Documento a cura di Simone Remoli.

FILE INDICIZZATI.

(Questa trattazione è stata già affrontata nel PDF precedente).

Cosa succede se effettuo una ricerca su degli attributi che non sono la chiave di ricerca?
Bisogna scorrere tutti gli attributi: **full Scan.**
E nel caso di file hash è peggio perché devo trovare l'attributo in più Bucket.

Per definizione **la chiave di ricerca è esattamente la chiave primaria.**
Però oggigiorno è molto probabile effettuare delle ricerche su degli attributi che **non sono** la chiave primaria.

Un **file indicizzato** è un file che utilizza una struttura di **indice** per ottimizzare la ricerca e l'accesso ai dati in un database.

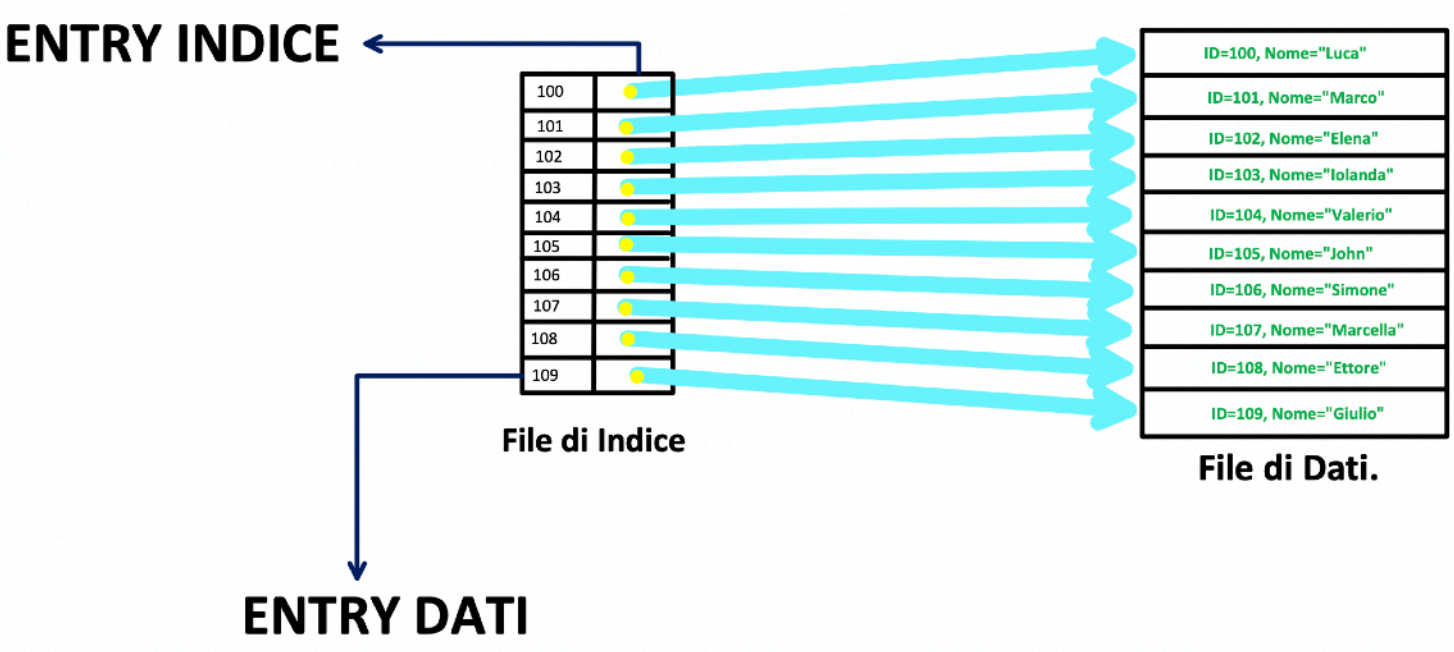
I file indicizzati servono ad ottimizzare la ricerca dei record nei file.

Il file di dati vero e proprio contiene i record effettivi della tabella: è separato dal file di indice.

Ogni qual volta che modifico il file originale devo andare a modificare l'indice, altrimenti non c'è corrispondenza tra i puntatori che questo indice mi sta fornendo rispetto alla posizione fisica dei record nel file.

Quindi, la presenza dell'indice favorisce i costi di ricerca ma ha un **costo di gestione.**

Nei file indicizzati, i dati vengono memorizzati in un file principale, mentre gli indici sono memorizzati in **una struttura separata** che mappa i valori chiave alle posizioni fisiche dei record.



L’entry dati permette di **trovare velocemente un record** conoscendo la **chiave di ricerca**!
L’entry indice aiuta il DBMS a trovare il record giusto nel file di dati.

Tipologie di Indici nei File Indicizzati.

Indice Primario (**Indice Clustered**).

L’indice è costruito sulla **chiave primaria**, quindi i dati sono ordinati fisicamente.
L’ordine nel file di dati è coerente con l’ordine del file di indice.
Un **Indice Clustered** è un **indice che ordina fisicamente i dati** nella tabella in base ai valori della colonna (PK) su cui è stato creato.
Quando una colonna è definita come **Primary Key (PK)**, il DBMS crea **automaticamente un Indice Clustered** su di essa (a meno che non sia specificato diversamente).
Quindi, se la tua PK è già indicizzata, non devi creare manualmente un Indice Clustered!

Esempio: Indice Clustered su **ID (immagine sopra).**
Ottimo per ricerche **WHERE ID = X.**

Le Range Query sono più veloci con un Indice Clustered?

Sì! Le Range Query (**BETWEEN, >, <, ORDER BY, LIKE 'A%'**) sono molto più efficienti con un Indice Clustered.

✅ Questo perché un Indice Clustered ordina fisicamente i dati, quindi il DBMS può scorrerli in modo sequenziale senza saltare tra blocchi di memoria non contigui.

