Gestione delle stringhe: Tries

Nicolas Nucifora - X81000452

Università degli studi di Catania
Dipartimento di Matematica e Informatica

Algoritmi e laboratorio - Prof. D.F.Santamaria

A.A 2020/2021

Indice

Prefazione

Lo scopo della relaione è analizzare, sotto ogni aspetto, la struttura dati per la gestione delle stringhe chiamata "Trie". Verrà data una sostanziosa descrizione della struttura dati: com'è nata, quali problemi risolve, la sua struttura, come funziona, i pro e i contro del loro utilizzo, e cosi via. Successivamente, si analizzerà lo pseudocodice per l'implementazione delle operazioni effettuabili su un Trie. Infine, il diagramma UML dell'approccio scelto per l'implementazione. La relazione verrà accompagnata dal codice, scritto in C++, che implementa la struttura, proponendo la risoluzione di uno dei problemi descritti.

1 Trie: cos'è?

Il problema di immagazzinare un insieme di parole, come un dizionario con le relative descrizioni o l'insieme delle parole più ricercate su Google, utilizzando poco spazio e la cui ricerca sia molto veloce è uno dei problemi più importanti nel mondo informatico.
Esistono diverse soluzioni, ognuno con i suoi pro e i suoi contro in termini di complessità spaziale e temporale. Un esempio sono le tabelle hash e i dizionari.
Ma esiste una specifica struttura dati, nata appositamente per risolvere il problema della rappresentazione di un insieme di parole: la struttura dati **Trie**
Il loro utilizzo è stato suggerito per la prima volta dal francese **René de la Briandais** nel 1959.
La parola "Trie" deriva da re**trie**val (recupero), in quanto permette di recuperare una parola in pochissimo tempo.

Sostanzialmente, un Trie è una struttura ad albero n-ario (il nome trie serve inoltre a distinguerlo da altre strutture ad albero). Ciò che lo distingue dalle altre strutture ad albero è il fatto che ogni nodo memorizza dentro di se l'intero alfabeto che vogliamo rappresentare (per esempio, 26 lettere per l'alfabeto anglosassone). Strutturando i nodi in modo particolare, è possibile recuperare (re**trie**ve) una parola/stringa attraversando verso il basso un ramo dell'albero.

Ma che forma ha un Trie?

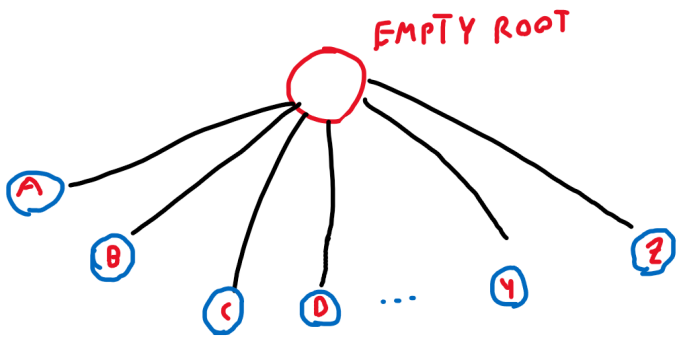

Figura 1: Forma base di un Trie

Ogni Trie inizia con un nodo radice vuoto. Esso mantiene dei riferimenti (o link) ai suoi nodi figli, uno per ogni possibile lettera dell'alfabeto. A loro volta, ogni nodo figlio mantiene ulteriori riferimenti alle lettere dell'alfabeto. Questo ci permette di stabilire che **la dimensione di un Trie è direttamente proporzionale alla grandezza dell'alfabeto che vogliamo rappresentare**.
Per esempio, se l'alfabeto rappresentato è quello anglosassone, ogni nodo mantiene 26 riferimenti alle 26 lettere dell'alfabeto. Mentre, se l'alfabeto rappresentato è **Khmer**, il numero di riferimenti a nodi figlo che un unico nodo manterrà sarà 74. Quindi, un Trie potrà essere molto piccolo o molto grande. E' possibile mantenere i riferimenti ai nodi figlo in un Trie in diversi modi, ognuno con i suoi pro e i suoi contro in termini di efficienza spaziale o temporale. Ma per spiegare più nel dettaglio come è strutturato un Trie e come è possibile mantenere e recuperare una stringa da essa, verrà utilizzato l'alfabeto anglosassone, mantenuto attraverso l'utilizzo di un array, il metodo più semplice per implementarlo.

1.1 Struttura di un Trie

Ogni nodo, compresa la radice, mantiene due informazioni:

- Un valore: che può essere **NULL**
- Un array di riferimenti ai nodi figlio: che potrebbero essere **NULL**

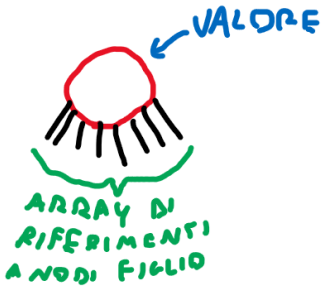

Figura 2: Nodo di un Trie

Quando un Trie viene creato, all'inizio sarà presente solamente il nodo radice. Esso rappresenta la **stringa vuota**. Il suo valore iniziale sarà **NULL**. Inoltre, manterrà un array di 26 link ai suoi nodi figlio, che inizialmente avranno valore **NULL**. Ogni qualvolta verranno inserite delle stringhe, tali riferimenti verranno riempiti con dei puntatori alle successive lettere che compongono la parola. L'inserimento e come il Trie prende forma verrà spiegato in seguito.
Il motivo della scelta di un array come struttura dati per il mantenimento dei riferimenti è dovuto alla sua semplicità e soprattutto per la velocità di accesso ad ogni cella (in tempo costante). Infatti, se l'alfabeto viene rappresentato in ordine, possiamo associare ad ogni indice dell'array una lettera dell'alfabeto: **0** per la lettera **A**, **1** per la lettera **B**, e cosi via, fino ad arrivare a **25** per la lettera **Z**.

