

ELO320 Estructuras de Datos y Algoritmos

Arboles Binarios

Tomás Arredondo Vidal

Este material está basado en:

- ❑ Robert Sedgewick, "Algorithms in C", (third edition), Addison-Wesley, 2001
- ❑ Thomas Cormen et al, "Algorithms", (eighth edition), MIT Press, 1992.
- ❑ material del curso ELO320 del Prof. Leopoldo Silva
- ❑ material en el sitio <http://es.wikipedia.org>

8-Árboles Binarios

8.1 Definiciones y tipos de datos

8.2 Cálculos de complejidad

8.3 Operaciones básicas y de recorrido

8.4 Operaciones de consulta

8.5 Operaciones de modificación

Definiciones

- ❑ Un árbol es una estructura de datos con nodos enlazados en forma jerárquica y orientada.
 - Es una estructura ordenada en que los hijos de un nodo generalmente tienen un valor menor que este y están ordenados de izquierda a derecha.
- ❑ La raíz es el punto de entrada a la estructura.
- ❑ La raíz puede tener cero o más nodos descendientes desde ella.
 - El conjunto de estos nodos forman subárboles de la raíz.
 - La raíz es el ancestro de estos subárboles.
 - Los nodos sin descendientes se llaman hojas.
 - Los nodos internos son todos los nodos menos las hojas.

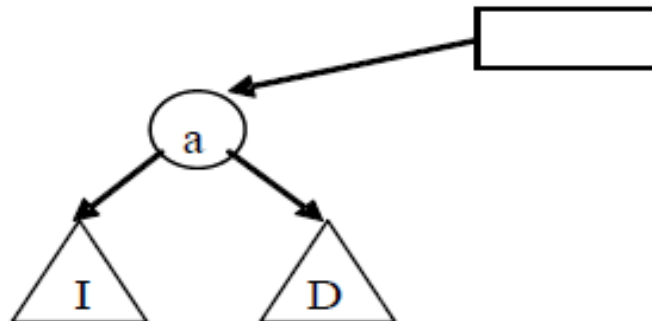
Definiciones II

- ❑ Una **trayectoria** del nodo n_i al nodo n_k , es una secuencia de nodos desde n_i hasta n_k , tal que n_i es el padre de n_{i+1} .
- ❑ Existe un solo enlace (link) entre un padre y cada uno de sus hijos.
- ❑ El **largo** de una trayectoria es el número de enlaces en la trayectoria.
 - Una trayectoria de k nodos tiene largo $k-1$.
- ❑ La **altura** de un nodo es el largo de la trayectoria más larga de ese nodo a una hoja.
- ❑ La **profundidad** de un nodo es el largo de la trayectoria desde la raíz a ese nodo.
 - La profundidad del árbol es la profundidad de la hoja mas profunda.
 - Nodos a una misma profundidad están al mismo **nivel**.

Definiciones III

□ Árboles binarios

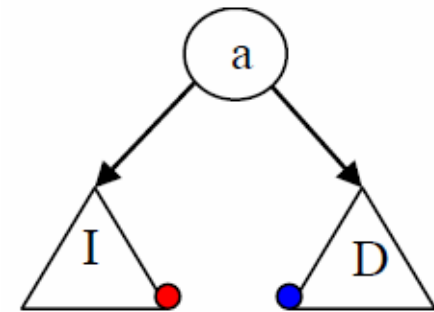
- Cada nodo puede tener un hijo izquierdo y/o un hijo derecho.
- Un árbol binario está formado por un nodo raíz, un subárbol izquierdo I y uno derecho D .
 - Donde I y D son árboles binarios. Los subárboles se suelen representar gráficamente como triángulos.



Definiciones IV

□ Árboles binarios de búsqueda

- Las claves de los nodos del subárbol **izquierdo** deben ser menores que la clave de la raíz.
- Las claves de los nodos del subárbol **derecho** deben ser mayores que la clave de la raíz.
- Esta definición no acepta elementos con claves duplicadas.
- Se indican el descendiente del subárbol izquierdo con mayor valor de clave y el descendiente del subárbol derecho con menor valor; los cuales son **el antecesor** y **sucesor** de la raíz.



Tipos de datos

- ❑ Para un árbol binario hay que usar dos punteros.
 - Se da un ejemplo con una clave entera, no se muestra espacio para los otros datos que puede estar asociada al nodo.
 - En la implementación de algunas operaciones conviene disponer de un puntero al padre del nodo (tampoco se muestra en este ejemplo).

```
❑ struct node
{
    int clave;
    struct node *left;
    struct node *right;
};
```

```
typedef node * pnode;
```

8-Árboles Binarios

8.1 Definiciones y tipos de datos

8.2 Cálculos de complejidad

8.3 Operaciones básicas y de recorrido

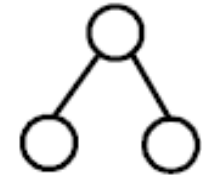
8.4 Operaciones de consulta

8.5 Operaciones de modificación

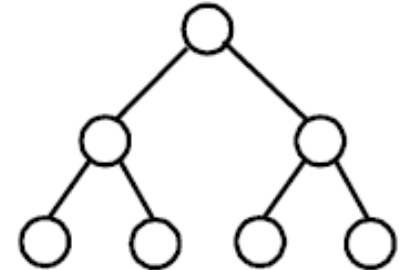
Complejidad: árboles completos

- Deduciremos, de manera inductiva la altura de las hojas en función del número de nodos.
- El caso más simple de un árbol **completo** tiene tres nodos, un nivel y altura (A) de dos.
- Hemos modificado levemente la definición de **altura**, como el **número de nodos que deben ser revisados desde la raíz a las hojas**, ya que la **complejidad** de los algoritmos dependerá de esta variable.
 - A medida que se avanza en la trayectoria se descarta $T(n/2)$ nodos en cada avance.
- Un árbol perfectamente balanceado que tiene n nodos internos tiene n+1 hojas.

