ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Αναδρομή σε δομές δεδομένων πάνω στο RAM μοντέλο

Χατζή Γεωργία **ΑΜ 11910**
Τσαμάκη Βασιλική **ΑΜ 11921**

1. Van Emde Boas δέντρο

1.1 Ορισμοί.

Η ιδέα αυτή είναι αρκετά παλιά [12, 13]. Το U θεωρούμε ότι είναι ακέραιος της μορφής $n = 2^{2^k}$. Το n είναι ίσο με τον αριθμό των φύλλων ενός δυαδικού δέντρου με ύψος 2^k. Αν ο αριθμός m είναι ακέραιος, τότε ο μεγαλύτερος αριθμός d τέτοιος ώστε το 2^d να διαιρεί το m, καλείται βαθμός του m. Για παράδειγμα rank(12)=2.

Ας φανταστούμε ένα δυαδικό δέντρο με ύψος 2^k. Το επίπεδο μιας κορυφής είναι η απόστασή της από τη ρίζα. Ο βαθμός μίας κορυφής είναι ο βαθμός του επιπέδου στο οποίο βρίσκεται.

Ένα κανονικό υποδέντρο (CS) είναι ένα υποδέντρο με ύψος 2^d που έχει μία ρίζα με βαθμό ≥ d. Ο αριθμός d καλείται βαθμός του CS. Το αριστερό (δεξί) υποδέντρο ενός CS (συμπεριλαμβανομένου και της ρίζας του) καλείται αριστερό (δεξί) κανονικό υποδέντρο. (LCS και RCS).

Μαζί με κάθε κορυφή v βαθμού d συνδέουμε τα παρακάτω κανονικά υποδέντρα:

UC(v): Το πάνω κανονικό υποδέντρο του v που είναι το μοναδικό CS με βαθμό d, που έχει το v σα φύλλο.

LC(v): Το κάτω CS του v είναι το CS με βαθμό d και ρίζα την κορυφή v.

LLC(v): Το αριστερό CS του v είναι το LCS με βαθμό d και ρίζα την κορυφή v.

RLC(v): Το δεξί CS του v είναι το LCS με βαθμό d και ρίζα την κορυφή v.

Η υπογραφή μιας κορυφής v σε κάποιο πρόγονο w είναι το "l" η το "r" ανάλογα αν η κορυφή v ανήκει στο δεξί ή στο αριστερό υποδέντρο του w.

Μια εσωτερική κορυφή ενός υποδέντρου με ύψος j και ρίζα v είναι ένας απόγονος w του v, που ικανοποιεί τη σχέση: level(v) < level(w) < level(v) + j.

Για κάθε κορυφή v, εκτός από τη ρίζα, με $0 \le j \le \kappa+1$ ορίζουμε father(j,v) να είναι ο πιο κοντινός πρόγονος του v του οποίου ο βαθμός είναι ≥ j. Προφανώς father(0,v) είναι ο πατέρας του v. Αν v είναι η ρίζα ενός κανονικού υποδέντρου με βαθμό d > 0, και αν w είναι ένα φύλλο αυτού του υποδέντρου, τότε ο κόμβος που βρίσκεται στα μισά της διαδρομής από τον w στον v, είναι ο father(w,d-1) και καλείται ο half-way κόμβος ανάμεσα στους v και w. Συμβολίζεται σαν hw(v,w).

Για ένα σύνολο $S \subset \{1,\ldots,n\}$ ορίζουμε τις ακόλουθες εντολές:

Min (max) : βρες το μικρότερο (μεγαλύτερο) στοιχείο του S.

Insert (j) : Εισήγαγε το j στο S, Αν δεν είναι ήδη παρόν.

Delete (j) : Σβήσε το j από το S αν είναι παρόν.

Member (j) : Βρες αν το j ανήκει στο S.

Extract min : Σβήσε το μικρότερο στοιχείο του S, αν το S δεν είναι άδειο.

Extract max : Σβήσε το μεγαλύτερο στοιχείο του S, αν το S δεν είναι άδειο.

Predecessor (i): Βρες το μεγαλύτερο στοιχείο του S που είναι < i, αν υπάρχει τέτοιο.

Predecessor (i): Βρες το μικρότερο στοιχείο του S που είναι > i, αν υπάρχει τέτοιο.

Για μία διαμέριση $\Pi=\{I_1, I_2, \ldots, I_r\}$ του διαστήματος $\{1,\ldots,n\}$ σε γειτονικά διαστήματα, ορίζουμε τις ακόλουθες εντολές:

Find (i) : Βρες το διάστημα στο οποίο περιέχεται το i.

Meld(I,J) : Ένωσε τα διαστήματα I και J αν είναι διαδοχικά.

Split(I,i) : Χώρισε το διάστημα I σε δύο διαστήματα. Το i περιέχεται στο δεύτερο διάστημα.

1.2 Αποθήκευση συνόλων σε ένα δέντρο.

Για να αναπαραστήσουμε ένα σύνολο S={1,...,n} όπου $n = 2^{2^k}$ χρησιμοποιούμε ένα δυαδικό δέντρο ύψους 2^k. Τα φύλλα του δέντρου αντιστοιχούν στα στοιχεία 1..n. Σε κάθε κόμβο είναι διαθέσιμες κάποιες πληροφορίες η φύση των οποίων θα περιγραφεί παρακάτω. Οι πληροφορίες αυτές θα χρησιμοποιηθούν για να βρίσκουμε ποια στοιχεία βρίσκονται κάθε στιγμή στο S.

Καλούμε τα φύλλα που αντιστοιχούν σε στοιχεία του S μαζί όλους τους προγόνους τους παρόντες. Όλοι οι άλλοι κόμβοι θα καλούνται απόντες. Η κατάσταση στην οποία καμία κορυφή στο δέντρο δεν είναι παρών, μπορεί να αποκλειστεί με το να θεωρήσουμε ότι το σύνολο S περιέχει πάντα το στοιχείο n.

Μια παραδοσιακή μέθοδος για την αναπαράσταση του συνόλου είναι το μαρκάρισμα όλων των παρόντων κόμβων. Αυτό όμως οδηγεί σε O(logn) αλγορίθμους για εισαγωγή και διαγραφή στο S. Αντί γι' αυτό, κάνουμε το εξής κόλπο. Βρίσκουμε την απαραίτητη πληροφορία κλειδί για να εντοπίσουμε τα μέλη του S, με το να κάνουμε δυαδικό ψάξιμο στα επίπεδα (levels).

Θεωρούμε ένα κανονικό υποδέντρο του οποίου η ρίζα είναι παρούσα. Επιπλέον, ακριβώς ένα από τα φύλλα στο αριστερό υποδέντρο είναι παρόν. Σε αυτή την

περίπτωση μπορούμε να αποθηκεύσουμε αυτό το ένα και μοναδικό φύλλο σε ένα συγκεκριμένο πεδίο πληροφορίας, στη ρίζα, προτρέποντας με αυτό τον τρόπο τον εαυτό μας να κατέβει από την κορυφή, στο επίπεδο των φύλων. Στη συνέχεια υποθέτουμε ότι περισσότερα από ένα φύλλα στο αριστερό υποδέντρο είναι παρόντα. Σε αυτή την περίπτωση, το αριστερό υποδέντρο θα περιέχει μία εσωτερική κορυφή της οποίας και τα δύο παιδιά θα είναι παρόντα. Μια τέτοια κορυφή θα καλείται branch point. Προφανώς ένα branch point είναι ένα λογικό σημείο για να αποθηκευτεί πληροφορία για τους δύο κόμβους στους οποίους οδηγεί. Το πρόβλημα είναι πώς να εντοπίσουμε ένα branch point. Αυτό μπορεί να γίνει με binary search. Αν υπάρχει ένα branch point ανάμεσα στον κόμβο t και το φύλλο v, αποθηκεύουμε στον κόμβο w που βρίσκεται στη μέση του μονοπατιού από τον t στον v, πληροφορία που μας λεει αν υπάρχει ένα branch point στον w, ανάμεσα στον t και τον w ή ανάμεσα στον w και τον v (μπορεί να συμβαίνουν και τα 3 ταυτόχρονα αν υπάρχουν περισσότερα από ένα branch points στο μονοπάτι από τον t στον v. Έστω v μία κορυφή στο επίπεδο $j<2^{\kappa}$ και έστω d ο βαθμός της κορυφής v. Η αποθηκευμένη πληροφορία στον κόμβο v, θα μας πει τι ακριβώς συμβαίνει στα κανονικά υποδέντρα του v.

Η πληροφορία θα αποθηκευτεί σε 5 πεδία: Τα πρώτα δύο πεδία ονομάζονται lmin(v) και rmin(v). Αν έχει οριστεί το lmin(v) θα περιέχει το πιο δεξί παρόν φύλλο του LLC(v). Αν δεν έχει οριστεί, τότε δεν υπάρχει τέτοιο φύλλο. Αν έχει οριστεί το rmin(v) θα περιέχει το πιο αριστερό παρόν φύλλο του RLC(v). Αν δεν έχει οριστεί, τότε δεν υπάρχει τέτοιο φύλλο. Τα υπόλοιπα 3 πεδία ονομάζονται ub(v), lb(v) και rb(v). Μπορούν να είναι +, - ή undefined. Αν lb(v)= +, αυτό σημαίνει ότι από το μονοπάτι του v στο lmin(v) υπάρχει ένα branch point. Στην περίπτωση που lb(v)=- αυτό σημαίνει ότι υπάρχει μόνο ένα παρόν φύλλο στο LLC(v) και το lb(v) είναι undefined αν δεν υπάρχει κανένα τέτοιο φύλλο. Το πεδίο rb(v) έχει το ίδιο νήμα για το RLC(v). Τέλος, το πεδίο ub(v) είναι + ή – ανάλογα με το αν υπάρχει branch point ανάμεσα στο v και τη ρίζα του UC(v). Υπάρχει μία περίπτωση στην οποία όλα τα πεδία πληροφορίας του v είναι μη ορισμένα παρόλο που ο κόμβος v είναι παρόν. Αυτό συμβαίνει όταν ο κόμβος v είναι παρόν, και υπάρχουν t, w τέτοια ώστε το w είναι το μοναδικό παρόν φύλλο στο LLC(t) ή στο RLC(t) και ο v βρίσκεται πάνω στο μονοπάτι από τον t στον w. Δεν υπάρχει λόγος να αποθηκεύσουμε πληροφορία στον κόμβο v, αφού ανάμεσα στον t και τον w τίποτα δεν αλλάζει.

Properness condition: Αν στο LLC(v) υπάρχει μόνο ένα παρόν φύλλο w, τότε για κάθε εσωτερικό κόμβο του LLC(v) όλα τα πεδία πληροφορίας είναι undefined. Αν

στο LLC(v) υπάρχουν περισσότερα από ένα παρόντα φύλλα και αν w είναι ένα παρόν φύλλο και x ο κόμβος στη μέση της απόστασης μεταξύ των v και w τότε: το πεδίο ub(x) και τα δύο αριστερά πεδία (lmin(x), lb(x)) ή τα δύο δεξιά πεδία είναι ορισμένα. Τα πεδία του x είναι όλα ορισμένα στην περίπτωση που ο x είναι branch-point.

1.3 Αλγόριθμοι και βασικές πράξεις

Έστω v ένας κόμβος του δέντρου και w ένας απόγονός του. Δεν ξέρουμε αν ο w ανήκει στο δεξί ή στο αριστερό υποδέντρο του v. Ορίζουμε τις ακόλουθες βασικές πράξεις και συμβολισμούς:

~	:undefined
mymin(w,v)	:το minfield του v στην κατεύθυνση του w.
yourmin(w,v)	:το minfield του v στην άλλη κατεύθυνση
myb(w,v)	:ορίζεται ανάλογα με mymin
yourb(w,v):	:ορίζεται ανάλογα με yourmin
myfriends(w,v)	:είναι το ζευγάρι mymin(w,v) και myb(w,v)
yourfriends(w,v)	:ορίζεται ανάλογα
minof(v)	:If min(v) ≠ ~ Then lmin(v) Else rmin(v) (δίνει το μικρότερο φύλλο στο LC(v) αν υπάρχει τέτοιο φύλλο
used(v)	:Ελέγχει αν υπάρχει στον κόμβο v αποθηκευμένη πληροφορία.
desc(v,w,h)	:βρίσκει τον κόμβο που έχει απόσταση 2^h από τον κόμβο v, στο μονοπάτι από τον v στον w, αν ο w είναι απόγονος του v και απέχει από τον v περισσότερο από 2^h.

Ορίζουμε επίσης τις ακόλουθες πράξεις:

1. Υπολόγισε το χαμηλότερο παρόντα κόμβο ένα μονοπάτι από ένα απών φύλλο μέχρι τη ρίζα.
2. Εισήγαγε ένα απών φύλλο και ενημέρωσε τη δομή.
3. Υπολόγισε το χαμηλότερο branch-point σε ένα μονοπάτι από ένα παρών φύλλο μέχρι τη ρίζα.
4. Σβήσε ένα παρόν φύλλο και ενημέρωσε τη δομή.

Οι παραπάνω πράξεις πρέπει να γίνονται με τέτοιο τρόπο ώστε να διατηρείται το properness condition. Στη συνέχεια περιγράφεται αναλυτικά οι αλγόριθμοι.

1.3.1 Εύρεση του πιο χαμηλού παρόντα κόμβου

```
Procedure locate(leaf, top, h)
(Υπολογίζει τον χαμηλότερο παρόντα κόμβο στο μονοπάτι από το leaf στο top.
Υποθέτουμε ότι ο κόμβος top είναι παρόν και used, ενώ ο κόμβος leaf δεν είναι
παρόν. Η τιμή που επιστρέφει η locate είναι η κορυφή που ψάχνουμε.)
Begin  if mymin(leaf, top) = ~
       then locate:=top; return
       fi;
       pres:=mymin(leaf, top);        pres:=desc(top, pres, h);
       hl:=hw(leaf, top); hp:=hw(leaf, top);
       if mymin(leaf, top)=-
              (δηλ. δεν υπάρχει branch-point στη διαδρομή αυτή)
       then   if      hp≠hl
              (υπάρχει branch-point)
              then locate(hl, top, h-1)
              (ψάξε στο πάνω μισό για να βρεις το χαμηλότερο παρόν κόμβο)
              else   myfields(pres, hp):=(pres, -)
                     locate:=locate(leaf, hp, h-1)
                     myfriends(pres, hp) = (~, ~)
              fi
       else   if used(hl)
              then locate:=locate(leaf, hl, h-1)
              else locate:=locate(hl, top, h-1)
              fi
       fi
       return; skip
end locate;
```

Η διαδικασία locate μπορεί να περιγραφεί με λόγια ως εξής: Έχω πάντα μπροστά μου ένα υποδέντρο (στην αρχή το υποδέντρο είναι ολόκληρο το δέντρο). Η κόμβος top είναι παρόν. Αν προς τη μεριά του leaf δεν υπάρχει παρόν μονοπάτι (mymin(leaf, top)=~), τότε ο κατώτερος παρόν κόμβος είναι ο top.

