

JUTGE PRO2 FIB

L6. Arbres Binaris

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

Índex de Fitxers

6.1 Bàsics

- S42599 Alçada d'un arbre binari.cc (pàgina 2)
- T78145 Mirall d'un arbre binari.cc (pàgina 3)
- T93544 Suma valors d'un arbre binari.cc (pàgina 4)
- W72736 Arbre binari de sumes.cc (pàgina 5)
- Z17905 Arbre binari de mides.cc (pàgina 6)
- Z53201 Cerca un valor en un arbre binari.cc (pàgina 7)
- bintree-io.hh (pàgina 8)
- bintree.hh (pàgina 11)

6.2 Consolidació

---- U38261

- bintree-io.hh (pàgina 13)
- bintree.hh (pàgina 16)
- main.cc (pàgina 18)
- prune.cc (pàgina 19)
- prune.hh (pàgina 20)

---- U38461

- bintree-io.hh (pàgina 21)
- bintree.hh (pàgina 24)
- eval.cc (pàgina 26)
- eval.hh (pàgina 27)
- main.cc (pàgina 28)
- util.cc (pàgina 29)
- util.hh (pàgina 30)

---- V80619

- bintree-io.hh (pàgina 31)
- bintree.hh (pàgina 34)
- main.cc (pàgina 36)
- prune.cc (pàgina 37)
- prune.hh (pàgina 38)

Exercici: 6.1 Bàsics

S42599 Alçada d'un arbre binari.cc

```
1 #include <iostream>
2 using namespace std;
3
4 #include "bintree-io.hh"
5 #include "bintree.hh"
6 using namespace pro2;
7
8 /** * @brief Calcula l'alçada d'un arbre binari * @param t Un arbre binari. *
9  @returns L'alçada de l'arbre, segons la definició anterior. */
10 int height(BinTree<int> t) {
11     if (t.empty()) return 0;
12     return 1 + max(height(t.left()), height(t.right()));
13 }
```

Exercici: 6.1 Bàsics

T78145 Mirall d'un arbre binari.cc

```
1 #include <iostream>
2 using namespace std;
3
4 #include "bintree-io.hh"
5 #include "bintree.hh"
6 using namespace pro2;
7
8 /** * @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`. */
9 BinTree<int> reverse_tree(BinTree<int> t) {
10     if (t.empty()) return BinTree<int>();
11
12     BinTree<int> left, right;
13     left = reverse_tree(t.right());
14     right = reverse_tree(t.left());
15     return BinTree<int>(t.value(), left, right);
16 }
```

Exercici: 6.1 Bàsics

T93544 Suma valors d'un arbre binari.cc

```
1 #include <iostream>
2 using namespace std;
3
4 #include "bintree-io.hh"
5 #include "bintree.hh"
6 using namespace pro2;
7
8 /** * @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`. */
9 int suma_valors(BinTree<int> t) {
10     if (t.empty()) return 0;
11     return t.value() + suma_valors(t.left()) + suma_valors(t.right());
12 }
```

Exercici: 6.1 Bàsics

W72736 Arbre binari de sumes.cc

```
1 #include <iostream>
2 using namespace std;
3
4 #include "bintree-io.hh"
5 #include "bintree.hh"
6 using namespace pro2;
7
8 /** * @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`. */
9 BinTree<int> tree_of_sums(BinTree<int> t) {
10     if (t.empty()) return BinTree<int>();
11
12
13     BinTree<int> left, right;
14     int suma = t.value();
15     if (!t.left().empty()) {
16         left = tree_of_sums(t.left());
17         suma += left.value();
18     }
19
20     if (!t.right().empty()) {
21         right = tree_of_sums(t.right());
22         suma += right.value();
23     }
24     return BinTree<int>(suma, left, right);
25 }
```

Exercici: 6.1 Bàsics

Z17905 Arbre binari de mides.cc

```
1 #include <iostream>
2 using namespace std;
3
4 #include "bintree-io.hh"
5 #include "bintree.hh"
6 using namespace pro2;
7
8 /** * @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`. */
9 BinTree<int> tree_of_sizes(BinTree<int> t) {
10     if (t.empty()) return BinTree<int>();
11
12
13     BinTree<int> left, right;
14     int mida = 1;
15     if (!t.left().empty()) {
16         left = tree_of_sizes(t.left());
17         mida += left.value();
18     }
19
20     if (!t.right().empty()) {
21         right = tree_of_sizes(t.right());
22         mida += right.value();
23     }
24     return BinTree<int>(mida, left, right);
25 }
26 }
```