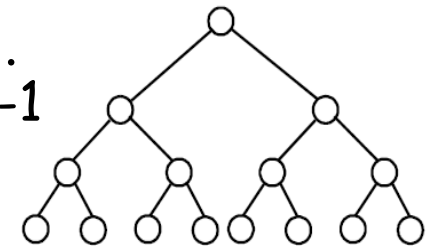
- Árbol de nivel 1.
Nodos = $3 = 2^2 - 1$
Altura = 2



- Árbol de nivel 2.
Nodos = $7 = 2^3 - 1$
Altura = 3



- Árbol de nivel 3.
Nodos = $15 = 2^4 - 1$
Altura = 4



- Árbol de n nodos:
 $n = 2^{A-1}$

$$A = \log_2(n+1) = O(\log n)$$

Complejidad: árboles con un nivel de desbalance

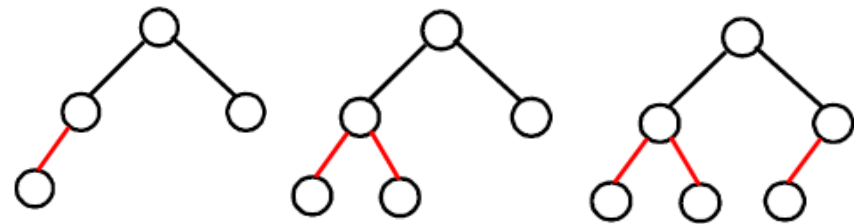
- Se ilustran los tres casos de árboles, de nivel dos, con un nivel de desbalance, para $n = 4, 5$ y 6 .
- En general para n nodos se tiene:
$$2^{A-1} \leq n \leq 2^A - 2$$

$A \leq (1 + \log_2(n))$ para la primera desigualdad y $A \geq \log_2(n+2)$ para la segunda.

- Cual es la complejidad de A ?
- Se pueden encontrar constantes (i.e. c_1, c_2) que acoten, por arriba y por abajo a ambas funciones para $n > 10,079$:

$$\begin{aligned} 1 * \log_2 n &\leq \log_2(n+2) \leq A \leq 1 + \log_2 n \\ &\leq 1.3 * \log_2 n \\ A &= \Theta(\log n) \end{aligned}$$

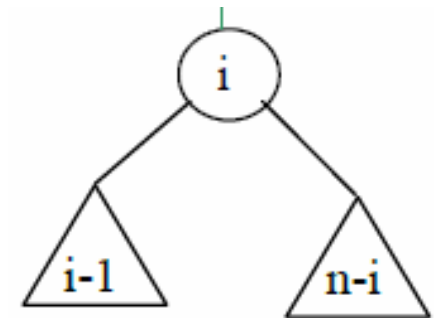
- Árboles de nivel 2.
Nodos de 4 a 6
De 2^{3-1} hasta $2^3 - 2$
Altura = 3



- Árboles de nivel 3.
Nodos de 8 a 14
De 2^{4-1} hasta $2^4 - 2$
Altura = 4
- Árbol de nivel m .
Nodos: $2^{A-1} \leq n \leq 2^A - 2$
Altura = A

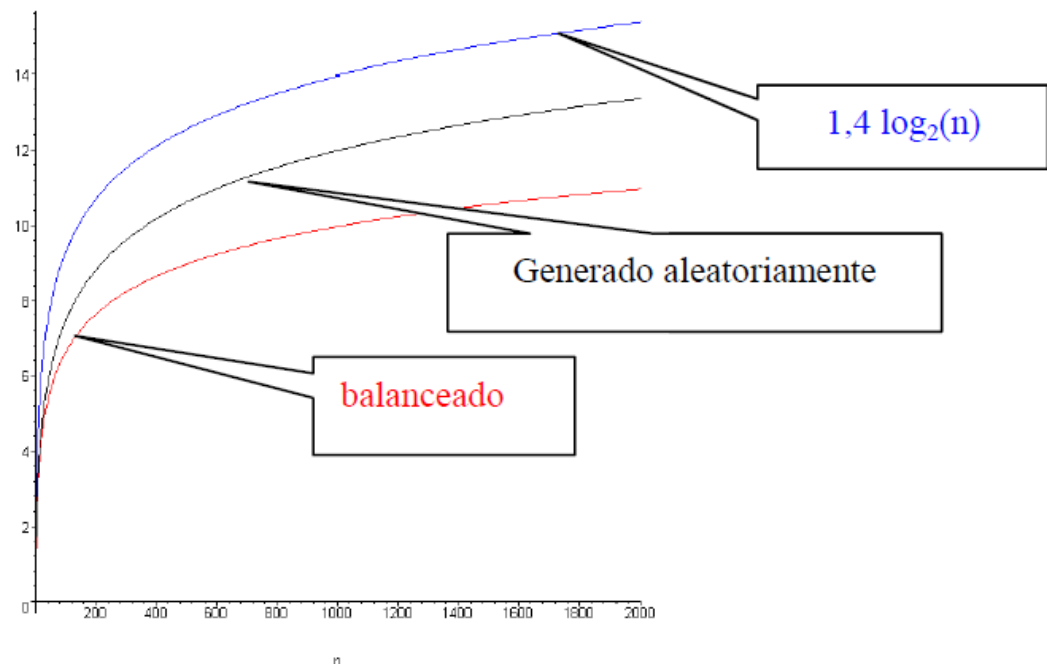
Complejidad: árboles contruidos en forma aleatoria

