Resumen teórico de Estructuras de Datos

TABLA DE CONTENIDO

1 – EXCEPCIONES

Excepciones: try-catch-finally

```
try  {
        .....
}  catch( tipo_excepcion_1 e ) {
        código para manejar la excepción e
} catch( tipo_excepcion_2 e ) {
        código para manejar la excepción e
} finally {
      código que siempre se ejecuta
}
```

- Los métodos deben especificar en su signatura las excepciones que pueden lanzar:
  tipo método(parámetros formales) throws clase_excepción
- Una excepción se lanza desde una sentencia throw:
  throw new clase_excepción(parámetros actuales);

```java
// PersonaException.java
public class PersonaException extends Exception {
    public PersonaException( String msg ) {
        super( msg );
    }
}
```

```java
public Persona( String nombre, int edad ) throws PersonaException {
    this.nombre = nombre;
    if( edad <= 0 )
        throw new PersonaException( "Edad negativa: " + edad );
    this.edad = edad;
}
```

Notas: Excepciones checked y unchecked

- Las excepciones como ArrayIndexOutOfBoundsException se llaman *unchecked* porque el programa compilará aunque el cliente no tengan un try-catch asociado. Esto ocurre con las RuntimeExceptions.
- Las excepciones como PersonaException se llaman *checked* porque el cliente no compilará si el código cliente no está protegido con el try-catch correspondiente.

Runtime exception	Significado
ArithmeticException	Error de aritmética, e.g. división por cero
ArrayIndexOutOfBoundsException	Indice de arreglo fuera de límites
ClassCastException	Cast inválido
IndexOutOfBoundsException	Algún índice de array fuera de límites
NegativeArraySizeException	Array creado con tamaño negativo
NullPointerException	Uso incorrecto de una referencia nula
NumberFormatException	Conversión inválida de string en número
StringIndexOutOfBounds	Se quiso acceder a una posición inexistente de un string

Método	Significado
String getMessage()	Retorna una descripción de la excepción
void printStackTrace()	Imprime a la consola la pila de registros de activación con nombre de método y número de línea en cada método donde se propagó la excepción hasta que fue capturada.
String toString()	Retorna una descripción de la excepción

2 – INTERFACES

Interfaces

- Una interfaz es una colección de declaraciones de métodos.

```
public interface nombre_interfaz
{
    public tipo método1( parámetros formales ) throws excepciones;
     ...
    public tipo métodon( parámetros formales ) throws excepciones;
}
```

Ventajas del uso de interfaces

- Al usar una interfaz uno especifica "qué" debe hacer una clase pero no "cómo" debe hacerlo.
- Las interfaces no hacen suposiciones de "cómo" serán implementadas.
- Una vez que se define una interfaz, ésta puede ser implementada por cualquier número de clases.
- Implementar una interfaz requiere implementar todos los métodos definidos por la misma.
- Brindan una forma limitada y segura de herencia múltiple (una clase puede implementar más de una interfaz).

Ventajas del uso de interfaces

- Las interfaces están pensadas para soportar la resolución de métodos en tiempo de ejecución.
- Esto se puede lograr con clases abstractas también pero las mismas deberían estar más y más arriba en la jerarquía oscurenciéndola (este problema se evita con las interfaces ya que dos clases no relacionadas pueden implementar la misma interfaz).

3 – ASIGNACIONES

Reglas para asignar en forma válida: a = b;

Supongamos que *a* es de tipo *U* y *b* de tipo *T*. Es válido asignar cuando:

- "a" y "b" son del mismo tipo de clase o interfaz (i.e. *T* = *U*).
- Cuando *T* es más *amplio* que *U*, se produce una conversión:
  - Widening conversion:
    - T y U son nombres de clases y U es superclase de T
    - T y U son nombres de interfaces y U es superinterfaz de T
    - T es una clase que implementa la interfaz U
    - (T es una subclase de una clase que implementa una interfaz que es subinterfaz de U).
- Son todas verificables por el compilador.

Narrowing conversions

Ocurre cuando un tipo T se convierte en un tipo S más *angosto*.

Supongamos "S a;" y "T b;" y quiero escribir "a = b;".

- Casos comunes:
  T y S son clases y S es subclase de T.
- T y S son interfaces y S es subinterfaz de T.
- T es una interfaz implementada por la clase S.

Se requiere casting explícito (e.g. "a = (S) b;") y su correctitud va a ser controlada en ejecución.

La falla produce una ClassCastException.

Autoboxing y Unboxing

Autoboxing: Conversión automática entre tipo primitive y tipo wrapper.

```
int i = 8;
Integer x = i;  // int se convierte en Integer
```

Unboxing: Conversión automática entre tipo wrapper y tipo primitivo.

```
int j  = x;  // Integer se convierte en int.
```

Ojo: Las clases wrapper no sirven para simular pasaje de parámetros por referencia.

4 – MÉTODOS DE ORDENAMIENTOS

Resumen de métodos de ordenamiento

- Multipasada:
  - Selección (selection sort)
  - Burbuja (bubble sort) o intercambio (exchange sort)
  - Inserción (insertion sort)
- Merge sort: Ordenamiento por mezcla
- Quick sort
- Heap sort (este lo daremos cuando demos *colas con prioridad* en la clase 14)

Intercambio de dos elementos del arreglo

```
public static void swap( int [] a, int i, int j )
{
        int temp = a[i];    // Hago una copia de a[i] en temp
        a[i] = a[j];        // Guardo a[j] en a[i]
        a[j] = temp;        // Guardo la copia de a[i] que tengo en temp en a[j]
}
```

Código Java:

```
public static void selectionSort( int [] a, int n ) {
        for( int i=0; i<n-1; i++ ) {
          // Hallar el mínimo a[p] de a[i], ..., a[n-1]:
          p = i;  // Asumo que el mínimo está en i
          for( int j=i+1; j<n; j++ )
            if( a[j] < a[p] ) p = j;  // Si el valor de j es menor al de  p, actualizo p.

          // Intercambiar a[i] y a[p]:
          swap( a, p, i );

          // Ahora a[0] <= ... <= a[i] y
          // a[i] <= a[j] para todo i y j con i < j < n
        }
}
```

Estructuras de datos - Dr. Sergio A. Gómez 6

Código Java:

```
public static void bubbleSort( int [] a, int n ) {
        for( int i=n-1; i>=0; i-- ) {
                // Burbujear el item más grande en a[0], ..., a[i] a a[i]
                for( int j=0; j<i; j++ )
                        if( a[j] > a[j+1] ) {  // si a_j y a_{j+1} están desordenados
                                // intercambiar a_j y a_{j+1}
                                swap( a, j,  j+1 );
                        }
                // Ahora a[i] <= ... <= a[n] y
                // a[j] <= a[i] para todo j con 0 <= j < i-1
        }
}
```

Estructuras de datos - Dr. Sergio A. Gómez 8

Optimización: Cuando nunca se entra al if en una iteración del for externo significa que el arreglo está ordenado.
La optimización consiste en ubicar una bandera para saber si se ejecutó al menos un burbujeo.
Si al terminar la iteración interna , no se hizo ningún burbujeo, la bandera estará en falso indicando que el arreglo está ordenado y que se puede terminar.

```
public static void bubbleSortMejorado( int [] a, int n ) {
        boolean seguir = true;  // Bandera que indica que hubo algún burbujeo.
        for( int i=n-1; i>=0 && seguir; i-- ) {
                seguir = false;  // Asumo que no va a haber burbujeo.
                for( int j=0; j<i; j++ )
                        if( a[j] > a[j+1] ) {
                                // Intercambiar ítems y anotar que hubo burbujeo
                                swap( a, j, j+1 );
                                seguir = true; // Indico que sí hubo burbujeo.
                        }
                // Si no hubo burbujeo, nunca se entró al if y seguir=falso indicando
                // que el arreglo está ordenado.
        }
}
```

Código Java:

```
public static void insertionSort( int [] a, int n ) {
    for( int i=1; i<n; i++ ) {      // Insertar a[i] en la secuencia ordenada a[0], ..., a[i-1]
        int item = a[i];        // item a insertar
        int j = i;              // j = cursor de inserción
        boolean found = false;  // found = encontré ubicación
        while( j>0 && !found ) {  // mientras haya lugar para mirar y no encontré el lugar
            if( a[j-1] <= item ) {  // si item debería ser a[j]
                found = true;  // entonces anoto que encontré su lugar
            } else {                // sino
                a[j] = a[j-1];    // Desplazar a[j-1] un lugar a la derecha
                j--;              // Actualizar j para el siguiente elemento a considerar
            }
        }
        a[j] = item;                // Insertar item en posición j
    }
}
```

```
private static void merge( int [] a, int ini, int medio, int fin) {
    int i = ini, j = medio+1, k = 0;    // Inicializo cursores i, j, k.
    int [] b = new int[fin-ini+1];      // Creo arreglo auxiliar b del tamaño de la porción de a.

    while (i<=medio && j<=fin)      // Mientras no se terminen las porciones:
      if (a[i] < a[j]) b[k++] = a[i++];  // Copio el menor de los elementos en b.
      else b[k++] = a[j++];              // y avanzo cursores correspondientes.

    while (i<=medio) b[k++] = a[i++];    // Si se terminó la 2da mitad, copio el resto de la 1era.
    while (j<=fin) b[k++] = a[j++];      // Si se terminó la 1era mitad, copio el resto de la 2da.

     for(i=ini, k=0; i<=fin; i++, k++) a[i]=b[k];  // Copio contenido de b en porción de a e/ini y fin.
}
```

Estructuras de datos - Dr. Sergio A. Gómez 13

5 – TIEMPO DE EJECUCIÓN

> Introducción

Factores que afectan el tiempo de ejecución

- Aumenta con el tamaño de la entrada de un algoritmo
- Puede variar para distintas entradas del mismo tamaño
- Depende del hardware (velocidad del reloj, procesador, cantidad de memoria, tamaño del disco, ancho de banda de la conexión a la red)
- Depende del sistema operativo
- Depende de la calidad del código generado por el compilador
- Depende de si el código es compilado o interpretado

Tiempo de un algoritmo

- El tiempo de ejecución de un algoritmo depende de la cantidad de operaciones primitivas realizadas.
- Las operaciones primitivas toman tiempo constante y son:
  - Asignar un valor a una variable
  - Invocar un método
  - Realizar una operación aritmética
  - Comparar dos números
  - Indexar un arreglo
  - Seguir una referencia de objeto
  - Retornar de un método
- Como estos tiempos dependen del compilador y del hardware subyacente, por ello los notaremos con constantes arbitrarias c_1, c_2, c_3, ...

Preliminares: Funciones

- Constante: f(n) = c
- Logaritmo: f(n) = $\log_b(n)$ para b>1
- Lineal: f(n) = n
- N-LogN: f(n) = n log(n)
- Cuadrática: f(n) = n^2
- Cúbica: f(n) = n^3
- Polinomial: f(n) = n^k, con k natural
- Exponencial: f(n) = a^n con a real positivo y n
- Factorial: f(n)=n!

> Big-Omega y Big-Theta

Big-Omega y Big-Theta

- Big-Omega: Sean f(n) y g(n) funciones de los naturales en los reales. f(n) es $\Omega(g(n))$ ssi existen c real positivo y n_0 natural tales que $f(n) \geq cg(n)$ para todo $n \geq n_0$.
- Big-Theta: Sean f(n) y g(n) funciones de los naturales en los reales. f(n) es $\Theta(g(n))$ ssi f(n) es $O(g(n))$ y f(n) es $\Omega(g(n))$.
- Nota: Big-Theta quiere decir $c_1g(n) \leq f(n) \leq c_2g(n)$. Por lo tanto, tienen un crecimiento asintótico equivalente.

Reglas de la suma y el producto

- Regla de la suma:
  Si $f_1(n)$ es $O(g_1(n))$ y
  $f_2(n)$ es $O(g_2(n))$ entonces
  $f_1(n) + f_2(n)$ es $O(\max(g_1(n),g_2(n))$
- Regla del producto:
  Si $f_1(n)$ es $O(g_1(n))$ y
  $f_2(n)$ es $O(g_2(n))$ entonces
  $f_1(n)*f_2(n)$ es $O(g_1(n)*g_2(n))$

Propiedades útiles

Para probar por inducción:

$$\sum_{i=1}^{n} i = 1+2+\ldots+n = \frac{n(n+1)}{2}$$

$$\sum_{i=0}^{n} 2^i = 2^{n+1}-1 \qquad \sum_{i=0}^{n} r^i = \frac{r^{n+1}-1}{r-1}$$

De las series:

$$\sum_{i=1}^{n}(a_i+b_i) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i \qquad \sum_{i=1}^{n} ca_i = c\sum_{i=1}^{n} a_i$$

Reglas para calcular T(n) a partir del código fuente de un algoritmo

Paso 1: Determinar el tamaño de la entrada n

Paso 2: Aplicar las siguientes reglas en forma sistemática:

- $T_P(n)$ = constante si P es una acción primitiva
- $T_{S1;\ldots;Sk}(n) = T_{S1}(n) + \ldots + T_{Sk}(n)$
- $T_{\text{if B then S1 else S2}}(n) = T_B(n) + \max(T_{S1}(n),T_{S2}(n))$
- $T_{for(m;S)}(n) = m*T_S(n)$ donde
  m=cant_iteraciones(for(m;S))
- $T_{\text{while B do S}}(n) = m*(T_B(n) + T_S(n)) + T_B(n)$ donde
  m=cant_iteraciones(while B do S)
- $T_{\text{call P}}(n) = T_S(n)$ donde procedure P; begin S end
- $T_{f(e)}(n) = T_{x:=e}(n) + T_S(n)$ donde function f(x); begin S end

Explicación en Clase 2 (1:02:41)

Tipo de dato abstracto

- Un *tipo de dato abstracto* (TDA) (en inglés, ADT por Abstract Data Type) es un tipo definido solamente en términos de sus operaciones y de las restricciones que valen entre las operaciones.
- Las restricciones las daremos en términos de comentarios.
- Cada TDA se representa en esta materia con una (o varias) interfaces.
- Una o más implementaciones del TDA se brindan en términos de clases concretas.

7 – TDA PILA

> Operaciones

TDA Pila (Stack)

Operaciones:

- push(e): Inserta el elemento e en el tope de la pila
- pop(): Elimina el elemento del tope de la pila y lo entrega como resultado. Si se aplica a una pila vacía, produce una situación de error.
- isEmpty(): Retorna verdadero si la pila no contiene elementos y falso en caso contrario.
- top(): Retorna el elemento del tope de la pila. Si se aplica a una pila vacía, produce una situación de error.
- size(): Retorna un entero natural que indica cuántos elementos hay en la pila.

