```cpp
#ifndef STRUTTUREDATI_LIST_HPP
#define STRUTTUREDATI_LIST_HPP

/*
 * Collezione di funzioni ricorsive sulle liste
 */

template <class LabelType>
struct elem {
    LabelType label;
    elem* next;
};

// Lunghezza lista
template <class LabelType>
int length(elem<LabelType>* p) {
    if (!p) return 0;
    return 1 + length(p->next);
};

// Quanti elementi valgono X
template <class LabelType>
int howMany(elem<LabelType>* p, LabelType x) {
    if (!p) return 0;
    return p->label == x + howMany(p->next, x);
};

// X appartiene alla lista
template <class LabelType>
bool belongs(elem<LabelType>* p, LabelType x) {
    if (!p) return false;
    if (p->label == x) return true;
    belongs(p->next, x);
};

// Inserimento in coda di un valore
template <class LabelType>
void tailInsert(elem<LabelType>* p, LabelType x) {
    if (!p) {
        p = new elem<LabelType>;
        p->next = nullptr;
        p->label = x;
    };
    tailInsert(p->next, x);
};

// Inserimento in coda di un elemento
template <class LabelType>
void append(elem<LabelType>* &p, elem<LabelType>* e) {
    if (!p) p = e;
    append(p->next, e);
};

// Eliminazione in coda
template <class LabelType>
void tailDelete(elem<LabelType>* &p) {
    if (!p) return;
    if (!p->next) {
        delete p;
        p = nullptr;
    }
```

```
    else tailDelete(p->next);
};


#endif //STRUTTUREDATI_LIST_HPP
```

```cpp
#ifndef STRUTTUREDATI_BINARYTREE_HPP
#define STRUTTUREDATI_BINARYTREE_HPP


#include <iostream>

template <class LabelType>
class BinaryTree {
    struct Node {
        LabelType label;
        Node* left;
        Node* right;
    };
    Node* root;

    void deleteTree(Node*&);
    void inspectLabel(LabelType&);
    void _preOrder(Node*);
    void _inOrder(Node*);
    void _postOrder(Node*);
    Node* _findNode(LabelType, Node*);
    int _nodes(Node*);
    int _leaves(Node*);

public:
    BinaryTree(){ root = nullptr; };
    ~BinaryTree(){ deleteTree(root); };

    /*
     * Le operazioni di linearizzazione sono lineari rispetto al numero
     * dei nodi o esponenziali rispetto al numero dei livelli
     */
    void preOrder() { _preOrder(root); };
    void inOrder() { _inOrder(root); };
    void postOrder() { _postOrder(root); };

    // Ricerca, inserimento e cancellazione di un nodo sono O(n)
    Node* findNode(LabelType label) { return _findNode(label, root); };
    bool insertNode(LabelType, LabelType, char);

    // Operazioni aggiuntive
    int nodes() { return _nodes(root); };
    int leaves() { return _leaves(root); };
};


template<class LabelType>
void BinaryTree<LabelType>::deleteTree(Node* &node) {
    if (!node) return;
    deleteTree(node->left);
    deleteTree(node->right);
    delete node;
    node = nullptr;
}

template<class LabelType>
void BinaryTree<LabelType>::inspectLabel(LabelType &label) {
    std::cout << label;
}


```

```cpp
template<class LabelType>
void BinaryTree<LabelType>::_preOrder(Node* node) {
    if (!node) return;
    inspectLabel(node->label);
    _preOrder(node->left);
    _preOrder(node->right);
}

template<class LabelType>
void BinaryTree<LabelType>::_inOrder(Node* node) {
    if (!node) return;
    _inOrder(node->left);
    inspectLabel(node->label);
    _inOrder(node->right);
}

template<class LabelType>
void BinaryTree<LabelType>::_postOrder(Node* node) {
    if (!node) return;
    _postOrder(node->left);
    _postOrder(node->right);
    inspectLabel(node->label);
}


template<class LabelType>
typename BinaryTree<LabelType>::Node* BinaryTree<LabelType>::_findNode(LabelType label, Node* node) {
    if (!node) return nullptr;
    if (label == node->label) return node;

    Node* check = _findNode(label, node->left);
    if (check) return check;
    return _findNode(label, node->right);
}

template<class LabelType>
bool BinaryTree<LabelType>::insertNode(LabelType new_label, LabelType father_label, char pos) {
    if (pos != 'l' && pos != 'r') return false;

    if (!root) {
        root = new Node;
        root->label = new_label;
        root->left = nullptr;
        root->right = nullptr;
        return true;
    }

    Node* father = findNode(father_label);
    if (!father) return false;

    if (pos == 'l' && !father->left) {
        father->left = new Node;
        father->left->label = new_label;
        father->left->left = nullptr;
        father->left->right = nullptr;
        return true;
    }
    if (pos == 'r' && !father->right) {