Exercici: 6.1 Bàsics

Z53201 Cerca un valor en un arbre binari.cc

```
1 #include <iostream>
2 using namespace std;
3
4 #include "bintree-io.hh"
5 #include "bintree.hh"
6 using namespace pro2;
7
8 /** * @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. */
9 bool cerca_valor(BinTree<int> t, int x) {
10 if (t.empty()) return false;
11 else if (t.value() == x) return true;
12
13 return cerca_valor(t.left(), x) || cerca_valor(t.right(), x);
14 }
```

Exercici: 6.1 Bàsics

bintree-io.hh

```

1 #ifndef BINTREE_IO_HH
2 #define BINTREE_IO_HH
3
4 #include <algorithm>
5 #include <cassert>
6 #include <cstdint>
7 #include <iostream>
8 #include <sstream>
9 #include <stack>
10 #include <string>
11 #include <vector>
12 #include "bintree.hh"
13
14 namespace pro2 {
15
16 enum Pieces {
17     none = -1,
18     through = 0,
19     fork = 1,
20     corner = 2,
21     empty = 3,
22 };
23
24 static constexpr const char *__thru__ = " | ";
25 static constexpr const char *__fork__ = " | -- ";
26 static constexpr const char *__crnr__ = " ' -- ";
27 static constexpr const char *__empty__ = " ";
28
29 static constexpr const int NUM_PIECES = 4;
30 static constexpr const int PIECE_LENGTH = 4;
31 static constexpr const char *pieces[NUM_PIECES] = {
32     __thru__,
33     __fork__,
34     __crnr__,
35     __empty__,
36 };
37
38 template <typename T>
39 class BinTreeReader {
40     std::istream& in_;
41     std::string line_;
42     bool error_ = false;
43     bool skip_next_getline_ = false;
44
45     BinTree<T> fail_() {
46         in_.setstate(std::ios::failbit);
47         return BinTree<T>();
48     }
49
50     void getline_() {
51         if (skip_next_getline_) {
52             skip_next_getline_ = false;
53         } else {
54             getline(in_, line_);
55         }

```



```

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 +
__empty__);
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     "") {
144         if (tree.empty()) {
145             out_ << prefix1 << "#" << std::endl;
146             return;
147         }
148         out_ << prefix1 << tree.value() << std::endl;
149         if (!tree.left().empty() || !tree.right().empty()) {
150             write2(tree.left(), prefix2 + __fork__, prefix2 + __thru__);
151             write2(tree.right(), prefix2 + __crnr__, prefix2 + __empty__);
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

```

1 #ifndef BINTREE_HH
2 #define BINTREE_HH
3
4 #include <cassert>
5 #include <memory>
6
7 namespace pro2 {
8
9 /** * @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. * `` */
10 template <typename T>
11 class BinTree {
12 private:
13 /** * @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. */
14 struct Node_ {
15 T value;
16 std::shared_ptr<Node_> left;
17 std::shared_ptr<Node_> right;
18
19 Node_(const T& value, std::shared_ptr<Node_> left, std::shared_ptr<Node_>
right)
20 : value(value), left(left), right(right) {}
21 };
22
23 /** * @brief Punter al node de l'arbre */
24 std::shared_ptr<Node_> pnode_;
25
26 /** * @brief Construeix un arbre a partir d'un punter a node. * * Aquest
constructor és privat per no exposar el punter. */
27 BinTree(std::shared_ptr<Node_> pnode) : pnode_(pnode) {}
28
29 public:
30 /** * @brief Construeix un arbre buit.  $\Theta(1)$ . */
31 BinTree() : pnode_(nullptr) {}
32
33 /** * @brief Construeix un arbre com a còpia d'un altre arbre.  $\Theta(1)$ . * *
@param t El `BinTree` del qual copiar. */
34 BinTree(const BinTree& t) {
35 pnode_ = t.pnode_;
36 }
37
38 /** * @brief Construeix un arbre amb un valor `x` i sense subarbres.  $\Theta(1)$ . * *
@param value El valor a mantenir a l'arrel del nou arbre. */
39 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`.  $\Theta(1)$ . * * @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.
 $\Theta(1)$ . * * @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.  $\Theta(1)$ . */
55 bool empty() const {
56 return pnode_ == nullptr;
57 }
58
59 /** * @brief Retorna el subarbre esquerre d'aquest arbre. Abort si està buit.
 $\Theta(1)$ . * * @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.
 $\Theta(1)$ . * * @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.  $\Theta(1)$ . * *
@param 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

