Alberi

Algoritmi 1 – Lezione 12

A. Monti
Sapienza Università di Roma

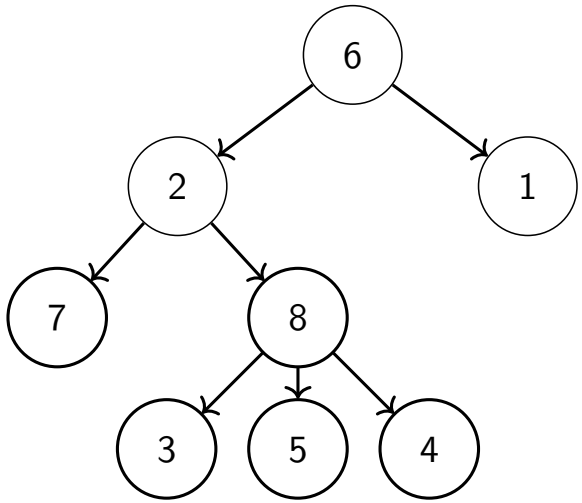
Corso: Algoritmi 1

Indice

Introduzione agli Alberi

Cos'è un albero (tree) in informatica?

Un **albero** è una **struttura dati gerarchica**, formata da elementi detti **nodi**, collegati tra loro da riferimenti (puntatori). Ogni nodo può avere **zero o più figli**, ma ha **al massimo un genitore**.



Confronto con le liste concatenate:

- Nelle **liste** i dati sono disposti linearmente, ogni nodo punta al successivo.
- Negli **alberi**, la disposizione è ramificata: un nodo può avere più rami (figli).

Terminologia di Base negli Alberi

Ecco alcuni termini fondamentali:

- **Radice** : nodo iniziale dell'albero, che non ha genitori.
- **Foglia**: nodo che non ha figli.
- **Nodo interno**: nodo che ha almeno un figlio.
- **Antenato di** x: un nodo nel cammino dalla radice ad x.
- **Discendente di** x: un nodo raggiungibile partendo da x.
- **Figlio di** x: nodo che ha x come genitore.
- **Fratelli**: nodi con lo stesso genitore.
- **Altezza dell' albero**: lunghezza del cammino più lungo dalla radice a una foglia.
- **Livello di un nodo**: distanza dalla radice (radice = livello 0).

