Tipo Dizionario (array)
Composizione struttura:
key: chiave dell'elemento
info: valore collegato alla chiave key

S = array s
k = chiave k da trovare
p = primo elemento dell'array
r = ultimo elemento dell'array

```
searchindex(S, k, p, r) -> indice
      if p > r
            return -1
      else
            m = parteIntera( (p+r)/2 )
            if S[m].key == k
                  return m
            else if S[m].key > k
                  return searchindex(S, k, p, m-1)
            else
                  return searchindex(S, k, m-1, r)
```

Complessità: TETA(log(n))

S = array s
e = nuova info da inserire (collegata alla chiave k)
k = nuova chiave da inserire

```
insert(S, e, k)
      reallocate(S, S.length+1)
      i = 1
      while i<S.length and S[i].key<k
            i=i+1
      for j=S.length downTo i+1
            S[j] = S[j-1]
      S[i].key = k
      S[i].info = e
```

Complessità: O(n)

S = array s
k = chiave k da trovare

```
search(S, k) -> valore
      i = searchindex(S, k, 1, S.length)
      if i==-1
            return NULL
      else
            return S[i].info
```

Complessità: O(log(n))

S = array S
k = chiave k dell'elemento da eliminare

```
delete(S, k)
      i = searchindex(S, k, 1, S.length)
      for j=i to S.length-1
            S[j]=S[j+1]
      reallocate(S, S.length-1)
```

Complessità: TETA(log(n)) + O(n) = O(n)

Tipo Dizionario basato su Liste

Nodo della lista:

elem: info del nodo
chiave: chiave del nodo
prev: nodo precedente
next: nodo successivo

Lista:

head: testa della lista

L = Dizionario L
e = info da inserire in L (collegata alla chiave k)
k = chiave da inserire in L

```
insert(L, e, k)
      p = creaNodo(e, k)
      p.next = L.head
      if L.head != NULL
            L.head.prev = p
            L.head = p
            p.prev = NULL
```

Complessità: TETA(1)

L = Dizionario L
k = chiave k dell'info da ricercare

```
search(L, k) -> valore
      x = L.head
      while x!=NULL and x.key!=k
            x=x.next
      if x!=NULL
            return x.info
      else
            return NULL
```

Complessità: O(n)

L = Dizionario L
k = chiave dell'elemento da eliminare

```
delete(L, k)
      x=L.head
      while x!=NULL and x.key!=k
            x=x.next
      if x.next!=NULL
            x.next.prev = x.prev
      if x.prev!=NULL
            x.prev.next = x.next
      else
            L.head = x.next
      remove(x)
```

Complessità: O(n)

Stack tramite Array
Composizione Stack:
v: vettore
top: indice dell'ultimo elemento inserito

```
initstack() -> stack
      S = allocate(n)
      S.top=0
      return S
```

Complessità: TETA(1)

S = Stack

```
stack_empty(S) -> bool
      if S.top==0
            return true
      else
            return false
```

Complessità: TETA(1)

S = Stack
e = elemento da inserire nello Stack

```
push(S, e)
      S.top = S.top+1
      S.v[S.top] = e
```

Complessità: TETA(1)

S = Stack

```
pop(Stack S) -> elem
      S.top=S.top-1
      return S.v[S.top+1]
```

Complessità: TETA(1)

S = Stack S
```
top(S) -> val
      return S.v[S.top]
```

Coda FIFO basata su Array Circolari
Composizione della coda:
v = vettore
head = indice del vettore che rappresenta la testa della coda
tail = indice del vettore in cui viene inserito l'elemento

```
initqueue() -> coda
      Q = allocate(n)
      Q.tail = Q.head = 1
      return Q
```

Complessità: TETA(1)

Q = Coda Q
```
queue_empty(Q) -> bool
      if Q.head==Q.tail
            return true
      else
            return false
```

Complessità: TETA(1)

Q = Coda Q
e = elemento da inserire nella Coda
```
enqueue(Q, e)
      Q.v[Q.tail] = e
      if Q.tail == Q.v.length
            Q.tail = 1
      else
            Q.tail = Q.tail + 1
```

Complessità: TETA(1)

Q = Coda Q
```
dequeue(Q) -> elem
      x = Q.v[Q.head]
      if Q.head == Q.v.length
            Q.head = 1
      else
            Q.head = Q.head+1
      return x
```

