Progetto in itinere del corso di Ingegneria degli algoritmi a.a. 2017-2018

Andrea Graziani - matricola 0189326

15 dicembre 2017

Indice

1 Un albero binario di ricerca con *lazy deletion*

1.1 Descrizione

Un albero binario di ricerca con *lazy deletion* rappresenta un particolare struttura dati che, a differenza dei classici BST, utilizza un differente approccio per gestire la rimozione dei nodi presenti all'interno dell'albero. Come vedremo anche in base a valutazioni sperimentali, l'uso della *lazy deletion* applicato agli alberi di ricerca, che d'ora in avanti supponiamo di tipo AVL, permette di ottenere prestazioni migliori in alcuni contesti rispetto ai classici BST.

1.1.1 Operazione `search`

Grazie alla *proprietà di ricerca*[1], l'implementazione dell'operazione `search` è molto semplice. Infatti, qualora volessimo eseguire la ricerca di un nodo x all'interno di un albero, dobbiamo innanzitutto confrontare la chiave k che stiamo cercando con la chiave v della radice dell'albero di ricerca: se sono uguali e la radice risulti contrassegnata come *non* eliminata abbiamo trovato l'elemento, altrimenti proseguiamo la ricerca nel sotto-albero sinistro qualora $k < v$ o in quello destro se $k > v$. Ripetiamo la ricerca a partire dal figlio sinistro o destro della radice usando la stessa strategia, finché non troviamo la prima occorrenza *valida* di un nodo avente chiave pari a k; in caso di insuccesso avremo $x \notin T$.

Precisiamo che l'algoritmo, una volta trovato un nodo avente chiave corrispondente a quella cercata, verifica che quest'ultimo non sia stato precedentemente contrassegnato come eliminato: in caso affermativo, il nodo viene scartato e la ricerca riprende a partire dal sotto-albero sinistro del nodo in questione[2], altrimenti esso verrà restituito alla funzione chiamante.

Dal momento che dopo ogni confronto, se non abbiamo trovato l'elemento, scendiamo di un livello nell'albero T, il tempo richiesto dalla ricerca $O(h)$ nel caso peggiore, dove h è pari all'altezza dell'albero; poiché l'albero T in questione è AVL, se esso possiede n allora ha altezza pari $O(\log n)$[3] e dunque il tempo di esecuzione dell'operazione `search` sarà $O(\log n)$[4]. Lo pseudo-codice dell'operazione `search` è riportato nella figura 1.

1.1.2 Operazione `delete`

Implementare l'operazione `delete` di un albero di ricerca AVL con *lazy deletion* è molto più semplice rispetto ai classici BST; infatti, una volta individuato il nodo da eliminare all'interno dell'albero di ricerca, operazione che, come potete notare anche dallo pseudo-codice in figura 2, si può effettuare utilizzando la stessa strategia usata nell'implementazione dell'operazione `search`, invece di rimuovere fisicamente il nodo bersaglio, procedura che potrebbe comportare costi aggiuntivi per mantenere il bilanciamento dell'albero, l'eliminazione del nodo avviene semplicemente contrassegnando il nodo obiettivo come *eliminato*; ciò è possibile modificando opportunamente un attributo booleano che verrà

[1]Cfr. Camil Demetrescu & Irene Finocchi & Giuseppe F. Italiano - *Algoritmi e strutture dati, Seconda edizione*, McGraw-Hill, pp. 142-143, Definizione 6.1

[2]Vedi l'implementazione dell'operazione `insert`.

[3]Cfr. *ivi*, pp. 149, Corollario 6.1

[4]Cfr. *ivi*, pp. 155, Teorema 6.2

Algorithm 1 Search

```
function Search(key)
    currentNode ← radice dell'albero
    while currentNode ≠ NULL do
        if (key > currentNode) then
            currentNode ← figlio destro di currentNode
        else if (key < currentNode) then
            currentNode ← figlio sinistro di currentNode
        else
            if (currentNode.isDeleted) then
                currentNode ← figlio sinistro di currentNode
            else
                break
    return currentNode
```

usato in seguito per stabilire se il nodo specificato sia effettivamente eliminato o meno.

