Todas-las-practicas-Arboles-con-...

Warabaringo

Estructuras de Datos no Lineales

2º Grado en Ingeniería Informática

Escuela Superior de Ingeniería
Universidad de Cádiz

Árboles EDNL

Warabaringo

Índice

Práctica 1

Ejercicio 1

Implementa un subprograma que calcule el número de nodos de un árbol binario

```cpp
#ifndef NUM_NODOS_HPP
#define NUM_NODOS_HPP

#include "../Abin/Abin_enla.hpp"

template <typename T>
int num_nodos(const Abin<T>& A) {
        return num_nodos_rec(A.raiz(), A);
}
template <typename T>
int num_nodos_rec(typename Abin<T>::nodo n, const Abin<T>&
 ↪  A) {
        if (n == Abin<T>::NODO_NULO)
                return 0;
        else
                return 1 + num_nodos_rec(A.hijoIzqdo(n), A)
                 ↪  + num_nodos_rec(A.hijoDrcho(n), A);
}

#endif // NUM_NODOS_HPP
```

Ejercicio 2

Implementa un subprograma que calcule la altura de un árbol binario.

```
#ifndef ALTURA_HPP
#define ALTURA_HPP

#include "../Abin/Abin_enla.hpp"

template <typename T>
int calcular_altura(const Abin<T>& A) {
        return calcular_altura_rec(A.raiz(), A);
}

template <typename T>
int calcular_altura_rec(typename Abin<T>::nodo n, const
↪  Abin<T>& A) {
        if (n == Abin<T>::NODO_NULO)
                return -1;
        else
                return 1 +
                ↪  std::max(calcular_altura_rec(A.hijoIzqdo(n),
                ↪  A), calcular_altura_rec(A.hijoDrcho(n),
                ↪  A));
}


#endif // ALTURA_HPP
```

Ejercicio 3

Implementa un subprograma que, dados un árbol binario y un nodo del mismo, determine la profundidad de este nodo en dicho árbol.

```
#ifndef PROFUNDIDAD_NODO_HPP
#define PROFUNDIDAD_NODO_HPP

#include "../Abin/Abin_enla.hpp"
#include <cassert>

template <typename T>
int calcular_profundidad_nodo(const Abin<T> &A, typename
 ↪  Abin<T>::nodo n) {
        assert(!A.arbolVacio() and n != Abin<T>::NODO_NULO);
        typename Abin<T>::nodo it = n;
        int profundidad = -1;
        while(it != Abin<T>::NODO_NULO) {
                it = A.padre(it);
                profundidad++;
        }
        return profundidad;
}


#endif // PROFUNDIDAD_NODO_HPP
```

Gemini

Lleva tu estudio al siguiente nivel con Gemini, tu asistente de IA de Google ✨

¡Pruébalo ahora!

Ejercicio 4

Añade dos nuevas operaciones al TAD árbol binario, una que calcule la profundidad de un nodo y otra que calcule la altura de un nodo en un árbol dado. Implementa esta operación para la representación vectorial (índices del padre, hijo izquierdo e hijo derecho).

```
1  template <typename T>
2  inline unsigned Abin<T>::profundidad(nodo n) const {
3          assert(n >= 0 && n < numNodos);
4          unsigned prof = 0;
5          while (n != 0) {
6                  n = nodos[n].padre;
7                  ++prof;
8          }
9          return prof;
10 }
11
12 template <typename T>
13 inline unsigned Abin<T>::altura(nodo n) const {
14         if (n == NODO_NULO)
15                 return -1;
16         else
17                 return 1 + std::max(altura(nodos[n].hizq),
                    ↪  altura(nodos[n].hder));
18 }
```

Ejercicio 5

Repite el ejercicio anterior para la representación enlazada de árboles binarios (punteros al padre, hijo izquierdo e hijo derecho).

```
1  template <typename T>
2  inline int Abin<T>::profundidad(nodo n) const {
3          int prof = 0;
4          while (n != r) {
5                  n = n->padre;
6                  ++prof;
7          }
8          return prof;
9  }
10
11 template <typename T>
12 inline int Abin<T>::altura(nodo n) const {
13         if (n == NODO_NULO)
14                 return -1;
15         else
16                 return 1 + std::max(altura(n->hizq),
                    ↪  altura(n->hder));
17 }
```

Ejercicio 6

Implementa un subprograma que determine el nivel de desequilibrio de un árbol binario, definido como el máximo desequilibrio de todos sus nodos. El desequilibrio de un nodo se define como la diferencia entre las alturas de los subárboles del mismo.

```
#ifndef DESEQUILIBRIO_HPP
#define DESEQUILIBRIO_HPP

// #include "../Abin/Abin_enla.hpp"
#include <cmath> // std::abs

template <typename T>
int desequilibrio(const Abin<T>& A) {
    return desequilibrio_rec(A.raiz(), A);
}
template <typename T>
int desequilibrio_rec(typename Abin<T>::nodo n, const
↪  Abin<T>& A) {
    if (n == Abin<T>::NODO_NULO)
        return 0;
    else
        return std::max(std::abs(A.altura(A.hijoIzqdo(n)) -
        ↪  A.altura(A.hijoDrcho(n))), std::max
        ↪  (desequilibrio_rec(A.hijoIzqdo(n), A),
        ↪  desequilibrio_rec(A.hijoDrcho(n), A)));
}

#endif // DESEQUILIBRIO_HPP
```

Ejercicio 7

Implementa un subprograma que determine si un árbol binario es o no pseudocompleto. En este problema entenderemos que un árbol es pseudocompleto, si en el penúltimo nivel del mismo cada uno de los nodos tiene dos hijos o ninguno.

```cpp
#ifndef PSEUDOCOMPLETO_HPP
#define PSEUDOCOMPLETO_HPP

#include "../Abin/Abin_enla.hpp"

// Versión propia

template <typename T>
bool pseudocompleto(const Abin<T>& A) {
    if (A.arbolVacio())
        return true;
    else {
        unsigned altura_max = A.altura(A.raiz());
        if(altura_max == 0)
            return true;
        else
            return pseudocompleto_rec(A.raiz(), A,
             ↪  altura_max, 0);
    }
}

template <typename T>
bool pseudocompleto_rec(typename Abin<T>::nodo n, const
 ↪  Abin<T> &A, unsigned altura_max, unsigned actual)
{
    if (n == Abin<T>::NODO_NULO)
        return true;
    else
    {
        if(actual == altura_max - 1) // Voy restando 1 a
         ↪  altura_max y acabo cuando valga 1
            return (A.hijoDrcho(n) != Abin<T>::NODO_NULO and
             ↪  A.hijoIzqdo(n) != Abin<T>::NODO_NULO) or
             ↪  (A.hijoDrcho(n) == Abin<T>::NODO_NULO and
             ↪  A.hijoIzqdo(n) == Abin<T>::NODO_NULO); // No
             ↪  debería valer solo con la primera?
```

```
        else
            return pseudocompleto_rec(A.hijoDrcho(n), A,
            ↪  altura_max, actual + 1) and
            ↪  pseudocompleto_rec(A.hijoIzqdo(n), A,
            ↪  altura_max, actual + 1);
    }
}

// Versión 1 (No óptima)

// template <typename T>
// bool pseudocompleto(const Abin<T> &A)
// {
//     if(A.arbolVacio() or A.altura(A.raiz()) == 0)
//         return true;
//     else
//         return pseudocompleto_rec(A.raiz(), A);
// }

// template <typename T>
// bool pseudocompleto_rec(typename Abin<T>::nodo n, const
↪  Abin<T> &A)
// {
//     if(A.altura(n) == 1)
//         return (A.hijoDrcho(n) != Abin<T>::NODO_NULO and
↪  A.hijoIzqdo(n) != Abin<T>::NODO_NULO)
//     else {
//         int altura_izq = A.altura(A.hijoIzqdo(n)),
↪  altura_der = A.altura(A.hijoDrcho(n));
//         if(altura_izq > altura_der)
//             return pseudocompleto_rec(A.hijoIzqdo(n),
↪  A);
//         else
//         {
//             if(altura_der > altura_izq)
//                 return
↪  pseudocompleto_rec(A.hijoDrcho(n), A);
//             else
//                 return
↪  pseudocompleto_rec(A.hijoDrcho(n), A) and
↪  pseudocompleto_rec(A.hijoDrcho, A);
```

```
61 //          }
62
63 //      }
64 // }
65
66
67
68 #endif // PSEUDOCOMPLETO_HPP
```

Práctica 2

Ejercicio 1

Dos árboles binarios son similares cuando tienen idéntica estructura de ramificación, es decir, ambos son vacíos, o en caso contrario, tienen subárboles izquierdo y derecho similares. Implementa un subprograma que determine si dos árboles binarios son similares.

```cpp
#ifndef SIMILARES_HPP
#define SIMILARES_HPP

#include "../Abin/Abin_enla.hpp"

template <typename T>
bool son_similares(const Abin<T>& A, const Abin<T>& B) {
  return son_similares_rec(A.raiz(), B.raiz(), A, B);
}

template <typename T>
bool son_similares_rec(typename Abin<T>::nodo a, typename
 ↪  Abin<T>::nodo b, const Abin<T> &A, const Abin<T> &B) {
    if(a == Abin<T>::NODO_NULO or b == Abin<T>::NODO_NULO)
        return a == Abin<T>::NODO_NULO and b ==
         ↪  Abin<T>::NODO_NULO;
    else
        return son_similares_rec(A.hijoIzqdo(a),
         ↪  B.hijoIzqdo(b), A, B) and
         ↪  son_similares_rec(A.hijoDrcho(a), B.hijoDrcho(b),
         ↪  A, B);
}

#endif // SIMILARES_HPP
```

Ejercicio 2

Para un árbol binario B, podemos construir el árbol binario reflejado BR cambiando los subárboles izquierdo y derecho en cada nodo. Implementa un subprograma que devuelva el árbol binario reflejado de uno dado.

```
#ifndef REFLEJAR_HPP
#define REFLEJAR_HPP

#include "../Abin/Abin_enla.hpp"


// Versión propia

// template <typename T>
// Abin<T> reflejar_arbol(const Abin<T> &B) {
//     Abin<T> BR;

//     if(!B.arbolVacio()) {
//         BR.insertarRaiz(B.elemento(B.raiz()));
//         reflejar_arbol_rec(B, BR, BR.raiz(),
 ↪  B.hijoIzqdo(B.raiz()), B.hijoDrcho(B.raiz()));
//     }
//     return BR;
// }

// template <typename T>
// void reflejar_arbol_rec(const Abin<T> &original, Abin<T>
 ↪  &reflejo, typename Abin<T>::nodo n, typename
 ↪  Abin<T>::nodo izq, typename Abin<T>::nodo der) {
//     if(izq != Abin<T>::NODO_NULO) {
//         reflejo.insertarHijoDrcho(n,
 ↪  original.elemento(izq));
//         n = reflejo.hijoDrcho(n);
//         reflejar_arbol_rec(original, reflejo, n,
 ↪  original.hijoIzqdo(izq), original.hijoDrcho(izq));
//     }
//     if(der != Abin<T>::NODO_NULO) {
//         reflejo.insertarHijoIzqdo(n,
 ↪  original.elemento(der));
//         n = reflejo.hijoIzqdo(n);
```

```
30  //         reflejar_arbol_rec(original, reflejo, n,
    ↪  original.hijoIzqdo(der), original.hijoDrcho(der));
31  //     }
32  // }
33
34
35  // DLH
36  template <typename T>
37  Abin<T> reflejar_arbol(const Abin<T> &B) {
38      Abin<T> reflejado;
39      if(!B.arbolVacio()) {
40          reflejado.insertarRaiz(B.elemento(B.raiz()));
41          reflejar_arbol_rec(B, reflejado, B.raiz(),
            ↪  reflejado.raiz());
42      }
43      return reflejado;
44  }
45
46  template <typename T>
47  void reflejar_arbol_rec(const Abin<T> &A, Abin<T> &B,
    ↪  typename Abin<T>::nodo n1, typename Abin<T>::nodo n2) {
48      if(A.hijoIzqdo(n1) != Abin<T>::NODO_NULO) {
49          B.insertarHijoDrcho(n2, A.elemento(A.hijoIzqdo(n1)));
50          reflejar_arbol_rec(A, B, A.hijoIzqdo(n1),
            ↪  B.hijoDrcho(n2));
51      }
52      if(A.hijoDrcho(n1) != Abin<T>::NODO_NULO) {
53          B.insertarHijoIzqdo(n2, A.elemento(A.hijoDrcho(n1)));
54          reflejar_arbol_rec(A, B, A.hijoDrcho(n1),
            ↪  B.hijoIzqdo(n2));
55      }
56  }
57
58  #endif // REFLEJAR_HPP
```

Ejercicio 3

El TAD árbol binario puede albergar expresiones matemáticas mediante un árbol de expresión. Dentro del árbol binario los nodos hojas contendrán los operandos, y el resto de los nodos los operadores.

- Define el tipo de los elementos del árbol para que los nodos puedan almacenar operadores y operandos.
- Implementa una función que tome un árbol binario de expresión (aritmética) y devuelva el resultado de la misma. Por simplificar el problema se puede asumir que el árbol representa una expresión correcta. Los operadores binarios posibles en la expresión aritmética serán suma, resta, multiplicación y división.

```cpp
#ifndef EXPRESION_HPP
#define EXPRESION_HPP

#include <string>
#include "../Abin/Abin_enla.hpp"


// Lo mejor seria usar un union en vez de struct
struct t_elto {
    double operando;
    char operador;
};
double operar(double a, char op, double b) {
    switch (op)
    {
    case '*':
        return a * b;
        break;
    case '/':
        return a / b;
        break;
    case '+':
        return a + b;
        break;
    case '-':
        return a - b;
```

```
27          break;
28      }
29  }
30
31  double realizar_expresion_rec(const Abin<t_elto> &A,
    ↪  typename Abin<t_elto>::nodo n) {
32      if(A.hijoIzqdo(n) == Abin<t_elto>::NODO_NULO and
        ↪  A.hijoDrcho(n) == Abin<t_elto>::NODO_NULO) //
        ↪  También es válido comprobar con un hijo nada más
33          return A.elemento(n).operando;
34      else
35          return operar(realizar_expresion_rec(A,
            ↪  A.hijoIzqdo(n)), A.elemento(n).operador,
            ↪  realizar_expresion_rec(A, A.hijoDrcho(n)));
36  }
37
38  double realizar_expresion(const Abin<t_elto> &A) {
39      return realizar_expresion_rec(A, A.raiz());
40  }
41
42
43
44  #endif // EXPRESION_HPP
```

Ejercicio 4

```cpp
#ifndef ABIN_INORDEN_HPP
#define ABIN_INORDEN_HPP

#include <cmath>
#include <cassert>

template <typename T>
class Abin {
public:
    typedef size_t nodo;

    static const nodo NODO_NULO;
    static const char ELTO_NULO;

    Abin(unsigned);

    void insertarRaiz(const T &e);
    void insertarHijoIzqdo(nodo n, const T &e);
    void insertarHijoDrcho(nodo n, const T &e);

    void eliminarHijoIzqdo(nodo n);
    void eliminarHijoDrcho(nodo n);
    void eliminarRaiz();

    bool arbolVacio() const;
    const T &elemento(nodo n) const;
    T &elemento(nodo n);

    nodo raiz() const;
    nodo padre(nodo n) const;
    nodo hijoIzqdo(nodo n) const;
    nodo hijoDrcho(nodo n) const;

    Abin(const Abin<T> &A);
    Abin &operator=(const Abin<T> &A);
    ~Abin();
private:
    T *nodos;
```

```
39      size_t maxNodos;
40      unsigned alturaMax;
41      unsigned altura(nodo n) const;
42      unsigned profundidad(nodo n, nodo objetivo, nodo inicio,
        ↪  nodo fin) const;
43  };
44
45  template <typename T>
46  static const typename Abin<T>::nodo
    ↪  Abin<T>::NODO_NULO(SIZE_MAX);
47  template <typename T>
48  static const char Abin<T>::ELTO_NULO('-');
49
50  template <typename T>
51  Abin<T>::Abin(unsigned altura) : alturaMax(altura),
    ↪  maxNodos(pow(2, altura + 1) - 1), nodos(new nodo[pow(2,
    ↪  altura + 1) - 1]) {
52      for(nodo n = 0; n < maxNodos; ++n)
53          nodos[n] = ELTO_NULO;
54  }
55
56  template <typename T>
57  unsigned Abin<T>::profundidad(nodo n, nodo objetivo, nodo
    ↪  inicio, nodo fin) const {
58      if(n == objetivo) // mitad == n
59          return 0;
60      else {
61          nodo mitad = (fin + inicio) / 2;
62          if(n < objetivo)
63              return 1 + profundidad(inicio + mitad, objetivo,
                ↪  inicio, mitad);
64          else
65              return 1 + profundidad(fin - mitad, objetivo,
                ↪  mitad, fin);
66      }
67  }
68
69  // Máxima altura que el nodo n puede alcanzar
70  template <typename T>
71  unsigned Abin<T>::altura(nodo n) const {
```

```
72      return alturaMax - profundidad((maxNodos - 1) / 2, n, 0,
        ↪  maxNodos - 1);
73  }
74
75  // Los métodos que pedía el ejercicio
76
77  template <typename T>
78  void Abin<T>::insertarRaiz(const T &e) {
79      assert(nodos[(maxNodos - 1) / 2] == ELTO_NULO);
80      nodos[(maxNodos - 1) / 2] = e;
81  }
82
83
84  template <typename T>
85  void Abin<T>::insertarHijoIzqdo(nodo n, const T &e) {
86      assert(n >= 0 and n < maxNodos - 1);
87      assert(nodos[n] != ELTO_NULO);
88      unsigned h = altura(n);
89      assert(n - pow(2, h - 1) >= 0);
90      assert(nodos[n - pow(2, h - 1)] == ELTO_NULO);
91      nodos[n - pow(2, h - 1)] = e;
92  }
93
94  template <typename T>
95  typename Abin<T>::nodo Abin<T>::padre(nodo n) const{
96      assert(n >= 0 and n < maxNodos);
97      assert(nodos[n] != ELTO_NULO);
98      if(n == ((maxNodos - 1) / 2))
99          return NODO_NULO;
100     else {
101         unsigned h = altura(n);
102         if((n % (pow(2, h + 2))) == pow(2, h) - 1)
103             return nodos[n + pow(2,h)];
104         else
105             return nodos[n - pow(2,h)];
106     }
107 }
108
109
```

```
110 #endif // ABIN_INORDEN_HPP
```

Gemini

Lleva tu estudio al siguiente nivel con Gemini, tu asistente de IA de Google ✨

¡Pruébalo ahora!

Práctica 3

Ejercicio 1

Implementa un subprograma que dado un árbol general nos calcule su grado.

```
#ifndef GRADO_AGEN_HPP
#define GRADO_AGEN_HPP

#include "../Agen/Agen_enla.hpp"
#include <algorithm>


// Versión propia

// template <typename T>
// unsigned calcular_grado_arbol(const Agen<T> &A) {
//     if(A.arbolVacio() or A.hijoIzqdo(A.raiz()) ==
↪  Agen<T>::NODO_NULO)
//         return 0;
//     else
//         return calcular_grado(A, A.raiz(), 0);
// }

// template <typename T>
// unsigned calcular_grado(const Agen<T> &A, typename
↪  Agen<T>::nodo n, unsigned max) {
//     if(n != Agen<T>::NODO_NULO) {
//         unsigned cont = 0;
//         typename Agen<T>::nodo m = n;
//         while(m != Agen<T>::NODO_NULO) {
//             cont++;
//             m = A.hermDrcho(m);
//         }
//         if(max < cont)
//             max = cont;
//         calcular_grado(A, A.hijoIzqdo(n), max);
//         return max;
//     }
// }
```

```
33
34  // Versión ineficiente
35
36  // template <typename T>
37  // unsigned calcular_grado(const Agen<T> &A) {
38  //     return calcular_grado_rec(A.raiz(), A);
39  // }
40
41  // template <typename T>
42  // unsigned num_hijos(typename Agen<T>::nodo n, const
    ↪  Agen<T> &A) {
43  //     unsigned nHijos = 0;
44  //     typename Agen<T>::nodo hijo = A.hijoIzqdo(n);
45  //     while(hijo != Agen<T>::NODO_NULO) {
46  //         hijo = A.hermDrcho(hijo);
47  //         nHijos++;
48  //     }
49  //     return nHijos;
50  // }
51
52  // template <typename T>
53  // unsigned calcular_grado_rec(typename Agen<T>::nodo n,
    ↪  const Agen<T> &A) {
54  //     if(n == Agen<T>::NODO_NULO)
55  //         return 0;
56  //     else {
57  //         unsigned grad = num_hijos(n, A);
58  //         typename Agen<T>::nodo hijo = A.hijoIzqdo(n);
59  //         while(hijo != Agen<T>::NODO_NULO) {
60  //             grad = std::max(grad,
    ↪  calcular_grado_rec(hijo, A));
61  //             hijo = A.hermDrcho(hijo);
62  //         }
63  //         return grad;
64  //     }
65  // }
66
67  // Versión definitiva
68
69  template <typename T>
```

```cpp
unsigned calcular_grado(const Agen<T> &A) {
    return calcular_grado_rec(A.raiz(), A);
}

template <typename T>
unsigned calcular_grado_rec(typename Agen<T>::nodo n, const
 ↪  Agen<T> &A) {
    if(n == Agen<T>::NODO_NULO)
        return 0;
    else {
        unsigned grad = 0;
        unsigned numHijos = 0;
        typename Agen<T>::nodo hijo = A.hijoIzqdo(n);
        while(hijo != Agen<T>::NODO_NULO) {
            numHijos++;
            grad = std::max(grad, calcular_grado_rec(hijo,
             ↪  A));
            hijo = A.hermDrcho(hijo);
        }
        return std::max(numHijos, grad);
    }
}

#endif // GRADO_AGEN_HPP
```

Ejercicio 2

Implementa un subprograma que dados un árbol y un nodo dentro de dicho árbol determine la profundidad de éste nodo en el árbol.

Es igual que en abin

```cpp
#ifndef PROFUNDIDAD_NODO_HPP
#define PROFUNDIDAD_NODO_HPP

#include "../Abin/Abin_enla.hpp"
#include <cassert>

template <typename T>
int calcular_profundidad_nodo(const Abin<T> &A, typename
 ↪  Abin<T>::nodo n) {
        assert(!A.arbolVacio() and n != Abin<T>::NODO_NULO);
        typename Abin<T>::nodo it = n;
        int profundidad = -1;
        while(it != Abin<T>::NODO_NULO) {
                it = A.padre(it);
                profundidad++;
        }
        return profundidad;
}


#endif // PROFUNDIDAD_NODO_HPP
```

Solo habría que cambiar Abin por Agen

Ejercicio 3

Se define el desequilibrio de un árbol general como la máxima diferencia entre las alturas de los subárboles más bajo y más alto de cada nivel. Implementa un subprograma que calcule el grado de desequilibrio de un árbol general.

```cpp
#ifndef DESEQUILIBRIO_AGEN_HPP
#define DESEQUILIBRIO_AGEN_HPP

#include "../Agen/Agen_enla.hpp"
#include <algorithm>

template <typename T>
int alturaMax(const Agen<T> &A, typename Agen<T>::nodo n) {
    if(n == Agen<T>::NODO_NULO)
        return -1;
    else {
        typename Agen<T>::nodo hijo = A.hijoIzqdo(n);
        int max = 0;
        while(hijo  != Agen<T>::NODO_NULO) {
            max = std::max(max, alturaMax(A, hijo));
            hijo = A.hermDrcho(hijo);
        }
        return 1 + max;
    }
}

template <typename T>
int alturaMin(const Agen<T> &A, typename Agen<T>::nodo n) {
    if(n == Agen<T>::NODO_NULO)
        return -1;
    else {
        typename Agen<T>::nodo hijo = A.hijoIzqdo(n);
        int min = 999;
        while(hijo  != Agen<T>::NODO_NULO) {
            min = std::min(min, alturaMin(A, hijo));
            hijo = A.hermDrcho(hijo);
        }
        return 1 + min;
    }
```

```
35  }
36
37  template <typename T>
38  int desequilibrio(const Agen<T> &A) {
39      return desequilibrio_rec(A, A.raiz());
40  }
41
42  template <typename T>
43  int desequilibrio_rec(const Agen<T> &A, typename
    ↪  Agen<T>::nodo n) {
44      if(n == Agen<T>::NODO_NULO)
45          return 0;
46      else {
47          typename Agen<T>::nodo hijo = A.hijoIzqdo(n);
48          int max = 0;
49          while(hijo != Agen<T>::NODO_NULO) {
50              max = std::max((alturaMax(A,n) - alturaMin(A,n)),
                ↪  desequilibrio_rec(A, hijo));
51              hijo = A.hermDrcho(hijo);
52          }
53          return max;
54      }
55  }
56
57
58
59  #endif // DESEQUILIBRIO_AGEN_HPP
```

Ejercicio 4

Dado un árbol general de enteros A y un entero x, implementa un subprograma que realice la poda de A a partir de x. Se asume que no hay elementos repetidos en A.

```
#ifndef PODA_HPP
#define PODA_HPP

#include "../Agen/Agen_enla.hpp"

void destruir_nodo(Agen<int> &A, int x, typename
↪  Agen<int>::nodo padre) {
    typename Agen<int>::nodo hijo = A.hijoIzqdo(padre);
    if(A.elemento(hijo) == x)
        A.eliminarHijoIzqdo(padre);
    else {
        while(A.elemento(A.hermDrcho(hijo)) != x)
            hijo = A.hermDrcho(hijo);
        A.eliminarHermDrcho(hijo);
    }
}

void destruir_nodos(Agen<int> &A, typename Agen<int>::nodo
↪  n) {
        typename Agen<int>::nodo hijo = A.hijoIzqdo(n);
        while(hijo != Agen<int>::NODO_NULO) {
            destruir_nodos(A,hijo);
            A.eliminarHijoIzqdo(n);
            // hijo = A.hermDrcho(hijo);
            hijo = A.hijoIzqdo(n);
        }
}

void poda_rec(Agen<int> &A, int x, typename Agen<int>::nodo
↪  n) {
    if(A.elemento(n) == x) {
        destruir_nodos(A, n); // Destruye hijos de n
                              // n se queda hoja
        if(n != A.raiz())
            destruir_nodo(A, x, A.padre(n)); // Destruye el
             ↪  nodo que tiene x
```

```
33          else
34              A.eliminarRaiz();
35      }
36      else {
37          typename Agen<int>::nodo hijo = A.hijoIzqdo(n);
38          while(hijo != Agen<int>::NODO_NULO) {
39              poda_rec(A,x,hijo);
40              if(hijo != Agen<int>::NODO_NULO)
41                  hijo = A.hermDrcho(hijo);
42          }
43      }
44  }
45
46  void poda(Agen<int> &A, int x) {
47      if(!A.arbolVacio())
48          poda_rec(A,x,A.raiz());
49  }
50
51
52
53
54  #endif // PODA_HPP
```

Práctica 4

Ejercicio 1

Implementa una nueva operación del TAD Abb que tomando un elemento del mismo elimine al completo el subárbol que cuelga de él.

```cpp
#ifndef PODA_ABB_HPP
#define PODA_ABB_HPP

#include "../ABB/abb.hpp"

template <typename T>
void poda_abb(Abb<T> &A, T elto)
{
    Abb<T> objetivo = A.buscar(elto);
    // if(!objetivo.vacio()) {
        // if(objetivo.izqdo().vacio() and
        ↪  objetivo.drcho().vacio())
            // A.eliminar(objetivo.elemento());
        // else {
    while(!objetivo.vacio())
    {
        T elemento_objetivo;

        if(!objetivo.izqdo().vacio())
            elemento_objetivo = objetivo.izqdo().elemento();
        else if(!objetivo.drcho().vacio())
            elemento_objetivo = objetivo.drcho().elemento();
        else
            elemento_objetivo = objetivo.elemento();

        A.eliminar(elemento_objetivo);
        objetivo = A.buscar(objetivo.elemento());
    }
        // }
    // }
}

#endif // PODA_ABB_HPP
```

Ejercicio 2

Un árbol binario de búsqueda se puede equilibrar realizando el recorrido en inorden del árbol para obtener el listado ordenado de sus elementos y a continuación, repartir equitativamente los elementos a izquierda y derecha colocando la mediana en la raíz y construyendo recursivamente los subárboles izquierdo y derecho de cada nodo. Implementa este algoritmo para equilibrar un ABB.

```
#ifndef EQUILIBRAR_ABB_HPP
#define EQUILIBRAR_ABB_HPP

#include "../ABB/abb.hpp"
#include <iostream>
template <typename T>
unsigned contar_elementos(const Abb<T> &A)
{
    if(A.vacio())
        return 0;
    else
        return 1 + contar_elementos(A.izqdo()) +
        ↪  contar_elementos(A.drcho());
}

template <typename T>
void rellenar_inorden(const Abb<T> &A, T *v, unsigned &i) {
    if(!A.vacio())
    {
        rellenar_inorden(A.izqdo(), v, i);
        v[i++] = A.elemento();
        rellenar_inorden(A.drcho(), v, i);
    }
}

template <typename T>
void equilibrar_abb_rec(Abb<T> &A, T *v, unsigned inicio,
↪  unsigned fin) {
    if(inicio < fin)
    {
        unsigned mediana = (inicio + (fin - 1)) / 2; // £Por
        ↪  qué el -1?
```

```
30
31          A.insertar(v[mediana]);
32
33          equilibrar_abb_rec(A, v, inicio, mediana);
34          equilibrar_abb_rec(A, v, mediana + 1, fin);
35      }
36  }
37
38  template <typename T>
39  void equilibrar_abb(Abb<T> &A)
40  {
41      unsigned n = contar_elementos(A);
42      if(n != 0) {
43          T *v = new T[n];
44          unsigned i = 0;
45          rellenar_inorden(A, v, i);
46          A.~Abb();
47          equilibrar_abb_rec(A, v, 0, n);
48
49          delete []v;
50      }
51  }
52
53  #endif // EQUILIBRAR_ABB_HPP
```

Ejercicio 3

Dados dos conjuntos representados mediante árboles binarios de búsqueda, implementa la operación unión de dos conjuntos que devuelva como resultado otro conjunto que sea la unión de ambos, representado por un ABB equilibrado.

```cpp
#ifndef UNION_HPP
#define UNION_HPP

#include "../ABB/abb.hpp"

template <typename T>
using Conjunto = Abb<T>;

template <typename T>
void unir(const Conjunto<T> &A, Conjunto<T> &C)
{
    if(!A.vacio())
    {
        C.insertar(A.elemento());
        unir(A.izqdo(), C);
        unir(A.drcho(), C);
    }
}

template <typename T>
Conjunto<T> union_abb(const Conjunto<T> &A, const
 ↪  Conjunto<T> &B)
{
    Conjunto<T> C(A);

    unir(B, C);

    equilibrar_abb(C);

    return C;
}

#endif // UNION_HPP
```

Ejercicio 3

Dados dos conjuntos representados mediante árboles binarios de búsqueda, implementa la operación intersección de dos conjuntos, que devuelva como resultado otro conjunto que sea la intersección de ambos. El resultado debe quedar en un árbol equilibrado.

```cpp
#ifndef INTERSECCION_HPP
#define INTERSECCION_HPP

#include "../ABB/abb.hpp"

template <typename T>
using Conjunto = Abb<T>;

template <typename T>
void intersectar(const Conjunto<T> &A, const Conjunto<T> &B,
 ↪  Conjunto<T> &C)
{
    if(!A.vacio())
    {
        if(B.pertenece(A.elemento())) // Obviamente
         ↪  pertenece a A
            C.insertar(A.elemento());
        intersectar(A.izqdo(), B, C);
        intersectar(A.drcho(), B, C);
    }
}

template <typename T>
Conjunto<T> interseccion_abb(const Conjunto<T> &A, const
 ↪  Conjunto<T> &B)
{
    Conjunto<T> C;
    intersectar(A,B,C);
    equilibrar_abb(C);
    return C;
}

#endif // INTERSECCION_HPP
```

Funcion pertenece:

```
template <typename T>
bool Abb<T>::pertenece(const T &e) const {
    return !buscar(e).vacio();
}
```

Ejercicio 5

Implementa el operador ♦ para conjuntos definido como $A \blacklozenge B = (A \cup B)$ - $(A \cap B)$. La implementación del operador ♦ debe realizarse utilizando obligatoriamente la operación $\in$, que nos indica si un elemento dado pertenece o no a un conjunto. La representación del tipo Conjunto debe ser tal que la operación de pertenencia esté en el caso promedio en *O(log n)*.

```
#ifndef DIAMANTE_HPP
#define DIAMANTE_HPP

#include "../ABB/abb.hpp"

template <typename T>
using Conjunto = Abb<T>;

// Forma original (Esta es más fácil)
// template <typename T>
// void diamante_rec(const Conjunto<T> &A, const Conjunto<T> &B, Conjunto<T> &C)
// {
//     if(!A.vacio())
//     {
//         if(!B.pertenece(A.elemento())) // Y pertenece a A obvio
//             C.insertar(A.elemento());
//         diamante_rec(A.izqdo(), B, C);
//         diamante_rec(A.drcho(), B, C);
//     }
// }

// template <typename T>
// Conjunto<T> diamante(const Conjunto<T> &A, const Conjunto<T> &B)
// {
//     Conjunto<T> C;
//     diamante_rec(A, B, C);
//     diamante_rec(B, A, C);
//     equilibrar_abb(C);
//     return C;
// }
```

```
31
32  // Otra forma
33
34  template <typename T>
35  void diamante_rec(const Conjunto<T> &union_a_b, const
    ↪  Conjunto<T> &interseccion_a_b, Conjunto<T> &C)
36  {
37      if(!union_a_b.vacio())
38      {
39          if(!interseccion_a_b.pertenece(union_a_b.elemento()))
40              C.insertar(union_a_b.elemento());
41          diamante_rec(union_a_b.izqdo(), interseccion_a_b, C);
42          diamante_rec(union_a_b.drcho(), interseccion_a_b, C);
43      }
44  }
45
46  template <typename T>
47  Conjunto<T> diamante(const Conjunto<T> &A, const Conjunto<T>
    ↪  &B)
48  {
49      Conjunto<T> union_a_b = union_abb(A,B);
50      Conjunto<T> interseccion_a_b = interseccion_abb(A,B);
51      Conjunto<T> C;
52      diamante_rec(union_a_b, interseccion_a_b, C);
53      equilibrar_abb(C);
54      return C;
55  }
56
57  #endif // DIAMANTE_HPP
```

