

```

1 #ifndef STRUTTUREDATI_LIST_HPP
2 #define STRUTTUREDATI_LIST_HPP
3
4 /*
5  * Collezione di funzioni ricorsive sulle liste
6  */
7
8 template <class LabelType>
9 struct elem {
10     LabelType label;
11     elem* next;
12 };
13
14 // Lunghezza lista
15 template <class LabelType>
16 int length(elem<LabelType>* p) {
17     if (!p) return 0;
18     return 1 + length(p->next);
19 };
20
21 // Quanti elementi valgono X
22 template <class LabelType>
23 int howMany(elem<LabelType>* p, LabelType x) {
24     if (!p) return 0;
25     return p->label == x + howMany(p->next, x);
26 };
27
28 // X appartiene alla lista
29 template <class LabelType>
30 bool belongs(elem<LabelType>* p, LabelType x) {
31     if (!p) return false;
32     if (p->label == x) return true;
33     belongs(p->next, x);
34 };
35
36 // Inserimento in coda di un valore
37 template <class LabelType>
38 void tailInsert(elem<LabelType>* p, LabelType x) {
39     if (!p) {
40         p = new elem<LabelType>;
41         p->next = nullptr;
42         p->label = x;
43     };
44     tailInsert(p->next, x);
45 };
46
47 // Inserimento in coda di un elemento
48 template <class LabelType>
49 void append(elem<LabelType>* &p, elem<LabelType>* e) {
50     if (!p) p = e;
51     append(p->next, e);
52 };
53
54 // Eliminazione in coda
55 template <class LabelType>
56 void tailDelete(elem<LabelType>* &p) {
57     if (!p) return;
58     if (!p->next) {
59         delete p;
60         p = nullptr;
61     }

```

```
62 |     else tailDelete(p->next);
63 | };
64 |
65 |
66 | #endif //STRUTTUREDATI_LIST_HPP
```

```

1 #ifndef STRUTTUREDATI_BINARYTREE_HPP
2 #define STRUTTUREDATI_BINARYTREE_HPP
3
4
5 #include <iostream>
6
7 template <class LabelType>
8 class BinaryTree {
9     struct Node {
10         LabelType label;
11         Node* left;
12         Node* right;
13     };
14     Node* root;
15
16     void deleteTree(Node*&);
17     void inspectLabel(LabelType&);
18     void _preOrder(Node*);
19     void _inOrder(Node*);
20     void _postOrder(Node*);
21     Node* _findNode(LabelType, Node*);
22     int _nodes(Node*);
23     int _leaves(Node*);
24
25 public:
26     BinaryTree(){ root = nullptr; };
27     ~BinaryTree(){ deleteTree(root); };
28
29     /*
30      * Le operazioni di linearizzazione sono lineari rispetto al numero
31      * dei nodi o esponenziali rispetto al numero dei livelli
32      */
33     void preOrder() { _preOrder(root); };
34     void inOrder() { _inOrder(root); };
35     void postOrder() { _postOrder(root); };
36
37     // Ricerca, inserimento e cancellazione di un nodo sono O(n)
38     Node* findNode(LabelType label) { return _findNode(label, root); };
39     bool insertNode(LabelType, LabelType, char);
40
41     // Operazioni aggiuntive
42     int nodes() { return _nodes(root); };
43     int leaves() { return _leaves(root); };
44 };
45
46
47 template<class LabelType>
48 void BinaryTree<LabelType>::deleteTree(Node* &node) {
49     if (!node) return;
50     deleteTree(node->left);
51     deleteTree(node->right);
52     delete node;
53     node = nullptr;
54 }
55
56 template<class LabelType>
57 void BinaryTree<LabelType>::inspectLabel(LabelType &label) {
58     std::cout << label;
59 }
60
61

