▼ 5.0 - Strutture dati elementari

Le **strutture dati** descrivono come i dati sono logicamente organizzati e le operazioni per accedervi e modificarli. Nessuna struttura dati descrive quali dati è in grado di memorizzare, ma solo la maniera in cui tali dati vengono memorizzati, in quanto ogni struttura dati è in grado di memorizzare qualunque tipologia di dato.

All'interno di questa sezione verranno analizzate le seguenti quattro **tipologie** di strutture dati elementari:

- **Dizionario** (Dictionary)
- **Liste concatenate** (Linked list)
- **Pile** (Stack)
- **Code** (Queue)
- **Alberi** (Tree)

Prototipo vs implementazione

È importante distinguere il **prototipo** dall'**implementazione** di una struttura dati in quanto il primo descrive solo la struttura e le operazioni della struttura dati, permettendo al programmatore di implementarla e all'utente di capire come usarla. L'implementazione consiste invece nella realizzazione della struttura dati in un certo linguaggio di programmazione, ed è da quest'ultima che solitamente dipendono i tempi di esecuzione.

Classi di strutture dati

È possibile individuare diverse classi nelle quali suddividere le strutture dati in base alle loro proprietà:

- **Lineari**: i dati vengono memorizzati in ordine sequenziale (primo, secondo, terzo ecc.).
- **Non lineari**: non esiste un ordine sequenziale in cui i dati vengono memorizzati.
- **Statiche**: il numero degli elementi memorizzati al suo interno rimane costante.
- **Dinamiche**: il numero degli elementi memorizzati al suo interno può variare in maniera dinamica.
- **Omogenee**: un solo tipo di dato può essere memorizzato al suo interno.
- **Eterogenee**: differenti tipi di dato possono essere memorizzati al suo interno.

▼ 5.1 - Dizionario con array

> Un **dizionario** è una struttura dati dinamica che consente di memorizzare oggetti che presentano una chiave e un valore. Le chiavi all'interno di un dizionario devono essere univoche, mentre i valori possono anche essere duplicati.

Operazioni basilari di un dizionario:

- **search(Key k)**: ritorna l'oggetto associato alla chiave k, se la chiave non è contenuta ritorna null.
- **insert(Key k, Data d)**: se la chiave non è già contenuta aggiunge la coppia (k, d) al dizionario, mentre se è già contenuta sostituisce solo il vecchio dato con d.
- **delete(Key k)**: rimuove l'oggetto con chiave k dal dizionario.

Implementazione con array non ordinato

L'idea è quella di implementare il dizionario tramite un array che contiene le coppie (Key, Data) non tenendo conto dell'ordine delle chiavi.

Search

Viene svolta una ricerca della chiave tramite un algoritmo di ricerca lineare e viene ritornato l'oggetto corrispondente oppure null.

```
Data search(Dict D, Key k) {
  i = linsearch(D.A, D.size, k)

  if (i != 1) return D.A[i].data
  else return NIL
}

int linsearch(Array A[1, ... , m], int n, Key k) {
  for (i = 1, ... , n) {
    if (A[i].key == k) return i
  }
  return −1
}
```

Costo **ottimo** (chiave nell'ultima posizione dell'array o non trovata): $O(1)$.

Costo **medio** (costo medio della ricerca lineare): $\Theta(n)$.

Costo **pessimo** (chiave nella prima posizione dell'array): $\Theta(n)$.

Insert

Controlla tramite ricerca lineare se la chiave è già presente nell'array. In caso affermativo sostituisce i dati, altrimenti inserisce la coppia (Key, Data) nella prima posizione libera.

```
void insert(Dict D, Key k, Data d) {
  i = linsearch(D.A, D.size, k)
  if (i == −1) {
    D.size = D.size + 1
    i = D.size
  }
  D.A[i].key = k
  D.A[i].data = d
}
```

Costo **ottimo**: $O(1)$.

Costo **medio e pessimo**: $\Theta(n)$.

