Estructures de dades i algorismes

Tema 3 Map i Taula Hash

Curs 2018-2019

El model Map

El model Map

Map és un diccionari o array associatiu:

- Guarda parells *(clau,valor)*.
- No es permeten claus repetides.

  El mètode `equals` permet comprovar si dues claus són iguals o no.

- Permet cercar per la clau: donada una clau, volem saber si està en el Map i recuperar el seu valor associat.

Implementació en Java: la interfície Map

```
package librerias.estructurasDeDatos.modelos;
public interface Map<C, V> {
  // insereix/actualitza la entrada (c,v) torna
  // valor anterior o null si no estava
  V inserir(C c, V v);
  // elimina entrada con clau c, torna su
  // valor associat, o null si no existeix
  V eliminar(C c);
  // torna el valor associat a la clau c
  // o null si no existia eixa entrada
  V recuperar(C c);
  boolean esBuida();
  int talla();
  // torna una  ListaConPI que conte totes les claus
  ListaConPI<C> claus();
}
```

Exemples d'ús

- Traducció automàtica:
  - Traduir paraula per paraula requeriria un diccionari.
  - Encara que traduir paraula per paraula funcionaria malament, molts sistemes de traducció reals utilitzen diccionaris (realment enormes, doncs solen traduir grups de paraules i a més els assignen diverses alternatives acompanyades de probabilitats). - La taula de símbols d'un compilador (va emmagatzemant els noms de variables, classes, mètodes, etc.). Aquesta taula es modifica al llarg del procés de compilació (alguns símbols s'introdueixen i després s'eliminen: les variables locals solament existeixen en el seu mètode, etc.).

Exemples d'ús

- Calcular la freqüència d'aparició de les paraules d'un vocabulari en un document pot ser molt útil en algunes tasques de processament del llenguatge: podríem tenir un diccionari que associe cadenes a enters i utilitzar aquests enters per a explicar el nombre d'aparicions (un histograma). Podríem crear un diccionari que utilitze:
  - **Clau:** de tipus `String` per a emmagatzemar les paraules.
  - **Valor** de tipus `Integer` (la classe *embolcall* de `int`) on guardarem, per a cada paraula, el nombre de vegades que ha aparegut.

Implementació d'un Map amb una llista enllaçada

- Coneguda com association list.
- Mantindríem una llista de parells *(clau,valor)*.
- Cost d'inserir, cercar i esborrar és **lineal** amb el nombre d'entrades:
  - En el cas pitjor cal recórrer tota la llista.
  - En un cas mitjana, si cada valor de clau tinguera la mateixa probabilitat, el cost mitjana seguiria sent lineal.
- Podríem tenir la llista ordenada per claus quins avantatges i inconvenients tindria?
  - El cas pitjor i mitjana segueix sent lineal.
  - Com cal comparar claus, ja no prou `equals`, seria necessari comparar-los (ja posats, usaríem arbres que permetrien implementacions amb cost logarítmic, com veurem en el tema 4).

Implementació d'un Map amb un vector

Si el conjunt de claus possibles anaren els nombres del `0` al `N-1`, és clar que podríem usar directament les claus com a índexs d'un vector de talla `N` on guardaríem els valors associats:

- Si una clau no està en el diccionari:
  - Si els valors són objectes, guardem `null`.
  - Si els valors són tipus bàsics, guardem (de ser possible) un *valor especial*, una implementació genèrica en Java no permet tipus bàsics (usa classes embolcall).
- El cost de cercar, inserir o esborrar és **constant**.
- Si el conjunt de valors no són els enters entre `0` i `N-1`:
  - Si és un conjunt xicotet, usem una funció que mapege cada valor a un enter diferent entre `0` i `N-1`.
  - Si el conjunt de valors és molt gran, seria molt ineficient en espai (si guardes un petit percentatge d'entrades) o directament impossible (ex: claus de tipus `String`).

Taules de dispersió

Taules de dispersió

- Les **llistes enllaçades** permeten implementar un diccionari per a qualsevol tipus de clau, però en aquest context lineal **és molt car**.
- D'altra banda, els **vectors** són molt **ràpids però** estan enormement **limitats** pel tipus de claus suportades.
- L'objectiu de les **taules de dispersió** és tenir un Map al mateix temps **general i eficient**.

