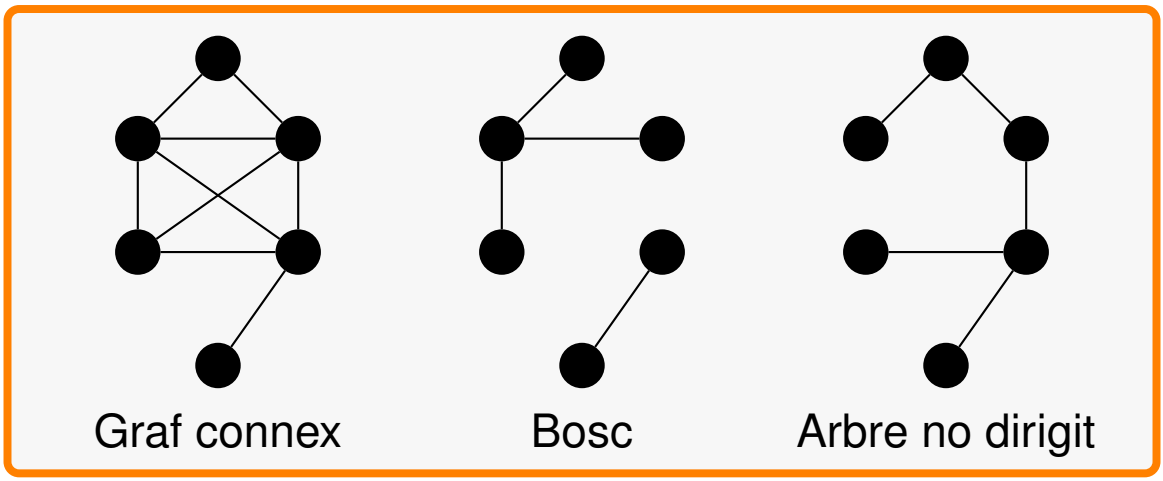


Figura 9.2: Exemple de graf connex, bosc i arbre no dirigit d'un graf no dirigit.

9.2.3.2 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 9.3 per tenir alguns exemples gràfics de subgraf, subgraf induït i arbre lliure.

