Algoritmi e Strutture Dati

Lezione 3

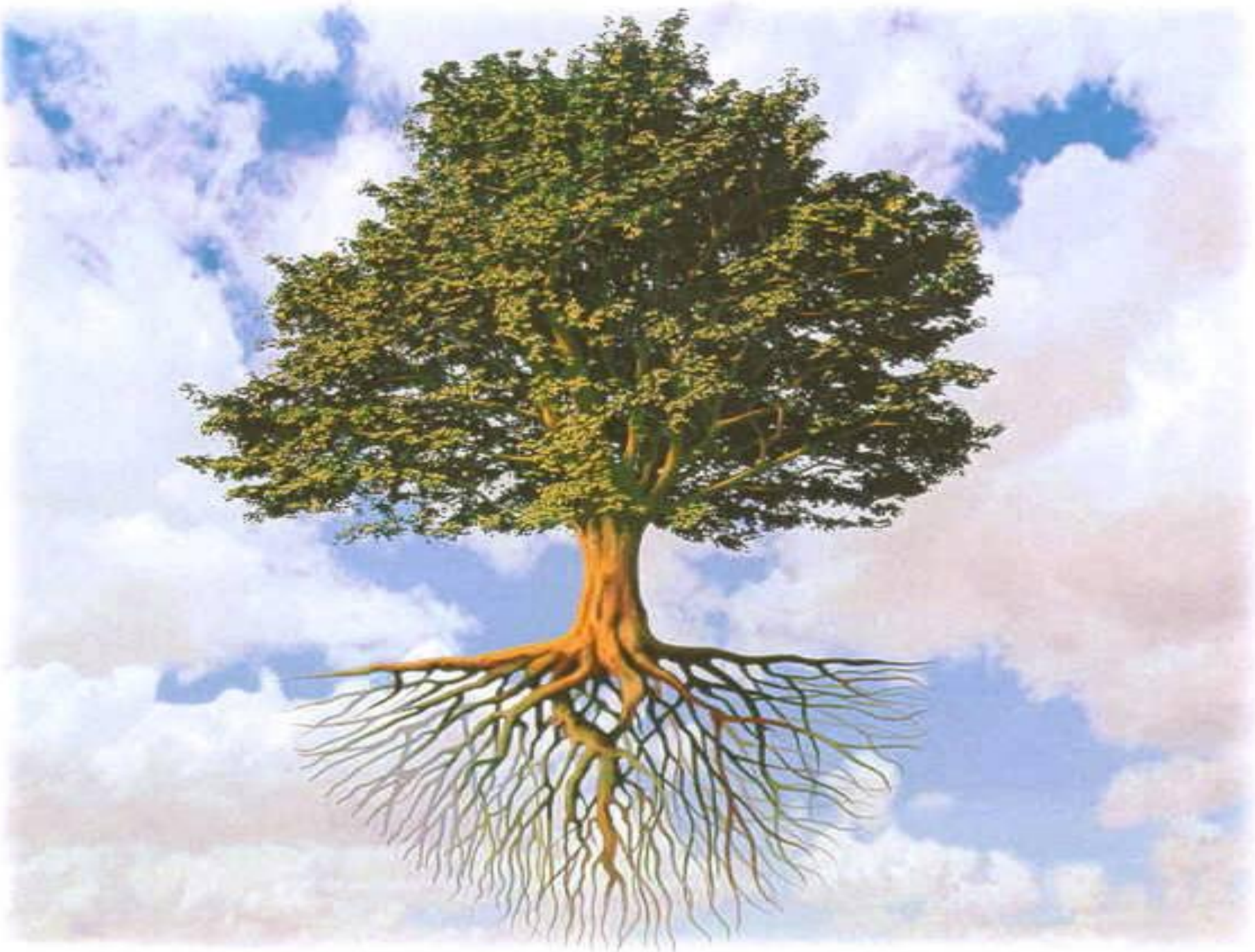
http://mlpi.ing.unipi.it/alfeo

Antonio Luca alfeo

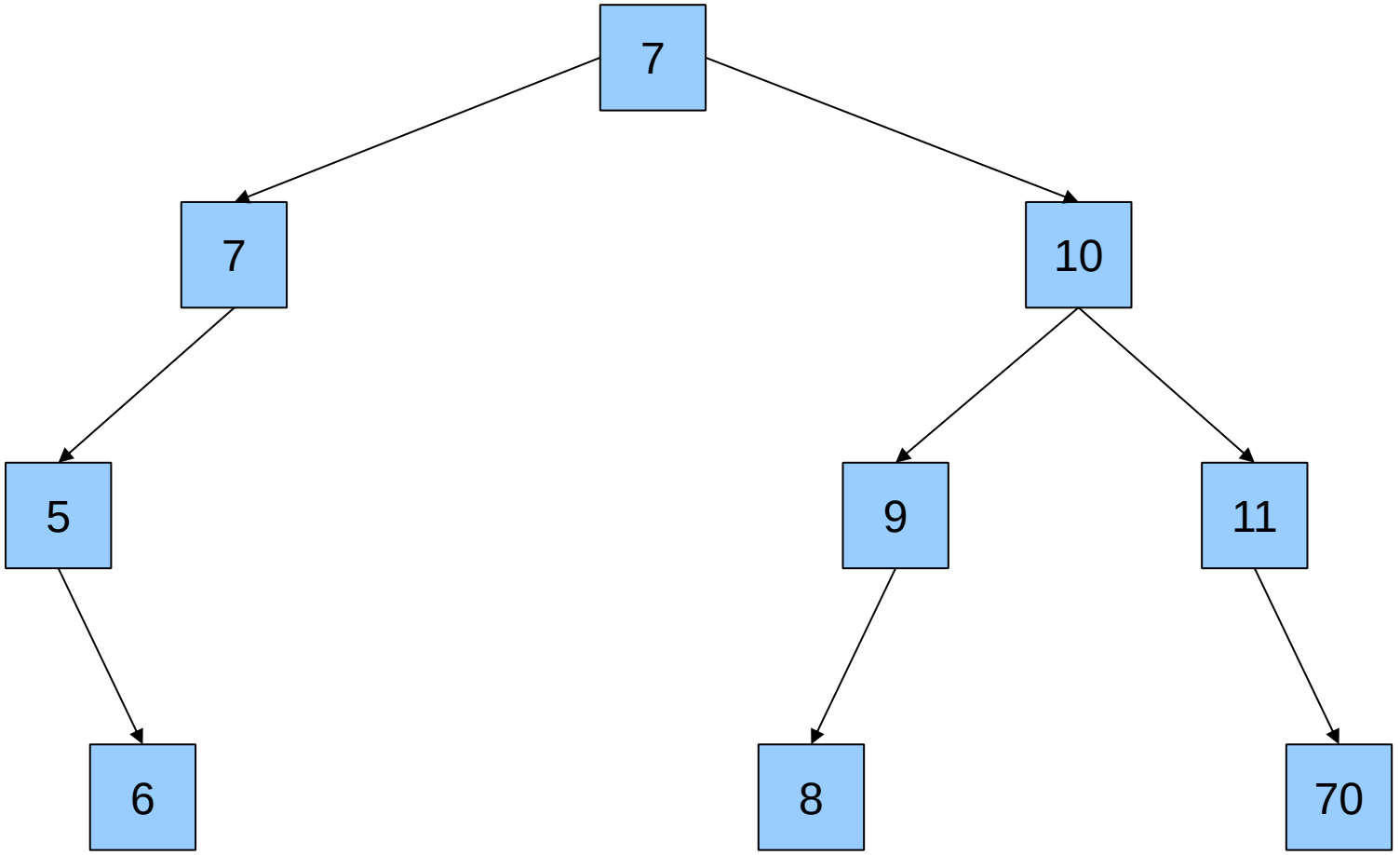
luca.alfeo@ing.unipi.it

Sommario

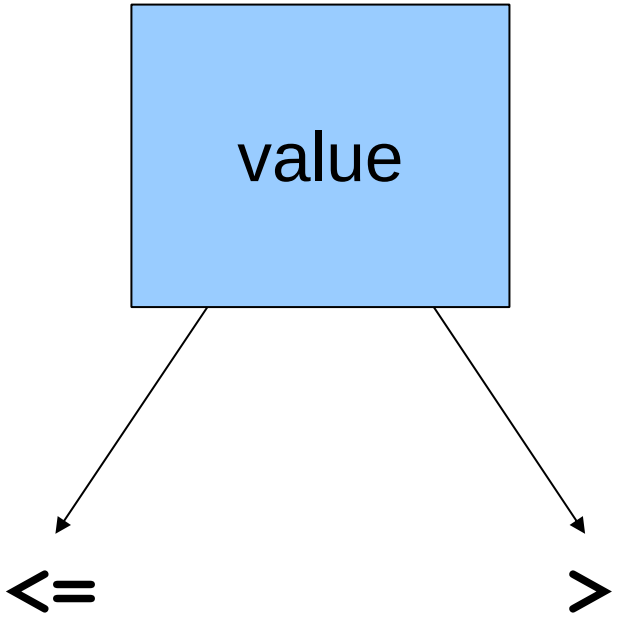
- Alberi Binari di Ricerca
- Progettazione e Inizializzazione
- Esempi Funzioni con Alberi Binari di Ricerca
- Alberi binari con etichette complesse
- Esercizi



Alberi Binari



Alberi Binari di Ricerca



binTree

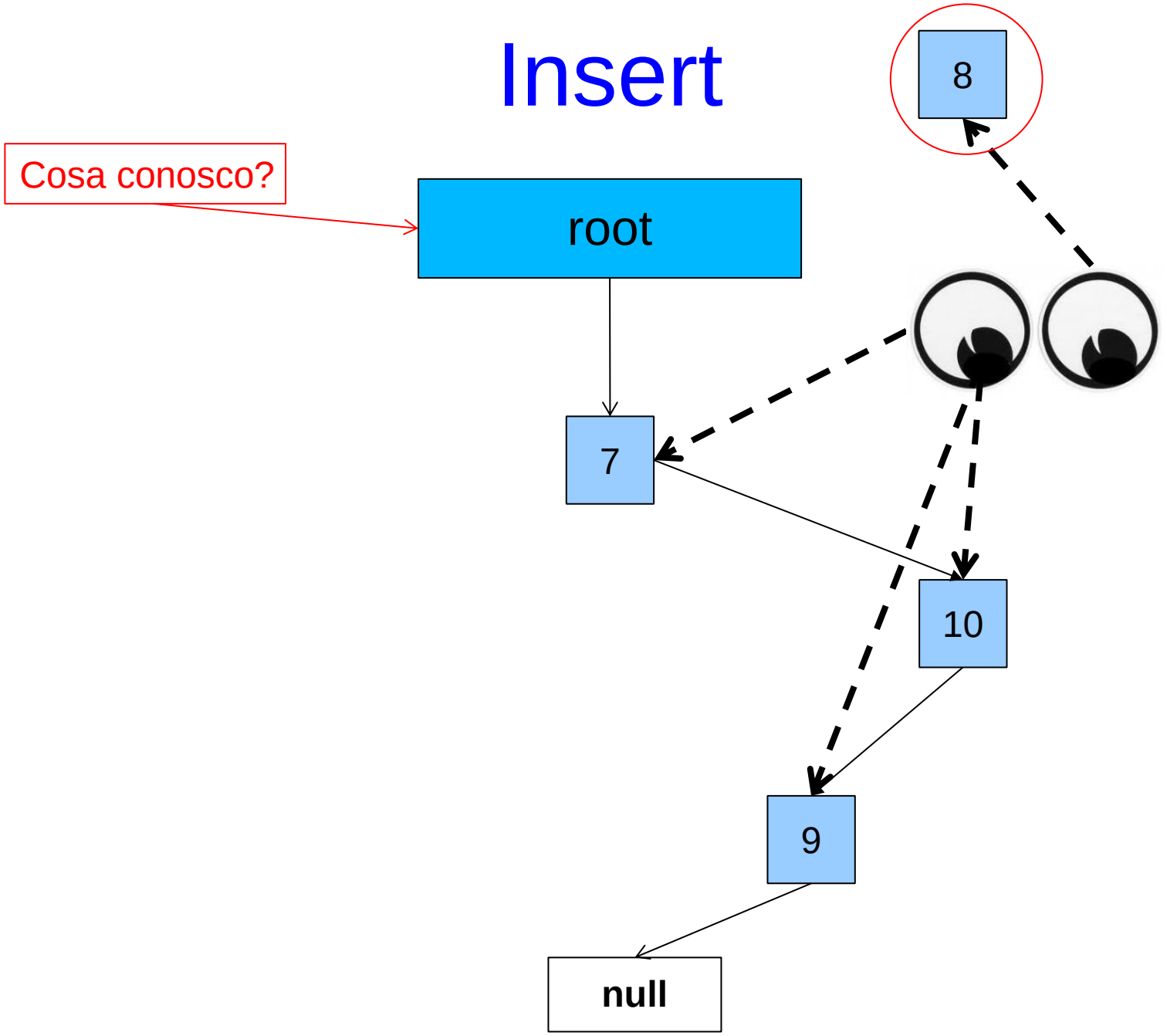
```
1   struct Node
2   {
3       int value;
4       Node * left;
5       Node * right;
6
7       Node(int val):
8           value(val) , left(NULL) , right(NULL) {}
9   };
10
11
12
13
14
15
16
17
18
19
20
```

binTree

```
1   struct Node
2   {
3       int value;
4       Node * left;
5       Node * right;
6
7       Node(int val):
8           value(val) , left(NULL) , right(NULL) {}
9   };
10
11  class BinTree
12  {
13      Node * root_;
14
15  public:
16
17      BinTree() { root_ = NULL; }
18
19      Node * getRoot() { return root_; }
20  }
```

N.B. = inizializzare i puntatori a NULL

Insert



Insert

```
1   void insert( int val )
2   {
3       // inizializzo nuovo elemento
4       // inizializzo variabili appoggio
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21  }
```

Insert

```
1   void insert( int val )
2   {
3       // inizializzo nuovo elemento
4       // inizializzo variabili appoggio
5
6
7       // finchè non arrivo ad una foglia
8       {
9           // aggiorno variabili
10          // se <=
11              // vado a sinistra
12          // altrimenti
13              // vado a destra
14      }
15
16
17
18
19
20
21  }
```

Insert

```
1   void insert( int val )
2   {
3       // inizializzo nuovo elemento
4       // inizializzo variabili appoggio
5
6
7       // finchè non arrivo una foglia
8       {
9           // aggiorno variabili
10          // se <=
11              // vado a sinistra
12          // altrimenti
13              // vado a destra
14      }
15      // se albero vuoto
16          // aggiorno radice
17
18      // decido se diventare figlio left o right
19
20
21  }
```

