Cuaderno de problemas
Estructuras de Datos. Árboles binarios

Prof. Isabel Pita

14 de marzo de 2022

Índice

1. Suma de los valores de un árbol

¿Cuánto vale el árbol?

Dado un árbol binario de enteros se pide calcular la suma de todos los valores de los nodos del árbol.
Requisitos de implementación.

En la implementación del problema, se utilizará el TAD `bintree` que se encuentra en el campus virtual. Se realizarán dos soluciones del problema: en la primera se implementará un método público sin parámetros en una clase que herede de la clase `bintree`, que llamará a un método recursivo privado que recibirá como parámetro la raíz del árbol; en la segunda se implementará una función externa a la clase que recibirá como parámetro un árbol binario.

Entrada

La entrada comienza con el número de casos de prueba. Cada caso se describe en una linea que muestra el recorrido en preorden del árbol incluyendo los punteros a nulo.

El número de nodos de un árbol será menor de 200. Los valores de los nodos serán números del 1 al 100. Los punteros a nulo de las hojas se representan con el valor -1.

Salida

Para cada caso de prueba se escribirá en una línea la suma de los valores de todos los nodos del árbol.

Entrada de ejemplo

```
4
-1
2 -1 -1
1 1 4 -1 -1 7 -1 -1 8 1 -1 -1 4 -1 -1
3 -1 4 -1 2 6 -1 -1 2 -1 3 -1 -1
```

Salida de ejemplo

```
0
2
26
20
```

Autor: Isabel Pita

1.1. Objetivos del problema

- Lectura de los datos de un árbol binario dados en preorden.
- Recorrido recursivo de un árbol visitando todos los nodos.
- Diferenciar entre resolver el problema con un método de la clase y resolverlo con una función externa a la clase.

1.2. Ideas generales.

- En la entrada de datos el árbol se describe mediante su recorrido en *preorden* (ver transparencias 16 y 17 del tema). En primer lugar se muestra el contenido de la raíz del árbol, seguido de la descripción del hijo izquierdo (en preorden) y seguido de la descripción del hijo derecho (en preorden). Para construir el árbol binario a partir de la entrada, utilizaremos la función `leerArbol` que se encuentra al final del fichero `bintree_eda.h`. Esta función tiene como parámetro el valor que representa el árbol vacío.
- Este tipo de problemas se pueden resolver de dos formas distintas: con una función externa a la clase utilizando las funciones públicas de la clase `bintree` o con un método implementado en una clase que herede de la clase `bintree`.
- Primera solución: implementación de una función externa a la clase que resuelve el problema. Se trata de una función recursiva que recibe como parámetro un árbol binario de la clase `bintree`. La función es genérica en el tipo de elementos que se almacenan en el árbol.
- Segunda solución: implementación de un método en una clase que herede de `bintree`. Para esta solución extendemos la clase `bintree` mediante herencia. Declaramos el `Link` para poder usarlo y definimos la función pública sin parámetros que resolverá el problema. La función pública hará uso de una función privada a la que pasaremos cómo parámetro un `Link` que representa la raíz del árbol que se está tratando. La función privada se implementa de forma recursiva.

  El árbol debe declararse del tipo de la clase que extiende a `bintree`. La función `leerArbol` implementada en el fichero devuelve un árbol de tipo `bintree`. Debemos implementar una función `leerArbol_ext` que devuelva un árbol del tipo `bintree_ext`. (Ver la implementación al final del problema)
- En ambas soluciones se recomienda:
  - Que la función recursiva devuelva los valores como resultado de la función. Cuando se deba devolver más de un valor utilizaremos un `pair`, o un struct para agruparlos y poder devolverlos todos. Solo devolveremos parámetros por referencia cuando sean del tipo vector, ya que devolverlos como resultado de la función realiza una copia del vector que es poco eficiente.
  - Al valor de retorno de la función se le debe dar valor en la función. Esto es importante. No se debe de dejar ningún campo sin dar valor, aunque lo tengan por defecto. Dar el valor explícitamente en cada caso a cada campo.
  - No se deben utilizar parámetros por referencia no constantes (es decir de salida) en las funciones recursivas, salvo que sean del tipo vector como se indicó antes. Si estos parámetros se llaman siempre con la misma variable son equivalentes a variables globales, ya que son posiciones de memoria que todas las llamadas recursivas pueden modificar. Este tipo de parámetros no están permitidos en la asignatura.