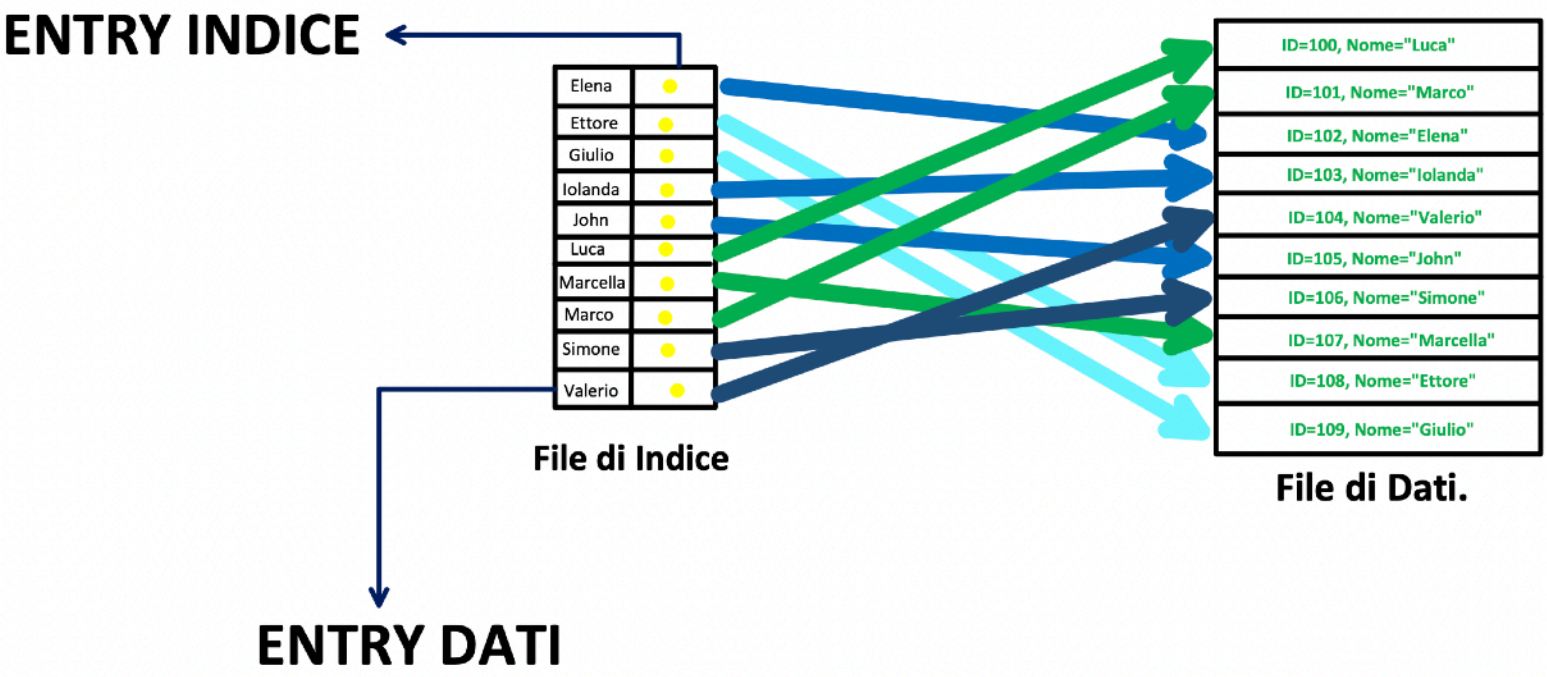
✅ Con un Indice Non-Clustered, invece, il DBMS deve accedere ai record in modo non sequenziale, rendendo la scansione più lenta.

Quando Creo un Indice Clustered su una Colonna **VARCHAR**, Essa Viene Ordinata Alfabeticamente?
Sì! Se crei un Indice Clustered su una colonna **VARCHAR**, il DBMS ordina fisicamente i dati in ordine alfabetico

La Modifica dell'Ordine in un Indice Clustered Avviene a Livello Fisico o nel File Indici?
Sì, la modifica avviene a livello fisico sulla tabella e NON solo nel file degli indici.
Un Indice Clustered riordina fisicamente i dati nella tabella in base alla colonna indicizzata.
Non esiste un "file indici" separato per un Indice Clustered, perché i dati stessi sono l'indice.

Indice Secondario (**Non-Clustered**).

L’indice è costruito su un **campo non chiave primaria** (es. Nome).
Permette ricerche su campi non chiave (**WHERE nome = ‘Luca’**).



Non supporta in maniera efficace la range query.
Negli indici **Non-Clustered**, l'ordine dell'indice non corrisponde all'ordine fisico dei dati nella tabella.
Se esegui una ***Range Query***, il DBMS deve recuperare ogni Record ID (RID) dalla lista di puntatori e accedere ai record reali uno per uno.
Questo genera accessi casuali al disco, rendendo la query più lenta!

Soluzioni principali:

1. **Usare un Indice Clustered sulla colonna della Range Query:**

```
CREATE CLUSTERED INDEX idx_ordini_data ON ordini(data_ordine);

SELECT * FROM ordini WHERE data_ordine BETWEEN '2024-01-01' AND '2024-01-31';
```

2. **Creare un Indice Coprente (Covering Index)**

Un Indice Coprente non clustered contiene tutti i dati richiesti dalla query, evitando accessi al file dati!

```
CREATE INDEX idx_ordini_data_totale ON ordini(data_ordine, totale);

SELECT * FROM ordini WHERE data_ordine BETWEEN '2024-01-01' AND '2024-01-31';
```

Gli indici primari **ordinano** i dati, mentre quelli secondari **accelerano** ricerche su altri campi (a una determinata pagina in quello slot, ricordiamo).
Negli indici Non-Clustered, l'indice è separato dal file dati, quindi il DBMS deve consultare prima l'indice e poi accedere ai record reali.

Regola fondamentale: Un indice di **secondo livello** (indice secondario) è tipicamente un indice **Non-Clustered** e viene ordinato secondo i valori della colonna su cui è costruito.
Se l'indice è su una colonna come **nome**, allora sarà ordinato **alfabeticamente**.
Non cambia l'ordine fisico dei dati nella tabella ma crea una struttura separata per accelerare la ricerca.

VANTAGGI E SVANTAGGI.

✅ Vantaggi

✔ **Ricerche Veloci (O(log n))** → Grazie all'indice, il DBMS trova subito il record.
✔ **JOIN più Efficienti** → Se la chiave di JOIN ha un indice, il DBMS evita scansioni complete.
✔ **Supporta Ricerche su Campi Non Chiave** → Utile per nome, email, ecc.

