Algoritmi e Strutture Dati

Lezione 9

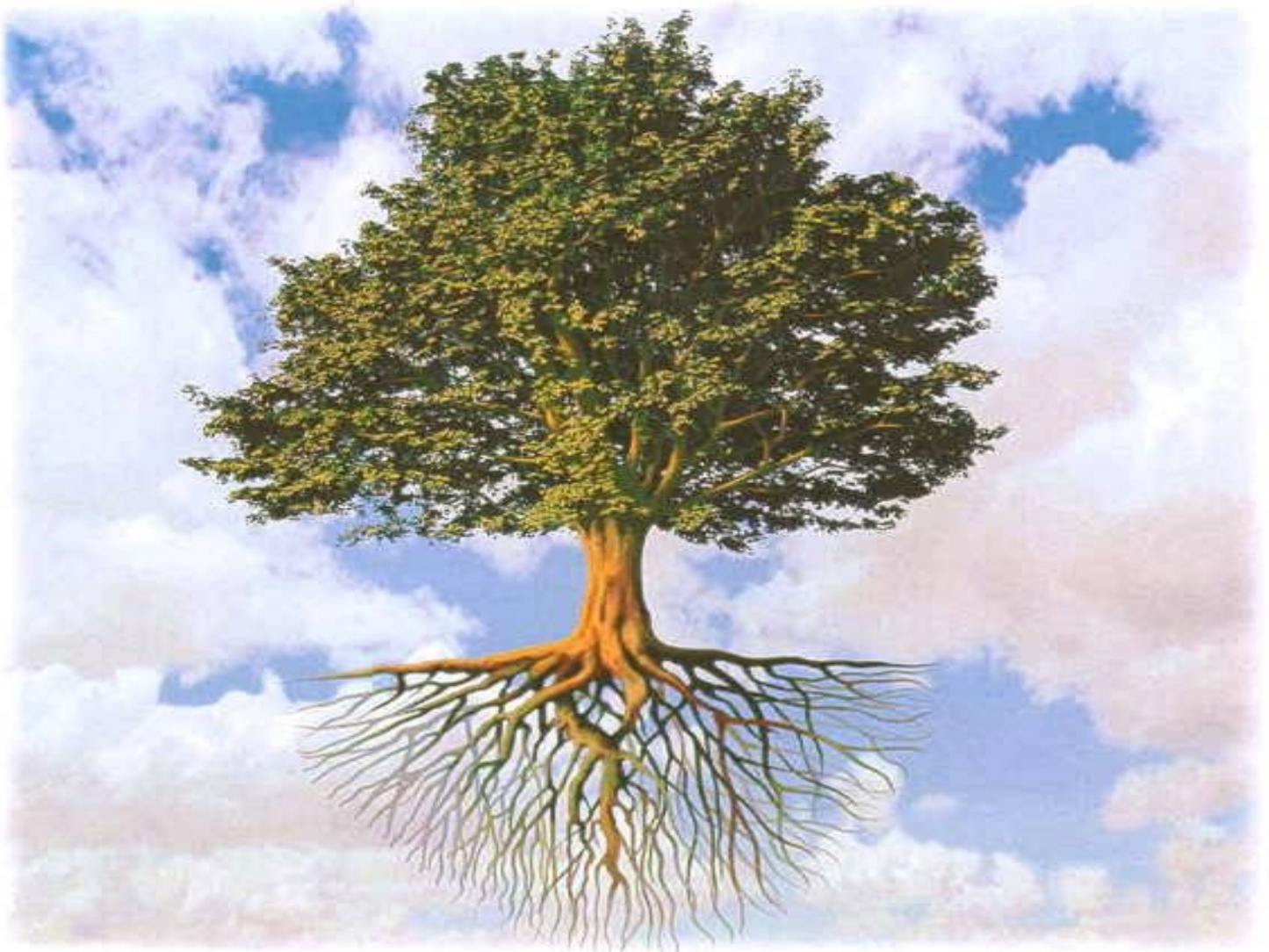
www.iet.unipi.it/a.virdis

Antonio Virdis

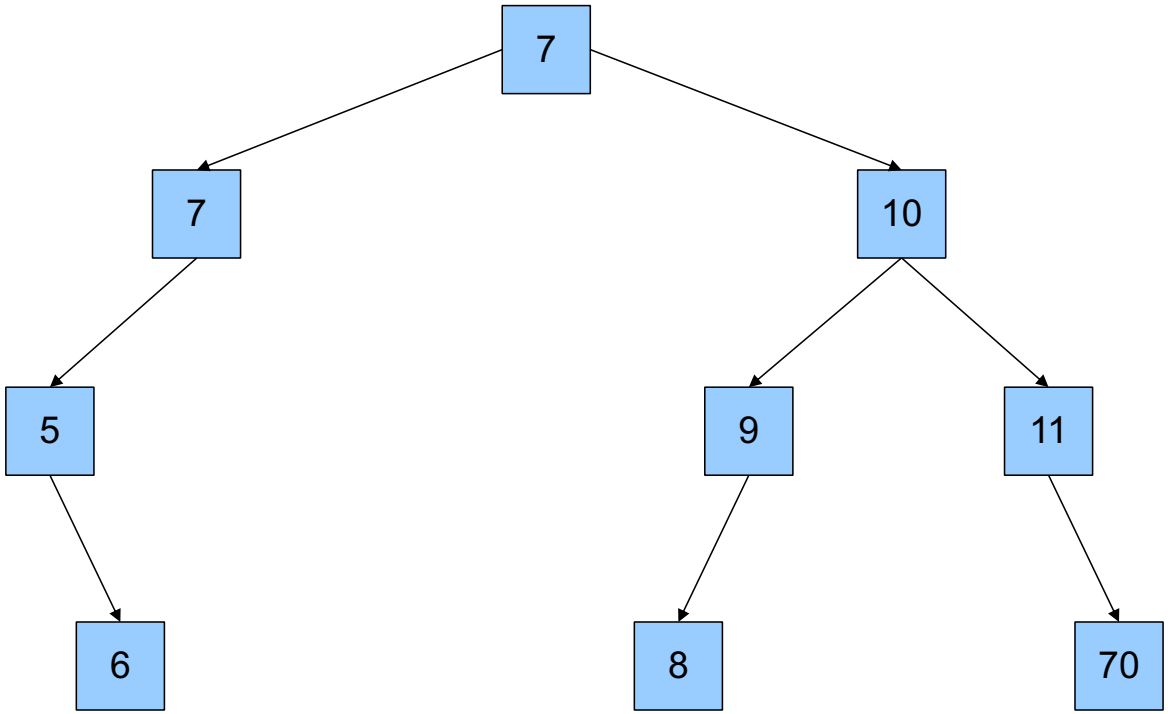
antonio.virdis@unipi.it

Sommario

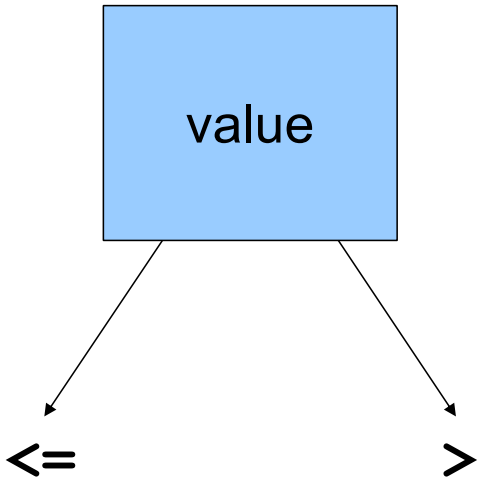
- Alberi Binari di Ricerca
- Gestione Stringhe
- Progettazione
- Esercizi



Alberi Binari



Alberi Binari di Ricerca



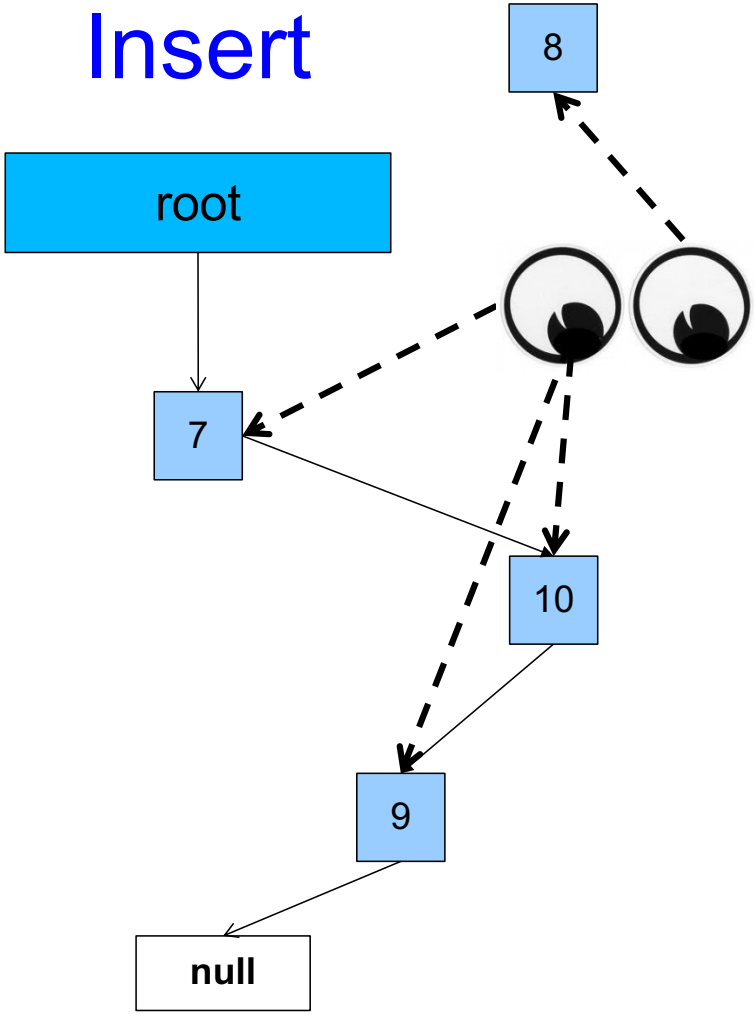
binTree

```
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  }
```

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 un posto per una foglia
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 un posto per 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 ad un posto per 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
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 un posto per 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