Αν υπάρχει παρόν μονοπάτι, ψάχνω να βρω αν υπάρχει ενδιάμεση πληροφορία στο μονοπάτι top-pres όπου pres είναι το πιο αριστερό παρόν φύλλο του υποδέντρου. Αν

δεν υπάρχει ενδιάμεση πληροφορία (myb(leaf, top)=-), βρίσκω τη μέση του μονοπατιού pres-top, έστω hp, και τη μέση του μονοπατιού leaf-top, έστω hl. Αν τα hp και hl δεν ταυτίζονται, τότε το σημείο τομής των δύο μονοπατιών βρίσκεται στο πάνω μέρος του hl και άρα πρέπει να ψάξουμε στο πάνω υποδέντρο του hl. Αν τα hp και hl ταυτίζονται, τότε πρέπει να ψέξουμε στο κάτω υποδέντρο του hl.

Σημείωση: εφόσον δεν υπάρχει ενδιάμεση πληροφορία, δεν υπάρχει branch-point, άρα υπάρχει ένα μόνο παρόν μονοπάτι. Άρα το σημείο που ψάχνουμε είναι το σημείο τομής του μονοπατιού αυτού με το μονοπάτι top-leaf.

Αν όμως ψάξω στο κάτω υποδέντρο, θα πρέπει να αποθηκεύσω πληροφορία στον παρόν κόμβο hp, γιατί ο κόμβος αυτός θα γίνει στην αναδρομική κλήση, top, του νέου υποδέντρου. Αποθηκεύω λοιπόν στον κόμβο hp όλες τις απαραίτητες για τον αλγόριθμο πληροφορίες. Βέβαια, αφού βρεθεί ο ζητούμενος κόμβος, θα πρέπει να σβηστούν οι πληροφορίες αυτές. Αν τώρα υπάρχει ενδιάμεση πληροφορία ανάμεσα στους κόμβους top και leaf, αυτό σημαίνει ότι υπάρχει branch-point στο υποδέντρο αυτό. Αν ο κόμβος hl έχει χρησιμοποιηθεί, αυτό σημαίνει ότι περνάει ένα παρόν μονοπάτι από τον κόμβο αυτό. Συνεπώς, ο κόμβος που ζητάμε θα βρίσκεται στο κάτω υποδέντρο του hl. Αν ο hl δεν έχει χρησιμοποιηθεί, αυτό σημαίνει ότι στο κάτω υποδέντρο του hl, δεν υπάρχει παρόν μονοπάτι. Άρα θα πρέπει να ψάξουμε στο πάνω υποδέντρο.

1.3.2 Εισαγωγή στοιχείων

```
procedure insert(leaf, top, h);
                                    (εισάγει ένα φύλλο στο δέντρο. Θεωρούμε ότι ο
κόμβος τοπ είναι παρόν ενώ ο κόμβος leaf δεν είναι παρόν)
begin  if mymin(leaf, top)=~
       then myfields(leaf, top):=(leaf, -); return
       fi
       pres:=mymin(leaf, top); pres:=desc(top, pres, leaf);
       hl:=hw(leaf, top); hp:=hw(pres, top);
       if mymin(leaf, top)=-
       then   myfields(leaf, hl):=(leaf, -);
              myfields(pres, hp):=(pres, -);
              ub(hl):=ub(hp):=+;
```

```
            insert(hl, top, h-1)
      else  myfields(pres, hp):=(pres, -); ub(hp):=-;
            insert(leaf, hp, h-1)
            myb(leaf, top):=+;
      fi;
      else  if used(hl)
            then  insert(leaf, hl, h-1)
            else  ub(hl):=ub(hp):=+;
                  insert(hl, top, h-1)
            fi;
      fi;
      mymin(leaf, top):=min(leaf, pres); return; skip
end insert;
```

Η διαδικασία insert δεν περιγράφεται με λόγια, αφού είναι σχεδόν ίδια με τη locate.

1.3.3 Εύρεση του πιο χαμηλού branch-point

Procedure lowbranchp(leaf, top, h)

(βρίσκει το χαμηλότερο branch-point στο μονοπάτι από το leaf στο top. Υποθέτουμε ότι το leaf είναι παρόν και ότι δεν είναι το μοναδικό παρόν φύλλο. Η τιμή που επιστρέφεται από τη lowbranchp είναι αυτό το branch-point.

```
Begin if    myb(leaf, hl)=+
            (υπάρχει branch-point κάτω από το hl)
      then  lowbranchp(leaf, hl, h-1)
      elsif yourb(leaf, hl) ≠ ~
                  (το hl είναι από μόνο του branch-point)
      then        hl
      else        lowbranchp(hl, top, h-1)
                  (υπάρχει branch-point πάνω από το hl)
      fi;   return; skip
end lowbranchp;
```

1.3.4 Διαγραφή στοιχείων

Στη διαδικασία delete δεν υπάρχει αναδρομή. Η delete βρίσκει το πιο χαμηλό branch-point. Από αυτό το σημείο βρίσκει την ακολουθία των CS's που το περιέχουν, σβήνει την πληροφορία που σχετίζεται με το φύλλο που σβήστηκε και αν το εναπομείναν παρόν κομμάτι του δέντρου μέσα στο τρέχον CS μειωθεί σε ένα μόνο παρόν μονοπάτι, σβήνει την εσωτερική πληροφορία που είναι τώρα περιττή. Στη συνέχεια μεταφέρει στο επόμενο CS την πληροφορία ότι το προηγούμενο περιέχει ένα μόνο παρόν μονοπάτι. Στη διαδικασία delete, χρησιμοποιούνται οι παρακάτω μεταβλητές:

Br : Αρχικά είναι το πιο χαμηλό branch-point στο μονοπάτι από το φύλλο που σβήστηκε μέχρι τη ρίζα. Κατά τη διάρκεια της εκτέλεσης της διαδικασίας, br είναι το midpoint του εναπομείναντος παρόντος μονοπατιού κατά μήκος του τρέχοντος CS.

top : το top του τρέχοντος CS.

Pres : ένα παρόν φύλλο του τρέχοντος CS. Αρχικά είναι το πιο αριστερό εναπομείναν παρόν φύλλο του πιο χαμηλού branch-point.

laf : το φύλλο του τρέχοντος CS στο μονοπάτι προς το φύλλο που πρόκειται να σβηστεί.

lif : Αρχικά είναι το πιο χαμηλό branch-point στο μονοπάτι από το φύλλο που θα σβηστεί ως το top. Κατά τη διάρκεια της εκτέλεσης της διαδικασίας το lif περιέχει την παλιά τιμή του laf.

```
Procedure delete(leaf);
                (θεωρούμε ότι το leaf δεν είναι το μοναδικό παρόν φύλλο του δέντρου)
begin   br:=lowbranchp(leaf, root, k); j:=rank(br);
        t:=father(br, j);
        pres:=yourmin(leaf, br);
(το br οδηγεί στο leaf από τη μια μεριά, και σε κάποια άλλα φύλλα από την άλλη. Στη
μεριά του leaf, δε μπορεί να οδηγεί σε άλλο φύλλο, γιατί τότε θα υπήρχε branch-point
πιο κάτω. Συνεπώς το πιο αριστερό εναπομείναν φύλλο στο οποίο οδηγεί το br, είναι
τώρα το yourmin(leaf, br).)
        laf:=mymin(leaf, br);
(ο πρώτος από μία αλυσίδα κόμβων στο μονοπάτι του leaf που πρέπει να σβηστούν)
        lif:=br;
        myfields(laf, lif):=(~, -);
        deletable:=true;
```

(παραμένει true για όσο κανένα άλλο branch-point δεν έχει ανιχνευτεί στο μονοπάτι από το t στο pres)

while j<k do

deletable:=deletable and ub(br)=- and ((lb(br)=- and rb(br)=~ and rb(br)=-));

(Δηλ. δεν υπάρχει κανένα άλλο branch-point από το t στο pres)

if deletable then clear(br) fi;

(Αν δεν υπάρχει άλλο branch-point τότε στο br Δεχρειάζεται να υπάρχει ενδιάμεση πληροφορία)

(Στη συνέχεια πηγαίνουμε στο CS βαθμού j+1 που περιέχει το τρέχον CS. Συγκρίνοντας τους βαθμούς των t και pres μπορούμε να αποφασίσουμε αν πρέπει να διαλέξουμε ένα νέο top ή bottom level για να βρούμε αυτό το CS)

if rank(t) > rank(pres)

(το top παραμένει top. Ο pres γίνεται εσωτερικός κόμβος)

then lif:=laf;

laf:=mymin(leaf, laf);

clear(leaf);

if deletable then ub(pres):=-; fi;

br:=pres; pres:=minof(pres);

else

(ο pres παραμένει φύλλο. Ο top γίνεται εσωτερικός κόμβος.)

if mymin(br, t)=laf then mymin(br, t):=pres; fi;

(ο κόμβος laf δεν είναι πλέον παρόν)

if deletable then myb(br, t):=-; fi;

(το τελευταίο branch-point στο LC(t) σβήστηκε)

br:=t; t:=father(t, j+1)

fi;

j:=j+1; od;

end delete;

1.3.5 Εύρεση του προηγούμενου ή του επόμενου

Procedure neighbor(j)

(υπολογίζει το προηγούμενο ή το επόμενο στοιχείο του j στο σύνολο, ανάλογα με το αν η υπογραφή του j στο χαμηλότερο branch-point – ή στο χαμηλότερο παρόν κόμβο αν το j δε βρίσκεται στο σύνολο S) από τον κόμβο j στη ρίζα, είναι ίση με r, ή l.)

```
begin  br:=  if present(j)  then  lowbranchp(j, root, k)
                            else  locate(j, root k)      fi;
       s:=sign(j, br); pres:= if    s=l   then  yourmin(j, br)
                                  else  yourmax(j, br)
                            fi;
       while rank(pres)<k do
             pres:= if s=l  then  minof(pres)  else  maxof(pres)  fi;
       od;
       neighbor:=pres;
end;
```

1.4 Ο O(loglogn) αλγόριθμος για το insert extract-min πρόβλημα.

Έστω k τέτοιο ώστε $m = 2^{2^k} > n$ (δε μπορεί να είναι n=m γιατί θεωρούμε ότι ένα στοιχείο πάντα υπάρχει στο σύνολο). Κατασκευάζουμε το δυαδικό δέντρο ύψους 2^k όπως αυτό περιγράφτηκε στις προηγούμενες παραγράφους. Οι λειτουργίες insert και extract-min μπορούν να γραφούν ως εξής:

```
insert(j)      ->   insert(j, root, k)
extract-min    ->   delete(minof(root))
```

Αν έχουμε στη διάθεσή μας και τη διπλή διασυνδεδεμένη λίστα, τότε μπορούμε να εκτελέσουμε τη λειτουργία predecessor ως εξής:

```
if    present(i)   then  predecessor:=pred(i)
else  j:=neighbor(i); predecessor:= if j<i  then  j  else  pred(j)  fi;
```

1.5 Ο O(loglogn) αλγόριθμος για το meld-split-find πρόβλημα.

Έστω $\{I_1, \ldots, I_\kappa\}$ ένα partition του διαστήματος $\{1,\ldots,n\}$ σε συνεχόμενα κομμάτια και έστω j_r το πιο αριστερό σημείο του διαστήματος I_r. Καλούμε το j_r end-point του I_r. Μπορούμε να αναπαραστήσουμε ένα συγκεκριμένο partition με τα πιο αριστερά σημεία όλων των μελών του. Έτσι, μπορούμε να εκφράσουμε meld(i,j), split(i,j) και find(i) ως εξής:

```
Procedure find(i);
    begin
        if member(i) then i else predecessor(i); fi;
    end find
Procedure meld(i,j);
    Begin
        if      i=successor(j) then   delete(i)
        elsif   j=successor(i) then   delte(j)
        else    error
        fi;
    end meld
procedure split(i,j)
    begin
        if predecessor(j)=i   and   not(present(j))
        then   insert(j)
        fi;
    end split
```

1.6 Σχόλια

Έστω S το σύνολο στο οποίο θέλουμε να κάνουμε δυναμικό ψάξιμο. Αν θεωρήσουμε ότι τα στοιχεία του S είναι φύλλα ενός δέντρου, τότε ο χρόνος για τη λύση του προβλήματος είναι αυτός που απαιτείται για τη διαπέραση του μονοπατιού. Για ένα δυαδικό δέντρο, η προφανής λύση είναι O(logn) όπου n είναι ο αριθμός των στοιχείων του S. Ο van E. Boas εισήγαγε για πρώτη φορά την ιδέα για χρήση δυαδικού ψαξίματος κατά μήκος του μονοπατιού. Δυαδικό ψάξιμο όμως, μπορεί να γίνει μόνο όταν τα φύλλα του δέντρου είναι τα στοιχεία του σύμπαντος U (στη δυναμική περίπτωση, όταν δηλαδή υποστηρίζονται οι πράξεις insert και delete). Έτσι ο χρόνος που προκύπτει για μία οποιαδήποτε πράξη από αυτές που αναφέρθηκαν παραπάνω είναι O(loglogU). Οι αλγόριθμοι του van Emde Boas είναι αρκετά πολύπλοκοι. Δυαδικό ψάξιμο στο μονοπάτι μπορεί να γίνει χωρίς να χρειάζονται τα κανονικά υποδέντρα (CS's) που μάλλον αποπροσανατολίζουν τον αναγνώστη. Το πρόβλημα αυτό επισημάνθηκε τα επόμενα χρόνια από αρκετούς επιστήμονες και εμφανίστηκαν μερικές παραλλαγές των παραπάνω αλγορίθμων, πολύ πιο κατανοητές.

2. Γρήγορες δενδρικές δομές με μικρό χώρο

2.1 Ορισμοί και θεωρητικό υπόβαθρο

Στο κεφάλαιο αυτό παρουσιάζονται λύσεις στο πρόβλημα του ψαξίματος που βασίζονται στα tries. Trie είναι στην πραγματικότητα και η δομή του van Emde Boas που παρουσιάστηκε στο προηγούμενο κεφάλαιο. Τα tries που θα παρουσιαστούν στο κεφάλαιο αυτό παρουσιάστηκαν από τον Willard [3,4] και επιχειρούν να μειώσουν τον απαιτούμενο χώρο . Υπενθυμίζουμε ότι ο χώρος που απαιτεί η δομή του van Emde Boas είναι O(U) όπου U είναι το μέγεθος του σύμπαντος.

Καλούμε S το σύνολο των N records. Οι λειτουργίες που μας ενδιαφέρουν είναι οι παρακάτω:

1. MEMBER(k)
2. SUCCESSOR(k)
3. PREDECESSOR(k)

Θα λέμε ότι ένα trie έχει μέγεθος (h, b) αν έχει ύψος h και branching factor b. Όλα τα κλειδιά σε ένα συμβατικό trie μεγέθους (h, b) μπορούμε να τα δούμε σα θετικούς ακεραίους μικρότερους από b^h. Από εδώ και στο εξής, η σειρά k_1, k_2, ..., k_h θα δηλώνει την h-digit αναπαράσταση ενός ακεραίου k, όταν αυτός γράφεται με βάση b. Σε ένα trie μεγέθους (h,b) ο συμβολισμός αυτός σημαίνει ότι το κλειδί k είναι αποθηκευμένο στο k_h παιδί του k_{h-1} παιδιού του...του k_1 παιδιού της ρίζας του trie. Το σύμβολο $child_u(D)$ δηλώνει το δείκτη που είναι αποθηκευμένος στον εσωτερικό κόμβο u και δείχνει στο D-οστό παιδί του u.

