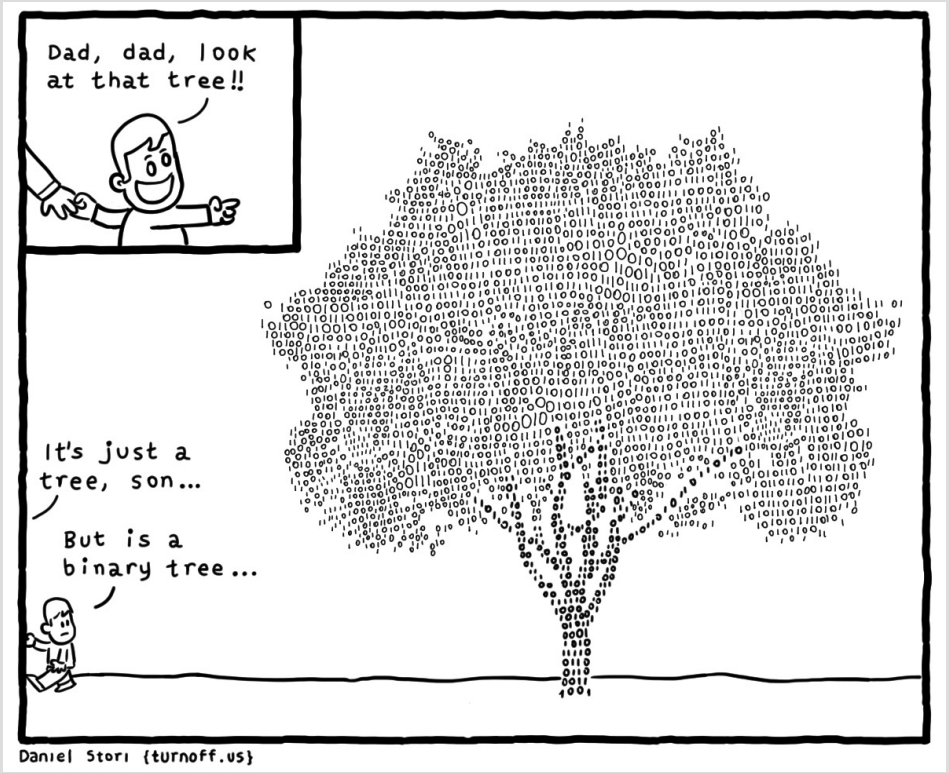
Parte 4 – Alberi

Alberi binari di ricerca



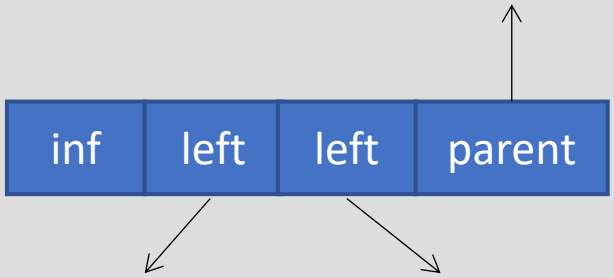
Alberi binari

- Gli **alberi binari** sono un caso particolare di albero n-ario dove il numero dei figli per ogni nodo è al più due
- Negli alberi binari si fa distinzione tra il figlio sinistro ed il figlio destro di un nodo
- La struttura dati per rappresentare i nodi degli alberi mantiene un riferimento esplicito al figlio sinistro e al figlio destro
- Le primitive associate ad un albero binario sono un caso particolare delle primitive per alberi n-ari

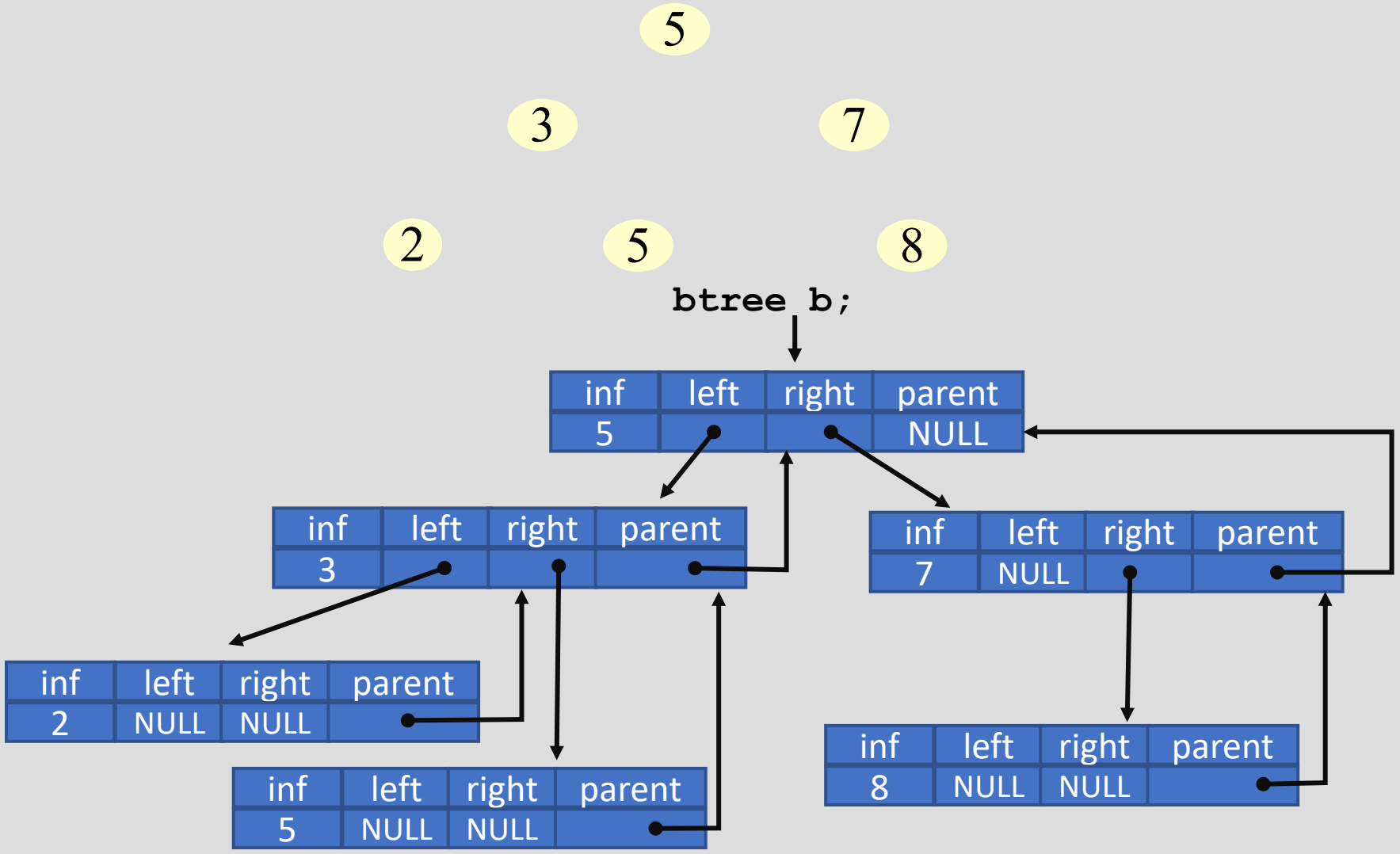
Struttura dati albero binario

```
//definizione della struttura nodo
struct bnode {
      tipo_inf inf;
      bnode* left;
      bnode* right;
      bnode* parent;
    };

//dichiarazione del tipo di dato binary tree
typedef bnode* btree;
```