Ovviamente, se il nodo da eliminare risultasse già contrassegnato come eliminato, quest'ultimo viene ignorato e la ricerca riprende a partire dal suo figlio sinistro[5].

Chiaramente il tempo di esecuzione dell'operazione `delete` è pari $O(\log n)$ nel caso peggiore[6].

Algorithm 2 DeleteNode

```
function DeleteNode(key)
    currentNode ← radice dell'albero
    while currentNode ≠ NULL do
        if (key > currentNode) then
            currentNode ← figlio destro di currentNode
        else if (key < currentNode) then
            currentNode ← figlio sinistro di currentNode
        else
            if (currentNode.isDeleted) then
                currentNode ← figlio sinistro di currentNode
            else
                currentNode.isDeleted ← True
                return True
    return False
```

[5] Vedi l'implementazione dell'operazione `insert`.

[6] *ibid.*

1.1.3 Operazione `insert`

L'operazione di inserimento di un nodo, di costo pari $O(\log n)$ nel caso peggiore[7], sebbene di semplice implementazione, deve essere adattata per gestire la tecnica *lazy deletion*.

Supponiamo per un istante che l'albero T non contenga nodi contrassegnati come eliminati. In tal caso, l'operazione `insert` è equivalente a quella di un comune BST dove un nuovo nodo con chiave x viene sempre inserito come foglia dell'albero di ricerca. L'operazione `insert` può quindi essere implementata in due passi:

1. Ricerca del nodo v che diventerà genitore del nuovo nodo.
2. Creazione del nuovo nodo con chiave x che diventerà figlio sinistro di v qualora $x \leqslant chiave(v)$ o destro qualora $x > chiave(v)$, in accordo alla proprietà di ricerca dei BST;

Se durante la ricerca di v viene trovato un nodo d contrassegnato come eliminato, l'algoritmo, nel rispetto della proprietà di ricerca, verifica la possibilità di eseguire la sostituzione del nodo d con x, prima di riprendere la ricerca di v.

Qualora ciò sia possibile, l'algoritmo effettua la sostituzione impostando $chiave(d) = chiave(x)$ e $valore(d) = valore(x)$ e modificando opportunamente l'attributo booleano usato per contraddistinguere i nodi eliminati, terminando così la sua esecuzione; facciamo notare che in questo caso non vi è necessità di allocare un nuovo nodo, pertanto l'altezza dell'albero e la sua occupazione di memoria rimangono inalterate.

Se dopo l'esecuzione dell'operazione di `insert` l'albero risultasse sbilanciato, è sufficiente eseguire una rotazione semplice o doppia sul nodo critico per ribilanciarlo in altezza[8]. Lo pseudo-codice dell'operazione `insert` è riportato nella figura 3.

[7] *ibid.*

[8] Cfr. *ivi*, pp. 153, Lemma 6.2

Algorithm 3 Insert

```
function INSERT(albero T, key, value)
    if (T è vuoto) then
        crea il nuovo nodo e impostalo come radice dell'albero T
        return
    else
        currentNode ← radice dell'albero
        parentNode ← NULL
        insertToLeft ← True
        while currentNode ≠ NULL do
            parentNode ← currentNode.parent
            if (not currentNode.isDeleted) then
                if (key > currentNode) then
                    currentNode ← figlio destro di currentNode
                    insertToLeft ← False
                else
                    currentNode ← figlio sinistro di currentNode
                    insertToLeft ← True
            else
                if (currentNode ha un figlio destro con chiave k > key) then
                    currentNode ← figlio destro di currentNode
                    insertToLeft ← False
                else if (currentNode ha un figlio sinistro con chiave k < key) then
                    currentNode ← figlio sinistro di currentNode
                    insertToLeft ← True
                else
                    Sostituisci currentNode con newNode
                    return
        newNode ← alloca un nuovo oggetto Node(key, value)
        if (insertToLeft) then
            parentNode.leftSon ← currentNode.leftSon
        else
            parentNode.rightSon ← currentNode.rightSon
    return
```

Tabella 1: Tempi esecuzione sperimentali (espessi in μs) ottenuti da vari test

Operazione	**BST AVL**	**BST AVL (*lazy deletion*)**
`delete`	50	4
`insert`	41	9
`search`	5	8

1.2 Vantaggi e svantaggi

Un albero binario siffatto presenta almeno due vantaggi:

Facile implementazione Come abbiamo già visto, l'utilizzo della tecnica *lazy deletion* rende l'operazione di rimozione dei nodi molto semplice da implementare.