Funcions de dispersió

Les taules de dispersió usen vectors i funcions que permeten convertir els elements a cercar en valors numèrics per a ser usats **com a índexs en el vector**. Aquestes funcions es coneixen com a **funcions de dispersió** (en anglès, *hash functions*).

Taules de dispersió

En general, una taula de dispersió (*hash*) es caracteritza per:

- Ús d'un vector per a accedir a les dades. Les components del mateix es denominen cubetes (o *buckets*).
- Una funció **de dispersió** per a obtenir un índex del vector a partir de la clau a cercar o a inserir.
- Un mecanisme per a **resoldre les col·lisions.** És possible que una funció de dispersió genere un mateix índex de vector per a elements diferents. Açò es coneix com a **col·lisió** i és **inevitable** si hi ha més tipus de clau que posicions en el vector.

Resolució de col·lisions

Existeixen diversos mecanismes per a tractar les col·lisions (veure enllaç), però en aquest tema ens limitem a descriure únicament dos:

- **Adreçament obert** *(open addressing)*: els valors es guarden en el propi vector. Les veurem molt per damunt (no veurem la seua implementació ni les utilitzarem en la resta de l'assignatura).
- **Encadenament** *(separate chaining)* o adreçament tancat: cada cubeta conté una llista amb tots els parells *(clau,valor)* corresponents a aquelles claus que han produït l'índex corresponent per mitjà de la funció de dispersió.

Taules de dispersió amb adreçament obert

- Els valors es guarden en el propi vector.
- Cada cubeta conté com a màxim un parell *(clau,valor)*.
- Si en inserir un element la cubeta ja està ocupada, cal cercar **una altra posició** fins a trobar una posició lliure.
- Existeixen diverses formes de cercar una altra posició (totes elles implementant circularitat):
  - **Exploració lineal**: anem avançant l'índex en intervals constants (normalment d'1 en 1).
  - **Doble hashing**: a partir de la clau obtenim tant l'índex inicial com el salt o increment.
  - **Exploració quadràtica**: des del primer índex anem sumant $1^2, 2^2, 3^2, \ldots, i^2, \ldots$

Taules de dispersió amb adreçament obert

Les taules amb adreçament obert tenen alguns inconvenients:

- No permeten emmagatzemar un nombre d'elements major que la talla del vector, si bé podem redispersar la taula per a augmentar la seua grandària.
- El cost d'inserir creix de manera dramàtica quan la taula comença a omplir-se.
- En esborrar un element cal marcar la posició de manera especial per a saber que en cercar cal seguir cercant però en inserir es pot considerar un buit.

Malgrat tot, són molt utilitzades i poden tenir avantatges en casos concrets (ex: permet esborrar algunes taules en temps constant, etc.).

Taules de dispersió amb encadenament

- Una vegada obtinguda la cubeta corresponent a una clau, no mirem en altres cubetes per a cercar aquesta clau.
- Resolució de col·lisions: permetem diversos parells *(clau,valor)* en la mateixa cubeta (normalment s'utilitza una llista).
- Vegem la idea de les taules de dispersió amb encadenament amb un exemple. En votar hi ha diverses taules electorals. Els votants determinen la seua taula segons la primera lletra del cognom:
  - En una mateixa taula et cerquen en el cens utilitzant una llista.
  - Si el nombre de votants en cada taula és similar, **el cost** de cercar en la llista **es divideix entre el nombre de taules**.

Taules de dispersió amb encadenament

Seguint el símil de les taules electorals:

- "cognoms i nom" seria la clau del Map.
- El conjunt de taules electorals seria el conjunt d'índexs de `0` a `N-1` en un vector de talla `N`.
- Cada taula electoral seria una cubeta.
- Determinar la taula amb el cognom seria aplicar una **funció de dispersió** a la clau per a obtenir un valor entre `0` i `N-1`.
- La llista de votants d'una taula electoral seria una llista enllaçada amb els parells *(clau,valor)* tal que totes aqueixes claus han caigut en aquesta cubeta en particular.

