# Strutture Dati
# (parte I: pile, code, liste, tabelle hash)

# Scopo delle strutture dati

- Le strutture dati sono “aggeggi” usati per *contenere oggetti*
  - rappresentano *collezioni* di oggetti
  - spesso (ma non sempre) gli oggetti di in una struttura dati hanno una *chiave*, che serve per indicizzare l'oggetto, e dei *dati satelliti* associati (che sono i dati di interesse che porta con sé l'oggetto)
    - per esempio, si fa ricerca sulla chiave per accedere ai dati satelliti, che possono essere qualunque
- Ci sono 2 tipi di operazioni sulle strutture dati:
  - operazioni che *modificano* la collezione
  - operazioni che *interrogano* la collezione

- Operazioni tipiche sulle strutture dati:
  - `SEARCH(S, k)`
    - restituisce l'oggetto (o, meglio il suo riferimento) *x* nella collezione *S* con chiave *k*, *NIL* se nessun oggetto nella collezione ha chiave *k*
    - è un'operazione di interrogazione
  - `INSERT(S, x)`
    - inserisce l'oggetto *x* nella collezione *S*
    - è un'operazione che modifica la collezione
  - `DELETE(S, x)`
    - cancella l'oggetto *x* dalla collezione *S* (op. di modifica)
  - `MINIMUM(S)`
    - restituisce l'oggetto nella collezione con la chiave più piccola (op. di interrogazione)
  - `MAXIMUM(S)`
    - restituisce l'oggetto nella collezione con la chiave più grande (op. di interrogazione)
  - `SUCCESSOR(S, x)`
    - restituisce l'oggetto che segue *x* nella collezione, secondo una qualche relazione di ordinamento (op. di interrogazione)
    - per esempio, potrebbe essere l'elemento con la prossima chiave più grande, se c'è un ordinamento sulle chiavi
      - potrebbe essere qualcosa d'altro (la sua definizione dipende dalla specifica struttura dati)
  - `PREDECESSOR(S,x)`
    - restituisce l'oggetto che precede *x* nella collezione, secondo una qualche relazione di ordinamento (op. di interrogazione)

# Pile (Stack)

- Cominciamo con un esempio semplicissimo di struttura dati: la pila
- Ad un livello astratto, una pila è una collezione di oggetti sulla quale possiamo fare le seguenti operazioni:
  - controllare se è vuota
  - inserire un elemento nella collezione (`PUSH`)
  - cancellare un elemento dalla collezione (`POP`)
    - l'operazione di `POP` *restituisce* l'elemento cancellato
- Una pila è gestita con una politica LIFO (Last In First Out)
  - l'elemento che viene cancellato (pop) è quello che è stato inserito per ultimo (cioè quello che è nella pila da *meno* tempo)
    - cioè, se viene fatta una PUSH di un oggetto *e* su una pila *S*, seguita immediatamente da una `POP` su *S*, l'elemento restituite dalla `POP` è lo stesso *e* di cui era stata fatta la PUSH

- Se la pila può contenere al massimo $n$ elementi, possiamo implementarla come un array di lunghezza $n$
  - per tenere traccia dell'indice dell'elemento che è stato inserito per ultimo viene introdotto un attributo, chiamato *top*
    - cioè, se una pila $S$ è implementata mediante un array, $S.top$ è l'indice dell'ultimo elemento inserito
      - se $S.top = t$, allora $S[1]$, $S[2]$, ... $S[t]$ contengono tutti gli elementi, e $S[1]$ è stato inserito prima di $S[2]$, che è stato inserito prima di $S[3]$, ecc.
    - se $S.top = 0$, la pila è vuota, e nessun elemento può essere cancellato
    - se $S.top = S.length$, la pila è piena, e nessun elemento vi può essere aggiunto

# pseudocodice per le operazioni sulle pile

- Se una pila è implementata tramite un array, lo pesudocodice per le operazioni su di essa sono le seguenti:

```
PUSH(S, x)
if S.top = S.length
    error “overflow”
S.top := S.top + 1
S[S.top] := x

POP(S)
if S.top = 0
    error “underflow”
S.top := S.top - 1
return S[S.top + 1]
```

