Estructura de dades

pràctica 1

Cristina Izquierdo Lozano i Aleix Mariné Tena

2017

Índex

[Disseny 2](#_Toc477300803)

[ENCRIPTAR i DESENCRIPTAR: 3](#_Toc477300804)

[TAD Cua 6](#_Toc477300805)

[Anàlisi del cost de les operacions 7](#_Toc477300806)

[Joc de proves 8](#_Toc477300807)

[Anàlisi temps d’execució del joc de proves 12](#_Toc477300808)

# **Disseny**

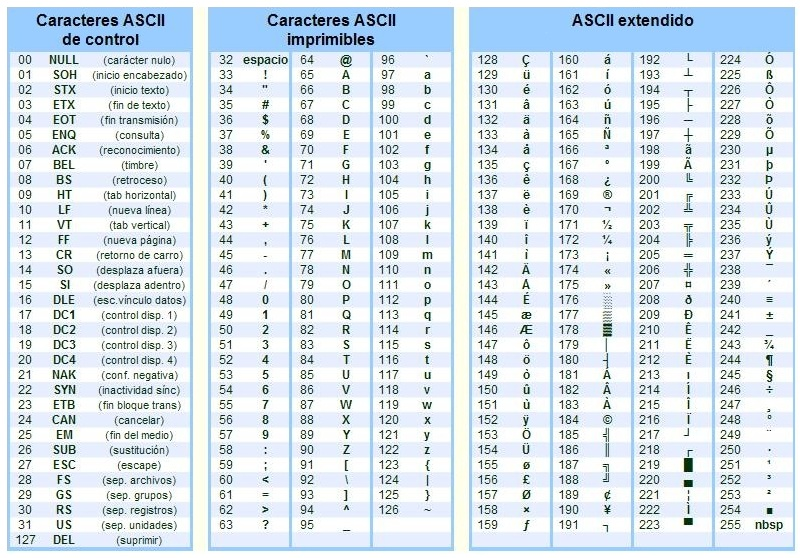
***breu explicació de com s’ha dissenyat la solució utilitzant el TAD Cua***

Hem dividit el nostre programa en diversos paquets per tal de separar les diferents funcions de cadascun, i programar de manera clara i ordenada. Aquests són els següents:

* **Paquet Aplicació**: En aquest paquet trobem una classe pel programa principal:
* **Aplicació:** Aquest fa la crida dels diferents mètodes i s’encarrega del tractament de dades. Hem utilitzat una estructura top-to-down, de manera que hem programat diferents funció que s’encarreguen d’una determinada tasca específica. Aquestes funcions son cridades al main. La descripció d’aquestes funcions i dels seus arguments així com dels seus retorns es pot trobar a la capçalera.
* **Paquet Dades**: En aquest paquet trobem les diferents classes creades per a cada tipus d’estructura, els quals implementaran els diferents mètodes definits a la interfície.
  + **Cua** - memòria estàtica, un vector i un índex de posició.
  + **CuaCircular** - memòria estàtica, cua circular sobre un vector.
  + **CuaDinamica** - memòria dinàmica, amb punters i referències. Dos índex de posició.
  + **JavaUtil** - Classe opcional, implementa la classe ArrayDeque, definida al java.util. La classe que hem creat nosaltres no és més que un wrapper d’una instància d’aquesta classe. D’aquesta manera podem fer que aquesta classe s’adapti a la interfície de cua.
  + **Node** - classe addicional per tal de poder implementar la cua dinàmica. Aquesta crearà una instància d’aquesta classe.
* **Paquet Exceptions**: Paquet que conté les diferents excepcions definides per tal d’evitar que el nostre programa es pengi.
  + **CuaBuida**: salta en cas de que no hi hagi cap valor a la cua, per exemple, quan intentem eliminar un valor d’una cua sense valors.
  + **CuaPlena**: salta en cas de que la cua hagi arribat al màxim d’elements. Per exemple, si intentem afegir un valor però no hi ha espai.
  + **Cadenabuida**: salta en cas de que introduïm una cadena de text buida. Per exemple, apretar intro sense haver escrit res.
  + **FitxerBuit**: salta en cas de que no es trobi cap missatge al fitxer indicat.
  + **valorImpossible**: salta en cas de que s’introdueixi un valor incorrecte (de manera general) o que no tingui sentit per teclat. Per exemple, un 0 a la clau.
* **Paquet Interface**: Paquet que conté la interfície del programa TADCua.
* **TADCua:** Ens serveix de guia per a implementar els mètodes de cada classe. També es abstracta i de la qual hereten les altres 4 classes tipus cua del paquet dades de manera que ens permet utilitzar una sola variable per a referenciar qualsevol dels tipus de tipus de cua que l’usuari vulgui utilitzar. Així només hem d’implementar les funcions del main un sol cop sobre una variable d’aquest tipus.

