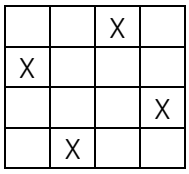
Note dalle lezioni di Programmazione Orientata agli Oggetti



N regine = 4 regine

<0,1><1,3><2,0><3,2>
<0,2><1,0><2,3><3,1>

Backtracking – tecnica di programmazione (risoluzione di problemi) per tentativi.

E' una tecnica enumerativa di tutte le possibili soluzioni, che esplora, cioè, esaustivamente tutto lo spazio delle possibili soluzioni.

PRIMO PASSO: schematizzare il problema in *punti-di-scelta* e *scelte*.

```
void tentativo( PS ps ){
        for( tutte le scelte s ammissibili su ps ){
                if( assegnabile(ps,s) ){
                        assegna(ps,s); //assegna s a ps
                        if( ps è l'ultimo punto di scelta ) scriviSoluzione();
                        else tentativo( succ(ps) );
                        deassegna(ps,s); //deassegna s da ps
                }
        }
}
```

Sviluppando tentativo(ps) in tutti i casi possibili, si generano tutte le possibili soluzioni, che vengono scritte su output.

Complessità di MergeSort

$T(n)=2T(n/2)+n$ a meno di qualche costante

$T(n)/n=T(n/2)/(n/2)+1$
$T(n/2)/(n/2)=T(n/4)/(n/4)+1$
$T(n/4)/(n/4)=T(n/8)/(n/8)+1$
…
$T(4)/4=T(2)/2+1$
$T(2)/2=T(1)/1+1$

Il numero delle equazioni è pari alle suddivisioni di n, ossia $\log_2 n$.

La somma di tutti i primi membri, sarà ovviamente uguale alla somma di tutti i secondi membri (somma telescopica). Eliminando i termini che sono sia al primo che al secondo membro della sommatoria, sia:

$T(n)/n=T(1)+\log n$

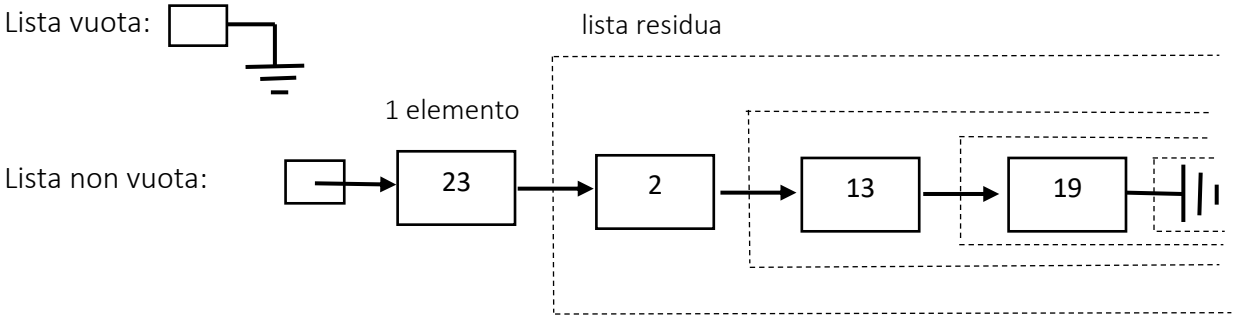
$T(n)=nT(1)+n \log n=n+ n \log n$

Facendo l'analisi asintotica, $n->\infty$, si ottiene: $T(n)=O(n*\log n)$

Gestione ricorsiva di una lista concatenata ordinata

Una lista concatenata (ma anche una lista in generale) ammette una naturale definizione ricorsiva, che traspare anche dalla definizione ricorsiva della classe Nodo. Infatti, una lista o è vuota o consiste di un primo elemento seguito da una lista (la cosiddetta lista residua).

$$lista = \begin{cases} vuota\ (nessun\ elemento), \ \ oppure \\ primo\ nodo, \ \ seguito\ da\ una\ lista\ (lista\ residua) \end{cases}$$

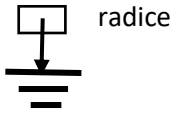


La struttura ricorsiva della lista si può sfruttare nell'elaborazione di varie operazioni. Es. dovendo cercare un elemento x, se la lista è vuota, la ricerca fallisce. Altrimenti ci si può confrontare col primo elemento. Se esso è uguale ad x, la ricerca termina con successo. Diversamente, la ricerca può essere continuata sulla lista residua, dove si possono applicare esattamente gli stessi ragionamenti.

Creazione di un albero binario di ricerca (ABR)

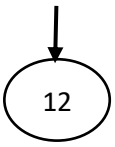
Si consideri la seguente successione di numeri: 12 34 -2 -4 5 1 7 38 -6 8

Si desidera inserire i numeri in un ABR, che inizialmente è vuoto.



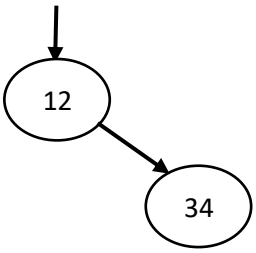
Inserimento di 12.

