Τμήμα Ψηφιακών Συστημάτων

Ακαδημαϊκό Έτος: 2020-2021

Μάθημα: Δομές Δεδομένων

Πανώριος Μιχαήλ Ε18127

Email: michaelpanorios@gmail.com

Εργαλεία για την υλοποίηση της εργασίας.

Ο editor που χρησιμοποίησα για την υλοποίηση της εργασίας ήταν το Intellij Idea με kit το JDK 11. Για την υλοποίηση της εργασίας χρησιμοποίησα ως δομή δεδομένων την συνδεδεμένη λίστα (singly linked list).

Τι είναι η singly linked list;

Όπως οι πίνακες, το Linked List είναι μια γραμμική δομή δεδομένων. Σε αντίθεση με τους πίνακες, τα συνδεδεμένα στοιχεία λίστας δεν αποθηκεύονται σε γειτονική τοποθεσία αλλά τα στοιχεία αυτά συνδέονται χρησιμοποιώντας δείκτες.

Γιατί συνδεδεμένη λίστα;

Οι πίνακες ArrayLists που παρέχει η Java μπορούν εξίσου να χρησιμοποιηθούν για την αποθήκευση δεδομένων δυστηχώς όμως προσδίδουν ταυτόχρονα περιορισμούς στον προγραμματιστή, όπως:

- Το μέγεθος τους είναι σταθερό: Πρέπει λοιπόν να γνωρίζουμε εκ των προτέρων το ανώτατο όριο του αριθμού των στοιχείων. Επίσης, γενικά, η εκχωρημένη μνήμη είναι ίση με το ανώτερο όριο ανεξάρτητα από τη χρήση.
- Η εισαγωγή ενός νέου στοιχείου σε μια σειρά στοιχείων είναι δαπανηρή, επειδή η αίθουσα πρέπει να δημιουργηθεί για τα νέα στοιχεία και για τη δημιουργία χώρου, τα υπάρχοντα στοιχεία πρέπει να μετατοπιστούν.

Πλεονεκτήματα συνδεδεμένης λίστας.

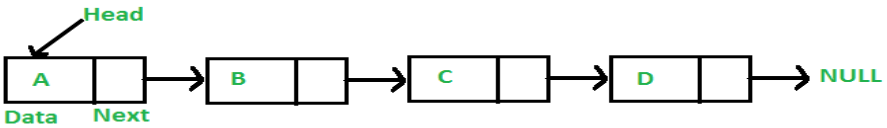
Η ύπαρξη της συνδεδεμένης λίστας παρέχει πλεονεκτήματα έναντι των απλών πινάκων που οι λειτουργίες τους ορισμένες φορές μπορεί να είναι περιορισμένες.

- Δυναμικό μέγεθος.
- Ευκολία εισαγωγής / διαγραφής

Μειονεκτήματα συνδεδεμένης λίστας.

- τυχαία πρόσβαση δεν επιτρέπεται. Πρέπει να έχουμε πρόσβαση σε στοιχεία διαδοχικά ξεκινώντας από τον πρώτο κόμβο. Επομένως, δεν μπορούμε να κάνουμε δυαδική αναζήτηση με συνδεδεμένες λίστες αποτελεσματικά
- Απαιτείται επιπλέον χώρος μνήμης για ένα δείκτη με κάθε στοιχείο της λίστας.
- Δεν είναι φιλικό προς την προσωρινή μνήμη. Δεδομένου ότι τα στοιχεία πίνακα είναι συνεχόμενες θέσεις, υπάρχει τοποθεσία αναφοράς που δεν υπάρχει στην περίπτωση συνδεδεμένων λιστών.

Αναπαράσταση μιας γενικότερης συνδεδεμένης λίστας.



Μια συνδεδεμένη λίστα αντιπροσωπεύεται από ένα δείκτη στον πρώτο κόμβο της συνδεδεμένης λίστας. Ο πρώτος κόμβος ονομάζεται head(ή current). Εάν η συνδεδεμένη λίστα είναι κενή, τότε η τιμή head είναι null.
Κάθε κόμβος σε μια λίστα αποτελείται από τουλάχιστον δύο μέρη:
1) Δεδομένα (Nodes)
2) Δείκτης στον επόμενο κόμβο(next)

*Στην Java μια LinkedList μπορεί να αναπαρασταθεί ως κλάση και ως κόμβος ως ξεχωριστή κλάση, ενώ σε άλλες γλώσσες η υλοποίηση τους ποικίλει.