Exemple de taula amb encadenament

Suposem que inserim els següents valors a la taula: 325, 12, 100, 30, 145, 89, 75, 237 i que la funció de dispersió retorna el propi enter mòdul la talla del vector. El resultat seria el següent:

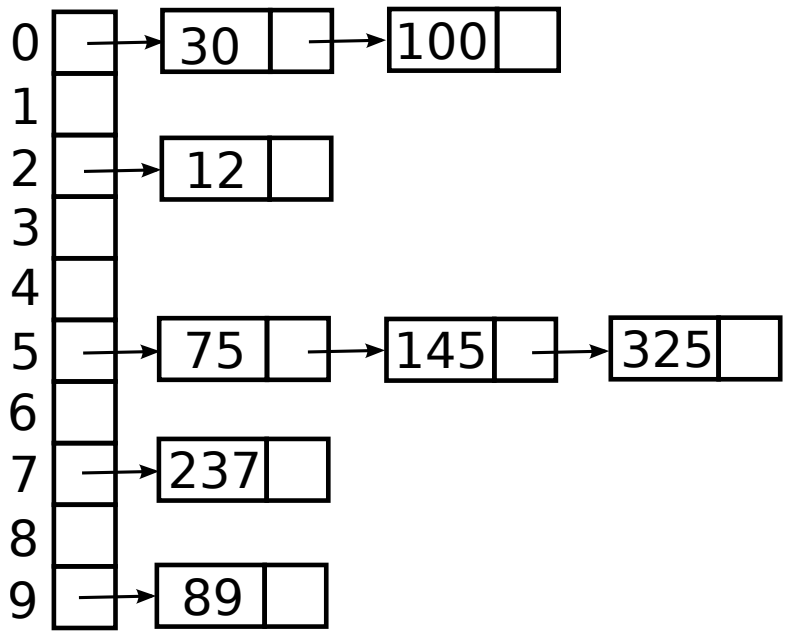


Figura: Exemple de taula de dispersió

Implementació mitjançant llistes directes

- En una taula amb resolució de col·lisions per encadenament els parells *(clau,valor)* es guarden en una llista enllaçada.
- Definim una classe genèrica que emmagatzeme conjuntament la clau i el valor d'una entrada.

```
class EntradaHash<C, V> {
   C clau;
   V valor;
   EntradaHash<C, V> seg;
   public EntradaHash(C clau, V valor,
                      EntradaHash<C, V> seg) {
     this.clau = clau;
     this.valor = valor;
     this.seg = seg;
   }
}
```

Implementació: constructor

- El constructor rep el nombre d'elements a emmagatzemar i reserva espai per a guardar-los amb un factor de càrrega de 0.75 (imitant `HashMap` de `java.util.Collections`).
- Per a moltes funcions de dispersió funciona millor un nombre inicial de cubetes.

```
public class TablaHash<C, V> implements Map<C, V> {
  // Array de llistes d'entrades
  private EntradaHash<C,V> elArray[];
  // Nombre de dades emmagatzemades en la taula
  private int talla;
  @SuppressWarnings("unchecked")
  public TablaHash(int tallaMaximaEstimada) {
   int capacitat= siguientePrimo((int)
          (tallaMaximaEstimada/0.75));
   elArray = new EntradaHash[capacitat];
   talla = 0;
}
```

Implementació: inserció

```
// Insereix l'entrada (c,v) i retorna l'antic valor
// que tenia aquesta clau (o null si no tenia valor)
public V inserir(C c, V v) {
  V anticValor = null;
  int pos = indexHash(c);
  EntradaHash<C, V> e = elArray[pos];
  while (e != null && !e.clau.equals(c))
    e = e.seg;
  if (e == null) { // Nova entrada
    elArray[pos] = new EntradaHash<C,V>(c, v,
    elArray[pos]);
    talla++;
  } else { // Entrada existeix
    anticValor = e.valor;
    e.valor = v;
  }
  return anticValor;
}
```

