estudios de postgrado
en computación

Análisis y Diseño de Algoritmos (AyDA)

Isabel Besembel Carrera

COLAS POR PRIORIDAD, AUMENTO DE ESTRUCTURAS DE DATOS

Transformación clave-dirección o Hashing

- Diccionario:
  - Conjunto dinámico de elementos
  - Operaciones: insertar, eliminar y consultar
  - Implementación: tablas hash
    - W(n)=O(n) , T(n)=1 con algunas restricciones
    - Universo de claves (U) pequeño, donde las claves son primarias, conjunto de claves (K). Función hash (h)
    - **h**: U$\rightarrow${0, 1, ..., m-1}
    - Problemas: U es normalmente muy grande y el número de claves en uso generalmente no es igual a |U|.

Direccionamiento directo

- Tabla hash (T)

buscarTabla(TipoClave: cl)

regrese T(cl)

insTabla(TipoEleTabla: e)

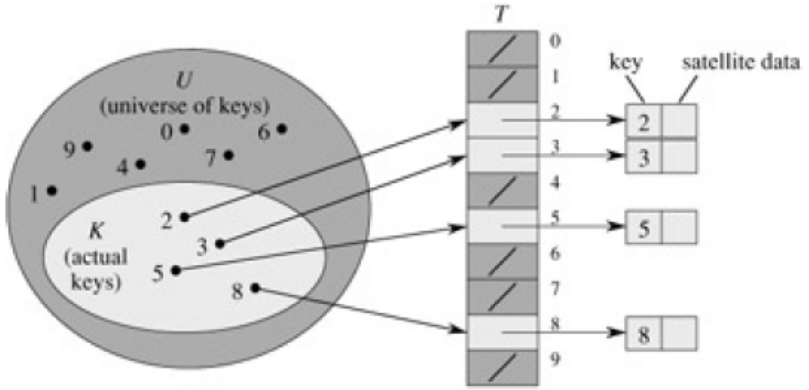
T(e.Clave( )) = e

eliTabla(TipoClave: cl)

T(cl) = Nulo

- ✓ Todas en O(1)

Los requerimientos de espacio se pueden reducir a |K| y la tabla sigue manteniendo su complejidad a O(1) en promedio, más no en el peor de los casos

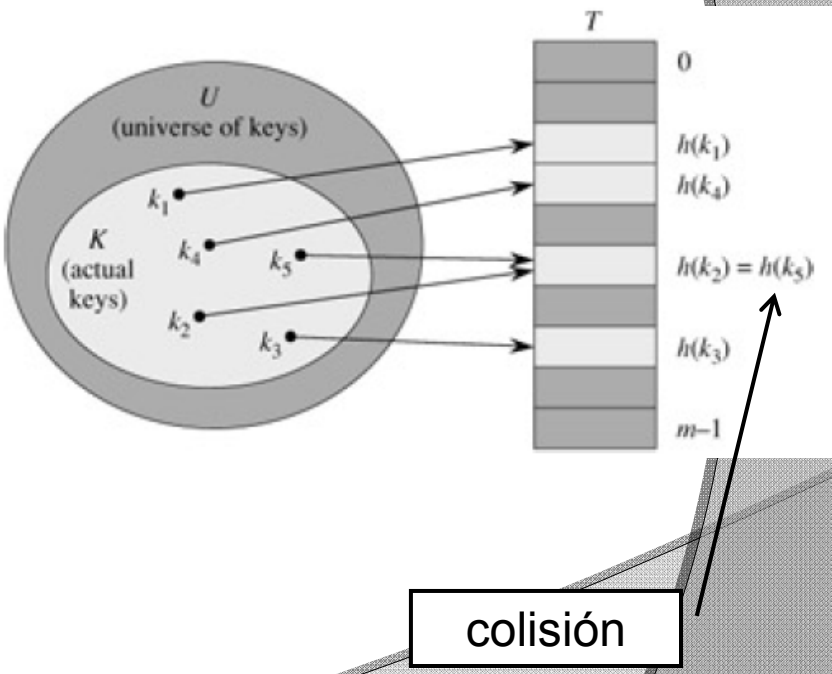


0	/
1	/
2	resto
3	resto
4	/
5	resto
6	/
7	/
8	resto
9	/

T

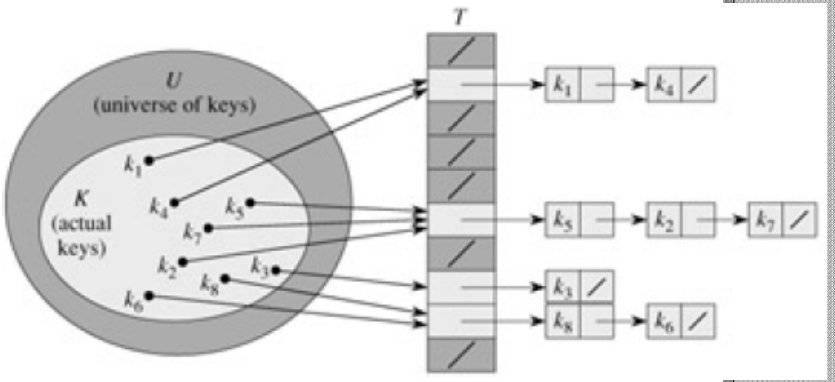
Tablas hash

- ranuras de la tabla T(0,..., m-1)
- |U| > m
- Problema: dos claves diferentes pueden ser transformadas con **h** y obtenerse la misma ranura, **colisión**
- **h** debe ser determinista para que siempre dé el mismo valor al ser aplicada a la misma clave



Colisiones

- Aunque una buena función **h** minimiza las colisiones, es imposible no tenerlas
- Resolución de colisiones:
  - Encadenamiento
  - Direccionamiento abierto
    - Prueba lineal o cuadrática
    - Doble hashing



