7 Strutture dati fondamentali

Sommario: *In questo capitolo si definisce cosa sia una struttura dati e si studiano le strutture dati principali. In particolare, oltre a riprendere le strutture dati già incontrate fin ora, si introdurranno esempi di strutture dati dinamiche (liste concatenate) e non lineari (alberi).*

L'algoritmo Heapsort che abbiamo illustrato nel capitolo precedente è un ottimo esempio di soluzione (al problema dell'ordinamento, nel caso specifico) nella quale un ruolo essenziale è rivestito dalla ***struttura dati*** su cui l'algoritmo lavora. Nel caso dello Heapsort la struttura dati garantisce una proprietà, quella di avere il massimo degli elementi sempre in prima posizione, senza la quale l'algoritmo non funzionerebbe correttamente e così efficientemente.
Questo ci fa capire come la scelta della struttura dati più adatta sia un aspetto fondamentale nel progetto di algoritmi efficienti. Naturalmente, per poter utilizzare di volta in volta le strutture dati più adatte alla risoluzione dei problemi bisogna conoscere le loro caratteristiche e le loro proprietà.

Una *struttura dati* è un particolare tipo di dato, caratterizzato dall'organizzazione dei dati, più che dal tipo di dato stesso. È quindi caratterizzata da un modo sistematico di organizzare i dati e mette a disposizione un insieme di operatori che permettono di manipolare la struttura. Come vedremo più approfonditamente in seguito, una struttura dati può essere *lineare* o *non lineare* (a seconda se esista

una sequenzializzazione, e se - informalmente - possa essere pensata con un inizio ed una fine), *statica* o *dinamica* (a seconda che possa variare la sua dimensione nel tempo), omogenea o disomogenea (rispetto ai contenuti).

Una struttura dati serve a memorizzare e manipolare ***insiemi dinamici***, cioè insiemi i cui elementi possono variare nel tempo in funzione delle operazioni compiute dall'algoritmo che li utilizza.

Gli elementi dell'insieme dinamico (spesso chiamati anche ***nodi, record***, o ***oggetti***) possono a loro volta essere piuttosto complessi e contenere ciascuno più di un dato "elementare". In tal caso è abbastanza comune che essi contengano:

- una ***chiave***, utilizzata per distinguere un elemento da un altro nell'ambito delle operazioni di manipolazione dell'insieme dinamico; normalmente i valori delle chiavi fanno parte di un insieme totalmente ordinato (ad. es., sono numeri interi);
- ulteriori dati, detti ***dati satellite***, che sono relativi all'elemento stesso ma non sono direttamente utilizzati nelle operazioni di manipolazione dell'insieme dinamico.

In altri casi, ogni elemento contiene solo la chiave e quindi coincide con essa. Nel seguito ci riferiremo sempre a tale situazione, a meno che non venga esplicitamente specificato il contrario. I termini chiave e valore saranno quindi interscambiabili.

Prima di procedere, si osservi che abbiamo già incontrato un caso in cui la presenza di dati satellite richiede la modifica di un algoritmo: il counting sort. In quel caso, lo pseudocodice si presentava più

semplice nel caso in cui non erano presenti dati satellite, anche se il costo computazionale rimaneva lo stesso del caso più generale.
Le tipiche operazioni che si compiono su un insieme dinamico *S* (e quindi sulla struttura dati che ne permette la gestione), che si suppone totalmente ordinabile, si dividono in due categorie: ***operazioni di interrogazione*** e ***operazioni di modifica***.
Tipici esempi di operazioni di interrogazione, che non modificano la consistenza dell'insieme dinamico, sono:

- Search(S, k): recuperare l'elemento con chiave di valore *k*, se è presente in *S*, restituire un valore speciale nullo altrimenti;
- Min(S): recuperare l'elemento di minimo valore presente in *S*;
- Max(S): recuperare l'elemento di massimo valore presente in *S*;
- Pred(S, k): recuperare l'elemento presente in *S* che precederebbe quello di valore *k* (di cui supponiamo di conoscere la locazione) se *S* fosse ordinato;
- Succ(S, k): recuperare l'elemento presente in *S* che seguirebbe quello di valore *k* (di cui supponiamo di conoscere la locazione) se *S* fosse ordinato.

Tipici esempi di operazioni di manipolazione, che invece modificano la consistenza dell'insieme dinamico, sono:

- Insert(S, k): inserire un elemento di valore *k* in *S*;
- Delete(S, k): eliminare da *S* l'elemento di valore *k* (di cui supponiamo di conoscere la locazione).

Le differenti strutture dati che vedremo, se da un lato hanno in comune la capacità di memorizzare insiemi dinamici, dall'altro differiscono anche profondamente fra loro per le ***proprietà*** che le caratterizzano. Sono proprio le proprietà della struttura dati ad essere l'elemento determinante nella scelta da effettuare quando si deve progettare un algoritmo per risolvere un problema.

Un'ultima considerazione a proposito delle strutture dati: poiché esse sono destinate alla gestione di insiemi dinamici, è piuttosto comune che esse vengano implementate mediante l'allocazione dinamica della memoria per contenerne gli elementi e l'uso di puntatori per accedere agli elementi stessi. Alternativamente, possono essere implementate per mezzo di array purché il numero massimo dei loro elementi possa essere stimato in anticipo. Si noti comunque che le proprietà di una struttura dati sono "intrinseche", e devono essere garantite qualunque sia l'implementazione.

Prima di studiare alcune strutture dati, da quelle più semplici (***lista, coda e pila***) a quelle un pò più complesse (***albero e grafo***), studiamo le strutture dati che già abbiamo incontrato, l'array (che chiameremo *semplice*) e l'array ordinato.

Innanzi tutto, osserviamo che un array (sia esso ordinato o no) ha le seguenti caratteristiche:

- Memorizza elementi omogenei, cioè dello stesso tipo;
- È statico, cioè ha una lunghezza che deve essere definita *a priori*;
- È una struttura ad accesso casuale, cioè possiamo accedere ad

ogni suo elemento in un tempo costante che è indipendente dalla sua posizione.

Abbiamo detto in precedenza che possiamo assimilare il tipo *list* del Python ad un array, anche se è, in verità, più potente. In effetti, una struttura di tipo *list* non ha lunghezza definita a priori e non deve necessariamente memorizzare elementi omogenei. In seguito vedremo da cosa deriva questa maggior potenza.

Cerchiamo ora di capire quale sia il costo computazionale di alcune delle operazioni sopra citate quando vengano implementate su vettori semplici o ordinati.

Search(S, k): se S è un array semplice di dimensione n, l'unico modo di procedere è quello di scorrere l'array, quindi nel caso peggiore, in cui l'elemento k non è presente, il costo è $\Theta(n)$. Se S è invece un array ordinato, si può usare l'algoritmo di ricerca binaria e pertanto il costo scende a $O(\log n)$.

Min(S): la ricerca del minimo in un array semplice si esegue di nuovo scorrendo l'intera struttura dati, con un costo di $\Theta(n)$, mentre nell'array ordinato (ad esempio in senso non decrescente) l'elemento di chave minima sarà quello memorizzato nella prima posizione, ed è quindi recuperabile in tempo $\Theta(1)$.

Analoghe considerazioni si possono fare per l'operazione Max(S).

Pred(S, k): anche per questa operazione l'ordinamento dell'array fa molta differenza, infatti bisogna scorrere per intero l'array semplice per trovare il predecessore di k, mentre è sufficiente conside-

rare l'elemento nella posizione che precede quella di *k* se l'array è ordinato in senso non decrescente. I costi computazionali sono rispettivamente $\Theta(n)$ e $\Theta(1)$.

