Unit 3: Tecnicas avanzadas de diseno y analisis de algoritmos

Jeremy Barbay

18 April 2011

Índice

1. Material relevante de los años previos:

- Colas de Prioridades http://www.leekillough.com/heaps/http://www.leekillough.com/heaps/
- Arboles 2-3 (para "Finger Search Trees"
  - http://www.dcc.uchile.cl/ bebustos/apuntes/cc3001/Diccionario/4http://www.dcc.uchile.cl/˜bebustos/apuntes/cc3001/Diccionario/#4
- http://www.wimp.com/justcoincidence/http://www.wimp.com/justcoincidence/ Birthday Paradox, in English.
- Interpolation Search
  - CC3001?
- Counting Sort
  - CLRS
  - CC3001

2. Introduccion

1. Dominios discretos y finitos
   a) Busqueda en Dominios discretos y finitos
      - Interpolacion/extrapolation
      - Tries o arboles digitales
      - Arboles y Arreglos de Sufijos
      - Hash y Hash en memoria Secundaria
   b) Algoritmos de Ordenamientos con universo finito
      - Counting Sort
      - Bucket Sort
      - Radix Sort

2. Tecnicas de Analisis

    a) Analisis amortizada
        - tecnicas
        - colas de prioridades
        - splay arboles
    b) Analisis parametrizada
        - busqueda doblada y finger search trees
        - ordenamiento adaptivo
        - operaciones de conjuntos adaptivos
    c) Analisis de Algoritmos en linea
        - “ski renting” problema
        - analisis competitiva (“Competitive Analysis”)

3. Dominios discretos y finitos

3.1. Busqueda en domanios discretos

3.1.1. Busqueda por Interpolacion/Extrapolacion

1. Introduccion:

    - Hagaria busqueda binaria en un anuario telefonico para el nombre “Barbay”? En un diccionario para la ciudad “Zanzibar”?
    - ojala que no: se puede provechar de la informacion que da la primera letra de cada palabra

2. Algoritmo

    - Interaccion

3. Analisis * La analisis **en promedio** es complicada: conversamos solamente la intuicion matematica (para mas ver la publicacion cientifica de SODA04, Demaine Jones y Patrascu):

    - si las llaves son **distribuidas uniformamente**, la distancia en promedio de la posicion calculada por interpolacion **lineal** hasta la posicion real es de $\sqrt{r-l}$.
    - entonces, se puede reducir el tamaño del subarreglo de n a $\sqrt{n}$ cada (dos) comparaciones
    - la busqueda por interpolacion
        - en promedio,
        - si las llaves son **distribuidas uniformamente**,
        - toma $O(\lg\lg n)$ comparaciones

4. Interaccion.

5. Variantas

    a) Interpolacion non-lineal
        - en un anuario telefonico o en un diccionario, las frecuencias de las letras **no** son uniformes
    b) Busqueda por Interpolacion Mixta con Binaria
        - Se puede buscar en tiempo
            - $O(\lg n)$ en el peor caso Y

        - $O(\lg \lg n)$ en el caso promedio?
      - Solucion facil
      - Solucion mas compleja
   c) Busqueda por Extrapolacion
      - Tarea 3
   d) Busqueda por Extrapolacion Mixta con Doblada
      - Tarea 3

6. Discussion:
   - Porque todavia estudiar la complejidad en el modelo de comparaciones?
     - Cuando el peor caso es importante
     - Cuando la distribucion no es uniforme o no es conocida
     - cuando el costo de la evaluacion es mas costo que una simple comparacion (en particular para la interpolacion non lineal)

3.1.2. Tries o Arboles Digitales

Ordenamos usualmente como pre-computacion para buscar despues. En el caso donde n es demasiado grande, ordenar puede ser demasiado carro. Consideramos alternativas para buscar.

- Ejemplo de trie
- Insertar los nodos siguiente en un trie
  1. hola
  2. holistico
  3. holograme
  4. hologramas
  5. ola
  6. ole
  7. Busqueda
- con arreglos de tamaño σ en cada nodo:
  - $O(l)$ tiempo, pero $O(L\sigma)$ espacio
- con arreglos de tamaño variables en cada nodo:
  - $O(l \lg sigma)$ tiempo (busqueda binaria), $O(L)$ espacio (optima).
- con hashing
  1. $O(l)$ tiempo en promedio, $O(L)$ espacio.
  2. Insercion
- Insertar "hora" en el arbol precedente
- Insertar "holistico" en el arbol precedente
- Borrar "hola" y "holistica"
  1. (TAREA)

2. Tiene de “limpiar”, pero no costo mas que un factor constante de la busqueda.
3. BONUS: PAT Trie

- Comprime las ramas de nodos de grado uno en una sola arista.
- La caldena (“string”) etiquetando la arista se guarda en el nodo hijo de la arista.
- Superio tan en tiempo que en espacio en practica.