        father->right = new Node;
        father->right->label = new_label;
```

```
        father->right->left = nullptr;
        father->right->right = nullptr;
        return true;
    }
    return false;
}


template<class LabelType>
int BinaryTree<LabelType>::_nodes(Node* node) {
    if (!node) return 0;
    return 1 + _nodes(node->left) + _nodes(node->right);
}

template<class LabelType>
int BinaryTree<LabelType>::_leaves(Node* node) {
    if (!node) return 0;
    if (!node->left && !node->right) return 1;
    return _leaves(node->left) + _leaves(node->right);
}


#endif // STRUTTUREDATI_BINARYTREE_HPP
```

```cpp
#ifndef STRUTTUREDATI_GENERICTREE_HPP
#define STRUTTUREDATI_GENERICTREE_HPP


#include <iostream> // Used for std::cout in inspectLabel function

/*
 * Implementazione di un albero generico tramite la rappresentazione binaria
 *  secondo la tecnica figlio-fratello
 */

template <class LabelType>
class GenericTree {
    struct Node {
        LabelType label;
        Node* left;
        Node* right;
    };
    Node* root;

    void deleteTree(Node*&);
    void inspectLabel(LabelType&);
    void _preOrder(Node*);
    void _inOrder(Node*);
    int _nodes(Node*);
    int _leaves(Node*);

public:
    GenericTree(){ root = nullptr; };
    ~GenericTree(){ deleteTree(root); };

    // Le operazioni di linearizzazione sono O(n)
        // La preOrder dell'albero generico equivale alla preorder dell'albero binario
    void preOrder() { _preOrder(root); };
        // La postOrder dell'albero generico equivale alla inorder dell'albero binario
    void postOrder() { _inOrder(root); };
        // La inOrder per un albero generico non ha senso

    // Ricerca, inserimento e cancellazione di un nodo sono O(n)
    void addBrother(LabelType, Node*&);
    bool addSon(LabelType, Node*&);

    // Operazioni aggiuntive
    int nodes() { return _nodes(root); };
    int leaves(){ return _leaves(root); }
};


template<class LabelType>
void GenericTree<LabelType>::deleteTree(Node* &node) {
    if (!node) return;
    deleteTree(node->left);
    deleteTree(node->right);
    delete node;
    node = nullptr;
}

template<class LabelType>
void GenericTree<LabelType>::inspectLabel(LabelType &label) {
    std::cout << label;
}
```

```cpp


template<class LabelType>
void GenericTree<LabelType>::_preOrder(Node* node) {
    if (!node) return;
    inspectLabel(node->label);
    _preOrder(node->left);
    _preOrder(node->right);
}

template<class LabelType>
void GenericTree<LabelType>::_inOrder(Node* node) {
    if (!node) return;
    _inOrder(node->left);
    inspectLabel(node->label);
    _inOrder(node->right);
}


template<class LabelType>
void GenericTree<LabelType>::addBrother(LabelType new_label, Node *&brother) {
    if (!brother) {
        brother = new Node;
        brother->label = new_label;
        brother->left = nullptr;
        brother->right = nullptr;
    }
    else addBrother(new_label, brother->right);
}

template<class LabelType>
bool GenericTree<LabelType>::addSon(LabelType new_label, Node *&father) {
    if (!father) return false;
    addBrother(new_label, father->left);
    return true;
}


template<class LabelType>
int GenericTree<LabelType>::_nodes(Node *node) {
    if (!node) return 0;
    return 1 + _nodes(node->left) + _nodes(node->right);
}

template<class LabelType>
int GenericTree<LabelType>::_leaves(Node *node) {
    if (!node) return 0;
    if (!node->left) return 1 + _leaves(node->right);
    return _leaves(node->left) + _leaves(node->right);
}


#endif // STRUTTUREDATI_GENERICTREE_HPP
```

```cpp
#ifndef STRUTTUREDATI_SEARCHTREE_HPP
#define STRUTTUREDATI_SEARCHTREE_HPP


#include <iostream>

template <class LabelType>
class SearchTree {
    struct Node {
        LabelType label;
        Node *left;
        Node *right;
    };
    Node *root;

    void deleteTree(Node*&);
    void inspectLabel(LabelType&);
    void _preOrder(Node*);
    void _inOrder(Node*);
    void _postOrder(Node*);
    Node *_findNode(LabelType, Node*);
    void _insertNodeRecursive(LabelType, Node*&);
    void deleteMin(Node*&, LabelType&);
    void _deleteNode(LabelType, Node*&);
    int _height(Node *);

public:
    SearchTree(){ root = nullptr; };
    ~SearchTree(){ deleteTree(root); };

    // Le operazioni di linearizzazione sono O(n)
    void preOrder() { _preOrder(root); };
    void inOrder() { _inOrder(root); }; // Produce le etichette in ordine crescente