Analoghe considerazioni valgono per l'operazione Succ(S,k).

Insert(S, k): le operazioni che modificano la struttura dati devono mantenerne le proprietà; se l'array è semplice non ci sono particolari requisiti per la posizione che deve assumere *k*, quindi esso potrà essere posizionato nella prima posizione libera dell'array (il cui indice dovrà essere mantenuto aggiornato), con costo $\Theta(1)$; viceversa, nel caso dell'array ordinato, l'inserimento di *k* dovrà avvenire mantenendo l'ordinamento pertanto, supponendo che la posizione di *k* sia già stata individuata tramite Search(S, k), bisognerà "fare posto" all'elemento spostando tutti gli elementi che seguono di una posizione verso destra; il costo di questa operazione è quindi $O(n)$.

Delete(S, k): anche in questo caso abbiamo un indice che dà indicazioni sulla posizione dell'elemento *k* da cancellare, quindi non è necessario anteporre un algoritmo di ricerca. Nel caso dell'array ordinato è necessario spostare verso sinistra tutti gli elementi memorizzati dopo l'elemento *k*, in modo da mantenere la struttura dati come un array pieno; l'operazione di cancellazione costa così $O(n)$. Viceversa, se l'array non è ordinato basta scambiare l'elemento da cancellare con l'ultimo elemento presente nell'array e cancellare quest'ultimo, riducendo poi la dimensione dell'array di uno, con costo $\Theta(1)$.

Struttura dati	**Search(S,k)**	**Min(S)** **Max(S)**	**Pred(S,k)** **Succ(S,k)**	**Insert(S,k)**	**Delete(S,k)**
Array qualunque	$O(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$O(1)$
Array ordinato	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$

Già da questo semplice confronto delle due strutture dati più semplici in assoluto, si può dedurre come non abbia senso dire che una struttura è migliore di un'altra: le strutture dati possono essere più o meno adatte ad un certo algoritmo e sta al buon progettista disegnare un algoritmo efficiente usando la struttura dati più adatta.

7.1 Lista concatenata semplice

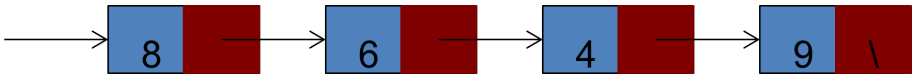
Un ***puntatore*** è una variabile di tipo particolare che contiene l'indirizzo in memoria di un'altra variabile. I puntatori sono necessari per costruire strutture dati dinamiche e per passare i parametri per riferimento. In alcuni linguaggi di programmazione, tra cui il Python, essi sono "nascosti" per rendere più semplice l'implementazione.
La ***lista concatenata semplice*** (o **lista semplice**, da non confondersi con il tipo *list* del Python) è una struttura dati nella quale gli elementi sono organizzati in successione. Essa fornisce un meccanismo semplice e flessibile per gestire insiemi dinamici e supporta tutte le

operazioni di interrogazione e manipolazione elencate in precedenza (non tutte in maniera efficiente, però).

Le proprietà specifiche delle liste sono le seguenti:

- l'accesso avviene sempre ad una estremità della lista, per mezzo di un'apposita informazione di accesso (puntatore alla testa della lista);
- è permesso solo un accesso sequenziale agli elementi: non è possibile l'accesso diretto (ossia l'accesso in tempo costante a uno qualunque degli elementi).

A differenza dell'array, nel quale il numero di elementi è prefissato e la successione degli elementi è realizzata mediante gli indici degli elementi, la lista può crescere di dimensioni e la successione degli elementi è implementata mediante un collegamento esplicito da un elemento ad un altro (ad es., tramite un puntatore). Il valore speciale *NONE* rappresenta "nessun puntatore" (nelle figure viene usato il carattere \).



Per inserire un elemento, che diviene il primo, nella lista concatenata lo pseudocodice è il seguente. In esso si presuppone che p sia il puntatore alla testa della lista e k il puntatore al record da inserire, già creato ma non ancora inserito; tale record contiene il valore della chiave nel campo *key* e il valore *NONE* nel campo *next*.

```
def Insert(p,k):
  if k!=None:
```

```
    k.next=p
  return k
```

Per avere una struttura dati su cui poter provare le funzioni che seguono in Python, è possibile usare il codice seguente:

```
class Nodo:
    def __init__(self, key=None, next=None):
        self.key=key
        self.next=next

def crea(A):
    'crea una lista concatenata con le chiavi contenute in A
     e ne restituisce il puntatore alla testa'
    p=None
    for x in lista:
        q=Nodo(x)
        q.next=p
        p=q
    return p

def stampa(p):
    'stampa la lista puntata da p'
    while p:
        print(p.key)
        p=p.next

>>> A=[1,2,3,4,5,6]
>>> p=crea(A)
>>> stampa(p)
```

Il costo computazionale dell'inserimento è $\Theta(1)$.

Si osservi che l'elemento è stato inserito come primo elemento della lista o, come si dice, ***in testa*** (alla lista). L'inserimento in testa è il modo più semplice di inserire un nuovo dato in una lista semplice, e può essere utilizzato tutte le volte che gli elementi della lista non devono avere un ordinamento particolare. Ma possiamo inserire l'elemento puntato da k anche in una posizione qualsiasi, ad esempio dopo l'elemento puntato da un terzo punatore q:

```
def Inserisci_dopo_q(p,k,q):
  k.next=q.next
  q.next=d
```

Proprio perché gli elementi della lista non hanno un ordinamento particolare, per cercare un elemento specifico può essere necessario scandire l'intera lista.

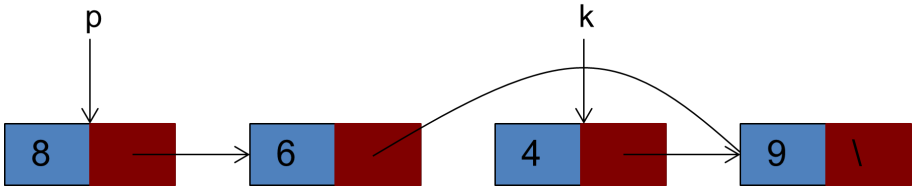
Lo pseudocodice è il seguente. Di nuovo p è il puntatore alla testa della lista, e viene restituito il puntatore all'elemento cercato se esso è presente, altimenti il valore None.

```
def Search(p,k):
  p_corr = p
  while ((q!=None) and (p_corr->key!=k)):
    p_corr = p_corr.next
  return p_corr
```

È facile convincersi che, per cercare un elemento in una lista concatenata di n elementi, il costo nel caso peggiore è $\Theta(n)$.

Con le liste semplici alcune delle altre operazioni non sono così naturali da realizzare. Ad esempio, per cancellare un elemento, pur

avendo un puntatore al record che lo contiene, è necessario scandire la lista dall'inizio fino al record che precede nella lista quello da cancellare, in modo da poter "cortocircuitare" l'elemento da cancellare, collegando il record che lo precede direttamente a quello che lo segue.



Nel seguente pseudocodice, per semplicità assumiamo che la lista da cui si vuole cancellare l'elemento puntato da k sia non vuota e puntata da p; non è difficile estendere lo pseudocodice affinché funzioni correttamente anche nel caso in cui la testa della lista punti a None:

```
def Delete(p, k):
 if k == p:                # cancel. 1° elem
    p=p.next
    return p
 p_corr = p
 while p_corr.next!=k:
    p_corr=p_corr.next  # qui p_corr punta all'elem. che precede k
 p_corr.next = k.next
 return p
```

Notiamo che dobbiamo scorrere l'intera lista, il costo è dunque $\Theta(n)$.