❌ Svantaggi

❌ **Inserimenti/Update più Lenti** → Gli indici devono essere aggiornati ogni volta che cambiano i dati.
❌ **Occupa più Spazio** → Gli indici devono essere memorizzati separatamente.
❌ **Non Sempre Utile** → Se la tabella è piccola, una scansione completa può essere più veloce.

Tipo di Indice	Ordinamento	Modifica l'Ordine Fisico dei Dati?	Quando usarlo?
Clustered Index	Sì (fisico)	Sì	Per ricerche su `id` o `data`

Non-Clustered Index	Sì (logico)	No	Per ricerche su `nome`, `email`, `città`

Puoi creare indici Non-Clustered su altre colonne per velocizzare le ricerche (`email`, `nome`).

Riassumendo:

- Un indice è **raggruppato** quando le sue entry dati *sono memorizzate secondo un ordine coerente* con (o identico a) l'ordine dei record dati nel file dati.
- In caso contrario, l'indice è **non raggruppato**.

Teorema Fondamentale dell'Indicizzazione:

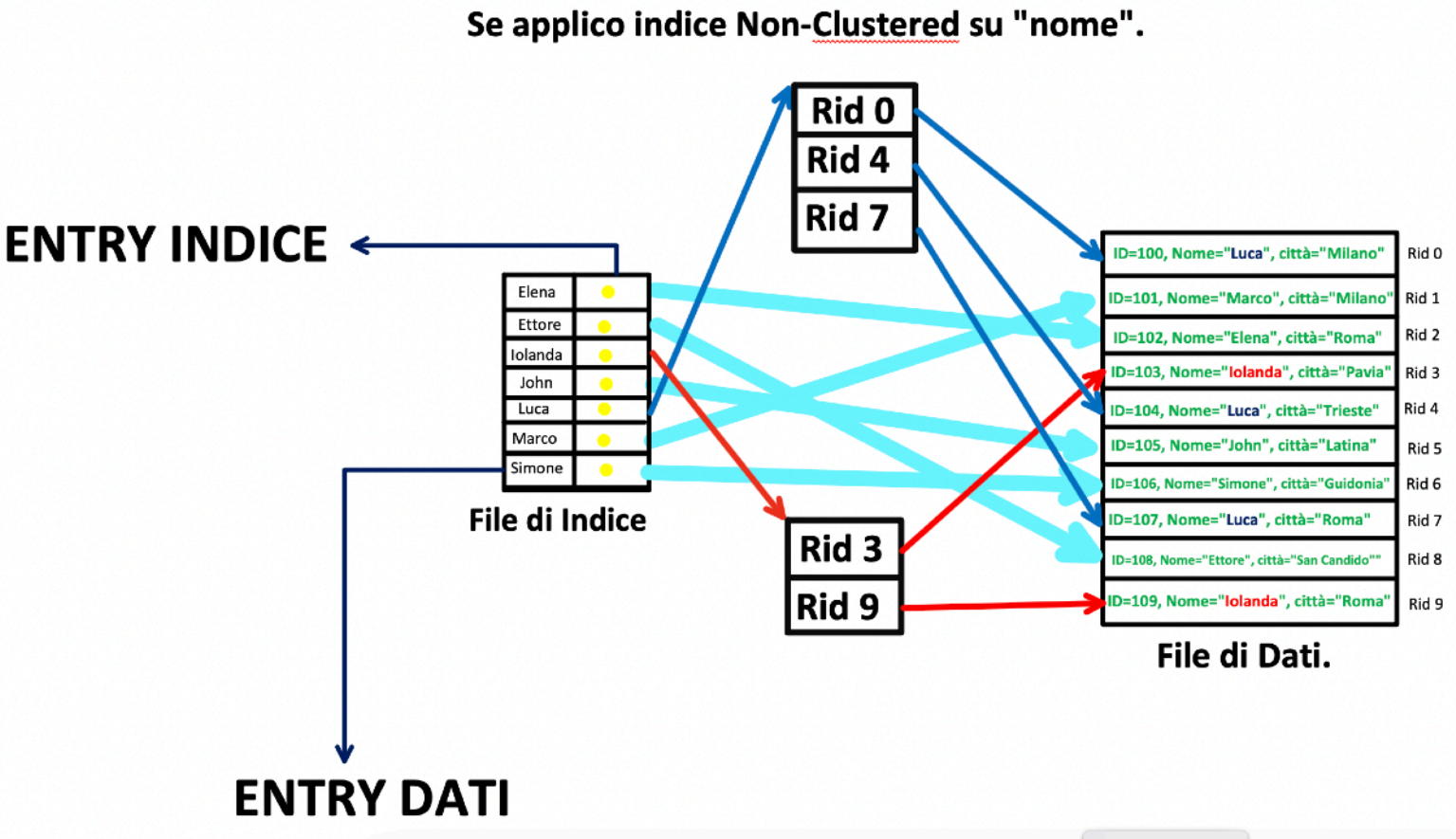
Qualora siano presenti due **entry** dati duplicate, ossia caratterizzate dallo stesso valore della **chiave di ricerca**, tale situazione risulta incompatibile con l'impiego di un **indice primario**.
Quest'ultimo, infatti, impone il vincolo di **unicità**, impedendo la presenza di chiavi duplicate all'interno della struttura dati.Di conseguenza, in un contesto in cui si utilizza un **indice primario**, ogni chiave deve necessariamente essere univoca, garantendo un accesso deterministico ai dati e preservando l'integrità della base di dati.

Se un **Indice Secondario (Non-Clustered)** è costruito su una colonna con **chiavi duplicate** (es. `nome`), il DBMS utilizza una **lista di puntatori (RID)** per collegare più record a un unico valore della chiave di ricerca.

Struttura di un Indice Secondario con Chiavi Duplicate.

```
SELECT * FROM clienti WHERE nome = 'Luca';
```

Il DBMS segue questi passi: Cerca `Luca` nell'indice ordinato, trova la lista di RID associati (`[RID 0, RID 4, RID 7]`) e scorre tutti i RID nella lista e accede ai record nel file dati.



Se il numero di duplicati per **nome** è elevato (troppi **Luca**) e vuoi ridurre il numero di RID letti e migliorare le prestazioni delle query, si può optare per un **indice composito.**

INDICE COMPOSITO.

