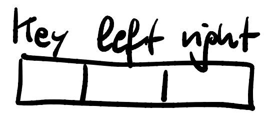
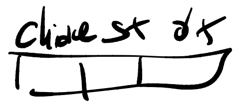
Appendice PUNTATORI



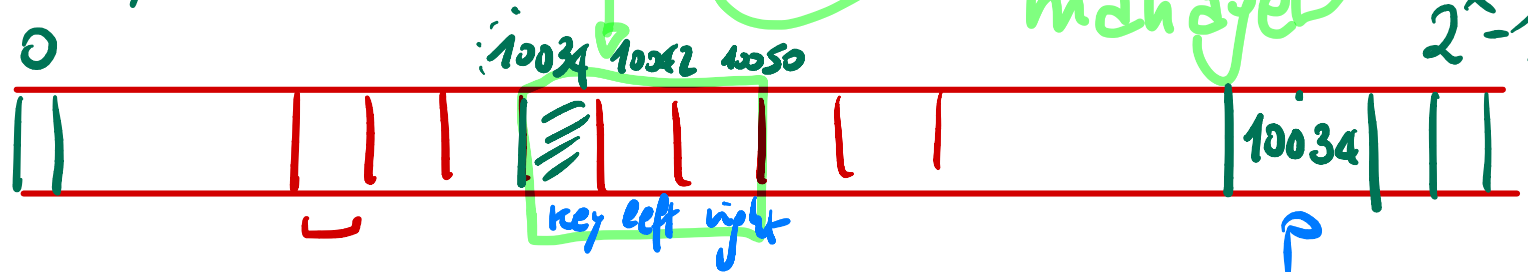



```
struct tree_node {
    int key;
    struct tree_node * left;
    struct tree_node * right;
}
```

3 celle di memoria (64 bit l'una)

```
using node =
    struct tree_node;
```

```
auto p = new struct tree_node;
```



stampa p : 10034

stampa *p :

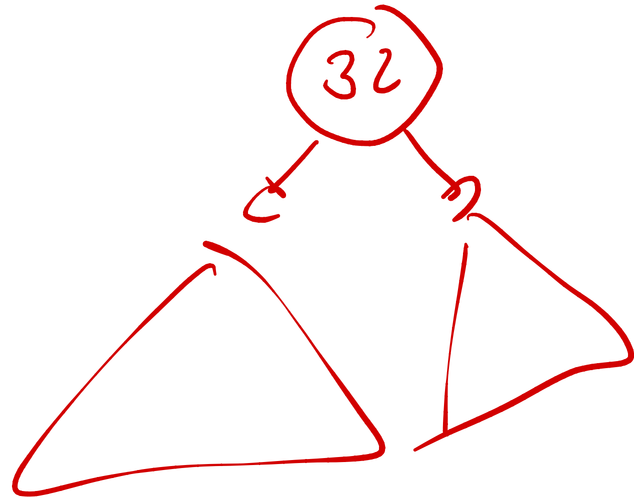
10034	10042	10050
32	i_1	i_2
key	left	right

