

Realizzazione Alberi N-ari

Laboratorio di Algoritmi e Strutture Dati

Marco de Gemmis
degemmis@di.uniba.it

**Materiale di base gentilmente concesso
dal dott. Nicola Di Mauro
Ricercatore presso l'Univ. di Bari
Adattamenti di Marco de Gemmis**

Rappresentazione con puntatori

La Classe Cella

```
#ifndef _CELLA_H
#define _CELLA_H
// stessa classe di ADT albero binario
// diversa denominazione dei campi
template <class T> class Cella
{
public:
    typedef T tipoelem;
    Cella();
    Cella(tipoelem);
    void setElemento(tipoelem);
    tipoelem getElemento() const;
    void setPRIMOFIGLIO(Cella<T>*); // primo figlio
    void setPROXFRATELLO(Cella<T>*); // fratello più prossimo
    void setPADRE(Cella<T>*); // padre
    Cella<T>* getPRIMOFIGLIO() const;
    Cella<T>* getPROXFRATELLO() const;
    Cella<T>* getPADRE() const;
    bool operator ==(Cella<T>);
    bool operator <=(Cella<T>);
private:
    tipoelem elemento;
    Cella<T>* PRIMOFIGLIO;
    Cella<T>* PROXFRATELLO;
    Cella<T>* PADRE;
};
```

Rappresentazione con puntatori

La Classe Cella

```
// Implementazione della classe Cella
// costruttori
template <class T> Cella<T>::Cella(){
    PRIMOFIGLIO=NULL;
    PROXFRATELLO=NULL;
    PADRE=NULL;
}
template <class T> Cella<T>::Cella(tipoelem e)
{
    elemento = e;
}
template <class T> void Cella<T>::setElemento(tipoelem label)
{
    elemento = label;
}
template <class T> T Cella<T>::getElemento() const
{
    return elemento;
}
template <class T> void Cella<T>::setPRIMOFIGLIO(Cella<T>* p)
{
    PRIMOFIGLIO=p;
}

template <class T> void Cella<T>::setPROXFRATELLO(Cella<T>* p)
{
    PROXFRATELLO=p;
}

template <class T> void Cella<T>::setPADRE(Cella<T>* p)
{
    PADRE=p;
}
```

Rappresentazione con puntatori

La Classe Cella

```
template <class T> Cella<T>* Cella<T>::getPRIMOFIGLIO() const
{
    return PRIMOFIGLIO;
}

template <class T> Cella<T>* Cella<T>::getPROXFRATELLO() const
{
    return PROXFRATELLO;
}

template <class T> Cella<T>* Cella<T>::getPADRE() const
{
    return PADRE;
}

// sovraccarico dell'operatore ==
template <class T> bool Cella<T>::operator==(Cella<T> cella)
{
    return (getElemento == cella.getElemento);
}

template <class T> bool Cella<T>::operator<=(Cella<T> cella)
{
    return (getElemento <= cella.getElemento);
}

#endif // _CELLA_H
```

Rappresentazione con puntatori

La Classe AlberoN

```
#ifndef _ALBERON_H_
#define _ALBERON_H_
#include "Cella.h"
#include <cassert>
#include <iostream>
using namespace std;

template <class T> class Albero {
public:
// tipi
typedef T tipoelem;
typedef Cella<T>* Nodo;

// costruttori e distruttori
Albero();
~Albero();
// operatori
void creaAlbero();
bool alberoVuoto();
void insRadice(Nodo);
Nodo radice();
Nodo padre(Nodo);
bool foglia(Nodo);
Nodo primoFiglio(Nodo);
bool ultimoFratello(Nodo);
Nodo succFratello(Nodo);
void insPrimoSottoalbero(Nodo,Albero<T>&);
void insSottoalbero(Nodo,Albero<T>&);
void cancSottoalbero(Nodo); //
// operazioni di lettura / scrittura sui nodi
void scriviNodo(Nodo,tipoelem);
tipoelem leggiNodo(Nodo);
```

Rappresentazione con puntatori

La Classe AlberoN

```
// servizio
void stampa(Nodo,int); // stampa il sottoalbero di radice Nodo posto a livello int
// gli algoritmi di visita sono gli stessi degli alberi binari

private:
bool appartiene(Nodo);
Nodo root; // un albero è identificato dalla sua radice
};
```

