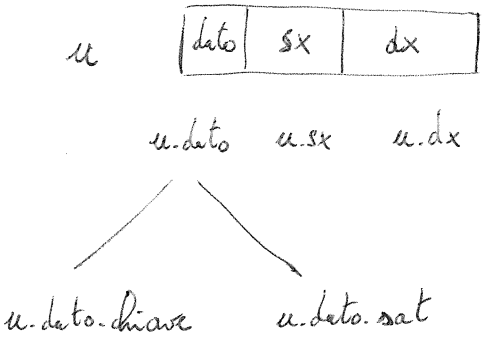
Alberi binari di ricerca

Utili per la realizzazione di dizionari ordinati.

Ricordiamo che ogni nodo è rappresentato in memoria con



Nel caso dei dizionari.

Un albero binario di ricerca è un albero binario che soddisfa la proprietà di ricerca: per ogni nodo u

- tutti gli elementi che si trovano nel sottoalbero sinistro di u, riferito da $u.sx$, sono minori dell'elemento $u.dato$ contenuto in u.
- tutti gli elementi che si trovano nel sottoalbero destro di u, riferito da $u.dx$, sono maggiori dell'elemento $u.dato$ contenuto in u.

Nell'ipotesi che gli elementi memorizzati nei nodi siano gli elementi del dizionario $S = \{e_0, e_1, \ldots, e_{n-1}\}$, una VISITA SIMMETRICA dell'albero restituisce la sequenza ordinata degli elementi di S, in tempo $O(n)$.

L'operazione di ricerca in un albero binario di ricerca, ricorda la ricerca binaria. Precisamente:

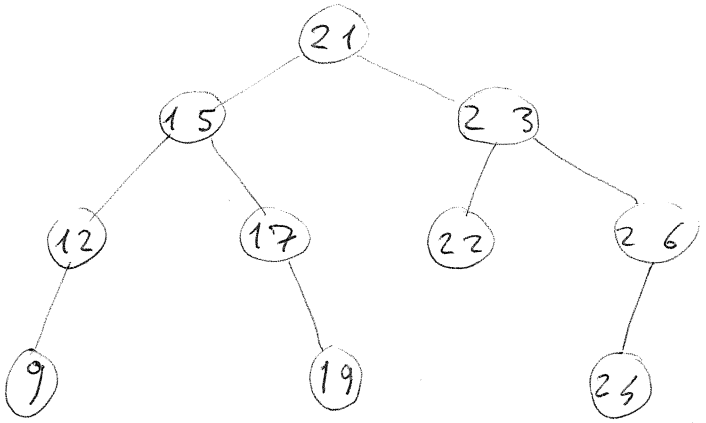
```
Ricerca (u, K):

IF (u == NULL) RETURN NULL;
IF (K == u.dato.chiave) {
    RETURN u.dato;
} ELSE IF (K < u.dato.chiave) {
    RETURN Ricerca (u.sx, K);
} ELSE {
    RETURN Ricerca (u.dx, K);
}
```

La funzione confronta la chiave K con la radice dell'albero:

- se K è minore dell'elemento contenuto nella radice la ricerca procede a sinistra (ricorsivamente)
- se K è maggiore, la ricerca procede a destra (ricorsivamente)
- se K è uguale, viene restituito l'elemento associato

Esempi. Albero binario di Ricerca



È facile verificare che ogni nodo u soddisfa la proprietà di ricerca

- i valori associati ai nodi del sottoalbero sinistro sono minori, quelli del sottalbero destro sono maggiori.

La visita in ordine simmetrico dell'albero restituisce

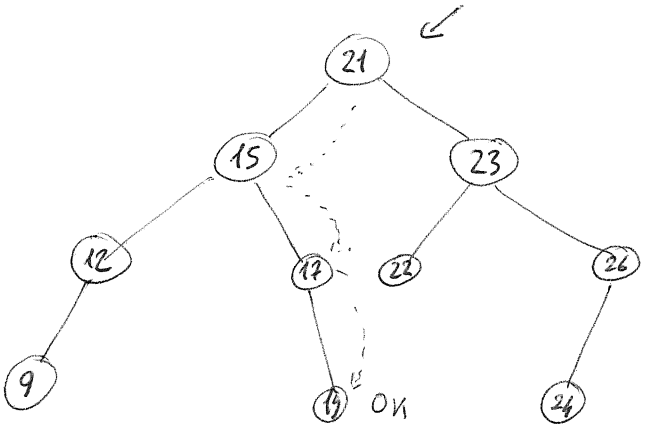
9, 12, 15, 17, 19, 21, 22, 23, 25, 26

ovvero la sequenza ordinata.

