**Problema del caching offline:**

[Sapete che la cache è quella parte di memoria piccola ma molto costosa che consente un accesso molto veloce, dove si tengono i dati che devono essere elaborati in quel momento. Man mano che servono altri dati, essi vengono importati dalla RAM alla cache.]

Ho una memoria centrale che contiene un insieme di n elementi, ho una cache che potrebbe contenere al più k elementi, dove k << n e un insieme di m richieste, d\_1, d\_2, ..., d\_m. [Ovviamente lo stesso elemento potrebbe apparire più volte nella sequenza quindi sarà tolto e messo in cache]

Si parla di cache hit quando l'elemento di cui ho bisogno è già in cache.

Si parla di cache miss quando l'elemento di cui ho bisogno non si trova in cache. In questo caso devo importarlo in cache.

Introduciamo il concetto di eviction schedule ridotto: esso è uno scheduling che, data una cache piena, individua qual è l'elemento da rimuovere dalla cache quando si ha bisogno di importare un altro elemento. [Ogni volta che si ha un cache miss bisogna sostituire un elemento in cache.]. In altre parole, un eviction schedule ridotto è una sequenza di elementi che vengono via via rimossi dalla cache. Si chiama ridotto perché gli elementi in cache vengono sostituiti solo quando è strettamente necessario ed il numero di inserimenti è proprio uguale al numero di cache miss. Ci sono degli eviction schedule che possono sostituire degli elementi in cache anche quando non è necessario.

L'eviction schedule ottimale è l'eviction schedule ridotto che espelle dalla memoria in caso di cache miss l'elemento che verrà richiesto più in là nel tempo. [Qui sto facendo un'assunzione non banale: io conosco prima di iniziare l'algoritmo quale sarà la sequenza delle richieste. Nella versione non offline la soluzione che si utilizza normalmente è l'LRU, che è il contrario di quella che utilizzeremo ora.] Dimostreremo che la strategia migliore è la Farthest-In-Future, cioè quando c'è un cache miss si sostituisce l'elemento della cache che verrà richiesto più in là nel tempo. Per il teorema di Belady, si dimostra che la strategia Farthest-In-Future è ottimale.

**Vediamo l'implementazione dell'algoritmo di Belady con code a priorità.**

Assume the requests d1,d2,…,dn are arranged in ascending order of arrival time

For each element d, let L[d] the list of j s.t.dj=d [s.t. = se e solo se]

Let Q be a priority queue

for j = 1 to n

{

if(list L[dj] is empty and dj is in the cache)

Insert(Q, d\_j, j)

append j to list L[d\_j]

}

for j = 1 to n

{

remove first element from L[d\_j]

if(d\_j is NOT in cache)

{

d\_h <- ExtractMax(Q)

evict d\_h from the cache and bring d\_j to the cache

}

else

remove(Q,d\_j)

if(L[d\_j] is empty)

Insert(Q,d\_j,n+1)

else

{

p <- first element of L[d\_j]

Insert(Q,d\_j,p)

}

}

**Spiegazione**

Assume the requests d1,d2,…,dn are arranged in ascending order of arrival time

*Prendiamo in input la sequenza in ordine d'arrivo delle richieste [Così come ci è data].*

For each element d, let L[d] the list of j s.t.dj=d [s.t. = se e solo se]

*Per ogni elemento d che può apparire nella sequenza manteniamo una lista L[d]. [L[d] che cosa contiene? Contiene le posizioni in cui incontriamo d nella sequenza.]*

Let Q be a priority queue

*Utilizziamo una coda a priorità che conterrà gli elementi che si trovano in quel momento in cache, quindi conterrà esattamente k elementi. [Le entrate della coda a priorità saranno nella forma (elemento, chiave), dove la chiave sarà il tempo in cui sarà richiesto la prossima volta].*