Concludiamo con delle osservazioni sull'oggetto Python **list**. Esso ha somiglianze e differenze tanto con gli array che con le liste semplici. Infatti, come si fa negli array:

- è possibile accedere ai suoi elementi tramite la loro posizione;
- la sua lunghezza è sempre nota tarmite il comando *len*.

Similmente a quanto accade nelle liste semplici:

- è possibile memorizzare dati non omogenei;
- la lunghezza non è necessariamente fissa.

Questo perché l'implementazione di questo oggetto è fatta collegando tramite puntatori elemento per elemento una lista semplice ad un array. Ciò consente di ereditare da entrambe le strutture alcune proprietà fondamentali, ma è necessario tenerne conto quando si calcola il costo delle operazioni semplici che compiamo su di essa:

- *append* (inserimento in ultima posizione) si esegue in $\Theta(1)$;
- *insert* (inserimento in i-esima posizione) si esegue in $O(n)$ poiché bisogna far scorrere in avanti tutti gli elementi in posizione successiva alla i-esima;
- *pop* oppure *del* (elimina e restituisce oppure elimina l'elemento in i-esima posizione) si esegue in $O(n)$ poiché bisogna far scorrere indietro gli elementi in posizione successiva alla i-esima;
- *concatenate* (unifica due oggetti di tipo *list* in uno solo) costa $\Theta(k)$ dove k è la lunghezza del secondo oggetto.

7.2 Lista doppia

Alcuni problemi riscontrati nella lista semplice (ad esempio costo computazionale lineare nella cancellazione) possono essere risolti organizzando la struttura dati in modo che da ogni suo elemento si possa accedere sia all'elemento che lo segue che a quello che lo precede nella lista, quando essi esistono. Tale struttura dati si chiama ***lista doppia*** (o ***lista doppiamente concatenata***):

Alle proprietà della lista semplice si aggiunge la seguente:

- la lista doppia può essere scandita in entrambe le direzioni.

Le operazioni di inserimento e ricerca sono analoghe alle precedenti (con le ovvie modifiche necessarie alla corretta gestione del campo *prev*). L'operazione di eliminazione di un elemento è la seguente, dove al solito p è il puntatore alla testa della lista mentre k è il puntatore all'elemento da eliminare:

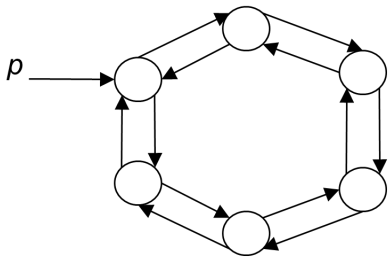
```
def Delete_Doppia(p,k)
  if k.prev!=None:
    k.prev.next=k.next
  else:
    p=k.next
  if k.next!=None:
    k.next.prev=k.prev
  return p
```

Il costo computazionale è $\Theta(1)$.

Riassumiamo la situazione nella seguente tabella:

Struttura dati	**Search(S,k)**	**Min(S)** **Max(S)**	**Pred(S,k)** **Succ(S,k)**	**Insert(S,k)**	**Delete(S,k)**
Lista semplice	$O(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$O(n)$
Lista doppia	$O(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

Laddove ci sia bisogno di non distinguere tra un elemento speciale, il primo, puntato dal puntatore alla testa della lista, si può operare un'ulteriore evoluzione della lista doppia: la ***lista circolare***, nella quale il primo e l'ultimo elemento sono collegati fra loro:



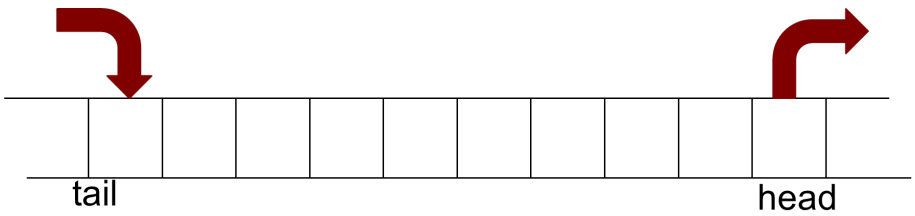
Essa può essere utile nelle situazioni in cui si voglia, ad esempio, effettuare una computazione organizzata per fasi, nella quale ciascuna fase richiede la scansione dell'intera struttura dati. Inoltre, semplifica l'implementazione delle operazioni standard poiché in esse non è necessario tenere conto dei casi speciali relativi al primo e all'ultimo elemento della lista.

7.3 Coda

La ***coda*** è una struttura dati che esibisce un comportamento ***FIFO*** (***First In First Out***). In altre parole, la coda ha la seguente proprietà:

> gli elementi vengono prelevati dalla coda esattamente nello stesso ordine col quale vi sono stati inseriti.

La coda può essere visualizzata come una coda di persone in attesa ad uno sportello ed uno dei suoi più classici utilizzi è la gestione della coda di stampa, in cui documenti mandati in stampa prima vengono stampati prima.
Su una coda sono definite solo due operazioni: l'inserimento (che di norma viene chiamata ***Enqueue***) e l'estrazione (che di norma viene chiamata ***Dequeue***). Viceversa, di solito non si scandiscono gli elementi di una coda né si eliminano elementi con mezzi diversi dalla Dequeue.
La particolarità che garantisce la proprietà FIFO è che l'operazione Enqueue opera su una estremità della coda (***tail***) e la Dequeue opera sull'altra estremità (***head***):



Osserviamo che la struttura dati coda, è una struttura dati astratta che ha una sua identità indipendentemente da come viene implementata.

Nel seguito mostreremo lo pseudocodice delle operazioni di Enqueue e di Dequeue nei due casi in cui la coda venga implementata come una lista e, poi, come un array.
Nel caso in cui l'implementazione avvenga mediante una lista, lo pseudocodice delle due operazioni è riportato nel seguito. L'elemento da inserire si presuppone già creato, con valore None nel suo camp *next* e puntato da un puntatore *e*.

```
def Enqueue_Lista (head, tail, e):
  if (tail == None):        #la coda è vuota
    tail = e
    head = e
  else:
    tail.next = e
    tail = e
  return head, e

def Dequeue_Lista (head, tail):
  if (head == None):
    print('Errore: coda vuota')
    return None, None, None
  e = head
  head = e.next
  if (head == None):
    tail = None
  return head,tail,e
```

Il costo computazionale di entrambe le operazioni *Enqueue* e *Dequeue* è $\Theta(1)$. Si noti che se nella figura precedente orientassimo i puntatori fra gli elementi della coda nell'altro verso l'operazione di *Dequeue* richiederebbe costo $\Theta(n)$.

Anche le code, come le liste, possono essere realizzate mediante arrays. In questa implementazione, però, c'è una complicazione da gestire legata al fatto che a seguito di ripetute operazioni di Enqueue e Dequeue gli elementi della coda si spostano progressivamente verso una delle due estremità dell'array e, quando la raggiungono, non vi è più apparentemente spazio per i successivi inserimenti. La ragione è che, al fine di garantire costo computazionale costante, i dati da estrarre vengono cancellati solo logicamente e non fisicamente: gli elementi estratti tramite Dequeue infatti rimangono nell'array ma sono considerati come eliminati grazie all'opportuna posizione degli indici head e tail. La soluzione a questo problema è gestire l'array in modo circolare, considerando cioè il primo elemento come successore dell'ultimo.

7.4 Coda con priorità

La ***coda con priorità*** è una variante della coda. Con quest'ultima ha in comune il fatto che l'inserimento avviene ad un'estremità e l'estrazione avviene all'estremità opposta.

