

CORrettezza e Complessità degli Algoritmi Iterativi.

Ci sono 4 caratteristiche fondamentali che ci interessano in un algoritmo:

- CORrettezza:** Restituisce sempre una risposta corretta

- Completezza:** Ogni risposta corretta viene prima o poi restituita

- Terminazione:**

- Complessità:** Definizione matematica del numero di passi necessari per eseguire un algoritmo, in termini del numero di elementi dell'input

```
proc InsertionSort (A)
for (j = 2 to A.length) => n - c1
    key = A[j] => (n - 1) · c2 (n-1) perché all'n-esimo
    i = j - 1 => (n - 1) · c3 (n-1) perché da for la condizione
    while ((i > 0) and (A[i] > key)) => c4 ·  $\sum_{t=0}^{i-1} t_j$ 
        { A[i + 1] = A[i] => c5 ·  $\sum_{t=0}^{i-1} t_j - 1$ 
        i = i - 1 => c6 ·  $\sum_{t=0}^{i-1} t_j - 1$ 
    A[i + 1] = key => (n - 1) · c7
```

Terminazione: autorizziamo i cicli

FOR: termina per $j > A.length$

WHILE: vincolato tra 0 e key

CORrettezza: tecnica dell'**INVARIANTE**

INvariante in for:

$A[1, \dots, j-1]$ è sempre ordinato

Stabilisco una proprietà di un ciclo, che sia vera prima, dopo e durante l'esecuzione

Complessità:

- Caso migliore: input già ordinato - $t_j = 1$ per $j = 2, \dots, n$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_4 \sum_{j=2}^n 1 \\ &= c_1 n + c_2 n - c_2 + c_3 n - c_3 + c_4 n - c_4 + c_4 n - c_4 \\ &= (c_1 + c_2 + c_3 + c_4) n + (-c_2 - c_3 - c_4 - c_4) = an + b = \Theta(n) \end{aligned}$$

$\Theta(f(n))$ se si ottiene da $f(n)$ eliminando tutti i termini di grado inferiore e le costanti.

- Caso peggiore: input in ordine inverso - $t_j = j$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^n j + c_5 \sum_{j=2}^n (j-1) + c_6 \sum_{j=2}^n (j-1) \\ &= an^2 + bn + c \rightarrow T(n) = \Theta(n^2) \end{aligned}$$

⚠ $\sum_{j=2}^n j = \frac{n(n+1)}{2} - 1$
 $\sum_{j=2}^n j-1 = \frac{n(n+1)}{2}$

CORrettezza e Complessità degli Algoritmi Ricorsivi:

array ordinato

```
proc RecursiveBinarySearch (A, low, high, k)
if (low > high)
    then return nil
mid = (high + low)/2
if (A[mid] = k)
    then return mid
if (A[mid] < k)
    then return RecursiveBinarySearch(A, mid + 1, high, k)
if (A[mid] > k)
    then return RecursiveBinarySearch(A, low, mid - 1, k)
```

L'elemento
che cerco

differenza se k è
prima o dopo la metà

Terminazione: ad ogni chiamata ricorsiva, la differenza tra low e high diminuisce

CORrettezza:

Invariante: se k è in A , allora è in $[low, high]$

Complessità: $T(n) = T(n/2) + 1$

?

Ricorrenza

NOTAZIONE ASINTOTICA:

Per una funzione $f(n)$ diremo che è **limitata dall'alto** da $g(n)$: $f(n) = O(g(n))$
 se e solo se esiste una costante tale che per qualche n_0 , per tutti gli $n \geq n_0$ e' il caso che:

$$0 \leq f(n) \leq c g(n)$$

In maniera simile $f(n)$ è **limitata dal basso** da $g(n)$: $f(n) = \Omega(g(n))$

Se e solo se:

$$0 \leq c \cdot g(n) \leq f(n)$$

Diremo invece che $f(n)$ è dello stesso ordine di $g(n)$: $f(n) = \Theta(g(n))$

Se e solo se:

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$$

→ $f(n) = \Theta(g(n))$ se e solo se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$

SOLUZIONI DI RICORRENZE:

Per ogni $T(n)$ esplicita possiamo trovare l'ordine di grandezza, ma prima dobbiamo esplicitare la ricorrenza, ci sono tre metodi:

1. ALBERO DI SVILUPPO
2. Master Theorem
3. METODO DELLA SOSTITUZIONE

1. SVILUPPO DA RICORRENZA PER ESTRAERNE IL COMPORTAMENTO:

$$\begin{cases} T(n) = T(n/2) + 1 & \text{ma } T(n/2) = T(n/4) + 1 \\ \Rightarrow T(n) = (T(n/4) + 1) + 1 & \text{ma } T(n/4) = T(n/8) + 1 \\ \Rightarrow T(n) = ((T(n/8) + 1) + 1) + 1 & \dots \end{cases}$$

↳ finché l'argomento di T diventa 1 o meno
Assumendo che $T(1) = 1 \rightarrow T(n) = \Theta(\log(n))$

$$\begin{array}{c} \text{GASTO:} \\ \left[\begin{array}{c} T(n) \\ T(n/2) \\ T(n/4) \\ T(n/8) \\ \vdots \\ T(1) \end{array} \right] \\ \xrightarrow{\quad n \quad} \\ \left[\begin{array}{c} 1 \\ 2 \\ 2 \\ 2 \\ \vdots \\ 1 \end{array} \right] \\ \xrightarrow{\quad n \quad} \\ n = \log(n) \end{array}$$

Esempio:
 $T(n) = 2T(n/2) + n$
 $= 2(2T(n/4) + n) + n$
 $= 2(2(2T(n/8) + n) + n) + n$
 \vdots
 $= 2^k T(n/2^k) + 2^k n$
 $T(n/2^k) = T(n/2^{k-1}) + 1$
 \vdots
 $T(1) = T(1) + 1 = T(1) = T(1) = \dots = T(1) = n$
 $n = \log(n)$

2. MASTER THEOREM.

Se $T(n) = aT(\frac{n}{b}) + f(n)$, allora:

$$T(n) = \begin{cases} \Theta(n^{\log_b(a)}) & \text{se } f(n) = O(n^{\log_b(a) - \varepsilon}) \text{ con } \varepsilon > 0 \\ \Theta(n^{\log_b(a)} \lg^{k+1}(n)) & \text{se } f(n) = \Theta(n^{\log_b(a)} \lg^k(n)) \text{ con } k \geq 0 \\ \Theta(f(n)) & \text{se } \exists c < 1 \text{ a } f\left(\frac{n}{b}\right) \leq c \cdot f(n) \quad \forall n > n_0 \end{cases}$$

3. SOSTITUZIONE:

Si indovina un risultato \rightarrow si verifica attraverso l'induzione

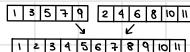
ALGORITMI DI ORDINAMENTO NON ELEMENTARI:

I metodi per l'ordinamento degli array possono essere:

Mergesort: formato da Merge e MergeSort

```
proc Merge(A, p, q, r)
  n1 = q - p + 1
  n2 = r - q
  let L[1, ..., n1] and R[1, ..., n2] be new array
  for (i = 1 to n1) L[i] = A[p + i - 1]
  for (j = 1 to n2) R[j] = A[q + j]
  i = 1
  j = 1
  for (k = p to r)
    if (i ≤ n1)
      then
        if (j ≤ n2)
          then
            if (L[i] ≤ R[j])
              then CopyFromL(i)
            else CopyFromR(j)
          else CopyFromR(j)
        else CopyFromL(i)
```

SelectionSort
 $T(n) = \Theta(n^2)$



elementari: ogni elemento viene spostato e messo al posto giusto separata mente dagli altri, in modo iterativo o ricorsivo.

non elementari: non fanno uso di ipotesi aggiuntive

Merge costruisce una sequenza ordinata partendo da due sequenze ordinate

Merge non è in place \leftarrow quanto è più grande l'input maggiore è lo spazio allocato

Invariante: $A[p, \dots, k-1]$ contiene i $k-p$ elementi più piccoli di $L[1, \dots, n_1]$ e di $R[1, \dots, n_2]$ ordinati; $L[i]$ e $R[j]$ sono i più piccoli elementi di L ed R non ancora ricoppiati in A

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

Invariante: Dopo ogni chiamata ricorsiva $A[p, r]$ è ordinato.

Complessità: $T(n) = 2T(\frac{n}{2}) + \Theta(n)$
 $= \Theta(n \lg(n))$

QuickSort: formato da Partition e Quicksort.

Prendiamo un elemento qualsiasi e riordiniamo l'array in modo che gli elementi più grandi stiano a destra ed i più piccoli a sinistra, in ordine casuale

Posizione del pivot

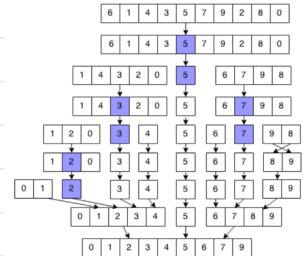
```
proc Partition (A, p, r)
  x = A[r] ← pivot
  i = p - 1
  for (j = p to r - 1)
    if (A[j] ≤ x)
      then
        { i = i + 1
          SwapValue(A, i, j)
        }
        SwapValue(A, i + 1, r)
  return i + 1 ← tra la sz di dx e quella di sx
```

Sceglieremo sempre arbitrariamente l'ultimo elemento dell'array come pivot.

Invariante:

- ① Se $p \leq k \leq i$, allora $A[k] \leq x$ - diviso la parte sx
- ② Se $i+1 \leq k \leq j-1$ allora $A[k] > x$ - diviso la parte dx
- ③ Se $k = r$, allora $A[k] = x$ - e' il pivot

Complessità: $T(n) = \Theta(n)$



```
proc Quicksort (A, p, r)
  if (p < r)
    then
      { q = Partition(A, p, r)
        Quicksort(A, p, q - 1)
        Quicksort(A, q + 1, r)
      }
```

Invariante: al termine di ogni chiamata ricorsiva su p, r , $A[p, \dots, r]$ è ordinato

Complessità:

① Caso peggiore: partizione sfiduciata ← a causa della scelta del pivot, ad ogni passo si crea una partizione di dimensione 0 ed una di dimensione dim-1
 $T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$

② Caso migliore: $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) = \Theta(n \lg(n))$

③ Partizione molto iniqua: 9 a 1 $T(n) \leq T\left(\frac{9}{10}n\right) + T\left(\frac{n}{10}\right) + \Theta(n) = O(n \lg(n))$

Randomizzo la scelta del pivot per garantire che tutti gli input siano egualmente probabili
 \hookrightarrow RandomizedQuicksort() ← caso medio: $T(n) = O(n \lg(n))$

CountingSort e RadixSort:

Un algoritmo si dice basato sui confronti se ogni passo può essere visto come una operazione di confronto fra due elementi, seguita da uno spostamento

\hookrightarrow Il processo di ordinamento può essere visto come un albero binario la cui radice è l'input, ad ogni nodo si associa una permutazione dell'oggetto.

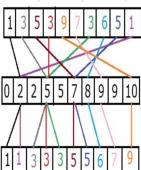
Per n elementi sono possibili $n!$ permutazioni \rightarrow Il limite inferiore alla lunghezza massima di un ramo dell'albero è $\log(n!) = \Theta(n \lg(n))$

Il limite dell'ordinamento basato sui confronti è che esiste una complessità minima: $\Theta(n \lg(n))$

```
proc CountingSort (A, B, k)
  let C[1..k] new array
  for (i = 0 to k) C[i] = 0
  for (j = 1 to A.length) C[A[j]] = C[A[j]] + 1
  for (i = 1 to k) C[i] = C[i] + C[i - 1]
  for (j = A.length downto 1)
    { B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
    }
```

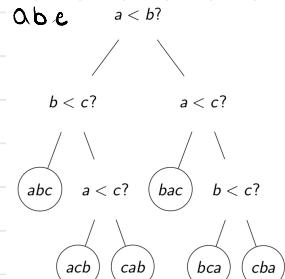
E' un algoritmo non basato sui confronti. Dovrebbe avere delle ipotesi aggiuntive:

L'input è un intero n :
 $0 \leq n \leq k$ con k intero



Invariante: $C[A[j]]$ è sempre la posizione corretta di $A[j]$ in B

Complessità: se $k = O(n)$ allora $T(n) = \Theta(n)$, se $k \gg n$ la complessità non è più lineare

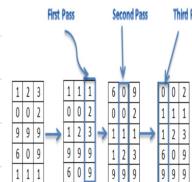


RadixSort è un meta-algoritmo e si usa per ordinare elementi multi indice
si basa su un algoritmo interno di ordinamento

```
proc RadixSort (A, d)
{for (i = 1 to d) AnyStableSort(A) on digit i
    ↓
    ↓ dal meno al più significativo
```

Invariante: dopo la i -esima esecuzione del ciclo gli elementi delle ultime i -colonne sono ordinati

	Caso Peggiora	Caso Medio	In Place	Stabile
IS	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓
HS	$\Theta(n \lg n)$	$\Theta(n \lg n)$		✓
QS	$\Theta(n^2)$	$\Theta(n \lg n)$	✓	
CS	$\Theta(n)$	$\Theta(n)$		✓
RS	$\Theta(n)$	$\Theta(n)$		✓



→ sotto opportune ipotesi
← CountingSort come algoritmo interno

Liste:

la tassonomia classica delle strutture dati prevede 3 dimensioni ortogonali tra loro.
statica

una struttura dati può essere:
e dinamica ← struttura pensata per aggiungere e togliere elementi durante l'esecuzione
e compatta
e sparsa ← non possiamo fare assunzioni sulla posizione fisica degli elementi in memoria
e può essere basata o non sull'ordinamento delle chiavi

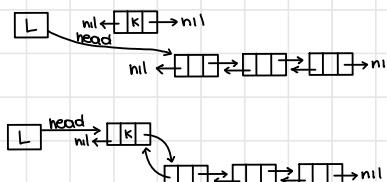
Array e liste sono considerate strutture dati concrete

una lista concatenata è una struttura dati dinamica, non basata sull'ordinamento e sparsa

↳ versione dinamica dell'array

Liste non ordinate, doppialmente collegate:

```
proc ListInsert (L, x)
    {elemento da inserire
     indirizzo della lista}
    x.next = L.head
    if (L.head ≠ nil)
        then L.head.prev = x
    L.head = x
    x.prev = nil
    Θ(1)
```



Inserimento
in testa

```
proc ListSearch (L, k)
    {x = L.head
     while (x ≠ nil) and (x.key ≠ k) x = x.next
     return x
    Θ(n)}
```

```
proc ListDelete (L, x)
    {if (x.prev ≠ nil)
     then x.prev.next = x.next
     else L.head = x.next
     if (x.next ≠ nil)
     then x.next.prev = x.prev
    Θ(1)}
```

Pile e code: sono strutture dati astratte, dinamiche e non basate sull'ordinamento

Possono essere entrambe implementate in maniera compatta (basate su array), o sparsa (basate su liste).

