quasi ordinato Θ(kn) → Θ(n)

	Caso Peggiore	Caso Medio	In Place	Stabile
IS	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓
MS	$\Theta(n\lg(n))$	$\Theta(n\lg(n))$	$T(n) = 2T(\frac{n}{2}) + \Theta(n)$	✓
✦ QS	$\Theta(n^2)$	Caso migliore: $\Theta(n\lg(n))$	✓*	** → CP: scelta sbagliata del PIVOT
CS	$\Theta(n)$	$\Theta(n)$		✓
RS	$\Theta(n)$	$\Theta(n)$		✓
SS	$\Theta(n^2)$	$\Theta(n^2)$	✓	non necessar.
HS	$\Theta(n\lg(n))$	$\Theta(n\lg(n))$	✓*	**

* In place a meno delle chiamate ricorsive (tail ricorsive)
** Tutti gli algoritmi di ordinamento possono essere resi stabili

Simmetria: aRb → bRa
Transitiva: aRb, bRc → aRc.
Riflessiva: aRa

Se $f(n) = \Theta(g(n))$
↳ $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$

$\log_a b = c \quad a^c = b \quad a, b > \emptyset \quad a \ne 1$

$\log_a(\frac{b}{c}) = \log_a b - \log_a c$, $\log_a b^c = c \log_a b$

> sotto opportune ipotesi
– CS come algoritmo interno

Se partizione molto iniqua: $O(n \log(n))$
✦ Randomizzo la scelta del PIVOT
↳ RandomizedQS() → caso medio $O(n\lg(n))$

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

$$\rightarrow T(n) = \begin{cases} \Theta(n^{\lg_b(a)}) & \text{se } f(n) = O(n^{\lg_b(a) - \varepsilon}) \quad \varepsilon > 0 \\ \Theta(n^{\lg_b(a)} \lg^{k+1}(n)) & \text{se } f(n) = \Theta(n^{\lg_b(a)} \lg^k(n)) \quad k \ge 0 \\ \Theta(f(n)) & \text{se } f(n) = \Omega(n^{\lg_b(a) + \varepsilon}), \exists c < 1 \mid af(\frac{n}{b}) \le c f(n) \ \forall n > n_0 \end{cases}$$

$2^3 + 2^5 = 2^{3+5}$, $(2^3)^5 = 2^{3 \cdot 5}$, $5^2 \cdot 3^2 = 15^2$

Implementare una coda circolare consente di risparmiare spazio in memoria

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.

* = (n(n+1))/2   ** = $\frac{n}{2}(n-1)$
CASO PEGGIORE: $T(n) = an^2 + bn + c = \Theta(n^2)$

```
proc InsertionSort (A)
n·c1          for (j = 2 to A.length)
(n-1)c2         key = A[j]       A[1,..,j-1] è sempre ordinato in maniera non decrescente.
(n-1)c3         i = j - 1        Dopo ogni esec. l'elemento j-esimo è ins. ordinatamente nell'oggetto A
c4 Σ_2^n t_j *  while ((i > 0) and (A[i] > key))
c5 Σ_2^n (t_j - 1)**  A[i + 1] = A[i]
c6 Σ_2^n (t_j - 1)    i = i - 1
(n-1)c7         A[i + 1] = key
```

anche quando quasi ordinato ← nel caso ottimo $\Theta(n)$ ← $t_j = 1 \ \forall j$ ← input ordinato all'inverso $T(n) = an + b$

Ricerca del max-min: $O(n)$ ← max e min $O(n)$

```
proc RecursiveBinarySearch (A, low, high, k)   data una chiave k, restituisce se esiste, il suo indice nell'array
  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)
```

$\Theta(\log(n))$ ← $T(n) = T(\frac{n}{2}) + 1$

```
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)   Nel 3° ciclo for A[p,...,k-1] contiene i k-p el. più piccoli di L e R ordinati e L[i] e R[j] sono i + piccoli el. di L e R non ancora copiati in A
  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]   Θ(n1)
  for (j = 1 to n2) R[j] = A[q + j]       Θ(n2)   = Θ(n), n = R-p+1
  i = 1
  j = 1   Non ha senso parlare di caso pessimo o medio, i cicli vincolati si comportano allo stesso modo in tutti i casi - Θ(n)
  for (k = p to r)  Θ(n)
    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)
```

