La struttura dati Rope

Francesco Coppola

20 settembre 2021

Indice

1 Introduzione alla struttura dati Rope

Le stringhe sono un tipo di dato fondamentale in moltissime applicazioni. Il classico metodo per rappresentarle in memoria consiste nell'uso di array monolitici di caratteri, i quali risultano abbastanza efficienti per quanto riguarda la quantità di memoria utilizzata, ma molto meno dal punto di vista della complessità delle operazioni più comuni: inserimento, concatenazione o cancellazione, tutte procedure molto frequenti nei software che lavorano con le stringhe, sono piuttosto inefficienti se effettuate sui classici array di caratteri.

La struttura dati ***rope*** consente di ridurre il costo computazionale di tutte queste operazioni al costo di un più elevato consumo di memoria rispetto all'implementazione con array. Infatti una rope è un albero binario in cui ogni foglia contiene una stringa e la sua lunghezza, o peso, mentre ogni nodo interno rappresenta la concatenazione dei suoi figli e contiene la somma dei pesi di tutte le foglie nel suo sottoalbero sinistro. In pratica, la stringa contenuta all'interno dell'albero è la concatenazione di tutte le sue foglie da sinistra a destra.

Proprio grazie a questa struttura ad albero, le rope possono effettuare operazioni di manipolazione delle stringhe in tempi estremamente ridotti. È bene notare, però, che l'utilizzo delle rope è conveniente soltanto quando si lavora con testi molto grandi che devono essere modificati spesso. In generale una stringa piccola rappresentata mediante array avrà prestazioni molto più elevate rispetto ad una lunga; l'utilizzo delle rope consente di ottenere prestazioni equivalenti indipendentemente dalla lunghezza del testo. Le operazioni che caratterizzano le rope sono la **concatenazione** e la **suddivisione**: la prima permette di unire due stringhe di dimensione anche molto grande semplicemente rendendole figlie di una nuova radice, mentre la seconda serve a separare una stringa in due a partire da un certo carattere. Grazie all'efficienza con cui possono essere effettuate è possibile modificare in maniera particolarmente conveniente dei testi molto grandi, ad esempio

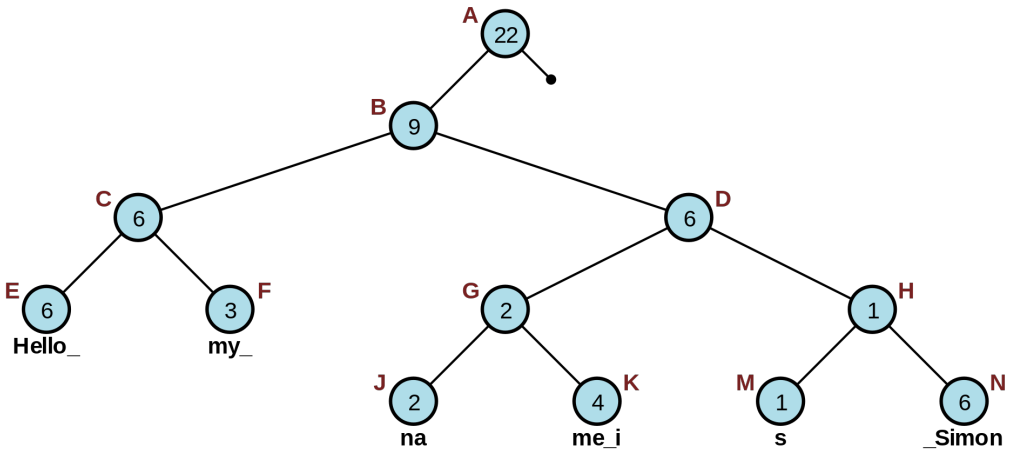

Figura 1.1: Un esempio di rope

con le classiche operazioni di taglia, copia e incolla; inoltre l'**inserimento** di caratteri in punti arbitrari del testo e la loro **cancellazione** possono essere realizzate mediante combinazioni di concatenazioni e suddivisioni. Un'altra operazione molto utilizzata quando si lavora con le stringhe, soprattutto come parte di altri algoritmi, è l'**indicizzazione**, ovvero la ricerca di un carattere all'interno della stringa a partire dalla sua posizione nella stessa. L'efficienza nell'effettuare piccole modifiche ripetute e la capacità di gestire testi particolarmente grandi rendono questa struttura dati particolarmente adatta all'utilizzo in editor di testo o client email, o più in generale per applicazioni in cui la dimensione delle stringhe deve poter scalare facilmente. Una rope è formata da due strutture dati: **Rope**, ovvero l'albero binario in cui è organizzata la struttura, e **Node**, che rappresenta il singolo nodo dell'albero.