Pilas nativas en Java: Operaciones

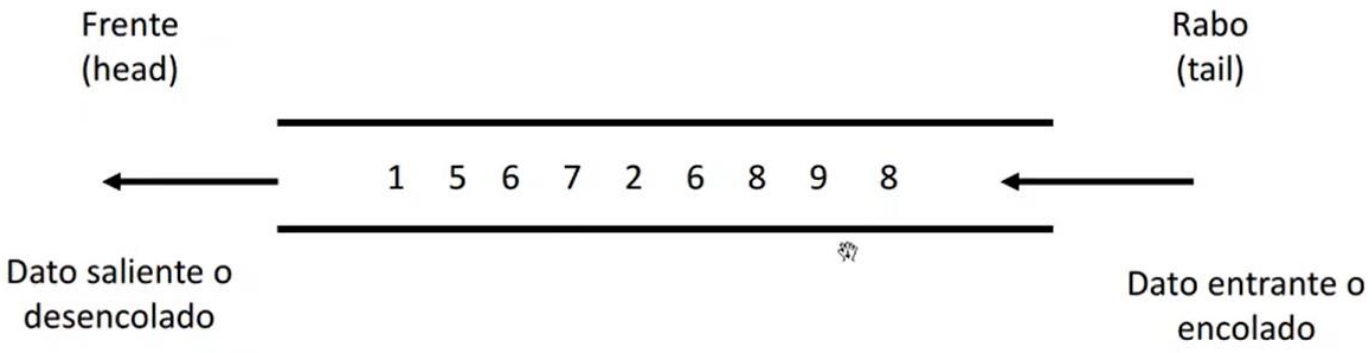
- Java brinda la clase java.util.Stack<E> (la cual es subclase de java.util.Vector<E>)

Constructor Summary
Stack() Creates an empty Stack.

Method Summary	
boolean	empty() Tests if this stack is empty.
E	peek() Looks at the object at the top of this stack without removing it from the stack.
E	pop() Removes the object at the top of this stack and returns that object as the value of this function.
E	push(E item) Pushes an item onto the top of this stack.
int	search(Object o) Returns the 1-based position where an object is on this stack.

8 – TDA COLA

TDA Cola



Una *cola* (queue) es una colección con modelo de secuencia cuyos datos ingresan por un extremo llamado *rabo* (tail) y salen por otro extremo llamado *frente* (head).

Esta política de actualización se llama FIFO (por *first-in first-out*) o también FCFS (*first-come first-served*).

> Operaciones

TDA Cola

Operaciones:

- enqueue(e): Inserta el elemento *e* en el rabo de la cola
- dequeue(): Elimina el elemento del frente de la cola y lo retorna. Si la cola está vacía se produce un error.
- front(): Retorna el elemento del frente de la cola. Si la cola está vacía se produce un error.
- isEmpty(): Retorna verdadero si la cola no tiene elementos y falso en caso contrario
- size(): Retorna la cantidad de elementos de la cola.

9 – TDA LISTA

> Operaciones

ADT Position List: (operaciones de iteración/recorrido)

- first(): retorna la posición del primer elemento de la lista; error si la lista está vacía
- last(): retorna la posición del último elemento de la lista; error si la lista está vacía
- prev(p): retorna la posición del elemento que precede al elemento en la posición p; error si p = first()
- next(p): retorna la posición del elemento que sigue al elemento en la posición p; error si p = last()

ADT Position List (métodos de actualización)

- set(p, e): Reemplaza al elemento en la posición p con e, retornando el elemento que estaba antes en la posición p
- addFirst(e): Inserta un nuevo elemento e como primer elemento
- addLast(e): Inserta un nuevo elemento e como último elemento
- addBefore(p, e): Inserta un nuevo elemento e antes de la posición p
- addAfter(p, e): Inserta un nuevo elemento e luego de la posición p
- remove(p): Elimina y retorna el elemento en la posición p invalidando la posición p.

> Ventajas y desventajas: ListaSE y la ListaDE

Listas con referencia al primer y último nodo: Ventajas y desventajas

- Estructura de datos: Igual a la cola con enlaces:
  - Lista simplemente enlazada
  - Se mantiene una referencia al primer y último elemento
- Ventajas:
  - *addLast(e)* y *last()* tienen orden 1.
- Desventajas:
  - Hay casos especiales cuando se elimina al principio y al final, cuando la lista mide 1 (igual que con la implementación de la cola enlazada).
  - *prev(p)* y *addBefore(p,x)* siguen siendo de orden lineal en la cantidad de elementos de la lista (hay que recorrer desde el comienzo).
  - *remove(p)* y *addAfter(p,x)* requieren un caso más cuando *p = last()* ya que requieren actualizar el rabo.

Listas doblemente enlazadas: Ventajas y desventajas

- Lista doblemente enlazada con referencia al primer y último nodo y dos celdas de encabezamiento (una al principio y otra al final).
- Ventajas:
  - Cada nodo conoce el siguiente nodo y al nodo anterior
  - Todas las operaciones del TDA PositionList tienen orden 1
  - Al usar celdas de encabezamiento las operaciones no tienen casos especiales (e.g., casos con referencias nulas).
- Desventajas:
  - Mayor uso de espacio porque para una lista que tiene n elementos hay n+2 celdas y cada celda tiene un enlace más.
- Leer: secciones 3.3 y 6.2.4 de Goodrich & Tamassia. Allí la lista está programada casi completamente.

10 – ITERADORES

Iteradores

- Un iterador es un patrón de diseño que abstrae el recorrido de los elementos de una colección
- Un iterador consiste de una secuencia S, un elemento corriente y una manera de avanzar al siguiente elemento de S haciéndolo el nuevo elemento corriente
- ADT Iterador (provisto por la interfaz java.util.Iterator):
  - hasNext(): Testea si hay elementos para seguir iterando
  - next(): Retorna el siguiente elemento
- ADT Iterable: Para poder ser iterable una colección debe brindar el método:
  - iterator(): Retorna un iterador para los elementos del la colección

```
import java.util.Iterator;
....
Iterator<E> it = colección.iterator();
while( it.hasNext() ) {
        E elem = it.next()
        ... procesar elem ...
}
```

La variable *colección* debe ser de un tipo que implementa la interfaz *java.lang.Iterable<E>*.

Si *colección* = [*a*, *b*, *c*, *d*, *e*], *elem* irá tomando los valores *a*, *b*, *c*, *d*, *e* a medida que itera el while.

Bucle for-each de Java

La sentencia for-each permite expresar un recorrido de una colección en alto nivel. Supongamos que "colección" es una expresión de un tipo que implementa la interfaz java.lang.Iterable<E>:

```
for( E elem : colección )
        sentencia(elem);
```

El for-each se lee "para cada elem de colección hacer sentencia(elem)" y permite ejecutar sentencia(elem) sobre cada elemento elem de colección. Además, el for-each es una abreviatura para este código:

```
Iterator<E> it = colección.iterator();
while( it.hasNext() )
{
        E elem = it.next()
        sentencia(elem);
}
```

> Ejemplos: Recorrer una lista

Listas iterables: Ejemplos

Hallar el máximo elemento de una lista de enteros positivos.

```
static int hallarMaximo( PositionList<Integer> lista )
{
        int maximo = 0;
        Iterator<Integer> it = lista.iterator();
        while ( it.hasNext() ) {
                Integer elem = it.next();
                if( elem > maximo )
                        maximo = elem;
        }
        return maximo;
}
```

Problema: Buscar un elemento x en una lista l.

Solución estructurada:

```
public static <E> boolean buscar(PositionList<E> l, E x) {
        Iterator<E> it = l.iterator();
        boolean encontre = false;
        while( it.hasNext() && !encontre )
          if( it.next().equals(x) )
                encontre = true;
        return encontre;
}
```

Solución no estructurada:

```
public static <E> boolean buscar(PositionList<E> l, E x) {
        for( E e : l )
          if( e.equals(x) ) return true;
        return false;
}
```

11 – TDA MAPEOS

> Operaciones: TDA Mapeo

TDA Mapeo (Map)

Dado un mapeo M:

- **size():** Retorna el número de entradas de M
- **isEmpty():** Testea si M es vacío
- **get(k):**
  - Si M contiene una entrada e con clave igual a k, retorna el valor de v;
  - sino retorna null
- **put(k,v):**
  - Si M no tiene una entrada con clave k, entonces agrega una entrada (k,v) a M y retorna null
  - Si M ya tiene una entrada e con clave k, reemplaza el valor con v en e y retorna el valor viejo de e.
- **remove(k):**
  - Remueve de M la entrada con clave k y retorna su valor
  - Si M no tiene entrada con clave k, retorna null.
- **keys():**
  - Retorna una colección iterable de las claves en M.
  - keys().iterator(): retorna un iterador de claves.
- **values():**
  - Retorna una colección iterable con los valores de las claves almacenadas en M
  - values().iterator(): retorna un iterador de valores
- **entries():**
  - Retorna una colección iterable con las entradas de M.
  - entries().iterator() retorna un iterador de entradas.

> Operaciones: TDA Diccionario

ADT Diccionario

Dado un diccionario no ordenado *D*:

- **size():** Retorna el número de entradas de *D*.
- **isEmpty():** Testea si *D* está vacío.
- **find(key):** Si *D* contiene una entrada *e* con clave igual a *key*, entonces retorna *e*, sino retorna *null*.
- **findAll(key):** Retorna una colección iterable conteniendo todas las entradas con clave igual a *key*.
- **insert(key,value):** Inserta en *D* una entrada *e* con clave *key* y valor *value* y retorna la entrada *e*.
- **remove(e):** Remueve de *D* la entrada *e*, retornando la entrada removida; ocurre un error si *e* no está en *D*.
- **entries():** Retorna una colección iterable con las entradas clave-valor de *D*.

> Operaciones: TDA Conjunto

Conjuntos

Un conjunto, caracterizado con el tipo Set<E>, tiene definidas las operaciones:

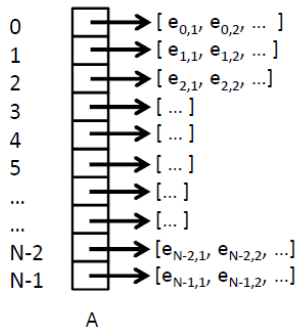
- insert(x): Inserta un elemento x el conjunto
- delete(x): Elimina el elemento x del conjunto
- member(x): Retorna true si x pertenece al conjunto
- intersection(S): Interseca al conjunto con otro conjunto S y retorna un nuevo conjunto intersección
- union(S): Une al conjunto con el conjunto S y retorna un nuevo conjunto union
- complement(): Calcula y retorna el complemento del conjunto (solo se puede implementar si el dominio es finito)
- iterator(): Retorna un iterador con los elementos del conjunto

12 – TABLAS DE HASH

> Tabla de hash abierto (separate chaining)

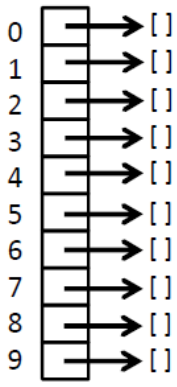
Hash Abierto: Definiciones

Arreglo de buckets: Un arreglo de cubetas para implementar una tabla de hash es un arreglo A de N componentes, donde cada celda de A es una colección de entradas (pares clave-valor) $e_{i,j}$.



Cada colección A[i] con i=0,...,N-1 se llama *bucket* o *cubeta*.

Nota simpática: *bucket* en inglés quiere decir *balde*.
Cuando los profesores éramos jóvenes, los únicos libros de computación editados en castellano eran de editoriales españolas, de ahí el término *cubeta*.

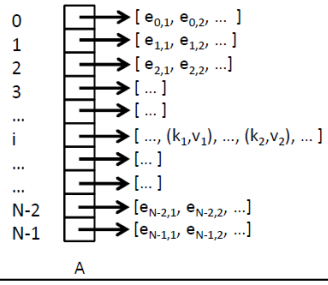


Hay 10 buckets (cubetas) numeradas de 0 a 9. Cada uno almacenará una colección de entradas. Cada bucket se inicializa con una lista vacía.

Colisión

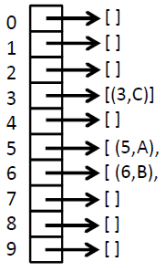
Colisión: Dadas dos claves k_1 y k_2 tales que $k_1 \neq k_2$, se produce una colisión cuando $h(k_1) = h(k_2)$.

Nota: Las entradas de una cubeta tienen claves colisionadas.



Si $h(k_1) = h(k_2) = i$, entonces las entradas (k_1,v_1) y (k_2,v_2) estarán en la misma cubeta A[i].

```
M ← new MapeoConHash();
M.put( 5, 'A' )
M.put( 6, 'B' )
M.put( 3, 'C' )
M.put( 15, 'D' )
M.put( 25, 'E' )
M.put( 35, 'F' )
M.put( 56, 'G' )
M.put( 25, 'H' )
```



Como h(25) = 25 mod 10 = 5, la entrada (25,E) es modificada a (25,H) en la lista A[5].

Factor de carga

Factor de carga

Supongamos n = 200 y N = 8, entonces $\lambda = n/N = 200/8 = 25$.

Esto quiere decir que cada bucket mide 25.

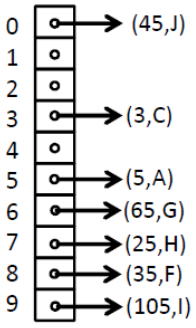
En el peor escenario, get, put y remove deben hacer 25 pasos, que es independiente de n y N, es decir es O(1) (con una constante muy grande).

> Tabla de hash cerrado (open addressing)

Hash Cerrado (open addressing)

- En hash cerrado se tiene un arreglo de N buckets
- Cada bucket almacena a lo sumo una entrada.
- Dada una clave k y un valor v, la función de hash h indica cuál es la componente h(k) del arreglo en la cual se almacena la entrada (k,v).
- Dadas dos claves k_1 y k_2 con $k_1 \neq k_2$, si $h(k_1)=h(k_2)$ entonces se produce una colisión.
- Políticas para la resolución de colisiones:
  - Lineal (linear probing)
  - Cuadrática (quadratic probing)
  - Hash doble (double hashing)

M ← new MapeoConHash()
M.put(5, 'A')
M.put(3, 'C')
M.put(15, 'D')
M.put(25, 'E')
M.put(35, 'F')
v ←M.get(25)
M.remove(15)
M.put(65, 'G')
M.put(25,'H'); M.put(105, 'I'); M.put(45, 'J')

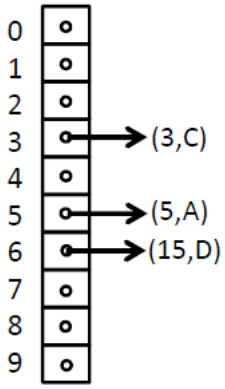


Estructuras de datos - Dr. Sergio A. Gómez

Colisiones

M ← new MapeoConHash()
M.put(5, 'A')
M.put(3, 'C')
M.put(15, 'D')

Como h(15) = 15mod 10 = 5 y la comp. 5 está ocupada, se produjo una colisión. Se busca una componente null o disponible en forma lineal y circular. Almaceno (15,D) en A[6].



