Recursividad

INF2240 – Estructura de datos
Escuela de Ingeniería Informática
Pontificia Universidad Católica de Valparaíso

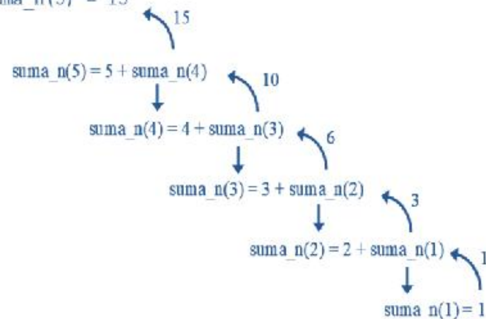
Recursividad

- Los programas generalmente se componen de una serie de funciones que se llaman una a otras siguiendo el principio de *modularidad*.
- Un subprograma recursivo se llama a sí mismo:
  - Recursión Directa, en donde el subprograma P contiene una sentencia en donde se invoca a P
  - Recursión Indirecta, en donde el subprograma P invoca a otro (Q) que a su vez puede llamar a otro.

109

Recursividad

— Ejemplo 1: La suma de 1 hasta n, es decir, F(n) = 1 + 2 + 3 + … + n




```
#include <stdio.h>
int suma_n(int n);

main() {
  int n = 5, sum = suma_n(n);
  printf("valor suma_n(%d) = %d",n,sum);
}

int suma_n(int n) {
  if (n==1) {
    printf("%d\n",1);
    return 1;
  } else {
    printf("%d+",n);
    return n+suma_n(n-1);
  }
}
```


Recursividad

— Un requisito fundamental de un algoritmo recursivo es que no termine llamándose a sí mismo indefinidamente. Para ello debe existir un condición de salida.

— Para una terminación normal se requiere de:

- Un test (if) que evalúe la condición de salida
- La llamada recursiva
- La condición de salida

Recursividad

- Cualquier problema solucionado de manera recursiva se puede plantear de manera iterativa.
- Si una solución a un problema se puede expresar de manera iterativa o recursiva con igual facilidad, entonces es preferible la solución iterativa pues:
  - Se ejecuta más rápidamente, al no existir llamada adicionales a funciones.
  - Utiliza menos memoria, pues se ahorra la pila que almacena las llamadas necesarias.

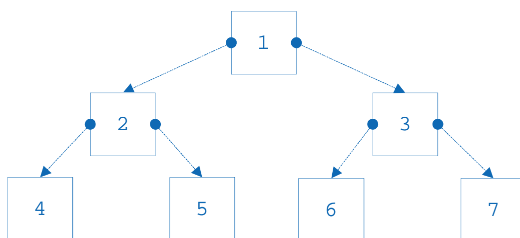
112

Árboles binarios

INF2240 – Estructura de datos
Escuela de Ingeniería Informática
Pontificia Universidad Católica de Valparaíso

Introducción

- Un árbol *A* es un conjunto finito de uno o más nodos:
  - Existe un nodo especial denominado RAIZ(V1) del árbol.
  - Los nodos restantes (V1 ,V2 ,...Vn) se dividen en m>=0 conjuntos disjuntos denominados A1, A2, A3,...Am, cada uno de los cuales es un árbol. Estos árboles se llaman subárboles de la RAIZ.



Definiciones generales

- Raíz: Nodo que no tiene antecesores
- Nodo: Vértices o elementos del árbol
- Nodo Terminal u hoja: Vértices o elementos del árbol que no contienen subárboles.
- Hermanos: Nodos de un mismo padre.
- Nodos Interiores: Nodos que no son hoja ni raíz.
- Bosque: Colección de dos o más árboles.
- Arista: Enlace entre dos nodos consecutivos.