1.3. Ideas detalladas.

- En las dos soluciones la función recursiva tiene como caso base el árbol vacío. En la primera solución se comprueba la condición de árbol vacío con el método público `empty()`. En la segunda solución empleamos el link a la raíz del subárbol que se está tratando en la llamada recursiva, comprobando si este link que es un puntero a un nodo tiene el valor `nullptr`.

- En la parte recursiva de la función, realizaremos dos llamadas recursivas, una que calcula la suma de los valores de la parte izquierda del árbol y otra que calcula la suma de los valores de la parte derecha del árbol. En la primera solución los subárboles izquierdo y derecho se obtienen con los métodos `left()` y `right()`. En la segunda solución accedemos a cada subárbol a través de los link `left` y `right` del nodo. En esta solución, al estar implementando un método que hereda de la clase `bintree` tenemos acceso a los atributos protegidos de la clase.

- A los valores que devuelven las llamadas recursivas con la suma de los valores de los subárboles izquierdo y derecho debemos sumar el valor de la raíz del subárbol que está tratando la llamada recursiva. En la primera solución lo obtenemos con el método público `root()`. En la segunda solución se obtiene el valor de la raíz a través del link a la raíz del subárbol que se pasa como parámetro `r->elem`.

1.4. Coste de la solución

Para justificar el coste de un algoritmo recursivo, lo mejor es plantear la recurrencia que define el problema como se explica en la asignatura de FAL.

Para plantear la recurrencia debemos saber el coste de resolver el caso base y el coste de resolver el caso recursivo. Por ejemplo, en el algoritmo que suma los valores de todos los nodos de un árbol:

```
template <class T>
int sumaNodos(bintree<T> const& arbol) { // O(N)
    if (arbol.empty()) {
        return 0;
    } else {
        int res_iz = sumaNodos(arbol.left());
        int res_dr = sumaNodos(arbol.right());
        return { res_iz + res_dr + arbol.root() };
    }
}
```

La ecuación de recurrencia que permite calcular el coste de este algoritmo es:

$$T(n) = \begin{cases} c_0 & \text{si } a.empty() \\ T(n_i) + T(n_d) + c_1 & \text{si } !a.empty() \end{cases}$$

donde n_i representa el número de nodos del árbol izquierdo y n_d el número de nodos del árbol derecho.

Podemos resolver esta recurrencia suponiendo el árbol degenerado, es decir que todos los nodos cumplen que uno de sus hijos es vacío. En este caso la recurrencia es:

$$T(n) = \begin{cases} c_0 & \text{si } a.empty() \\ T(n-1) + c_1 & \text{si } !a.empty() \end{cases}$$

donde n representa el número de nodos del árbol.

También se puede resolver para un árbol equilibrado. En este caso la recurrencia es:

$$T(n) = \begin{cases} c_0 & \text{si } a.empty() \\ T(n/2) + T(n/2) + c_1 & \text{si } !a.empty() \end{cases}$$

donde n representa el número de nodos del árbol.

En ambos casos la solución de la recurrencia es $\mathcal{O}(n)$, esto es un orden lineal respecto al número de nodos del árbol.

En los casos sencillos en que se recorre todo el árbol y el coste de tratar cada nodo es constante, se puede justificar el coste indicando que *el algoritmo trata una vez cada nodo del árbol y el coste de tratar cada nodo es constante, por lo tanto el coste del algoritmo es lineal en el número de nodos del árbol.*

1.5. Problemas semejantes.

- Calcular el número de nodos, hojas y altura de un arbol binario. El problema se resuelve implementando tres funciones o tres métodos diferentes. Cada método calcula una de las cosas pedidas. Cada función solo calcula un valor que devolverá como resultado.

