JUTGE PRO2 FIB

L6. Arbres Binaris

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

Índex de Fitxers

6.3 Avançats

Exercici: 6.1 Bàsics

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: 6.1 Bàsics

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: 6.1 Bàsics

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: 6.1 Bàsics

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: 6.1 Bàsics

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: 6.1 Bàsics

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: 6.1 Bàsics

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_);
}
```

```
56 }
57
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: 6.1 Bàsics

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) {
```

```
40 pnode_ = std::make_shared<Node_>(value, nullptr, nullptr);
41 }
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
```

Exercici: U38261 Podar un arbre binari sense repeticions

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;

static bool only_spaces_(std::string s) {
return std::all_of(s.begin(), s.end(), isspace);
}

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

void getline_() {
if (skip_next_getline_) {
```

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

```
116
117 public:
118 BinTreeReader(std::istream& in) : in_(in), error_(false) {
119 // NOTE(pauek): The first line read should have some content, so skip // any
empty lines. getline_();
120 while (!in_.eof() && only_spaces_(line_)) {
121 getline_();
122 }
123 }
124
125 BinTree<T> read_tree() {
126 auto tree = parse_tree_();
127 getline_();
128 if (!line_.empty()) {
129 return fail_();
130 }
131 return tree;
132 }
133 };
134
135 template <typename T>
136 std::istream& operator>>(std::istream& i, BinTree<T>& tree) {
137 BinTreeReader<T> reader(i);
138 tree = reader.read_tree();
139 return i;
140 }
141
142 template <typename T>
143 class BinTreeWriter {
144 std::ostream& out_;
145
146 public:
147 BinTreeWriter(std::ostream& out) : out_(out) {}
148
149 void write2(BinTree<T> tree, std::string prefix1 = "", std::string prefix2 =
"") {
150 if (tree.empty()) {
151 out_ << prefix1 << "#" << std::endl;
152 return;
153 }
154 out_ << prefix1 << tree.value() << std::endl;
155 if (!tree.left().empty() || !tree.right().empty()) {
156 write2(tree.left(), prefix2 + __fork__, prefix2 + __thru__);
157 write2(tree.right(), prefix2 + __crnr__, prefix2 + __emty__);
158 }
159 }
160 };
161
162 template <typename T>
163 std::ostream& operator<<(std::ostream& o, BinTree<T> tree) {
164 BinTreeWriter<T> writer(o);
165 writer.write2(tree);
166 o << std::endl;
167 return o;
168 }
169
170 } // namespace pro2
171 #endif
```

Exercici: U38261 Podar un arbre binari sense repeticions

bintree.hh

```cpp
#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) {
```

```
40 pnode_ = std::make_shared<Node_>(value, nullptr, nullptr);
41 }
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
```

Exercici: U38261 Podar un arbre binari sense repeticions

main.cc

```cpp
#include <iostream>
#include "bintree-io.hh"
#include "prune.hh"
using namespace pro2;
using namespace std;

int main() {
int x;
BinTree<int> t;
while (cin >> t >> x) {
auto res = prune_tree(t, x);
if (res.second) {
cout << res.first << endl;
} else {
cout << "No s'ha trobat " << x << endl;
}
}
}
```

Exercici: U38261 Podar un arbre binari sense repeticions

prune.cc

```cpp
using namespace std;
#include "prune.hh"
using namespace pro2;


/** * @brief Poda les branques amb valor `x` d'un arbre binari. * * Retorna un nou arbre binari que és una còpia de `t` * però sense les branques que contenen el valor `x`. * Alhora retorna si la poda ha tingut èxit o no s'ha podat res. * * @param t Arbre binari. * @param x Valor de les branques a podar. * * @return Retorna una parella (`std::pair`) amb l'arbre podat * i un booleà que és `true` si s'ha podat alguna branca * i `false` si l'arbre resultat és igual que `t`. */
std::pair<pro2::BinTree<int>, bool> prune_tree(pro2::BinTree<int> t, int x) {
if (t.empty()) return {BinTree<int>(), false};
if (t.value() == x) return {BinTree<int>(), true};

pair<BinTree<int>, bool> left, right;
left = prune_tree(t.left(), x);
right = prune_tree(t.right(), x);
BinTree<int> arbre_podat(t.value(), left.first, right.first);
bool haSigutPodat = left.second || right.second;
return {arbre_podat, haSigutPodat};
}
```