La struttura *Rope* ha come unico attributo un puntatore *root* al nodo radice dell'albero e definisce le operazioni citate sopra, oltre a eventuali altre operazioni secondarie come la stampa.

La struttura *Node* definisce alcune procedure di supporto e presenta i seguenti attributi:

- *parent*, un puntatore al padre del nodo;
- *left*, un puntatore al figlio sinistro;
- *right*, un puntatore al figlio destro;
- *weight*, il peso del nodo, cioè il numero totale di caratteri presenti nel suo sottoalbero sinistro;

- *len*, la lunghezza della sottostringa contenuta nel nodo; è un campo utilizzato solo per rendere più efficienti alcuni algoritmi;
- *str*, una stringa primitiva rappresentata mediante array di caratteri, di dimensione limitata a un valore fisso (ad esempio la dimensione della word del calcolatore) in modo da garantire che le operazioni su di essa siano eseguite in tempo costante; poiché i caratteri in una rope sono distribuiti soltanto sulle foglie, questo campo rimane vuoto nei nodi interni.

2 Operazioni sulle Rope

Di seguito si riporta una descrizione dei principali metodi relativi alle rope.

2.1 Concatenazione

L'operazione di concatenazione di due stringhe rappresentate mediante array richiede di ricopiare il contenuto delle stringhe di partenza in un terzo array, impiegando quindi un tempo lineare, oltre all'allocazione di spazio aggiuntivo per effettuare la copia. Poiché le rope sono degli alberi binari, la loro concatenazione consiste semplicemente nella creazione di una nuova radice, alla quale verranno assegnate come figli le rope di partenza. È necessario, inoltre, calcolare il peso della nuova radice: un approccio molto comune per farlo consiste nell'uso di un'apposita funzione *WEIGHT* (Algoritmo 1), la quale permette di calcolare il peso di un nodo in maniera ricorsiva. Si osservi che la funzione non prende in input direttamente il nodo del quale si vuole calcolare il peso, bensì il suo figlio sinistro.

```
1 WEIGHT(p)
2     x = p.weight
3     if p.right ≠ NIL
4         return x + WEIGHT (p.right)
5     return x
```

Algoritmo 1: Calcolo del peso di un nodo

Sebbene l'effettiva operazione di concatenazione, ovvero la creazione del nuovo nodo radice, venga effettuata in un tempo $\mathcal{O}(1)$, il calcolo del peso ad esso associato mediante la funzione appena descritta richiede tempo $\mathcal{O}(\log n)$. Infatti, la procedura richiede di scorrere i nodi della rope a partire dalla radice e procedendo sempre verso il figlio destro, visitando un solo nodo per ogni livello dell'albero. Quindi il numero di chiamate ricorsive della funzione *WEIGHT* è al più *h-1*, dove h è l'altezza dell'albero. Chiaramente se l'albero

è bilanciato l'operazione viene effettuata in tempo logaritmico.
In realtà, però, si può notare come il peso di un determinato nodo sia dato dalla lunghezza della sottostringa radicata nel suo sottoalbero sinistro, la quale coincide con l'attributo *len*. Mantenendo, quindi, aggiornato quest'ultimo valore è possibile calcolare il peso della nuova radice in tempo costante, permettendo di svolgere l'intera operazione di concatenazione in tempo $\mathcal{O}(1)$. È possibile vedere questa versione ottimizzata nell'Algoritmo 2.

```
1  CONCATENATE(R1, R2)
2      if R1.root == NIL
3          return R2
4      if R2.root == NIL
5          return R1
6      allocate a new node newRoot
7      newRoot.left = R1.root
8      newRoot.left.parent = newRoot
9      newRoot.right = R2.root
10     newRoot.right.parent = newRoot
11     newRoot.weight = newRoot.left.len
12     newRoot.len = newRoot.left.len + newRoot.right.len
13     S.root = newRoot
14     return S
```

Algoritmo 2: Procedura di concatenazione

Nell'esempio in Figura 2.1, si può vedere la concatenazione tra le rope con radici rispettivamente il nodo C e il nodo G: viene creato un nuovo nodo radice Q e i nodi C e G vengono assegnati come suoi figli; successivamente viene calcolato il peso di Q, ovvero la lunghezza della stringa "Hello my ".

