Alberi ed Alberi Binari di Ricerca

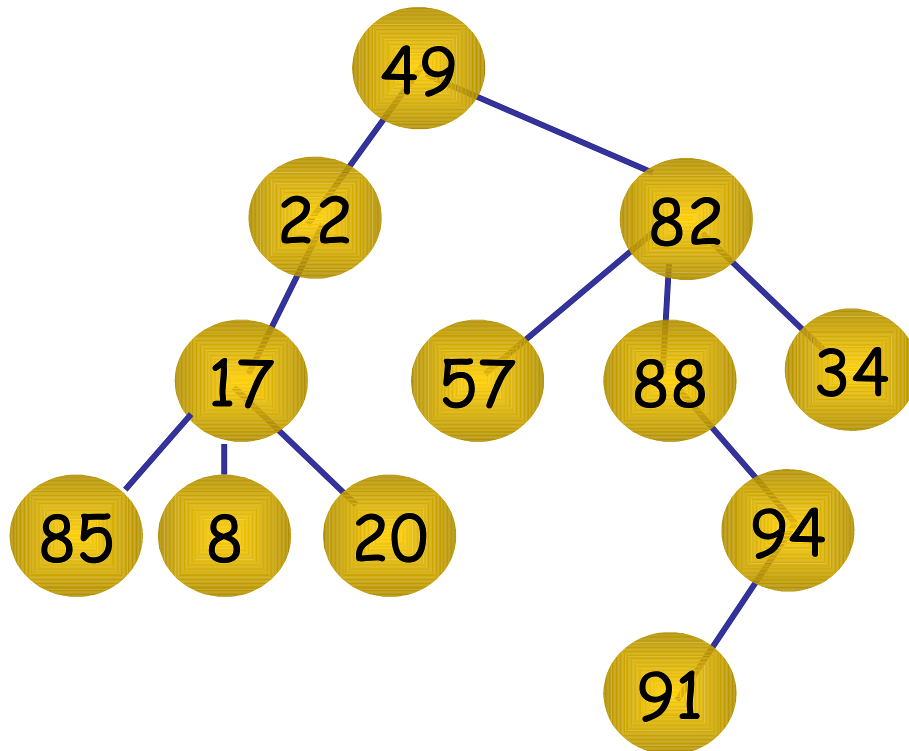
Il tipo di dato Albero

- Un **albero** è una struttura di data organizzata gerarchicamente.
- È costituito da un insieme di **nodi** collegati tra di loro:
  - ogni nodo contiene dell'informazione
  - i nodi collegati ad un certo nodo *n* vengono detti **figli** di *n*
  - i nodi che non hanno figli sono detti **foglie** dell'albero
  - c' è un unico nodo che non è figlio di alcun altro nodo, detto **radice** dell'albero
    (scendendo a partire dalla radice si raggiungono tutti i nodi dell'albero)

- **Albero vuoto**: non ha nessun nodo (nemmeno la radice)

Il tipo di dato Albero

- *Esempio*:
  - Albero in cui le etichette sono degli interi



La radice ha etichetta 49

I figli della radice hanno etichette 22 e 82

Le foglie hanno etichette 85, 8, 20, 57, 91, 34

Alberi: alcune definizioni

- **Cammino** in un albero:
  sequenza di nodi, in cui ogni nodo è figlio del nodo che lo precede nella sequenza

- **Livello** (o **profondità**) di un nodo è la sua distanza dalla radice (quanto "in basso" si trova nell'albero).
  - Definizione formale (induttiva) di livello di un nodo:
    - la radice ha livello ***0***
    - se un nodo ha livello ***i***, allora i suoi figli hanno livello ***i + 1***

- Il **livello i** di un albero è formato da tutti i nodi a livello ***i***.