Exercici: U38261 Podar un arbre binari sense repeticions

prune.hh

```hpp
#ifndef PRUNE_HH
#define PRUNE_HH

#include <utility>
#include "bintree.hh"

/** * @brief Poda les branques amb valor `x` d'un arbre binari. * * Cerca en un arbre binari totes les branques amb valor `x` i les elimina, retornant un nou arbre. * * @param t Arbre binari. * @param x Valor de les branques a podar. * * @return Retorna una parella (`std::pair`) amb l'arbre podat i un booleà que és `true` si s'ha * podat alguna branca i `false` si l'arbre resultat és igual que `t`. */
std::pair<pro2::BinTree<int>, bool> prune_tree(pro2::BinTree<int> t, int x);

#endif
```

Exercici: U38461 Avaluar expressions binàries (1)

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_);
}
```

```
56 }
57
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: U38461 Avaluar expressions binàries (1)

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) {
```

```
40 pnode_ = std::make_shared<Node_>(value, nullptr, nullptr);
41 }
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
```

Exercici: U38461 Avaluar expressions binàries (1)

eval.cc

```cpp
#include "eval.hh"
#include "util.hh"
using namespace std;
using namespace pro2;


/** * @brief Avalua un arbre no buit que representa una expressió binària. * * L'expressió binària és sobre els naturals i els operadors +, -, i *. * Les operacions de l'arbre no produeixen errors de sobreeiximent * (_overflow_). * * @pre L'arbre és no buit i l'expressió binària és correcta. * * @param t Arbre que representa l'expressió binària. * @return Resultat de l'avaluació de l'expressió. */
int evaluate(BinTree<string> t) {
if (t.empty()) return 0;

string s = t.value();
if (is_number(s)) return string_to_int(s);

int l = evaluate(t.left());
int r = evaluate(t.right());

if (s == "+") return l + r;
else if (s == "-") return l - r;
else return l*r;
}
```

Exercici: U38461 Avaluar expressions binàries (1)

eval.hh

```
1 #include <iostream>
2 #include <string>
3 #include "util.hh"
4 #include "bintree.hh"
5
6
7 /** * @brief Avalua un arbre no buit que representa una expressió binària. * *
L'expressió binària és sobre els naturals i els operadors +, -, i *. * Les
operacions de l'arbre no produeixen errors d'_overflow_ (sobreeiximent). * * @pre
L'arbre és no buit i l'expressió binària és correcta. * * @param t Arbre que
representa l'expressió binària. * @return Resultat de l'avaluació de l'expressió.
*/
8 int evaluate(pro2::BinTree<std::string> t);
```

Exercici: U38461 Avaluar expressions binàries (1)

main.cc

```
#include <iostream>
#include "bintree-io.hh"
#include "eval.hh"
using namespace pro2;
using namespace std;

int main() {
BinTree<string> t;
while (cin >> t and not t.empty()) {
cout << evaluate(t) << endl;
}
}
```

Exercici: U38461 Avaluar expressions binàries (1)

util.cc

```cpp
#include "util.hh"
#include <sstream>
#include <cassert>
using namespace std;

bool is_number(string s) {
if (s.empty()) {
return false;
}
for (char c : s) {
if (!isdigit(c)) {
return false;
}
}
return true;
}

bool is_variable(string s) {
if (s.empty()) {
return false;
}
for (char c : s) {
if (!islower(c)) {
return false;
}
}
return true;
}

int string_to_int(string s) {
istringstream iss(s);
int x = 0;
iss >> x;
assert(!iss.fail());
return x;
}
```

Exercici: U38461 Avaluar expressions binàries (1)

util.hh

```hh
#ifndef UTILS_HH
#define UTILS_HH

#include <string>
#include <utility>

/** * @brief Comprova si un string està format només per dígits * * Un string buit no és un número. NO es comprova si la longitud * de `s` és massa llarga per poder-se convertir a un `int`. * * @param s String a comprovar * @return `true` si és un número, `false` altrament. */
bool is_number(std::string s);

/** * @brief Comprova si un string està format només per lletres minúscules * * @b NO mira les regles de noms de variables més complexes de C o C++. * * @param s String a comprovar * @return `true` si és una variable, `false` altrament. */
bool is_variable(std::string s);

/** * @brief Converteix un string a un enter * * Per simplicitat, si el string no és un número, retorna 0 * * @param s String a convertir * @return `int` El resultat de la conversió. Si la conversió no és possible el programa aborta. * * @note */
int string_to_int(std::string s);

#endif
```

Exercici: V80619 Podar un arbre binari

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;

static bool only_spaces_(std::string s) {
return std::all_of(s.begin(), s.end(), isspace);
}

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

void getline_() {
if (skip_next_getline_) {
```

