

# Realizzazione Alberi Binari

## Laboratorio di Algoritmi e Strutture Dati

**Domenico Redavid**  
**redavid@di.uniba.it**

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

# Rappresentazione con puntatori

## La Classe Cella

```
#ifndef _CELLA_H
#define _CELLA_H

template <class T> class Cella {

public:
    typedef T tipoelem;
    Cella();
    Cella(tipoelem);
    void setElemento(tipoelem);
    tipoelem getElemento() const;
    void setSX(Cella<T>*);
    void setDX(Cella<T>*);
    void setDAD(Cella<T>*);
    Cella<T>* getSX() const;
    Cella<T>* getDX() const;
    Cella<T>* getDAD() const;
    bool operator ==(Cella<T>);
    bool operator <=(Cella<T>);
private:
    tipoelem elemento;
    Cella<T>* SX;
    Cella<T>* DX;
    Cella<T>* DAD;
};
```

# Rappresentazione con puntatori

## La Classe Cella

```
// Implementazione della classe Cella
// -----
// costruttori
template <class T> Cella<T>::Cella(){
    SX=NULL;
    DX=NULL;
    DAD=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>::setSX(Cella<T>* p)
{
    SX=p;
}

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

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

# Rappresentazione con puntatori

## La Classe Cella

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

template <class T> Cella<T>* Cella<T>::getDX() const
{
    return DX;
}

template <class T> Cella<T>* Cella<T>::getDAD() const
{
    return DAD;
}

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

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

# Rappresentazione con puntatori

## La Classe BinAlbero

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

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

// costruttori e distruttori
BinAlbero(); // OK
~BinAlbero();
// operatori
void creaBinAlbero();
bool binAlberoVuoto();
Nodo binRadice();
Nodo binPadre(Nodo);
Nodo figlioSinistro(Nodo);
Nodo figlioDestro(Nodo);
bool sinistroVuoto(Nodo);
bool destroVuoto(Nodo);
// costruzione dalle foglie alla radice
//void costrBinAlbero(BinAlbero<T>,BinAlbero<T>);
void cancSottoBinAlbero(Nodo);
T leggiNodo(Nodo);
void scriviNodo(Nodo, tipoelem);
void insBinRadice(Nodo);
void insFiglioSinistro(Nodo);
void insFiglioDestro(Nodo);
```

### LA SPECIFICA SINTATTICA

#### TIPI: ALBEROBIN, BOOLEANO, NODO

**CREABINALBERO** : ( ) → ALBEROBIN  
**BINALBEROVUOTO** : (ALBEROBIN) → BOOLEANO  
**BINRADICE** : (ALBEROBIN) → NODO  
**BINPADRE** : (NODO,ALBEROBIN) → NODO  
**FIGLIOSINISTRO** : (NODO,ALBEROBIN) → NODO  
**FIGLIODESTRO** : (NODO,ALBEROBIN) → NODO  
**SINISTROVUOTO** : (NODO,ALBEROBIN) → BOOLEANO  
**DESTROVUOTO** : (NODO,ALBEROBIN) → BOOLEANO  
**COSTRBINALBERO** : (ALBEROBIN,ALBEROBIN) → ALBEROBIN  
**CANCSOTTOBINALBERO** : (NODO,ALBEROBIN) → ALBEROBIN

### OPERATORI UTILI

*Tipi: Va aggiunto TIPOELEM del tipo dell'etichetta*

**LEGGINODO**: (NODO,ALBEROBIN) → TIPOELEM  
**SCRIVINODO**: (TIPOELEM,NODO,ALBEROBIN) → ALBEROBIN

### OPERATORI SOSTITUTIVI DI COSTRBINALBERO

**INSBINRADICE** : (NODO,ALBEROBIN) → ALBEROBIN  
**INFIGLIOSINISTRO** : (NODO,ALBEROBIN) → ALBEROBIN  
**INFIGLIODESTRO** : (NODO,ALBEROBIN) → ALBEROBIN