2.2 Indicizzazione

L'operazione di indicizzazione, ovvero il recupero dell'i-esimo carattere di una stringa, può essere realizzata in modo molto simile alla ricerca di una chiave in un normale albero binario di ricerca. In questo caso, però, si confronta l'indice i con il peso di un nodo per decidere se continuare la ricerca nel sottoalbero sinistro o in quello destro; se l'indice è minore del peso significa che il carattere cercato si trova nel sottoalbero sinistro, altrimenti si trova nel sottoalbero destro. In quest'ultimo caso si deve decrementare il valore di i del peso del nodo, in quanto sappiamo già che a sinistra di un nodo si trova un numero di caratteri pari al suo peso. Nell'Algoritmo 3 è presentata

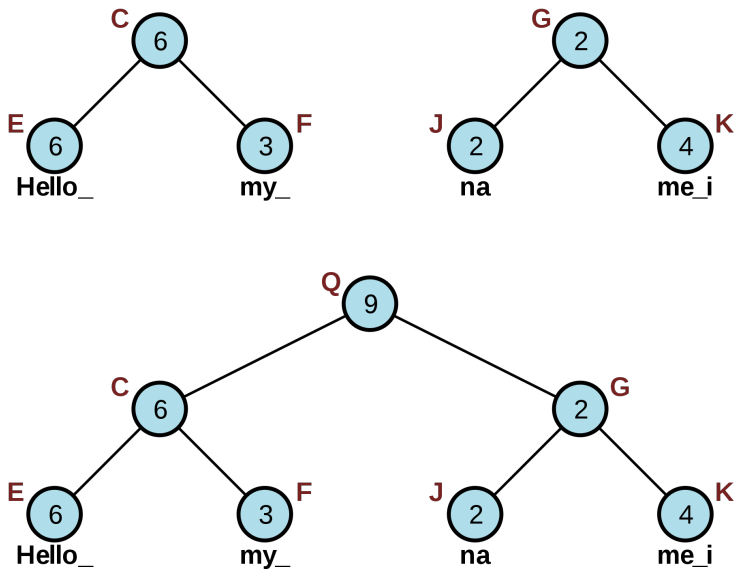

Figura 2.1: Esempio di concatenazione tra due rope

una versione ricorsiva di questa procedura, la quale si aspetta che i abbia un valore compreso tra 0 e la lunghezza della rope sulla quale viene invocata.

```
1 INDEX(p, i)
2     if p == NIL
3         return null character
4     if i < p.weight and p.left ≠ NIL
5         return INDEX(p.left, i)
6     if i ≥ p.weight and p.right ≠ NIL
7         return INDEX(p.right, i - p.weight)
8     return p.str[i]
```

Algoritmo 3: Ricerca dell'i-esimo carattere

Proprio come la ricerca in un albero di ricerca, la funzione *INDEX* richiede un tempo $\mathcal{O}(\log n)$ se l'albero è bilanciato: analogamente a quanto avviene per la funzione *WEIGHT*, viene visitato un solo nodo per ogni livello dell'albero.
In Figura 2.2 si osserva la ricerca del carattere in posizione 10, ovvero dell'undicesimo carattere. Partendo dalla radice A con peso 22, si confronta tale valore con 10 e poiché 10 è minore di 22 si prosegue verso il figlio sinistro B; infatti, essendoci 22 caratteri nel sottoalbero sinistro di A, l'undicesimo sarà necessariamente in quest'ultimo. Successivamente si confronta il peso di B

con 10: questa volta 10 è maggiore di 9 e il carattere cercato sarà, quindi, nel sottoalbero destro; si procede verso tale sottoalbero sottraendo il peso di B da 10, cioè adesso cerchiamo il carattere in posizione 1 nel sottoalbero destro di B. Si procede in questo modo fino a raggiungere una foglia: in questo caso la foglia che contiene il carattere è il nodo J e il carattere è "a".