Rappresentazione di albero binario



Alberi di ricerca

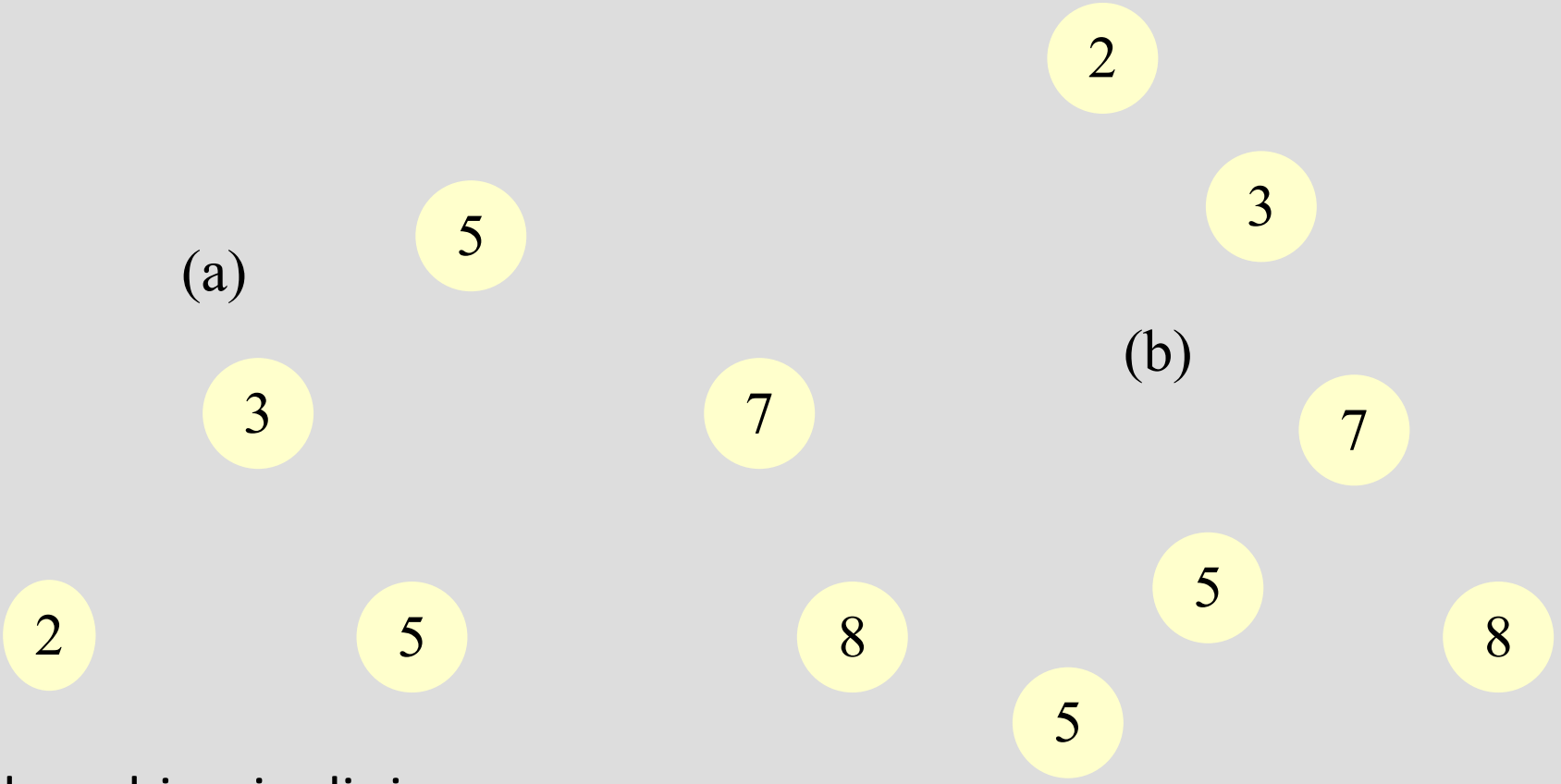
- Gli **alberi di ricerca** sono strutture dati dinamiche ad albero usate per localizzare efficientemente valori di chiave in insiemi di elementi *(chiave, valore)* dove esiste un ordinamento totale sulle chiavi
- Gli alberi di ricerca possono essere utilizzati per implementare, ad esempio, dizionari o code di priorità
- Esistono diverse tipologie di alberi di ricerca:
  - Alberi binari di ricerca
  - Alberi AVL
  - B-tree
  - Alberi 2-3-4
- In questa lezione vedremo gli **ALBERI BINARI DI RICERCA**