# Rappresentazione con puntatori

## La Classe BinAlbero

```
// SERVIZIO
void stampa(Nodo,int); // stampa il sottoalbero a partire da Nodo
int altezza (Nodo); // restituisce l'altezza di nodo
int count (Nodo); // restituisce il numero di nodi nel sottoalbero a partire da Nodo
// VISITA
void visitaPreordine(Nodo);
void visitaPostordine(Nodo);
void visitaSimmetrica(Nodo);
void visitaAmpiezza(Nodo);

private:
Nodo radice; // un albero è identificato dalla sua radice
};
```

# Implementazione

```
template <class T> BinAlbero<T>::BinAlbero()
{creaBinAlbero();}

template <class T> void BinAlbero<T>::creaBinAlbero()
{
    radice=NULL;
}

template <class T> bool BinAlbero<T>::binAlberoVuoto()
{
    return (radice==NULL);
}

template <class T> Cella<T>* BinAlbero<T>::binRadice()
{
    return (radice);
}

template <class T> Cella<T>* BinAlbero<T>::binPadre(Nodo n)
{
    if (n!=radice)
        return (n->getDAD());
    else
        return NULL;
}

template <class T> Cella<T>* BinAlbero<T>::figlioSinistro(Nodo n)
{
    return (n->getSX());
}

template <class T> Cella<T>* BinAlbero<T>::figlioDestro(Nodo n)
{
    return (n->getDX());
}
```

# Implementazione

```
template <class T> bool BinAlbero<T>::sinistroVuoto(Nodo n)
{
    return (n->getSX()==NULL);
}

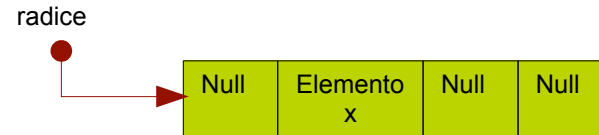
template <class T> bool BinAlbero<T>::destroVuoto(Nodo n)
{
    return (n->getDX()==NULL);
}

template <class T> void BinAlbero<T>::cancSottoBinAlbero(Nodo n)
{
    if (n!=NULL)
    {
        if (!sinistroVuoto(n))
            cancSottoBinAlbero(figlioSinistro(n));
        if (!destroVuoto(n))
            cancSottoBinAlbero(figlioDestro(n));
        if (n != radice) {
            Nodo padre = binPadre(n);
            if (figlioSinistro(padre)==n)
            {
                padre->setSX(NULL);
            }
            else
            {
                padre->setDX(NULL);
            }
        }
        else
            radice = NULL;
    }
}
```