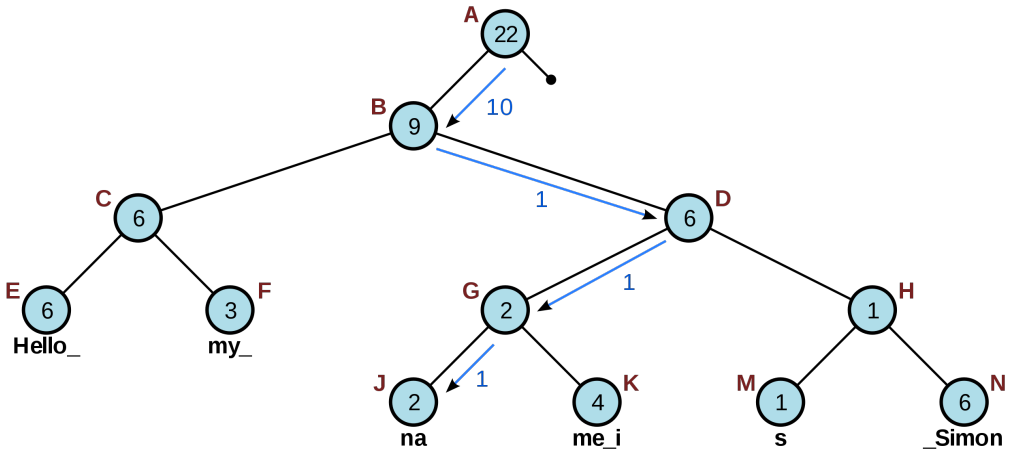

Figura 2.2: Ricerca del carattere in posizione 10

2.3 Suddivisione

L'operazione di suddivisione consente di separare la stringa S intorno al carattere in posizione i in due stringhe distinte S1 e S2, che conterranno rispettivamente i caratteri $S_1, ..., S_i$ e $S_{i+1}, ..., S_m$ della stringa di partenza. Si possono distinguere due casi:

1. Il carattere i-esimo è l'ultimo della stringa contenuta in una foglia
2. Il carattere i-esimo è in mezzo alla stringa

Il caso 2 si può ricondurre al caso 1 suddividendo la stringa contenuta nel nodo intorno al carattere i-esimo creando due nuove foglie e un nuovo padre per queste ultime, il quale andrà a sostituire il nodo di partenza.
Per quanto riguarda il caso 1, si procede individuando il nodo A contenente il carattere $(i+1)$-esimo con una versione iterativa della funzione *INDEX*; una volta trovato si risale verso la radice fino a quando non si sale da un figlio destro e si rimuove il collegamento di quest'ultimo con il padre. Successivamente si risale ancora verso la radice, rimuovendo i collegamenti a tutti i figli

