Programación II
Arboles Binarios(AB)

Definición

Un árbol consta de un conjunto finito de elementos, denominados **nodos**, y un conjunto finito de líneas dirigidas, denominadas **enlaces**, que conectan los nodos.

Hay tres tipos de nodos:

- **Nodo Raíz**: El único nodo que no tiene padre
- **Nodo Hoja**: Los nodos que no tiene hijos
- **Nodos Internos**: Los nodos que no son hojas

Intuitivamente el concepto de árbol implica una estructura en la que lo datos se organizan de modo que los elementos de información están organizados entre sí a través de ramas.

En la Figura1 se pueden ver los tres tipos de nodos:

- Raíz: A
- Internos: B,C,D
- Hojas: E,F,G,H,I

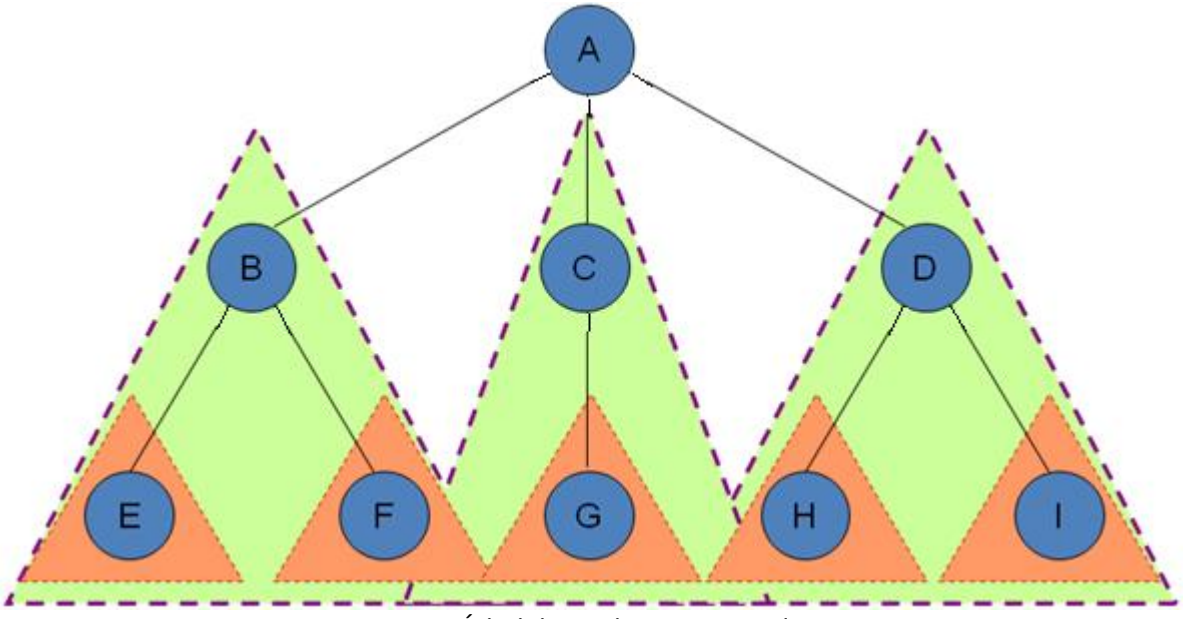

Figura1: Árbol de grado 3, con 9 nodos

La ***altura*** de un árbol se define como la distancia de la hoja más lejana a la raíz.

La altura del árbol de la Figura1 es 3.

Grado es el número máximo de hijos que tienen los nodos del árbol.
Así, en el ejemplo anterior el árbol es de grado 3

Árbol binario

Definición 1:
Es un árbol de grado 2.

Definición 2:
Un Árbol binario es aquel que:

- *es vacío, ó*
- *está formado por un nodo cuyos subárboles izquierdo y derecho son a su vez árboles binarios.*

El árbol de la Figura1 no es binario por ser de grado 3.

El de la Figura2 es binario por ser grado 2.