bintree-io.hh

```

1 #ifndef BINTREE_IO_HH
2 #define BINTREE_IO_HH
3
4 #include <algorithm>
5 #include <cassert>
6 #include <cstdint>
7 #include <iostream>
8 #include <sstream>
9 #include <stack>
10 #include <string>
11 #include <vector>
12 #include "bintree.hh"
13
14 namespace pro2 {
15
16 enum Pieces {
17     none = -1,
18     through = 0,
19     fork = 1,
20     corner = 2,
21     empty = 3,
22 };
23
24 static constexpr const char *__thru__ = "| ";
25 static constexpr const char *__fork__ = "|-- ";
26 static constexpr const char *__crnr__ = "'-- ";
27 static constexpr const char *__empty__ = " ";
28
29 static constexpr const int NUM_PIECES = 4;
30 static constexpr const int PIECE_LENGTH = 4;
31 static constexpr const char *pieces[NUM_PIECES] = {
32     __thru__,
33     __fork__,
34     __crnr__,
35     __empty__,
36 };
37
38 template <typename T>
39 class BinTreeReader {
40     std::istream& in_;
41     std::string line_;
42     bool error_ = false;
43     bool skip_next_getline_ = false;
44
45     static bool only_spaces_(std::string s) {
46         return std::all_of(s.begin(), s.end(), isspace);
47     }
48
49     BinTree<T> fail_() {
50         in_.setstate(std::ios::failbit);
51         return BinTree<T>();
52     }
53
54     void getline_() {
55         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 +
__empty__);
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 + __empty__);
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

bintree.hh

```

1 #ifndef BINTREE_HH
2 #define BINTREE_HH
3
4 #include <cassert>
5 #include <memory>
6
7 namespace pro2 {
8
9 /** * @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. * `` */
10 template <typename T>
11 class BinTree {
12 private:
13 /** * @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. */
14 struct Node_ {
15     T value;
16     std::shared_ptr<Node_> left;
17     std::shared_ptr<Node_> right;
18
19     Node_(const T& value, std::shared_ptr<Node_> left, std::shared_ptr<Node_>
    right)
20 : value(value), left(left), right(right) {}
21 };
22
23 /** * @brief Punter al node de l'arbre */
24 std::shared_ptr<Node_> pnode_;
25
26 /** * @brief Construeix un arbre a partir d'un punter a node. * * Aquest
    constructor és privat per no exposar el punter. */
27 BinTree(std::shared_ptr<Node_> pnode) : pnode_(pnode) {}
28
29 public:
30 /** * @brief Construeix un arbre buit.  $\Theta(1)$ . */
31 BinTree() : pnode_(nullptr) {}
32
33 /** * @brief Construeix un arbre com a còpia d'un altre arbre.  $\Theta(1)$ . * *
    @param t El `BinTree` del qual copiar. */
34 BinTree(const BinTree& t) {
35     pnode_ = t.pnode_;
36 }
37
38 /** * @brief Construeix un arbre amb un valor `x` i sense subarbres.  $\Theta(1)$ . * *
    @param value El valor a mantenir a l'arrel del nou arbre. */
39 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`.  $\Theta(1)$ . * * @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.
 $\Theta(1)$ . * * @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.  $\Theta(1)$ . */
55 bool empty() const {
56 return pnode_ == nullptr;
57 }
58
59 /** * @brief Retorna el subarbre esquerre d'aquest arbre. Abort si està buit.
 $\Theta(1)$ . * * @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.
 $\Theta(1)$ . * * @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.  $\Theta(1)$ . * *
@param 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

main.cc

```
1 #include <iostream>
2 #include "bintree-io.hh"
3 #include "prune.hh"
4 using namespace pro2;
5 using namespace std;
6
7 int main() {
8     int x;
9     BinTree<int> t;
10    while (cin >> t >> x) {
11        auto res = prune_tree(t, x);
12        if (res.second) {
13            cout << res.first << endl;
14        } else {
15            cout << "No s'ha trobat " << x << endl;
16        }
17    }
18 }
```