```

```

62 template<class LabelType>
63 void BinaryTree<LabelType>::_preOrder(Node* node) {
64     if (!node) return;
65     inspectLabel(node->label);
66     _preOrder(node->left);
67     _preOrder(node->right);
68 }
69
70 template<class LabelType>
71 void BinaryTree<LabelType>::_inOrder(Node* node) {
72     if (!node) return;
73     _inOrder(node->left);
74     inspectLabel(node->label);
75     _inOrder(node->right);
76 }
77
78 template<class LabelType>
79 void BinaryTree<LabelType>::_postOrder(Node* node) {
80     if (!node) return;
81     _postOrder(node->left);
82     _postOrder(node->right);
83     inspectLabel(node->label);
84 }
85
86
87 template<class LabelType>
88 typename BinaryTree<LabelType>::Node* BinaryTree<LabelType>::_findNode(LabelType label,
Node* node) {
89     if (!node) return nullptr;
90     if (label == node->label) return node;
91
92     Node* check = _findNode(label, node->left);
93     if (check) return check;
94     return _findNode(label, node->right);
95 }
96
97 template<class LabelType>
98 bool BinaryTree<LabelType>::insertNode(LabelType new_label, LabelType father_label, char
pos) {
99     if (pos != 'l' && pos != 'r') return false;
100
101     if (!root) {
102         root = new Node;
103         root->label = new_label;
104         root->left = nullptr;
105         root->right = nullptr;
106         return true;
107     }
108
109     Node* father = findNode(father_label);
110     if (!father) return false;
111
112     if (pos == 'l' && !father->left) {
113         father->left = new Node;
114         father->left->label = new_label;
115         father->left->left = nullptr;
116         father->left->right = nullptr;
117         return true;
118     }
119     if (pos == 'r' && !father->right) {
120         father->right = new Node;
121         father->right->label = new_label;

```

```
122     father->right->left = nullptr;
123     father->right->right = nullptr;
124     return true;
125 }
126 return false;
127 }
128
129
130 template<class LabelType>
131 int BinaryTree<LabelType>::_nodes(Node* node) {
132     if (!node) return 0;
133     return 1 + _nodes(node->left) + _nodes(node->right);
134 }
135
136 template<class LabelType>
137 int BinaryTree<LabelType>::_leaves(Node* node) {
138     if (!node) return 0;
139     if (!node->left && !node->right) return 1;
140     return _leaves(node->left) + _leaves(node->right);
141 }
142
143
144 #endif // STRUTTUREDATI_BINARYTREE_HPP
```

```

1 #ifndef STRUTTUREDATI_GENERICTREE_HPP
2 #define STRUTTUREDATI_GENERICTREE_HPP
3
4
5 #include <iostream> // Used for std::cout in inspectLabel function
6
7 /*
8  * Implementazione di un albero generico tramite la rappresentazione binaria
9  * secondo la tecnica figlio-fratello
10 */
11
12 template <class LabelType>
13 class GenericTree {
14     struct Node {
15         LabelType label;
16         Node* left;
17         Node* right;
18     };
19     Node* root;
20
21     void deleteTree(Node*&);
22     void inspectLabel(LabelType&);
23     void _preOrder(Node*);
24     void _inOrder(Node*);
25     int _nodes(Node*);
26     int _leaves(Node*);
27
28 public:
29     GenericTree(){ root = nullptr; };
30     ~GenericTree(){ deleteTree(root); };
31
32     // Le operazioni di linearizzazione sono O(n)
33     // La preOrder dell'albero generico equivale alla preOrder dell'albero binario
34     void preOrder() { _preOrder(root); };
35     // La postOrder dell'albero generico equivale alla inOrder dell'albero binario
36     void postOrder() { _inOrder(root); };
37     // La inOrder per un albero generico non ha senso
38
39     // Ricerca, inserimento e cancellazione di un nodo sono O(n)
40     void addBrother(LabelType, Node*&);
41     bool addSon(LabelType, Node*&);
42
43     // Operazioni aggiuntive
44     int nodes() { return _nodes(root); };
45     int leaves(){ return _leaves(root); }
46 };
47
48
49 template<class LabelType>
50 void GenericTree<LabelType>::deleteTree(Node* &node) {
51     if (!node) return;
52     deleteTree(node->left);
53     deleteTree(node->right);
54     delete node;
55     node = nullptr;
56 }
57
58 template<class LabelType>
59 void GenericTree<LabelType>::inspectLabel(LabelType &label) {
60     std::cout << label;
61 }

