estudios de postgrado en computación

Análisis y Diseño de Algoritmos (AyDA)

Isabel Besembel Carrera

ÁRBOLES MONOCRITERIO VEB Y TREAP

Contenido

- Introducción a la indexación
- Modelo de cálculo RAM
- Árboles monocriterio
- Árboles VEB
- Treap

Introducción

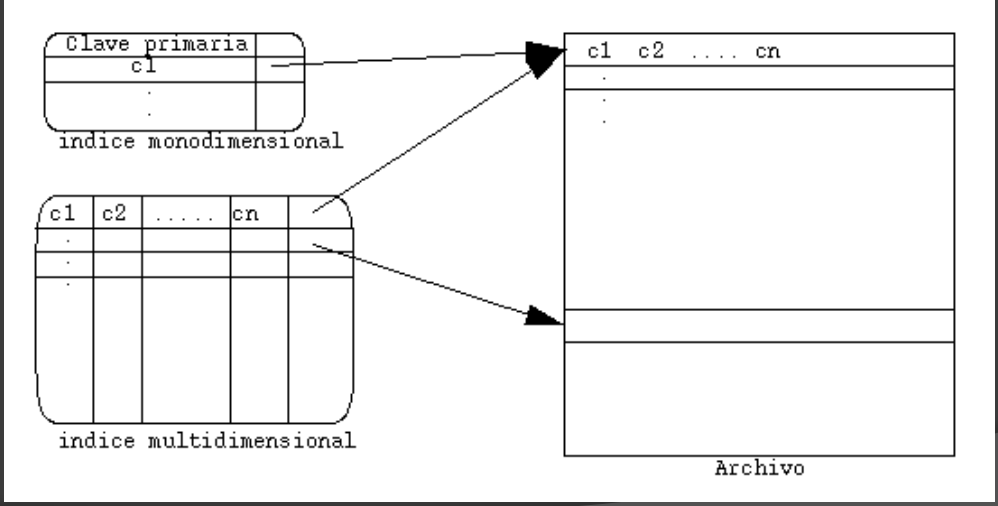
- **Estructuras de datos:** Colección de elementos de datos individuales, relacionados entre sí mediante una organización dada por sus primitivas de acceso
- Básicas:
  - Cadenas de caracteres
  - Listas
  - Conjuntos
  - Grafos

Estructuras de datos avanzadas

- Utilizadas en la construcción de índices agilizan los procesos de búsqueda y recuperación de los datos almacenados en memoria secundaria
- Indexación de archivos: permite tener varios caminos de acceso a los datos almacenados en los archivos
- Índices:
  - Monocriterio o monodimensionales
  - Multicriterio o multidimensionales

Índices monocriterio vs. multicriterio

- Monocriterio: Sus entradas se construyen según una única clave que normalmente es una clave primaria (uno o varios campos concatenados de un registro que identifican unívocamente cada registro del archivo de datos)
- Multicriterio: Sus entradas están construidas con varios campos no concatenados (claves secundarias) de un registro del archivo de datos



Modelo de cálculo RAM

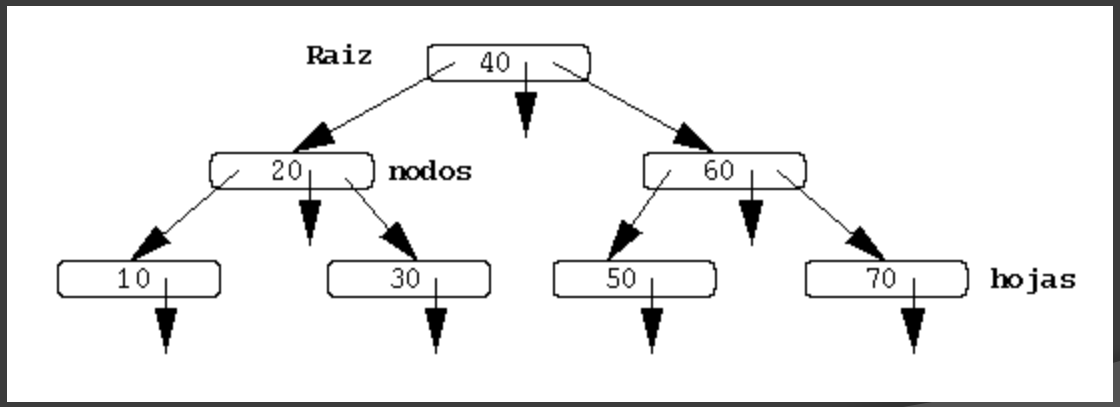
- Random Access Machine (RAM):
  - La memoria es un vector de ranuras
  - Si se conoce la posición en el vector, se accede a la posición en O(1)
  - El costo de un algoritmo usando RAM es lineal en el número total de instrucciones