Alberi binari

- Un **albero binario** è un albero in cui ogni nodo ha **al massimo 2 figli**
  - i due figli vengono detti **figlio sinistro** e **figlio destro**
  - se consideriamo la parte di albero costituita dal figlio sinistro, dai suoi figli, dai figli dei suoi figli, . . . , questa è di nuovo un albero binario, detto **sottoalbero sinistro** (analogamente si definisce il **sottoalbero destro**)

Rappresentazione collegata degli alberi binari

```
struct nodo {
  <contenuto>
  struct nodo *dx, *sx;
}
typedef struct nodo Ttree;
```

NB: sx e' il puntatore al figlio sinistro

dx e' il puntatore al figlio destro

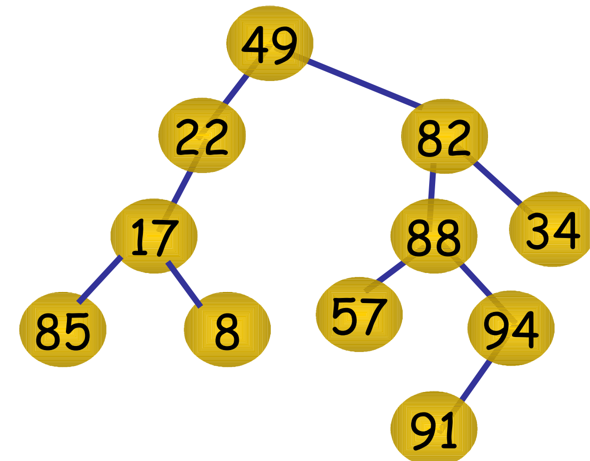
<contenuto> varia di caso in caso. Assumeremo <contenuto> = int num;

Visita di un albero

- E' l'analisi in sequenza di tutti i nodi dell'albero.
- La maggior parte delle operazioni sugli alberi possono essere considerate delle varianti di visite.
- Diversi **tipi di visita**, che differiscono per l'ordine in cui vengono visitati i nodi.
  - visita in **preordine**: si analizza la radice, poi si visita in preordine il sottoalbero sinistro, poi il sottoalbero destro
  - visita in **postordine**: si visita in postordine il sottoalbero sinistro, poi il sottoalbero destro, infine si analizza la radice
  - visita in ordine **simmetrico** (inorder): si visita in ordine simmetrico il sottoalbero sinistro, poi si analizza la radice, infine si visita in ordine simmetrico il sottoalbero destro
  - visita **per livelli**: si visita prima la radice (livello 0), poi tutti i nodi a livello 1, poi tutti i nodi a livello 2, . . .

Visita di un albero: esempio

- Preordine
  - 49, 22, 17, 85, 8, 82, 88, 57, 94, 91, 34
- Postordine
  - 85, 8, 17, 22, 57, 91, 94, 88, 34, 82, 49
- Simmetrica
  - 85, 17, 8, 22, 49, 57, 91, 94, 88, 82, 34
- A livelli
  - 49, 22, 82, 17, 88, 34, 85, 8, 57, 94, 91



Implementazione delle visite

- Supponiamo di aver definito una funzione Analizza, a cui passiamo il puntatore ad un nodo, la quale effettua l'operazione richiesta dalla visita (assumiamo che sia semplicemente la stampa del contenuto).

- Le visite in preordine, postordine, simmetrica hanno una implementazione banale se si adotta un algoritmo di tipo ricorsivo

- La visita per livelli presenta un problema:
  - dopo aver visitato un nodo, dobbiamo visitare i suoi fratelli, ma dobbiamo anche ricordarci i suoi figli (per visitarli dopo)
  - È necessario utilizzare una struttura dati di appoggio (una coda), i cui elementi sono i nodi ancora da analizzare

Visita in preordine

```
void VisitaPreordine(Ttree *a) {
   if (a != NULL) {
                                                  /* analizza la radice */
   Analizza(a);
                                    /* analizza il sottoalbero sinistro */
   VisitaPreordine(a->sx);
                                                      /* e poi il destro */
   VisitaPreordine(a->dx);
   }
}
```

Visita in postordine

```
void VisitaPostordine(Ttree * a) {
   if (a != NULL) {
                         /* analizza il sottoalbero sinistro */
   VisitaPostordine(a->sx);
                                     /* e poi il destro */
   VisitaPostordine(a->dx);
                                  /* analizza la radice */
   Analizza(a);
   }
}
```