Complessità: TETA(1)

Q = Coda Q
```
first(Q) -> elem
      return Q.v[Q.head]
```

Complessità: TETA(1)

Coda FIFO implementata con due Stack
Composizione Coda:
s1: primo Stack
s2: secondo Stack

Q = Coda Q
e = elemento e da inserire nella Coda
```
enqueue(Q, e)
      push(Q.s1, e)
```

Complessità: TETA(1)

Q = Coda Q
```
dequeue(Queue Q) -> elem
      if stack_empty(Q.s2)
            while not stack_empty(Q.s1)
                  push(Q.s2, pop(Q.s1))
      return pop(Q.s2)
```

Complessità: TETA(n)

Stack implementato con due Code FIFO
Composizione Stack:
q1: prima Coda
q2: seconda Coda

S = Stack S
e = elemento e da inserire nello Stack
```
push(S, e)
      enqueue(S.q1, e)
```

Complessitò: TETA(1)

S = Stack S
```
pop(S) -> elem
      x = dequeue(S.q1)
      while not queue_empty(S.q1)
            enqueue(S.q2, x)
            x = dequeue(S.q1)
      p = S.q1
      S.q1 = S.q2
      S.q2 = p
      return x
```

Complessità: TETA(n)

Lista Semplice
Composizione Nodo:
- *key*: valore del Nodo
- *next*: prossimo Nodo

Composizione Lista:
- *head:* Nodo in testa alla Lista

Lista Doppia
Composizione Nodo:
- *key*: valore del Nodo
- *next:* prossimo Nodo
- *prev*: Nodo precedente

Composizione Lista:
- *head:* Nodo in testa alla Lista

Lista Doppia con Nodo Sentinella
Composizione Nodo:
- *key:* valore del Nodo
- *next:* prossimo Nodo
- *prev*: Nodo precedente

Composizione Lista:
- *null:* Nodo Sentinella

```
create() -> lista
      L = allocate()
      L.null.prev = L.null
      L.null.next = L.null
      return L
```

L = Lista
x = Nodo da eliminare
```
delete(L, x)
      x.prev.next = x.next
      x.next.prev = x.prev
      remove(x)
```

L = Lista
x = Nodo da inserire
```
insert(L, x)
      x.next=L.null.next
      x.next.prev=x
      x.prev=L.null
```

Lista rappresentate con più array
Composizione Lista:
- *next:* vettore contenente l'indice del prossimo elemento nell'array key
- *key*: vettore contenente gli elementi della lisa
- *prev*: vettore contenente l'indice dell'elemento precedente nell'array key
- *free:* indice dell'array next che indica il prossimo elemento allocabile

```
create() -> list
      L = allocate(n)
      L.prev[1] = 0
      L.next[n-1] = 0
      return L
```

L = Lista
```
allocate_object(L) -> indice
      if L.free==0
            errore "Spazio esaurito!"
      else
            x = L.free
            L.free = L.next[x]
      return x
```

L = Lista
x = indice dell'elemento da liberare
```
free_object(L, x)
      L.next[x]=L.free
      L.free=x
```

Rappresentazione di un albero tramite array
Composizione Nodo:
info: valore del Nodo
parent: indice del padre
Composizione Albero:
p: vettore dei nodi dell'albero

T = Albero T
v = Indice del nodo di cui si vuole sapere chi è il padre
```
padre(T, v) -> indicePadre
      if(T.p[v]==0)
            return NULL
      else
            return T.p[v].parent
```

Complesssità: TETA(1)

T = Albero T
v = Indice del nodo di cui vogliamo sapere i figli
```
figli(T, v) -> lista
      l = creaLista()
      for i=1 to n
            if T.p[i].parent == v
                  aggiungi(l, v)
      return l
```

Complessità: TETA(n)

Rappresentazione Albero con vettori posizionali
Composizione Albero:
n: vettore dei nodi dell'albero

P = Albero P
v = indice del nodo di cui si vuole sapere chi è il padre

```
padre(P, v) -> indicePadre
      if v==0
            return NULL
      else
            return parteIntera( (v-1)/k )
```

Complessità: TETA(1)

