Tema 7: Estructuras de datos no lineales

Tecnología de la Programación

Índice

Índice

Introducción

- Estructura de datos lineal → cada elemento tiene como mucho un siguiente
- Estructura de datos no lineal → cada elemento puede tener varios siguientes
- Estructuras de datos no lineales:
    - Árboles
    - Tablas
    - Grafos

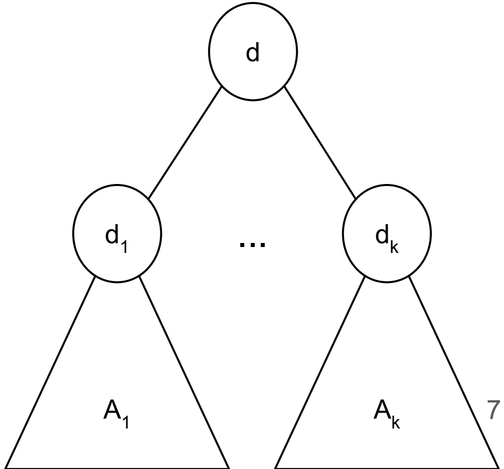
Índice

Árboles

- Un árbol es un conjunto de elementos llamados nodos con una relación (ser antecesor o ser padre de; ser descendiente o ser hijo de) que impone una estructura jerárquica
- Se denomina raíz al único nodo del árbol que no tiene antecesor
- Estructura adecuada para representar conjuntos entre cuyos elementos hay establecida una relación jerárquica (matemáticamente, un orden parcial)
- Aplicaciones:
  - Representar la estructura de fórmulas y expresiones matemáticas
  - Analizadores sintácticos
  - Transformación de programas recursivos en iterativos

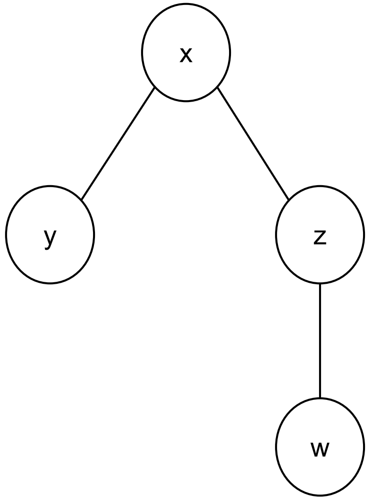
Definición formal

- Base:
  - Un árbol puede tener 0 nodos, caso en el que se denomina árbol nulo
  - Un nodo es un árbol
- Recurrente:
  - Si d es un dato de tipo T, y A_1, ..., A_k son árboles con raíces d_1, …, d_k, entonces se puede construir un árbol haciendo que d sea el nodo padre de d_1, …, d_k
  - A esta operación se la conoce como <u>enraizar</u>
  - En el árbol resultante, d es la raíz y A_1, ..., A_k son los subárboles



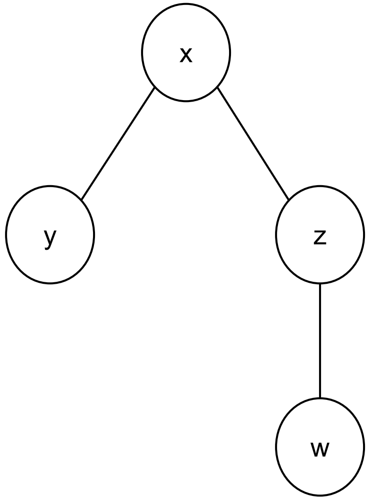
Términos asociados al concepto de árbol

- Llamaremos <u>nodo padre de un nodo</u> al primer ascendiente propio
  - Ejemplo: x es padre de y, y z es padre de w
- Llamaremos <u>camino</u> a una sucesión de nodos $d_1, \ldots, d_k$ tal que d_i es el padre de d_{i+1} $\forall$ i, 1≤i<k
  - Ejemplo: x, z, w
- Llamaremos <u>nivel</u> al conjunto de los nodos cuyos caminos desde la raíz tienen la misma longitud
  - Ejemplo: x está en en nivel 0, y y z en el nivel 1, y w en el nivel 2



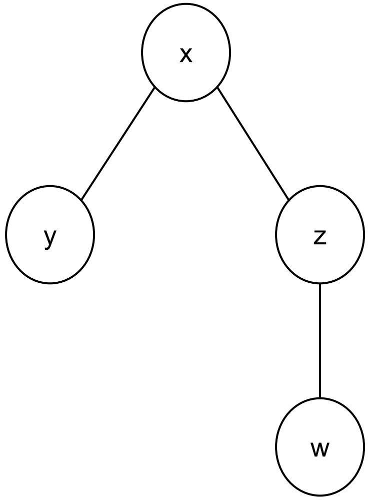
Términos asociados al concepto de árbol

- Llamaremos <u>nodo antecesor de un nodo</u> a cualquier nodo que se encuentra en cualquier nivel superior y tal que existe un camino entre ellos
  - Ejemplo: x es antecesor de z y w
- Llamaremos <u>nodo descendiente de un nodo</u> a cualquier nodo que se encuentre en cualquier nivel inferior y tal que exista un camino entre ellos
  - Ejemplo: w es descendiente de x y z
- Llamaremos <u>nodo hoja o nodo terminal</u> a un nodo que no tiene descendientes
  - Ejemplo: w e y son nodos hoja



Términos asociados al concepto de árbol

- Llamaremos nodo interior a un nodo no terminal
  - Ejemplo: x y z son nodos interiores
- Llamaremos profundidad o altura de un árbol al máximo de los niveles de sus nodos
- Llamaremos grado de un nodo al número de hijos del nodo
- Llamaremos grado de un árbol al máximo de los grados de sus nodos