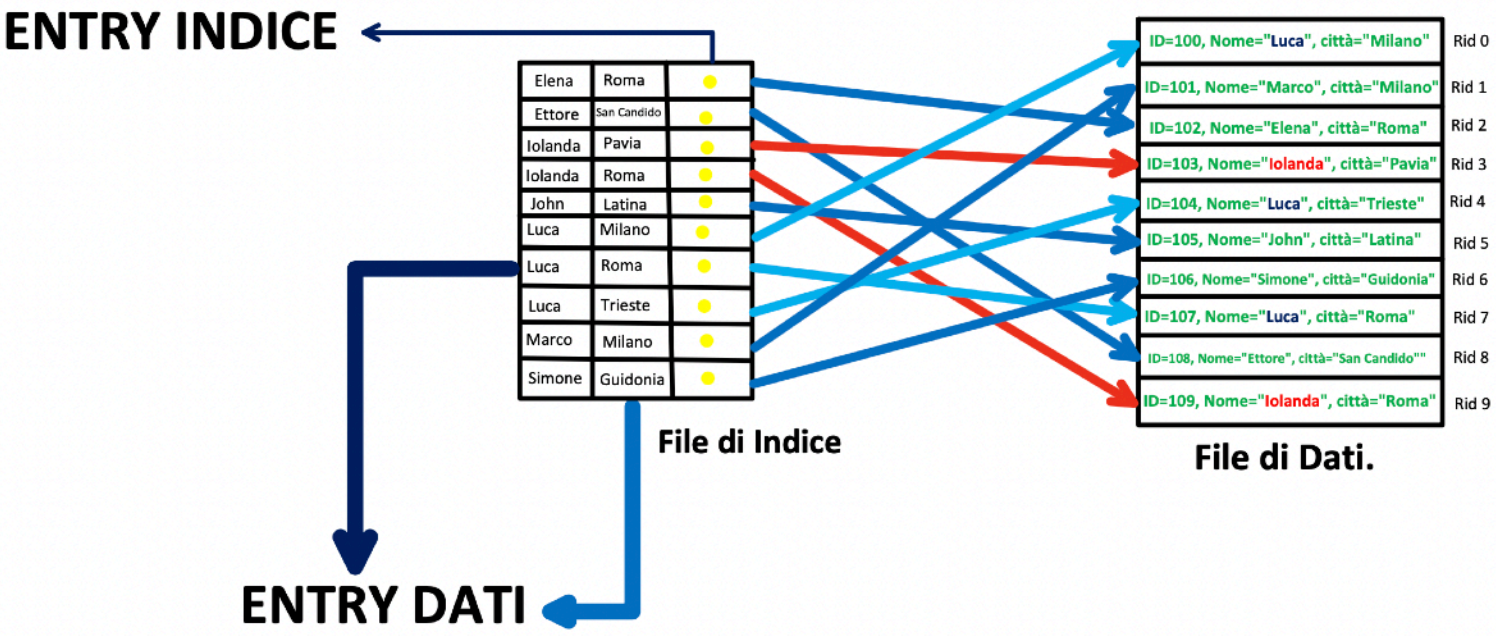
```
SELECT * FROM clienti WHERE nome = 'Luca' AND città = 'Milano';
```

```
CREATE INDEX idx_clienti_nome_citta ON clienti(nome, città);
```

Prima vengono ordinati i valori della colonna **nome** in ordine alfabetico.Poi, all'interno di ogni gruppo con lo stesso **nome**, i valori di **città** sono ordinati in ordine alfabetico. Questo significa che il DBMS può trovare rapidamente **Luca, Milano** senza dover scansionare tutti i **Luca**.

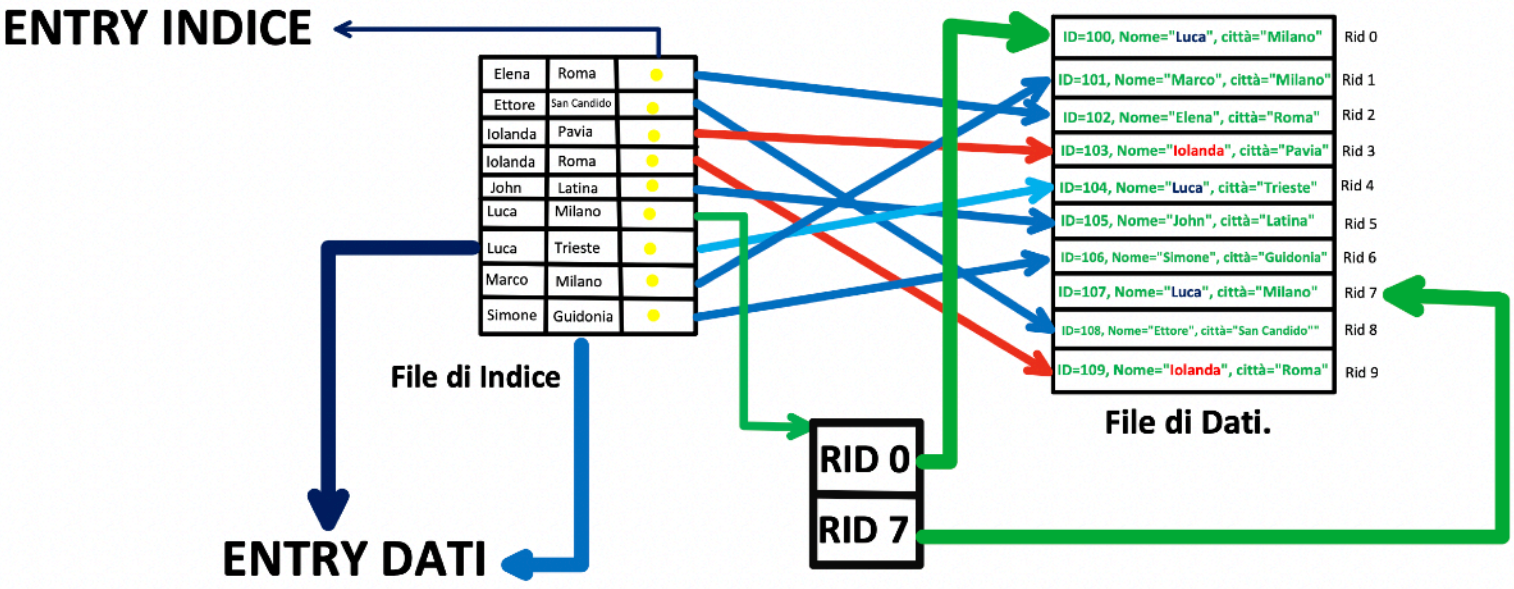
Indice Composito con Duplicati.