Albero binario di ricerca

Un **albero binario di ricerca (binary search tree BST)** è un albero binario che soddisfa le seguenti proprietà:

- Ogni **nodo *n*** ha
  - un contenuto informativo ***value(n)***
  - una chiave ***key(n)*** presa da un dominio totalmente ordinato (ovvero su cui è definita una *relazione d'ordine totale* <)
- Sia *n'* un nodo nel **sottoalbero sinistro** di *n* allora *key(n')≤key(n) (oppure key(n')<key(n))*
- Sia *n'* un nodo nel **sottoalbero destro** di *n* allora *key(n')> key(n) (oppure key(n') ≥ key(n))*

Esempi di alberi binari di ricerca



Albero binario di ricerca con
chiave intera e relazione d'ordine minore (<).
I valori mostrati sono relativi al valore della chiave.

Implementazione degli alberi binari di ricerca

- Negli alberi binari di ricerca, ogni nodo contiene un valore di chiave che ne determina l'accesso e il valore ad esso associato
- Supponiamo ad esempio di avere una chiave intera e un valore semplice di tipo stringa
- Il nodo può essere implementato aggiungendo esplicitamente il campo chiave

```
typedef int tipo_key;
typedef char* tipo_inf;
struct bnode {
      tipo_key key;
      tipo_inf inf;
      bnode* left;
      bnode* right;
      bnode* parent;};
typedef bnode* bst;
```

Primitive

Per semplicità assumiamo che i valori di chiave siano univoci nel BST.

Le primitive associate ad un albero binario di ricerca sono:

PRIMITIVE DI CREAZIONE E ACCESSO AL NODO DI UN BST

- `bnode* bst_newNode(tipo_key, tipo_inf)`: crea un nuovo nodo con chiave e contenuto informativo dato
- `tipo_key get_key(bnode*)`: restituisce la chiave del nodo in ingresso

Primitive (cont.)

- `tipo_inf get_value(bnode*)`: restituisce il valore del nodo in ingresso
- `bst get_left(bst)`: restituisce il sottoalbero sinistro dell'albero in ingresso
- `bst get_right(bst)`: restituisce il sottoalbero destro dell'albero in ingresso
- `bnode* get_parent(bnode* n)` : restituisce il padre dell'albero in ingresso

Primitive (cont.)

PRIMITIVE AGGIORNAMENTO BST

- `void bst_insert(bst&, bnode*)`: Aggiunge un nodo all'albero di ricerca
- `void bst_delete(bst&,bnode*)`: Cancella un nodo dall'albero di ricerca

PRIMITIVE DI ACCESSO A BST

- `bnode* bst_search(bst,tipo_key)`: Restituisce il nodo associato alla chiave in ingresso, se esiste

La primitiva
`void bst_insert(bst&,bnode*)`

- Ogni inserimento richiede di aggiungere un nodo all'albero, in modo da *mantenere la proprietà di ricerca*
- Il nodo viene sempre aggiunto come figlio di un **nodo foglia**
- L'inserimento richiede quindi di scandire l'albero dalla radice verso il nodo foglia che diventerà il padre del nuovo nodo

Percorso di scansione dell'albero

```
void bst_insert(bst& b, bnode* n)
```

Partendo dalla radice, il *percorso di scansione dipende dall'esito del confronto* tra il nodo corrente `z` e il nodo da inserire `n`: ***key(z)*** e ***key(n)***

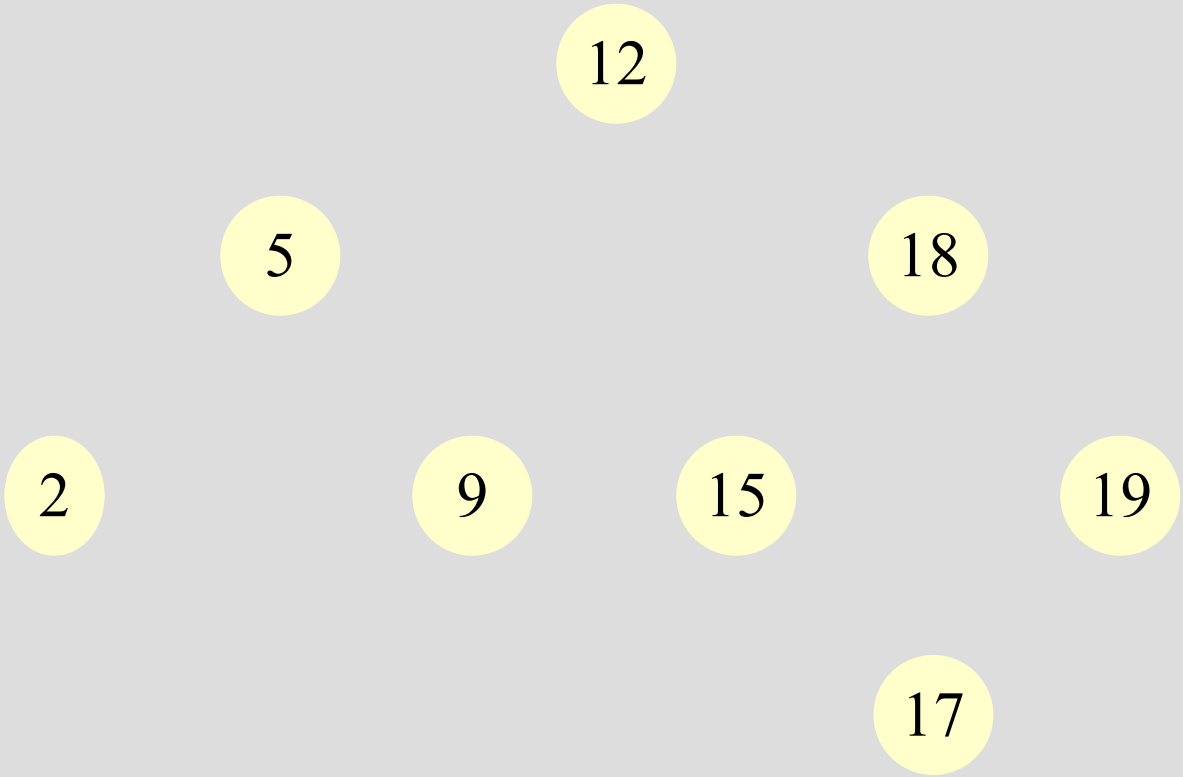
- Se ***key(n) < key(z)*** la scansione prosegue nel sottoalbero sinistro
- Se ***key(z) < key(n)*** la scansione prosegue nel sottoalbero di destra