- ❑ Para n nodos, con claves: $1, 2, 3, \dots, n$, se pueden construir $n!$ árboles. Ya que existen $n!$ permutaciones de n elementos.
- ❑ El orden de los elementos de la permutación, es el orden en que se ingresan las claves a partir de un árbol vacío.
- ❑ Lo que se desea conocer es la altura A_n , definida como la altura promedio de las búsquedas de las n claves y promediadas sobre los $n!$ árboles que se generan a partir de las $n!$ permutaciones que se pueden generar con la n claves diferentes.
- ❑ Si el orden de llegada de las claves que se insertan al árbol se genera en forma aleatoria, la probabilidad de que la primera clave, que es la raíz, tenga valor i es $1/n$.
- ❑ Esto dado que todas las claves son igualmente probables.
- ❑ El subárbol izquierdo contiene $(i-1)$ nodos; por lo tanto el subárbol derecho contiene $(n-i)$ nodos.
- ❑ Para este tipo de árbol en promedio el largo de cualquier trayectoria es: $A_n \approx 2\ln(n) = O(\log 2)$



Complejidad: árboles construidos en forma aleatoria (cont)

- ❑ La gráfica muestra que para árboles de tipo 1000 nodos, deben recorrerse cerca de nueve nodos desde la raíz hasta las hojas (peor caso), si está balanceado.
- ❑ El largo promedio de los recorridos es 12 en un árbol generado aleatoriamente, y 1000 en peor caso.
- ❑ Contrastando con un árbol balanceado para n grande:
 $An/A \approx 2\ln(n)/2\log_2(n) \approx 1,3$
- ❑ Para n grande en promedio el alargue es entre un 20 - 39%



8-Árboles Binarios

8.1 Definiciones y tipos de datos

8.2 Cálculos de complejidad

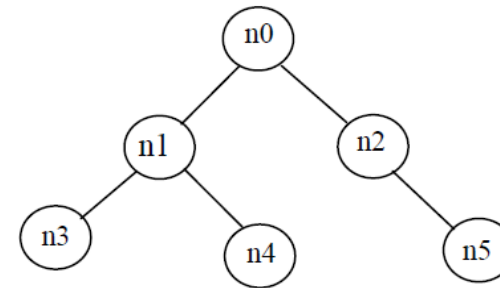
8.3 Operaciones básicas y de recorrido

8.4 Operaciones de consulta

8.5 Operaciones de modificación

Recorridos en árboles

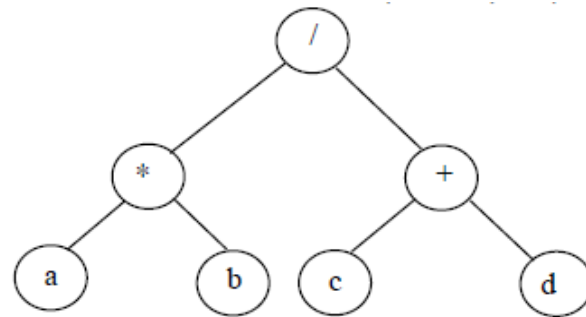
- Existen tres modos de recorrido, con las siguientes definiciones recursivas:
 - En orden: Se visita el subárbol izquierdo en orden; Se visita la raíz; Se visita el subárbol derecho en orden.
 - Pre orden: Se visita la raíz; Se visita el subárbol izquierdo en preorden; Se visita el subárbol derecho en preorden.
 - Post orden: Se visita el subárbol izquierdo en postorden; Se visita el subárbol derecho en postorden; Se visita la raíz.
- Ejemplo: Como se recorrería el árbol formado con el siguiente conjunto de claves usando los tres modos?
 $\{ n0, n1, n2, n3, n4, n5 \}$



- En orden:
 $\{n1, n3, n4\}, n0, \{n2, n5\}$
 $n3, n1, n4, n0, \{n2, n5\}$
 $n3, n1, n4, n0, n2, n5$
- Pre orden:
 $n0, \{n1, n3, n4\}, \{n2, n5\}$
 $n0, n1, n3, n4, \{n2, n5\}$
 $n0, n1, n3, n4, n2, n5$
- Post orden:
 $\{n1, n3, n4\}, \{n2, n5\}, n0$
 $n3, n4, n1, \{n2, n5\}, n0$
 $n3, n4, n1, n5, n2, n0$

Recorridos en árboles II

- Notación in situ, corresponde a recorrido en orden: I, nO, D
- Como seria el arbol? $(a * b) / (c + d)$



- Como seria un recorrido en post orden (RPN)?

$a b * c d + /$

- Tarea, encontrar expresión in situ para la polaca inversa:

$a b c / + d e f * - *$

Operaciones básicas: crear nodo

- ❑ Crear árbol vacío:

```
pnode arbol=NULL;
```

- ❑ Crea nodo inicializado con un valor:

```
pnode CreaNodo(int valor)
{ pnode pi=NULL;
  if ( (pi= (pnode) malloc(sizeof(nodo))) ==NULL)
    return (0);
  else
  {
    pi->clave=valor;
    pi->left=NULL;
    pi->right=NULL;
  }
  return(pi);
}
```


Operaciones básicas: recorrer en orden

```
void RecorraEnOrden(pnodo p)
{  if (p!= NULL) //si no llegó a hojas y no es árbol vacío.
    {
        RecorraEnOrden(p->left);    // primero recorre el
                                     // subárbol izquierdo.
        printf ("%d \n", p->clave);  // imprime el nodo
        RecorraEnOrden(p->right);    // recorre subarbol der.
    }
}
```

- Si se tiene un árbol de n nodos, y si se asume arbitrariamente que el subárbol izquierdo tiene k nodos, se puede plantear que la complejidad temporal del recorrido es: $T(n) = T(k) + \Theta(1) + T(n-k-1)$