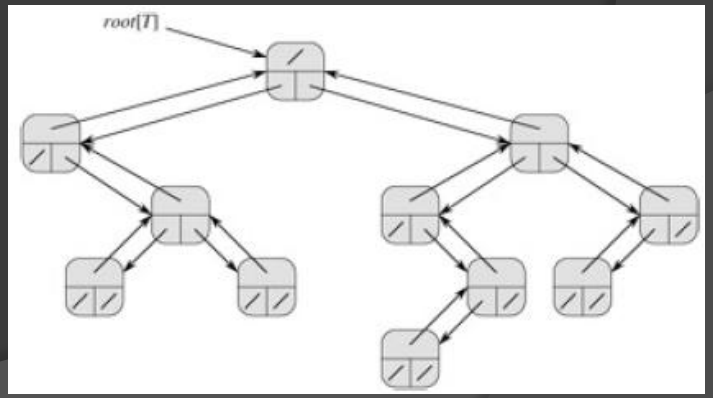
0 1 2 ….

$Direccion_j = DireccionBase + j * tamaño\ de\ la\ ranura$

Árbol binario

- Un árbol binario de tipo T es una estructura que puede ser vacía o que puede estar formada por:
  - un dato de tipo T denominado raíz del árbol binario
  - un árbol binario de tipo T denominado subárbol izquierdo
  - un árbol binario de tipo T denominado subárbol derecho
- Altura del árbol binario (**h**)
- Número máximo de nodos = $2^{h+1} - 1$





Especificación del TAD ArbolBin

<table>
<tr><td colspan="2">17/4/98
Especificación ArbolBin[TipoEle]</td><td></td></tr>
<tr><td>1</td><td>Sintáctica:
creaArbolBin() ArbolBin,
insArbolBin(ArbolBin,TipoEle) ArbolBin,
eliArbolBin(ArbolBin, TipoEle) ArbolBin,
busArbolBin(ArbolBin, TipoEle) Lógico,
preorden(ArbolBin) ArbolBin,
enorden(ArbolBin) ArbolBin,
posorden(ArbolBin) ArbolBin,
altura(ArbolBin) Entero+,
vacíoArbolBin(ArbolBin) Lógico,
destruyeArbolBin(ArbolBin) .</td><td rowspan="3">-creaArbolBin(): Crea un árbol binario de búsqueda vacío.
-insArbolBin(): Ingresa un nuevo elemento al árbol binario en la posición que le corresponde según el orden.
-eliArbolBin(): Elimina el elemento en el árbol binario, si existe. Si está vacío no hace ninguna eliminación.
-destruyeArbolBin(): Destruye el árbol binario.
-vacíoArbolBin(): Regresa Verdadero si el árbol binario está vacío.
-busArbolBin(): Devuelve Verdadero si el elemento está en el árbol binario, de lo contrario regresa Falso.
-preorden(): Despliega los elementos según el orden raíz, izquierdo y derecho.
-posorden(): Despliega los elementos del árbol binario de búsqueda ordenados ascendentemente.
-enorden(): Despliega los elementos según el orden izquierdo, derecho y raíz.
-altura(): Regresa la altura actual.</td></tr>
<tr><td>2</td><td>Declaraciones:
TipoEle: e, {TipoEleNoDef}</td></tr>
<tr><td>3</td><td>Semántica:
vacíoArbolBin(creaArbolBin()) = Verdadero
vacíoArbolBin(insArbolBin(creaArbolBin(), e)) = Falso
busArbolBin(creaArbolBin(), e) = Falso
busArbolBin(insArbolBin(creaArbolBin(), e), e) = Verdadero
eliArbolBin(creaArbolBin()) = creaArbolBin()
altura(creaArbolBin()) = 0</td></tr>
</table>