Implementació: esborrat

```
// Elimina l'entrada amb clau c i retorna el seu valor
// associat (o null si no aquesta aqueixa clau)
public V eliminar(C c) {
  int pos = indexHash(c);
  EntradaHash<C, V> e = elArray[pos], ant = null;
  while (e != null && !e.clau.equals(c)) {
    ant = e;
    e = e.seg;
  }
  if (e == null) return null; // No trobat
  if (ant == null)
    elArray[pos]  = e.seg;
  else
    ant.seg = e.seg;
  talla--;
  return e.valor;
}
```

Implementació: recuperar, esBuida, talla

```
// Cerca la clau c i retorna la seua informacio associada
// o null si no hi ha una entrada amb aquesta clau
public V recuperar(C c) {
  EntradaHash<C, V> e = elArray[indexHash(c)];
  while (e != null && !e.clau.equals(c))
    e = e.seg;
  if (e == null) return null;
  return e.valor;
}

// Retorna true si el Map esta buit
public boolean esBuida() { return talla == 0; }

// Retorna el nombre d'entrades que conte el Map
public int talla() { return talla; }
```

Implementació alternativa: llistes amb PI

Hem vist una implementació basada en llistes directes. També es poden utilitzar llistes amb punt d'interès:

```
package librerias.estructurasDeDatos.deDispersion;
class EntradaHash<C, V> {
  C clau; V valor;
  public EntradaHash(C clau, V valor){
    this.clau = clau; this.valor = valor;
  }
}
public class TablaHash<C, V> implements Map<C, V> {
  protected ListaConPI<EntradaHash<C,V>> elArray[];
  protected int talla;
  ... /* EXERCICI: COMPLETAR */
}
```

La classe `EntradaHash` ja no porta una referència a *seg*. Aquesta versió també serveix per a taules amb adreçament obert.

Funcions de dispersió

Funció hash o de dispersió

- L'objectiu de la funció de dispersió és generar un enter entre `0` i `N-1` que complisca certes propietats.
- En Java existeix una funció estàndard en la classe `Object` que es denomina `hashCode`:

```
public int hashCode() { ... }
```

- `hashCode` **no coneix el nombre de cubetes** així que retorna un valor en el rang dels enters (4 bytes) incloent potencialment els negatius.
- El mètode `hashCode` heretat de `Object` no sempre és adequat, per la qual cosa normalment es redefineix o s'utilitza per a implementar la funció que obté l'índex del vector o la cubeta associada a l'element.

Funció de compressió

- Per a obtenir l'índex de la cubeta a partir del valor calculat per `hashCode` hem de convertir aquest valor al rang `0` a `N-1`.
- La funció de compressió normalment es basa en el mòdul de la divisió.
- És important recordar que `hashCode` pot retornar negatius i que l'operador `%` retorna, en Java, el valor amb el signe del dividend, d'ací el `if` d'aquesta implementació del mètode `indexHash` de la classe `TablaHash`:

```
public int indexHash(C c) {
  int valorHash  = c.hashCode();
  int indexHash = valorHash % elArray.length;
  if (indexHash<0) indexHash += elArray.length;
  // 0 <= indexHash < elArray.length
  return indexHash;
}
```

Funció hash o de dispersió

La funció de dispersió ha de complir algunes propietats:

- **Obligatori:** que siga determinista i que done el mateix valor per a claus que siguen iguals d'acord amb `equals`.
- **Inevitable:** és possible que dos valors diferents acaben caient en la mateixa cubeta, açò es denomina una **col·lisió** i és inevitable si el nombre de claus possibles és major que el nombre de cubetes.
- **Desitjable:** que siga ràpida d'avaluar i que totes les cubetes reben aproximadament el mateix nombre d'elements per a equilibrar-les el més possible: els índexs de les claus que anem inserint s'ha de semblar a traure números entre `0` i `N-1` de manera equiprobable i amb reemplaçament.

Funcions de dispersió habituals

Algunes funcions de dispersió habituals són:

- Multiplicativa.
- Divisiva.
- Polinòmica.

