

PROGRAMACIÓN Y ESTRUCTURA DE DATOS

Temario de Teoría y resumen del Seminario de C++



Universitat d'Alacant
Universidad de Alicante

Resumen

Este documento contendrá propiedades de las distintas estructuras de datos además de las herramientas que se utilizarán para estudiarlas y definir las en el lenguaje C++.

Considerar que por ahora solo está realizado desde el tema 1 al 3.

Eduardo Espuch

Curso 2019-2020

Índice

1. Introducción a los TADs, los tipos lineas	2
1.1. Introducción a los TADs	2
1.2. Vectores	5
1.3. Listas	6
1.4. Pilas	9
1.5. Colas	10
2. La eficiencia de los algoritmos	11
2.1. Noción de complejidad	11
2.1.1. Complejidad temporal, tamaño del problema y paso	11
2.2. Cotas de complejidad	12
2.3. Notación asintótica	13
2.4. Obtención de cotas de complejidad	13
3. El tipo árbol	14
3.1. Definiciones generales	14
3.2. Árboles Binarios	17
3.3. Árboles de búsqueda	22
3.3.1. Árboles binarios de búsqueda	23
3.3.2. Árboles AVL	25
3.3.3. Árboles 2-3	28
3.3.4. Árboles 2-3-4	28
4. Conjuntos	29
4.1. Definiciones generales	29
4.2. Diccionario	29
4.2.1. Tabla de dispersión	29
4.3. Cola de prioridad	29
4.3.1. Montículo	29
4.3.2. Cola de prioridad doble	29
5. Grafos	29
5.1. Concepto de grafo y terminología	29
5.2. Especificación algebraica	29
5.3. Representación de grafos	29
5.4. Grafos dirigidos	29
5.4.1. Recorrido en profundidad o DFS	30
5.4.2. Recorrido en anchura o BFS	30
5.4.3. Grafos acíclicos dirigidos o GAD	30
5.4.4. Componentes fuertemente conexos	30
5.5. Grafos no dirigidos	30
5.5.1. Algoritmos de recorrido	30
6. Seminario de C++	30
6.1. Clases y objetos	30
6.2. Composición y herencia	30
6.3. Constructor y destructor	30
6.4. Funciones y clases amigas, reserva de memoria	30
6.5. Sobrecarga de operadores	30

prueba

1. Introducción a los TADs, los tipos lineas

1.1. Introducción a los TADs

Los TADs, Tipo Abstracto de Datos es un concepto nacido a partir de:

- Tipo de datos: una forma de clasificar los objetos de los programas(ya sean variables, parámetros, contantes) y determinar los valores que pueden tomar, ademas de determinar las operaciones que se aplican (los enteros con operaciones aritméticas y los booleanos con operaciones lógicas, por ejemplo).
- Abstracto: la manipulación de los datos solo dependen del comportamiento descrito en su especificación (que hace) y es independiente de su implementacion (como se hace), pudiendo existir múltiples implementaciones para una especificación.
 - Especificación de un TAD: consiste en establecer las propiedades que lo definen, usando la definición informal (lenguaje natural) o formal (lenguaje algebraico). Para que sea útil debe ser:
 - Precisa: solo produzca lo imprescindible
 - General: sea adaptable a diferentes contextos
 - Legible: sea un comunicador entre especificador e implementador
 - No ambigua: evite problemas de interpretación
 - Implementacion de un TAD: consiste en determinar una representación para los valores del tipo y en codificar sus operaciones a partir de esta representación. Para que sea útil debe ser:
 - Estructurada: facilita su desarrollo
 - Eficiente: optimiza el uso de recursos, la evaluación de distintas soluciones mediante la complejidad (espacial y temporal)
 - Legible: facilita su modificación y mantenimiento

Especificación algebraica

La especificacion algebraica(ecuacional): establece las prioridades de un TAD mediante ecuaciones con variables cuantificadas universalmente, de manera que las propiedades dadas se cumplen para cualquier valor que tomen las variables. Los pasos a seguir son:

- Identificación de los objetos del TAD y sus operaciones (declaración del TAD, módulos que usa, parámetros)
- Definición de la signatura (sintaxis) de un TAD (nombre del TAD y perfil de las operaciones)
- Definición de la semántica (significado de las operaciones)

Consideraremos una operaciones a una función que toma como parámetros (entrada) cero o mas valores de diversos tipos, y produce como resultado un solo valor de otro tipo. El caso de cero parámetros representa una **constante** del tipo de resultado.

Sintaxis y Semántica**Modulo** *nombre_TAD*Usa *lista_datos_usados***Parámetro tipo** *tipo_dato_parametros***Operaciones** de los parametros

...

Sintaxis de las operaciones

definidas para los parametros

...

FParamétro**Tipo** *nombre_TAD***Operaciones** del TAD

...

Se denota la signatura de las operaciones
definidas para el TAD actual, SINTAXIS

...

VAR *lista_variables***Ecuaciones** del TAD

...

Se realiza la implementacion de las
operaciones definidas para el TAD, SEMANTICA

...

FModulo

Ademas, hay que considerar dos propiedades muy importantes, consistencia y completitud. Si se ponen ecuaciones de mas, se pueden igualar términos que están en clases de equivalencia diferentes, mientras que si se ponen de menos, se puede generar un numero indeterminado de términos incongruentes con los representantes de las clases existentes.

Clasificaremos a las operaciones de la siguiente forma:

- Constructoras: devuelven un valor del tipo_TAD, hay:
 - Generadores: permiten generar, por aplicaciones sucesivas, todos los valores del TAD a especificar. Llamaremos canonicas a las operaciones generadores que crean el TAD vacio (denotado con *crear*) o añaden datos al TAD (denotado con *insertar*).
 - Modificadores: otro tipo de operacion que no sea generadora, modificando el contenido del TAD, por ejemplo.
- Consultores: devuelve un valor de un tipo diferente

En general, las operaciones modificadoras y consultoras se especifican en terminos de las generadores. En ocasiones, una operacion modificadora puede especificarse en terminos de otras modificadoras o consultoras. Diremos que se trata de una **operacion derivada**. Basicamente, se utilizan las funciones generadoras como parametros para especificar algun caso en particular a la hora de implementar la operacion.

Las **operaciones auxiliares** se introducen en una especificacion para facilitar su escritura y legibilidad. Son invisibles para los usuarios del TAD (tambien llamadas **ocultas o privadas**).

Por lo general, se puede usar tratamiento de errores, añadiendo una constante a la singatura que modeliza un valor de error, es decir, una operacion generadora (denotado por lo general con *error()*). Ademas, en ciertas operaciones se pueden definir casos en los que se devuelva directamente el error. Puede darse que, en lugar de usarse con operaciones generadores, se haga con consultoras, teniendo que el parametro devuelto denote a un error, teniendo definido para ese un caso de error (en estos casos, se denotan con *error_dato()*).

Tendremos una ecuación condicional, la cual es equivalente a un conjunto finito de ecuaciones no condicionales, podemos usar dos formas para implementarlo:

Asignación al resultado de una condición TOP	Condición previamente a la asignación
$\begin{array}{l} \text{op}(p_1, \dots, p_n) = \\ \text{si } (\text{condición}) \text{ entonces } \textbf{valor1} \\ \text{sino } \textbf{valor2} \\ \text{fsi} \end{array}$	$\begin{array}{l} \text{si } (\text{condición}) \text{ entonces} \\ \text{op}(p_1, \dots, p_n) = \textbf{valor1} \\ \text{sino} \\ \text{op}(p_1, \dots, p_n) = \textbf{valor2} \\ \text{fsi} \end{array}$

Tabla de desigualdades con números	
$==$ igualdad	$<>$ desigualdad
$<$ menor que	$>$ mayor que
\leq igual o menor que	\geq igual o mayor que

Cabe decir, que a la hora de realizar las ecuaciones para una operación, tenemos que considerar los siguientes pasos:

1. Estudio del caso base, se pasa como parámetro una operación canónica o un valor controlado, en estos casos el valor que se devuelve es trivial.

$$\begin{array}{c} \text{op}(\text{crear_item}()) = \text{item} \\ \text{o} \\ \text{op}(\text{valor_particular}) = \text{valor} \end{array}$$

2. Estudio del caso recursivo, recomendable pasar como parámetro una operación canónica con el cual sabemos que al menos contiene un valor, usando recursividad hasta alcanzar un caso base. Si la operación es del tipo constructora (modificadora) entonces al tratar el item puede usarse la propiedad conmutativa ($a(b(x)) \Leftrightarrow b(a(x))$) con la operación canónica del tipo *insertar*

$$\begin{array}{c} \text{op}(\text{insertar_item}(\text{item}, x)) = // \text{trata el item, con } \text{insertar_item}(\text{op}(\text{item}), x) \\ \text{o} \\ \text{op}(\text{valor}) = // \text{trata el valor} \end{array}$$

Pueden darse varios casos bases, siempre irán delante del caso recursivo debido a que se interpretan de forma secuencial.

Implementación

Dada una especificación de un tipo, se pueden construir diversas implementaciones, cada una definiéndose en un módulo diferente, llamado módulo de implementación. La construcción de estos módulos consta de dos fases:

- Elección de una representación para los diferentes tipos definidos en la especificación
- Codificación de las operaciones en términos de la representación elegida

Los mecanismos de abstracción en los lenguajes de programación son:

- **Encapsulamiento** de la representación del TAD
- **Ocultación de la información**, para limitar las operaciones posibles sobre el TAD

- **Generacidad**, para lograr implementaciones genéricas validas para distintos tipos
- **Herencia**, para reutilizar implementaciones

Los lenguajes tradiciones son ineficientes (Fortran, C, ...) pero los mas modernos son capaces (C++, Java, ...).

1.2. Vectores

Un vector es un conjunto ordenador de pares $\langle \text{indice}, \text{valor} \rangle$, donde para cada indice definido dentro de un rango finito existe asociado un valor. En terminos matematicos, es una correspondencia entre los elementos de un conjunto de indices y los de un conjunto de valores.

Sintaxis

Modulo Vector

Usa BOOL, NATURAL

Parámetro tipo Item

Operaciones del parametro

$c, f \rightarrow \text{int}$, limites inf y sup

en todas las ecuaciones, $c \leq i, j \leq f$

$\text{==}(\text{item}, \text{item}) \rightarrow \text{bool}$

$\text{error_item}() \rightarrow \text{item}$

(?)

