Algoritmi e Strutture di Dati I

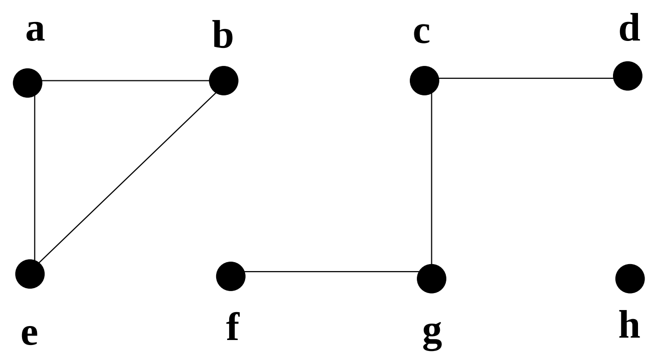
Massimo Franceschet
http://www.sci.unich.it/~francesc

Grafo

Un grafo G è una coppia (V, E) ove V è un insieme di **nodi** e E è un insieme di **archi**. Un arco è un insieme $\{u, v\}$ di due nodi distinti.

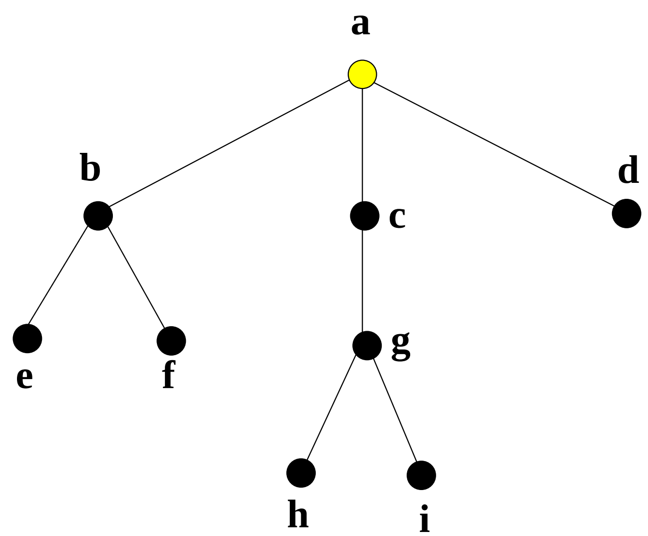
Un **cammino** è una sequenza di nodi collegati da archi. Un **ciclo** è un cammino che ritorna al punto di partenza.

Un grafo è **connesso** se ogni coppia di nodi è collegata da un cammino. Un grafo si dice **aciclico** se non contiene cicli.



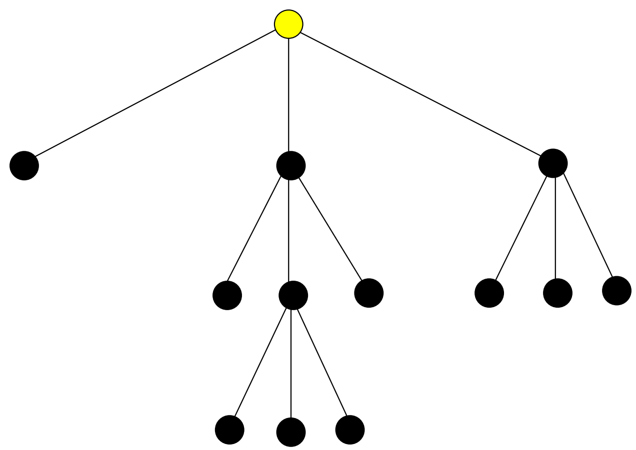
Albero radicato

Un **albero radicato** è un grafo **aciclico** e **connesso** in cui esiste un nodo speciale chiamato **radice**.



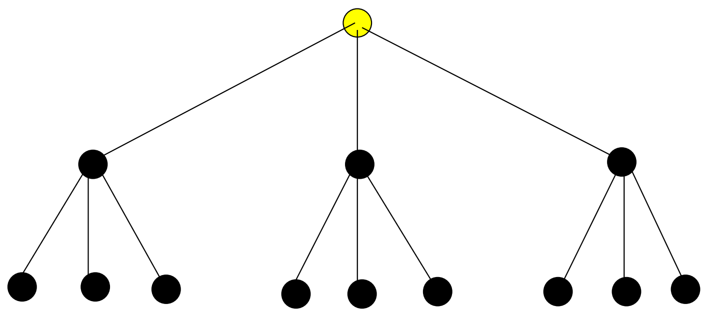
Albero ternario pieno

Dato un intero $k \leq 1$, un albero si dice **k-ario** se ogni nodo interno ha al massimo k figli. Un albero k-ario è **pieno** se ogni nodo interno ha esattamente k figli.



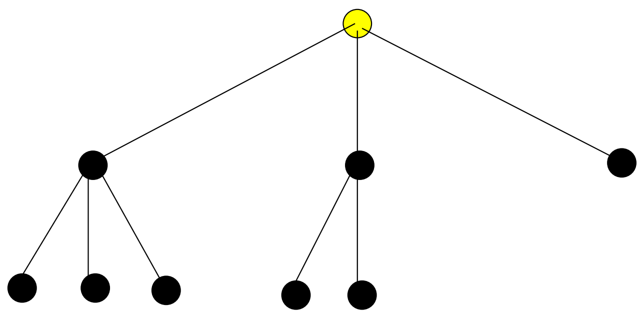
Albero ternario completo

Un albero è **completo** se è pieno e ogni sua foglia ha la medesima profondità.



Albero ternario quasi completo

Un albero è **quasi completo** se è completo fino al penultimo livello e i nodi dell'ultimo livello sono inseriti da sinistra a destra.



Qual è il numero di nodi di un albero binario completo di altezza h?

Sia T un albero binario completo di altezza h e sia n il numero di nodi di T. Sommando i nodi per livelli, abbiamo che

$$n = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1.$$

Dunque $n = \Theta(2^h)$.

Qual è l'altezza h di un albero binario completo di n nodi?

Abbiamo che $n = 2^{h+1} - 1$. Passando al logaritmo,

$$h = \log(n + 1) - 1 = \Theta(\log n).$$

Qual è il minimo e il massimo numero di nodi di un albero binario quasi completo di altezza h?

Il minimo numero di nodi di un albero quasi completo di altezza h è 2^h (l'albero ha un solo nodo all'ultimo livello), il massimo è $2^{h+1} - 1$ (l'albero è completo). Cioè

$$2^h \leq n \leq 2^{h+1} - 1$$

e dunque $n = \Theta(2^h)$ e $h = \Theta(\log n)$.

Qual è il minimo e il massimo numero di nodi di un albero binario pieno di altezza h?

Il minimo numero di nodi di un albero pieno di altezza h è $2h+1$ (ogni nodo interno ha almeno un figlio foglia), il massimo è $2^{h+1}-1$ (l'albero è completo). Cioè

$$2h+1 \leq n \leq 2^{h+1}-1.$$

Dunque $n = \Omega(h)$ e $n = O(2^h)$, e quindi $h = \Omega(\log n)$ e $h = O(n)$.

Qual è il minimo e il massimo numero di nodi di un albero binario di altezza h?

Il minimo numero di nodi di un albero di altezza h è $h+1$ (l'albero è lineare, cioè ogni nodo interno ha un unico figlio), il massimo è $2^{h+1}-1$ (l'albero è completo). Cioè

$$h+1 \leq n \leq 2^{h+1}-1.$$

Dunque $n = \Omega(h)$ e $n = O(2^h)$, e quindi $h = \Omega(\log n)$ e $h = O(n)$.

Rappresentazione di alberi radicati

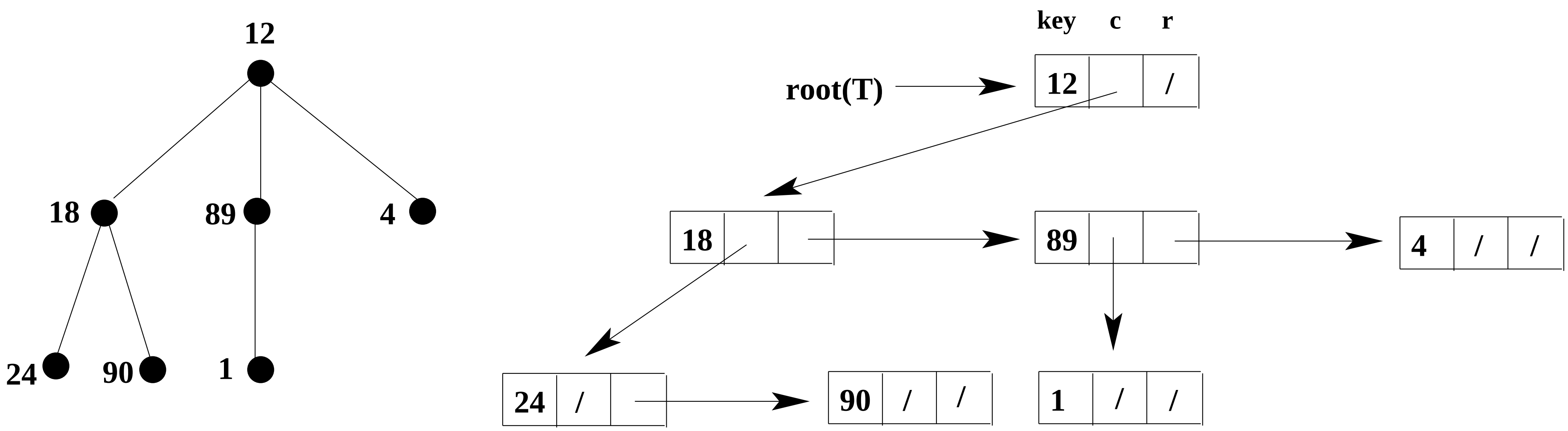
Rappresento ogni nodo x dell'albero con un oggetto dotato dei seguenti campi:

- $key[x]$ che contiene la **chiave** di x;
- $c[x]$ che punta al **figlio più a sinistra** di x, oppure è NIL se x non ha figli;
- $r[x]$ che punta al **fratello destro** di x, oppure è NIL se x non ha fratelli destri.

Se il nodo x non ha figli, allora $c[x] = \text{NIL}$, se x non ha fratelli destri, allora $r[x] = \text{NIL}$.

La struttura di dati albero radicato possiede un attributo $root(T)$ che contiene un puntatore alla radice dell'albero T.

Rappresentazione di alberi radicati



Visite di alberi

Visitare un albero significa esplorare tutti i suoi nodi. Vogliamo farlo in maniera efficiente, cioè senza ripassare per zone già visitate. Ci sono due modi per visitare un albero (e più in generale, un grafo):

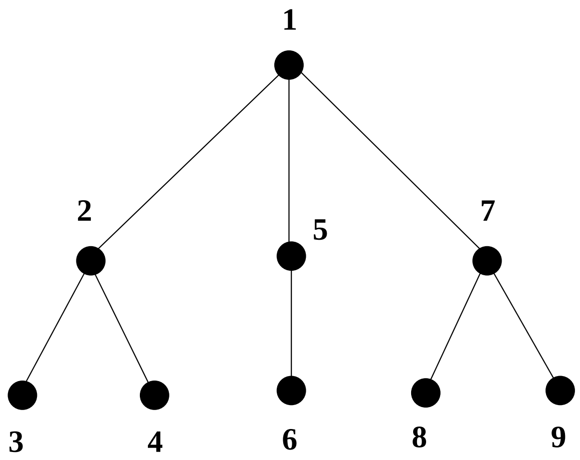
- **visita in profondità** (**depth-first visit**): mi muovo il più possible in profondità (di padre in figlio) e ritorno sui miei passi solo quando non posso più spingermi oltre;
- **visita in ampiezza** (**breadth-first visit**): mi muovo il più possible in ampiezza (di fratello in fratello) e ritorno sui miei passi solo quando non posso più spingermi oltre;

Visite in profondità

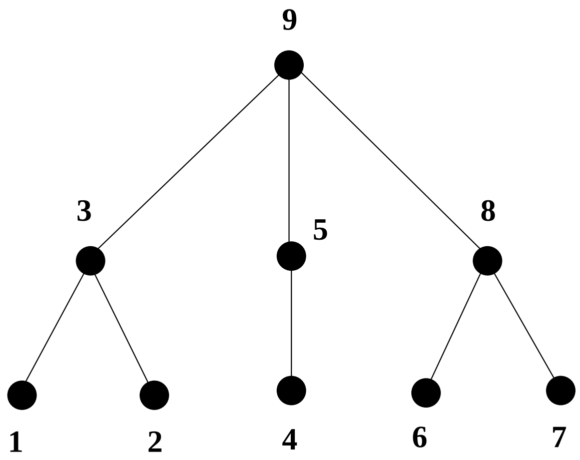
Possiamo visitare in profondità in tre modi:

- L'**ordinamento anticipato** dei nodi di un albero radicato si ottiene ordinando prima la radice e poi i figli da sinistra a destra;
- l'**ordinamento posticipato** dei nodi di un albero radicato si ottiene ordinando prima i figli da sinistra a destra e poi la radice;
- l'**ordinamento intermedio** dei nodi di un albero radicato si ottiene ordinando prima il figlio più a sinistra, poi la radice e infine tutti gli altri figli da sinistra a destra.

