ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
DIPARTIMENTO DI
INFORMATICA - SCIENZA E INGEGNERIA

LE STRUTTURE DATI DINAMICHE: GLI ALBERI

COSIMO LANEVE

COSIMO.LANEVE@UNIBO.IT

CORSO 00819 – PROGRAMMAZIONE

ARGOMENTI [SAVITCH, PP 772-773]

1. definizione di alberi e nozioni relative
2. implementazione degli alberi, creazione, visita
3. algoritmo di visita iterativa e sua implementazione
4. esempi/esercizi
5. alberi binari di ricerca
    a. la ricerca di elementi e la complessità computazionale
    b. operazione di cancellazione di elementi
    c. esempi/esercizi

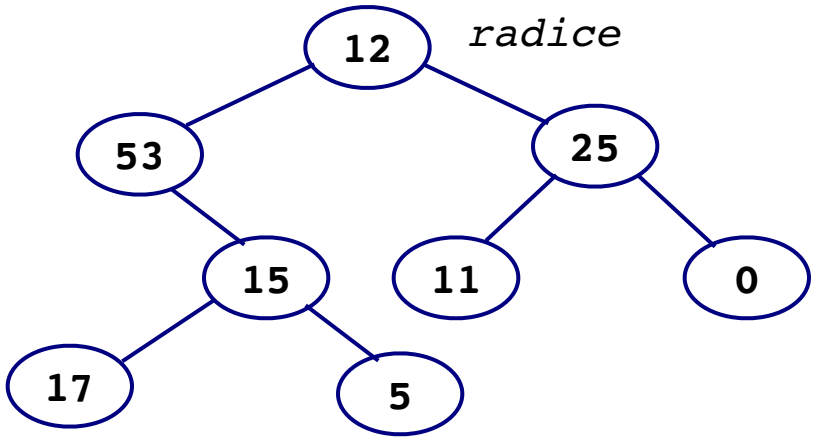
DEFINIZIONE

un **albero** è una collezione di nodi e di archi, per cui

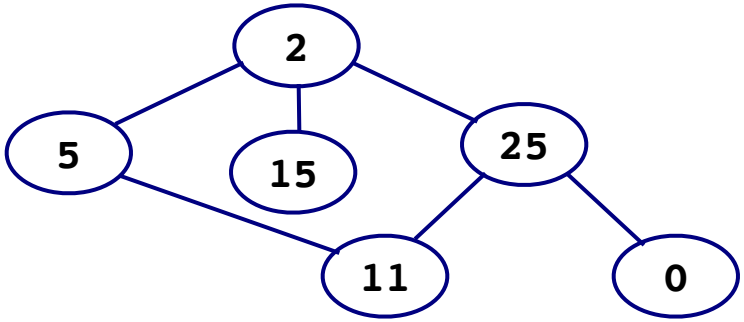
- ogni nodo, eccetto uno, detto **radice**, *ha un solo predecessore* e *0 o più successori*, la radice non ha predecessori
- *esiste* **un unico cammino dalla radice ad ogni altro nodo**
- ogni nodo contiene un *valore* di un qualche tipo

esempio:

un albero di interi

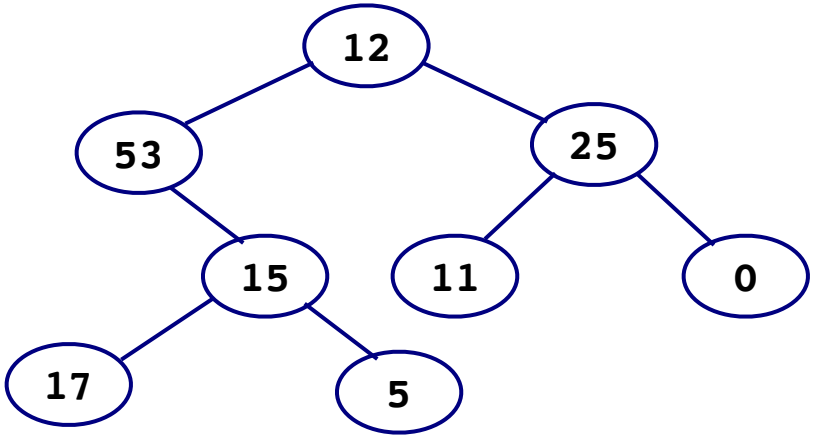


una struttura che **non è** un albero



FOGLIE, PADRI, FRATELLI

- un nodo che non ha successori è detto **foglia**
- **padre di un nodo `n`**: è il nodo immediatamente prima di `n` nel cammino verso la radice
- **antenato di un nodo `n`**: ogni nodo che si trova nel cammino di `n` verso la radice (**successore** è la relazione inverso)
- due nodi sono **fratelli** se il loro padre è lo stesso
- **profondità di un nodo**: è la lunghezza del cammino redice-nodo