Términos asociados al concepto de árbol

- Un árbol se dice n-ario si cada nodo es, como máximo, de grado n
- Un árbol se dice binario si es un árbol de grado 2

Índice

Árbol binario

Definición. Un árbol binario es un árbol tal que:

- Es el conjunto vacío, en cuyo caso se denomina árbol vacío; o,
- Existe un elemento distinguido llamado raíz y el resto de elementos se distribuyen en dos conjuntos disjuntos, cada uno de los cuales es un árbol binario, llamados subárboles izquierdo y derecho del árbol

Especificación de los árboles binarios

- Constructores:
  - Crear árbol vacío
  - Formar un árbol (enraizar):
    - no es ir añadiendo datos ya que no es una estructura lineal.
    - Idea: tenemos estructuras de datos que vamos a agrupar
- Acesores:
  - Árbol izquierdo
  - Árbol derecho
  - Raíz
- Deconstructores:
  - No hay
- Auxiliares:
  - Comprobar si árbol es vacío

TAD Árbol Binario

```
tad árbolBinario(tElemento)
usa
    tElemento
género
    arbolBin
operaciones
    acción iniciarArbol(sal arbolBin A)
    {Pre: }
    {Post: inicia A como un árbol vacío}
```

TAD Árbol Binario

Operaciones (cont)

```
acción enraizar(sal arbolBin A, e/s arbolBin Aiz, e/s arbolBin Ader,
                ent tElemento d)
{Pre: Aiz y Ader son árboles iniciados o construidos previamente}
{Post: Construye el árbol A cuya raíz es el dato d, y del cual penden los
       árboles Aiz y Ade. El acceso a los subárboles de A mediante Aiz y Ade
       queda anulado}

función izquierdo(arbolBin A) dev arbolBin
{Pre: A es un árbol no vacío}
{Post: devuelve el árbol izquierdo que pende del nodo raíz de A}

función derecho(arbolBin A) dev arbolBin
{Pre: A es un árbol no vacío}
{Post: devuelve el árbol derecho que pende del nodo raíz de A}
```

TAD Árbol Binario

Operaciones (cont)

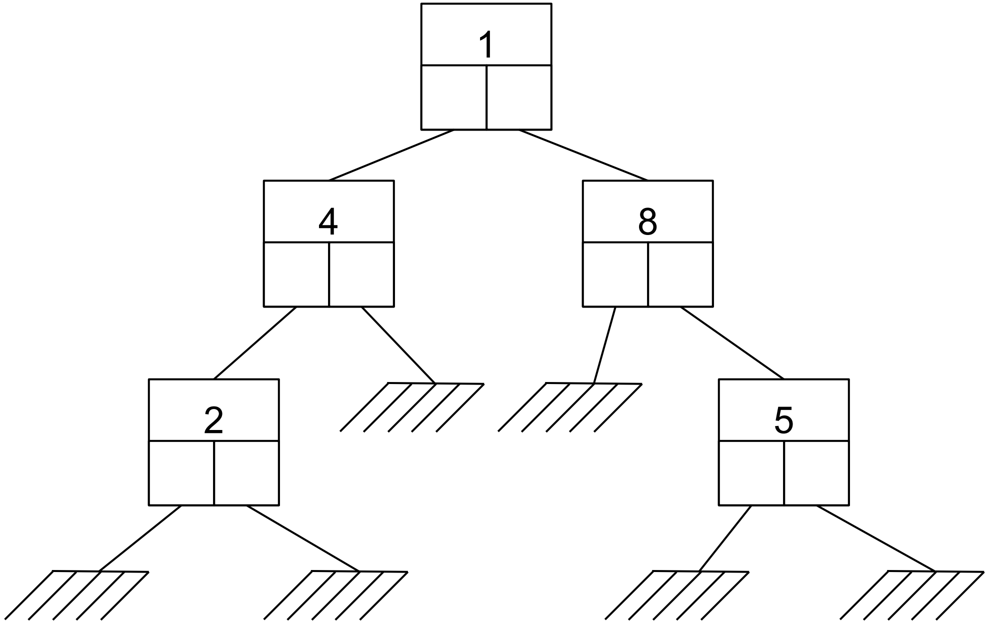
```
    función raíz(arbolBin A) dev tElemento
    {Pre: A es un árbol no vacío}
    {Post: devuelve el dato situado en el nodo raíz de A}

    función árbolVacío(arbolBin A) dev booleano
    {Pre: A es un árbol iniciado o creado previamente}
    {Post: devuelve Verdad si A está vacío y Falso en caso contrario}
```

Ejercicios usando especificación árboles binarios

- Contar número de nodos de un árbol binario
- Construir árbol simétrico a árbol binario
- Calcular profundidad de árbol binario

Implementación dinámica árboles binarios



Implementación dinámica de árboles binarios

```
tipo
    Celda = registro
        tElemento dato
        puntero a Celda izdo, dcho
    freg
    arbolBin = puntero a Celda
```

Interpretación. Un árbol es un puntero a una celda en la que se encuentran la raíz del árbol y dos punteros, `izdo` que apunta a al subárbol izquierdo, y `dcho` que apunta al subárbol derecho. Los punteros de los nodos hoja del árbol apuntan a `NULL`

TAD Árbol Binario

```
acción iniciarArbol(sal arbolBin A)
{Pre: }
{Post: inicia A como un árbol vacío}
principio
    A = NULL
fin

función izquierdo(arbolBin A) dev arbolBin
{Pre: A es un árbol no vacío}
{Post: devuelve el árbol izquierdo que pende del nodo raíz de A}
principio
    dev(dest(A).izdo)
fin
```