La loro caratteristica distintiva è che l'accesso agli elementi non è libero, ma vincolato ad una politica.

La ragione per cui può essere utile avere una politica di accesso è che questa può fare risparmiare dei dettagli implementativi, assicurando un certo ordine di inserimento ed estrazione.

Per implementare una pila ci possiamo basare su un array; avremo quindi una implementazione compatta.

L'idea è quella di mascherare la struttura portante all'utente finale.

Le pile implementano una politica LIFO

controllare se la pila è vuota

```
proc Empty (S)  
{ if (S.top = 0)  
    then return true  
return false
```

proc Push (S, x) inserimento

```
{ if (S.top = S.max)  
    then return "overflow"  
S.top = S.top + 1  
S[S.top] = x
```

proc Pop (S) eliminazione

```
{ if (Empty(S))  
    then return "underflow"  
S.top = S.top - 1  
return S[S.top + 1]
```

O(1)

O(1)

S è un array strutturato, dotato dei parametri

S.top inizializzato a 0
numero di elementi nulla pila

S.max che indica la sua massima capacità

Una coda (o queue) è una struttura dati elementare che implementa una politica FIFO.

O(1)

```
proc Enqueue (Q, x)  
{ if (Q.dim = Q.length)  
    then return "overflow"  
Q[Q.tail] = x  
if (Q.tail = Q.length)  
    then Q.tail = 1  
else Q.tail = Q.tail + 1  
Q.dim = Q.dim + 1
```

proc Dequeue (Q)

```
{ if (Q.dim = 0)  
    then return "underflow"  
x = Q[Q.head]  
if (Q.head = Q.length)  
    then Q.head = 1  
else Q.head = Q.head + 1  
Q.dim = Q.dim - 1  
return x
```

Q è un array dotato di struttura, con parametri

Q.head inizializzato a 0
Q.tail inizializzato a 0
Q.head = Q.tail = 1
dim inizializzato a 0

per le pile su liste:

```
proc Empty (S)  
{ if (S.head = nil )  
    then return true  
return false
```

proc Push (S, x)
{ Insert(S, x)

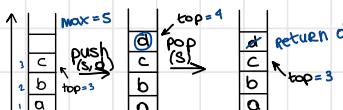
```
proc Pop (S)  
{ if (Empty(S))  
    then return "underflow"  
x = S.head  
Delete(S, x)  
return x.key
```

O(1)

```
proc Empty (Q)  
{ if (Q.head = nil )  
    then return true  
return false
```

proc Enqueue (Q, x)
{ Insert(Q, x)

```
proc Dequeue (Q)  
{ if (Empty(Q))  
    then return "underflow"  
x = Q.tail  
Q.tail = x.prev  
Delete(Q, x)  
return x.key
```



Heap, code di priorità e HeapSort

L'heap è una struttura dati astratta, basata sull'ordinamento e

necessariamente compatta → basata su array

Una heap mantiene le chiavi semi-ordinate

Una (min/max) heap è un array H che può essere visto come un albero binario quasi completo.

L'elemento $H[1]$ dell'array è la radice dell'albero e, normalmente, si tende a differenziare i valori

$H.length$ (lunghezza dell'array che contiene una heap) e $H.heapsize$ (numero di elementi della heap contenuta in H).

Tipicamente si ha che $0 \leq H.heapsize \leq H.length$.

tutti i livelli pieni meno eventualmente l'ultimo



proc Parent (i)

{return $\lfloor \frac{i}{2} \rfloor$ }

Ricordiamo però che in questo caso la posizione minima in un array è 1 e non 0.

proc Left (i)
{return $2 \cdot i$ }

proc Right (i)
{return $2 \cdot i + 1$ }

La corretta implementazione di una heap prevede che i figli di un nodo nella posizione i siano precisamente gli elementi nelle posizioni $2 \cdot i$ e $2 \cdot i + 1$ (sinistro e destro rispettivamente).

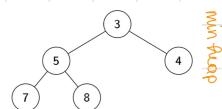
Per conseguenza, il padre di un nodo i è identificato dall'indice $\lfloor \frac{i}{2} \rfloor$.

Distinzione tra max heap e min heap

Entrambe soddisfano una proprietà: nel primo caso, abbiamo che per ogni i , $H[\text{Parent}(i)] \geq H[i]$, e nel secondo caso, per ogni i , $H[\text{Parent}(i)] \leq H[i]$.

Conseguentemente, il massimo elemento di una max-heap si trova alla radice, mentre nel caso di una min-heap è il minimo elemento a trovarsi alla radice.

Vista come un albero binario, l'altezza di una heap è la lunghezza del massimo cammino dalla radice ad una foglia.



In generale: una heap di altezza h contiene al minimo 2^h elementi, ed al massimo $2^{h+1} - 1$.

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

```
proc MinHeapify (H, i)
  l = Left(i)
  r = Right(i)
  smallest = i
  if ((l ≤ H.heapsize) and (H[l] < H[i]))
    then smallest = l
  if ((r ≤ H.heapsize) and (H[r] < H[smallest]))
    then smallest = r
  if smallest ≠ i
    then
      {SwapValue(i, smallest)
      MinHeapify(H, smallest)}
```

Dato un array H , ed un indice i su di esso tale che $H[\text{Left}(i)]$ e $H[\text{Right}(i)]$ sono già delle min-heap, MinHeapify trasforma H in un array tale che anche $H[i]$ è una min-heap.

Invariante ricorsiva: dopo ogni chiamata a MinHeapify su un nodo di altezza h tale che entrambi i figli sono radici di min-heap prima della chiamata, quel nodo è la radice di una min-heap.

Complessità:

caso peggiore: $\Theta(n) = \Theta(\log(n))$

caso migliore: $\Theta(1)$ ← nessuna chiamata ricorsiva

Dato un array H di interi voglio convertirlo in una min-heap

```
proc BuildMinHeap (H)
  H.heapsize = H.length
  {for (i = ⌊ H.length / 2 ⌋) downto 1) MinHeapify(H, i)}
```

Invariante: all'inizio di ogni iterazione del ciclo for, ogni elemento $H[i + 1], H[i + 2], \dots$ è la radice di una min-heap, e all'uscita dall'iterazione, anche $H[i]$ lo è.

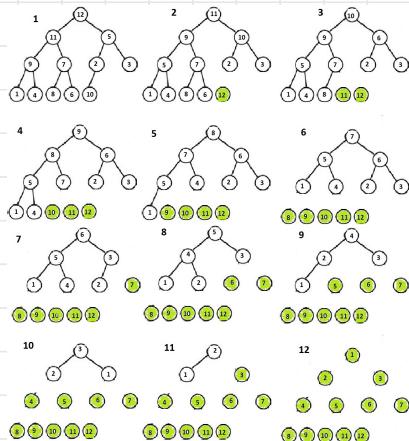
Complessità: $\Theta(n)$

```

proc HeapSort (H)
  BuildMaxHeap(H)
  for ( $i = H.length$  downto 2)
    {
      SwapValue( $i, 1$ ) scambio i valori di H[1] e H[i]
       $H.heapsize = H.heapsize - 1$ 
      MaxHeapify( $H, 1$ )
    }
  
```

COMPLESSITÀ : $\Theta(n \log(n))$

	Caso peggiore	Caso Medio	In place	Stabile
IS	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓
HS	$\Theta(n \lg(n))$	$\Theta(n \lg(n))$		✓
QS	$\Theta(n^2)$	$\Theta(n \lg(n))$	✓ *	**
CS	$\Theta(n)$	$\Theta(n)$		✓
RS	$\Theta(n)$	$\Theta(n)$		✓
IIS	$\Theta(n \lg(n))$	$\Theta(n \lg(n))$	✓ *	**



* In place a meno delle chiamate ricorsive (tail recursive)

** Tutti gli algoritmi di ordinamento possono essere resi stabili

> sotto opportune ipotesi

← CountingSort come algoritmo interno

Code di priorità:

Una coda di priorità è una struttura dati astratta basata sull'ordinamento e necessariamente compatta.

Possiamo costruire una coda di priorità basandoci su una min-heap.

A differenza di una coda classica, che implementa una politica FIFO, una coda di priorità associa ad ogni elemento una chiave, la priorità, e serve (cioè estrae) l'elemento a priorità più bassa. Questa estrazione è associata ad una operazione che aggiusta la struttura dati, ed anche alla possibilità di inserire nuovi elementi, o cambiare la priorità di un elemento inserito. Sia quindi Q una min-heap senza campi aggiuntivi.

```

proc Enqueue (Q, priority)
  if ( $Q.heapsize = Q.length$ )
    then return "overflow"
   $Q.heapsize = Q.heapsize + 1$ 
   $Q[Q.heapsize] = \infty$ 
  DecreaseKey(Q, Q.heapsize, priority)
  
```

```

proc DecreaseKey (Q, i, priority)
  if ( $priority > Q[i]$ )
    then return "error"
   $Q[i] = priority$ 
  while ( $(i > 1)$  and ( $Q[Parent(i)] > Q[i]$ ))
    {
      SwapValue(i, Parent(i))
       $i = Parent(i)$ 
    }
  
```

```

proc ExtractMin (Q)
  if ( $Q.heapsize < 1$ )
    then return "underflow"
   $min = Q[1]$ 
   $Q[1] = Q[Q.heapsize]$ 
   $Q.heapsize = Q.heapsize - 1$ 
  MinHeapify(Q, 1)
  return min
  
```

$\Theta(\log(n))$

Code di priorità su array:

Q è un array (che immaginiamo sempre con un campo Q.length), e ogni posizione i ha tre valori: i è stesso (la chiave), Q[i] (la sua priorità), e Q[i].empty (che ci indica se l'elemento a chiave i è stato virtualmente cancellato), e immaginiamo di aver inizializzato tutti i valori Q[i].empty a falso (partiamo da una situazione in cui ci sono tutti). In questo modo, abbiamo fatto l'equivalente della costruzione della coda: ogni elemento (chiave) ha la sua priorità e si trova dentro la coda. Il costo di questa inizializzazione, il cui codice non vediamo, è ovviamente di $\Theta(n)$.

	array (c. peggiore e medio)	heaps (c. peggiore e medio)
inizial.	$\Theta(n)$	$\Theta(n)$
inserimento	$\Theta(1)$	$\Theta(\log(n))$
decremento	$\Theta(1)$	$\Theta(\log(n))$
estr. minimo	$\Theta(n)$	$\Theta(\log(n))$

```

proc ExtractMin (Q)
  MinIndex = 0
  MinPriority =  $\infty$ 
  for ( $i = 1$  to Q.length)
    if (( $Q[i] < MinPriority$ ) and ( $Q[i].empty = 0$ ))
      {
         $MinPriority = Q[i]$ 
         $MinIndex = i$ 
        if ( $MinIndex = 0$ )
          then return "underflow"
         $Q[MinIndex].empty = 1$ 
        return MinIndex
      }
  
```

```

proc Enqueue (Q, i, priority)
  if ( $i > Q.length$ )
    then return "overflow"
   $Q[i] = priority$ 
  Θ(1)
  
```

```

proc DecreaseKey (Q, i, priority)
  if (( $Q[i] < priority$ ) or ( $Q[i].empty = 1$ ))
    {
      then return "error"
       $Q[i] = priority$ 
    }
  Θ(1)
  
```

Tabelle hash.

Una tabella hash è una struttura dati astratta.

Nella nostra implementazione, le tabelle hash sono dinamiche, parzialmente compatte, e non basate sull'ordinamento.

Una tabella hash ad accesso diretto T è un semplice array di puntatori ad oggetti; $T[key]$ punta all'oggetto la cui chiave assegnata è key. Diciamo che gli oggetti sono denotati con x, y, \dots , e che per ognuno di essi il campo key è la sua chiave. Per una tabella ad accesso diretto stiamo immaginando che:

$x.key$ è sempre piccolo (se chiamiamo m la dimensione della tabella, questo vuol dire $x.key \leq m$ per ogni x)
e $x.key \neq y.key$ per ogni coppia $x \neq y$.

Purtroppo, nella realtà, per garantire queste proprietà bisogna usare valori di m troppo grandi.

Per m molto grande, anche operazioni elementari prendono troppo tempo.

Per risolvere questo problema usiamo abbandoniamo l'accesso diretto e creiamo una cosiddetta funzione hash, che ci permette di indirizzare l'elemento k alla posizione $h(k)$. Ovviamente h non può essere iniettiva.

Quando $k \neq k'$ e $h(k) = h(k')$ la situazione si dice di **confitto**.

Funzioni hash chiavi dall'universo diventano comprese tra 1 e m

La tecnica chiamata chaining risolve i conflitti utilizzando una lista doppiamente collegata. La testa della lista è memorizzata in $T[h(k)]$, che quando è vuota contiene nil.



hash Search ha complessità $\Theta(n)$ nel caso peggiore, ma in quel caso non lo usiamo
caso medio: $\Theta(1 + \frac{n}{m})$

```

proc HashInsert (T, k) in testa
{ let x be a new node with key k
  i = h(k)
  ListInsert(T[i], x) } Θ(1)

proc HashSearch (T, k)
{ i = h(k)
  return ListSearch(T[i], k) }

proc HashDelete (T, k) Θ(1)
{ i = h(k)
  x = ListSearch(T[i], k)
  ListDelete(T[i], x) }
  
```

```

proc HashComputeModulo (w, B, m) Θ(d)
{ let d = |w|
  z0 = 0
  for (i = 1 to d) zi+1 = ((zi · B) + ai) mod m
  return zd + 1 }
  
```

HashComputeModulo prende i parametri $w = a_1 a_2 \dots$ ad (parola qualsiasi di un alfabeto), B (base, cioè dimensione dell'alfabeto) e m (numero primo tale che $(m + 1) \cdot B$ possa essere memorizzato in una parola di memoria).