## **ENCRIPTAR i DESENCRIPTAR:**

Per a poder dur a terme les operacions criptogràfiques calia **definir un alfabet amb una equivalència numèrica** per a fer el desplaçament de caràcter indicat per la clau. La especificació de la pràctica no deia quin alfabet de caràcters havíem d’utilitzar així que nosaltres vam decidir utilitzar les lletres majúscules dels caràcters en ASCII (*American Standard Code for Information Interchange*). D’aquesta manera utilitzàvem l’equivalència numèrica entre caràcters i nombres donada per aquesta codificació (taula 1).



Taula 1

El conjunt de lletres majúscules eren els caràcters del 65 fins al 90, tots dos inclosos. Aquest rang de caràcters representa el conjunt de les majúscules en anglès i conformarà l’ **alfabet d’encriptació** **(AE)**.Val a dir que la ‘ç’ i la ‘ñ’ així com les seves majúscules no es troben incloses en aquest conjunt. La resta de caràcters que no estan presents en aquest conjunt s’ignoren, aquests caràcters seran considerats **caràcters especials (CE)**.

Per a dur a terme qualsevol dels dos algoritmes es duen a terme les següents operacions:

* Primer de tot es llegeix tot un fitxer el contingut del qual s’acumula en un sol String.
* Se li aplica el mètode *upperCase()* de la classe String, que serveix per a transformar totes les minúscules en majúscules. Aquest pas ens permet fer que tot el que sigui potencialment informació (lletres) es trobi en l’AE.
* L’String es transforma en un vector de caràcters mitjançant el mètode de la classe String *char[] toCharArray(String)*
* Es fa un recorregut de tot l’array de caràcters en que a cada posició de l’array se li fa un casting a int per a obtenir el corresponent nombre ASCII
* Es mira que es trobi en el rang de caràcters que formen el AE. Si es tracta d’un CE es passa a la següent posició. En el cas que sigui AE
  + Es desencua el valor de la clau i es torna a reencuar al mateix TAD escollit.
  + Aquest valor es resta (desencriptar) (Imatge 1) o es suma (encriptar) (imatge 2) o al nombre ASCII del caràcter.
  + En el cas de que el nombre que obtenim correspongui a un caràcter ASCII que es surti del nostre AE aplicarem operacions per a simular un AE circular. Això consisteix en obtenir el nombre de posicions que ens hem sortit del nostre alfabet per a començar a recomptar per l’altra banda.

Per exemple, si tenim el caràcter ‘X’ i la clau del missatge ens indica que hem de sumar 4 al caràcter, obtindríem el símbol ‘\’, però com que aquest caràcter no es troba en l’AE, busquem el nombre de caràcters que ens hem sortit de l’alfabet (en aquest cas ens sortim de dos caràcters), i ho sumem com si vinguéssim de la posició 0 de l’alfabet, de manera que obtenim el caràcter ‘B’. De manera anàloga succeeix a la hora de desfer la encriptació, però en lloc de sortir-nos per la part del final de l’alfabet, ens sortiríem del principi, pel que les operacions de correcció serien les inverses.