Se applico indice Non-Clustered su "nome" e "città".



Se esistono più Luca, Milano, il DBMS mantiene una **lista di puntatori (RID)** per accedere ai record corrispondenti. Questa lista di RID funziona come negli Indici Non-Clustered tradizionali, solo che è organizzata secondo l'ordinamento dell'Indice Composito.

Se applico indice Non-Clustered su "nome" e "città".



Indice Secondario Unico (Unique Index).

Un **Indice Secondario** è chiamato "Unico" (**UNIQUE INDEX**) se la sua chiave di ricerca è una colonna che non contiene valori duplicati.

Non è una chiave primaria (**PRIMARY KEY**) poiché una (**PK**) ha un Indice Clustered Unico per impostazione predefinita, ma garantisce comunque che i valori siano univoci nella colonna indicizzata.

Un Indice Secondario Unico (**UNIQUE INDEX**) può essere creato su qualsiasi colonna che non sia una PK, ma che deve comunque contenere valori unici.

Può essere creato su una singola colonna o su più colonne (Indice Secondario Unico Composito).

Esempio: Creazione di un Indice Secondario Unico su email.

```
CREATE UNIQUE INDEX idx_clienti_email ON clienti(email);
```

Se proviamo ad inserire un duplicato:

```
INSERT INTO clienti (id, nome, email) VALUES (4, 'Daniele', 'alice@email.com');
```

Supponendo che alice@email.com sia già presente nella tabella clienti colonna indice, la query restituisce **ERRORE**: L'indice **UNIQUE** impedisce il duplicato. ❌

Anche se email non è PK, funziona lo stesso.

Indice Secondario Unico Composito.

Possiamo garantire l'unicità su una combinazione di colonne con un Indice Composito Unico.

```
CREATE UNIQUE INDEX idx_ordini_cliente_prodotto ON ordini(id_cliente, id_prodotto);
```

Il file di indice sarà così composto:

id_cliente | id_prodotto

----

1 | 100 ✅

1 | 101 ✅

2 | 100 ✅

1 | 100 ❌ ERRORE! (Duplicato)

Ovviamente con i relativi entry indice che punta al record fisico per ogni riga del file indice (ho omesso volontariamente per evitare di occupare il foglio).

Creazione di un Indice Composito per Velocizzare **EXISTS**.

La query **EXISTS** serve per verificare se almeno una riga esiste in una sottoselezione.

1. CREATE INDEX idx_ordini_cliente_data ON ordini(id_cliente, data_ordine);

[Indice Ordinato per id_cliente, data_ordine]

(1, '2024-01-10') → RID 100

(1, '2024-02-15') → RID 120

(2, '2024-01-10') → RID 150

(3, '2024-01-12') → RID 180

---

2. SELECT *

FROM clienti c

WHERE EXISTS (

SELECT 1 FROM ordini o

WHERE o.id_cliente = c.id

AND o.data_ordine = '2024-01-10'

);

Ora sfrutta l'Indice Composito!

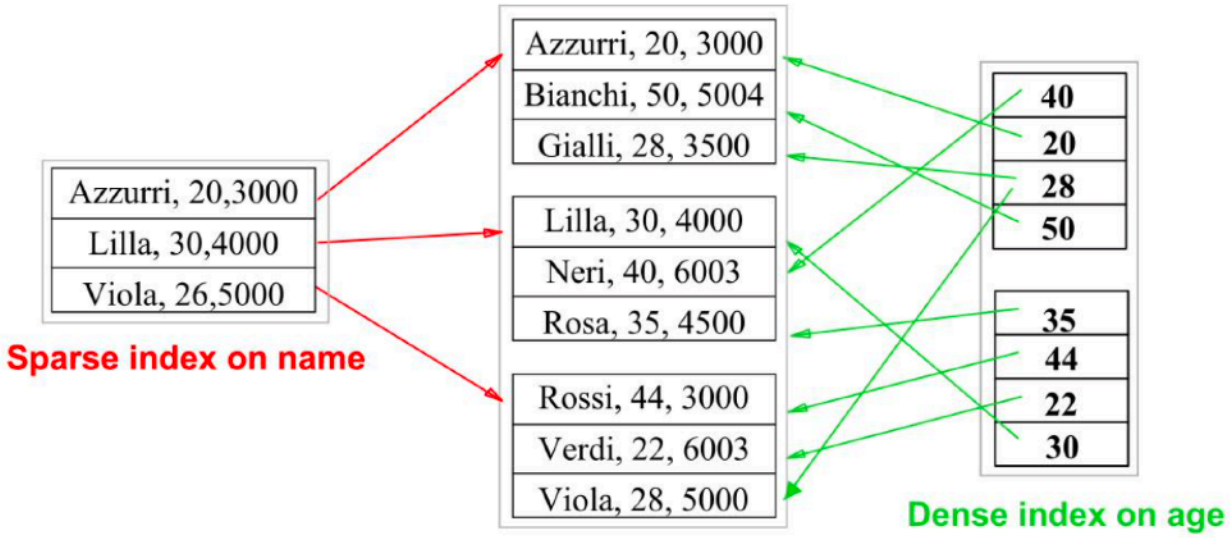
Ora il DBMS può trovare immediatamente `id_cliente = 1` e `data_ordine = '2024-01-10'` senza dover scansionare altri record!

Cosa Sono gli Indici Densi e Sparsi?

Indice Denso → Contiene una entry per ogni record nel file dati.

Indice Sparso → Contiene solo una entry per blocco di dati, non per ogni record, riducendo lo spazio occupato.

In generale, gli **Indici Densi sono più veloci**, ma gli **Indici Sparsi sono più compatti**!



Usa un Indice Denso se:

✅ La colonna ha valori molto variabili (es. `email`, `codice_fiscale`).
✅ Le ricerche devono essere **molto veloci**.

Usa un Indice Sparso se:

✅ La tabella è molto grande e **vuoi risparmiare spazio**.
✅ La chiave di ricerca è **spesso ordinata** (es. `ID`, `data_ordine`).