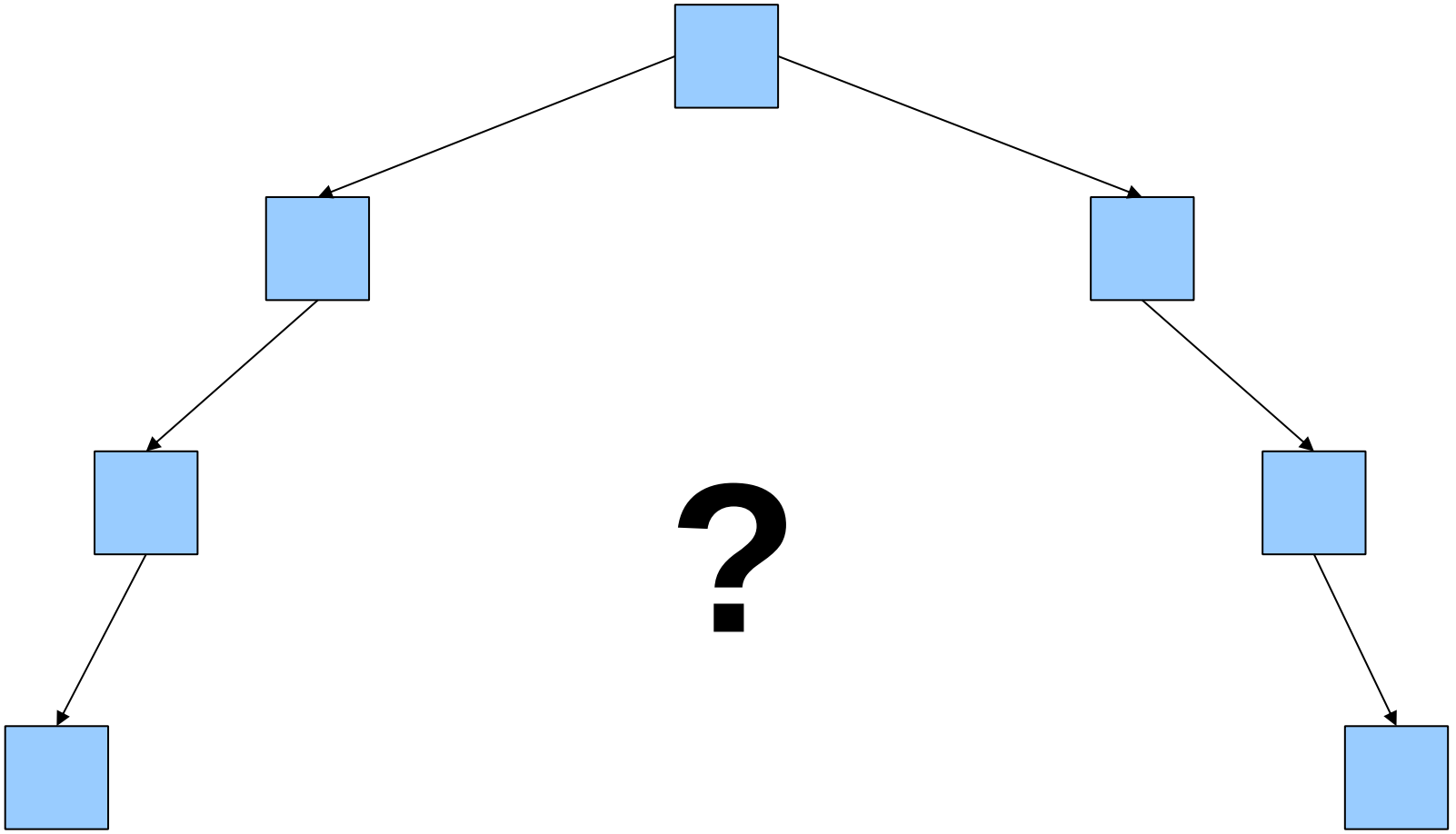
INIZIALIZZAZIONE

INDIVIDUO POSIZIONE

INSERIMENTO

Esempio: trova Min e Max

min & MAX



Min & Max

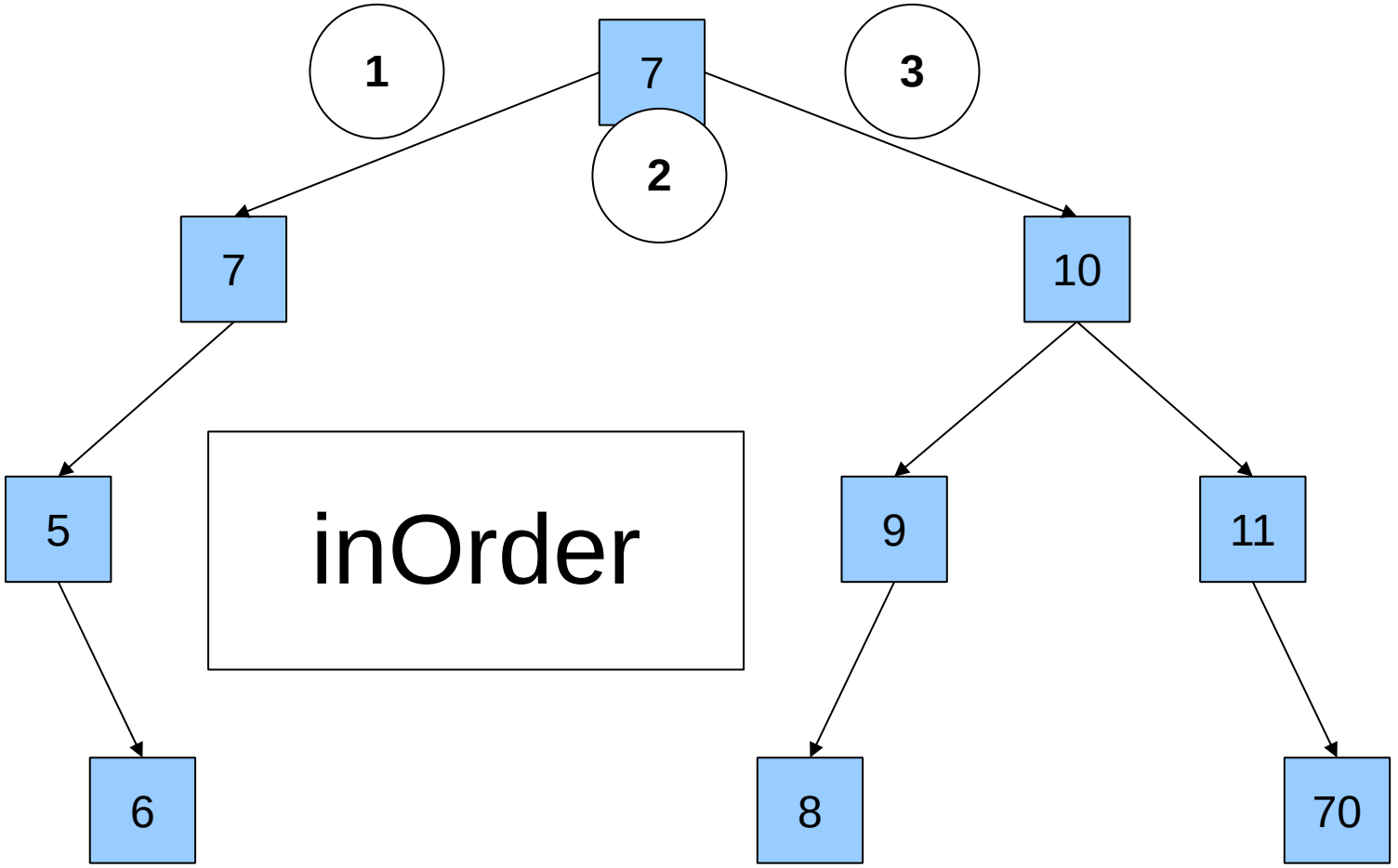
```
1   Node * min()
2   {
3       Node * temp = root_;
4       while( temp->left != NULL )
5           temp = temp->left;
6       return temp;
7   }
8
9
10
11
12
13
14
15
16
17
18
19
20
```

Min & Max

```
1   Node * min()
2   {
3       Node * temp = root_;
4       while( temp->left != NULL )
5           temp = temp->left;
6       return temp;
7   }
8
9
10
11  Node * max()
12  {
13      Node * temp = root_;
14      while( temp->right != NULL )
15          temp = temp->right;
16      return temp;
17  }
18
19
20
```

Esempio: visita l'albero

Visita l'albero



In-Order

```
1
2
3
4   void inOrder( Node * tree )
5   {
6       // se l'albero non e' terminato
7        {
8
9
10
11
12
13
14
15       }
16  }
17
18
19
20
```

In-Order

```
1
2
3
4   void inOrder( Node * tree )
5   {
6       // se l'albero non e' terminato
7       {
8
9           // visito verso left
10
11          // stampo questo valore
12
13          // visito verso right
14
15      }
16  }
17
18
19
20
```