Θα θεωρήσουμε την πολυπλοκότητα ανάκτησης ενός trie συνάρτηση του μεγέθους του. Ο χρόνος απάντησης για ένα query τύπου 1 είναι διαφορετικός από τους χρόνους για τα queries τύπου 2, και 3 αφού στην πρώτη περίπτωση χρειάζεται να ξοδέψουμε Θ(1) μονάδες χρόνου σε κάθε κόμβο, ενώ στη δεύτερη περίπτωση χρειάζεται να ξοδέψουμε Θ(b) μονάδες χρόνου. Έτσι, για τα queries PREDECESSOR και SUCCESSOR η συνολική πολυπλοκότητα ανάκτησης στη χειρότερη περίπτωση για τα συμβατικά tries μεγέθους (h, b) είναι Θ(hb). Ο χρόνος αυτός μπορεί να βελτιωθεί αν αλλάξει λίγο η δομή δεδομένων. Ο χρόνος αυτός για το p-fast trie είναι O(h+logb) και χρησιμοποιείται ασυμπτωτικά ο ίδιος χώρος με ένα συμβατικό trie.

2.2 p-fast tries

Στα tries που θα παρουσιαστούν στο κεφάλαιο αυτό, ο αριθμός των εσωτερικών κόμβων είναι όσο το δυνατό μικρότερος, και αυτό επιτυγχάνεται ως εξής: Κάθε εσωτερικός κόμβος του trie είναι πρόγονος στοιχείων του S.

Οι εσωτερικοί κόμβοι στα p-fast tries περιέχουν, εκτός από τους δείκτες προς τα παιδιά τους, τα επόμενα επιπλέον πεδία:

LOWKEY(υ): είναι ένας δείκτης στο μικρότερο φύλλο που είναι απόγονος του υ.

HIGHKEY(υ): είναι ένας δείκτης στο μεγαλύτερο φύλλο που είναι απόγονος του υ.

INNERTREE(υ): είναι ένα δυαδικό δέντρο που έχει βάθος στη χειρότερη περίπτωση O(logb) και αναπαριστά το σύνολο των ψηφίων D για τα οποία $child_D(υ)≠NULL$.

Επίσης σε κάθε φύλλο του trie υπάρχουν δύο δείκτες, ένας στο προηγούμενο και ένας στο επόμενο φύλλο του trie.

Ορίζουμε CLOSEMATCH(K) το query που:

1. Επιστρέφει τη διεύθυνση του record στο οποίο είναι αποθηκευμένο το κλειδί K στο trie T (Αν το trie όντως περιέχει το K)
2. Αλλιώς επιστρέφει τη διεύθυνση είτε του PREDECESSOR(K) είτε του SUCCESSOR(K).

Ο αλγόριθμος του CLOSEMATCH(K) αποτελείται από 3 βήματα. Στο πρώτο βήμα ξεκινάμε από τη ρίζα του p-fast trie και διασχίζουμε προς τα κάτω δέντρο περνώντας από όλους τους προγόνους του K. Αν το K πράγματι περιέχεται στο trie, η διαπέραση του δέντρου θα καταλήξει στο K. Αλλιώς κάποια στιγμή θα φτάσουμε σε ένα NULL δείκτη. Στην περίπτωση αυτή ο αλγόριθμος θα τελειώσει εκτελώντας δύο επιπλέον βήματα. Μέχρι αυτό το σημείο το κόστος είναι O(h).

Το βήμα 2, το πρώτο από τα δύο επιπλέον βήματα χρησιμοποιεί το πεδίο INNERTREE. Έστω ότι το βήμα 1 σταμάτησε σε ένα κόμβο βάθους i-1. Ψάχνουμε στο INNERTREE(υ) για να βρούμε το μικρότερο ψηφίο D που είναι μεγαλύτερο από το K_i και για το οποίο ισχύει $child_u(D)≠NULL$, ή για να βρούμε το μεγαλύτερο ψηφίο E που είναι μικρότερο από το K_i και για το οποίο ισχύει $child_u(E)≠NULL$. Το ψάξιμο στο INNERTREE κοστίζει O(logb).

Το επόμενο βήμα είναι να πάμε στο LOWKEY(D) ή στο HIGHKEY(E) ανάλογα με το αν στο βήμα 2 βρήκαμε το D ή το E.

Είναι προφανές από τα παραπάνω ότι ο αλγόριθμος CLOSEMATCH χρειάζεται O(h +logb) χρόνο. Αυτός είναι και ο χρόνος για τους αλγορίθμους MEMBER, SUCCESSOR και PREDECESSOER. Το trie χρειάζεται χώρο O(hbN).

Αν θέσουμε $h=\lceil \sqrt{\log M} \rceil$ και $b=\lceil 2^{\sqrt{\log M}} \rceil$ τότε οι αλγόριθμοι για τις παραπάνω πράξεις κοστίζουν $O(\sqrt{\log M})$ και ο χώρος του trie είναι $O(N\sqrt{\log M}\ 2^{\sqrt{\log M}})$. Ο χώρος αυτός μειώνεται σε O(N) με τα q-fast tries, η περιγραφή των οποίων ακολουθεί.

2.3 q-fast tries.

Έστω S ένα αρχικό σύνολο, S^* ένα διατεταγμένο σύνολο από κλειδιά $0 = K_1^* < K_2^* < \ldots < K_L^* < M$, S_i το υποσύνολο $\{K \in S : K_{i1}^* \le K < K_{i+1}^*\}$ για i<L, και S_L το υποσύνολο $\{K \in S : K \ge K_L^*\}$. Λέμε ότι το σύνολο S^* δημιουργεί ένα c-partition του συνόλου S, αν και μόνο αν κάθε σύνολο S_i έχει πληθάριθμο ανάμεσα σε c και 2c-1.

Ένα q-fast trie μεγέθους (h, b, c) είναι μία δομή δεδομένων που αναπαριστά το σύνολο S και αποτελείται από 2 βασικές δομές, το πάνω και το κάτω κομμάτι. Το πάνω κομμάτι είναι ένα p-fast trie μεγέθους (h, b) που αναπαριστά κάποιο σύνολο S^*, που αποτελεί ένα c-partition του S. Το κάτω κομμάτι είναι ένα δάσος από 2-3 δέντρα και το δέντρο T_i αναπαριστά το σύνολο S_i. Το φύλλο του κλειδιού K_i του trie T έχει ένα δείκτη στο δέντρο T_i. Επίσης τα φύλλα των 2-3 δέντρων σχηματίζουν μία διπλή διασυνδεδεμένη λίστα.

Λήμμα 2.1. Ο αλγόριθμος για την πράξη CLOSEMATCH σε ένα q-fast trie μεγέθους (h, b, c) είναι O(h+logb+logc) και ο χώρος που καταλαμβάνει το trie είναι O(N(1+hb/c)) όταν αναπαριστά ένα σύνολο από N records.

Απόδειξη: Αρχικά ψάχνουμε στο πάνω κομμάτι για να βρούμε το κλειδί K_i^* που είναι το predecessor κλειδί του K στο σύνολο S^*. Αυτό κοστίζει O(h+logb) χρόνο. Στη συνέχεια ψάχνουμε στο δέντρο T_i. Αυτό κοστίζει O(logc) χρόνο. Συνεπώς ο συνολικός αλγόριθμος κοστίζει O(h+logb+logc) χρόνο.

Όπως προκύπτει από την ανάλυση των p-fast tries, ο χώρος του άνω κομματιού είναι O(Nhb/c). Το κάτω κομμάτι καταλαμβάνει χώρο O(N) και συνεπώς ο συνολικός χώρος είναι O(N(1+hb/c)). □

Όπως και στα p-fast tries, αν θέσουμε h=$\lceil \sqrt{\log M} \rceil$ και b=$\lceil 2^{\sqrt{\log M}} \rceil$ και c=hb τότε οι αλγόριθμοι αλγορίθμους MEMBER, SUCCESSOR και PREDECESSOR χρειάζονται O($\sqrt{\log M}$) χρόνο, ενώ ο χώρος είναι O(N).

2.4 Δυναμικές λειτουργίες στα p και q-fast tries.

Θα εξετάσουμε τώρα τις λειτουργίες insert και delete στα p-fast και q-fast tries.
Για τα p-fast tries θα δείξουμε ότι αλλαγές που πρέπει να γίνουν στα INNERTREE πεδία αμέσως μετά το σβήσιμο, ή την εισαγωγή ενός κλειδιού κοστίζουν O(h+logb). Εισάγουμε έναν AVL αλγόριθμο για να εξασφαλίσουμε λογαριθμικό ύψος στα INNERTREE και λογαριθμική πολυπλοκότητα για εισαγωγή και σβήσιμο σε αυτά. Ο αλγόριθμος delete αποτελείται από τέσσερα βήματα και μπορεί να περιγραφεί ως εξής:

Delete(k) {σβήνει το κλειδί k από το p-fast trie}

Βήμα 1: Βρες το κλειδί k που πρόκειται να σβηστεί από το p-fast trie. Έστω k^- ο προηγούμενος και k^+ ο επόμενος του k. Σβήνουμε το k από τη διατεταγμένη λίστα των φύλλων έτσι ώστε οι k^- και k^+ να γίνουν γειτονικοί.

Βήμα 2: Για κάθε πρόγονο υ του k κάνουμε το εξής:

if LOWKEY(υ)=k then LOWKEY(υ)← k^+

if HIGHKEY(υ)=k then HIGHKEY(υ)← k^-

Βήμα 3: αποδέσμευσε το χώρο μνήμης για το record που αποθηκεύει το κλειδί k και για κάθε πρόγονο του k για τον οποίο ισχύει LOWKEY(υ)=k^+ και HIGHKEY(υ)=k^-.

Βήμα 4: Έστω υ ο υψηλότερος κόμβος του οποίου ο χώρος μνήμης αποδεσμεύτητκε από το βήμα 3 και έστω f ο πατέρας του υ. Σβήσε από το inertree το ψηφίο που συνδέει τον f με τον υ χρησιμοποιώντας τον AVL αλγόριθμο.

Τα πρώτα 3 βήματα του αλγορίθμου χρειάζονται χρόνο O(h) ενώ το τέταρτο βήμα χρειάζεται χρόνο O(logb). Άρα μπορούμε να σβήσουμε ένα record από ένα trie μεγέθους (h, b) σε χρόνο O(h +logb). Ο αλγόριθμος insert είναι ανάλογος με τον delete και έχει την ίδια πολυπλοκότητα. Το ίδιο ισχύει για τους αλγόριθμους insert και delete των q-fast tries όπου είναι εύκολο να αποδειχθεί ότι η πολυπλοκότητα είναι O(h+logb+logc).

2.5 x-fast tries

Το x-fast trie είναι ένα δυαδικό trie ύψους h, όπου τα records αποθηκεύονται στο επίπεδο των φύλλων. Αν v είναι ένας κόμβος σε ύψος h, τότε όλα τα φύλλα που είναι απόγονοι του v έχουν τιμές ανάμεσα στις ποσότητες $(i-1)2^j+1$ και $i2^j$, για κάποιον ακέραιο i. Καλούμε την τιμή i identifier του v, και τη συμβολίζουμε σαν ID(v). Ο αριθμός count(v) είναι ο αριθμός των στοιχείων του S που είναι απόγονοι του v. Αν ένας κόμβος v δεν έχει αριστερό (δεξί) παιδί, τότε descendant(v) είναι ένας δείκτης στο φύλλο με το μικρότερο (αντίστοιχα μεγαλύτερο) κλειδί, που βρίσκεται στο υποδέντρο με ρίζα τον v. Τα φύλλα του x-fast trie ανήκουν σε μία διπλή διασυνδεδεμένη λίστα. Για να εξοικονομήσουμε χώρο, ένας κόμβος v υπάρχει το x-fast trie μόνο όταν count(v)>0.

Το x-fast trie περιλαμβάνει και ένα level-search structure (LSS) που χρησιμοποιεί κάποιες ιδιότητες που παρουσιάστηκαν από τους Fredman, Komolos και Szemeredi [8]. Αυτοί παρουσίασαν μία δομή δεδομένων που χρησιμοποιεί Θ(n) χώρο και εκτελεί find-queries σε Θ(1) χρόνο στη χειρότερη περίπτωση, για ένα σύνολο από n θετικούς ακεραίους που είναι μικρότεροι από την τιμή M. Το x-fast trie χρησιμοποιεί h+1 αντίγραφα της δομής αυτής. Το j-οστό αντίγραφο, το οποίο συμβολίζεται LSS(j) αναπαριστά το σύνολο των κόμβων του trie με ύψος j, χρησιμοποιώντας τους identifiers των κόμβων σαν κλειδιά. Ένας κόμβος v σε ύψος j θα λέγεται πρόγονος ενός ακεραίου K αν και μόνο αν $\lfloor K/2^j \rfloor$=ID(v). Ο συμβολισμός bottom(K) δηλώνει τον πιο χαμηλό πρόγονο του K που υπάρχει στο trie. Για να βρούμε τον κόμβο bottom(K) κάνουμε δυαδικό ψάξιμο στο μονοπάτι του trie, χρησιμοποιώντας τα level search structures για να βρούμε σε O(1) τον κόμβο του μονοπατιού που θέλουμε κάθε φορά να επισκεφτούμε. Συνεπώς για έναν αριθμό K, ο κόμβος bottom(K) βρίσκεται σε χρόνο O(loglogM), αφού το μήκος του μονοπατιού είναι logM. Το descendant field του bottom(K) δείχνει στον predecessor ή τον successor του K. Χρησιμοποιώντας τη διπλή διασυνδεδεμένη λίστα, από τον ένα, βρίσκουμε τον άλλο. Συνεπώς οι αλγόριθμοι για τις πράξεις MEMBER, SUCCESSOR και PREDECESSOR χρειάζονται Θ(loglogM) χρόνο. Όσον αφορά τώρα το χώρο, ένα trie ύψους logM που έχει N φύλλα, δεν έχει περισσότερους από NlogM κόμβους. Τα level search structures χρησιμοποιούν γραμμικό χώρο, και συνεπώς ο συνολικός χώρος του x-fast trie είναι O(NlogM).

2.6 y-fast tries

Το y-fast trie προκύπτει από το x-fast trie με τον ίδιο τρόπο που προέκυψε το q-fast trie από το p-fast trie. Αυτό δηλαδή που γίνεται, είναι μία μείωση του ύψους του x-fast trie και κάθε φύλλο του γίνεται ρίζα ενός 2-3 δέντρου. Με τον τρόπο αυτό ο χώρος μειώνεται σε O(N) όπως ακριβώς και στο q-fast trie.