La función que calcula el número de hojas debe tener dos casos base. Uno cuando el subárbol es vacío y otro cuando es una hoja. Esto es necesario porque hay nodos que tienen un solo hijo y dan lugar a una llamada recursiva con un hijo vacío.

- Calcular la frontera de un árbol. La frontera es la lista de las hojas del árbol de izquierda a derecha. La función será de tipo `void` y tendremos un parámetro de tipo vector, por referencia en el que formaremos la frontera. Cuando se encuentra una hoja se añade al vector por la derecha.

- Calcular el mínimo de un árbol. En este caso el árbol puede ser de números enteros o de cadenas de caracteres. En la función `resuelveCaso` se diferenciará entre un tipo u otro, para llamar a la función `leerArbol` con un carácter vacío u otro. La función debe devolver únicamente el valor del mínimo.

1.6. Implementación.

- Implementación con función externa.

```
template <class T>
int sumaNodos(bintree<T> const& arbol) { // O(N)
    if (arbol.empty()) {
        return 0;
    } else {
        int res_iz = sumaNodos(arbol.left());
        int res_dr = sumaNodos(arbol.right());
        return { res_iz + res_dr + arbol.root() };
    }
}

void resuelveCaso() {
    bintree<int> arbol = leerArbol(-1);
    std::cout << sumaNodos(arbol) << '\n';
}

int main() {
#ifndef DOMJUDGE
    std::ifstream in("datos.txt");
    auto cinbuf = std::cin.rdbuf(in.rdbuf()); //save old buf and redirect std::cin to casos.
#endif

    int numCasos = 1;   std::cin >> numCasos;
    for (int i = 0; i < numCasos; ++i) resuelveCaso();


#ifndef DOMJUDGE // para dejar todo como estaba al principio
    std::cin.rdbuf(cinbuf);
    system("PAUSE");
#endif

    return 0;
}
```

- Implementación con método de una clase que hereda de `bintree`.

```
template <class T>
class bintree_ext: public bintree<T> {
    using Link = typename bintree<T>::Link;
public:
    bintree_ext() : bintree<T>() {}
    bintree_ext(bintree_ext<T> const& l, T const& e, bintree_ext<T> const& r) :
          bintree<T>(l,e,r){}
    int sumaNodos()
    {
```

```
        return sumaNodos(this->raiz);
    }
private:
    int sumaNodos(Link r) {
        if (r == nullptr) { return 0;}
        else {
            int Izda = sumaNodos(r->left);
            int Drch = sumaNodos(r->right);
            return {Izda + Drch + r->elem};
        }
    }
};

template <typename T>
inline bintree_ext<T> leerArbol_ext(T vacio) {
    T raiz;
    std::cin >> raiz;
    if (raiz == vacio) { // es un arbol vacio
        return {};
    } else { // leer recursivamente los hijos
        bintree_ext<T> iz = leerArbol_ext(vacio);
        bintree_ext<T> dr = leerArbol_ext(vacio);
        return {iz, raiz, dr};
    }
}

void resuelveCaso() {
    bintree_ext<int> arbol = leerArbol_ext(-1);
    std::cout << sumaNodos(arbol) << '\n';
}

int main() {
#ifndef DOMJUDGE
    std::ifstream in("datos.txt");
    auto cinbuf = std::cin.rdbuf(in.rdbuf()); //save old buf and redirect std::cin to casos.
#endif

    int numCasos = 1;   std::cin >> numCasos;
    for (int i = 0; i < numCasos; ++i) resuelveCaso();


#ifndef DOMJUDGE // para dejar todo como estaba al principio
    std::cin.rdbuf(cinbuf);
    system("PAUSE");
#endif

    return 0;
}

int main() {
    // Para la entrada por fichero.
#ifndef DOMJUDGE
    std::ifstream in("datos1.txt");
    auto cinbuf = std::cin.rdbuf(in.rdbuf()); //save old buf and redirect std::cin to casos.
#endif


    int numCasos;
    std::cin >> numCasos;
    for (int i = 0; i < numCasos; ++i)
    {resuelveCaso(); std::cin.ignore();}
```

```
#ifndef DOMJUDGE // para dejar todo como estaba al principio
    std::cin.rdbuf(cinbuf);
    system("PAUSE");
#endif

    return 0;
}
```