Insert

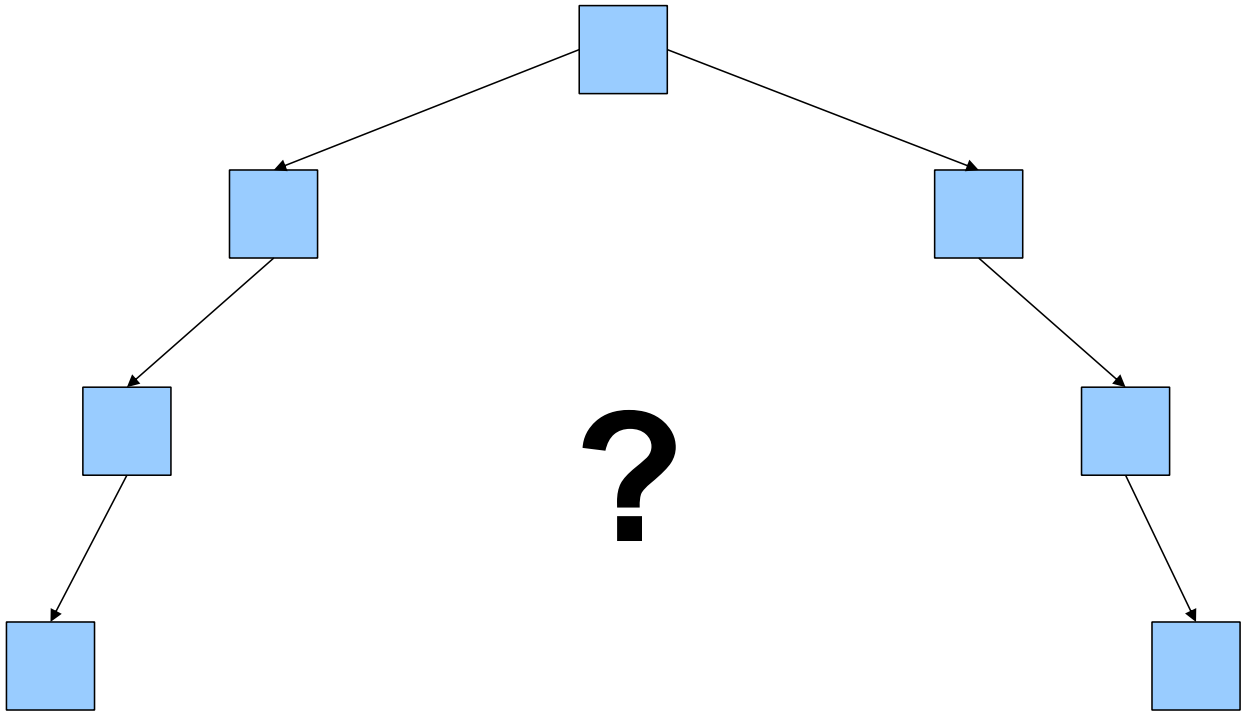
```
1   void insert( int val )
2   {
3       Node * node = new Node(val);
4       Node * pre = NULL;
5       Node * post = root_;
6
7       while( post != NULL )
8       {
9           pre = post;
10          if( val <= post->value )
11              post = post->left;
12          else
13              post = post->right;
14      }
15      if( pre == NULL )
16          root_ = node;
17      else if( val <= pre->value )
18          pre->left = node;
19      else
20          pre->right = node;
21  }
```