P = Albero P
v = indice del nodo in cui si vuole sapere quali sono i figli

```
figli(P, v) -> lista
      l = creaLista()
      if v*k+1 >= n
            return l
      else
            for i=0 to k-1
                  aggiungi(l,v)
      return l
```

Complessità: TETA(k)

Rappresentazione con strutture collegate (figlioSX, fratelloDX) di alberi binari
Composizione Nodo:
key: valore del Nodo
p: puntatore al padre
left_child: figlio più a sinistra
right_sibiling: fratello a destra
Composizione Albero:
root: Nodo radice

v = Nodo v di cui si vogliono sapere i figli

```
figli(v) -> lista
      l = creaLista()
      iter = v.left_child
      while iter != NULL
            aggiungi(l, iter)
            iter = iter.right_sibiling
      return l
```

u = Nodo di un albero

```
dimezzaValoriInLVPari(u)
      if u!=NULL
            u.key = u.key/2
            dimezzaValoriInLVPari(u.right_sibiling)
            iter = u.left_child
            while iter!=NULL
                  dimezzaValoriInLVPari(iter.left_child)
                  iter=iter.right_sibiling
```

Complessità: TETA(n)

Rappresentazione con strutture collegate di alberi binari

Composizione Nodo:

p: puntatore al padre
left: puntatore al figlio sinistro
right: puntatore al figlio destro
key: valore del nodo

v = Nodo di un albero

```
padre(v) -> nodoPadre
      return v.p
```

Complessità: TETA(1)

v = Nodo di un albero

```
figli(v) -> lista
      l = creaLista()
      if v.left != NULL
            aggiungi(l, v.left)
      if v.right != NULL
            aggiungi(l, v.right)
      return l
```

Complessità: TETA(1)

r = Nodo di un albero

```
visitaDFS(r)
      S = initstack()
      push(S,r)
      while not stack_empty(S)
            u = pop(S)
            if u!=NULL
                  visita u
                  push(S, u.left)
                  push(S, u.right)
```

Complessità: O(n)

r = Nodo di un albero

```
visitaBFS(r)
      C = initqueue()
      enqueue(C, r)
      while not queue_empty(C)
            u = dequeue(C)
            If u!=NULL
                  visita u
                  enqueue(C, u.left)
                  enqueue(C, u.right)
```

Complessità: O(n)

u = Nodo di un albero

```
altezza(u) -> h
      if u==NULL
            return -1
      else
            return 1 + max(altezza(u.left), altezza(u.right))
```

Complessità: TETA(n)

u = Nodo di un albero

```
numFoglie(u) -> tot
      if u==NULL
            return 0
      else
            if u.left==NULL and u.right==NULL
                  return 1
            else
                  return foglie(u.left) + foglie(u.right)
```

Complessità: TETA(n)

r = albero di cui si vuole sapere il t_bilanciamento

```
tbilanciato(r) -> <t_bilanciamento, altezza>
      if r == NULL
            return <0, -1>
      <tl, hl> = tbilanciato(r->left)
      <tr, hr> = tbilanciato(r->right)
      max = hl - hr
      if max < 0
            max = max * (-1)
      if max < tl
            max = tl
      if max < tr
            max = tr
      if hl > hr
            h = hl + 1
      else
            h = hr + 1
      return <max, h>
```

Complessità: O(n)

u = Nodo di un albero
sum = somma per determinare se un nodo è nodo centrale

```
contaNodiCentrali(u, sum) -> <numNodiCentrali, numFoglie>
      if u==NULL
            return <0,0>
      else
            if u.left==NULL and u.right==NULL
                  foglie=1
                  nodi = 0
            else
                  <numCentroSX, foglieSX> =
contaNodiCentrali(u.left, sum+u.key)
                  <numCentroDX, foglieDX> =
contaNodiCentrali(u.right, sum+u.key)
                  foglie = foglieSX + foglieDX
                  nodi = numCentroSX + numCentroDX
      if foglie == sum+u.key
            nodi = nodi+1
      return <nodi, foglie>
```

