

JUTGE PRO2 FIB

L7. Arbres 2

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

Índex de Fitxers

7.1 Exercicis bàsics d'Arbres Generals

- V21234 Arbre mirall.cc (pàgina 2)
- W23082 Cerca un valor en un arbre.cc (pàgina 3)
- Z14339 Imprimir expressions.cc (pàgina 4)
- Z80280 Alçada d'un arbre.cc (pàgina 5)
- tree-io.hh (pàgina 15)
- tree.hh (pàgina 19)
- **Z82639 Avaluar expressions Booleans**
- eval.cc (pàgina 6)
- eval.hh (pàgina 7)
- main.cc (pàgina 8)
- tree-io.hh (pàgina 9)
- tree.hh (pàgina 13)

7.2 Problemes sobre arbres (binaris i generals) amb inmersió

- **T47104 Arbre binari de graus de desequilibri**
- T47104 Arbre binari de graus de desequilibri.cc (pàgina 21)
- bintree-inline.hh (pàgina 22)
- bintree-io.hh (pàgina 25)
- bintree.hh (pàgina 28)
- main.cc (pàgina 30)
- vector-io.hh (pàgina 31)
- **Y97108 Reemplaça els nodes d'un arbre binari a profunditat parell per la suma per sota**
- Y97108 Reemplaça els nodes d'un arbre binari a profunditat parell per la suma per sota.cc (pàgina 32)
- bintree-inline.hh (pàgina 33)
- bintree-io.hh (pàgina 36)
- bintree.hh (pàgina 39)
- main.cc (pàgina 41)
- **Z19994 Mostra carpetes indentades**
- Z19994 Mostra carpetes indentades.cc (pàgina 42)

----- main.cc (pàgina 43)
----- tree-io.hh (pàgina 44)
----- tree.hh (pàgina 48)
---- **Z78925 Avaluar expressions binàries amb variables**
----- Z78925 Avaluar expressions binàries amb variables.cc (pàgina 50)
----- bintree-io.hh (pàgina 51)
----- bintree.hh (pàgina 55)
----- main.cc (pàgina 57)
----- util.cc (pàgina 58)
----- util.hh (pàgina 59)

Exercici: 7.1 Exercicis bàsics d'Arbres Generals

V21234 Arbre mirall.cc

```
1 #include "tree-io.hh"
2 #include "tree.hh"
3 using namespace pro2;
4
5 #include <iostream>
6 using namespace std;
7
8 /** * @brief Retorna un arbre que és el mirall de l'arbre `t`. * * Un arbre és
el mirall d'un altre si les seves branques esquerra i dreta * estan
intercanviades recursivament en tots els nodes. * * @param t L'arbre original. *
* @returns Un arbre que és el mirall de l'arbre `t`. */
9 Tree<int> tree_mirror(Tree<int> t) {
10 if (t.empty()) return Tree<int>();
11
12 vector<Tree<int>> fills;
13 for (int i = t.num_children()-1; i >= 0; --i) {
14 Tree<int> fill = tree_mirror(t.child(i));
15 fills.push_back(fill);
16 }
17 return Tree<int>(t.value(), fills);
18 }
```

Exercici: 7.1 Exercicis bàsics d'Arbres Generals

W23082 Cerca un valor en un arbre.cc

```
1 #include <iostream>
2 using namespace std;
3
4 #include "tree-io.hh"
5 #include "tree.hh"
6 using namespace pro2;
7
8 /** * @brief Cerca un valor en un arbre. * * @param t Arbre. * @param x Valor a
cercar. * * @returns Retorna `true` si `x` es troba en algun node de l'arbre `t`,
* `false` en cas contrari. */
9 bool tree_search(Tree<int> t, int x) {
10 if (t.empty()) return false;
11 if (t.value() == x) return true;
12
13 for (int i = 0; i < t.num_children(); ++i) {
14 if (tree_search(t.child(i), x)) return true;
15 }
16 return false;
17 }
```

Exercici: 7.1 Exercicis bàsics d'Arbres Generals

Z14339 Imprimir expressions.cc

```
1 #include <iostream>
2 using namespace std;
3
4 #include "tree-io.hh"
5 #include "tree.hh"
6
7 using namespace pro2;
8
9 /** * @brief Transforma una expressió en la seva representació com a `string`.
10  * * Una expressió està formada per operadors (`+`, `*` i `-`), i * operands
11  (naturals), que són nodes d'un arbre. Els operands tenen dos * fills o més, i els
12  operands són fulles (no tenen fills). * * L'expressió representada com a `string`
13  és de la següent manera. Per a * operands: cal retornar l'operand mateix. Per a
14  operadors, cal retornar * els fills de l'operador, separats per l'operador, amb
15  un espai entre * operand i operadors, i tot el conjunt sempre entre parèntesis. *
16  * @pre L'arbre representa una expressió ben formada * * @param t L'arbre que
17  representa l'expressió. * @returns La representació com a `string` de `t` */
18 string expression_to_string(Tree<string> t) {
19     if (t.empty()) return "";
20     if (t.num_children() == 0) return t.value();
21
22     string expression = "(";
23     expression.append(expression_to_string(t.child(0)));
24
25     for (int i = 1; i < t.num_children(); ++i) {
26         expression.append(" ");
27         expression.append(t.value());
28         expression.append(" ");
29         expression.append(expression_to_string(t.child(i)));
30     }
31
32     expression.append(")");
33     return expression;
34 }
```

Exercici: 7.1 Exercicis bàsics d'Arbres Generals

Z80280 Alçada d'un arbre.cc

```
1 #include <iostream>
2 using namespace std;
3
4 #include "tree-io.hh"
5 #include "tree.hh"
6 using namespace pro2;
7
8 /** * @brief Calcula l'alçada d'un arbre * @param t Un arbre. * @returns
    L'alçada de l'arbre, segons la definició anterior. */
9 int tree_height(Tree<int> t) {
10     if (t.empty()) return 0;
11
12     int max_size = 0;
13     for (int i = 0; i < t.num_children(); ++i) {
14         int size = tree_height(t.child(i));
15         if (size > max_size) max_size = size;
16     }
17     return 1+max_size;
18 }
```