# Code (queue)

- Le code sono simili alle pile, salvo che una coda è gestita con una politica FIFO (First In First Out)
- A livello astratto, una coda è una collezione di oggetti sulla quale si possono fare le seguenti operazioni:
  - (controllare se è vuota)
  - inserire un elemento nella collezione (`ENQUEUE`)
  - cancellare un elemento dalla collezione (`DEQUEUE`)
    - si noti che l'operazione di `DEQUEUE` *restituisce* l'elemento cancellato
- Una coda è gestita con una politica FIFO
  - l'elemento che viene cancellato è quello che era stato inserito per primo (cioè quello che è rimasto nella coda per *più* tempo)

Se una coda può contenere al più *n* elementi, allora, come per le pile, possiamo implementarla tramite un array di lunghezza *n*

- ora però dobbiamo tenere traccia di 2 indici:
  - l'indice del prossimo elemento da eliminare (quello che è nella coda da più tempo),
  - l'indice della cella nell'array in sarà memorizzato il prossimo elemento inserito nella coda
- utilizziamo 2 attributi, *head* e *tail*
  - se *Q* è una coda implementata mediante un array, *Q*.*head* è l'indice dell'elemento da più tempo nell'array
  - *Q*.*tail* è l'indice in cui il prossimo elemento inserito dovrà essere memorizzato
    - cioè, *Q*.*tail*-1 è l'indice dell'ultimo elemento inserito

# pseudocodice per le operazioni sulle code

- Prima di introdurre lo pseudocodice di ENQUEUE e DEQUEUE, analizziamo come funziona una coda implementata come array
  - gli elementi di una coda *Q* hanno indici *Q*.*head*, *Q*.*head*+1, ... *Q*.*tail*-1
  - se *Q*.*tail* = *Q*.*length* e un nuovo elemento è inserito, il prossimo valore di *tail* sarà 1
    - la coda funziona in modo "circolare"
    - per esempio, se la coda la lunghezza 10, *Q*.*tail* = 10 e noi inseriamo un nuovo elemento, dopo l'accodamento abbiamo che *Q*.*tail* = 1
  - se *Q*.*head* = *Q*.*tail* la coda è *vuota*
  - se Q.*head* = Q.*tail*+1 la coda è *piena*
    - se la coda non è piena, c'è sempre almento una cella libera tra *Q*.*tail* e *Q*.*head*
    - quindi, se dobbiamo implementare mediante un array una coda *Q* che contiene al massimo *n* elementi, l'array deve avere *n*+1 celle

```
ENQUEUE(Q, x)
if Q.tail = Q.length and Q.head = 0 or Q.head = Q.tail + 1
   error "overflow"
Q[Q.tail] := x
if Q.tail = Q.length
   Q.tail := 1
else Q.tail := Q.tail + 1

DEQUEUE(Q)
if Q.head = Q.tail
   error "underflow"
x := Q[Q.head]
if Q.head = Q.length
   Q.head := 1
else Q.head := Q.head + 1
return x
```

tempo di esecuzione:
$T(n) = O(1)$ per entrambi

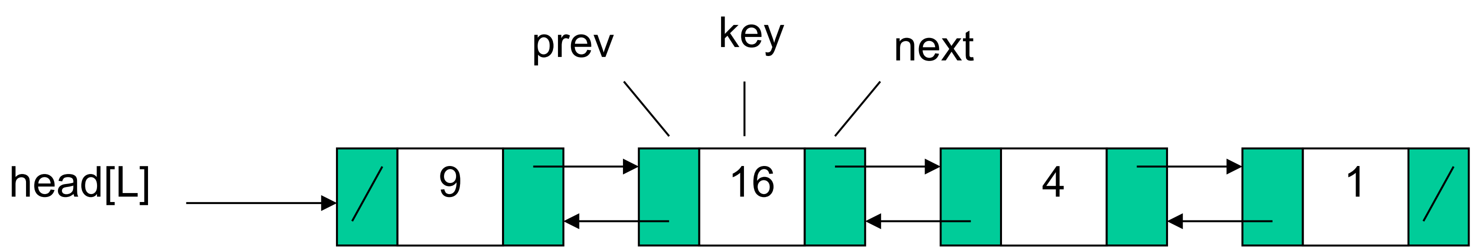
# Pile e code con liste semplici

- Molto facile realizzare una **pila** con lista semplice: lavoro sulla testa della lista: *push* inserisce e *pop* toglie
- **Coda**: leggermente più complicato:
- la testa (*head*) coincide con la testa della lista.
- la coda è la coda della lista: per fare *enqueue* e *dequeue* in tempo costante devo tenere un puntatore *tail* all'ultimo elemento della lista.
- Implementazione: provate come esercizio.

