ALGORITMI E STRUTTURE DATI

Corso di studio: Informatica (D.M. 270/04)
A.A. 2012-2013

- Documentazione degli Abstract Data Types

- Implementazione degli algoritmi

- Esercizi e problemi risolti

A cura di:
Gianvito Taneburgo
Mat. 587645

Note:

- *Per ogni ADT è stata fornita più di una realizzazione. Il "tester" abbinato a ciascun ADT è stato provato, senza venir modificato, con tutte le realizzazioni, producendo il medesimo output;*

- *Per ogni ADT è stato implementato il costruttore di copia ed i seguenti operatori:*

  - ✓ *Operatore =*
  - ✓ *Operatore <<*
  - ✓ *Operatore ==*
  - ✓ *Operatore !=*

- *Gli ADT realizzati con strutture di natura statica (vettori, matrici, ecc…) sono stati forniti di metodi per il raddoppiamento di tali strutture, superando così il limite sul numero di elementi inseribili;*

- *Tutto il codice presente nella documentazione è stato compilato correttamente sull'IDE Eclipse con il compilatore MinGW, impostando i parametri di compilazione –Werror, -Wall e –Wextra (ISO/IEC 14882:1998);*

- *Alcuni programmi e ADT utilizzano delle strutture di supporto (e relativi operatori/iteratori) e/o algoritmi forniti dalla Standard Template Library (STL) del C++;*

- *Il codice, non utilizzando librerie proprie di alcun sistema operativo, è portabile;*

Indice:

ADT

Algoritmi

Esercizi e problemi risolti

Liste – Classe astratta

```
#ifndef _LINLIST_H
#define _LINLIST_H

#include <iostream>
#include <stdexcept>
#include <vector>

using std::cout;
using std::endl;
using std::ostream;
using std::vector;

template<class T, class P>
class Linear_list
{
public:

	typedef T value_type;
	typedef P position;

	virtual ~Linear_list()
	{
	}

	virtual void create() = 0; // create the list
	virtual bool empty() const = 0; // true if the list is empty
	virtual value_type read(position) const = 0; // read the element in the position
	virtual void write(const value_type&, position) = 0; // write the element at the position
	virtual position begin() const = 0; // return the position of the first element of the list
	virtual bool end(position) const = 0; // true if the position is the last of the list
	virtual position next(position) const = 0; // return the next position
	virtual position previous(position) const = 0; // return the previous position
	virtual void insert(const value_type&, position) = 0; // add an element before the position
	virtual void erase(position) = 0; // erase the element in the position

	bool operator ==(const Linear_list<T, P>&) const;
	bool operator !=(const Linear_list<T, P>&) const;

	position endnode() const; // return the position of the last element of the list
	int size() const; // return the number of element in the list
	void push_front(const value_type&); // insert a new element at the beginning
	void push_back(const value_type&); // insert a new element at the end
	void pop_front(); // remove the first element
	void pop_back(); // remove the last element
	void clear(); // erase all the elements

	bool findOrd(const value_type&) const; // search an element in an orderly list
	void insOrd(const value_type&); // insert an element in an orderly list in the right position
	void merge(const Linear_list<T, P>&, const Linear_list<T, P>&); // merge two orderly list
	void clean(); // erase duplicates
	bool find(const value_type&) const; // true if the element is in the list
	void add_new(const value_type&); // add the element only if it isn't already present in the
list
	void join(const Linear_list<T, P>&, const Linear_list<T, P>&); // join two lists
	void sort(); // selection sort on the element of the list
	void reverse(); // change the order of the element
	void common(const Linear_list<T, P>&, const Linear_list<T, P>&); // search the common
elements in two lists
```

```
	void cleanVal(const value_type&); // erase all the occurrences of the element from the list
};

template<class T, class P>
ostream& operator <<(ostream& os, const Linear_list<T, P>& l)
{
	if (l.empty())
		os << "Empty List" << endl;
	else
	{
		typename Linear_list<T, P>::position p = l.begin();
		os << "[ ";
		while (!l.end(p))
		{
			if (p != l.begin())
				os << ", " << l.read(p);
			else
				os << l.read(p);

			p = l.next(p);
		}
		os << " ]" << endl;
	}

	return os;
}

template<class T, class P>
bool Linear_list<T, P>::operator ==(const Linear_list<T, P>& list2) const
{
	if (list2.size() != this->size())
		return false;

	position p = this->begin();
	position p2 = list2.begin();

	for (int i = 0; i < this->size(); i++)
	{
		if (this->read(p) != list2.read(p2))
			return false;
		else
		{
			p = this->next(p);
			p2 = list2.next(p2);
		}
	}

	return true;
}

template<class T, class P>
bool Linear_list<T, P>::operator !=(const Linear_list<T, P>& list2) const
{
	return (!(*this == list2));
}

template<class T, class P>
typename Linear_list<T, P>::position Linear_list<T, P>::endnode() const
{
	position p = this->begin();

	while (!this->end(this->next(p)))
```

```
		p = this->next(p);

	return p;
}

template<class T, class P>
int Linear_list<T, P>::size() const
{
	int len = 0;

	if (!this->empty())
	{
		position p = this->begin();
		len++;

		while (p != this->endnode())
		{
			len++;
			p = this->next(p);
		}
	}

	return len;
}

template<class T, class P>
void Linear_list<T, P>::push_front(const value_type& el)
{
	this->insert(el, this->begin());
}

template<class T, class P>
void Linear_list<T, P>::push_back(const value_type& el)
{
	this->insert(el, this->next(this->endnode()));
}

template<class T, class P>
void Linear_list<T, P>::pop_front()
{
	this->erase(this->begin());
}

template<class T, class P>
void Linear_list<T, P>::pop_back()
{
	this->erase(this->endnode());
}

template<class T, class P>
void Linear_list<T, P>::clear()
{
	while (!this->empty())
		this->pop_front();
}

template<class T, class P>
bool Linear_list<T, P>::findOrd(const value_type& el) const
{
	bool flag = false;
	int size = this->size();
```

```
	position p = this->begin();

	for (int i = 0; (i < size && flag == false && this->read(p) <= el); i++)
	{
		if (this->read(p) == el)
			flag = true;

		p = this->next(p);
	}

	return flag;
}

template<class T, class P>
void Linear_list<T, P>::insOrd(const value_type& el)
{
	position p = this->endnode();

	if (this->empty())
		this->insert(el, this->begin());
	else if (el >= this->read(p))
		this->push_back(el);
	else
	{
		position p = this->begin();
		while (el > this->read(p) && !this->end(p))
			p = this->next(p);

		this->insert(el, p);
	}
}

template<class T, class P>
void Linear_list<T, P>::merge(const Linear_list<T, P>& list1, const Linear_list<T, P>& list2)
{
	for (position p = list1.begin(); !list1.end(p); p = list1.next(p))
		this->push_back(list1.read(p));

	for (position p = list2.begin(); !list2.end(p); p = list2.next(p))
		this->insOrd(list2.read(p));
}

template<class T, class P>
void Linear_list<T, P>::clean()
{
	if (this->empty())
		throw std::logic_error("Linear List (exception) - Unable to erase duplicates (empty
list)");

	position p = this->begin();
	position r;
	position q;

	while (!this->end(p))
	{
		q = this->next(p);
		while (!this->end(q))
		{
			if (this->read(p) == this->read(q))
			{
				r = this->previous(q);
				this->erase(q);
```

```
                                q = r;
                        }

                        q = this->next(q);
                }
                p = this->next(p);
        }
}

template<class T, class P>
bool Linear_list<T, P>::find(const value_type& el) const
{
        bool flag = false;

        if (!this->empty())
                for (Linear_list<T, P>::position p = this->begin(); !this->end(p); p = this->next(p))
                        if (el == this->read(p))
                                flag = true;

        return flag;
}

template<class T, class P>
void Linear_list<T, P>::add_new(const value_type& el)
{
        if (!this->find(el))
                this->push_back(el);
}

template<class T, class P>
void Linear_list<T, P>::join(const Linear_list<T, P>& list1, const Linear_list<T, P>& list2)
{
        for (position p = list1.begin(); !list1.end(p); p = list1.next(p))
                this->push_back(list1.read(p));

        for (position p = list2.begin(); !list2.end(p); p = list2.next(p))
                this->push_back(list2.read(p));
}

template<class T, class P>
void Linear_list<T, P>::sort()
{
        if (!this->empty())
        {
                int dim = this->size();
                value_type* elements = new value_type[dim];
                value_type temp;
                int i = 0;

                for (position p = this->begin(); !this->end(p); p = this->next(p))
                {
                        elements[i] = this->read(p);
                        i++;
                }

                //Selection sort
                int j, imin = 0;
                for (i = 0; i < dim - 1; i++)
                {
                        imin = i;
                        for (j = i + 1; j < dim; j++)
                                if (elements[j] < elements[imin])
```

```
                                    imin = j;

                        temp = elements[i];
                        elements[i] = elements[imin];
                        elements[imin] = temp;
                }

                i = 0;
                for (position p = this->begin(); !this->end(p); p = this->next(p))
                {
                        this->write(elements[i], p);
                        i++;
                }
        }
}

template<class T, class P>
void Linear_list<T, P>::reverse()
{
        if (!this->empty())
        {
                int dim = this->size();
                dim = dim / 2;
                position p1 = this->begin();
                position p2 = this->endnode();
                value_type temp;

                for (int i = 0; i < dim; i++)
                {
                        temp = this->read(p1);
                        this->write(this->read(p2), p1);
                        this->write(temp, p2);
                        p1 = this->next(p1);
                        p2 = this->previous(p2);
                }
        }
}

template<class T, class P>
void Linear_list<T, P>::common(const Linear_list<T, P>& list1, const Linear_list<T, P>& list2)
{
        for (position p = list1.begin(); !list1.end(p); p = list1.next(p))
                if ((list2.find(list1.read(p)) && !this->find(list1.read(p))))
                        this->push_back(list1.read(p));
}

template<class T, class P>
void Linear_list<T, P>::cleanVal(const value_type& value)
{
        if (!this->empty())
        {
                vector<T> temp;
                for (position p = this->begin(); !this->end(p); p = this->next(p))
                        if (this->read(p) != value)
                                temp.push_back(this->read(p));

                this->clear();
                for (typename vector<T>::iterator it = temp.begin(); it != temp.end(); it++)
                        this->push_back(*it);
        }
}
#endif // _LINLIST_H
```

Liste – Realizzazione con puntatori

```
#ifndef _LISTAP_H
#define _LISTAP_H

#include "cellalp.h"
#include "linear_list.h"

template<class T>
class List_pointer: public Linear_list<T, Cell<T>*>
{
public:
	typedef typename Linear_list<T, Cell<T>*>::position position;
	typedef typename Linear_list<T, Cell<T>*>::value_type value_type;

	List_pointer();
	List_pointer(const List_pointer<T>&);
	~List_pointer();

	List_pointer<T>& operator =(const List_pointer<T>&);

	void create(); // create the list
	bool empty() const; // true if the list is empty
	value_type read(position) const; // read the element in the position
	void write(const value_type&, position); // write the element at the position
	position begin() const; // return the position of the first element of the list
	bool end(position) const; // true if the position is the last of the list
	position next(position) const; // return the next position
	position previous(position) const; // return the previous position
	void insert(const value_type&, position); // add an element before the position
	void erase(position); // erase the element in the position

private:
	position list;
};

template<class T>
List_pointer<T>::List_pointer()
{
	this->create();
}

template<class T>
List_pointer<T>::List_pointer(const List_pointer<T>& list)
{
	this->create();
	*this = list;
}

template<class T>
List_pointer<T>::~List_pointer()
{
	this->clear();
}

template<class T>
List_pointer<T>& List_pointer<T>::operator =(const List_pointer<T>& s)
{
	if (&s != this) // avoid auto-assignment
	{
```

```
		this->clear();
		position p = s.begin();

		for (int i = 0; i < s.size(); i++)
		{
			this->push_back(s.read(p));
			p = this->next(p);
		}
	}

	return *this;
}

template<class T>
void List_pointer<T>::create()
{
	value_type valueNull = value_type();
	list = new Cell<value_type>(valueNull);
	list->setValue(valueNull);
	list->setNext(list);
	list->setPrev(list);
}

template<class T>
bool List_pointer<T>::empty() const
{
	return ((list->getNext() == list) && (list->getPrev() == list));
}

template<class T>
typename List_pointer<T>::position List_pointer<T>::begin() const
{
	return list->getNext();
}

template<class T>
typename List_pointer<T>::position List_pointer<T>::next(position p) const
{
	if (p == 0)
		throw std::logic_error("List_pointer (exception) - Unable to get next (invalid
position)");

	return p->getNext();
}

template<class T>
typename List_pointer<T>::position List_pointer<T>::previous(position p) const
{
	if (p == 0)
		throw std::logic_error("List_pointer (exception) - Unable to get previous (invalid
position)");

	return p->getPrev();
}

template<class T>
bool List_pointer<T>::end(position p) const
{
	if (p == 0)
		throw std::logic_error("List_pointer (exception) - Unable to check if is last position
(invalid position)");
```

```
	return (p == list);
}

template<class T>
typename Linear_list<T, Cell<T>*>::value_type List_pointer<T>::read(position p) const
{
	if (p == 0)
		throw std::logic_error("List_pointer (exception) - Unable to read label (invalid
position)");

	return p->getValue();
}

template<class T>
void List_pointer<T>::write(const value_type& a, position p)
{
	if (p == 0)
		throw std::logic_error("List_pointer (exception) - Unable to write label (invalid
position)");

	p->setValue(a);
}

template<class T>
void List_pointer<T>::insert(const value_type& a, position p)
{
	if (p == 0)
		throw std::logic_error("List_pointer (exception) - Unable to insert (invalid
position)");

	position temp = new Cell<value_type>;

	temp->setValue(a);
	temp->setPrev(p->getPrev());
	temp->setNext(p);
	(p->getPrev())->setNext(temp);
	p->setPrev(temp);
	p = temp;
}

template<class T>
void List_pointer<T>::erase(position p)
{
	if (p == 0)
		throw std::logic_error("List_pointer (exception) - Unable to erase (invalid
position)");

	position temp = p;
	(p->getNext())->setPrev(p->getPrev());
	(p->getPrev())->setNext(p->getNext());
	p = p->getNext();
	delete temp;
	temp = 0;
}

#endif // _LISTAP_H
```

Liste – Realizzazione con puntatori
File: “cellalp.h”

```
#ifndef _CELLPL_H
#define _CELLPL_H

template<class T>
class Cell
{
public:

	typedef Cell* position;
	typedef T value_type;

	Cell();
	Cell(const value_type&, position, position);
	Cell(const Cell<T>&);
	~Cell();

	Cell<T>& operator =(const Cell<T>&);

	void setValue(const value_type);
	value_type getValue() const;
	void setNext(position);
	position getNext() const;
	void setPrev(position);
	position getPrev() const;

private:
	value_type value;
	position prev;
	position next;
};

template<class T>
Cell<T>::Cell()
{
	value = value_type();
	prev = 0;
	next = 0;
}

template<class T>
Cell<T>::Cell(const value_type& element, position nextp = 0, position prevp = 0)
{
	value = element;
	prev = prevp;
	next = nextp;
}

template<class T>
Cell<T>::Cell(const Cell<T>& c2)
{
	*this = c2;
}

template<class T>
Cell<T>::~Cell()
{
	value = value_type();
```

```
	prev = 0;
	next = 0;
}

template<class T>
Cell<T>& Cell<T>::operator =(const Cell<T>& c2)
{
	if (this != &c2)
	{
		this->value = c2.value;
		this->prev = c2.prev;
		this->next = c2.next;
	}
}

template<class T>
void Cell<T>::setValue(const value_type element)
{
	value = element;
}

template<class T>
typename Cell<T>::value_type Cell<T>::getValue() const
{
	return value;
}

template<class T>
void Cell<T>::setNext(position nextp)
{
	next = nextp;
}

template<class T>
typename Cell<T>::position Cell<T>::getNext() const
{
	return next;
}

template<class T>
void Cell<T>::setPrev(position prevp)
{
	prev = prevp;
}

template<class T>
typename Cell<T>::position Cell<T>::getPrev() const
{
	return prev;
}

#endif // _CELLPL_H
```

Liste – Realizzazione con vettore

```
#ifndef _LISTVT_H
#define _LISTVT_H

#include "linear_list.h"

template<class T>
class List_vector: public Linear_list<T, int>
{
public:
	typedef typename Linear_list<T, int>::value_type value_type;
	typedef typename Linear_list<T, int>::position position;

	List_vector();
	List_vector(int); // size
	List_vector(const List_vector<T>&);
	~List_vector();

	List_vector<T>& operator =(const List_vector<T>&);

	void create(); //create the list
	bool empty() const; // true if the list is empty
	value_type read(position) const; //read the element in the position
	void write(const value_type&, position); // write the element at position
	position begin() const; // returns a position pointing to the beginning of the list
	bool end(position) const; // true if the position is the last of the list
	position next(position) const; // returns the next position
	position previous(position) const; // return the previous position
	void insert(const value_type&, position); // add an element before the position
	void erase(position pos); // erases the element in that position

private:
	void arrayDoubling(value_type*&, const int, const int);
	value_type* elements_;
	int length_; // the length of the list
	int array_dimension_; // array's dimension
};

template<class T>
List_vector<T>::List_vector()
{
	elements_ = 0;
	length_ = 0;
	array_dimension_ = 10;
	this->create();
}

template<class T>
List_vector<T>::List_vector(int dim)
{
	if (dim <= 0)
		dim = 10; // default size

	elements_ = 0;
	length_ = 0;
	array_dimension_ = dim;
	this->create();
}
```

```
template<class T>
List_vector<T>::List_vector(const List_vector<T>& Lista)
{
	this->array_dimension_ = Lista.array_dimension_;
	this->length_ = Lista.length_;
	this->elements_ = new value_type[array_dimension_];
	for (int i = 0; i < Lista.array_dimension_; i++)
		this->elements_[i] = Lista.elements_[i];
}

template<class T>
List_vector<T>::~List_vector()
{
	delete[] elements_;
}

template<class T>
List_vector<T>& List_vector<T>::operator =(const List_vector<T>& l)
{
	if (this != &l)
	{
		this->array_dimension_ = l.array_dimension_;
		this->length_ = l.length_;
		delete this->elements_;
		this->elements_ = new value_type[array_dimension_];
		for (int i = 0; i < l.array_dimension_; i++)
			this->elements_[i] = l.elements_[i];
	}
	return *this;
}

template<class T>
void List_vector<T>::create()
{
	this->elements_ = new value_type[array_dimension_];
	this->length_ = 0;
}

template<class T>
bool List_vector<T>::empty() const
{
	return (length_ == 0);
}

template<class T>
typename List_vector<T>::position List_vector<T>::begin() const
{
	return 1;
}

template<class T>
typename List_vector<T>::position List_vector<T>::next(position p) const
{
	if ((0 < p) && (p < length_ + 1))
		return (p + 1);
	else
		return p;
}

template<class T>
typename List_vector<T>::position List_vector<T>::previous(position p) const
{
```

```
	if ((1 < p) && (p < length_ + 1))
		return (p - 1);
	else
		return p;
}

template<class T>
bool List_vector<T>::end(position p) const
{
	if ((0 < p) && (p <= length_ + 1))
		return (p == length_ + 1);
	else
		return false;
}

template<class T>
typename List_vector<T>::value_type List_vector<T>::read(position p) const
{
	if ((0 < p) && (p < length_ + 1))
		return (elements_[p - 1]);
	else
		throw std::logic_error("List_vector: (exception) - Unable to read node (invalid
position)");
}

template<class T>
void List_vector<T>::write(const value_type& a, position p)
{
	if ((0 < p) && (p < length_ + 1))
		elements_[p - 1] = a;
	else
		throw std::logic_error("List_vector: (exception) - Unable to write value (invalid
position)");
}

template<class T>
void List_vector<T>::insert(const value_type &a, position p)
{
	if ((0 < p) && (p <= length_ + 1))
	{
		if (length_ == array_dimension_)
		{
			arrayDoubling(elements_, array_dimension_, array_dimension_ * 2);
			array_dimension_ = array_dimension_ * 2;
		}

		for (int i = length_; i >= p; i--)
			elements_[i] = elements_[i - 1];
		elements_[p - 1] = a;
		length_++;
	}
	else
		throw std::logic_error("List_vector: (exception) - Unable to insert node (invalid
position)");
}

template<class T>
void List_vector<T>::erase(position p)
{
	if (this->empty())
		throw std::logic_error("List_vector: (exception) - Unable to erase node (empty
list)");
```

```
	if ((0 < p) && (p < length_ + 1))
	{
		for (int i = p - 1; i < (length_ - 1); i++)
			elements_[i] = elements_[i + 1];
		length_--;
	}
	else
		throw std::logic_error("List_vector: (exception) - Unable to erase node (invalid 
position)");
}

template<class T>
void List_vector<T>::arrayDoubling(value_type*& a, const int vecchiaDim, const int nuovaDim)
{
	value_type* temp = new value_type[nuovaDim];
	int number;
	if (vecchiaDim < nuovaDim)
		number = vecchiaDim;
	else
		number = nuovaDim;

	for (int i = 0; i < number; i++)
		temp[i] = a[i];

	delete[] a;
	a = temp;
}

#endif // _LISTVT_H
```

Liste – Tester

```
#include "list_vector.h"

#include <string>
using std::string;

int main ( )
{
	List_vector<string> list1, list2, list3, list4, list5, list6;

	list1.push_back("Alessandro");
	list1.push_back("Davide");
	list1.push_back("Nicola");
	cout << "List1: " << list1 << endl;

	list2.insOrd("Roberto");
	list2.insOrd("Barbara");
	list2.insOrd("Tiziano");
	list2.insOrd("Gianvito");
	list2.insOrd("Nicola");
	cout << "List2: " << list2 << endl;

	list3.merge(list1, list2);
	cout << "List3: " << list3 << endl;

	list4.join(list1, list2);
	cout << "List4: " << list4 << endl;

	if (list3 == list4)
		cout << "List3 = List4" << endl;
	else
		cout << "List3 != List4" << endl;

	cout << endl << "Sorting List4..." << endl;
	list4.sort();
	cout << "List4: " << list4 << endl;

	if (list3 == list4)
		cout << "List3 = List4" << endl;
	else
		cout << "List3 != List4" << endl;

	cout << endl;

	list5.insOrd("Barbara");
	list5.insOrd("Tiziano");
	list5.insOrd("Gianvito");
	list5.insOrd("Mauro");
	list5.insOrd("Stefano");
	cout << "List5: " << list5 << endl;

	cout << "List6 = elements in common between List2 e List5" << endl;
	list6.common(list2, list5);
	cout << "List6: " << list6 << endl;
}
```

Liste – Output tester

```
List1: [ Alessandro, Davide, Nicola ]

List2: [ Barbara, Gianvito, Nicola, Roberto, Tiziano ]

List3: [ Alessandro, Barbara, Davide, Gianvito, Nicola, Nicola, Roberto, Tiziano ]

List4: [ Alessandro, Davide, Nicola, Barbara, Gianvito, Nicola, Roberto, Tiziano ]

List3 != List4

Sorting List4...
List4: [ Alessandro, Barbara, Davide, Gianvito, Nicola, Nicola, Roberto, Tiziano ]

List3 = List4

List5: [ Barbara, Gianvito, Mauro, Stefano, Tiziano ]

List6 = elements in common between List2 e List5
List6: [ Barbara, Gianvito, Tiziano ]
```

Pile – Classe astratta

```
#ifndef _STACK_H
#define _STACK_H

#include <iostream>
#include <deque>

using std::cout;
using std::endl;
using std::ostream;
using std::deque;

template<class T, class N>
class Stack
{
public:

	typedef T value_type;
	typedef N position;

	virtual ~Stack()
	{
	}

	virtual void create() = 0; // create the stack
	virtual bool empty() const = 0; // true if the stack is empty
	virtual value_type read() const = 0; // return the top of the stack (don't erase it)
	virtual void push(const value_type&) = 0; // add an element to the top of the stack
	virtual void pop() = 0; // erase the first element

	bool operator ==(Stack<value_type, position>&);
	bool operator !=(Stack<value_type, position>&);

	int size(); // return the number of element in the stack
	void clear(); // erase all the elements
	void invert(); // invert the order of the elements in the stack
	bool find(const value_type); // true if the element is present in the stack
	void sort(); // sort the elements: smallest at the top

};

template<class T, class N>
ostream& operator <<(ostream& os, Stack<T, N>& currstack)
{
	deque<T> elements;

	if (currstack.empty())
		os << "Empty Stack" << endl;
	else
	{
		os << "Top -> [ " << currstack.read();
		elements.push_back(currstack.read());
		currstack.pop();

		while (!currstack.empty())
		{
			os << ", " << currstack.read();
			elements.push_back(currstack.read());
			currstack.pop();
```

```
		}

		os << " ] <- Bottom" << endl;

		for (typename deque<T>::const_reverse_iterator it = elements.rbegin(); it !=
elements.rend(); it++)
			currstack.push(*it);
	}

	return os;
}

template<class T, class N>
bool Stack<T, N>::operator ==(Stack<T, N>& s2)
{
	bool flag = true;

	if (this->size() != s2.size())
		return false;

	deque<value_type> s1elem;
	deque<value_type> s2elem;

	while (!this->empty())
	{
		if (this->read() != s2.read())
			flag = false;

		s1elem.push_back(this->read());
		this->pop();
		s2elem.push_back(s2.read());
		s2.pop();
	}

	for (typename deque<value_type>::const_reverse_iterator it = s1elem.rbegin(); it !=
s1elem.rend(); it++)
		this->push(*it);

	for (typename deque<value_type>::const_reverse_iterator it2 = s2elem.rbegin(); it2 !=
s2elem.rend();
			it2++)
		s2.push(*it2);

	return flag;
}

template<class T, class N>
bool Stack<T, N>::operator !=(Stack<T, N>& s2)
{
	return (!(*this == s2));
}

template<class T, class N>
void Stack<T, N>::clear()
{
	while (!this->empty())
		this->pop();
}

template<class T, class N>
int Stack<T, N>::size()
{
```

```
	if (this->empty())
		return 0;

	int len = 0;
	deque<value_type> elements;

	while (!this->empty())
	{
		elements.push_back(this->read());
		this->pop();
		len++;
	}

	for (typename deque<value_type>::const_reverse_iterator it = elements.rbegin(); it !=
elements.rend();
			it++)
		this->push(*it);

	return len;
}

template<class T, class N>
void Stack<T, N>::invert()
{
	if (!this->empty())
	{
		deque<value_type> elements;

		while (!this->empty())
		{
			elements.push_back(this->read());
			this->pop();
		}

		for (typename deque<value_type>::iterator it = elements.begin(); it != elements.end();
it++)
			this->push(*it);
	}

}

template<class T, class N>
bool Stack<T, N>::find(const value_type val)
{
	if (!this->empty())
	{
		deque<value_type> elements;
		bool flag = false;

		while (flag == false && !this->empty())
		{
			if (this->read() == val)
				flag = true;

			elements.push_front(this->read());
			this->pop();
		}

		for (typename deque<value_type>::iterator it = elements.begin(); it != elements.end();
it++)
			this->push(*it);
```

```
			return flag;
		}

		return false;
}

template<class T, class N>
void Stack<T, N>::sort()
{
	if (!this->empty())
	{
		int stackSize = this->size();
		value_type *elements = new value_type[stackSize];
		int i = 0, j = 0, imin = 0;
		value_type tempValue;

		while (!this->empty())
		{
			elements[i] = this->read();
			this->pop();
			i++;
		}

		//Selection sort of the array
		for (i = 0; i < stackSize - 1; i++)
		{
			imin = i;
			for (j = i + 1; j < stackSize; j++)
			{
				if (elements[j] < elements[imin])
					imin = j;
			}

			tempValue = elements[i];
			elements[i] = elements[imin];
			elements[imin] = tempValue;
		}

		// Insert element in the stack (first the greatest )
		for (i = 0; i < stackSize; i++)
			this->push(elements[stackSize - 1 - i]);
	}
}

#endif // _STACK_H
```

Pile – Realizzazione con puntatori

```
#ifndef _STACKP_H
#define _STACKP_H

#include <stdexcept>
#include "stack.h"
#include "elemsp.h"

template<class T>
class Stack_pointer: public Stack<T, Element<T>*>
{
public:
	typedef typename Stack<T, Element<T>*>::value_type value_type;
	typedef typename Stack<T, Element<T>*>::position elemp;

	Stack_pointer();
	Stack_pointer(const Stack_pointer<T>&);
	~Stack_pointer();

	Stack_pointer<T>& operator =(const Stack_pointer<T>&);

	void create(); // create the stack
	bool empty() const; // true if the stack is empty
	value_type read() const; // return the top of the stack (don't erase it)
	void push(const value_type&); // add an element to the top of the stack
	void pop(); // erase the first element

private:
	elemp top;
};

template<class T>
Stack_pointer<T>::Stack_pointer()
{
	this->create();
}

template<class T>
Stack_pointer<T>::Stack_pointer(const Stack_pointer<T>& s2)
{
	this->create();
	*this = s2;
}

template<class T>
Stack_pointer<T>::~Stack_pointer()
{
	this->clear();
	top = 0;
}

template<class T>
Stack_pointer<T>& Stack_pointer<T>::operator =(const Stack_pointer<T>& s)
{
	if (&s != this) // avoid auto-assignment
	{
		this->clear();
		top = 0;
```

```
		elemp curr = s.top;
		while (curr->getPrev() != 0)
			curr = curr->getPrev();
		while (curr != 0)
		{
			this->push(curr->getValue());
			curr = curr->getNext();
		}
	}
	return *this;
}

template<class T>
void Stack_pointer<T>::create()
{
	top = 0;
}

template<class T>
bool Stack_pointer<T>::empty() const
{
	return top == 0;
}

template<class T>
typename Stack_pointer<T>::value_type Stack_pointer<T>::read() const
{
	if (this->empty())
		throw std::logic_error("Stack_pointer: exception - Unable to read (empty stack)");
	return top->getValue();
}

template<class T>
void Stack_pointer<T>::push(const value_type& el)
{
	elemp newtop = new Element<value_type>;
	newtop->setValue(el);
	if (top != 0)
	{
		newtop->setPrev(top);
		top->setNext(newtop);
	}
	top = newtop;

}

template<class T>
void Stack_pointer<T>::pop()
{
	if (this->empty())
		throw std::logic_error("Stack_pointer: exception - Unable to pop (empty stack)");

	elemp newtop = top->getPrev();
	delete top;
	top = 0;
	if (newtop != 0)
	{
		top = newtop;
		top->setNext(0);
	}
}
#endif // _STACKP_H
```

Pile – Realizzazione con puntatori
File: “elemsp.h”

```
#ifndef _ELEMSP_H
#define _ELEMSP_H

template<class T>
class Element
{
public:
	typedef T value_type;
	typedef Element* elemp;
	Element();
	Element(const value_type&);
	~Element();
	Element<T>& operator =(const Element<T>&);
	bool operator ==(const Element<T>&) const;
	bool operator !=(const Element<T>&) const;
	void setValue(const value_type);
	value_type getValue() const;
	void setPrev(elemp);
	void setNext(elemp);
	elemp getPrev() const;
	elemp getNext() const;

private:
	value_type value;
	elemp prev;
	elemp next;
};

template<class T>
Element<T>::Element()
{
	value = value_type();
	prev = 0;
	next = 0;
}

template<class T>
Element<T>::Element(const value_type& val)
{
	value = val;
	prev = 0;
	next = 0;
}

template<class T>
Element<T>::~Element()
{
	value = value_type();
	prev = 0;
	next = 0;
}

template<class T>
Element<T>& Element<T>::operator =(const Element<T>& c2)
{
	if (&c2 != this) // avoid auto-assignment
	{
```

```
			this->setValue(c2.getValue());
			this->setPrev(c2.getPrev());
			this->setNext(c2.getNext());
		}
		return *this;
}

template<class T>
bool Element<T>::operator ==(const Element<T>& c2) const
{
		if (this->value != c2.value)
			return false;
		if (this->prev != c2.prev)
			return false;
		if (this->next != c2.next)
			return false;

		return true;
}

template<class T>
bool Element<T>::operator !=(const Element<T>& c2) const
{
		return (!(c2 == *this));
}

template<class T>
void Element<T>::setValue(const value_type element)
{
		value = element;
}

template<class T>
typename Element<T>::value_type Element<T>::getValue() const
{
		return value;
}

template<class T>
void Element<T>::setPrev(elemp prevp)
{
		prev = prevp;
}

template<class T>
void Element<T>::setNext(elemp nextp)
{
		next = nextp;
}

template<class T>
typename Element<T>::elemp Element<T>::getPrev() const
{
		return prev;
}

template<class T>
typename Element<T>::elemp Element<T>::getNext() const
{
		return next;
}
#endif // _ELEMSP_H
```

Pile – Realizzazione con vettore

```
#ifndef _STACKVT_H
#define _STACKVT_H

#include <stdexcept>
#include "stack.h"

template<class T>
class Stack_vector: public Stack<T, int>
{
public:

	typedef typename Stack<T, int>::value_type value_type;

	Stack_vector();
	Stack_vector(int); // size
	Stack_vector(const Stack_vector<T>&);
	~Stack_vector();

	Stack_vector<T>& operator =(const Stack_vector<T>&);

	void create(); // create the stack
	bool empty() const; // true if the stack is empty
	value_type read() const; // return the top of the stack (don't erase it)
	void push(const value_type&); // add an element to the top of the stack
	void pop(); // erase the first element

private:
	value_type* element;
	int maxLen;
	int topIndex;

	void arrayDoubling(value_type*&, const int, const int);
};

template<class T>
Stack_vector<T>::Stack_vector()
{
	maxLen = 25; // default size
	topIndex = 0;
	element = 0;
	this->create();
}

template<class T>
Stack_vector<T>::Stack_vector(int size)
{
	if (size <= 0)
		size = 25; // default size

	maxLen = size;
	topIndex = 0;
	element = 0;
	this->create();
}

template<class T>
Stack_vector<T>::Stack_vector(const Stack_vector<T>& s)
{
```

```
	*this = s;
}

template<class T>
Stack_vector<T>::~Stack_vector()
{
	delete[] element;
}

template<class T>
Stack_vector<T>& Stack_vector<T>::operator =(const Stack_vector<T>& s)
{
	if (&s != this) // avoid auto-assignment
	{
		// copy the vector's elements and its size
		this->maxLen = s.maxLen;

		// allocates memory for the vector
		this->create();
		this->topIndex = s.topIndex;

		for (int i = 0; i < this->topIndex; i++)
			this->element[i] = s.element[i];
	}

	return *this;
}

template<class T>
void Stack_vector<T>::create()
{
	element = new value_type[maxLen];
	topIndex = 0;
}

template<class T>
bool Stack_vector<T>::empty() const
{
	return (topIndex == 0);
}

template<class T>
typename Stack_vector<T>::value_type Stack_vector<T>::read() const
{
	if (this->empty())
		throw std::logic_error("Stack_vector: (exception) - Unable to read (empty stack)");

	return element[topIndex - 1];
}

template<class T>
void Stack_vector<T>::push(const value_type& el)
{
	if (topIndex == maxLen)
	{
		this->arrayDoubling(element, maxLen, maxLen * 2);

		topIndex = maxLen;
		maxLen = maxLen * 2;
	}

	element[topIndex] = el;
```

```
	topIndex++;
}

template<class T>
void Stack_vector<T>::pop()
{
	if (this->empty())
		throw std::logic_error("Stack_vector: (exception) - Unable to pop (empty stack)");

	topIndex--;
}

template<class T>
void Stack_vector<T>::arrayDoubling(value_type*& a, const int oldSize, const int newSize)
{
	value_type* temp = new value_type[newSize];

	for (int i = 0; i < oldSize; i++)
		temp[i] = a[i];

	delete[] a;
	a = temp;
}

#endif // _STACKVTVT_H
```

Pile – Tester

```
#include "stackvt.h"

int main()
{
	Stack_vector<int> stack1, stack2;

	stack1.push(11);
	stack1.push(6);
	stack1.push(5);
	stack1.push(4);
	stack1.push(9);
	stack1.push(6);
	stack1.push(10);
	stack1.push(3);

	cout << "stack1: " << endl << stack1 << endl;

	stack1.sort();
	cout << "stack1 after 'sort': " << endl << stack1 << endl;

	stack2 = stack1;

	cout << "stack2: " << endl << stack2 << endl;

	if(stack1 == stack2)
		cout << "stack1 = stack2" << endl;
	else
		cout << "stack1 != stack2" << endl;

	stack2.invert();
	cout << endl << "stack2 after 'invert': " << endl << stack2 << endl;

	cout << "stack2 size: " << stack2.size() << endl;

	stack2.pop();
	stack2.pop();
	cout << endl << "stack2 after 2 pop: " << endl << stack2 << endl;

	cout << "new stack2 size: " << stack2.size() << endl;

}
```

Pile – Output tester

```
stack1:
Top -> [ 3, 10, 6, 9, 4, 5, 6, 11 ] <- Bottom

stack1 after 'sort':
Top -> [ 3, 4, 5, 6, 6, 9, 10, 11 ] <- Bottom

stack2:
Top -> [ 3, 4, 5, 6, 6, 9, 10, 11 ] <- Bottom

stack1 = stack2

stack2 after 'invert':
Top -> [ 11, 10, 9, 6, 6, 5, 4, 3 ] <- Bottom

stack2 size: 8

stack2 after 2 pop:
Top -> [ 9, 6, 6, 5, 4, 3 ] <- Bottom

new stack2 size: 6
```

Code – Classe astratta

```
#ifndef _QUEUE_H
#define _QUEUE_H

#include <iostream>
#include <vector>

using std::endl;
using std::ostream;
using std::vector;

template<class T, class P>
class Queue
{
public:

	typedef T value_type;
	typedef P position;

	virtual ~Queue()
	{
	}

	virtual void create() = 0; // create the queue
	virtual bool empty() const = 0; // true if the queue is empty
	virtual value_type read() const = 0; // return the first element of the queue (don't erase
it)
	virtual void enqueue(const value_type&) = 0; // add an element at the end of the queue
	virtual void dequeue() = 0; // erase the first element of the queue

	bool operator ==(Queue<T, P>&);
	bool operator !=(Queue<T, P>&);

	int size(); // return the number of element in the queue
	void clear(); // erase all the elements
	void invert(); // invert the order of the elements in the queue
	bool find(const value_type); // true if the element is present in the queue
};

template<class T, class P>
ostream& operator <<(ostream& os, Queue<T, P>& currQueue)
{
	vector<T> elements;

	if (currQueue.empty())
		os << "Empty Queue" << endl;
	else
	{
		os << "[ " << currQueue.read();
		elements.push_back(currQueue.read());
		currQueue.dequeue();

		while (!currQueue.empty())
		{
			os << ", " << currQueue.read();
			elements.push_back(currQueue.read());
			currQueue.dequeue();
		}
```

```
			os << " ]" << endl;

			for (typename vector<T>::iterator it = elements.begin(); it != elements.end(); it++)
				currQueue.enqueue(*it);
		}

		return os;
}

template<class T, class P>
bool Queue<T, P>::operator ==(Queue<T, P>& q2)
{
		bool flag = true;

		if (this->size() != q2.size())
			return false;

		vector<T> q1elem;
		vector<T> q2elem;

		while (!this->empty())
		{
			if (this->read() != q2.read())
				flag = false;

			q1elem.push_back(this->read());
			this->dequeue();
			q2elem.push_back(q2.read());
			q2.dequeue();
		}

		for (typename vector<T>::iterator it = q1elem.begin(); it != q1elem.end(); it++)
			this->enqueue(*it);

		for (typename vector<T>::iterator it2 = q2elem.begin(); it2 != q2elem.end(); it2++)
			q2.enqueue(*it2);

		return flag;
}

template<class T, class P>
bool Queue<T, P>::operator !=(Queue<T, P>& q2)
{
		return (!(*this == q2));
}

template<class T, class P>
void Queue<T, P>::clear()
{
		while (!this->empty())
			this->dequeue();
}

template<class T, class P>
int Queue<T, P>::size()
{
		if (this->empty())
			return 0;

		int len = 0;
		vector<T> elements;
```

```
	while (!this->empty())
	{
		elements.push_back(this->read());
		this->dequeue();
		len++;
	}

	for (int i = 0; i < len; i++)
		this->enqueue(elements[i]);

	return len;
}

template<class T, class P>
void Queue<T, P>::invert()
{
	if (!this->empty())
	{
		vector<value_type> elements;

		while (!this->empty())
		{
			elements.push_back(this->read());
			this->dequeue();
		}

		for (typename vector<value_type>::reverse_iterator it = elements.rbegin(); it !=
elements.rend(); it++)
			this->enqueue(*it);
	}

}

template<class T, class P>
bool Queue<T, P>::find(const value_type val)
{
	if (!this->empty())
	{
		vector<value_type> elements;
		bool flag = false;

		while (!this->empty())
		{
			if (this->read() == val)
				flag = true;

			elements.push_back(this->read());
			this->dequeue();
		}

		for (typename vector<value_type>::iterator it = elements.begin(); it !=
elements.end(); it++)
			this->enqueue(*it);

		return flag;
	}

	return false;
}

#endif // _QUEUE_H
```

Code – Realizzazione con puntatori

```
#ifndef _QUEUEPT_
#define _QUEUEPT_

#include <stdexcept>
#include "queue.h"
#include "elemqp.h"

template<class T>
class Queue_pointer: public Queue<T, Element<T>*>
{
public:

	typedef typename Queue<T, Element<T>*>::value_type value_type;
	typedef typename Queue<T, Element<T>*>::position elemp;

	Queue_pointer();
	Queue_pointer(const Queue_pointer<T>&);
	~Queue_pointer();

	Queue_pointer<T>& operator =(const Queue_pointer<T>&);

	void create(); // create the queue
	bool empty() const; // true if the queue is empty
	value_type read() const; // return the first element of the queue (don't erase it)
	void enqueue(const value_type&); // add an element at the end of the queue
	void dequeue(); // erase the first element of the queue

private:

	elemp head;
	elemp tail;
};

template<class T>
Queue_pointer<T>::Queue_pointer()
{
	head = 0;
	tail = 0;
}

template<class T>
Queue_pointer<T>::Queue_pointer(const Queue_pointer<T>& q2)
{
	*this = q2;
}

template<class T>
Queue_pointer<T>::~Queue_pointer()
{
	this->clear();
	head = 0;
	tail = 0;
}

template<class T>
Queue_pointer<T>& Queue_pointer<T>::operator =(const Queue_pointer<T>& q2)
{
	if (&q2 != this) // avoid auto-assignment
```

```
	{
		elemp currElem = q2.head;

		while (currElem != 0)
		{
			this->enqueue(currElem->getValue());
			currElem = currElem->getNext();
		}
	}

	return *this;
}

template<class T>
void Queue_pointer<T>::create()
{
	this->clear();
	head = 0;
	tail = 0;
}

template<class T>
bool Queue_pointer<T>::empty() const
{
	return (head == 0);
}

template<class T>
typename Queue<T, Element<T>*>::value_type Queue_pointer<T>::read() const
{
	if (this->empty())
		throw std::logic_error("Queue_pointer (exception) - Unable to read (empty queue)");

	return head->getValue();
}

template<class T>
void Queue_pointer<T>::enqueue(const value_type& el)
{
	elemp newtail = new Element<value_type>;
	newtail->setValue(el);

	if (tail == 0)
		head = newtail;
	else
	{
		newtail->setPrev(tail);
		tail->setNext(newtail);
	}
	tail = newtail;
}

template<class T>
void Queue_pointer<T>::dequeue()
{
	if (this->empty())
		throw std::logic_error("Queue_pointer (exception) - Unable to dequeue (empty queue)");

	elemp newhead = head->getNext();
	delete head;
	head = 0;
	if (newhead != 0)
```

```
	{
		head = newhead;
		head->setPrev(0);
	}
	else
		tail = 0;
}

#endif //_QUEUEPT_
```

Code – Realizzazione con puntatori
File: “elemqp.h”

```
#ifndef _ELEMQP_H
#define _ELEMQP_H

template<class T>
class Element
{
public:
	typedef T value_type;
	typedef Element* elemp;
	Element();
	Element(const value_type&);
	~Element();
	Element<T>& operator =(const Element<T>&);
	bool operator ==(const Element<T>&) const;
	bool operator !=(const Element<T>&) const;
	void setValue(const value_type);
	value_type getValue() const;
	void setPrev(elemp);
	void setNext(elemp);
	elemp getPrev() const;
	elemp getNext() const;

private:
	value_type value;
	elemp prev;
	elemp next;
};

template<class T>
Element<T>::Element()
{
	value = value_type();
	prev = 0;
	next = 0;
}

template<class T>
Element<T>::Element(const value_type& val)
{
	value = val;
	prev = 0;
	next = 0;
}

template<class T>
Element<T>::~Element()
{
	value = value_type();
	prev = 0;
	next = 0;
}

template<class T>
Element<T>& Element<T>::operator =(const Element<T>& c2)
{
	if (&c2 != this) // avoid auto-assignment
	{
```

```
		this->setValue(c2.getValue());
		this->setPrev(c2.getPrev());
		this->setNext(c2.getNext());
	}
	return *this;
}

template<class T>
bool Element<T>::operator ==(const Element<T>& c2) const
{
	if (this->value != c2.value)
		return false;
	if (this->prev != c2.prev)
		return false;
	if (this->next != c2.next)
		return false;

	return true;
}

template<class T>
bool Element<T>::operator !=(const Element<T>& c2) const
{
	return (!(c2 == *this));
}

template<class T>
void Element<T>::setValue(const value_type element)
{
	value = element;
}

template<class T>
typename Element<T>::value_type Element<T>::getValue() const
{
	return value;
}

template<class T>
void Element<T>::setPrev(elemp prevp)
{
	prev = prevp;
}

template<class T>
void Element<T>::setNext(elemp nextp)
{
	next = nextp;
}

template<class T>
typename Element<T>::elemp Element<T>::getPrev() const
{
	return prev;
}

template<class T>
typename Element<T>::elemp Element<T>::getNext() const
{
	return next;
}
#endif // _ELEMQP_H
```

