Ejercicios de Árboles

Información del Proyecto

Descripción	Detalles
Profesores	Sergio Cavero y Salvador Sánchez
Asignatura	Estructuras de Datos
Universidad	Universidad Rey Juan Carlos
Curso	2024/2025

Ejercicio 1: operaciones simples sobre árboles binarios

En estos ejercicios vamos a trabajar con árboles binarios desde un punto de vista general, es decir, no vamos a centrarnos en las operaciones de inserción, eliminación o búsqueda ya que dependen de la estructura del árbol. En su lugar, nos centraremos en los recorridos y en la creación de nuevas funciones que nos permitan trabajar con los árboles binarios.

Para realizar estos ejercicios, vamos a utilizar un árbol binario con información entera para cada nodo. Para ello, utilizaremos el siguiente código base:

- `arboles_ej1.pas`: programa principal que ejecutaremos para comprobar que hemos implementado correctamente los ejercicios.

- `uBinaryTree.pas`: unidad que contiene la definición del árbol binario. Deberemos implementar el recorrido en esta unidad.

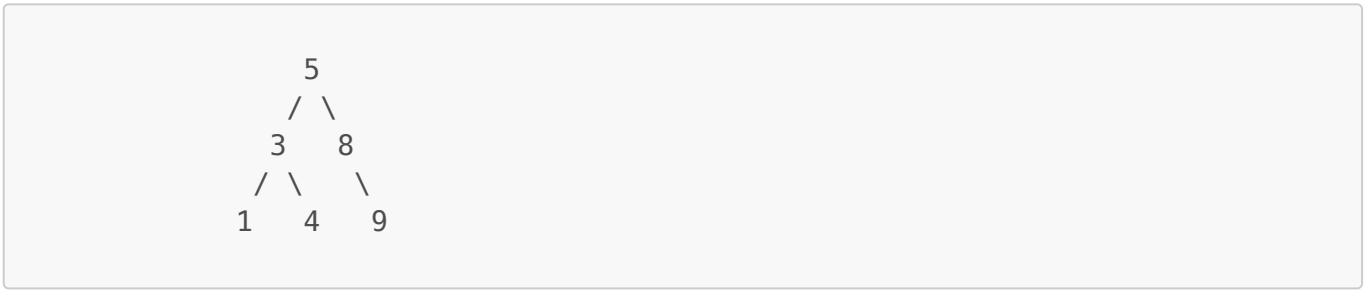
Ejercicio 1.0: crear el árbol binario

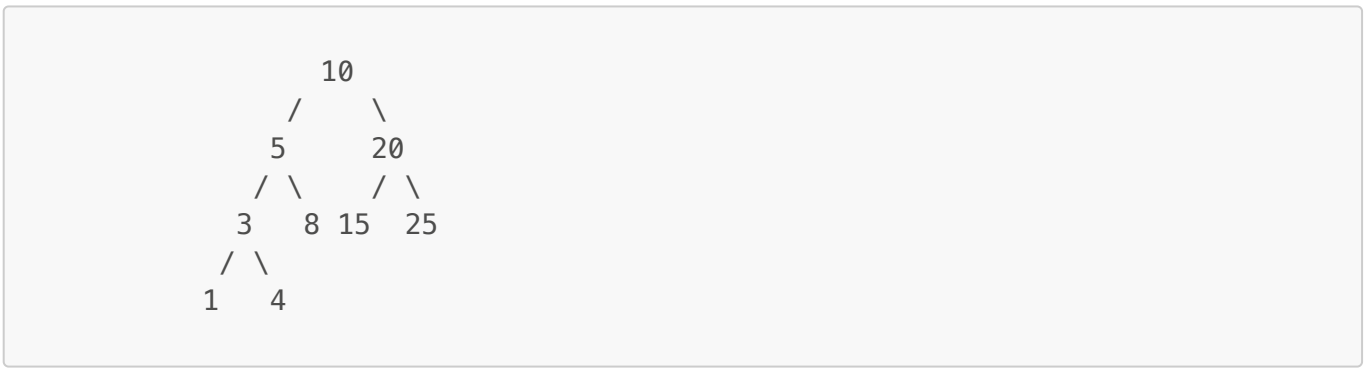
En el fichero `arboles_ej1.pas` hay dos funciones: `procedure crear_arbol(var a: tbinaryTree);` y `procedure crear_arbol2(var a: tbinaryTree);`. Debes implementar estos métodos suponiendo que el método `insertar` (`add`) realiza la inserción de un nodo en el árbol binario. Concretamente, la inserción se realiza como en un árbol binario de búsqueda, es decir:

- Si el árbol está vacío, se inserta el nodo como raíz.
- Si el árbol no está vacío, se compara el valor del nodo a insertar con el valor del nodo raíz. Si el valor es menor, se inserta en el subárbol izquierdo. Si el valor es mayor, se inserta en el subárbol derecho.

Suponemos para este ejercicio que el árbol binario no tiene nodos duplicados.

Los árboles a crear son los siguientes:





Ejercicio 1.1: in-order inverso

Un recorrido in-orden es un recorrido de un árbol binario en el que se visitan primero los nodos de la izquierda de un nodo, luego el propio nodo (si pensamos en el árbol completo, la raíz) y finalmente los nodos de la derecha. En este ejercicio vamos a implementar un recorrido in-orden inverso, es decir, primero se visitan los nodos de la derecha, luego el nodo raíz y finalmente los nodos de la izquierda.

Ejercicio 1.2: profundidad máxima

Implementar una función que devuelva la profundidad máxima de un árbol binario. La profundidad de un árbol binario es la longitud del camino más largo desde la raíz hasta una hoja. Como sabes, un nodo hoja es un nodo que no tiene hijos.

Ejercicio 1.3: contar nodos

Implementar una función que cuente el número de nodos de un árbol binario. La función debe devolver el número total de nodos del árbol, sean estos hojas o nodos intermedios.

Ejercicio 1.4: contar hojas

Implementar una función que cuente el número de nodos hoja de un árbol binario. Ya sabes que un nodo hoja es aquél que no tiene hijos. La función debe devolver el número total de hojas del árbol.

Ejercicio 1.5: contar nodos internos

Implementar una función que cuente el número de nodos internos de un árbol binario. Un nodo interno es un nodo que tiene al menos un hijo. La función debe devolver el número total de nodos internos del árbol.

Ejercicio 1.6: es completo

Implementar una función que determine si un árbol binario es completo. Un árbol binario es completo si todos sus niveles están completamente llenos. Es decir, si todos los nodos tienen dos hijos, excepto las hojas que no tendrían hijos. La función debe devolver verdadero si el árbol es completo y falso en caso contrario. Cabe mencionar que se espera que el árbol tenga al menos un nodo.

Ejercicio 1.7: el mayor valor de una hoja

Implementar una función que devuelva el mayor valor de una hoja de un árbol binario. La función debe devolver el valor máximo de las hojas del árbol. En caso de que el árbol no tenga ningún nodo, la función debe devolver cero.

Ejercicio 1.8: suma del valor de las hojas

Implementar una función que devuelva la suma de los valores de las hojas de un árbol binario. Si el árbol es vacío la función debe devolver cero.

Ejercicio 1.9: cuenta número de nodos pares

Implementar una función que cuente el número de nodos pares de un árbol binario. Un nodo es par si su valor es par. La función debe devolver el número total de nodos pares del árbol.

Ejercicio 1.10: contar nodos en un nivel

Implementar una función que cuente el número de nodos en un nivel determinado de un árbol binario. La raíz del árbol se considera el nivel 0. Los hijos inmediatos de la raíz nivel 1, etc.

Ejercicio 2: operaciones avanzadas sobre árboles binarios de búsqueda

En este ejercicio trabajaremos con árboles binarios de búsqueda. Implementaremos funciones que nos permitan realizar operaciones avanzadas sobre estos árboles.