Efficienza L'algoritmo risulta essere molto efficiente in quei scenari d'uso in cui sono previsti molti inserimenti in grado di rimpiazzare un nodo precedentemente contrassegnato come eliminato per due motivi:

1. Non sono necessarie operazioni di bilanciamento.
2. Si può evitare l'allocazione di un nuovo nodo copiando semplicemente il valore e la chiave da inserire all'interno del nodo da sostituire e contrassegnarlo come non eliminato.

In modo analogo, anche le operazioni di eliminazione dei nodi sono molto efficienti dal momento che richiedono semplicemente la modifica di un attributo booleano. Per rendersi conto delle prestazioni raggiunte dagli BST con *lazy deletion* è sufficiente osservare la tabella 1.

Tuttavia ci sono un serie di svantaggi che occorre analizzare:

Incremento dell'altezza dell'albero Dal momento che i nodi non vengono fisicamente eliminati dall'albero di ricerca, benché contrassegnati come eliminati, questi, continuando a persistere all'interno dell'albero, determinano un aumento dell'altezza dell'albero.

Degradamento delle prestazioni Come conseguenza del punto precedente, soprattutto in tutti quei scenari di utilizzo in cui avvengono molte cancellazioni senza reinserimenti capaci di rimpiazzare i nodi eliminati, tutte le operazioni sui nodi subiscono un degradamento delle prestazioni. Per esempio la ricerca di un nodo all'interno dell'albero potrebbe richiedere la visita ("*inutile*") di molti nodi contrassegnati come eliminati prima di raggiungere, se esiste, il nodo richiesto; vedere a tal proposito i risultati della tabella 1.

Spreco di memoria Dal momento che, come già detto, i nodi eliminati, continuando a persistere all'interno dell'albero, qualora quest'ultimi non venissero rimpiazzati, causano un enorme spreco di memoria.

1.3 Una possibile ottimizzazione

Per limitare lo spreco di memoria e il degradamento delle prestazioni della struttura dati a causa della crescita senza controllo del numero di nodi contrassegnati come eliminati, eventualità possibile qualora il numero di inserimenti in grado di sostituire i suddetti nodi sia insufficiente, l'albero di ricerca che abbiamo implementato permette, *se esplicitamente richiesta dall'utente*, la possibilità di eseguire delle procedure di ottimizzazione con lo scopo di limitare il numero di nodi eliminati.

Questo è stato fatto con una semplice modifica alla procedura `search`: come si può vedere dallo pseudo-codice in figura 4, durante un'ordinaria operazione di ricerca, l'algoritmo mantiene un riferimento a tutti i nodi contrassegnati come eliminati incontrati durante il percorso; se richiesto dall'utente, prima di ritornare il risultato dell'operazione, l'algoritmo procede all'eliminazione fisica dei suddetti nodi riducendone così il numero e aumentando così le prestazioni delle operazioni di ricerca successive. Potete osservare le differenze di prestazioni dalla tabella 2 da cui notiamo il costo maggiore dell'operazione `search-2` rispetto a `search` e un tempo di esecuzione inferiore delle successive `search`.

Algorithm 4 Search-2

```
function SEARCH-2(key, bool optimizationAllowed)
    currentNode ← radice dell'albero
    deletedNodeList ← nuova lista vuota
    while currentNode ≠ NULL do
        if (optimizationAllowed and currentNode.isDeleted) then
            Aggiungi currentNode a deletedNodeList
        if (key > currentNode) then
            currentNode ← currentNode.rightSon
        else if (key < currentNode) then
            currentNode ← currentNode.leftSon
        else
            if (currentNode.isDeleted) then
                currentNode ← currentNode.leftSon
            else
                break
    for all item in deletedNodeList do
        Elimina fisicamente item
    return currentNode
```

1.4 Scenari di utilizzo

Dall'analisi precedente si evince chiaramente che le prestazioni di un albero binario di ricerca che fa uso della *lazy deletion* dipendono in particolar modo dal rapporto esistente tra il numero di cancellazioni e il numero di inserimenti capaci di sostituire i nodi eliminati; se c'è uno squilibrio a sfavore di quest'ultimi, si verificherà col tempo un degradamento delle prestazioni tali da dover adottare contromisure come la `search-2`.