TAD Árbol Binario

```
función derecho(arbolBin A) dev arbolBin
{Pre: A es un árbol no vacío}
{Post: devuelve el árbol derecho que pende del nodo raíz de A}
principio
    dev(dest(A).dcho)
fin

función raíz(arbolBin A) dev tElemento
{Pre: A es un árbol no vacío}
{Post: devuelve el dato situado en el nodo raíz de A}
principio
    dev(dest(A).dato)
fin
```

TAD Árbol Binario

```
función árbolVacío(arbolBin A) dev booleano
{Pre: A es un árbol iniciado o creado previamente}
{Post: devuelve Verdad si A está vacío y Falso en caso contrario}
principio
    dev(A==NULL)
fin
```

TAD Árbol Binario

```
acción enraizar(sal arbolBin A, e/s arbolBin Aiz, e/s arbolBin Ader, ent tElemento d)
{Pre: Aiz y Ader son árboles iniciados o construidos previamente}
{Post: Construye el árbol A cuya raíz es el dato d, y del cual penden los árboles Aiz y Ade. El
acceso a los subárboles de A mediante Aiz y Ade queda anulado}
principio
      A = reservar(Celda)
      si A != NULL hacer
            dest(A).dato = d
            dest(A).izdo = Aiz
            dest(A).dcho = Ader
            Aiz = NULL // Acceso a Aiz queda anulado
            Ader = NULL // Acceso a Ader queda anulado
      fsi
fin
```

Complejidad de las operaciones O(1)

Índice

1. Introducción
2. **Árboles**
    1. Definiciones
    2. Árboles binarios
    3. **Enriquecimiento de árboles binarios**
    4. Árboles de búsqueda
    5. Monticulos

Enriquecimiento de árboles binarios

- Enriquecemos TAD árbol binario con operaciones de recorrido en profundidad (frente a lo que sería un recorrido en anchura)
- Un <u>recorrido</u> consiste en visitar todos los datos (nodos) de un árbol exáctamente una vez
- Hay dos tipos de recorrido:
    - En anchura:
        - Útil en árboles sin huecos ya que árbol se puede representar como estructura lineal
        - Nodo que ocupa posición i tiene por hijos a los que ocupan posiciones 2i y 2i+1
        - Representación estructura de árbol con estructura lineal
    - En profundidad:
        - Preorden
        - Inorden
        - Postorden

Recorridos en profundidad

- Hay 3 recorridos en profundidad característicos para árboles binarios
- Forma más sencilla de describir operaciones en árboles binarios es hacerlo recursivamente en tres pasos:
  - Acceso a raíz de un nodo
  - Acceso al subárbol izquierdo de ese nodo
  - Acceso al subárbol derecho de ese nodo
- Cada uno de los 3 recorridos se caracteriza por orden de acceso a cada uno de los tres elementos anteriores:
  - Preorden: raíz → izquierdo → derecho
  - Inorden: izquierdo → raíz → derecho
  - Postorden: izquierdo → derecho → raíz

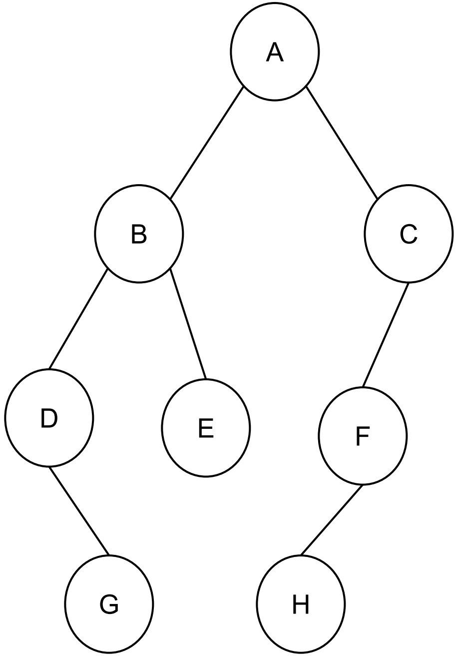
Recorrido preorden

Preorden: raíz → izquierdo → derecho

Recorrido:

1. Trata raíz
2. Recorre en preorden subárbol izquierdo
3. Recorre en preorden subárbol derecho

Ejemplo: A - B - D - G - E - C - F - H



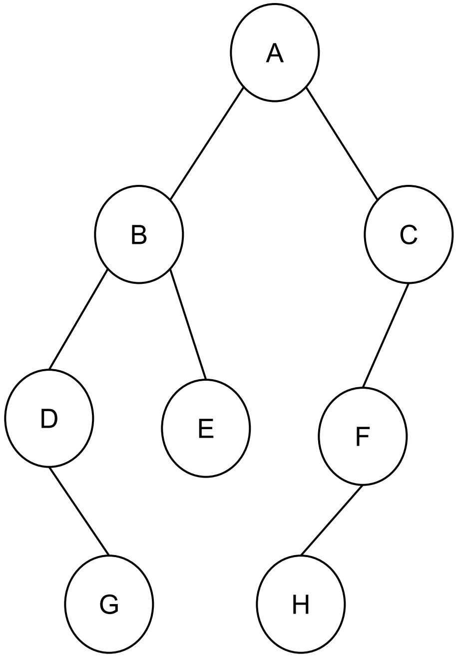
Recorrido inorden

Inorden: izquierdo → raíz → derecho

Recorrido:

1. Recorre en inorden subárbol izquierdo
2. Trata raíz
3. Recorre en inorden subárbol derecho

Ejemplo: D - G - B - E - A - H - F - C



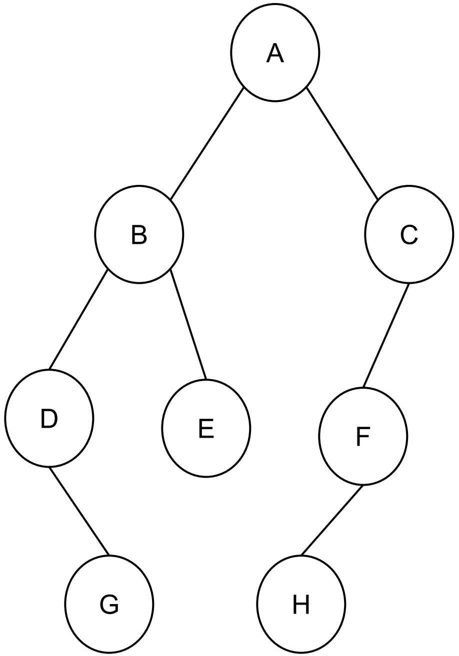
Recorrido postorden

Postorden: izquierdo → derecho → raíz

Recorrido:

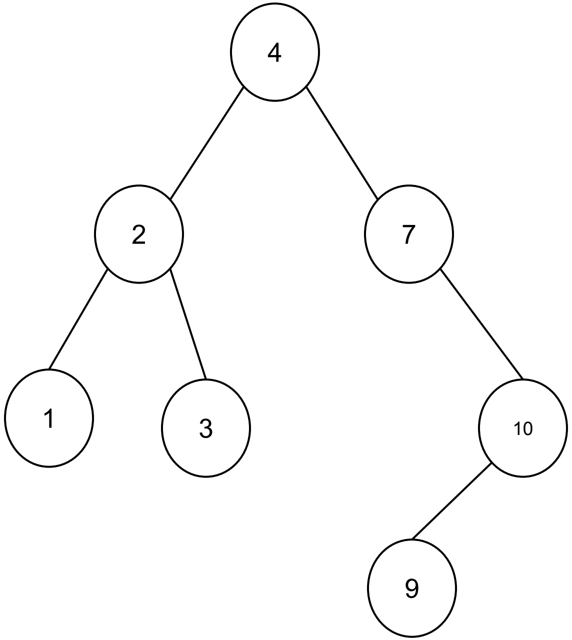
1. Recorre en postorden subárbol izquierdo
2. Recorre en postorden subárbol derecho
3. Trata raíz

Ejemplo: G - D - E - B - H - F - C - A



Ejercicio

Recorrido preorden, inorden y postorden del siguiente árbol binario



Enriquecimiento de árboles binarios

Enriquecemos el TAD árbol binario añadiendo estas operaciones de recorrido

Nota. Un árbol binario viene caracterizado por 2 recorridos (no por 1) ya que pueden existir árboles distintos con un mismo recorrido

Árboles binarios enriquecidos:

- Especificación
- Implementación

TAD Árbol Binario Enriquecido

```
tad árbolBinarioEnriquecido
usa
    arbolBin
parámetros
    géneros
        tElemento
    operaciones
        acción tratamiento(e/s tElemento d)
        {cualquier operación sobre el dato d}
```

TAD Árbol Binario Enriquecido

<u>**operaciones**</u>

```
    acción preOrden(ent arbolBin A)
    {Pre: A es un árbol iniciado previamente}
    {Post: recorre en preOrden el árbol A aplicando la operación tratamiento}

    acción inOrden(ent arbolBin A)
    {Pre: A es un árbol iniciado previamente}
    {Post: recorre en inOrden el árbol A aplicando la operación tratamiento}

    acción postOrden(ent arbolBin A)
    {Pre: A es un árbol iniciado previamente}
    {Post: recorre en postOrden el árbol A aplicando la operación tratamiento}
```

Implementación TAD Árbol Binario Enriquecido

```
acción preOrden(ent arbolBin A)
{Pre: A es un árbol iniciado previamente}
{Post: recorre en preOrden el árbol A aplicando la operación tratamiento}
principio
     si not(árbolVacío(A)) entonces
          tratamiento(raíz(A))
          preOrden(izquierdo(A))
          preOrden(derecho(A))
     fsi
fin
```

Implementación TAD Árbol Binario Enriquecido

```
acción inOrden(ent arbolBin A)
{Pre: A es un árbol iniciado previamente}
{Post: recorre en inOrden el árbol A aplicando la operación tratamiento}
principio
     si not(árbolVacío(A)) entonces
          inOrden(izquierdo(A))
          tratamiento(raíz(A))
          inOrden(derecho(A))
     fsi
fin
```

Implementación TAD Árbol Binario Enriquecido

```
acción postOrden(ent arbolBin A)
{Pre: A es un árbol iniciado previamente}
{Post: recorre en postOrden el árbol A aplicando la operación tratamiento}
principio
     si not(árbolVacío(A)) entonces
          postOrden(izquierdo(A))
          postOrden(derecho(A))
          tratamiento(raíz(A))
     fsi
fin
```