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

```
116
117 public:
118 BinTreeReader(std::istream& in) : in_(in), error_(false) {
119 // NOTE(pauek): The first line read should have some content, so skip // any
empty lines. getline_();
120 while (!in_.eof() && only_spaces_(line_)) {
121 getline_();
122 }
123 }
124
125 BinTree<T> read_tree() {
126 auto tree = parse_tree_();
127 getline_();
128 if (!line_.empty()) {
129 return fail_();
130 }
131 return tree;
132 }
133 };
134
135 template <typename T>
136 std::istream& operator>>(std::istream& i, BinTree<T>& tree) {
137 BinTreeReader<T> reader(i);
138 tree = reader.read_tree();
139 return i;
140 }
141
142 template <typename T>
143 class BinTreeWriter {
144 std::ostream& out_;
145
146 public:
147 BinTreeWriter(std::ostream& out) : out_(out) {}
148
149 void write2(BinTree<T> tree, std::string prefix1 = "", std::string prefix2 =
"") {
150 if (tree.empty()) {
151 out_ << prefix1 << "#" << std::endl;
152 return;
153 }
154 out_ << prefix1 << tree.value() << std::endl;
155 if (!tree.left().empty() || !tree.right().empty()) {
156 write2(tree.left(), prefix2 + __fork__, prefix2 + __thru__);
157 write2(tree.right(), prefix2 + __crnr__, prefix2 + __emty__);
158 }
159 }
160 };
161
162 template <typename T>
163 std::ostream& operator<<(std::ostream& o, BinTree<T> tree) {
164 BinTreeWriter<T> writer(o);
165 writer.write2(tree);
166 o << std::endl;
167 return o;
168 }
169
170 } // namespace pro2
171 #endif
```

Exercici: V80619 Podar un arbre binari

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) {
```

```
40 pnode_ = std::make_shared<Node_>(value, nullptr, nullptr);
41 }
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
```

Exercici: V80619 Podar un arbre binari

main.cc

```
#include <iostream>
#include "bintree-io.hh"
#include "prune.hh"
using namespace pro2;
using namespace std;

int main() {
int x;
BinTree<int> t;
while (cin >> t >> x) {
auto res = prune_tree(t, x);
if (res.second) {
cout << res.first << endl;
} else {
cout << "No s'ha trobat " << x << endl << endl;
}
}
}
```

Exercici: V80619 Podar un arbre binari

prune.cc

```cpp
using namespace std;
#include "prune.hh"
using namespace pro2;


/** * @brief Poda les branques amb valor `x` d'un arbre binari. * * Retorna un nou arbre binari que és una còpia de `t` * però sense les branques que contenen el valor `x`. * Alhora retorna si la poda ha tingut èxit o no s'ha podat res. * * @param t Arbre binari. * @param x Valor de les branques a podar. * * @return Retorna una parella (`std::pair`) amb l'arbre podat * i un booleà que és `true` si s'ha podat alguna branca * i `false` si l'arbre resultat és igual que `t`. */
std::pair<pro2::BinTree<int>, bool> prune_tree(pro2::BinTree<int> t, int x) {
if (t.empty()) return {BinTree<int>(), false};
if (t.value() == x) return {BinTree<int>(), true};

pair<BinTree<int>, bool> left, right;
left = prune_tree(t.left(), x);
right = prune_tree(t.right(), x);
BinTree<int> arbre_podat(t.value(), left.first, right.first);
bool haSigutPodat = left.second || right.second;
return {arbre_podat, haSigutPodat};
}
```

Exercici: V80619 Podar un arbre binari

prune.hh

```hpp
#ifndef PRUNE_HH
#define PRUNE_HH

#include <utility>
#include "bintree.hh"

/** * @brief Poda les branques amb valor `x` d'un arbre binari. * * Cerca en un arbre binari totes les branques amb valor `x` i les elimina, retornant un nou arbre. * * @param t Arbre binari. * @param x Valor de les branques a podar. * * @return Retorna una parella (`std::pair`) amb l'arbre podat i un booleà que és `true` si s'ha * podat alguna branca i `false` si l'arbre resultat és igual que `t`. */
std::pair<pro2::BinTree<int>, bool> prune_tree(pro2::BinTree<int> t, int x);