# Liste (doppiamente) concatenate

- Una lista concatenata  una struttura dati in cui gli elementi sono sistemati in un ordine lineare, in modo simile ad un array
  - l'ordine è dato non dagli indici degli elementi, ma da una “catena” di puntatori
- Una **lista doppiamente concatenata** è fatta di oggetti con 3 attributi:
  - *key*, che rappresenta il contenuto dell'oggetto
  - *next*, che è il puntatore all'oggetto seguente
    - cioè il successore dell'oggetto nell'ordinamento lineare
  - *prev*, che è il puntatore all'oggetto precedente
    - cioè il predecessore

- Se *x* è un oggetto nella lista, se *x*.*next* = *NIL*, *x* non ha successore
  - cioè è l'ultimo elemento della lista
- Se *x*.*prev* = *NIL*, *x* non ha predecessore
  - cioè è il primo elemento della lista, la *testa* (head)
- ogni lista *L* ha un attributo *L*.*head*, che è il puntatore al primo elemento della lista
- Esempio di lista doppiamente concatenata



- Altri tipi di liste:
  - concatenate in modo singolo
    - gli elementi non hanno il puntatore *prev*
  - ordinate
    - l'ordinamento degli elementi nella lista è quello delle chiavi
    - il primo elemento ha la chiave minima, l'ultimo la massima
  - non ordinate
  - circolari
    - il puntatore *prev* di *head* punta alla coda (*tail*), e il puntatore *next* della coda punta alla testa

# Operazioni su una lista doppiamente concatenata

- Ricerca
  - *input*: la lista *L* in cui cercare e la chiave *k* desiderata
  - *output*: il puntatore ad un elemento che ha *k* come chiave, *NIL* se la chiave non è nella lista

```
LIST-SEARCH(L, k)
x := L.head
while x ≠ NIL and x.key ≠ k
  x := x.next
return x
```

Nel caso pessimo (quando la chiave non è nella lista): $T(n) = \Theta(n)$

- Inserimento
  - *input*: la lista *L*, e l'oggetto *x* da aggiungere, inizializzato con la chiave desiderata
  - *output*: inserisce *x* all'inizio della lista *L*
    - (anche se un elemento con la stessa chiave esiste già nella lista)

```
LIST-INSERT(L, x)
x.next := L.head
if L.head ≠ NIL
  L.head.prev := x
L.head := x
x.prev := NIL
```

$T(n) = O(1)$

# Operazioni (2)

- Cancellazione
  - *input*: la lista *L*, e l'oggetto *x* da cancellare
    - si noti che non si passa come argomento la chiave da cancellare, ma l'oggetto (il nodo) stesso
  - *output*: cancella *x* dalla lista