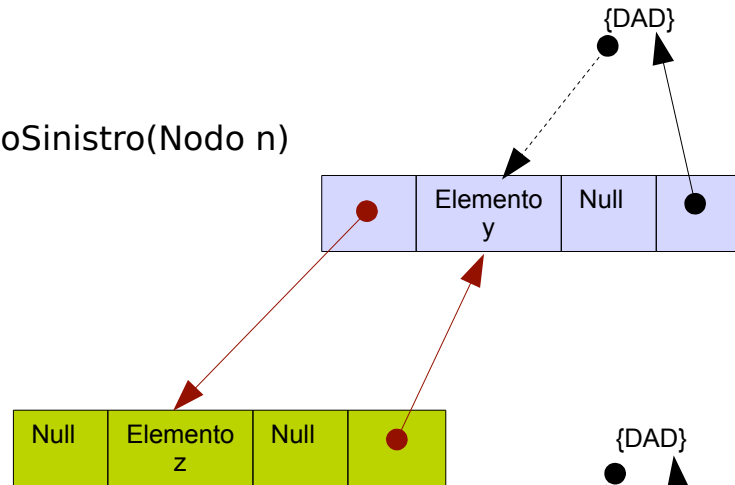


# Implementazione

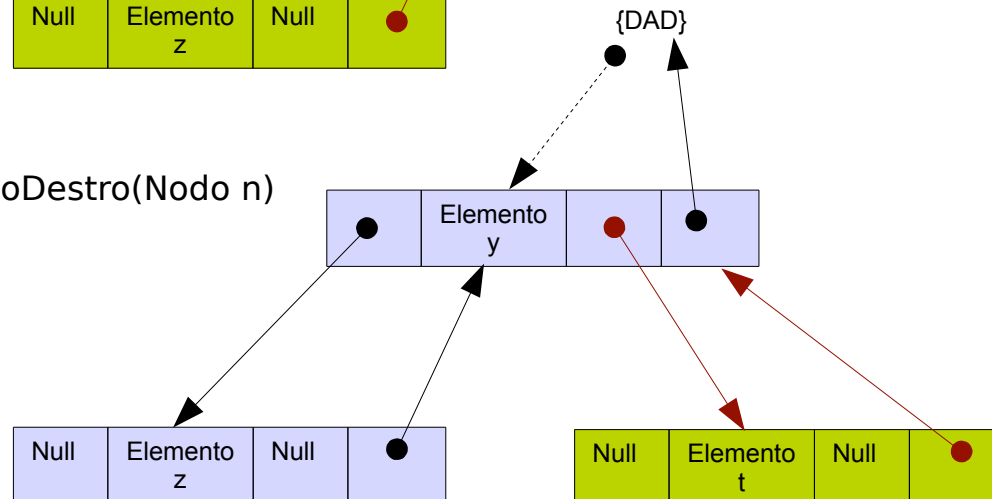
```
template <class T> void BinAlbero<T>::insBinRadice(Nodo n)
{
  if (radice == NULL)
  {
    radice=n;
    radice->setSX(NULL); // non ha figli
    radice->setDX(NULL); // non ha figli
    radice->setDAD(NULL); // non ha genitore
  }
}
```



```
template <class T> void BinAlbero<T>::insFiglioSinistro(Nodo n)
{
  if (sinistroVuoto(n))
  {
    n->setSX(new Cella<T>);
    n->getSX()->setDAD(n);
    n->getSX()->setSX(NULL);
    n->getSX()->setDX(NULL);
  }
}
```



```
template <class T> void BinAlbero<T>::insFiglioDestro(Nodo n)
{
  if (destroVuoto(n))
  {
    n->setDX(new Cella<T>);
    n->getDX()->setDAD(n);
    n->getDX()->setSX(NULL);
    n->getDX()->setDX(NULL);
  }
}
```



# Implementazione

```
template <class T> T BinAlbero<T>::leggiNodo(Nodo n)
{
    if (n != NULL)
        return (n->getElemento());
}

template <class T> void BinAlbero<T>::scriviNodo(Nodo n, tipoelem a)
{
    if (n != NULL)
        n->setElemento(a);
}

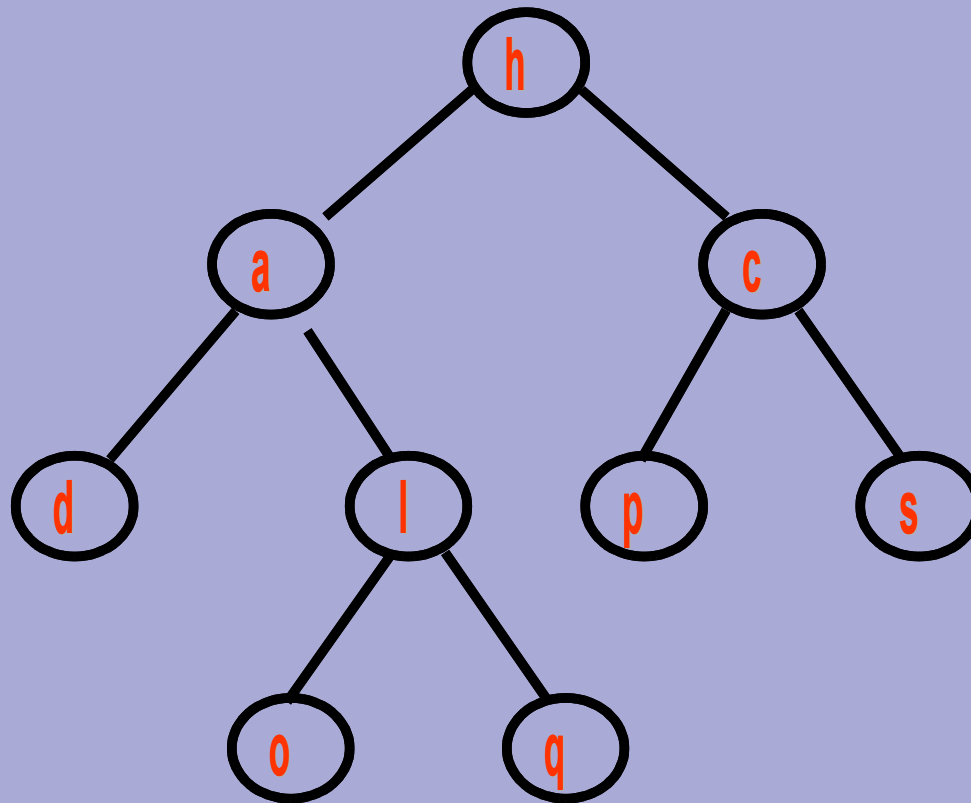
template <class T> void BinAlbero<T>::stampa(Nodo n, int livello)
{
    const int Nspaces=4;
    int spaces=livello*Nspaces;
    if (n != NULL)
    {
        for (int i=1;i<spaces;i++)
            cout << " ";
        cout << leggiNodo(n) << endl;
        livello++;
        if (!sinistroVuoto(n))
            stampa(figlioSinistro(n),livello);
        if (!destroVuoto(n))
            stampa(figlioDestro(n),livello);
    }
}
```

