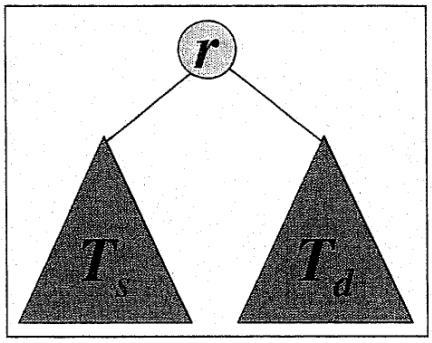
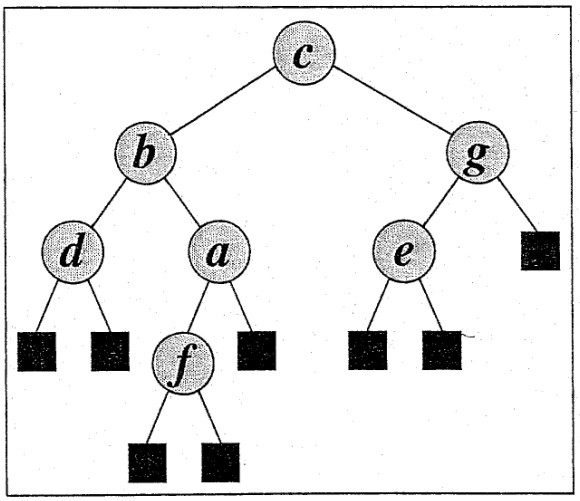
Induzione: Se $T_s, T_d \in \text{Tree}(N)$ sono alberi binari e $r \in N$ allora la terna ordinata (r, Ts, Td) è un albero binario, cioè $(r, T_s, T_d) \in \text{Tree}(N)$.

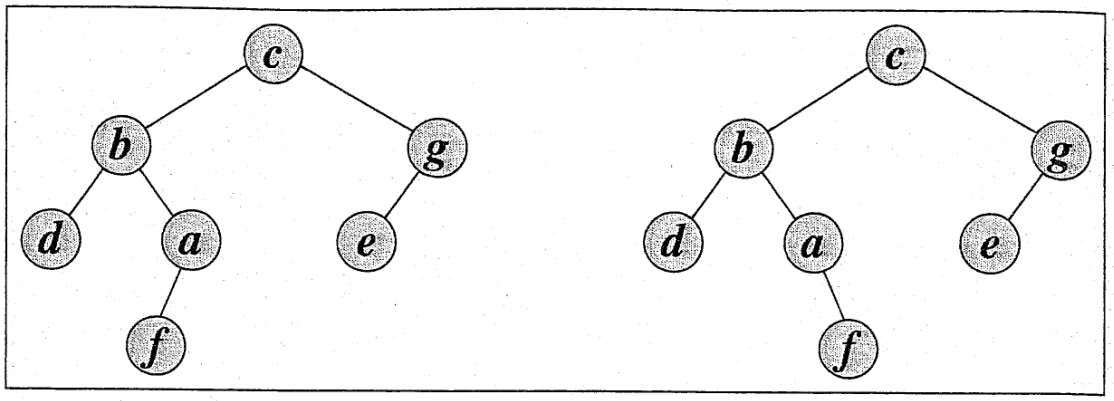
Quando T è l'insieme vuoto diciamo che esso è l'*albero vuoto*. Nell'albero binario $T = (r, T_s, T_d)$ il primo elemento r della terna è la *radice* dell'albero T, il secondo elemento T_s è il *sottoalbero sinistro* di T ed il terzo elemento T_d è il *sottoalbero destro* di T. Rappresentiamo graficamente l'albero vuoto $\varnothing$ con un quadratino nero ■. Invece l'albero non vuoto $T = (r, T_s, T_d)$ si rappresenta graficamente con un nodo etichettato r e sotto di esso, ricorsivamente, le due rappresentazioni dei sottoalberi T_s e T_d, con T_s alla sinistra di T_d.