Exercici: Z82639 Avaluar expressions Booleanes

eval.cc

```
1 #include "eval.hh"
2
3 bool evaluate(pro2::Tree<std::string> t) {
4     if (t.empty()) return false;
5     if (t.num_children() == 0) return t.value() == "1";
6     if (t.value() == "not") return !evaluate(t.child(0));
7
8     bool result = evaluate(t.child(0));
9     for (int i = 1; i < t.num_children(); ++i) {
10         if (t.value() == "and") {
11             result = result and evaluate(t.child(i));
12         }
13         else if (t.value() == "or") {
14             result = result or evaluate(t.child(i));
15         }
16     }
17     return result;
18 }
```

Exercici: Z82639 Avaluar expressions Booleanes

eval.hh

```
1 #include <iostream>
2 #include <string>
3 #include "tree.hh"
4
5 /** * @brief Avalua un arbre no buit que representa una expressió Booleana. * *
L'expressió és sobre l'1 (true) i el 0 (fals) i els operadors * 'and', 'or', i
'not'. * * @pre L'arbre és no buit i l'expressió és correcta, és a dir, els
operands * 'and' i 'or' tenen més d'un operand, i l'operador 'not' en té només 1.
* * @param t Arbre que representa l'expressió. * @return Resultat de l'avaluació
de l'expressió. */
6 bool evaluate(pro2::Tree<std::string> t);
```


Exercici: Z82639 Avaluar expressions Booleanes

main.cc

```
1 #include <iostream>
2 using namespace std;
3
4 #include "tree-io.hh"
5 #include "eval.hh"
6 using namespace pro2;
7
8 int main() {
9     Tree<string> t;
10    while (cin >> t) {
11        cout << evaluate(t) << endl;
12    }
13 }
```

Exercici: Z82639 Avaluar expressions Booleanes

tree-io.hh

```

1 #ifndef TREE_IO_HH
2 #define TREE_IO_HH
3
4 #include <algorithm>
5 #include <cstdint>
6 #include <iostream>
7 #include <sstream>
8 #include <stack>
9 #include <string>
10 #include <vector>
11 #include "tree.hh"
12
13 namespace pro2 {
14
15     enum Pieces {
16         none = -1,
17         through = 0,
18         fork = 1,
19         corner = 2,
20         empty = 3,
21     };
22
23     static constexpr const char *__thru__ = " | ";
24     static constexpr const char *__fork__ = " |-- ";
25     static constexpr const char *__crnr__ = " ' -- ";
26     static constexpr const char *__emty__ = "  ";
27
28     static constexpr const int NUM_PIECES = 4;
29     static constexpr const int PIECE_LENGTH = 4;
30     static constexpr const char *pieces[NUM_PIECES] = {
31         __thru__,
32         __fork__,
33         __crnr__,
34         __emty__,
35     };
36
37     template <typename T>
38     class TreeReader {
39     public:
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             for (char c : s) {
47                 if (!isspace(c)) {
48                     return false;
49                 }
50             }
51             return true;
52         }
53
54         Tree<T> fail_() {
55             in_.setstate(std::ios::failbit);
56             return Tree<T>();
57         }
58     };
59 }
60
61 #endif

```

```

56 }
57
58 void getline_() {
59 if (skip_next_getline_) {
60 skip_next_getline_ = false;
61 } else {
62 getline(in_, line_);
63 }
64 }
65
66 T read_value_(std::string s) {
67 std::istringstream iss(s);
68
69 T t;
70 bool read_ok = bool(iss >> t);
71 iss.get(); // Force reading after to set eof bit bool read_all = iss.eof();
72 error_ = !read_ok || !read_all;
73 return t;
74 }
75
76 Tree<T> parse_tree_(std::string expected_prefix1 = "", std::string
expected_prefix2 = "") {
77 assert(expected_prefix1.size() == expected_prefix2.size());
78 const int prefix_size = expected_prefix1.size();
79
80 if (in_.eof()) {
81 return fail_();
82 }
83
84 // Header if (line_.substr(0, prefix_size) != expected_prefix1) {
85 return fail_();
86 }
87 std::string content = line_.substr(prefix_size);
88 if (content == "#") {
89 return Tree<T>();
90 }
91
92 T value = read_value_(content);
93 if (error_) {
94 return fail_();
95 }
96
97 std::vector<Tree<T>> children;
98
99 // Children except last getline_();
100 while (line_.substr(0, prefix_size) == expected_prefix2 &&
101 line_.substr(prefix_size, PIECE_LENGTH) == __fork__) {
102 const auto child =
103 parse_tree_(expected_prefix2 + __fork__, expected_prefix2 + __thru__);
104 if (in_.fail()) {
105 return Tree<T>();
106 }
107 children.push_back(child);
108 getline_();
109 }
110
111 // Last child if (line_.substr(0, prefix_size) != expected_prefix2 ||
112 line_.substr(prefix_size, PIECE_LENGTH) != __crnr__) {
113 if (children.empty()) {
114 skip_next_getline_ = true;
115 return Tree<T>(value);
116 }
117 return fail_();

```

```

118 }
119 const auto child = parse_tree_(expected_prefix2 + __crnr__, expected_prefix2
+ __empty__);
120 if (in_.fail()) {
121 return Tree<T>();
122 }
123 children.push_back(child);
124
125 return Tree<T>(value, children);
126 }
127
128 public:
129 TreeReader(std::istream& in) : in_(in), error_(false) {
130 // NOTE(pauek): The first line read should have some content, so skip // any
empty lines. getline_();
131 while (!in_.eof() && only_spaces_(line_)) {
132 getline_();
133 }
134 }
135
136 Tree<T> read_tree() {
137 auto tree = parse_tree_();
138 getline_();
139 if (!line_.empty()) {
140 return fail_();
141 }
142 return tree;
143 }
144 };
145
146 template <typename T>
147 std::istream& operator>>(std::istream& i, Tree<T>& tree) {
148 TreeReader<T> reader(i);
149 tree = reader.read_tree();
150 return i;
151 }
152
153 template <typename T>
154 class TreeWriter {
155 std::ostream& out_;
156
157 public:
158 TreeWriter(std::ostream& o) : out_(o) {}
159
160 void write(Tree<T> tree, std::string prefix1 = "", std::string prefix2 = "")
{
161 if (tree.empty()) {
162 out_ << prefix1 << "#" << std::endl;
163 return;
164 }
165 out_ << prefix1 << tree.value() << std::endl;
166 for (int i = 0; i < tree.num_children() - 1; i++) {
167 write(tree.child(i), prefix2 + __fork__, prefix2 + __thru__);
168 }
169 if (tree.num_children() > 0) {
170 Tree<T> last = tree.child(tree.num_children() - 1);
171 write(last, prefix2 + __crnr__, prefix2 + __empty__);
172 }
173 }
174 };
175
176 template <typename T>
177 std::ostream& operator<<(std::ostream& o, Tree<T> tree) {