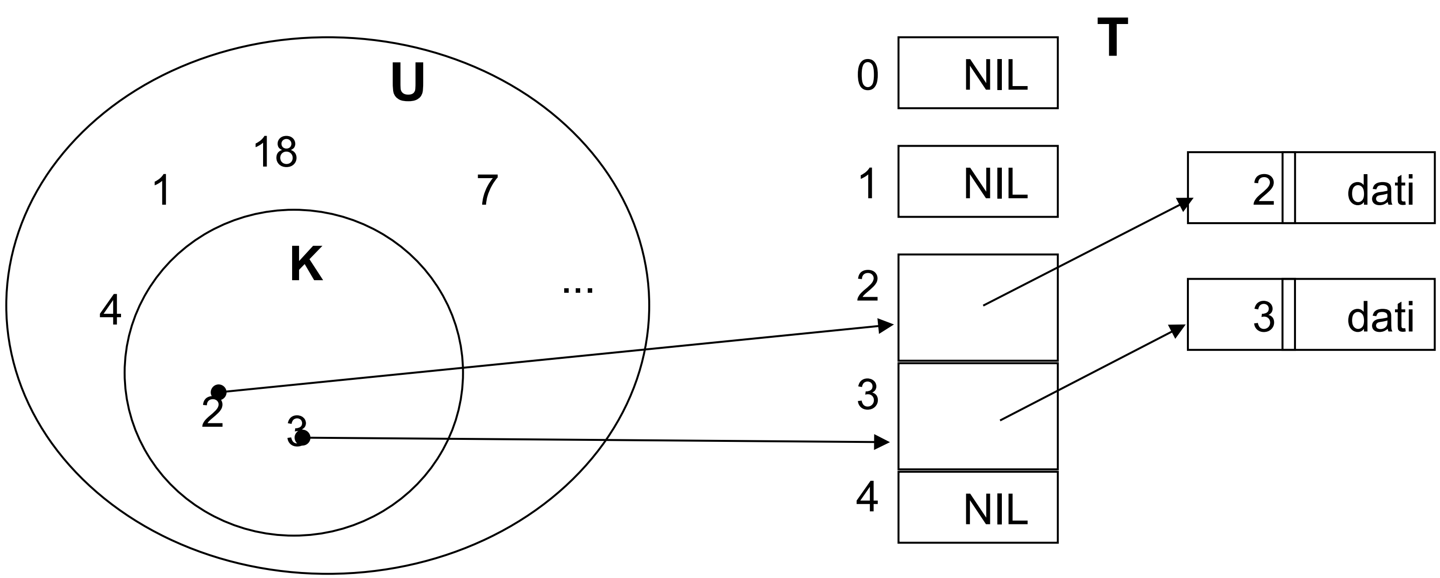
```
LIST-DELETE(L, x)
if x.prev ≠ NIL
  x.prev.next := x.next
else L.head := x.next
if x.next ≠ NIL
  x.next.prev := x.prev
```

T(n) = O(1)

# Dizionari e indirizzamento diretto

- Dizionario: insieme dinamico che supporta solo le operazioni di `INSERT`, `DELETE`, `SEARCH`
- Agli oggetti di un dizionario si accede tramite le loro chiavi
- Assumiamo che le chiavi siano numeri naturali (in caso contrario, si può sempre usare la rappresentazione in memoria come stringhe di bit)
- Se la cardinalità $m$ dell'insieme delle possibili chiavi $U$ ($m=|U|$) è ragionevolmente piccola, la maniera più semplice di realizzare un dizionario è tramite un array di $m$ elementi
  - con questo si ha l'*indirizzamento diretto*
  - in questo caso l'array si dice *tabella a indirizzamento diretto*

- Ogni elemento *T*[*k*] dell'array contiene il riferimento all'oggetto di chiave *k*, se un tale oggetto è stato inserito in tabella, `NIL` altrimenti



# Operazioni su una tabella a indirizzamento diretto

```
DIRECT-ADDRESS-SEARCH(T, k)
1  return T[k]

DIRECT-ADDRESS-INSERT(T, x)
1  T[x.key] := x

DIRECT-ADDRESS-DELETE(T, x)
1  T[x.key] := NIL
```

- Hanno tutte T(n) = O(1)
- Se il numero effettivamente memorizzato di chiavi è molto più piccolo del numero di chiavi possibili, c'è un sacco di spreco di spazio; potrebbe essere impossibile memorizzare la tabella in RAM