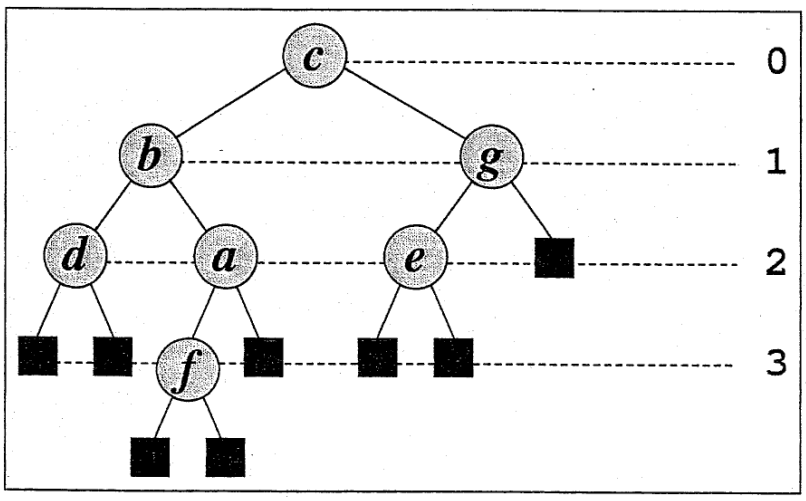
Ad esempio, l'albero $(c, (b, (d, \varnothing, \varnothing), (a, (f, \varnothing, \varnothing), \varnothing)), (g, (e, \varnothing, \varnothing), \varnothing)) \in \text{Tree}(N)$, con $N = \{a, b, c, d, e, f, g\}$, si rappresenta graficamente nel seguente modo:



Spesso la rappresentazione dei sottoalberi vuoti è omessa. In tal caso occorre prestare attenzione a distinguere tra figlio sinistro e destro nel caso in cui un nodo abbia un solo figlio. Ad esempio i due alberi binari seguenti sono diversi:



In un albero (r, T_s, T_d) se il sottoalbero sinistro T_s non è vuoto allora la sua radice r_s si dice *figlio sinistro* di r (e quindi r sarà il *padre* di r_s). Se T_s è vuoto diciamo che r non ha figlio sinistro. Analogamente per T_d. I nodi che non hanno né figlio sinistro né figlio destro si dicono *foglie* dell'albero. I *discendenti* di un nodo sono i figli e tutti i discendenti dei figli. Gli *ascendenti* (o *antenati*) di un nodo sono il padre e tutti gli ascendenti del padre. La *profondità* di un nodo n in un albero T è la lunghezza del cammino dalla radice dell'albero al nodo n. Ad esempio i nodi dell'albero $T = (c, (b, (d, \varnothing, \varnothing), (a, (f, \varnothing, \varnothing), \varnothing)), (g, (e, \varnothing, \varnothing), \varnothing))$ hanno le seguenti profondità:



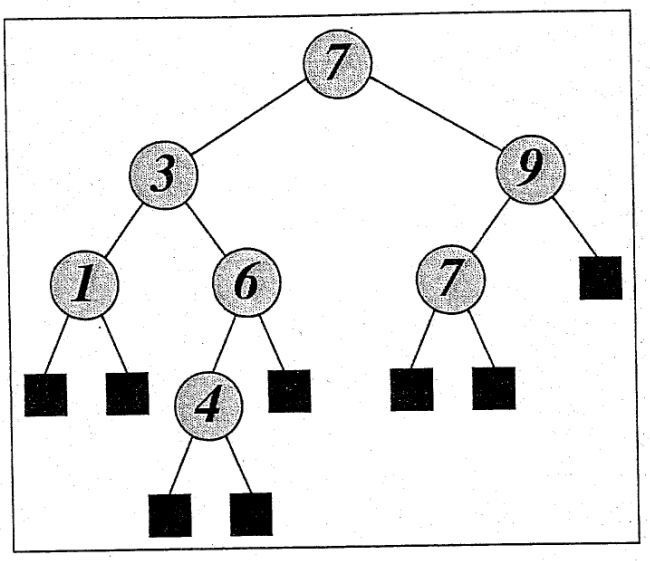
L'*altezza* di un albero binario è la profondità massima delle foglie dell'albero. Ad esempio, il precedente albero ha altezza tre.