Implementación del TAD ArbolBin

ArbolBin[TipoEle] Clases: Entero, Lógico, ApuntadorA, TipoEle, NodoAB[TipoEle]		
1	Estructura: privado: n : Entero+ = 0 raiz : ApuntadorA NodoAB[TipoEle] = Nulo	-n: Número actual de elementos. -raiz: Apuntador al nodo raíz. -ArbolBin(). Constructor. Crea un árbol vacío.
2	Operaciones: público: ArbolBin() ~ArbolBin() insArbolBin(TipoEle: e) eliArbolBin(TipoEle: e) busArbolBin(TipoEle: e): Lógico vacíoArbolBin(): Lógico numEle(): Entero+ altura(): Entero preorden() enorden() posorden() despliegue() privado: limpiarArbolBin(ApuntadorA NodoAB[TipoEle]: pa) alturaArbolBin(Entero+: c, ApuntadorA NodoAB[TipoEle]: pa): Entero enordenArbolBin(ApuntadorA NodoAB[TipoEle]: pa) preordenArbolBin(ApuntadorA NodoAB[TipoEle]: pa) posordenArbolBin(ApuntadorA NodoAB[TipoEle]: pa)	-~ArbolBin(). Destructor. Destruye el árbol . -insArbolBin(). Transformador. Ingresa un nuevo elemento en la posición que le corresponde según el recorrido enorden. -eliArbolBin(). Transformador. Elimina el elemento, si el árbol binario de búsqueda no está vacío. -busArbolBin(). Observador. Regresa Verdadero si el elemento existe. -vacíoArbolBin(). Observador. Regresa Verdadero si el árbol binario está vacío. -numEle(). Observador. Regresa el número actual de elementos en la estructura. -altura(). Observador. Regresa la altura actual de la estructura. -preorden(). Observador. Despliega los elementos del árbol binario de búsqueda en preorden (RID). -enorden(). Observador. Despliega los elementos del árbol binario de búsqueda en enorden (IRD). -posorden(). Observador. Despliega los elementos del árbol binario de búsqueda en posorden (IDR). -despliegue(). Observador. Despliega el contenido del árbol. -limpiarArbolBin(). Transformador. Devuelve todos los nodos. -alturaArbolBin(). Observador. Devuelve la altura actual del árbol. -enordenArbolBin(). Observador. Recorrido recursivo del árbol. -preordenArbolBin(). Observador. Recorrido recursivo del árbol. -posordenArbolBin(). Observador. Recorrido recursivo del árbol.

Nodo del árbol binario

NodoAB[TipoEle]		
1	Estructura: privado: pPad : ApuntadorA NodoAB[TipoEle]=Nulo info: TipoEle = {TipoEleNoDef} niv: Entero+ = 0 pIzq : ApuntadorA NodoAB[TipoEle]=Nulo pDer : ApuntadorA NodoAB[TipoEle]= Nulo	-pPad: Dirección del nodo padre. -info: Información del nodo. -niv: Nivel del nodo en el árbol binario de búsqueda. -pIzq: Dirección del hijo izquierdo. -pDer: Dirección del hijo derecho. -NodoAB(). Constructores. Crea un nodo con su información definida y dos apuntadores nulos. -PPad(). Observador y transformador. Manejo del apuntador al papá. -Info(). Observador y transformador. Manejo de la información del nodo. -Niv(). Observador y transformador. Manejo del nivel del nodo en el árbol. -PIzq(). Observador y transformador. Manejo del apuntador al hijo izquierdo. -PDer(). Observador y transformador. Manejo del apuntador al hijo derecho.
2	Operaciones: público: NodoAB() NodoAB(TipoEle: in, Entero+: ni) NodoAB(TipoEle: in) PPad(ApuntadorA NodoAB[TipoEle]: izq) PPad() : ApuntadorA NodoAB[TipoEle] Info(TipoEle: e) Info(): TipoEle Niv(): Entero+ Niv(Entero+: ni) PIzq(ApuntadorA NodoAB[TipoEle]: izq) PIzq() : ApuntadorA NodoAB[TipoEle] PDer(ApuntadorA NodoAB[TipoEle]: der) PDer(): ApuntadorA NodoAB[TipoEle]	