En Java, la funció `hashCode` per a la classe `String` es calcula com si els caràcters de la cadena foren els coeficients d'una funció de dispersió polinomial en base 31 (s'ha escollit aquest valor perquè és primer). Per a una cadena `s` de talla `n` es calcularia així (amb la restricció que ha de cabre en un enter):

$$s[0] * 31^{(n-1)} + s[1] * 31^{(n-2)} + \cdots + s[n-2] * 31 + s[n-1]$$

Com de bé està funcionant la taula de dispersió?

- Quants elements hi han de mitjana per cubeta? o quantes col·lisions s'han produït?
- La mitjana o **factor de càrrega** ens contesta aquesta pregunta, i es defineix com el nombre d'elements dividit entre el nombre de cubetes.
- Com de ben repartits estan els elements en les cubetes?
- La variància ens diu com de pròxima està la ocupació de totes les cubetes a la mitjana.

Histograma d'ocupació

És una manera gràfica de veure com de ben dispersats estan els elements d'una taula. Indica quantes cubetes tenen 0 elements, 1 element, etc.

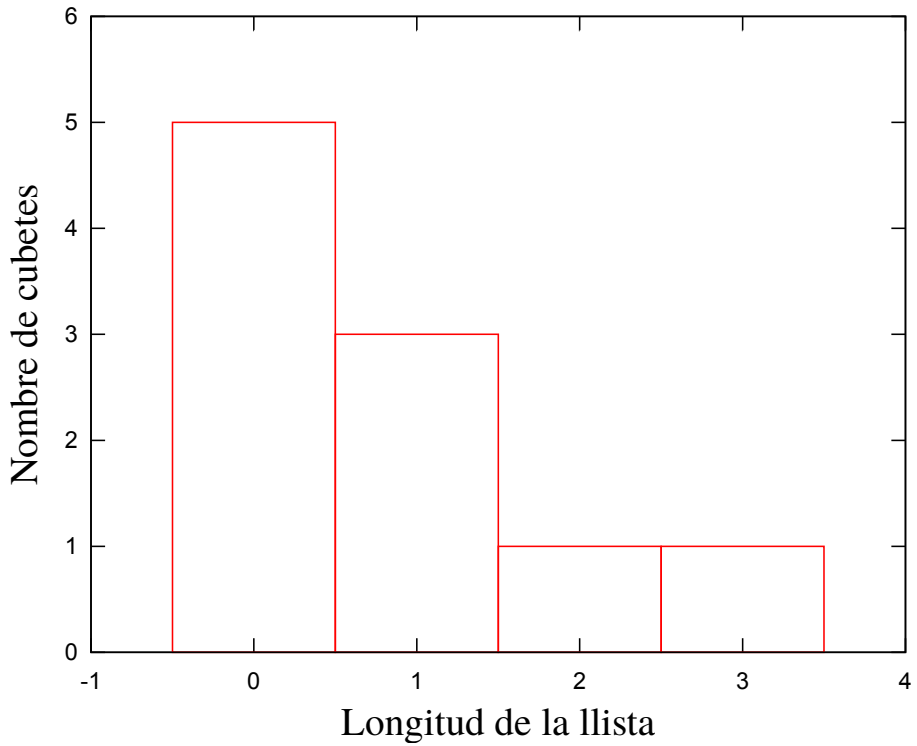
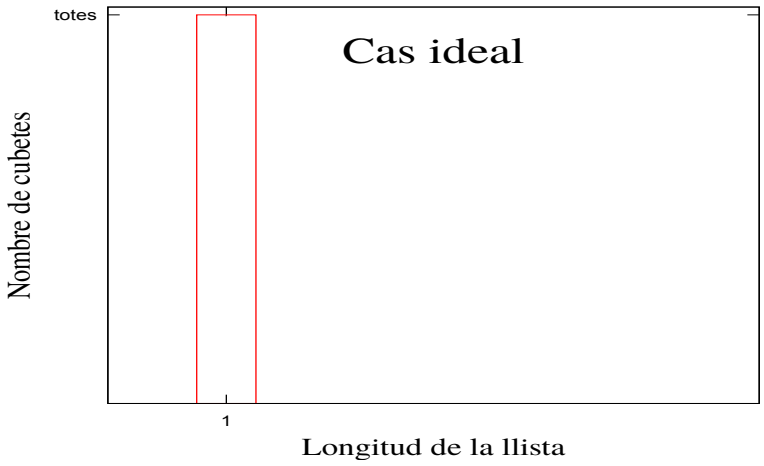
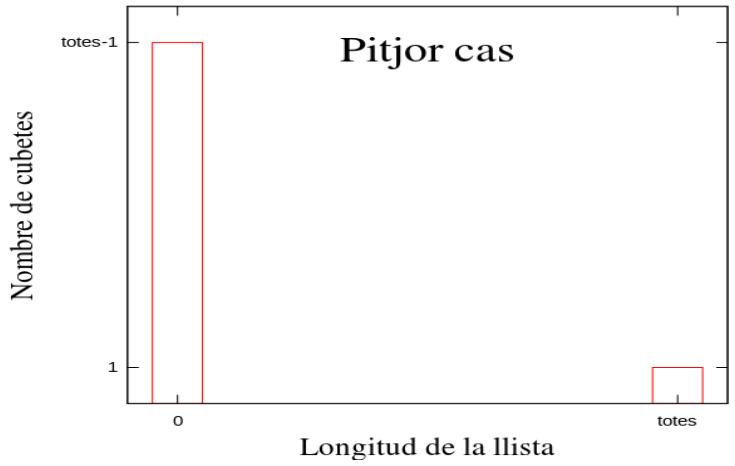