Siccome l'albero è vuoto, si ha banalmente (i campi fS e fD del nodo 12 sono null e non si esplicitano per semplicità):



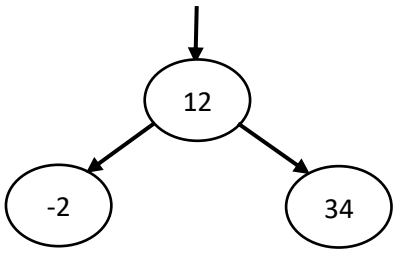
Inserimento di 34.

Confrontando 34 con il nodo radice, si vede che 34 va inserito a destra di 12. Siccome a destra di 12 c'è un sotto albero vuoto, si crea un nuovo nodo con info=34 e lo si connette come fD di 12. Il nodo 34 ha entrambi i figli fS e fD a null.



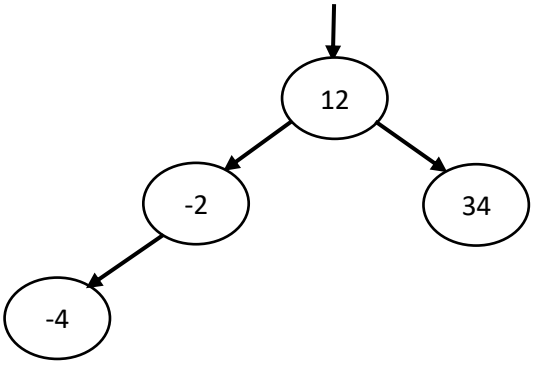
Inserimento di -2.

Confrontandosi con la radice, si capisce che -2 va inserito nel sotto albero sinistro della radice, che al momento è vuoto. Dunque si ha:



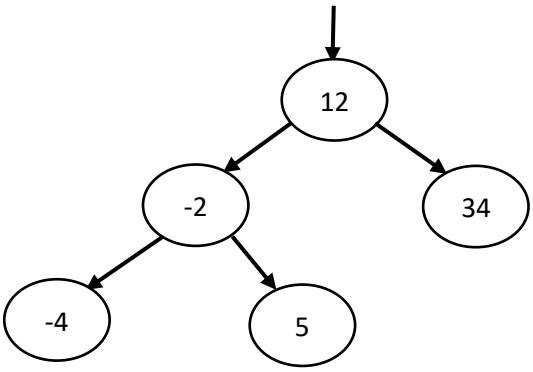
Inserimento di -4.

Confrontandosi con la radice (12) si capisce che occorre inserire a sinistra. Confrontandosi con la radice del sotto albero sinistro di 12, cioè con -2, siccome -4 è più piccolo di -2, si conclude che esso va posto alla sinistra di -2:



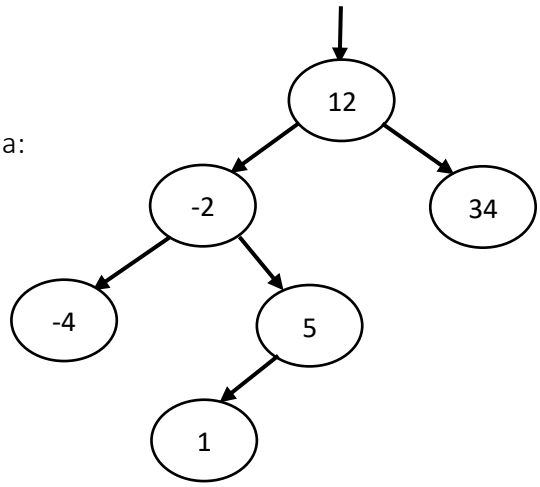
Inserimento di 5.

5 va a sinistra di 12 e alla destra di -2, in quanto più grande. Si ha:



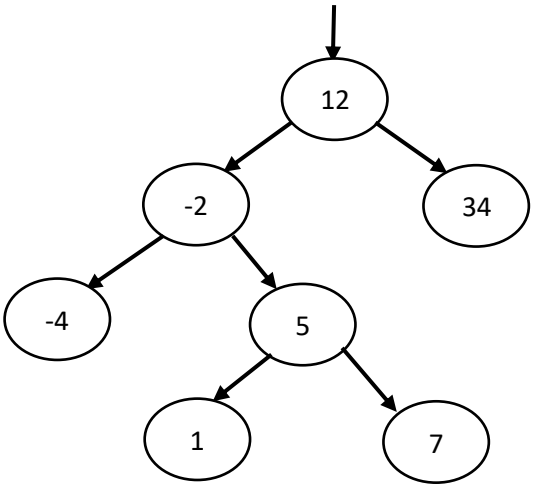
Inserimento di 1.

A sinistra di 12, a destra di -2, a sinistra di 5. Si ha:



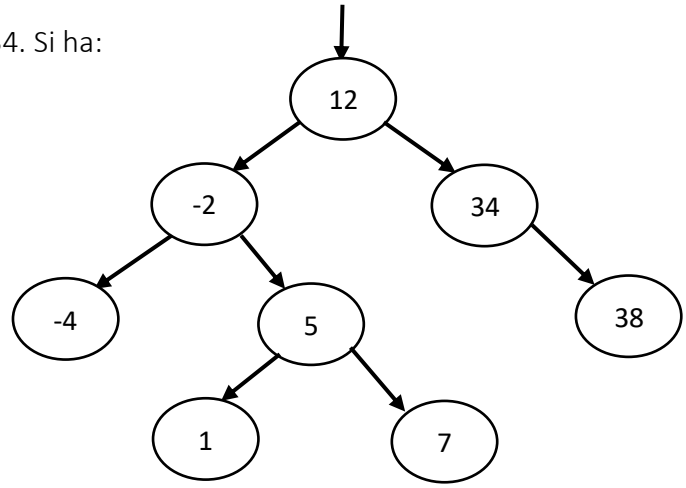
Inserimento di 7.