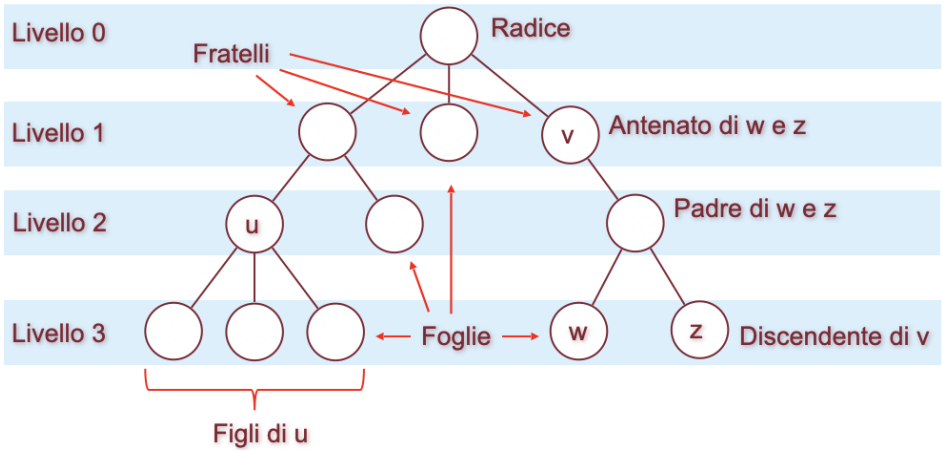


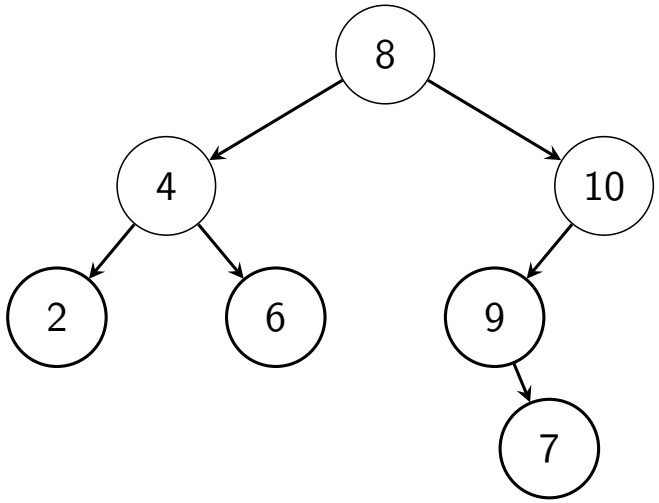
Figura: Esempio di albero con altezza $h = 3$

Osservazione: Un albero con n nodi ha sempre $n - 1$ archi (collegamenti), perché ogni nodo tranne la radice ha un solo genitore.

Alberi Binari Ordinati

Nel seguito ci occuperemo esclusivamente di **alberi binari ordinati**, una particolare struttura ad albero con le seguenti caratteristiche:

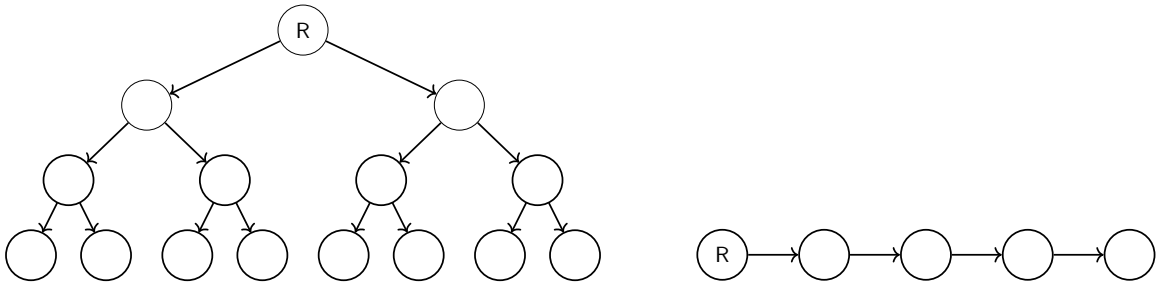
- Ogni nodo può avere **al massimo due figli**.
- I figli, se presenti, sono distinti e ordinati:
  - **Figlio sinistro**
  - **Figlio destro**



Confronto: Alberi Binari vs Liste Concatenate

Liste Concatenate e **Alberi Binari** sono strutture dati dinamiche, ma presentano una grande differenza nella quantità di dati accessibili in un numero limitato di passi:

- In una **lista concatenata**, ogni nodo punta solo al successivo.
  In al più t passi si possono raggiungere dalla radice fino a $t + 1$ nodi.
- In un **albero binario**, ogni nodo può avere due figli.
  In al più t passi si possono raggiungere dalla radice fino a $2^{t+1} - 1$ nodi (in un albero binario completo).
- Questo rende gli alberi particolarmente efficienti quando è necessario accedere o organizzare grandi quantità di dati in modo gerarchico.



Nell'albero binario completo di sinistra con al più 3 passi posso raggiungere $2^4 - 1 = 15$ nodi mentre nella lista concatenata di destra con al più 3 passi posso raggiungerne solo 4.

Rappresentazione in Memoria degli Alberi Binari

Un albero binario ordinato è generalmente rappresentato in memoria come una collezione di nodi, ciascuno con:

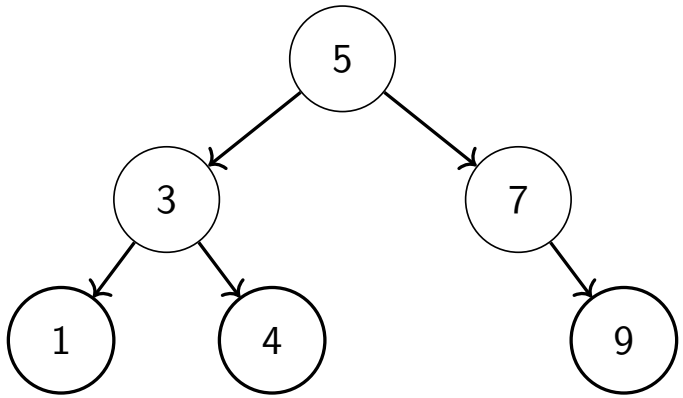
- Un campo `key` (o valore) che contiene il dato.
- Un puntatore al **figlio sinistro** (o `None` se assente).
- Un puntatore al **figlio destro** (o `None` se assente).

