BIBBIA ALGORITMI

La ricerca binaria può essere implementata con un algoritmo simile a quello seguente:

```
BinSearch-Ric(x, A, i, j)
        if i > j then ◀ A[i..j] = ∅
                return false
        else m ← ⌊(i + j)/2⌋ if x = A[m]
                then return true
                else if x < A[m] then return BinSearch-Ric(x, A, i, m − 1)
                        else ◀ A[m] < x return BinSearch-Ric(x, A, m +
                        1, j)
                        end if
                end if
        end if
```

ARRAY STATICI

```
ARRAY_INSERT(A, k) //INSERIMENTO
  if A.N ≠ A.M then
    A.N ← A.N + 1
    A[N] ← k
    return k
  else return nil
```

```
ARRAY_DELETE(A, k)  //CANCELLAZIONE
  for i ← 1 to A.N do
    if A[i] == k then
      A.N ← A.N − 1
      for j ← i to A.N do        A[j] ← A[j + 1]
      return k
  return nil
```

```
ARRAYSEARCH(A, k)  //RICERCA
  for i ← 1 to A.N do
    if A[i] == k then
      return k
  return  nil
```

ARRAY DINAMICI (con dimensione non nota a priori, dunque variabile)

DYN_ARRAY_INSERT(A, k) //INSERIMENTO

if A.N == A.M **then**

A ← ARRAYEXTEND(A, Z)
ARRAYINSERT(A, k)

DYN_ARRAY_DELETE(A, k) // CANCELLAZIONE
ARRAYDELETE(A, k)
if A.N ≤ 1/4 · A.M **then**

B ← un array di dimensione A.M/2
B.M ← A.M/2
B.N ← A.N
for i ← 1 **to** A.N **do**

B[i] ← A[i]
A ← B

ARRAY_EXTEND(A, n) //ESPANSIONE: serve solo per gli array dinamici
B ← un array con A.M + n elementi
B.M ← A.M + n B.N ← A.N
for i ← 1 **to** A.N **do**

B[i] ← A[i]
return B

LISTE

Ricerca: si può fare in due modi. Uno normale e invece uno con due sentinelle

LIST_SEARCH(L, k)
x ← L.head
while x ≠ *nil* **and** x.key ≠ k **do**

x ← x.next **return** x

LIST_SEARCH_SEN(L, k)
x ← L.sen.next
while x ≠ L.sen **and** x.key ≠ k **do**

x ← x.next

return x

Inserimento in **testa**

LIST_INSERT(L, x)
x.next ← L.head
if L.head ≠ *nil* **then**

L.head.prev ← x
L.head ← x
x.prev ← *nil*

LIST_INSERT_SEN(L, x) //con sentinelle

x.next ← L.sen.next

```
L.sen.next.prev ← x
L.sen.next ← x
x.prev ← L.sen
```

Cancellazione

```
LIST_DELETE(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

//con sentinelle
LIST_DELETE_SEN(L, x)
x.prev.next ← x.next
 x.next.prev ← x.prev
```

TAVOLE A INDIRIZZAMENTO DIRETTO

3 semplici operazioni:

```
TABLEINSERT(T, x)
  T[x.key] ← x
TABLEDELETE(T, x)
  T[x.key] ← nil
TABLESEARCH(k)
  return T[k]
```

TAVOLE HASH (si usa come struttura dati di appoggio la lista)

```
HASHINSERT(T, x)
        L ← T[h(x.key)]
        LISTINSERT(L, x)
HASHSEARCH(T, k)
        L ← T[h(k)]
        return LISTSEARCH(L, k)
HASHDELETE(T, x)
        L ← T[h(x.key)]
        LISTDELETE(L, x)
```

TAVOLE HASH (INDIRIZZAMENTO APERTO)

```
HASHINSERT(T, x)
        i ← 0  while i < m do
                j ← h(x.key, i)
```

```
            if T[j] == nil then  T[j]
            ← x  return j  i ← i + 1
        return nil


HASHSEARCH(T, k)
            i ← 0
            while i < m do
                j ← h(k, i)
                if T[j] == nil then
                        return nil
                if T[j].key == k then
                        return T[j]
                i ← i +1
            return nil
```

