

INTRODUZIONE AGLI ALGORITMI

INFORMATICA 2024/2025

该笔记由九九精心整理

ORDINAMENTO 4

ORDINAMENTI NAIVE 4

Selection Sort (Ordinamento per selezione).....	4
Selection Sort: implementazione in Python	5
Selection Sort: complessità computazionale	5
Insertion Sort (Ordinamento per inserzione).....	6
Insertion Sort: implementazione in Python.....	7
Insertion Sort: complessità computazionale	7
Bubble Sort (Ordinamento a bolle).....	8
Bubble Sort: implementazione in Python	9
Bubble Sort: complessità computazionale	9
Bubble Sort Ottimizzato	10
Bubble Sort Ottimizzato: implementazione in Python.....	10
Bubble Sort Ottimizzato: complessità computazionale	10
Complessità dell'ordinamento.....	11
Albero di Decisione generico	11
Determinazione della limitazione inferiore	12

ORDINAMENTI LINEARI 13

Counting Sort.....	13
Counting Sort: implementazione in Python	14
Counting Sort: complessità computazionale.....	14
Bucket Sort	15
Bucket Sort: implementazione in Python.....	15
Bucket Sort: complessità computazionale	16

ORDINAMENTI DIVIDE ET IMPERA 17

Merge Sort (Ordinamento per fusione)	17
Merge Sort: implementazione in Python.....	18
Merge Sort: complessità computazionale	19
Quicksort (Ordinamento veloce)	21

Quicksort: implementazione in Python (versione non in loco).....	22
Quicksort: complessità computazionale (versione non in loco)	22
Quicksort: implementazione in Python (versione in loco).....	23
Quicksort: complessità computazionale (versione in loco)	24
Heapsort	25
Heapsort: implementazione in Python.....	25
Heapsort: complessità computazionale	25
STRUTTURA DATI	26
HEAP MINIMO	27
Heap: albero binario completo (o quasi completo)	27
Heap: mapping su array.....	29
Heap: implementazione e complessità di heappush(A, x) in Python	29
Heap: implementazione e complessità di heapify(A) in Python.....	30
Heap: implementazione e complessità di heappop(A) in Python	34
Heap massimo (adattato da heap minimo)	34
ARRAY	35
LISTA CONCATENATA	36
Lista concatenata: funzione Crea(A)	37
Lista concatenata: funzione Stampa(p)	38
Lista concatenata: funzione Ricerca(p, x)	38
Lista concatenata: funzioni di inserimento.....	39
Lista concatenata: funzioni di eliminazione.....	40
Liste in Python.....	41
Lista doppiamente concatenata.....	41
PILA (STACK)	43
Pila: implementazione con liste di Python	43
Pila: implementazione con liste concatenate	44
CODA (QUEUE)	45
Coda: implementazione con liste di Python	45
Coda: implementazione con liste concatenate	46
Coda: deque (Double–Ended Queue) dal modulo collections in Python.....	47
ALBERO	48

Alberi binari	49
Alberi binari: rappresentazione tramite array	50
Alberi binari: rappresentazione tramite vettore dei padri.....	51
Alberi binari: rappresentazione tramite puntatori.....	51
Alberi binari: implementazione in Python (rappresentazione tramite puntatori).....	52
Alberi binari: esempio di funzione stampa(p, h = 0)	53
Alberi binari: esempio di funzione generaAlbero(n, m)	54
Visite di alberi	54
Visite di alberi: costo computazionale	56
Applicazioni delle visite:.....	57
Applicazioni delle visite: contaNodi(p).....	57
Applicazioni delle visite: cerca(p, x).....	58
Applicazioni delle visite: altezza(p)	59
Applicazioni delle visite: contaLivello(p, k)	60
Applicazioni delle visite: stampaPerLivello(p)	61
Alberi Binari di Ricerca (ABR).....	63
Alberi Binari di Ricerca: ricerca.....	64
Alberi Binari di Ricerca: inserimento	64
Alberi Binari di Ricerca: cancellazione	65
Alberi Binari di Ricerca: stampaABR(p).....	67
Alberi Binari di Ricerca: minimoABR(p).....	67
Alberi Binari di Ricerca: generaABR(A)	68
DIZIONARIO	69

[Tutta la parte iniziale sul calcolo del costo computazionale da aggiungere (Lezioni 01 – 06)]

ORDINAMENTO

L'**ordinamento** consiste nel riorganizzare gli elementi di un **insieme** (numeri interi contenuti in un vettore) in un **ordine** specifico (crescente o decrescente).

In **contesti reali** i dati sono dei **record** che contengono informazioni non sempre omogenee relative allo stesso soggetto, quindi l'ordinamento avviene su un (ad esempio il codice fiscale).

ORDINAMENTI NAIVE

Selection Sort (Ordinamento per selezione)

Selection Sort **suddivide** l'array in **due parti**:

- **Array ordinato** (inizialmente vuoto);
- **Array non ordinato** (contiene tutti gli elementi dell'array iniziale).

Ordinato				Non ordinato			
1	3	4	6	9	10	7	8

L'algoritmo trova l'**elemento minimo** nell'array non ordinato e lo scambia con il **primo elemento** dell'array non ordinato per espandere l'array ordinato.

Ordinato				Non ordinato			
1	3	4	6	9	10	7	8

Min nell'array non ordinato è 7, quindi l'algoritmo lo scambia con il primo elemento dell'array non ordinato, che è 9

Ordinato					Non ordinato		
1	3	4	6	7	10	9	8

Ripete i passi fino a quando **tutti gli elementi sono ordinati**.

Ordinato					Non ordinato		
1	3	4	6	7	10	9	8

Ordinato							
1	3	4	6	7	8	9	10

Tutti gli elementi dell'array sono ordinati

Selection Sort: implementazione in Python

```
def SelectionSort(A):
    # numero di elementi di A (array)
    n = len(A)

    # ciclo esterno
    for i in range(n-1):
        # indice del minimo inizializzato al primo elemento
        indice_min = i

        # ciclo interno
        for j in range(i+1, n):
            # confronto tra elemento corrente e minimo
            if A[j] < A[indice_min]:
                # aggiorna indice del minimo
                indice_min = j

        # scambio tra primo elemento e minimo
        A[i], A[indice_min] = A[indice_min], A[i]
```

Selection Sort: complessità computazionale

La sua complessità computazionale dipende dai **due cicli for annidati**:

- **Ciclo esterno**: esegue **n-1** iterazioni;
- **Ciclo interno**: esegue **n-i-1** confronti per ogni iterazione esterna, con **i** come l'indice del ciclo esterno.

Quindi la sua **complessità totale** è data da:

$$T(n) = \sum_{i=0}^{n-2} (n - i - 1) \cdot \Theta(1) = \sum_{j=1}^{n-1} j \cdot \Theta(1) = \Theta(n^2)$$

(Sostituendo $j=n-i-1$, se $i=0$ allora $j=n-1$, se $i=n-2$ allora $j=n-(n-2)-1=1$)

In questo algoritmo non c'è distinzione tra caso migliore e caso peggiore.

L'algoritmo essendo **in-place**, non richiede ulteriore spazio per contenere gli elementi da ordinare.

Insertion Sort (Ordinamento per inserzione)

Insertion Sort **suddivide** l'array in **due parti**:

- **Parte ordinata** (inizialmente contenente solo il primo elemento);
- **Parte non ordinata** (contiene tutti gli altri elementi).

Ordinato				Non ordinato			
1	4	6	9	3	10	7	8

L'algoritmo prende il **primo elemento** della parte non ordinata e lo **confronta** con gli elementi della parte ordinata partendo da destra verso sinistra, **spostando** gli elementi **maggiori** verso destra finché non trova un elemento **minore o uguale**, per poi **inserirlo** nella posizione liberata.



Ordinato						Non ordinato	
1	4	5	6	9	10	7	8

L'algoritmo ripete questi passaggi fino a quando **tutti gli elementi sono ordinati**.

Insertion Sort: implementazione in Python

```
def InsertionSort(A):
    # numero di elementi di A (array)
    n = len(A)

    # ciclo esterno
    for i in range(1, n):
        # elemento corrente
        x = A[i]
        # indice del precedente
        j = i - 1

        # ciclo interno
        while j >= 0 and A[j] > x:
            # sposta l'elemento corrente a destra
            A[j + 1] = A[j]
            # decrementa l'indice del precedente
            j -= 1

        # inserimento dell'elemento corrente nella posizione corretta
        A[j + 1] = x
```

Insertion Sort: complessità computazionale

La sua complessità computazionale dipende dal **for** e dal **while** in esso **annidati**:

- **For esterno:** esegue **n-1** iterazioni;
- **While interno:**
 - **Caso migliore:** quando l'array è **già ordinato**, l'algoritmo non esegue spostamenti, quindi esegue $\Theta(1)$ iterazioni per ogni iterazione for. La sua **complessità totale** è data da:

$$T(n) = \sum_{i=1}^{n-1} \Theta(1) = (n-1) \cdot \Theta(1) = \Theta(n)$$

- **Caso peggiore:** quando l'array è in **ordine decrescente**, ogni elemento dovrà essere spostato verso sinistra attraversando l'intero array, quindi all'**i**-esima iterazione richiederà $\Theta(i)$ iterazioni. La sua **complessità totale** è data da:

$$T(n) = \sum_{i=1}^{n-1} \Theta(i) = \Theta(n^2)$$

- **Caso medio:** la complessità nel caso migliore e nel caso peggiore sono asintoticamente differenti, possiamo assumere che in **media** ogni elemento viene spostato a **metà** della distanza, quindi l'algoritmo esegue in media $\Theta\left(\frac{i}{2}\right)$ interazioni per ogni for.

$$T(n) = \sum_{i=1}^{n-1} \Theta\left(\frac{i}{2}\right) = \Theta(n^2)$$

L'algoritmo essendo **in-place**, non richiede ulteriore spazio per contenere gli elementi da ordinare.

Bubble Sort (Ordinamento a bolle)

Bubble Sort confronta ripetutamente coppie di elementi **adiacenti** in un array e li **scambia** se sono in ordine sbagliato, fino a quando l'intero array è ordinato.

L'algoritmo è formato da **due cicli for** annidati:

- **For esterno:** esegue **n-1** iterazioni, ad ogni iterazione **i** garantisce che l'**i**-esimo elemento più grande vada nella corretta posizione;
- **For interno:** confronta ogni coppia di elementi **adiacenti**, se sono nell'ordine sbagliato, li scambia, spostando l'elemento più grande **verso destra**.

Non ordinato				
6	2	7	1	3
(6 è maggiore di 2 quindi li scambia)				

Non ordinato				
2	6	7	1	3
(6 è minore di 7 quindi non li scambia)				

Non ordinato				
2	6	7	1	3
(7 è maggiore di 1 quindi li scambia)				

Non ordinato				
2	6	1	7	3
(7 è maggiore di 3 quindi li scambia)				

Non ordinato				
2	6	1	3	7
(Il numero 7 è ordinato)				

L'algoritmo ripete questi passaggi dall'inizio fino all'indice **n-i-1** (per escludere gli ultimi elementi già ordinati) fino a quando **tutti gli elementi sono ordinati**.

Bubble Sort: implementazione in Python

```
def BubbleSort(A):
    # numero di elementi di A (array)
    n = len(A)

    # ciclo esterno
    for i in range(n-1):

        # ciclo interno
        for j in range(n-i-1):

            # confronto tra elemento corrente e successivo
            if A[j] > A[j + 1]:
                # scambio tra elemento corrente e successivo
                A[j], A[j + 1] = A[j + 1], A[j]
```

Bubble Sort: complessità computazionale

La sua complessità computazionale dipende dai **due cicli for annidati**:

- **For esterno:** esegue **n-1** iterazioni;
- **For interno:** esegue **n-i-1** confronti per ogni iterazione esterna, con **i** come l'indice del ciclo esterno.

Quindi la sua **complessità totale** è data da:

$$T(n) = \sum_{i=0}^{n-2} (n - i - 1) \cdot \Theta(1) = \sum_{j=1}^{n-1} j \cdot \Theta(1) = \Theta(n^2)$$

(Sostituendo $j=n-i-1$, se $i=0$ allora $j=n-1$, se $i=n-2$ allora $j=n-(n-2)-1=1$)

In questo algoritmo non c'è distinzione tra caso migliore e caso peggiore.

L'algoritmo essendo **in-place**, non richiede ulteriore spazio per contenere gli elementi da ordinare.

Bubble Sort Ottimizzato

Se durante un passaggio **non avvengono scambi**, l'algoritmo termina anticipatamente perché l'array è già ordinato. Questa ottimizzazione utilizza un **flag** che rileva l'assenza di scambi che permette l'interruzione immediata in caso di array già ordinato.

Bubble Sort Ottimizzato: implementazione in Python

```
def BubbleSort2(A):
    # numero di elementi di A (array)
    n = len(A)

    # ciclo esterno
    for i in range(n-1):
        # variabile di controllo per verificare se ci sono stati scambi
        scambi = False

        # ciclo interno
        for j in range(n-i-1):

            # confronto tra elemento corrente e successivo
            if A[j] > A[j + 1]:
                # scambio tra elemento corrente e successivo
                A[j], A[j + 1] = A[j + 1], A[j]

            # aggiorna la variabile di controllo
            scambi = True

    # se non sono stati effettuati scambi, l'array è già ordinato
    if not scambi:
        # esce dal ciclo esterno
        return
```

Bubble Sort Ottimizzato: complessità computazionale

- **Caso migliore:** $\Theta(n)$, quando l'array è **già ordinato**, l'algoritmo **termina** dopo la prima iterazione del ciclo esterno;
- **Caso peggiore:** $\Theta(n^2)$, quando l'array è in **ordine decrescente**, l'algoritmo esegue tutte le **n-1** iterazioni del ciclo esterno:

$$T(n) = \sum_{i=0}^{n-2} (n - i - 1) \cdot \Theta(1) = \sum_{j=1}^{n-1} j \cdot \Theta(1) = \Theta(n^2)$$

(Sostituendo $j=n-i-1$, se $i=0$ allora $j=n-1$, se $i=n-2$ allora $j=n-(n-2)-1=1$)

- **Caso medio:** $\Theta(n^2)$, l'analisi del caso medio è piuttosto complessa, ma alla fine si conclude che anche al caso medio la complessità è $\Theta(n^2)$.

Complessità dell'ordinamento

Selection Sort, **Insertion Sort** e **Bubble Sort** hanno tutti un costo computazionale asintotico che cresce come il **quadrato** del numero di elementi da ordinare.

Esistono algoritmi più **efficienti** (es. **Merge Sort** e **Heapsort**) con complessità asintotica $\Theta(n \log n)$.

Per gli algoritmi di ordinamento basati su **confronti**, il limite **inferiore** teorico è $\Omega(n \log n)$, questo limite si dimostra con l'**Albero di Decisione**.

Albero di Decisione generico

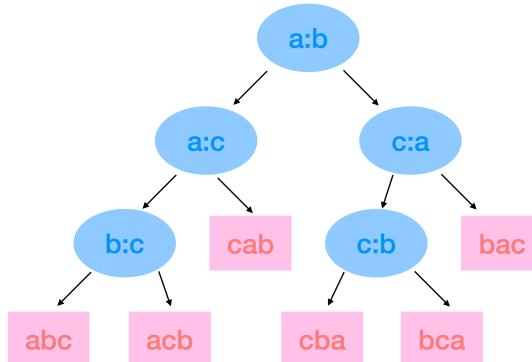
Un albero di decisione è un **albero binario** che modella tutte le possibili **esecuzioni** di un algoritmo basato su confronti (assumiamo che i numeri da ordinare siano tutti distinti).

Ogni **nodo interno** rappresenta un **confronto** tra due elementi e ha esattamente **due figli** che rappresentano i due possibili esiti del confronto.

Ogni **foglia** rappresenta una possibile **permutazione ordinata** dell'input.

Esempio:

Albero di decisione che rappresenta tutti i possibili confronti tra 3 elementi a, b, c durante l'esecuzione dell'algoritmo.



Ogni **nodo interno** corrisponde a un confronto (es. $a \leq b$), con due rami che rappresentano i due possibili esiti (sì o no).

Ogni **foglia** rappresenta una permutazione ordinata dell'input.

L'esecuzione corrisponde a un **cammino** dalla radice a una foglia, la **lunghezza** del cammino indica il numero di confronti necessari per ottenere l'ordinamento.

L'**altezza** dell'albero (cammino più lungo) rappresenta il numero **massimo** di confronti richiesti nel caso **peggiore**.

Determinazione della limitazione inferiore

Trovare un **limite inferiore** all'altezza dell'albero di decisione di un algoritmo di ordinamento basato su confronti **equivale** a stabilire un limite inferiore al suo tempo di esecuzione nel **caso peggiore**.

La soluzione può essere una qualunque delle **permutazioni** dell'input, ogni permutazione corrisponde a una **foglia** nell'albero di decisione, il numero di foglie sono $n!$ per un problema di dimensione n .

Un albero binario di **altezza** h ha al massimo 2^h foglie (tutte le foglie al livello più profondo).