2. Comprobar si un árbol binario está equilibrado

Árbol equilibrado

Existen distintas condiciones de equilibrio para los árboles binarios. En este problema diremos que un árbol es equilibrado si es vacío o si son equilibrados su hijo izquierdo, y su hijo derecho y además la diferencia de altura entre sus dos hijos es menor o igual que 1.

Dado un árbol binario de enteros se pide comprobar si es equilibrado.

Requisitos de implementación.

En la implementación del problema, se utilizará el TAD `bintree` que se encuentra en el campus virtual. Se realizarán dos soluciones del problema: en la primera se implementará un método sin parámetros en una clase que herede de la clase `bintree` que llamará a un método recursivo privado que recibirá como parámetro la raíz del árbol; en la segunda se implementará una función externa a la clase que recibirá como parámetro un árbol binario.

Entrada

La entrada comienza indicando el número de casos de prueba que vendrán a continuación. Cada caso consiste en una cadena de caracteres con la descripción de un árbol binario (correspondiente al recorrido en preorden): el árbol vacío se representa con un punto (`.`); un árbol no vacío se representa con un `*` (que denota la raíz), seguido primero de la descripción del hijo izquierdo y después de la descripción del hijo derecho.

Salida

Para cada caso de prueba se escribirá `SI` si el árbol es equilibrado y `NO` si no es equilibrado.

Entrada de ejemplo

```
3
***..*..*.*..
**.*..**..*.*..
**..**..*.*..
```

Salida de ejemplo

```
SI
SI
NO
```

Autor: Isabel Pita

2.1. Objetivos del problema

- Cómo obtener la información necesaria para resolver el problema al recorrer el árbol, para que la solución sea eficiente.

2.2. Ideas generales.

- En la entrada de datos el árbol se describe mediante su recorrido en *preorden*. Ver solución al problema que suma los valores de un árbol. Para construir el árbol binario a partir de la entrada, utilizaremos la función `leerArbol` que se encuentra al final del fichero `bintree_eda.h`. Esta función tiene como parámetro el valor que representa el árbol vacío.

- Un árbol está equilibrado, si es un árbol vacío o si están equilibrados sus dos subárboles y además la diferencia de altura entre los dos subárboles no es mayor que uno. Por lo tanto para ver si el árbol está equilibrado tenemos que saber la altura de los subárboles. La altura debe calcularse al mismo tiempo que se comprueba si el árbol es equilibrado. La función `equilibrado` por lo tanto, debe calcular dos cosas, por un lado la altura del árbol y por otro si este está equilibrado.

  Se recomienda:

  - Que la función recursiva devuelva los valores como resultado de la función. Cuando se deba devolver más de un valor utilizaremos un `pair`, o un struct para agruparlos y poder devolverlos todos. Sólo devolveremos parámetros por referencia cuando sean del tipo vector, ya que devolverlos como resultado de la función realiza una copia del vector que es poco eficiente.
  - Al valor de retorno de la función se le debe dar valor en la función. Esto es importante. No se debe de dejar ningún campo sin dar valor, aunque lo tengan por defecto. Dar el valor explícitamente en cada caso a cada campo.
  - No se deben utilizar parámetros por referencia no constantes (es decir de salida) en las funciones recursivas, salvo que sean del tipo vector como se indicó antes. Si estos parámetros se llaman siempre con la misma variable son equivalentes a variables globales, ya que son posiciones de memoria que todas las llamadas recursivas pueden modificar. Este tipo de parámetros no están permitidos en la asignatura.