Delete

Cerca tramite ricerca lineare se la chiave è presente nell'array. In caso affermativo rimuove la coppia (Key, Data) e sposta di una posizione a sinistra tutte le coppie dopo Key.

```
void delete(Dict D, Key k) {
  i = linsearch(D.A, D.size, k)
  if (i != −1)
    leftshift(D.A, D.size, i)
  D.size = D.size − 1
}

void leftshift(Array A[1, · · · , m], int n, int i) {
  for (j = i, ... , n − 1)
    A[j] = A[j + 1]
}
```

Notiamo che il costo di linsearch + leftshift è equivalente a $\Theta(i) + \Theta(n-i) = \Theta(n)$, dunque il caso **ottimo, medio e pessimo** coincidono e sono uguali a: $\Theta(n)$.

Implementazione con array ordinato

L'idea è quella di implementare il dizionario sempre tramite un array ma di mantenere gli oggetti al suo interno ordinati in base al valore delle chiavi.

Tutti gli algoritmi per le operazioni search, insert e delete rimangono dunque uguali tranne per il fatto che viene utilizzata la ricerca binaria al posto di quella lineare e che nell'insert vengono spostati parte degli elementi a destra se il nuovo elemento viene inserito in posizione centrale.

A seguito di questa modifica i costi diventano dunque i seguenti:

- **search**

  Costo ottimo: $O(1)$.

  Costo medio e pessimo: $O(\log n)$.

- **insert**

  Costo **ottimo**: $O(\log n)$. Questo perchè la posizione ottimale di inserimento è quella in fondo a destra, in quanto poi non sarà necessario spostare nessun elemento a destra. Per trovare però l'ultima posizione a destra tramite ricerca binaria occorrono $\log n$ passi.

  Costo **medio**: $O(n)$. Assumiamo infatti che ogni posizione sia equiprobabile, notiamo che la ricerca binaria è asintoticamente logaritmica, mentre in media vengono shiftati $n/2$ elementi.

  Costo **pessimo**: $\Theta(n)$. Questo perchè il caso pessimo è quello in cui l'elemento deve essere inserito nella prima posizione dell'array. In questo caso la ricerca binaria svolge $\log n$ passi e lo shift n passi.

- **delete**

  Costo **ottimo**: $O(\log n)$.

  Costo **medio**: $O(n)$.

  Costo **pessimo**: $\Theta(n)$.

Notiamo dunque che in generali i costi delle operazioni elementari di un dizionario utilizzando un array ordinato sono migliorate. Tali costi sono riassunti nella seguente tabella:

	SEARCH		INSERT		DELETE	
	Medio	Pessimo	Medio	Pessimo	Medio	Pessimo
Array non ordinati	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$\Theta(n)$	$\Theta(n)$
Array ordinati	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Tabella dei costi asintotici delle operazioni elementari dei dizionari implementati tramite array ordinati e non.

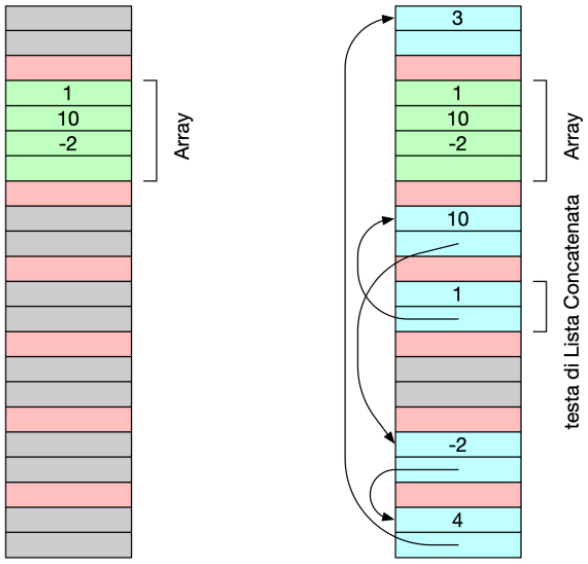
▼ 5.2 - Lista

Una **lista** è una struttura dati in cui tutti gli elementi sono organizzati in ordine sequenziale.

Operazioni basilari di una lista:

- **search(Key k)**: ritorna l'oggetto associato alla chiave k, se la chiave non è contenuta ritorna null.
- **insert(Key k, Data d)**: se la chiave non è già contenuta aggiunge la coppia (k, d) al dizionario, mentre se è già contenuta sostituisce solo il vecchio dato con d.
- **delete(Key k)**: rimuove l'oggetto con chiave k dal dizionario.