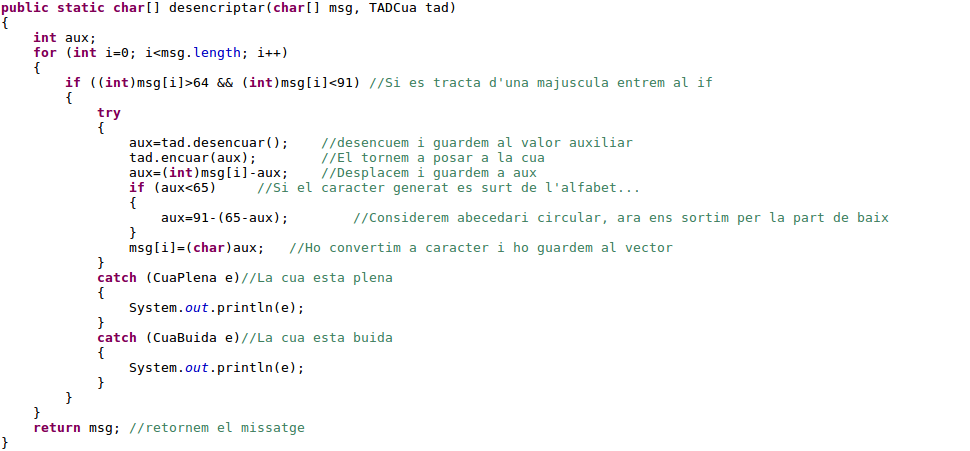
* + Actualitzem la mateixa posició de l’array de caràcters que estem tractant amb el caràcter que hem obtingut.
* Retornem l’array de caràcters i el escrivim a fitxer transformant-lo prèviament a String amb el constructor de la classe String.

Figura : mètode per a desencriptar

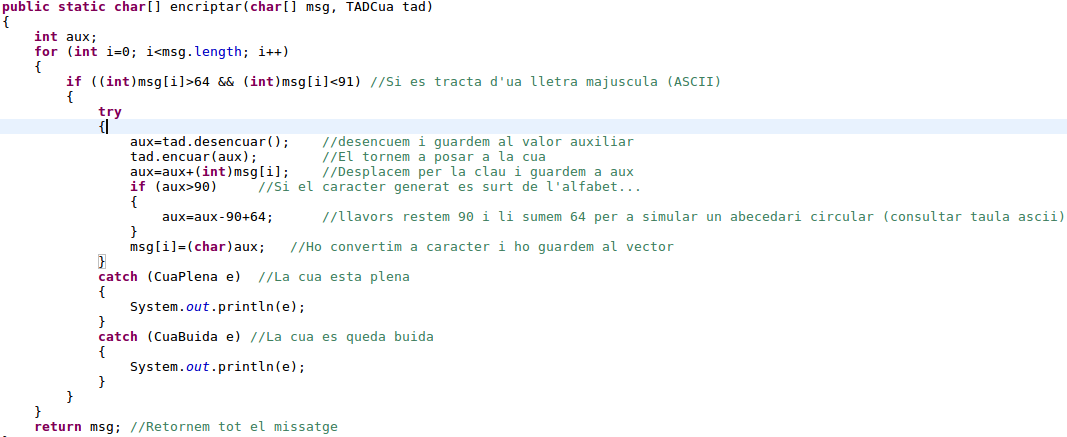
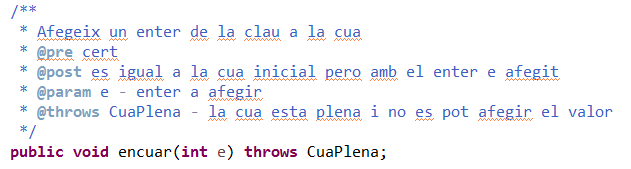


