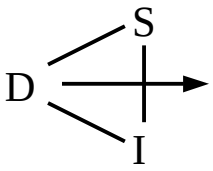


Fondamenti di Informatica
Parte 4

CCL - Amb-Ter

Facoltà di Ingegneria
Università degli Studi di Firenze

Ver.: 0.01
Data: 3/6/2000

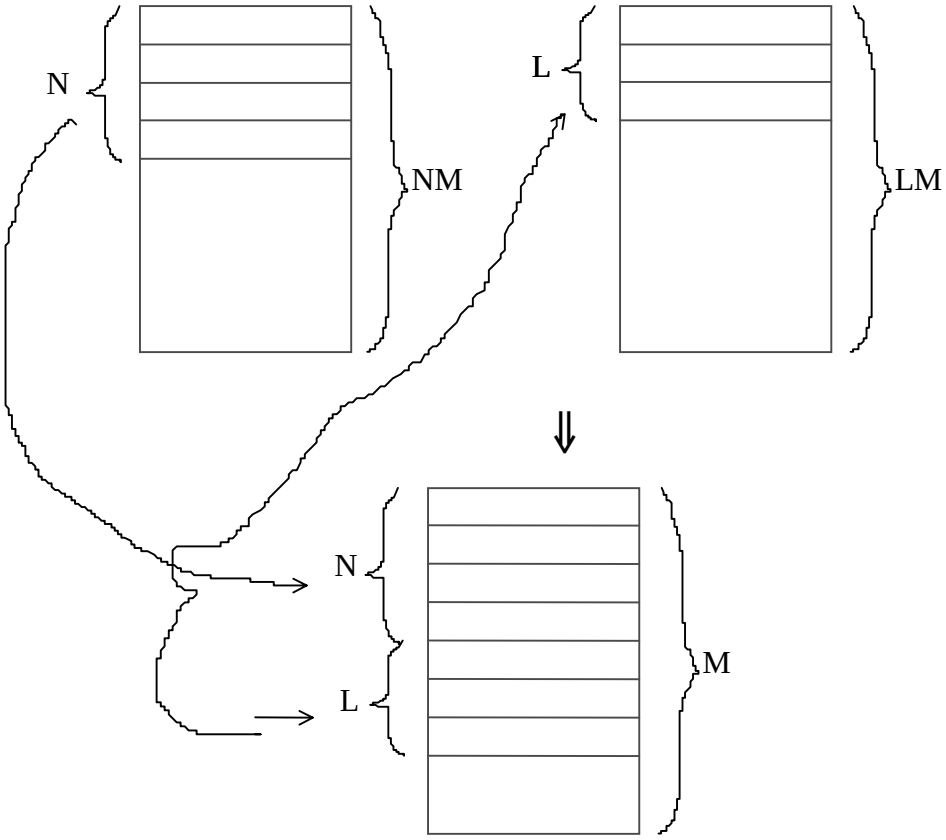
Questo documento fa parte di una serie di dispense prodotte dallo sforzo volontario di molti (in questa parte in particolare vi sono molti pezzi estratti da lezioni trascritte da nastri) ma che sono state riviste e corrette dal docente solo in parte. Anche se in questo stato preliminare ho deciso di distribuirle per dare comunque una traccia sugli argomenti del corso in mancanza di un libro di riferimento. Non rappresentano pertanto ne' il libro ne' il programma di riferimento del corso, a questo fine fanno fede le lezioni stesse che in molte parti differiranno da queste dispense.
Si prega pertanto di segnalare ogni mancanza e correzione inviando una e-mail al Prof. P. Nesi al seguente indirizzo di posta elettronica: nesi@dsi.unifi.it , con la speranza di arrivare a produrre una versione piu' corretta e completa in breve tempo, anche con il Vostro aiuto, grazie!

INDI CE

1 Catene

1.1 Concatenazione di vettori

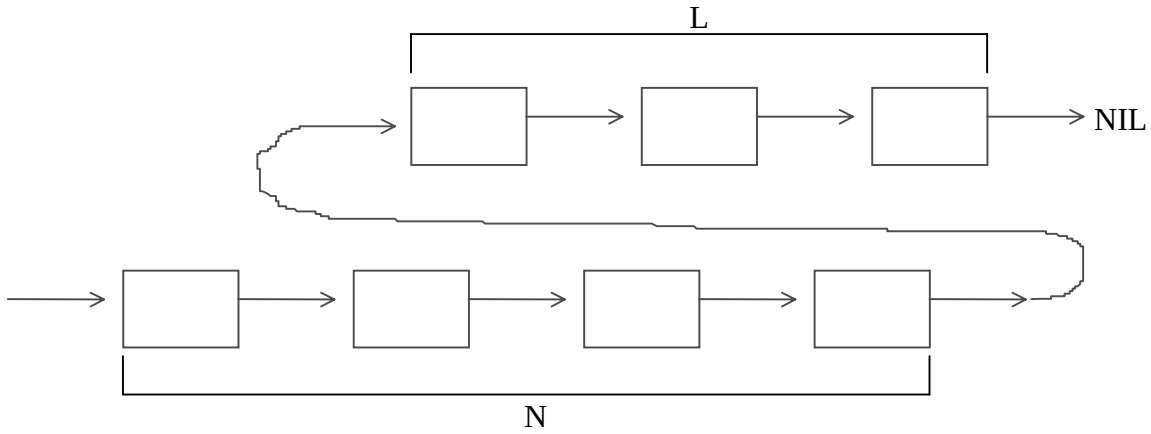
L'operazione di concatenazione consiste nel realizzare una struttura dati combinando altre due strutture con una operazione per la quale prima vengono acquisiti gli elementi della prima struttura ed in seguito vengono inseriti in sequenza gli elementi della seconda.
Si abbia un oggetto/struttura dati che contenga un vettore di N elementi base su NM possibili posizioni.
Si abbia poi un'altra struttura dati con L nominativi su LM possibili.
Vogliamo unire le due agende per ottenere un'unica agenda contenuta in un unico vettore:



Tutto ciò è possibile se N + L <= M; e come caso limite se NM+LM<=M.
Nell'operazione descritta sopra vengono effettuate L + N operazioni di copia. Tale operazione risulta essere la dominante.

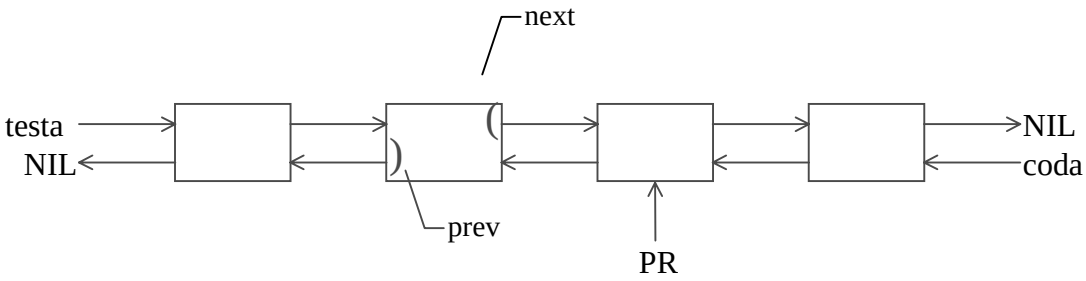
1.2 Concatenazione di Liste

Vediamo adesso il caso di liste, in cui si hanno due catene/liste separate: l'operazione di concatenazione risulta molto semplice, poiché è sufficiente che l'ultimo puntatore di una catena vada a puntare all'inizio dell'altra, come illustrato nella figura seguente:

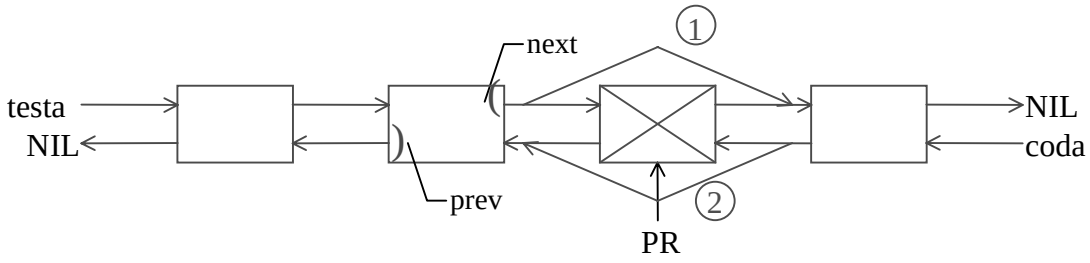


1.3 Catena doppia o lista bidirezionale

E' fatta nel seguente modo:



In tale catena si può accedere sia dalla coda che dalla testa (dove coda e testa sono messe in modo convenzionale).
Anche su questo tipo di struttura bidirezionale si possono realizzare code, pile, stack, ecc.
Il vantaggio della lista bidirezionale è che utilizzando questo tipo di struttura è sufficiente avere il puntatore all'elemento stesso per poterlo cancellare e che le operazioni di ricerca possono essere effettuate nei due sensi.
Quindi, fatta la ricerca e trovato il puntatore all'elemento ricercato, la cancellazione è molto semplice poiché non deve essere modificata la procedura di ricerca che è identica a quella vista in precedenza.
Vediamo un esempio di *cancellazione* (interventi in verde nella fig. seguente):



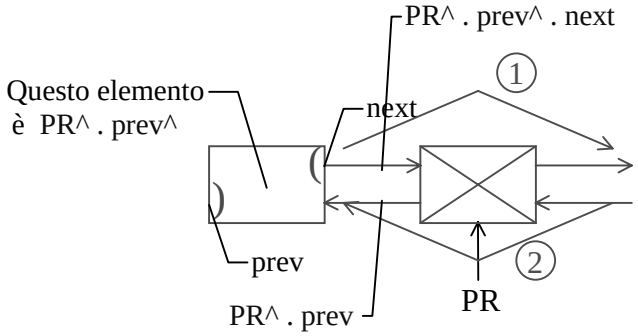
Per cancellare l'elemento con la croce dobbiamo prima stabilire i due legami in verde nella fig. e poi cancellare l'elemento stesso:

```
PR->prev->next  = PR->next ;

PR->next->prev  = PR-> prev ;

Dispose (PR) ;
```

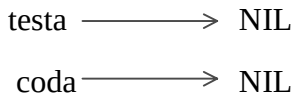
NB:



1.3.1 Costruzione della catena doppia

Viene mostrato ora come si costruisce la catena doppia:

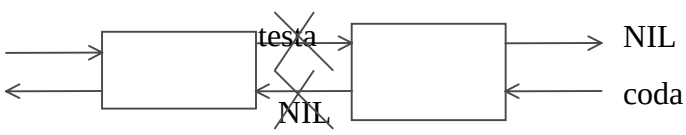
1) Inizialmente non si hanno elementi, i due puntatori della lista puntano a NIL:



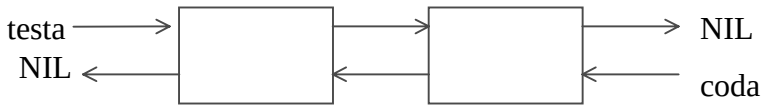
2) Quando si ha un solo elemento, si ha questa situazione:

NOTA: l'inserimento del primo elemento è un caso particolare, e quindi presenta modalità diverse, rispetto all'inserimento dell'elemento generico.

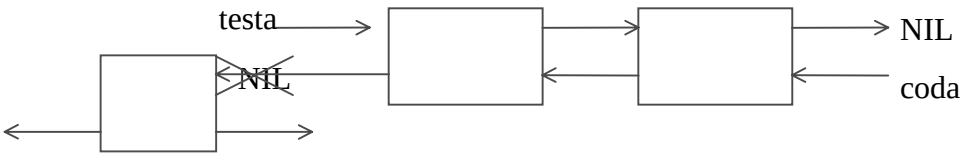
3) Inseriamo poi un altro elemento. E' necessario decidere se fare l'inserimento in testa o in coda; se viene effettuato in testa, si tagliano i due puntatori con la croce e l'elemento viene collegato in tale positione:



Si veda più in dettaglio l'inserimento in testa alla lista bidirezionale.
Si deve avere un puntatore temporaneo all'elemento.



New (testa^ . prev) ; ⇒ così si crea l'elemento in verde nella fig. seguente



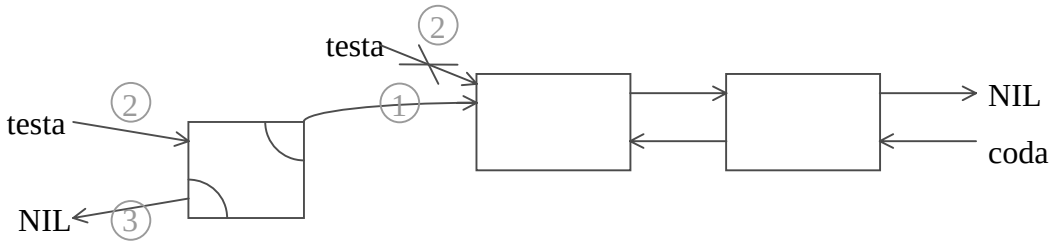
Pertanto, si devono effettuare le seguenti operazioni:

① testa^ . prev^ . next : = testa ;

② testa : = testa^ . prev ;

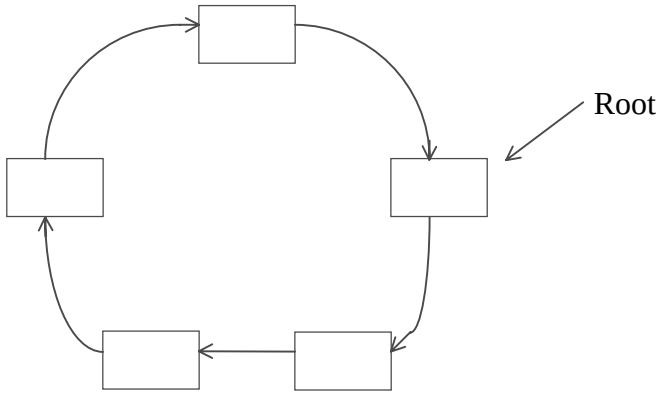
③ testa^ . prev : = NIL ;

Tali operazioni sono visualizzate nella figura seguente:



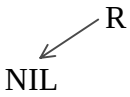
1.4 Catena circolare

La catena circolare puo'essere **semplice** e **doppia**. In tale struttura, l'ultimo elemento non punta a NULL bensì al primo; inoltre per poter accedere alla catena circolare ho bisogno di un puntatore Root/inizio che punta a un elemento come si può vedere dalla figura seguente:

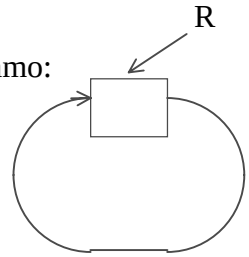


Le operazioni di ricerca, cancellazione, modifica non cambiano rispetto al caso di catena non rishiusa. La ricerca, si può effettuare come in una lista semplice, con un'unica differenza nel criterio di arresto (infatti mentre nella catena semplice ci arrestiamo quando si trova NULL, nella catena circolare ci deve arrestare quando si ritrova Root).

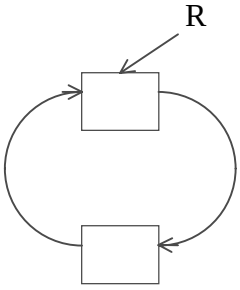
L'operazione di l'*inizializzazione e' effettivamente diversa*:
- all'inizio si ha



- appena viene aggiunto il primo elemento, esso punta a se stesso cioè abbiamo:

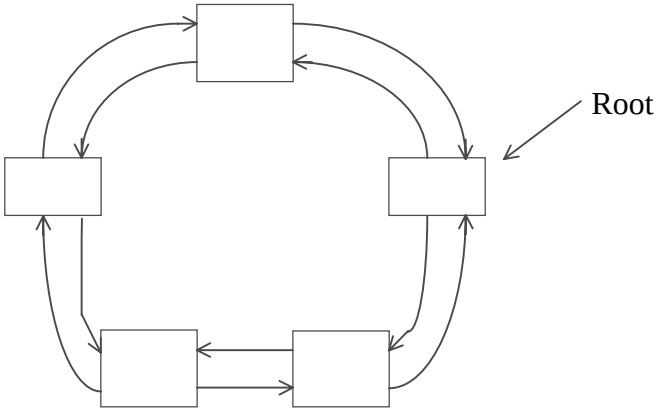


quindi lo stratagemma è che quando si ha un solo elemento bisogna subito cominciare a collegare il next dell'elemento a se stesso. Quando ne aggiungiamo un altro, la struttura continua ad estendersi circolarmente, come si può vedere in figura:

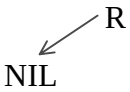


1.5 Catena circolare bidirezionale

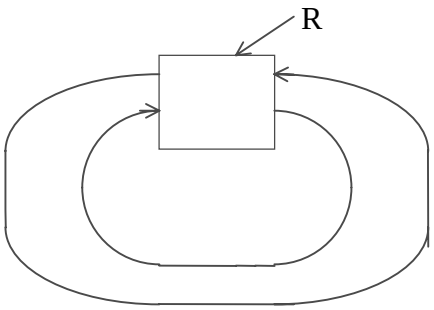
La catena circolare bidirezionale ha puntatori in entrambe le direzioni. Essendo circolare ha la necessita' di avere un solo puntatore per accedere a tutti i suoi elementi.



