STRUTTURE DATI

D'ANGELO CARMINE

APPUNTI TEORIA

STRUTTURE DATI

Sviluppo dei programmi analisi e specifica: è la fase in cui si individuno i dati di ingresso e di uscita di un programma (precondizione e post condizione). Se la precondizione è vere e il programma è eseguito allora la postcondizione è vera.

ES: ordinamento di una sequenza di interi
• *Dati di ingresso: sequenza s di n interi*
• *Precondizione: n > 0*
• *Dati di uscita: sequenza s1 di n interi*
• *Postcondizione: s1 è una permutazione di s dove per ogni i appartiene [0, n-2],* $s1_i \leq s1_{i+1}$

Progettazione: Definizione di come il programma effettua la trasformazione delle specifiche definite prima, progettazione dell'algoritmo per raffinamenti successivi(stepwise refinement) e infine decomposizione funzionale.

Codifica e verifica: Codifica dell'algoritmo nel linguaggio che si è scelto, testing del programma con dei casi di prova e infine verifica dei risultati attesi.

Algoritmi di ordinamento

Selection sort: Effetttua una visita totale delle posizioni dell'array, cioè visita in sequenza tutti gli elementi dell'array. Per ogni posizione visitata individua l'elemento che dovrebbe occupare quella posizione nell'array e lo scambia con quello che occupa l'attuale posizione.
Insertion sort: Ad ogni passo gli elementi che precedono l'elemento corrente sono ordinati, cioè controlla ogni volta che gli elementi precedenti quello corrente siano ordinati.
Bubble sort: finchè l'array non risulta ordinato si effettua una visita durante la quale si scambiano gli elementi adiacenti che non risultano ordinati, quando in un operazione non saranno effettuati scambi allora l'array sarà ordinato.

Puntatori e allocazione dinamica

Puntattori, deferenziazione operatore di indirizzo: Un puntatore è una variabile che contiene l'indirizzo di un'altra variabile, i puntatori sono di tipo *type bound* **,c**ioè ad ogni puntatore è associato il tipo a cui il puntatore si riferisce. L'accesso all'oggetto puntato avviene attraverso l'operatore di deferenziazione *, per ottenere l'indirizzo di un oggetto si usa *&*.
Il c supporta l'allocazione dinamica della memoria: infatti possiamo allocare la memoria durante l'esecuzione di un programma e assegnare l'indirizzo del blocco di memoria allocato ad un puntatore.
Abbiamo delle funzioni specifiche per fare questo, contenute in stdlib.h:

- ***Void *malloc(size_t size):*** alloca un blocco di memoria di size bytes senza inizializzarlo e restituisce il puntatore al blocco.
- ***Void *calloc(size_t nelements, size_t elementSize):*** alloca un blocco di memoria di nelements *nelementsSize bytes e lo inizializza a 0 (clear) e restituisce il puntatore al blocco.
- ***Void *realloc(void *pointer, size_t size):*** cambia la dimensione del blocco di memoria precedentemente allocato puntato da pointer. Restituice il puntatore ad una zona di memoria di dimensione size, che contiene gli stessi dati della vecchia regione indirizzata da pointer , anche troncata se la dimensione è minore.

Se non è possibile allocare la qantità di memori richiesta la funzione di allocazione restituisce un puntatore nullo. Per liberare la memoria allocata dinamicamente si usa la funzione:

void free(void *p).

Una variabile globale è visibile a tutte le funzioni la cui definizione segue la dichiarazione della variabile nel file sorgente, tale variabili sono dette statiche perchè la loro allocazione avviene all'atto del caricamento del programma e la loro deallocazione al termine.
Nella dichiarazione delle variabili abbiamo tre concetti importanti:

- **Scope:** parte del programma in cui è attiva una dichiarazione.
- **Visibilità:** dice quali variabili sono accessibili in una determinata parte del programma.
- **Durata:** periodo durante il quale una variabile è allocata in memoria.

Dichiarazioni static di funzioni: servono per realizzare l'information hiding, in generale una funzione definita in un file.c è utilizzabile in un altro file, anche in assenza della dichiarazioni esplicita del prototipo nell'header file. Dichiararle static le rende private al file in cui sono dichiarate. Anche per le variabili globali la situazione è analoga, dichiarare static una variabile locale ne cambia sola la durata non lo scope, infatti come effetto abbiamo che il suo valore rimane all'esecuzione della funzione in cui è definita.

Astrazione e modularizzazione

Modulazizzazione: dividere il programma in funzioni, procedure, ecc.. per gestire le sue varie complessità. La modularizzazione genera dei moduli, che sono unità di codice che mettono a disposizioni risorse e servizi computazionali (dati, funzioni, ecc...). in c per definie un modulo si usa un particolare tipo di file chiamato header (*.h*), per accedere a queste risorse bisogna includere il suo header file dove opportuno. Il modulo implementa astrazioni funzionali e mette a disposizione attraverso la sua interfaccia funzioni e procedure che realizzano le astrazioni.
Astrazione: é un procedimento che consente da una parte di evidenziare le caratteristiche importanti di un problema e di ignorare gli aspetti che si ritengono secondari rispetto ad un determinato obiettivo. Se ne fatto un argo uso durante la creazioni di funzioni (*astrazione funzionale*), lo scopo di quest'ultima è di concentrarsi su cos'à fa un determinato sottoprogrammma.
Abbaimo anche altri tipi di astrazione: sui dati(un dato è totalmente definito insieme alle operazioni che sul dato possono essere fatte), astrazione sul controllo.

Makefile e comando make

Il comando make ci permette la compilazione e il collegamento dei vari moduli che compongono il progetto, il *Makefile* è costituito da specifiche del tipo:

target_file: dipendeza_da_file
comandi (gcc,ecc...)

Nella compilazione i moduli possono essere compilati indipendentemente:

- *gcc -c utile.c*
- *gcc -c vettore.c*
- *gcc -c ordina_array.c*

Oppure compilarli insieme:

- *gcc -c utile.c vettore.c ordina_array.c*

In entrambi i casi si producono file .o, per collegarli e produrre un eseguibile :

- gcc utile.o vettore.o ordina_array.o -o ordina_array.exe

Nellla creazione di un Makefile l'ordine delle specifiche non è importante, ma è buona norma inserire com prima specifica quella per la costruzione del programma eseguibile. Per lanciare il processo basta digitare il comondo make. É buona norma inserire anche un comando clean per eliminare i file intermedi prodotti:

- *rm -f utile.o vettore.o ordina_array.o ordina_array..exe*

Testing e uso dei file

Il testing è una fase della programmazione in cui si provano dei dati di test sul programma per verificare che il suo comportamento sia quello atteso. *L'oracolo* è l'output atteso, *il malfunzionamento* è un comportamento inaspettato da quello che ci si aspettava. Per eseguire un buon testing è consigliato preparare un insieme di tutti i casi da provare(***test suite***).
In caso di malfunzionamento si può passare alla fase di *debugging* per individuare e correggere il difetto causato dal malfunzionamento. Un modo per cercare il difetto può essere fatto inserendo nel codice sorgente punti di ispezione dello stato delle variabili.
Per automatizzare il testing possiamo usare i ***file***. Leggere e scrivere su un file è effettuato tramite l'uso di stream, che possono essere di input e file. Un file memorizza i dati in maniera non volatile, in C il tipo **FILE** è definito in *stdio.h*, ed astrae il concetto di sequenza di dati contenuti nella memoria di massa.
Esistono 3 tipi di flussi,che sono pronti per essere usati e non c'è bisogno di dichiararli:

- ***STDIN:*** standard input.
- ***STDOUT:*** standard output.
- ***STDERR:***standard error.

Abbiamo due tipi di file(*testo,binario*), un file di testo è rappresentato da bytes di caratteri, e quindi possono essere stampati,letti e modificati con un editor di testo. In un file binario viceversa i byte possono assumere qualsiasi valore.
Per aprire un file si fa uso della funzione ***open***, ***file_pointer_variable = fopen(file_name, mode);***
Le modalità di apertura di un file sono diverse, e sono: r,w,a,r+,w+,a+.
Per chiudere un file si usa la funzione ***fclose***, il suo argomento è un puntatore ad un file precedentemente aperto, se la chiusura riesce restituisce 0 altrimenti *EOF*.
Per scrivere e leggere usiamo le funzioni *fprintf e fscanf*.

- ***fprintf(fp, "Total: %d\n", total);***
- ***fscanf(fp, "%d%d", &i, &j)***;

Entrambe le funzioni restituiscono un intero pari al numero di dati in input che è stato possibile assegnare ad altre variabili, restituiscono EOF se c'è un errore di input prima che si possa leggere un byte. Esitono tre tipi di EOF:

- ***EOF (End of file):*** la funzione ha trovato il segnalatore di fine file.
- ***Errore di lettura:*** la funzione non riesce a leggere altri caratteri dal flusso di input.
- ***Mancata corrispondeza di formatato:*** l'input non corrisponde al formato.

Ogni flusso ha due segnalatori ad esso associati, un segnalatore di errore ed uno di fine file, questi indicatori vengono azzerati (FALSE) quando il file viene aperto. Se si arriva alla fine del file viene settato a TRUE il segnalatore di fine file. Se si verifica un errore di input(o output) viene attivato il segnalatore di errore. Una mancata corrispondenza con il formato non attiva nessun segnalatore.
Per controllare se si è verificata una condizione di errore si usano le funzioni ***feof (fp)*** che restituisce un valore non zero se il segnalatore EOF è stato attivato dalla precedente operazione sullo stream fp; o la funzione ***ferror (fp)*** che restituisce un valore non zero se il segnalatore di errore di input è attivato.
In caso che il programma abbia più funzioni, ci sono varie strategie per eseguire il testing:

- **Strategia big-bang:** integra il programma con tutti i sottoprogrammi e lo verifica nel suo insieme.
- **Strategie incrementali:** testare e integrare un sottoprogramma alla volta, considerando la struttura delle chiamate tra sottoprogrammi.
- **Strategia bottom-up:** verifica prima i sottoprogrammi terminali e poi via via quelli di livelli superiore.
- **Strategia bottom-up e driver:** per ogni sottoprogramma da verificare è necessario costrutire un programma main che acquisisce dati di ingresso necessari al sottoprogramma, invoca il sottoprogramma passandogli i dati di ingresso e ottenendo i dati di uscita, visualizzare i dati di uscita del sottoprogramma.

DATI ASTRATTI

L'astrazione dati ricalca ed estende il concetto di astrazione funzionale , permettendo di ampliare i tipi di dati disponibili attraverso la creazione di nuovi tipi di dati e operatori. L'astrazione dati sollecita ad individuare le organizzazioni dati più adatte per risolvere un problema.

- ***Specifica:*** descrivere un nuovo tipo di dati e gli operatori applicabili. La specifica descrive l'astrazione dati e il modo in cui può essere utilizzata attraverso I suoi operatori (specifica sintattica e semantica).
    1. ***Sintattica***: nome del tipo di dato di riferimento e gli eventuali tipi di dati usati già definiti, i nomi delle operazioni del tipo di dati di riferimento, i tipi di dati di input e di output per ogni operatore.
    2. ***Semantica***: l'insieme dei valori associati al tipo di dati di riferimento, la funzione associata ad ogni nome di operatore specifica le seguenti condizioni:
        I. ***Precondizione***: definita sui valori dei dati di input, definisce quando l'operatore è applicabile.
        II. ***Postcondizione***: definita sui valori dei dati di output e di input, stabilisce la relazione tra gli argomenti e risultato.
- ***Realizzazione:*** come il nuovo dato e i nuovi operatori vengono ricondotti ai dati e agli operatori già disponibili. Utilizzo di meccanismi di programmazione modulare offerti dal linguaggio di programmazione utilizzato per mettere a disposizione l'astrazione attraverso un'interfaccia e nascondere i dettagli dell'implementazione.

ES: Libro:

Specifica dei tipi di dati

- Specifica ***sintattica:***
    Tipo di riferimento: libro
    Tipi usati: stringa, intero
- Specifica ***semantica:***
    Il tipo libro è l'insieme delle quadruple (autore, titolo, editore, anno) dove autore, titolo e editore sono stringhe e anno è un intero.

Specifica degli operatori

- Specifica ***sintattica***
    creaLibro(stringa, stringa, stringa, intero) ->libro
    autore (libro)→stringa
    titolo (libro)→stringa
    editore (libro)→stringa
    anno (libro)→intero
- Specifica ***semantica***
    1. *creaLibro(aut, tit, ed, an) = lb*
        Post: lb = (aut, tit, ed, an)
    2. *autore(lb) = aut*
        Post: lb = (aut, tit, ed, an)
    3. *titolo(lb) = tit*
        Post: lb = (aut, tit, ed, an)
    4. *editore(lb) = ed*
        Post: lb = (aut, tit, ed, an)
    5. *anno(lb) = an*
        Post: lb = (aut, tit, ed, an)

Per far in modo che la struttura non si inserida nell'header file e quindi nascosta, possiamo definire la struttura come un puntatore:*"typedef struct pto *punto;"*. L'implementazione di pto è definita nel file punto.c, in modo così da non renderla visibile al client tramite l'inclide dell'header file *punto.h*.

ADT LISTA

Una lista è una sequenza di un determinato tipo di elementi a cui è possibile aggiungere o togliere elementi dello stesso tipo. Per poter far ciò occorre specificare la posizione relatva all'interno della sequenza nella quale il nuovo elemento va aggiunto o nella quale il vecchio elemento va tolto. A differenza di un array, la lista è a dimensione variabile e si può accedere direttamente solo al primo elemento, per poter accedere ad un altro elemento occorre scandire sequenzialmente gli elementi della lista.

Specifica sintattica

- **Tipo di riferimento:** list
- **Tipi usati:** item, boolean
- **Operatori:**

  newList() → list
  emptyList(list) → boolean
  consList(item, list) → list
  tailList(list) → list
  getFirst(list) → item

Specifica semantica

- **Tipo di riferimento list:** list e l'insieme delle sequenze L=<a1,a2,…,an> di tipo item, l'insieme list contiene inoltre un elemento **nil** che rappresenta la lista vuota (priva di elementi).
- **Operatori:**
  1. *newList() → l*
     Post: l = nil
  2. *emptyList(l) → b*
     Post: se l=nil allora b = true altrimenti b = false
  3. *consList(e, l) → l'*
     Post: l = <a1, a2, … an> AND l' = <e, a1, a2, …, an>
  4. *tailList(l) → l'*
     Pre: l = <a1, a2, …, an> n>0
     Post: l' = <a2, …, an>
  5. *getFirst(l) → e*
     Pre: l = <a1, a2, …, an> n>0
     Post: e = a1

LISTE CONCATENATE

Ogni elemento di una lista concatenata è un record diviso in due parti, una in cui c'è l'elemento e un'altra in cui c'è il puntatore al record successivo. Si accede alla struttura attraverso il puntatore al primo record, e il campo puntatore del'ultimo record contiene il valore NULL. In una lista concatenata è molto più semplice inserire e cancellare un elemeto perchè si utilizza solo la memoria strettamente necessaria. L'unico svantaggio è che si perde la capacità di accedere in modo diretto agli elemeni della lista. Il modo più semplice per inserire un nuovo elemento in una lista è quello di inserire un elemento in un nodo che sarà poi aggiunto in testa alla lista.
Per poter usare una lista serve una strttura che rappresenti i nodi, la struttura conterrà i dati necessari ed un pntatore al prossimo elemento della lista:

```
struct node{
        item value;
        struct node *next;
};
```

nota: node deve essere un'etechitta di struttura e non un tipo definito con typedef, altrimenti non sarebbe possibile dichiarare il tipo di next.
Il passo successivo è quello di dichiarare il tipo lista: *typedef struct node *list;*, una variabile di tipo

lista punterà al primo node della lista: *list l=NULL*, assegnare a l il valore NULL indica che la lista è inizialmente vuota. Man mano che costruiamo la lista, creiamo dei nuovi nodi da aggiungere alla lista, I passi per creare un nodo sono:

- **Allocare la memoria necessaria:** per creare un nodo ci serve un puntatore temporaneo che punti al nodo: *struct node *new_node;*, per allocare memoria si può usare la funzione malloc: *new_node=malloc(sizeof(struct node));*, da questo momento in poi node punta ad un blocco di memoria che contiene la struttura di tipo node.
- **Memorizzare i dati nel nodo.**
- **Inserire il nodo nella lista**.

Inserimento**(insertList(l, pos, val))**/rimuovere in una lista **(remove list(l, pos))**: per inserire scorriamo la lista l e inseriamo in una lista di appoggio tutti gli elementi della lista l di input che precedono la posizione pos in cui inserire val, inseriamo val nella lista di appoggio e inseriamo i restanti elementi di l nella lista di appoggio, infine restituiamo il reverse della lista di appoggio. Stessa cosa vale per la rimozione.
Questa implementazione porta al problema del garbage in c, che puo portare ad un heap overflow. Quindi se nell'implementazione dei nuovi operatori di inserimento, invece di usare gli operatori, si può inserire un elemento in una lista concatenata come successore di uno qualunque dei record già presenti, lavorando direttamente sulla struttura a puntatori non abbiamo bisogno degli operatori di base, quindi continuiamo ad usare solo ***consList*** perchè ci serve per allocare memoria per un nuovo nodo da inserire nella lista, stesso discorso vale per la rimozione.
Il nuovo inseriemento quindi sarà: si inserisce un elemento in una lista concatenata in qualunque posizione oltre alla prima, come successore di uno dei record già presenti, nel seguente modo:

- per prima cosa si crea il collegamento con il record successivo.
- poi si crea il collegamento con il record precedente.

Per l'eliminazione il metodo è:

- per prima cosa si salva il collegamento al record successivo in una variabile temporanea.
- poi si collega il record precedente al record successivo.
- infine si elimina il record.

Con questa soluzione deallochiamo anche la memoria.
Questa implementazione però porta a diversi problemi, per esempio se si copia una lista in un altra lista e poi si opera su una delle due, entrambe saranno modificate perchè condividono la stessa area di memoria, portando alla nascita di un effetto collaterale ***(side effect)***.
Se vogliamo evitare il side effect in questo caso basta inserire un operatore che duplica la struttura e il contenuto della lista: *list cloneList (list)*, mentre per insert e remove possono essere utilizzati solo sulla variabile che prendono in input. Se queste covenzioni non fossero rispettate ci si troverrebbe con molti problemi, per evitare ciò si potrebbero progettare gli operatori in modo che il parametro list sia al contempo sia di ingresso che di uscita, passandolo come puntatore, ma questa soluzione non è elegante, quindi per evitare di passare un puntatore abbiamo bisogno di progettare in maniera diversa la struttura dati, la soluzione sarà quella di creare un ulteriore struttura node che contiene a sua volta un riferimeto al primo elemento della lista, in questo modo le modifiche possono essere effetuate anche senza passare un puntatore a list.

ADT STACK (PILA)

Lo stack è un tipo di dato astratto, spesso è chiamato anche pila, è una sequenza di elementi di un determinato tipo, in cui è possibile aggiungere o togliere elementi esclusivamente da un unico lato (***top*** dello stack). Questo significa che la sequenza viene gestita in modalità **LIFO** *(Last in first out)*.
La pila è una struttura dati lineare a dimensione variabile in cui si può accedere direttamente solo al primo elemento della lista. Non è possibile accedere ad un elemento diverso dal primo, cioè l'ultimo che è stato inserito, se non dopo aver eliminato tutti gli elementi che lo precedono.
Push: inserire, **Pop:** togliere.

Specifica sintattica

- **Tipo di riferimento**: stack, e l'insieme delle sequenze *S=<a1,a2,...,an>* di tipo *item.*
- **Tipi usati**: item, boolean, L'insieme stack contiene inoltre un elemento ***nil*** che rappresenta la pila vuota (priva di elementi).
- **Operatori**

  newStack() → stack
  emptyStack(stack) → boolean
  push(elem, stack) → stack
  pop(stack) → stack
  top(stack) → item

Specifica semantica

- **Operatori:**

1. *newStack()* → *s*
   Post: s = nil
2. *emptyStack(s)* → *b*
   Post: se s=nil allora b = true altrimenti b = false
3. *push(e, s)* → *s'*
   Post: s = <a1, a2, … an> AND s' = <e, a1, a2, …, an>
4. *pop(s)* → *s'*
   Pre: s = <a1, a2, …, an> n>0
   Post: s' = <a2, …, an>
5. *top(s)* → *e*
   Pre: s = <a1, a2, …, an> n>0
   Post: e = a1

Il tipo stack può essere implementata attraverso vari modi, le più usate sono *array* o *lista concatenata.*

- **Implementazione con array:** è implementato come un puntatore ad una *struct c_stack* che contiene due elementi, un array di MAXSTACK elementi e un intero che indica la posizione del top dello stack. Quando lo stack si riempie non è più possibile eseguire l'operazione push.
- **Implementazione senza dimensione max:** bisogna usare l'allocazione dinamica della memoria e due costanti, *STARTSIZE* definisce la dimensione iniziale dello stack, mentre *ADDSIZE* definisce di quanto allargare lo stack nel caso in cui si riempia. Questo vuol dire che occorre anche una variabile size che ci dica quanti elementi può contenere lo stack in ogni momento.
- **Implementazione con liste collegate:** il tipo stack è definito come un puntatore ad una struct che contiene: un intero *numelem* che indica il numero di elementi dello stack, e un puntatore *top* ad una *struct nodo*.
- **Implementazione basata sull'uso del modulo lista:** il tipo stack è definito come un puntatore ad una struct che contiene: un elemento *top* di tipo *list*, l'intero che indicava il numero di elementi all'interno dello stack non serve più, anche se abbiamo un solo elemento nella struct , continiamo a definire il tipo stack come puntatore a *struct c_stack* per non cambiare la definizione nell'header file.
- **Implementazione con una singola instanza:** in questo caso non abbiamo bisogno di definire ed esportare un tipo, la struttura dati dello stack la manteniamo in variabili globali accessibili solo all'interno del modulo(dichiarazioni static). Gli operatori nella lista non usano parametri di tipo stack, ma operano sulle variabili globali. Stiamo realizzando un singolo oggetto stack.