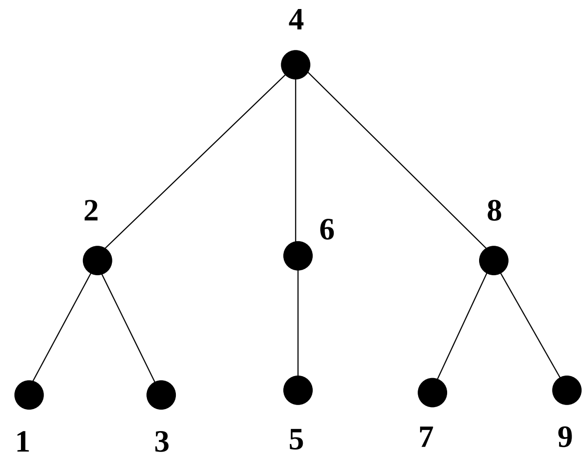
Anticipato



Posticipato



Intermedio



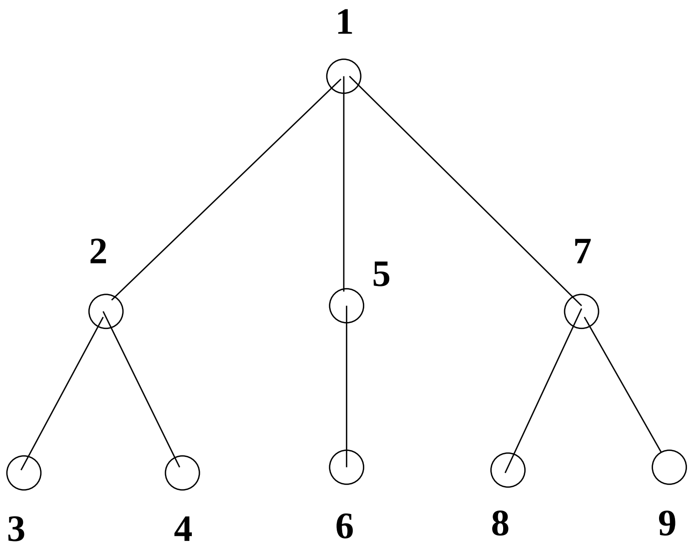
Visita anticipata

PreorderVisit(x)

```
if x ≠ NIL then
    print key[x]
    x ← c[x]
    while x ≠ NIL do
        PreorderVisit(x)
        x ← r[x]
    end while
end if
```

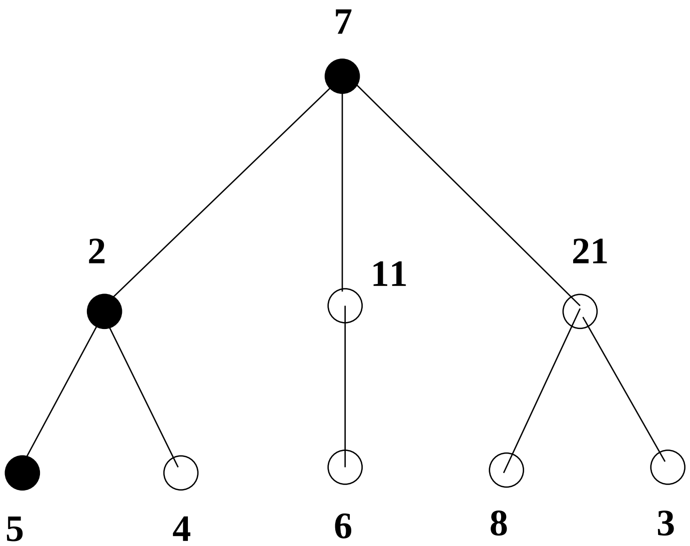
PreorderVisit(x)

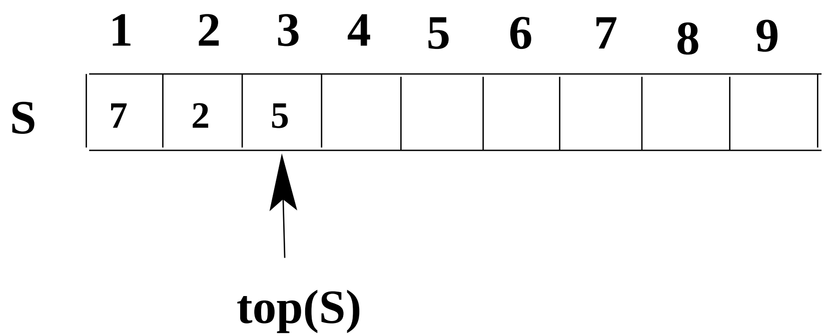
```
S ← ∅
while x ≠ NIL do
    print key[x]
    Push(S, x)
    x ← c[x]
end while
while not StackEmpty(S) do
    x ← r[Pop(S)]
    while x ≠ NIL do
        print key[x]
        Push(S, x)
        x ← c[x]
    end while
end while
```