È possibile implementare una lista sia utilizzando **array** che **liste concatenate**. Nel primo caso si ha una dimensione allocata in maniera statica, ma l'accesso agli elementi della lista è più veloce in quanto avviene tramite l'utilizzo degli indici. Nel secondo caso invece la lista viene implementata tramite un insieme di puntatori, dunque la dimensione è dinamica e ne viene allocata di nuova su richiesta ma il costo dell'accesso agli elementi della lista dipende dalla loro posizione. Notiamo inoltre dalla seguente figura che per memorizzare una lista tramite un array occorre avere uno spazio libero contiguo in memoria, mentre ciò non è necessario se si utilizza una lista concatenata.



Visualizzazione grafica della memoria per una lista implementata tramite array e lista concatenata.

Lista concatenata semplice

Analizziamo ora il costo delle operazioni basilari di una lista implementata tramite l'utilizzo di una lista concatenata semplice.

Search

```
Node search(SLList L, Key k) {
  tmp = L.head
  while (tmp != null) {
  if (tmp.key == k)
      return tmp
    tmp = tmp.next
  }
  return null
}
```

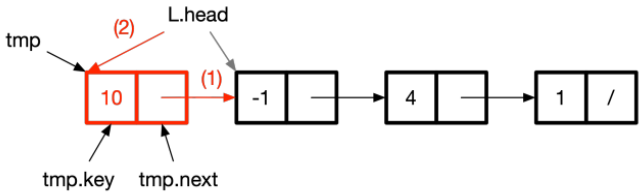
Costo **ottimo**: $O(1)$.

Costo **medio e pessimo**: $\Theta(n)$.

Insert

Possono essere utilizzati due metodi di inserimento in una lista concatenata, headInsert e tailInsert. Come suggeriscono i loro nomi il primo inserisce il nuovo elemento in testa alla lista, mentre il secondo in coda.

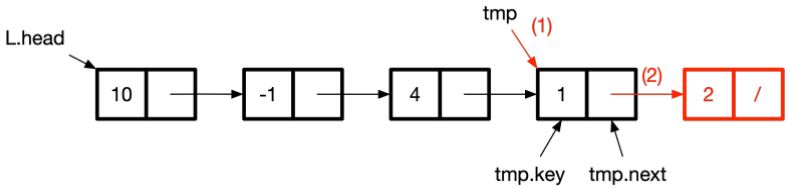
```
void headInsert(SLList L, Key k) {
  tmp = new Node(k)
  tmp.next = L.head
  L.head = tmp
}
```



Visualizzazione grafica di headInsert.

Costo **ottimo, medio e pessimo**: $O(1)$.

```
void tailInsert(SLList L, Key k) {
  if (L.head == null)
    L.head = new Node(k)
  else
    tmp = L.head
    while (tmp.next != null)
      tmp = tmp.next
    tmp.next = new Node(k)
}
```



Visualizzazione grafica di tailInsert.

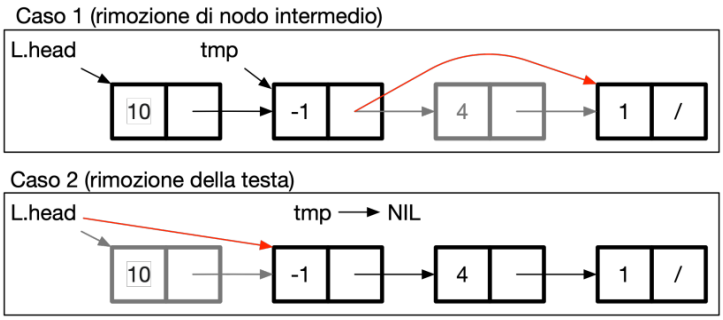
Costo **ottimo, medio e pessimo**: $\Theta(n)$.

Delete

```
void delete(SLList L, Key k) {
  tmp = searchPrev(L, k)
  if (tmp != null)
    tmp.next = tmp.next.next
  else if (L.head != null && L.head.key == k)
    L.head = L.head.next
}

void searchPrev(SLList L, Key k) {
  prev = null
  curr = L.head
  while (curr != null) {
    if (curr.key == key)
      return prev
    prev = curr
```

```
    curr = curr.next
  }
  return null
}
```



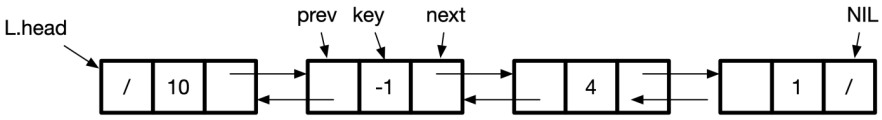
Visualizzazione grafica di delete.