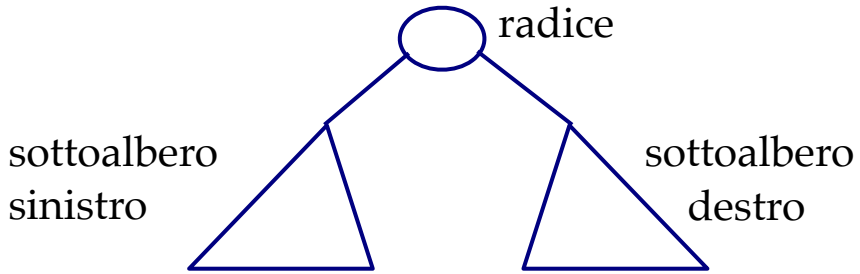


- `17` e `5` sono foglie
- il padre di `53` è `12`
- i nodi `53` e `25` sono fratelli
- `53` è antenato di `5`
- `12` ha profondità 0 ; `5` ha profondità 3

ALBERI BINARI/DEFINIZIONE

un albero binario con valori di tipo `T` è:

- l'albero vuoto (senza nodi)
- oppure è costituito da un nodo, detto *radice*, che contiene un valore di tipo `T` e da due sottoalberi binari disgiunti, chiamati **sottoalbero sinistro** e **sottoalbero destro**

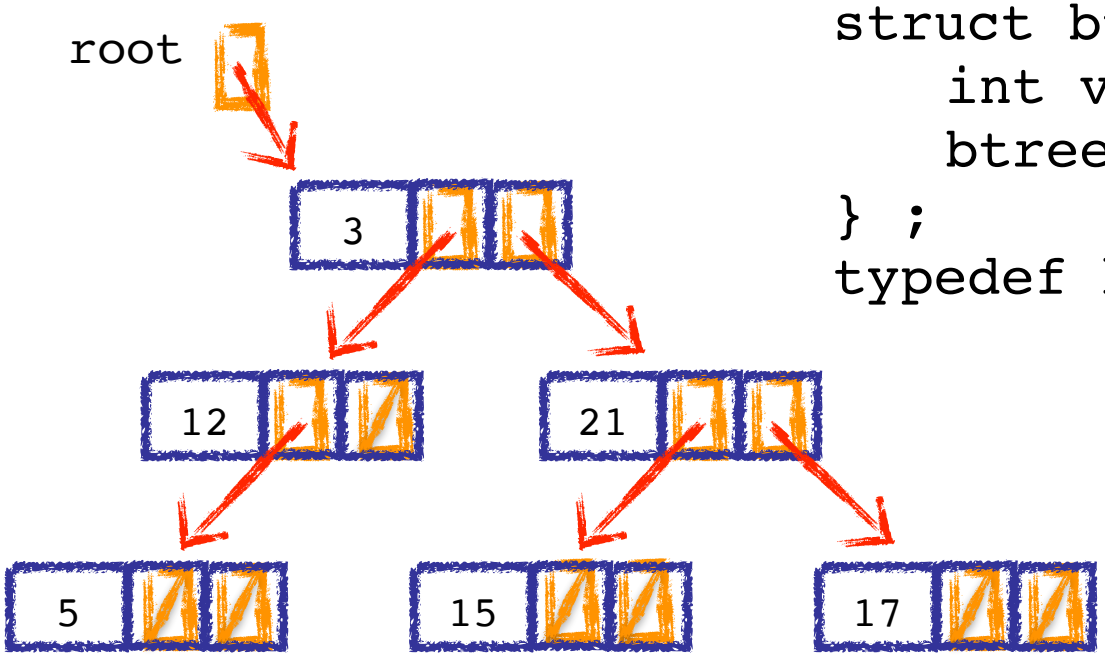


osservazione: la definizione di albero è "**ricorsiva**"

