Ingegneria degli Algoritmi

Antonio D'Orazio
0242178

Prima prova pratica – Traccia 1 (Albero binario Con Lazy Deletion)

Introduzione

Dato un nodo da rimuovere dall'albero, l'implementazione della **lazy deletion** in un albero binario di ricerca consiste nel **non eliminare fisicamente** l'elemento ma segnarlo come *eliminato* tramite un apposito campo. Il nodo sarà ancora fisicamente presente, ma non sarà visibile attraverso una ricerca, e sarà sovrascritto da un nuovo nodo attivo che sarà inserito in quella posizione.

Il principale **vantaggio** è quello di *risparmiare tempo* durante la cancellazione, in quanto il tutto si riduce alla modifica di un flag.

Di **contro** però, se sono effettuate tante cancellazioni senza effettuare reinserimenti questo si tradurrebbe in uno *spreco di spazio in memoria*, dato da elementi fisicamente presenti ma non più utili.

Pertanto, è consigliabile utilizzare questa tecnica se si prevede di effettuare **poche eliminazioni** o se ci saranno molte sovrascritture date da **continui inserimenti**, in questo modo i nodi non più utili possono essere rimpiazzati da informazioni utilizzabili, ottimizzando lo spazio a disposizione.

Per quanto riguarda l'implementazione che verrà a breve analizzata, ci sono due premesse da fare:

- Per consentire l'inserimento di chiavi alfanumeriche oltre che numeriche, ogni chiave è convertita in **string** tramite il metodo str() prima di essere inserita o analizzata (ad esempio tramite ricerca). In questo modo l'ordinamento è *consistente* e non vi sono conflitti sui tipi
- La classe *LazyDictionary* è quella al **livello più alto** per l'utente, le sue funzioni restituiscono immediatamente i valori del dizionario. Le classi *LazyBinaryTree* e *BinaryNode* sono ad un **livello inferiore**: gestiscono i dati a livello di BinaryNode e non di output diretto.

Esempio: Dalla classe LazyDictionary chiamo il metodo search(self, key). Questo metodo chiamerà LazyBinaryTree.searchNode(self, key) che restituirà un BinaryNode. Da qui Search si occuperà di elaborare il BinaryNode e

La **modularità** è garantita dal fatto che la **logica** dell'albero risiede nel file *LazyBinaryTree*.py, mentre in *LazyDictionary*.py troviamo solamente il **livello più esterno** del dizionario, con le chiamate alle varie funzioni. Ciò consente di riciclare la classe LazyDictionary appoggiandosi ad un albero gestito in maniera differente, a patto di mantenere gli stessi nomi e parametri delle funzioni in LazyBinaryTree.

Descrizione dei metodi ed analisi del tempo teorico

Classe BinaryNode

Costruttore	Assegna i valori info, father, leftSon e rightSon. In particolare, info è una lista di tipo [chiave, valore, attivo]. Attivo è un valore booleano per implementare la Lazy Deletion. Se attivo = true l'elemento è visibile. Se attivo = false l'elemento è cancellato. **Tempo:** O(1) poiché esegue delle semplici assegnazioni e la lista ha grandezza costante
toString(self)	Semplice metodo che restituisce una stringa contenente tutte le informazioni del nodo. **Tempo:** O(1) poiché costruisce semplicemente una stringa

Classe LazyBinaryTree

Costruttore	Stabilisce la radice root dell'albero. Di default: albero vuoto. **Tempo:** O(1) poiché l'assegnamento è su un singolo nodo
insert(self, key, value):	Inserisce coppia chiave-valore nell'albero secondo la seguente logica: • Creo una tripla di valori [chiave, valore, True (valore di attivo)] • Creo prima un Nodo e poi un Albero su questa tripla di valori • Se l'albero su cui vogliamo effettuare l'inserimento è vuoto, allora la radice dell'albero diventa quella dell'albero appena creato • Altrimenti scorro tutti i nodi dell'albero, sfruttando le proprietà dell'ordinamento, finché non trovo un nodo nullo o non attivo. • Se il nodo è nullo: inserisco il nuovo nodo come figlio destro o sinistro del nodo nullo • Se il nodo è disattivo: sostituisco le sue informazioni con quelle del nuovo nodo Insert restituisce True se il nodo è stato inserito da zero, False se ne è stato sovrascritto uno già presente (perché disattivo o perché con la stessa chiave). **Tempo:** O(1) nel caso migliore, quando la radice è nulla o viene sovrascritta. O(logn) nel caso peggiore, quando si inserisce una nuova foglia o ne viene sovrascritta un'altra, poiché corrisponde all'altezza di un albero binario [Ricontrolla]
InsertAsLeftSubTree(self, father, subtree):	Inserisce la radice di un sottoalbero come figlio sinistro del nodo father. Semplicemente assegna all'attributo father.leftSon il sottoalbero. **Tempo:** : O(1) poiché si limita ad assegnare al valore leftSon un nodo
InsertAsRightSubTree(self, father, subtree):	Inserisce la radice di un sottoalbero come figlio destro del nodo father. Semplicemente assegna all'attributo father.rightSon il sottoalbero. **Tempo:** O(1) poiché si limita ad assegnare al valore rightSon un nodo