destri che contengano sottoalberi facenti riferimento ai caratteri successivi all'i-esimo; in pratica gli eventuali figli destri vanno rimossi soltanto quando si sale da un figlio sinistro. Inoltre, durante l'operazione si sottrae la lunghezza dei nodi tagliati in questo modo dal peso di tutti i nodi che li avevano nel sottoalbero sinistro. Sebbene sarebbe possibile calcolare tali lunghezze con la funzione *WEIGHT*, ancora una volta risulta molto più efficiente fare ricorso al campo *len*. Infine si concatenano i nodi che sono stati tagliati dall'albero di partenza per ottenere la rope S2 ed eventualmente si bilanciano i due alberi così ottenuti. La procedura è presentata nell'Algoritmo 4.
Il primo ciclo while, essendo praticamente equivalente a una indicizzazione, richiede un tempo $\mathcal{O}(\log n)$; il blocco if successivo è eseguito in tempo $\mathcal{O}(1)$, poiché opera su stringhe di dimensione costante; infine, i cicli while alla fine prendono ancora un tempo $\mathcal{O}(\log n)$, in quanto risalgono da una foglia in direzione della radice, visitando al più h nodi. Quindi la procedura *SPLIT* ha una complessità totale di $\mathcal{O}(\log n)$, a meno dell'eventuale tempo richiesto per il bilanciamento dei due alberi ottenuti.
Nella Figura 2.3 è presentata la suddivisione della rope "Hello my name is Simon" a metà. Essendo la stringa lunga 22 caratteri, la suddivisione avviene intorno al carattere in posizione 10, ovvero "a"; le rope ottenute dalla suddivisione sono, quindi, "Hello my na" e "me is Simon", che hanno come radici rispettivamente i nodi A e P. Per prima cosa, similmente a quanto avviene per la funzione *INDEX*, viene individuato il nodo contenente il primo carattere della seconda sottostringa, cioè il nodo K. Poiché tale carattere, ovvero la "m", è il primo della stringa memorizzata in K, ci troviamo nel primo caso e si può procedere risalendo verso il nodo G, in quanto questo è il primo nodo nel percorso verso la radice che si raggiunge da un figlio destro. A questo punto si rimuove il collegamento tra G e il suo figlio destro K e si sottrae il suo peso (che, essendo una foglia, equivale alla lunghezza) da tutti i nodi che si raggiungono risalendo da sinistra nel percorso verso la radice, ovvero D e A. Successivamente si continua a risalire verso la radice fino a quando non si sale da un figlio sinistro, raggiungendo il nodo D; quindi, si taglia il collegamento con il suo figlio destro H e si sottrae ancora una volta il numero di caratteri contenuti nel sottoalbero destro dai pesi di tutti i nodi che lo avevano nel sinistro, in questo caso soltanto A. Infine si continua la risalita verso la radice fino a fermarsi proprio su di essa: poiché è stata raggiunta da B, che è un figlio sinistro, si tenta ancora una volta di rimuovere il collegamento con il figlio destro di A, che però in questo caso non esiste. Una volta terminata questa operazione si procede a concatenare fra loro tutti i sottoalberi così rimossi, cioè quelli radicati in K e H.

```
1  SPLIT(S, i)
2      if S.root == NIL or i < 0 or i ≥ S.root.len
3          return NIL, NIL
4      iter = S.root
5      index = i+1
6      while iter.left ≠ NIL or iter.right ≠ NIL do
7          if index < iter.weight and iter.left ≠ NIL
8              iter = iter.left
9          else if index ≥ iter.weight and iter.right ≠ NIL
10             index = index - iter.weight
11             iter = iter.right
12     if index ≠ 0
13         allocate new nodes s1, s2 and p
14         s1.str = SUBSTRING(iter.str, 0, index)
15         s2.str = SUBSTRING(iter.str, index, iter.str.length - index)
16         p.left = s1
17         p.right = s2
18         s1.parent = p
19         s2.parent = p
20         p.weight = s1.len
21         p.len = s1.len + s2.len
22         p.parent = iter.parent
23         if iter == S.root
24             S.root = p
25         else if iter == iter.parent.left
26             p.parent.left = p
27         else
28             p.parent.right = p
29         iter = p.right
30     allocate empty ropes S2 and tmp
31     subtractCount = 0
32     fromLeft = true
33     while iter.parent ≠ NIL and iter == iter.parent.left do
34         iter = iter.parent
35     while iter.parent ≠ NIL do
36         iter = iter.parent
37         if fromLeft == true
38             iter.weight = iter.weight - subtractCount
39             if iter.right ≠ NIL
40                 tmp.root = iter.right
41                 subtractCount = subtractCount + iter.right.len
42                 iter.right.parent = NIL
43                 iter.right = NIL
44                 if S2 == NIL
45                     S2 = tmp
46                 else
47                     S2 = CONCATENATE(S2, tmp)
48         iter.len = iter.len - subtractCount
49         if iter.parent == NIL
50             break
51         if iter == iter.parent.left
52             fromLeft = true
53         else
54             fromLeft = false
55     return S, S2
```