Però, a differenza della coda, nella coda con priorità la posizione di ciascun elemento non dipende dal momento in cui è stato inserito, ma dal valore di una determinata grandezza, detta ***priorità***, la quale in generale è associata ad uno dei campi presenti nell'elemento

stesso. Di conseguenza, gli elementi di una coda con priorità sono collocati in ordine crescente (o decrescente, a seconda dei casi) rispetto alla grandezza considerata come priorità.
Ad esempio, se la priorità è associata al valore numerico del campo *key*, quando un nuovo elemento avente $key = x$ viene inserito in una coda con priorità (crescente) esso viene collocato come predecessore del primo elemento presente in coda che abbia $key \geq x$. Solo se la coda contiene solo elementi con priorità minore di quella dell'elemento che viene inserito, esso diviene l'elemento di testa della coda, il primo quindi che verrà estratto.
In questo senso, un array ordinato è una coda con priorità, e la priorità coincide con la chiave.
Un altro esempio di coda con priorità è la struttura dati heap, che abbiamo già incontrato studiando gli algoritmi di ordinamento.
Anche una coda può essere intesa come una coda con priorità, in cui il parametro di priorità è il maggior tempo di permanenza nella struttura dati. Viceversa, la pila (che vedremo tra poco) è una coda con priorità dettata dal minor tempo di permanenza nella struttura.
Si noti che la coda con priorità presenta un potenziale pericolo di ***starvation*** (***attesa illimitata***): un elemento potrebbe non venire mai estratto, se viene continuamente scavalcato da altri elementi di priorità maggiore che vengono via via immessi nella struttura dati.

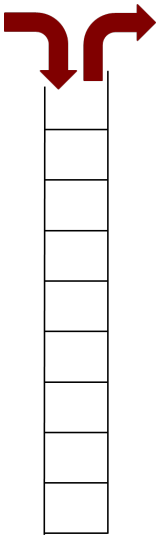
7.5 Pila

La ***pila*** si ispira all'intuitivo concetto di una pila di oggetti: ad esempio, si pensi ad una pila di piatti: ne aggiungiamo uno appog-

giandolo sopra quello in cima alla pila, e quando dobbiamo prenderne uno preleviamo quello più in alto.
La pila è una struttura dati che esibisce un comportamento ***LIFO*** (***Last In First Out***). In altre parole, la pila ha la seguente proprietà:

> gli elementi vengono prelevati dalla pila nell'ordine inverso rispetto a quello col quale vi sono stati inseriti.

Un esempio di utilizzo di questa struttura dati è la pila di sistema, con la quale vengono ad esempio gestite le chiamate a funzione (ricorsive e non).
Alla fine di questo paragrafo si parlerà più diffusamente di questo utilizzo.
Un altro esempio è la struttura dati che memorizza gli URL attraverso cui navighiamo e che possiamo recuperare tramite il pulsante *Back*: la pagina corrente è in cima alla pila e la prima pagina visitata è in fondo.
Su una pila sono definite solo due operazioni: l'inserimento (che di norma viene chiamata ***Push***) e l'estrazione (che di norma viene chiamata ***Pop***). Non è previsto né scandire gli elementi di una pila né eliminare elementi con mezzi diversi dalla Pop.
La particolarità che garantisce la proprietà LIFO è che le operazioni Push e Pop operano sulla stessa estremità della pila (attraverso il puntatore o l'indice top, a seconda che sia implementata tramite una pila o un array).



Lo pseudocodice delle due operazioni nel caso di implementazione tramite lista è riportato nel seguito. L'elemento da inserire, puntato da e, si presuppone già creato, con valore None nel suo campo next.

```
Funzione Push_Lista (top, e):
   e.next = top
   top = e
   return top

Funzione Pop_Lista (top)
   if (top == None)
      print('Errore: pila vuota')
      return None,None
   e = top
   top = e.next
   e.next = None
   return e,top
```

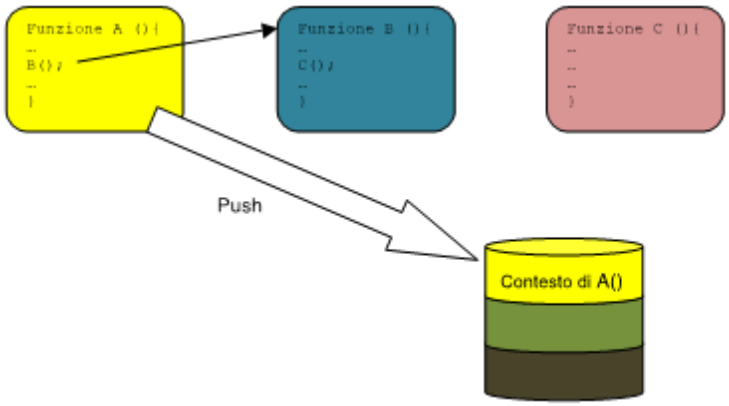
Il costo computazionale di entrambe le operazioni è $\Theta(1)$.

La pila è una struttura dati di grande importanza perché riveste un ruolo centrale in relazione al funzionamento stesso dei moderni elaboratori. Quando un programma in esecuzione deve effettuare una chiamata a una funzione (o procedura) *A()*, deve essere eseguito il cosiddetto ***context switching***, ossia ***cambio di contesto***, perché il controllo della CPU deve essere trasferito dal flusso di esecuzione corrente alla funzione *A()* che viene chiamata. Questo, ad esempio, significa che la funzione *A()* modificherà fra le altre cose:

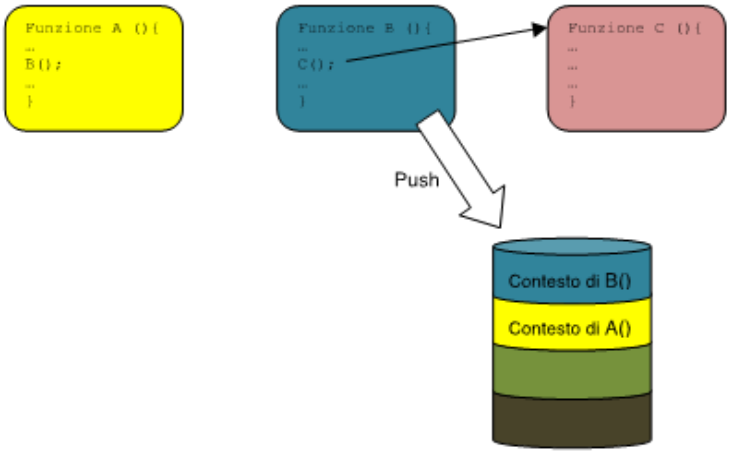
- il ***contatore di programma*** (l'indirizzo di memoria della prossima istruzione da eseguire);
- il contenuto dei registri della CPU.

Tutte queste informazioni, che fanno parte del cosiddetto ***contesto di esecuzione***, devono quindi essere salvate prima di cedere il controllo alla funzione *A()* altrimenti, nel momento in cui *A()* terminerà ed il controllo dovrà essere restituito al programma chiamante, esso non potrà riprendere l'esecuzione in modo corretto.
Naturalmente, la funzione *A()* potrebbe a sua volta chiamare un'altra funzione *B()*, per cui anche il contesto di *A()* dovrà essere salvato prima che si possa iniziare ad eseguire *B()*.
Infine, dato che i ritorni dalle varie chiamate di funzione si verificano esattamente nell'ordine inverso rispetto all'ordine in cui sono avvenute le chiamate stesse, la struttura dati nella quale si salvano i vari contesti di esecuzione deve essere una pila. Le figure seguenti mostrano un esempio di tale situazione.