Code – Realizzazione con vettore

```
#ifndef _QUEUEVT_
#define _QUEUEVT_

#include <stdexcept>
#include "queue.h"

template<class T>
class Queue_vector: public Queue<T, int>
{
public:

	typedef typename Queue<T, int>::value_type value_type;

	Queue_vector();
	Queue_vector(int);
	Queue_vector(const Queue_vector<T>&);
	~Queue_vector();

	Queue_vector<T>& operator =(const Queue_vector<T>&);

	void create(); //create the queue
	bool empty() const; // true if the queue is empty
	value_type read() const; //return the first element of the queue (don't erase it)
	void enqueue(const value_type&); //add an element at the end of the queue
	void dequeue(); //erase the first element of the queue

private:
	void arrayDoubling(value_type*&, const int, const int, const int);

	value_type* element;
	int head, len, maxLen;
};

template<class T>
Queue_vector<T>::Queue_vector()
{
	maxLen = 25;
	head = 0;
	len = 0;
	element = 0;
	this->create();
}

template<class T>
Queue_vector<T>::Queue_vector(int size)
{
	if (size <= 0)
		maxLen = 25;

	maxLen = size;
	head = 0;
	len = 0;
	element = 0;
	this->create();
}
```

```
template<class T>
Queue_vector<T>::Queue_vector(const Queue_vector<T>& q2)
{
	*this = q2;
}

template<class T>
Queue_vector<T>::~Queue_vector()
{
	delete[] element;
}

template<class T>
Queue_vector<T>& Queue_vector<T>::operator =(const Queue_vector<T>& q2)
{
	if (&q2 != this) // avoid auto-assignment
	{
		// copy the vector's elements and its size
		this->maxLen = q2.maxLen;

		// allocates memory for the element vector
		this->create();
		this->head = q2.head;
		this->len = q2.len;

		for (int i = 0; i < len; i++)
			this->element[((q2.head + i) % this->maxLen)] = q2.element[((q2.head + i) %
q2.maxLen)];
	}

	return *this;
}

template<class T>
void Queue_vector<T>::create()
{
	element = new value_type[maxLen];
	head = 0;
	len = 0;
}

template<class T>
bool Queue_vector<T>::empty() const
{
	return (len == 0);
}

template<class T>
typename Queue<T, int>::value_type Queue_vector<T>::read() const
{
	if (this->empty())
		throw std::logic_error("Queue_vector (exception) - Unable to read (empty queue)");

	return (element[head]);
}

template<class T>
void Queue_vector<T>::enqueue(const value_type& el)
{
	if (len == maxLen)
	{
		this->arrayDoubling(element, head, maxLen, maxLen * 2);
```

```
		head = 0;
		maxLen = maxLen * 2;
	}

	element[(head + len) % maxLen] = el;
	len++;
}

template<class T>
void Queue_vector<T>::dequeue()
{
	if (this->empty())
		throw std::logic_error("Queue_vector (exception) - Unable to dequeue (empty queue)");

	head = (head + 1) % maxLen;
	len--;
}

template<class T>
void Queue_vector<T>::arrayDoubling(value_type*& a, const int head, const int oldDim, const int
newDim)
{

	value_type* temp = new value_type[newDim];

	for (int i = 0; i < oldDim; i++)
		temp[i] = a[(head + i) % oldDim];

	delete[] a;
	a = temp;
}

#endif //QUEUEVT_
```

Code – Tester

```
#include "queue_pointer.h"

using std::cout;

int main()
{
	Queue_pointer<int> queue1, queue2;

	queue1.enqueue(1);
	queue1.enqueue(2);
	queue1.enqueue(3);
	queue1.enqueue(4);
	queue1.enqueue(5);
	queue1.enqueue(6);
	queue1.enqueue(7);

	cout << "queue1: " << queue1 << endl;

	if (queue1.find(9))
		cout << "9 present in queue1" << endl;
	else
		cout << "9 not present in queue1" << endl;

	queue2 = queue1;

	cout << endl << "queue2: " << queue2 << endl;

	if (queue2 == queue1)
		cout << "queue2 = queue1" << endl;
	else
		cout << "queue2 != queue1" << endl;

	queue2.enqueue(9);

	cout << endl << "queue2: " << queue2 << endl;

	if (queue2.find(9))
		cout << "9 present in queue2" << endl;
	else
		cout << "9 not present in queue2" << endl;

	cout << endl;

	if (queue2 == queue1)
		cout << "queue2 = queue1" << endl;
	else
		cout << "queue2 != queue1" << endl;

	queue2.invert();
	cout << endl << "queue2 after 'invert': " << queue2 << endl;

}
```

Code – Output Tester

```
queue1: [ 1, 2, 3, 4, 5, 6, 7 ]

9 not present in queue1

queue2: [ 1, 2, 3, 4, 5, 6, 7 ]

queue2 = queue1

queue2: [ 1, 2, 3, 4, 5, 6, 7, 9 ]

9 present in queue2

queue2 != queue1

queue2 after 'invert': [ 9, 7, 6, 5, 4, 3, 2, 1 ]
```

Insiemi – Classe astratta

```
#ifndef _SET_H
#define _SET_H

#include <iostream>
#include <stdexcept>
#include <vector>
#include <algorithm>

using std::cout;
using std::endl;
using std::ostream;
using std::vector;

template<class T>
class Set
{
public:

	typedef T value_type;

	virtual ~Set()
	{
	}

	virtual void create() = 0; // create the set
	virtual bool empty() const = 0; // true if the set is empty
	virtual bool find(const value_type) const = 0; // true if the element is in the set
	virtual void insert(const value_type) = 0; // add an element to the set
	virtual void erase(const value_type) = 0; // erase the element from the set
	virtual void unionOp(Set<T>&, Set<T>&) = 0; // return the union of the set
	virtual void intersection(Set<T>&, Set<T>&) = 0; // return the elements in common
	virtual void difference(Set<T>&, Set<T>&) = 0; // return the elements of the 1st set not
present in the 2nd set

	virtual value_type pickAny() const = 0; // take one element from the set

	bool operator ==(Set<T>&);
	bool operator !=(Set<T>&);

	int size();
	void clear();

};

template<class T>
ostream& operator <<(ostream& os, Set<T>& s)
{
	if (s.empty())
		os << "Empty Set" << endl;
	else
	{
		vector<T> elements;
		os << "[  ";
		while (!s.empty())
		{
			T temp = s.pickAny();
			os << temp << "  ";
			elements.push_back(temp);
```

```
                        s.erase(temp);
                }
                os << "]" << endl;

                for (typename vector<T>::iterator it = elements.begin(); it != elements.end(); it++)
                        s.insert(*it);
        }

        return os;
}

template<class T>
bool Set<T>::operator ==(Set<T>& set2)
{
        if(this->size() != set2.size())
                return false;

        vector<T> elem1;
        vector<T> elem2;
        bool flag = true;

        while (!this->empty())
        {
                value_type temp = this->pickAny();
                elem1.push_back(temp);
                this->erase(temp);
        }

        while (!set2.empty())
        {
                value_type temp = set2.pickAny();
                elem2.push_back(temp);
                set2.erase(temp);
        }

        for(typename vector<T>::iterator it = elem1.begin(); it != elem1.end(); it++)
                if (std::find(elem2.begin(), elem2.end(), *it) == elem2.end()) // if the element isn't
present
                        flag = false;

        for(typename vector<T>::iterator it = elem1.begin(); it != elem1.end(); it++)
                this->insert(*it);

        for(typename vector<T>::iterator it = elem2.begin(); it != elem2.end(); it++)
                set2.insert(*it);

        return flag;
}

template<class T>
bool Set<T>::operator !=(Set<T>& set2)
{
        return (!(*this == set2));
}

template<class T>
int Set<T>::size()
{
        int len = 0;
        vector<value_type> elements;

        while (!this->empty())
```

```
	{
		value_type temp = this->pickAny();
		elements.push_back(temp);
		this->erase(temp);
		len++;
	}

	for (typename vector<value_type>::iterator it = elements.begin(); it != elements.end(); it++)
		this->insert(*it);

	return len;
}

template<class T>
void Set<T>::clear()
{
	while (!this->empty())
	{
		value_type temp = this->pickAny();
		this->erase(temp);
	}
}

#endif // _SET_H
```

Insiemi – Realizzazione con lista non ordinata

```
#ifndef _SETP_H
#define _SETP_H

#include "listap.h"
#include "set.h"

template<class T>
class Set_pointer: public Set<T>
{
public:
	typedef typename Set<T>::value_type value_type;
	Set_pointer();
	Set_pointer(Set<T>&);
	Set_pointer(const Set_pointer<T>&);
	~Set_pointer();

	Set_pointer<T>& operator =(Set<T>&);
	bool operator ==(const Set_pointer<T>&) const; // hide this method to let Set_pointer class
use Set<T> operator ==
	bool operator !=(const Set_pointer<T>&) const; // hide this method to let Set_pointer class
use Set<T> operator !=

	template <class E>
	friend ostream& operator <<(ostream& os, const Set_pointer<E>&); // operator << with CONST
set
	void create(); // create the set
	bool empty() const; // true if the set is empty
	bool find(const value_type) const; // true if the element is in the set
	void insert(const value_type); // add an element to the set
	void erase(const value_type); // erase the element from the set
	void unionOp(Set<T>&, Set<T>&); // return the union of the set
	void intersection(Set<T>&, Set<T>&); // return the elements in common
	void difference(Set<T>&, Set<T>&); // return the elements of the 1st set not present in the
2nd set
	value_type pickAny() const; // take one element from the set

private:
	List_pointer<T> set;
};

template<class T>
ostream& operator <<(ostream& os, const Set_pointer<T>& s)
{
	if (s.empty())
		os << "Empty Set" << endl;
	else
	{
		os << "[  ";

		for(typename List_pointer<T>::position p = s.set.begin(); !s.set.end(p); p =
s.set.next(p))
			os << s.set.read(p) << "  ";

		os << "]" << endl;
	}

	return os;
}
```

```
template<class T>
Set_pointer<T>::Set_pointer()
{
	this->create();
}

template<class T>
Set_pointer<T>::Set_pointer(const Set_pointer<T>& set)
{
	this->create();
	*this = set;
}

template<class T>
Set_pointer<T>::~Set_pointer()
{
	set.clear();
}

template<class T>
Set_pointer<T>& Set_pointer<T>::operator =(Set<T>& s)
{
	if (&s != this) // avoid auto-assignment
	{
		vector<T> elements;
		this->clear();
		while(!s.empty())
		{
			value_type temp = s.pickAny();
			elements.push_back(temp);
			this->insert(temp);
			s.erase(temp);
		}

		while(!elements.empty()) // re-insert the elements in the set
		{
			s.insert(elements.back());
			elements.pop_back();
		}
	}
	return *this;
}

template<class T>
bool Set_pointer<T>::operator ==(const Set_pointer<T>& s) const
{
	if(set.size() != s.set.size())
		return false;

	for(typename List_pointer<T>::position p = set.begin(); !set.end(p); p = set.next(p))
		if(!s.find(set.read(p)))
			return false;

	return true;
}

template<class T>
bool Set_pointer<T>::operator !=(const Set_pointer<T>& set2) const
{
	return (!(*this == set2));
}
```

```
template<class T>
void Set_pointer<T>::create()
{
	set.create();
}

template<class T>
bool Set_pointer<T>::empty() const
{
	return set.empty();
}

template<class T>
bool Set_pointer<T>::find(const value_type a) const
{
	return set.find(a);
}

template<class T>
void Set_pointer<T>::insert(const value_type a)
{
	set.add_new(a);
}

template<class T>
void Set_pointer<T>::erase(const value_type a)
{
	if (!set.find(a))
		throw std::logic_error("Set_pointer (exception) - Unable to erase (element not present
in the set)");

	set.eraseVal(a);
}

template<class T>
void Set_pointer<T>::unionOp(Set<T>& s1, Set<T>& s2)
{
	this->clear();
	vector<T> elements;

	while (!s1.empty())
	{
		value_type temp = s1.pickAny();
		this->set.add_new(temp);
		elements.push_back(temp);
		s1.erase(temp);
	}

	while(!elements.empty()) // re-insert the elements in the set
	{
		s1.insert(elements.back());
		elements.pop_back();
	}

	while (!s2.empty())
	{
		value_type temp = s2.pickAny();
		this->set.add_new(temp);
		elements.push_back(temp);
		s2.erase(temp);
	}
```

```
	while(!elements.empty()) // re-insert the elements in the set
	{
		s2.insert(elements.back());
		elements.pop_back();
	}
}

template<class T>
void Set_pointer<T>::intersection(Set<T>& s1, Set<T>& s2)
{
	this->clear();
	vector<T> elements;

	while (!s1.empty())
	{
		value_type temp = s1.pickAny();
		if(s2.find(temp))
			this->set.add_new(temp);
		elements.push_back(temp);
		s1.erase(temp);
	}

	while(!elements.empty()) // re-insert the elements in the set
	{
		s1.insert(elements.back());
		elements.pop_back();
	}
}

template<class T>
void Set_pointer<T>::difference(Set<T>& s1, Set<T>& s2)
{
	this->clear();
	vector<T> elements;

	while (!s1.empty())
	{
		value_type temp = s1.pickAny();
		if(!s2.find(temp))
			this->set.add_new(temp);
		elements.push_back(temp);
		s1.erase(temp);
	}

	while(!elements.empty()) // re-insert the elements in the set
	{
		s1.insert(elements.back());
		elements.pop_back();
	}
}

template<class T>
typename Set_pointer<T>::value_type Set_pointer<T>::pickAny() const
{
	if(set.empty())
		throw std::logic_error("Set_pointer (exception) - Unable to pick an element (empty
set)");

	return set.read(set.begin());
}
#endif // _SETP_H
```

Insiemi – Realizzazione con lista ordinata

```
#ifndef _SETPO_H
#define _SETPO_H

#include "listap.h"
#include "set.h"

template<class T>
class Set_pointerOrd: public Set<T>
{
public:
	typedef typename Set<T>::value_type value_type;

	Set_pointerOrd();
	Set_pointerOrd(Set<T>&);
	Set_pointerOrd(const Set_pointerOrd<T>&);
	~Set_pointerOrd();

	Set_pointerOrd<T>& operator =(Set<T>&);
	bool operator ==(const Set_pointerOrd<T>&) const; // hide this method to let Set_pointerOrd
class use Set<T> operator ==
	bool operator !=(const Set_pointerOrd<T>&) const; // hide this method to let Set_pointerOrd
class use Set<T> operator !=

	template <class E>
	friend ostream& operator <<(ostream& os, const Set_pointerOrd<E>&); // operator << with CONST
set

	void create(); // create the set
	bool empty() const; // true if the set is empty
	bool find(const value_type) const; // true if the element is in the set
	void insert(const value_type); // add an element to the set
	void erase(const value_type); // erase the element from the set
	void unionOp(Set<T>&, Set<T>&); // return the union of the set
	void intersection(Set<T>&, Set<T>&); // return the elements in common
	void difference(Set<T>&, Set<T>&); // return the elements of the 1st set not present in the
2nd set

	value_type pickAny() const; // take one element from the set

private:
	List_pointer<T> set;
};

template<class T>
ostream& operator <<(ostream& os, const Set_pointerOrd<T>& s)
{
	if (s.empty())
		os << "Empty Set" << endl;
	else
	{
		os << "[  ";

		for(typename List_pointer<T>::position p = s.set.begin(); !s.set.end(p); p =
s.set.next(p))
			os << s.set.read(p) << "  ";

		os << "]" << endl;
	}
```

```
	return os;
}

template<class T>
Set_pointerOrd<T>::Set_pointerOrd()
{
	this->create();
}

template<class T>
Set_pointerOrd<T>::Set_pointerOrd(Set<T>& set)
{
	this->create();
	*this = set;
}

template<class T>
Set_pointerOrd<T>::Set_pointerOrd(const Set_pointerOrd<T>& set)
{
	this->create();
	*this = set;
}

template<class T>
Set_pointerOrd<T>::~Set_pointerOrd()
{
	set.clear();
}

template<class T>
Set_pointerOrd<T>& Set_pointerOrd<T>::operator =(Set<T>& s)
{
	if (&s != this) // avoid auto-assignment.
	{
		vector<T> elements;
		this->clear();
		while(!s.empty())
		{
			value_type temp = s.pickAny();
			elements.push_back(temp);
			this->insert(temp);
			s.erase(temp);
		}

		while(!elements.empty()) // re-insert the elements in the set
		{
			s.insert(elements.back());
			elements.pop_back();
		}
	}

	return *this;
}

template<class T>
bool Set_pointerOrd<T>::operator ==(const Set_pointerOrd<T>& s) const
{
	if(set.size() != s.set.size())
		return false;

	for(typename List_pointer<T>::position p = set.begin(); !set.end(p); p = set.next(p))
```

```
			if(!s.find(set.read(p)))
				return false;

	return true;
}

template<class T>
bool Set_pointerOrd<T>::operator !=(const Set_pointerOrd<T>& set2) const
{
	return (!(*this == set2));
}

template<class T>
void Set_pointerOrd<T>::create()
{
	set.create();
}

template<class T>
bool Set_pointerOrd<T>::empty() const
{
	return set.empty();
}

template<class T>
bool Set_pointerOrd<T>::find(const value_type a) const
{
	return set.findOrd(a);
}

template<class T>
void Set_pointerOrd<T>::insert(const value_type a)
{
	if(!set.findOrd(a))
		set.insOrd(a);
}

template<class T>
void Set_pointerOrd<T>::erase(const value_type a)
{
	if (!set.findOrd(a))
		throw std::logic_error("Set_pointerOrd: exception - Unable to erase (element not in
the set)");

	set.eraseVal(a);
}

template<class T>
void Set_pointerOrd<T>::unionOp(Set<T>& s1, Set<T>& s2)
{
	this->clear();
	vector<T> elements;

	while (!s1.empty())
	{
		value_type temp = s1.pickAny();
		this->set.insOrd(temp);
		elements.push_back(temp);
		s1.erase(temp);
	}

	while(!elements.empty()) // re-insert the elements in the set
```

```
	{
		s1.insert(elements.back());
		elements.pop_back();
	}

	while (!s2.empty())
	{
		value_type temp = s2.pickAny();
		if(!set.findOrd(temp))
			this->set.insOrd(temp);
		elements.push_back(temp);
		s2.erase(temp);
	}

	while(!elements.empty()) // re-insert the elements in the set
	{
		s2.insert(elements.back());
		elements.pop_back();
	}
}

template<class T>
void Set_pointerOrd<T>::intersection(Set<T>& s1, Set<T>& s2)
{
	this->clear();
	vector<T> elements;

	while (!s1.empty())
	{
		value_type temp = s1.pickAny();
		if(s2.find(temp))
			this->set.insOrd(temp);
		elements.push_back(temp);
		s1.erase(temp);
	}

	while(!elements.empty()) // re-insert the elements in the set
	{
		s1.insert(elements.back());
		elements.pop_back();
	}
}

template<class T>
void Set_pointerOrd<T>::difference(Set<T>& s1, Set<T>& s2)
{
	this->clear();
	vector<T> elements;

	while (!s1.empty())
	{
		value_type temp = s1.pickAny();
		if(!s2.find(temp))
			this->set.insOrd(temp);
		elements.push_back(temp);
		s1.erase(temp);
	}

	while(!elements.empty()) // re-insert the elements in the set
	{
		s1.insert(elements.back());
		elements.pop_back();
```

```
        }

}

template<class T>
typename Set_pointerOrd<T>::value_type Set_pointerOrd<T>::pickAny() const
{
        if(set.empty())
                throw std::logic_error("Set_pointerOrd: exception - Unable to pick an element (empty
set)");

        return set.read(set.begin());
}

#endif // _SETPO_H
```

Insiemi – Realizzazione con vettore booleano

```
#ifndef _SETBOOL_H
#define _SETBOOL_H

#include <stdexcept>
#include <iostream>

using std::cout;
using std::endl;
using std::ostream;

class Set_bool
{
public:

	Set_bool();
	Set_bool(int); // size
	Set_bool(const Set_bool&);
	~Set_bool();

	Set_bool& operator =(const Set_bool&);
	bool operator ==(const Set_bool&);
	bool operator !=(const Set_bool&);

	void create(); // create the set
	bool empty() const; // true if the set is empty
	bool find(const int) const; // true if the element is in the set
	void insert(const int); // add an element to the set
	void erase(const int); // erase the element from the set
	void unionOp(const Set_bool&, const Set_bool&); // return the union of the set
	void intersection(const Set_bool&, const Set_bool&); // return the elements in common
	void difference(const Set_bool&, const Set_bool&); // return the elements of the 1st set not
present in the 2nd set

	int pickAny() const; // take one element from the set

	void clear();
	int size() const;
	int getLen() const; // return length

private:
	bool* set;
	int length;

	void arrayExpansion(bool*&, const int, const int);
};

ostream& operator <<(ostream& os, const Set_bool& s)
{
	if (s.empty())
		os << "Empty Set" << endl;
	else
	{
		os << "[  ";
		for(int i = 0; i < s.getLen(); i++ )
			if(s.find(i))
				os << i << "  ";

		os << "]" << endl;
```

```
	}

	return os;
}

Set_bool::Set_bool()
{
	set = 0;
	length = 10;
	this->create();
}

Set_bool::Set_bool(int dim)
{
	if (dim <= 0)
		dim = 10; // default

	set = 0;
	length = dim;
	this->create();
}

Set_bool::Set_bool(const Set_bool& set)
{
	length = set.length;
	this->create();
	*this = set;
}

Set_bool::~Set_bool()
{
	delete[] set;
}

Set_bool& Set_bool::operator =(const Set_bool& s)
{
	if (&s != this) // avoid auto-assignment
	{
		this->clear();
		length = s.length;
		this->create();
		for (int i = 0; i < s.length; i++)
			set[i] = s.set[i];
	}

	return *this;
}

bool Set_bool::operator ==(const Set_bool& set2)
{
	if(this->size() != set2.size())
		return false;

	for(int i=0; i < length; i++)
		if( set[i] != set2.set[i] )
			return false;

	return true;
}
```

```
bool Set_bool::operator !=(const Set_bool& set2)
{
	return (!(*this == set2));
}


void Set_bool::create()
{
	set = new bool[length];
	for( int i = 0; i < length; i++ )
		set[i] = false;
}

bool Set_bool::empty() const
{
	return (this->size() == 0);
}

bool Set_bool::find(const int a) const
{
	if (a >= length)
		return false;

	return set[a];
}

void Set_bool::insert(const int a)
{
	if (a >= length)
		arrayExpansion(set, length, a+1);

	set[a] = true;
}

void Set_bool::erase(const int a)
{
	if (!set[a])
		throw std::logic_error("Set_bool (exception) - Unable to erase (element not in the
set)");

	set[a] = false;
}

void Set_bool::unionOp(const Set_bool& s1, const Set_bool& s2)
{
	this->clear();

	this->length = s1.length;
	if (s2.length > this->length)
		this->length = s2.length;

	this->create();
	for (int i = 0; i < s1.length; i++)
		if (s1.set[i])
			set[i] = true;

	for (int i = 0; i < s2.length; i++)
		if (s2.set[i])
			set[i] = true;
}
```

```
void Set_bool::intersection(const Set_bool& s1, const Set_bool& s2)
{
	this->clear();

	this->length = s1.length;
	this->create();

	for (int i = 0; i < s1.length; i++)
		if (s1.set[i] && s2.set[i])
			set[i] = true;
}

void Set_bool::difference(const Set_bool& s1, const Set_bool& s2)
{
	this->clear();

	this->length = s1.length;
	this->create();

	for (int i = 0; i < s1.length; i++)
		if (s1.set[i] && !s2.set[i])
			set[i] = true;
}

int Set_bool::pickAny() const
{
	if(this->size() == 0)
		throw std::logic_error("Set_bool (exception) - Unable to pick any element (empty
set)");

	for(int i = 0; true; i++)
		if(set[i] == true)
			return i;

	throw std::logic_error("Set_bool (exception) - Unable to pick any element (corrupted set)");
}

void Set_bool::clear()
{
	for(int i = 0; i < length; i++ )
		set[i] = false;
}

int Set_bool::size() const
{
	int counter = 0;
	for(int i = 0; i < length; i++ )
		if(set[i] == true)
			counter++;

	return counter;
}

void Set_bool::arrayExpansion(bool*& a, const int oldSize, const int newSize)
{
	bool* temp = new bool[newSize];
	for (int i = 0; i < newSize; i++)
		temp[i] = false;

	for (int i = 0; i < oldSize; i++)
		temp[i] = a[i];
```

```
        delete[] a;
        a = temp;

        length = newSize;
}

int Set_bool::getLen() const
{
        return length;
}

#endif // _SETBOOL_H
```

Insiemi – Tester

```
#include "set_pointer.h"

int main ( )
{
	Set_pointer<int> set1, set2, set3;

	set1.insert(1);
	set1.insert(2);
	set1.insert(3);
	set1.insert(4);
	set1.insert(5);

	cout << "set1: " << set1 << endl;

	set2 = set1;
	set2.erase(3);
	set2.erase(4);
	set2.insert(7);
	set2.insert(9);

	cout << "set2: " << set2 << endl;

	set3.unionOp(set1, set2);
	cout << "set3 = set1 + set2: " << set3 << endl;
	cout << "set3 size: " << set3.size() << endl << endl;

	set3.clear();
	cout << "clear set3: " << set3 << endl;

	set3.difference(set2, set1);
	cout << "set3 = set2 - set1: " << set3 << endl;

}
```

Insiemi – Output Tester

```
set1: [  1  2  3  4  5  ]

set2: [  1  2  5  7  9  ]

set3 = set1 + set2: [  1  2  3  4  5  7  9  ]

set3 size: 7

clear set3: Empty Set

set3 = set2 - set1: [  9  7  ]
```

MFSet - Classe astratta

```
#ifndef _MFSET_H
#define _MFSET_H

#include "set.h"

template<class T, class C>
class MFSet
{
public:

	typedef T value_type;
	typedef C component;

	virtual ~MFSet()
	{
	}

	virtual void create(Set<T>&) = 0; // create the MFSet
	virtual void merge (const value_type&, const value_type&) = 0; // merge to two disjoint
components
	virtual component find(const value_type&) const = 0; // return the component in which is
present the element

	bool findSame(const value_type&, const value_type&) const; // true if the elements are in the
same component

};

template<class T, class C>
bool MFSet<T, C>::findSame(const value_type& elem1, const value_type& elem2) const
{
	if(this->find(elem1) == this->find(elem2))
		return true;

	return false;
}

#endif // _MFSET_H
```

MFSet – Realizzazione con lista di liste

```
#ifndef _MFSETLIST_H
#define _MFSETLIST_H

#include "mfset.h"
#include "listap.h"
#include "set_pointer.h"

#include <iostream>
#include <stdexcept>
#include <vector>

using std::endl;
using std::ostream;
using std::vector;

template<class T>
class MFSet_list: public MFSet<T, List_pointer<T> >
{
public:
	typedef typename MFSet<T, List_pointer<T> >::value_type value_type;
	typedef typename MFSet<T, List_pointer<T> >::component component;

	MFSet_list();
	MFSet_list(Set<T>&);
	MFSet_list(const MFSet_list<T>&);
	~MFSet_list();

	template<class E>
	friend ostream& operator <<(ostream& os, const MFSet_list<E>& set);

	MFSet_list<T>& operator =(const MFSet_list<T>&);
	bool operator ==(const MFSet_list<T>&) const;
	bool operator !=(const MFSet_list<T>&) const;

	void create(Set<T>&); // create the MFSet
	void merge(const value_type&, const value_type&); // merge to two disjoint components
	component find(const value_type&) const; // return the component in which is present the
element

private:
	List_pointer<component> components;
};

template<class T>
ostream& operator <<(ostream& os, const MFSet_list<T>& set)
{
	os << "MFSet: " << endl;

	for (typename List_pointer<List_pointer<T> >::position p = set.components.begin();
!set.components.end(p);
			p = set.components.next(p))
		os << set.components.read(p);

	os << endl;

	return os;
}
```

```
template<class T>
MFSet_list<T>::MFSet_list()
{
	components = List_pointer<component>();
}

template<class T>
MFSet_list<T>::MFSet_list(Set<T>& set)
{
	vector<value_type> elements;
	while (!set.empty())
	{
		value_type temp = set.pickAny();
		elements.push_back(temp);
		set.erase(temp);
		component tempList;
		tempList.push_back(temp);
		components.push_back(tempList);
		tempList.clear();
	}

	while (!elements.empty()) // re-insert the elements in the set
	{
		set.insert(elements.back());
		elements.pop_back();
	}
}

template<class T>
MFSet_list<T>::MFSet_list(const MFSet_list<T>& set2)
{
	components = set2.components;
}

template<class T>
MFSet_list<T>::~MFSet_list()
{
}

template<class T>
MFSet_list<T>& MFSet_list<T>::operator =(const MFSet_list<T>& set)
{
	if (this != &set)
	{
		while (!components.empty())
			components.pop_back();

		for (typename component::position p = set.components.begin(); !set.components.end(p);
				p = set.components.next(p))
			components.push_back(set.components.read(p));
	}

	return *this;
}

template<class T>
bool MFSet_list<T>::operator ==(const MFSet_list<T>& mfset2) const
{
	if (components.size() != mfset2.components.size())
		return false;

	/* convert lists (components) in sets
```

```
         *
         * two MFSets are equal if they are made of the same components
         * (the order of the components isn't relevant;
         * the order of the elements in the components isn't relevant)
         *
         * == for set is different from == for list
         * == for list consider the order of the elements
         *
         */

        List_pointer < Set_pointer<T> > l1;
        List_pointer < Set_pointer<T> > l2;

        for(typename List_pointer<component>::position p = components.begin(); !components.end(p); p
= components.next(p))
        {
                component currComp = components.read(p);
                Set_pointer<T> temp;
                for(typename component::position currPos = currComp.begin(); !currComp.end(currPos);
currPos = currComp.next(currPos))
                        temp.insert(currComp.read(currPos));

                l1.push_back(temp);
                temp.clear();
        }

        for(typename List_pointer<component>::position p = mfset2.components.begin();
!mfset2.components.end(p); p = mfset2.components.next(p))
        {
                component currComp = mfset2.components.read(p);
                Set_pointer<T> temp;
                for(typename component::position currPos = currComp.begin(); !currComp.end(currPos);
currPos = currComp.next(currPos))
                                temp.insert(currComp.read(currPos));

                l2.push_back(temp);
                temp.clear();
        }

        for(typename List_pointer<Set_pointer<T> >::position p = l1.begin(); !l1.end(p); p =
l1.next(p))
                if(!l2.find(l1.read(p)))
                        return false;

        /* If the order of the elements in a component is relevant,
         * we can use the operator == for lists (components of the MFSet)
         * that will be called by the method find

                for(typename List_pointer<component>::position p = components.begin();
!components.end(p); p = components.next(p))
                        if(!mfset2.components.find(components.read(p)))
                                return false;
        */

        return true;
}

template<class T>
bool MFSet_list<T>::operator !=(const MFSet_list<T>& mfset2) const
{
        return (!(*this == mfset2));
}
```

```
template<class T>
void MFSet_list<T>::create(Set<T>& set)
{
	while (!components.empty())
		components.pop_back();

	vector<value_type> elements;
	while (!set.empty())
	{
		value_type temp = set.pickAny();
		set.erase(temp);
		component tempList;
		tempList.push_back(temp);
		components.push_back(tempList);
		tempList.clear();
		elements.push_back(temp);
	}

	while (!elements.empty()) // re-insert the elements in the set
	{
		set.insert(elements.back());
		elements.pop_back();
	}
}

template<class T>
void MFSet_list<T>::merge(const value_type& elem1, const value_type& elem2)
{
	if (this->find(elem1) == this->find(elem2))
		throw std::logic_error("MFSet_list (exception) - Unable to merge (non-disjoint 
components)");

	component temp;
	temp.join(this->find(elem1), this->find(elem2));
	components.push_back(temp);
	components.eraseVal(this->find(elem1));
	components.eraseVal(this->find(elem2));
}

template<class T>
typename MFSet_list<T>::component MFSet_list<T>::find(const value_type& val) const
{
	for (typename List_pointer<component>::position p = components.begin(); !components.end(p);
			p = components.next(p))
		if ((components.read(p)).find(val))
			return components.read(p);

	throw std::logic_error("MFSet_list (exception) - Unable to find component");
}

#endif // _MFSETLIST_H
```

MFSet – Realizzazione con insieme di insiemi

```
#ifndef _MFSETSET_H
#define _MFSETSET_H

#include "mfset.h"

#include <iostream>
#include <stdexcept>
#include <vector>

using std::endl;
using std::ostream;
using std::vector;

template<class T>
class MFSet_set: public MFSet<T, Set_pointer<T> >
{
public:
	typedef typename MFSet<T, Set_pointer<T> >::value_type value_type;
	typedef typename MFSet<T, Set_pointer<T> >::component component;

	MFSet_set();
	MFSet_set(Set<T>&);
	MFSet_set(MFSet_set<T>&);
	~MFSet_set();

	template<class E>
	friend ostream& operator <<(ostream&, MFSet_set<E>&);

	MFSet_set<T>& operator =(MFSet_set<T>&);
	bool operator ==(MFSet_set<T>&);
	bool operator !=(MFSet_set<T>&);

	void create(Set<T>&); // create the MFSet
	void merge(const value_type&, const value_type&); // merge to two disjoint components
	component find(const value_type&); // return the component in which is present the element

	component find(const value_type&) const // will always be ignored
	{ // (it's const - impossible to retrieve a set without disassembling it)
		throw std::logic_error("MFSet_set (exception) - Unable to retrieve component");
	}

private:
	Set_pointer<component> components;
};

template<class T>
ostream& operator <<(ostream& os, MFSet_set<T>& set)
{
	os << "MFSet: " << endl;

	MFSet_set<T> temp = set;
	while (!temp.components.empty())
	{
		Set_pointer<T> curr = temp.components.pickAny();
		temp.components.erase(curr);
		os << curr;
	}
```

```
	os << endl;

	return os;
}

template<class T>
MFSet_set<T>::MFSet_set()
{
	components = Set_pointer<component>();
}

template<class T>
MFSet_set<T>::MFSet_set(Set<T>& set)
{
	vector<value_type> elements;
	while (!set.empty())
	{
		value_type temp = set.pickAny();
		elements.push_back(temp);
		set.erase(temp);
		component tempSet;
		tempSet.insert(temp);
		components.insert(tempSet);
		tempSet.clear();
	}

	while (!elements.empty()) // re-insert the elements in the set
	{
		set.insert(elements.back());
		elements.pop_back();
	}
}

template<class T>
MFSet_set<T>::MFSet_set(MFSet_set<T>& set2)
{
	*this = set2;
}

template<class T>
MFSet_set<T>::~MFSet_set()
{
}

template<class T>
MFSet_set<T>& MFSet_set<T>::operator =(MFSet_set<T>& set)
{
	if (this != &set)
	{
		while (!components.empty()) // clear the MFSet
		{
			component temp = components.pickAny();
			components.erase(temp);
		}

		components = set.components;
	}

	return *this;
}
```

```
template<class T>
bool MFSet_set<T>::operator ==(MFSet_set<T>& mfset2)
{
	if (components.size() != mfset2.components.size())
		return false;

	bool flag = true;
	vector<component> backup;

	while(!components.empty())
	{
		component temp = components.pickAny();
		backup.push_back(temp);
		components.erase(temp);

		if(!mfset2.components.find(temp))
			flag = false;
	}

	while(!backup.empty())
	{
		components.insert(backup.back());
		backup.pop_back();
	}

	return flag;
}

template<class T>
bool MFSet_set<T>::operator !=(MFSet_set<T>& mfset2)
{
	return (!(*this == mfset2));
}

template<class T>
void MFSet_set<T>::create(Set<T>& set)
{
	if (!components.empty())
		components.clear();

	vector<value_type> elements;
	while (!set.empty())
	{
		value_type temp = set.pickAny();
		elements.push_back(temp);
		set.erase(temp);
		component tempSet;
		tempSet.insert(temp);
		components.insert(tempSet);
		tempSet.clear();
	}

	while (!elements.empty()) // re-insert the elements in the set
	{
		set.insert(elements.back());
		elements.pop_back();
	}
}
```

```
template<class T>
void MFSet_set<T>::merge(const value_type& elem1, const value_type& elem2)
{
	component comp1 = this->find(elem1);
	component comp2 = this->find(elem2);

	if (comp1 == comp2)
		throw std::logic_error("MFSet_set (exception) - Unable to merge (non-disjoint sets)");

	component temp;
	temp.unionOp(comp1, comp2);
	components.insert(temp);
	components.erase(comp1);
	components.erase(comp2);
}

template<class T>
typename MFSet_set<T>::component MFSet_set<T>::find(const value_type& val)
{
	Set_pointer<component> backup;
	backup = components;
	while (!backup.empty())
	{
		component temp = backup.pickAny();
		backup.erase(temp);
		if (temp.find(val))
			return temp;
	}

	throw std::logic_error("MFSet_set (exception) - Unable to find component");
}

#endif // _MFSETSET_H
```

MFSet – Realizzazione con foresta di alberi n-ari radicati

```
#ifndef _MFSETTREE_H
#define _MFSETTREE_H

#include "mfset.h"
#include "N-aryTree_pointer.h"

template<class T>
class MFSet_tree: public MFSet<T, NaryTree_pointer<T> >
{
public:
	typedef typename MFSet<T, NaryTree_pointer<T> >::value_type value_type;
	typedef typename MFSet<T, NaryTree_pointer<T> >::component component;

	MFSet_tree();
	MFSet_tree(Set<T>&);
	MFSet_tree(const MFSet_tree<T>&);
	~MFSet_tree();

	template<class E>
	friend ostream& operator <<(ostream&, const MFSet_tree<E>&);

	MFSet_tree<T>& operator =(const MFSet_tree<T>&);
	bool operator ==(const MFSet_tree<T>&) const;
	bool operator !=(const MFSet_tree<T>&) const;

	void create(Set<T>&); // create the MFSet
	void merge(const value_type&, const value_type&); // merge to two disjoint components
	component find(const value_type&) const; // return the component in which is present the
element

private:
	List_pointer<component> components;
};

template<class T>
ostream& operator <<(ostream& os, const MFSet_tree<T>& set)
{
	for (typename List_pointer<NaryTree_pointer<T> >::position p = set.components.begin();
!set.components.end(p);
			p = set.components.next(p))
		os << set.components.read(p);

	os << endl;

	return os;
}

template<class T>
MFSet_tree<T>::MFSet_tree()
{
	components = List_pointer<component>();
}

template<class T>
MFSet_tree<T>::MFSet_tree(Set<T>& set)
{
	vector<value_type> elements;
```

```
	while (!set.empty())
	{
		value_type temp = set.pickAny();
		set.erase(temp);
		component tempTree(temp);
		components.push_back(tempTree);
		tempTree.erase(tempTree.root());
		elements.push_back(temp);
	}

	while (!elements.empty()) // re-insert the elements in the set
	{
		set.insert(elements.back());
		elements.pop_back();
	}
}

template<class T>
MFSet_tree<T>::MFSet_tree(const MFSet_tree<T>& set2)
{
	*this = set2;
}

template<class T>
MFSet_tree<T>::~MFSet_tree()
{
}

template<class T>
MFSet_tree<T>& MFSet_tree<T>::operator =(const MFSet_tree<T>& set)
{
	if (this != &set)
	{
		if (!components.empty()) // clear the MFSet
			components.clear();

		components = set.components;
	}

	return *this;
}

template<class T>
bool MFSet_tree<T>::operator ==(const MFSet_tree<T>& mfset2) const
{
	if (components.size() != mfset2.components.size())
		return false;

	/* convert trees (components) in sets
	 *
	 * two MFSets are equal if they are made of the same components
	 * (the order of the components isn't relevant;
	 * the order of the elements in the components isn't relevant)
	 *
	 * == for set is different from == for tree
	 * == for tree consider the order of the elements
	 *
	 */

	List_pointer<Set_pointer<T> > l1;
	List_pointer<Set_pointer<T> > l2;
```

```
	for (typename List_pointer<component>::position p = components.begin(); !components.end(p);
			p = components.next(p))
	{
		component currComp = components.read(p);
		vector<T> elements;
		elements = currComp.elementsArray();
		Set_pointer<T> newSet;
		while (!elements.empty())
		{
			newSet.insert(elements.back());
			elements.pop_back();
		}
		l1.push_back(newSet);
		newSet.clear();
	}

	for (typename List_pointer<component>::position p = mfset2.components.begin();
!mfset2.components.end(p);
			p = mfset2.components.next(p))
	{
		component currComp = mfset2.components.read(p);
		vector<T> elements;
		elements = currComp.elementsArray();
		Set_pointer<T> newSet;
		while (!elements.empty())
		{
			newSet.insert(elements.back());
			elements.pop_back();
		}
		l2.push_back(newSet);
		newSet.clear();
	}

	for (typename List_pointer<Set_pointer<T> >::position p = l1.begin(); !l1.end(p); p =
l1.next(p))
		if (!l2.find(l1.read(p)))
			return false;

	/* If the order of the elements in a component is relevant,
	 * we can use the operator == for lists (components of the MFSet)
	 * that will be called by the method find

	 for(typename List_pointer<component>::position p = components.begin(); !components.end(p); p
= components.next(p))
	 if(!mfset2.components.find(components.read(p)))
	 return false;
	 */

	return true;
}

template<class T>
bool MFSet_tree<T>::operator !=(const MFSet_tree<T>& mfset2) const
{
	return (!(*this == mfset2));
}

template<class T>
void MFSet_tree<T>::create(Set<T>& set)
{
	if (!components.empty())
		components.clear();
```

```
	vector<value_type> elements;
	while (!set.empty())
	{
		value_type temp = set.pickAny();
		set.erase(temp);
		component tempTree(temp);
		components.push_back(tempTree);
		tempTree.erase(tempTree.root());
		elements.push_back(temp);
	}

	while (!elements.empty()) // re-insert the elements in the set
	{
		set.insert(elements.back());
		elements.pop_back();
	}
}

template<class T>
void MFSet_tree<T>::merge(const value_type& elem1, const value_type& elem2)
{
	component tree1 = this->find(elem1);
	component tree2 = this->find(elem2);

	if (tree1 == tree2)
		throw std::logic_error("MFSet_tree (exception) - Unable to merge (non-disjoint
components)");

	components.eraseVal(tree1);
	components.eraseVal(tree2);

	component temp(tree1);

	if(temp.is_leaf(temp.root()))
	{
		temp.insFirstSubTree(temp.root(), tree2);
		components.push_back(temp);
	}
	else
	{
		temp.insSubTree(temp.lastChild(temp.root()), tree2);
		components.push_back(temp);
	}
}

template<class T>
typename MFSet_tree<T>::component MFSet_tree<T>::find(const value_type& val) const
{
	for (typename List_pointer<component>::position p = components.begin(); !components.end(p);
			p = components.next(p))
		if ((components.read(p)).find(val))
			return components.read(p);

	throw std::logic_error("MFSet_tree (exception) - Unable to find component");
}

#endif // _MFSETTREE_H
```

MFSet – Tester

```
#include "mfset_tree.h"

int main()
{
	Set_pointer<int> set;
	set.insert(1);
	set.insert(2);
	set.insert(3);
	set.insert(4);
	set.insert(5);
	cout << "Starting set: " << set << endl;

	MFSet_tree<int> MFSet1(set);
	cout << "MFSet1: " << MFSet1 << endl;

	MFSet_tree<int> MFSet2;
	MFSet2.create(set);
	cout << "MFSet2: " << MFSet2 << endl;

	MFSet1.merge(1, 2);
	MFSet1.merge(1, 4);
	cout << "MFSet1: " << MFSet1 << endl;

	MFSet2.merge(2, 4);
	MFSet2.merge(2, 3);
	cout << "MFSet2: " << MFSet2 << endl;

	if(MFSet1 == MFSet2)
		cout << "MFSet1 = MFSet2" << endl << endl;
	else
		cout << "MFSet1 != MFSet2" << endl << endl;

	MFSet_tree<int> MFSet3(MFSet2);
	cout << "MFSet3: " << MFSet3 << endl;

	if(MFSet2 == MFSet3)
		cout << "MFSet2 = MFSet3" << endl;
	else
		cout << "MFSet2 != MFSet3" << endl;

}
```

MFSet – Output Tester

```
Starting set: [  1  2  3  4  5  ]

MFSet1: [ 1 ]
[ 2 ]
[ 3 ]
[ 4 ]
[ 5 ]

MFSet2: [ 5 ]
[ 4 ]
[ 3 ]
[ 2 ]
[ 1 ]

MFSet1: [ 3 ]
[ 5 ]
[ 1: [ 2 ], [ 4 ] ]

MFSet2: [ 5 ]
[ 1 ]
[ 2: [ 4 ], [ 3 ] ]

MFSet1 != MFSet2

MFSet3: [ 5 ]
[ 1 ]
[ 2: [ 4 ], [ 3 ] ]

MFSet2 = MFSet3
```