INIZIALIZZAZIONE

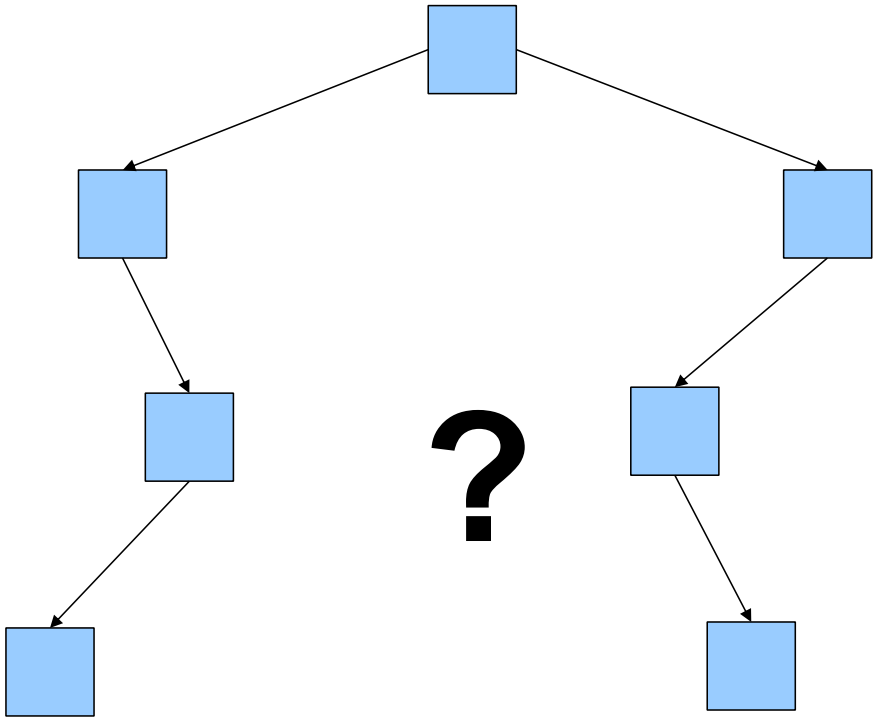
INDIVIDUO POSIZIONE

INSERIMENTO

min & MAX



min & MAX



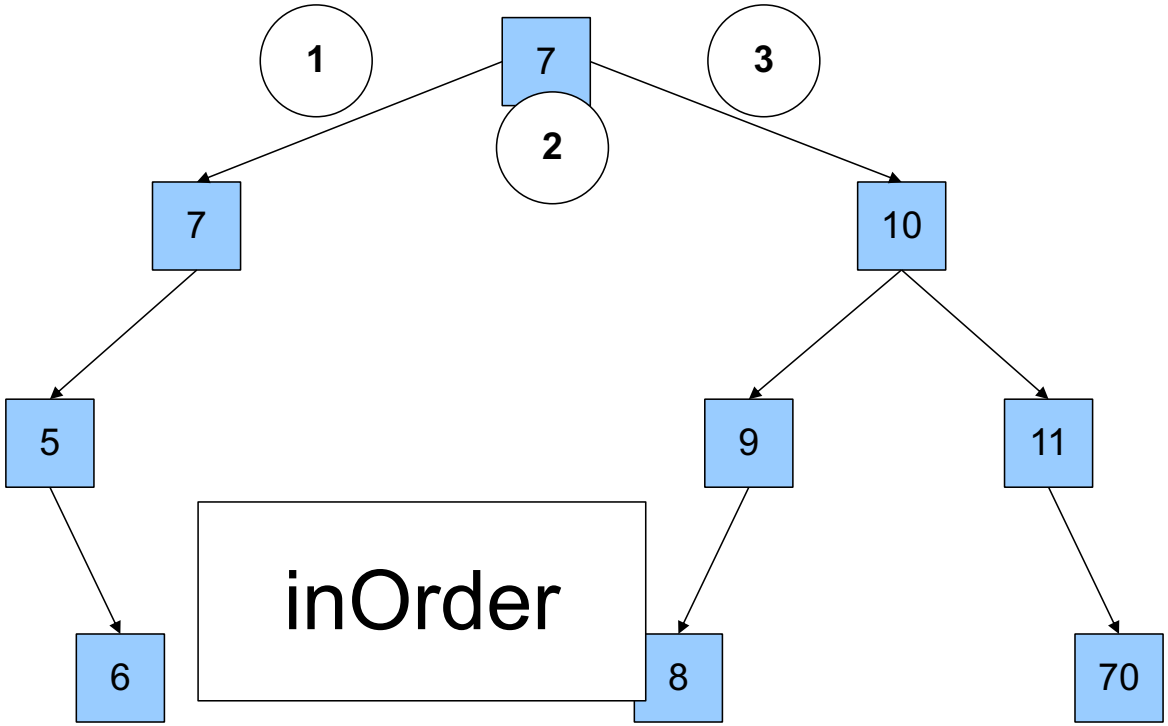
Min/Max

```
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
```

visite



In-Order

```
1
2
3
4   void inOrder( Node * tree )
5   {
6       // se l'albero non è 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 è 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

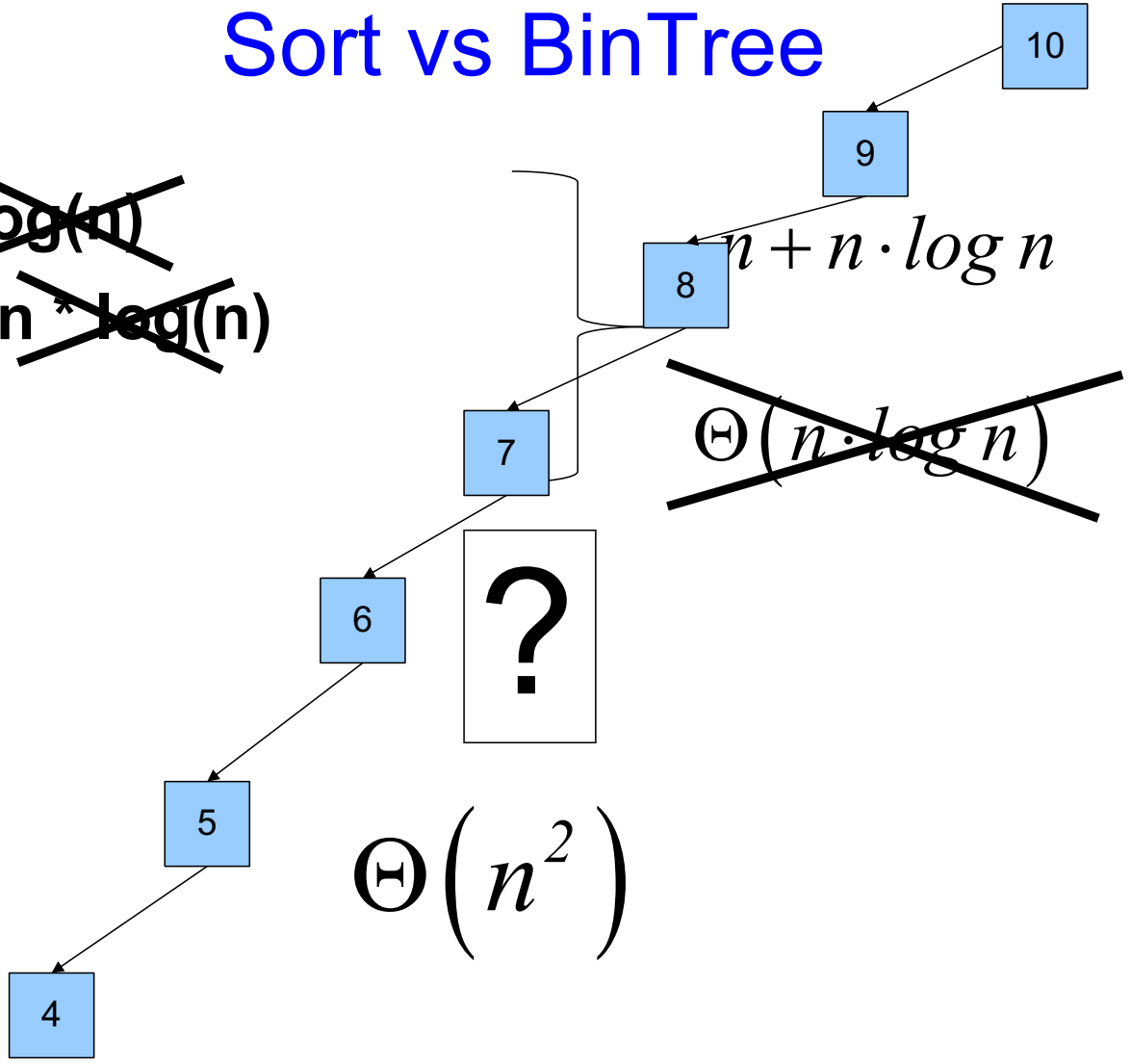
```
1
2
3
4   void inOrder( Node * tree )
5   {
6       if(tree!=NULL)
7       {
8