Visita in ordine simmetrico

```
void VisitaInordine(Ttree * a) {
   if (a != NULL) {
                                   /* analizza il sottoalbero sinistro */
   VisitaInordine(a->sx);
                                               /* analizza la radice */
   Analizza(a);
                                                  /* e poi il destro */
   VisitaInordine(a->dx);
   }
}
```

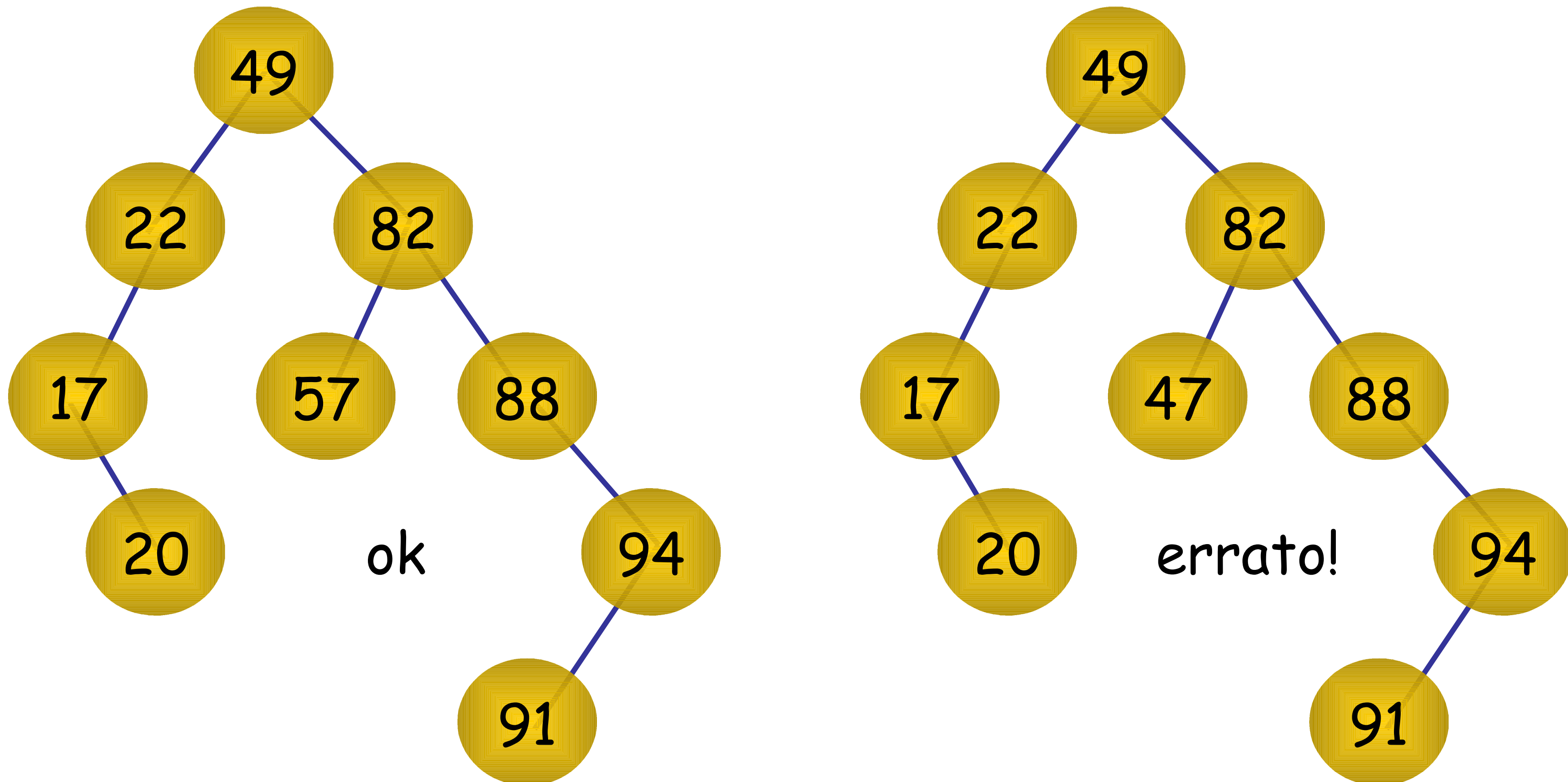
Albero binario di ricerca

- E' un albero binario che soddisfa le seguenti proprietà:
  - Ogni elemento ha una **chiave** (*su cui è definito un ordinamento*); due elementi non possono avere la stessa chiave (*cioè le chiavi sono uniche*);
  - Le chiavi in un sottoalbero sinistro non vuoto devono essere più piccole della chiave nella radice dell'albero
  - Le chiavi in un sottoalbero destro non vuoto devono essere più grandi della chiave nella radice dell'albero
  - Anche i sottoalberi sinistro e destro sono alberi di ricerca binari

Albero binario di ricerca