Successivamente, verranno messi a confronto **Trie** e **Tabella hash** per mostrare perchè è conveniente utilizzare la prima per mantenere un set di parole.
Nel paragrafo successivo, verrà mostrato come riempire un Trie e come funziona la ricerca di una parola.

1.2 Costruire un Trie

Nella figura seguente è mostrato un esempio di un Trie, riempito con le seguenti parole: **Algoritmo, Amazon, Astronauta, Asterix, Sogno**.

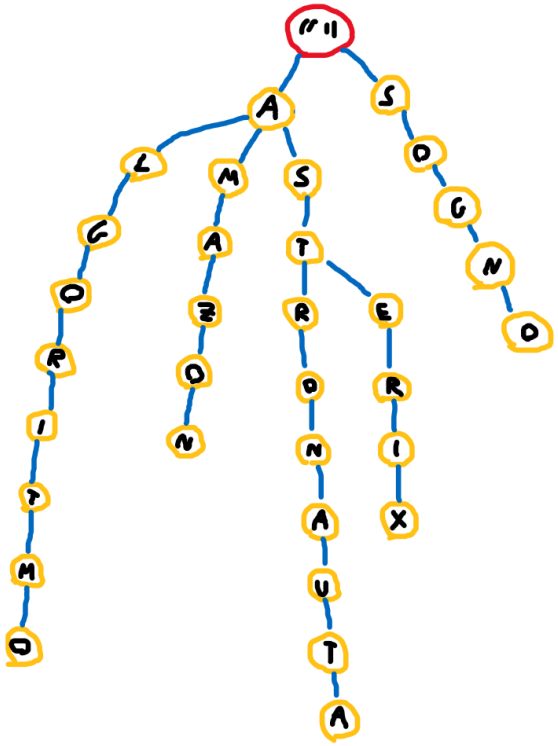

Figura 3: Esempio di un Trie riempito

Per semplicità, non sono stati mostrati i nodi NULL, ovverro che non contengono riferimenti ad altre lettere.

Come è gia stato spiegato, il nodo radice rappresenta la stringa vuota. Dei suoi 26 riferimenti, l'indice **0** mantiene un puntatore ad un nodo figlio, in questo caso il puntatore al nodo **A**, mentre la cella di indice **19** mantiene un riferimento al nodo **S**. Tutti gli altri nodi sono **NULL**.
Ci sono in totale cinque rami, uno per ogni parola rappresentata. Tutte le parole, tranne **Sogno**, condividono il nodo **A**, mentre altre parole condividono più nodi. In questo esempio, **Astronauta** e **Asterix** condividono i nodi **A**, **S** e **T**.

1.3 Operazioni

Ma come si inserisce/cerca/elimina una parola?

1.3.1 Inserimento

Vogliamo inserire una nuova parola nella struttura:
"Amazzone".
Per farlo, dobbiamo:

- Verificare che la parola non esita già nel Trie, esplorando i nodi.
- Se non esiste, riempire i riferimenti ai nodi figlio dove le lettere della parola dovrebbero stare.

Ma come facciamo a sapere se una parola esiste o meno?
Abbiamo detto che ogni nodo, oltre ai riferimenti ai nodi figlio, mantiene anche un valore. Tale valore può avere qualunque forma: un valore alfanumerico, una struttura dati, o semplicemente un booleano. Quel valore, se non è **NULL**, rappresenta **la fine della stringa**. Quindi, se una parola esiste, la sua ultima lettera, ovvero il suo ultimo nodo, avrà un valore diverso da **NULL**.
Prendendo come esempio la parola **Amazzone**, descriviamo passo per passo l'inserimento di una nuova parola:

- Partendo dalla radice, controlliamo se il riferimento alla lettera **A** è presente, essendo la prima lettera che compone la parola **Amazzone**.
- Il riferimento al nodo **A** è diverso da **NULL**, quindi è presente. Scendiamo verso il basso, fino a raggiungere il primo riferimento al nodo **Z**.
- La prossima lettera da controllare è la seconda **Z** di **Amazzone**. Il primo nodo **Z** della parola **Amazon** non ha un riferimento ad una seconda **Z**, infatti è **NULL**. Possiamo creare un nuovo nodo e inserire il suo riferimento alla cella 25 dell'array mantenuto dalla prima lettera **Z**.
- Incontreremo ulteriori link **NULL** anche per le successive lettere **O**, **N** ed **E**. Non faremo altro che creare nuovi nodi e posizionare i riferimenti rispettivamente nelle celle: **15** (secondo nodo **Z**), **14** (nodo **O**), **5** (nodo **N**).
- Infine, essendo **E** l'ultima lettera della parola **Amazzone**, per indicare che questo nodo rappresenta la fine della parola, inseriamo un valore diverso da NULL (in questo esempio **25**). Se avessimo utilizzato un booleano, bastava settarlo a **TRUE**.