MODULI E ASTRAZIONI SUI DATI: TIPI DI DATI ASTRATTI E OGGETTI

- **Tipo di dato astratto: i**l modulo incapsula la definizione del tipo e gli operatori, esporta il nome del tipo e la signature degli operatori. Il tipo di riferimento compare tra i parametri degli operatori. É consentito definire variabili di tale tipo **(istanziazione)** e utilizzarle **(referenziazione)** come parametri degli operatori.
- **Oggetto (astratto):** il modulo incapsula una struttura dati (istanza) e gli operatori ed esporta la signature degli operatori. Non è consentito referenziare la struttura dati fuori dal modulo, e l'uso e la manipolazione dell'oggetto sono consentiti unicamente attraverso l'uso dei suoi operatori. Un oggetto ha uno stato che può cambiare in seguito all'appliczione di determinate operazioni.
- **Genericità (modulo generico):** un template dal quale è possibile istanziare più moduli, tipicamente è parametrico rispetto a un tipo base o al numero di componenti di tipo base: *es: ADT (o oggetto) stack generico*. Un modulo cliente dovrebbe prima istanziare il modulo specificando i parametri di struttura e poi ... in c non abbiamo costrutti per fare questo, invece dobbiamo definire un tipo generico item e delle costanti, che possiamo cambiare all'occorrenza.

ADT CODA (QUEUE)

Una cosa è una sequenza di elementi di un determinato tipo, in cui gli elementi si aggiungono da un lato(**tail**) e si tolgono dall'altro lato(**head**). La sequenza è gestita in modalità **FIFO**, la coda è una struttura di dati lineare a dimensione variabile in cui si può accedere direttamente solo all'head della lista. Non è possibile accedere ad un elemento diverso da esso,se non dopo aver eliminato prima tutti gli elementi che li precedono.

Specifica sintattica

- **Tipo di riferimento**: queue, e l'insieme delle sequenze *S=<a1,a2,...,an>* di tipo ***item.*** *L'*insieme queue contiene inoltre un elemento *nil* che rappresenta la coda vuota (priva di elementi).
- **Tipi usati**: item, boolean
- **Operatori:**

  newQueue() → queue
  emptyQueue(queue) → boolean
  enqueue(item, queue) → boolean
  dequeue(queue) → item

Specifica semantica

- **Operatori:**
  1. *newQueue() → q*
     Post: q = nil
  2. *emptyQueue(q) → b*
     Post: se q=nil allora b = true altrimenti b = false
  3. *enqueue(e, q) → b e la coda diventa q'*
     Pre: q = <a1, a2, … an> con n >= 0
     Post: se q' = <e, a1, a2, …, an> allora b = true
  4. *dequeue(q) → an e la coda diventa q'*
     Pre: q = <a1, a2, …, an-1, an> n>0
     Post: q' = <a1,a2, …, an-1>

Il tipo *coda* può essere implementata attraverso vari modi, le più usate sono *array* o *lista concatenata.*

- **Implementazione con liste collegate:** a differenza dello stack, per gestire la politica FIFO conviene aver accesso sia al primo elemento (estrazione) sia all'ultimo (inserimento). Il tipo queue è definito come un puntatore ad una struct che contiene: un intero numelem che

indica il numero di elementi della cosa, un puntatore head ad uno struct nodo ed uno tail ad un'altra struct nodo.

- **Implementazione con array:** la coda è implementata come un puntatore ad una struct c_queue che contiene due elementi: un array di *MAXQUEUE* elementi, un intero che indica la posizione head della coda e un intero tail che indica la posizione della coda. Quando la coda si riempie, non è possibile eseguire l'operazione enqueue. Con questa rappresentazione possono esserci dei problemi se HeadQ < TailQ, infatti se su un array di 7 posizioni eliminamo le prime 3(1,2,3) rimanendo disponibili le ultime 3(5,6,7) ci risulterà sempre che sono disponibili solo quest'ultime. Le soluzioni possono essere 2:
    1. Ad ogni rimozione si compatta l'array nelle posizioni iniziali (MOLTO COSTOSO).
    2. Si gestisce l'array in modo circolare. Inizialmente abbiamo *HeadQ=TailQ=1*, in ogni istante, gli elementi della coda si trovano nel segmento *HeadQ,HeadQ+1,...,TailQ-1*, però non necessariamente *HeadQ=TailQ*. Infatti, dopo aver inserito in posizione N, se c'è ancora spazio in coda, si inseriscono ulteriori elementi a partire dalla prima posizione, in questo modo si riesce a garantire che ad ogni istante la coda abbia capacità massima di N elementi. Quando *HeadQ=TailQ* la coda è vuota, se *HeadQ=TailQ+1* la coda è piena, questo comporta ad avere la presenza di una locazione vuota per capire se il buffer è vuoto o pieno. Le operazioni di *cancellazione/inserimento* in testa spostano *avanti/indietro* l'indice **head**, mentre per l'indice **tail** lo spostano *indietro/avanti*. Combinando le diverse operazioni il segmento occupato dalla sequenza di elementi ruota nel buffer con l'effetto di allungarsi in senso orario/antiorario per gli inserimenti testa/coda, e in senso antiorario/orario per le cancellazioni testa/coda.

CONSIDETAZIONI

Abbiamo visto due implementazioni diverse dell'ADT coda.

Vettore circolare:

- **pro:** occupa poco spazio in memoria e gli elementi sono memorizzati in modo contiguo.
- **contro:** bisogna conoscere a priori il numero massimo di elementi che la coda deve contenere.

Lista

- **pro:** è una implementazione espandibile.
- **contro:** gli elementi non sono memorizzati in celle di memoria contigue e l'occupazione di spazio è maggiore.

RICORSIONE

RECORD DI ATTIVAZIONE

Ogni volta che viene invocata una funzione, si crea una nuova attivazione (istanza) del servitore, viene quindi allocata la memoria per i parametri e per le variabili locali. Si effettua il passaggio dei parametri e si trasferisce il controllo al servitore, infine si esegue il codice della funzione.
Al momento dell'invocazione viene creata dinamicamente una struttura dati che contiene i binding dei parametri e degli identificatori localmente alla funzione detta RECORD DI ATTIVAZIONE.
É il "mondo della funzione": contiene tutto ciò che serve per la chiamata alla quale è associato:

- **parametri formali**, **variabili locali**;
- **indirizzo di ritorno** *(Return address RA)* che indica il punto a cui tornare (nel codice del cliente) al termine della funzione, per permettere al cliente di proseguire una volta che la funzione termina.
- Un collegamento al record di attivazione del cliente **(Link Dinamico DL)**.
- **Indirizzo del codice** della funzione*(puntatore alla prima istruzione del corpo)*.

Il record di attivazione associato a una chiamata di una funzione f, è creato nel momento dell'invocazione di f e permane per tutto il tempo in cui la funzione f è in esecuzione, è

distrutto(deallocato) al termine dell'esecuzione della funzione stessa.
Ad ogni chiamata di funzione viene creato un nuovo record, specifico per quella chiamata di quella funzione. La dimensione del record di attivazione varia da una funzione all'altra, per una data funzione, è fissa e calcolabile a priori. Funzioni che chiamano altre funzioni danno luogo a una sequenza di record di attivazioni, allocati secondo l'ordine delle chiamate e deallocati in ordine inverso. La sequenza dei link dinamici costituisce la cosidetta catena dinamica, che rappresenta la storia delle attiviazioni. L'area di memoria in cui vengono allocati i record di attivazione viene gestite come una pila.

Es: chiamate annidate

Programma:

```
int R(int A) { return A+1; }
int Q(int x) { return R(x); }
int P(void) { int a=10; return Q(a); }
main() { int x = P(); }
```

Sequenza chiamate:

S.O. → main → P() → Q() → R()

Sequenza di attivazione:

R() → Q() → P() → main

RICORSIONE

Una funziona matematica è definita ricorsivamente quando nella sua definizione compare un riferimento a se stessa. La ricorsione consiste nella possibilità di definire una funzione in termine di se stessa. È basata sul principio di induzione matematica:
se una proprietà P vale per $n=n_0$ (**CASO BASE**), e si può provare che, **assumendola valida per n,** allora vale per n+1, allora P vale per ogni n >= n0.
Operativamente, risolvere un problema con un approccio ricorsivo comporta di identificare un caso base, con soluzione nota, e di riuscire a esprimere la soluzione al caso generico n in termini dello stesso problema in uno o più casi più semplici (n-1, n-2, ecc...). Un sottoprogramma ricorsivo è un sottoprogramma che richiama direttamente o indirettamente se stesso. Non tutti i linguaggi realizzano il meccanismo della ricorsione. Quelli che lo realizzano possono utlizzare due tecniche:

- **Gestione LIFO** più copie della stessa funzione, ciascuna con il proprio insieme di variabili locali.
- **Gestione mediante record di attivazione:** un'unica copia del sottoprogramma ma ad ogni chiamata è associata un record di attivazioni(variabili locali e punto di ritorno).

Esempio Fattoriale:

fact(n)=n!
n! : N → N allora n! vale 1 se n==0
n! vale n(n-1)! Se n>0*

Con n=5 significa:

```
0!=1
1!=1*(1-1)!=1*0!=1
2!=2*(2-1)!=2*1!=2
3!=3*(3-1)!=3*2!=3*2*1=6
4!=4*(4-1)!=4*3!=4*3*2*1=24
5!=5*(5-1)!=5*4!=5*4*3*2*1=120
```

Quindi per ogni n intero positivo, il fattoriale di n è il prodotto dei primi n numeri interi positivi.
In C è possibile definire funzioni ricorsive, il corpo di ogni funzione ricorsiva contiene almeo una chiamata alla funzione stessa.

ES: fattoriale

```
int fact(int n)
{
        if (n==0) return 1;
                else return n*fact(n-1);
}
```

Nota: fact è sia servitore che cliente (di se stessa).

```
main()
{
        int fz,f6,z = 5;
        fz = fact(z-2);
}
```

In una chiamta ricorsiva le funzioni sono inserite in uno stack, per esempio prendiamo n=3, avremo il seguente schema di stack:

fact(0)
fact(1)
fact(2)
fact(3)
main

Altri esempi di ricorsione:

1) Calcolare la somma dei primi N interi

- *Algoritmo ricorsivo*: Se N vale 1 allora la somma vale 1, altrimenti la somma vale N+ il risultato della somma dei primi N-1 interi.
- *Specifica*: Considera la somma 1+2+3+...+(N-1)+N come composta di due termini:
    I. **(1+2+3+...+(N-1))**: Il primo termine non è altro che lo stesso problema in un caso più semplice, calcolare la somma dei primi N-1 interi.
    II. **N**: Valore noto.
- *Esiste un caso banale* ovvio: CASO BASE = la somma fino a 1 vale 1.
- *Codifica:* ***int sommaFinoA(int n)***

```
{
    if(n==1) return 1;
    else return sommaFinoA(n-1)+n;
}
```

2) Calcolare l'N-esimo numero di Fibonacci

$$fib(n) = \begin{cases} 0, & \text{se } n=0 \\ 1, & \text{se } n=1 \\ fib(n-1) + fib(n-2), & \text{altrimenti} \end{cases}$$

Codifica: ***unsigned fibonacci(unsigned n)***

```
{
    if (n<2) return n;
    else return fibonacci(n-1) + fibonacci(n-2);
}
```

Questa ricorsione è chiamata *ricorsione non lineare*, cioè ogni invocazione del servitore causa due nuove chiamate al servitore medesimo.

RIFLESSIONI SULLA RICORSIONE

Negli esempi visti finora si inizia a sintetizzare il risultato, ***SOLO DOPO*** che si sono aperte tutte le chiamate, "a ritroso", mentre le chiamate si chiudono.
Nota: Le chiamate ricorsive decompongono via via il problema, **ma non calcolano nulla.**
Il risultato viene sintetizzato *a partite dalla fine,* perchè occorre arrivare al caso "banale":

- il caso "banale" fornisce il valore di partenza.
- Poi si sintetizzan, "a ritroso", i successivi i risultati parziali.

Si potrebbe pensare che le chiamate ricorsive si possano succedere una dopo l'altra, all'infinito.
Invece, se:

- ad ogni invocazione il problema diventa sempre più semplice e si avvicina al caso base.
- la soluzione del caso base non richiede ricorsione.

Allora, per quanto complesso possa essere il problema, certamente la soluzione viene calcolata in un numero finito di passi. Quindi la soluzione ricorsiva di un problema è un **algoritmo** in quanto arriva a conclusione in numero finito di passi. Se manca il caso base, o ad ogni passo ricorsivo la soluzione non si semplifica, il metodo continua a richiamare se stesso all'infinito dando luogo ad una *ricorsione infinita:*

```
int main(void)
{
        main();
}
```

Costruiamo ora una funzione che calcola il fattoriale in modo iterativo

```
int fact(int n){
      int i=1;
      int F=1; /*inizializzazione del fattoriale*/
      while (i <= n)
            { F=F*i;
              i=i+1; }
      return F;
      }
```

DIFFERENZA CON LA VERSIONE RICORSIVA: ad ogni passo viene accumulato un risultato intermedio

La variabile F accumula risultati intermedi: se n = 3 inizialmente `F=1`, ***poi al primo ciclo*** `F=1`, ***poi al secondo ciclo*** `F` ***assume il valore 2. Infine all'ultimo ciclo*** `i=3` ***e*** `F` ***assume il valore 6***

- Al primo passo F accumula il fattoriale di 1
- Al secondo passo F accumula il fattoriale di 2
- Al passo i-esimo F accumula il fattoriale di i

21

ITERAZIONE

Nell'esempio precedente il risultato viene sintetizzato "in avanti", l'esecuzione di un algoritmo di calcolo che computi "in avanti", per accumulo, è un *processo computazionale iterativo.*
La caratteristica fondamentale di un processo computazionale iterativo è che a ogni passo è disponibile un risultato parziale, dopo K passi, si ha a disposizione il risultato parziale relativo al

caso K, questo non è vero nei processi computazionali ricorsivi, in cui nulla è disponibile finchè non si è giunti fino al caso elementare.
Un processo computazionale iterativo si può realizzare anche tramite funzioni ricorsive. Si basa sulla disponibilità di una variabile, detta accumulatore, destinata a esprimere in ogni istante la soluzione corrente. Si imposta identificando quell'operazione di modifica dell'accomulatore che lo porta a esprimere, dal valore relativo al passo k, il valore relativo al passo k+1.

Definizione:

$$n! = 1 * 2 * 3 *\ldots * n$$

Detto $v_k = 1 * 2 * 3 *\ldots * k$**:**

$$1! = v_1 = 1$$

$$(k+1)! = v_{k+1} = (k+1) * v_k \qquad \textit{per } k \geq 1$$

$$n! = v_n \qquad \textit{per } k=n$$

RICORSIONE VS ITERAZIONE

- **Ripetizione:** *iterazione*: ciclo esplicito.
  ricorsione: chiamate di funzione ripetute.
- **Terminazione:** *iterazione*: il ciclo fallisce la condizione.
  ricorsione: il caso base è riconosciuto.

Entrambe possono dar luogo a cicli infiniti, la scelta finale è data all'utente, per una buona perfomance(**iterazione**), per una buona ingegneria del software(**ricorsione**).

RICORSIONE

- Uso di un costrutto di selezione
- Condizione di terminazione
- Non convergenza

ITERAZIONE

- Uso di un costrutto di iterazione
- Condizione di terminazione
- Loop infinito

A differenza dell'iterazione, la ricorsione richiede un notevole sovraccarico (*overhead*) a tempo d'esecuzione dovuto alle chiamate di funzione.

Dato che un programma ricorsivo può essere sempre trasformato in un programma iterativo, perché usare la ricorsione?

Gli algoritmi che per loro natura sono ricorsivi piuttosto che iterativi dovrebbero essere formulati con procedure ricorsive.
Ad esempio alcune strutture dati sono interamente ricorsive: a**lberi** e **sequenze**. La formulazione ricorsiva di algoritmi su di esse risulta più naturale. La ricorsione deve essere evitata quando esiste una soluzione iterativa ovvia e in situazioni in cui le prestazioni del sistema sono un elemento critico.

COMPLESSITÁ COMPUTAZIONALE

È utilizzata per verificare il costo degli algoritmi in termini di risorse di calcolo(Tempo,spazio di memoria).
Es: dato un vettore v di interi n ordinati in maniera non decrescente verificare se un intero k è presente o meno in v:

```
Int ricerca(int v[], int size, int k)
{
        int i;
        for (i=0; i<size; i++)
                if (v[i] == k) return i;

        return -1;
}
```

Non c'è nessun vantaggio perchè il vettore è ordinato.
Per la valutazione del tempo ci sono molti fattori da prendere in considerazione: *la macchina usata, configurazione dei dati, dimensione dei dati, modello di macchina astratta, caso peggiore di configurazione dei dati, funzione della dimensione dell'input, comportamento asintotico,* di seguito analizzeremo più nel dettaglio ognuno di questi casi.
Macchina astratta: costo unitario delle istruzioni e delle condizioni atomiche. Le strutture di controllo hanno un costo pari alla somma dei costi dell'esecuzione delle istruzioni interne, più la somma dei costi delle condizioni. Le chiamate a funzioni hanno invece un costo pari al costo di tutte le sue istruzioni e condizioni; il passaggio di parametri ha costo *null*. Infine istruzioni e condizioni con chiamate a funzioni hanno costo pari alla somma del costo delle funzioni invocate più uno.
ES: calcolare il costo per l'esempio precedente nel caso v[n] = {1,3,4,17,34,95,96,101} e k = 9

Inizializzazioni (i=0)	*1 +*
Confronti (i<size)	*3 +*
Confronti (v[i]==k)	*3 +*
Instruzioni (i++)	*2 +*
Istruzioni (return i)	*1 =*
	10

Cosa cambia se k=10 ?
Caso peggiore: caso che a parità di demsione produce il costo massimo, se accettabile nel caso peggiore. Nel caso dell'esempio, k non presente:

Inizializzazioni (i=0)	*1 +*
Confronti (i<size)	*n+1 +*
Confronti (v[i]==k)	*n +*
Istruzioni (i++)	*n +*
Istruzioni (return -1)	*1 =*
	3n+3

Caso medio: Equiprobabilità dell'input. La probabilità che k sia in posizione i(1<=i<=n) vale 1/n. Costo del caso in posizione i: 3i+1. Costo del caso medio:

$$\frac{1}{n}\sum_{i=1}^{n}(3i+1)=\frac{1}{n}\left(3\frac{n^2+n}{2}+n\right)=$$

$$=\frac{3n+5}{2}$$

Costo come funzione della dimensione dell'input: cos'è la dimensione?

- Vettore: numero di elementi.
- Albero: numero dei nodi.
- Grafo: numero archi più numero nodi.

Es: calcolo del fattoriale, con tipo intero non limitato.

```
int fattoriale(int n)
{
        int i = 1;                  inizializzazioni (i=1)          1 +
        int fatt = 1;               inizializzazioni (fatt=1)       1 +
        while (i <=n)               confronti (i<=N)                n+1 +
        {
                fatt = fatt * i;    istruzioni (fatt=fatt*i)        n +
                i++;                istruzioni (i++)                n +
        }
        return fatt;                istruzioni (return fatt)        1 =
}                                                                   3n+4
```

Dimensione dell'input: n, 3n+4 oppure numero *d* di cifre decimali necessarie per rappresentare *n* in decimale... $d \approx \log_{10} n$ ... $3 * 10^4$... esponenziale.

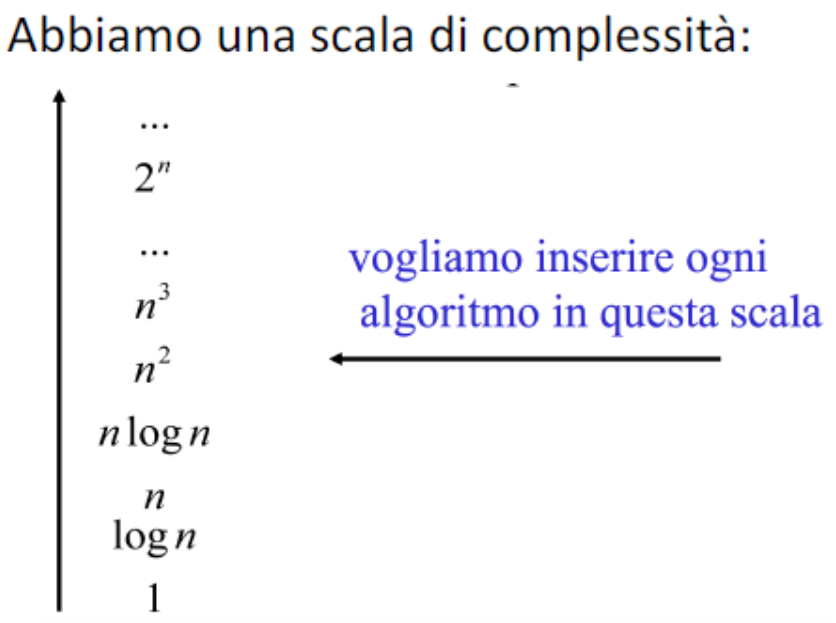
Es: somma inefficiente:

```
int somma (int n, int m)
{
        while (m>0) { n=n+1; m=m-1;}
        return n;
}
```

lineare nel valore di m, esponenziale nella sua dimensione.