```

```

62
63
64 template<class LabelType>
65 void GenericTree<LabelType>::_preOrder(Node* node) {
66     if (!node) return;
67     inspectLabel(node->label);
68     _preOrder(node->left);
69     _preOrder(node->right);
70 }
71
72 template<class LabelType>
73 void GenericTree<LabelType>::_inOrder(Node* node) {
74     if (!node) return;
75     _inOrder(node->left);
76     inspectLabel(node->label);
77     _inOrder(node->right);
78 }
79
80
81 template<class LabelType>
82 void GenericTree<LabelType>::addBrother(LabelType new_label, Node *&brother) {
83     if (!brother) {
84         brother = new Node;
85         brother->label = new_label;
86         brother->left = nullptr;
87         brother->right = nullptr;
88     }
89     else addBrother(new_label, brother->right);
90 }
91
92 template<class LabelType>
93 bool GenericTree<LabelType>::addSon(LabelType new_label, Node *&father) {
94     if (!father) return false;
95     addBrother(new_label, father->left);
96     return true;
97 }
98
99
100 template<class LabelType>
101 int GenericTree<LabelType>::_nodes(Node *node) {
102     if (!node) return 0;
103     return 1 + _nodes(node->left) + _nodes(node->right);
104 }
105
106 template<class LabelType>
107 int GenericTree<LabelType>::_leaves(Node *node) {
108     if (!node) return 0;
109     if (!node->left) return 1 + _leaves(node->right);
110     return _leaves(node->left) + _leaves(node->right);
111 }
112
113
114 #endif // STRUTTUREDATI_GENERICTREE_HPP

```

```

1 #ifndef STRUTTUREDATI_SEARCHTREE_HPP
2 #define STRUTTUREDATI_SEARCHTREE_HPP
3
4
5 #include <iostream>
6
7 template <class LabelType>
8 class SearchTree {
9     struct Node {
10         LabelType label;
11         Node *left;
12         Node *right;
13     };
14     Node *root;
15
16     void deleteTree(Node*&);
17     void inspectLabel(LabelType&);
18     void _preOrder(Node*);
19     void _inOrder(Node*);
20     void _postOrder(Node*);
21     Node *_findNode(LabelType, Node*);
22     void _insertNodeRecursive(LabelType, Node*&);
23     void deleteMin(Node*&, LabelType&);
24     void _deleteNode(LabelType, Node*&);
25     int _height(Node *);
26
27 public:
28     SearchTree(){ root = nullptr; };
29     ~SearchTree(){ deleteTree(root); };
30
31     // Le operazioni di linearizzazione sono O(n)
32     void preOrder() { _preOrder(root); };
33     void inOrder() { _inOrder(root); }; // Produce Le etichette in ordine crescente
34     void postOrder() { _postOrder(root); };
35
36     // Ricerca, inserimento e cancellazione di un nodo sono mediamente O(log(n)) (albero
    bilanciato o quasi)
37     bool findNode(LabelType label) { return _findNode(label, root); };
38     void insertNode(LabelType label) { _insertNodeRecursive(label, root); };
39     void insertNodeIterative(LabelType label);
40     void deleteNode(LabelType label) { _deleteNode(label, root); };
41
42     // Operazioni aggiuntive
43     LabelType min();
44     LabelType max();
45     int height() { return _height(root); };
46     bool isAVL(Node*, int&);
47 };
48
49
50 template<class LabelType>
51 void SearchTree<LabelType>::deleteTree(Node* &node) {
52     if (!node) return;
53     deleteTree(node->left);
54     deleteTree(node->right);
55     delete node;
56     node = nullptr;
57 }
58
59 template<class LabelType>
60 void SearchTree<LabelType>::inspectLabel(LabelType &label) {

```



```

61     std::cout << label << '\t';
62 }
63
64
65 template<class LabelType>
66 void SearchTree<LabelType>::_preOrder(Node* node) {
67     if (!node) return;
68     inspectLabel(node->label);
69     _preOrder(node->left);
70     _preOrder(node->right);
71 }
72
73 template<class LabelType>
74 void SearchTree<LabelType>::_inOrder(Node* node) {
75     if (!node) return;
76     _inOrder(node->left);
77     inspectLabel(node->label);
78     _inOrder(node->right);
79 }
80
81 template<class LabelType>
82 void SearchTree<LabelType>::_postOrder(Node* node) {
83     if (!node) return;
84     _postOrder(node->left);
85     _postOrder(node->right);
86     inspectLabel(node->label);
87 }
88
89
90 template<class LabelType>
91 typename SearchTree<LabelType>::Node *SearchTree<LabelType>::_findNode(LabelType label,
SearchTree::Node *node) {
92     if (!node) return nullptr;
93     if (node->label == label) return node;
94     if (node->label > label) return _findNode(label, node->left);
95     return _findNode(label, node->right);
96 }
97
98 template<class LabelType>
99 void SearchTree<LabelType>::_insertNodeRecursive(LabelType label, Node *&node) {
100     if (!node) {
101         node = new Node;
102         node->label = label;
103         node->left = nullptr;
104         node->right = nullptr;
105         return;
106     }
107     if (node->label >= label) _insertNodeRecursive(label, node->left);
108     if (node->label < label) _insertNodeRecursive(label, node->right);
109 }
110
111 template<class LabelType>
112 void SearchTree<LabelType>::insertNodeIterative(LabelType label) {
113     Node *node = new Node;
114     node->label = label;
115     node->left = nullptr;
116     node->right = nullptr;
117     if (!root) {
118         root = node;
119         return;
120     }