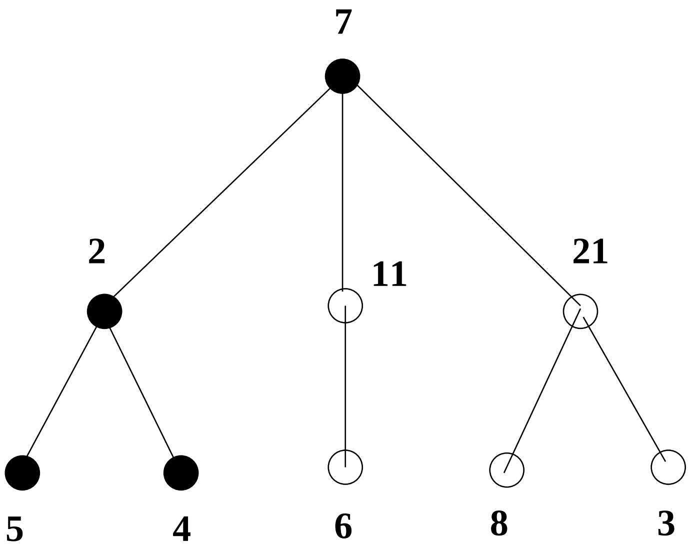
	1	2	3	4	5	6	7	8	9
S									

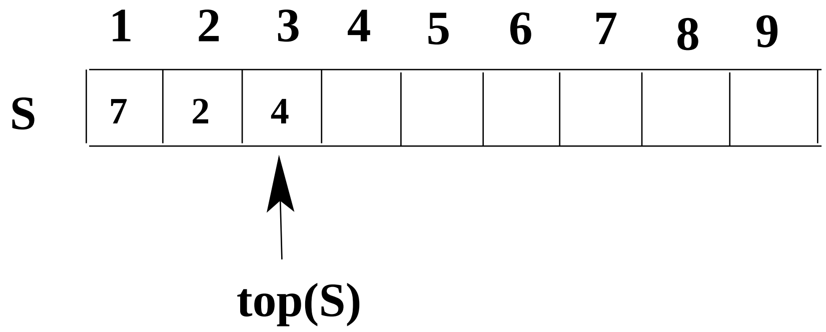
top(S) = 0



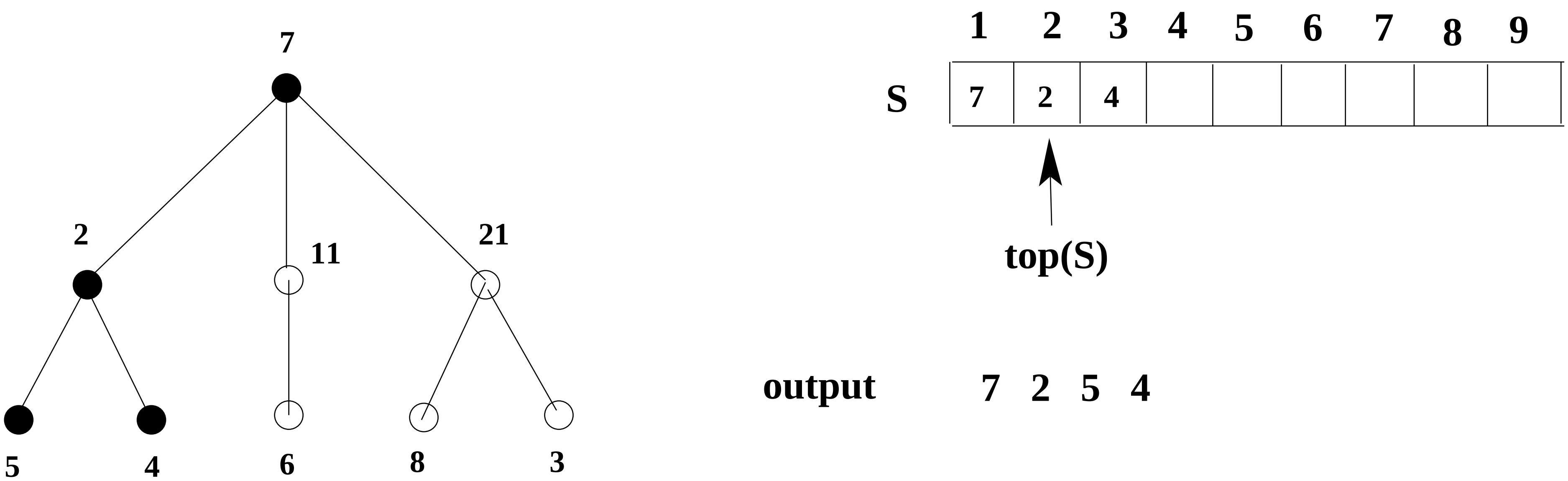


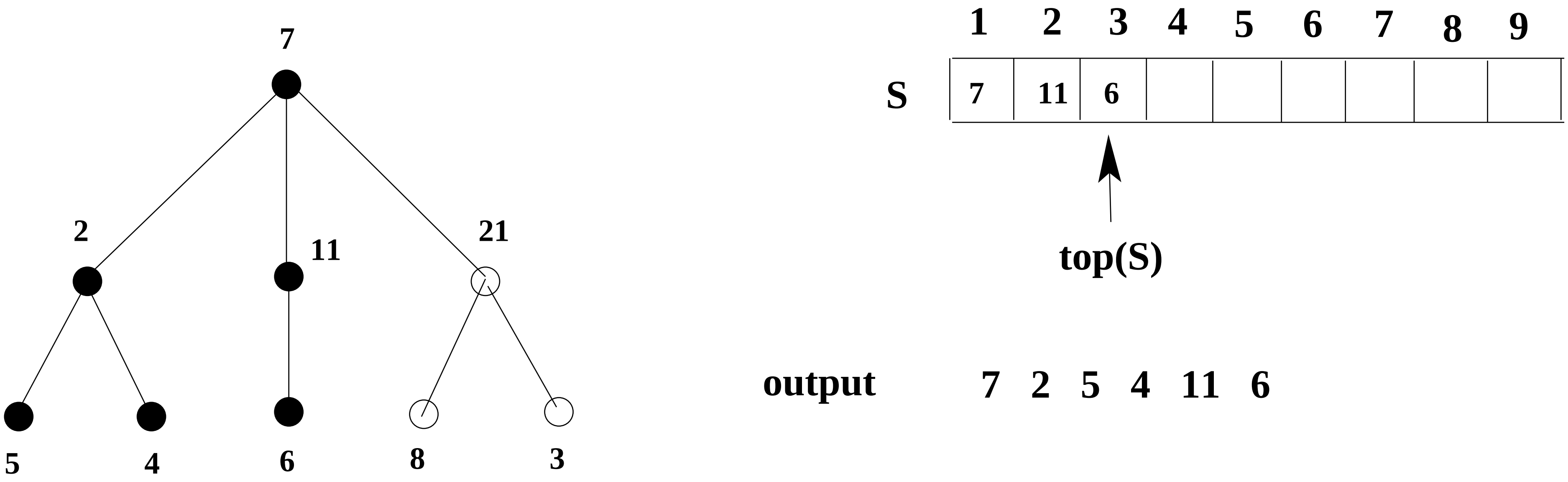
output **7 2 5**

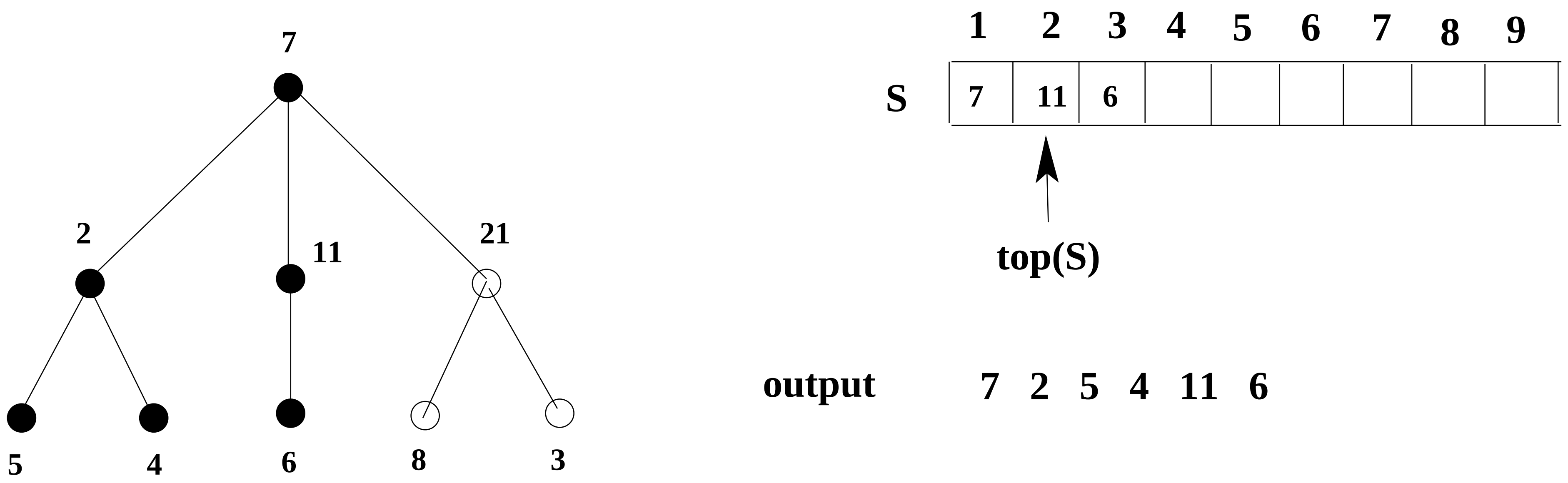


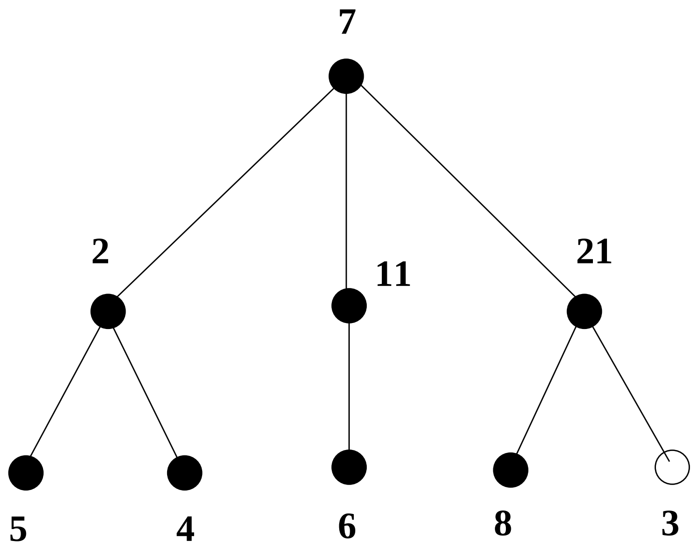


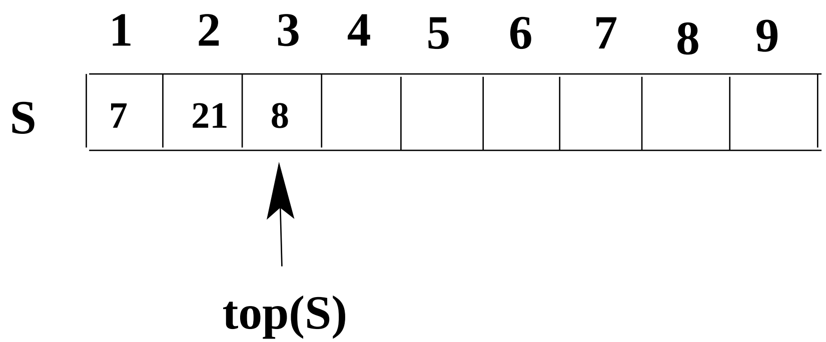
output **7 2 5 4**


7
2
11
21
5
4
6
8
3
1 2 3 4 5 6 7 8 9
S
7 2 4
top(S)
output
7 2 5 4

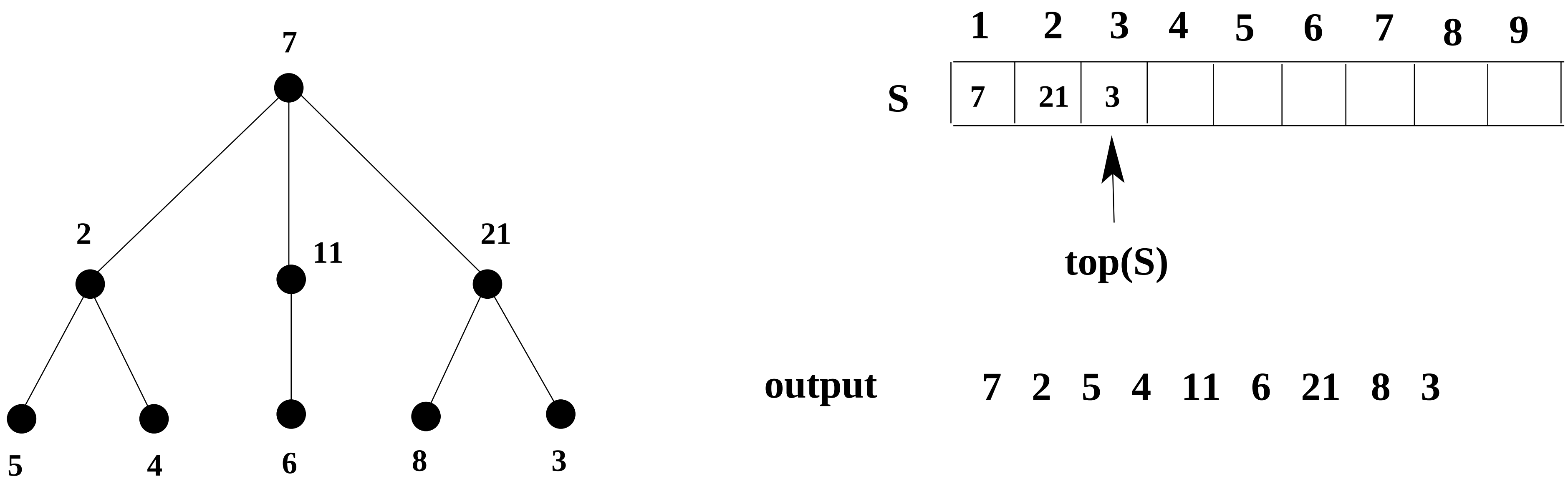

7
2
11
21
5
4
6
8
3
1 2 3 4 5 6 7 8 9
S
7 11 6
top(S)
output 7 2 5 4 11 6

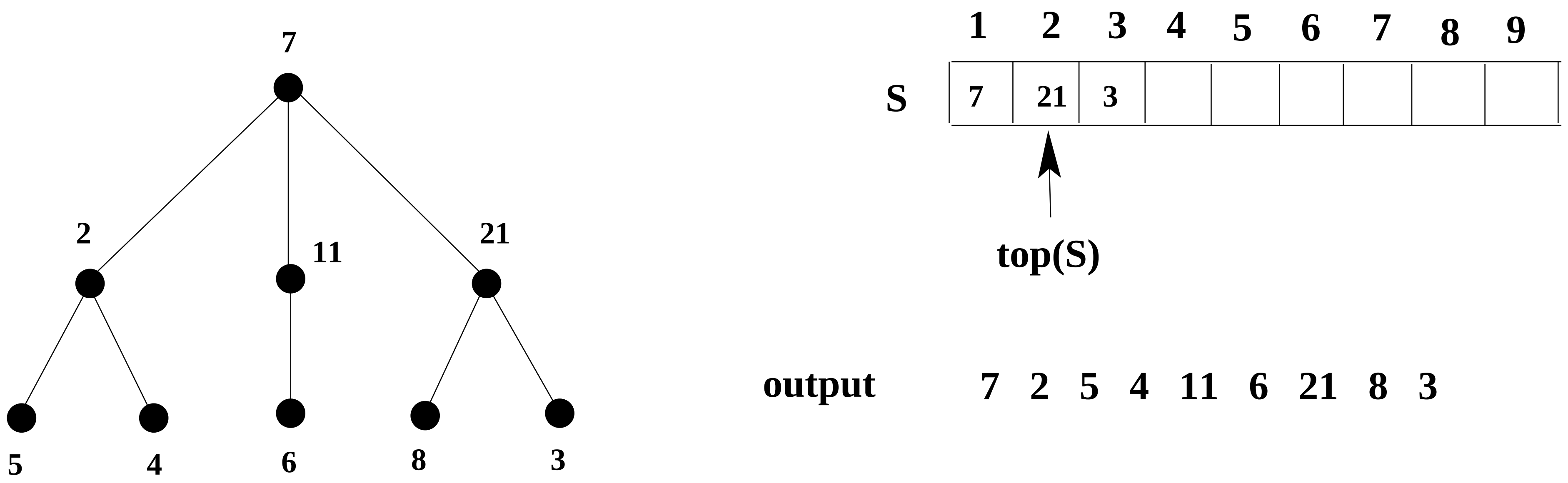

7
2
11
21
5
4
6
8
3
1 2 3 4 5 6 7 8 9
S
7 11 6
top(S)
output 7 2 5 4 11 6

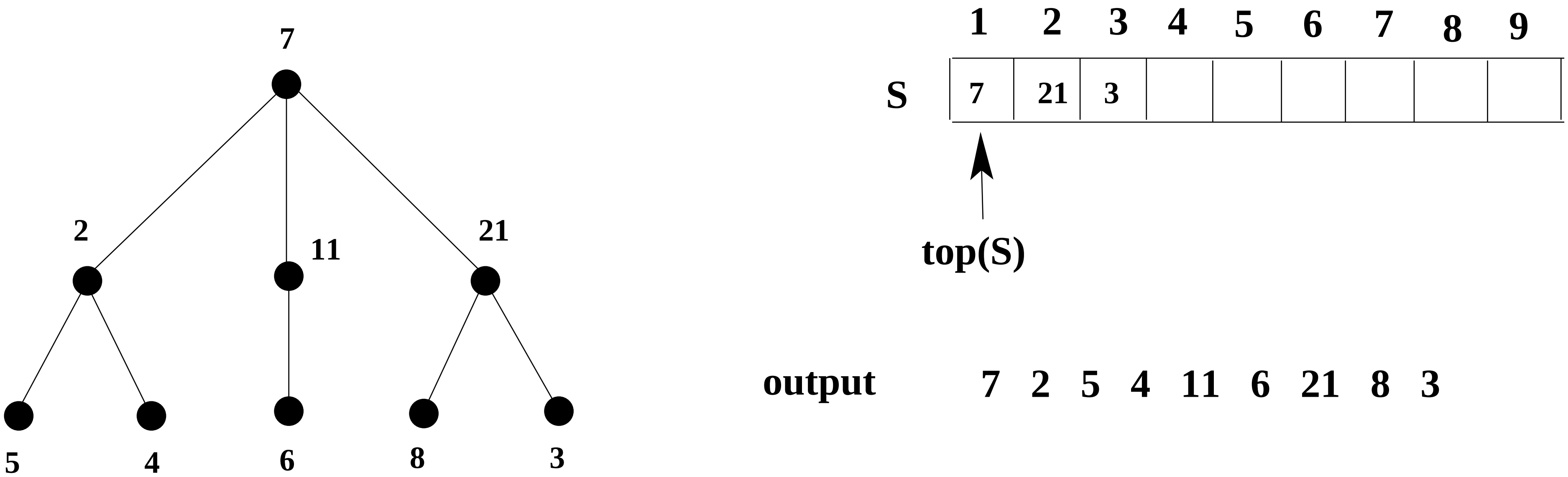


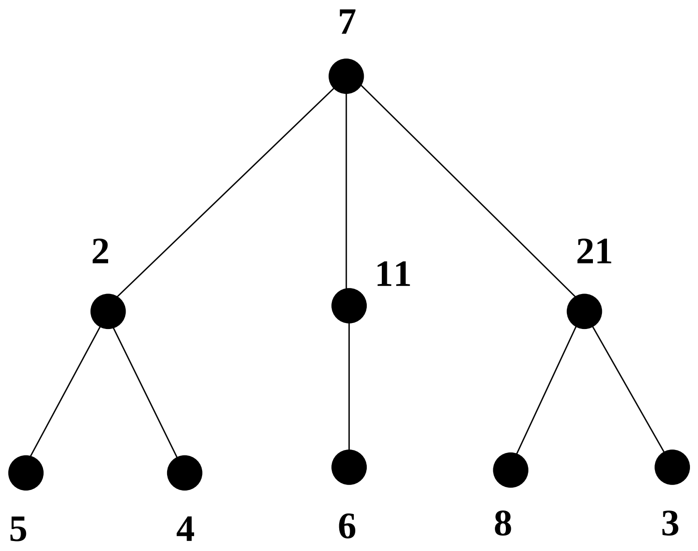


output **7 2 5 4 11 6 21 8**


7
2
11
21
5
4
6
8
3
1 2 3 4 5 6 7 8 9
S
7 21 3
top(S)
output 7 2 5 4 11 6 21 8 3


7
2
11
21
5
4
6
8
3
1 2 3 4 5 6 7 8 9
S
7 21 3
top(S)
output
7 2 5 4 11 6 21 8 3


7
2
11
21
5
4
6
8
3
1 2 3 4 5 6 7 8 9
S
7 21 3
top(S)
output
7 2 5 4 11 6 21 8 3



	1	2	3	4	5	6	7	8	9
S	7	21	3						

top(S) = 0

output **7 2 5 4 11 6 21 8 3**

Complessità della visita anticipata

Ragioniamo in questo modo:

- Ogni nodo viene caricato e scaricato dalla pila esattamente una volta. Le operazioni di *Push* e *Pop* hanno complessità costante;
- la chiave di un nodo viene stampata appena prima aver caricato il nodo sulla pila;

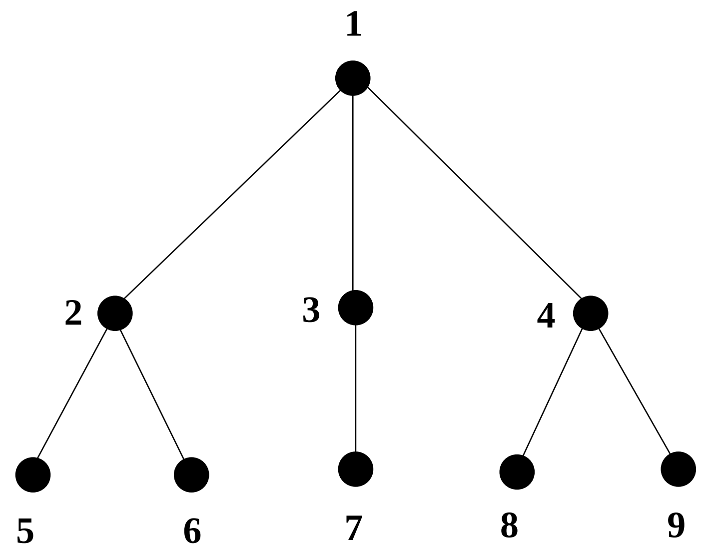
- il figlio sinistro di un nodo viene assegnato appena dopo aver caricato il nodo sulla pila;
- il fratello destro di un nodo viene assegnato appena dopo aver scaricato il nodo dalla pila.

Supponiamo ci siamo n nodi nell'albero. Per ogni nodo faccio una operazione di $Push$, una di Pop e due assegnamenti (del figlio sinistro e del fratello destro).

Quindi la **complessità globale** della visita risulta $\Theta(4n) = \Theta(n)$.

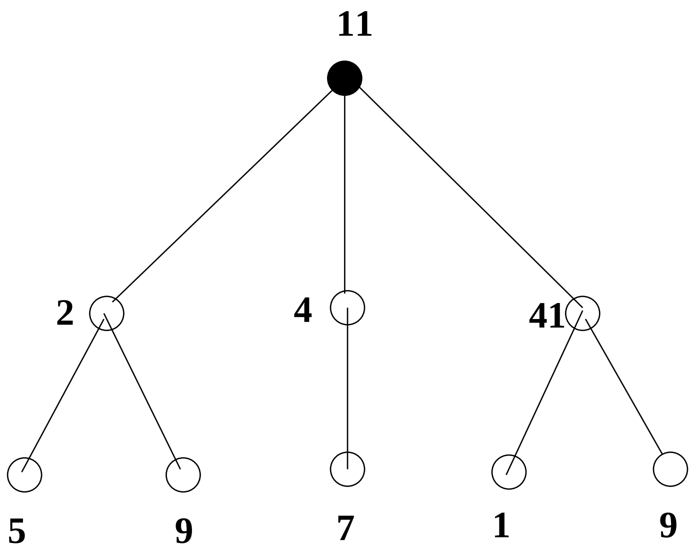
Esercizio Scrivere una versione ricorsiva e una iterativa della visita posticipata e della visita intermedia. La complessità delle procedure deve essere $O(n)$, con n il numero di nodi dell'albero.

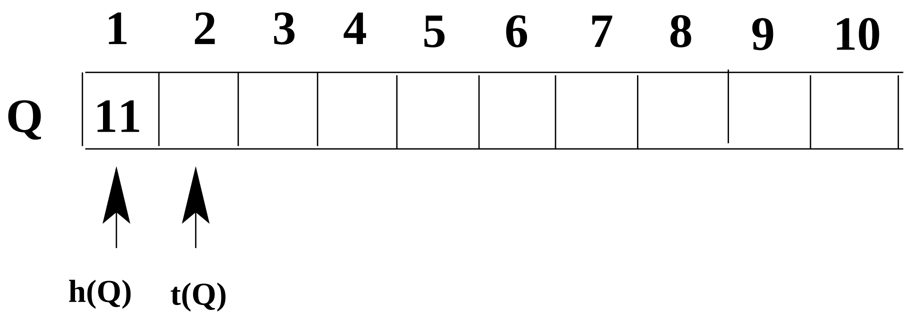
Visita in ampiezza

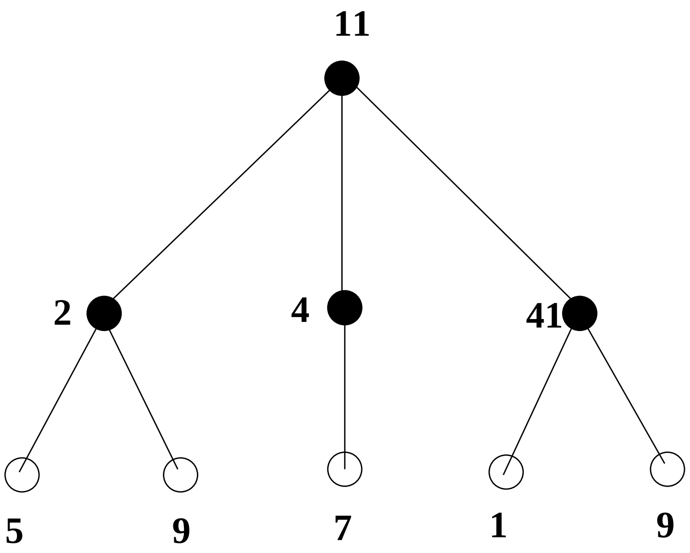


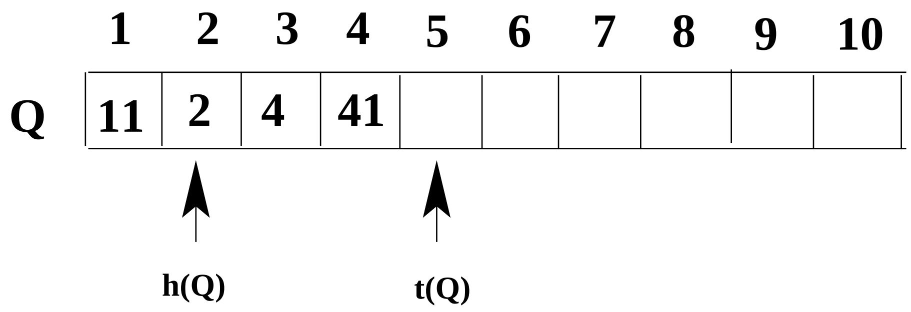
BreadthFirstVisit(x)

```
Q ← ∅
if x ≠ NIL then
    Enqueue(Q, x)
end if
while not QueueEmpty(Q) do
    y ← Dequeue(Q)
    print key[y]
    y ← c[y]
    while y ≠ NIL do
        Enqueue(Q, y)
        y ← r[y]
    end while
end while
```

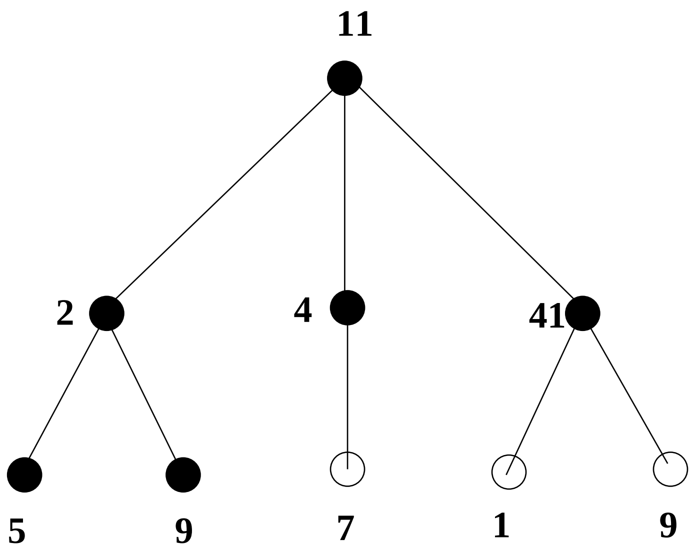

11
2
4
41
5
9
7
1
9

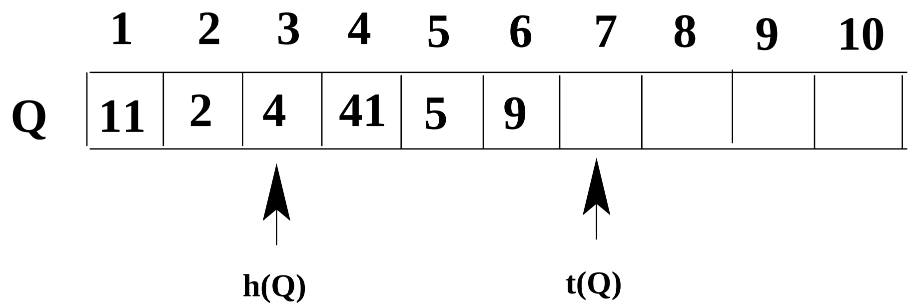

1 2 3 4 5 6 7 8 9 10
Q
11
h(Q)
t(Q)