Un Indice Sparso NON è un Indice Composito con duplicati.

L’Indice Sparso viene creato automaticamente su un Indice Clustered.

CREATE CLUSTERED INDEX idx_clienti_id ON clienti(id);

L’indice è sparso perché contiene solo una entry per ogni blocco di dati, non per ogni record!

Il DBMS memorizza solo alcune entry di riferimento, riducendo lo spazio occupato.

[**Indice Sparso su ID**] [**File Dati (ordinato per ID)**]

ID 1 → Offset 1000 → [ID 1, ID 2, ID 3]

ID 4 → Offset 1600 → [ID 4, ID 5, ID 6]

ID 7 → Offset 2200 → [ID 7, ID 8, ID 9]

Come Garantire che un Indice sia Sparso?

Creare un Indice Clustered sulla colonna che vuoi rendere sparsa.
Se la tabella è già Clustered su un'altra colonna, tutti gli indici secondari saranno DENSI!
Un Indice Sparso si ottiene automaticamente quando si crea un Indice Clustered.
Gli Indici Clustered su colonne già ordinate (es. **ID**, **data**) sono quasi sempre SPARSI.

Quando si Usa un Indice Sparso?

Quando i dati sono già ordinati in un Indice Clustered (es. **ID**, **data**).

Per impostazione predefinita, la Primary Key crea un Indice Clustered (se non specificato diversamente).

INDICE DENSO SULLA PK:

[Indice Denso su ID] [File Dati]

ID 1 → Offset 100

ID 2 → Offset 120

ID 3 → Offset 140

ID 4 → Offset 160

ID 5 → Offset 180

Se i dati non sono ordinati, il DBMS non può sapere in quale blocco cercare un valore specifico.

Un Indice Sparso, che ha solo una entry per blocco, non sarebbe efficace, perché il DBMS non saprebbe in quale blocco si trova il record esatto.

L'Indice Denso invece contiene una entry per ogni record, permettendo un accesso diretto.

Quale Tipo di Indice Adotta il DBMS?

Se la tabella ha un Indice Clustered, i dati sono ordinati fisicamente, quindi il DBMS può usare un **Indice Sparso**.

Se l'indice è Non-Clustered, i dati nel file non sono ordinati, quindi il DBMS deve avere una entry per ogni record. In questo caso, il DBMS usa un **Indice Denso**.

Creiamo un Indice Non-Clustered su **nome:**

CREATE INDEX idx_clienti_nome ON clienti(nome);

Il DBMS usa un Indice Denso perché i dati non sono ordinati e ha bisogno di una entry per ogni record.

[Indice Denso su Nome] [File Dati (Non Ordinato)]

Nome = Azzurri → Offset 3500

Nome = Bianchi → Offset 6004

Nome = Gialli → Offset 5002

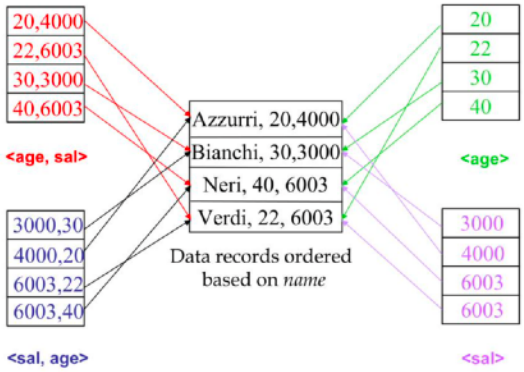
Nome = Lilla → Offset 4020

Nome = Viola → Offset 5200

Dal punto di vista grafico indici compositi o semplici.

Vale tutto quello che fin ora abbiamo detto.

L'indice ordinato per età è unclastered, punta a record differenti.



INDICIZZAZIONE BASATA SU ALBERI.

Indici e organizzazioni degli indici.

Da ciascun nodo escono (possono uscire) più rami.

Nota: l'albero è scritto su disco.

Il B+-Tree è una **struttura ad albero bilanciato** utilizzata nei database per velocizzare ricerche, inserimenti e cancellazioni. Il B+-Tree è un albero M-ario bilanciato, con nodi interni e nodi foglia.

Mantiene i dati ordinati e permette un accesso efficiente ai record, specialmente per range query e operazioni su grandi quantità di dati.

Ogni nodo può contenere più chiavi e puntatori ai figli.

I nodi foglia sono collegati tra loro in una lista concatenata per facilitare le range query.

Ogni nodo descrive una pagina perché lo carico in RAM con una singola operazione di I/O.

- **Le pagine con le entry dati sono le foglie dell'albero.**

Questa è un'indicizzazione ad albero.

Come funziona?

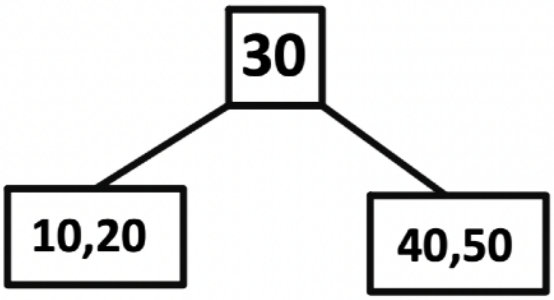
Immaginiamo di avere una tabella **clienti** in un database MySQL, PostgreSQL o SQL Server.
Vogliamo ottimizzare le ricerche per **id_cliente**, quindi creiamo un Indice Clustered basato su B+-Tree.

```
CREATE TABLE clienti (
id_cliente INT PRIMARY KEY,
-- Creazione automatica di un B+-Tree su id_cliente
nome VARCHAR(50),
città VARCHAR(50)
);
```

Supponiamo di inserire:

```
INSERT INTO clienti (id_cliente, nome, città) VALUES
(10, 'Mario Rossi', 'Roma'),
(20, 'Luca Bianchi', 'Milano'),
(30, 'Giulia Verdi', 'Napoli'),
(40, 'Francesca Neri', 'Torino'),
(50, 'Andrea Gialli', 'Bologna');
```