*[Cominciamo la fase di inizializzazione, devo inizializzare le due strutture dati]*

for j = 1 to n

*per j che va da 1 a n, scandisco la sequenza quindi da sinistra verso destra.*

{

if(list L[dj] is empty and dj is in the cache)

*Se L[j] è vuoto vuol dire che quell'elemento d[j] è stato incontrato per la prima volta e dobbiamo vedere se d\_j è in cache.*

Insert(Q, d\_j, j)

*Se lo è dobbiamo inserirlo nella coda a priorità con chiave = j perché quello è l'istante in cui per la prima volta verrà richiesto j [perché qui dobbiamo togliere, se necessario farlo, un elemento dalla cache in base al tempo in cui questo verrà richiesto, togliamo ogni volta l'elemento che verrà richiesto più tardi. Dobbiamo tenere traccia di queste informazioni all'interno della coda a priorità]*

append j to list L[d\_j]

*Teniamo traccia di quest'informazione nella coda a priorità: ogni elemento della cache ha associato nella coda a priorità il prossimo istante in cui sarà richiesto.*

}

*[Riassunto: Ovviamente in fase di inizializzazione noi associeremo ad ogni elemento della cache il primo istante in cui verrà richiesto. Se d\_j è stato incontrato per la prima volta nella sequenza vuol dire che quell'elemento vienen richiesto per la prima volta al tempo j. Vado a vedere se d\_j è in cache. Se è in cache vado ad inserire nella coda a priorità (d\_j, j). Sia che d\_j fosse stato incontrato in precedenza, sia che fosse stato incontrato per la prima volta, vado ad appendere l'elemento j in L[d\_j] perché L[d\_j] contiene tutti i momenti in cui quell'elemento viene richiesto. Ragazzi, notato che le liste non sono n perché lo stesso elemento può essere richiesto più volte.][Ora cominciamo ad eseguire l'algoritmo]*

for j = 1 to n

*per j che va da 1 ad n, quindi vado a considerare le varie richieste, qui comincia l'algoritmo di Belady vero e proprio.*

{

remove first element from L[d\_j]

*[Al tempo j incontriamo l'elemento d\_j(Noi indichiamo con d\_j l'elemento j-esimo della sequenza). Sicuramente in testa alla lista ci sarà j]. Dobbiamo togliere il primo elemento dalla lista L[d\_j] in modo tale che in testa alla lista ora ci sarà il prossimo tempo in cui d\_j sarà richiesto. [Quell'informazione è fondamentale per aggiornare la coda a priorità.]*

if(d\_j is NOT in cache)

*Vado a vedere se d\_j è in cache o meno. Se non è in cache devo decidere quale elemento togliere dalla cache per far posto a d\_j. [La strategia mi dice di togliere l'elemento che sarà richiesto più in là. Ora siccome le chiavi nella coda a priorità rappresentano i tempi in cui i vari elementi in cache saranno richiesti in futuro, devo estrarre il massimo dalla coda a prorità in modo da estrarre dalla coda l'elemento che sarà richiesto più in là nel tempo, quindi faccio un ExtractMax(Q) dalla coda a priorità]*

{

d\_h <- ExtractMax(Q)

*Estraiamo il massimo dalla coda a priorità e lo salviamo in una variabile d\_h, che è l'elemento della cache che devo eliminare per far posto a d\_j.*

evict d\_h from the cache and bring d\_j to the cache

*Tolgo d\_h dalla cache per far posto a d\_j e porto d\_j in cache.*

}

else

*Se invece d\_j è già presente in cache, d\_j deve rimanere nella coda a priorità, ma devo modificarne la chiave. [Per sintesi, per scrivere quanto meno codice è possibile ho deciso di rimuoverlo tanto poi lo reinserirò con la nuova chiave.]*

remove(Q,d\_j)

*rimuovo d\_j dalla coda a priorità. [Assumo di avere una funzione remove che rimuove un'arbitraria entrata dalla coda a priorità.*