- In altri termini, definito un ordinamento sull'informazione dei nodi (es: interi, stringhe, ma anche strutture complesse con un campo chiave), un albero binario è un albero binario di ricerca se:
- Per ogni nodo N dell'albero:
  - L'informazione associata ad ogni nodo nel sottoalbero sinistro di N è strettamente minore dell'informazione associata ad N;
  - L'informazione associata ad ogni nodo nel sottoalbero destro di N è strettamente maggiore dell'informazione associata ad N;

Albero binario di ricerca: esempio

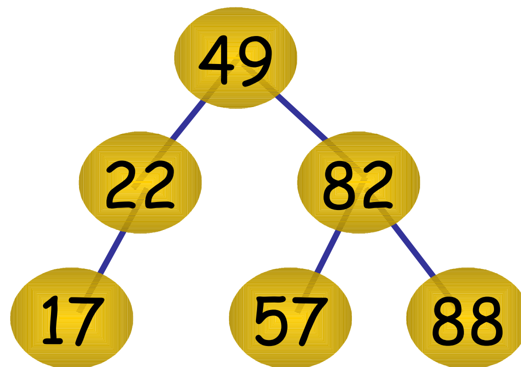


Perché si usano gli ABR?

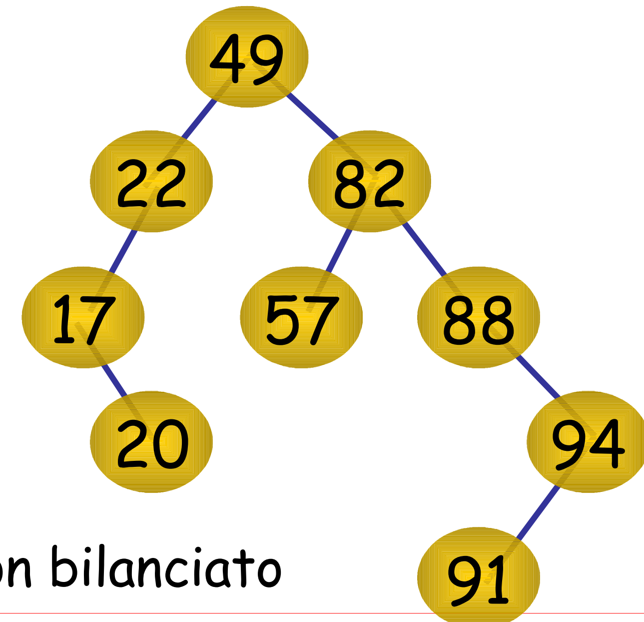
- La ricerca di un elemento in un albero binario di ricerca è una operazione molto efficiente (se l'albero è bilanciato)

Alberi bilanciati

- Un albero (binario) si dice **bilanciato** se, detta P la profondità dell'albero, tutte le foglie stanno a livello P o P-1, e tutti i nodi interni hanno due figli, tranne al più un nodo al livello P-1 che ne ha uno solo.