Algoritmo 4: Suddivisione intorno all'i-esimo carattere

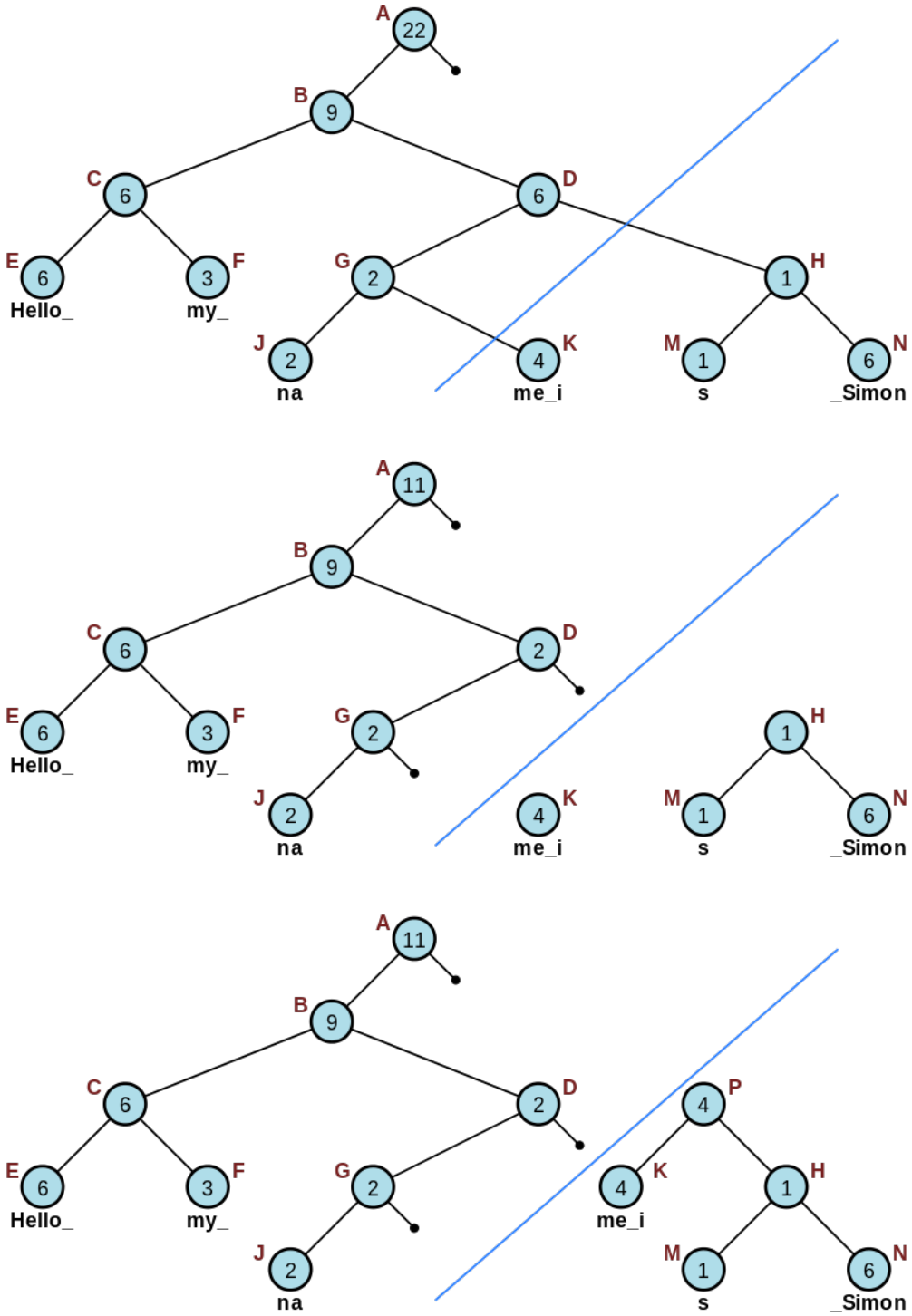

Figura 2.3: Suddivisione di una rope esattamente a metà

2.4 Inserimento

L'inserimento di una stringa $S^{'}$ nella posizione i-esima della rope S si può effettuare in modo molto semplice mediante operazioni di suddivisione e concatenazione, come è possibile osservare nell'Algoritmo 5. Effettuando una *SPLIT* intorno all'i-esimo carattere si ottengono le sottostringhe $S_1, ..., S_i$ e $S_{i+1}, ..., S_m$; successivamente si concatena la prima con $S^{'}$ e infine la stringa risultante con la seconda, ottenendo la stringa $S_1, ..., S_i, S^{'}, S_{i+1}, ..., S_m$. Essendo composta da una suddivisione e due concatenazioni, la procedura ha un costo computazionale pari alla somma delle tre operazioni, ovvero $\mathcal{O}(\log n)$ se non è necessario bilanciare l'albero risultante.

```
1 INSERT(S, S', i)
2     if S'.root == NIL or i < 0 or i ≥ S.root.len
3         return S
4     S1, S2 = SPLIT(S, i)
5     return CONCATENATE(CONCATENATE(S1, S'), S2)
```