Caso **ottimo**: $O(1)$.

Caso **medio e pessimo**: $\Theta(n)$.

Lista doppiamente concatenata

Una **lista doppiamente concatenata** è una lista concatenata semplice in cui ogni nodo contiene anche un puntatore al node precedente della lista. Il nodo in testa alla lista ha valore null nel puntatore al nodo precedente.

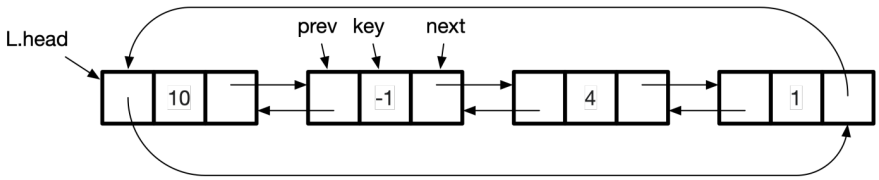


Esempio grafico di lista doppiamente concatenata.

Notiamo dunque che può essere visitata in entrambe le direzioni. I costi computazionali corrispondono a quelli di una lista concatenata semplice e l'unico pseudocodice che viene modificato è quella della funzione delete, la quale non necessita della ricerca del nodo precedente per via della presenza del puntatore ad esso.

Lista concatenata circolare

Una **lista concatenata circolare** è una lista doppiamente concatenata in cui il puntatore a next dell'ultimo nodo punta al primo nodo e il puntatore a prev del primo nodo punta all'ultimo nodo.



Esempio grafico di lista concatenata circolare.

Notiamo che anch'essa può essere visitata in entrambe le direzioni e l'accesso alla coda della lista è più veloce, ma diventa più complesso visitare la lista in quanto il nodo in coda non ha valore null

come puntatore al nodo successivo. I costi computazioni corrispondono dunque a quelli di una lista concatenata semplice tranne per la funzione di tailInsert, la quale assume un costo costante $O(1)$.

Lista con puntatori a testa e coda

Una lista con puntatori a testa e coda è una lista concatenata semplice o doppiamente concatenata in cui vengono mantenuti i puntatori alla testa e alla coda della lista.

In questo modo vengono velocizzate le operazioni di accesso alla testa della lista senza rendere più complessa la visita della lista, come avveniva per le liste concatenate circolari.

Riassunto dei costi

Type	SEARCH	INSERT (testa)	INSERT (coda)	DELETE
Concatenata semplice	$O(n)$	$O(1)$	$\Theta(n)$	$O(n)$
Doppiamente concatenata	$O(n)$	$O(1)$	$\Theta(n)$	$O(n)$
Circolare	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Puntatori testa e coda	$O(n)$	$O(1)$	$O(1)$	$O(n)$

Riassunto dei costi per le operazioni basilari delle diverse varianti di liste concatenate.

Dizionario con lista concatenata

È possibile implementare un dizionario utilizzando anche una lista concatenata, in modo da evitare la shift che domina i costi del dizionario implementato con array. Purtroppo però utilizzando le liste c'è necessità di svolgere la ricerca del nodo per ogni operazione, dunque i costi vengono dominati da quest'ultima e non si ottengono miglioramenti rispetto agli array ordinati.

	SEARCH		INSERT		DELETE	
	Medio	Pessimo	Medio	Pessimo	Medio	Pessimo
Array non ordinati	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$\Theta(n)$	$\Theta(n)$
Array ordinati	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Lista concatenata	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Riassunto dei costi per le operazioni basilari delle diverse implementazioni di un dizionario.

▼ 5.3 - Pila

Una **pila** è una struttura dati di tipo **LIFO (Last In First Out)** che supporta due operazioni basilari:

- **push**: aggiunge un nuovo elemento alla pila.
- **pop**: elimina l'elemento aggiunto più di recente e lo restituisce.

Una pila consente l'accesso solo all'ultimo elemento, e può avere applicazioni in diversi ambiti, come nella gestione dei record di attivazione, nei linguaggi stack oriented e nell'undo/redo degli editor di testo.

