Corso di laurea in Informatica
Introduzione agli Algoritmi

Dizionari: Alberi binari di ricerca

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

Alberi Binari di Ricerca (ABR)

- In una lista basata su puntatori, la ricerca di un elemento ha un costo asintotico di $\Omega(n)$. Questo accade perché alcuni elementi possono essere posizionati a una distanza significativa dalla testa della lista, richiedendo un'esplorazione sequenziale di tutti i nodi precedenti.

- Negli alberi binari, invece, un qualsiasi nodo può essere raggiunto potenzialmente in tempo $O(h)$, dove h è l'altezza dell'albero (ricorda che l'altezza di un albero è definita come la lunghezza del cammino più lungo dalla radice a una foglia).

- Inoltre, in un albero binario di altezza h, il numero totale di nodi può essere $\Theta(2^h)$. Nel caso ideale, con un **albero bilanciato**, ovvero con $h \approx \log_2 n$, posso potenzialmente raggiungere uno qualunque degli n nodi in tempo $O(\log n)$.

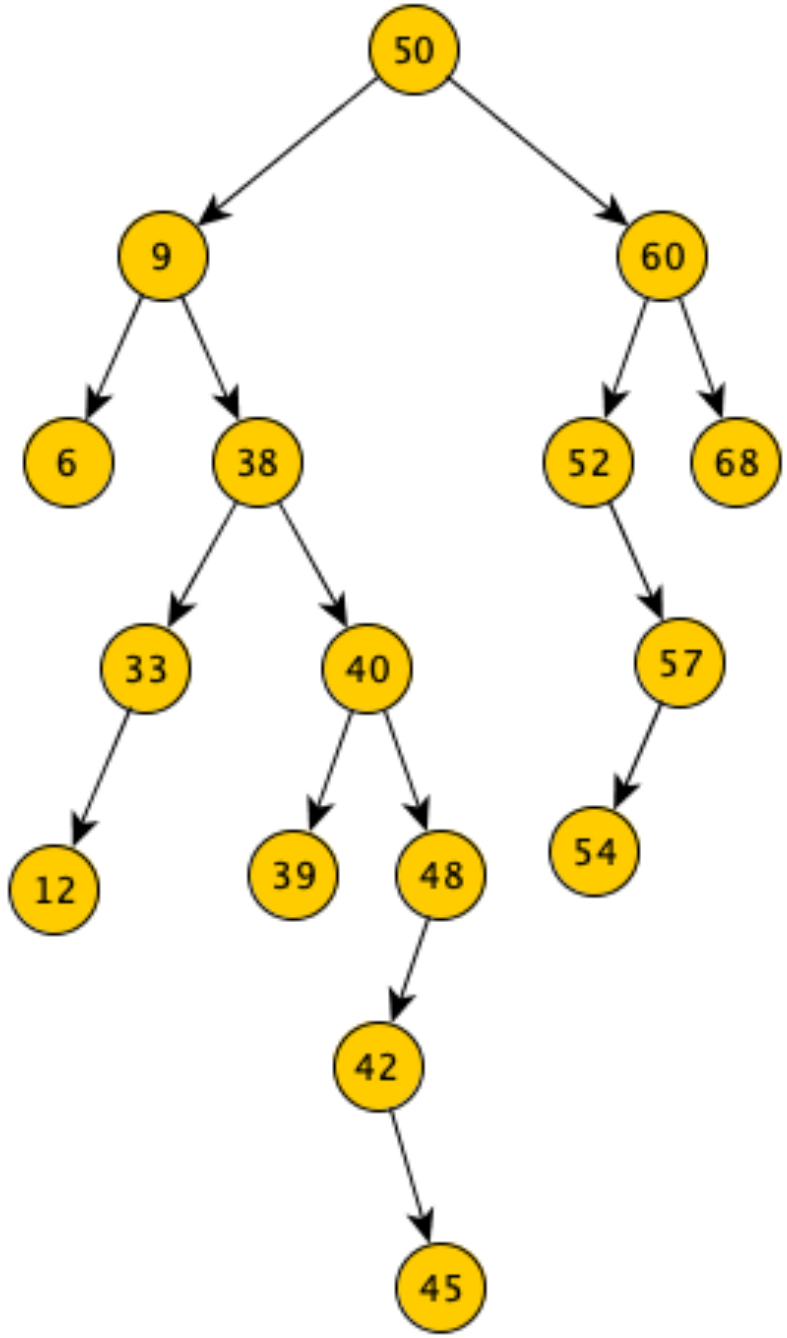
Affinché sia possibile raggiungere un elemento in tempo proporzionale all'altezza dell'albero è fondamentale che ci sia una regola che guidi la ricerca a partire dalla radice:

Un albero binario di ricerca (ABR) è un albero nel quale viene mantenuta la seguente proprietà:

- ogni nodo contiene una chiave (le chiavi sono distinte)
- il valore della chiave contenuta in ogni nodo è maggiore della chiave contenuta in ciascun nodo del suo sottoalbero sinistro (se esiste)
- il valore della chiave contenuta in ogni nodo è minore della chiave contenuta in ciascun nodo del suo sottoalbero destro (se esiste)

Questa proprietà garantisce che, ad ogni livello dell'albero, si possa decidere rapidamente quale ramo seguire per raggiungere il nodo desiderato.