    void postOrder() { _postOrder(root); };

    // Ricerca, inserimento e cancellazione di un nodo sono mediamente O(log(n)) (albero bilanciato o quasi)
    bool findNode(LabelType label) { return _findNode(label, root); };
    void insertNode(LabelType label) { _insertNodeRecursive(label, root); };
    void insertNodeIterative(LabelType label);
    void deleteNode(LabelType label) { _deleteNode(label, root); };

    // Operazioni aggiuntive
    LabelType min();
    LabelType max();
    int height() { return _height(root); };
    bool isAVL(Node*, int&);
};


template<class LabelType>
void SearchTree<LabelType>::deleteTree(Node* &node) {
    if (!node) return;
    deleteTree(node->left);
    deleteTree(node->right);
    delete node;
    node = nullptr;
}

template<class LabelType>
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     // Rimpiazzo 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>
```

```cpp
LabelType SearchTree<LabelType>::min() {
    // MIN = primo figlio sinistro che non ha figli sinistri
    if (!root) return NULL;
    Node* aux = root;
    while (aux->left) aux = aux->left;
    return aux->label;
}

template<class LabelType>
LabelType SearchTree<LabelType>::max() {
    // MAX = primo figlio destro che non ha figli destri
    if (!root) return NULL;
    Node* aux = root;
    while (aux->right) aux = aux->right;
    return aux->label;
}

template<class LabelType>
int SearchTree<LabelType>::_height(SearchTree::Node *node) {
    if (!node) return 0;
    int h_left = _height(node->left);
    int h_right = _height(node->right);
    return 1 + std::max(h_left, h_right);
}

template<class LabelType>
bool SearchTree<LabelType>::isAVL(Node* node, int &maxH) {
    /*
     * Un albero AVL è un albero binario in cui l'altezza di ogni
     * sottoalbero sinistro dista al massimo di uno dall'altezza
     * di ogni sottoalbero destro
     */

    // Foglia
    if (!node) {
        maxH = 0;
        return true;
    }

    int hl, hr;
    bool avlL, avlR, amIAVL;

    avlL = isAVL(node->left, hl);
    avlR = isAVL(node->right, hr);
    amIAVL = std::abs(avlL-avlR) <= 1;

    maxH = std::max(hl, hr) + 1;

    if (avlL && avlR && amIAVL) return true;
    else return false;
}


#endif // STRUTTUREDATI_SEARCHTREE_HPP
```

```cpp
#ifndef STRUTTUREDATI_HEAP_HPP
#define STRUTTUREDATI_HEAP_HPP


#include <iostream>

/*
 * Implementazione di un Max Heap che gestisce interi
 *
 * Questa struttura dati è particolarmente indicata per
 *  implementare una coda con priorità
 */

class Heap {

protected:
    int *arr;
    int dim; // Dimensione fisica dell'array che rappresento lo heap
    int last; // Indice dell'ultimo elemento dello heap

    void exchange(int i, int j);
    void up(int);
    void down(int);

    static inline int father(int i) { return std::floor((i - 1) / 2); };
    static inline int left_son(int i) { return 2*i+1; };
    static inline int right_son(int i) { return 2*i+2; };
    static inline bool is_first_child(int i);

public:
    explicit Heap(int);
    ~Heap() { delete[] arr; };
    void insert(int);
    int extract();
    void print();
};


Heap::Heap(int n) {
    dim = n;
    last = -1;
    arr = new int[dim];
}

inline void Heap::exchange(int i, int j) {
    int tmp = arr[i];
    arr[i] = arr[j];
    arr[j] = tmp;
}

void Heap::up(int i) {
    /*
     * Funzione che fa risalire l'elemento i per rispettare le proprietà dello heap
     * La complessità di questa funzione è O(log(n)) perchè ogni chiamata risale di un livello
     */
    if (i <= 0 || arr[i] <= arr[father(i)]) return;
    exchange(i, father(i));
    up(father(i));
}

```

```cpp
void Heap::down(int i) {
    /*
     * Funzione che fa scendere l'elemento i per rispettare le proprietà dello heap
     * La complessità di questa funzione è O(log(n)) perchè ogni chiamata risale di un livello
     */
    int son = left_son(i);

    // Se il primo figlio di i è l'ultimo dell'array allora i ha un solo figlio
    if (son == last) {
        // Lo scambia col padre se è maggiore
        if (arr[son] > arr[i]) {
            exchange(i, son);
        }
    }

    // Se i non ha figli non fa nulla
    // Resta solo il caso in cui i abbia due figli
    else if (son < last) {
        // Prende il figlio maggiore
        if (arr[son] < arr[son+1]) son++;
        // Lo scambia col padre se è maggiore
        if (arr[son] > arr[i]) {
            exchange(i, son);
            // Controlla se è necessario far scendere ancora il padre