A sinistra di 12, a destra di -2, a destra di 5. Si ha:

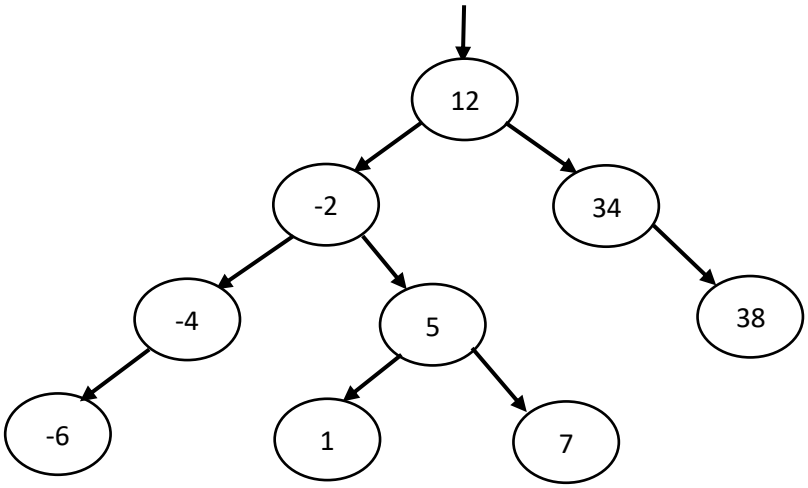


Inserimento di 38.

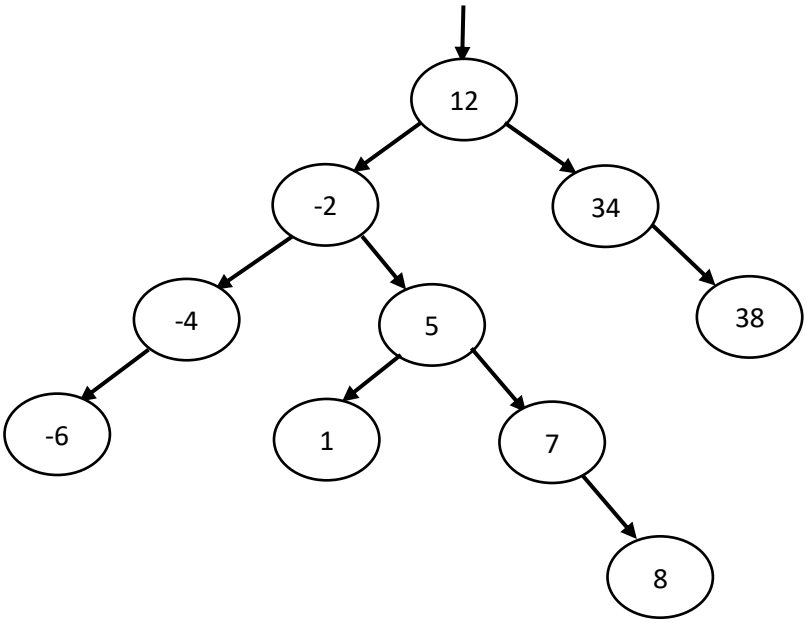
Alla destra di 12, alla destra di 34. Si ha:



Inserimento di -6.
A sinistra di 12, a sinistra di -2, a sinistra di -4, dove si trova un sotto albero vuoto. Si ha:



Inserimento di 8.
Va alla destra di 7. Si ha, infine, l'ABR:



L'esperienza di costruzione dell'ABR ha mostrato chiaramente che:

1. il primo elemento della sequenza va a costituire definitivamente la radice dell'ABR.
2. quando arrivano i successivi elementi, secondo il confronto per l'ordinamento, si vanno a collocare a sx o a dx della radice e cosi via ricorsivamente.
3. ogni nuovo elemento va ad occupare/costituire la radice di un sotto albero vuoto. In altre parole, la posizione di inserimento di un elemento x, coincide sempre con un sotto albero vuoto di un nodo dell'albero rispetto al quale x è maggiore o minore. Il caso di uguaglianza potrebbe essere risolto ponendo x comunque nel sotto albero sinistro.

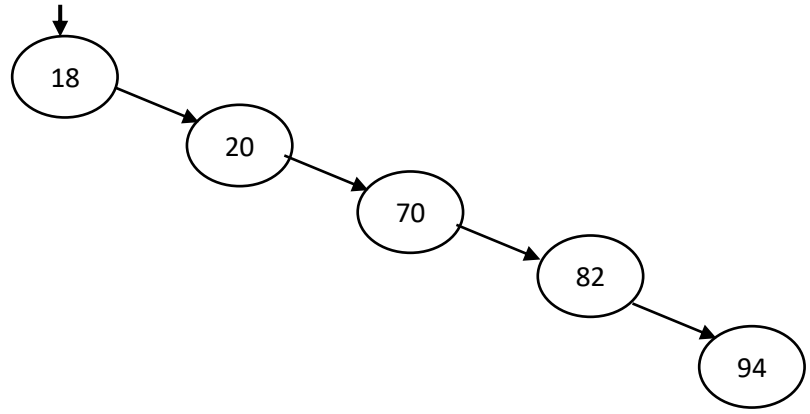
Il risultato, ovviamente, non è in generale un ABR bilanciato.