Quindi si ha che l'altezza h dell'albero di decisione di qualunque algoritmo di ordinamento basato su confronti deve essere tale per cui $2^h \geq n! \iff h \geq \log_2(n!)$.

Valutiamo $\log_2(n!)$ (assumendo n pari)

$$\begin{aligned} \log_2(n!) &= \log_2(n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot \dots \cdot 2 \cdot 1) \geq \\ &\geq \log_2\left(n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot \dots \cdot \frac{n}{2}\right) \geq \quad (\text{con } \frac{n}{2} \text{ prodotti}) \\ &\geq \log_2\left(\frac{n}{2} \cdot \frac{n}{2} \cdot \frac{n}{2} \cdot \frac{n}{2} \cdot \dots \cdot \frac{n}{2}\right) \geq \quad (\text{con } \frac{n}{2} \text{ prodotti}) \\ &\geq \log_2\left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2} \log_2 \frac{n}{2} = \Omega(n \log n) \end{aligned}$$

Siccome abbiamo che $h \geq \log_2(n!)$ e che $\log_2(n!) = \Omega(n \log n)$ abbiamo $h = \Omega(n \log n)$

Quindi il **costo computazionale** di qualunque algoritmo di **ordinamento** basato su **confronti** è $\Omega(n \log n)$.

ORDINAMENTI LINEARI

Abbiamo dimostrato che gli algoritmi di ordinamento basati sui confronti hanno una complessità minima nel caso peggiore pari a $\Omega(n \log n)$.

Esistono alcuni algoritmi di ordinamento che in particolari condizioni raggiungono una complessità temporale lineare ($O(n)$), ad esempio **Counting Sort** e **Bucket Sort**.

Counting Sort

Il **Counting Sort** è un algoritmo non basato su confronti che richiede in input valori interi.

Il suo funzionamento si articola in **due fasi** principali:

- **Conteggio delle occorrenze:** si crea un array di **conteggio** per registrare il numero di occorrenze di ciascun valore presente nell'input;
- **Costruzione dell'array ordinato:** si **itera** sugli elementi dell'input e si **posizionano** nell'array di output in base alle **informazioni** memorizzate nell'array di conteggio.

Array input A: $n = 12$ (`len(A)`)

8	6	7	2	5	6	1	8	4	4	1	6
---	---	---	---	---	---	---	---	---	---	---	---

Array di conteggio C: $k = 8$ (`max(A)`) (`len(C)` è $k + 1$)

Index	0	1	2	3	4	5	6	7	8
Valori	0	0	0	0	0	0	0	0	0

(Index di C rappresentano i valori di A, valori di C rappresentano le occorrenze degli elementi di A)

Scorrendo **A** conteggiamo in **C** il numero di occorrenze di ciascun valore.

Array di conteggio C

Index	0	1	2	3	4	5	6	7	8
Valori	0	2	1	0	2	1	3	1	2

Scorrendo **C** ricopiamo in **A** ciascun **indice** di C tante volte quanto è il **valore** in C di quell'indice.

Array input A

1	1	2	4	4	5	6	6	6	7	8	8
---	---	---	---	---	---	---	---	---	---	---	---

Counting Sort: implementazione in Python

```
def CountingSort(A):
    # trova il valore massimo in A (array)
    k = max(A)
    # numero di elementi di A (array)
    n = len(A)

    # inizializza un array C di zeri con dimensione k+1 per il conteggio
    C = [0] * (k + 1)

    # ciclo per contare le occorrenze di ogni elemento in A
    for i in range(n):
        # incrementa il conteggio dell'elemento A[i]
        C[A[i]] += 1

    # inizializzazione di indice per l'array ordinato
    j = 0

    # ricostruzione dell'array ordinato A
    for i in range(k + 1):

        # per ogni elemento i in C, aggiunge i all'array ordinato A
        for _ in range(C[i]):
            # aggiunge l'elemento i all'array ordinato A
            A[j] = i
            # incrementa l'indice per l'array ordinato
            j += 1
```

Counting Sort: complessità computazionale

- Il calcolo di k costa $\Theta(n)$;
- L'inizializzazione dell'array di conteggio C costa $\Theta(k)$;
- Il primo for per il conteggio delle occorrenze costa $\Theta(n)$;
- I due for annidati per la ricostruzione dell'array ordinato costano $\Theta(n)$.

Quindi la sua **complessità totale** è $O(n + k)$ dove n è la dimensione dell'input e k è il valore massimo presente in input.

Quando k è **limitato**, l'algoritmo raggiunge una complessità lineare $O(n)$, quindi se k cresce **significativamente** rispetto a n , la complessità **non** sarà più **lineare**. Questo algoritmo è adatto quando il massimo valore dell'input è **limitato**.

Bucket Sort

Il **Bucket Sort** è un algoritmo di ordinamento che **suddivide** l'input in k **intervalli** (bucket) e li ordina separatamente.

L'intervallo dei valori $[0, M]$ viene diviso in k **sottointervalli** di uguale dimensione:

$$\text{Lunghezza bucket} = \frac{M + 1}{k}$$

Ogni elemento x viene assegnato al **bucket** i tramite la formula:

$$i = \left\lfloor \frac{x \cdot k}{M + 1} \right\rfloor \quad i = \text{intero compreso tra } 0 \text{ e } k - 1$$

Gli elementi dell'input vengono **distribuiti** nei bucket corrispondenti, ciascun bucket viene ordinato **individualmente** (es. con Insertion Sort) e infine i bucket ordinati vengono **concatenati** per ottenere l'array finale.

Array input A

8	9	7	5	6	4	3	2
---	---	---	---	---	---	---	---

Assumendo $k = 4$, in questo caso $M = 9$ quindi l'elemento x finisce nel bucket $\left\lfloor x \cdot \frac{4}{10} \right\rfloor$.

Array buckets

2	4, 3	7, 5, 6	8, 9
---	------	---------	------

Ogni bucket viene ordinato **separatamente**.

Array buckets

2	3, 4	5, 6, 7	8, 9
---	------	---------	------

Concatenazione dei bucket ordinati.

Array output B

2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---

Bucket Sort: implementazione in Python

```
def BucketSort(A, k):
    # inizializza k bucket vuoti
    buckets = [[] for _ in range(k)]
    # trova il valore massimo in A (array)
    M = max(A)

    # distribuisce gli elementi di A nei bucket
    for x in A:
        # calcola l'indice del bucket per l'elemento x
        i = x * k // (M+1)
```

```

# aggiunge l'elemento x al bucket corrispondente
buckets[i].append(x)

# ordina ogni bucket
for bucket in buckets:
    # ordina il bucket corrente
    bucket.sort()

# costruisce l'array ordinato B
B = []
# unisce i bucket ordinati in un array B
for bucket in buckets:
    # estende l'array B con gli elementi del bucket corrente
    B.extend(bucket)
# restituisce l'array ordinato
return B

```

Bucket Sort: complessità computazionale

- La creazione dei bucket costa $\Theta(k)$;
- Il calcolo del massimo M costa $\Theta(n)$;
- La distribuzione degli elementi dei bucket costa $\Theta(n)$;
- L'ordinamento degli elementi nei bucket costa:

$$\Theta\left(\sum_{i=0}^{k-1} (\text{costo per ordinare gli elementi del bucket } \text{buckets}[i])\right);$$

- Concatenazione degli elementi ordinati dei k buckets: $O(k + n)$.

Quindi la sua **complessità totale** è:

$$\Theta\left(\sum_{i=0}^{k-1} (\text{costo per ordinare gli elementi del bucket } \text{buckets}[i])\right)$$

Assumiamo di prendere un numero di buckets k **proporzionale** alla dimensione dell'input, quindi $k = \Theta(n)$:

- **Caso peggiore:** tutti gli elementi finiscono nello **stesso** bucket, in questo caso si ha una **complessità** $O(\text{costo per ordinare } n \text{ elementi})$ che dipenderà dall'algoritmo di ordinamento utilizzato;
- **Caso migliore:** gli elementi sono **equidistribuiti** nell'intervallo $[0, M]$, in questo caso in ogni bucket finiranno circa $\frac{n}{k} = \Theta(1)$ elementi, cioè un numero **costante** di elementi e quindi il costo di ordinamento di ciascun bucket sarà $\Theta(1)$ **indipendentemente** dall'algoritmo di ordinamento utilizzato, la sua **complessità** risulta $O(n)$.

Riassumendo, per garantire una complessità $\Theta(n)$ nel Bucket Sort è necessario assumere una distribuzione **uniforme** degli elementi tra i bucket e un numero di bucket $k = \Theta(n)$, così che ogni bucket contenga in media $O(1)$ elementi con costo di ordinamento **costante**.

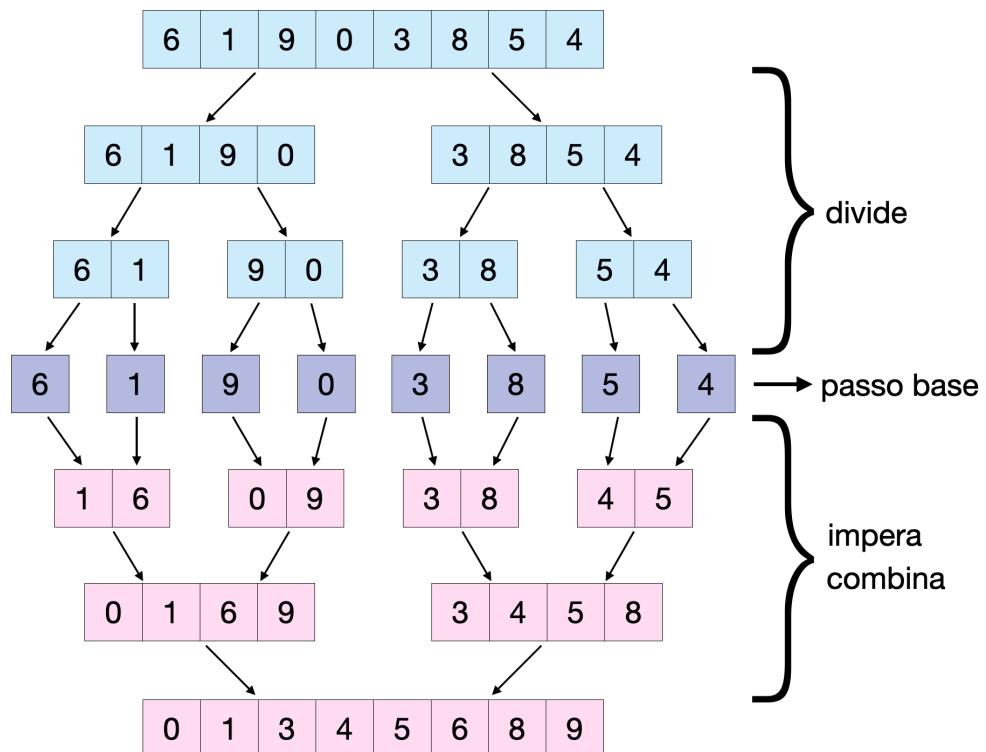
ORDINAMENTI DIVIDE ET IMPERA

Un ordinamento **divide et impera** è un algoritmo che risolve il problema di ordinamento suddividendolo in due **sottoproblemi** di dimensione inferiore (**divide**), li risolve **ricorsivamente** (**impera**), e infine ricompone le soluzioni parziali per ottenere l'array ordinato (**combina**).

Merge Sort (Ordinamento per fusione)

Il **Merge Sort** segue un approccio divide et impera articolato in 4 fasi:

- **Divide**: la sequenza di n elementi viene **divisa** in due sottosequenze di $\frac{n}{2}$ elementi ciascuna;
- **Impera**: le due sottosequenze vengono ordinate **ricorsivamente** usando lo stesso metodo;
- **Passo base**: la ricorsione termina quando una sottosequenza è composta da **un solo elemento** (per definizione è già ordinata perché non ci sono elementi con cui confrontarla);
- **Combina**: le due sottosequenze ordinate di $\frac{n}{2}$ elementi vengono **fuse** in un'unica sequenza ordinata di n elementi.



Merge Sort: implementazione in Python

```
def MergeSort(A, i, j):
    # se l'array ha più di un elemento
    if i < j:
        # calcola l'indice medio
        m = (i+j) // 2
        # ordina la prima metà dell'array
        MergeSort(A, i, m)
        # ordina la seconda metà dell'array
        MergeSort(A, m+1, j)
        # unisce le due metà ordinate
        fondi(A, i, m, j)

def fondi(A, i, m, j):
    # inizializza gli indici iniziali per le due metà
    a, b = i, m+1

    # crea un array temporaneo per memorizzare gli elementi ordinati
    B = []

    # finché ci sono elementi in entrambe le metà
    while a <= m and b <= j:

        # confronta gli elementi correnti delle due metà
        if A[a] <= A[b]:
            B.append(A[a]) # elemento corrente della prima metà -> B
            a += 1 # incrementa l'indice della prima metà
        # altrimenti
        else:
            B.append(A[b]) # elemento corrente della seconda metà -> B
            b += 1 # incrementa l'indice della seconda metà

    # aggiunge gli elementi rimanenti della prima metà
    while a <= m:
        B.append(A[a]) # elemento corrente della prima metà -> B
        a += 1 # incrementa l'indice della prima metà

    # aggiunge gli elementi rimanenti della seconda metà
    while b <= j:
        B.append(A[b]) # elemento corrente della seconda metà -> B
        b += 1 # incrementa l'indice della seconda metà

    # elementi ordinati dall'array temporaneo B -> array originale A
    for x in range(len(B)):
        A[i + x] = B[x] # elemento ordinato -> array originale A
```

Se si vuole ordinare l'**intero array**, la chiamata iniziale sarà: `MergeSort(A, 0, len(A) - 1)`

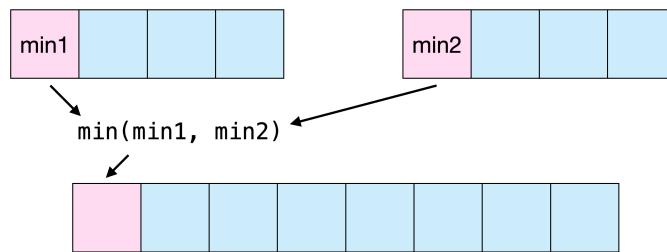
Merge Sort: complessità computazionale

Complessità di `MergeSort()`: indicando con $S(n)$ il costo della **fusion**e delle due sottoliste con dimensione complessiva n , si ha la seguente **equazione di ricorrenza**:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + S(n) & \text{altrimenti} \end{cases}$$

Funzionamento della funzione `fondi()`:

- Le due sottosequenze sono già **ordinate**;
- Il **minimo globale** della sequenza complessiva sarà sempre il minimo tra i **minimi attuali** delle due sottosequenze (se sono uguali, la scelta dei due non fa differenza);
- Dopo aver selezionato e rimosso il minimo corrente da una delle due sottosequenze, il nuovo minimo sarà **nuovamente** il minimo tra i nuovi minimi delle sottosequenze rimanenti.



Complessità di `fondi()`:

- Inizializzazione delle variabili costa $\Theta(1)$;
- **Primo ciclo while**: ogni iterazione costa $\Theta(1)$, quindi il costo è tra $\frac{n}{2}$ e n , ossia $\Theta(n)$;
- **Secondo e terzo while**: copiano la coda residua nel vettore B, costa $O(n)$;
- Copia del vettore B nell'opportuna porzione del vettore A, costa $\Theta(n)$.

Quindi la **complessità totale** di `fondi()`: $S(n) = \Theta(1) + \Theta(n) + O(n) + \Theta(n) = \Theta(n)$

Adesso possiamo calcolare la **complessità totale** di `MergeSort()`:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{altrimenti} \end{cases} = \Theta(n \log n)$$

Osservazione: il Merge Sort non può fondere le sottosequenze direttamente nell'array originale (**in loco**) in modo efficiente. Se provassimo a farlo, dovremmo spostare **manualmente** gli elementi per ogni inserimento, portando il costo della fusione da $\Theta(n)$ a $\Theta(n^2)$. Per questo si utilizza un **array ausiliario** per la fusione mantenendo l'efficienza ottimale.

Complessità spaziale di Merge Sort:

- Stack di chiamate ricorsive:

- Profondità massima è $\log_2 n$ (l'array viene diviso a metà ad ogni chiamata);
- Spazio per chiamata è $O(1)$ per variabili locali;
- Contributo complessivo delle variabili locali è quindi $O(\log n)$;

- Vettori temporanei:

- La **prima chiamata** crea due sottoliste di dimensione $\frac{n}{2}$ che occupano $O(n)$ in totale;
- La **seconda chiamata** divide una delle sottoliste in due nuove sottoliste di dimensione $\frac{n}{4}$ che occupano $\frac{1}{2}O(n)$ in totale;
- La **somma totale** della memoria usata risulta:

$$O(n) + \frac{1}{2}O(n) + \frac{1}{4}O(n) + \dots = O(n) \cdot \sum_{i=0}^{\log_2 n} \frac{1}{2^i} = O(n) \cdot O(1) = O(n)$$

Riassumendo il Merge Sort ha una **complessità temporale** $\Theta(n \log n)$ ma richiede **spazio aggiuntivo** $\Theta(n)$ per le operazioni di **fusione**, non essendo un algoritmo **in-place**.