CODE

Implementazione con array

Note: Q.head indica la posizione da dove estrarre l'elemento successivo
Q.tail indica la posizione dove inserire l'elemento successivo

```
Size(Q)
        if Q.tail >= Q.head then
                return Q.tail – Q.head
        return Q.M – (Q.head – Q.tail)
```

```
Empty(Q)
        if Q.tail == Q.head then
                return true
        return false
```

```
NextCell(Q, c)
if c ≠ Q.M then
        return c + 1
return 1
```

```
Equeue(Q, t)
        if Size(Q) ≠ Q.M – 1 then
                Q[Q.tail] ← t
                Q.tail ← NextCell(Q, Q.tail)
        else
                error overflow
```

```
Front(Q)
        if Size(Q) == 0 then
                error underflow
        else
                return Q[Q.head]
```

```
Dequeue(Q)
        if Size(Q) == 0 then
                error underflow
        else
                t ← Q[Q.head]
                Q.head ← NextCell(Q, Q.head)
                return t
```

Implementazione con lista

```
Enqueue(Q, t)
        if Q.N == 0 then
                Q.head ← t
                Q.tail ←t
        else
                Q.tail.next ← t
                Q.tail ← t
        Q.N ← Q.N + 1
```

```
Size(Q)
        return Q.N
```

```
Empty(Q)
        if Q.N == 0 then
                return true
        return false
```

```
Front(Q)
if Q.N == 0 then
        error underflow
else
        return Q.head
```

```
Dequeue(Q)
if Q.N == 0 then
        error underflow
else
        t ← Q.head
        Q.head ← Q.head.next
        Q.N ← Q.N – 1
        return t
```

ALBERI (BINARI K-ARI)

```
CARDINALITY_BINARY(T)

if T=nil

        return 0

return 1+CARDINALITY_BINARY(T.left) + CARDINALITY_BINARY(T.right)
```

```
CARDINALITY_K(T)
if T=nil
        return 0
C ← 1
S ← T.child
while S != nil
        C ← C + CARDINALITY_K(S)
        S ← S.sibling
return c
```

Oppure potremmo usare CARDINALITY_BINARY con left = child e right = sibling

---

```
HEIGHT_BINARY(T)
if T.left = nil ∧ T.right = nil return 0

if T.left != nil
        as ← Height_binary(T.left) ◀ Altezza a sinistra
else as ← 0
if T.right != nil
        ad ← HeightBinary(T.right) ◀ Altezza a destra
else ad ← 0
return MAX(as,ad) + 1
```

```
HEIGHT_K(T)
if T.child = nil
        return 0
a ← 0
c ← T.child
while c != nil
        a ← MAX(a, Height_k(c))
        c ← c.sibling
return a+1
```

ora vediamo l'algoritmo per la stampa per livelli

```
VISITA_TREE(T)
S ← pila vuota
Push(S,T)
while ! Empty(S)
        T' ← Pop(S) <Stampa
        T'.e> c ← T'.child
```

```
    while c != nil
        Push(S,c)
        c ← c.sibling
```

```
VISITA_GENERICA(T) //visita generica
    F ← empty set
    <Insert T in F>
    while F not empty
        T' ← <Remove node
        from F>
        print T'.key ◀ key = etichetta
        C ← T'.child
        while C != nil
            <Insert C in F>
            C ← C.sibling
```

ALBERI BINARI DI RICERCA

```
Stampa_inOrdine (T)
    if T = nil
        return nil
    Stampa_inOrdine(T.left)
    print(T.key)
    Stampa_inOrdine(T.right)
```

```
SearchRecursive(T,k) ◀ Recursive Search
    if T = nil
        return nil
    if T.key = k
        return T
    if T.key > k

        return SearchRecursvie(T.left, k)
    return SearchRecursvie(T.right,k)
```

```
SearchIterative(T,k) ◀ Iterative Search (è meglio evitare la ricorsione quando possibile)

    S ← T
    while S != nil∧ S.key != k
        if S.key > k
            S ← S.left
```

```
        else
                S ← S.right

        return S
```

```
TrovaMassimo(T) ◀ Trova il massimo. T != nil
        S ← T
        whileS.left != nil
                S ← S.left
        return S
```

```
TrovaSuccessore(N nodo) //trova il successore di un nodo in un albero binario di ricerca
        if N.right != nil
                return TM(N.right)
        else
                P ← N.parent
                while P != nil⋀ N = P.right

                        N ← P
                        P ← N.parent
                return P
```

```
Inserimento_BRT (T tree, N node) // inizialmente N.left = N.right = nil
        P ← nil
        S ← T
        while S != nil
                P ← S
                if N.key = S.key // se l'elemento è già presente
                        return
                if S.key > N.key
                        S ← S.left
                else
                        S ← S.right
        N.parent ← p
        if P = nil // se T era vuoto, N diventa la nuova radice
                T ← N
        else
                if P.key > N.key
```

```
            P.left ← N
        else
            P.right ← N
```