VISITA SIMMETRICA -> ALG. DECOMPONIBILI -> $O(n)$

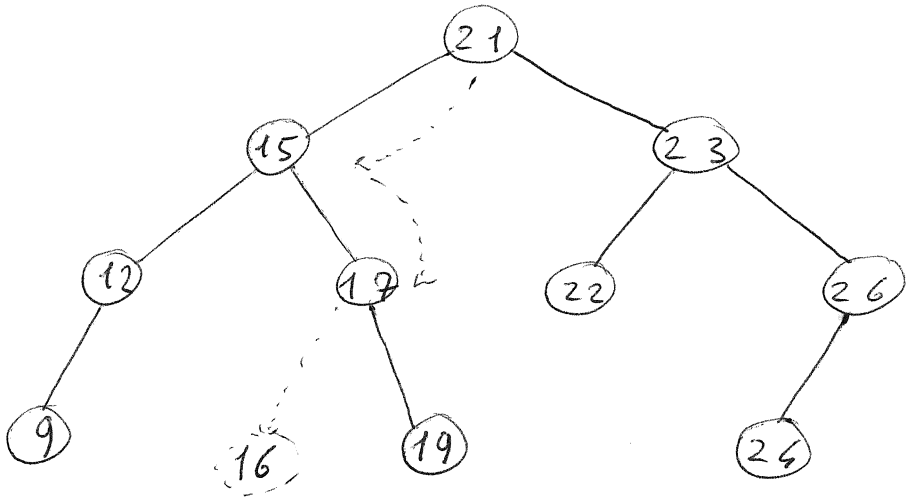
Operazioni

Ricerca di 19



Viene trovato procedendo lungo la path tratteggiata.

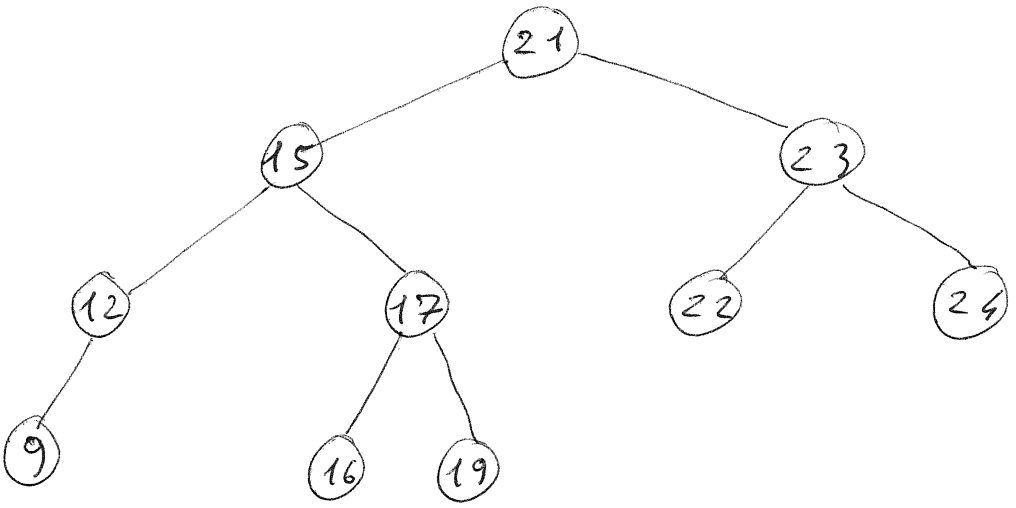
Ricerca di 20 segue la stessa path e stabilisce che l'elemento non c'è



Proviamo ad inserire **16**. Occorre:

- trovare il posto giusto, procedendo come per la ricerca
- modificare l'albero aggiungendo un nuovo nodo e facendolo diventare figlio destro o sinistro di un nodo preesistente.

