Capitolo 12

Strutture dati dinamiche

Sommario: Strutture dati dinamiche

Strutture dati

- Le strutture dati sono le strutture utilizzate per memorizzare e organizzare i dati

Strutture dati

- Le strutture dati sono le strutture utilizzate per memorizzare e organizzare i dati
- Le strutture dati dinamiche sono quelle la cui organizzazione evolve dinamicamente durante l'esecuzione del programma:

Strutture dati

- Le strutture dati sono le strutture utilizzate per memorizzare e organizzare i dati
- Le strutture dati dinamiche sono quelle la cui organizzazione evolve dinamicamente durante l'esecuzione del programma:

  pile, code, insiemi, liste, alberi, ...

Strutture dati

- Le strutture dati sono le strutture utilizzate per memorizzare e organizzare i dati
- Le strutture dati dinamiche sono quelle la cui organizzazione evolve dinamicamente durante l'esecuzione del programma:

  pile, code, insiemi, liste, alberi, ...

- La scelta della struttura dati dipende principalmente dalle operazioni che dovranno essere effettuate

Pila (stack)

- Struttura LIFO (Last In First Out)
  È possibile aggiungere o eliminare elementi solo in cima alla pila

Pila (stack)

- Struttura LIFO (Last In First Out)
  È possibile aggiungere o eliminare elementi solo in cima alla pila
- Operazioni
  - `push`: per aggiungere un elemento in cima alla pila

Pila (stack)

- Struttura LIFO (Last In First Out)
  È possibile aggiungere o eliminare elementi solo in cima alla pila
- Operazioni
  - `push`: per aggiungere un elemento in cima alla pila
  - `pop`: per prelevare ed eliminare l'elemento in cima alla pila

Pila (stack)

- Struttura LIFO (Last In First Out)
  È possibile aggiungere o eliminare elementi solo in cima alla pila
- Operazioni
  - `push`: per aggiungere un elemento in cima alla pila
  - `pop`: per prelevare ed eliminare l'elemento in cima alla pila
  - `empty`: per verificare se la pila è vuota

Implementazione di `Stack<E>`

Costruttore

- `public Stack()`
  Crea un oggetto che rappresenta una pila vuota.

Implementazione di `Stack<E>`

Costruttore

- `public Stack()`
  Crea un oggetto che rappresenta una pila vuota.

Metodi

- `public void push(E o)`
  Aggiunge in cima alla pila che esegue il metodo l'oggetto fornito tramite il parametro.

Implementazione di Stack<E>

Costruttore

- `public Stack()`
  Crea un oggetto che rappresenta una pila vuota.

Metodi

- `public void push(E o)`
  Aggiunge in cima alla pila che esegue il metodo l'oggetto fornito tramite il parametro.

- `public E pop()`
  Restituisce un riferimento all'oggetto che si trova in cima alla pila che esegue il metodo eliminandolo dalla pila stessa. Se la pila è vuota, il metodo solleva l'eccezione non controllata `EmptyStackException`.

Implementazione di Stack<E>

Costruttore

- `public Stack()`
  Crea un oggetto che rappresenta una pila vuota.

Metodi

- `public void push(E o)`
  Aggiunge in cima alla pila che esegue il metodo l'oggetto fornito tramite il parametro.

- `public E pop()`
  Restituisce un riferimento all'oggetto che si trova in cima alla pila che esegue il metodo eliminandolo dalla pila stessa. Se la pila è vuota, il metodo solleva l'eccezione non controllata `EmptyStackException`.

- `public boolean empty()`
  Restituisce `true` se e solo se la pila che esegue il metodo è vuota.

EmptyStackException

Definiamo la classe per l'eccezione non controllata che può essere sollevata dal metodo `pop`.

EmptyStackException

Definiamo la classe per l'eccezione non controllata che può essere sollevata dal metodo pop.

```
public class EmptyStackException extends RuntimeException {
}
```

Implementazione di pile mediante array

```
public class Pila {
  //CAMPI
  private static final int SIZE = 10;
  private Object[] dati;
  private int top;  //indica la prima posizione libera

  //COSTRUTTORI
  public Pila() {
    dati = new Object[SIZE];
    top = 0;
  }

  //METODI
  ...
}
```

I metodi push e pop

```
public class Pila {
  //CAMPI
  private static final int SIZE = 10;
  private Object[] dati;
  private int top;  //indica la prima posizione libera
  ...
  //METODI
  public void push(Object o) {
    dati[top++] = o;
  }

  public Object pop() {
    if (top == 0)
      throw new EmptyStackException();
    else
      return dati[--top];
  }
  ...
```

Il metodo empty

```
public class Pila {
  //CAMPI
  private static final int SIZE = 10;
  private Object[] dati;
  private int top;  //indica la prima posizione libera

  ...

  public boolean empty() {
    return top == 0;
  }
}
```

Implementazione mediante strutture dinamiche

Anziché creare subito lo spazio per la pila (come nel caso dell'array), lo creiamo dinamicamente, man mano che risulta necessario.

Implementazione mediante strutture dinamiche

Anziché creare subito lo spazio per la pila (come nel caso dell'array), lo creiamo dinamicamente, man mano che risulta necessario.

- Creiamo lo spazio per il nuovo elemento da aggiungere alla pila ogni volta che effettuiamo l'operazione `push`

Implementazione mediante strutture dinamiche

Anziché creare subito lo spazio per la pila (come nel caso dell'array), lo creiamo dinamicamente, man mano che risulta necessario.

- Creiamo lo spazio per il nuovo elemento da aggiungere alla pila ogni volta che effettuiamo l'operazione `push`
- Quando eseguiamo un'operazione `pop`, rilasciamo lo spazio occupato dall'elemento prelevato dalla pila

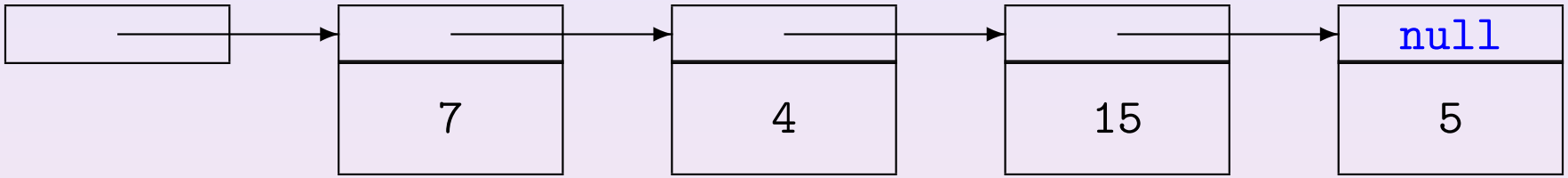
Implementazione mediante strutture dinamiche

Anziché creare subito lo spazio per la pila (come nel caso dell'array), lo creiamo dinamicamente, man mano che risulta necessario.

- Creiamo lo spazio per il nuovo elemento da aggiungere alla pila ogni volta che effettuiamo l'operazione `push`
- Quando eseguiamo un'operazione `pop`, rilasciamo lo spazio occupato dall'elemento prelevato dalla pila

Utilizziamo una lista concatenata: un insieme di nodi, collegati tra loro mediante riferimenti.

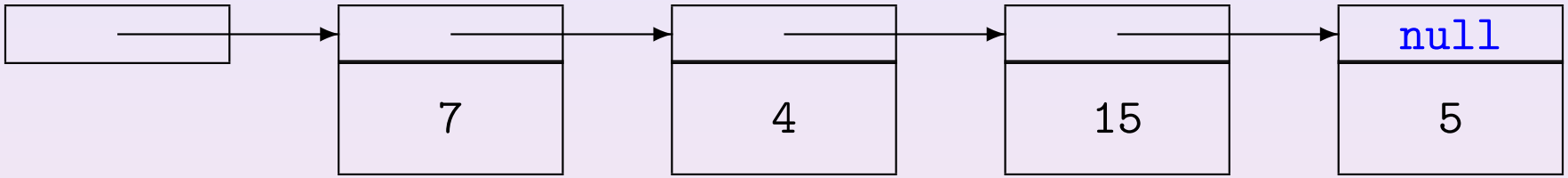
Esempio: una lista di interi



Ogni nodo contiene:

- un'informazione

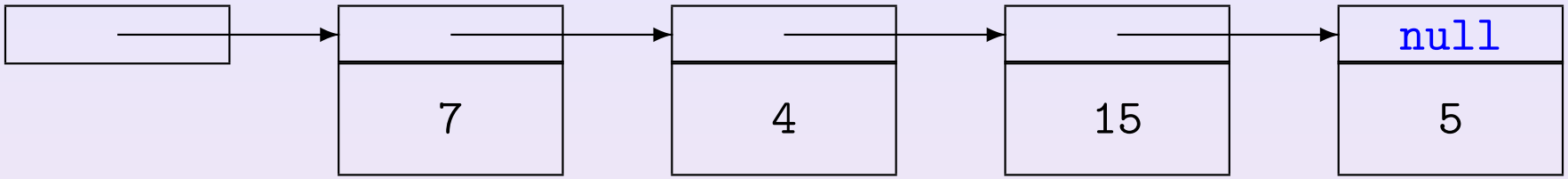
Esempio: una lista di interi



Ogni nodo contiene:

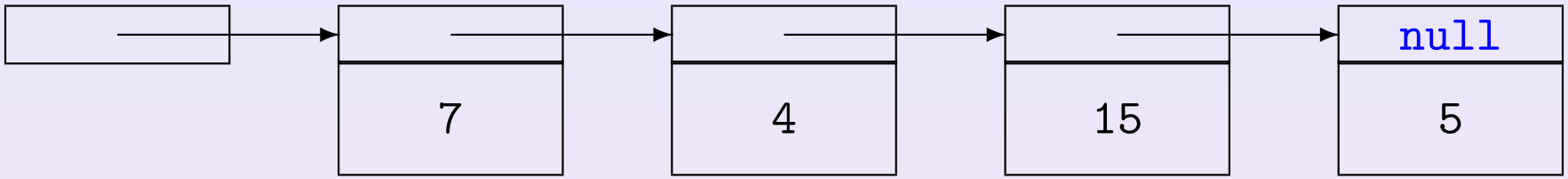
- un'informazione
- un riferimento (puntatore) al nodo successivo

Esempio: rappresentazione della pila



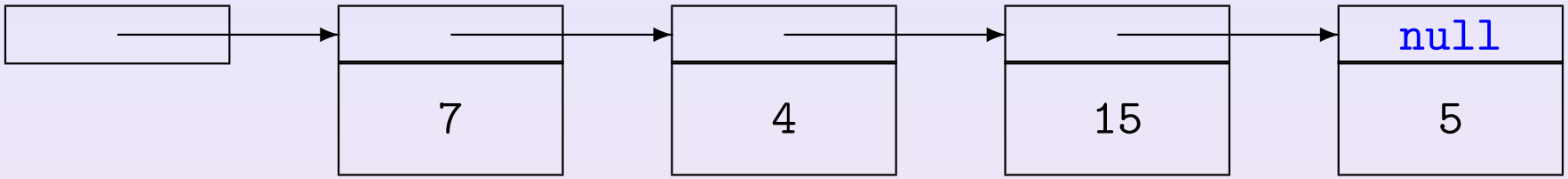
- Il primo elemento della lista corrisponde alla cima della pila

Esempio: rappresentazione della pila



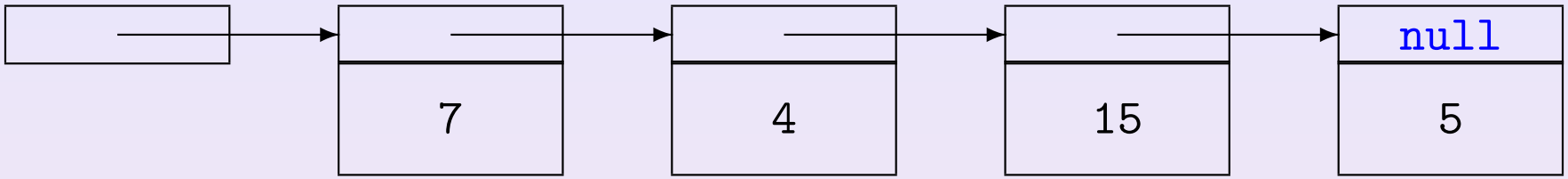
- Il primo elemento della lista corrisponde alla cima della pila
- Ogni elemento è seguito dall'elemento che nella pila si trova sotto di esso

Esempio: rappresentazione della pila



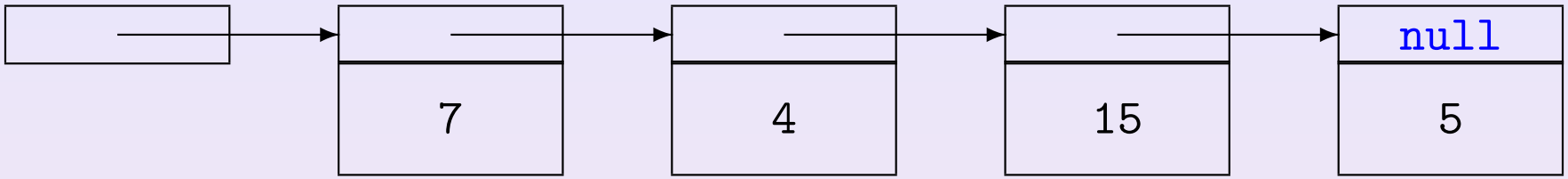
- Il primo elemento della lista corrisponde alla cima della pila
- Ogni elemento è seguito dall'elemento che nella pila si trova sotto di esso
- L'ultimo elemento della lista non ha successori e corrisponde all'elemento che si trova più in basso nella pila

Esempio: rappresentazione della pila



- Il primo elemento della lista corrisponde alla cima della pila
- Ogni elemento è seguito dall'elemento che nella pila si trova sotto di esso
- L'ultimo elemento della lista non ha successori e corrisponde all'elemento che si trova più in basso nella pila
- Quindi:
  - `push` dovrà aggiungere gli elementi in testa alla lista

Esempio: rappresentazione della pila



- Il primo elemento della lista corrisponde alla cima della pila
- Ogni elemento è seguito dall'elemento che nella pila si trova sotto di esso
- L'ultimo elemento della lista non ha successori e corrisponde all'elemento che si trova più in basso nella pila
- Quindi:
  - `push` dovrà aggiungere gli elementi in testa alla lista
  - `pop` dovrà eliminare gli elementi dalla testa della lista

Implementazione del nodo

```
class NodoStack {
  E dato;
  NodoStack pros;
}
```

Implementazione del nodo

```
class NodoStack {
  E dato;
  NodoStack pros;
}
```

- Il campo `dato` conterrà il riferimento all'oggetto che contiene le informazioni

Implementazione del nodo

```
class NodoStack {
  E dato;
  NodoStack pros;
}
```

- Il campo `dato` conterrà il riferimento all'oggetto che contiene le informazioni
- Il campo `pros` permette di accedere all'elemento successivo della lista

Implementazione del nodo

```
class NodoStack {
  E dato;
  NodoStack pros;
}
```

- Il campo `dato` conterrà il riferimento all'oggetto che contiene le informazioni
- Il campo `pros` permette di accedere all'elemento successivo della lista

Osservazione

La definizione di `NodoStack` è ricorsiva.

Implementazione della pila

```
public class Stack<E> {
  private NodoStack cima;
  ...
}
```

Implementazione della pila

```
public class Stack<E> {
  private NodoStack cima;
  ...
}
```

- `cima` è il riferimento alla cima della pila

Implementazione della pila

```
public class Stack<E> {
  private NodoStack cima;
  ...
}
```

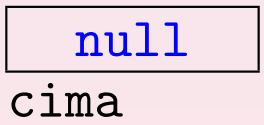
- `cima` è il riferimento alla cima della pila
- La pila vuota è rappresentata da un'istanza in cui `cima` contiene `null`

Il costruttore

```
public class Stack<E> {
  private NodoStack cima;

  //COSTRUTTORE
  public Stack() {
    cima = null;
  }
  ...
}
```

Il costruttore

```
public class Stack<E> {
  private NodoStack cima;

  //COSTRUTTORE
  public Stack() {
    cima = null;
  }
  ...
}
```



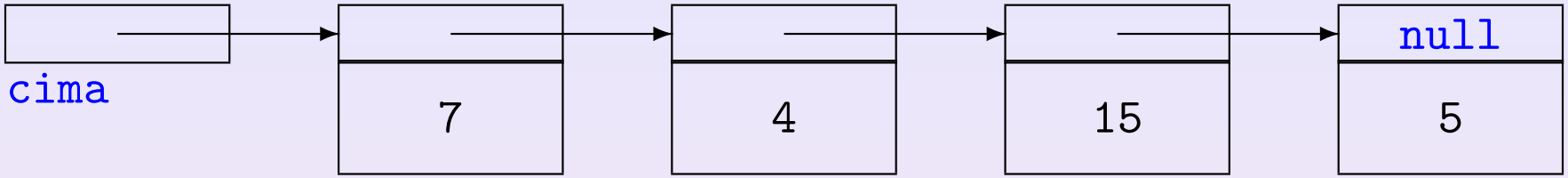
Il metodo empty

```
public class Stack<E> {
  private NodoStack cima;

  ...

  public boolean empty() {
    return cima == null;
  }

  ...
}
```

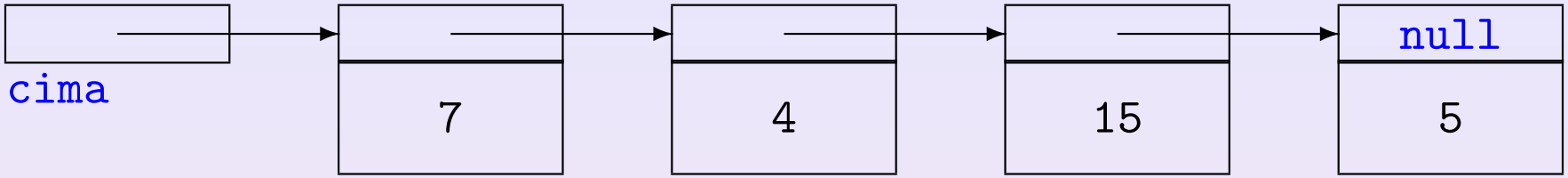
Implementazione di push

```
public class Stack<E> {
  private NodoStack cima;

  ...

  public void push(E o) {
    NodoStack t = new NodoStack();
    t.dato = o;
    t.pros = cima;
    cima = t;
  }

  ...
}
```

Esempio

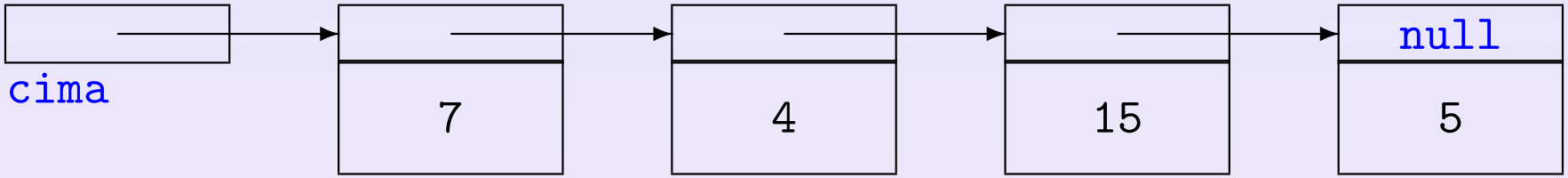


Esempio

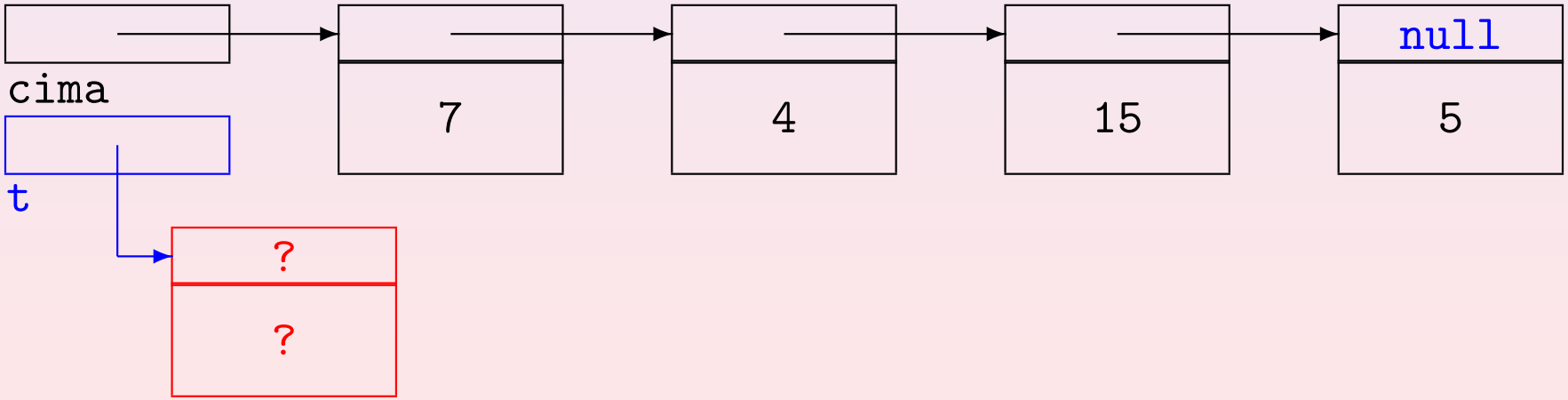



```
NodoStack t = new NodoStack();
t.dato = o;
t.pros = cima;
cima = t;
```

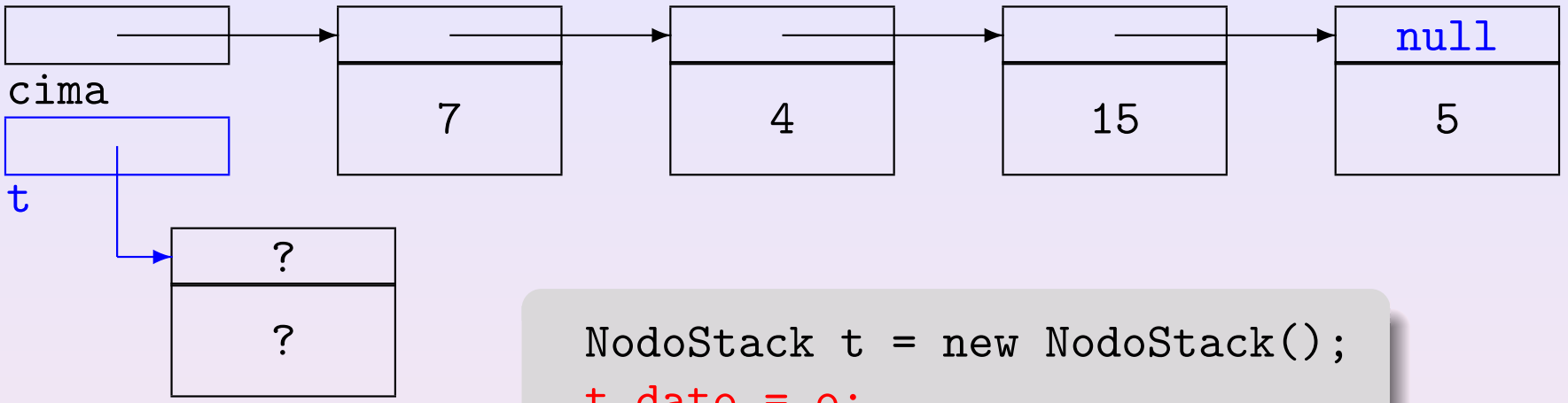
Esempio




```
NodoStack t = new NodoStack();
t.dato = o;
t.pros = cima;
cima = t;
```

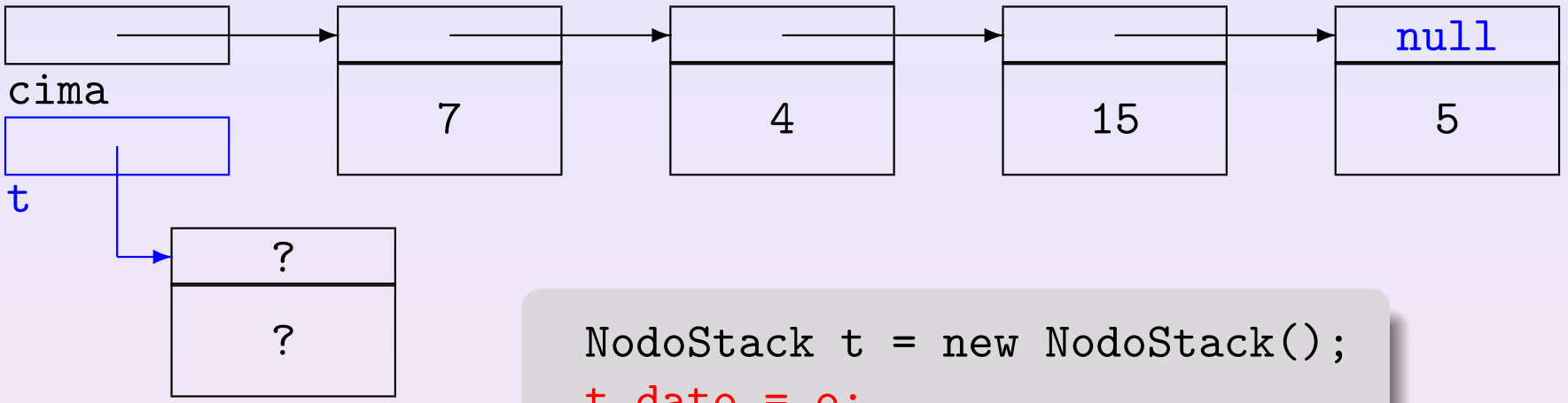


Esempio

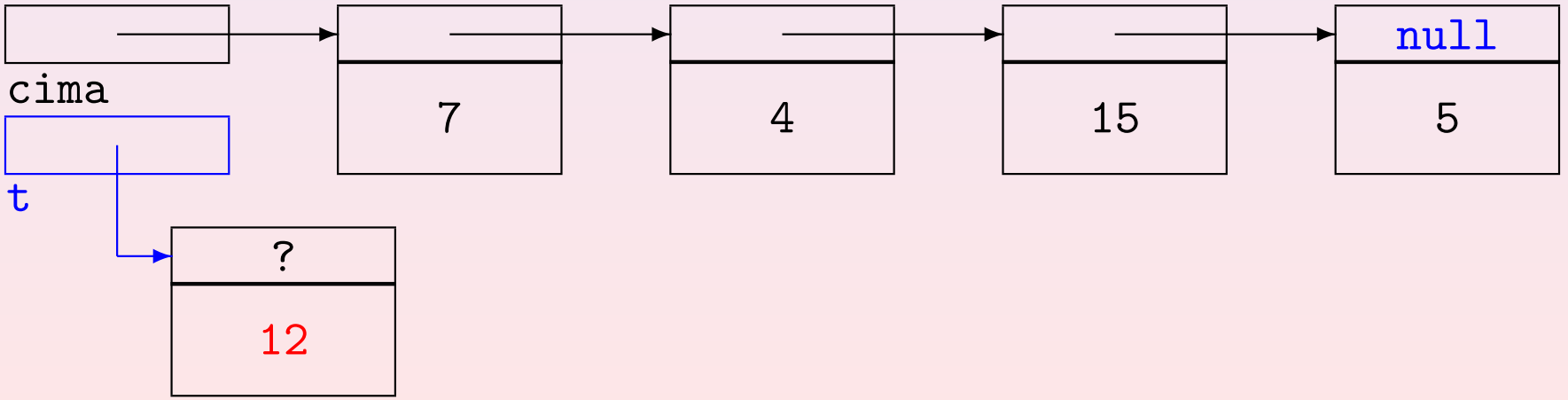



```
NodoStack t = new NodoStack();
t.dato = o;
t.pros = cima;
cima = t;
```