Comportamento asintotico: nell'analizzare la complessità del tempo di un algoritmo, siamo interessati a come aumenta il tempo a crescere della taglia n dell'iput. Siccome per valori "piccoli" di n il tempo richiesto è comunque poco, ci interessa soprattutto il comportamento per valori "grandi" di n (**comportamento asintotico**). Questo comportamento al crescere della dimensione n dei dati all'infinito, trascura tutte le costanti moltiplicative ed additive e tutti i termini di ordine inferiore, suddivide gli algoritmi in classi di complessità:

$a\,n+b$	*lineare*
$a\,n^2 + bn + c$	*quadratica*
$a\,\log_g n + h$	*logaritmica*
a^n	*esponenziale*
n^n	*esponenziale*



Supponiamo di avere, per uno stesso problema, sette algoritmi diversi con diversa complessità. Supponiamo che un passo base venga eseguito in un microsecondo (10^{-6}). Tempi di esecuzione (in secondi) dei sette algoritmi per diversi valori di n:

	n=10	n=100	n=1000	n=10^6
$\sqrt{n}$	$3*10^{-6}$	10^{-5}	$3*10^{-5}$	10^{-3}
$n+5$	$15*10^{-6}$	10^{-4}	10^{-3}	1 sec
$2*n$	$2*10^{-5}$	$2*10^{-4}$	$2*10^{-3}$	2 sec
n^2	10^{-4}	10^{-2}	1 sec	10^6 (~12gg)
n^2+n	10^{-4}	10^{-2}	1 sec	10^6 (~12gg)
n^3	10^{-3}	1 sec	10^5 (~1g)	10^{12} (~300 secoli)
2^n	10^{-3}	~$4*10^{14}$ secoli	~$3*10^{287}$ secoli	~$3*10^{301016}$ secoli

Per piccole dimensioni dell'input, osserviamo che tutti gli algoritmi hanno tempi di risposta non significativamente differenti. L'algoritmo di complessità esponenziale ha tempi di risposta ben diversi da quelli degli altri algoritmi (migliaia di miliardi di secoli contro secondi, ecc...).
Per grandi dimensioni dell'input (n=10^6), i sette algoritmi si partizionano nettamente in cinque classi in base ai tempi di risposta:

Algoritmo rad(n)	*frazioni di secondo*
*Algoritmo n+5, 2*n*	*secondi*
Algoritmo n^2, n^{2+n}	*giorni*
Algoritmo n^3	*secoli*
Algoritmo 2^n	*miliardi di secoli*

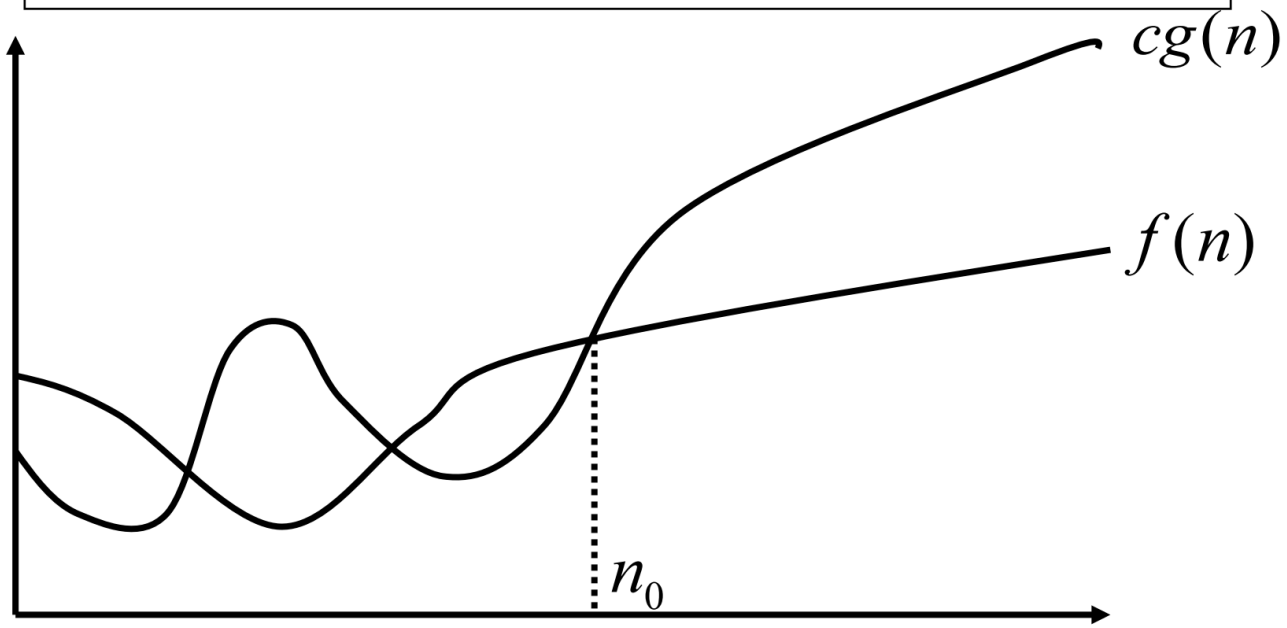
Notazione O ed Ω:

- f e g funzioni dai naturali ai reali positivi.
- f(n) è O di g(n), f(n)∈O(g(n)), se esistono due costanti positive c ed n0 tali che se n ≥ nO' f(n) ≤ c * g(n). Applicata alla funzione di complessità f(n), la notazione O ne limita superiormente la crescita e fornisce quindi una indicazione della bontà dell'algoritmo.
- f(n) è Omega di g(n), f(n)∈ Ω(g(n)), se esistono due costanti positive c ed nO tali che se n≥

nO' c*g(n)<) ì= f(n). La notazione Ω limita inferiormente la complessità, indicando così che il comportamento dell'algoritmo non è migliore di un comportamento assegnato.

Notazione asintotica O(limite superiore asintotico)

$$O(g(n)) = \{f(n) : \text{esistono } c > 0 \text{ ed } n_0 \text{ tali che } 0 \le f(n) \le cg(n) \text{ per ogni } n \ge n_0\}$$



ES:

$$f(n) = 2n^2 + 5n + 5 = O(n^2)$$

infatti $0 \le 2n^2 + 5n + 5 \le cn^2$

per $c = 4$ ed $n_0 = 5$

Vedremo che in generale per $a_2 > 0$

$$f(n) = a_2n^2 + a_1n + a_0 = O(n^2)$$

$$f(n) = 2 + \sin n = O(1)$$

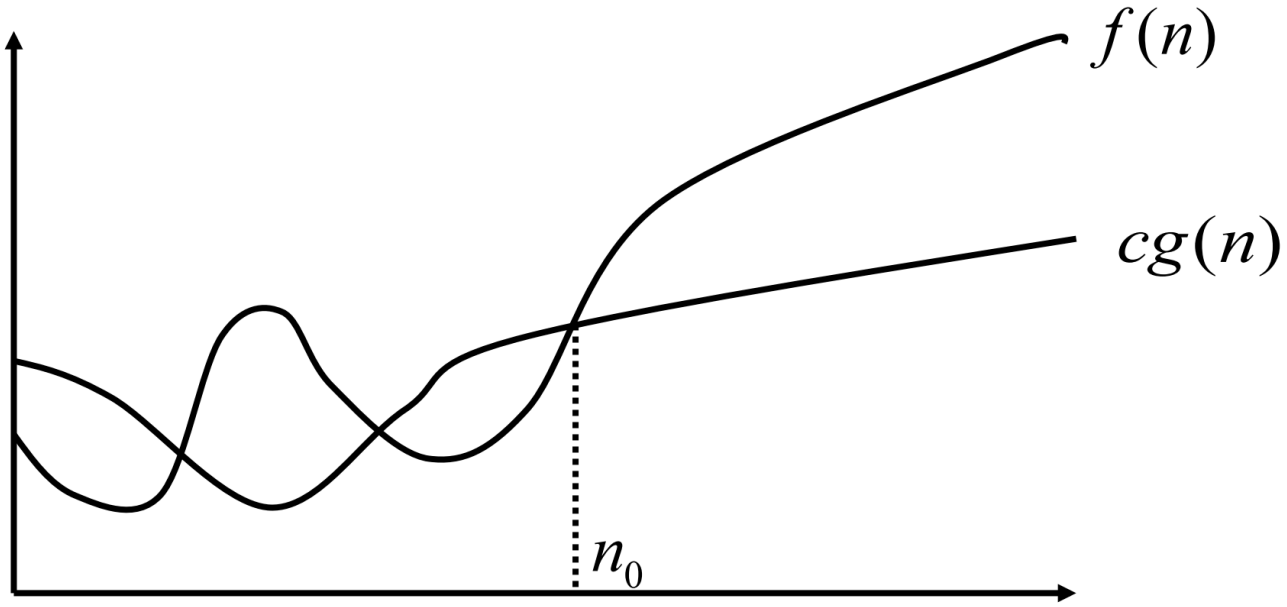
infatti $0 \le 2 + \sin n \le c \cdot 1$

per $c = 3$ ed $n_0 = 1$

Limiti superiori O(g(n)): scoprire un algoritmo per la risoluzione di un problema equivale a stabilire un limite superiore di complessità, il problema è risolubile entro i limiti di tempo stabiliti dalla funzione di complessità. Suddivisione di algoritmi in classi di complessità, algoritmi diversi per la risoluzione dello stesso problema possono avere diversa complessità e saranno confrontati sulla base della complessità asintotica nel caso medio o pessimo.

Notazione asintotica Ω (limite inferiore asintotico)

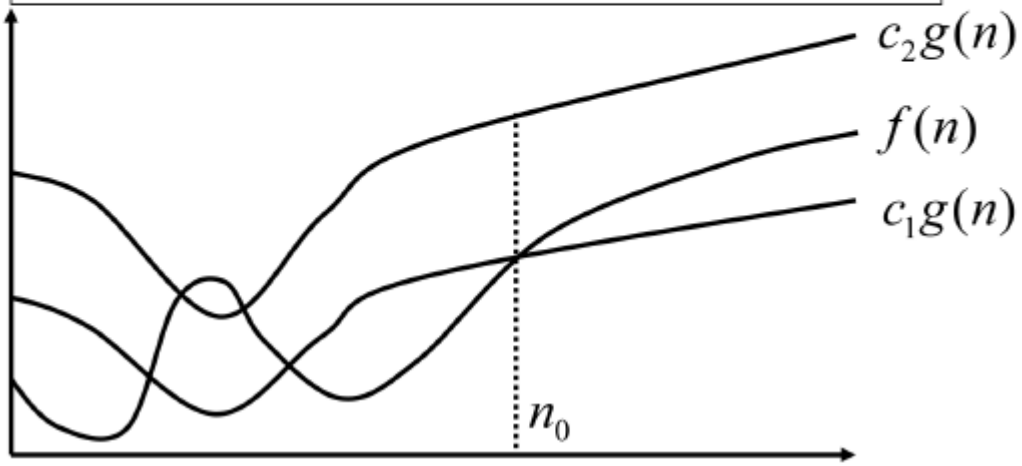
$$\Omega(g(n)) = \{f(n) : \text{esistono } c > 0 \text{ ed } n_0 \text{ tali che } f(n) \geq cg(n) \geq 0 \text{ per ogni } n \geq n_0\}$$



La ricerca di limiti inferiori di complessità risponde alla domanda se non si possano determinare algoritmi più efficienti di quelli noti. Quando la complessità di un algoritmo è pari al limite inferiore di complessità determinato per un problema, l'algoritmo si dice ottimo (in ordine di grandezza). Non esiste una teoria generale all'individuazione di limiti inferiori alla complessità dei problemi.

Notazione asintotica Θ(limite asintotico stretto)

$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n)) = \{f(n) : \text{esistono } c_1, c_2 > 0 \text{ ed } n_0 \text{ tali che per ogni } n \geq n_0 \; 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$



Alcune tecniche per l'individuazione di limiti inferiori

- **Dimensione n dei dati:** se nel caso pessimo occorre analizzare tutti i dati allora Ω(n) è un limite inferiore alla complessità del problema (esempio: ricerca di un elemento o del massimo in un array). È una tecnica banale, la maggior parte dei problemi hanno limiti inferiori più alti.
- **Eventi contabili:** se la ripetizione di un evento un dato numero di volte è essenziale per la risoluzione di un problema. *ES:* nella generazione delle permutazioni di n lettere l'evento è la generazione di una nuova permutazione che si ripete per tutte le permutazioni, ossia n! volte. **Nota: n!**

Regole per la valutazione della complessità

Scomposizione:

- alg è la sequenza di alg1 ed alg2.
- Alg1 è O(g1(n)); alg2 è O(g2(n)).
- alg è O(max(g1(n),g2(n))).

ES:

```
i=0;
while(i<n) {                 g1(n)
  Stampastelle(i);
  i=i+1;
}
for(i=0; i<2*n; i++)         g2(n)
  scanf("%d", &numero);
```

Blocchi annidati:

- alg è composto da due blocchi annidati.
- Blocco esterno è O(g1(n)); blocco interno è O(g2(n)).
- alg è O(g1(n)*g2(n)).

```
for(i=0; i<n; i++) {                 g1(n)
  scanf("%d", &j);
  printf("%d", j*j);
  do {                               g2(n)
    scanf("%d", &numero);
    j=j+1;
  } while (j<=n);
```

ES:

- Prodotto di matrici

```
float A[N][M], B[M][P], C[N][P];
int i, j, k;

for(i=0; i<N; i++)
  for(j=0; j<P; j++) {
    C[i][j]=0;
    for(k=0; k<M; k++)
      C[i][j]+=A[i][k] * B[k][j];
  }
```

O(N) { O(P) { O(M) }

- Complessità asintotica del programma: O(N*P*M).

Sottoprogrammi ripetuti: alg applica ripetutamente un certo insieme di istruzioni la cui complessità all'i-esima esecuzione vale $f_i(n)$; il numero di ripetizioni è g(n).

$$\text{alg è } O\left(\sum_{i=1}^{g(n)} f_i(n)\right)$$

per $f_i(n)$ tutte uguali ... O(g(n) f(n))

Operazione dominante: sia f(n) il costo di esecuzione di un algoritmo alg; un'istruzione *i* è dominante se viene eseguita g(n) volte, con $f(n) \leq a^* g(n)$. Se un algoritmo ha un operazione domiate allora è O(g(n)).

ES:

Ricerca binaria

```
int ricerca(int v [], int size, int k)
{ int inf =0, sup = size-1;
   while (sup >=inf)
   {  int med = (sup + inf ) / 2;
      if (k==v[med])
            return med;
      else if (k>v[med])
            inf = med+1;
      else sup = med-1
   }
   return -1;
}
```

- **Istruzioni dominanti:**
  sup ≥ inf
  med = (sup + inf) / 2
- **Sequenza di elementi utili da considerare**: n, n/2, n/4, … $n/2_i$
- **Terminazione**: numero elementi pari ad 1, ossia $n/2_i=1$
- $i=\log_2 n$, ossia l'algoritmo è O(log n).

Ricorsione e valutazione della complessità
Negli algoritmi ricorsivi la soluzione di un problema si ottieme applicando lo stesso algoritmo ad uno o più sottoproblemi. La valutazione della complessità basata sulla soluzione di relazioni di ricorrenza: la funzione di complessità f(n) è definita in termini di se stessa su una dimensione inferiore dei dati.
La complessità dipende anche dal lavoro di combinazione: preparazione delle chiamate ricorsive e combinazione dei risultati ottenuti.

Relazioni di ricorrenza con lavoro costante
Il lavoro di combinazione è indipendente dalla dimensione dei dati. Relazioni lineari di ordine h
- $C(1) = c_1$, $C(2) = c_2$, … C(h) = ch
- $C(n) = a_1$ C(n-1) + a_2 C(n-2) + … ah C(n-h) + b per n > h
- la soluzione è di ordine esponenziale con n

ES: fibonacci(h=2)
- C(0) = C(1) = c
- *C(n) = C(n-1) + C(n-2) + b*

```
unsigned fib(unsigned n) {
    if (n<2) return n;
   else return fib(n-1)+fib(n-2);
}
```

Relazioni lineari di ordine h con ah = 1 e aj =0 per $1 \leq j \leq h-1$
$C(1) = c_1$, $C(2) = c_2$, … C(h) = ch
C(n) = C(n-h) + b per n > h
La soluzione è di ordine lineare con n
Es: fattoriale (h=1)
- C(0) = C(1) = C
- C(n) = C(n1) +b

```
int fact(int n) {
   if (n==0) return 1;
   else return n*fact(n-1);
}
```

Relazioni con partizione dei dati
- C(1) = c
- *C(n) = a C(n/p) + b per n > 1*
- la soluzione è di ordine: log n se a = 1 , $n^{\log} * p^a$

ES: ricerca binaria(complessità logaritmica)
- C(1) = c
- C(n) = C(n/2) + b

Ricerca binaria ricorsiva

```
int ricercaBinaria(int valore, int vettore[], int primo, int
ultimo)
{
    if (primo > ultimo) return -1;
    int mid=(primo+ultimo)/2;
    if (valore==vettore[mid])
         return mid;
    if (valore<vettore[mid])
         return ricercaBinaria(valore,vettore,primo,mid-1);
   else
         return ricercaBinaria(valore,vettore,mid+1,ultimo);
}
```

Relazioni di ricorrenza con lavoro lineare

Il lavoro di combinazione è proporzionale alla dimensione dei dati. Relazioni lineari di ordine h con ah = 1 e a_j = O per 1≤ j ≤≤ h-1:

- *C(1) = c_1, C(2) = c_2 , … C(h) = ch*
- C(n) = C(n-h) + b n + d per n > h
- *La soluzione è di ordine quadratico in n*

Relazioni con partizione dei dati

- C(1) = c
- C(n) = a C(n/p) + b n + d per n > 1
- La soluzione è di ordine

lineare	se a < p
n log n	se a = p
$n^{log} * p^a$	se a > p

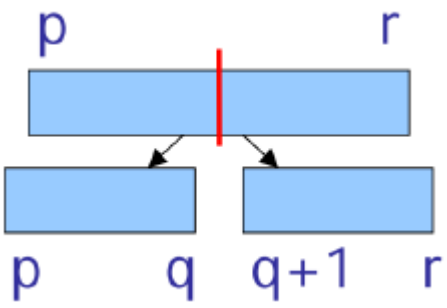
Es: Mergesort(complessità n log n)

- C(1) = c
- C(n) = 2 C(n/2) + M(n) + c
- *La complessità dell'algoritmo di Merge M(n) è O(n)*

ALTRI ALGORITMI DI ORDINAMENTO

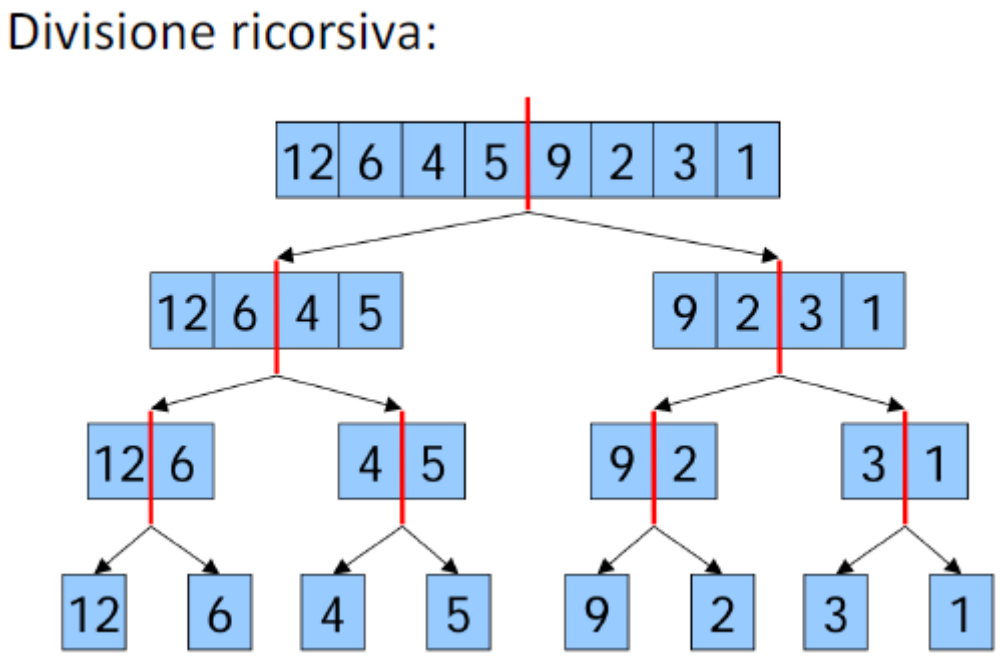
Mergesort: inventato da von Neumann nel 1945, esempio del paradigma algoritmico del divide et impera, richiede spazio ausiliario (O(N)), ed è implementato come algoritmo standard nelle librerie di alcuni linguaggio (Perl, Java). Ma è facile implementare una versione stabile. Si divide il vettore dei dati in due parti ordinate separatamente, quindi si fondono le parti per ottenere un vettore ordinato globalmente, il problema è la fusione.

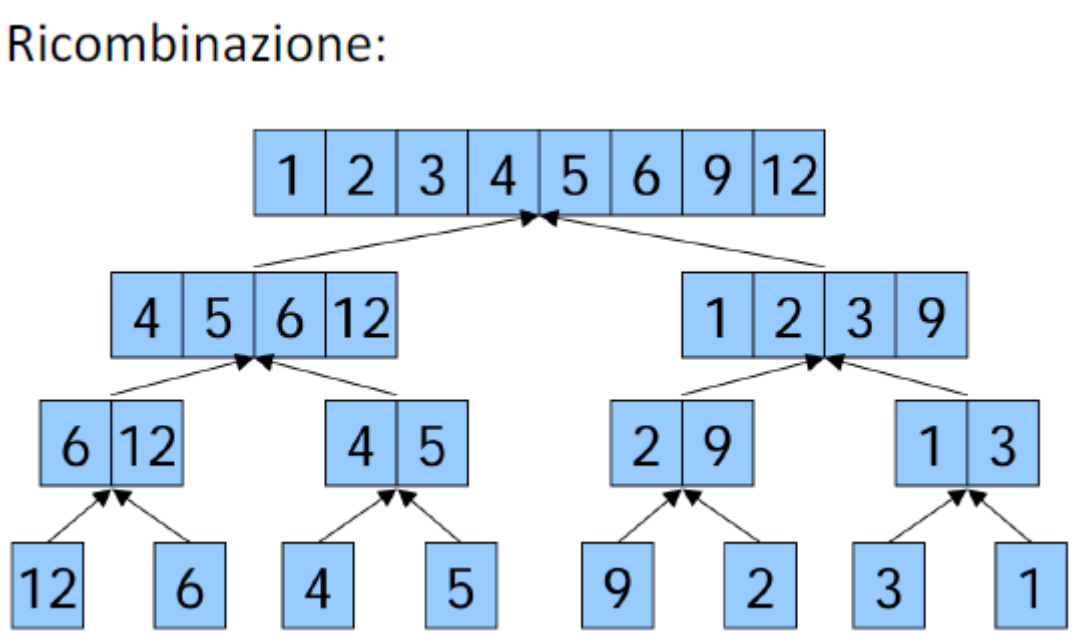
È un algoritmo ricorsivo, "divide et impera" ed è stabile. Si basa sulla divisione in due sottovettori SX e DX rispetto al centro del vettore.