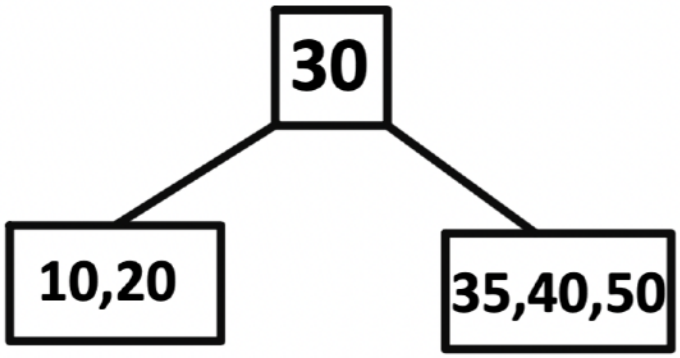
Struttura del B+-Tree con M=4 dopo l’inserimento:



Se inserisco:

INSERT INTO clienti (id_cliente, nome, città) **VALUES** (35, 'Elisa Marrone', ‘Firenze');

Struttura del B+-Tree aggiornata:



M è l'ordine dell’albero, ovvero il numero massimo di puntatori ai figli che un nodo può avere.
Di conseguenza, ogni nodo può contenere fino a **M – 1** chiavi e **M** puntatori ai figli.

Se M è grande, il B+-Tree avrà meno livelli e quindi meno accessi I/O.

Se M è piccolo, il B+-Tree avrà più livelli e la ricerca sarà leggermente più lenta.

La profondità di un B-tree deve essere mantenuta al minimo possibile per garantire efficienza nelle operazioni di ricerca, inserimento e cancellazione. Questo è uno degli obiettivi principali di questa struttura dati.

Se la chiave da cancellare k si trova nel nodo x ed x è una foglia allora è sufficiente eliminare la chiave k senza ulteriori operazioni (caso banale).

Ogni nodo dovrebbe essere **il più pieno possibile**, evitando di avere troppi nodi poco popolati che causano un albero più profondo del necessario.

Nessun puntatore nei nodi intermedi può essere NULL, perché ogni chiave in un nodo intermedio deve sempre puntare a un sottoalbero valido.

Per capire il funzionamento facciamo un esempio di costruzione dell'albero.

PROVA D'ESAME DEL 5_07_2024.

Si consideri un B-tree con nodi intermedi che contengono tre chiavi e quattro puntatori e foglie con tre chiavi, in cui vengano inserite chiavi (a partire dall'albero vuoto) nel seguente ordine:
41, 57, 11, 32, 20, 27, 28, 31, 34, 35, 36

- Mostrare l'albero dopo l'inserimento di quattro chiavi, di otto chiavi e alla fine.
- Mostrare poi l'albero dopo l'eliminazione della chiave 20 dall'ultimo albero ottenuto

Svolgimento.

Dobbiamo inserire: **41, 57, 11, 32.**
Ciascun nodo da 3 elementi ha 4 puntatori.

Come procediamo?

Conviene partire dalla fine.

41, 57, 11, 32, 20, 27, 28, 31, 34, 35, 36

Ordiniamoli:

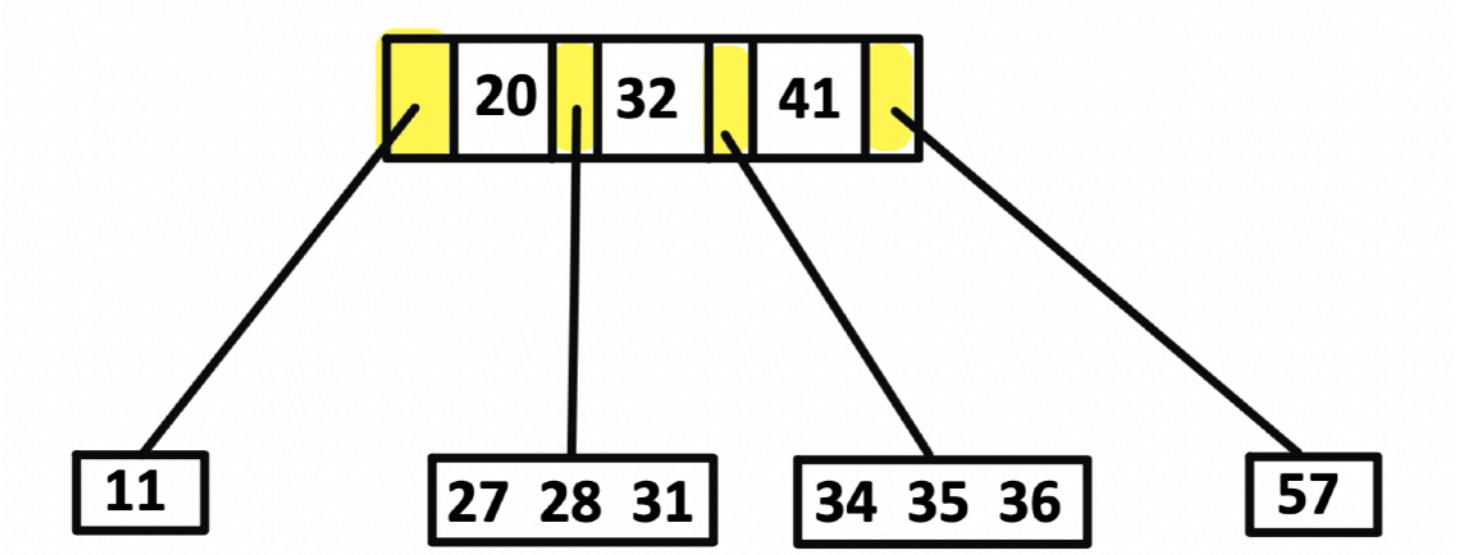
11,20,27,28,31,32,34,35,36,41,57.

Vogliamo la profondità minima.