delete(self, key):	Metodo per la cancellazione. Cancella dall'albero il nodo con chiave key. Prima di tutto effettua una search per ottenere il nodo con chiave key. Dopodiché se il nodo ha 0 o 1 figli la funzione può chiamare il metodo oneSonDeletion, che imposta a False il campo attivo del nodo senza conseguenze per i figli. Se invece il nodo ha due figli, il metodo il seguente algoritmo: • Cerco il predecessore del nodo (il figlio con chiave piu grande) sfruttando maxKeySon; • Scambio il contenuto dei due nodi; • Elimino con oneSonDeletion il nodo da eliminare, poiché ora è in una posizione sicura Delete restituisce True se il nodo è stato eliminato, False se non era presente **Tempo:** O(logn) (dato da search) + tempo O(1) dato da oneSonDeletion + O(logn) dato da maxKeySon = O(logn) nel caso peggiore. [Ricontrolla + caso migliore]
oneSoneDeletion: [attenzione mancano parametri]	Implementa la lazy deletion: si limita ad impostare a False il campo active del nodo **Tempo:** O(1) poiché si limita ad una assegnazione
search(self, key):	Restituisce il nodo corrispondente alla chiave key in ingresso. Dopo aver verificato che l'albero non sia vuoto, esegue il seguente algoritmo: • assegna alla variabile curr il nodo radice dell'albero; • finché il nodo curr non è nullo, confronta la sua chiave secondo tre casi: • se la chiave è la stessa inserita in input, restituisce curr se il nodo attivo, None altrimenti • se la chiave in input è minore o maggiore della chiave di curr, assegna a curr rispettivamente curr.leftSon o curr.rightSon • se alla fine curr è un nodo nullo e non ha trovato nulla, restituisce None **Tempo:** O(1) nel caso peggiore, quando la radice è nulla o la radice corrisponde al nodo da cercare. Tempo O(logn) nel caso peggiore, che corrisponde all'altezza dell'albero e si verifica quando l'elemento si trova nelle foglie più in profondità.
key(self, node):	Metodo di appoggio, restituisce la chiave del nodo (None se nodo è nullo) **Tempo:** O(1) poiché si limita a restituire un campo di un nodo
value(self, node):	Metodo di appoggio, restituisce il valore del nodo (None se nodo è nullo) **Tempo:** O(1) poiché si limita a restituire un campo di un nodo
isActive(self, node):	Metodo di appoggio, restituisce lo stato del nodo (False se nodo è nullo) **Tempo:** O(1) poiché si limita a restituire un campo di un nodo
info(self, node):	Metodo di appoggio, restituisce le informazioni [chiave, valore, attivo] (None se nodo è nullo) **Tempo:** O(1) poiché si limita a restituire tre campi di un nodo

maxKeySon(self, root):	Restituisce il nodo figlio con chiave più grande: scorre nei sottoalberi destri e restituisce il nodo più in profondità. **Tempo:** O(logn) nel caso peggiore, corrispondente all'altezza dell'albero. O(1) nel caso migliore, quando la radice non ha figlio destro.
DFS(self):	Restituisce una lista di BinaryNode.info ordinati secondo il criterio della visita in profondità. Per farlo prima di tutto inizializza una pila inserendo la radice (se non nulla). A questo punto, finché lo stack non è vuoto, esegue questi passaggi: • estrae dalla pila l'ultimo elemento in ordine di inserimento • se marcato come attivo, lo inserisce nella lista da restituire • inserisce nella pila, se non nulli, il figlio destro e sinistro del nodo estratto al punto uno **Tempo:** visita l'albero in O(n) iterazioni occupando spazio O(n) (l'array da restituire)
BFS(self):	Restituisce una lista di BinaryNode.info ordinati secondo il criterio della visita in ampiezza. Per farlo prima di tutto inizializza una coda inserendo la radice (se non nulla). A questo punto, finché la coda non è vuota, esegue questi passaggi: • estrae dalla coda il primo elemento in ordine di inserimento • se marcato come attivo, lo inserisce nella lista da restituire; • inserisce nella coda, se non nulli, il figlio destro e sinistro del nodo estratto al punto uno **Tempo:** visita l'albero in O(n) iterazioni occupando spazio O(n) (l'array da restituire)
stampa(self):	Consente di stampare l'albero completo, compresi gli elementi disabilitati, al fine di analizzarne visivamente la gerarchia. Sfrutta la tecnica della visita in profondità. **Tempo:** O(n), corrispondente al tempo della visita in profondità