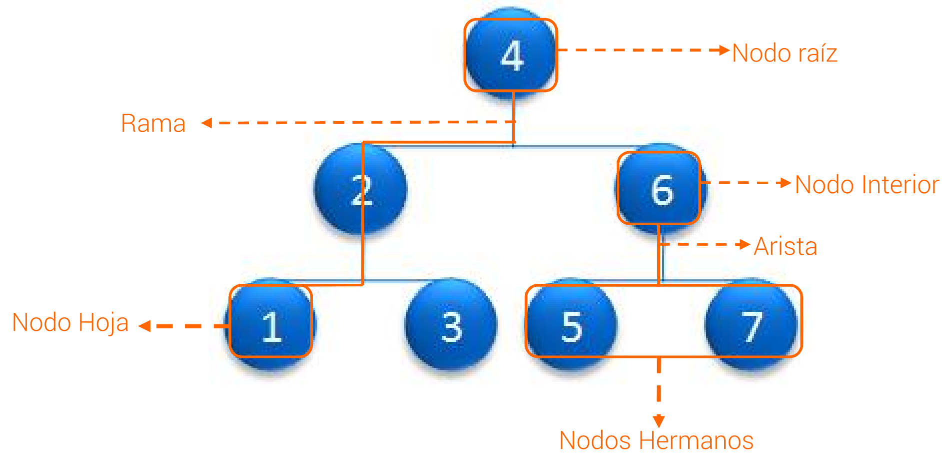
115

Definiciones generales

- Camino: Secuencia de aristas consecutivas.
- Rama: Camino que termina en hoja.
- Nivel: Longitud que se determina por la longitud del camino desde la raíz al nodo específico.
- Altura o profundidad: el número máximo de nodos de una rama. Equivale al nivel más alto de los nodos más uno.
- Peso: es el número de nodos terminales.
- Grado: el número de hijos que salen de un nodo.

116

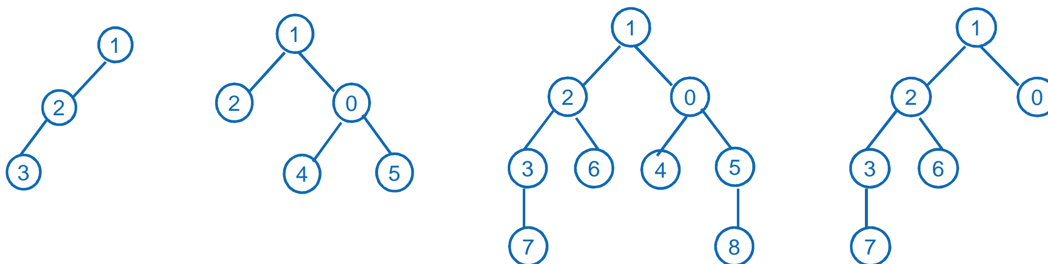
Definiciones generales



117

Árboles binarios

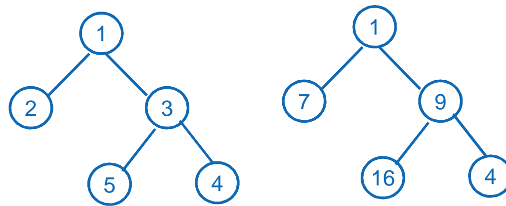
- Un **Árbol Binario** es un conjunto finito de cero o más nodos tales que:
  - Existe un nodo denominado *raíz del árbol*.
  - Cada nodo tiene 0, 1 o 2 subárboles, conocidos como subárbol izquierdo y subárbol derecho.



118

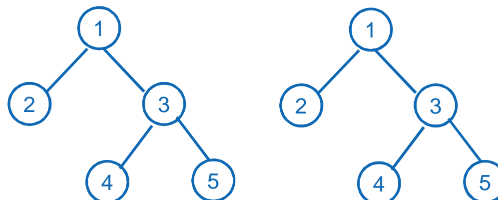
Árboles binarios

– Se dice que dos árboles binarios son similares si tienen la misma estructura.



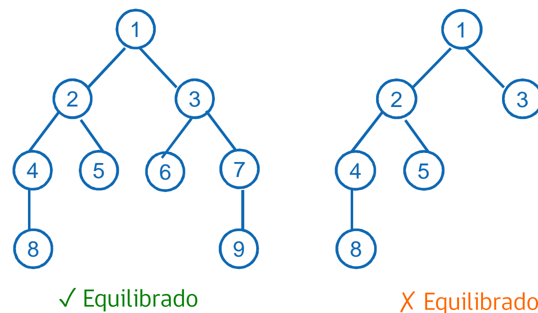
Árboles binarios

– Se dice que dos árboles binarios son equivalentes si tienen la misma estructura y además la misma información.