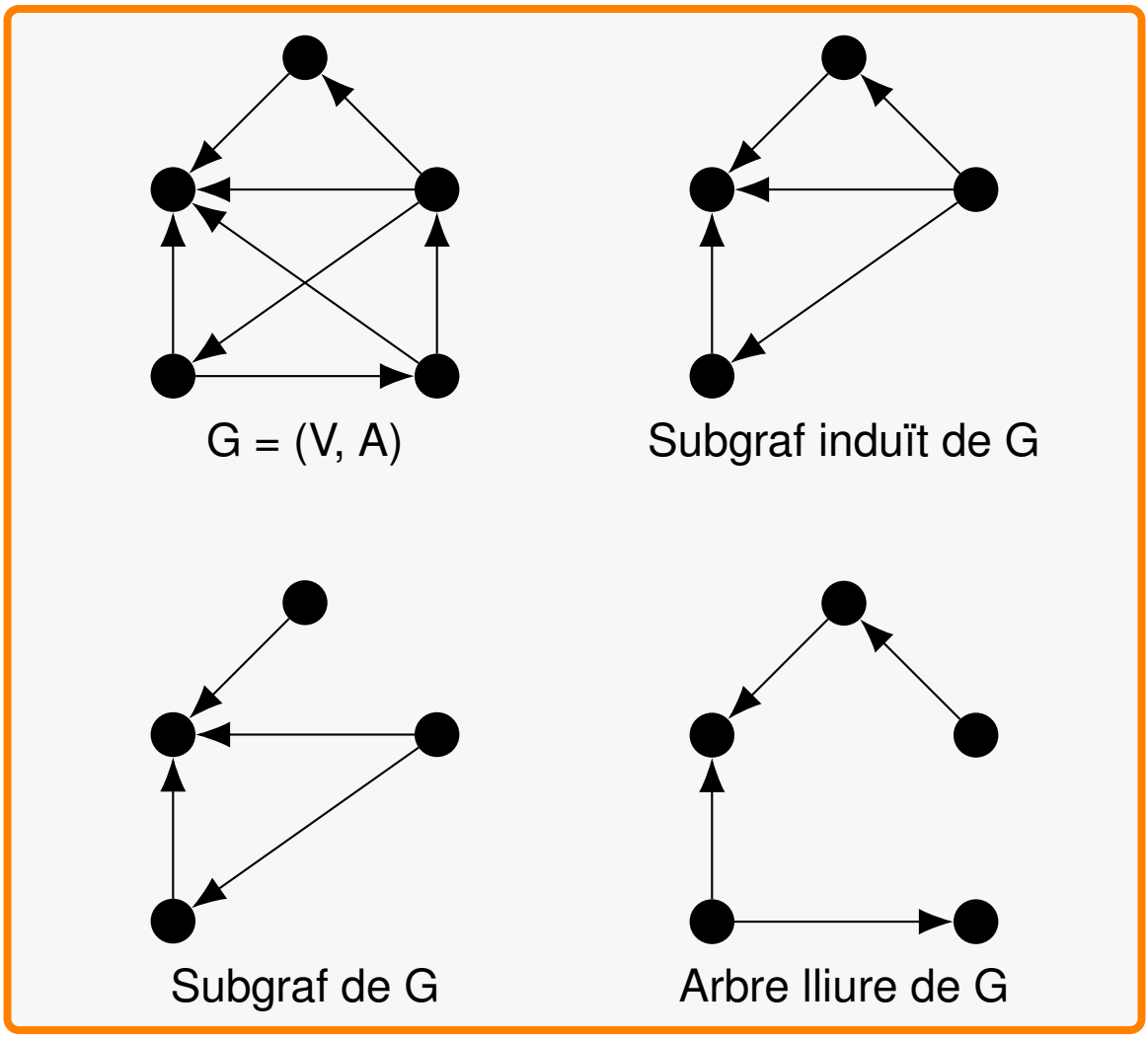


Figura 9.3: Exemples de subgraf, subgraf induït i arbre lliure d'un graf dirigit.

9.2.4 Alguns grafs particulars

9.2.4.1 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 $|A| = n \cdot \frac{n-1}{2}$.

9.2.4.2 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 9.1. *Un graf no dirigit i connex és eulerià si i només si el grau de tot vèrtex es parell.*

Lemma 9.2. *Un graf dirigit i fortament connex és eulerià si i només si el grau de tot vèrtex és zero.*

Exemples: Veure la figura 9.4 per tenir alguns exemples gràfics de graf connex, bosc i arbre no dirigit.

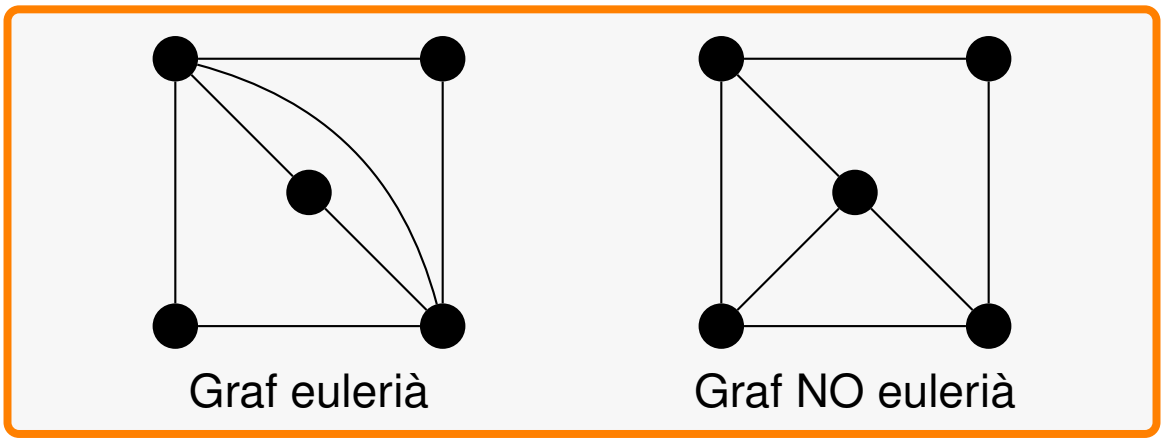


Figura 9.4: Exemples de graf eularià i no eularià.

9.2.4.3 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 9.5 per tenir alguns exemples gràfics de grafs hamiltonians i no hamiltonians.