Classe LazyDictionary

Costruttore	Inizializza un Dizionario costruendo un albero binario di ricerca che implementa la lazy deletion, e ne salva la lunghezza. Supporta la creazione di un dizionario partendo da una lista precedentemente creata, con sintassi [[chiave1, valore1], [chiave2, valore2], [.. , ..], [chiaveN, valoreN]]. Per implementare questa funzionalità è stato sufficiente scorrere ogni elemento della lista, richiamando il metodo per l'inserimento nel dizionario per ogni coppia **Tempo:** Costruisce un Dizionario in tempo O(nlogn), dove n è la grandezza della lista in input. Questo perché corrisponde ad effettuare n volte la procedura Insert che ha tempo O(logn)
add(self, key, val):	Aggiunge una voce al dizionario. Prende in input chiave e valore da inserire e richiama il metodo insert dell'albero. Se insert ha inserito un nuovo nodo (restituendo True) add incrementa self.length, altrimenti il valore non viene modificato. **Tempo:** O(1) nel caso migliore ed O(logn) nel caso peggiore, poiché chiama insert ed eventualmente esegue un incremento

remove(self, key, val):	Aggiunge una voce al dizionario. Prende in input chiave e valore da inserire e richiama il metodo delete dell'albero. Se delete restituisce True decrementa self.length. **Tempo:** O(logn) nel caso peggiore e O(1) nel caso migliore, poiché effettua una Delete ed eventualmente esegue un decremento
get(self, key):	Restituisce il valore del nodo con chiave key. Chiama il metodo search dell'albero e dal nodo che ottiene restituisce value(nodo). **Tempo:** O(1) nel caso migliore ed O(logn) nel caso peggiore, poiché effettua una Search ed una Value
size(self):	restituisce il numero di elementi nel dizionario, presenti nella variabile self.length **Tempo:** O(1) poiché restituisce il valore di un attributo
allPairs(self):	restituisce la lista di coppie [chiave, valore] degli elementi nel dizionario. Per farlo effettua una visita chiamando la funzione DFS, dopodiché ne restituisce il risultato. **Tempo:** O(n) poiché effettua una visita DFS
keys(self):	restituisce la lista di tutte le chiavi degli elementi del dizionario. Per farlo effettua una visita chiamando la funzione DFS, dopodiché scorrendo il risultato salva tutte le chiavi in una nuova lista, per poi restituirla **Tempo:** O(n) poiché ho O(n) + O(n), rispettivamente per la visita DFS e poi per estrarre ogni chiave
values(self):	restituisce la lista di tutte i valori degli elementi del dizionario. Per farlo effettua una visita chiamando la funzione DFS, dopodiché scorrendo il risultato salva tutti i valori in una nuova lista, per poi restituirla **Tempo:** O(n) poiché ho O(n) + O(n), rispettivamente per la visita DFS e poi per estrarre ogni valore

Analisi del tempo sperimentale

Nel file demo.py è presente una sezione con delle funzioni che racchiudono piccole prove sul dizionario, strutturate in modo da poterle profilare attraverso la seguente fila di istruzioni:

```
popolaDizionario()
cProfile.run(funzioneDaAnalizzare()', "output.txt")
p = pstats.Stats("output.txt")
p.strip_dirs().sort_stats("time").print_stats()
```

Di seguito è mostrato l'output prodotto da ogni profilazione

`def creaDizionario():` `    registro = LazyDictionary()`	6 function calls in 0.000 seconds

```
Ordered by: internal time
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
     1    0.000    0.000    0.000    0.000 LazyDictionary.py:10(__init__)
     1    0.000    0.000    0.000    0.000 Demo.py:59(creaDizionario)
     1    0.000    0.000    0.000    0.000 <string>:1(<module>)
     1    0.000    0.000    0.000    0.000 LazyBinaryTree.py:33(__init__)
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