```

```
178 TreeWriter<T> writer(o);
179 writer.write(tree);
180 o << std::endl;
181 return o;
182 }
183
184 } // namespace pro2
185 #endif
```

Exercici: Z82639 Avaluar expressions Booleanes

tree.hh

```

1 #ifndef BINTREE_HH
2 #define BINTREE_HH
3
4 #include <cassert>
5 #include <memory>
6 #include <vector>
7
8 namespace pro2 {
9
10 /** * @file Tree.hh * @class Tree * @brief A class representing a tree with
    any number of children. */
11 template <typename T>
12 class Tree {
13 private:
14 /** * @brief Struct that holds the node's information */
15 struct Node_ {
16 T value;
17 std::vector<std::shared_ptr<Node_>> children;
18
19 Node_(const T& value) : value(value) {}
20
21 Node_(const T& value, std::vector<std::shared_ptr<Node_>> children)
22 : value(value), children(children) {}
23 };
24
25 /** * @brief Pointer to the node of the tree */
26 std::shared_ptr<Node_> pnode_;
27
28 /** * @brief Constructs a tree from a node pointer. */
29 Tree(std::shared_ptr<Node_> pnode) : pnode_(pnode) {}
30
31 public:
32 /** * @brief Constructs an empty tree.  $\Theta(1)$ . */
33 Tree() : pnode_(nullptr) {}
34
35 /** * @brief Constructs a tree as a copy of another tree.  $\Theta(1)$ . */
36 Tree(const Tree& tree) { pnode_ = tree.pnode_; }
37
38 /** * @brief Constructs a tree with a value `x` and no children.  $\Theta(1)$ . */
39 explicit Tree(const T& value) { pnode_ = std::make_shared<Node_>(value); }
40
41 /** * @brief Constructs a tree with a value `x` and a list of `children`.
     $\Theta(1)$ . */
42 explicit Tree(const T& value, const std::vector<Tree>& children) {
43 std::vector<std::shared_ptr<Node_>> children_pnodes;
44 for (const auto& child : children) {
45 children_pnodes.push_back(child.pnode_);
46 }
47 pnode_ = std::make_shared<Node_>(value, children_pnodes);
48 }
49
50 /** * @brief Assigns the tree `other` to this tree, returns itself.  $\Theta(1)$ . */
51 Tree& operator=(const Tree& other) {
52 pnode_ = other.pnode_;
53 return *this;

```

```
54 }
55
56 /** * @brief Returns `true` if this tree is empty, `false` otherwise.  $\Theta(1)$ . */
57 bool empty() const { return pnode_ == nullptr; }
58
59 /** * @brief Returns the i-th child subtree of this tree. Aborts if empty.
 $\Theta(1)$ . */
60 Tree child(int i) const {
61     assert(not empty());
62     assert(i >= 0 && i < pnode_>children.size());
63     return Tree(pnode_>children[i]);
64 }
65
66 /** * @brief Returns the number of children of this tree. Aborts if empty.
 $\Theta(1)$ . */
67 int num_children() const {
68     assert(not empty());
69     return pnode_>children.size();
70 }
71
72 /** * @brief Returns the value of this tree. Aborts if empty.  $\Theta(1)$ . */
73 const T& value() const {
74     assert(not empty());
75     return pnode_>value;
76 }
77 };
78
79 } // namespace pro2
80 #endif
```

Exercici: 7.1 Exercicis bàsics d'Arbres Generals

tree-io.hh

```

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

```



```

56 }
57
58 void getline_() {
59     if (skip_next_getline_) {
60         skip_next_getline_ = false;
61     } else {
62         getline(in_, line_);
63     }
64 }
65
66 T read_value_(std::string s) {
67     std::istringstream iss(s);
68
69     T t;
70     bool read_ok = bool(iss >> t);
71     iss.get(); // Force reading after to set eof bit bool read_all = iss.eof();
72     error_ = !read_ok || !read_all;
73     return t;
74 }
75
76 Tree<T> parse_tree_(std::string expected_prefix1 = "", std::string
expected_prefix2 = "") {
77     assert(expected_prefix1.size() == expected_prefix2.size());
78     const int prefix_size = expected_prefix1.size();
79
80     if (in_.eof()) {
81         return fail_();
82     }
83
84     // Header if (line_.substr(0, prefix_size) != expected_prefix1) {
85     return fail_();
86     }
87     std::string content = line_.substr(prefix_size);
88     if (content == "#") {
89         return Tree<T>();
90     }
91
92     T value = read_value_(content);
93     if (error_) {
94         return fail_();
95     }
96
97     std::vector<Tree<T>> children;
98
99     // Children except last getline_();
100     while (line_.substr(0, prefix_size) == expected_prefix2 &&
101 line_.substr(prefix_size, PIECE_LENGTH) == __fork__) {
102         const auto child =
103         parse_tree_(expected_prefix2 + __fork__, expected_prefix2 + __thru__);
104         if (in_.fail()) {
105             return Tree<T>();
106         }
107         children.push_back(child);
108         getline_();
109     }
110
111     // Last child if (line_.substr(0, prefix_size) != expected_prefix2 ||
112 line_.substr(prefix_size, PIECE_LENGTH) != __crnr__) {
113     if (children.empty()) {
114         skip_next_getline_ = true;
115         return Tree<T>(value);
116     }
117     return fail_();

```

```

118 }
119 const auto child = parse_tree_(expected_prefix2 + __crnr__, expected_prefix2
+ __empty__);
120 if (in_.fail()) {
121 return Tree<T>();
122 }
123 children.push_back(child);
124
125 return Tree<T>(value, children);
126 }
127
128 public:
129 TreeReader(std::istream& in) : in_(in), error_(false) {
130 // NOTE(pauek): The first line read should have some content, so skip // any
empty lines. getline_();
131 while (!in_.eof() && only_spaces_(line_)) {
132 getline_();
133 }
134 }
135
136 Tree<T> read_tree() {
137 auto tree = parse_tree_();
138 getline_();
139 if (!line_.empty()) {
140 return fail_();
141 }
142 return tree;
143 }
144 };
145
146 template <typename T>
147 std::istream& operator>>(std::istream& i, Tree<T>& tree) {
148 TreeReader<T> reader(i);
149 tree = reader.read_tree();
150 return i;
151 }
152
153 template <typename T>
154 class TreeWriter {
155 std::ostream& out_;
156
157 public:
158 TreeWriter(std::ostream& o) : out_(o) {}
159
160 void write(Tree<T> tree, std::string prefix1 = "", std::string prefix2 = "")
{
161 if (tree.empty()) {
162 out_ << prefix1 << "#" << std::endl;
163 return;
164 }
165 out_ << prefix1 << tree.value() << std::endl;
166 for (int i = 0; i < tree.num_children() - 1; i++) {
167 write(tree.child(i), prefix2 + __fork__, prefix2 + __thru__);
168 }
169 if (tree.num_children() > 0) {
170 Tree<T> last = tree.child(tree.num_children() - 1);
171 write(last, prefix2 + __crnr__, prefix2 + __empty__);
172 }
173 }
174 };
175
176 template <typename T>
177 std::ostream& operator<<(std::ostream& o, Tree<T> tree) {