Inizializzazione della catena circolare
- all'inizio si ha



in tale situazione avremo un inserimento particolare perché sarà quello del primo elemento.
- appena aggiunto il primo elemento, si ha un doppio legame richiuso sul primo elemento che riferisce a se stesso



A questo punto, cioè con almeno un elemento già presente, l'inserimento sarà di tipo generale: esso può essere, per es., l'inserimento in testa alla lista bidirezionale visto in precedenza.

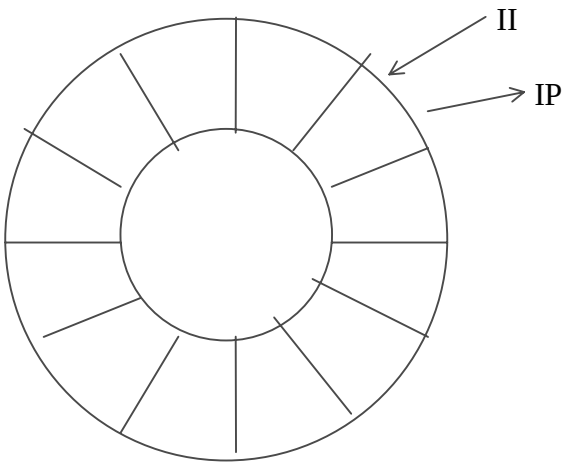
1.6 Buffer (coda) circolare

Il Buffer circolare e' una struttura logica molto simile alla struttura fisica vista in precedenza. Il buffer circolare è stato studiato per risolvere il problema tipico dei buffer o code: esse, specialmente se realizzate con vettori, hanno bisogno ogni tanto di una riorganizzazione perché scorre verso dove si aggiungono le informazioni; invece il buffer circolare non ha tale problema.

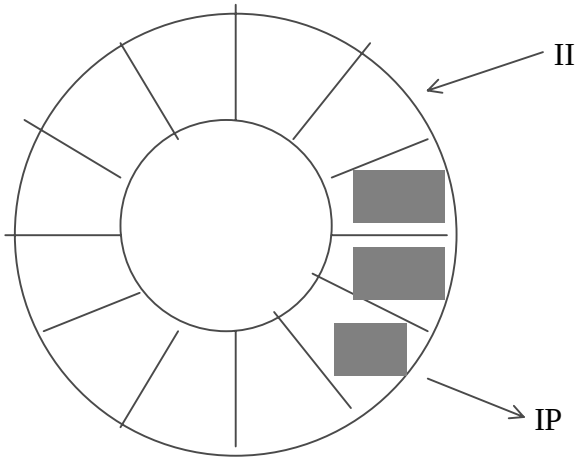
Nel buffer circolare si hanno due indici:

- II = Indice di Immissione : indica, come nella coda, la posizione di inserimento;
- IP = Indice di Prelievo : indica l'elemento da prelevare.

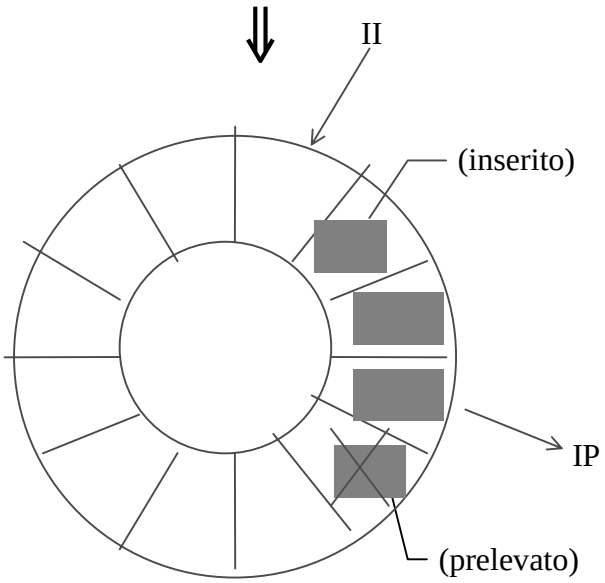
All'inizializzazione non ci sono elementi e si ha II == IP



A regime si hanno degli elementi nella struttura ma questa non e' piena. In tale contesto, IP indica l'ultimo, mentre II è prima dell'ultimo inserito. Ogni volta che viene inserito un elemento l'indice II di immissione viene spostato indietro in senso antiorario. In caso di prelievo si ha uno spostamento dell'indice IP in senso antiorari.
Se gli inserimenti e i prelievi si alternano, ci vede spostare in senso antiorario il gruppo dei dati come viene mostrato nella figura seguente:



Dopo un inserimento e un prelievo



Quindi se nel tempo non si hanno mai più di N informazioni o elementi in coda, allora possiamo usare un buffer circolare di N elementi, che in certi casi sarà quasi pieno e in altri casi tenderà a vuotarsi.

La condizione di controllo è:

- all'inizio si ha II == IP (buffer vuoto);
- se si inseriscono elementi II ruota in senso antiorario e alla fine si sovrappone a IP, II == IP (buffer pieno)

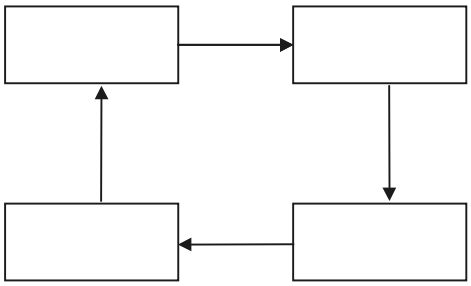
Poiché le condizioni di buffer vuoto e di buffer pieno sono identiche (in entrambe i casi II == IP), dobbiamo riuscire a distinguere le due situazioni.

Alcune possibili soluzione sono o fermare l'inserimento alla posizione precedente a IP ('sprecando' così una posizione), oppure inserire fino a IP mantenendo II spostato avanti.
Per concludere, il buffer circolare è una struttura logica che possiamo realizzare a livello fisico usando o una lista lineare semplice, o una lista doppia, o un vettore a seconda delle necessità.

2 Liste Non lineari

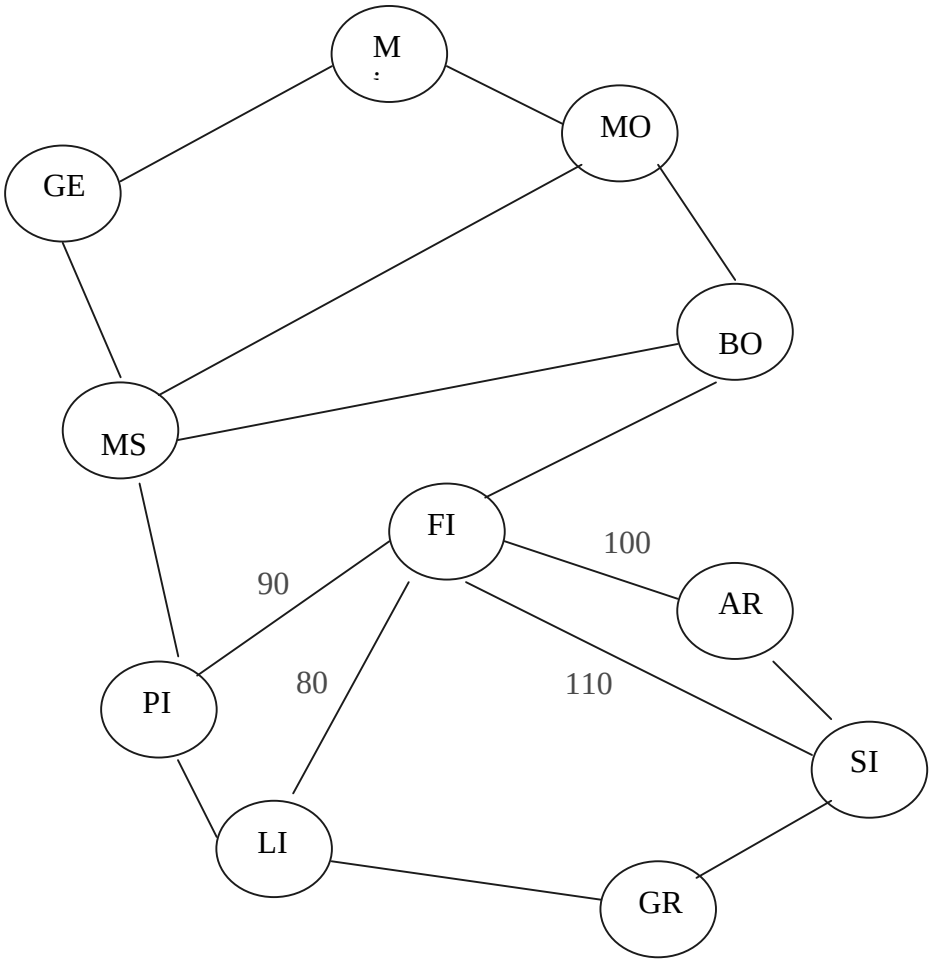
2.1 Grafo

Le liste non lineari sono rappresentabili con *grafi* dove con grafo si intende un insieme di nodi collegati da rami (record/oggetti collegati da puntatori nel nostro esempio).
Il grafo può essere:



Grafo *ciclico*, in quanto presenta un cammino chiuso, al contrario diremo che il grafo è *aciclico*.

Un esempio di come sia possibile rappresentare un tipo di informazione attraverso l'uso di un grafo, è quella di una cartina geografica (che ha le località indicate nei nodi e le distanze in Km, indicate sui rami):



Uno scopo di una tale rappresentazione può essere la realizzazione di un programma che permetta di calcolare un percorso "ottimo" per andare da una località ad un'altra. Al posto delle citta' potrebbero parlare di fenomeni fisici, tra misure ecc. e loro relazioni. Si ricorda che un grafo è orientato se presenta un senso di percorrenza sui rami, in caso contrario si parla di grafo non orientato.

Come possiamo realizzare questa struttura con delle strutture concatenate?
Nel grafo si puo' gestire questo tipo di informazione:

- i nodi che contengono il nome della città e che posso così formalizzare :

```
città = record ;
  nome : = string [ 20 ] ;
  Lrami : Pramo ; ( ogni città ha una lista di città vicine cioè di possibili direzioni di
                    percorrenza  e sarà un puntatore a un ramo )
end;
```

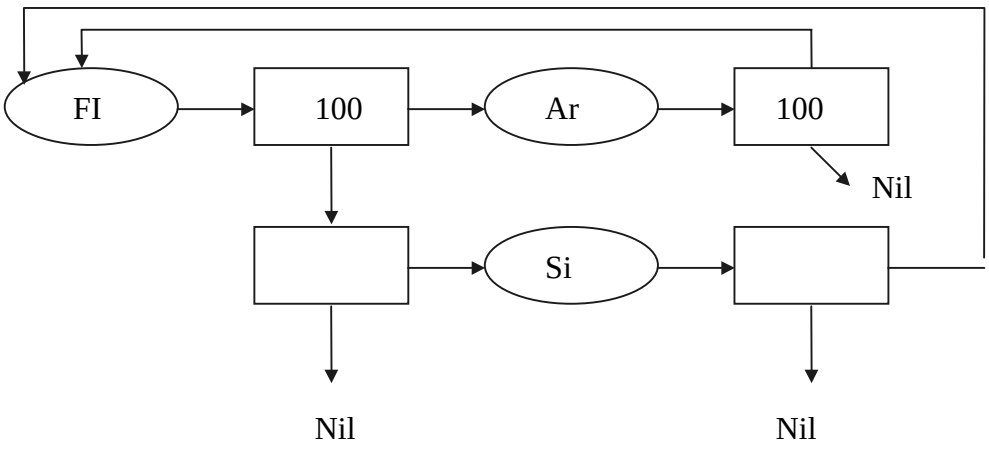
dove il puntatore a ramo sarà:

```
Pramo : = ^ Ramo
Ramo  = record ;
    CS :  Pcittà ( Città Successiva );
    Next : Pramo ;
    Dist:  integer  ;
end;
```

vediamo di chiarire disegnando la città con un cerchio e i rami con dei quadrati:

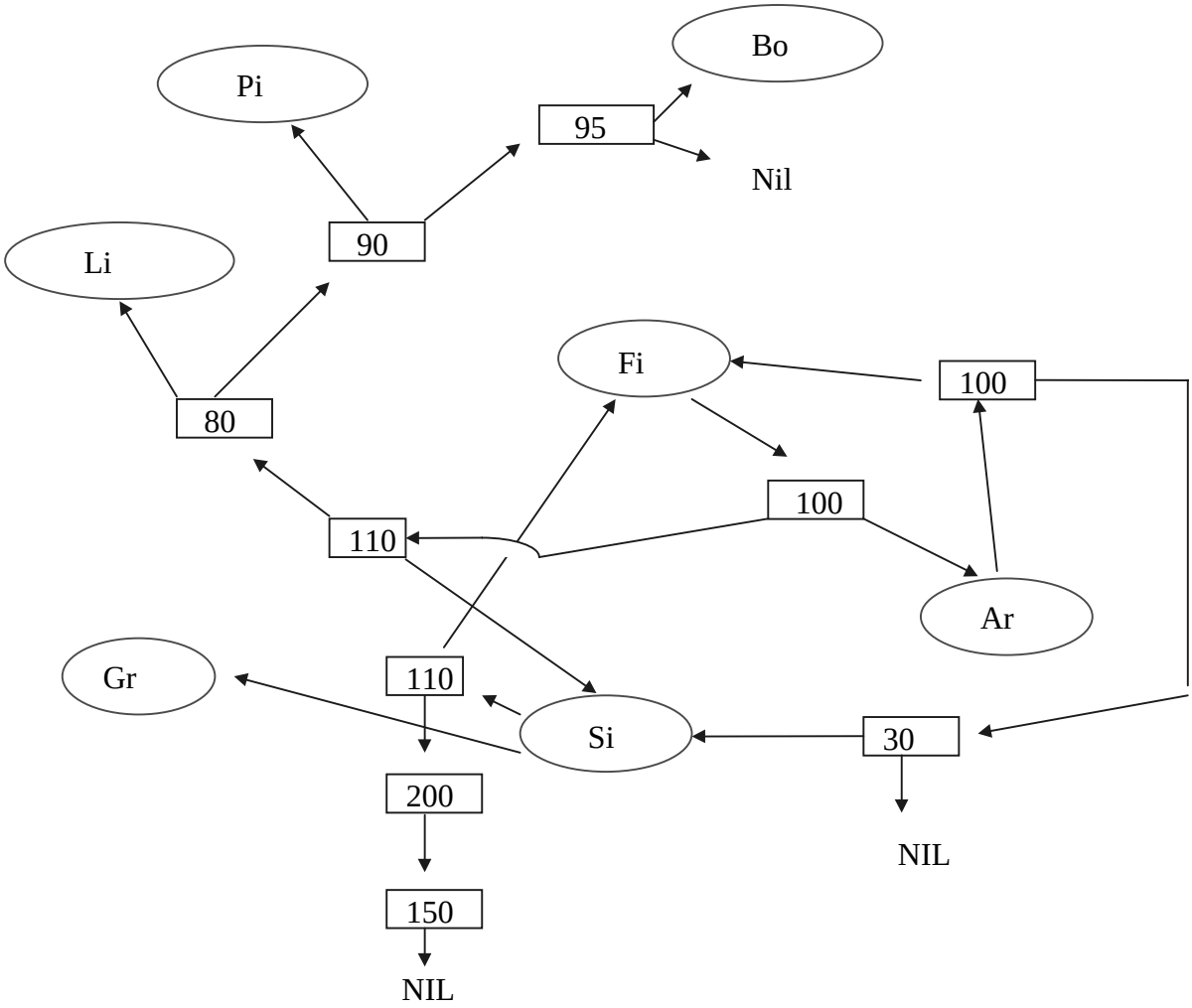
- il ramo ha un puntatore a città ed un puntatore a ramo
- la città contiene una informazione e una lista di rami

Esempio :



Vediamo un po' meglio in dettaglio ad esempio Firenze:

dobbiamo esprimere che Firenze è connessa con Arezzo, Siena, Pisa e Livorno mediante dei rami che contiene delle informazione di distanza, tenendo presenza che il cammino è doppio nel senso che posso andare da Firenze a Siena ma anche da Siena e Firenze:



ecc.