Nell'esempio 16 diventa figlio sinistro di 17.



Il nodo (24), figlio sinistro del rimosso nodo (26), diventa figlio destro del nodo (23).

Il nodo "nonno" accoglie il "nipote".

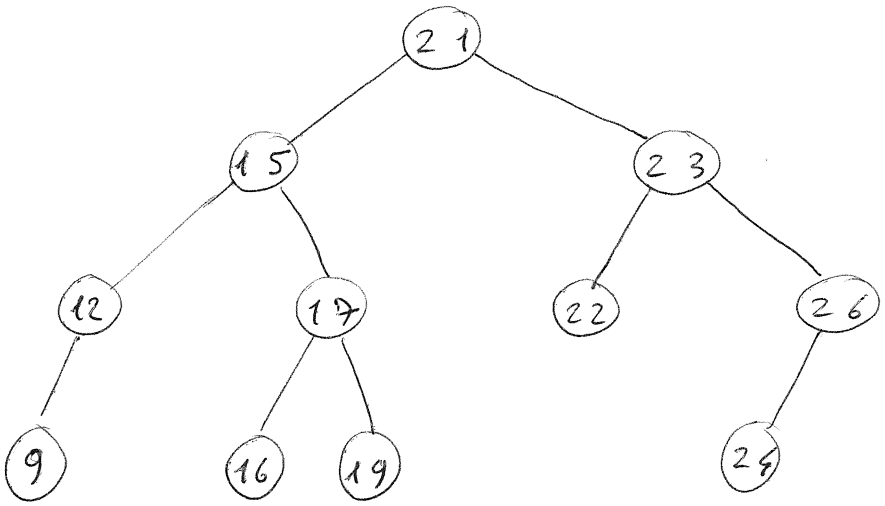
Nota: Prestate attenzione al codice di cancellazione. I due casi, foglia e nodo con un solo figlio, vengono gestiti assieme - stesse linee di codice.

Il caso più complesso si verifica quando vogliamo rimuovere un nodo con due figli.

Proviamo a rimuovere (15). L'idea è:

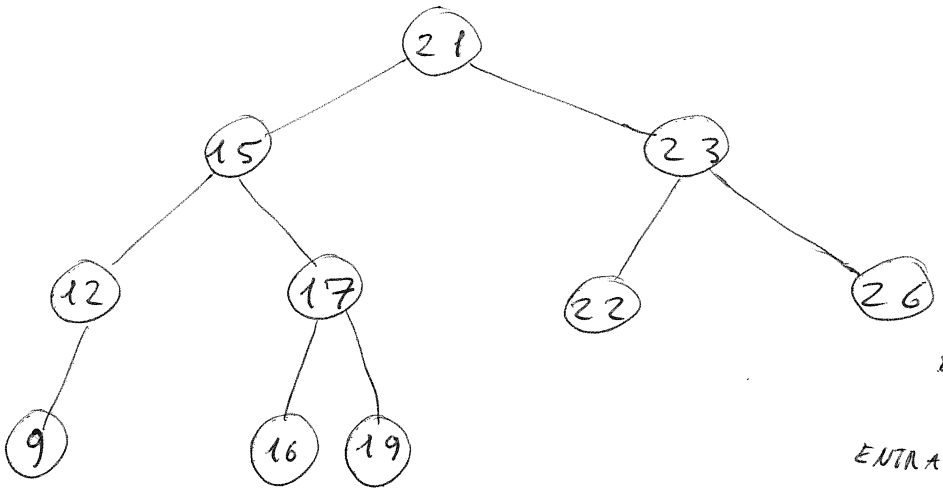
- trovare il "successore"
- metterlo al posto di 15
- rimuovere fisicamente il successore.

Cancellazione



La rimozione di una foglia è immediata.