- **Ricorsione:** merge sort su sottovettore DX, mergesort su sottovettore SX, termina con 1 se p = r o 0 se p > r, gli elementi sono ordinati.
- **Ricombinazione:** fondi i due sottovettori ordinati in un vettore ordinato.

ES:

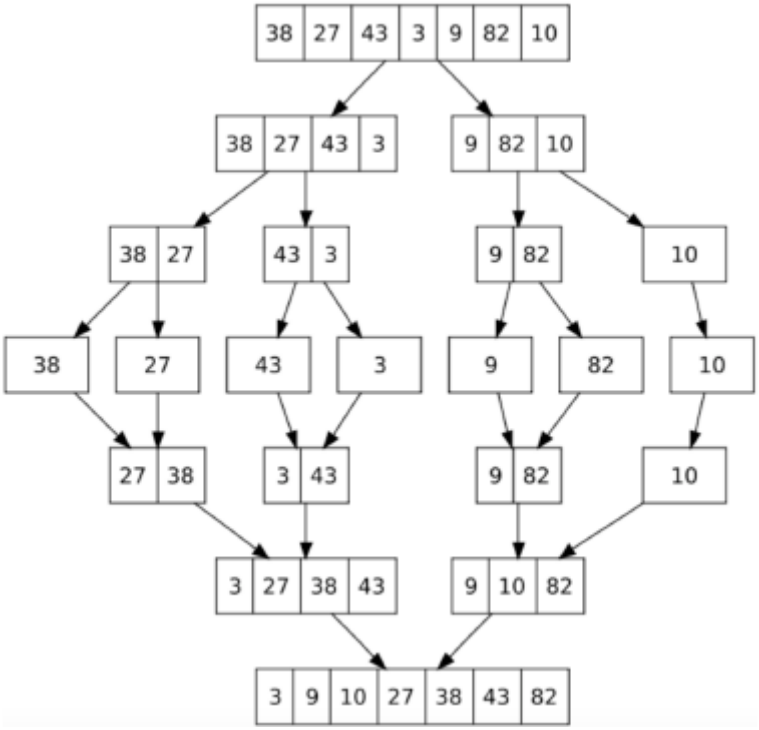




Implementazione:

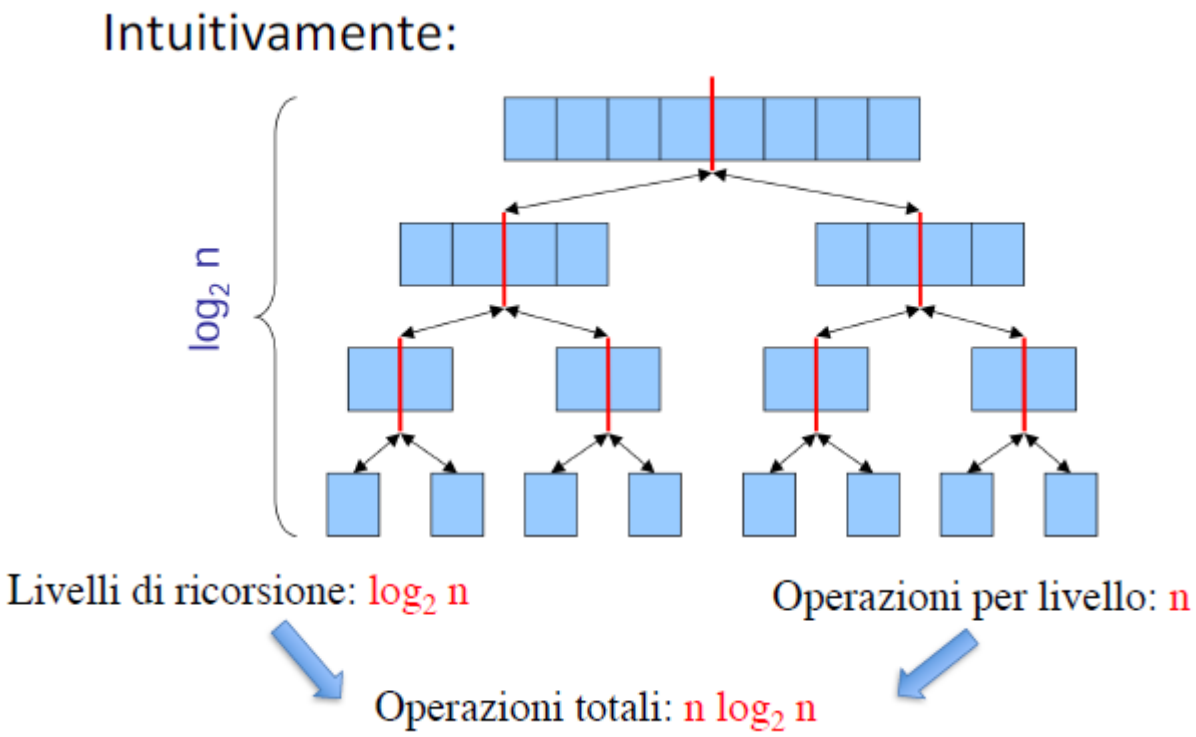
```
void MergeSort(int A[], int p, int r)
{
        int q;
        if (p < r)
         {
                q = (p + r)/2;
                MergeSort(A, p, q);
                MergeSort(A, q+1, r);
                Merge(A, p, q, r);
        }
}
```

```
void Merge(int A[], int p, int q, int r)
{
        int B[MAX], i, j, k;
        for (i=p, j=q+1, k=p; i<=q && j<=r; )
        {
                if ( A[i] < A[j] )
                {
                        B[k++] = A[i++];
                }
        else
        {
                B[k++] = A[j++];
        }
        for ( ; i<=q; ) B[k++] = A[i++];
        for ( ; j<=r; ) B[k++] = A[j++];
        for ( k=p; k<=r; k++) A[k] = B[k];
}
```



Costo:

- **Ipotesi:** $n = 2^k$
- **Dividi:** calcola la metà di un vettore
  $D(n)=\Theta(1)$
- **Risolvi:** risolve 2 sottoproblemi di dimensione n/2 ciascuno
  $2\ C(n/2)$
- **Terminazione:** semplice test
  $\Theta(1)$
- **Combina:** basata su Merge
  $C(n)=\Theta(n)$



Equazione alle ricorrenze:

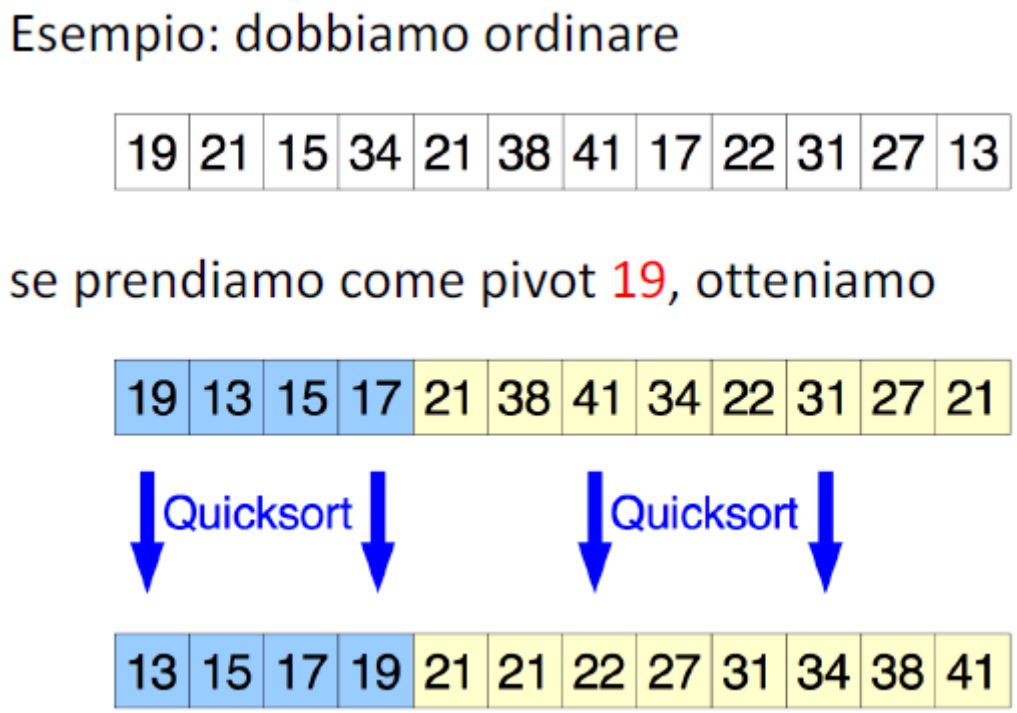
- $T(n) = 2T(n/2) + 2 \qquad n \geq 2$
- $T(1) = 1$

Soluzione:

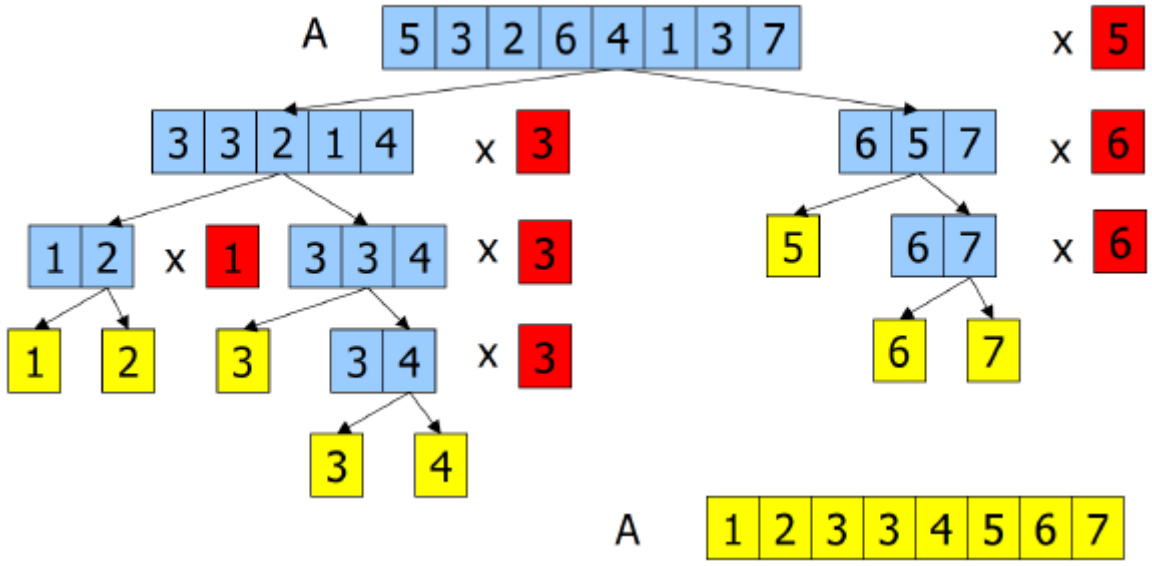
- $T(n) = \Theta(n \log n)$

Quicksort: è un algoritmo ricorsivo, di tipo "divide et impera", in loco, non è stabile. La sua idea di base per ordinare un array:

- scegli un elemento (detto **pivot**)
- metti a sinistra gli elementi ≤ **pivot**
- *metti a destra gli elementi* ≥ **pivot**
- ordina ricorsivamente la parte destra e la parte sinistra.



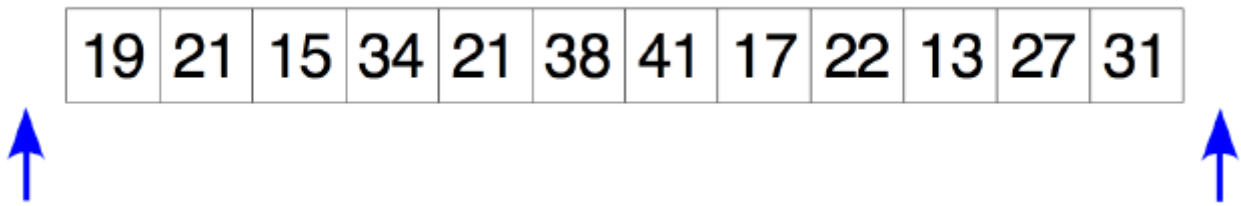
Divisione: partiziona il vettore A[p..r] in due sottovettori SX e DX. Il pivot ***x*** di Sx A[p...q] contiene tutti elementi ≤ x, mentre DX A[q+1...r] contiene elementi ≥ x, la divisione non è necessariamente a metà.
Ricorsione: si fa un quicksort su un sottovettore Sx A[p...q] , e su un sottovettore DX A[q+1...r] . Termina se il vettore ha almeno 1 elemento ordinato.
Ricombinazione: non viene fatto nulla.
ES:



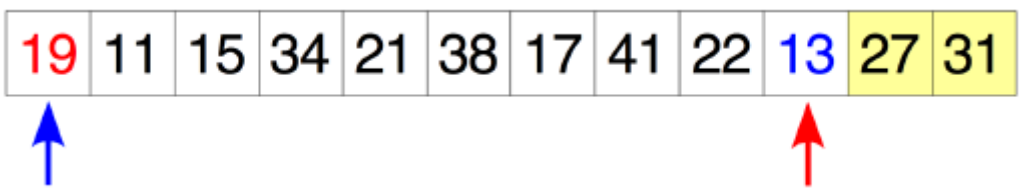
Implementazione:

```
void quicksort(int A[], int p, int r)
{
        int q;
        if(p < r)
        {
                q = partition(A, p, r);
                quicksort(A, p, q);
                quicksort(A, q+1, r);
        }
}

// per ordinare A[0]...A[n-1]
quicksort(A,0,n-1)
```

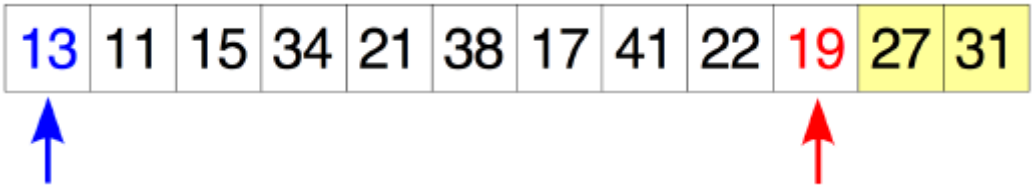
Partition: idea di base
Facciamo crescere contemporaneamente la regione con gli elementi ≤ pivot sulla sinistra, e la regione con gli elementi ≥ pivot sulla destra.
ES: pivot=17



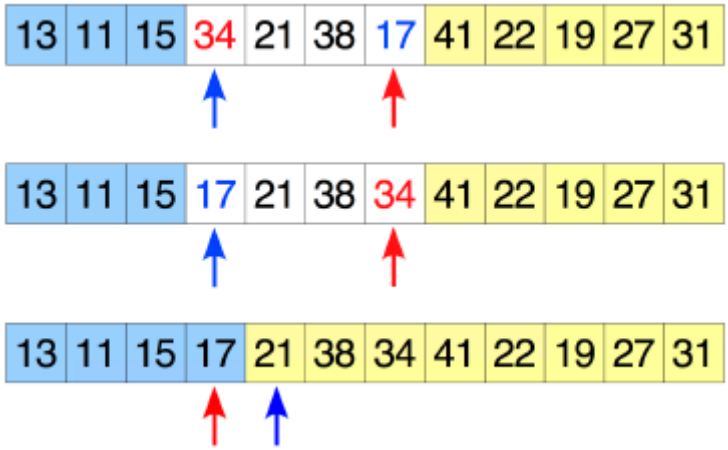
Avanziamo i due puntatori, facendo crescere le due regioni, fino a quando è possibile.



Quando entrambi i puntatori non possono avanzare scambiamo gli elementi.

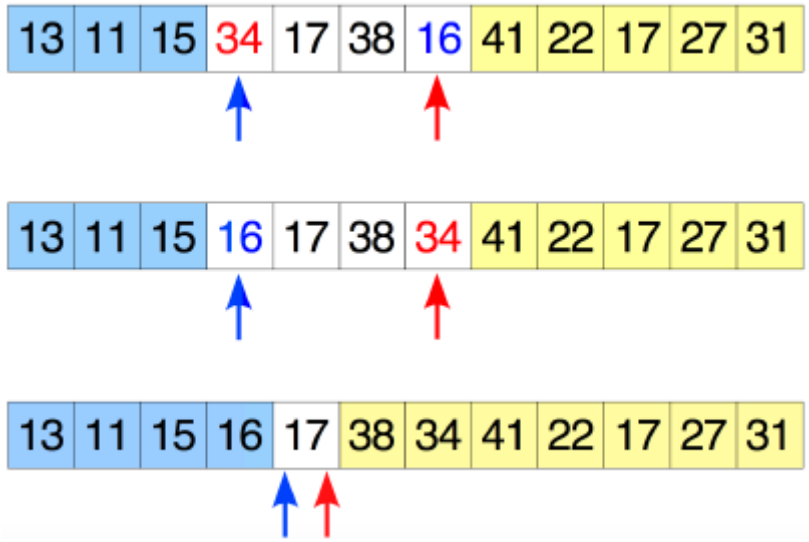


A questo punto i due puntatori possono riprendere ad avanzare.



La procedura termina quando i due puntatori si incrociano.

È possibile che i due puntatori si fermino nella stessa posizione se essa conviene il pivot.
ES: supponiamo che il pivot sia 17



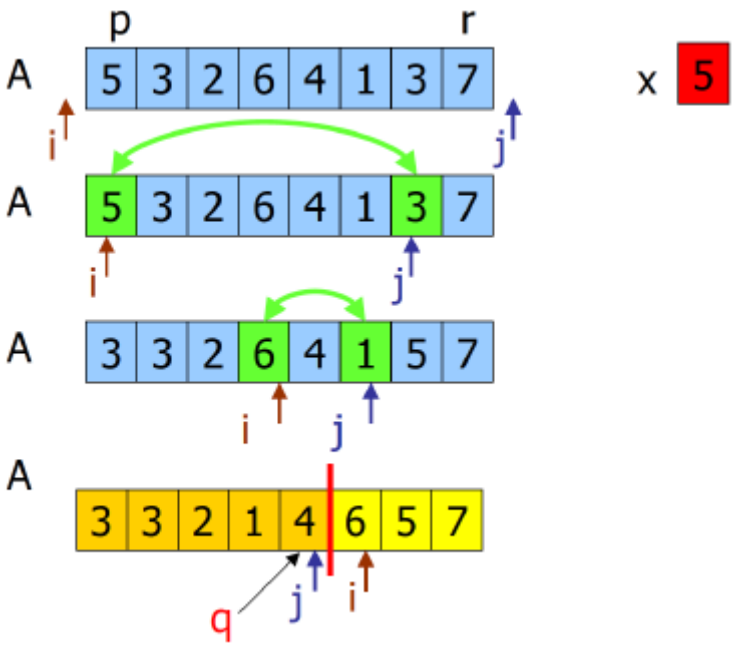
Partition: nell'implementazione della procedura *PARTITION* ci sono due scelte fondamentali da fare:

- come scegliere il pivot;
- come trattare gli elementi uguali al pivot durante la scansione.

Nella nostra prima versione prendiamo come pivot il primo elemento dell'array(vedremo che si tratta di una ***pessima scelta***), e fermiamo la scansione quando incontriamo elementi uguali al pivot.
Pivot x = A[p], individua A[i] e A[j] elementi "fuori posto", usiamo un ciclo discendente ***j*** fino a

trovare un elemento minore del pivot x, e un ciclo crescente su ***i*** fino a trovare un elemento maggiore del pivot x. Scambiamo A[i] e A[j], ripetiamo fintanto che i<j, T(n) = Θ(n).

ES:



Implementazione:

```
int partition (int A[], int p, int r)
{
        int x, i, j, temp;
        x = A[p]; // pivot
        i=p-1;
        j=r+1;

        while (i < j)
        {
                while(A[--j] > x); // esce se A[j]<=x
                while(A[++i] < x); // esce se A[i]>=x
                if(i<j) // scambia A[i] <-> A[j]
                {
                        temp = A[i]; A[i] = A[j]; A[j] = temp;
                }
        }
        return(j);
}
```

Analisi

Il quicksort ordina in loco, la sua strategia di ordinamento è simile a quella del mergesort, se non che la procedura di fusione (merge) è stata sostituita dal partizionamento. Ora analizzeremo il tempo di esecuzione dell'algoritmo quicksort. Trattandosi di un algoritmo di ordinamento basato sui confronti, analizziamo il numero di confronti in funzione del numero n di oggetti da ordinare.

Osserviamo che:

- tutti i confronti sono eseguiti nella procedura PARTITION.
- Quando PARTITION è eseguita su un sottoarray di lunghezza *m* vengono eseguiti *m* confronti (ogni elemento è confrontato una volta con il pivot).