Dopo ogni chiamata ricorsiva, A[p,R] è ordinato

```
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)
```

All'inizio di ogni iterazione per ogni indice k: se p≤k≤i ⇒ A[k]<x e se k=R ⇒ A[k]=x

```
proc Partition (A, p, r)   Gli elementi più grandi di x a destra e gli altri a sinistra.
  x = A[r]  ← PIVOT                                   Θ(n)
  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  = Θ(n)   └ non sta facendo i = i+1 !!!
```

Al termine di ogni ricorsione p,R, A[p,...,R] è ordinato

```
proc QuickSort (A, p, r)
  if (p < r)   CP: T(n) = T(n-1) + Θ(n) = Θ(n²)  └ partizione sbilanciata 0 e n-1
    then       CM: 2T(n/2) + Θ(n) = Θ(n lg(n))  └ partizione equa
      q = Partition(A, p, r)
      QuickSort(A, p, q - 1)
      QuickSort(A, q + 1, r)
```

CMe: $O(n\lg(n)) = T(n) = 2/n \sum_0^{n-1} (T(k) + T(n-k-1)) + \Theta(n)$

```
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
```

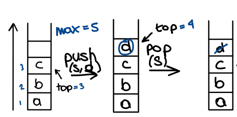
```
proc RandomizedQuickSort (A, p, r)   an/4 > b + Θ(n)
  if (p < r)
    then                                    Random CP O(n)
      q = RandomizedPartition(A, p, r)
      RandQuickSort(A, p, q - 1)
      RandQuickSort(A, q + 1, r)
```

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

LIFO pila

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

$O(1)$




```
proc Pop (S)
  if (Empty(S))
    then return "underflow"
  S.top = S.top - 1   ultima casella occupata "più in alto"
  return S[S.top + 1]
```

FIFO-coda

```
proc Enqueue (Q, x)   ← coda   iniz. Q.head Q.tail = 1, dim = 0 (per diff pila vuota e pila piena)
  if (Q.dim = Q.length)
    then return "overflow"
  Q[Q.tail] = x        O(1)
  if (Q.tail = Q.length)
    then Q.tail = 1 •
    else Q.tail = Q.tail + 1
  Q.dim = Q.dim + 1
```

```
proc Dequeue (Q)   • array circolare
  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
```

Heap code liste

per pile su liste — O(1)

```
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
```

per code su liste — 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
```

```
proc Enqueue (A, i, priority)
  if (i > A.length)
    then return "overflow"
  A[i] = priority
```

```
proc DecreaseKey (A, i, priority)
  if ((A[i] < priority) or (A[i].empty = 1))
    then return "error"
  A[i] = priority
```

```
proc ExtractMin (A)
  MinIndex = 0
  MinPriority = ∞
  for (i = 1 to A.length)
    if ((A[i] < MinPriority) and (A[i].empty = 0))
      then
        MinPriority = A[i]
        MinIndex = i
  if (MinIndex = 0)
    then return "underflow"
  A[MinIndex].empty = 1
  return MinIndex
```

A: max numero di archi da R a f. → minimo 2^h el., massimo $2^{h+1}-1$ $h = \log(n)$
il minimo el di una min-heap è alla radice

```
proc Parent (i)
  {return ⌊i/2⌋
```

```
proc Left (i)
  {return 2 · i
```

```
proc Right (i)
  {return 2 · i + 1
```

Dopo ogni chiamata su un nodo di altezza h tale che entrambi i figli sono radici di min-heap prima della call, quel nodo è la R. di una min-heap

è un array visto ad albero quasi completo