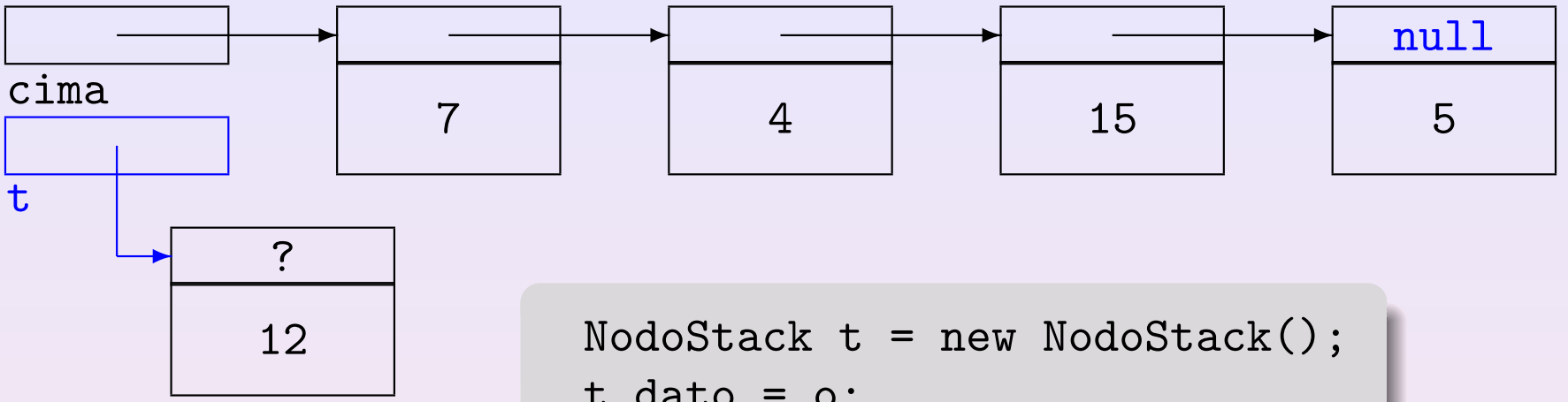
Esempio




```
NodoStack t = new NodoStack();
t.dato = o;
t.pros = cima;
cima = t;
```

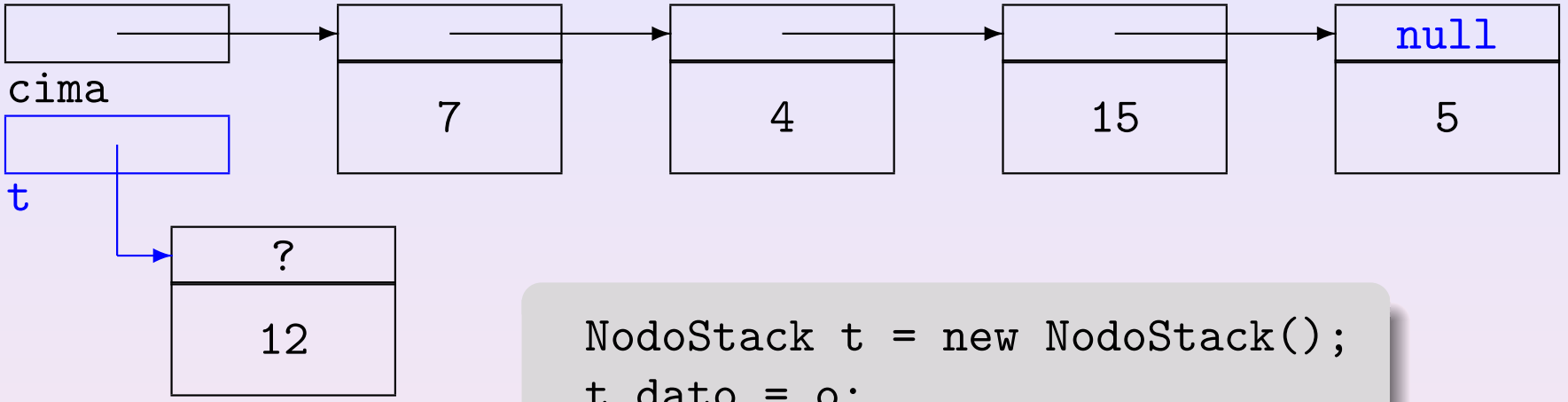


Esempio

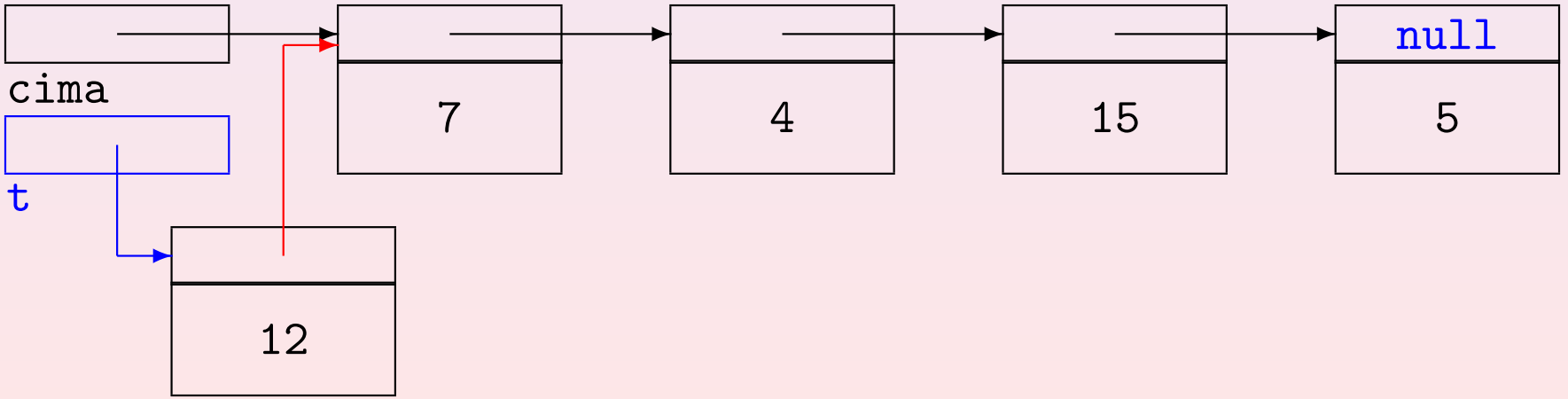



```
NodoStack t = new NodoStack();
t.dato = o;
t.pros = cima;
cima = t;
```

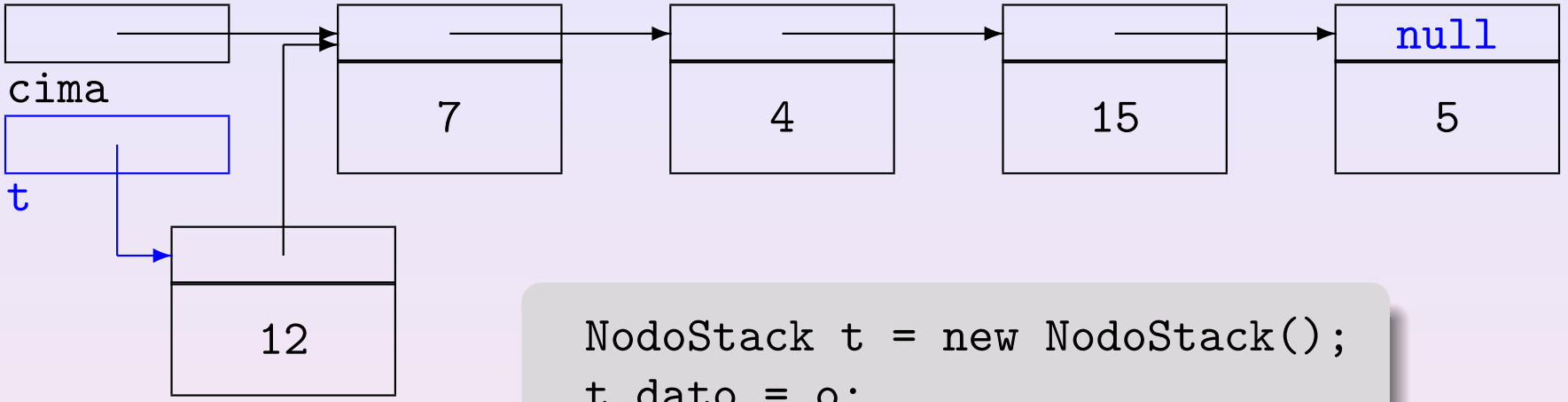
Esempio




```
NodoStack t = new NodoStack();
t.dato = o;
t.pros = cima;
cima = t;
```

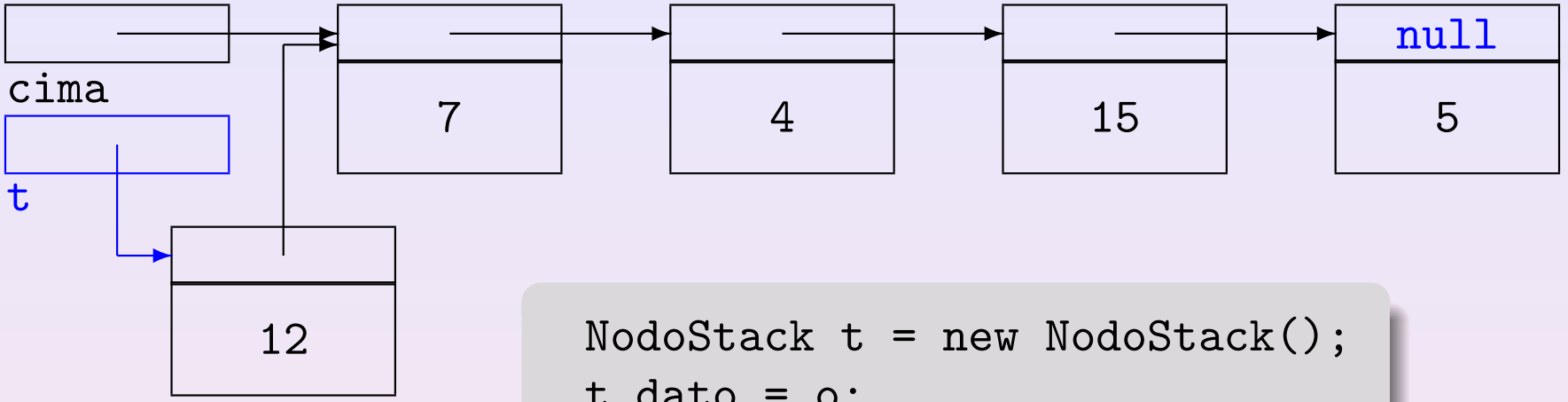


Esempio

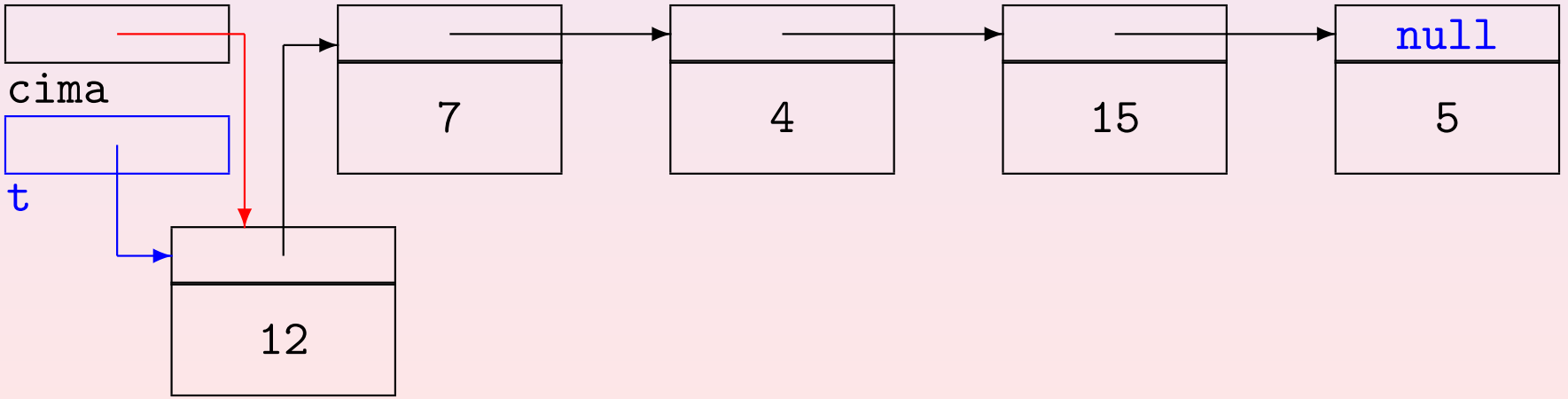



```
NodoStack t = new NodoStack();
t.dato = o;
t.pros = cima;
cima = t;
```

Esempio



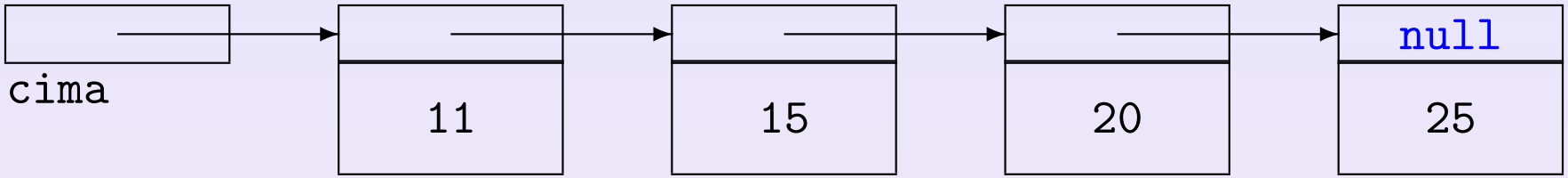

```
NodoStack t = new NodoStack();
t.dato = o;
t.pros = cima;
cima = t;
```



Implementazione di pop

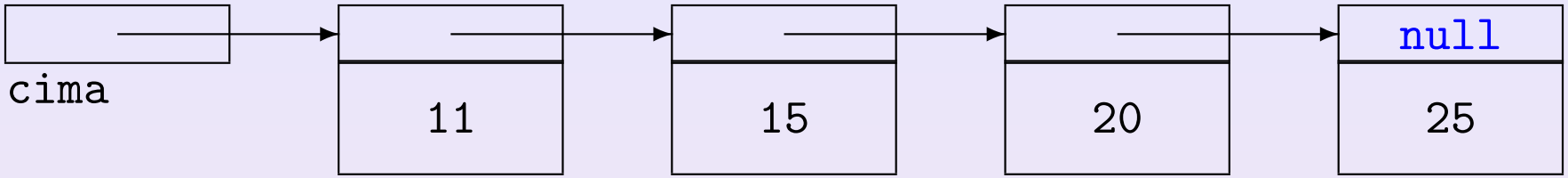
```
public class Stack<E> {
  private NodoStack cima;

  ...

  public E pop() {
    if (cima == null)
      throw new EmptyStackException();
    else {
      E risultato = cima.dato;
      cima = cima.pros;
      return risultato;
    }
  }
  ...
}
```

Esempio

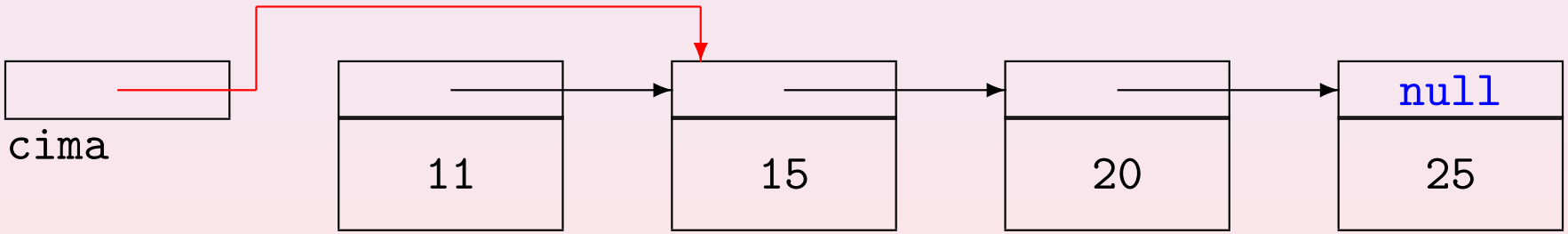



```
...
cima = cima.pros;
...
```

Esempio




```
...
cima = cima.pros;
...
```



Osservazione: l'elemento non è stato fisicamente cancellato, ci penserà il garbage collector

NodoStack come classe interna

```
public class Stack<E> {
  private NodoStack cima;

  private class NodoStack {
    E dato;
    NodoStack pros;
  }

  ...
  //COSTRUTTORE
  ...
  //METODI
  ...
}
```

Sommario: Strutture dati dinamiche

Coda

- Struttura FIFO (First In First Out)
  Il primo elemento che può essere prelevato è quello inserito per primo

Coda

- Struttura FIFO (First In First Out)
  Il primo elemento che può essere prelevato è quello inserito per primo
- Una coda può essere realizzata con una lista in cui:

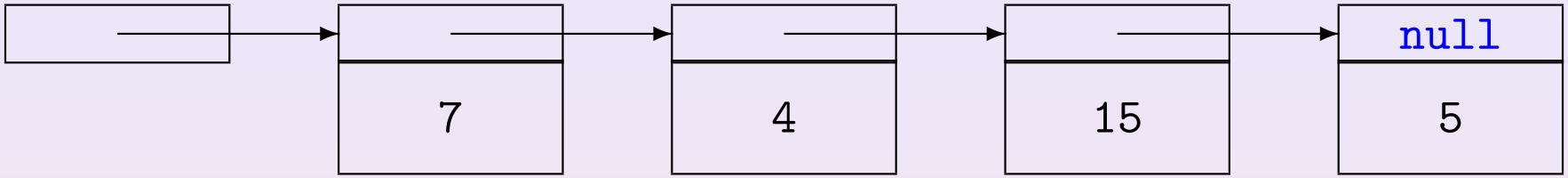
Coda

- Struttura FIFO (First In First Out)
  Il primo elemento che può essere prelevato è quello inserito per primo
- Una coda può essere realizzata con una lista in cui:
  - gli inserimenti vengono effettuati alla fine della lista

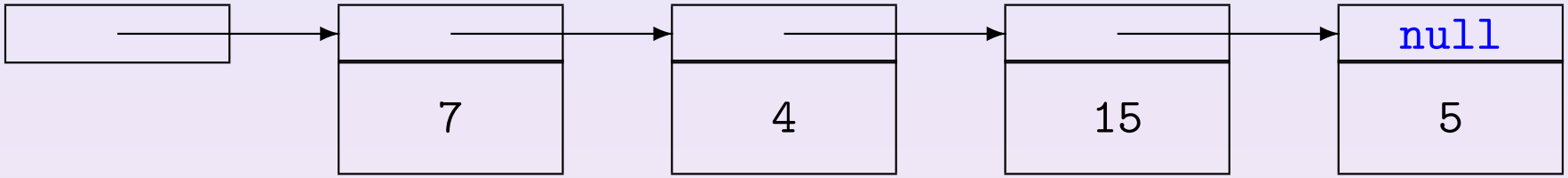
Coda

- Struttura FIFO (First In First Out)
  Il primo elemento che può essere prelevato è quello inserito per primo
- Una coda può essere realizzata con una lista in cui:
  - gli inserimenti vengono effettuati alla fine della lista
  - gli elementi vengono prelevati dalla testa della lista

Implementazione mediante liste



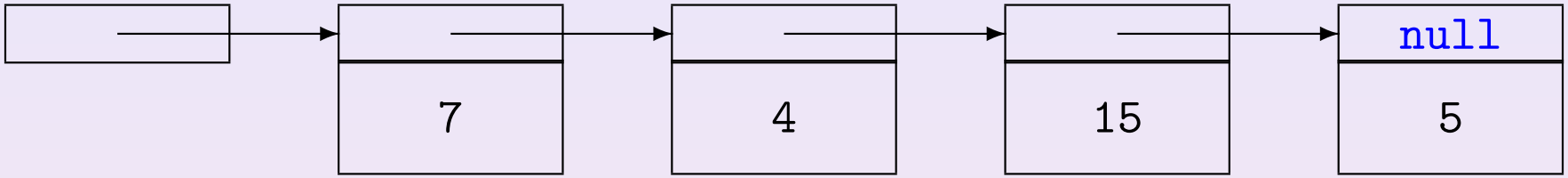
Implementazione mediante liste



Per inserire un nuovo nodo occorre:

- creare un nuovo nodo

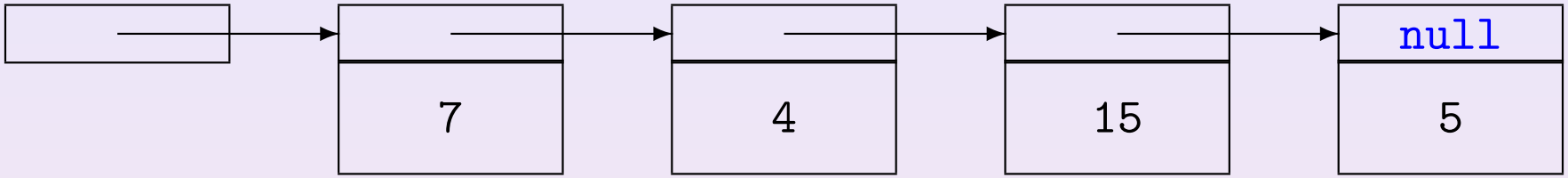
Implementazione mediante liste



Per inserire un nuovo nodo occorre:

- creare un nuovo nodo
- percorrere l'intera lista fino a raggiungere l'ultimo nodo

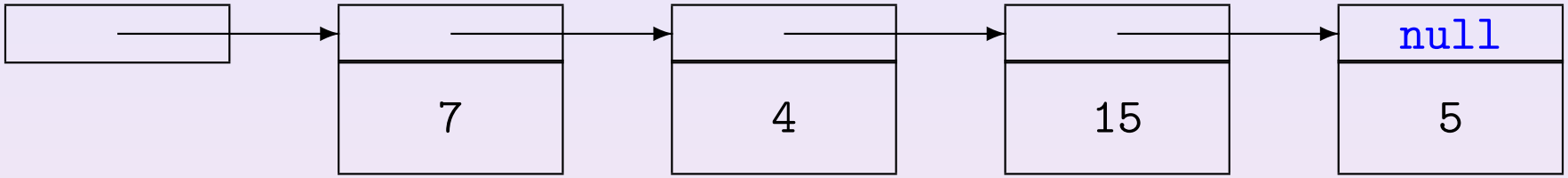
Implementazione mediante liste



Per inserire un nuovo nodo occorre:

- creare un nuovo nodo
- percorrere l'intera lista fino a raggiungere l'ultimo nodo
- collegare il nuovo nodo alla lista facendo puntare a esso il riferimento contenuto nell'ultimo nodo

Implementazione mediante liste



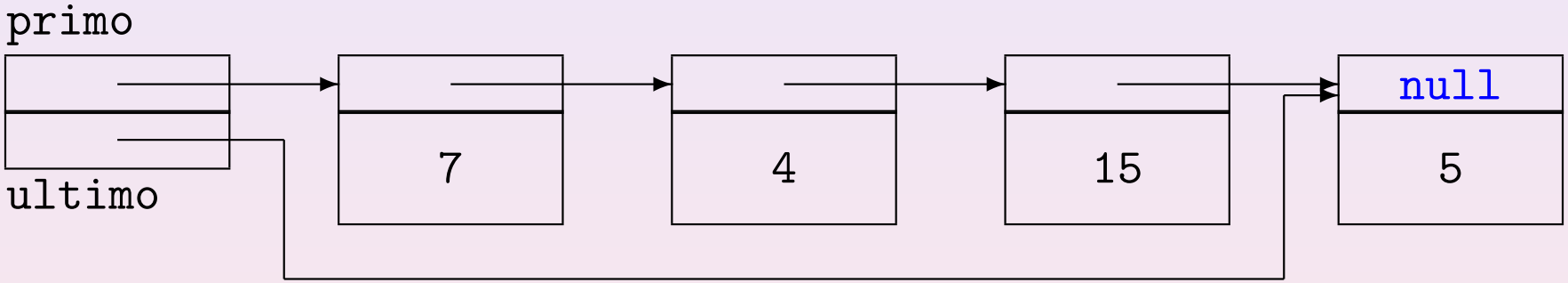
Per inserire un nuovo nodo occorre:

- creare un nuovo nodo
- percorrere l'intera lista fino a raggiungere l'ultimo nodo
- collegare il nuovo nodo alla lista facendo puntare a esso il riferimento contenuto nell'ultimo nodo

Osservazione: In termini di tempo questa tecnica è molto dispendiosa.