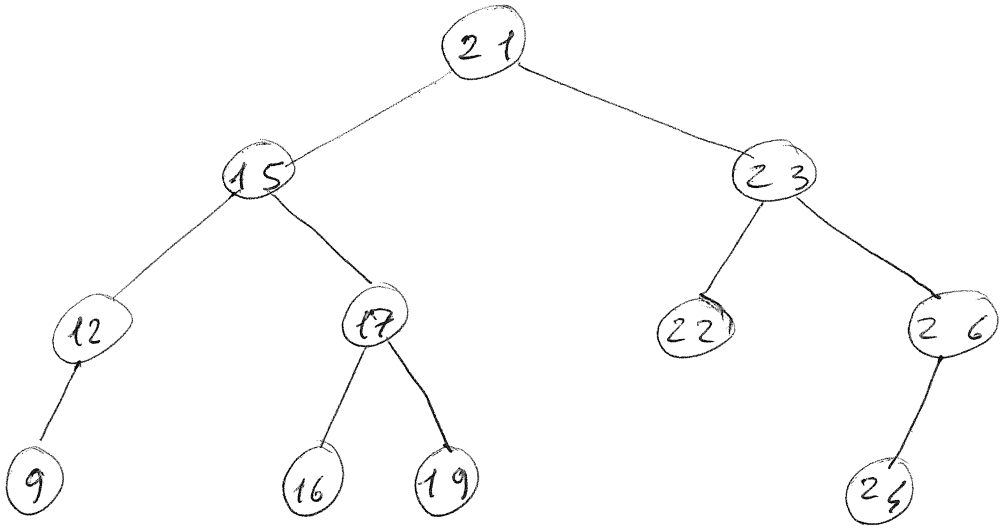
Per esempio, rimuoviamo 24.

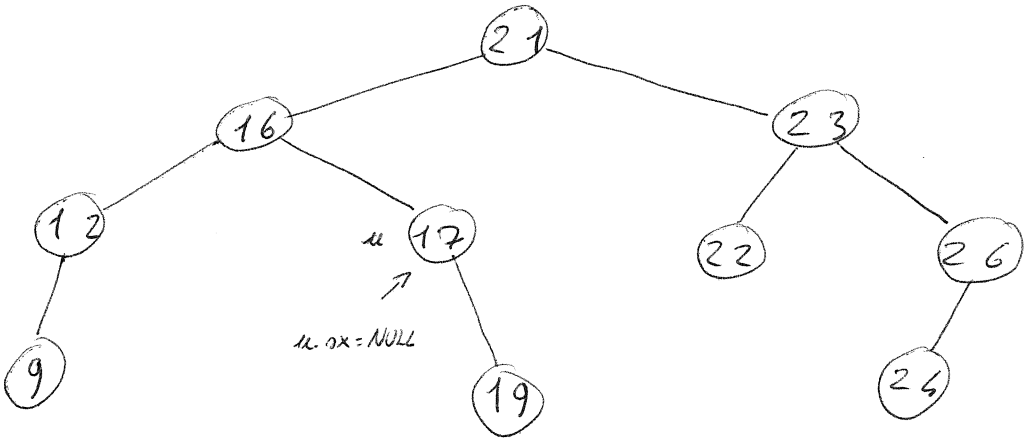


ENTRAMBI I PUNTATORI
a.sx e a.dx
del nodo (26) sono
NULL

Proviamo invece a rimuovere (26).

È tutto di un nodo CON UN SOLO FIGLIO





Il successore di (15) è l'elemento più piccolo del sottoalbero destro di (15).

Viene trovato "scendendo sempre a sinistra" nel sottoalbero.

Il successore NON può avere un figlio sinistro!

Se lo avesse, per la proprietà di ricerca, sarebbe minore del padre e maggiore del nodo che si vuole cancellare, cioè il suo SUCCESSORE!

Il codice che implementa Cancellazione usa

- una procedura per individuare il successore del nodo che si intende rimuovere.

- la procedura è essenzialmente una ricerca unidirezionale all'interno di un sottoalbero.

- il suo costo è $O(h)$, dove h indica l'altezza dell'albero.