3.1.3. Arboles y Arreglos de Sufijos

1. Arbol de Sufijos
   - Espacio $O(n)$
   - Construccion $O(n)$
   - Busqueda $O(m)$
   - expresion regular $O(n^{\lambda})$ donde $0 \leq \lambda \leq 1$
2. Arreglo de Sufijos
   - Lista de sufijos ordenados
   - busqueda de patrones = dos busquedas binarias, donde cada comparacion costa $\leq m$, resultando en una complejidad de $O(m \lg n)$
3. BONUS: Rank en Bitmaps
   - `rank`(B, i) = cantidad de unos en $B[1, i]$
   - consideramos
     - B estatico
     - se puede almacenar $\lg n$ bits.
   - Solucion de Munro, Raman y Raman:
     - $b = 1/2 \lg n$ y $s = \lg^2 n$
     - Dividimos el index de B en
       - s Superbloques de tamaño $n/s \lg n$ bits
       - b Mini bloques de tamaño $n/b \lg s$ bits

$$\frac{n}{1/2 \lg n} \lg(\lg^2 n) = \frac{4n \lg\lg n}{\lg n} \in o(n)$$

       - un diccionario con todos los bit vectores de tamaño

$$\sqrt{n} \lg n / 2 \lg\lg n \in o(n)$$

3.1.4. Hashing

Introduccion * Motivaciones

- Mejor tiempo **en promedio**
- Uso de todos la herramientas que tenemos
  - dominio de las valores
  - distribuciones de probabilidades de las valores

* Terminologia

  - Tabla de Hash
    - Arreglo de tamaño N
    - que contiene n elementos a dentro de un universo $[1..U]$
  - Funccion de Hash $h(K)$
    - $h : [1..U] \rightarrow [0..N-1]$
    - se calcula rapidamente
    - distribue uniformemente (mas o menos) las llaves en la tabla en el caso ideal, $P[h(K) = i] = 1/N, \forall K, i$
  - Collision
    - cuando $h(K_1) = h(K_2)$
    - la probabilidad es alta: "Paradoxe del cumpleaños"
      - (http://www.wimp.com/justcoincidence/http://www.wimp.com/justcoincidence/)
    - Cual es la probabilidad que en una pieca de n personas, dos tiene la misma fecha de cumpleaños (a dentro de 365 dias)?
      - probabilidad que cada cumpleaños es unico:

$$\frac{364!}{(365-n)! \times 365^{n-1}}$$

      - Probabilidad que hay al menos un cumpleaños compartido:

$$\frac{1-364!}{(365-n)! \times 365^{n-1}}$$

n	Proba
10	.12
23	.5
50	.97
100	.9999996

Hashing Abierto

1. Idea principal:
   - resolver las colisions con caldenas
2. Ejemplo: (muy irealistico)
   - $h(K) = K \texttt{mod} 10$
   - Secuencia de insercion $52, 18, 70, 22, 44, 38, 62$
     - Insertando al final (si hay que probar por repeticiones)

0	70
1	
2	52,22,62
3	
4	44
5	
6	
7	
8	18,38
9	

- Insertando al final (si no hay que probar por repeticiones)

0	70
1	
2	62,22,52
3	
4	44
5	
6	
7	
8	38,18
9	

3. Analisis:
   - factor de carga es $\lambda = \frac{n}{N}$
   - Rendimiento en el peor caso: $O(n)$

Hashing Cerrado

1. Idea principal:
   - resolver las colisiones con busqueda, i.e.
     - $(h(K) + f(i))\texttt{mod}N$
   - differentes tipos de busqueda:
     - lineal $f(i) = i$
       - primary "clustering" (formacion de secuencias largas)
     - cuadratica $f(i) = i^2$
       - secundario "clustering" (si $h(K_1) = h(K_2)$, la secuencias son las mismas.
     - doble hashing $f(i) = i.h'(K)$
       - $h'(K)$ debe ser prima con N
2. Ideal Hashing
   - Imagina una funcion de hash que genera una secuencia que parece aleatoria.
   - cada posicion tiene la misma probabilidad de ser la proxima
     - Probabilidad λ de elegir una posicion ocupada
     - Probabilidad $1 - \lambda$ de elegir une posicion libre
     - la secuencia de prueba puede tocar la misma posicion mas que una vez.
     - llaves identicas todavia siguen la misma secuencia.
   - Cual es el costo promedio u_j de una busqueda negativa con j llaves en la tabla?
     - $\lambda = \frac{j}{N}$
     - $u_j = 1(1 - \lambda) + 2\lambda(1 - \lambda) + r\lambda^2(1 - \lambda) + \dots$
     - $= 1 + \lambda + \lambda^2 + \dots$
     - $= \frac{1}{1-\lambda}$
     - $= \frac{1}{1-j/N}$
     - $= \frac{N}{N-j} \in [1..N]$
   - Cual es el costo promedio de una busqueda positiva s_i para el i-th elemento insertado?
     - $s_i = \frac{1}{1-i/N} = \frac{N}{N-i}$

- $s_n = 1/n \sum \frac{N}{N-i}$
- $= \frac{N}{n} \sum_{i=0}^{n-1} \frac{1}{N-i}$
- $= \frac{1}{\alpha} \sum_{i=0}^{n-1} \frac{1}{N-i}$ con $\alpha = \frac{n}{N}$
- $< \frac{1}{\alpha} \int_{N-n}^{N} 1/x dx$
- $= 1/\alpha \ln \frac{N}{N-n}$
- $= 1/\alpha \ln \frac{1}{\alpha}$

- Para $\alpha = 1/2$, el costo promedio es 1,387
- Para $\alpha = 0{,}9$, el costo promedio es 2,559

Universal Hashing

- $h(K) = ((aK + b)\mathtt{mod}p)\mathtt{mod}N$
- $a \in [1..p-1]$ elegido al azar
- $b \in [0..p-1]$ elegido al azar
- p es primo y mas grande que N
- N no es necesaramente primo

Hashing en memoria externa

- Que pasa si la tabla de hashing no queda en memoria?
- IDEA: Simula un B-arbol de altura dos

  1. organiza el dato con valores de hash
  2. guarda un index en el nodo raiz
  3. usa solamente **una parte** de la valor de hash para elegir el sobre-arbol
  4. extiende el index cuando mas dato es agregado
  5. Descripcion