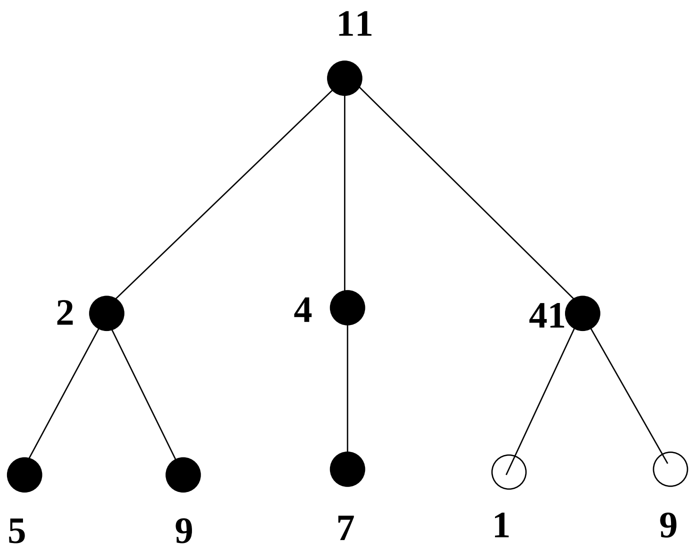


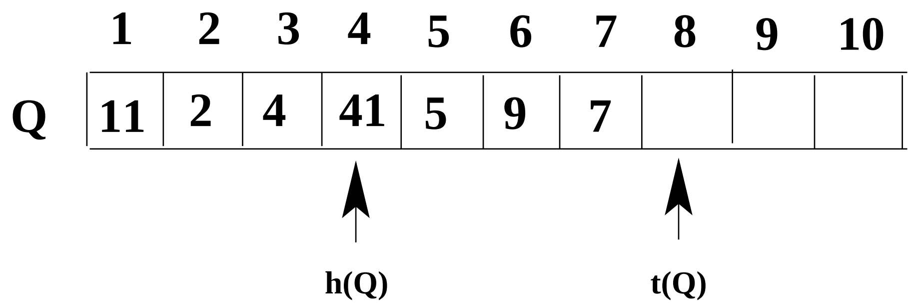
output **11**



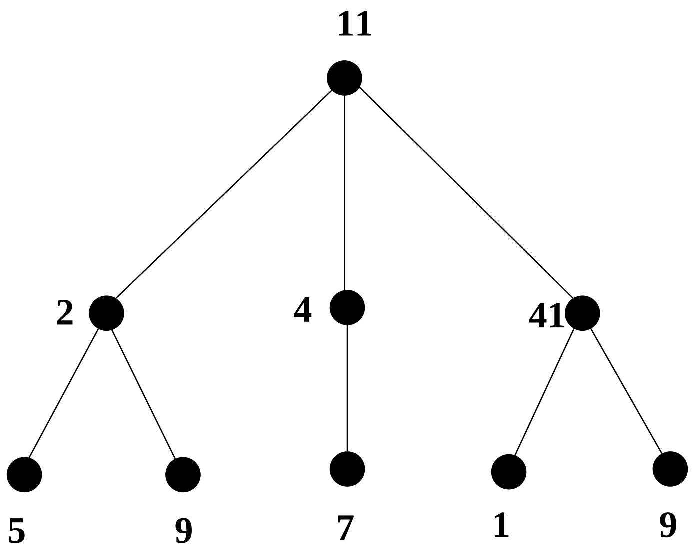


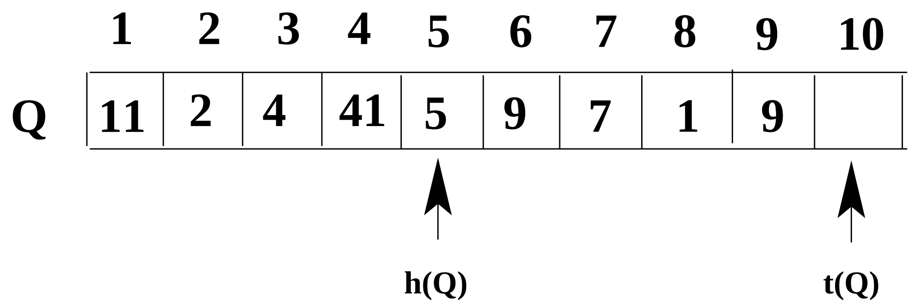
output 11 2



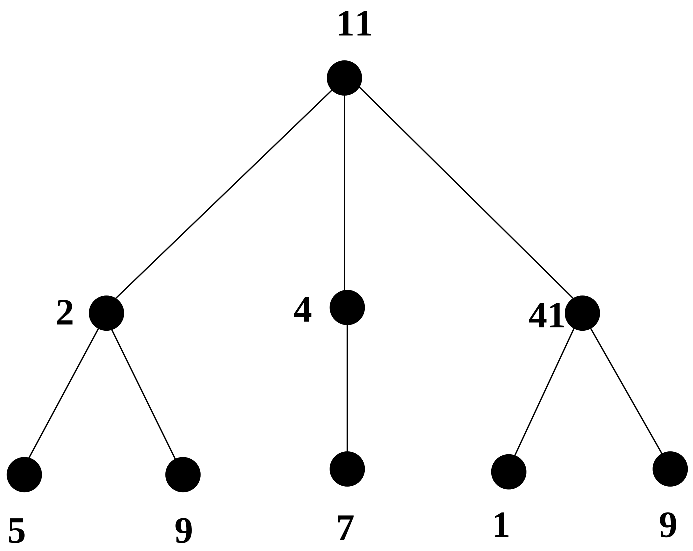


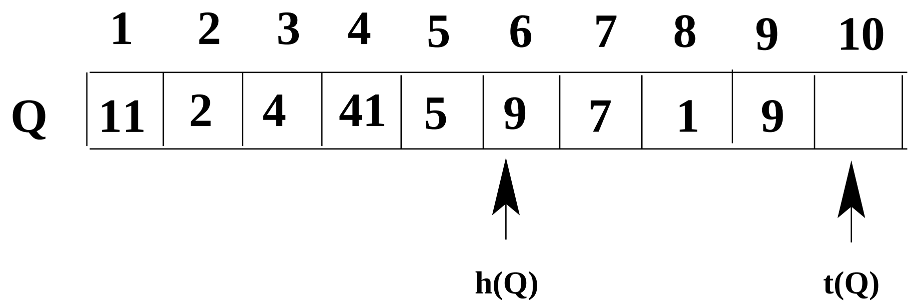
output **11 2 4**



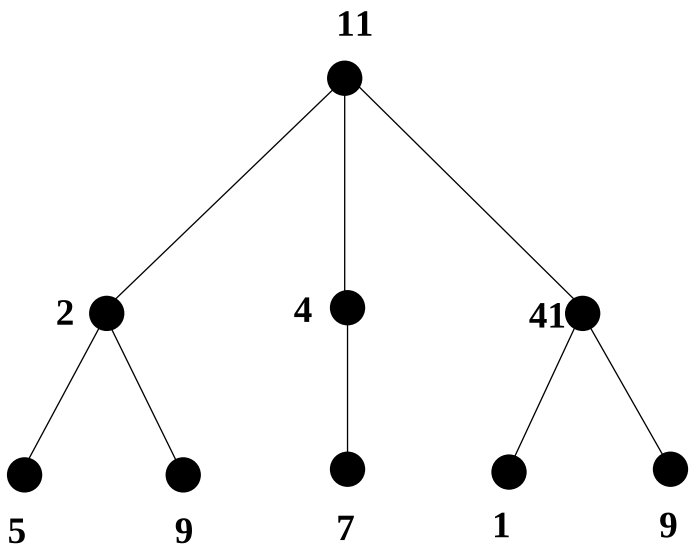


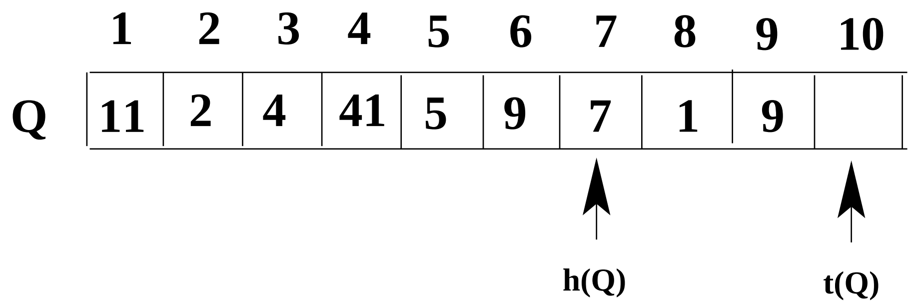
output 11 2 4 41



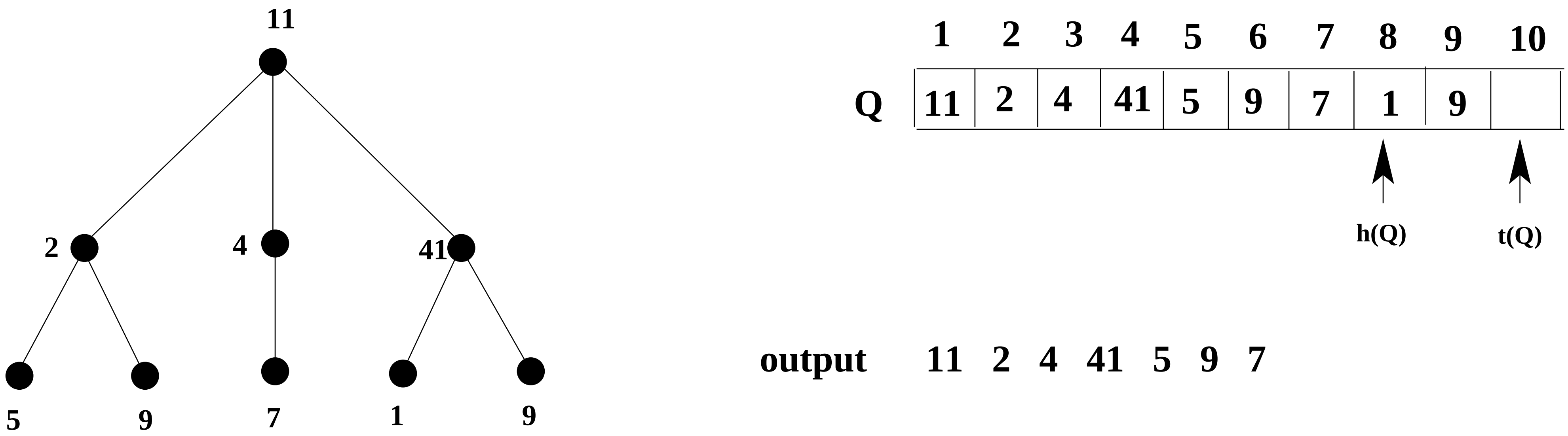


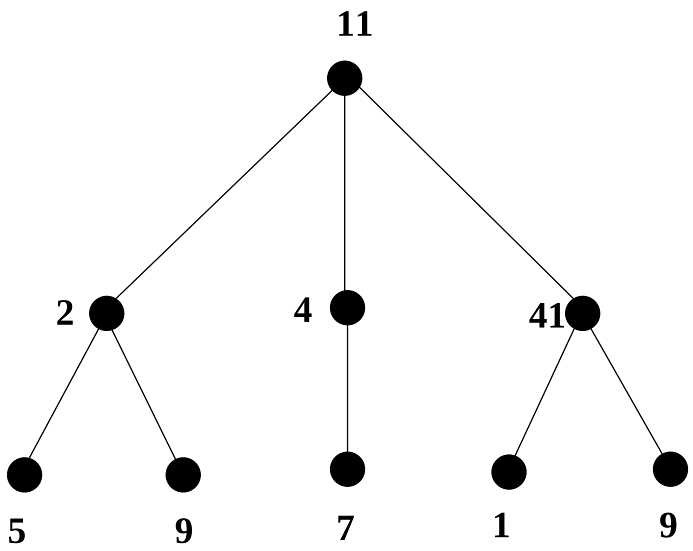
output **11 2 4 41 5**

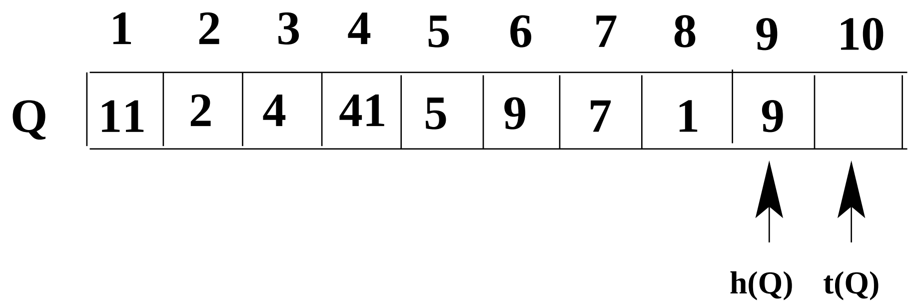




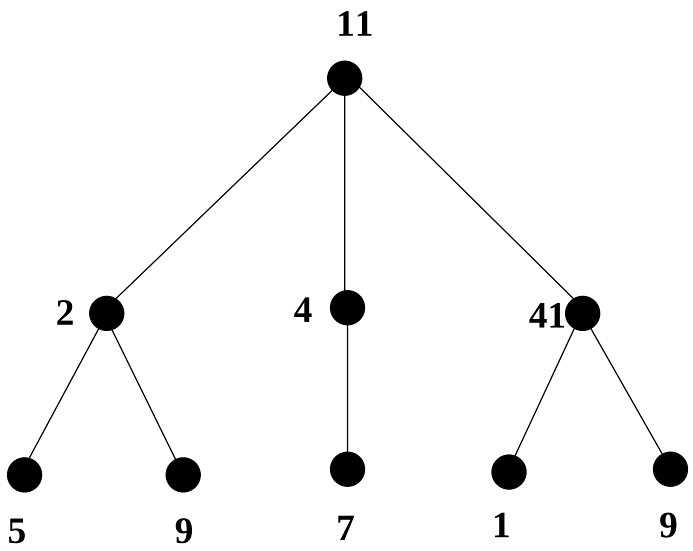
output **11 2 4 41 5 9**


11
2
4
41
5
9
7
1
9
1 2 3 4 5 6 7 8 9 10
Q
11 2 4 41 5 9 7 1 9
h(Q)
t(Q)
output 11 2 4 41 5 9 7





output 11 2 4 41 5 9 7 1



	1	2	3	4	5	6	7	8	9	10
Q	**11**	**2**	**4**	**41**	**5**	**9**	**7**	**1**	**9**	

t(Q)

h(Q)

output **11 2 4 41 5 9 7 1 9**

Complessità della visita anticipata

Ragioniamo in questo modo:

- Ogni nodo viene caricato e scaricato dalla coda esattamente una volta. Le operazioni di *Enqueue* e *Dequeue* hanno complessità costante;
- la chiave di un nodo viene stampata appena dopo aver scaricato il nodo dalla coda;

- il figlio sinistro di un nodo viene assegnato appena dopo aver stampato la chiave del nodo;
- il fratello destro di un nodo viene assegnato appena dopo aver caricato il nodo nella coda.