Υλοποίηση μεθόδων της κλάσης RankList.

`public int insert(Record poi)` Δημιουργεί κόμβο κλάσης **Node**, στον οποίο εγγράφει αντίγραφο της εγγραφής **poi** και εισάγει τον κόμβο στη λίστα στην οποία καλείται η μέθοδος. Επιστρέφει το (ενημερωμένο) πλήθος κόμβων της λίστας.

```
//Inserting a new node to a list.
public int insert(Record poi) {
    Node node = new Node(poi); //making space for a node of type Node
    node.setNext(null);        //setting next pointer to null
    if (null == first) {       //when linked list is empty
        first = node;
        current = node;
    } else {                   //when linked list is non-empty
        current.setNext(node); //setting current pointer to inserted node
        current = node;        //setting current pointer to that node
    }
    return ++nodeCount;        //returning the size of the list
}
```

Παραπάνω φαίνεται η υλοποίηση της μεθόδου insert. Κάθε φορά που την καλώ ουσιαστικά εντάσσω στην συνδεδεμένη λίστα έναν κόμβο με τα απαραίτητα δεδομένα που ζητούνται από την άσκηση. Στην συγκικριμένη περίπτωση, η *πολυπλοκότητα της μεθόδου είναι* ***O(1)*** επειδή κάθε φορά που την καλούμε οι δείκτες βρίσκονται στις σωστές θέσεις έτσι ώστε ο κόμβος να δημιουργηθεί δίπλα από τον τελευταίο. Εφόσον δεν αναζητώ συγκικριμένο κόμβο όπως για παράδειγμα τον μεσαίο η ανάθεση γίνεται απευθείας στο τέλος της λίστας.

Πολυπλοκότητα της μεθόδου insert είναι: O(1)

`public RankList nearest(Point p, int k)` Κατασκευάζει και επιστρέφει λίστα κλάσης **RankList**, που περιέχει τις εγγραφές των **k** εγγύτερων σημείων ενδιαφέροντος από τη λίστα στην οποία καλείται η μέθοδος, στο σημείο **p**.

```
public RankList nearest(Point p, int k) {
    RankList scoreList = new RankList();
    if(size()>1){
        for(int i=0;i<size();i++){
            Node current=this.first;
            Node next=this.first.next;
            for(int j=0;j<size()-1;j++){
                if(current.getPoi().getLocation().dist(p)<next.getPoi().getLocation().dist(p)){
                    Record temp=current.getPoi();
                    current.poi=next.poi;
                    next.poi=temp;
                }
                current=next;
                next=next.next;
            }
        }
    }
    Node current1=this.first;
    if(k>0 && k<=size()){
        for(int i=0;i<k;i++){
            scoreList.insert(current1.getPoi());
            current1=current1.getNext();
        }
        return scoreList;
    }
    return null;
}
```

Η μέθοδος nearest δέχεται ως ορίσματα το Point p (σημείο αναφοράς χρήστη) και int k. Αρχικά δημιουργώ μια κενή λίστα scoreList στην οποία θα αποθηκεύω τα δεδομένα που πληρούν τις προυποθέσεις. Αν η λίστα είναι άδεια τότε η μέθοδος επιστρέφει την τιμή null. Εάν όμως περιέχει τουλάχιστον έναν κόμβο τότε αναθέτω θέσεις στους δύο δείκτες current και next που θα χρειαστούν στην πορεία. Χρησιμοποιώ εμφωλευμένες for loops διότι χρειάζομαι να ανατρέξω όλους τους κόμβους και να κάνω συγκρίσεις με όλους για να βρώ και να κατατάξω τα πιο κοντινά σημεία k. (σχόλια στις γραμμές του κώδικα για περαιτέρω ανάλυση). Από την αλλή δοθέντος του k το οποίο αναπαριστά το πλήθος των κοντινότερων σημείων, με μία for loop ανατρέχω την λίστα που έχει καταταχθεί ανάλογα και προσθέτω στην κενή scoreList τους k κόμβους. Η πολυπλοκότητα του πρώτου κομματιού του κώδικα δείχνει να είναι **O(**n^2**)** και του δεύτερου **O(n).**