9           visitaNodo(tree->left);
10
11          cout << tree->value << "\t";
12
13          visitaNodo (tree->right);
14
15      }
16  }
17
18
19
20
```

In-Order

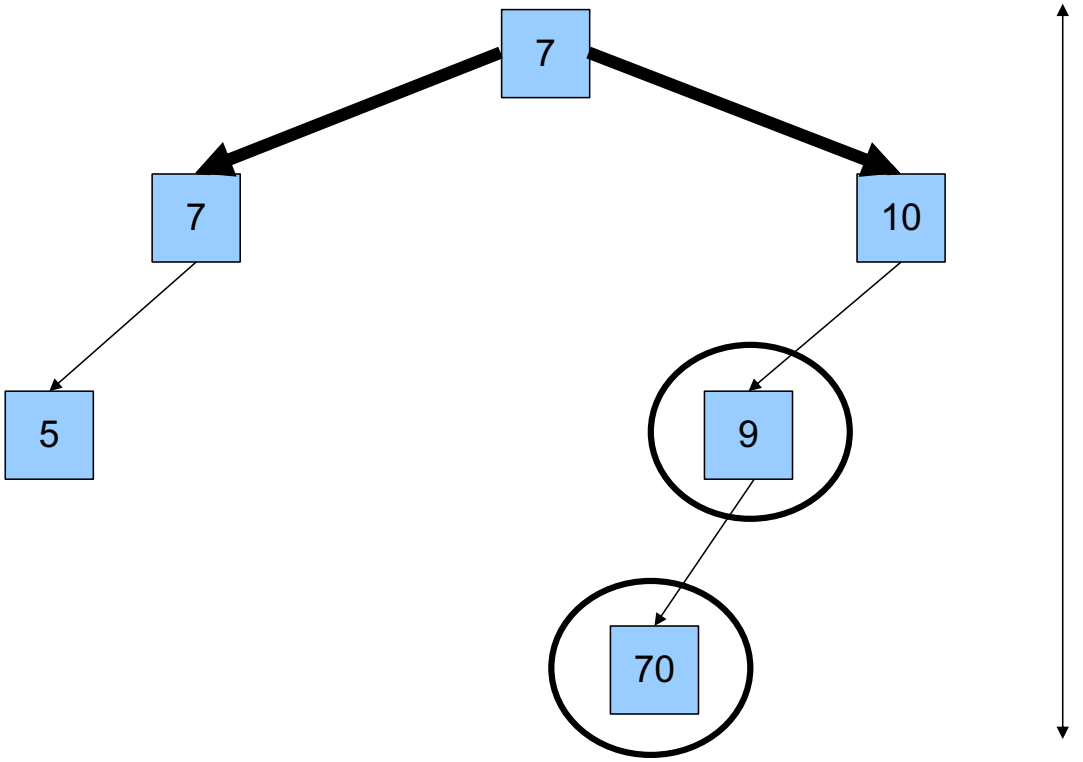
```
1
2
3
4   void inOrder( Node * tree )
5   {
6       if(tree!=NULL)
7       {
8
9           inOrder(tree->left);
10
11          cout << tree->value << "\t";
12
13          inOrder(tree->right);
14
15      }
16  }
17
18
19
20
```

Sort vs BinTree

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



Height



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
```