```
proc MinHeapify (H, i)        MaxHeapify(H, i)
  l = Left(i)      CP: Θ(h) = Θ(ln(n))   C.M.: Θ(1) ← no chiamate ricorsive
  r = Right(i)
  smallest = i = biggest
  if ((l ≤ H.heapsize) and (H[l] ≶ H[i]))
    then smallest = l = biggest
  if ((r ≤ H.heapsize) and (H[r] ≶ H[smallest biggest]))
    then smallest = r = biggest
  if smallest ≠ i ≠ biggest
    then
      SwapValue(H, i, smallest biggest)
      MinHeapify(H, smallest biggest)
```

CP: Θ(h) = Θ(lg(n))
CM: Θ(1) ← no call Ric.

In una min heap l'estrazione del minimo costa O(log(n))

alberi quasi completi: All'inizio di ogni iterazione for ogni H[i+1], H[i+2], … è la radice di una min-heap e all'uscita anche H[i]

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

Se in un albero bin. quasi completo ci sono n el. → max $\frac{n}{2^h+1}$ sono ad alt. h

```
proc HeapSort (H)
  Θ(n)     BuildMaxHeap(H)
           for (i = H.length downto 2)
  O(1)       SwapValue(H, i, 1)
             H.heapsize = H.heapsize − 1
  Θ(lg(n))   MaxHeapify(H, 1)
```

Θ(n lg(n)), non stabile, in place

coda di priorità su minheap ← iniz. Θ(n)

```
proc Enqueue (Q, priority)     el. 5 4 ← priorità (massima)
  if (Q.heapsize = Q.length)
    then return "overflow"
  Q.heapsize = Q.hepsize + 1
  Q[heapsize] = ∞
  DecreaseKey(Q, Q.heapsize, priority)
Θ(log(n))
```

```
(max heap) Increase
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(Q, i, Parent(i))
    i = Parent(i)
Θ(log(n))
```

```
proc ExtractMin (Q)     ← prossimo el. da servire
  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
Θ(log(n))  n = numero di el della heap
```

L'estrazione del max da una min heap è O(n)

code di priorità su array → iniz Θ(n)

```
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)
```

```
proc ExtractMin (Q)
  MinIndex = 0
  MinPriority = ∞
  for (i = 1 to Q.length)
    if ((Q[i] < MinPriority) and (Q[i].empty = 0))
      then
        MinPriority = Q[i]
        MinIndex = i
  if (MinIndex = 0)
    then return "underflow"
  Q[MinIndex].empty = 1
  return MinIndex
Θ(n)
```

Se invertiamo l'ordine dell'ultimo for → non stabile

ogni el. dell'input è un intero che varia tra 0 e k. Quando k = O(n) allora T = Θ(n) ← Θ(n+k)

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

al j-esimo passo C[A[j]] è la posizione corretta di A[j] in B

Dopo la i-esima esec. di for gli el. delle ultime i col. sono ordinati

```
Θ(d(n+k)) CP: O(d · f(n))
proc RadixSort (A, d)
  {for (i = 1 to d) AnyStableSort(A) on digit i
```

L ordina dal meno al più significativo

```
Θ(1)
proc ListInsert (L, x)     el. da inserire in testa
  x.next = L.head
  if (L.head ≠ nil )
    then L.head.prev = x
  L.head = x
  x.prev = nil
```

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

```
Θ(1)
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
```

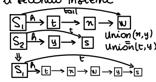
LISTE

```
proc MakeSet (𝒮, S, x, i)
  { S[i].set = x
    S.head = x
    S.tail = x
```

O(1)

Commutativa, distrugge il secondo insieme

```
proc Union (x, y)
  { S1 = x.head
    S2 = y.head
    if (S1 ≠ S2)
      then
      { S1.tail.next = S2.head
        z = S2.head
        while (z ≠ nil )
        { z.head = S1
          z = z.next
        S1.tail = S2.tail
```



$\Theta(n^2)$ n^2 è la lunghezza della seconda lista

```
proc FindSet (x)
  { return x.head.head
```

O(1)

↳ n makeSet e poi n-1 Union nel caso peggiore $\Theta(n \cdot n) = \Theta(n^2)$

Aggiornamenti sull'insieme + piccolo

```
proc Union (x, y)
  { S1 = x.head
    S2 = y.head
    if (S1 ≠ S2)
      then
      { if (S2.rank > S1.rank)
          then
          { Stemp = S1
            S1 = S2
            S2 = Stemp
        S1.tail.next = S2.head
        z = S2.head
        while (z ≠ nil )
        { z.head = S1
          z = z.next
        S1.tail = S2.tail
        S1.rank = S1.rank + S2.rank
```

$\Theta(n\lg(n))$

LISTE CON UNIONE PESATA