Árbol binario

- Ventajas:
  - Facilidad y simplicidad en los algoritmos de tratamiento.
  - Para encontrar la información buscada no es siempre necesario llegar al nivel de las hojas.
  - Los costos de tratamiento son logarítmicos.
- Desventajas:
  - Depende del orden de inserción de los datos, por lo que es muy frecuente obtener árboles desequilibrados o desbalanceados (una ramas muy largas en comparación con otras). Este desequilibrio obliga a la aplicación de algoritmos para equilibrar, que por lo general no son simples.
  - Se hace difícil realizar un tratamiento secuencial de las claves, dependiendo de la política de inserción

Árboles VEB (van Emde Boas)

- Diccionario ordenado
  - Operaciones:
    - Inserción: insercion()
    - Eliminación: eliminacion()
    - Consulta: consulta()
    - Predecesor: regresa la entrada anterior más pequeña
    - Sucesor: regresa la entrada siguiente más grande
  - Implementaciones posibles
    - Arreglo de bits: insercion() y eliminacion() en O(1), pero sucesor() W(n)=O(n)
    - Árboles roji-negros: todas las operaciones en O(lg n)

Árboles VEB (van Emde Boas)

- Árbol VEB
  - Problema del sucesor en un universo fijo U={0, 1, 2, .., u-1} |U|=u, con S un subconjunto dinámico de {1, 2, …, n} en U
  - Operaciones:
    - Insercion($x \in U \wedge x \notin S$): agrega x a S
    - Eliminacion($x \in S$): remueve x de S
    - Sucesor($x \in U$): regresa el elemento más pequeño $\in$ S y que es > x
    - Predecesor($x \in U$): regresa el más grande elemento $\in$ S y que es < x
  - Colas por prioridad con O(lg lg u) por operación

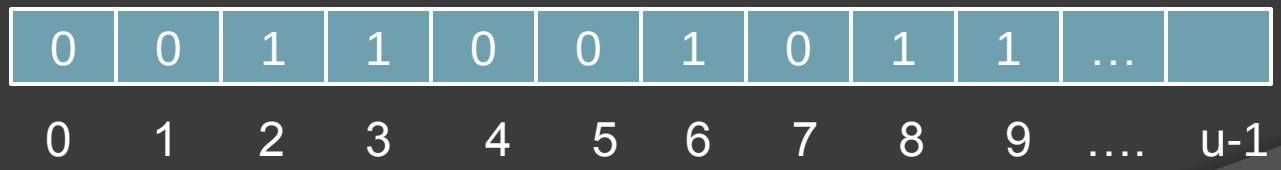
Vector precalculado

- Solución 1 al problema del predecesor o sucesor en un universo fijo
- Usar un vector lleno con los sucesores de cada elemento de U (igual para los predecesores)
- Sucesor() o predecesor() en O(1)
- Actualizaciones en W(n)=Θ(n)