Se l'array da ordinare viene partizionato in due array di dimensione *r* e *n-r* abbiamo che il numero di confronti *C(n)* soddisfa alla ricorrenza:

$$C(n) = \begin{cases} 0 & \text{se } n = 1 \\ C(r) + C(n-r) + n & \text{se } n > 1 \end{cases}$$

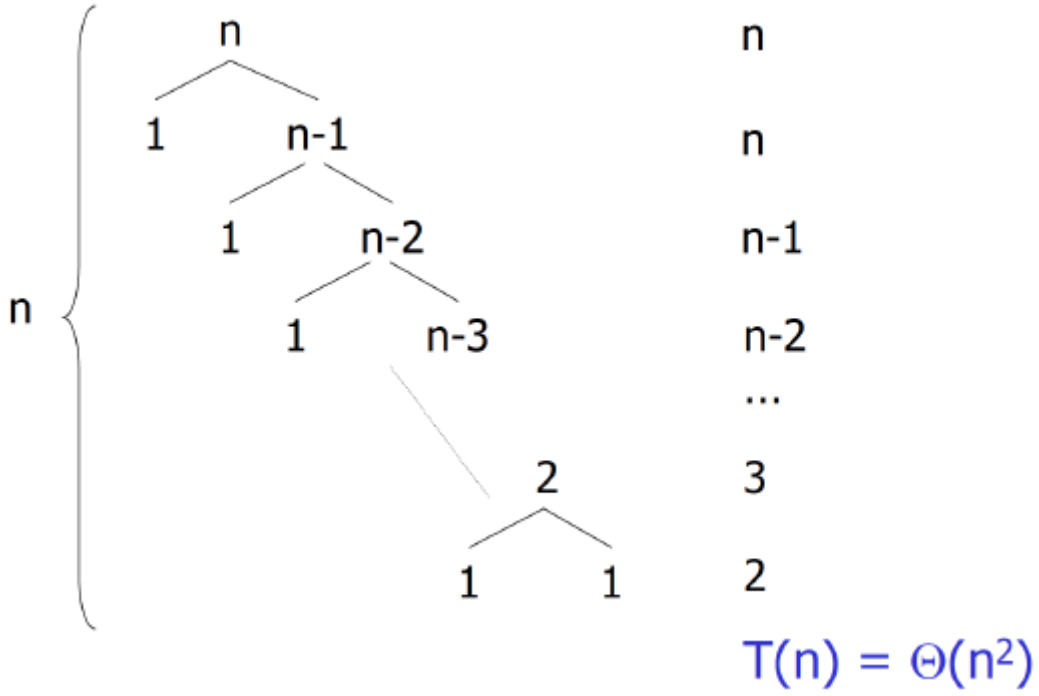
Questa ricorrenza non è del tipo che si può ricondurre ai casi presentati(anche perchè il parametro *r* cambia ad ogni chiamata della procedura!). Studieremo il caso peggiore e il caso migliore, e cercheremo di farci un'idea del caso medio.
Efficienza legata al bilanciamento delle partizioni, ad ogni passo PARTITION ritorna:

- caso peggiore un vettore da n-1 elementi e l'altro da 1.
- caso migliore due vettori da n/2 elementi.
- Caso medio due vettori di dimensioni diverse.

Il bilanciamento è quindi legato alla scelta del pivot.

Caso peggiore:
Il caso peggiore si può avvere con il pivot = al minimo o al massimo degli elementi dell'array. Sfortunatamente, con la nostra scelta del pivot il caso peggiore si presenta quando l'array è già ordinato (sia in ordine crescente che decrescente).
Equazione di ricorrenza:

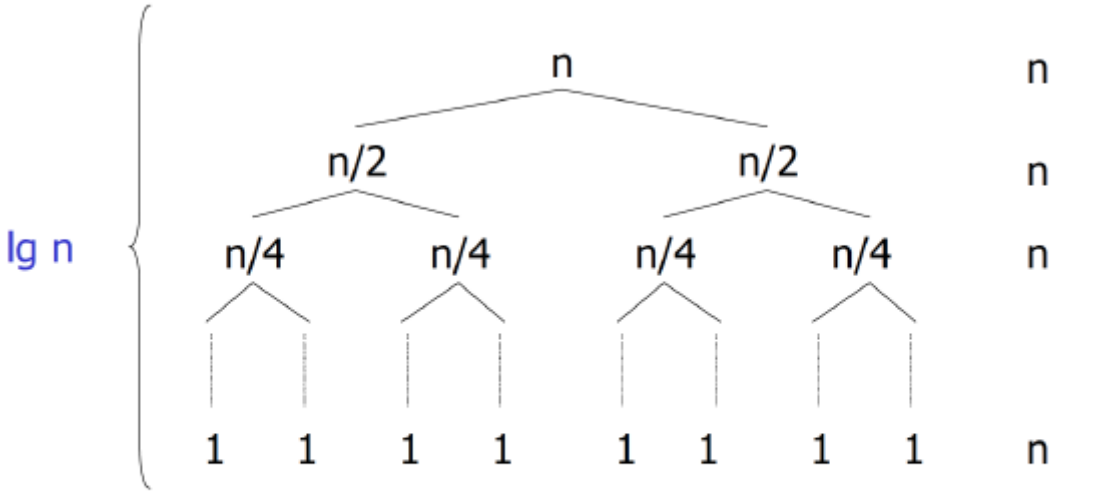
$$T(n) = T(n-1) + n \qquad n \geq 2$$
$$T(1) = 1$$
$$T(n) = \Theta(n^2)$$



Caso migliore:
Il caso migliore si verifica quando ad ogni passo l'array viene partizionato in due regioni uguali. Abbiamo quindi che:

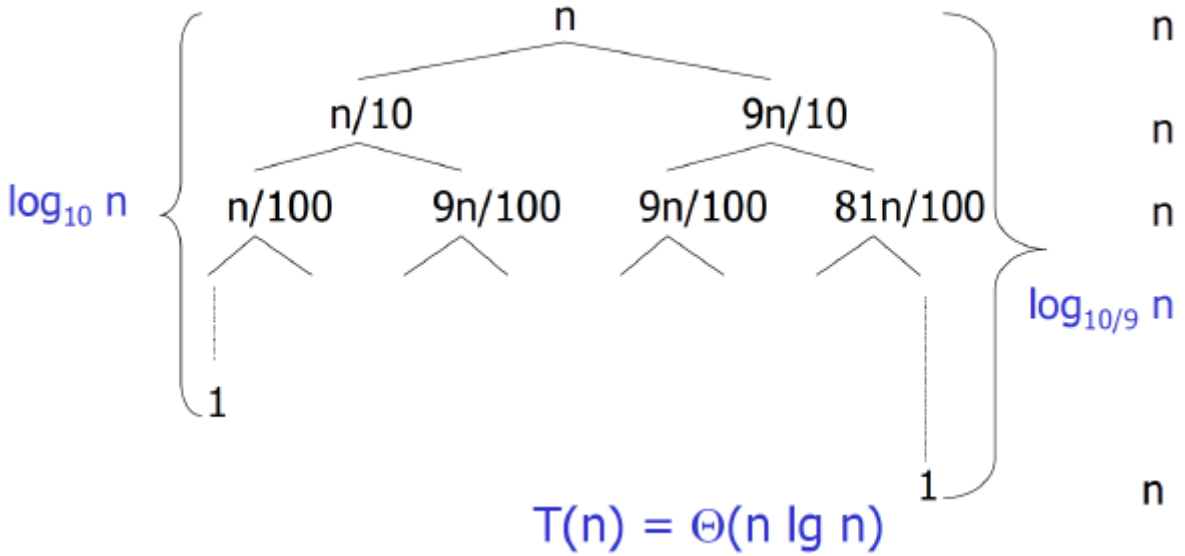
$$T(n) = 2T(n/2) + n \qquad n \geq 2$$
$$T(1) = 1$$

$$T(n) = \Theta(n \log n)$$



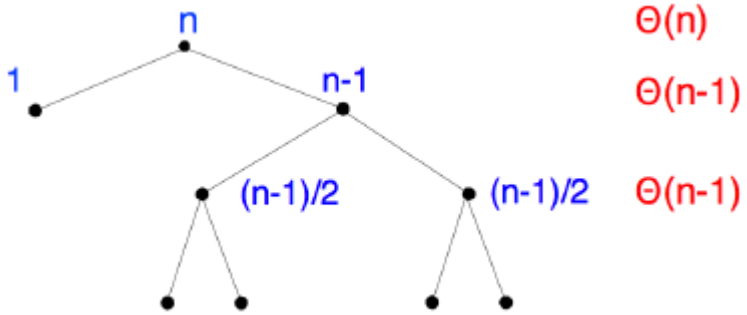
$$T(n) = \Theta(n \lg n)$$

Caso medio:
Ancora una volta il caso peggiore e il caso migliore risultano molto diversi tra loro. È naturale chiedersi se dobbiamo aspettarci più frequentemente un tempo $\Theta(n^2)$ o un tempo $\Theta(n \log n)$ o qualcosa di intermedio. Per farci un'idea sul caso medio studiamo il caso*(altamente improbabile)* che la procedura PARTITION crei sempre due regioni di cui una sia 9 volte più grande dell'altra.



$$T(n) = \Theta(n \lg n)$$

Studiamo ora il caso in cui si alternano una partizione cattiva e una buona. Nei livelli dispari il problema viene spezzato in [1. n-1], nei livelli pari in [n/2, n/2].



Per scendere di un livello nell'albero il costo è $\Theta(n)$. L'altezza dell'albero è circa 2 log n quindi il costo complessivo è $\Theta(n \log n)$.

Riassumendo
Se le partizioni sono spesso molto sbilanciate, il costo può risultare molto alto. Se invece non sono troppo sbilanciate si ha un costo $\Theta(n \log n)$ cioè asintoticamente uguale al caso ottimo.

Random pivoting
Il random pivot, è un numero *n* generato casualmente con $p \leq i \leq r$, poi scambia A[1] e A[n], usando come pivot A[1]. Si può dimostrare matematicamente che il tempo di esecuzione di **quicksort** con pivot casuale è $\Theta(n \log n)$ con probabilità molto vicina ad 1. In altre parole, devo essere molto sfortunato per avere un tempo di esecuzione asintoticamente superiore a $\Theta(n \log n)$.

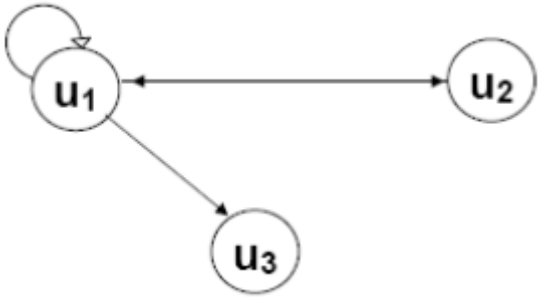
Pivot medio di 3
La generazione dei numeri casuali rallenta la procedura PARTITION e quindi tutto l'algoritmo quicksort. Per questo motivo sono state proposte altre strategie di selezione del pivot che in pratica risultano leggermente più veloci. La strategia *medio di 3* consiste nel considerare gli elementi che sono nella prima e ultima posizione dell'array, e l'elemento che si trova in posizione mediana.
ES:

39	15	34	21	38	18	47	22	13	27	31

Di questi tre valori viene preso quello intermedio (nell'esempio qui sopra 31).

ALBERO BINARIO

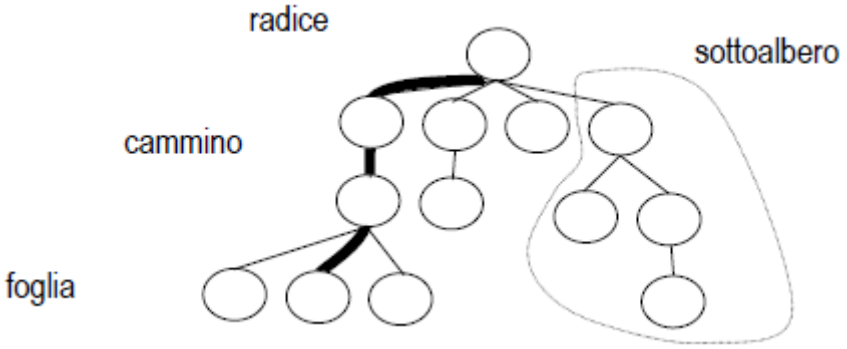
I grafi: un grafo orientato G è una coppia <N,A> dove N è un insieme finito non vuoto(insieme di nodi) e a $\subseteq$ NxN è un insieme finito di coppie ordinate di nodi, detti archi (o spigoli o linee).

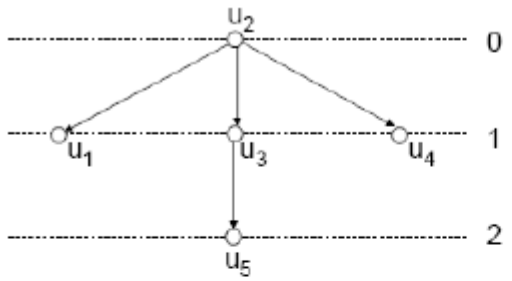


Se $\langle u_i, u_j \rangle \in A$ nel grafo vi è un arco da u_i ad u_j
Nell'esempio
- $N = \{u_1, u_2, u_3\}$,
- $A = \{(u_1,u_1),(u_1,u_2),(u_2,u_1),(u_1,u_3)\}$.

Gli alberi: il grafo è una struttura dati alla quale si possono ricondurre strutture più sempici come *liste* ed *alberi.* L'albero è una struttura informativa per rappresentare partizioni successive di un insieme in sottoinsiemi disgiunti, sono organizzazioni gerarchiche di dati e procedimenti decisionali enumerativi. Per esempio un file system di un sistema operativo è un grafo. In un albero ogni nodo ha un unico arco entrante, tranne un nodo particolare chiamato ***radice***, che non ha archi entranti.

Ogni nodo può avere zero o più archi uscenti, i nodi senza archi uscenti sono detti foglie. Un arco nell'albero induce una relazione padre-figlio, ed a ciascuno nodo è solitamente associato un valore, detto etichetta del nodo.





ALBERO RADICATO ORDINATO

LA RADICE r E' A LIVELLO 0 E TUTTI I NODI u, ∋' < u, r > ∈ A, SONO FIGLI DI r E STANNO A LIVELLO 1 (r E' PADRE). NODI CON LO STESSO PADRE SONO FRATELLI. NODI TERMINALI SENZA FIGLI SONO DETTI FOGLIE.

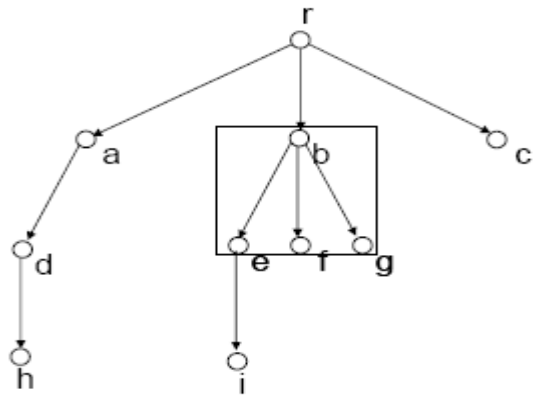
UN ALBERO ORDINATO E' OTTENUTO DA UNO RADICATO STABILENDO UN ORDINAMENTO TRA NODI ALLO STESSO LIVELLO.

Di seguito sono elencati alcuni concetti appartenenti ad un albero:

- **Grado di un nodo:** numero di figli del nodo.
- **Cammino:** sequenza di nodi $<n_0,n_1,...,n_k>$ dove il nodo n_i è padre del nodo n_i+1, per $0 \leq i < k$, la lungehzza del cammino è k, e dato un nodo, esiste un unico cammino dalla radice dell'albero al nodo.
- **Livello di un nodo:** lunghezza del cammino dalla radice al nodo, la sua definizione ricorsiva è: il livello della radice è 0, il livello di un nodo non radice è 1+ il livello del padre.
- **Altezza dell'albero:** la lunghezza del più lungo cammino nell'albero, parte dalla radice e termina in una foglia.

Si definisce un albero di ordine K un albero in cui ogni nodo ha al massimo k figli.

ES:



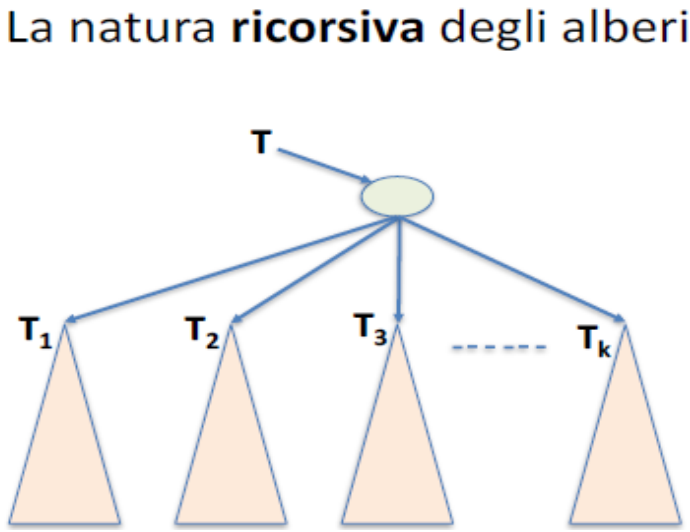
ALBERO TERNARIO

Un'albero ha diverse proprietà:

- è un grafo aciclico, in cui per ogni nodo esiste un solo arco entrante (tranne che per la radice che non ne ha nessuno).
- è un grafo debolmente connesso.
- se esiste un cammino che va da un nodo u ad un altro nodo v, tale cammino è unico.
- in un albero esiste un solo cammino che va dalla radice a qualunque altro nodo.
- tutti i nodi di un albero t (esclusa la radice) possono essere ripartiti in insiemi disgiunti, ciascuno dei quali individua un albero: dato un nodo u, i suoi discedenti costituiscono un albero detto sottoalbero di radice *u*.

Un albero per sua natura è di tipo ricorsivo, infatti può essere definito ricorsivamente. Infatti un albero è un'insieme di nodi ai quali sono associate delle informazioni, tra tutti i nodi ne esiste uno particolare denominato radice (livello 0). Gli altri nodi sono partizionati in sottoinsiemi che sono a loro volta alberi (livelli successivi):

- vuoto o costituito da un solo nodo (detto radice).
- oppure è una radice cui sono connessi altri alberi.



ALBERI BINARI

Gli alberi binari sono un particolare tipo di alberi ordinati, in cui ogni nodo ha al più due figli e si fa sempre distinzione tra il figlio sinistro, che viene prima nell'ordinamento e il figlio destro. Nell'esempio seguente gli alberi sono etichettati con interi e con caratteri:

Un albero binario è un grafo orientato che o è vuoto o è costitutio da un solo nodo o è formato da un nodo n (detto radice) e da due sottoalberi binari, chiamati rispettivamente sottoalbero (o figlio) sinistro e sottoalbero (o figlio) destro. Gli alberi binari sono particolari alberi n-ari con caratteristiche molto importanti:

- Ogni nodo può avere al più due figli: sottoalbero sinistro e sottoalbero destro.
- Definizione ricorsiva: un albero binario è vuoto oppure è una terna (s, r, d), dove r è il nodo radice, *s* e *d* sono alberi binari.

- Le funzioni degli alberi binari semplici sono:
    1. Costruttore bottom-up.
    2. Operatore di selezione.
    3. Operatori di visita.

Specifica sintattica

- ***TIPI***: ALBEROBIN, BOOLEAN, NODO, ITEM
- ***OPERATORI***:
    I. *CREABINALB :* *( ) → ALBEROBIN*
    II. ALBVUOTO : *(ALBEROBIN) → BOOLEAN*
    III. RADICE : *(ALBEROBIN) → NODO*
    IV. FIGLIOSX : *(ALBEROBIN) → ALBEROBIN*
    V. FIGLIODX : *(ALBEROBIN) → ALBEROBIN*
    VI. COSTRBINALB : *(ITEM, ALBEROBIN, ALBEROBIN) → ALBEROBIN*

Specifica semantica

- **TIPI**:
    I. ALBEROBIN = insieme degli alberi binari, dove: $\Lambda \in$ ALBEROBIN (albero vuoto) se N $\in$ NODO, T1 e T2 $\in$ ALBEROBIN allora <N, T1, T2> $\in$ ALBEROBIN
    II. BOOLEAN = {vero, falso}
    III. NODO è un qualsiasi insieme non vuoto
    IV. ITEM è un qualsiasi insieme non vuoto
- **OPERATORI**:
    I. CREABINALB ( ) = T
    pre:
    post: T = Λ
    II. ALBVUOTO (T) = v
    pre:
    post: se T e vuoto, allora v = vero, altrimenti v = falso
    III. RADICE (T) = N'
    pre: T = <N, Tsx, Tdx> non e l'albero vuoto
    post: N = N'
    IV. FIGLIOSX (T) = T'
    pre: T = <N, Tsx, Tdx> non e l'albero vuoto
    post: T' = Tsx
    V. FIGLIODX (T) = T'
    pre: T = <N, Tsx, Tdx> non e l'albero vuoto
    post: T' = Tdx
    VI. COSTRBINALB (elem, T1, T2) = T'
    pre:
    post: T' = <N, T1, T2> N e un nodo con etichetta elem

RELAZIONE DI UN ALBERO BINARIO

La realizzazione più diffusa è tramite una struttura a puntatori con nodi doppiamente concatenati.
Ogni nodo è una struttura con 3 componenti:

- Puntatore alla radice del sottoalbero sinistro.
- Puntatore alla radice del sottoalbero destro.
- Etichetta (useremo il tipo generico ITEM per questo campo).

Un albero binario è definito come puntatore ad un nodo:

- Se l'albero binario è vuoto, puntatore nullo.
- Se l'albero binario non è vuoto, puntatore al nodo radice.

CODICE C DI UN ALBERO

Dichiarazione del tipo nodo

Per usare una lista concatenata serve una struttura che rappresenti i nodi. La struttura conterrà i dati necessari (un intero nel seguente esempio) ed un puntatore al prossimo elemento della lista:

```
struct node
 {
        itemvalue;          /* etichetta del nodo*/
        struct node *left;  /* puntatore al sottoalbero sinistro*/
        struct node *right; /* puntatore al sottoalbero destro*/
};
```

Dichiarazione del tipo Btree

Il passo successivo è quello di dichiarare il tipo Btree: *typedef struct node *Btree;*
Una variabile di tipo Btree punterà al nodo radice dell'albero. Se assegnamo a T il valore di NULL indica che l'albero è inizialmete vuoto: *Btree T = NULL;*
Un albero binario viene costruito in maniera bottom-up. Man mano che costruiamo l'albero, creiamo dei nuovi nodi da aggiungere come nodo radice. I passi per creare un nodo sono:

- Allocare la memoria necessaria.
- Memorizzare i dati nel nodo.
- Collegare il sottoalbero sinistro e il sottoalbero destro, già costruiti in precedenza.