Si è visto praticamente che costruendo, ad es., un ABR con le parole di un testo/documento di dimensione non bananle, statisticamente può succedere che la distribuzione dei nodi/parole possa seguire naturalmente il bilanciamento.

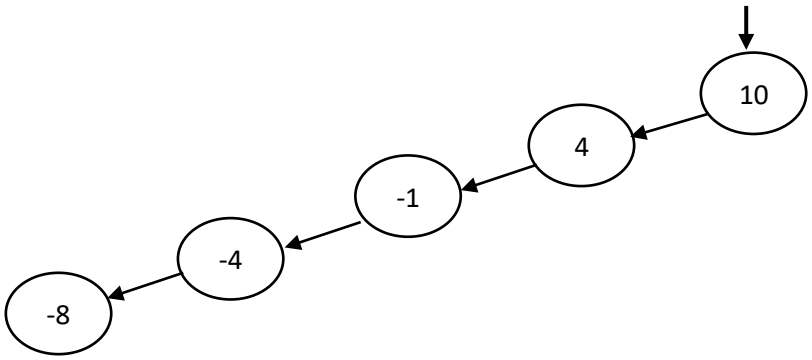
Il bilanciamento può essere assicurato mediante l'adozione di un qualche algoritmo durante l'aggiunta (add) o la rimozione (remove) di elementi. Gli alberi AVL, oggetto di studio nei corsi successivi, mirano sistematicamente a costruire ABR bilanciati. Altre info verranno fornite, cmq, in una prossima lezione del corso di POO.

E' evidente che un ABR può degenerare in una lista se la successione degli elementi è crescente o decrescente. Nel primo caso la lista è stabilita dai puntatori fD. Nel secondo dai puntatori fS. Si vedano gli esempi che seguono:

18 20 70 82 94



10 4 -1 -4 -8



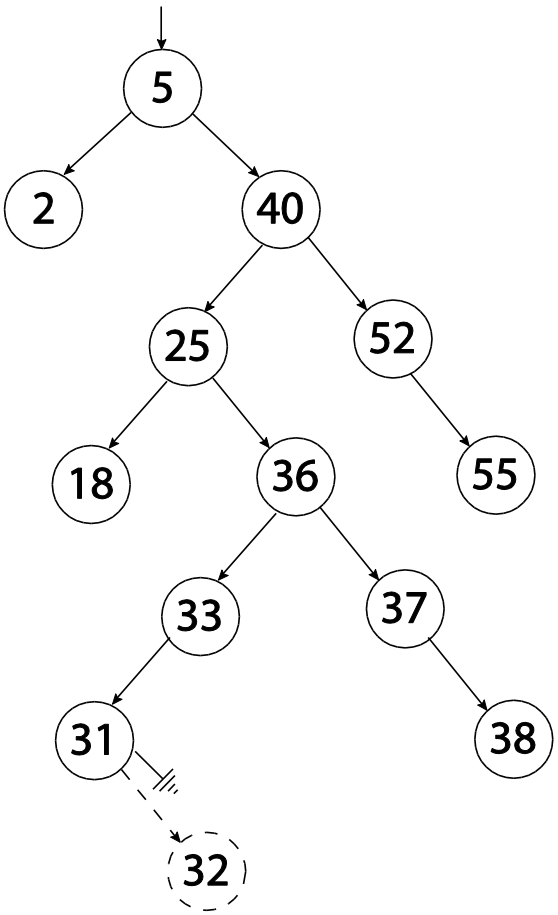
Eliminazione di un elemento dall'albero binario

Mentre l'aggiunta di un elemento avviene sempre in un (sotto) albero vuoto, dunque genera sempre una nuova (temporanea) foglia, la rimozione di un elemento esistente è complicata dal fatto che l'elemento potrebbe appartenere ad un nodo foglia, ad un nodo con un solo figlio, ad un nodo che ha entrambi i figli (caso più complicato).

Esaminiamo graficamente il da farsi nei tre casi suddetti.