Quicksort (Ordinamento veloce)

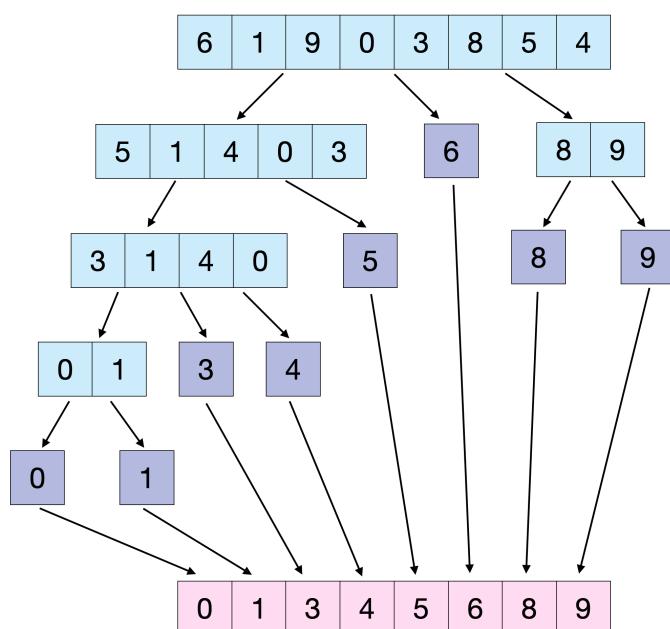
Il Quicksort presenta un **costo computazionale** di $O(n^2)$ nel caso **peggiore**, ma è spesso la scelta ottimale per grandi valori di n grazie a:

- Tempo di esecuzione **atteso** pari a $\Theta(n \log n)$;
- Fattori costanti nascosti molto piccoli;
- Capacità di ordinamento **in loco**.

Combina i **vantaggi** del Selection sort (in loco) e del Merge sort (ridotto tempo di esecuzione), ma ha lo svantaggio di un elevato costo nel caso peggiore.

Il Quicksort è un algoritmo di ordinamento basato sul paradigma **divide et impera**, funziona selezionando un elemento chiamato **pivot** e **partizionando** l'array attorno a esso:

- Un elemento **pivot** viene scelto dalla lista di n elementi (es. il primo elemento);
- Il pivot viene posizionato nella sua posizione **corretta**, dividendo l'array in due sottosequenze:
 - Elementi **minori** del pivot;
 - Elementi **maggiori o uguali** al pivot;
- L'algoritmo viene applicato **ricorsivamente** alle due sottosequenze ottenute.



Quicksort: implementazione in Python (versione non in loco)

```
def QuickSort(A):
    # se l'array ha più di un elemento
    if len(A) <= 1:
        # restituisce l'array originale
        return A

    # inizializza il pivot come il primo elemento dell'array
    pivot = A[0]
    # inizializza un array per gli elementi minori di pivot
    As = []
    # inizializza un array per gli elementi maggiori di pivot
    Ad = []

    # ciclo per separare gli elementi in base al pivot
    for i in range(1, len(A)):
        # se l'elemento corrente è minore di pivot
        if A[i] < pivot:
            # aggiunge l'elemento corrente all'array As
            As.append(A[i])
        # se l'elemento corrente è maggiore di pivot
        else:
            # aggiunge l'elemento corrente all'array Ad
            Ad.append(A[i])
    # restituisce l'array ordinato unendo le tre parti
    return QuickSort(As) + [pivot] + QuickSort(Ad)
```

Quicksort: complessità computazionale (versione non in loco)

Per quanto riguarda la **complessità temporale**, si ha la seguente **equazione di ricorrenza**:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ T(k) + T(n - 1 - k) + \Theta(n) & \text{altrimenti} \end{cases}$$

Dove k , con $0 \leq k \leq n - 1$, rappresenta la posizione corretta del **pivot** all'interno della lista ordinata. Si dimostra per **sostituzione** che: $\Omega(n \log n) \leq T(n) \leq O(n^2)$.

Quindi la complessità temporale si distinguono in **due casi**:

- **Caso migliore:**

- Il pivot divide l'array in due parti **bilanciate**, ciascuna di dimensione circa $\frac{n}{2}$;
- Costo della **partizione** è $\Theta(n)$, ogni elemento viene confrontato con il pivot;
- Costo della **ricomposizione** è $\Theta(n)$;
- La **complessità totale** risulta: $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) = \Theta(n \log n)$;

- **Caso peggiore:**

- Si verifica quando il pivot è sempre l'elemento **minimo** o **massimo** (es. array ordinato);
- Divisione **sbilanciata**: una sottolinea contiene **n-1** elementi, l'altra è vuota;
- La **complessità totale** risulta: $T(n) = T(n - 1) + \Theta(n) = \Theta(n^2)$.

Analisi della **complessità spaziale**: l'implementazione proposta del Quicksort presenta un problema principale, ogni operazione di partizionamento crea nuove liste (As e Ad), aumentando l'uso di memoria e il costo di copia.

Caso migliore:

- La **profondità della ricorsione** è $O(\log n)$ perché le sottoliste si dimezzano a ogni passo;
- Lo **spazio per livello i** è una lista di dimensione $\frac{n}{2^i}$;
- Lo **spazio totale** risulta: $\sum_{i=0}^{\log_2 n} \frac{n}{2^i} \cdot \Theta(1) = \Theta(n) \cdot \sum_{i=0}^{\log_2 n} \frac{1}{2^i} = \Theta(n) \cdot \Theta(1) = \Theta(n)$.

Caso peggiore:

- La **profondità della ricorsione** è $n - 1$ perché la lista si riduce di un elemento a ogni passo;
- Lo **spazio per livello i** è una lista di dimensione $n - i$;
- Lo **spazio totale** risulta: $\sum_{i=0}^{n-1} (n - i) \cdot \Theta(1) = \sum_{i=1}^n j \cdot \Theta(1) = \frac{n(n + 1)}{2} \cdot \Theta(1) = \Theta(n^2)$.

Quicksort: implementazione in Python (versione in loco)

A differenza del Merge Sort, il **Quicksort** può eseguire l'operazione di partizione “**in loco**”, evitando la creazione di nuove sottoliste e la successiva concatenazione durante la ricorsione.

```
def QuickSort2(A, primo, ultimo):
    # se l'array ha più di un elemento
    if primo < ultimo:
        # calcola l'indice del pivot
        pivot = Partition(A, primo, ultimo)
        # ordina la parte sinistra dell'array
        QuickSort2(A, primo, pivot - 1)
        # ordina la parte destra dell'array
        QuickSort2(A, pivot + 1, ultimo)

def Partition(A, primo, ultimo):
    # inizializza il pivot come il primo elemento dell'array
    pivot = A[primo]
    # inizializza gli indici i e j
    i, j = primo + 1, ultimo

    # ciclo per separare gli elementi in base al pivot
    while True:
        # trova il primo elemento maggiore di pivot
        while i <= ultimo and A[i] <= pivot:
            i += 1
        # trova il primo elemento minore di pivot
        while A[j] > pivot:
            j -= 1
        # se gli indici non si sono incrociati
        if i < j:
            # scambia gli elementi correnti
            A[i], A[j] = A[j], A[i]
```

```

# altrimenti
else:
    # scambia il pivot con l'elemento corrente
    A[primo], A[j] = A[j], A[primo]
    # restituisce l'indice del pivot
    return j

```

La funzione **Partition** seleziona un **pivot** nel sotto–array $A[\text{primo}, \text{ultimo}]$, riorganizza gli elementi con un ciclo **while** che **incrementa** i (per elementi minori) e **decrementa** j (per elementi maggiori/uguali), posizionando il pivot correttamente in $\Theta(n)$ operazioni senza spazio **ausiliario**, e restituisce la sua **posizione finale**.

Quicksort: complessità computazionale (versione in loco)

Per quanto riguarda la **complessità temporale**, si ha la seguente **equazione di ricorrenza**:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ T(k) + T(n - 1 - k) + \Theta(n) & \text{altrimenti} \end{cases}$$

Dove k , con $0 \leq k \leq n - 1$, rappresenta la posizione corretta del **pivot** all'interno della lista ordinata. Si dimostra per **sostituzione** che: $\Omega(n \log n) \leq T(n) \leq O(n^2)$.

Il Quicksort, pur avendo complessità temporali significativamente diverse tra caso **migliore** $\Omega(n \log n)$ e **peggiore** $O(n^2)$, dimostra una complessità **media** di $\Theta(n \log n)$, confermandosi così un algoritmo altamente efficiente nella maggior parte degli scenari pratici.

Per la dimostrazione del caso medio si veda il pdf del professore riguardante il Quicksort.

Heapsort

L'**Heapsort** è un algoritmo di ordinamento **complesso** ma **efficiente** che combina i vantaggi del Merge Sort e del Selection Sort:

- Ha un **costo computazionale** di $O(n \log n)$ anche nel caso peggiore, come il Merge Sort;
- Opera **in loco**, senza necessità di spazio aggiuntivo, analogamente al Selection Sort.

Per ottenere queste prestazioni, l'algoritmo utilizza una struttura dati appositamente organizzata (lo **heap**), il cui mantenimento e rispetto delle **proprietà** è fondamentale per il corretto funzionamento dell'Heapsort.

L'Heapsort opera secondo i seguenti passi:

- **Trasforma** l'array A in un heap minimo utilizzando `heapify(A)`;
- **Estrae** ripetutamente l'elemento minimo dall'heap con `heappop(A)` e lo accoda a un nuovo array B;
- **Restituisce** B, ottenendo così l'ordinamento crescente degli elementi.

Heapsort: implementazione in Python

```
def HeapSort(A):
    # importa le funzioni heapify e heappop dalla libreria heapq
    from heapq import heapify, heappop
    # crea un heap a partire dall'array A
    heapify(A)
    # inizializza un array vuoto B
    B = []
    # finché ci sono elementi nell'heap A
    while A:
        # estrae il minimo dall'heap A e lo aggiunge all'array B
        B.append(heappop(A))
    # restituisce l'array ordinato B
    return B
```

Per ulteriori informazioni riguardanti `heapify()` e `heappop()` si veda il paragrafo sulla **struttura dati heap**.

Heapsort: complessità computazionale

Complessità temporale: $\Theta(n \log n)$ sia nel caso **migliore** che **peggiore**;

Spazio aggiuntivo: $\Theta(n)$ dovuto all'uso dell'array B, ma è possibile implementarlo **in loco** con spazio $O(1)$.

STRUTTURA DATI

Una **struttura dati** è definita da:

- Un'**organizzazione sistematica** dei dati;
- Un'**insieme di operatori** per la **manipolazione** della struttura.

Le strutture dati si **classificano** in:

- **Omogenee o disomogenee** (basato sulla **tipologia** dei dati contenuti);
- **Statiche o dinamiche** (in base alla possibilità di **modificare** la dimensione nel tempo).

Un **insieme dinamico** è una struttura dati che può **modificare** la sua dimensione durante l'esecuzione del programma, adattandosi automaticamente agli elementi senza una dimensione fissa, come avviene nelle liste a puntatori.

Un **insieme statico** è una struttura dati **omogenea** a **dimensione fissa**, definita alla creazione e **immutabile** durante l'esecuzione, come gli array in C, ottimizzati per efficienza quando la dimensione è nota a priori.

La scelta tra insiemi statici e dinamici si basa sulle **esigenze** del programma: i primi sono più **efficienti** ma **rigidi**, i secondi **flessibili** ma con maggiori **costi** computazionali.

Gli elementi di un insieme dinamico (o **record**) possono essere strutturati in due modi:

- **Con chiave e dati satellite:**
 - La **chiave** (es. numeri interi) identifica **univocamente** l'elemento ed è usata per le operazioni sull'insieme;
 - I **dati satellite** (es. nome, data di nascita) sono informazioni **accessorie** non direttamente coinvolte nelle operazioni;
- **Solo chiave:** l'elemento coincide con la chiave stessa.

Nel seguito ci riferiremo sempre alla struttura **solo chiave**, a meno che non venga specificato esplicitamente il contrario.

Le operazioni su un insieme dinamico si dividono in:

- **Interrogazione** (**non modificano** l'insieme):
 - **Search(S, x)**: cerca l'elemento con chiave **x** in **S**;
 - **Min(S)**: restituisce il valore minimo in **S**;
 - **Max(S)**: restituisce il valore massimo in **S**;
- **Manipolazione** (**modificano** l'insieme):
 - **Insert(S, x)**: aggiunge l'elemento **x** a **S**;
 - **Delete(S, x)**: rimuove l'elemento **x** da **S**.

HEAP MINIMO

Per gestire una lista **A** di **n** elementi con operazioni frequenti di **estrazione del minimo** e **aggiunta di elementi**, esistono **tre approcci** con complessità diverse:

- **Lista non strutturata:**
 - Crea struttura: $O(1)$;
 - Estrazione del minimo: $\Theta(|A|)$ (scansione completa);
 - Aggiunta di un elemento: $O(1)$ (inserimento in coda);
- **Lista ordinata:**
 - Crea struttura: $\Theta(n \log n)$ (costo dell'ordinamento iniziale);
 - Estrazione del minimo: $O(1)$ (accesso all'ultimo elemento, se ordinamento decrescente);
 - Aggiunta di un elemento: $O(|A|)$ (inserimento ordinato);
- **Heap:**
 - Crea struttura: $\Theta(n)$ (con `heapify()`);
 - Estrazione del minimo: $O(\log |A|)$ (con `heappop()`);
 - Aggiunta di un elemento: $O(\log |A|)$ (con `heappush()`).

La **libreria standard heapq** di Python fornisce un'implementazione **efficiente** della struttura dati **heap minimo**, con le seguenti operazioni principali:

- `heapify(A)`: trasforma una lista generica in un heap minimo valido in tempo **lineare** $O(n)$;
- `heappop(A)`: estrae e restituisce l'elemento minimo dall'heap mantenendo la struttura in tempo **logaritmico** $O(\log |A|)$;
- `heappush(A, x)`: inserisce un elemento x nell'heap preservando la proprietà di heap minimo in tempo **logaritmico** $O(\log |A|)$.

Heap: albero binario completo (o quasi completo)

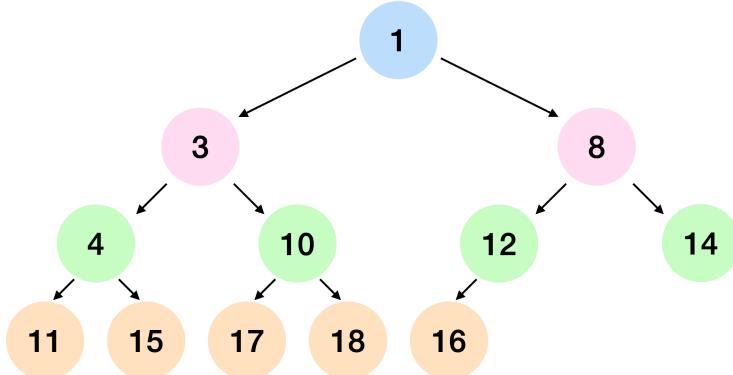
Esempio sull'utilizzo di `heapify(A)`:

```
import heapq
lista = [12, 3, 8, 4, 18, 16, 14, 11, 15, 17, 10, 1]
heapq.heapify(lista) # trasforma lista in un heap
print(lista)
```

Viene stampato: [1, 3, 8, 4, 10, 12, 14, 11, 15, 17, 18, 16]

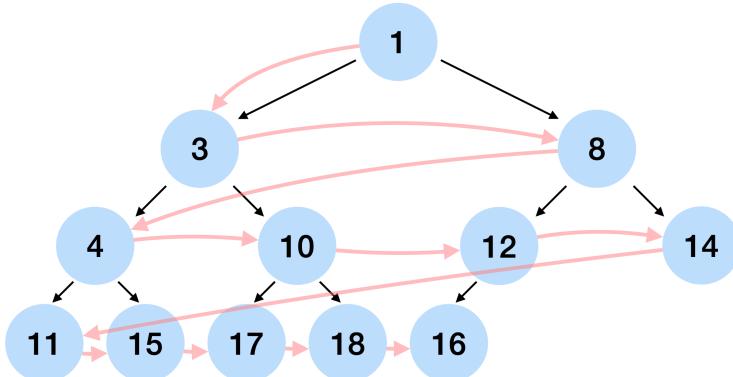
A prima vista l'ordinamento può sembrare strano, ma se visualizziamo la lista come un **albero binario completo**, scopriamo che ogni percorso dalla **radice** alle **foglie** mostra una sequenza **crescente** di valori.

1	3	8	4	10	12	14	11	15	17	18	16
---	---	---	---	----	----	----	----	----	----	----	----



Si noti che **scorrere** il vettore da sinistra a destra corrisponde a **muoversi** sull'albero per livelli, dall'alto verso il basso e da sinistra a destra in ciascun livello.

1	3	8	4	10	12	14	11	15	17	18	16
---	---	---	---	----	----	----	----	----	----	----	----



Uno **heap minimo** è un **albero binario completo** o quasi completo (con nodi addensati a sinistra nell'ultimo livello), dove la chiave di ogni nodo è **minore o uguale** a quella dei suoi figli (come esempio l'immagine precedente).