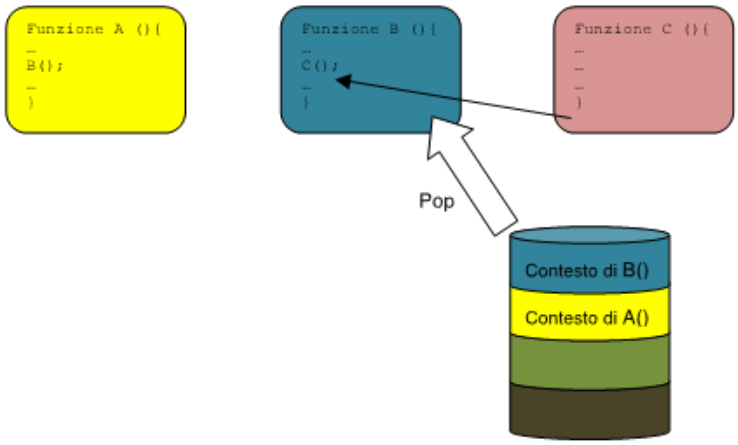
La funzione A() contiene una chiamata a B(); immediatamente prima di eseguire tale chiamata, il sistema operativo salva su una pila di sistema il contesto di esecuzione di A():



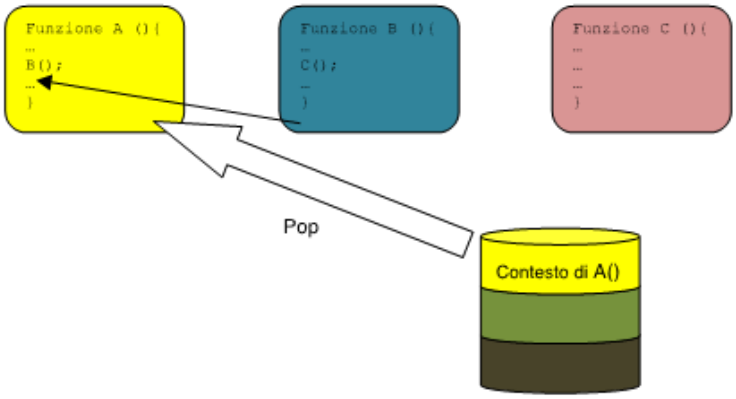
La funzione B() contiene una chiamata a C(); immediatamente prima di eseguire tale chiamata, il sistema operativo salva su una pila di sistema il contesto di esecuzione di B():



La funzione C() termina; immediatamente prima di restituire il controllo a B(), il sistema operativo recupera dalla pila di sistema il contesto di esecuzione di B() e lo ricarica nella cpu:



La funzione B() termina; immediatamente prima di restituire il controllo ad A(), il sistema operativo recupera dalla pila di sistema il contesto di esecuzione di A() e lo ricarica nella CPU:

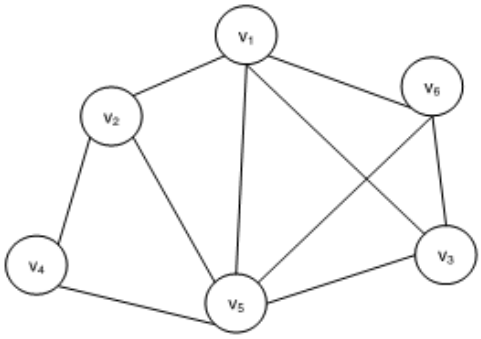


Inoltre, anche nell'ambito del progetto di algoritmi la pila è spesso molto utile. Come vedremo, essa è in molti casi lo strumento più adatto per trasformare con relativa facilità soluzioni di natura ricorsiva in soluzioni di tipo iterativo.

Concludiamo questo paragrafo osservando che, nel caso della **pila**, possiamo sfruttare i metodi dell'oggeto *list* del Python così come sono, infatti *append()* corrisponde all'operazione di push(), ed esiste *pop()*. Entrambi questi metodi possono essere eseguiti in tempo costante.
Non altrettanto si può dire per la **coda**: gli *append()* e i *pop()* alla fine di una struttura *list* sono veloci, ma gli *insert()* ed i *pop()* all'inizio hanno costo lineare, perché tutti gli elementi che seguono devono slittare di una posizione; se vogliamo implementare la coda su un oggetto *list* e garantire che le operazioni abbiano costo costante, dobbiamo scrivere la funzione Dequeue in modo che gli elementi rimasti in coda non slittino, ma rimangano dove sono.

7.6 Albero

L'***albero*** è una struttura dati estremamente versatile, utile per modellare una grande quantità di problemi reali e progettare le relative soluzioni algoritmiche.
Abbiamo già incontrato la struttura ad albero (in particolare ad albero binario) varie volte, ma l'abbiamo sempre considerata in modo intuitivo. Per dare la definizione formale di albero è necessario prima fornire alcune definizioni relative ad un'altra stuttura dati, il ***grafo***, che sarà discussa più approfonditamente nel seguito.



Un ***grafo*** *G = (V, E)* è costituito da una coppia di insiemi:

- un insieme finito *V* dei ***nodi***, o ***vertici***;
- un insieme finito $E \subseteq VxV$ di ***coppie non ordinate di nodi***, dette ***archi*** o ***spigoli***.

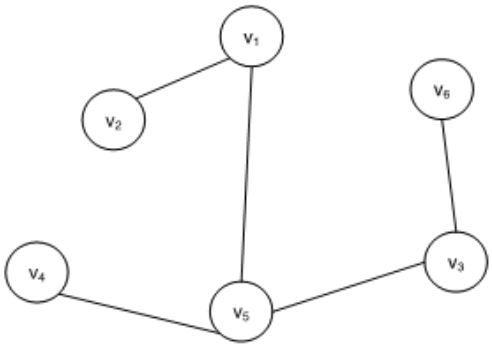
Un ***cammino*** in un grafo *G = (V, E)* è una sequenza $(v_1, v_2, \dots, v_k)$ di nodi distinti di *V* tale che (v_i, v_{i+1}) sia un arco di *E* per ogni $1 \leq i \leq k-1$.

Se ad un cammino $(v_1, v_2, \dots, v_k)$ si aggiunge l'arco(v_i, v_1) si parla di ***ciclo***.

Un grafo *G* è ***connesso*** se, per ogni coppia di nodi *(u, v)*, esiste un cammino tra *u* e *v*. Un grafo G è ***aciclico*** se non contiene cicli.

Sulla base delle precedenti definizioni possiamo dare la seguente:

Definzione. *Un albero è un grafo G = (V, E) connesso e aciclico.*



Lemma. *Sia G = (V,E) un grafo connesso aciclico; eliminando da G un arco qualsiasi, G si disconnette, cioè si suddivide in due grafi* $G_1 = (V_1, E_1)$ *e* $G_2 = (V_2, E_2)$*, entrambi connessi e aciclici.*

Dimostrazione. Supponiamo, per assurdo, che dopo l'eliminazione dell'arco *e = (u,v)* il grafo rimanga connesso; questo significa che nel nuovo grafo privato di *e* esiste un cammino da *u* a *v*.

Ma allora, nel grafo originario *G*, tale cammino, unitamente all'arco *e*, forma un ciclo, contro l'ipotesi che *G* sia aciclico.

Infine, banalmente entrambe le componenti generatesi devono rimanere connesse e acicliche. CVD

Il seguente teorema dà una fondamentale caratterizzazione degli alberi.

Teorema: *Sia G = (V,E) un grafo. Le seguenti due affermazioni sono equivalenti.*

1. *G è* connesso e aciclico (in altre parole, *G* è un albero).
2. *G è* connesso ed *|E| = |V| - 1*.