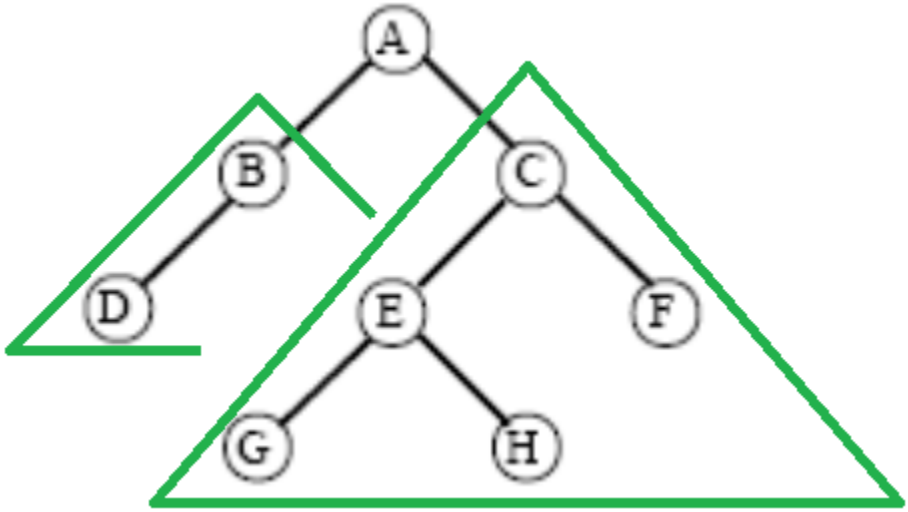

Figura2: Árbol binario y los primeros dos subarboles.

Todo árbol de mas de un nodos tiene al menos un subarbol.
En la Figura2 marcamos en verde los primeros subarboles:
El subarbol izquierdo(A y B)
El subarbol derecho(C,E,F,G,H)

Subarboles:
Por la naturaleza recursiva de los AB, todo subarbol sigue siendo árbol.
La altura de un subarbol es a lo sumo la altura del árbol padre - 1.

Existen algunos tipos especiales de árboles binarios en función de ciertas propiedades.
Por ejemplo

Árbol binario balanceado es aquel en el que en todos sus nodos se cumple la siguiente propiedad:La altura(subárbol_izquierdo) - altura(subárbol_derecho) es menor o igual a 1.

En la Figura 3 se muestran dos ejemplos:

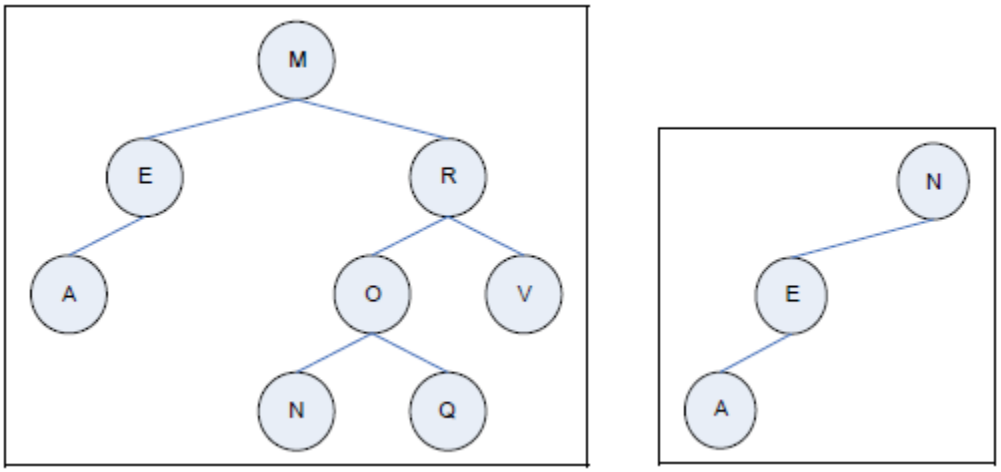

Figura3a: Árbol balanceado. Figura3b: Árbol desbalanceado

Árbol binario completo Sea A un árbol de altura h.
A es completo sii en el nivel h, A tiene todas las hojas posibles.

Todo árbol binario completo (Ver Figura4) tiene $2^{altura} - 1$ nodos

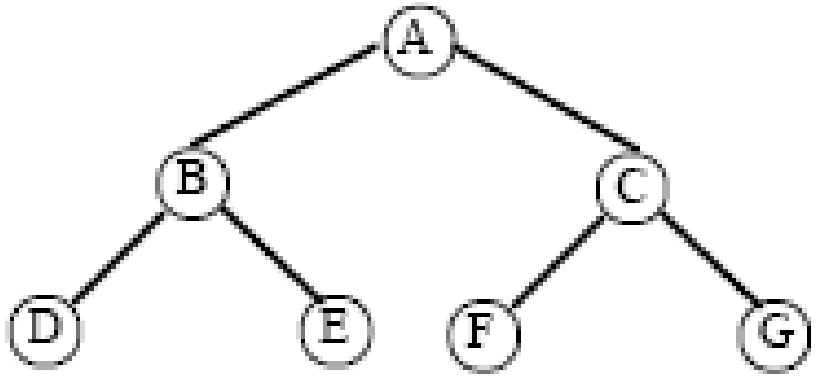

Figura4: Árbol binario completo de nivel 3, con 2^3-1 nodos

Invariante de representación

Se pueden llegar a todos los nodos desde la raíz
Cada nodo tiene un solo padre
No tiene ciclos

Implementación de Árbol binario

Un árbol solo necesita un puntero al nodo raíz.