```python
def popolaDizionario():
    # Popolo dizionario
    registro.add("Matematica", 8)
    registro.add("Storia", 6.5)
    registro.add("Scienze", 6.5)
    registro.add("Fisica", 6.5)
    registro.add("Geografia", 6.5)
    registro.add("Informatica", 6.5)
    registro.add("Arte", 6.5)
    registro.add("Italiano", 6.5)
    registro.add("Ed. Fisica",
"Distinto")
```

```
94 function calls in 0.000 seconds
Ordered by: internal time
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     9    0.000    0.000    0.000    0.000 LazyBinaryTree.py:38(insert)
     1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
    45    0.000    0.000    0.000    0.000 LazyBinaryTree.py:139(key)
     9    0.000    0.000    0.000    0.000 LazyDictionary.py:18(add)
     1    0.000    0.000    0.000    0.000 Demo.py:62(popolaDizionario)
     9    0.000    0.000    0.000    0.000 LazyBinaryTree.py:15(__init__)
     9    0.000    0.000    0.000    0.000 LazyBinaryTree.py:33(__init__)
     9    0.000    0.000    0.000    0.000 LazyBinaryTree.py:158(info)
     1    0.000    0.000    0.000    0.000 <string>:1(<module>)
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

```python
def chiavi():
    #Chiavi del dizionario
    chiavi = registro.keys()
```

```
100 function calls in 0.000 seconds
Ordered by: internal time
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
     1    0.000    0.000    0.000    0.000 LazyBinaryTree.py:173(DFS)
     9    0.000    0.000    0.000    0.000 Stack.py:66(pop)
     9    0.000    0.000    0.000    0.000 Stack.py:63(push)
    10    0.000    0.000    0.000    0.000 Stack.py:76(isEmpty)
    27    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
     1    0.000    0.000    0.000    0.000 LazyDictionary.py:44(keys)
     9    0.000    0.000    0.000    0.000 {method 'pop' of 'list' objects}
    10    0.000    0.000    0.000    0.000 LazyBinaryTree.py:151(isActive)
    19    0.000    0.000    0.000    0.000 {built-in method builtins.len}
     1    0.000    0.000    0.000    0.000 Demo.py:74(stampaChiavi)
     1    0.000    0.000    0.000    0.000 <string>:1(<module>)
     1    0.000    0.000    0.000    0.000 Stack.py:60(__init__)
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

```python
def valori():
    #Valori del dizionario
    valori = registro.values()
```