Alberi binari – Classe astratta

```
#ifndef _BINTREE_H_
#define _BINTREE_H_

#include <iostream>
#include <deque>
#include <sstream>
#include <stdexcept>
#include <vector>

using std::cout;
using std::endl;
using std::ostream;
using std::vector;

template<class T, class N>
class BinTree
{
public:
	typedef T value_type;
	typedef N node;

	virtual ~BinTree()
	{
	}

	virtual void create() = 0; // create the binary tree
	virtual bool empty() const = 0; // true if the tree is empty
	virtual node root() const = 0; // return the root node
	virtual node parent(node) const = 0; // return the father of the node
	virtual node left(node) const = 0; // return the left son
	virtual node right(node) const = 0; // return the right son
	virtual bool left_empty(node) const = 0; // true if there isn't a left son
	virtual bool right_empty(node) const = 0; // true if there isn't a right son
	virtual void erase(node) = 0; // erase the subtree having root in the node
	virtual value_type read(node) const = 0; // read the element in the node
	virtual void write(node, value_type) = 0; // write the element in the node
	virtual void ins_root(node) = 0; // insert the root
	virtual void ins_left(node) = 0; // insert an empty node in the left son
	virtual void ins_right(node) = 0; // insert an empty node in the right son

	bool operator ==(const BinTree<T, N>&) const;
	bool operator !=(const BinTree<T, N>&) const;

	void printSubTree(const node, ostream&) const; // useful for operator <<
	bool is_leaf(const node) const; // true if the node is a leaf
	int size() const; // return the numbers of nodes of the tree
	bool find(const value_type&) const; // true if the element is present in the tree
	int height() const; // return the height of the tree
	void eraseLeaves(const value_type&); // erases the leaves having that value
	void swap_node(node, node); // swap the values of the two nodes
	value_type min() const; // return the minimum value of tree
	value_type max() const; // return the maximum value of tree
	void ins_subtree(node&, const BinTree<T, N>&, node); // insert a subtree: node must be a
leaf; the 2nd tree not empty
	void build(const BinTree<T, N>&, const BinTree<T, N>&); // build a new tree: implicit tree
must be empty; the other two mustn't
	bool hasSubTree(const BinTree<T, N>&); // true if the parameter tree is a subtree of the
implicit tree
```

```
	vector<value_type> elementsArray() const; // return a vector with the values of the nodes in
the tree
	vector<node> nodesArray() const; // return a vector with the nodes of the tree
	int numberLeaves() const; // number of leaves in the tree

	void preorder() const;
	void inorder() const; //symmetric
	void postorder() const;
	void breadth() const;

private:
	// methods needed for recursive functions
	int size(const node&) const; // return the number of descendants of the node
	bool find(const node&, const value_type&) const; // true if the element is present in the
node or in it's subtree
	int height(const node&) const; // return the height of the tree having root in the node
	void eraseLeaves(node, const value_type&); // erases the leaves having a certain value from
the subtree having root in the node
	value_type min(const node&) const; // return the minimum in the subtree having root in the
node
	value_type max(const node&) const; // return the maximum in the subtree having root in the
node
	bool hasSubTree(const node&, const BinTree<T, N>&, const node&) const; // true if the
parameter tree is a subtree of the implicit tree
	vector<node> findOccurrences(const value_type&); // return an array of nodes whose values are
equal to the parameter
	void elementsArray(const node&, vector<value_type>&) const;
	void nodesArray(const node&, vector<node>&) const;

	void preorder(const node&) const;
	void inorder(const node&) const; //symmetric
	void postorder(const node&) const;
	void breadth(const node&) const;
};

template<class T, class N>
ostream& operator <<(ostream& out, const BinTree<T, N>& tree)
{
	if (!tree.empty())
		tree.printSubTree(tree.root(), out);
	else
		out << "Empty Tree" << endl;

	out << endl;
	return out;
}

template<class T, class N>
bool BinTree<T, N>::operator ==(const BinTree<T, N>& t) const
{
	std::stringstream tree1, tree2;

	tree1 << *this; // save the output of operator << in a string
	tree2 << t; // save the output of operator << in a string

	// compare the two strings
	// equal trees produce the same output for operator <<

	return (tree1.str() == tree2.str());
}
```

```
template<class T, class N>
bool BinTree<T, N>::operator !=(const BinTree<T, N>& tree2) const
{
	return (!(*this == tree2));
}

template<class T, class N>
void BinTree<T, N>::printSubTree(const node n, ostream& out) const
{
	out << "[" << this->read(n) << ", ";

	if (!this->left_empty(n))
		this->printSubTree(this->left(n), out);
	else
		out << "NIL";

	out << ", ";

	if (!this->right_empty(n))
		this->printSubTree(this->right(n), out);
	else
		out << "NIL";

	out << " ]";
}

template<class T, class N>
bool BinTree<T, N>::is_leaf(const node n) const
{
	return (this->left_empty(n) && this->right_empty(n));
}

template<class T, class N>
int BinTree<T, N>::size() const
{
	if (this->empty())
		return 0;

	return (this->size(this->root()) + 1); // +1 because root won't be counted
}

template<class T, class N>
bool BinTree<T, N>::find(const value_type& val) const
{
	return (this->find(this->root(), val));
}

template<class T, class N>
int BinTree<T, N>::height() const
{
	return (this->height(this->root()));
}

template<class T, class N>
void BinTree<T, N>::eraseLeaves(const value_type& val)
{
	this->eraseLeaves(this->root(), val);
}

template<class T, class N>
void BinTree<T, N>::swap_node(node n1, node n2)
{
```

```
	if (this->read(n1) != this->read(n2))
	{
		value_type temp = this->read(n1);
		this->write(n1, this->read(n2));
		this->write(n2, temp);
	}
}

template<class T, class N>
typename BinTree<T, N>::value_type BinTree<T, N>::min() const
{
	return (this->min(this->root()));
}

template<class T, class N>
typename BinTree<T, N>::value_type BinTree<T, N>::max() const
{
	return (this->max(this->root()));
}

template<class T, class N>
void BinTree<T, N>::ins_subtree(node& n1, const BinTree<T, N>& t, node n2)
{
	if (t.empty() || !this->is_leaf(n1))
		throw std::logic_error("BinTree (exception) - Unable to ins_subtree");

	this->write(n1, t.read(n2));

	if (!t.left_empty(n2))
	{
		this->ins_left(n1);
		node temp = this->left(n1);
		this->ins_subtree(temp, t, t.left(n2));
	}

	if (!t.right_empty(n2))
	{
		this->ins_right(n1);
		node temp = this->right(n1);
		this->ins_subtree(temp, t, t.right(n2));
	}
}

template<class T, class N>
void BinTree<T, N>::build(const BinTree<T, N>& t1, const BinTree<T, N>& t2)
{
	if (!this->empty() || t1.empty() || t2.empty())
		throw std::logic_error("BinTree (exception) - Unable to build tree");

	// prepare the new tree
	node n = 0;
	this->ins_root(n);
	n = this->root();
	this->ins_left(n);
	this->ins_right(n);

	//copy the two trees
	node temp = this->left(n);
	this->ins_subtree(temp, t1, t1.root());
	temp = this->right(n);
	this->ins_subtree(temp, t2, t2.root());
}
```

```
template<class T, class N>
bool BinTree<T, N>::hasSubTree(const BinTree<T, N>& t2)
{
	if (t2.empty())
		return true;

	if (this->empty())
		throw std::logic_error("BinTree (exception) - Unable to check if tree has a subtree");

	//1st !empty e 2nd !empty
	if (this->size() < t2.size())
		return false;

	//1st !empty >= 2nd !empty
	if (t2.size() == 1 && this->find(t2.read(t2.root())))
		return true;
	if (t2.size() == 1 && !this->find(t2.read(t2.root())))
		return false;

	//1st !empty >= 2nd (>1)
	vector<node> nodes = this->findOccurrences(t2.read(t2.root())); // get all the nodes having 
the same value

	bool flag = 0;
	for (typename vector<node>::iterator it = nodes.begin(); it != nodes.end() && !flag; it++)
		flag = flag + hasSubTree(*it, t2, t2.root()); // check if there is the parameter 
subtree at least starting from one node

	return flag;
}

template<class T, class N>
vector<T> BinTree<T, N>::elementsArray() const
{
	vector<T> elements;
	if (!this->empty())
		elementsArray(this->root(), elements);

	return elements;
}

template<class T, class N>
vector<N> BinTree<T, N>::nodesArray() const
{
	vector<N> nodes;
	if (!this->empty())
		nodesArray(this->root(), nodes);

	return nodes;
}

template<class T, class N>
int BinTree<T, N>::numberLeaves() const
{
	int counter = 0;
	vector<node> nodes = this->nodesArray();

	for (typename vector<node>::iterator it = nodes.begin(); it != nodes.end(); it++)
		if (this->is_leaf(*it))
			counter++;
	return counter;
}
```

```
template<class T, class N>
void BinTree<T, N>::preorder() const
{
	if (this->empty())
		throw std::logic_error("BinTree (exception) - Unable to perform preorder visit (empty
tree)");

	this->preorder(this->root());
}

template<class T, class N>
void BinTree<T, N>::inorder() const
{
	if (this->empty())
		throw std::logic_error("BinTree (exception) - Unable to perform inorder visit (empty
tree)");

	this->inorder(this->root());
}

template<class T, class N>
void BinTree<T, N>::postorder() const
{
	if (this->empty())
		throw std::logic_error("BinTree (exception) - Unable to perform postorder visit (empty
tree)");

	this->postorder(this->root());
}

template<class T, class N>
void BinTree<T, N>::breadth() const
{
	if (this->empty())
		throw std::logic_error("BinTree (exception) - Unable to perform breadth visit (empty
tree)");

	this->breadth(this->root());
}

template<class T, class N>
void BinTree<T, N>::preorder(const node& n) const
{
	cout << this->read(n) << " ";

	if (!this->left_empty(n))
		this->preorder(this->left(n));

	if (!this->right_empty(n))
		this->preorder(this->right(n));
}

template<class T, class N>
void BinTree<T, N>::inorder(const node& n) const //symmetric
{
	if (!this->left_empty(n))
		this->inorder(this->left(n));

	cout << this->read(n) << " ";
	if (!this->right_empty(n))
		this->inorder(this->right(n));
}
```

```
template<class T, class N>
void BinTree<T, N>::postorder(const node& n) const
{
	if (!this->left_empty(n))
		this->postorder(this->left(n));

	if (!this->right_empty(n))
		this->postorder(this->right(n));

	cout << this->read(n) << " ";
}

template<class T, class N>
void BinTree<T, N>::breadth(const node& n) const
{
	node temp = 0;
	std::deque<node> nodes;

	nodes.push_back(n);

	while (!nodes.empty())
	{
		temp = nodes.front();
		cout << this->read(temp) << " ";
		nodes.pop_front();

		if (!this->left_empty(temp))
			nodes.push_back(this->left(temp));

		if (!this->right_empty(temp))
			nodes.push_back(this->right(temp));
	}
}

template<class T, class N>
int BinTree<T, N>::size(const node& n) const
{
	int num = 0;
	node curr = n;
	std::deque<node> nodes;

	nodes.push_back(curr);

	while (!nodes.empty())
	{
		curr = nodes.front();
		num++;

		if (!this->left_empty(curr))
			nodes.push_back(this->left(curr));

		if (!this->right_empty(curr))
			nodes.push_back(this->right(curr));

		nodes.pop_front();
	}

	num--; // the starting node must not be counted
	return num;
}
```

```
template<class T, class N>
bool BinTree<T, N>::find(const node& n, const value_type& val) const
{
	if (this->read(n) == val)
		return true;

	// we have to search in the sons, if they are present
	if (!this->left_empty(n) && !this->right_empty(n))
		return (this->find(this->left(n), val) || this->find(this->right(n), val));

	if (!this->left_empty(n) && this->right_empty(n))
		return (this->find(this->left(n), val));

	if (this->left_empty(n) && !this->right_empty(n))
		return (this->find(this->right(n), val));

	return false;
}

template<class T, class N>
int BinTree<T, N>::height(const node& n) const
{
	if (this->empty())
		throw std::logic_error("BinTree (exception) - Unable to calculate height of the tree
(empty tree)");

	if (this->is_leaf(n))
		return 1; // 0 to have height decreased by 1
	else
	{
		int leftDepth = 0;
		int rightDepth = 0;

		// find each depth
		if (!this->left_empty(n))
			leftDepth = this->height(this->left(n));
		if (!this->right_empty(n))
			rightDepth = this->height(this->right(n));

		// use the biggest one
		if (leftDepth > rightDepth)
			return (leftDepth + 1);
		else
			return (rightDepth + 1);
	}
}

template<class T, class N>
void BinTree<T, N>::eraseLeaves(node n, const value_type& val)
{
	if (this->empty())
		throw std::logic_error("BinTree (exception) - Unable to erase leaves visit (empty
tree)");

	if (!this->is_leaf(n))
	{
		if (!this->left_empty(n))
			this->eraseLeaves(n->getLeft(), val);
		if (!this->right_empty(n))
			this->eraseLeaves(n->getRight(), val);
	}
	else
```

```
	{
		if (this->read(n) == val)
			this->erase(n);
	}
}

template<class T, class N>
typename BinTree<T, N>::value_type BinTree<T, N>::min(const node& n) const
{
	if (this->is_leaf(n))
		return (this->read(n));
	else
	{
		if (!this->left_empty(n) && !this->right_empty(n))
			return (this->read(n) < this->min(this->left(n)) ? (
						this->read(n) < this->min(this->right(n)) ? this->read(n) :
this->min(this->right(n))) : (
						this->min(this->right(n)) < this->min(this->left(n)) ? this-
>min(this->right(n)) : this->min(this->left(n)))); //return minimum between read(n), min(left(n))
and min(right(n))

		if (!this->left_empty(n) && this->right_empty(n))
			return (this->read(n) < this->min(this->left(n)) ? this->read(n) : this-
>min(this->left(n))); //return minimum between read(n) and min(left(n))

		if (this->left_empty(n) && !this->right_empty(n))
			return (this->read(n) < this->min(this->right(n)) ? this->read(n) : this-
>min(this->right(n))); //return minimum between read(n) and min(right(n))
	}

	throw std::logic_error("BinTree (exception) - Unable to retrieve minimum value");
}

template<class T, class N>
typename BinTree<T, N>::value_type BinTree<T, N>::max(const node& n) const
{
	if (this->is_leaf(n))
		return (this->read(n));
	else
	{
		if (!this->left_empty(n) && !this->right_empty(n))
			return (this->read(n) > this->max(this->left(n)) ? (
						this->max(this->right(n)) > this->read(n) ? this->max(this-
>right(n)) : this->read(n)) : (
						this->max(this->right(n)) > this->max(this->left(n)) ? this-
>max(this->right(n)) : this->max(this->left(n)))); //return maximum between read(n), max(left(n))
and max(right(n))

		if (!this->left_empty(n) && this->right_empty(n))
			return (this->read(n) > this->max(this->left(n)) ? this->read(n) : this-
>max(this->left(n))); //return maximum between read(n) and max(left(n))

		if (this->left_empty(n) && !this->right_empty(n))
			return (this->read(n) > this->max(this->right(n)) ? this->read(n) : this-
>max(this->right(n))); //return maximum between read(n) and max(right(n))
	}

	throw std::logic_error("BinTree (exception) - Unable to retrieve maximum value");
}
```

```
template<class T, class N>
bool BinTree<T, N>::hasSubTree(const node& n1, const BinTree<T, N>& t2, const node& n2) const
{
	if (!t2.left_empty(n2) && this->left_empty(n1))
		return false;

	if (!t2.right_empty(n2) && this->right_empty(n1))
		return false;

	// node is leaf (same number of children)
	if ((this->is_leaf(n1) && t2.is_leaf(n2)) && (this->read(n1) == t2.read(n2)))
		return true;
	if ((this->is_leaf(n1) && t2.is_leaf(n2)) && (this->read(n1) != t2.read(n2)))
		return false;


	if ((!this->left_empty(n1)) && (!t2.left_empty(n2) && t2.right_empty(n2)))
		return (this->hasSubTree(this->left(n1), t2, t2.left(n2)));

	if ((!this->right_empty(n1)) && (t2.left_empty(n2) && !t2.right_empty(n2)))
		return (this->hasSubTree(this->right(n1), t2, t2.right(n2)));

	if ((!this->left_empty(n1) && !this->right_empty(n1)) && (!t2.left_empty(n2) &&
!t2.right_empty(n2)))
		return (this->hasSubTree(this->left(n1), t2, t2.left(n2)) && this->hasSubTree(this-
>right(n1), t2, t2.right(n2)));

	return false;
}

template<class T, class N>
vector<N> BinTree<T, N>::findOccurrences(const value_type& val)
{
	vector<node> occurrences;

	if (!this->empty())
	{
		node temp = this->root();
		std::deque<node> nodes;

		nodes.push_back(temp);

		while (!nodes.empty())
		{
			temp = nodes.front();

			if (val == this->read(temp))
				occurrences.push_back(temp);

			nodes.pop_front();

			if (!this->left_empty(temp))
				nodes.push_back(this->left(temp));

			if (!this->right_empty(temp))
				nodes.push_back(this->right(temp));
		}

	}
	return occurrences;
}
```

```
template<class T, class N>
void BinTree<T, N>::elementsArray(const node& n, vector<value_type>& elements) const
{
	elements.push_back(this->read(n));

	if (!this->left_empty(n))
		elementsArray(this->left(n), elements);

	if (!this->right_empty(n))
		elementsArray(this->right(n), elements);
}

template<class T, class N>
void BinTree<T, N>::nodesArray(const node& n, vector<node>& nodes) const
{
	nodes.push_back(n);

	if (!this->left_empty(n))
		nodesArray(this->left(n), nodes);

	if (!this->right_empty(n))
		nodesArray(this->right(n), nodes);
}

#endif /* _BINTREE_H_ */
```

Alberi binari – Realizzazione con cursori

```
#ifndef _BINTREEC_H
#define _BINTREEC_H

#include "Bin_tree.h"

template<class T>
class BinTree_cursor: public BinTree<T, int>
{
	static const int NIL = -1;

public:

	typedef typename BinTree<T, int>::value_type value_type;
	typedef typename BinTree<T, int>::node node;

	typedef struct _cella
	{
		node father;
		node left;
		node right;
		value_type value;
	} Cell;

	BinTree_cursor();
	BinTree_cursor(int); // size
	BinTree_cursor(const BinTree_cursor<T>&);
	~BinTree_cursor();

	BinTree_cursor<T>& operator =(const BinTree_cursor<T>&);

	void create(); // create the binary tree
	bool empty() const; // true if the tree is empty
	node root() const; // return the root node
	node parent(node) const; // return the father of the node
	node left(node) const; // return the left son
	node right(node) const; // return the right son
	bool left_empty(node) const; // true if there isn't a left son
	bool right_empty(node) const; // true if there isn't a right son
	void erase(node); // erase the subtree having root = node
	value_type read(node) const; // read the element in the node
	void write(node, value_type); // write the element in the node
	void ins_root(node); // insert the root
	void ins_left(node); // insert an empty node in the left son
	void ins_right(node); // insert an empty node in the right son

private:
	int MAXLENGTH;
	Cell* space;
	int nodesNum;
	node start;
	node freeCell;

	void arrayDoubling(Cell*&, const int, const int);
};

template<class T>
BinTree_cursor<T>::BinTree_cursor()
{
```

```
	MAXLENGTH = 100;
	space = new Cell[MAXLENGTH];
	nodesNum = 0;
	this->create();
}

template<class T>
BinTree_cursor<T>::BinTree_cursor(int size)
{
	if (size <= 0)
		throw std::logic_error("BinTree_cursor (exception) - Unable to initialise (non-
positive size)");

	MAXLENGTH = size;
	space = new Cell[size];
	nodesNum = 0;
	this->create();
}

template<class T>
BinTree_cursor<T>::BinTree_cursor(const BinTree_cursor<T>& tree)
{
	nodesNum = 0;
	*this = tree;
}

template<class T>
BinTree_cursor<T>::~BinTree_cursor()
{
	if (!this->empty())
		this->erase(this->root());

	delete[] space;
	MAXLENGTH = 0;
	nodesNum = 0;
	start = 0;
	freeCell = 0;
}

template<class T>
BinTree_cursor<T>& BinTree_cursor<T>::operator =(const BinTree_cursor<T>& tree)
{
	if (&tree != this) // avoid auto-assignment
	{
		if (!this->empty())
		{
			this->erase(this->root());
			delete[] space;
		}

		MAXLENGTH = tree.MAXLENGTH;
		nodesNum = tree.nodesNum;
		start = tree.start;
		freeCell = tree.freeCell;
		space = new Cell[MAXLENGTH];

		for (int i = 0; i < MAXLENGTH; i++)
			space[i] = tree.space[i];
	}

	return *this;
}
```

```
template<class T>
void BinTree_cursor<T>::create()
{
	start = NIL;

	for (int i = 0; i < MAXLENGTH; i++)
	{
		space[i].left = (i + 1) % MAXLENGTH; // we save in "left" field an index of the next
(free) position in space[]
		space[i].value = value_type();
	}

	freeCell = 0;
	nodesNum = 0;
}

template<class T>
bool BinTree_cursor<T>::empty() const
{
	return (nodesNum == 0);
}

template<class T>
typename BinTree_cursor<T>::node BinTree_cursor<T>::root() const
{
	if (this->empty())
		throw std::logic_error("BinTree_cursor (exception) - Unable to read root (empty
tree)");

	return start;
}

template<class T>
typename BinTree_cursor<T>::node BinTree_cursor<T>::parent(node n) const
{
	if (n != start)
		return space[n].father;
	else
		return n;
}

template<class T>
typename BinTree_cursor<T>::node BinTree_cursor<T>::left(node n) const
{
	if (!this->left_empty(n))
		return (space[n].left);
	else
		return n;
}

template<class T>
typename BinTree_cursor<T>::node BinTree_cursor<T>::right(node n) const
{
	if (!this->right_empty(n))
		return (space[n].right);
	else
		return n;
}
```

```
template<class T>
bool BinTree_cursor<T>::left_empty(node n) const
{
	return (space[n].left == NIL);
}

template<class T>
bool BinTree_cursor<T>::right_empty(node n) const
{
	return (space[n].right == NIL);
}

template<class T>
void BinTree_cursor<T>::erase(node n)
{
	if (n != NIL)
	{
		if (!this->left_empty(n))
			this->erase(space[n].left);
		if (!this->right_empty(n))
			this->erase(space[n].right);
		if (n != start)
		{
			node p = this->parent(n);
			if (space[p].left == n)
				space[p].left = NIL;
			else
				space[p].right = NIL;
		}
		else
			start = NIL;
		nodesNum--;
		space[n].left = freeCell;
		freeCell = n;
	}
	else
		throw std::logic_error("BinTree_cursor (exception) - Unable to erase (null node)");
}

template<class T>
typename BinTree_cursor<T>::value_type BinTree_cursor<T>::read(node n) const
{
	return space[n].value;
}

template<class T>
void BinTree_cursor<T>::write(node n, value_type a)
{
	space[n].value = a;
}

template<class T>
void BinTree_cursor<T>::ins_root(node n)
{
	if (start != NIL)
		throw std::logic_error("BinTree_cursor (exception) - Unable to insert root (already
exists)");

	start = freeCell; // free = 0
	freeCell = space[freeCell].left; // set the next free space
	space[start].left = NIL; // set left son
	space[start].right = NIL; // set right son
```

```
	space[start].father = NIL; // set father
	nodesNum++;

	if (n != 0)
		this->write(this->root(), this->read(n));
}

template<class T>
void BinTree_cursor<T>::ins_left(node n)
{
	if (start == NIL)
		throw std::logic_error("BinTree_cursor (exception) - Unable to insert node (empty tree)");
	if (n == NIL)
		throw std::logic_error("BinTree_cursor (exception) - Unable to insert node (null node)");
	if (space[n].left != NIL)
		throw std::logic_error("BinTree_cursor (exception) - Unable to insert node (already exists)");

	if (nodesNum >= MAXLENGTH)
		this->arrayDoubling(space, MAXLENGTH, MAXLENGTH*2);

	node q = freeCell; // q = position of the son
	freeCell = space[freeCell].left; // set the next free space
	space[n].left = q; // set the left son
	space[q].father = n;
	space[q].left = NIL; // set an empty son
	space[q].right = NIL; // set an empty son
	nodesNum++;
}

template<class T>
void BinTree_cursor<T>::ins_right(node n)
{
	if (start == NIL)
		throw std::logic_error("BinTree_cursor (exception) - Unable to insert node (empty tree)");
	if (n == NIL)
		throw std::logic_error("BinTree_cursor (exception) - Unable to insert node (null node)");
	if (space[n].right != NIL)
		throw std::logic_error("BinTree_cursor (exception) - Unable to insert node (already exists)");

	if (nodesNum >= MAXLENGTH)
		this->arrayDoubling(space, MAXLENGTH, MAXLENGTH*2);

	node q = freeCell; // position of the son
	freeCell = space[freeCell].left; // set the next free space
	space[n].right = q; // set the right son
	space[q].father = n;
	space[q].left = NIL; // set an empty son
	space[q].right = NIL; // set an empty son
	nodesNum++;
}
```

```
template<class T>
void BinTree_cursor<T>::arrayDoubling(Cell*& a, const int vecchiaDim, const int nuovaDim)
{
	Cell* temp = new Cell[nuovaDim];

	int number;
	if (vecchiaDim < nuovaDim)
		number = vecchiaDim;
	else
		number = nuovaDim;

	freeCell = MAXLENGTH;
	MAXLENGTH = nuovaDim;

	for (int i = 0; i < nuovaDim; i++)
	{
		temp[i].left = (i + 1) % nuovaDim;
		temp[i].value = value_type();
	}

	for (int i = 0; i < number; i++)
		temp[i] = a[i];

	delete[] a;
	a = temp;
}

#endif // _BINTREEC_H
```

Alberi binari – Realizzazione con puntatori

```
#ifndef _BINTREEP_H
#define _BINTREEP_H

#include "cellbt.h"
#include "Bin_tree.h"

template<class T>
class BinTree_pointer: public BinTree<T, Cell<T>*>
{
public:
	typedef typename BinTree<T, Cell<T>*>::value_type value_type;
	typedef typename BinTree<T, Cell<T>*>::node node;

	BinTree_pointer();
	BinTree_pointer(const value_type&); // root
	BinTree_pointer(const BinTree_pointer<T>&);
	~BinTree_pointer();

	BinTree_pointer<T>& operator =(const BinTree_pointer<T>&);

	void create(); // create the binary tree
	bool empty() const; // true if the tree is empty
	node root() const; // return the root node
	node parent(node) const; // return the father of the node
	node left(node) const; // return the left son
	node right(node) const; // return the right son
	bool left_empty(node) const; // true if there isn't a left son
	bool right_empty(node) const; // true if there isn't a right son
	void erase(node); // erase the subtree having root = node
	value_type read(node) const; // read the element in the node
	void write(node, value_type); // write the element in the node
	void ins_root(node); // insert the root
	void ins_left(node); // insert an empty node in the left son
	void ins_right(node); // insert an empty node in the right son

	void mutation(node, BinTree_pointer<T>&, node); // swap two subtrees from different trees
	bool isLeftSon(node) const; // true if the node is the left son of his parent
	bool isRightSon(node) const; // true if the node is the right son of his parent

private:
	node tree_root;
};

template<class T>
BinTree_pointer<T>::BinTree_pointer()
{
	this->create();
}

template<class T>
BinTree_pointer<T>::BinTree_pointer(const value_type& value)
{
	this->create();
	node newroot = new Cell<value_type>(value);
	this->ins_root(newroot);
}
```

```
template<class T>
BinTree_pointer<T>::BinTree_pointer(const BinTree_pointer<T>& tree)
{
	*this = tree;
}

template<class T>
BinTree_pointer<T>::~BinTree_pointer()
{
	this->erase(this->root());
}

template<class T>
BinTree_pointer<T>& BinTree_pointer<T>::operator =(const BinTree_pointer<T>& tree)
{
	if (&tree != this) // avoid auto-assignment
	{
		if (!this->empty())
			this->erase(this->root());

		node newroot = new Cell<value_type>;
		this->ins_root(newroot);
		newroot = this->root();
		this->ins_subtree(newroot, tree, tree.root());
	}

	return *this;
}

template<class T>
void BinTree_pointer<T>::create()
{
	tree_root = 0;
}

template<class T>
bool BinTree_pointer<T>::empty() const
{
	return (this->root() == 0);
}

template<class T>
typename BinTree_pointer<T>::node BinTree_pointer<T>::root() const
{
	return tree_root;
}

template<class T>
typename BinTree_pointer<T>::node BinTree_pointer<T>::parent(node n) const
{
	if (n == this->root())
		throw std::logic_error("BinTree_pointer (exception) - Unable to retrieve parent (node
is root)");

	return n->getFather();
}

template<class T>
typename BinTree_pointer<T>::node BinTree_pointer<T>::left(node n) const
{
	if (this->left_empty(n))
```

```
		throw std::logic_error("BinTree_pointer (exception) - Unable to retrieve left son 
(node is left_empty)");

	return n->getLeft();
}

template<class T>
typename BinTree_pointer<T>::node BinTree_pointer<T>::right(node n) const
{
	if (this->right_empty(n))
		throw std::logic_error("BinTree_pointer (exception) - Unable to retrieve right son 
(node is right_empty)");

	return n->getRight();
}

template<class T>
bool BinTree_pointer<T>::left_empty(node n) const
{
	if (n == 0)
		throw std::logic_error("BinTree_pointer (exception) - Unable to check if node has a 
left son (invalid node)");

	return (n->getLeft() == 0);
}

template<class T>
bool BinTree_pointer<T>::right_empty(node n) const
{
	if (n == 0)
		throw std::logic_error("BinTree_pointer (exception) - Unable to check if node has a 
right son (invalid node)");

	return (n->getRight() == 0);
}

template<class T>
void BinTree_pointer<T>::erase(node n)
{
	if (!this->empty())
	{
		if (!this->is_leaf(n))
		{
			if (!this->left_empty(n))
				this->erase(n->getLeft());

			if (!this->right_empty(n))
				this->erase(n->getRight());
		}

		if (n != this->root())
		{
			if (this->isLeftSon(n))
				(parent(n))->setLeft(NULL);
			else if (isRightSon(n))
				(parent(n))->setRight(NULL);
			delete (n);
			n = 0;
		}
		else
		{
			delete tree_root;
```

```
                    tree_root = 0;
            }
        }
}

template<class T>
typename BinTree<T, Cell<T>*>::value_type BinTree_pointer<T>::read(node n) const
{
        if (n == 0)
                throw std::logic_error("BinTree_pointer (exception) - Unable to read node (invalid
node)");

        return n->getValue();
}

template<class T>
void BinTree_pointer<T>::write(node n, value_type value)
{
        if (n == 0)
                throw std::logic_error("BinTree_pointer (exception) - Unable to write the value in the
node (invalid node)");

        n->setValue(value);
}

template<class T>
void BinTree_pointer<T>::ins_root(node n = 0)
{
        if (n == 0)
                tree_root = new Cell<value_type>;
        else
                tree_root = new Cell<value_type>(n->getValue());
}

template<class T>
void BinTree_pointer<T>::ins_left(node n)
{
        if (!this->left_empty(n) || n == 0)
                throw std::logic_error("BinTree_pointer (exception) - Unable to insert left son");

        node son = new Cell<value_type>;
        son->setFather(n);
        n->setLeft(son);
}

template<class T>
void BinTree_pointer<T>::ins_right(node n)
{
        if (!this->right_empty(n) || n == 0)
                throw std::logic_error("BinTree_pointer (exception) - Unable to insert right son");

        node son = new Cell<value_type>;
        son->setFather(n);
        n->setRight(son);
}

template<class T>
void BinTree_pointer<T>::mutation(node n1, BinTree_pointer<T>& tree2, node n2)
{
        if (n1 == 0 || n2 == 0)
                throw std::logic_error("BinTree_pointer (exception) - Unable to perform mutation
(invalid node)");
```

```
        BinTree_pointer temp(0);

        temp.ins_subtree(temp.root(), *this, n1);

        if (!this->left_empty(n1))
                this->erase(n1->getLeft());
        if (!this->right_empty(n1))
                this->erase(n1->getRight());

        this->ins_subtree(n1, tree2, n2);

        if (!tree2.left_empty(n2))
                tree2.erase(n2->getLeft());
        if (!tree2.right_empty(n2))
                tree2.erase(n2->getRight());
        tree2.ins_subtree(n2, temp, temp.root());
}

template<class T>
bool BinTree_pointer<T>::isLeftSon(node n) const
{
        if (n == 0 || n == this->root())
                throw std::logic_error("BinTree_pointer (exception) - Unable to check if node is left
son");

        return (n == (n->getFather())->getLeft());
}

template<class T>
bool BinTree_pointer<T>::isRightSon(node n) const
{
        if (n == 0 || n == this->root())
                throw std::logic_error("BinTree_pointer (exception) - Unable to check if node is right
son");

        return (n == (n->getFather())->getRight());
}

#endif // _BINTREEP_H
```

Alberi binari – Realizzazione con puntatori
File: "cellbt.h"

```
#ifndef _CELLBT_H
#define _CELLBT_H

template<class T>
class Cell
{
public:

	typedef T value_type;
	typedef Cell* node;

	Cell();
	Cell(const value_type&);
	~Cell();

	Cell<T>& operator =(const Cell<T>&);
	bool operator ==(const Cell<T>&) const;
	bool operator !=(const Cell<T>&) const;

	void setValue(const value_type);
	value_type getValue() const;
	void setFather(node);
	void setRight(node);
	void setLeft(node);
	node getFather() const;
	node getRight() const;
	node getLeft() const;

private:
	value_type value;
	node father;
	node right;
	node left;
};

template<class T>
Cell<T>::Cell()
{
	value = value_type();
	father = 0;
	right = 0;
	left = 0;
}

template<class T>
Cell<T>::Cell(const value_type& val)
{
	value = val;
	father = 0;
	right = 0;
	left = 0;
}

template<class T>
Cell<T>::~Cell()
{
	value = value_type();
```

```
        father = 0;
        right = 0;
        left = 0;
}

template<class T>
Cell<T>& Cell<T>::operator =(const Cell<T>& c2)
{
        if (&c2 != this) // avoid auto-assignment
        {
                this->setValue(c2.getValue());
                this->setFather(c2.getFather());
                this->setLeft(c2.getLeft());
                this->setRight(c2.getRight());
        }
        return *this;
}

template<class T>
bool Cell<T>::operator ==(const Cell<T>& c2) const
{
        if (this->value != c2.value)
                return false;
        if (this->father != c2.father)
                return false;
        if (this->right != c2.right)
                return false;
        if (this->left != c2.left)
                return false;

        return true;
}

template<class T>
bool Cell<T>::operator !=(const Cell<T>& c2) const
{
        return (!(c2 == *this));
}

template<class T>
void Cell<T>::setValue(const value_type element)
{
        value = element;
}

template<class T>
typename Cell<T>::value_type Cell<T>::getValue() const
{
        return value;
}

template<class T>
void Cell<T>::setFather(node fatherp)
{
        father = fatherp;
}

template<class T>
void Cell<T>::setRight(node rightp)
{
        right = rightp;
}
```

```cpp
template<class T>
void Cell<T>::setLeft(node leftp)
{
	left = leftp;
}

template<class T>
typename Cell<T>::node Cell<T>::getFather() const
{
	return father;
}

template<class T>
typename Cell<T>::node Cell<T>::getRight() const
{
	return right;
}

template<class T>
typename Cell<T>::node Cell<T>::getLeft() const
{
	return left;
}

#endif // _CELLBT_H
```

Alberi binari – Tester

```
#include "bintree_cursor.h"

int main()
{
	BinTree_cursor<int> t(3), t2(3), t3, t4;
	BinTree_cursor<int>::node n = 0, n2 = 0;

	t.ins_root(n);
	n = t.root();
	t.write(n, 1);
	t.ins_right(n);
	n = t.right(n);
	t.write(n, 2);
	n = t.parent(n);
	t.ins_left(n);
	n = t.left(n);
	t.write(n, 3);

	t2.ins_root(n2);
	n2 = t2.root();
	t2.write(n2, 5);
	t2.ins_right(n2);
	n2 = t2.right(n2);
	t2.write(n2, 6);
	n2 = t2.parent(n2);
	t2.ins_left(n2);
	n2 = t2.left(n2);
	t2.write(n2, 7);
	t2.ins_left(n2);
	t2.ins_right(n2);
	n2 = t2.left(n2);
	t2.write(n2, 8);
	n2 = t2.parent(n2);
	n2 = t2.right(n2);
	t2.write(n2, 9);

	cout << "t: " << t << endl;;
	cout << "t2: " << t2 << endl;;

	t3.build(t, t2);
	cout << "t3: " << t3 << endl;

	t4 = t3;
	cout << "t4: " << t4;
	if(t4 == t3)
		cout << "t4 and t3 are equal" << endl << endl;

	cout << "t3 number of nodes: " << t3.size();
	cout << endl;
	cout << "t3 number of leaves: " << t3.numberLeaves();
	cout << endl;
	cout << "t3 height: " << t3.height();
	cout << endl;
	cout << "t3 number of leaves: " << t3.numberLeaves();
	cout << endl;
	cout << "t3 minimum label: " << t3.min();
	cout << endl;
	cout << "t3 maximum label: " << t3.max();
```

```
        cout << endl;

        cout << "t3 preorder visit: ";
        t3.preorder();
        cout << endl;

        cout << "t3 inorder visit: ";
        t3.inorder();
        cout << endl;

        cout << "t3 postorder visit: ";
        t3.postorder();
        cout << endl;

        cout << "t3 breadth visit: ";
        t3.breadth();
        cout << endl;

}
```

Alberi binari – Output tester

```
t: [1, [3, NIL, NIL ], [2, NIL, NIL ] ]

t2: [5, [7, [8, NIL, NIL ], [9, NIL, NIL ] ], [6, NIL, NIL ] ]

t3: [0, [1, [3, NIL, NIL ], [2, NIL, NIL ] ], [5, [7, [8, NIL, NIL ], [9, NIL, NIL ] ],
[6, NIL, NIL ] ] ]

t4: [0, [1, [3, NIL, NIL ], [2, NIL, NIL ] ], [5, [7, [8, NIL, NIL ], [9, NIL, NIL ] ],
[6, NIL, NIL ] ] ]
t4 and t3 are equal

t3 number of nodes: 9
t3 number of leaves: 5
t3 height: 4
t3 number of leaves: 5
t3 minimum label: 0
t3 maximum label: 9
t3 preorder visit: 0 1 3 2 5 7 8 9 6
t3 inorder visit: 3 1 2 0 8 7 9 5 6
t3 postorder visit: 3 2 1 8 9 7 6 5 0
t3 breadth visit: 0 1 5 3 2 7 6 8 9
```

Alberi binari di ricerca – Classe astratta

```
#ifndef _BST_H_
#define _BST_H_

#include <iostream>
#include <deque>
#include <sstream>
#include <stdexcept>

using std::cout;
using std::endl;
using std::ostream;

template<class T, class N> //Values are keys
class BST
{
public:
	typedef T value_type;
	typedef N node;

	virtual ~BST()
	{
	}

	virtual void insert(value_type) = 0; // insert the element in the first available node (if it isn't already present in the tree)
	virtual void erase(value_type) = 0; // erase the node with that value (values are key)

	template<class X, class Y>
	friend ostream& operator<<(ostream &, const BST<X, Y>&);

	bool operator ==(const BST<T, N>&) const;
	bool operator !=(const BST<T, N>&) const;

	int size() const; // return the numbers of nodes of the tree
	bool find(const value_type&) const; // true if the element is present in the tree
	value_type min() const;
	value_type max() const;

	void preorder() const;
	void inorder() const; //symmetric
	void postorder() const;
	void breadth() const;

protected:
	virtual void create() = 0; // create the BST
	virtual bool empty() const = 0; // true if the tree is empty
	virtual node root() const = 0; // return the root node
	virtual node parent(node) const = 0; // return the father of the node
	virtual node left(node) const = 0; // return the left son
	virtual node right(node) const = 0; // return the right son
	virtual bool left_empty(node) const = 0; // true if there isn't a left son
	virtual bool right_empty(node) const = 0; // true if there isn't a right son
	virtual void erase(node) = 0; // erase the subtree having root in the node
	virtual value_type read(node) const = 0; // read the element in the node
	virtual void write(node, value_type) = 0; // write the element in the node
	virtual void ins_root(node) = 0; // insert the root
	virtual void ins_left(node) = 0; // insert an empty node in the left son
	virtual void ins_right(node) = 0; // insert an empty node in the right son
```

```
	void printSubTree(const node, ostream&) const; // useful for operator <<
	int height() const; // return the height of the tree
	bool is_leaf(const node) const; // true if the node is a leaf
	void ins_subtree(node&, const BST<T, N>&, node); // insert a subtree starting from the node
	int size(const node) const; // return the number of descendants of the node
	int height(const node) const; // return the height of the tree having root in the node
	node getNode(const value_type&) const; // return the node of the value
	value_type min(const node) const; // return the minimum in the subtree having root in the 
node

	void preorder(const node) const;
	void inorder(const node) const; // symmetric
	void postorder(const node) const;
	void breadth(const node) const;
};

template<class T, class N>
ostream& operator <<(ostream& out, const BST<T, N>& tree)
{
	if (!tree.empty())
		tree.printSubTree(tree.root(), out);
	else
		out << "Empty Tree" << endl;

	out << endl;

	return out;
}

template<class T, class N>
bool BST<T, N>::operator ==(const BST<T, N>& t) const
{
	std::stringstream tree1, tree2;

	tree1 << *this;
	tree2 << t;

	return (tree1.str() == tree2.str());
}

template<class T, class N>
bool BST<T, N>::operator !=(const BST<T, N>& tree2) const
{
	return (!(*this == tree2));
}

template<class T, class N>
void BST<T, N>::printSubTree(const node n, ostream& out) const
{
	out << "[" << this->read(n) << ", ";

	if (!this->left_empty(n))
		this->printSubTree(this->left(n), out);
	else
		out << "NIL";

	out << ", ";

	if (!this->right_empty(n))
		this->printSubTree(this->right(n), out);
	else
```

```
			out << "NIL";

		out << " ]";
}

template<class T, class N>
int BST<T, N>::size() const
{
	if (this->empty())
		return 0;

	return (this->size(this->root()) + 1);
}

template<class T, class N>
bool BST<T, N>::find(const value_type& val) const
{
	if (this->empty())
		return false;

	bool flag = false;
	bool skip = false;
	node curr = this->root();

	do
	{
		if (val == this->read(curr))
			flag = true;
		else if (val < this->read(curr))
		{
			if (!this->left_empty(curr))
				curr = this->left(curr);
			else
				skip = true;
		}
		else if (val > this->read(curr))
		{
			if (!this->right_empty(curr))
				curr = this->right(curr);
			else
				skip = true;
		}
	} while (flag == false && skip == false);

	return flag;
}

template<class T, class N>
int BST<T, N>::height() const
{
	if (this->empty())
		return 0;

	return (this->height(this->root()));
}

template<class T, class N>
typename BST<T, N>::value_type BST<T, N>::min() const
{
	if (this->empty())
		throw std::logic_error("BST (exception) - Unable to retrieve minimum (empty tree)");
```

```
	value_type min = this->read(this->root());
	node curr = this->root();

	while (!this->left_empty(curr))
	{
		curr = this->left(curr);
		min = this->read(curr);
	}

	return min;
}

template<class T, class N>
typename BST<T, N>::value_type BST<T, N>::max() const
{
	if (this->empty())
		throw std::logic_error("BST (exception) - Unable to retrieve maximum (empty tree)");

	value_type max = this->read(this->root());
	node curr = this->root();

	while (!this->right_empty(curr))
	{
		curr = this->right(curr);
		max = this->read(curr);
	}

	return max;
}

template<class T, class N>
void BST<T, N>::preorder() const
{
	this->preorder(this->root());
}

template<class T, class N>
void BST<T, N>::inorder() const
{
	this->inorder(this->root());
}

template<class T, class N>
void BST<T, N>::postorder() const
{
	this->postorder(this->root());
}

template<class T, class N>
void BST<T, N>::breadth() const
{
	this->breadth(this->root());
}

template<class T, class N>
bool BST<T, N>::is_leaf(const node n) const
{
	return (this->left_empty(n) && this->right_empty(n));
}
```

```
template<class T, class N>
void BST<T, N>::ins_subtree(node& n1, const BST<T, N>& t, node n2)
{
	if (t.empty() || !this->is_leaf(n1))
		throw std::logic_error("BST (exception) - Unable to insert subtree");

	this->write(n1, t.read(n2));

	if (!t.left_empty(n2))
	{
		this->ins_left(n1);
		node temp = this->left(n1);
		this->ins_subtree(temp, t, t.left(n2));
	}

	if (!t.right_empty(n2))
	{
		this->ins_right(n1);
		node temp = this->right(n1);
		this->ins_subtree(temp, t, t.right(n2));
	}
}

template<class T, class N>
int BST<T, N>::size(const node n) const
{
	int num = 0;
	node curr = n;
	std::deque<node> nodes;

	nodes.push_back(curr);

	while (!nodes.empty())
	{
		curr = nodes.front();
		num++;

		if (!this->left_empty(curr))
			nodes.push_back(this->left(curr));

		if (!this->right_empty(curr))
			nodes.push_back(this->right(curr));

		nodes.pop_front();
	}

	num--; // the starting node must not be counted

	return num;
}

template<class T, class N>
int BST<T, N>::height(const node n) const
{
	if (this->empty())
		return 0;

	if (this->is_leaf(n))
		return 1;
	else
	{
		int leftDepth = 0;
```

```
		int rightDepth = 0;

		// find each depth
		if (!this->left_empty(n))
			leftDepth = this->height(n->getLeft());
		if (!this->right_empty(n))
			rightDepth = this->height(n->getRight());

		// use the biggest one
		if (leftDepth > rightDepth)
			return (leftDepth + 1);
		else
			return (rightDepth + 1);
	}
}

template<class T, class N>
typename BST<T, N>::node BST<T, N>::getNode(const value_type& val) const
{
	if (this->empty() || !this->find(val))
		throw std::logic_error("BST (exception) - Unable to retrieve node");

	node curr = this->root();
	bool skip = false;

	do
	{
		if (val == this->read(curr))
			skip = true;
		else if (val > this->read(curr))
			curr = this->right(curr);
		else if (val < this->read(curr))
			curr = this->left(curr);
	} while (skip == false);

	return curr;
}

template<class T, class N>
typename BST<T, N>::value_type BST<T, N>::min(const node n) const
{
	if (this->empty())
		throw std::logic_error("BST (exception) - Unable to retrieve minimum (empty tree)");

	value_type min = this->read(n);
	node curr = n;

	while (!this->left_empty(curr))
	{
		curr = this->left(curr);
		min = this->read(curr);
	}

	return min;
}

template<class T, class N>
void BST<T, N>::preorder(const node n) const
{
	if (this->empty())
		throw std::logic_error("BST (exception) - Unable to perform pre-order visit (empty
tree)");
```

```
	cout << this->read(n);

	if (!this->left_empty(n))
		this->preorder(this->left(n));

	if (!this->right_empty(n))
		this->preorder(this->right(n));
}

template<class T, class N>
void BST<T, N>::inorder(const node n) const
{
	if (this->empty())
		throw std::logic_error("BST (exception) - Unable to perform in-order visit (empty
tree)");

	if (!this->left_empty(n))
		this->inorder(this->left(n));

	cout << this->read(n);

	if (!this->right_empty(n))
		this->inorder(this->right(n));
}

template<class T, class N>
void BST<T, N>::postorder(const node n) const
{
	if (this->empty())
		throw std::logic_error("BST (exception) - Unable to perform post-order visit (empty
tree)");

	if (!this->left_empty(n))
		this->postorder(this->left(n));

	if (!this->right_empty(n))
		this->postorder(this->right(n));

	cout << this->read(n);
}

template<class T, class N>
void BST<T, N>::breadth(const node n) const
{
	if (this->empty())
		throw std::logic_error("BST (exception) - Unable to perform breadth visit (empty
tree)");

	node temp;
	std::deque<node> nodes;

	nodes.push_back(n);

	while (!nodes.empty())
	{
		temp = nodes.front();
		cout << this->read(temp);
		nodes.pop_front();

		if (!this->left_empty(temp))
			nodes.push_back(this->left(temp));
```

```
            if (!this->right_empty(temp))
                nodes.push_back(this->right(temp));
    }
}

#endif /* _BST_H_ */
```