*p.key ≡ p->key

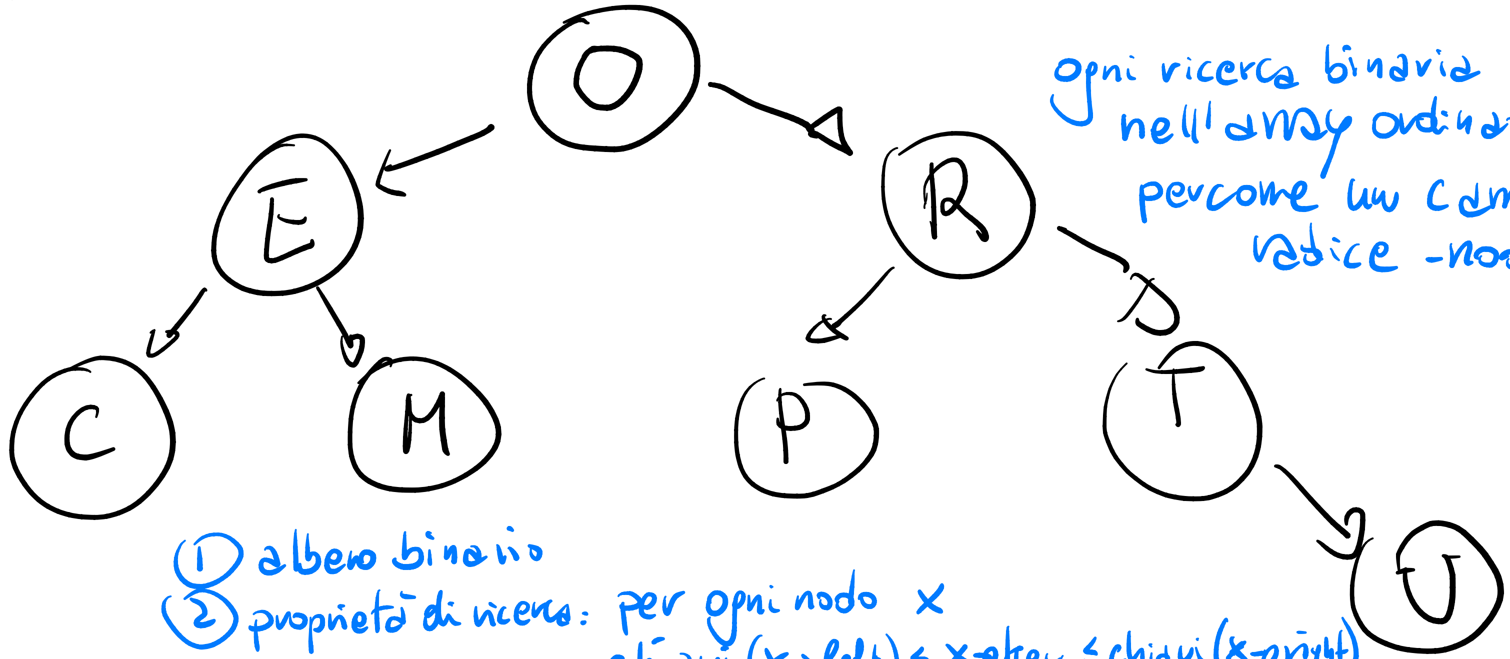
stampa p->key : 32

stampa p->left : i_1

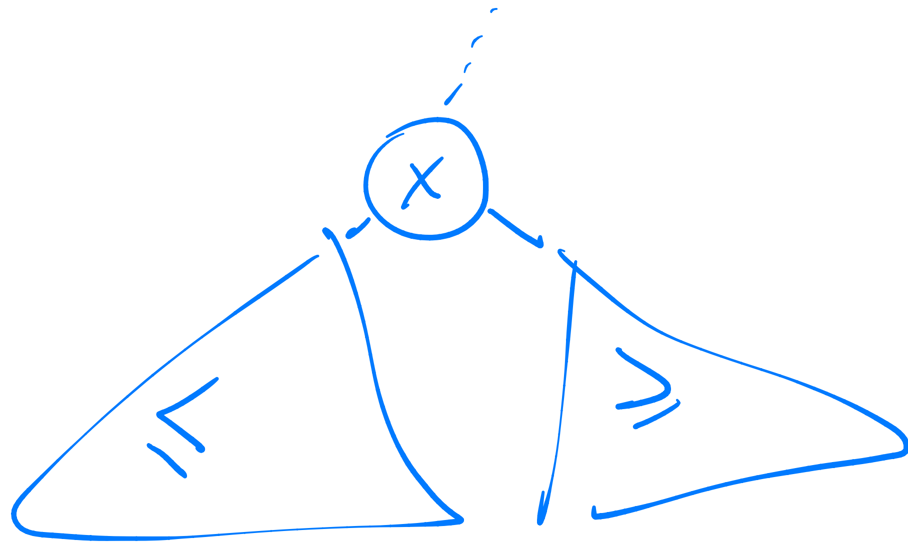
stampa p->right : i_2



Alberi binari di ricerca



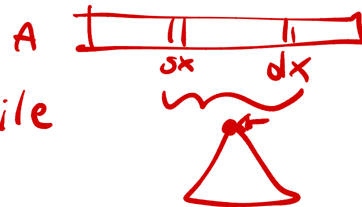
0	1	2	3	4	5	6	7
C	E	M	O	P	R	T	U

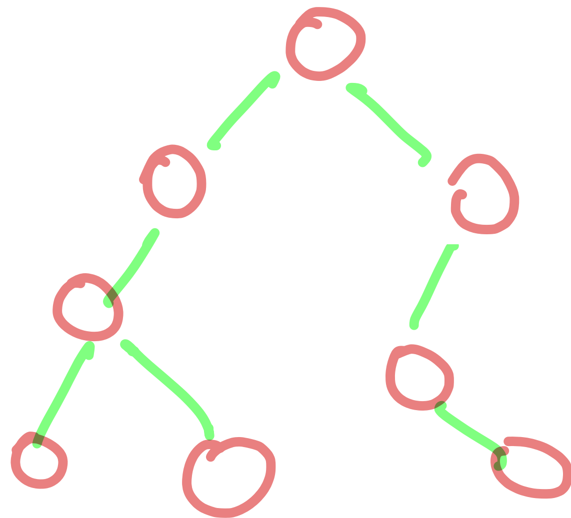

x
≤
≥

ORDINATO

```
struct tree_node * buildTree(vector<int> &A, int sx, int dx)
    if(sx>dx) return NULL
    auto p = new struct tree_node;
    auto cx = random(sx, dx)
    //auto cx = (sx+dx)/2
    p->key = A[cx]
    p->left = buildTree(A, sx, cx-1)
    p->right = buildTree(A, cx+1, dx)
    return p
```

decomponibile





RICERCA IN ALBERO BINARIO DI RICERCA

radice
chiave

```
1  Ricerca( u, k ):
2    IF (u == null) RETURN null;
3    IF (k == u.dato.chiave) {
4      RETURN u.dato;
5    } ELSE IF (k < u.dato.chiave) {
6      RETURN Ricerca( u.sx, k );
7    } ELSE {
8      RETURN Ricerca( u.dx, k );
9    }
```

nodo che contiene k (senza duplicati)

u -> key

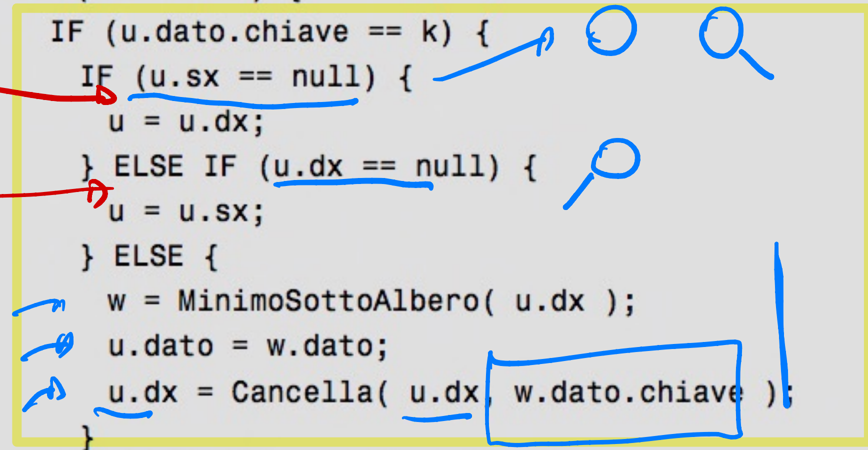
u -> left

u -> right

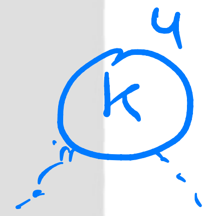
```
1  Ricerca( u, k ):
2    IF (u == null) RETURN null;