Para realizar estos ejercicios, vamos a utilizar un árbol binario de búsqueda con información entera para cada nodo. Para ello, utilizaremos el siguiente código base:

- `arboles_ej2.pas`: programa principal que ejecutaremos para comprobar que hemos implementado correctamente los ejercicios 2.1, 2.2 y 2.3.
- `arboles_ej2_4.pas`: programa principal que ejecutaremos para comprobar que hemos implementado correctamente el ejercicio 2.4.
- `uBinarySearchTree.pas`: unidad que contiene la definición del árbol binario de búsqueda. Deberemos implementar el recorrido en esta unidad.

Ejercicio 2.1: mismos nodos en los subárboles izquierdo y derecho

Implementar una función que determine si un árbol binario de búsqueda tiene la misma cantidad de nodos en su subárbol izquierdo y derecho. La función debe devolver verdadero si ambos subárboles tienen el mismo número de nodos y falso en caso contrario.

Ejercicio 2.2: niveles completos

Implementar una función que determine si todos los niveles de un árbol binario de búsqueda están completos. Un nivel está completo si todos los nodos de ese nivel tienen dos hijos o son hojas. La función debe devolver verdadero si el árbol tiene todos sus niveles completos y falso en caso contrario.

Ejercicio 2.3: añadir nodos de un árbol a otro

Implementar un procedimiento que añada todos los nodos de un árbol binario de búsqueda `b` a otro árbol binario de búsqueda `a`. Los nodos de `b` deben añadirse a `a` siguiendo un recorrido preorden.

Ejercicio 2.4: multiplicidad de claves

Modifica el árbol binario para que cada nodo puede tener una multiplicidad asociada. La multiplicidad indica cuántas veces aparece una clave en el árbol.

Para ello, deberás hacer lo siguiente:

1. Añadir multiplicidad a los nodos: Modificar la estructura de los nodos del árbol binario de búsqueda para que incluyan un campo adicional llamado `multiplicidad`. Este campo debe almacenar el número de veces que una clave aparece en el árbol. Por defecto, la multiplicidad de un nodo será 1.
2. Añadir claves repetidas: Modificar el procedimiento `add` para que, si se intenta añadir una clave que ya existe en el árbol, se incremente la multiplicidad del nodo correspondiente en lugar de crear un nuevo nodo.
3. Eliminar claves con multiplicidad: Modificar el procedimiento `remove` para que, si se elimina una clave con una multiplicidad mayor a 1, se reduzca la multiplicidad del nodo en lugar de eliminarlo completamente. Si la multiplicidad llega a 0, el nodo debe eliminarse del árbol.
4. Nuevo método para obtener la multiplicidad de un nodo: Implementar una función que devuelva la multiplicidad de un nodo dado en el árbol binario de búsqueda. Si la clave no está presente en el árbol, la función debe devolver 0.

Ejercicio 3: Ordenar una pila utilizando un árbol binario de búsqueda

En este ejercicio, vamos a implementar un procedimiento que permita ordenar una pila de caracteres utilizando un árbol binario de búsqueda. Este ejercicio combina el uso de estructuras de datos como pilas y árboles binarios de búsqueda para resolver un problema práctico.

Para realizar este ejercicio, utilizaremos los siguientes ficheros:

- `uPilaChar.pas`: Unidad que contiene la implementación de una pila de caracteres con las operaciones básicas como `push`, `pop`, `peek`, y otras funciones auxiliares.
- `uBinaryCharSearchTree.pas`: Unidad que contiene la implementación de un árbol binario de búsqueda con caracteres como información en los nodos. Incluye métodos básicos como `add`, `remove`, y recorridos del árbol.
- `arboles_ej3.pas`: Programa principal que ejecutaremos para comprobar que hemos implementado correctamente el procedimiento de ordenación de la pila.

El objetivo es implementar un procedimiento llamado `ordenar_pila` que ordene los elementos de una pila de caracteres en orden ascendente utilizando un árbol binario de búsqueda como estructura auxiliar.