```
proc MakeSet (𝒮, S, x, i)
  { S[i].set = x
    S.head = x
    S.tail = x
    S.rank = 1
```

Rango: limite superiore dell'altezza
No puntatori ai figli
Nei nodi il rango
FORESTE di ALBERI

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

```
proc Union (x, y)
  { 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
```

compressione del percorso

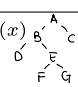
```
proc FindSet (x)
  { if (x ≠ x.p)
      then x.p = FindSet(x.p)
    return x.p
```

O(m) – O(h) nel CP

↳ $h = \Theta(\log(n))$
altrimenti $h = \Theta(n)$ nel C.P.

BST

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

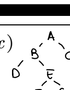


DBFEGAC

K: nr. nodi sotto albero sx

$T(n) = T(K) + T(n-K-1) + d^{\leftarrow cost.} = \Theta(n)$

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



ABDEFGC

DFGEBCA

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

BST: a sinistra le chiavi minori ← parzialmente ordinato

radice dell'albero

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

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

$\Theta(h) < \Theta(n)$ / $\Theta(\log(n))$

```
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
```

$\Theta(h)$

la posizione corretta di z è nel sottoalbero radicato in x e y ne mantiene il padre

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

$\Theta(h)$

non commutativa

```
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, y, y.right)
            y.right = z.right
            y.right.p = y
        BSTTransplant(T, z, y)
        y.left = z.left
        y.left.p = y
```

$\Theta(h)$

trapiantiamo v su u

```
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
```

```
proc BSTTree~~Left~~RightRotate (T, x)
  { y = x.~~right~~ left
    x.~~right~~ left = y.~~left~~ right
    if (y.~~left~~ right ≠ T.nil )
      then y.~~left~~ right.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~~ right = x
    x.p = y
```

$\Theta(1)$ conserva p. BST

RB: bilanciati per costruzione, $h = \log(n)$, $bh(x)$: numero nodi neri da x (esc.) a foglia esterna (in.)

1. Ogni nodo è rosso o nero
2. La radice è nera
3. Ogni foglia è nera
4. Se un nodo è rosso, i suoi figli sono neri
5. $\forall$ nodo, tutti i percorsi semplici da lui alle foglie contengono lo stesso numero di nodi neri.

Un albero B è sempre completo
$t \geq 2$ grado minimo
1. Ogni nodo ha $t-1 \leq$ chiavi $\leq 2t-1 \leftarrow 2$
3. Per ogni nodo x, $x.k_1 \leq x.k_2 \leq x.k_3 \dots$
4. $\forall x$ se y è contenuto nel sottoalbero in $x.c_i \rightarrow$ tutte le $y.k$ sono $\leq x.k_i$
5. e viceversa

Un nodo interno ha massimo $2t$ figli
$h_{max} \leq \log_t\left(\frac{n+1}{2}\right)$

Funzione hash:
- chiavi naturali, m primo e lontano da potenza di 2
  metodo della divisione: $h(k) = \underbrace{k \bmod m}_{13 \bmod 3 = 1} + 1$
- chiavi naturali, $m = 2^p$
  metodo della moltiplicazione $\lfloor m(kA - \lfloor kA \rfloor) \rfloor + 1$
- chiavi non naturali, m primo lontano da 2^p
  metodo dell'addizione: HCM $\Theta(d)$

Probing lineare: $h(k,i) = ((h'(k)+i) \bmod m) + 1$
Probing quadra.: $= ((h'(k) + c_1 i + c_2 i^2) \bmod m) + 1$

Se z diventa radice violo proprietà 2, se figlio di nodo rosso → 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)
```
$\Theta(\log(n))$

```
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 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
          LeftRotate(T, z)
      z.p.color = BLACK
      z.p.p.color = RED
      TreeRightRotate(T, z.p.p)
```

```
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)
```