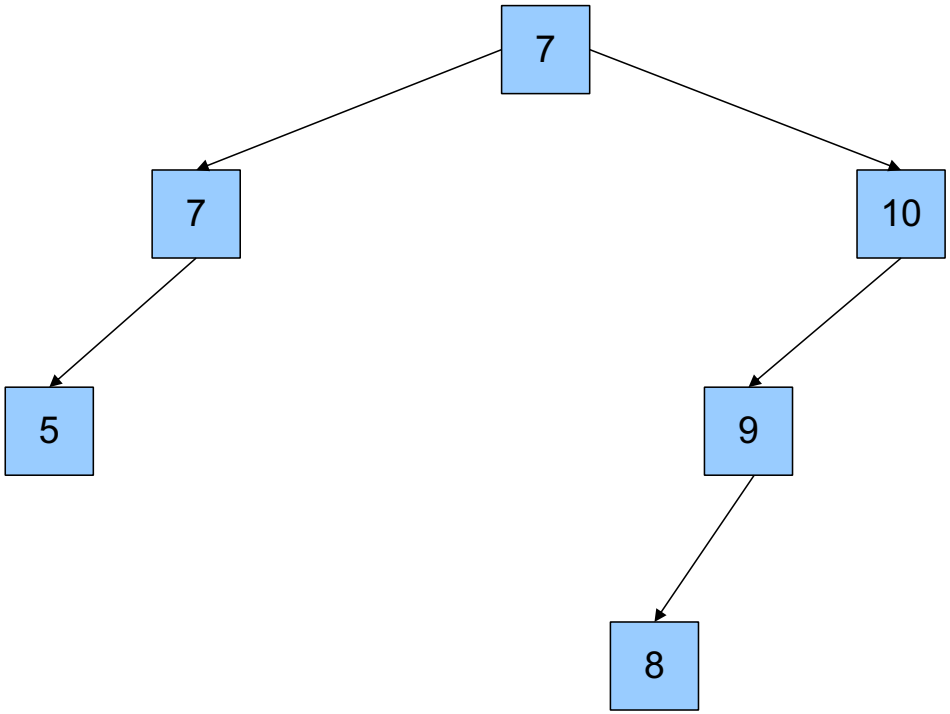
Altezza albero

```
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
```