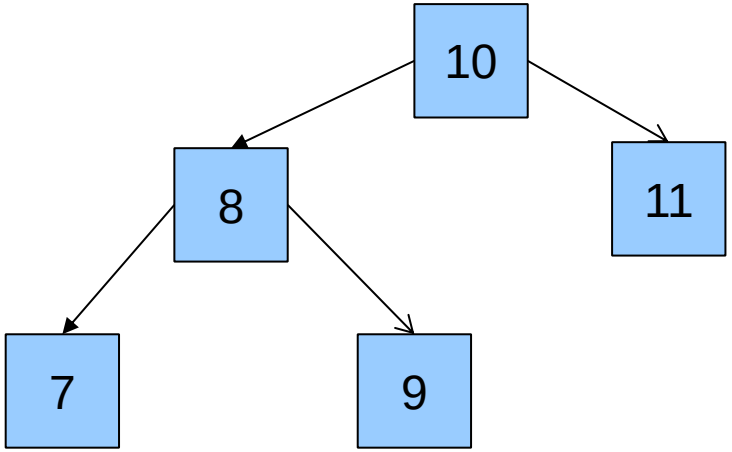
In-Order

```
1
2
3
4   void inOrder( Node * tree )
5   {
6       if(tree!=NULL)
7       {
8
9           inOrder(tree->left);
10
11          cout << tree->value << "\t";
12
13          inOrder(tree->right);
14
15      }
16  }
17
18
19
20
```

Sort vs BinTree

AVG CASE

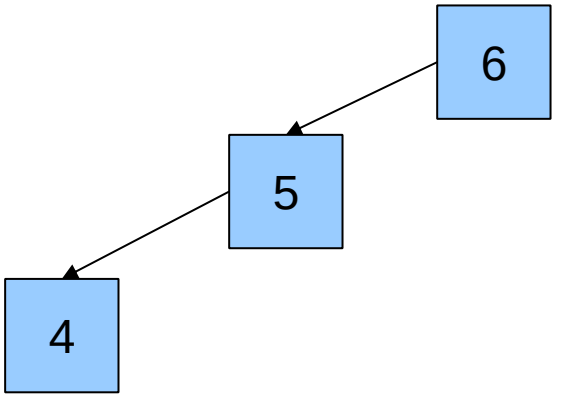
- Albero alto: **log(n)**
- Inserimento: **n * log(n)**
- Sort/visita: **n**



$$\Theta(n \cdot log\, n)$$

WORST CASE

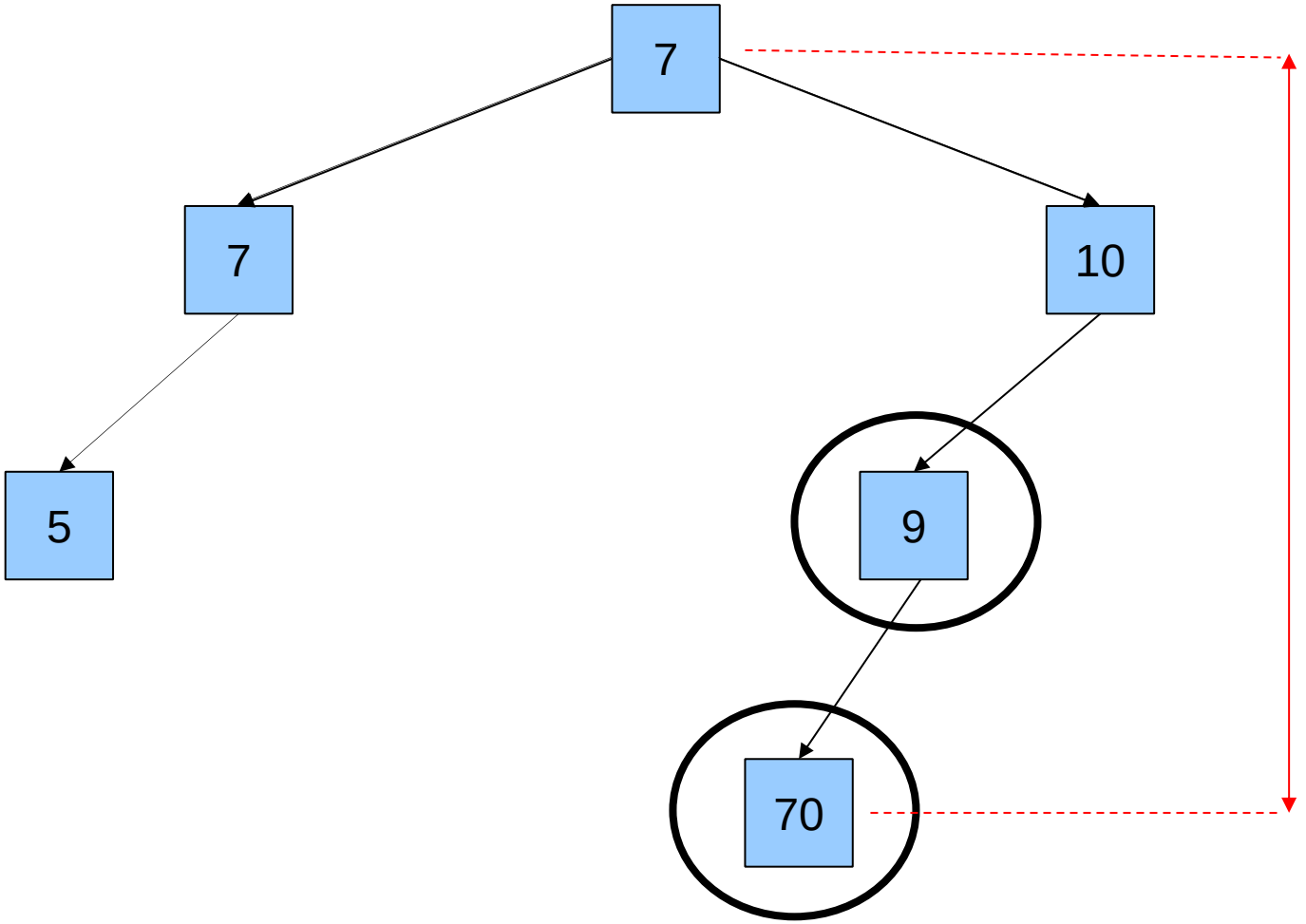
- Albero alto: **n**
- Inserimento: **n * n**
- Sort/visita: **n**



$$\Theta(n^2)$$

Esempio: calcolare altezza albero

Altezza albero



Altezza albero

```
1   int height( Node * tree )
2   {
3       int hLeft;
4       int hRight;
5
6       if( tree == NULL )
7           return 0;
8
9
10
11
12
13
14
15
16
17  }
18
19
20
```

Altezza albero

```
1   int height( Node * tree )
2   {
3       int hLeft;
4       int hRight;
5
6       if( tree == NULL )
7           return 0;
8
9       hLeft = height(tree->left);
10
11      hRight = height(tree->right);
12
13
14  }
15
16
17
18
19
20
```