Per quanto riguarda la cancellazione di un elemento da un albero binario di ricerca ci sono tre possibili casi:

- 1) Z è una foglia → si sgancia dal padre
- 2) Z ha esattamente un figlio; chiamata S la radice del suo sottoalbero, dobbiamo riaggiornare i riferimenti da e verso Z in funzione di S
- 3) Z ha due figli → chiamato il sottoalbero destro DS e quello sinistro SS, copio il minimo di SD in Z e tolgo il minimo di SD (che avrà al più un figlio), oppure faccio un ragionamento analogo per il massimo di SS.

```
1_Cancellazione (Z node, T tree) // rimozione con un SOLO figlio. Z è in T

    if Z = T ◀ Z è radice
        if Z.left != nil
            T ← Z.left
        else    T ← Z.right
    else //Riferimento verso l'alto
    if Z.left != nil
        Z.left.parent ← Z .parent S ← Z.left
        else
            Z.right.parent ← Z.parent
            S ← Z.right //Riferimento verso il basso
        If Z.parent.left = Z
            Z.parent.left ← S
        esle
            Z.parent.right ← S
```

```
2_Cancellazione (Z node, T tree) Z è in T: con due figli

    if Z.left = nil∧ Z.right = nil
        if Z = T
            T ← nil
        else
            if Z.parent.left = Z
                Z.parent.left ← nil
            else
                Z.parent.right ← nil
    else
        ifZ.left != nil XOR Z.right != nil
```

```
            1_Cancellazione(Z, T)

        else

            y ← TrovaMinimo(Z.right) ◀ Minimo

            Z.key ← y.key

            2_Cancellazione(y, T)
```

ALBERI ROSSO – NERI

Rotazione a sinistra

```
LEFT-ROTATE(T,x)
    y ← x.right
    x.right ← y.left
    if y.left != nil
        y.left.parent ← x
    y.parent ← x.parent
    if x.parent = nil
        T ← y
    else
        if x = x.parent.left
            x.parent.left ← y
        else
            x.parent.right ← y
    y.left ← x
    x.parent ← y
```

```
RB-INSERT-FIXUP(T, x)          ▷ x è il nodo che può creare problemi
  while x.p.color = red do
    if x.p = x.p.p.left then              ▷ padre di x è figlio sinistro?
      u ← x.p.p.right                     ▷ u è lo zio di x
      if u.color = red then               ▷ caso 1?
        x.p.color ← black                 ▷ caso 1
        u.color ← black                   ▷ caso 1
        x.p.p.color ← red                 ▷ caso 1
        x ← x.p.p                         ▷ caso 1
      else
        if x = x.p.right then             ▷ caso 3?
          x ← x.p                         ▷ caso 3
          LEFT-ROTATE(T, x)               ▷ caso 3
        x.p.color ← black                 ▷ caso 2 e 3
        x.p.p.color ← red                 ▷ caso 2 e 3
        RIGHT-ROTATE(T, x.p.p)            ▷ caso 2 e 3
    else
      {tutto il corpo del if esterno con
      left e right scambiati}             ▷ padre di x è figlio destro
  T.root.color ← black
```

```
RB-DELETE-FIXUP(T, x)                    ▷ x è il nodo che può creare problemi
  while x ≠ T.root ∧ x.color = black do
    if x = x.p.left then                                  ▷ x è figlio sinistro?
      w ← x.p.right                                       ▷ w è il fratello di x
      if w.color = red then                               ▷ caso 4?
        w.color ← black; x.p.color ← red                  ▷ caso 4
        LEFT-ROTATE(T, x.p); w ← x.p.right                ▷ caso 4
      if w.left.color = black ∧ w.right.color = black then ▷ caso 3?
        w.color ← red; x ← x.p                            ▷ caso 3
      else
        if w.right.color = black then                     ▷ caso 2?
          w.left.color ← black; w.color ← red             ▷ caso 2
          RIGHT-ROTATE(T, w); w ← x.p.right               ▷ caso 2
        w.color ← x.p.color; x.p.color ← black            ▷ caso 1
        w.right.color ← black; LEFT-ROTATE(T, x.p)        ▷ caso 1
        x ← T.root                                        ▷ caso 1
    else
      {tutto il corpo del if esterno con left e right scambiati}
  x.color ← black
```