Implementazione

```
template <class T> Albero<T>::Albero()
{creaAlbero();}

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

template <class T> void Albero<T>::creaAlbero()
{
    root=NULL;
}

template <class T> bool Albero<T>::alberoVuoto()
{
    return (root==NULL);
}

template <class T> void Albero<T>::insRadice(Nodo n)
{
    assert (alberoVuoto());
    root=n;
    root->setPRIMOFIGLIO(NULL); // non ha figli
    root->setPROXFRATELLO(NULL); // non ha fratelli
    root->setPADRE(NULL); // non ha genitore
}

template <class T> Cella<T>* Albero<T>::radice()
{
    assert (!alberoVuoto());
    return (root);
}

template <class T> Cella<T>* Albero<T>::padre(Nodo n)
{
    assert (n!=radice());
    assert (appartiene(n));
    return (n->getPADRE());
}
```

Implementazione

```
template <class T> bool Albero<T>::appartiene(Nodo n)
```

```
{  
    // va implementato il controllo se il nodo  
    // appartiene all'albero  
    return (true);  
}
```

```
template <class T> bool Albero<T>::foglia(Nodo n)
```

```
{  
    assert(appartiene(n));  
    return (n->getPRIMOFIGLIO()==NULL); // NODO FOGLIA SE NON HA FIGLI  
}
```

```
template <class T> Cella<T>* Albero<T>::primoFiglio(Nodo n)
```

```
{  
    assert (appartiene(n));  
    assert (n->getPRIMOFIGLIO()!=NULL);  
    return (n->getPRIMOFIGLIO());  
}
```

```
template <class T> bool Albero<T>::ultimoFratello(Nodo n)
```

```
{  
    assert (appartiene(n));  
    return (n->getPROXFRATELLO()==NULL); // ULTIMO FRATELLO SE NON HA FRATELLI  
}
```

```
template <class T> Cella<T>* Albero<T>::succFratello(Nodo n)
```

```
{  
    assert (appartiene(n));  
    assert (!ultimoFratello(n));  
    return (n->getPROXFRATELLO());  
}
```


Implementazione

```
template <class T> void Albero<T>::insPrimoSottoalbero(Nodo n, Albero<T> &a)
{
    assert (!a.alberoVuoto()); // a non è vuoto
    assert (appartiene(n)); // n è nell'albero corrente
    a.radice()->setPROXFRATELLO(n->getPRIMOFIGLIO()); // quello che era il primo figlio di n diventa fratello
della radice di a
    a.radice()->setPADRE(n); // n diventa padre della radice di a
    n->setPRIMOFIGLIO(a.radice()); // la radice di a diventa primo figlio di n
}
```

```
template <class T> void Albero<T>::insSottoalbero(Nodo n, Albero<T> &a)
{
    assert (!a.alberoVuoto()); // a non è vuoto
    assert (appartiene(n)); // n è nell'albero corrente
    assert (radice()!=n); // n è nell'albero corrente
    a.radice()->setPROXFRATELLO(n->getPROXFRATELLO()); // quello che era il fratello di n diventa fratello
della radice di a
    a.radice()->setPADRE(radice()); // n diventa padre della radice di a
    n->setPROXFRATELLO(a.radice()); // la radice di a diventa primo figlio di n
}
```

Implementazione

```
template <class T> void Albero<T>::cancSottoalbero(Nodo n)
{
    assert (appartiene(n)); // n è nell'albero corrente
    cout << "entro nella procedura per eliminare nodo: " << n->getElemento() << endl;
    // spezzo i legami tra n, il padre ed i fratelli
    if (radice()==n)
    {
        delete root;
    }
    else
    {
        if (n==primoFiglio(padre(n))) // n è il primo figlio
        {
            if (ultimoFratello(n))
                padre(n)->setPRIMOFIGLIO(NULL);
            else
                padre(n)->setPRIMOFIGLIO(succFratello(n)); // allora ha un fratello che diventa il primo figlio
        }
        else
        {
            // sono nel ramo else dunque n non è primo figlio
            Nodo prec=primoFiglio(padre(n));
            Nodo current=succFratello(prec);
            while (current!=n)
            {
                prec=current;
                current=succFratello(current);
            }
            // current=n
            if (ultimoFratello(n))
                prec->setPROXFRATELLO(NULL);
            else
                prec->setPROXFRATELLO(succFratello(n));
        }
    }
    delete n;
}
```

Implementazione

```
template <class T> void Albero<T>::scriviNodo(Nodo n, tipoelem e)
```

```
{  
    assert (appartiene(n)); // n è nell'albero corrente  
    n->setElemento(e);  
}
```

```
template <class T> T Albero<T>::leggiNodo(Nodo n)
```