Concludiamo adesso studiando una tecnica chiamata open hashing che ha le seguenti caratteristiche importanti: si eliminano le liste e quindi il chaining: una tabella hash di m elementi potrà ospitare al massimo m elementi, e, per ottenere questo risultato, si rinuncia (in pratica) ad implementare la funzione di cancellazione.

L'indirizzamento aperto si basa sull'idea di provare più di una posizione sulla tabella, finché se ne trova una libera oppure si ha la certezza che la tabella è piena.

Data una chiave k da inserire in una tabella ad indirizzamento aperto, possiamo usare una funzione di hash qualsiasi tra quelle viste precedentemente, per ottenere una posizione $h(k)$. Se questa è libera, la chiave può essere inserita. Se invece la posizione è già occupata, proviamo un'altra posizione: questa sequenza di tentativi si chiama sequenza di probing. Non tutte le sequenze di probing funzionano bene: la condizione è che tutte le posizioni della tabella devono essere provate prima o poi.

Questa condizione è chiamata hashing uniforme e generalizza la condizione di hashing uniforme semplice.

```

proc OpenHashInsert (T, k)
{ i = 0
  repeat
    { j = h(k, i)
      if T[j] = nil
      then
        { T[j] = k
          return j
        }
      else
        { i = i + 1
        }
    until (i = m)
  return "overflow" }
  
```

```

proc OpenHashSearch (T, k)
{ i = 0
  repeat
    { j = h(k, i)
      if T[j] = k
      then return j
      i = i + 1
    }
  until ((T[j] = nil) or (i = m))
  return nil }
  
```

FUNZIONI HASH !

- MOLTIPLICAZIONE
- DIVISIONE
- ADDITIONE
- PROBING LINEARE
- PROBING QUADRATICO

Insiemi disgiunti

Gli insiemi disgiunti sono una struttura dati astratta, parzialmente dinamica, parzialmente sparsa e non basata sull'ordinamento.

La caratteristica principale di un insieme di insiemi disgiunti è che le operazioni ad esso associate sono, tipicamente: **MakeSet**, che costruisce un nuovo insieme disgiunto; **Union**, unisce due insiemi disgiunti in uno solo; e **FindSet**: trova il rappresentante dell'insieme al quale appartiene l'elemento.

Ogni insieme è quindi dotato di un elemento rappresentativo (che può essere uno qualsiasi).

Gli insiemi crescono solo in due modi: quando vengono creati (e contengono esattamente un elemento), e quando vengono uniti due insiemi in uno solo che contiene gli elementi di entrambi.

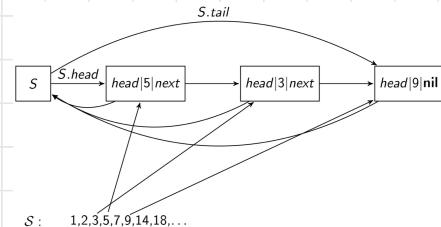
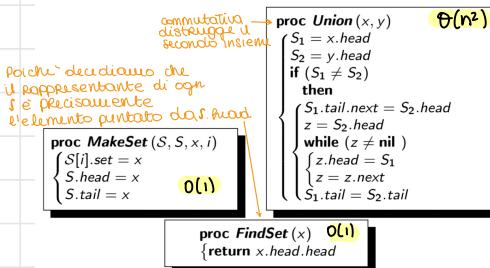
Implementazione tramite liste.

L'elemento $S \in S$ è quindi una lista dotata degli attributi

$S.head$ (che punta al primo elemento) e $S.tail$ (che punta all'ultimo elemento). Ogni elemento x è dotato di $x.next$ (come sempre) e $x.set$ che punta all'insieme S che lo contiene.

ci permette di ricostituire la sua struttura

In questa versione, l'informazione aggiuntiva che contiene ogni $S[i]$ è un puntatore all'elemento i in memoria, cioè alla casella x che contiene la chiave i . Lo chiamiamo $S[i].set$.

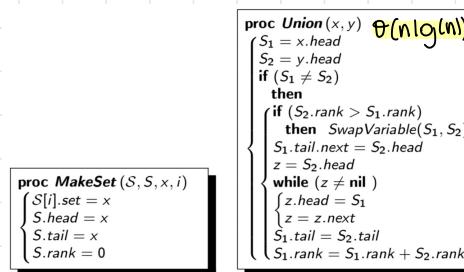


In ogni operazione, gli elementi sono ipotizzati già creati e nella memoria principale. Creare un nuovo insieme (**MakeSet**) significa: creare un oggetto **S** (che noi ipotizziamo già esistente), creare un oggetto **x** con la chiave che vogliamo (che noi ipotizziamo già creato), e collegarli.

L'operazione di **Union** di **x** e **y** consiste nel trovare **S1** ed **S2** (rispettivamente, gli insiemi di **x** e **y**), e, se sono diversi, aggiornare **S1.tail.next** a **S2.head** e, per ogni **z** tale che **z.head = S2**, impostare **z.head = S1**.

Liste con unione pesata

Una prima strategia che possiamo usare per migliorare la situazione è chiamata unione pesata. Il principio sul quale si basa è semplice: se manteniamo in ogni insieme **S** anche il numero degli elementi dell'insieme, allora possiamo implementare **Union** in maniera che gli aggiornamenti dei puntatori si facciano sempre sull'insieme più piccolo.



Foreste di alberi

La rappresentazione è basata in alberi piuttosto che liste. Un nodo x (un elemento) contiene le seguenti informazioni: $x.p$ (il padre), e $x.rank$ (un limite superiore all'altezza del sotto-albero radicato in x). Gli alberi (gli insiemi disgiunti) sono k-ari, e formano una foresta S .

Un nodo di un albero k-ario non ha, in generale, nessun puntatore ai figli. Infatti, non abbiamo un limite superiore a quanti figli possiamo avere e, cosa più importante, non ci interessa agli scopi di questa struttura dati.

Il rango non è la misura dell'altezza, ma, come detto sopra, un suo limite superiore.

L'operazione di unione, in due fasi, consiste, come prima, nel trovare i rappresentanti degli elementi utilizzati come indici; se le due radici sono x e y , rispettivamente, si sceglie quello il cui rango sia inferiore, e si aggiorna solo il padre, rendendolo uguale all'altro elemento. Con il criterio di unione per rango (il corrispondente dell'unione pesata nella versione con le liste), il rango dell'insieme risultante cambia (e si incrementa) solo se i ranghi dei due componenti erano uguali, e rimane inviarciato (uguale al massimo tra i due) negli altri casi.

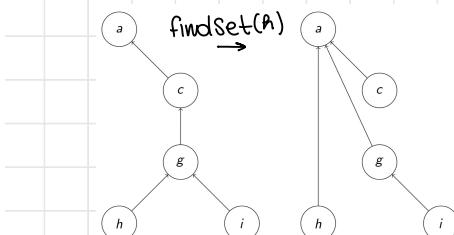
```
proc Union (x, y) O(m)
  x = Findset(x)
  y = Findset(y)
  if (x.rank > y.rank)
    then y.p = x
  if (x.rank ≤ y.rank)
    then
      x.p = y
      if (x.rank = y.rank)
        then y.rank = y.rank + 1
```

```
proc MakeSet (x)
  {x.p = x
   x.rank = 0}
```

```
proc FindSet (x)
  {if (x ≠ x.p)
   then x.p = FindSet(x.p)
   return x.p}
```

L'operazione FindSet diventa attiva.

Non solo restituisce, come sempre, il rappresentante, ma, mentre scorre i puntatori verso l'alto alla sua ricerca, li aggiorna appiattendo il ramo al quale appartiene. Chiamiamo compressione del percorso questa strategia.



Alberi:

Gli alberi sono strutture dati fondamentali dinamiche e sparse. A seconda degli usi che se ne fanno, possono essere basate sull'ordinamento oppure no. Da un lato, possiamo dire che gli alberi generalizzano le liste. In questo senso, possiamo anche dire che i grafi, che vedremo più avanti, a loro volta generalizzano gli alberi.

→ nelle liste i puntatori next
e uno solo, mentre i nodi di un albero possono avere
più figli.

Un albero radicato è un grafo aciclico connesso tale che ogni coppia di vertici è connessa

da al più un cammino. Un albero è k-ario se ogni nodo ha al più k figli distinti (albero binario).

Un albero può essere:

completo: ogni livello è completo, cioè tutti i nodi di uno stesso livello hanno esattamente zero o due figli;

quasi completo: ogni livello, tranne eventualmente l'ultimo, è completo;

bilanciato: tra il percorso semplice più breve e quello semplice più lungo la radice ed una foglia esiste una differenza, al massimo, costante;

e pieno: ogni nodo ha zero o due figli.

Gli alberi binari completi, e quasi completi, con n elementi hanno altezza $\Theta(\log(n))$;

I nodi di un albero binario possono essere pensati come oggetti che possiedono, almeno, una chiave ($x.key$), e tre puntatori: il padre ($x.p$), il figlio destro ($x.right$), ed il figlio sinistro ($x.left$).

correttezza: tutti gli elementi sono visitati esattamente una volta.

```
proc TreeInOrderTreeWalk (x) Θ(n)
  if (x ≠ nil )
    then
      {TreeInOrderTreeWalk(x.left)
       Print(x.key)
       TreeInOrderTreeWalk(x.right)}
```

```
proc TreePreOrderTreeWalk (x)
  if (x ≠ nil )
    then
      {Print(x.key)
       TreeInOrderTreeWalk(x.left)
       TreeInOrderTreeWalk(x.right)}
```

```
proc TreePostOrderTreeWalk (x)
  if (x ≠ nil )
    then
      {TreeInOrderTreeWalk(x.left)
       TreeInOrderTreeWalk(x.right)
       Print(x.key)}
```

Alberi binari di ricerca:

Tra tutti i possibili alberi, un caso particolare sono gli alberi binari di ricerca (BST), che quindi sono una struttura dinamica, basata sull'ordinamento, e implementata in maniera sparsa.

Associamo agli alberi binari di ricerca le operazioni di inserimento, cancellazione, ricerca, minimo, massimo, successore, e predecessore. Possiamo farlo perché la struttura è basata sull'ordinamento delle chiavi.

Le regole che un albero binario di ricerca deve rispettare (anche note come proprietà BST), sono:

- 1 Per ogni nodo x , se un nodo y si trova nel sotto-albero sinistro, allora $y.key \leq x.key$;
- 2 Per ogni nodo x , se un nodo y si trova nel sotto-albero destro, allora $y.key > x.key$.

Dunque si può dire che un BST è parzialmente ordinato.

```
proc BSTTreeSearch(x, k)
  if ((x = nil) or (x.key = k))
    then return x ← puntatore alla radice
  if (k ≤ x.key)
    then return BSTTreeSearch(x.left, k)
  else return BSTTreeSearch(x.right, k)
```

puntatore alla radice
chiave da cercare

```
proc BSTTreeMinimum(x)
  if ((x.left = nil) & A)
    then return x
  return BSTTreeMinimum(x.left)
```

compleSSità

La complessità di BSTTreeSearch è direttamente proporzionale alla sua altezza. Nel caso peggiore, l'albero assume l'aspetto di una lista e la ricerca di una chiave che non esiste prende tempo $\Theta(n)$. Nel caso medio, l'albero ha altezza logaritmica, e la ricerca di una chiave, anche non esistente, ha tempo $\Theta(\log(n))$.

Il nodo che contiene la chiave minima di un BST si trova sempre sull'ultimo nodo del ramo che dalla radice percorre sempre rami sinistri, mentre quello che contiene la chiave massima sull'ultimo nodo del ramo che dalla radice percorre sempre rami destri.

```
proc BSTTreeSuccessor(x)
  if (x.right ≠ nil)
    then return BSTTreeMinimum(x.right)
  y = x.p
  while ((y ≠ nil) and (x = y.right))
    { x = y
      y = y.p
    }
  return y
```

← individua la chiave con il valore subito successivo a $x.key$

Anche la complessità di BSTTreeSuccessor è proporzionale all'altezza dell'albero

```
proc BSTTreeInsert(T, z)
  y = nil
  x = T.root
  while (x ≠ nil)
    { y = x
      if (z.key ≤ x.key)
        then x = x.left
      else
        then x = x.right
    }
  z.p = y
  if (y = nil)
    then T.root = z
  if ((y ≠ nil) and (z.key ≤ y.key))
    then y.left = z
  if ((y ≠ nil) and (z.key > y.key))
    then y.right = z
```

caso la degenza
caso la inserzione

inserimento

Vogliamo mostrare che BSTTreeInsert è corretta, cioè se T è un BST e T' è il risultato di un inserimento, allora T' è un BST.

Se T è vuoto, il ciclo while non si esegue, e tra le istruzioni restanti si solo esegue la prima, mettendo z come radice di T' , che diventa un albero con un solo nodo e quindi corretto. Sia quindi T un BST corretto non vuoto.

L'invariante del ciclo è: la posizione corretta di z è nel sottoalbero radicato in x , e y ne mantiene il padre.