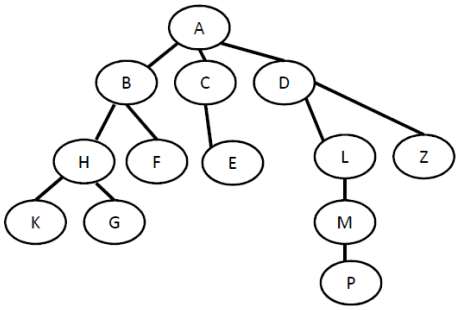
Estructuras de datos - Dr. Sergio A. Gómez

13 – TDA ÁRBOL

> Definición y relaciones

- Un árbol es un TDA que almacena los elementos llamados *nodos* jerárquicamente
- Con la excepción del nodo *raíz*, cada nodo en un árbol tiene un nodo padre y cero o más nodos hijos.
- La raíz se dibuja arriba.

El nodo con rótulo A es la raíz.
El nodo con rótulo B es el padre los nodos con rótulos H y F.
El nodo con rótulo F es el hijo del nodo con rótulo B.

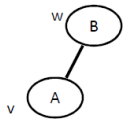


Definición formal de árbol

Un árbol T se define como un conjunto de nodos almacenando elementos tales que los nodos tienen una relación padre-hijo, que satisface:

- Si T es no vacío, tiene un nodo especial, llamado la raíz de T, que no tiene padre.
- Cada nodo v de T diferente de la raíz tiene un único nodo padre w
- Cada nodo v con padre w es un hijo de w.

A es el rótulo de v.
B es el rótulo de w.
Cuando no haya confusión, nos referiremos al nodo con su rótulo.



Relaciones entre nodos

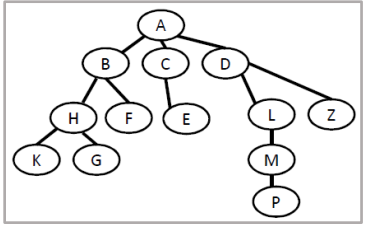
- Nodos Hermanos: Dos nodos con el mismo padre se llaman hermanos. Ej. B y D son hermanos, L y Z lo son, H y E no son hermanos (son primos).
- Nodos externos u hojas: Un nodo v es externo si no tiene hijos. Ej: F, K, G, E, P y Z son hojas.
- Nodo interno: Un nodo v es interno si tiene uno o más hijos. Ej: A, B, C, D, H, L y M son nodos internos.

- Ancestro(u,v): Un nodo u es ancestro de un nodo v si u=v o u es un ancestro del padre de v.

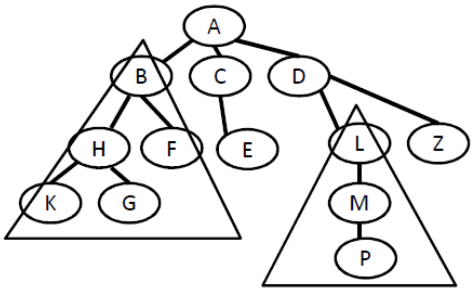
  Ej: ancestro(C,C), ancestro(A,E), ancestro(D,P)
- Ancestro propio(u,v): u es ancestro propio de v si u es ancestro de v y u ≠ v.

  Ej: acentropropio(A,E), ancestropropio(D, P), ancestropropio(D,L)

- Descendiente(u,v): Un nodo u es descendiente de un nodo v si v es un ancestro de u.
- Ej: descendiente(C,C), descendiente(E,A), descendiente(P,D)
- Descendiente propio(u,v): u es descendiente propio de v si u es descendiente de v y u ≠ v.
- Ej: descendientepropio(E,A).



- Subárbol: El subárbol de T con raíz en el nodo v es el árbol consistiendo de todos los descendientes de v.
- Ej: Mostramos dos subárboles con raíces B y L respectivamente.



> Operaciones

ADT Árbol

- Position:
  - element(): retorna el objeto almacenado en esta posición
- Tree: Métodos de acceso (reciben y retornan posiciones)
  - root(): Retorna la raíz del árbol, error si el árbol está vacío
  - parent(v): Retorna el padre de v, error si v es la raíz
  - children(v): Retorna una colección *iterable* conteniendo los hijos del nodo v

Nota: Si el árbol es ordenado, children los mantiene en orden.
Si v es una hoja children(v) está vacía.

- Tree: Métodos de modificación (agregados por la cátedra a [GT])
  - createRoot(e): crea un nodo raíz con rótulo *e*
  - addFirstChild(p, e): agrega un primer hijo al nodo *p* con rótulo *e*
  - addLastChild(p, e): agrega un último hijo al nodo *p* con rótulo e
  - addBefore(p, rb, e): Agrega un nodo con rótulo *e* como hijo de un nodo padre *p* dado. El nuevo nodo se agregará delante de otro nodo hermano *rb* también dado.
  - addAfter(p, lb, e): Agrega un nodo con rótulo *e* como hijo de un nodo padre *p* dado. El nuevo nodo se agregará detrás de otro hermano *lb* también dado.

- Tree: Métodos genéricos
  - size(): Retorna el número de nodos del árbol
  - isEmpty(): Testea si el árbol tiene o no nodos
  - iterator(): Retorna un iterador con los elementos ubicados en los nodos del árbol
  - positions(): Retorna una colección iterable de los nodos del árbol
  - replace(v,e): Reemplaza con e y retorna el elemento ubicado en v.

- Tree: Métodos de consulta
  - isInternal(v): Testea si v es un nodo interno
  - isExternal(v): Testea si v es una hoja
  - isRoot(v): Testea si v es la raíz

- Tree: Métodos de modificación (agregados por la cátedra a [GT])
  - removeExternalNode (p): Elimina la hoja *p*
  - removeInternalNode (p): Elimina el nodo interno *p*. Los hijos del nodo eliminado lo reemplazan en el mismo orden en el que aparecen. La raíz se puede eliminar si tiene un único hijo.
  - removeNode (p): Elimina el nodo *p*.

> Arboles ordenados

Árboles ordenados

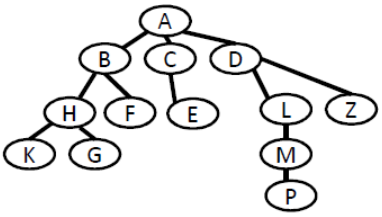
- Un árbol se dice ordenado si existe un orden lineal para los hijos de cada nodo,
- Es decir, se puede identificar el primer hijo, el segundo hijo y así sucesivamente
- Tal orden se visualiza de izquierda a derecha de acuerdo a tal ordenamiento.
- Ejemplo:
  - Los componentes de un libro:
  - El libro es la raíz,
  - parte, capítulo, sección, subsección, subsubsección, son los nodos internos
  - Los párrafos, figuras, tablas son las hojas.

-Árbol parcialmente ordenado: para cada nodo v distinto de la raíz, la clave almacenada en v es mayor o igual que la clave almacenada en el padre de v.

> Profundidad y altura

Profundidad y altura

- Profundidad de un nodo v en un árbol T: Longitud del camino de la raíz de T al nodo v = cantidad de ancestros propios de v
- Longitud de un camino: Cantidad de arcos del camino
- Altura de un nodo v en un árbol T: Longitud del camino más largo a una hoja en el subárbol con raíz v.
- Altura de un árbol T: Altura del nodo raíz de T.
- Ej: profundidad de A = 0
- Ej: profundidad de E = 2
- Ej: profundidad de P = 4
- Ej: Altura de B = 2
- Ej: Altura de D = 3
- Ej: Altura de G = 0
- Ej: Profundidad de G = 3



> Recorridos de árboles

Recorridos de árboles

- Un recorrido de un árbol T es una manera sistemática de visitar todos los nodos de T.
- Los recorridos básicos son:
  - Preorden (orden previo)
  - Postorden (orden posterior)
  - Inorden (orden simétrico)
  - Por niveles (level ordering)

Preorden (orden previo)

Recorridos de árboles: Preorden

- En el recorrido preorden de un árbol T, la raíz r de T se visita primero y luego se visitan recursivamente cada uno de los subárboles de r.
- Si el árbol es ordenado los subárboles se visitan en el orden en el que aparecen.
- El algoritmo se llama con preorden(T, T.root())
- Algoritmo preorden(T, v)
  Visitar(T, v)
  Para cada hijo w de v en T hacer
  preorden(T, w)
- La acción "visitar(T,v)" dependerá del problema.

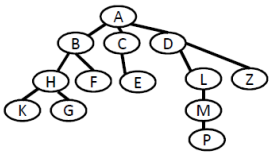
- Planteo Preorden:
  CB: pre(hoja(n)) = [n]
  CR: pre(nodo(n, [t_1 t_2 ... t_k])) =
  [n] + pre(t_1) + pre(t_2) + ... + pre(t_k)

Recorridos en árboles: preorden

Algoritmo PreordenShell(T)
preorden(T, T.root())

Algoritmo preorden(T, v)
Visitar(T, v)
Para cada hijo w de v en T hacer
preorden(T, w)

- Planteo Preorden:
  CB: pre(hoja(n)) = [n]
  CR: pre(nodo(n, [t_1 t_2 ... t_k])) =
  [n] + pre(t_1) + pre(t_2) + ... + pre(t_k)



Ejemplo: El recorrido preorden es:

A – B - H - K - G - F - C - E - D - L - M - P - Z

Tiempo de ejecución de preorden

- Entrada: un árbol T
- Tamaño de la entrada: n = cantidad de nodos del árbol T
- Tiempo de ejecución: Vemos que el algoritmo pasa una vez por cada nodo *i* y en el nodo toma un tiempo constante c_2 y luego ejecuta un bucle que realiza h_i iteraciones, con h_i = la cantidad de hijos del nodo i.

$T(n) = c_1 + \sum_{i=1}^{n}(c_2 + c_3 h_i) =$

$= c_1 + c_2 n + c_3(n-1) = O(n)$

Notar que $\sum_{i=1}^{n} h_i = n - 1$ pues corresponde a la cantidad de arcos del árbol.

La expresión $\sum_{i=1}^{n} h_i$ corresponde a la cantidad de hijos del nodo1 más la cantidad de hijos del nodo2 más la cantidad de hijos del nodo3 y así sucesivamente, lo que da la cantidad de arcos del árbol. Como todos los nodos tienen un único padre menos la raíz que no tiene, entonces si hay *n* nodos, entonces la cantidad de arcos del árbol es *n-1*.

Postorden (orden posterior)

Recorridos de árboles: Postorden

- En el recorrido postorden de un árbol T, la raíz r de T se visita luego de visitar recursivamente cada uno de los subárboles de r.
- Si el árbol es ordenado los subárboles se visitan en el orden en el que aparecen.
- El algoritmo se llama con postorden(T, T.root())
- Algoritmo postorden(T, v)

```
Para cada hijo w de v en T hacer
    postorden( T, w )
Visitar(T, v)
```

- Planteo Postorden:

CB: post(hoja(n)) = [n]

CR: post(nodo(n, [t_1 t_2 ... t_k])) =

post(t_1) + post(t_2) + ... + post(t_k) + [n]

- La acción "visitar(T,v)" dependerá del problema.

Recorridos en árboles: postorden

Algoritmo PostordenShell(T)

```
postorden( T, T.root() )
```

Algoritmo postorden(T, v)

```
Para cada hijo w de v en T hacer
    postorden( T, w )
Visitar(T, v)
```

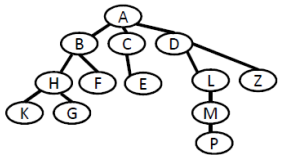
- Planteo Postorden:

CB: post(hoja(n)) = [n]

CR: post(nodo(n, [t_1 t_2 ... t_k])) =

post(t_1) + post(t_2) + ... + post(t_k) + [n]

Ejemplo: El recorrido postorden es:
K, G, H, F, B, E, C, P, M, L, Z, D, A



Tiempo de ejecución: Si el árbol T tiene n nodos entonces:
$T_{postorden}(n) = O(n)$ asumiendo visita de O(1)

Inorden (orden simétrico)

Recorridos de árboles: Inorden

- En el recorrido inorder (o simétrico) de un árbol T con raíz r, primero se recorre recursivamente el primer hijo de la raíz r, luego se visita la raíz y luego se visita recursivamente al resto de los hijos de r.
- Si el árbol es ordenado los subárboles se visitan en el orden en el que aparecen.
- El algoritmo se llama con inorden(T, T.root())
- Algoritmo inorden(T, v)

```
Si v es hoja en T entonces
    Visitar( T, v )
Sino
    w ← primer hijo de v en T
    inorden( T, w )
    Visitar(T, v)
    mientras w tiene hermano en T hacer
        w ← hermano de w en T
        inorden( T, w )
```

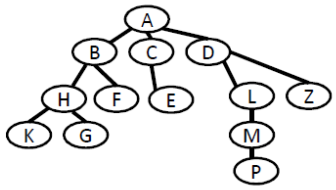
- El recorrido inorden para el ejemplo es:
  K H G B F A E C P M L D Z

- Planteo In order:

CB: in(hoja(n)) = [n]

CR: in(nodo(n, [t_1 t_2 ... t_k])) =

in(t_1) + [n] + in(t_2) + ... + in(t_k)

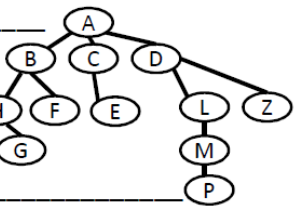


Por niveles (level ordering)

Recorridos de árboles: por niveles

- Nivel: Subconjunto de nodos que tienen la misma profundidad
- Recorrido por niveles (level numbering): Visita todos los nodos con profundidad p antes de recorrer todos los nodos con profundidad p+1.
- Ejemplo: El recorrido/listado por niveles es:

A *nivel 0*
B C D *nivel 1*
H F E L Z *nivel 2*
K G M *nivel 3*
P *nivel 4*




```
Algoritmo niveles( T )
Cola ← new Cola()
Cola.enqueue( T.root() )
Mientras not cola.isEmpty()
    v ← cola.dequeue()
    mostrar v  // visita de v
    para cada hijo w de v en T hacer
        cola.enqueue( w )
```

Tiempo de ejecución de listado por niveles

```
Algoritmo niveles( T )
Cola ← new Cola()   cte
Cola.enqueue( T.root() )
Mientras not cola.isEmpty() n vueltas, condición cte
    v ← cola.dequeue() cte
    mostrar v  cte
    para cada hijo w de v en T hacer    h_i vueltas
        cola.enqueue( w )  cte
```

- Entrada: árbol T
- Tamaño de la entrada: n = cantidad de nodos del árbol T
- Tiempo de ejecución:

Sea h_i la cantidad de hijos del nodo i

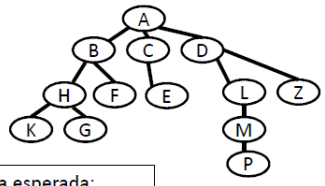
$T(n) = c_1 + \sum_{i=1}^{n}(c_2 + c_3 h_i) =$

$= c_1 + c_2 n + c_3(n-1) = O(n)$

Recordar que *la suma de la cantidad de hijos de todos los nodos* es n-1, porque es igual a la cantidad de padres en el árbol y todos los nodos tienen exactamente un padre menos la raíz que no tiene.