ALBERI BINARI/IMPLEMENTAZIONE

si usano strutture con tre campi

- uno contiene il valore
- gli altri due contengono i puntatori ai sottoalberi sinistro e destro




```
struct btree {
    int val;
    btree *ltree, *rtree ;
} ;
typedef btree *ptr_btree ;
```

ALBERI BINARI/COSTRUZIONE

costruire un albero binario `t` di interi (presi da una lista)

- all'inizio l'albero `t` è vuoto (`t` = `NULL`)
- ogni volta che bisogna inserire un intero, l'algoritmo in modo casuale lo inserisce nel sottoalbero sinistro o destro
- l'algoritmo termina quando a sinistra o a destra del nodo visitato non c'è niente, e crea un nuovo nodo che contiene il valore

ALBERI BINARI/COSTRUZIONE

```
ptr_btree create_btree(ptr_btree t, int n) {
   int x ;
   if (t == NULL) { t = new btree ;
                    t->val = n ;
                    t->ltree = NULL ;
                    t->rtree = NULL ;
   } else { x = rand()%2 ;
            if (x==0)
                  t->ltree = create_btree(t->ltree, n);
            else t->rtree = create_btree(t->rtree, n);
   }
   return(t) ;
}
```

esercizio: dare la definizione iterativa

ALBERI BINARI/VISITA

stampare i valori memorizzati nell'albero

- stampa il valore memorizzato nella radice e poi **stampa i valori nei sottoalberi sinistro** e **destro**
- "**stampa i valori nel sottoalbero sinistro**" e "**stampa i valori nel sottoalbero destro**" sono chiamate ricorsive

```
void visit(ptr_btree t){
   if (t != NULL) { cout << t->val ;
                    visit(t->ltree) ;
                    visit(t->rtree) ; }
}
```

ALBERI BINARI/VISITA

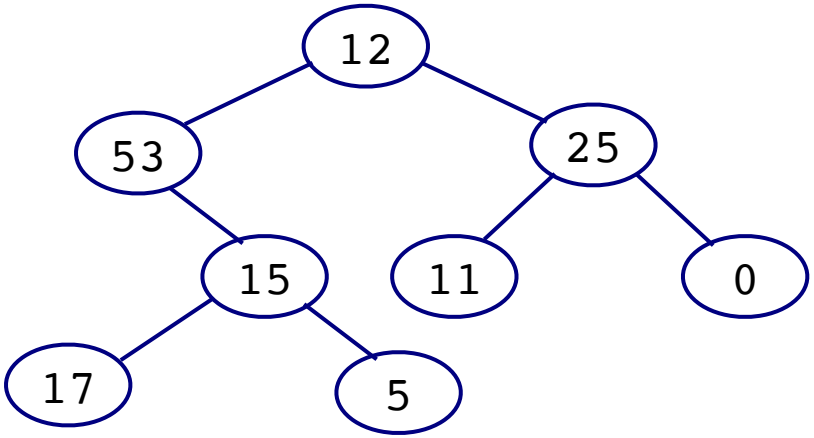
questa visita si chiama **prefissa**: altri tipi di visita sono possibili, modificando il corpo dell' `if`

- **visita infissa** : viene stampato il sottoalbero di sinistra, poi il valore del nodo e infine il sottoalbero di destra

- **visita postfissa** : viene stampato il sottoalbero di sinistra, poi il sottoalbero di desta e infine il valore della radice

ALBERI BINARI/VISITA/ESEMPI

nell'albero



la **visita prefissa** produce:

12 53 15 17 5 25 11 0

la **visita infissa** produce:

53 17 15 5 12 11 25 0

la **visita postfissa** produce

17 5 15 53 11 0 25 12

ALBERI BINARI: ESERCIZI

1. creare un albero binario di interi positivi con la seguente forma
   - la radice ha valore n, preso in input
   - il figlio sinistro ha valore n/2 (se il risultato è 0 il figlio non esiste)
   - il figlio destro ha valore n-2 (se il risultato è 0 o negativo il figlio non esiste)

2. scrivere una funzione che stampa i valori dei nodi a profondità 3

3. creare un albero di interi che rappresenti la struttura dei risultati parziali del calcolo di fib(n)

ALBERI BINARI/VISITA ITERATIVA

scrivere la visita prefissa in maniera iterativa

tentativo 1:

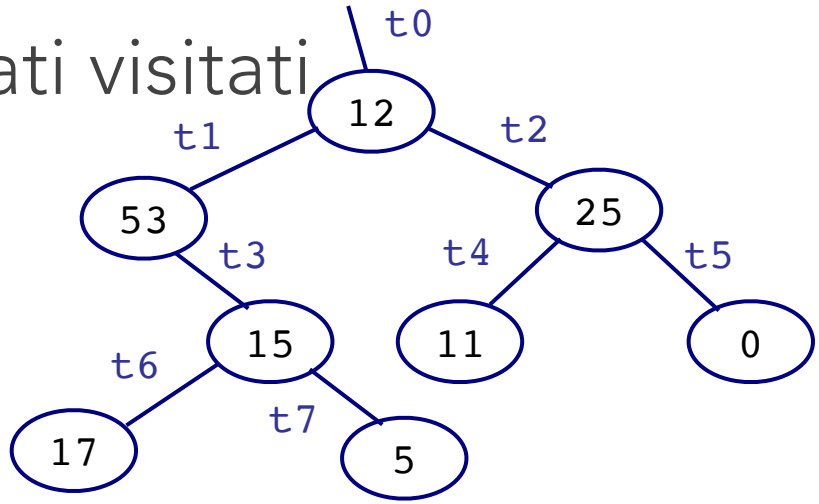
```
void visit_it(ptr_btree t){
                while (t != NULL) { cout << t->val ;
                                    t = t->ltree ; }
        }
// viene stampato solamente il cammino di sx
```

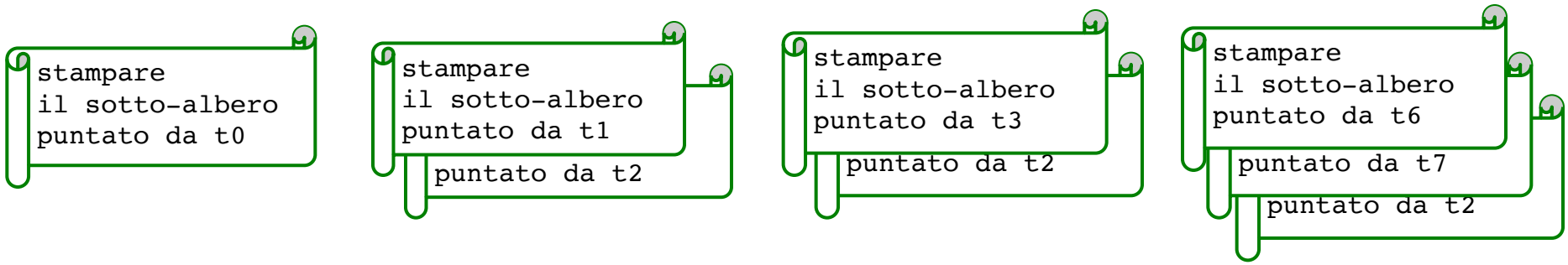
tentativo 2:

```
void visit_it(ptr_btree t){
          btree *s = t ;
          while (t != NULL) {
              cout << t->val ; t = t->ltree ; }
          if (s != NULL) s = s->rtree ;
          while (s != NULL) {
              cout << s->val ; s = s->rtree ; }
        }
// vengono stampati solamente i cammino di sx e di dx
```

ALBERI BINARI/VISITA ITERATIVA/CONT.

idea : bisogna tener traccia di tutti i sottoalberi che non sono stati visitati






```
stampa: 12   53   15
```

osservazione: pile di obbligazioni "stampare il sotto-albero…"

ALBERI BINARI/VISITA ITERATIVA/IMPLEMENTAZIONE

```
struct t_stack {
        ptr_btree val ;
        t_stack *next ;
    } ;
typedef t_stack *ptrt_stack ;

ptrt_stack push(ptrt_stack q, ptr_btree t){
        ptrt_stack tmp = new t_stack ;
        tmp->val = t ;
        tmp->next = q ;
        return(tmp) ;
}
ptrt_stack pop(ptrt_stack q) {
        if (q == NULL) return(NULL) ;
        else { ptrt_stack tmp = q ;
            q = q->next ;
            delete(tmp) ;
            return(q) ; }
}
ptr_btree top(ptrt_stack q) {
        if (q == NULL) return(NULL) ;
        else return(q->val);
}
```