Eliminazione di un nodo foglia, es. quello con valore 32

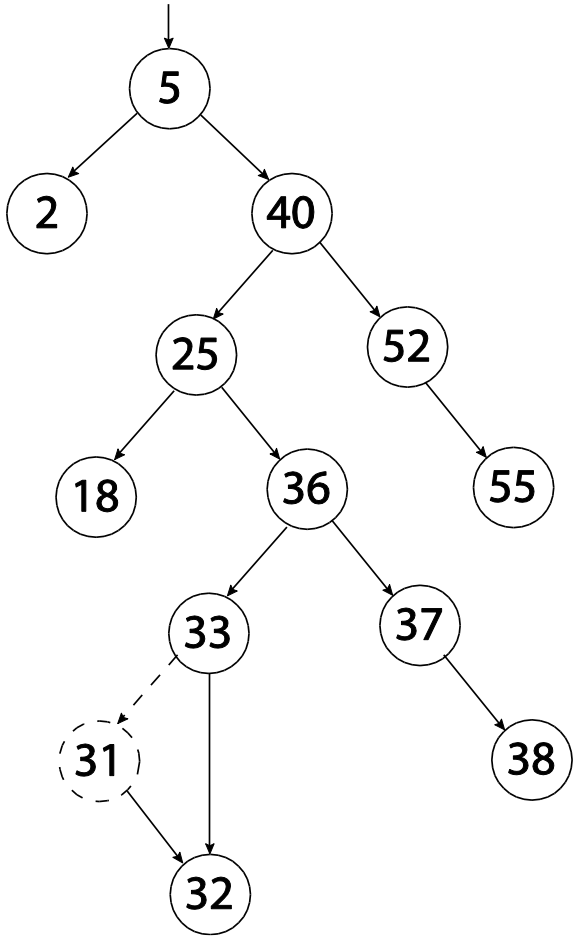
Si porta a termine annullando il riferimento al nodo da eliminare, che può essere il campo figlio sinistro o figlio destro del nodo padre del nodo foglia da rimuovere.



Eliminazione del nodo con 32 (nodo foglia).

Siccome 32 è alla destra di 31, per portare a termine l'eliminazione del nodo 32 occorre assegnare null al campo fD di 31.

Eliminazione dell'elemento 31 (nodo con un solo figlio).



Eliminazione del nodo con 31 (nodo avente un solo figlio).

In questo caso, per portare a compimento la rimozione di 31, è sufficiente modificare il puntatore, nel nodo padre (33), al nodo 31 in modo che adesso punti all'unico figlio di 31, ossia al nodo foglia 32. In particolare, siccome 31 è puntato dal campo fS di 33, è a questo campo fS che occorre assegnare il riferimento all'unico figlio di 31, ossia 32. Dopo questo, 31 diventa irraggiungibile e dunque garbage.

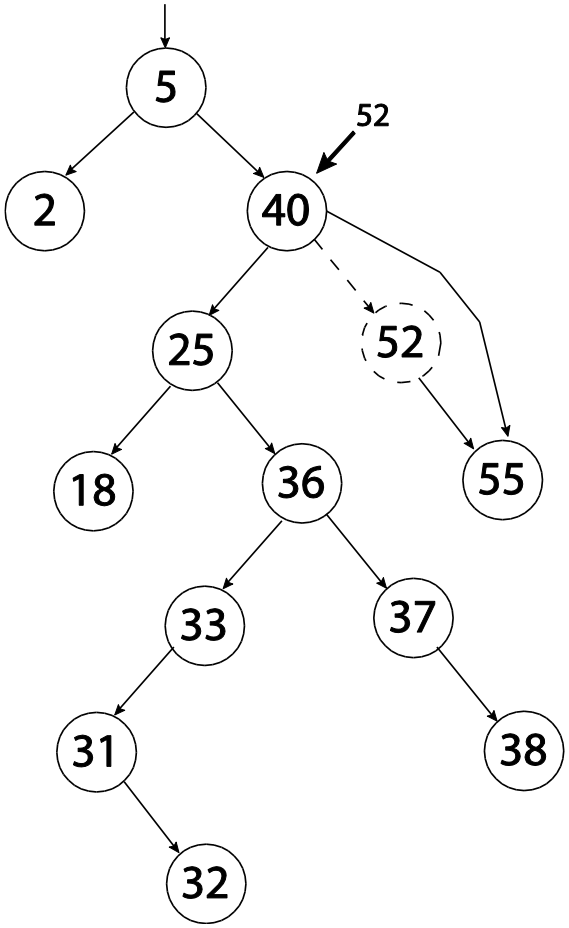
Eliminazione del nodo 40 che ha entrami i figli.

Quando il nodo da rimuovere ammette entrambi i figli, la situazione si complica, dal momento che, naturalmente, non è possibile sostituire al puntatore al nodo, i due puntatori ai figli del nodo!

Di conseguenza, occorre seguire un'altra strada. Si rinuncia ad eliminare fisicamente il nodo e si cerca una "vittima" sacrificale. In questo corso si decide di andare nel sotto albero destro del nodo 40, e si cerca l'informazione più piccola (minimo del sotto albero destro del nodo da rimuovere). Questa informazione appartiene necessariamente ad un nodo che è o una foglia o un nodo con un solo figlio. Dunque un caso tra i primi due già considerati. L'informazione del

nodo vittima (ripetiamo: il minimo nel sotto albero destro del nodo da rimuovere) la si promuove al posto del nodo target della rimozione, quindi si elimina il nodo vittima, secondo le modalità già spiegate nei primi due casi.

Tuttavia esistono due sotto casi allorquando il nodo da rimuovere dispone di entrambi i figli. Il primo caso (più semplice da trattare) si verifica quando la radice del sotto albero destro del nodo da rimuovere, è esso stesso il nodo vittima. Ciò accade quando la radice del sotto albero destro, non ha figlio sinistro. Le operazioni da compiere sono riassunte nella seguente figura.