Dimostrazione. **1.** ⇒ **2.** Dimostreremo per induzione che, se *G* è aciclico, allora *|E| = |V| - 1.*

Passo base: se $|V| = 1$ oppure $|V| = 2$ l'affermazione è banalmente vera.
Passo induttivo: rimuovendo un arco qualsiasi, per il lemma provato precedentemente, il grafo G si disconnette in due grafi $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$, entrambi connessi e aciclici.
Per i due grafi G_1 e G_2 vale l'ipotesi induttiva, e quindi:

$$|E_1| = |V_1| - 1 \quad \text{e} \quad |E_2| = |V_2| - 1$$

Osserviamo ora che:

$$|V| = |V_1| + |V_2|$$

$$|E| = |E_1| + |E_2| + 1 = |V_1| - 1 + |V_2| - 1 + 1 = |V| - 1$$

2. $\Rightarrow$ **1**. Il grafo $G = (V, E)$ connesso, con $|V| = n$ e con $|E| = |V| - 1$ per assurdo contenga un ciclo. Sia $(v_1, v_2, \dots, v_k, v_1)$ tale ciclo. Consideriamo il grafo $G_k = (V_k, E_k)$ costituito dal solo ciclo. In esso abbiamo:

$$|E_k| = |V_k|$$

Se $k = n$ si contraddice l'ipotesi iniziale che sia $|E| = |V| - 1 = n - 1$. Se invece $k < n$ allora deve esistere un nodo v_{k+1} connesso a G_k tramite un arco (poiché G è connesso). Dunque, sia G_{k+1} il grafo costituito da G_k più il nodo v_{k+1} più l'arco che collega v_{k+1} a G_k; esso ha un numero di archi pari a:

$$|E_{k+1}| = |V_{k+1}|$$

Proseguiamo in questo modo fino a costruire $G_n = (V_n, E_n)$. In esso si avrà:

$$V = V_n \quad \text{ed} \quad E \supseteq E_n \Rightarrow |E| \geq |E_n|$$

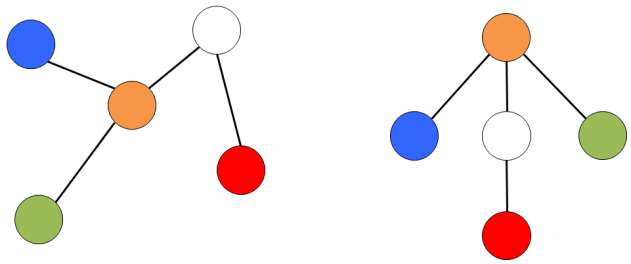
Per ipotesi abbiamo $|E| = |V| - 1$, per cui si avrebbe:

$$|V| - 1 = |E| \geq |E_n| = |V_n| = |V|$$

il che è chiaramente assurdo. CVD

7.6.1 Alberi binari

Una particolare categoria di alberi è quella degli ***alberi radicati***, in cui cioè si distingue un nodo particolare tra gli altri, detto ***radice***.



Allora l'albero si può rappresentare in modo tale che i cammini da ogni nodo alla radice seguano un percorso dal basso verso l'altro, come se l'albero venisse, in qualche modo, "appeso" per la radice, come mostrato nella precedente figura, in cui viene rappresentato lo stesso albero, non radicato (a sinistra) e radicato (a destra). In un albero radicato, i nodi sono organizzati in ***livelli***, numerati in ordine crescente allontanandosi dalla radice (di norma la radice è posta a livello zero); l'***altezza*** di un albero radicato è la lunghezza del più lungo cammino dalla radice ad una foglia; si noti che un albero di altezza h contiene $(h + 1)$ livelli, di norma numerati *da 0 ad* h.

Dato un qualunque nodo v di un albero radicato che non sia la radice, il primo nodo (unico) che si incontra sul cammino da v alla radice viene detto ***padre di v;*** nodi che hanno lo stesso padre sono detti ***fratelli*** e la radice è l'unico nodo che non ha padre. Ogni nodo sul cammino da v alla radice viene detto ***antenato di v***. Viceversa, tutti i nodi che ammettono v come padre sono detti ***figli di v***, ed i nodi

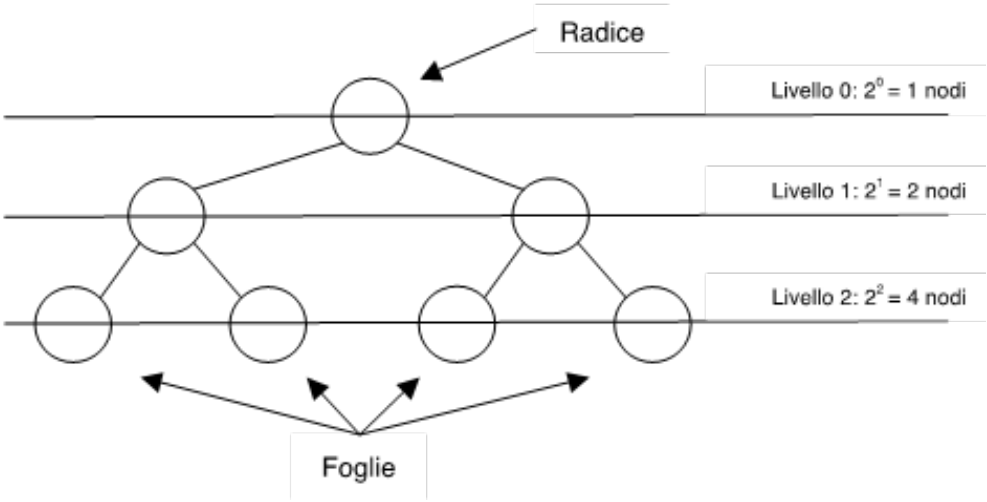
che non hanno figli sono detti ***foglie***; tutti i nodi che ammettono *v* come antenato vengono detti ***discendenti di v.***

Si noti, in questa nomenclatura, una non casuale somiglianza con le genealogie che, in effetti, vengono rappresentate proprio tramite alberi radicati.

Un albero radicato si dice ***ordinato*** se attribuiamo un qualche ordine ai figli di ciascun nodo, nel senso che se un nodo ha *k* figli, allora vi è un figlio che viene considerato primo, uno che viene considerato secondo, ..., uno che viene considerato *k*-esimo.

Una particolare sottoclasse di alberi radicati e ordinati è quella degli ***alberi binari***, che hanno la particolarità che ogni nodo ha al più ***due figli***. Poiché sono alberi ordinati, i due figli di ciascun nodo si distinguono in ***figlio sinistro*** e ***figlio destro.***

Un albero binario nel quale tutti i livelli contengono il massimo numero possibile di nodi è chiamato ***albero binario completo***. Se invece tutti i livelli tranne l'ultimo contengono il massimo numero possibile di nodi mentre l'ultimo livello è riempito completamente da sinistra verso destra solo fino ad un certo punto, l'albero è chiamato ***albero binario quasi completo***. La figura seguente illustra un ***albero binario completo di altezza 2***, costituito quindi di tre livelli (0, 1, 2).



Con semplici tecniche di conteggio, è facile vedere che in un un albero binario completo di altezza h si ha che:

- il numero delle foglie è 2^h;
- il numero dei nodi interni è $\sum_{i=0}^{h-1} 2^i = \frac{2^h-1}{2-1} = 2^h - 1$;
- il numero totale dei nodi è $2^h + 2^h - 1 = 2^{h+1} - 1$.

Stimiamo ora l'altezza di un albero binario completo contenente n nodi.