Note sul Btree: L'insieme degli operatori così definiti costituisce l'insieme degli operatori di base (il minimo insieme di operatori) di un albero binario. Ogni altro operatore che si volesse aggiungere all'ADT albero binario potrebbe essere implementato utilizzando gli operatori dell'insieme di base. È frequente la pratica di arricchire il tipo Btree con l'aggiunta di operatori per inserire o cancellare nodi in determinate posizioni dell'albero.

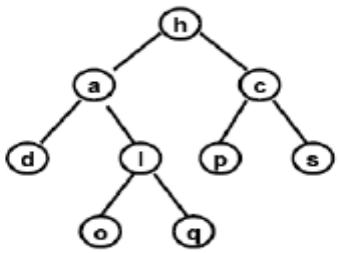
Algoritmi di visita

La visita di un albero consiste nel seguire una rotta di viaggio che consenta di esaminare ogni nodo dell'albero esattamente una volta.

- Visita in **pre-ordine:** si applica ad un albero non vuoto e richiede prima l'analisi della radice dell'albero e, poi, la visita, effettuata con lo stesso metodo, dei due sottoalberi, prima il sinistro, poi il destro.
- Visita in **post-ordine:** si applica ad un albero non vuoto e richiede prima la visita, effettuata con lo stesso metodo, dei sottoalberi, prima il sinistro e poi il destro, e, in seguito, l'analisi della radice dell'albero.
- Visita **simmetrica:** richiede prima la visita del sottoalbero sinistro (effettuata sempre con lo stesso metodo), poi l'analisi della radice, e poi la visita del sottoalbero destro.

ESEMPIO:

SIA UN ALBERO BINARIO CHE HA DEI CARATTERI NEI NODI



LA VISITA IN PREORDINE: h a d l o q c p s

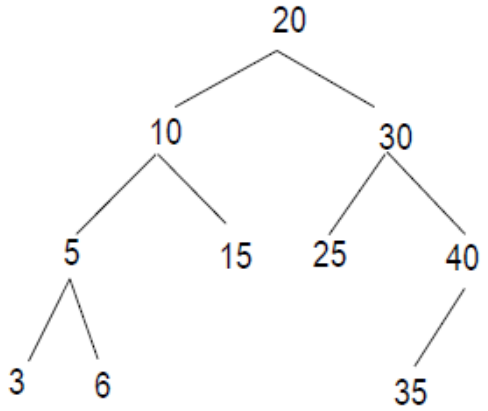
LA VISITA IN POSTORDINE: d o q l a p s c h

LA VISITA SIMMETRICA: d a o l q h p c s

ALBERI DI RICERCA BINARI

Sono utilizzati per la realizzazione di insiemi ordinati, le operazioni più efficienti sono di ricerca, inseriemento, cancellazione.

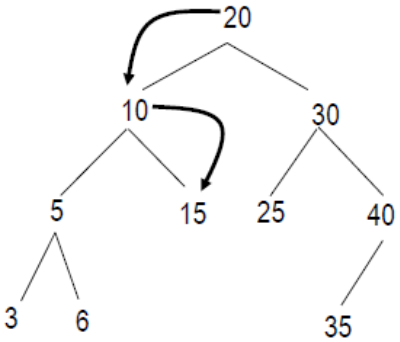
Definizione: se l'albero non è vuoto allora ogni elemento del sottoalbero di sinistra precede (<) la radice, ogni elemento del sottoalbero di destra segue (>) la radice, i sottoalberi sinistro e destro sono alberi di ricerca binaria.



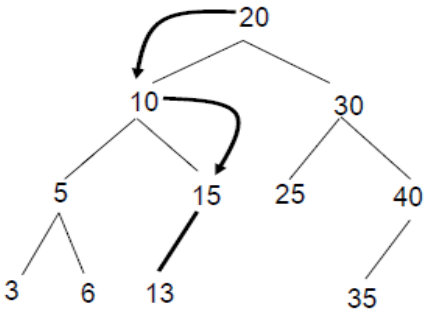
Operazioni:

- **Contains (**ricerca di un elemento): se l'albero è vuoto allora restituisce *false*, se l'elemento cercato coincide con la radice dell'albero restituisce *true*, se l'elemento cercato è minore della radice restituisce il risultato della ricerca dell'elemento nel sottoalbero sinistro, se l'elemento cercato è maggiore della radice restituisce il risultato della ricerca dell'elemento nel sottoalbero destro.

Esempio: ricerca di 15

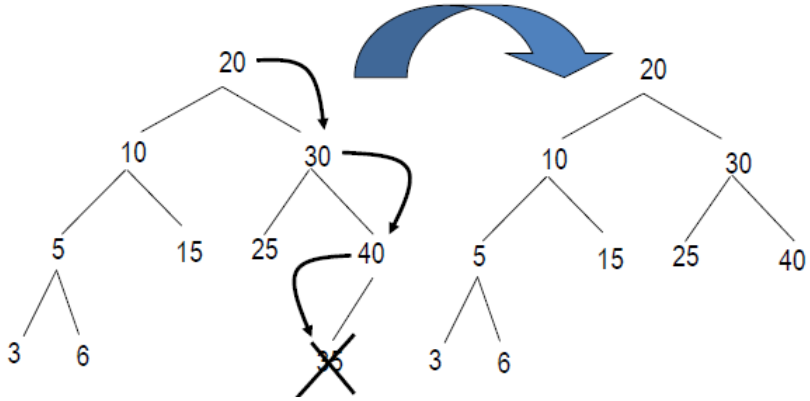


- **Insert (**Inserimento di un elemento): se l'albero è vuoto allora crea un nuovo albero con un solo elemento, se l'albero non è vuoto si possono verificare 3 casi:
  1. Se l'elemento coincide con la radice non fa niente.
  2. Se l'elemento è minore della radice allora lo inserisce nel sottoalbero sinistro.
  3. Se l'elemento è maggiore della radice allora lo inserisce nel sottoalbero destro.

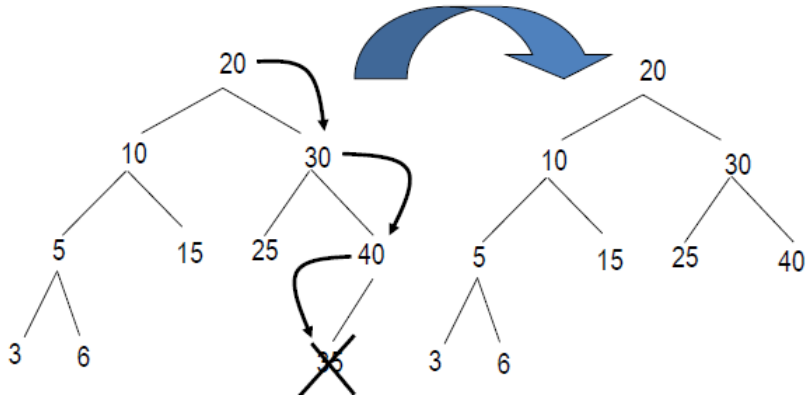


- **Delete**(cancellazione di un elemento e): nessun problema se il nodo è una foglia, se il nodo invece ha un solo sotto albero di radice *r* si possono verificare i 3 seguenti casi:
    1. Se il nodo da rimuovere è la radice allora r prende il suo posto (diventa radice dell'albero).
    2. Se il nodo ha un padre *p*, allora viene rimosso e *r* prende il suo posto (diventa figlio di *p*).

ES: eliminazione di 35

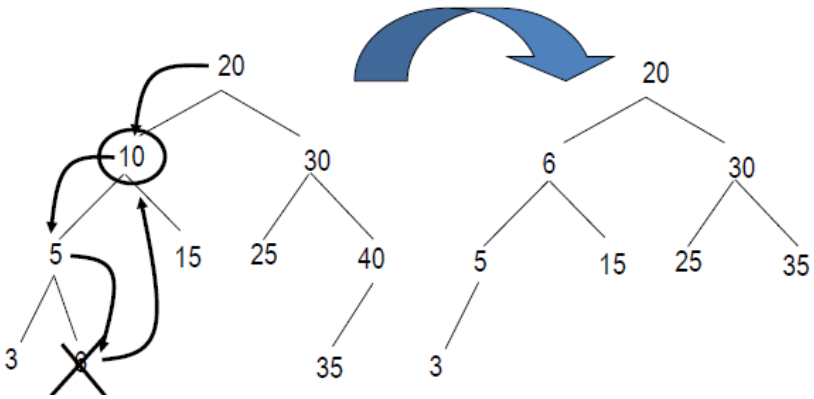


ES: eliminazione di 40



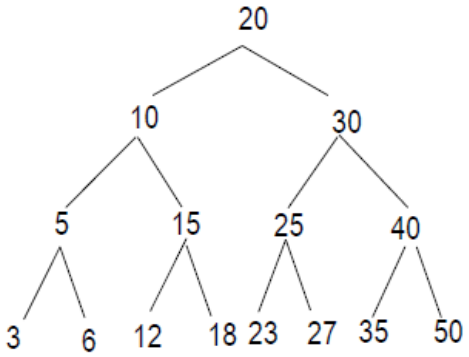
3. Se il nodo ha entrambi i sotto alberi, si cerca l'elemento max nel sottoalbero sinistro (da notare che tale elemento non ha sottoalbero destro, la cui radice altrimenti sarebbe maggiore), alternativamente si cerca l'elemento minimo nel sottoalbero destro. Il nodo contenente l'elemento max viene eliminato, mentre a quello contenente l'elemento da eliminare si assegna max. L'albero risultante è un albero di ricerca binaria.

ES: eliminazione di 10



ALBERI PERFETTAMENTE BILANCIATI
E ALBERI Δ BILANCIATI

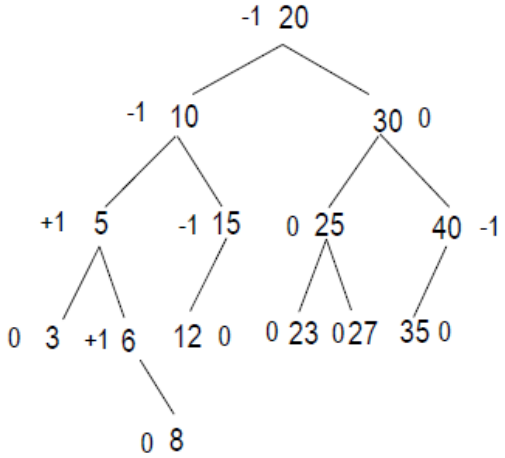
Le operazioni sull'albero di ricerca binaria hanno complessità logaritmica se l'albero è (permettamente) bilaciato. In un albero bilanciato tutti i nodi interni hanno entrambi i sottoalberi e le foglie sono a livello massimo, se l'albero ha *n* nodi l'altezza dell'albero è $\log_2 n$. Un albero di ricerca binaria si dice Δ bilanciato se per ogni nodo accade che la differenza (in valore assoluto) tra le altezze dei suoi due sottoalberi è minore o uguale a Δ, si può dimostrare che l'altezza dell'albero è $\Delta + \log_2 n$.



ALBERI AVL

Per Δ = 1 si parla di alberi AVL (dal nome dei suoi ideatori *Adel'son, Vel'skii e Landis*). Sono usati per prevenire il non bilanciamento ad ogni nodo bisogna aggiungere un indicatore che può assumere i seguenti valori:

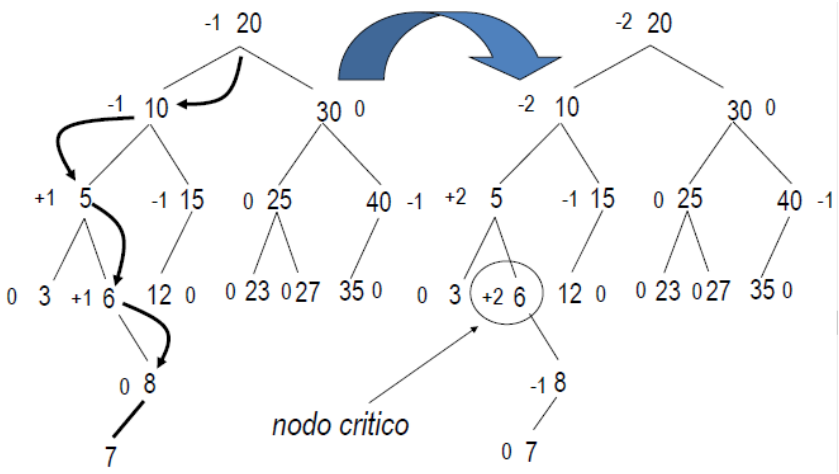
- -1: se l'altezza del sottoalbero sinistro è maggiore di 1 dell'altezza del sottoalbero destro.
- 0: se l'altezza del sottoalbero sinistro è uguale all'altezza del sottoalbero destro.
- +1: se l'altezza del sottoalbero sinistro è minore di 1 dell' altezza del sottoalbero destro.



Ribilanciamento di alberi AVL

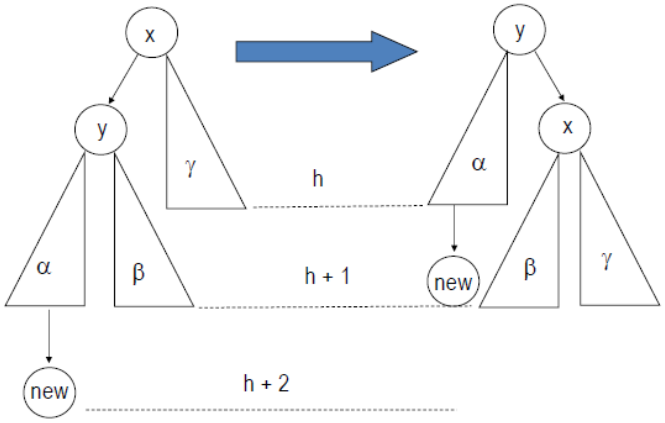
Un inserimento di una foglia può provacare uno sbilanciamento dell'albero, se per almeno uno dei nodi l'indicatore non rispetta più uno dei tre stati precedente, in tal caso bisogna ribilanciare l'albero con operazioni di rotazione (semplice o doppia) agendo sul nodo x a profondità massima che presenta un non bilanciamento, tale nodo viene detto nodo critico e si trova sul percorso che va dalla radice al nodo foglia inserito. Considerazioni simili si possono fare anche per la rimozione di un nodo.

ES: sbilanciamento con inserimento di 7

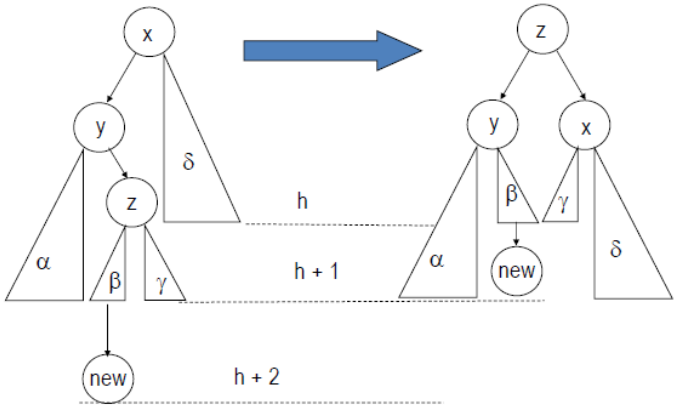


ES: di bilanciamento di alberi AVL:

rotazione semplice



rotazione doppia



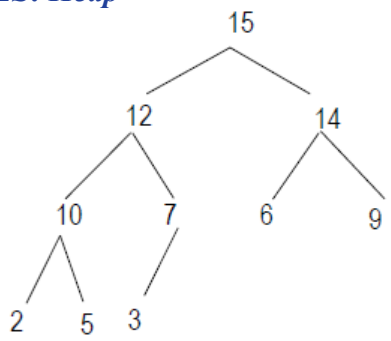
HEAP

È un albero quasi perfettamente bilanciato di altezza h, è un albero perfettamente bilanciato fino a livello h-1. Un heap è un albero binario quasi perfettamente bilanciato con le seguenti proprietà:

- Le foglie a livello h sono tutte addossate a sinistra.
- Ogni nodo *v* ha la caratteristica che l'informazione ad esso associata è la più grande tra tutte le informazioni presenti nel sotto albero che ha *v* come radice.

È utilizzato per realizzare le code a priorità, le operazioni sono inserimento di un elmento e rimozione del max. Per le sue caratteristiche un heap può essere realizzato con un array, i nodi sono disposti nell'array per livelli. La radice occupa la posizione 0, se un nodo occupa la posizione i, il suo figlio sinistro occupa la posizione 2*i+1 e il suo figlio destro ccupa la posizione 2*i+2.

ES: Heap

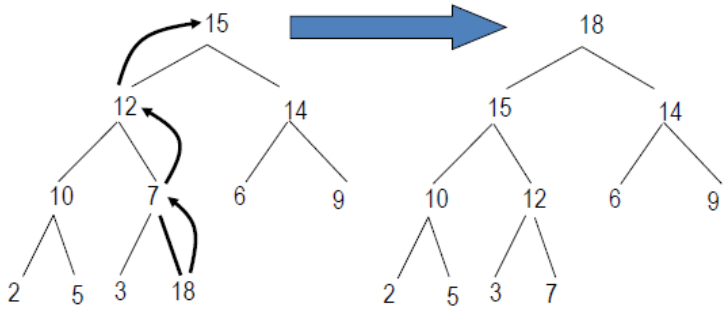


rappresentazione

15	12	14	10	7	6	9	2	5	3

Operazioni:

- **Inserimento:** si inserisce il nuovo nodo come ultima foglia, il nodo inserito risale lungo il percorso che porta alla radice per individuare la posizione gusta.
  ES: inserimento di 18



- **Rimozione:** si rimuove sempre la radice (l'informazione maggiore, in quanto un heap viene usato per realizzare code a priorità) e si pone l'ultima foglia al posto della radice. Si scambia l'informazione contenuta nella radice con la maggiore dei suoi due sottoalberi e si ripete il procedimento fino ad arrivare ad una foglia.
  ES:

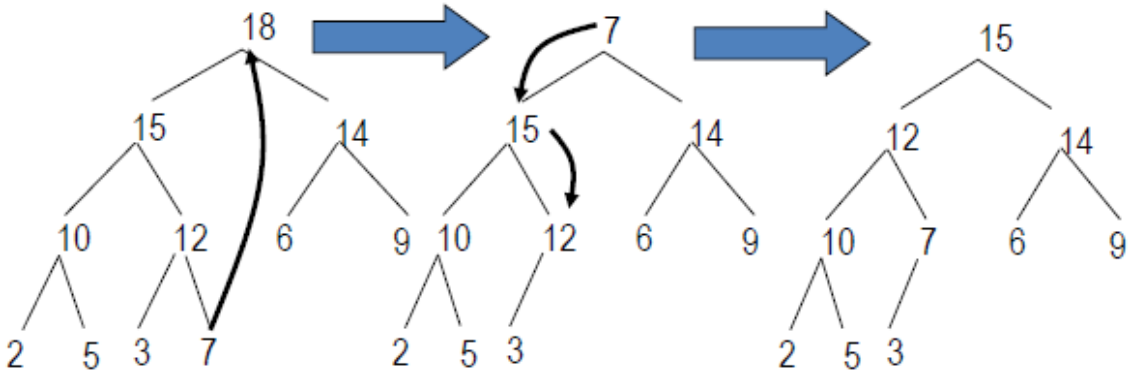


TABELLE HASH

ADT Dizionario

In molte applicazioni è necessario che un insieme dinamico fornisca solamente le seguenti operazioni:

- **INSERT**(key, value): inserisce un elemento nuovo, con un certo valore (unico) di un campo *chiave*.
- **SEARCH**(key): determina se un elemento con un certo valore *chiave* esiste; se esiste, lo restituisce.
- **DELETE**(key): elimina l'elemento identificato dal campo *chiave*, se esiste.

Non è necessario dover ordinare l'insieme dei dati o restituire l'elemento massimo, o il successore. Queste strutture dati prendono talvolta il nome di ***dizionari***.

Per poter implementare un dizionario abbiamo bisogno di due valori:

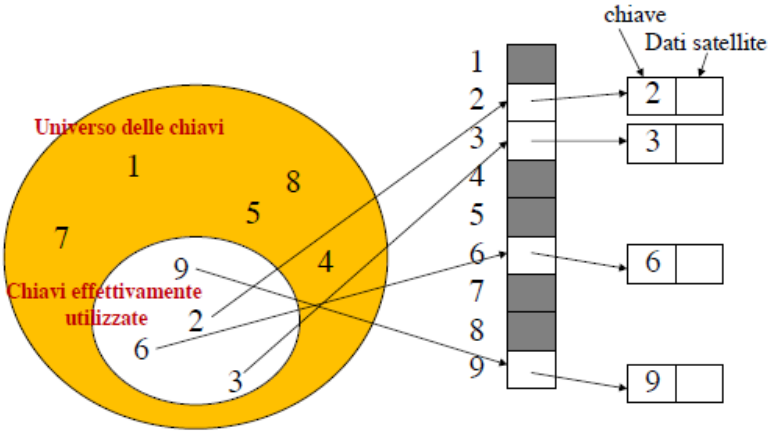
- Universo di tutte le possibili chiavi(***U***).
- Insieme delle chiavi effettivamente memorizzate(***K***).

Abbiamo due tipi di impementazioni:

1. Tabelle ad indirizzamento diretto: U corrisponde all'intervallo ***[0..m-1],*** |***K***| ~ |***U***|.
2. Tabelle Hash: U è un insiemo generico, |***K***| << |***U***| .

Tabelle ad indirizzamento diretto

Se l'universo delle chiavi è piccolo allora è sufficiente utilizzare una tabella ad indirizzamento diretto. Una tabella ad indirizzamento diretto corrisponde al concetto di array, infatti ad ogni chiave possibile corrisponde una posizione, o slot, nella tabella. Una tabella restituisce il dato memorizzato nello slot di posizione indicato tramite la chiave in tempo **O(1)**.