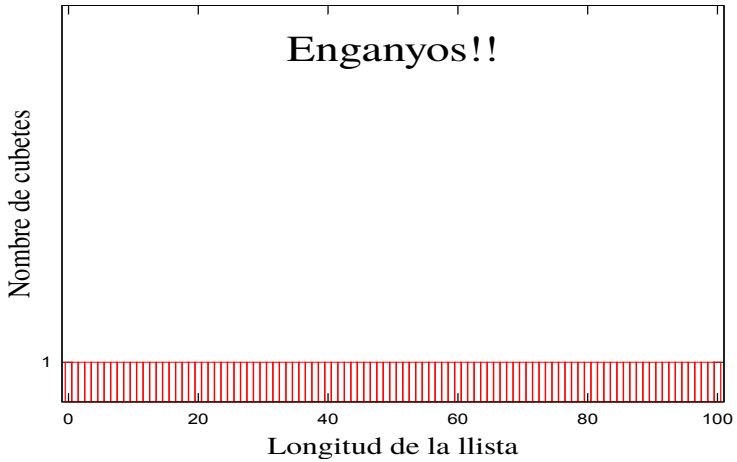
Figura: Histograma d'ocupació de l'exemple anterior

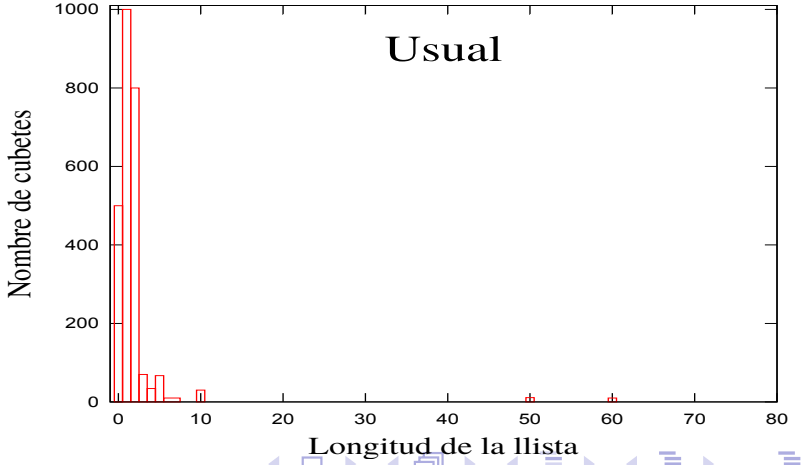
Histograma d'ocupació

L'histograma d'ocupació permet veure "a colp de vista" com estan distribuïdes les claus en una taula de dispersió:









Dispersió perfecta (perfect hashing)

Algunes aplicacions utilitzen **diccionaris que no es modifiquen**:

> *Un GPS porta un mapa amb un diccionari que associa a cada ciutat les seues coordenades geogràfiques.*

No es tracta de la interfície Map vist en la introducció, ja que un diccionari estàtic no permet insercions ni esborrats.