La scansione termina quando il sottoalbero selezionato è vuoto ovvero ha valore **NULL**

Al termine della scansione si inserisce il nuovo nodo

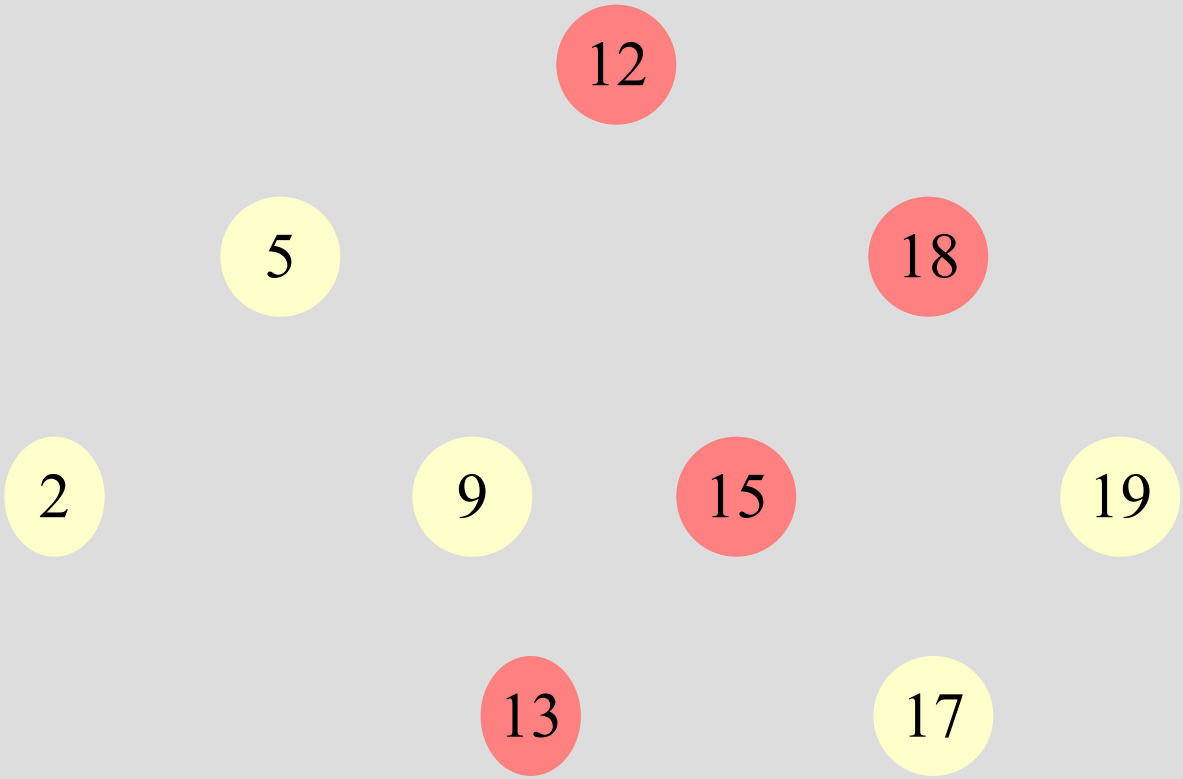
Esempio

Inserimento del valore 13



Esempio

Inserimento del valore 13



E quando key(z) == key(n)?

Esercizio

Creare un progetto per la gestione di bst:

- Costruire il modulo «BST» contenente
  1. i tipi di dato per implementare un BST
  2. la primitiva `compare_key` (con la stessa semantica di compare) utilizzata dalla primitiva `bst_insert`
  3. La primitiva `copy_key` utilizzata dalla primitiva `bst_newNode`
  4. le primitive di creazione e accesso al nodo
  5. la primitiva per l'aggiunta di un nodo `bst_insert`
- Creare il modulo «main» contenente un main di prova dove viene creato un BST con valori inseriti dall'utente

Vedi cartella *bst_crea:* Il file *bst.cc* contiene 1-4 e lascia 5 da completare .

SOLUZIONE Vedi file *bst_sol.cc*

Versione ricorsiva di `bst_insert`

```
void bst_insert(bst& b, bnode* n){
if(b==NULL){  b=n;
              return;}
if (compare_key(get_key(n),get_key(b))<0)
      if(get_left(b)!=NULL)
              bst_insert(b->left,n);
      else {  b->left=n;
              n->parent=b;}
else
      if(get_right(b)!=NULL)
              bst_insert(b->right,n);
      else {  b->right=n;
              n->parent=b;}
}
```

Scansione di un BST

- Immaginiamo di voler *stampare* il contenuto di un BST in ordine crescente di chiave
- … o più in generale di voler *accedere* a tutti i nodi di un BST in ordine crescente di chiave
- Dobbiamo implementare un algoritmo di visita di un BST
- Qual è l'algoritmo di visita che accede ai nodi in ordine crescente di chiave?