Para S={1,3,7}

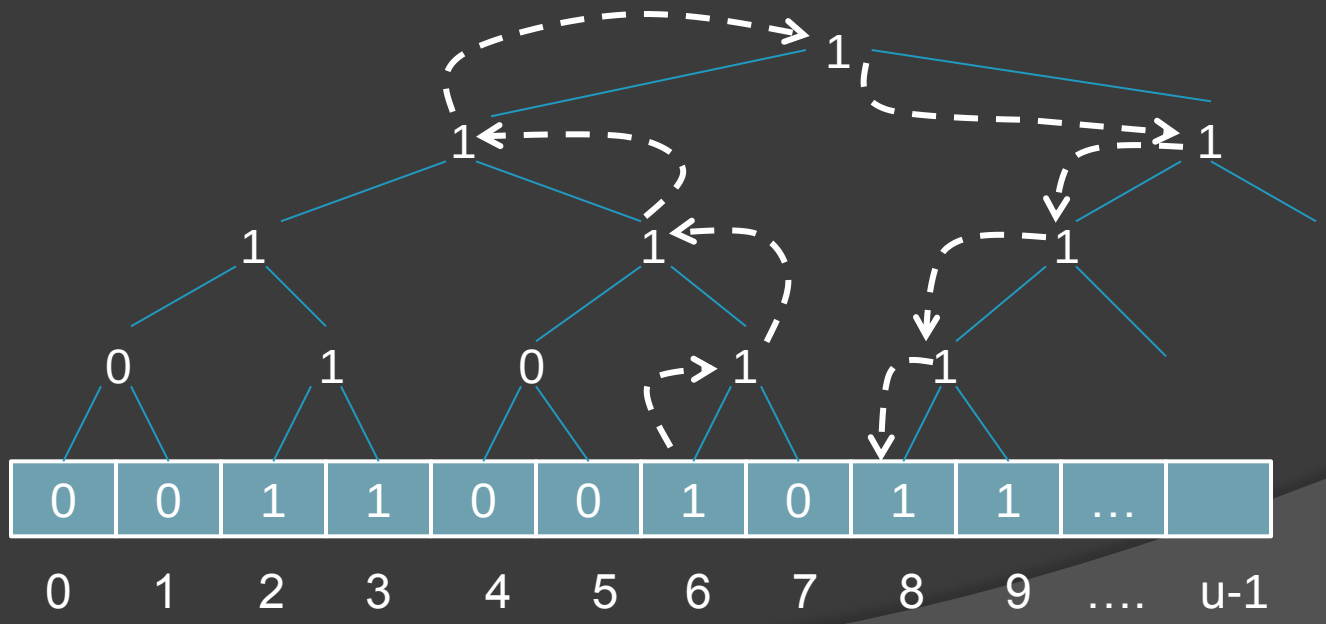
Vector de bits

- Consultas, sucesor() o predecesor() en O(n), por tener que recorrer la sección del vector para encontrar el próximo 1
- Actualizaciones en O(1), porque solo se modifica la ranura indicada
- Para S={2,3,6,8,9}



Mejora del vector de bits

- Construcción de un árbol binario de relaciones $\vee$ (or) sobre el vector de bits
- O(lg u) para cualquier operación



Uso de árboles de altura constante

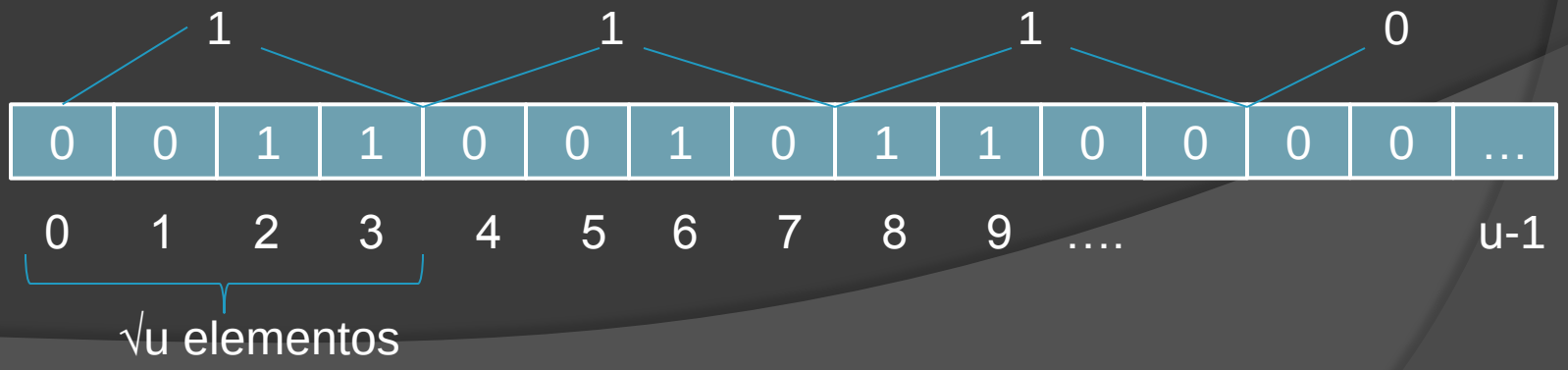
- Agrupar el vector de bits en $\sqrt{u}$ grupos de tamaño $|\sqrt{u}|$
- Cada grupo sub[0], sub[1], …, sub[$\sqrt{u} - 1$]
- Como cada grupo tiene $\sqrt{u}$ elementos, sub[i] representa los elementos $\{i\sqrt{u}, i\sqrt{u} + 1, \ldots, (i+1)\sqrt{u} - 1\} \in U$
- Inicio de la búsqueda del sucesor en el grupo donde está el elemento, se busca linealmente dentro del grupo por el sucesor
- Si no está, consultar la estructura auxiliar que indica el próximo grupo no vacío, buscando en dicho grupo en forma lineal se encontrará el sucesor, si existe