Alberi binari di ricerca – Realizzazione con puntatori

```
#ifndef _BSTP_H
#define _BSTP_H

#include "cellbst.h"
#include "BST.h"

// values are keys
// inserting 2 times the same value an exception will be thrown

template<class T>
class BST_pointer: public BST<T, Cell<T>*>
{
public:

	typedef typename BST<T, Cell<T>*>::value_type value_type;
	typedef typename BST<T, Cell<T>*>::node node;

	BST_pointer();
	BST_pointer(const value_type&);
	BST_pointer(const BST_pointer<T>&);
	~BST_pointer();

	BST_pointer<T>& operator =(const BST_pointer<T>&);

	bool empty() const; // true if the tree is empty
	void insert(value_type); // insert the element in the first available node (if it isn't
already present in the tree)
	void erase(value_type); // erase the node with that value (values are key)

private:
	node tree_root;

	void create(); // create the binary tree
	node root() const; // return the root node
	node parent(node) const; // return the father of the node
	node left(node) const; // return the left node
	node right(node) const; // return the right node
	bool left_empty(node) const; // true if there isn't a left son
	bool right_empty(node) const; // true if there isn't a right son
	void erase(node); // erase the subtree having root in the node
	value_type read(node) const; // read the element in the node
	void write(node, value_type); // write the element in the node
	void ins_root(node); // insert the root
	void ins_left(node); // insert an empty node in the left son
	void ins_right(node); // insert an empty node in the right son

	bool isLeftSon(node) const; // true if the node is the left son of his parent
	bool isRightSon(node) const; // true if the node is the right son of his parent
};


template<class T>
BST_pointer<T>::BST_pointer()
{
	this->create();
}
```

```
template<class T>
BST_pointer<T>::BST_pointer(const value_type& value)
{
	this->create();
	node newroot = new Cell<value_type>(value);
	this->ins_root(newroot);
}

template<class T>
BST_pointer<T>::BST_pointer(const BST_pointer<T>& tree)
{
	*this = tree;
}

template<class T>
BST_pointer<T>::~BST_pointer()
{
	this->erase(this->root());
}

template<class T>
BST_pointer<T>& BST_pointer<T>::operator =(const BST_pointer<T>& tree)
{
	if (&tree != this) // avoid auto-assignment assignment
	{
		if (!this->empty())
			this->erase(this->root());

		node newroot = new Cell<value_type>;
		this->ins_root(newroot);
		node temp = this->root();
		this->ins_subtree(temp, tree, tree.root());
	}

	return *this;
}

template<class T>
bool BST_pointer<T>::empty() const
{
	return (tree_root == 0);
}

template<class T>
void BST_pointer<T>::insert(value_type val)
{
	if (this->empty())
	{
		node newroot;
		newroot = new Cell<value_type>;
		newroot->setValue(val);
		this->ins_root(newroot);
	}
	else //!empty
	{
		if (this->find(val))
			throw std::logic_error("BST (exception) - Unable to insert element (already 
present)");

		node curr = this->root();
		bool skip = false;
```

```
		do
		{
			if (val > this->read(curr))
			{
				if (this->right_empty(curr))
				{
					this->ins_right(curr);
					this->write(this->right(curr), val);
					skip = true;
				}
				else //!right_empty
				{
					curr = this->right(curr);
				}
			}
			else //val < read (curr)
			{
				if (this->left_empty(curr))
				{
					this->ins_left(curr);
					this->write(this->left(curr), val);
					skip = true;
				}
				else
				{
					curr = this->left(curr);
				}
			}
		} while (skip == false);
	}
}

template<class T>
void BST_pointer<T>::erase(value_type val)
{
	node n = this->getNode(val);

	if (this->is_leaf(n))
	{
		if (n != tree_root)
		{
			if (this->parent(n)->getRight() == n)
				this->parent(n)->setRight(0);
			else
				this->parent(n)->setLeft(0);
			delete n;
			n = 0;
		}
		else
		{
			delete tree_root;
			tree_root = 0;
		}
	}
	else if (this->left_empty(n) && !this->right_empty(n)) //1 son (left)
	{
		if (n != tree_root)
		{
			node p = this->parent(n), r_child = this->right(n); // save the parent

			// if n is on the right
			if (p->getRight() == n)
```

```
                    p->setRight(r_child); // the new right child of the parent of n is the
right-child of n
                else
                    p->setLeft(r_child);

                r_child->setFather(p); // the parent of the right child of n is the parent of n
            }
            else
            {
                tree_root = this->right(n);
            }

            delete n;
            n = 0;
        }
        else if (!this->left_empty(n) && this->right_empty(n)) // 1 son (right)
        {
            if (n != tree_root)
            {
                node p = this->parent(n), l_child = this->left(n);

                if (p->getRight() == n)
                    p->setRight(l_child);
                else
                    p->setLeft(l_child);

                l_child->setFather(p);

            }
            else
            {
                tree_root = this->left(n);
            }

            delete n;
            n = 0;
        }
        else //2 sons
        {
            value_type minimum = this->min(right(n));
            node min = this->getNode(minimum);

            // it is not a leaf node, so we have to fix correctly this case
            if (!this->is_leaf(min) && this->parent(min)->getRight() == min)
                this->parent(min)->setRight(this->right(min));
            else
            {
                if (this->parent(min)->getRight() == min)
                    this->parent(min)->setRight(0);
                else
                    this->parent(min)->setLeft(0);
            }

            this->write(n, this->read(min)); // change the label of the n node
            delete min;
            min = 0;
        }
}
```

```
template<class T>
void BST_pointer<T>::create()
{
	tree_root = 0;
}

template<class T>
typename BST_pointer<T>::node BST_pointer<T>::root() const
{
	return tree_root;
}

template<class T>
typename BST_pointer<T>::node BST_pointer<T>::parent(node n) const
{
	if (n == this->root())
		throw std::logic_error("BST_pointer (exception) - Unable to get parent (node is
root)");

	return n->getFather();
}

template<class T>
typename BST_pointer<T>::node BST_pointer<T>::left(node n) const
{
	if (this->left_empty(n))
		throw std::logic_error("BST_pointer (exception) - Unable to get left son");

	return n->getLeft();
}

template<class T>
typename BST_pointer<T>::node BST_pointer<T>::right(node n) const
{
	if (this->right_empty(n))
		throw std::logic_error("BST_pointer (exception) - Unable to get right son");

	return n->getRight();
}

template<class T>
bool BST_pointer<T>::left_empty(node n) const
{
	return (n->getLeft() == 0);
}

template<class T>
bool BST_pointer<T>::right_empty(node n) const
{
	return (n->getRight() == 0);
}

template<class T>
void BST_pointer<T>::erase(node n)
{
	if (!this->empty())
	{
		if (!is_leaf(n))
		{
			if (!this->left_empty(n))
				erase(n->getLeft());
```

```
                        if (!this->right_empty(n))
                                erase(n->getRight());
                }

                if (n != tree root)
                {
                        if (this->isLeftSon(n))
                                (this->parent(n))->setLeft(0);
                        else if (this->isRightSon(n))
                                (this->parent(n))->setRight(0);
                        delete (n);
                        n = 0;
                }
                else
                {
                        delete tree root;
                        tree root = 0;
                }
        }
}

template<class T>
typename BST_pointer<T>::value_type BST_pointer<T>::read(node n) const
{
        if (n == 0)
                throw std::logic_error("BST_pointer (exception) - Unable to read value (invalid
node)");

        return n->getValue();
}

template<class T>
void BST_pointer<T>::write(node n, value_type value)
{
        if (n == 0)
                throw std::logic_error("BST_pointer (exception) - Unable to write value (invalid
node)");

        n->setValue(value);
}

template<class T>
void BST_pointer<T>::ins_root(node n = 0)
{
        if (n == 0)
                tree root = new Cell<value_type>;
        else
        {
                tree root = new Cell<value_type>;
                tree root->setValue(n->getValue());
        }
}

template<class T>
void BST_pointer<T>::ins_left(node n)
{
        if (!this->left_empty(n))
                throw std::logic_error("BST_pointer (exception) - Unable to insert left node (already
present)");

        node son;
        son = new Cell<value_type>;
```

```
	son->setFather(n);
	n->setLeft(son);
}

template<class T>
void BST_pointer<T>::ins_right(node n)
{
	if (!this->right_empty(n))
		throw std::logic_error("BST_pointer (exception) - Unable to insert right node (already present)");

	node son;
	son = new Cell<value_type>;
	son->setFather(n);
	n->setRight(son);
}

template<class T>
bool BST_pointer<T>::isLeftSon(node n) const
{
	if (n == 0 || n == this->root())
		throw std::logic_error("BST_pointer (exception) - Unable to establish if node is left son");

	return (n == (n->getFather())->getLeft());
}

template<class T>
bool BST_pointer<T>::isRightSon(node n) const
{
	if (n == 0 || n == this->root())
		throw std::logic_error("BST_pointer (exception) - Unable to establish if node is right son");

	return (n == (n->getFather())->getRight());
}

#endif // _BSTP_H
```

Alberi binari di ricerca – Realizzazione con puntatori
File: “cellbst.h”

```
#ifndef _CELLBST_H
#define _CELLBST_H

template<class T>
class Cell
{
public:

	typedef T value_type;
	typedef Cell* node;

	Cell();
	Cell(const value_type&);
	~Cell();

	Cell<T>& operator =(const Cell<T>&);
	bool operator ==(const Cell<T>&) const;
	bool operator !=(const Cell<T>&) const;

	void setValue(const value_type);
	value_type getValue() const;
	void setFather(node);
	void setRight(node);
	void setLeft(node);
	node getFather() const;
	node getRight() const;
	node getLeft() const;

private:
	value_type value;
	node father;
	node right;
	node left;
};

template<class T>
Cell<T>::Cell()
{
	value = value_type();
	father = 0;
	right = 0;
	left = 0;
}

template<class T>
Cell<T>::Cell(const value_type& val)
{
	value = val;
	father = 0;
	right = 0;
	left = 0;
}

template<class T>
Cell<T>::~Cell()
{
	value = value_type();
```

```
        father = 0;
        right = 0;
        left = 0;
}

template<class T>
Cell<T>& Cell<T>::operator =(const Cell<T>& c2)
{
        if (&c2 != this) // avoid auto-assignment
        {
                this->setValue(c2.getValue());
                this->setFather(c2.getFather());
                this->setLeft(c2.getLeft());
                this->setRight(c2.getRight());
        }
        return *this;
}

template<class T>
bool Cell<T>::operator ==(const Cell<T>& c2) const
{
        if (this->value != c2.value)
                return false;
        if (this->father != c2.father)
                return false;
        if (this->right != c2.right)
                return false;
        if (this->left != c2.left)
                return false;

        return true;
}

template<class T>
bool Cell<T>::operator !=(const Cell<T>& c2) const
{
        return (!(c2 == *this));
}

template<class T>
void Cell<T>::setValue(const value_type element)
{
        value = element;
}

template<class T>
typename Cell<T>::value_type Cell<T>::getValue() const
{
        return value;
}

template<class T>
void Cell<T>::setFather(node fatherp)
{
        father = fatherp;
}

template<class T>
void Cell<T>::setRight(node rightp)
{
        right = rightp;
}
```

```
template<class T>
void Cell<T>::setLeft(node leftp)
{
	left = leftp;
}

template<class T>
typename Cell<T>::node Cell<T>::getFather() const
{
	return father;
}

template<class T>
typename Cell<T>::node Cell<T>::getRight() const
{
	return right;
}

template<class T>
typename Cell<T>::node Cell<T>::getLeft() const
{
	return left;
}

#endif // _CELLBST_H
```

Alberi binari di ricerca – Tester

```
#include "BST_pointer.h"
#include <iostream>

int main()
{
	BST_pointer<int> t1;

	t1.insert(20);
	t1.insert(15);
	t1.insert(30);
	t1.insert(12);
	t1.insert(18);
	t1.insert(17);
	t1.insert(28);
	t1.insert(32);

	cout << t1 << endl;

	t1.erase(17);

	cout << t1 << endl;

	t1.erase(15);

	cout << t1 << endl;

}
```

Alberi binari di ricerca – Output Tester

```
[20, [15, [12, NIL, NIL ], [18, [17, NIL, NIL ], NIL ] ], [30, [28, NIL, NIL ], [32, NIL,
NIL ] ] ]

[20, [15, [12, NIL, NIL ], [18, NIL, NIL ] ], [30, [28, NIL, NIL ], [32, NIL, NIL ] ] ]

[20, [18, [12, NIL, NIL ], NIL ], [30, [28, NIL, NIL ], [32, NIL, NIL ] ] ]
```

Alberi n-ari – Classe astratta

```
#ifndef _NARYTREE_H_
#define _NARYTREE_H_

#include <iostream>
#include <deque>
#include <sstream>
#include <stdexcept>
#include <vector>
#include <numeric>

using std::cout;
using std::ostream;
using std::endl;
using std::vector;
using std::accumulate;

template<class T, class N>
class NaryTree
{
public:
	typedef T value_type;
	typedef N node;

	virtual ~NaryTree()
	{
	}

	virtual void create() = 0; // create the binary tree
	virtual bool empty() const = 0; // true if the three is empty
	virtual void ins_root(node) = 0; // insert the root
	virtual node root() const = 0; // return the root node
	virtual node parent(const node&) const = 0; // return the father of the node
	virtual bool is_leaf(const node&) const = 0; // true if the node is a leaf
	virtual node firstChild(const node&) const = 0; // return the first child of the node
	virtual bool lastSibling(const node&) const = 0; // true if the node doesn't have a right 
sibling
	virtual node nextSibling(const node&) const = 0; // return the right sibling of the node
	virtual void insFirstSubTree(node, const NaryTree<value_type, node>&) = 0; // add the tree 
like the first child of the node
	virtual void insSubTree(node, const NaryTree<value_type, node>&) = 0; // add the tree like 
sibling of the node
	virtual void erase(node) = 0; // erase the subtree having root = node
	virtual value_type read(const node&) const = 0; // read the element in the node
	virtual void write(node, const value_type&) = 0; // write the element in the node

	bool operator ==(const NaryTree<value_type, node>&) const;
	bool operator !=(const NaryTree<value_type, node>&) const;

	void printSubTree(const node, ostream&) const; // useful for operator <<
	int numChild(const node) const; // return the number of children of the node
	void swapNodes(node, node); // swap the two nodes
	int size() const; // return the numbers of nodes of the tree
	int size(const node) const; // return the number of descendants of the node
	bool find(const value_type&) const; // true if the element is present in the tree
	value_type min() const; // return the minimum element of the tree
	value_type max() const; // return the maximum element of the tree
	int gradeN() const; // return the grade of the tree
	int height() const; // return the height of the tree
```

```
	void eraseLeaves(const value_type&); // erase the leaves having the same value of the function
	vector<node> evenRoots() const; // return an array of nodes. Every node have children whose sum is an even value
	vector<node> oddRoots() const; // return an array of nodes. Every node have children whose sum is an odd value
	vector<value_type> elementsArray() const; // return a vector with the elements of the tree
	void elementsArray(const node&, vector<value_type>&) const; // auxiliary for elementsArray
	vector<node> nodesArray() const; // return a vector with the nodes of the tree
	void nodesArray(const node&, vector<node>&) const; // auxiliary for nodesArray
	int numberLeaves() const; // return the number of leaves in the tree
	bool findOnPath(node, const value_type&) const; // true if the element is present in the path from root to the node
	bool findOnChildren(node, const value_type&) const; // true if the element is present in one of the children
	value_type pathSum(node) const; // return the sum of the element on the path from the root to the node
	vector<value_type> pathElements(node) const; // return a vector of the elements in the path from the root to the node

	void preorder() const;
	void inorder() const; // symmetric
	void postorder() const;
	void breadth() const;
	void preorder(const node&) const;
	void postorder(const node&) const;
	void inorder(const node&) const;
	void breadth(const node&) const;

private:
	bool checkEquality(const node&, const NaryTree<T, N>&, const node&) const;
	int height(const node&) const; // return the height of the tree starting from the node
	void eraseLeaves(const value_type&, node&); // erase the leaves having a certain value from the subtree having root in the node
};

template<class T, class N>
ostream& operator <<(ostream& out, const NaryTree<T, N>& tree)
{
	if (!tree.empty())
		tree.printSubTree(tree.root(), out);
	else
		out << "Empty Tree";

	out << endl;

	return out;
}

template<class T, class N>
bool NaryTree<T, N>::operator ==(const NaryTree<T, N>& t2) const
{
	if ((t2.empty() + this->empty()) == 2) //if ( !this->empty() && !t2.empty() )
		return true;
	if (this->size() != t2.size())
		return false;

	return checkEquality(this->root(), t2, t2.root());
}
```

```
template<class T, class N>
bool NaryTree<T, N>::operator !=(const NaryTree<T, N>& tree2) const
{
	return (!(*this == tree2));
}

template<class T, class N>
void NaryTree<T, N>::printSubTree(const node n, ostream& out) const
{
	out << "[ " << this->read(n);

	if (!this->is_leaf(n))
	{
		node temp;
		out << ": ";
		for (temp = this->firstChild(n); !this->lastSibling(temp); temp = this-
>nextSibling(temp))
		{
			this->printSubTree(temp, out);
			out << ", ";
		}

		this->printSubTree(temp, out);

	}

	out << " ]";
}

template<class T, class N>
int NaryTree<T, N>::numChild(const node n) const
{
	if (this->empty())
		throw std::logic_error("BinTree (exception) - Unable to get number of children (empty
tree)");

	if (this->is_leaf(n))
		return 0;

	node temp = this->firstChild(n);
	int counter = 0;

	while (!this->lastSibling(temp))
	{
		counter++;
		temp = this->nextSibling(temp);
	}

	counter++; //the last sibling wasn't counted

	return counter;
}

template<class T, class N>
void NaryTree<T, N>::swapNodes(node n1, node n2)
{
	if (this->read(n1) != this->read(n2))
	{
		value_type temp = this->read(n1);

		this->write(n1, this->read(n2));
		this->write(n2, temp);
```

```
	}
}

template<class T, class N>
int NaryTree<T, N>::size() const
{
	if (this->empty())
		return 0;

	return (this->size(this->root()) + 1);
}

template<class T, class N>
int NaryTree<T, N>::size(const node n) const
{
	int num = 0;
	node curr = n;
	std::deque<node> nodes;

	nodes.push_back(curr);

	while (!nodes.empty())
	{
		curr = nodes.front();
		num = num + this->numChild(curr);

		if (!this->is_leaf(curr))
		{
			node temp = this->firstChild(curr);
			for (int i = 0; i < this->numChild(curr) - 1; i++)
			{
				nodes.push_back(temp);
				temp = this->nextSibling(temp);
			}
			nodes.push_back(temp);
		}

		nodes.pop_front();
	}

	return num;
}

template<class T, class N>
bool NaryTree<T, N>::find(const value_type& val) const
{
	vector<value_type> elements = this->elementsArray();

	for (typename vector<value_type>::iterator it = elements.begin(); it != elements.end(); it++)
		if (*it == val)
			return true;

	return false;
}

template<class T, class N>
typename NaryTree<T, N>::value_type NaryTree<T, N>::min() const
{
	vector<value_type> elements = this->elementsArray();

	if (elements.empty())
```

```
		throw std::logic_error("BinTree (exception) - Unable to retrieve minimum (empty
tree)");

	value_type min = elements.front();

	for (typename vector<value_type>::iterator it = elements.begin(); it != elements.end(); it++)
		if (*it < min)
			min = *it;

	return min;
}

template<class T, class N>
typename NaryTree<T, N>::value_type NaryTree<T, N>::max() const
{
	vector<value_type> elements = this->elementsArray();

	if (elements.empty())
		throw std::logic_error("BinTree (exception) - Unable to retrieve maximum (empty
tree)");

	value_type max = elements.front();

	for (typename vector<value_type>::iterator it = elements.begin(); it != elements.end(); it++)
		if (*it > max)
			max = *it;

	return max;
}

template<class T, class N>
int NaryTree<T, N>::gradeN() const
{
	vector<node> nodes = nodesArray();

	if (nodes.empty())
		throw std::logic_error("BinTree (exception) - Unable to retrieve grade (empty tree)");

	int grade = this->numChild(nodes.front());

	for (typename vector<node>::iterator it = nodes.begin(); it != nodes.end(); it++)
		if (this->numChild(*it) > grade)
			grade = this->numChild(*it);

	return grade;
}

template<class T, class N>
int NaryTree<T, N>::height() const
{
	return (this->height(this->root()));
}

template<class T, class N>
void NaryTree<T, N>::eraseLeaves(const value_type& val)
{
	if (this->empty())
		throw std::logic_error("BinTree (exception) - Unable to erase leaves (empty tree)");

	node n = this->root();
	this->eraseLeaves(val, n);
}
```

```
template<class T, class N>
vector<N> NaryTree<T, N>::evenRoots() const
{
	vector<node> evenRoots;
	vector<node> nodes = this->nodesArray();
	for (typename vector<node>::iterator it = nodes.begin(); it != nodes.end(); it++)
	{
		vector<T> temp;
		this->elementsArray(*it, temp);
		if (((accumulate(temp.begin(), temp.end(), 0)) % 2) == 0)
			evenRoots.push_back(*it);
	}

	return evenRoots;
}

template<class T, class N>
vector<N> NaryTree<T, N>::oddRoots() const
{
	vector<node> oddRoots;
	vector<node> nodes = this->nodesArray();
	for (typename vector<node>::iterator it = nodes.begin(); it != nodes.end(); it++)
	{
		vector<T> temp;
		this->elementsArray(*it, temp);
		if (((accumulate(temp.begin(), temp.end(), 0)) % 2) != 0)
			oddRoots.push_back(*it);
	}

	return oddRoots;
}

template<class T, class N>
vector<T> NaryTree<T, N>::elementsArray() const
{
	if (this->empty())
		throw std::logic_error("BinTree (exception) - Unable to return vector (empty tree)");

	vector<value_type> elements;

	node temp = this->root();
	elements.push_back(this->read(temp));

	if (!this->is_leaf(temp))
	{
		node curr = this->firstChild(temp);

		for (int i = 0; i < this->numChild(temp) - 1; i++)
		{
			this->elementsArray(curr, elements);
			curr = this->nextSibling(curr);
		}

		this->elementsArray(curr, elements);
	}

	return elements;
}
```

```
template<class T, class N>
void NaryTree<T, N>::elementsArray(const node& n, vector<value_type>& elements) const
{
	elements.push_back(this->read(n));

	if (!this->is_leaf(n))
	{
		node curr = this->firstChild(n);

		for (int i = 0; i < this->numChild(n) - 1; i++)
		{
			this->elementsArray(curr, elements);
			curr = this->nextSibling(curr);
		}

		this->elementsArray(curr, elements);
	}
}

template<class T, class N>
vector<N> NaryTree<T, N>::nodesArray() const
{
	if (this->empty())
		throw std::logic_error("BinTree (exception) - Unable to return vector (empty tree)");

	vector<node> nodes;

	node temp = this->root();
	nodes.push_back(temp);

	if (!this->is_leaf(temp))
	{
		node curr = this->firstChild(temp);

		for (int i = 0; i < this->numChild(temp) - 1; i++)
		{
			this->nodesArray(curr, nodes);
			curr = this->nextSibling(curr);
		}

		this->nodesArray(curr, nodes);
	}

	return nodes;
}

template<class T, class N>
void NaryTree<T, N>::nodesArray(const node& n, vector<node>& nodes) const
{
	nodes.push_back(n);
	if (!this->is_leaf(n))
	{
		node curr = this->firstChild(n);

		for (int i = 0; i < this->numChild(n) - 1; i++)
		{
			this->nodesArray(curr, nodes);
			curr = this->nextSibling(curr);
		}
		this->nodesArray(curr, nodes);
	}
}
```

```
template<class T, class N>
int NaryTree<T, N>::numberLeaves() const
{
	int counter = 0;
	vector<node> nodes = this->nodesArray();
	for (typename vector<node>::iterator it = nodes.begin(); it != nodes.end(); it++)
		if (this->is_leaf(*it))
			counter++;
	return counter;
}

template<class T, class N>
bool NaryTree<T, N>::findOnPath(node currNode, const value_type& val) const
{
	if (currNode->getValue() == val)
		return true;

	while (currNode != this->root())
	{
		currNode = this->parent(currNode);
		if (currNode->getValue() == val)
			return true;
	}
	return false;
}

template<class T, class N>
bool NaryTree<T, N>::findOnChildren(node currNode, const value_type& val) const
{
	if (this->is_leaf(currNode))
		return false;

	currNode = this->firstChild(currNode);

	if (currNode->getValue() == val)
		return true;

	while (!this->lastSibling(currNode))
	{
		currNode = this->nextSibling(currNode);
		if (currNode->getValue() == val)
			return true;
	}

	return false;
}

template<class T, class N>
T NaryTree<T, N>::pathSum(node currNode) const
{
	value_type counter = value_type();

	counter += currNode->getValue();

	while (currNode != this->root())
	{
		currNode = this->parent(currNode);
		counter += currNode->getValue();
	}

	return counter;
}
```

```
template<class T, class N>
vector<typename NaryTree<T, N>::value_type> NaryTree<T, N>::pathElements(node currNode) const
{
	vector<value_type> elements;

	elements.push_back(currNode->getValue());
	while (currNode != this->root())
	{
		currNode = this->parent(currNode);
		elements.push_back(currNode->getValue());
	}

	return elements;
}

template<class T, class N>
void NaryTree<T, N>::preorder() const
{
	if (this->empty())
		throw std::logic_error("NaryTree (exception) - Unable to perform pre-order visit
(empty tree)");

	this->preorder(this->root());
}

template<class T, class N>
void NaryTree<T, N>::inorder() const
{
	if (this->empty())
		throw std::logic_error("NaryTree (exception) - Unable to perform in-order visit (empty
tree)");

	this->inorder(this->root());
}

template<class T, class N>
void NaryTree<T, N>::postorder() const
{
	if (this->empty())
		throw std::logic_error("NaryTree (exception) - Unable to perform post-order visit
(empty tree)");

	this->postorder(this->root());
}

template<class T, class N>
void NaryTree<T, N>::breadth() const
{
	if (this->empty())
		throw std::logic_error("NaryTree (exception) - Unable to perform breadth visit (empty
tree)");

	this->breadth(this->root());
}

template<class T, class N>
void NaryTree<T, N>::preorder(const node& n) const
{
	if (this->empty())
		throw std::logic_error("BinTree (exception) - Unable to perform pre-order visit (empty
tree)");
```

```
	cout << this->read(n) << " ";

	if (!this->is_leaf(n))
	{
		node temp = this->firstChild(n);

		for (int i = 0; i < this->numChild(n) - 1; i++)
		{
			this->preorder(temp);
			temp = this->nextSibling(temp);
		}
		this->preorder(temp);
	}
}

template<class T, class N>
void NaryTree<T, N>::postorder(const node& n) const
{
	if (this->is_leaf(n))
		cout << this->read(n) << " ";
	else
	{
		node temp = this->firstChild(n);
		while (!this->lastSibling(temp))
		{
			this->postorder(temp);
			temp = this->nextSibling(temp);
		}
		this->postorder(temp);
		cout << this->read(n) << " ";
	}
}

template<class T, class N>
void NaryTree<T, N>::inorder(const node& n) const
{
	if (this->is_leaf(n))
		cout << this->read(n) << " ";
	else
	{
		node temp = this->firstChild(n);
		this->inorder(temp);
		cout << this->read(n) << " ";
		while (!this->lastSibling(temp))
		{
			temp = this->nextSibling(temp);
			this->inorder(temp);
		}
	}
}

template<class T, class N>
void NaryTree<T, N>::breadth(const node& n) const
{
	node temp = 0;
	std::deque<node> nodes;

	nodes.push_back(n);

	while (!nodes.empty())
	{
		temp = nodes.front();
```

```
			cout << this->read(temp) << " ";
			nodes.pop_front();

			if (!this->is_leaf(temp))
			{
				node curr = this->firstChild(temp);
				while (!this->lastSibling(curr))
				{
					nodes.push_back(curr);
					curr = this->nextSibling(curr);
				}
				nodes.push_back(curr);
			}
		}
}

template<class T, class N>
bool NaryTree<T, N>::checkEquality(const N& n1, const NaryTree<T, N>& t2, const N& n2) const
{
		if (this->read(n1) != t2.read(n2))
			return false;

		if (this->numChild(n1) != t2.numChild(n2))
			return false;

		if (!this->is_leaf(n1) && !t2.is_leaf(n2))
		{
			node temp1 = this->firstChild(n1);
			node temp2 = t2.firstChild(n2);

			while (!this->lastSibling(temp1))
			{
				if (this->checkEquality(temp1, t2, temp2))
				{
					temp1 = this->nextSibling(temp1);
					temp2 = t2.nextSibling(temp2);
				}
				else
					return false;
			}

			return this->checkEquality(temp1, t2, temp2);
		}

		return true;
}

template<class T, class N>
int NaryTree<T, N>::height(const node& n) const
{
		if (this->empty())
			throw std::logic_error("NaryTree (exception) - Unable to retrieve height of the tree
(empty tree)");

		if (this->is_leaf(n))
			return 1;
		else
		{
			vector<int> heights;
			node curr = this->firstChild(n);
			while (!this->lastSibling(curr))
			{
```

```
                        int currHeight = 0;
                        currHeight = this->height(curr);
                        heights.push_back(currHeight);
                        curr = this->nextSibling(curr);
                }

                int currHeight = 0;
                currHeight = this->height(curr);
                heights.push_back(currHeight);

                int maxHeight = heights.front();

                for (vector<int>::iterator it = heights.begin(); it != heights.end(); it++)
                        if (*it > maxHeight)
                                maxHeight = *it;

                return maxHeight + 1;
        }
}

template<class T, class N>
void NaryTree<T, N>::eraseLeaves(const value_type& val, node& n)
{
        if (!this->is_leaf(n))
        {
                node curr = this->firstChild(n);
                while (!this->lastSibling(curr))
                {
                        this->eraseLeaves(val, curr);
                        curr = this->nextSibling(curr);
                }
                this->eraseLeaves(val, curr);
        }
        else if (this->read(n) == val)
                this->erase(n);
}

#endif /* _NARYTREE_H_ */
```

Alberi n-ari – Realizzazione con puntatori

```
#ifndef _NARYTREEP_H
#define _NARYTREEP_H

#include "cellnt.h"
#include "N-aryTree.h"
#include <vector>

template<class T>
class NaryTree_pointer: public NaryTree<T, Cell<T>*>
{
public:
	typedef typename NaryTree<T, Cell<T>*>::value_type value_type;
	typedef typename NaryTree<T, Cell<T>*>::node node;

	NaryTree_pointer();
	NaryTree_pointer(const value_type&);
	NaryTree_pointer(const NaryTree_pointer<T>&);
	~NaryTree_pointer();

	NaryTree_pointer<T>& operator =(const NaryTree_pointer<T>&);

	bool empty() const; // true if the three is empty
	void ins_root(node); // insert the root
	node root() const; // return the root node
	node parent(const node&) const; // return the father of the node
	bool is_leaf(const node&) const; // true if the node is a leaf
	node firstChild(const node&) const; // return the first child of the node
	bool lastSibling(const node&) const; // true if the node doesn't have a right sibling
	node nextSibling(const node&) const; // return the right sibling of the node
	void insFirstSubTree(node, const NaryTree<value_type, node>&); // add the tree like the first
child of the node (node!=0)
	void insSubTree(node, const NaryTree<value_type, node>&); // add the tree like sibling of the
node (node!=0)
	void erase(node); // erase the subtree having root = node
	value_type read(const node&) const; // read the element in the node
	void write(node, const value_type&); // write the element in the node

	node lastChild(const node) const; // return the last child of the node
	bool firstSibling(const node) const; // true if the node is the first child
	node prevSibling(const node) const; // return the left sibling of the node
	void insFirstChild(node); // add a child to the node (leftmost)
	void insLastChild(node); // add a child to the node (rightmost)
	void insRightSibling(node); // add a right sibling to the node
	void build(NaryTree_pointer<T>, NaryTree_pointer<T>); // build a new tree: implicit tree must
be empty, other two !empty
	void erase(); // erase the all tree

private:
	node tree_root;

	void create(); // create the binary tree
	void copyTree(node, const NaryTree_pointer<T>&, node); // insert, starting from the node
(created and empty), the tree
	void copyAbstractTree(node&, const NaryTree<value_type, node>&, const node&); // insert,
starting from the node (created and empty), the tree
};
```

```
template<class T>
NaryTree_pointer<T>::NaryTree_pointer()
{
	this->create();
}

template<class T>
NaryTree_pointer<T>::NaryTree_pointer(const value_type& value)
{
	this->create();
	node newroot = new Cell<value_type>(value);
	this->ins_root(newroot);
}

template<class T>
NaryTree_pointer<T>::NaryTree_pointer(const NaryTree_pointer<T>& tree)
{
	this->create();
	*this = tree;
}

template<class T>
NaryTree_pointer<T>::~NaryTree_pointer()
{
	this->erase(this->root());
}

template<class T>
NaryTree_pointer<T>& NaryTree_pointer<T>::operator =(const NaryTree_pointer<T>& tree)
{
	if (&tree != this) // avoid auto-assignment
	{
		if (!this->empty())
			this->erase(this->root());

		if (tree.root() != 0)
		{
			node newroot = new Cell<value_type>;
			this->ins_root(newroot);
			this->copyTree(this->root(), tree, tree.root());
		}
		else
			tree_root = 0;
	}
	return *this;
}

template<class T>
void NaryTree_pointer<T>::copyTree(node n1, const NaryTree_pointer<T>& t2, node n2)
{
	this->write(n1, t2.read(n2));

	if (!t2.is_leaf(n2))
	{
		node temp;
		for (temp = t2.firstChild(n2); !t2.lastSibling(temp); temp = t2.nextSibling(temp))
		{
			this->insLastChild(n1);
			this->copyTree(this->lastChild(n1), t2, temp);
		}

		//insert the last sibling
```

```
            this->insLastChild(n1);
            this->copyTree(lastChild(n1), t2, temp);
        }
}

template<class T>
void NaryTree_pointer<T>::create()
{
        tree_root = 0;
}

template<class T>
bool NaryTree_pointer<T>::empty() const
{
        return (this->root() == 0);
}

template<class T>
void NaryTree_pointer<T>::ins_root(node n)
{
        if (n == 0)
                tree_root = new Cell<value_type>;
        else
                tree_root = new Cell<value_type>(n->getValue());
}

template<class T>
typename NaryTree_pointer<T>::node NaryTree_pointer<T>::root() const
{
        return tree_root;
}

template<class T>
typename NaryTree_pointer<T>::node NaryTree_pointer<T>::parent(const node& n) const
{
        if (n == this->root())
                throw std::logic_error("NaryTree_pointer (exception) - Unable to retrieve parent (node
is root)");

        return n->getFather();
}

template<class T>
bool NaryTree_pointer<T>::is_leaf(const node& n) const
{
        if (n == 0)
                throw std::logic_error("NaryTree_pointer (exception)");

        return (n->getChild() == 0);
}

template<class T>
typename NaryTree_pointer<T>::node NaryTree_pointer<T>::firstChild(const node& n) const
{
        if (this->is_leaf(n))
                throw std::logic_error("NaryTree_pointer (exception) - Unable to retrieve first child
(node is leaf)");

        return (n->getChild());
}
```

```
template<class T>
bool NaryTree_pointer<T>::lastSibling(const node& n) const
{
	if (this->is_leaf(this->parent(n)))
		throw std::logic_error("NaryTree_pointer (exception) - Unable to check if node is last
sibling");

	return (n == this->lastChild(this->parent(n)) ? true : false);
}

template<class T>
typename NaryTree_pointer<T>::node NaryTree_pointer<T>::nextSibling(const node& n) const
{
	if (this->lastSibling(n))
		throw std::logic_error("NaryTree_pointer (exception) - Unable to get next sibling
(node is last sibling)");

	return n->getRight();
}


template<class T>
void NaryTree_pointer<T>::insFirstSubTree(node n, const NaryTree<value_type, node>& t2)
{
	if (n == 0 || t2.empty())
		throw std::logic_error("NaryTree_pointer (exception) - Unable to insert subtree");

	this->insFirstChild(n);
	node temp = this->firstChild(n);
	node root = t2.root();
	this->copyAbstractTree(temp, t2, root);
}

template<class T>
void NaryTree_pointer<T>::insSubTree(node n, const NaryTree<value_type, node>& t2)
{
	if (n == 0 || t2.empty() || n == this->root())
		throw std::logic_error("NaryTree_pointer (exception) - Unable to insert subtree");

	this->insRightSibling(n);
	node temp = this->nextSibling(n);
	node root = t2.root();
	this->copyAbstractTree(temp, t2, root);
}


template<class T>
void NaryTree_pointer<T>::erase(node n)
{
	if (!this->empty())
	{
		if (!this->is_leaf(n))
		{
			node curr = this->lastChild(n);
			while (curr != this->firstChild(n))
			{
				node temp2 = curr;
				curr = this->prevSibling(curr);
				this->erase(temp2);
			}
			this->erase(curr);
		}
```

```
            if (n != tree_root)
            {
                  if (this->size(this->parent(n)) == 1) // last child
                        (this->parent(n))->setChild(0);

                  if (n->getLeft() != 0)
                        (n->getLeft())->setRight(n->getRight());

                  if (n->getRight() != 0)
                        (n->getRight())->setLeft(n->getLeft());

                  delete n;
                  n = 0;
            }
            else
            {
                  delete tree_root;
                  tree_root = 0;
            }
      }
}

template<class T>
typename NaryTree<T, Cell<T>*>::value_type NaryTree_pointer<T>::read(const node& n) const
{
      if (n == 0)
            throw std::logic_error("NaryTree_pointer (exception) - Unable to read label (invalid
node)");

      return n->getValue();
}

template<class T>
typename NaryTree_pointer<T>::node NaryTree_pointer<T>::lastChild(const node n) const
{
      node temp = firstChild(n);
      while (temp->getRight() != 0)
            temp = temp->getRight();

      return temp;
}

template<class T>
void NaryTree_pointer<T>::write(node n, const value_type& value)
{
      if (n == 0)
            throw std::logic_error("NaryTree_pointer (exception) - Unable to write label (invalid
node)");

      n->setValue(value);
}

template<class T>
bool NaryTree_pointer<T>::firstSibling(const node n) const
{
      return (n == (this->parent(n))->getChild() ? true : false);
}

template<class T>
typename NaryTree_pointer<T>::node NaryTree_pointer<T>::prevSibling(const node n) const
{
      if (this->firstSibling(n))
```

```
		throw std::logic_error("NaryTree_pointer (exception) - Unable to retrieve previous 
sibling (node is first sibling)");

	return n->getLeft();
}

template<class T>
void NaryTree_pointer<T>::insFirstChild(node n)
{
	if (n == 0)
		throw std::logic_error("NaryTree_pointer (exception) - Unable to insert child (invalid 
node)");

	node son = new Cell<value_type>;

	if (!this->is_leaf(n))
	{
		(this->firstChild(n))->setLeft(son);
		son->setRight(this->firstChild(n));
	}

	son->setFather(n);
	n->setChild(son);
}

template<class T>
void NaryTree_pointer<T>::insLastChild(node n)
{
	if (n == 0)
		throw std::logic_error("NaryTree_pointer (exception) - Unable to insert child (invalid 
node)");

	node son = new Cell<value_type>;

	if (!this->is_leaf(n))
	{
		node temp = this->lastChild(n);
		(this->lastChild(n))->setRight(son);
		son->setLeft(temp);
	}

	if (this->is_leaf(n))
		n->setChild(son);

	son->setFather(n);
}

template<class T>
void NaryTree_pointer<T>::insRightSibling(node n)
{
	if (n == 0 || n == root())
		throw std::logic_error("NaryTree_pointer (exception) - Unable to insert child (invalid 
node)");

	if (this->lastSibling(n))
		this->insLastChild(this->parent(n));
	else
	{
		node brother = new Cell<value_type>;

		brother->setFather(parent(n));
		brother->setLeft(n);
```

```
            brother->setRight(nextSibling(n));

            (nextSibling(n))->setLeft(brother);

            n->setRight(brother);
    }
}

template<class T>
void NaryTree_pointer<T>::build(NaryTree_pointer<T> t1, NaryTree_pointer<T> t2)
{
    if (!this->empty() || t1.empty() || t2.empty())
            throw std::logic_error("NaryTree_pointer (exception) - Unable to build tree");

    node newroot = new Cell<value_type>(value_type());
    this->ins_root(newroot);
    node temp = this->root();
    this->insFirstSubTree(temp, t1);
    temp = this->firstChild(root());
    this->insSubTree(temp, t2);
}

template<class T>
void NaryTree_pointer<T>::copyAbstractTree(node& n1, const NaryTree<value_type, node>& t2, const
node& n2)
{
    this->write(n1, t2.read(n2));

    if (!t2.is_leaf(n2))
    {
            node temp;
            for (temp = t2.firstChild(n2); !t2.lastSibling(temp); temp = t2.nextSibling(temp))
            {
                    this->insLastChild(n1);
                    node temp2 = this->lastChild(n1);
                    this->copyAbstractTree(temp2, t2, temp);
            }

            //insert the last sibling
            this->insLastChild(n1);
            node temp2 = this->lastChild(n1);
            this->copyAbstractTree(temp2, t2, temp);
    }
}

template<class T>
void NaryTree_pointer<T>::erase( )
{
    this->erase(this->root());
}

#endif // _NARYTREE_H
```

Alberi n-ari – Realizzazione con puntatori
File: “cellnt.h”

```
#ifndef _CELLNT_H
#define _CELLNT_H

template<class T>
class Cell
{
public:
	typedef T value_type;
	typedef Cell* node;
	Cell();
	Cell(const Cell<T>&);
	Cell(const value_type&);
	~Cell();

	Cell<T>& operator =(const Cell<T>&);

	void setValue(const value_type);
	value_type getValue() const;
	void setFather(node);
	void setRight(node);
	void setLeft(node);
	void setChild(node);
	node getFather() const;
	node getRight() const;
	node getLeft() const;
	node getChild() const;

private:
	value_type value;
	node father;
	node right; // Right sibling
	node left; // Left sibling
	node child; // First child
};

template<class T>
Cell<T>::Cell()
{
	value = value_type();
	father = NULL;
	right = NULL;
	left = NULL;
	child = NULL;
}

template<class T>
Cell<T>::Cell(const Cell<T>& c2)
{
	*this = c2;
}

template<class T>
Cell<T>::Cell(const value_type& val)
{
	value = val;
	father = NULL;
	right = NULL;
```

```
        left = NULL;
        child = NULL;
}

template<class T>
Cell<T>::~Cell()
{
        value = value_type();
        father = NULL;
        right = NULL;
        left = NULL;
        child = NULL;
}

template<class T>
Cell<T>& Cell<T>::operator =(const Cell<T>& c2)
{
        if (&c2 != this) // avoid auto-assignment
        {
                setValue(c2.getValue());
                setLeft(c2.getLeft());
                setRight(c2.getRight());
                setChild(c2.getChild());
        }
        return *this;
}

template<class T>
void Cell<T>::setValue(const value_type element)
{
        value = element;
}

template<class T>
void Cell<T>::setFather(node fatherp)
{
        father = fatherp;
}

template<class T>
void Cell<T>::setRight(node rightp)
{
        right = rightp;
}

template<class T>
void Cell<T>::setLeft(node leftp)
{
        left = leftp;
}

template<class T>
void Cell<T>::setChild(node childp)
{
        child = childp;
}

template<class T>
typename Cell<T>::value_type Cell<T>::getValue() const
{
        return value;
}
```

```cpp
template<class T>
typename Cell<T>::node Cell<T>::getFather() const
{
	return father;
}

template<class T>
typename Cell<T>::node Cell<T>::getRight() const
{
	return right;
}

template<class T>
typename Cell<T>::node Cell<T>::getLeft() const
{
	return left;
}

template<class T>
typename Cell<T>::node Cell<T>::getChild() const
{
	return child;
}

#endif // _CELLNT_H
```