```

```
178 TreeWriter<T> writer(o);
179 writer.write(tree);
180 o << std::endl;
181 return o;
182 }
183
184 } // namespace pro2
185 #endif
```

Exercici: 7.1 Exercicis bàsics d'Arbres Generals

tree.hh

```

1 #ifndef BINTREE_HH
2 #define BINTREE_HH
3
4 #include <cassert>
5 #include <memory>
6 #include <vector>
7
8 namespace pro2 {
9
10 /** * @file Tree.hh * @class Tree * @brief A class representing a tree with
    any number of children. */
11 template <typename T>
12 class Tree {
13 private:
14 /** * @brief Struct that holds the node's information */
15 struct Node_ {
16 T value;
17 std::vector<std::shared_ptr<Node_>> children;
18
19 Node_(const T& value) : value(value) {}
20
21 Node_(const T& value, std::vector<std::shared_ptr<Node_>> children)
22 : value(value), children(children) {}
23 };
24
25 /** * @brief Pointer to the node of the tree */
26 std::shared_ptr<Node_> pnode_;
27
28 /** * @brief Constructs a tree from a node pointer. */
29 Tree(std::shared_ptr<Node_> pnode) : pnode_(pnode) {}
30
31 public:
32 /** * @brief Constructs an empty tree.  $\Theta(1)$ . */
33 Tree() : pnode_(nullptr) {}
34
35 /** * @brief Constructs a tree as a copy of another tree.  $\Theta(1)$ . */
36 Tree(const Tree& tree) { pnode_ = tree.pnode_; }
37
38 /** * @brief Constructs a tree with a value `x` and no children.  $\Theta(1)$ . */
39 explicit Tree(const T& value) { pnode_ = std::make_shared<Node_>(value); }
40
41 /** * @brief Constructs a tree with a value `x` and a list of `children`.
     $\Theta(1)$ . */
42 explicit Tree(const T& value, const std::vector<Tree>& children) {
43 std::vector<std::shared_ptr<Node_>> children_pnodes;
44 for (const auto& child : children) {
45 children_pnodes.push_back(child.pnode_);
46 }
47 pnode_ = std::make_shared<Node_>(value, children_pnodes);
48 }
49
50 /** * @brief Assigns the tree `other` to this tree, returns itself.  $\Theta(1)$ . */
51 Tree& operator=(const Tree& other) {
52 pnode_ = other.pnode_;
53 return *this;

```

```
54 }
55
56 /** * @brief Returns `true` if this tree is empty, `false` otherwise.  $\Theta(1)$ . */
57 bool empty() const { return pnode_ == nullptr; }
58
59 /** * @brief Returns the i-th child subtree of this tree. Aborts if empty.
 $\Theta(1)$ . */
60 Tree child(int i) const {
61     assert(not empty());
62     assert(i >= 0 && i < pnode_->children.size());
63     return Tree(pnode_->children[i]);
64 }
65
66 /** * @brief Returns the number of children of this tree. Aborts if empty.
 $\Theta(1)$ . */
67 int num_children() const {
68     assert(not empty());
69     return pnode_->children.size();
70 }
71
72 /** * @brief Returns the value of this tree. Aborts if empty.  $\Theta(1)$ . */
73 const T& value() const {
74     assert(not empty());
75     return pnode_->value;
76 }
77 };
78
79 } // namespace pro2
80 #endif
```

Exercici: T47104 Arbre binari de graus de desequilibri

T47104 Arbre binari de graus de desequilibri.cc

```
1 #include "bintree-inline.hh"
2 #include "bintree-io.hh"
3 #include "bintree.hh"
4 #include "vector-io.hh"
5 using namespace pro2;
6
7 #include <chrono>
8 #include <iostream>
9 using namespace std;
10 using namespace std::chrono;
11
12 BinTree<int> aux_bintree_of_height_diffs(BinTree<int> t, int& height) {
13     if (t.empty()) return BinTree<int>();
14
15     int left_height = 0, right_height = 0;
16     BinTree<int> left = aux_bintree_of_height_diffs(t.left(), left_height);
17     BinTree<int> right = aux_bintree_of_height_diffs(t.right(), right_height);
18
19     height = 1 + max(left_height, right_height);
20     int balance_factor = left_height - right_height;
21
22     return BinTree<int>(balance_factor, left, right);
23 }
24
25 /** * @brief Retorna l'arbre de graus de desequilibri de `t`. * * @param t
    L'arbre binari original. * @returns L'arbre de graus de desequilibri de `t`. */
26 BinTree<int> bintree_of_height_diffs(BinTree<int> t) {
27     int height;
28     return aux_bintree_of_height_diffs(t, height);
29 }
```

Exercici: T47104 Arbre binari de graus de desequilibri

bintree-inline.hh

```

1 #ifndef INLINE_HH
2 #define INLINE_HH
3
4 #include <iostream>
5 #include <sstream>
6 #include <stdexcept>
7 #include <string>
8 #include "bintree.hh"
9
10 template <typename T>
11 class BinTreeInlineReader {
12     std::string line_;
13     int i_;
14
15     T parse_value_(std::string token) const;
16     std::string read_token_();
17     pro2::BinTree<T> read_delimited_(char delimiter);
18     pro2::BinTree<T> read_();
19
20 public:
21     BinTreeInlineReader(std::string line) : line_(line), i_(0) {}
22
23     pro2::BinTree<T> read();
24 };
25
26 template <typename T>
27 std::string BinTreeInlineReader<T>::read_token_() {
28     int start = i_;
29     while (i_ < line_.size()) {
30         const char c = line_[i_];
31         if (c == ',' || c == '(' || c == ')') {
32             break;
33         }
34         i_++;
35     }
36     return line_.substr(start, i_ - start);
37 }
38
39 template <typename T>
40 T BinTreeInlineReader<T>::parse_value_(std::string token) const {
41     std::istringstream iss(token);
42     T value;
43     iss >> value;
44     return value;
45 }
46
47 template <typename T>
48 pro2::BinTree<T> BinTreeInlineReader<T>::read_delimited_(char delimiter) {
49     auto t = read_();
50     if (line_[i_] != delimiter) {
51         throw std::runtime_error("Unexpected character");
52     }
53     i_++;
54     return t;
55 }