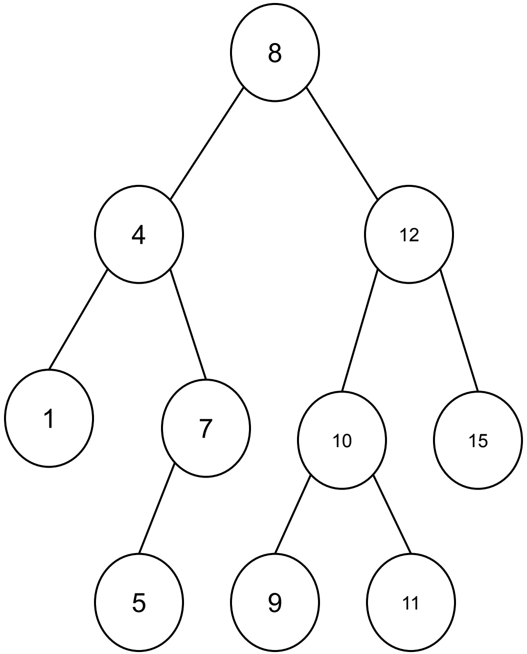
Índice

Árboles de búsqueda

- Tipo particular de árboles binarios
- Pueden definirse cuando el tipo de los elementos del árbol posee relación de orden total:
  - Reflexiva: $\forall\ x \in A,\ x \sim x$
  - Antisimétrica: $\forall\ x,y \in A,\ x \sim y$ e $y \sim x \Rightarrow x = y$
  - Transitiva: $\forall\ x,y,z \in A,\ x \sim y$ e $y \sim z \Rightarrow x \sim z$
  - Total: $\forall\ x,y \in A,\ x \sim y$ o $y \sim x$
- Propiedades:
  - Todos los elementos del subárbol izquierdo son menores estrictos que el elemento raíz
  - Elemento raíz es menor estricto que todos los elementos del subárbol derecho
  - Subárboles izquierdo y derecho son árboles de búsqueda
- Algunas variantes admiten elementos repetidos:
  - Todos los elementos del subárbol izquierdo serán menores o iguales que la raíz

Árboles de búsqueda

Ejemplo:



Especificación árboles de búsqueda

Operaciones que no tenían sentido en general con árboles binarios, sí que lo tienen en árboles de búsqueda:

- Insertar: en un árbol de búsqueda se sabe donde, en uno binario hay muchas posibilidades
- Eliminar elementos
- Saber si un elemento está:
  - En árbol binario general se puede hacer con un recorrido (pre-in-postorden) siendo la acción tratamiento el ver si el elemento coincide con la raíz → Complejidad O(n)
  - Árboles de búsqueda → incluimos operación en la especificación para implementarla de manera más eficiente → O(log n)

Árboles de búsqueda

- Especificación
- Implementación

TAD Árbol Búsqueda

```
tad árbolBúsqueda
usa
     arbolBin
parámetros
     géneros
          tElemento
     operaciones
          función esIgual(tElemento d1, tElemento d2)
          {Pre: }
           {Post: devuelve verdad si d1 es igual a d2 y falso en caso contrario}

          función esMenor(tElemento d1, tElemento d2)
          {Pre: }
           {Post: devuelve verdad si d1 es menor a d2 y falso en caso contrario}
```

TAD Árbol Búsqueda

```
operaciones
     acción insertar(e/s arbolBin A, ent tElemento d)
     {Pre: A es un árbol de búsqueda iniciado previamente, y d no está en A}
     {Post: Inserta en A el elemento d de manera que A sigue siendo un árbol
            de búsqueda.}

     acción borrar(e/s arbolBin A, ent tElemento d)
     {Pre: A es un árbol de búsqueda iniciado previamente, y d está en A}
     {Post: Elimina de A el elemento d de manera que A sigue siendo un árbol de
            búsqueda}

     función está?(arbolBin A, tElemento d)
     {Pre: A es un árbol de búsqueda iniciado previamente}
     {Post: devuelve Verdad si d está en A y Falso en caso contrario}
```

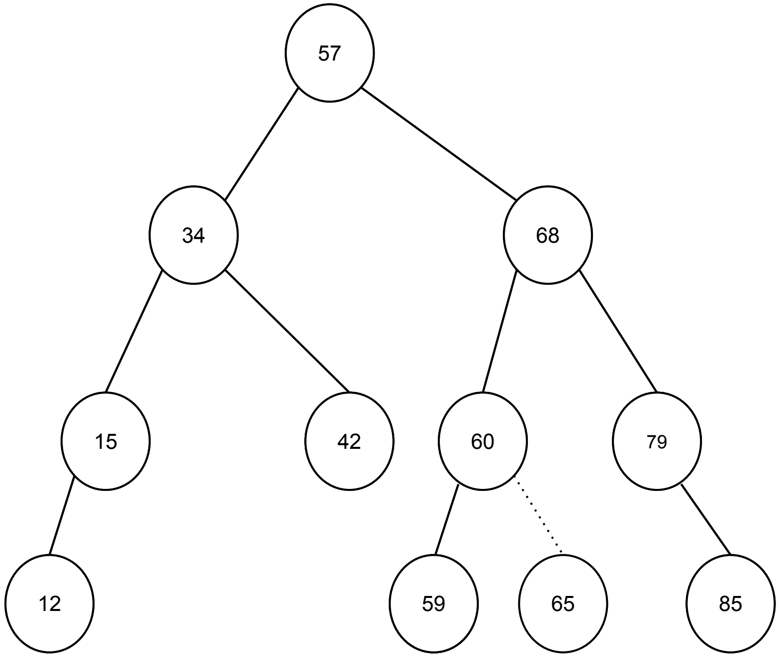
Implementación TAD Árbol Búsqueda

```
función está?(arbolBin A, tElemento d)
{Pre: A es un árbol de búsqueda iniciado previamente}
{Post: devuelve Verdad si d está en A y Falso en caso contrario}
principio
     si árbolVacío(A) entonces
          dev(FALSO)
     sino
          si esIgual(raíz(A),d) entonces
               dev(VERDAD)
          sino
               si esMenor(d,raíz(A)) entonces
                    dev(está?(izquierdo(A),d))
               sino
                    dev(está?(derecho(A),d))
               fsi
     fsi
fin
```

Implementación TAD Árbol Búsqueda

Método insertar: sólo hay una posición posible como nodo hijo

Insertar el 65 en el siguiente árbol de búsqueda



Implementación TAD Árbol Búsqueda

Método insertar:

- Caso base: árbol vacío → este caso se alcanza al encontrar punto de inserción
- Caso recurrente:
  - d > raíz(A) → insertar en el subárbol derecho
  - d < raíz(A) → insertar en el subárbol izquierdo