HEAP

Creazione e inizializzazione

PARENT(H,i) ◂ H = heap, i = indice

```
return i/2
```

LEFT(H,i) ◂ Precondizione = precondizione di PARENT; Postcondizione = restituisce l'indice del figlio sinistro se esiste, i altrimenti.

```
if 2i ≤ H.N ◂ Se il figlio sinistro esiste
    return 2*i
else
    return i
```

RIGHT(H,i) ◂ Precondizione sempre uguale; Restituisce l'indice del figlio destro se esiste, i altrimenti.

```
if2i + 1 ≤ H.N
    return 2i + 1
else
    return i
```

Precondizione: H heap, x etichetta (numero intero come nel nostro esempio o qualunque altro tipo purché ordinato). **Postcondizione**: x è inserito e H rimane un heap

HEAP_INSERT(H, x)

```
        H.N←H.N+1
    p ← H.N ◂ Nuova posizione
    H[p] ← x
    while x > H[PARENT(H,p)] ⋀ p > 1 ◂ Se p non è > 1 allora è radice.
        <Scambia H[p] e H[PARENT(H,p)]>
        p ← PARENT(H,p)
```

Precondizione: i sottoalberi di i sono heap, e che 1 ≤ i ≤ H.N

Postcondizione: H è heap

HEAPIFY(H,i) ◂ "Heapifica" una parte del vettore H a partire dalla posizione i

```
    m ← posizione dell'etichetta massima nelle posizioni i, LEFT(H,i), RIGHT(H,i)
    if i != m
        <Scambia H[i] con H[m]>
        HEAPIFY(H,m)
```

Precondizione: H è heap

Postcondizione: H è heap, con etichetta in radice tolta

HEAP-EXTRACT(H)

```
    H[1] ← H[H.N]
    H.N←H.N-1
    HEAPIFY(H,1)
```

Precondizione: V è un vettore di M elementi
Postcondizione: V rappresenta un heap
BUILD-HEAP chiamerà HEAPIFY(V,i) N/2 volte; HEAPIFY costa log n → complessità: O(n log n)

BUILD_HEAP(V)

```
    V.N ← V.M ◀ N = numero di elementi dell'heap
    for i← V.M/2 downto1
        HEAPIFY(V,i)
```

HEAP-SORT(V) ◂ **Precondizione**: V è un vettore di M elementi; **Postcondizione**: V è ordinato BUILD-HEAP(V) ◂ O(n log n)

```
    for i← V.N downto2 ◂ n log n
        <Scambia V[1] con V[i]>
        V.N←V.N-1
        HEAPIFY(V,1)
```

ALGORITMI SULL VISITE DI GRAFI

Visita(G,s) ◂ G= grafo, s = nodo, s ∈ V

s.c ← grigio ◂ .c sta per “colour”

while ∃ u ∈ V tale che u.c = grigio

u ← nodo grigio ◂ è sempre la stessa u del *while*. Non vediamo come è scelto. **if** ∃ v ∈ V tale che (u,v) ∈ E ∧ v.c = bianco

v ← nodo bianco adiacente di u ◂ come sopra.

v.c ← grigio

else

u.c ← nero

Ora introduciamo gli attributi di inizio/fine visita e un attributo per avere riferimento al padre

Visita(G,s) ◂ G= grafo, s = nodo, s ∈ V

s.c ← grigio ◂ .c sta per “colour”

s.i ← 1

s.π ← *nil*

t ← 2 ◂ “Timer”

while ∃ u ∈ V tale che u.c = grigio

u ← nodo grigio ◂ è sempre la stessa u del *while*. Non vediamo come è scelto.