1. B - cantidad de elementos en una pagina
2. h - funcion de hash $\rightarrow [0.,2^k - 1]$
3. D - **profundidad general**, con $D \leq k$
   - la raiz tiene 2^D punteros a las paginas horas
   - la raiz es indexada con los D primeros bits de cada valor de hash.
4. d_l - **profundidad local** de cada hora l
   a) Las valores de hash en l tienen en comun los primeros d_l bits.
   b) Hay 2^{D-d_l} punteros a la hora l
   c) Siempre, $d_l \leq D$
   d) Ejemplo
5. $B = 4, k = 6, D = 2$

$$
\begin{array}{ll}
d_l = 2 & 000100 \\
 & 001000 \\
 & 001011 \\
 & 001100 \\
 & \\
d_l = 2 & 010101 \\
 & 011100 \\
 & \\
d_l = 1 & 100100 \\
 & 101101 \\
 & 110001 \\
 & 111100
\end{array}
$$

6. Algoritmos
   - Buscar
   - Insertar
   - Remover

7. Analisis
   - Buscar, insertar remover
     - 1 acceso a la memoria secundariaa si el index se queda
   - cantidad Promedio de paginas para tener n llaves
     - $\frac{n}{B \lg 2} \approx 1{,}44 \frac{n}{B}$
     - paginas son llenas a 69 % mas o menos.

3.2. Algoritmos de Ordenamiento (Counting Sort, Bucket sort, radix sort, string sort)

3.2.1. Counting Sort $O(\sigma + n)$

1. for $j = 1$ to σ do $C[j] \leftarrow 0$
2. for $i = 1$ to n do $C[A[i]] + +$
3. $p \leftarrow 1$
4. for $j = 1$ to σ do
   - for $i = 1$ to $C[j]$ do
     - $A[p + +] \leftarrow j$
       Este algoritmo es bueno para ordenar multi conjuntos (donde cada elementos puede ser presente muchas veces), pero pobre para diccionarios, para cual es mejor usar la extension logica, Bucket Sort.

3.2.2. Bucket Sort $O(\sigma + n)$

1. for $j = 1$ to σ do $C[j] \leftarrow 0$
2. for $i = 1$ to n do $C[A[i]] + +$
3. $P[1] \leftarrow 1$
4. for $j \leftarrow 2$ to σ do
   - $P[j] \leftarrow P[j-1] + C[j-1]$
5. for $i \leftarrow 1$ to n
   - $B[P[A[i]] + +] \leftarrow A[i]$
     Este algoritmo es particularmente practica para ordenar llaves asociadas con objetos, donde dos llaves pueden ser asociadas con algunas valores distintas. Nota que el ordenamiento es **estable**.

3.2.3. Radix Sort $O(n \lg_n \sigma) = O(cn)$

- Considera un arreglo A de tamaño n sobre alfabeto σ
- si $\sigma = n$, se recuerdan que bucket sort puede ordenar A en $O(n)$
- si $\sigma = n^2$, bucket sort puede ordenar A en $O(n)$:
  - 1 ves con los $\lg n$ bits de la derecha
  - 1 ves con los $\lg n$ bits de la izquierda (utilizando la estabilidad de bucket sort)
- si $|A| = n^c$, bucket sort puede ordenar A
  - en tiempo $O(cn)$
  - con espacio $2n + \sigma \approx 3n$ ($\sigma \approx n$ a cada iteracion de bucket sort)
    El espacio se puede reducir a $2n + \sqrt{n}$ con $\lg n/2$ bits a cada iteracion de Bucketsort, cambiando la complejidad solamente por un factor de 2.
    En final, si A es de tamaño n sobre un alfabeto de tamaño σ, radix sort puede ordenar A en tiempo $O(n\lceil \frac{\lg \sigma}{\lg n} \rceil)$

3.2.4. BONUS Provechando de las repeticiones en el modelo de Comparaciones

- Se puede o no?
- Ordenar en nH_i comparaciones

3.2.5. BONUS String Sort

- Problema: Ordenar k strings sobre alfabeto $[\sigma]$, de largo total $n = \sum_i n_i$.
- Si $\sigma \leq k$, y cada string es de mismo tamaño.
  - Si utilizamos bucket-sort de la derecha a la izquierda,

podemos ordenar en tiempo $O(n)$, porque $O(n\lceil \lg \sigma / \lg l \rceil)$ y $\sigma < n$.

- Si $\sigma \in O(1)$
  - Radix Sort sobre c simboles, donde c es el tamaño minima de una string, y iterar recursivamente sobre el restos de la strings mas grande con mismo prefijo.
  - En el peor caso, la complejidad corresponde a la suma de las superficias de los bloques, aka $O(n)$.

3.3. MAYB Heap

4. Tecnicas de Analisis

4.1. Analisi amortizada

4.1.1. MATERIAL A LEER

- "Amortized Analysis Explained" by Rebecca Fiebrink
- http://www.cs.princeton.edu/ fiebrink/423/AmortizedAnalysisExplained$_{F}iebrink.pdf http : //www.cs.princeton.ec$
- Amortized Analysis
  - CLRS, Chapter 17: Amortized Analizis p.405-430
- Árbol biselado (Splay Trees)
  - http://es.wikipedia.org/wiki/

4.1.2. Principio de Analisi amortizada

* Costo Amortizado:

- Se tiene una secuencia de n operaciones con costos $c_1, c_2, ..., c_n$.
- Se quiere determinar $C = \sum_{i\in[1..n]} c_i$.
- Se puede tomar el peor caso de $ci \leq t$ para tener una cota superior de $C \leq tn$.
- Un mejor analisis puede analizar el costo amortizado, con varias tecnicas:
  - analisis agragada
  - contabilidad de costos
  - funcion potencial.