Alberi n-ari – Realizzazione con vettore di nodi e liste di figli

```
#ifndef _NARYTREECHILDLIST_H
#define _NARYTREECHILDLIST_H

const int defaultNumber = 15;

#include "listap.h"
#include "N-aryTree.h"

template<class T>
class NaryTree_childList: public NaryTree<T, int>
{
public:

	typedef typename NaryTree<T, int>::value_type value_type;
	typedef typename NaryTree<T, int>::node node;
	typedef List_pointer<int>::position position;

	struct Record
	{
		value_type value;
		bool used;
		List_pointer<int> children;
	};

	NaryTree_childList();
	NaryTree_childList(const value_type&);
	NaryTree_childList(const NaryTree_childList<T>&);
	~NaryTree_childList();

	NaryTree_childList<T>& operator =(const NaryTree_childList<T>&);

	void create(); // create the binary tree
	bool empty() const; // true if the three is empty
	void ins_root(node); // insert the root
	node root() const; // return the root node
	node parent(const node&) const; // return the father of the node
	bool is_leaf(const node&) const; // true if the node is a leaf
	node firstChild(const node&) const; // return the first child of the node
	bool lastSibling(const node&) const; // true if the node doesn't have a right sibling
	node nextSibling(const node&) const; // return the right sibling of the node
	void insFirstSubTree(node, const NaryTree<value_type, node>&); // add the tree like the first
child of the node
	void insSubTree(node, const NaryTree<value_type, node>&); // add the tree like sibling of the
node
	void erase(node); // erase the subtree having root = node
	value_type read(const node&) const; // read the element in the node
	void write(node, const value_type&); // write the element in the node

	node lastChild(const node) const; // return the last child of the node
	bool firstSibling(const node) const; // true if the node is the first child
	node prevSibling(const node) const; // return the left sibling of the node
	void insFirstChild(node); // add a child to the node (leftmost)
	void insLastChild(node); // add a child to the node (rightmost)
	void insRightSibling(node); // add a right sibling to the node
	void build(NaryTree_childList<T>, NaryTree_childList<T>); // build a new tree: implicit tree
must be empty, other two !empty
	void erase(); // erase the all tree
```

```
	void copyGeneralTree(node, const NaryTree<value_type, node>&, node); // copy a tree in the
implicit subtree, starting from the nodes

private:
	Record* nodes;
	node tree_root;
	int numNodes;
	int MAXNODES;

	void arrayDoubling(Record*&, const int, const int);
};

template<class T>
NaryTree_childList<T>::NaryTree_childList()
{
	nodes = 0;
	tree_root = 0;
	numNodes = 0;
	MAXNODES = defaultNumber;
	this->create();
}

template<class T>
NaryTree_childList<T>::NaryTree_childList(const value_type& value)
{
	nodes = 0;
	numNodes = 0;
	tree_root = 0;
	MAXNODES = defaultNumber;
	this->create();

	node new_root = 0;
	this->ins_root(new_root);
	this->write(this->root(), value);
}

template<class T>
NaryTree_childList<T>::NaryTree_childList(const NaryTree_childList<T>& tree)
{
	nodes = 0;
	numNodes = 0;
	tree_root = 0;
	MAXNODES = defaultNumber;
	*this = tree;
}

template<class T>
NaryTree_childList<T>::~NaryTree_childList()
{
	delete[] nodes;
}

template<class T>
NaryTree_childList<T>& NaryTree_childList<T>::operator =(const NaryTree_childList<T>& tree)
{
	if (&tree != this) // avoid auto-assignment
	{
		if (!this->empty())
			this->erase();

		this->MAXNODES = tree.MAXNODES;
		this->create();
```

```
		if (!tree.empty())
		{
			node newroot = 0;
			this->ins_root(newroot);
			this->copyGeneralTree(this->root(), tree, tree.root());
		}
	}
	return *this;
}

template<class T>
void NaryTree_childList<T>::create()
{
	numNodes = 0;
	tree_root = 0;

	nodes = new Record[MAXNODES];

	for (int i = 0; i < MAXNODES; i++)
	{
		nodes[i].value = value_type();
		nodes[i].used = false;
		nodes[i].children.create();
	}
}

template<class T>
bool NaryTree_childList<T>::empty() const
{
	return (numNodes == 0);
}

template<class T>
void NaryTree_childList<T>::ins_root(node n)
{
	if (!this->empty())
		throw std::logic_error("NaryTree_listChild (exception) - Unable to insert root
(already present)");

	tree_root = 0;
	nodes[0].used = true;
	numNodes++;

	if (n != 0)
		this->write(this->root(), this->read(n));
}

template<class T>
typename NaryTree_childList<T>::node NaryTree_childList<T>::root() const
{
	if (this->empty())
		throw std::logic_error("NaryTree_listChild (exception) - Unable to retrieve root
(already present)");

	return tree_root;
}

template<class T>
typename NaryTree_childList<T>::node NaryTree_childList<T>::parent(const node& n) const
{
	if (this->empty())
```

```
		throw std::logic_error("NaryTree_listChild (exception) - Unable to retrieve parent
(empty tree)");

	if (n == tree_root)
		throw std::logic_error("NaryTree_listChild (exception) - Unable to get parent
(root)");

	if (nodes[n].used == false)
		throw std::logic_error("NaryTree_listChild (exception) - Unable to retrieve parent
(invalid node)");

	position child;
	int p;
	for (int i = 0; i < MAXNODES; i++)
	{
		if (!nodes[i].children.empty())
		{
			child = nodes[i].children.begin();
			bool found = false;
			while (!nodes[i].children.end(child) && !found)
			{
				if (nodes[i].children.read(child) == n)
				{
					found = true;
					p = i;
				}
				child = nodes[i].children.next(child);
			}
			if (found)
				return (i);
		}
	}

	throw std::logic_error("NaryTree_listChild (exception) - Unable to get parent (root)");
}

template<class T>
bool NaryTree_childList<T>::is_leaf(const node& n) const
{
	if (this->empty())
		throw std::logic_error("NaryTree_listChild (exception) - Unable to check if node is
leaf (empty tree)");

	if (nodes[n].used == false)
		throw std::logic_error("NaryTree_listChild (exception) - Unable to check if node is
leaf (invalid node)");

	return (nodes[n].children.empty());
}

template<class T>
typename NaryTree_childList<T>::node NaryTree_childList<T>::firstChild(const node& n) const
{
	if (this->empty())
		throw std::logic_error("NaryTree_listChild (exception) - Unable to retrieve first
child (empty tree)");

	if (nodes[n].used == false)
		throw std::logic_error("NaryTree_listChild (exception) - Unable to retrieve first
child (invalid node)");

	if (this->is_leaf(n))
```

```
		throw std::logic_error("NaryTree_listChild (exception) - Unable to retrieve firstChild
(node is leaf)");

	return nodes[n].children.read(nodes[n].children.begin());
}

template<class T>
bool NaryTree_childList<T>::lastSibling(const node& n) const
{
	if (this->empty())
		throw std::logic_error("NaryTree_listChild (exception) - Unable to check if node is
last sibling (empty tree)");

	if (nodes[n].used == false)
		throw std::logic_error("NaryTree_listChild (exception) - Unable to check if node is
last sibling (invalid node)");

	if (n == tree_root)
		throw std::logic_error("NaryTree_listChild (exception) - Unable to check node (root
doesn't have sibling)");

	node p = this->parent(n);
	position c = nodes[p].children.begin();

	while (!nodes[p].children.end(nodes[p].children.next(c)))
		c = nodes[p].children.next(c);

	return (n == nodes[p].children.read(c));
}

template<class T>
typename NaryTree_childList<T>::node NaryTree_childList<T>::nextSibling(const node& n) const
{
	if (this->empty())
		throw std::logic_error("NaryTree_listChild (exception) - Unable to check node (empty
tree)");

	if (nodes[n].used == false)
		throw std::logic_error("NaryTree_listChild (exception) - Unable to retrieve next
sibling (invalid node)");

	if (n == tree_root)
		throw std::logic_error("NaryTree_listChild (exception) - Unable to check node (root
doesn't have sibling)");

	if (this->lastSibling(n))
		throw std::logic_error("NaryTree_listChild (exception) - Unable to get next sibling
(node is last sibling)");

	node p = this->parent(n);
	position c = nodes[p].children.begin();
	while (!nodes[p].children.end(c))
	{
		if (nodes[p].children.read(c) == n)
			return nodes[p].children.read(nodes[p].children.next(c));
		c = nodes[p].children.next(c);
	}

	throw std::logic_error("NaryTree_listChild (exception) - Unable to get next sibling");
}
```

```
template<class T>
void NaryTree_childList<T>::insFirstSubTree(node n1, const NaryTree<value_type, node>& t)
{
	if (this->empty())
		throw std::logic_error("NaryTree_listChild (exception) - Unable to insert subtree
(empty tree)");

	if (nodes[n1].used == false)
		throw std::logic_error("NaryTree_listChild (exception) - Unable to insert subtree
(invalid node)");

	if (t.empty())
		throw std::logic_error("NaryTree_listChild (exception) - Unable to insert subtree
(empty subtree)");

	if (!this->is_leaf(n1))
		throw std::logic_error("NaryTree_listChild (exception) - Unable to insert subtree
(node is not a leaf)");

	this->insFirstChild(n1);
	node temp = this->firstChild(n1);

	this->copyGeneralTree(temp, t, t.root());
}

template<class T>
void NaryTree_childList<T>::insSubTree(node n, const NaryTree<value_type, node>& t2)
{
	if (this->empty())
		throw std::logic_error("NaryTree_listChild (exception) - Unable to insert subtree
(empty tree)");

	if (nodes[n].used == false)
		throw std::logic_error("NaryTree_listChild (exception) - Unable to insert subtree
(invalid node)");

	if (t2.empty())
		throw std::logic_error("NaryTree_listChild (exception) - Unable to insert subtree
(empty subtree)");

	if (n == this->root())
		throw std::logic_error("NaryTree_listChild (exception) - Unable to insert subtree
(node is root)");

	this->insRightSibling(n);
	node temp = this->nextSibling(n);
	node root = t2.root();
	this->copyGeneralTree(temp, t2, root);
}

template<class T>
void NaryTree_childList<T>::erase(node n)
{
	if (this->empty())
		throw std::logic_error("NaryTree_listChild (exception) - Unable to erase node (empty
tree)");

	if (nodes[n].used == false)
		throw std::logic_error("NaryTree_listChild (exception) - Unable to erase node (invalid
node)");

	if (!this->is_leaf(n))
```

```
		while (!nodes[n].children.empty())
			this->erase(nodes[n].children.read(nodes[n].children.begin()));

	if (n != 0)
	{
		node p = parent(n);
		position c = nodes[p].children.begin();
		while (nodes[p].children.read(c) != n)
			c = nodes[p].children.next(c);
		nodes[p].children.erase(c);
	}
	nodes[n].value = value_type();
	nodes[n].used = false;
	numNodes--;
}

template<class T>
typename NaryTree_childList<T>::value_type NaryTree_childList<T>::read(const node& n) const
{
	if (nodes[n].used == false)
		throw std::logic_error("NaryTree_listChild (exception) - Unable to read label (invalid
node)");

	return (nodes[n].value);
}

template<class T>
void NaryTree_childList<T>::write(node n, const value_type& el)
{
	if (nodes[n].used == false)
		throw std::logic_error("NaryTree_listChild (exception) - Unable to write label
(invalid node)");

	nodes[n].value = el;
}

template<class T>
typename NaryTree_childList<T>::node NaryTree_childList<T>::lastChild(const node n) const
{
	if (nodes[n].used == false)
		throw std::logic_error("NaryTree_listChild (exception) - Unable to retrieve last child
(invalid node)");

	if (this->is_leaf(n))
		throw std::logic_error("NaryTree_listChild (exception) - Unable to read child (node is
leaf)");

	position c = nodes[n].children.begin();
	while (!nodes[n].children.end(nodes[n].children.next(c)))
		c = nodes[n].children.next(c);

	return nodes[n].children.read(c);
}

template<class T>
bool NaryTree_childList<T>::firstSibling(const node n) const
{
	if (nodes[n].used == false)
		throw std::logic_error("NaryTree_listChild (exception) - Unable to check if node is
first sibling (invalid node)");

	if (n == tree_root)
```

```
		throw std::logic_error("NaryTree_listChild (exception) - Unable to get sibling (root doesn't have)");

	return n == this->firstChild(this->parent(n));
}

template<class T>
typename NaryTree_childList<T>::node NaryTree_childList<T>::prevSibling(const node n) const
{
	if (nodes[n].used == false)
		throw std::logic_error("NaryTree_listChild (exception) - Unable to retrieve previous sibling (invalid node)");

	if (this->firstSibling(n))
		throw std::logic_error("NaryTree_listChild (exception) - Unable to get previous sibling (first child)");

	if (n == tree_root)
		throw std::logic_error("NaryTree_listChild (exception) - Unable to get sibling (root doesn't have)");

	node previous = this->firstChild(this->parent(n));

	while (this->nextSibling(previous) != n)
		previous = this->nextSibling(previous);

	return previous;
}

template<class T>
void NaryTree_childList<T>::insFirstChild(node n)
{
	if (nodes[n].used == false)
		throw std::logic_error("NaryTree_listChild (exception) - Unable to insert child (invalid node)");

	if (numNodes >= MAXNODES)
		this->arrayDoubling(nodes, MAXNODES, MAXNODES * 2);

	int k = 0;
	while (k < MAXNODES && nodes[k].used == true) // find a free position in nodes[]
		k++;

	nodes[k].used = true;
	nodes[n].children.insert(k, nodes[n].children.begin());
	numNodes++;

}

template<class T>
void NaryTree_childList<T>::insLastChild(node n)
{
	if (nodes[n].used == false)
		throw std::logic_error("NaryTree_listChild (exception) - Unable to insert child (invalid node)");

	if (numNodes >= MAXNODES)
		this->arrayDoubling(nodes, MAXNODES, MAXNODES * 2);

	if (!nodes[n].children.empty())
	{
		node temp = this->lastChild(n);
```

```
                this->insRightSibling(temp);
        }
        else
                this->insFirstChild(n);
}

template<class T>
void NaryTree_childList<T>::insRightSibling(node n)
{
        if (nodes[n].used == false)
                throw std::logic_error("NaryTree_listChild (exception) - Unable to insert child
(invalid node)");

        if (n == tree_root)
                throw std::logic_error("NaryTree_listChild (exception) - Unable to insert sibling
(root can't have)");

        if (numNodes >= MAXNODES)
                this->arrayDoubling(nodes, MAXNODES, MAXNODES * 2);

        int k = 0;
        while (k < MAXNODES && nodes[k].used == true) // find a free position in nodes[]
                k++;

        nodes[k].used = true;
        node p = this->parent(n);
        position child = nodes[p].children.begin();
        bool found = false;
        while (!nodes[p].children.end(child) && !found)
        {
                if (nodes[p].children.read(child) == n)
                        found = true;
                child = nodes[p].children.next(child);
        }
        nodes[p].children.insert(k, child);
        numNodes++;

}

template<class T>
void NaryTree_childList<T>::build(NaryTree_childList<T> t1, NaryTree_childList<T> t2)
{
        if (!this->empty() || t1.empty() || t2.empty())
                throw std::logic_error("NaryTree_childList (exception) - Unable to build tree");

        node temp = 0;
        this->ins_root(temp);
        this->write(this->root(), value_type());
        temp = this->root();
        this->insFirstSubTree(temp, t1);
        temp = this->firstChild(root());
        this->insSubTree(temp, t2);
}

template<class T>
void NaryTree_childList<T>::erase()
{
        this->erase(this->root());
}
```

```
template<class T>
void NaryTree_childList<T>::copyGeneralTree(node n1, const NaryTree<value_type, node>&t2, node n2)
{
	this->write(n1, t2.read(n2));

	if (!t2.is_leaf(n2))
	{
		node temp;
		node temp2;

		for (temp = t2.firstChild(n2); !t2.lastSibling(temp); temp = t2.nextSibling(temp))
		{
			// we are not on the first child of n2
			if (t2.firstChild(n2) != temp)
			{
				node t = this->lastChild(n1);
				this->insRightSibling(t);
				temp2 = this->lastChild(n1);
			}
			else
			{
				this->insFirstChild(n1);
				temp2 = this->firstChild(n1);
			}

			this->copyGeneralTree(temp2, t2, temp);
		}

		// we are not on the first child of n2
		if (t2.firstChild(n2) != temp)
		{
			this->insRightSibling(this->lastChild(n1));
			temp2 = this->lastChild(n1);
		}
		else
		{
			this->insFirstChild(n1);
			temp2 = this->firstChild(n1);
		}
		this->copyGeneralTree(temp2, t2, temp);
	}
}

template<class T>
void NaryTree_childList<T>::arrayDoubling(Record*& a, const int vecchiaDim, const int nuovaDim)
{
	MAXNODES = (MAXNODES * 2);
	Record* temp = new Record[nuovaDim];

	for (int i = 0; i < nuovaDim; i++)
	{
		temp[i].value = value_type();
		temp[i].used = false;
		temp[i].children.create();
	}

	int number;
	if (vecchiaDim < nuovaDim)
		number = vecchiaDim;
	else
		number = nuovaDim;
```

```
	for (int i = 0; i < number; i++)
	{
		temp[i].value = a[i].value;
		temp[i].used = a[i].used;
		temp[i].children = a[i].children;
	}

	delete[] a;
	a = temp;
}

#endif // _NARYTREECHILDLIST_H
```

Alberi n-ari – Tester

```
#include <iostream>
#include "N-aryTree_childList.h"

int main()
{

       NaryTree_childList<int> t1, t2, t3, t4, t5;
       NaryTree_childList<int>::node n1 = 0, n2 = 0, n3 = 0;

       t1.ins_root(n1);
       n1 = t1.root();
       t1.write(n1, 1);
       t1.insLastChild(n1);
       n1 = t1.lastChild(n1);
       t1.write(n1, 2);
       n1 = t1.parent(n1);
       t1.insLastChild(n1);
       n1 = t1.lastChild(n1);
       t1.write(n1, 3);
       n1 = t1.parent(n1);
       t1.insLastChild(n1);
       n1 = t1.lastChild(n1);
       t1.write(n1, 4);
       n1 = t1.parent(n1);
       n1 = t1.firstChild(n1);
       t1.insLastChild(n1);
       n1 = t1.lastChild(n1);
       t1.write(n1, 6);
       n1 = t1.parent(n1);
       t1.insLastChild(n1);
       n1 = t1.lastChild(n1);
       t1.write(n1, 11);
       n1 = t1.parent(n1);
       t1.insLastChild(n1);
       n1 = t1.lastChild(n1);
       t1.write(n1, 7);

       t2.ins_root(n2);
       n2 = t2.root();
       t2.write(n2, 5);
       t2.insLastChild(n2);
       n2 = t2.lastChild(n2);
       t2.write(n2, 9);
       n2 = t2.parent(n2);
       t2.insLastChild(n2);
       n2 = t2.lastChild(n2);
       t2.write(n2, 20);
       n2 = t2.parent(n2);
       t2.insLastChild(n2);
       n2 = t2.lastChild(n2);
       t2.write(n2, 10);
       n2 = t2.parent(n2);
       n2 = t2.firstChild(n2);
       t2.insLastChild(n2);
       n2 = t2.lastChild(n2);
       t2.write(n2, 6);

       t3.ins_root(n3);
```

```
        n3 = t3.root();
        t3.write(n3, 1);
        t3.insLastChild(n3);
        n3 = t3.lastChild(n3);
        t3.write(n3, 2);
        n3 = t3.parent(n3);
        t3.insLastChild(n3);
        n3 = t3.lastChild(n3);
        t3.write(n3, 3);
        t3.insRightSibling(n3);
        n3 = t3.parent(n3);
        n3 = t3.lastChild(n3);
        t3.write(n3, 4);
        n3 = t3.parent(n3);
        n3 = t3.firstChild(n3);
        t3.insRightSibling(n3);
        n3 = t3.nextSibling(n3);
        t3.write(n3, 6);

        t4 = t3;
        t4 = t2;

        cout << "T1 " << t1 << endl;

        cout << "T2 " << t2 << endl;

        cout << "T3 " << t3 << endl;

        cout << "T4 " << t4 << endl;

        t5.build(t4, t3);

        cout << "T5 " << t5 << endl;

        cout << "T5 height: " << t5.height() << endl;

        cout << "T5 grade: " << t5.gradeN() << endl;

        cout << "T5 max label: " << t5.max() << endl;

        cout << endl;

        if (t4 == t2)
               cout << "T4 = T2" << endl;
        else
               cout << "T4 != T2" << endl;

}
```

Alberi n-ari – Output Tester

```
T1 [ 1: [ 2: [ 6 ], [ 11 ], [ 7 ] ], [ 3 ], [ 4 ] ]

T2 [ 5: [ 9: [ 6 ] ], [ 20 ], [ 10 ] ]

T3 [ 1: [ 2 ], [ 6 ], [ 3 ], [ 4 ] ]

T4 [ 5: [ 9: [ 6 ] ], [ 20 ], [ 10 ] ]

T5 [ 0: [ 5: [ 9: [ 6 ] ], [ 20 ], [ 10 ] ], [ 1: [ 2 ], [ 6 ], [ 3 ], [ 4 ] ] ]

T5 height: 4
T5 grade: 4
T5 max label: 20

T4 = T2
```

Code con priorità – Classe astratta

```
#ifndef _PRIORQUEUE_H_
#define _PRIORQUEUE_H_

#include <iostream>
#include <sstream>
#include <stdexcept>
#include <vector>

using std::cout;
using std::ostream;
using std::endl;
using std::vector;

template<class E, class P>
class PriorQueue
{
public:
	typedef E elem;
	typedef P priority;

	virtual ~PriorQueue()
	{
	}

	virtual bool empty() const = 0; // true if the tree is empty
	virtual void insert(const elem&) = 0; // insert an element in the queue
	virtual void pop() = 0; // delete the first element having highest priority (for integer: 
minimum integer = max priority)
	virtual elem getMin() const = 0; // return the the first element with the highest priority 
(for integer: minimum integer = max priority)
	virtual elem getMax() const = 0; // return the the last element with the lowest priority (for 
integer: minimum integer = max priority)

	template<class X, class Y>
	friend ostream& operator <<(ostream &, PriorQueue<X, Y>&);

	void clear(); // delete the queue
	bool find(const elem&); // true if the element is present in the list
	int size(); // return the element of the queue
	void join(PriorQueue<E, P>&, PriorQueue<E, P>&); // join two queues: implicit queue must be 
empty
	void eraseElem(const elem&); // erase from the queue the element
	void changeMin(const elem&); // change the current min with a new element

protected:
	virtual void create() = 0; // create the queue

};

template<class E, class P>
ostream& operator <<(ostream& out, PriorQueue<E, P>& queue)
{
	if (!queue.empty())
	{
		out << "Element (Priority):" << endl;
		vector<E> elements;

		while (!queue.empty())
```

```
		{
			elements.push_back(queue.getMin());
			queue.pop();
		}

		for (typename vector<E>::iterator it = elements.begin(); it != elements.end(); it++)
		{
			out << *it;
			out << endl;
			queue.insert(*it);
		}
	}
	else
		out << "Empty Priority Queue" << endl;

	out << endl;
	return out;

}

template<class E, class P>
void PriorQueue<E, P>::clear()
{
	while (!this->empty())
		this->pop();
}

template<class E, class P>
bool PriorQueue<E, P>::find(const elem& el)
{
	if (this->empty())
		return false;

	vector<E> elements;
	bool flag = false;

	while (!this->empty())
	{
		elements.push_back(this->getMin());
		if (this->getMin() == el)
			flag = true;
		this->pop();
	}

	while (!elements.empty())
	{
		this->insert(elements.back());
		elements.pop_back();
	}

	return flag;
}

template<class E, class P>
int PriorQueue<E, P>::size()
{
	vector<E> elements;
	int size = 0;

	while (!this->empty())
	{
		elements.push_back(this->getMin());
```

```
            this->pop();
            size++;
        }

        while (!elements.empty())
        {
            this->insert(elements.back());
            elements.pop_back();
        }

        return size;
}

template<class E, class P>
void PriorQueue<E, P>::join(PriorQueue<E, P>& queue1, PriorQueue<E, P>& queue2)
{
        if (!this->empty())
            throw std::logic_error("Priority Queue (exception)");

        vector<E> elements;

        //Copy the first queue
        while (!queue1.empty())
        {
            elements.push_back(queue1.getMin());
            this->insert(queue1.getMin());
            queue1.pop();
        }

        while (!elements.empty())
        {
            queue1.insert(elements.back());
            elements.pop_back();
        }

        //Copy the second queue
        while (!queue2.empty())
        {
            elements.push_back(queue2.getMin());
            this->insert(queue2.getMin());
            queue2.pop();
        }

        while (!elements.empty())
        {
            queue2.insert(elements.back());
            elements.pop_back();
        }
}

template<class E, class P>
void PriorQueue<E, P>::eraseElem(const elem& el)
{
        if (this->find(el))
        {
            vector<E> elements;

            while (!this->empty())
            {
                if (this->getMin() != el)
                    elements.push_back(this->getMin());
                this->pop();
```

```
		}

		while (!elements.empty())
		{
			this->insert(elements.back());
			elements.pop_back();
		}
	}
}

template<class E, class P>
void PriorQueue<E, P>::changeMin(const elem& el)
{
	if (!this->empty())
		if (el < this->getMin())
		{
			this->pop();
			this->insert(el);
		}
}

#endif /* _PRIORQUEUE_H_ */
```

Code con priorità – Realizzazione con lista ordinata

```
#ifndef _PQLIST_H_
#define _PQLIST_H_

#include "priority_queue.h"
#include "elempq.h"
#include "listap.h"

// Labels can be duplicated
// Priority can be duplicated
// For numbers: minimum number = max priority; for string: first in alphabetical order = max
priority
// Can't change one item's label
// Can't change one item's priority

template<class T, class P>
class PriorityQueue_list: public PriorQueue<Elem<T, P>, P>
{
public:
	typedef typename PriorQueue<Elem<T, P>, P>::elem elem;
	typedef typename PriorQueue<Elem<T, P>, P>::priority priority;
	typedef T value_type;

	PriorityQueue_list();
	PriorityQueue_list(const PriorityQueue_list<T, P>&);
	~PriorityQueue_list();

	PriorityQueue_list<T, P>& operator =(const PriorityQueue_list<T, P>&);
	bool operator == (const PriorityQueue_list<T, P>&);
	bool operator != (const PriorityQueue_list<T, P>&);

	bool empty() const; // true if the queue is empty
	void insert(const elem&); // insert an element in the queue
	void pop(); // delete the first element having highest priority
	elem getMin() const; // return the the first element with the highest priority
	elem getMax() const; // return the the last element with the lowest priority

	void insert(const value_type&, const priority&); // insert an element in the queue
	void erasePriorities(const priority&); // erase all the elements having that priority
	void eraseItems(const value_type&); // erase all the elements having that items

private:
	List_pointer<Elem<T, P> > queue;

	void create(); // create the queue
};

template<class T, class P>
PriorityQueue_list<T, P>::PriorityQueue_list()
{
	this->create();
}

template<class T, class P>
PriorityQueue_list<T, P>::PriorityQueue_list(const PriorityQueue_list<T, P>& queue)
{
	this->create();
	*this = queue;
}
```

```
template<class T, class P>
PriorityQueue_list<T, P>::~PriorityQueue_list()
{
	this->clear();
}

template<class T, class P>
PriorityQueue_list<T, P>& PriorityQueue_list<T, P>::operator =(const PriorityQueue_list<T, P>&
queue2)
{
	if (&queue2 != this) // avoid auto-assignment
	{
		if (!this->queue.empty())
			this->queue.clear();

		this->queue = queue2.queue;
	}

	return *this;
}

template<class T, class P>
bool PriorityQueue_list<T, P>::operator == (const PriorityQueue_list<T, P>& h2)
{
	return (this->queue == h2.queue);
}

template<class T, class P>
bool PriorityQueue_list<T, P>::operator != (const PriorityQueue_list<T, P>& h2)
{
	return (!(*this == h2));
}

template<class T, class P>
bool PriorityQueue_list<T, P>::empty() const
{
	return queue.empty();
}

template<class T, class P>
void PriorityQueue_list<T, P>::insert(const elem& e)
{
	queue.insOrd(e);
}

template<class T, class P>
void PriorityQueue_list<T, P>::insert(const value_type& val, const priority& pri)
{
	this->insert(Elem<T, P>(val, pri));
}

template<class T, class P>
void PriorityQueue_list<T, P>::pop()
{
	if (this->empty())
		throw std::logic_error("PriorityQueue_list (exception) - Unable to pop (empty
queue)");

	queue.pop_front();
}
```

```
template<class T, class P>
typename PriorityQueue_list<T, P>::elem PriorityQueue_list<T, P>::getMin() const
{
	if (this->empty())
		throw std::logic_error("PriorityQueue_list (exception) - Unable to getMin (empty
queue)");

	return queue.read(queue.begin());
}

template<class T, class P>
typename PriorityQueue_list<T, P>::elem PriorityQueue_list<T, P>::getMax() const
{
	if (this->empty())
		throw std::logic_error("PriorityQueue_list (exception) - Unable to getMax (empty
queue)");

	return queue.read(queue.endnode());
}

template<class T, class P>
void PriorityQueue_list<T, P>::create()
{
	queue.create();
}

template<class T, class P>
void PriorityQueue_list<T, P>::erasePriorities(const priority& p)
{
	vector<elem> elements;

	//Copy the first queue
	while (!this->empty())
	{
		if ((this->getMin()).getPriority() != p)
			elements.push_back(this->getMin());
		this->pop();
	}

	while (!elements.empty())
	{
		this->insert(elements.back());
		elements.pop_back();
	}
}

template<class T, class P>
void PriorityQueue_list<T, P>::eraseItems(const value_type& i)
{
	vector<elem> elements;

	//Copy the first queue
	while (!this->empty())
	{
		if ( (this->getMin()).getValue() != i)
			elements.push_back(this->getMin());
		this->pop();
	}

	while (!elements.empty())
	{
		this->insert(elements.back());
```

```
            elements.pop_back();
        }
}

#endif /* _PQLIST_H_ */
```

Code con priorità – Realizzazione con heap

```
#ifndef _PQHEAP_H_
#define _PQHEAP_H_

#include "priority_queue.h"
#include "elempq.h"

// Labels can be duplicated
// Priority can be duplicated
// For numbers: minimum number = max priority; for string: first in alphabetical order = max
priority
// Can't change one item's label
// Can't change one item's priority

template<class T, class P>
class Heap: public PriorQueue<Elem<T, P>, P>
{

public:
	typedef typename PriorQueue<Elem<T, P>, P>::elem elem;
	typedef typename PriorQueue<Elem<T, P>, P>::priority priority;
	typedef T value_type;

	Heap();
	Heap(int); // size
	Heap(const Heap<T, P>&);
	~Heap();

	Heap<T, P>& operator =(const Heap<T, P>&);
	bool operator == (const Heap<T, P>&);
	bool operator != (const Heap<T, P>&);

	bool empty() const; // true if the tree is empty
	void insert(const elem&); // insert an element in the queue
	void pop(); // delete the first element having highest priority
	elem getMin() const; // return the the first element with the highest priority
	elem getMax() const; // return the the last element with the lowest priority

	void insert(const value_type&, const priority&); // insert an element in the queue
	void erasePriorities(const priority&); // erase all the elements having that priority
	void eraseItems(const value_type&); // erase all the elements having that items

private:
	int MAXLEN;
	elem *heap;
	int freePos; // first empty position of the heap (= # of elements in the heap)

	void create(); // create the queue

	void arrayDoubling(elem*&, const int, const int);
	void fixUp();
	void fixDown(int, int);

};

template<class T, class P>
Heap<T, P>::Heap()
{
	MAXLEN = 100;
```

```
	heap = 0;
	freePos = 0;
	this->create();
}

template<class T, class P>
Heap<T, P>::Heap(int size)
{
	if (size <= 0)
		size = 100; // default

	MAXLEN = size;
	heap = 0;
	freePos = 0;
	this->create();
}

template<class T, class P>
Heap<T, P>::Heap(const Heap<T, P>& queue)
{
	*this = queue;
}

template<class T, class P>
Heap<T, P>::~Heap()
{
	this->clear();
}

template<class T, class P>
Heap<T, P>& Heap<T, P>::operator =(const Heap<T, P>& queue2)
{
	if (&queue2 != this) // avoid auto-assignment
	{
		if (!this->empty())
			this->clear();

		this->MAXLEN = queue2.MAXLEN;
		this->freePos = queue2.freePos;
		this->heap = new elem[MAXLEN];

		for (int i = 0; i < freePos; i++)
			this->heap[i] = queue2.heap[i];
	}

	return *this;
}

template<class T, class P>
bool Heap<T, P>::operator == (const Heap<T, P>& h2)
{
	if (this->freePos != h2.freePos)
		return false;

	for(int i = 0; i < freePos; i++)
		if(this->heap[i] != h2.heap[i])
			return false;

	return true;
}
```

```
template<class T, class P>
bool Heap<T, P>::operator != (const Heap<T, P>& h2)
{
	return (!(*this == h2));
}

template<class T, class P>
bool Heap<T, P>::empty() const
{
	return (freePos == 0);
}

template<class T, class P>
void Heap<T, P>::insert(const elem& e)
{
	if (freePos == MAXLEN)
	{
		this->arrayDoubling(heap, MAXLEN, MAXLEN * 2);
		MAXLEN = MAXLEN * 2;
	}

	heap[freePos] = e;
	freePos++;
	this->fixUp();
}

template<class T, class P>
void Heap<T, P>::insert(const value_type& val, const priority& pri)
{
	this->insert(Elem<T, P>(val, pri));
}

template<class T, class P>
void Heap<T, P>::pop()
{
	if (this->empty())
		throw std::logic_error("Heap (exception) - Unable to pop min (empty queue)");

	heap[0] = heap[freePos - 1];
	freePos--;
	this->fixDown(1, freePos);
}

template<class T, class P>
typename Heap<T, P>::elem Heap<T, P>::getMin() const
{
	if (this->empty())
		throw std::logic_error("Heap (exception) - Unable to get min (empty queue)");

	return heap[0];
}

template<class T, class P>
typename Heap<T, P>::elem Heap<T, P>::getMax() const
{
	if (this->empty())
		throw std::logic_error("Heap (exception) - Unable to get max (empty queue)");

	return heap[freePos - 1];
}
```

```
template<class T, class P>
void Heap<T, P>::create()
{
	heap = new elem[MAXLEN];
}

template<class T, class P>
void Heap<T, P>::arrayDoubling(elem*& a, const int oldSize, const int newSize)
{
	elem* temp = new elem[newSize];

	for (int i = 0; i < oldSize; i++)
		temp[i] = a[i];

	delete[] a;
	a = temp;
}

template<class T, class P>
void Heap<T, P>::fixUp()
{
	int k = freePos;

	while (k > 1 && heap[k - 1] < heap[k / 2 - 1])
	{
		elem tmp = elem();
		tmp = heap[k - 1];
		heap[k - 1] = heap[k / 2 - 1];
		heap[k / 2 - 1] = tmp;
		k = k / 2;
	}
}

template<class T, class P>
void Heap<T, P>::fixDown(int k, int N)
{
	short int scambio = 1;

	while (k <= N / 2 && scambio)
	{
		int j = 2 * k;
		elem tmp = elem();
		if (j < N && heap[j - 1] > heap[j])
			j++;
		if ((scambio = heap[j - 1] < heap[k - 1]))
		{
			tmp = heap[k - 1];
			heap[k - 1] = heap[j - 1];
			heap[j - 1] = tmp;
			k = j;
		}
	}
}

template<class T, class P>
void Heap<T, P>::erasePriorities(const priority& p)
{
	vector<elem> elements;

	//Copy the first queue
	while (!this->empty())
	{
```

```
        if ( (this->getMin()).getPriority() != p)
            elements.push_back(this->getMin());
        this->pop();
    }

    while (!elements.empty())
    {
        this->insert(elements.back());
        elements.pop_back();
    }
}

template<class T, class P>
void Heap<T, P>::eraseItems(const value_type& i)
{
    vector<elem> elements;

    //Copy the first queue
    while (!this->empty())
    {
        if ( (this->getMin()).getValue() != i)
            elements.push_back(this->getMin());
        this->pop();
    }

    while (!elements.empty())
    {
        this->insert(elements.back());
        elements.pop_back();
    }
}

#endif /* _PQHEAP_H_ */
```

Code con priorità
File: “elempq.h”

```
#ifndef _ELEMPQ_H
#define _ELEMPQ_H

template<class T, class P>
class Elem
{
public:

	typedef T value_type;
	typedef P priority;

	Elem();
	Elem(const Elem<T, P>&);
	Elem(const value_type&, const priority&);
	~Elem();

	Elem<T, P>& operator =(const Elem<T, P>&);
	bool operator ==(const Elem<T, P>&) const;
	bool operator !=(const Elem<T, P>&) const;
	bool operator >(const Elem<T, P>&) const;
	bool operator >=(const Elem<T, P>&) const;
	bool operator <(const Elem<T, P>&) const;

	void setValue(const value_type&);
	value_type getValue() const;
	void setPriority(const priority&);
	priority getPriority() const;

private:
	value_type value;
	priority prior;
};

template<class T, class P>
ostream& operator <<(ostream& out, const Elem<T, P>& elem)
{
	out << elem.getValue() << " (" << elem.getPriority() << ")";
	return out;
}

template<class T, class P>
Elem<T, P>::Elem()
{
	value = value_type();
	prior = priority();
}

template<class T, class P>
Elem<T,P>::Elem(const Elem<T, P>& el2)
{
	*this = el2;
}

template<class T, class P>
Elem<T, P>::Elem(const value_type& val, const priority& pri)
{
	value = val;
```

```
        prior = pri;
}

template<class T, class P>
Elem<T, P>::~Elem()
{
        value = value_type();
        prior = priority();
}

template<class T, class P>
Elem<T, P>& Elem<T, P>::operator =(const Elem<T, P>& elem2)
{
        if (&elem2 != this) // avoid auto-assignment
        {
                this->setValue(elem2.getValue());
                this->setPriority(elem2.getPriority());
        }
        return *this;
}

template<class T, class P>
bool Elem<T, P>::operator ==(const Elem<T, P>& elem2) const
{
        if (this->getValue() != elem2.getValue())
                return false;

        if (this->getPriority() != elem2.getPriority())
                return false;

        return true;
}

template<class T, class P>
bool Elem<T, P>::operator !=(const Elem<T, P>& elem2) const
{
        return (!(*this == elem2));
}

template<class T, class P>
bool Elem<T, P>::operator >(const Elem<T, P>& elem2) const
{
        return (this->getPriority() > elem2.getPriority());
}

template<class T, class P>
bool Elem<T, P>::operator >=(const Elem<T, P>& elem2) const
{
        return (this->getPriority() >= elem2.getPriority());
}

template<class T, class P>
bool Elem<T, P>::operator <(const Elem<T, P>& elem2) const
{
        return (this->getPriority() < elem2.getPriority());
}

template<class T, class P>
void Elem<T, P>::setValue(const value_type& element)
{
        value = element;
}
```

```
template<class T, class P>
typename Elem<T, P>::value_type Elem<T, P>::getValue() const
{
	return value;
}

template<class T, class P>
void Elem<T, P>::setPriority(const priority& pri)
{
	prior = pri;
}

template<class T, class P>
typename Elem<T, P>::priority Elem<T, P>::getPriority() const
{
	return prior;
}

#endif // _ELEMPQ_H
```

Code con priorità – Tester

```
#include "heap.h"

#include <string>
using std::string;

int main()
{
	Heap<string, int> queue(3);
	Heap<string, int> queue2(3);

	queue.insert("Turing", 2);
	queue.insert("Babbage", 3);
	queue.insert("Dijkstra", 4);
	queue.insert("Chomsky", 3);
	queue.insert("Ada", 7);
	queue.insert(Elem<string, int>("Turing", 5));

	cout << "queue1: " << endl << queue;

	queue2 = queue;

	cout << "queue2: " << endl << queue2;


	if (queue2 == queue)
		cout << "queue2 = queue" << endl;
	else
		cout << "queue2 != queue" << endl;

	cout << endl;

	cout << "queue1 max: " << queue.getMax() << endl;
	cout << "queue1 min: " << queue.getMin() << endl;

	cout << endl;
	queue.eraseItems("Turing");
	queue.erasePriorities(3);
	cout << "queue1: " << endl << queue;

	queue.pop();
	cout << "queue1: " << endl << queue;

}
```

Code con priorità – Output Tester

```
queue1:
Element (Priority):
Turing (2)
Babbage (3)
Chomsky (3)
Dijkstra (4)
Turing (5)
Ada (7)

queue2:
Element (Priority):
Turing (2)
Babbage (3)
Chomsky (3)
Dijkstra (4)
Turing (5)
Ada (7)

queue2 = queue

queue1 max: Ada (7)
queue1 min: Turing (2)

queue1:
Element (Priority):
Dijkstra (4)
Ada (7)

queue1:
Element (Priority):
Ada (7)
```

Dizionari – Classe astratta

```
#ifndef _DICTIONARY_H
#define _DICTIONARY_H

#include <stdexcept>
#include "elemdic.h"

template<class K, class I>
class Dictionary
{
public:

	typedef K key;
	typedef I item;

	virtual ~Dictionary()
	{
	}

	virtual bool empty() const = 0;
	virtual Element<K, I>* find(const key&) const = 0;
	virtual void insert(const Element<K, I>) = 0;
	virtual void erase(const key&) = 0;
	virtual void modify(const key&, const item&) = 0;

protected:
	virtual void create() = 0;
};

#endif //_DICTIONARY_H
```

Dizionari – Realizzazione con liste di trabocco

```
#ifndef _OVERFLOW_LIST_H
#define _OVERFLOW_LIST_H

#include <vector>
#include <iostream>
#include <algorithm>

#include "dictionary.h"
#include "hash.h"
#include "listap.h"

using std::vector;
using std::cout;
using std::endl;

//Keys are unique. Inserting two times the same keys while cause overwrite of the label

template<class K, class E>
class Overflow_list: public Dictionary<K, E>
{
public:

	typedef typename Dictionary<K, E>::key key;
	typedef typename Dictionary<K, E>::item element;

	Overflow_list();
	Overflow_list(int); // size
	Overflow_list(const Overflow_list<K, E>&);
	~Overflow_list();

	Overflow_list<K, E>& operator =(const Overflow_list<K, E>&);
	bool operator ==(const Overflow_list<K, E>&) const;
	bool operator !=(const Overflow_list<K, E>&) const;

	bool empty() const;
	Element<K, E>* find(const key&) const;
	void insert(const Element<K, E>);
	void erase(const key& k);
	void modify(const key& k, const element& e);

	int size() const;
	int getBucket(const key&) const; // find the home bucket of the tombstone
	vector<K> keyArray() const;
	vector<E> itemArray() const;
	vector<Element<K, E> > pairArray() const;
	bool containsValue(const element&) const;
	void join(const Overflow_list<K, E>&); // add in the implicit dictionary all the elements of 
the parameter
	bool hasSubset(const Overflow_list<K, E>&) const; // true if all the elements in the 
parameter are present in the implicit dictionary
	bool find(const Element<K, E>) const; // true if the element is present in the dictionary

private:

	List_pointer<Element<K, E>*>* lists; // overflow lists
	Hash<K> hashm; // maps type K to non-negative integer
	int MAXSIZE; // hash function divisor
	int dsize; // number of elements in the dictionary
```

```
        void create();
        void destroy(); // delete all the element of the dictionary
};

template<class K, class E>
ostream& operator <<(ostream& out, Overflow_list<K, E>& dictionary)
{
        if (!dictionary.empty())
        {
                vector<K> keys = dictionary.keyArray();
                for (typename vector<K>::iterator it = keys.begin(); it != keys.end(); it++)
                {
                        out << *(dictionary.find(*it));
                        out << endl;
                }
        }
        else
                out << "Empty Dictionary" << endl;

        out << endl;
        return out;
}

template<class K, class E>
Overflow_list<K, E>::Overflow_list()
{
        lists = 0;
        MAXSIZE = 97; // default size (prime number is better)
        dsize = 0;
        this->create();
}

template<class K, class E>
Overflow_list<K, E>::Overflow_list(int size)
{
        lists = 0;
        MAXSIZE = size;
        dsize = 0;
        this->create();
}

template<class K, class E>
Overflow_list<K, E>::Overflow_list(const Overflow_list<K, E>& dic2)
{
        *this = dic2;
}

template<class K, class E>
Overflow_list<K, E>::~Overflow_list()
{
        this->destroy();
}

template<class K, class E>
Overflow_list<K, E>& Overflow_list<K, E>::operator =(const Overflow_list<K, E>& dic2)
{
        if (this != &dic2)
        {
                this->destroy();
                this->MAXSIZE = dic2.MAXSIZE;
                this->create();
```

```
		vector<Element<K, E> > pairs = dic2.pairArray();
		while (!pairs.empty())
		{
			this->insert(pairs.back());
			pairs.pop_back();
		}
	}

	return *this;
}

template<class K, class E>
bool Overflow_list<K, E>::operator ==(const Overflow_list<K, E>& d) const
{
	if (this->dsize != d.dsize)
		return false;

	if (this->hasSubset(d) && d.hasSubset(*this))
		return true;

	return false;
}

template<class K, class E>
bool Overflow_list<K, E>::operator !=(const Overflow_list<K, E>& d) const
{
	return (!(*this == d));
}

template<class K, class E>
void Overflow_list<K, E>::create()
{
	lists = new List_pointer<Element<K, E>*> [MAXSIZE];
}

template<class K, class E>
bool Overflow_list<K, E>::empty() const
{
	return (dsize == 0);
}

template<class K, class E>
Element<K, E>* Overflow_list<K, E>::find(const key& the_key) const
{
	int b = getBucket(the_key);

	for (typename List_pointer<Element<K, E>*>::position p = lists[b].begin(); !lists[b].end(p);
			p = lists[b].next(p))
		if ((lists[b].read(p))->getKey() == the_key)
			return lists[b].read(p);

	return 0;
}


template<class K, class E>
void Overflow_list<K, E>::insert(const Element<K, E> the_pair)
{
	int b = getBucket(the_pair.getKey());

	if (!this->find(the_pair))
```

```
	{
		Element<K, E>* temp = new Element<K, E>(the_pair);
		lists[b].push_back(temp);
		dsize++;
	}
	else
		this->modify(the_pair.getKey(), the_pair.getItem());
}

template<class K, class E>
void Overflow_list<K, E>::erase(const key & k)
{
	if (this->find(k) == 0)
		throw std::logic_error("Overflow_list (exception) - Unable to erase (key not 
present)");

	int b = this->getBucket(k);
	lists[b].eraseVal(this->find(k));
	dsize--;
}

template<class K, class E>
void Overflow_list<K, E>::destroy()
{
	for (int i = 0; i < MAXSIZE; i++)
		while(!lists[i].empty())
		{
			delete lists[i].read(lists[i].begin());
			lists[i].pop_front();
		}

	delete[] lists;
	lists = 0;
	dsize = 0;
}

template<class K, class E>
void Overflow_list<K, E>::modify(const key& k, const element& e)
{
	Element<K, E>* b = this->find(k);
	if (b == 0)
		throw std::logic_error("Overflow_list (exception) - Unable to modify (key not 
present)");
	b->setItem(e);
}

template<class K, class E>
int Overflow_list<K, E>::size() const
{
	return dsize;
}

template<class K, class E>
int Overflow_list<K, E>::getBucket(const key& the_key) const
{
	int i = (int) hashm(the_key) % MAXSIZE; // i = home bucket

	if (i < 0 || i >= MAXSIZE)
		i = 0;

	return i;
}
```

```
template<class K, class E>
vector<K> Overflow_list<K, E>::keyArray() const
{
	vector<K> keys;

	if (!this->empty())
		for (int i = 0; i < MAXSIZE; i++)
			if (!lists[i].empty())
				for (typename List_pointer<Element<K, E>*>::position p = 
lists[i].begin(); !lists[i].end(p);
						p = lists[i].next(p))
					keys.push_back((lists[i].read(p))->getKey());

	return keys;
}

template<class K, class E>
vector<E> Overflow_list<K, E>::itemArray() const
{
	vector<E> items;

	if (!this->empty())
		for (int i = 0; i < MAXSIZE; i++)
			if (!lists[i].empty())
				for (typename List_pointer<Element<K, E>*>::position p = 
lists[i].begin(); !lists[i].end(p);
						p = lists[i].next(p))
					items.push_back((lists[i].read(p))->getItem());

	return items;
}

template<class K, class E>
vector<Element<K, E> > Overflow_list<K, E>::pairArray() const
{
	vector<Element<K, E> > pairs;

	if (!this->empty())
		for (int i = 0; i < MAXSIZE; i++)
			if (!lists[i].empty())
				for (typename List_pointer<Element<K, E>*>::position p = 
lists[i].begin(); !lists[i].end(p);
						p = lists[i].next(p))
					pairs.push_back(*(lists[i].read(p)));

	return pairs;
}

template<class K, class E>
bool Overflow_list<K, E>::containsValue(const element& el) const
{
	if (this->empty())
		return false;

	vector<element> elements = this->itemArray();
	for (typename vector<element>::iterator it = elements.begin(); it != elements.end(); it++)
		if (*it == el)
			return true;

	return false;
}
```

```
template<class K, class E>
void Overflow_list<K, E>::join(const Overflow_list<K, E>& d2)
{
	if (!d2.empty())
	{
		vector<Element<K, E> > pairs = d2.pairArray();
		for (typename vector<Element<K, E> >::iterator it = pairs.begin(); it != pairs.end();
it++)
			this->insert(*it);
	}
}

template<class K, class E>
bool Overflow_list<K, E>::hasSubset(const Overflow_list<K, E>& d2) const
{
	if ((this->empty() && d2.empty()) || d2.empty())
		return true;

	if (this->empty())
		return false;

	vector<Element<K, E> > pairs = d2.pairArray();
	for (typename vector<Element<K, E> >::iterator it = pairs.begin(); it != pairs.end(); it++)
		if (!this->find(*it))
			return false;

	return true;
}

template<class K, class E>
bool Overflow_list<K, E>::find(const Element<K, E> el) const
{
	vector<Element<K, E> > pairs = this->pairArray();

	if (std::find(pairs.begin(), pairs.end(), el) != pairs.end())
		return true;

	return false;
}

#endif //_OVERFLOW_LIST_H
```