Esempio di definizione di nododi albero binario in Python:

```
class Nodo:
    def __init__(self, key, left=None, right=None):
        self.key = key
        self.left = left     # figlio sinistro
        self.right = right   # figlio destro
```

Esempio di Struttura in Memoria di un Albero Binario

Consideriamo la seguente struttura di albero binario:



Questo albero può essere costruito con oggetti della classe `Nodo` visti in precedenza:

```
a = Nodo(1)
b = Nodo(4)
c = Nodo(3, a, b)
d = Nodo(9)
e = Nodo(7, None, d)
radice = Nodo(5, c, e)
```

Nota: ogni nodo punta ai suoi figli con i campi `left` e `right`. Il nodo radice (5) ha due figli, il 3 (a sinistra) e il 7 (a destra).

Creazione di un Albero Binario

Ecco una funzione che, dato intero n, restituisce il puntatore alla radice di un albero binario casuale di n nodi (con chiavi 0, 1, ...$n-1$) in tempo $O(n)$.

```
import random

def genera_albero_binario_casuale(n):
    if n <= 0:
        return None
    # 1. Creiamo i nodi e mescoliamoli per avere chiavi casuali
    nodi = [NodoA(i) for i in range(n)]
    random.shuffle(nodi)
    # 2. La radice è il primo nodo della lista mescolata
    radice = nodi[0]
    # Lista dei nodi che hanno ancora almeno un posto libero (max 2)
    disponibili = [radice]
    # 3. Iteriamo dal secondo nodo in poi
    for i in range(1, n):
        nuovo_nodo = nodi[i]
        # Scegliamo un padre casuale tra quelli con posti liberi
        x = random.randint(0, len(disponibili) - 1)
        padre = disponibili[x]
        # Assegniamo il nuovo_nodo a sinistra o a destra
        if padre.left is None and padre.right is None:
            # Se entrambi sono liberi, scegliamo a caso
            if random.random() < 0.5:
                padre.left = nuovo_nodo
            else:
                padre.right = nuovo_nodo
        elif padre.left is None:
            padre.left = nuovo_nodo
        else:
            padre.right = nuovo_nodo
        # IMPORTANTE: Aggiungiamo il nuovo nodo ai potenziali padri
        disponibili.append(nuovo_nodo)
        # Se il padre è ora pieno (ha 2 figli), lo rimuoviamo dai disponibili in O(1)
        if padre.left is not None and padre.right is not None:
            disponibili[x] = disponibili[-1]
            disponibili.pop()
    return radice
```

Ordini di Visita in un Albero Binario

Nelle **liste concatenate**, i nodi sono collegati linearmente: c'è un unico modo naturale per visitarli, seguendo i puntatori da testa a coda.

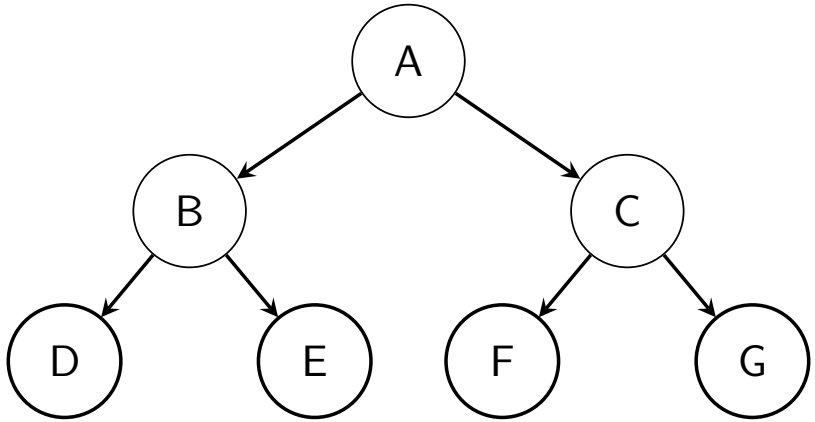
Negli **alberi binari** invece, la struttura ramificata permette diversi modi di attraversare i nodi. I tre principali ordini di visita ricorsiva sono:

- **Pre-ordine (preorder)**:
  - Visita il nodo corrente
  - Visita il sottoalbero sinistro
  - Visita il sottoalbero destro
- **In-ordine (inorder)**:
  - Visita il sottoalbero sinistro
  - Visita il nodo corrente
  - Visita il sottoalbero destro
- **Post-ordine (postorder)**:
  - Visita il sottoalbero sinistro
  - Visita il sottoalbero destro
  - Visita il nodo corrente

Nota: Questi metodi sono tutti basati su **ricorsione** e seguono percorsi diversi,

Esempio di Visite in un Albero Binario

Consideriamo il seguente albero binario:



Visite dell'albero in profondità :

- **Preorder** (Radice, Sinistra, Destra): `A B D E C F G`
- **Inorder** (Sinistra, Radice, Destra): `D B E A F C G`
- **Postorder** (Sinistra, Destra, Radice): `D E B F G C A`

Visite Ricorsive: Preorder / Inorder / Postorder

Di seguito le versioni ricorsive per stampare i nodi di un albero binario nei tre ordini:

```
def Preorder(p):
    if p is None:
        return
    print(p.key)            # visita il nodo
    Preorder(p.left)        # poi il sottoalbero sinistro
    Preorder(p.right)       # infine il sottoalbero destro


def Inorder(p):
    if p is None:
        return
    Inorder(p.left)         # visita il sottoalbero sinistro
    print(p.key)            # poi il nodo
    Inorder(p.right)        # infine visita il sottoalbero destro


def Postorder(p):
    if p is None:
        return
    Postorder(p.left)       # visita il sottoalbero sinistro
    Postorder(p.right)      # poi il sottoalbero destro
    print(p.key)            # infine il nodo
```

Osservazione: in tutti e tre i casi la complessità temporale è $O(n)$, dato che ciascun nodo viene visitato esattamente una volta.

Visita dell'albero per Livello

La **visita per livello** consiste nel visitare i nodi dell'albero **livello per livello**, partendo dalla radice e, all'interno di uno stesso livello, da sinistra a destra.

Implementazione con puntatori Versione semplice (non ottimale):

```
def stampa(p):
    if p is None:
        return
    coda = [p]
    while coda:
        nodo = coda.pop(0)
        print(nodo.key)
        if nodo.left:
            coda.append(nodo.left)
        if nodo.right:
            coda.append(nodo.right)
```

Problema: l'operazione `pop(0)` richiede tempo $O(n)$, portando la complessità complessiva a $O(n^2)$ nel caso peggiore.

Visita per Livello con Complessità Ottimale $O(n)$

Per ottenere una **complessità lineare** $O(n)$, possiamo evitare le rimozioni costose dalla lista:

Uso di *deque* dal modulo *collections*

```
from collections import deque

def stampa_per_livello(p):
    if p is None:
        return
    coda = deque([p])
    while coda:
        nodo = coda.popleft()  # O(1)
        print(nodo.key)
        if nodo.left:
            coda.append(nodo.left)
        if nodo.right:
            coda.append(nodo.right)
```

Vantaggio: deque è una coda efficiente: *popleft*( ) e *append*( ) richiedono tempo costante $O(1)$.

Calcolare il Numero di Foglie dell'Albero

Una **foglia** è un nodo che non ha figli. Per calcolare il numero di foglie in un albero, possiamo usare una funzione ricorsiva che attraversa l'albero e conta quanti nodi soddisfano la condizione di avere entrambi i figli *None*.

```
def conta_foglie(p):
    if p is None:
        # L'albero è vuoto: restituisci 0
        return 0
    if p.left is None and p.right is None:
        # L'albero è una  foglia: restituisci 1
        return 1
    # restituisci la somma delle foglie nei sottoalberi sinistro e destro
    return conta_foglie(p.left) + conta_foglie(p.right)
```