`Questo è solo un esempio, anche se un po' complesso, per far capire quale è il metodo per costruire, con una struttura informativa, la carta geografica..
Quello che ci interessa di più, una volta capito come stanno le cose, è la complessità computazionale della ricerca di un percorso tra due città contigue : dipende ovviamente dal numero delle connessioni; in particolare se N è il numero delle connessioni, al massimo la complessità sarà N.

E se conosciamo esattamente il numero di città che dobbiamo percorrere, ad esempio da Bo a Gr attraverso la strada Bo - Fi - Li - Gr ?

Ragioniamo in generale dicendo che le città da attraversare sono c, la complessità sarà (c - 1) N che asintoticamente diventa o (cN); dove N è il numero medio di connessioni.

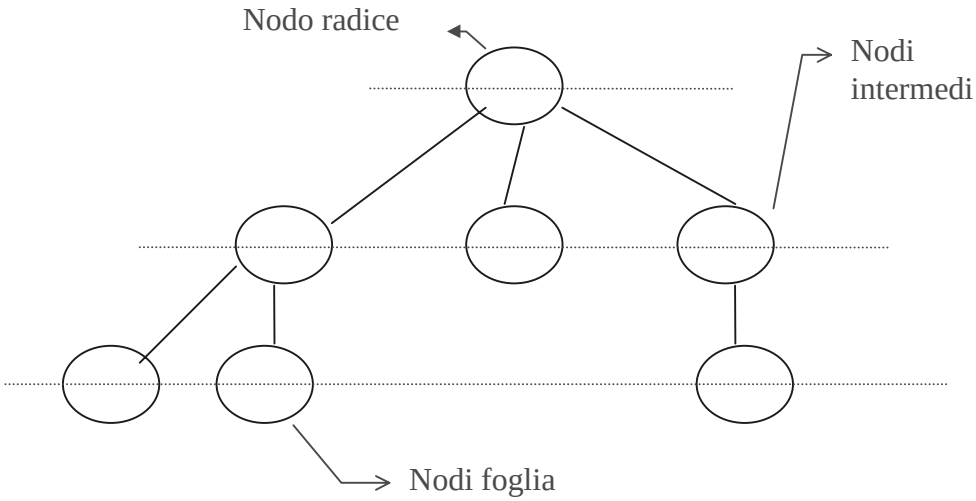
Quanto costa adesso inserire una nuova città ?
Se inserisco ad esempio Empoli, sarà connessa ad un certo numero di città, supponiamo L, quello che facciamo è, per prima cosa allocare il nuovo record dopodiché devo andare in tutte le città a cui collego

Empoli ed inserire un nuovo elemento (Empoli). A questo punto devo decidere dove fare l'inserimento: se lo faccio in testa il tutto mi costa asintoticamente o (L), se lo faccio in coda o ordinato invece, è diverso.

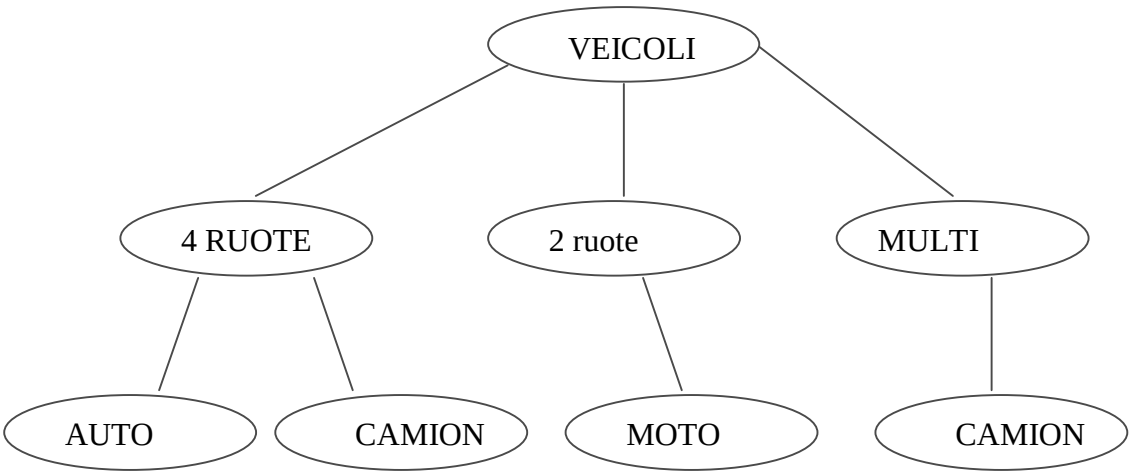
Infatti in questo caso ho : L inserimenti nella propria lista (quella di Empoli) più LN per l'inserimento in tutte le altre liste, in totale asintoticamente o (LN); ripetiamo : questo solo se si ipotizza un inserimento in coda nelle liste delle altre città. (sarebbe una pessima scelta).

2.2 Alberi

Un albero è una struttura con una *radice* (root) da cui si dipartano dei *rami* che possono avere o no dei *sottorami.*
Un esempio, con le relative terminologie, è riportato in figura:

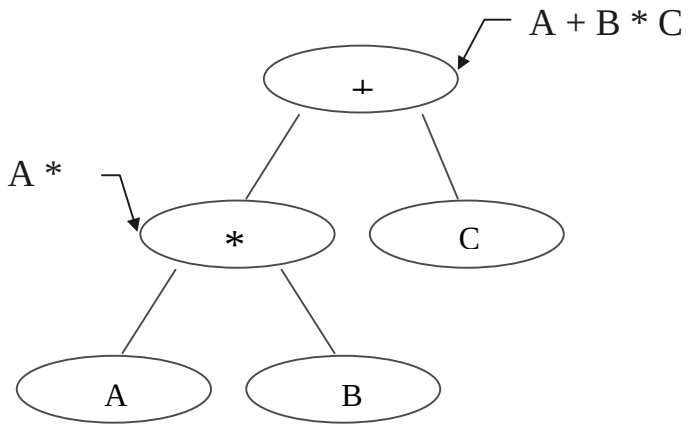


L'utilizzo di un albero è abbastanza frequente; esempi sono quello genealogico, di classificazione ecc.
Esempio di albero di classificazione:



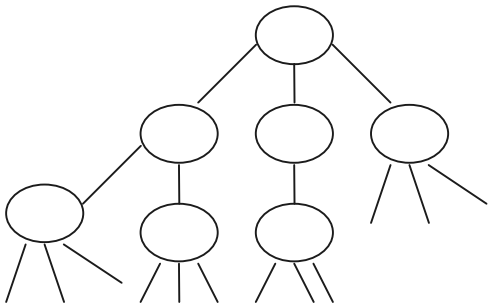
Esempio di albero di classificazione per mezzi stradali

Posso anche usare questa struttura per descrivere l'espressione A * B + C; in questo caso abbiamo un albero sintattico:



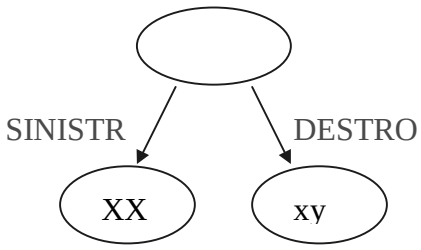
Il questo caso il nodo radice fornisce la soluzione, mentre sulle foglie ci sono le variabili (operandi) e nei nodi intermedi i risultati parziali.
Gli alberi possono avere diverse dimensioni, dove con dimensione si intende o la quantità di nodi che ha tutto l'albero o anche il tipo dell'albero (quanti nodi figli può vedere ogni nodo).

Esempio di albero ternario



Questo albero è ternario perché ogni nodo può attaccare al più tre figli (alcuni nodi possono avere meno di tre figli). Quando il numero dei figli è noto e limitato, si può realizzare la struttura albero con il Pascal in modo molto semplice.

Ad esempio possiamo usare il nodo per contenere l'informazione + 2 puntatori, in questo modo:



Il nodo è visto come un normale record ma con due campi, Destro e Sinistro che sono due puntatori a nodo:

```
Pnodo = ^ nodo
   nodo = record   ( informazione )
   xxxxxxxx
   destro, sinistra = Pnodo
end ;
```

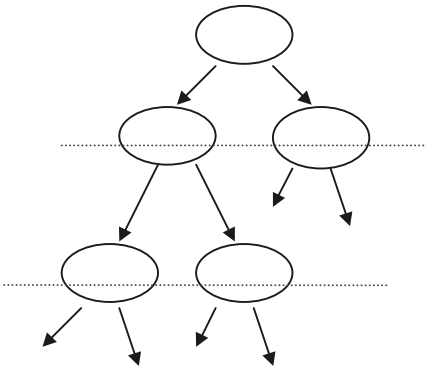
2.3 Alberi binari

In questa sessione vengono mostrate ed approfondite alcune caratteristiche di un albero binario. Queste daranno la possibilità di formalizzarne gli algoritmi per l'attraversamento degli alberi. L'operazione di attraversamento consiste nella navigazione sulla struttura con l'intenzione di toccare, attraversare tutti gli elementi della struttura.

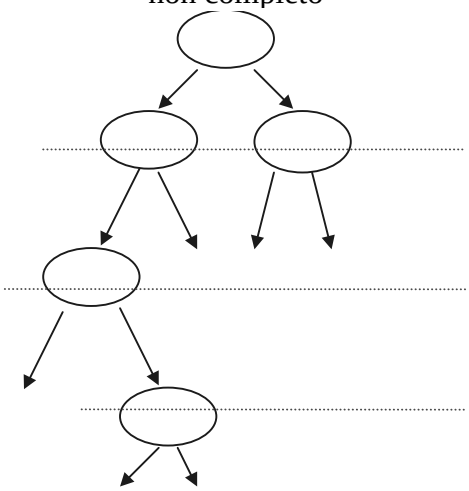
Un albero è binario se ogni nodo presenta al più due figli;
In seguito vengono elencare alcune proprietà.

Albero binario completo -- un albero binario è completo quando le sue foglie sono distribuite su gli ultimi due livelli:

Completo

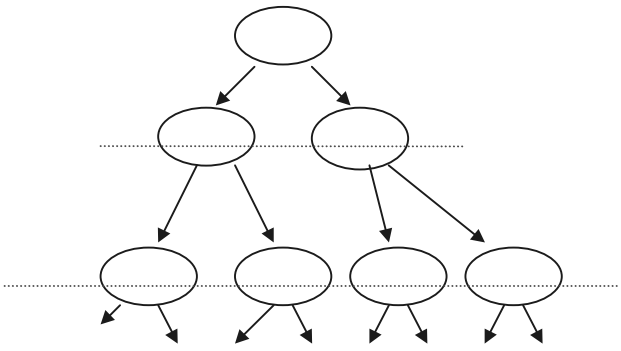


non completo



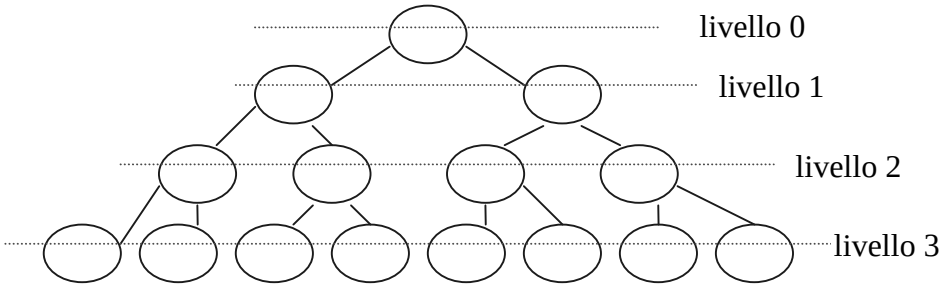
Albero binario perfetto -- Un albero binario è perfetto se presenta le foglie distribuite solo sull'ultimo livello, ultimo livello che deve essere inoltre completo.

albero binario perfetto



Un albero perfetto è bilanciato e simmetrico, inoltre ogni sottoalbero è perfetto; ovviamente vale che un albero perfetto è anche completo ma non vale il viceversa.

Nel caso di albero perfetto è possibile calcolarsi, dati i livelli, il numero totale di nodi :



Al livello 0 ho 2^0 = 1 nodo

Al livello 1 ho 2^1 = 2 nodi

Al livello 2 ho 2^2 = 4 nodi

Al livello 3 ho 2^3 = 8 nodi

In totale ho 4 livelli (0 1 2 3) ed ho quindi

$2^K - 1 = 2^4 - 1 = 15$ nodi.

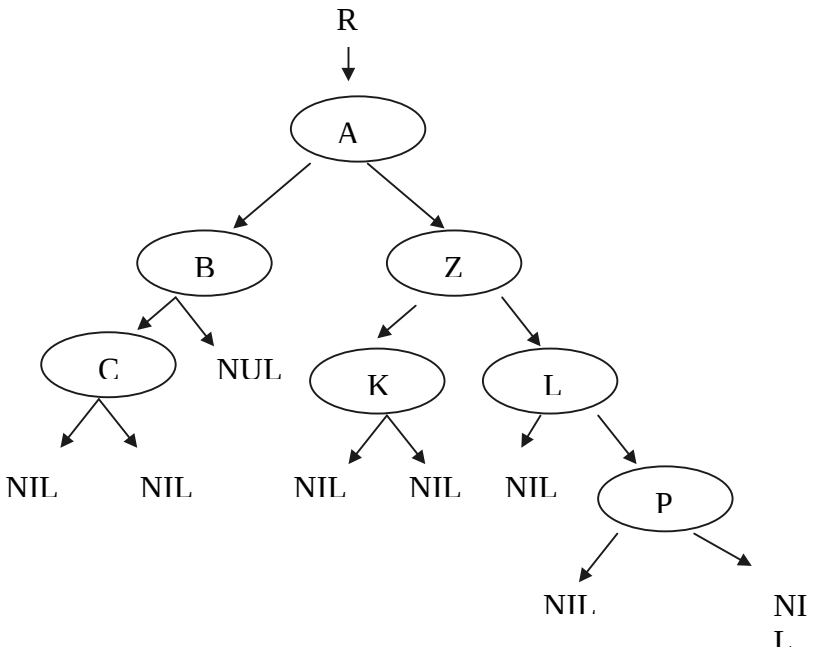
Nel caso di alberi binari ho diversi modi di fare un attraversamento :
- simmetrico
- anticipato
- posticipato

2.4 Procedura di attraversamento simmetrico

```
Procedure attsimm ( R : Pnodo );
 begin
  if R < > nil then begin
     Attsimm ( R ^ . sinistro ) ;
     writeln ( R ) ;
     Attsimm ( R ^ . destro ) ;
     end;
  end;
```

in questo tipo di attraversamento, quello cioè simmetrico, l'informazione viene stampata nel mezzo tra l'attraversamento sinistro e quello destro avendo cioè una gestione simmetrica dell'attraversamento, da qui il nome.

vediamo un esempio:



Cosa produce un attraversamento simmetrico di R ?