Supponiamo ci siamo n nodi nell'albero. Per ogni nodo faccio una operazione di *Enqueue*, una di *Dequeue* e due assegnamenti (del figlio sinistro e del fratello destro).

Quindi la **complessità globale** della visita risulta $\Theta(4n) = \Theta(n)$.

Esercizio Scrivere una procedura $TreeInsert(T, x)$ che inserisce nell'albero T il nodo x e una procedura $TreeDelete(T, x)$ che, qualora il nodo x sia diverso dalla radice, cancella x dall'albero T. Calcolare la complessità delle procedure scritte.

Esercizio Modificare opportunamente la rappresentazione degli alberi radicati in modo da poter implementare una versione della procedura $TreeDelete(T, x)$ con complessità costante.

Rappresentazione di alberi binari

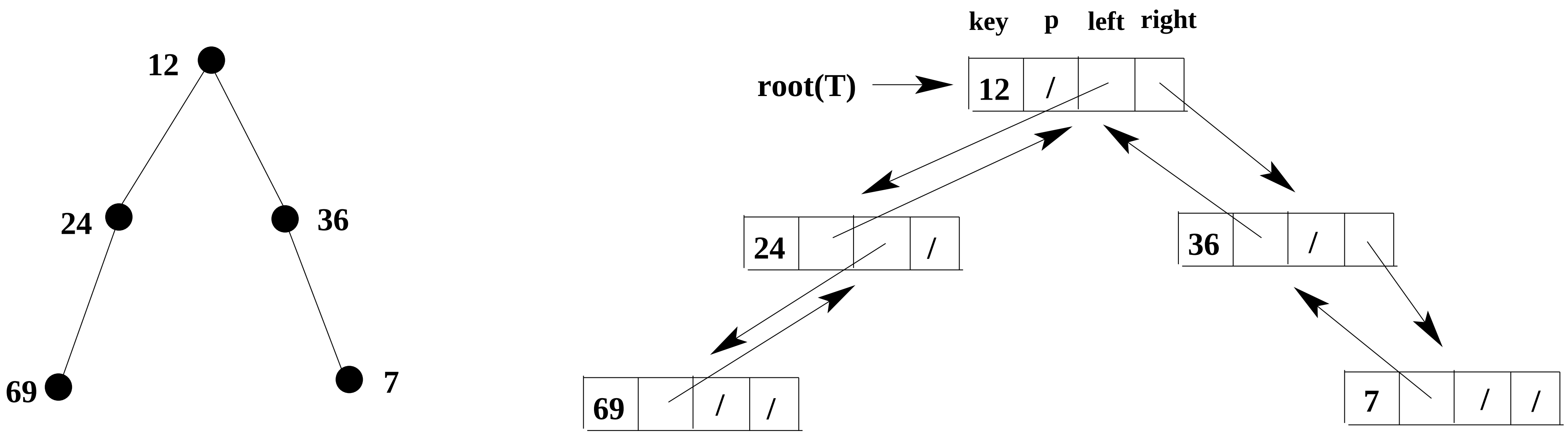
Ogni nodo è rappresentato da un oggetto dotato di quattro campi:

- una **chiave** key;
- un puntatore al **padre** p;
- un puntatore al **figlio sinistro** $left$;
- un puntatore al **figlio destro** $right$.

Se il nodo x è la radice, allora $p[x] = \text{NIL}$, se x non ha un figlio sinistro, allora $left[x] = \text{NIL}$, se x non ha un figlio destro, allora $right[x] = \text{NIL}$.

La struttura di dati albero binario possiede un attributo $root(T)$ che contiene un puntatore alla radice dell'albero T.

Rappresentazione di alberi binari

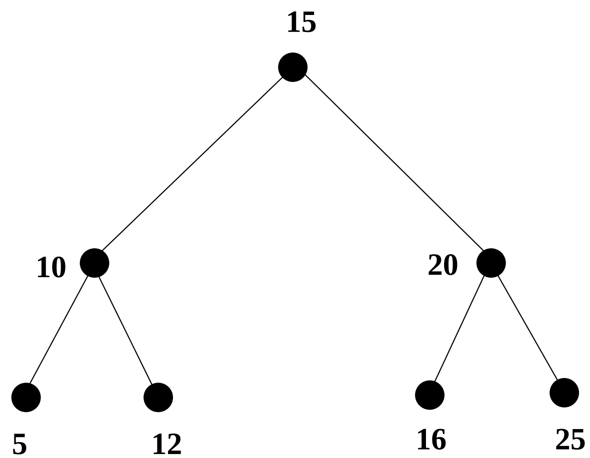


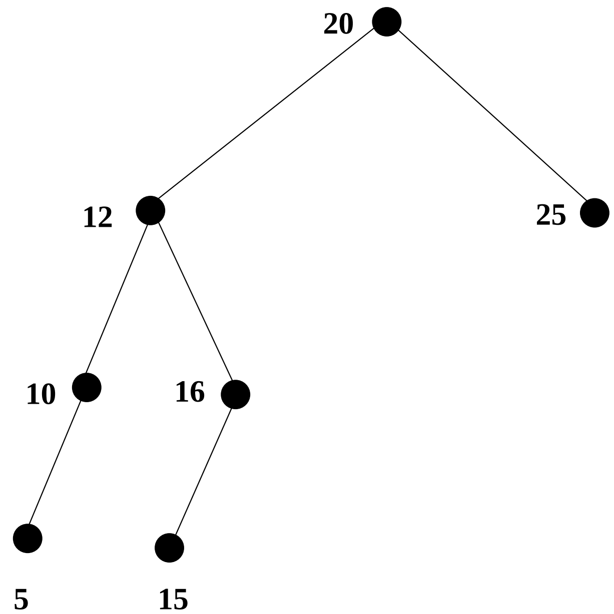
Alberi binari di ricerca (ABR)

Un **albero binario di ricerca** (**binary search tree**) è un albero binario le cui chiavi soddisfano la seguente proprietà:

Se y è un nodo del sottoalbero sinistro di x, allora $key[y] \leq key[x]$.

Se z è un nodo del sottoalbero destro di x, allora $key[z] \geq key[x]$.





Operazioni su ABR

- $TreeEmpty(T)$ che controlla se l'albero T è vuoto;
- $TreeSearch(x, k)$ che ritorna, se esiste, un puntatore all'oggetto con chiave k cercando nell'albero radicato in x, oppure NIL altrimenti;
- $TreeMin(x)$ che ritorna un puntatore all'oggetto con chiave minima dell'albero radicato in x, oppure NIL se l'albero radicato in x è vuoto;
- $TreeMax(x)$ che ritorna un puntatore all'oggetto con chiave massima dell'albero radicato in x, oppure NIL se l'albero radicato in x è vuoto;

- $TreeSuccessor(x)$ che ritorna un puntatore all'oggetto successore di x, oppure NIL se x è il massimo;
- $TreePredecessor(x)$ che ritorna un puntatore all'oggetto predecessore di x, oppure NIL se x è il minimo;
- $TreeInsert(T, x)$ che inserisce l'oggetto puntato da x nell'albero T mantenendo la proprietà degli ABR;
- $TreeDelete(T, x)$ che cancella l'oggetto puntato da x dall'albero T mantenendo la proprietà degli ABR.

TreeEmpty(T)

```
1: if root(T) = NIL then
2:    return TRUE
3: else
4:    return FALSE
5: end if
```

La complessità è costante $\Theta(1)$.

TreeSearch(x,k)

```
if x = NIL then
    return NIL
end if
if k = key[x] then
    return x
end if
if k < key[x] then
    return TreeSearch(left[x], k)
else
    return TreeSearch(right[x], k)
end if
```

La complessità è **lineare nell'altezza** dell'albero radicato in x, cioè $\Theta(h)$, con h l'altezza dell'albero.

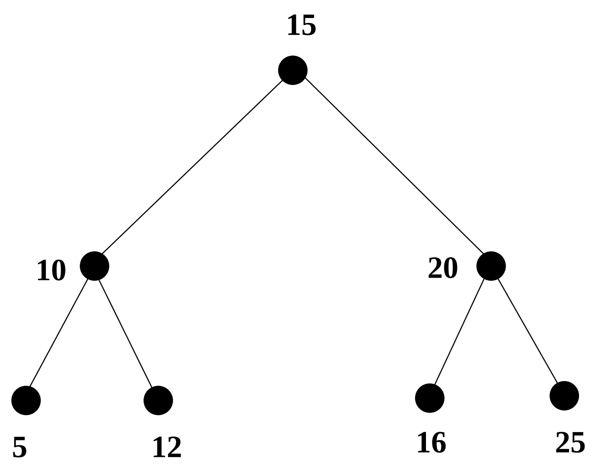
TreeSearch(x,k)

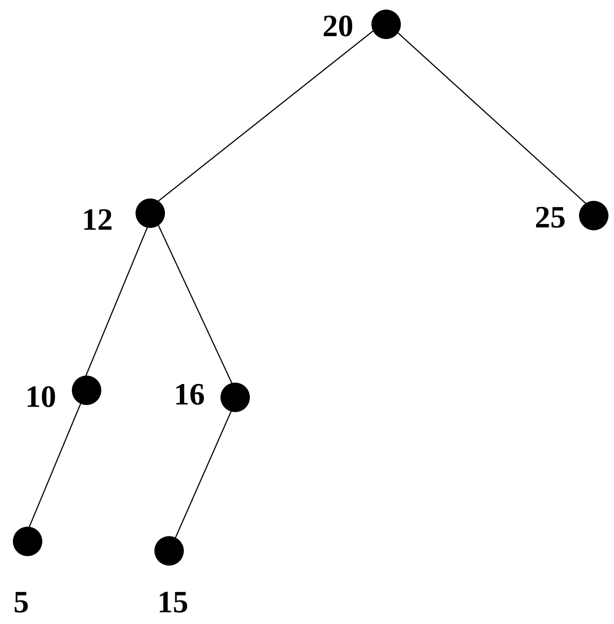
```
while x ≠ NIL and key[x] ≠ k do
    if k < key[x] then
        x ← left[x]
    else
        x ← right[x]
    end if
end while
return x
```

Minimo e Massimo

Teorema *Il minimo di un ABR è l'ultimo nodo del cammino più a sinistra che parte dalla radice.*

Il massimo di un ABR è l'ultimo nodo del cammino più a destra che parte dalla radice.





TreeMin(x)

```
if x = NIL then
    return NIL
end if
while left[x] ≠ NIL do
    x ← left[x]
end while
return x
```

TreeMin(x)

```
if x = NIL then
    return NIL
end if
if left[x] = NIL then
    return x
else
    TreeMin(left[x])
end if
```

TreeMax(x)

```
if x = NIL then
    return NIL
end if
while right[x] ≠ NIL do
    x ← right[x]
end while
return x
```

TreeMax(x)

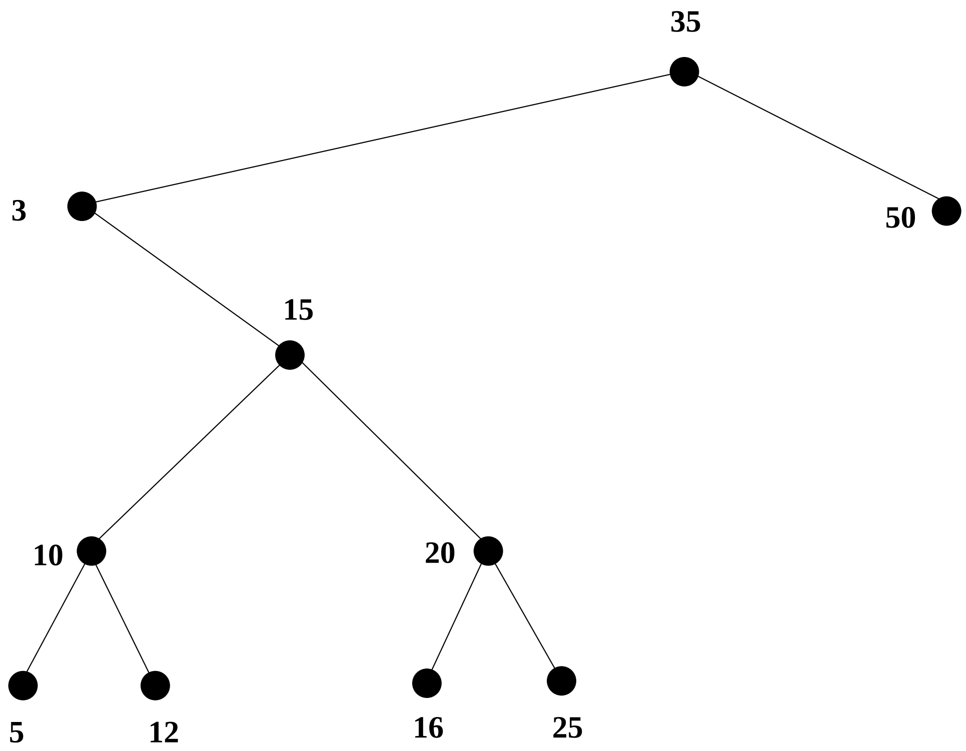
```
if x = NIL then
    return NIL
end if
if right[x] = NIL then
    return x
else
    TreeMax(right[x])
end if
```

Le procedure $TreeMin(x)$ e $TreeMax(x)$ hanno complessità pessima $\Theta(h)$, ove h è l'altezza dell'albero radicato in x.