FParamétro

Tipo vector

Operaciones

$\text{Crear}() \rightarrow \text{vector}$

$\text{recu}(\text{vector}, \text{int}) \rightarrow \text{item}$

$\text{eliminar}(\text{vector}, \text{int}) \rightarrow \text{vector}$

$\text{palindromo}(\text{vector}) \rightarrow \text{bool}$

$\text{asig}(\text{vector}, \text{int}, \text{item}) \rightarrow \text{vector}$

$\text{esVacíoPos}(\text{vector}, \text{int}) \rightarrow \text{bool}$

$\text{buscar}(\text{vector}, \text{item}) \rightarrow \text{bool}$

Semántica

VAR v:vector; i,j:int; x,y:item;

Ecuaciones

$\text{Crear}() \rightarrow \text{vector}$

f. canónica se denotara con CR

$\text{recu}(\text{vector}, \text{int}) \rightarrow \text{item}$

recupera un item, se denotara con RC

$\text{RC}(\text{CR}(), i) = \text{error_item}()$

$\text{RC}(\text{AS}(v, i, x), j) =$

si $(i == j)$ entonces x

sino $\text{RC}(v, j)$

fsi

$\text{asig}(\text{vector}, \text{int}, \text{item}) \rightarrow \text{vector}$

f. canónica, se denotara con AS

$\text{AS}(\text{AS}(v, i, x), j, y) =$

si $(i < j)$ entonces $\text{AS}(\text{AS}(v, j, y), i, x)$

sino $\text{AS}(v, j, y)$

fsi

$\text{esVacíoPos}(\text{vector}, \text{int}) \rightarrow \text{bool}$

comprueba una posicion, se denotara con EVP

$\text{EVP}(\text{CR}(), i) = \text{TRUE}$

$\text{EVP}(\text{AS}(v, i, x), j) =$

si $(i == j)$ entonces FALSE

sino $\text{EVP}(v, j)$

fsi

<pre> eliminar(vector,int)→vector elimina un elemento del vector, se denotara con EL EL(CR(),i)=CR() EL(AS(v,i,x),j)= si(i==j) entonces v sino AS(EL(v,j),i,x) fsi </pre>	<pre> buscar(vector,item)→bool buscara un item del vector, se denotara con BSC BSC(CR(),x)=FALSE BSC(AS(v,i,x),y)= si(x==y) entonces TRUE sino BSC(v,y) fsi </pre>
---	--

palindromo(vector)→bool

Comprueba si un vector es un palindromo, se denotara con PLD

PLD(CR())=TRUE //caso base, un vector vacio es palindromo

PLD(v)=

si (RC(v,c)==RC(v,f)) entonces PLD(EL(EL(v,c),f))

sino FALSE

fsi

FModulo

Hay mas funciones, pero estas son un ejemplo. Cabe destacar la de palindromo. La idea detras de este es que, utilizando la funcion *recuperar* y la cota inferior y superior (sabiendo que son las posiciones de los extremos del vector) se realizara una igualdad entre los extremo.

Si son iguales, se hace recursion sobre el vector resultante de eliminar los dos extremos, localizados en las posiciones c y f. Esto continuara hasta que el vector quede vacio(puede darse que haya un numero impar de elementos, aun asi, comprobara que un valor es igual a si mismo y lo eliminara dos veces) o hasta que haya un caso que no se cumpla.

Si no lo son, la recursion finalizara y devolvera FALSE.

1.3. Listas

Una lista es una secuencia de cero o mas elementos de un mismo tipo tal que $\{e_1, e_2, \dots, e_n\}$, $\forall \geq 0$ o, de forma mas general, $\{e_p, e_{sig_1(p)}, \dots, e_{sig_{n-1}(\dots sig_1(p) \dots)}\}$, siendo n la longitud de la lista y, si se da que $n = 0$, tendremos una lista vacia.

e_1 o e_p es el primer elemento y e_n o $e_{sig_{n-1}(\dots sig_1(p) \dots)}$ es el ultimo elemento de la lista.

Distinguimos las siguientes propiedades:

- Se establece un orden secuencial estricto sobre sus elementos por la posicion que ocupan dentro de la misma. De esta forma e_i precede a $e_{sig(i)}$ para $1 \leq i \leq n - 1$ y, de igual forma $e_{sig(i)}$ sucede a e_i , teniendo que e_i ocupa la posicion i .
- La lista nos permite conocer cualquier elemento de la misma accediendo a su posicion, algo que no podremos hacer con las pilas y las colas. Utilizaremos el concepto generalizado de posicion, con una ordenacion definida sobre la misma, por lo tanto no tiene por que corresponderse exactamente con los numeros enteros, como clasicamente se ha interpretado este concepto.

Una lista ordenada es un tipo especial de lista en el que se establece una relacion de orden definida entre los items de la lista (si se inserta un elemento a una lista, se debera de asegurar que se cumpla la relacion entre los elementos)

Representacion de listas

El TAD lista se utiliza para almacenar listas de un numero variable de objetos, con dos tipos de representacion:

- Representacion secuencial (internamente un array): a partir de tipos base (arrays) o tipos definidos por el usuario (un vector usando herencia o layering)
- Representacion enlazada (internamente punteros a nodo): se construyen a partir de tipos base (punteros a nodo) donde cada objetos se almacena en un nodo el cual se enlaza con el siguiente. De esta forma, la lista es un puntero al primer nodo o a NULL si ésta es vacia.

Los nodos son un contenedor para almacenar informacion el cual contiene dos aprtes: la informacion del objeto que guarda y el puntero o punteros a otros nodos con los que construir la estructura (puede tratarse de una lista doble enlazada si tiene punteros para el previo y el siguiente).

Veamos algunas funciones basicas sobre las listas:

- Averiguar si la lista esta vacia
- Busqueda de un elemento
- Insercion y borrado de un elemento: hay que distinguir si es al principio, en una posicion intermedia o final de la lista. En cada caso habra que considerar modificar el puntero del previo y del siguiente al elemento en cuestion. Tambien el nodo que contiene la lista.

Listas ordenadas

Una lista ordenada es una lista que en todo momento se mantiene ordenada gracias al hecho se insertar siempre los nodos en la posicion que se les corresponde segun el orden definido (el borran no desordena). En comparacion con una lista simple, el tiempo medio de busqueda se reduce (ya que sabemos que si existe debe de estar alrededor de una posicion y por lo tanto no recorreremos todo el vector) ademas de que no hace falta ordenar la lista.

Vamos a ver la especificacion algebraica para la lista, hay MUUUUUUCHAS ecuaciones, con lo cual no espereis que esten todas, que me da pereza.

Sintaxis

Modulo Lista

Usa BOOL, NATURAL

Parámetro tipo Item, poscion

Operaciones del parametro

$==(posicion,posicion) \rightarrow bool$

$error_item() \rightarrow item$

$error_posicion() \rightarrow posicion$

FParamétro

Tipo lista

Operaciones

$Crear() \rightarrow lista$

$esVacia(lista) \rightarrow bool$

$concatenar(lista,lista) \rightarrow lista$

$anterior,siguiente(lista,posicion) \rightarrow posicion$

$borrar(lista,posicion) \rightarrow lista$

$sublista(lista,posicion,natural) \rightarrow lista$

$borrarultimo(lista) \rightarrow lista$

$insCabeza(lista,item) \rightarrow lista$

$longitud(lista) \rightarrow natural$

$primera,ultima(lista) \rightarrow posicion$

$insertar(lista,posicion,item) \rightarrow lista$

$obtener(lista,posicion) \rightarrow item$

$inversa(lista) \rightarrow lista$

$quitaPares(lista) \rightarrow lista$

Semantica

VAR l_1, l_2 :lista; p:posicion; n:natural; x,y:item;

Ecuaciones

<p>Crear() → lista f. canónica se denotara con CR esVacia(lista) → bool comprueba si es vacia, se denotara con EV EV(CR()) = TRUE EV(IC(l_1, x)) = FALSE concatenar(lista, lista) → lista concatena dos listas, se denotara con CNC CNC(CR(), l_1) = l_1 CNC(l_1, CR()) = l_1 CNC(IC(l_1, x), l_2) = IC(CNC(l_1, l_2), x) primera, ultima(lista) → posicion posicion del primer/ultimo elemento, se denotaran con FRS y LST FRS(CR()) = error_posicion() LST(CR()) = error_posicion() LST(IC(l_1, x)) = si (EV(l_1)) entonces FRS(IC(l_1, x)) sino LST(l_1) fsi anterior, siguiente(lista, posicion) → posicion posicion del anterior/siguiente elemento, se denotaran con PREV y NEXT PREV(l_1, FRS(l_1)) = error_posicion() NEXT(l_1, LST(l_1)) = error_posicion() si (p = LST(l_1)) entonces PREV(l_1, NEXT(l_1, p)) = p PREV(IC(l_1, x), FRS(l_1)) = FRS(IC(l_1, x)) si (p ≠ FRS(l_1)) entonces NEXT(l_1, PREV(l_1, p)) = p NEXT(IC(l_1, x), FRS(l_1)) = FRS(l_1) borrarultimo(lista) → lista borra el ultimo elemento, se denotara con BRLS BRLS(CR()) = CR() BRLS(IC(l_1, x)) = si (EV(l_1)) entonces CR() sino BRLS(l_1) fsi</p>	<p>insCabeza(lista, item) → lista f. canónica, se denotara con IC longitud(lista) → natural obtiene la longitud, se denotara con LNG LNG(CR()) = 0 LNG(IC(l_1, x)) = 1 + LNG(l_1) obtener(lista, posicion) → item devuelve el item indicado, se denotara con SRC SRC(CR(), p) = error_item() SRC(IC(l_1, x), p) = si (p = FRS(IC(l_1, x))) entonces x sino SRC(l_1, p) fsi borrar(lista, posicion) → lista borra el item indicado, se denotara con BRR BRR(CR(), p) = CR() BRR(IC(l_1, x), p) = si (p = FRS(IC(l_1, x))) entonces l_1 sino IC(BRR(l_1, p), x) fsi sublista(lista, posicion, natural) → lista genera una sublista se denotara con SBL SBL(l_1, p, 0) = CR() SBL(CR(), p, n) = CR() SBL(IC(l_1, x), p, n) = si (p = FRS(IC(l_1, x))) entonces IC(SBL(l_1, FRS(l_1), n-1), x) sino SBL(l_1, p, n) fsi insertar(lista, p, item) → lista inserta un item en la lista se denotara con INS INS(CR(), p, x) = CR() INS(IC(l_1, x), p, y) = si (p = FRS(IC(l_1, x))) entonces IC(IC(l_1, y), x) sino IC(INS(l_1, y), x)</p>
--	---

quitaPares(lista)→lista (?)	fsi
elimina las posiciones pares,	inversa(lista)→lista
se denotara con QP	invierte la lista
QP(CR())=CR()	se denotara con INV
QP(l ₁)=	INV(CR())=CR()
si(LST(l ₁)MOD2==0) entonces	INV(IC(CR(),x))=IC(CR(),x)
QP(BRLS(l ₁))	INV(IC(l ₁ ,x))=
sino	INS(INV(l ₁),LST(INV(l ₁)),x)
INS(QP(BRLS(l ₁)), LST(l ₁),SRC(l ₁ ,LS(l ₁)))	INV(IC(CR(),x))=IC(CR(),x)
fsi	
FModulo	

1.4. Pilas

Una pila es una lista en la que todas las inserciones y borrados se realizan en un unico extremo, llamado tope o cima. Sabemos por tanto que el ultimo elemento insertado en la pila sera el primero en ser borrado de la misma (LIFO). Tambien podemos conocer cual es el elemento que se encuentra en la cima. Diferenciamos las siguientes representaciones:

- Representacion secuencial (internamente un array): se construye a partir de tipos base (array) o definidos por el usuario (un vector usando herencia o layering).