Recorridos por niveles mostrando dónde termina cada nivel

```
Algoritmo niveles( T )
Cola ← new Cola()
Cola.enqueue( T.root() )
Cola.enqueue( null )
Mientras not cola.isEmpty()
    v ← cola.dequeue()
    si v ≠ null entonces
        mostrar v
        para cada hijo w de v en T hacer
            cola.enqueue( w )
    sino
        imprimir fin de línea
        si not cola.isEmpty() entonces
            cola.enqueue( null )
```



Salida esperada:
A
B C D
H F E L Z
K G M
P

14 – TDA ÁRBOL BINARIO (AB)

> Definición

Árboles binarios

- Un árbol binario es un árbol ordenado que cumple:
  1) Cada nodo tiene a lo sumo dos hijos
  2) Cada nodo hijo es o bien hijo izquierdo o hijo derecho
  3) El hijo izquierdo precede al hijo derecho en el orden de los hijos de un nodo
- El subárbol que tiene como raíz al hijo izquierdo se llama subárbol izquierdo.
- El subárbol que tiene como raíz al hijo derecho se llama subárbol derecho.

> Árbol propio/impropio

- En un árbol binario propio, cada nodo tiene 0 o dos hijos (GT también le dice full BT).
- Si un árbol binario no es propio, entonces es impropio.

> Operaciones

ADT Arbol Binario [GT]

El Árbol Binario (de acuerdo a GT) es una especialización (subinterfaz) de Tree que además soporta los métodos de acceso adicionales:

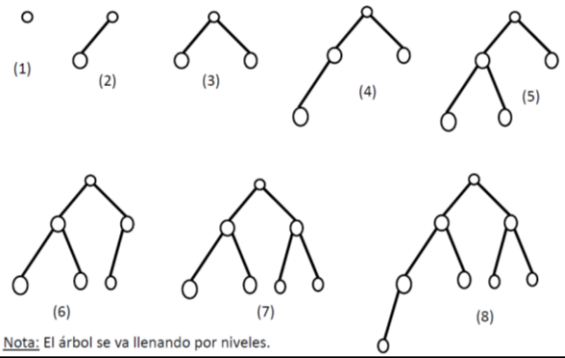
- left(v): Retorna el hijo izquierdo de v, ocurre error si v no tiene hijo izquierdo
- right(v): Retorna el hijo derecho de v, ocurre error si v no tiene hijo derecho
- hasLeft(v): Testea si v tiene hijo izquierdo
- hasRight(v): Testea si v tiene hijo derecho

Implementación del árbol binario (cont.)

- Métodos de modificación:
  - addRoot(e) (o createRoot(e)): Agrega un nodo raíz con rótulo e, error si ya hay raíz
  - insertLeft(v, e): Crea un nodo w hijo izquierdo de v con rótulo e, error si v ya tiene hijo izquierdo
  - insertRight(v,e): Crea un nodo w hijo derecho de v con rótulo e, error si v ya tiene hijo derecho
  - remove(v): Elimina el nodo v (si v tiene un hijo, reemplaza a v por su hijo, si v tiene dos hijos entonces error).
  - attach(v, T_1, T_2): Setea T_1 como hijo izq de v y T_2 como hijo derecho de v (error si v no es hoja).

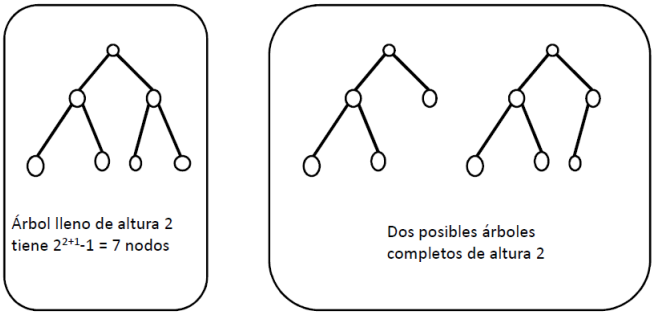
> Árbol Binario completo/lleno

Ejemplos de árboles binarios completos



Nota: El árbol se va llenando por niveles.

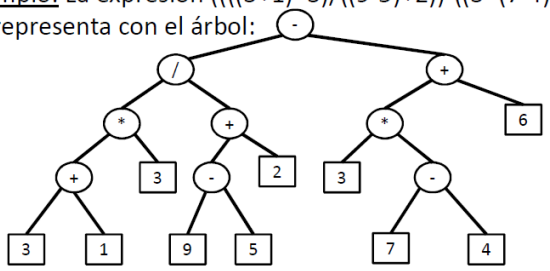
Árbol lleno versus árbol completo



Propiedad: Un árbol binario lleno de altura h tiene $n = 2^{h+1}-1$ nodos.

Aplicaciones: Expresiones aritméticas

- Una expresión aritmética puede representarse con un árbol binario.
- Las hojas almacenan constantes o variables
- Los nodos internos almacenan los operadores +, -, * y /.
- Ejemplo: La expresión ((((3+1)*3)/((9-5)+2))-((3*(7-4))+6)) se representa con el árbol:



Obtención del árbol binario de expresión aritmética

Ejemplo: exp = 5 + 6 – 7 (expresión solo formada por + y -)

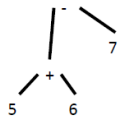
Algoritmo Parse(exp)
Si exp no contiene un operador **entonces**
  retornar un árbol binario hoja conteniendo rótulo de exp
Sino
  op ← último_operador(exp)
  T_1 ← Parse(izquierda(exp, op))
  T_2 ← Parse(derecha(exp, op))
  retornar un árbol binario con op como rótulo de la raíz y T_1 y T_2 como hijos izquierdo y derecho resp.

Nota:
5 + 6 - 7 = (5 + 6) – 7 = =11 – 7 = 4 *ok*
5 + (6-7) = 5 + (-1) = 4
7 – 2 – 4 = (7 - 2) – 4 = = 5 – 4 = 1 *ok*
7-(2-4)=7-(-2)=7+2=9 *que no es lo que quiero.*

Ultimo_operador(exp) = busca el último operador de la expresión exp recorriendo de derecha a izquierda. En el caso de exp=5+6-7 retorna -.
Izquierda(exp,op) = retorna la porción izquierda de exp considerando la ubicación de op. En el caso de exp=5+6-7 y op=- retorna 5+6.
Derecha(exp,op) = retorna la porción derecha de exp considerando la ubicación de op. En el caso de exp=5+6-7 y op=- retorna 7

Obtención del árbol binario de expresión aritmética sin paréntesis

Ejemplo: exp = 5 + 6 – 7 (expresión solo formada por + y -)
Algoritmo Parse(exp)
Si exp no contiene un operador **entonces**
  retornar un árbol binario hoja conteniendo rótulo de exp
Sino
  op ← último_operador(exp)
  T_1 ← Parse(izquierda(exp, op))
  T_2 ← Parse(derecha(exp, op))
  retornar un árbol binario con op como rótulo de la raíz y T_1 y T_2 como hijos izquierdo y derecho resp.



Parse(5+6-7)
exp=5+6-7
op=-
Izq(exp,op) =5+6
Der(exp,op) = 7
Resultado=t5=AB(-,t3,t4)

Parse(5+6)
Exp=5+6
Op=+
Izq(exp,op)=5
Der(exp,op)=6
Resultado=t3=AB(+,t1,t2)

Parse(5)
Exp=5
Resultado=t1=AB(5)

Parse(6)
Exp=6
Resultado=t2=AB(6)

Parse(7)
Exp=7
Resultado=t4=AB(7)

Obtención del árbol binario de expresión aritmética totalmente parentizada

Algoritmo Parse(exp)
Si exp no contiene una operador **entonces**
  retornar un árbol binario hoja conteniendo rótulo de exp
Sino
  exp ← Eliminar primer y último paréntesis de exp
  op ← último_operador(exp)
  T_1 ← Parse(izquierda(exp, op))
  T_2 ← Parse(derecha(exp, op))
  retornar un árbol binario con op como rótulo de la raíz y T_1 y T_2 como hijos izquierdo y derecho resp.

Evaluación de un árbol de expresión aritmética

Cada nodo en un árbol de expresión tiene un valor asociado:

- Si el nodo es externo, su valor es el de la variable o constante
- Si el nodo es interno, su valor está definido por la aplicación de la operación a los valores de sus hijos.

Algoritmo Evaluar(arbol_exp)
Si arbol_exp es una hoja **entonces**
  retornar el valor del rótulo de arbol_exp
Sino
  op ← rótulo de raíz de arbol_exp
  v_1 ← Evaluar(hijo izquierdo de arbol_exp)
  v_2 ← Evaluar(hijo derecho de arbol_exp)
  retornar aplicar(op, v_1, v_2) // Aplicar realiza la operación v_1 op v_2

En el ejemplo solo tenemos constantes. Si queremos tener variables, necesitamos un mapeo de string en número, para recuperar el valor de cada referencia a una variable.

Formatos para expresiones aritméticas

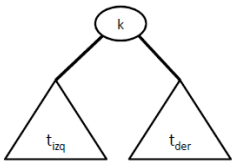
- Infija: El operador va al medio de los operandos
  E → N | E_1 + E_2 | E_1 - E_2 | E_1 * E_2 | E_1 / E_2
  N → 0 | 1 | … | 9
- Prefija: El operador va delante de los operandos (notación polaca)
  E → N | + E_1 E_2 | - E_1 E_2 | * E_1 E_2 | / E_1 E_2
- Posfija: El operador va detrás de los operandos (notación polaca inversa)
  E → N | E_1 E_2 + | E_1 E_2 -| E_1 E_2 *| E_1 E_2 /

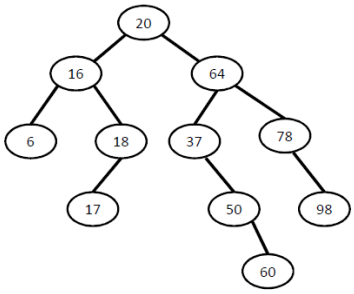
15 – ÁRBOL BINARIO DE BÚSQUEDA (ABB)

> Definición

Definición

- Un ABB es un árbol binario tal que:
  - Es vacío, o,
  - Es un nodo con rótulo k e hijos t_{izq} y t_{der} tales:
    - k > claves de t_{izq},
    - k < claves de t_{der}, y,
    - t_{izq} y t_{der} son ABB.

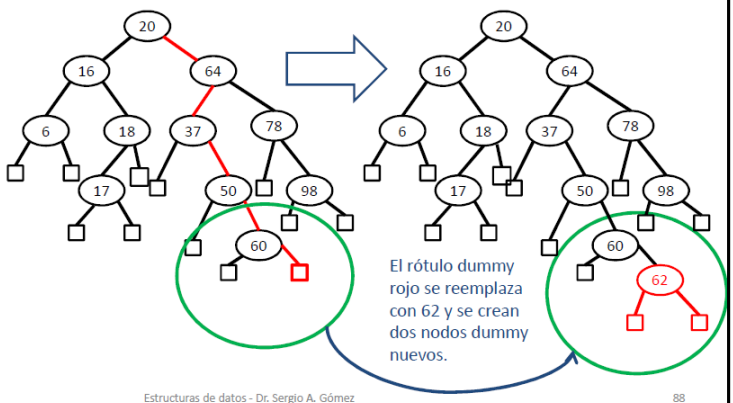




- El árbol binario de búsqueda (ABB) es una estructura de datos útil para implementar conjuntos, mapeos y diccionarios.
- En un ABB las claves en los nodos se hallan ordenadas de una manera particular.
- El tiempo de insertar, recuperar y eliminar es proporcional a la altura del ABB.
- Si el árbol tiene n claves, la altura del ABB se halla entre $\log_2(n)$ y n.

> Inserción y búsqueda

Supongamos que deseamos insertar el 62, para ello buscamos su ubicación:



Inserción en un ABB

- Las nuevas claves siempre se insertan como hijo de un nodo hoja.
- Algoritmo insertar(k, p) { Comienza en la raíz del ABB }

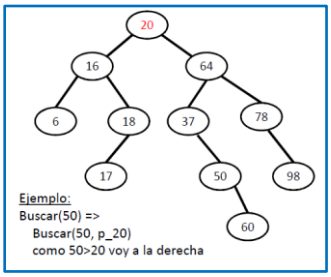
```
Si p es vacío entonces
    crear un nodo hoja con rótulo k
Sino
    si k < clave(p) entonces
            insertar(k, hijo izquierdo de p)
    sino si k > clave(p) entonces
            insertar( k, hijo derecho de p )
    sino si k = clave(p) entonces
            reemplazar rótulo de p
```

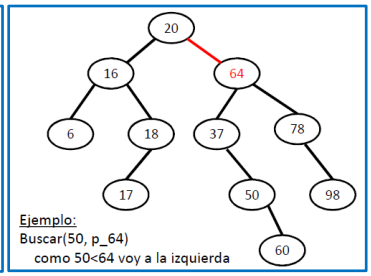
Nota: La operación "reemplazar rótulo de p" dependerá de si:
(a) Implemento un conjunto: no hacer nada
(b) Implemento un mapeo: modifico el valor de la entrada
(c) Implemento un diccionario: Agrego una entrada con clave k

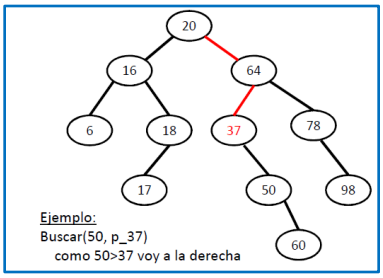
Búsqueda de rótulos

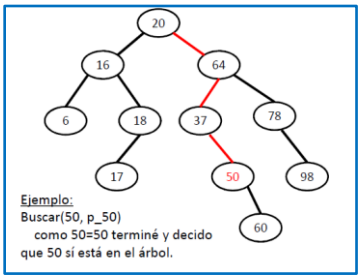
La búsqueda usa el ordenamiento de los rótulos para podar el árbol.