Hashing universal

- Selección aleatoria de una función hash entre un conjunto de funciones hash bien definidas, lo cual hace que ella sea independiente de las claves que serán almacenadas
- Sea H una colección finita de funciones hash de U al rango {0, 1, ..., m-1}
- H es una colección universal si para cada par de claves distintas x, y $\in$ U, el número de funciones hash h $\in H$ para las cuales h(x) = h(y) es exactamente $|H| / m$

Es decir, con una h de H el chance de una colisión entre x e y cuando x $\neq$ y, es exactamente 1/m

Dispersión universal

- Teorema 1: Si h se selecciona de H y se usa para dispersar **n** claves en una tabla de tamaño **m**, donde **n** ≤ **m**, el número esperado de colisiones para una clave particular **x** es menor que 1.
- Teorema 2: La clase H definida por:

$$h_a(x) = \sum_{i=0}^{r} a_i x_i \bmod m \quad \text{entonces} \quad H = \bigcup_a \{h_a\} \text{ tiene } m^{r+1} \text{ miembros.}$$

es una clase universal de funciones hash

Tablas de dispersión

Direccionamiento abierto

- Todos los elementos se almacenan dentro de la tabla.
- Se selecciona una secuencia de prueba. [h (c, 0), h (c,1),..., h (c, m-1)]

insTabHash(TipoClave: c): Cardinal		
1	i = 0	-i. Entero. Seleccionador de la secuencia de prueba. -j. Entero. Posición en la tabla, subíndice.
2	[j = h (c, i) Si (T(j) = ClaveNoDef) entonces T(j) = c regrese j sino i = i + 1 fsi] (i = m)	
3	despliegue “error, desborde!”	

Normalmente no se eliminan claves, pues habría que marcar la ranura con un valor especial y rehacer los algoritmos

busTabHash(TipoClave: c): Cardinal		
1	busTabHash, i, j = CardinalNoDef, 0, h (c, i)	-i. Entero. Seleccionador de la secuencia de prueba. -j. Entero. Posición en la tabla, subíndice.
2	(T(j) $\neq$ ClaveNoDef $\wedge$ i $<$ m) [i, j = i +1, h (c, i)]	
3	Si (T(j) = c) entonces busTabHash = j	
4	regrese	

Direccionamiento abierto

- Análisis: $\alpha = n/m$ $\quad n \leq m \Rightarrow \alpha \leq 1$ con hashing uniforme, por lo que cualquier secuencia de prueba es igualmente probable de usar.
- Teorema 3: Dada una tabla hash con direccionamiento abierto y un $\alpha < 1$, el número esperado de pruebas para una búsqueda infructuosa es a lo más $1 / (1 - \alpha)$ asumiendo hashing uniforme.
- Si α es una constante entonces O(1)

Ejemplo:

Una tabla hash medio llena $\alpha = 0.5 \Rightarrow 1 / (1 – 0.5) = 2$ búsquedas infructuosas.

Si está 90% llena $\alpha = 0.9 \Rightarrow 1 / (1 – 0.9) = 10$

Direccionamiento abierto

- Teorema 4: Dada una tabla hash con direccionamiento abierto y un $\alpha < 1$, el número esperado de pruebas para una búsqueda exitosa es a lo más $(1 / \alpha) \ln (1 / (1 - \alpha)) + (1 / \alpha)$ asumiendo hashing uniforme e igualdad de probabilidades de búsqueda para las claves.

Ejemplo:

Una tabla hash medio llena $\alpha = 0.5 \Rightarrow 3.387$

Si está 90% llena $\alpha = 0.9 \Rightarrow 3.67$

Dispersión uniforme

Lineal y cuadrática

- Prueba lineal: $h(c, i) = (h'(c) + i) \bmod m$
  - donde $h'(c)$ es una función hash auxiliar
- Problema: agrupamiento primario lo cual incrementa el tiempo promedio de búsqueda

- Prueba cuadrática: $h(c, i) = (h'(c) + c_1 i + c_2 i^2) \bmod m$ con $c_1, c_2 \neq 0$
  - mejor que el anterior, pero hay que escoger c_1, c_2 y m para hacer uso de toda la tabla.
- Problema: agrupamiento segundario de las claves

Doble hash

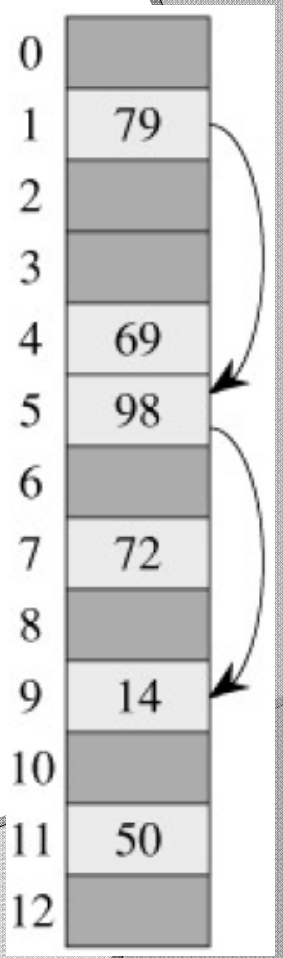
- Doble hashing: $h(c, i) = (h_1(c) + i\, h_2(c)) \bmod m$
- Escogencia de h_1 y h_2:
  - m una potencia de 2 y h_2 que siempre regrese un número par.
  - m un número primo y h_2 que siempre regrese un número positivo $< m$

Ejemplo: $h_1(c) = c \bmod m$ y

$h_2(c) = 1 + (c \bmod m')$

con $m' < m$, $m' = m-1$ o $m-2$

- Se usan $\theta(m^2)$ secuencias de prueba en vez de $\theta(m)$. Es de los mejores métodos.
- El rendimiento está muy cercano al esquema ideal de un hashing uniforme