Cada nodo contiene tres valores (Ver Figura5):

- La información
- Un puntero al nodo izquierdo
- Un puntero al nodo derecho

Figura5: Nodo

```
public class AB<T> {      //arbol binario
private Nodo<T> raiz;

public class Nodo<T>{     //clase interna
	private T info;
	private Nodo<T> izq;
	private Nodo<T> der;

	public Nodo(T info){
		this.info = info;
	}
	@Override
	public String toString(){
		return info.toString();
	}
}
```

```
public void agregar(T elem){
	Nodo<T> nuevo = new Nodo<T>(elem);
	if (raiz  == null) raiz = nuevo;
	else
		agregar(raiz,nuevo);
}
private void agregar(Nodo<T> padre, Nodo<T> nuevo){
	if (padre.izq == null) padre.izq = nuevo;
	else
		if (padre.der == null) padre.der = nuevo;
		else
		//Decisión de implementación: genera el arbol a derecha
			agregar(padre.der,nuevo);
}


public Nodo<T> buscar(T elem){
	return (raiz == null) ? null : buscar(raiz, elem);
}
private Nodo<T> buscar(Nodo<T> nodo, T elem){
	if (nodo.info.equals(elem)) return nodo;
	else{
		Nodo<T> izq = null;
		Nodo<T> der = null;
		if (nodo.izq != null)
			izq = buscar(nodo.izq, elem);
		if (nodo.der != null)
			der = buscar(nodo.der, elem);
		//Decisión de implementación: si esta en ambos lado, mostramos el
izquierdo primero
		if (izq != null) return izq;
		else return der;
	}
}

public int cantNodos(){
	return (raiz == null) ? 0 : cantNodos(raiz);
}
private int cantNodos(Nodo<T> nodo){
	int cantIzq = (nodo.izq  == null) ? 0 : cantNodos(nodo.izq) ;
	int cantDer = (nodo.der  == null) ? 0 : cantNodos(nodo.der) ;
	return 1 + cantIzq + cantDer;
}

public int altura(){
	return (raiz == null) ? 0 : altura(raiz);
}
private int altura(Nodo<T> nodo){
	int altIzq = (nodo.izq  == null) ? 0 : altura(nodo.izq) ;
	int altDer = (nodo.der  == null) ? 0 : altura(nodo.der) ;
	return 1 + Math.max(altIzq, altDer);
}
```

Casos base en árboles: dos enfoques

```
//version1:
public boolean balanceado(){
	return (raiz == null) ? true : balanceado(raiz);
}
private  boolean balanceado(Nodo<T> nodo){
	boolean ret= true;
	int altIzq = 0 ;
	int altDer = 0 ;
	if (nodo.izq != null){
		altIzq =altura(nodo.izq);
		ret = ret && balanceado(nodo.izq);
	}
	if (nodo.der != null){
		altDer =altura(nodo.der);
		ret = ret && balanceado(nodo.der);
	}
	ret=ret && Math.abs(altIzq - altDer) <= 1 ;
	return ret;
}

version2: a veces conviene hacer el caso base sobre el nodo
public boolean balanceado(){
	return  balanceado(raiz);
}
private  boolean balanceado(Nodo<T> nodo){
	if (nodo == null) return true;
	else{
		int altIzq = (nodo.izq == null) ? 0 : altura(nodo.izq) ;
		int altDer = (nodo.der == null) ? 0 : altura(nodo.der) ;
		return Math.abs(altIzq - altDer) <= 1  && balanceado(nodo.izq) &&
balanceado(nodo.der) ;
	}
}


@Override
public String toString(){
	return (raiz == null) ? "" : toString(raiz);
}
private String toString(Nodo<T> nodo){
	String ret = nodo.info.toString();
	if (nodo.izq != null) ret = ret + toString(nodo.izq);
	if (nodo.der != null) ret = ret + toString(nodo.der);
	return ret;
}
```

Ejercicio1:

Cual será el orden de complejidad de insertar un nodo para el peor caso sí:

a) El árbol no está completo

b) El árbol está completo

Ejercicio2:

Cual será el orden de complejidad de buscar un nodo para el peor caso sí:

a) El árbol no está completo

b) El árbol está completo