Dizionario mediante tabella associativa

- *T: tabella associativa, k: chiave, x: elemento*
- *Search(T, k)*
  Return T[k]
- *Insert(T, x)*
  T[x->key] ← *x*
- *Delete(T, x)*
  T[x->key] ← *NIL*

Complessita O(1), occupazione O(|U|)

Memorizzazione

È possibile memorizzare i dati satellite direttamente nella tabella oppure memorizzare solo puntatori agli oggetti veri e propri, si deve distinguere l'assenza di un oggetto di un oggetto (oggetto NIL) dal caso particolare di un valore dell'oggetto stesso, se il dato è un intero positivo è possibile assegnare il codice **-1** per indicare ***NIL***.

Universo grande delle chiave

Se l'universo delle possibili chiavi è molto grande non è possibile o conveniente utilizzare il metodo delle tabelle ad indirizzamento diretto. Questo può non essere possibile a causa della limitatezza delle risorse di memoria, e può non essere conveniente perchè se il numero di chiavi effettivamente utilizzato è piccolo si hanno tabelle quasi vuote. Viene quindi allocato spazio inutilizzato.
Le ***tabelle hash*** sono strutture dati che trattano il problema della ricerca permettendo di mediare i requisiti di memoria ed efficieza nelle operazioni.

Quindi nel caso della ricerca si deve attuare un compromesso fra spazio e tempo:

- **Spazio:** se le risorse di memoria sono sufficienti si può impiegare la chiave come indice(accesso diretto).
- **Tempo:** se il tempo di elaborazione non rappresenta un problema si potrebbero memorizare solo le chiavi effettive in una lista ed utilizzare un metodo di ricerca sequenziale.

Quindi:

- Se $|K| \sim |U|$: non sprechiamo(troppo) spazio, operazioni in tempo O(1) nel caso peggiore.
- Se $|K| << |U|$: soluzione non praticabile

ES: studenti PSD con chiave "numero di matricola":

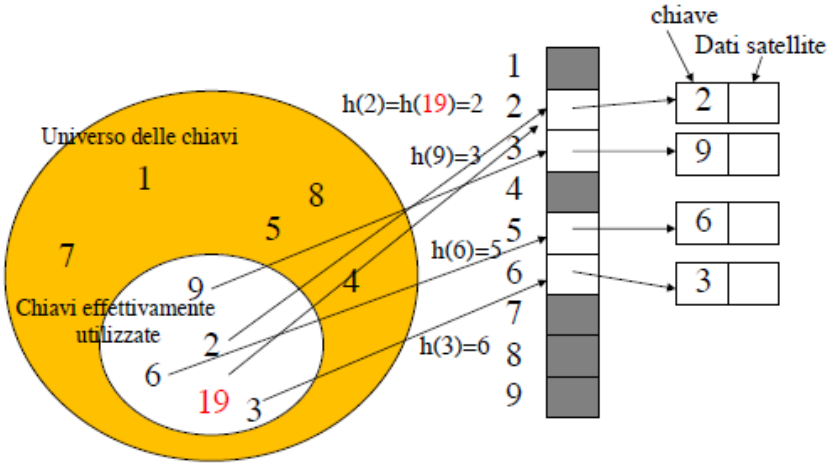
Se il numero di matricola ha 6 cifre, l'array deve avere spazio per contenere 10^6 elementi. Se però gli studenti del corso sono ad esempio 30, lo spazio realmente occupato dalle chiavi memorizzzate è $30/10^6 = 0.00003 = 0.003\%$.

L'***Hashing*** permette di impiegare una quantità ragionevole sia di memoria che di tempo operando un compromesso tra i casi precedenti.
L'hashing è un problema classico dell'informatica, gli algoritmi usati sono stati studiati intensivamente da un punto di vista teorico e sperimentale. Gli algoritmi di hashing sono largamente usati in un vasto insieme di applicazioni(per esempio nei compilatori dei linguaggi di programmazione si usano hash che hanno come chiavi le stringhe che corrisponono agli identificatori del linguaggio.

Tabelle Hash

Con il metodo di indirizzamento diretto un elemento con chiave *k* viene memorizzato nella tabella in posizione k. Con il metodo hash un elemento con chiave k viene memorizzato nella tabella in posizione *h(k)*. La funzione h() è detta funzione hash. Lo scopo della funzione hash è di definire una corrispondenza tra l'universo U delle chiavi e le posizioni di una tabella hash *T[0..m-1] con m << | U| h : U ® {0,1,...,m-1}*. Un elemento con chiave k ha posizione pari al valore hash di k denotato con h(k). Tramite l'uso di funzioni hash il range di variabilità degli indici passa da |U| a *m*. Utilizzando delle dimensioni *m* comparabili con il numero di dati effettivi da gestire si riduce la dimensione della struttura dati garantendo al contempo tempi di esecuzione di O(1).
Necessariamente la funzione hash non può essere ***iniettiva***, ovvero due chiavi distinte possono produrre valore hash. Ogniqualvolta h(ki)=h(kj) quando ki ≠ kj, si verifica una ***collisione***, quindi occorre:

- Minimizzare il numero di collisioni (ottimizzando la funzione di hash).
- Gestire le collisioni residue, quando avvengono (permettendo a più elementi di risiedere nella stessa locazione).



Specifica sintattica

- ***Tipo di riferimento**: hashtable*
- ***Tipi usati**: item, boolean, key* ***//Si usa un tipo generico item contenente un attributo chiave di tipo Key.***
- ***Operatori***

  newHashTable() → hashtable
  insertHash(item, hashtable) → boolean
  searchHash(key, hashtable) → item
  deleteHash(key, hashtable) → boolean

Specifica semantica

- **Tipo di riferimento hashtable**: hashtable e l'insieme di elementi T={a1,a2,…,an} di tipo item. Un item contiene un campo chiave di tipo key e dei dati associati.
- **Operatori:**
  1. *newHashtable() → t*
     Post: t = {}
  2. *insertHash(e, t) → t'*
     Post: t = {a1, a2, ... an}, t' = {a1, a2, ..., e,..., an}
  3. *deleteHash(k,t) → t'*
     Pre: t = {a1, a2, ..., an} n>0, ai->key=k con 1 ≤ i ≤ n
     Post: l' = <a1, a2, ..., ai-1, ai+1, ..., an>
  4. *searchHash(k,t) → e*
     Pre: t = {a1, a2, ..., an} n>0
     Post: e = ai con 1 ≤ i ≤ n se ai->key=k

Funzioni Hash

Una funzione hash ha le seguenti caratteristiche:

- **Criterio di uniformità sempice:** il valore hash di una chiave k è uno dei valori 0...m-1 in modo equiprobabile.
- **Formalmente:** se si estrae in modo indipendente una chiave k dall'universo U con probabilità P(k) allora: $\sum_{k:h(k)=j} P(k) = 1/m$ per j=0,1,…,m-1, cioè se si partiziona l'universo U in sottoinsiemi tali per cui nello stesso sottoinsieme consideriamo tutte le chiavi che sono mappate dalla funzione *h* in *j*, allora vi è la stessa probabilità di prendere un elemento da uno qualsiasi di questi sottoinsiemi.

Tuttavia non sempre si conosce la distribuzione di probabilità delle chiavi P.

ES: se si ipotizza che le chiavi siano numeri reali distribuiti in modo indipendente ed uniforme nell'intervallo [0, 1], allora la funzione $h(k)=\lfloor k\ m \rfloor$, soddisfa il criterio di uniformità semplice. Un altro requisito è che una "buona" funzione hash dovrebbe utilizzare tutte le cifre della chiave per produrre un valore hash, valgono le ipotesi sulla distribuzione dei valori delle chiavi nella loro interezza, altrimenti dovremmo considerare la distribuzione solo della parte di chiave utilizzata.

In genere le funzioni hash assumono che l'univeso delle chiavi sia un sottoinsieme dei numeri naturali, quando questo non è verificato si procede convertendo le chiavi in un numero naturale (anche se grande). Un metodo molto usato è quello di stabilire la conversione fra sequenze di simboli interpretati come numeri in sistemi di numerazione in base.

Per convertire una stringa in un numero naturale si considera la stringa come un numero in base 128, cioè esistono 128 simboli diversi per ogni cifra di una stringa, ed è possibile stabilire una conversione fra ogni simbolo e i numeri naturali (codifica ASCII ad esempio), la conversione viene poi fatta nel modo tradizionale.

ES: per convertire la stringa "pt" si ha: 'p'*128^1 + 't'*128^0= 112 * 128 + 116 *1= 14452

Metodo di divisione
La funzione hash è del tipo: *h(k)=k mod m,* cioè il valore hash è il resto della divisione di k per m, le sue caratteristiche:

- Il metodo è veloce.
- Si deve fare attenzione ai valori di m.
- *m* deve essere diverso da 2^p per qualche, altrimenti fare il modulo in base *m* corrisponderebbe a considerare solo i *p* bit meno signidicativi della chiave, in questo caso dovremmo garantire che la distribuzione dei *p* bit meno significativi sia uniforme.
- Analoghe considerazioni per *m* pari a potenze del 10, dei buoni valorio numeri primi non troppo vicini a potenze del due.

Chiavi molto grandi
Spesso capita che le chiavi abbiano dimensione tale da non poter essere rappresentate come numeri interi per una data architettura.
ES: la chiave per la stringa: "averylongkey" è:
$97*127^{11}+ 118*127^{10}+ 101*127^{9}+ 114*127^{8}+ 121*127^{7}+108*127^{6}+ 111*127^{5}+ 110*127^{4}+ 103*127^{3}+ 107*127^{2}+101*127^{1}+ 121*127^{0}$, che è troppo grande per poter essere rappresentata. Un modo alternativo di procedere è di utilizzare una funzione hash modulare, trasformando un pezzo di chiave alla volta. Per fare questo basta sfruttare le proprietà aritmetiche dell'operazione modulo e usare l'algoritmo di ***Horner*** per scrivere la conversione, si ha infatti che il numero precedente può essere scritto come:
((((((((((97*128 + 118)*128 + 101)*128 + 114)*128+121)*128+ 108)*128 + 111)*128 + 110)*128 + 103)*128 + 107)*128 +101)*128 + 12). Possiamo perciò calcolare il valore finale moltiplicando per 128 la cifra, aggiungendo la seconda cifra, moltiplicando nuovamente per 128 e così via. Il procedimento iterativo arrivrà a colcolare un interno non rappresentabile, ma dato che siamo interessati al resto della divisione per m non è necessario arrivare a calcolare numeri molto grandi infatti basta eliminare in ogni momento i multipli di *m*.
Tutte le volte che eseguiamo una moltiplicazione ed una somma dobbiamo solo ricordarci il resto del modulo di *m*. Se ogni volta estraiamo il modulo otteniamo lo stesso risultato che avremmo se potessimo calcolare l'intero numero e dividessimo per *m* alla fine.
Codice c per funzione hash per stringhe

```
int hash(char *v, int m)
{
        int h = 0, a = 128;
        for (; *v != '\0'; v++)
                h = (a*h + *v) % m;
        return h;
}
```

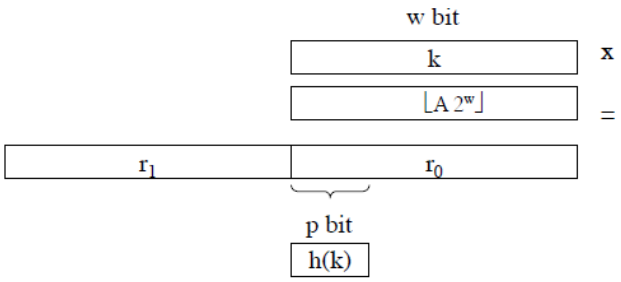
Metodo di moltiplicazione
Il metodo di moltiplicazione per definire le funzioni hash opera in due passi:

- Si moltiplica la chiave per una costante A in [0, 1] e si estrae la parte frazionaria del risultato.
- Si moltiplica questo valore per *m* e si prende la parte intera.

Analiticamente si ha: h(k)=⌊m(k A mod 1)⌋, il come segliere il valore di A ci è suggerito da Knuth: $A \approx (\sqrt{5} - 1)/2$.
Un suo svantaggio è che è più lento del metodo di divisione, un suo vantaggio è che non ha valori critici di *m*.
ES:
A=(√5 -1)/2=0.61803… sia k=123456 e m=10000 allora:
*h(k)= ⌊10000(123456*0.61803.. mod 1)⌋= ⌊10000(76300.0041151.. mod 1)⌋*
= ⌊10000(0.0041151..)⌋= ⌊41.151..⌋=41

Spesso si sceglie $m=2^p$ per un qualche *p* in modo da semplicare i calcoli. Un' implementazione veloce di una *h* per moltiplicazione è il seguente:

- Supponiamo che la chiave *k* sia un numero codificabile entro *w* bit dove *w* è la dimensione di una parola del calcolatore.
- Si consideri l'intero anch'esso di *w* bit $\lfloor A\,2^w \rfloor$.
- Il prodotto $k^* \lfloor A\,2^w \rfloor$ sarà un numero intero di al più 2^w bit.
- Consideriamo tale numero come $r_1 2^w + r_0$.
- r_1 è la parola più significativa del risultato e r_0 quella meno significativa.
- Il valore hash di *p* bit più significativi di r_0.



Metodo della funzione universale

È possibile che la scelta di chiavi sia tale da avere un elevato numero di collisioni, il caso peggiore è che tutte le chiavi collidano in un unico valore hash. Le prestazioni delle operazioni con tabelle hash dove la maggior parte delle chiavi hanno un unico valore hash peggiorano fino a $\Theta(n)$. Si può (come per il quicksort) utilizzare un algoritmo randomizzato di hashing in modo da garantire che non esista nessun insieme di chiavi che porti al caso peggiore.

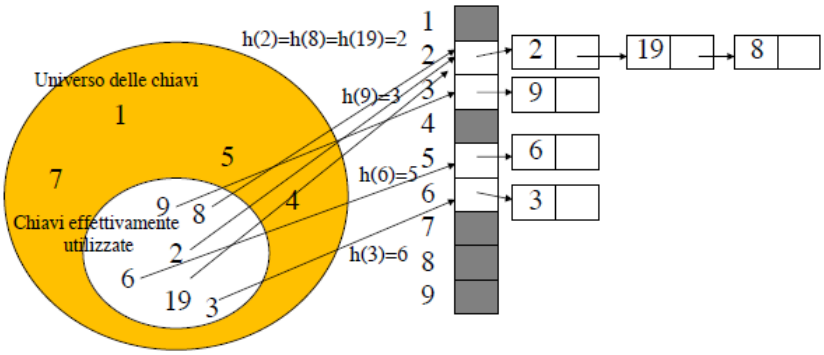
L'idea di base è di scegliere per ogni chiave una funzione hash casualmente da una famiglia di funzioni che rispettano specifiche proprietà, in questo modo per qualsiasi insieme di chiavi si possono avere molte collisioni solo a causa del generatore pseudo casuale, ma si può rendere piccola la probabilità di questo evento.

Sia H un insieme finito di funzioni hash che vanno da U in {0, 1,..., m-1}, H è un insieme universale se per ogni coppia di chiavi distinti x, $y \in U$, il numero di funzioni hash $h \in H$, per cui h(x)=h(y) è esattamente |H|/m. Questa definizione equivale a dire che con una *h* scelta a casso la probabilità che h(x)=h(y) è 1/m.

Metodi per la risoluzione delle collisioni

Per risolvere il problema delle collisioni si impiegano principalmente due strategie:

- **Metodo di concatenazione:** l'idea è di mettere tutti gli elementi che collidono in una lista concatenata, la tabella contiene in posizione j: un puntatore alla testa della j-esima lista, oppure un puntatore nullo se non ci sono elementi.



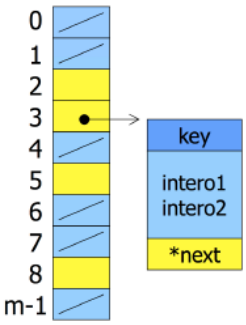
Hash tipi di dato

- Vettore di hash T

```
struct hash
 {
        int size;
        struct item **table;
};ypedef struct hash* hashtable;
```

- Elemento

```
struct item {
        char *key; /* chiave */
        int intero1;
        int intero2;
        struct item *next;
};
```



- ***NEW(T,m)***
  void NewHashtable(hashtable h, int size);
- ***Chained-InsertHash(T,x)***
  int InsertHash(hashtable h, item x);
- ***Chained-SearchHash(T,k)***
  *struct item *SearchHash(hashtable h, char *key);*
- ***Chained-DeleteHash(T,k)***
  *struct item *DeleteHash(hashtable h, char *key);*
- ***DESTROY(T)***
  void DestroyHashtable(hashtable h);

Codici

Si assuma di disporre di un'opportuna funzione h(k) del tipo:

```
int hash(char *k, int m)
{
        int val;
        int len = strlen(k);
        val = k[0] + k[len-1] + len;
        return (val % m);
}

hashtable newHashtable(int size)
 {
        int i;
        hashtable h = malloc(sizeof(struct hash));
        h->size = size;
        h->table = (struct item **) malloc (size*sizeof(struct item *));
        for(i=0; i<size; i++)
         {
                h->table[i] = NULL;
        }
        return h;
}
```

```
int InsertHash(hashtable h, struct item *elem)
{
	int idx;
	struct item *head, *curr;
	idx = hash(elem->key, h->size);
	curr = head = h->table[idx];
	while(curr)  // controllo se esiste già un item con la stessa chiave di elem
	{
		if( strcmp(curr->key, elem->key)==0 )
		return (-1);
		curr = curr->next;
	}
	// inserisce in testa alla lista
	h->table[idx] = elem;
	h->table[idx]->next = head;
	return 0;
}

struct item* SearchHash(hashtable h, char* key)
{
	int idx;
	struct item *curr;
	idx = hash(elem->key, h->size);
	curr = h->table[idx];
	while(curr)  // controllo se esiste un item con la stessa chiave key
	{
		if( strcmp(curr->key, key)==0 )
		return curr;
		curr = curr->next;
	}
	// elemento non trovato
	return NULLITEM;
}

int DeleteHash(hashtable h, char *key)
 {
	int idx;
	struct item *prev, *curr, *head;
	idx = hash(key, h->size);
	prev = curr = head = h->hash[idx];
	while(curr)
	 {
		if( strcmp(curr->key, key)==0 )
		{
			if(curr == head) // cancello in testa
				h->hash[idx] = curr->next;
			else prev->next = curr->next;
			return 1;
		}
		prev = curr;
		curr = curr->next;
	}return 0;}
```

```
void DestroyHash(hashtable h)
{
        int i;
        for(i=0; i < h->size; i++)
        {
                deleteList(h->hash[i]);
        }
        free(h->hash);
        return;
}

static void deleteList(struct item *p)
 {
        if (p == NULL) return;
        deleteList(p->next);
        free(p);
}
```

Fattore di carico

Data una tabella hash T con *n* il numero di elementi memorizzati, *m* la dimensione della tabella hash. Il fattore di carico di ti si definisce con α=n/m, dove α è il numero medio di elementi memorizzati in ogni lista concatenata, α è in genere ≥ 1. L'analisi della complessità viene fatta in funzione di α, si suppone che α sia definito anche se *n* e *m* tendono all'infinito.

Tempo di calcolo

Si fa l'ipotesi che il *tempo di calcolo di h* sia O(1) così che il tempo di calcolo delle varie operazioni dipende solo dalla lunghezza delle liste. Il tempo di esecuzione per *l'inserimento* è 0(1) nel caso peggiore, il tempo di esecuzione per la *cancellazione* è 0(1) se le liste sono *bidirezionali*. Se le liste sono *semplici* si deve prima trovare il predecessore di x per poterne aggiornare il link next. In questo caso la cancellazione e la ricerca hanno lo stesso tempo di esecuzione.

Tempo di calcolo per la ricerca

I. **Il caso peggiore** si ha quando tutte le chiavi collidono e la tabella consiste in una unica lista di lunghezza *n* in questo caso il tempo di calcolo è O(n).
II. **Il caso medio** dipende da quanto in media la funzione hash distribuisca l'insieme delle chiavi sulle *m* posizioni.
III. Se la funzione hash soddisfa l'ipotesi di *uniformità semplice* allora ogni lista ha lunghezza media α.

Tempo di calcolo per la ricerca con insuccesso

Il tempo medio è il tempo impiegato per generare il valore hash data la chiave e il tempo necessario per scandire una sequenza fino alla fine**,** dato che la lunghezza media di una sequenza è α si ha $O(1+\alpha)$.

- **Metodo di indirizzamento aperto:** l'idea è di memorizzare tutti gli elementi nella tabella stessa, in caso di collisione si memorizza l'elemento nella successiva; per l'operazione di ricerca si esaminano tutte le posizioni ammesse per la data chiave in sequenza, non vi sono liste né elementi memorizzati fuori dalla tabella. Il fattore di carico α è sempre ≤ 1. Per eseguire l'inserzione si genera un valore hash data la chiave e si esamina una successione di posizioni della tabella (*scansione*) a partire dal valore hash fino a trovare una posizione vuota dove inserire l'elemento.

Sequenza di scansione
La sequenza di scansione dipende dalla chiave che deve essere inserita, per fare questo si estende la funzione hash perchè generi non solo un valore hash ma una sequenza di scansione:
$h : U \times \{0,1,\ldots,m-1\} \rightarrow \{0,1,\ldots,m-1\}$, cioè prenda in ingresso una chiave e un indice di posizione e generi una nuova posizione. Data una chiave k, si parte dalla posizione 0 e si ottiene h(k, 0), la seconda posizione da scansionare sarà h(k, 1) e così via h(h, i), ottenendo una sequenza <h(k,0),h(k,1),…,h(k,m-1)>, h(k, i) al variare di *i* deve essere una permutazione degli elementi {0,1,...,m-1}.

Caratteristiche di h
Quali sono le caratteristiche di una buona funzione hash per il metodo di indirizzamento aperto? Si estende il concetto di uniformità semplice, dove la *h* deve soddisfare la **proprietà di uniformità della funzione hash:**

- per ogni chiave k la sequenza di scansione generata da h deve essere una qualunque delle m! Permutazioni di {0, 1,...,m-1}.