Cioè chiamiamo la procedura passando R , e poiché non è Nil chiamiamo l'attraversamento simmetrico del Sx di A (questa è solo una notazione per evidenziare quali sono le chiamate che effettuiamo) che è B; prima di poter stampare qualcosa devo adesso esplodere questa chiamata.
Questo però, a sua volta significa effettuare l'attraversamento simmetrico del Sx di B (cioè C) e quindi dobbiamo risolvere prima l'attraversamento simmetrico del Sx di C.
Questo significa entrare nella procedura con un nil, la procedura si chiude ed quindi per concludere l'attraversamento simmetrico del Sx di B devo fare il write, stampo cioè C.
A questo punto eseguo l'attraversamento simmetrico del Dx di C che è nil, si chiude la procedura e faccio il write di B.
In maniera del tutto analoga faccio l'attraversamento simmetrico del Dx di B (nil), chiudo il Dx di B e stampo A.
Adesso chiamo l'attraversamento simmetrico del Dx di A, questo comporta l'attraversamento simmetrico del Sx di Z che a sua volta comporta l'attraversamento simmetrico del Sx di K.

Si chiude perché K punta a nil, stampando K, adesso si effettua l'attraversamento simmetrico del Dx di K (nil), si chiude la chiamata e si effettua un write di Z ; in questo modo si prosegue fino all'attraversamento completo dell'albero.
In definitiva l'attraversamento simmetrico produce, come lettere :

C, B, A, K, Z, L, P

Questo che abbiamo visto è il risultato di un attraversamento di tipo simmetrico che, come già detto, non è l'unico; vediamo adesso l'attraversamento anticipato:

2.5 Attraversamento anticipato

Fare un attraversamento anticipato prevede che il comando di write venga eseguita prima dell'attraversamento sinistro, a parole, si stampa, si attraversa a sinistra e poi a destra.
Formalizzando :

```
if R < > nil then begin
    writeln ( R ) ;
    Attsimm ( R ^ . sinistro ) ;
    Attsimm ( R ^ . destro ) ;
end;
```

Cosa produce nel nostro esempio ?

A, B, C, Z, K, L, P

2.6 Attraversamento posticipato

```
if R < > nil then begin
   Attsimm ( R ^ . sinistro ) ;
   Attsimm ( R ^ . destro ) ;
   writeln ( R ) ;
end;
```

Cosa produce nel nostro esempio ?
In questo caso la stampa è l'ultima istruzione dopo i due attraversamenti, avremo quindi :

C, B, K, Z, P, L, Z, A

3 Algoritmi di Ricerca

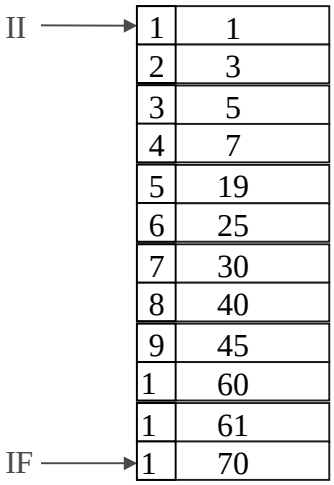
Vogliamo adesso parlare di alcuni algoritmi di ricerca; abbiamo già visto quanto costa a livello puntuale e asintotico, eseguire una ricerca di tipo *esaustivo* in strutture lineari come vettori e liste; in particolare abbiamo che la ricerca è un o (N).

Se invece la lista ed il vettore erano strutture di tipo ordinato, avevamo che il costo medio per la ricerca di un elemento era N/2, perché mediamente la posizione del record si troverà a metà della lista o del vettore. (Nota : per noi una lista o un vettore è ordinato per nome e per cognome).

Torniamo alla ricerca, riprendiamo il discorso delle liste e vettori ordinati, per loro vale che:

STRUTTURA	COSTO MEDIO	COMPLESSITA' ASINTOTICA
VETTORE ORDINATO	N / 2	o (N)
LISTA ORDINATA	N / 2	o (N)

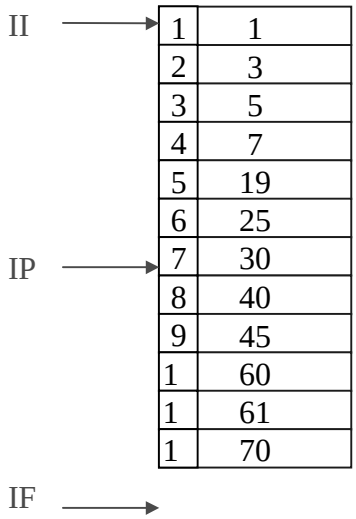
3.1 Ricerca Dicotomica su vettori



Supponiamo di avere un vettore ordinato di numeri (o stringhe) e di voler effettuare una ricerca di una chiave ricercata (CR) in maniera dicotomica.
Definisco due indici, uno iniziale II ed uno finale IF che mi indicano il campo all'interno del quale vado ad effettuare la ricerca. Supponiamo che sto ricercando l'elemento 40, nel senso che sto cercando non il numero, ma l'informazione associata alla chiave che sto cercando e cioè 40.
La prima cosa che faccio è quella di dividere a metà la spazio di ricerca definendo un indice di posizione IP.

$$IP = \left\lfloor \frac{II + IF}{2} \right\rfloor$$

Adesso ho individuato un indice che mi divide il campo di ricerca, e poiché il vettore è ordinato, posso confrontare il valore indicato da IP con quello che sto ricercando.

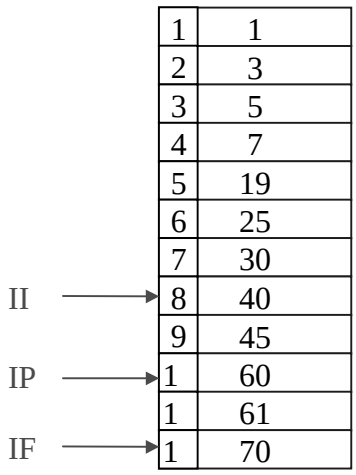


Nel nostro caso specifico abbiamo IP = (12 + 1) / 2 = 6.5 e approssimando all'intero superiore IP = 7.
Adesso passo al confronto tra il valore indicato da IP e quello che sto ricercando e riaggiorno l'indice IF o IP a seconda che la chiave ricercato sia maggiore o minore del valore indicato da IP cioè:

if V[IP] = CR then ....(ho trovato la chiave ricercata)

else if V [IP] > CR then IF : = IP - 1;

else II : = IP + 1;



Vediamo in nostro caso : IP = 7, V[7] = 30 , 30 < 40
allora ricomincio la ricerca con II = IP + 1 = 8.
Questo fa si che al primo passo sono passato da N a N/2 elementi.
Adesso ricalcolo il nuovo IP = (8 + 12) / 2 = 10 rieffettuo il confronto e così via fino, nella peggiore delle ipotesi, rimango con il solo elemento ricercato.
Con la ricerca dicotomica si opera una successione sull'insieme di ricerca del tipo N, N/2, N/4, ...1.
Se abbiamo un vettore di N elementi quante iterazioni (m) dobbiamo compiere per portare a termine il metodo di ricerca dicotomico ?

Poiché la successione appena trovata converge a $\frac{N}{2^{m-1}}$, avremo che :

$$N = 2^{m-1} \Leftrightarrow \log_2 (N) = m - 1 \Leftrightarrow \quad m = \log_2 (N) + 1$$

cioè, per ricercare l'elemento desiderato, effettuerò al massimo $m = \log_2 (N) + 1$ di iterazioni. Questo comporta un risparmio enorme rispetto al caso di ricerca esaustiva; vediamo perché:
se devo effettuare una ricerca su 1.000.000 di elementi ho

- Costo ricerca esaustiva = 1.000.000 di confronti
- Costo della ricerca dicotomica = $m = \log_2 (N) + 1$ = 20 confronti !!
-
- Il risparmio è enorme.

Si ha quindi la seguente tabella.

Ricerca dicotomica

STRUTTURA	COSTO PUNTUALE	COMPLESSITA' ASINTOTICA
VETTORE ORDINATO	$\log_2 (N) + 1$	$o (\log_2 (N))$

Per questa dipendenza dal logaritmo questo tipo di ricerca è detta *dicotomica, logaritmica, binaria o metodo di bisezione.*

3.2 Ricerca Dicotomica su Liste

Il metodo lo si può applicare anche al caso di liste ordinate, ma non è conveniente perché ha un costo enorme perché non posso calcolarmi come prima IP semplicemente come (II + IF) /2 perché la lista è una struttura sequenziale e non ad accesso diretto !
Si può fare invece su un file che contiene molti record ordinati, attraverso la funzione seek che permette un accesso diretto ai record stessi.
La lista tutto questo non lo consente; anche se avessi la fortuna di avere due puntatori, uno iniziale ed uno finale, non posso certo sommarli fra loro e dividerli per due! (questo è dovuto anche al fatto che tutte le informazioni non sono allocate nell'ordine in cui le vediamo disegnate su un foglio, ma fisicamente in memoria i record sono sicuramente in tutt'altra posizione; intervengono inoltre anche i meccanismi di garbage collection).
Per trovare il puntatore che mi indica il record di mezzo della lista, devo percorrerne metà; se N è la lunghezza della lista stessa, questa ricerca mi costa N/2, a questo punto decido se andare sopra o sotto l'operazione successiva mi costa N/4 , N/8 .....
Se a prima vista questo esempio sembra uguale a quello di prima, bisogna notare che questi sono costi mentre prima la stessa successione indicava il numero di elementi entro il quale effettuavo la ricerca. In questo caso l'operazione dominante è la ricerca della metà della lista e questo fa si che le operazioni si sommino con un costo totale di:

$$C = \frac{N}{2} + \frac{N}{4} + ... = N - 1$$

Riaggiornando la tabella si ha:

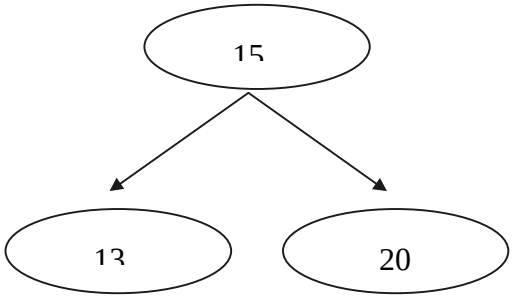
Ricerca dicotomica

STRUTTURA	COSTO PUNTUALE	COMPLESSITA' ASINTOTICA
VETTORE ORDINATO	$\log_2 (N) + 1$	$o (\log_2 (N))$
LISTA ORDINATA	N- 1	o (N)

3.3 Ricerca su Albero binario di Ricerca

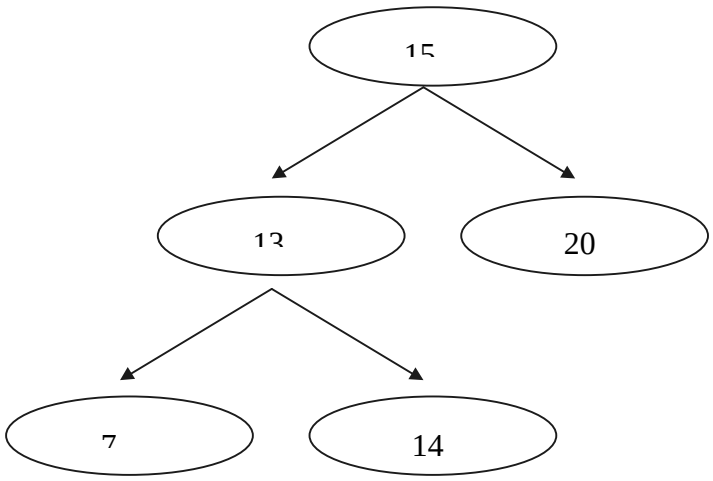
Passiamo adesso allo studio della ricerca ed inserimento in un albero.
Intanto dobbiamo definire il concetto di albero ordinato e in particolare prendiamo in esame il caso di un albero binario.

Un tale albero che contiene al suo interno una informazione ordinata viene detto *albero binario di ricerca*; come primo passo si deve 'creare' una regola per ordinare l'informazione: ad esempio si assume che sulla destra ci siano chiavi più pesanti mentre sulla sinistra ci siano chiavi più leggeri .



Questo è un possibile albero binario di ricerca in cui ho sulla destra nodi più pesanti, nel senso che hanno una chiave maggiore della radice (in generale maggiore della chiave del padre). Noi abbiamo messo dei numeri per rendere il concetto immediato, ma niente ci vietava di inserire una qualunque chiave alfanumerica.

Si supponga di dover inserire nodi con chiave: 7 , 14
Si parte con 7. Entriamo e troviamo 15, poiché 7 < 15 vado a sinistra, trovo 13,
7 < 13 e vado di nuova a sinistra, inserendo 7. Entro con 14, vado a sinistra, trovo 13 e poiché 14 > 13 vado a destra, ottengo perciò :



Adesso supponiamo di dover inserire : 95, 2, 9, 93, 53, 16.
L'organizzazione finale dell'albero dipende direttamente dalla sequenza con cui questi numeri vengono inseriti (e quindi dati).
Partiamo con il 95, entriamo e confrontiamo:

95 > 15 ➧ vado a destra

95 > 20 ➧ vado a destra e lo inserisco.

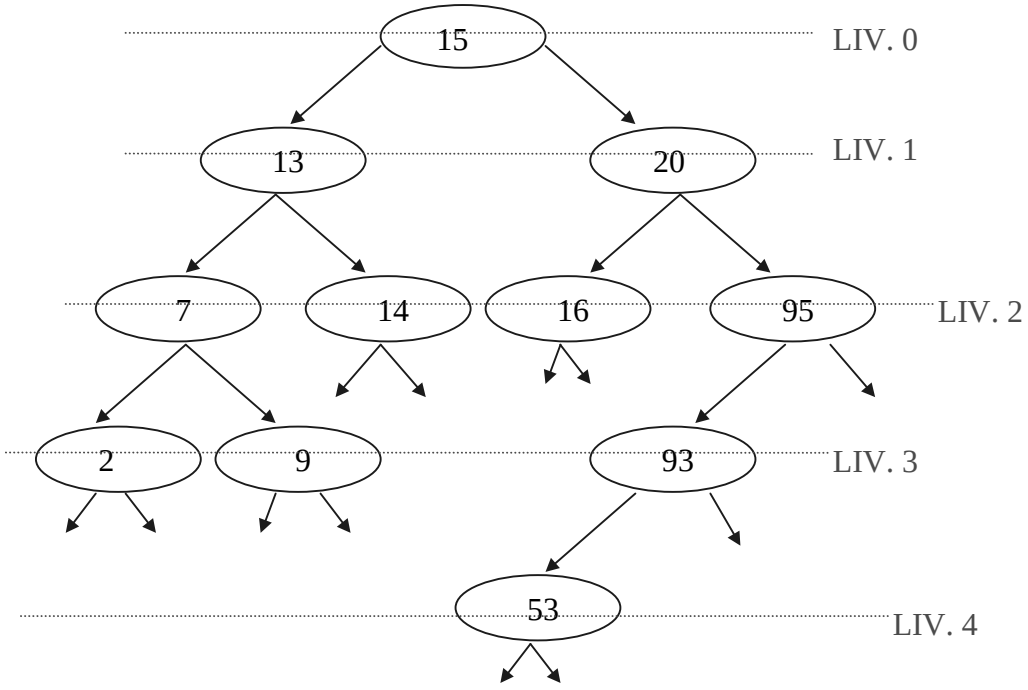
Prendo 2 :

2 < 15 ➧ vado a sinistra

2 < 13 ➧ vado a sinistra

2 < 7 ➧ vado a sinistra e inserisco.