Trova chiave

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

Search K=9



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
```

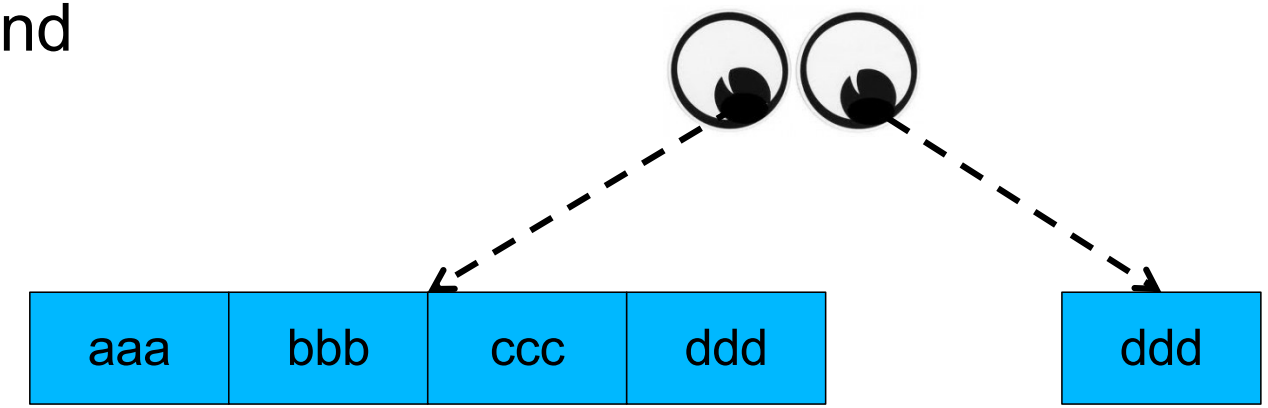
Search

```
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

stringhe

- Creazione
- Concatenazione
- Compare
- Find



Stringhe

```
1   #include <string>
2
3   String parola = "liste";
4
5
6
7
8
9
10
11
12
13
14
```

Stringhe

```
1   #include <string>
2
3   String parola = "liste";
4
5   String frase = "mi piacciono le liste";
6
7
8
9
10
11
12
13
14
```

Stringhe

```
1   #include <string>
2
3   String parola = "liste";
4
5   String frase = "mi piacciono le liste";
6
7   String parola2 = "non ";
8
9   String frase2 = parola2 + frase;
10
11
12
13
14
```

Stringhe

```
1   #include <string>
2
3   String parola = "liste";
4
5   String frase = "mi piacciono le liste";
6
7   String parola2 = "non ";
8
9   String frase2 = parola2 + frase;
10
11  frase.find(parola);
12  // se fallisce -> string::npos
13
14
```

Stringhe

```
1   #include <string>
2
3   String parola = "liste";
4
5   String frase = "mi piacciono le liste";
6
7   String parola2 = "non ";
8
9   String frase2 = parola2 + frase;
10
11  frase.find(parola);
12  // se fallisce -> string::npos
13
14  parola.compare(parola2);
```

http://www.cplusplus.com/reference/string/string/

Esercizio Stringhe

- Input
  - Una testo T formato da più parole
  - Un insieme S di N parole

- Output:
  1. Le parole di S *contenute* in T, ordinate per posizione in T (insieme R1)
  2. Le parole di S *non contenute* in T, in ordine lessicografico (insieme R2)

Analisi

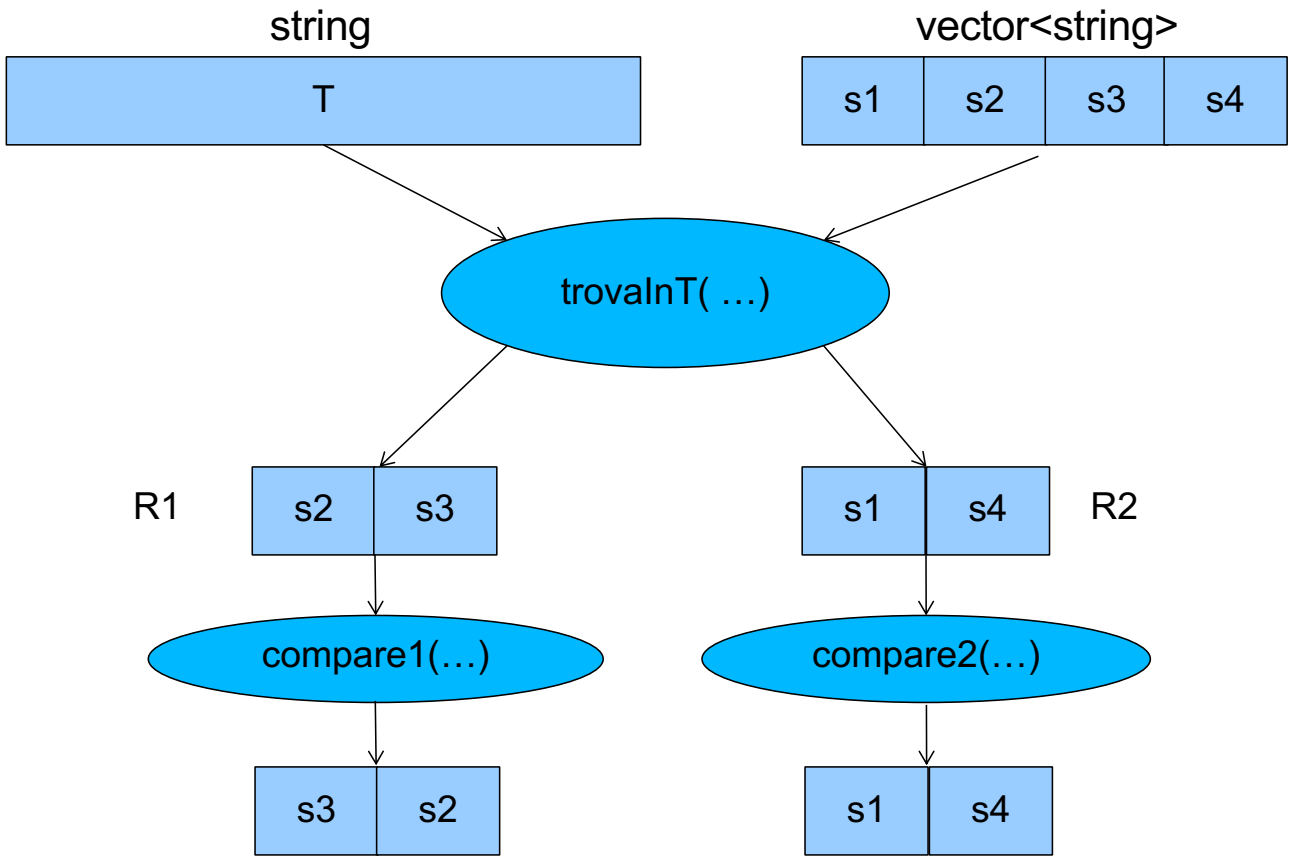
Strutture Dati:

- Dove salvo T?
- Dove salvo S?

Operazioni:

- Come ottengo gli elementi di 1?
- Come ottengo gli elementi di 2?
- Come ordino 1?
- Come ordino 2?

Analisi (2)


string
vector<string>
T
s1
s2
s3
s4
trovaInT(…)
R1
s2
s3
s1
s4
R2
compare1(…)
compare2(…)
s3
s2
s1
s4

Implementazione

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

Implementazione

```
1   // cerca stringhe di S dentro T
2   void trovaInT( … ){    }
3
4   // implementa confronto per posizione
5   bool compare1(string a, string b){   }
6
7   // implementa confronto lessicografico
8   bool compare2(string a, string b){   }
9
10
11
12
13
14
15
16
17
18
19
20
```

Implementazione

```
1   // cerca stringhe di S dentro T
2   void trovaInT( … ){    }
3
4   // implementa confronto per posizione
5   bool compare1(string a, string b){   }
6
7   // implementa confronto lessicografico
8   bool compare2(string a, string b){   }
9
10  int main()
11  {
12      string T;
13      vector <string> S, R1, R2;
14
15
16
17
18
19
20
    }
```

Implementazione

```
1   // cerca stringhe di S dentro T
2   void trovaInT( … ){    }
3
4   // implementa confronto per posizione
5   bool compare1(string a, string b){   }
6
7   // implementa confronto lessicografico
8   bool compare2(string a, string b){   }
9
10  int main()
11  {
12      string T;
13      vector <string> S, R1, R2;
14
15      // lettura T ed S
16
17      trovaInT( … );
18      sort( R1.begin(), R1.end(), compare1 )
19      sort( R2.begin(), R2.end(), compare2 )
20
    }
```

Implementazione

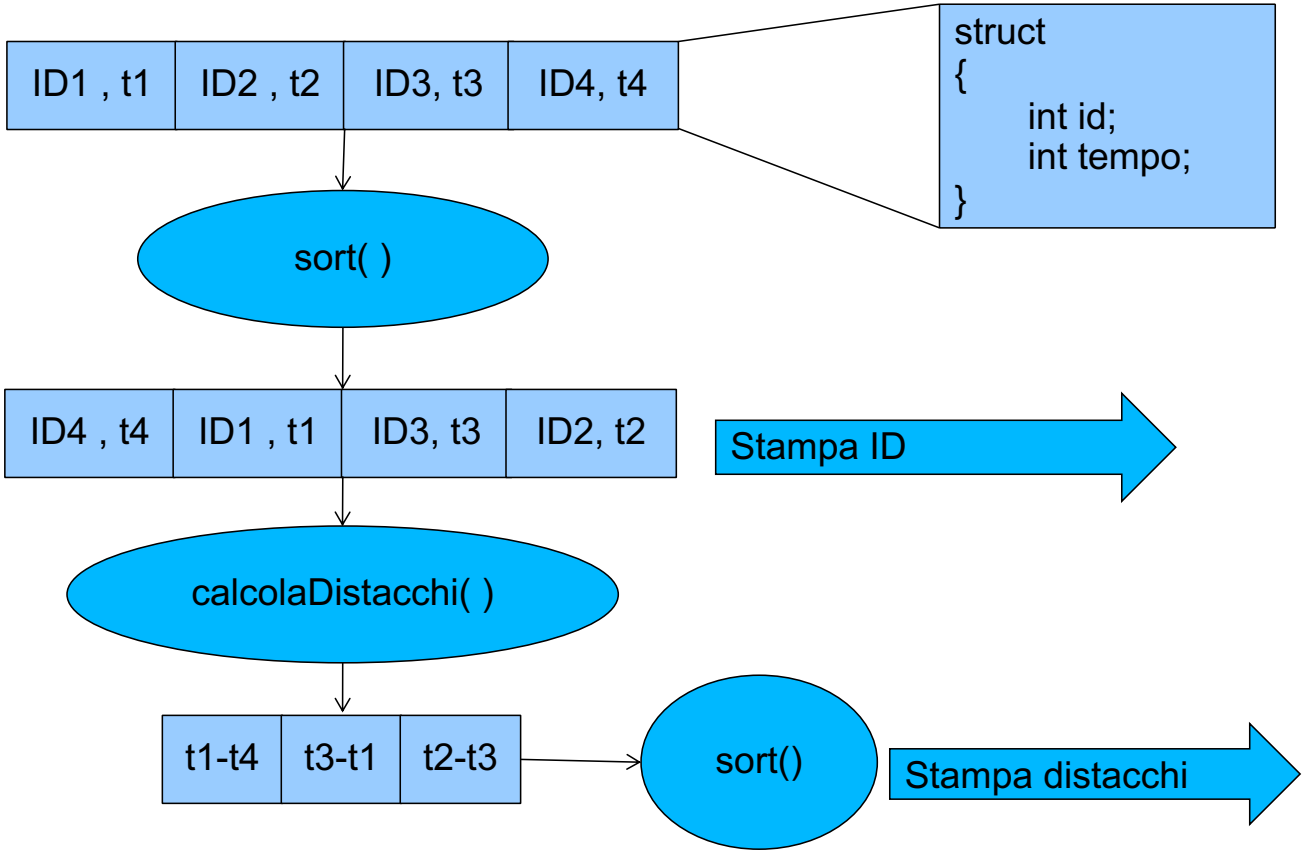
```
1   // cerca stringhe di S dentro T
2   void trovaInT( … ){    }
3
4   // implementa confronto per posizione
5   bool compare1(string a, string b){   }
6
7   // implementa confronto lessicografico
8   bool compare2(string a, string b){   }
9
10  int main()
11  {
12      string T;
13      vector <string> S, R1, R2;
14
15      // lettura T ed S
16
17      trovaInT( … );
18      sort( R1.begin(), R1.end(), compare1 )
19      sort( R2.begin(), R2.end(), compare2 )
20      print();
    }
```