Implementación TAD Árbol Búsqueda

```
acción insertar(e/s arbolBin A, ent tElemento d)
{Pre: A es un árbol de búsqueda iniciado previamente, y d no está en A}
{Post: Inserta en A el elemento d de manera que A sigue siendo un árbol de búsqueda.}
variables
      arbolBin iz,de
principio
      si árbolVacío(A) entonces
            iniciarArbol(iz)
            iniciarArbol(de)
            enraizar(A,iz,de,d)
      sino
            si esMenor(d,raíz(A)) entonces
                  iz = izquierdo(A)
                  insertar(iz,d)
            sino
                  de = derecho(A)
                  insertar(de,d)
            fsi
      fsi
fin
```

Implementación TAD Árbol Búsqueda

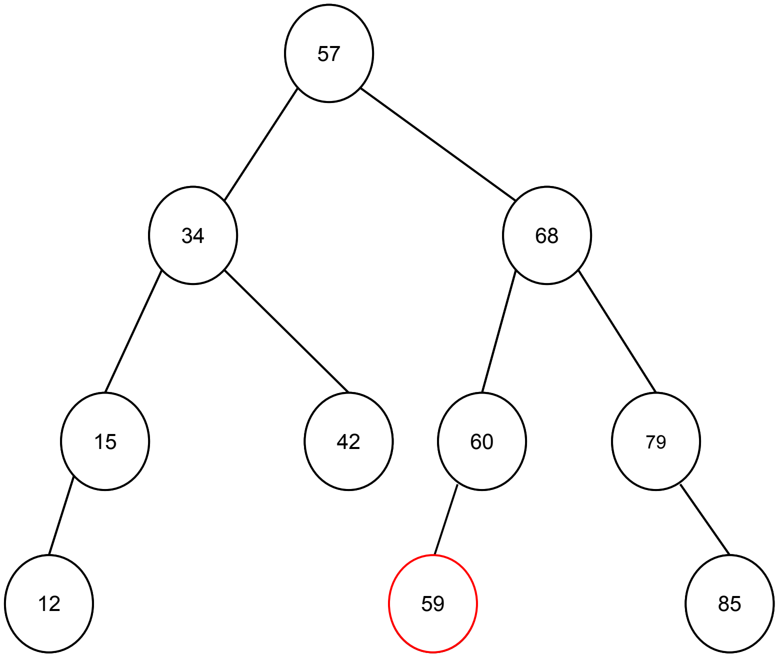
Método eliminar:

- Caso base: árbol vacío → no hay nada
- Caso recurrente:
  - d > raíz(A) → eliminar en el subárbol derecho
  - d < raíz(A) → eliminar en el subárbol izquierdo
  - d = raíz(A) → Casos

Implementación TAD Árbol Búsqueda

Método eliminar: d = raíz(A)

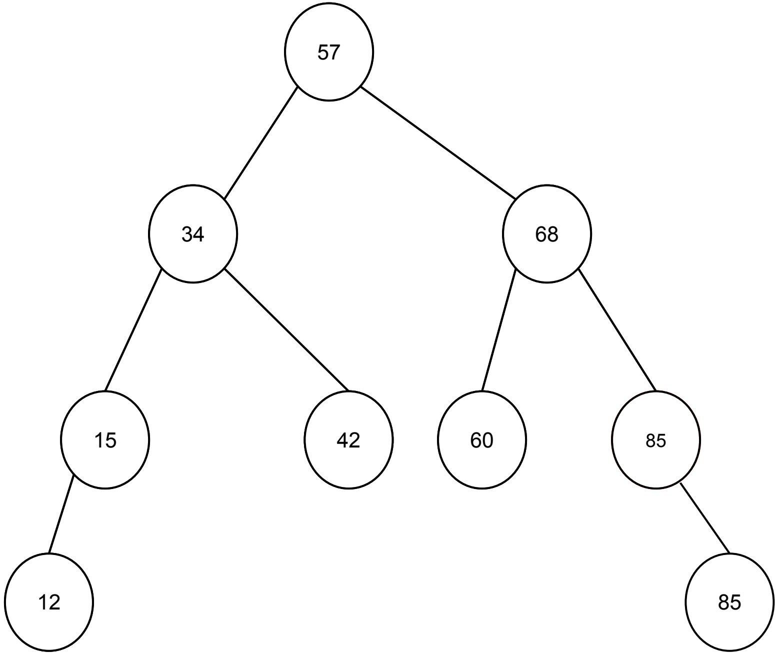
A. Borrar nodo hoja



Implementación TAD Árbol Búsqueda

Método eliminar: d = raíz(A)