- Συνήθως οι εμφωλευμένες επαναλήψεις έχουν πολυπλοκότητα O(n^2). Διότι στην χειρότερη κάθε επανάληψη τρέχει για n στοιχεία. (n*n= n^2). Κάποιες φορές φορές μπορεί στην πράξη να είναι n οι φορές που θα εκτελεστεί αλλά πάντα παίρνουμε το χειρότερο σενάριο. Μαθηματικά λοιπόν, 1+2+3+4...+n = (n^2 + n)/2 = n^2/2 + n/2. Πώς αυτό μετατρέπεται όμως σε O(n^2); Στην ουσία έχουμε n^2 >= n^2/2 + n/2. Κάνοντας τις πράξεις, πολλαπλασιάζοντας δηλαδή με 2 είναι: 2n^2 >= n^2 + n. Σπάμε το 2n^2 για να πάρουμε : n^2 + n^2 >= n^2 + n και αφαιρούμε n^2 και από τα δυο μέλη: n^2 >= n. Είναι προφανές λοιπόν ότι n^2 >= n (“=” λόγω n=0,1.. ως πιθανές τιμές).
- Το τελευταίο κομμάτι του κώδικα, αυτό της εμφάνισης έχει πολυπλοκότητα O(n),αφού το insert έχει O(1), λόγω του for loop που θα εκτελεστεί στην χειρότερη n φορές.

Πολυπλοκότητα της μεθόδου nearest είναι: O(n^2)

public RankList nearest(Point p, double maxDist) Κατασκευάζει και επιστρέφει λίστα κλάσης **RankList**, που περιέχει τις εγγραφές σημείων ενδιαφέροντος από τη λίστα στην οποία καλείται η μέθοδος, που βρίσκονται σε απόσταση το πολύ **maxDist** από το σημείο **p**.

```
public RankList nearest (Point p,double maxDist){
    RankList nearList = new RankList();
    Node current = this.first;
    while (current != null) {
        if((current.getPoi().getLocation().dist(p))<maxDist){//Finding the distance between the added points with p argument
            nearList.insert(current.getPoi()); //Insert the distances shorter than maxDist in a new list.
        }
        current = current.getNext();
    }
    return nearList;
}
```

Η μέθοδος nearest δέχεται ως ορίσματα το Point p (σημείο αναφοράς χρήστη) και double maxDist. Αρχικά δημιουργώ μια κενή λίστα στην οποία πρόκειται να εισάγω τους κόμβους που συμφωνούν στην προυπόθεση που θέτω. Επίσης δημιουργώ τον δείκτη current που δείχνει στον πρώτο κόμβο της λίστας. Στην συνέχεια με μια while loop χρησιμοποιώ τον δείκτη ως μετρητή, δηλαδή μέχρι να δείξει το κενό null τόσο αυτή να τρέχει. Μέσα στην επανάληψη υπολογίζω την ευκλίδεια απόσταση μεταξύ του σημείου p και των συντεταγμένων του εκάστοτε σημείου ενδιαφέροντος. Έαν αυτή είναι μικρότερη από την δοθείσα μέγιστη απόσταση τότε εισάγω τον κόμβο της υπάρχουσας λίστας στην nearList που έχω δημιουργήσει. Αφού γίνει το πέρασμα στην αρχική γεμάτη λίστα τότε η επανάληψη σταματά και επιστρέφει την nearList.

Για τον υπολογισμό της πολυπλοκότητας καταλαβαίνω ότι η while θα εκτελεσθεί στην χειρότερη n φορές. Κοιτάζοντας το σώμα της επανάληψης βλέπω την if που υπακούει σε μία συνθήκη όπου έπειτα περιλαμβάνεται το insertion που είναι O(1). Συνεπώς έχουμε O(n)+O(1). Οπότε παίρνοντας πάλι την χειρότερη περίπτωση με βάση το Big-O notation καταλήγω στο συμπέρασμα,