Pasos a realizar

1. Modifica `uBinaryCharSearchTree.pas` para que almacene caracteres en los nodos del árbol binario de búsqueda. Asegúrate de que el árbol pueda manejar correctamente la inserción y eliminación de caracteres.

2. Crea un procedimiento `ordenar_pila` en `uBinaryCharSearchTree.pas` que realice lo siguiente:

    - **Crear un árbol binario de búsqueda**: Inicializar un árbol binario de búsqueda vacío.
    - **Desapilar los elementos de la pila**: Extraer los elementos de la pila uno por uno y añadirlos al árbol binario de búsqueda utilizando el método `add`.
    - **Recorrer el árbol en orden**: Realizar un recorrido in-orden del árbol binario de búsqueda para obtener los elementos en orden ascendente. Durante este recorrido, volver a apilar los elementos en la pila. Para ello, deberás implementar un procedimiento auxiliar que recorra el árbol y apile los elementos en el orden correcto.

3. **Reapilar los elementos**: Una vez que el recorrido in-orden ha terminado, la pila contendrá los elementos en orden ascendente.

Ejemplo de ejecución:

Supongamos que la pila inicial contiene los elementos: `d`, `a`, `c`, `b` (en este orden, con `d` en la cima).

1. Antes de ordenar: `d a c b`
2. Después de ordenar: `a b c d`

El programa principal debe mostrar estos resultados para verificar que el procedimiento funciona correctamente.

Ejercicio 4: TreeMap

En este ejercicio, vamos a implementar un árbol binario de búsqueda que almacene pares clave-valor. La clave será un número entero y el valor será una lista de palabras.

Para realizar este ejercicio, utilizaremos los siguientes ficheros:

- `uTreeMap.pas`: unidad que contiene la definición del árbol binario de búsqueda. Deberemos implementar los métodos en esta unidad.
- `uListaSimple.pas`: unidad que contiene la definición de una lista simple. Deberemos implementar los métodos en esta unidad.
- `arboles_ej4.pas`: programa principal que ejecutaremos para comprobar que hemos implementado correctamente el ejercicio.

La estructura del árbol debe permitir realizar las siguientes operaciones:

Métodos a implementar:

1. Lo primero que debes hacer es definir la estructura del TreeMap basándote en la definición del árbol binario de búsqueda. La estructura del árbol debe ser similar a la siguiente:

```
type
  tTreeMap = ^tnode;
  tnode = record
    key: integer;
    value: tListaSimple;
    hi, hd: tTreeMap;
  end;
```

2. `procedure add(var a: tTreeMap; key: integer; value: string);`: Añade un par clave-valor al árbol. Si la clave ya existe, el valor se añade a la lista asociada a esa clave. Si la clave no existe, se crea un nuevo nodo con la clave y el valor. Puede haber valores duplicados en la lista asociada a una clave. Por ejemplo, si se añade la clave `5` con el valor `"hola"` y luego se añade nuevamente la clave `5` con el valor `"mundo"`, el árbol tendrá un nodo con la clave `5` y una lista que contiene `"hola"` y `"mundo"`.

3. `procedure get(a: tTreeMap; key: integer; var value: tListaSimple);`: Recupera la lista de palabras asociada a una clave específica. Si la clave no existe, la lista devuelta estará vacía.

4. `function contains(a: tTreeMap; key: integer): boolean;`: Comprueba si una clave específica existe en el árbol. Devuelve `true` si la clave está presente y `false` en caso contrario.

5. `procedure remove(var a: tTreeMap; x: integer);`: Elimina un nodo del árbol basado en la clave proporcionada. Si la clave no existe, no se realiza ninguna acción. Si la clave existe y tiene una lista asociada, se eliminará la lista completamente.

6. `procedure remove_value(var a: tTreeMap; x: integer; value: string);`: Elimina un valor específico de la lista asociada a una clave. Si la lista queda vacía después de eliminar el valor, el

nodo correspondiente también se elimina del árbol.