Dispersión estática

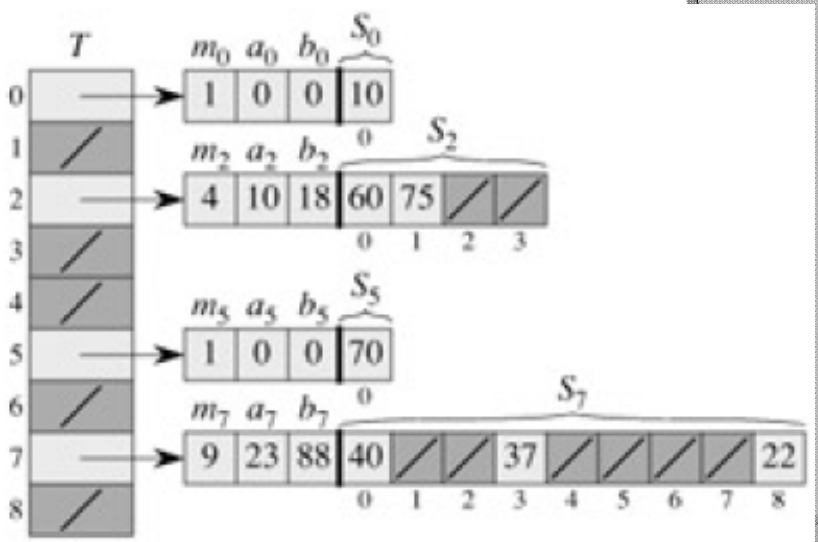
Hashing perfecto

- Cuando K no cambia (K estático). W(n)=O(1)
  Ejemplo: archivos en un CDROM
- Dos niveles de esquema de dispersión con hashing perfecto en cada nivel
- Primer nivel con hashing encadenado
- Segundo nivel con otra tabla hash S_j con una h_j asociada
- Para garantizar la ausencia de colisiones m_j (tamaño de S_j) = n_j^2 (nj número de claves dispersas en la ranura j), con buenas h el espacio total utilizado O(n)
- Teorema: sea m=|T| y n=|K|, si m=n con $h \in H$ y se coloca el tamaño de la segunda tabla hash $m_j=n_j^2$, j=0,1,…,m-1, entonces la probabilidad que el espacio total usado en la segunda tabla hash exceda 4n es menor que 1/2

Dispersión perfecta

- K={10,22,37,40,60,70, 75}
- Primer nivel h(k)=((ak+b) mod p) mod m, a=3, b= 42, p=101 y m=9
- Segundo nivel $h_j(k)$= $((a_j k+b_j) \bmod p) \bmod m_j$

h(75)=2 $h_2(75)=1$



Montículos

Colas por prioridad

- Contienen un conjunto de entradas formadas por una clave (segundaria) y un valor
- Utilizadas en: sistemas operativos y simulación
- Las claves se mantienen ordenadas
- Tipos:
  - Por máxima prioridad
  - Por mínima prioridad
- Implementación con montículos binarios
- Un montículo (*heap*) binario es un árbol binario semi completo implantado según el método secuencial donde el elemento de la posición **i** tiene como: padre el $A_{\lfloor i/2 \rfloor}$
  hijo izquierdo A_{2i}
  hijo derecho A_{2i+1}

Especificación

13/10/98 Especificación Montículo[Entrada]		
1	Sintáctica: creaMontículo()→Montículo, insEntMontículo(Montículo,Entrada)→Montículo, eliMaxMontículo(Montículo) →Entrada, max(Montículo) →Entrada, vacíoMontículo(Montículo) →Lógico, destMontículo(Montículo) →.	-creaMontículo(): Crea un Montículo vacío. -insEntMontículo(): Ingresa una nueva entrada en el Montículo. -eliMaxMontículo(): Elimina y regrese la entrada con valor mayor del Montículo, si existe. -max(): Regresa la entrada con mayor clave del Montículo. -vacíoMontículo(): Regresa verdadero si el Montículo está vacío. -destMontículo(): Destruye el Montículo.
2	Declaraciones Entrada: e, {EntradaNoDef}	
3	Semántica: eliMaxMontículo(creaMontículo()) = {EntradaNoDef} max(creaMontículo()) = {EntradaNoDef} max(insEntMontículo(creaMontículo(), e)) = e vacíoMontículo(creaMontículo()) = Verdadero vacíoMontículo(insEntMontículo(creaMontículo(), e)) = Falso	

Complejidad de la clase Montículo O(lg n) = Cola por prioridades

Implementación de montículo binario

14/10/98	Montículo[Entrada] Clases: Entero, Lógico, Entrada, Arreglo[max]De [Entrada]	
1	Superclase: Ninguna	-n: Entero+. Número actual de elementos en el Montículo.
2	Estructura: privado: max: Entero+ = 100 n: Entero = 0 m: Arreglo[max] De [Entrada]	-max: Entero+.Número máximo de entradas en el Montículo. -m:Arreglo[100] De [Entrada]. Arreglo de entradas para implementar el Montículo. -Montículo(). Constructor. Crea un Montículo vacío. O(1)
3	Operaciones: público: Montículo() insEntMontículo(Entrada: e) eliMaxMontículo(): Entrada max(): Entrada vacíoMontículo(): Lógico numEle(): Entero privado: mont(Entero) hacerMontículo(Entero, Arreglo)	-insEntMontículo(). Transformador. Inserta una nueva entrada en el Montículo. O(lg n). -eliMaxMontículo(). Transformador. Elimina la mayor entrada del Montículo, regresandola si existe. O(lg n). -max(). Observador. Regresa la entrada con mayor clave que se encuentra en el Montículo. O(1). -vacíoMontículo(). Observador. Regresa verdadero si el Montículo está vacío, de lo contrario regresa falso. O(1). -numEle(). Observador. Regresa el número actual de entradas. O(1). -mont(). Transformador. Actualiza el montículo luego de eliminar.O(h) -hacerMontículo(). Llena un montículo con los elementos del arreglo.