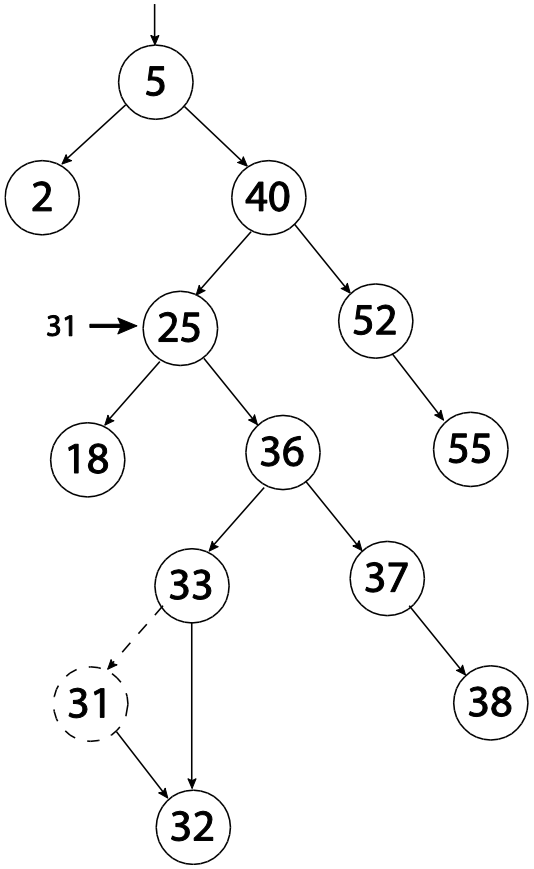
Nodo da rimuovere 40. Ha entrambi i figli e 52, la radice del sotto albero destro di 40, è il minimo del sotto albero destro.

Si promuove la vittima (52) al posto del nodo da rimuovere (40). Quindi si elimina *fisicamente* il nodo 52 con un semplice by-pass.

Come si vede, è il nodo vittima che viene eliminato. Il nodo con 40, è rimosso *logicamente*, nel senso che non lo si trova più nell'albero.

Il secondo sotto caso (caso più generale) si verifica quando il minimo nel sotto albero destro, non coincide con la radice del sotto albero destro, ma è il nodo più a sinistra nel sotto albero destro, che si individua muovendosi il più possibile a sinistra della radice del sotto albero destro. Vediamo un caso concreto.

Nodo da eliminare 25 (ammette entrambi i figli, ma si verifica il caso più generale)



Nodo da eliminare 25. Il nodo vittima è quello più a sinistra nel sotto albero destro di 25.

In questo caso, si parte dalla radice del sotto albero destro di 25, ossia il nodo 36, e si continua a seguire i campi fS. Il nodo vittima è quello più a sinistra, ossia 31, che è un nodo con un solo figlio. Si promuove il 31 al posto del 25, e si elimina la vittima, assegnando al figlio sinistro di 33, il puntatore al figlio di 31, che è la foglia 32.

Ovviamente può anche succedere che il nodo vittima (minimo del sotto-albero destro del nodo da rimuovere) sia una foglia, nel qual caso è sufficiente porre a null il puntatore al nodo vittima. Un esempio, nell'albero di figura, ricorre quando si chiede di rimuovere il 5, ossia l'elemento nel nodo radice. In questo caso, il nodo più a sinistra nel sotto-albero destro è il 18, che è un nodo foglia. Si promuove 18 al posto del 5, e si elimina fisicamente il 18 ponendo a null il campo figlio sinistro (fS) del nodo padre 25.

Visita di un albero binario

Esistono tre metodi di visita, intrinsecamente ricorsivi: inOrder, preOrder, postOrder. La visita in ordine (inOrder), detta anche visita simmetrica, visita gli elementi dell'albero dal più piccolo al più grande. Tenendo conto che un albero vuoto non richiede alcuna visita o è già visitato, si può dire che la visita inOrder procede come segue:

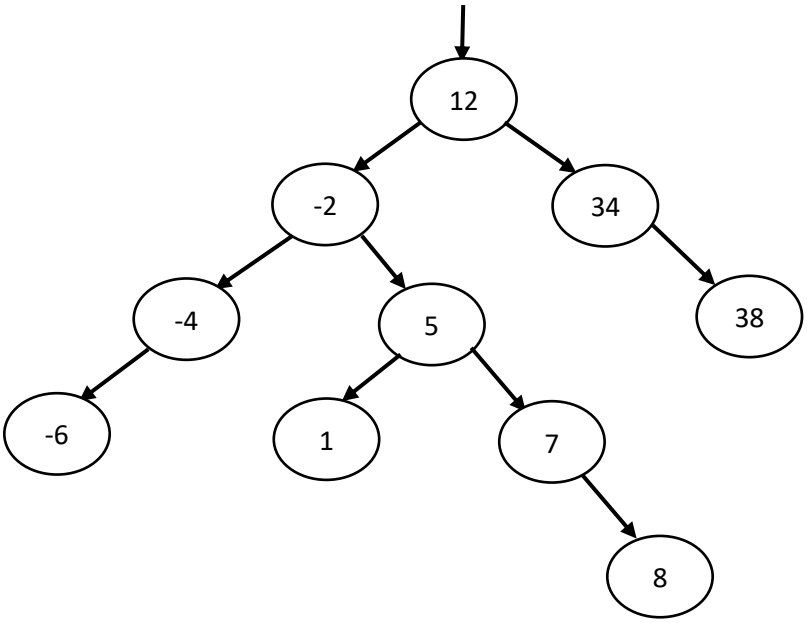
se l'albero non vuoto:
    visita il sotto albero sinistro della radice
    visita il nodo radice
    visita il sotto albero destro della radice