L'**altezza** h di uno heap è legata al **numero di nodi** n dalla relazione: $h = \Theta(\log n)$, possiamo **dimostrarlo** seguendo il ragionamento che segue.

Osservazione: Il numero di nodi che possiamo avere in un albero completo (o quasi completo) è la **somma** dei nodi su ciascun livello da 0 a $h - 1$ **più** i nodi sull'ultimo livello, che potrebbe non essere completamente pieno.

Il **numero di nodi** sui primi $h - 1$ livelli è $\sum_{i=0}^{h-1} 2^i = 2^h - 1$, per cui $n \geq 2^h \iff \log n \geq h$ e quindi:

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

Il **numero di nodi** sul livello h è al più 2^h , per cui $n \leq 2^h + (2^h - 1) < 2^{h+1}$, da questa disegualanza otteniamo che $\log n < h + 1$ e quindi:

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

Abbiamo dimostrato che $h = O(\log n)$ e $h = \Omega(\log n)$ possiamo **concludere** che $h = \Theta(\log n)$.

Heap: mapping su array

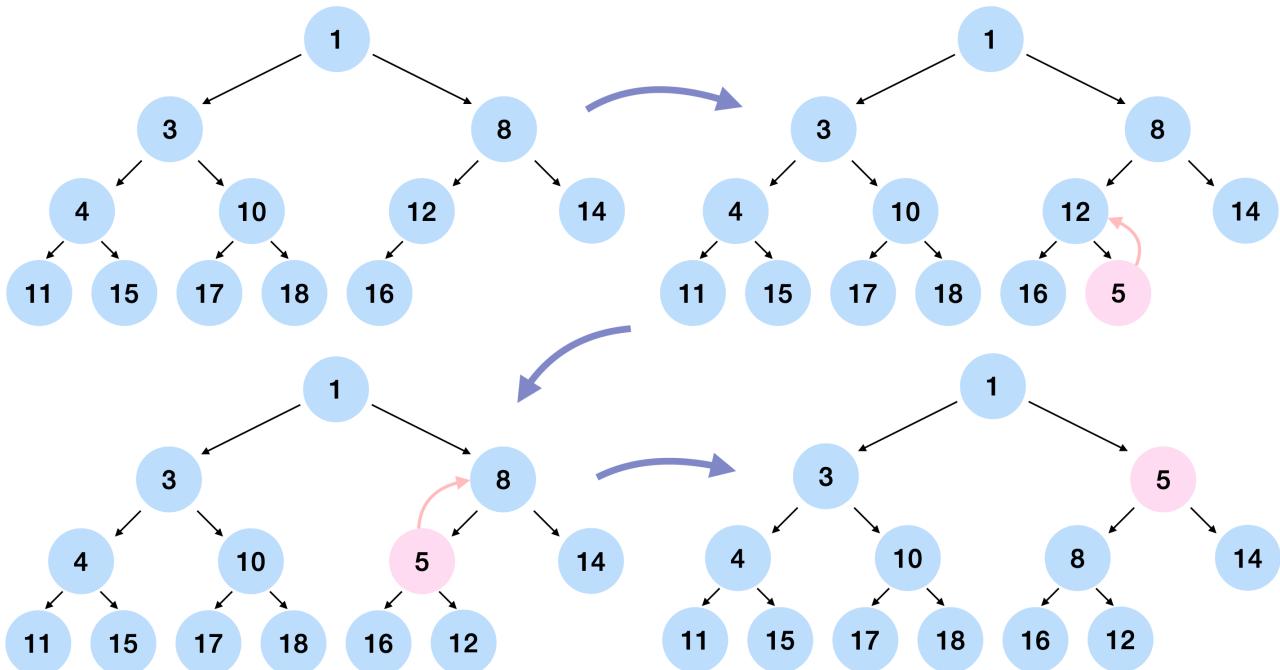
- Ogni **nodo** dell'heap corrisponde esattamente ad un elemento dell'array A;
- La **radice** dell'heap corrisponde ad $A[0]$ dove contiene l'elemento **minimo** dell'heap, con accesso in tempo **costante** $\Theta(1)$;
- Il **figlio sinistro** del nodo $A[i]$ corrisponde all'elemento $A[2*i+1]$ (se esiste), quindi $\text{left}(i) = 2 \cdot i + 1$;
- Il **figlio destro** del nodo $A[i]$ corrisponde all'elemento $A[2*i+2]$ (se esiste), quindi $\text{right}(i) = 2 \cdot i + 2$;
- Il **padre** del nodo $A[i]$ corrisponde all'elemento $A[(i-1)//2]$: $\text{parent}(i) = \left\lfloor \frac{i-1}{2} \right\rfloor$.

Heap: implementazione e complessità di heappush(A, x) in Python

Per **inserire** un elemento x in uno heap A mantenendo la proprietà di **heap minimo**:

- **Append**: aggiungi x in fondo all'array posizionandolo come **foglia** usando $A.append(x)$;
- **Risalita**: facciamo **risalire** x nell'heap fino a che non risulta **maggiore** del padre o raggiungere la radice.

Esempio di **heappush(A, 5)**:



```

def heappush(A, x):
    # aggiunge x in fondo all'array
    A.append(x)
    # indice di x (in fondo all'array)
    i = len(A) - 1

    # se x non è la radice e se x è minore del suo padre
    while i > 0 and A[i] < A[(i - 1) // 2]:
        # scambia x con il padre
        A[i], A[(i - 1) // 2] = A[(i - 1) // 2], A[i]
        # aggiorna l'indice di x con il padre
        i = (i - 1) // 2

```

La **complessità** dell'operazione `heappush(A, x)` è determinata dal numero di iterazioni del ciclo `while`, che nel caso **peggiore** è $O(\log n)$.

Il caso **peggiore** si verifica quando l'elemento inserito è il **nuovo minimo** e deve risalire fino alla radice. Poiché l'heap è un **albero binario** quasi completo, il numero **massimo** di operazioni è $\Theta(h)$, dove h è l'**altezza** dell'heap. Dato che l'heap contiene n elementi, l'altezza h è $\Theta(\log n)$.

Heap: implementazione e complessità di heapify(A) in Python

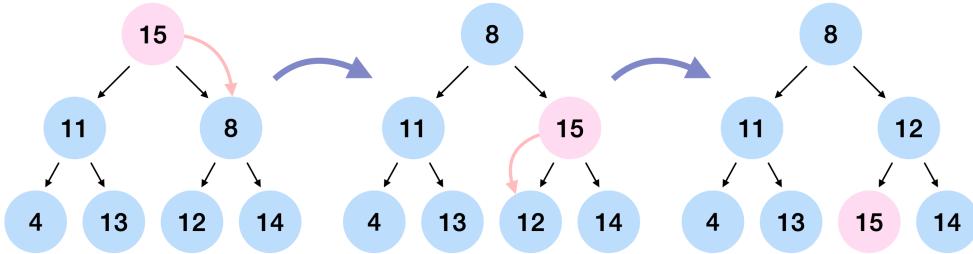
L'algoritmo di `heapify(A)` utilizza una funzione **ausiliaria** `heapify1(A, i)` per garantire il corretto funzionamento:

- La funzione `heapify1(A, i)` **ripristina** la proprietà di heap, assumendo che i sottoalberi sinistro e destro del nodo su cui viene invocata rispettino già tale proprietà, quindi l'**unico nodo** che potrebbe violarla è quello su cui viene chiamata la funzione, che potrebbe essere **minore** di uno o entrambi i figli.
- La funzione `heapify1(A, i)` **confronta** il nodo con i suoi figli e, se necessario, lo scambia con il minore dei due.
- Dopo lo scambio, `heapify1(A, i)` **verifica** se la violazione si è spostata sul figlio coinvolto nello scambio, ripetendo l'operazione su quel nodo se necessario.

Descrizione più tecnica di `heapify1(A, i)`:

1. **Inizializzazione:** parte da un nodo i che **potrebbe** violare la proprietà dell'heap;
2. **Verifica violazione:** **confronta** i con i suoi figli, se uno dei figli è più piccolo, la proprietà è violata;
3. **Scambio con il figlio minore:** se necessario, **scambia** i con il figlio più piccolo, dopo lo scambio, il nodo i **potrebbe** ancora violare la proprietà;
4. **Aggiornamento del nodo:** **sposta** i sul figlio con cui è stato scambiato, poiché ora potrebbe essere lui a violare la proprietà;
5. **Ripetizione:** **ripete** i passi 2–4 finché i non è minore dei suoi figli o diventa una foglia;
6. **Termine:** l'algoritmo si conclude quando i **rispetta** la proprietà dell'heap o non ha più figli.

Esempio di `heapify(A, 0)`:



```
def heapify1(A, i):
    # n è la lunghezza dell'array
    n = len(A)

    # while infinite loop
    while True:
        # inizializza m (indice minimo) come nodo i (nodo corrente)
        m = i
        # calcola gli indici dei figli sinistro e destro
        L = 2 * i + 1
        R = 2 * i + 2

        # se il figlio sinistro esiste ed è minore del nodo corrente
        if L < n and A[L] < A[m]:
            # aggiorna m con l'indice del figlio sinistro
            m = L
        # se il figlio destro esiste ed è minore del nodo corrente
        if R < n and A[R] < A[m]:
            # aggiorna m con l'indice del figlio destro
            m = R

        # se m è diverso da i
        if m != i:
            # scambia i nodi
            A[i], A[m] = A[m], A[i]
            # aggiorna i con m
            i = m
        # altrimenti esce dal ciclo
        else:
            break
```

La **complessità** di `heapify1(A, i)` è determinata dal numero di **scambi** necessari per **ripristinare** la proprietà di heap. Poiché ogni iterazione fa “**scendere**” il nodo corrente di un livello nell’albero, il costo dipende dall’**altezza** del sottoalbero radicato in quel nodo.

In **generale**, essendo l’altezza di un **heap bilanciato** pari a $\Theta(\log n)$ (dove n è il numero di nodi), il numero massimo di confronti e scambi è $O(\log n)$.

Nel caso **peggiore** si verifica quando il nodo di partenza è la **radice** ($i=0$) e contiene il valore **massimo**, costringendo a scendere fino a una **foglia**. In questo caso, l’algoritmo compie esattamente $\Theta(\log n)$ operazioni, pari all’altezza dell’albero.

Definiamo la **procedura** di **heapify(A)** che **trasforma** un vettore qualsiasi in un heap chiamando **ripetutamente** **heapify1(A, i)** sui nodi **appropriati**:

- Deve essere eseguita **dal basso verso l'alto** (da destra verso sinistra nel vettore), questo perché **heapify1(A, i)** richiede che entrambi i sottoalberi del nodo siano già heap **validi**;
- Le foglie sono già heap per definizione, si inizia dal nodo interno più a destra (padre dell'ultimo elemento) che ha come **indice** $\left\lfloor \frac{n-2}{2} \right\rfloor$, quindi l'**elemento A[(n-2) // 2]**.

```
def heapify(A):
    for i in range((len(A) - 2) // 2, -1, -1):
        heapify1(A, i)
```

Complessità di **heapify(A)**: la funzione effettua $\frac{n}{2}$ chiamate di **heapify1(A, i)**, ciascuna delle quali ha complessità $O(\log n)$, di conseguenza per la **complessità temporale** di **heapify(A)** abbiamo $T(n) = O(n \log n)$.

Con un'analisi più **accurata**, otteniamo che la complessità è in realtà $O(n)$:

- Consideriamo un **heap** di altezza h , nel livello i dell'heap ci sono al più 2^i nodi, di conseguenza il **numero di nodi** dell'heap che sono **radici** di un sottoalbero di altezza i sono $2^{h-i} = \frac{2^h}{2^i}$;
- Il **tempo richiesto** da **heapify1(A, i)** applicata ad un nodo che è radice di un sottoalbero di altezza i è $O(i)$.

Possiamo quindi scrivere:

$$T(n) \leq \sum_{i=1}^h 2^{h-i} O(i) = O\left(2^h \sum_{i=1}^h \frac{i}{2^i}\right) = O\left(2^h \sum_{i=1}^h \left(\frac{3}{4}\right)^i\right)$$

Dove nell'ultima diseguaglianza abbiamo sfruttato il fatto che $\frac{i}{2^i} < \left(\frac{3}{4}\right)^i$.

Essendo una **serie geometrica** di ragione minore di 1 quindi **converge**, possiamo concludere che:

$$T(n) = O(2^h) = O(n)$$

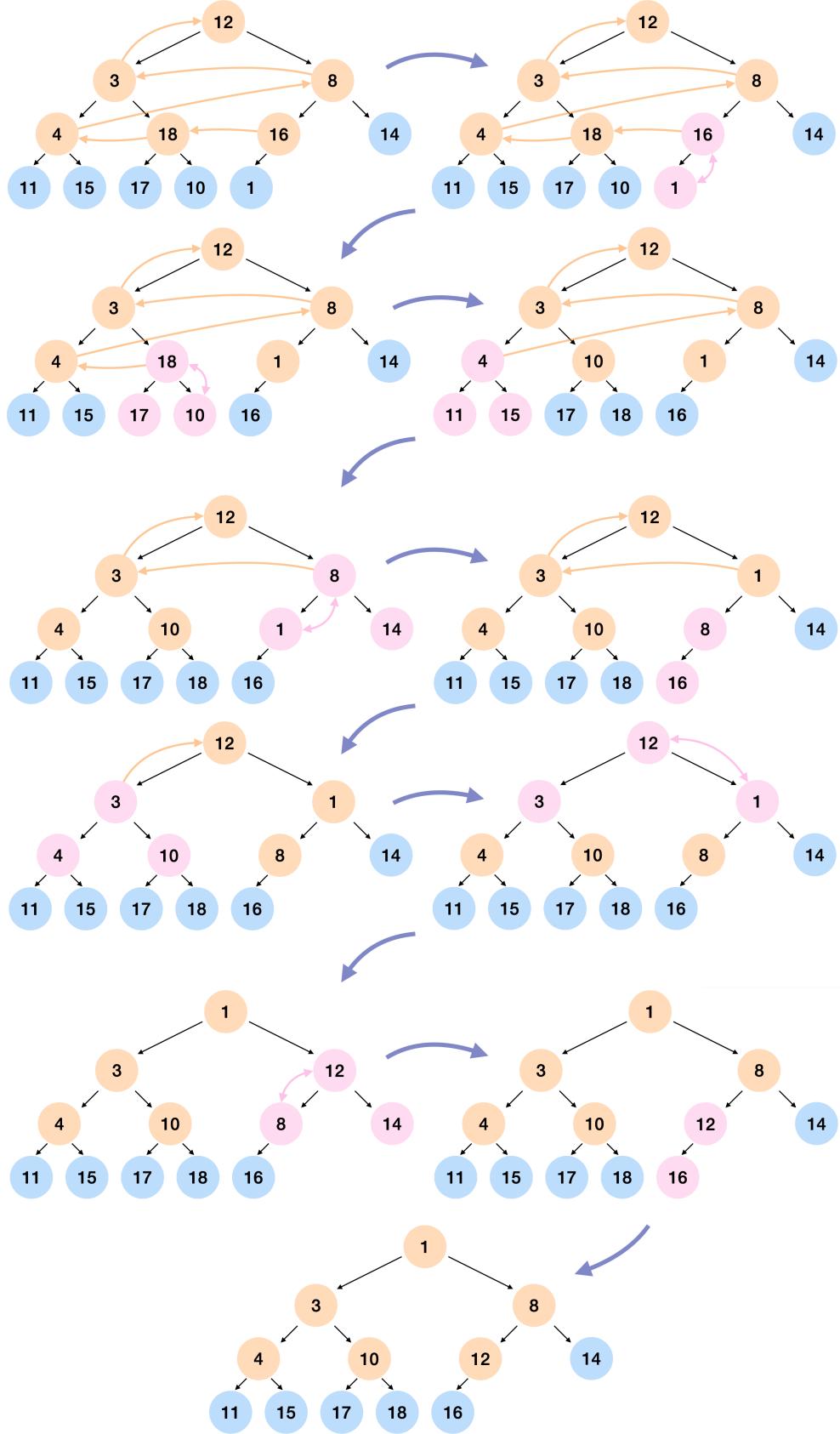
Esempio su **heapify(A)**:

Codice:

```
import heapq
lista = [12, 3, 8, 4, 18, 16, 14, 11, 15, 17, 10, 1]
heapq.heapify(lista) # trasforma lista in un heap
print(lista)
```

Viene stampato: [1, 3, 8, 4, 10, 12, 14, 11, 15, 17, 18, 16]

Immagine illustrativo su tutti i passaggi sostenuti da `heapify(A)` e `heapify1(A, i)`:



Heap: implementazione e complessità di heappop(A) in Python

La funzione `heappop(A)` estrae e restituisce il **minimo** elemento dall'heap `A` mantenendo la proprietà di heap, con **complessità** $O(\log n)$.