Práctica 5

Ejercicio 1

Dado un árbol binario de enteros donde el valor de cada nodo es menor que el de sus hijos, implementa un subprograma para eliminar un valor del mismo preservando la propiedad de orden establecida. Explica razonadamente la elección de la estructura de datos.
Nota: Se supone que en el árbol no hay elementos repetidos, y que el número de nodos del mismo no está acotado
Gracias a la nota sabemos que no es un APO, debido a que en un APO puede haber elementos repetidos, entonces tiene que ser un Abin.

```
#ifndef ELIMINAR_ORDEN_HPP
#define ELIMINAR_ORDEN_HPP

#include "../Abin/Abin_enla.hpp"


// Modificar si encuentro algo menor paro
template <typename T>
typename Abin<T>::nodo buscar(const Abin<T> &A, typename
 ↪  Abin<T>::nodo n ,T elto)
{
    if(n == Abin<T>::NODO_NULO) // or A.elemento(n) == elto;
     ↪  return n
        return Abin<T>::NODO_NULO;
    else
    {
        if(A.elemento(n) == elto)
            return n;
        else
        {
            typename Abin<T>::nodo izq =
             ↪  buscar(A,A.hijoIzqdo(n), elto);
            if(izq != Abin<T>::NODO_NULO)
                return izq;
            else
                buscar(A,A.hijoDrcho(n),elto);
        }
```

```
25      }
26  }
27
28  template <typename T>
29  typename Abin<T>::nodo hundir(Abin<T> &A, typename
    ↪  Abin<T>::nodo n)
30  {
31      // if(n != Abin<T>::NODO_NULO) // Sobra?
32      // {
33      if(A.hijoDrcho(n) != Abin<T>::NODO_NULO or A.hijoIzqdo(n)
        ↪  != Abin<T>::NODO_NULO) // No es hoja, es decir no he
        ↪  acabado
34      {
35          // A partir de aqui alguno debe ser no nulo
36          typename Abin<T>::nodo hijo;
37          if(A.hijoDrcho(n) != Abin<T>::NODO_NULO and
            ↪  A.hijoIzqdo(n) != Abin<T>::NODO_NULO) // Tiene
            ↪  los dos hijos
38          {
39              if(A.elemento(A.hijoDrcho(n)) <
                ↪  A.elemento(A.hijoIzqdo(n)))
40                  hijo = A.hijoDrcho(n);
41              else
42                  hijo = A.hijoIzqdo(n);
43          }
44          else
45              if(A.hijoDrcho(n) != Abin<T>::NODO_NULO) // Solo
                ↪  tiene hijo derecho
46                  hijo = A.hijoDrcho(n);
47              else // Solo tiene hijo izquierdo
48                  hijo = A.hijoIzqdo(n);
49          // Burbuja
50          // T aux = A.elemento(n);
51          A.elemento(n) = A.elemento(hijo);
52          // A.elemento(hijo) = aux;
53          hundir(A, hijo);
54      }
55      else // He acabado
56          return n;
57      // }
```

```
58  }
59
60  template <typename T>
61  void eliminar_orden(Abin<T> &A, T elto)
62  {
63      if(!A.arbolVacio())
64      {
65          if(A.hijoIzqdo(A.raiz()) == Abin<T>::NODO_NULO and
            ↪  A.hijoDrcho(A.raiz()) == Abin<T>::NODO_NULO) //
            ↪  Solo está la raíz
66          {
67              if(A.elemento(A.raiz()) == elto)
68                  A.eliminarRaiz();
69          }
70          else
71          {
72              typename Abin<T>::nodo buscado = buscar(A,
                ↪  A.raiz(), elto);
73              if(buscado != Abin<T>::NODO_NULO)
74              {
75                  typename Abin<T>::nodo objetivo =
                    ↪  hundir(A,buscado);
76                  if(A.hijoDrcho(A.padre(objetivo)) ==
                    ↪  objetivo)
77                      A.eliminarHijoDrcho(A.padre(objetivo));
78                  else
79                      A.eliminarHijoIzqdo(A.padre(objetivo));
80              }
81          }
82      }
83  }
84
85
86
87
88  #endif // ELIMINAR_ORDEN_HPP
```

Ejercicio 2

Un montículo min-max tiene una estructura similar a la de un montículo ordinario (árbol parcialmente ordenado), pero la ordenación parcial consiste en que los elementos que se encuentran en un nivel par (0, 2, 4,...) son menores o iguales que sus elementos descendientes, mientras que los elementos que se encuentran en un nivel impar (1, 3, 5,...) son mayores o iguales que sus descendientes. Esto quiere decir que para cualquier elemento e de un nivel par se cumple $abuelo \leq e \leq padre$ y para cualquier elemento e de un nivel impar $padre \leq e \leq abuelo$. Implementa una operación de orden logarítmico para añadir un elemento a un montículo min-max almacenado en un vector de posiciones relativas.

```
#ifndef APO_MIN_MAX_HPP
#define APO_MIN_MAX_HPP

#include <cassert>
#include <cmath>

template <typename T>
class Apo {
public:
    explicit Apo(size_t maxNodos); // constructor
    void insertar(const T& e);
    void suprimir();

    const T& cima() const;
    bool vacio() const;

    Apo(const Apo<T>& A); // ctor. de copia
    Apo<T>& operator =(const Apo<T>& A); // asignación de
    ↪ apo
    ~Apo();
private:
    typedef size_t nodo; // índice del vector entre 0 y
    ↪ maxNodos-1

    T* nodos; // vector de nodos

    size_t maxNodos; // tamaño del vector
    size_t numNodos; // último nodo del árbol
```

```
27
28      nodo padre(nodo i) const { return (i-1)/2; }
29      nodo hIzq(nodo i) const { return 2*i+1; }
30      nodo hDer(nodo i) const { return 2*i+2; }
31
32      void flotar(nodo i);
33      void hundir(nodo i);
34  };
35
36  template <typename T>
37  inline void Apo<T>::insertar(const T& e)
38  {
39      assert(numNodos < maxNodos); // Apo no lleno.
40      nodos[numNodos] = e;
41
42      if (++numNodos > 1)
43          flotar(numNodos - 1);
44  }
45
46  template <typename T>
47  void Apo<T>::flotar(nodo i) // Casi igual que en APO pero
    ↪  con abuelos
48  {
49      T e = nodos[i];
50      unsigned nivel = log2(numNodos);
51      if((nivel % 2) == 0)
52      {
53          while (i > 1 && e < nodos[padre(padre(i))])
54          {
55              nodos[i] = nodos[padre(padre(i))];
56              i = padre(padre(i));
57          }
58          // Comparar con padre luego
59          if(i > 0 and e > nodos[padre(i)])
60          {
61              nodos[i] = nodos[padre(i)];
62              i = padre(i);
63          }
64      }
65      else
```

```
66      {
67          while (i > 1 && e > nodos[padre(padre(i))])
68          {
69              nodos[i] = nodos[padre(padre(i))];
70              i = padre(padre(i));
71          }
72          // Comparar con padre luego
73          if(i > 0 and e < nodos[padre(i)])
74          {
75              nodos[i] = nodos[padre(i)];
76              i = padre(i);
77          }
78      }
79      nodos[i] = e;
80  }
81
82  template <typename T>
83  void Apo<T>::suprimir()
84  {
85      assert(numNodos > 0);
86      nodo hijo;
87      --numNodos;
88      if(numNodos > 1) // No esta solo la raíz, en ese caso
        ↪ simplemente elimino la raiz
89      {
90          if(hDer(0) > numNodos) // Existe mas de un hijo
91          {
92              if(nodos[hIzq(0)] >= nodos[hDer(0)])
93                  hijo = hIzq(0);
94              else
95                  hijo = hDer(0);
96          }
97          else // Solo esta el hIzq
98              hijo = hIzq(0);
99          nodos[hijo] = numNodos;
100         hundir(hijo);
101     }
102 }
103
104 template <typename T>
```

```
105 void Apo<T>::hundir(nodo i)
106 {
107     // Esto es para nivel impar unicamente
108     bool fin = false;
109     T e = nodos[i];
110 
111     while(hIzq(hIzq(i)) < numNodos and !fin) // Mientras
    ↪  exista al menos un nieto y no haya acabado
112     {
113         nodo nieto_mayor = hIzq(hIzq(i));
114         if(hDer(hIzq(i)) < numNodos and nodos[hDer(hIzq(i))]
        ↪  > nodos[nieto_mayor])
115             nieto_mayor = hDer(hIzq(i));
116         if(hIzq(hDer(i)) < numNodos and nodos[hIzq(hDer(i))]
        ↪  > nodos[nieto_mayor])
117             nieto_mayor = hIzq(hDer(i));
118         if(hDer(hDer(i)) < numNodos and nodos[hDer(hDer(i))]
        ↪  > nodos[nieto_mayor])
119             nieto_mayor = hDer(hDer(i));
120         if(nodos[nieto_mayor] > e) // Si el nieto mayor es
        ↪  mayor que mi elemento, continúo
121         {
122             nodos[i] = nodos[nieto_mayor];
123             i = nieto_mayor;
124         }
125         else // Si no, esta bien colocado
126             fin = true;
127     }
128     // Luego comparar con los hijos si tuviera, si el
    ↪  elemento que tengo es menor que el mayor hijo lo
    ↪  cambio
129     nodo hijo;
130     if(hIzq(i) < numNodos)
131     {
132         if(hDer(i) < numNodos and nodos[hDer(i)] >
        ↪  nodos[hIzq(i)])
133             hijo = hDer(i);
134         else
135             hijo = hIzq(i);
136     }
```

```
137     if(e < nodos[hijo])
138     {
139         nodos[i] = nodos[hijo];
140         i = hijo;
141     }
142     nodos[i] = e;
143 }
144 
145 template <typename T>
146 inline Apo<T>::Apo(size_t maxNodos) : nodos(new
    ↪  T[maxNodos]), maxNodos(maxNodos), numNodos(0) {}
147 
148 template <typename T>
149 inline const T& Apo<T>::cima() const
150 {
151     assert(numNodos > 0); // Apo no vacío.
152     return nodos[0];
153 }
154 
155 template <typename T>
156 inline bool Apo<T>::vacio() const
157 {
158     return (numNodos == 0);
159 }
160 
161 template <typename T>
162 inline Apo<T>::~Apo()
163 {
164     delete[] nodos;
165 }
166 
167 template <typename T>
168 Apo<T>::Apo(const Apo<T>& A) : nodos(new T[A.maxNodos]),
    ↪  maxNodos(A.maxNodos), numNodos(A.numNodos)
169 {
170     // Copiar el vector.
171     for (nodo n = 0; n < numNodos; n++)
172         nodos[n] = A.nodos[n];
173 }
174 
```

```
175 template <typename T>
176 Apo<T>& Apo<T>::operator =(const Apo<T>& A)
177 {
178     if (this != &A) // Evitar autoasignación.
179     { // Destruir el vector y crear uno nuevo si es
        ↪  necesario.
180 
181         if (maxNodos != A.maxNodos)
182         {
183             delete[] nodos;
184             maxNodos = A.maxNodos;
185             nodos = new T[maxNodos];
186         }
187 
188         numNodos = A.numNodos;
189 
190         // Copiar el vector
191         for (nodo n = 0; n < numNodos; n++)
192             nodos[n] = A.nodos[n];
193     }
194     return *this;
195 }
196 
197 #endif // APO_MIN_MAX_HPP
```

Ejercicio 3

Implementa una operación de orden logarítmico para eliminar el elemento máximo de un montículo min-max definido como en el problema anterior.

Está hecho en el ejercicio anterior

Ejercicio 4

Un árbol es estrictamente ternario si todos sus nodos son hojas o tienen tres hijos. Escribe una función que, dado un árbol de grado arbitrario, nos indique si es o no estrictamente ternario.

```
#ifndef ARBOL_TERNARIO
#define ARBOL_TERNARIO

#include "../Agen/Agen_enla.hpp"

template <typename T>
unsigned num_hijos(const Agen<T> &A, typename Agen<T>::nodo
 ↪  n)
{
    unsigned cont = 0;
    if(n != Agen<T>::NODO_NULO)
    {
        typename Agen<T>::nodo hijo = A.hijoIzqdo(n);
        while(hijo != Agen<T>::NODO_NULO)
        {
            hijo = A.hermDrcho(hijo);
            cont++;
        }
    }
    return cont;
}

template <typename T>
bool es_ternario_rec(const Agen<T> &A, typename
 ↪  Agen<T>::nodo n)
{
    if(n != Agen<T>::NODO_NULO)
    {
        unsigned n_hijos = num_hijos(A,n);
        if(n_hijos == 3)
            return es_ternario_rec(A, A.hijoIzqdo(n)) and
             ↪  es_ternario_rec(A,
             ↪  A.hermDrcho(A.hijoIzqdo(n))) and
             ↪  es_ternario_rec(A,
             ↪  A.hermDrcho(A.hermDrcho(A.hijoIzqdo(n))));
```

```
30          else if(n_hijos == 0)
31              return true;
32          else
33              return false;
34      } // else return true?
35
36  }
37
38  template <typename T>
39  bool es_ternario(const Agen<T> &A)
40  {
41      if(A.arbolVacio())
42          return true;
43      else // En realidad else sobra pero bueno
44          return es_ternario_rec(A, A.raiz());
45  }
46
47  #endif // ARBOL_TERNARIO
```

Ejercicio 4

Una forma de representar una figura plana en blanco y negro consiste en utilizar un árbol cuaternario en el que cada nodo o tiene exactamente cuatro hijos, o bien es una hoja. Un nodo hoja puede ser blanco o negro y un nodo interno no tiene color.
Una figura dibujada dentro de un cuadrado de lado 2^k se representa de la forma siguiente: Se divide el cuadrado en cuatro cuadrantes y cada uno se representa como un hijo del nodo raíz. Si un cuadrante está completamente negro corresponde a una hoja negra; si, por el contrario, el cuadrante está completamente blanco, éste corresponde a una hoja blanca; y si un cuadrante está parcialmente ocupado por negro y blanco, entonces corresponde a un nodo interno del árbol y este cuadrante se representa siguiendo el mismo método subdividiéndolo en otros cuatro cuadrantes. Como ejemplo se muestra una figura en blanco y negro y su árbol asociado, tomando los cuadrantes en el sentido de las agujas del reloj a partir del cuadrante superior izquierdo.
Implementa una función que dado un árbol de esta clase, con k+1 niveles, devuelva la figura asociada, representada como una matriz cuadrada de tamaño 2^k en la que cada celda representa un punto blanco o negro.
Nota: Por simplificar el problema, se asume que en cada nodo del árbol se incluyen las coordenadas de la esquina superior izquierda y de la esquina inferior derecha del cuadrante que representa.

```
#ifndef FIGURA_PLANA_HPP
#define FIGURA_PLANA_HPP

#include "../Agen/Agen_enla.hpp"
#include <algorithm>
#include <vector>
#include <cmath>
#include <utility>
#include <iostream>

/* PRIMERO CON LA NOTA */

// struct nodo_cuadrante
// {
//     unsigned color; // 0 blanco, 1 negro, 2 no tiene
 ↪  color
//     unsigned fila_inicio, columna_inicio;
//     unsigned fila_fin, columna_fin;
```

```
18 // };
19
20 // unsigned calcularNiveles_rec(const Agen<nodo_cuadrante>
   ↪  &A, typename Agen<nodo_cuadrante>::nodo n)
21 // {
22 //     if(n == Agen<nodo_cuadrante>::NODO_NULO)
23 //         return 0;
24 //     else
25 //     {
26 //         unsigned maxNiveles = 0;
27 //         typename Agen<nodo_cuadrante>::nodo hijo =
   ↪  A.hijoIzqdo(n);
28 //         while(hijo != Agen<nodo_cuadrante>::NODO_NULO)
29 //         {
30 //             maxNiveles = std::max(maxNiveles,
   ↪  calcularNiveles_rec(A, hijo));
31 //             hijo = A.hermDrcho(hijo);
32 //         }
33 //         return maxNiveles + 1;
34 //     }
35 // }
36
37 // unsigned calcularNiveles(const Agen<nodo_cuadrante> &A)
38 // {
39 //     return calcularNiveles_rec(A,A.raiz());
40 // }
41
42
43 // bool es_hoja(typename Agen<nodo_cuadrante>::nodo n,
   ↪  const Agen<nodo_cuadrante> &A)
44 // {
45 //     return A.hijoIzqdo(n) ==
   ↪  Agen<nodo_cuadrante>::NODO_NULO; // Si no tiene hijo
   ↪  izquierdo no tiene ninguno
46 // }
47
48 // void rellenar_matriz_figura(typename
   ↪  Agen<nodo_cuadrante>::nodo n, const
   ↪  Agen<nodo_cuadrante> &A,
   ↪  std::vector<std::vector<unsigned>> &M)
```

```
49 // {
50 //     if(es_hoja(n,A))
51 //             for(unsigned i = A.elemento(n).fila_inicio;
    ↪  i < A.elemento(n).fila_fin; ++i)
52 //                 for(unsigned j =
    ↪  A.elemento(n).columna_inicio; j <
    ↪  A.elemento(n).columna_fin; ++j)
53 //                     M[i][j] = A.elemento(n).color;
54 //     else
55 //     {
56 //         typename Agen<nodo_cuadrante>::nodo hijo =
    ↪  A.hijoIzqdo(n);
57 //         for(unsigned i = 0; i < 4; i++)
58 //         {
59 //             rellenar_matriz_figura(hijo,A,M);
60 //             hijo = A.hermDrcho(hijo);
61 //         }
62 //     }
63 // }
64
65 // // Pre: A no puede ser arbol vacio
66 // std::vector<std::vector<unsigned»
    ↪  creacion_figura_plana(const Agen<nodo_cuadrante> &A)
67 // {
68 //     assert(!A.arbolVacio());
69 //     unsigned niveles = calcularNiveles(A);
70 //     // Creamos la matriz
71 //     std::vector<std::vector<unsigned» M; // Matriz de
    ↪  colores
72 //     M.reserve(pow(2,niveles-1) * pow(2, niveles-1));
73 //     rellenar_matriz_figura(A.raiz(), A, M);
74 //     return M;
75 // }
76
77 /* AHORA SIN NOTA */
78
79 typedef unsigned color;
80
81 unsigned calcularNiveles_rec(const Agen<color> &A, typename
    ↪  Agen<color>::nodo n)
```

```
82  {
83      if(n == Agen<color>::NODO_NULO)
84          return 0;
85      else
86      {
87          unsigned maxNiveles = 0;
88          typename Agen<color>::nodo hijo = A.hijoIzqdo(n);
89          while(hijo != Agen<color>::NODO_NULO)
90          {
91              maxNiveles = std::max(maxNiveles,
                ↪  calcularNiveles_rec(A, hijo));
92              hijo = A.hermDrcho(hijo);
93          }
94          return maxNiveles + 1;
95      }
96  }
97
98  unsigned calcularNiveles(const Agen<color> &A)
99  {
100     return calcularNiveles_rec(A,A.raiz());
101 }
102
103
104 bool es_hoja(typename Agen<color>::nodo n, const Agen<color>
    ↪  &A)
105 {
106     return A.hijoIzqdo(n) == Agen<color>::NODO_NULO; // Si
        ↪  no tiene hijo izquierdo no tiene ninguno
107 }
108
109 struct coordenadas {
110     unsigned x, y;
111 };
112
113 void rellenar_matriz_figura(const typename Agen<color>::nodo
    ↪  &n, coordenadas inicio, coordenadas fin ,const
    ↪  Agen<color> &A, std::vector<std::vector<unsigned>> &M)
114 {
115     if(es_hoja(n,A))
116     {
```

```
117         for(unsigned i = inicio.x; i <= fin.x; i++)
118             for(unsigned j = inicio.y; j <= fin.y; j++)
119                 M[i][j] = A.elemento(n);
120     }
121     else
122     {
123         unsigned n_cuadrante = 1;
124         Agen<color>::nodo hijo = A.hijoIzqdo(n);
125         while(hijo != Agen<color>::NODO_NULO)
126         {
127             coordenadas nuevo_inicio = inicio, nuevo_fin =
                ↪  fin;
128             switch(n_cuadrante)
129             {
130                 case 1:
131                     nuevo_fin.x = (inicio.x + fin.x) / 2;
132                     nuevo_fin.y = (inicio.y + fin.y) / 2;
133                     break;
134                 case 2:
135                     nuevo_inicio.y = ((inicio.y + fin.y) / 2)
                        ↪  + 1;
136                     nuevo_fin.x = (inicio.x + fin.x) / 2;
137                     break;
138                 case 3:
139                     nuevo_inicio.x = ((inicio.x + fin.x) / 2)
                        ↪  + 1;
140                     nuevo_inicio.y = ((inicio.y + fin.y) / 2)
                        ↪  + 1;
141                     break;
142                 case 4:
143                     nuevo_inicio.x = ((inicio.x + fin.x) / 2)
                        ↪  + 1;
144                     nuevo_fin.y = ((inicio.y + fin.y) / 2);
145                     break;
146             }
147             rellenar_matriz_figura(hijo, nuevo_inicio,
                ↪  nuevo_fin, A, M);
148             n_cuadrante++;
149             hijo = A.hermDrcho(hijo);
150         }
```

```
151     }
152 }
153
154 // Pre: A no puede ser arbol vacio
155 std::vector<std::vector<color>> creacion_figura_plana(const
    ↪  Agen<color> &A)
156 {
157     assert(!A.arbolVacio());
158     unsigned niveles = calcularNiveles(A);
159     // Creamos la matriz
160     unsigned N = pow(2, niveles - 1);
161     std::vector<std::vector<color>> M(N,
        ↪  std::vector<color>(N));
162     rellenar_matriz_figura(A.raiz(), {0,0}, {N - 1, N - 1},
        ↪  A, M);
163     return M;
164 }
165
166 #endif // FIGURA_PLANA_HPP
```