Ma come si visita il sotto albero sinistro o destro? Semplice: con la stessa tecnica! Per ragioni di generalità, è opportuno conservare gli elementi visitati su una lista che poi può essere stampata etc. SI ha dunque il seguente metodo di visita inOrder:

```
public void inOrder( List<T> lista ){
    inOrder( radice, lista );
}

private void inOrder( Nodo<T> radice, List<T> lista ){
    if( radice!=null ){
        inOrder( radice.fS, lista );
        lista.add( radice.info );
        inOrder( radice.fD, lista );
    }
}
```

Per l'albero ABR:



la visita in ordine genera la successione: -6 -4 -2 1 5 7 8 12 34 38.

La visita preOrder (ordine anticipato), visita prima la radice, poi il sotto albero sinistro, quindi quello destro:

```
public void preOrder( List<T> lista ){
        preOrder( radice, lista );
}
Private void preOrder( Nodo<T> radice, List<T> lista ){
        If( radice!=null ){
                lista.add( radice.info );
                preOrder( radice.fS, lista );
                preOrder( radice.fD, lista );
        }
}
```

La vista postOrder (ordine posticipato), infine, visita prima il sotto albero sinistro, poi quello destro, quindi la radice:

```
public void postOrder( List<T> lista ){
        postOrder( radice, lista );
}
Private void postOrder( Nodo<T> radice, List<T> lista ){
        If( radice!=null ){
                postOrder( radice.fS, lista );
                postOrder( radice.fD, lista );
                lista.add( radice.info );
        }
}
```

Con riferimento all'ABR precedente si ha:

preOrder: 12 -2 -4 -6 5 1 7 8 34 38
postOrder: -6 -4 1 8 7 5 -2 38 34 12

Su un ABR ha senso solo inOrder. La successione preOrder/postOrder, infatti, non rappresenta una sequenza significativa degli elementi. In altri alberi binari (es. gli alberi di espressione) hanno invece senso, come vedremo, tutte e tre i metodi di visita.

Visita per livelli

12 -2 34 -4 5 38 -6 1 7 8

Struttura di iterazione di un ABR

Come ci si aspetta, l'iteratore su un ABR deve ritornare in sequenza tutti gli elementi dell'ABR dal minimo al massimo.

Una prima soluzione consiste nel costruirsi una lista con la visita in ordine dell'ABR, e di iterare sulla lista, stando attenti che quando si rimuove un elemento dall'iteratore non è sufficiente eliminare l'elemento dalla lista ma occorre rimuovere anche dall'albero. Vediamo come (per semplicità si ignora l'eccezione ConcurrentModificationException):

```
public Iterator<T> iterator(){ return new ABRIterator(); }

private class ABRIterator implements Iterator<T>{
	private LinkedList<T> lista=new LinkedList<>();
	private Iterator<T> it;
	private T cor;
	public ABRIterator(){
		inOrder( lista );
		it=lista.iterator();
	}
	public boolean hasNext(){ return it.hasNext(); }//hasNext
	public T next(){
		cor=it.next();
		return cor;
	}//next
	public void remove(){
		it.remove();
		ABR.this.remove(cor);
		cor=null; //inutile
	}//remove
}//ABRIterator
```

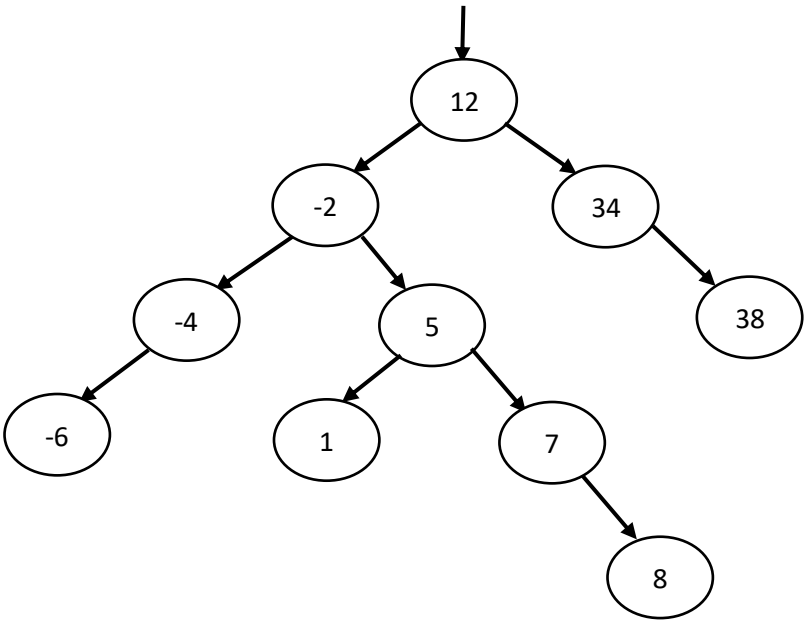
Una differente soluzione

Si fa uso di uno stack di nodi (alberi), con la proprietà che in ogni momento l'elemento affiorante dallo stack sia il nodo con il valore più piccolo non ancora consumato, candidato a diventare nodo corrente a seguito di una next().