Useremo l'abituale rappresentazione ricorsiva per i nodi di un albero binario: una classe `nodo` avente un campo dati `info` in cui viene memorizzato il valore nel nodo e tre campi dati puntatore `padre`, `sinistro` e `destro` che puntano, rispettivamente, al nodo padre, al nodo figlio sinistro ed al nodo figlio destro.

Alberi binari di ricerca. Supponiamo che l'insieme N sia ordinato. Un albero binario di ricerca $T \in \text{Tree}(N)$ è un albero binario in cui il valore di ogni nodo di T è maggiore

dei valori dei nodi del suo sottoalbero sinistro e minore o uguale dei valori dei nodi del suo sottoalbero destro.

Ad esempio, l'albero binario della seguente figura in cui i valori dei nodi sono dei numeri interi è un albero binario di ricerca.



Per rappresentare gli alberi binari di ricerca useremo un template di classe `AlbBinRic<T>` parametrico rispetto al tipo `T` dei valori dei nodi. I nodi saranno rappresentati con il template di classe `Nodo<T>`. Per ogni istanza del template di classe `AlbBinRic` ci sarà la possibilità di creare, modificare e accedere ai campi dei nodi appartenenti all'istanza associata del template di classe `Nodo`. D'altra parte l'operazione di ricerca in un albero binario di ricerca dovrà ritornare un puntatore al nodo trovato. Deve quindi essere possibile usare puntatori ai nodi anche all'esterno della classe. Una soluzione consiste nel dichiarare privati tutti i campi dati e metodi del template di classe `Nodo`, compresi i costruttori, e dichiarare quindi friend le istanze associate del template di classe `AlbBinRic`. Definiremo quindi i template di classe `Nodo<T>` e `AlbBinRic<T>` come segue.

```
// dichiarazione incompleta di AlbBinRic
template <class T> class Nodo;

// dichiarazione di operator<<
template <class T>
ostream& operator<< (ostream&, Nodo<T>*);

template <class T>
class Nodo {
friend class AlbBinRic<T>; // classe friend associata
// funzione friend associata
friend ostream& operator<< <T>(ostream&, Nodo<T>*);
private:
  T info;
```

```
  // notare il costruttore privato
  Nodo *sinistro, *destro, *padre;
  Nodo(const T& v, Nodo* p=0, Nodo* s=0, Nodo* d=0) :
       info(v), padre(p), sinistro(s), destro(d) {}
};

template <class T>
ostream& operator<< (ostream& os, Nodo<T>* p) {
  if (!p) os << '@'; // caso base
  else os << "(" << p->info << "," << p->sinistro
          << "," << p->destro << ")"; // passo ricorsivo
  return os;
}
```

```
// dichiarazione di operator<<
template <class T>
ostream& operator<< (ostream&, AlbBinRic<T>&);

template <class T>
class AlbBinRic {
friend ostream& operator<< <T>(ostream&,const AlbBinRic<T>&);
public:
  AlbBinRic() : radice(0) {}
  Nodo<T>* Find(T) const;
  Nodo<T>* Minimo() const;
  Nodo<T>* Massimo() const;
  Nodo<T>* Succ(Nodo<T>*) const;
  Nodo<T>* Pred(Nodo<T>*) const;
  void Insert(T);
  static T Valore(Nodo<T>* p) { return p->info; }
private:
  Nodo<T>* radice; // puntatore alla radice
  // metodi privati di utilità
  static Nodo<T>* FindRic(Nodo<T>*, T);
  static Nodo<T>* MinimoRic(Nodo<T>*);
  static Nodo<T>* MassimoRic(Nodo<T>*);
  static void InsertRic(Nodo<T>*, T);
};
```

Passiamo ad esaminare le definizioni dei metodi del template di classe `AlbBinRic<T>`. Gli algoritmi sugli alberi binari di ricerca sono approfonditamente studiati in un corso di Algoritmi e Strutture Dati.