Implementazione di una pila

Una pila può essere implementata in diversi modi, ad esempio.

- **Lista concatenata**

Pro: dimensione illimitata.

Contro: overhead di memoria (occorre memorizzare anche i puntatori ai nodi della lista).

- **Array**

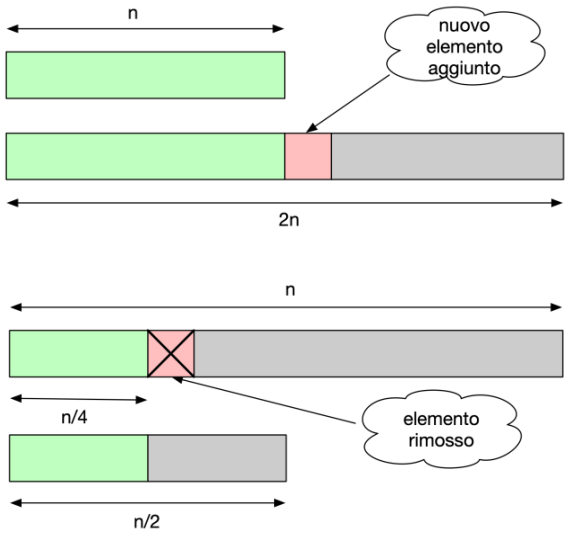
  Pro: nessun overhead di memoria (non occorre memorizzare anche i puntatori ai nodi della lista).

  Contro: dimensione limitata

In entrambe le implementazioni i **costi** delle funzioni push e pop rimangono $O(1)$.

Impemlentazione con array dinamico

È possibile implementare una pila tramite un array e mantenere il pro di avere una dimensione illimitata tramite l'utilizzo di un array dinamico, il quale adatta la sua dimensione al numero degli elementi contenuti in esso. In genere viene adottata la seguente strategia: viene raddoppiata la dimensione dell'array quando non c'è più spazio libero e viene dimezzata quando l'occupazione è di $1/4$.



Aggiunta e rimozione di un elemento in un array dinamico.

L'implementazione di un array dinamico prevede la copia di tutti gli elementi presenti all'interno di esso nel nuovo array quando questo viene creato per aumentare o diminuire la dimensione dell'array iniziale.

Lo pseudocodice per le operazioni di aggiunta e di rimozione di un elemento da un array dinamico è dunque il seguente:

```
void push(Stack S, Int x) {
  if (S.top == S.length) {
    // ridimensionamento dell'array
    n = S.length
    T = new array[1, ... , 2n]
    for (i = 1, ..., n)
      T[i] = S.stack[i]
    S.stack = T
    S.length = 2n
  }
  // push
  S.top = S.top + 1
  S.stack[S.top] = x
}
```

```
int pop(Stack S) {
  if (S.top == 0)
    error ”underflow”
  else
    // pop
    e = S.stack[S.top]
    S.top = S.top − 1
    if (S.top <= ⌊S.length / 4⌋) {
      // ridimensionamento dell'array
      n = S.length
      T = new array[1, ... , n / 2]
      for (i = 1, ..., n / 4)
        T[i] = S.stack[i]
      S.stack = T
      S.length = ⌈n / 2⌉
    }
    return e
}
```

Notiamo che il costo computazionale di entrambe le funzioni è equivalente e distinguiamo il **caso ottimo**, ovvero quando l'array non è ancora pieno per l'operazione di push e quando l'array non è utilizzato per più di $1/4$ nell'operazione di pop, in cui il costo è costante, ovvero $O(1)$, e il caso pessimo, ovvero l'opposto del caso precedente, nel quale i costi vengono dominati dall'operazione di copia di tutti gli elementi all'interno del nuovo array, dunque si ottiene un costo lineare, ovvero $O(n)$.

Possiamo inoltre approfondire l'analisi del costo di tali funzioni effettuando un **analisi ammortizzata** al fine di comprendere quanto sia il costo di n push/pop partendo da una pila vuota.

Calcoliamo dunque il costo di una sequenza di n push utilizzando il metodo dell'aggregazione e otteniamo che il costo ammortizzato è costante, ovvero $O(1)$.

È possibile effettuare la stessa analisi per l'operazione di pop, e utilizzando il metodo degli accantonamenti otteniamo anche qui che il costo ammortizzato di n operazioni di pop è costante, ovvero $O(1)$.