És possible dissenyar una funció de dispersió *ad hoc* per al conjunt de claus conegut a priori de manera que no es produïsquen col·lisions.

Açò es coneix com perfect hashing i permet implementacions molt eficients de diccionaris estàtics.

Costos i redispersió

Costos

- El **cost espacial** d'una taula amb M elements i N cubetes és $O(N + M)$.
- El **cost temporal** d'inserir i de cercar ve donat pel cost de la funció de dispersió seguida pel cost de cercar en la llista de la cubeta corresponent. Aquest cost depèn de la longitud de la llista ponderat pel cost del mètode `equals`:
  - *El cost mitjana* de cercar en la taula es correspon amb el factor de càrrega.
  - *El cas pitjor* depèn de la longitud més llarga de les llistes. Aquest valor pot ser molt major que el valor mitjà si la longitud de les llistes té molta variància.
- **Per a un nombre de cubetes fix, el cost asimptòtic** d'inserir/cercar/esborrar creix de manera **lineal** amb el nombre d'elements, si bé amb una constant $1/N$ molt baixeta.

Costos i dispersió

El **cost amortitzat** és el cost mitjà d'un conjunt d'operacions.

> ***Exemple:*** *si solament vas a realitzar un trajecte amb metro, el cost d'un bitllet senzill és menor que el cost d'un bonometro, però si tens en compte que vas a realitzar molts viatges, arriba un moment en què compensa comprar el bonometro.*

De manera similar, volem que el cost mitjana d'una sèrie d'insercions en una taula de dispersió siga en mitjana constant (que el cost total de M insercions siga $O(M)$). Per a açò el nombre de cubetes ha d'augmentar quan el factor de càrrega supere un límit, encara que aqueix augment siga **puntualment** car.

La redispersió (o "rehashing")

Consisteix a modificar el nombre de cubetes. Aquest procés, per a passar de N1 a N2 cubetes, es divideix en 3 fases *(aquest mètode és un exercici de pràctiques)*:

1. Crear una taula nova de talla N2 sense perdre (encara) la referència a la taula anterior.
2. Traslladar cadascun dels M nodes de les llistes enllaçades de les cubetes antigues a les noves. En general, els nodes van a canviar de cubeta i cal tornar a calcular el hashCode. En inserir en la cubeta nova **no fa falta** comprovar que l'element a inserir ja existeix, com ocorre amb una inserció convencional.
3. La taula original s'ha quedat sense cubetes, deixem de referenciar-la perquè el recol·lector de brossa la puga eliminar.

El cost temporal és $O(N1 + N2 + M)$.

Cost amortitzat de redispersar

Analitzem el cost amortitzat si redispersem doblant el nombre de cubetes cada vegada que el factor de càrrega arribe a 2:

elements	cubetes	factor de càrrega	cost redispersar
0	100	0	
199	100	≈ 2	
200	200	1	$100 * (1 + 2^2)$
399	200	≈ 2	
400	400	1	$100 * 5 * 2$
799	200	≈ 2	
800	400	1	$100 * 5 * 2^2$
$\vdots$	$\vdots$		

Si anem sumant el cost total d'inserir tots els elements (cadascun fitat pel factor de càrrega menor a 2) més el cost de les redispersions i ho dividim entre el nombre total d'insercions es comprova que el **cost mitjana per inserció és constant**.

Exercicis

Alguns exercicis d'auto-estudie

- **Exercici 1** Inserir els `Integer` 9, 7, 3, 17, 18, 16 i 12 en una Taula Hash de `elArray.lenght=10` i `c.hashCode()=c.intValue()`. Calcular el seu Factor de Càrrega i dibuixar l'histograma d'ocupació.
- **Exercici 2** Inserir els `Integer` 17, 29, 50, 24, 46, 35, 15, 14, 11, 32, 8, 44 i 25 en una Taula Hash de `elArray.lenght=10` i `c.hashCode()=c.intValue()`, fent `rehashing` doblegant la grandària de la taula cada vegada que s'aconseguisca un factor de càrrega igual a 2.
- **Exercici 3** Proposa un interfície `Set` per a representar un conjunt. Adapta la implementació de `TablaHash` basada en llistes directes per a implementar una classe `ConjuntHash` que implemente la interfície `Set`.