Successore e Predecessore

Teorema *Il successore di un nodo x, se esiste, è il minimo del sottoalbero di destra del nodo x, se tale albero non è vuoto. Altrimenti è il più basso antenato di x il cui figlio sinistro è un antenato di x.*

Il predecessore di un nodo x, se esiste, è il massimo del sottoalbero di sinistra del nodo x, se tale albero non è vuoto. Altrimenti è il più basso antenato di x il cui figlio destro è un antenato di x.


35
3
50
15
10
20
5
12
16
25

TreeSuccessor(x)

```
if right[x] ≠ NIL then
    return TreeMin(right[x])
end if
y ← p[x]
while (y ≠ NIL) and (x = right[y]) do
    x ← y
    y ← p[y]
end while
return y
```

TreePredecessor(x)

```
if left[x] ≠ NIL then
    return TreeMax(left[x])
end if
y ← p[x]
while (y ≠ NIL) and (x = left[y]) do
    x ← y
    y ← p[y]
end while
return y
```

Le procedure $TreeSuccessor(x)$ e $TreePredecessor(x)$ hanno complessità pessima $\Theta(h)$, ove h è l'altezza dell'albero che contiene x.

Inserimento

```
TreeInsert(T,z) // p[z] = left[z] = right[z] = NIL
y ← NIL
x ← root(T)
while x ≠ NIL do
    y ← x
    if key[z] < key[x] then
        x ← left[x]
    else
        x ← right[x]
    end if
end while
```

$p[z] \leftarrow y$
if $y = \text{NIL}$ **then**
    $root(T) \leftarrow z$
else
    if $key[z] < key[y]$ **then**
        $left[y] \leftarrow z$
    else
        $right[y] \leftarrow z$
    end if
end if

La procedura $TreeInsert(T, z)$ inserisce il nodo z sempre come foglia. La sua complessità pessima è dunque $\Theta(h)$, ove h è l'altezza dell'albero T.

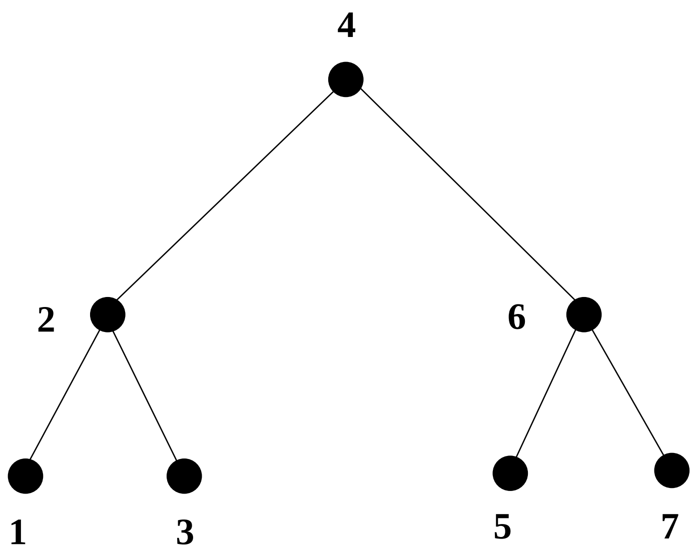
Esercizio.

1. *Trovare una sequenza di 7 chiavi il cui inserimento con la procedura* TreeInsert *genera un albero binario completo.*

2. *Trovare una sequenza di 7 chiavi il cui inserimento con la procedura* TreeInsert *genera un albero lineare.*

3. *Qual è il la complessità ottima e pessima di n inserimenti con la procedura* TreeInsert*?*

1. La sequenza è $\langle 4, 2, 6, 1, 3, 5, 7 \rangle$.



2. La sequenza è $\langle 1, 2, 3, 4, 5, 6, 7 \rangle$ oppure la sua inversa. In generale ogni sequenza di chiavi oridinate in qualche senso genera un albero lineare.

3. Nel caso pessimo, l'albero T risultante dopo gli n inserimenti è lineare, cioè la sequenza di inserimenti è ordinata. L'i-esimo inserimento avviene su un albero di altezza $i - 2$.

   La complessità di $TreeInsert$ è dell'ordine dell'altezza dell'albero in cui il nodo viene inserito, cioè $\Theta(i - 2) = \Theta(i)$.

   Dunque la **complessità pessima** dell'inserimento delle n chiavi nell'albero T è pari a:

$$\sum_{i=1}^{n} \Theta(i) = \Theta(\sum_{i=1}^{n} i) = \Theta(n(n+1)/2) = \Theta(n^2).$$

Nel caso ottimo, l'albero T risultante dopo gli n inserimenti è un albero completo fino al penultimo livello. L'i-esimo inserimento avviene su un albero di altezza $\lfloor \log i \rfloor - 1$.

La complessità di $TreeInsert$ è dell'ordine dell'altezza dell'albero in cui il nodo viene inserito, cioè $\Theta(\lfloor \log i \rfloor - 1) = \Theta(\log i)$.

Dunque la **complessità ottima** dell'inserimento delle n chiavi nell'albero T è pari a

$$\sum_{i=1}^{n} \Theta(\log i) = \Theta(\sum_{i=1}^{n} \log i) = \Theta(\log \prod_{i=1}^{n} i) = \Theta(\log n!) = \Theta(n \log n).$$

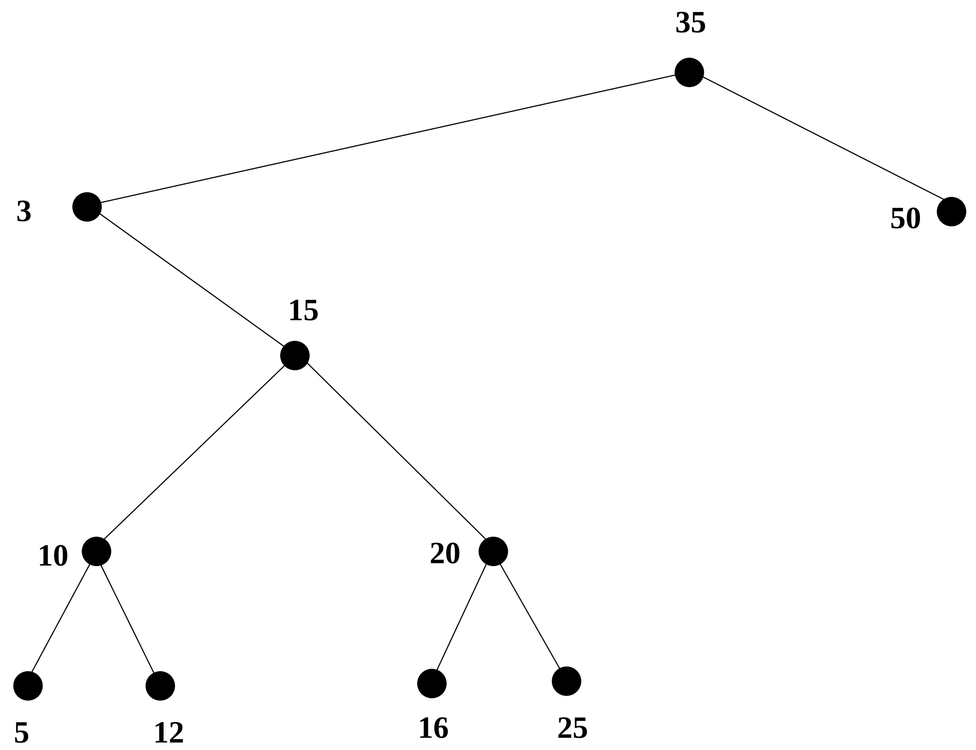
Esercizio *Sia A un vettore. Si consideri la seguente procedura di ordinamento $TreeSort(A)$. La procedura consiste di due passi:*

1. *gli elementi di A vengono inseriti in un ABR T usando la procedura $TreeInsert$;*

2. *l'albero T viene visitato in ordine intermedio e gli elementi di T vengono reinseriti nel vettore A.*

Argomentare la correttezza dell'algoritmo e calcolarne la complessità ottima e pessima.

L'algoritmo è corretto perchè la visita intermedia visita prima a sinistra, poi la radice, e poi a destra, quindi, a causa alla proprietà degli ABR, in ordine crescente.



Sia n la lunghezza di A. Nel **caso pessimo**, la complessità dell'inserimento degli n elementi di A nell'albero T è pari a $\Theta(n^2)$. La complessità pessima della visita intermedia di T è $\Theta(n)$. Dunque la complessità pessima di $TreeSort$ risulta $\Theta(n^2 + n) = \Theta(n^2)$.

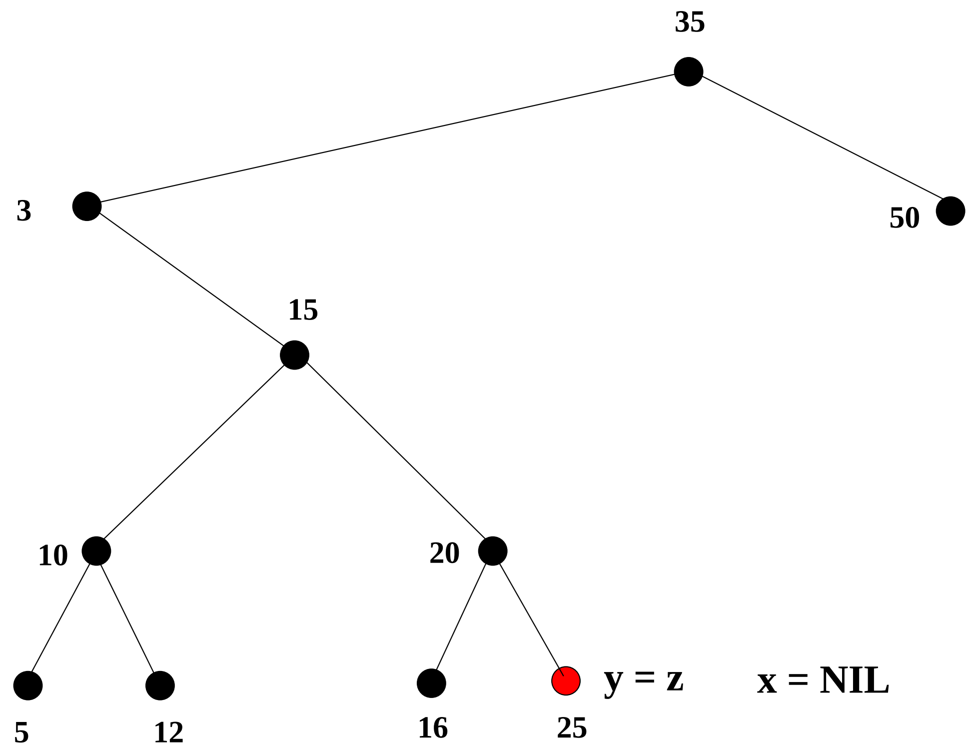
Nel **caso ottimo**, la complessità dell'inserimento degli n elementi di A nell'albero T è pari a $\Theta(n \log n)$. La complessità ottima della visita intermedia di T è $\Theta(n)$. Dunque la complessità ottima di $TreeSort$ risulta $\Theta(n \log n + n) = \Theta(n \log n)$.