2.7 Σχόλια

Βασικό χαρακτηριστικό όλων των δενδρικών δομών που παρουσιάστηκαν στο κεφάλαιο αυτό είναι ότι φύλλα των δέντρων είναι τα στοιχεία του συνόλου. Σε μία δομή αυτής της μορφής αυτό που πρέπει να κάνουμε είναι να διαπεράσουμε τη δομή από τη ρίζα μέχρι τα φύλλα. Το μονοπάτι έχει μήκος logU όπου U είναι το σύμπαν. Η προφανής λύση λοιπόν απαιτεί χρόνο O(logU). Ο Van Emde Boas κατόρθωσε να επιτύχει δυαδικό ψάξιμο στο μονοπάτι και να λύσει το πρόβλημα στη δυναμική περίπτωση. Καλές λύσεις που χρειάζονται λιγότερο χώρο, είναι τα p και q-fast tries. Τα τελευταία μάλιστα έχουν γραμμικό χώρο, και επιτυγχάνουν χρόνο O($\sqrt{\log U}$). Τα x και y-fast tries, στηρίζονται και αυτά στο δυαδικό ψάξιμο πάνω το μονοπάτι, όπως και η δομή του Van Emde Boas. Τα y-fast tries χρειάζονται γραμμικό χώρο. Το πρόβλημα είναι ότι δεν είναι δυναμικά, δεν υποστηρίζουν δηλαδή insert και delete στον ίδιο χρόνο. Αυτό είναι και το τίμημα που πληρώνουν για το μικρό χώρο που απαιτούν σε σχέση με τη δομή του Van Emde Boas.

3. Τα fusion δέντρα

3.1 Εισαγωγή

Η εργασία αυτή [6] εισάγει μία νέα δομή για ψάξιμο, το fusion δέντρο. Τα δέντρα αυτά οδηγούν σε βελτιωμένους αλγορίθμους για sorting και searching που ξεπέρασαν τα γνωστά σύμφωνα με τη θεωρία (μέχρι εκείνη τη στιγμή) lower bounds. Ορίζουμε το ψάξιμο έτσι ώστε ένα ανεπιτυχές ψάξιμο να επιστρέφει το predecessor στοιχείο.
Σύμφωνα με το information theoretic bound, η ταξινόμηση N αριθμών απαιτεί NlogN συγκρίσεις στη χειρότερη περίπτωση. Αυτό, ισχύει υπό την υπόθεση ότι η RAM μηχανή υποστηρίζει πρόσθεση, αφαίρεση, πολλαπλασιασμό και σύγκριση με το 0 (αλλά δεν υποστηρίζει διαίρεση και bitwise boolean operations). Αν προστεθούν η διαίρεση και τα bitwise boolean operations, τότε μπορεί να προκύψει ένας γραμμικός αλγόριθμος. Ο αλγόριθμος αυτός ωστόσο, δημιουργεί κατά τη διάρκειά του αριθμούς με μήκος N^2 φορές επί το μήκος του μεγαλύτερου αριθμού της εισόδου, θεωρώντας παράλληλα ότι κάθε πράξη γίνεται σε O(1) χρόνο.
Στην εργασία αυτή θεωρούμε ότι το μήκος της λέξης είναι b bits και κάθε ένας από τους N αριθμούς της εισόδου είναι ένας θετικός ακέραιος μικρότερος από 2^b. Επιπλέον, θεωρούμε επιθυμητό ο αλγόριθμος να χρησιμοποιεί O(N) λέξεις μνήμης. Ο αριθμός των στοιχείων που υπάρχουν στη δομή δεν υπερβαίνει ποτέ το 2^b, που είναι το μέγεθος του σύμπαντος.
Τα fusion δέντρα επιτυγχάνουν δυναμικά search operations σε κατανεμημένο χρόνο O(logN/loglogN) για κάθε operation σε μία RAM που έχει μήκος λέξης b bits και υποστηρίζει πρόσθεση, αφαίρεση, πολλαπλασιασμό σύγκριση και bitwise AND operations. Χρησιμοποιώντας fusion δέντρα, κατασκευάζουμε έναν αλγόριθμο ταξινόμησης με worst case χρόνο O(NlogN/loglogN) που χρειάζεται γραμμικό χώρο. Επιπλέον, αν επιτρέψουμε randomization και διαίρεση το δυναμικό ψάξιμο και η ταξινόμηση μπορούν να γίνουν σε χρόνους $O(\sqrt{\log N})$ και $O(N\sqrt{\log N})$ αντίστοιχα.

3.2 Συμβολισμοί

Συμβολίζουμε με bin(α_1, α_2, ...) το άθροισμα $2^{a_1}+2^{a_2}+....$ όπου τα α_i είναι μη αρνητικά. Για να αναφερθούμε στα bits ενός δυαδικού αριθμού, μετράμε τις θέσεις

των bits με το μηδέν να αντιστοιχεί στη θέση του λιγότερο σημαντικού bit. Θεωρούμε ότι ο πολλαπλασιασμός δημιουργεί ένα αποτέλεσμα που χωράει σε δύο λέξεις. Πολλαπλασιάζοντας έναν αριθμό x με την ποσότητα bin(b-r) επιτυγχάνουμε shift του x κατά r θέσεις προς τα αριστερά.

Δοσμένου ενός συνόλου S και ενός αριθμού x, καλούμε $rank_s(x)$ την τιμή $|\{t|t\in S, t\leq x\}|$

3.3 Overview

Μπορούμε να πούμε ότι το fusion tree μοιάζει με ένα b-tree στο οποίο:

α) ο βαθμός B ενός εσωτερικού κόμβου είναι συνάρτηση του N και

β) όταν κάνουμε ψάξιμο, το παιδί που πρέπει να ακολουθήσουμε υπολογίζεται σε O(1) χρόνο, ανεξάρτητα από το B

Αυτό αρκεί για να επιτύχουμε o(logN) χρόνο ψαξίματος. Παρόλα αυτά, η ενημέρωση ενός εσωτερικού κόμβου θα απαιτεί χρόνο B^4. Για να διατηρούμε υπό έλεγχο το κατανεμημένο κόστος των update operations, τροποποιούμε τη δομή έτσι ώστε να είναι ρίζες weight balanced binary search trees, καθένα από τα οποία έχει μέγεθος περίπου B^4. Όταν το μέγεθος ενός από αυτά τα δέντρα γίνει πολύ μεγάλο, το δέντρο χωρίζεται στα δύο και η ρίζα του (το φύλλο του B-tree) χωρίζεται και αυτή στα δύο. Αυτό μειώνει το κατανεμημένο κόστος των updates ενώ ο χρόνος ψαξίματος αυξάνει κατά μία ποσότητα logB. Αν μπορέσουμε να εξασφαλίσουμε ότι $logB=\Theta(loglogN)$ τότε έχουμε ένα search tree στο οποίο τα operations γίνονται σε O(logN/loglogN) κατανεμημένο χρόνο.

Οι αλγόριθμοι που θα παρουσιαστούν χρειάζονται κάποιες σταθερές, οι τιμές των οποίων εξαρτώνται από το μήκος της λέξης, b. Θεωρούμε ότι αυτές οι σταθερές δίνονται και δεν επιβαρυνόμαστε με τον υπολογισμό τους.

Το βασικό χαρακτηριστικό του fusion tree είναι ότι οι κόμβοι αναπαρίστανται με τέτοιο τρόπο ώστε:

1. Όταν κάνουμε ψάξιμο μπορούμε να βρούμε το σωστό παιδί σε constant time.
2. Μπορούμε να ενημερώσουμε ένα κόμβο σε χρόνο B^4.

Έστω ένας κόμβος ενός B-tree που αποτελείται από k κλειδιά ($B/2 \leq k \leq B$) τα οποία δίνονται από το σύνολο $S=\{u_1, \ldots, u_k\}$. για να βρούμε το σωστό παιδί ενός κόμβου όταν κάνουμε ψάξιμο για ένα στοιχείο u, χρειάζεται να βρούμε την τιμή $rank_s(u)$.

Αυτή η τιμή πρέπει να υπολογίζεται σε constant time. Βασική για την αναπαράσταση ενός κόμβου, είναι η έννοια compressed key representation (συμπιεσμένη αναπαράσταση κλειδιού) η οποία παρουσιάζεται στη συνέχεια.

3.4 Compressed key representation

Δοσμένου ενός συνόλου $S=\{u_1, \ldots, u_k\}$ που αποτελείται από αριθμούς μήκους b (bits) θα κατασκευάσουμε ένα σύνολο B(S) από διαφορετικές θέσεις bit, και ένα σύνολο $K(S)=\{\hat{u}_1,\ldots,\hat{u}_k\}$ από αριθμούς μήκους r, όπου $r=|B(S)|\leq k-1$.

Ο ορισμός του συνόλου B(S) είναι αναδρομικός. Αν $|S|=1$, τότε το B(S) είναι το κενό σύνολο. Αλλιώς, έστω p η πιο σημαντική θέση bit που διαφοροποιεί δύο από τους αριθμούς του συνόλου S. Έστω $S_0=\{u_1, \ldots, u_g\}$ το σύνολο που αποτελείται από τα στοιχεία του S που έχουν μηδέν στη θέση p και $S_1=\{w_1, .., w_h\}$ το σύνολο που αποτελείται από τα στοιχεία του S που έχουν άσσο στη θέση αυτή. Τότε $B(S)=\{p\}\cup B(S_0)\cup B(S_1)$.

Ορίζουμε τώρα $\hat{u}_i$ να είναι το αποτέλεσμα του σβησήματος από το u_i όλων των bits εκτός από εκείνων που καταλαμβάνουν θέσεις που περιέχονται στο σύνολο B(S). Ειδικότερα, έστω $c_1<c_2<\ldots<c_r$ τα στοιχεία του B(S) ταξινομημένα. Τότε το bit στη θέση j-1 του $\hat{u}_i$ είναι το j-οστό bit του u_i. Καλούμε το $\hat{u}_i$ compressed key representative του u_i και ορίζουμε $K(S)=\{\hat{u}_1,\ldots,\hat{u}_k\}$. Παρατηρούμε ότι η διαδικασία της συμπίεσης διατηρεί τη σειρά ανάμεσα στα στοιχεία του S.

Δοσμένου ενός τυχαίου b-bit αριθμού u ο οποίος μπορεί και να μην ανήκει στο S, ορίζουμε $\hat{u}(S)$ το αποτέλεσμα της εξαγωγής ακριβώς εκείνων των bits του u που καταλαμβάνουν θέσεις του B(S). Το ακόλουθο λήμμα παρουσιάζει μία σημαντική ιδιότητα του compressed key representation.

Λήμμα 3.1: Έστω $c_1<c_2<\ldots<c_r$ τα στοιχεία του B(S) ταξινομημένα και ορίζουμε $c_0=-1$ και $c_{r+1}=b$. Έστω $\hat{u}_i$ ένα τυχαίο συμπιεσμένο κλειδί του K(S) και υποθέτουμε για ένα τυχαίο b-bit αριθμό $u\neq u_i$, η πιο σημαντική θέση bit στην οποία διαφέρουν τα $\hat{u}(S)$ και $\hat{u}_i$ είναι η θέση m-1 (με άλλα λόγια, οι αριθμοί u και u_i συμφωνούν στις θέσεις $c_{m+1}, \ldots, c_r$ αλλά διαφωνούν στη θέση c_m. Αν $\hat{u}(S)=\hat{u}_i$ τότε ορίζουμε m=0). Υποθέτουμε ότι η πιο σημαντική θέση p στην οποία διαφέρουν τα u και u_i ικανοποιεί

τη σχέση $p>c_m$. τότε το $rank_S(u)$ καθορίζεται από το διάστημα (c_{j-1}, c_j) που περιέχει το p, μαζί με τη σχετική σειρά ανάμεσα στα u και u_i.

Απόδειξη: αρχικά παρατηρούμε ότι η υπόθεση $p>c_m$. μαζί με το γεγονός ότι τα u και u_i συμφωνούν στις θέσεις $c_{m+1}, \ldots, c_r$, υπονοεί ότι το p δε μπορεί να είναι κανένα από τα c_i. Θα προχωρήσουμε την απόδειξη με επαγωγή στο $k=|S|$. Αν $k=1$ η απόδειξη είναι προφανής. Έστω τώρα $p\in(c_r, c_{r+1})$. Από τον ορισμό του c_r, όλα τα στοιχεία του S συμφωνούν στις θέσεις που είναι πιο σημαντικές από τη c_r, και συνεπώς το $rank_S(u)$ είναι 0 ή k, ανάλογα με το αν $u<u_i$ ή $u>u_i$. Έστω τώρα ότι $p<c_r$. Τότε, τα u και u_i συμφωνούν στις θέσεις μέχρι τη c_r (ξεκινώντας από την πιο σημαντική). Χωρίς βλάβη της γενικότητας υποθέτουμε ότι τα u και u_i έχουν μηδέν στη θέση c_r. Τότε, $u_i \in S_0=\{u_1, \ldots, u_g\}$ και $rank_S(u)\le g$ αφού $u\le w_1, \ldots, w_h$. Προκύπτει ότι $rank_S(u)=rank_{S_0}(u)$ και προχωρούμε στην περίπτωση όπου $S=S_0$ εφαρμόζοντας την υπόθεση της επαγωγής. Παρατηρούμε ότι $B(S_0) \subseteq B(S)$ και συνεπώς η ταξινομημένη ακολουθία $c_1' < c_2' < \ldots$ που αντιστοιχεί στα στοιχεία του $B(S_0)$, είναι μία υποακολουθία της ακολουθίας $c_1<c_2<\ldots$Η πιο σημαντική θέση c_q' στην οποία διαφέρουν τα u και u_i σίγουρα ικανοποιεί τη σχέση $c_q' \le c_m$. Συνεπώς, $p > c_q'$. Εφαρμόζοντας την υπόθεση της επαγωγής για το S_0 και παρατηρώντας ότι τα διαστήματα (c_{j-1}', c_j') είναι εσωτερικά των (c_{j-1}, c_j), προκύπτει ότι το λήμμα ισχύει. □

Το παραπάνω λήμμα μας προτρέπει να αντικαταστήσουμε τον υπολογισμό του $rank_S(u)$ με τον υπολογισμό του $rank_{K(S)}(\hat{u}(S))$ όπως θα δούμε παρακάτω. Υπολογίζοντας το $rank_{K(S)}(\hat{u}(S))$, βρίσκουμε τον predecessor $\hat{u}_j$ ή τον successor $\hat{u}_{j+1}$ του $\hat{u}(S)$ στο K(S) (μία από τις δύο ποσότητες $\hat{u}_j$ και $\hat{u}_{j+1}$ μπορεί να μην υπάρχει). Στη συνέχεια συγκρίνουμε το u με τα u_j, u_{j+1}. Αν ισχύει $u_j\le u\le u_{j+1}$ τότε έχουμε τελειώσει. Αλλιώς, διαλέγουμε $\hat{u}_i$ (i=j ή j+1) να είναι εκείνο από τα $\hat{u}_j$ και $\hat{u}_{j+1}$ που έχει το μεγαλύτερο κοινό prefix με το $\hat{u}(S)$. Επειδή $u<u_j$ ή $u>u_{j+1}$, πρέπει να είναι $p>c_m$ και συνεπώς το λήμμα 1 μπορεί να εφαρμοστεί για τον υπολογισμό του $rank_S(u)$, από τη στιγμή που ξέρουμε το διάστημα (c_f, c_{f+1}) που περιέχει το p.