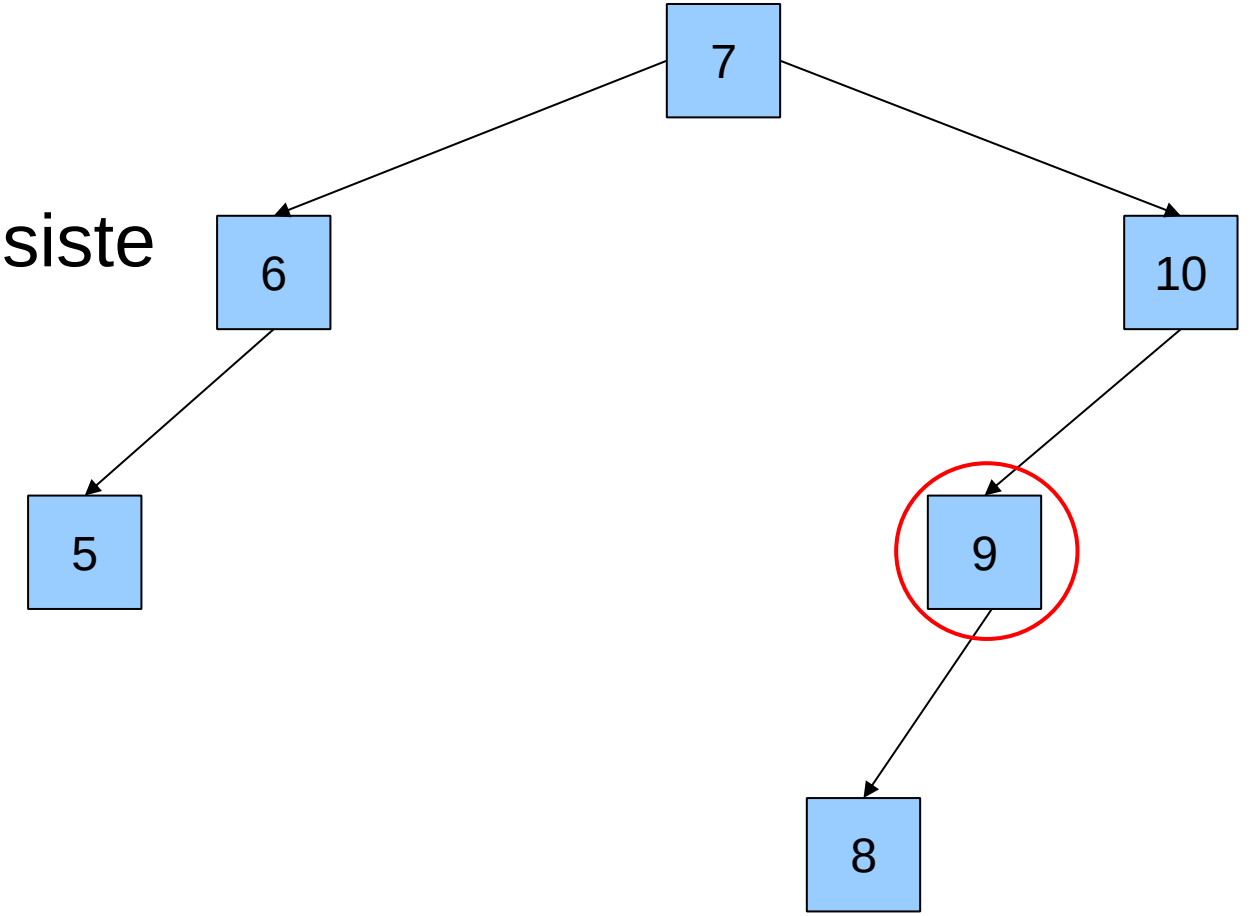
Altezza albero

```
1   int height( Node * tree )
2   {
3       int hLeft;
4       int hRight;
5
6       if( tree == NULL )
7           return 0;
8
9       hLeft = height(tree->left);
10
11      hRight = height(tree->right);
12
13      return 1 +  max(hLeft,hRight);
14  }
15
16
17
18
19
20
```

Esempio: trova un elemento

Trova elemento

- Dato
  - Un albero binario con valori distinti
  - Un valore K
- Trovare
  - Se il valore esiste



Search

```
1   bool search( Node * tree , int val )
2   {
3       if( tree == NULL )
4           return false;
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19  }
20
```

Search

```
1   bool search( Node * tree , int val )
2   {
3        if( tree == NULL )
4            return false;
5
6       bool found;
7
8        if( tree->value == val )
9            return true;
10
11
12
13
14
15
16
17
18
19  }
20
```

Search

```
1   bool search( Node * tree , int val )
2   {
3       if( tree == NULL )
4           return false;
5
6      bool found;
7
8       if( tree->value == val )
9           return true;
10
11      else if( val <= tree->value )
12         found = search( tree->left , val );
13      else
14         found = search( tree->right , val );
15
16     return found;
17  }
18
19
20
```

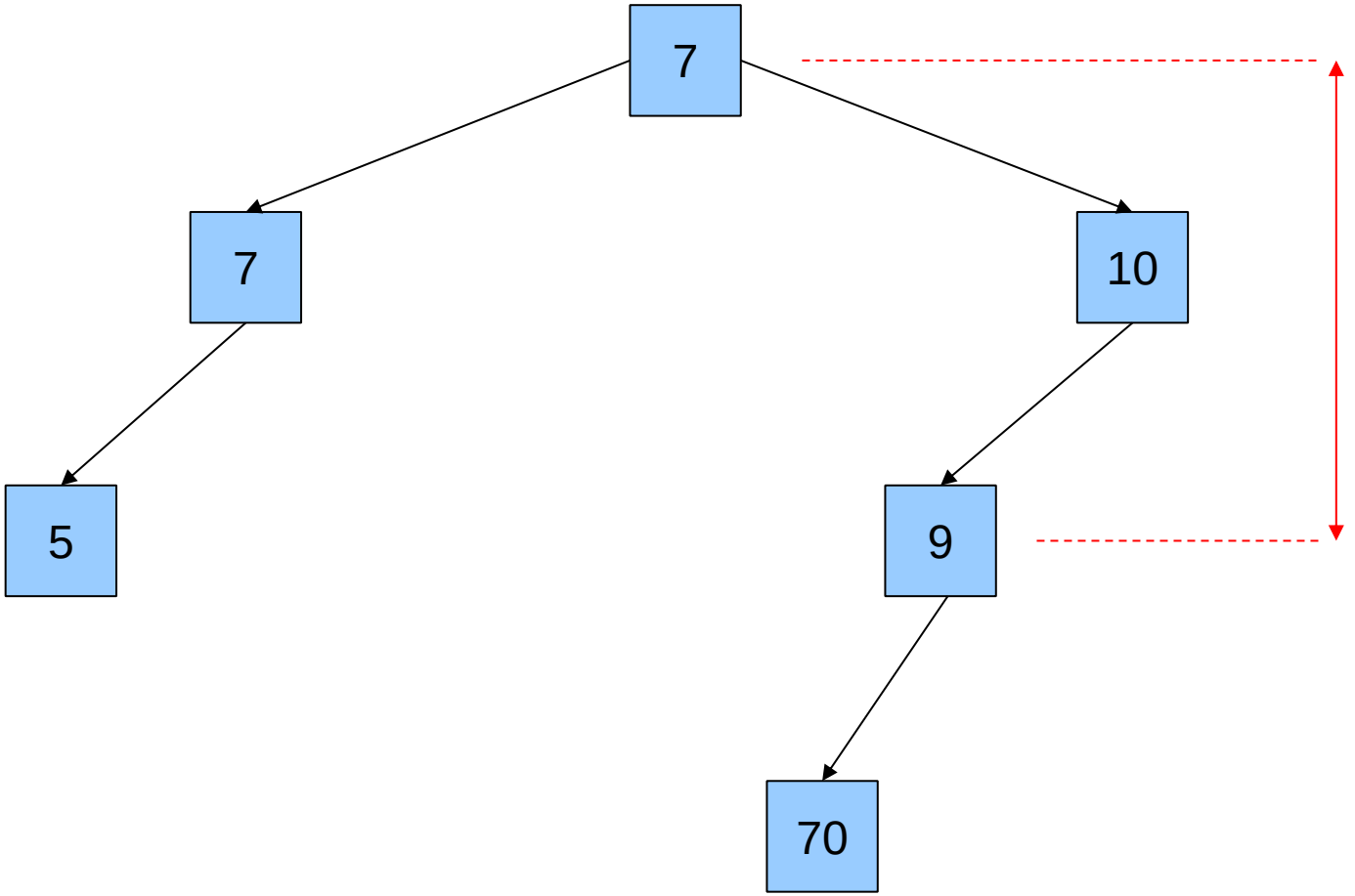
INDIVIDUO DIREZIONE

Esempio: altezza di un elemento

Trovare altezza chiave K

- Input
  - Una sequenza di N interi positivi
  - Chiave K
- Output
  - L’altezza della chiave K dentro l’albero (se esiste)