Passaggi dell'algoritmo `heappop(A)`:

- L'elemento **minimo** è sempre in `A[0]` (radice dell'heap), viene **memorizzato** in una variabile `x` per il ritorno;
- L'**ultimo** elemento dell'heap `A[len(A)-1]` viene **copiato** in `A[0]`, poi viene **rimosso** dall'array con `A.pop()`;
- Dopo la **sostituzione**, solo la **radice** potrebbe **violare** la proprietà di heap, viene **chiamata** `heapify1(A, 0)` per **riordinare** l'heap a partire dalla radice;
- Infine l'elemento `x` salvato inizialmente è **restituito** come risultato.

```
def heappop(A):  
    # salva il valore della radice  
    x = A[0]  
    # sposta l'ultimo elemento in cima all'heap  
    A[0] = A[len(A)-1]  
    # rimuove l'ultimo elemento  
    A.pop()  
    # ripristina la proprietà dell'heap  
    heapify1(A, 0)  
    # restituisce il valore della radice  
    return x
```

L'operazione `heapify1(A, 0)` costa $O(\log n)$ (altezza dell'heap), tutte le altre operazioni costano $O(1)$, quindi la **complessità totale** di `heappop(A)` è $O(\log n)$.

Heap massimo (adattato da heap minimo)

Possiamo adattare un **heap minimo** (che estrae sempre il valore più piccolo) per gestire un **heap massimo**, dove l'operazione principale è l'**estrazione del massimo** anziché del minimo.

Idea principale:

- Ogni elemento `x` viene **inserito** come `-x` nell'heap minimo;
- Quando l'elemento viene **estratto**, il suo segno viene **invertito** per ottenere il valore originale.

```
def heapifyMax(A):  
    for i in range(len(A)):  
        A[i] = -A[i]  
    heapify(A)  
  
def heappopMax(A):  
    return -heappop(A)  
  
def heappushMax(A, x):  
    heappush(A, -x)
```

ARRAY

Un **array** (o **vettore**) è una struttura dati che può essere utilizzata per memorizzare **insiemi dinamici**. Prima di esaminare strutture dati dinamiche più complesse, è utile analizzare il **costo computazionale** delle operazioni principali su array, sia **disordinati** che **ordinati**.

Gli array sono generalmente considerati strutture **statiche**, sebbene alcuni linguaggi di programmazione ne permettono la **ridimensionabilità** dinamica. Questa flessibilità è ottenuta **simulando** l'array con una struttura dati dinamica sottostante.

Ad esempio, in **Python**, la lista può essere vista come un array **dinamico**.

Complessità delle operazioni su array:

- **Search(S, x)**:
 - Array disordinato: **scansione** sequenziale → $O(n)$;
 - Array ordinato: **ricerca** binaria → $O(\log n)$;
- **Min(S), Max(S)**:
 - Array disordinato: **scansione** completa → $\Theta(n)$;
 - Array ordinato: **accesso** diretto al primo/ultimo elemento → $\Theta(1)$;
- **Insert(S, x)**:
 - Array disordinato: **inserimento** in prima posizione libera → $\Theta(1)$;
 - Array ordinato: **ricerca** posizione + **scorrimento** elementi maggiori → $O(n)$;
- **Delete(S, x)**:
 - Array disordinato: **eliminazione** con **scambio** con ultimo elemento → $\Theta(1)$;
 - Array ordinato: **eliminazione** + **scorrimento** elementi per riempire il vuoto → $O(n)$.

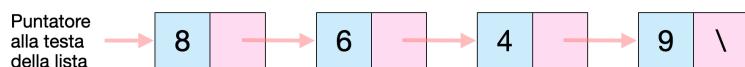
Non esiste una struttura dati **universalmente** migliore. L'adeguatezza dipende dall'**algoritmo** e dal **contesto**. Un buon progettista sceglie la struttura ottimizzata per le operazioni richieste.

LISTA CONCATENATA

La **lista concatenata** (lista semplice o lista puntata) è una struttura dati in cui gli elementi sono collegati in successione tramite **puntatori**.

Proprietà:

- **Accesso sequenziale** partendo da un puntatore alla testa della lista;
- Ogni elemento contiene un **valore** e un **puntatore** all'elemento successivo (o “\” (Null) se è l'ultimo);
- Non è possibile accedere **direttamente** a un elemento specifico (richiede **scorrimento**).



I **puntatori** sono variabili che memorizzano **indirizzi di memoria** (del primo byte del record puntato), il puntatore alla testa è l'**unico riferimento** per accedere alla lista.

Differenze fra vettori e lista:

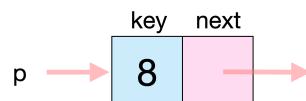
- **Vettori**: dimensione **fissa**, accesso **diretto** in $O(1)$ tramite indice, veloce per accesso **casuale**;
- **Liste**: dimensione **dinamica**, accesso **sequenziale** in $O(n)$ tramite puntatori, efficienti per **inserimenti/cancellazioni**.

Struttura:

- **key**: **valore** memorizzato;
- **next**: **puntatore** al nodo successivo (o None se è l'ultimo indicato con “\”).

Accesso:

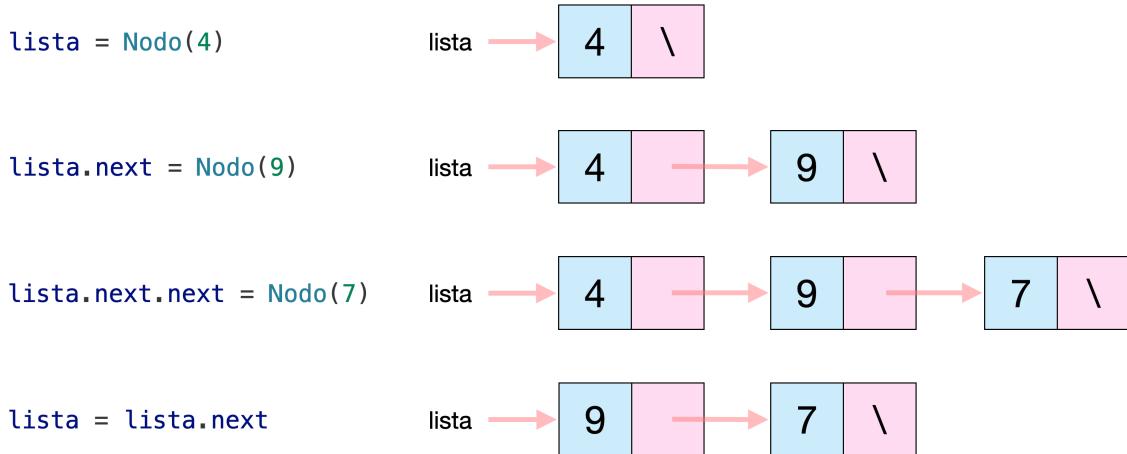
- **p.key**: **dato** del nodo puntato da p;
- **p.next**: **indirizzo** del nodo successivo (o None).



Definizione della classe **Nodo** in Python:

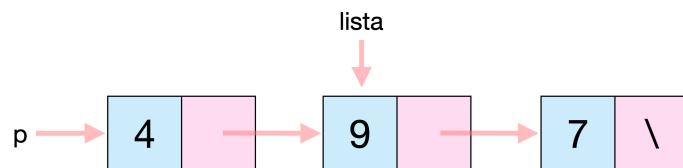
```
class Nodo:  
    def __init__(self, key = None, next = None):  
        self.key = key  
        self.next = next
```

Adesso possiamo **generare** nodi e **concatenarli** ad ottenere delle liste:



Se un nodo (es. valore 4) è **accessibile** solo tramite il puntatore `lista`, e `lista` viene **modificato** (es. `lista = lista.next`), il nodo 4 diventa **irraggiungibile**, la memoria viene automaticamente **liberata** per essere riutilizzata.

Se un'altro puntatore (es. `q`) fa ancora riferimento al nodo 4, esso non viene eliminato.



Lista concatenata: funzione Crea(A)

La funzione `Crea(A)` deve **generare** una lista semplice da un **array** dato, con un **nodo** per ogni elemento, e **restituire** la **testa** della lista.

Ad esempio il risultato di `Crea([8, 6, 4, 9])` è la creazione della seguente lista:



```

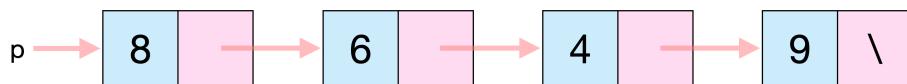
def Crea(A):
    if A == []:
        return None
    p = Nodo(A[0]) # p è la testa della lista
    q = p           # q è un puntatore alla lista
    for i in range(1, len(A)):
        q.next = Nodo(A[i])
        q = q.next
    return p
  
```

Il **costo computazionale** della creazione è $\Theta(n)$.

Lista concatenata: funzione Stampa(p)

La funzione `Stampa(p)` deve **visualizzare** i valori di tutti i **nodi** di una lista semplice partendo dal **puntatore p**.

Considerando la seguente lista:



Il risultato di `Stampa(p)` è la seguente:

```
8  
6  
4  
9
```

```
def Stampa(p):  
    while p:  
        print(p.key)  
        p = p.next
```

Il **costo computazionale** della stampa è $\Theta(n)$.

Lista concatenata: funzione Ricerca(p, x)

La funzione `Ricerca(p, x)` deve **trovare** e **restituire** il puntatore al primo nodo con valore **x** in una lista semplice partendo da **p**, o **None** se non trovato.

```
def Ricerca(p, x):  
    q = p  
    while q != None and q.key != x:  
        q = q.next  
    return q
```

I **puntatori** vengono passati come **argomenti** per valore, le **modifiche** al puntatore all'interno della funzione **non influenzano** il puntatore originale del chiamante perché si lavora su una **copia** locale, quindi il risultato sarebbe stato lo stesso con il seguente programma:

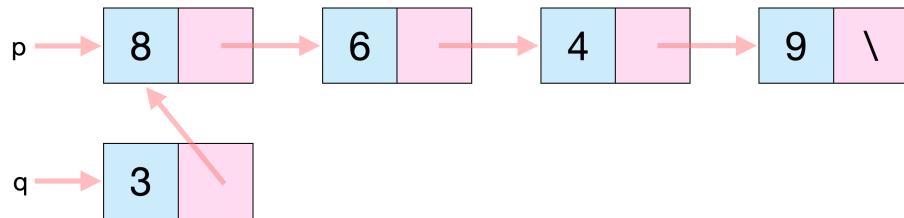
```
def Ricerca(p, x):  
    while p != None and p.key != x:  
        p = p.next  
    return p
```

Il **costo computazionale** della ricerca è $O(n)$.

Lista concatenata: funzioni di inserimento

La funzione `Inserisci(p, x)` aggiunge un nuovo nodo con valore `x` in testa alla lista, restituendo il puntatore alla `nuova testa`.

Ad esempio il risultato di `Inserisci(p, 3)` è la seguente lista con testa `q`:

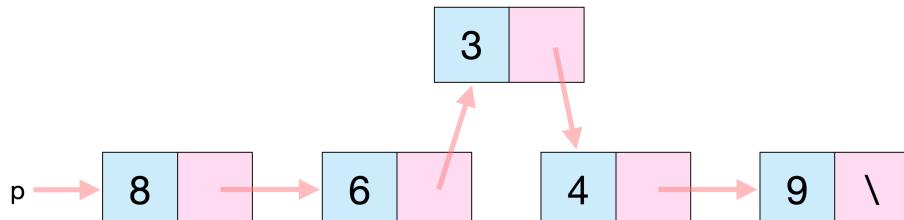


```
def Inserisci(p, x):
    q = Nodo(x, p)
    return q
```

Il **costo computazionale** di `Inserisci(p, x)` è $\Theta(1)$.

La funzione `Inserisci1(p, x, y)` aggiunge un nuovo nodo con valore `x` subito dopo la **prima** occorrenza di `y` nella lista.

Ad esempio il risultato di `Inserisci1(p, 3, 6)` è la seguente lista:



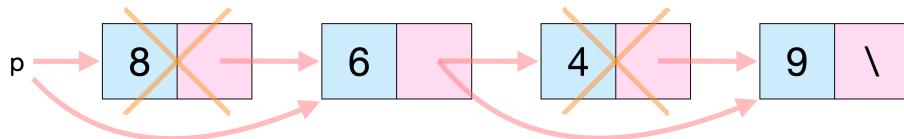
```
def Inserisci1(p, x, y):
    while p != None and p.key != y:
        p = p.next
    if p != None:
        p.next = Nodo(x, p.next)
```

Il **costo computazionale** di `Inserisci1(p, x, y)` è $O(n)$.

Lista concatenata: funzioni di eliminazione

La funzione `Cancella(p, x)` rimuove la prima occorrenza del valore `x` dalla lista semplice, restituendo la testa della lista (eventualmente modificata).

Illustrazione dell'effetto della cancellazione di 8 e 4 in una lista:



```
def Cancella(p, x):
    if p != None:
        if p.key == x: # se il nodo da cancellare è la testa
            p = p.next
        else:           # se il nodo da cancellare non è la testa
            q = p
            while q.next != None and q.next.key != x:
                q = q.next
            if q.next != None:
                q.next = q.next.next
    return p
```

Il **costo computazionale** della cancellazione è $O(n)$.

Eliminazione di nodi:

Quando si **crea** un nodo con `q = Nodo()`, esso viene **puntato** da `q`. Per **eliminarlo** e **liberare memoria**, basta rendere il nodo **inaccessibile** assegnando `q = None`.

Se il nodo non è più **referenziato** da alcun puntatore, il sistema lo **recupera** automaticamente per riutilizzarne lo spazio.

Tuttavia, se altri puntatori fanno ancora riferimento al nodo, la memoria non viene liberata perché il nodo rimane accessibile nonostante `q = None`.

Le liste sono strutture dati **inherentemente ricorsive**, tutti gli algoritmi proposti precedentemente possono essere implementati anche in versione ricorsiva.

Ad esempio un'implementazione **ricorsiva** di Cancellazione:

```
def CancellaR(p, x):
    if p == None:
        return p
    if p.key == x:
        return p.next
    p.next = CancellaR(p.next, x)
    return p
```

Liste in Python

Le **list** in **Python** combinano caratteristiche degli **array** (accesso per indice e lunghezza nota in tempo costante) e delle **liste concatenate** (dati eterogenei e lunghezza variabile).

In **C** l'accesso agli elementi di un array è $O(1)$ grazie alla memoria **contigua** (memorizzati in blocchi consecutivi) e agli elementi **omogenei**.

In **Python**, le liste sono implementate come **array di puntatori a oggetti** (eterogenei), ma l'accesso rimane $O(1)$ perché il calcolo della posizione del puntatore è **fisso** e **efficiente**, anche se richiede un ulteriore passaggio per raggiungere l'oggetto.

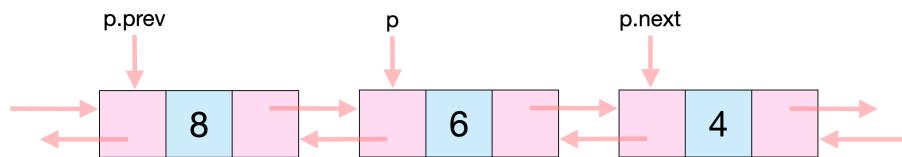
Operazioni sulle liste in Python e il loro **costo computazionale**:

- **A.append(x)**: inserisce **x** in coda alla lista $\rightarrow \Theta(1)$ (tempo costante);
- **A.insert(i, x)**: inserisce **x** in posizione **i**, spostando gli elementi successivi $\rightarrow O(n)$ (tempo lineare);
- **A.pop()**: rimuove l'ultimo elemento $\rightarrow \Theta(1)$ (tempo costante);
- **A.pop(i)**: rimuove l'elemento in posizione **i**, spostando gli elementi successivi $\rightarrow O(n)$ (tempo lineare);
- **A.extend(B)**: aggiunge gli elementi della lista **B** in coda ad **A** $\Theta(n_1)$ (dove n_1 è la lunghezza di **B**).

Lista doppiamente concatenata

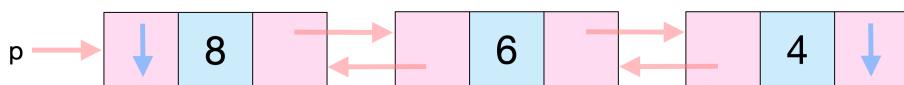
Una **lista doppiamente concatenata** (lista doppia o lista doppiamente puntata) permette di navigare sia in **avanti** (**next**) che all'**indietro** (**prev**) tra gli elementi, come illustrato dalla classe **NodoD** con campi **key**, **prev** e **next**:

```
class NodoD:  
    def __init__(self, key = None, prev = None, next = None):  
        self.key = key  
        self.next = next  
        self.prev = prev
```



La funzione **CreaDoppia(A)** crea una lista doppiamente concatenata a partire da un array **A**, restituendo il puntatore alla testa della lista.