# Tabelle hash

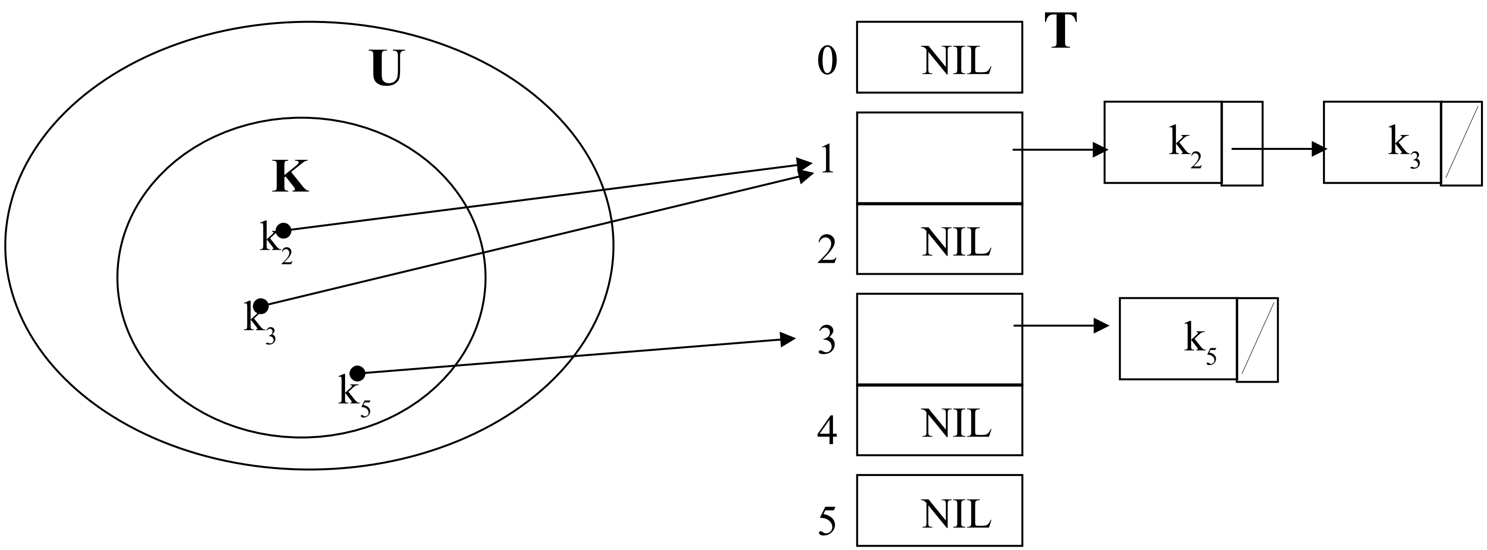
- Una tabella hash usa una memoria proporzionale al numero di chiavi *effettivamente memorizzate* nel dizionario
  - indipendentemente dalla cardinalità dell'insieme $U$ di chiavi
- Idea fondamentale: un oggetto di chiave $k$ è memorizzato in tabella in una cella di indice $h(k)$, con $h$ una **funzione hash**
  - se $m$ è la dimensione della tabella, $h$ è una funzione $h: U \rightarrow \{0..m\text{-}1\}$
  - la tabella $T$ ha $m$ celle, $T[0]$, $T[1]$, ... , $T[m\text{-}1]$
  - $h(k)$ è il valore hash della chiave k

- Problema: ho $|U|$ possibili chiavi ed una funzione che le deve mappare su un numero $m$ ($< |U|$, ma tipicamente $\ll |U|$) di slot della tabella
  - necessariamente avrò delle chiavi diverse (tante!) $k_1$, $k_2$ tali che $h(k_1)=h(k_2)$
  - in questo caso ho delle **collisioni**

- Ci sono diverse tecniche per risolvere le collisioni
- Una tipica è quella del *concatenamento* (chaining)

# Risoluzione di collisioni tramite concatenamento

- Idea della tecnica del concatenamento: gli oggetti che vengono mappati sullo stresso slot vengono messi in una lista concatenata



- Operazioni sulle tabelle in questo caso:

```
CHAINED-HASH-INSERT(T, x)
  inserisci x in testa alla lista T[h(x.key)]


CHAINED-HASH-SEARCH(T, k)
  cerca un elemento con chiave k nella lista T[h(k)]


CHAINED-HASH-DELETE(T, x)
  cancella x dalla lista T[h(x.key)]
```

- INSERT si fa in tempo O(1) (assumendo l'elemento da inserire non sia già in tabella)
- SEARCH si fa in tempo proporzionale alla lunghezza di $T[h(k)]$
- DELETE si fa in tempo O(1) se la lista è doppiamente concatenata
  - in input c'è l'oggetto da eliminare, non solo la chiave
  - se singolarmente concatenata, proporzionale alla lunghezza di $T[h(x.key)]$