Árboles de altura constante

- Consultas en $O(\sqrt{u})$, por la búsqueda lineal en los dos grupos y en la estructura auxiliar
- Inserciones en O(1), porque solo debe escribir el bit en el grupo y posiblemente en la estructura auxiliar
- Manipulación de los bits:
  - Representación binaria de $x \in U$, con $\lceil \lg u \rceil$ bits. Ej: 55 en $|U|=256$, 00110111_2

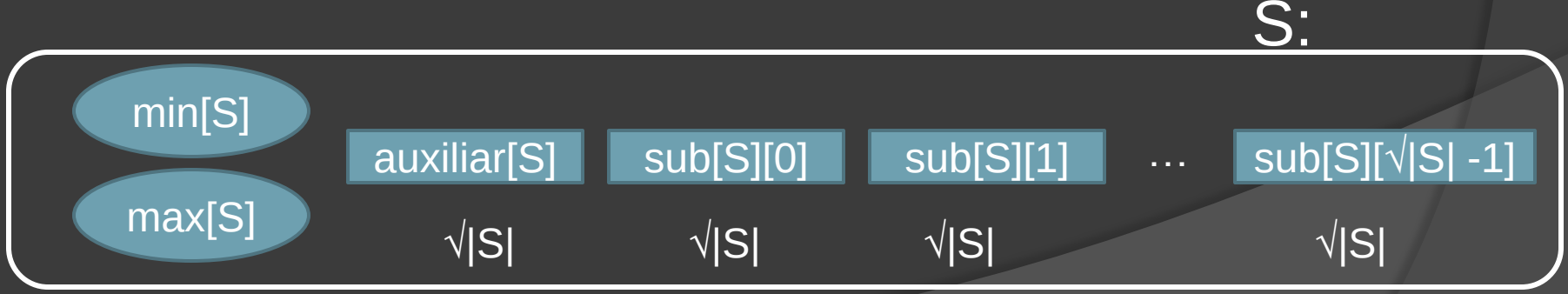
Árboles de altura constante

- Alto(x)=mitad de los bit de orden superior (0011_2) = $\lfloor x/\sqrt{u} \rfloor$ = en cuál de los $\sqrt{u}$ grupos x está o debería estar
- bajo(x)=mitad de los bits de orden inferior (0111_2) = x mod $\sqrt{u}$ = subíndice dentro del grupo
- Tales funciones mapean un elemento a una localidad dentro de una cubeta



Árboles VEB

21/10/09	insertar(x, S)	
1	a, x, min[S]=Si(x<min[S]) entonces x, min[S], a fsi	-S: Conjunto de elementos de tamaño |S| -sub[S][i]: Grupo o cluster de S, con i=0,1,…, √|S| - 1 -auxiliar(): vector auxiliar que controla los grupos vacíos -min, max: Valor mínimo y máximo de los elementos contenidos en S -a: variable auxiliar para el intercambio -alto(), bajo(): funciones de mapeo de los elementos al vector de bits
2	si(min[sub[S][alto(x)]] es vacío) entonces insertar(bajo(x), sub[S][alto(x)]) min[sub[S][alto(x)]]=bajo(x) sino insertar(alto(x), auxiliar[S]) fsi	
3	max[S] = Si(x>max[S]) entonces x fsi	