L'inserimento viene invece realizzato tramite la seguente funzione:

```
Inserisci (u, e):
IF (u == NULL){
   u = NuovoNodo();
   u.dato = e;
   u.sx = u.dx = NULL;
} ELSE IF (e.chiave < u.dato.chiave){
      u.sx = Inserisci (u.sx, e);
} ELSE IF (e.chiave > u.dato.chiave){
      u.dx = Inserisci (u.dx, e);
}
RETURN u;
```

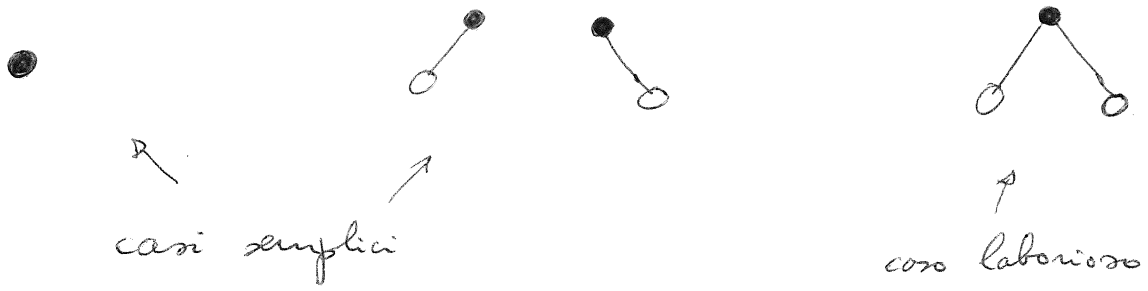
La funzione, quando raggiunge un riferimento vuoto, crea un nuovo nodo, memorizza e, ed "estende l'albero" (RETURN u).

Il "posto giusto" viene raggiunto a partire dalla radice, procedendo nel sottoalbero destro o sinistro ricorsivamente, a seconda della relazione tra e.chiave e u.dato.chiave, per il nodo u in esame.

La cancellazione è leggermente più complicata.

Occorre distinguere tre casi.

Il nodo è una foglia | Il nodo ha un solo figlio | Il nodo ha due figli



casi semplici

caso laborioso

```
Cancella (u, K)

IF (u != NULL) {
  IF (u.dato.chiave == K) {
    IF (u.sx == NULL) {
      u = u.dx;
    } ELSE IF (u.dx == NULL) {
      u = u.sx;
    } ELSE {
      w = MinimoSottoAlbero (u.dx);
      u.dato = w.dato;
      u.dx = Cancella (u.dx, w.dato.chiave);
    }
  ELSE IF (K < u.dato.chiave) {
    u.sx = Cancella (u.sx, K);
  } ELSE IF (K > u.dato.chiave) {
    u.dx = Cancella (u.dx, K);
  }
}
RETURN u;
```

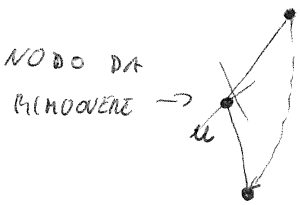
MinimoSottoAlbero (u):

```
WHILE (u.sx != NULL)
    u = u.sx;
RETURN u;
```

Questa funzione individua il minimo elemento nel sottoalbero radicato in u.

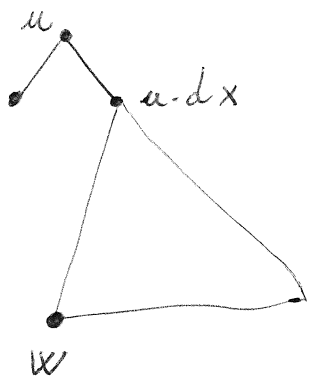
Se il nodo da rimuovere è una foglia oppure è un nodo con un solo figlio eliminiamo il nodo mettendo il puntatore a NULL oppure creiamo un ponte tra il padre di u e il figlio di u





Quando u ha due figli:

- individuiamo il successore di u. Sia esso w.
- copiamo il contenuto di w in u (implicitamente rimuovendolo)
- rimuoviamo fisicamente w (che non può avere il figlio sinistro essendo il valore minimo nel sottoalbero radicato in u.dx)



Complessità delle operazioni

- Ricerca $\longrightarrow$ $O(h)$
- Inserimento $\longrightarrow$ $O(h)$ h = altezza dell'albero
- Cancellazione $\longrightarrow$ $O(h)$

Se l'albero degenera in una lista il costo è $O(n)$.

Quando l'albero è bilanciato, ovvero $h = O(\log n)$,

il costo di tutte e tre le operazioni è $O(\log n)$.