Operaciones básicas: recorrer II

- ❑ Para simplificar el cálculo podemos asumir un árbol balanceado:
$$T(n) = T(n/2) + \Theta(1) + T(n/2 - 1)$$
- ❑ Y para grandes valores de n , podemos simplificar aún más:
$$T(n) = 2 * T(n/2)$$
 que tiene por solución: $T(n) = n = \Theta(n)$
- ❑ Otro cálculo es considerar el peor caso para el subárbol derecho:
$$T(n) = T(1) + \Theta(1) + T(n - 2)$$
- ❑ La que se puede estudiar como $T(n) = T(n-2) + 2$ asumiendo $\Theta(1) = T(1) = 1$, $T(2) = 1$ que tiene por solución $T(n) = n - (1/2)(1+(-1)^n)$.
- ❑ El segundo término toma valor cero para n par, y menos uno para n impar.
- ❑ Puede despreciarse para grandes valores de n , resultando:
$$T(n) = \Theta(n)$$

Operaciones básicas: recorrer mostrando nivel

- Recorrer en orden mostrando nivel:

```
void inorder(pnodo t, int nivel)
{
    if (t != NULL)
    {
        inorder(t->left, nivel+1);
        printf ("%d %d \n", t->clave, nivel);
        inorder(t->right, nivel +1);
    }
}
```

- Ejemplo de uso:

```
inorder(arbol, 0); //Imprime considerando raíz de nivel 0.
```

- Mostrar en post-orden y pre-orden son análogos a como se implemento RecorreEnOrden() e inorder().

8-Árboles Binarios

8.1 Definiciones y tipos de datos

8.2 Cálculos de complejidad

8.3 Operaciones básicas y de recorrido

8.4 Operaciones de consulta

8.5 Operaciones de modificación

Buscar mínimo

```
pnode BuscarMinimoIterativo(pnode t) {
    while ( t != NULL)
    {
        if ( t->left == NULL )
            return (t); //apunta al mínimo.
        else
            t=t->left; //desciende por la izquierda
    }
    return (t); /* NULL si árbol vacío*/
}

pnode BuscaMinimoRec(pnode t) {
    if (t == NULL)
        return(NULL); //si árbol vacío retorna NULL
    else // Si no es vacío
        if (t->left == NULL)
            return(t); // Si no tiene hijo izquierdo: lo encontró.
        else
            return( BuscaMinimoRec (t->left) ); //busca en subárbol izquierdo.
}
```

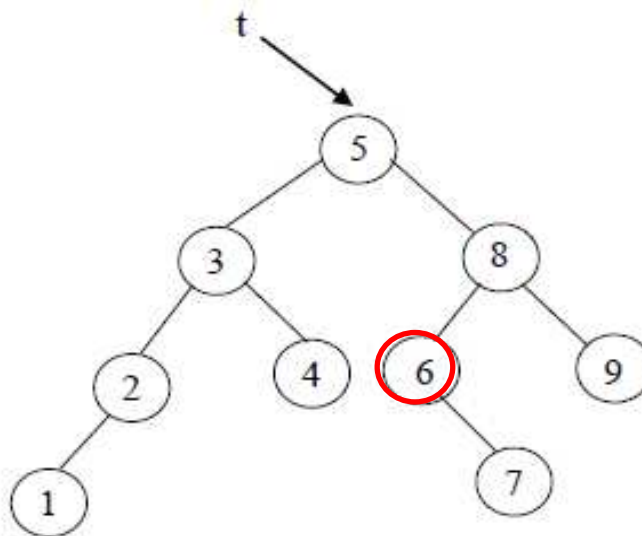
Buscar máximo

```
pnode BuscarMaximoIterativo(pnode t) {  
    while ( t != NULL) {  
        if ( t->right == NULL )  
            return (t); //apunta al máximo.  
        else  
            t=t->right; //desciende  
    }  
    return (t); /* NULL Si árbol vacío*/  
}
```

```
pnode BuscaMaximoRec(pnode t) {  
    if (t == NULL)  
        return(NULL); //si árbol vacío retorna NULL  
    if (t->right == NULL)  
        return(t ); // Si no tiene hijo derecho: lo encontró.  
    return( BuscaMaximoRec (t->right) ); //sigue buscando en subárbol der.  
}
```

Nodo descendiente del subárbol derecho con menor valor de clave

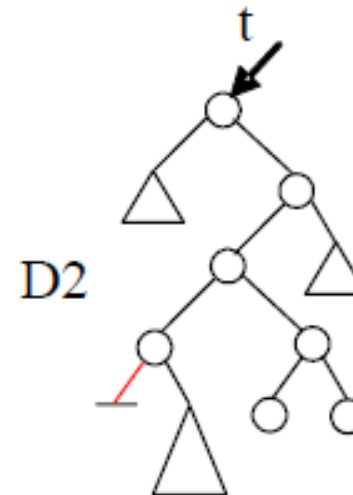
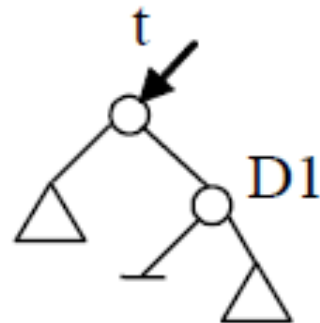
```
pnode MenorDescendienteSD(pnode t) {  
    if (t == NULL)  
        return(NULL); //si árbol vacío retorna NULL  
    if (t->right == NULL)  
        return(NULL); // Si no tiene hijo derecho no hay sucesor.  
    return( BuscaMinimo (t->right) ); //sigue buscando en subárbol der.  
}
```



Nodo descendiente del subárbol derecho con menor valor de clave II

□ Para el diseño iterativo, deben estudiarse dos casos:

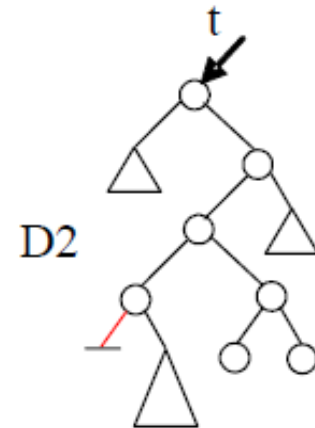
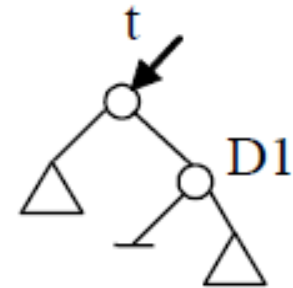
- El caso D1, un nodo sin hijo izquierdo, indica que se encontró el mínimo.
- El caso D2, debe descenderse por el subárbol derecho de t , por la izquierda, mientras se tengan hijos por la izquierda.



Nodo descendiente del subárbol derecho con menor valor de clave III

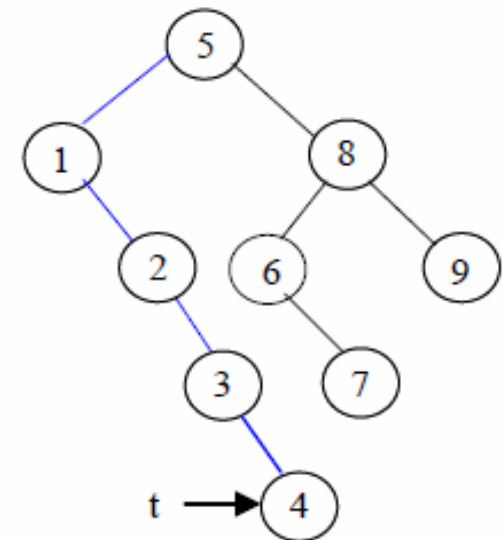
- Menor descendiente de subárbol derecho

```
pnodo MenorDescendienteIterativoSD(pnodo t) {  
    pnodo p;  
    if (t == NULL)  
        return(NULL); // si árbol vacío retorna NULL  
    if (t->right == NULL)  
        return(NULL); // sin hijo derecho no hay sucesor.  
    else  
        p = t->right;  
    while (p->left != NULL)  
    { // mientras no tenga hijo izq descender por izq.  
        p = p->left;  
    } // al terminar while p apunta al menor descendiente  
    return (p); // retorna el menor  
}
```



Nodo sucesor

- ❑ Dado un nodo encontrar su sucesor no es el mismo problema anterior, ya que el nodo podría ser una hoja o un nodo sin subárbol derecho.
- ❑ Por ejemplo en la Figura, el sucesor del nodo con clave 4 es el nodo con clave 5. El sucesor del nodo 2 es el nodo con valor 3.
- ❑ Se requiere disponer de un **puntero al padre** del nodo, para que la operación sea de costo logarítmico, en promedio.
- ❑ Si un nodo tiene subárbol derecho, el sucesor de ese nodo es el ubicado más a la izquierda en ese subárbol; si no tiene subárbol derecho, es el menor ancestro (que está sobre el nodo en la trayectoria hacia la raíz) que tiene a ese nodo (e.g. 4) en su subárbol izquierdo.
- ❑ Como en peor caso debe ascenderse un trayectoria del nodo hacia la raíz, el costo será $\mathcal{O}(a)$, donde a es la altura del árbol.



Nodo sucesor

□ Algoritmo Sucesor:

Si el árbol no es vacío:

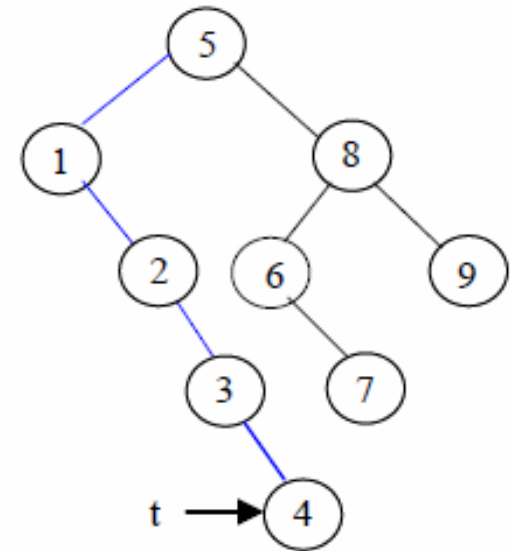
Si no tiene subárbol derecho:

Mientras exista el padre y **éste apunte al nodo dado por la derecha se asciende:**

Hasta encontrar el primer padre por la izquierda.

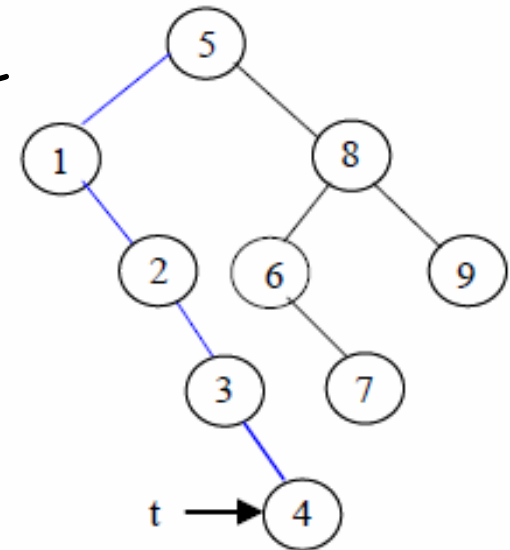
Si no existe ese padre, se retorna NULL, t era el nodo con valor máximo

Si tiene subárbol derecho, el sucesor es el mínimo del subárbol derecho.