# Implementazione

```
template <class T> int BinAlbero<T>::altezza(Nodo n)
{
    if (sinistroVuoto(n) && destroVuoto(n)) // nodo foglia
        return 0;
    int altsx = 0;
    if (!sinistroVuoto(n))
        altsx = altezza(figlioSinistro(n));
    int altdx = 0;
    if (!destroVuoto(n))
        altdx = altezza(figlioDestro(n));
    int res=1;
    if (altsx > altdx)
        res+=altsx;
    else
        res+=altdx;
    return res;
}
```

```
template <class T> int BinAlbero<T>::count(Nodo n)
{
    int sx=0;
    int dx=0;
    if (!sinistroVuoto(n))
        sx = count(figlioSinistro(n));
    if (!destroVuoto(n))
        dx = count(figlioDestro(n));
    return (sx + dx + 1);
}
```

# Algoritmi di visita



# Algoritmi di visita

- Visita in pre-ordine (**h a d l o q c p s**)
  - Si applica ad un albero non vuoto e richiede dapprima l'analisi della radice dell'albero e, poi, la visita, effettuata con lo stesso metodo, dei due sottoalberi, prima il sinistro, poi il destro
- Visita in post-ordine (**d o q l a p s c h**)
  - Si applica ad un albero non vuoto e richiede dapprima la visita, effettuata con lo stesso metodo, dei sottoalberi, prima il sinistro e poi il destro, e, in seguito, l'analisi della radice dell'albero
- Visita simmetrica (**d a o l q h p c s**)
  - richiede dapprima la visita del sottoalbero sinistro (effettuata sempre con lo stesso metodo), poi l'analisi della radice, e poi la visita del sotto albero destro

# Visita per stampa

```
template <class T> void BinAlbero<T>::visitaPreordine(Nodo n)
{
    if (n!=NULL)
    {
        cout << leggiNodo(n) << " "; // esame del nodo
        visitaPreordine(figlioSinistro(n));
        visitaPreordine(figlioDestro(n));
    }
}
```

```
template <class T> void BinAlbero<T>::visitaPostordine(Nodo n)
{
    if (n!=NULL)
    {
        visitaPostordine(figlioSinistro(n));
        visitaPostordine(figlioDestro(n));
        cout << leggiNodo(n) << " "; // esame del nodo
    }
}
```

```
template <class T> void BinAlbero<T>::visitaSimmetrica(Nodo n)
{
    if (n!=NULL)
    {
        visitaSimmetrica(figlioSinistro(n));
        cout << leggiNodo(n) << " "; // esame del nodo
        visitaSimmetrica(figlioDestro(n));
    }
}
```