Algoritmo 5: Inserimento della stringa $S^{'}$ dopo l'i-esimo elemento di S

2.5 Cancellazione

La cancellazione di una sottostringa di S che inizia al carattere i e di lunghezza j, ovvero composta dai caratteri $S_i, ..., S_{i+j-1}$, similmente all'inserimento, si può realizzare in modo molto semplice, come si vede nell'Algoritmo 6. È sufficiente suddividere la stringa di partenza S intorno ai caratteri (i-1)-esimo e (i+j-1)-esimo in modo da estrarre la sottostringa da cancellare e, successivamente, concatenare le stringhe $S_1, ..., S_{i-1}$ e $S_{i+j}, ..., S_m$. Anche la cancellazione, essendo formata da due suddivisioni e una concatenazione, richiede un tempo $\mathcal{O}(\log n)$.

```
1 CANCEL(S, i, j)
2     if i < 0 or i ≥ S.root.len or i + j - 1 ≥ S.root.len
3         return S
4     S1, S2 = SPLIT(S, i - 1)
5     S2, S3 = SPLIT(S2, j - 1)
6     return CONCATENATE(S1, S3)
```

Algoritmo 6: Cancellazione della sottostringa da i a i+j-1 da S

2.6 Confronto delle complessità con gli array

In Tabella 2.1 si riportano le complessità temporali nel caso pessimo delle operazioni sopra descritte per le rope e per i classici array di caratteri. È bene ricordare che la complessità della maggior parte delle operazioni effettuate su una rope hanno in realtà complessità $\mathcal{O}(h)$ nel caso pessimo, dove h è l'altezza dell'albero rappresentante la rope; se, però, tale albero viene mantenuto approssimativamente bilanciato la complessità diventa $\mathcal{O}(\log n)$ nel caso pessimo.

Operazione	Rope	Array
Insert	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Concatenate	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Index	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
Cancel	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Split	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$

Tabella 2.1: Confronto tra le complessità temporali di rope e array

3 Implementazione della struttura dati Rope

3.1 Progettazione delle classi

Per quanto riguarda i dettagli implementativi delle strutture dati Rope e Node, la scelta più conveniente è risultata quella di definire una singola classe in C++ per ognuna. Non è stato fatto uso di ereditarietà in quanto, nonostante la Rope presenti una struttura di albero binario, buona parte delle operazioni sono specifiche per l'utilizzo con le rope e le stringhe e non si avrebbe avuto un riuso vantaggioso del codice.
Come è possibile osservare in Figura 3.1, la classe Node prevede tutti i campi già descritti nell'Introduzione, ovvero parent, left, right, weight e str; in aggiunta sono presenti due ulteriori campi introdotti appositamente:

- *len*, anch'esso già descritto in precedenza, rappresenta il numero totale di caratteri presenti nel sottoalbero radicato in un determinato nodo; viene citato come campo specifico all'implementazione in quanto il suo unico scopo è quello di ottimizzare le operazioni di suddivisione e concatenazione, ma sarebbe stato anche possibile farne a meno utilizzando la funzione *WEIGHT* descritta nell'Algoritmo 1;

- *MAX_STR_LEN*, rappresenta la lunghezza massima che può avere il campo str presente in un nodo; il suo valore è stato impostato a 10 per permettere di testarne la corretta applicazione con testi di lunghezza più ridotta, ma ovviamente in un'implementazione reale potrebbe essere anche molto maggiore.

È stato, inoltre, implementato il metodo *init* per inizializzare i campi del nodo separatamente dai costruttori, in modo da poter fare uso di una procedura ricorsiva. La versione che prende in input altri due nodi permette di inizializzare un nodo di concatenazione, ovvero che non contiene nessuna

stringa, mentre l'overload che prende in input una stringa inizializza il nodo con quest'ultima o, qualora la sua lunghezza dovesse superare il valore MAX_STR_LEN, lo rende un nodo di concatenazione e inizializza ricorsivamente i figli ciascuno con metà della stringa di partenza.