```
100 function calls in 0.000 seconds
Ordered by: internal time
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
     1    0.000    0.000    0.000    0.000 LazyBinaryTree.py:173(DFS)
     9    0.000    0.000    0.000    0.000 Stack.py:63(push)
```

	9 0.000 0.000 0.000 0.000 Stack.py:66(pop) 19 0.000 0.000 0.000 0.000 {built-in method builtins.len} 10 0.000 0.000 0.000 0.000 LazyBinaryTree.py:151(isActive) 10 0.000 0.000 0.000 0.000 Stack.py:76(isEmpty) 1 0.000 0.000 0.000 0.000 LazyDictionary.py:52(values) 27 0.000 0.000 0.000 0.000 {method 'append' of 'list' objects} 9 0.000 0.000 0.000 0.000 {method 'pop' of 'list' objects} 1 0.000 0.000 0.000 0.000 <string>:1(<module>) 1 0.000 0.000 0.000 0.000 Demo.py:77(valori) 1 0.000 0.000 0.000 0.000 {method 'disable' of '_lsprof.Profiler' objects} 1 0.000 0.000 0.000 0.000 Stack.py:60(__init__)

`def coppie():`
`  #Coppie [chiave, valore]`
`  coppie = registro.allPairs()`
	91 function calls in 0.000 seconds
	Ordered by: internal time
	ncalls tottime percall cumtime percall filename:lineno(function)
	1 0.000 0.000 0.000 0.000 {built-in method builtins.exec}
	1 0.000 0.000 0.000 0.000 LazyBinaryTree.py:173(DFS)
	9 0.000 0.000 0.000 0.000 Stack.py:63(push)
	9 0.000 0.000 0.000 0.000 Stack.py:66(pop)
	10 0.000 0.000 0.000 0.000 Stack.py:76(isEmpty)
	18 0.000 0.000 0.000 0.000 {method 'append' of 'list' objects}
	9 0.000 0.000 0.000 0.000 {method 'pop' of 'list' objects}
	10 0.000 0.000 0.000 0.000 LazyBinaryTree.py:151(isActive)
	19 0.000 0.000 0.000 0.000 {built-in method builtins.len}
	1 0.000 0.000 0.000 0.000 Demo.py:81(coppie)
	1 0.000 0.000 0.000 0.000 LazyDictionary.py:39(allPairs)
	1 0.000 0.000 0.000 0.000 Stack.py:60(__init__)
	1 0.000 0.000 0.000 0.000 <string>:1(<module>)
	1 0.000 0.000 0.000 0.000 {method 'disable' of '_lsprof.Profiler' objects}

`def rimuovi():`
`  #Elimino un elemento`
`  registro.remove("Storia")`
	11 function calls in 0.000 seconds
	Ordered by: internal time
	ncalls tottime percall cumtime percall filename:lineno(function)
	1 0.000 0.000 0.000 0.000 {built-in method builtins.exec}
	1 0.000 0.000 0.000 0.000 LazyBinaryTree.py:91(delete)
	1 0.000 0.000 0.000 0.000 LazyBinaryTree.py:117(search)
	1 0.000 0.000 0.000 0.000 LazyDictionary.py:24(remove)
	2 0.000 0.000 0.000 0.000 LazyBinaryTree.py:139(key)
	1 0.000 0.000 0.000 0.000 <string>:1(<module>)
	1 0.000 0.000 0.000 0.000 Demo.py:85(rimuovi)
	1 0.000 0.000 0.000 0.000 {method 'disable' of '_lsprof.Profiler' objects}
	1 0.000 0.000 0.000 0.000 LazyBinaryTree.py:112(oneSonDeletion)
	1 0.000 0.000 0.000 0.000 LazyBinaryTree.py:151(isActive)

`def albero():` `  # Stampo l'intero albero`	81 function calls in 0.000 seconds Ordered by: internal time ncalls tottime percall cumtime percall filename:lineno(function)

```
DA VEDERE SE SERVE PRINT O STA GIA
DENTRO LA FUNZIONE !!!!!!!!!!
    print(registro.tree.stampa())
```

```
     1    0.000    0.000    0.000    0.000 LazyBinaryTree.py:207(stampa)
    10    0.000    0.000    0.000    0.000 {built-in method builtins.print}
     1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
     9    0.000    0.000    0.000    0.000 Stack.py:63(push)
     9    0.000    0.000    0.000    0.000 Stack.py:66(pop)
    10    0.000    0.000    0.000    0.000 Stack.py:76(isEmpty)
     9    0.000    0.000    0.000    0.000 {method 'pop' of 'list' objects}
    19    0.000    0.000    0.000    0.000 {built-in method builtins.len}
     9    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
     1    0.000    0.000    0.000    0.000 Demo.py:89(albero)
     1    0.000    0.000    0.000    0.000 <string>:1(<module>)
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
     1    0.000    0.000    0.000    0.000 Stack.py:60(__init__)
```

```
def elemento():
    # Stampo un preciso elemento
    print(registro.get("Arte"))
```

```
12 function calls in 0.000 seconds
Ordered by: internal time
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
     1    0.000    0.000    0.000    0.000 {built-in method builtins.print}
     1    0.000    0.000    0.000    0.000 Demo.py:93(elemento)
     1    0.000    0.000    0.000    0.000 LazyBinaryTree.py:117(search)
     1    0.000    0.000    0.000    0.000 LazyDictionary.py:29(get)
     3    0.000    0.000    0.000    0.000 LazyBinaryTree.py:139(key)
     1    0.000    0.000    0.000    0.000 <string>:1(<module>)
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
     1    0.000    0.000    0.000    0.000 LazyBinaryTree.py:145(value)
     1    0.000    0.000    0.000    0.000 LazyBinaryTree.py:151(isActive)
```

Dalla seguente analisi possiamo dedurre che, dato un piccolo campione di dati, il *tempo di esecuzione* di ogni funzione è pressoché **trascurabile**. Inoltre, notiamo che i metodi che hanno eseguito *più chiamate* ad altre funzioni sono **chiavi()** e **valori()** con 100 chiamate, mentre quello che ne ha eseguite *di meno* è **creaDizionario()** con 6, seguito da **rimuovi()** con 11.