if ∃ v ∈ V tale che (u,v) ∈ E ∧ v.c = bianco

v ← nodo bianco adiacente di u ◂ come sopra.

v.c ← grigio

v.i ← t

t ← t + 1

v. π ← u

else

u.c ← nero

u.f ← t

t ← t + 1

VISITA IN PROFONDITA' DEI GRAFI (DFS)

```
INIT(G)
    for v ∈ V
        v.π ← nil
        v.c ← b
        v.i ← ∞
        v.f ← ∞
    time ← 1
```

```
VisitAll(G)
    INIT(G)
    while ∃ s ∈ V tale che s.c = b
        Visit(G,s) // vedere algoritmo appena sotto
```

```
Visit(G,s)
    S ← <Empty Stack>
        s.i ← time
    s.c ← g
    time++
    Push(S,s) ◂ Inserisco nella pila (=stack) S l'elemento s
    whileNonEmpty(S)
        u ← Top(S)
        if ∃ v ∈ V tale che (u,v) ∈ E ∧ v.c = b ◂ E: insieme degli archi (Edges)
            v ← nodo tale che (u,v) ∈ E ∧ v.c = b
            v.i ← time
            time++
            v.π ← u
            v.c ← g
            Push(S,v)
        else
            u.f ← time
            time++
            u.c ← n
            Pop(S)
```

Quando usiamo uno stack spesso stiamo usando la ricorsione, che possiamo usare anche per implementare l'algoritmo di visita.

```
Visit(G,s)
    s.i ← time
    time++
    s.c ← g
    while ∃ v ∈ V tale che (s,v) ∈ E ∧ v.c = b
        v ← nodo tale che (s,v) ∈ E ∧ v.c = b
        v.π ← s
        Visit(G,v)
    s.f ← time
    time++
    s.c ← n
```

```
CFC(G,s) ◂ Calcola le COMPONENTI FORTEMENTE CONNESSE DI s
    Init(G)
    Visit(G,s)
    A ← insieme dei nodi neri
    F←G^T
    Init(F)
    Visit(F,s)
    B ← insieme dei nodi neri
    return A ∩B
```

```
CFC(G) ◂ Calcola TUTTE le COMPONENTI FORTEMENTE CONNESSE
    Init(G)
    VisitAll(G)
    l ← Nodi in ordine decrescente di f (attributo di fine visita)
    F←G^T
    Init(F)
    VisitAll(F)
    //Ciascun albero corrisponde ad una cfc dopo la visita di F
```

ALGORITMO KRUSKAL

MST-Kruskal (G) //G : grafo (V,E); Complessità: O (|E| log |V|)

A ← *insieme vuoto*

for *per ogni* v Є V

make_set(v) //crea l'insieme con il vertice v

O ← *archi in ordine crescente di peso*

for *per ogni* (u, v) Є E nell'ordine O

if Find (u) ≠ Find (v) **then** //non hanno ancora un cammino quindi non

A ← A ∪ (u, v) //creano ciclo

Union (u, v) //per unire i due insiemi

ALGORITMO DI PRIMM

Pre: G : grafo; s : nodo di partenza; *d* : attributo per il riferimento all'arco più leggero per raggiungere un nodo; Π : padre di ogni nodo. (Primm si implementa con una coda di priorità).

Primm (G, s)

A ← insieme vuoto

for *per ogni* v Є V

v.*d* ← infinito

v. Π ← nil

s.*d* ← 0

Q ← creo coda di priorità con TUTTI i nodi

while Q not_empty

u ← estraggo dalla coda Q il nodo con il valore *d* più basso

for *per ogni* v Є adj(u)

if v Є Q AND w(u, v) < v.*d*

v.*d* ← w(u, v)

v.Π ← u

ALGORITMO DI DIJKSTRA

Dijkstra (G, s) //V: insieme di nodi di cui conosco il cammino minimo, D: nodi da scoprire

for *per ogni* v Є V

v.*d* ← infinito

v.∏ ← nil

s.*d* ← 0

Q ← coda con priorità

while Q not_empty

u ← nodo con *d* minimo in Q

for v Є adj(u)

if v Є Q AND v.*d* > u.*d* + w(u, v)

v.*d* ← u.*d* + w(u, v)

v.∏ ← u