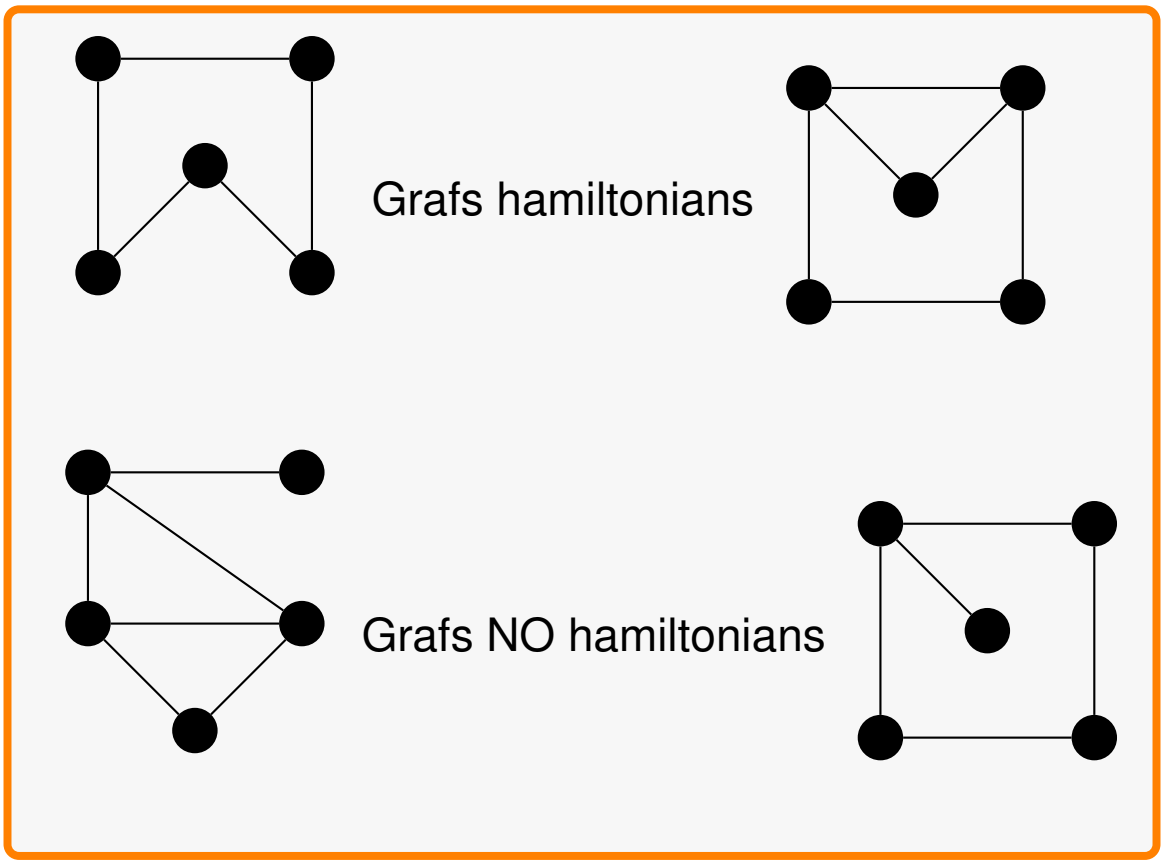


Figura 9.5: Exemples de grafs hamiltonians i no hamiltonians.