Montículos o heaps

14/10/98

insEntMontículo(Entrada: e)

{pre: max = 100 $\wedge$ 0 $\leq$ n $\leq$ max $\wedge$ e $\neq$ {EntradaNoDef}} { pos: max = 100 $\wedge$ 0 $\leq$ n $\leq$ max $\wedge$ m' = m + {e}}

1 2 3 4	i = n + 1 (i > 1 $\wedge$ $m_{\lfloor i/2 \rfloor}$ < e) [m_i, i = $m_{\lfloor i/2 \rfloor}$, $\lfloor i/2 \rfloor$] m_i , n = e, n+1 regrese	-i: Entero. Subíndice para recorrer el montículo. -n, m. Definidos en la clase Montículo.
1 2	mon1= (100, 0), e = 4 $\Rightarrow$ (100, 1, 4) mon1 = (100, 3, 2, 1, 1), e = 4 $\Rightarrow$ (100, 4, 4, 2, 1, 1)	Primera inserción en mon1[Entero]. Regresa el número de entradas en mon1[Entero].

14/10/98

eliMaxMontículo(): Entrada

{pre: max = 100 $\wedge$ 0 $\leq$ n $\leq$ max } { pos: max = 100 $\wedge$ 0 $\leq$ n $\leq$ max $\wedge$ m' = m - {e}}

1 2	Si (n < 1) entonces despliegue "Error, montículo vacío" sino mayor, m_1, n = m_1, m_n, n – 1 mont (1) fsi regrese mayor	-mayor: Entrada. Entrada mayor en el montículo. -mont(). Función recursiva que actualiza el montículo luego de la eliminación del mayor. -n, m. Definidos en la clase Montículo.
1 2	mon1= (100, 0) $\Rightarrow$ Error, montículo vacío mon1 = (100, 4, 4, 2, 1, 1) $\Rightarrow$ (100, 3, 2, 1, 1), mayor = 4	Eliminación en vacío. Regresa la entrada mayor luego de eliminarla.

Montículos o heaps

14/10/98	max(): Entrada	
{pre: max = 100 $\wedge$ n $\geq$ 0}		{ pos: max = 100 $\wedge$ n $\geq$ 0}
1	regresa m(1)	-m: Definido en la clase Montículo.
1	mon1= (100, 0, EntradaNoDef)$\Rightarrow$ EntradaNoDef	La variable mon1 está vacía
2	mon1= (100, 3, 2, 1, 1) $\Rightarrow$ 2	Regresa la entrada mayor de mon1.

14/10/98	mont(Entero: i)	
{pre: max = 100 $\wedge$ 0 $\leq$ n, i $\leq$ max }		{ pos: max = 100 $\wedge$ 0 $\leq$ n $\leq$ max }
1	izq, der = 2 i, 2 i+1	-izq, der, mayor: Entero. Subíndices para indicar el hijo izquierdo, el hijo derecho y el mayor actual (del nodo i) en el montículo. -x. Entrada. Variable auxiliar para el intercambio. -n, m. Definidos en la clase Montículo.
2	mayor = Si (izq $\leq$ n $\wedge$ $m_{izq} > m_{der}$) entonces izq sino der fsi	
3	mayor = Si (der $\leq$ n $\wedge$ $m_{der} > m_{izq}$) entonces der fsi	
4	Si (mayor $\leq$ n $\wedge$ mayor $\neq$ i) entonces x, m_i, $m_{mayor} = m_i$, m_{mayor} , x mont (mayor) fsi	
5	regrese	
1	mon1 = (100, 3, 1, 2, 1), i = 1 $\Rightarrow$ (100, 3, 2, 1, 1)	Actualización de mon1[Entero] luego de eliminar 4.

Montículos o heaps

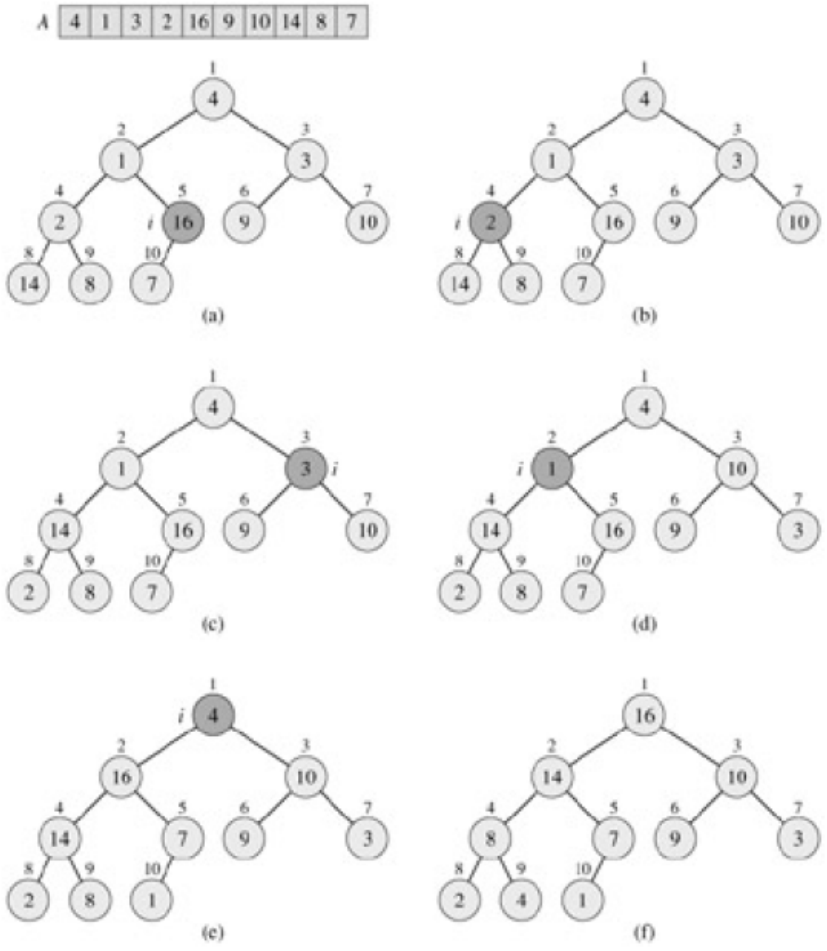
14/10/98 hacerMontículo(Entero: k, Arreglo[k]De [Entrada]: A) {pre: $k \geq 0$} {pos: max = 100 $\wedge$ n = k}		
1	Si (k > 0) entonces [insEntMontículo(A_i)] i = 1, k fsi	-insEntMontículo(): Definida en la clase Montículo.
1	mon1 = (100, 0), k = 3, A = (8,10,20) $\Rightarrow$ mon1 = (100,3, 20, 8, 10)	Llena el montículo asociado a la variable mon1[Entero], el arreglo queda intacto.