Exercici: U38261

prune.cc

```
1 using namespace std;
2 #include "prune.hh"
3 using namespace pro2;
4
5
6 /** * @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`. */
7 std::pair<pro2::BinTree<int>, bool> prune_tree(pro2::BinTree<int> t, int x) {
8 if (t.empty()) return {BinTree<int>(), false};
9 if (t.value() == x) return {BinTree<int>(), true};
10
11 pair<BinTree<int>, bool> left, right;
12 left = prune_tree(t.left(), x);
13 right = prune_tree(t.right(), x);
14 BinTree<int> arbre_podat(t.value(), left.first, right.first);
15 bool haSigutPodat = left.second || right.second;
16 return {arbre_podat, haSigutPodat};
17 }
```

Exercici: U38261

prune.hh

```
1 #ifndef PRUNE_HH
2 #define PRUNE_HH
3
4 #include <utility>
5 #include "bintree.hh"
6
7 /** * @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`. */
8 std::pair<pro2::BinTree<int>, bool> prune_tree(pro2::BinTree<int> t, int x);
9
10 #endif
```

Exercici: U38461

bintree-io.hh

```

1 #ifndef BINTREE_IO_HH
2 #define BINTREE_IO_HH
3
4 #include <algorithm>
5 #include <cassert>
6 #include <cstdint>
7 #include <iostream>
8 #include <sstream>
9 #include <stack>
10 #include <string>
11 #include <vector>
12 #include "bintree.hh"
13
14 namespace pro2 {
15
16 enum Pieces {
17     none = -1,
18     through = 0,
19     fork = 1,
20     corner = 2,
21     empty = 3,
22 };
23
24 static constexpr const char *__thru__ = " | ";
25 static constexpr const char *__fork__ = " |-- ";
26 static constexpr const char *__crnr__ = " ' -- ";
27 static constexpr const char *__empty__ = " ";
28
29 static constexpr const int NUM_PIECES = 4;
30 static constexpr const int PIECE_LENGTH = 4;
31 static constexpr const char *pieces[NUM_PIECES] = {
32     __thru__,
33     __fork__,
34     __crnr__,
35     __empty__,
36 };
37
38 template <typename T>
39 class BinTreeReader {
40     std::istream& in_;
41     std::string line_;
42     bool error_ = false;
43     bool skip_next_getline_ = false;
44
45     BinTree<T> fail_() {
46         in_.setstate(std::ios::failbit);
47         return BinTree<T>();
48     }
49
50     void getline_() {
51         if (skip_next_getline_) {
52             skip_next_getline_ = false;
53         } else {
54             getline(in_, line_);
55         }

```

```

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 +
__empty__);
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     "") {
144         if (tree.empty()) {
145             out_ << prefix1 << "#" << std::endl;
146             return;
147         }
148         out_ << prefix1 << tree.value() << std::endl;
149         if (!tree.left().empty() || !tree.right().empty()) {
150             write2(tree.left(), prefix2 + __fork__, prefix2 + __thru__);
151             write2(tree.right(), prefix2 + __crnr__, prefix2 + __empty__);
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

bintree.hh

```

1 #ifndef BINTREE_HH
2 #define BINTREE_HH
3
4 #include <cassert>
5 #include <memory>
6
7 namespace pro2 {
8
9 /** * @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. * `` */
10 template <typename T>
11 class BinTree {
12 private:
13 /** * @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. */
14 struct Node_ {
15 T value;
16 std::shared_ptr<Node_> left;
17 std::shared_ptr<Node_> right;
18
19 Node_(const T& value, std::shared_ptr<Node_> left, std::shared_ptr<Node_>
  right)
20 : value(value), left(left), right(right) {}
21 };
22
23 /** * @brief Punter al node de l'arbre */
24 std::shared_ptr<Node_> pnode_;
25
26 /** * @brief Construeix un arbre a partir d'un punter a node. * * Aquest
  constructor és privat per no exposar el punter. */
27 BinTree(std::shared_ptr<Node_> pnode) : pnode_(pnode) {}
28
29 public:
30 /** * @brief Construeix un arbre buit.  $\Theta(1)$ . */
31 BinTree() : pnode_(nullptr) {}
32
33 /** * @brief Construeix un arbre com a còpia d'un altre arbre.  $\Theta(1)$ . * *
  @param t El `BinTree` del qual copiar. */
34 BinTree(const BinTree& t) {
35 pnode_ = t.pnode_;
36 }
37
38 /** * @brief Construeix un arbre amb un valor `x` i sense subarbres.  $\Theta(1)$ . * *
  @param value El valor a mantenir a l'arrel del nou arbre. */
39 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`.  $\Theta(1)$ . * * @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.
 $\Theta(1)$ . * * @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.  $\Theta(1)$ . */
55 bool empty() const {
56 return pnode_ == nullptr;
57 }
58
59 /** * @brief Retorna el subarbre esquerre d'aquest arbre. Abort si està buit.
 $\Theta(1)$ . * * @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.
 $\Theta(1)$ . * * @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.  $\Theta(1)$ . * *
@param 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

eval.cc

```
1 #include "eval.hh"
2 #include "util.hh"
3 using namespace std;
4 using namespace pro2;
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 de sobreiximent * (_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.
  */
8 int evaluate(BinTree<string> t) {
9     if (t.empty()) return 0;
10
11     string s = t.value();
12     if (is_number(s)) return string_to_int(s);
13
14     int l = evaluate(t.left());
15     int r = evaluate(t.right());
16
17     if (s == "+") return l + r;
18     else if (s == "-") return l - r;
19     else return l*r;
20 }
```

Exercici: U38461

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

main.cc

```
1 #include <iostream>
2 #include "bintree-io.hh"
3 #include "eval.hh"
4 using namespace pro2;
5 using namespace std;
6
7 int main() {
8     BinTree<string> t;
9     while (cin >> t and not t.empty()) {
10         cout << evaluate(t) << endl;
11     }
12 }
```

Exercici: U38461

util.cc

```
1 #include "util.hh"
2 #include <sstream>
3 #include <cassert>
4 using namespace std;
5
6 bool is_number(string s) {
7     if (s.empty()) {
8         return false;
9     }
10    for (char c : s) {
11        if (!isdigit(c)) {
12            return false;
13        }
14    }
15    return true;
16 }
17
18 bool is_variable(string s) {
19     if (s.empty()) {
20         return false;
21     }
22     for (char c : s) {
23         if (!islower(c)) {
24             return false;
25         }
26     }
27     return true;
28 }
29
30 int string_to_int(string s) {
31     istringstream iss(s);
32     int x = 0;
33     iss >> x;
34     assert(!iss.fail());
35     return x;
36 }
```

Exercici: U38461

util.hh

```
1 #ifndef UTILS_HH
2 #define UTILS_HH
3
4 #include <string>
5 #include <utility>
6
7 /** * @brief Comprova si un string està format només per dígitos * * 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. */
8 bool is_number(std::string s);
9
10 /** * @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. */
11 bool is_variable(std::string s);
12
13 /** * @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 */
14 int string_to_int(std::string s);
15
16 #endif
```

Exercici: V80619

bintree-io.hh

```

1 #ifndef BINTREE_IO_HH
2 #define BINTREE_IO_HH
3
4 #include <algorithm>
5 #include <cassert>
6 #include <cstdint>
7 #include <iostream>
8 #include <sstream>
9 #include <stack>
10 #include <string>
11 #include <vector>
12 #include "bintree.hh"
13
14 namespace pro2 {
15
16 enum Pieces {
17     none = -1,
18     through = 0,
19     fork = 1,
20     corner = 2,
21     empty = 3,
22 };
23
24 static constexpr const char *__thru__ = " | ";
25 static constexpr const char *__fork__ = " | -- ";
26 static constexpr const char *__crnr__ = " ' -- ";
27 static constexpr const char *__empty__ = " ";
28
29 static constexpr const int NUM_PIECES = 4;
30 static constexpr const int PIECE_LENGTH = 4;
31 static constexpr const char *pieces[NUM_PIECES] = {
32     __thru__,
33     __fork__,
34     __crnr__,
35     __empty__,
36 };
37
38 template <typename T>
39 class BinTreeReader {
40     std::istream& in_;
41     std::string line_;
42     bool error_ = false;
43     bool skip_next_getline_ = false;
44
45     static bool only_spaces_(std::string s) {
46         return std::all_of(s.begin(), s.end(), isspace);
47     }
48
49     BinTree<T> fail_() {
50         in_.setstate(std::ios::failbit);
51         return BinTree<T>();
52     }
53
54     void getline_() {
55         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 +
__empty__);
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 + __empty__);
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

bintree.hh