```

```

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 template <typename T>
93 void bintree_inline_write__(pro2::BinTree<T> t) {
94     if (t.empty()) {
95         return;
96     }
97     std::cout << t.value();
98     auto left = t.left();
99     auto right = t.right();
100    if (left.empty() and right.empty()) {
101        return;
102    }
103    std::cout << "(";
104    bintree_inline_write__(left);
105    std::cout << ",";
106    bintree_inline_write__(right);
107    std::cout << ")";
108 }
109
110 template <typename T>
111 void bintree_inline_write(pro2::BinTree<T> t) {
112     if (t.empty()) {
113         std::cout << "()";
114         return;
115     }
116     bintree_inline_write__(t);
117 }
118

```


119 *#endif*

Exercici: T47104 Arbre binari de graus de desequilibri

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: T47104 Arbre binari de graus de desequilibri

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.  $\Theta(1)$ . * *
@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: T47104 Arbre binari de graus de desequilibri

main.cc

```
1 #include "bintree-inline.hh"
2 #include "bintree-io.hh"
3 #include "bintree.hh"
4 #include "vector-io.hh"
5 using namespace pro2;
6
7 #include <chrono>
8 #include <iostream>
9 using namespace std;
10 using namespace std::chrono;
11
12 #include "T47104 Arbre binari de graus de desequilibri.cc"
13
14 void main_inline() {
15     string line;
16     while (getline(cin, line)) {
17         auto t = bintree_inline_read<int>(line);
18         auto D = bintree_of_height_diffs(t);
19         bintree_inline_write(D);
20     }
21 }
22
23 void main_visual() {
24     BinTree<int> t;
25     while (cin >> t) {
26         auto D = bintree_of_height_diffs(t);
27         cout << D;
28     }
29 }
30
31 int main() {
32     string format, line;
33     getline(cin, format); // determina el format dels arbres if (format ==
34     "inline") {
35         main_inline();
36     } else {
37         main_visual();
38     }
```

Exercici: T47104 Arbre binari de graus de desequilibri

vector-io.hh

```
1 #ifndef VECTOR_IO_HH
2 #define VECTOR_IO_HH
3
4 #include <vector>
5 #include <iostream>
6
7 template <typename T>
8 void print_vector(const std::vector<T>& v) {
9     std::cout << "[";
10    if (!v.empty()) {
11        std::cout << v[0];
12        for (int i = 1; i < v.size(); ++i) {
13            std::cout << ", " << v[i];
14        }
15    }
16    std::cout << "]" << std::endl;
17 }
18
19 #endif
```


Exercici: Y97108 Reemplaça els nodes d'un arbre binari a profunditat parell per la suma per sota

Y97108 Reemplaça els nodes d'un arbre binari a profunditat parell per la suma per sota.cc

```
1 #include <iostream>
2 using namespace std;
3
4 #include "bintree.hh"
5 using namespace pro2;
6
7 BinTree<int> aux_sum_below_at_even_depth(BinTree<int> t, int depth, int& sum) {
8     if (t.empty()) return BinTree<int>();
9
10    int sum_left = 0, sum_right = 0;
11    BinTree<int> left, right;
12    left = aux_sum_below_at_even_depth(t.left(), depth+1, sum_left);
13    right = aux_sum_below_at_even_depth(t.right(), depth+1, sum_right);
14
15    sum = t.value() + sum_left + sum_right;
16
17    if (depth % 2 == 0) return BinTree<int>(sum, left, right);
18    else return BinTree<int>(t.value(), left, right);
19 }
20
21 /** * @brief Retorna l'arbre `t` reemplaçant els valors dels nodes a
    profunditat parell per la suma per sota * * @param t L'arbre binari original. * *
    @returns Un arbre binari R amb la mateixa estructura que t. * Per a cada posició
    p de t i R, si p és a profunditat senar, * llavors t i R tenen el mateix valor a
    posició p. * En canvi, si p es a profunditat parell, llavors el valor de R a
    posició * p és la suma de tots els valors que es troben a t a posició p i per
    sota. */
22 BinTree<int> sum_below_at_even_depth(BinTree<int> t) {
23     int suma = 0;
24     return aux_sum_below_at_even_depth(t, 0, suma);
25 }
```

Exercici: Y97108 Reemplaça els nodes d'un arbre binari a profunditat parell per la suma per sota

bintree-inline.hh

```

1 #ifndef INLINE_HH
2 #define INLINE_HH
3
4 #include <iostream>
5 #include <sstream>
6 #include <stdexcept>
7 #include <string>
8 #include "bintree.hh"
9
10 template <typename T>
11 class BinTreeInlineReader {
12     std::string line_;
13     int i_;
14
15     T parse_value_(std::string token) const;
16     std::string read_token_();
17     pro2::BinTree<T> read_delimited_(char delimiter);
18     pro2::BinTree<T> read_();
19
20 public:
21     BinTreeInlineReader(std::string line) : line_(line), i_(0) {}
22
23     pro2::BinTree<T> read();
24 };
25
26 template <typename T>
27 std::string BinTreeInlineReader<T>::read_token_() {
28     int start = i_;
29     while (i_ < line_.size()) {
30         const char c = line_[i_];
31         if (c == ',' || c == '(' || c == ')') {
32             break;
33         }
34         i_++;
35     }
36     return line_.substr(start, i_ - start);
37 }
38
39 template <typename T>
40 T BinTreeInlineReader<T>::parse_value_(std::string token) const {
41     std::istringstream iss(token);
42     T value;
43     iss >> value;
44     return value;
45 }
46
47 template <typename T>
48 pro2::BinTree<T> BinTreeInlineReader<T>::read_delimited_(char delimiter) {
49     auto t = read_();
50     if (line_[i_] != delimiter) {
51         throw std::runtime_error("Unexpected character");
52     }