14/10/98 vacíoMontículo(): Lógico {pre: max = 100 $\wedge$ $n \geq 0$} { pos: max = 100 $\wedge$ $n \geq 0$}		
1	regresa (n = 0)	-n: Definido en la clase Montículo.
1	mon1.vacíoMontículo$\Rightarrow$ Verdadero	La variable mon1 está vacía

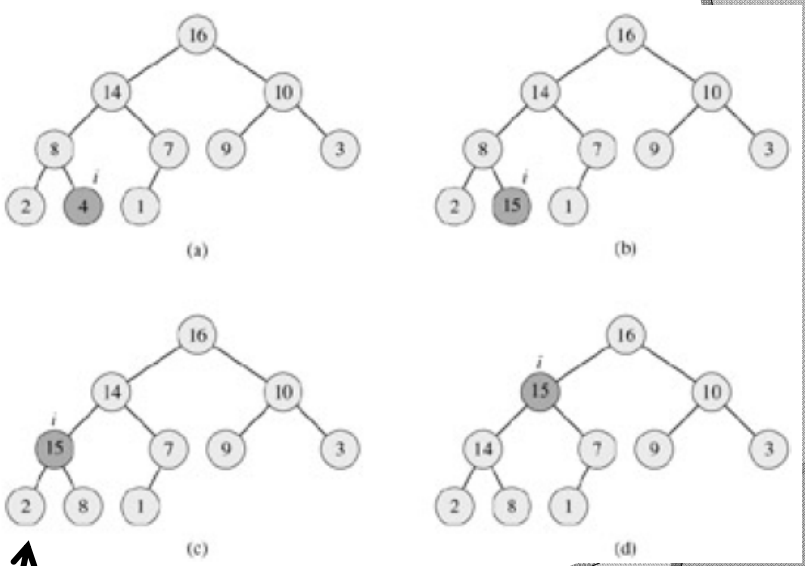
14/10/98 Montículo() {pos: max = 100 $\wedge$ n = 0}		
1	max, n = 100, 0	-max, n: Definidos en la clase Montículo.
1	Montículo[Entero] mon1 $\Rightarrow$ mon1.max = 100, mon1.n = 0	Crea la variable mon1[Entero]

Montículos o heaps

hacerMonticulo



El T(n) de **mont** sigue
$T(n) \leq T(2n/3) + \theta(1)$
que cae en el caso 2 del teorema maestro, $T(n) = O(\lg n)$