ALBERI BINARI/VISITA ITERATIVA/IMPLEMENTAZIONE/CONT.

```
void visit_it(ptr_btree t){
  if (t != NULL){
          ptr_btree tmp ;
          ptrt_stack p ;
          p = push(NULL, t) ;
          while (p != NULL) {
                  tmp = top(p) ;
                  p = pop(p) ;
                  cout << tmp->val ;
                  if (tmp->rtree != NULL) p = push(p, tmp->rtree) ;
                  if (tmp->ltree != NULL) p = push(p, tmp->ltree) ;
                  }
          }
 }
```

osservazione: questa tecnica mostra come viene implementata la ricorsione da parte dei compilatori/supporto run-time

- lo *stato del calcolo* (cioè la parte "devo stampare il sotto-albero…") viene salvato su una opportuna pila prima della invocazione ricorsiva

ESERCIZIO

scrivere una funzione C++ che prende in input un albero binario e restituisce la profondità massima (la lunghezza del cammino più lungo radice-foglia)

```
int max_depth(btree *t){
    if (t == NULL) return(0) ;
    else if ((t->ltree == NULL) && (t->rtree == NULL)) return(0) ;
    else return(1 + max(max_depth(t->ltree), max_depth(t->rtree))) ;
}
```

ESERCIZI

1. scrivere una funzione ricorsiva ed una iterativa che prende in input un albero binario e restituisce il numero di nodi in esso

2. supponiamo di sapere che un albero può avere al massimo `n` nodi. Scrivere le funzioni di "creazione di un albero", "visita ricorsiva" e "visita iterativa" utilizzando gli array

3. implementare le visite infisse e postfisse in maniera iterativa

RICERCA DI VALORI NELL'ALBERO

problema: definire una funzione che prende un albero binario e un valore e restituisce **`true`** o **`false`** a seconda che il valore sia presente o meno nell'albero

```
bool search(ptr_btree t, int n){
      if (t == NULL) return(false) ;
      else if (t->val == n) return(true) ;
      else return(search(t->ltree,n) || search(t->rtree,n)) ;
}
```

RICERCA DI VALORI NELL'ALBERO

questione: quanti passi di calcolo esegue **`search`**? (quante chiamate ricorsive ci sono?)

- se trova subito il valore non c'è alcuna chiamata ricorsiva
- se il valore non lo trova, oppure si trova in una foglia, ci sono **`k`** chiamate ricorsive, dove **`k`** è il numero di nodi dell'albero

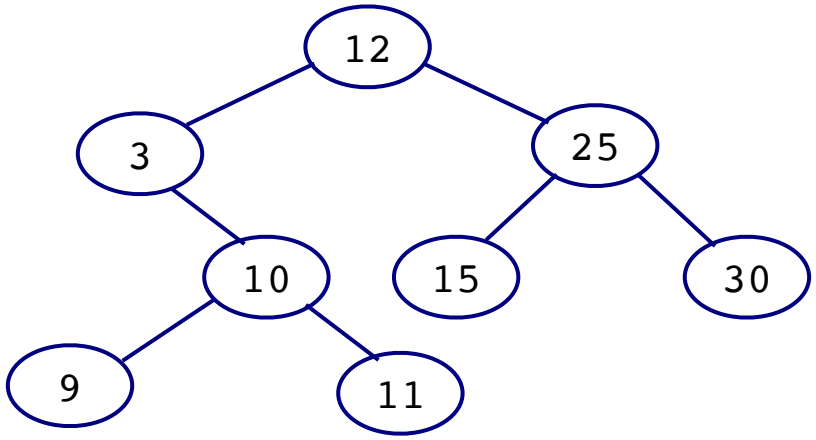
risposta: al peggio esegue un numero di passi proporzionale al numero di nodi nell'albero

esercizio: scrivere "**`search`**" in maniera iterativa

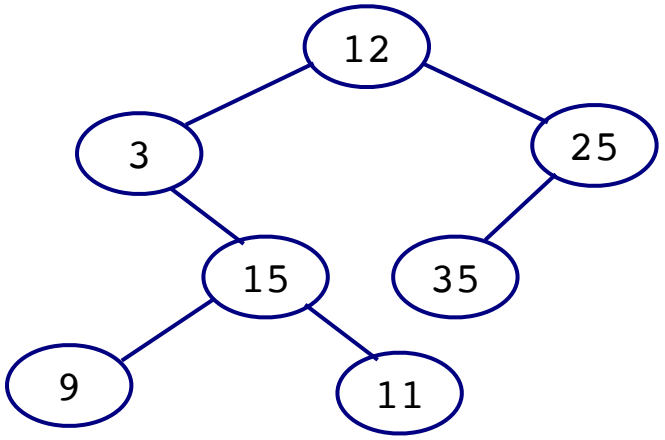
ALBERI BINARI DI RICERCA

gli **alberi binari di ricerca** sono

- alberi binari
- i valori dei nodi dell'albero sono a due a due diversi
- il valore di ogni nodo è:
  - maggiore di tutti i valori nel sottoalbero sinistro
  - minore di tutti i valori nel sottoalbero destro