Πολυπλοκότητα της μεθόδου nearest είναι: O(n)

public RankList highScore(int k) Κατασκευάζει και επιστρέφει λίστα κλάσης **RankList**, που περιέχει τις εγγραφές των **k** σημείων ενδιαφέροντος από τη λίστα στην οποία καλείται η μέθοδος, με τις υψηλότερες βαθμολογίες.

```
public RankList highScore ( int k){
    RankList highscoreList = new RankList();
    Node current = this.first;
    if(k>0 && k<=size()){
        for(int i=0;i<k;i++){
            highscoreList.insert(current.getPoi());
            current=current.getNext();
        }
        return highscoreList;
    }
    return null;
}
```

Η μέθοδος αυτή δέχεται ως όρισμα έναν ακέραιο k που προσδιορίζει τον αριθμό των υψηλόβαθμων κόμβων που θέλουμε να περάσουμε στην highscoreList που αρχικοποιώ στην πρώτη γραμμή της. Αξίζει να αναφέρω ότι έχω δημιουργήσει και μια μέθοδο ταξινόμησης bubblesort με ονομασία bubblesort η οποία ταξινομεί την λίστα με βάση το σκορ του κάθε κόμβου. Ξανά, κρίνεται απαραίτητος ένας δείκτης current που θα τρέξω την λίστα. Εάν ο ακέραιος k είναι θετικός ΑΛΛΑ και μικρότερος από το πλήθος κόμβων της λίστα τότε ανατρέχω την ταξινομημένη λίστα k φορές και εντάσσω σε αυτή του k κόμβους έναν προς έναν. Τέλος, επιστρέφω την highscoreList.

Για τον υπολογισμό της πολυπλοκότητας αντιλαμβάνομαι ότι το k παίρνει ακέραιες τιμές και εμφανίζει πλήθος κόμβων. Το πλήθος όμως αυτό μπορεί να μην είναι σταθερού χρόνου πχ 4 ή 5 αλλα ακόμη υφίσταται και η περίπτωση να εμφανίζω όλη την λίστα που θα 'ναι στην χειρότερη μεγέθους n. Άρα σύμφωνα με την επανάληψη for η πολυπλοκότητα είναι **O(n).**

Πολυπλοκότητα της μεθόδου highScore είναι: O(n)

`public RankList highScore(double minScore)` Δημιουργεί και επιστρέφει λίστα κλάσης **`RankList`**, που περιέχει τις εγγραφές των σημείων ενδιαφέροντος από τη λίστα στην οποία καλείται η μέθοδος, με τις βαθμολογίες τουλάχιστον **`minScore`**.

```
public RankList highScore ( double minScore){
    RankList minscoreList = new RankList();
    Node current = this.first;
    for(int i=0;i<size();i++){
        if(current.getPoi().score>=minScore){
            minscoreList.insert(current.getPoi());
            current=current.getNext();
        }
    }
    return minscoreList;
}
```

Σε αυτή την μέθοδο δημιουργώ ξανά μια κενή λίστα minScoreList που θα εντάξω τα δεδομένα μου. Ξανά ορίζω έναν δείκτη current που δείχνει στην κεφαλή της λίστας. Στην συνέχεια εκτελώ μια επανάληψη for στην οποία εμπεριέχεται μια δομή επιλογής η οποία γίνεται αληθής μόνο εάν το σκορ του κόμβου που δείχνει ο δείκτης είναι μεγαλύτερο ή ίσο της παραμέτρου minScore. Αν ισχύει καλώντας την insert εντάσσω τα δεδομένα του κόμβου στην κενή λίστα και στην συνέχεια μεταφέρω τον δείκτη στην επόμενη θέση.

Όσο αφορά την πολυπλοκότητα, ξανά με την επανάληψη αυτή αντιλαμβάνομαι ότι θα εκτελεστεί size() φορές, όσες το πλήθος των κόμβων δηλαδή. Στην χειρότερη περίπτωση το πλήθος αυτό μπορεί να είναι μεγέθους n.