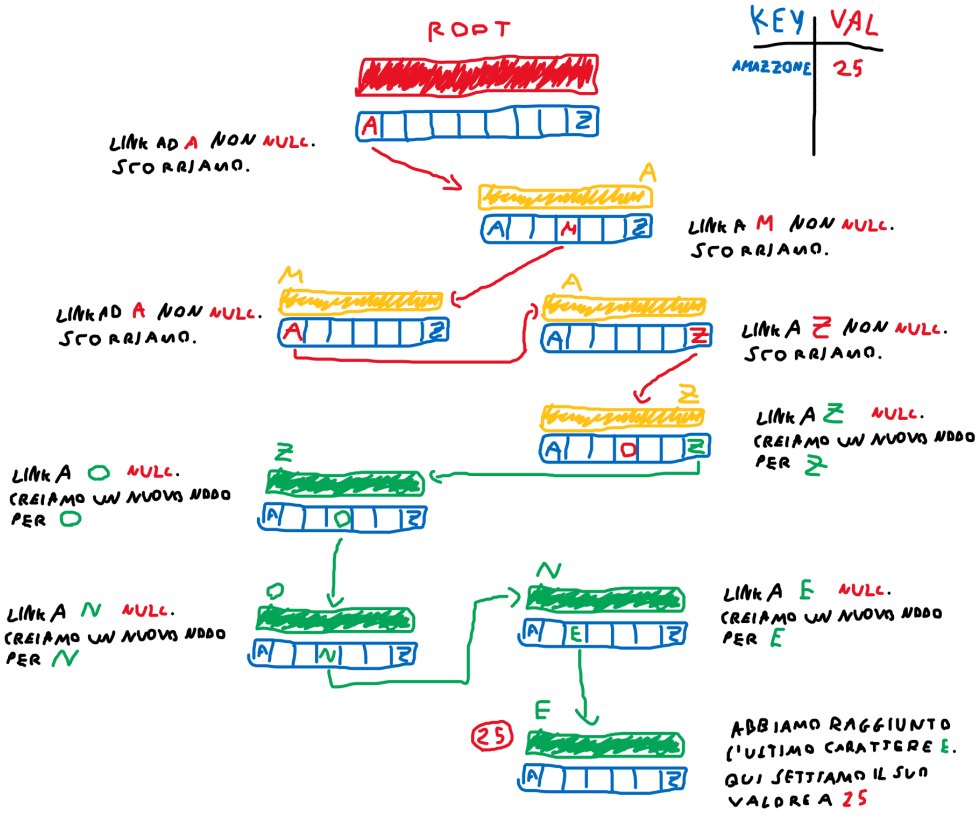

Figura 4: Esempio di inserimento "Amazzone"

E se volessimo inserire il prefisso **Astro** della parola **Astronauta**? I nodi **A - S - T - R - O** sono gia stati creati grazie all'inserimento della parola **Astronauta**. Quindi, per poter inserire **Astro** nel set delle parole del Trie, basta percorrere il ramo che compone la parola **Astronauta**. Ovviamente, non sarà necessario percorrere tutto il ramo. Infatti, ci fermeremo prima. I nodi attraversati, per la precisione, saranno: **A - S - T - R - O**. Raggiunto il nodo **O**, basterà settare il suo valore con uno diverso da **NULL**, rendendo **Astro** una parola del Trie.

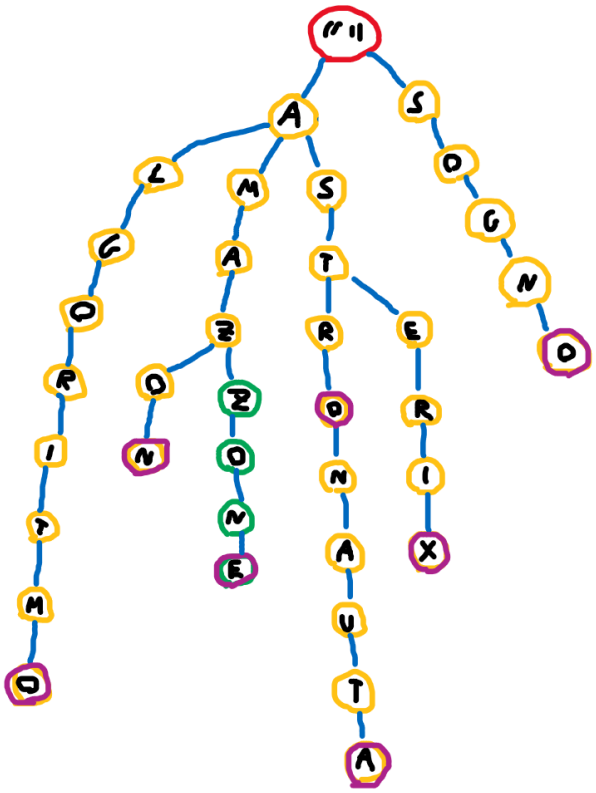

Figura 5: Trie dopo l'inserimento di "Amazzone" e del prefix "Astro"