Gara

- Ad una gara partecipano N concorrenti
- Ogni concorrente e' caratterizzato da:
  - Un ID intero
  - Un tempo di arrivo espresso in secondi

- Calcolare:
  - Classifica
  - K distacchi più ampi di utenti **consecutivi**

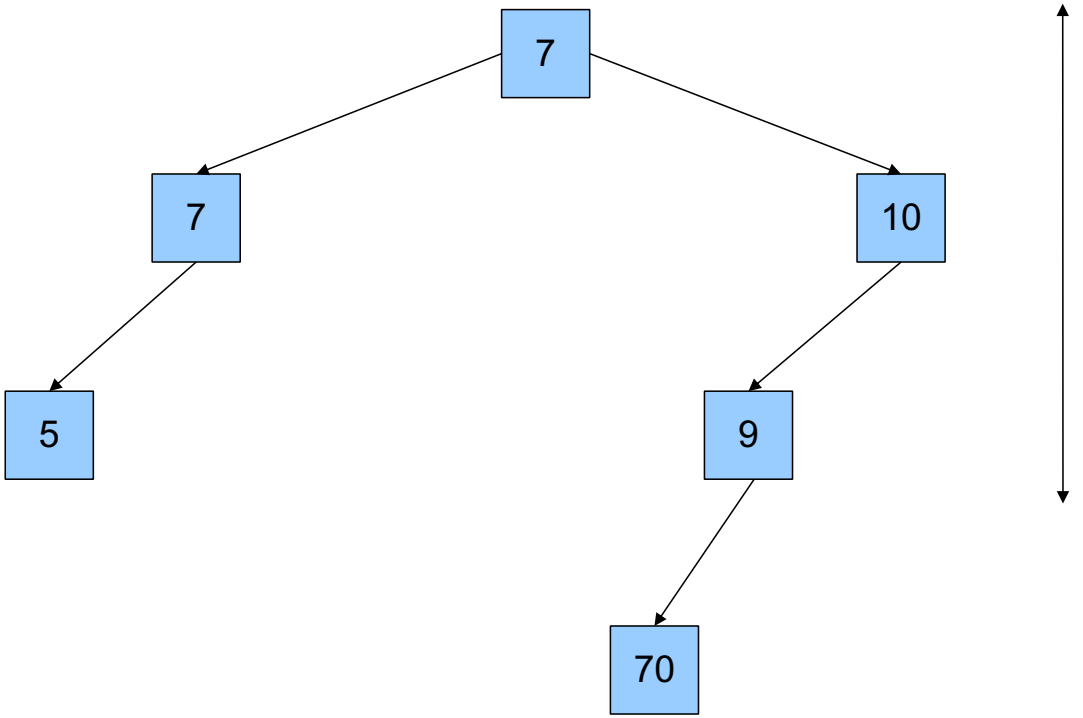
Analisi


ID1 , t1
ID2 , t2
ID3, t3
ID4, t4
struct
{
int id;
int tempo;
}
sort( )
ID4 , t4
ID1 , t1
ID3, t3
ID2, t2
Stampa ID
calcolaDistacchi( )
t1-t4
t3-t1
t2-t3
sort()
Stampa distacchi

Trovare livello chiave K

- Input
  - Una sequenza di N interi positivi
  - Chiave K
- Output
  - il livello della chiave K dentro l'albero (se esiste)

Search K=9



Livello

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

Livello

```
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
```

Livello

```
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
```

Livello

```
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

Esercizi

- Esperimenti
  - Sort vs Binary Tree
  - Sort vs Min/Max

- Esercizi
  - Visite pre- e post-order
  - Esercizi di progettazione