Ripetendo per tutti i numeri questo ragionamento ottengo alla fine:



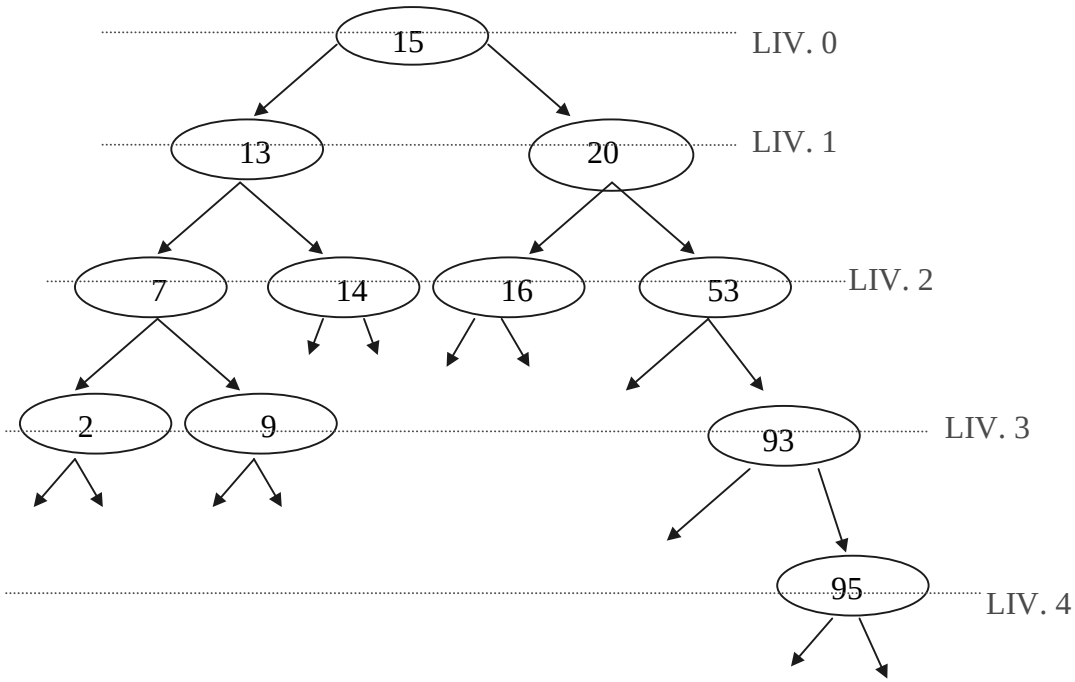
Questo albero è il risultato di questo particolare ordine di inserimento; abbiamo ottenuto:

- albero non perfetto

- albero non completo

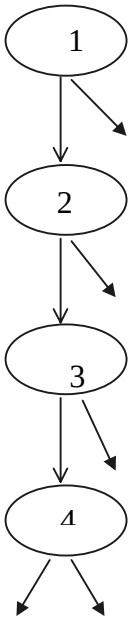
- albero con 5 livelli (0, 1, 2, 3, 4)

Adesso voglio inserire gli stessi elementi con una differente sequenza, ad esempio
2, 9, 53, 93, 95, 16, ottenendo un albero diverso, come quello rappresentato in figura:

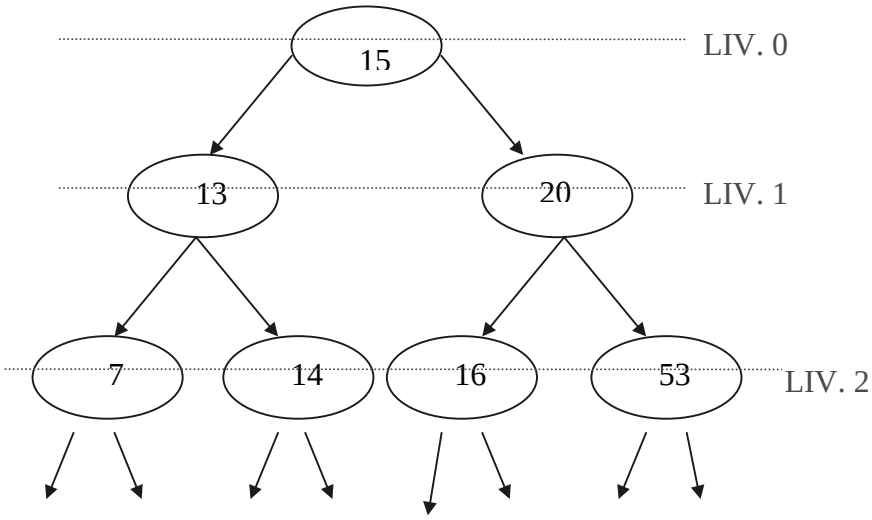


In questo caso ho la stessa altezza dell'albero mentre è cambiata la disposizione dei numeri all'interno di due nodi .
L'organizzazione dell'albero cambia quindi a seconda dell' ordine di inserimento dei dati.
Un caso limite è quello in cui devo inserire dei numeri già ordinati che porta alla formazione di un albero detto *degenere* .

Se supponiamo infatti di dover introdurre i numeri già ordinati 1, 2, 3, 4 ecc. quello che otteniamo è riportato di seguito:



Questa situazione limite porta ad un albero degenere, che altro non è che una lista !
La ricerca in questo caso è ovviamente uguale al caso della lista e cioè avrò che la complessità asintotica è un o (N).
Vediamo adesso il caso di un albero perfetto, che ricordiamo è quello che ha tutti i livelli pieni e in cui ogni nodo ha esattamente due figli.



In questo caso ho un albero binario di ricerca perfetto e il numero massimo di confronti che devo effettuare è pari al numeri di livelli cioè:
$2^3 - 1 = 7 = N$ = numero di foglie (di elementi)

e se ho 1.000.000 di dati da organizzare lo posso fare usando un albero di 20 livelli e nel caso peggiore, per la ricerca di un elemento, dovrò effettuare 20 confronti.

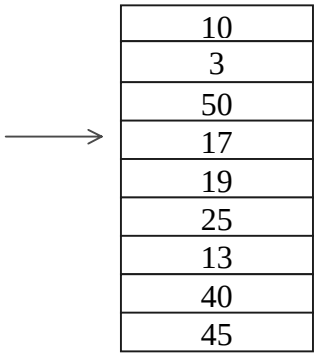
Quindi anche in questo caso la complessità asintotica è o ($\log_2 N$).

Riepilogando:

Struttura dati	Complessità asintotica
Vettore ordinato	o (N)
Lista ordinata	o (N)
Albero binario di ricerca	o ($\log_2$ (N))
Albero perfetto non ordinato (non di ricerca)	o (N)

Vediamo adesso l'operazione di cancellazione nelle varie strutture con l'ipotesi che abbiamo già individuato l'informazione da cancellare.

Vettore disordinato



Se voglio cancellare l'elemento indicato, non è un grosso problema perché prendo l'ultimo elemento del vettore e lo copio al posto dell'elemento che ho cancellato; tutto questo è possibile perché tanto il vettore è disordinato.
La cancellazione in sé stessa ha costo 1, mentre la ricerca dell'elemento sappiamo essere un o (N).

Vettore ordinato

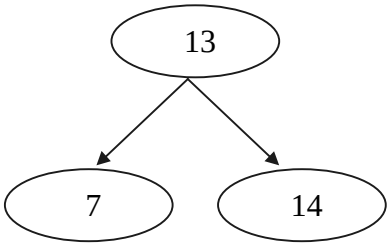
In questo caso invece, dopo la cancellazione devo riorganizzare per mantenere la struttura ordinata, questa riorganizzazione ha un costo minimo di 1 (cancello proprio l'ultimo elemento), costo medio N / 2, comunque una complessità asintotica o (N).

Quindi la cancellazione in questo caso mi costa N.

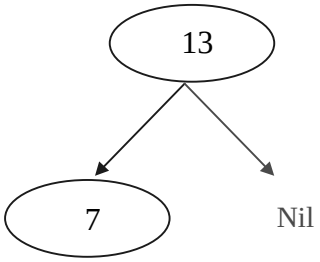
Lista ordinata

La cancellazione mi costa 1 perché non ho bisogno di riorganizzazione, mentre la ricerca, al solito, costa N.

Albero
Il caso dell'albero è più complicato e presenta molte situazioni ; vediamo perché:



Se voglio cancellare 14 metto il suo puntatore a nil e ne faccio successivamente un dispose; quello che ottengo è indicato in figura:



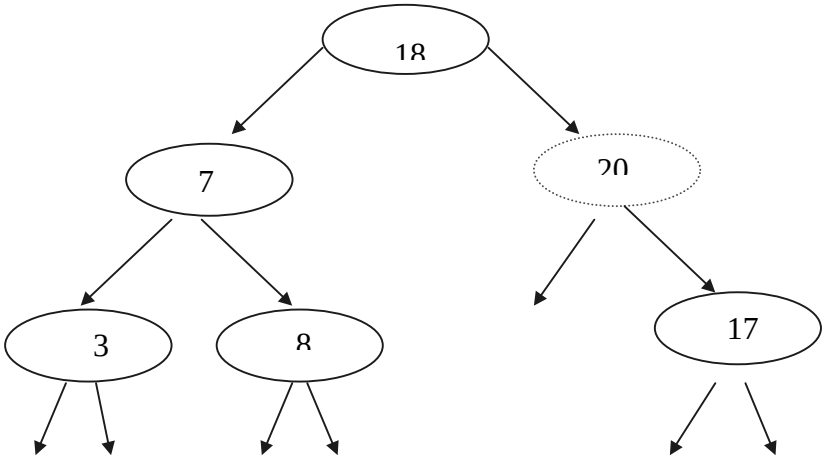
Il caso invece più complicato è invece quello della cancellazione della radice, o in generale, la cancellazione di un nodo che abbia tutti e due i figli connessi.
Il problema nasce dal fatto che cancellando un nodo intermedio doppiamente connesso ho il problema successivo di dover poi riattaccare i rami disconnessi.
Un caso intermedio è quello invece della cancellazione di un nodo che ha un figlio vuoto, che punta cioè a nil.

I casi possibili sono cioè:

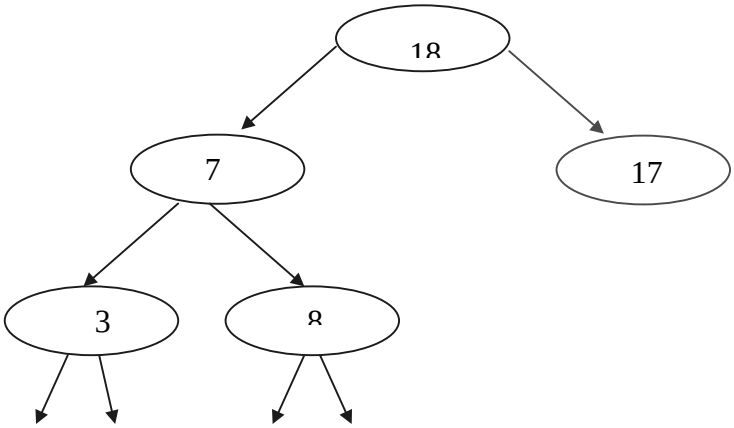
1. Cancello nodo con Dx, Sx = nil
2. Cancello nodo con Dx = Nil , Sx ≠ Nil
3. Cancello nodo con Dx ≠ Nil , Sx = Nil
4. Cancello nodo con Dx e Sx ≠ Nil

Vediamo il caso 3

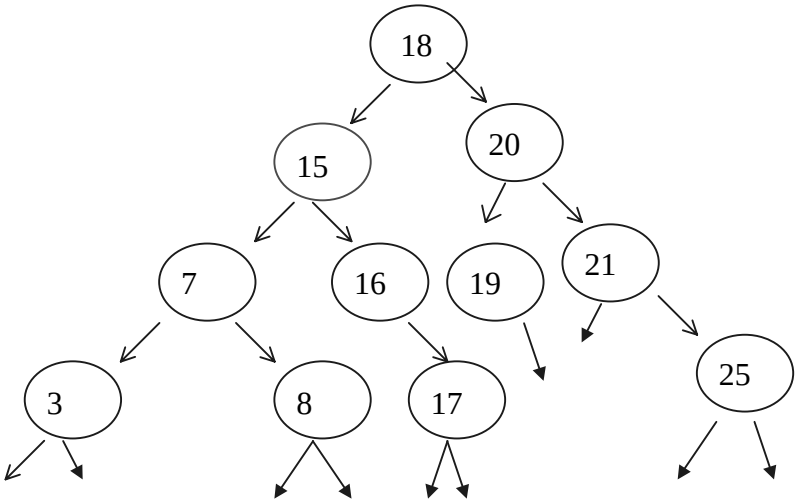
Cancellazione di un nodo con Dx ≠ Nil.



Questo caso è abbastanza semplice in quanto, per la cancellazione del nodo 20, è sufficiente rimuoverlo e attaccare il puntatore di 18 all'elemento 17.
Quello che otteniamo è riportato in figura seguente:

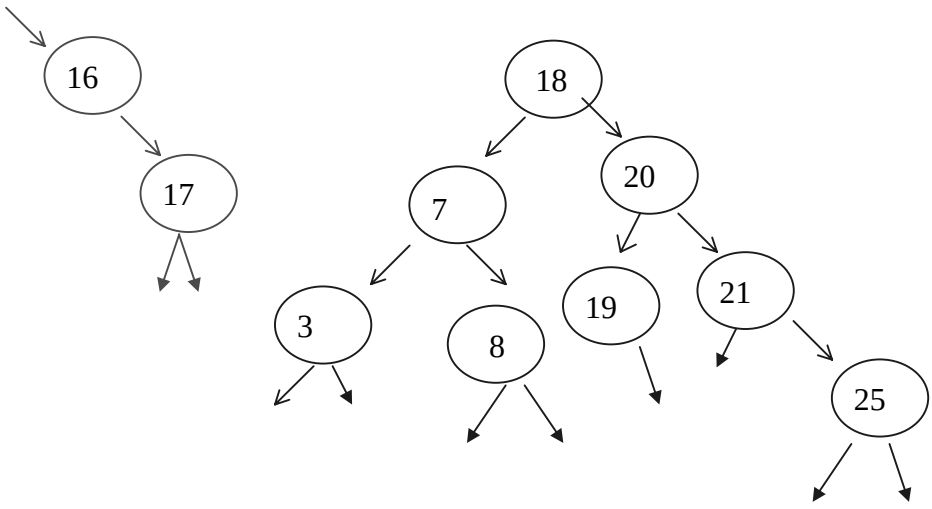


Il caso più complesso è la cancellazione di un nodo che ha entrambi i figli attivi.



Supponiamo adesso di voler cancellare 15.
Come prima cosa devo scegliere una direzione privilegiata per lo sviluppo dell'albero; supponiamo di aver scelto la sinistra; ho eliminato 15 (nodo rosso), quello che mi rimane è un troncone che può avere un numero qualunque di figli, nel nostro caso solo 2: 16 e 17.
Adesso appoggio ciò che rimane ' isolato ' dalla cancellazione di 15 (cioè i nodi 16 e 17) in un vettore temporaneo.

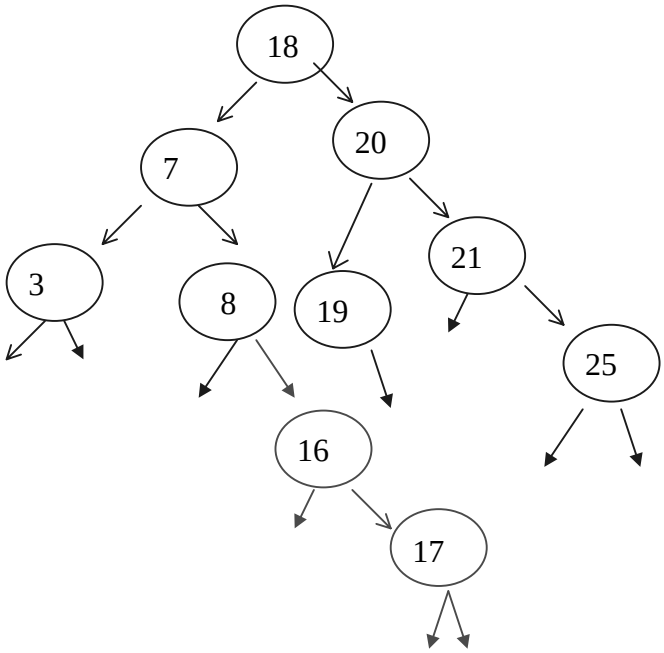
TMP



L'operazione che facciamo adesso è quella di reinserire il primo nodo del sottoalbero troncato, cioè 16, a partire dalla radice.