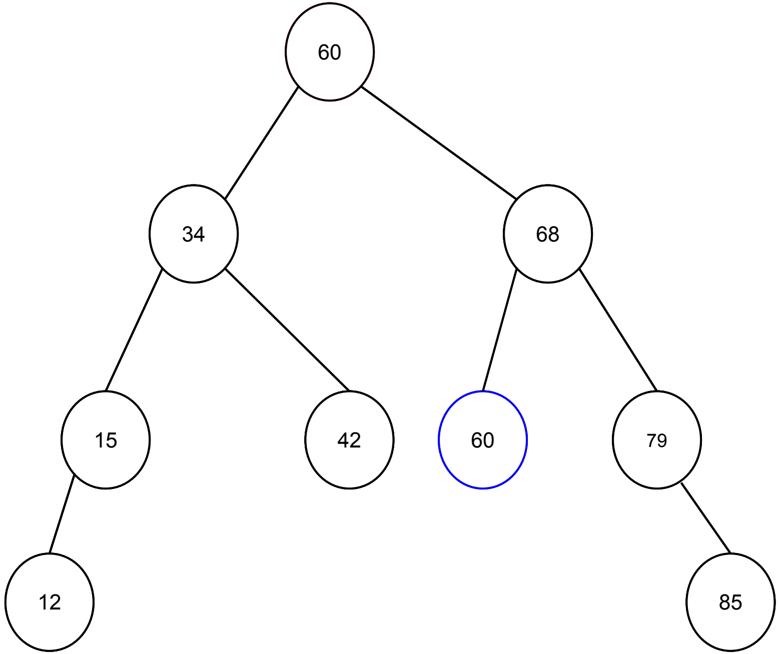
B. Borrar nodo con un único hijo

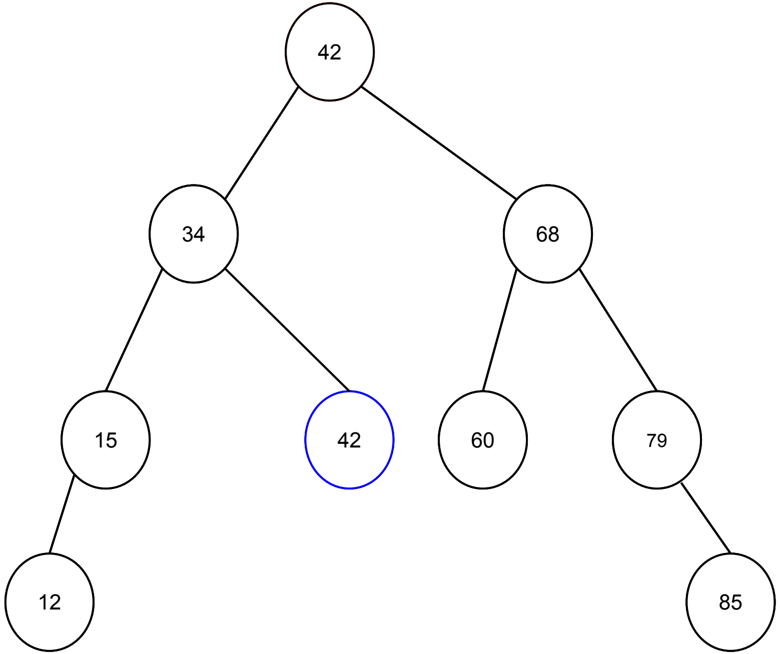


Implementación TAD Árbol Búsqueda

Método eliminar: d = raíz(A)

C. Borrar nodo con dos hijos





Implementación TAD Árbol Búsqueda

```
acción borrar(e/s arbolBin A, ent tElemento d)
{Pre: A es un árbol de búsqueda iniciado previamente, y d está en A}
{Post: Borra de A el elemento d de manera que A sigue siendo un árbol de búsqueda.}
variables
      puntero a Celda aux
principio
      si not(árbolVacío(A)) entonces
            si esMenor(d,raíz(A)) entonces
                  aux = dest(A).izdo
                  borrar(aux,d)
            sino
                  si esMenor(raíz(A),d) entonces
                        aux = dest(A).dcho
                        borrar(aux,d)
                  sino
                        si árbolVacío(izquierdo(A)) entonces
                              aux = A
                              A = dest(A).dcho
                              liberar(aux)
                        sino
                              dest(A).dato = máximo(izquierdo(A))
                              aux = dest(aux).izdo
                              borrar(aux,dest(A).dato)
                        fsi
                  fsi
fin   fsi   fsi
```

Implementación TAD Árbol Búsqueda

```
función máximo(arbolBin A) dev tElemento
{Pre: A es un árbol de búsqueda iniciado previamente y es no vacío}
{Post: Devuelve el máximo elemento del árbol de búsqueda A}
principio
     si(árbolVacío(derecho(A)) entonces
          dev raíz(A)
     sino
          dev  máximo(derecho(A))
     fsi
fin
```

El máximo elemento de un árbol de búsqueda es el hijo más a la derecha del árbol derecho

Árboles de búsqueda

Comentarios:

- Realizar búsqueda en árbol de búsqueda de n nodos tiene coste O(log n)
- Por ello se utilizan, como su nombre indica, fundamentalmente para esta tarea: realizar búsquedas
- Si búsqueda en un vector es habitual:
    1. Paso del vector al árbol de búsqueda → O(n)
    2. Búsqueda en el árbol → O(log n)

Índice

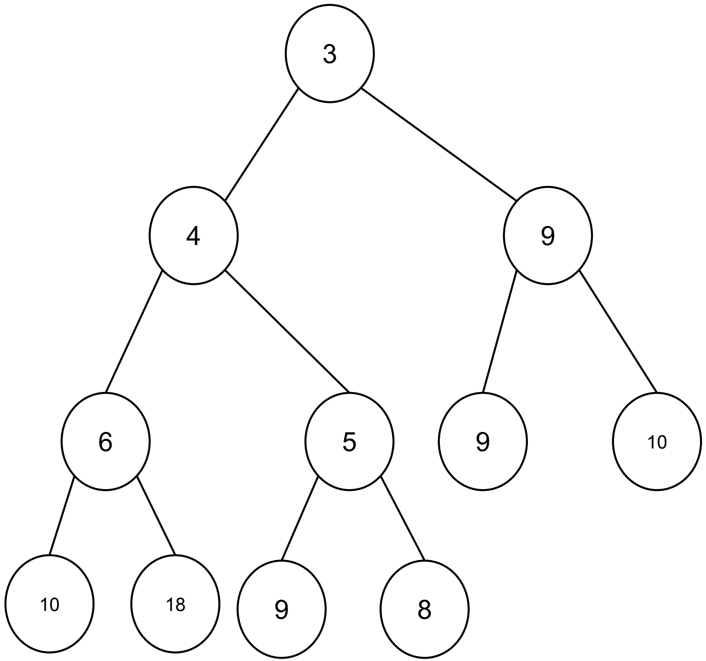
1. Introducción
2. **Árboles**
    1. Definiciones
    2. Árboles binarios
    3. Enriquecimiento de árboles binarios
    4. Árboles de búsqueda
    5. **Monticulos**