ABR



Un dizionario è una struttura dati che permette di gestire un insieme dinamico di dati, che di norma è un insieme totalmente ordinato, tramite queste tre sole operazioni:

- **insert**: si inserisce un elemento;
- **search**: si ricerca un elemento;
- **delete**: si elimina un elemento.

Nel contesto dei dizionari, la **chiave** è un concetto fondamentale. Essa rappresenta l'identificatore univoco di un elemento all'interno della struttura. Ogni elemento del dizionario è composto da una *chiave* e un *valore* associato, dove la chiave funge da indice per accedere al valore. La relazione tra chiave e valore è definita come una mappatura, ovvero ogni chiave è associata a un solo valore.

Esempio 1: Studente universitario e Matricola In un sistema che gestisce gli studenti di un'università, una matricola può essere utilizzata come chiave unica per ciascun studente. Ogni studente ha una matricola che lo identifica in modo univoco nel database dell'università. In questo caso, la matricola funge da chiave nel dizionario: la chiave è il numero della matricola e il valore associato potrebbe essere il nome dello studente, i suoi corsi, i voti, ecc. La chiave (matricola) è unica per ogni studente, e ogni volta che si inserisce o si cerca uno studente nel sistema, la ricerca può essere eseguita utilizzando questa chiave.

Esempio 2: Libri in una Biblioteca e il Codice ISBN Nel contesto di una biblioteca, i libri sono comunemente identificati da un codice ISBN (International Standard Book Number), che funge da chiave unica per ogni libro. In questo caso, l'ISBN diventa la chiave nel dizionario. La chiave è il numero ISBN e il valore associato potrebbe essere il titolo, l'autore, la data di pubblicazione, o altre informazioni pertinenti sul libro.

Esempio 3: Cittadini e Codice fiscale Nel contesto di una nazione i cittadini sono identificati dal loro codice fiscale. In questo caso, il codice fiscale è utilizzato come chiave e i valori associati potrebbero essere nome, cognome, data di nascita, indirizzo, ecc.

Nelle prossime sezioni verranno descritti gli algoritmi che permettono di implementare in un albero di ricerca le tre operazioni tipiche di un dizionario, ovvero:

- Ricerca;
- Inserimento;
- Cancellazione.

in tempo $O(h)$, dove h è l'altezza dell'albero.

Nota che se l'albero di ricerca è anche bilanciato allora ciascuna delle tre operazioni avrà costo $O(\log n)$. Dove n è il numero di chiavi in cui cercare.

Nelle implementazioni degli algoritmi, si assumerà che i nodi dell'albero siano rappresentati utilizzando la seguente classe Python:

```
class NodoABR:
    def __init__(self, key=None,  left=None, right=None):
        self.key = key
        self.left = left
        self.right = right
```

ABR – Ricerca

Concettualmente simile alla ricerca binaria: la funzione riceve il puntatore alla radice dell'albero e la chiave da ricercare ed esegue una discesa dalla radice che viene guidata dai valori memorizzati nei nodi che si incontrano lungo il cammino.

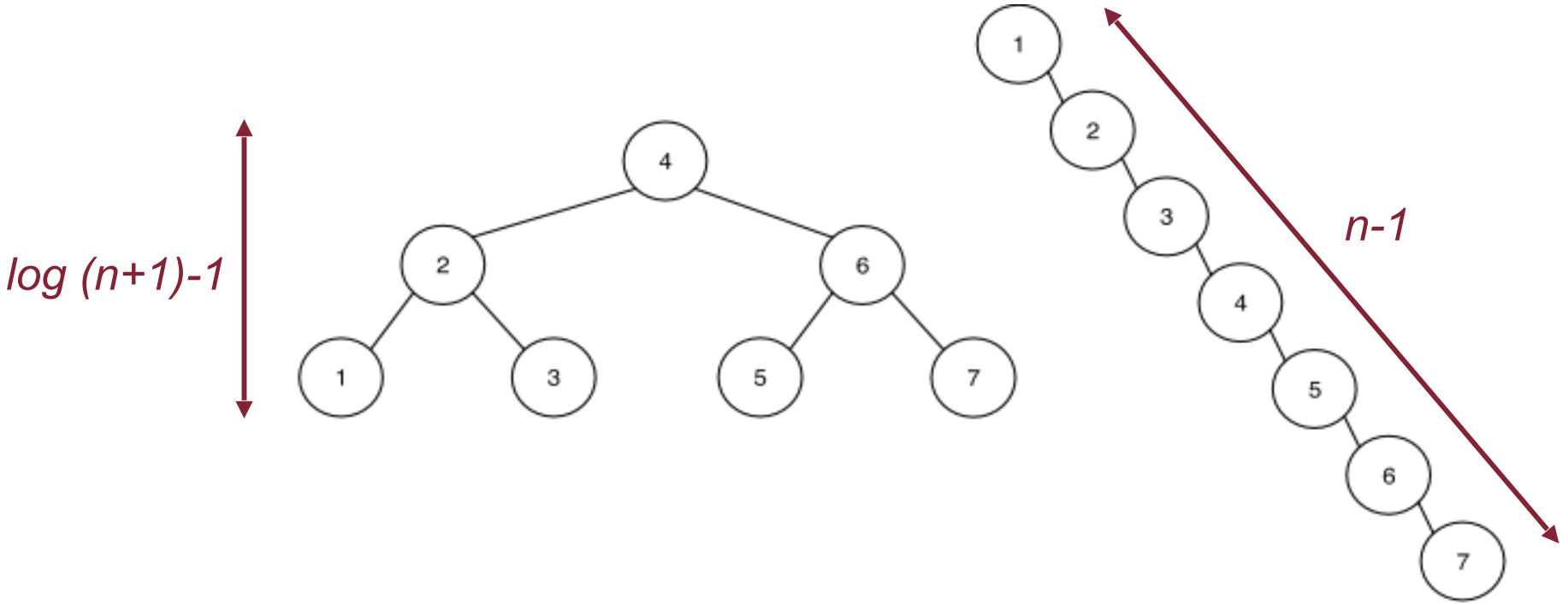
```
def ricercaABR(p, x):
    '''ricerca x nell'albero e restituisce il nodo
    che lo contiene, restituisce None se x non c'è'''
    while p is not None:
        if p.key == x:
            # Valore trovato
            return p
        if x < p.key:
            # Vai nel sottoalbero sinistro
            p = p.left
        else:
            # Vai nel sottoalbero destro
            p = p.right
    # Valore non trovato
    return None
```