Μια εναλλακτική περιγραφή του συνόλου B(S) είναι η εξής: έστω ότι τα u_i's είναι ταξινομημένα έτσι ώστε $u_1<u_2<\ldots$ Έστω d_i η πιο σημαντική θέση bit στην οποία διαφέρουν τα u_i, u_{i+1}. Τότε, $B(S)=\{d_1, d_2, \ldots, d_{k-1}\}$.

Η δυσκολία με τη χρήση των συμπιεσμένων κλειδιών και με το μετασχηματισμό του u σε $\hat{u}(S)$ (σε constant time) είναι ότι δεν υπάρχει κάποιος προφανής τρόπος για την επανατοποθέτηση των χρήσιμων bits σε διαδοχικές θέσεις. Για να διατηρούνται όμως οι επιθυμητές ιδιότητες δε χρειάζεται οι νέες θέσεις να είναι διαδοχικές. Αρκεί ένας μετασχηματισμός που επανατοποθετεί τα χρήσιμα bits έτσι ώστε αυτά να βρίσκονται σε ένα σχετικά στενό διάστημα σε σύγκριση με το μήκος της λέξης. Τα ενδιάμεσα κενά μπορούμε να τα γεμίσουμε με μηδενικά. Αυτός ο μετασχηματισμός μπορεί να επιτευχθεί με πολλαπλασιασμό και bitwise AND operations.

3.5 Fusion

Θα δούμε τώρα τη μορφή ενός κόμβου του B-tree. Έστω $u_1<u_2<\ldots<u_k$ τα κλειδιά που είναι αποθηκευμένα στον κόμβο ($k\leq B$). Έστω $S=\{u_1, \ldots, u_k\}$ και $B(S)=\{c_1, \ldots, c_r\}$. Στον κόμβο του B-tree είναι επίσης αποθηκευμένη η ποσότητα $C=bin(c_1, c_2, \ldots)$ μαζί με τις ποσότητες $M=bin(m_1, m_2,\ldots)$ και $D=bin(c_1+m_1, c_2+m_2, \ldots)$ που χρησιμεύουν για τον υπολογισμό του $\hat{u}(S)$. Παρόντα επίσης στον κόμβο είναι τα u_i σε ταξινομημένη σειρά, καθώς και τα $\hat{u}_i$ σε ταξινομημένη σειρά.

Προς το παρόν θα θερήσουμε ότι οι ποσότητες $rank_{K(S)}(\hat{u}(S))$, $rank_{B(S)}(m)$ (όπου m είναι ένας ακέραιος στο διάστημα [0, b]) και msb(u,υ) (το πιο σημαντικό bit στο οποίο διαφέρουν οι αριθμοί u, υ) μπορούν να υπολογιστούν σε constant time.

Όπως αναφέρθηκε στην προηγούμενη παράγραφο, μετά τον υπολογισμό του $j=rank_{K(S)}(\hat{u}(S))$ συγκρίνουμε το u με τα u_j και u_{j+1}. Αν θεωρήσουμε ότι το u δεν πέφτει μέσα στο διάστημα $[u_j, u_{j+1}]$, Αλλιώς, βρίσκουμε εκείνο από τα $\hat{u}_j$ και $\hat{u}_{j+1}$ που έχει το μεγαλύτερο κοινό prefix με το $\hat{u}(S)$ συγκρίνοντας τις ποσότητες $\hat{u}(S)\,\mathrm{XOR}\,\hat{u}_j$ και $\hat{u}(S)\,\mathrm{XOR}\,\hat{u}_{j+1}$ (η μικρότερη τιμή αντιστοιχεί στο μεγαλύτερο prefix). Αν υποθέσουμε ότι $\hat{u}_h$ (h=j ή j+1) έχει το μεγαλύτερο κοινό prefix με το $\hat{u}(S)$, υπολογίζουμε το $m=msb(u,u_h)$ και βρίσκουμε το διάστημα (i, i+1) που περιέχει το m υπολογίζοντας το $i=rank_{B(S)}(m)$. Τέλος, βρίσκουμε το $rank_S(u)$ εκτελώντας ένα table look-up στις τιμές h,i και είναι ή $u<u_h$ ή $u>u_h$. Ο χώρος που καταλαμβάνει ο κόμβος καθορίζεται από το χώρο που χρειάζεται το table αυτό, που είναι $O(B^2)$. Αφού ο αριθμός ων κόμβων του B-tree είναι $O(N/B^4)$, ο συνολικός χρόνος που απαιτείται για την αναπαράσταση του fusion tree είναι O(N).

Ο υπολογισμός του $rank_{K(S)}(\hat{u}(S))$ σε constant time γίνεται ως εξής: Στον κόμβο βρίσκεται αποθηκευμένη η ποσότητα K_S, που είναι τα συμπιεσμένα κλειδιά του K(S) συγκεντρωμένα σε μία λέξη μνήμης. Ειδικότερα, τα bits του K_s, χωρίζονται σε $E=\lfloor b^{1/6} \rfloor$ κομμάτια ίσου μεγέθους (αν το b δε διαιρείται από το E, τότε τα πιο δεξιά $E\lfloor b/E \rfloor$ bits του K(S) χωρίζονται σε τμήματα ίσου μεγέθους). Κάθε συμπιεσμένο κλειδί καταλαμβάνει ένα ξεχωριστό κομμάτι. Άρα, ο μέγιστος επιτρεπόμενος βαθμός B ενός κόμβου θα πρέπει να ικανοποιεί τη σχέση $B \leq E$. Εφόσον $N \leq 2^b$, υπάρχουν αρκετά κομμάτια ώστε $\log B = \Omega(\log\log N)$ και έτσι επιτυγχάνεται η επιτάχυνση σε σχέση με τη συμβατική ταξινόμηση. Ο περιορισμός για το μέγεθος του E προέρχεται από το γεγονός ότι τα συμπιεσμένα κλειδιά χρειάζονται μέχρι k^4 bits το κάθε ένα. Τα συμπιεσμένα κλειδιά είναι στοιχισμένα στα δεξιά και τα αρχικά δύο bits κάθε κομματιού είναι μηδέν. Τα κομμάτια που δεν περιέχουν κλειδιά είναι γεμάτα από άσσους εκτός από το πρώτο bit που είναι μηδέν. Η δομή του K(S) φαίνεται παρακάτω.

K_S=01…1	…	00…$\hat{u}_k$	00…$\hat{u}_{k-1}$	…	00…$\hat{u}_1$
Κομμάτι E		Κομμάτι k	Κομμάτι k-1		κομμάτι 1

Τώρα, εκτελώντας έναν κατάλληλο πολλαπλασιασμό και ένα bitwise OR operation που περιλαμβάνει το $\hat{u}(S)$, παίρνουμε τον αριθμό Y που φαίνεται παρακάτω.

Y = 10…0$\hat{u}(S)$	…	10…0$\hat{u}(S)$
Κομμάτι E		κομμάτι 1

Στη συνέχεια, αφαιρούμε το K(S) από το Y και μηδενίζουμε τα πάντα εκτός από το πρώτο bit κάθε κομματιού (εκτελώντας ένα bitwise AND). Το αποτέλεσμα θα περιέχει άσσο στην αρχή κάθε πεδίου $\hat{u}_i$ για το οποίο ισχύει $\hat{u}_i \leq \hat{u}(S)$. Ένας κατάλληλος πολλαπλασιασμός θα αθροίσει αυτά τα bits, έτσι ώστε ένα τμήμα του αποτελέσματος να περιέχει τον επιθυμητό βαθμό, ο οποίος μπορεί στη συνέχεια να εξαχθεί με ένα AND operation.

Χρησιμοποιούμε την ίδια μέθοδο για να υπολογίσουμε το $rank_{B(S)}(m)$ σε constant time. Αντί να χρησιμοποιήσουμε την ποσότητα K(S), χρησιμοποιούμε την ποσότητα B_S στην οποία τα στοιχεία του B(S) είναι συγκεντρωμένα σε μία λέξη μνήμης.

4. Q-heap

4.1 Εισαγωγή

Το Q-heap [7] είναι μία δομή που υποστηρίζει εισαγωγή, διαγραφή και ψάξιμο σε constant worst case χρόνο (θεωρούμε ένα ανεπιτυχές ψάξιμο επιστρέφει τον successor του κλειδιού). Σημειώνουμε ότι η πράξη findmin είναι μία ειδική περίπτωση ψαξίματος και έτσι η δομή αυτή μπορεί να χρησιμοποιηθεί σα heap. Το Q-heap είναι βασισμένο στις τεχνικές του fusion tree.

4.2 Βασική ιδέα

Έστω L το πάνω όριο του επιτρεπόμενου μεγέθους του Q-heap. Το L μεγαλώνει σα συνάρτηση του n και η τιμή του θα υπολογιστεί αργότερα.

Δοσμένου ενός συνόλου $S=\{u_1, u_2, \ldots, u_k\}$ από b-bit ακεραίους, με $u_1<u_2<\ldots<u_k$, για $1\le i\le k-1$, έστω $c_i = msb(u_i, u_{i+1})$ όπου msb(x, y) είναι η πιο σημαντική θέση bit στην οποία διαφέρουν οι x και y. Έστω B(S) το σύνολο $\{c_1, c_2, \ldots, c_{k-1}\}$ και τ_s η ακολουθία $c_1, c_2, \ldots, c_{k-1}$. Στη συνέχεια ορίζουμε ένα δυαδικό δέντρο $Tree(\tau_s)$ ως εξής: Έστω c_j ο μέγιστος από τους όρους της ακολουθίας τ_s. Τότε η ρίζα του $Tree(\tau_s)$ είναι ένας κόμβος με τιμή c_j. Το αριστερό υποδέντρο της ρίζας ορίζεται αναδρομικά ως $Tree(c_1, c_2, \ldots, c_{j-1})$ ενώ το δεξί υποδέντρο είναι το $Tree(c_{j+1}, c_{j+2}, \ldots, c_{k-1})$. Το δέντρο $Tree(\tau_s)$ έχει k φύλλα αριθμημένα από αριστερά προς τα δεξιά, με το πιο αριστερό φύλλο να έχει την τιμή 1. Ένας b-bit αριθμός u, ορίζει ένα μονοπάτι στο δέντρο αυτό. Ορίζουμε $Leaf(\tau_s, u)$ το φύλλο που τερματίζει το μονοπάτι που ορίζει ο u. Δοσμένου ενός πεπερασμένου συνόλου Q από ακεραίους, ορίζουμε $rank_Q(x)=|\{t : t \in Q, t \le x\}|$. Ο υπολογισμός της τιμής $rank_S(x)$ είναι η κεντρική διαδικασία του ψαξίματος για ένα κλειδί x.

Λήμμα 4.1: Η ποσότητα $rank_S(x)$ ορίζεται επακριβώς από τα: $Tree(\tau_s)$, $i=Leaf(\tau_s, u)$ και $rank_{B(S)}(msb(u,u_i))$ μαζί με τη σχετική σειρά ανάμεσα στα u και u_i.

Απόδειξη: Η απόδειξη γίνεται με επαγωγή στο r-m, όπου $r=|B(S)|$ και $m=rank_{B(S)}(msb(u,u_i))$. Ας δούμε πρώτα την περίπτωση όπου r-m=0. Αν $u=u_i$, τότε δεν έχουμε τίποτε να αποδείξουμε. Αλλιώς, αφού $u\neq u_i$ και m=r, συμπεραίνουμε ότι η πιο σημαντική θέση bit στην οποία τα u και u_i διαφέρουν είναι πιο σημαντική από

όλες τις θέσεις που διαφοροποιούν οποιαδήποτε δύο από τα στοιχεία του S. Συνεπώς, το $rank_S(u)$ είναι 0, ή |S|, ανάλογα από το αν $u<u_i$ ή $u>u_i$.

Έστω τώρα ότι r-m>0. Όπως και πριν, μπορούμε να υποθέσουμε ότι $u \neq u_i$. Το μεγαλύτερο στοιχείο του B(S) είναι η πιο σημαντική θέση bit c_j στην οποία διαφέρουν οποιαδήποτε δύο στοιχεία του S. Έστω S_0 το σύνολο που περιέχει τα στοιχεία του S που έχουν μηδέν στη θέση c_j και S_1 το σύνολο των στοιχείων του S που έχουν άσσο στη θέση c_j. Τα στοιχεία του S_0 είναι μικρότερα από τα στοιχεία του S_1. Επιπλέον τα σύνολα S_0 και S_1 καθορίζουν το αριστερό και δεξί υποδέντρο του τ_s αντίστοιχα. Λέμε ότι το u ταιριάζει με το S_0 αν είναι μικρότερο από όλα τα στοιχεία του S_1, και ότι ταιριάζει με το S_1 αν είναι μεγαλύτερο από όλα τα στοιχεία του S_0. Επειδή m<r, τα u και u_i συμφωνούν στις πιο σημαντικές θέσεις, τουλάχιστον μέχρι τη θέση c_j. Προκύπτει ότι το u ταιριάζει με το ίδιο υποσύνολο με το οποίο ταιριάζει το u_i. Οι τιμές $Tree(\tau_s)$ και i, συνεπώς, καθορίζουν επακριβώς με ποιο από τα δύο σύνολα ταιριάζει το u. Συνεπώς, ο βαθμός του u στο S ($rank_S(u)$) μπορεί να βρεθεί αν βρεθεί ο βαθμός του u στο σύνολο με το οποίο ταιριάζει ο u (S_0 ή S_1).

Για να βρούμε το βαθμό του u μέσα στο υποσύνολο, (έστω το S_0), έστω r_1 και m_1, οι τιμές που αντιστοιχούν στα r, m, για το νέο σύνολο, δηλαδή το S_0 (οι τιμές r_1 και m_1 αν έχουμε τα m, i και το $Tree(\tau_s)$). Επιπλέον, αφού το c_j δεν ανήκει στο S_0, συμπεραίνουμε ότι $r_1-m_1<r-m$ (*παρατηρούμε ότι το r-m είναι ο αριθμός των τιμών του B(S) που είναι αυστηρά μεγαλύτερες από την τιμή $msb(u, u_i)$*). Εφαρμόζουμε τώρα την υπόθεση της επαγωγής στο σύνολο S_0 για να βρούμε τον επιθυμητό βαθμό. Ολοκληρώνεται έτσι η απόδειξη του λήμματος. □

Υπενθύμιση: Οι κόμβοι του fusion tree δε χρησιμοποιούν το $Tree(\tau_s)$ αλλά την ιδέα των συμπιεσμένων κλειδιών και έτσι μπορούν να αναπαρασταθούν μεγαλύτερα σύνολα S, με αποτέλεσμα όμως να μη μπορούν τα σύνολα αυτά να ενημερωθούν αποτελεσματικά.