Montículos

- Tipo particular de árbol binario que puede definirse cuando entre los elementos del árbol existe definida una relación de orden total
- Montículo es un árbol que:
    - Es casi completo (no hay huecos)
    - El elemento raíz es menor o igual que el resto de los elementos del árbol
    - Los subárboles izquierdo y derecho son montículos

Montículos

Ejemplo:



Montículos

- Imaginad que el valor representa algún tipo de prioridad (mayor prioridad cuanto más pequeño es el valor)
- Este tipo corresponde con lo que serían las colas con prioridad
- En este tipo de estructuras tienen sentido algunas operaciones no definidas para árboles en general

Especificación del TAD montículo

```
especificación TAD Montículo
parámetros
     géneros
          tElemento
     operaciones
          función esMenor(d1: tElemento, d2: tElemento) dev booleano
          {Pre:}
          {Post: Devuelve VERDAD si d1 es menor que d2, y FALSO en caso contrario}
          acción permutar(e/s d1: tElemento, e/s d2: tElemento)
          {Pre:}
          {Post: Intercambia los valores de d1 y d2}
géneros
     montículo
```

Especificación del TAD montículo

```
operaciones
     acción iniciarMontículo(sal M: montículo)
     {Pre:}
     {Post: inicia M como el montículo vacío}
     acción insertar(e/s M: montículo, ent d: tElemento)
     {Pre: M ha sido iniciado previamente}
     {Post: inserta en el montículo M el elemento d}
     función mínimo (M: montículo) dev tElemento
     {Pre: M ha sido iniciado previamente}
     {Post: devuelve el elemento más pequeño del montículo M}
     acción eliminarMinimo(e/s M: montículo)
     {Pre: M ha sido iniciado previamente}
     {Post: elimina del montículo M el elemento mínimo}
     función monticuloVacio (M: montículo) dev booleano
     {Pre: M ha sido iniciado previamente}
     {Post: devuelve VERDAD si M está vacío y FALSO en caso contrario}
     función alturaMontículo (M: montículo) dev entero
     {Pre: M ha sido iniciado previamente}
     {Post: devuelve la altura del montículo M}
```

Insertar un dato en un montículo

Al insertar un nodo en un montículo:

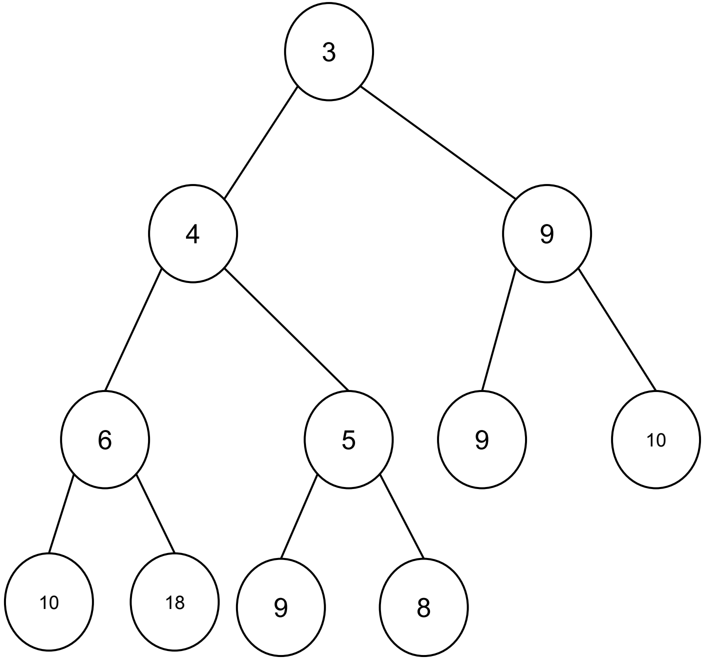
- la posición de ese nuevo nodo es única, pero
- se pierde la propiedad de montículo

Solución:

- Utilizar una acción llamada flotar

Insertar un dato en un montículo

Ejemplo. Suponer que queremos insertar el nodo 2 en el siguiente árbol



Insertar un dato en un montículo

Complejidad:

- Orden de la operación flotar → O(log n) siendo n el número de nodos
- Orden de insertar → O(log n)

Eliminar elemento mínimo del montículo

En montículos existe elemento distinguido:

- Elemento más pequeño del montículo → raíz
- Tiene sentido eliminar el elemento al que se ha tenido acceso

Eliminar elemento mínimo:

- Coger el último nodo y poner su dato como la raíz
- Hundirlo: elegir una estrategia (e.g. cambiarlo por el menor de sus hijos)
  - Hundir recorre una rama → O(log n)
  - Orden de borrar → O(log n)

TAD montículo para ordenar vectores

Método Heapsort:

- Versión 1: creando un montículo
- Versión 2: sin crear montículo pero organizando datos del vector de manera adecuada

TAD montículo para ordenar vectores. Versión 1

1. A partir del vector obtener un montículo

```
para i := 0 hasta n-1 hacer
    insertar(M, v[i])
fpara
```

2. Extraer mínimo, almacenarlo en el vector, y eliminar mínimo

```
para i := 0 hasta n-1 hacer
    v[i] := minimo(M)
    eliminarMinimo(M)
fpara
```

Pega: necesita espacio adicional en memoria para almacenar vector y montículo