1.3.2 Ricerca

In questo paragrafo, verrà mostrato quanto è semplice e veloce recuperare una parola da un Trie, e perché è necessario mantenere un valore (o un booleano) come segnale di fine stringa, e come verrà utilizzato.
Supponendo che la parola da recuperare sia la nuova parola inserita, ovvero **Amazzone**, descriviamo i passaggi per ricercare la stringa:

- A partire dalla radice, che ricordiamo essere la stringa vuota, cerchiamo tra i suoi link se è presente un riferimento alla lettera **A**.
- Non essendo **NULL**, sappiamo che il Trie contiene la lettera **A**. Continuando attraverso questo ramo, scendiamo giù, scandendo tutte le lettere che compongono la parola per trovare i link alle lettere successive. Per la precisione, verranno attraversati i seguenti nodi: **A - M - A - Z - Z - O - N - E**.
- Raggiunto l'ultimo nodo della parola (**E**), controlliamo il contenuto del valore da esso mantenuto. Esso contiene **25** (diverso da **NULL**), che rappresenta la fine della stringa.

  Abbiamo recuperato (re**trie**ved) la parola **Amazzone**! Questo equivale ad un **Search Hit**, in quanto possiamo ritornare un valore per la chiave ricercata.

Ma cosa succederebbe se volessimo ricercare una parola che non esiste, come una sottostringa di **Amazzone**, per esempio **"Amaz"**? Come facciamo a capire che tale parola non è presente nel Trie?

Seguendo il procedimento della ricerca di una parola in un Trie:

- Sempre a partire dalla radice, scorriamo i link **A - M - A - Z**.
- Raggiunto l'ultimo nodo della parola ricercata (**Z**), controlliamo il suo valore.
- Ci accorgiamo che il suo valore è **NULL**. Ciò equivale ad un **Search Miss**, in quanto non è possibile ottenere un valore dalla parola ricercata. Questo significa che la parola **Amaz** non esiste nel nostro Trie.

Osservazione
Possiamo notare che, se volessimo aggiungere la parola **Amaz** nel nostro Trie, basterebbe scorrere fino all'ultimo nodo **Z** e settare un valore ad esso! Non sarà necessario allocare altro spazio per i nodi, in quanto i nodi **A - M - A - Z** sono gia stati creati nell'inserimento delle due parole **Amazon** e **Amazzone**.

Figura 6: Esempio ricerca "Amazzone"

1.3.3 Cancellazione

L'operazione di cancellazione è molto semplice e veloce. E' necessario verificare certe condizioni e il gioco è fatto!
I casi in cui si incorre durante la cancellazione in un Trie sono i seguenti:

1. La chiave non è presente nel Trie. In questo caso, il Trie non viene alterato.
2. La chiave è **unica**. Non sono presenti **prefissi** (chiavi che sono sotto-stringhe della chiave da cancellare); la chiave stessa da eliminare non è prefisso di una chiave più lunga. **Possiamo cancellare tutti i nodi**.
3. La chiave è **prefisso** di una chiave più lunga. Si setta semplicemente il suo valore a **NULL**, rimuovendola logicamente.
4. La chiave è presente nel Trie, ma è presente almeno una parola come **prefisso** della chiave da cancellare. Vengono cancellati i nodi a partire dalla fine della chiave vittima, fino ad arrivare al primo nodo foglia della chiave **prefisso** più lunga.

Analizziamo ogni caso di una cancellazione utilizzando le seguenti parole: **Algo**, **Sogno**, **Astro**, **Astronauta**.

Caso 1: Algo

Questa parola permette di analizzare il primo caso: la chiave non è presente nel Trie. Effettuiamo la ricerca della chiave. Attraversiamo i seguenti nodi: **A - L - G - O**, tutti presenti grazie alla parola pre-esistente **Algoritmo**. Arrivato all'ultimo nodo **O**, controlliamo il suo valore. Quest'ultimo è **NULL**. Ciò significa che la chiave **Algo** non è presente nel Trie. Usciamo dalla procedura, senza alterare la struttura.

Caso 2: Sogno

Siamo nel secondo caso: la chiave è unica, non sono presenti prefissi e la chiave stessa non è un prefisso. Ricerchiamo la chiave. Percorriamo i seguenti nodi: **S - O - G - N - O**. L'ultimo nodo ha un valore diverso da **NULL**, quindi è presente nel Trie. Prima di tutto, settiamo il valore a **NULL**. Dopodiché, controlliamo se i suoi riferimenti sono anche essi a **NULL**. Se sono tutti vuoti, vuol dire che non sono presenti altre parole o ramificazioni sotto la chiave, e sarà possibile cancellare i nodi. Incominciamo ad eliminare un nodo alla volta (in **Bottom-up**), fin quando non incontriamo un nodo il cui valore è diverso da **NULL** (che indica che è presente un **prefisso** della chiave che vogliamo cancellare). Essendo nel secondo caso, dove la chiave da eliminare è unica, risalendo il ramo non troveremo alcun prefisso, e tutti i nodi della parola **Sogno** verranno eliminati.