Complessità: TETA(n)

vant = Array di valori di un albero visitati in preordine
vsim = Array di valori di un albero visitati in ordine simmetrico
infant = indice inferiore di vant
supant = indice superiore di vant
infsim = indice inferiore di vsim
supsim = indice superiore di vsim

```
ricostruisci(vant, vsim, infant, supant, infsim, supsim) ->
albero
      if infant > supant
            return NULL
      if infant == supant
            return creaNodo(vant[infant])
      i = infsim
            while vsim[i] != vant[infant]
                  i++
      r = creaNodo(vant[infant])
      r.left = ricostruisci(vant, vsim, infant+1, infant + (i -
infsim), i-1)
      r.right = ricostruisci(vant, vsim, infant + 1 + (i-infsim),
supant, i+1, supsim)
      return r
```

Complessità: O(n^2)

Alberi binari di ricerca

Composizione nodo:

key: valore di riferimento del nodo
right: nodo di destra (la sua chiave deve essere >= key)
left: nodo di sinistra (la sua chiave deve essere < key)
p: padre del nodo

Composizione albero:

root: radice dell'albero

x = Nodo di un albero binario di ricerca
k = elemento da ricercare nell'abr di x

```
tree_search(x, k) -> nodo
      if x==NULL or x.key==k
            return x
      else if k < x.key
            return tree_search(x.left, k)
      else
            return tree_search(x.right, k)
```

Complessità: O(h)

x = Nodo di un albero binario di ricerca
k = elemento da ricercare nell'abr di x

```
tree_search_iter(x, k) -> nodo
      while x!=NULL and x.key!=k
            if k<x.key
                  x=x.left
            else
                  x=x.right
      return x
```

Complessità: O(h)

x = Nodo di un albero binario di ricerca

```
tree_minium(x) -> nodo
      while x.left!=NULL
      x=x.left
      return x
```

Complessità: O(h)

x = Nodo di un albero binario di ricerca

```
tree_successor(x) -> nodo
      if x.right!=NULL
            return tree_minium(x.right)
      else
            y=x.p
      while y!=NULL and x==y.right
            x=y
            y=y.p
      return y
```

Complessità: O(h)

T = albero binario di ricerca
z = nodo da inserire in T

```
tree_insert(T, z)
      y = NULL
      x = T.root
      while x!=NULL
            y=x
            if z.key < x.key
                  x=x.left
            else
                  x=x.right
      z.p=y
      if y == NULL
            T.root=z
      else if z.key<y.key
            y.left=z
      else
            y.right=z
```

Complessità: O(h)

T = albero binario di ricerca
u = sotto-albero da sostituire
v = sotto-albero che sostituirà u

```
transplant(T, u, v)
      if u.p == NULL
            T.root = v
      else
            if u == u.p.left
                  u.p.left=v
            else
                  u.p.right=v
      if v!=NULL
            v.p=u.p
```

Complessità: O(1)

T = albero binario di ricerca
z = nodo da eliminare

```
tree_delete(t, z)
      if z.left == NULL
            transplant(T, z, z.right)
      else if z.right == NULL
            transplant(T, z, z.left)
      else
            y=tree_minium(z.right)
            if(y.p != z)
                  transplant(T, y, y.right)
                  y.right = z.right
                  z.right.p = y
            transplant(T, z, y)
            y.left = z.left
            z.left.p = y
```

Complessità: O(h)

Algoritmi di ordinamento

A = Array
k = elemento da inserire

```
insertionsort(A, k)
      for j=k to A.length
            i = j-1
            while i>0 and k<A[i]
                  A[i+1] = A[i]
                  i = i-1
      A[i+1] = k
```

Complessità: THETA(n^2)

A = Array
p = indice primo elemento array
r = indice ultimo elemento array

```
mergesort(A, p, r)
      if p<r
            q = parteIntera( (p+r)/2 )
            mergesort(A,p,q)
            mergesort(A,q+1,r)
            merge(A, p, q, r)
```

*Complessità: THETA(n*log(n))*

A = Array
p = indice inizio "primo" array
q = indice fine "primo" array
r = indice fine "secondo" array

```
merge(A, p, q, r)
      n1 = q-p+1
      n2 = r-q
      L=creaArray(n1+1)
      R=creaArray(n2+1)
      for i=1 to n1
            L[i] = A[p+i-1]
      for i=1 to n2
            R[i] = A[q+i]
      L[n1+1] = inf
      R[n2+1] = inf
      i = j = 1
      for k=p to r
            if L[i] <= R[j]
                  A[k] = L[i]
                  i = i+1
            else
                  A[k] = R[j]
                  j = j+1
```

Complessità: THETA(n)