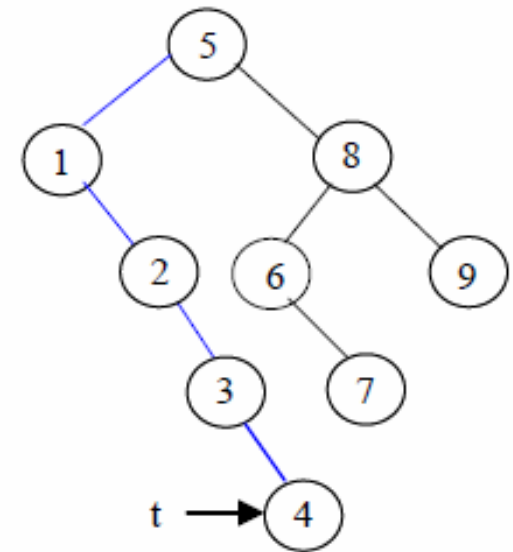
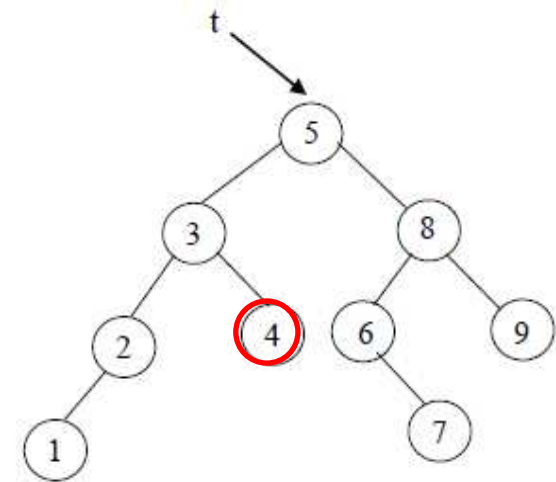
Nodo sucesor

```
pnode Sucesor(pnode t) {  
    pnode p;  
    if (t == NULL)  
        return(NULL); //si árbol vacío retorna NULL  
    if (t->right == NULL) {  
        p = t->padre; //p apunta al padre de t  
        while ( p!=NULL && t == p->right) {  
            t=p; p=t->padre;  
        } //se asciende  
        return(p); //  
    }  
    else  
        return( BuscaMinimo (t->right) ); //busca mín. en subárbol der.  
}
```



Algoritmos relacionados

- ❑ Nodo descendiente del subárbol izquierdo con mayor valor de clave.
 - Basta intercambiar left por right, right por left y min por max en los diseños desarrollado previamente para MenorDescendienteSD.
- ❑ Predecesor.
 - El código de la función predecesor es la imagen especular del código de sucesor.



Buscar

- Debido a la propiedad de los árboles binarios de búsqueda, si el valor buscado no es igual al de nodo actual, sólo existen dos posibilidades: que sea mayor o que sea menor.

Lo que implica que **el nodo buscado puede pertenecer a uno de los dos subárboles.**

- Cada vez que se **toma la decisión** de buscar en uno de los subárboles de un nodo, se están **descartando los nodos del otro subárbol.**
- En caso de **árboles balanceados**, se descarta la mitad de los elementos de la estructura, esto cumple el modelo: $T(n) = T(n/2) + c$, lo cual asegura complejidad logarítmica.

Buscar

```
pnode BuscarIterativo( pnode t, int valor) {  
    while ( t != NULL) {  
        if ( t->clave == valor ) // se debe implementar para distintos  
            return (t);        // tipos de datos  
        else {  
            if (t->clave < valor )  
                t = t->right; //desciende por la derecha  
            else  
                t = t->left;  //desciende por la izquierda  
        }  
    }  
    return (t); /* NULL No lo encontré*/  
}
```

Buscar II

```
pnode BuscarRecursivo( pnode t, int valor ) {  
    if ( t == NULL)  
        return (NULL); // árbol vacío o hijo de hoja  
    else {  
        if ( t->clave == valor )  
            return(t); // lo encontró  
        else {  
            if ( t->clave > valor )  
                t = BuscarRecursivo ( t->left, valor);  
            else  
                t = BuscarRecursivo ( t->right, valor);  
        }  
    }  
    return ( t ); // los retornos de las llamadas recursivas se pasan via t  
}
```


Buscar III

- Pueden eliminarse las asignaciones y el retorno final de esta forma:

```
pnodo BuscarRecursivo2( pnodo t, int valor ) {  
    if ( t == NULL)  
        return (NULL); /* árbol vacío o hijo de hoja */  
    else {  
        if ( t->clave == valor )  
            return (t); /* lo encontró */  
        else {  
            if ( t->clave > valor )  
                return ( BuscarRecursivo2 ( t->left, valor) );  
            else  
                return ( BuscarRecursivo2 ( t->right, valor));  
        }  
    }  
}
```

Buscar: Complejidad

- Si $T(a)$ es la complejidad de la búsqueda en un árbol de altura a . En cada iteración, el problema se reduce a uno similar, pero con la altura disminuida en uno, y tiene costo constante el disminuir la altura.