Στόχος μας είναι να χρησιμοποιήσουμε το παραπάνω λήμμα σαν τη βάση για ένα table look-up σχήμα για τον υπολογισμό της ποσότητας $rank_S(u)$. Όπως αναφέρεται στην περιγραφή των fusion trees, η τιμή msb(x, y) υπολογίζεται σε constant time. Επιπλέον, αν θέσουμε d=msb(x, y), τότε ο υπολογισμός αυτός επιστρέφει δύο επιπλέον ποσότητες, τις 2^d και 2^{b-d}. Διατηρούμε μία b-bit ποσότητα B_S που περιέχει τις (μικρές) δυαδικές αναπαραστάσεις των κλειδιών του B(S), πακεταρισμένες σε ίσα

κομμάτια, καθένα από τα οποία έχει logb +4 bits. Χρησιμοποιώντας το B_S, ο υπολογισμός του $rank_{B(S)}(msb(\mu, u_i))$ μπορεί να γίνει σε constant time.

Υπενθύμιση: Ο υπολογισμός του $rank_S(u)$ δεν πρέπει να μπερδεύεται με τον υπολογισμό του $rank_{B(S)}(\alpha)$ όπου $0 \leq \alpha \leq b$. Ο δεύτερος υπολογισμός δεν περιλαμβάνει table look-up. Επειδή το μήκος ενός κομματιού του B_S είναι logb +4 το πολύ, προκύπτει ότι το μέγιστο μέγεθος του B(S) είναι b/(logb +4). Αν θεωρήσουμε ότι $b \geq \log n$, τότε το μέγεθος του B(S) είναι $\Theta(\log n/\log\log n)$. Άρα, $L=O(\log n/\log\log n)$.

Αυτά που απομένει να δούμε είναι πως διατηρούμε το $Tree(\tau_s)$ (καθώς κάνουμε εισαγωγές και διαγραφές) και πως υπολογίζουμε τη συνάρτηση $Leaf(\tau_s, u)$. Μία plausible προσέγγιση περιλαμβάνει την κατασκευή, κατά τη διάρκεια της φάσης προεπεξεργασίας, μιας μηχανής πεπερασμένων καταστάσεων, Οι καταστάσεις της οποίας αντιστοιχούν στο τ_s. Παρόλ' αυτά υπάρχει ένα εμπόδιο που αφορά το γεγονός ότι ο αριθμός των πιθανών αποτελεσμάτων για το τ_s εξαρτάται από το μέγεθος b της λέξης, αφού οι όροι του τ_s δηλώνουν θέσεις bits. Ο χώρος που απαιτείται για να αναπαρασταθούν όλες αυτές οι καταστάσεις δεν είναι διαθέσιμος αφού στόχος μας είναι να τον μειώσουμε όσο το δυνατό περισσότερο. Μια καλύτερη λύση είναι να ενοποιηθούν καταστάσεις σε ισοδύναμες καταστάσεις για να επιτευχθεί μείωση χώρου. Δοσμένου του συνόλου $B(S)=\{c_1, c_2, \ldots, c_{k-1}\}$ και της ακολουθίας $\tau_s=c_1, c_2, \ldots, c_{k-1}$, ορίζουμε κανονικό αντιπρόσωπο της ακολουθίας τ_s την ακολουθία $\sigma_s= d_1, d_2, .., d_{k-1}$ όπου $d_i=rank_{B(S)}(c_i)$. Ο αριθμός των κλάσεων ισοδυναμίας είναι αυστηρά συνάρτηση του |S|. Το λήμμα Α μπορεί να γίνει τώρα ως εξής:

Λήμμα 4.2: Η ποσότητα $rank_S(x)$ ορίζεται επακριβώς από τα: $Tree(\sigma_s)$, $i=Leaf(\tau_s, u)$ και $rank_{B(S)}(msb(u,u_i))$ μαζί με τη σχετική σειρά ανάμεσα στα u και u_i.

Απόδειξη: προφανής.

Τα αντικείμενα του λήμματος Λήμμα 6.2 δίνουν ένα αριθμό από $S^{O(|s|)}$ πιθανές τιμές. Το μέγεθος αυτού του χώρου είναι ανεξάρτητο από το μήκος λέξης b. Με κατάλληλους περιορισμούς στο μέγεθος του S, τα αντικείμενα αυτά, δημιουργούν ένα plausible index για το table lookup σχήμα για τον υπολογισμό του $rank_S(u)$. Υπάρχουν όμως κάποια τεχνικά προβλήματα που πρέπει να ξεπεραστούν. Το ένα πρόβλημα αφορά τον υπολογισμό της ποσότητας $Leaf(\tau_s, u)$ ο οποίος πρέπει να γίνει σε constant time.

Clarification: Όταν λέμε ότι ένα συγκεκριμένο αντικείμενο αποτελεί index για ένα table, αυτό που εννοούμε είναι ότι το αντικείμενο αυτό έχει το πολύ logn bits, ώστε

να περιορίζεται έτσι το εύρος των τιμών που μπορούμε να ψάξουμε. Έστω λοιπόν ότι έχουμε την τιμή i=Leaf(τ_s, u), το αποτέλεσμα α της σύγκρισης των u και u_i, την τιμή d= $rank_{B(S)}(msb(u,u_i))$ και το αντικείμενο σ_s. Αν θέσουμε (π.χ.) $L<\frac{1}{10}\log n/\log\log n$, τότε τα αντικείμενα αυτά μπορούν να κωδικοποιηθούν όλα μαζί σαν ένας index που έχει το πολύ logn bits, και να προσπελάσουμε μέσω του index αυτού ένα table για να βρούμε την τιμή του $rank_S(u)$ (απαιτούμε αυτά τα tables να μπορούν να χτίζονται σε γραμμικό χρόνο και χώρο). Θεωρώντας ότι το table έχει φτιαχτεί και ότι ο index που αναφέρθηκε παραπάνω υπάρχει, ο βαθμός μπορεί να βρεθεί σε constant time.
Ο υπολογισμός του Leaf(τ_s, u) γίνεται σε constant time αλλά η διαδικασία είναι πολύπλοκη δεν θα αναφερθούμε σε αυτή.

Τα insertions και deletions γίνονται επίσης σε constant time. Τελικά προκύπτει ότι:
Το Q-heap υποστηρίζει insertions, deletions και search operations σε constant time, μπορεί να «χωρέσει» $(\log n)^{1/4}$ στοιχεία με δεδομένο ότι έχουμε O(n) χρόνο και χώρο για προεπεξεργασία και ότι b≥logn όπου είναι το μήκος λέξης.

4.3 Σχόλια.

Τα Q-heaps είναι ένας άλλος τρόπος να αναπαρασταθούν οι εσωτερικοί κόμβοι του fusion tree. Αν χρησιμοποιηθούν τα q-heaps στους κόμβους του fusion tree, τότε οι χρόνοι του fusion tree από κατανεμημένοι γίνονται worst case.

5. Packed B-tree.

5.1 Περιγραφή της δομής

Λήμμα 4.1: Αν $(k+1)D \leq w$ και $k \geq \log n$ τότε ένα σύνολο που περιέχει το πολύ n k-bit ακεραίους μπορεί να διατηρηθεί σε μία unit cost RAM υποστηρίζοντας τις λειτουργίες insert x, delete x και find x (βρες το x ή τον πλησιέστερο γείτονά του) σε χρόνο $O(\log D+\log n/\log D)$ στη χειρότερη περίπτωση.

Απόδειξη: Χρησιμοποιούμε ένα packed B-tree [1].

Σε ένα κόμβο βαθμού D ενός packed B-tree υπάρχουν D-1 κλειδιά που οδηγούν το ψάξιμο και D δείκτες στα παιδιά του κόμβου. Όλα τα κλειδιά του κόμβου είναι συμπιεσμένα σε μία θέση μνήμης. Επίσης, όλοι οι δείκτες είναι συμπιεσμένοι σε μία θέση μνήμης.

Χρησιμοποιούμε ένα packed B-tree με μέγιστο branching factor D. Αυτό σημαίνει ότι ο βαθμός κάθε κόμβου είναι ανάμεσα σε D/2 και D. Κάθε κλειδί αναπαρίσταται από ένα (k+1)-bit πεδίο. Το πρώτο (πιο αριστερό bit), το test bit είναι 1, ενώ τα υπόλοιπα bits αποτελούν το κλειδί. Τα D-1 κλειδιά είναι αποθηκευμένα σε ταξινομημένη από αριστερά προς τα δεξιά σειρά, στις πιο δεξιές (d-1)(k+1) θέσεις της λέξης μνήμης. Για ένα κόμβο υ, τα συγκεντρωμένα κλειδιά συμβολίζονται K_υ. Ένα packed B-tree κρατάει τους κόμβους του σε ένα array μήκους n, έτσι ώστε ένας δείκτης να χωράει σε log n bits. Συνεπώς οι D δείκτες μπορούν να αποθηκευτούν στα πιο δεξιά $D\log n<kD<w$ bits μιας λέξης μηχανής.

Όταν ψάχνουμε για ένα k-bit κλειδί x στο packed B-tree, πρώτα κατασκευάζουμε έναν ακέραιο X που περιέχει D αντίγραφα του x. Πάλι, κάθε κλειδί αναπαρίσταται από ένα (k+1)-bit πεδίο. Αυτή τη φορά τα test bits είναι 0. Το X δημιουργείται από το x, με διαδοχικά shifts και bitwise OR operations. Το X δημιουργείται σε χρόνο O(logD). Για να καθορίσουμε το βαθμό του x ανάμεσα στα κλειδιά του κόμβου υ, κάνουμε μία πολλαπλή σύγκριση μέσω της αφαίρεσης $R=(K_\upsilon-X)$ AND M όπου M είναι μία έτοιμη μάσκα στην οποία όλα τα test bits είναι 1 ενώ όλα τα υπόλοιπα bits είναι 0. Στο R, κάθε test bit που αντιστοιχεί σε ένα κλειδί μεγαλύτερο από το x γίνεται 1, ενώ τα test bits που αντιστοιχούν σε κλειδιά μικρότερα του x γίνονται 0. Αφού τα κλειδιά στο K_υ είναι ταξινομημένα, υπάρχουν το πολύ D+1 διαφορετικές τιμές του R. Κάθε τιμή αντιστοιχεί σε ένα βαθμό του x ανάμεσα στα κλειδιά του K_υ.

Αυτό σημαίνει ότι μπορούμε να κατασκευάσουμε ένα lookup table που να έχει όλες τις πιθανές τιμές του R σαν indices (το ίδιο table μπορεί να χρησιμοποιηθεί για όλους τους κόμβους του δέντρου και φυσικά είτε ακόμη και για πολλά δέντρα). Ψάχνοντας στο table μπορούμε να βρούμε το βαθμό του x στο K_v. (Ο χώρος που απαιτείται για το table είναι $O(2^{D(k+1)})$). Το table μπορεί να κατασκευαστεί σε χρόνο O(D). Άρα για να χρησιμοποιήσουμε packed B-trees, χρειαζόμαστε O(D) χρόνο προεπεξεργασίας. Αυτό μπορεί να γίνει κατά τη διάρκεια των global rebuildings όπως θα δούμε παρακάτω.

Έχοντας βρει το βαθμό του x όπως περιγράφηκε παραπάνω, χρησιμοποιούμε την πληροφορία αυτή για να βρούμε το σωστό υποδέντρο, με shift και bitwise λογικά operations. Το δέντρο διατηρείται με τις γνωστές λειτουργίες των B-trees. Λειτουργίες όπως η πρόσθεση ενός κλειδιού σε ένα κόμβο, η αφαίρεση ενός κλειδιού από ένα κόμβο, η ένωση δύο κόμβων ή η διάσπαση ενός κόμβου γίνονται εύκολα σε constant time με shifts και bitwise λογικά operations.

Θεώρημα: Δοσμένου του w σαν παράμετρο, υπάρχει μία δομή δεδομένων για την αποθήκευση ενός συνόλου από n w-bit ακεραίους σε ένα unit-cost RAM υποστηρίζει:

Insert x: $O(\sqrt{\log n})$

Delete x: $O(\sqrt{\log n})$

Βρες το x ή ανέφερε ότι το x δεν υπάρχει: O(1)

Βρες τα s μεγαλύτερα ή μικρότερα στοιχεία: O(s)

Βρες τα s μεγαλύτερα στοιχεία < x ή τα s μικρότερα στοιχεία >x: $O(\sqrt{\log n}+s)$

Βρες όλα τα στοιχεία ανάμεσα στο x και το y: $O(\sqrt{\log n}$ +μέγεθος της εξόδου)

Η αναπαράσταση μπορεί να είναι ντετερμινιστική ή randomized. Στην ντετερμινιστική αναπαράσταση χρησιμοποιούνται μόνο AC^0 εντολές, τα παραπάνω κόστη είναι worst case και ο απαιτούμενος χώρος είναι $O(n2^{\varepsilon w})$ για κάποια σταθερά $\varepsilon>0$. Στη randomized αναπαράσταση, οι παραπάνω χρόνοι είναι expected και ο χώρος που απαιτείται είναι O(n).

Αν ο χρόνος για να βρεθεί ο x ή για να αναφερθεί ότι ο x δεν υπάρχει, επιτραπεί να είναι $O(\sqrt{\log n})$, τότε ο χώρος που απαιτεί η ντετερμινιστική εκδοχή είναι $O(n+2^{\varepsilon w})$.

Απόδειξη: Στη συνέχεια θα θεωρούμε ότι το n παραμένει περίπου το ίδιο κατά τη διάρκεια των updates. Όταν η τιμή του n αλλάξει σημαντικά, πρέπει η δομή να χτιστεί από την αρχή.

Αφού ένα packed B-tree μπορεί να χειριστεί αποτελεσματικά k-bit κλειδιά, είναι φυσικό επακόλουθο να θεωρήσουμε ότι τα w-bit κλειδιά αποτελούνται από k-bit κομμάτια και να τα αποθηκεύσουμε σε ένα trie ύψους w/k. Σε κάθε εσωτερικό κόμο τα παιδιά είναι ένα σύνολο από k-bit κλειδιά, τα οποία μπορούν να αποθηκευτούν σε ένα packed B-tree. Όταν ψάχνουμε για ένα κλειδί x, ή τον κοντινότερο γείτονα του x, πρώτα ψάχνουμε με τον κλασικό τρόπο στο trie. Αν το x είναι παρόν, το ψάξιμο τελειώνει σε ένα κόμβο του trie. Αλλιώς, η διαπέραση του trie τελειώνει όταν προσπαθήσουμε να βρούμε ένα παιδί που δεν υπάρχει. Τότε, για να βρούμε τον κοντινότερο γείτονα του x, ψάχνουμε ανάμεσα στα παιδιά του κόμβου, χρησιμοποιώντας το packed B-tree. Όταν εισάγουμε ένα καινούριο στοιχείο, απλά προσθέτουμε ένα καινούριο παιδί σε κάποιο κόμβο του trie. Το αντίστοιχο k-bi κλειδί εισάγεται στο packed B-tree. Το σβήσιμο γίνεται με τον ίδιο τρόπο.

Για παράδειγμα, αν θέσουμε $k=\Theta(\frac{w}{\sqrt{\log n}})$ το ύψος του trie καθώς και ο βαθμός του packed B-tree γίνονται $\Theta(\sqrt{\log n})$ και το κόστος για ένα update ή neighbor search γίνεται Θ(logn/loglogn).