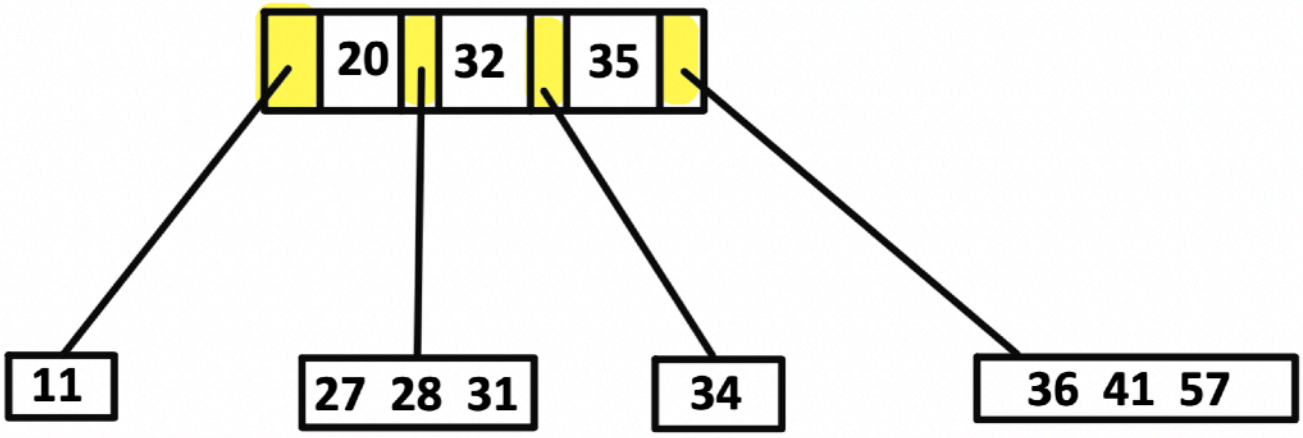
Prima di 11 non si può andare, quindi evitiamolo di metterlo come radice. Scegliamo radice 20.
Prima di 20 c'è 11.
Ora abbiamo da scegliere il numero successivo a 20, contando 3+1

posizioni, scelgo 32 e in mezzo avrà i nodi saltati (27,28,31).
Ora dopo 32 conto 3+1 e metto 41 come successore.
Dopo 41 nel livello inferiore avrò 57.

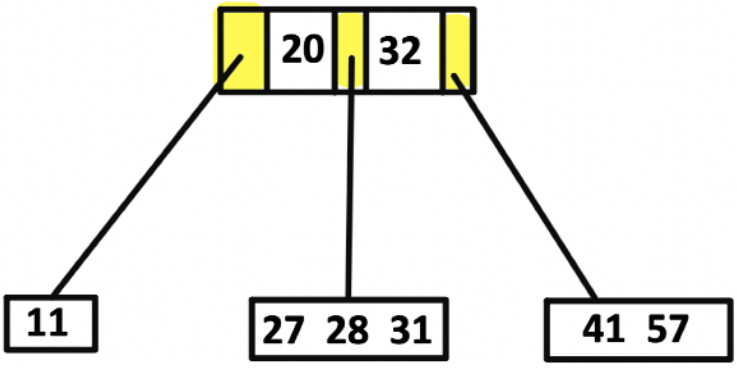
Questo è l'albero finale.



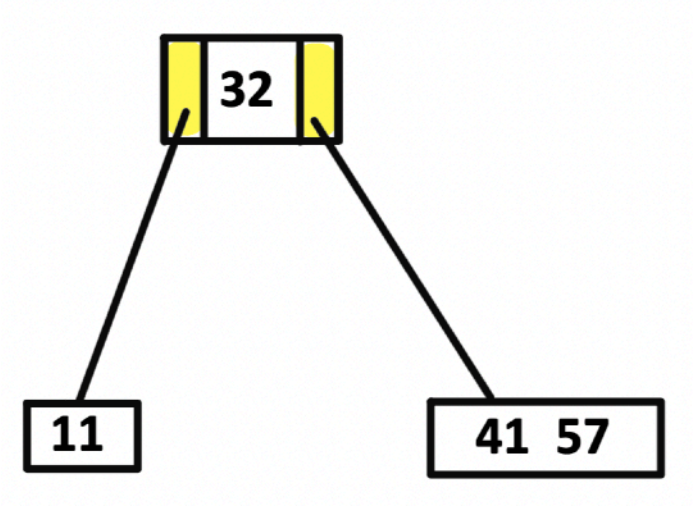
Ora devo rimuovere 36,35,34: questo mi fa diventare un puntatore NULL, quindi ricostruisco l'albero finale diversamente.



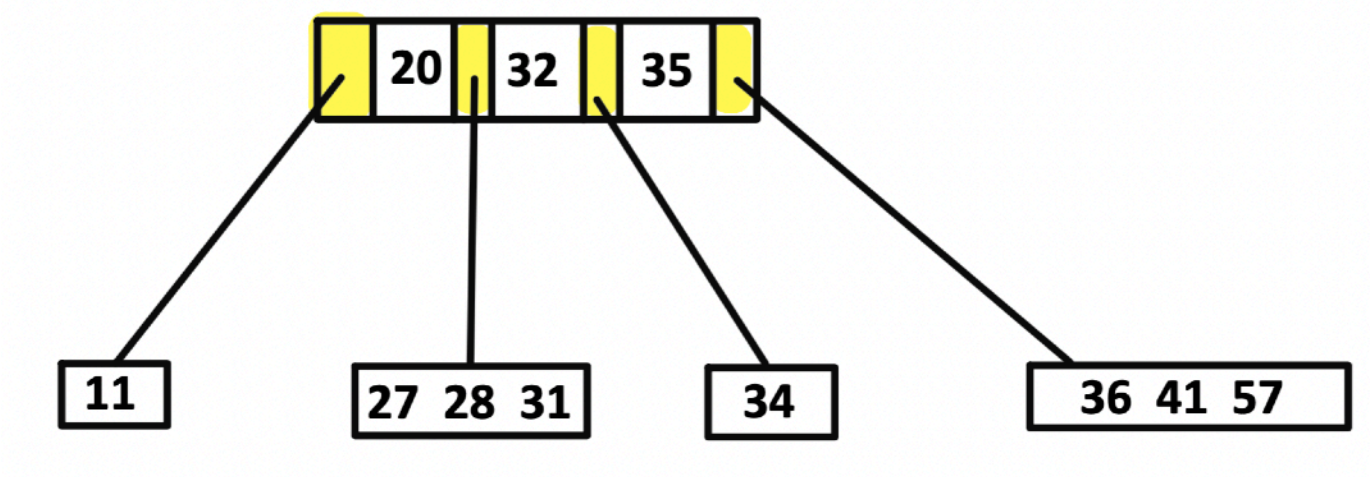
Rimuovo 36,35,34.



Ora rimuovo 31,28,27,20.



Ora eliminiamo la chiave 20 dall'ultimo albero.



Chiave 20 eliminata: