JUTGE PRO2 FIB

L6. Arbres Binaris

GitHub: https://github.com/MUX-enjoyer/PRO2-FIB-2025

Índex de Fitxers

Exercici: S42599 Alçada d'un arbre binari.cc

```cpp
#include <iostream>
using namespace std;

#include "bintree-io.hh"
#include "bintree.hh"
using namespace pro2;

/** * @brief Calcula l'alçada d'un arbre binari * @param t Un arbre binari. * @returns L'alçada de l'arbre, segons la definició anterior. */
int height(BinTree<int> t) {
 if (t.empty()) return 0;
 return 1 + max(height(t.left()), height(t.right()));
 }
```

Exercici: T78145 Mirall d'un arbre binari.cc

```cpp
#include <iostream>
using namespace std;

#include "bintree-io.hh"
#include "bintree.hh"
using namespace pro2;

/** * @brief Retorna un arbre binari que és el mirall de l'arbre `t`. * * Un arbre binari és el mirall d'un altre si les seves branques esquerra i dreta * estan intercanviades recursivament en tots els nodes. * * @param t L'arbre binari original. * * @returns Un arbre binari que és el mirall de l'arbre `t`. */
BinTree<int> reverse_tree(BinTree<int> t) {
if (t.empty()) return BinTree<int>();

BinTree<int> left, right;
left = reverse_tree(t.right());
right = reverse_tree(t.left());
return BinTree<int>(t.value(), left, right);
}
```

Exercici: T93544 Suma valors d'un arbre binari.cc

```cpp
#include <iostream>
using namespace std;

#include "bintree-io.hh"
#include "bintree.hh"
using namespace pro2;

/** * @brief Suma els valors d'un arbre binari. * * Si un node de l'arbre és buit, el seu valor és 0. * * @param t Arbre binari. * * @returns La suma dels valors dels nodes de l'arbre `t`. */
int suma_valors(BinTree<int> t) {
if (t.empty()) return 0;
return t.value() + suma_valors(t.left()) + suma_valors(t.right());
}
```

Exercici: W72736 Arbre binari de sumes.cc

```cpp
#include <iostream>
using namespace std;

#include "bintree-io.hh"
#include "bintree.hh"
using namespace pro2;

/** * @brief Retorna l'arbre de sumes de `t`. * * L'arbre de sumes és un arbre binari amb la mateixa forma * de `t` però a on cada valor conté la suma dels valors * del subarbre que penja de la mateixa posició a `t`. * * @param t L'arbre binari original. * * @returns L'arbre de sumes de `t`. */
BinTree<int> tree_of_sums(BinTree<int> t) {
if (t.empty()) return BinTree<int>();


BinTree<int> left, right;
int suma = t.value();
if (!t.left().empty()) {
left = tree_of_sums(t.left());
suma += left.value();
}

if (!t.right().empty()) {
right = tree_of_sums(t.right());
suma += right.value();
}
return BinTree<int>(suma, left, right);
}
```

Exercici: Z17905 Arbre binari de mides.cc

```cpp
#include <iostream>
using namespace std;

#include "bintree-io.hh"
#include "bintree.hh"
using namespace pro2;

/** * @brief Retorna l'arbre de mides de `t`. * * L'arbre de mides és un arbre binari amb la mateixa forma * que `t` però a on cada valor conté la quantitat total de * nodes del subarbre que penja de la mateixa posició a `t`. * * @param t L'arbre binari original. * * @returns L'arbre de mides de `t`. */
BinTree<int> tree_of_sizes(BinTree<int> t) {
 if (t.empty()) return BinTree<int>();


 BinTree<int> left, right;
 int mida = 1;
 if (!t.left().empty()) {
 left = tree_of_sizes(t.left());
 mida += left.value();
 }

 if (!t.right().empty()) {
 right = tree_of_sizes(t.right());
 mida += right.value();
 }
 return BinTree<int>(mida, left, right);

 }
```

Exercici: Z53201 Cerca un valor en un arbre binari.cc

```cpp
#include <iostream>
using namespace std;

#include "bintree-io.hh"
#include "bintree.hh"
using namespace pro2;

/** * @brief Cerca un valor en un arbre binari. * * @param t Arbre binari. * @param x Valor a cercar. * * @returns `true` si `x` es troba en algun node de l'arbre `t`, * `false` en cas contrari. */
bool cerca_valor(BinTree<int> t, int x) {
 if (t.empty()) return false;
 else if (t.value() == x) return true;

 return cerca_valor(t.left(), x) || cerca_valor(t.right(), x);
 }
```