La complessità di BSTTreeInsert, come tutte le altre operazioni che abbiamo visto, è proporzionale all'altezza dell'albero,

```
proc BSTTreeDelete(T, z)
  if ((z.left = nil))
    then BSTTransplant(T, z, z.right)
  if ((z.left ≠ nil) and (z.right = nil))
    then Transplant(T, z, z.left)
  if ((z.left ≠ nil) and (z.right ≠ nil))
    then
      { y = BSTTreeMinimum(z.right)
        if (y.p ≠ z)
          then
            { BSTTransplant(T, z, y, right)
              y.right = z.right
              y.right.p = y
            }
        BSTTransplant(T, z, y)
        y.left = z.left
        y.left.p = y
      }
```

Se z non ha figli sinistri, o è una foglia, allora trapiantiamo il sotto-albero $z.right$ al posto di z (anche se $z.right$ è nil). Se z ha figlio sinistro, ma non destro, allora trapiantiamo il sotto-albero $z.left$ al posto di z ($z.left$ non può essere nil, altrimenti saremmo nel caso anteriore).

```
proc BSTTreeTransplant(T, u, v)
  if (u.p = nil)
    then T.root = v
  if ((u.p ≠ nil) and (u = u.p.left))
    then u.p.left = v
  if ((u.p ≠ nil) and (u = u.p.right))
    then u.p.right = v
  if (v ≠ nil)
    then v.p = u.p
```

Se invece z ha due figli, allora andiamo a prendere il suo successore immediato y , che si trova nel sotto-albero destro di z e non ha al più un figlio. Il nodo y va a prendere il posto di z e se y è figlio immediato di z allora il figlio destro di z diventa il figlio destro di y , e il resto rimane invariato, altrimenti (y è nel sotto-albero destro di z ma non è suo figlio immediato) prima rimpiazziamo y con il suo figlio destro, e poi rimpiazziamo z con y .

La complessità di BSTTreeDelete, che non contiene cicli, è, nuovamente, $\Theta(n)$ nel caso peggiore e $\Theta(\log(n))$ nel caso medio; questo si deve naturalmente alla presenza di una chiamata a BSTTreeMinimum.

	Liste	BST c. medio	BST c. peggiore
Inserimento	$\Theta(1)$	$\Theta(\log(n))$	$\Theta(n)$
Cancellazione	$\Theta(1)$	$\Theta(\log(n))$	$\Theta(n)$
Visita	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Ricerca	$\Theta(n)$	$\Theta(\log(n))$	$\Theta(n)$
Successore	$\Theta(n)$	$\Theta(\log(n))$	$\Theta(n)$
Predecessore	$\Theta(n)$	$\Theta(\log(n))$	$\Theta(n)$
Massimo	$\Theta(n)$	$\Theta(\log(n))$	$\Theta(n)$
Minimo	$\Theta(n)$	$\Theta(\log(n))$	$\Theta(n)$

Alberi red-black:

Un albero red-black (RBT) è un albero binario di ricerca (BST) bilanciato per costruzione. Possiede tutte le caratteristiche di un BST, ma la sua altezza è sempre $\Theta(\log(n))$, dove n è il numero di elementi dell'albero.

Un RBT, come un BST, è una struttura dati dinamica, basata sull'ordinamento, e sparsa.

È ovvio che tutte le operazioni, ed in particolare la ricerca di un elemento, che funzionano in tempo proporzionale all'altezza diventano esponenzialmente più efficienti su un RBT.

Ogni nodo in un RBT ha un'informazione in più rispetto a un nodo in un BST: oltre a un puntatore al padre, i puntatori ai due figli, e la chiave, abbiamo il colore ($x.\text{color}$), che per convenzione è rosso o nero. Inoltre ogni foglia (ogni nodo senza figli) possiede due figli virtuali, che non contengono chiave e sono sempre di colore nero.

Il padre della radice, per convenzione, è anche lui un nodo virtuale senza chiave, senza figli, e di colore nero.

Le regole che ogni albero rosso-nero deve rispettare, in aggiunta alla proprietà di ordinamento dei BST, sono:

- 1 Ogni nodo è rosso o nero;
- 2 La radice è nera;
- 3 Ogni foglia (esterna, nil) è nera;
- 4 Se un nodo è rosso, entrambi i suoi figli sono neri;
- 5 Per ogni nodo, tutti i percorsi semplici da lui alle sue foglie contengono lo stesso numero di nodi neri.

Chiameremo i nodi di un albero RB interni, per distinguerli dai nodi esterni che aggiungiamo in maniera artificiale a ogni albero RB. Una foglia esterna è un nodo che ha tutte le proprietà di ogni altro nodo ma non porta alcuna chiave, ed è sempre di colore nero. Quindi ogni nodo interno di un RBT ha sempre due figli (che possono essere entrambi esterni o uno solo dei due), ed ogni nodo esterno non ha figli. Dal punto di vista

implementativo, definiamo una sentinella $T.\text{Nil}$ come un nodo con tutte le proprietà di un nodo di T , e colore fissato a nero, per il ruolo di foglia esterna.

Cominciamo definendo l'altezza nera ($\text{bh}(x)$) di un nodo x in T come il numero di nodi neri su qualsiasi percorso semplice da x (senza contare x) a una foglia esterna (contandola). L'altezza nera di T è $\text{bh}(T.\text{root})$.

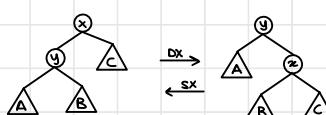
Se T è un RBT con n nodi interni (quindi escludendo le foglie esterne), allora la sua altezza massima è $2 \cdot \log(n+1)$. $\log(n) - 1 \leq h \leq 2 \cdot \log(n+1)$, cioè $h = \Theta(\log(n))$.

Inserimento ed eliminazioni in un RBT possono violare le proprietà, e che la maggiore difficoltà nell'implementare queste procedure consiste precisamente nel modicare la struttura dell'albero per ripristinare queste proprietà.

Un passo intermedio fondamentale per questa riparazione è la rotazione, che può essere destra o sinistra, e che preserva le proprietà BST (non le proprietà RBT). L'idea è che possiamo ribilanciare l'albero e poi preoccuparci dei colori. Risolviamo il problema della rotazione sinistra: dato un RBT T , ed un nodo x in T , con figlio destro y , ottenere un nuovo albero T' , dove y ha come figlio sinistro x .

Simmetricamente, potremo denire il problema della rotazione destra.

In entrambi i casi la complessità è $\Theta(1)$.



```

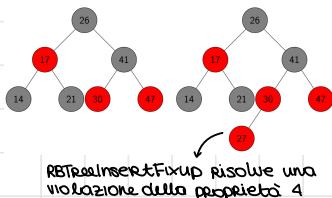
proc BSTTreeLeftRotate (T, x)
  y = x.right
  x.right = y.left
  if (y.left ≠ T.Nil)
    then y.left.p = x
  y.p = x.p
  if (x.p = T.Nil)
    then T.root = y
  if ((x.p ≠ T.Nil) and (x = x.p.left))
    then x.p.left = y
  if ((x.p ≠ T.Nil) and (x = x.p.right))
    then x.p.right = y
  y.left = x
  x.p = y
  
```

Usiamo BSTTreeInsert così com'è in quanto abbiamo la garanzia che la proprietà BST sia rispettata. Se il nodo inserito è colorato rosso, allora anche la proprietà 5 è rispettata; inoltre, poiché z sarà sempre una nuova foglia, inserendo correttamente le sue foglie esterne, garantiamo anche la proprietà 3. La proprietà 1 è rispettata semplicemente assegnando il colore (rosso) a z . Quindi, solo due proprietà possono essere violate: se z diventa la radice allora violiamo la 2, se, invece, z diventa figlio di un nodo rosso, allora violiamo la 4.

```
proc RBTreeInsert (T, z)
{y = T.Nil
x = T.root
while (x ≠ T.Nil)
{y = x
if (z.key < x.key)
then x = x.left
else x = x.right
z.p = y
if (y = T.Nil)
then T.root = z
if ((y ≠ T.Nil) and (z.key < y.key))
then y.left = z
if ((y ≠ T.Nil) and (z.key ≥ y.key))
then y.right = z
z.left = T.Nil
z.right = T.Nil
z.color = RED
RBTreeInsertFixup(T, z)}
```

CP: $\Theta(\log(n))$

```
proc RBTreeInsertFixup (T, z)
{while (z.p.color = RED)
{if (z.p = z.p.left)
then RBTreeInsertFixUpLeft(T, z)
else RBTreeInsertFixUpRight(T, z)
T.root.color = BLACK}
```



```
proc RBTreeInsertFixupLeft (T, z)
{y = z.p.p.right
if (y.color = RED)
then
{z.p.color = BLACK
y.color = BLACK
z.p.p.color = RED
z = z.p.p}
else
{if (z = z.p.right)
then
{z = z.p
TreeLeftRotate(T, z)
z.p.color = BLACK
z.p.p.color = RED
TreeRightRotate(T, z.p.p)}}
3
```

caso 1

caso 2

3

```
proc RBTreeInsertFixupRight (T, z)
{y = z.p.p.left
if (y.color = RED)
then
{z.p.color = BLACK
y.color = BLACK
z.p.p.color = RED
z = z.p.p}
else
{if (z = z.p.left)
then
{z = z.p
TreeRightRotate(T, z)
z.p.color = BLACK
z.p.p.color = RED
TreeLeftRotate(T, z.p.p)}}}
```

La scelta che si fa all'inizio di RBTreeInsertFixup genera 2 casi, che dipendono dal fatto che $z.p$ sia figlio destro o sinistro di $z.p.p$. All'interno di ogni caso vi sono tre sotto-casi, che si distinguono dal colore di y (lo zio di z): se è rosso, è un caso, se è nero e z è figlio destro è un secondo caso, mentre se è nero e z è figlio sinistro è un terzo caso. Il totale è quindi di 6 casi, i primi tre completamente simmetrici ai secondi tre.

	Liste	BST c. medio	BST c. peggiore	RBT c. peggiore
Inserimento	$\Theta(1)$	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(\log(n))$
Cancellazione	$\Theta(1)$	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(\log(n))$
Visita	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Ricerca	$\Theta(n)$	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(\log(n))$
Successore	$\Theta(n)$	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(\log(n))$
Predecessore	$\Theta(n)$	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(\log(n))$
Massimo	$\Theta(n)$	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(\log(n))$
Minimo	$\Theta(n)$	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(\log(n))$

Albero BT:

Un albero BT generalizza un albero RBT con fini differenti, ma mantenendo la sua proprietà fondamentale di bilanciamento. Anzi, un albero BT è sempre completo.

Gli alberi B sono una struttura dati ottimizzata per minimizzare gli accessi a disco.

Sono una struttura dinamica, basata sull'ordinamento, e memorizzata in maniera sparsa.

Un albero B si caratterizza per: possedere una arietà (che in questo contesto si conosce come branching factor) superiore a 2, spesso dell'ordine delle migliaia, un'altezza proporzionale a un logaritmo a base molto alta di n, dove n è il numero di chiavi, per avere nodi che contengono molte chiavi, tra loro ordinate, e per crescere verso l'alto, non verso il basso: un nodo comincia con essere la radice, e poi si converte in nodo interno generando una nuova radice. La complessità delle operazioni è proporzionale all'altezza: quindi i BT possono essere usati come gli RBT.

Un nodo x in un BT si caratterizza per avere comunque il puntatore al nodo padre (x.p), ma gli altri dati sono diversi dai nodi degli alberi binari visti fino ad adesso. Infatti, abbiamo:

il numero delle chiavi memorizzate nel nodo (x.n),

l'informazione sull'essere, o meno, una foglia (x.leaf, che è a uno se e solo se x è foglia),

i puntatori agli $n + 1$ figli di x ($x.c_1, \dots, x.c_{n+1}$), che sono indefiniti se x è foglia, e n chiavi ($x.key_1, \dots, x.key_n$), invece di una.

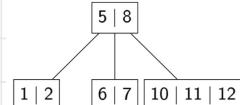
Il sotto-albero puntato da $x.c_i$ è legato alle chiavi $x.key_{i-1}$ (se c'è) e $x.key_i$ (se c'è); il legame viene formalizzato nelle proprietà degli alberi B.

In un nodo x il numero di chiavi, e quindi il branching factor, è vincolato da un parametro che si chiama grado minimo, si denota con t, ed è sempre maggiore o uguale a 2. → un nodo interno ha fino a $2t$ figli ← nodo pieno

Le proprietà di un albero B, che sostituiscono quelle di un albero red-black e generalizzano quelle di un albero binario di ricerca sono:

- 1 Ogni nodo, tranne la radice, ha almeno $t - 1$ chiavi;
- 2 Ogni nodo può contenere al massimo $2 \cdot t - 1$ chiavi;
- 3 Per ogni nodo x, $x.key_1 \leq x.key_2 \leq \dots \leq x.key_{n-1}$;
- 4 Per ogni nodo x, se un nodo y è contenuto nel sotto-albero radicato in $x.c_i$, allora tutte le sue chiavi sono minori o uguali a $x.key_i$ (se c'è);
- 5 Per ogni nodo x, se un nodo y è contenuto nel sotto-albero radicato in $x.c_i$, allora tutte le sue chiavi sono maggiori di $x.key_{i-1}$ (se c'è).

L'altezza massima di un BT T con n chiavi e grado minimo $t \geq 2$ è: $R \leq \log_t \left(\frac{n+1}{2} \right)$
e tutte le foglie sono alla stessa altezza. → Sempre completo

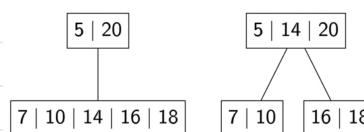


Inserimento:

In sintesi, cerchiamo di riempire un nodo fino a quando diventa pieno; quando il nodo è pieno, dividiamo il nodo in questione (che ha $2 \cdot t - 1$ chiavi) in due nodi di $t - 1$ chiavi ciascuno, ed inseriamo la nuova chiave nel nodo padre: se il padre diventa pieno in seguito a tale inserimento, ripetiamo l'operazione un livello più in alto. Quindi T cresce solamente quando la divisione ha luogo sulla radice: in questo caso si crea un nuovo nodo radice e si opera la divisione.

```
proc BTreeSplitChild(x, i)
{z = Allocate() nodo interno non
y = x.c_i pieno già in memoria
z.leaf = y.leaf principale
z.n = y.n
for j = 1 to t - 1 z.key_j = y.key_{j+1}
if (y.leaf = False)
    then for j = 1 to t z.c_j = y.c_{j+1}
y.n = t - 1
for j = x.n + 1 downto i + 1 x.c_{j+1} = x.c_j
x.c_{i+1} = z
for j = x.n downto i x.key_{j+1} = x.key_j
x.key_i = y.key_t
x.n = x.n + 1
DiskWrite(y)
DiskWrite(z)
DiskWrite(x)}
```

Il numero di operazioni CPU di BTreeSplitChild è $\Theta(t)$, mentre il costo in termini di operazioni su disco è $\Theta(1)$.