Complessità $O(h)$ dove h è l'altezza dell'albero

ABR – ricerca

Considerando che la ricerca su un ABR ricorda molto da vicino la ricerca binaria che ha costo computazionale $O(\log n)$, come mai non riusciamo a garantire un costo logaritmico anche per la ricerca su un ABR?

Problema: l'ABR non include fra le sue proprietà alcunché in relazione alla sua altezza.



Quindi: se vogliamo garantire che il costo computazionale della ricerca su ABR sia limitata superiormente da un logaritmo, dovremo ricorrere a qualche tecnica che ci permetta di tenere sotto controllo la crescita dell'altezza: ***bilanciamento in altezza***.

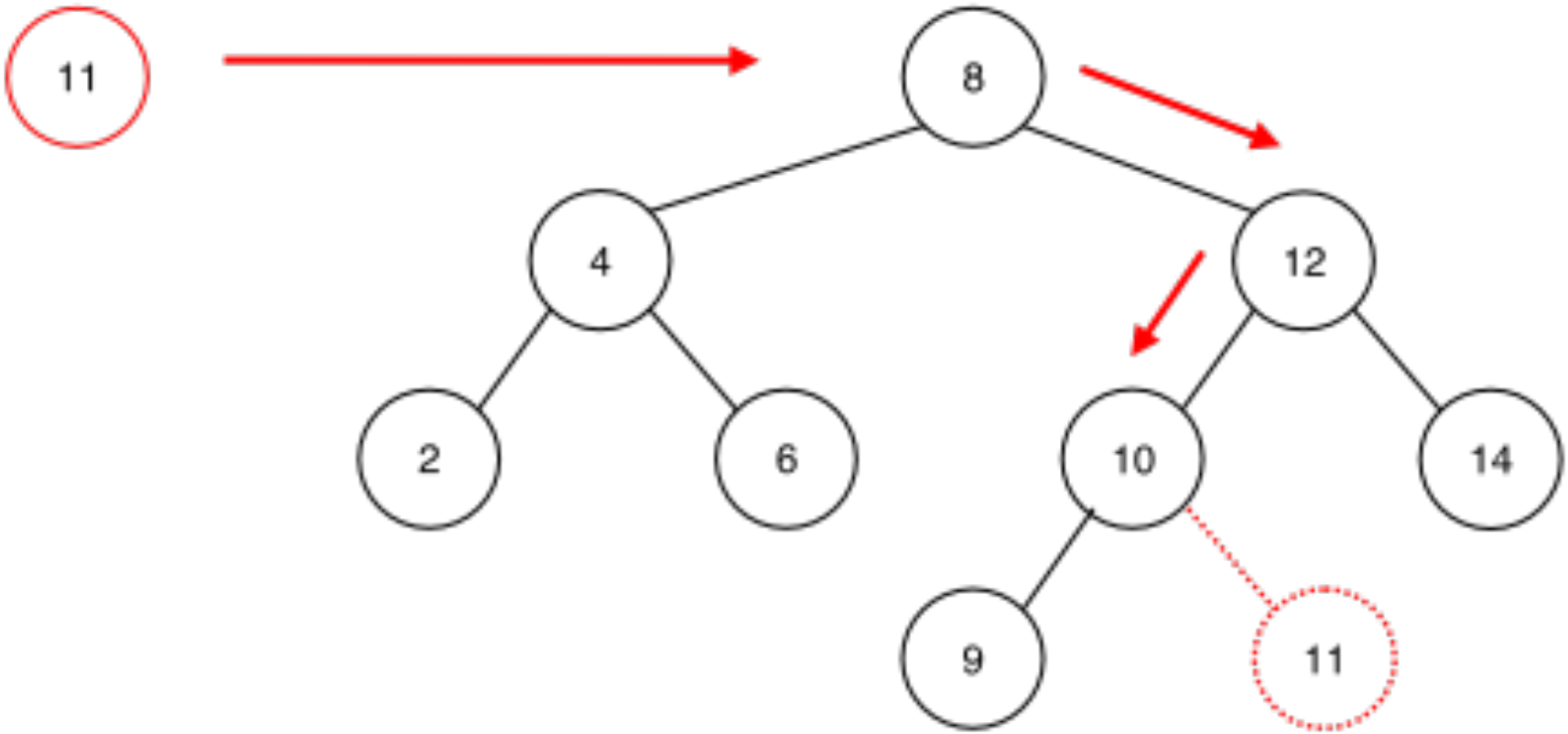
ABR – inserimento

- Si esegue una discesa che, come nel caso della ricerca, viene guidata dai valori memorizzati nei nodi che si incontrano lungo il cammino.