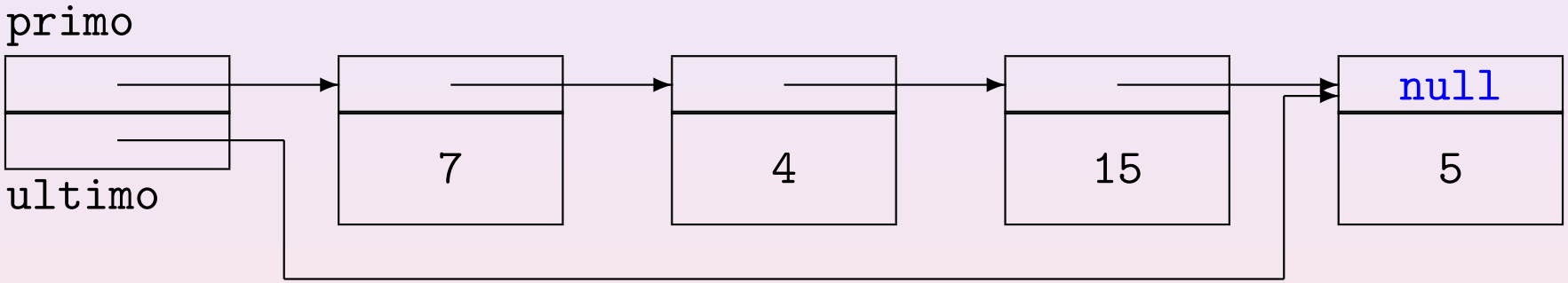
Implementazione mediante liste

Per facilitare l'inserimento rappresentiamo la struttura con due riferimenti, uno al primo, l'altro all'ultimo elemento della lista



Implementazione mediante liste

Per facilitare l'inserimento rappresentiamo la struttura con due riferimenti, uno al primo, l'altro all'ultimo elemento della lista



La coda vuota viene rappresentata ponendo a `null` ambedue i riferimenti `primo` e `ultimo`.

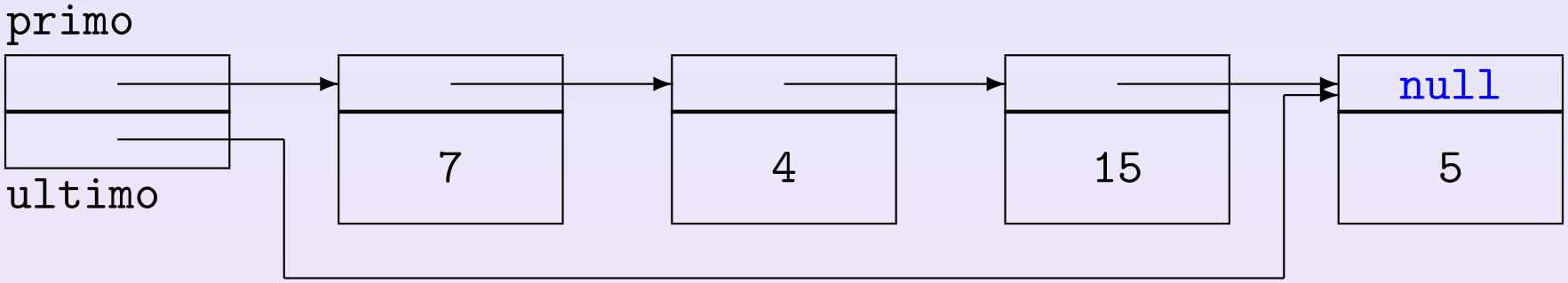
Implementazione

```
public class Coda<E> {
  private NodoCoda primo, ultimo;

  private static class NodoCoda {
    E dato;
    NodoCoda pros;
  }

  //COSTRUTTORE
  public Coda() {
    primo = ultimo = null;
  }

  //METODI
  ...
}
```

Il metodo aggiungi

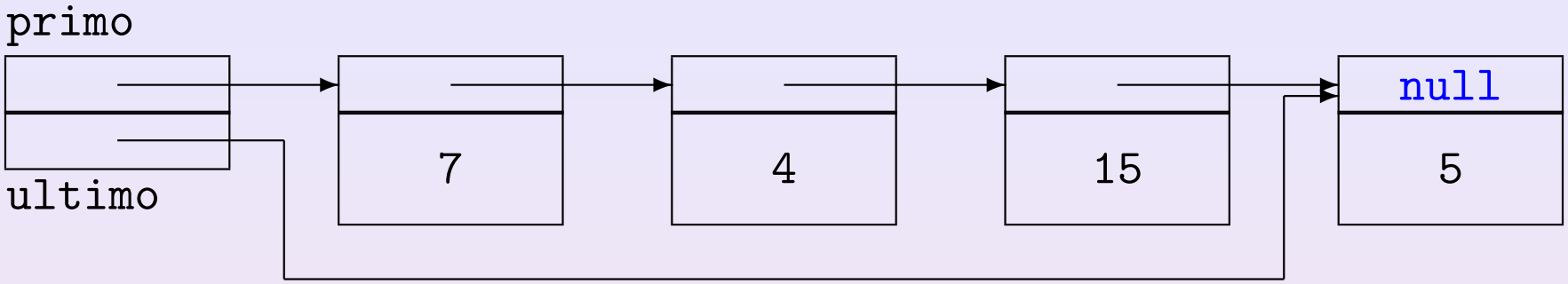
```
public class Coda<E> {
  private NodoCoda primo, ultimo;

  ...
   public void aggiungi(E x) {
    //creazione del nuovo nodo
    NodoCoda t = new NodoCoda();
    t.dato = x;
    t.pros = null;
    //inserimento del nodo
    if (primo == null)      //caso di coda inizialmente vuota
      primo = ultimo = t;
    else {                  //caso di coda non vuota
      ultimo.pros = t;      //collega il nuovo nodo dopo l'ultimo
      ultimo = t;           //aggiorna il riferimento ultimo
    }
  }
  ...
}
```

Esempio

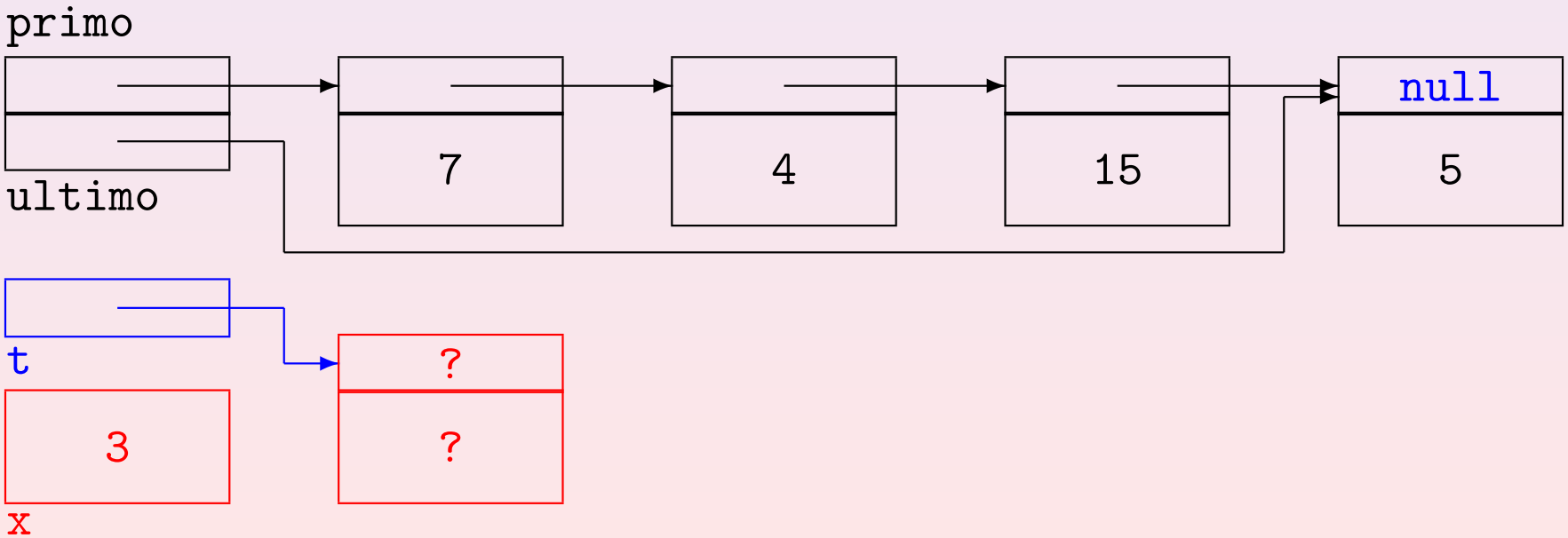



```
NodoCoda t = new NodoCoda();
t.dato = x;
t.pros = null;
```

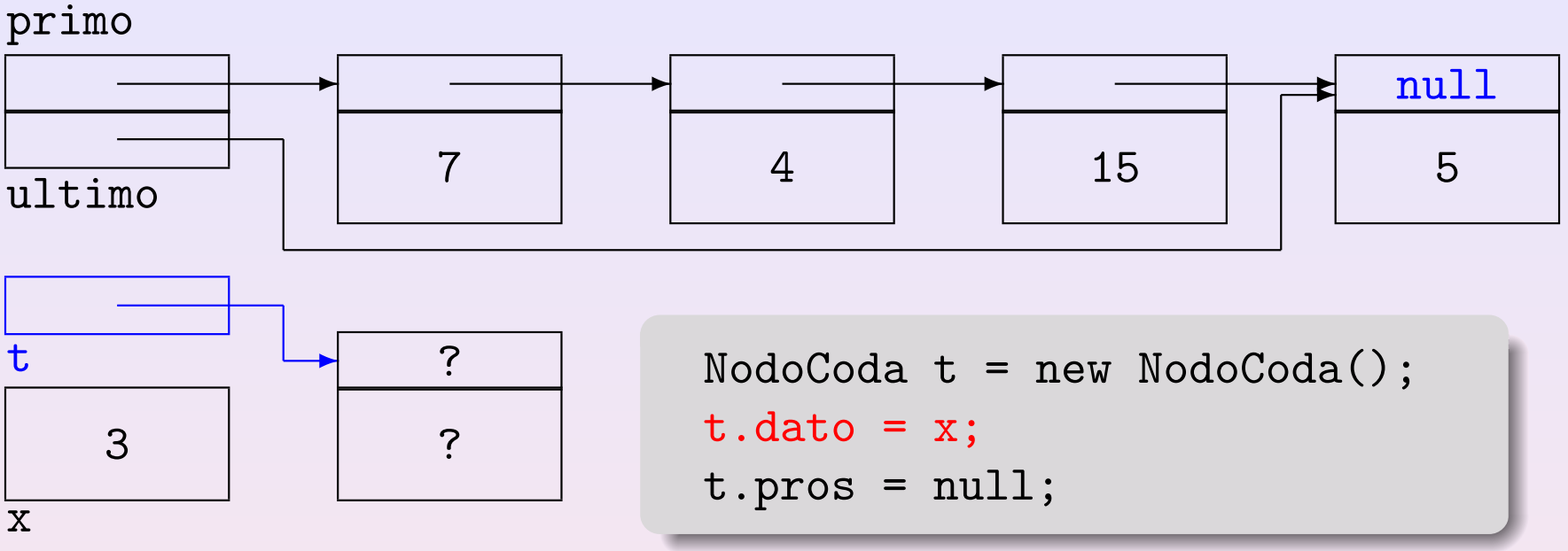
Esempio




```
NodoCoda t = new NodoCoda();
t.dato = x;
t.pros = null;
```

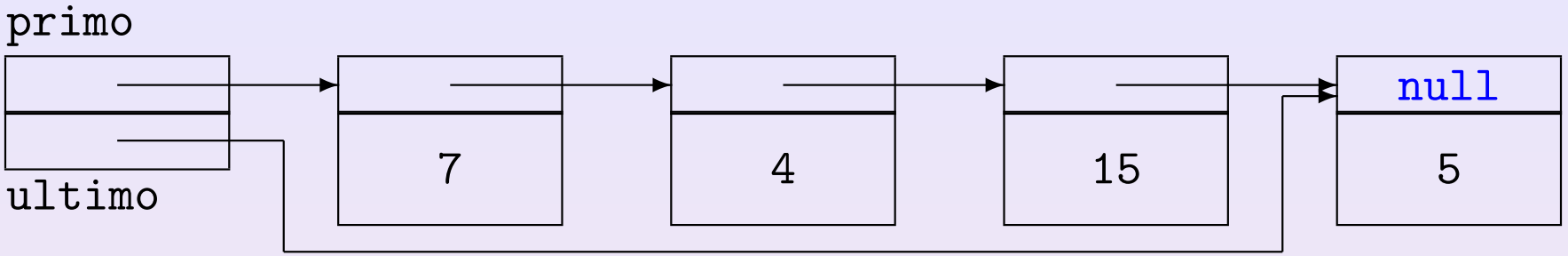


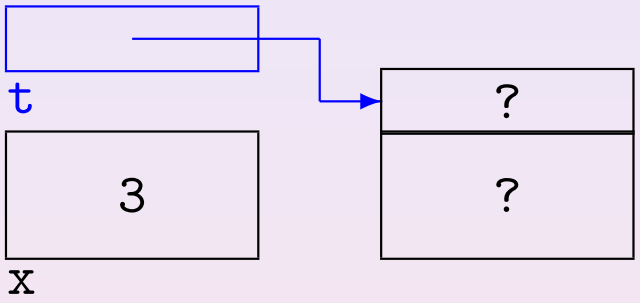
Esempio



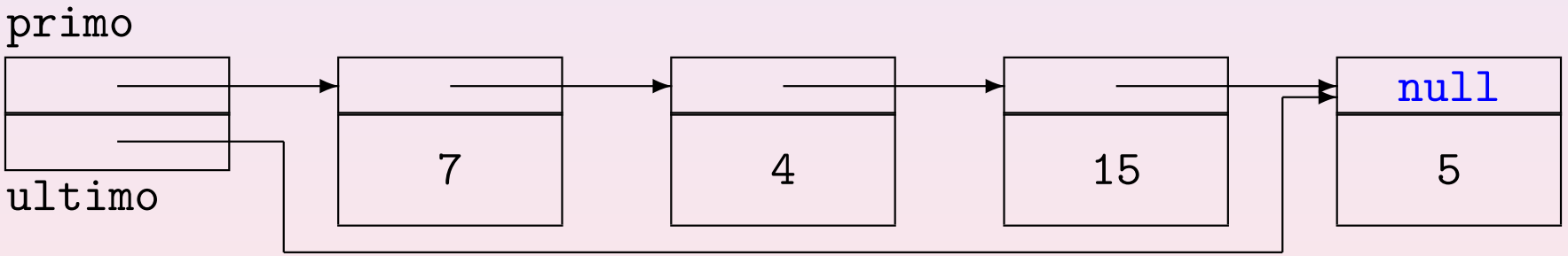

```
NodoCoda t = new NodoCoda();
t.dato = x;
t.pros = null;
```

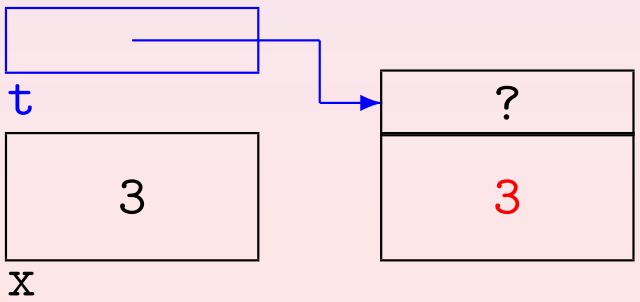
Esempio



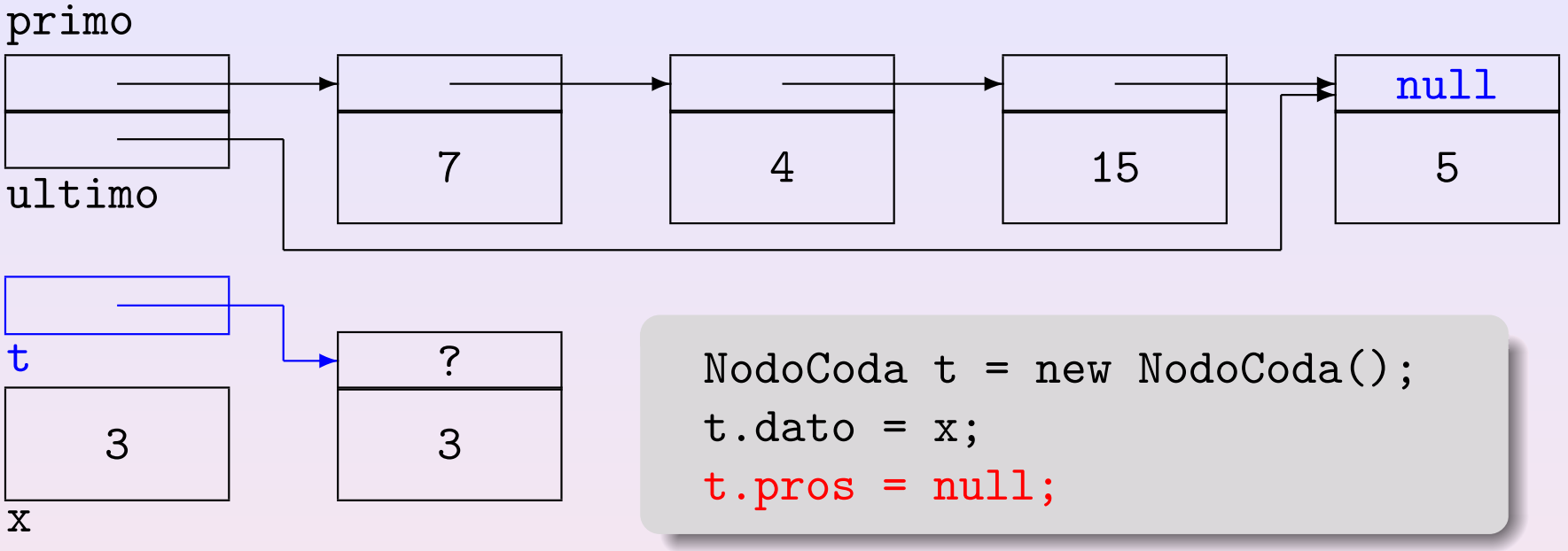



```
NodoCoda t = new NodoCoda();
t.dato = x;
t.pros = null;
```



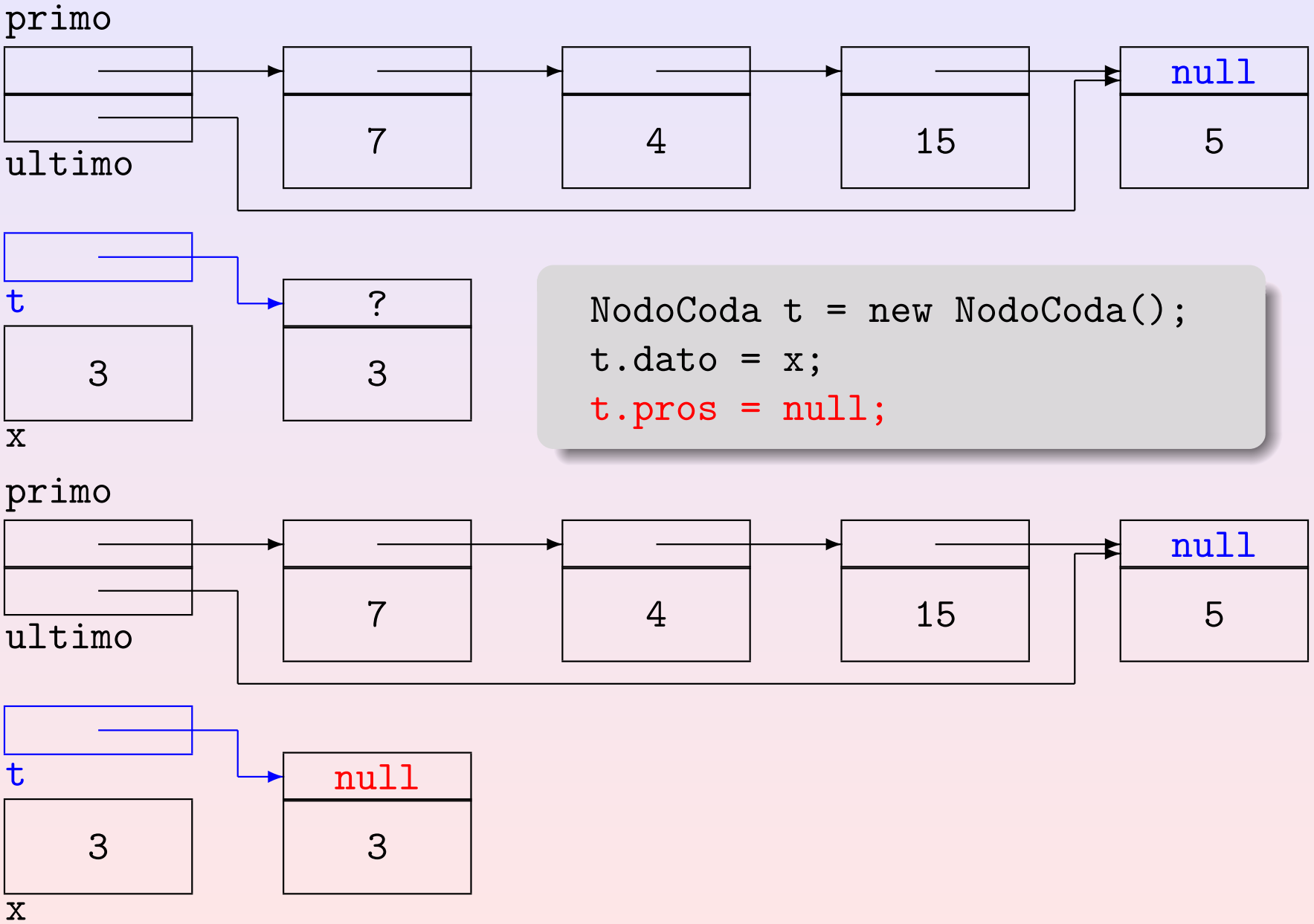


Esempio

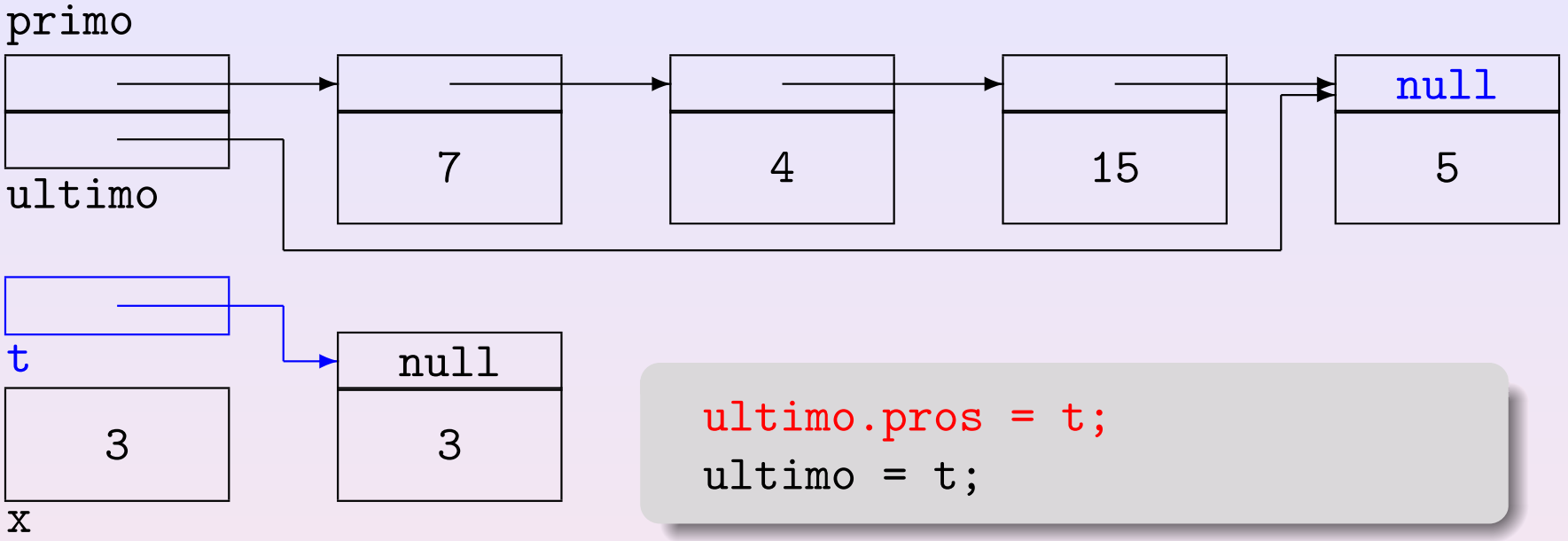



```
NodoCoda t = new NodoCoda();
t.dato = x;
t.pros = null;
```