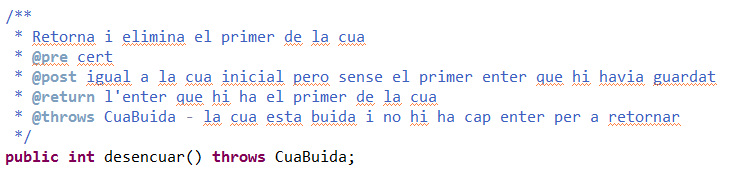
Figura : mètode per a encriptar

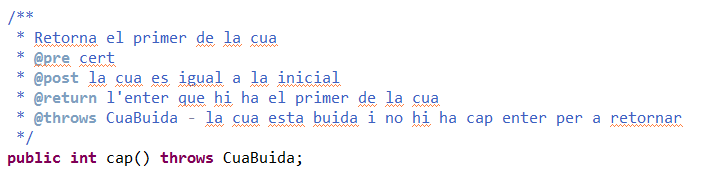
Podem observar que el cos dels dos mètodes es gairebé igual.

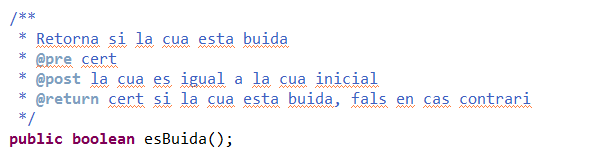
# **TAD Cua**

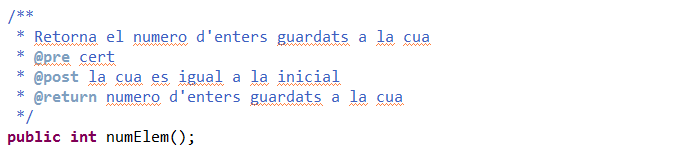
***l’especificació del TAD Cua amb el format pre/post***











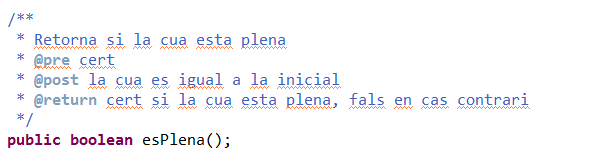


Figura 1 especificació del TADCua

# **Anàlisi del cost de les operacions**

***anàlisi de cost de les operacions en les diferents implementacions***

L’anàlisi del cost de les operacions es durà a terme amb una clau de 20 xifres (clau1).

Per tal d’analitzar el cost de les operacions, hem escrit 5 fitxers de text, amb el mateix missatge però cada vegada augmentava més (original, doble, triple, quàdruple i quíntuple).

Cost de les operacions:

1. **Cua estàtica**:
   1. **Contructor:** constant
   2. **Encuar:** constant
   3. **Desencuar:** lineal
   4. **Cap:** constant
   5. **esBuida:** constant
   6. **numElem:** constant
   7. **esPlena:** constant
2. **Cua circular:**
   1. **Contructor:** constant
   2. **Encuar:** constant
   3. **Desencuar:** constant
   4. **Cap:** constant
   5. **esBuida:** constant
   6. **numElem:** constant
   7. **esPlena:** constant
3. **Cua dinàmica:**
   1. **Contructor:** constant
   2. **Encuar:** constant
   3. **Desencuar:** constant
   4. **Cap:** constant
   5. **esBuida:** constant
   6. **numElem:** constant
   7. **esPlena:** constant
4. **Java.utils:** 
   1. **Contructor:** constant
   2. **Encuar:** constant
   3. **Desencuar:** constant
   4. **Cap:** constant
   5. **esBuida:** constant
   6. **numElem:** constant
   7. **esPlena:** constant

Com es pot observar, la majoria dels mètodes tenen un cost constant. L’explicació és que la majoria de mètodes no depenen del nombre d’elements que ja es troben a la cua. L’únic mètode que té un cost lineal és el desencuar de la cua estàtica ja que cal moure tots els valors una posició, pel que el cost d’aquesta operació depèn del nombre d’elements.