un albero binario di ricerca



un albero binario *non* di ricerca

ALBERI BINARI RICERCA/SEARCH

numero di chiamate ricorsive
=
profondità massima dell'albero

versione ricorsiva di **search**:

```
bool search(ptr_btree t, int n){
      if (t == NULL) return(false) ;
      else if (t->val == n) return(true) ;
      else if (t->val < n) return(search(t->rtree,n)) ;
      else return(search(t->ltree,n)) ;
}
```

versione iterativa di **search**:

```
bool search_it(ptr_btree t, int n){
       bool trovato = false ;
       while ((t != NULL) && !trovato){
       if (t->val == n) trovato == true ;
       else if (t->val < n) t = t->rtree ;
       else t = t->ltree ;
       }
      return(trovato) ;
}
```

numero di iterazioni
=
profondità massima dell'albero

ALBERI BINARI RICERCA/COMPLESSITÀ DI SEARCH

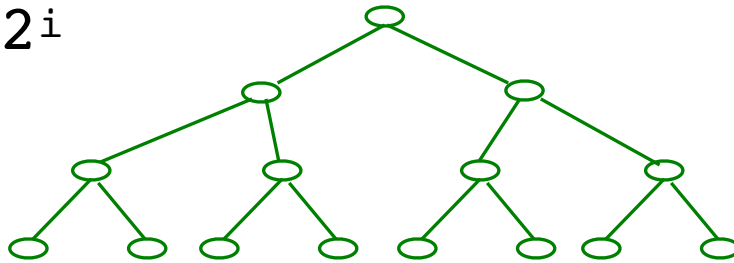
- l'albero può essere *completamente sbilanciato*, cioè avere la forma accanto

- in questo caso il cammino più lungo per raggiungere un nodo (profondità max di un nodo) ha lunghezza `n`

- l'albero può essere **completamente bilanciato**, cioè avere questa forma: *ogni nodo ha sempre 2 figli, eccetto le foglie che hanno 0 figli*

- in questo caso, a profondità `0` ci sono $2^0=1$ nodo, a profondità `1` ci sono $2^1=2$ nodi, . . ., a profondità `i` ci sono 2^i nodi

- se l'albero ha `n` nodi, allora $n = \sum_{i=0}^{k} 2^i$
  - `k` è `log(n+1) - 1`



ALBERI BINARI DI RICERCA/ESERCIZI

1. scrivere una funzione iterativa che prende in input un albero binario di ricerca e stampa l'elemento massimo memorizzato

2. scrivere una funzione che prende in input un albero binario di ricerca e stampa gli elementi memorizzati in maniera crescente

3. scrivere una funzione che prende in input un albero binario di ricerca e stampa gli elementi memorizzati in maniera decrescente

ALBERI BINARI DI RICERCA/OPERAZIONI

algoritmo di costruzione di un albero binario di ricerca:

1. se **l'albero è vuoto** allora inserisci il nuovo dato come radice

2. se **l'albero non è vuoto e se il valore del nuovo dato è uguale** al valore della radice, il nuovo dato non viene inserito

3. se il valore del nuovo dato **è più piccolo/grande** del valore della radice inserisci il nuovo dato **nel sottoalbero sinistro/destro**

ALBERI BINARI DI RICERCA/OPERAZIONI

```
ptr_btree t_insert(ptr_btree t, int n) {
    if (t == NULL) { t = new btree ;
                     t->val = n ;
                     t->ltree = NULL ;
                     t->rtree = NULL ;
    } else if (t->val == n) return(t) ;
      else if (t->val > n) t->ltree = t_insert(t->ltree, n) ;
      else t->rtree = t_insert(t->rtree, n) ;
   return(t) ;
}
```

osservazione: se l'albero è completamente bilanciato, l'inserimento ha costo logaritmico nel numero di nodi (in termini di iterazioni o chiamate ricorsive); se l'albero è sbilanciato, l'inserimento ha costo lineare