Καρακάσης Χρήστος ΑΕΜ: 2751
Ανδρεάδης Φίλλιπος ΑΕΜ: 2730

ΜΑΘΗΜΑ: Δομές Δεδομένων
PROJECT #1 - LinkedPages

Παρουσίαση προγράμματος οργάνωσης σελίδων και συνδέσμων.

Το πρόγραμμα χωρίζεται σε 3 κύρια μέρη:

- Κλάση Controller
- Κλάση InputParser
- Δομές

Ανάλυση βασικών κλάσεων

Κλάση: *Controller*

- Είναι υπεύθυνη για τη σειρά εκτέλεσης των εντολών
- Αποκωδικοποιεί τις εντολές από το αρχείο με τα commands, και καλεί τις αντίστοιχες μεθόδους
- Δέχεται ως όρισμα το argument του προγράμματος
- Χρησιμοποιεί enums για την συνδεση εντολών και strings

Κλάση: InputParser

- Είναι υπεύθυνη για το διάβασμα των εγγραφών απο το input και την εισαγωγή τους ως σελίδα-σύνδεσμος στην αντίστοιχη δομή
- Έχει 4 βασικές μεθόδους, οι 3 εκ των οποίων συνδέονται με την δικιά τους δομή και η 4η είναι υπεύθυνη για την εμφάνιση του progress bar, όση ώρα διαβάζει από το αρχείο και για το άνοιγμα του αρχείου input
- Οι 3 μέθοδοι επιστρέφουν την ίδια την δομή στον controller αφού την γεμίσουν και είναι έτοιμη να δεχθεί τις εντολές

Ανάλυση Δομών

1. Array

Η υλοποίηση σε πίνακες έγινε με την χρήση vector.

Προσθήκη O(1)

- 1ος (εξωτερικός) vector υποστηρίζεται από έναν manual έλεγχο δυναμικής δέσμευσης μνήμης. Αντί για την βασική push_back του vector αρχικοποιούμε τον πίνακα με ένα βήμα allocation_step που ορίζεται είτε από τον χρήστη είτε παίρνει default τιμή 1 (μετά από test η χρυσή τομή μεταξύ χώρου και χρόνου). Η βάση με την οποία γεμίζει ο “εξωτερικός” πίνακας είναι :
  - Αν η σελίδα είναι μεγαλύτερη του μεγέθους του πίνακα, κάνε re allocate μέχρι να μην είναι.
  - Αν η θέση στον πίνακα με index την σελίδα είναι πιασμένη κανε push_back σε αυτη την θέση (*).
  - Αν δεν είναι πιασμένη κάνε new vector και πρόσθεσε το σελίδα-σύνδεσμος

- 2ος (εσωτερικοί) vectors έχουν την default υλοποίηση του vector, μέσω push_back. Σε κάθε θέση του 1ου πίνακα αρχικοποιούμε και ορίζουμε έναν 2ο vector που αναπαριστα:
  - Στην 1η θέση του vector είναι η σελίδα
  - Στις υπόλοιπες μπαίνουν τα λινκς
  - Head - Tail λογική

  (*) Αν η θέση είναι πιασμένη τότε έχει ήδη προστεθεί η σελίδα μία φορά οπότε εισάγουμε στον πίνακα της τον σύνδεσμο.

Διαγραφή O(1)

- Λόγω συνθήκης, δεν διαγράφουμε σελίδες παρά μόνο ακμές(συνδέσμους), οπότε κατά την διαγραφή ζητάμε την θέση του εξωτερικού πίνακα (ίση με την σελίδα), και με binary search βρίσκουμε το στοιχείο προς διαγραφή στον “εσωτερικό” ταξινομημένο πίνακα.

Εύρεση γειτόνων O(1)

- Η πολυπλοκότητα έχει να κάνει με το access οπότε όπως και στην διαγραφή απλά ζητάμε την σελίδα και μας επιστρέφεται ο “εσωτερικός” πίνακας, μείον το 1ο στοιχείο (η σελίδα δηλαδή)

Εύρεση συνεκτικών συνιστωσών

- Κατά μήκος του προγράμματος χρησιμοποιούμε την ίδια λογική cloning, δηλαδή αντί να κάνουμε διπλά insert και να διατηρούμε δύο πίνακες, όταν ζητηθεί η εντολή παίρνουμε έναν deep clone της δομής και σε αυτόν εφαρμόζεται ο καθρεπτισμός (από κατευθυνόμενος γίνεται μη-κατευθυνόμενος γράφος). Τελικά μετά την κλωνοποίηση με τον DFS και την τεχνική colors βρίσκουμε τις συνιστώσες στον κλώνο.
  Σε αυτό το σημείο καλό είναι να σημειωθεί ότι αν οι συνιστώσες του γράφου ζητούνται τακτικά ίσως να κάνει περισσότερο κακό παρά καλό αυτή η τακτική, οπότε σε αυτή την περίπτωση θα πρέπει να αλλάξει η λογική και απλά να διατηρούμε 2 πίνακες.

2. AVL

Το avl υλοποιήθηκε πιο ανεξάρτητα από ότι τα arrays, και υποστηρίζει templates. Έτσι μπορούμε θεωρητικά να ζητήσουμε πιο πολύπλοκες δομές της μορφής AVL<AVL<T>>

Απαραίτητη συνθήκη είναι ο πιο εσωτερικός τύπος να είναι βασικός τύπος(int, float, char κλπ) για να μπορέσουν να έχουν υπόσταση οι συγκρίσεις για το avl.

Βασική δομή του avl είναι το struct Node, που περιέχει

- Ένα id που προσδιορίζει την θέση στο avl (page)
- Ένα γενικό αντικείμενο T που αποθηκεύει κάποιο data (στην προκειμένη περίπτωση αποθηκεύουμε το εσωτερικό avl με τους συνδέσμους της σελίδας)
- Ένα struct left και ένα right, για να στηθεί το δέντρο
- Έναν unsigned char height που κρατάει το ύψος κάθε κόμβου στο δέντρο(*ανάλυση παρακάτω)

Επίσης κάθε avl κρατάει έναν pointer στην ρίζα του δέντρου και λόγω της φύσης του avl αντί για raw pointers χρησιμοποίθηκαν unique_smart pointers. (κάθε κόμβος έχει έναν αποκλειστικά γονέα)

Σύσταση

- Ύψος
  Κανονικά τα δέντρα avl δεν αποθηκεύουν το ύψος του κόμβου, αλλά την διαφορά μεταξύ ενός αριστερού και δεξιού υποδέντρου που παίρνει μόνο 3 τιμές (-1 ,0 ,1) , που με την μικρότερη μονάδα αποθήκευσης πιάνει 1 byte τουλάχιστον. (Επιλέγουμε να την αποθηκεύσουμε σε unsigned char (-127,127) αντι για integer. Έτσι κερδίζουμε 3 byte για κάθε κόμβο.)
  Όμως το ύψος ενός avl κατά ορισμού δεν ξεπερνά το $1.44 \log_2(n + 2)$ όπου n ο αριθμός των κόμβων. Ακόμα και για 1 δις κλειδία δηλαδή το ύψος δεν ξεπερνά το 44, το οποίο μπορούμε εύκολα να αποθηκεύσουμε σε έναν char.
  Έτσι αντί να κρατάμε το “balance factor”, κρατάμε το ύψος του κόμβου, και δεν αυξάνουμε την μνήμη ενώ μειώνεται δραματικά ο χρόνος εκτέλεσης, αφού τώρα όλες οι εργασίες υπολογισμου ύψους ανάγονται σε O(1) αντί για αναδρομικούς υπολογισμούς του balance factor.

- Εσωτερική δομή
  Όπως αναφέρθηκε πιο πρίν κάθε κόμβος αποθηκεύει εσωτερικά ένα αντικείμενο γενικού τύπου. Στην προκειμένη περίπτωση το 1ο avl “εξωτερικό” αποθηκεύει για T = AVL<int> ένα 2ο avl “εσωτερικό” που κρατάει τους συνδέσμους. Όπου int ο βασικός τύπος συγκρίσεων δηλαδή για την εισαγωγή, διαγραφή και εύρεση ενός κόμβου ψάχνουμε με κλειδιά τύπου int. Χωρίς περαιτέρω ανάλυση οπότε για εισαγωγή γίνεται get(key) που επιστρέφει το “εσωτερικό” και εκεί γίνεται insert, στην διαγραφή get και delete και στην εύρεση απλά get.

- Εύρεση γειτόνων
  Με βάση τα παραπάνω, για να βρούμε τους γείτονες μίας σελίδας, μέσω get στο εξωτερικό avl, βρίσκουμε το avl με τους γείτονες και ύστερα με έναν traverse in - order αλγόριθμο απαριθμούμε τους συνδέσμους σε αύξουσα σειρά.

- Εύρεση συνεκτικών συνιστωσών
  Εδώ η διαδικασία γίνεται λίγο περίπλοκη λόγω των ανεξάρτητων δομών, και λόγω του ότι η “εσωτερική” δομή δεν γνωρίζει τίποτα για τον πατέρα της. Οπότε για την επίλυση αυτού του προβλήματος πάνω στην color τεχνική προστέθηκε μία 2η μνήμη, με χρήση vector. Απλά όσους γείτονες βρούμε τους προσθέτουμε στην colors καθώς

επίσης και στην memory. Ύστερα αφού τελειώσει η εξωτερική επανάληψη, για κάθε γείτονα εφαρμόζουμε την ίδια τεχνική (παίρνοντας τους γείτονες έναν-έναν από την μνήμη vector), μέχρις ότου να εξαντλήσουμε όλους τους γείτονες στην μνήμη. Snippet:

```
509         void connected_inner(Node& _node, AVL<var<T>>& memory
510               , std::vector<std::vector<var<T>>>& connected_components, AVL<T>& avl)
511       {
512
513           if(_node != nullptr){
514           //std::cout<<"here"<<std::endl;
515               connected_inner(_node->left, memory, connected_components, avl); //traverse to left of 1st avl
516               if(!memory.contains(_node->unique_id)){ //if node hasnt been assigned to a component
517
518                   std::vector<var<T>> comp; //start a new component
519                   comp.push_back(_node->unique_id); //store component's id
520                   memory.add(_node->unique_id); //write node to memory avl
521                   int vec_size = comp.size();//save prev size
522                   int cur_node_id = _node->unique_id;
523                   do{ //repeat
524                       connected_links(avl.get(cur_node_id).getRoot(),memory,comp); //traverse links of 2nd avl
525                       cur_node_id = comp[vec_size-1]; //refresh new size of component
526                       vec_size++; //check next element in component
527                   }while(comp.size() > vec_size); //until no new nodes have not been traversed
528
529                   connected_components.push_back(comp); // finally store component
530               }
531               connected_inner(_node->right,memory, connected_components, avl); // traverse to right of 1st avl;
532           }
533       }
534
535         void connected_links(auto& _node, AVL<var<T>>& memory , std::vector<var<T>>& comp){
536           if(_node != nullptr){
537               connected_links(_node->left, memory, comp); //traverse to left of 2nd avl
538
539               if(!memory.contains(_node->unique_id)){ //if node hasnt been assigned to a component
540                   comp.push_back(_node->unique_id);//store component's id to current component
541                   memory.add(_node->unique_id);//write node to memory avl
542               }
543
544               connected_links(_node->right,memory, comp); // traverse to right of 2nd avl;
545           }
546       }
547
```

(Χρησιμοποιείται και εδώ η λογική του cloning)
Στην 519 εισάγεται στο memory ο κόμβος που επισκεφτήκαμε και ένας counter δείχνει στο μέγεθος του memory.
Στην 524 μαρκάρονται οι σύνδεσμοι του current page, και το memory παίρνει όλους αυτούς τους συνδέσμους.
Στην 526, μέσα στην while αυξάνω τον counter κατά 1 και παίρνω το αντίστοιχο στοιχείο του memory.
Επανάληψη έως ότου δεν έχει νέα στοιχεία στο memory που δεν έχουν “ελεγχθεί” , δηλαδη έως γραμμη 527 (τέλος while).
Ένα + αυτού του τρόπου είναι ότι κρατάμε ταυτόχρονα τα στοιχεία της συνεκτικής συνιστώσας έτοιμα για τύπωμα.
Επίσης τα στοιχεία είναι ήδη ταξινομημένα.

3.Hashtable

Η υλοποίηση της δομής του πίνακα κατακερματισμού έγινε με δύο διαφορετικές κλάσεις hashtable. Η κλάση HashTable αποτελεί έναν πίνακα κατακερματισμού ανοιχτής διεύθυνσης δυναμικού μεγέθους στον οποίο ως κλειδιά αποθηκεύονται οι σελίδες του αρχείου input και ως τιμές ξεχωριστοί πίνακες κατακερματισμού (κλάση HashTableLinks) που περιέχουν όλες τις σελίδες που συνδέονται με την αντίστοιχη σελίδα-κλειδί. Η κλάση HashTableLinks αποτελεί έναν πίνακα κατακερματισμού δυναμικού μεγέθους στον οποίο χρησημοποιείται chaining για την επίλυση των συγκρούσεων. Οι σελίδες που εισάγονται σε αυτόν αποτελούν συγχρόνως και το κλειδί και την τιμή μιας εισαγωγής καθώς στον συγκεκριμένο πίνακα επιθυμούμε να αποθηκεύσουμε μόνο τις σελίδες που συνδέονται με κάποια άλλη σύμφωνα με το αρχείο input. Οι κλάσεις HashEntry και LinkedHashEntry αποτελούν βοηθητικές κλάσεις για τις HashTable και HashTableLinks αντίστοιχα, υλοποιόντας την δομή της κάθε εισαγωγής στους δύο πίνακες.