Buscar(x, p):
CB: Si estoy en un nodo vacío p => x no está en el árbol
CB: Si estoy en un nodo no vacío p y x=rótulo(p) => x está en el árbol
CR: Si estoy en un nodo no vacío p y x<rótulo(p) => Buscar(x,p) = Buscar(x, hijo izquierdo de p)
CR: Si estoy en un nodo no vacío p y x>rótulo(p) => Buscar(x,p) = Buscar(x, hijo derecho de p)

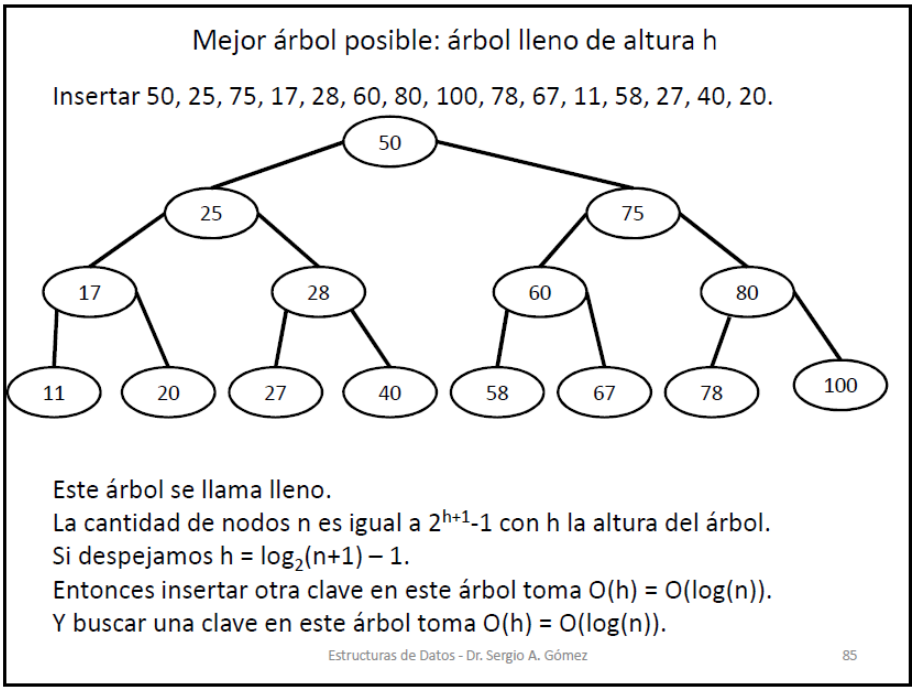




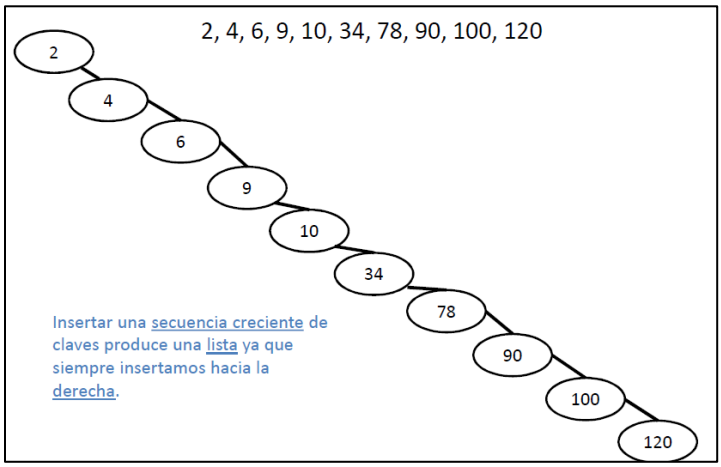


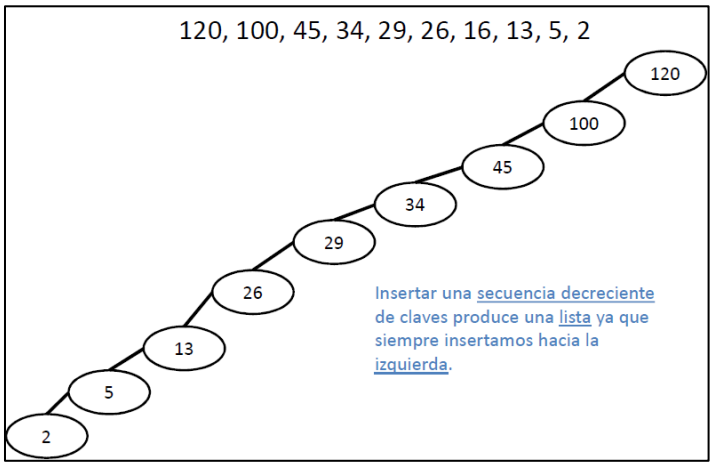


> Mejor árbol posible: ABB lleno



> Casos especiales





> Complejidad temporal

Análisis de la complejidad temporal

- Sea h = altura del ABB y sea n = cantidad de claves del ABB.

$$T(h) = \begin{cases} c_1 & si \quad h = 0 \\ c_2 + T(h-1) & si \quad h > 0 \end{cases}$$

con lo cual $T(h) = O(h)$.

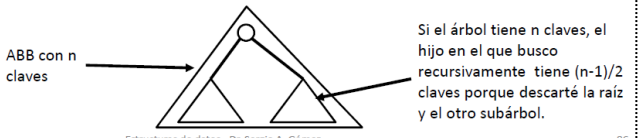
- La altura en el peor caso es $h = n - 1 = O(n)$ (se da cuando se hicieron inserciones de claves en forma ascendente o descendente)
- La altura en el mejor caso es $h = O(log_2(n))$ ya que las inserciones producen un árbol lleno que tiene $n = 2^{h+1-1}$ nodos; de ahí, despejo $h = log_2(n+1) - 1$.
- El árbol lleno de altura *h* es aquel que todos los nodos internos tienen 2 hijos y todas las hojas tienen la misma profundidad *h*.

Análisis de la complejidad temporal

- Otra forma de estudiar el peor caso del mejor caso del ABB (que se da cuando el árbol está lleno y no encuentro la clave buscada):
- Sea n = cantidad de claves del ABB.

$$T(n) = \begin{cases} c_1 & si \quad n = 0 \\ c_2 + T((n-1)/2) & si \quad n > 0 \end{cases}$$

con lo cual $T(n) = O(log_2(n))$.



16 – TDA COLA CON PRIORIDAD

> Definición y operaciones

ADT Cola con prioridad

- Una cola con prioridad almacena una colección de elementos que soporta:
  - Inserción de elementos arbitraria
  - Eliminación de elementos en orden de prioridad (el elemento con 1era prioridad puede ser eliminado en cualquier momento)
- Nota: Una cola con prioridad almacena sus elementos de acuerdo a su prioridad relativa y no expone una noción de "posición" a sus clientes.

- **size():** Retorna el número de entradas en P.
- **isEmpty():** Testea si P es vacía
- **min():** Retorna (pero no remueve) una entrada de P con la prioridad más pequeña; ocurre un error si P está vacía.
- **insert(k,x):** Inserta en P una entrada con prioridad k y valor x; ocurre un error si k es inválida (e.g. k es nula).
- **removeMin():** Remueve de P y retorna una entrada con la prioridad más pequeña; ocurre una condición de error si P está vacía.

> Método de ordenamiento (Heap)

Cola con prioridad implementada con Heap

Un **(mín)heap** es un árbol binario que almacena una colección de entradas en sus nodos y satisface dos propiedades adicionales:

- **Propiedad de orden del heap (árbol parcialmente ordenado):** En un heap T, para cada nodo v distinto de la raíz, la clave almacenada en v es mayor o igual que la clave almacenada en el padre de v.
- **Propiedad de árbol binario completo:** Un heap T con altura h es un árbol binario completo si los nodos de los niveles 0,1,2,…,h-1 tienen el máximo número de nodos posibles y en el nivel h-1 todos los nodos internos están a la izquierda de las hojas y si hay un nodo con un hijo, éste debe ser un hijo izquierdo *(y el nodo debiera ser el nodo interno de más a la derecha)*.

Aplicación: Heap Sort

- Objetivo: Ordenar un arreglo A de N enteros en forma ascendente
- Estrategia: Insertar los n elementos del arreglo en un heap inicialmente vacío y luego eliminarlos de a uno y almacenarlos en el arreglo.
- Algoritmo HeapSort(a, n)

```
cola ← new ColaConPrioridad()
para i ← 0..n-1 hacer
    cola.insert( a[i] )
para i ← 0..n-1 hacer
        a[i] ← cola.removeMin()
```

Ejemplo de uso

```
public class Principal {
   public static void main(String[] args) {
    // Creo una cola con prioridad implementada con un Heap
    // con prioridades de tipo entero y valores de tipo string.
    // El constructor recibe el tamaño y el comparador de prioridades.
    PriorityQueue<Integer, String> cola = new Heap<Integer, String>( 20,
            new DefaultComparator<Integer>() );
    try {
            cola.insert(40, "Sergio");    // Inserto a Sergio con prioridad 40.
            cola.insert(30, "Martin");    // Inserto a Martín con prioridad 30.
            cola.insert(15, "Matias");    // Inserto a Matías con prioridad 15.
            cola.insert(5, "Carlos");     // Inserto a Carlos con prioridad 5.
            cola.insert(100, "Marta");    // Inserto a Marta con prioridad 100.
            // Imprimo la entrada con mínima prioridad: (5, Carlos).
            System.out.println( "Min: " +  cola.min() );
            // Vacío la cola: puede lanzar EmptyPriorityQueueException
            while ( !cola.isEmpty() ){
                    Entry<Integer, String> e = cola.removeMin();
                    System.out.println( "Entrada: " + e);
            } // Salen las prioridades: 5, 15, 30, 40 y 100 en ese orden.
   } catch(InvalidKeyException | EmptyPriorityQueueException e) {
            e.printStackTrace();
    }
   }
}
```

Complejidad temporal de Heap Sort

Tamaño de la entrada:
n = cantidad de componentes de a

```
Algoritmo HeapSort( a, n )
cola ← new ColaConPrioridad()  c1
para i ← 0..n-1 hacer  Realiza n iteraciones
  cola.insert( a[i] )  la iteración i cuesta c2log2(i)
para i ← 0..n-1 hacer Realiza n iteraciones
  a[i] ← cola.removeMin() la iteración i cuesta c3log2(n-i)
```

Complejidad:

$T_{heapsort}(n) = c_1 + c_2 n\log_2(n) + c_3 n\log_2(n) = O(n\log_2(n))$

$SPACE_{heapsort}(n) = O(n)$ porque usa una estructura auxiliar (la heap) de tamaño n

Recordar que $SPACE_A(n)$ es la cantidad de memoria extra que usa el algoritmo A para resolver el problema de tamaño n

> Default Comparator

ADT Comparador

Problema: ¿Cómo comparar claves de tipo genérico K?

compare(a,b) = Retorna un entero i tal que:

- i<0, si a<b
- i=0, si a=b
- i>0, si a>b

Ocurre un error si a y b no pueden ser comparados.

Implementación

Comparador por defecto

El comparador por defecto delega su comportamiento en el comportamiento de la operación compareTo del tipo básico E:

```
public class DefaultComparator<E extends Comparable<E>>
   implements java.util.Comparator<E> {
        public int compare( E a, E b ) {
                return a.compareTo( b );
        }
}
```

Ejemplo

Ejemplo de Comparador

```
public class Persona {   // Archivo: Persona.java
        protected String nombre;
        protected float peso;
        public Persona(String nombre; float peso ) { … }
        public float getPeso() { return peso; }
        … otras operaciones…
}

// Archivo: ComparadorPersona.java
public class ComparadorPersona<E extends Persona>
        implements java.util.Comparator<E> {
   public int compare( E a, E b ) {   // Comparo las personas por su peso
      return  (int) (a.getPeso()  - b.getPeso());
   } Notar que cuando a pesa menos que b, se retorna un negativo; si pesan
(más o menos) lo mismo, se retorna 0 y si a pesa más que b, se retorna un
positivo. Pensar cómo programar la operación con if's anidados.
}
```

Crear un objeto comparador

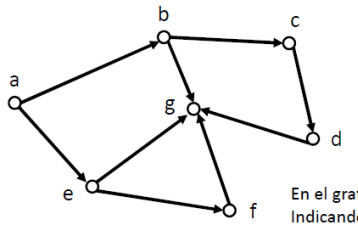
```
DefaultComparator<E> comp = new DefaultComparator<E>();
```

17 – TDA GRAFO

> Definiciones: Grafo dirigido y grafo no dirigido

Definición: Grafo dirigido o digrafo

- Un grafo G=(V,E) es un conjunto V de vértices (o nodos) y un conjunto E ⊆ V×V de aristas dirigidas (o arcos dirigidos).
- Intuitivamente E permite representar una relación entre elementos de V.

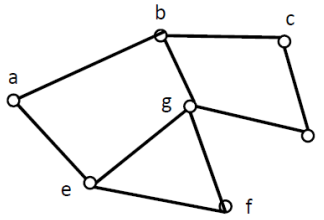


V = { a, b, c, d, e, f, g }
E = { (a,b), (b,c), (c,d), (d,g), (e,g), (e,f), (f,g), (a,e), (b,g) }

En el grafo dirigido, el arco (u,v) es un par ordenado Indicando una flecha del vértice u al vértice v.

Definición: Grafo no dirigido

- Un grafo G=(V,E) es un conjunto V de vértices (o nodos) y un conjunto E ⊆ V×V de aristas (o arcos).
- Intuitivamente E permite representar una relación (simétrica) entre elementos de V.



V = { a, b, c, d, e, f, g }
E = { (a,b), (b,c), (c,d), (d,g), (e,g), (e,f), (f,g), (a,e), (b,g) }

En el grafo no dirigido, el arco (u,v) en realidad es un conjunto {u,v} ya que (u,v) es lo mismo que (v,u).