Alguns exercicis d'auto-estudie

- **Exercici 4** Fes una classe amb un mètode estàtic genèric que rep un vector d'elements i que calcule la moda d'aquesta col·lecció (que retorne l'element que es repeteix més vegades, en cas d'haver-hi diversos elements que es repeteixen aqueixes vegades serviria retornar qualsevol d'ells).
- **Exercici 5** Ens interessa estendre la interfície `ListaConPI<E>` i la corresponent implementació de la classe genèrica `LEGListaConPI<E>` vistes en el tema 1 perquè:
  - Es puguen eliminar les entrades repetides (es deixaria la primera de cadascuna).
  - Un mètode `resta` que reba una altra llista `altra` i que elimine els elements de la pròpia llista que estan en `una altra`.

Alguns exercicis d'auto-estudie

- **Exercici 6** Utilitzant única i exclusivament els mètodes de la classe TablaHash i amb cost lineal, es demana dissenyar un mètode barrejar que donat un array v retorne un altre vector amb les mateixes components però barrejades, i.e. desordenades. Aquest mètode s'ha de basar que els elements inserits en un taula hash s'agrupen en cubetes seguint una funció de dispersió de manera que les claus queden en un ordre normalment diferent al que tenia el vector original.

Alguns exercicis d'auto-estudie

Exercici 7 Una aplicació de radars de tràfic utilitza un `Map` per a explicar les vegades que un cotxe ha passat superant el límit de velocitat. Sabent que una matrícula consta de 4 nombres seguit de 3 lletres, completa la classe `Matricula` afegint els mètodes i constructors necessaris:

```
public class Matricula {
  private int numeros; private String lletres;
  private String any;
  public Matricula(int n, String l){
      numeros = n; lletres = l;
  }
  public int getNumeros(){ return numeros; }
  public String getLletres(){ return lletres; }
  public String getAny(){ return any; }
  public String toString() {
    return numeros+" "+lletres+" "+any;
  }
}
```

Alguns exercicis d'auto-estudie

Exercici 7 (continuació) Per a comptabilitzar el nombre de vegades que ha passat un cotxe pel radar, es proposa completar el mètode:

```
public static void registrarMatricula(Map dic,
                                      Matricula matr) {
        /* COMPLETAR */
}
```

que insereix aquesta matrícula en la taula si el cotxe ha sigut vist per primera vegada o bé actualitza el nombre de vegades que ha sigut vist.

Alguns exercicis d'auto-estudie

- **Exercici 8** amb l'objectiu d'estalviar espai en memòria, es pot representar una matriu dispersa, i.e. amb molts elements nuls, mitjançant un diccionari en lloc d'un array bidimensional tal com s'explica en "aquest enllaç". És a dir, en aquest exercici has de:
  - Descriure breument què s'ha de considerar com a clau d'un diccionari que representa a una matriu dispersa. I què s'ha de considerar com el seu valor associat?
  - Definir una classe `Clau_MD` per a representar una clau d'una matriu dispersa; en concret, s'han de definir els seus atributs, el seu mètode constructor no buit, els mètodes consultors dels atributs definits i els seus mètodes `equals` i `hashCode`.
  - Escriure un programa Java que mostre per pantalla les posicions d'una matriu dispersa de `Integer` (`matriuD`) que contenen elements no nuls juntament amb el valor d'aquests.

Bibliografia

Bibliografia

- *Data structures, algorithms, and applications in Java* (Sahni, Sartaj) Universities Press.
  - Capítol 11, apartats 1 y 5 (excepte 11.5.3)
- *Estructuras de Datos en Java.* Weiss, M.A. Adisson-Wesley, 2000.
  - Capítol 6, apartat 7, y capítol 19.
- *Data structures and algorithms in Java* (Goodrich, Michael T.). John Wiley & Sons, Inc.
  - Capítol 9, apartats 1 y 2.