Caso 3: Astro

Essa è prefisso della chiave **Astronauta**. Come nei casi precedenti, effettuiamo la ricerca del valore di **Astro**. Essendo diverso da **NULL**, esso è presente nella struttura. Controlliamo i riferimenti nell'ultimo nodo **O**, e ci accorgiamo che è presente un riferimento ad un nodo rappresentante la lettera **N**, appartentente alla parola **Astronauta**. In questo caso, non possiamo cancellare alcun nodo, altrimenti non sarà più possibile recuperare la parola **Astronauta**. L'unico modo per rimuovere la parola **Astro** dal Trie è farlo logicamente, settando il suo valore a **NULL**.

Caso 4: Astronauta

L'ultimo caso è dovuto alla presenza di un prefisso all'interno della parola **Astronauta**, ovvero **Astro**. Dovremo stare attenti a non cancellare alcun nodo che componga quest'ultimo. Ipotizziamo che la cancellazione di **Astronauta** venga effettuata prima della cancellazione di **Astro**. Effettuiamo la ricerca del valore di **Astronauta**. Raggiunto l'ultimo nodo **A**, verifichiamo che non ci siano altri riferimenti. Essi sono tutti vuoti, quindi possiamo procedere in bottom-up alla rimozione dei nodi. Ma essendoci il prefisso **Astro** di mezzo, la procedura dovrà fermarsi una volta scoperto che il nodo **O** presenta un valore diverso da **NULL**.

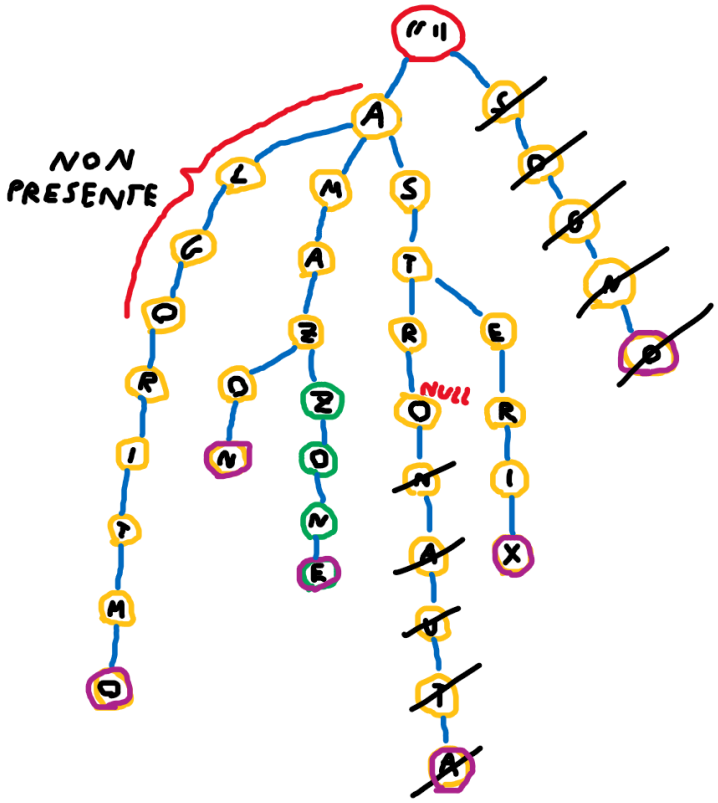

Figura 7: Trie dopo le cancellazioni

1.4 Complessità

Analizziamo le complessità di creazione di un Trie, insieme a quelle delle operazioni di inserimento, ricerca e cancellazione.

Operazione	**Caso pessimo**
Creazione	O(m * n)
Inserimento	O(k)
Ricerca	O(k)
Cancellazione	O(k)

Tabella 1: Complessità Trie

- **Creazione**: la complessità spaziale di un Trie è direttamente proporzionale al numero di parole/chiavi che il Trie contiene. Sia **m** la lunghezza della parola più lunga nel Trie, e sia **n** il numero di parole contenute in esso, la complessità per la creazione di un Trie è: **O(m * n)**. Questo perché, se il Trie è vuoto, dovremo inizializzare un nodo per ogni lettera della stringa da inserire, in tempo costante. Questa operazione è ripetuta nel caso peggiore **m** volte, dove **m** è la lunghezza della stringa più lunga. Il tutto, ripetuto **n** volte, ovvero il numero di parole da inserire nel Trie.
- **Inserimento, ricerca, cancellazione**: sia **k** la lunghezza della parola considerata, la complessità temporale di ognuna delle tre operazioni è: **O(k)**.
  - **Inserimento**: nel caso peggiore, la radice non ha un riferimento alla prima lettera della parola da inserire. Verranno istanziati **k** nodi, ognuno in tempo **O(1)**. **O(k)**
  - **Ricerca**: nel caso peggiore, la stringa che stiamo cercando è la più lunga nel Trie. Se la sua lunghezza è **k**, verranno effettuati **k** accessi. **O(k)**

- **Cancellazione**: nel caso peggiore, la stringa è unica (non ha prefissi e a sua volta non è prefisso di una parola). Raggiunto il **k-esimo** nodo, ad uno ad uno vengono eliminati in Bottom-Up. **O(k)**