- Entonces: $T(a) = T(a-1) + \Theta(1)$

- La solución de esta recurrencia, es:

$$T(a) = a \Theta(1) = \Theta(a)$$

Pero en árboles de búsqueda se tiene que:

$$\log n \leq a \leq n$$

Entonces:

$$\Theta(\log n) \leq T(a) \leq \Theta(n)$$

8-Árboles Binarios

8.1 Definiciones y tipos de datos

8.2 Cálculos de complejidad

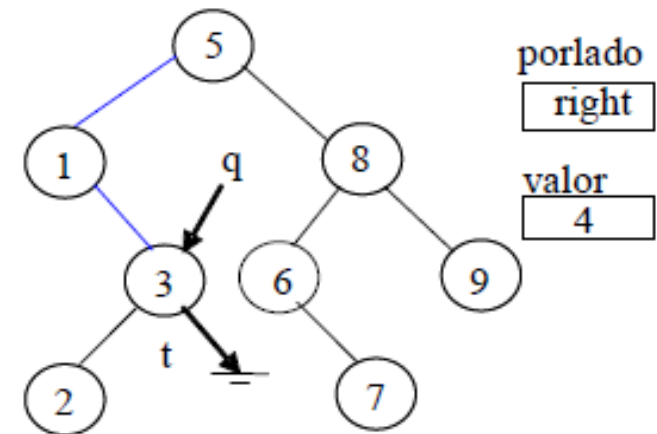
8.3 Operaciones básicas y de recorrido

8.4 Operaciones de consulta

8.5 Operaciones de modificación

Insertar Nodos: Iterativo I

- ❑ Primero se busca el sitio para insertar. Si el valor que se desea insertar ya estaba en el árbol, no se efectúa la operación; ya que no se aceptan claves duplicadas. Entonces: se busca el valor; y si no está, se inserta el nuevo nodo.
- ❑ Es preciso almacenar en una variable local q , la posición de la hoja en la que se insertará el nuevo nodo. q permite conectar el nuevo nodo creado al árbol.
- ❑ Se recorre una trayectoria de la raíz hasta una hoja para insertar. Entonces, si a es la altura, la complejidad de la inserción es: $\mathcal{T}(a)$.



Insertar Nodos: Iterativo I

```
typedef enum {left, right, vacio} modo;
pnodo InsertarIterativo(pnodo t, int valor) {
    pnodo q= t; modo porlado=vacio;
    while ( t != NULL) {
        if ( t->clave == valor ) {
            /* lo encontró, no inserta nodo */
            return (t);
        }
        else {
            q=t ;
            if (t->clave < valor){
                t = t->right;
                porlado=right;
            }
            else {
                t = t->left;
                porlado=left;
            }
        }
    }
}
```

```
/* Al salir del while q apunta al nodo en
   el arbol donde se insertará el nuevo
   nodo, y porlado la dirección */
/* El argumento t apunta a NULL */
t = CreaNodo(valor);
if (porlado==left)
    q->left=t;
else if(porlado==right)
    q->right=t;
return (t); /* Apunta al recién
insertado. Null si no se pudo
insertar*/
}
```

Insertar Nodos: Iterativo II

```
pnodo Insertar(pnodo t, int valor)
{
    pnodo *p = &t;
    while (*p != NULL)
    {
        if ((*p)->clave < valor)
            p = &((*p)->right);
        else if ((*p)->clave > valor)
            p = &((*p)->left);
        else
        {
            /* Ya estaba. No hace nada */
            return t;
        }
    }
    *p = CreaNodo(valor);
    return t;
}
```

```
int main()
{
    pnodo pRoot = NULL;
    pRoot = Insertar(pRoot, 5);
    Insertar(pRoot, 3);
    Insertar(pRoot, 7);
    Insertar(pRoot, 1);
    Insertar(pRoot, 4);

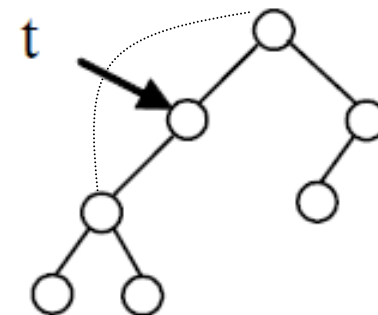
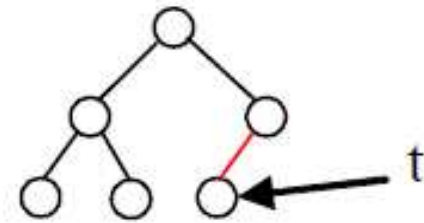
    return 0;
}
```

Insertar Nodos: Recursivo

```
pnode InsertarRecursivo( pnode t, int valor) {  
    if (t == NULL) {  
        t = CreaNodo(valor); //insertar en árbol vacío o en hoja.  
    }  
    else if (valor < t->clave) { //insertar en subárbol izquierdo.  
        t->left = InsertarRecursivo(t->left, valor);  
    }  
    else if (valor > t->clave) { //insertar en subárbol derecho  
        t->right = InsertarRecursivo (t->right, valor);  
    }  
    /* else: valor ya estaba en el árbol. No hace nada. */  
    return(t);  
}
```

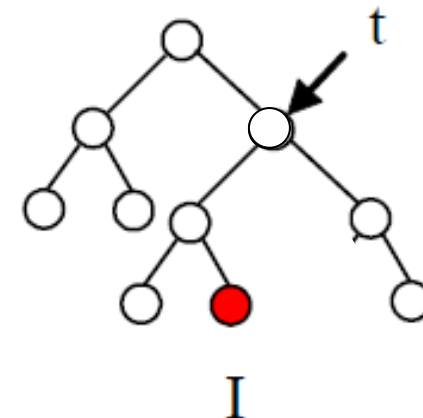
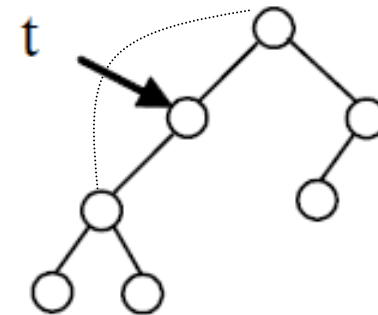
Descartar Nodos

- Primero se busca el nodo cuyo valor de clave es igual al valor pasado como argumento. Si no lo encuentra retorna NULL. Si lo encuentra, se producen varios casos. Lo importante es mantener la vinculación entre los elementos del árbol:
 - a) El nodo que se desea descartar es una hoja. En este caso, la operación es trivial, basta escribir un puntero con valor nulo. La estructura se conserva.
 - b) El nodo que se desea descartar es un nodo interno. i) con un hijo por la izquierda o la derecha el padre debe apuntar al nieto, para conservar la estructura de árbol. Esto implica mantener un puntero al padre, en el descenso.



Descartar Nodos

- b) El nodo que se desea descartar es un nodo interno.
 - i) con un hijo por la izquierda o la derecha el padre debe apuntar al nieto, para conservar la estructura de árbol. Esto implica mantener un puntero al padre, en el descenso.
 - ii) con dos hijos para conservar la estructura del árbol, se debe buscar I, el mayor descendiente del hijo izquierdo; o bien D, el menor descendiente del hijo derecho. Luego reemplazar la hoja obtenida por el nodo a descartar.
 - Se muestra la operación buscando D.



Descartar Nodos II

```
pnode Descartar(pnode t, int valor) {  
    pnode temp;  
    if (t == NULL)  
        printf("Elemento no encontrado\n");  
    else if (valor < t->clave) /* por la izquierda */  
        t->left = Descartar(t->left, valor);  
    else if (valor > t->clave) /* por la derecha */  
        t->right = Descartar(t->right, valor);  
    else { /* se encontró el elemento a descartar */  
        if (t->left && t->right) { /* dos hijos: remplazar con D */  
            temp = MenorDescendiente(t->right);  
            t->clave = temp->clave; //copia el nodo y borra la hoja  
            t->right = Descartar(t->right, temp->clave);  
        }  
        else { /* un hijo o ninguno */
```

...continua...

Descartar Nodos III

...continuacion...

```
        else { /* un hijo o ninguno */
            temp = t;
            if (t->left == NULL) /* sólo hijo derecho o sin hijos */
                t = t->right;
            else if (t->right == NULL) /* solamente un hijo izquierdo
*/
                t = t->left;
            free(temp); /*libera espacio */
        }
    }
    return(t);
}
```

Descartar Árbol

- Primero borra los subarboles y luego la raiz

```
pnode deltree(pnode t) {  
    if (t != NULL) {  
        t->left = deltree(t->left);  
        t->right = deltree(t->right);  
        free(t);  
    }  
    return NULL;  
}
```

Profundidad del Árbol

```
int Profundidad(pnodo t) {  
    int left=0, right = 0;  
    if(t==NULL)  
        return 0; //Si árbol vacío, profundidad 0  
    if(t->left != NULL)  
        left = Profundidad(t->left); //calcula prof. sub arb. I  
    if(t->right != NULL)  
        right = Profundidad(t->right); //calcula prof. sub arb.D  
    if( left > right) //si el izq tiene mayor profundidad  
        return left+1; //retorna profundidad del sub arb izq + 1  
    else  
        return right+1; //retorna prof. del sub arb der + 1  
}
```

Altura del Árbol

```
int Altura(pnodo T) {  
    int h, max;  
    if (T == NULL)  
        return -1;  
    else {  
        h = Altura (T->left);  
        max = Altura (T->right);  
        if (h > max)  
            max = h;  
        return(max+1);  
    }  
}
```

Numero de Hojas del Árbol

```
int NumerodeHojas(pnodo t) {  
    int total = 0; //Si árbol vacío, no hay hojas  
    if(t==NULL)  
        return 0;  
    // Si es hoja, la cuenta  
    if(t->left == NULL && t->right == NULL)  
        return 1;  
    //cuenta las hojas del subárbol izquierdo  
    if(t->left!= NULL)  
        total += NumerodeHojas(t->left);  
    //cuenta las hojas del subárbol derecho  
    if(t->right!=0)  
        total += NumerodeHojas(t->right);  
    return total; //total de hojas en subárbol  
}
```

Contar Nodos del Arbol

```
int ContarNodos(pnodo t) {  
    if (t == NULL)  
        return 0;  
    return (1 + ContarNodos(t->left) +  
            ContarNodos(t->right) );  
}
```

- ❑ Algunas otras posibilidades:
 - Contar nodos internos.
 - Contar nodos con valores menores o mayores que un valor dado.
 - Etc...

Partir el Arbol

```
pnodo split(int key, pnodo t,  
            pnodo *l, pnodo *r) {  
    while (t != NULL && t->  
        >clave != key) {  
        if (t->clave < key) {  
            *l = t;  
            t = t->right;  
            l = &((*l)->right);  
        } else {  
            *r = t;  
            t = t->left;  
            r = &((*r)->left);  
        }  
    }  
    // fin del while
```

```
        if (t == NULL) {  
            *l = NULL;  
            *r = NULL;  
        }  
        else { /* t->clave == key */  
            *l = t->left;  
            *r = t->right;  
        }  
        return t;  
    }
```

Insertar Nueva Raíz

```
pnode InsertarRaiz(int key, pnode t) {  
    pnode l, r;  
    t = split(key, t, &l, &r);  
    if (t == NULL) {  
        t = CreaNodo(key);  
        t->left = l;  
        t->right = r;  
    }  
    else {  
        t->left = l;  
        t->right = r;  
        Error();  
    }  
    return t;  
}
```

Unir dos Árboles

```
pnodo join(pnodo l, pnodo r) {  
    pnodo t = NULL;  
    pnodo *p = &t;  
    while (l != NULL && r != NULL) {  
        if (rand()%2) { //cara y sello.  
            *p = l;  
            p = &((*p)->right);  
            l = l->right;  
        } else {  
            *p = r;  
            p = &((*p)->left);  
            r = r->left;  
        }  
    }  
} // fin del while
```

```
if (l == NULL)  
    *p = r;  
else /* (r == NULL)  
    */  
    *p = l;  
return t;  
}
```