Esempio

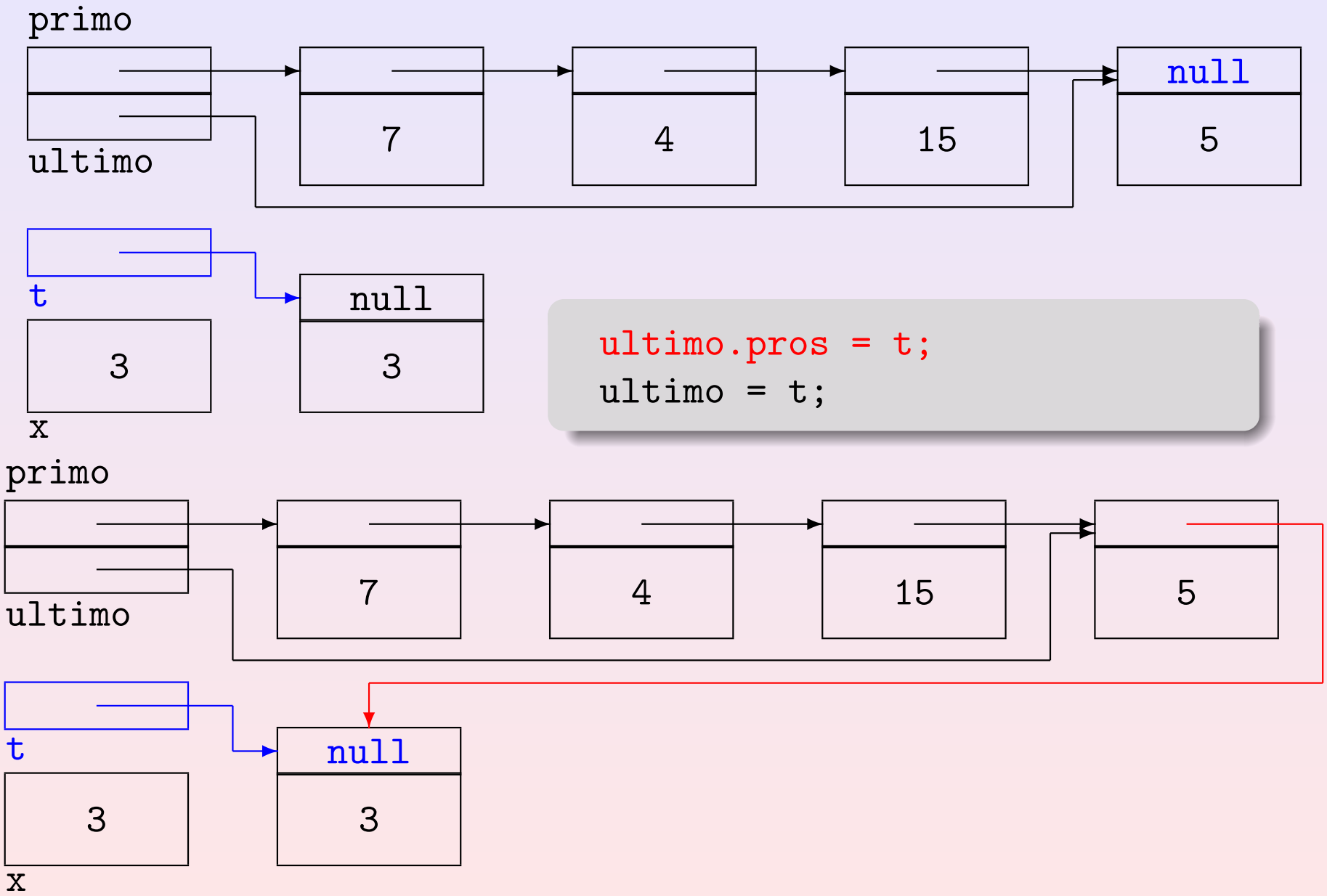

primo
ultimo
7
4
15
null
5
t
3
x
?
3
NodoCoda t = new NodoCoda();
t.dato = x;
t.pros = null;
primo
ultimo
7
4
15
null
5
t
3
x
null
3

Esempio

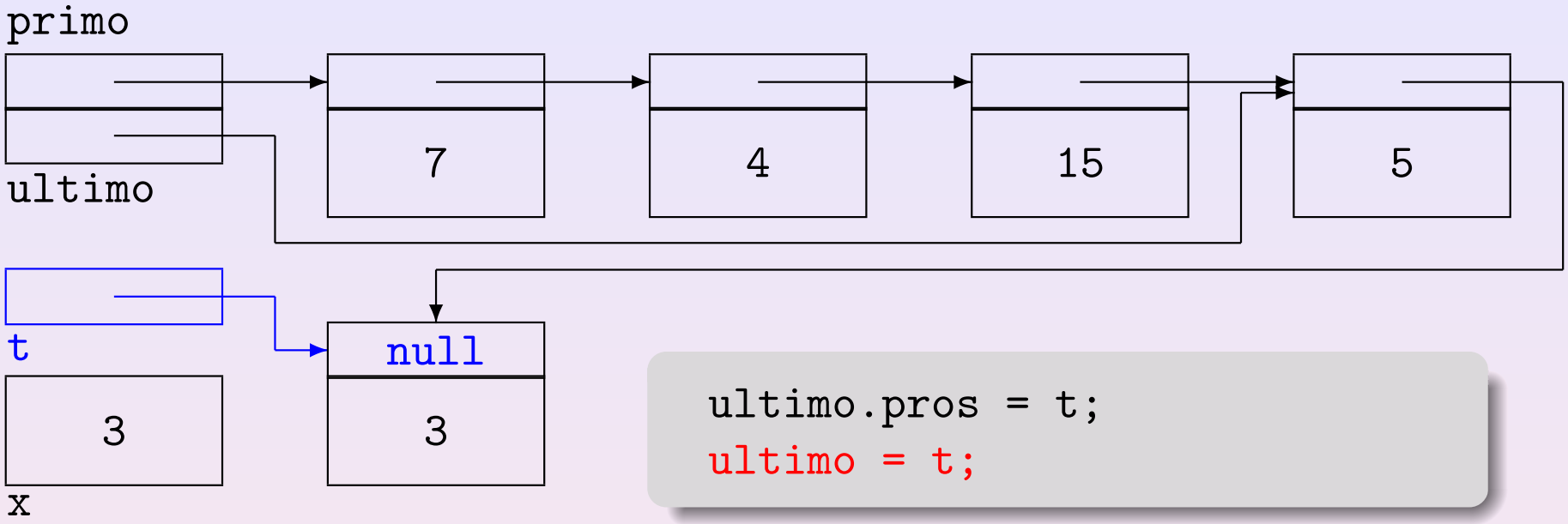



```
ultimo.pros = t;
ultimo = t;
```

Esempio

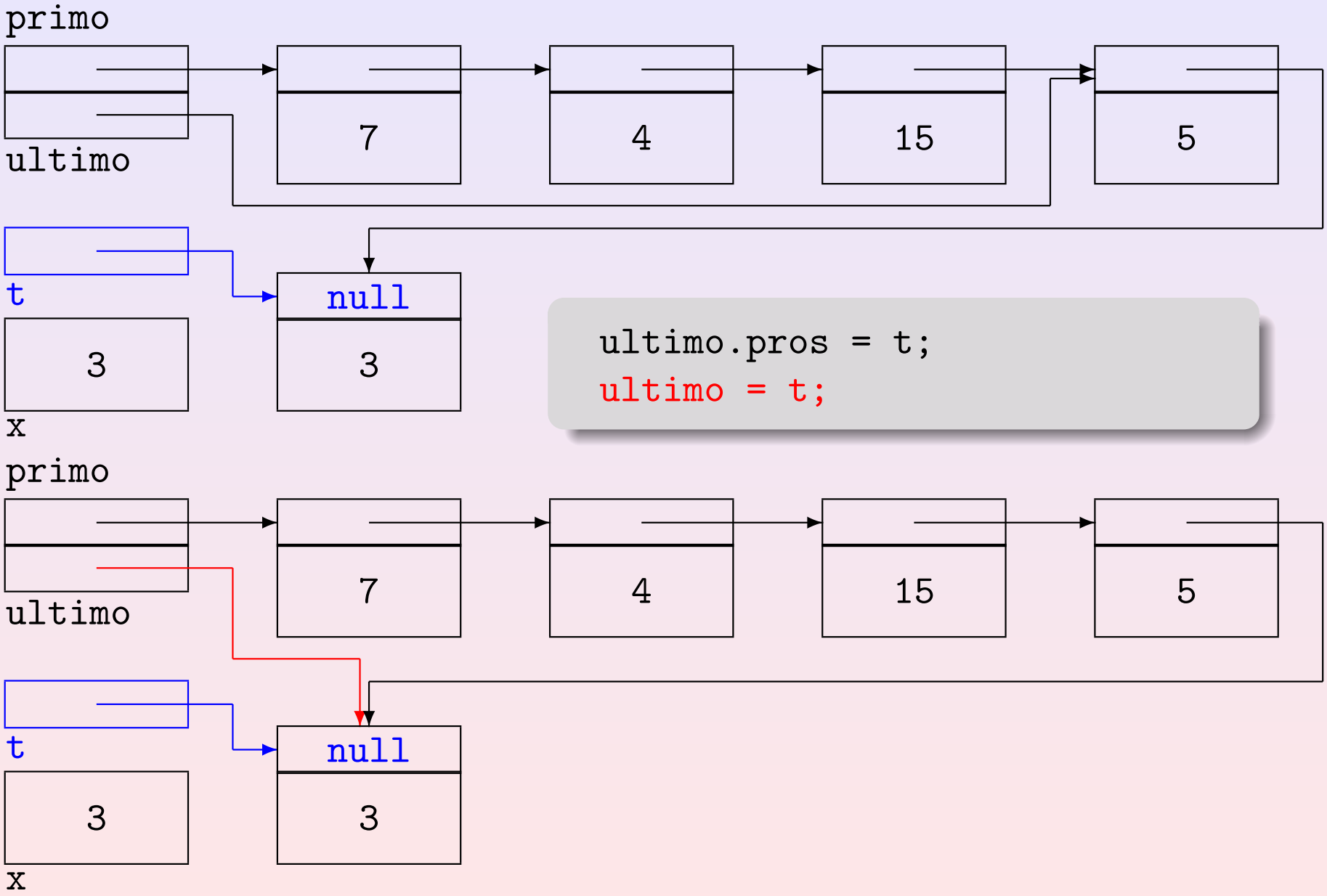

primo
ultimo
7
4
15
null
5
t
3
x
null
3
ultimo.pros = t;
ultimo = t;
primo
ultimo
7
4
15
5
t
3
x
null
3

Esempio




```
ultimo.pros = t;
ultimo = t;
```

Esempio


primo
ultimo
7
4
15
5
t
null
3
3
x
ultimo.pros = t;
ultimo = t;
primo
ultimo
7
4
15
5
t
null
3
3
x

Il metodo preleva

```
public class Coda<E> {
  private NodoCoda primo, ultimo;

  ...
  public E preleva() {
    if (primo == null)
      throw new CodaVuotaException();
    else {
      E risultato = primo.dato;
      primo = primo.pros;
      if (primo == null) //caso in cui la coda sia rimasta vuota
        ultimo = null;
      return risultato;
    }
  }
  ...
}
```

Il metodo toString

```
public class Coda<E> {
  private NodoCoda primo, ultimo;

  ...
  public String toString() {
    String s = "";
    for (NodoCoda nodo = primo; nodo != null; nodo = nodo.pros)
      s = s + nodo.dato.toString() + " ";
    return s;
  }
  ...
}
```

Il metodo toString con separatore

```
public class Coda<E> {
  private NodoCoda primo, ultimo;

  ...
  public String toString(String separatore) {
    if (primo == null)
      return "";
    else {
      String s = primo.dato.toString();
      for (NodoCoda nodo = primo.pros; nodo != null;
                                          nodo = nodo.pros)
        s = s + separatore + nodo.dato.toString();
      return s;
    }
  }
  ...
}
```

Sommario: Strutture dati dinamiche

Liste ordinate

- La posizione degli elementi all'interno della lista è determinata in base a una relazione di ordinamento

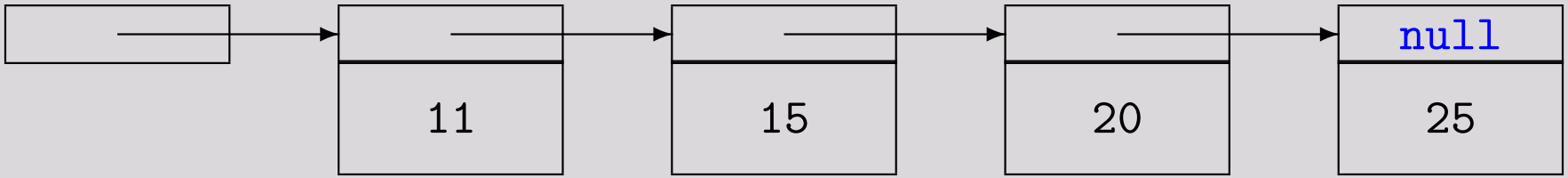
Liste ordinate

- La posizione degli elementi all'interno della lista è determinata in base a una relazione di ordinamento
- Le operazioni di inserimento vengono effettuate in modo che la struttura sia mantenuta ordinata

Liste ordinate

- La posizione degli elementi all'interno della lista è determinata in base a una relazione di ordinamento
- Le operazioni di inserimento vengono effettuate in modo che la struttura sia mantenuta ordinata

Una lista ordinata di interi



La classe `ListaOrdinata`

- Costruiremo una classe `ListaOrdinata`, nella quale i dati memorizzati saranno istanze di classi di un tipo parametro `E` che implementi l'interfaccia `Comparable`

La classe `ListaOrdinata`

- Costruiremo una classe `ListaOrdinata`, nella quale i dati memorizzati saranno istanze di classi di un tipo parametro `E` che implementi l'interfaccia `Comparable`
- Quindi richiederemo che il tipo parametro

  `E` estenda `Comparable<? super E>`

La classe ListaOrdinata

- Costruiremo una classe `ListaOrdinata`, nella quale i dati memorizzati saranno istanze di classi di un tipo parametro `E` che implementi l'interfaccia `Comparable`
- Quindi richiederemo che il tipo parametro

  `E` estenda `Comparable<? super E>`

```
public class ListaOrdinata<E extends Comparable<? super E>> {
   ...
}
```

La classe ListaOrdinata

```
public class ListaOrdinata<E extends Comparable<? super E>> {
  private NodoLista inizio;

  private class NodoLista {
    E dato;
    NodoLista pros;
  }

  //COSTRUTTORE
  public ListaOrdinata() {
    inizio = null;
  }

  //METODI
  ...
}
```

Il metodo `trova`

- `public int trova(E x)`
  Cerca l'oggetto fornito tramite il parametro e ne restituisca la posizione nella lista. Nel caso la lista non contenga l'elemento cercato, il metodo restituisce 0

Il metodo trova

- `public int trova(E x)`
  Cerca l'oggetto fornito tramite il parametro e ne restituisca la posizione nella lista. Nel caso la lista non contenga l'elemento cercato, il metodo restituisce 0

```
public int trova(E x) {
  NodoLista p = inizio;
  int posizione = 1;
  while (p != null && p.dato.compareTo(x) < 0) {
    p = p.pros;
    posizione++;
  }
  if (p == null || p.dato.compareTo(x) > 0) //se non c'e'
    return 0;
  else
    return posizione;
}
```

Il metodo inserisci

- `public void inserisci(E x))`
  Inserisce l'oggetto fornito tramite il parametro nella lista che esegue il metodo mantenendo la lista ordinata rispetto all'ordinamento stabilito dal metodo `compareTo`.

Il metodo `inserisci`

- `public void inserisci(E x))`
  Inserisce l'oggetto fornito tramite il parametro nella lista che esegue il metodo mantenendo la lista ordinata rispetto all'ordinamento stabilito dal metodo `compareTo`.
- Il metodo è composto di due parti:
  (1) ricerca della posizione dove effettuare l'inserimento

Il metodo `inserisci`

- `public void inserisci(E x))`
  Inserisce l'oggetto fornito tramite il parametro nella lista che esegue il metodo mantenendo la lista ordinata rispetto all'ordinamento stabilito dal metodo `compareTo`.
- Il metodo è composto di due parti:
  (1) ricerca della posizione dove effettuare l'inserimento
  (2) creazione e inserimento del nuovo nodo.

Ricerca della posizione

```
p = inizio;
while (p != null && p.dato.compareTo(x) < 0)
  p = p.pros;
```

Ricerca della posizione

```
p = inizio;
while (p != null && p.dato.compareTo(x) < 0)
  p = p.pros;
```

All'uscita dal ciclo la variabile p contiene:

- un riferimento al nodo prima del quale effettuare l'inserimento

Ricerca della posizione

```
p = inizio;
while (p != null && p.dato.compareTo(x) < 0)
  p = p.pros;
```

All'uscita dal ciclo la variabile `p` contiene:

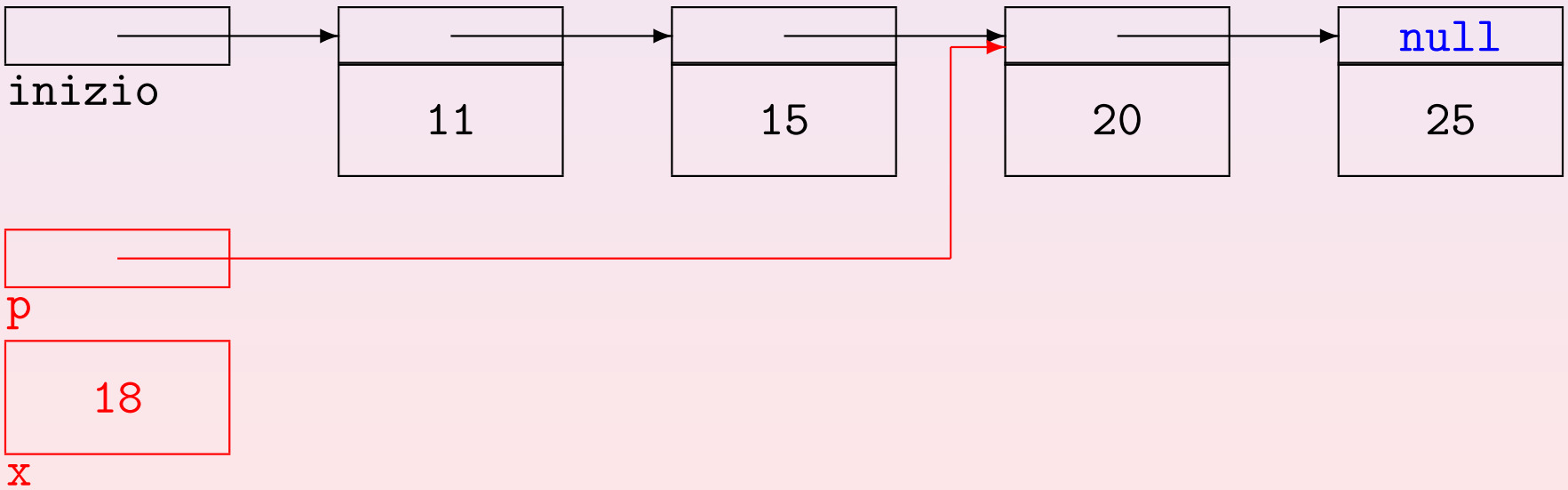
- un riferimento al nodo prima del quale effettuare l'inserimento
- oppure `null` nel caso l'inserimento vada fatto alla fine della lista

Ricerca della posizione

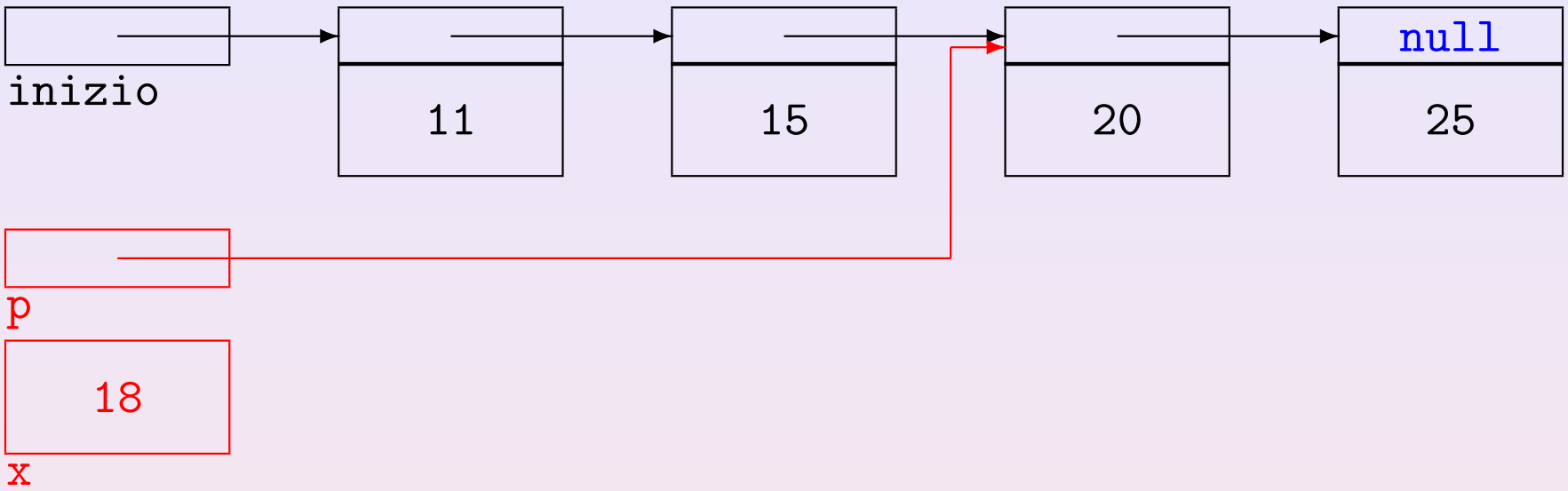
```
p = inizio;
while (p != null && p.dato.compareTo(x) < 0)
  p = p.pros;
```

All'uscita dal ciclo la variabile p contiene:

- un riferimento al nodo prima del quale effettuare l'inserimento
- oppure null nel caso l'inserimento vada fatto alla fine della lista

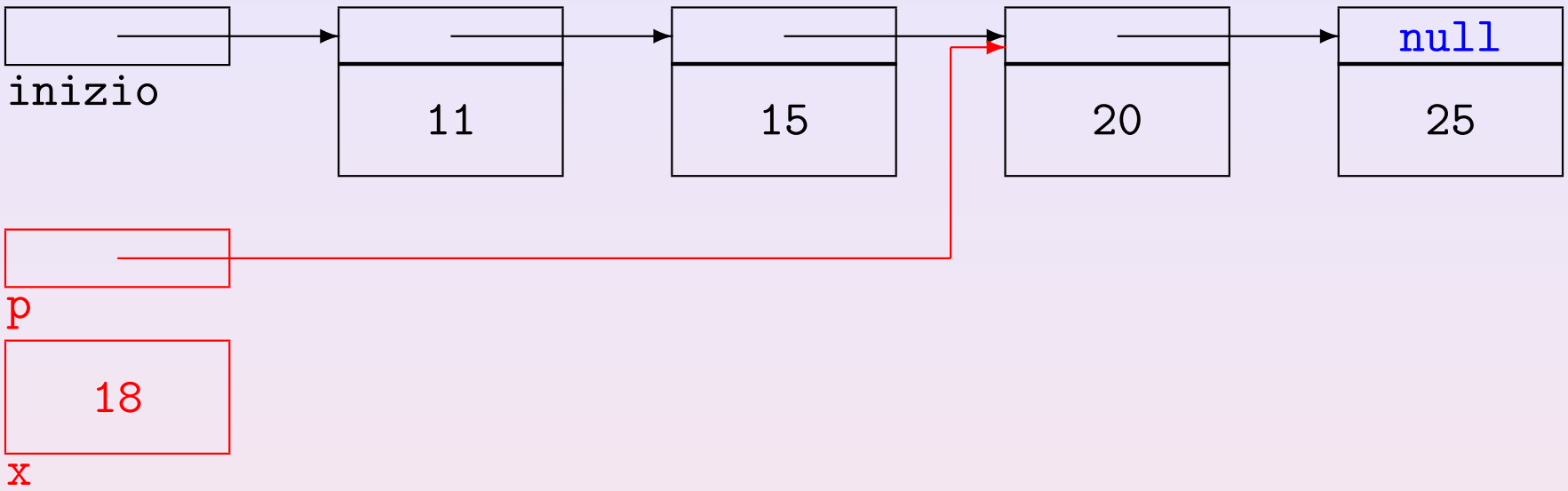


Ricerca della posizione



- Per effettuare l'inserimento occorre modificare il campo `pros` del nodo che precede quello riferito da `p`

Ricerca della posizione



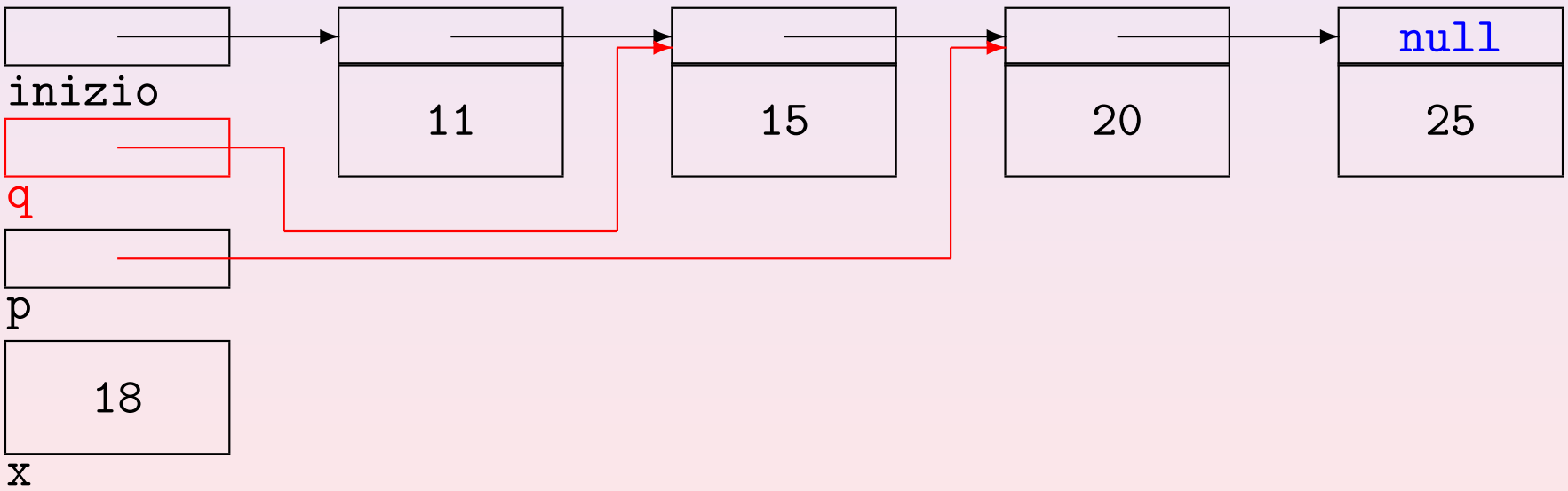
- Per effettuare l'inserimento occorre modificare il campo `pros` del nodo che precede quello riferito da `p`
- Introduciamo un riferimento ausiliario `q` che a ogni passo della fase di ricerca punterà al nodo che precede quello puntato da `p`