2.3. Ideas detalladas.

- Para implementar la función debemos resolver el caso base y el caso recursivo.

- En el caso base el árbol está equilibrado y la altura es cero.

- En el caso recursivo se hacen las dos llamadas recursivas, una al hijo izquierdo y otra al hijo derecho. La solución de la función se calcula a partir de las soluciones de estas dos llamadas, que nos indican la altura de cada subárbol y si el subárbol está equilibrado. La altura del árbol de la llamada actual es el máximo de la altura del hijo izquierdo y la altura del hijo derecho mas 1, que es la raíz del árbol. Y el árbol está equilibrado si lo está el hijo izquierdo y lo está el hijo derecho y el valor absoluto de la diferencia de las alturas es menor o igual que 1. (Ver implementación al final del problema)

2.4. Errores frecuentes.

- Utilizar una función que calcule la altura en cada llamada recursiva. Esta solución recorre el árbol completo para calcular su altura en cada llamada y por lo tanto su coste es cuadrático respecto al número de nodos del árbol.

2.5. Coste de la solución.

- En la solución planteada para el problema y cuya implementación se encuentra al final de la sección, tenemos la misma recurrencia que para el problema que suma los valores de los nodos de un árbol y por lo tanto el coste es lineal en el número de nodos del árbol.

$$T(n) = \begin{cases} c_0 & \text{si } a.empty() \\ T(n_i) + T(n_d) + c_1 & \text{si } !a.empty() \end{cases}$$

- Sin embargo, si se realizase una implementación que llama a una función altura para calcular la altura del árbol en cada llamada recursiva, la recurrencia que permite calcular el coste de la supuesta implementación sería:

$$T(n) = \begin{cases} c_0 & \text{si } a.empty() \\ T(n_i) + T(n_d) + n_i + n_d & \text{si } !a.empty() \end{cases}$$

  En este caso la solución de esta recurrencia pertenece a $O(n^2)$ si el árbol es degenerado y a $O(n \log n)$ si el árbol es equilibrado, siendo n el número de nodos del árbol.

2.6. Problemas semejantes en el juez.

- Diámetro de un árbol. La longitud del camino mas largo que pasa por un nodo se calcula a partir de la altura de los subárboles izquierdo y derecho del nodo. El diámetro es el máximo de todos los caminos. En este caso, al igual que ocurre en el problema del árbol equilibrado la función debe calcular dos valores: el diámetro del subárbol que se está considerando y la altura de este subárbol. De forma que cada nodo al hacer la llamada a su subárbol izquierdo y a su subárbol derecho obtiene el diámetro de cada uno de ellos y su altura. Con estos datos puede calcular el diámetro del árbol (el máximo entre el diámetro del hijo izquierdo, el diámetro del hijo derecho y la suma de las alturas de los hijos izquierdo y derecho y el propio nodo) y su altura (máximo de las alturas de los hijos más uno del propio nodo).

- Excursionistas atrapados. Este problema se resolverá solamente con una función externa a la clase. Para resolverlo debéis pensar la información que es necesaria que la llamada recusiva proporcione para resolver el problema. En un árbol, al hacer la llamada al subárbol izquierdo y al subárbol derecho podemos obtener información sobre los excursionistas atrapados en cada uno de los subárboles. Cómo en el caso de los árboles equilibrados hace falta más de un valor en cada caso.

- Árboles zurdos. Se definen los árboles zurdos como los árboles vacíos o aquellos árboles que tienen más de la mitad de sus descendientes en el hijo izquierdo y a su vez los hijos son árboles zurdos. Se debe pensar en la información que es necesaria conocer de cada hijo para poder calcular si el árbol es zurdo.