Los algoritmos que se usan deben de ir realizando inserciones por la primera componente, muy ineficiente, usando un cursor que indique la posicion actual del primer elemento de la pila. Es muy sencillo de implementar pero el tamaño maximo para esta puede ser una desventaja.

- Representacion enlazada (internamente punteros a nodo): se construye a partir de tipos base (punteros a nodo) o definidos por el usuario(una lista usando herencia o layering), de esta forma solventamos el problema del tamaño maximo ya que no hara falta definirlo.

Sintaxis

Modulo pila

Usa BOOL

Parámetro tipo Item

Operaciones del parametro

error_item()→item

FParamétro

Tipo pila

Operaciones

Crear()→pila

desapilar(pila)→pila

esvacia(pila)→bool

apilar(pila,item)→pila

cima(pila)→item

base(pila)→item

Semántica

VAR p:pila; x:item;

Ecuaciones

Crear() \rightarrow pila

f. canónica se denotara con CR

desapilar(pila) \rightarrow pila

quita el elemento de la cima,

se denotara con DAP

DAP(CR())=CR()

DAP(AP(p,x))=p

esvacia(pila) \rightarrow bool

comprueba si es vacia,

se denotara con EV

EV(CR())=TRUE

EV(AP(p,x))=FALSE

apilar(pila,item) \rightarrow pila

f. canónica, se denotara con AP

cima(pila) \rightarrow item

obtiene el elemento en la cima,

se denotara con TOP

TOP(CR())=error_item()

TOP(AP(p,x))=x

base(pila) \rightarrow item

devuelve la base,

se denotara con BAS

BAS(CR())=

BAS(AP(p,x))=

si (EV(p)) entonces x

sino BAS(p)

fsi

FModulo

1.5. Colas

Una cola es otro tipo de lista en la cual los elementos se insertan por un extremo (fondo) y se suprimen por el otro (tope), actuando como una FIFO. Las operaciones definidas sobre una cola son similares a las definidas para las pilas con la salvedad del modo en el cual se extraen los elementos. Distinguimos las siguientes representaciones:

- Representación secuencial (internamente un array): se construye a partir de tipos base (array) o definidos por el usuario (un vector usando herencia o layering).

Los algoritmos utilizan un array para almacenar los elementos y dos enteros (tope y fondo) para indicar la posición de ambos extremos. Al inicializar, el tope estará a 0 y el fondo a -1, una cola estará vacía si el fondo es menor que el tope, al hacer inserción se aumenta el fondo y se almacena el elemento en la posición del fondo y al borrar se aumenta el tope.

Con esto dicho, vemos que habrán huecos donde no se puede insertar, la forma de solucionarlo es que cada vez que se borra un elemento, el resto se desplaza una posición a la izquierda para que siempre esté en la primera posición, pero esto aumentará la complejidad de la operación desencolar. También se pueden usar colas circulares, donde un array actúa como un círculo donde la primera posición sigue a la última, en este caso la condición de vacía sería cuando tope es igual a fondo.

Es muy sencillo de implementar pero presenta la desventaja de un tamaño máximo de la cola.

- Representación enlazada (internamente punteros a nodo): se construye a partir de tipos base (punteros a nodo) o definidos por el usuario (una lista usando herencia o layering), de esta forma solventamos el problema del tamaño máximo ya que no hace falta definirlo.

Usando colas circulares enlazadas solo hace falta un puntero ya que el siguiente elemento apuntado por el fondo es el primero en desencolar.

Sintaxis

Modulo cola
Usa BOOL
Parámetro tipo Item
Operaciones del parametro
 error_item()→item
FParamétro
Tipo cola
Operaciones

Crear()→cola	encolar(cola,item)→cola
desencolar(cola)→cola	cabeza(cola)→item
esvacia(cola)→bool	concatenar(cola,cola)→cola

Semántica

VAR c,q:cola; x:item;

Ecuaciones

Crear()→cola	encolar(cola,item)→cola
f. canónica se denotara con CR	f. canónica, se denotara con EC
desencolar(cola)→cola	cabeza(cola)→item
quita el elemento del tope,	obtiene el elemento del tope,
se denotara con DEC	se denotara con TOP
DEC(CR())=CR()	TOP(CR())=error_item()
DEC(AP(CR(),x))=CR()	TOP(AP(CR(),x))=x
DEC(AP(c,x))=DEC(c)	TOP(AP(c,x))=TOP(c)
esvacia(pila)→bool	concatenar(cola,cola)→cola
comprueba si es vacia,	concatena dos colas,
se denotara con EV	se denotara con CNC
EV(CR())=TRUE	CNC(CR(),c)=c
EV(EC(c,x))=FALSE	CNC(c,CR())=c
	CNC(EC(c,x),q)=EC(CNC(c,q),x)

FModulo

2. La eficiencia de los algoritmos

2.1. Noción de complejidad

El calculo de complejidad es la determinacion de dos parametros o funciones de coste:

- Complejidad espacial: cantidad de recursos espaciales (de almacen) que un algoritmo consume o necesita para su ejecucion.
- Complejidad temporal: cantidad de tiempo que un algoritmo necesita para su ejecucion.

Con esto podemos valorar el algoritmo o compararlo con otros.

Vamos a ver la complejidad temporal con mas detalle.

2.1.1. Complejidad temporal, tamaño del problema y paso

Para calcular la complejidad temporal tenemos que tener en cuenta los siguientes factores:

- Externos: como la maquina en la que se va ejecutar, el compilador (variables y modelo de memoria) o la experiencia del programador.

- Internos: el numero de instrucciones asociadas al algoritmo.

La funcion general que se usa es $Tiempo(A) = C + f(T)$ donde C es la contribucion de los factores externos (es constante) y $f(T)$ es una funcion que depende de T (talla o tamaño del problema).

- Talla o tamaño de un problema: es el valor o conjunto de valores asociados a la entrada del problema que representa una medidad de su tamaño respecto de otras entradas posibles.
- Paso de programa: es la secuencia de operaciones con contenido semantico cuyo coste es independiente de la Talla del problema, es la unidad de medida de la complejidad de un algoritmo.
- Expresion de la complejidad temporal: es la funcion que expresa el numero de pasos de programa que un algoritmo necesita ejecutar para cualquier entrada posible(para cualquier talla posible), no se tendran en cuenta los factores externos.

No pondremos ejemplos ahora ya que mas adelante veremos que hay mejores y peores casos, es ahi cuando veremos ejemplos.

Cabe decir que solo nos ocuparemos de la complejidad temporal y que normalmente la complejidad temporal y la espacial son objetivos contrapuestos.

El calculo de la complejidad temporal se considerara a priori (contando los pasos) o a posteriori (generando instancias para distintos valores y cronometrando), tratandose de obtener una funcion donde las unidades de medida no son relevantes (todo se traduce a un cambio de escala). El numero de pasos que se ejecutan siempre es en funcion del tamaño (o talla) del problema.

2.2. Cotas de complejidad

Cuando aparecen diferentes casos para una misma talla generica n , se introducen las cotas de complejidad:

- Caso peor: cota superior del algoritmo $\rightarrow C_s(n)$
- Caso mejor: cota inferior del algoritmo $\rightarrow C_i(n)$
- Termino medio: cota promedio del algoritmo $\rightarrow C_m(n)$

Todas dependen del tamaño del problema n , teniendo que la cota promedio es dificil de evaluar a priori, ya que es necesario conocer la distribucion de la probabilidad de entrada y ademas no es la media de las cotas superior e inferior.

En resumen, la cota promedio es complicada de obtener por lo tanto solo se hablara de ella cuando la inferior y la superior coinciden, ademas, el estudio de la complejidad se hace para tamlos grandes del problema ya que los resultados para tamaños pequeños no son fiables o no proporcionan suficiente informacion. Tambien por el hecho de que si invertimos tiempo en el desarrollo de un buen algoritmo, este sea capaz de realizar un gran volumen de operaciones.

A este tipo e complejidad que trata con tamaños grandes se le denomina complejdiad asintotica y la notacion utilizada es la notacion asintotica.

Para resolver problemas, es importante fijarse en condiciones de parada influencias por valores externos o si estos influncian a la repeticion de una secuencia de operaciones. Estos indicaran el tamaño y que caso es el problema.

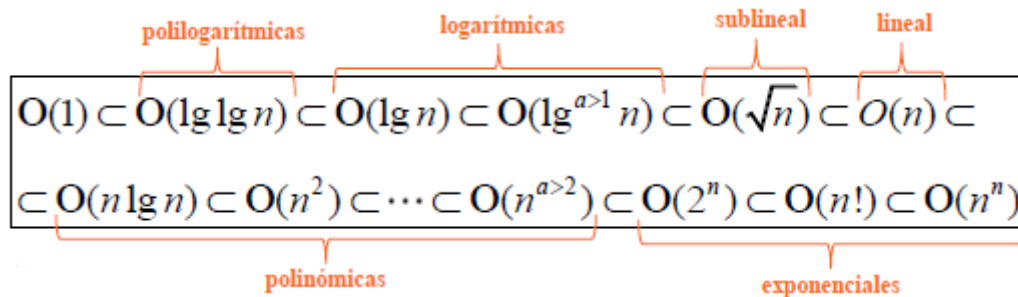
2.3. Notación asintótica

Como ya se ha dicho, usamos la notación asintótica para cuando n se considera muy grande, tanto como para asumir que tiende a infinito ($n \rightarrow \infty$), definiremos tres para los distintos casos:

- Notación O (big-omicron): caso peor
- Notación Ω (omega): caso mejor
- Notación Θ (big-theta): caso promedio

Por lo general, usaremos la notación big-omicron para definir un caso sin confirmar.