Ad esempio il risultato di **CreaDoppia([8, 6, 4])** è la seguente lista:



```

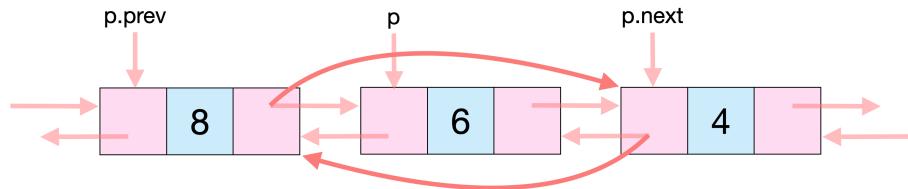
def CreaDoppia(A):
    if A == []:
        return None
    p = NodoD(A[0]) # p è la testa della lista
    q = p           # q è un puntatore alla lista
    for i in range(1, len(A)):
        q.next = NodoD(A[i], q) # q.next.prev = q
        q = q.next
    return p

```

Il **costo computazionale** di `CreaDoppia(A)` è $\Theta(n)$.

Per **rimuovere** un nodo puntato da `p` in una lista doppiamente concatenata, si **aggiornano** i puntatori dei nodi **adiacenti**:

- `p.prev.next = p.next`: il nodo **precedente** a `p` ora punta al **successivo** di `p`;
- `p.next.prev = p.prev`: il nodo **successivo** a `p` ora punta al **precedente** di `p`.



PILA (STACK)

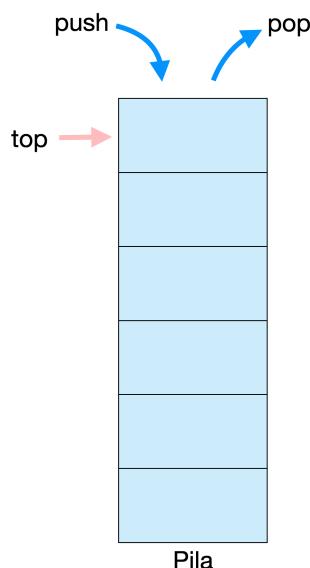
La **pila (stack)** è una struttura dati di tipo **LIFO** (**Last In, First Out**), dove l'**ultimo** elemento inserito è il **primo** a essere rimosso.

Si può paragonare a una pila di piatti: si aggiungono e si rimuovono elementi solo dalla **cima**.

Un **esempio** comune è la **pila di sistema**, che gestisce le chiamate a funzione, incluse quelle ricorsive.

La pila supporta solo **due operazioni fondamentali**:

- **Push (inserimento):** **aggiunge** un elemento in cima alla pila (top), e **modifica** il puntatore **top** per indicare il nuovo elemento;
- **Pop (estrazione):** **rimuove** l'elemento in cima alla pila (top), restituisce il valore rimosso e **aggiorna** top all'elemento sottostante.



Caratteristiche della pila:

- Non è possibile accedere, modificare o eliminare elementi **al di fuori della cima** (top);
- Tutte le operazioni avvengono tramite il **puntatore top**, garantendo il comportamento LIFO;
- La struttura è **ottimizzata** per velocità (operazioni in tempo costante $O(1)$).

Pila: implementazione con liste di Python

```
def push(Pila, x):
    Pila.append(x)

def pop(Pila):
    if Pila == []:
        return None
    return Pila.pop()
```

Il **costo computazionale** di entrambe le operazioni **push** e **pop** è $\Theta(1)$.

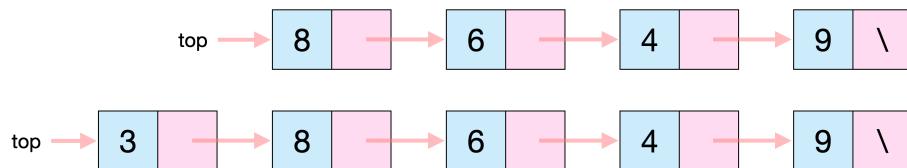
Se nell'implementazione decidessimo di estrarre e inserire **all'inizio** della lista, le due operazioni costerebbero $\Theta(n)$.

Pila: implementazione con liste concatenate

Inserimento: la funzione `push` inserisce un elemento x in cima a una pila implementata con una lista concatenata, aggiornando e restituendo il nuovo puntatore alla cima (top).

```
def push(top, x):
    return Nodo(x, top)
```

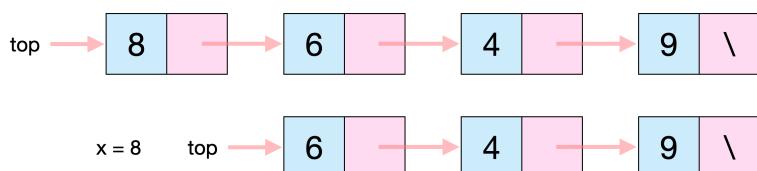
Ad esempio, il risultato dell'invocazione di `top = push(top, 3)`:



Estrazione: la funzione `pop` rimuove il nodo in cima a una pila implementata con lista concatenata, restituendo il suo valore e il nuovo puntatore alla cima (top).

```
def pop(top):
    if top == None:
        return None, None
    return top.key, top.next
```

Ad esempio, il risultato dell'invocazione di `x, top = pop(top)`:



Il **costo computazionale** di entrambe le operazioni `push` e `pop` è $\Theta(1)$.

CODA (QUEUE)

La **coda** (queue) è una struttura dati di tipo **FIFO** (First In, First Out), dove il **primo** elemento inserito è il **primo** a essere estratto.

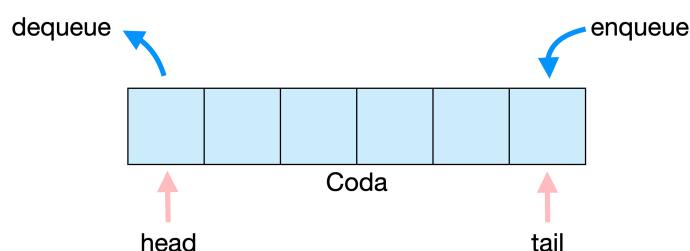
Si può paragonare a una fila alle poste: chi arriva prima viene servito per primo.

Un **esempio** d'uso è la **gestione della coda di stampa** (i documenti vengono stampati nell'ordine di arrivo).

La coda supporta **due operazioni fondamentali**:

- **Enqueue (inserimento)**: aggiunge un elemento alla coda (**tail**);
- **Dequeue (estrazione)**: rimuove l'elemento dalla testa (**head**).

Di solito non si **scandiscono** gli elementi di una coda e non si **eliminano** elementi con mezzi diversi da Dequeue.



Coda: implementazione con liste di Python

```
def enqueue(Coda, x):
    Coda.append(x)

def dequeue(Coda):
    if Coda == []:
        return None
    return Coda.pop(0)
```

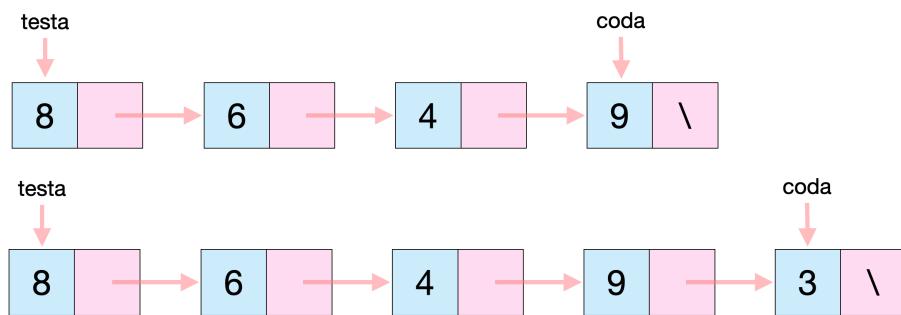
Con l'implementazione tramite lista, il **costo computazionale** dell'operazione di **inserimento** è $\Theta(1)$ mentre per l'**estrazione** si ha un costo $\Theta(n)$.

Coda: implementazione con liste concatenate

Inserimento (enqueue): la funzione **riceve** i puntatori alla testa (head) e coda (tail) della lista e il valore da inserire, **aggiunge** un nuovo nodo con il valore in coda, e **restituisce** i puntatori aggiornati a head e tail.

```
def enqueue(testa, coda, x):
    p = Nodo(x)
    if coda == None:
        return p, p
    coda.next = p
    return testa, p
```

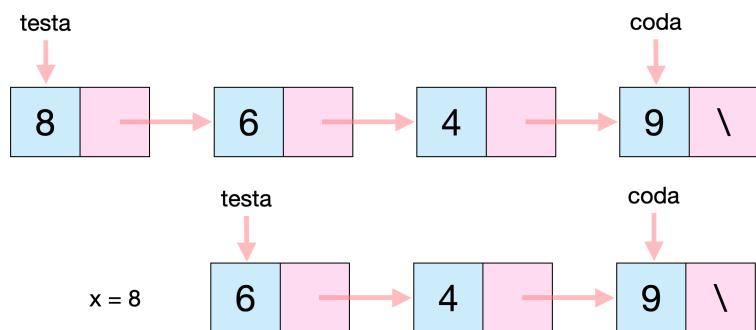
Ad esempio, il risultato dell'invocazione di `testa, coda = enqueue(testa, coda, 3)`:



Estrazione (dequeue): la funzione **riceve** i puntatori alla testa e coda, **rimuove** il nodo in testa e ne **restituisce** il valore, **restituisce** anche i puntatori aggiornati a head e tail.

```
def dequeue(testa, coda):
    if testa == None:
        return None, None, None
    if testa == coda:
        return testa.key, None, None
    return testa.key, testa.next, coda
```

Ad esempio, il risultato dell'invocazione di `x, testa, coda = dequeue(testa, coda)`:



Coda: deque (Double-Ended Queue) dal modulo collections in Python

La **cancellazione** da una coda implementata con array richiede richiede $\Theta(n)$ perché gli elementi successivi devono essere spostati **manualmente** (operazione costosa).

Possiamo utilizzare **deque** dal modulo **collections**:

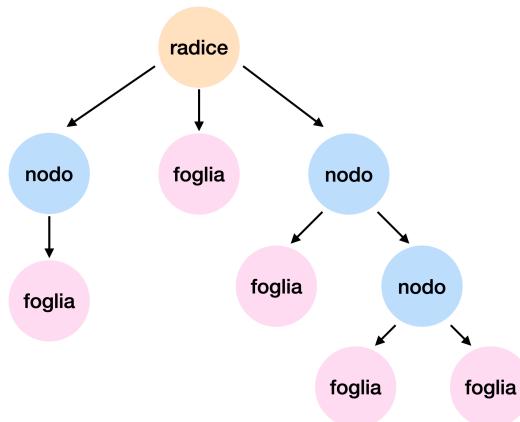
- **Efficienza:** inserimenti/cancellazioni in $O(1)$ da **entrambi i lati**;
- **Operazioni principali:**
 - `lista = deque([])`: **crea** una coda vuota;
 - `lista.appendleft(x)`: **inserisce** `x` in testa;
 - `lista.popleft()`: **rimuove** l'elemento in testa;
 - `lista.append(x)/lista.pop()`: operazioni **standard** in coda.
- **Limitazioni:** **cercare** l' i -esimo elemento costa $O(n)$ (deve scorrere la lista).

ALBERO

L'**albero** è una struttura dati versatile per **modellare** situazioni reali e **progettare** algoritmi. L'**albero binario** è un caso particolare già incontrato più volte, ma finora considerato solo in modo **intuitivo**.

L'**albero radicato** è una struttura **gerarchica** di nodi:

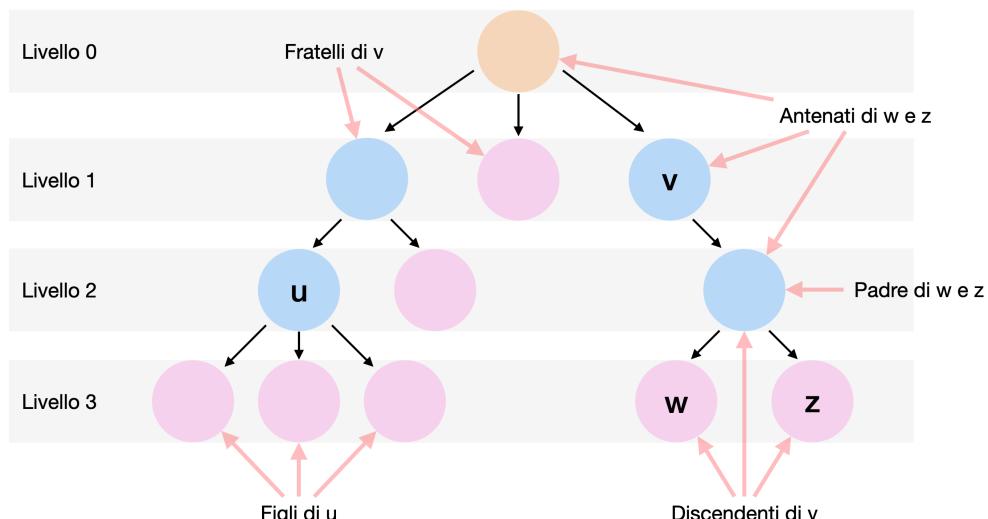
- **Radice**: nodo speciale **senza** genitori;
- **Nodi**: ogni nodo (tranne la radice) ha **un solo** genitore e può avere **zero o più** figli;
- **Foglia**: nodo senza figli, rappresenta un punto **terminale**.



Termini relativi a un **nodo x**:

- **Antenato**: ogni nodo sul **cammino** dalla radice a x;
- **Padre**: il nodo che **precede direttamente** x (tranne la radice che non ha padre);
- **Fratelli**: nodi con lo **stesso padre** di x;
- **Discendenti**: tutti i nodi per cui x è **antenato**.

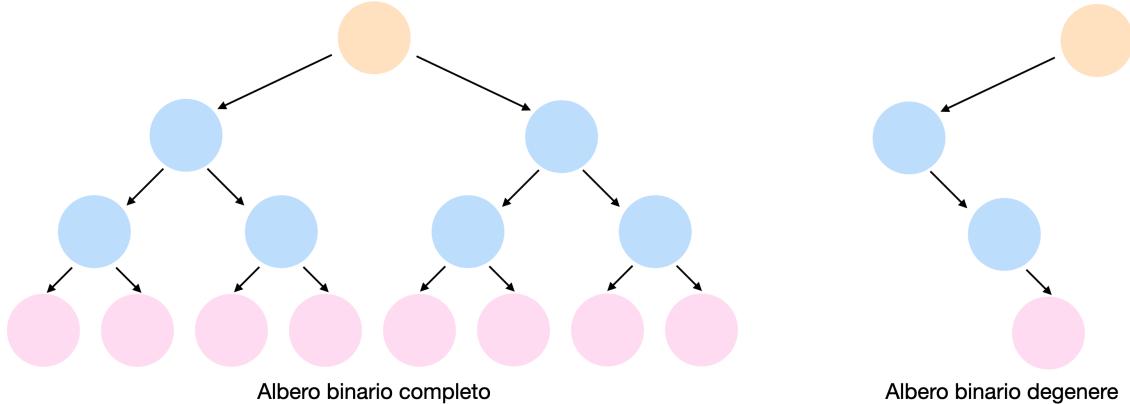
I nodi sono organizzati in **livelli** (con radice al livello 0), l'**altezza** è la lunghezza del cammino **più lungo** dalla radice a una foglia, quindi se l'albero ha altezza h , si ha $h + 1$ livelli, l'albero **vuoto** ha come altezza -1 per definizione.



Alberi binari

L'**albero binario** è un albero radicato in cui ogni nodo ha al **massimo due figli** (sinistro e destro), ci concentreremo esclusivamente sugli alberi binari.

- **Albero binario completo:** tutti i livelli contengono il **massimo** numero possibile di **nodi**;
- **Albero binario degenere:** ogni livello contiene **solo un nodo** (forma una catena).



L'albero binario **completo** ha il **massimo** numero di nodi per altezza h , in questo caso il numero di nodi e l'altezza dell'albero sono dati da:

$$n = \sum_{i=0}^h 2^i = 2^{h+1} - 1 \quad h = \log_2 \left(\frac{n+1}{2} \right)$$

L'albero binario **degenero** ha il **minimo** numero di nodi per altezza h , in questo caso l'altezza coincide sostanzialmente con il numero di nodi:

$$h = n - 1$$

Combinando i due risultati otteniamo la **relazione generale** per alberi binari:

$$\log_2 \left(\frac{n+1}{2} \right) \leq h \leq n - 1 \quad \Omega(\log n) \leq h \leq O(n)$$

L'altezza di un albero (h) rappresenta il **massimo** numero di passi per raggiungere qualsiasi nodo partendo dalla radice, una **buona proprietà** dell'albero binario è avere $h = \Theta(\log n)$.

- **Albero binario bilanciato:** albero con altezza $h = \Theta(\log n)$, che garantisce **ottime** prestazioni.

Esempi di alberi bilanciati:

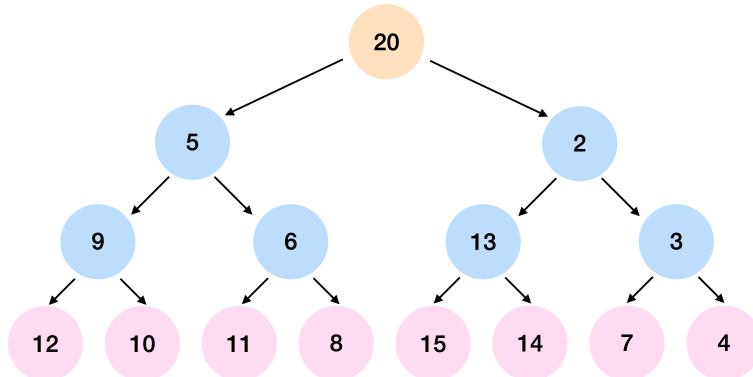
- **Albero binario completo:** **tutti** i livelli pieni;
- **Albero binario quasi completo:** tutti i livelli **tranne l'ultimo** pieni, ultimo livello riempito da sinistra a destra.