> Operaciones: Grafo no dirigido

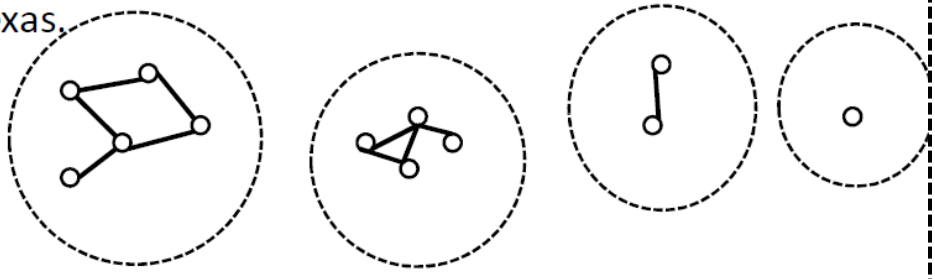
- vertices(): Retorna una colección iterable con todos los vértices del grafo.
- edges(): Retorna una colección iterable con todos los arcos del grafo.
- replace(v,x) : Reemplaza el rótulo del vértice v con x
- replace(e,x): Reemplaza el rótulo del arco e con x
- insertVertex(x): Inserta y retorna un nuevo vértice con rótulo x
- insertEdge(v, w, x): Inserta un arco con rótulo x entre los vértices v y w
- opposite(v,e): Retorna el otro vértice w del arco e=(v,w); ocurre un error si e no es incidente (o emergente de v).
- endVertices(e): Retorna un arreglo (de 2 componentes) conteniendo los vértices del arco e.
- areAdjacent(v,w): Testea si los vértices v y w son adyacentes.

incidentEdges(v): Retorna una colección iterable con todos los arcos incidentes sobre un vértice v

> Componentes conexas

Definiciones

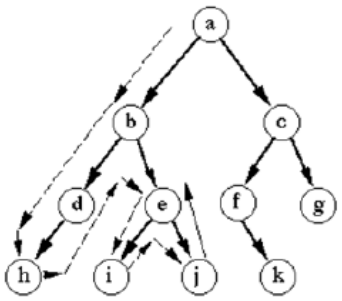
- Grafo conexo: Un grafo G es conexo si para dos vértices cualquiera de G hay un camino entre ellos.
- Componentes conexas: Si un grafo no es conexo, sus subgrafos maximales conexos se llaman componentes conexas.



Este grafo no es conexo porque tiene 4 componentes conexas.

> Recorridos de grafos y algoritmos especiales

Depth-First Search o DFS



Depth-first search

- En profundidad (Depth-First Search o DFS): (Equivale al recorrido pre o post orden en árboles + un testeo para no volver a recorrer un subgrafo ya explorado): a, b, d, h, e, i, j, c, f, k, g

Búsqueda en profundidad (DFS)

- Una búsqueda en profundidad (DFS o Depth-First Search) permite recorrer todos los vértices de un grafo de manera ordenada.
- Su funcionamiento consiste en ir expandiendo todos y cada uno de los nodos que va localizando, de forma recurrente, en un camino concreto.
- Cuando ya no quedan más nodos que visitar en dicho camino, regresa (backtracking), de modo que repite el mismo proceso con cada uno de los hermanos del nodo ya procesado.

Análisis del tiempo de ejecución

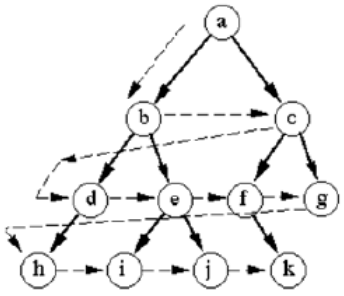
- Sea un grafo G=(V,A)
- Sean n = #V y m = #A (#S quiere decir el cardinal de S)
- Para simplificar el análisis, supongamos que el grafo es conexo.
- El algoritmo visita 1 vez cada vértice y en cada vértice i recorre su lista de adyacentes que mide A_i, *entonces:*

$$\begin{aligned} T(n,m) &= c_1 + \sum_{i=1}^{n}\left(c_2 + \sum_{j=1}^{A_i} c_3\right) \\ &= c_1 + \sum_{i=1}^{n}(c_2 + A_i c_3) \\ &= c_1 + \sum_{i=1}^{n} c_2 + \sum_{i=1}^{n} A_i c_3 \\ &= c_1 + \sum_{i=1}^{n} c_2 + c_3 \sum_{i=1}^{n} A_i \\ &= c_1 + nc_2 + mc_3 \\ &\leq (n+m)c_1 + (n+m)c_2 + (n+m)c_3 \\ &= (n+m)(c_1 + c_2 + c_3) = O(n+m) \end{aligned}$$

Note que $m = O(n^2)$

Nota: $\sum_{i=1}^{n} A_i$ $= A_1 + A_2 + \ldots + A_n = m$

Breadth-First Search o BFS



Breadth-first search

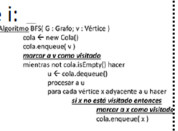
- En anchura (Breadth-First Search o BFS): (Equivale al recorrido por niveles en árboles + un testeo para no volver a recorrer un subgrafo ya explorado): a, b, c, d, e, f, g, h,i, j, k

Búsqueda en anchura (BFS)

- La búsqueda en anchura (BFS o Breadth First Search) es un algoritmo para recorrer o buscar elementos en un grafo.
- Se comienza eligiendo algún nodo como elemento raíz y se exploran todos los vecinos de este nodo.
- A continuación para cada uno de los vecinos se exploran sus respectivos vecinos adyacentes, y así hasta que se recorra todo el grafo.

Análisis del tiempo de ejecución

- Sea un grafo G=(V,A)
- Sean n = #V y m = #A (#S quiere decir el cardinal de S)
- Para simplificar el análisis, supongamos que el grafo es conexo.
- Sea A_i la cantidad de adyacentes del vértice i:



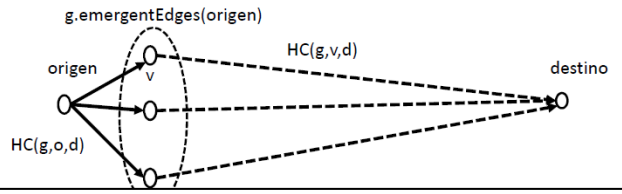
$$\begin{aligned} T(n,m) &= c_1 + \sum_{i=1}^{n}\left(c_2 + \sum_{j=1}^{A_i} c_3\right) \\ &= c_1 + \sum_{i=1}^{n}(c_2 + A_i c_3) \\ &= c_1 + \sum_{i=1}^{n} c_2 + \sum_{i=1}^{n} A_i c_3 \\ &= c_1 + nc_2 + mc_3 = O(n+m) \end{aligned}$$

Note que $m = O(n^2)$

St-path search (DFS pinchado)

st-path search: Hallar camino de o a d en g

- Planteo recursivo:
- HallarCamino(g: Grafo, o : Vertice, d : Vertice) : Boolean
- CB: Si o=d => HallarCamino(g,o,d) = true
- CB: Si o≠d y o no tiene adyacentes => HallarCamino(g,o,d) = false
- CR: Si o ≠ d y o tiene adyacentes => HallarCamino(g,o,d) = (∃v ∈ g.emergentEdges(o)).(HallarCamino(g,v,d) = true)



DFS pinchado (s-t path search)

Permite hallar un camino en el digrafo G entre Origen y Destino y retorna el camino hallado en Camino (el cual es una lista vacía al principio).
Si encontró un camino, retorna verdadero, en caso contrario retorna falso.

```
Algoritmo HallarCamino( G, origen, destino, camino ) : boolean
origen.put( Estado, Visitado ) { marco a origen como visitado }
camino.addLast( origen ) { encolo a origen en el camino }
Si origen = destino entonces { CB positivo }
    retornar verdadero { encontré el camino }
Sino {CB negativo y CR }
    para cada adyacente v de origen en G hacer
        si v.get( Estado ) = NoVisitado entonces
            encontre ← HallarCamino( G, v, destino, camino )
            si encontre entonces retornar verdadero { ¡Retornar rompe el para! }
{ Exploré todos los adyacentes y no encontré el camino, ú origen no tiene adyacentes
=> hago backtracking: borro a origen del camino y retorno falso. }
camino.remove( camino.last() )
retornar falso
```

Complejidad: T(n,m) = O(n+m)

Ejercicio: Codificar en Java

Dijkstra

Algoritmo de Dijkstra

- Suponga un digrafo G tal que cada arco tiene costo no negativo, Dijkstra computa los caminos de costo mínimo desde un vértice *a* a todos los otros vértices de G.
- El vértice *a* se conoce como la *fuente*.
- El costo del camino es la suma de los pesos de los arcos del camino.
- El algoritmo mantiene un conjunto S con los vértices cuyo camino con la distancia más corta es conocida.
- S inicialmente está vacío.
- En cada paso se agrega a S el vértice *u* cuya distancia a la fuente es tan cercana como es posible.
- El algoritmo termina cuando S contiene todos los vértices.
- La salida del algoritmo son dos mapeos D : V → Float y P: V → V tal que:
  - D(v) es la distancia a v desde la fuente
  - P(v) es el vértice anterior a v en el camino desde la fuente a v.

```
Algoritmo Dijkstra
Entrada: G : digrafo simple conexo con todos lo pesos positivos y a : Vertice
{ G tiene vértices a=v0, v1, ..., vn y pesos w(vi,vj) donde w(vi,vj)=∞ si (vi,vj) no es un arco en G }
Salida: D : mapeo de vértice en float y P : mapeo de vértice en vértice
for i := 1 to n do begin    { para cada vértice i }
        D(i) := ∞           { Asumo que la distancia de la fuente a a i es infinita }
        P(i) := 0           { Asumo que no hay anterior a i en el camino de a a i }
end
D(a) := 0                   { La distancia de a a a es 0 }
S := ∅                      { No hay vértices procesados }
for i := 1 to n do begin    { repetir n veces }
        u := un vértice no en S con D(u) mínimo   { elegir el vértice u más cercano a a }
        S := S U { u }                            { marcar a u como procesado }
        for cada vértice v adyacente a u y que no está en S do
            if D(u) + w(u,v) < D(v) then begin    { hacer edge relaxation }
                D(v) := D(u) + w(u,v)             { anotar nueva distancia de a a v }
                P(v) := u                         { anotar que el anterior de v es u en el camino }
            end
end
return (P,D)                { retornar resultados }
```

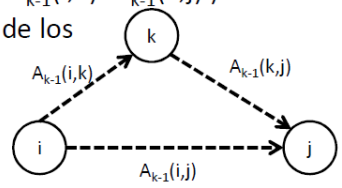
$T_{Dijkstra}(n) = O(n^2)$ si G tiene n vértices y está representado con matriz

$T_{Dijkstra}(n,m) = O((m + n)\log(n))$ (son m actualizaciones a una cola con prioridades adaptable de n elementos), si G está representado con lista de adyacencias y get y put del mapeo tienen O(1).

Floyd

Algoritmo de Floyd: Caminos mínimos

- Dado un digrafo pesado G=(V,A) donde cada arco tiene un peso numérico no negativo.
- Queremos calcular los caminos de costo mínimo de todos los vértices a todos los vértices.
- Supongamos que C(i,j) es peso del arco (i,j) de A.
- Usaremos una matriz A de nxn tal que inicialmente A(i,j)=C(i,j), o ∞ si no hay arco entre i y j
- En la iteración k-ésima veremos si actualizamos a A de acuerdo a $A_k(i,j) = \min(A_{k-1}(i,j), A_{k-1}(i,k)+A_{k-1}(k,j))$.
- P(i,j) almacenará a k (i.e., uno de los vértices intermedios en el camino de i a j.

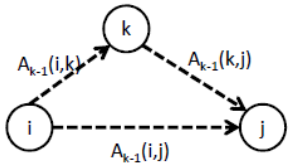


Algoritmo Floyd

```
Para i←1..n hacer
        para j← 1..n hacer
                si hay arco (i,j) entonces A(i,j) ← C(i,j)
                sino A(i,j) ← ∞
                P(i,j) ← 0  { por defecto el camino es directo }

Para i ← 1 .. n hacer
        A(i,i) ← 0

Para k ← 1..n hacer
        para i ← 1..n hacer
                para j ← 1..n hacer
                        si a(i,k) + a(k,j) < a(i,j) entonces
                                a(i,j) ← a(i,k) + a(k,j)
                                p(i,j) ← k
```



$T_{Floyd}(n) = O(n^3)$

DFS con marca y desmarca

DFS con marca y desmarca

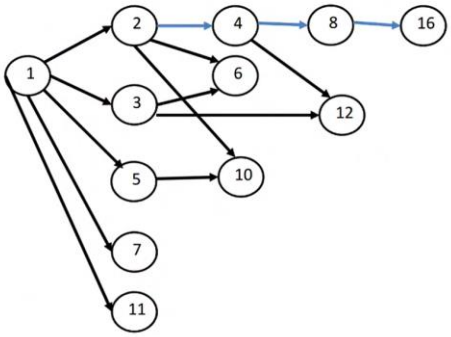
Dado un digrafo pesado con números reales, permite hallar un camino de costo mínimo entre dos vértices origen y destino computando el camino y su costo (entendido como la suma de los pesos de los arcos).

El tiempo de ejecución para un grafo que tiene todos los arcos entre cada par de nodos es O(n!) (en la práctica esto es mucho menos, porque n! se da en el peor caso que es cuando todos los vértices están conectados con todos los otros vértices)

Hacia el algoritmo de Warshall: Cómputo de la clausura transitiva R* de una relación binaria R

- Dado un grafo que representa una relación R, el algoritmo de Warshall permite computar la clausura transitiva de R, notada como R*.

Ejemplo de camino en grafo dirigido: Como hay un camino de 2 a 16, entonces inferimos que 2 divide a 16.



- Cuando R es representada con un grafo G dirigido, si R no es transitiva, entonces G no contiene todos los arcos para los vértices que pueden ser unidos mediante caminos.
- Ej: R = { (1,2), (2,3) } entonces R* = R U {(1,3)} pues es posible ir de 1 a 3 (pasando por 2).

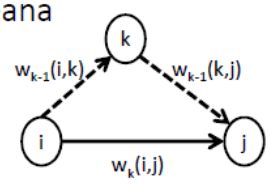
Cómo calcular R*

- Si la relación R entre n elementos se representa con una matriz booleana de nxn (booleana quiere decir formada por 1s y 0s), entonces R^n (es decir, R x R x R x...xR realizado n veces, con "x" representando el producto booleano de matrices).
- La clausura R* se calcula como R U R^2 U R^3 U .... U R^n, donde U representa el join-booleano (or-booleano componente a componente) entre matrices.

Tiempo de ejecución

Estrategia: Para cada vértice k, para cada par de vértices (i,j) ver si puedo conectar i con j a través de k y si es así, agregar (i,j) a la clausura transtiva.