La procedura BTreelInsert utilizza una seconda procedura (BTreelInsertNonFull) che si occupa, ricorsivamente, di inserire una chiave assumendo che il nodo considerato (ma non ancora scelto per l'inserimento) non sia pieno. Poiché la chiave nuova va sempre inserita in una foglia, se quello considerato è già una foglia, allora la chiave si inserisce semplicemente. Se invece non è una foglia, allora scendendo ricorsivamente da un nodo non pieno, potremmo trovarci su un nodo pieno; in questo caso, eseguiamo la divisione e procediamo ricorsivamente.

```
proc BTreelInsert (T, k)
  r = T.root
  if (r.n = 2 · t - 1)
    then
      s = Allocate()
      T.root = s
      s.leaf = False
      s.n = 0
      s.c1 = r
      BTreelSplitChild(s, 1)
      BTreelInsertNonFull(s, k)
    else
      BTreelInsertNonFull(r, k)
```

```
proc BTreelInsertNonFull (x, k)
  i = x.n
  if (x.leaf = True)
    then
      while ((i ≥ 1) and (k < x.keyi))
        { x.keyi+1 = x.keyi; i = i - 1 }
        x.keyi+1 = k
        x.n = x.n + 1
        DiskWrite(x)
      else
        while ((i ≥ 1) and (k < x.keyi)) i = i - 1
        i = i + 1
        DiskRead(x.ci)
        if (x.ci.n = 2 · t - 1)
          then
            BTreelSplitChild(x, i)
            if (k > x.keyi)
              then i = i + 1
            BTreelInsertNonFull(x.ci, k)
```

Grafi: Un grafo è una struttura statica, non basata sull'ordinamento, e rappresentata in maniera sparsa o compatta a seconda del caso.

Un grafo è una tripla $G = (V, E, W)$ composta da un insieme V di vertici, un insieme $E \subseteq V \times V$ di archi, e una funzione $W : E \rightarrow R$ che assegna un peso ad ogni arco.

Il grafo G è indiretto se vale sia $(u, v) \in E \leftrightarrow (v, u) \in E$ ← archi senza direzione che $W(u, v) = W(v, u)$, e diretto altrimenti.

Quando da un vertice u possiamo raggiungere un vertice v usiamo il simbolo $u \sim v$. Il grado entrante di un nodo di un grafo è il numero di archi che lo raggiungono, ed il grado uscente il numero di archi che lo lasciano. Il grafo G è pesato se W non è costante, e non pesato altrimenti $G = (V, E)$

Chiamiamo sparso un grafo tale che $|E| \ll |V|^2$, e lo definiamo denso altrimenti.

Chiaramente $|E|$ può, al massimo, arrivare a $|V|^2$.

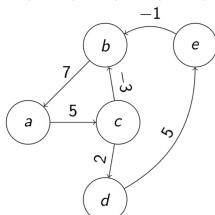
Ci sono due modi standard di rappresentare un grafo: con liste di adiacenza o con una matrice di adiacenza. In entrambi i casi, è conveniente pensare che ogni vertice $v \in V$ sia identificabile con un numero naturale da 1 a $|V|$:

Rappresentazione a liste di adiacenza

Usiamo un array $\text{Adj}[1, \dots, |V|]$ dove ogni elemento punta a una lista. Per ogni vertice v , la lista puntata da $\text{Adj}[v]$ contiene la chiave u se e solo se $(v, u) \in E$, e nel caso più generale, il nodo della lista contiene anche il peso dell'arco. Si preferisce questa rappresentazione per grafi sparsi.

Rappresentazione a matrice di adiacenza

Nella sua versione più generale, usiamo una matrice W che contiene sia l'informazione sul peso di ogni arco sia quella sulla sua esistenza. Chiaramente è una matrice quadrata di lato $|V|$, e si preferisce questo metodo per grafi densi. Invece di $W[i, j]$, si usa la scrittura W_{ij} .



a		b	c	d	e
b	7				
c		-3	2		
d				5	
e		-1			

Tipicamente, un algoritmo lineare nel caso peggiore avrà complessità $\Theta(|V| + |E|)$ o $O(|V| + |E|)$.

Vista in ampiezza

Dato un grafo $G = (V, E)$ diretto o indiretto, non pesato, ed un vertice particolare chiamato sorgente $s \in V$, vogliamo sapere quanti archi sono necessari a raggiungere qualunque altro vertice (raggiungibile) da s . A questo fine, utilizzeremo una visita di G chiamata in ampiezza. BreadthFirstSearch esplora sistematicamente gli archi di G , e cerca di scoprire nuovi vertici raggiungibili da quelli già conosciuti. Computa la distanza minima in termini di numero di archi che esiste tra s ed ogni vertice scoperto e produce un albero di visita in ampiezza che contiene tutti i vertici raggiungibili. → non ha molto senso usare questa vista per i grafici pesati

Utilizzeremo i colori bianco, grigio e nero per colorare i vertici mentre sono scoperti. Tutti sono bianchi all'inizio: quando un nuovo vertice è scoperto, diventa grigio, e quando anche tutti i vertici ad esso adiacente sono stati scoperti, diventa nero (ed il suo ruolo termina). Solo s è colorato di grigio all'inizio. Durante la scoperta, s diventa la radice dell'albero di visita in ampiezza. Alla scoperta di un nuovo vertice v (da bianco diventa grigio) a partire da un vertice già scoperto u , l'arco (u, v) diventa parte dell'albero (virtualmente), e v viene marcato come predecessore di v nell'albero stesso. Un vertice bianco viene scoperto (e colorato di grigio) al massimo una volta: quindi l'albero risultante è ben definito.

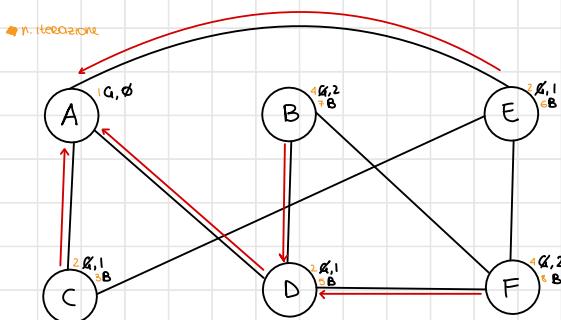
```

proc BreadthFirstSearch (G, s)
  for (u ∈ G.V \ {s})
    { u.color = WHITE
      { u.d = ∞ ← d. unknown
        u.π = nil ← π. predecessor unknown
        s.color = GRAY
        s.d = 0
        s.π = nil
        Q = ∅ ← initializzazione coda
        Enqueue(Q, s)
      }
    while (Q ≠ ∅)
      { u = Dequeue(Q)
        for (v ∈ G.Adj[u])
          { if (v.color = WHITE)
            then
              { v.color = GRAY
                v.d = u.d + 1
                v.π = u
                Enqueue(Q, v)
              }
            u.color = BLACK
          }
        }
      }
    
```

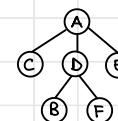
Definiamo la distanza più corta di v dalla sorgente s (denotata con $\delta(s, v)$) come il numero minimo di archi che sono necessari per raggiungere v da s . Le sue proprietà sono il fatto che essa sia zero tra un vertice e se stesso (riessività, $\delta(s, s) = 0$), che sia infinito da s a v quando il secondo è irraggiungibile dal primo ($\delta(s, v) = \infty$), che sia una distanza (disugualanza triangolare, cioè per ogni coppia di vertici v, u tali che esiste un arco (u, v) , succede che $\delta(s, v) \leq \delta(s, u) + 1$).

Invariante: per ogni $v \in V$ inserito in Q , si ha che $v.d \geq \delta(s, v)$.

Con questa analisi, se il grafo è connesso, allora la complessità è $\Theta(|V| + |E|)$ in tutti i casi. Se, invece, il grafo è sconnesso, allora è $O(|V| + |E|)$, perché una parte degli archi potrebbe non essere mai vista. La funzione $|V| + |E|$ diventa $|V|^2$ quando il grafo è denso. Quindi, nel caso peggiore (grafo connesso e denso) è $\Theta(|V|^2)$, ma normalmente si accetta $\Theta(|V| + |E|)$.



Sorgente: A



Vista in ampiezza: avere componenti irraggiungibili è comune

Come nel caso della visita in ampiezza, è indifferente se G è o no pesato, o se G è o no diretto. Ma se $G = (V, E)$ è, in effetti, diretto, allora visitarlo in maniera diversa da quella vista precedentemente può fornire informazioni utili.

Ci proponiamo, in particolare, di risolvere i seguenti tre problemi (definiti solo su grafi diretti):

- 1) stabilire se G è ciclico: stabilire se contiene almeno un ciclo;
- 2) costruire un ordinamento topologico di G : elencare tutti i suoi vertici in un ordine qualsiasi tale che ogni vertice v è elencato solo se tutti i vertici dai quali v si può raggiungere sono stati elencati prima;
- 3) conoscere ed enumerare tutte le componenti fortemente connesse di G : elencare tutti i sottoinsiemi massimali di V tali che, ogni vertice in ogni sottoinsieme raggiunge ogni altro vertice in quel sottoinsieme.

Le soluzioni a questi problemi hanno in comune la stessa procedura: la visita in profondità.

Il proposito della visita in profondità è quello di scoprire tutti i vertici raggiungibili da ogni potenziale sorgente s . La differenza tra le due visite è che i vertici nella visita in profondità vengono scoperti il prima possibile a partire da quelli già scoperti: la visita in profondità è inerentemente ricorsiva.

Anche DepthFirstSearch assume che G sia rappresentato con liste di adiacenza, e riempie un campo $v.\pi$ per generare un albero di visita in profondità come risultato della visita; la differenza qui è che invece di un solo albero, si produce una foresta di alberi, uno per ogni sorgente. ← per quando i componenti sono irraggiungibili tra loro

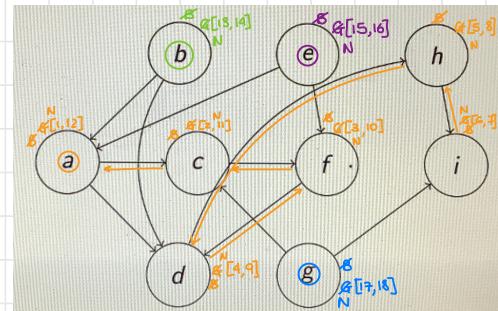
DepthFirstSearch riempie anche dei campi $v.d$ e $v.f$: il primo rappresenta il momento della scoperta, e il secondo il momento nel quale il vertice viene abbandonato (tutto il suo sotto-grafo è stato scoperto). I campi $v.d$ e $v.f$ sono interi tra 1 e $2 \cdot |V|$. Vengono chiamati genericamente tempi (di scoperta e di abbandono). Ogni nuovo evento (una scoperta o un abbandono) provoca che il tempo si incrementa.

Chiaramente, per ogni vertice u , abbiamo $u.d < u.f$.

Ogni volta che DepthVisit viene chiamata, si inizia un nuovo albero della foresta degli alberi di visita in profondità. Se G è tale che tutti i vertici sono raggiungibili dal primo vertice visitato, allora ci sarà un solo albero; altrimenti ce ne saranno di più.

```
proc DepthFirstSearch (G)
  for (u ∈ G.V)
    { u.color = WHITE
      u.π = Nil
      time = 0
      for (u ∈ G.V)
        {if (u.color = WHITE)
          then DepthVisit(G, u)
```

```
proc DepthVisit (G, u)
  time = time + 1
  u.d = time
  u.color = GREY
  for (v ∈ G.Adj[u])
    {if (v.color = WHITE)
      then
        {v.π = u
          DepthVisit(G, v)
          u.color = BLACK
          time = time + 1
          u.f = time}
```



BreadthFirstSearch esegue un ciclo che costa $|V|$ per la colorazione iniziale, e poi, per ogni vertice chiama DepthVisit. Il massimo numero di chiamate a DepthVisit, anche contando le chiamate ricorsive, è $|E|$.

La complessità è $\Theta(|V| + |E|)$, che, nel caso peggiore di grafi densi, diventa $\Theta(|V|^2)$ (ma, come nel caso della visita in ampiezza, accettiamo $\Theta(|V| + |E|)$).

In un grafo diretto, chiamiamo **ciclo** un percorso v_1, v_2, \dots, v_k di vertici tali che per ogni i esiste l'arco (v_i, v_{i+1}) , e che $v_1 = v_k$. Quando un grafo diretto è privo di cicli, lo chiamiamo **DAG** (Directed Acyclic Graph). Come abbiamo visto precedentemente, uno degli utilizzi di DepthFirstSearch è quello di stabilire se un dato grafo G diretto è o no acilico.

Risolviamo dunque il seguente problema: dato un grafo diretto G stabilire se presenta, o no, un ciclo. L'algoritmo **CycleDet** che usiamo prevede di eseguire DepthFirstSearch modicata in maniera da interrompersi se si visita un nodo grigio: al momento di visitare un nodo grigio, siamo certi di aver trovato un ciclo.

```
proc CycleDet (G)
  cycle = False
  for (u ∈ G.V) u.color = WHITE
  for (u ∈ G.V)
    if (u.color = WHITE)
      then DepthVisitCycle(G, u)
  return cycle
```

```
proc DepthVisitCycle (G, u)
  u.color = GREY
  for (v ∈ G.Adj[u])
    if (v.color = WHITE)
      then DepthVisitCycle(G, v)
    if (v.color = GREY)
      then cycle = True
  u.color = BLACK
```

Chiaramente la complessità del nostro algoritmo è la stessa della visita in profondità, e la terminazione è ovvia.