            down(son);
        }
    }
}

void Heap::insert(int n) {
    /*
     * Funzione che inserisce un elemento nello heap se non è pieno
     *
     * Memorizza l'elemento della prima posizione libera dell'array
     * Lo fa risalire fino a quando non rispetta le proprietà dello heap
     */
    if (last >= dim) {
        std::cerr << "\nImpossibile inserire un nuovo elemento: heap pieno";
        return;
    }
    arr[++last] = n;
    up(last);
}

int Heap::extract() {
    /*
     * Funzione che estrae l'elemento maggiore dallo heap (radice)
     *
     * Viene estratta la radice e l'ultimo elemento viene messo al suo posto,
     * facendolo scendere finchè non rispetta le proprietà dello heap
     */
    int root = arr[0];
    arr[0] = arr[last--];
    down(0);
    return root;
}

inline bool Heap::is_first_child(int i) {
    /*
     * 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 potenzza 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
```

```
#ifndef STRUTTUREDATI_HASH_HPP
#define STRUTTUREDATI_HASH_HPP

#include <iostream>
//#include <cstring> Funzione memset

/*
 * Implementazione tramite liste di trabocco
 */

class Hash {
    /*
     * Per gestire le eliminazioni in modo più efficiente
     *  è possibile utilizzare qui un doppio puntatore
     */
    struct Elem {
        int key;
        Elem *next;
        explicit Elem(int _key) : key(_key), next(nullptr) {};
    };

    Elem **table;
    // Una buona size si rivela essere circa il doppio degli elementi che intendo mantenere
    int size;

    int hash(int) const;
public:
    explicit Hash(int);
    ~Hash();
    bool find(int);
    void insert(int);
    void print();
    void printOccupancy();
};

inline Hash::Hash(int _size) {
    size = _size;
    table = new Elem*[size];
    for (int i = 0; i < size; ++i) {
        table[i] = nullptr;
    }
}

inline Hash::~Hash() {
    for (int i = 0; i < size; ++i) {
        Elem* aux = table[i];
        Elem* del;
        while (aux) {
            del = aux;
            aux = aux->next;
            delete del;
        }
    }
    delete []table;
}

int Hash::hash(int key) const {
    /*
     * Nel caso in cui dovessi fare hash di stringhe una buona
     *  funzione hash è quella index = (index + key[i]) % size;
     *  che per ogni carattere lo somma all'indice facendo il resto
```

```
     */
    return key % size;
}

bool Hash::find(int key) {
    int index = hash(key);
    Elem* aux = table[index];
    while (aux) {
        if (aux->key == key) return true;
        aux = aux->next;
    }
    return false;
}

void Hash::insert(int key) {
    int index = hash(key);
    Elem* elem = new Elem(key);
    elem->next = table[index];
    table[index] = elem;
}

void Hash::print() {
    for (int i = 0; i < size; ++i) {
        Elem* aux = table[i];
        while (aux) {
            std::cout << aux->key << '\n';
            aux = aux->next;
        }
    }
}

void Hash::printOccupancy() {
    for (int i = 0; i < size; ++i) {
        Elem* aux = table[i];
        std::cout << '\n' << i << ") ";
        while (aux) {
            std::cout << '*';
            aux = aux->next;
        }
    }
}


#endif //STRUTTUREDATI_HASH_HPP
```

```cpp
#ifndef STRUTTUREDATI_GRAPH_HPP
#define STRUTTUREDATI_GRAPH_HPP


#include <iostream>

/*
 * Implementazione di un grafo orientato tramite liste di adiacenza
 */

template <class LabelType>
class Graph {
    struct Node {
        int index;
        Node* next;
    };

    const int N;
    Node** graph;
    LabelType* labels;
    bool* mark;

    void nodeVisit(int);
    void inspectLabel(LabelType);

public:
    explicit Graph(int N);
    void depthVisit();
};

template <class LabelType>
Graph<LabelType>::Graph(int n) : N(n) {
    graph = new Node*[N];
    labels = new LabelType[N];
    mark = new bool[N];
}

template <class LabelType>
void Graph<LabelType>::inspectLabel(LabelType label) {
    std::cout << label << '\n';
}

template <class LabelType>
void Graph<LabelType>::nodeVisit(int index) {
    mark[index] = true;
    inspectLabel(labels[index]);

    Node* aux;
    int j;
    for (aux = graph[index]; aux; aux = aux->next) {
        j = aux->index;
        if (!mark[j]) nodeVisit(j);
    }
}

template <class LabelType>
void Graph<LabelType>::depthVisit() {

    // Complessità O(Nodi + Archi)
    for (int i = 0; i < N; ++i) {
        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
```