Árboles binarios

— Un árbol binario es equilibrado si las alturas de los dos subárboles de cada nodo del árbol se diferencian en una unidad como máximo.



121

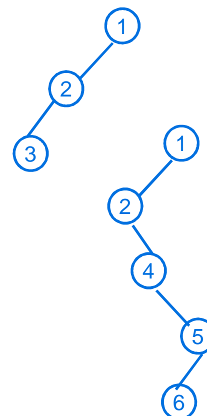
Árboles binarios

— Un árbol binario es degenerado si todos sus nodos excepto el último tienen sólo un subárbol.

— Un árbol binario es completo si todos sus nodos, excepto las hojas, tienen exactamente dos subárboles.

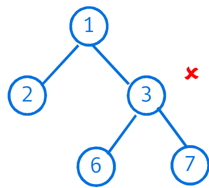
— Un árbol binario es lleno si todas sus hojas están al mismo nivel y todos sus nodos interiores tienen cada uno 2 hijos.

— Un árbol binario de altura h puede tener como máximo 2h-1 nodos.



122

Árboles binarios

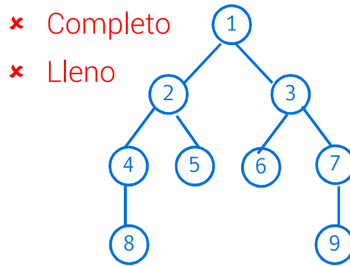


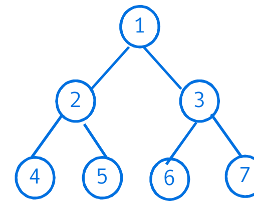
Completo

× Lleno

× Completo

× Lleno



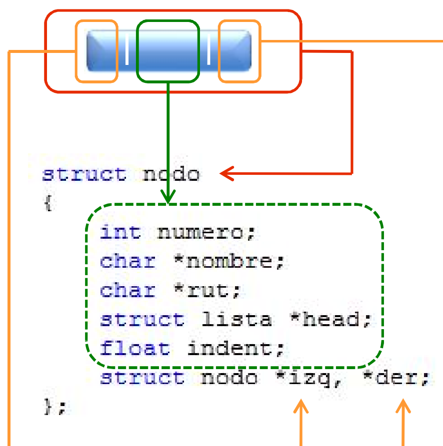


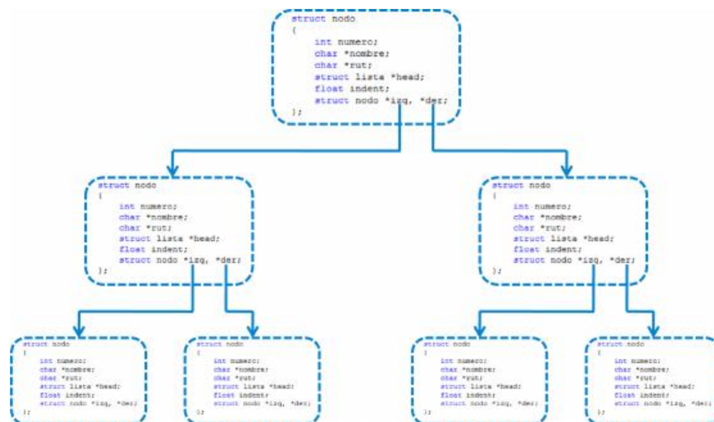
Completo

Lleno

123

Representación de árboles binarios





124

Recorrido de árboles binarios

- Se denomina recorrido de un árbol al proceso que permite acceder una sola vez a cada uno de los nodos del árbol y el conjunto completo de nodos se examina.
- Recorrido pre-orden
  - Visitar nodo raíz
  - Recorrer el subárbol izquierdo en pre-orden
  - Recorrer el subárbol derecho en pre-orden

Recorrido de árboles binarios

- Recorrido in-orden
  - Recorrer el subárbol izquierdo en in-orden
  - Visitar nodo raíz
  - Recorrer el subárbol derecho en in-orden
- Recorrido post-orden
  - Recorrer el subárbol izquierdo en post-orden
  - Recorrer el subárbol derecho en post-orden
  - Visitar nodo raíz