Procedure Warshall(M_R : Matriz booleana de n x n)




```
W := M_R
for k := 1 to n do
        for i := 1 to n do
                for j := 1 to n do
                        w(i,j) := w(i,j) or (w(i,k) and w(k,j))
End { W = [w_ij] es M_R* }
```

$T_{warshall}(n) = O(n^3)$

```
public static void warshall( int [][]a, int n, int [][]w) {
   copiar(a, w, n, n);    // O(n^2)
   for(int k=0; k<n; k++ )   // n iteraciones
     for(int i=0; i<n; i++) // n iteraciones
        for(int j=0; j<n; j++)   // n iteraciones
           w[i][j] = Math.max(w[i][j],
                Math.min(w[i][k], w[k][j])); // O(1)
    // El procedimiento tiene O(n^3)
}
```

Aplicaciones y estrategias

Aplicaciones del DFS para grafos no dirigidos en O(n+m)

- Testear si G es conexo (todos los vértices quedan visitados si y sólo si el grafo es conexo)
- Calcular un árbol abarcador si G es conexo (formado por los vértices de G y por sus arcos tree)
- Calcular las componentes conexas (por cada iteración de DFSShell incremento un contador indicando el número de componente conexa y con ese contador etiqueto los vértices de cada componente)
- Encontrar un camino entre dos nodos (clase siguiente)
- Encontrar un ciclo (clase siguiente)

Algoritmos para encontrar caminos en digrafos pesados con números reales

- DFS pinchado (st-path search): Permite encontrar un camino entre dos vértices s y t
- BFS para hallar camino con cantidad mínima de arcos.
- DFS con backtracking, marca y desmarca: Permite encontrar un camino de costo mínimo entre dos vértices s y t
- Dijkstra: Permite hallar todos caminos de costo mínimo entre un vértice *a* y todos los otros vértices
- Floyd: Permite hallar el camino de costo mínimo entre cada par de vértices s y t.

Aplicaciones del BFS

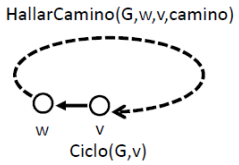
Idem DFS y además:

- Hallar el camino más corto (en cantidad de arcos) entre dos vértices (en O(n+m)) (en clase siguiente).

Estrategia para hallar ciclos

{ Encuentra un ciclo que contenga a v, buscando caminos desde los adyacentes de v (que se llaman w) hacia v. }

```
Algoritmo HallarCiclo( G, v ) : Lista
  encontre ← falso
  mientras hay adyacentes para considerar y no encontre hacer
        w ← siguiente adyacente de v en G
        encontre ← HallarCamino( G, w, v, camino)
  finmientras
  ciclo ← new Lista()
  si encontre  entonces
     ciclo.addLast( v )
     para cada vertice x de camino hacer
        ciclo.addLast( x )
     finpara
  finsi
  retornar ciclo
```



$T_{hallarciclo}(n,m) = O(grado(v)*(n+m) + n)$

18 – ÁRBOLES DE BÚSQUEDA BALANCEADOS

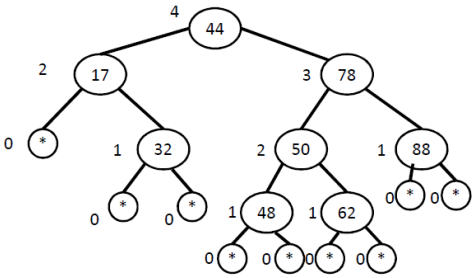
- Hay estructuras alternativas que garantizan tiempo de acceso de orden logarítmico en la cantidad de elementos y se los conoce como árboles de búsqueda balanceados: Árbol AVL, Árbol 2-3 y Árbol B.

> Arboles AVL

- Agregaremos una corrección al árbol binario de búsqueda para mantener una altura del árbol proporcional al logaritmo de la cantidad de nodos del árbol.
- Recordemos que el tiempo de búsqueda, inserción y borrado en un árbol binario de búsqueda es lineal en la altura del árbol.
- Entonces, si n=cantidad de elementos de un arbol T, tendríamos así que $T(n) = O(\log_2(n))$.
- Propiedad del balance de la altura: Para cada nodo interno v de T, las alturas de los hijos difieren en a lo sumo 1.
- Cualquier árbol binario de búsqueda que satisface esta propiedad se dice "árbol AVL" (por Adel'son-Vel'skii y Landis).

Árboles AVL: Ejemplo

- Propiedad del balance de la altura: Para cada nodo interno v de T, las alturas de los hijos difieren en a lo sumo 1.
- Ejemplo: Los * corresponden a nodos nulos (con altura 0 de acuerdo a GT).



Complejidad temporal de la inserción

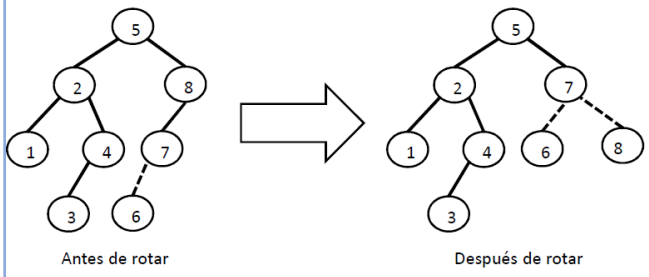
- Noten que las rotaciones se hacen en los nodos del camino desde la raíz hasta la hoja donde se insertó la nueva clave.
- Como las rotaciones se implementan con asignaciones de referencias (posiciones), cada rotación se hace en tiempo constante.
- La cantidad de rotaciones es del orden de la altura del árbol.
- La altura es proporcional al logaritmo de la cantidad de nodos del árbol.
- Por lo tanto, el tiempo de insertar es del orden del logaritmo de la cantidad de nodos del árbol.

Rotaciones

Son cuatro correspondientes a las cuatro combinaciones para la inserción de una clave a partir de un nodo raíz del subárbol considerado:

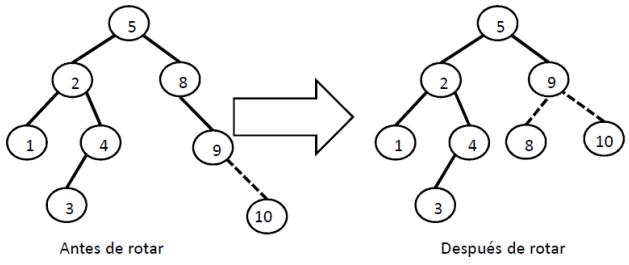
1) izquierda – izquierda: rotación simple de izquierda a derecha
2) izquierda – derecha: Rotación doble de izquierda a derecha
3) derecha – derecha: Rotación simple de derecha a izquierda
4) derecha – izquierda: Rotación doble de derecha a izquierda

Ejemplo de rotación simple de izquierda a derecha



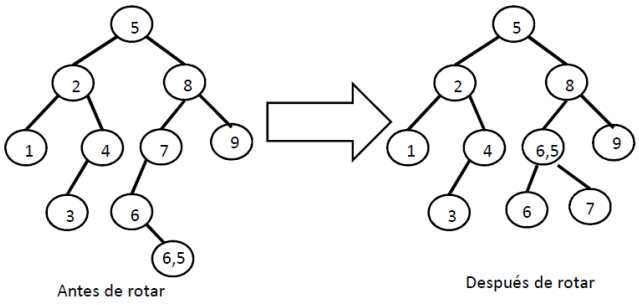
La inserción del 6 destruye la propiedad de AVL en el nodo 8, lo que se resuelve con una rotación simple de izquierda a derecha (tomado de Mark Allen Weiss, Data Structures

Ejemplo de rotación de derecha a izquierda



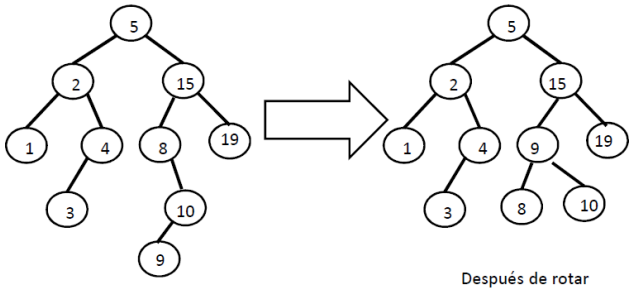
La inserción del 10 destruye la propiedad de AVL en el nodo 8, lo que se resuelve con una rotación simple de derecha y izquierda.

Ejemplo de rotación doble de izquierda a derecha



La inserción del 6,5 destrute la propiedad de AVL en el nodo 7, entonces hay que rotar.

Ejemplo de rotación doble de derecha a izquierda



La inserción del 9 destrute la propiedad de AVL en el nodo 8, entonces hay que rotar.

> Árbol 2-3

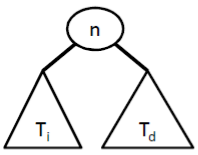
Árboles 2-3: Definiciones

Un "árbol 2-3" es un árbol tal que cada nodo interno (no hoja) tiene dos o tres hijos, y **todas las hojas están al mismo nivel**.

La definición recursiva es: T es un árbol 2-3 de altura h si:

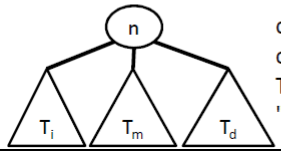
a) T es vacío (es decir de altura 0)

b) T es de la forma:



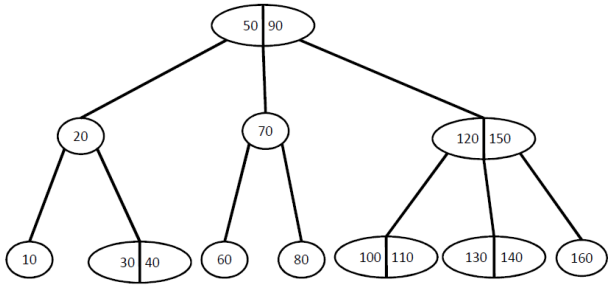
donde n es un nodo y T_i y T_d son árboles 2-3 cada uno de altura h-1.
T_i se dice "subárbol izquierdo" y T_d "subárbol derecho".

c) T es de la forma:



donde n es un nodo y T_i, T_m y T_d son árboles 2-3 cada uno de altura h-1.
T_i se dice "subárbol izquierdo", T_m se dice "subárbol medio" y T_d "subárbol derecho".

Ejemplo de árbol 2-3



Árboles 2-3: Definiciones

- Propiedad: Si un árbol 2-3 no contiene ningún nodo con 3 hijos entonces su forma corresponde a un árbol binario lleno.

Inserción de nodos

RESUMEN:

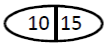
Para insertar un valor X en un árbol 2-3,
primero hay que ubicar la hoja L en la cual X terminará.
Si L contiene ahora dos claves, terminamos.
Si L contiene tres claves, hay que partirla en dos hojas L1 y L2. L1 se queda con la clave más pequeña, L2 con la más grande, y la del medio se manda al padre P de L.
Los nodos L1 y L2 se convierten en los hijos de P.

Si P tiene sólo 3 hijos (y 2 claves), terminamos.
En cambio, si P tiene 4 hijos (y 3 claves), hay que partir a P igual que como hicimos con una hoja sólo que hay que ocuparse de sus 4 hijos. Partimos a P en P1 y P2, a P1 le damos la clave más pequeña y los dos hijos de la izquierda y a P2 le damos la clave más grande y los dos hijos de la derecha.

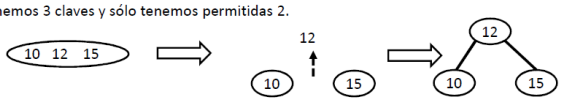
Luego de esto, la clave que sobra se manda al padre de P en forma recursiva. El proceso termina cuando la clave sobrante termina en un nodo con dos claves o el árbol crece 1 en altura (al crear una nueva raíz).

1) Insertamos 10 en el árbol vacío:

2) Insertamos 15: como hay lugar en la única hoja, allí ponemos la nueva clave y terminamos

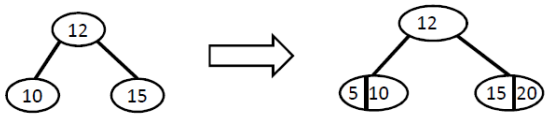


3) Insertamos 12: vamos a la única hoja y ponemos el 12 allí, pero hay rebalse porque tenemos 3 claves y sólo tenemos permitidas 2.

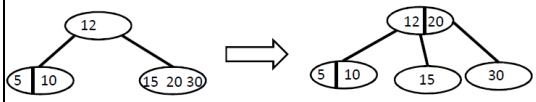


Partimos el nodo en 2 nodos dividiendo las claves y le pasamos al padre los 2 nodos junto con la clave del medio. Como no hay padre, el árbol crece en un nivel al crear un nuevo nodo para acomodar la clave con los 2 nodos como sus hijos.

4) Cualquier clave que inserte, siempre va a una hoja siguiendo el criterio de búsqueda. Inserto 20 y 5. Como no hay rebalse porque había lugar en las hojas, terminamos.



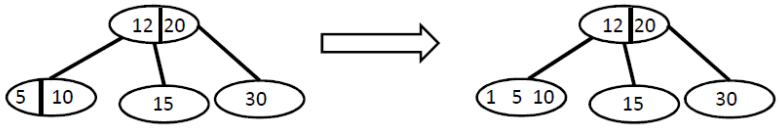
5) Inserto 30, el cual termina en el hijo derecho de 12.



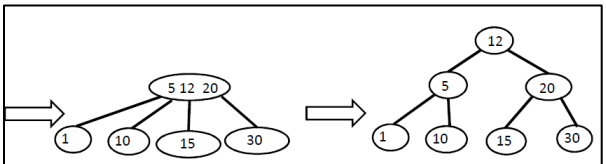
Hay rebalse porque tengo 3 claves y solo tengo permitidas 2

Parto el nodo rebalsado en 2 nodos y se los paso a su padre junto con la clave del medio (que es el 20) y terminamos porque la raíz tiene lugar para otra clave y otro hijo extra.

6) Inserto 1 en el árbol del paso (5):



Hay rebalse



Parto el nodo y subo el 5 al padre (que es la raíz). Pero ahora la raíz tiene rebalse (3 claves y 4 hijos, situación no permitida).

Parto el nodo raíz en 2 nodos y subo el 12 creando una nueva raíz

> Árbol B

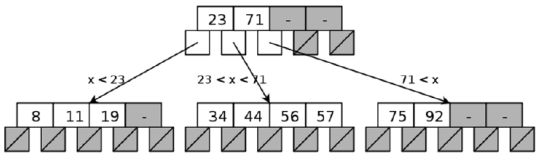
Árbol B

- Un árbol-B (B-tree) es un árbol balanceado (o auto-balanceante) que mantiene los datos ordenados y permite realizar búsquedas, inserciones y eliminaciones en tiempo logarítmico.
- Se puede ver también como una generalización del árbol 2-3 porque sus nodos pueden tener más de tres hijos.
- A diferencia de los árboles 2-3, el árbol-B está optimizado para sistemas que leen y escriben grandes bloques de datos.
- Los árboles-B son un buen ejemplo de una estructura de datos para memoria externa (en disco) y se usan comúnmente en bases de datos y sistemas de archivos.