Η παραπάνω ιδέα μπορεί να εφαρμοστεί σε ένα Van Emde Boas δέντρο. Αρχικά θεωρούμε ότι ένας w-bit ακέραιος αποτελείται από εw-bit μικρούς ακεραίους και αναπαριστούμε το σύνολό μας σαν ένα path compressed trie με ύψος 1/ε. Κάθε κόμβος έχει ένα array μεγέθους $2^{\varepsilon w}$ που περιέχει όλες τις πιθανές τιμές εξόδου. Τα παιδιά αποτελούν ένα σύνολο από εw-bit κλειδιά τα οποία είναι αποθηκευμένα σε ένα partial VEB tree όπως περιγράφεται παρακάτω.

Ένα VEB tree, είναι μία αναδρομική δομή trie όπου το μήκος των αναπαριστούμενων κλειδιών υποδιπλασιάζεται σε κάθε αναδρομική κλήση. Η δομή αυτή περιλαμβάνει O(n) κόμβους και (n) ακμές και υποστηρίζει αποτελεσματικά neighbor searches. Κανονικά, ένα VEB που αναπαριστά εw-bit κλειδιά, θα είχε ύψος O(log(εw)). Ωστόσο, αντί να αφήσουμε το VEB να κάνει όλη τη δουλειά, ίσως είναι καλύτερα να σταματήσουμε τις αναδρομές σε κάποιο σημείο, και να γυρίσουμε σε ένα packed B-tree. Σταματώντας μετά από $O(\sqrt{\log n})$ αναδρομικές κλήσεις, το μήκος των ακεραίων έχει μειωθεί σε $\min(\log n, \frac{\varepsilon w}{2^{\sqrt{\log n}}})$. Αν το μήκος των ακεραίων είναι το πολύ logn, απλά συνεχίζουμε στο VEB, εκτελώντας άλλες loglogn αναδρομές. Αλλιώς, στο

κατώτατο επίπεδο του VEB χρησιμοποιούμε packed B-trees με $k=\frac{\varepsilon w}{2^{\sqrt{\log n}}}$ και $D=2^{\sqrt{\log n}}$. Η ζητούμενη πολυπλοκότητα προκύπτει από το προηγούμενο λήμμα. Τα στοιχεία του συνόλου σχηματίζουν μία διπλή διασυνδεδεμένη λίστα. Με τη λίστα, μπορούμε με αποτελεσματικό τρόπο να απαντήσουμε range queries. Ο συνολικός αριθμός των εσωτερικών κόμβων στο path-compressed trie είναι μικρότερος από n, και κάθε κόμβος χρειάζεται $O(2^{\varepsilon w})$ χώρο. Υπάρχουν λιγότερα από 2n VEB's που περιέχουν ένα σύνολο από 2n το πολύ μικρούς (short) ακεραίους. Κάθε κόμβος σε ένα VEB χρησιμοποιεί $O(2^{\varepsilon w})$ χώρο και ο συνολικός αριθμός τέτοιων κόμβων είναι O(n). Τα packed B-trees χρησιμοποιούν συνολικά O(n) χώρο ενώ το global lookup table χρησιμοποιεί $O(n2^{\varepsilon w})$ χώρο.

5.2 Μείωση του χώρου.

Αν επιτρέψουμε να είναι το κόστος για τα member queries $O(\sqrt{\log n})$, τότε ο χώρος μπορεί να μειωθεί σε $O(n+2^{\varepsilon w})$ ή αν U είναι το μέγεθος του σύμπαντος, σε $O(n+U^{\varepsilon})$. Χρησιμοποιούμε την παραπάνω δομή δεδομένων με δύο τροποποιήσεις.

Πρώτα χρησιμοποιούμε μία ιδέα παρόμοια με αυτή των Willard [4] και Carlsson [14] για να μειώσουμε τον αριθμό των κόμβων του trie διατηρώντας 2-3 δέντρα ύψους $\Theta(\sqrt{\log n})$ στον πάτο του trie. Όλα τα 2-3 δέντρα έχουν το ίδιο ύψος και τα updates στο trie αντιστοιχούν σε ενώσεις και σπασίματα 2-3 δέντρων. Διαλέγοντας κατάλληλες σταθερές, σιγουρεύουμε ότι ο συνολικός αριθμός των κλειδιών που αποθηκεύονται στο trie είναι μικρότερος από $\frac{n}{2^{2\sqrt{\log n}}}$.

Η δεύτερη αλλαγή που κάνουμε είναι να αντικαταστήσουμε το ε στην απόδειξη με το $\frac{\varepsilon}{2\sqrt{\log n}}$.

Οι αλλαγές ατές αυξάνουν το κόστος του ψαξίματος και των updates κατά $O(\sqrt{\log n})$. Τα B-trees χρησιμοποιούν γραμμικό χώρο και το trie χρησιμοποιεί συνολικό χώρο $O(\frac{n}{2^{2\sqrt{\log n}}} 2^{(\frac{\varepsilon w}{2\sqrt{\log n}})})$. Για να δούμε ότι ο χώρος είναι $O(n+2^{\varepsilon w})$ ξεχωρίζουμε 2 περιπτώσεις.

1. $n \le 2^{\varepsilon w/2}$. **Και οι δύο όροι είναι $O(2^{\varepsilon w/2})$.**

2. $n>2^{\varepsilon w/2}$. Ο δεύτερος όρος είναι $O(2^{2\sqrt{\log n}})$.

5.3 Σχόλια

Μετά την παρουσίαση του packed B-tree υπάρχουν δύο λύσεις για τη διατήρηση ενός συνόλου n αριθμών, με worst case χρόνο $O(\sqrt{\log n})$ για predecessor (search) queries και updates. Η μία λύση είναι το fusion tree αν στους εσωτερικούς κόμβους του χρησιμοποιηθούν Q-heaps. Η άλλη λύση είναι το packed B-tree, που σε σύγκριση με το fusion tree καταλαμβάνει μεγαλύτερο χώρο, αλλά δε χρησιμοποιεί πολλαπλασιασμό.

6 Exponential search tree

6.1 Εισαγωγή

Το exponential search tree είναι ένα δέντρο στο οποίο ο βαθμός των κόμβων αυξάνει εκθετικά όσο προχωράμε προς τα κάτω. Για να υποστηρίζεται γρήγορο ψάξιμο, σε κάθε κόμβο αποθηκεύεται επιπλέον πληροφορία

6.2 Περιγραφή του δέντρου.

Λήμμα 6.1: Ας θεωρήσουμε μία στατική δομή δεδομένων που περιέχει d κλειδιά και μπορεί να κατασκευαστεί σε χρόνο $O(d^4)$, ενώ $O(d^4)$ είναι και ο χώρος που χρειάζεται, και μπορεί να υποστηρίζει neighbor queries σε O(S(d)) worst case χρόνο. Τότε υπάρχει μία δυναμική δομή δεδομένων με τα ακόλουθα χαρακτηριστικά:

- Χρησιμοποιεί O(n) χώρο.
- Μπορεί να κατασκευαστεί σε O(n) worst case χρόνο και χώρο.
- Το κόστος στη χειρότερη περίπτωση για το ψάξιμο (συμπεριλαμβανομένου και του neighbor search) ικανοποιεί τη σχέση $T(n)=O(S(n^{1/5}))+T(n^{4/5})$.
- Το κατανεμημένο κόστος για την επανακατασκευή κατά τη διάρκεια των updates είναι O(loglogn).

Απόδειξη: Θα χρησιμοποιήσουμε ένα exponential search tree το οποίο έχει τις ακόλουθες ιδιότητες:

- Η ρίζα του έχει βαθμό $\Theta(n^{1/5})$
- Τα κλειδιά της ρίζας είναι αποθηκευμένα σε μία τοπική δομή δεδομένων. Κατά τη διάρκεια ενός ψαξίματος, η τοπική δομή χρησιμοποιείται για να βρεθεί το σωστό υποδέντρο στο οποίο πρέπει να συνεχιστεί το ψάξιμο.
- Τα υποδέντρα είναι exponential search trees με μέγεθος $\Theta(n^{4/5})$.

Πρώτα θα δείξουμε ότι, δοσμένων n ταξινομημένων κλειδιών, ένα exponential search tree μπορεί να κατασκευαστεί σε γραμμικό χρόνο και χώρο. Αφού το κόστος για την κατασκευή ενός κόμβου βαθμού d είναι $O(d^4)$ το συνολικό κόστος κατασκευής C(n) δίνεται από τη σχέση:

$$C(n) = O((n^{1/5})^4 + n^{1/5}C(n^{4/5}) \Rightarrow C(n) = O(n)$$

Επιπλέον, με μία παρόμοια εξίσωση μπορούμε να δείξουμε ότι ο χώρος που απαιτείται είναι O(n).

Στη συνέχεια θα βρούμε το κόστος ψαξίματος T(n). Από την περιγραφή των exponential search trees προκύπτει ότι $T(n)=O(S(n))+T(n^{1/4})$.

Τέλος θα αναλύσουμε το κόστος των updates. Θα εξετάσουμε μόνο το κόστος της επανακατασκευής της δομής και θα αγνοήσουμε το κόστος για την εύρεση της θέσης που πρέπει να ενημερωθεί. Το τελευταίο κόστος είναι ίσο με το κόστος για ψάξιμο.

Η ισορροπία (balance) διατηρείται με global και partial rebuilding. Έστω n_0 ο αριθμός των στοιχείων κατά τη διάρκεια του τελευταίου global rebuilding. Το επόμενο global rebuilding θα συμβεί όταν $|n-n_0|\geq n_0/2$. Συνεπώς το γραμμικό κόστος για το global rebuilding είναι κατανεμημένο πάνω σε ένα γραμμικό αριθμό από updates και το κατανεμημένο κόστος είναι O(1).

Στο global rebuilding, θέτουμε $n_0=n$ και ο βαθμός της ρίζας επιλέγεται να είναι $\lceil n_0^{1/5} \rceil$, ενώ το μέγεθος κάθε υποδέντρου είναι $(n_0 / \lceil n^{4/5} \rceil) \pm 1$. Ανάμεσα στα global rebuildings κάνουμε σίγουρο ότι τα υποδέντρα έχουν μέγεθος τουλάχιστον $(n_0 / 2\lceil n^{4/5} \rceil) \pm 1$ και το πολύ $(2n_0 / \lceil n^{4/5} \rceil) \pm 1$. Όταν ένα update προκαλεί παραβίαση της συνθήκης αυτής, εξετάζουμε το άθροισμα των μεγεθών του υποδέντρου αυτού και ενός από τους γείτονές του. Το άθροισμα αυτό είναι ανάμεσα στις τιμές $(n_0 / \lceil n^{4/5} \rceil) \pm 1$ και $(4n_0 / \lceil n^{4/5} \rceil) \pm 1$. Επανακατασκευάζοντας τα δύο αυτά υποδέντρα σε 1, 2, ,3 ή 4 νέα υποδέντρα, κάνουμε σίγουρο ότι τα μεγέθη των νέων δέντρων θα απέχουν πολύ από τα όρια. Με τον τρόπο αυτό κάνουμε σίγουρο ότι απαιτείται ένας γραμμικός – ως προς το μέγεθος του υποδέντρου – αριθμός από updates μέχρι να χρειαστεί να επανακατασκευαστεί το υποδέντρο. Αφού το κόστος της κατασκευής ενός υποδέντρου είναι γραμμικό ως προς το μέγεθος του υποδέντρου, το κατανεμημένο κόστος για την επανακατασκευή ενός υποδέντρου είναι O(1).

Κάθε φορά που κάποια δέντρα επανακατασκευάζονται τα περιεχόμενα της ρίζας θα αλλάξουν και έτσι πρέπει να επανακατασκευαστεί και η ρίζα. Το κόστος γι' αυτό είναι $O((n^{1/5})^4)$. Πάλι, το κόστος αυτό είναι γραμμικό ως προς το μέγεθος του υποδέντρου. Συνεπώς το κατανεμημένο κόστος για την επανακατασκευή της ρίζας είναι O(1). Αυτό δίνει την ακόλουθη εξίσωση για το κατανεμημένο κόστος επανακατασκευής R(n):

$$R(n)=O(1)+R(n^{4/5}) \quad \Longrightarrow \quad R(n)=O(\log\log n) \qquad \square$$

6.3 Μια βελτίωση των fusion trees

Το κεντρικό κομμάτι του fusion tree είναι μία στατική δομή δεδομένων με τις ακόλουθες ιδιότητες.

Λήμμα 6.2: (Fredman και Willard) για κάθε d με $d=O(w^{1/6})$, μία στατική δομή δεδομένων που περιέχει d κλειδιά μπορεί να κατασκευαστεί σε χρόνο και χώρο $O(d^4)$, έτσι ώστε η δομή να υποστηρίζει neighbor queries σε O(1) worst case time.

Οι Fredman και Willard χρησιμοποίησαν αυτή τη στατική δομή για να υλοποιήσουν ένα B-tree. Τα φύλλα του B-tree είναι ρίζες ισοζυγισμένων δέντρων. Το κατανεμημένο κόστος για search και update είναι O(logn/logd +logd) για κάθε $d=O(w^{1/6})$.

Χρησιμοποιώντας ένα exponential search tree αντί για τη δομή των Fredman και Willard αποφεύγουμε την ανάγκη για ισοζυγισμένα δέντρα στο τελευταίο επίπεδο και την ίδια στιγμή βελτιώνουμε την πολυπλοκότητα όταν το μέγεθος της λέξης είναι μεγάλο.

Λήμμα 6.3: μία στατική δομή δεδομένων που περιέχει d κλειδιά μπορεί να κατασκευαστεί σε χρόνο και χώρο $O(d^4)$, έτσι ώστε η δομή να υποστηρίζει neighbor queries σε O((logd/logw)+1) worst case time.

Απόδειξη: Απλά κατασκευάζουμε ένα στατικό B-tree στο οποίο κάθε κόμβος έχει το μεγαλύτερο δυνατό βαθμό σύμφωνα με το λήμμα 6.2. Δηλαδή έχει βαθμό min(d, $w^{1/6}$). Το δέντρο ικανοποιεί τις συνθήκες του λήμματος. □

Λήμμα 6.4: Υπάρχει μία στατική δομή δεδομένων που καταλαμβάνει γραμμικό χώρο και για την οποία το κόστος στη χειρότερη περίπτωση για ένα ψάξιμο και το κατανεμημένο κόστος για ένα update είναι O((logn/logw)+loglogn).

Απόδειξη: Έστω T(n) το κόστος στη χειρότερη πρίπτωση. Συνδυάζοντας τα λήμματα 6.1 και 6.3 έχουμε:

$$T(n) = O(\frac{\log n}{\log w} + 1 + T(n^{5/4})) \Rightarrow T(n) = O(\frac{\log n}{\log w} + \log\log n)$$ □

6.4 Van Emde Boas δέντρα και perfect hashing.

Λήμμα 6.5: Μία στατική δομή δεδομένων που περιέχει d κλειδιά μπορεί να κατασκευαστεί σε χρόνο και χώρο $O(d^4)$, έτσι ώστε η δομή να υποστηρίζει neighbor queries σε O(logw) worst case time.

Απόδειξη: Υπάρχουν 2 περιπτώσεις:

Περίπτωση 1: logw > $\sqrt{\log d}$. Το λήμμα προκύπτει απευθείας από το λήμμα 6.3.