*[A questo punto devo fare un'aggiornamento delle liste della coda a priorità.Nella coda a priorità io ho inserito d\_j, a questo punto d\_j deve essere inserito nella coda a priorità perché è presente in cache. Anche nel caso in cui era già presente, l'ho rimosso e deve essere re-inserito, quindi sia che fosse presente in cache sia che non lo fosse, io devo fare un inserimento in coda a priorità. Devo decidere qual è la chiave da assegnare a d\_j.]*

if(L[d\_j] is empty)

*Può succedere che la lista sia vuota, cioè che che j è l'ultima volta in cui viene richiesto d\_j. Assegno come chiave n + 1 perché le richieste sono n, e n + 1 è un intero sufficientemente grande da garantirmi che nessun'altra richiesta avrà una chiave così grande, così nel caso in cui in futuro ci sarà bisogno di togliere qualche elemento dalla cache verrà tolto questo elemento o qualche altro che non verrà richiesto in futuro.*

Insert(Q,d\_j,n+1)

else

*Se d\_j è già presente in cache. [d\_j deve rimanere nella coda a priorità ma devo modificarne la chiave. La chiave da assegnare si trova in testa alla lista L[d\_j] perché io ho tolto il primo elemento che conteneva proprio j e quindi ora alla testa della lista si trova proprio il prossimo tempo in cui d\_j sarà richiesto.]*

{

p <- first element of L[d\_j]

*Prendo questo tempo e lo uso come chiave di d\_j nella coda a priorità, quindi*

Insert(Q,d\_j,p)

*prendo questa coppia (d\_j, p) e lo inserisco nella coda a priorità.*

}

}

**Analisi tempo**

Assume the requests d1,d2,…,dn are arranged in ascending order of arrival time

For each element d, let L[d] the list of j s.t dj=d [s.t. = se e solo se]

*Per inserire gli elementi nella lista impiego tempo costante per ciascun elemento, ci sono n inserimenti, per un totale di n volte.*

Let Q be a priority queue

*Istanziazione della coda a priorità, tempo costante*

for j = 1 to n

*il for viene eseguito n volte. La cache contiene k elementi, k << n.*

{

if(list L[dj] is empty and dj is in the cache)

Insert(Q, d\_j, j)

*Ogni inserimento richiede tempo log(k), faccio k inserimenti.*

append j to list L[d\_j]

*L'append ad una lista (inserimento in coda) mi costa tempo costante (che per tutte le iterazioni del for è O(n))*

}

for j = 1 to n

*il for viene eseguito n volte*

{

remove first element from L[d\_j]

*La rimozione mi costa tempo costante (che per tutte le iterazioni del for è O(n))*

if(d\_j is NOT in cache)

*Supponiamo per efficienza di mantenere un flag per ogni elemento che indica se è in cache o no*

{

d\_h <- ExtractMax(Q)

*L'estrazione mi costa log(k), che nel peggiore dei casi, per tutte le iterazioni del for è O(nlog(k)))*

evict d\_h from the cache and bring d\_j to the cache

*Tempo costante, che nel peggiore dei casi, per tutte le iterazioni del for è O(n))*

}

else

remove(Q,d\_j)

*La remove mi costa log(k)*

if(L[d\_j] is empty)

*Test che richiede tempo costante*

Insert(Q,d\_j,n+1)

*L'insert costa log(k) (che per tutte le iterazioni del for è O(nlog(k)))*

else

{

p <- first element of L[d\_j]

*Lettura che richiede tempo costante*

Insert(Q,d\_j,p)

*L'insert costa log(k) (che per tutte le iterazioni del for è O(nlog(k)))*

}

}

*Quindi, in totale, grazie all'utilizzo di determinate strutture dati e di particolari accorgimenti, il totale è O(n) + O(klog(k)) + O(nlog(k)). Siccome k << n, Il tempo prevalente è O(nlog(k))*

**Dimostriamone l'ottimalità**

Paolo: mi scoccio di sbobinare, l'avevo già fatto ed ho perso il lavoro :D