Poiché l'albero è completo, il numero di nodi è:

$$n = 2^{h+1} - 1$$

e dunque, passando ai logaritmi, abbiamo:

$$\log(n + 1) = h + 1 \quad \text{da cui:} \quad h = \log(n+1) - 1 = \log \frac{n+1}{2}$$

Attenzione a un dettaglio: alcuni testi definiscono l'altezza come il numero dei livelli anziché la lunghezza del più lungo cammino, otteniamo delle relazioni diverse perché in tal caso:

- il numero delle foglie è 2^{h-1};

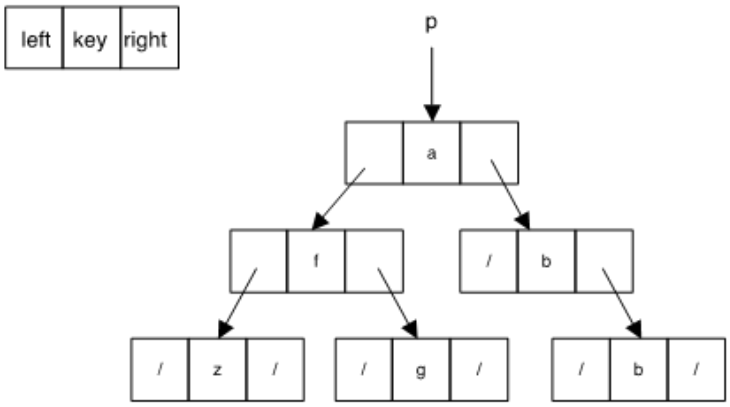
- il numero dei nodi interni è $2^{h-1} - 1$;
- il numero totale dei nodi è $2^h - 1$;
- l'altezza di un albero binario pieno di n nodi è $log(n + 1)$.

Entrambe le notazioni sono corrette ed usate in letteratura, quindi, grande attenzione a definire sempre cosa si intende per altezza!

7.6.2 Rappresentazione in memoria degli alberi binari

Abbiamo accennato altre volte alla differenza tra struttura dati astratta e sua implementazione.
Fin ora abbiamo parlato di alberi, indipendentemente dalla loro rappresentazione in memoria che, tuttavia, è necessaria, se vogliamo risolvere dei problemi su alberi tramite algoritmi.
Nel seguito, descriviamo alcuni modi classici per memorizzare gli alberi binari.
Memorizzazione tramite record e puntatori: Il modo più naturale di rappresentare e gestire gli alberi binari è per mezzo dei puntatori. Ogni singolo nodo è costituito da un record contenente:

- ***key***: le opportune informazioni pertinenti al nodo stesso;
- ***left***: il puntatore al figlio sinistro (oppure *NULL* se il nodo non ha figlio sinistro);
- ***right***: il puntatore al figlio destro (oppure *NULL* se il nodo non ha figlio destro);



Si accede all'albero per mezzo del puntatore alla radice.

Questo tipo di memorizzazione ha tutti i vantaggi e l'elasticità delle strutture dinamiche basate sui puntatori (si possono inserire nuovi nodi, spostare dei nodi ecc.), ma ne presenta svantaggi moltiplicati: l'unico modo per accedere all'informazione memorizzata in un nodo è scendere verso di esso partendo dalla radice e poi spostandosi di padre in figlio, ma non è chiaro se ad ogni passo si debba andare verso il figlio sinistro o verso il figlio destro. Nel seguito vedremo come risolvere questo tipo di problemi.

Rappresentazione posizionale: Un altro modo di rappresentare un albero binario è quello che abbiamo già discusso in merito all'heap: i nodi vengono memorizzati in un array, nel quale la radice occupa la posizione di indice *0* ed i figli sinistro e destro del nodo in posizione *i* si trovano rispettivamente nelle posizioni *2i-1* e *2i*.

Questa rappresentazione presenta in generale i seguenti svantaggi rispetto alla gestione mediante puntatori:

- richiede di conoscere in anticipo la massima altezza *h* dell'albero e, una volta noto tale valore, richiede l'allocazione di

un array in grado di contenere un albero binario completo di altezza *h*;

- a meno che l'albero non sia abbastanza "denso" di nodi, si verifica uno spreco di memoria.

Vettore dei padri: Un ulteriore modo di rappresentare un albero binario è il vettore dei padri. La struttura dati è costituita da un array in cui ogni elemento è associato ad un nodo dell'albero. Introducendo una biezione tra gli *n* nodi dell'albero e gli indici *0, …, n-1*, l'elemento *i* dell'array contiene l'indice del padre del nodo *i* nell'albero. Si osservi che questo metodo di memorizzazione funziona senza alcuna modifica anche per alberi non necessariamente binari, in cui cioè ogni nodo può avere un numero qualunque di figli.

Concludiamo questa carrellata sulle strutture dati per memorizzare gli alberi binari confrontandole tra loro su alcune operazioni tipiche su alberi.

Dato un albero *T* ed un suo nodo *v*, ci possiamo chiedere quale nodo sia suo padre, quanti figli abbia e quale sia la sua distanza dalla radice.

Trovare il padre di un nodo (di cui abbiamo opportune indicazioni: il puntatore al record o l'indice) sulla struttura a puntatori è una cosa che al momento non siamo in grado di fare; infatti dobbiamo accedere all'albero tramite il puntatore alla sua radice, ma poi non possiamo scorrere la struttura come fosse una lista, perché non sappiamo se dirigerci a destra o a sinistra; vedremo nel seguito come procedere, utilizzando un'operazione che ancora non conosciamo,

che prende il nome di *visita dell'albero*, e che viene descritta nel prossimo paragrafo.

Nella memorizzazione tramite array, il padre del nodo con indice *i* è banalmente quello con indice $\lfloor (i - 1)/2 \rfloor$ mentre nel vettore dei padri esso è memorizzato nell'elemento di indice *i*. Impieghiamo quindi in entrambi i casi tempo $\Theta(1)$.

Per **determinare se il nodo *i* abbia 0, 1 o 2 figli**, nella struttura che utilizza i puntatori basta verificare se i campi *left* e *right* siano settati a NULL oppure no; nella struttura vettoriale bisogna vedere se gli elementi di indice $2i-1$ e $2i$ sono settati a 0 oppure no; in entrambi i casi impieghiamo tempo $\Theta(1)$. Nel vettore dei padri, invece, è necessario scorrere l'intero array e contarvi il numero di occorrenze dell'elemento *i*, con un costo computazionale di $O(n)$.

Per calcolare la **distanza dalla radice di un nodo**, quando si ha a disposizione la notazione posizionale, il numero del livello su cui si trova il nodo di indice *i* è pari a $\lfloor log(i+1) \rfloor$ e quindi sappiamo rispondere in tempo costante; quando ci troviamo in presenza di un vettore dei padri *P*, a partire da *P[i]* dobbiamo risalire di padre in padre, passando per *P[P[i]]*, *P[P[P[i]]]*, e così via, fino a giungere alla radice, e compiendo così un lavoro dell'ordine dell'altezza dell'albero; nel caso di struttura a puntatori, infine, abbiamo lo stesso problema che nella ricerca del padre, e dobbiamo quindi ricorrere alle visite.

7.6.3 Visita di alberi binari

Un'operazione basilare sugli alberi, che spesso è il presupposto per poter risolvere problemi su di essi, è l'accesso a tutti i suoi nodi, uno

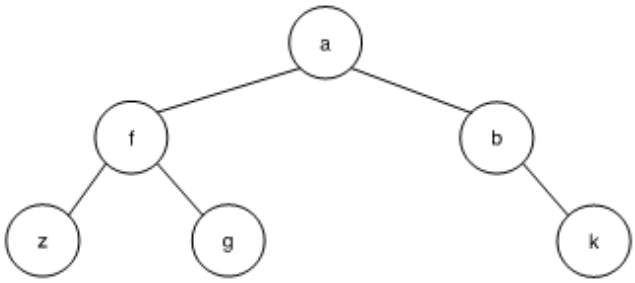
dopo l'altro, al fine di poter effettuare una specifica operazione (che dipende ovviamente dal problema posto) su ciascun nodo. Abbiamo visto come tale operazione sulle liste sia una semplice iterazione, ma sugli alberi la situazione è più complessa dato che la loro struttura è ben più articolata.
L'accesso progressivo a tutti i nodi di un albero si chiama ***visita dell'albero***.