La escala de complejidad es:



Cabe decir que una función compuesta pertenecerá al rango de mayor grado

2.4. Obtención de cotas de complejidad

Veamos como afrontar un problema a través de las siguientes etapas para obtener las cotas de complejidad, que son:

1. Determinación de la talla o tamaño (de la instancia) del problema
2. Determinación del mejor y peor caso: instancias para las que el algoritmo tarda más o menos (instancias de la talla). Es posible no existan mejor y peor casos ya que para cualquier instancia del mismo tamaño el algoritmo actúe siempre igual.
3. Obtención de las cotas para cada paso, ya sea contando los pasos o con relaciones de recurrencia (funciones recursivas).

Usemos unos algoritmos de ordenación como ejemplo:

- Inserción directa: divide lógicamente el vector en dos partes: origen y destino. Al comenzar, el destino tiene el primer elemento del vector mientras que el origen tiene los $n - 1$ restantes.

Se irá tomando el primer elemento de origen y se insertará en destino en el lugar adecuado, de forma que destino siempre estará ordenado. El algoritmo finaliza cuando no queden elementos en origen.

```

funcion INSERCIÓN_DIRECTA (var a:vector[natural]; n: natural)
var i,j: entero; x:natural fvar
comienzo
  para i:=2 hasta n hacer
    x:=a[i]; j:=i-1
    mientras (j>0) ^ (a[j]>x) hacer
      a[j+1]:=a[j]
      j:=j-1
    a[j+1]:=x
  fpara
fin
  
```

1. La talla viene en función al tamaño del vector, que denotaremos con n ya que el código realizara una diferente cantidad de secuencias según este tamaño. Cabe decir que para un mismo tamaño de vector, el contenido de este puede variar. Es en este suceso donde veremos las diferentes instancias para una misma talla.
2. Analizando el código, vemos que en f_2 existe una condición que provoca la finalización de un bucle que de normal haría unas n secuencias. Si la condición se cumple, estaremos en el peor caso que ronda a las n secuencias mientras que si no se cumple, será el mejor caso (en especial si se cumple a la primera).
3. Obtengamos ahora las cotas para cada paso,

$$T(n) = \begin{cases} \sum_{i=2}^n (C_1 + f_2(i-1)) \\ f_2(n) = \begin{cases} \sum_{i=0}^n (C_2) \in O(n) \text{ peor caso} \\ C_2 \in \Theta(1) \text{ mejor caso} \end{cases} \end{cases} \quad T(n) = \begin{cases} \sum_{i=2}^n (C_1 + O(n)) \in O(n^2) \\ \sum_{i=2}^n (C_1 + \Theta(1)) \in \Theta(n) \end{cases}$$

- Me da pereza hacer las demás

3. El tipo árbol

Vamos a comenzar recordando brevemente que es un árbol, dado previamente en la asignatura de Matemáticas Discreta. Podéis verlo con más detalle en los apuntes de esa asignatura.

Los árboles son grafos conexos y acíclicos y diremos que es un árbol generador si es árbol y subgrafo generador a la vez.

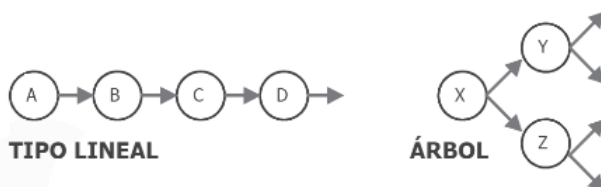
1. Dos vértices cualesquiera están conectados por un único camino.
2. Un grafo será conexo si y solo si tiene un árbol generador.
3. El número de relaciones en un árbol difiere en uno al número de vértices de este
4. Todo árbol no trivial tiene al menos 2 vértices de grado 1.

Dado un vértice r_0 , el cual actúa como extremo inicial, únicamente y todo vértice está conectado con éste, podemos definir un árbol enraizado a r_0 y distinguiremos los distintos niveles o alturas n a los vértices cuyos caminos sean de longitud n .

Cabe decir que, aunque es recomendable conocer que es un árbol en un contexto matemático, no es necesario.

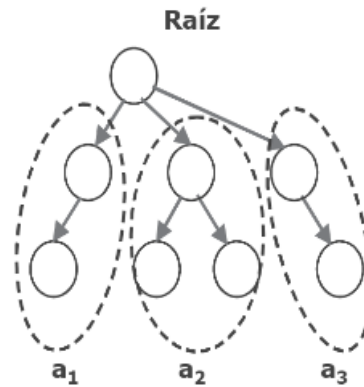
3.1. Definiciones generales

Diremos que la estructura de datos del tipo árbol aparece a la hora de establecer una jerarquía entre los elementos que lo constituyen, obtenido a partir de eliminar el requisito de que cada elemento en una estructura de datos del tipo lineal solo podría tener como máximo un sucesor. Llamaremos a los elementos de estructura como **nodos**.

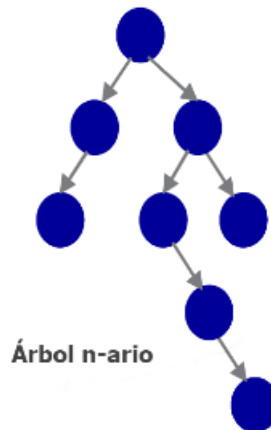


Procedamos a enlistar una serie de propiedades y definiciones respecto al concepto de la estructura del tipo árbol:

- Un único nodo puede constituir un árbol, además de ser la raíz de éste.
- Diremos que un **árbol es vacío** o **nulo** si este tiene 0 nodos, es decir, no tiene una raíz.
- La información almacenada dentro del nodo se denomina **etiqueta**.
- Dados n árboles a_1, \dots, a_n se puede construir uno nuevo al enraizar todos árboles a un nuevo nodo, pasando estos a ser **subárboles** del nuevo árbol, siendo el nuevo nodo la raíz de éste.

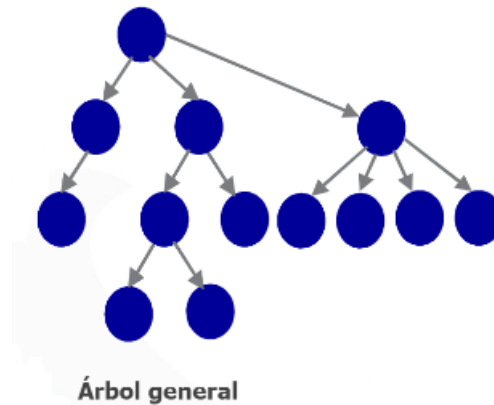


- **Árbol n-ario** es un arraigado árbol en el que cada nodo no tiene más que n sucesores. Se podría decir que todo árbol solo podrá tener como máximo n árboles. Lo llamaremos **árbol binario** cuando $n = 2$.

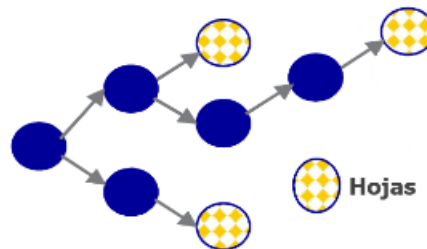


Es más, llamaremos grado de un árbol al número máximo de sucesores que podrán tener los subárboles, siendo en el árbol n -ario un grado de n .

- **Árbol general** es un árbol sin una restricción en el número de sucesores por nodos, es decir, que podrá tener una cantidad de subárboles indeterminada.

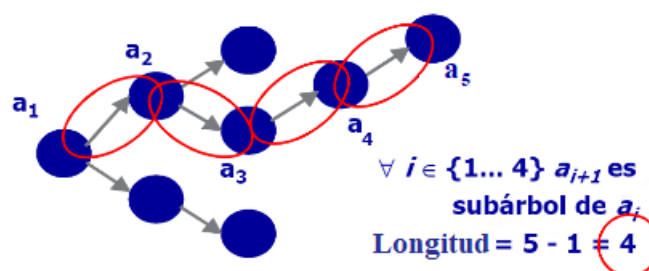


- Llamaremos **hojas** a los arboles compuestos por un único nodo, o dicho de otra manera, cuando un nodo no tiene sucesores.



- **Camino:**

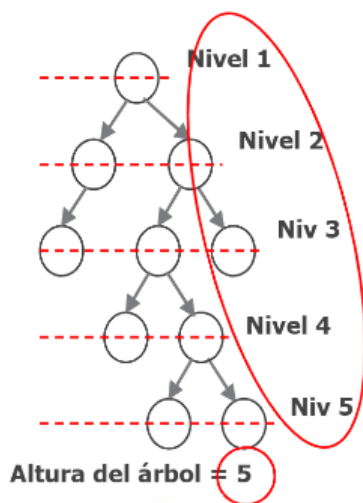
- una secuencia a_1, \dots, a_s de árboles tal que $\forall i \in \{1, \dots, s-1\}$, a_{i+1} es subárbol de a_i . Pueden darse casos de caminos de un árbol a si mismo, siendo la sucesión el propio árbol.
- La **longitud** del camino corresponde a la cantidad árboles menos uno ya. Se considera que la longitud de un árbol a si mismo es de 0. Esto se debe a que los caminos consisten en enlistar los nodos que son sucesores pero la longitud es el numero de sucesiones necesarias para alcanzar desde la raíz el ultimo elemento del camino.



- Terminologías para sucesores:

- Diremos que x es **ascendiente** de y si existe un camino con x como la raíz del árbol e y uno de los sucesores directos o indirectos de x . También podemos decir que y es **descendiente** de x .
Debido a la definición de camino, todo árbol es ascendiente(descendiente) de sí mismo.
- Diremos que son **ascendiente(descendientes) propios** todos los ascendiente(descendientes) excluidos del propio árbol.
- **Padre** es el primer ascendiente propio de un árbol. No todos los arboles tienen padre.

- **Hijos** son los primeros descendientes propios, si existen, de un árbol.
- **Hermanos** son los subárboles con el mismo padre.
- **Profundidad** de un subárbol es la longitud del único camino desde la raíz a dicho subárbol. Recordad que un árbol/subárbol consiste en el nodo y sus sucesores.



● **Nivel** de un nodo:

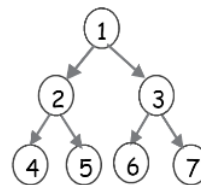
- el nivel de un árbol vacío es 0
- el nivel de la raíz es 1
- si un nodo está en el nivel i , sus hijos están en el nivel $i + 1$

● **Altura(profundidad)** de un árbol:

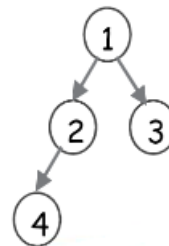
- es el máximo nivel de los nodos de un árbol

Hay casos en los que la gente inicializa la raíz al nivel 0 o el nivel mas bajo es donde esta la hoja con el camino de mayor longitud a la raíz (de igual manera, pudiendo ser el nivel inicial 1 o 0). Y la altura, según la definición dada, se aplicaría sin ningún problema a la forma que se decida usar. Para esta asignatura se considerara lo mostrado previamente, pero considerar esto como posible ya que la gente puede hacer lo que quiera siempre que se explique al principio.