```

```

121
122     Node *aux = root;
123     Node *pre = nullptr;
124
125     // Finchè non sono nel posto giusto per inserire il nuovo elemento
126     while (aux) {
127         pre = aux;
128         if (label <= aux->label) aux = aux->left;
129         else aux = aux->right;
130     }
131
132     if (label <= pre->label) pre->left = node;
133     else pre->right = node;
134 }
135
136 template<class LabelType>
137 void SearchTree<LabelType>::deleteMin(SearchTree::Node *&node, LabelType &label) {
138     // Elimina il nodo più piccolo e ne restituisce l'etichetta
139
140     // Cerca in nodo più piccolo del sottoalbero dato
141     if (node->left) deleteMin(node->left, label);
142     // Rimpiazza l'etichetta
143     label = node->label;
144     Node* aux = node;
145     // Rimpiazza il nodo da eliminare con il suo sottoalbero destro
146     // Questo perchè il sottoalbero sinistro è vuoto per la condizione iniziale
147     node = node->right;
148     delete aux;
149 }
150
151 template<class LabelType>
152 void SearchTree<LabelType>::_deleteNode(LabelType label, SearchTree::Node *&node) {
153     // Ricerca del nodo da eliminare in base alla label
154     if (!node) return;
155     if (label < node->label) _deleteNode(label, node->left);
156     if (label > node->label) _deleteNode(label, node->right);
157
158     // Se il nodo non ha il figlio sinistro allora viene eliminato e gli elementi del
destro scalano al father
159     if (!node->left) {
160         Node* aux = node;
161         node = node->right;
162         delete aux;
163         return;
164     }
165
166     // Se il nodo non ha il figlio destro allora viene eliminato e gli elementi del
sinistro scalano al father
167     if (!node->right) {
168         Node* aux = node;
169         node = node->left;
170         delete aux;
171         return;
172     }
173
174     // Se il nodo ha entrambi i figli prendo l'elemento più piccolo dal sottoalbero di
destra e lo metto al suo posto
175     deleteMin(node->right, node->label);
176 }
177
178
179 template<class LabelType>

```

```

180 LabelType SearchTree<LabelType>::min() {
181     // MIN = primo figlio sinistro che non ha figli sinistri
182     if (!root) return NULL;
183     Node* aux = root;
184     while (aux->left) aux = aux->left;
185     return aux->label;
186 }
187
188 template<class LabelType>
189 LabelType SearchTree<LabelType>::max() {
190     // MAX = primo figlio destro che non ha figli destri
191     if (!root) return NULL;
192     Node* aux = root;
193     while (aux->right) aux = aux->right;
194     return aux->label;
195 }
196
197 template<class LabelType>
198 int SearchTree<LabelType>::_height(SearchTree::Node *node) {
199     if (!node) return 0;
200     int h_left = _height(node->left);
201     int h_right = _height(node->right);
202     return 1 + std::max(h_left, h_right);
203 }
204
205 template<class LabelType>
206 bool SearchTree<LabelType>::isAVL(Node* node, int &maxH) {
207     /*
208      * Un albero AVL è un albero binario in cui l'altezza di ogni
209      * sottoalbero sinistro dista al massimo di uno dall'altezza
210      * di ogni sottoalbero destro
211      */
212
213     // Foglia
214     if (!node) {
215         maxH = 0;
216         return true;
217     }
218
219     int hl, hr;
220     bool avlL, avlR, amIAVL;
221
222     avlL = isAVL(node->left, hl);
223     avlR = isAVL(node->right, hr);
224     amIAVL = std::abs(avlL-avlR) <= 1;
225
226     maxH = std::max(hl, hr) + 1;
227
228     if (avlL && avlR && amIAVL) return true;
229     else return false;
230 }
231
232
233 #endif // STRUTTUREDATI_SEARCHTREE_HPP