* Applicaciones:

- Move To Front
- "Self-adjusting and balanced binary trees" (Splay Trees"
- union-find data-structures
- max flow
- Fibonacci heaps
- dynamic array (vector en Java)

* Tres tecnicas basicas:

1. "Aggregate analysis"
   - bound las proporciones de operaciones de cada tipo
   - (e.g. mas inserciones que deleciones)
2. "Accounting Method"
   - asigna un costo (positivo o negativo) a cada operacion

- ejemplos:
  - stack
  - java vector
- costo amortizado es la suma de los costos.

3. "Potential Method" (CLRS p.405)
   - asigna una funcion de "energia potencial" (como en fisica)
     - (accounting method = height, potential method = potential energy)
   - costo amortizado de una operacion es su costo mas el cambio de funcion de potencial que resulta de la operacion.
   - es sufficiente de asegurarse que la funcion de potencial es siempre mas grande que su valor inicial para mostrar que el costo total amortizado es una cota superior sobre el costo total de las operaciones.
   - ejemplo:
     - analisis del algoritmo para min y max
     - analisis de MTF

* Ejemplo: Incremento binario

- Incrementar n veces un numero binario de k bits,
- e.g. desde cero hasta $2^k - 1$, con $n = 2^k$.
- costo $\leq kn$ (brute force)
- costo $\leq n + n/2 + ... \leq 2n$ (costo amortizado)
- La tecnica usada aqui es la contabilidad de costos:
  - un flip de 0 a 1 cuesta 2
  - un flip de 1 a 0 cuesta 0
  - cada incremento cuesta 2.
- Analisis
  - ϕ = cantidad de unos en el numero
  - $\phi_0 = 0$
  - $c_i = l + 1$ cuando hay l unos
  - $\Delta\phi_i = -l + 1$
  - $\sum \overline{c_i} = \sum c_i + \phi_n - \phi_0 \geq 2$

* Ejemplo: arreglo dinamico, e.g. java Vector

- considera el tipo "Vector" en Java.
- de tamaño fijo n
- cuando accede a $n + 1$, crea un otro arreglo de tamaño $2n$, y copia todo.
- cual es el costo amortizado si agregando elementos uno a uno?

* Ejemplo: Move-to-Front

- analisis amortizada se puede usar para mostrar que MTF siempre performa a dentro de un factor de 4 de cualquier algoritmo (incluido un algoritmo optimal que conoce la secuencia de busquedas desde el inicio).
- Foncion de potencial es $2× Inv(MTF)

* Ejemplo: Splay Arboles

- el costo amortizado de cada insercion es $O(\lg n)$.

* Ejemplo: Fibonacci Heap

4.1.3. Analisi amortizada de Colas de Prioridades

* Problema: Dado un conjunto (dinamica) de n tareas con valores, elegir y remudar la tarea de valor maxima.
* operaciones basicas:

- M.build({e1,e2,...,2n})
- M.insert(e)
- M.min
- M.deleteMin

* operaciones adicionales ("Addressable priority queues")

- M.insert(e), volviendo un puntero h ("handle") al elemento insertado
- M.remove(h), remudando el elemento especificado para h
- M.decreaseKey(h,k), reduciendo la llave del elemento especificado para h
- M.merge(Q), agregando el heap Q al heap M.

* Soluciones (conocidas o no):

	Linked List	Binary Tree	(Min-)Heap	Fibonacci Heap	Brodal Queue [1]
insert	$O(1)$	$O(\lg n)$	$O(\lg n)$	$O(1)$	$O(1)$
accessmin	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
deletemin	$O(n)$	$O(\lg n)$	$O(\lg n)$	$O(\lg n)^*$	$O(\lg n)$
decreasekey	$O(1)$	$O(\lg n)$	$O(\lg n)$	$O(1)^*$	$O(1)$
delete	$O(n)$	$O(n)$	$O(\lg n)$	$O(\lg n)^*$	$O(\lg n)$
merge	$O(1)$	$O(m \lg(n+m))$	$O(m \lg(n+m))$	$O(1)$	$O(1)$

1. Colas de prioridades binarias ("Binary Heaps")

   * La solucion tradicional

   - un arreglo de n elementos
   - hijos del nodo i en posiciones $2i$ y $2i+1$
   - la valor de cada nodo es mas pequena que las valores de su hijos.

2. Cuidado de no implementar M.build({e1,e2,...,2n}) con n inserciones (sift up $\rightarrow O(n \lg n)$), pero con $n/2$ sift-down ($\rightarrow O(n)$).

3. En M.deleteMin(), algunas variantes de implementacion (despues de cambiar el min con $A[n]$):

   a) dos comparaciones en cada nivel hasta encontrar la posicion final de $A[n]$

- $2 \lg n$ comparaciones en el peor caso
    - $\lg n$ copias

  b) una comparacion en cada nivel para definir un camino de tamaño lg n, y una busqueda binaria para encontrar la posicion final
    - $\lg n + O(\lg \lg n)$ comparaciones en el peor caso
    - $\lg n$ copias en el peor caso

  c) una comparacion en cada nivel para definir un camino de tamaño lg n, y una busqueda **secuencial** up para encontrar la posicion final
    - $2 \lg n$ comparaciones en el peor caso
    - $\lg n$ copias en el peor caso
    - pero en practica y promedio mucho mejor.

4. Porque la complejidad es mejor con un heap que con un arreglo ordenado?
    - Para cada conjunto, hay solamente un arreglo ordenado que puede representarlo, pero muchos heaps posibles: eso da mas flexibilidad para la mantencion dinamica de la estructura de datos.