● **Árbol lleno** es árbol en el que todos sus subárboles tienen n hijos (con n siendo el grado del árbol) y todas sus hojas tienen la misma profundidad.



● **Árbol completo** es un árbol cuyos nodos corresponden a los nodos numerados (la numeración se realiza desde la raíz hacia las hojas y, en cada nivel, de izquierda a derecha) de 1 a n en el árbol lleno del mismo grado, siendo todo lleno uno completo.



En algunos casos, se dice que un árbol completo n -ario cuando todos los nodos tienen n sucesores o son hoja, parecido a lo que sería un árbol lleno, pero aquí se dice que es un árbol que se va llenando de izquierda a derecha por nivel, nunca se podrá tener una hoja a profundidad x y otra a un nivel inferior a $x - 1$.

3.2. Árboles Binarios

El árbol binario es un caso particular de árbol n -ario donde $n = 2$. Podemos definir lo como un conjunto de elementos del mismo tipo tal que, distinguiremos un elemento que llamaremos raíz y el resto de elementos del conjunto se distribuyen en dos subconjuntos disjuntos, siendo estos a su vez dos arboles binarios. Si el árbol es un conjunto vacío, es decir, no hay elementos, será un árbol vacío o nulo.

Para entendernos, la idea es que tu tienes el árbol que esta compuesto por un nodo o ninguno, y el nodo siempre se esperara que tenga 2 arboles asociados a él, pudiendo estar estos con o sin nodo.

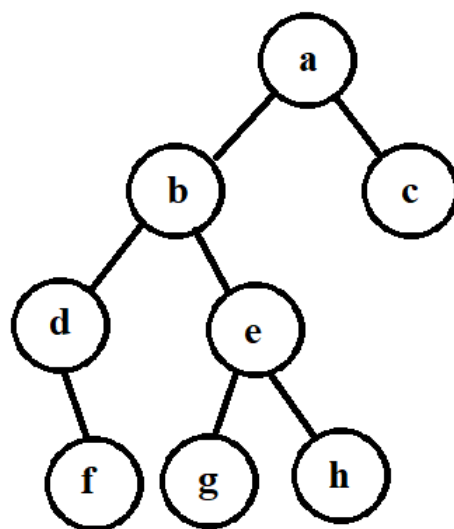
Vamos a definir una serie de propiedades:

- El máximo numero de nodos en un nivel i de un árbol binario será $N(i) = 2^{i-1}, i \geq 1$. Podemos probarlo para $N(1) = 2^0 = 1$ como la base, y, asumiendo que es cierto para $N(i-1) = 2^{(i-1)-1} = 2^{i-2}$, podemos inducir, como para el siguiente nivel serán el doble de nodos, $N(i) = N(i-1) * 2 = 2^{i-2+1} = 2^{i-1}$. Haremos uso de esta función en el documento usando $n(i)$
- El máximo numero de nodos en un árbol binario de altura k es $N(k) = 2^k - 1, k \geq 1$. Básicamente se obtiene de hacer el sumatorio del numero máximo de nodos por nivel, es decir, $\sum_{i=1}^k n(i) = 2^0 + 2^1 + \dots + 2^{k-1} = 2^k - 1$. Cuando hagamos referencia a esta formula en el documento usando $h(i)$.

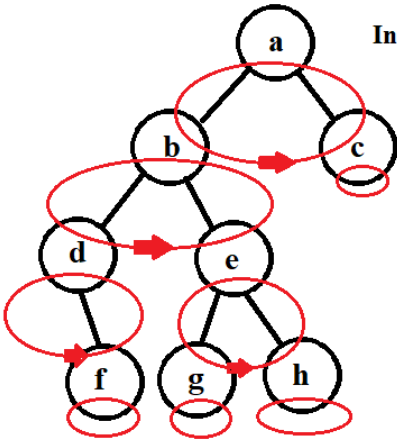
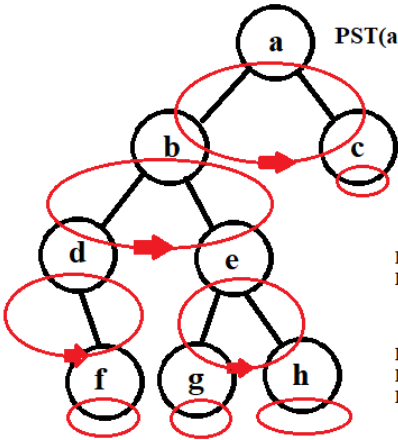
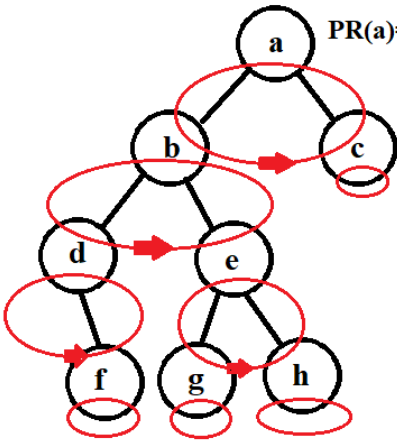
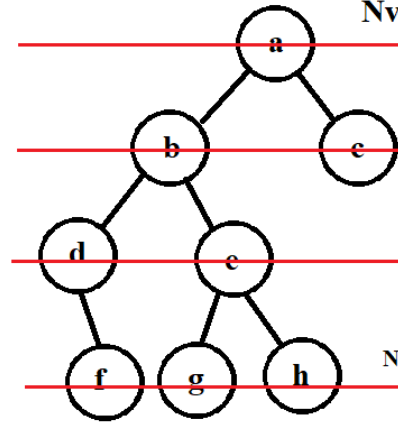
Llamamos **recorrer** un árbol a la acción de visitar cada nodo del árbol una sola vez, siendo el **recorrido** la lista de etiquetas del árbol ordenado según se visitan los nodos. Distinguiremos dos categorías, **recorridos en profundidad** y **recorridos en anchura o por niveles**.

- Recorrido en profundidad: representando desplazarse por el árbol por la izquierda con **I**, desplazarse por la derecha como **D** y acceder a la etiqueta del nodo como **R**, hay 6 formas de recorrerlo diferenciados por el sentido, de derecha a izquierda (**DIR, RDI, DRI**) y de izquierda a derecha (**IDR, IRD, RID**), veremos estos tres últimos con mas detalle:
 - **IDR** Postorden o orden posterior
 - **IRD** Inorden o orden simétrico
 - **RID** Preorden o orden previo
- Recorrido en anchura: consiste en visitar los nodos desde la raíz hacia las hojas, de izquierda a derecha dentro de cada nivel.

Veamos un ejemplo de los distintos recorridos para un mismo árbol, siendo el siguiente:



Notese que este árbol no es uno completo pero si recibe la numeración apropiada.

Inorden	Postorden
 <p> $In(a)=In(b)+a+In(c)$ $In(b)=In(d)+b+In(e)$ $In(c)=null+c+null$ $In(d)=null+d+In(f)$ $In(e)=In(g)+e+In(h)$ $In(f)=null+f+null$ $In(g)=null+g+null$ $In(h)=null+h+null$ </p> <p>In(a)=d f b g e h a c</p>	 <p> $PST(a)=PST(b)+PST(c)+a$ $PST(b)=PST(d)+PST(e)+b$ $PST(c)=null+null+c$ $PST(d)=null+PST(f)+d$ $PST(e)=PST(g)+PST(h)+e$ $PST(f)=null+null+f$ $PST(g)=null+null+g$ $PST(h)=null+null+h$ </p> <p>PST(a)=f d g h e b c a</p>
Preorden	Niveles
 <p> $PR(a)=a+PR(b)+PR(c)$ $PR(b)=b+PR(d)+PR(e)$ $PR(c)=c+null+null$ $PR(d)=d+null+PR(f)$ $PR(e)=e+PR(g)+PR(h)$ $PR(f)=f+null+null$ $PR(g)=g+null+null$ $PR(h)=h+null+null$ </p> <p>PR(a)=a b d f e g h c</p>	 <p> $Nv(1)=a+Nv(2)$ $Nv(2)=b+c+Nv(3)$ $Nv(3)=d+e+Nv(4)$ $Nv(4)=f+g+h+Nv(5)$ $Nv(5)=null$ </p> <p>Nv(1)=a b c d e f g h</p>

Programación y estructuración de un árbol binario

Definamos la sintaxis y la semántica para este tipo de dato, no sin antes añadir un listado de conceptos a tener en cuenta:

- **Crear():***ab*, devuelve un árbol vacío.
- **Enraizar(ab,item,ab):***ab*, devuelve un árbol con item como raíz que contiene los dos subárboles indicados.
- **raiz(ab):***item*, devuelve el item que se encuentra en la raíz del arbol. Puede darse el caso que el árbol este vacío y la raíz sea nula.
- **esVacio(ab):***bool*, comprueba si un árbol es vacío comprobando únicamente si su raíz es nula.

- **hijoiz, hijode(ab):ab**, devuelve el subárbol enraizado a la izquierda o la derecha del árbol. Si el árbol es nulo, entonces se devolverían árboles vacíos.
- **altura(ab):int**, devolverá la longitud del camino que vaya desde la raíz a una de sus hojas que sea mayor.
- **nodosHoja(ab):int**, devolverá el numero de nodos que no están enraizados a arboles no vacíos. Cabe decir que un árbol vacío no tiene nodos.
- **simétricos(ab,ab):bool**, comprobara si dos arboles son simétricos (reflexión de espejo).
- **iguales(ab,ab):bool**, similar a **simétricos**, comprobara si el contenido de dos arboles es idéntico y tienen la misma estructura.
- **quita_hojas(ab):ab**, dado un árbol, devolverá otro el cual no contenga los nodos hoja del introducido.
- Tipos de recorrido de profundidad:
 - **Inorden(ab):lista**, usando el recorrido en inorden, se van añadiendo a la lista las etiquetas que se van leyendo durante el recorrido.
 - **Preorden(ab):lista**, usando el recorrido en preorden, se van añadiendo a la lista las etiquetas que se van leyendo durante el recorrido.
 - **Postorden(ab):lista**, usando el recorrido en postorden, se van añadiendo a la lista las etiquetas que se van leyendo durante el recorrido.
- Recorrido en anchura:
 - **Niveles(ab):lista**, usando el recorrido en niveles, se van añadiendo a la lista las etiquetas que se van leyendo durante el recorrido.
- Tipos de representación:
 - Representación secuencial: consiste en llenar los nodos hipotéticamente numerándolos de izquierda a derecha por nivel, desde el nivel 1. De esta forma, cuando añadamos un item al árbol en una posición, esta posición ya estaría considerada previamente a la asignación.
 - Representación enlazada: cada árbol tiene una dirección única y, recordando que todo árbol contiene un nodo el cual apunta a otros dos arboles, se puede asignar un subárbol al nodo apropiado e ir modificándose en el caso de se deba de hacer.
- Asignacion(copia) entre arboles e iteradores:
 - **ab::operator=(ab):void**, copia el arbol completamente en el árbol al que se asigna.
 - **ab::operator=(ab_iter):void**, copia la rama a la que apunta el iterador en el árbol al que se asigna.
 - **ab_iter::operator=(ab):void**, copia el árbol en la rama a la que apunta el iterador al que se asigna.
 - **ab_iter::operator=(ab_iter):void**, sirve para inicializar el iterador al que se asigna para que apunte al mismo iterador.
- # Esto se traduciría usando puntero e ir accediendo a los subárboles de un árbol.
- Movimiento de ramas entre arboles e iteradores:

- **mover(ab,ab):void**, el contenido del segundo árbol pasaría a estar en el primero, el segundo queda vacío.
- **mover(ab,ab_iter):void**, el contenido de la rama a la que apunta el iterador pasara a ser el contenido del árbol.
- **mover(ab_iter,ab):void**, mueve el árbol a la posición del iterador, dejando el árbol vacío.
- **mover(ab_iter,ab_iter):void**, el contenido de la rama del segundo sustituirá la rama a la que apunta el primero iterador.