Calcolare l'Altezza dell'Albero

L'**altezza** di un albero è definita come la lunghezza del cammino più lungo dalla radice a una foglia. Nella rappresentazione tramite puntatori, può essere calcolata in modo ricorsivo.

```
def altezza(p):
    if p is None:
        # L'Albero vuoto ha altezza -1
        return -1
    # Altezza = 1 più il  massimo tra le altezze dei sottoalberi
    return 1 + max(altezza(p.left), altezza(p.right))
```

Rappresentazione di un Albero Binario con un Array

Un albero binario può essere rappresentato tramite un **array**, dove il nodo all'indice i ha:

- il **figlio sinistro** in posizione $2i + 1$
- il **figlio destro** in posizione $2i + 2$

Questa rappresentazione è particolarmente efficiente se l'albero è **completo** o **quasi completo**, come nel caso degli **heap** binari.

Esempio: heap rappresentato in array:

```
H = [10, 15, 30, 40, 50, 100]
```

In questo caso:

- Nodo con valore 15 è figlio sinistro di 10
- Nodo con valore 30 è figlio destro di 10
- ...

Vantaggi:

- Accesso diretto ai figli e al padre tramite formule aritmetiche
- Nessun overhead di puntatori

Limiti della Rappresentazione con Array per Alberi Sbilanciati

Se l'albero è molto sbilanciato, ad esempio se ogni nodo ha solo il figlio destro, l'array diventa inefficiente.

Esempio: Albero con solo figli destri.

- La radice si trova in posizione $a_0 = 0$
- Il secondo nodo (figlio destro) in posizione $a_1 = 2$
- Il terzo nodo in posizione $a_2 = 2 \cdot a_1 + 2 = 6$
- Il quarto nodo in posizione $a_3 = 2 \cdot a_2 + 2 = 14$
- Il quinto in posizione $a_4 = 2 \cdot a_3 + 2 = 30$
- ...

In generale: $a_k = 2 \cdot a_{k-1} + 2 \quad \text{con} \quad a_0 = 0 \Rightarrow a_k = 2^{k+1} - 2$

Conclusione: per un albero di n nodi lineari (solo figli destri), la dimensione necessaria dellarray è $\boldsymbol{O(2^n)}$.

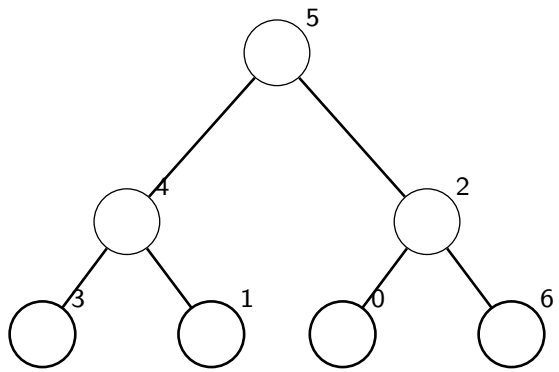
Morale: *la rappresentazione tramite array è adatta solo ad alberi binari compatti.*

Rappresentazione dell'Albero con un Vettore dei Padri

Un albero può essere rappresentato tramite un **vettore dei padri**, ovvero un array `P` di lunghezza n, dove n è il numero di nodi.
Ad ogni nodo dell'albero viene associato un indice da 0 a $n-1$, e l'array `P` è definito così:

- $P[i]$ contiene l'indice del **padre** del nodo i
- La radice ha $P[i] = -1$

Esempio: `P = [2,4,5,4,5,-1,2]`



In figura l'indice del nodo è scritto accanto ad esso. La struttura dell'albero è dedotta da `P`, leggendo per ogni nodo l'indice del proprio padre.

Pregi e Limiti della Rappresentazione tramite Vettore dei Padri

Il vettore dei padri rappresenta in modo compatto la struttura dell'albero, indicando per ogni nodo chi è il suo genitore.

Pregi principali:

- Semplice e compatta: richiede solo un array di lunghezza n.
- Facilita l'accesso diretto al padre di un nodo in tempo $O(1)$.
- Utile in contesti in cui l'albero è statico o la gerarchia è l'informazione principale.