5. Colas de prioridades con punteros ("Addressable priority queues")
    - Algun que mas flexible que los arreglos ordenados, las colas de prioridades binarias todavia son de estructura muy estricta, por ejemplo para la union de filas. Estructuras mas flexibles consideran un "bosque" de arboles. Ademas, estas estructuras de arboles son implementadas con puntadores (en ves de implementarlos en arreglos).
    - Hay diferentes variantes. Todas tienen en comun los puntos siguientes:

      a) un puntero **minPtr** indica el nodo de valor minima en el bosque, raiz de alguno arbol.

      b) **insert** agregas un nuevo arbol al bosque en tiempo $O(1)$

      c) **deleteMin** remudas el nodo indicado par minPtr, dividiendo su arbol en dos nuevos arboles. Buscamos para el nuevo min y fusionamos algunos arboles (los detalles diferencian las variantes)

      d) **decreaseKey(h,k)** es implementado cortando el arbol al nodo indicado por h, y rebalanceando el arbol cortado.

      e) **delete()** es reducido a **decreaseKey(h,0)** y **deleteMin**

      f) "Pairing Heaps"
        - Malo rendimiento en el peor caso, pero bastante buena en

6. rebalancea los arboles solamente en deleteMin, y solamente con pares de raises (i.e. la cantidad de arboles es reducida por dos a cada deleteMin).

Operacion	Amortizado
Insert(C,x)	$O(1)$
Merge	$O(1)$
ExtractMin	$O(\lg n)$
decreaseKey(h,k)	$\Omega(n \lg n \lg \lg n)$

7. Colas de prioridades binomiales ("Binomial Heaps")

   http://en.wikipedia.org/wiki/Binomial$_h eaphttp : //en.wikipedia.org/wiki/Binomial_heapop136deMelhornySand$

   * Definicion

    - Un **arbol binomial** (de orden k) tiene exactamente k hijos de orden distintos $k-1, k-2, ..., 0$. [Un arbol binomial de orden 0 tiene 0 hijos.]

- Un **bosque binomial** es un conjunto de arboles binomiales de orden **distintas** (i.e. hay cero o uno arboles de cada orden).

8. Para cada arbol T
   - $h(T) \leq lg|T|$
   - $|T| \geq 2^{h(T)}$
9. Para el bosque
   - $\forall n$ hay solamente uno bosque binomial con n nodos.
   - al maxima tiene $\lfloor \lg(n+1) \rfloor$ arboles.
   - la descomposicion del bosque en arboles de orden k corresponde a la descomposicion de n en base de dos.
10. una **cola binomial** es un bosque binomial donde cada nodo almacena una clave, y siempre la clave de un padre es inferior o igual a la clave de un hijo.

    * Operaciones

Operacion	Peor Caso
Merge	$O(\lg n)$
FindMin	$O(\lg n)$
ExtractMin	$O(\lg n)$
Insert(C,x)	$O(\lg n)$
Heapify	$O(n)$
remove(h)	$O(\lg n)$
decreaseKey(h,k)	$O(\lg n)$
merge(Q)	$O(\lg n)$

    * Union

11. Union de dos arboles binomiales de mismo orden:
    - agrega T_2 a T_1 si T_1 tiene la raiz mas pequena.
12. Union de dos bosques binomiales:
    - si hay uno arbol de orden k, es lo de la union
    - si hay dos arboles de orden k, calcula la union en un arbol de orden $k+1$
    - la propagacion es similar a la suma de enteros en binario.
13. Complejidad
    - $O(\lg n)$ en el peor caso
14. agrega un arbol de orden 0 y hace la union si necesitado
15. Complejidad $O(\lg n)$ en el peor caso
16. Puede ser $O(1)$ sin corregir el bosque, que tiene de ser corregido mas tarde, que puede ser en tiempo $O(n)$ peor caso, pero sera $O(\lg n)$ en tiempo amortizado.

    * Minima

17. lei la lista de al maxima $\lfloor \lg(n+1) \rfloor$ raices

18. Complejidad $O(\lg n)$
19. Puede ser $O(1)$ si precalculando un puntero al minima, que tiene de ser corregido (en tiempo $O(\lg n)$) a cada modificacion.

    * DeleteMin

20. encontra el min
21. remuda el min de su arbol (la raiz)
22. reordena su hijos para su orden, en un bosque binomial
23. hace la union con el bosque binomial original, menos el arbol del min
24. complejidad $O(\lg n)$

    * DecreaseKey

25. sigue el camino abajo hasta que la condicion del heap es corregida.
26. cada arbol tiene altura lg n, entonces la complejidad es $O(\lg n)$ en el peor caso.

    * Delete

27. reducido a DecreaseKey+DeleteMin
28. Colas de prioridades de Fibonacci ("Fibonacci Heaps")

    http://en.wikipedia.org/wiki/Fibonacci$_heaphttp://en.wikipedia.org/wiki/Fibonacci_heapopagina135deMelhorn$

    * Diferencia con la cola binomial:

    - relax la estructura de los arboles (heap-forma), pero de forma controlada.
    - el tamaño de un sub-arbol cual raiz tiene k hijos es al maxima $F_k + 2$, donde F_k es el k-esimo numero de Fibonacci.