Recorrido de árboles binarios

— Uso de pilas

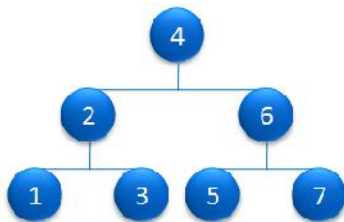
```
struct pila
{
    struct nodo *valor;
    struct pila *sgte;
};
```

```
void push(struct pila **cima, struct nodo *valor)
{
    struct pila *nuevo;
    nuevo = (struct pila*)malloc(sizeof(struct pila));
    nuevo->valor = valor;
    nuevo->sgte = *cima;
    *cima = nuevo;
}
```

127

Recorrido de árboles binarios: pre-orden

```
struct nodo
{
    int elem;
    struct nodo *izq, *der;
};
```

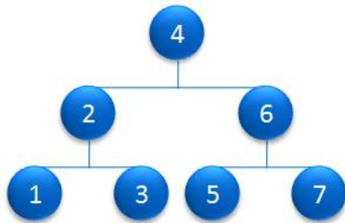



```
void preorden(struct nodo *ptr)
{
    struct pila *cima=NULL;
    push(&cima,NULL);
    while(ptr)
    {
        printf("%d-", ptr->elem);
        if(ptr->der)
            push(&cima,ptr->der);
        if(ptr->izq)
            ptr=ptr->izq;
        else
            ptr=pop(&cima);
    }
}
```

128

Recorrido de árboles binarios: in-orden

```
struct nodo
{
    int elem;
    struct nodo *izq, *der;
};
```

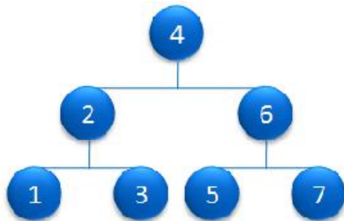



```
void inorden(struct nodo *ptr)
{
    struct pila *cima=NULL;
    push(&cima,NULL);
    while(cima)
    {
        while(ptr)
        {
            push(&cima,ptr);
            ptr=ptr->izq;
        }
        if(cima)
        {
            ptr=pop(&cima);
            if(cima)
                printf("%d-", ptr->elem);
        }
        ptr=ptr->der;
    }
}
```

129

Recorrido de árboles binarios: post-orden

```
struct nodo
{
    int elem;
    struct nodo *izq, *der;
};
```

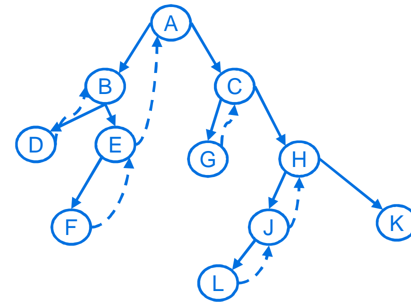



```
void postorden(struct nodo *ptr)
{
    struct pila *cima=NULL;
    push(&cima,NULL);
    while(cima)
    {
        while(ptr)
        {
            push(&cima,ptr);
            if(ptr->der)
            {
                push(&cima,ptr->der);
                push(&cima,NULL);
            }
            ptr=ptr->izq;
        }
        ptr=pop(&cima);
        while(ptr)
        {
            printf("%d-", ptr->elem);
            ptr=pop(&cima);
        }
        if(!ptr)
            ptr=pop(&cima);
    }
}
```

130

Árboles atados

- Se dice que un árbol binario está atado si cada nodo del árbol posee un campo adicional de tipo puntero al nodo siguiente en un determinado recorrido.
- Existen distintas formas de atar un árbol binario, pero cada una corresponderá al recorrido en cuestión.
- Ejemplo: Atadura Unidireccional



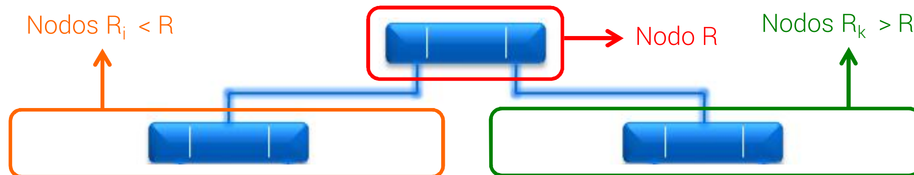
131

Árboles binarios de búsqueda (ABB)

- Supongamos que se tiene n claves distintas en orden ascendente *K1, K2, K3,…,Kn* y que se tiene un árbol binario *T* de *n* nodos.
- Se define *Ni* como el iésimo nodo visitado si se recorre *T* en In-orden. Luego si se almacena la clave *Ki* en el nodo *Ni*, el árbol resultante tiene la siguiente propiedad:
  - Para cada nodo Ni con clave Ki,
  - todas las claves en los nodos del subárbol izquierdo son menores que Ki y
  - todas las claves en los nodos del subárbol derecho son mayores que Ki.