Limiti principali:

- **Nessuna informazione sull'ordine dei figli:** non si può distinguere tra figlio sinistro e figlio destro.
- **Navigazione meno efficiente:** per trovare i figli di un nodo bisogna scansionare tutto il vettore.
- **Non adatta per operazioni tipiche degli alberi binari**, come visite in ordine.

ESERCIZI

Nelle soluzioni ricorsive degli esercizi che seguono, non devono essere utilizzate variabili globali, la complessità della soluzione proposta va sempre valutata e se possibile dovrebbe essere $O(n)$, dove n è il numero di nodi dell'albero.

Suggerimento: Prima di cercare le soluzioni in rete, prova a ragionarci da solo: gli esercizi che seguono sono pensati per aiutarti a mettere alla prova la tua comprensione, la tua capacità di sviluppare una prova di correttezza e costruire un algoritmo corretto.
Solo dopo averci riflettuto, confronta la tua soluzione con altre possibili versioni.

Esercizi

1. Progettare una funzione che, data la radice di un albero binario non vuoto, restituisca la coppia di valori (a, b) dove:
   - a è il nodo massimo dell'albero
   - b è nodo minimo dell'albero
2. Progettare una funzione che, data la radice di un un albero binario, restituisca la lista dei figli unici. Un nodo è figlio unico se è l'unico figlio del suo genitore.
3. Progettare una funzione che, data la radice di un un albero binario, cancella i nodi foglia dell'albero e restituisce la radice eventualmente modificata.
4. Progettare una funzione che, data la radice di un un albero binario ed un intero x, restituisca *True* se esiste nell'albero un cammino radice-foglia la somma dei cui nodi dia x, *False* altrimenti.
5. Progettare una funzione che, data la radice di un un albero binario, restituisca il valore massimo tra quelli dei cammini radice-foglia, dove il valore di un cammino è dato dalla somma dei suoi nodi.
6. Progettare una funzione che, data la radice di un albero binario, restituisca la somma massima tra quelle che si ottengono sommano i nodi che si trovano sui cammini che uniscono due foglie.
7. Progettare una funzione che, data la radice di un un albero binario, restituisca il valore che appare con maggior frequenza. In caso di parità di frequenza massima, restituisce il valore massimo.

Esercizi

1. Progettare una funzione che, data la radice di un un albero binario, cancella dall'albero tutti i nodi *semplici*. e restituisce la radice eventualmente modificata. Un nodo è semplice se ha un unico figlio.
2. Progettare una funzione che, data la radice di un un albero binario contenente interi distinti e il valore di due suoi nodi, restituisca **il primo antenato comune** ai due nodi. Il primo antenato comune a due nodi è definito come il primo nodo comune ai due cammini che dai nodi portano alla radice.
3. Progettare una funzione che, data la radice di un un albero binario ed un intero positivo k, restituisce la somma di tutti i nodi dell'albero che si trovano nei livelli al più k.
4. Progettare una funzione che, data la radice di un un albero binario, restituisce *True* se l'albero è **completo** ovvero che tutti i livelli dell'albero siano pieni), *False* altrimenti.
5. Progettare una funzione che, data la radice di un un albero binario, restituisce *True* se l'albero è **completo o quasi completo**, ovvero che tutti i livelli dell'albero siano pieni tranne al più l'ultimo che deve avere i nodi a sinistra, *False* altrimenti.
6. Progettare una funzione che, data la radice di un un albero binario ed un intero x, inserisce il nodo a valore x nel primo livello non pieno e all'interno di quel livello nella posizione libera più a sinistra.

Esercizi

1. Progettare una funzione che, data la radice di un albero binario ed una lista contenente le cifre della propria matricola, restituisca *True* se esiste un cammino radice-foglia le cui chiavi nell'ordine siano esattamente le cifre della matricola, *False* altrimenti. La complessità della funzione deve essere $O(1)$.
2. Progettare una funzione che, data la radice di un albero binario, restituisce la rappresentazione dell'albero tramite array. La dimensione dell'array deve essere minima.
3. Progettare una funzione che, dato un albero rappresentato tramite array, ne costruisca la rappresentazione tramite puntatori e restituisca la radice dell'albero. La complessità dell'algoritmo deve essere $O(n)$.