```

```

1 #ifndef STRUTTUREDATI_HEAP_HPP
2 #define STRUTTUREDATI_HEAP_HPP
3
4
5 #include <iostream>
6
7 /*
8  * Implementazione di un Max Heap che gestisce interi
9  *
10 * Questa struttura dati è particolarmente indicata per
11 * implementare una coda con priorità
12 */
13
14 class Heap {
15
16 protected:
17     int *arr;
18     int dim; // Dimensione fisica dell'array che rappresento lo heap
19     int last; // Indice dell'ultimo elemento dello heap
20
21     void exchange(int i, int j);
22     void up(int);
23     void down(int);
24
25     static inline int father(int i) { return std::floor((i - 1) / 2); };
26     static inline int left_son(int i) { return 2*i+1; };
27     static inline int right_son(int i) { return 2*i+2; };
28     static inline bool is_first_child(int i);
29
30 public:
31     explicit Heap(int);
32     ~Heap() { delete[] arr; };
33     void insert(int);
34     int extract();
35     void print();
36 };
37
38
39 Heap::Heap(int n) {
40     dim = n;
41     last = -1;
42     arr = new int[dim];
43 }
44
45 inline void Heap::exchange(int i, int j) {
46     int tmp = arr[i];
47     arr[i] = arr[j];
48     arr[j] = tmp;
49 }
50
51 void Heap::up(int i) {
52     /*
53      * Funzione che fa risalire l'elemento i per rispettare le proprietà dello heap
54      * La complessità di questa funzione è O(Log(n)) perchè ogni chiamata risale di un
55      livello
56      */
57     if (i <= 0 || arr[i] <= arr[father(i)]) return;
58     exchange(i, father(i));
59     up(father(i));
60 }

```

```

61 void Heap::down(int i) {
62     /*
63      * Funzione che fa scendere l'elemento i per rispettare le proprietà dello heap
64      * La complessità di questa funzione è  $O(\log(n))$  perchè ogni chiamata risale di un
        Livello
65      */
66     int son = left_son(i);
67
68     // Se il primo figlio di i è l'ultimo dell'array allora i ha un solo figlio
69     if (son == last) {
70         // Lo scambia col padre se è maggiore
71         if (arr[son] > arr[i]) {
72             exchange(i, son);
73         }
74     }
75
76     // Se i non ha figli non fa nulla
77     // Resta solo il caso in cui i abbia due figli
78     else if (son < last) {
79         // Prende il figlio maggiore
80         if (arr[son] < arr[son+1]) son++;
81         // Lo scambia col padre se è maggiore
82         if (arr[son] > arr[i]) {
83             exchange(i, son);
84             // Controlla se è necessario far scendere ancora il padre
85             down(son);
86         }
87     }
88 }
89
90 void Heap::insert(int n) {
91     /*
92      * Funzione che inserisce un elemento nello heap se non è pieno
93      *
94      * Memorizza l'elemento della prima posizione libera dell'array
95      * Lo fa risalire fino a quando non rispetta le proprietà dello heap
96      */
97     if (last >= dim) {
98         std::cerr << "\nImpossibile inserire un nuovo elemento: heap pieno";
99         return;
100     }
101     arr[++last] = n;
102     up(last);
103 }
104
105 int Heap::extract() {
106     /*
107      * Funzione che estrae l'elemento maggiore dallo heap (radice)
108      *
109      * Viene estratta la radice e l'ultimo elemento viene messo al suo posto,
110      * facendolo scendere finchè non rispetta le proprietà dello heap
111      */
112     int root = arr[0];
113     arr[0] = arr[last--];
114     down(0);
115     return root;
116 }
117
118 inline bool Heap::is_first_child(int i) {
119     /*
120      * Funzione che riconosce quando un nodo è il primo figlio di un livello

```

```

121     *
122     * Il primo nodo di ogni livello per costruzione ha indice di una potenza di due
123     * Un indice è potenza di due se e solo se la sua rappresentazione binaria ha 1
124     * nella posizione i e tutti zeri, questa relazione è vera se e solo se i è una
125     * potenza di due poichè il numero precedente ad una potenza di due ha 1 in ogni
126     * bit tranne che in quello della potenza di due successiva
127     */
128
129     if ( (i != 0) && (i & (i-1)) == 0) return true;
130     return false;
131 }
132
133 void Heap::print() {
134     /*
135     * Funzione che stampa lo heap per livelli
136     */
137
138     for (int i = 0; i < last; ++i) {
139         if (is_first_child(i+1)) std::cout << '\n';
140         std::cout << arr[i] << '\t';
141     }
142 }
143
144
145 #endif //STRUTTUREDATI_HEAP_HPP