Facendo riferimento all'ordine col quale si accede ai nodi dell'albero, è evidente che esiste più di una possibilità.

Nel caso specifico degli alberi binari, nei quali i figli di ogni nodo (e quindi i sottoalberi) sono al massimo due, e volendo comportarsi nello stesso modo su tutti i nodi, le possibili decisioni in merito a questa scelta danno luogo a tre diverse visite:

- ***visita in preordine (preorder):*** il nodo è visitato prima di proseguire la visita nei suoi sottoalberi;
- ***visita inordine (inorder):*** il nodo è visitato dopo la visita del sottoalbero sinistro e prima di quella del sottalbero destro;
- ***visita postordine (postorder):*** il nodo è visistato dopo dopo entrambe le visite dei sottoalberi.

Consideriamo l'albero binario seguente.

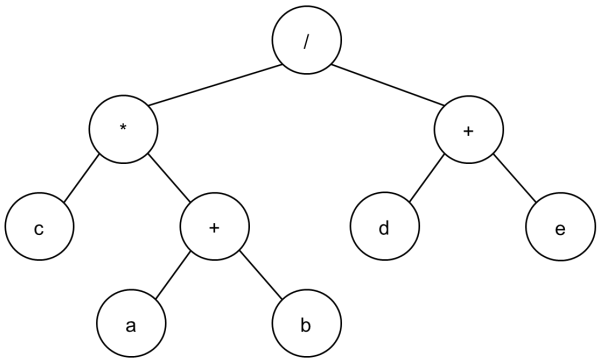


Le tre visite sopra elencate, eseguite su tale albero, visiterebbero i suoi nodi seguendo un ordine diverso:

- visita in preordine: **a f z g b k**
- visita in inordine: **z f g a b k**
- visita in postordine: **z g f k b a**

Alcuni esempi d'uso delle visite sono i seguenti.

- La visita inordine applicata agli alberi binari di ricerca (che saranno illustrati approfonditamente nel par. 8.3) visita i nodi secondo l'ordinamento dei relativi valori.
- Se l'albero contiene una espressione aritmetica, come quello in figura:



la visita in preordine permette di costruire la forma prefissa dell'espressione, e la visita in postordine permette di calcolarla.

Lo pseudocodice, ricorsivo, delle tre visite su un albero binario implementato mediante puntatori è il seguente. Si suppone l'albero già esistente ed accessibile mediante il puntatore *p*. L'unica differenza fra i tre casi è la posizione della pseudoistruzione relativa all'accesso al nodo per effettuarvi le operazioni desiderate.

```
def Visita_preordine (p):
  if (p != None):
    # qui istruzioni di accesso al nodo e operazioni conseguenti
    Visita_preordine (p.left)
    Visita_preordine (p.right)
  return

def Visita_inordine (p):
  if (p != None):
    Visita_inordine (p.left)
    # qui istruzioni di accesso al nodo e operazioni conseguenti
    Visita_inordine (p.right)
  return

def Visita_postordine (p):
  if (p != None):
    Visita_postordine (p.left)
    Visita_postordine (p.right)
    # qui istruzioni di accesso al nodo e operazioni conseguenti
  return
```

Valutiamo il costo computazionale delle visite, che è ovviamente lo stesso per tutte e tre. Esso varia al variare della struttura dati

utilizzata per memorizzare l'albero. Nel caso di memorizzazione tramite record e puntatori, detto k il numero di nodi del sottoalbero sinistro della radice, l'equazione di ricorrenza è:

- $T(n) = T(k) + T(n - k - 1) + \Theta(1)$
- $T(0) = T(1) = \Theta(1)$

L'unico metodo per risolvere questa ricorrenza è quello di sostituzione. Dobbiamo quindi farci un'idea di quale possa esserne la soluzione. Procediamo intuitivamente. Studiamo due casi estremi, in cui l'algoritmo ricorsivo viene richiamato sulle parti sinistra e destra dell'input iniziale che sono, rispettivamente, perfettamente bilanciate o totalmente sbilanciate. Il primo caso corrisponde all'albero completo. In tal caso, se h è il numero dei suoi livelli, si ha $n = 2^{h+1} - 1$ ed entrambi i sottoalberi di ogni nodo sono completi. L'equazione quindi diviene: $T(n) = 2T(n/2) + \Theta(1)$. Questa equazione ricade nel caso 1 del teorema principale, ed ha quindi soluzione:
$T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n)$.
Il secondo caso si verifica quando $k = 0$ (oppure, simmetricamente, quando $n-k-1=0$), per cui: $T(n) = T(n - 1) + \Theta(1)$ che ha banalmente, ad esempio col metodo iterativo, la soluzione: $T(n) = n\ \Theta(1) = \Theta(n)$.
Visto che le due soluzioni coincidono, possiamo ipotizzare che anche il costo computazionale nel caso generale sia $\Theta(n)$.
Ora che abbiamo un'idea della soluzione, risolviamo l'equazione con il metodo di sostituzione. Eliminando la notazione asintotica abbiamo:

- $T(n)=T(k)+T(n-1-k)+c$
- $T(1)=d$

Tentiamo la soluzione $T(n) \leq an$ per qualche costante a da determinare. Sostituendo nel caso base otteniamo $d \leq a$, mentre sostituendo nel caso generico l'ipotesi induttiva: $T(k)+T(n-1-k)+c \leq ak+a(n-1-k)+c=an-a+c \leq an$ se e solo se $c \leq a$. Poiché entrambe le disuguaglianze sono ammissibili, si deduce che $T(n)=O(n)$.
Con analoghe considerazioni si giunge a dimostrare che $T(n) \geq bn$ da cui si deduce che $T(n)=\Omega(n)$ concludendo quindi che $T(n)=\Theta(n)$.

7.6.4 Applicazione delle visite di alberi

Le visite di alberi sono uno strumento estremamente utile per ispezionare l'albero in input e dedurne delle proprietà. A seconda della proprietà, può essere più utile una delle tre visite studiate, come mostrano i seguenti esempi, nei quali si predilige la chiarezza espositiva rispetto alla brevità del codice.

Esempio 7.1: Dato un albero binario T tramite il puntatore p alla sua radice, si calcoli il numero dei suoi nodi.

Soluzione. Si può procedere secondo il seguente pseudocodice:

```
def Calcola_n (p):
  if (p != None):
    num_l = Calcola_n(p.left)
    num_r = Calcola_n(p.right)
    num = num_l+num_r+1  /* accesso al nodo
    return num
  else return 0
```

Questa funzione può ovviamente essere intesa come una visita in postordine, visto che, per calcolare il numero di nodi in un sottoalbero,

è necessario conoscere il numero di nodi nei suoi sottoalberi sinistro e destro.

Esempio 7.2: Dato un albero binario T tramite il puntatore p alla sua radice ed un valore k, si restituisca TRUE se k è tra le chiavi memorizzate in T, FALSE altrimenti.

Soluzione. Si può procedere secondo il seguente pseudocodice:

```
def Cerca(p, k)
  if (p != None)
    if (p.info=k): return TRUE
    else: if (Cerca(p.left,k)==TRUE):
                return TRUE
            else: return Cerca(p.right,k)
```

Questa funzione può ovviamente essere intesa come una visita in preordine.