Il problema dell'**ordinamento topologico** prende in input un grafo connesso G diretto senza cicli e restituisce una lista collegata $v_1, \dots, v_{|V|}$ di vertici topologicamente ordinati; per ogni coppia v_i, v_j di vertici, v_i appare prima nella lista di v_j se e solo se v_i precede topologicamente v_j .

Nell'algoritmo **TopologicalSort**, prima, chiamiamo DepthFirstSearch su G per computare $v.f$ per ogni $v ∈ G.V$, e, poi, per ogni nodo nito v lo inseriamo in testa a una lista collegata; finalmente, restituiamo la lista.

```
proc TopologicalSort (G)
  for (u ∈ G.V) u.color = WHITE
  L = ∅
  time = 0
  for (u ∈ G.V)
    if (u.color = WHITE)
      then DepthVisitTS(G, u)
  return L
```

```
proc DepthVisitTS (G, u)
  time = time + 1
  u.d = time
  u.color = GREY
  for (v ∈ G.Adj[u])
    if (v.color = WHITE)
      then DepthVisitTS(G, v)
    u.color = BLACK
  time = time + 1
  u.f = time
  ListInsert(L, u)
```

Chiaramente la complessità del nostro algoritmo è la stessa della visita in profondità, e la terminazione è ovvia.

Trovare le componenti fortemente connesse di un grafo diretto G è la terza applicazione classica di DepthFirstSearch che vediamo. Dato un grafo diretto G , una componente fortemente connessa (SCC) è un sottoinsieme massimale $V' ⊆ V$ tale che, per ogni $u, v ∈ V'$, succede che $u \rightarrow v$ e che $v \rightarrow u$.

Un elemento fondamentale nello studio delle SCC è il **grafo trasposto** di G .

Dato G diretto, il grafo trasposto G^T di G è ottenuto invertendo la direzione di ogni arco. Questo si può ottenere facilmente, ed il problema ha complessità $\Theta(|V| + |E|)$ quando G è rappresentato con liste di adiacenza.

La proprietà più interessante di G è che G e G^T hanno le stesse SCC,

```
proc StronglyConnectedComponents (G)
  for (u ∈ G.V)
    { u.color = WHITE
    { u.π = Nil
    time = 0
    for (u ∈ G.V)
      if (u.color = WHITE)
        then DepthVisit(G, u)
    for (u ∈ G.V)
      if (u.color = WHITE)
        then DepthVisit(G^T, u)
        u.π = Nil
      time = 0
      L = ∅
      for (u ∈ G^T.V in rev. finish time order)
        if (u.color = WHITE)
          then DepthVisit(G^T, u)
          ListInsert(L, u)
    return L
```

Grafi e alberi di copertura minima

Ci concentriamo adesso sui grafi pesati indiretti connessi. Sia quindi $G = (V, E, W)$.

Possiamo domandarci qual è il costo di visitare ogni vertice, ed, in particolare, se c'è una scelta di archi ottima, che minimizza il costo.

La strategia generale per risolvere questo problema è di tipo greedy. Questo significa che ad ogni passo faremo la scelta localmente migliore, e vogliamo in questo modo ottenere il risultato globalmente migliore.

Questo che vediamo è un esempio, tra molti possibili, di applicazione di questa strategia di programmazione, che va vista, epistemologicamente, sullo stesso piano della strategia che abbiamo chiamato divide and conquer

Cominciamo con osservare che un MST A è un insieme di archi tali che, per denizione, formano un albero (indiretto) che tocca tutti i nodi del grafo. Quindi, eliminando qualunque arco da un albero di copertura (anche uno non minimo) si ottengono due alberi sconnessi (nel caso limite, uno dei due è composto da un solo nodo).

In generale, qualunque partizione di V in due sottoinsiemi S e $V \setminus S$ viene chiamata taglio del grafo (semplicemente, taglio). Costruire un MST T significa considerare un taglio ed aggiungere progressivamente un arco, partendo dal taglio più semplice che comprende un solo nodo in S

Considerata qualunque situazione di costruzione di un MST, cioè qualunque taglio, il prossimo passo per la costruzione è scegliere sempre l'arco di peso minimo che lo attraversa.

Chiamiamo questo arco sicuro (per il taglio).

Se tutti i pesi di archi di G sono diversi tra loro, allora l'MST è unico: infatti, per ogni taglio ci sarebbe sempre una sola scelta. In maniera simile, se così non fosse, cioè se ci fossero archi di peso uguale, allora l'MST potrebbe non essere unico. L'algoritmo di Prim è un modo eciente di codicare quanto visto.

L'idea alla base dell'algoritmo di Prim è quella di partire da un vertice qualsiasi, e, ad ogni passo, aggiungere un arco (ed un vertice) in modo che l'arco aggiunto sia un arco sicuro (come precedentemente denito). La corretta struttura dati in questo caso deve permettere di mantenere un insieme di vertici in maniera da poter facilmente individuare ed estrarre, tra questi, quello che comporta una spesa minima in termini del peso dell'arco che viene scelto per aggiungere quel vertice. È chiaro che abbiamo bisogno di una coda di priorità.

Ogni vertice di G viene arricchito con due campi: $v.\text{key}$ (il peso minimo, inizialmente ∞ , tra gli archi che connettono qualche vertice di T con v), e $v.\pi$ (il padre di v , inizialmente Nil , nel MST risultante). Inizialmente tutti i vertici si trovano nella coda di priorità Q semi-ordinata su $v.\text{key}$, dove, sempre inizialmente, tutti gli elementi sono a ∞ . La radice r è data esplicitamente, e si cerca un MST radicato in r . Ad ogni scelta, i pesi degli elementi in Q (la coda di priorità) vengono modicati in base al principio che abbiamo spiegato, il vertice viene estratto da Q e inserito in T (T viene mantenuto in maniera virtuale: costruiamo l'albero modicando i predecessori $v.\pi$) e ripetiamo finché Q si svuota, e tutti i vertici sono stati coperti.

```
proc MST-Prim ( $G, w, r$ )
  for ( $v \in G.V$ )
    do
       $\{ v.\text{key} = \infty$ 
       $\{ v.\pi = \text{Nil}$ 
       $r.\text{key} = 0$ 
       $Q = G.V$ 
    while ( $Q \neq \emptyset$ )
      do
         $\{ u = \text{ExtractMin}(Q)$ 
        for ( $v \in G.\text{Adj}[u]$ )
          do
            if (( $v \in Q$ ) and ( $W(u, v) < v.\text{key}$ ))
              do
                 $\{ v.\pi = u$ 
                 $v.\text{key} = W(u, v)$ 
```

La costruzione della coda costa $\Theta(|V|)$, l'estrazione del minimo $\Theta(|V|)$, e il decremento $\Theta(1)$. Immaginiamo che il grafo sia denso. Allora avremo:

$\Theta(|V|)$ (inizializzazione), $\Theta(|V|)$ (costruzione della coda), $\Theta(|V|^2)$ (per le estrazioni del minimo, analisi aggregata), e $\Theta(|E|) = \Theta(|V|^2)$ (per i decrementi, analisi aggregata), per un totale di $\Theta(|V|^2)$. Osserviamo che, con questa implementazione, anche se il grafo fosse sparso, non cambierebbe nulla dal punto di vista asintotico.

In alternativa possiamo usare una heap binaria per implementare la coda, il che ci permetterà di avere un vantaggio, in caso di grafi sparsi. Avremo che la costruzione della coda costa sempre $\Theta(|V|)$, l'estrazione del minimo costa $\Theta(\log(|V|))$, e il decremento costa $\Theta(\log(|V|))$. Quindi, con un grafo sparso, avremo: $\Theta(|V|)$ (inizializzazione), $\Theta(|V|)$ (costruzione della coda), $\Theta(|V| \cdot \log(|V|))$ (per le estrazioni del minimo, analisi aggregata), e $\Theta(|E| \cdot \log(|V|))$ (per i decrementi, analisi aggregata), per un totale di $\Theta(|E| \cdot \log(|V|))$ (anche nei grafi sparsi ci sono più archi che vertici - altrimenti avremmo vertici sconnessi non coinvolti nel problema).

Il caso peggiore dunque si verifica con un grafo denso, e, per minimizzare il danno, si preferisce l'implementazione con una coda senza struttura.

Una valida alternativa a MST-Prim è l'algoritmo noto come algoritmo di Kruskal, che utilizza una generalizzazione del concetto di taglio e del concetto di arco sicuro per il taglio al fine di ottenere un albero di copertura minimo.

L'idea di MST-Kruskal è che possiamo ordinare gli archi in ordine crescente di peso, e, analizzandoli uno ad uno in questo ordine, stabilire se inserirlo come parte dell'albero di copertura minimo oppure no. Quale sarebbe la ragione di non farlo, ad un certo punto dell'esecuzione? Semplicemente, un arco (u, v) (che è chiaramente l'arco di peso minimo non ancora considerato, visto che li stiamo analizzando in ordine) non è parte di nessun MST se u e v sono già connessi da qualche altro arco precedentemente scelto. Come prima, sia T l'insieme di archi che abbiamo scelto fino ad un certo punto. T , a differenza del caso di MST-Prim, non è necessariamente un albero in ogni momento, ma lo sarà certamente alla fine della computazione. Dati gli archi di T , e dato l'insieme V di vertici, diciamo che un sottoinsieme S di V è T -connesso se, considerando solo archi in T , è un albero, ed è massimale.

Un arco di peso minimo tra quelli non ancora considerati che attraversa un taglio generalizzato è un arco sicuro. Passare da un insieme T ad un insieme T' scegliendo un arco di peso minimo tra quelli non ancora considerati, assumendo che attraversi il taglio, garantisce che, se T era sottoinsieme di qualche MST, allora lo sarà anche T' . Un arco (u, v) attraversa un taglio generalizzato se u e v appartengono a diverse componenti T -connesse. Quindi abbiamo bisogno di una struttura dati per insiemi disgiunti.

L'operazione di MakeSet costruisce un nuovo insieme (che per noi sarà una componente T -connessa), quella di FindSet stabilisce se due elementi appartengono allo stesso insieme (alla stessa componente T -connessa), e la Union unisce due insiemi in uno solo (unisce due componenti T -connesse in una sola, come conseguenza di aver scelto un arco).

```
proc MST-Kruskal ( $G, w$ )
   $T = \emptyset$ 
  for  $(v \in G.V)$ 
    do MakeSet( $v$ )
  SortNoDecreasing( $G.E$ )
  for  $((u, v) \in G.E - \text{in order})$ 
    do
      if  $(\text{FindSet}(u) \neq \text{FindSet}(v))$ 
      then
         $\left\{ \begin{array}{l} T = T \cup \{(u, v)\} \\ \text{Union}(u, v) \end{array} \right.$ 
    return  $T$ 
```

Invariante: ogni arco che viene aggiunto nel ciclo è un arco sicuro.

Se il grafo è denso, allora abbiamo: inizializzazione ($O(1)$), ordinamento, $(\Theta(|E| \cdot \log(|E|)) = \Theta(|V|^2 \cdot \log(|E|)) = \Theta(|V|^2 \cdot \log(|V|)))$, più $O(|V| + |E|)$ diverse operazioni su insiemi, di cui $O(|V|)$ sono MakeSet (per un totale di $O((|V| + |E|) \cdot \alpha(|V|)) = O(|V| + |E|)$).

In totale, $\Theta(|V|^2 \cdot \log(|V|))$. Se invece il grafo è sparso abbiamo: inizializzazione ($O(1)$), ordinamento: $(\Theta(|E| \cdot \log(|E|))$), e $O(|V| + |E|)$ diverse operazioni su insiemi, di cui $O(|V|)$ sono MakeSet (per un totale di $O((|V| + |E|) \cdot \alpha(|V|)) = O(|V| + |E|)$). Totale, $(\Theta(|E| \cdot \log(|E|))$).

Grafi e percorsi minimi con sorgente singola

Ci concentriamo adesso su grafi diretti, pesati, e connessi. Dato $G = (V, E, W)$, e dato un vertice s , ci proponiamo di trovare per ogni $v \in V$ il percorso di peso minimo che porta da s a v . Questo problema, chiamato percorsi minimi con sorgente singola è da considerarsi la generalizzazione sia di BreadthFirstSearch che degli algoritmi per la ricerca dell'albero di copertura minimo. Nel primo caso da grafi non pesati a pesati e nel secondo da grafi indiretti a diretti.

Immaginiamo di individuare un vertice speciale s (la sorgente, come nella visita in ampiezza), e diciamo che $\delta(s, v)$ è la distanza più corta tra s e v , e la definiamo come segue:

Dato un certo percorso $p: v_0 \rightsquigarrow v_k$, il suo peso è $w(p) = \sum_{i=1}^k w(v_{i-1}, v_i)$ e

$$\delta(s, v) = \begin{cases} \min w(p) & p = s, \dots, v \\ \infty & \text{se } v \text{ non si raggiunge da } s \end{cases}$$

Un percorso di peso minimo da s a v , denotato con $s = v_0, v_1, \dots, v_k = v$ ha una sotto struttura ottima.

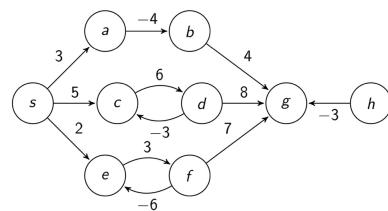
Se da s c'è un ciclo negativo raggiungibile, allora non ha senso denire un percorso di peso minimo da s : ogni nuovo passaggio attraverso quel ciclo diminuirebbe il peso. In questo caso diciamo che il peso del percorso da s a v , $\delta(s, v)$ è $-\infty$. In maniera simile, se v semplicemente non è raggiungibile da s , diremo che il peso del percorso minimo è ∞ .

Indipendentemente dalla presenza di pesi negativi, possiamo sempre dire che un percorso minimo è aciclico, e quindi ci limitiamo a studiare percorsi minimi di lunghezza $|V| - 1$ al massimo.

Cerchiamo l'albero dei cammini di peso minimo in G (questa volta si tratta di un albero radicato diretto), che non è necessariamente unico, e tale che ogni (presso di ogni) ramo rappresenta un cammino di peso minimo da s al vertice in questione.