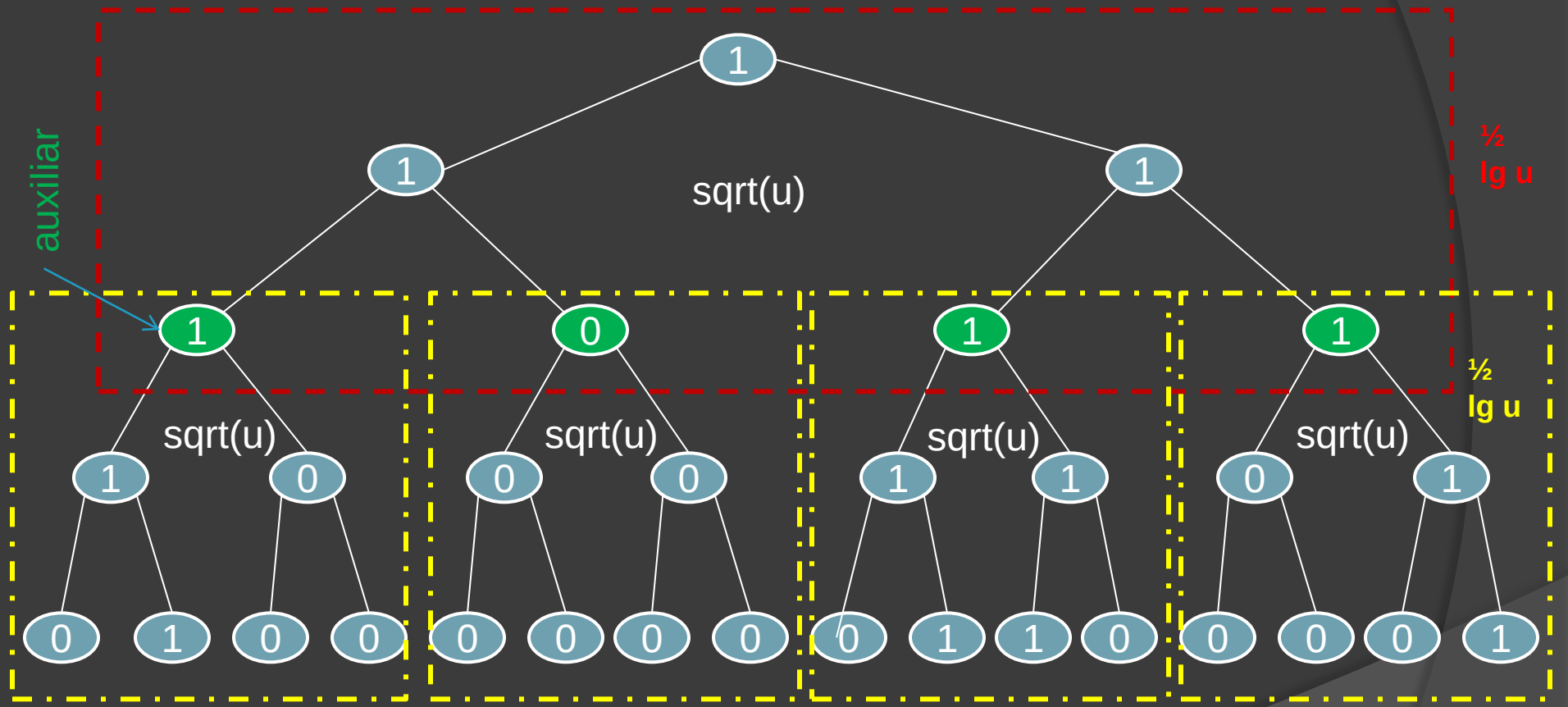
Árboles VEB

21/10/09	sucesor(x, S)	
1	Si(x < min[S]) entonces regrese min[S] fsi	-S: Conjunto de elementos de tamaño |S| -sub[S][i]: Grupo o cluster de S, con i = 0,1,…, √|S| - 1 -min, max: Valor mínimo y máximo de los elementos contenidos en S -auxiliar(): vector auxiliar que controla los grupos vacíos -alto(), bajo(): funciones de mapeo de los elementos al vector de bits
2	Si(bajo(x) < max[sub[S][alto(x)]]) entonces regrese alto(x) * √|S| + j=sucesor(bajo(x), sub[S][alto(x)]) sino i=sucesor(alto(x), auxiliar[S]) regrese min[sub[S][i]] + i * √|S| fsi	