3    IF (k == u.dato.chiave) {
4      RETURN u.dato;
5    } ELSE IF (k < u.dato.chiave) {
6      RETURN Ricerca( u.sx, k );
7    } ELSE {
8      RETURN Ricerca( u.dx, k );
9    }
```

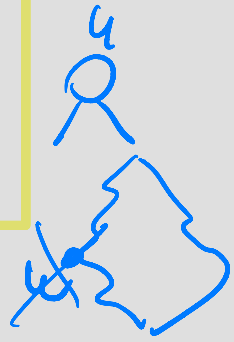
```
1  Inserisci( u, e ):
2    IF (u == null) {
3      u = NuovoNodo();
4      u.dato = e;
5      u.sx = u.dx = null;
6    } ELSE IF (e.chiave < u.dato.chiave) {
7      u.sx = Inserisci( u.sx, e );
8    } ELSE IF (e.chiave > u.dato.chiave) {
9      u.dx = Inserisci( u.dx, e );
10   }
11   RETURN u;
```

⟨*post: se* k *appare già in* u, *non viene memorizzata*⟩



```
1  Cancella( u, k ):
2    IF (u != null) {
3      IF (u.dato.chiave == k) {
4        IF (u.sx == null) {
5          u = u.dx;
6        } ELSE IF (u.dx == null) {
7          u = u.sx;
8        } ELSE {
9          w = MinimoSottoAlbero( u.dx );
10         u.dato = w.dato;
11         u.dx = Cancella( u.dx, w.dato.chiave );
12       }
13     } ELSE IF (k < u.dato.chiave) {
14       u.sx = Cancella( u.sx, k );
15     } ELSE IF (k > u.dato.chiave) {
16       u.dx = Cancella( u.dx, k );
17     }
18   }
19   RETURN u;
```





tmp = u
u = u.dx
delete tmp;