In ogni caso esaminiamo ora un possibile scenario di utilizzo: i *database*.

Tabella 2: Analisi sperimentale delle ricerche

Operazione	Tempo (s)
`search`	0.01590916500026651192
`search-2`	0.12874723000095400494
`search` eseguita dopo `search-2`	0.00748537400068016723

Supponiamo, quindi, di dover implementare un database utilizzando un albero binario di ricerca AVL che fa uso della *lazy deletion*: in questo contesto, i record del nostro database saranno rappresentati dai nodi mentre la tabella dall'albero. Possiamo utilizzare la chiave dei nodi come chiave primaria della tabella per individuare univocamente un record.

Supponiamo inoltre che l'amministratore del sistema, per motivi di sicurezza e prevenzione riguardante la perdita dei dati, decidesse di avere più copie della base dati su più supporti fisici separati. Avendo la necessità di mantenere le varie copie della basi dati sincronizzate e coerenti fra loro, sarà quindi necessario replicare tutte le operazioni di aggiornamento a tutte le copie: questo comporterà inevitabilmente un degradamento delle prestazioni.

Evidentemente, data la ridondanza dei dati su più supporti, l'operazione di eliminazione di un nodo potrebbe rivelarsi molto costosa motivo per cui l'uso della tecnica *lazy deletion* potrebbe rivelarsi una buona soluzione qualora sia prevista però la possibilità di sostituire nodi precedentemente eliminati altrimenti si verificherà una crescita senza controllo della dimensione della base dati provocata dalla presenza di enorme mole di nodi eliminati.

Qualora si volesse privilegiare le operazioni di ricerca, mantenendo i vantaggi della *lazy deletion*, l'amministratore del sistema potrebbe anche adottare anche la strategia descritta in `search-2` aumentando così la velocità di accesso a nodi di particolare interesse per gli utenti.

2 Implementazione

Da un punto di vista implementativo, la realizzazione dell'albero binario di ricerca AVL con *lazy deletion* è stata effettuata utilizzando le seguenti classi:

`binarySearchTreeAVL` usata per modellare un comune BST AVL.

`binarySearchTreeLazyDeletionAVL` figlia della classe precedente, modella il BST AVL con *lazy deletion*.

`binarySearchTreeNode` modella un nodo di un comune albero di ricerca.

`binarySearchTreeNodeAVL` questa classe, figlia di quella precedente, modella un nodo di un albero di ricerca AVL.

`binarySearchTreeNodeAVLwithLazyDeletion` questa classe, figlia della precedente, modella un nodo di un BST AVL con *lazy deletion*.

Queue modella una coda.[9]

BinarySearchTreeLazyDelectionAVLTest questa classe contiene metodi utilizzati per eseguire test.

Per ovvi motivi di spazio, omettiamo in questa sede una descrizione dettagliata delle varie classi e dei loro metodi in ogni caso già presente all'interno del codice sorgente. Ricordiamo che per ottenere informazioni dettagliate sull'interfaccia di tutti i metodi definiti nelle varie classi è sufficiente digitare help(<nome della classe>) dalla console dell'interprete *Python* dopo aver eseguito l'importazione della classe desiderata.

Riportiamo per comodità le implementazioni delle operazioni search, delete e insert descritte all'interno della classe binarySearchTreeLazyDeletionAVL.