Funzioni Hash per indirizzamento aperto
È molto difficile scrivere funzioni *h* che rispettino la proprietà di uniformità , si usano generalmente tre approssimazioni:

- ***scansione lineare*:**
  Data una funzione hash h*':U → {0,1,…,m-1}* il metodo di scasione lineare costruisce una h(k, i) nel modo seguente: *h(k,i)=(h'(k)+i) mod m,*data la chiave k si genera la posizione h'(k), quindi la posizione h'(k)+1, e così via fino alla posizione m-1. Poi si scandisce in modo circolare la posizione 0, 1, 2, fino a ritornare a h'(k)-1.
  Es:
  Inserimento delle chiavi
  10, 22, 31, 4, 15, 28, 17, 88, 59
  In tabella con m = 11 con open addressing

  10, 22, 31, 4, 15, 28, 17, 88, 59

0	22
1	88
2	
3	
4	4
5	15
6	28
7	17
8	59
9	31
10	10

  Linear probing
  h(k, i) = (k mod 11 + i) mod 11

  Collisione: h(59, 0) = 4
  h(59, 1) = 5
  h(59, 2) = 6
  h(59, 3) = 7
  h(59, 4) = 8

  Agglomerazione primaria
  La scansione lineare è facile da realizzare ma presenta il fenomeno di agglomerazione(*clustering*) primaria: si formano lunghi tratti di posizioni occupate aumentando i tempi di ricerca.
  Si pensi ad esempio il caso in cui vi siano n=m/2 chiavi nella tabella: se le chiavi sono disposte in modo da alternare una posizione occupata con una vuota, allora la ricerca senza successo richiede 1,5 accessi. Se le chiavi sono disposte tutte nelle prime m/2 posizioni allora si devono effettuare n/4 accessi in media per la ricerca senza successo.
  scansione quadratica: Data una funzione hash h':U → {0,1,…,m-1} il metodo di scansione quadratca costruisce una h(k, i) nel modo seguente: *h(k,i)=(h'(k) + c1i + c2i2) mod m, dove* c_1 *e* c_2 e *m* sono costanti vincolate per poter far usa dell'intera tabella, questo metodo funziona meglio della scansione lineare perché evita l'agglomerazione primaria ma non

risolve il problema della agglomerazione (che in questo caso viene detta secondaria) in quanto se si ha una collisione sul primo elemeto le due sequenze di scansione sono comunque identiche.

ES:

10, 22, 31, 4, 15, 28, 17, 88, 59

0	22
1	
2	88
3	17
4	4
5	
6	28
7	59
8	15
9	31
10	10

Quadratic probing

$h(k, i) = (k \bmod 11 + i + 3i^2) \bmod 11$

Collisione: $h(59, 0) = 4$

$h(59, 1) = 8$

$h(59, 2) = 7$

- ***hashing doppio:*** l'hashing doppio risolvere il problema delle agglomerazioni ed approssima in modo migliore la proprietà di uniformità.
  Date due funzioni hash:
  $\mathbf{h_1}$***:*** $U \rightarrow \{0,1,\ldots,m-1\}$
  h_2***:*** $U \rightarrow \{0,1,\ldots,m'-1\}$
  Il metodo di scansione quadratica costruisce una h(k, i) nel modo seguente:

$$h(k,i)=(h1(k) + i\ h2(k)) \bmod m$$

L'idea è di partire da un valore hash e di esaminare le posizioni successive saltando di una quantità pari a multipli di un valore determinato da un'altra funzione hash, in questo modo in prima posizione esaminata è $h_1(k)$ mentre le successive sono distanziate di $h_2(k)$ dalla prima.

ES:

Si consideri

- m=13
- $h_1(k)$= k mod 13
- $h_2(k)$=1+(k mod 11)

si consideri l'inserimento della chiave 14 nella seguente tabella:

0	1	2	3	4	5	6	7	8	9	10	11	12
	79			69	98		72				50	

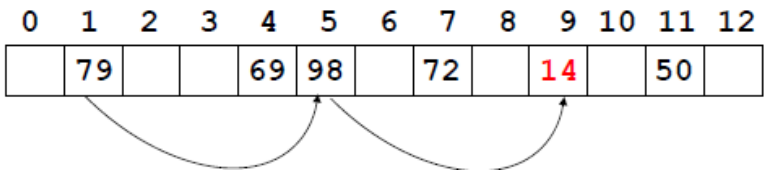
$h_1(14)= 14 \bmod 13= 1$

$h_2(14)=1+(14 \bmod 11) = 1 +3 = 4$

1 posizione esaminata 1

2 posizione esaminata 1 + 4 = 5

3 posizione esaminata 1 + 4*2 = 9



Considerazioni su h_1 e h_2
Si deve fare attenzione a far si che $h_2(k)$ sia prima rispetto alla dimensione della tabella *m*, infatti, se *m* e $h_2(k)$ hanno un massimo comun divisore *d*, allora la sequenza cercata per k esaminerebbe solo *m/d* elementi.
ES: m=10 e $h_2(k)=2$, partendo da 0 si ha la sequenza 0 2 4 6 8 0 2 4 6 8….
Per garantire che $h_2(k)$ sia primo rispetto *m* si può:

- Prendere $m=2^p$ e segliere $h_2(k)$ in modo che produca sempre un numero *dispari*.
- Un altro modo è di prendere *m primo* e segliere $h_2(k)$ in modo che produca sempre un intero positivo *minore di m*.

ES:
$h_1(k)=k \bmod m$
$h_2(k)=1+(k \bmod m')$
dove ***m'*** è di poco minore di *m*; ***m'=m-1*** o ***m'=m-2***.

Considerazioni sull'hashing doppio
L'hashing doppio approssima in modo migliore la proprietà di uniformità rispetto alla scansione lineare o quadratica.
Infatti:

- La scansione lineare (quadratica) genera solo **O(m) sequenze**; una per ogni chiave.
- L'hashing doppio genera **$O(m^2)$ sequenze**; una per ogni coppia (chiave, posizione).

La posizione iniziale $h_1(k)$ e la distanza $h_2(k)$ possono variare indipendentemente l'una dall'altra.
ES:

10, 22, 31, 4, 15, 28, 17, 88, 59

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

Double probing
h(k, i) = (k mod 11 +
i (1 + k mod 10)
) mod 11

Tutte queste classi di funzioni garantiscono di generare una permutazione ma nessuna riesce a generare tutte le m! permutazioni.

Funzioni hash in C

- **Indirizzamento aperto:** ***int hash_oa(char *k, int i, int size)***
- **Linear probing:** $h(k, i) = (h'(k)+i) \bmod m$
  (hash(k,size) + i) % size
- **Quadratic probing:** $h(k, i) = (h'(k)+ c_1 i + c_2 i^2) \bmod m$
  (hash(k,size) + K1*i + K2*(i*i)) % size
- **Double hashing**: $h(k,i)=(h_1(k) + i\, h_2(k)) \bmod m$; ES: $h_1(k) = k \bmod m$ $h_2(k) =1+ (k \bmod m')$
  (hash(k,size) + i*hash2(k,size)) % size

```
int hash(char *k, int size)
	{
		int len = strlen(k);
		int v = (k[0] + k[len-1] + len);
		return ( v % size);
	}

	int hash2(char *k, int size)
	{
		int sum = 0;
		while(*k!='\0')
		{
			sum += *k;
			k++;
		}
		return (sum % size);
	}
```

Implemetazione in C

```
	int HashInsert( struct hash *h, char *key, int intero1,int intero2 )
	{
		int idx, start; int i = 0, len = h->size;
		start = idx = hash_oa(key,i,h->size);
		do
		{
			if(h->hash[idx] == NULL)  /* posto libero */
			{
				h->hash[idx]=newItem(key,intero1,intero2);
				return (1);
			} else /* prova successivo */
			{
				i++;
				idx = hash_oa(key,i,h->size);
			}
		} while (idx != start);
		return (0);
	}

	item HashSearch(struct hash *h, char *key)
	{
		int idx, start; int i = 0;
		start = idx = hash_oa(key,i,h->size);
		do
		{
			if(h->hash[idx] == NULL)  /* posto libero */
			{
				return (NULL);
			} else
			{
				if ( !strcmp(h->hash[idx]->key, key) )
				{
```

```
                return h->hash[idx];
            } else
            {
                i++; idx = hash_oa(key,i,h->size);
            }
        } while( idx != start );
        return (NULL);
}
```

Analisi del tempo di calcolo
Si suppone di lavorare con *funzioni hash uniformi* in questo schema ideale, la sequenza di scansione ***<h(k,0),h(k,1),…,h(k,m-1)>*** è in modo equiprobabile una qualsiasi delle m! Permutazioni di <0, 1,..., m-1>, in una ricerca senza successo si accede ogni volta ad una posizione occupata che non contiene la chiave e l'ultimo accesso è poi fatto ad una posizione vuota.
Quindi possiamo dire che:

- Un accesso viene sempre fatto.
- Un secondo accesso viene fatto con probabilità α.
- Un terzo accesso con probabilità α^2.
- In media si dimostra che il numero medio di accessi per ricerca senza successo è $1+ \Sigma_{i=1..\infty} \alpha^i = 1/(1- \alpha)$.

Data una tabella hash ad indirizzamento aperto con fattore di carico α, il numero medio di accessi per una ricerca con successo è al più $\boldsymbol{1/\alpha \ln 1/(1-\alpha)}$, nell'ipotesi di funzione hash uniforme con chiavi equprobabili.

La cancellazione
L'operazione di cancellazione è difficile, questo perchè non si può marcare la posizione come vuota con ***NIL*** perchè questo impedirebbe la ricerca degli elementi successivi nella sequenza, per ovviare a questo problema si potrebbe usare uno speciale marcatore ***DELETED*** invece che di ***NIL***. Le soluzioni sono quindi due: *marcatori* e *reinserimento*. In genere se si prevedono molte cancellazioni si utilizzano i metodi con concatenazione.
Con il marcatore ***DELETED*** la ricerca non si ferma quando si trovano celle marcate con ***DELETED,*** ma prosegue. La procedura di inserzione sovracrive il contenuto delle celle marcate con *deleted*.
NOTA: in questo modo però i tempi non dipendono più solo dal fattore di carico.
In alternativa all'uso del marcatore si può tenere conto di quanti elementi sono stati cancellati e quando il conteggio supera una soglia critica si può cambiare la dimensione del dizionario.
Per il reinserimento, si reinseriscono tutti gli elementi che la cancellazione renderebbe irraggiungibili, cioè tutti gli elementi nella sequenza di scansione a partire dall'elemento cancellato fino alla cella libera (marcata con nil). Se la tabella è sparsa (fattore di carico basso) questa operazione richiede il reinserimento solo di pochi valori.

Tabelle Hash dinamiche
Quando il fattore di carico tende a uno (*cresce*) nelle tabelle hash che utilizzano l'indirizzamento aperto le prestazioni decrescono. Per mantenere prestazioni accettabili si ricorre al *ridimensionamento* della tabella:

- Quando si oltrepassa un certo fattore di carico si raddoppia la dimensione della tabella.
- Quando si scende sotto un certo fattore di carico si dimezza la dimensione della tabella.

Ogni volta che si ridemensiona una tabella hash cambia anche la funzione hash e l'associazione *chiave-valore* hash calcolata precedentemente non è più valida pertando si devono reinserire tutti gli elementi!

Stategia per evitare ciò
Si deve fare attenzione a non avere soglie per innescare l'espansione e la riduzione troppo vicine, altrimenti per frequenti operazioni di inserimento/cancellazione si consuma molto tempo a ridimensionare la tabella. Una buona strategia è:

- Raddoppiare la dimensione quando il fattore carico supera $^1/_2$.
- Dimezzare la dimensione quando il fattore di carico scende sotto $^1/_8$.

In questo modo si garantisce la proprietà per la quale una sequenza di *t* operazioni fra ricerche, inserimenti e cancellazioni può essere eseguita in tempo proporzionale a *t.* Inoltre si usa sempre una quantità di memoria al più pari a una costante moltiplicata per il numero di chiavi nella tabella.

Prestazioni
Sebbene l'operazione di espansione o riduzione della dimensione di una tabella sia molto costosa, questa viene fatta solo raramente. Inoltre ogni volta che raddoppiamo o dimezziamo la dimensione il fattore di carico della nuova tabella è $^1/_4$ quindi l'inserimento sarà efficiente (ogni inseriemento provecherà < 2 collisioni). Si dimostra quindi che *raddoppiando/dimezzando* e *inserendo/cancelando* elementi al massimo raddoppia il numero di operazioni (inserimenti/cancellazioni) pertando la complessità rimane lineare rispetto alla complessità del metodo hash non dinamico.

Considerazioni finali
È complicato confrontare i metodi di hashing con concatenazione ed indirizzamento aperto, infatti il fattore di carico α deve tenere conto del fatto che nel caso di indirizzamento aperto si memorizza direttamente la chiave mentre ne caso di concatenazione si ha la memorizzazione aggiuntiva dei puntatori per gestire le liste, in generale si preferisce ricorrere al metodo delle concatenazioni quando non è noto il fattore di carico, mentre si ricorre all'indirizzamneto aperto quando esiste la possibilità di predire la dimensione dell'insieme delle chiavi.

Alberi binari di ricerca e hash
La scelta di utilizzare strutture dati di tipo alberi binari di ricerca o hash prende in considerazione i seguenti fattori:

- ***vantaggi per hashing:*** è di facile e veloce implementazione e tempi di ricerca rapidissimi.
- ***Vantaggi per alberi binari di ricerca:*** minori requisiti di memoria, diventano dinamiche, buone prestazioni anche nel caso peggiore e infine supportano un numero maggiore i operazioni (ordinamento).

ES:
Data una tabella hash di lunghezza m=11, si supponga di dover inserire (in ordine) le chiavi:

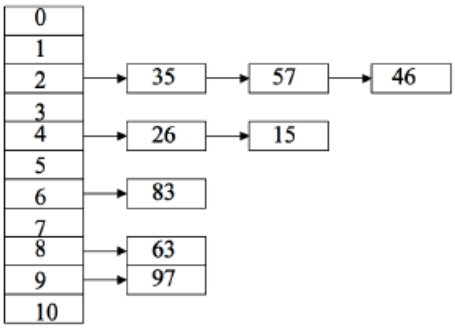
35, 83, 57, 27, 15, 63, 97, 46

utilizzando la funzione di *hash h(k)= k mod m.*
Si illustrino i risultati dell'inserimento usando:

- Concatenazione.
- Scansione lineare.
- Scansione quadratica (h(k, i) = (h(k)+i^2)mod m).
- Hashing doppio con h_2(K) = 1+(k mod(m-1)).

1) Concatenazione
Calcolo di h(k):

- *h(35)=35 mod11=2*
- *h(83)=83 mod11=6*
- *h(57)=57 mod11=2*
- *h(26)=26 mod11=4*
- *h(15)=15 mod11=4*
- *h(63)=63 mod11=8*



- *h(97)=97 mod11=9*
- *h(46)=46 mod11=2*

2) Scansione lineare

0	1	2	3	4	5	6	7	8	9	10
		35	57	26	15	83	46	63	97	

h(57)=2 -> lo slot 2 è occupato $h_1(57)=3$

h(15)=4 -> lo slot 4 è occupato $h_1(15)=5$

h(46)=2 -> lo slot 2 è occupato

$h_1(46)=3$ -> lo slot 3 è occupato

$h_2(46)=4$ -> lo slot 4 è occupato

$h_3(46)=5$ -> lo slot 5 è occupato

$h_4(46)=6$ -> lo slot 6 è occupato $h_5(46)=7$

h(35)=2
h(83)=6
h(57)=2
h(26)=4
h(15)=4
h(63)=8
h(97)=9
h(46)=2

3) Scansione quadratica

0	1	2	3	4	5	6	7	8	9	10
46		35	57	26	15	83		63	97	

h(57)=2 -> lo slot 2 è occupato $h_1(57)=3$

h(15)=4 -> lo slot 4 è occupato $h_1(15)=5$

h(46)=2 -> lo slot 2 è occupato

$h_1(46)=3$ -> lo slot 3 è occupato

$h_2(46)=6$ -> lo slot 6 è occupato $h_3(46)=0$

h(35)=2
h(83)=6
h(57)=2
h(26)=4
h(15)=4
h(63)=8
h(97)=9
h(46)=2

4) hashing doppio

0	1	2	3	4	5	6	7	8	9	10
	46	35		26	15	83		63	97	57

h(57)=2 -> lo slot 2 è occupato $h_1(57)=2+1*8=10$

h(15)=4 -> lo slot 4 è occupato $h_1(15)=4+1*6=10$

-> lo slot 10 è occupato $h_2(15)=4+2*6=5$

h(46)=2 -> lo slot 2 è occupato $h_1(46)=2+1*7=9$

-> lo slot 9 è occupato $h_2(46)=2+2*7=5$

-> lo slot 5 è occupato $h_3(46)=2+3*7=1$

h(35)=2
h(83)=6
h(57)=2
h(26)=4
h(15)=4
h(63)=8
h(97)=9
h(46)=2

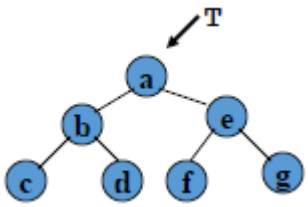
VISITE DI ALBERI BINARI

Visita di alberi

Gli alberi possono essere visitati (o attraversati) in diversi modi:

- ***Visita in preordine (preorder)***: prima si vista il nodo e poi i suoi sottoalberi.

```
Visita-Preordine(node *T)
{
        IF (T)
        {
                Vista(T);
                Visita-Preordine(T->sx);
                Visita-Preordine(T->dx);
        }
}
```



Sequenza di stampa: *a b c d e f g*

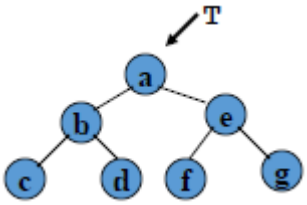
Visita preorder iterativa

```
Visita-preorder-iter(node * T)
{
        stack *st = NULL;
        node *curr = T;
        while(st || curr)
        {
                if(curr) /* visita e discesa a sx */
                {
                        Visita(curr);
                        st = push(st,curr);
                        curr = curr->sx
                } else /* discesa a dx */
                {
                        curr = top(st);
                        st = pop(st);
                        curr = curr->dx;
                }
        }

}

Visita-preorder-iter2(node * T)
{
        stack *st = NULL;
        node *curr = T;
        while(curr)
        {
                Visita(curr);
                if(curr->dx) /* Salva per discesa a dx */
                        st = push(st,curr->dx);
                if (curr->sx) /* Discesa a sx */
                        curr = curr->sx
                else /* Discesa a dx */
```

```
                {
                        if (st)
                        {
                                curr = top(st);
                                st = pop(st);
                        }
                else curr = NULL;
                }
        }
}
```

- ***Visita in inordine(inorder),(solo se binario):*** prima si visita il sottoalbero sinistro, poi il nodo e infine il sottoalbero destro.

```
Visita-Inordine(T)
        IF T ≠ NIL THEN
                Visita-Inordine(T->sx)
                "vista T"
                Visita-Inordine(T->dx)
```



Sequenza di stampa: *c b d a f e g*

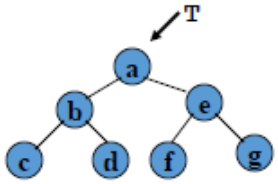
Visita inorder iterativa

```
        Visita-inorder-iter(node * T)
        {
                stack *st = NULL;
                node *curr = T;
                while(st || curr)
                {
                        if(curr) /* Discesa a sx */
                        {
                                st = push(st,curr);
                                curr = curr->sx
                        }
                        else /* Visita e discesa a dx */
                        {
                                curr = top(st);
                                st = pop(st);
                                Visita(curr);
                                curr = curr->dx;
                        }
                }
        }

        Visita-inorder-iter2(node *T)
        {
                stack *st = NULL;
                node *curr, *last = NULL;
                if (T) st = push(T);
                while (st) /* discesa lungo l'albero corrente */
                {
                        curr = top(st);
                        if (!last || curr == last->sx || curr == last->dx)
```

```
        {
                if (curr→sx)
                        st = push(st,curr->sx);
                else
                {
                        Visita(curr);
                        st = pop(st);
                        if (curr->dx) st = push(st,curr->dx);
                }
        else if (last == curr→sx) /* risalita da sinistra */
        {
                Visita(curr);
                if (curr→dx)
                        st = push(st,curr->dx);
        else if (last == curr→dx)  /* risalita da destra */
                        st = pop(st);
        last = curr;
        }
}
```

- ***Visita in postordine(postorder):*** prima si visitano i sottoalberi, poi il nodo.

```
Visita-Postordine(T)
    IF T ≠ NIL THEN
        Visita-Postordine(T->sx)
        Visita-Postordine(T->dx)
        "vista T"
```



Sequenza di stampa: *c d b f g e a*

Visita postorder iterativa

```
Visita-postorder-iter(node * T)
{
        stack *st = NULL;
        node *last, curr = T;
        while(st || curr)
        {
                if(curr) /* discesa a sx */
                {
                        st = push(st,curr);
                        curr = curr->sx
                }
                else /* visita o discesa a dx */
                {
                        curr = top(st);
                        if (!curr->dx || last == curr→dx)
                        {
                                Visita(curr);
                                last = curr;
                                st = pop(st);
                                curr = NULL;
                        }
                        else /* Discesa a dx */
```

```
                    curr = curr->dx;
                }
        }
}

Visita-postorder-iter2(node *T)
{
        stack *st = NULL;
        node *curr, *last = NULL;
        if (T) st = push(T);
        while (st) /* discesa lungo l'albero corrente */
        {
                curr = top(st);
                if (!last || curr == last->sx || curr == last->dx)
                {
                        if (curr->sx) st = push(st,curr->sx);
                        else if (curr->dx) st = push(st,curr->dx);
                        else
                        {
                                Visita(curr); st = pop(st);
                        }
                }
                else if (last == curr→sx)  /* risalita da sinistra */
                {
                        if (curr->dx) st = push(st,curr->dx);
                        else { Visita(curr); st = pop(st);
                }
                else if (last == curr→dx) /* risalita da destra */
                {
                        Visita(curr); st = pop(st);
                }
                last = curr;
        }
}
```