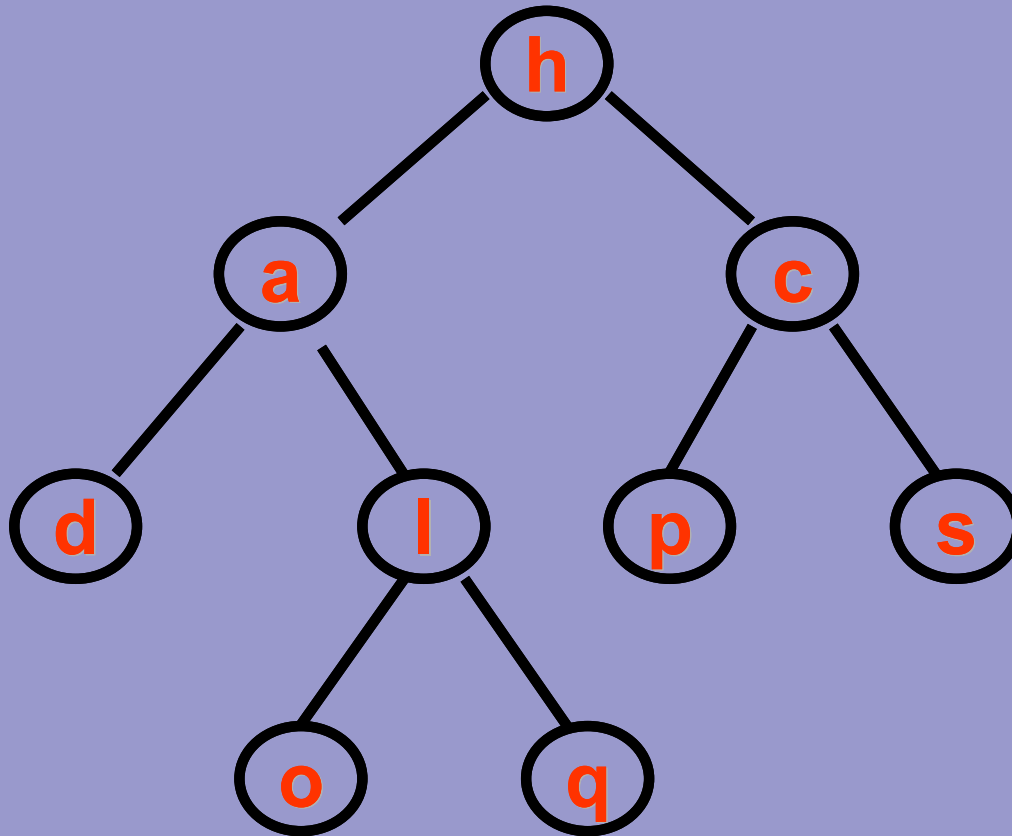
```
{  
    assert (appartiene(n)); // n è nell'albero corrente  
    return (n->getElemento());  
}
```

Implementazione

```
template <class T> void Albero<T>::stampa(Nodo n, int livello)
{
    assert(appartiene(n));
    assert(livello>=0);
    if (alberoVuoto())
    {
        cout << "albero vuoto!" << endl;
        return;
    }
    const int Nspaces=4;
    int spaces=livello*Nspaces;
    // stampo il nodo
    for (int i=1;i<spaces;i++)
        cout << " ";
    cout << leggiNodo(n) << endl; // stampo il nodo
    if (!foglia(n))
    {
        livello++; // ed i suoi eventuali figli: passo al livello successivo
        stampa(primoFiglio(n),livello);
        Nodo iter=primoFiglio(n);
        bool ancora_fratelli = !ultimoFratello(iter);
        while (ancora_fratelli)
        {
            iter=succFratello(iter);
            stampa(iter,livello);
            ancora_fratelli=!ultimoFratello(iter);
        }
    }
}
```

Algoritmi di visita

- Sono gli stessi visti per gli alberi binari



Esercizio #1

- Scrivere un programma che:
 - acquisisca un albero n-ario di elementi di tipo intero e lo stampi.
- Scrivere una funzione `somma(AlberoN Tree)` che restituisca la somma di tutti gli elementi di Tree
- Scrivere una funzione `sommaLivello(AlberoN Tree, int k)` che restituisca la somma di tutti gli elementi di livello k di Tree

Esercizio #2

- Scrivere un programma che:
 - acquisisca un albero n-ario di elementi di tipo intero e lo stampi.
- Scrivere una funzione incrementa(AlberoN Tree, int k) che modifichi Tree aggiungendo ad ogni livello un nodo che contenga la somma di tutti gli elementi di livello k di Tree ($k > 0$)
 - Sfruttare quanto realizzato nell'esercizio #1