# Attraversamento iterativo in ampiezza

```
template <class T> void BinAlbero<T>::visitaAmpiezza(Nodo n)
{
    if (n!=NULL)
    {
        Coda<Nodo> s;
        s.inCoda(n);
        while (!s.codaVuota()){
            Nodo nn=s.leggiCoda(); // visito un nodo ...
            cout << leggiNodo(nn) << " ";
            s.fuoriCoda();
            if (!sinistroVuoto(nn)) s.inCoda(figlioSinistro(nn)); // ... e accodo i figli
            if (!destraVuoto(nn)) s.inCoda(figlioDestra(nn));
        }
    }
}
```

# Esercizio

- Scrivere un programma che:
  - acquisisca un albero binario di elementi di tipo intero e lo stampi. Per ogni nodo:
  - se e' un figlio destro, incrementi di 1 il valore dei figli
  - se e' un figlio sinistro, scambi i valori dei figli (se ci sono entrambi)
  - stampi l'albero risultante



# Svolgimento: il Main

```
int main(int argc, char *argv[])
{
    cout << "albero input" << endl;
    BinAlbero<int> tree = acquisisci();
    tree.stampa(tree.binRadice(),0);
    cout << "Processing..." << endl;
    visita(tree);
    cout << "albero output" << endl;
    tree.stampa(tree.binRadice(),0);

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

# Svolgimento: acquisisci()

```
BinAlbero<int> acquisisci()  
{  
    BinAlbero<int> tree;  
    BinAlbero<int>::Nodo radice = new Cella<int>;  
    tree.insBinRadice(radice);  
    tree.scriviNodo(tree.binRadice(),1);  
    tree.insFiglioSinistro(radice);  
    tree.insFiglioDestro(radice);  
    tree.scriviNodo(tree.figlioSinistro(tree.binRadice()),200);  
    tree.scriviNodo(tree.figlioDestro(tree.binRadice()),300);  
    radice = tree.figlioSinistro(radice);  
    tree.insFiglioSinistro(radice);  
    tree.insFiglioDestro(radice);  
    tree.scriviNodo(tree.figlioSinistro(radice),4);  
    tree.scriviNodo(tree.figlioDestro(radice),4400);  
    radice = tree.figlioDestro(radice);  
    tree.insFiglioSinistro(radice);  
    tree.insFiglioDestro(radice);  
    tree.scriviNodo(tree.figlioSinistro(radice),125);  
    tree.scriviNodo(tree.figlioDestro(radice),2);  
    radice=tree.binRadice();  
    radice=tree.figlioDestro(radice);  
    tree.insFiglioSinistro(radice);  
    tree.insFiglioDestro(radice);  
    tree.scriviNodo(tree.figlioSinistro(radice),13);  
    tree.scriviNodo(tree.figlioDestro(radice),17);  
    return tree;  
}
```

# Svolgimento: visita()

```
void visita (BinAlbero<int>& alb)
{
  if (!alb.binAlberoVuoto())
  {
    Coda<BinAlbero<int>::Nodo> s;
    s.inCoda(alb.binRadice());
    while (!s.codaVuota()){
      BinAlbero<int>::Nodo nn=s.leggiCoda(); // visito un nodo ...
      s.fuoriCoda();
      BinAlbero<int>::Nodo padre = alb.binPadre(nn);
      if (padre!=NULL)
      {
        if (alb.figlioDestro(padre) == nn)
        {
          int valore=alb.leggiNodo(alb.figlioSinistro(nn));
          valore++;
          alb.scriviNodo(alb.figlioSinistro(nn),valore);
          valore=alb.leggiNodo(alb.figlioDestro(nn));
          valore++;
          alb.scriviNodo(alb.figlioDestro(nn),valore);
        }
        else
        {
          int sx=alb.leggiNodo(alb.figlioSinistro(nn));
          int dx=alb.leggiNodo(alb.figlioDestro(nn));
          alb.scriviNodo(alb.figlioDestro(nn),sx);
          alb.scriviNodo(alb.figlioSinistro(nn),dx);
        }
      }
      if (!alb.sinistroVuoto(nn)) s.inCoda(alb.figlioSinistro(nn)); // ... e accodo i figli
      if (!alb.destroVuoto(nn)) s.inCoda(alb.figlioDestro(nn));
    }
  }
}
```