# Analisi della complessità delle operazioni

- Nel caso pessimo, in cui tutti gli *n* elementi memorizzati finiscono nello stesso slot la complessità è quella di una ricerca in una lista di *n* elementi, cioè O(n)
- In media, però, le cose non vanno così male...
- Siano:
  - *m* la dimensione della tabella (il numero di slot disponibili)
  - $\alpha$ il **fattore di carico**, $\alpha$ = n/m
  - siccome $0 \leq n \leq |U|$ avremo $0 \leq \alpha \leq |U|/m$

- Ipotesi dell'**hashing uniforme semplice**:
  ogni chiave ha la stessa probabilità $1/m$ di finire in una qualsiasi delle $m$ celle di $T$, indipendentemente dalle chiavi precedentemente inserite
- Sotto questa ipotesi, la lunghezza media di una lista è

$$E[n_j] = \frac{1}{m}\sum_{i=1}^{m} n_i = \frac{n}{m} = \alpha$$

  quindi il tempo medio per cercare una chiave $k$ **non presente** nella lista è $\Theta(1+\alpha)$
  - 1 è il tempo per calcolare $h(k)$, che si suppone sia costante

- $\Theta(1+\alpha)$ è anche il tempo medio per cercare una chiave *k* che sia **presente** nella lista
  - la dimostrazione però richiede qualche calcolo in più.

- In pratica:
  - se $n = O(m)$, allora $\alpha = n/m = O(m)/m = O(1)$
  - quindi in media ci mettiamo un tempo costante
- Quindi la complessità temporale è $O(1)$ (in media) per tutte le operazioni (`INSERT`, `SEARCH`, `DELETE`)

# Funzioni hash

- Come scelgo una buona funzione hash *h*?
- In teoria, ne dovrei prendere una che soddisfa l'ipotesi di hashing uniforme semplice
  - per fare ciò, però, dovrei sapere quale è la distribuzione di probabilità delle chiavi che devo inserire
    - se le chiavi sono tutte "vicine", la funzione hash dovrebbe essere tale da riuscire a separarle
    - se invece so che le chiavi sono distribuite in modo uniforme in [0..K-1] mi basta prendere $h(k) = \lfloor (k/K)m \rfloor$
  - tipicamente si usano delle euristiche basate sul dominio delle chiavi
- Attenzione: tipica assunzione delle funzioni hash: la chiave *k* è un intero non-negativo (cioè è in $\mathbb{N}$)
  - facile convertire una qualunque informazione trattata da un calcolatore in un intero non-negativo, basta per esempio interpretare come tale la sequenza di bit corrispondente

# Metodo della divisione

- $h(k) = k \bmod m$
  - facile da realizzare e veloce (una sola operazione)
  - evitare certi valori di *m*:
    - potenze di 2 (m non deve essere della forma $2^p$)
      - se no *k* mod *m* sono solo i *p* **bit meno significativi** di *k*
      - meglio rendere *h*(*k*) dipendente da tutti i bit di *k*
  - spesso si prende per *m* un numero primo non troppo vicino ad una potenza esatta di 2
    - per esempio *m*=701, che ci darebbe, se *n*=2000, in media 3 elementi per lista concatenata

# Esempio

Prendiamo m = 5
e facciamo i seguenti inserimenti di chiavi: 38, 12, 18

Si ha h(38) = 3, h(12) = 2, h(18) = 3

Quindi:

T = [NIL, NIL, lista(12), lista(38, 18), NIL]

# Metodo della moltiplicazione

- Moltiplichiamo $k$ per una costante $A$ reale tale che $0 < A < 1$, quindi prendiamo la parte frazionaria di $kA$; il risultato lo moltiplichiamo per m, e ne prendiamo la parte intera
- Cioè:
  $h(k) = \lfloor m(kA \bmod 1) \rfloor$
  - in cui $x \bmod 1 = x - \lfloor x \rfloor$ è la parte frazionaria di $x$
- In questo caso il valore di $m$ non è critico, funziona bene con qualunque valore di $A$
  - spesso come $m$ si prende una potenza di 2 (cioè $m=2^p$), che rende semplice fare i conti con un calcolatore