DFS inordine

DFS inorder: funzionamento

Si visita prima la parte sinistra dell'albero, il nodo padre e poi la parte destra dell'albero

Algoritmo invisitaDFS-BST (nodo n)

1. If (get_left(n)!=NULL)

   invisitaDFS-BST(get_left(n))
2. Visita nodo n
3. If (get_right(n)!=NULL)

   invisitaDFS-BST(get_right(n))

Esercizio

- Estendere il modulo «bst» dell'esercizio precedente aggiungendo la primitiva `print_key` per la stampa del valore della chiave
- Estendere il modulo «main» dell'esercizio precedente aggiungendo la funzione `void print_BST(bst b)` che stampa il contenuto dell'albero dato in ordine crescente di chiave
- Richiamare la funzione dal `main`

SOLUZIONE Vedi cartella *bst_print*

Ricerca di un elemento

`bnode* bst_search(bst,tipo_key):`

Tipica operazione su un albero binario di ricerca: ***ricerca di una chiave k***

Dati una chiave e un BST, la primitiva restituisce:

- un puntatore a un nodo con chiave data, se esiste
- il valore **NULL** altrimenti

Ricerca: funzionamento

Simile all'inserimento: scansione dell'albero a partire dalla radice verso il basso

il *percorso di scansione dipende dall'esito del confronto* tra la chiave del nodo corrente `z` e la chiave da cercare `k`: ***key(z)*** e ***k***

- Se ***k* = *key(z)*** restituisco `z`
- Se ***k* < *key(z)*** la scansione prosegue nel sottoalbero sinistro
- Se ***key(z)* < *k*** la scansione prosegue nel sottoalbero di destra

Esercizio

- Estendere il modulo «bst» dell'esercizio precedente aggiungendo la funzione `bst_search` per la ricerca di un elemento
- Estendere il `main` consentendo all'utente di cercare valori di chiave fino a quando non digita un carattere speciale. Ad ogni ricerca, richiamare la funzione `bst_search`

SOLUZIONE Vedi directory *bst_search*

Cancellazione di un elemento

```
void bst_delete(bst& t, bnode* n)
```

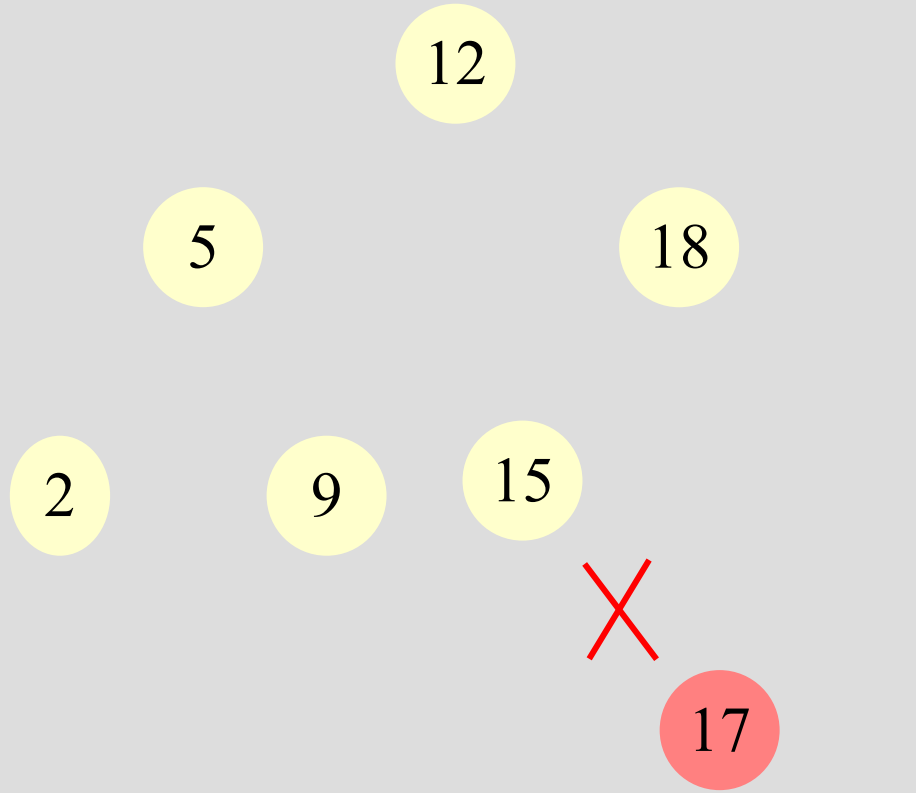
- La cancellazione di un nodo dell'albero è l'operazione più complessa
- Ogni cancellazione richiede di eliminare un nodo dall'albero, *mantenendo*
  1. *La struttura ad albero binario*
  2. *La proprietà di ricerca*

3 casi possibili (in ordine di complessità crescente):

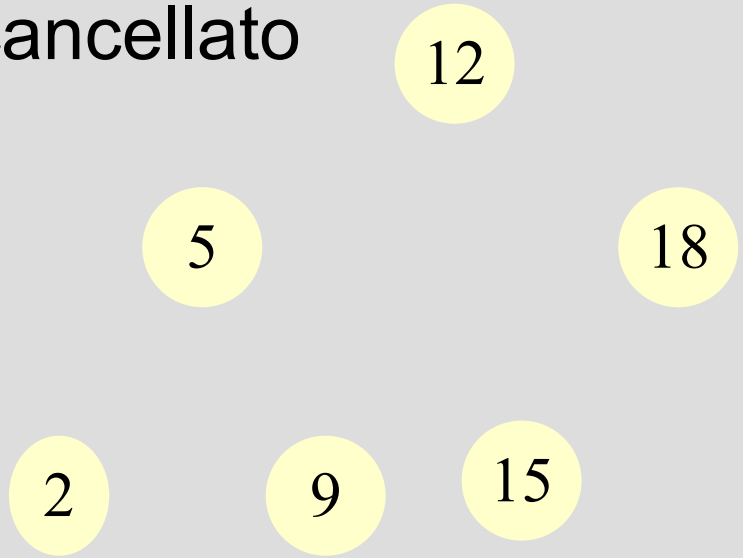
1. Il nodo `n` è una foglia
2. Il nodo `n` ha un solo figlio
3. Il nodo `n` ha due figli