29. Overview (copidao de http://en.wikipedia.org/wiki/Fibonacci$_heaphttp://en.wikipedia.org/wiki/Fibonacci_heap$)

— — Linked List — Binary Tree — (Min-)Heap — Fibonacci Heap — Brodal Queue — ————-+————-+—————-+—————-+—————-+—————— — insert — $O(1)$ — $O(\lg n)$ — $O(\lg n)$ — $O(1)$ — $O(1)$ — — accessmin — $O(n)$ — $O(1)$ — $O(1)$ — $O(1)$ — $O(1)$ — — deletemin — $O(n)$ — $O(\lg n)$ — $O(\lg n)$ — $O(\lg n)^*$ — $O(\lg n)$ — — decreasekey — $O(1)$ — $O(\lg n)$ — $O(\lg n)$ — $O(1)^*$ — $O(1)$ — — delete — $O(n)$ — $O(n)$ — $O(\lg n)$ — $O(\lg n)^*$ — $O(\lg n)$ — — merge — $O(1)$ — $O(m \lg(n+m))$ — $O(m \lg(n+m))$ — $O(1)$ — $O(1)$ —

1. BONUS Heapsort

    - in place
    - $O(n \lg n)$ con cualquiera de estas variantes.

4.1.4. Árbol biselado ("Splay Tree")

APUNTES

4.2. Analisi adaptativa

4.2.1. MATERIAL A LEER

- Árbol 2-3 http://es.wikipedia.org/wiki/

- Finger tree http://en.wikipedia.org/wiki/Finger$_treehttp://en.wikipedia.org/wiki/Finger_tree$

4.2.2. Analisi en el peor caso: a dentro de que?

- peor caso a dentro de las instancias de tamaño fijo
  - el tamaño puede ser multidimensional (e.g. grafos)
- peor caso a dentro de las instancias de tamaño de resultado fijado
  - cantidad infinita
- peor caso a dentro de las instancias de **dificultad** fijada
  - cantidad infinita
- peor caso a dentro de las instancias de **tamaño fijo y dificultad** fijada

4.2.3. Busqueda Doblada: $1 + 2\lceil \lg p \rceil$ comparaciones

El algoritmo de busqueda doblada

- encuentra en $1 + \lceil \lg p \rceil$ comparaciones un intervalo de tamaño $p/2$ que contiene x,
- encuentra p en $1+\lceil \lg p \rceil$ comparaciones adicionales (usando la segunda variante de la busqueda binaria),
- por un total de $1 + 2\lceil \lg p \rceil$ comparaciones

4.2.4. Finger Search Tree: la busqueda doblada de los arboles de busqueda

APUNTES

- Estructura de datos
- algorimo de busqueda
- analisis: busqueda en $O(\lg p)$

4.2.5. Algoritmo de Ordenamiento adaptivos basicos

Merge Sort Adaptivo: Runs

Local Insertion Sort: Inv

Añother Insertion Sort: REM

4.2.6. Computacion de la Union

4.2.7. Computacion de la Interseccion

4.3. Algoritmos en linea

4.3.1. List Accessing

REFERENCIA: Capitulo 1 en “Online Computation and Competitive Analysis”, de Allan Borodin y Ran El-Yaniv

- “List Accessing”
  - Considera la secuencia de busqueda de tamaño n, en un diccionario de tamaño σ: “1,1,1,1,1,1,2,2,2,3,3,3,4,4,4,5,5
  - Cual es el rendimiento de una estructura de diccionario **estatica** (tal que AVL) en este secuencia?

        - $n \lg \sigma$
    - Se puede mejorar?
        - si, utilizando la **localidad** de las consultas, en **estructuras de datos dinamicas**.

- Soluciones

    1. MTF ("Move To Front"):
        - pone las llaves en un arreglo desordenado
        - buscas secuencialmente en el arreglo
        - muda la llave encontrada en frente
    2. TRANS ("Transpose"):
        - pone las llaves en un arreglo desordenado
        - buscas secuencialmente en el arreglo
        - muda la llave encontrada de una posicion mas cerca del frente
    3. FC ("Frequency Count"):
        - mantiene un contador para la frecuencia de cada elemento
        - mantiene la lista ordenada para frecuencia decreciente.
    4. Splay Trees (y otras estructuras con propiedades de localidad) (http://www.dcc.uchile.cl/ cc30a/apuntes/Diccio

- Estos son "Algoritmos en Linea"

    - algoritmo de optimizacion
    - que conoce solamente una parte de la entrada al tiempo t.
    - se compara a la competitividad con el algoritmo offline que conoce toda la instancia.
    - Como se puede medir su complejidad?
        - cada algoritmo ejecuta $O(n)$ comparaciones para cada busqueda en el peor caso!!!!
        - tiene de considerar instancies "faciles" y "dificiles"
        - una medida de dificultad
        - e.g. el rendimiento del *mejor algoritmo "offline"*

- Competitive Analysis: instancias "dificiles" o "faciles"

    - Las estructuras de datos dinamicas pueden aprovechar de secuencias "faciles" de consultas: eso se llama "online".
    - pero para muchos problemas online, todas las heuristicas se comportan de la misma manera en el peor caso.
    - Por eso se identifica una medida de dificultad de las instancias, y se comparan los rendimientos de los algoritmos sobre instancias que tienen una valor fijada de este medida de dificultad.
    - Tradicionalmente, esta medida de dificultad es el rendimiento del mejor algoritmo "offline": eso se llama **competitive analysis**, resultando en el **competitive ratio**, el ratio entre la complejidad del algoritmo ONLINE y la complejidad del mejor algoritmo OFFLINE.
        - por ejemplo, veamos que MTF tiene un competitive ratio de 2
    - Pero todavia hay algoritmos con performancia practicas muy distintas que tienen el mismo competitive ratio. Por eso se introduce otras medidas de dificultadas mas sofisticadas, y mas especialidades en cada problema.