- in questo caso, è utile prendere come $A$ un valore che sia della forma $s/2^w$, con $w$ dimensione della parola di memoria del calcolatore (chiaramente con $0 < s < 2^w$)
  - se $k$ sta in una sola parola ($k < 2^w$), $ks = kA2^w$ è un numero di $2w$ bit della forma $r_1 2^w + r_0$, ed i suoi $w$ bit meno significativi (cioè $r_0$) costituiscono $kA \bmod 1$
  - il valore di hash cercato (con $m=2^p$) è costituito dai $p$ bit più significativi di $r_0$

- Un valore di $A$ proposto da D. Knuth è:

$$A = (\sqrt{5} - 1) / 2$$

  - cioè l'inverso della *sezione aurea*
  - se si vuole applicare il calcolo precedente, occorre prendere come $A$ la frazione della forma $s/2^w$ più vicina all'inverso della sezione aurea
    - dipende dalla lunghezza della parola $w$

# Esempio

- A = 0.618
- Come esempio prendiamo m = 5 (anche se non è potenza di 2) e inseriamo: 38, 12, 18
- h(38) = 2, h(12) = 2, h(18) = 0
- Otteniamo:

  T = [lista(18), NIL, lista(38, 12), NIL, NIL]

# Indirizzamento aperto

- Un altro modo di evitare collisioni è tramite la tecnica dell'**indirizzamento aperto**
- In questo caso la tabella contiene *tutte le chiavi*, senza memoria aggiuntiva
  - quindi il fattore di carico $\alpha$ non potrà mai essere più di $1$

- L'idea è quella di calcolare l'indice dello slot in cui va memorizzato l'oggetto; se lo slot è già occupato, si cerca nella tabella uno slot libero
  - la ricerca dello slot libero però non viene fatta in ordine 0,1,2,...,m-1; la sequenza di ricerca (detta **sequenza di ispezione**) è un valore calcolato dalla funzione hash
    - dipende anche dalla chiave da inserire
    - la sequenza deve essere esaustiva, deve coprire tutte le celle

- La funzione hash ora diventa:
  $h : U \times \{0,1,..., m\text{-}1\} \rightarrow \{0,1,..., m\text{-}1\}$

- la sequenza di ispezione $\langle h(k, 0), h(k, 1),..., h(k, m\text{-}1)\rangle$ deve essere una permutazione di $\langle 0, 1, ..., \mathrm{m}\text{-}1\rangle$

# Operazioni in caso di indirizzamento aperto

- Inserimento di un oggetto:

```
HASH-INSERT(T, k)
i := 0
repeat
  j := h(k, i)
  if T[j] = NIL
    T[j] := k
    return j
  else i := i + 1
until i = m
error “hash table overflow”
```

# ricerca di un oggetto:

```
HASH-SEARCH(T, k)
i := 0
repeat
  j := h(k, i)
  if T[j] = k
    return j
  else i := i + 1
until T[j] = NIL or i = m
return NIL
```

- La cancellazione è più complicata, in quanto non possiamo limitarci a mettere lo slot desiderato a NIL, altrimenti non riusciremmo più a trovare le chiavi inserite dopo quella cancellata
  - una soluzione è quella di mettere nello slot, invece che NIL, un valore convenzionale come DELETED
  - però così le complessità non dipendono più dal fattore di carico

# Analisi di complessità

- Il tempo impiegato per trovare lo slot desiderato (quello che contiene la chiave desiderata, oppure quello libero in cui inserire l'oggetto) dipende (anche) dalla sequenza di ispezione restituita dalla funzione *h*
  - quindi dipende da come è implementata *h*
- Per semplificare un po' il calcolo facciamo un'ipotesi sulla distribuzione di probabilità con la quale vengono estratte non solo le chiavi, ma anche le sequenze di ispezione
- Ipotesi di **hashing uniforme**:
  ognuna delle *m*! permutazioni di (0,1, ... m-1) è ugualmente probabile che venga selezionata come sequenza di ispezione
  - è l'estensione dell'hashing uniforme semplice visto prima al caso in cui l'immagine sia non più solo lo slot in cui inserire l'elemento, ma l'intera sequenza di ispezione
- L'analisi viene fatta in funzione del fattore di carico $\alpha$ = n/m
  - siccome abbiamo al massimo un oggetto per slot della tabella, n $\leq$ m, e $0 \leq \alpha \leq 1$