9.3 Especificació de la classe graf

9.3.1 Definició de la classe

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();
};
```

9.4 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.

9.4.1 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 9.7 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 9.6.

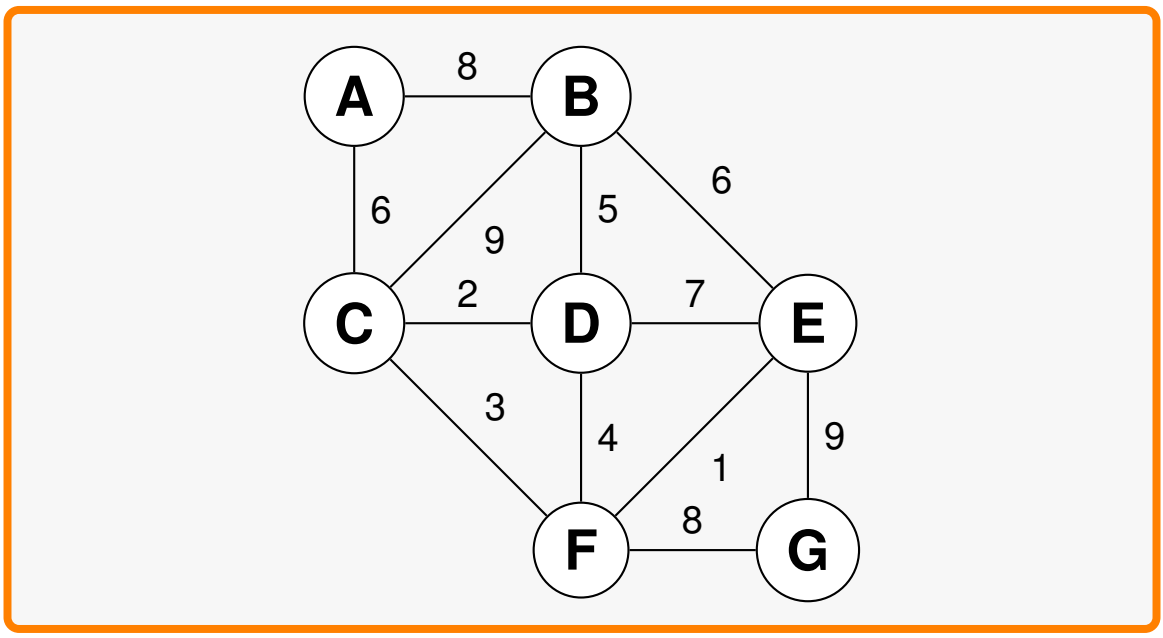


Figura 9.6: Graf no dirigit i etiquetat usat com a exemple de la representació amb una matriu d'adjacències.

		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 9.7: Representació amb una matriu d'adjacències del graf de la figura 9.6 (només es es mostren les caselles que s'utilitzaran).

9.4.1.1 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.

9.4.2 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 9.9 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 9.8.

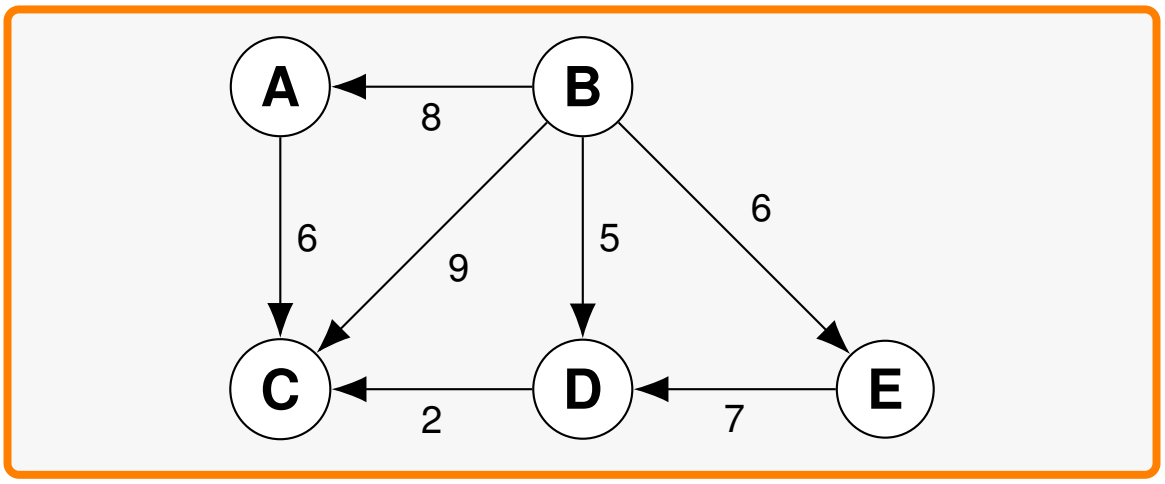


Figura 9.8: Graf dirigit i etiquetat usat com a exemple de la representació amb una llista d'adjacències.

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.

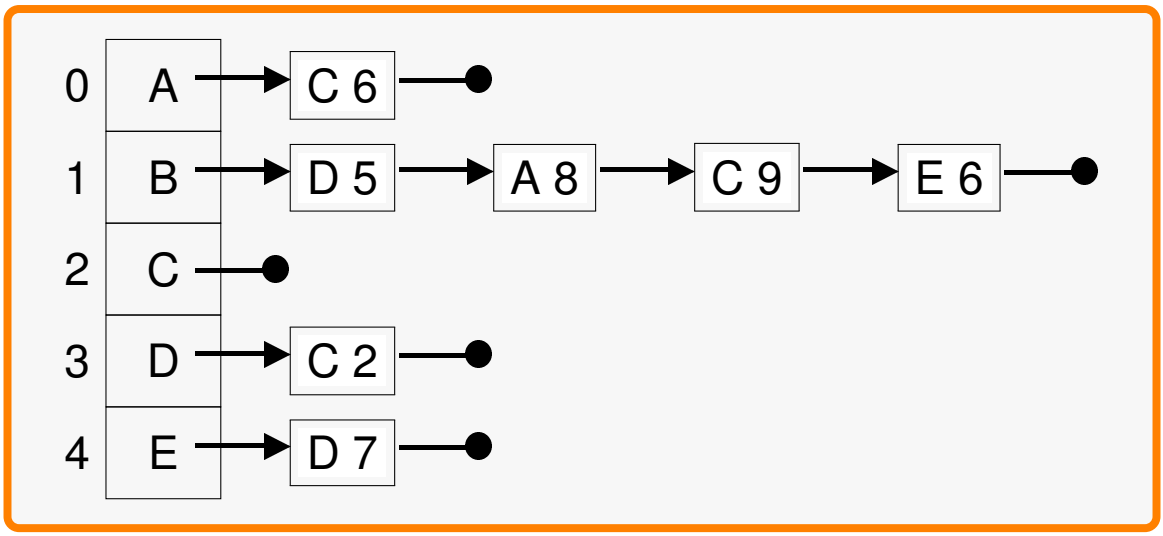