K=9



Altezza elemento

```
1   int search( Node * tree , int val )
2   {
3       if( tree == NULL )
4           return 0;
5
6       int cont = 0;
7       if( tree->value == val )
8           return 1;
9
10
11
12
13
14
15
16
17
18
19  }
20
```

Altezza elemento

```
1   int search( Node * tree , int val )
2   {
3       if( tree == NULL )
4           return 0;
5
6       int cont = 0;
7       if( tree->value == val )
8           return 1;
9
10      else if( val <= tree->value )
11          cont = search( tree->left , val );
12      else
13          cont = search( tree->right , val );
14
15
16
17
18
19  }
20
```

Altezza elemento

```
1   int search( Node * tree , int val )
2   {
3       if( tree == NULL )
4           return 0;
5
6       int cont = 0;
7       if( tree->value == val )
8           return 1;
9
10      else if( val <= tree->value )
11          cont = search( tree->left , val );
12      else
13          cont = search( tree->right , val );
14
15      if( cont != 0 )
16          return cont+1;
17
18      else return 0;
19  }
20
```

SEARCH

HEIGHT


IN SUPREMÆ DIGNITATIS
1343

ESERCIZI

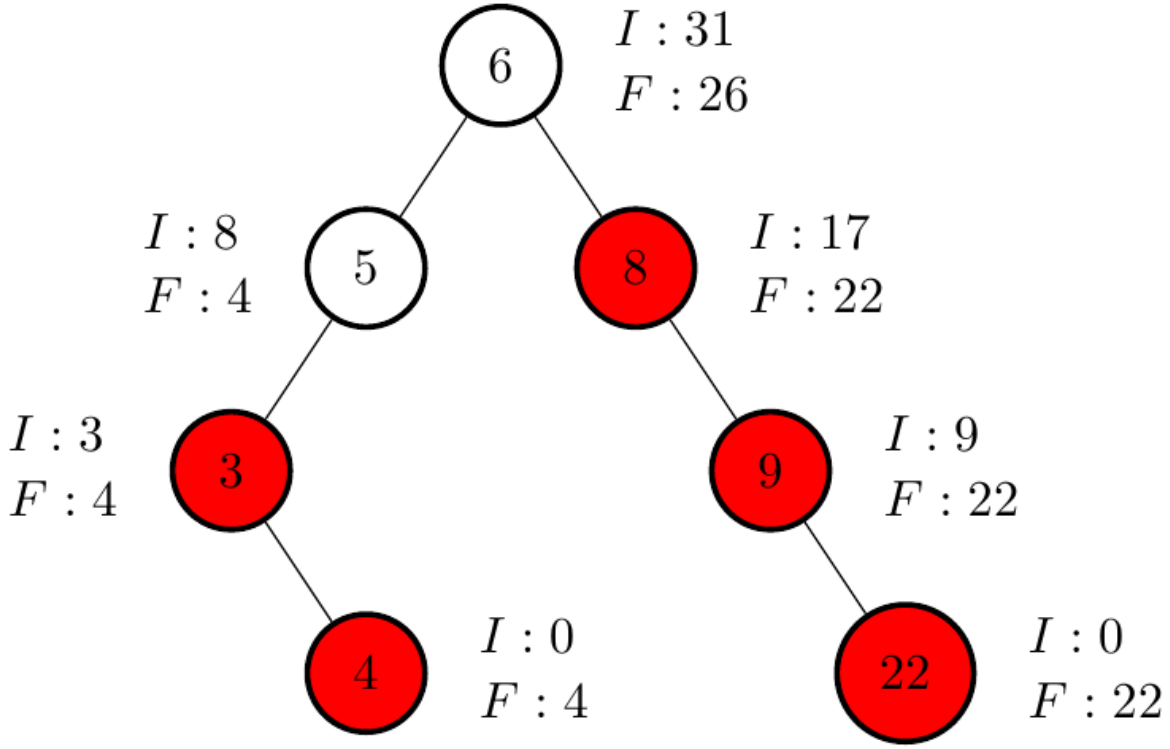
Esercizio: Somma Nodi

- Input:
  - Un intero N
  - N interi
- Operazioni:
  - Inserire gli N interi in un albero binario di ricerca
  - Per ogni nodo *u*, calcolare *I(u)* e *F(u)* (vedi slide seguente)
- Output:
  - Stampare le etichette dei nodi tali che *I(u) <= F(u)*

Somma Nodi

- ***I(u)*: somma delle chiavi dei nodi interni del sottoalbero radicato in *u*, compresa la radice se e solo se questa non è anche foglia**
- ***F(u)*: somma delle chiavi delle foglie del sottoalbero radicato in *u***

STAMPARE ETICETTE DEI NODI IN CUI:
I(u) <= F(u)



Calcolo I(u) e F(u)

- Devo visitare tutto l'albero.
- I valori di I(u) e F(u) di un nodo padre, dipendono dagli stessi valori calcolati per i nodi figli.
- Di quali nodi posso calcolare I(u) e F(u) "al volo"?

- **Suggerimento:** come facevamo a calcolare l'altezza di un nodo? (relazione padre/figli)

Esercizio: Altezza Figli

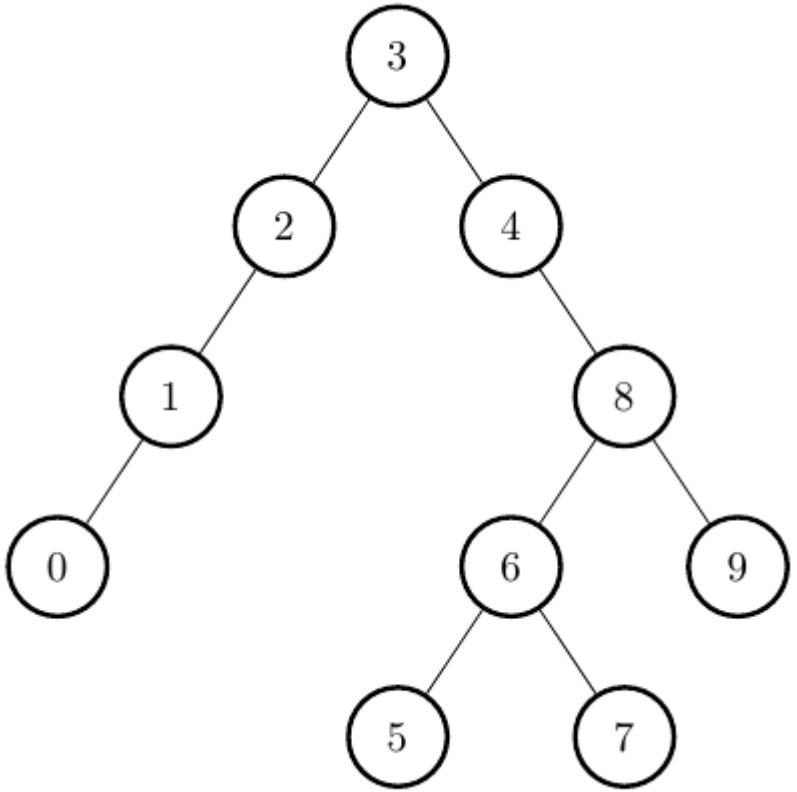
Input:

- N interi da inserire in un albero binario di ricerca

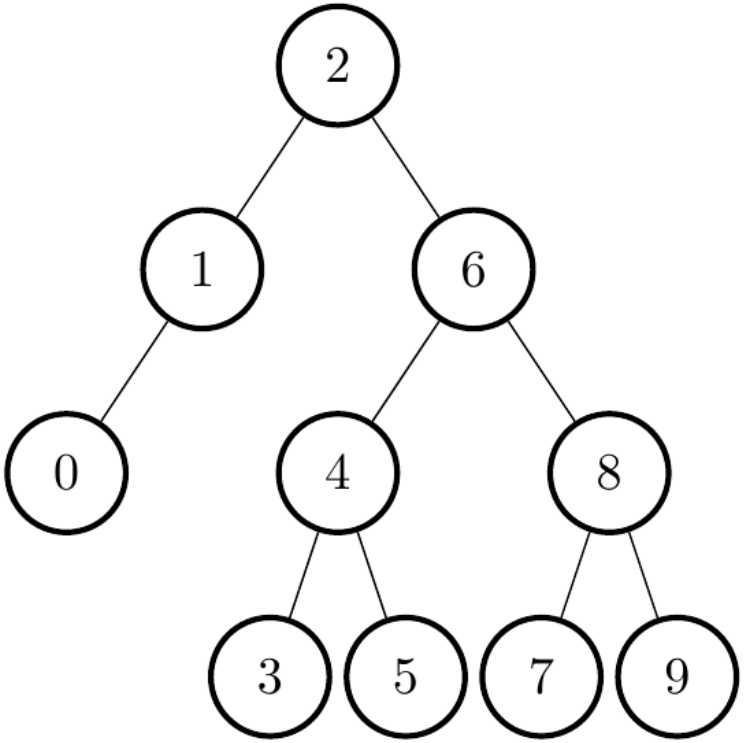
Verificare che :

- Per ciascun nodo, l'altezza dei suoi sottoalberi sinistro e destro deve differire al massimo di uno

Outcome dell'esercizio Altezza figli



no



ok

```
1   bool isOk( Node * tree, int & maxH )
2   {
3      // Controllo se ho raggiunto una foglia
4         return true;
5
6      // Controllo i figli sinistro e destro
7      bool propL = isOk(tree->left,hl);
8      bool propR = isOk(tree->right,hr);
9
10     // ottengo l'altezza del nodo corrente
11     // .. il massimo tra quella sx e dx
12
13     // se la proprietà e' verificata da:
14     // nodo corrente, nodo sx, nodo dx
15        return true;
16     else
17        return false;
18  }
```

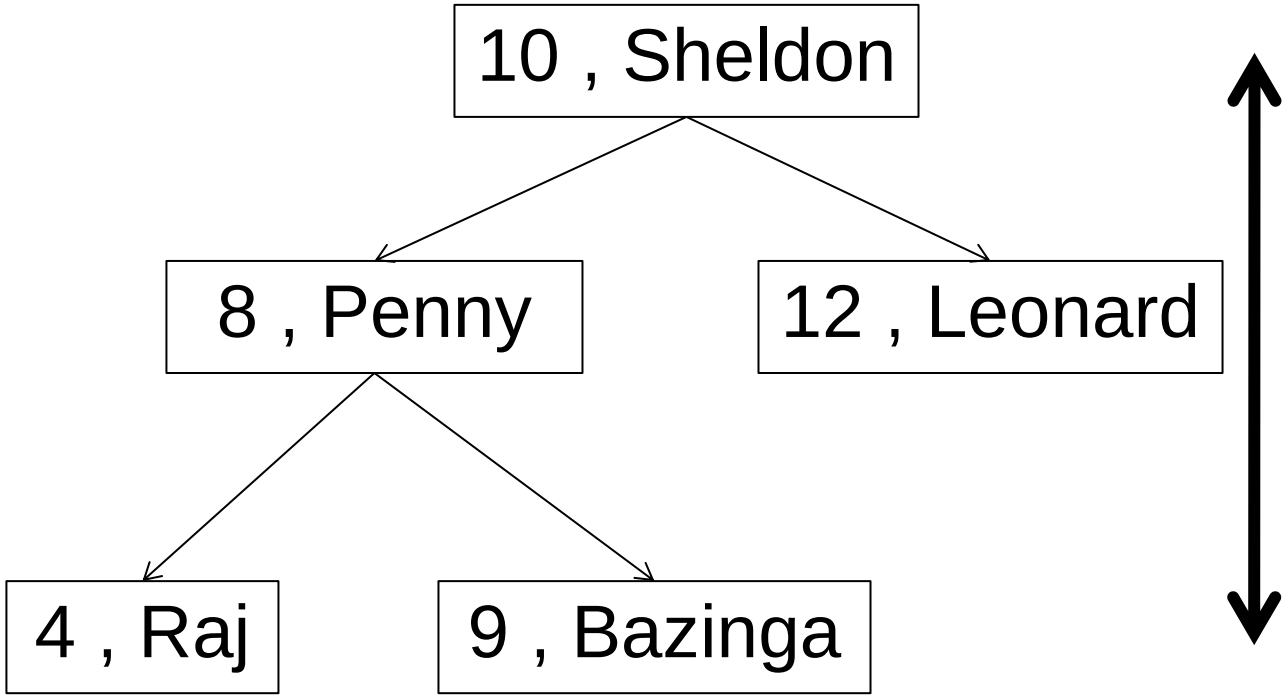
Esercizio: Albero Binario a etichette complesse

- Input:
  - Un intero **N**
  - Un intero **H**
  - N coppie **[intero,stringa]**
- Operazioni:
  - Inserire le N coppie in un albero binario di ricerca (usando il valore intero come chiave)
- Output:
  - stringhe che si trovano in nodi ad altezza H, stampate in ordine lessicografico

Albero Binario a etichette complesse

input

```
5
2
10
Sheldon
12
Leonard
8
Penny
4
Raj
9
Bazinga
```



output

```
Bazinga
Raj
```

Analisi Traccia

- Input:
  - Un intero N
  - Un intero H
  - N coppie [intero,stringa]
- Operazioni:
  - Inserire le N coppie in un albero binario di ricerca
- Output:
  - stringhe che si trovano in nodi ad altezza H, stampate in ordine lessicografico

Analisi Traccia

Implementare struttura dati che supporti
- Albero binario
- Etichette multi valore

```
struct node
{
        int key;
        string str;
        struct node* right;
        struct node* left;
};
```

Funzioni

- Insert su albero binario → insert()
- Trovare nodi ad altezza H → visita+altezza
- Sort su string → sort+compare


IN SUPREMÆ DIGNITATIS
1343

Esercizio: nodi “concordi” e “discordi”

Si consideri un sistema di memorizzazione che legga una sequenza di N interi unici e non negativi e li inserisca dentro un albero binario di ricerca (ABR). Si definiscono:

- Concordi i nodi c caratterizzati da altezza del nodo pari (dispari) e etichetta pari (dispari).
- Discordi i nodi d con altezza pari e etichetta dispari, o viceversa.

L’ altezza del nodo x corrisponde al numero di nodi compresi tra x e la radice dell’albero, x escluso. La radice ha altezza 0. Scrivere un programma che:

- legga da tastiera N, il numero di interi da memorizzare nell’albero;
- legga da tastiera N interi, ovvero le etichette dei nodi da inserire nell’ABR. I valori devono essere inseriti nello stesso ordine con cui vengono letti (le etichette ≤ vanno inserite a sinistra);
- stampi la differenza tra la sommatoria S delle etichette dei nodi discordi e la stessa dei nodi concordi

$$S = \sum label(d) - \sum label(c)$$

Outcome: “concordi” e “discordi”

- L’input è formattato nel seguente modo: la prima riga contiene l’intero N. Seguono N righe contenenti un’etichetta ciascuna.
- L’output contiene la soluzione su un unica riga.