Complejidad en tiempo O(lg lg u) para todas las operaciones: insertar, eliminar, sucesor y predecesor

Peter van Emde Boas, R. Kaas y E. Zijlstra. Design and implementation of an efficient priority queue. Math. Systems Theory, 10:99-127, 1977.

Árbol VEB



Se puede ver como un árbol de árboles

S={1, 9, 10, 15}

$9_{10} = 1001_2$, “der, izq, izq, der” , alto(9)=10_2, bajo(9)=01_2

lg u = 2(½ lg u)

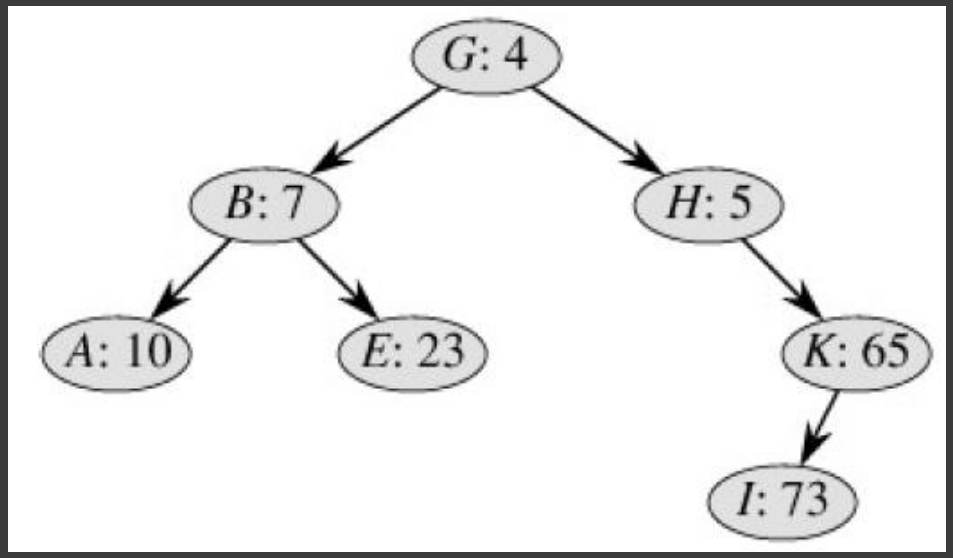
Treap (tree and heap)

- Un árbol binario de búsqueda puede llegar a una estado de desbalance total (lista)
- Si se construye con valores aleatorios, el árbol binario de búsqueda tiende a ser balanceado
- Una buena estrategia es permutar los valores de entrada aleatoriamente antes de insertarlos, y si no se tienen todos?
- **Treap:** es un árbol binario de búsqueda con una modificación en el orden de los nodos

Treap

- Se agrega al nodo un campo que contendrá un valor aleatorio llamado prioridad
- Se asume: todas las claves y las prioridades distintas
- Las claves obedecen al orden de los árboles binarios de búsqueda
- Las prioridades obedecen al orden de los montículos binarios por el mínimo

Treap



Si v es un hijo izquierdo de u, entonces clave(v) < clave(u)

Si v es un hijo derecho de u, entonces clave(v) > clave(u)

Si v es un hijo de u, entonces la prioridad(v) > prioridad(u)

• Si se insertan las claves x1, x2,…, xn en un treap. El treap resultante es un árbol formado como si las claves se hubieran insertado en un árbol binario de búsqueda en el orden dado por sus prioridades (generadas aleatoriamente).

• Si prioridad(x_i) < prioridad(x_j) significa que x_i fue insertado antes de x_j

Inserción en un treap