Figura 9.9: Representació amb una llista d'adjacència del graf de la figura 9.8.

9.4.2.1 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 successores 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 predecessores, 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 $\Theta(n+a)$.

9.4.3 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 9.10 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 9.8.

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.

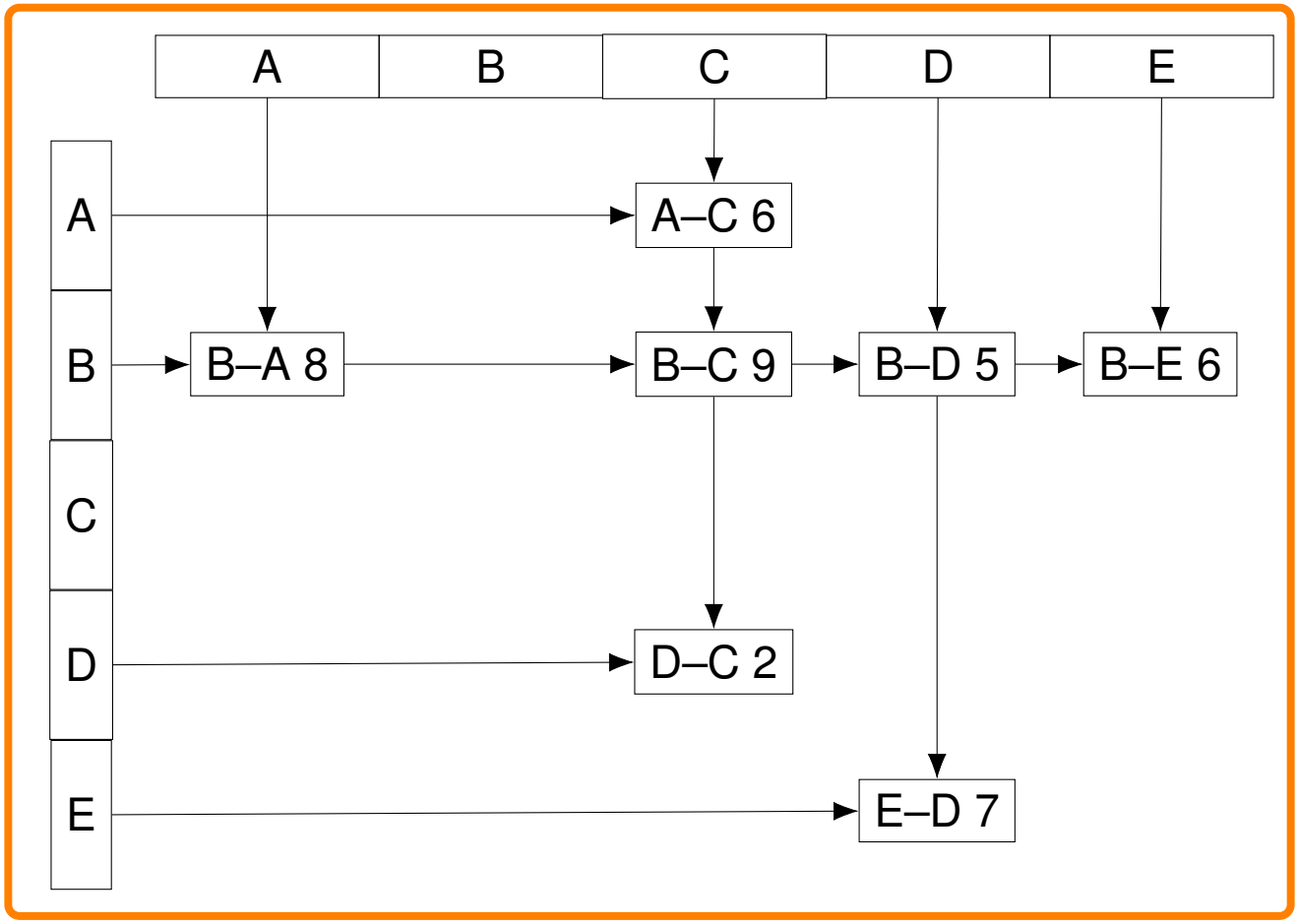


Figura 9.10: Representació amb una multillista d'adjacències del graf de la figura 9.8.

9.5 Recorreguts sobre grafs

9.5.1 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 9.11 per tenir un exemple de recorregut en profunditat prioritària i recorregut en profunditat cap enrere.

9.5.1.1 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.

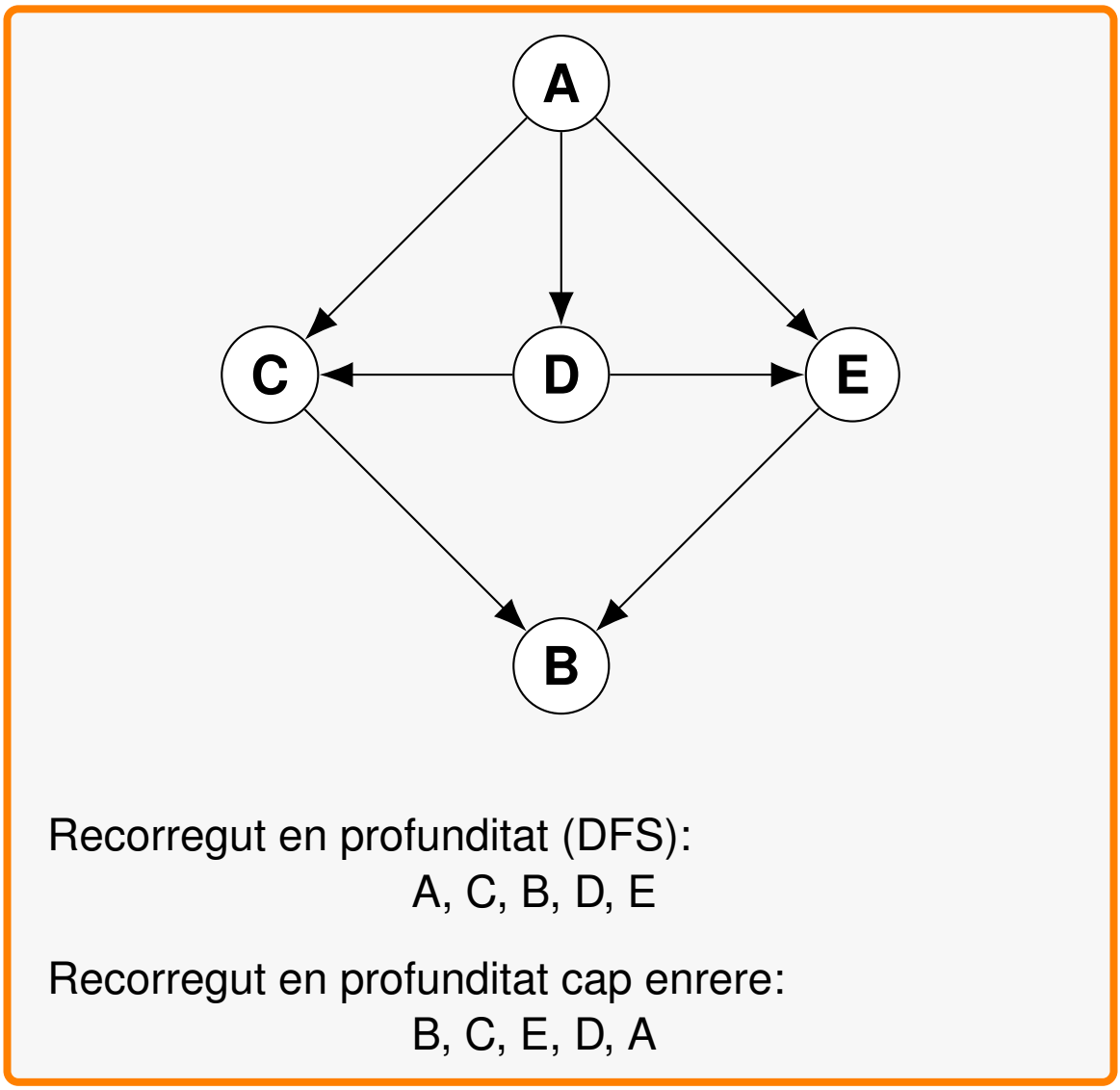


Figura 9.11: Exemple de recorregut en profunditat i en profunditat cap enrere partint del vèrtex A.

9.5.1.2 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.