L'algoritmo di ricerca sfrutta la definizione di albero binario di ricerca.

```
template <class T>
```

```
Nodo<T>* AlbBinRic<T>::Find(T v) const {
  return FindRic(radice,v);
}

template <class T>
Nodo<T>* AlbBinRic<T>::FindRic(Nodo<T>* x, T v) {
  if (!x) return x; // caso base: albero vuoto
  if(x->info == v) return x; // caso base: v è nella radice
  if (v < x->info)  // cerca ricorsivamente nel sottoalbero sx
    return FindRic(x->sinistro,v);
  else              // cerca ricorsivamente nel sottoalbero dx
    return FindRic(x->destro,v);
}
```

L'idea dell'algoritmo che calcola il valore massimo di un albero binario di ricerca è la seguente: il valore massimo è memorizzato nella "foglia più a destra".

```
template <class T>
Nodo<T>* AlbBinRic<T>::Massimo() const {
  if (!radice) return 0;
  return MassimoRic(radice);
}

template <class T>
Nodo<T>* AlbBinRic<T>::MassimoRic(Nodo<T>* x) {
  if (!(x->destro)) return x;
  else return MassimoRic(x->destro);
}
```

Naturalmente, il minimo è duale al massimo.

```
template <class T>
Nodo<T>* AlbBinRic<T>::Minimo() const {
  if (!radice) return 0;
  return MinimoRic(radice);
}

template <class T>
Nodo<T>* AlbBinRic<T>::MinimoRic(Nodo<T>* x) {
  if (!(x->sinistro)) return x;
  else return MinimoRic(x->sinistro);
}
```

L'idea dell'algoritmo che calcola il successore del valore di un nodo puntato da x in un albero T è la seguente: se (il nodo puntato da) x ha sottoalbero destro x_d allora il

successore di x è il minimo di x_d; se il sottoalbero destro di x è vuoto allora il successore di x è l'antenato di x più giovane di cui x è discendente sinistro

```
template <class T>
Nodo<T>* AlbBinRic<T>::Succ(Nodo<T>* x) const {
  if (!x) return 0;
  if (x->destro) return MinimoRic(x->destro);
  // caso x->destro == 0
  while (x->padre && x->padre->destro == x) x = x->padre;
  return x->padre;
}
```

La funzione `Pred` è duale alla precedente `Succ`.

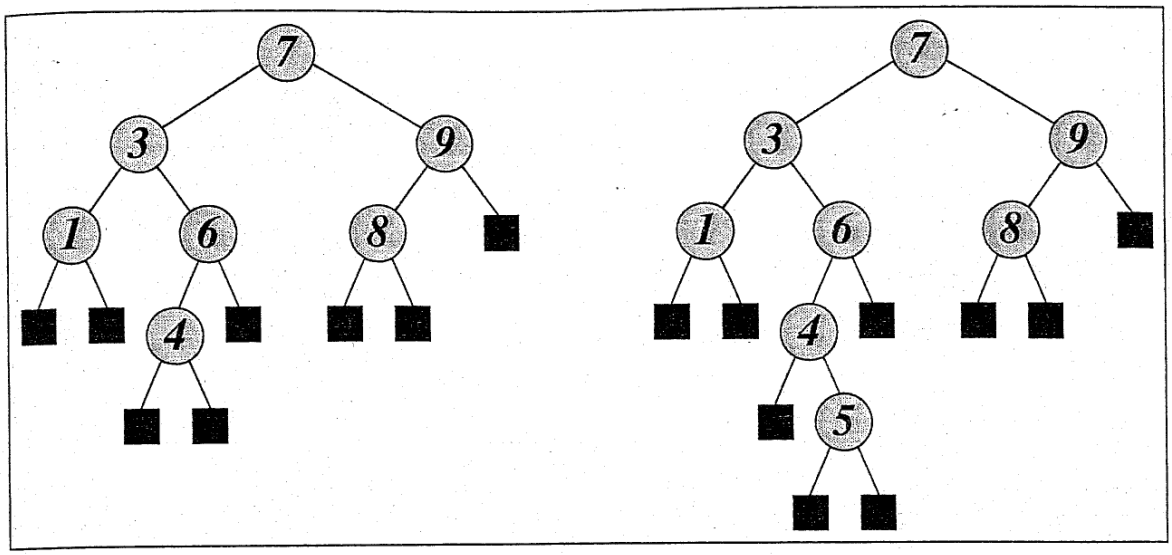
```
template <class T>
Nodo<T>* AlbBinRic<T>::Pred(Nodo<T>* x) const {
  if (!x) return 0;
  if (x->sinistro) return MassimoRic(x->sinistro);
  while (x->padre && x->padre->sinistro == x) x = x->padre;
  return x->padre;
}
```