#endif
```

Exercici: 6.3 Avançats

V22704 Camí més llarg en un arbre binari.cc

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

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

/** * @brief Retorna els valors dels nodes del camí més llarg de l'arbre binari `t`. * * Un camí va des de l'arrel de l'arbre fins a una full (un node sense fills). * Si hi ha més d'un camí màxim, cal retornar el que va per les branques de més * a l'esquerra possible. * * @param t L'arbre binari. * * @returns Els valors dels nodes del camí més llarg de `t`. */
vector<int> longest_leftmost_path(BinTree<int> t) {
if (t.empty()) return vector<int>();

vector<int> left, right;
left = longest_leftmost_path(t.left());
right = longest_leftmost_path(t.right());

if (left.size() < right.size()) {
right.insert(right.begin(), t.value());
return right;
} else {
left.insert(left.begin(), t.value());
return left;
}
}
```

Exercici: 6.3 Avançats

W90730 Ordenar un arbre binari per sumes de subarbres.cc

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

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


pair<BinTree<int>, int> sort_tree_aux(const BinTree<int>& t) {
if (t.empty()) return {BinTree<int>(), 0};

pair<BinTree<int>, int> left, right;
left = sort_tree_aux(t.left());
right = sort_tree_aux(t.right());
int total_sum = left.second + right.second + t.value();

if (right.second < left.second) {
return {BinTree<int>(t.value(), right.first, left.first), total_sum};
}
else {
return {BinTree<int>(t.value(), left.first, right.first), total_sum};
}
}

/** * @brief Retorna l'arbre `t` ordenat per sumes. * * @param t L'arbre binari original. * * @returns L'arbre resultat d'ordenar `t` per sumes. */
BinTree<int> sort_tree(BinTree<int> t) {
return sort_tree_aux(t).first;
}
```

Exercici: 6.3 Avançats

bintree-inline.hh

```cpp
#ifndef INLINE_HH
#define INLINE_HH

#include <iostream>
#include <sstream>
#include <stdexcept>
#include <string>
#include "bintree.hh"

template <typename T>
class BinTreeInlineReader {
std::string line_;
int i_;

T parse_value_(std::string token) const;
std::string read_token_();
pro2::BinTree<T> read_delimited_(char delimiter);
pro2::BinTree<T> read_();

public:
BinTreeInlineReader(std::string line) : line_(line), i_(0) {}

pro2::BinTree<T> read();
};

template <typename T>
std::string BinTreeInlineReader<T>::read_token_() {
int start = i_;
while (i_ < line_.size()) {
const char c = line_[i_];
if (c == ',' || c == '(' || c == ')') {
break;
}
i_++;
}
return line_.substr(start, i_ - start);
}

template <typename T>
T BinTreeInlineReader<T>::parse_value_(std::string token) const {
std::istringstream iss(token);
T value;
iss >> value;
return value;
}

template <typename T>
pro2::BinTree<T> BinTreeInlineReader<T>::read_delimited_(char delimiter) {
auto t = read_();
if (line_[i_] != delimiter) {
throw std::runtime_error("Unexpected character");
}
i_++;
return t;
}
```

```
56
57 // El empty debe estar delimitado template <typename T>
58 pro2::BinTree<T> BinTreeInlineReader<T>::read_() {
59 std::string token = read_token_();
60 if (token.empty()) {
61 return pro2::BinTree<T>();
62 }
63 T value = parse_value_(token);
64 if (line_[i_] != '(') {
65 return pro2::BinTree<T>(value);
66 }
67 i_++;
68 auto left = read_delimited_(',');
69 auto right = read_delimited_(')');
70 return pro2::BinTree<T>(value, left, right);
71 }
72
73 template <typename T>
74 pro2::BinTree<T> BinTreeInlineReader<T>::read() {
75 try {
76 pro2::BinTree<T> t = read_();
77 if (i_ != line_.size()) {
78 throw std::runtime_error("Expected to reach end of input");
79 }
80 return t;
81 } catch (const std::runtime_error& e) {
82 std::cerr << "Format error!" << std::endl;
83 return pro2::BinTree<T>();
84 }
85 }
86
87 template <typename T>
88 pro2::BinTree<T> bintree_inline_read(std::string line) {
89 return BinTreeInlineReader<T>(line).read();
90 }
91
92 #endif
```

Exercici: 6.3 Avançats

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_);
}
```

```
56 }
57
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: 6.3 Avançats

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) {
```

```
40 pnode_ = std::make_shared<Node_>(value, nullptr, nullptr);
41 }
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
```

Exercici: 6.3 Avançats

vector-io.hh

```
#ifndef VECTOR_IO_HH
#define VECTOR_IO_HH

#include <vector>

template <typename T>
void print_vector(const std::vector<T>& v) {
cout << "[";
if (!v.empty()) {
cout << v[0];
for (int i = 1; i < v.size(); ++i) {
cout << "," << v[i];
}
}
cout << "]" << endl;
}

#endif
```