```
}
```

9.5.1.3 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.

9.5.2 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.

9.5.2.1 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.

9.5.2.2 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;
      }
    }
  }
}
```

9.5.2.3 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: $\Theta(n + a)$

9.5.3 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 9.12 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).

9.5.3.1 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

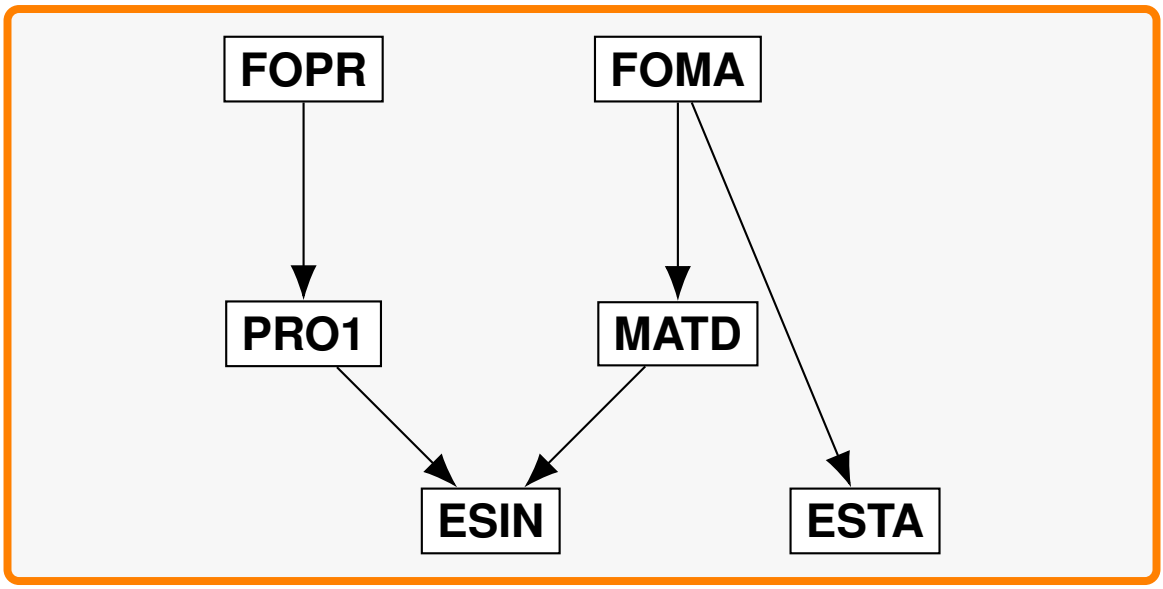


Figura 9.12: Graf que conté els prerrequisits entre assignatures del pla d'estudis d'informàtica.

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.

9.5.3.2 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;
}
```

9.5.3.3 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: $\Theta(n + a)$

9.6 Connectivitat i ciclicitat

9.6.1 Connectivitat

9.6.1.1 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.

9.6.1.2 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.

9.6.1.3 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.

```
}
```

9.6.1.4 Cost de l'algorisme

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

9.6.2 Test de ciclicitat

9.6.2.1 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.

9.6.2.2 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.

9.6.2.3 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.