Bilanciato



Non bilanciato

Ricerca in un ABR

algoritmo cerca *elem* nell'albero binario di ricerca *Alb*

if *Alb* è vuoto

then *elem* non è presente in *Alb*

else if *elem* coincide con l'etichetta della radice di *Alb*

then *elem* \`e stato trovato

else if *elem* precede l'etichetta della radice di *Alb*

then cerca *elem* nel sottoalbero sinistro di *Alb*

else cerca *elem* nel sottoalbero destro di *Alb*

Ricerca in un ABR (versione iterativa)

```
Ttree * search_tree(Ttree n, Ttree * a)
{
    while (a!=NULL) {
        if (n.num == a->num)
            return a;
        else if (n.num < a->num)
            a = a->sx;
        else
            a = a->dx;
    }
    return NULL;
}
```

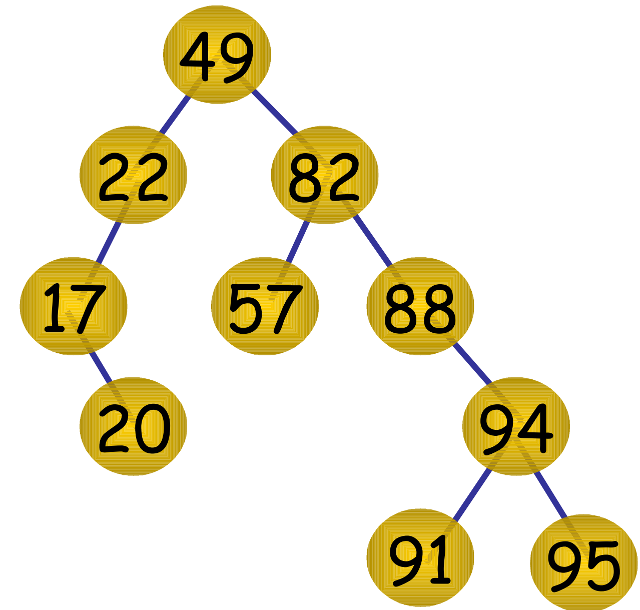
Ricerca in un ABR (versione ricorsiva)

/* Effettua la ricerca di **n** nell'albero binario di ricerca **a**. Il valore di ritorno e` il sottoalbero di a la cui radice e` pari ad **n**, se **n** e` presente in a, **NULL** altrimenti.*/

```
Ttree * recursive_search(Ttree n, Ttree * a)
{
    while (a!=NULL) {
        if (a->num == n.num)
            return a;
        else if (n.num<a->num)
             return recursive_search(n,a->sx);
        else
             return recursive_search(n,a->dx);
    }
    return NULL;
}
```

Costo della ricerca in un ABR

- ABR di n nodi
- caso peggiore
  - Albero che degenera in lista
  - O(n)
- caso medio
  - dipende dalla distribuzione
  - O(lg n) per alberi bilanciati
- caso migliore
  - O(1) (poco interessante)



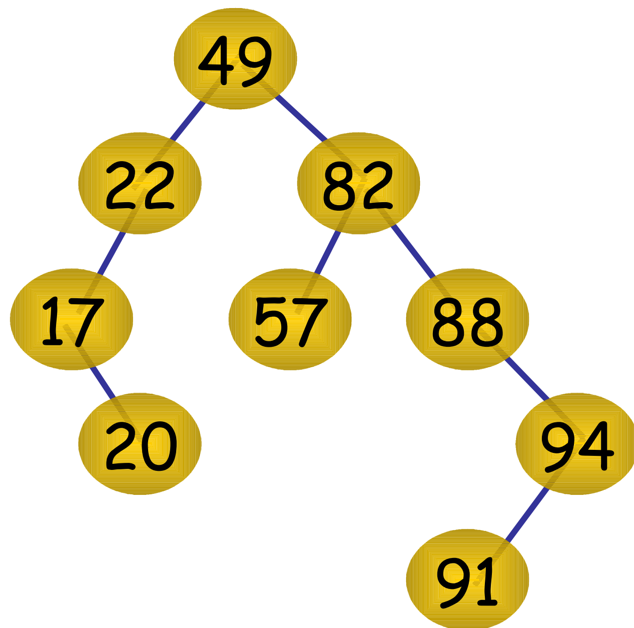
Costo della ricerca in un ABR

- nel caso di **distribuzione uniforme** delle chiavi il valore atteso dell'altezza dell'albero è O(lg n)
  - N.B. L'altezza di un albero binario di n nodi varia in $\{\lfloor \lg_2 n \rfloor + 1, \dots, n\}$