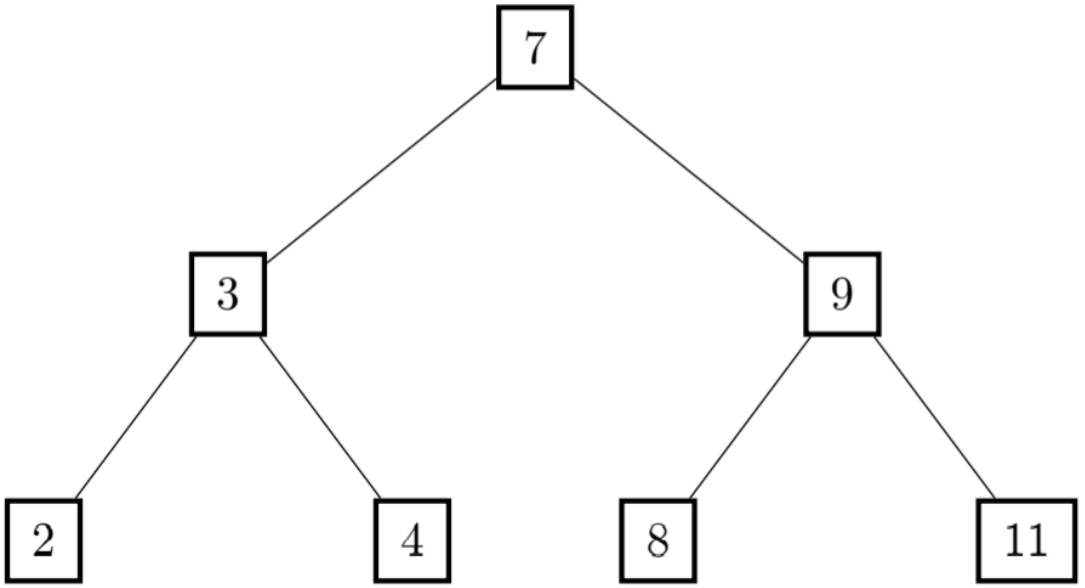
Esempio

Input

```
7
7
3
9
2
4
8
11
```

Output

```
-8
```



Possibile Soluzione: nodi “concordi” e “discordi”

```
#include <iostream>      // std::cout
#include <algorithm>     // std::sort
#include <vector>        // std::vector
#include <fstream>
#include <math.h>        /* floor */
#include <stdlib.h>
#include <cmath>         /* pow  */

using namespace std;


struct Node
{
    int value;
    Node * left;
    Node * right;

    Node( int i ): value(i) , left(NULL) , right(NULL) {}
};
```

```
class BinTree
{
    Node * root_;
public:
    BinTree() { root_ = NULL ; }

    Node * getRoot() { return root_; cout << "getRoot" << endl;}

    void insert( int i )
    {
        Node * node = new Node(i);

        Node * pre = NULL;
        Node * post = root_;
        while( post != NULL)
        {
            pre = post;
            if( i <= post->value )
            {
                post = post->left;
            }
            else
            {
                post = post->right;
            }
        }

        if( pre == NULL )
            root_ = node;
        else if( i <= pre->value )
        {
            pre->left = node;
        }
        else
        {
            pre->right = node;
        }
        return;
    }
};
```

```
int layerTree( Node * tree, int altezza)
{
     // Nodo non trovato
     if( tree == NULL)  {
        return 0;
     }

     if (altezza++%2 != tree->value%2)  {
          //cout<<tree->value<<" : "<<altezza-1<<" discorde"<<endl;
          return layerTree( tree->left , altezza) + layerTree( tree->right , altezza) + tree->value;
     }
     else   {
          //cout<<tree->value<<" : "<<altezza-1<<" concorde"<<endl;
          return layerTree( tree->left , altezza) + layerTree( tree->right , altezza) - tree->value;
     }
}


int main()
{
    int N ;
    int x ;
    BinTree albero ;

    cin >> N ;

    // riempio l' albero
    for(int i=0 ; i<N ; ++i )   {
        cin >> x;
        albero.insert(x);
    }

    cout<<layerTree(albero.getRoot(), 0)<<endl;
}
```

Esercizio: nodi “completi”

Si consideri un sistema di memorizzazione che legga una sequenza di N interi unici e non negativi e li inserisca dentro un albero binario di ricerca (ABR). Si definiscono completi i nodi c aventi due nodi figli, incompleti i nodi i restanti. Scrivere un programma che:

- legga da tastiera N, il numero di interi da memorizzare nell’albero;
- legga da tastiera N interi, ovvero le etichette dei nodi da inserire nell’ABR. I valori devono essere inseriti nello stesso ordine con cui vengono letti (le etichette ≤ vanno inserite a sinistra);
- stampi la differenza tra la sommatoria S delle etichette dei nodi completi e la stessa dei nodi incompleti

$$S = \sum label(c) - \sum label(i)$$

Outcome: “completi”

- L’input è formattato nel seguente modo: la prima riga contiene l’intero N. Seguono N righe contenenti un’etichetta ciascuna.
- L’output contiene la soluzione su un unica riga

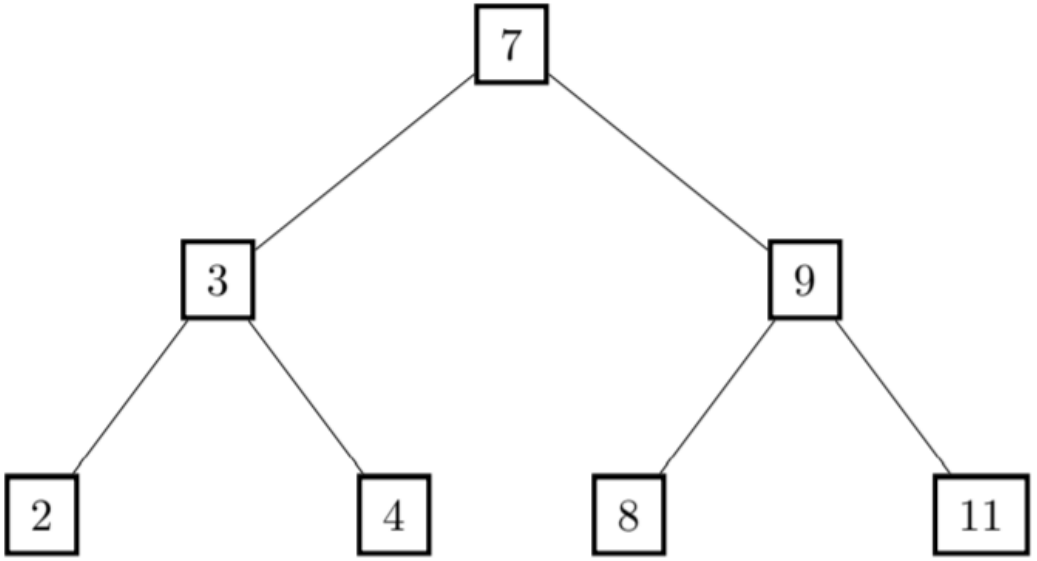
Esempio

Input

```
7
7
3
9
2
4
8
11
```

Output

```
-6
```



Possibile soluzione: nodi “completi”

```
#include <iostream>      // std::cout
#include <algorithm>     // std::sort
#include <vector>        // std::vector
#include <fstream>
#include <math.h>        /* floor */
#include <stdlib.h>
#include <cmath>         /* pow  */

using namespace std;


struct Node
{
    int value;
    Node * left;
    Node * right;

    Node( int i ): value(i) , left(NULL) , right(NULL) {}
};
```

```
class BinTree
{
    Node * root_;
public:
    BinTree() { root_ = NULL ; }

    Node * getRoot() { return root_; cout << "getRoot" << endl;}

    void insert( int i )
    {
        Node * node = new Node(i);

        Node * pre = NULL;
        Node * post = root_;
        while( post != NULL)
        {
            pre = post;
            if( i <= post->value )
            {
                post = post->left;
            }
            else
            {
                post = post->right;
            }
        }

        if( pre == NULL )
            root_ = node;
        else if( i <= pre->value )
        {
            pre->left = node;
        }
        else
        {
            pre->right = node;
        }
        return;
    }
};
```

```
int complete( Node * tree)
{
     // Nodo non trovato
     if( tree == NULL)  {
        return 0;
     }

     if (tree->left != NULL && tree->right != NULL)  {
          return complete( tree->left ) + complete( tree->right ) + tree->value;
     }
     else   {
          return complete( tree->left ) + complete( tree->right ) - tree->value;
     }
}


int main()
{
     int N ;
     int x ;
     BinTree albero ;

     cin >> N ;

     // riempio l' albero
     for(int i=0 ; i<N ; ++i )   {
          cin >> x;
          albero.insert(x);
     }

     cout<<complete(albero.getRoot())<<endl;
}
```


IN SUPREMÆ DIGNITATIS
1343