1.5 Vantaggi e svantaggi: Trie vs hash Table

Trie e tabelle hash sono tanto simili quanto diversi. Entrambi fanno uso di un array, ma lo fanno in modo diverso. La tabella hash usa un array insieme a liste collegate, mentre il Trie usa un array in combinazione con dei puntatori a nodi.
La differenza sostanziale è che un Trie non fa uso di **funzioni hash**. Questo perché ogni parola ha una sua ramificazione unica che permette di raggiungere il suo valore. Quindi, è impossibile che ci siano **collisioni**, come accade con le tabelle hash.
Inoltre, ogni parola è inserita in ordine alfabetico. Sarà sufficiente utilizzare gli indici dell'array che rappresentano le lettere dell'alfabeto, per accedervi.
Il lato negativo di avere una struttura semplice e veloce è che molto spazio viene sprecato. Ogni volta che viene inserita una lettera, un array viene istanziato con 26 puntatori a **NULL**, che probabilmente non verranno utilizzati del tutto.
Per fare un esempio, l'inserimento della parola inglese **"Honorificabilitudinitatibus"**, lunga 26 caratteri, istanzierà 26 (array) * 26 (lettere alfabeto anglosassone) puntatori a **NULL**. Molto probabilmente, non verranno create delle sotto-ramificazioni a partire da questa parola, quindi molti dei suoi puntatori (o tutti), non verranno utilizzati, sprecando tantissima memoria.
Nonostante questo svantaggio, ulteriori benefici favoriscono l'utilizzo di un Trie come struttura dati per mantenere un set di stringhe.

Inizialmente, ovviamente, verrà fatto tanto lavoro nell'inserimento delle prime parole, dovendo istanziare un nuovo array per ogni lettera inserita. Ma più il Trie cresce in dimensioni, meno lavoro sarà necessario per aggiungere un valore al suo interno, in quanto i nodi saranno gia stati inizializzati con i loro valori e riferimenti.
Un ulteriore vantaggio è dovuto dal fatto che la dimensione dell'alfabeto scelto rimane costante nel tempo. Nel caso dell'alfabeto anglosassone, ogni volta che vogliamo inserire una nuova lettera, sappiamo già che sarà necessario solamente controllare 26 possibili indici di un array presente in un nodo.

1.6 Quando utilizzare un Trie

La struttura **Trie** ha delle applicazioni particolari e molto interessanti. Quello più conosciuto, in quanto utilizzato tutti i giorni, è l'**auto-completamento**. Utilizzando il motore di ricerca Google, inserendo due/tre lettere, verranno proposte diverse parole che iniziano per quelle lettere che abbiamo inserito. Questo perché, una volta inserite le lettere, molti rami vengono tagliati, riducendo la dimensione dell'albero delle possibili parole, permettendo una facile enumerazione di quest'ultime. Ogni parola, come è gia stato descritto, può mantenere dei valori. Essi, oltre a rappresentare la fine della stringa, potrebbero essere scelti per indicare la popolarità di una parola, in modo tale che venga proposta una parola rispetto ad un'altra.
Ulteriori utilizzi di un Trie sono gli **algoritmi di matching** per l'implementazione di **spell checker**, oppure per implementare algorimti di **matching delle stringhe**.
Infine, un ulteriore utilizzo può essere quello dell'**ordinamento**. Un attraversamento **Pre-order** dell'albero darà come risultato un output di ordine crescente.

1.7 Passare ad un altro alfabeto

Fino ad ora abbiamo descritto il Trie facendo uso dell'alfabeto anglosassone. Esso è composto da 26 lettere, quindi per ogni nodo viene istanziato un array di dimensione 26. Ma se volessimo utilizzare la struttura, cosi per come l'abbiamo definita, in un'altra lingua, per esempio il **Cinese**? Sarà necessario modificare l'intero codice? Dipende dall'alfabeto. Partiamo con il presupposto che abbiamo consegnato il codice in versione inglese ad un'azienda cinese che vuole implementare un suo motore di ricerca con auto-completamento. Come è giusto che sia, tale azienda vorrebbe che le parole contenute nel Trie fossero in cinese. L'alfabeto cinese è composto da **21 lettere** (non vengono usate le lettere: j, k, w, x, y). Un numero inferiore rispetto all'alfabeto anglosassone. Sarà necessario modificare l'intera struttura per adattare il Trie all'alfabeto cinese?
Potremmo mantenere la struttura in versione inglese (con array da 26 lettere), consapevoli del fatto che 5 lettere non verranno utilizzati, in quanto l'alfabeto cinese è composto da 21 lettere (incrementando lo spreco di memoria). Ma c'è un ulteriore problema: come ottenere l'indice associato ad una determinata lettera. Nella versione anglosassone, ogni lettera (in ASCII) è rappresentabile con **un byte** (**8 bit**). Risulta semplice estrarre dal carattere il suo indice: basta sottrarre al carattere considerato il carattere **'a'** (verrà effettuata una sottrazione bit a bit, che ci da come risultato un intero di 8 bit corrispondente all'indice). per fare un esempio:

- **'c' - 'a' = 1100011 - 1100001 = 0000010 = 2**

Purtroppo questo non è possibile con l'alfabeto cinese. Ogni carattere è codificato in **UTF-8** (servono **3 byte** per rappresentare un carattere cinese). Come se non bastasse,

l'alfabeto cinese non è ordinato come quello anglosassone. In ASCII, i caratteri minuscoli, rappresentati in decimale, hanno un ordine numerico: **a = 97**, **b = 98**, **c = 99**, etc. Ciò non accade con l'alfabeto cinese. Quindi, l'approccio utilizzato con la versione inglese del Trie non è applicabile con l'alfabeto cinese.
L'unico modo per costruite un **Trie cinese** è:

- Usare la struttura dati **Map** al posto dell'array
- Ogni elemento in un **Map** è una coppia [**"carattere cinese", puntatore**].