# **Joc de proves**

Dissenyem un joc de proves amb diferents claus. Una introduïda correctament i les altres amb errors que haurien de fer saltar les excepcions que hem afegit.

1. clau1: 12345678912345678912 (clau normal)
2. clau2: 98760432190765032198 (clau amb 0)
3. clau3: -1472583-694-7258364 (clau amb negatius)
4. clau4: 9G385274A9638B274189 (clau amb caràcters)

A més, també creem diversos fitxers de text amb errors que hauria de detectar o amb paraules que podrien ser conflictives pel programa, per exemple, introduir una Z i comprovar que l’alfabet sigui circular (que desplaci cap a la A en comptes d’anar-se’n als signes definits a l’alfabet ASCII). També introduïm símbols, tabulacions i accents, igual que números o lletres en minúscula (ja que tractem totes les lletres en majúscules i controlem que el programa les canviï).

Seguidament veiem els fitxers utilitzats:

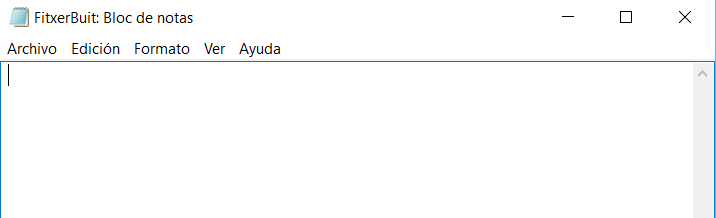


Figura 2 Fitxer buit, fa saltar la excepció "FitxerBuit"

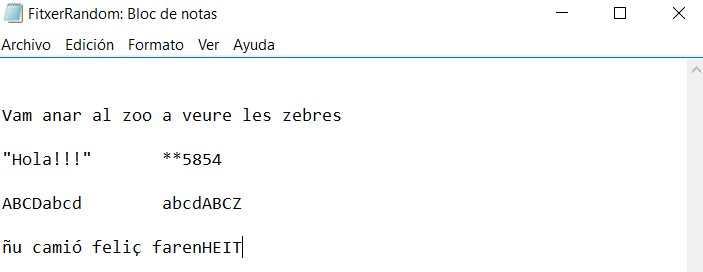


Figura 3 Fitxer amb text conflictiu, testeja els límits del programa

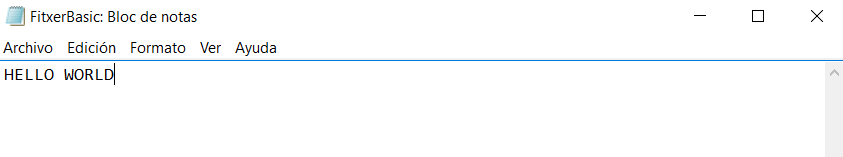


Figura 4 Fitxer amb text simple, per a calcular el cost de les operacions sense dificultats afegides

Ja que aquests excepcions es troben al programa principal, qualsevol de les diferents implementacions ha de donar la mateixa sortida, així que ho provarem amb les 4 TAD diferents.

Sortida dels jocs de proves:

1. Clau1, fitxer buit:

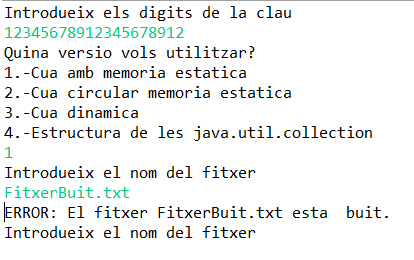


Figura 5

1. Clau1, fitxer inexistent:

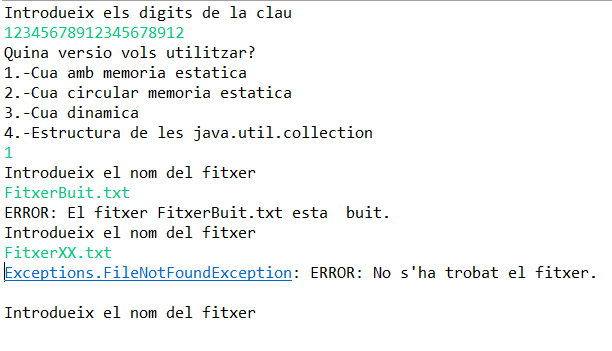


Figura 6

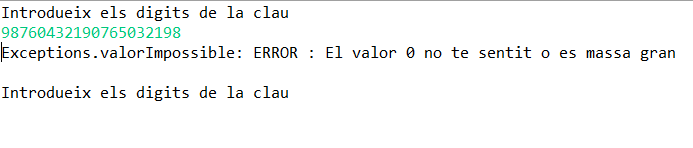
1. Clau2:

Figura 7

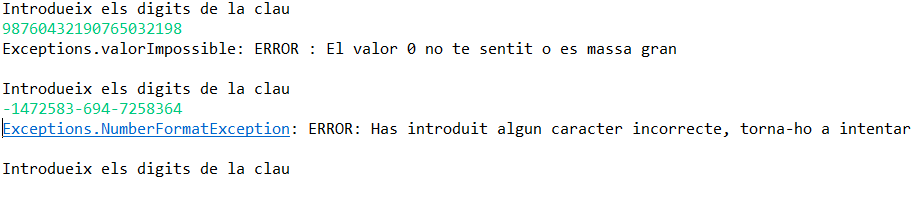
1. Clau3:

Figura 8

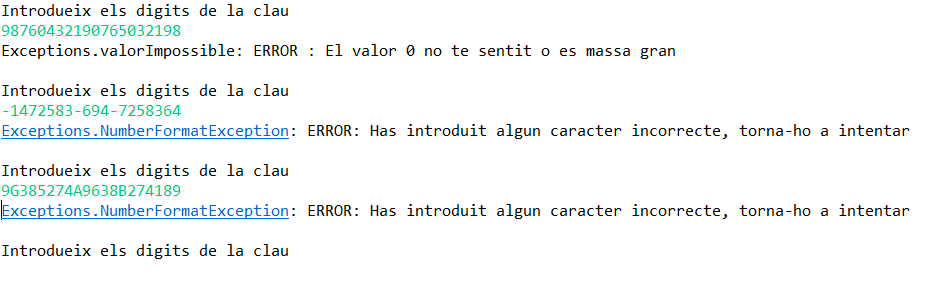
1. Clau4:

Figura 9

La sortida del joc de proves amb el FitxerRandom.txt és:

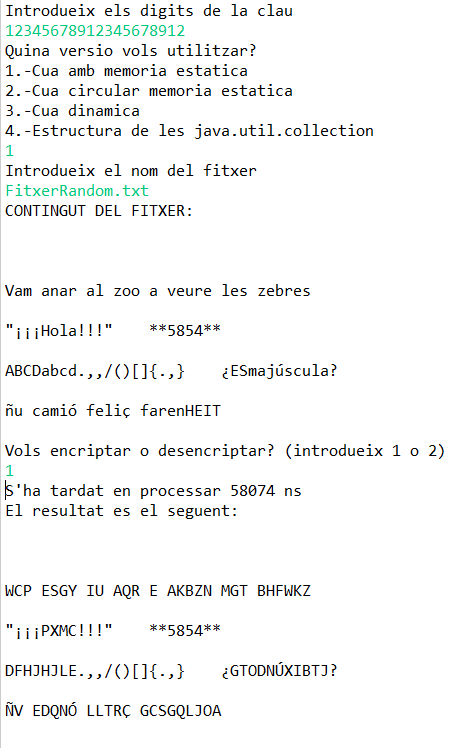


Figura 10 FitxerRandom, abans del tractament de dades

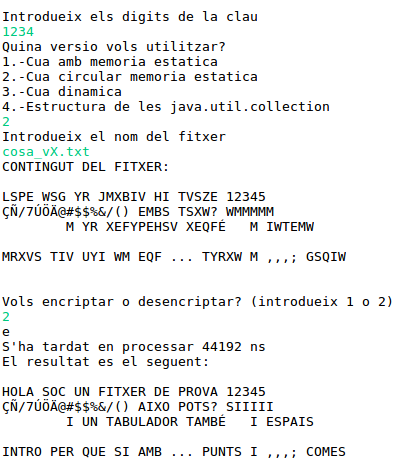


Figura 11 FitxerRandom, després del tractament de dades

# **Anàlisi temps d’execució del joc de proves**

***anàlisi dels temps d’execució dels diferents jocs de proves comparant les diferents implementacions. Utilitzar claus d’un mínim de 20 xifres.***

Hem separat el joc de proves de l’anàlisi de temps, fent un joc de proves específic per a aquesta part. L’anàlisi del temps d’execució el farem amb la clau1, que és la que no conté cap error que pugui fer saltar alguna excepció, i amb un text llarg que conté diferents tipus de caràcters. Hem utilitzat diferents arxius que contenen un nombre variable de repeticions del text (fins a 5 repeticions). Representem la mitjana dels temps d’encriptació i desencriptació front al nombre de repeticions del text en l’arxiu.

Els processos d’encriptació i desencriptació utilitzen bàsicament els mètodes d’encuar i desencuar, pel que el seu cost depèn bàsicament del cost d’aquestes dues operacions. A més els dos algoritmes són semblants (figura 1) (figura 2); és per això que utilitzem la mitjana del temps transcorregut entre aquestes dues operacions com a valor final pel gràfic. Per a cada TAD obtenim un gràfic:

Gràfic 1 Cua estàtica (lineal)

Podem observar com el gràfic té una certa forma quadràtica. Això és degut a que la funció representada té la forma

On x és el nombre de repeticions del text, x’ és un factor de cost que depèn de x de manera directament proporcional, k és un factor de proporcionalitat que indica com augmenta el temps front al nombre de repeticions i c és una constant. Així, podem substituir x’ en kx i dir que la nostra funció representada és:

D’aquesta manera, basant-nos en la primera equació podem veure que si substituïm x amb el valor de cada cas i assimilant les constant a c (constant) obtenim una funció:

És a dir, que el cost de l’algoritme en aquest TAD és lineal. Això és degut a que el mètode desencuar té un cost lineal com s’ha senyalat abans. La resta d’operacions tenen cost constant.

Gràfic 2 Cua circular (constant)

Gràfic 3 Cua Dinàmica (constant)

Gràfic 4 java.util (constant)

En els gràfics anteriors podem veure com les gràfiques tenen una forma:

On x és el nombre de repeticions del text, k és un factor de proporcionalitat que indica com augmenta el temps front al nombre de repeticions i c és una constant.

Amb un raonament anàleg al anterior podem veure com si substituïm x pel valor en cada cas i assimilem els valors constants a c, obtenim una funció:

És a dir, que el cost de les operacions en els TAD anterior és constant.

Cal destacar els diferents valors de k (Cost d’operació) que hi ha en cada cas:

* Veiem com el valor més baix de k és en la cua dinàmica, pel que el seu cost és el més baix entre tots els cues implementats. Això podria ser degut a que les operacions amb punters són més ràpides i també degut a que no cal consultar si la cua està plena al encuar, mentre que a les altres cues si que cal fer-ho.
* Els valors de k entre la cua estàtica circular i la cua java.util són força semblants degut a que implementen el mateix tipus d’estructura i operacions.
* Podem pensar que la k de la cua java.util és menor degut a que al tractar-se d’una llibreria aquesta classe ha sigut molt optimitzada, però realment la diferència entre ks no és significativa.

CONCLUSIÓ: La implementació amb menor cost és la cua dinàmica, amb un **cost lineal més baix**.