L'idea dell'algoritmo per inserire un valore v nell'albero T è la seguente: se T è vuoto inserisco v come radice, altrimenti se $T = (r, T_s, T_d)$ e $v < r$ allora inserisco v in T_s, mentre se $v \geq r$ inserisco v in T_d.

```
template <class T>
void AlbBinRic<T>::Insert(T v) {
  if (!radice) radice = new Nodo<T>(v);
  else InsertRic(radice,v);
}

template <class T>
void AlbBinRic<T>::InsertRic(Nodo<T>* x, T v) {
  if (v < x->info)
    if (x->sinistro == 0) x->sinistro = new Nodo<T>(v,x);
    else InsertRic(x->sinistro,v);
  else
    if (x->destro == 0) x->destro = new Nodo<T>(v,x);
    else InsertRic(x->destro,v);
}
```

Ad esempio, se `t` è un puntatore alla radice dell'albero di destra della seguente figura allora `t.insert(5)` punterà alla radice dell'albero di sinistra.



Infine, l'operatore di output semplicemente richiama l'operatore di output per puntatori a `Nodo` sulla radice dell'albero.

```
template <class T>
ostream& operator<<(ostream& os,const AlbBinRic<T>& A) {
  os << A.radice << endl; // stampa di Nodo<T>*
  return os;
}
```

Il seguente esempio di `main()` istanzia il template `AlbBinRic<int>` e usa le varie funzionalità disponibili.

```
// file "main-abr.cpp"
#include<iostream>
#include "nodo.h"
#include "AlbBinRic.h"
using namespace std;

int main() {
 AlbBinRic<int> a;
 a.Insert(3); a.Insert(2); a.Insert(3);
 a.Insert(1); a.Insert(6); a.Insert(5);
 cout << a;

 cout << "minimo: " << AlbBinRic<int>::Valore(a.Minimo()) << endl;
       // minimo: 1
 cout << "massimo: "<< AlbBinRic<int>::Valore(a.Massimo()) <<endl;
       // massimo: 6

 cout << "successore minimo: "
```

```
      << AlbBinRic<int>::Valore(a.Succ(a.Minimo())) << endl;
       // successore minimo: 2
 cout << "predecessore masssimo: "
      << AlbBinRic<int>::Valore(a.Pred(a.Massimo())) << endl;
       // predecessore massimo: 5

 if(a.Find(2)) cout << "Il valore " << 2 << " è presente\n";
       // Il valore 2 è presente
}
```

Esercizio 4.4.1. Definire il metodo di cancellazione:

```
template<class T> void Delete(T);
```

che rimuove un nodo contenente un dato valore, se un tale nodo esiste. Si tratta della funzione più complicata, studiata dettagliatamente in un corso di Algoritmi e Strutture Dati.