▼ **5.4 - Coda**

Una **coda** è una struttura dati di tipo **FIFO (First In First Out)** che supporta due operazioni basilari:

- **enqueue**: aggiunge un elemento in fondo alla coda.
- **dequeue**: rimuove l'elemento in testa alla coda.

Una coda non consente l'accesso agli elementi interni ad essa, e intuitivamente rappresenta una fila di elementi, ad esempio una file di persone in attesa di un servizio.

Implementazione di una coda

Una coda può essere implementata in diversi modi, ad esempio:

- **Lista concatenata circolare**

  Pro: dimensione illimitata.

  Contro: overhead di memoria (2 puntatori per ogni nodo).

- **Lista con puntatori a testa e coda**:

  Pro: dimensione illimitata.

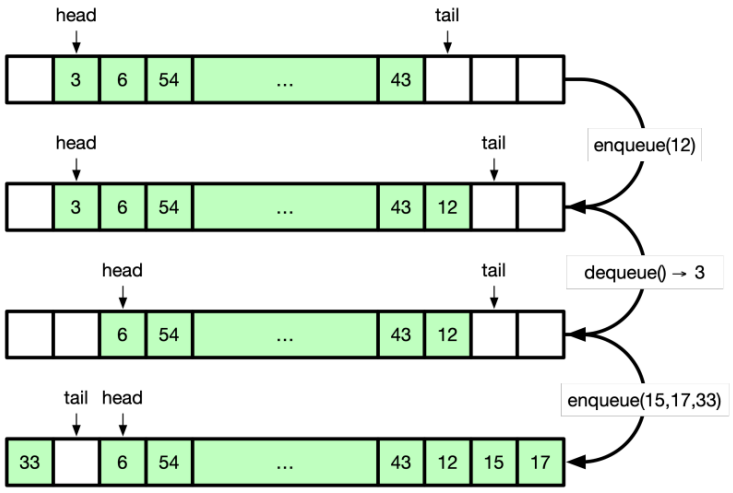
Contro: overhead di memoria.

- **Array circolari**

  Gli array circolari permettono, nel caso in cui la coda sia giunta all'ultima posizione dell'array, di sfruttare le posizioni iniziali dell'array nel caso in cui queste siano libere.

  Pro: nessun overhead di memoria.

  Contro: dimensione limitata.



Implementazione con array circolari

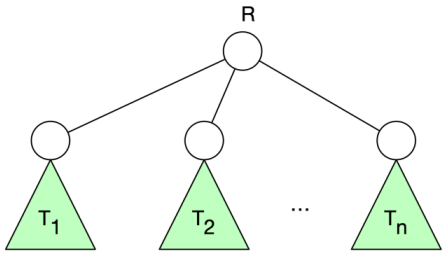
In tutti e 3 le implementazioni i **costi** delle operazioni di enqueue e dequeue rimangono $O(1)$.

▼ 5.5 - Albero

> Un **albero** è una struttura dati non lineare ad albero gerarchico. Esso contiene un insieme di nodi e un insieme di archi che connettono i nodi, ed esiste un solo percorso per andare da un nodo all'altro.

Un albero si dice ordinato se i figli di ogni nodo sono ordinati, ovvero se è possibile identificare un ordine di tale figli (es. primo figlio, secondo figlio ecc.)

Un albero si dice **radicato** se uno dei suoi nodi è identificato come radice. È possibile dare una definizione ricorsiva di albero radicato, dicendo che esso è un insieme vuoto di nodi oppure una radice R e zero o più alberi disgiunti le cui radici sono connesse ad R.



Esempio grafico di albero radicato.

Alcune definizioni

La **profondità** di un nodo è la lunghezza del percorso che va dalla radice a tale nodo.

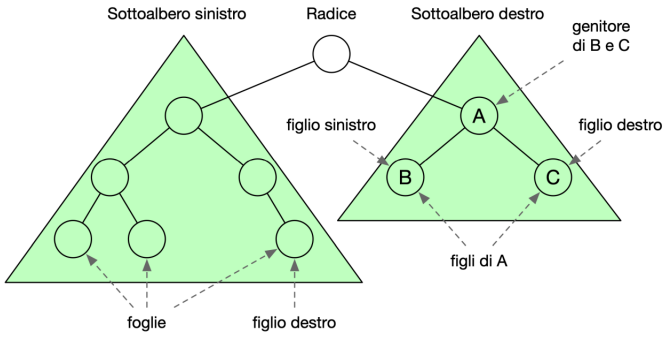
Un **livello** è l'insieme di tutti i nodi alla stessa profondità.