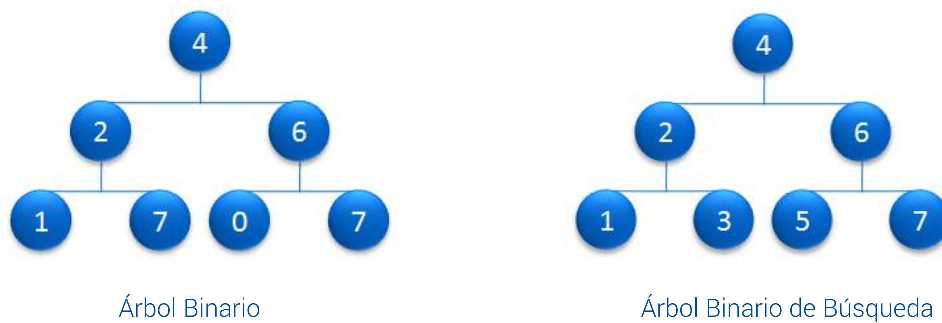
132

Árboles binarios de búsqueda (ABB)



Árboles binarios de búsqueda (ABB)

- Así, se puede observar la diferencia entre un AB y un ABB:



Árbol Binario

Árbol Binario de Búsqueda

Árboles binarios de búsqueda: búsqueda

```
struct nodo* buscar(struct nodo *arbol, struct nodo **ant, int valor)
{
    int enc=0;
    while(!enc && (arbol))
    {
        if(arbol->elem==valor)
            enc=1;
        else
        {
            *ant=arbol;
            if(valor<arbol->elem)
                arbol=arbol->izq;
            else
                arbol=arbol->der;
        }
    }
    return arbol;
}
```


Árboles binarios de búsqueda: insertar

```
void insertar(struct nodo **arbol, int valor)
{
    struct nodo *nuevo=NULL,*ant=NULL, *ptr=NULL;

    /* Buscar posicion de insercion */
    ptr=buscar(*arbol, &ant, valor);
    if(!ptr)
    {
        /*crear nodo nuevo*/
        nuevo=(struct nodo*)malloc(sizeof(struct nodo));
        nuevo->elem = valor;
        nuevo->izq=nuevo->der=NULL;
        if(!ant)
            *arbol=nuevo;
        else
        {
            if(valor<ant->elem)
                ant->izq=nuevo;
            else
                ant->der=nuevo;
        }
    }
    else
        printf("nodo ya esta en el arbol");
}
```

Árboles binarios de búsqueda: borrar

```
int eshoja(struct nodo *ptr)
{
    if(!ptr->der&&!ptr->izq)
        return 1;
    return 0;
}
```


Árboles binarios de búsqueda: borrar

```
void borrar(struct nodo **abb, int valor)
{
    struct nodo *padre=NULL,*actual=NULL,*nodo=NULL;
    int aux;
    actual=buscar(*abb, &padre, valor);
    while(actual)
    {
        if(eshoja(actual))
        {
            if(padre)
            {
                if(padre->der == actual)
                    padre->der = NULL;
                else
                    padre->izq = NULL;
            }
            free(actual);
            actual=NULL;
            return;
        }
        else
        {
            padre = actual;
            if(actual->der)
            {
```

```
/*buscar nodo mas a la izquierda del subarbol derecho*/
    nodo = actual->der;
    while(nodo->izq)
    {
        padre = nodo;
        nodo = nodo->izq;
    }
}
else
{
    /*buscar nodo mas a la derecha del subarbol izquierdo */
    nodo = actual->izq;
    while(nodo->der)
    {
        padre = nodo;
        nodo = nodo->der;
    }
}
/* Intercambio */
aux = actual->elem;
actual->elem = nodo->elem;
nodo->elem = aux;
actual = nodo;
}
```