Esto se traduciría usando puntero e ir accediendo a los subárboles de un árbol.

En los apuntes del profesorado hay mas funciones como **todos**, **transforma**, **dos_hijos** (aunque modificado se podría usar para comprobar si un árbol es lleno o completo),. Los que se muestran son los considerados de mayor utilidad.

Definamos ahora la sintaxis y la semantica para un tipo arbol:

Sintaxis

Modulo Arbol_Binario

Usa BOOL, NATURAL, LISTA

Parámetro tipo Item

Operaciones del Item

error_item()→item

FParamétro

Tipo ab

Operaciones

Crear()→ab

raiz(ab)→item

hijoiz,hijode(ab)→ab

nodosHoja(ab)→int

iguales(ab,ab)→bool

inorden(ab)→lista

postorden(ab)→lista

Operaciones que no definirán en la semántica:

asignacion(ab,ab)→void

Enraizar(ab,item,ab)→ab

esVacío(ab)→bool

altura(ab)→int

simetricos(ab,ab)→bool

quita_hojas(ab)→ab

preorden(ab)→lista

niveles(ab)→lista

mover(ab,ab)→void

Semántica

VAR i,d: ab; x:item;

Ecuaciones

Crear()→ab

f. canónica se denotara con CR

raiz(ab)→item

raiz(CR())=error_item

raiz(ER(i,x,d))=x

hijoiz,hijode(ab)→ab

hijoiz(CR())=CR()

hijoiz(ER(i,x,d))=i

hijode(CR())=CR()

hijode(ER(i,x,d))=d

Enraizar(ab,item,ab)→ab

f. canónica, se denotara con ER

esVacío(ab)→bool

esVacío(ab)=T

esVacío(ER(i,x,d))=F

altura(ab)→int

altura(CR())=0

altura(ER(i,x,d))=

1+max(altura(i),altura(d))

$\text{iguales}(\text{ab}, \text{ab}) \rightarrow \text{bool}$ $\text{iguales}(\text{CR}(), \text{CR}()) = \text{V}$ $\text{iguales}(\text{CR}(), \text{d}) = \text{F}$ $\text{iguales}(\text{d}, \text{CR}()) = \text{F}$ $\text{iguales}(\text{i}, \text{d}) =$ $\text{SI raiz}(\text{i}) = \text{raiz}(\text{d})$ ENTONCES $\text{iguales}(\text{hijoiz}(\text{i}), \text{hijoiz}(\text{d})) \& \&$ $\text{iguales}(\text{hijode}(\text{i}), \text{hijode}(\text{d}))$ SINO F $\text{nodosHoja}(\text{ab}) \rightarrow \text{int}$ se denotara con NH $\text{NH}(\text{CR}()) = 0$ $\text{NH}(\text{ER}(\text{CR}(), \text{x}, \text{CR}())) = 1$ $\text{NH}(\text{ER}(\text{i}, \text{x}, \text{d})) = \text{NH}(\text{i}) + \text{NH}(\text{d})$ $\text{inorden}(\text{ab}) \rightarrow \text{lista}$ se denotara con IN $\text{IN}(\text{CR}()) = \text{CR_lista}()$ $\text{IN}(\text{ER}(\text{i}, \text{x}, \text{d})) = \text{conc}(\text{insDe}(\text{IN}(\text{i}), \text{x}), \text{IN}(\text{d}))$ $\text{postorden}(\text{ab}) \rightarrow \text{lista}$ se denotara con PS $\text{PS}(\text{CR}()) = \text{CR_lista}()$ $\text{PS}(\text{ER}(\text{i}, \text{x}, \text{d})) = \text{insDe}(\text{conc}(\text{PS}(\text{i}), \text{PS}(\text{d})), \text{x})$	$\text{simetricos}(\text{ab}, \text{ab}) \rightarrow \text{bool}$ se denotara con SMT $\text{SMT}(\text{CR}(), \text{CR}()) = \text{V}$ $\text{SMT}(\text{CR}(), \text{d}) = \text{F}$ $\text{SMT}(\text{d}, \text{CR}()) = \text{F}$ $\text{SMT}(\text{i}, \text{d}) =$ $\text{SI raiz}(\text{i}) = \text{raiz}(\text{d})$ ENTONCES $\text{SMT}(\text{hijoiz}(\text{i}), \text{hijode}(\text{d})) \& \&$ $\text{SMT}(\text{hijode}(\text{i}), \text{hijoiz}(\text{d}))$ SINO F $\text{quita_hojas}(\text{ab}) \rightarrow \text{ab}$ se denotara con QH $\text{QH}(\text{CR}()) = \text{CR}()$ $\text{QH}(\text{ER}(\text{CR}(), \text{x}, \text{CR}())) = \text{CR}()$ $\text{QH}(\text{ER}(\text{i}, \text{x}, \text{d})) = \text{ER}(\text{QH}(\text{i}), \text{x}, \text{QH}(\text{d}))$ $\text{preorden}(\text{ab}) \rightarrow \text{lista}$ se denotara con PR $\text{PR}(\text{CR}()) = \text{CR_lista}()$ $\text{PR}(\text{ER}(\text{i}, \text{x}, \text{d})) = \text{conc}(\text{insIZ}(\text{x}, \text{PR}(\text{i})), \text{PR}(\text{d}))$ $\text{niveles}(\text{ab}) \rightarrow \text{lista}$ se denotara con NV $\text{NV}(\text{CR}()) = \text{CR_lista}()$ $\text{NV}(\text{ER}(\text{i}, \text{x}, \text{d})) = \text{PENDIENTE}$
Funciones usadas de tipo lista :	
$\text{CR_lista}() \rightarrow \text{lista}$	$\text{conc}(\text{lista}, \text{lista}) \rightarrow \text{lista}$
crea una lista vacia	concatena dos listas
$\text{insIZ}(\text{item}, \text{lista}) \rightarrow \text{lista}$	$\text{insDe}(\text{lista}, \text{item}) \rightarrow \text{lista}$
inserta el item a la izquierda de la lista	inserta el item a la derecha de la lista

FModulo

3.3. Árboles de búsqueda

Los árboles de búsqueda, que también pueden ser árboles n-arios de búsqueda o árboles multicamino de búsqueda, son un tipo particular de arboles que pueden definirse cuando el tipo de los elementos del árbol posee una relación de orden total.

Para aclarar, un arbol multicamino de busqueda, que denotaremos como T es un arbol n-ario vacio o que cumple las siguientes propiedades:

- La raíz de T contiene A_0, \dots, A_{n-1} subárboles y K_1, \dots, K_{n-1} etiquetas.

NOTA personal recordad que tratamos con árboles n-arios, un árbol que contiene un nodo el cual puede tener un máximo de n sucesores tratados como arboles, el nodo seria T , la raíz del árbol pero vería con más sentido que tuviésemos K_0, \dots, K_{n-1} y A_1, \dots, A_{n-1} siendo K_0 la etiqueta de la raíz. La forma en la que se define la veo rara pero mas adelante veremos que esto se debe a que habrán tantas etiquetas como sucesores que tenga T y no hace falta considerar la etiqueta para K_n , que estaría asociado con el árbol A_{n-1} ya que por el algoritmo. Dicho de otra manera, la etiqueta K_i correspondería al nodo raíz del subárbol A_{i-1} , con $1 \leq i \leq n$.

- $K_i < K_{i+1}$, $1 \leq i < n - 1$
- Todas las etiquetas del subárbol A_i son:

menores que K_{i+1} , $0 \leq i < n - 1$

mayores que K_i , $0 < i \leq n - 1$

- Los subárboles A_i , $0 \leq i \leq n - 1$ son también árboles multicamino de búsqueda.

El algoritmo de búsqueda a seguir se basa en los siguientes pasos:

- Para buscar un valor x en el árbol, primero se mira el nodo raíz y se realiza la comparación respecto a la relación total(usaremos por comodidad la relación de orden total \leq).

1 $x = K_i$ Hemos encontrado el elemento.

2 Si $x < K_i$, por la definición de un árbol multicaminos x debe de estar en el subárbol A_{i-1} en el caso de que x exista en éste. Si no lo hiciera, probaríamos con $i = i + 1$.

3 $x > K_i$, de igual manera, asumimos que x debe de estar en el subárbol A_i . Si no lo hiciera, probaríamos con $i = i + 1$.

Los arboles multicamino son útiles cuando la memoria principal es insuficiente para utilizarla como almacenamiento permanente y, en el caso de usar una representacion enlazada, los punteros pueden representar direcciones de disco en lugar de direcciones de memoria principal.

3.3.1. Árboles binarios de búsqueda

Cuando el arbol de busqueda es binario, veremos las siguientes propiedades:

- Todos los elementos en el subárbol de la izquierda son previos a la raíz con respecto a la relación de orden total. Ésto se debe a que, con las definiciones y propiedades previas (y definiendo la relación como leq), $x \leq K_1$ teniendo todos los elementos previos organizados en el subárbol A_0 .
- Todos los elementos en el subárbol de la izquierda son posteriores a la raíz con respecto a la relación de orden total. De igual manera, si se da $K_1 \leq x$ tendremos todos los elementos posteriores organizados en A_1 .
- Ambos subárboles son a su vez binarios de búsqueda y, en algunos casos no se permite la repetición de etiquetas.