Il funzionamento è lo stesso. Per chiarire le idee, verrà presentato un esempio con il Trie costruito in precedenza:

- Cerco la parola **Astro**.
- A partire dalla radice, controllo se dentro la **mappa** sia presente l'elemento **A**. La lettera **A** è presente nella mappa della radice, e da essa riesco a ricavare il puntatore al nodo **A**.
- Raggiunto il nodo **A**, cerco dentro la sua mappa l'elemento **S**. Lo trovo ed estraggo il puntatore al carattere successivo.
- E cosi via, fino a raggiungere il nodo **O**, dentro la quale trovo il valore di fine stringa.

Questo approccio è applicabile con l'alfabeto cinese, con le lettere cinesi al posto di quelle latine.

Osservazione
Utilizzando le **mappe**, è possibile risparmiare tantissimo spazio. Infatti, la mappa all'interno di un nodo manterrà solamente i riferimenti ai nodi figli successivi necessari, senza allocare altro spazio inutile come accade con gli array.

2 Pseudocodice

Adesso che è stata data una visione completa del Trie, del suo utilizzo e delle operazioni che si possono effettuare su di esso, in questa sezione verranno presentati gli pseudocodici delle seguenti operazioni: **GetNode** (per creare un nuovo nodo), **Insert**, **Search**, **Delete**.

2.1 GetNode

Ricordiamo che ogni nodo (corrispondente ad una lettera di una word) mantiene un array di riferimenti a nodi figlio, corrispondenti alle lettere successive che compongono la parola. Se una lettera (nodo) di una parola che sta per essere inserita non è presente, è necessario crearlo con **GetNode**:

Algorithm 1: Creazione di un nuovo nodo

```
Function getNode()
node ← new TrieNode;
node.value ← NULL;
for i ← 0 to ALPHABET_SIZE do
    node.children[i] ← NULL;
end
return node;
```

2.2 Insert

Partendo dalla radice, attraversiamo l'abero verso il basso, creando solamente i nodi mancanti. Raggiunto l'ultimo nodo, settiamo il suo valore diversamente da **NULL**.

Algorithm 2: Inserimento di una nuova parola

```
Procedure Insert(trie, key, value)
node ← trie.root;
for i ← 0 to key.length do
    index ← key[i] − 'a';
    if !node.children[index] then
        node.children[index] ← GetNode();
    end
    node ← node.children[index];
end
node.value ← value;
```

2.3 Search

La funzione di ricerca è molto simile all'inserimento. A partire dalla radice, attraversiamo l'albero verso il basso. Se durante la discesa incontriamo un riferimento a **NULL** ancor prima di aver scansionato tutta la word che intendiamo cercare, significa che essa non è presente nel Trie. Se raggiungiamo l'ultimo nodo che compone la stringa, controlliamo se il suo valore è diverso da **NULL**. Se si, possiamo dire che la parola è presente nel Trie, altrimenti torniamo esito negativo.

Algorithm 3: Ricerca di una parola

```
Function Search(trie, key)
node ← trie.root;
for i ← 0 to key.length do
    index ← key[i] − 'a';
    if !node.children[index] then
        return FALSE;
    end
    node ← node.children[index];
end
if node ≠ NULL & node.value ≠ NULL then
    return TRUE;
else
    return FALSE;
end
```

2.4 Remove

La rimozione di una parola può essere implementata con una funzione ricorsiva. Questo perché, una volta raggiunto l'ultimo nodo componente la parola, dobbiamo risalire l'albero, eliminando un nodo alla volta (se possibile). Bisogna fare attenzione ai casi che si riscontrano durante la cancellazione, spiegati in **Pag.14**.

<u>Caso base</u>
I casi base sono due:

1. Il Trie è vuoto.
2. Abbiamo raggiunto l'ultimo carattere. In questo caso, controlliamo se il nodo contiene un valore di fine stringa

diverso da **NULL**. In caso affermativo, lo settiamo a **NULL** per rimuoverlo logicamente dal set delle parole. Infine, controlliamo se tale nodo mantiene dei riferimenti a nodi figlo. Se sono tutti vuoti, possiamo procedere all'eliminazione del nodo.

Passo ricorsivo

Ricaviamo l'indice del nodo figlio attraverso la stringa passata come parametro della funzione e chiamiamo ricorsivamente la funzione passando come parametri il Trie, la word da cancellare e la profondità dell'albero incrementata di uno, in modo tale da analizzare il carattere successivo. Dopo la chiamata ricorsiva, controlliamo se il nodo correntemente processato ha dei riferimenti a nodi figlio e se è segnato come fine stringa. Se entrambe le condizioni sono soddisfatte, vuol dire che ci sono rispettivamente altre ramificazioni che compongono altre word o che tale nodo appartiene ad un prefisso della stringa che vogliamo cancellare. In questo caso, non dobbiamo eliminare il nodo, o perderemo la possibilità di recuperare altre parole del set. Altrimenti, possiamo tranquillamente eliminarlo.