Ricerca della posizione

```
NodoLista p = inizio, q = null;
while (p != null && p.dato.compareTo(x) < 0) {
  q = p;
  p = p.pros;
}
```

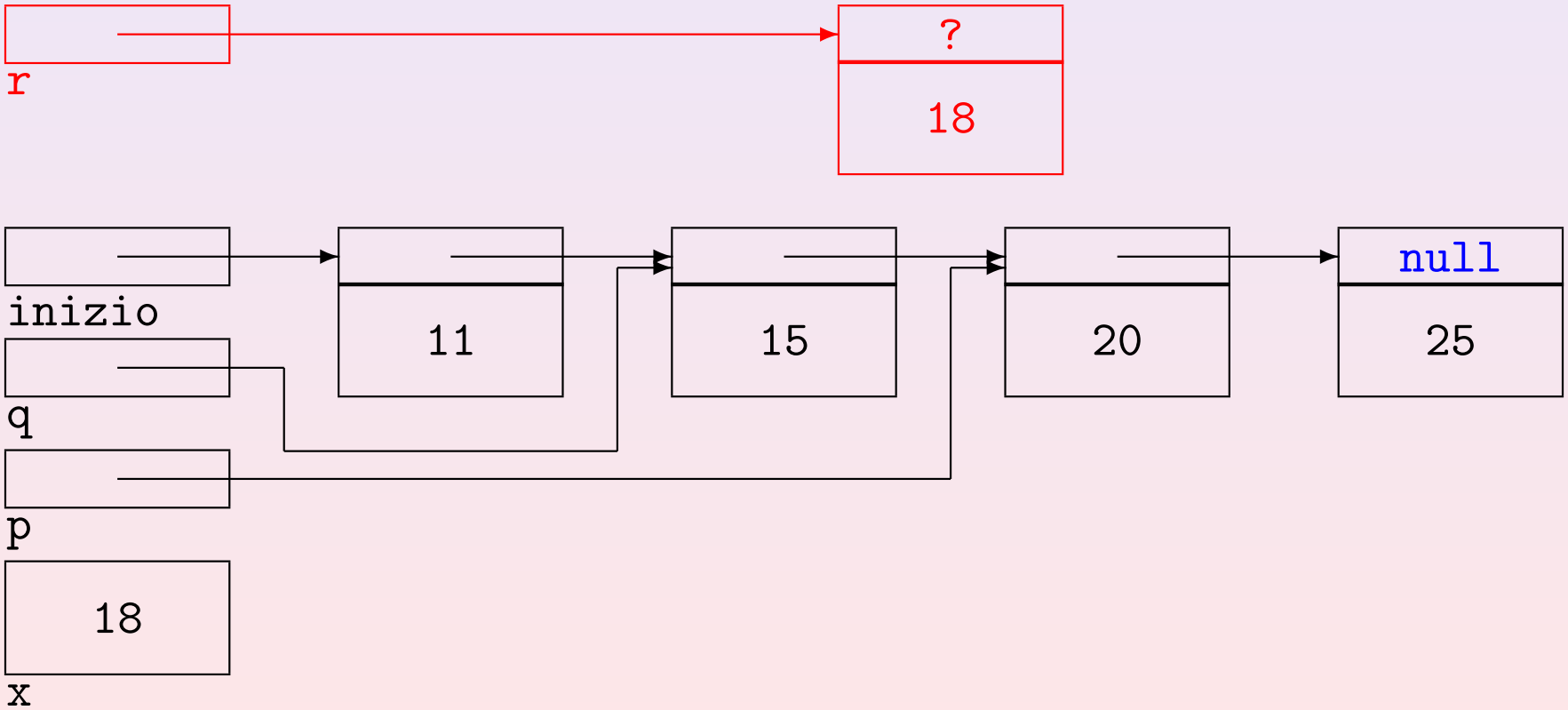
Ricerca della posizione

```
NodoLista p = inizio, q = null;
while (p != null && p.dato.compareTo(x) < 0) {
  q = p;
  p = p.pros;
}
```



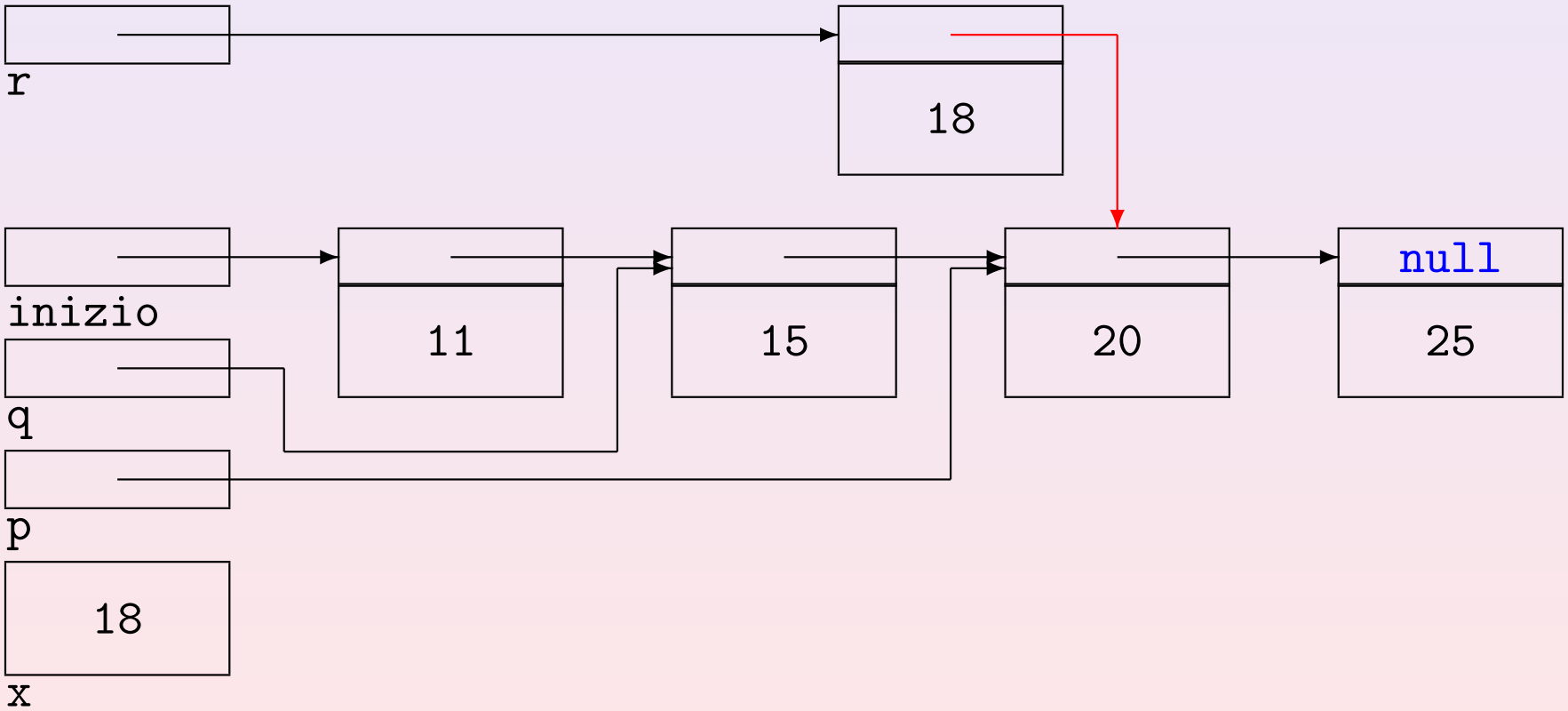
Inserimento del nuovo nodo

```
...
NodoLista r = new NodoLista();
r.dato = x;
...
```



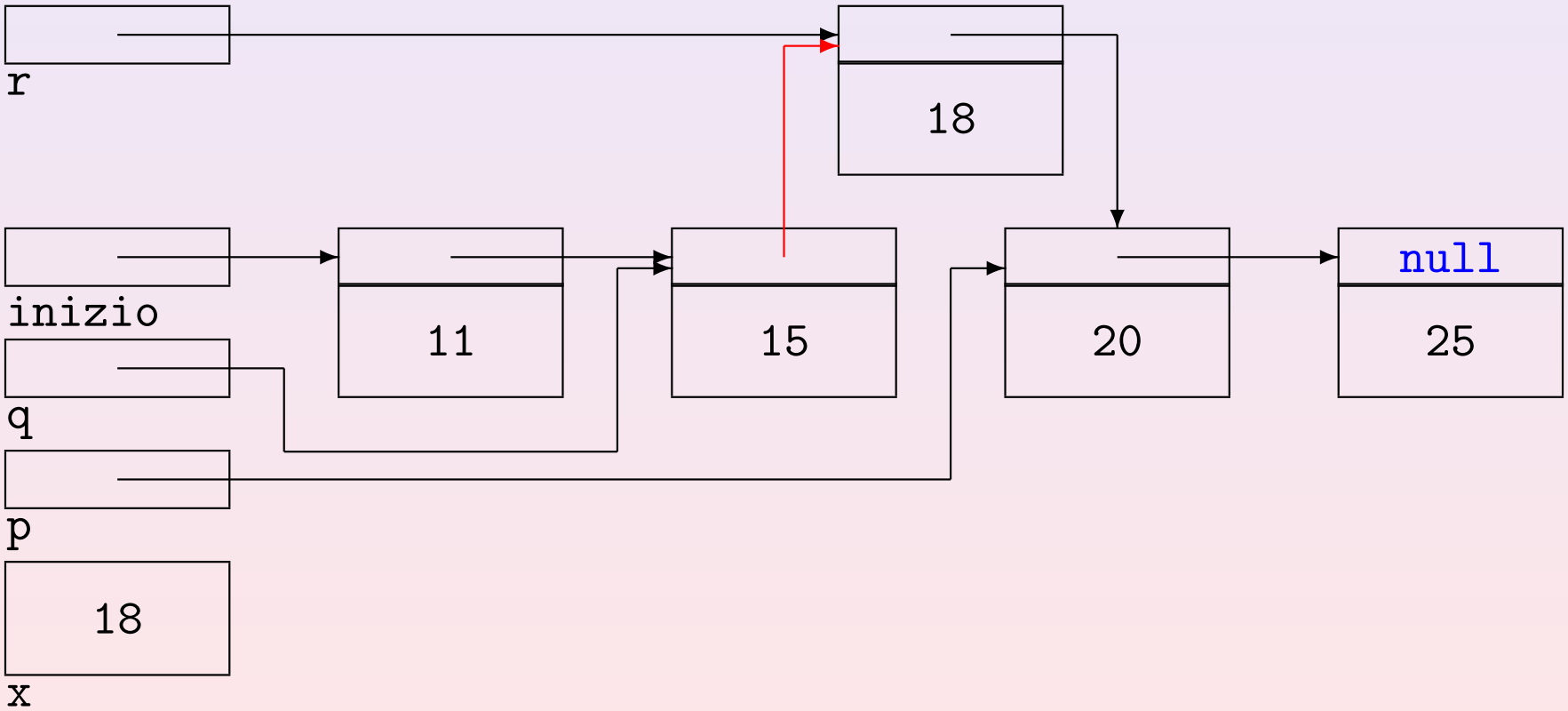
Inserimento del nuovo nodo

```
...
r.pros = p
...
```



Inserimento del nuovo nodo

```
...
q.pros = r
...
```



Il metodo inserisci

```
public void inserisci(E x) {
  //ricerca della posizione per l'inserimento
  NodoLista p = inizio, q = null;
  while (p != null && p.dato.compareTo(x) < 0) {
    q = p;
    p = p.pros;
  }
  //creazione del nuovo nodo
  NodoLista r = new NodoLista();
  r.dato = x;
  //inserimento del nodo nella lista
  r.pros = p;
  if (q == null)  //inserimento all'inizio
    inizio = r;
  else            //inserimento dopo il nodo riferito da q
    q.pros = r;
}
```

Il metodo `cancella`

- `public void cancella(E x)`
  Cancella dalla lista ordinata la prima occorrenza di un oggetto uguale a quello fornito tramite l'argomento, se presente.

Il metodo `cancella`

- `public void cancella(E x)`
  Cancella dalla lista ordinata la prima occorrenza di un oggetto uguale a quello fornito tramite l'argomento, se presente.
- Struttura del metodo:
  (1) ricerca della posizione dove effettuare la cancellazione

Il metodo `cancella`

- `public void cancella(E x)`
  Cancella dalla lista ordinata la prima occorrenza di un oggetto uguale a quello fornito tramite l'argomento, se presente.
- Struttura del metodo:
  (1) ricerca della posizione dove effettuare la cancellazione
  (2) rimozione dell'elemento (da effettuare solo nel caso l'elemento sia stato trovato)

Ricerca dell'elemento da cancellare

```
NodoLista p = inizio, q = null;
while (p != null && p.dato.compareTo(x) < 0) {
  q = p;
  p = p.pros;
}
```

Ricerca dell'elemento da cancellare

```
NodoLista p = inizio, q = null;
while (p != null && p.dato.compareTo(x) < 0) {
  q = p;
  p = p.pros;
}
```

Al termine dell'esecuzione del ciclo possono verificarsi le seguenti situazioni:

- `p` contiene `null` (l'elemento non è stato trovato)

Ricerca dell'elemento da cancellare

```
NodoLista p = inizio, q = null;
while (p != null && p.dato.compareTo(x) < 0) {
  q = p;
  p = p.pros;
}
```

Al termine dell'esecuzione del ciclo possono verificarsi le seguenti situazioni:

- `p` contiene `null` (l'elemento non è stato trovato)
- `p` contiene il riferimento a un oggetto diverso (maggiore) da quello cercato (l'elemento non è stato trovato)

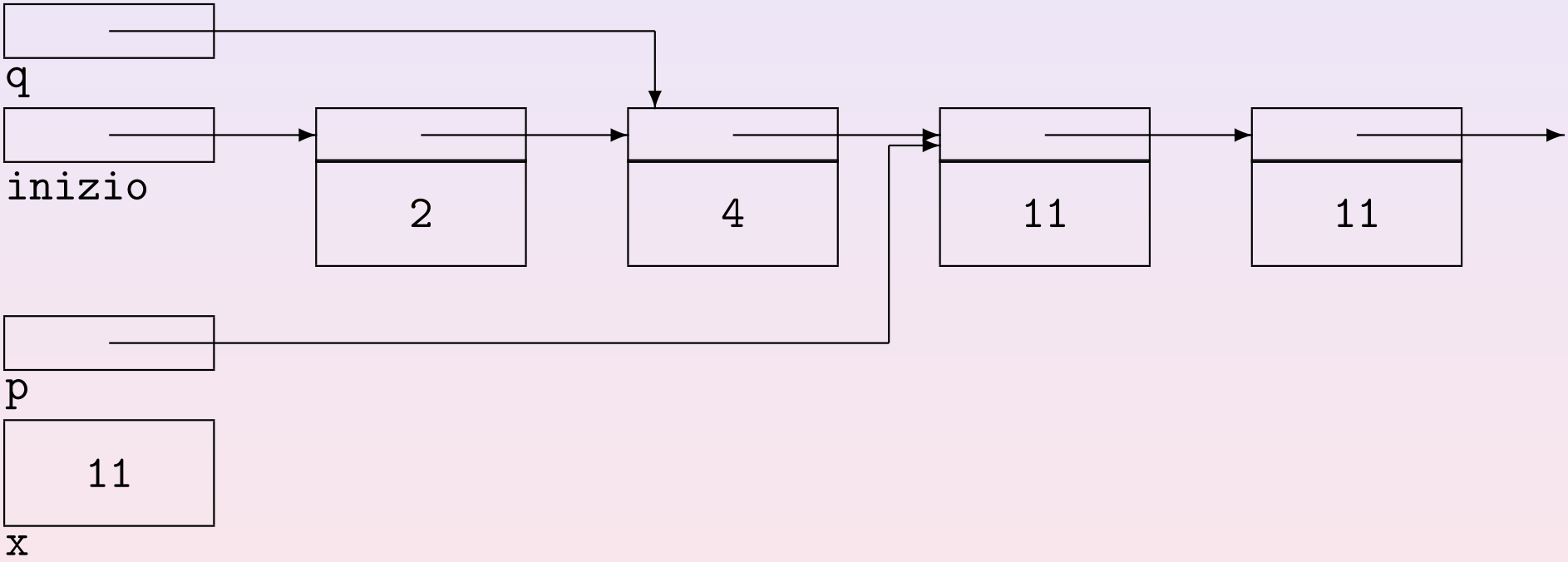
Ricerca dell'elemento da cancellare

```
NodoLista p = inizio, q = null;
while (p != null && p.dato.compareTo(x) < 0) {
  q = p;
  p = p.pros;
}
```

Al termine dell'esecuzione del ciclo possono verificarsi le seguenti situazioni:

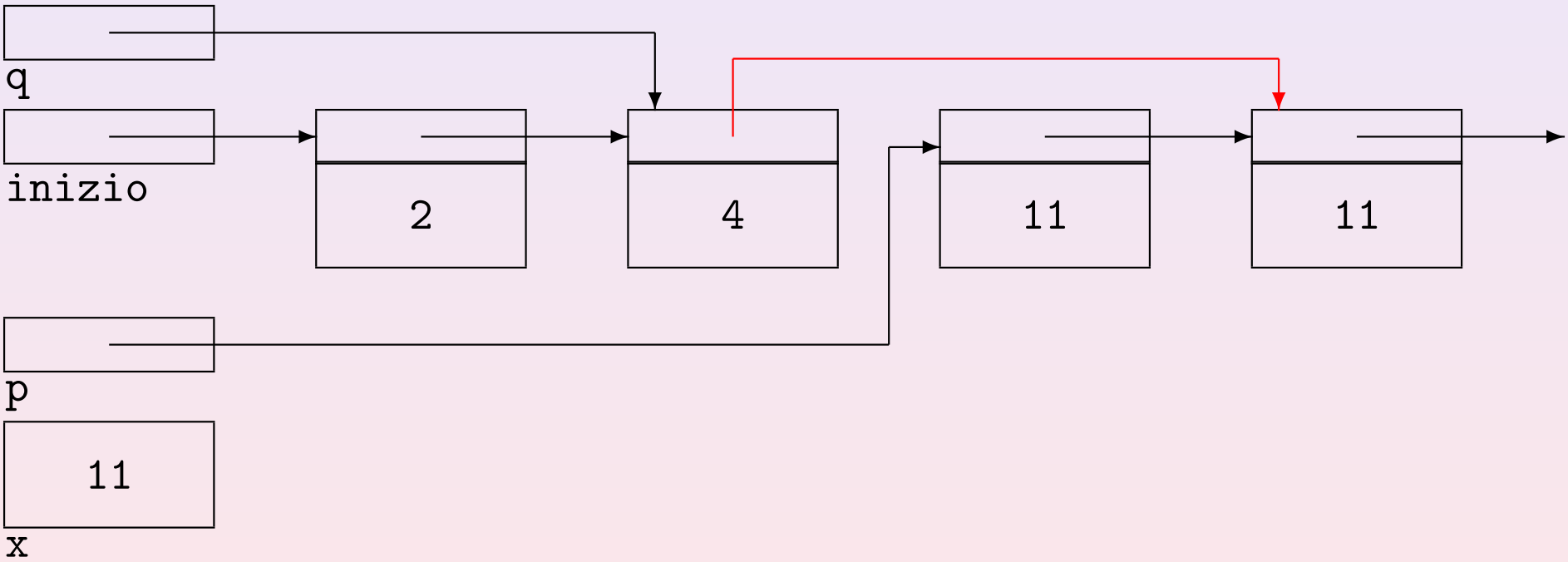
- `p` contiene `null` (l'elemento non è stato trovato)
- `p` contiene il riferimento a un oggetto diverso (maggiore) da quello cercato (l'elemento non è stato trovato)
- `p` contiene il riferimento a un oggetto uguale a quello da eliminare