Exercici: bintree-io.hh

```
#ifndef BINTREE_IO_HH
#define BINTREE_IO_HH

#include <algorithm>
#include <cassert>
#include <cstddef>
#include <iostream>
#include <sstream>
#include <stack>
#include <string>
#include <vector>
#include "bintree.hh"

namespace pro2 {

enum Pieces {
none = -1,
through = 0,
fork = 1,
corner = 2,
empty = 3,
};

static constexpr const char *__thru__ = "| ";
static constexpr const char *__fork__ = "|-- ";
static constexpr const char *__crnr__ = "'-- ";
static constexpr const char *__emty__ = "  ";

static constexpr const int NUM_PIECES = 4;
static constexpr const int PIECE_LENGTH = 4;
static constexpr const char *pieces[NUM_PIECES] = {
__thru__,
__fork__,
__crnr__,
__emty__,
};

template <typename T>
class BinTreeReader {
std::istream& in_;
std::string line_;
bool error_ = false;
bool skip_next_getline_ = false;

BinTree<T> fail_() {
in_.setstate(std::ios::failbit);
return BinTree<T>();
}

void getline_() {
if (skip_next_getline_) {
skip_next_getline_ = false;
} else {
getline(in_, line_);
}
}

```

```
58 T read_value_(std::string s) {
59 std::istringstream iss(s);
60
61 T t;
62 bool read_ok = bool(iss >> t);
63 iss.get(); // Force reading after to set eof bit bool read_all = iss.eof();
64 error_ = !read_ok || !read_all;
65 return t;
66 }
67
68 BinTree<T> parse_tree_(std::string expected_prefix1 = "", std::string
expected_prefix2 = "") {
69 assert(expected_prefix1.size() == expected_prefix2.size());
70 const int prefix_size = expected_prefix1.size();
71
72 if (in_.eof()) {
73 return fail_();
74 }
75
76 // Header if (line_.substr(0, prefix_size) != expected_prefix1) {
77 return fail_();
78 }
79 std::string content = line_.substr(prefix_size);
80 if (content == "#") {
81 return BinTree<T>();
82 }
83
84 T value = read_value_(content);
85 if (error_) {
86 return fail_();
87 }
88
89 // Left child getline_();
90 if (line_.substr(0, prefix_size) != expected_prefix2 ||
91 line_.substr(prefix_size, PIECE_LENGTH) != __fork__) {
92 skip_next_getline_ = true;
93 return BinTree<T>(value);
94 }
95 auto left = parse_tree_(expected_prefix2 + __fork__, expected_prefix2 +
__thru__);
96 if (in_.fail()) {
97 return BinTree<T>();
98 }
99
100 // Right child getline_();
101 if (line_.substr(0, prefix_size) != expected_prefix2 ||
102 line_.substr(prefix_size, PIECE_LENGTH) != __crnr__) {
103 return fail_();
104 }
105 auto right = parse_tree_(expected_prefix2 + __crnr__, expected_prefix2 +
__emty__);
106 if (in_.fail()) {
107 return BinTree<T>();
108 }
109
110 return BinTree<T>(value, left, right);
111 }
112
113 public:
114 BinTreeReader(std::istream& in) : in_(in), error_(false) {
115 getline_();
116 }
117
```

```
118 BinTree<T> read_tree() {
119 auto tree = parse_tree_();
120 getline_();
121 if (!line_.empty()) {
122 return fail_();
123 }
124 return tree;
125 }
126 };
127
128 template <typename T>
129 std::istream& operator>>(std::istream& i, BinTree<T>& tree) {
130 BinTreeReader<T> reader(i);
131 tree = reader.read_tree();
132 return i;
133 }
134
135 template <typename T>
136 class BinTreeWriter {
137 std::ostream& out_;
138
139 public:
140 BinTreeWriter(std::ostream& out) : out_(out) {}
141
142 void write2(BinTree<T> tree, std::string prefix1 = "", std::string prefix2 =
"") {
143 if (tree.empty()) {
144 out_ << prefix1 << "#" << std::endl;
145 return;
146 }
147 out_ << prefix1 << tree.value() << std::endl;
148 if (!tree.left().empty() || !tree.right().empty()) {
149 write2(tree.left(), prefix2 + __fork__, prefix2 + __thru__);
150 write2(tree.right(), prefix2 + __crnr__, prefix2 + __emty__);
151 }
152 }
153 };
154
155 template <typename T>
156 std::ostream& operator<<(std::ostream& o, BinTree<T> tree) {
157 BinTreeWriter<T> writer(o);
158 writer.write2(tree);
159 o << std::endl;
160 return o;
161 }
162
163 } // namespace pro2
164 #endif
```