- Competitividad

  * Optimizacion/aproximacion

- A es $k(n)$ **competitiva** para un problema de **minimizacion** si

$$\exists b \forall n, x,\ |x| = n,\ C_A(x) - k(n) C_{OPT}(x) \leq b$$

- A es **k(n) competitiva** para un problema de **maximizacion** si

$$\exists b, \forall n, x,\ |x| = n,\ C_{OPT}(x) - k(n) C_A(x) \leq b$$

- an algoritmo en linea es **c-competitiva** si

$$\exists \alpha, \forall I ALG(I) \leq cOPT(I) + \alpha$$

- an algoritmo en linea es **estrictamente c-competitiva** si

$$\forall I ALG(I) \leq cOPT(I)$$

- Sleator-Tarjan sobre MTF
  - costo de una busqueda negativa (la llave NO esta en el diccionario)
    - σ
  - costo de una busqueda positiva (la llave esta en el diccionario)
    - la posicion de la llave, no mas que σ
    - en promedio para una distribucion de probabilidad fijada:
      - $MTF \leq 2OPT$,
    - Prueba: (from my notes in my CS240 slides)
      How does MTF compare to the optimal ordering?
      - Assume that:
      - the keys $k_1, \ldots, k_n$ have probabilities $p_1 \geq p_2 \geq \ldots \geq p_n \geq 0$
      - the list is used sufficiently to reach a steady state.
      - Then:

$$C_{MTF} < 2 \cdot C_{OPT}$$

      - Proof:
      - $C_{OPT} = \sum_{j=1}^{n} j p_j$
      - $C_{MTF} = \sum_{j=1}^{n} p_j($ cost of finding $k_j)$
      - $C_{MTF} = \sum_{j=1}^{n} p_j(1 +$ number of keys before $k_j)$
      - To compute the average number of keys before k_j:

$$\Pr[\ k_i \text{ before } k_j] = \frac{p_i}{p_i + p_j}$$

$$E(\text{ number of keys before } k_j) = \sum_{i \neq j} \frac{p_i}{p_i + p_j}$$

      - k_i is before k_j if and only if k_i was accessed more recently than k_j.
      - Consider the last time either k_i or k_j was looked up. What is the probability that it was k_i?

$$P(k_i \text{ before } k_j) = P(k_i \text{ chosen } \mid k_i \text{ or } k_j \text{ chosen })$$

$$P(k_i \text{ before } k_j) = \frac{P(k_i \text{ chosen })}{P(k_i \text{ or } k_j \text{ chosen })}$$

$$P(k_i \text{ before } k_j) = \frac{p_i}{p_i + p_j}$$

◇ Therefore,

◇ Joining both previous formulas:

$$C_{MTF} = \sum_{j=1}^{n} p_j (1 + \sum_{i \neq j} \frac{p_i}{p_i + p_j})$$

◇ reordering the terms:

$$C_{MTF} = 1 + 2 \sum_{j=1}^{n} \sum_{i<j} \frac{p_i p_j}{p_i + p_j}$$

◇ Because $\frac{p_i}{p_i+p_j} \leq 1$:

$$C_{MTF} \leq 1 + 2 \sum_{j=1}^{n} p_j (\sum_{i<j} 1)$$

$$C_{MTF} = 1 + 2 \sum_{j=1}^{n} p_j (j-1)$$

$$C_{MTF} = 1 + 2C_{OPT} + 2 \sum_{j=1}^{n} (-p_j)$$

◇ Because $\sum_{j=1}^{n} (p_j) = 1$:

$$C_{MTF} = 2C_{OPT} - 1$$

BONUS Applicaciones a la compression de textos

* Bentley, Sleator, Tarjan and Wei proponieron de comprimir un texto utilizando una lista dinamica, donde el codigo para un simbolo es la posicion del simbolo en la lista.

- Experimentalmente, se compara a Huffman:
  - a veces mucho mejor
  - nunca mucho peor.

- Experimentalmente, 6 % mejor que GZip, que es enorme!

4.3.2. Paginamiento Deterministico

REFERENCIA: Capitulo 2 en “Online Computation and Competitive Analysis”, de Allan Borodin y Ran El-Yaniv

1. Paginamiento
   - Definicion:
     - elegir cual paginas guardar en memoria, dado
       - una secuencia online de n consultas para paginas, y
       - un cache de k paginas.
   - Politicas:
     - LRU (Least Recently Used)

        - CLOCK (1bit LRU)
        - FIFO (First In First Out)
        - LFU (Least Frequently Used)
        - LIFO = MRU (Most Recently Used)
        - FWF (Flush When Full)
        - LFD (Offline, Longuest Forward Distance)
    - Ustedes tienen una idea de cuales son las peores/mejores?

2. Relacion con "List Accessing"
    a) Cada "List accessing" algoritmo corresponde a un algoritmo de paginamiento:
        - cada miss, borra el ultimo elemento de la lista y "inserta" el nuevo elemento.
    b) No hay una reduccion tan clara en la otra direccion.

3. Offline analysis
    - LFD performa O(n/k) misses
    - Cualquier algoritmo Offline performa Omega(n/k) en el peor caso.

4. Online analisis: resultados basicos
    a) $\forall A$ online, hay una entrada con n fallas.
        - Estrategia de adversario.
    b) No algoritmo online puede ser mejor que k competitivo.
        - Obvio, comparando con LFD.
    c) MRU=LIFO NO es competitivo
        - Considera S= $\mathrm{p}_1,\mathrm{p}_2,\ldots,\mathrm{p}_k,\mathrm{p}_{k+1},\mathrm{p}_k,\mathrm{p}_{k+1},\mathrm{p}_k,\mathrm{p}_{k+1},\mathrm{p}_k,\ldots$
        - despues las k primeras consultas, MRU va a tener un miss cada consulta, cuando LFD nunca mas.
    d) LFU no es competitivo
        - Considera $l > 0$ y $S = p_1^l, p_2^l, \ldots, p_{k-1}^l, (p_k, p_{k+1})^{l-1}$
        - Despues de las $(k-1)l$ primeras consultas, LFU va a tener un miss cada consulta, cuando LFD solamente dos.
    e) Que tal de FWF?
        - MRU=LIFO es un poco estupido, su mala rendimiento no es una sorpresa.