L'elemento è stato trovato

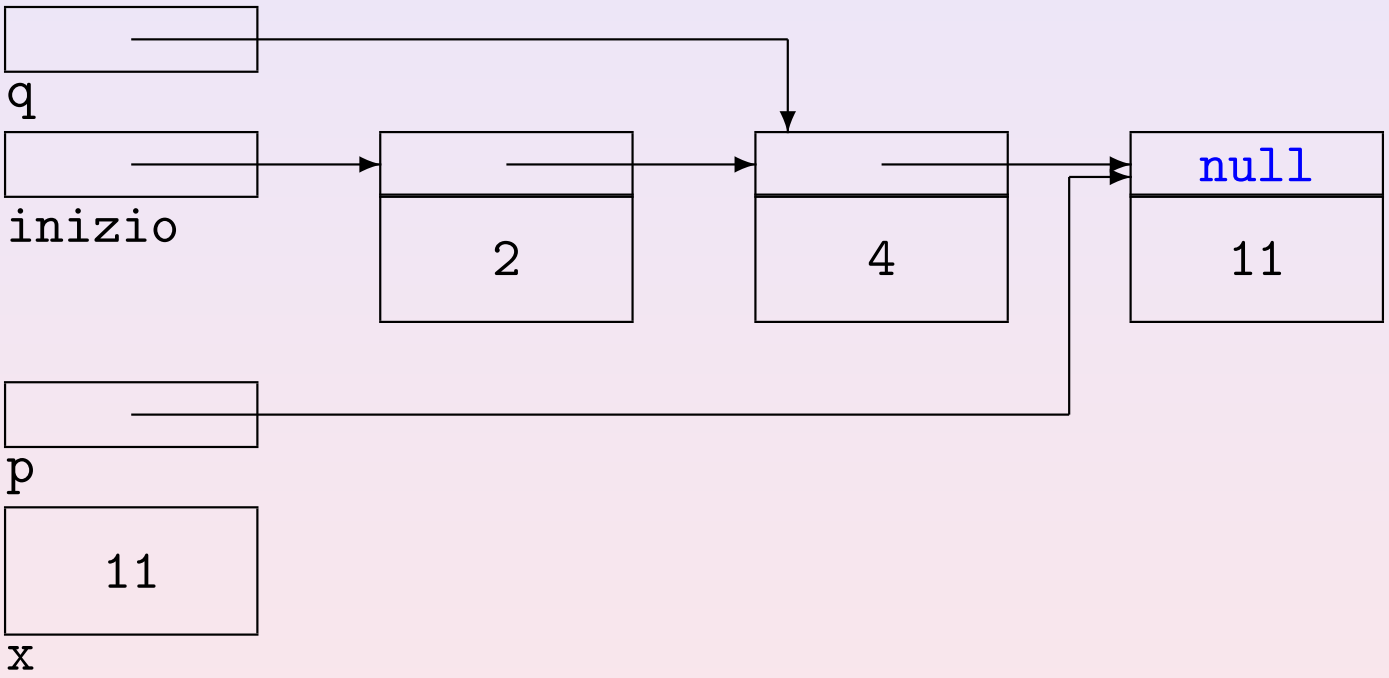


Sganciare l'elemento

```
q.pros = p.pros
```

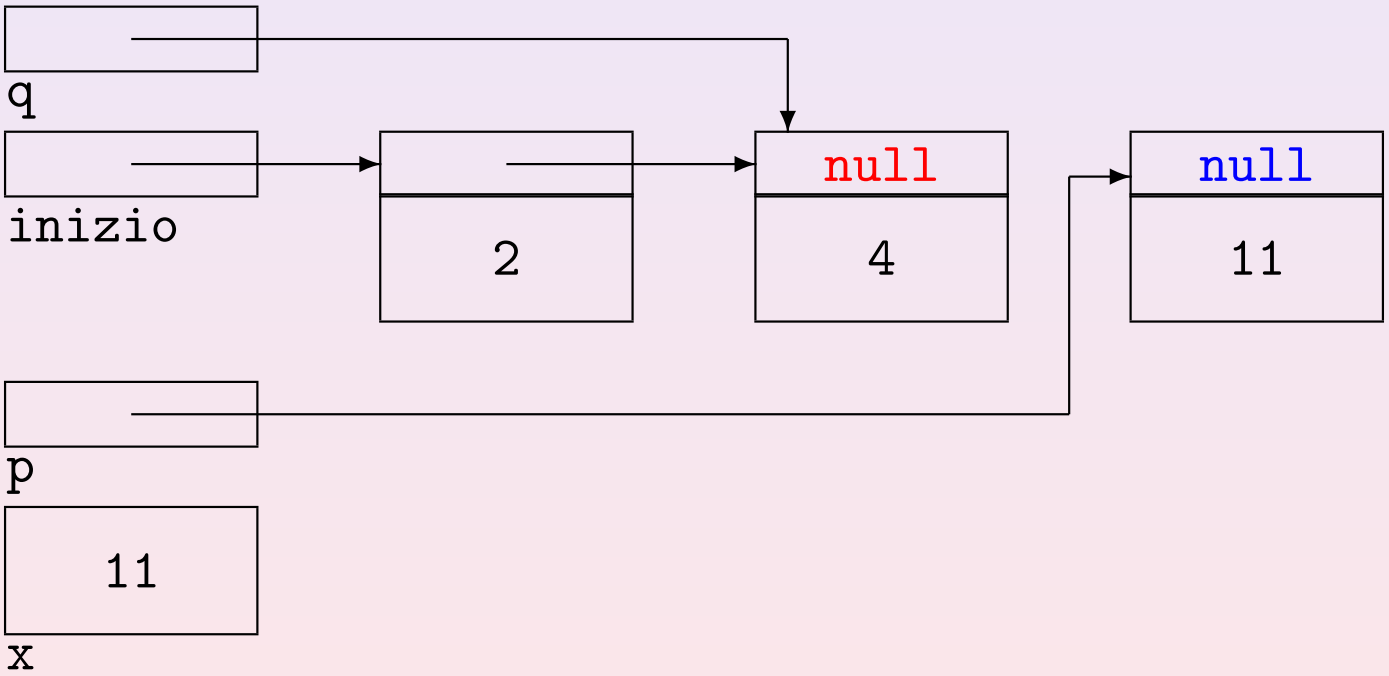


Cancellazione: ultimo elemento della lista

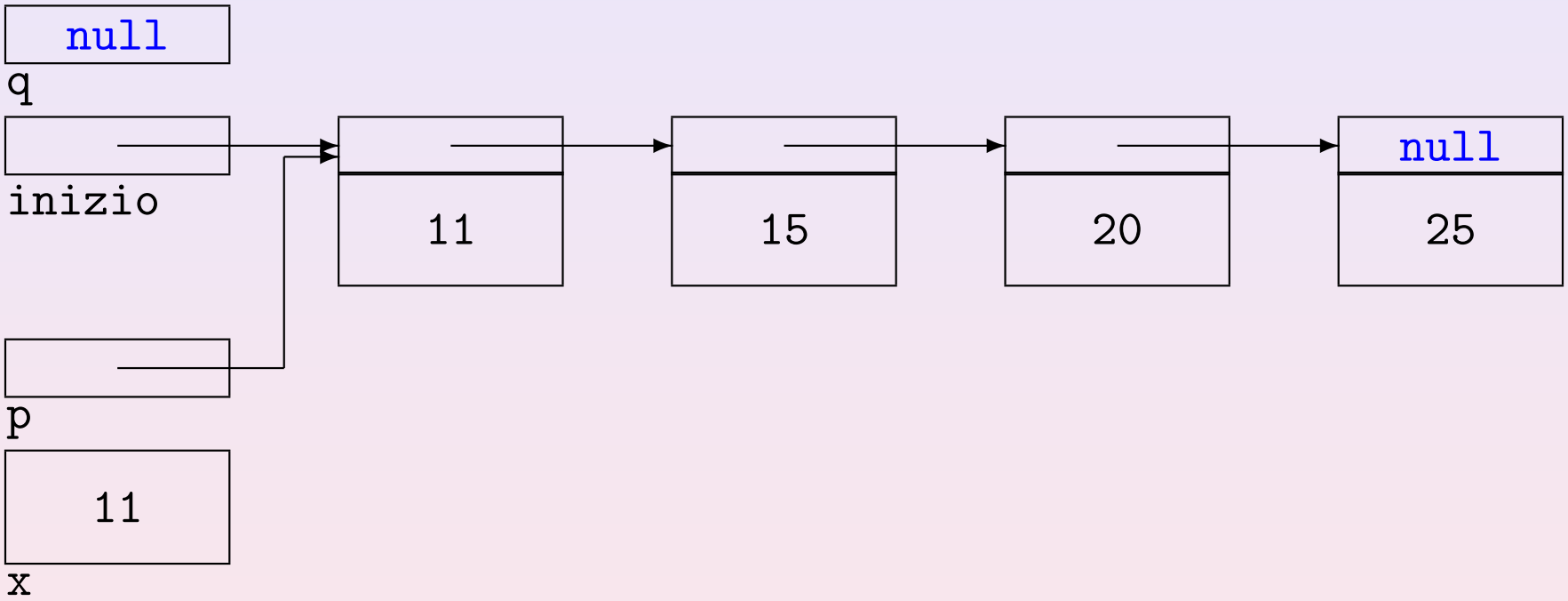


Cancellazione: ultimo elemento della lista

```
q.pros = p.pros;
```

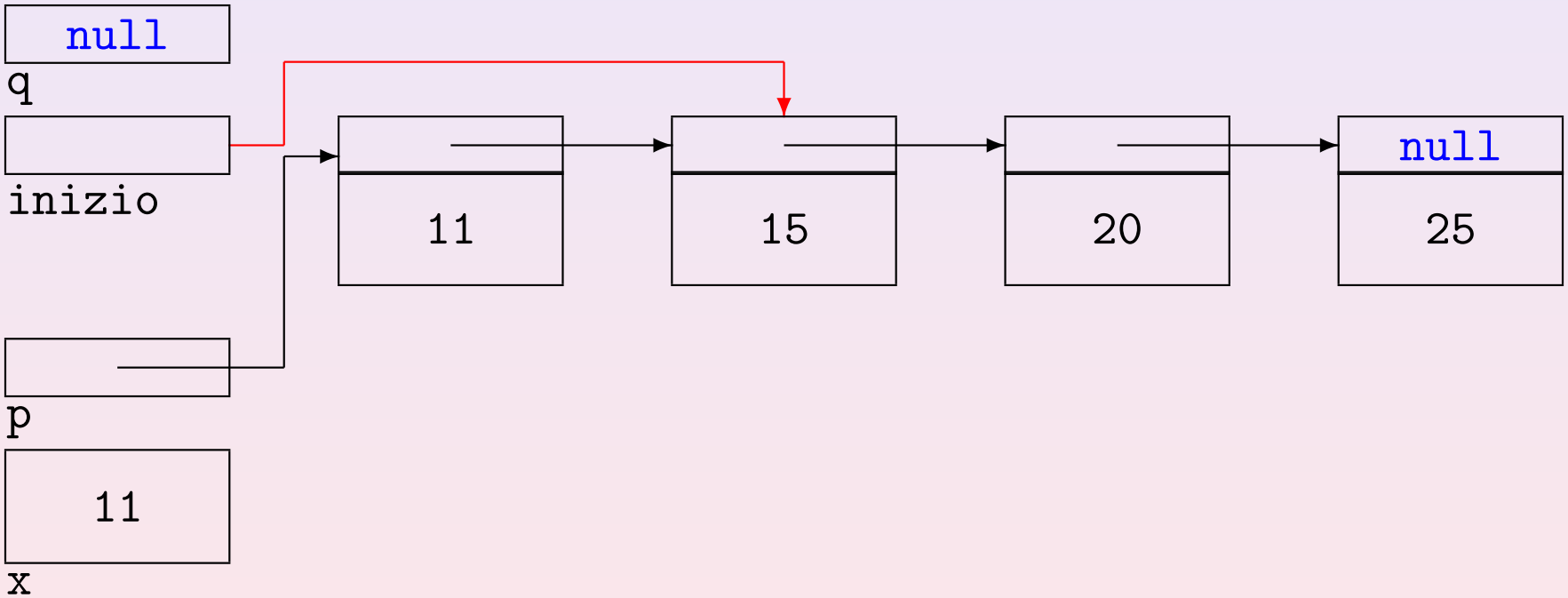


Cancellazione: primo elemento della lista



Cancellazione: primo elemento della lista

```
inizio = inizio.pros
```



Il metodo cancella

```
public void cancella(E x) {
  //ricerca della posizione per l'inserimento
  NodoLista p = inizio, q = null;
  while (p != null && p.dato.compareTo(x) < 0) {
    q = p;
    p = p.pros;
  }

  //eliminazione del nodo
  if (p != null && p.dato.equals(x))
    if (q == null)  //cancellazione all'inizio
      inizio = inizio.pros;
    else            //cancellazione dopo il nodo riferito da q
      q.pros = p.pros;
}
```

Sommario: Strutture dati dinamiche

Alberi binari

Albero binario

Un albero binario è

Alberi binari

Albero binario

Un albero binario è

- la struttura vuota, oppure

Alberi binari

Albero binario

Un albero binario è

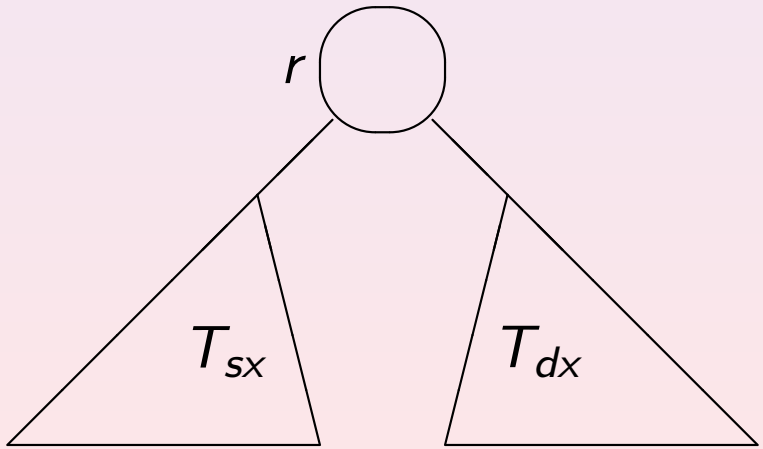
- la struttura vuota, oppure
- un nodo (*radice*), cui sono associate due alberi binari (*sottoalbero sinistro* e *sottoalbero destro*)

Alberi binari

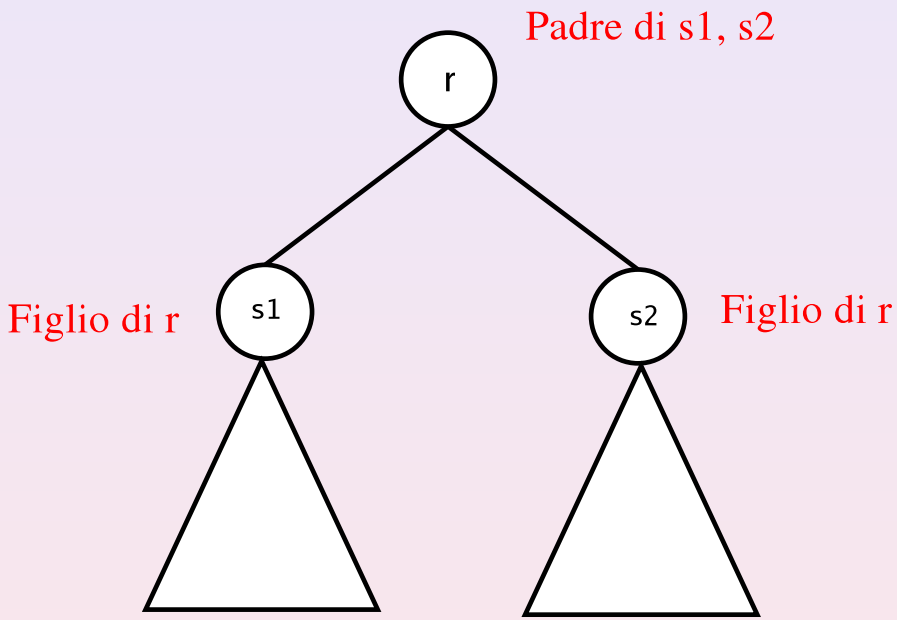
Albero binario

Un albero binario è

- la struttura vuota, oppure
- un nodo (*radice*), cui sono associate due alberi binari (*sottoalbero sinistro* e *sottoalbero destro*)



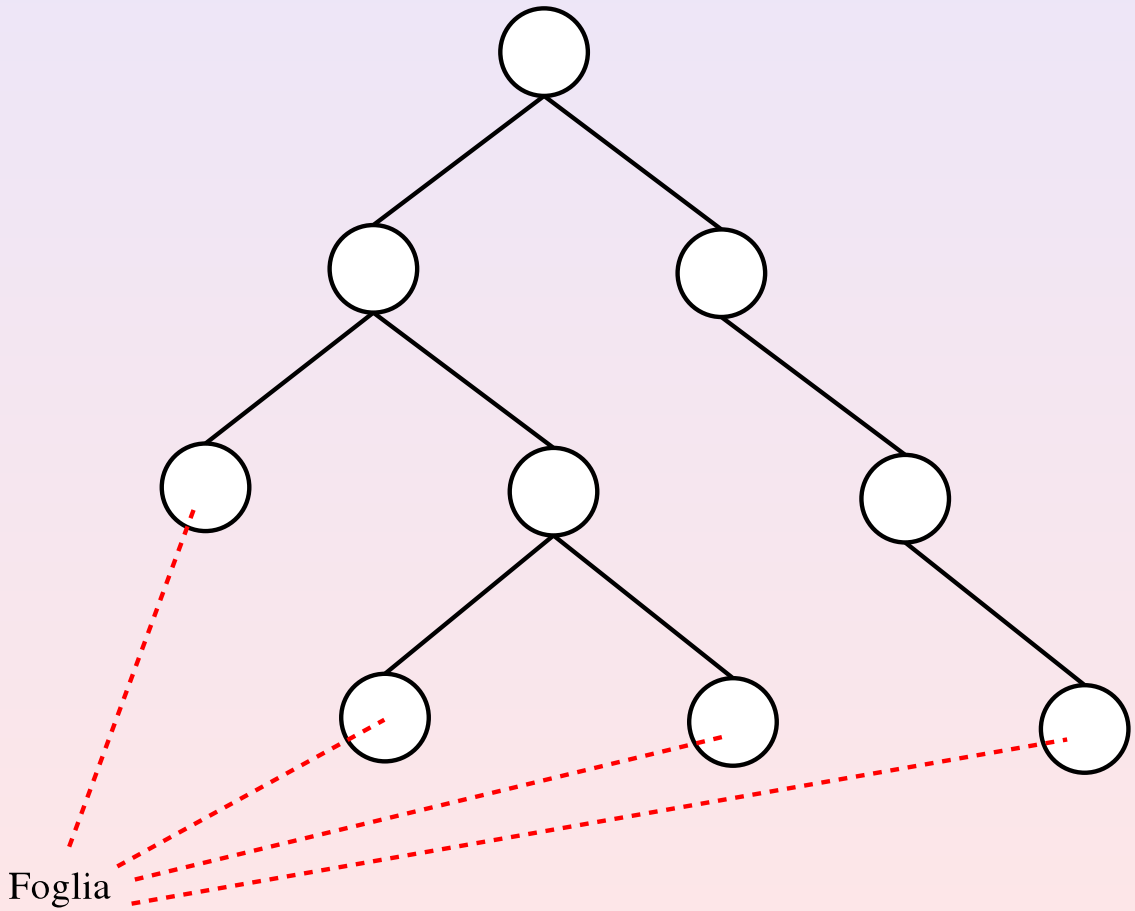
Terminologia: padre, figlio



Terminologia: foglia

Foglia

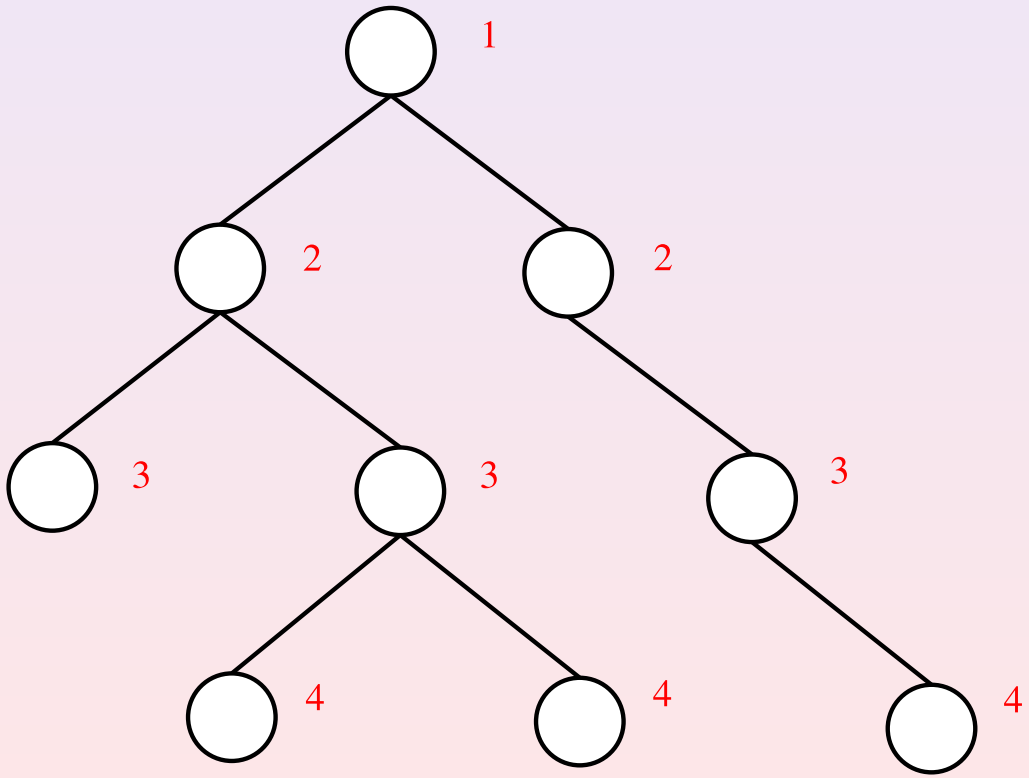
Una foglia è un nodo privo di figli.



Terminologia: livello di T

Livello

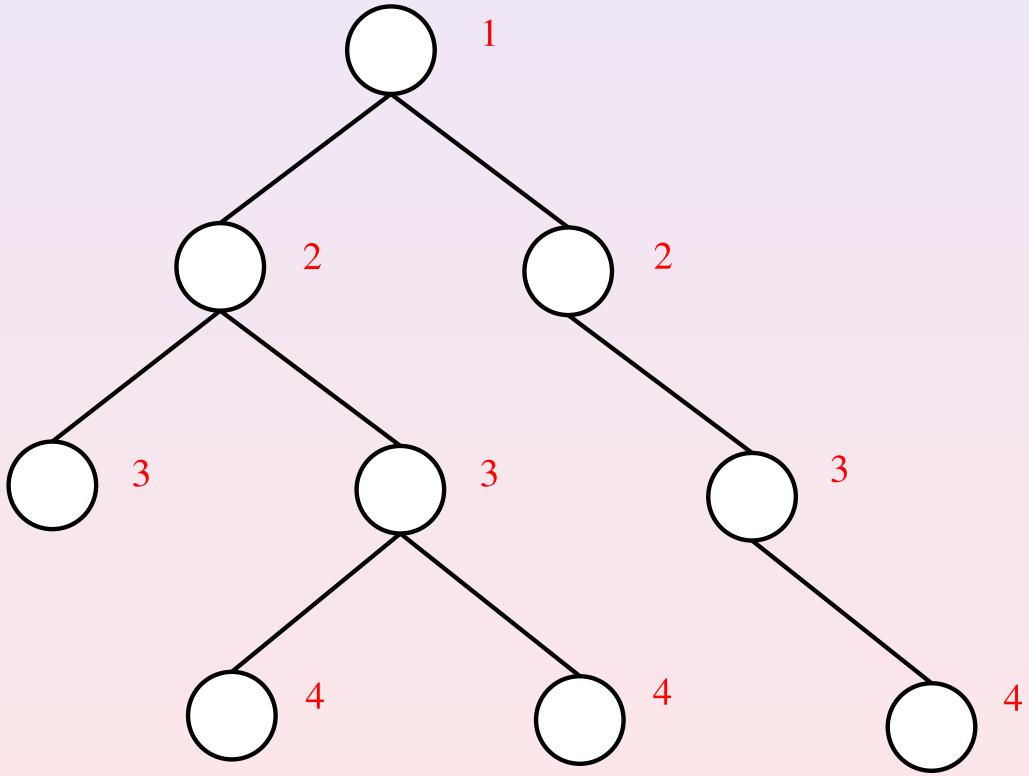
- il livello della radice di T è 1
- il livello dei figli di un nodo di livello i è $i + 1$



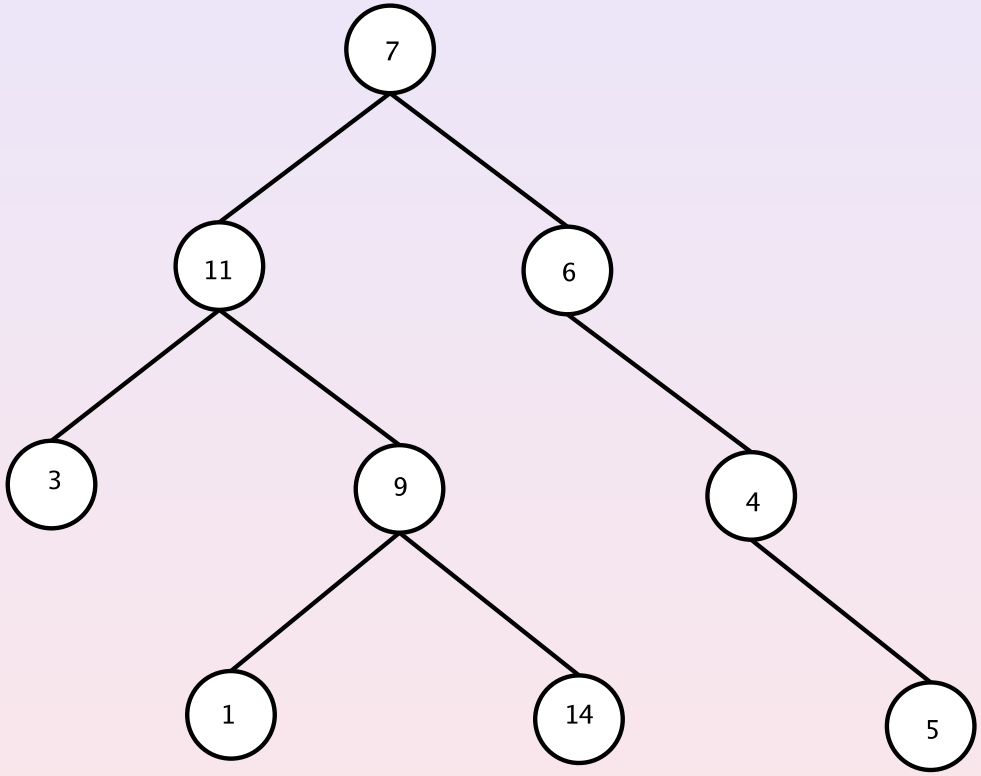
Terminologia: profondità di T

Profondità

Il massimo fra i livelli dei nodi dell'albero.



Un albero di interi

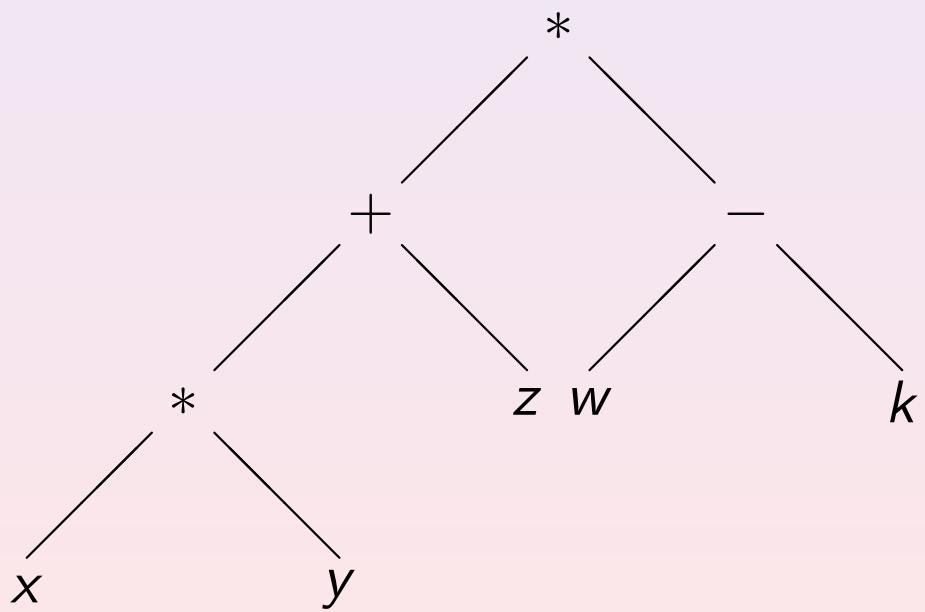


Rappresentazione di espressioni

L'espressione

$$(x * y + z) * (w - k)$$

può essere rappresentata dal seguente albero:



Attraversamento di un albero

Per visitare tutti i nodi di un albero, si possono definire strategie differenti.

- Ordine anticipato (preordine)

  radice $\Rightarrow$ sottoalbero sinistro $\Rightarrow$ sottoalbero destro

Attraversamento di un albero

Per visitare tutti i nodi di un albero, si possono definire strategie differenti.

- Ordine anticipato (preordine)

  radice $\Rightarrow$ sottoalbero sinistro $\Rightarrow$ sottoalbero destro

- Ordine simmetrico (inordine)

  sottoalbero sinistro $\Rightarrow$ radice $\Rightarrow$ sottoalbero destro

Attraversamento di un albero

Per visitare tutti i nodi di un albero, si possono definire strategie differenti.