Implementación libro texto
Algoritmo HEAP-INCREASE_KEY

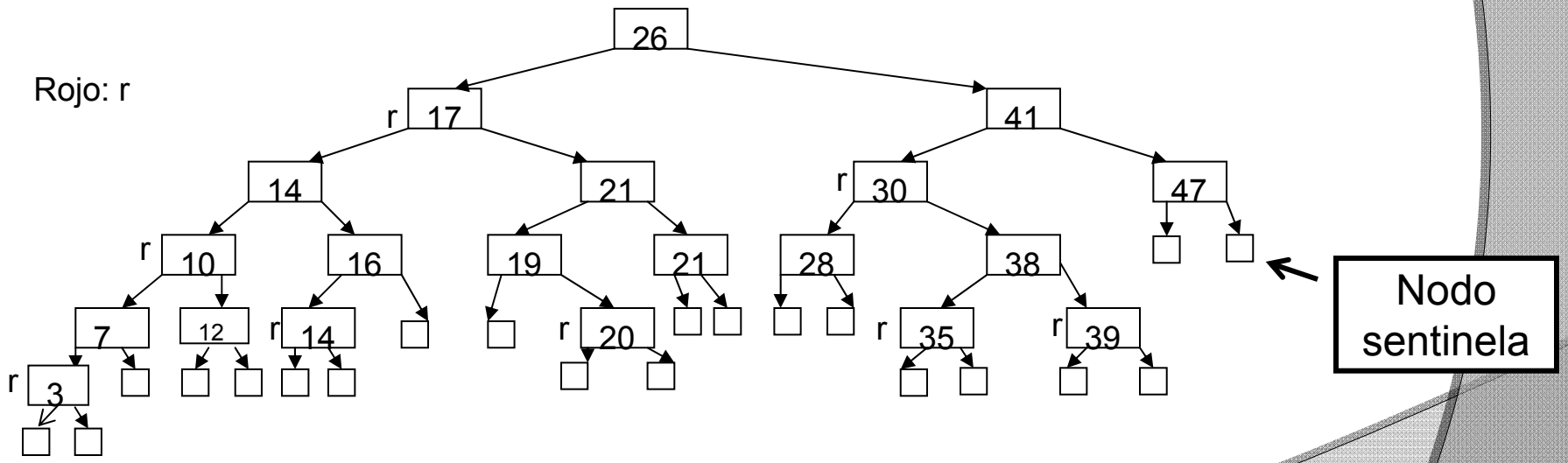
Aumentando las estructuras de datos

- Diseño de nuevas estructuras de datos vs modificación de estructuras de datos existentes
- Modificación de los árboles roji-negros
  - Son árboles semi balanceados que garantizan una complejidad logarítmica para las operaciones sobre conjuntos dinámicos
  - Es un árbol binario de búsqueda con un campo adicional dentro del nodo para indicar el **color** del mismo, que puede ser: rojo o negro
  - Definición:
    1. Cada nodo es negro o rojo
    2. Cada hoja es negra
    3. Si un nodo es rojo, sus hijos son negros
    4. Cada camino desde un nodo rama a una de sus hojas contiene el mismo número de nodos negros

Árboles roji-negros

- Altura negra de un nodo: es el número de nodos negros en el camino desde él hasta una de sus hojas
- Formato de un nodo:

color	clave	p	izq	der



Un árbol rojinegro con **n** nodos ramas tiene una altura de al menos 2 lg (n + 1).
Para mantener semi-balanceada la estructura se utilizan rotaciones a la izquierda y a la derecha

Árboles roji-negros

19/10/98

rotarIzq(ApuntadorA NodoABin: x)

{pre: x ≠ NodoABinNoDef ∧ x→Der( ) ≠ Nulo} { pos: altura (A') ≤ altura (A) }

1	y = x→Der( )	-y. ApuntadorA NodoABin. Hijo derecho de x.
2	x→Der(y→Izq( ))	-Izq( ), Der( ), P( ). Definidos en la clase NodoABin.
3	Si (y→Izq( ) ≠ Nulo) entonces y→Izq( ) →P(x) fsi	-raíz. Definido en la clase ArbBin.
4	y→P(x→P( ))	
5	Si (x→P( ) = Nulo) entonces raíz = y sino Si (x = x→P( ) →Izq( )) entonces x→P( ) →Izq(y) sino x→P( ) →Der(y) fsi fsi	
6	y→Izq(x)	
7	x→P(y)	
8	regrese	

Rotación a la izquierda de un nodo del árbol. O(1)

Rotación a la derecha es similar

Árboles roji-negros

Inserción de un nuevo elemento en un árbol rojinegro. O(lg n).

19/10/98

insARN(ApuntadorA NodoRN: x)

{pre: x ≠ NodoRNNoDef } { pos: A'= A ∪ {x} }

```
1   InsArBin(ContenidoDe x)
2   x→Color("rojo")
3   (x ≠ raiz ∧ x→P( ) →Color( ) = "rojo")
      [ Si (x→P( ) = x→P( ) →P( ) →Izq( )) entonces
          y = x→P( ) →P( ) →Der( )
          Si (y→Color( ) = "rojo")  entonces
             x→P( ) →Color("negro")              // Caso 1
             y→Color("negro"), x→P( ) →P( ) →Color("rojo"), x = x→P( ) →P( )
          sino  Si (x = x→P( ) →Der( ) )  entonces
                   x = x→P( )                    // Caso 2
                   rotarIzq(x)
                fsi
                x→P( ) →Color("negro")           // Caso 3
                x→P( ) →P( ) →Color("rojo"), rotarDer(x→P( ) →P( ))
          fsi
        sino
          y = x→P( ) →P( ) →Izq( )
          Si (y→Color( ) = "rojo")  entonces
             x→P( ) →Color("negro")              // Caso 1
             y→Color("negro"), x→P( ) →P( ) →Color("rojo"), x = x→P( ) →P( )
          sino  Si (x = x→P( ) →Izq( ) )  entonces
                   x = x→P( )                    // Caso 2
                   rotarDer(x)
                fsi
                x→P( ) →Color("negro")           // Caso 3
                x→P( ) →P( ) →Color("rojo"), rotarIzq(x→P( ) →P( ))
           fsi
        fsi]
4   raiz→Color("negro")
5   regrese
```

-y. ApuntadorA NodoRN. Nodo tío de x.
-Izq( ), Der( ), P( ), Color( ). Definidos en la clase NodoRN.
-raíz. Definido en la clase ArbRN.

Árboles roji-negros

Eliminación de un elemento en un árbol rojinegro. O(lg n).

19/10/98

eliARN(ApuntadorA NodoRN: z)

{pre: z ≠ NodoRNNoDef }

{ pos: A'= A - {z} }

1	y = Si (z→Izq( ) = sen ∨ z→Der( ) = sen) entonces z sino z→sucesor( ) fsi	-y. ApuntadorA NodoRN. Variable auxiliar. -Izq( ), Der( ), P( ), Color( ). Definidos en la clase NodoRN. -raíz. Definido en la clase ArbRN. -sen. ApuntadorA NodoRN. Nodo nulo con el campo color en negro.
2	x = Si (y→Izq( ) ≠ sen) entonces y→Izq( ) sino y→Der( ) fsi	
3	x→P(y→P( ))	
4	Si (y→P( ) = sen) entonces raiz = x sino Si (y = y→P( ) →Izq( )) entonces y→P( ) →Izq(x) sino y→P( ) →Der(x) fsi fsi	
5	Si (y ≠ z) entonces z→Color(y→Color( )), z→Clave(y→Clave( )) fsi	
6	Si (y→Color( ) = "negro") entonces FijarColor(x) Fsi	
7	regrese	

Árboles roji-negros

19/10/98

fijarColor(ApuntadorA NodoRN: x)