Πολυπλοκότητα της μεθόδου highScore είναι: O(n)

`public RankList inCommonWith(RankList rankList)` Κατασκευάζει και επιστρέφει λίστα κλάσης **RankList**, που περιέχει τις εγγραφές των σημείων ενδιαφέροντος που περιέχονται τόσο στη λίστα στην οποία καλείται η μέθοδος, όσο και στη λίστα **rankList** (δύο εγγραφές αφορούν στο ίδιο σημείο ενδιαφέροντος, αν έχουν το ίδιο **id**).

```
public RankList inCommonWith (RankList rankList){
RankList newRankList = new RankList();
Node temp=first;
Node current=this.first;
Node previous=null;
if(temp!=null && temp.getPoi().id==current.getPoi().id){
    first=temp.next;
    newRankList.insert(current.getPoi());
}
while(temp!=null && temp.getPoi().id!=current.getPoi().id){
    newRankList.insert(current.getPoi());
    previous=temp;
    temp=temp.next;
    current=current.next;
}
if(temp==null)
    previous.next=temp.next;

return newRankList;
}
```

Από την περιγραφή της μεθόδου inCommonWith που παίρνει ως όρισμα μια λίστα τύπου RankList καταλαβαίνω ότι πρέπει να την υλοποιήσω με σκοπό να εντοπίζει τις διπλοεγγραφές της αρχικής λίστας με κύριο παράγοντα το id. Για να εισάγω τα δεδομένα που βρίσκω δημιουργώ ξανά μια κενή λίστα και αυτή την φορά τρεις δείκτες temp,current,previous που θα με βοηθήσουν στην διαχείρηση της λίστας. Στο πρώτο if() ελέγχω εάν υπάρχει διπλοεγγραφή στους πρώτους κόμβους O(1). Στην συνέχεια με την while ελέγχω εάν υπάρχει διπλοεγγραφή κρατώντας σταθερό τον έναν κόμβο και παιρνώντας την άλλη διαθέσιμη λίστα διαδοχικά. Εάν βρεθεί διπλοεγγραφή τότε διαχειρίζομαι τους δείκτες έτσι ώστε αφαιρώντας την, η λίστα να συνεχίσει να αποτελεί ενιαίο σώμα. Έπειτα στην newRankList εντάσσω τους κόμβους που βρήκα ως διπλοεγγραφές. Τέλος την επιστρέφω. Η while ανατρέχει θεωρητικά δύο λίστες ωστόσο αυτές διαχειρίζονται από τους δείκτες οπότε θα εκτελεστεί στην χειρότερη n φορές, μέχρι δηλαδή να φτάσει στο τέλος της λίστας.

Πολυπλοκότητα της μεθόδου inCommonWith είναι: O(n)

Υλοποίηση των δοθέντων ερωτημάτων της εργασίας

(α) των k εγγύτερων σημείων ενδιαφέροντος σε δεδομένο σημείο με δεδομένη ελάχιστη βαθμολογία,
(β) των k πιο υψηλόβαθμων σημείων ενδιαφέροντος σε απόσταση το πολύ d από δεδομένο σημείο,
(γ) των σημείων ενδιαφέροντος με δεδομένη ελάχιστη βαθμολογία, σε απόσταση το πολύ d από δεδομένο σημείο,
(δ) των σημείων ενδιαφέροντος που απέχουν το πολύ d από δύο διαφορετικά σημεία.

Για την υλοποίηση των μεθόδων αυτών είναι απαραίτητη η χρήση των παραπάνω μεθόδων.

```
public RankList answerOfA(Point p,int k,double minScore){
RankList aList= new RankList();
aList=nearest(p,size()).highScore(minScore);
return aList;
}
```

Για την απάντηση του πρώτου ερωτήματος δημιουργώ μια μέθοδο με τρία ορίσματα Point p,int k,double minScore. Στην συνέχεια δημιουργώ μια κενή λίστα. Για να βρω τα κοντινότερα σημεία από το σημείο p πρέπει να υπολογιστεί η ευκλείδια απόσταση μεταξύ του p και των υπάρχοντων σημείων αναφοράς και έπειτα να εξάγω τα δεδομένα με βάση την ελάχιστη βαθμολογία που δίνει ο χρήστης. Η λίστα γεμίζει με τους κόμβους που συμφωνούν στις προυποθέσεις και τέλος την επιστρέφω. Λόγω της μεθόδου nearest η πολυπλοκότητα της answerOfA είναι O(n^2)

Πολυπλοκότητα της μεθόδου answerOfA είναι: O(n²)

```
public RankList answerOfB(Point p,double maxDist){
RankList bList = new RankList();
bList=nearest(p,maxDist);
return bList;
}
```