Algorithm 4: Eliminazione di una parola

```
Function Remove(root, key, depth)
if !root then
    return NULL;
end
if depth = key.size then
    if root.value ≠ NULL then
        root.value ← NULL;
    end
    if isEmpty(root) then
        Delete root;
        root ← NULL;
    end
    return root;
end
index ← key[depth] − 'a';
root.children[index] ←
 Remove(root.children[index], key, depth + 1);
if isEmpty(root) & root.value = NULL then
    Delete root;
    root ← NULL;
end
return root;
```

3 UML

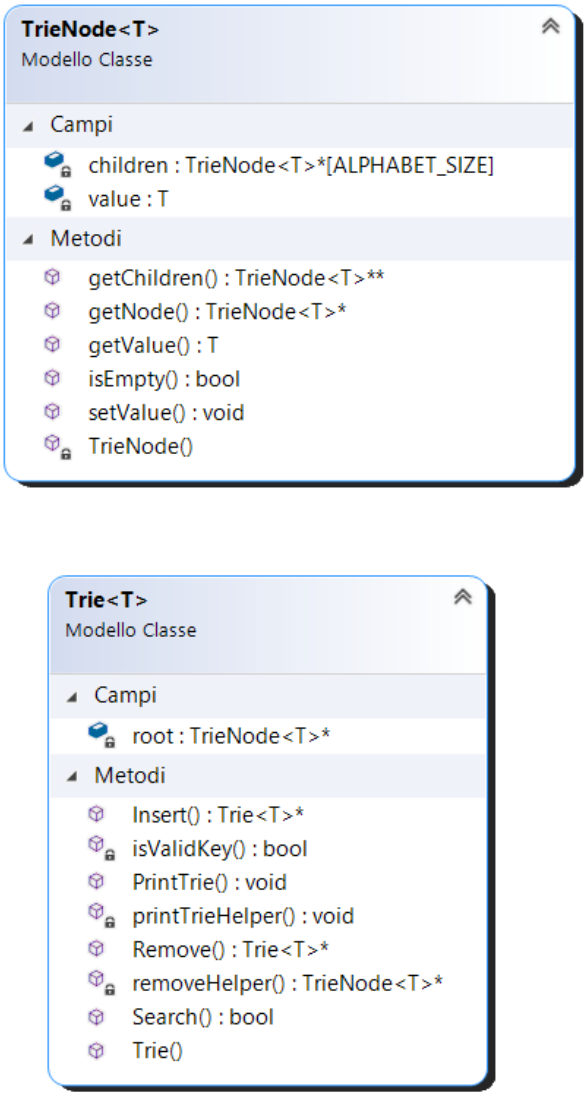

Figura 8: Diagramma UML Trie

Come è possibile notare dal diagramma UML, il Trie è una struttura dati semplice, composta da due sole classi: una classe **TrieNode** corrispondente al singolo nodo che va a comporre un Trie, e la classe **Trie** stessa, composta da tanti TrieNode.
Ho deciso di utilizzare l'approccio dell'array come contenitore di riferimenti ai nodi figlio per la sua semplicità di utilizzo attraverso gli indici ordinati. Questo grazie al fatto che l'alfabeto utilizzato è ordinato ed è possibile ricavare da ogni lettera il suo indice di posizione (**A = 0, B = 1, ... , Z = 25**). Inoltre, l'aggiornamento dell'array è semplice: dato un nodo, se vogliamo aggiungere un nuovo riferimento all'i-esima lettera componente una parola, basta assegnare all'i-esima cella del suo array di riferimenti un nuovo nodo. Se vogliamo rimuovere quel riferimento, perché stiamo rimuovendo la parola a cui appartiene quel nodo, basta cancellare il nodo (con una **delete**) e assegnare **NULL** alla stessa i-esima cella.
Ad ogni inserimento, grazie agli array, la parola viene posizionata in ordine alfabetico in automatico, senza dover fare modifiche alla struttura.

Riferimenti bibliografici

[1] Frank Pfenning, "*Lecture Notes on Tries*" https://www.cs.cmu.edu/ fp/courses/15122-f10/lectures/18-tries.pdf

[2] Alex Chumbley, Agnishom Chattopadhyay, Karleigh Moore, and 3 others contributed. "*Tries*" https://brilliant.org/wiki/tries/

[3] Alex Chumbley, Agnishom Chattopadhyay, Karleigh Moore, and 3 others contributed. "*Tries - Implementation*" https://brilliant.org/wiki/tries/

[4] Daniel Ellard "*Tries - Implementation*" http://ellard.org/dan/www/libsq/cb_1998/c06.pdf

[5] geeksforgeeks.org "*Tries - Insert, Search, Remove*" https://www.geeksforgeeks.org/trie-insert-and-search/?ref=rp https://www.geeksforgeeks.org/trie-delete/?ref=rp