In generale gli algoritmi per il calcolo di questo albero utilizzano una tecnica chiamata rilassamento, che funziona sulla base tanto di $v.\pi$ come di $v.d$, rappresentando, quest'ultimo, il peso del cammino minimo che si conosce no ad un determinato momento da s a v .

Il valore $v.d$ deve essere visto pertanto come una stima del peso del cammino minimo da s a v . Come tale, deve essere inizializzata.



In questo esempio, i percorsi minimi da s a b, c, f e h , costano, rispettivamente, $-1, 5, -\infty$ e ∞ .

```
proc InitializeSingleSource ( $G, s$ )
  for ( $v \in G.V$ )
    {  
       $v.d = \infty$   
       $v.\pi = Nil$   
       $s.d = 0$   
    }
```

```
proc Relax ( $u, v, W$ )
  if ( $v.d > u.d + W(u, v)$ )
    then
      {  
         $v.d = u.d + W(u, v)$   
         $v.\pi = u$   
      }
```

Il primo algoritmo che vediamo, il più generale, noto come algoritmo di Bellman-Ford, lavora con pesi negativi e con cicli negativi: restituisce i cammini minimi ed i loro pesi se non ci sono cicli negativi raggiungibili dalla sorgente, e false altrimenti.

```
proc Bellman-Ford ( $G, s$ )  $\Theta(|V| \cdot |E|)$ 
  InitializeSingleSource( $G, s$ )
  for  $i = 1$  to  $|G.V| - 1$ 
    {for  $((u, v) \in G.E)$  Relax( $u, v, W$ )
     for  $((u, v) \in G.E)$ 
       if ( $v.d > u.d + W(u, v)$ )
         then return false
     return true}
```

Nel caso particolare in cui possiamo assumere che G sia aciclico, trovare i percorsi minimi da una sorgente s , anche in presenza di pesi negativi, è più semplice. Infatti un grafo aciclico può essere ordinato topologicamente: osservando che se u precede v in un percorso minimo, allora u precede v nell'ordine topologico. Quindi possiamo operare un ordinamento topologico prima, e poi una serie di rilassamenti in ordine, risparmiando quindi molti passi di rilassamento. L'algoritmo, che noi chiamiamo DAGShortestPath, è in realtà Bellman-Ford ottimizzato per il caso aciclico.

```
proc DAGShortestPath ( $G, s$ )  $\Theta(|V| \cdot |E|)$ 
  TopologicalSort( $G$ )
  InitializeSingleSource( $G, s$ )
  for ( $u \in G.V - in\ order$ )
    {for ( $v \in G.Adj[u]$ ) Relax( $u, v, W$ )}
```

Possiamo migliorare le prestazioni di Bellman-Ford se aggiungiamo l'ipotesi che tutti gli archi hanno peso positivo o zero? L'algoritmo di Dijkstra risponde positivamente a questa domanda, utilizzando una tecnica che ricorda molto l'algoritmo di Prim. Come in MST-Prim, abbiamo bisogno di una coda di priorità. La chiave sarà il valore $v.d$ di ogni vertice.

```
proc Dijkstra ( $G, s$ )
  InitializeSingleSource( $G, s$ )
   $S = \emptyset$ 
   $Q = G.V$ 
  while ( $Q \neq \emptyset$ )
    { $u = ExtractMin(Q)$ 
      $S = S \cup \{u\}$ 
     for ( $v \in G.Adj[u]$ ) if ( $v \in Q$ )
       then Relax( $u, v, W$ )}
```

Complessità:

Se il grafo è denso $\Theta(|V|^2)$

Invece se è sparso $\Theta(|E| \cdot \log(|V|))$.

Algoritmi trattati a lezione

Regole: Si ammette questo documento all'esame; è consentito arricchire questo documento con appunti personali. Si ammette una copia di questo documento per studente; **non** si ammette in nessun caso la condivisione dello stesso tra diversi candidati. Inoltre, non si ammettono fotocopie di questo documento che riportino appunti personali di altre persone, e non si ammette alcun altro documento al di fuori di questo. Non si farà nessuna eccezione.

```
proc InsertionSort (A)
  for (j = 2 to A.length)
    key = A[j]
    i = j - 1
    while ((i > 0) and (A[i] > key))
      A[i + 1] = A[i]
      i = i - 1
    A[i + 1] = key
```

```
proc RecursiveBinarySearch (A, low, high, k)
  if (low > high)
    then return nil
  mid = (high + low)/2
  if (A[mid] = k)
    then return mid
  if (A[mid] < k)
    then return RecursiveBinarySearch(A, mid + 1, high, k)
  if (A[mid] > k)
    then return RecursiveBinarySearch(A, low, mid - 1, k)
```

```
proc SelectionSort (A)
  for (j = 1 to A.length - 1)
    min = j
    for (i = j + 1 to A.length)
      if (A[i] < A[min])
        then min = i
    SwapValue(A, min, j)
```

```
proc Merge (A, p, q, r)
  n1 = q - p + 1
  n2 = r - q
  let L[1, ..., n1] and R[1, ..., n2] be new array
  for (i = 1 to n1) L[i] = A[p + i - 1]
  for (j = 1 to n2) R[j] = A[q + j]
  i = 1
  j = 1
  for (k = p to r)
    if (i ≤ n1)
      then
        if (j ≤ n2)
          then
            if (L[i] ≤ R[j])
              then CopyFromL(i)
            else CopyFromR(j)
          else CopyFromL(i)
        else CopyFromR(j)
```

```
proc MergeSort (A, p, r)
  if (p < r)
    then
      q = [(p + r)/2]
      MergeSort(A, p, q)
      MergeSort(A, q + 1, r)
      Merge(A, p, q, r)
```

```
proc Partition (A, p, r)
  x = A[r]
  i = p - 1
  for (j = p to r - 1)
    if (A[j] ≤ x)
      then
        i = i + 1
        SwapValue(A, i, j)
  SwapValue(A, i + 1, r)
  return i + 1
```

```
proc QuickSort (A, p, r)
  if (p < r)
    then
      q = Partition(A, p, r)
      QuickSort(A, p, q - 1)
      QuickSort(A, q + 1, r)
```

```
proc RandomizedPartition (A, p, r)
  s = p ≤ Random() ≤ r
  SwapValue(A, s, r)
  x = A[r]
  i = p - 1
  for j = p to r - 1
    if (A[j] ≤ x)
      then
        i = i + 1
        SwapValue(A, i, j)
  SwapValue(A, i + 1, r)
  return i + 1
```

```
proc RandomizedQuickSort (A, p, r)
  if (p < r)
    then
      q = RandomizedPartition(A, p, r)
      RandQuickSort(A, p, q - 1)
      RandQuickSort(A, q + 1, r)
```

```
proc Empty (S)
  if (S.top = 0)
    then return true
  return false
```

```
proc Push (S, x)
  if (S.top = S.max)
    then return "overflow"
  S.top = S.top + 1
  S[S.top] = x
```

```
proc Pop (S)
  if (Empty(S))
    then return "underflow"
  S.top = S.top - 1
  return S[S.top + 1]
```

```
proc Enqueue (Q, x)
  if (Q.dim = Q.length)
    then return "overflow"
  Q[Q.tail] = x
  if (Q.tail = Q.length)
    then Q.tail = 1
  else Q.tail = Q.tail + 1
  Q.dim = Q.dim + 1
```

```
proc Dequeue (Q)
  if (Q.dim = 0)
    then return "underflow"
  x = Q[Q.head]
  if (Q.head = Q.length)
    then Q.head = 1
  else Q.head = Q.head + 1
  Q.dim = Q.dim - 1
  return x
```



```

proc Union (x, y)
   $\begin{cases} S_1 = x.\text{head} \\ S_2 = y.\text{head} \\ \text{if } (S_1 \neq S_2) \\ \quad \text{then} \\ \quad \left\{ \begin{array}{l} S_1.\text{tail}.next = S_2.\text{head} \\ z = S_2.\text{head} \\ \text{while } (z \neq \text{nil}) \\ \quad \left\{ \begin{array}{l} z.\text{head} = S_1 \\ z = z.\text{next} \end{array} \right. \\ S_1.\text{tail} = S_2.\text{tail} \end{array} \right. \end{cases}$ 

proc MakeSet (S, S, x, i)
   $\begin{cases} S[i].\text{set} = x \\ S.\text{head} = x \\ S.\text{tail} = x \end{cases}$ 

proc FindSet (x)
   $\{ \text{return } x.\text{head}.head \}$ 

proc Union (x, y)
   $\begin{cases} S_1 = x.\text{head} \\ S_2 = y.\text{head} \\ \text{if } (S_1 \neq S_2) \\ \quad \text{then} \\ \quad \left\{ \begin{array}{l} \text{if } (S_2.\text{rank} > S_1.\text{rank}) \\ \quad \text{then} \\ \quad \left\{ \begin{array}{l} S_{\text{temp}} = S_1 \\ S_1 = S_2 \\ S_2 = S_{\text{temp}} \\ S_1.\text{tail}.next = S_2.\text{head} \\ z = S_2.\text{head} \\ \text{while } (z \neq \text{nil}) \\ \quad \left\{ \begin{array}{l} z.\text{head} = S_1 \\ z = z.\text{next} \end{array} \right. \\ S_1.\text{tail} = S_2.\text{tail} \\ S_1.\text{rank} = S_1.\text{rank} + S_2.\text{rank} \end{array} \right. \end{array} \right. \end{cases}$ 

proc MakeSet (S, S, x, i)
   $\begin{cases} S[i].\text{set} = x \\ S.\text{head} = x \\ S.\text{tail} = x \\ S.\text{rank} = 0 \end{cases}$ 

proc Union (x, y)
   $\begin{cases} x = \text{Findset}(x) \\ y = \text{Findset}(y) \\ \text{if } (x.\text{rank} > y.\text{rank}) \\ \quad \text{then } y.p = x \\ \text{if } (x.\text{rank} \leq y.\text{rank}) \\ \quad \text{then} \\ \quad \left\{ \begin{array}{l} x.p = y \\ \text{if } (x.\text{rank} = y.\text{rank}) \\ \quad \text{then } y.\text{rank} = y.\text{rank} + 1 \end{array} \right. \end{cases}$ 

proc MakeSet (x)
   $\{ \begin{array}{l} x.p = x \\ x.\text{rank} = 0 \end{array} \}$ 

proc FindSet (x)
   $\{ \begin{array}{l} \text{if } (x \neq x.p) \\ \quad \text{then } x.p = \text{FindSet}(x.p) \end{array} \}$ 

proc TreeInOrderTreeWalk (x)
   $\{ \begin{array}{l} \text{if } (x \neq \text{nil}) \\ \quad \text{then} \\ \quad \left\{ \begin{array}{l} \text{TreeInOrderTreeWalk}(x.\text{left}) \\ \text{Print}(x.\text{key}) \\ \text{TreeInOrderTreeWalk}(x.\text{right}) \end{array} \right. \end{array} \}$ 

proc TreePreOrderTreeWalk (x)
   $\{ \begin{array}{l} \text{if } (x \neq \text{nil}) \\ \quad \text{then} \\ \quad \left\{ \begin{array}{l} \text{Print}(x.\text{key}) \\ \text{TreeInOrderTreeWalk}(x.\text{left}) \\ \text{TreeInOrderTreeWalk}(x.\text{right}) \end{array} \right. \end{array} \}$ 

proc TreePostOrderTreeWalk (x)
   $\{ \begin{array}{l} \text{if } (x \neq \text{nil}) \\ \quad \text{then} \\ \quad \left\{ \begin{array}{l} \text{TreeInOrderTreeWalk}(x.\text{left}) \\ \text{TreeInOrderTreeWalk}(x.\text{right}) \\ \text{Print}(x.\text{key}) \end{array} \right. \end{array} \}$ 

proc BSTTreeSearch (x, k)
   $\{ \begin{array}{l} \text{if } ((x = \text{nil}) \text{ or } (x.\text{key} = k)) \\ \quad \text{then return } x \\ \text{if } (k \leq x.\text{key}) \\ \quad \text{then return } \text{BSTTreeSearch}(x.\text{left}, k) \\ \quad \text{else return } \text{BSTTreeSearch}(x.\text{right}, k) \end{array} \}$ 

proc BSTTreeMinimum (x)
   $\{ \begin{array}{l} \text{if } ((x.\text{left} = \text{nil})) \\ \quad \text{then return } x \\ \quad \text{return } \text{BSTTreeMinimum}(x.\text{left}) \end{array} \}$ 

proc BSTTreeSuccessor (x)
   $\{ \begin{array}{l} \text{if } (x.\text{right} \neq \text{nil}) \\ \quad \text{then return } \text{BSTTreeMinimum}(x.\text{right}) \\ y = x.p \\ \text{while } ((y \neq \text{nil}) \text{ and } (x = y.\text{right})) \\ \quad \left\{ \begin{array}{l} x = y \\ y = y.p \end{array} \right. \\ \text{return } y \end{array} \}$ 

proc BSTTreeInsert (T, z)
   $\{ \begin{array}{l} y = \text{nil} \\ x = T.\text{root} \\ \text{while } (x \neq \text{nil}) \\ \quad \left\{ \begin{array}{l} y = x \\ \text{if } (z.\text{key} \leq x.\text{key}) \\ \quad \text{then } x = x.\text{left} \\ \quad \text{else } x = x.\text{right} \end{array} \right. \\ z.p = y \\ \text{if } (y = \text{nil}) \\ \quad \text{then } T.\text{root} = z \\ \text{if } ((y \neq \text{nil}) \text{ and } (z.\text{key} \leq y.\text{key})) \\ \quad \text{then } y.\text{left} = z \\ \text{if } ((y \neq \text{nil}) \text{ and } (z.\text{key} > y.\text{key})) \\ \quad \text{then } y.\text{right} = z \end{array} \}$ 

proc BSTTreeDelete (T, z)
   $\{ \begin{array}{l} \text{if } (z.\text{left} = \text{nil}) \\ \quad \text{then } \text{BSTTransplant}(T, z, z.\text{right}) \\ \text{if } ((z.\text{left} \neq \text{nil}) \text{ and } (z.\text{right} = \text{nil})) \\ \quad \text{then } \text{Transplant}(T, z, z.\text{left}) \\ \text{if } ((z.\text{left} \neq \text{nil}) \text{ and } (z.\text{right} \neq \text{nil})) \\ \quad \text{then} \\ \quad \left\{ \begin{array}{l} y = \text{BSTTreeMinimum}(z.\text{right}) \\ \text{if } (y.p \neq z) \\ \quad \text{then} \\ \quad \left\{ \begin{array}{l} \text{BSTTransplant}(T, y, y.\text{right}) \\ y.\text{right} = z.\text{right} \\ y.\text{right}.p = y \end{array} \right. \\ \text{BSTTransplant}(T, z, y) \\ y.\text{left} = z.\text{left} \\ y.\text{left}.p = y \end{array} \right. \end{array} \}$ 

proc BSTTreeTransplant (T, u, v)
   $\{ \begin{array}{l} \text{if } (u.p = \text{nil}) \\ \quad \text{then } T.\text{root} = v \\ \text{if } ((u.p \neq \text{nil}) \text{ and } (u = u.p.\text{left})) \\ \quad \text{then } u.p.\text{left} = v \\ \text{if } ((u.p \neq \text{nil}) \text{ and } (u = u.p.\text{right})) \\ \quad \text{then } u.p.\text{right} = v \\ \text{if } (v \neq \text{nil}) \\ \quad \text{then } v.p = u.p \end{array} \}$ 

proc BSTTreeLeftRotate (T, x)
   $\{ \begin{array}{l} y = x.\text{right} \\ x.\text{right} = y.\text{left} \\ \text{if } (y.\text{left} \neq T.\text{nil}) \\ \quad \text{then } y.\text{left}.p = x \\ y.p = x.p \\ \text{if } (x.p = T.\text{nil}) \\ \quad \text{then } T.\text{root} = y \\ \text{if } ((x.p \neq T.\text{nil}) \text{ and } (x = x.p.\text{left})) \\ \quad \text{then } x.p.\text{left} = y \\ \text{if } ((x.p \neq T.\text{nil}) \text{ and } (x = x.p.\text{right})) \\ \quad \text{then } x.p.\text{right} = y \\ y.\text{left} = x \\ x.p = y \end{array} \}$ 