Gli alberi binari possono essere rappresentati in **tre modi principali**: tramite **array**, **vettore dei padri** o **puntatori**, ciascuno con diversi vantaggi e svantaggi.

Alberi binari: rappresentazione tramite array

Nella **rappresentazione tramite array** di un albero binario:

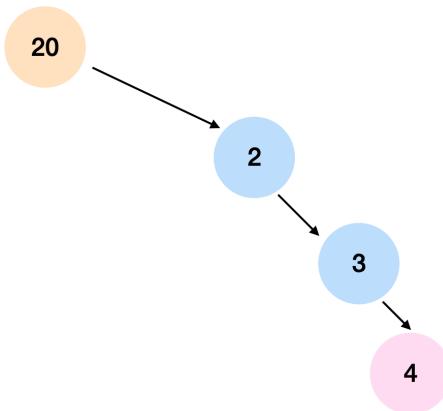
- Figlio sinistro del nodo in posizione i : $2i + 1$;
- Figlio destro del nodo in posizione i : $2i + 2$;
- Genitore del nodo in posizione i : $\left\lfloor \frac{i - 1}{2} \right\rfloor$.



La **rappresentazione** tramite array dell'albero **completo** è:

[20, 5, 2, 9, 6, 13, 3, 12, 10, 11, 8, 15, 14, 7, 4]

Svantaggio principale: la rappresentazione tramite array non è efficiente per alberi **sbilanciati**, poiché può richiedere uno spazio di memoria **esponenziale** ($\Theta(2^n)$ per rappresentare un albero con n nodi).



La **rappresentazione** tramite array dell'albero **degenero** è:

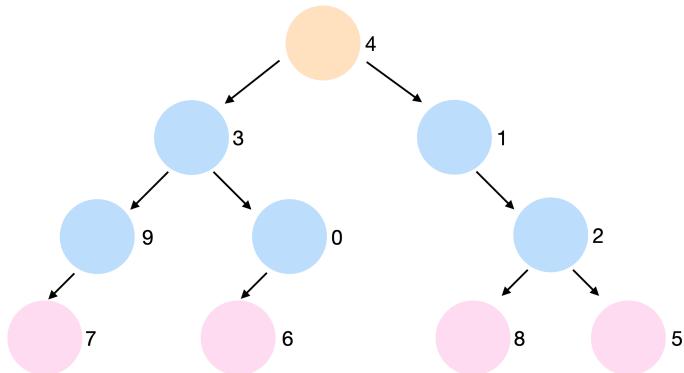
[20, None, 2, None, None, None, 3, None, None, None, None, None, None, 4]

Alberi binari: rappresentazione tramite vettore dei padri

Nella **rappresentazione tramite vettore dei padri**:

- Ogni **nodo** è associato a un **indice** in un array di dimensione n ;
- L'array **memorizza**, per ogni posizione i , l'**indice del nodo padre** corrispondente;
- La **radice** è identificata da un valore speciale (es. **-1**) che indica l'assenza del padre.

Ad **esempio** se l'albero è la seguente, con l'indice assegnato scritto accanto:



La **rappresentazione** tramite vettore dei padri è:

[3, 4, 1, 4, -1, 2, 0, 9, 2, 3]

Vantaggi:

- **Semplice e intuitiva**: ogni nodo è **identificato** da un indice e il padre è accessibile **direttamente**;
- **Efficienza in memoria**: uso di memoria **proporzionale** al numero di nodi ($\Theta(n)$), a differenza della rappresentazione con array che può richiedere $\Theta(2^n)$.

Svantaggi:

- **Operazioni limitate**: **ricerca** dei figli o **analisi** della struttura (bilanciamento, altezza) meno efficienti rispetto ad altre rappresentazioni;
- **Mancanza di informazioni sugli figli**: non distingue tra figlio **sinistro** e **destro**, memorizzando solo l'indice del padre.

Alberi binari: rappresentazione tramite puntatori

La **struttura del nodo** nella **rappresentazione tramite puntatori**:

- **Dato (key)**: contiene le informazioni del **nodo**;
- **Puntatore left**: riferimento al figlio **sinistro** (None se assente);
- **Puntatore right**: riferimento al figlio **destro** (None se assente).

L'**accesso** all'albero avviene esclusivamente attraverso il puntatore alla **radice**.

Caratteristiche:

- Ogni nodo è una struttura **autonoma** con riferimenti diretti ai figli;
- Rappresentazione **dinamica** che mantiene esplicitamente le relazioni padre–figli;
- Permette un **accesso diretto** alla struttura gerarchica dell’albero.

Vantaggi:

- **Flessibilità**: non richiede di conoscere a priori la **dimensione** dell’albero;
- **Efficienza**: **inserimenti** ed **eliminazioni** più efficienti rispetto alla rappresentazione con array.

Svantaggi:

- **Gestione memoria complessa**: necessità di allocazione **dinamica**;
- **Overhead di memoria**: ogni nodo occupa **più spazio** per via dei puntatori rispetto alla rappresentazione con array.

Alberi binari: implementazione in Python (rappresentazione tramite puntatori)

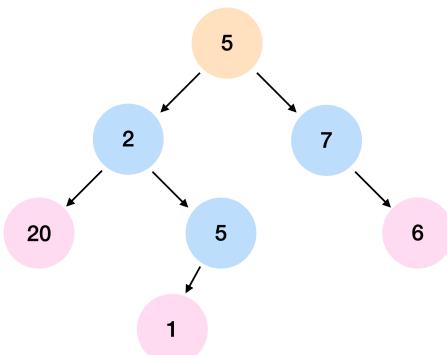
```
class NodoAB:  
    def __init__(self, key = None, left = None, right = None):  
        self.key = key  
        self.left = left  
        self.right = right
```

Ogni nodo è un **oggetto** con campo **key** per il **valore** e campi **left** e **right** per i riferimenti ai figli, gli alberi si costruiscono collegando **manualmente** i nodi.

Ad **esempio** con il seguente codice:

```
radice = NodoAB(5)  
radice.left = NodoAB(2)  
radice.right = NodoAB(7)  
radice.left.right = NodoAB(5)  
radice.right.right = NodoAB(6)  
radice.left.right.left = NodoAB(1)  
radice.left.left = NodoAB(20)
```

Si ottiene il seguente **albero binario**:

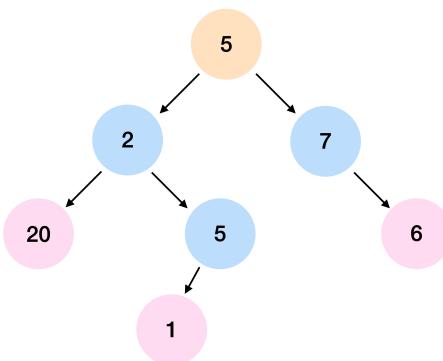


Alberi binari: esempio di funzione stampa(p, h = 0)

La funzione `stampa(p, h=0)` **stampà ricorsivamente** le chiavi di un albero binario con la chiave del nodo corrente e le chiavi dei sottoalberi sinistro e destro, **indentate**.

```
def stampa(p, h=0):
    if p == None:
        print('| ' * h + '-')
    else:
        print('| ' * h + str(p.key))
        stampa(p.left, h + 1)
        stampa(p.right, h + 1)
```

Ad **esempio** dato il seguente albero con la radice che si chiama **radice**:



Usando `stampa(radice)` si ottiene la seguente **stampa**:

```
5
| 2
| | 20
| | |
| | |
| | 5
| | | 1
| | | |
| | | |
| | |
| | 7
| | |
| | 6
| | |
| | |
```

Alberi binari: esempio di funzione generaAlbero(n, m)

La funzione `generaAlbero(n, m)` genera un albero binario casuale con `n` nodi e chiavi casuali nell'intervallo [1, `m`].

```
import random

def generaAlbero(n, m):
    if n == 0:
        return None
    p = NodoAB(random.randint(1, m))
    s = random.randint(0, n - 1)
    p.left = generaAlbero(s, m)
    p.right = generaAlbero(n - 1 - s, m)
    return p
```

Visite di alberi

La **visita dell'albero** è un'operazione fondamentale che consiste nell'**accesso sistematico** a tutti i nodi di un albero per eseguire operazioni specifiche su ciascuno.

Mentre nelle **liste** l'iterazione è **lineare**, negli alberi la struttura più complessa richiede **approcci specifici**.

Quindi il processo di **accesso sequenziale** ai nodi è chiamato visita dell'albero.

Negli alberi binari, esistono tre principali **strategie di visita**, ciascuna definita dall'ordine in cui si accede al nodo corrente rispetto ai suoi sottoalberi:

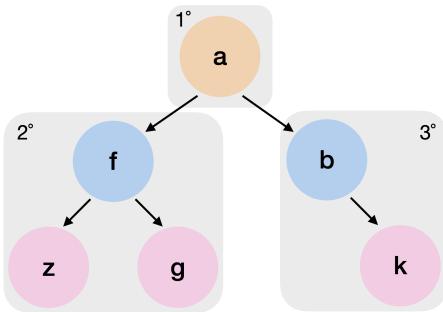
- **Preordine (preorder)**: visita prima il nodo **corrente**, poi il sottoalbero **sinistro**, infine il sottoalbero **destro**;
- **Inordine (inorder)**: visita prima il sottoalbero **sinistro**, poi il nodo **corrente**, infine il sottoalbero **destro**.
- **Postordine (postorder)**: visita prima il sottoalbero **sinistro**, poi il sottoalbero **destro**, infine il nodo **corrente**.

Tutte e tre le visite applicano **ricorsivamente** lo stesso criterio a ogni sottoalbero, garantendo coerenza nella traversata.

Esempio di visite dell'albero:

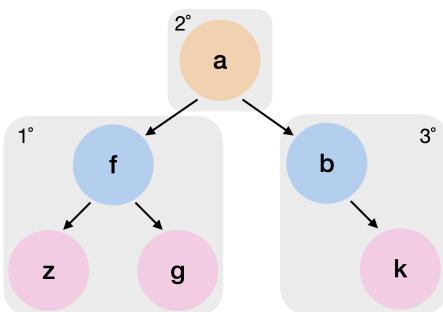


Visita in **preordine**:



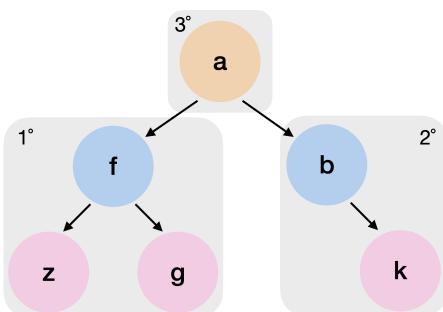
```
def VisitaPreordine(p):
    if p != None:
        # [accesso al nodo e operazioni consequenti]
        VisitaPreordine(p.left)
        VisitaPreordine(p.right)
```

Visita in **inordine**:



```
def VisitaPreordine(p):
    if p != None:
        VisitaPreordine(p.left)
        # [accesso al nodo e operazioni consequenti]
        VisitaPreordine(p.right)
```

Visita in **postordine**:



```
def VisitaPreordine(p):
    if p != None:
        VisitaPreordine(p.left)
        VisitaPreordine(p.right)
        # [accesso al nodo e operazioni consequenti]
```

Visite di alberi: costo computazionale

Tutte e tre le visite hanno lo **stesso costo computazionale**.

L'**equazione di ricorrenza** per rappresentazione tramite puntatori è:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \\ T(k) + T(n - 1 - k) + \Theta(1) & \text{altrimenti} \end{cases}$$

Dove:

- n è il numero **totale** di nodi;
- k è il numero di nodi nel sottoalbero **sinistro** ($0 \leq k \leq n - 1$);
- $\Theta(1)$ è il costo per visitare la **radice**.

Nel caso **ottimale**, quando l'albero è **completo** (bilanciato perfettamente), la suddivisione è:

$$k \approx \frac{n}{2} \implies 2T\left(\frac{n}{2}\right) + \Theta(1)$$

Applicando il caso 1 del **teorema principale** si ottiene:

$$T(n) = \Theta(n^{\log_2 2}) = \Theta(n)$$

Nel caso **peggiore**, quando l'albero è **completamente sbilanciato** (tutti i nodi su un solo lato) si ha:

$$k = 0 \quad \text{oppure} \quad n - 1 - k = 0 \implies T(n) = T(n - 1) + \Theta(1)$$

Applicando il **metodo iterativo** si ottiene:

$$T(n) = \Theta(n)$$

Dai risultati precedentemente ottenuti, la complessità risulta **sempre** $\Theta(n)$ indipendentemente dalla forma dell'albero, possiamo dimostrarlo formalmente usando il **metodo di sostituzione**:

- Eliminiamo la notazione asintotica sostituendo con due **costanti** a e b :

$$T(n) = \begin{cases} a & \text{se } n = 1 \\ T(k) + T(n - 1 - k) + b & \text{altrimenti} \end{cases}$$

- **Ipotizziamo** $T(n) \leq cn$, dove c è una costante da determinare:

- **Passo base:** $T(1) = a \leq c$ che è vero se prendiamo $c \geq a$;
- **Passo induttivo:**

$$\begin{aligned} T(n) &= T(k) + T(n - 1 - k) + b \leq ck + c(n - 1 - k) + b \leq cn \\ ck + cn - c - ck + b &\leq cn \\ cn - c + b &\leq cn \\ b &\leq c \end{aligned}$$

Il passo induttivo è vero se prendiamo $c \geq b$

- Quindi possiamo **scegliere** $c = \max(a, b)$ per garantire il passo base e il passo induttivo. Così abbiamo **dimostrato** che $T(n) = O(n)$.

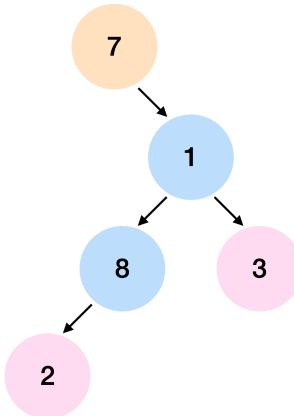
Si **dimostra** analogamente che $T(n) \geq c'n$ quindi che $T(n) = \Omega(n)$, ne segue che $T(n) = \Theta(n)$.

Applicazioni delle visite:

Le **visite** sono strumenti fondamentali per **ispezionare** la struttura dell'albero e **dedurre** proprietà specifiche, la scelta del tipo di visita dipende dalla proprietà da esaminare, poiché ciascuna visita offre un diverso ordine di accesso ai nodi.

Applicazioni delle visite: contaNodi(p)

Progettare una funzione che, **dato** un albero binario tramite puntatore alla radice, **restituisca** in tempo $\Theta(n)$ il **numero totale di nodi** (es. 5 nodi per l'albero mostrato).



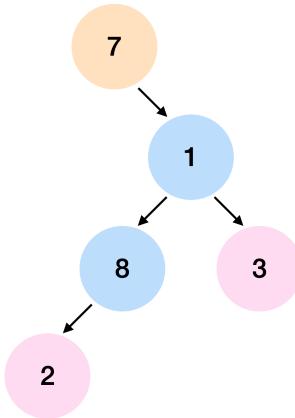
```
def contaNodi(p):
    if p == None:
        return 0
    ns = contaNodi(p.left) # nodi sinistro
    nd = contaNodi(p.right) # nodi destro
    return ns + nd + 1
```

Oppure in modo più **compatto**:

```
def contaNodi(p):
    if p == None:
        return 0
    return contaNodi(p.left) + contaNodi(p.right) + 1
```

Applicazioni delle visite: cerca(p, x)

Progettare una funzione che, **dato** un albero tramite puntatore alla radice e un intero **x**, **restituisca** in tempo $O(n)$ **True** se **x** è **presente** tra le chiavi dei nodi, **False** altrimenti (es. **True** per **x=8**, **False** per **x=5**, per l'albero mostrato).



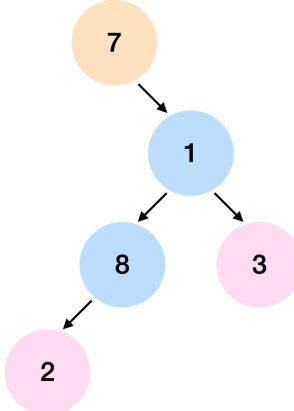
```
def cerca(p, x):
    if p == None:
        return False
    if p.key == x:
        return True
    if cerca(p.left, x):
        return True
    return cerca(p.right, x)
```

Oppure in modo più **compatto**:

```
def cerca(p, x):
    if p == None:
        return False
    return p.key == x or cerca(p.left, x) or cerca(p.right, x)
```

Applicazioni delle visite: altezza(p)

Progettare una funzione ricorsiva che, **dato** il puntatore alla radice di un albero, **calcoli** in tempo $\Theta(n)$ l'**altezza** (con albero vuoto = -1 e singolo nodo = 0; esempio: altezza 3 per l'albero in figura).