Dizionari – Realizzazione con hash table

```
#ifndef HASH_TABLE_H
#define HASH_TABLE_H

#include <string>
#include <vector>
#include <iostream>

#include "dictionary.h"
#include "hash.h"

using std::string;
using std::vector;
using std::cout;
using std::endl;

//Keys are unique. Inserting two times the same keys while cause overwrite of the label
//There is always at least one empty bucket in every moment in the dictionary
//Dictionary is never full at 70% or more
//Tombstones are used for erased elements
//Dictionary is never full at 20% or more of tombstones

template<class K, class E>
class Hash_table: public Dictionary<K, E>
{
public:

	typedef typename Dictionary<K, E>::key key;
	typedef typename Dictionary<K, E>::item element;

	Hash_table();
	Hash_table(int); // size
	Hash_table(const Hash_table<K, E>&);
	~Hash_table();

	Hash_table<K, E>& operator =(const Hash_table<K, E>&); // doesn't copy tombstones
	bool operator ==(const Hash_table<K, E>&) const;
	bool operator !=(const Hash_table<K, E>&) const;

	bool empty() const;
	Element<K, E>* find(const key&) const;
	void insert(const Element<K, E>);
	void erase(const key& k);
	void modify(const key& k, const element& e);

	int size() const;
	int getBucket(const key&) const; // find the home bucket of the tombstone
	int availableBucket(const key&) const; // the same as getBucket, but it doesn't skip 
tombstone
	vector<K> keyArray() const; // return a vector with the keys of the elements in the 
dictionary
	vector<E> itemArray() const; // return a vector of the elements in the dictionary
	vector<Element<K, E> > pairArray() const; // return a vector of pairs (element-key) in the 
dictionary
	bool containsValue(const element&) const; // true if dictionary has the element
	void join(const Hash_table<K, E>&); // add in the implicit dictionary all the elements of the 
parameter
	bool hasSubset(const Hash_table<K, E>&) const; // true if all the elements in the parameter 
are present in the implicit dictionary
```

```
	bool find(const Element<K, E>) const; // true if the element is present in the dictionary

private:
	Element<K, E>** table; // the hash table
	Hash<K> hashm; // maps type K to non-negative integer
	int dsize; // number of pairs in dictionary
	int MAXSIZE; // hash function divisor
	Element<K, E>* tombstone; // tombstone
	int tombCounter; // number of tombstones

	void create();
	void destroy(); // delete all the element of the dictionary
	void fixTable(); // delete all the tombstones from the dictionary
	void dictionaryDoubling();
};

template<class K, class E>
ostream& operator <<(ostream& out, Hash_table<K, E>& dictionary)
{
	if (!dictionary.empty())
	{
		vector<K> keys = dictionary.keyArray();
		for (typename vector<K>::iterator it = keys.begin(); it != keys.end(); it++)
		{
			out << *(dictionary.find(*it));
			out << endl;
		}
	}
	else
		out << "Empty Dictionary" << endl;

	out << endl;
	return out;
}

template<class K, class E>
Hash_table<K, E>::Hash_table()
{
	MAXSIZE = 97; // default (prime number is better)
	dsize = 0;
	table = 0;
	tombstone = 0;
	tombCounter = 0;
	this->create();
}

template<class K, class E>
Hash_table<K, E>::Hash_table(int size)
{
	MAXSIZE = size;
	dsize = 0;
	table = 0;
	tombstone = 0;
	tombCounter = 0;
	this->create();
}

template<class K, class E>
Hash_table<K, E>::Hash_table(const Hash_table<K, E>& dic2)
{
	*this = dic2;
}
```

```
template<class K, class E>
Hash_table<K, E>::~Hash_table()
{
	this->destroy();
	MAXSIZE = 0;
	dsize = 0;
	tombstone = 0;
	tombCounter = 0;
}

template<class K, class E>
Hash_table<K, E>& Hash_table<K, E>::operator =(const Hash_table<K, E>& dic2)
{
	if (this != &dic2)
	{
		this->destroy();
		this->MAXSIZE = dic2.MAXSIZE;
		this->create();

		for (int i = 0; i < dic2.MAXSIZE; i++)
			if (dic2.table[i] != 0 && dic2.table[i] != dic2.tombstone) // erases the
tombstones
				this->insert(*(dic2.table[i]));
	}
	return *this;
}

template<class K, class E>
bool Hash_table<K, E>::operator ==(const Hash_table<K, E>& t) const
{
	if (this->dsize != t.dsize)
		return false;

	if (this->hasSubset(t) && t.hasSubset(*this))
		return true;

	return false;
}

template<class K, class E>
bool Hash_table<K, E>::operator !=(const Hash_table<K, E>& t) const
{
	return (!(*this == t));
}

template<class K, class E>
void Hash_table<K, E>::create()
{
	table = new Element<K, E>*[MAXSIZE];
	for (int i = 0; i < MAXSIZE; i++)
		table[i] = 0;

	tombstone = new Element<K, E>;
	tombCounter = 0;
}

template<class K, class E>
bool Hash_table<K, E>::empty() const
{
	return (dsize == 0);
}
```

```
template<class K, class E>
Element<K, E>* Hash_table<K, E>::find(const key& the_key) const
{
	int b = getBucket(the_key);

	if (table[b] == 0)
		return 0;

	return table[b];
}

template<class K, class E>
void Hash_table<K, E>::insert(const Element<K, E> the_pair)
{
	if (((10 * (dsize + 1)) / 7) >= MAXSIZE) // if the dictionary will be full at 70% (at least)
with the element you are going to add
		this->dictionaryDoubling();

	int b = availableBucket(the_pair.getKey());

	if (table[b] == 0)
	{
		table[b] = new Element<K, E>(the_pair);
		dsize++;
	}
	else if (table[b] == tombstone)
	{
		table[b] = new Element<K, E>(the_pair);
		dsize++;
		tombCounter--;
	}
	else
	{
		if (table[b]->getKey() == the_pair.getKey())
			table[b]->setItem(the_pair.getItem());
	}
}

template<class K, class E>
void Hash_table<K, E>::erase(const key & k)
{
	int bucket = this->getBucket(k);

	if (table[bucket] == 0)
		throw std::logic_error("Hash_table (exception) - Unable to erase (key not present)");

	delete table[bucket];
	table[bucket] = tombstone;
	dsize--;
	tombCounter++;

	if ((tombCounter * 5) >= MAXSIZE) // if the dictionary is full at 20% (at least) of
tombstones
		this->fixTable();
}

template<class K, class E>
void Hash_table<K, E>::destroy()
{
	for (int i = 0; i < MAXSIZE; i++)
		if (table[i] == tombstone)
			table[i] = 0;
```

```
	delete tombstone;
	tombstone = 0;

	for (int i = 0; i < MAXSIZE; i++)
		if (table[i] != 0)
		{
			delete table[i];
			table[i] = 0;
		}

	delete[] table;
	table = 0;

	dsize = 0;
}

template<class K, class E>
void Hash_table<K, E>::modify(const key& k, const element& e)
{
	Element<K, E>* b = this->find(k);

	if (b == 0)
		throw std::logic_error("Hash_table (exception) - Unable to modify element (key not
present)");
	else
		b->setItem(e);
}

template<class K, class E>
int Hash_table<K, E>::size() const
{
	return dsize;
}

template<class K, class E>
int Hash_table<K, E>::getBucket(const key& the_key) const
{
	int i = (int) hashm(the_key) % MAXSIZE; // i = home bucket
	if (i < 0 || i >= MAXSIZE)
		i = 0;

	int j = i;
	do
	{
		if (table[j] != tombstone && (table[j] == 0 || table[j]->getKey() == the_key)) // skip
tombstones
			return j;
		j = (j + 1) % MAXSIZE; // the next bucket
	} while (j != i);

	throw std::logic_error("Hash_table (exception) - Unable to get bucket");
}

template<class K, class E>
int Hash_table<K, E>::availableBucket(const key& the_key) const
{
	int i = (int) hashm(the_key) % MAXSIZE; // i = home bucket
	if (i < 0 || i >= MAXSIZE)
		i = 0;

	int j = i;
```

```
	do
	{
		if (table[j] == tombstone || table[j] == 0 || table[j]->getKey() == the_key)
			return j;
		j = (j + 1) % MAXSIZE; // the next bucket
	} while (j != i);

	throw std::logic_error("Hash_table (exception) - Unable to get bucket");
}

template<class K, class E>
vector<K> Hash_table<K, E>::keyArray() const
{
	vector<K> keys;

	if (!this->empty())
		for (int i = 0; i < MAXSIZE; i++)
			if (table[i] != 0 && table[i] != tombstone)
				keys.push_back(table[i]->getKey());

	return keys;
}

template<class K, class E>
vector<E> Hash_table<K, E>::itemArray() const
{
	vector<E> items;

	for (int i = 0; i < MAXSIZE; i++)
		if (table[i] != 0 && table[i] != tombstone)
			items.push_back(table[i]->getItem());

	return items;

}

template<class K, class E>
vector<Element<K, E> > Hash_table<K, E>::pairArray() const
{
	vector<Element<K, E> > pairs;

	if (!this->empty())
		for (int i = 0; i < this->MAXSIZE; i++)
			if (this->table[i] != 0 && table[i] != tombstone)
				pairs.push_back(*this->table[i]);

	return pairs;
}

template<class K, class E>
bool Hash_table<K, E>::containsValue(const element& el) const
{
	if (this->empty())
		return false;

	vector<element> elements = this->itemArray();
	for (typename vector<element>::iterator it = elements.begin(); it != elements.end(); it++)
		if (*it == el)
			return true;

	return false;
}
```

```
template<class K, class E>
void Hash_table<K, E>::join(const Hash_table<K, E>& ht2)
{
	if (!ht2.empty())
		for (int i = 0; i < ht2.MAXSIZE; i++)
			if (ht2.table[i] != 0 && ht2.table[i] != tombstone)
				this->insert(*ht2.table[i]);
}

template<class K, class E>
bool Hash_table<K, E>::hasSubset(const Hash_table<K, E>& ht2) const
{
	if ((this->empty() && ht2.empty()) || ht2.empty())
		return true;

	if (this->empty())
		return false;

	vector<Element<K, E> > pairs = ht2.pairArray();
	for (typename vector<Element<K, E> >::iterator it = pairs.begin(); it != pairs.end(); it++)
		if (!this->find(*it))
			return false;

	return true;
}

template<class K, class E>
bool Hash_table<K, E>::find(const Element<K, E> el) const
{
	for (int i = 0; i < MAXSIZE; i++)
		if (table[i] != 0 && table[i] != tombstone && *table[i] == el)
			return true;

	return false;
}

template<class K, class E>
void Hash_table<K, E>::dictionaryDoubling()
{
	vector<Element<K, E> > elements;

	for (int i = 0; i < MAXSIZE; i++)
		if (table[i] != 0 && table[i] != tombstone)
			elements.push_back(*table[i]);

	this->destroy();
	MAXSIZE = MAXSIZE * 2;
	this->create();
	for (typename vector<Element<K, E> >::iterator it = elements.begin(); it != elements.end();
it++)
		this->insert(*it);
}

template<class K, class E>
void Hash_table<K, E>::fixTable()
{
	vector<Element<K, E> > elements;

	for (int i = 0; i < MAXSIZE; i++)
		if (table[i] != 0 && table[i] != tombstone)
			elements.push_back(*table[i]);
```

```
        this->destroy();
        this->create();
        for (typename vector<Element<K, E> >::iterator it = elements.begin(); it != elements.end();
it++)
                this->insert(*it);
}

#endif //HASH_TABLE_H
```

Dizionari
File: “hash.h”

```
#ifndef _HASHFUN_H
#define _HASHFUN_H

#include <string>
using std::string;

template<class T>
class Hash
{
public:

	typedef T type;

	unsigned int operator()(const type the_key) const
	{
		unsigned long hash_value = 0;

		hash_value = (unsigned int) sizeof(the_key);
		return (unsigned int) (hash_value);
	}
};

template<>
class Hash<string>
{
public:
	unsigned int operator()(const string str) const
	{
		// a bitwise hash function written by Justin Sobel
		unsigned int b = 378551;
		unsigned int a = 63689;
		unsigned int hash = 0;

		for (std::size_t i = 0; i < str.length(); i++)
		{
			hash = hash * a + str[i];
			a = a * b;
		}

		return hash;
	}
};

template<>
class Hash<int>
{
public:
	unsigned int operator()(const int key) const
	{
		// middle-square method of hashing
		unsigned int const exp = 10; // Mb = 1024 = 2^10
		unsigned int const k = 8 * sizeof(unsigned int);

		return ((key * key) >> (k - exp));
	}
};
```

```
template<>
class Hash<char>
{
public:
	unsigned int operator()(const char key) const
	{
		// middle-square method of hashing
		unsigned int const exp = 10; // Mb = 1024 = 2^10
		unsigned int const k = 8 * sizeof(unsigned int);
		unsigned int elem = (unsigned int) key;

		return ((elem * elem) >> (k - exp));
	}
};

#endif // _HASHFUN_H
```

Dizionari – Tester

```
#include "overflow_list.h"

int main()
{
	Overflow_list<string, int> dic(1), dic2(1), dic3;

	dic.insert(Element<string, int>("BABBAGE", 2));
	dic.insert(Element<string, int>("TURING", 3));
	dic.insert(Element<string, int>("PASCAL", 5));
	dic.insert(Element<string, int>("ADA", 20));

	cout << "dic:" << endl << dic;

	dic.erase("BABBAGE");
	dic.erase("ADA");

	cout << "dic:" << endl << dic;

	dic2.insert(Element<string, int>("Stringa1", 20));
	dic2.insert(Element<string, int>("Stringa2", 35));

	cout << "dic2:" << endl << dic2;

	dic2 = dic;
	cout << "dic2:" << endl << dic2;

	if (dic == dic2)
		cout << "dic = dic2" << endl;
	else
		cout << "dic != dic2" << endl;

	dic2.modify("PASCAL", 10);

	cout << endl << "After changing dic2: ";

	if (dic == dic2)
		cout << "dic = dic2" << endl;
	else
		cout << "dic != dic2" << endl;

	cout << endl;

	dic3.insert(Element<string, int>("Stringa1", 20));
	dic3.insert(Element<string, int>("Stringa2", 35));
	cout << "dic3:" << endl << dic3;

	dic.join(dic3);
	cout << "dic:" << endl << dic;

}
```

Dizionari – Output Tester

```
dic:
Key: BABBAGE // Value: 2
Key: TURING // Value: 3
Key: PASCAL // Value: 5
Key: ADA // Value: 20

dic:
Key: TURING // Value: 3
Key: PASCAL // Value: 5

dic2:
Key: Stringa1 // Value: 20
Key: Stringa2 // Value: 35

dic2:
Key: PASCAL // Value: 5
Key: TURING // Value: 3

dic = dic2

After changing dic2: dic != dic2

dic3:
Key: Stringa1 // Value: 20
Key: Stringa2 // Value: 35

dic:
Key: TURING // Value: 3
Key: PASCAL // Value: 5
Key: Stringa1 // Value: 20
Key: Stringa2 // Value: 35
```

Grafi – Classe astratta

```
#ifndef _GRAPH_H_
#define _GRAPH_H_

#include "edge.h"
#include "set_pointer.h"
#include "mfset_set.h"

#include <iostream>
#include <list>
#include <queue>
#include <vector>
#include <stdexcept>
#include <algorithm>
#include <climits>

using std::list;
using std::queue;
using std::vector;
using std::endl;
using std::ostream;
using std::cout;

template<class N, class L, class W>
class Graph
{
public:
	typedef N node;
	typedef L label;
	typedef W weight;

	virtual ~Graph()
	{
	}

	virtual void create() = 0;
	virtual bool empty() const = 0;
	virtual void insNode(node&) = 0;
	virtual void insEdge(const node, const node) = 0;
	virtual bool isNode(const node) const = 0;
	virtual bool isEdge(const node, const node) const = 0;
	virtual void eraseNode(const node) = 0;
	virtual void eraseEdge(const node, const node) = 0;
	virtual list<node> adjacents(const node) const = 0;
	virtual list<node> nodesList() const = 0;
	virtual label getLabel(const node) const = 0;
	virtual void setLabel(const node, const label) = 0;
	virtual weight getWeight(const node, const node) const = 0;
	virtual void setWeight(const node, const node, const weight) = 0;

	int numNodes() const; // number of the nodes in the graph
	int numEdges() const; // number of edges in the graph
	bool completeGraph() const; // true if every node is directly linked with all the others (in 
both directions)
	bool connectedGraph() const; // true if for every pair of node (u,j) there is a path from u 
to j or from j to u
	bool stronglyConnectedGraph() const; // true if for every pair of node (u,j) there is a path 
from u to j and from j to u
	int inGrade(const node) const; // number of edges starting from the node
```

```
	int outGrade(const node) const; // number of edges ending in the node
	bool linked(const node) const; // true if the node has at least one edge (from or to the 
node)
	list<node> daisyChained(const node) const; // list of the nodes reachable with a path 
starting from the node
	bool isLoopNode(const node) const; // true if the node is present in its list of reachable 
nodes
	bool acyclicGraph() const; // true if every node isn't a loopNode
	list<node> adjacentsKLevel(const node, const int) const; // list of nodes reachable with K 
edges from the node
	bool isPath(const node, const node) const; // true if there is a path from the starting node 
to the ending
	bool isNewPath(const node, const node, const list<node>) const; // true if there is a path 
from the starting node to the ending that avoids nodes in the list
	list<node> getPath(const node, const node) const; // return a list of nodes: together they 
are a path between the two nodes
	void DFS(const node) const;
	void BFS(const node) const;
	vector<edge<N, W> > shortestPath(const node) const; // return a vector of edges by which is 
possible to create the shortest paths
	list<node> getShortestPath(const node, const node) const; // return a list of nodes: together 
they are the best path between the two nodes
	void clearEdges(); // erase all the edges of the graph
	list<edge<N, W> > edgesList() const; // return a list of all the edges of the graph
	Set_pointer<edge<N,W> > MST(); // return the set of edges of the MST
	void convertToMST(); // convert a graph to its MST

private:
	void DFS(const node, list<node>&) const;
	weight pathDistance(const node, const vector<edge<N, W> >) const; // return the current 
shortest distance to the node
	void setFatherPath(const node, const node, vector<edge<N, W> >&) const; // set the second 
node as the "father" of the first
	void setWeightPath(const node, const weight, vector<edge<N, W> >&) const; // set the weight 
in the edge ending in the node
};

template<class N, class L, class W>
ostream& operator <<(ostream& out, const Graph<N, L, W>& graph)
{
	if (!graph.empty())
	{
		out << "Label [ID]: Adjacent1 [ID1] (Weight1) // Adjacent2 [ID2] (Weight2) // ... //" 
<< endl;

		list<typename Graph<N, L, W>::node> temp = graph.nodesList();
		for (typename list<typename Graph<N, L, W>::node>::const_iterator it = temp.begin(); 
it != temp.end();
				it++)
		{
			out << graph.getLabel(*it);
			out << (*it) << ": ";
			list<typename Graph<N, L, W>::node> adj = graph.adjacents((*it));
			for (typename list<typename Graph<N, L, W>::node>::const_iterator it2 = 
adj.begin(); it2 != adj.end();
					it2++)
				out << graph.getLabel(*it2) << (*it2) << " (" << graph.getWeight((*it), 
(*it2)) << ") // ";
			out << endl;
		}

	}
```

```
	else
		out << "Empty Graph";

	out << endl;
	return out;
}

template<class N, class L, class W>
int Graph<N, L, W>::numNodes() const
{
	return (this->nodesList()).size();
}

template<class N, class L, class W>
int Graph<N, L, W>::numEdges() const
{
	if (this->empty())
		return 0;

	int counter = 0;
	list<node> nodes = this->nodesList();
	list<node> temp = nodes;
	for (typename list<node>::iterator it = nodes.begin(); it != nodes.end(); it++)
		for (typename list<node>::iterator it2 = temp.begin(); it2 != temp.end(); it2++)
			if (this->isEdge(*it, *it2))
				counter++;
	return counter;
}

template<class N, class L, class W>
bool Graph<N, L, W>::completeGraph() const
{
	// in a complete direct graph with N nodes the number of edges is N*(N-1)
	int num = this->numNodes();

	return (this->numEdges() == (num * (num - 1)));
}

template<class N, class L, class W>
bool Graph<N, L, W>::connectedGraph() const
{
	list<node> nodes = this->nodesList();
	for (typename list<node>::iterator it = nodes.begin(); it != nodes.end(); it++)
		for (typename list<node>::iterator it2 = nodes.begin(); it2 != nodes.end(); it2++)
			if ((*it != *it2) && !this->isPath(*it, *it2) && !this->isPath(*it2, *it))
				return false;

	return true;

}

template<class N, class L, class W>
bool Graph<N, L, W>::stronglyConnectedGraph() const
{
	list<node> nodes = this->nodesList();
	for (typename list<node>::iterator it = nodes.begin(); it != nodes.end(); it++)
		for (typename list<node>::iterator it2 = nodes.begin(); it2 != nodes.end(); it2++)
			if ((*it != *it2) && !this->isPath(*it, *it2))
				return false;

	return true;
}
```

```
template<class N, class L, class W>
int Graph<N, L, W>::inGrade(const node curr) const
{
	if (!this->isNode(curr))
		throw std::logic_error("Graph (exception)");

	int counter = 0;
	list<node> temp = this->nodesList();
	for (typename list<node>::iterator it = temp.begin(); it != temp.end(); it++)
		if (this->isEdge((*it), curr))
			counter++;

	return counter;
}

template<class N, class L, class W>
int Graph<N, L, W>::outGrade(const node curr) const
{
	if (!this->isNode(curr))
		throw std::logic_error("Graph (exception)");

	return this->adjacents(curr).size();
}

template<class N, class L, class W>
bool Graph<N, L, W>::linked(const node currNode) const
{
	if (!this->isNode(currNode))
		throw std::logic_error("Graph (exception) - Unable to check if linked (node not
present)");

	if (this->outGrade(currNode) > 0)
		return true;

	if (this->inGrade(currNode) > 0)
		return true;

	return false;
}

template<class N, class L, class W>
list<typename Graph<N, L, W>::node> Graph<N, L, W>::daisyChained(const node n) const
{
	list<node> nodes = this->adjacents(n);
	int size = (this->numNodes() - 2); // we have already put the adjacent nodes (new max length
of the path is size-2)
	int counter = nodes.size(); // number of elements added
	typename list<node>::iterator itStart = nodes.begin();

	for (int i = 0; (i < size && counter != 0); i++) // iterating "size" times we explore all the
graph (if we don't add any elements, the search is completed)
	{
		int temp = nodes.size(); // size of the list before adding elements (necessary to know
the new starting element)
		int newElementsNumber = counter; // number of new elements added to the list (we'll
repeat "newElementsNumber" the following for() )
		counter = 0;

		for (int j = 0; j < newElementsNumber; j++) // it will repeat the for() only for new
elements added to the list
		{
			list<node> newNodes = this->adjacents(*itStart);
```

```
                        for (typename list<node>::iterator it2 = newNodes.begin(); it2 !=
newNodes.end(); it2++)
                                if (std::find(nodes.begin(), nodes.end(), (*it2)) == nodes.end()) // if
the node isn't already present in the list
                                {
                                        nodes.push_back(*it2); // add the node
                                        counter++; // increase the number of elements added
                                }

                        itStart++; // we'll get the next element in the list
                        newNodes.clear();
                }

                // we need to set itStart to the first element added
                itStart = nodes.begin();
                for (int j = 0; j < temp; j++) // temp stores the size of the list before adding new
elements
                        itStart++;
                // itStart will point to the first new element added
        }

        return nodes;
}

template<class N, class L, class W>
bool Graph<N, L, W>::isLoopNode(const node n) const
{
        list<node> temp = this->daisyChained(n);
        if (std::find(temp.begin(), temp.end(), n) != temp.end())
                return true;

        return false;
}

template<class N, class L, class W>
bool Graph<N, L, W>::acyclicGraph() const
{
        list<node> nodes = this->nodesList();
        int counter = 0;

        for (typename list<node>::iterator it = nodes.begin(); it != nodes.end(); it++)
                counter += this->isLoopNode(*it);

        if (counter == 0)
                return true;

        return false;
}

template<class N, class L, class W>
list<typename Graph<N, L, W>::node> Graph<N, L, W>::adjacentsKLevel(const node n, const int k) const
{
        if (k < 0)
                throw std::logic_error("Graph (exception) - Unable to get adjacent nodes (negative
distance)");

        if (k == 0) // empty list
        {
                list<node> nodes;
                return nodes;
        }
```

```
	list<node> nodes = this->adjacents(n);
	list<node> adjNodesK;
	if (k == 1)
		adjNodesK = nodes;
	int size = (k - 1); // we have already put the adjacent nodes (new max length of the path is
size-2)
	int counter = nodes.size(); // number of elements added
	typename list<node>::iterator itStart = nodes.begin();

	for (int i = 0; (i < size && counter != 0); i++) // iterating "size" times we explore all the
graph
	{
		int temp = nodes.size(); // size of the list before adding elements (necessary to know
the new starting element)
		int newElementsNumber = counter; // number of new elements added to the list (we'll
repeat "newElementsNumber" the following for() )
		counter = 0;

		for (int j = 0; j < newElementsNumber; j++) // it will repeat the for() only for new
elements added to the list
		{
			list<node> newNodes = this->adjacents(*itStart);
			for (typename list<node>::iterator it2 = newNodes.begin(); it2 !=
newNodes.end(); it2++)
			{
				if (std::find(nodes.begin(), nodes.end(), (*it2)) == nodes.end()) // if
the node isn't already present in the list
				{
					nodes.push_back(*it2); // add the node
					counter++; // increase the number of elements added
				}
				if (i == (k - 2)) // we are on K level
					if (std::find(adjNodesK.begin(), adjNodesK.end(), (*it2)) ==
adjNodesK.end()) // if the node isn't already present in the list
						adjNodesK.push_back(*it2);
			}

			itStart++; // we'll get the next element in the list
			newNodes.clear();
		}

		// we need to set itStart to the first element added
		itStart = nodes.begin();
		for (int j = 0; j < temp; j++) // temp stores the size of the list before adding new
elements
			itStart++;
		// itStart will point to the first new element added
	}

	return adjNodesK;

}

template<class N, class L, class W>
bool Graph<N, L, W>::isPath(const node start, const node end) const
{
	list<node> nodes = this->adjacents(start);

	if (std::find(nodes.begin(), nodes.end(), end) != nodes.end())
		return true;
```

```
	int size = (this->numNodes() - 2); // we have already put the adjacent nodes (new max length
of the path is size-2)
	int counter = nodes.size(); // number of elements added
	typename list<node>::iterator itStart = nodes.begin();

	for (int i = 0; (i < size && counter != 0); i++) // iterating "size" times we explore all the
graph (if we don't add any elements, the search is completed)
	{
		int temp = nodes.size(); // size of the list before adding elements (necessary to know
the new starting element)
		int newElementsNumber = counter; // number of new elements added to the list (we'll
repeat "newElementsNumber" the following for() )
		counter = 0;

		for (int j = 0; j < newElementsNumber; j++) // it will repeat the for() only for new
elements added to the list
		{
			list<node> newNodes = this->adjacents(*itStart);
			for (typename list<node>::iterator it2 = newNodes.begin(); it2 !=
newNodes.end(); it2++)
			{
				if ((*it2) == end)
					return true;

				if (std::find(nodes.begin(), nodes.end(), (*it2)) == nodes.end()) // if
the node isn't already present in the list
				{
					nodes.push_back(*it2); // add the node
					counter++; // increase the number of elements added
				}
			}

			itStart++; // we'll get the next element in the list
			newNodes.clear();
		}

		// we need to set itStart to the first element added
		itStart = nodes.begin();
		for (int j = 0; j < temp; j++) // temp stores the size of the list before adding new
elements
			itStart++;
		// itStart will point to the first new element added
	}

	return false;
}

template<class N, class L, class W>
bool Graph<N, L, W>::isNewPath(const node start, const node end, const list<node> holes) const
{
	if (std::find(holes.begin(), holes.end(), end) != holes.end())
		throw std::logic_error("Graph (exception) - Unable to get new path (end node is
forbidden)");

	list<node> nodes;
	list<node> adjStart = this->adjacents(start);
	for (typename list<node>::iterator it = adjStart.begin(); it != adjStart.end(); it++) // we
can start the algorithm only for valid nodes
		if (std::find(holes.begin(), holes.end(), *it) == holes.end()) // erase holes
			nodes.push_back(*it);

	if (std::find(nodes.begin(), nodes.end(), end) != nodes.end())
```

```
		return true;

	int size = (this->numNodes() - 2); // we have already put the adjacent nodes (new max length
of the path is size-2)
	int counter = nodes.size(); // number of elements added
	typename list<node>::iterator itStart = nodes.begin();

	for (int i = 0; (i < size && counter != 0); i++) // iterating "size" times we explore all the
graph (if we don't add any elements, the search is completed)
	{
		int temp = nodes.size(); // size of the list before adding elements (necessary to know
the new starting element)
		int newElementsNumber = counter; // number of new elements added to the list (we'll
repeat "newElementsNumber" the following for() )
		counter = 0;

		for (int j = 0; j < newElementsNumber; j++) // it will repeat the for() only for new
elements added to the list
		{
			list<node> newNodes;

			adjStart.clear();
			adjStart = this->adjacents(*itStart);
			for (typename list<node>::iterator it = adjStart.begin(); it != adjStart.end();
it++) // we can continue the algorithm only for valid nodes
				if (std::find(holes.begin(), holes.end(), *it) == holes.end()) // erase
holes
					newNodes.push_back(*it);

			for (typename list<node>::iterator it2 = newNodes.begin(); it2 !=
newNodes.end(); it2++)
			{
				if ((*it2) == end)
					return true;

				if (std::find(nodes.begin(), nodes.end(), (*it2)) == nodes.end()) // if
the node isn't already present in the list
				{
					nodes.push_back(*it2); // add the node
					counter++; // increase the number of elements added
				}
			}

			itStart++; // we'll get the next element in the list
			newNodes.clear();
		}

		// we need to set itStart to the first element added
		itStart = nodes.begin();
		for (int j = 0; j < temp; j++) // temp stores the size of the list before adding new
elements
			itStart++;
		// itStart will point to the first new element added
	}

	return false;
}

template<class N, class L, class W>
list<typename Graph<N, L, W>::node> Graph<N, L, W>::getPath(const node start, const node end) const
{
	if (!this->isPath(start, end))
```

```
            throw std::logic_error("Graph (exception) - Unable to get path (second node not
reachable)");

        int flag = true;
        int stop = false;
        list<node> path;
        list<node> visit;
        path.push_back(start);
        visit.push_back(start);

        while (flag)
        {
            list<node> temp = this->adjacents(path.back()); // check if end is present in the
adjacent nodes of the last one added
            if (std::find(temp.begin(), temp.end(), end) != temp.end())
            {
                path.push_back(end);
                flag = false; // search is over: path has been completed
            }
            else
            {
                for (typename list<node>::iterator it = temp.begin(); (it != temp.end() &&
!stop); it++)
                    if (this->isNewPath(*it, end, visit) && std::find(visit.begin(),
visit.end(), *it) == visit.end())
                    {
                        path.push_back(*it);
                        visit.push_back(*it);
                        stop = true;
                    }

                stop = false;
            }
        }

        return path;
}

template<class N, class L, class W>
void Graph<N, L, W>::DFS(const node n) const
{
        if (!this->isNode(n))
            throw std::logic_error("Graph (exception) - Unable to perform DFS (invalid node)");

        list<node> visited;

        cout << n << " ";

        visited.push_back(n);

        list<node> adjNodes = this->adjacents(n);
        for (typename list<node>::iterator it = adjNodes.begin(); it != adjNodes.end(); it++)
            if (std::find(visited.begin(), visited.end(), *it) == visited.end()) // if the node
hasn't been visited yet
                DFS(*it, visited);
}

template<class N, class L, class W>
void Graph<N, L, W>::DFS(const node n, list<node>& visited) const
{
        cout << n << " ";
```

```
	visited.push_back(n);

	list<node> adjNodes = this->adjacents(n);
	for (typename list<node>::iterator it = adjNodes.begin(); it != adjNodes.end(); it++)
		if (std::find(visited.begin(), visited.end(), *it) == visited.end()) // if the node
hasn't been visited yet
			DFS(*it, visited);
}

template<class N, class L, class W>
void Graph<N, L, W>::BFS(const node n) const
{
	queue<node> temp;
	list<node> visited;
	list<node> adjNodes;

	temp.push(n);
	while (!temp.empty())
	{
		node curr = temp.front();
		cout << curr << " ";
		temp.pop();
		visited.push_back(curr);

		adjNodes = this->adjacents(curr);
		for (typename list<node>::iterator it = adjNodes.begin(); it != adjNodes.end(); it++)
			if (std::find(visited.begin(), visited.end(), *it) == visited.end()) // if the
node hasn't been visited yet
				temp.push(*it);
		adjNodes.clear();

	}
}

template<class N, class L, class W>
vector<edge<N, W> > Graph<N, L, W>::shortestPath(const node n) const
{
	if (!this->isNode(n))
		throw std::logic_error("Graph (exception) - Unable to calculate shortest path (invalid
node)");

	if (!this->connectedGraph())
		throw std::logic_error("Graph (exception) - Unable to calculate shortest path (graph
not connected)");

	vector<edge<N, W> > paths;
	list<node> nodes = this->nodesList();
	Set_pointer<node> nodesSet;
	nodesSet.insert(n);
	node temp;

	for (typename list<node>::iterator it = nodes.begin(); it != nodes.end(); it++)
		if (n != *it)
			paths.push_back(edge<N, W>(n, *it, INT_MAX));
		else
			paths.push_back(edge<N, W>(n, *it, 0));

	while (!nodesSet.empty())
	{
		temp = nodesSet.pickAny();
		nodesSet.erase(temp);
		list<node> adjTemp = this->adjacents(temp);
```

```
		for (typename list<node>::iterator it = adjTemp.begin(); it != adjTemp.end(); it++)
			if ((this->pathDistance(temp, paths) + this->getWeight(temp, *it)) < this-
>pathDistance(*it, paths))
			{
				this->setFatherPath(*it, temp, paths);
				this->setWeightPath(*it, (this->pathDistance(temp, paths) + this-
>getWeight(temp, *it)), paths);
				if (!nodesSet.find(*it))
					nodesSet.insert(*it);
			}
	}

	return paths;
}

template<class N, class L, class W>
typename Graph<N, L, W>::weight Graph<N, L, W>::pathDistance(const node n, const vector<edge<N, W> >
paths) const
{
	weight currWeight;
	for (typename vector<edge<N, W> >::const_iterator it = paths.begin(); it != paths.end();
it++)
		if ((*it).endNode == n)
			currWeight = (*it).weight;

	return currWeight;
}

template<class N, class L, class W>
void Graph<N, L, W>::setFatherPath(const node son, const node father, vector<edge<N, W> >& paths)
const
{
	for (typename vector<edge<N, W> >::iterator it = paths.begin(); it != paths.end(); it++)
		if ((*it).endNode == son)
			(*it).startNode = father;
}

template<class N, class L, class W>
void Graph<N, L, W>::setWeightPath(const node n, const weight newWeight, vector<edge<N, W> >& paths)
const
{
	for (typename vector<edge<N, W> >::iterator it = paths.begin(); it != paths.end(); it++)
		if ((*it).endNode == n)
			(*it).weight = newWeight;
}

template<class N, class L, class W>
list<typename Graph<N, L, W>::node> Graph<N, L, W>::getShortestPath(const node start, const node
end) const
{
	if (!this->isNode(start) || !this->isNode(end))
		throw std::logic_error("Graph (exception) - Unable to calculate shortest path (invalid
node)");

	if (!this->isPath(start, end))
		throw std::logic_error("Graph (exception) - Unable to calculate shortest path (path
not present)");

	vector<edge<N, W> > minPaths = this->shortestPath(start);
	list<node> bestPath;
	bestPath.push_front(end);
```

```
	typename vector<edge<N, W> >::iterator it = minPaths.begin();
	node tempNode;

	while ((*it).endNode != end)
		it++; // set it on the edge ending in end

	while ((*it).startNode != start)
	{
		bestPath.push_front((*it).startNode);
		tempNode = (*it).startNode;
		it = minPaths.begin();
		while ((*it).endNode != tempNode)
			it++; // set it on the edge ending in the node that has just been added
	}

	bestPath.push_front(start);

	return bestPath;
}

template<class N, class L, class W>
list<edge<N, W> > Graph<N, L, W>::edgesList() const
{
	list<edge<N, W> > edgesList;

	list<node> list1 = this->nodesList();
	list<node> list2 = list1;

	for (typename list<node>::iterator it = list1.begin(); it != list1.end(); it++)
		for (typename list<node>::iterator it2 = list2.begin(); it2 != list2.end(); it2++)
			if (this->isEdge(*it, *it2))
				edgesList.push_back(edge<N, W>(*it, *it2, this->getWeight(*it, *it2)));

	return edgesList;
}

template<class N, class L, class W>
void Graph<N, L, W>::clearEdges()
{
	list<node> list1 = this->nodesList();
	list<node> list2 = list1;

	for (typename list<node>::iterator it = list1.begin(); it != list1.end(); it++)
		for (typename list<node>::iterator it2 = list2.begin(); it2 != list2.end(); it2++)
			if (this->isEdge(*it, *it2))
				this->eraseEdge(*it, *it2);
}

template<class N, class L, class W>
Set_pointer<edge<N,W> > Graph<N, L, W>::MST()
{
	if (!this->connectedGraph())
		throw std::logic_error("Graph (exception) - Unable to calculate MST (not connected
graph)");

	Set_pointer<edge<node, weight> > edges; // set of edges of the MST
	Set_pointer<node> nodes; // set with all the nodes of the graph
	list<node> nodesList = this->nodesList();

	while (!nodesList.empty()) // convert the list of nodes in a set of nodes
	{
		nodes.insert(nodesList.front());
```

```
		nodesList.pop_front();
	}

	MFSet_set<node> mfset(nodes); // create the MFSet with the set that has just been built

	list<edge<N, W> > edgesList = this->edgesList();
	edgesList.sort(); // sort the list of edges

	for (typename list<edge<N, W> >::iterator it = edgesList.begin(); it != edgesList.end();
it++) // for each edge
		if (mfset.find((*it).startNode) != mfset.find((*it).endNode)) // if nodes belong to
different components
		{
			mfset.merge((*it).startNode, (*it).endNode);
			edges.insert(*it);
			edges.insert(edge<N, W>((*it).endNode, (*it).startNode, (*it).weight)); //
insert the edge also in the opposite direction
		}

	return edges;
}

template<class N, class L, class W>
void Graph<N, L, W>::convertToMST()
{
	Set_pointer<edge<N,W> > edges = this->MST();

	this->clearEdges(); // remove edges of the graph

	while (!edges.empty()) // reinsert only the strictly necessary edges
	{
		edge<node, weight> temp = edges.pickAny();
		edges.erase(temp);
		this->insEdge(temp.startNode, temp.endNode);
		this->setWeight(temp.startNode, temp.endNode, temp.weight);
	}
}

#endif /* _GRAPH_H_ */
```

Grafi
File: “edge.h”

```
#ifndef _EDGE_H
#define _EDGE_H

#include <iostream>
using std::ostream;

template<class N, class W>
class edge
{
public:
	typedef N node;

	node startNode;
	node endNode;
	W weight;

	edge()
	{
		startNode = node(); //-1
		endNode = node(); //-1
		weight = W();
	}

	edge(const node start, const node end, const W newWeight)
	{
		startNode = start;
		endNode = end;
		weight = newWeight;
	}

	edge(const edge<N, W>& newEdge)
	{
		*this = newEdge;
	}
	edge<N, W>& operator =(const edge<N, W>& newEdge)
	{
		if (&newEdge != this) // avoid auto-assignment
		{
			this->startNode = newEdge.startNode;
			this->endNode = newEdge.endNode;
			this->weight = newEdge.weight;
		}
		return *this;
	}
	~edge()
	{
	}

	bool operator ==(const edge<N, W>& e2) const
	{
		if (this->startNode != e2.startNode)
			return false;

		if (this->endNode != e2.endNode)
			return false;

		if (this->weight != e2.weight)
```

```
                        return false;

                return true;
        }

        bool operator !=(const edge<N, W>& e2) const
        {
                return (!(*this == e2));
        }

        bool operator <(const edge<N, W>& e2) const
        {
                if (this->weight < e2.weight)
                        return true;

                return false;
        }

};

template<class N, class W>
ostream& operator <<(ostream& out, const edge<N, W>& edge)
{
        out << edge.startNode;
        out << " -> ";
        out << edge.endNode;
        out << ": ";
        out << edge.weight;
        out << std::endl;

        return out;

}

#endif // _EDGE_H
```

Grafi
File: “Node.h”

```
#ifndef _NODE_H
#define _NODE_H

class Node
{
public:
	Node()
	{
		ID = -1;
	}

	Node(const Node& node2)
	{
		*this = node2;
	}

	Node& operator =(const Node& node2)
	{
		if (&node2 != this) // avoid auto-assignment
			this->ID = node2.ID;
		return *this;
	}

	~Node(){ }

	bool operator ==(const Node& n2) const
	{
		return (this->ID == n2.ID);
	}

	bool operator !=(const Node& n2) const
	{
		return (!(*this == n2));
	}

	void setID(const int newID)
	{
		ID = newID;
	}

	int getID() const
	{
		return ID;
	}

private:
	int ID;
};

ostream& operator <<(ostream& out, const Node& node)
{
	out << " [ID ";
	out << node.getID();
	out << "]";
	return out;
}
#endif // _NODE_H
```

Grafi – Realizzazione con matrice di adiacenza

```
#ifndef _GRAPHADJMATRIX_H
#define _GRAPHADJMATRIX_H

#include "Graph.h"
#include "Node.h"
#include "Row.h"

#include <vector>
using std::vector;
using std::cout;

template<class V, class W>
class Graph_matrix: public Graph<Node, V, W>
{
public:
	typedef typename Graph<Node, V, W>::node node;
	typedef typename Graph<Node, V, W>::label label;
	typedef typename Graph<Node, V, W>::weight weight;

	Graph_matrix();
	Graph_matrix(const Graph_matrix<V, W>&);
	~Graph_matrix();

	Graph_matrix<V, W>& operator =(const Graph_matrix<V, W>&);

	void create();
	bool empty() const;
	void insNode(node&);
	void insEdge(const node, const node);
	bool isNode(const node) const;
	bool isEdge(const node, const node) const;
	void eraseNode(const node);
	void eraseEdge(const node, const node);
	list<node> adjacents(const node) const;
	list<node> nodesList() const;
	label getLabel(const node) const;
	void setLabel(const node, const label);
	weight getWeight(const node, const node) const;
	void setWeight(const node, const node, const weight);

private:
	vector<Row<V, W> > matrix;
	int currID;

	typename vector<Row<V, W> >::const_iterator getRow(const node) const;
	typename vector<Row<V, W> >::iterator getRow(const node);

};

template<class V, class W>
Graph_matrix<V, W>::Graph_matrix()
{
	currID = 0;
	this->create();
}
```

```
template<class V, class W>
Graph_matrix<V, W>::Graph_matrix(const Graph_matrix<V, W>& g2)
{
	*this = g2;
}

template<class V, class W>
Graph_matrix<V, W>::~Graph_matrix()
{
	while (!this->matrix.empty())
		this->matrix.pop_back();

	this->currID = 0;
}

template<class V, class W>
Graph_matrix<V, W>& Graph_matrix<V, W>::operator =(const Graph_matrix<V, W>& g2)
{
	if (&g2 != this) // avoid auto-assignment
	{
		while (!this->matrix.empty())
			this->matrix.pop_back();

		this->matrix = g2.matrix;
		this->currID = g2.currID;
	}
	return *this;
}

template<class V, class W>
void Graph_matrix<V, W>::create()
{
	currID = 0;
	for (int i = 0; i < (int) MAXNODES; i++)
		matrix.push_back(Row<V, W>(i));

}

template<class V, class W>
bool Graph_matrix<V, W>::empty() const
{
	return (currID == 0);
}

template<class V, class W>
void Graph_matrix<V, W>::insNode(node& newNode)
{
	if (currID >= (int) MAXNODES)
		throw std::logic_error("Graph_matrix (exception) - Unable to insert node (full
matrix)");

	if (newNode.getID() != -1) // the node belongs to a graph
		throw std::logic_error("Graph_matrix (exception) - Unable to insert node (already
used)");

	newNode.setID(this->currID);
	currID++;

	(*this->getRow(newNode)).flag = true;
}
```

```
template<class V, class W>
void Graph_matrix<V, W>::insEdge(const node start, const node end)
{
	if (!this->isNode(start) || !this->isNode(end))
		throw std::logic_error("Graph_matrix (exception) - Unable to insert edge (node not
present)");

	if (this->isEdge(start, end))
		throw std::logic_error("Graph_matrix (exception) - Unable to insert edge (edge already
present)");

	bool flag = true;
	for (typename vector<edgeRow<W> >::iterator it = (*(this->getRow(start))).adjacents.begin();
			(it != (*(this->getRow(start))).adjacents.end() && flag); it++)
		if ((*it).destID == end.getID())
		{
			(*it).flag = true;
			flag = false;
		}
}

template<class V, class W>
bool Graph_matrix<V, W>::isNode(const node currNode) const
{
	if (this->empty())
		return false;

	for (typename vector<Row<V, W> >::const_iterator it = this->matrix.begin(); it != this-
>matrix.end(); it++)
		if ((*it).nodeID == currNode.getID() && (*it).flag == true)
			return true;

	return false;
}

template<class V, class W>
bool Graph_matrix<V, W>::isEdge(const node start, const node end) const
{
	if (!this->isNode(start) || !this->isNode(end))
		throw std::logic_error("Graph_matrix (exception) - Unable to check edge (node not
present)");

	Row<V, W> currRow = *(this->getRow(start));

	for (typename vector<edgeRow<W> >::iterator it = currRow.adjacents.begin(); it !=
currRow.adjacents.end();
			it++)
		if ((*it).destID == end.getID() && ((*it).flag == true))
			return true;

	return false;
}

template<class V, class W>
void Graph_matrix<V, W>::eraseNode(const node currNode)
{
	if (!this->isNode(currNode))
		throw std::logic_error("Graph_matrix (exception) - Unable to erase node (node not
present)");

	if (this->linked(currNode))
```

```
		throw std::logic_error("Graph_matrix (exception) - Unable to erase node (node linked
to others)");

	(*this->getRow(currNode)).flag = true;
}

template<class V, class W>
void Graph_matrix<V, W>::eraseEdge(const node node1, const node node2)
{
	if (!this->isNode(node1) || !this->isNode(node2))
		throw std::logic_error("Graph_matrix (exception) - Unable to erase edge (node not
present)");
	if (!this->isEdge(node1, node2))
		throw std::logic_error("Graph_matrix (exception) - Unable to erase edge (edge not
present)");

	bool flag = true;

	for (typename vector<edgeRow<W> >::iterator it = (*(this->getRow(node1))).adjacents.begin();
			(it != (*(this->getRow(node1))).adjacents.end() && flag); it++)
		if ((*it).destID == node2.getID())
		{
			(*it).flag = false;
			flag = false;
		}
}

template<class V, class W>
list<typename Graph_matrix<V, W>::node> Graph_matrix<V, W>::adjacents(const node currNode) const
{
	list<node> nodes;

	Row<V, W> currRow = *(this->getRow(currNode));

	for (typename vector<edgeRow<W> >::iterator it = currRow.adjacents.begin(); it !=
currRow.adjacents.end();
			it++)
		if ((*it).flag == true)
		{
			node temp;
			temp.setID((*it).destID);
			nodes.push_back(temp);
		}
	return nodes;
}

template<class V, class W>
list<typename Graph_matrix<V, W>::node> Graph_matrix<V, W>::nodesList() const
{
	list<node> nodes;
	for (typename vector<Row<V, W> >::const_iterator it = this->matrix.begin(); it != this-
>matrix.end(); it++)
	{
		if ((*it).flag == true)
		{
			node temp;
			temp.setID((*it).nodeID);
			nodes.push_back(temp);
		}
	}
	return nodes;
}
```

```
template<class V, class W>
typename Graph_matrix<V, W>::label Graph_matrix<V, W>::getLabel(const node currNode) const
{
	if (!this->isNode(currNode))
		throw std::logic_error("Graph_matrix (exception) - Unable to get label (node not
present)");

	return (*(this->getRow(currNode))).nodeValue;
}

template<class V, class W>
void Graph_matrix<V, W>::setLabel(const node currNode, const label newLabel)
{
	if (!this->isNode(currNode))
		throw std::logic_error("Graph_matrix (exception) - Unable to set label (node not
present)");

	(*(this->getRow(currNode))).nodeValue = newLabel;
}

template<class V, class W>
typename Graph_matrix<V, W>::weight Graph_matrix<V, W>::getWeight(const node start, const node end)
const
{
	if (!this->isNode(start) || !this->isNode(end))
		throw std::logic_error("Graph_matrix (exception) - Unable to get weight (node not
present)");

	if (!this->isEdge(start, end))
		throw std::logic_error("Graph_matrix (exception) - Unable to get weight (edge not
present)");

	Row<V, W> currRow = *(this->getRow(start));
	for (typename vector<edgeRow<W> >::iterator it = currRow.adjacents.begin(); it !=
currRow.adjacents.end();
			it++)
		if ((*it).destID == end.getID()) // we already know that flag is true, since we have
previously checked if isEdge
			return (*it).weight;

	throw std::logic_error("Graph_matrix (exception) - Unable to get weight");
}

template<class V, class W>
void Graph_matrix<V, W>::setWeight(node start, node end, const weight newWeight)
{
	if (!this->isNode(start) || !this->isNode(end))
		throw std::logic_error("Graph_matrix (exception) - Unable to set weight (node not
present)");
	if (!this->isEdge(start, end))
		throw std::logic_error("Graph_matrix (exception) - Unable to set weight (edge not
present)");

	bool flag = true;
	for (typename vector<edgeRow<W> >::iterator it = (*(this->getRow(start))).adjacents.begin();
			(it != (*(this->getRow(start))).adjacents.end() && flag); it++)
		if ((*it).destID == end.getID())
		{
			(*it).weight = newWeight;
			flag = false;
		}
}
```

```
template<class V, class W>
typename vector<Row<V, W> >::const_iterator Graph_matrix<V, W>::getRow(const node currNode) const
{
	if (currNode.getID() >= (int) MAXNODES)
		throw std::logic_error("Graph_matrix (exception) - Unable to get row of the node 
(invalid node)");

	typename vector<Row<V, W> >::const_iterator it = this->matrix.begin();

	while ((*it).nodeID != currNode.getID())
		it++;

	return it;
}

template<class V, class W>
typename vector<Row<V, W> >::iterator Graph_matrix<V, W>::getRow(const node currNode)
{
	if (currNode.getID() >= (int) MAXNODES)
		throw std::logic_error("Graph_matrix (exception) - Unable to get row of the node 
(invalid node)");

	typename vector<Row<V, W> >::iterator it = this->matrix.begin();

	while ((*it).nodeID != currNode.getID())
		it++;

	return it;
}

#endif // _GRAPHADJMATRIX_H
```