- Subárbol completo más grande. El subárbol completo más grande que hay en un árbol puede encontrarse en cualquier lugar del árbol. Es decir, puede estar en el hijo izquierdo, puede estar en el hijo derecho, o puede formarse con la raíz del árbol, tomando el mismo número de nodos de cada hijo. Debe pensarse, como en los problemas anteriores, la información que cada hijo debe devolver. Observar que es necesario conocer la altura del subárbol completo máximo que se haya encontrado y también la longitud de la rama más corta. Esta última información hace falta para calcular la altura del árbol completo que se forma a partir de la raíz del árbol.

2.7. Implementación.

- Implementación con función externa.

```
struct tSol {
    bool eq;
    int altura;
};

template <class T>
tSol equilibrado(bintree<T> const& arbol) { // O(N)
    if (arbol.empty()) {
        return { true, 0 };
    } else {
        tSol res_iz = equilibrado(arbol.left());
```

```
        tSol res_dr = equilibrado(arbol.right());
        return { res_iz.eq && res_dr.eq &&
            abs(res_iz.altura - res_dr.altura) < 2,
            std::max(res_iz.altura, res_dr.altura) + 1 };
    }
}

void resuelveCaso() {
    bintree<char> arbol = leerArbol('.');
    if (equilibrado(arbol).eq) std::cout << "SI\n";
    else std::cout << "NO\n";
}

int main() {
    // Para la entrada por fichero.
    // Comentar para acepta el reto
#ifndef DOMJUDGE
    std::ifstream in("datos.txt");
    auto cinbuf = std::cin.rdbuf(in.rdbuf()); //save old buf and redirect std::cin to casos.
#endif


    int numCasos;
    std::cin >> numCasos;
    for (int i = 0; i < numCasos; ++i)
    {resuelveCaso(); std::cin.ignore();}

    // Para restablecer entrada. Comentar para acepta el reto
#ifndef DOMJUDGE // para dejar todo como estaba al principio
    std::cin.rdbuf(cinbuf);
    system("PAUSE");
#endif

    return 0;
}
```

- Implementación con método de una clase que hereda de `bintree`.

```
template <class T>
class bintree_ext: public bintree<T> {
    using Link = typename bintree<T>::Link;
public:
    bintree_ext() : bintree<T>() {}
    bintree_ext(bintree_ext<T> const& l, T const& e, bintree_ext<T> const& r) :
             bintree<T>(l,e,r){}
    bool equilibrado()
    {
        return equilibrado(this->raiz).eq;
    }
private:
  struct tSol {
    bool eq;
    int altura;
  };
    tSol equilibrado(Link r) {
        if (r == nullptr) { return {true,0};}
        else {
            tSol Izda = equilibrado(r->left);
            tSol Drch = equilibrado(r->right);
            return {Izda.eq && Drch.eq &&
                 std::abs(Izda.altura-Drch.altura) <= 1,
                 std::max(Izda.altura, Drch.altura) +1};
```

```
        }
    }
};

template <typename T>
inline bintree_ext<T> leerArbol_ext(T vacio) {
    T raiz;
    std::cin >> raiz;
    if (raiz == vacio) { // es un arbol vacio
        return {};
    } else { // leer recursivamente los hijos
        bintree_ext<T> iz = leerArbol_ext(vacio);
        bintree_ext<T> dr = leerArbol_ext(vacio);
        return {iz, raiz, dr};
    }
}


void resuelveCaso() {
    bintree_ext<char> arbol = leerArbol_ext('.');
    if (arbol.equilibrado()) std::cout << "SI\n";
    else std::cout << "NO\n";
}

int main() {
    // Para la entrada por fichero.
#ifndef DOMJUDGE
    std::ifstream in("datos.txt");
    auto cinbuf = std::cin.rdbuf(in.rdbuf()); //save old buf and redirect std::cin to casos.
#endif


    int numCasos;
    std::cin >> numCasos;
    for (int i = 0; i < numCasos; ++i)
    {resuelveCaso(); std::cin.ignore();}

#ifndef DOMJUDGE // para dejar todo como estaba al principio
    std::cin.rdbuf(cinbuf);
    system("PAUSE");
#endif

    return 0;
}
```