Sintaxis

Modulo Arbol_Binario_Busqueda

Usa BOOL, Arbol_Binario

Parámetro tipo Item

Operaciones del Item

$<, \dots, > : \text{item}, \text{item} \rightarrow \text{bool}$

$\text{error_item}() \rightarrow \text{item}$

FParamétro

Operaciones

$\text{Insertar}(\text{ab}, \text{item}) \rightarrow \text{ab}$ | $\text{Buscar}(\text{ab}, \text{item}) \rightarrow \text{bool}$

$\text{Borrar}(\text{ab}, \text{item}) \rightarrow \text{ab}$ | $\text{Min}(\text{ab}) \rightarrow \text{item}$

Para indicar que hay operaciones previamente definidas en el árbol binario se pondrán las terminaciones **OP**_ab detrás de cada operación (**OP**).

Semántica

VAR i,d: ab; x,y:item;

Ecuaciones

Insertar(ab,item)→ab denotado con Ins
 Ins(CR_ab,x)=ER_ab(CR_ab,x,CR_ab)
 Ins(ER_ab(i,x,d),y)=
 si (y<x) **entonces**
 ER_ab(Ins(i,y),x,d)
 si no si (y>x)
 ER_ab(i,x,Ins(d,y))
 fsi

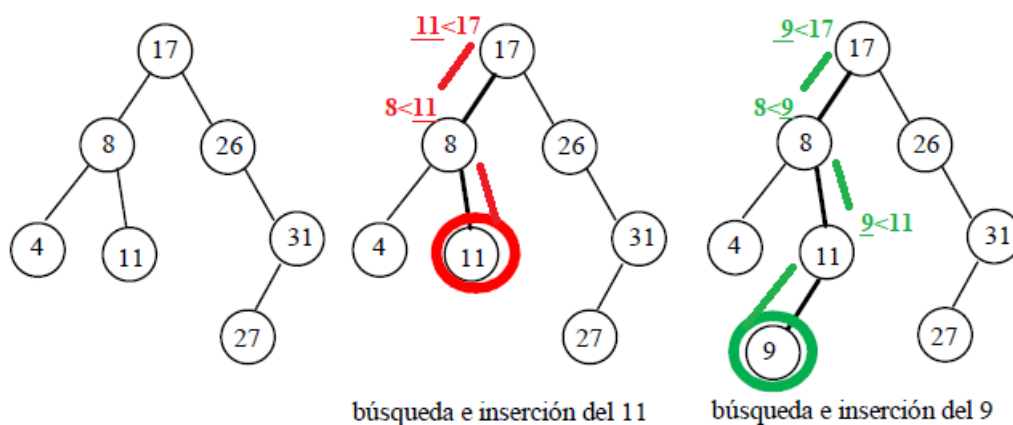
Borrar(ab,item)→ab denotado con Brr
 Brr(CR_ab(),x)=CR_ab()
 Brr(ER_ab(i,x,d),y)=
 si (y<x) **entonces**
 ER_ab(Brr(i,y),x,d)
 si no si (x<y) **entonces**
 ER_ab(i,x,Brr(d,y))
 Brr(ER_ab(i,x,d),y)=
 si (y==x) && esVacío_ab(d) **entonces** i **fsi**
 Brr(ER_ab(i,x,d),y)=
 si (y==x) && esVacío_ab(i) **entonces** d **fsi**
 Brr(ER_ab(i,x,d),y)=
 si (y==x) && esVacío_ab(d) && esVacío_ab(i)
 entonces ER_ab(i,min(d),Brr(d,min(d))) **fsi**

Buscar(ab,item)→bool denotado con Bsc
 Bsc(CR_ab,x)=F
 Bsc(ER_ab(i,x,d),y)=
 si (y<x) **entonces**
 Bsc(i,y)
 si no si (y>x) **entonces**
 Bsc(d,y)
 si no T **fsi**

Min(ab)→item
 Min(CR_ab())=error_item()
 Min(ER_ab(i,x,d))=
 si esVacío_ab(i) **entonces** x
 si no Min(i) **fsi**

FModulo

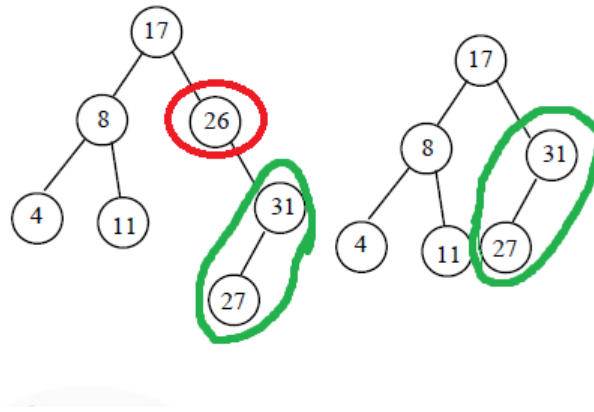
El planteamiento que sigue los algoritmos de búsqueda e inserción de un elemento es el siguiente:



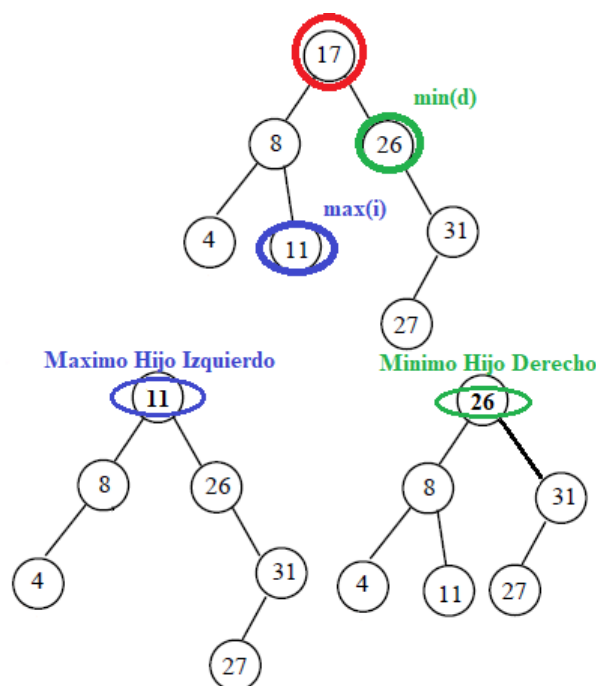
Si el recorrido usado es en orden simétrico (Inorden) se dispondrán las etiquetas en orden ascendente.

Para el borrado de un elemento consideraremos 3 casos:

- El nodo a eliminar es un nodo hoja: se elimina el nodo **Regla 'hoja'**.
- El nodo a eliminar tiene un hijo: el nodo a eliminar se sustituye por su hijo, **Regla '1 hijo'**.



- El nodo a eliminar tiene dos hijos, en estos casos se consideran dos posibles aplicaciones: sustituir por el máximo hijo izquierdo (**Regla '2 hijos, MAX IZQ.'**) y sustituir por el mínimo hijo derecho (**Regla '2 hijos, MIN DCHO.'**).



3.3.2. Árboles AVL

La eficiencia en la búsqueda de un elemento en un árbol binario de búsqueda se mide en términos del número de comparaciones y la altura del árbol.

Un árbol completamente equilibrado es un árbol donde los elementos de este deben estar repartidos en igual número entre el subárbol izquierdo y derecho, de tal forma que la diferencia en número de nodos entre ambos subárboles sea como mucho de 1. El principal problema es el mantenimiento.

Los árboles AVL (Adelson-Velskii y Landis) son árboles balanceados (equilibrados) con respecto a la altura de los subárboles. 'Un árbol está equilibrado si y solo si para cada uno de sus nodos ocurre que las alturas de los dos subárboles difieren en como mucho de 1.'

Este concepto presenta dos consecuencias, un árbol vacío estará siempre equilibrado y un árbol equilibrado es óptimo si cumple que el número de nodos es igual a $2^h - 1$ con h siendo la altura del árbol.

Si consideramos T como un árbol binario no vacío y T_L y T_R como los subárboles izquierdo y derecho, si T está balanceado con respecto a la altura si y solo si

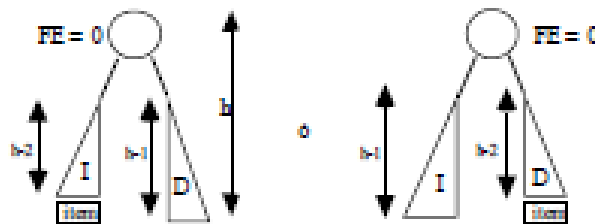
- T_L y T_R son balanceados respecto a la altura
- $|h_R - h_L| \leq 1$ donde h_L y h_R son las alturas de los árboles izquierdo y derecho.

El factor de equilibrio $FE(T)$ de un nodo T en un árbol binario se define como $h_R - h_L$. Para cualquier nodo T en un árbol AVL, se cumple $FE(T) \in \{-1, 0, 1\}$.

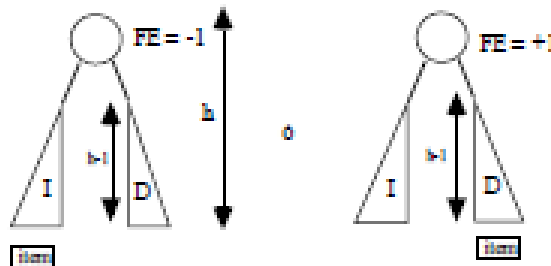
Los árboles AVL se representan con la información necesaria para mantener el equilibrio de forma implícita en la estructura, de esta forma el nodo contiene el FE de forma explícita para mantener dicho equilibrio.