Grafi – Realizzazione con matrice di adiacenza
File: “Row.h”

```
#ifndef _ROW_H
#define _ROW_H

#include <vector>
using std::vector;

const unsigned MAXNODES = 100;

template<class W>
class edgeRow
{
public:
	int startID;
	int destID;
	W weight;
	bool flag;

	edgeRow()
	{
		startID = -1;
		destID = -1;
		weight = W();
		flag = false;
	}

	edgeRow(const edgeRow<W>& newEdge)
	{
		*this = newEdge;
	}
	edgeRow<W>& operator =(const edgeRow<W>& newEdge)
	{
		if (&newEdge != this) // avoid auto-assignment
		{
			this->startID = newEdge.startID;
			this->destID = newEdge.destID;
			this->weight = newEdge.weight;
			this->flag = newEdge.flag;
		}
		return *this;
	}
	~edgeRow()
	{
	}
};

template<class V, class W>
class Row
{
public:

	int nodeID;
	V nodeValue;
	bool flag;
	vector<edgeRow<W> > adjacents;

	Row(const int newID)
	{
```

```
		nodeID = newID;
		nodeValue = V();
		flag = false;
		adjacents.resize(MAXNODES);
		int counter = 0;
		for(typename vector<edgeRow<W> >::iterator it = adjacents.begin(); it != 
adjacents.end(); it++)
		{
			(*it).startID = newID;
			(*it).destID = counter;
			counter++;
		}
	}

	Row(const Row<V, W>& row2)
	{
		*this = row2;
	}

	~Row()
	{
		while(!adjacents.empty())
			adjacents.pop_back();
	}

	Row<V, W>& operator =(const Row<V, W>& row2)
	{
		if (&row2 != this) // avoid auto-assignment
		{
			this->nodeID = row2.nodeID;
			this->nodeValue = row2.nodeValue;
			this->flag = row2.flag;
			this->adjacents = row2.adjacents;
		}
		return *this;
	}
};

#endif // _ROW_H
```

Grafi – Realizzazione con lista di nodi e liste di adiacenti

```
#ifndef _GRAPHADJLIST_H
#define _GRAPHADJLIST_H

#include "Graph.h"
#include "Node.h"
#include "Row.h"

template<class V, class W>
class Graph_list: public Graph<Node, V, W>
{
public:
	typedef typename Graph<Node, V, W>::node node;
	typedef typename Graph<Node, V, W>::label label;
	typedef typename Graph<Node, V, W>::weight weight;

	Graph_list();
	Graph_list(const Graph_list<V, W>&);
	~Graph_list();

	Graph_list<V, W>& operator =(const Graph_list<V, W>&);

	void create();
	bool empty() const;
	void insNode(node&);
	void insEdge(const node, const node);
	bool isNode(const node) const;
	bool isEdge(const node, const node) const;
	void eraseNode(const node);
	void eraseEdge(const node, const node);
	list<node> adjacents(const node) const;
	list<node> nodesList() const;
	label getLabel(const node) const;
	void setLabel(const node, const label);
	weight getWeight(const node, const node) const;
	void setWeight(const node, const node, const weight);

private:
	list<Row<V, W> > matrix;
	int currID;

	typename list<Row<V, W> >::const_iterator getRow(const node) const;
	typename list<Row<V, W> >::iterator getRow(const node);

};

template<class V, class W>
Graph_list<V, W>::Graph_list()
{
	currID = 0;
	this->create();
}

template<class V, class W>
Graph_list<V, W>::Graph_list(const Graph_list<V, W>& g2)
{
	*this = g2;
}
```

```
template<class V, class W>
Graph_list<V, W>::~Graph_list()
{
	while (!this->matrix.empty())
		this->matrix.pop_back();

	this->currID = 0;
}

template<class V, class W>
Graph_list<V, W>& Graph_list<V, W>::operator =(const Graph_list<V, W>& g2)
{
	if (&g2 != this) // avoid auto-assignment
	{
		while (!this->matrix.empty())
			this->matrix.pop_back();

		this->matrix = g2.matrix;
		this->currID = g2.currID;
	}
	return *this;
}

template<class V, class W>
void Graph_list<V, W>::create()
{
	currID = 0;
}

template<class V, class W>
bool Graph_list<V, W>::empty() const
{
	return (currID == 0);
}

template<class V, class W>
void Graph_list<V, W>::insNode(node& newNode)
{
	if (newNode.getID() != -1) // the node belongs to a graph
		throw std::logic_error("Graph_list (exception) - Unable to insert node (already
used)");

	newNode.setID(this->currID);
	currID++;

	matrix.push_back(Row<V, W>(newNode.getID()));
}

template<class V, class W>
void Graph_list<V, W>::insEdge(const node start, const node end)
{
	if (!this->isNode(start) || !this->isNode(end))
		throw std::logic_error("Graph_list (exception) - Unable to insert edge (node not
present)");

	if (this->isEdge(start, end))
		throw std::logic_error("Graph_list (exception) - Unable to insert edge (edge already
present)");

	(*this->getRow(start)).adjacents.push_back(edgeRow<W>(start.getID(), end.getID()));
}
```

```
template<class V, class W>
bool Graph_list<V, W>::isNode(const node currNode) const
{
	if (this->empty())
		return false;

	for (typename list<Row<V, W> >::const_iterator it = this->matrix.begin(); it != this-
>matrix.end(); it++)
		if ((*it).nodeID == currNode.getID())
			return true;

	return false;
}

template<class V, class W>
bool Graph_list<V, W>::isEdge(const node start, const node end) const
{
	if (!this->isNode(start) || !this->isNode(end))
		throw std::logic_error("Graph_list (exception) - Unable to check edge (node not
present)");

	Row<V, W> currRow = *(this->getRow(start));

	for (typename list<edgeRow<W> >::iterator it = currRow.adjacents.begin(); it !=
currRow.adjacents.end(); it++)
		if ((*it).destID == end.getID())
			return true;

	return false;
}

template<class V, class W>
void Graph_list<V, W>::eraseNode(const node currNode)
{
	if (!this->isNode(currNode))
		throw std::logic_error("Graph_list (exception) - Unable to erase node (node not
present)");

	if (this->linked(currNode))
		throw std::logic_error("Graph_list (exception) - Unable to erase node (node linked to
others)");

	matrix.erase(this->getRow(currNode));
}

template<class V, class W>
void Graph_list<V, W>::eraseEdge(const node node1, const node node2)
{
	if (!this->isNode(node1) || !this->isNode(node2))
		throw std::logic_error("Graph_list (exception) - Unable to erase edge (node not
present)");

	if (!this->isEdge(node1, node2))
		throw std::logic_error("Graph_list (exception) - Unable to erase edge (edge not
present)");

	bool flag = true;

	for (typename list<edgeRow<W> >::iterator it = (*this->getRow(node1)).adjacents.begin();
			(it != (*this->getRow(node1)).adjacents.end() && flag); it++)
		if ((*it).destID == node2.getID())
		{
```

```
                        (*this->getRow(node1)).adjacents.erase(it);
                        flag = false;
                }
}

template<class V, class W>
list<typename Graph_list<V, W>::node> Graph_list<V, W>::adjacents(const node currNode) const
{
        list<node> nodes;

        Row<V, W> currRow = *(this->getRow(currNode));

        for (typename list<edgeRow<W> >::iterator it = currRow.adjacents.begin(); it !=
currRow.adjacents.end(); it++)
        {
                node temp;
                temp.setID((*it).destID);
                nodes.push_back(temp);
        }

        return nodes;
}

template<class V, class W>
list<typename Graph_list<V, W>::node> Graph_list<V, W>::nodesList() const
{
        list<node> nodes;

        for (typename list<Row<V, W> >::const_iterator it = this->matrix.begin(); it != this-
>matrix.end(); it++)
        {
                node temp;
                temp.setID((*it).nodeID);
                nodes.push_back(temp);
        }

        return nodes;
}

template<class V, class W>
typename Graph_list<V, W>::label Graph_list<V, W>::getLabel(const node currNode) const
{
        if (!this->isNode(currNode))
                throw std::logic_error("Graph_list (exception) - Unable to get label (node not
present)");

        return (*(this->getRow(currNode))).nodeValue;
}

template<class V, class W>
void Graph_list<V, W>::setLabel(const node currNode, const label newLabel)
{
        if (!this->isNode(currNode))
                throw std::logic_error("Graph_list (exception) - Unable to set label (node not
present)");

        (*(this->getRow(currNode))).nodeValue = newLabel;
}
```

```
template<class V, class W>
typename Graph_list<V, W>::weight Graph_list<V, W>::getWeight(const node start, const node end)
const
{
	if (!this->isNode(start) || !this->isNode(end))
		throw std::logic_error("Graph_list (exception) - Unable to get weight (node not
present)");

	if (!this->isEdge(start, end))
		throw std::logic_error("Graph_list (exception) - Unable to get weight (edge not
present)");

	Row<V, W> currRow = *(this->getRow(start));

	for (typename list<edgeRow<W> >::iterator it = currRow.adjacents.begin(); it !=
currRow.adjacents.end();
			it++)
		if ((*it).destID == end.getID())
			return (*it).weight;

	throw std::logic_error("Graph_list (exception) - Unable to get weight");
}

template<class V, class W>
void Graph_list<V, W>::setWeight(node start, node end, const weight newWeight)
{
	if (!this->isNode(start) || !this->isNode(end))
		throw std::logic_error("Graph_list (exception) - Unable to set weight (node not
present)");

	if (!this->isEdge(start, end))
		throw std::logic_error("Graph_list (exception) - Unable to set weight (edge not
present)");

	bool flag = true;

	for (typename list<edgeRow<W> >::iterator it = (*this->getRow(start)).adjacents.begin();
			(it != (*this->getRow(start)).adjacents.end() && flag); it++)
		if ((*it).destID == end.getID())
		{
			(*it).weight = newWeight;
			flag = false;
		}
}

template<class V, class W>
typename list<Row<V, W> >::const_iterator Graph_list<V, W>::getRow(const node currNode) const
{
	if (!this->isNode(currNode))
		throw std::logic_error("Graph_list (exception) - Unable to get row of the node
(invalid node)");

	typename list<Row<V, W> >::const_iterator it = this->matrix.begin();

	while ((*it).nodeID != currNode.getID())
		it++;

	return it;
}
```

```
template<class V, class W>
typename list<Row<V, W> >::iterator Graph_list<V, W>::getRow(const node currNode)
{
	if (!this->isNode(currNode))
		throw std::logic_error("Graph_list (exception) - Unable to get row of the node 
(invalid node)");

	typename list<Row<V, W> >::iterator it = this->matrix.begin();

	while ((*it).nodeID != currNode.getID())
		it++;

	return it;
}

#endif // _GRAPHADJLIST_H
```

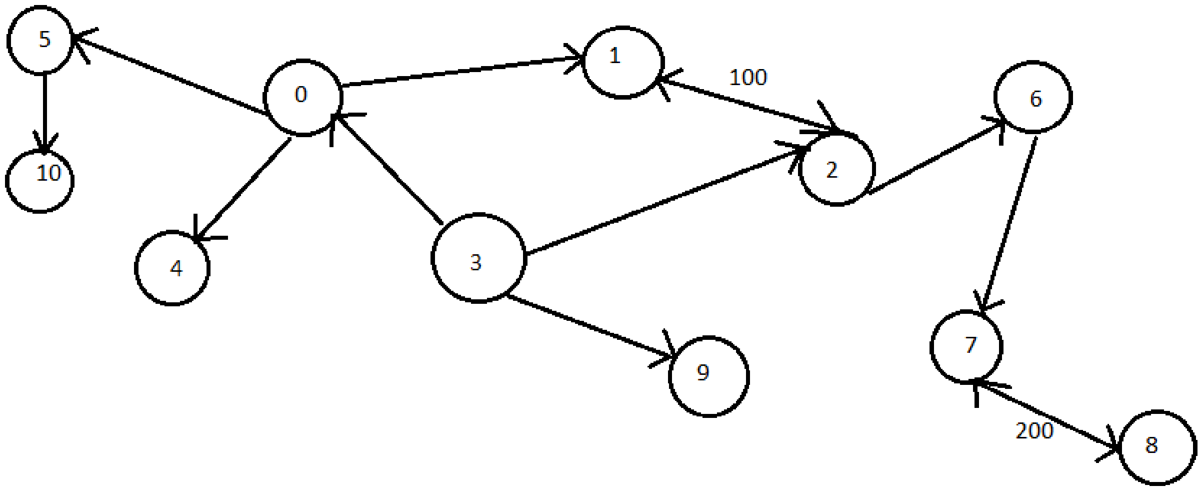
Grafi – Realizzazione con lista di nodi e liste di adiacenti
File: “Row.h”

```
#ifndef _ROW_H
#define _ROW_H

#include <list>
using std::list;

template<class W>
class edgeRow
{
public:
	int startID;
	int destID;
	W weight;

	edgeRow(const int start, const int end)
	{
		startID = start;
		destID = end;
		weight = W();
	}

	edgeRow(const edgeRow<W>& newEdge)
	{
		*this = newEdge;
	}
	edgeRow<W>& operator =(const edgeRow<W>& newEdge)
	{
		if (&newEdge != this) // avoid auto-assignment
		{
			this->startID = newEdge.startID;
			this->destID = newEdge.destID;
			this->weight = newEdge.weight;
		}
		return *this;
	}
	~edgeRow()
	{
	}
};

template<class V, class W>
class Row
{
public:

	int nodeID;
	V nodeValue;
	list<edgeRow<W> > adjacents;

	Row(const int newID)
	{
		nodeID = newID;
		nodeValue = V();
	}
```

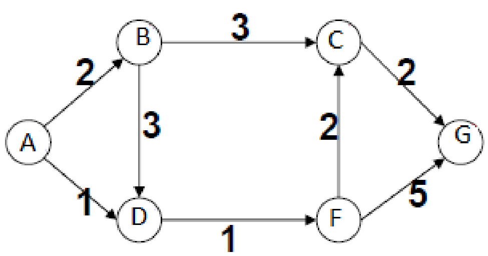
```
	Row(const Row<V, W>& row2)
	{
		*this = row2;
	}

	~Row()
	{
		while(!adjacents.empty())
			adjacents.pop_back();
	}

	Row<V, W>& operator =(const Row<V, W>& row2)
	{
		if (&row2 != this) // avoid auto-assignment
		{
			this->nodeID = row2.nodeID;
			this->nodeValue = row2.nodeValue;
			this->adjacents = row2.adjacents;
		}
		return *this;
	}
};

#endif // _ROW_H
```

Grafi – Tester

graph



graph2




```
#include "Graph_list.h"
using std::cout;

int main()
{

        Graph_list<int, int> graph;

        Node n0;
        Node n1;
        Node n2;
        Node n3;
        Node n4;
        Node n5;
        Node n6;
        Node n7;
        Node n8;
        Node n9;
        Node n10;

        graph.insNode(n0);
```

```
graph.setLabel(n0, 0);
graph.insNode(n1);
graph.setLabel(n1, 1);
graph.insNode(n2);
graph.setLabel(n2, 2);
graph.insNode(n3);
graph.setLabel(n3, 3);
graph.insNode(n4);
graph.setLabel(n4, 4);
graph.insNode(n5);
graph.setLabel(n5, 5);
graph.insNode(n6);
graph.setLabel(n6, 6);
graph.insNode(n7);
graph.setLabel(n7, 7);
graph.insNode(n8);
graph.setLabel(n8, 8);
graph.insNode(n9);
graph.setLabel(n9, 9);
graph.insNode(n10);
graph.setLabel(n10, 10);

graph.insEdge(n0, n5);
graph.insEdge(n0, n4);
graph.insEdge(n0, n1);
graph.insEdge(n3, n0);
graph.insEdge(n1, n2);
graph.insEdge(n2, n1);
graph.insEdge(n3, n2);
graph.insEdge(n3, n9);
graph.insEdge(n2, n6);
graph.insEdge(n6, n7);
graph.insEdge(n7, n8);
graph.insEdge(n8, n7);
graph.insEdge(n5, n10);

graph.setWeight(n1, n2, 100);
graph.setWeight(n7, n8, 200);

cout << graph;

if (graph.connectedGraph())
       cout << "Graph is connected" << endl;
else
       cout << "Graph isn't connected" << endl;

if (graph.stronglyConnectedGraph())
       cout << "Graph is strongly connected" << endl;
else
       cout << "Graph isn't strongly connected" << endl;

if (graph.acyclicGraph())
       cout << "Graph is acyclic" << endl;
else
       cout << "Graph isn't acyclic" << endl;

cout << endl << "Path from n0 to n7: " << endl;
list<Graph_list<int, int>::node> temp = graph.getPath(n0, n7);
for (list<Graph_list<int, int>::node>::iterator it = temp.begin(); it != temp.end(); it++)
       cout << *it << " ";

cout << endl << endl << "DFS: " << endl;
```

```
        graph.DFS(n0);

        cout << endl << endl;

        cout << "BFS: " << endl;
        graph.BFS(n0);

        cout << endl << endl;

        Graph_list<char, int> graph2;

        Node n11; // A
        Node n12; // B
        Node n13; // C
        Node n14; // D
        Node n15; // F
        Node n16; // G

        graph2.insNode(n11);
        graph2.setLabel(n11, 'A');
        graph2.insNode(n12);
        graph2.setLabel(n12, 'B');
        graph2.insNode(n13);
        graph2.setLabel(n13, 'C');
        graph2.insNode(n14);
        graph2.setLabel(n14, 'D');
        graph2.insNode(n15);
        graph2.setLabel(n15, 'F');
        graph2.insNode(n16);
        graph2.setLabel(n16, 'G');

        graph2.insEdge(n11, n12);
        graph2.setWeight(n11, n12, 2);
        graph2.insEdge(n11, n14);
        graph2.setWeight(n11, n14, 1);
        graph2.insEdge(n12, n14);
        graph2.setWeight(n12, n14, 3);
        graph2.insEdge(n12, n13);
        graph2.setWeight(n12, n13, 3);
        graph2.insEdge(n14, n15);
        graph2.setWeight(n14, n15, 1);
        graph2.insEdge(n15, n13);
        graph2.setWeight(n15, n13, 2);
        graph2.insEdge(n13, n16);
        graph2.setWeight(n13, n16, 2);
        graph2.insEdge(n15, n16);
        graph2.setWeight(n15, n16, 5);

        cout << endl << graph2;

        list<Node> temp2 = graph2.getShortestPath(n11, n16);
        cout << "Shortest path from n11 to n16 is:" << endl;

        for (list<Node>::iterator it = temp2.begin(); it != temp2.end(); it++)
                cout << *it << " ";

        list<Node> temp3 = graph2.adjacentsKLevel(n11, 3);
        cout << endl << endl << "Adjacent nodes of n11 of grade 3:" << endl;

        for (list<Node>::iterator it = temp3.begin(); it != temp3.end(); it++)
                cout << *it << " ";
}
```

Grafi – Output Tester

```
Label [ID]: Adjacent1 [ID1] (Weight1) // Adjacent2 [ID2] (Weight2) // ... //
0 [ID 0]: 5 [ID 5] (0) // 4 [ID 4] (0) // 1 [ID 1] (0) //
1 [ID 1]: 2 [ID 2] (100) //
2 [ID 2]: 1 [ID 1] (0) // 6 [ID 6] (0) //
3 [ID 3]: 0 [ID 0] (0) // 2 [ID 2] (0) // 9 [ID 9] (0) //
4 [ID 4]:
5 [ID 5]: 10 [ID 10] (0) //
6 [ID 6]: 7 [ID 7] (0) //
7 [ID 7]: 8 [ID 8] (200) //
8 [ID 8]: 7 [ID 7] (0) //
9 [ID 9]:
10 [ID 10]:

Graph isn't connected
Graph isn't strongly connected
Graph isn't acyclic

Path from n0 to n7:
 [ID 0]  [ID 1]  [ID 2]  [ID 6]  [ID 7]

DFS:
 [ID 0]  [ID 5]  [ID 10]  [ID 4]  [ID 1]  [ID 2]  [ID 6]  [ID 7]  [ID 8]

BFS:
 [ID 0]  [ID 5]  [ID 4]  [ID 1]  [ID 10]  [ID 2]  [ID 6]  [ID 7]  [ID 8]


Label [ID]: Adjacent1 [ID1] (Weight1) // Adjacent2 [ID2] (Weight2) // ... //
A [ID 0]: B [ID 1] (2) // D [ID 3] (1) //
B [ID 1]: D [ID 3] (3) // C [ID 2] (3) //
C [ID 2]: G [ID 5] (2) //
D [ID 3]: F [ID 4] (1) //
F [ID 4]: C [ID 2] (2) // G [ID 5] (5) //
G [ID 5]:

Shortest path from n11 to n16 is:
 [ID 0]  [ID 3]  [ID 4]  [ID 2]  [ID 5]

Adjacent nodes of n11 of grade 3:
 [ID 5]  [ID 2]
```

Heap Sort

```
#ifndef _HEAPSORT_H_
#define _HEAPSORT_H_

#include "heap.h"

template<class T>
void heapSort(T vector[], const int len)
{
	Heap<T, T> heap(len);

	for (int i = 0; i < len; i++)
		heap.insert(vector[i], vector[i]);

	for (int i = 0; i < len; i++)
	{
		vector[i] = (heap.getMin()).getValue();
		heap.pop();
	}
}

#endif /* _HEAPSORT_H_ */
```

Heap Sort – Tester

```
#include "heapsort.h"

int main()
{
	char array[10];

	array[0] = 'a';
	array[1] = 'b';
	array[2] = 'f';
	array[3] = 'j';
	array[4] = 'z';
	array[5] = 'g';
	array[6] = 'l';
	array[7] = 'a';
	array[8] = 'n';
	array[9] = 'e';

	for(int i = 0; i < 10; i++)
		cout << array[i] << " ";

	heapSort(array, 10);

	cout << endl;

	for(int i = 0; i < 10; i++)
		cout << array[i] << " ";

}
```

Heap Sort – Output tester

```
a b f j z g l a n e
a a b e f g j l n z
```

Minimo e massimo contemporaneamente

```
#ifndef _MINMAX_H
#define _MINMAX_H

template<class T>
struct Pair
{
	typedef T value_type;

	value_type min;
	value_type max;
};

template<class T>
Pair<T> minMAX(T* array, int start, int end)
{
	Pair<T> solution;

	if ((end - start) < 2) // 0 or 1 element
	{
		if (array[start] < array[end])
		{
			solution.min = array[start];
			solution.max = array[end];
		}
		else
		{
			solution.min = array[end];
			solution.max = array[start];
		}
	}
	else // 2 or more elements
	{
		Pair<T> temp1 = minMAX(array, start, ((end + start) / 2)); // first half
		Pair<T> temp2 = minMAX(array, (((end + start) / 2) + 1), end); // second half

		if (temp1.min < temp2.min)
			solution.min = temp1.min;
		else
			solution.min = temp2.min;

		if (temp1.max > temp2.max)
			solution.max = temp1.max;
		else
			solution.max = temp2.max;
	}

	return solution;
}

#endif // _MINMAX_H
```

Minimo e massimo contemporaneamente – Tester

```
#include <iostream>
#include "minmax.h"

using std::cout;
using std::endl;

const unsigned int maxsize = 9;

int main()
{
	int array[maxsize];

	array[0] = 7;
	array[1] = 9;
	array[2] = 6;
	array[3] = 4;
	array[4] = 5;
	array[5] = 3;
	array[6] = 1;
	array[7] = 8;
	array[8] = 2;

	Pair<int> values = minMAX(array, 0, maxsize-1);

	cout << "min: " << values.min << endl;

	cout << "MAX: " << values.max << endl;

}
```

Minimo e massimo contemporaneamente – Output tester

```
min: 1
MAX: 9
```

Natural Merge Sort (ricorsivo)

```
#ifndef _NATURALMERGESORT_H
#define _NATURALMERGESORT_H

#include <list>
using std::list;

template<class T>
bool sorted(list<T>& L)
{
	if (L.empty())
		return true;

	typename list<T>::iterator it = L.begin();
	typename list<T>::iterator it2 = L.begin();

	for (it2++; it2 != L.end(); it2++)
	{
		if (*it > *it2)
			return false;
		it++;
	}

	return true;
}

template<class T>
void NMS(list<T>& L)
{
	if (!sorted(L))
	{
		list<T> list1;
		list<T> list2;
		int currList = 1;

		T temp = L.front();
		L.pop_front();
		list1.push_back(temp);

		while (!L.empty())
		{
			if (L.front() < temp) // we have to switch the current list
			{
				if (currList == 1)
					currList = 2;
				else //currList == 2
					currList = 1;
			}

			temp = L.front();
			L.pop_front();

			if (currList == 1)
				list1.push_back(temp);
			else
				list2.push_back(temp);

		}
```

```
            NMS(list1);
            NMS(list2);

            while (!list1.empty() && !list2.empty())
            {
                  if (list1.front() < list2.front())
                  {
                        L.push_back(list1.front());
                        list1.pop_front();
                  }
                  else
                  {
                        L.push_back(list2.front());
                        list2.pop_front();
                  }
            }

            while (!list1.empty())
            {
                  L.push_back(list1.front());
                  list1.pop_front();
            }

            while (!list2.empty())
            {
                  L.push_back(list2.front());
                  list2.pop_front();
            }
      }
}

#endif // _NATURALMERGESORT_H
```

Natural Merge Sort (ricorsivo) – Tester

```
#include <iostream>
#include "nms.h"

using std::cout;
using std::endl;

int main()
{
	list<int> l1;

	l1.push_back(10);
	l1.push_back(2);
	l1.push_back(5);
	l1.push_back(3);
	l1.push_back(7);
	l1.push_back(9);
	l1.push_back(4);
	l1.push_back(1);

	cout << "list: " << endl;
	for (list<int>::iterator it = l1.begin(); it != l1.end(); it++)
		cout << *it << " ";

	NMS(l1);

	cout << endl << "After NMS: " << endl;
	for (list<int>::iterator it = l1.begin(); it != l1.end(); it++)
		cout << *it << " ";

}
```

Natural Merge Sort (ricorsivo) – Output tester

```
list:
10 2 5 3 7 9 4 1

After NMS:
1 2 3 4 5 7 9 10
```

String Matching (Knuth-Morris-Pratt)

```
#ifndef _STRINGMATCHING_H
#define _STRINGMATCHING_H

#include <vector>
using std::vector;

void buildTable(vector<int>& Table, vector<char> Pattern)
{
	int pos = 2;
	int index = 0; // index starting from 0 in pattern of the next char of the candidate
substring

	Table[0] = -1;
	Table[1] = 0;

	while (pos < (int) Pattern.size())
	{
		if (Pattern[pos - 1] == Pattern[index])
		{
			Table[pos] = index + 1;
			pos++;
			index++;
		}
		else if (index > 0)
		{
			index = Table[index];
		}
		else
		{
			Table[pos] = 0;
			pos++;
		}
	}
}

int KMP(vector<char> Pattern, vector<char> Text, vector<int> Table)
{
	int m = 0; // current position in the text
	int i = 0; // current position in the pattern (char that will be checked)

	while (Text[m + i] != '\0' && Pattern[i] != '\0')
		if (Text[m + i] == Pattern[i])
			++i;
		else
		{
			m += i - Table[i];
			if (i > 0)
				i = Table[i];
		}

	if (Pattern[i] == '\0')
		return m;
	else
		return -1;
}

#endif // _STRINGMATCHING_H
```

String Matching (Knuth-Morris-Pratt) - Tester

```
#include "string_matching.h"
#include <iostream>
#include <fstream>
#include <stdexcept>

using std::cout;
using std::endl;
using std::vector;
using std::ios;
using std::fstream;

int main()
{
	vector<char> pattern; // the pattern we want to search in the text
	vector<char> text; // the text to be searched
	vector<int> table; // required for KMP

	// acquire pattern from file
	fstream f1;
	f1.open("pattern.txt", ios::in); // open the file in read mode
	if (!f1)
		throw std::logic_error("Unable to open pattern.txt");

	while (!f1.eof())
	{
		char c = f1.get();
		if (c != EOF) // the text isn't completely analysed
			pattern.push_back(c);
	}
	f1.close();

	pattern.push_back('\0');
	table.resize(pattern.size()); // set table size = pattern size

	// acquire text from file
	f1.open("text.txt", ios::in); // open the file in read mode
	if (!f1)
		throw std::logic_error("Unable to open text.txt");

	while (!f1.eof())
	{
		char c = f1.get();
		if (c != EOF) // the text isn't completely analysed
			text.push_back(c);
	}
	f1.close();

	text.push_back('\0');

	buildTable(table, pattern);

	int firstMatch = KMP(pattern, text, table);

	if (firstMatch == -1)
		cout << "No matching. Pattern isn't present in the text." << endl;
	else
		cout << "First match starts from position: " << firstMatch << endl;
}
```

String Matching (Knuth–Morris–Pratt)
File: “text.txt”

```
ABC ABCDAB ABCDABCDABDE
```

String Matching (Knuth–Morris–Pratt)
File: “pattern.txt”

```
ABCDABD
```

String Matching (Knuth–Morris–Pratt) – Output tester

```
First match starts from position: 15
```

Codifica di Huffman

```
#ifndef _HUFFMAN_H_
#define _HUFFMAN_H_

#include "set_pointer.h"
#include "heap.h"
#include "Graph_list.h"
#include <fstream>
#include <string>

using std::ios;
using std::fstream;
using std::list;
using std::ostream;
using std::string;

struct Pair
{
	char letter;
	int occurr;

	Pair(char newLetter = char(), int newOccurr = int())
	{
		letter = newLetter;
		occurr = newOccurr;
	}
};

ostream& operator <<(ostream& os, const Pair& p)
{
	if(p.letter != '\n')
		os << "Char: " << p.letter << " - Occurrences: " << p.occurr;
	else
		os << "Char: RETURN - Occurrences: " << p.occurr;
	return os;
}

struct EncodedPair
{
	char letter;
	int occurr;
	string code;

	EncodedPair(char newLetter = char(), int newOccurr = int(), string newCode = string())
	{
		letter = newLetter;
		occurr = newOccurr;
		code = newCode;
	}
};

ostream& operator <<(ostream& os, const EncodedPair& p)
{
	if(p.letter != '\n')
		os << "Char: " << p.letter << " - String: (" << p.code << ")";
	else
		os << "Char: RETURN - String: (" << p.code << ")";
	return os;
}
```

```
List_pointer<Pair> getOccurr()
{
	List_pointer<Pair> occurrences; // list of pair
	Set_pointer<char> characters; // set of characters that have already been read

	fstream f1;
	f1.open("text.txt", ios::in); // open the file in read mode

	if (!f1)
		throw std::logic_error("Unable to open text.txt");

	char c;
	while (!f1.eof())
	{
		c = f1.get();
		if (c != EOF && !characters.find(c)) // we have read a new character (and the text
isn't completely analysed)
		{
			characters.insert(c); // add the character that has just been read
			occurrences.push_back(Pair(c, 1));
		}
		else if (c != EOF) // we have re-read a character (and the text isn't completely
analysed)
		{
			bool flag = true;
			for (List_pointer<Pair>::position p = occurrences.begin(); flag; p =
occurrences.next(p)) // iterate on the list until the character has been found
				if (occurrences.read(p).letter == c)
				{
					Pair temp = occurrences.read(p);
					temp.occurr++; // increase the number of occurrences
					occurrences.write(temp, p); // update the value
					flag = false;
				}
		}
	}
	f1.close();

	return occurrences;
}

void buildGraph(Graph_list<Pair, int>& graph, List_pointer<Pair> occurrences)
{
	Heap<Graph_list<Pair, int>::node, int> priQueue;

	while (!occurrences.empty()) // convert the list of pairs in a priority queue of nodes of the
graph
	{
		Pair tempPair = occurrences.read(occurrences.begin()); // first pair of the list
		occurrences.pop_front();

		Graph_list<Pair, int>::node tempNode;
		graph.insNode(tempNode);
		graph.setLabel(tempNode, tempPair);

		priQueue.insert(tempNode, tempPair.occurr);
	}

	while (priQueue.size() != 1) // when there is only one node the graph has been built
	{
		// acquire the two pairs with lowest frequency
		Graph_list<Pair, int>::node min1 = priQueue.getMin().getValue();
```

```
			priQueue.pop();
			Graph_list<Pair, int>::node min2 = priQueue.getMin().getValue();
			priQueue.pop();

			// add to the graph a new node, having 2 sons (the 2 pairs)
			// the label will be a pair ( \b as char, sum of frequencies as number of occurrences)
			Graph_list<Pair, int>::node temp;
			graph.insNode(temp);
			int freq = graph.getLabel(min1).occurr + graph.getLabel(min2).occurr;
			graph.setLabel(temp, Pair('\b', freq));
			graph.insEdge(temp, min1);
			graph.insEdge(temp, min2);

			// add the node to the queue
			priQueue.insert(temp, freq);
		}
}

void assignWeight(Graph_list<Pair, int>& graph)
{
		list<Graph_list<Pair, int>::node> nodes = graph.nodesList();

		for (list<Graph_list<Pair, int>::node>::iterator it = nodes.begin(); it != nodes.end(); it++)
		{
			// we know that adjacent nodes list will be made up of 0 or 2 elements (graph is a
tree)
			list<Graph_list<Pair, int>::node> adjTemp = graph.adjacents(*it);
			if (adjTemp.size() != 0)
			{
				graph.setWeight(*it, adjTemp.front(), 0); // we set 0 on the firt adjacent node
				graph.setWeight(*it, adjTemp.back(), 1); // and 1 on the other adjacent node
			}
		}
}

string getString(Graph_list<Pair, int>& graph, char c)
{
		string code; // code of the char

		Graph_list<Pair, int>::node nodeC; // node of the character
		list<Graph_list<Pair, int>::node> nodes = graph.nodesList();
		Graph_list<Pair, int>::node root = nodes.back(); // we know that the root is the last element
added in the graph

		bool flag = true;
		for (list<Graph_list<Pair, int>::node>::iterator it = nodes.begin(); flag; it++)
			if (graph.getLabel(*it).letter == c)
			{
				flag = false;
				nodeC = *it;
			}

		nodes.clear();

		nodes = graph.getPath(root, nodeC);
		for (list<Graph_list<Pair, int>::node>::iterator it = nodes.begin(), it2 = it; it2 !=
nodes.end(); it++)
		{
			it2++;
			if (graph.isEdge(*it, *it2))
			{
```

```
                    char temp = (graph.getWeight(*it, *it2) + '0'); // we convert the weight (0 or
1) in the char 0 or 1
                    code.push_back(temp);
              }
       }

       return code;
}

List_pointer<EncodedPair> getCodes(Graph_list<Pair, int>& graph, List_pointer<Pair> occurrences)
{
       List_pointer<EncodedPair> encodedList;

       // first we insert letter and frequency (without the encoded string)
       for (List_pointer<Pair>::position p = occurrences.begin(); !occurrences.end(p); p =
occurrences.next(p))
              encodedList.push_back(EncodedPair(occurrences.read(p).letter,
occurrences.read(p).occurr));

       // now we add the encoded string
       for (List_pointer<EncodedPair>::position p = encodedList.begin(); !encodedList.end(p); p =
encodedList.next(p))
       {
              char c = encodedList.read(p).letter;
              int occurr = encodedList.read(p).occurr;
              encodedList.write(EncodedPair(c, occurr, getString(graph, c)), p); // update the
EncodedPair
       }

       return encodedList;
}

void compareResults(List_pointer<EncodedPair>& encodedList)
{
       float numChars = 0; // number of characters in the text
       float numBit = 0; // number of bits needed with Huffman encoding system

       for (List_pointer<EncodedPair>::position p = encodedList.begin(); !encodedList.end(p); p =
encodedList.next(p))
              numChars += encodedList.read(p).occurr;

       for (List_pointer<EncodedPair>::position p = encodedList.begin(); !encodedList.end(p); p =
encodedList.next(p))
       {
              float temp = encodedList.read(p).occurr / numChars; // percentage of appearance
              numBit += (temp * encodedList.read(p).code.length());
       }

       numBit = numBit * numChars;

       cout << endl << endl;

       cout << "# bit needed with ASCII encoding: " << 8 * numChars << endl;
       cout << "# bit needed with Huffman encoding: " << numBit;
}

#endif /* _HUFFMAN_H_ */
```

Codifica di Huffman – Tester

```
#include "huffman.h"

using std::cout;
using std::endl;

int main()
{
	List_pointer<Pair> occurrences;
	Graph_list<Pair, int> huffmanGraph; // graph of pairs; edges are 0, 1

	occurrences = getOccurr(); // analyse the text and acquire frequencies

	cout << "Text has been analysed. Here is the result:" << endl;
	for (List_pointer<Pair>::position p = occurrences.begin(); !occurrences.end(p); p =
occurrences.next(p))
		cout << endl << occurrences.read(p);

	cout << endl << endl;

	buildGraph(huffmanGraph, occurrences);

	assignWeight(huffmanGraph);

	List_pointer<EncodedPair> encodedList = getCodes(huffmanGraph, occurrences);

	cout << "Encoded letters:" << endl;
	for(List_pointer<EncodedPair>::position p = encodedList.begin(); !encodedList.end(p); p =
encodedList.next(p))
		cout << endl << encodedList.read(p);

	compareResults(encodedList);

}
```

Codifica di Huffman
File: “text.txt”

A lan Turing
Da Wikipedia, l'enciclopedia libera.
Alan Turing (1927 circa)

Alan Mathison Turing (Londra, 23 giugno 1912 – Wilmslow, 7 giugno 1954) è stato un matematico, logico e crittografo britannico, considerato uno dei padri dell'informatica e uno dei più grandi matematici del XX secolo.
Alan Turing giovane sportivo
Ritratto in ardesia di Turing al Bletchley Park. Sullo sfondo, inquadrata, una foto di Turing a 39 anni

Il suo lavoro ebbe vasta influenza sullo sviluppo dell'informatica, grazie alla sua formalizzazione dei concetti di algoritmo e calcolo mediante la macchina di Turing, che a sua volta ha svolto un ruolo significativo nella creazione del moderno computer. Per questi contributi Turing è solitamente considerato il padre della scienza informatica e dell'intelligenza artificiale.

Fu anche uno dei più brillanti crittoanalisti che operavano in Inghilterra, durante la seconda guerra mondiale, per decifrare i messaggi scambiati da diplomatici e militari delle Potenze dell'Asse. Durante la Seconda Guerra Mondiale Turing lavorò infatti a Bletchley Park, il principale centro di crittoanalisi del Regno Unito, dove ideò una serie di tecniche per violare i cifrari tedeschi, incluso il metodo della Bomba, una macchina elettromeccanica in grado di decodificare codici creati mediante la macchina Enigma.

Omosessuale, morì suicida a soli 41 anni, probabilmente in seguito alle persecuzioni subite da parte delle autorità britanniche a causa della sua omosessualità. Nel 1952 era stato infatti dichiarato colpevole di "grave indecenza" per essere stato sorpreso in rapporti sessuali con un altro uomo e condannato alla castrazione chimica. In suo onore la Association for Computing Machinery (ACM) ha creato nel 1966 il Turing Award, massima riconoscenza nel campo dell'informatica, dei sistemi intelligenti e dell'intelligenza artificiale.
Indice

Infanzia e giovinezza

Turing venne concepito in India, durante uno dei frequenti viaggi di suo padre, Julius Mathison Turing, membro del Indian Civil Service. Sia Julius sia sua moglie, Ethel Sara Stoney, madre del futuro Alan Turing, decisero tuttavia che il piccolo dovesse nascere sul suolo inglese. Tornarono quindi a Londra dove il 23 giugno 1912 nacque Alan Mathison Turing.