{pre: x ≠ NodoRNNoDef } { pos: A'= A }

1	(x ≠ raiz ∧ x→Color( ) = "negro")
	[Si (x = x→P( ) →Izq( )) entonces
	w = x→P( )→Der( )
	Si (w→Color( ) = "rojo") entonces
	w→Color("negro") // Caso 1
	x→P( )→Color("rojo"), rotarIzq(x→P( )), w = x→P( ) →Der( )
	fsi
	Si (w→Izq( )→Color( ) = "negro" ∧ w→Der( )→Color( ) = "negro") ent.
	w→Color("rojo") // Caso 2
	x = x→P( )
	sino Si (w→Der( )→Color( ) = "negro") entonces
	w→Izq( )→Color("negro") // Caso 3
	w→Color("rojo"), rotarDer(w), w = x→P( )→Der( )
	fsi
	w→Color(x→P( )→Color( )) // Caso 4
	x→P( ) →Color("negro"), w→Der( ) →Color("negro")
	rotarIzq(x→P( )), x = raiz
	sino
		-w. ApuntadorA NodoRN. Variable auxiliar.
		-Izq( ), Der( ), P( ), Color( ). Definidos en la clase NodoRN.
		-raíz, rotarIzq( ), rotarDer( ). Definidos en la clase ArbRN.

Árboles roji-negros

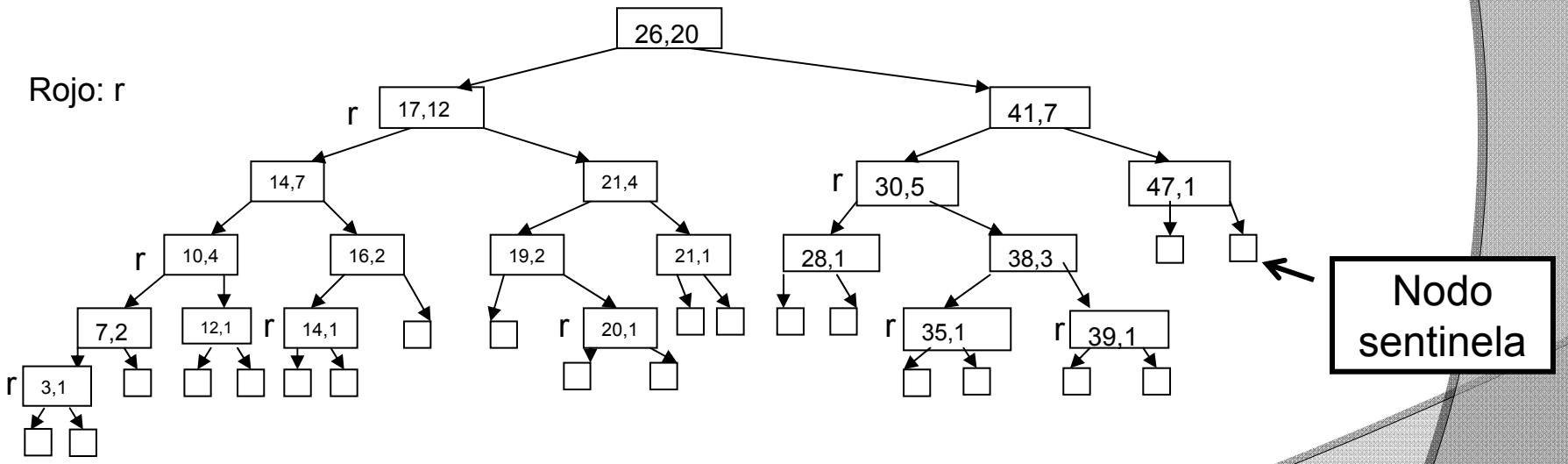
```
    w = x→P( )→Izq( )
      Si (w→Color( ) = “rojo”)  entonces
        w→Color(“negro”)                                    // Caso 1
        x→P( )→Color(“rojo”),    rotarDer(x→P( ))
        w = x→P( ) →Izq( )
      fsi
      Si (w→Der( )→Color( ) = “negro” ∧ w→Izq( )→Color( ) = “negro”)  ent.
        w→Color(“rojo”)                                     // Caso 2
        x = x→P( )
      sino   Si (w→Izq( )→Color( ) = “negro”)  entonces
               w→Der( )→Color(“negro”)                      // Caso 3
               w→Color(“rojo”),    rotarIzq(w)
               w = x→P( )→Izq( )
             fsi
             w→Color(x→P( )→Color( ))                       // Caso 4
             x→P( ) →Color(“negro”),    w→Izq( ) →Color(“negro”)
             rotarDer(x→P( )),        x = raiz
       fsi
    fsi    ]
2   x→Color(“negro”)
3   Regrese
```

Continuación de fijarColor()

Árboles roji-negros aumentados

- Número de nodos en el subárbol incluyendo la raíz: nro
- Formato de un nodo:

color	clave	nro	p	izq	der



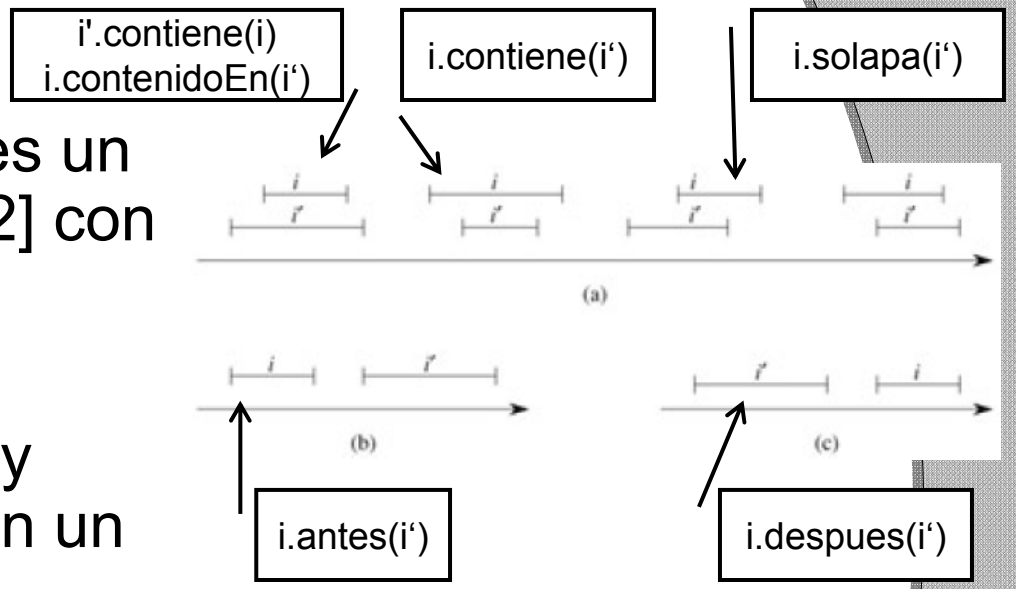
Para el nodo sentinela nro=0, nro(x)=nro(izq(x))+nro(der(x))+1
Para un nodo x, su rango o posición de la clave en el conjunto de claves ordenadas viene dado por: número de nodos que lo preceden +1

Modificación de un TAD

Aumento o modificación de una estructura de datos

- Escoger la estructura de datos
- Determinar la información adicional que debe ser mantenida en la estructura de datos
- Verificar que la información adicional puede ser mantenida mediante una modificación básica en las operaciones de la estructura de datos
- Desarrollar nuevas operaciones
- Documentar la nueva estructura de datos aumentada

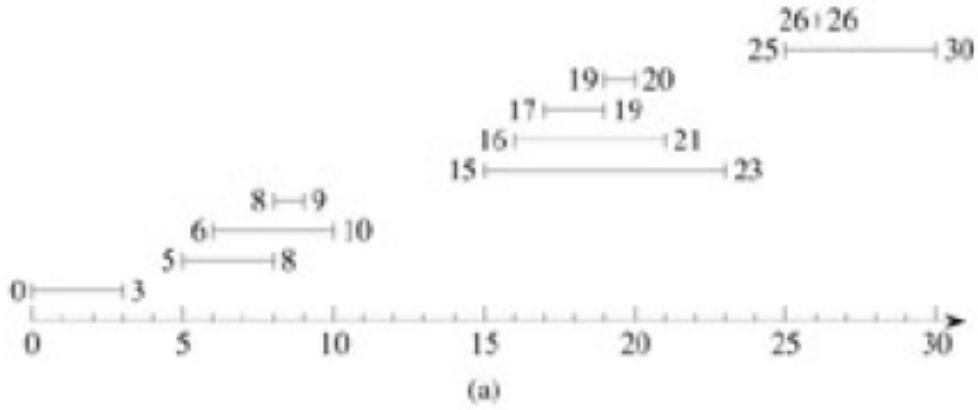
Árboles de intervalos



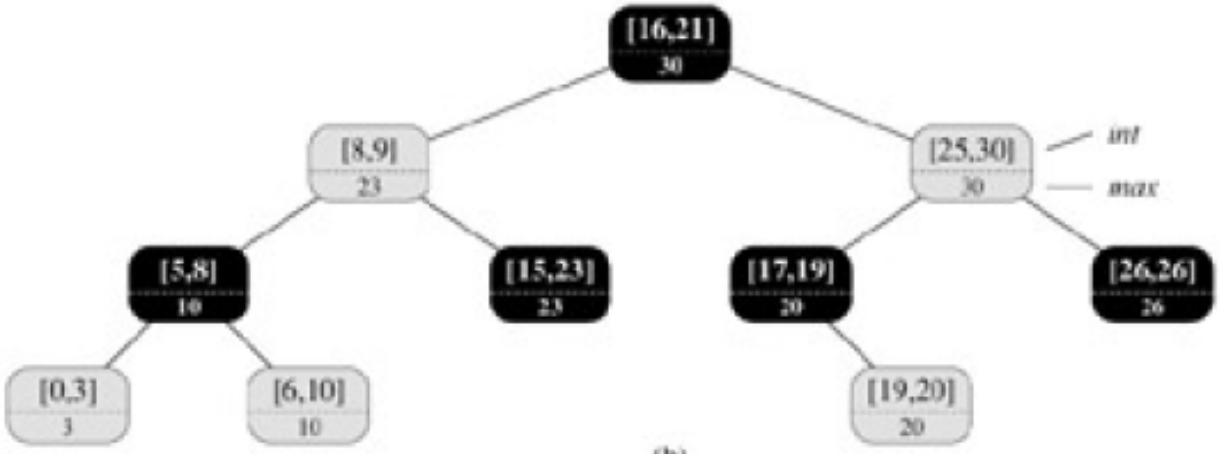
- Intervalo cerrado: es un par ordenado [t1, t2] con t1≤ t2
- {t / ℜ: t1 ≤ t ≤ t2}
- Intervalos abiertos y semi-abiertos tienen un tratamiento similar
- [bajo, alto] = [t1, t2]
- Dos intervalos i e i' solapan si i ∩ i' ≠∅
- Árbol de intervalos es un árbol roji-negro donde la clave es un intervalo

- Inserta(T, x) anexa x que contiene un intervalo
- Elimina(T, x) remueve x
- Consulta(T, i) regresa el x que solapa i, o regresa nulo en caso contrario

Árboles de intervalos



(a)



(b)

- Verifica O(lg n)
- max(x) = MAX(alto.int(x), max(izq(x)), max(der(x)))
- Actualizar max luego de una rotación en O(1)