Parto cioè con 16 e, seguendo le usuali regole dell'inserimento, ottengo:

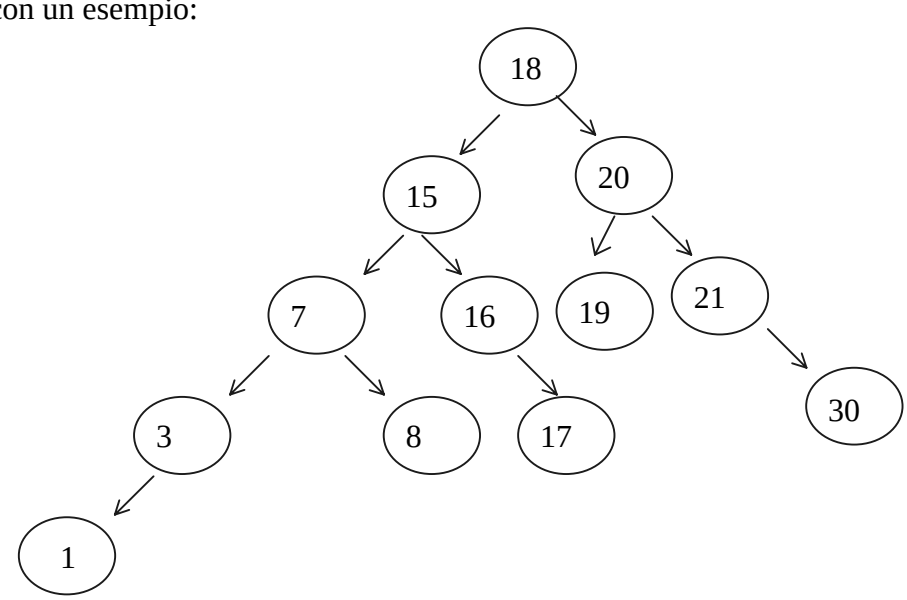
16 < 18 (radice) ➧ vado a Sx, 16 > 7 ➧ vado a Dx,16 > 8 ➧ vado a Dx, adesso posso inserire 16.



Il risultato finale è che 16 è passato dal livello 2 al livello 4.
Quale è il costo dunque di una cancellazione fatta così ?
E' quello equivalente al reinserimento dell'elemento, che è pari a quello della ricerca della sua posizione che è dunque o ($\log_2 N$) .
Riassumiamo quanto detto fino ad ora:

	RICERCA	**CANCELLAZIONE**
VETTORE NON ORDINATO	o (N)	o (1)
VETTORE ORDINATO	o (N)	o (N)
LISTA NON ORDINATA	o (N)	1
LISTA ORDINATA	o (N)	1
ALBERO BINARIO DI RICERCA	o ($\log_2 N$)	o ($\log_2 N$)
ALBERO BINARIO	o (N)	o ($\log_2 N$)
VETTORE CON RICERCA DICOTOMICA	o ($\log_2 N$)	------

Concludiamo con un esempio:



Cosa stampo in seguito ad un attraversamento simmetrico ?

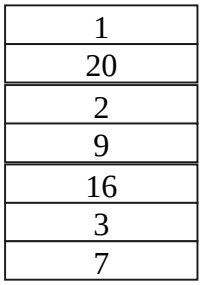
1- 3 - 7 - 8 - 15 - 16 - 17 - 18 - 19 - 20 - 21 - 30

Quindi nel caso di un albero binario di ricerca l'attraversamento simmetrico produce una lista ordinata per chiave di record.

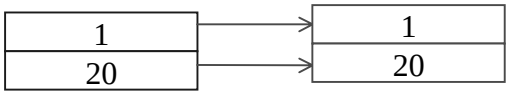
4 Metodi di ordinamento

Dato un insieme di record posso ordinarli in base ad una chiave univoca di tipo alfanumerico; questo tipo di ordinamento può essere crescente (inizio con chiavi più leggere) o decrescente (inizio con le chiavi più pesanti).Questo lo abbiamo usato per vettori, liste ed alberi binari.

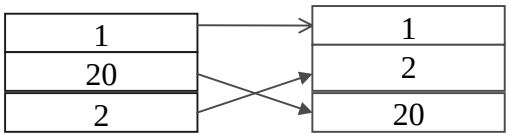
4.1 Ordinamento per inserzione



Questo metodo presuppone l'utilizzo di un vettore aggiuntivo nel quale colloco in maniera ordinata gli elementi del vettore di partenza; parto con 1, lo posiziono nel vettore ausiliario, prendo 20 che è > 1 e lo inserisco quindi dopo.

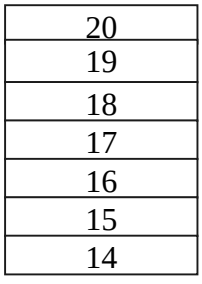


Adesso prendo il 2, faccio posto tra 1 e 20 e lo inserisco



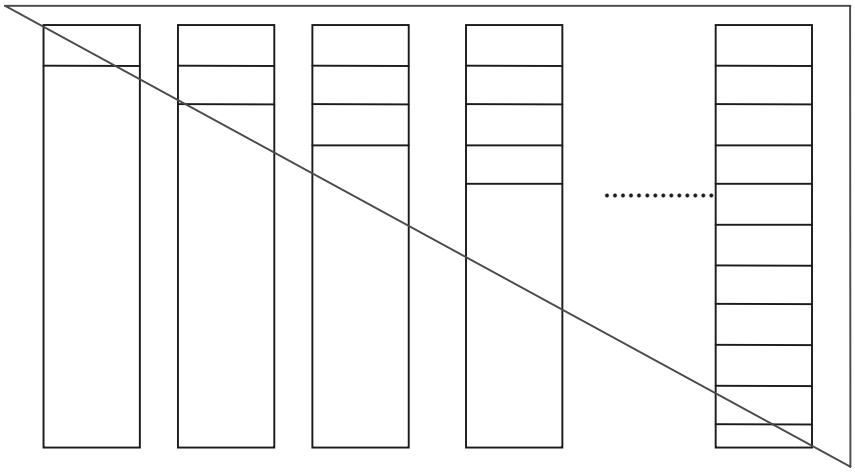
Quindi le operazioni che compio sono: ricerca, riorganizzazione e inserzione.

Il caso peggiore per questo metodo è quello in cui il vettore è ordinato in senso opposto a come voglio ordinarlo:



Infatti, parto con 20 e lo copio nel vettore ausiliario, al secondo passo ho 19 che è minore di 20, devo allora spostare 20 ed inserire 19.
Prendo 18 < 19 < 20, prima di inserirlo devo spostare le altre chiavi già inserite cioè 20 e 19.....
Quali sono allora le operazioni che devo compiere ?

L'operazione dominante in questo caso è la riorganizzazione e quindi di spostamento elementare.
Al primo passo ho dovuto fare uno spostamento, al secondo passo due, al terzo 3 ...fino a N - 1 spostamenti.
Questa successione converge a $\frac{N \cdot (N - 1)}{2}$ perché il numero di inserimenti che compio è pari al numero degli elementi indicati in figura:



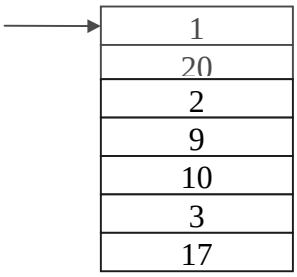
Riferendoci alla ‘ matrice ’ di sopra, il metodo appena visto equivale allo spostamento degli elementi oltre la diagonale, che sono appunto $\frac{N \cdot (N - 1)}{2}$.
La complessità asintotica in questo caso è o (N^2), e un altro problema è che ho bisogno di un vettore d’appoggio.

Ordinamento per inserzione in una lista
Sappiamo che una lista non ha bisogno di riorganizzazione e il costo principale diventa quello relativo ricerca poiché l’inserimento ha costo 1.

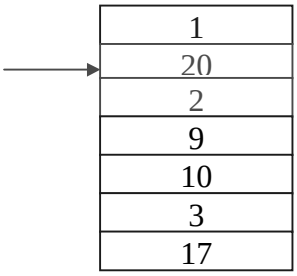
E’ interessante notare come avere tutti i record ordinati all’incontrario rispetto al senso di inserimento, sia, all’incontrario del vettore, *il caso migliore* perché faccio inserimenti in testa (che sono i più convenienti) con costo 1, l’inserimento ha costo 1, quindi in questo caso ho una complessità asintotica o (N).
Nel caso peggiore invece in cui non è vero che faccio sempre inserimenti in testa, ma devo fare un operazione di ricerca per andare a cercare la posizione utile per l’inserimento, la complessità segue una successione 1,2,3,4, .. , N - 1 (questo perché al primo passo devo ricercare su una lista di un elemento, poi in una con 2 elementi ecc., fino ad N - 1 perché l’ N - esimo elemento lo sto inserendo in quel momento).
Questa successione converge a $\frac{N \cdot (N - 1)}{2}$ per cui, nel caso peggiore della lista, l’ordinamento per inserzione è o (N^2).

4.2 Inserzione diretta

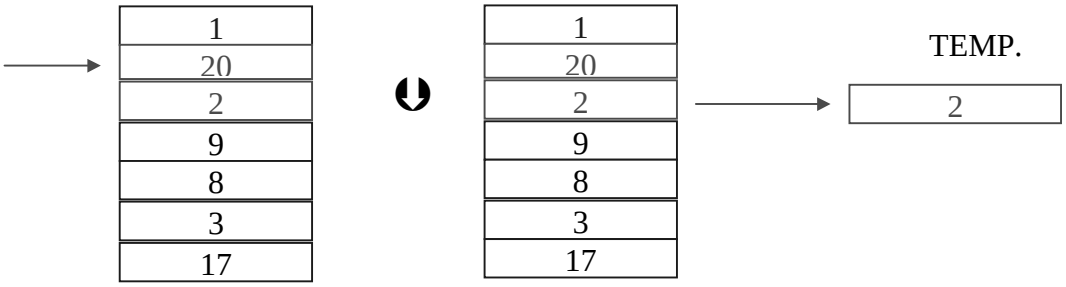
Questo algoritmo deriva dall'algoritmo appena visto ma non utilizza però il vettore di appoggio.

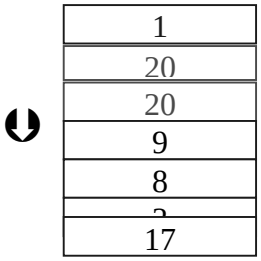


Ho un indice che mi indica il primo elemento (1), faccio un confronto con l'elemento indicato ed il successivo (20), poiché sono ordinati sposto di una posizione l'indice .

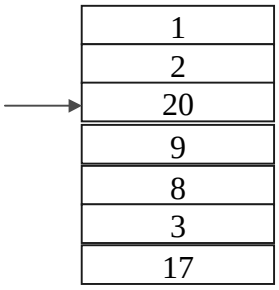


Adesso effettuo il confronto fra 20 e 2; qui dobbiamo riorganizzare. Allora si agisce così : salvo 2 su una variabile temporanea, ricerco la posizione di inserimento della chiave (dopo 1 e prima di 20), si fa una riorganizzazione, spostando l'elemento indicato dall'indice verso il basso, si ottiene:

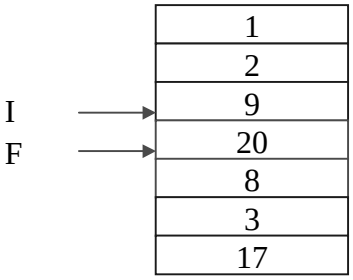




A questo punto si puo' effettuare l'inserimento vero e proprio che consiste nel copiare l'elemento salvato (2) nella posizione indicata dall'indice vecchio, otteniamo così :

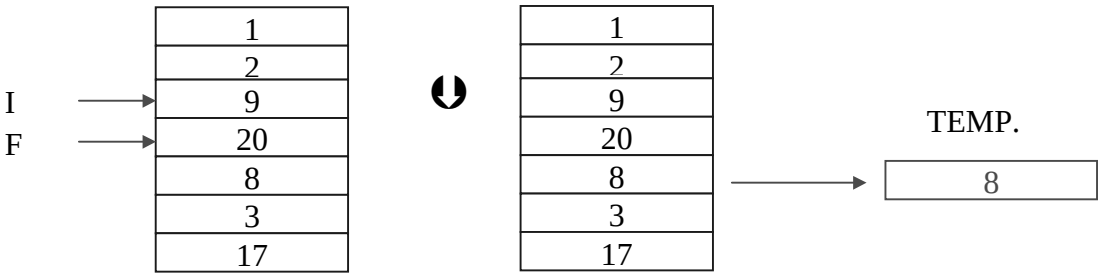


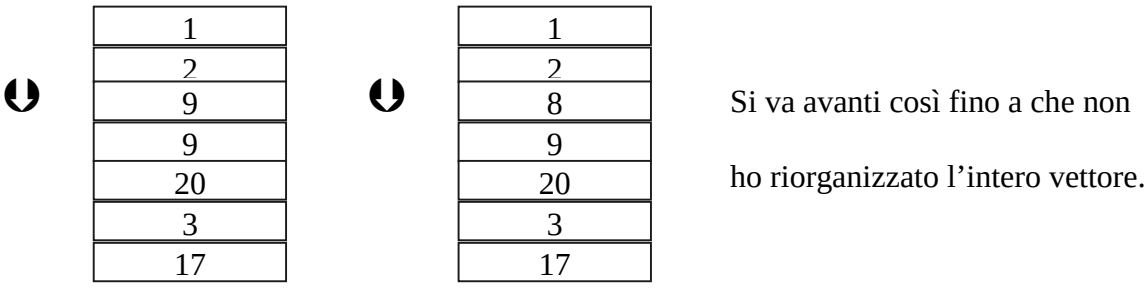
Adesso possiamo andare avanti applicando sempre lo stesso meccanismo : confrontiamo 20 con 9; dobbiamo riorganizzare; salviamo 9, scorriamo 20 verso il basso, e copiamo 9 dove punta l'indice.....



Siamo in questa situazione, confronto 20 con 8 e vedo che devo riorganizzare, ricerco la posizione di inserimento di 8 (tra 2 e 9), salvo 8, riorganizzo spostando gli elementi da I a

F verso il basso di un posto, ottenendo:





Si va avanti così fino a che non

ho riorganizzato l'intero vettore.

Quali sono le operazioni che eseguo ed il loro costo ?

- Confronto costo 1
- Ricerca dipende da quanti elementi ho
- Riorganizzazione dipende da quanti elementi ho e da dove devo inserirli
- Inserimenti costo 1

Quale è il caso migliore ?
Il caso migliore è ovviamente quello in cui il vettore è già ordinato, in questo caso la complessità o (N) perché non devo né inserire né riorganizzare e né ricercare la posizione utile all'inserimento, faccio solo N confronti.

Quale è il caso peggiore ?
Il caso peggiore è quello in cui ho un vettore ordinato in senso contrario a quello che voglio:

20
19
18
17
16
15
14

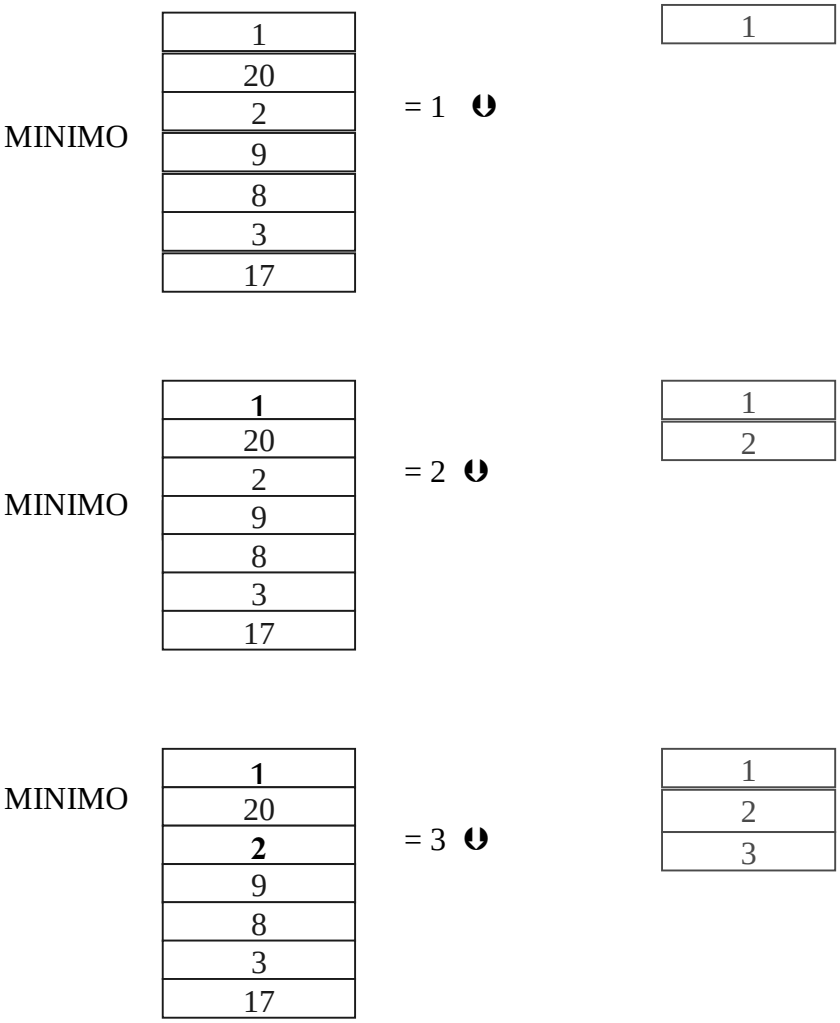
Se ho infatti questo vettore, ordinato in senso decrescente e lo voglio, all'incontrario, in ordine crescente, devo compiere un certo numero di riorganizzazione che coinvolgono un numero sempre maggiore di elementi: 1,2,3,4, ... , N-1.