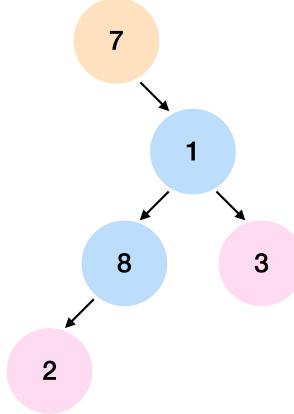
```
def altezza(p):
    if p == None:
        return -1
    ns = altezza(p.left)
    nd = altezza(p.right)
    return max(ns, nd) + 1
```

Oppure in modo più **compatto**:

```
def altezza(p):
    if p == None:
        return -1
    return max(altezza(p.left), altezza(p.right)) + 1
```

Applicazioni delle visite: contaLivello(p, k)

Progettare una funzione che, dato un albero (tramite puntatore alla radice) e un intero k , restituisca in tempo $O(2^k)$ il numero di nodi presenti al livello k (es. 2 nodi per $k=2$, 0 per $k=5$, 1 per $k=0$).



```
def contaLivello(p, k):
    if p == None:
        return 0
    if k == 0:
        return 1
    ks = contaLivello(p.left, k-1)
    kd = contaLivello(p.right, k-1)
    return ks + kd
```

Oppure equivalentemente:

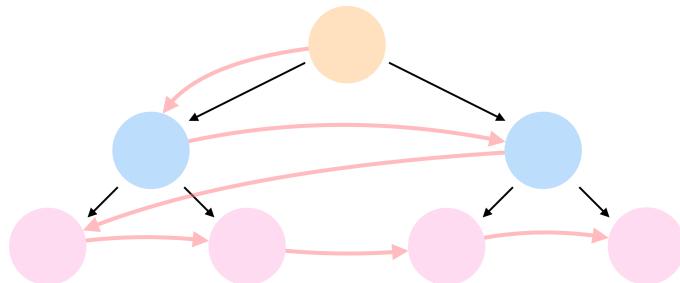
```
def contaLivello(p, k, i=0):
    if p == None:
        return 0
    if k == i:
        return 1
    ks = contaLivello(p.left, k, i+1)
    kd = contaLivello(p.right, k, i+1)
    return ks + kd
```

Il costo computazionale in entrambi i casi è:

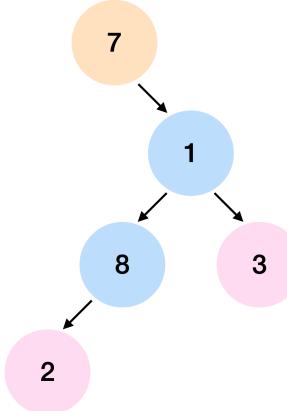
$\Theta(\text{numero di nodi che si trovano ad un livello } \leq k)$.

Applicazioni delle visite: stampaPerLivello(p)

Per **attraversare** un albero **livello per livello** (dalla radice alle foglie) è necessario utilizzare una **coda** d'appoggio, poiché le tradizionali visite ricorsive non lo consentono.



Progettare una funzione che, **dato** un albero tramite puntatore alla radice, **stampi** in tempo $\Theta(n)$ le **chiavi dei nodi** in ordine **crescente** di livello e **da sinistra a destra** (es. output 7, 1, 8, 3, 2 per l'albero in figura).



Si utilizza una **coda FIFO** partendo dalla **radice**, estraendo e stampando i nodi in **ordine** mentre si **accordano** i loro figli, garantendo che tutti i nodi di livello i siamo processati prima di quelli di livello $i + 1$.

```
def stampaPerLivello(p):
    if p != None:
        coda = [p]
        while coda:
            nodo = coda.pop(0)
            print(nodo.key, end = " ")
            if nodo.left:
                coda.append(nodo.left)
            if nodo.right:
                coda.append(nodo.right)
```

La visita per livelli implementata con **pop(0)** su lista ha **complessità** $O(n^2)$ poiché ogni estrazione costa $O(n)$ e viene ripetuta per tutti gli n nodi.

Utilizzando una `deque` dal modulo `collections` di Python si ottiene una visita per livelli con **complessità ottimale $O(n)$** .

```
from collections import deque

def stampaPerLivello(p):
    if p != None:
        coda = deque([p])
        while coda:
            nodo = coda.popleft()
            print(nodo.key, end = " ")
            if nodo.left:
                coda.append(nodo.left)
            if nodo.right:
                coda.append(nodo.right)
```

Per ottenere **complessità $O(n)$** è necessario usare una coda con estrazioni a tempo **costante $O(1)$** , ad esempio implementando una cancellazione logica tramite indice **incrementabile**.

```
def stampaPerLivello(p):
    if p != None:
        coda = [p]
        testa = 0
        while testa < len(coda):
            nodo = coda[testa]
            testa += 1
            print(nodo.key, end = " ")
            if nodo.left:
                coda.append(nodo.left)
            if nodo.right:
                coda.append(nodo.right)
```

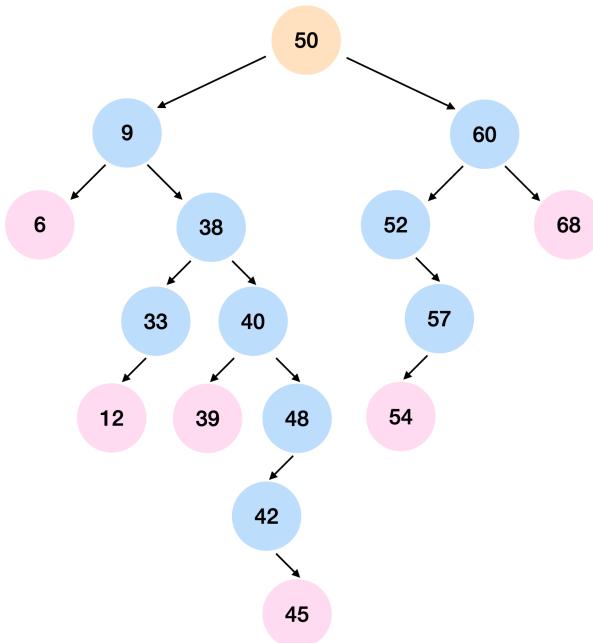
Alberi Binari di Ricerca (ABR)

Nelle **liste concatenate** la ricerca ha complessità $\Omega(n)$ poiché richiede l'esplorazione sequenziale dei nodi. Negli **alberi binari** un nodo può essere raggiunto in tempo $O(h)$, dove h è l'altezza dell'albero. Con **alberi bilanciati** ($h \approx \log_2 n$), la ricerca ottiene **complessità ottimale** $O(\log n)$, poiché il numero di nodi è $\Theta(2^h)$.

Proprietà degli alberi binari di ricerca:

- Ogni nodo contiene una **chiave unica e distinta**;
- La chiave di ogni nodo è **maggior**e di tutte le chiavi nel suo sottoalbero **sinistro**;
- La chiave di ogni nodo è **minore** di tutte le chiavi nel suo sottoalbero **destro**.

Questa struttura garantisce ricerche efficienti grazie alla proprietà di **ordinamento** che guida la navigazione dall'**alto** verso il **basso**.



Un **dizionario** è una struttura dati **dinamica** che gestisce elementi (**chiave–valore**) tramite tre operazioni fondamentali: **insert**, **search** e **delete**, dove la chiave funge da **identificatore univoco** per l'accesso ai valori.

Esempi di chiavi univoche nei dizionari:

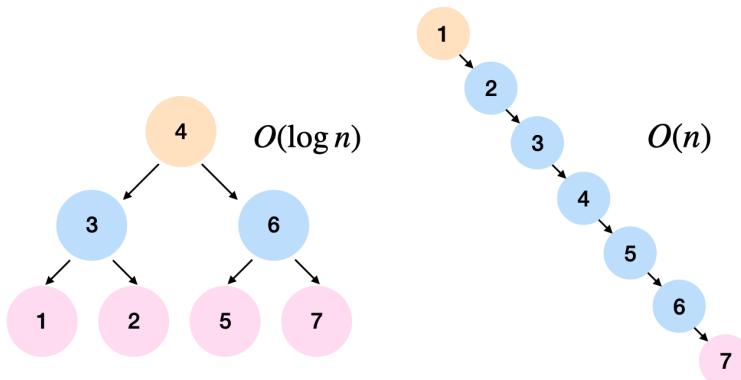
- **Matricola studente**: la matricola funge da chiave univoca per identificare gli studenti, associando a ciascuna il relativo nome, corsi e voti;
- **Codice ISBN**: l'ISBN è la chiave unica per i libri in una biblioteca, collegata a titolo, autore e altre informazioni;
- **Codice fiscale**: il codice fiscale identifica i cittadini, associandolo a dati personali come nome, cognome e indirizzo.

Alberi Binari di Ricerca: ricerca

L'algoritmo di **ricerca** in un ABR esegue una **discesa** guidata dai valori dei nodi (simile alla ricerca binaria) con complessità $O(h)$, dove h è l'altezza dell'albero.

```
def ricercaABR(p, x):
    while p is not None:
        if p.key == x:
            return p
        if x < p.key:
            p = p.left
        else:
            p = p.right
    return None
```

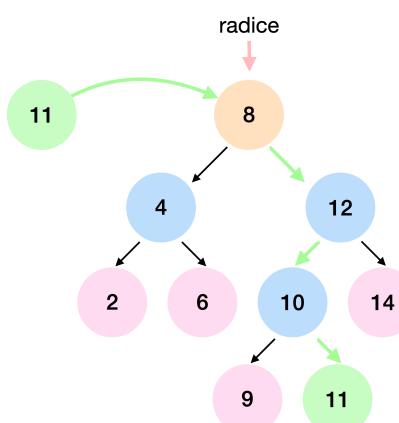
La ricerca in un ABR non garantisce automaticamente complessità $O(\log n)$ perché la struttura non controlla l'altezza, richiedendo tecniche di **bilanciamento** per ottenere prestazioni ottimali.



Alberi Binari di Ricerca: inserimento

L'**inserimento** in un ABR avviene tramite una **discesa** guidata dai valori dei nodi fino a trovare la posizione corretta (puntatore **None**) dove **inserire** il nuovo nodo, **mantenendo** la struttura dell'albero binario di ricerca.

Ad **esempio**, illustrazione di **inserisciABR(radice, 11)**:



```

def inserisciABR(root, x):
    z = NodoABR(x)

    if root == None:
        return z

    nodo = root

    while True:

        if z.key < nodo.key:
            if nodo.left is None:
                nodo.left = z
                break
            nodo = nodo.left

        elif z.key > nodo.key:
            if nodo.right is None:
                nodo.right = z
                break
            nodo = nodo.right

        else:
            break

    return root

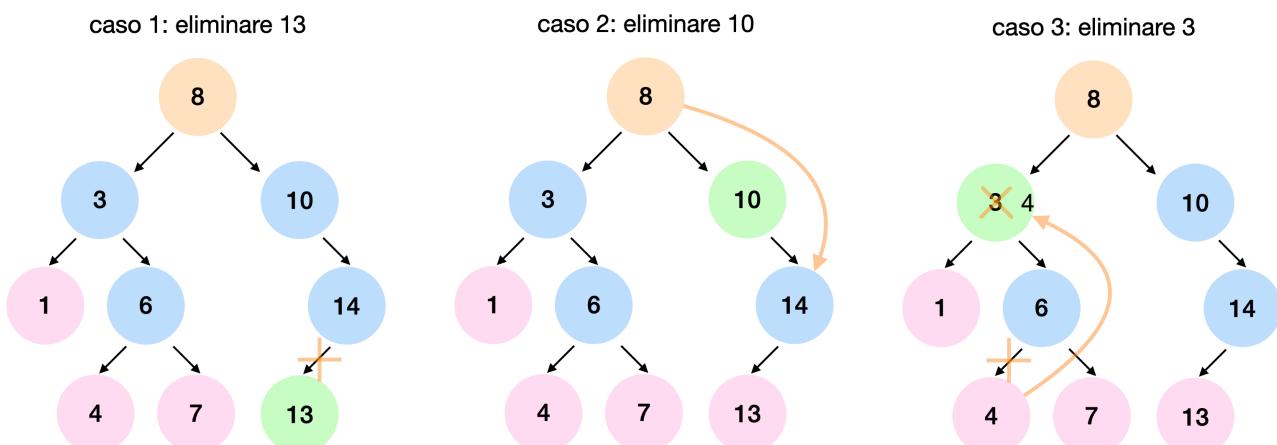
```

Il costo computazionale di `inserisciABR(root, x)` è $O(h)$, dove h è l'altezza dell'albero.

Alberi Binari di Ricerca: cancellazione

La **cancellazione** in un ABR prevede **tre casi** in base al **numero di figli** del nodo da eliminare:

- **Nodo senza figli:** si **elimina** la foglia **sostituendo** il puntatore del padre con **None**;
- **Nodo con un solo figlio:** si **collega** direttamente il padre al figlio unico, mantenendo la struttura dell'albero;
- **Nodo con due figli:** si **sostituisce** il suo valore con quello del successore e si procede a **eliminare** quest'ultimo (che avrà al massimo un figlio).



```

def cancellaABR(root, x):

    if root is None:
        return None
    nodo = root
    genitore = None

    # Trova il nodo da cancellare e il suo genitore
    while nodo is not None and nodo.key != x:
        genitore = nodo
        if x < nodo.key:
            nodo = nodo.left
        else:
            nodo = nodo.right
    if nodo is None:
        return root

    # Caso 1: nodo da cancellare è una foglia
    if nodo.left is None and nodo.right is None:
        if genitore is None:
            return None
        elif genitore.left == nodo:
            genitore.left = None
        else:
            genitore.right = None

    # Caso 2: nodo da cancellare ha un solo figlio
    elif nodo.left is None or nodo.right is None:
        if nodo.left is not None:
            figlio = nodo.left
        else:
            figlio = nodo.right
        if genitore is None:
            return figlio
        elif genitore.left == nodo:
            genitore.left = figlio
        else:
            genitore.right = figlio

    # Caso 3: nodo da cancellare ha due figli
    else:
        successore_gen = nodo
        successore = nodo.right
        while successore.left:
            successore_gen = successore
            successore = successore.left
        nodo.key = successore.key
        if successore_gen.left == successore:
            successore_gen.left = successore.right
        else:
            successore_gen.right = successore.right

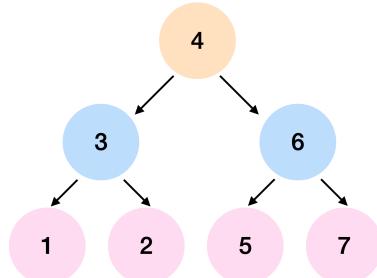
return root

```

Alberi Binari di Ricerca: stampaABR(p)

Per **stampare** in ordine **crescente** le chiavi di un ABR è sufficiente eseguire una visita **inorder**, che processa prima il sottoalbero sinistro, poi la radice e infine il destro.

Ad **esempio** dato l'albero:

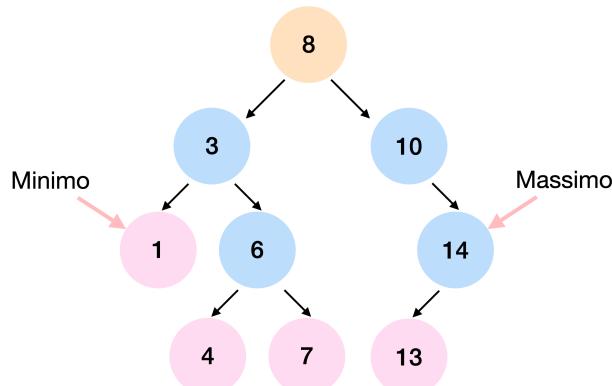


Viene stampato: 1 2 3 4 5 6 7

```
def stampaABR(p):
    if p != None:
        stampaABR(p.left)
        print(p.key, end = " ")
        stampaABR(p.right)
```

Alberi Binari di Ricerca: minimoABR(p)

Per trovare il **minimo/massimo** in un ABR si scende **ricorsivamente a sinistra/destra** fino a trovare un nodo senza **figlio sinistro/destro**, con complessità $O(h)$.



```
def minimoABR(p):
    if not p:
        return None
    while p.left:
        p = p.left
    return p
```

```
def massimoABR(p):
    if not p:
        return None
    while p.right:
        p = p.right
    return p
```

Alberi Binari di Ricerca: generaABR(A)

Progettare una funzione che **generi** un ABR da una lista di n interi, con complessità $O(n \log n)$, utilizzando un approccio **ricorsivo** che assegna **casualmente** i sottoalberi sinistro e destro.

```
def generaABR(A):
    lista = sorted(A, reverse = True)
    return generaABR1(len(A), lista)

def generaABR1(n, lista):
    if n == 0:
        return None
    p = NodoABR()
    s = random.randint(0, n - 1)
    p.left = generaABR1(s, lista)
    p.key = lista.pop()
    p.right = generaABR1(n - 1 - s, lista)
    return p
```

DIZIONARIO

[Parte sui dizionari da aggiungere (Lezione 22)]