- Quando si arriva al punto di voler proseguire la discesa verso un puntatore vuoto (*None*) allora, in quella posizione, si aggiunge un nuovo nodo contenente il valore da inserire.

- Nota che: Il padre di tale nuovo nodo potrebbe essere una foglia (entrambi i suoi figli sono *None*), ma, più in generale, è un nodo a cui manca il figlio corrispondente alla direzione che si dovrebbe prendere per proseguire.

ABR – inserimento



```python
def inserisciABR(root, x):
    '''restituisce la radice dell'albero di ricerca, che potrebbe
    essere cambiata a seguito dell'inserimento di x '''
    z = NodoABR(x)
    if root is None:
        # Se l'albero è inizialmente vuoto
        return z
    # esplora l'albero iterativamente
    nodo = root
    while True:
        if x < nodo.key:
            # Se il valore è minore,
            # spostati nel sottoalbero sinistro
            if nodo.left is None:
                nodo.left = z
                break
            nodo = nodo.left
        elif x > nodo.key:
            # Se il valore è maggiore,
            # spostati nel sottoalbero destro
            if nodo.right is None:
                nodo.right = z
                break
            nodo = nodo.right
        else:
            # Se il valore è uguale, non inserire
            # nulla (valore già presente)
            break
    return root
```

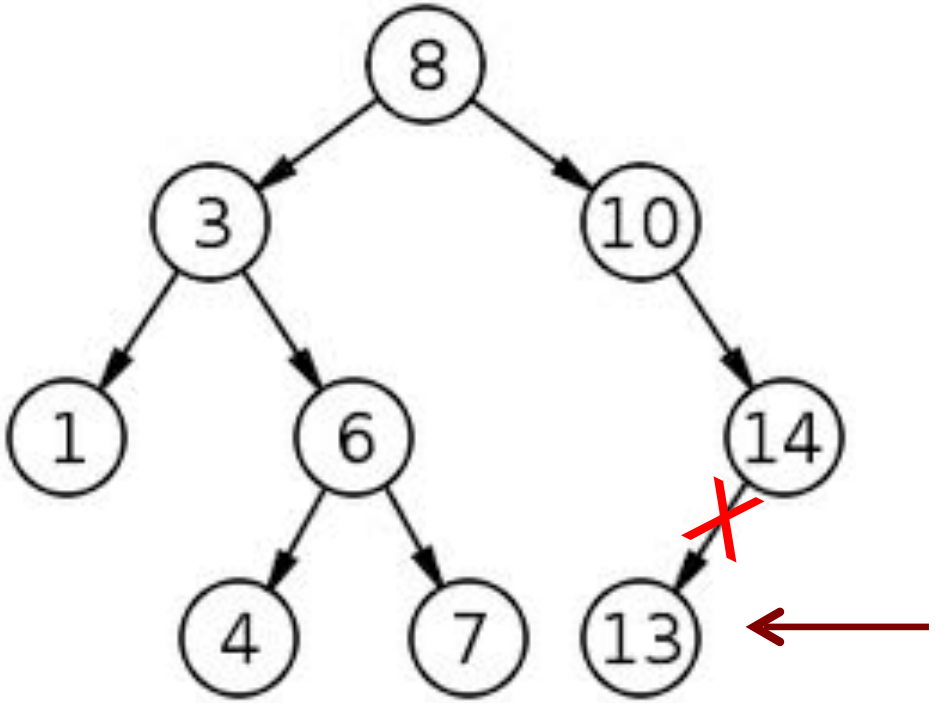
Complessità $O(h)$ dove h è l'altezza dell'albero

ABR – cancellazione

Per eliminare un nodo in un ABR distinguiamo essenzialmente **3 casi** in funzione del numero di figli che ha il nodo da cancellare:

- **Caso 1**: se il nodo è una foglia lo si elimina, ponendo a *None* l'opportuno campo nel nodo padre;