Codifica di Huffman – Output tester

```
Text has been analysed. Here is the result:

Char: A - Occurrences: 12
Char: l - Occurrences: 135
Char: a - Occurrences: 209
Char: n - Occurrences: 158
Char:   - Occurrences: 387
Char: T - Occurrences: 16
Char: u - Occurrences: 76
Char: r - Occurrences: 126
Char: i - Occurrences: 232
Char: g - Occurrences: 51
Char: RETURN - Occurrences: 28
Char: D - Occurrences: 2
Char: W - Occurrences: 2
Char: k - Occurrences: 3
Char: p - Occurrences: 30
Char: e - Occurrences: 199
Char: d - Occurrences: 79
Char: , - Occurrences: 26
Char: ' - Occurrences: 7
Char: c - Occurrences: 95
Char: o - Occurrences: 155
Char: b - Occurrences: 14
Char: . - Occurrences: 13
Char: ( - Occurrences: 3
Char: 1 - Occurrences: 11
Char: 9 - Occurrences: 8
Char: 2 - Occurrences: 7
Char: 7 - Occurrences: 3
Char: ) - Occurrences: 3
Char: M - Occurrences: 6
Char: t - Occurrences: 117
Char: h - Occurrences: 22
Char: s - Occurrences: 78
Char: L - Occurrences: 2
Char: 3 - Occurrences: 4
Char: – - Occurrences: 1
Char: m - Occurrences: 46
Char: w - Occurrences: 2
Char: 5 - Occurrences: 3
Char: 4 - Occurrences: 3
Char: è - Occurrences: 2
Char: f - Occurrences: 25
Char: ù - Occurrences: 2
Char: X - Occurrences: 2
Char: v - Occurrences: 25
Char: R - Occurrences: 2
Char: B - Occurrences: 4
Char: y - Occurrences: 4
Char: P - Occurrences: 4
Char: S - Occurrences: 7
Char: q - Occurrences: 5
```

```
Char: I - Occurrences: 9
Char: z - Occurrences: 21
Char: F - Occurrences: 1
Char: G - Occurrences: 1
Char: ò - Occurrences: 2
Char: U - Occurrences: 1
Char: E - Occurrences: 2
Char: O - Occurrences: 2
Char: ì - Occurrences: 1
Char: à - Occurrences: 2
Char: N - Occurrences: 2
Char: " - Occurrences: 2
Char: C - Occurrences: 4
Char: 6 - Occurrences: 3
Char: 8 - Occurrences: 1
Char: V - Occurrences: 1
Char: 0 - Occurrences: 1
Char: J - Occurrences: 2

Encoded letters:

Char: A - String: (10100111)
Char: l - String: (0101)
Char: a - String: (1111)
Char: n - String: (1001)
Char:   - String: (110)
Char: T - String: (0010111)
Char: u - String: (01101)
Char: r - String: (0100)
Char: i - String: (000)
Char: g - String: (101111)
Char: RETURN - String: (001001)
Char: D - String: (0110001001)
Char: W - String: (0110000000)
Char: k - String: (1010011010)
Char: p - String: (001010)
Char: e - String: (1110)
Char: d - String: (10001)
Char: , - String: (1011101)
Char: ' - String: (00101100)
Char: c - String: (10101)
Char: o - String: (0111)
Char: b - String: (0010000)
Char: . - String: (10110101)
Char: ( - String: (1011010010)
Char: 1 - String: (10100001)
Char: 9 - String: (01100011)
Char: 2 - String: (00100011)
Char: 7 - String: (1011010011)
Char: ) - String: (1010000001)
Char: M - String: (101001100)
Char: t - String: (0011)
Char: h - String: (1010010)
Char: s - String: (10000)
Char: L - String: (0110010100)
Char: 3 - String: (011001100)
```

```
Char: – - String: (01100101101)
Char: m - String: (101100)
Char: w - String: (0110000001)
Char: 5 - String: (1011010001)
Char: 4 - String: (1011010000)
Char: è - String: (0110000101)
Char: f - String: (1011100)
Char: ù - String: (0110000110)
Char: X - String: (0110010101)
Char: v - String: (1011011)
Char: R - String: (1010000000)
Char: B - String: (011000101)
Char: y - String: (001011011)
Char: P - String: (011000001)
Char: S - String: (00100010)
Char: q - String: (101000001)
Char: I - String: (01100111)
Char: z - String: (1010001)
Char: F - String: (01100010000)
Char: G - String: (01100100011)
Char: ò - String: (0010110101)
Char: U - String: (01100101100)
Char: E - String: (0110000100)
Char: O - String: (0110010111)
Char: ì - String: (01100100010)
Char: à - String: (0110000111)
Char: N - String: (0110011010)
Char: " - String: (0110011011)
Char: C - String: (011001001)
Char: 6 - String: (1010011011)
Char: 8 - String: (01100100000)
Char: V - String: (01100100001)
Char: 0 - String: (01100010001)
Char: J - String: (0010110100)

# bit needed with ASCII encoding: 20112
# bit needed with Huffman encoding: 11335
```

Colorazione di un grafo con 4 colori

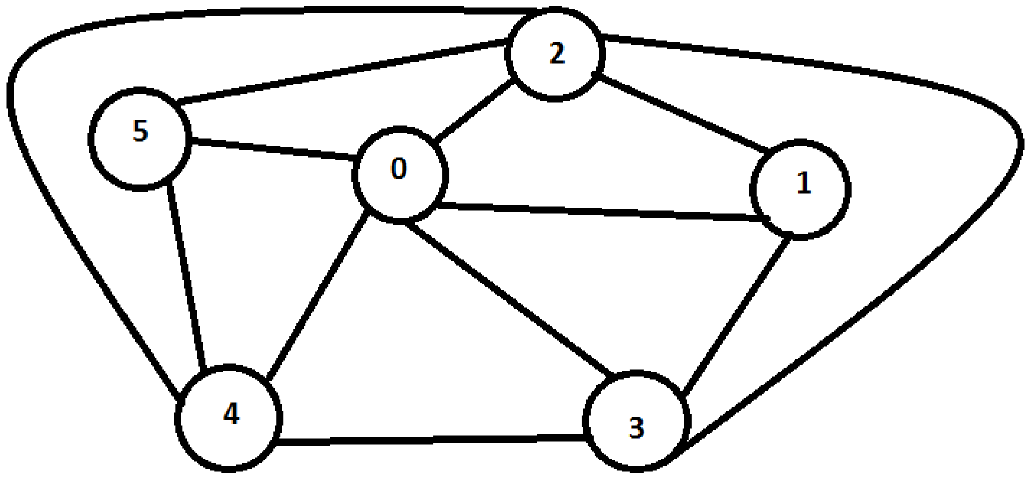
```
#ifndef _GRAPHACOLORING_H
#define _GRAPHCOLORING_H

#include <list>
#include <string>
#include <stdexcept>

#include "Graph.h"

using std::list;
using std::string;

typedef string colour;

/*
 * PRE-CONDITIONS:
 *
 * The graph must be a PLANAR GRAPH (and undirected).
 *
 * In graph theory, a planar graph is a graph that can be
 * drawn in such a way that no edges cross each other.
 *
 */

class Palette
{
public:
	list<colour> palette;
	unsigned int size;

	Palette(const unsigned int newSize)
	{
		size = newSize;
	}

	Palette(const Palette& newPalette)
	{
		*this = newPalette;
	}

	Palette& operator =(const Palette& newPalette)
	{
		if (&newPalette != this) // avoid auto-assignment
		{
			this->size = newPalette.size;
			this->palette = newPalette.palette;
		}
		return *this;
	}

	~Palette()
	{
	}

	void addColour(colour newColour)
	{
		if (palette.size() < size)
			palette.push_back(newColour);
```

```
		else
			throw std::logic_error("Palette (exception) - Unable to add colour (full 
palette)");
	}
};

template<class N, class W>
bool isAvailableColour(Graph<N, colour, W>& graph, colour currColour, typename Graph<N, colour, 
W>::node& currNode) // true if the colour isn't used by the adjacent nodes
{
	list<typename Graph<N, colour, W>::node> nodes = graph.adjacents(currNode);
	list<colour> adjColour;
	for (typename list<typename Graph<N, colour, W>::node>::iterator it = nodes.begin(); it != 
nodes.end(); it++)
		adjColour.push_back(graph.getLabel(*it));
	if (std::find(adjColour.begin(), adjColour.end(), currColour) == adjColour.end())
		return true;

	return false;
}

template<class N, class W>
bool completelyColoured(Graph<N, colour, W>& graph, Palette& currPal) // true if every node has a 
label present in the palette
{
	list<typename Graph<N, colour, W>::node> nodes = graph.nodesList();
	for (typename list<typename Graph<N, colour, W>::node>::iterator it = nodes.begin(); it != 
nodes.end(); it++)
		if (std::find(currPal.palette.begin(), currPal.palette.end(), graph.getLabel(*it)) == 
currPal.palette.end())
			return false;

	return true;
}

template<class N, class W>
bool notColouredNode(Graph<N, colour, W>& graph, Palette& currPal, typename Graph<N, colour, 
W>::node& currNode) // true if the label of the node isn't present in the palette
{
	if (std::find(currPal.palette.begin(), currPal.palette.end(), graph.getLabel(currNode)) == 
currPal.palette.end())
		return true;

	return false;
}

template<class N, class W>
void graphColour(Graph<N, colour, W>& graph, Palette& currPal, typename Graph<N, colour, W>::node& 
currNode)
{
	static bool endFlag = false; // true if all the nodes are coloured

	typename list<colour>::iterator it;
	for (it = currPal.palette.begin(); (it != currPal.palette.end() && !completelyColoured(graph, 
currPal)); it++) // for every colour (and until there are some uncoloured nodes)
		if (isAvailableColour(graph, *it, currNode)) // if there is a colour unused by 
adjacent nodes
		{
			graph.setLabel(currNode, *it); // put the colour in the current node
			if (!completelyColoured(graph, currPal)) // if there are other nodes to be 
coloured
			{
```

```
                              list<typename Graph<N, colour, W>::node> nodes = graph.nodesList();
                              bool flag = true;
                              for (typename list<typename Graph<N, colour, W>::node>::iterator it2 =
nodes.begin();
                                          it2 != nodes.end() && flag; it2++)
                                    if (notColouredNode(graph, currPal, *it2)) // find the first
uncoloured node
                                    {
                                          flag = false; // the first uncoloured node has been found
                                          graphColour(graph, currPal, *it2); // recursive-call
                                    }
                        }
                        else
                              endFlag = true; // set the flag true (algorithm will terminate)
                  }

      if (it == currPal.palette.end() && !endFlag) // if the aren't any available colours for the
node, clear is label
            graph.setLabel(currNode, colour()); // backtrack method will assign another colour to
the previous node
}

#endif // _GRAPHCOLORING_H
```

Colorazione di un grafo con 4 colori - Tester

graph

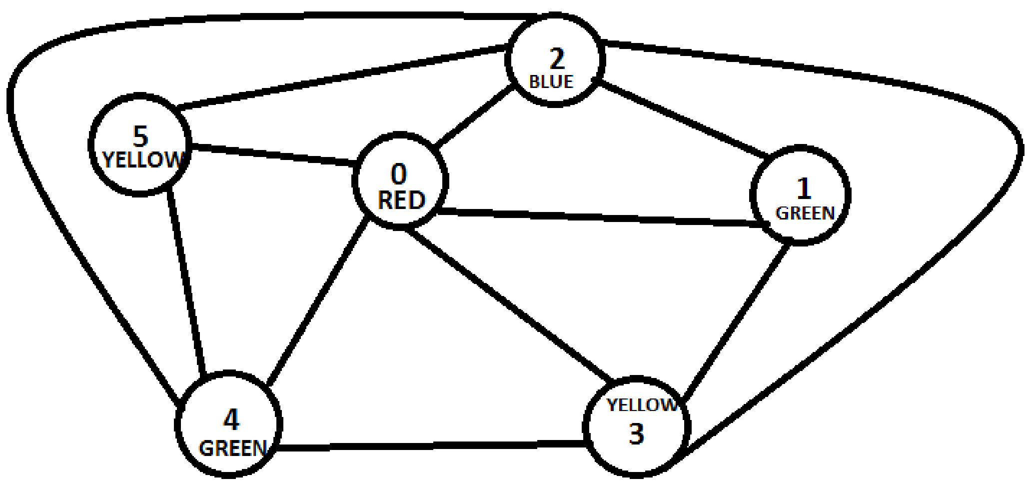



```
#include "Graph_matrix.h"
#include "graph_colouring.h"

int main()
{
	Graph_matrix<colour, int> graph;
	Palette palette(4);

	palette.addColour("Red");
	palette.addColour("Green");
	palette.addColour("Blue");
	palette.addColour("Yellow");

	Node n0;
	Node n1;
	Node n2;
	Node n3;
	Node n4;
	Node n5;

	graph.insNode(n0);
	graph.insNode(n1);
	graph.insNode(n2);
	graph.insNode(n3);
	graph.insNode(n4);
	graph.insNode(n5);

	graph.insEdge(n0, n1);
	graph.insEdge(n1, n0);

	graph.insEdge(n0, n2);
	graph.insEdge(n2, n0);
```

```
        graph.insEdge(n0, n3);
        graph.insEdge(n3, n0);

        graph.insEdge(n0, n4);
        graph.insEdge(n4, n0);

        graph.insEdge(n0, n5);
        graph.insEdge(n5, n0);

        graph.insEdge(n1, n2);
        graph.insEdge(n2, n1);

        graph.insEdge(n1, n3);
        graph.insEdge(n3, n1);

        graph.insEdge(n2, n3);
        graph.insEdge(n3, n2);

        graph.insEdge(n2, n4);
        graph.insEdge(n4, n2);

        graph.insEdge(n2, n5);
        graph.insEdge(n5, n2);

        graph.insEdge(n3, n4);
        graph.insEdge(n4, n3);

        graph.insEdge(n4, n5);
        graph.insEdge(n5, n4);

        cout << graph;

        graphColour(graph, palette, n0);

        cout << "After algorithm: " << endl << graph;
}
```

Colorazione di un grafo con 4 colori – Output tester

```
Label [ID]: Adjacent1 [ID1] (Weight1) // Adjacent2 [ID2] (Weight2) // ... //
 [ID 0]:  [ID 1] (0) //  [ID 2] (0) //  [ID 3] (0) //  [ID 4] (0) //  [ID 5] (0) //
 [ID 1]:  [ID 0] (0) //  [ID 2] (0) //  [ID 3] (0) //
 [ID 2]:  [ID 0] (0) //  [ID 1] (0) //  [ID 3] (0) //  [ID 4] (0) //  [ID 5] (0) //
 [ID 3]:  [ID 0] (0) //  [ID 1] (0) //  [ID 2] (0) //  [ID 4] (0) //
 [ID 4]:  [ID 0] (0) //  [ID 2] (0) //  [ID 3] (0) //  [ID 5] (0) //
 [ID 5]:  [ID 0] (0) //  [ID 2] (0) //  [ID 4] (0) //

After algorithm:
Label [ID]: Adjacent1 [ID1] (Weight1) // Adjacent2 [ID2] (Weight2) // ... //
Red [ID 0]: Green [ID 1] (0) // Blue [ID 2] (0) // Yellow [ID 3] (0) // Green [ID 4] (0)
// Yellow [ID 5] (0) //
Green [ID 1]: Red [ID 0] (0) // Blue [ID 2] (0) // Yellow [ID 3] (0) //
Blue [ID 2]: Red [ID 0] (0) // Green [ID 1] (0) // Yellow [ID 3] (0) // Green [ID 4] (0)
// Yellow [ID 5] (0) //
Yellow [ID 3]: Red [ID 0] (0) // Green [ID 1] (0) // Blue [ID 2] (0) // Green [ID 4] (0)
//
Green [ID 4]: Red [ID 0] (0) // Blue [ID 2] (0) // Yellow [ID 3] (0) // Yellow [ID 5] (0)
//
Yellow [ID 5]: Red [ID 0] (0) // Blue [ID 2] (0) // Green [ID 4] (0) //
```



Controllo di parentesi bilanciate

```cpp
#ifndef _BRACKETS_H
#define _BRACKETS_H

#include "stack_pointer.h"
#include <fstream>

using std::ios;
using std::fstream;

bool checkBrackets()
{
	fstream f1;
	f1.open("brackets.txt", ios::in); // open the file in read mode

	if (!f1)
		throw std::logic_error("Unable to open brackets.txt");

	char c;
	Stack_pointer<char> stack;

	while (!f1.eof())
	{
		c = f1.get();
		if (c != EOF) // the text isn't completely analysed
		{
			if (c == '{' || c == '(')
				stack.push(c);
			else if (c == '}')
			{
				if (stack.empty() || stack.read() != '{')
					return false;

				stack.pop();
			}
			else if (c == ')')
			{
				if (stack.empty() || stack.read() != '(')
					return false;

				stack.pop();
			}
		}
	}
	f1.close();

	if(stack.size() != 0)
		return false;

	return true;
}

#endif // _BRACKETS_H
```

Controllo di parentesi bilanciate – Tester

```
#include "brackets.h"

int main()
{

	if(checkBrackets())
		cout << "Balanced brackets in the file brackets.txt";
	else
		cout << "Unbalanced brackets in the file brackets.txt";

	cout << endl;
}
```

Controllo di parentesi bilanciate
File: “brackets.txt”

```
#include <iostream>

int main()
{
int number = 0;
int happiness = 0;

for(int i = 0; i < 30; i++)
{
number++;
happiness++;
}

std::cout << "Your final mark is: " << number;
}
```

Controllo di parentesi bilanciate – Output tester

```
Balanced brackets in the file brackets.txt
```

Crivello di Eratostene

```
#ifndef _SIEVE_H
#define _SIEVE_H

#include "set_bool.h"

Set_bool sieve(const long unsigned max)
{
	Set_bool prime;
	long unsigned curr = 2; // current number
	long unsigned eraser = 2; // number we are going to erase from the set
	long unsigned counter = 2; // multiplier: x2, x3, x4, x5, ecc...

	// fill the sieve
	for (long unsigned i = 0; i < max; i++)
		prime.insert(i + 1);

	while (curr * curr < max) // the algorithm finish when curr^2 >= max
	{
		while (curr * counter <= max)
		{
			eraser = curr * counter;
			if (prime.find(eraser))
				prime.erase(eraser);
			counter++;
		}
		counter = 2;

		curr++; // find next divisor

		// if the divisor isn't present in the set (it may have been removed previously)
		// we have to find another one
		while (!prime.find(curr) && curr <= max)
			curr++;
	}

	return prime;
}

#endif // _SIEVE_H
```

Crivello di Eratostene – Tester

```
#include "sieve.h"

int main()
{
	long unsigned max = 1000; // up to 4700 ca.
	Set_bool primeNumbers;

	primeNumbers = sieve(max);

	cout << primeNumbers;
	cout << "Elements found: " << primeNumbers.size();
}
```

Crivello di Eratostene – Output tester

```
[  1  2  3  5  7  11  13  17  19  23  29  31  37  41  43  47  53  59  61  67  71  73  79
83  89  97  101  103  107  109  113  127  131  137  139  149  151  157  163  167  173  179
181  191  193  197  199  211  223  227  229  233  239  241  251  257  263  269  271  277
281  283  293  307  311  313  317  331  337  347  349  353  359  367  373  379  383  389
397  401  409  419  421  431  433  439  443  449  457  461  463  467  479  487  491  499
503  509  521  523  541  547  557  563  569  571  577  587  593  599  601  607  613  617
619  631  641  643  647  653  659  661  673  677  683  691  701  709  719  727  733  739
743  751  757  761  769  773  787  797  809  811  821  823  827  829  839  853  857  859
863  877  881  883  887  907  911  919  929  937  941  947  953  967  971  977  983  991
997  ]
Elements found: 169
```

Notazione polacca inversa (shunting-yard algorithm)

```
#ifndef _RPN_H
#define _RPN_H

#include <string>
#include <iostream>
#include "stack_pointer.h"

using std::cout;
using std::endl;
using std::string;

#define is_operator(c) (c == '+' || c == '-' || c == '/' || c == '*')
#define is_number(c) (c >= '0' && c <= '9')

int op_preced(const char c)
{
	if (c == '*' || c == '/')
		return 2;
	else
		// c == '+' || c == '-'
		return 1;
}

bool RPNConverter(const string input, string& output)
{
	string::const_iterator inputPos = input.begin(); // iterator to the current position of the
input string

	char c; // temp buffer

	Stack_pointer<char> stack; // operator stack
	char tempChar; // used for record stack element

	while (inputPos != input.end())
	{
		c = *inputPos; // read one token from the input stream
		if (c != ' ')
		{
			if (is_number(c)) // if it's a number, then add it to output string.
			{
				output.push_back(c);
			}
			else if (is_operator(c)) // if it's an operator
			{
				while (stack.size() > 0)
				{
					tempChar = stack.read();
					// evaluate the priority of the operators
					if (is_operator(tempChar) && (((op_preced(c) <=
op_preced(tempChar))) || (op_preced(c) < op_preced(tempChar))))
					{
						stack.pop(); // pop op2 off the stack
						output.push_back(tempChar); // onto the output string
					}
					else
					{
						break;
					}
```

```
                }
                stack.push(c); // push op1 onto the stack
            }
            else if (c == '(') // if the token is a left parenthesis, then push it onto the
stack
            {
                stack.push(c);
            }
            else if (c == ')') // if the token is a right parenthesis
            {
                bool balancedPar = false;
                // Until the token at the top of the stack is a left parenthesis,
                // pop operators off the stack onto the output string
                while (stack.size() > 0)
                {
                    tempChar = stack.read();
                    if (tempChar == '(')
                    {
                        balancedPar = true;
                        break;
                    }
                    else
                    {
                        output.push_back(tempChar);
                        stack.pop();
                    }
                }

                if (!balancedPar) // if the stack runs out without finding a left
parenthesis there are mismatched parentheses
                {
                    cout << "Error: parentheses mismatched" << endl;
                    return false;
                }

                stack.pop(); // pop the left parenthesis from the stack, but not onto
the output string.
            }
            else
            {
                cout << "Unknown token " << c << endl;
                return false;
            }
        }
        ++inputPos;
    }

    // When there are no more tokens to read:
    // While there are still operator tokens in the stack:
    while (stack.size() > 0)
    {
        tempChar = stack.read();
        if (tempChar == '(' || tempChar == ')')
        {
            cout << "Error: parentheses mismatched" << endl;
            return false;
        }
        output.push_back(tempChar);
        stack.pop();
    }
    return true;
}
```

```
int calculateRPN(const string input)
{
	Stack_pointer<int> stack;

	string::const_iterator inputPos = input.begin(); // iterator to the current position of the
input string

	while (inputPos != input.end())
	{
		if (is_number(*inputPos))
			stack.push(*inputPos - '0');
		else // is an operator
		{
			int op1 = stack.read();
			stack.pop();
			int op2 = stack.read();
			stack.pop();

			if (*inputPos == '+')
				stack.push(op1 + op2);
			else if (*inputPos == '-')
				stack.push(op2 - op1);
			else if (*inputPos == '*')
				stack.push(op2 * op1);
			else if (*inputPos == '/')
				stack.push(op2 / op1);
			else
				throw std::logic_error("Unable to calculate RPN");
		}

		++inputPos;
	}

	if (stack.size() == 1)
		return stack.read();
	else
		throw std::logic_error("Unable to calculate RPN");

	return 0;
}

#endif // _RPN_H
```

Notazione polacca inversa - Tester

```
#include "rpn.h"
#include <fstream>

using std::ios;
using std::fstream;

int main()
{
	string input;

	fstream f1;
	f1.open("infix.txt", ios::in); // open the file in read mode

	if (!f1)
		throw std::logic_error("Unable to open infix.txt");

	char c;
	while (!f1.eof())
	{
		c = f1.get();
		if (c != EOF) // the text isn't completely analysed
			input.push_back(c);
	}
	f1.close();

	cout << "input: " << input << endl;

	string output;

	if (RPNConverter(input, output))
		cout << "output: " << output << endl << endl << "Result: " << calculateRPN(output) <<
endl;
	else
		cout << "unable to convert input string to RPN" << endl;

}
```

Notazione polacca inversa
File: “infix.txt”

```
5*(2+4)/3+3*4+2
```

Notazione polacca inversa – Output tester

```
input: 5*(2+4)/3+3*4+2
output: 524+*3/34*+2+

Result: 24
```

Problema dello zaino (backtracking e greedy)

```
#ifndef _KNAPSACK_H
#define _KNAPSACK_H

#include <iostream>
#include <algorithm>
#include "Bag.h"

using std::cout;
using std::cin;

template<class T, class N>
void getItems(vector<Item<T, N> >& items)
{
	int flag = 1;
	int counter = 1;

	do
	{
		Item<T, N> currItem;
		cout << "Insert item name: ";
		getline(cin, currItem.name);
		cout << "Insert item value: ";
		cin >> currItem.value;
		cout << "Insert item weight: ";
		cin >> currItem.weight;
		currItem.ID = counter;

		items.push_back(currItem);
		counter++;

		cout << "Press 1 to add an item or 0 to start algorithm: ";
		cin >> flag;
		cin.ignore();
		cout << endl;
	} while (flag);
}

template<class T, class N>
void printItems(vector<Item<T, N> >& items)
{
	cout << "Items:" << endl;

	for (typename vector<Item<T, N> >::iterator it = items.begin(); it != items.end(); it++)
	{
		cout << (*it).name << " - Value: " << (*it).value << " - Weight: " << (*it).weight <<
" - ID: " << (*it).ID;
		cout << endl;
	}

}

template<class T, class N>
void buildTree(Bag<T, N>& bag, vector<Item<T, N> >& items, typename NaryTree_pointer<Item<T, N>
>::node currNode)
{
	for (typename vector<Item<T, N> >::iterator it = items.begin(); it != items.end(); it++)
		if (!bag.config.findOnPath(currNode, *it) // the element hasn't already been added in
the previous step
```

```
            && !bag.config.findOnChildren(currNode, *it) // avoid adding the same element on two
different branches
            && (predictWeight(bag, currNode, *it) <= bag.maxWeight)) // check if the budget is
exceeded
            {
                  bag.config.insLastChild(currNode);
                  typename NaryTree_pointer<Item<T, N> >::node newSon =
bag.config.firstChild(currNode);
                  while (!bag.config.lastSibling(newSon))
                        newSon = bag.config.nextSibling(newSon);
                  newSon->setValue(*it);
                  buildTree(bag, items, newSon);
            }
}

template<class T, class N>
N predictWeight(Bag<T, N>& bag, typename NaryTree_pointer<Item<T, N> >::node currNode, Item<T, N>&
item)
{
      N counter = N();

      counter += currNode->getValue().weight;

      while (currNode != bag.config.root())
      {
            currNode = bag.config.parent(currNode);
            counter += currNode->getValue().weight;
      }

      counter += item.weight;

      return counter;
}

template<class T, class N>
vector<vector<Item<T, N> > > findBestConfig(Bag<T, N>& bag)
{
      vector<vector<Item<T, N> > > solutions;
      vector<typename NaryTree_pointer<Item<T, N> >::node> bagNodes = bag.config.nodesArray();
      vector<T> sums;

      for (typename vector<typename NaryTree_pointer<Item<T, N> >::node>::iterator it =
bagNodes.begin();
                  it != bagNodes.end(); it++)
            if (bag.config.is_leaf(*it)) // for each path
                  sums.push_back(getValuesSum(bag, *it)); // store the sum

      cout << "Found sums: ";
      for (typename vector<T>::iterator it = sums.begin(); it != sums.end(); it++)
            cout << *it << " ";
      cout << endl;

      bag.maxSum = sums.front();
      for (typename vector<T>::iterator it = sums.begin(); it != sums.end(); it++) // find the
highest sum
            if (bag.maxSum < *it)
                  bag.maxSum = *it;

      cout << endl << "Best sum: " << bag.maxSum;

      for (typename vector<typename NaryTree_pointer<Item<T, N> >::node>::iterator it =
bagNodes.begin();
```

```
			it != bagNodes.end(); it++) // for each node of the tree
		if (bag.config.is_leaf(*it) && (getValuesSum(bag, *it) == bag.maxSum)) // if is a leaf
and is the best path
			solutions.push_back(bag.config.pathElements(*it)); // store the paths with
highest sum

	return solutions;
}

template<class T, class N>
T getValuesSum(Bag<T, N>& bag, typename NaryTree_pointer<Item<T, N> >::node currNode)
{
	T counter = T();

	counter += currNode->getValue().value;

	while (currNode != bag.config.root())
	{
		currNode = bag.config.parent(currNode);
		counter += currNode->getValue().value;
	}

	return counter;
}

template<class T, class N>
void erasePermutations(vector<vector<Item<T, N> > >& solutions)
{
	vector<vector<Item<T, N> > > differentSolutions;
	int counter = 0;

	typename vector<vector<Item<T, N> > >::iterator it = solutions.begin();

	differentSolutions.push_back(*it); // the first is always good
	for (it++; it != solutions.end(); it++) // from the second to the last solution
	{
		for (typename vector<vector<Item<T, N> > >::iterator it2 = differentSolutions.begin();
				it2 != differentSolutions.end(); it2++)
			counter += isPermutation(*it, *it2); // check if it's a permutation of one of
the solutions in the vector

		if (counter == 0) // it's not a permutation
			differentSolutions.push_back(*it);

		counter = 0;
	}

	while (!solutions.empty())
		solutions.pop_back();

	solutions = differentSolutions;
}

template<class T, class N>
bool isPermutation(vector<Item<T, N> >& v1, vector<Item<T, N> >& v2)
{
	if (v1.size() != v2.size())
		return false;

	unsigned int counter = 0;

	for (typename vector<Item<T, N> >::iterator it = v1.begin(); it != v1.end(); it++)
```

```
		for (typename vector<Item<T, N> >::iterator it2 = v2.begin(); it2 != v2.end(); it2++)
			counter += (*it == *it2); // count the number of elements in common

	if (counter != v1.size()) // if it's a permutation, the number of elements in common is =
v1.size()
		return false;

	return true;
}

template<class T, class N>
vector<Item<T, N> > findGreedySolution(Bag<T, N> bag, vector<Item<T, N> > items)
{
	std::sort(items.begin(), items.end()); // check out operator < for items to understand sort

	N currWeight = N();

	vector<Item<T, N> > solution;

	for (typename vector<Item<T, N> >::iterator it = items.begin(); it != items.end(); it++)
		if ((currWeight + (*it).weight) <= bag.maxWeight)
		{
			currWeight += (*it).weight;
			solution.push_back(*it);
		}

	return solution;
}

#endif // _KNAPSACK_H
```

Problema dello zaino (backtracking e greedy)
File: “Bag.h”

```
#ifndef _BAG_H
#define _BAG_H
#include "N-aryTree_pointer.h"
#include "Item.h"

template<class T, class N>
class Bag
{
public:
	typedef T value_type;
	typedef N weight_type;
	Bag();
	Bag(const Bag<T, N>&);
	~Bag();
	Bag<T, N>& operator =(const Bag<T, N>&);
	NaryTree_pointer<Item<T, N> > config;
	weight_type maxWeight;
	value_type maxSum;
};

template<class T, class N>
Bag<T, N>::Bag()
{
	Cell<Item<T, N> > emptyItem;
	config.ins_root(&emptyItem);
	maxWeight = weight_type();
	maxSum = value_type();
}

template<class T, class N>
Bag<T, N>::Bag(const Bag<T, N>& bag2)
{
	*this = bag2;
}

template<class T, class N>
Bag<T, N>::~Bag()
{
	if (!config.empty())
		config.erase(config.root());

	maxWeight = weight_type();
	maxSum = value_type();
}

template<class T, class N>
Bag<T, N>& Bag<T, N>::operator =(const Bag<T, N>& bag)
{
	if (&bag != this) // avoid auto-assignment
	{
		config = bag.config;
		maxWeight = bag.maxWeight;
		maxSum = bag.maxSum;
	}
	return *this;
}
#endif // _BAG_H
```

Problema dello zaino (backtracking e greedy)
File: “Item.h”

```
#ifndef _BAGITEM_H
#define _BAGITEM_H

#include <string>
using std::string;

template<class T, class N>
class Item
{
public:
	typedef T value_type;
	typedef N weight_type;

	string name;
	value_type value;
	weight_type weight;
	int ID;

	Item();
	Item(const Item<T, N>&);
	~Item();

	Item<T, N>& operator =(const Item<T, N>&);
	bool operator ==(const Item<T, N>&) const;
	bool operator !=(const Item<T, N>&) const;

	bool operator <(const Item<T, N>&) const; // useful for greedy (to sort items)
};

template<class T, class N>
ostream& operator <<(ostream& out, const Item<T, N>& item)
{
	out << "Name: " << item.name << " - Value: " << item.value << " - Weight: " << item.weight <<
" - ID: " << item.ID;
	return out;
}

template<class T, class N>
Item<T, N>::Item()
{
	name = string();
	value = value_type();
	weight = weight_type();
	ID = 0;
}

template<class T, class N>
Item<T, N>::Item(const Item<T, N>& item2)
{
	*this = item2;
}

template<class T, class N>
Item<T, N>::~Item()
{
	name = string();
	value = value_type();
```

```
        weight = weight_type();
        ID = 0;
}

template<class T, class N>
Item<T, N>& Item<T, N>::operator =(const Item<T, N>& item2)
{
        if (&item2 != this) // avoid auto-assignment
        {
                name = item2.name;
                value = item2.value;
                weight = item2.weight;
                ID = item2.ID;
        }
        return *this;
}

template<class T, class N>
bool Item<T, N>::operator ==(const Item<T, N>& i2) const
{
        if (name != i2.name)
                return false;
        if (value != i2.value)
                return false;
        if (weight != i2.weight)
                return false;
        if (ID != i2.ID)
                return false;

        return true;
}

template<class T, class N>
bool Item<T, N>::operator !=(const Item<T, N>& i2) const
{
        return (!(*this == i2));
}

template<class T, class N>
bool Item<T, N>::operator <(const Item<T, N>& i2) const
{
        return !((this->value / this->weight) < (i2.value / i2.weight));
}

#endif // _BAGITEM_H
```

Problema dello zaino – Tester

```
#include "Knapsack.h"

int main()
{
	Bag<int, int> bag;
	vector<Item<int, int> > items;

	bag.maxWeight = 30;

	// procedure to acquire items via keyboard
	// getItems(items);

	Item<int, int> item1;
	Item<int, int> item2;
	Item<int, int> item3;
	Item<int, int> item4;
	Item<int, int> item5;
	Item<int, int> item6;

	item1.name = "Nokia Lumia";
	item1.value = 200;
	item1.weight = 30;
	item1.ID = 1;

	item2.name = "iPad Mini";
	item2.value = 150;
	item2.weight = 20;
	item2.ID = 2;

	item3.name = "HTC";
	item3.value = 130;
	item3.weight = 10;
	item3.ID = 3;

	item4.name = "iPhone";
	item4.value = 100;
	item4.weight = 10;
	item4.ID = 4;

	item5.name = "Samsung Galaxy";
	item5.value = 120;
	item5.weight = 10;
	item5.ID = 5;

	item6.name = "Blackberry";
	item6.value = 310;
	item6.weight = 21;
	item6.ID = 6;

	items.push_back(item1);
	items.push_back(item2);
	items.push_back(item3);
	items.push_back(item4);
	items.push_back(item5);
	items.push_back(item6);

	printItems(items);
```

```
	buildTree(bag, items, bag.config.root());
	cout << endl << "Bag configurations tree: " << endl << bag.config << endl;

	vector<vector<Item<int, int> > > solutions = findBestConfig(bag);

	erasePermutations(solutions);

	cout << endl << "Best solution(s) is(are):" << endl;

	for (vector<vector<Item<int, int> > >::iterator it = solutions.begin(); it != 
solutions.end(); it++)
	{
		for (vector<Item<int, int> >::iterator it2 = (*it).begin(); it2 != (*it).end(); it2++)
			if (*it2 != Item<int, int>()) // hides the root
				cout << *it2 << endl;

		cout << endl;
	}

	vector<Item<int, int> > greedySolution;
	bag.maxSum = 0;
	greedySolution = findGreedySolution(bag, items);
	cout << endl << "Greedy solution is:" << endl;
	for (vector<Item<int, int> >::iterator it = greedySolution.begin(); it != 
greedySolution.end(); it++)
		cout << *it << endl;

	cout << endl;
}
```

Problema dello zaino – Output tester

```
Items:
Nokia Lumia - Value: 200 - Weight: 30 - ID: 1
iPad Mini - Value: 150 - Weight: 20 - ID: 2
HTC - Value: 130 - Weight: 10 - ID: 3
iPhone - Value: 100 - Weight: 10 - ID: 4
Samsung Galaxy - Value: 120 - Weight: 10 - ID: 5
Blackberry - Value: 310 - Weight: 21 - ID: 6

Bag configurations tree:
[ Name:  - Value: 0 - Weight: 0 - ID: 0: [ Name: Nokia Lumia - Value: 200 - Weight: 30 -
ID: 1 ], [ Name: iPad Mini - Value: 150 - Weight: 20 - ID: 2: [ Name: HTC - Value: 130 -
Weight: 10 - ID: 3 ], [ Name: iPhone - Value: 100 - Weight: 10 - ID: 4 ], [ Name: Samsung
Galaxy - Value: 120 - Weight: 10 - ID: 5 ] ], [ Name: HTC - Value: 130 - Weight: 10 - ID:
3: [ Name: iPad Mini - Value: 150 - Weight: 20 - ID: 2 ], [ Name: iPhone - Value: 100 -
Weight: 10 - ID: 4: [ Name: Samsung Galaxy - Value: 120 - Weight: 10 - ID: 5 ] ], [ Name:
Samsung Galaxy - Value: 120 - Weight: 10 - ID: 5: [ Name: iPhone - Value: 100 - Weight: 10
- ID: 4 ] ] ], [ Name: iPhone - Value: 100 - Weight: 10 - ID: 4: [ Name: iPad Mini -
Value: 150 - Weight: 20 - ID: 2 ], [ Name: HTC - Value: 130 - Weight: 10 - ID: 3: [ Name:
Samsung Galaxy - Value: 120 - Weight: 10 - ID: 5 ] ], [ Name: Samsung Galaxy - Value: 120
- Weight: 10 - ID: 5: [ Name: HTC - Value: 130 - Weight: 10 - ID: 3 ] ] ], [ Name: Samsung
Galaxy - Value: 120 - Weight: 10 - ID: 5: [ Name: iPad Mini - Value: 150 - Weight: 20 -
ID: 2 ], [ Name: HTC - Value: 130 - Weight: 10 - ID: 3: [ Name: iPhone - Value: 100 -
Weight: 10 - ID: 4 ] ], [ Name: iPhone - Value: 100 - Weight: 10 - ID: 4: [ Name: HTC -
Value: 130 - Weight: 10 - ID: 3 ] ] ], [ Name: Blackberry - Value: 310 - Weight: 21 - ID:
6 ] ]

Found sums: 200 280 250 270 280 350 350 250 350 350 270 350 350 310

Best sum: 350
Best solution(s) is(are):
Name: Samsung Galaxy - Value: 120 - Weight: 10 - ID: 5
Name: iPhone - Value: 100 - Weight: 10 - ID: 4
Name: HTC - Value: 130 - Weight: 10 - ID: 3


Greedy solution is:
Name: Blackberry - Value: 310 - Weight: 21 - ID: 6
```

Problema delle regine

```
#ifndef NQUEEN_H_
#define NQUEEN_H_

#include <vector>
#include <iostream>
#include <cstdlib>
#include <stdexcept>

using std::vector;
using std::cout;
using std::endl;

struct Board
{
	int n_queens;
	vector<int> board;

	Board(int n = 4)
	{
		if (n > 20)
			n_queens = 20;
		else
			n_queens = n;

		board.resize(n_queens);

		board[0] = 0;

	}

	~Board()
	{
	}

	void printBoard()
	{
		static int num_solutions = 1;

		cout << "Solution # " << num_solutions << ":" << endl;
		for (int i = 0; i < n_queens; i++)
		{
			for (int j = 0; j < n_queens; j++)
				if (board[j] == i)
					cout << "Q ";
				else
					cout << "+ ";
			cout << endl;
		}
		cout << endl;

		num_solutions++;
	}
};

bool underAttack(int i, int j, int col, const Board& b)
{
	return (b.board[j] == i || abs(b.board[j] - i) == abs(col - j));
}
```

```
void insertQueens(Board& b, int col)
{

	if (col == b.n_queens)
		b.printBoard();
	else
	{
		int i, j;

		for (i = 0; i < b.n_queens; i++)
		{
			for (j = 0; j < col && !underAttack(i, j, col, b); j++) // Search the first
available free position on that row
				;

			if (j >= col) // The queen isn't under attack
			{
				b.board[col] = i; // Insert the queen
				insertQueens(b, col + 1); // Try to add another queen
			}
		}
	}
}

void generateSolutions(Board& b)
{
	insertQueens(b, 0);
}

#endif /* NQUEEN_H_ */
```

Problema delle regine – Tester

```
#include "nqueen.h"

int main()
{
	Board b(6);

	generateSolutions(b);

}
```

Problema delle regine – Output tester

```
Solution # 1:
+ + + Q + +
Q + + + + +
+ + + + Q +
+ Q + + + +
+ + + + + Q
+ + Q + + +

Solution # 2:
+ + + + Q +
+ + Q + + +
Q + + + + +
+ + + + + Q
+ + + Q + +
+ Q + + + +

Solution # 3:
+ Q + + + +
+ + + Q + +
+ + + + + Q
Q + + + + +
+ + Q + + +
+ + + + Q +

Solution # 4:
+ + Q + + +
+ + + + + Q
+ Q + + + +
+ + + + Q +
Q + + + + +
+ + + Q + +
```

Risolutore di labirinti

```
#ifndef _LABYRINTH_H_
#define _LABYRINTH_H_

#include "Graph_matrix.h"
#include <fstream>

using std::list;
using std::endl;
using std::cout;
using std::ios;
using std::fstream;
using std::vector;

vector<vector<char> > analyzePattern()
{
	fstream f1;
	f1.open("pattern.txt", ios::in); // open the file in read mode

	vector<vector<char> > matrix;

	if (!f1)
		throw std::logic_error("Unable to open pattern.txt");

	vector<char> temp;
	char c;

	while (!f1.eof())
	{
		c = f1.get();
		if (c != '\n' && c != ' ') // ignore blanks
			temp.push_back(c);
		else if (c == '\n')
		{
			matrix.push_back(temp);
			temp.clear();
		}
	}
	f1.close();

	cout << "Pattern acquired: " << endl;
	for (vector<vector<char> >::iterator it = matrix.begin(); it != matrix.end(); it++)
	{
		for (vector<char>::iterator it2 = (*it).begin(); it2 != (*it).end(); it2++)
			cout << *it2;

		cout << endl;
	}
	cout << endl;

	return matrix;
}

vector<vector<int> > convertPatternToID(vector<vector<char> >& matrix, int& counter)
{
	vector<vector<int> > matrixID;

	for (vector<vector<char> >::iterator it = matrix.begin(); it != matrix.end(); it++)
	{
```

```
		vector<int> temp;
		for (vector<char>::iterator it2 = (*it).begin(); it2 != (*it).end(); it2++)
		{
			if ((*it2) == '0')
			{
				temp.push_back(counter);
				counter++;
			}
			else
				temp.push_back(-1);
		}
		matrixID.push_back(temp);
		temp.clear();
	}

	cout << "Pattern converted to ID: " << endl;
	for (vector<vector<int> >::iterator it = matrixID.begin(); it != matrixID.end(); it++)
	{
		for (vector<int>::iterator it2 = (*it).begin(); it2 != (*it).end(); it2++)
			cout << *it2 << " ";

		cout << endl;
	}

	cout << endl;

	return matrixID;
}

template<class V, class W>
void buildGraph(Graph_matrix<V, W>& graph, vector<vector<int> > matrixID)
{
	for (vector<vector<int> >::iterator it = matrixID.begin(); it != matrixID.end(); it++)
		for (vector<int>::iterator it2 = (*it).begin(); it2 != (*it).end(); it2++)
			if ((*it2) != -1)
			{
				Node temp;
				graph.insNode(temp);
				graph.setLabel((graph.nodesList()).back(), (*it2));
			}

	int width = (matrixID.front()).size();
	int height = matrixID.size();

	// now, let's add the edges

	for (int i = 0; i < height; i++)
		for (int j = 0; j < width; j++)
			if (matrixID[i][j] != -1)
			{
				// check if I've to add an edge towards left
				if (j != 0 && matrixID[i][j - 1] != -1)
				{
					graph.insEdge(findValue(graph, matrixID[i][j]), findValue(graph,
matrixID[i][j - 1]));
					graph.setWeight(findValue(graph, matrixID[i][j]),
findValue(graph, matrixID[i][j - 1]), 1);
				}

				// check if I've to add an edge towards right
				if (j != (width - 1) && matrixID[i][j + 1] != -1)
				{
```

```
                                    graph.insEdge(findValue(graph, matrixID[i][j]), findValue(graph,
matrixID[i][j + 1]));
                                    graph.setWeight(findValue(graph, matrixID[i][j]),
findValue(graph, matrixID[i][j + 1]), 1);
                             }

                             // check if I've to add an edge towards up
                             if (i != 0 && matrixID[i - 1][j] != -1)
                             {
                                    graph.insEdge(findValue(graph, matrixID[i][j]), findValue(graph,
matrixID[i - 1][j]));
                                    graph.setWeight(findValue(graph, matrixID[i][j]),
findValue(graph, matrixID[i - 1][j]), 1);
                             }

                             // check if I've to add an edge towards down
                             if (i != (height - 1) && matrixID[i + 1][j] != -1)
                             {
                                    graph.insEdge(findValue(graph, matrixID[i][j]), findValue(graph,
matrixID[i + 1][j]));
                                    graph.setWeight(findValue(graph, matrixID[i][j]),
findValue(graph, matrixID[i + 1][j]), 1);
                             }
                      }

//     cout << "The graph has been built:";
//     cout << endl;
//     cout << graph;

}

template<class N, class L, class W>
typename Graph<N, L, W>::node findValue(Graph<N, L, W>& graph, L value)
{
       list<typename Graph<N, L, W>::node> temp = graph.nodesList();

       for (typename list<typename Graph<N, L, W>::node>::iterator it = temp.begin(); it !=
temp.end(); it++)
              if (graph.getLabel(*it) == value)
                     return (*it);

       throw std::logic_error("Unable to retrieve value");
}

void printSolution(Graph_matrix<int, int> graph, list<Graph_matrix<int, int>::node> solution,
vector<vector<int> > matrixID)
{
       int width = (matrixID.front()).size();
       int height = matrixID.size();
       bool flag = false;

       for (int i = 0; i < height; i++)
       {
              for (int j = 0; j < width; j++)
                     if (matrixID[i][j] == -1)
                            cout << "- ";
                     else
                     {
                            for (list<Graph_matrix<int, int>::node>::iterator it = solution.begin();
                                          (it != solution.end() && !flag); it++)
                                   if (graph.getLabel((*it)) == matrixID[i][j])
                                   {
```

```
                    cout << "X ";
                    flag = true;
                }

            if (!flag)
                cout << "0 ";

            flag = false;
        }

        cout << endl;
    }
}

#endif /* _LABYRINTH_H_ */
```

Risolutore di labirinti – Tester

```
#include <iostream>
#include "Graph_matrix.h"
#include "labyrinth.h"

/* PRE-CONDITION ON "pattern.txt"
 *
 * Labyrinth must have a rectangular shape;
 * Use - to build the walls;
 * Blanks can be used;
 * Add a \n at the end of the file;
 * Use 0 to indicate a legit tile */


int main()
{
	vector<vector<char> > matrix = analyzePattern();
	int counter = 0; // number of nodes in the graph
	vector<vector<int> > matrixID = convertPatternToID(matrix, counter);
	Graph_matrix<int, int> graph;

	buildGraph(graph, matrixID);

	Graph_matrix<int, int>::node startingNode = findValue(graph, 0);
	Graph_matrix<int, int>::node endingNode = findValue(graph, counter - 1);

	if (graph.isPath(startingNode, endingNode))
	{
		cout << "Labyrinth can be solved." << endl << endl;
		cout << "Please wait, I'm calculating the best solution... ";
		list<Graph_matrix<int, int>::node> solution = graph.getShortestPath(startingNode,
endingNode);
		cout << endl << endl << "One possible best solution is (look at X to follow the
path):" << endl << endl;
		printSolution(graph, solution, matrixID);
	}
	else
		cout << "Labyrinth can't be solved.";

}
```

Risolutore di labirinti – Tester
File: “pattern.txt”

```
- - - - - - 0 - - - - -
- 0 - 0 0 0 0 0 0 0 0 -
- 0 - 0 - 0 0 0 - - 0 -
- 0 - 0 - 0 0 - 0 0 0 -
- 0 - 0 - 0 0 - 0 0 0 -
- 0 - - - 0 - - 0 - 0 -
- 0 0 0 0 0 0 - 0 0 0 -
- 0 - - 0 - 0 - 0 0 - -
- 0 - - - 0 - - 0 - 0 -
- 0 0 0 0 0 0 - 0 0 0 -
- 0 - - 0 - 0 - 0 0 - -
- - - 0 0 - 0 - 0 - - -
- 0 - 0 0 - 0 - 0 0 - -
- 0 0 0 - - 0 - - 0 - -
- 0 - 0 0 0 0 0 0 - - -
- 0 0 - - 0 0 0 0 0 - -
- - - - - - - - - 0 - -
```

Risolutore di labirinti – Output tester

```
Pattern acquired:
------0-----
-0-00000000-
-0-0-000--0-
-0-0-00-000-
-0-0-00-000-
-0---0--0-0-
-000000-000-
-0--0-0-00--
-0---0--0-0-
-000000-000-
-0--0-0-00--
---00-0-0---
-0-00-0-00--
-000--0--0--
-0-000000---
-00--00000--
---------0--
```

```
Pattern converted to ID:
-1 -1 -1 -1 -1 -1 0 -1 -1 -1 -1 -1
-1 1 -1 2 3 4 5 6 7 8 9 -1
-1 10 -1 11 -1 12 13 14 -1 -1 15 -1
-1 16 -1 17 -1 18 19 -1 20 21 22 -1
-1 23 -1 24 -1 25 26 -1 27 28 29 -1
-1 30 -1 -1 -1 31 -1 -1 32 -1 33 -1
-1 34 35 36 37 38 39 -1 40 41 42 -1
-1 43 -1 -1 44 -1 45 -1 46 47 -1 -1
-1 48 -1 -1 -1 49 -1 -1 50 -1 51 -1
-1 52 53 54 55 56 57 -1 58 59 60 -1
-1 61 -1 -1 62 -1 63 -1 64 65 -1 -1
-1 -1 -1 66 67 -1 68 -1 69 -1 -1 -1
-1 70 -1 71 72 -1 73 -1 74 75 -1 -1
-1 76 77 78 -1 -1 79 -1 -1 80 -1 -1
-1 81 -1 82 83 84 85 86 87 -1 -1 -1
-1 88 89 -1 -1 90 91 92 93 94 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 95 -1 -1

Labyrinth can be solved.

Please wait, I'm calculating the best solution...

One possible best solution is (look at X to follow the path):

- - - - - - X - - - - -
- 0 - 0 0 X X 0 0 0 0 -
- 0 - 0 - X 0 0 - - 0 -
- 0 - 0 - X 0 - 0 0 0 -
- 0 - 0 - X 0 - 0 0 0 -
- 0 - - - X - - 0 - 0 -
- X X X X X 0 - 0 0 0 -
- X - - 0 - 0 - 0 0 - -
- X - - - 0 - - 0 - 0 -
- X X X X X X - 0 0 0 -
- 0 - - 0 - X - 0 0 - -
- - - 0 0 - X - 0 - - -
- 0 - 0 0 - X - 0 0 - -
- 0 0 0 - - X - - 0 - -
- 0 - 0 0 0 X X X - - -
- 0 0 - - 0 0 0 X X - -
- - - - - - - - - X - -
```