```
}
```

9.6.2.4 Cost de l'algorisme

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

9.7 Arbres d'expansió mínima

9.7.1 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*.

9.7.2 Algorisme de Kruskal

9.7.2.1 Descripció de l'algorisme

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

1. Es parteix d'un subgraf de G, T=(V,∅).
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 fa créixer fins que 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 9.13.

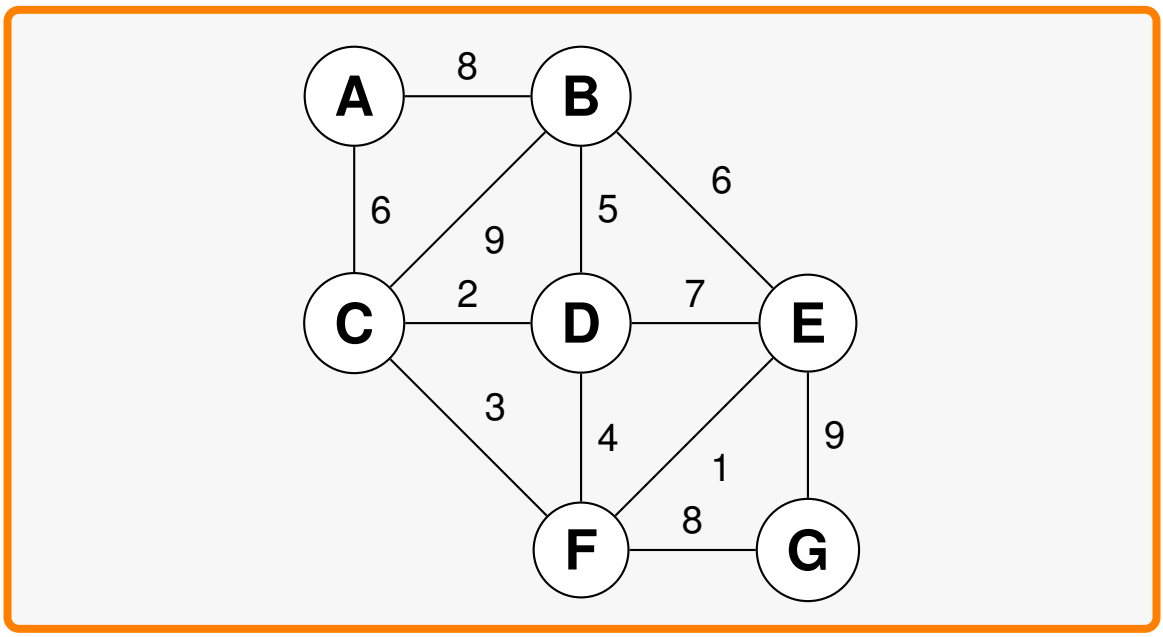


Figura 9.13: Graf no dirigit, etiquetat i connex.

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 9.1 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.

L'arbre d'expansió mínima resultant es pot veure a la figura 9.14.

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 9.1: Passos en aplicar l'algorisme de Kruskal en el graf de la figura 9.13.

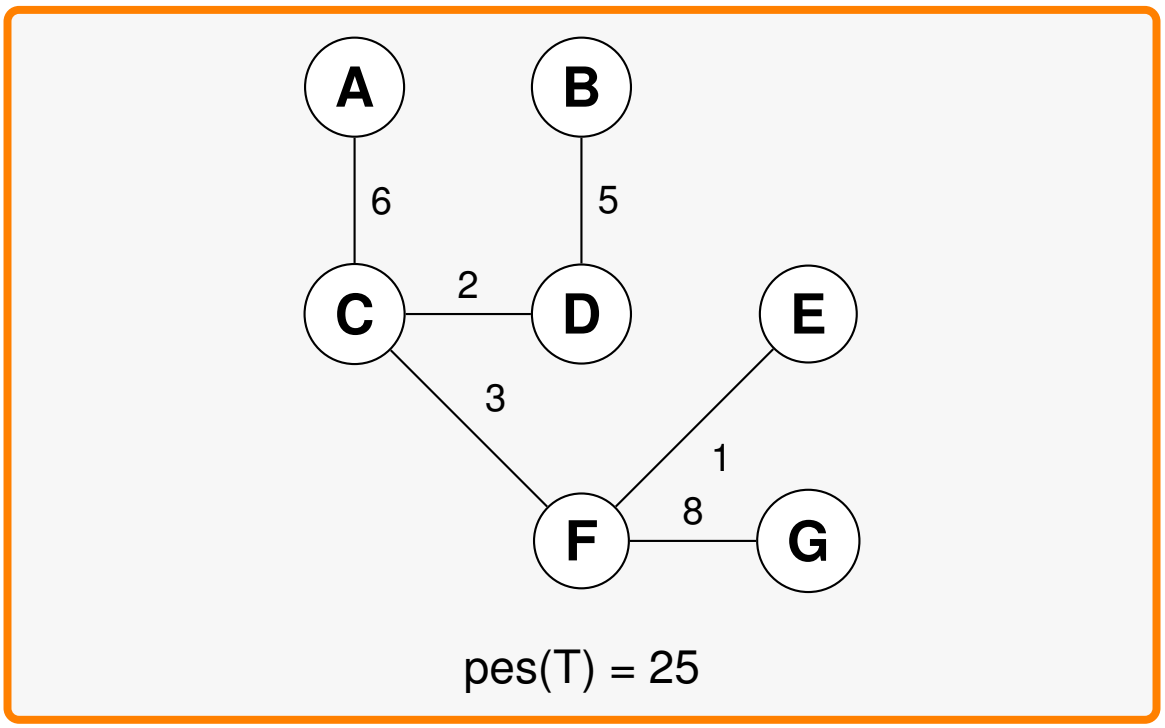


Figura 9.14: Arbre d'expansió mínima resultat d'aplicar l'algorisme de Kruskal sobre el graf 9.13.

9.7.2.2 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;
  }
}
```

9.7.2.3 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 $\Theta(a \cdot \log a)$.

9.7.3 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.

9.7.3.1 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.

9.7.3.2 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.

9.8 Algorismes de camins mínims

9.8.1 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.

9.8.2 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.

9.8.2.1 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 9.15, observem com funciona l'algorisme si triem com a vèrtex inicial el vèrtex A.

La taula 9.2 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.