```

```

1 #ifndef STRUTTUREDATI_HASH_HPP
2 #define STRUTTUREDATI_HASH_HPP
3
4 #include <iostream>
5 // #include <cstring> Funzione memset
6
7 /*
8  * Implementazione tramite liste di trabocco
9  */
10
11 class Hash {
12     /*
13      * Per gestire le eliminazioni in modo più efficiente
14      * è possibile utilizzare qui un doppio puntatore
15      */
16     struct Elem {
17         int key;
18         Elem *next;
19         explicit Elem(int _key) : key(_key), next(nullptr) {};
20     };
21
22     Elem **table;
23     // Una buona size si rivela essere circa il doppio degli elementi che intendo mantenere
24     int size;
25
26     int hash(int) const;
27 public:
28     explicit Hash(int);
29     ~Hash();
30     bool find(int);
31     void insert(int);
32     void print();
33     void printOccupancy();
34 };
35
36 inline Hash::Hash(int _size) {
37     size = _size;
38     table = new Elem*[size];
39     for (int i = 0; i < size; ++i) {
40         table[i] = nullptr;
41     }
42 }
43
44 inline Hash::~~Hash() {
45     for (int i = 0; i < size; ++i) {
46         Elem* aux = table[i];
47         Elem* del;
48         while (aux) {
49             del = aux;
50             aux = aux->next;
51             delete del;
52         }
53     }
54     delete []table;
55 }
56
57 int Hash::hash(int key) const {
58     /*
59      * Nel caso in cui dovessi fare hash di stringhe una buona
60      * funzione hash è quella index = (index + key[i]) % size;
61      * che per ogni carattere lo somma all'indice facendo il resto

```

```

62     */
63     return key % size;
64 }
65
66 bool Hash::find(int key) {
67     int index = hash(key);
68     Elem* aux = table[index];
69     while (aux) {
70         if (aux->key == key) return true;
71         aux = aux->next;
72     }
73     return false;
74 }
75
76 void Hash::insert(int key) {
77     int index = hash(key);
78     Elem* elem = new Elem(key);
79     elem->next = table[index];
80     table[index] = elem;
81 }
82
83 void Hash::print() {
84     for (int i = 0; i < size; ++i) {
85         Elem* aux = table[i];
86         while (aux) {
87             std::cout << aux->key << '\n';
88             aux = aux->next;
89         }
90     }
91 }
92
93 void Hash::printOccupancy() {
94     for (int i = 0; i < size; ++i) {
95         Elem* aux = table[i];
96         std::cout << '\n' << i << ") ";
97         while (aux) {
98             std::cout << '*';
99             aux = aux->next;
100         }
101     }
102 }
103
104
105 #endif //STRUTTUREDATI_HASH_HPP

```



```

1 #ifndef STRUTTUREDATI_GRAPH_HPP
2 #define STRUTTUREDATI_GRAPH_HPP
3
4
5 #include <iostream>
6
7 /*
8  * Implementazione di un grafo orientato tramite liste di adiacenza
9  */
10
11 template <class LabelType>
12 class Graph {
13     struct Node {
14         int index;
15         Node* next;
16     };
17
18     const int N;
19     Node** graph;
20     LabelType* labels;
21     bool* mark;
22
23     void nodeVisit(int);
24     void inspectLabel(LabelType);
25
26 public:
27     explicit Graph(int N);
28     void depthVisit();
29 };
30
31 template <class LabelType>
32 Graph<LabelType>::Graph(int n) : N(n) {
33     graph = new Node*[N];
34     labels = new LabelType[N];
35     mark = new bool[N];
36 }
37
38 template <class LabelType>
39 void Graph<LabelType>::inspectLabel(LabelType label) {
40     std::cout << label << '\n';
41 }
42
43 template <class LabelType>
44 void Graph<LabelType>::nodeVisit(int index) {
45     mark[index] = true;
46     inspectLabel(labels[index]);
47
48     Node* aux;
49     int j;
50     for (aux = graph[index]; aux; aux = aux->next) {
51         j = aux->index;
52         if (!mark[j]) nodeVisit(j);
53     }
54 }
55
56 template <class LabelType>
57 void Graph<LabelType>::depthVisit() {
58
59     // Complessità O(Nodi + Archi)
60     for (int i = 0; i < N; ++i) {
61         mark[i] = false;

```

```
62     }
63     for (int i = 0; i < N; ++i) {
64         if (!mark[i]) nodeVisit(i);
65     }
66 }
67
68 #endif //STRUTTUREDATI_GRAPH_HPP
```