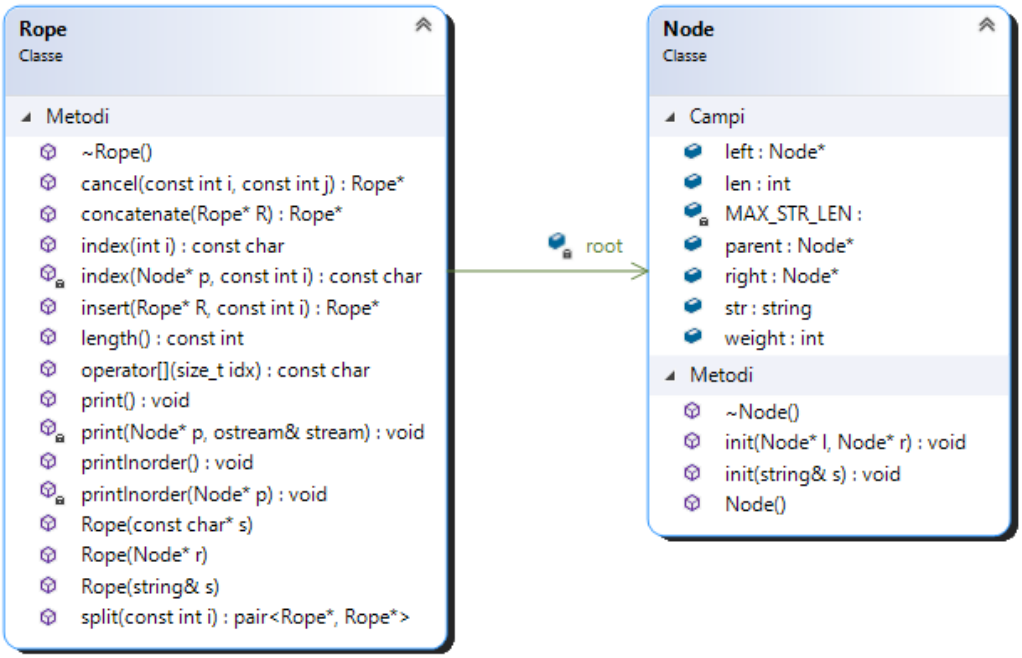

Figura 3.1: Diagramma UML delle classi

Mentre la classe Node rappresenta il singolo nodo della struttura dati, è la classe Rope a definire tutte le operazioni fondamentali che essa deve supportare; presenta un unico attributo, ovvero un puntatore *root* al nodo radice della struttura, e implementa i metodi *cancel*, *concatenate*, *index*, *insert* e *split*. Ovviamente, trattandosi di una implementazione in C++, poiché i metodi lavorano principalmente con i puntatori, è stato necessario effettuare controlli aggiuntivi sulla validità degli stessi. Inoltre, è presente un overload del metodo index che effettua i controlli sulla validità dei parametri, permettendo di invocarlo automaticamente sulla radice senza bisogno di avere riferimenti a oggetti di tipo Node e sono stati implementati alcuni metodi di supporto non trattati dal punto di vista teorico nel capitolo precedente:

- *length*, che restituisce la lunghezza totale della stringa contenuta nella rope;

- *printInorder*, che effettua ricorsivamente una visita inorder della rope stampando a schermo i contenuti di ogni nodo, ovvero il peso per i nodi di concatenazione e il frammento di testo per le foglie;
- *print*, che, similmente alla precedente, fa una visita inorder dell'albero stampando a schermo l'intera stringa contenuta al suo interno.

Infine, è stato effettuato l'overloading degli operatori di inserimento in uno stream «, di indicizzazione [] e di somma +.

3.2 Bilanciamento dell'albero

Come già visto, poiché la struttura dati rope è implementata come albero binario, la maggior parte delle sue operazioni ha una complessità computazionale dipendente dall'altezza dell'albero. Questo implica che se l'albero dovesse diventare eccessivamente sbilanciato, tale complessità rischierebbe di diventare $\mathcal{O}(n)$ nel caso pessimo. Nonostante non sia stato fatto per evitare di appesantire l'implementazione e di distogliere l'attenzione dagli aspetti più importanti di questa struttura, è ovviamente possibile realizzare una versione della rope in grado di rimanere approssimativamente bilanciata e, quindi, garantire le prestazioni logaritmiche di cui si parla nella descrizione teorica. Una possibile soluzione sarebbe l'impiego di altre strutture dati arboree auto-bilancianti, come gli splay-tree o i treap. In alternativa è possibile utilizzare un algoritmo di bilanciamento descritto in *"Ropes: An Alternative to Strings", Software Practice and Experience 25, 12 (Dec 1995), pp. 1315 - 1330* di Boehm, Atkinson e Plass; in questo caso sono possibili due approcci: lasciare che sia esclusivamente il client a invocare in modo esplicito l'operazione di bilanciamento della rope, oppure effettuarla in automatico secondo un qualche criterio che indichi che la rope è diventata eccessivamente sbilanciata. Inoltre, secondo la documentazione della libreria STL realizzata da SGI disponibile all'indirizzo `https://www.boost.org/sgi/stl/ropeimpl.html`, questo algoritmo riesce ad avere una complessità nel caso pessimo di $\mathcal{O}(n)$ e, in generale, sembra non appesantire particolarmente l'implementazione.