Exercici: bintree.hh

```
#ifndef BINTREE_HH
#define BINTREE_HH

#include <cassert>
#include <memory>

namespace pro2 {

/** * @file BinTree.hh * @class BinTree * * @brief Una classe que representa un arbre binari. * * Un arbre binari pot estar buit, o contenir un valor de qualsevol tipus (`T`) així com dos * subarbres, `left` i `right`. * * Si un arbre binari està buit, el mètode `empty` retornarà `true`. * * Si no ho està, el seu valor es pot accedir amb el mètode `value`, i * ambdues branques amb els mètodes `left` i `right`, que retornen els subarbres. * * Exemple: * ```c++ * BinTree<int> a; // arbre buit * BinTree<int> b(1); // arbre només amb un valor però amb les branques buides * BinTree<int> c(0, a, b); // arbre amb un 0 i `a` i `b` com a subbranques. * ``` */
template <typename T>
class BinTree {
private:
/** * @brief Estructura que conté la informació del node * * Cada node té un valor i dos punters a les branques (si n'hi ha). * Com que cada punter pot ser `nullptr`, cadascuna de les branques pot estar buida. * * `std::shared_pointer` s'utilitza aquí perquè fa recompte de referències i * un `Node_` serà alliberat de la memòria un cop cap punter hi apunti. */
struct Node_ {
T value;
std::shared_ptr<Node_> left;
std::shared_ptr<Node_> right;

Node_(const T& value, std::shared_ptr<Node_> left, std::shared_ptr<Node_> right)
: value(value), left(left), right(right) {}
};

/** * @brief Punter al node de l'arbre */
std::shared_ptr<Node_> pnode_;

/** * @brief Construeix un arbre a partir d'un punter a node. * * Aquest constructor és privat per no exposar el punter. */
BinTree(std::shared_ptr<Node_> pnode) : pnode_(pnode) {}

public:
/** * @brief Construeix un arbre buit. Θ(1). */
BinTree() : pnode_(nullptr) {}

/** * @brief Construeix un arbre com a còpia d'un altre arbre. Θ(1). * * @param t El `BinTree` del qual copiar. */
BinTree(const BinTree& t) {
pnode_ = t.pnode_;
}

/** * @brief Construeix un arbre amb un valor `x` i sense subarbres. Θ(1). * * @param value El valor a mantenir a l'arrel del nou arbre. */
explicit BinTree(const T& value) {
pnode_ = std::make_shared<Node_>(value, nullptr, nullptr);
}
```

```
42
43 /** * @brief Construeix un arbre amb un valor `x` i dos subarbres `left` i
`right`. Θ(1). * * @param value El valor a mantenir a l'arrel del nou arbre. *
@param left El `left` subarbre en el nou arbre. * @param right El `right`
subarbre en el nou arbre. */
44 explicit BinTree(const T& value, const BinTree& left, const BinTree& right) {
45 pnode_ = std::make_shared<Node_>(value, left.pnode_, right.pnode_);
46 }
47
48 /** * @brief Assigna l'arbre `t` a aquest arbre, i retorna l'objecte mateix.
Θ(1). * * @param other L'arbre a assignar (substituirà l'antic). */
49 BinTree& operator=(const BinTree& other) {
50 pnode_ = other.pnode_;
51 return *this;
52 }
53
54 /** * @brief Retorna `true` si aquest arbre està buit, `false` en cas
contrari. Θ(1). */
55 bool empty() const {
56 return pnode_ == nullptr;
57 }
58
59 /** * @brief Retorna el subarbre esquerre d'aquest arbre. Abort si està buit.
Θ(1). * * @pre El `BinTree` no està buit. */
60 BinTree left() const {
61 assert(not empty());
62 return BinTree(pnode_->left);
63 }
64
65 /** * @brief Retorna el subarbre dret d'aquest arbre. Abort si està buit.
Θ(1). * * @pre El `BinTree` no està buit. */
66 BinTree right() const {
67 assert(not empty());
68 return BinTree(pnode_->right);
69 }
70
71 /** * @brief Retorna el valor d'aquest arbre. Abort si està buit. Θ(1). * *
@pre El `BinTree` no està buit. */
72 const T& value() const {
73 assert(not empty());
74 return pnode_->value;
75 }
76 };
77
78 } // namespace pro2
79 #endif
```