⇒ un ABR con chiavi **uniformemente distribuite** ha un costo atteso di ricerca O(lg n)

Visita in un ABR

- Di tutte le visite analizzate per un albero binario, nel caso di un ABR una riveste una particolare importanza:
  - La visita in ordine simmetrico (**inorder**) produce un elenco ordinato delle chiavi!



17, 20, 22, 49, 57, 82, 88, 91, 94

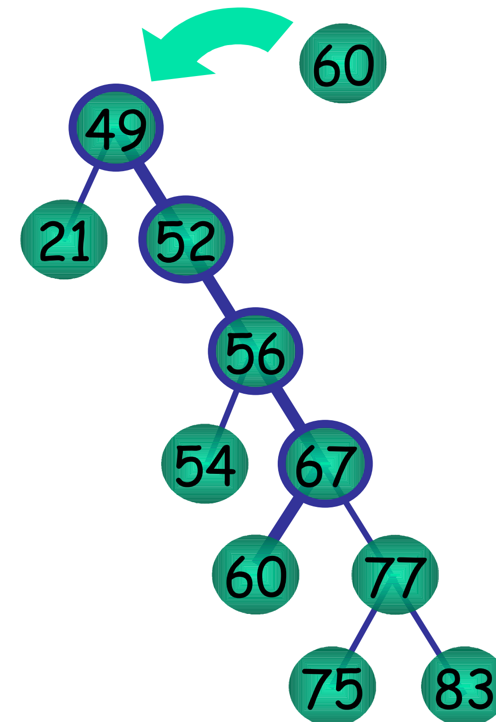
Inserimento in un ABR

Ogni nuovo nodo **u** viene inserito come foglia

- fase 1: cerca il nodo genitore **v**
- fase 2: inserisci **u** come figlio di **v**

Inserimento in un ABR

- la fase 1 termina quando si raggiunge un nodo del BST privo del figlio in cui avrebbe avuto senso continuare la ricerca
  - **non necessariamente una foglia**
  - *è il nodo inserito che diviene foglia*
- la fase 2 si limita a collegare una nuova foglia



Inserimento in un ABR (1/2)

```
Ttree * add_tree(Ttree n, Ttree * a)
{
    Ttree *nuovo, *prec;
    Ttree *current = a;

    if (a==NULL){  /* caso speciale: albero vuoto*/
        nuovo = (Ttree*)malloc(sizeof(Ttree));
        *nuovo = n;
        nuovo->sx = nuovo->dx = NULL;
        return nuovo;
    }
    while(current!=NULL){   /* FASE 1...*/
       prec = current;
       if (n.num < current->num)
           current = current->sx;
        else
           current = current->dx;
    }
CONTINUA...
```

Inserimento in un ABR (2/2)

… continua

```
    /* FASE 2*/
    nuovo = (Ttree*)malloc(sizeof(Ttree));
    *nuovo = n;
    nuovo->sx = nuovo->dx = NULL;

    if (nuovo->num < prec->num)
        prec->sx = nuovo;
    else
        prec->dx = nuovo;

    return a; /* la radice dell'albero... */
}
```

Costo dell'inserimento in un ABR

- Ogni inserimento introduce una nuova foglia
- il costo è (proporzionale a) la lunghezza del ramo radice-foglia interessato all'operazione
- nel caso peggiore: O(n)

Costo dell'inserimento in un ABR

caso peggiore

- costo fase 1: O(*n*)
- costo fase 2: O(1)
- costo totale: O(*n*)

caso medio (distrib. unif.)

- costo fase 1: O(lg *n*)
- costo fase 2: O(1)
- costo totale: O(lg *n*)