Nel caso peggiore quindi, non ho costi di ricerca ma solo di riorganizzazione con una successione convergente a $\frac{N \cdot (N - 1)}{2}$ che comporta quindi, come in alcuni casi precedenti, ad una complessità asintotica che è un o (N^2).
Nel caso della lista non abbiamo mai riorganizzazione e abbiamo solo costi di ricerca che viene effettuata su un numero via via crescente di elementi (1,2,3,4....., N-1); questo fa si che i costi di ricerca siano $\frac{N \cdot (N - 1)}{2}$ per una complessità asintotica che è un o (N^2); infatti, nel caso peggiore, l'elemento da inserire è sempre minore di quelli ordinati (cioè costa 1 la ricerca per l'inserimento).

4.3 Metodo di ordinamento per selezione

1
20
2
9
8
3
17

E' un metodo che presuppone l'uso di un vettore di appoggio ausiliario, lo si realizza andando a selezionare direttamente dal vettore non ordinato, l'elemento minimo e copiandolo nel vettore di appoggio:



..... alla fine ottengo un vettore ordinato.

Le operazioni che compio sono la ricerca del minimo e l'inserimento.

Per quanto riguarda i costi ho che l'inserimento ha costo = 1, mentre invece per la ricerca del minimo, la effettuo sempre su un vettore disordinato di N elementi, per cui non posso usare la dicotomica, ma una ricerca esaustiva per cui mi costa o (N);
ma questa operazione di ricerca la devo eseguire N volte; cioè devo fare N ricerche per N volte, questo mi porta ad un costo asintotico = o (N^2).

4.4 Metodo di ordinamento per selezione diretta

E' un metodo che non fa uso del vettore di appoggio, e questo può consentire un notevole risparmio di memoria soprattutto quando si lavora un elevato numero di elementi.
Questo risparmio è importante poiché la valutazione della ' bontà ' di un algoritmo la si effettua in base allo spreco di memoria e come complessità, intesa come velocità di esecuzione.
L'idea del metodo è quello di identificare il minimo e la sua posizione giusta, senza dover ricorrere ad altri vettori ausiliari.

1
20
2
9
8
3
17

Partiamo dal solito vettore di prima, ne faccio la ricerca del minimo che, in questo caso, porta all'individuazione di 1.
Divido il vettore in due parti di cui una già ordinata (che per ora contiene il solo elemento 1). Ottengo cioè :

1

20
2
9
8
3
17

La parte superiore è già ordinata, quindi la ricerca successiva del minimo la effettuo soltanto sulla parte non ordinata.
Trovo 2 , adesso compio uno scambio con il primo elemento del sotto - vettore non ordinato (20). Ottengo perciò :

1
2

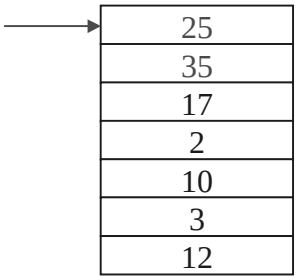
20
9
8
3
17

Ho così un a parte del vettore ordinata e un sotto - vettore disordinato.
Proseguo la ricerca del minimo, lavorando però su sottovettori con un numero decrescente di elementi. Nel nostro caso trovo 3, 8, 9,17, 20. A questo punto ho ottenuto un vettore ordinato.

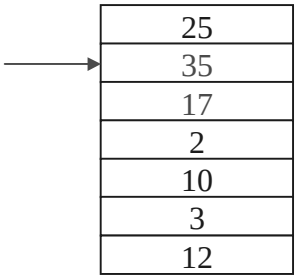
Le operazioni che compio in questo algoritmo sono ricerca ed inserimento.
L'inserimento si riduce ad uno scambio, con costo 1, mentre la ricerca invece la effettuo su N, N - 1, N - 2 , ... , 2 elementi. Questa non è altro che la successione precedente (vista alla rovescia) che converge a $\frac{N \cdot (N - 1)}{2}$. Questo ci dice che la selezione diretta costa asintoticamente o (N^2) .

4.5 Ordinamento per scambio : Bubble Sort

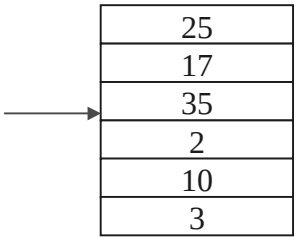
Supponiamo di avere il seguente vettore:



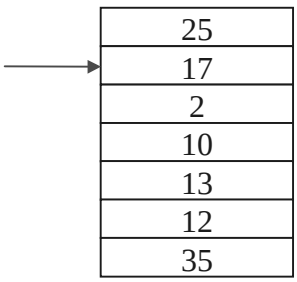
Questo tipo di algoritmo si basa su un confronto tra due elementi ed un loro eventuale scambio. Supponiamo di voler ordinare in senso crescente.
Definiamo un indice che mi punta ad un elemento, e si verifica localmente la condizione di ordinamento. Confrontiamo inizialmente il 25 con il 35, ed essendo ordinati non compio su di loro nessuna azione, se non quello di incrementare l'indice.



Adesso confronto 35 e 17.
In questo caso non è rispettato l'ordine crescente e devo perciò riorganizzarli; questo lo posso fare semplicemente scambiandoli. Questo mi porta alla seguente configurazione:



Adesso devo confrontare 35 con 2; sono in ordine sbagliato, inverto e vado avanti così, fino a che, nel nostro esempio non si ha 35 sul fondo del vettore:



E' accaduto che la chiave più pesante è stata ' sistemata'; infatti con l'algoritmo del bubble sort ' trascino' sul fondo le chiavi più pesanti e questo fa si che alla passata succesiva non mi preoccupi più di loro. Alla prima passata, ho speso N - 1 confronti (e un certo numero di scambi che però non è dominante) e mi ha permesso di mettere a posto un elemento. Al giro successivo lavoro però su tutti gli elementi tranne l'ultimo, che so già sistemato, questo vuol dire che lavoro su un vettore

che mi consente di effettuare N - 2 confronti, successivamente N - 3 ecc. , secondo una successione che converge, al solito a $\frac{N \cdot (N - 1)}{2}$, abbimo allora una complessità asintotica che è un o (N^2) .
Se volevamo, al contrario, un ordinamento per chiavi decrescenti, saranno le chiavi più leggere ad essere spinte verso il basso.

Vediamo adesso alcuni esempi:

23
1
5
21
17
4
2
18

Cosa producano 3 iterazioni del metodo di inserzione diretta ?

1
23
5
21
17
4
2
18

1° iterazione

1
5
23
21
17
4
2
18

2 ° iterazione

1
5
21
23
17
4
2
18

3 ° iterazione

Quale è il risultato invece di una iterazione completa ?

1
5
21
17
4
2
18
23

2 ° iterazione

4.6 Quick sort

Si basa sull'ipotesi iniziale di avere un insieme da ordinare disordinato.
Vediamo un esempio.
Prendiamo un insieme di numeri:

35, 2, 10, 46, 6, 40, 45, 37, 5
↑ ↑
i j

Si abbiano 2 indici i e j.
Lo scopo dell'operazione che facciamo ora è quello di trovare la posizione per il *pernio*: esso è uno dei numeri sopra che divide l'insieme non ordinato in 2 parti, dove per es. a sinistra del pernio ci saranno tutti gli elementi aventi peso minore e a destra tutti gli elementi aventi peso maggiore.
Per identificare la posizione del pernio, la prima ipotesi che viene fatta è che il primo valore a sinistra, nel nostro caso 35, sia un buon pernio.
Ora non è detto che tale elemento sia un equo divisore in due dell'insieme: nel nostro caso lo è dato che ho 4 numeri > 35 e 4 numeri < 35.
Quindi dati due indici, di cui uno (che è i) identifica il pernio, vediamo come si fa a trovare nel minore tempo possibile la posizione per il pernio in modo tale che tutte le chiavi alla sua sinistra siano inferiori e tutte le chiavi alla sua destra siano superiori.
Si fa un confronto tra il pernio e l'altro indice j:
se v [i] > v [j] allora scambio il pernio con l'indice j e ottengo un vettore del tipo seguente, dove oltre a fare l'operazione di scambio eseguiamo uno scorrimento:

5, 2, 10, 46, 6, 40, 45, 37, 35
↑ ↑
j i
(è sempre il pernio)

Faccio un confronto: in questo caso l'ordinamento è corretto perché a sinistra del pernio ho un valore inferiore quindi va bene.

Poi scorriamo:

5, 2, 10, 46, 6, 40, 45, 37, 35
↑ ↑
j i

e facciamo il confronto: ho ancora che 35 > 10 ⇒ non facciamo alcuna operazione di scambio

Di nuovo scorriamo: 5, 2, 10, 46, 6, 40, 45, 37, 35

j i

e facciamo il confronto tra 35 e 46: da questo vediamo che 35 < 46 ⇒ scambiamo e

otteniamo: 5, 2, 10, 35, 6, 40, 45, 37, 46

i j

(pernio)

Quindi scorriamo sempre con j: 5, 2, 10, 35, 6, 40, 45, 37, 46

i j

(pernio)

[NOTA: si può osservare che quando j si trova alla destra del pernio allora j decrementa, mentre quando si trova alla sinistra del pernio allora j incrementa]

Facciamo il confronto tra 35 e 37 ⇒ O.K.

Ancora scorro: 5, 2, 10, 35, 6, 40, 45, 37, 46

i j

(pernio)

e confronto ⇒ O.K.

Ancora scorro: 5, 2, 10, 35, 6, 40, 45, 37, 46

i j

(pernio)

e confronto ⇒ O.K.

Ancora scorro: 5, 2, 10, 35, 6, 40, 45, 37, 46

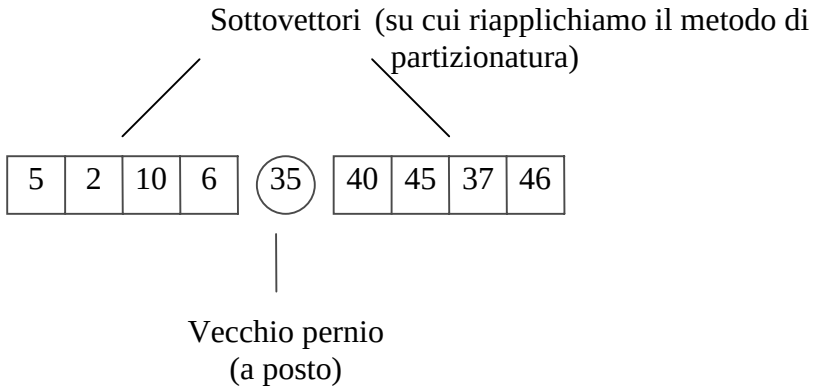
i j

(pernio)

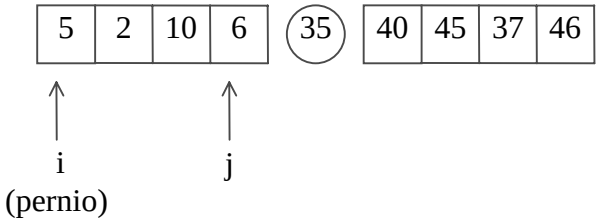
e confronto 35 e 6 ⇒ li scambiamo e otteniamo

5, 2, 10, 6, 35, 40, 45, 37, 46 (1° riduzione)

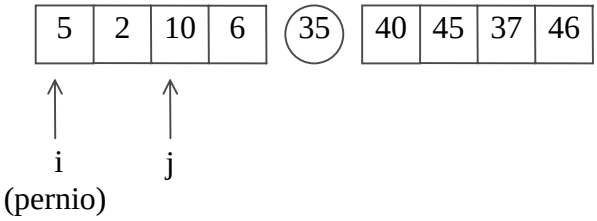
Come era ipotizzabile fin dall'inizio, il pernio ha 4 chiavi inferiori alla sua sinistra e 4 chiavi superiori alla sua destra, dividendo così l'insieme disordinato in 2 insiemi ancora disordinati (in cui il pernio è già nella posizione giusta).
Adesso applichiamo il metodo di partizionatura, visto finora, a ciascuno dei 2 sottoinsiemi:



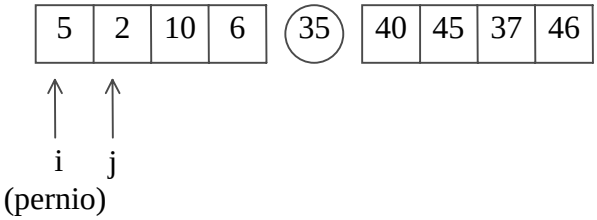
Vediamo il metodo di partizionatura per il sottovettore a sinistra del vecchio pernio:



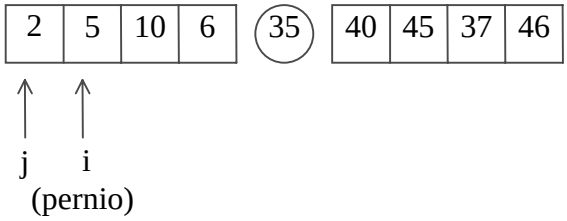
Confronto 5 e 6 ⇒ O.K. ; quindi scorro ⇒



Confronto 5 e 10 ⇒ O.K. ; quindi scorro ⇒

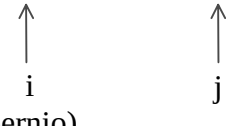


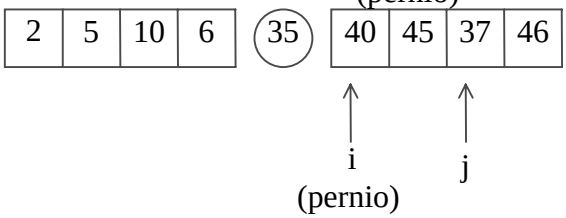
Confronto 5 e 2 ⇒ non va bene ⇒ scambio ⇒



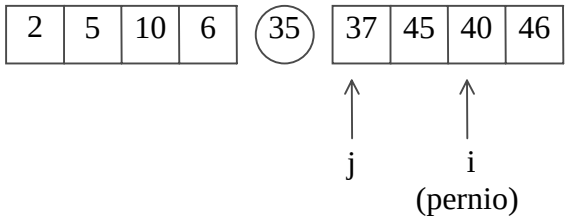
Adesso vediamo il metodo di partizionatura per il sottovettore a destra del vecchio pernio:

Confronto 40 e 46 ⇒ O.K. ; quindi scorro:

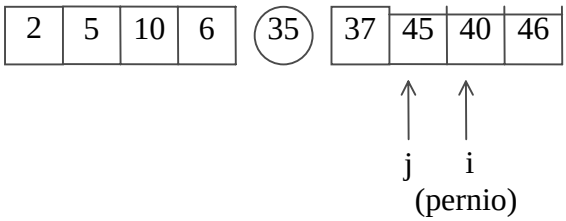




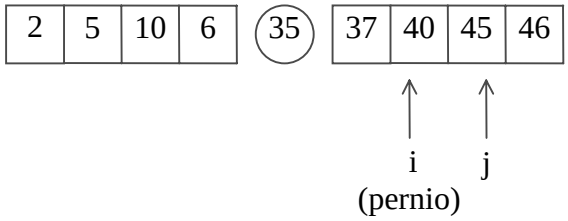
Poi confronto 40 e 37 ⇒ non va bene ⇒ scambio ⇒



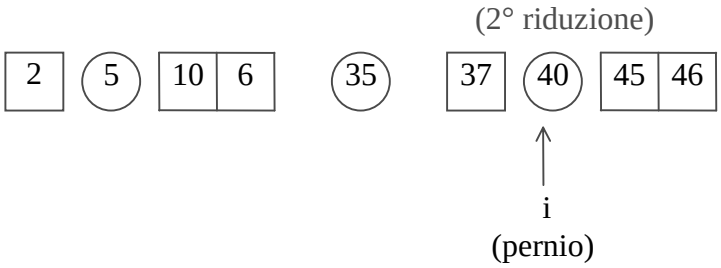
Ora scorriamo:



Poi confronto 45 e 40 ⇒ non va bene ⇒ scambio ⇒



Quindi siamo giunti alla situazione seguente (i valori nei cerchi sono perni):



A questo punto eseguiamo ancora la partizionatura sui sottovettori ottenuti:

- il 1° sottovettore è composto soltanto da 2 ⇒ siamo già a posto;
- il 2° sottovettore è formato da 10 e 6: li confronto ⇒ non va bene ⇒ scambio

6	10

- il 3° sottovettore è composto soltanto da 37 ⇒ siamo già a posto;
- il 4° sottovettore è formato da 45 e 46: li confronto ⇒ O.K.