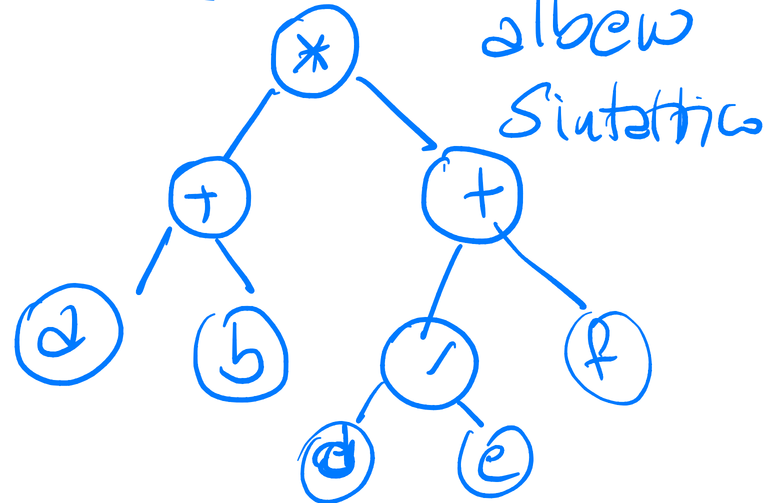
tmp = u;
u = u.sx
delete tmp

```
1  MinimoSottoAlbero( u ):                    ⟨pre: u ≠ null⟩
2    WHILE (u.sx != null)
3      u = u.sx;
4    RETURN u;
```

```
void tree_delete (struct tree_node * u) {
    if (u ≠ NULL) {
        tree_delete (u -> left)
        tree_delete (u -> right)
        delete u ;
    }
}
```

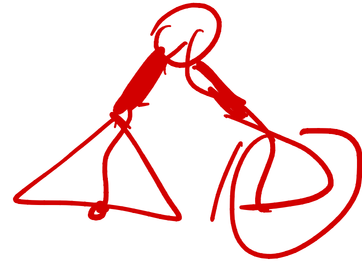
// visita post-order posticipata

$(a+b)*((d/e)+f)$



- altezza di ogni nodo
- int alt (u){

```
    if (u==NULL) return -1
    return 1 + max(alt(u->left), alt(u->right))
}
```



```
struct tree_node {
    int key;
    int height;
    struct tree_node * left;
    struct tree_node * right;
}
```

invariante

$\forall u: u\text{->}height = alt(u)$

senza eseguire alt(u)!

```
1  Inserisci( u, e ):
2    IF (u == null) {
3      u = NuovoNodo();
4      u.dato = e;
5      u.sx = u.dx = null;
6    } ELSE IF (e.chiave < u.dato.chiave) {
7      u.sx = Inserisci( u.sx, e );
8    } ELSE IF (e.chiave > u.dato.chiave) {
9      u.dx = Inserisci( u.dx, e );
10   }
11   RETURN u;
```

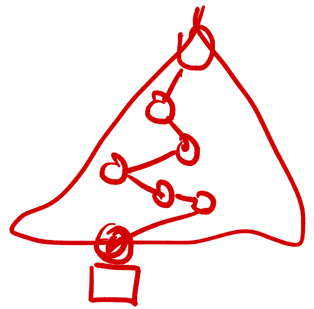
⟨*post: se* k *appare già in* u, *non viene memorizzata*⟩

u = new struct tree_node

u->height = 0

per induzione è quella corretta

u->height = 1 + max(myalt(u.sx), myalt(u.dx))



```
myalt(u):
  if (u==NULL) return -1
  return u->height
```

} O(1) tempo

OBIETTIVO: pagare O(altezza) tempo, non O(n.ro totale di nodi)!

mantenere l'altezza durante l'inserimento, evitando di lanciare la funzione alt( ) perché richiede tempo lineare, altrimenti ...

1. modificare la insert in modo da mantenere il nuovo campo u→size
2. Utilizzando il campo u→size
   - implementare rank(u, val) = # nodi che hanno key ≤ val
   - " " select(u, j) = nodo contenente la j-esima (se esiste) chiave in ordine crescente

esempio