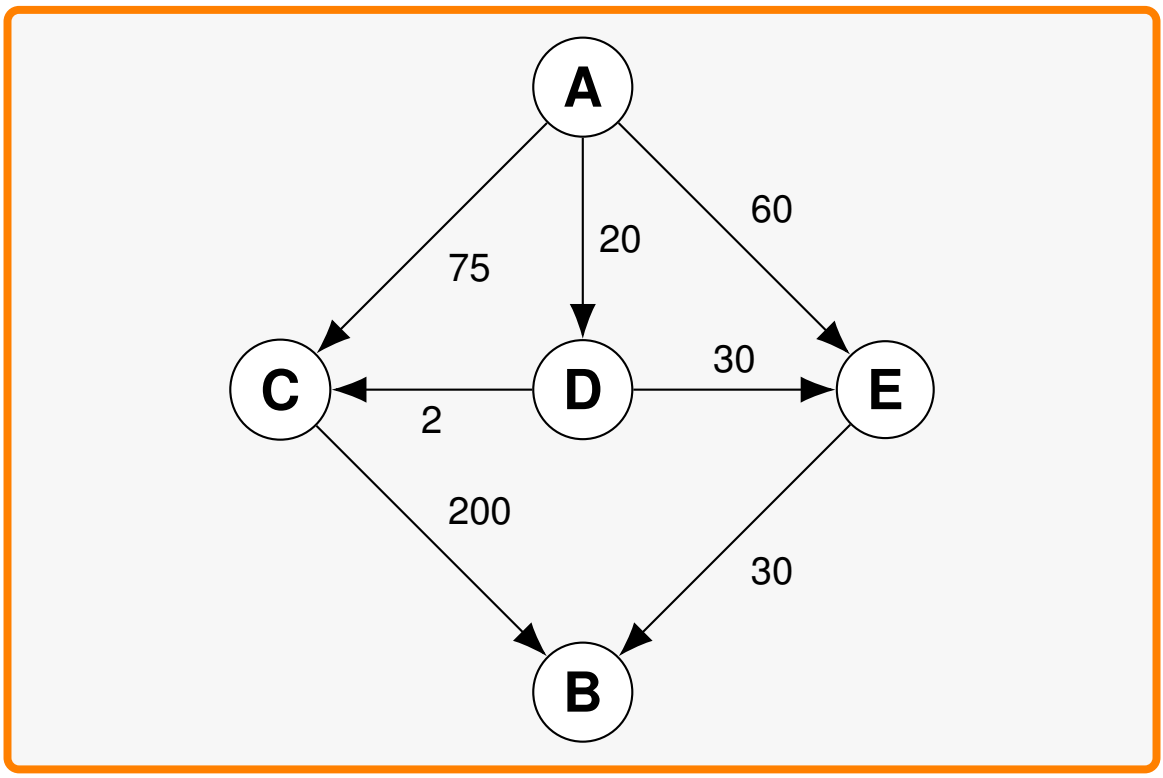


Figura 9.15: Graf dirigit i etiquetat

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 9.2: Estat de les estructures de dades en aplicar l'algorisme de Dijkstra en el graf 9.15. 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.

9.8.2.2 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;
}
```

9.8.2.3 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;
}
```

9.8.2.4 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.

9.8.3 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.

9.8.3.1 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 9.16 volem trobar els camins mínims entre tots els vèrtexs usant l'algorisme de Floyd.

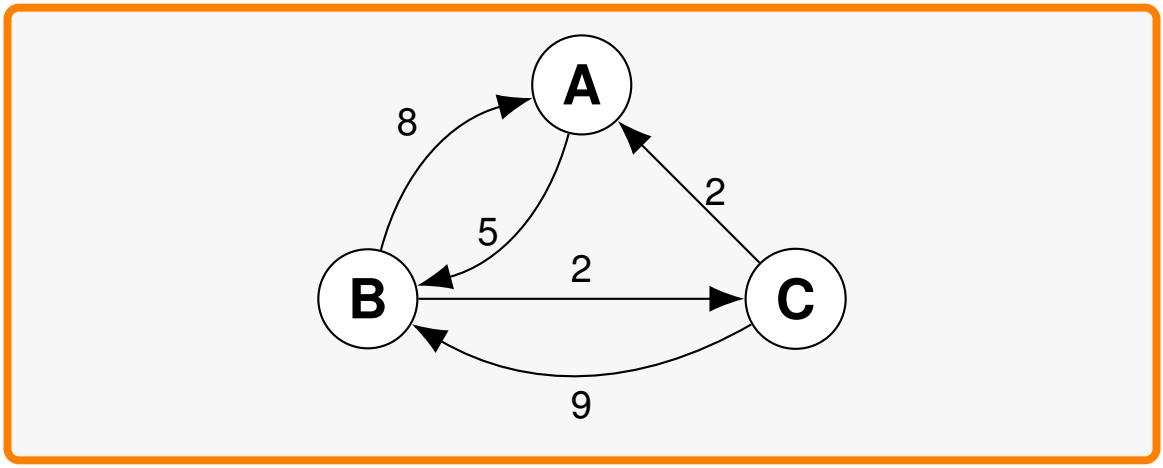


Figura 9.16: Graf dirigit i etiquetat

El resultat d'aplicar aquest algorisme es pot veure a la taula 9.3. 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 9.3: Esquema de l'aplicació de l'algorisme de Floyd en el graf 9.16.

9.8.3.2 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;
}
```

9.8.3.3 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.

9.8.3.4 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