- Ordine anticipato (preordine)

  radice $\Rightarrow$ sottoalbero sinistro $\Rightarrow$ sottoalbero destro

- Ordine simmetrico (inordine)

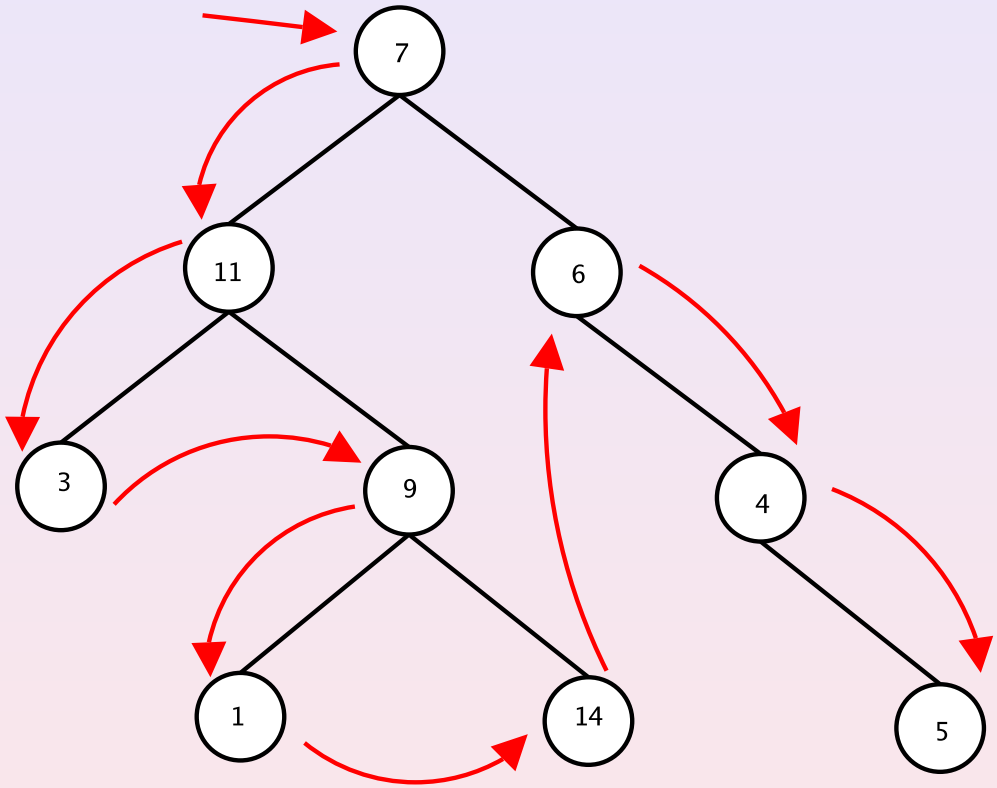
  sottoalbero sinistro $\Rightarrow$ radice $\Rightarrow$ sottoalbero destro

- Ordine posticipato (postordine)

  sottoalbero sinistro $\Rightarrow$ sottoalbero destro $\Rightarrow$ radice

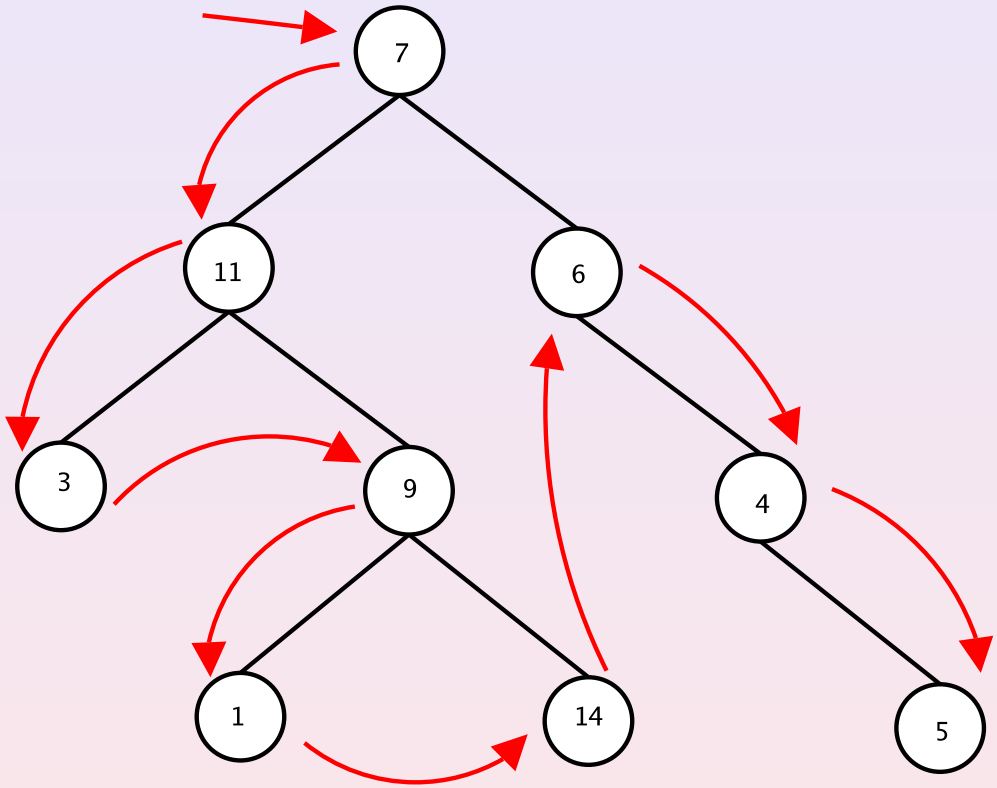
Visita in ordine anticipato (preordine)

radice $\Rightarrow$ sottoalbero sinistro $\Rightarrow$ sottoalbero destro



Visita in ordine anticipato (preordine)

radice $\Rightarrow$ sottoalbero sinistro $\Rightarrow$ sottoalbero destro

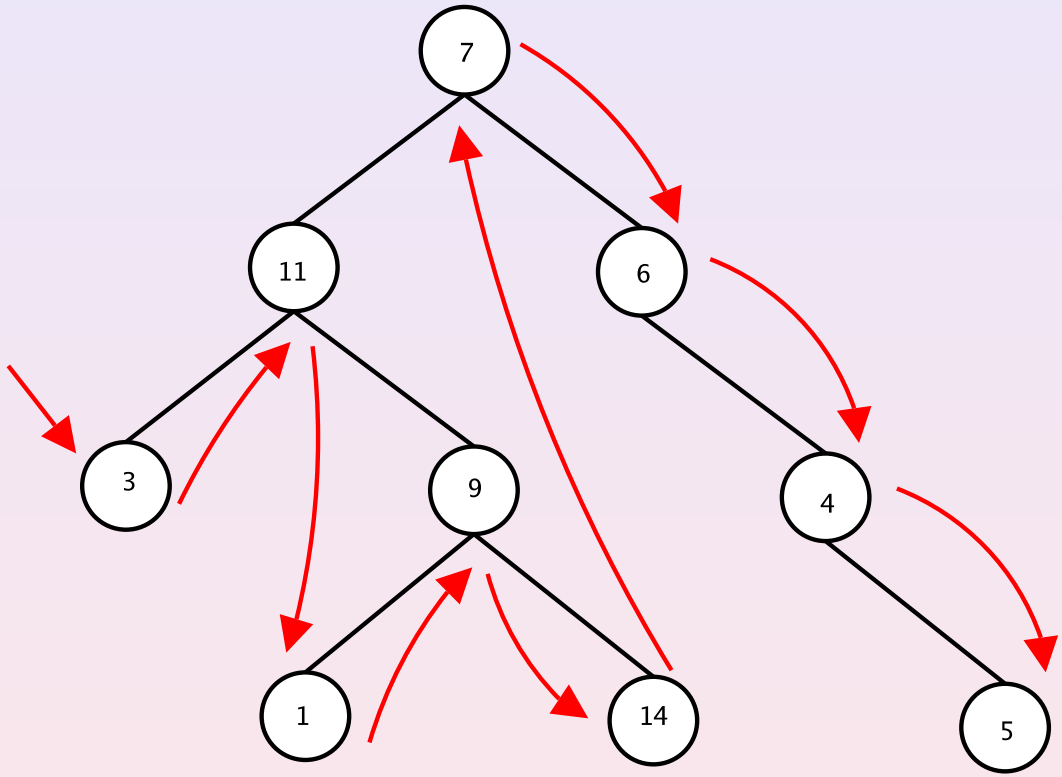


Ordine in cui sono visitati i nodi

7 11 3 9 1 14 6 4 5

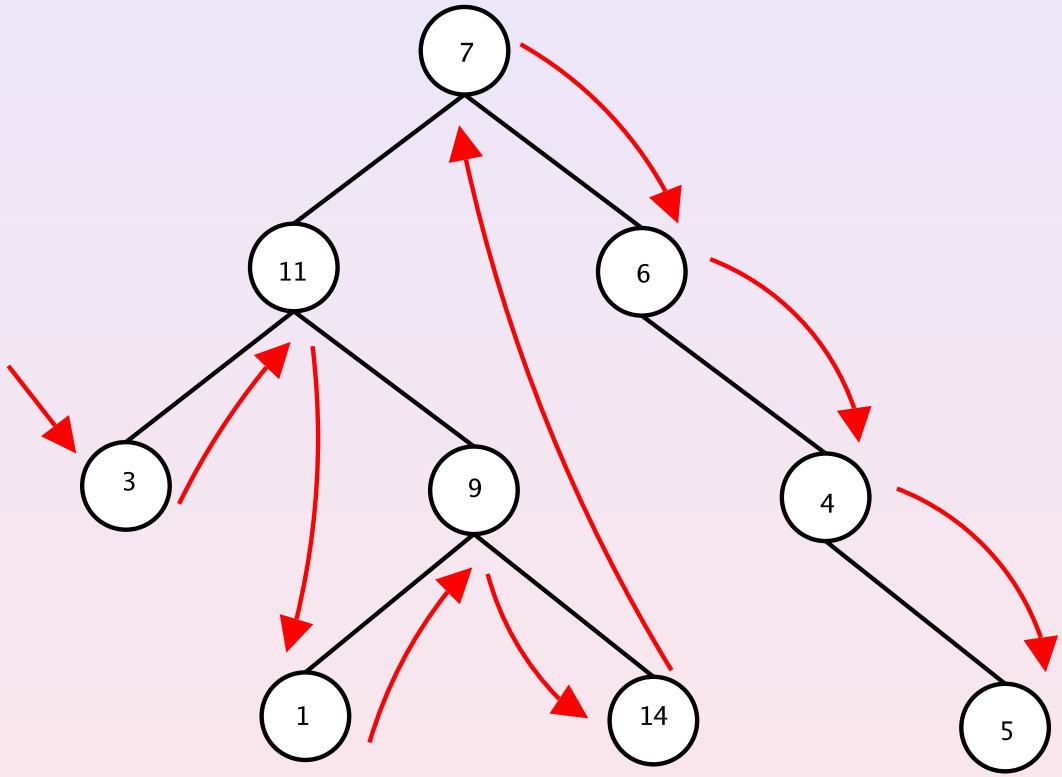
Visita in ordine simmetrico (inordine)

sottoalbero sinistro $\Rightarrow$ radice $\Rightarrow$ sottoalbero destro



Visita in ordine simmetrico (inordine)

sottoalbero sinistro $\Rightarrow$ radice $\Rightarrow$ sottoalbero destro

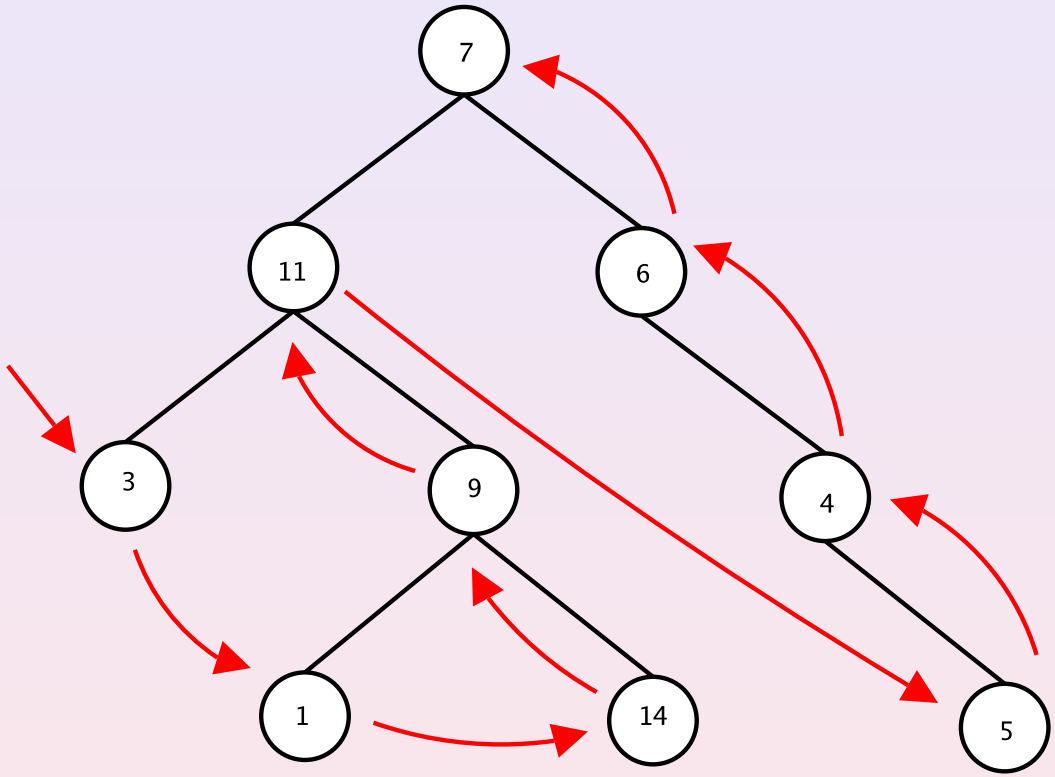


Ordine in cui sono visitati i nodi

3 11 1 9 14 7 6 4 5

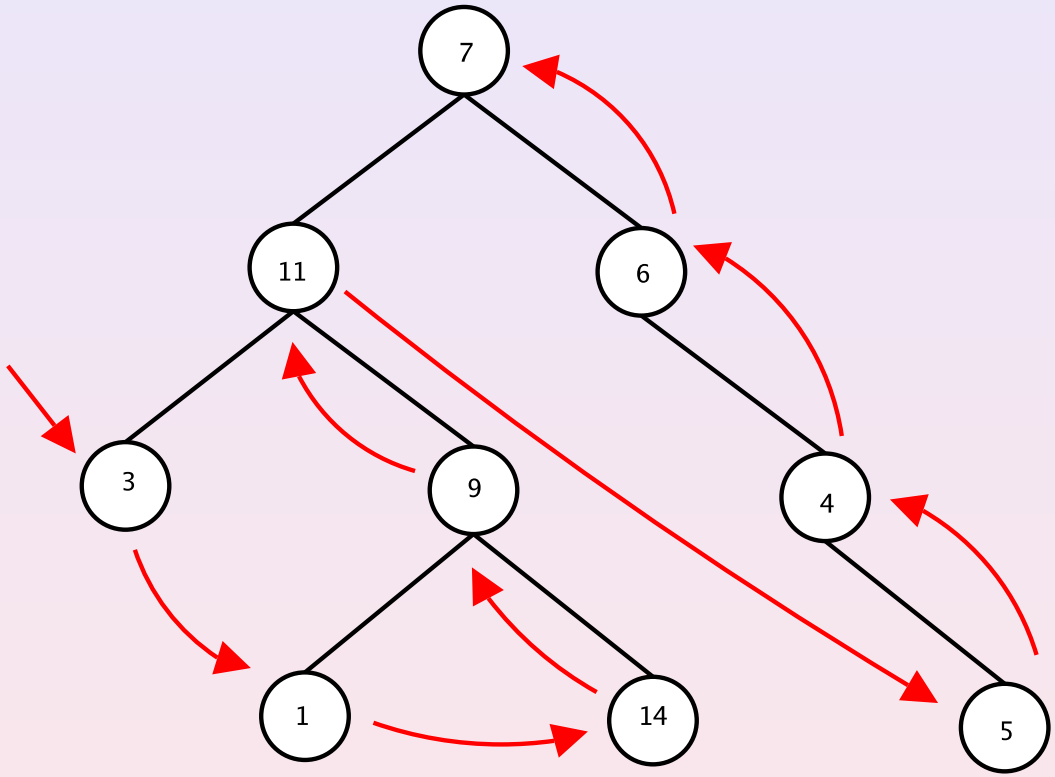
Visita in ordine posticipato (postordine)

sottoalbero sinistro $\Rightarrow$ sottoalbero destro $\Rightarrow$ radice



Visita in ordine posticipato (postordine)

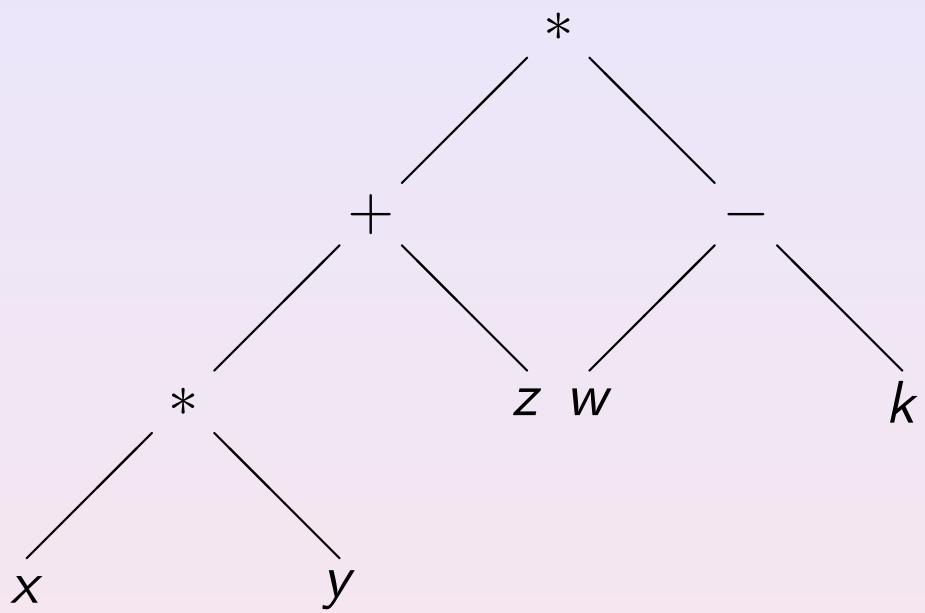
sottoalbero sinistro $\Rightarrow$ sottoalbero destro $\Rightarrow$ radice



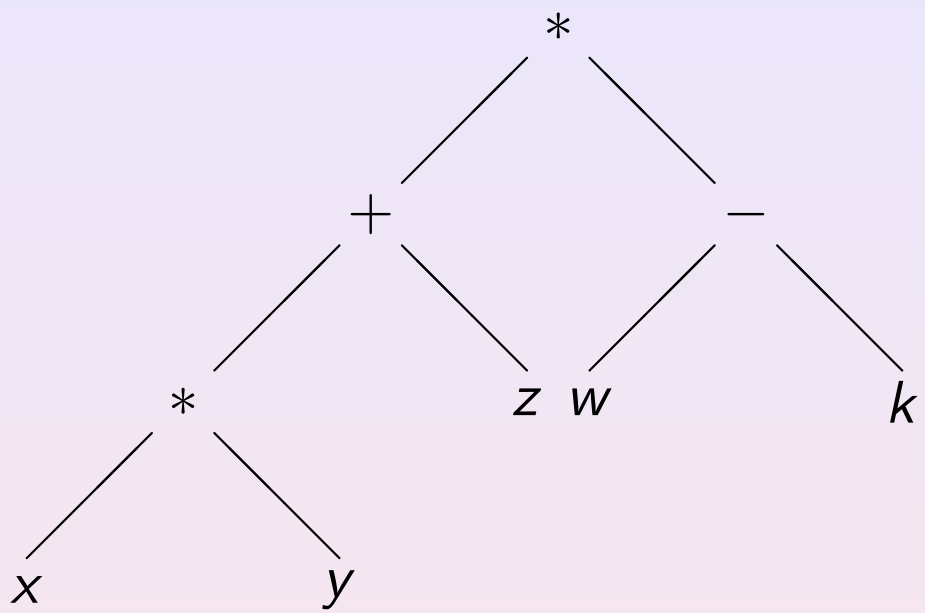
Ordine in cui sono visitati i nodi

3 1 14 9 11 5 4 6 7

Visita

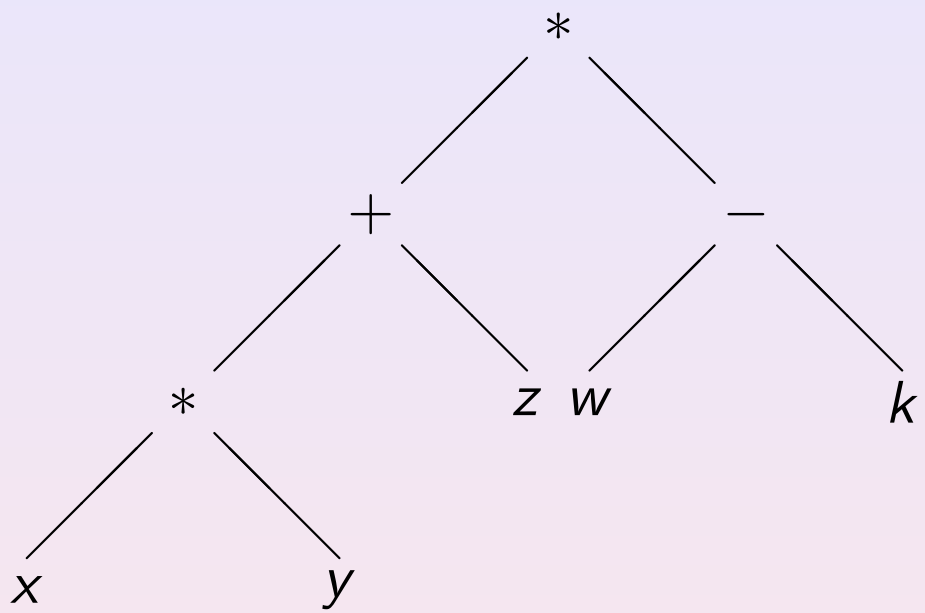


Visita



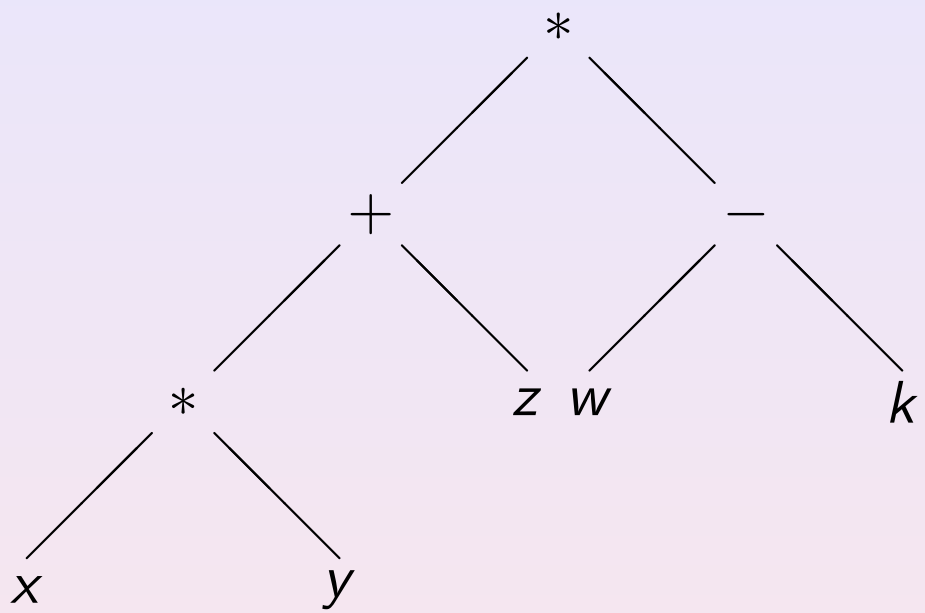
- Ordine anticipato: $* \; + \; * \; x \; y \; z \; - \; w \; k$

Visita



- Ordine anticipato: $* \; + \; * \; x \; y \; z \; - \; w \; k$
- Ordine simmetrico: $x \; * \; y \; + \; z \; * \; w \; - \; k$

Visita



- Ordine anticipato: $* \; + \; * \; x \; y \; z \; - \; w \; k$
- Ordine simmetrico: $x \; * \; y \; + \; z \; * \; w \; - \; k$
- Ordine posticipato: $x \; y \; * \; z \; + \; w \; k \; - \; *$

Ricerca in un albero

Alberi di ricerca

Usati per rappresentare insiemi di dati ordinati in base a una chiave.

Albero di ricerca

Un albero binario in cui per ogni nodo x

- tutti i valori contenuti nel sottoalbero sinistro di x sono minori del valore contenuto in x

Alberi di ricerca

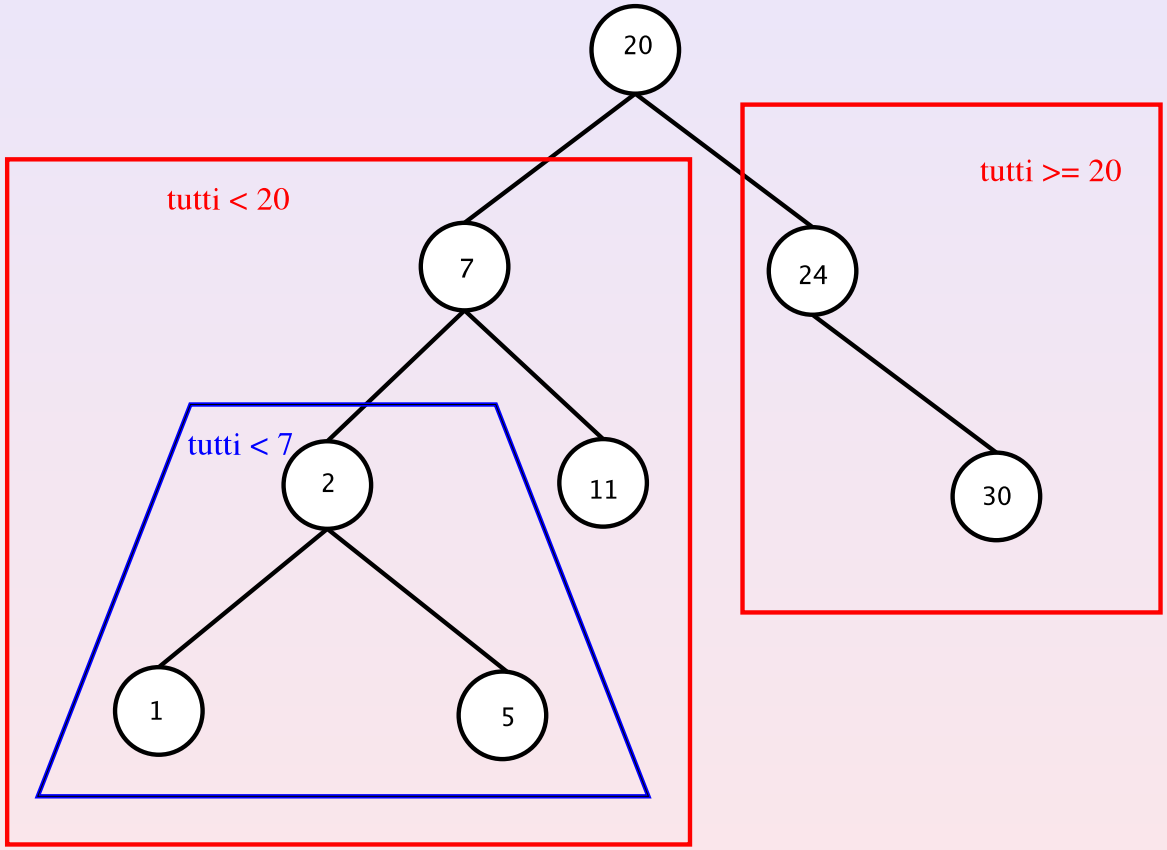
Usati per rappresentare insiemi di dati ordinati in base a una chiave.