Generalizza BSTSearch ← binaria $O(\lg(n))$

```
proc BTreeSearch (x, k)
  i = 1
  while ((i ≤ x.n) and (k > x.key_i)) i = i + 1
  if ((i ≤ x.n) and (k = x.key_i))
    then return (x, i)
  if (x.leaf = true )
    then return nil
  DiskRead(x.c_i)
  return BTreeSearch(x.c_i, k)
```
$O(h) \rightarrow O(\log_t(n))$ max

```
proc BTreeCreate (T)
  x = Allocate()
  x.leaf = true
  x.n = 0
  DiskWrite(x)
  T.root = x
```
$\Theta(1)$

```
proc BTreeSplitChild (x, i)
  z = Allocate()
  y = x.c_i
  z.leaf = y.leaf
  for j = 1 to t − 1 z.key_j = y.key_{j+t}
  if (y.leaf = false )
    then for j = 1 to t z.c_j = y.c_{j+t}
  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)
```
$\Theta(1)$

All'inizio di ogni esecuzione di BTNF x è non pieno e k va inserito nel sottoalbero radicato in x

```
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.c_1 = r
      BTreeSplitChild(s, 1)
      BTreeInsertNonFull(s, k) •
    else BTreeInsertNonFull(r, k)
```
$\Theta(h)$, CPU: $\Theta(t \cdot \log_t(n))$

```
proc BTreeInsertNonFull (x, k)
  i = x.n
  if (x.leaf = true )
    then
      while ((i ≥ 1) and (k < x.key_i))
        x.key_{i+1} = x.key_i
        i = i − 1
      x.key_{i+1} = k
      x.n = x.n + 1
      DiskWrite(x)
    else   dell'else! non del while!
      while ((i ≥ 1) and (k < x.key_i)){i = i − 1}
      i = i + 1
      DiskRead(x.c_i)
      if (x.c_i.n = 2 · t − 1)
        then
          BTreeSplitChild(x, i) •
          if (k > x.key_i)
            then i = i + 1
      BTreeInsertNonFull(x.c_i, k)
```

Se array $\Theta(1)$ per tutto

k diversi
h(k) uguali

inserimento in testa

```
proc HashInsert (T, k)
  let x be a new node with key k
  i = h(k)
  ListInsert(T[i], x)
```
$\Theta(1)$

```
proc HashSearch (k)
  i = h(k)
  return ListSearch(T[i], k)
```
$\Theta(n)$ $\Theta(1+\frac{n}{m})$

```
proc HashDelete (k)
  i = h(k)
  x = ListSearch(T[i], k)
  ListDelete(T[i], x)
```
$\Theta(n)$

OpenHashing: si eliminano liste e chaining, provo posizioni finché non trovo una libera
↳ sequenza di probing

```
proc OaHashInsert (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"
```

Il grado entrante di un nodo è il numero di archi che lo raggiungono
sparso se $|E| << |V|^2$ altrimenti denso
$\Theta(|V|+|E|)$
ordinamento topologico: elenco v solo se ho già elencato tutti i vertici da cui può essere raggiunto
└ non ha senso se il grafo diretto è ciclico
└ non unico

```
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
```

$\Theta(|V|+|E|)$

per ogni v inserito in Q, $v.d \geq d(s,v)$
Quanti archi per raggiungere un altro v da s

```
proc HashComputeModulo (w, B, m)
  let d = |w|
  z_0 = 0
  for (i = 1 to d) z_{i+1} = ((z_i · B) + a_i) mod m
  return z_d + 1
```

```
proc BreadthFirstSearch (G, s)
  for (u ∈ G.V \ {s})
    u.color = WHITE
    u.d = ∞
    u.π = nil
  s.color = GREY
  s.d = 0          ← distanza
  s.π = nil        ← padre
  Q = ∅            | s | a | b | . . .
  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
```

grafo connesso e denso
CP: $\Theta(V^2)$ CMe: $\Theta(|V|+|E|)$

ordine topologico - cicli?

```
proc DepthFirstSearch (G)    da quali nodi è raggiunto
  for (u ∈ G.V)
    u.color = WHITE
    u.π = nil
  time = 0
  for (u ∈ G.V)
    if (u.color = WHITE)
      then DepthVisit(G, u)
```