Caso 1: Nodo foglia

Eliminazione del nodo 17

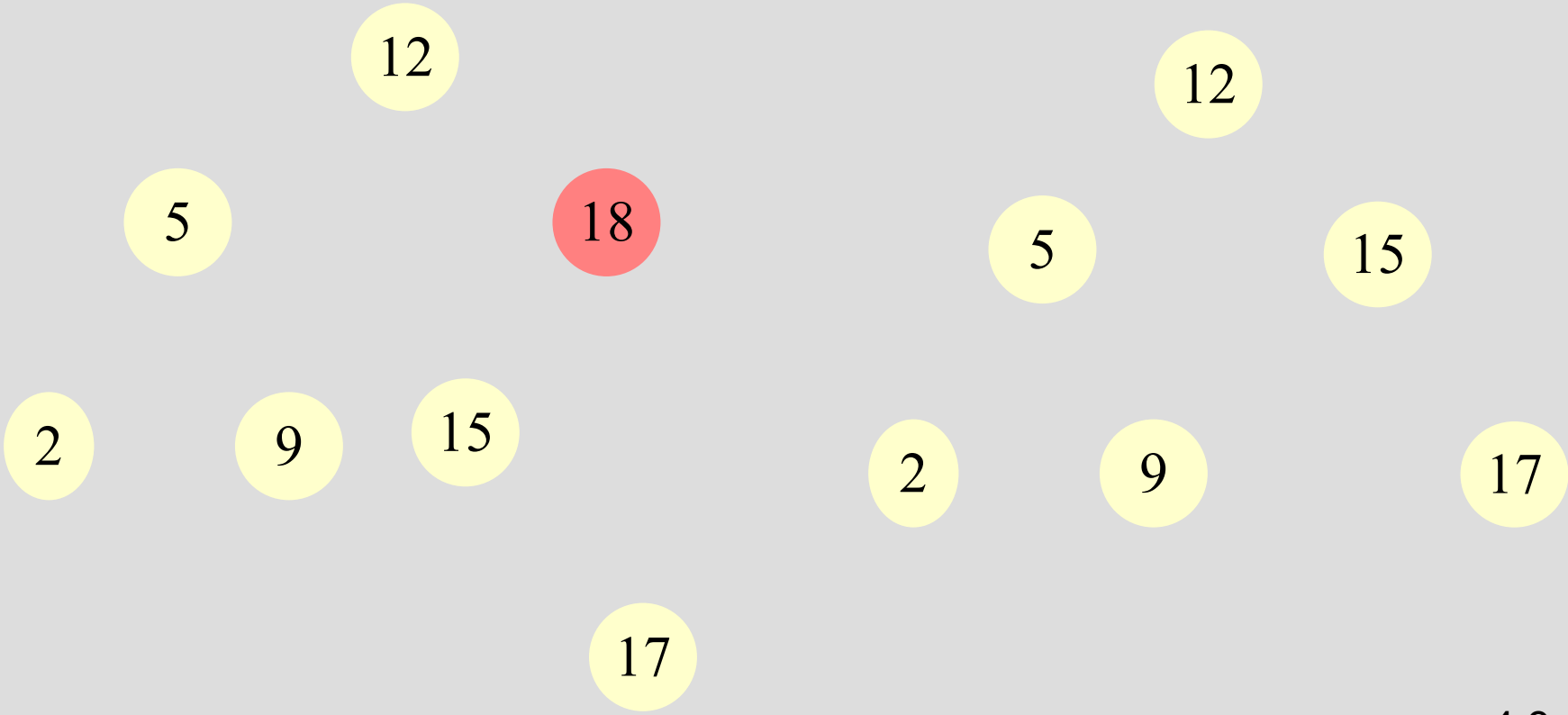


Il nodo viene semplicemente cancellato



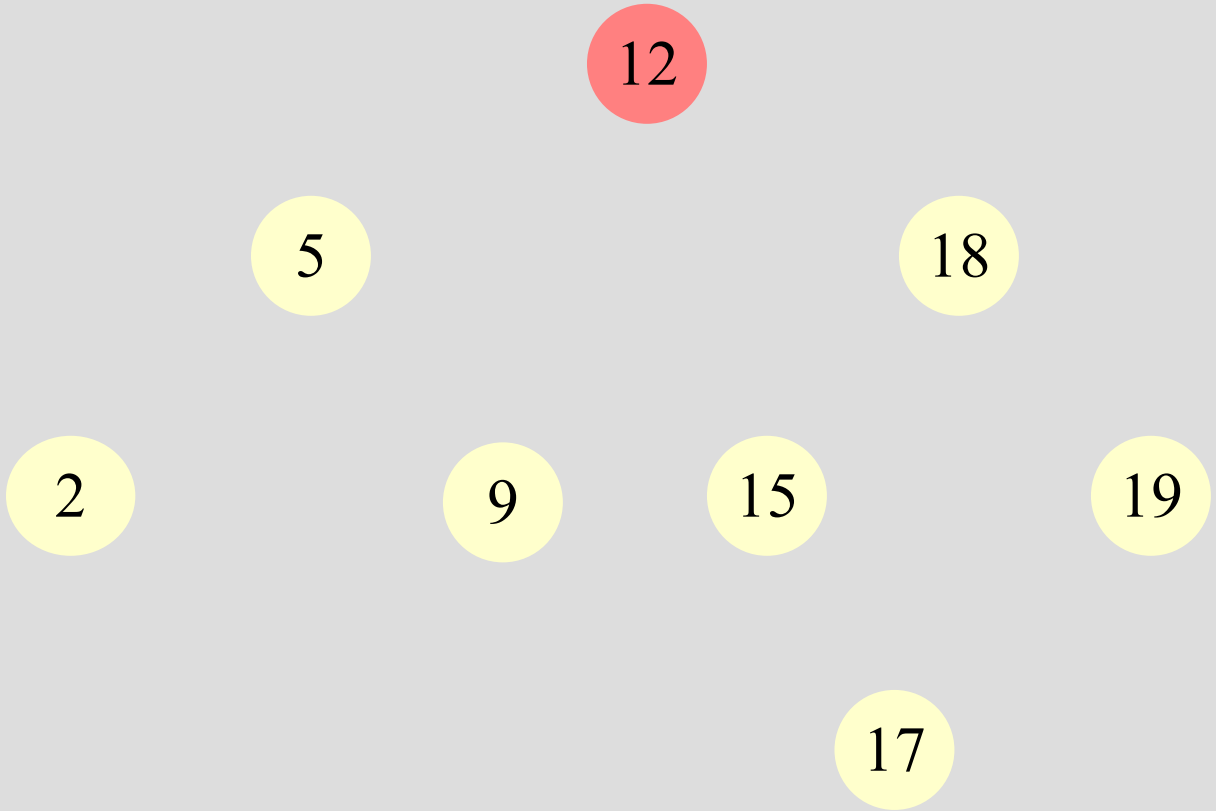
Caso 2: Nodo con un solo figlio

Il nodo `n` viene cancellato creando un collegamento tra il padre e il figlio di `n`

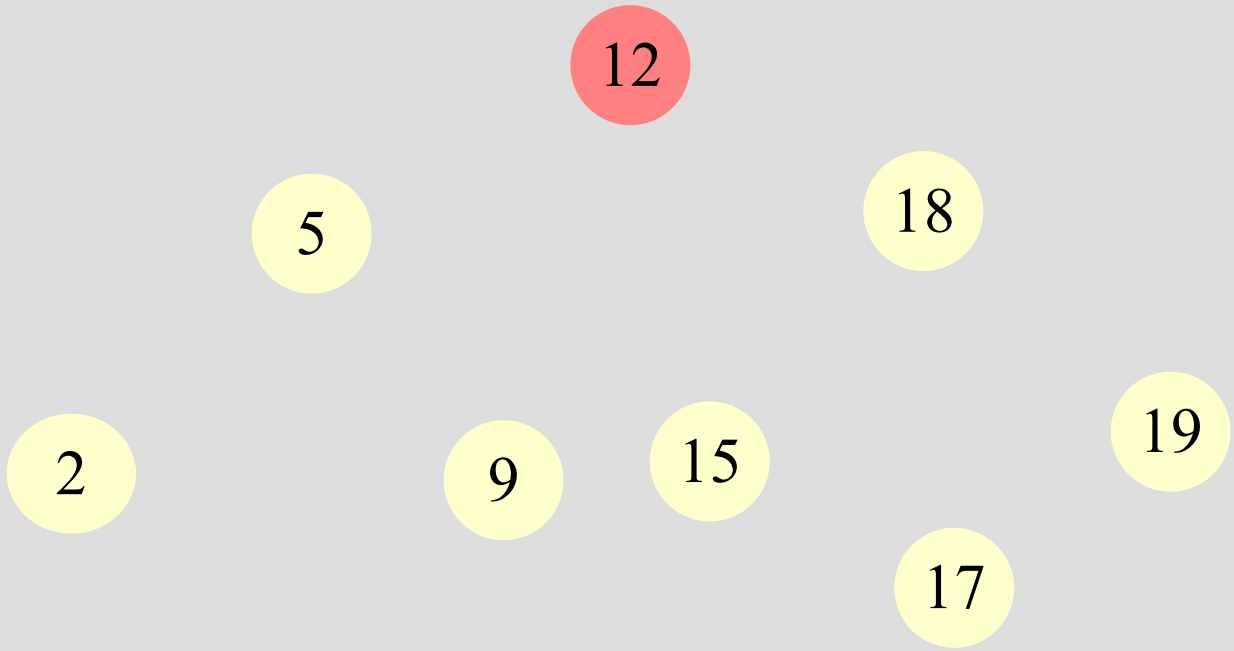


Caso 3: nodo con due figli

Eliminazione del valore 12



Caso 3: Nodo con due figli



- Cerchiamo il ***minore dei suoi successori*** (o equivalentemente il ***maggiore dei suoi predecessori***) per **sostituirlo**
- Il minore dei successori di `n` è il nodo più a sinistra del sottoalbero destro di `n`