2.1 Operazione search

```python
def search(self, key, allowRestructuring):
    """
    This function is used to search first occurrence of a node with specified key into tree.

    @param key: Represents a key.
    @param allowRestructuring: A boolean value used to enable or disable 'tree restructuring'.
    @return: If specified node exists, returns it; otherwise it returns 'None'
    """

    # This list object is used to store met invalid nodes...
    invalidNodeList = list()
    currentNode = self._rootNode

    # Searching...
    # ---------------------------------------------------- #
    while(currentNode):

        # Keeping track invalid nodes...
        if (allowRestructuring and (not currentNode._isValid)):
            invalidNodeList.append(currentNode)

        # CASE 1:
        # ------------------------------------------------ #
        if (key < currentNode._key):
            currentNode = currentNode._leftSon

        # CASE 2:
        # ------------------------------------------------ #
        elif(key > currentNode._key):
            currentNode = currentNode._rightSon

```

[9] Cfr. *ivi*, pp. 70-71

```
        # CASE 3:
        # ------------------------------------------------ #
        else:
            if (currentNode._isValid):
                break
            else:
                currentNode = currentNode._leftSon

    # If requested, delete found invalid nodes physically...
    # ---------------------------------------------------- #
    for item in invalidNodeList:
        self.deleteNode(item)

    return currentNode
```

2.2 Operazione delete

```
def delete(self, key):
    """
    This function is used to delete first occurrence of a node with specified key from tree.

    @param key: Represents a key.
    @return: A boolean value: 'True' if specified node is correctly deleted, otherwise 'False'.
    """

    currentNode = self._rootNode

    # Searching...
    # ---------------------------------------------------- #
    while(currentNode):

        # CASE 1:
        # ---------------------------------------------------- #
        if (key < currentNode._key):
            currentNode = currentNode._leftSon

        # CASE 2:
        # ---------------------------------------------------- #
        elif(key > currentNode._key):
            currentNode = currentNode._rightSon

        # CASE 3:
        # ---------------------------------------------------- #
        else:
            if (currentNode._isValid):
```

```
                currentNode._isValid = False
                return True
            else:
                currentNode = currentNode._leftSon

    return False
```

2.3 Operazione insert

```
def insert(self, key, value):
    """
    This function is used to insert a new (key, value) pair
       into tree.

    @param key: Represents a key.
    @param value: Represents a value.
    """

    # If tree is empty, newly created node become root...
    # ---------------------------------------------------- #
    if (not self._rootNode):
        self._rootNode =
            binarySearchTreeNodeAVLwithLazyDeletion(key,
            value)
        return
    else:

        currentNode = self._rootNode
        parentNode = None
        insertToLeft = True

        # Search parent of newly created node...
        # ------------------------------------------------ #
        while (currentNode is not None):

            parentNode = currentNode

            # Node is not deleted...
            # --------------------------------------------#
            if (currentNode._isValid):

                # CASE 1:
                # ---------------------------------------- #
                if (key <= currentNode._key):
                    currentNode = currentNode._leftSon
                    insertToLeft = True

                # CASE 2:
                # -------------------------------------- #
                else:
                    currentNode = currentNode._rightSon
                    insertToLeft = False

```

```
            # Node is deleted...
            # ----------------------------------------- #
            else:

                # CASE 1:
                # --------------------------------------- #
                if (currentNode.hasRightSon() and (key > currentNode._rightSon._key)):
                    currentNode = currentNode._rightSon
                    insertToLeft = False

                # CASE 2:
                # ---------------------------------------- #
                elif (currentNode.hasLeftSon() and (key < currentNode._leftSon._key)):

                    currentNode = currentNode._leftSon
                    insertToLeft = True

                # CASE 3:
                # -------------------------------------- #
                else:

                    # Copy data into deleted node...
                    # --------------------------------- #
                    currentNode._isValid = True
                    currentNode._key = key
                    currentNode._value = value
                    return

        # Now allocate a new 'binarySearchTreeNodeAVLwithLazyDeletion' object...
        # ---------------------------------------------- #
        newNode = binarySearchTreeNodeAVLwithLazyDeletion(key, value)

        # Add parent...
        newNode._parent = parentNode

        # Insert new node...
        # ---------------------------------------------- #
        if (insertToLeft):
            parentNode._leftSon = newNode
            newNode._isLeftSon = True

        else:
            parentNode._rightSon = newNode
            newNode._isLeftSon = False

        # If necessary, update balance factor...
        # ---------------------------------------------- #
        if (parentNode.hasOnlyOneSon()):
            self._updateNodeSubtreesHeightAlongTree(newNode)
```