L'**altezza** di un albero è la sua massima profondità.

Il **grado** di un nodo è il numero dei suoi figli.

Albero binario

Un **albero binario** è un albero ordinato in cui ogni nodo ha al massimo due figli, e tali figli vengono identificati come destro e sinistro.



Esempio grafico di albero binario.

Un albero binario è **completo** se ogni nodo intermedio ha due figli.

Un albero binario è **perfetto** se è completo e tutte le foglie hanno la stessa profondità.

Proprietà fondamentale di un albero

Ogni albero non vuoto con n nodi ha esattamente $n - 1$ archi.

Algoritmi di visita su alberi

Un algoritmo di visita su albero consente di visitare tutti i nodi di una struttura dati albero.

Esistono principalmente due tipologie di visite su albero:

- **Visita in profondità (DFS - Depth First Search)**

  La ricerca va in profondità il più possibile prima di visitare il nodo successivo sullo stesso livello.

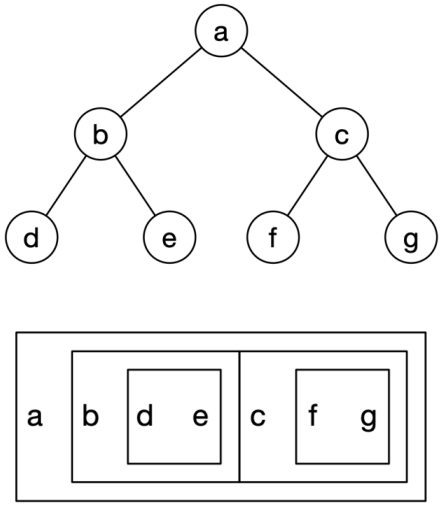
  Esistono tre tipologie di visite in profondità, ovvero pre-ordine, post-ordine e in-ordine.

- **Visita in ampiezza (BFS - Breadth First Search)**

  Vengono visitati tutti i nodi appartenenti ad un livello prima di passare a quello successivo.

Visita in profondità: pre-ordine.

```
void preorder(Node T) {
  if (T != null) {
    visit(T)
    preorder(T.left)
    preorder(T.right)
  }
}
```

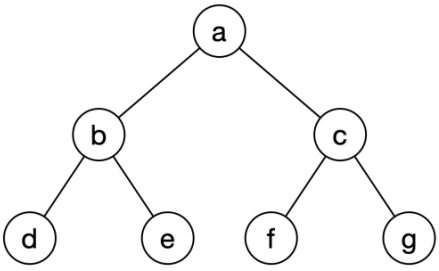


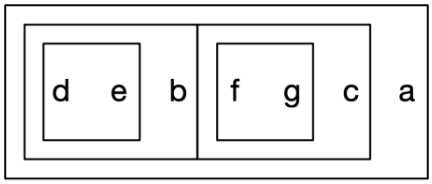
Esempio di visita pre-ordine.

Assumendo che visit abbia un costo costante, la funzione preorder ha un costo computazionale equivalente a $\Theta(n)$ (n = numero di nodi).

Visita in profondità: post-ordine

```
void postorder(Node T) {
  if (T != null) {
    postorder(T.left)
    postorder(T.right)
    visit(T)
  }
}
```

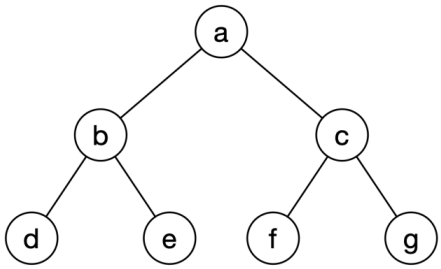


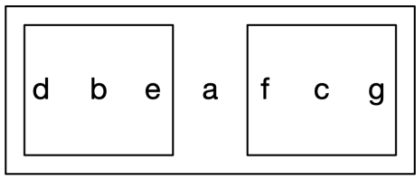


Esempio di visita post-ordine.