$\Theta(V^2) - \Theta(V+E)$

```
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)    per sapere se ciclico
  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
```

```
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)
```

Una SCC è un sottoinsieme massimale $V' \subseteq V$ che $\forall u,v \in V'$ $u \rightsquigarrow v$ e $v \rightsquigarrow u$

```
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 = ∅          ↙ grafo trasposto
                   └ direzione degli archi invertita
  for (u ∈ G^T.V in rev. finish time order)
    if (u.color = WHITE)
      then DepthVisit(G^T, u)
    ListInsert(L, u)
  return L
```

$\Theta(|V|+|E|)$ a causa di G^T

MST: albero di copertura minimo - eliminando qualunque albero si ottengono due alberi sconnessi.
arco di peso minimo = arco sicuro

greedy

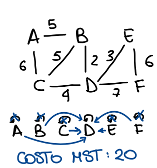
```
proc MST-Prim (G, w, r)
  for (v ∈ G.V)          T è sempre sottoinsieme di
   do                    qualche MST
    v.key = ∞            ← peso minimo
    v.π = nil
  r.key = 0
  Q = G.V                costruzione array Θ|V|
  while (Q ≠ ∅)
   do
    u = ExtractMin(Q)    Θ(V)
    for (v ∈ G.Adj[u])
     do
      if ((v ∈ Q) and (W(u, v) < v.key))
       do
        v.π = u
        v.key = W(u, v)
```

$\Theta(|V|^2)$ ← array (densi) $\Theta(|E| \lg |V|)$ ← heap (sparsi)

(Non è sempre un tree ma lo sarà a fine computazione)

```
proc MST-Kruskal (G, w)
  T = ∅                  Ogni arco aggiunto è
  for (v ∈ G.V)          un arco sicuro
   do MakeSet(v)
  SortNoDecreasing(G.E)
  for ((u, v) ∈ G.E − in order)
   do
    if (FindSet(u) ≠ FindSet(v))
     then
      T = T ∪ {(u, v)}
      Union(u, v)
  return A
```

$\Theta(|V|^2 \lg |V|)$ ← denso, $\Theta(|E| \lg |E|)$ ← sparso



COSTO MST: 20

BD 2
DF 3
CD 4
AB 5
~~BC 5~~
~~AC 6~~
EF 6

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

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

proc ***InizializeSingleSource*** (G, s)
for $(v \in G.V)$
$v.d = \infty$
$v.\pi =$ **nil**
$s.d = 0$

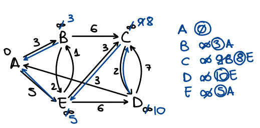
proc ***Relax*** (u, v, w)
if $(v.d > u.d + W(u, v))$
then
$v.d = u.d + W(u, v)$
$v.\pi = u$

↓Il problema sono solo i cicli negativi ← restituisce false

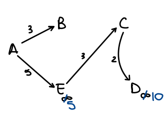
Dopo i esecuzioni del ciclo più esterno $v_i.d = \delta(s, v_i)$ ↙ distanza più corta

proc ***Bellman-Ford*** (G, w, s)
$InizializeSingleSource(G, s)$
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 $\Theta(|V||E|), O(|V|^3)$ ($\Theta(|V|^3)$ se denso)

A volte fallisce se ci sono pesi negativi



proc ***Dijkstra*** (G, w, s)
$InizializeSingleSource(G, s)$
$S = \emptyset$ All'inizio di ogni iteraz.
$Q = G.V$ di while, $v.d = \delta(s, v)$
while $(Q \neq \emptyset)$
$u = ExtractMin(Q)$
$S = S \cup \{u\}$
for $(v \in G.Adj[u])$
if $(v \in Q)$
then $Relax(u, v, w)$



denso coda senza struttura ↓ $\Theta(|V|^2)$ | $\Theta(|E| lg(|V|))$

DAGShortestPath se non ci sono cicli

↙ percorsi minimi tra tutte le coppie di v

proc ***ExtendShortestPaths*** (L, W)
$n = L.rows$
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 $i + 1$

$\Theta(|V|^3 lg(|V|))$

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

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

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

$\Theta(|V|^3)$

DAG SHORTEST PATH