ABB: recorridos recursivos

- Preorden
  - (1) Nodo raíz → (2) Subárbol izquierdo → (3) Subárbol derecho
- InOrden
  - (1) Subárbol izquierdo → (2) Nodo raíz → (3) Subárbol derecho
- PostOrden
  - (1) Subárbol izquierdo → (2) Subárbol derecho → (3) Nodo raíz

139

ABB: recorridos recursivos

```c
void preorden (struct nodo *arbol)
{
    if (arbol)
    {
        printf("%d\n", arbol->elem);
        preorden(arbol->izq);
        preorden(arbol->der);
    }
    return;
}
```

```c
void inorden (struct nodo *arbol)
{
    if (arbol)
    {
        inorden(arbol->izq);
        printf("%d\n", arbol->elem);
        inorden(arbol->der);
    }
    return;
}
```

```c
void postorden (struct nodo *arbol)
{
    if (arbol)
    {
        postorden(arbol->izq);
        postorden(arbol->der);
        printf("%d\n", arbol->elem);
    }
    return;
}
```

140

ABB: búsqueda recursiva

```
struct nodo *buscar (struct nodo *ptr, int num)
{
    if (!ptr || ptr->elem==num)
        return ptr;
    else
    {
        if (ptr->elem>num)
            return(buscar(ptr->izq,num));
        else
            return(buscar(ptr->der,num));
    }
}
```


ABB: inserción recursiva

```
void insertar (struct nodo **abb, int num)
{
    if (!(*abb))
    {
        (*abb)=(struct nodo *) malloc(sizeof(struct nodo));
        (*abb)->elem=num;
        (*abb)->izq =(*abb)->der=NULL;
    }
    else
    {
        if ((*abb)->elem != num)
        {
            if ((*abb)->elem>num)
                insertar(&((*abb)->izq), num);
            else
                insertar(&((*abb)->der), num);
        }
    }
}
```

ABB: eliminación recursiva

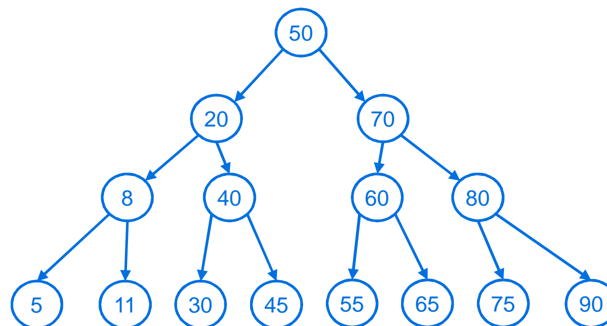
```
void borrar (struct nodo **arbol, int num)
{
    struct nodo *aux=NULL;

    if (!(*arbol))
        return;
    if ((*arbol)->elem < num)
        borrar(&(*arbol)->der, num);
    else
    {
        if ((*arbol)->elem > num)
            borrar(&(*arbol)->izq, num);
        else
        {
            if ((*arbol)->elem == num)
            {
                aux = *arbol;
                if (!((*arbol)->izq))
                    *arbol = (*arbol)->der;
                else
                {
                    if (!((*arbol)->der))
                    *arbol = (*arbol)->izq;
                    else
                        reemplazar(&(*arbol)->izq, &aux);
                }
                    free(aux);
            }
        }
    }
}
```

```
void reemplazar(struct nodo **abb, struct nodo **aux)
{
    if (!((*abb)->der))
    {
        (*aux)->elem = (*abb)->elem;
        *aux = *abb;
        *abb = (*abb)->izq;
    }
    else
        reemplazar(&(*abb)->der, &(*aux));
}
```

143

Actividad: árboles binarios de búsqueda

— Usando como referencia:



144

Actividad: árboles binarios de búsqueda

- Rutear:
  - Función borrar iterativa
  - Funciones de recorrido y eliminación recursivas
- Investigar:
  - Desarrolle una función recursiva para eliminar un elementos del árbol. Esta función puede ser con recursividad directa o indirecta considerando:
    - No puede usar la función recursiva entregada por el profesor o ayudante.
    - La actividad es individual y en la prueba/proyecto/controles podrá ser solicitado su uso.

145