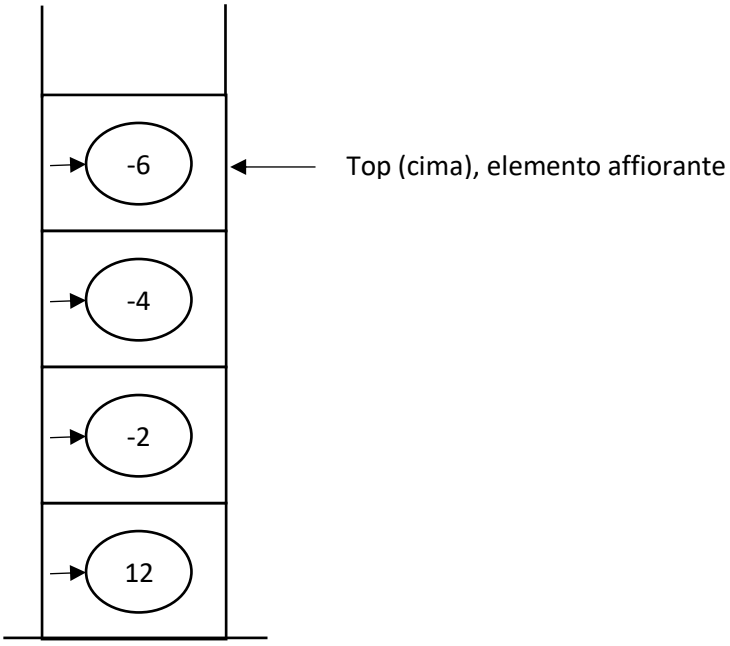
Inizialmente si carica sullo stack la successione di nodi, a partire dalla radice dell'ABR, che arriva al nodo più a sinistra nel sotto albero sinistro della radice. L'elemento affiorante in questo momento è il minimo dell'ABR.

- hasNext() ritorna true se lo stack non è vuoto.
- next() fa una pop dallo stack. Il nodo prelevato diventa il nodo con l'elemento corrente. A questo punto, se il nodo corrente ammette un sotto albero destro (essendo il corrente il minimo, certamente esso non ha sotto albero sinistro!), si carica sullo stack la successione di nodi del sotto albero destro, partendo dalla radice del sotto albero destro, che arriva sull'elemento più a sinistra. Se non esiste sotto albero destro del nodo corrente, la prossima next() attinge semplicemente al prossimo nodo sullo stack.
- remove(), se effettuabile, toglie l'elemento (info) del nodo corrente dall'albero, opportunamente delegando i dettagli alla remove() dell'ABR. A questo punto dovrebbe essere facile verificare, per il modus operandi della remove() dell'ABR, che il nodo affiorante dallo stack, cosi come realizzato dalla next() che precede questa remove() dell'iteratore, nei casi più generali di un nodo che ammetta entrambi i figli, già contiene il nuovo minimo a seguito della promozione. E' per questa ragione che si è implementata la remove() dell'ABR andando a cercare, come vittima, l'elemento più piccolo nel sotto albero destro!

Con riferimento all'ABR:



al tempo del costruttore dell'iteratore, lo stack è il seguente:



Contenuto iniziale dello stack di nodi (ogni nodo è un albero, la cui radice è il nodo stesso).

Applicazione fpfpq

Mappa fp

casa	3

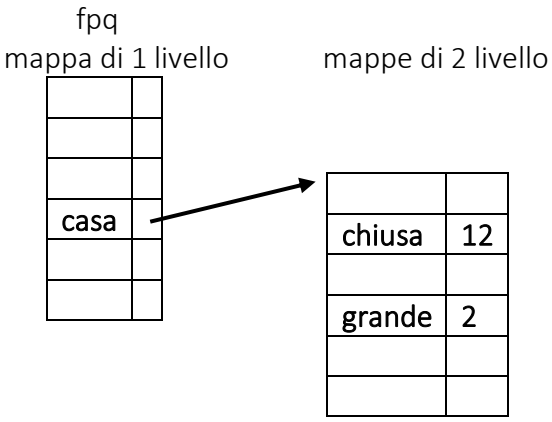
Chiave: String (parola)
Valore: Integer (frequenza di uso della parola)

Detta f la frequenza (assoluta) di una parola p, dividendo f per il numero totale di parole, si ottiene la frequenza relativa di p, un numero tra 0 ed 1 (tra 0% e 100%).

Mappa fpq

Chiave: una parola (String)
Valore: una sotto mappa <String,Integer>

Data una parola p, la mappa associata a p contiene le parole q usate subito dopo p, e la frequenza con cui ogni possibile coppia <p,q> si ripete.



Si può parlare di mappa di primo livello, che è la mappa fpq. Per ogni parola p, nella mappa di primo livello c'è il riferimento ad una mappa di secondo livello che elenca tutte le parole q che nel testo sono usate subito dopo p. Per ogni coppia <p,q>, la mappa di secondo livello fornisce la frequenza con cui si ripete la coppia <p,q> nel testo. Dividendo la frequenza di <p,q>, per il numero di volte in cui p è usata nel testo, si ha una frequenza relativa.
Si dice pure che la mappa di secondo livello associata ad una parola p, fornisce tutte le parole adiacenti a p, ossia che sono usate subito dopo p, nel testo.