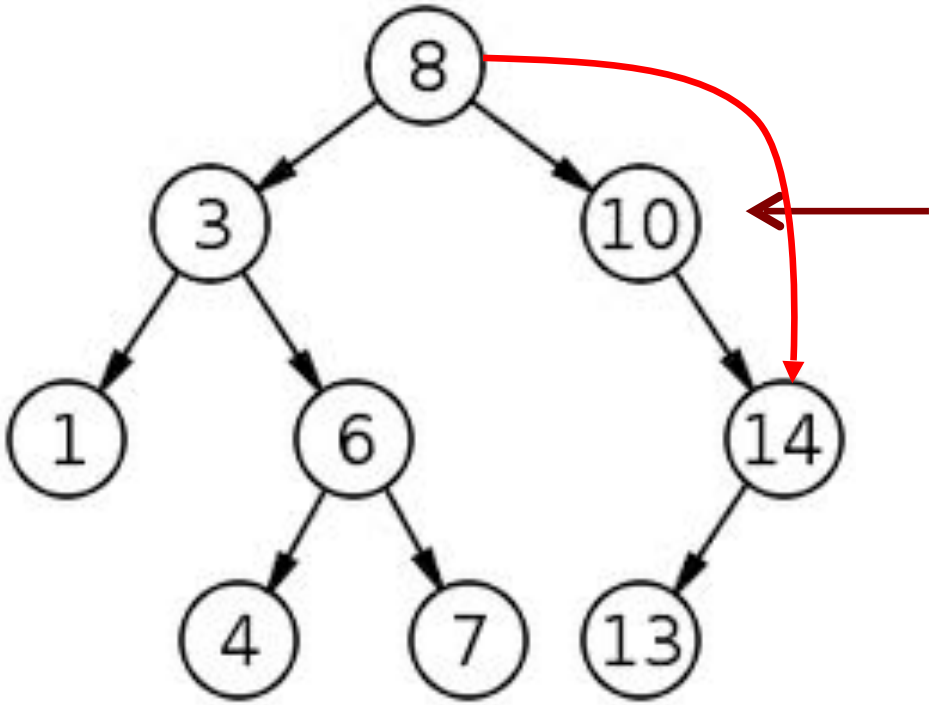
Nell'esempio eliminazione di 13.



ABR – cancellazione

- **Caso 2**: se il nodo ha un solo figlio lo si "cortocircuita", cioè si collegano direttamente fra loro suo padre e il suo unico figlio, indipendentemente che sia destro o sinistro;

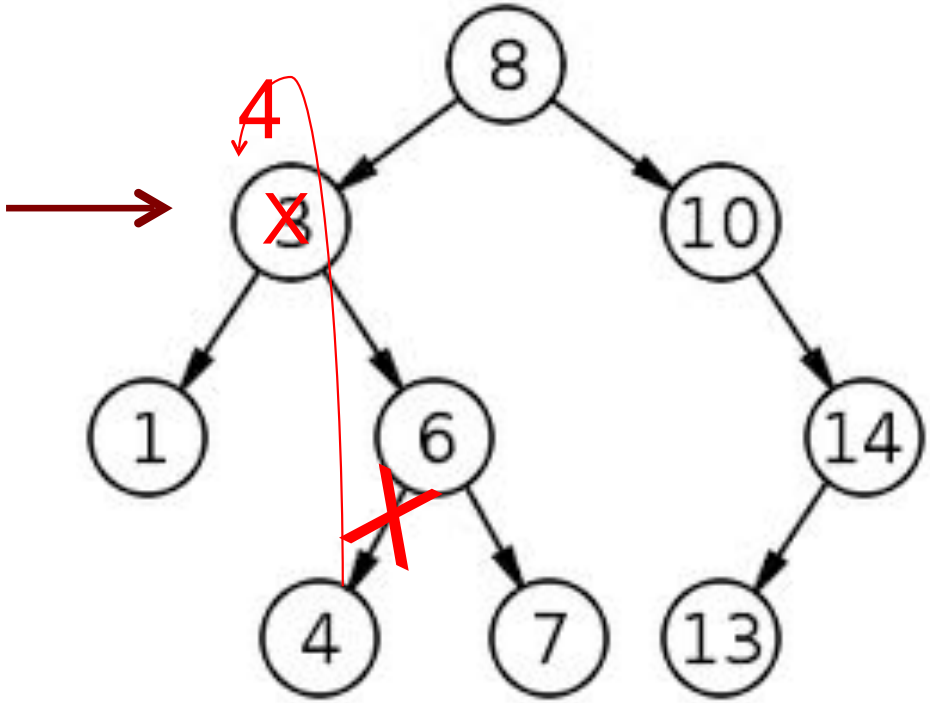
Nell'esempio eliminazione di 10.



ABR – cancellazione

Caso 3: se il nodo ha entrambi i figli si cancella il suo valore sostituendolo con quello del successore e si scende a cancellare il successore (che sarà una foglia o un nodo con un solo figlio, quello destro).

Nell'esempio eliminazione di 3.



```
def cancellaABR(root, x):
    '''cancella l'elemento x dall'albero radicato in root e
    restituisce la radice eventualmente modificata '''
    if root is None:
        return None
    nodo = root
    genitore = None
    # Cerca iterativamente il nodo da cancellare
    while nodo is not None and nodo.key != x:
        genitore = nodo
        if x < nodo.key:
            nodo = nodo.left
        else:
            nodo = nodo.right
    if nodo is None:
        return root  #nodo non trovato
    .....
```

```
.........
# Caso 1: Nodo da cancellare è una foglia
if nodo.left is None and nodo.right is None:
    if genitore is None:
        return None  # nodo cancellato era radice
    elif genitore.left == nodo:
        genitore.left = None
    else:
        genitore.right = None
....
```

```
.....
# Caso 2: Nodo da cancellare ha un solo figlio
elif nodo.left is None or nodo.right is None:
    # Determina il figlio da mantenere
    if nodo.left is not None:
        figlio = nodo.left
    else:
        figlio = nodo.right
    if genitore is None:
        return figlio  # La radice cambia
    elif genitore.left == nodo:
        genitore.left = figlio
    else:
        genitore.right = figlio
...
```