```

```

53 i_++;
54 return t;
55 }
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 template <typename T>
93 void bintree_inline_write__(pro2::BinTree<T> t) {
94     if (t.empty()) {
95         return;
96     }
97     std::cout << t.value();
98     auto left = t.left();
99     auto right = t.right();
100     if (left.empty() and right.empty()) {
101         return;
102     }
103     std::cout << "(";
104     bintree_inline_write__(left);
105     std::cout << ",";
106     bintree_inline_write__(right);
107     std::cout << ")";
108 }
109
110 template <typename T>
111 void bintree_inline_write(pro2::BinTree<T> t) {
112     if (t.empty()) {
113         std::cout << "()";
114         return;
115     }

```

```
116 bintree_inline_write__(t);  
117 std::cout << std::endl;  
118 }  
119  
120 #endif
```

Exercici: Y97108 Reemplaça els nodes d'un arbre binari a profunditat parell per la suma per sota

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     public:
41         std::istream& in_;
42         std::string line_;
43         bool error_ = false;
44         bool skip_next_getline_ = false;
45
46         BinTree<T> fail_() {
47             in_.setstate(std::ios::failbit);
48             return BinTree<T>();
49         }
50
51         void getline_() {
52             if (skip_next_getline_) {
53                 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: Y97108 Reemplaça els nodes d'un arbre binari a profunditat parell per la suma per sota

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.  $\Theta(1)$ . * *
    @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: Y97108 Reemplaça els nodes d'un arbre binari a profunditat parell per la suma per sota

main.cc

```
1 #include <chrono>
2 #include <iostream>
3 using namespace std;
4 using namespace std::chrono;
5
6 #include "bintree-inline.hh"
7 #include "bintree-io.hh"
8 #include "bintree.hh"
9 using namespace pro2;
10
11 #include "Y97108 Reemplaça els nodes d'un arbre binari a profunditat parell
per la suma per sota.cc"
12
13 void main_visual() {
14     BinTree<int> t;
15     while (cin >> t) {
16         cout << sum_below_at_even_depth(t);
17     }
18 }
19
20 void main_inline() {
21     string line;
22     while (getline(cin, line)) {
23         BinTree<int> t = bintree_inline_read<int>(line);
24         BinTree<int> D = sum_below_at_even_depth(t);
25         bintree_inline_write(D);
26     }
27 }
28
29 int main() {
30     std::ios::sync_with_stdio(false);
31
32     string format, line;
33     getline(cin, format); // determina el format dels arbres if (format ==
"inline") {
34         main_inline();
35     } else {
36         main_visual();
37     }
38 }
```

Exercici: Z19994 Mostra carpetes indentades

Z19994 Mostra carpetes indentades.cc

```
1 #include <iostream>
2 using namespace std;
3
4 #include "tree-io.hh"
5 using namespace pro2;
6
7
8 void aux_print_folders(Tree<string> t, int indent_size, int depth) {
9     if (t.empty()) return;
10
11     for (int i = 0; i < depth*indent_size; ++i) cout << " ";
12     cout << t.value() << endl;
13     for (int i = 0; i < t.num_children(); ++i) {
14         aux_print_folders(t.child(i), indent_size, depth+1);
15     }
16 }
17
18 /** * @brief Mostra un arbre de carpetes i fitxers a la sortida en format *
19     indentat * * Per exemple: * `` * projecte * documents * codi font * main.cc *
20     Makefile * jocs de prova * publics * sample-1.inp * sample-2.inp * privats *
21     Reunio 2024-12-10 * * `` * * Cal notar que la última línia és buida per poder
22     veure la separació * amb altres arbres de carpetes * * Malgrat això no és
23     rellevant per al problema, les fulles de l'arbre * són fitxers o carpetes i la
24     resta són sempre carpetes. * * @param t Un arbre de strings a on cada `string`
25     representa una carpeta * o un fitxer. * @param indent_size Número d'espais
26     d'indentació per a cada nivell. * * @pre `indent_size` > 0. * * @post S'ha
27     mostrat per la sortida estàndard l'arbre `t` amb un nivell * d'indentació
28     d'`indent_size` espais per a cada nivell de profunditat. */
29 void print_folders(Tree<string> t, int indent_size) {
30     aux_print_folders(t, indent_size, 0);
31     cout << endl;
32 }
```

Exercici: Z19994 Mostra carpetes indentades

main.cc

```
1 #include <iostream>
2 using namespace std;
3
4 #include "tree-io.hh"
5 using namespace pro2;
6
7 #include "Z19994 Mostra carpetes indentades.cc"
8
9 int main() {
10     Tree<string> t;
11     int indent_size;
12     while (cin >> indent_size >> t) {
13         print_folders(t, indent_size);
14     }
15 }
```

Exercici: Z19994 Mostra carpetes indentades

tree-io.hh

```

1 #ifndef TREE_IO_HH
2 #define TREE_IO_HH
3
4 #include <algorithm>
5 #include <cstdint>
6 #include <iostream>
7 #include <sstream>
8 #include <stack>
9 #include <string>
10 #include <vector>
11 #include "tree.hh"
12
13 namespace pro2 {
14
15     enum Pieces {
16         none = -1,
17         through = 0,
18         fork = 1,
19         corner = 2,
20         empty = 3,
21     };
22
23     static constexpr const char *__thru__ = "| ";
24     static constexpr const char *__fork__ = "|-- ";
25     static constexpr const char *__crnr__ = "'-- ";
26     static constexpr const char *__emty__ = " ";
27
28     static constexpr const int NUM_PIECES = 4;
29     static constexpr const int PIECE_LENGTH = 4;
30     static constexpr const char *pieces[NUM_PIECES] = {
31         __thru__,
32         __fork__,
33         __crnr__,
34         __emty__,
35     };
36
37     template <typename T>
38     class TreeReader {
39     public:
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             for (char c : s) {
47                 if (!isspace(c)) {
48                     return false;
49                 }
50             }
51             return true;
52         }
53
54         Tree<T> fail_() {
55             in_.setstate(std::ios::failbit);
56             return Tree<T>();
57         }
58     };
59 }
60
61 #endif