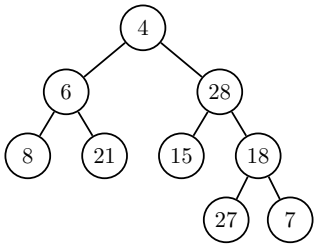
3. Buscar el nodo más cercano a la raíz que cumple una propiedad

Buscar el nodo más cercano a la raíz que cumple una propiedad

Dado un árbol binario estamos interesados en buscar el nodo más cercano a la raíz que cumpla una determinada propiedad. Si hay varios nodos en el mismo nivel que cumplen la propiedad se devolverá el que se encuentre más a la izquierda.

El problema se aplicará sobre árboles de números enteros positivos y sobre árboles de caracteres. En el primer caso buscaremos el nodo más cercano a la raíz que sea un valor impar. En el segundo caso buscaremos el nodo más cercano a la raíz que sea una vocal (en mayúsculas o minúsculas).

Por ejemplo, en el siguiente árbol el número buscado es el 21, que se encuentra a profundidad 3.



Entrada

Cada caso de prueba ocupa dos líneas. En la primera aparecerá una `N` si el árbol es de números enteros positivos, o una `C` si el árbol es de caracteres. En la segunda línea aparecerá la descripción del árbol: primero la raíz, después la descripción del hijo izquierdo y después la descripción del hijo derecho. Si el árbol es de números, se utilizará `-1` para indicar el árbol vacío; si es de caracteres, se utilizará `#`.

Salida

Para cada árbol se escribirá el valor del nodo accesible lo más cercano a la raíz (y en caso de empate el colocado más a la izquierda) seguido del nivel en el que se encuentra. En caso de que no haya ninguno se escribirá `NO HAY`.

Entrada de ejemplo

```
N
2 4 1 -1 -1 3 -1 -1 2 -1 -1
C
r l M # # C # # s # #
C
k # 2 # @ # G # A # e # #
```

Salida de ejemplo

```
1 3
NO HAY
A 5
```

Autor: Isabel Pita

3.1. Objetivos del problema

- Practicar el recorrido en anchura de un árbol.
- Practicar las funciones genéricas.

3.2. Ideas generales.

- En el problema nos piden calcular el nodo más cercano a la raíz que cumple una propiedad. Este tipo de problemas se resuelven fácilmente utilizando un recorrido en anchura del árbol. El motivo es que el recorrido en anchura va comprobando la propiedad en cada nivel del árbol, por lo tanto el primer nodo que cumpla la propiedad será el nodo más cercano a la raíz que la cumple.
- El recorrido en anchura se realiza con un algoritmo iterativo que utiliza una cola para almacenar los subárboles que todavía no se han recorrido. Para saber el nivel en que se encuentra cada subárbol, se utilizará una cola de pares, en la que el primer elemento del par sea el nivel y el segundo elemento del par el subárbol que todavía no se ha recorrido

3.3. Ideas detalladas.

- Inicialmente se añade a la cola la raíz del árbol.
- En cada vuelta del bucle se obtiene el primer elemento de la cola, que es el subárbol que se debe tratar. Si el valor de la raíz cumple la propiedad hemos terminado. En otro caso si los subárboles izquierdo y derecho no son vacíos se añadirán a la cola en ese orden.
- Los árboles vacíos no se añaden a la cola para evitar que esta crezca innecesariamente.