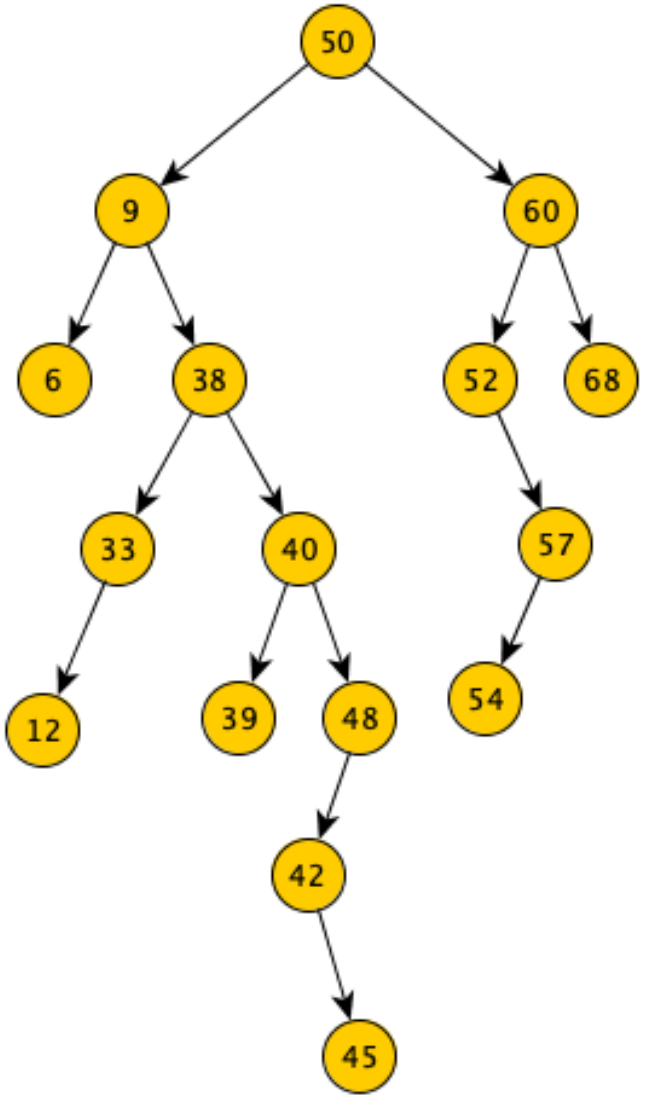
```
.....

# Caso 3: Nodo da cancellare ha due figli
else: # Trova il successore (minimo del sottoalbero destro)
    successore_gen = nodo
    successore = nodo.right
    while successore.left:
        successore_gen = successore
        successore = successore.left
    # Copia la chiave del successore nel nodo
    nodo.key = successore.key
    # Rimuovi il successore dal suo posto
    if successore_gen.left == successore:
        successore_gen.left = successore.right
    else:
        successore_gen.right = successore.right

return root
```

ESERCIZIO 1. Stampa le chiavi di un albero di ricerca in ordine crescente.

NOTA: Per elencare tutte le chiavi in ordine crescente basta compiere una visita inorder.




```
def stampaABR(p):
    #stampa inorder degli elementi dell'albero
    if p != None:
        stampaABR( p.left )
        print( p.key )
        stampaABR( p.right )
```

stampa chiavi in inorder:

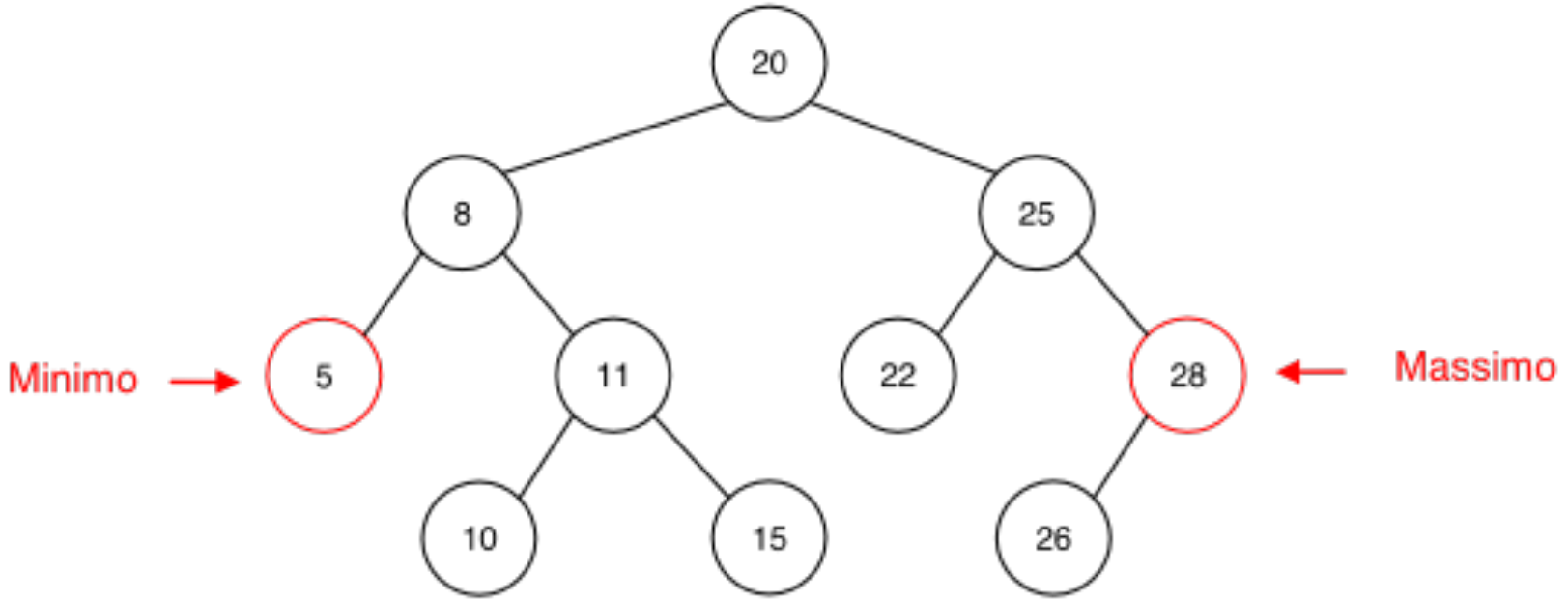
6 9 12 33 38 39 40 42 45 48 50 52 54 57 60 68