```

```

56 }
57
58 void getline_() {
59     if (skip_next_getline_) {
60         skip_next_getline_ = false;
61     } else {
62         getline(in_, line_);
63     }
64 }
65
66 T read_value_(std::string s) {
67     std::istringstream iss(s);
68
69     T t;
70     bool read_ok = bool(iss >> t);
71     iss.get(); // Force reading after to set eof bit bool read_all = iss.eof();
72     error_ = !read_ok || !read_all;
73     return t;
74 }
75
76 Tree<T> parse_tree_(std::string expected_prefix1 = "", std::string
expected_prefix2 = "") {
77     assert(expected_prefix1.size() == expected_prefix2.size());
78     const int prefix_size = expected_prefix1.size();
79
80     if (in_.eof()) {
81         return fail_();
82     }
83
84     // Header if (line_.substr(0, prefix_size) != expected_prefix1) {
85     return fail_();
86     }
87     std::string content = line_.substr(prefix_size);
88     if (content == "#") {
89         return Tree<T>();
90     }
91
92     T value = read_value_(content);
93     if (error_) {
94         return fail_();
95     }
96
97     std::vector<Tree<T>> children;
98
99     // Children except last getline_();
100     while (line_.substr(0, prefix_size) == expected_prefix2 &&
101 line_.substr(prefix_size, PIECE_LENGTH) == __fork__) {
102         const auto child =
103         parse_tree_(expected_prefix2 + __fork__, expected_prefix2 + __thru__);
104         if (in_.fail()) {
105             return Tree<T>();
106         }
107         children.push_back(child);
108         getline_();
109     }
110
111     // Last child if (line_.substr(0, prefix_size) != expected_prefix2 ||
112 line_.substr(prefix_size, PIECE_LENGTH) != __crnr__) {
113     if (children.empty()) {
114         skip_next_getline_ = true;
115         return Tree<T>(value);
116     }
117     return fail_();

```

```

118 }
119 const auto child = parse_tree_(expected_prefix2 + __crnr__, expected_prefix2
+ __empty__);
120 if (in_.fail()) {
121 return Tree<T>();
122 }
123 children.push_back(child);
124
125 return Tree<T>(value, children);
126 }
127
128 public:
129 TreeReader(std::istream& in) : in_(in), error_(false) {
130 // NOTE(pauek): The first line read should have some content, so skip // any
empty lines. getline_();
131 while (!in_.eof() && only_spaces_(line_)) {
132 getline_();
133 }
134 }
135
136 Tree<T> read_tree() {
137 auto tree = parse_tree_();
138 getline_();
139 if (!line_.empty()) {
140 return fail_();
141 }
142 return tree;
143 }
144 };
145
146 template <typename T>
147 std::istream& operator>>(std::istream& i, Tree<T>& tree) {
148 TreeReader<T> reader(i);
149 tree = reader.read_tree();
150 return i;
151 }
152
153 template <typename T>
154 class TreeWriter {
155 std::ostream& out_;
156
157 public:
158 TreeWriter(std::ostream& o) : out_(o) {}
159
160 void write(Tree<T> tree, std::string prefix1 = "", std::string prefix2 = "")
{
161 if (tree.empty()) {
162 out_ << prefix1 << "#" << std::endl;
163 return;
164 }
165 out_ << prefix1 << tree.value() << std::endl;
166 for (int i = 0; i < tree.num_children() - 1; i++) {
167 write(tree.child(i), prefix2 + __fork__, prefix2 + __thru__);
168 }
169 if (tree.num_children() > 0) {
170 Tree<T> last = tree.child(tree.num_children() - 1);
171 write(last, prefix2 + __crnr__, prefix2 + __empty__);
172 }
173 }
174 };
175
176 template <typename T>
177 std::ostream& operator<<(std::ostream& o, Tree<T> tree) {

```

```
178 TreeWriter<T> writer(o);
179 writer.write(tree);
180 o << std::endl;
181 return o;
182 }
183
184 } // namespace pro2
185 #endif
```


Exercici: Z19994 Mostra carpetes indentades

tree.hh

```

1 #ifndef BINTREE_HH
2 #define BINTREE_HH
3
4 #include <cassert>
5 #include <memory>
6 #include <vector>
7
8 namespace pro2 {
9
10 /** * @file Tree.hh * @class Tree * @brief A class representing a tree with
    any number of children. */
11 template <typename T>
12 class Tree {
13 private:
14 /** * @brief Struct that holds the node's information */
15 struct Node_ {
16 T value;
17 std::vector<std::shared_ptr<Node_>> children;
18
19 Node_(const T& value) : value(value) {}
20
21 Node_(const T& value, std::vector<std::shared_ptr<Node_>> children)
22 : value(value), children(children) {}
23 };
24
25 /** * @brief Pointer to the node of the tree */
26 std::shared_ptr<Node_> pnode_;
27
28 /** * @brief Constructs a tree from a node pointer. */
29 Tree(std::shared_ptr<Node_> pnode) : pnode_(pnode) {}
30
31 public:
32 /** * @brief Constructs an empty tree.  $\Theta(1)$ . */
33 Tree() : pnode_(nullptr) {}
34
35 /** * @brief Constructs a tree as a copy of another tree.  $\Theta(1)$ . */
36 Tree(const Tree& tree) { pnode_ = tree.pnode_; }
37
38 /** * @brief Constructs a tree with a value `x` and no children.  $\Theta(1)$ . */
39 explicit Tree(const T& value) { pnode_ = std::make_shared<Node_>(value); }
40
41 /** * @brief Constructs a tree with a value `x` and a list of `children`.
     $\Theta(1)$ . */
42 explicit Tree(const T& value, const std::vector<Tree>& children) {
43 std::vector<std::shared_ptr<Node_>> children_pnodes;
44 for (const auto& child : children) {
45 children_pnodes.push_back(child.pnode_);
46 }
47 pnode_ = std::make_shared<Node_>(value, children_pnodes);
48 }
49
50 /** * @brief Assigns the tree `other` to this tree, returns itself.  $\Theta(1)$ . */
51 Tree& operator=(const Tree& other) {
52 pnode_ = other.pnode_;
53 return *this;

```

```
54 }
55
56 /** * @brief Returns `true` if this tree is empty, `false` otherwise.  $\Theta(1)$ . */
57 bool empty() const { return pnode_ == nullptr; }
58
59 /** * @brief Returns the i-th child subtree of this tree. Aborts if empty.
 $\Theta(1)$ . */
60 Tree child(int i) const {
61     assert(not empty());
62     assert(i >= 0 && i < pnode_>children.size());
63     return Tree(pnode_>children[i]);
64 }
65
66 /** * @brief Returns the number of children of this tree. Aborts if empty.
 $\Theta(1)$ . */
67 int num_children() const {
68     assert(not empty());
69     return pnode_>children.size();
70 }
71
72 /** * @brief Returns the value of this tree. Aborts if empty.  $\Theta(1)$ . */
73 const T& value() const {
74     assert(not empty());
75     return pnode_>value;
76 }
77 };
78
79 } // namespace pro2
80 #endif
```