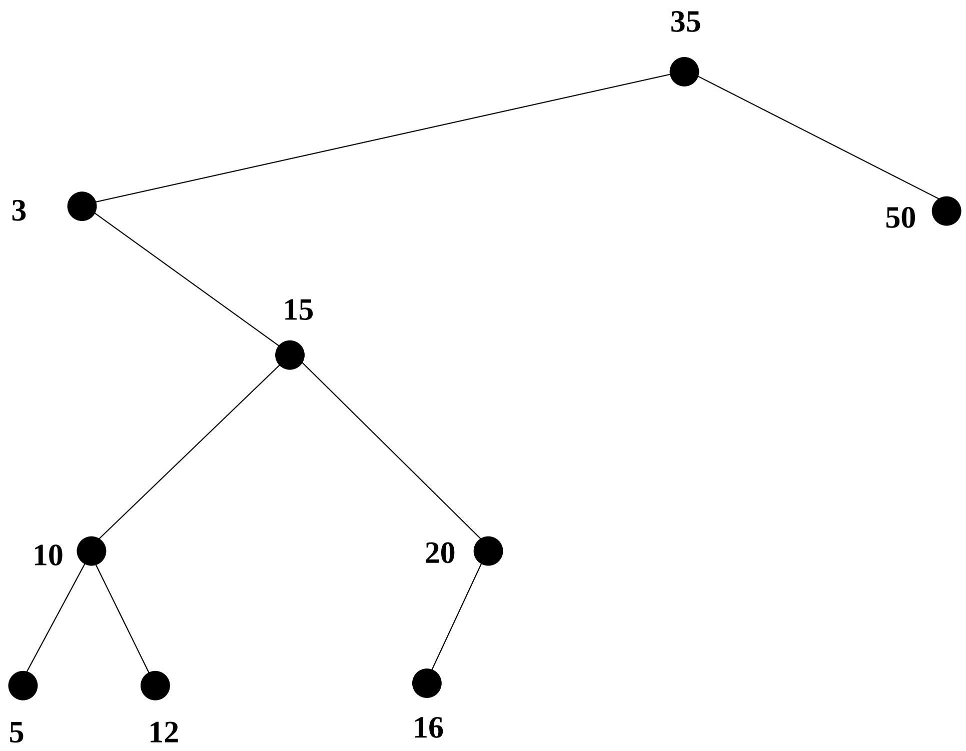
Cancellazione

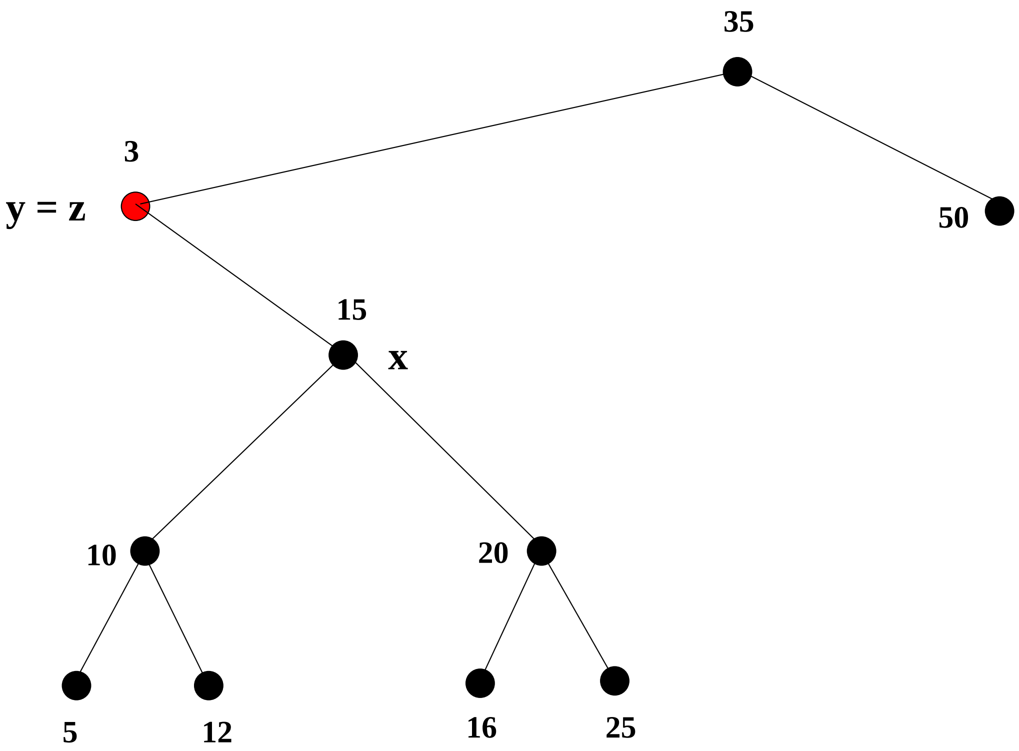
TreeDelete(T,z)
if $(left[z] = \textsc{nil})$ **or** $(right[z] = \textsc{nil})$ **then**
    $y \leftarrow z$
else
    $y \leftarrow TreeSuccessor(z)$
end if
if $left[y] \neq \textsc{nil}$ **then**
    $x \leftarrow left[y]$
else
    $x \leftarrow right[y]$
end if
if $x \neq \textsc{nil}$ **then**
    $p[x] \leftarrow p[y]$
end if

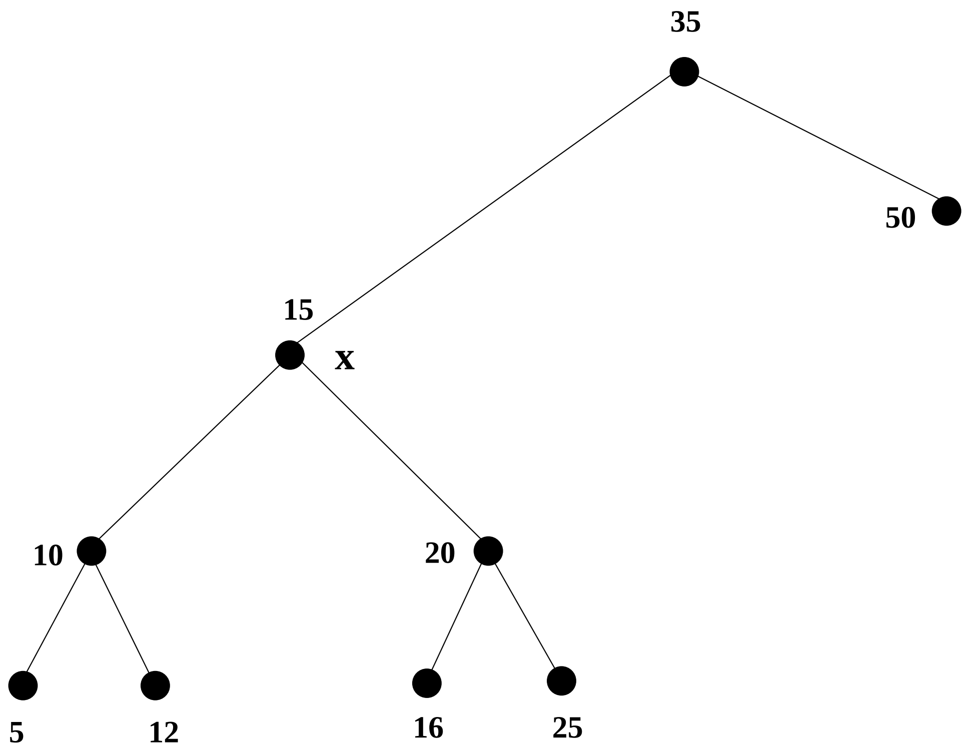
if $p[y] = \text{NIL}$ **then**
    $root(T) \leftarrow x$
else
    if $y = left[p[y]]$ **then**
        $left[p[y]] \leftarrow x$
    else
        $right[p[y]] \leftarrow x$
    end if
end if
if $y \neq z$ **then**
    $key[z] \leftarrow key[y]$
    // copia i dati satellite di y in z
end if
return y

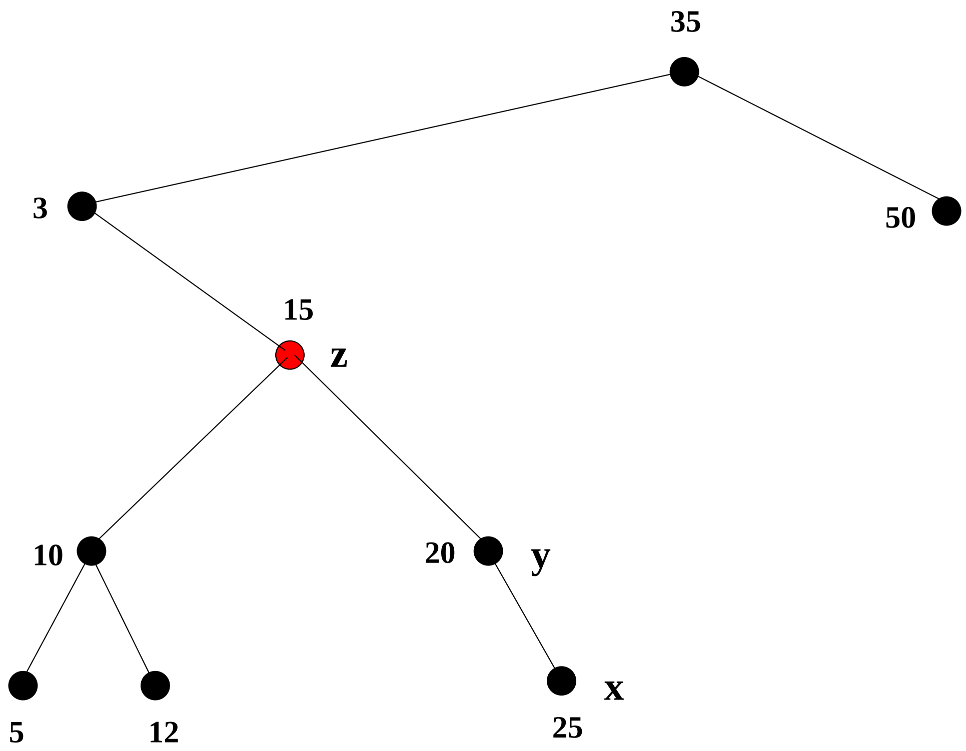
La complessità di $TreeDelete(T, z)$ è costante se z non ha entrambi i figli, altrimenti è $\Theta(h)$, ove h è l'altezza dell'albero T.

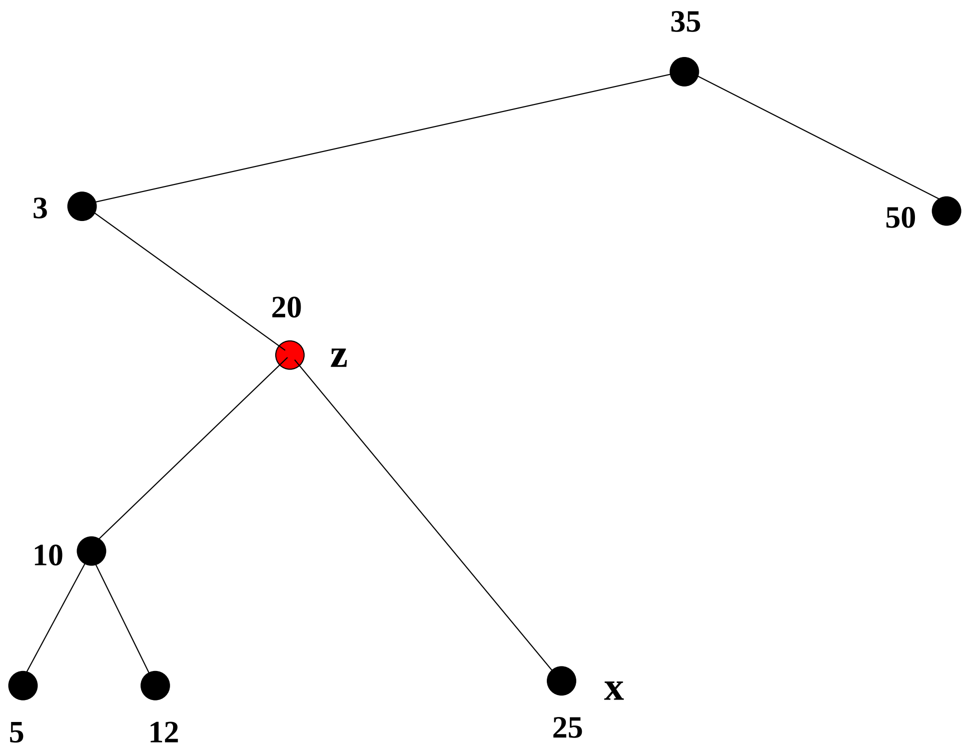

35
3
50
15
10
20
5
12
16
25
y = z
x = NIL


35
3
50
15
10
20
5
12
16


35
3
y = z
50
15
x
10
20
5
12
16
25


35
50
15
x
10
20
5
12
16
25


35
3
50
15
z
10
20
y
5
12
25
x


35
3
50
20
z
10
5
12
25
x

Complessità delle operazioni su ABR

Le operazioni di ricerca e di modifica sugli ABR hanno complessità $\Theta(h)$, ove h è l'altezza dell'albero. Se n è il numero di nodi dell'albero, allora

$$h = \Omega(\log n)$$

e

$$h = O(n).$$

Nel **caso ottimo**, l'altezza è logaritmica nel numero di nodi, e dunque la complessità ottima di tutte le operazioni risulta logaritmica.

Nel **caso pessimo**, l'altezza è lineare nel numero dei nodi, e dunque la complessità pessima di tutte le operazioni risulta lineare (non meglio delle operazioni su liste).

Qual è la complessità media?

Teorema *Supponiamo di avere* n *chiavi distinte a disposizione. Scegliamo casualmente una permutazione delle* n *chiavi e inseriamo la permutazione in un albero binario inzialmente vuoto usando la procedura* TreeInsert*. Allora l'altezza media dell'albero risultante è* $O(\log n)$*.*

Questo ci permette di concludere che la **complessità media** di tutte le operazioni su ABR è logaritmica nel numero dei nodi.