ESERCIZIO 2: minimo e massimo

Minimo e massimo

Il minimo (massimo) si trova nel nodo più a sinistra (destra), quindi per trovarlo si scende sempre a sinistra (destra) a partire dalla radice. Ci si ferma quando si arriva a un nodo che non ha figlio sinistro (destro): quel nodo contiene il minimo (massimo).

$T(n) = O(h)$



```
def minimoABR(p):
    ''' restituisce il nodo con la chiave minima
    nell'albero di ricerca, se l'albero è vuoto
    restituisce None'''
    if not p:
        return None
    while p.left:
        p = p.left
    return p
```

ESERCIZIO 3:

Progettare una procedura che, data una lista di n interi, restituisca il puntatore ad un albero binario di ricerca che contenga gli interi della lista. La procedura deve avere complessità $O(n \log n)$.

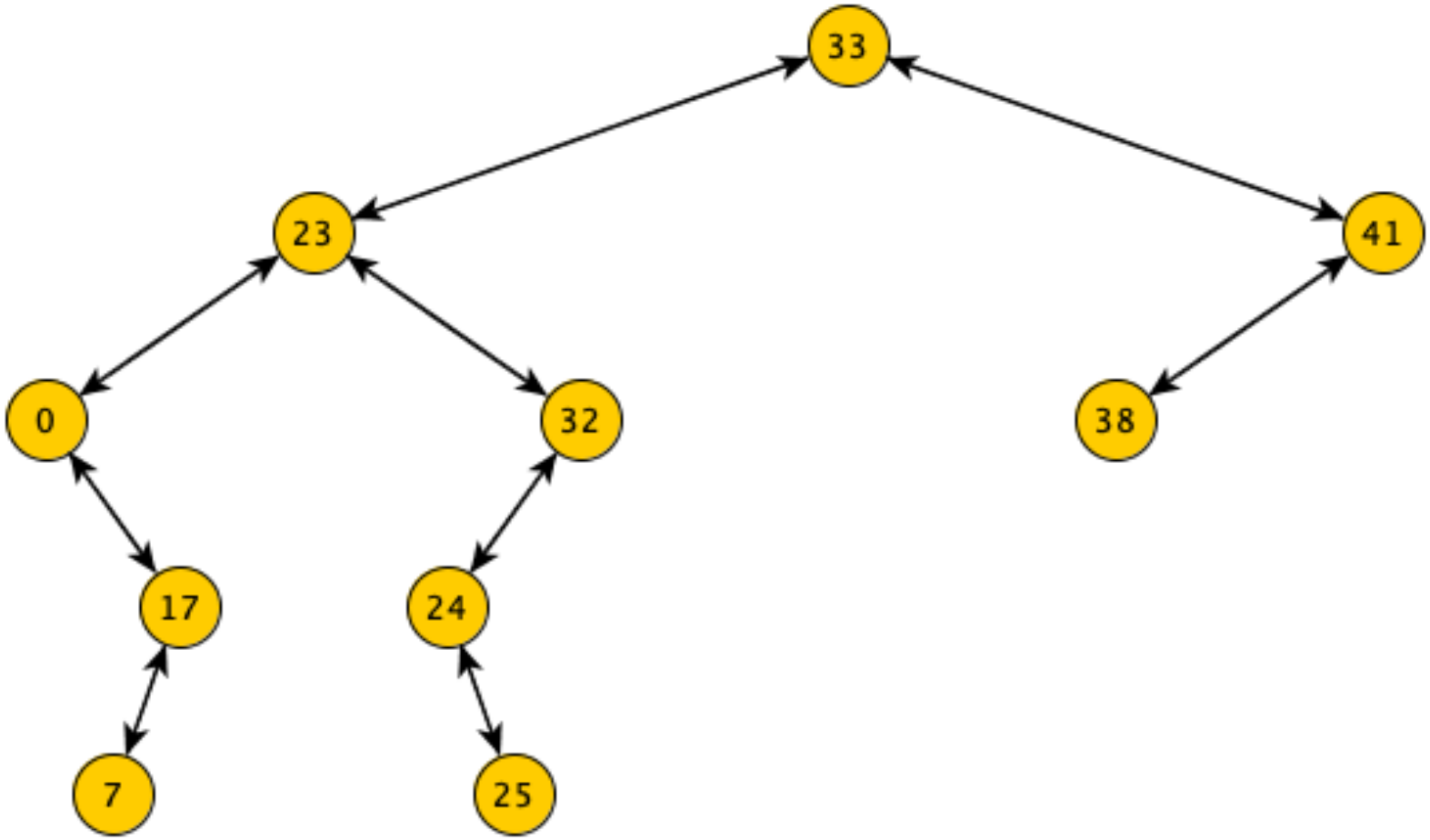
```
import random

def generaABR(A):
    lista = sorted(A, reverse = True)
    return generaABR1(len(A),lista)

def generaABR1(n, lista):
    if n == 0: return None
    p = NodoABR()
    #genera a caso il numero s di figli del sottoalbero di sinistra di p
    s = random.randint(0, n-1)
    # genera il sottoalbero di sinistra di p
    p.left = generaABR1(s,lista)
    #assegna la chiave al nodo p
    p.key = lista.pop()
    # genera il sottoalbero di sinistra di p
    p.right = generaABR1(n-1-s, lista)
    return p
```

```
>>> A = [7, 33, 25, 17, 24, 0, 32, 38, 41, 23]
>>> r = generaABR(A)
>>> stampaABR(r)
```

```
33
| 23
| | 0
| | | -
| | | 17
| | | | 7
| | | | | -
| | | | | -
| | | | -
| | 32
| | | 24
| | | | -
| | | | 25
| | | | | -
| | | | | -
| | | -
| 41
| | 38
| | | -
| | | -
| | -
```