```

1 #ifndef BINTREE_HH
2 #define BINTREE_HH
3
4 #include <cassert>
5 #include <memory>
6
7 namespace pro2 {
8
9 /** * @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. * `` */
10 template <typename T>
11 class BinTree {
12 private:
13 /** * @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_ptr`
  s'utilitza aquí perquè fa recompte de referències i * un `Node_` serà alliberat
  de la memòria un cop cap punter hi apunti. */
14 struct Node_ {
15 T value;
16 std::shared_ptr<Node_> left;
17 std::shared_ptr<Node_> right;
18
19 Node_(const T& value, std::shared_ptr<Node_> left, std::shared_ptr<Node_>
  right)
20 : value(value), left(left), right(right) {}
21 };
22
23 /** * @brief Punter al node de l'arbre */
24 std::shared_ptr<Node_> pnode_;
25
26 /** * @brief Construeix un arbre a partir d'un punter a node. * * Aquest
  constructor és privat per no exposar el punter. */
27 BinTree(std::shared_ptr<Node_> pnode) : pnode_(pnode) {}
28
29 public:
30 /** * @brief Construeix un arbre buit.  $\Theta(1)$ . */
31 BinTree() : pnode_(nullptr) {}
32
33 /** * @brief Construeix un arbre com a còpia d'un altre arbre.  $\Theta(1)$ . * *
  @param t El `BinTree` del qual copiar. */
34 BinTree(const BinTree& t) {
35 pnode_ = t.pnode_;
36 }
37
38 /** * @brief Construeix un arbre amb un valor `x` i sense subarbres.  $\Theta(1)$ . * *
  @param value El valor a mantenir a l'arrel del nou arbre. */
39 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`.  $\Theta(1)$ . * * @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.
 $\Theta(1)$ . * * @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.  $\Theta(1)$ . */
55 bool empty() const {
56 return pnode_ == nullptr;
57 }
58
59 /** * @brief Retorna el subarbre esquerre d'aquest arbre. Abort si està buit.
 $\Theta(1)$ . * * @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.
 $\Theta(1)$ . * * @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.  $\Theta(1)$ . * *
@param 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

main.cc

```
1 #include <iostream>
2 #include "bintree-io.hh"
3 #include "prune.hh"
4 using namespace pro2;
5 using namespace std;
6
7 int main() {
8     int x;
9     BinTree<int> t;
10    while (cin >> t >> x) {
11        auto res = prune_tree(t, x);
12        if (res.second) {
13            cout << res.first << endl;
14        } else {
15            cout << "No s'ha trobat " << x << endl << endl;
16        }
17    }
18 }
```

Exercici: V80619

prune.cc

```
1 using namespace std;
2 #include "prune.hh"
3 using namespace pro2;
4
5
6 /** * @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`. */
7 std::pair<pro2::BinTree<int>, bool> prune_tree(pro2::BinTree<int> t, int x) {
8 if (t.empty()) return {BinTree<int>(), false};
9 if (t.value() == x) return {BinTree<int>(), true};
10
11 pair<BinTree<int>, bool> left, right;
12 left = prune_tree(t.left(), x);
13 right = prune_tree(t.right(), x);
14 BinTree<int> arbre_podat(t.value(), left.first, right.first);
15 bool haSigutPodat = left.second || right.second;
16 return {arbre_podat, haSigutPodat};
17 }
```

Exercici: V80619

prune.hh

```
1 #ifndef PRUNE_HH
2 #define PRUNE_HH
3
4 #include <utility>
5 #include "bintree.hh"
6
7 /** * @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`. */
8 std::pair<pro2::BinTree<int>, bool> prune_tree(pro2::BinTree<int> t, int x);
9
10 #endif
```