3.4. Algunas cuestiones sobre implementación.

- La función que resuelve el problema es genérica en el tipo de elementos del árbol y también en la propiedad que se quiere probar.
- Se crean objetos función para cada una de las propiedades que se quieren probar, y se llamará a la función que resuelve el problema con estos objetos.
- La función devuelve un valor booleano que indica si se ha encontrado algún elemento que cumpla la propiedad y un par con el valor encontrado y la profundidad a la que se encuentra.
- Para evitar realizar copias de los árboles se utilizan referencias para acceder más fácilmente a algunos datos. Se observa que se declara una referencia `par` al primer elemento de la cola y otra referencia `a` al segundo elemento del par que es el árbol binario:

```
auto const& par = cola.front();
auto const& a = par.second;
```

- Con la referencia `par` evitamos llamar al método `front()` cada vez que queramos acceder a su información, al tiempo que evitamos realizar una copia del mismo en una variable estática.
- Con la referencia `a` accedemos más fácilmente a los datos. Asi en lugar de escribir `par.second.root()` escribimos simplemente `a.root()`.

3.5. Coste de la solución

- Se trata de un algoritmo iterativo, por lo que estudiamos el coste de cada instrucción.
- Tenemos un bucle que se ejecuta mientras la cola no sea vacía. Observamos que en el caso peor todos los nodos del árbol, pasan una vez por la cola, por lo tanto el número de vueltas que da el bucle coincide, en el caso peor, con el número de nodos del árbol.

- En cada vuelta del bucle se obtiene el primer elemento de la cola y se elimina. Luego se añaden dos elementos a la cola. En cuanto a los árboles, se utilizan las operaciones `root()`, `left()` t `right()` . También se comprueba la propiedad que se quiere comprobar sobre los elementos. Como todas las operaciones de las colas y de los árboles binarios tienen coste constante, el coste de cada vuelta del bucle será el coste de comprobar la propiedad, que en este problema es constante.

3.6. Implementación.

- Implementación con función externa.

```
#include "bintree_eda.h"

class par {
public:
    bool operator() (int n) {
        return n%2 == 1;
    }
};

class vocal {
public:
    bool operator() (char c) {
        c = tolower(c);
        return c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u';
    }
};

template <typename T, typename P>
std::pair<bool,std::pair<T,int>> resolver (bintree<T> const& arbol, P propiedad) {
    bool encontrado = false;
    T nodo{}; int nivel = 0;

    if (arbol.empty()) return {false, {}};
    else if (arbol.left().empty() && arbol.right().empty())
        return {propiedad(arbol.root()),{arbol.root(),1}};

    std::queue<std::pair<int,bintree<T>>> cola;
    cola.push({1,arbol});

    // en la cola solamente guardamos arboles no vacios
    while (!cola.empty() && !encontrado) {
        auto const& par = cola.front();
        auto const& a = par.second;
        if (propiedad(a.root())) {
            encontrado = true; nodo = a.root(); nivel = par.first;
        }
        else {
            if (!a.left().empty())
                cola.push({par.first+1, a.left()});
            if (!a.right().empty())
                cola.push({par.first+1, a.right()});
        }
         cola.pop();
    }
    return {encontrado,{nodo, nivel}};
}

bool resuelveCaso() {
    char letra;
    std::cin >> letra;
    if (!std::cin) return false;
    if (letra == 'N'){
```

```
        auto arbol = leerArbol(-1);
        auto p = resolver(arbol,par());
        if (p.first)
            std::cout << p.second.first << ' ' << p.second.second << '\n';
        else
            std::cout << "NO HAY\n";
    }
    else {
        auto arbol = leerArbol('#');
        auto p = resolver(arbol,vocal());
        if (p.first)
            std::cout << p.second.first << ' ' << p.second.second << '\n';
        else
            std::cout << "NO HAY\n";
    }
    return true;
}

int main() {

#ifndef DOMJUDGE
    std::ifstream in("datos.txt");
    auto cinbuf = std::cin.rdbuf(in.rdbuf()); //save old buf and redirect std::cin to casos.
#endif


    while (resuelveCaso()) {} //Resolvemos todos los casos


#ifndef DOMJUDGE // para dejar todo como estaba al principio
    std::cin.rdbuf(cinbuf);
    system("PAUSE");
#endif
 *
    return 0;
}
```