Exercici: Z78925 Avaluar expressions binàries amb variables

Z78925 Avaluar expressions binàries amb variables.cc

```
1 #include <iostream>
2 #include <fstream>
3 #include <map>
4 using namespace std;
5
6 #include "bintree-io.hh"
7 #include "bintree.hh"
8 using namespace pro2;
9
10 #include "util.hh"
11
12 /** * @brief Avalua una expressió binària amb variables * * L'expressió és
sobre els naturals i els operadors `+`, `-`, i `*`. * A més, hi ha un diccionari
`env` que emmagatzema els valors d'un * conjunt de variables que poden aparèixer
a l'arbre. * * @pre `t` és no buit. Totes les variables que apareixen a `t` *
estan definides a `env`. Les operacions expressades per * l'arbre no produeixen
errors d'_overflow_ * * @param t Arbre amb l'expressió binària. * @param env
Diccionari amb parelles (nom de variable, valor). Aquest * diccionari no es pot
modificar. */
13 int tree_eval_env(BinTree<string> t, const map<string, int>& env) {
14 if (t.empty()) return 0;
15 if (is_number(t.value())) return string_to_int(t.value());
16 if (is_var_name(t.value())) return env.at(t.value());
17
18 int left, right;
19 left = tree_eval_env(t.left(), env);
20 right = tree_eval_env(t.right(), env);
21 if (t.value() == "-") return left-right;
22 else if (t.value() == "+") return left+right;
23 else return left*right;
24 }
```

Exercici: Z78925 Avaluar expressions binàries amb variables

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 // Define operator>> for pairs template <typename A, typename B>
39 inline std::istream& operator>>(std::istream& i, std::pair<A, B>& p) {
40     return i >> p.first >> p.second;
41 }
42
43 template <typename T>
44 class BinTreeReader {
45     std::istream& in_;
46     std::string line_;
47     bool error_ = false;
48     bool skip_next_getline_ = false;
49
50     static bool only_spaces_(std::string s) {
51         for (char c : s) {
52             if (!isspace(c)) {
53                 return false;
54             }
55         }
56     }
57 }

```

```

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

```

```

117 return fail_();
118 }
119 auto right = parse_tree_(expected_prefix2 + __crnr__, expected_prefix2 +
__empty__);
120 if (in_.fail()) {
121 return BinTree<T>();
122 }
123
124 return BinTree<T>(value, left, right);
125 }
126
127 public:
128 BinTreeReader(std::istream& in) : in_(in), error_(false) {
129 // NOTE(pauek): The first line read should have some content, so skip // any
empty lines. getline_();
130 while (!in_.eof() && only_spaces_(line_)) {
131 getline_();
132 }
133 }
134
135 BinTree<T> read_tree() {
136 auto tree = parse_tree_();
137 getline_();
138 if (!line_.empty()) {
139 return fail_();
140 }
141 return tree;
142 }
143 };
144
145 template <typename T>
146 std::istream& operator>>(std::istream& i, BinTree<T>& tree) {
147 BinTreeReader<T> reader(i);
148 tree = reader.read_tree();
149 return i;
150 }
151
152 // Define operator<< for pairs template <typename A, typename B>
153 inline std::ostream& operator<<(std::ostream& o, const std::pair<A, B>& p) {
154 return o << p.first << ' ' << p.second;
155 }
156
157 template <typename T>
158 class BinTreeWriter {
159 std::ostream& out_;
160
161 public:
162 BinTreeWriter(std::ostream& out) : out_(out) {}
163
164 void write2(BinTree<T> tree, std::string prefix1 = "", std::string prefix2 =
"") {
165 if (tree.empty()) {
166 out_ << prefix1 << "#" << std::endl;
167 return;
168 }
169 out_ << prefix1 << tree.value() << std::endl;
170 if (!tree.left().empty() || !tree.right().empty()) {
171 write2(tree.left(), prefix2 + __fork__, prefix2 + __thru__);
172 write2(tree.right(), prefix2 + __crnr__, prefix2 + __empty__);
173 }
174 }
175 };
176

```

```
177 template <typename T>
178 std::ostream& operator<<(std::ostream& o, BinTree<T> tree) {
179 BinTreeWriter<T> writer(o);
180 writer.write2(tree);
181 return o << std::endl;
182 }
183
184 template<typename T>
185 T read_value_(std::string text) {
186 std::istringstream iss(text);
187 T elem;
188 iss >> elem;
189 return elem;
190 }
191
192 template <typename T>
193 pro2::BinTree<T> bintree_from_preorder(std::istream& in) {
194 std::string token;
195 in >> token;
196 if (token == "#" || !in) {
197 return pro2::BinTree<T>();
198 }
199 T value = read_value_<T>(token);
200 auto left = bintree_from_preorder<T>(in);
201 auto right = bintree_from_preorder<T>(in);
202 return pro2::BinTree<T>(value, left, right);
203 }
204
205 } // namespace pro2
206 #endif
```

Exercici: Z78925 Avaluar expressions binàries amb variables

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: Z78925 Avaluar expressions binàries amb variables

main.cc

```
1 #include <iostream>
2 #include <fstream>
3 #include <map>
4 using namespace std;
5
6 #include "bintree-io.hh"
7 #include "bintree.hh"
8 using namespace pro2;
9
10 #include "Z78925 Avaluar expressions binàries amb variables.cc"
11
12 map<string, int> read_env(istream& in) {
13     map<string, int> env;
14     string name;
15     int value;
16     while (in >> name && name != ".") {
17         in >> value;
18         env[name] = value;
19     }
20     return env;
21 }
22
23 int main() {
24     BinTree<string> t;
25     while (cin >> t) {
26         assert(!t.empty());
27         auto env = read_env(cin);
28         cout << tree_eval_env(t, env) << endl;
29     }
30 }
```

Exercici: Z78925 Avaluar expressions binàries amb variables

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_var_name(string s) {
19     if (s.empty()) {
20         return false;
21     }
22     for (char c : s) {
23         if (!isalpha(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: Z78925 Avaluar expressions binàries amb variables

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, * i per tant
    és un nom de variable vàlid. * * Un string buit no és un nom de variable. * *
    @param s String a comprovar * @return `true` si és un nom de variable, `false`
    altrament. */
11 bool is_var_name(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
```