```

```

proc RBTreeInsert(T, z)
  y = T.nil
  x = T.root
  while (x ≠ T.nil)
    { y = x
      { if (z.key < x.key)
        { then x = x.left
        else x = x.right
      }
      z.p = y
      if (y = T.nil)
        then T.root = z
      if ((y ≠ T.nil) and (z.key < y.key)
        then y.left = z
      if ((y ≠ T.nil) and (z.key ≥ y.key)
        then y.right = z
      z.left = T.nil
      z.right = T.nil
      z.color = RED
      RBTreeInsertFixup(T, z)
    }
  }

```

```

proc BTreeSplitChild(x, i)
  z = Allocate()
  y = x.ci
  z.leaf = y.leaf
  for j = 1 to t - 1 z.keyj = y.keyj+1
  if (y.leaf = false)
    then for j = 1 to t z.cj = y.cj+1
  y.n = t - 1
  for j = x.n + 1 downto i + 1 x.cj+1 = x.cj
  x.ci+1 = z
  for j = x.n downto i x.keyj+1 = x.keyj
  x.keyi = y.keyt
  x.n = x.n + 1
  DiskWrite(y)
  DiskWrite(z)
  DiskWrite(x)

```

```

proc RBTreeInsertFixup(T, z)
  while (z.p.color = RED)
    { if (z.p = z.p.p.left)
      then RBTreeInsertFixUpLeft(T, z)
    else RBTreeInsertFixUpRight(T, z)
    T.root.color = BLACK
  }

```

```

proc BTreeInsert(T, k)
  r = T.root
  if (r.n = 2 · t - 1)
    then
      { s = Allocate()
        T.root = s
        s.leaf = false
        s.n = 0
        s.c1 = r
        BTreeSplitChild(s, 1)
        BTreeInsertNonFull(s, k)
      else BTreeInsertNonFull(r, k)
    }

```

```

proc RBTreeInsertFixupLeft(T, z)
  y = z.p.p.right
  if (y.color = RED)
    then
      { z.p.color = BLACK
        { y.color = BLACK
          { z.p.p.color = RED
            { z = z.p.p
              else
                { z = z.p.right
                  then
                    { z = z.p
                      LeftRotate(T, z)
                      z.p.color = BLACK
                      z.p.p.color = RED
                      TreeRightRotate(T, z.p.p)
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  }

```

```

proc BTreeInsertNonFull(x, k)
  i = x.n
  if (x.leaf = true)
    then
      { while ((i ≥ 1) and (k < x.keyi))
        { x.keyi+1 = x.keyi
          { i = i - 1
          x.keyi+1 = k
          x.n = x.n + 1
          DiskWrite(x)
        }
      }
    else
      { while ((i ≥ 1) and (k < x.keyi)) i = i - 1
        { i = i + 1
          DiskRead(x.ci)
          if (x.ci.n = 2 · t - 1)
            then
              { BTreeSplitChild(x, i)
                { if (k > x.keyi)
                  then i = i + 1
                  BTreeInsertNonFull(x.ci, k)
                }
              }
            }
          }
        }
      }
    }
  }

```

```

proc RBTreeInsertFixupRight(T, z)
  y = z.p.left
  if (y.color = RED)
    then
      { z.p.color = BLACK
        { y.color = BLACK
          { z.p.p.color = RED
            { z = z.p.p
              else
                { z = z.p.left
                  then
                    { z = z.p
                      TreeRightRotate(T, z)
                      z.p.color = BLACK
                      z.p.p.color = RED
                      TreeLeftRotate(T, z.p.p)
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  }

```

```

proc HashInsert(T, k)
  { let x be a new node with key k
    i = h(k)
    ListInsert(T[i], x)
  }

```

```

proc HashSearch(k)
  { i = h(k)
    return ListSearch(T[i], k)
  }

```

```

proc BTreeSearch(x, k)
  i = 1
  while ((i ≤ x.n) and (k > x.keyi)) i = i + 1
  if ((i ≤ x.n) and (k = x.keyi))
    then return (x, i)
  if (x.leaf = true)
    then return nil
  DiskRead(x.ci)
  return BTreeSearch(x.ci, k)

```

```

proc HashDelete(k)
  { i = h(k)
    x = ListSearch(T[i], k)
    ListDelete(T[i], x)
  }

```

```

proc BTreeCreate(T)
  { x = Allocate()
    x.leaf = true
    x.n = 0
    DiskWrite(x)
    T.root = x
  }

```

```

proc OaHashInsert(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)
  return "overflow"

```

```

proc OaHashSearch (T, k)
  i = 0
  repeat
    j = h(k, i)
    if (T[j] = k)
      then return j
    i = i + 1
  until ((T[j] = nil) or (i = m))
  return nil

```

```

proc TopologicalSort (G)
  for (u ∈ G.V) u.color = WHITE
  L = ∅
  time = 0
  for (u ∈ G.V)
    if (u.color = WHITE)
      then DepthVisitTS(G, u)
  return L

```

```

proc HashComputeModulo (w, B, m)
  let d = |w|
  z0 = 0
  for (i = 1 to d) zi+1 = ((zi · B) + ai) mod m
  return zd + 1

```

```

proc DepthVisitTS (G, u)
  time = time + 1
  u.d = time
  u.color = GREY
  for (v ∈ G.Adj[u])
    if (v.color = WHITE)
      then DepthVisitTS(G, v)
    v.color = BLACK
  time = time + 1
  u.f = time
  ListInsert(L, u)

```

```

proc BreadthFirstSearch (G, s)
  for (u ∈ G.V \ {s})
    u.color = WHITE
    u.d = ∞
    u.π = nil
    s.color = GREY
    s.d = 0
    s.π = nil
    Q = ∅
    Enqueue(Q, s)
    while (Q ≠ ∅)
      u = Dequeue(Q)
      for (v ∈ G.Adj[u])
        if (v.color = WHITE)
          then
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            Enqueue(Q, v)
            u.color = BLACK

```

```

proc StronglyConnectedComponents (G)
  for (u ∈ G.V)
    u.color = WHITE
    u.π = nil
    time = 0
  for (u ∈ G.V)
    if (u.color = WHITE)
      then DepthVisit(G, u)
  for (u ∈ G.V)
    u.color = WHITE
    u.π = nil
    time = 0
    L = ∅
  for (u ∈ GT.V in rev. finish time order)
    if (u.color = WHITE)
      then DepthVisit(GT, u)
    ListInsert(L, u)
  return L

```

```

proc DepthFirstSearch (G)
  for (u ∈ G.V)
    u.color = WHITE
    u.π = nil
    time = 0
  for (u ∈ G.V)
    if (u.color = WHITE)
      then DepthVisit(G, u)

```

```

proc DepthVisit (G, u)
  time = time + 1
  u.d = time
  u.color = GREY
  for (v ∈ G.Adj[u])
    if (v.color = WHITE)
      then
        v.π = u
        DepthVisit(G, v)
        u.color = BLACK
      time = time + 1
      u.f = time

```

```

proc CycleDet (G)
  cycle = False
  for (u ∈ G.V) u.color = WHITE
  for (u ∈ G.V)
    if (u.color = WHITE)
      then DepthVisitCycle(G, u)
  return cycle

```

```

proc MST-Kruskal (G, w)
  T = ∅
  for (v ∈ G.V)
    do
      v.key = ∞
      v.π = nil
    r.key = 0
    Q = G.V
    while (Q ≠ ∅)
      do
        u = ExtractMin(Q)
        for (v ∈ G.Adj[u])
          do
            if ((v ∈ Q) and (W(u, v) < v.key))
              do
                v.π = u
                v.key = W(u, v)

```

```

proc DepthVisitCycle (G, u)
  u.color = GREY
  for (v ∈ G.Adj[u])
    if (v.color = WHITE)
      then DepthVisitCycle(G, v)
    if (v.color = GREY)
      then
        cycle = True
        u.color = BLACK

```

```

proc Union ( $x, y$ )
   $S_1 = \text{FindSet}(x)$ 
   $S_2 = \text{FindSet}(y)$ 
  if ( $S_1 \neq S_2$ )
    then
       $S_1.\text{tail.next} = S_2.\text{head}$ 
       $z = S_2.\text{head}$ 
      while ( $z \neq \text{nil}$ )
         $\{ z.\text{head} = S_1.\text{head}$ 
         $\{ z = z.\text{next}$ 
  return  $S_1$ 

```

```

proc Union ( $x, y$ )
   $S_1 = \text{FindSet}(x)$ 
   $S_2 = \text{FindSet}(y)$ 
  if ( $|S_1| > |S_2|$ )
    then
       $S_2 = \text{FindSet}(x)$ 
       $S_1 = \text{FindSet}(y)$ 
    if ( $S_1 \neq S_2$ )
      then
         $S_1.\text{tail.next} = S_2.\text{head}$ 
         $z = S_2.\text{head}$ 
        while ( $z \neq \text{nil}$ )
           $\{ z.\text{head} = S_1.\text{head}$ 
           $\{ z = z.\text{next}$ 
  return  $S_1$ 

```

```

proc InitializeSingleSource ( $G, s$ )
  for ( $v \in G.V$ )
     $\{ v.d = \infty$ 
     $\{ v.\pi = \text{nil}$ 
   $s.d = 0$ 

```

```

proc Relax ( $u, v, w$ )
  if ( $v.d > u.d + W(u, v)$ )
    then
       $v.d = u.d + W(u, v)$ 
       $v.\pi = u$ 

```

```

proc Bellman-Ford ( $G, w, s$ )
  InitializeSingleSource( $G, s$ )
  for  $i = 1$  to  $|G.V| - 1$ 
    for ( $(u, v) \in G.E$ ) Relax( $u, v, w$ )
    for ( $(u, v) \in G.E$ )
      if ( $v.d > u.d + W(u, v)$ )
        then return false
  return true

```

```

proc Dijkstra ( $G, w, s$ )
  InitializeSingleSource( $G, s$ )
   $S = \emptyset$ 
   $Q = G.V$ 
  while ( $Q \neq \emptyset$ )
     $\{ u = \text{ExtractMin}(Q)$ 
     $\{ S = S \cup \{u\}$ 
    for ( $v \in G.\text{Adj}[u]$ )
      if ( $v \in Q$ )
        then Relax( $u, v, w$ )

```

```

proc ExtendShortestPaths ( $L, W$ )
   $n = L.\text{rows}$ 
  let  $L'$  be a new matrix
  for  $i = 1$  to  $n$ 
    for  $j = 1$  to  $n$ 
       $L'_{ij} = \infty$ 
      for  $k = 1$  to  $n$ 
         $L'_{ij} = \min\{L'_{ij}, L_{ik} + W_{kj}\}$ 
  return  $i + 1$ 

```

```

proc FastAllPairsMatrix ( $W$ )
   $n = L.\text{rows}$ 
   $L^1 = W$ 
   $m = 1$ 
  while ( $m < n - 1$ )
    do
       $\{ L^{2 \cdot m} = \text{ExtendShortestPaths}(L^m, L^m)$ 
       $\{ m = 2 \cdot m$ 
  return  $L^m$ 

```

```

proc Floyd-Warshall ( $W$ )
   $n = W.\text{rows}$ 
   $D^0 = W$ 
  for  $k = 1$  to  $n$ 
    let  $D^k$  be a new matrix
    for  $i = 1$  to  $n$ 
      for  $j = 1$  to  $n$ 
         $D^k_{ij} = \min\{D^{k-1}_{ij}, D^{k-1}_{ik} + D^{k-1}_{kj}\}$ 

```

```

proc SlowAllPairsMatrix ( $W$ )
   $n = L.\text{rows}$ 
   $L^1 = W$ 
  for  $m = 2$  to  $n - 1$ 
     $L^m = \text{ExtendShortestPaths}(L^{m-1}, W)$ 
  return  $L^{n-1}$ 

```