Caso 3: Nodo con due figli

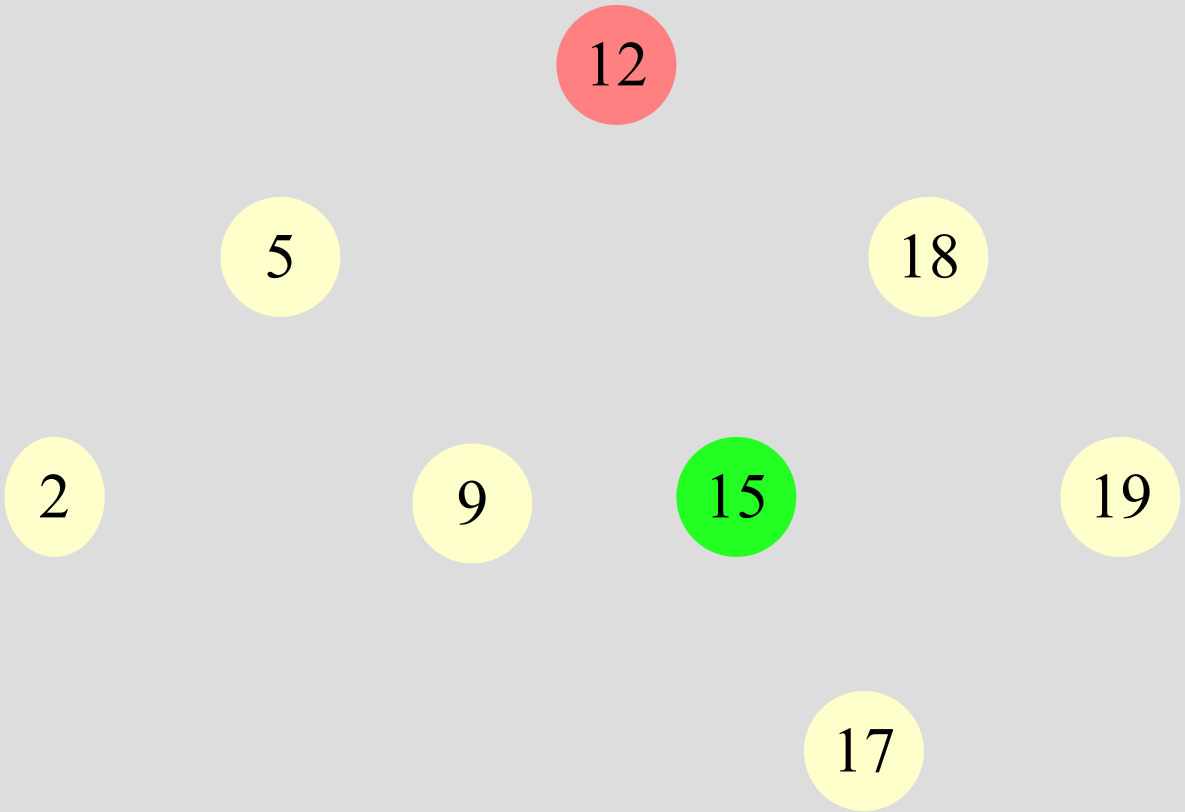
- Sicuramente `n'` il minore dei successori non ha un figlio sinistro (***ha al più un figlio destro***)
- Perché?

Perché se `n'` ***avesse un figlio sinistro*** `s-n'` ***allora***
`s-n'` ***sarebbe sicuramente minore di*** `n'`
ma maggiore di `n`***, trovandosi alla destra di*** `n`
ovvero `n<s-n'<n'`

Quindi per assurdo
`n'` ***non potrebbe essere minore dei successori di*** `n`

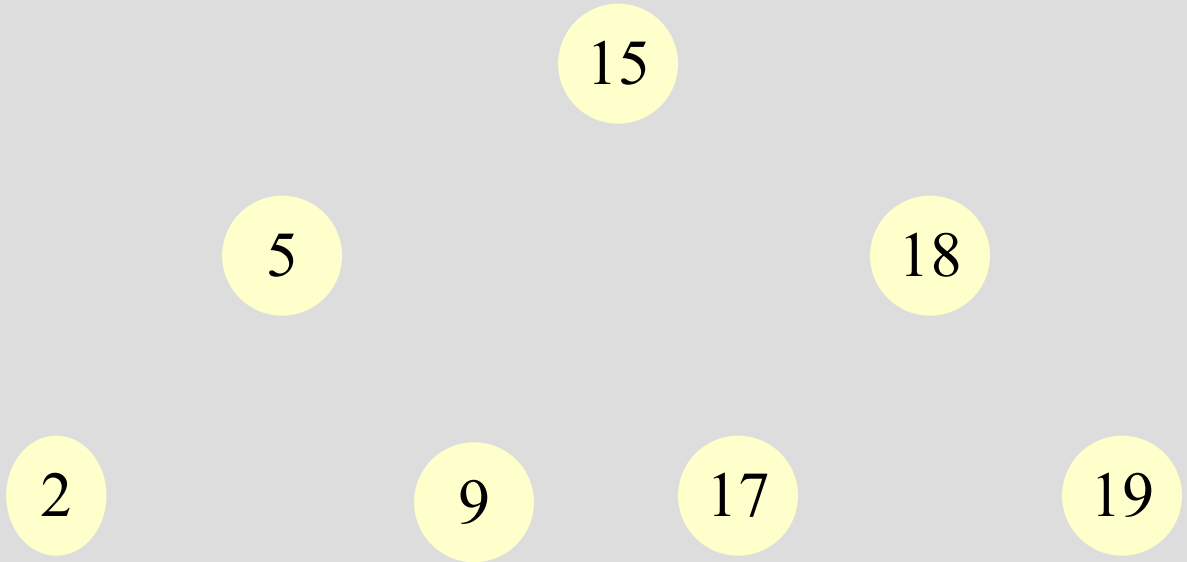
Esempio: nodo con due figli

15 è il più piccolo dei successori di z



Esempio: nodo con due figli

Situazione finale



Esercizio

- Estendere il modulo «bst» dell'esercizio precedente aggiungendo la funzione `bst_delete` per la cancellazione di un elemento
- Estendere il `main` che deve presentare un menu per consentire all'utente di aggiungere un nodo al BST con chiave data, cancellare un nodo con chiave data, cercare un nodo con chiave data e stampare l'elenco dei valori con chiave crescente.

SOLUZIONE Vedi directory *bst_delete*

Considerazioni finali

Quanto costano le primitive di ricerca di valore di chiave e aggiornamento (inserimento e cancellazione) attraverso un BST?

- Il loro costo proporzionale è alla profondità dell'albero
- Lo stesso vale per altre *operazioni tipiche di insiemi ordinati* come la ricerca del valore massimo e minimo di chiave, del predecessore o del successore di un valore di chiave

Qual è la profondità di un BST?

- Dipende dall'ordine di inserimento cancellazione delle chiavi
- In alcuni casi può risultare una profondità pari al numero di chiavi inserite
- In altri casi la profondità può essere ottimale: log(n) con n numero di chiavi

Alberi AVL

Esiste un albero di ricerca binario **bilanciato in altezza**:

> Un albero è bilanciato in altezza se le altezze dei sottoalberi sinistro e destro di ogni nodo differiscono al più di una unità

Questo albero si chiama **albero AVL**

Poiché inserimenti e cancellazioni potrebbero far perdere il bilanciamento, gli alberi AVL prevedono dei meccanismi di bilanciamento tramite rotazioni