- Sotto l'ipotesi di hashing uniforme valgono i seguenti risultati (i calcoli sono sul libro):
- il **numero medio** di ispezioni necessarie per effettuare l'**inserimento** di un nuovo oggetto nella tabella è
  - *m* se $\alpha = 1$ (cioè se la tabella è piena)
  - non più di $1/(1-\alpha)$ se $\alpha < 1$
- il **numero medio** di ispezioni necessarie per **trovare** un elemento presente in tabella è
  - $(m+1)/2$ se $\alpha = 1$
  - non più di $1/\alpha \log(1/(1-\alpha))$ se $\alpha < 1$

# Tecniche di ispezione

- In pratica, costruire funzioni hash che soddisfino l'ipotesi di hashing uniforme è molto difficile
- Si accettano quindi delle approssimazioni che, nella pratica, si rivelano soddisfacenti
- Tre tecniche:
  - ispezione lineare
  - ispezione quadratica
  - doppio hashing
    - nessuna di queste tecniche produce le m! permutazioni che sarebbero necessarie per soddisfare l'ipotesi di hashing uniforme
    - tuttavia, nella pratica si rivelano abbastanza buone
- Tutte e 3 le tecniche fanno uso di una (o più) **funzione hash ausiliaria** (ordinaria) $h'$: $U \rightarrow \{0,1,..., m\text{-}1\}$

# Ispezione lineare

- $h(k,i) = (h'(k)+i) \bmod m$
  - in questo caso l'ispezione inizia dalla cella $h'(k)$, e prosegue in $h'(k)+1$, $h'(k)+2$, ... fino a che non si arriva a $m$-1, quindi si ricomincia da 0 fino a esplorare tutti gli slot di $T$
  - genera solo $m$ sequenze di ispezione distinte
    - la prima cella ispezionata identifica la sequenza di ispezione
  - soffre del fenomeno dell'**addensamento** (clustering) **primario**
    - lunghe sequenze di celle occupate consecutive, che aumentano il tempo medio di ricerca

# Esempio

- Come prima prendiamo m = 5 ed inseriamo 38, 12, e 18
- Quindi: h(38,0) = 3, h(12,0) = 2, h(18,0) = 3 occupato! h(18,1) = 4
- Otteniamo quindi: T = [NIL, NIL, 12, 38, 18]
- Cancelliamo 38: T = [NIL, NIL, 12, DEL, 18]
- Inseriamo 43: h(43,0) = 3
- T = [NIL, NIL, 12, 43, 18]

# Ispezione quadratica

- Nel caso dell'**ispezione quadratica**:
  $h(k,i) = (h'(k)+c_1 i+c_2 i^2) \bmod m$
  - $c_1$ e $c_2$ sono costanti ausiliarie (con $c_2 \neq 0$)
  - $c_1$ e $c_2$ non possono essere qualsiasi, ma devono essere scelte in modo che la sequenza percorra tutta la tabella
  - ancora una volta, la posizione di ispezione iniziale determina tutta la sequenza, quindi vengono prodotte *m* sequenze di ispezione distinte
  - soffre del fenomeno dell'**addensamento secondario**: chiavi con la stessa posizione iniziale danno luogo alla stessa sequenza di ispezione

# Hashing doppio

- $h(k,i) = (h_1(k)+\mathrm{i}\ h_2(k)) \bmod m$
  - $h_1$ e $h_2$ sono funzioni hash ausiliarie
  - perché la sequenza prodotta sia una permutazione di (0, ..., m-1) $h_2(k)$ deve essere primo rispetto a $m$ (non deve avere divisori comuni tranne l'1)
    - posso ottenere questo prendendo come $m$ una potenza di 2, e facendo in modo che $h_2$ produca sempre un valore dispari
    - oppure prendendo come $m$ un numero primo, e costruendo $h_2$ in modo che restituisca sempre un valore < $m$
    - esempio:
      $h_1(k) = k \bmod m$
      $h_2(k) = 1 + (k \bmod m')$
      - con $m' < m$ (per esempio $m' = m$-1)
  - numero di sequenze generate ora è $\Theta(m^2)$ in quanto ogni coppia $(h_1(k), h_2(k))$ produce una sequenza di ispezione distinta