Para la inserción, veremos varios casos:

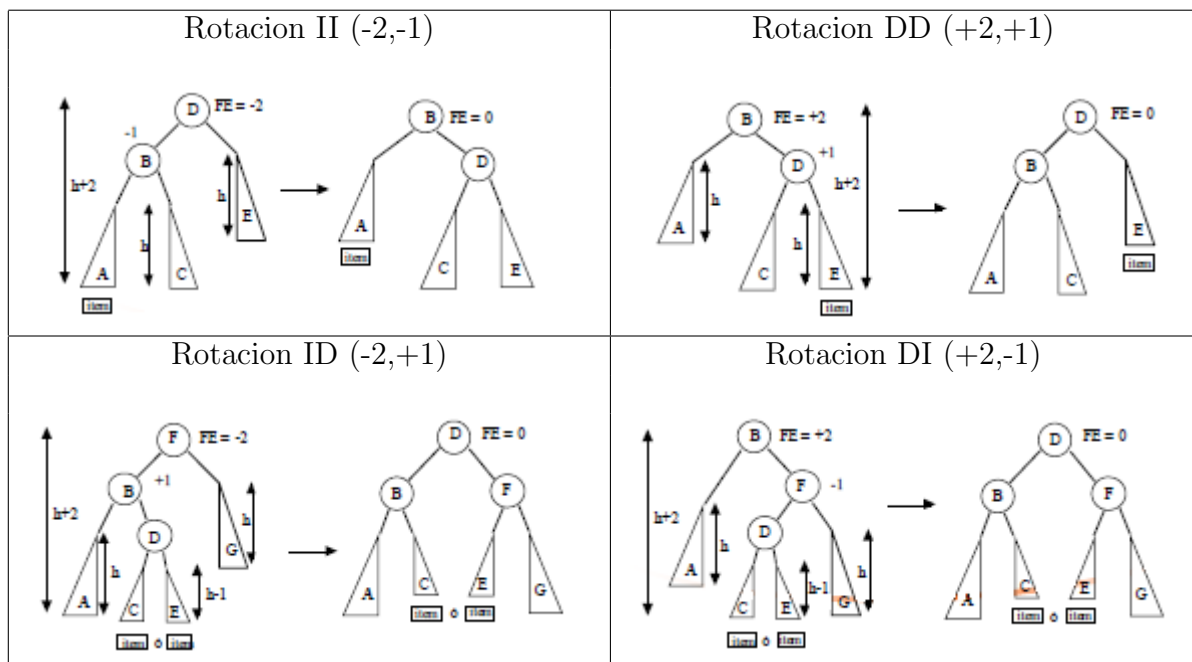
- Después de la inserción del ítem, los subárboles I y D igualarán sus alturas.



- Después de la inserción del ítem, los subárboles I y D tendrán distintas alturas, pero sin vulnerar la condición de equilibrio.



- Si $h_L > h_D$ y se realiza inserción en I , o $h_L < h_D$ y se realiza inserción en D , teniendo varias rotaciones:

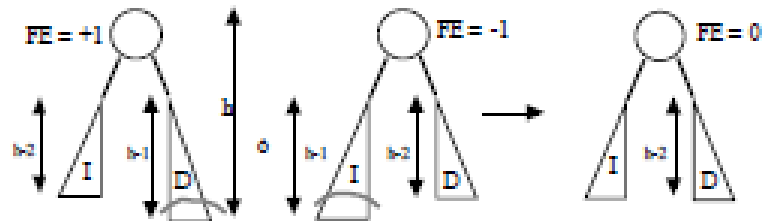


Hay que tener en cuenta que la actualización del FE de cada nodo se efectúa desde las hojas hacia la raíz del árbol.

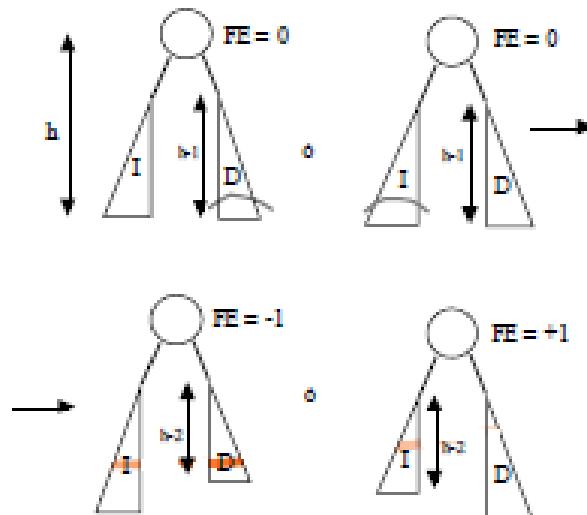
Algoritmo pendiente

Para el borrado, veremos varios casos:

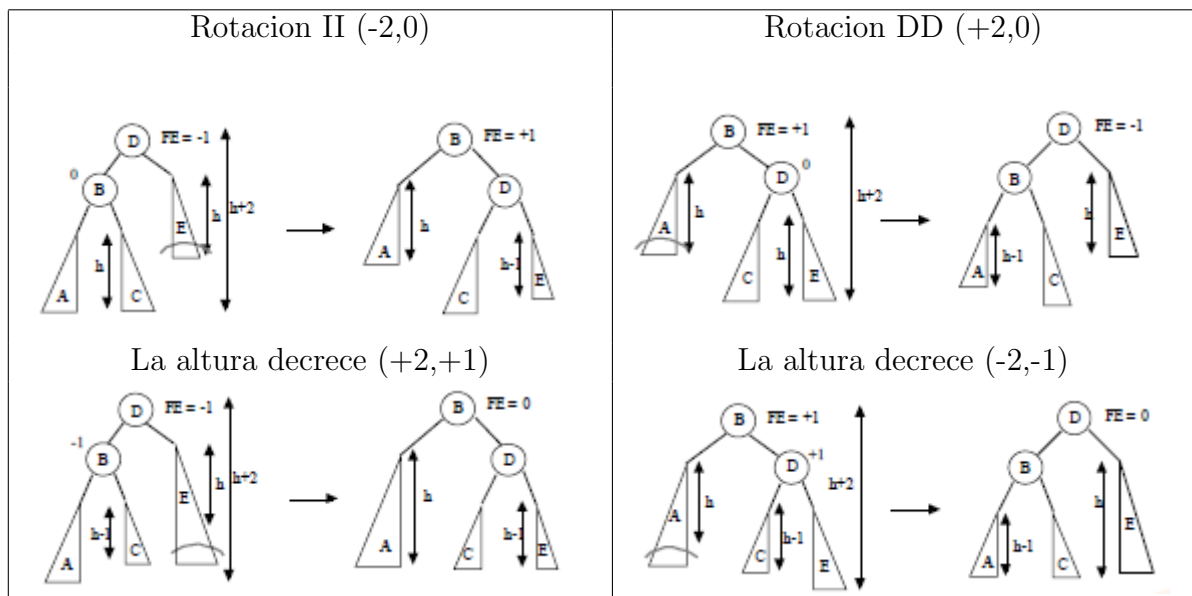
- Al borrar el ítem se equilibrará, con $FE = 0$.



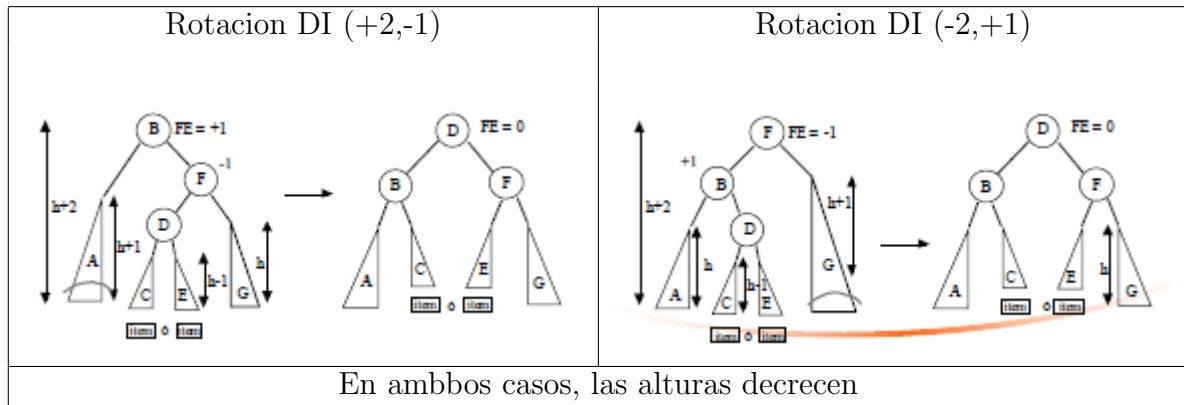
- Al borrar el ítem, llevará el árbol a $FE = \pm 1$, también equilibrado.



- Veamos las rotaciones simples



- Veamos las rotaciones simples



Algoritmo pendiente

Para ambos algoritmos, el estudio de las complejidades consideramos:

- El analisis matematico del algoritmo de insercion es un problema todavia no resuelto. Los ensayos empiricos apoyan la conjetura de que la altura esperada para un arbol deAVL de n nodos es

$$h = \log_2(n) + c$$

, con c siendo una cosntante pequeña.

- Estos arboles deben utilizarse solo si las recuperaciones de informacion (busquedas) son considerablemente mas frecuentes que las inserciones, esto se debe a la complejidad de las operaciones de equilibrado.
- Se puede borrar un elemento en un arbol equilibrado con $\log(n)$ operaciones (en el caso desfavorable)

Las diferencias operacionales de borrado e insercion:

- Al realizar una insercion de una sola clave se puede producir como maximo una rotacion (de dos o tres nodos)
- El borrado puede requerir una rotacion en todos los nodos del camino de busqueda
- Los analisis empiricos dan como resultado que, mientras se presenta una rotacion por cada dos inserciones, solo se necesita una por cada cinco borrados. El borrado en arboles equilibrados es, pues, igual de complejo como la insercion.

3.3.3. Árboles 2-3

prueba

3.3.4. Árboles 2-3-4

prueba

4. Conjuntos

prueba

4.1. Definiciones generales

prueba

4.2. Diccionario

prueba

4.2.1. Tabla de dispersión

prueba

4.3. Cola de prioridad

prueba

4.3.1. Montículo

prueba

4.3.2. Cola de prioridad doble

prueba

5. Grafos

prueba

5.1. Concepto de grafo y terminología

prueba

5.2. Especificación algebraica

prueba

5.3. Representación de grafos

prueba

5.4. Grafos dirigidos

prueba

5.4.1. Recorrido en profundidad o DFS

prueba

5.4.2. Recorrido en anchura o BFS

prueba

5.4.3. Grafos acíclicos dirigidos o GAD

prueba

5.4.4. Componentes fuertemente conexos

prueba

5.5. Grafos no dirigidos

prueba

5.5.1. Algoritmos de recorrido

prueba

6. Seminario de C++

prueba

6.1. Clases y objetos

prueba

6.2. Composición y herencia

prueba

6.3. Constructor y destructor

prueba

6.4. Funciones y clases amigas, reserva de memoria

prueba

6.5. Sobrecarga de operadores

prueba

Ejemplo de referencias:

Referencias

- [1] *X86 Assembly/Floating Point*
- [2] *x86 and amd64 instruction reference*
- [3] *x86/x64 SIMD Instruction List (SSE to AVX512)*
- [4] *Biblioteca de instrucciones para MMX y SSE*
- [5] *Pagina de wikipedia de SPEC CPU*
- [6] *Página donde se explica que es GCC y expone todo lo que lleva, estilo g++, .c, etc*
- [7] *Información respecto al paquete de pruebas .mcf, asociado a las tarjetas de red*
- [8] *Información respecto a OODBMS y como funciona la transcripción de datos a dispositivos de almacenamiento*