Corso di laurea in Informatica
Introduzione agli Algoritmi

Esercizi per casa

SAPIENZA
UNIVERSITÀ DI ROMA

ESERCIZI:

1. Progetta le versioni ricorsive dei tre algoritmi di ricerca, inserimento e cancellazione in un albero di ricerca.

2. Progetta una procedura che data la radice di un albero di ricerca ne stampi gli elementi in ordine decrescente. Valutare La complessità temporale e spaziale della vostra procedura.

3. Costruisci un albero di ricerca binario inserendo, nell'ordine indicato, le seguenti chiavi: 20, 7, 18, 15, 30, 19 e 12. Successivamente, rimuovi dall'albero la radice e poi la chiave 18. Per ogni passaggio, mostra la struttura dell'albero risultante.

4. Si desidera costruire un albero di ricerca contenente le chiavi da 1 a 7. Determinare un ordine di inserimento delle chiavi tale da ottenere un albero di ricerca con altezza pari a 6 e un ordine di inserimento delle chiavi tale da ottenere un albero di ricerca con altezza pari a 2.

5. Progettare una funzione che, data la radice ad un albero binario di ricerca, restituisce la chiave di valore massimo presente nell'albero. Se l'albero è vuoto la funzione deve restituire $None$. Scrivete prima la versione ricorsiva e poi quella iterativa.

6. Progettare una funzione che, data la radice ad un albero binario di ricerca ed una chiave x, restituisce il predecessore massimo di x presente nell'albero. Se nell'albero non esistono chiavi con valore inferiore a x, la funzione deve restituire $None$. Calcolare la complessità della funzione proposta.

ESERCIZI:

1. Progettare una funzione che, data la radice ad un albero binario di ricerca e due chiavi x e y con $x \leq y$, restituisce la lista delle chiavi presenti nell'albero con valori compresi tra x e y. Calcolare la complessità della funzione proposta.

2. Progettare una funzione che, data la radice ad un albero binario, verifichi che l'albero sia un albero binario di ricerca. La funzione deve controllare se l'albero rispetta le proprietà di un albero di ricerca, ovvero, i valori dei nodi sono distinti e per ogni nodo con chiave x, le chiavi del sottoalbero sinistro devono essere inferiori a x mentre quelle del sottoalbero destro devono essere maggiori di x.

3. Progettare una funzione che, date le radici di due alberi binari di ricerca, fonde i due alberi in uno e ne restituisca la radice. Si può assumere che le chiavi del primo albero sono minori di quelle del secondo. La funzione deve avere complessità $O(\min(h_1, h_2))$ dove h_1 è l'altezza del primo albero ed h_2 quella del secondo. La complessità spaziale della procedura deve essere $O(1)$.

4. Progettare una funzione che, date le radici di due alberi binari di ricerca contenenti chiavi distinte, li fonde in un unico albero binario di ricerca. Discutere la complessità della procedura in termini dei nodi n_1 del primo albero ed n_2 del secondo.

5. Progettare una funzione che, data la radice ad un albero binario di ricerca e due sue chiavi, restituisce il minimo antenato comune (LCA) in tempo $O(h)$ dove h è l'altezza dell'albero. Il minimo antenato comune a due distinti nodi x e y di un albero è il primo nodo in comune ai due cammini che portano da da questi nodi alla radice dell'albero.