Quindi questo è il metodo quick sort, che si applica prima a tutto il vettore e poi si riapplica in modo ricorsivo ai suoi sottovettori.

Questo metodo funziona male se il vettore è già ordinato (o anche se è ordinato in senso opposto): infatti in tal caso il 1° elemento non viene certo a trovarsi alla metà circa dell'insieme e così ci troviamo il vettore di partenza diviso in 2 sottovettori di cui uno è composto da un solo elemento e l'altro da N - 1 elementi ⇒ così non si è ottenuto alcun guadagno.

Invece il caso migliore è quando gli elementi che iniziano il vettore principale e i sottovettori sono dei perni ottimi (cioè perni che dividono l'insieme considerato in due parti identiche) ⇒ così lavoriamo con un metodo che permette di operare sulla metà degli elementi, e abbiamo una progressione simile a quella dell'albero.

Il n° di riduzioni o decomposizioni che si fanno è pari all'altezza dell'albero.
Nel nostro esempio abbiamo fatto 3 decomposizioni e così abbiamo messo a posto 2^3+1 elementi.
Questo significa che:

- al 1° passo della partizionatura mettiamo a posto 1 elemento;
- al 2° passo della partizionatura mettiamo a posto 3 elementi;

- vediamo con l'esempio seguente quanti elementi mettiamo a posto al 3° passo della partizionatura:

insieme generale (15 elementi) X X X X X X X X X X X X X X X

1° passo: metto a posto 2^1-1=1 elemento [X X X X X X X] (X) [X X X X X X X]

2° passo: metto a posto 2^2-1=3 elementi [X X X] (X) [X X X] (X) [X X X] (X) [X X X]

3° passo: metto a posto 2^3-1=7 elementi X(X)X (X) X(X)X (X) X(X)X (X) X(X)X

In questo caso abbiamo finito perché siamo arrivati ad elementi semplici, ma se avessimo avuto un insieme più grande, sarebbe accaduto che al

4° passo: metto a posto 2^4-1=15 elementi
Quindi in generale:

n° di elementi messi a posto = 2^k - 1 (progressione logaritmica)

dove k = n° di passi.
Da questa si ricava:
$k = \log_2 n$ ⇒ o ($\log_2 n$) è la complessità che mi dice il n° di passi che dobbiamo fare, dove ogni passo è un passo di partizionatura.

Perciò, nel caso in cui si abbiano perni ottimi, devo fare $k = \log_2 n$ partizionature.
Nel caso in cui non si abbiano perni ottimi, il ragionamento decade: infatti nel caso peggiore, quello in cui l'insieme è già ordinato, non riusciamo ad arrivare a dividere l'insieme.

Vediamo che cosa succede proprio nel caso di vettore già ordinato.
Si abbia, per es., il vettore: 1 2 3 4 5 6 7

Non essendoci mai da scambiare, non si fa altro che scorrere per arrivare a vedere che la posizione del pernio su 1 era già corretta:

(1) [2 3 4 5 6 7] (fine 1° passo)

Al passo successivo si opera sul sottovettore 2 3 4 5 6 7 e accade esattamente come sopra:

(1) (2) [3 4 5 6 7] (fine 2° passo)

Si procede sempre allo stesso modo per i passi successivi:

(1) (2) (3) [4 5 6 7] (fine 3° passo)

(1) (2) (3) (4) [5 6 7] (fine 4° passo)

Si osserva quindi che, mentre nel caso migliore dovevamo fare $\log_2 n$ passi di partizionatura, nel caso peggiore ne dobbiamo fare n ⇒ abbiamo un o (n) come passi di partizionatura.
Ma la complessità non è data solo da questo, bensì anche da un altro fattore che è il costo del singolo passo di partizionatura: nella partizionatura costano i confronti che sono (nel caso peggiore):

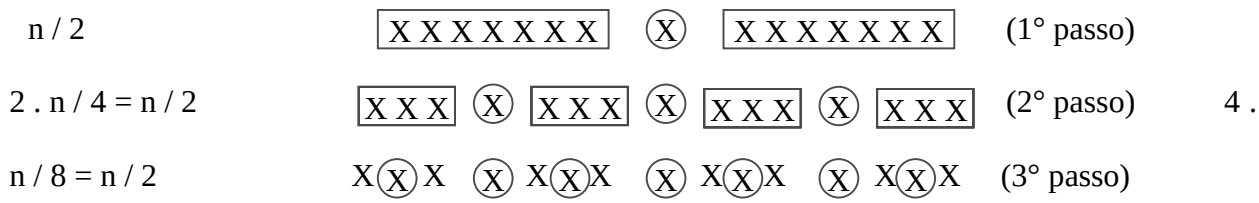
n - 1 al 1° giro

n - 2 al 2° giro

n - 3 al 3° giro

.
.
.

Quindi questa progressione costa n-1 al 1° giro, n-2 al 2° giro, n-3 al 3° giro, ...
Invece nel caso in cui si abbiano perni ottimi i costi sono:



Quindi ogni passo ci costa al più n, cioè dobbiamo fare al più n confronti (alcuni di questi porteranno allo scambio e altri no).
Perciò se il n° dei confronti è un o (n), la complessità generale dell'algoritmo è:

- o (n^2), nel caso peggiore
- o ($n \log_2 n$), nel caso migliore

NOTA: la strutture che si prestano meglio a questo tipo di ordinamento sono i vettori e le liste bidirezionali (permettono di scorrere sia a destra che a sinistra), mentre la lista tradizionale non conviene.

4.7 Algoritmi di ordinamento per fusione (merge)

Si basano sul principio che risulta conveniente ordinare un insieme partendo da due insiemi a sua volta ordinati.
ESEMPIO:

- si abbia un insieme di n = 1000 elementi;
- si scomponga in due insiemi di 500 elementi ciascuno;

– si faccia un ordinamento con un metodo qualsiasi, per es. n^2 ⇒ costa n^2 = 250000 per ciascun insieme;

– si usi un algoritmo di fusione, che ora analizzeremo, tra insiemi ordinati che ha complessità pari a n; quindi riusciamo a ottenere, con un costo addizionale lineare (che potrà essere n o 2 n, ma comunque lineare), un insieme ordinato di 1000 elementi.

⇒ il costo finale sarà = $250000 \cdot 2 + \underbrace{(n \text{ oppure } 2\,n)}_{\text{trascurabile}} = 2\,(n/2)^2$

Se invece avessimo fatto direttamente l'ordinamento del vettore di 1000 elementi, avremmo speso n^2 = 1000000, che è molto maggiore (il doppio) di $2\,(n/2)^2$ = 500000.
Negli algoritmi di ordinamento per fusione si parte con l'ipotesi iniziale di avere due sottoinsiemi già ordinati:

n / 2

1	2
5	9
7	16
10	21
15	30
20	35
25	40

Da questi 2 sottoinsiemi ordinati possiamo ottenere un unico insieme di dimensione n ordinato in modo semplice: basta fare uno scorrimento dei 2 vettori per creare un nuovo vettore sulla base dei numeri che trovo.
Per esempio:

- confronto 1 e 2: il minore è 1 e lo riporto nel vettore unico;
- nel vettore a sinistra ci spostiamo sotto e confrontiamo 5 e 2: il minore è 2 e lo riporto nel vettore unico;
- nel vettore a destra ci spostiamo sotto e confrontiamo 5 e 9: il minore è 5 e lo riporto nel vettore unico;
- nel vettore a sinistra ci spostiamo sotto e confrontiamo 7 e 9: il minore è 7 e lo riporto nel vettore unico;
- nel vettore a destra ci spostiamo sotto e confrontiamo 10 e 9: il minore è 9 e lo riporto nel vettore unico;
- nel vettore a sinistra ci spostiamo sotto e confrontiamo 10 e 16: il minore è 10 e lo riporto nel vettore unico;

- nel vettore a sinistra ci spostiamo sotto e confrontiamo 15 e 16: il minore è 15 e lo riporto nel vettore unico;
- nel vettore a sinistra ci spostiamo sotto e confrontiamo 20 e 16: il minore è 16 e lo riporto nel vettore unico;
- nel vettore a destra ci spostiamo sotto e confrontiamo 20 e 21: il minore è 20 e lo riporto nel vettore unico;
- nel vettore a sinistra ci spostiamo sotto e confrontiamo 25 e 21: il minore è 21 e lo riporto nel vettore unico;
- nel vettore a destra ci spostiamo sotto e confrontiamo 25 e 30: il minore è 25 e lo riporto nel vettore unico;
- a questo punto abbiamo finito e quindi da lì in poi copio i restanti elementi del vettore a destra nel vettore unico

Abbiamo così ottenuto un unico vettore:

1
2
5
7
9
10
15
16
20
21
25
30
35
40

Abbiamo così fatto 2 . n / 2 confronti (operazioni di scorrimento).
L'operazione di fusione o merge si può fare anche su insiemi aventi dimensioni diverse.
Quindi l'operazione di merge costa un $o(n)$.
L'operazione totale, cioè di ordinamento separato delle due parti che costa ciascuna $o(n^2 / 2)$, costa perciò $2 . o(n^2 / 2) = o(n^2)$.

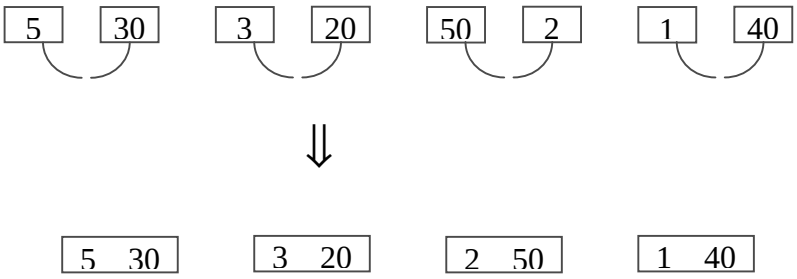
Pertanto la complessità asintotica di questo algoritmo di merge è ancora $o(n^2)$ cioè la stessa dell'ordinamento diretto usando il metodo n^2, e il risparmio lo abbiamo nella complessità di dettaglio cioè nella effettiva esecuzione.
NOTA: l'algoritmo di fusione non è il meno costoso: infatti costa n solo la fusione, ma non l'ordinamento che costerà a secondo del tipo usato (nel nostro caso abbiamo usato n^2).
NOTA: vale sempre l'ipotesi di chiave univoca, cioè che non ci siano 2 elementi uguali

4.8 Sort merge binario

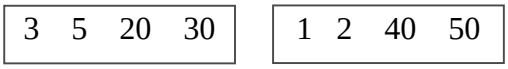
Supponiamo sempre di partire da un insieme di numeri disordinato, per es.:

5, 30, 3, 20, 50, 2, 1, 40

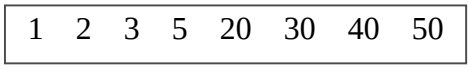
Si basa sul principio dell'algoritmo di fusione già visto; ricordiamo che esso era meno costoso dell'ordinamento poiché l'algoritmo di fusione è o (n), con n = somma del n° di elementi delle 2 liste.
Si ipotizza di partire dal massimo numero di insiemi su cui possa fare la fusione, in altre parole consideriamo coppie di elementi (che sono l'insieme più piccolo su cui possiamo fare la fusione).
Facendo la fusione ordinata di insiemi costituiti da un singolo elemento, eseguiamo una sorta di ordinamento poiché otteniamo altri insiemi di 2 elementi ordinati:



Così ora abbiamo insiemi di 2 elementi ordinati, ciascuno ottenuto dalla fusione di due insiemi da un elemento. Adesso fondiamo gli insiemi da 2 elementi per ottenere insiemi da 4 elementi:



Infine possiamo fare la fusione totale e trovare l'insieme finale:



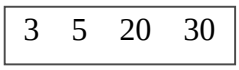
Si può notare che se abbiamo n elementi si arriva all'insieme finale ordinato in k passi, con

$$n = 2^k \Rightarrow k = \log_2 n$$

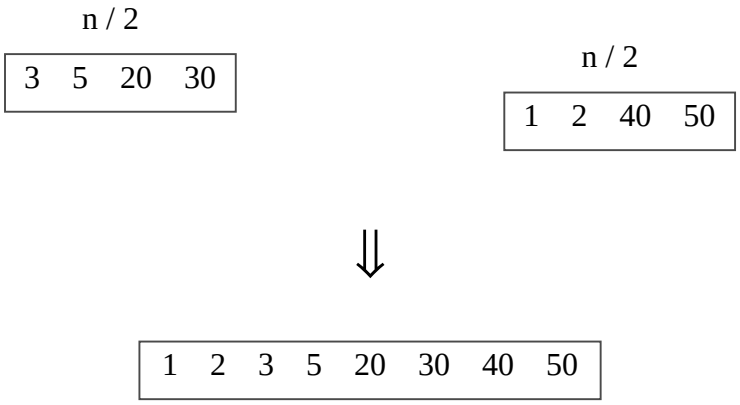
Infatti nel nostro caso n = 8 ⇒ k = 3.
Quindi la complessità intesa come n° di passi è $\log_2 n$; ma questa non è la complessità dell'algoritmo perché a ogni passo facciamo l'operazione di confronto.
Quando facciamo la fusione di due vettori di n / 4, eseguiamo al massimo 2 . n / 4 confronti.
[NOTA: allo stesso modo avevamo fatto per la fusione di 2 insiemi ordinati uno di dimensioni n e l'altro m: al massimo dovevamo fare asintoticamente n + m + 1 confronti cioè n + m].

Quindi per passare, per es., dai 2 insiemi:

all'insieme:



abbiamo bisogno, nel caso peggiore, di 2 . n / 4 = n / 2 confronti
La stessa cosa avviene per la parte a destra, per cui in totale facciamo n confronti.
Nella fusione successiva abbiamo che:



quindi abbiamo ancora n / 2 + n / 2 = n confronti.
Perciò ad ogni passo facciamo sempre n confronti (mentre nel quick sort li facciamo solo nel caso migliore cioè al più ne facciamo n) ⇒ la complessità asintotica dell'algoritmo di fusione è lineare cioè è $o(n)$.
Ora dato che la complessità asintotica delle operazioni che facciamo a ogni passo è $o(n)$ e la complessità asintotica del n° di passi è $o(\log_2 n)$, ne consegue che la complessità totale è $o(n \cdot \log_2 n)$, che è la stessa del quick sort.
Questo algoritmo visto adesso è molto semplice da realizzare con i vettori (basta andare sugli indici che possiamo calcolare facilmente); invece con le liste ci possono essere dei problemi, poiché ci sono tanti puntatori da dover ricercare e non li possiamo calcolare facilmente come gli indici dei vettori: un modo è utilizzare liste separate, ma devo gestire parecchi puntatori.

……..FINE……
Questo documento fa parte di una serie di dispense prodotte dallo sforzo volontario di molti (in questa parte in particolare vi sono molti pezzi estratti da lezioni trascritte da nastri) ma che sono state riviste e corrette dal docente solo in parte. Anche se in questo stato preliminare ho deciso di distribuirle per dare comunque una traccia sugli argomenti del corso in mancanza di un libro di riferimento. Non rappresentano pertanto ne' il libro ne' il programma di riferimento del corso, a questo fine fanno fede le lezioni stesse che in molte parti differiranno da queste dispense.
Si prega pertanto di segnalare ogni mancanza e correzione inviando una e-mail al Prof. P. Nesi al seguente indirizzo di posta elettronica: nesi@dsi.unifi.it , con la speranza di arrivare a produrre una versione piu' corretta e completa in breve tempo, anche con il Vostro aiuto, grazie!