Για την απάντηση του δεύτερου ερωτήματος δημιουργώ μια μέθοδο με δύο ορίσματα Point p, double maxDist. Αφού έχω ταξινομήσει την λίστα με την μέθοδο sort() τότε απλώς καλώ την nearest() που περνάω ως παραμέτρους τα ορίσματα της μεθόδου για να εντοπίσω τα πιο υψηλόβαθμα σημεία ενδιαφέροντος που απέχουν το πολύ συγκικριμένη απόσταση. Τέλος δημιουργώ μια λίστα, καλώ την μέθοδο η οποία την γεμίζει και έπειτα την επιστρέφω. Η πολυπλοκότητα είναι η ίδια με την μέθοδο nearest(p,maxDist).

Πολυπλοκότητα της μεθόδου answerOfB είναι: O(n)

```
public RankList answerOfC(Point p,double maxDist,double minScore){
RankList cList = new RankList();
cList=highScoreC(minScore).nearest(p, maxDist);
return cList;
}
```

Για την απάντηση του τρίτου ερωτήματος δημιουργώ μια μέθοδο με τρία ορίσματα Point p, double maxDist, double minScore. Σε αυτή δημιουργώ μια κενή λίστα την οποία γεμίζω καλώντας την highscoreC πολυπλοκότητας O(n) και έπειτα την nearest εξίσου . Με την κήση αυτών των μεθόδων καταφέρνω να βρώ τα σημεία ενδιαφέροντος με την ελάχιστη βαθμολογία και απόσταση το πολύ maxDist από το σημείο αναφοράς p.

Πολυπλοκότητα της μεθόδου answerOfC είναι: O(n)

```
public RankList answerOfD(Point p,double maxDist){
RankList dList = new RankList();
Node current = this.first;
while (current != null) {
    int counter=0;
    if((current.getPoi().getLocation().dist(p))<maxDist){//Finding the distance between the added points with p argument
        counter++;
    }
    if(counter>=2)dList.insert(current.getPoi());
        current = current.getNext();
    }
return dList;
}
```

Για την απάντηση του τελευταίου ερωτήματος δημιουργώ ξανά μια κενή λίστα που θα συμπεριλάβω τους κόμβους που συμφωνούν στις προυποθέσεις. Ουσιαστικά η προσέγγιση είναι η ίδια με την μέθοδο nearest(p,maxDist) με την μόνη διαφορά την προσθήκη ενός μετρητή counter που θα υπολογίζει εάν η απόσταση του ενός κόμβου με τους υπόλοιπους είναι μικρότερος από το maxDist. Κάθε φορά που η απόσταση είναι μικρότερη τότε ο μετρητής θα αυξάνεται κατά μια μονάδα και όταν είναι μεγαλύτερος ή ίσος του 2 τότε εισάγει τον κόμβο αυτόν στην dList καθώς σημαίνει ότι απέχει το πολύ d από 2 τουλάχιστον σημεία. Όσον αφορά την πολυπλοκότητα υπάρχει η while που θα εκτελεστεί στην χειρότερη περίπτωση n φορές και εφόσον στις υπόλοιπες γραμμές του κώδικα υπάρχουν απλές αναθέσεις τιμών και if τότε είναι ξεκάθαρο πως η πολυπλοκότητα της μεθόδου είναι O(n).

Πολυπλοκότητα της μεθόδου answerOfD είναι: O(n)

Εχω προσθέσει την μέθοδο sortH και sortL (highest και lowest αντίστοιχα) για την ταξινόμηση της λίστας για κάθε περίπτωση που κρίνεται αναγκαία. Ο κώδικας που χρησιμοποίησα ήταν bubblesort όπου η πολυπλοκότητα τους είναι **O(n²)**.

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	nearest(p,maxDist)	nearest(p,k)	highscore(k)	highscore(minScore)	inCommonWith(ranklist)	insert(poi)	answerOfA	answerOfB	answerOfC	answerOfD	sortH	sortL	highscoreC
2	O(n)	O(n2)	O(n)	O(n)	O(n)	O(1)	O(n2)	O(n)	O(n)	O(n)	O(n2)	O(n2)	O(n)