Visita in profondità: in-ordine

```
void inorder(Node T) {
  if (T != null) {
    postorder(T.left)
    visit(T)
    postorder(T.right)
  }
}
```





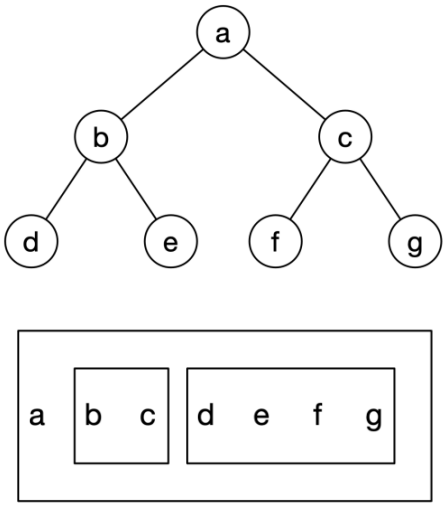
Esempio di visita in-ordine.

Assumendo che visit abbia un costo costante, la funzione inorder ha un costo computazionale equivalente a $\Theta(n)$ (n = numero di nodi).

Visita in ampiezza

È possibile utilizzare una coda per imporre un ordine di visita per livello.

```
void BFS(Tree T) {
  Q = new Queue
  if (T.root != null)
    enqueue(Q, T.root)
  while (Q.size != 0) {
    x = dequeue(Q)
    visit(x)
    if (x.left != null)
      enqueue(Q, x.left)
    if (x.right != null)
      enqueue(Q, x.right)
  }
}
```



Esempio di visita in ampiezza.

Assumendo che visit abbia un costo costante, la funzione BFS ha un costo computazionale equivalente a $\Theta(n)$ (n = numero di nodi).

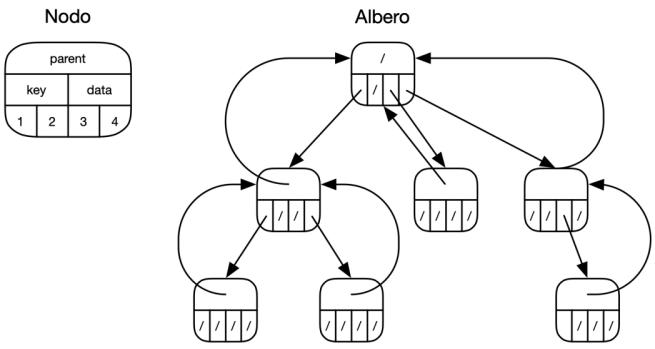
Visita su alberi non binari

Le tipologie di visite che abbiamo appena visto possono essere generalizzate ad alberi non binari. Solo per la visita in-ordine sono richieste specifiche aggiuntive, infatti occorre specificare su quanti nodi figli richiamare la funzione prima di visitare il nodo corrente.

Implementazione di un albero non binario

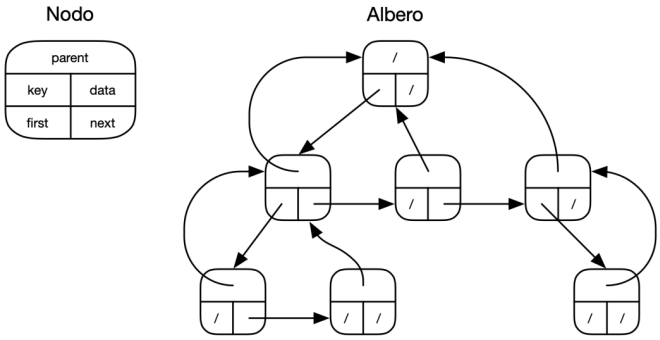
È possibile implementare un albero non binario in diversi modi.

Un esempio è quello di utilizzare per ogni nodo un array di puntatori a k figli. Il lato negativo però risiede nel fatto che si rischia di sprecare spazio se molti nodi hanno meno di k figli.



Implementazione con array di puntatori.

Per risolvere questo problema si può pensare di utilizzare per ogni nodo un puntatore al primo nodo figlio e al fratello successivo.



Implementazione con puntatore al fratello successivo.

Utilizzando queste implementazioni i costi delle funzioni presentate in precedenza rimangono equivalenti a $\Theta(n)$.