- FWF es un algoritmo muy ingenuo tambien, pero vamos a ver que no tiene un rendimiento tan mal “en teoria”.

5. BONUS: Competive Analysis: Algoritmos a Marcas

   a) k-fases particiones

   Para cada secuencia S, partitionala en secuencias $S_1, \ldots, S_\delta$ tal que

   - $S_0 = \emptyset$
   - S_i es la secuencia Maxima despues de S_{i-1} que contiene al maximum k consultas distintas.
   - Llamamos “fase i.el tiempo que el algoritmo considera elementos de la subsecuencia S_i.
   - Nota que eso es independiente del algoritmo considerado.

   b) Algoritmo con marcas

   - agrega a cada pagina de memoria lenta un bit de marca.
   - al inicio de cada fase, remuda las marcas de cada pagina en memoria.
   - a dentro de una fase, marca una pagina la primera vez que es consultada.
   - un algoritmo a marca (“marking algorithm”) es un algoritmo que nunca remuda una pagina marcada de su cache.

   c) Un algoritmo con marcas es k-competitiva

   - En cada fase,
     - un algoritmo ONLINE con marcas performa al maximum k miss.
     - Un algoritmo OFFLINE (e.g. LFD) performa al minimum 1 miss.
   - QED

   d) LRU, CLOCK y FWF son algoritmos con marcas

   e) LRU, CLOCK y FWF tienen un ratio competitivo OPTIMO

6. Mas resultados:

   a) La analisis se puede generalizar al caso donde el algoritmo offline tiene h paginas, y el algoritmo online tiene $k \geq h$ paginas.

   - Cada algoritmo **con marcas** es $\frac{k}{k-h+1}$-competitiva.

   b) Definicion de algoritmos (conservadores) da resultado similar para FIFO (que no es con marcas pero es conservador).

   c) En practica, sabemos que LRU es mucho mejor que FWF (for instancia). Habia mucha investigacion para intentar de mejorar la analisis por 20 años, ahora parece que hay una analisis que explica la mejor rendimiento de LRU sobre FWF, y de variantes de LRU que pueden saber x pasos en el futuro (Reza Dorrigiv y Alex Lopez-Ortiz).

4.3.3. BONUS: "Ski Renting"

http://en.wikipedia.org/wiki/Ski$_rental_problemhttp$: $//en.wikipedia.org/wiki/Ski\_rental\_problem$

4.4. MAYB Complejidad Parametrizada

4.5. PREGUNTAS [0/0] :PREGUNTAS:

4.5.1. Analisis Amortizada: arreglo dinamico (Part 1)

:CONTEXT: :END:

Queremos implementar una pila ("stack") en un arreglo. Iniciamos con un arreglo de tamaño $s = 1$, y cuando se llena, creamos un arreglo mas grande, copiamos todo en en nuevo arreglo y sigamos.

Cual es el costo amortizado de una insercion si el nuevo arreglo es de tamaño $n + 1$?

1. $O(1)$
2. $O(\lg n)$
3. $O(n)$
4. $O(n^2)$
5. otra respuesta

4.5.2. Analisis Amortizada: arreglo dinamico (Part 2)

:CONTEXT: Queremos implementar una pila ("stack") en un arreglo. Iniciamos con un arreglo de tamaño $s = 1$, y cuando se llena, creamos un arreglo mas grande, copiamos todo en en nuevo arreglo y sigamos. :END:

Cual es el costo amortizado de una insercion si el nuevo arreglo es de tamaño $2n$?

1. $O(1)$
2. $O(\lg n)$
3. $O(n)$
4. $O(n^2)$
5. otra respuesta

4.5.3. Analisis Amortizada: arreglo dinamico (Part 3)

:CONTEXT: Queremos implementar una pila ("stack") en un arreglo. Iniciamos con un arreglo de tamaño $s = 1$, y cuando se llena, creamos un arreglo mas grande, copiamos todo en en nuevo arreglo y sigamos. :END:

Cual es el costo amortizado de una insercion si el nuevo arreglo es de tamaño $4n$?

1. menos que 2
2. 2
3. entre 2 y 3
4. 3
5. mas que 3

4.5.4. Analisis Amortizada: arreglo dinamico (Part 4)

:CONTEXT: Queremos implementar una pila ("stack") en un arreglo. Iniciamos con un arreglo de tamaño $s = 1$, y cuando se llena, creamos un arreglo mas grande, copiamos todo en en nuevo arreglo y sigamos. :END:

Cual es el costo amortizado de una insercion si el nuevo arreglo es de tamaño n^2 (y el primero arreglo de tamaño 2)?

1. $O(1)$
2. $O(\lg n)$
3. $O(n)$
4. $O(n^2)$
5. otra respuesta

5. RESUMEN Unidad 3

* Resultados de Aprendisajes de la Unidad

- Comprender las tecnicas de algoritmos de
  - costo amortizado,
  - uso de finitud, y
  - algorimos competitivos
- Ser capaz de disenar y analzar algoritmos y estructuras de datos basados en estos principios.
- conocer algunos casos de estudio relevantes

* Principales casos de estudio:

- estructuras para union-find,
- colas binomiales
- splay trees,
- busqueda por interpolacion
- radix sort
- arboles de van Emde Boas
- arboles de sufijos
- tecnica de los cuatro rusos,
- paginamiento
- busqueda no acotada (unbounded search, doubling search)