Arboles B (o M-arios de búsqueda)

- Los árboles B se optimizan para manejar grandes volúmenes de datos.
- Los árboles B se almacenan en disco y el tamaño del nodo coincide con un múltiplo entero del tamaño del sector del disco.
- Se utilizan para implementar índices en bases de datos.
- El grado de ramificación d del árbol es un entero, indicando que un nodo tiene entre d y 2d claves y entre d+1 y 2d+1 hijos (excepto la raíz que puede tener menos de d claves y tiene por lo menos 1 clave y 2 hijos).
- Cuando d=1 tenemos un árbol 2-3.
- Dependiendo de la formalización, al número 2d+1 se lo llama M (i.e. tenemos M-1 claves y M hijos a lo sumo por nodo; en un árbol 2-3, M vale 3)

Árbol B



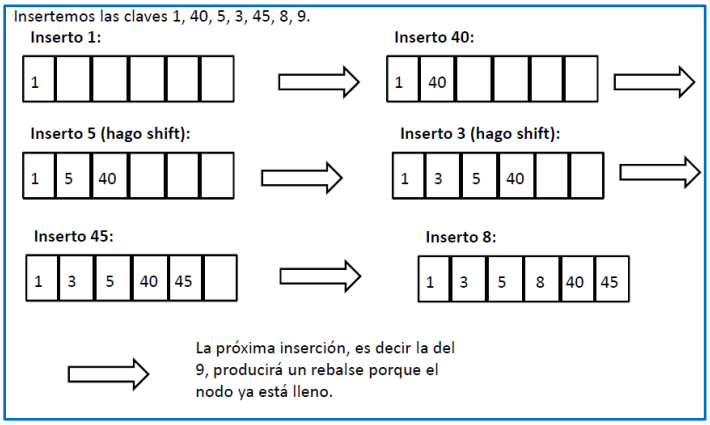
El tiempo de búsqueda, inserción y eliminación es en tiempo logarítmico puesto que es del orden de la altura del árbol.
Si d=1000, cada nodo tiene d claves y d+1 hijos por lo menos, entonces la altura del árbol es del orden $\log_d(n)$ con n = cantidad de claves del árbol.
Ej: si d=1.000 y n=1.000.000, entonces $\log_d(1.000.000)=2$ entonces son 3 lecturas (como la altura es 2, el camino tiene 3 nodos).
Generalmente hay espacio desperdiciado en los nodos (i.e. fragmentación interna).

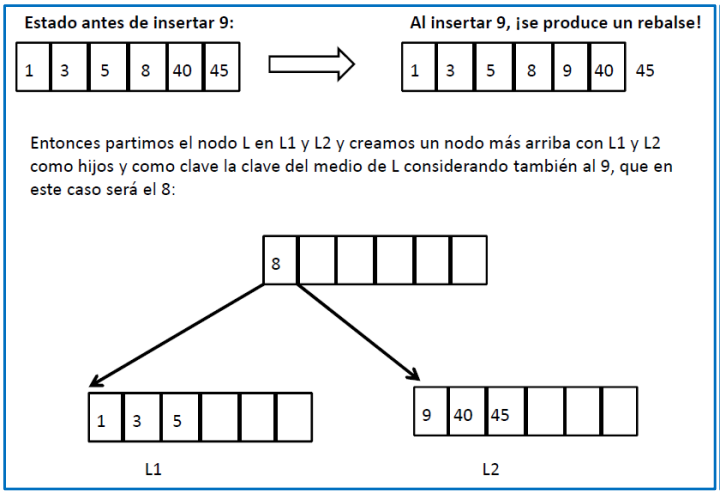
Ejemplo

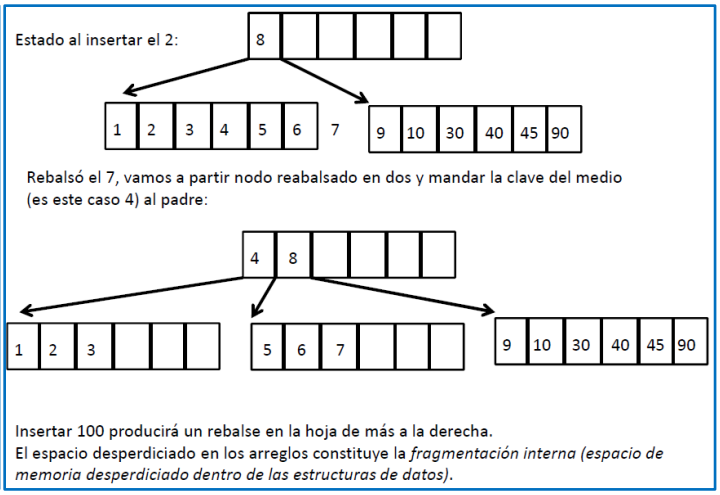
Inserción de claves

Árboles B: Inserción

- Comenzamos con un nodo vacío (el cual funciona como un arreglo ordenado).
- Las claves se insertan (en forma lineal y ordenada) en el nodo hasta tener a lo sumo 2d = m-1 claves.
- Luego, al insertar la siguiente clave, el nodo rebalsa.
- El nodo se parte en dos (cada uno con d claves): la clave del medio va al nodo de arriba (incrementando la altura del árbol cuando tal nodo de arriba no existiera) y los nodos que quedan se quedan cada uno con la mitad de las claves menores a d y mayores a d respectivamente.

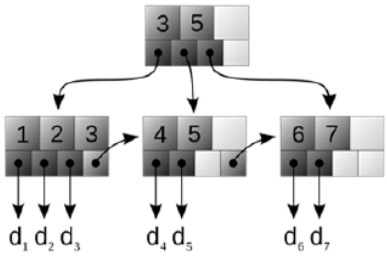






Árbol B+

Variante del árbol B: Árbol B+



El árbol B+ almacena entradas (clave,valor) sólo en las hojas, los nodos internos almacenan solo claves que sirven para guiar la búsqueda en O(h).
Recorrer las hojas de izquierda a derecha iterativamente permite listar las claves en forma ascendente en O(n).
Ejemplo: 3 en la raíz indica la mayor clave del primer hijo del siguiente nivel.
5 en la raíz indica la mayor clave del segundo hijo del siguiente nivel.
d_1, d_2, d_3, d_5, d_6, d_7 corresponden a los valores de las entradas para las claves 1,2,3,4,5,6,7 (punteros al archivo de datos).

19 – PROCESAMIENTO DE TEXTO

> Tries

Definición

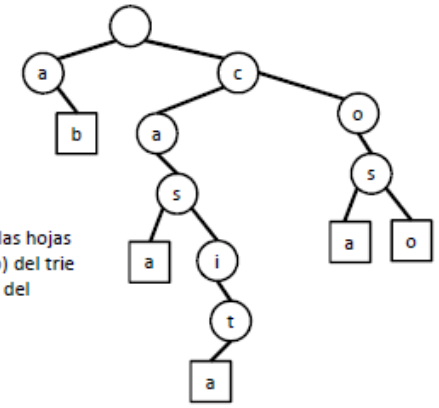
- Sea S un conjunto de s strings sobre un alfabeto Σ.
- Un trie para S es un árbol ordenado T tal que:
  - Cada nodo de T, excepto la raíz, está etiquetado con un carácter de Σ.
  - El orden los hijos de un nodo interno de T está determinado por el orden canónico de Σ.
  - T tiene s nodos externos, cada uno asociado con un string de S, tal que la concatenación de los rótulos de los nodos del camino de la raíz a una hoja v produce el string de S asociado a v.

Tries

- Un trie es una estructura de datos que se usa para implementar conjuntos de strings, y mapeos y diccionarios de string en un tipo E.
- Un trie es un árbol que factoriza prefijos comunes entre las cadenas almacenadas en el mismo.
- Los caminos de la raíz a las hojas representan las palabras del conjunto o las claves del mapeo.

Ejemplo (notación de GT)

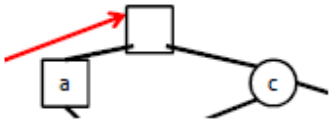
Sea S = { "ab", "casa", "casita", "cosa", "coso" } un conjunto de cadenas de texto.



Los caminos de la raíz a las hojas (marcadas con cuadrado) del trie representan las cadenas del conjunto S.

- Nota: Para permitir que una cadena sea un prefijo de otra cadena del trie se puede usar una marca en el nodo interno.
  Ej: "a" y "ab" están en S.
  "casa" y "casas" están en el trie. "casa" es un prefijo de "casas".

- Nota: La raíz se asocia con el string vacío. Al marcar el nodo raíz como terminador, tenemos al string vacío "" como un elemento del conjunto de cadenas determinado por el trie.



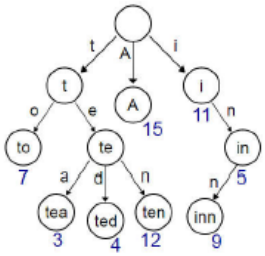
Propiedades

Un trie T almacenando:

- una colección S de s strings de longitud total n (es decir, la suma de las longitudes de todos los strings del conjunto es n)
- sobre un alfabeto de tamaño d cumple:
- Cada nodo interno de T tiene a lo sumo d hijos
- T tiene s nodos externos
- La altura de T es igual a la longitud del string más largo de S
- El número de nodos de T es O(n).

Complejidad temporal

- Sea un conjunto S implementado con un trie T sobre un alfabeto Σ.
- Sea s=cardinal de S, d=cardinal de Σ, m=largo de un string a procesar
- T_{put} (s,d,m) = O(m)
- T_{get}(s,d,m) = O(m)
- T_{remove}(s,d,m) = O(dm)



> Comprensión de datos y codificación de Huffman

Definición de compresión de datos

Tipos de compresión

- **Compresión sin pérdida de información:**
  - los datos antes y después de comprimirlos son exactos en la compresión sin pérdida.
  - En el caso de la compresión sin pérdida una mayor compresión solo implica más tiempo de proceso.
  - Se utiliza principalmente en la compresión de texto.
- **Compresión con pérdida de información:**
  - Puede eliminar datos para reducir aún más el tamaño, con lo que se suele reducir la calidad.
  - Hay que tener en cuenta que una vez realizada la compresión, no se puede obtener la señal original, aunque sí una aproximación cuya semejanza con la original dependerá del tipo de compresión.
  - Se utiliza principalmente en la compresión de imágenes, videos y sonidos. Ej: Formatos jpg, mpeg, mp3

Compresión de datos

- La **compresión de datos** consiste en la reducción del volumen de información tratable (procesar, transmitir o grabar).
- En principio, con la compresión se pretende transportar la misma información, pero empleando menor cantidad de espacio.

Codificación binaria

Codificación binaria

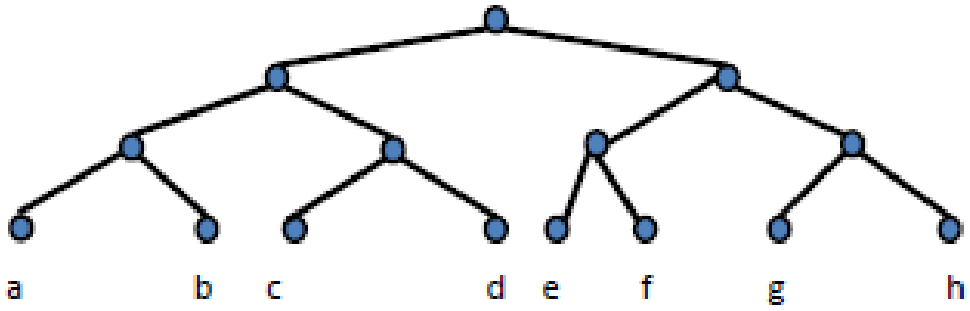
- Si el alfabeto Σ tiene n símbolos se necesitan $\lceil \log_2(n) \rceil$ bits para codificar cada símbolo
- Ejemplo: Si $\Sigma = \{ a, b, c, d, e, f, g, h \}$, entonces n = 8

  Como $\log_2(8) = 3$, entonces una codificación posible es:

  a = 000 b = 001 c = 010 d = 011

  e = 100 f = 101 g = 110 h = 111

  Árbol de codificación: hijo izquierdo significa 0 e hijo derecho, 1:



La cantidad de bits es la altura del árbol de codificación.

La cadena "aaabbccdefad" se codifica sin compresión como (los puntos son para leerla más fácil y no se almacenan):
000.000.000.001.001.010.010.011.100.101.000.011

Codificación de Huffman

Codificación de Huffman: Idea

- Construye un árbol binario T que representa la codificación para una cadena X
- Cada nodo, excepto la raíz, representa un bit del código.
- El hijo izquierdo representa un 0 y el derecho un 1
- Cada hoja representa un carácter c
- La codificación de un carácter c se define como la secuencia de bits determinada por el camino de la raíz a la hoja que contiene a c en el árbol T.
- Cada hoja tiene una frecuencia f(v) correspondiente a la frecuencia del carácter c almacenado en v
- Cada nodo interno tiene una f(v) que corresponde a la suma de las frecuencias de los caracteres en dicho subárbol.

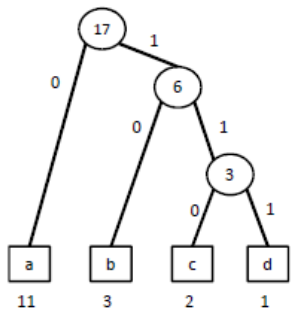
Algoritmo de Huffman

Algoritmo Huffman(X)
Entrada: Un string X de longitud n sobre alfabeto de tamaño d
Salida: árbol para codificar X
Computar la frecuencia f(c) para cada carácter c de X
Crear una cola con prioridad C
Para cada carácter c en X **hacer**
    Crear un árbol binario hoja T con rótulo c
    Insertar T en Q con prioridad f(c)
Mientras Q.size() > 1 **hacer**
    f_1 ← Q.min().key(); T_1 ← Q.removeMin()
    f_2 ← Q.min().key(); T_2 ← Q.removeMin()
    Crear un nuevo árbol binario T con hijo izquierdo T_1 e hijo derecho T_2
    Insertar T en Q con prioridad f_1+f_2
Retornar el árbol Q.removeMin()

Ejemplos

Ejemplo (cont)

El algoritmo genera el siguiente código:



a = 0
b = 10
c = 110
d = 111
Por lo tanto, la cadena X = aaaaaabbbaaaaaccd se comprime como:
00000010101000000110110111, la cual mide 26 bits.
Así tengo una tasa de compresión del:
34 bits _____ 100%
26 bits _____ x =
100%*26bits/34bits = 76%

Ejemplo

Supongamos queremos comprimir la cadena X = aaaaaabbbaaaaaccd (que tiene longitud 17)

El alfabeto tiene 4 símbolos, entonces, sin compresión, necesito 2 bits para representar cada uno:

a=00, b=01, c=10, y d=11 .

La cadena sin comprimir se representa como:

00.00.00.00.00.00.01.01.01.00.00.00.00... etc

Necesito 17*2 = 34 bits para representarla.

El cálculo de las frecuencias de apariciones da: f(a) = 11, f(b)=3, f(c)=2 y f(d)=1.