Albero di ricerca

Un albero binario in cui per ogni nodo x

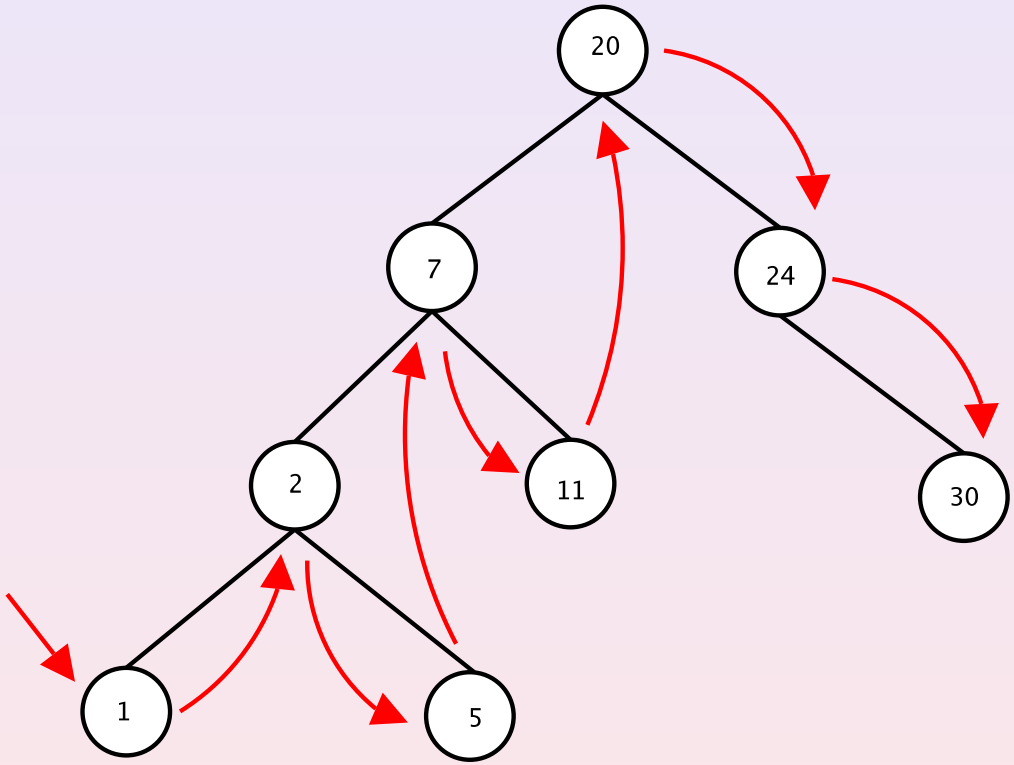
- tutti i valori contenuti nel sottoalbero sinistro di x sono minori del valore contenuto in x
- tutti i valori contenuti nel sottoalbero destro di x sono maggiori (o uguali) del valore contenuto in x

Esempio



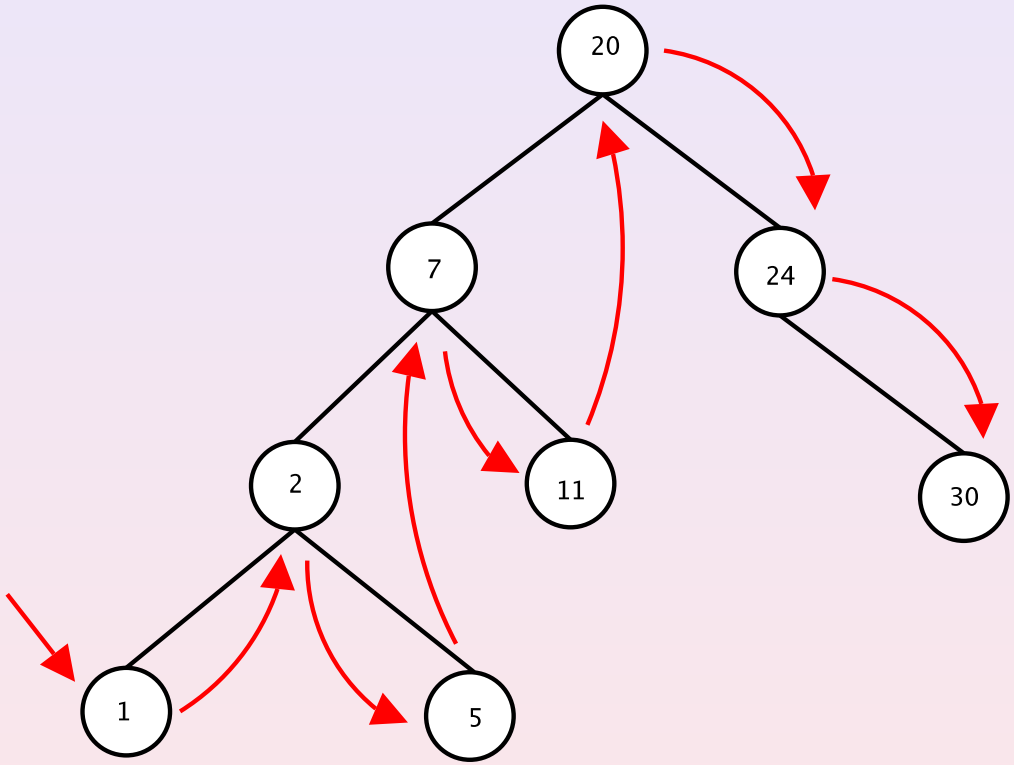
Visita in ordine simmetrico

sottoalbero sinistro $\Rightarrow$ radice $\Rightarrow$ sottoalbero destro



Visita in ordine simmetrico

sottoalbero sinistro $\Rightarrow$ radice $\Rightarrow$ sottoalbero destro

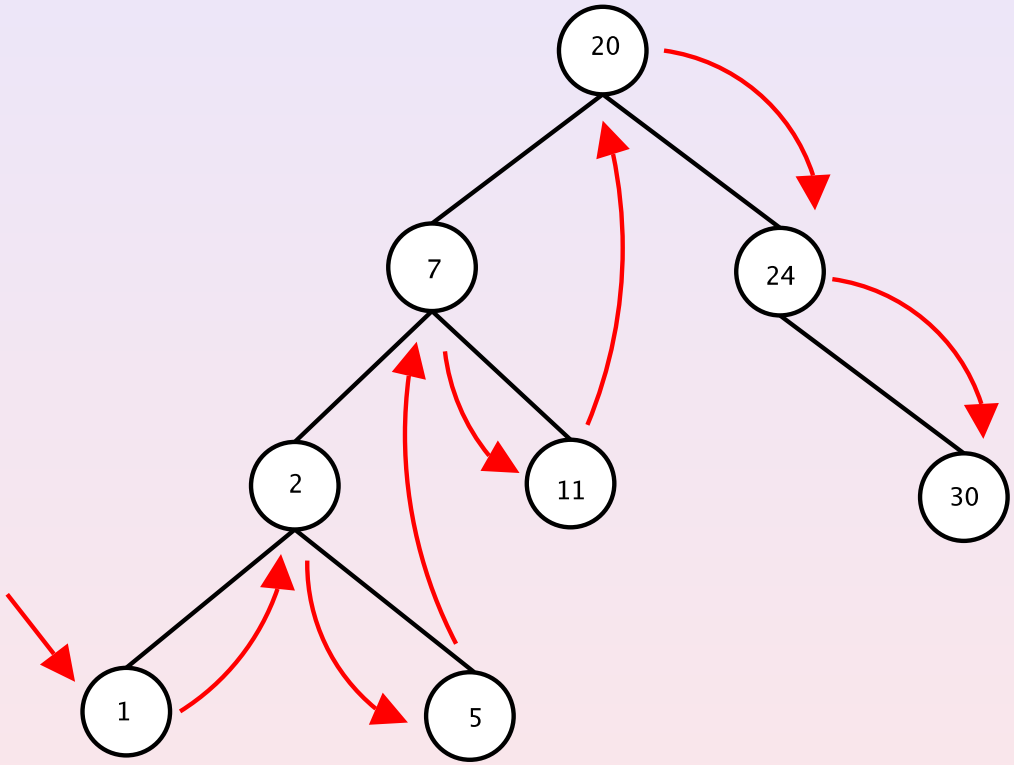


Ordine in cui sono visitati i nodi

1 2 5 7 11 20 24 30

Visita in ordine simmetrico

sottoalbero sinistro $\Rightarrow$ radice $\Rightarrow$ sottoalbero destro



Ordine in cui sono visitati i nodi

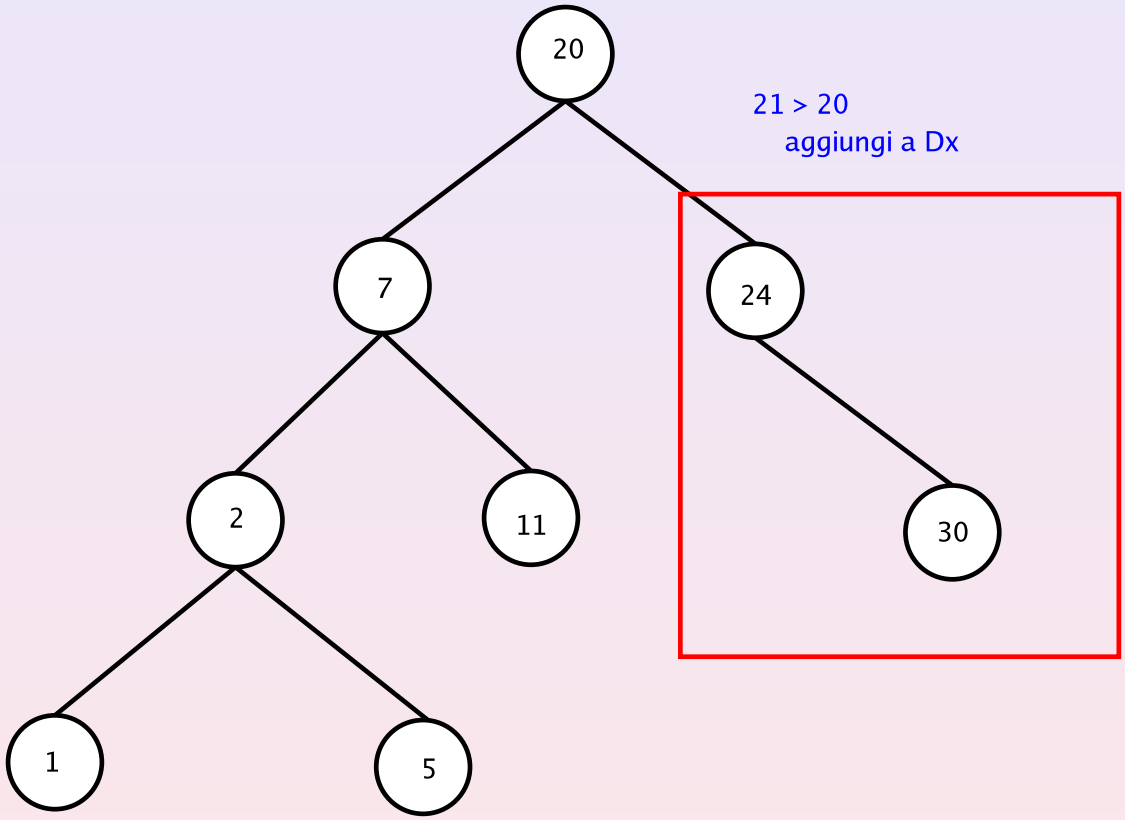
1 2 5 7 11 20 24 30 La sequenza è ordinata

Costruzione di un albero di ricerca

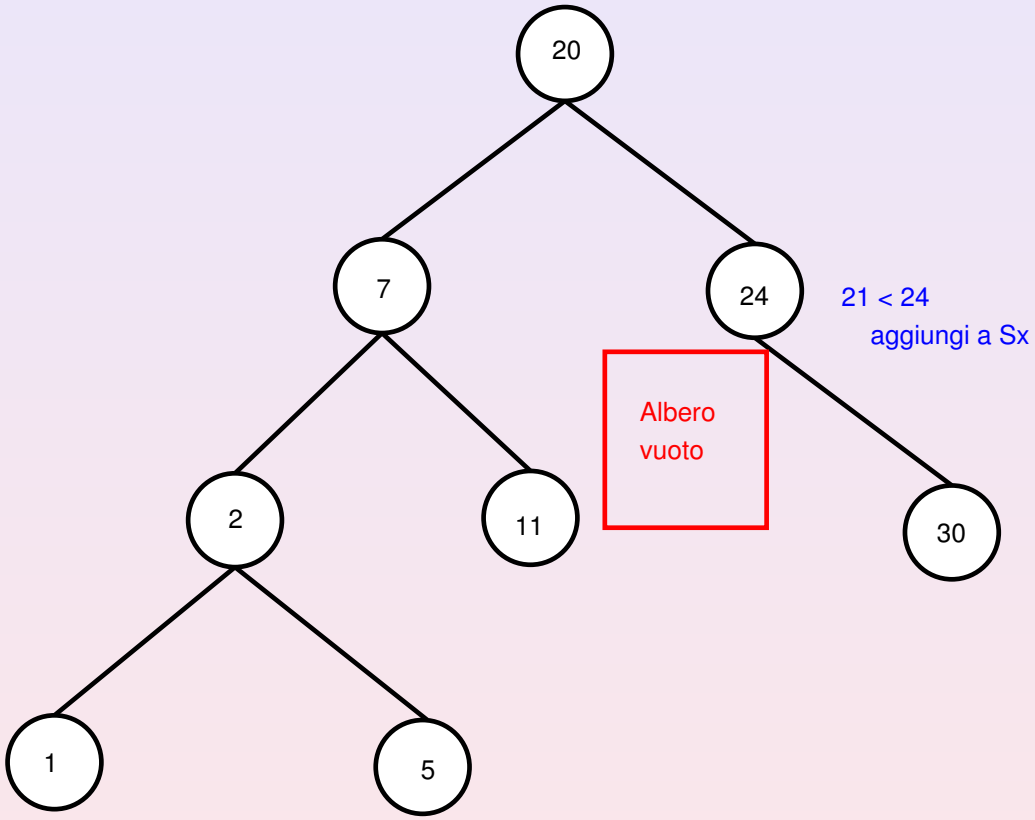
- Un albero vuoto è un albero di ricerca
- Aggiungere un elemento (chiave `x`) ad un albero di ricerca:

```
se l'albero è vuoto
    inserisci l'elemento
altrimenti
    se x < chiave della radice
      aggiungi l'elemento al sottoalbero di sinistra
    altrimenti
      aggiungi l'elemento al sottoalbero di destra
```

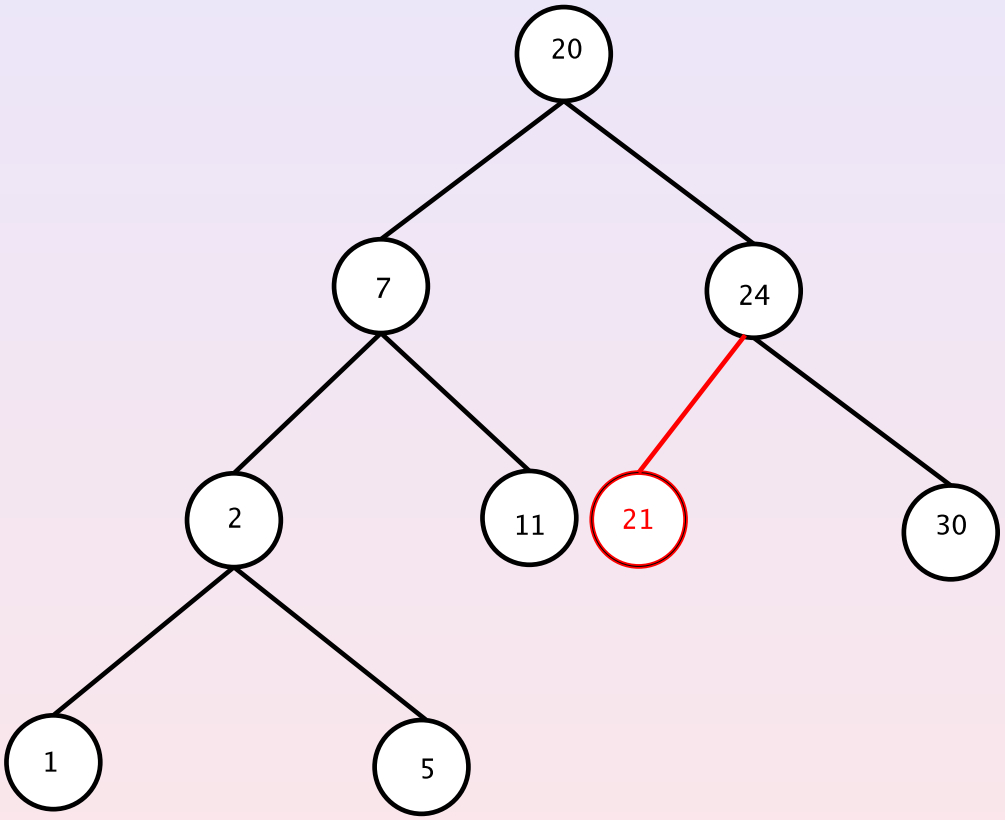
Inserimento di 21



Inserimento di 21



Inserimento di 21



Implementazione degli alberi di ricerca - AG

```
public class Albero<E extends Comparable<? super E>> {

  private NodoAlbero root;

  private class NodoAlbero {
    E dato;
    NodoAlbero sx, dx;
  }

  //COSTRUTTORE
  public Albero() {
    root = null;
  }

  //METODI
  ...
}
```

Metodi helper - AG

Spesso useremo per implementare una funzionalità, due metodi:

- Un metodo che prende un NodoAlbero come parametro, detto metodo helper, che contiene l'implementazione reale della funzionalità.
- Questo metodo è spesso ricorsivo e si può mettere privato.
- Un metodo che chiama il metodo helper con il nodo root dell'albero e che "espone" la funzionalità

Visualizzazione - AG

Visualizzazione inorder dell'albero

```
/** questo è un metodo "helper" -> visualizzazione del nodo
*   a: nodo da cui far partire la visualizzazione */
private void visualizza(NodoAlbero a){
  if (a != null) {
    visualizza(a.sx);
    System.out.println(a.dato);
    visualizza(a.dx);
  }
}
/** visualizza tutto l'albero */
public void visualizza() {
  visualizza(root);
}
```

Visualizzazione - AG

Cambiando l'ordine in cui si visita sx, dx e dato si ottengono i diversi tipi di vista

Visualizzazione - AG

Cambiando l'ordine in cui si visita sx, dx e dato si ottengono i diversi tipi di vista

- **preorder**: prima il nodo e poi i sottoalberi

```
System.out.println(a.dato);
visualizza(a.sx);
visualizza(a.dx);
```

Visualizzazione - AG

Cambiando l'ordine in cui si visita sx, dx e dato si ottengono i diversi tipi di vista

- **preorder**: prima il nodo e poi i sottoalberi

```
System.out.println(a.dato);
visualizza(a.sx);
visualizza(a.dx);
```

- **postorder**: il nodo per ultimo:

```
visualizza(a.sx);
visualizza(a.dx);
System.out.println(a.dato);
```

Trova - AG

Ricerca di un elemento nell'albero

Trova - AG

Ricerca di un elemento nell'albero Ancora nella classe Albero con metodo helper

```
/** cerca x a partire dal nodo a
 *  restuisce null se x non \‘e in a */
public E trova(NodoAlbero a, E x) {
  if (a == null)   return null;
  else if (x.compareTo(a.dato) < 0)
    return trova(a.sx,x);
  else if (x.compareTo(a.dato) > 0)
    return trova(a.dx,x);
  else
    return a.dato;
}
/** cerca x nell'albero */
public E trova(E x) {
  return trova(root,x);
}
```

Visualizzazione (2) - AG

Definizione di un metodo toString - stavolta il metodo helper è messo nella classe NodoAlbero. In questo caso bisogna controllare che i figli non siano null.

```
private class NodoAlbero {
...
   /** la stringa dell'albero a partire da questo nodo */
   public String toString() {
      String sinistra = sx == null ? "" : sx.toString();
      String centro = dato.toString();
      String destra = dx == null ? "" : dx.toString();
      return sinistra + centro + destra;
   }
  ...
  // toString di Albero
  public String toString(){
     if (root != null) return root.toString();
     else      return "";
  }
```

Visualizzazione (3) - AG

Metodo helper è messo nella classe NodoAlbero e separatore *sep*

```
  private class NodoAlbero {
...
       /** la stringa dell'albero a partire da questo nodo */
       public String toString(String separatore) {
    String sinistra = sx == null ? "" : sx.toString(sep);
            String centro = dato.toString() + sep;
            String destra = dx == null ? "" : dx.toString(sep);
            return sinistra + centro + destra;
       }
  ...
  // toString di Albero
  public String toString(String sep){
     if (root != null)
        return root.toString(sep);
     else
        return "";
  }
```

Inserimento - AG

Nella classe Albero con helper

```
// inserisce x a partire da y come root
private void insert(NodoAlbero y, E x) {
// se x è minore di y
if (x.compareTo(y.dato) < 0) {
// inserisci a sinistra
// se a sinistra c'è posto, lo metto lì
if (y.sx == null) {
y.sx = new NodoAlbero();
y.sx.dato = x;
} else
insert(y.sx, x);
} else {
// inserisci a destra
if (y.dx == null) {
y.dx = new NodoAlbero();
y.dx.dato = x;
} else
```

Inserimento - AG

Nella classe NodoAlbero

```
/** insert x starting from this node */
public void insert(E x){
//in left subtree if x < dato
if (x.compareTo(dato) < 0)){
  // insert new NodoAlbero
  if ( sx == null )
    sx = new NodoAlbero( x );
  else // continue traversing left subtree
    sx.insert( x );
} else if ( x.compareTo(dato) > 0){ // insert in right subtree
  // insert new NodoAlbero
  if ( dx == null )
    dx = new NodoAlbero( x );
  else // continue traversing right subtree
    dx.insert( x );
} // end else if
} // end method insert
```

Inserimento - AG

Nella classe Albero, invece c'è l'inserimento diretto

```
// insert a new node in the binary search tree
public void insert( E insertValue ){
  if ( root == null )
    root = new NodoAlbero( insertValue ); // create the root node he
  else
    root.insert( insertValue ); // call the insert method
} // end method insertNode
```

Implementazione degli alberi di ricerca - LIBRO

```
public class Albero<E extends Comparable<? super E>> {

  private NodoAlbero a;

  private class NodoAlbero {
    E dato;
    Albero<E> sx, dx;
  }

  //COSTRUTTORE
  public Albero() {
    a = null;
  }

  //METODI
  ...
}
```

Visualizzazione - LIBRO

```
public void visualizza() {
  if (a != null) {
    a.sx.visualizza();
    System.out.println(a.dato);
    a.dx.visualizza();
  }
}

public String toString(String separatore) {
  if (a != null) {
    String sinistra = a.sx.toString(separatore);
    String centro = a.dato.toString() + separatore;
    String destra = a.dx.toString(separatore);
    return sinistra + centro + destra;
  } else
    return "";
}
```

Inserimento: schema - LIBRO

```
public void inserisci(E x) {
  if (a == null) {
    inserisci qui un nuovo nodo contenente x,
    con sottoalberi sinistro e destro vuoti
  } else if (x e' minore di a.dato)
    inserisci x nel sottoalbero sinistro
  else
    inserisci x nel sottoalbero destro
}
```

Inserimento - LIBRO

```
public void inserisci(E x) {
  if (a == null) {
    a = new NodoAlbero();
    a.dato = x;
    a.sx = new Albero<E>();
    a.dx = new Albero<E>();
  } else if (x.compareTo(a.dato) < 0)
      a.sx.inserisci(x);
  else
    a.dx.inserisci(x);
}
```

Trova - LIBRO

```
public E trova(E x) {
  if (a == null)
    return null;
  else if (x.compareTo(a.dato) < 0)
    return a.sx.trova(x);
  else if (x.compareTo(a.dato) > 0)
    return a.dx.trova(x);
  else
    return a.dato;
}
```

Trova e inserisci - LIBRO

```
public E trovaEInserisci(E x) {
  if (a == null) {
    a = new NodoAlbero();
    a.dato = x;
    a.sx = new Albero<E>();
    a.dx = new Albero<E>();
    return null;
  } else if (x.compareTo(a.dato) < 0)
    return a.sx.trovaEInserisci(x);
  else if (x.compareTo(a.dato) > 0)
    return a.dx.trovaEInserisci(x);
  else
    return a.dato;
}
```