Περίπτωση 2: logw ≤ $\sqrt{\log d}$. Στην περίπτωση αυτή χρησιμοποιούμε μία παραλαγή του van Emde Boas δέντρου (VEB). Κανονικά το VEB δέντρο απαιτεί μεγάλο χώρο εξαιτίας των πολύ μεγάλων arrays που χρησιμοποιούνται. Παρόλ' αυτά, αν τα arrays αναπαρασταθούν σαν ένα perfect hash table, ο συνολικός χώρος που καταλαμβάνεται από το VEB είναι ανάλογος προς τον συνολικό αριθμό των κόμβων, ο οποίος είναι O(d) [3,12,13].

Όταν το VEB είναι υπό κατασκευή, δε μπορούμε να το αποθηκεύσουμε χρησιμοποιώντας perfect hashing. Αντί γι' αυτό, κατασκευάζουμε μία προσωρινή pointer based έκδοση του VEB στην οποία κάθε array από ακμές αντικαθίσταται από ένα binary search tree. Το κόστος για να ακολουθήσουμε μία ακμή του trie είναι O(logd) και το κόστος για να εισάγουμε ένα κλειδί είναι O(logd logw). Συνεπώς, το συνολικό κόστος για την κατασκευή του pointer based VEB είναι O(d logd logw).

Αφού κατασκευαστεί το pointer-based VEB κατασκευάζουμε ένα perfect hash table για όλες τις Θ(d) ακμές του trie. Χρησιμοποιείται η μέθοδος των Fredman Komolos και Szemeredi [8]. Στην αρχική του μορφή, ο αλγόριθμος αυτός κοστίζει χρόνο $O(d^3w)$ και χρησιμοποιεί διαίρεση. Αφού θέλουμε να αποφύγουμε τη διαίρεση, εξομοιώνουμε κάθε διαίρεση σε O(w) χρόνο. Με αυτό το επιπλέον κόστος το hash table μπορεί να κατασκευαστεί σε χρόνο $O(d^3w^2)$ χρόνο. Επίσης προϋπολογίζουμε κάποιες σταθερές που μας βοηθούν να υπολογίζουμε τις hash συναρτήσεις χωρίς διαίρεση. Για να υπολογίσουμε την ποσότητα x MOD p για κάποια γνωστή τιμή p, κάνουμε την ακόλουθη προεπεξεργασία. Σε O(w) χρόνο υπολογίζουμε την τιμή $r=2^w$ DIV p. Μπορούμε τώρα να υπολογίσουμε την τιμή x DIV p σαν rx DIV 2^w όπου η τελευταία διαίρεση είναι απλά ένα δεξί shift κατά w θέσεις.

Μια εναλλακτική μέθοδος για perfect hashing χωρίς διαίρεση, αναπτύχθηκε πρόσφατα από τον Raman [17]. Η μέθοδός του όχι μόνο αποφεύγει τη διαίρεση αλλά είναι και ασυμπτωτικά πιο γρήγορη ($O(d^2w)$). Αφού $w \leq 2^{\sqrt{\log d}}$ αυτό το κόστος είναι $O(d^4)$.

Λήμμα 6.6: Υπάρχει μία δομή δεδομένων που χρειάζεται γραμμικό χώρο και το κόστος στη χειρότερη περίπτωση για ένα ψάξιμο, καθώς και το κατανεμημένο κόστος για ένα update είναι O(logw loglogn)

Απόδειξη: Από τα λήμματα 6.1 και 6.5 προκύπτει:

$T(n)=O(\log w)+T(n^{4/5}) \rightarrow T(n)=O(\log w \log\log n)$. □

Θεώρημα 6.1: Σε ένα unit cost RAM με μήκος λέξης w, ένα διατεταγμένο σύνολο από n w-bit κλειδιά μπορεί να διατηρηθεί σε

$$O(\min(\frac{\log n}{\log w}+\log\log n,\ \log\lceil\frac{\log n}{2(\log w)^2}\rceil \log w)) =$$

$$[=O(\min(\sqrt{\log n},\frac{\log n}{\log w}+\log\log n,\log w\log\log n))]$$

χρόνο στη χειρότερη περίπτωση συμπεριλαμβανομένων των πράξεων insert, delete, member, search και neighbor search. Το κόστος για ψάξιμο είναι worst case ενώ το για τα updates είναι amortized. Για range queries υπάρχει και ένα επιπλέον κόστος για την αναφορά (reporting) των κλειδιών που βρέθηκαν. Η δομή χρησιμοποιεί γραμμικό χώρο.

Απόδειξη: Αν συνδυάσουμε τα λήμματα 6.1, 6.3 και 6.5 παίρνουμε την ακόλουθη εξίσωση για το κόστος T(n) ενός search ή update σε ένα exponential search tree.

$$T(n)=O(\min(1+\frac{\log n}{\log w},\log w))+T(n^{4/5}) \qquad (1)$$

Παρατηρώντας ότι η έκφραση min είναι $O(\sqrt{\log n})$ έχουμε ότι:

$$T(n)=O(\sqrt{\log n}+\sqrt{\log(n^{4/5})}+\sqrt{\log(n^{16/25})}+\ldots)=O(\sqrt{\log n})$$

και έτσι προκύπτει ότι

$$O(\min(\sqrt{\log n},\frac{\log n}{\log w}+\log\log n,\log w\log\log n))$$

Παρόλ' αυτά η έκφραση αυτή δεν είναι και τόσο κατατοπιστική (στενή ασυμπτωτικά) για όλους τους δυνατούς συνδυασμούς των n και w. Για να βρούμε μία ασυμπτωτικά στενή έκφραση σα λύση της εξίσωσης (1), παρατηρούμε ότι η παράμετρος n μειώνεται όσο προχωρούν οι αναδρομές ενώ το μήκος λέξης w παραμένει το ίδιο. Συνεπώς το δεξί κομμάτι της min έκφρασης θα χρησιμοποιείται σε κάποια από α υψηλά επίπεδα, ενώ τα χαμηλά επίπεδα θα χρησιμοποιούν το αριστερό κομμάτι. Συνεπώς ξεχωρίζουμε δύο περιπτώσεις.

1. $\log n \le 2(\log w)^2$. Το αριστερό κομμάτι θα χρησιμοποιηθεί από όλα τα επίπεδα και έτσι το συνολικό κόστος είναι $O(\frac{\log n}{\log w}+\log\log n)$

2. $\log n > 2(\log w)^2$. Το δεξί κομμάτι θα χρησιμοποιηθεί $O(\log\lceil \frac{\log n}{2(\log w)^2} \rceil)$ φορές δίνοντας ένα κόστος $O(\log\lceil \frac{\log n}{2(\log w)^2} \rceil \log w)$. Το αριστερό κομμάτι θα δώσει κόστος $O(\sqrt{\log w} + \log\log n)$. Έτσι, το συνολικό κόστος θα είναι

$$O(\log\lceil \frac{\log n}{2(\log w)^2} \rceil \log w + \sqrt{\log w} + \log\log n) = O(\log\lceil \frac{\log n}{2(\log w)^2} + 1 \rceil \log w)$$

Οι εκφράσεις για τις δύο περιπτώσεις γίνονται ίσες όταν $\log n = \Theta(2(\log w)^2)$. Αν τις συνδυάσουμε έχουμε την έκφραση

$$O(\min(\frac{\log n}{\log w} + \log\log n,\ \log\lceil \frac{\log n}{2(\log w)^2} + 1 \rceil \log w))$$

και η απόδειξη είναι τώρα πλήρης. □

6.5 Σχόλια.

Ενώ το fusion δέντρο επιτυγχάνει κατανεμημένο χρόνο O(logn/oglogn), το exponential search tree επιτυγχάνει κατανεμημένο χρόνο $O(\sqrt{\log n})$ ενώ ο χώρος παραμένει γραμμικός. Έχουμε συνεπώς μία βελτίωση σε σχέση με το fusion tree.

7. Ένας νέος αλγόριθμος για rectangle enclosure reporting

7.1 Γενικά

Παρουσιάζουμε ένα νέο αλγόριθμο για την αναφορά όλων των enclosures σε ένα σύνολο ορθογωνίων στο επίπεδο σε χρόνο O(*n logn loglogn+kloglogn*) και χώρο O(n) (το k είναι το μέγεθος της εξόδου). Το αποτέλεσμα αυτό είναι ήδη γνωστό αλλά ο αλγόριθμος που προτείνουμε χρησιμοποιεί απλές δομές δεδομένων. Για να επιτευχθεί το αποτέλεσμα χρησιμοποιείται το vEB δέντρο που παρουσιάζεται στο κεφάλαιο ένα, και είναι ένα καλό παράδειγμα για το πώς μπορούν οι δομές αυτές να χρησιμοποιηθούν στην υπολογιστική γεωμετρία.

7.2 Εισαγωγή.

Μελετάμε το rectangle enclosure reporting πρόβλημα, το οποίο ορίζεται ως εξής: δοσμένου ενός συνόλου S από n iso-oriented ορθογώνια στο επίπεδο, ανέφερε αποτελεσματικά όλα τα ζευγάρια ορθογωνίων (R, R′) όπου R, R′ ∈ S και το R′ περικλείει το R. Το 1982 οι Lee και Preparata [2] έδωσαν μία λύση στο πρόβλημα αυτό που χρειαζόταν O(n) χώρο και έτρεχε σε χρόνο $O(\log^2 n + k)$ (k είναι ο αριθμός των ζευγαριών). Ήταν ανοιχτό πρόβλημα για παραπάνω από 10 χρόνια το πώς θα μπορούσε να μειωθεί ο όρος $O(\log^2 n)$. Το 1995 οι Gupta et al [10] έδωσαν ένα αλγόριθμο με O(n+k) χώρο και O(n logn loglogn +k loglogn) χρόνο. Ο αλγόριθμος χρησιμοποιούσε μία γνωστή ισοδυναμία [5] ανάμεσα στο rectangle enclosure reporting πρόβλημα και στο 4-dimensional dominance reporting πρόβλημα, το οποίο είναι το εξής: Δοσμένου ενός συνόλου P από σημεία στον τετραδιάστατο χώρο, ανέφερε αποτελεσματικά όλα τα ζευγάρια (p, p′) όπου p, p′ ∈ S και το p «κυριαρχείται» από το p′. Ένα σημείο $p(p_1, p_2, p_3, p_4)$ κυριαρχείται από ένα άλλο σημείο p′ (p_1', p_2', p_3', p_4') αν και μόνο αν $p_i \le p_i'$ για κάθε i. Ο μετασχηματισμός από το ένα πρόβλημα στο άλλο μπορεί να γίνει σε γραμμικό χρόνο, αν αντικαταστήσουμε κάθε παραλληλόγραμμο R=[left, right]x[bottom, top] με το σημείο p(-left, -bottom, right, top). Η conference version του paper έδινε χώρο O(n) για τον αλγόριθμο. Όμως απεδείχθη [9] ότι ο χώρος που στην πραγματικότητα απαιτείται είναι O(n+k). Το πρόβλημα αυτό προέρχεται από μία υπορουτίνα του αρχικού αλγορίθμου. Στην ίδια

εργασία παρουσιάστηκε μία λύση για το πρόβλημα αυτό η οποία βασιζόταν στην full persistence και σε περιοδικό επαναχτίσιμο κάποιων δομών [9]. Με τον τρόπο αυτό ο χώρος έγινε γραμμικός ενώ η πολυπλοκότητα χρόνου δεν άλλαξε.

Στην εργασία αυτή θα δούμε ένα νέο αλγόριθμο που επιτυγχάνει γραμμικό χώρο τροποποιώντας με διαφορετικό τρόπο την «ένοχη» υπορουτίνα. Ο αλγόριθμος αυτός έχει ενδιαφέρον γιατί είναι απλός και κάνει έτσι το πρόβλημα πιο κατανοητό.

7.3 Περιγραφή του προβλήματος

Έστω P ένα σύνολο από n σημεία στον R^4. Θέλουμε να αναφέρουμε όλα τα ζευγάρια (p, p') όπου p, p' ∈ P και το p' κυριαρχεί πάνω στο p. Ο αλγόριθμος στην εργασία [11] βασίζεται στην τακτική διαίρει και βασίλευε. Μετά το αρχικό βήμα κανονικοποίησης, που μετασχηματίζει το χώρο από R^4 σε U^4 (U={1,2,..,n}), το σύνολο p διαιρείται ως προς την τέταρτη διάσταση σε δύο ίσα σύνολα P_1 και P_2, και ο αλγόριθμος συνεχίζει αναδρομικά στα σύνολα αυτά. Έτω R, B το σύνολο των σημείων που παίρνουμε από τα P_1, και P_2, αν αφαιρέσουμε από κάθε σημείο την τέταρτη διάσταση. Στο merge step του αλγορίθμου, όλα τα ζευγάρια κυριαρχίας (r, b) πρέπει να αναφερθούν.

Το merge step βασίζεται στο iterative application των δύο sweep steps, του clean step και του reporting step. Στο cleaning step, όλα τα σημεία του R που δεν κυριαρχούνται από κανένα σημείο του B και όλα τα σημεία του B που δεν κυριαρχούν σε κανένα σημείο του R, απομακρύνονται. Υποθέτουμε χωρίς βλάβη της γενικότητας ότι μετά το cleaning step, |R|>|B| και έστω B' το σύνολο των maximal σημείων του B (ένα σημείο του B είναι maximal αν δεν κυριαρχείται από κανένα άλλο σημείο του B. Στο reporting step αναφέρονται όλα τα ζευγάρια (r, b) όπου r ∈ R και b ∈ B. τότε τα σημεία του B' αφαιρούνται από το B και τα cleaning και reporting βήματα γίνονται ξανά μέχρι τα σύνολα R, B να αδειάσουν. Το cleaning step είναι μία απλή sweep ρουτίνα που μπορεί εύκολα να υλοποιηθεί έτσι ώστε να τρέχει σε γραμμικό χώρο. Το πρόβλημα υπάρχει στο reporting step και σε αυτό θα επικεντρωθούμε στη συνέχεια.

Έστω R, B δύο σύνολα σημείων του U^3, τέτοια ώστε:

- n≥|R|≥|B|.
- Κάθε στοιχείο του R κυριαρχείται από κάποιο στοιχείο του B.
- Κάθε στοιχείο του B κυριαρχεί σε κάποια στοιχεία του R.

- Τα σύνολα R, B είναι από την αρχή ταξινομημένα ως προς τη συντεταγμένη z.

Ορίζουμε k_{RB} τον αριθμό των ζευγαριών κυριαρχίας (r, b) όπου $r \in R$ και $b \in B'$. Το πρόβλημα είναι να βρούμε έναν αλγόριθμο να υπολογίζει όλα αυτά τα ζευγάρια σε χρόνο $O(k_{RB}\text{loglogn})$ και χώρο O(n). Αν το καταφέρουμε αυτό, τότε ο four-dimensional dominance reporting αλγόριθμος θα τρέχει σε χρόνο O(n logn loglogn + k loglogn) και O(n) χώρο.

Για να λύσουμε το παραπάνω πρόβλημα, κάνουμε δύο κρίσιμες παρατηρήσεις.

- Αφού κάθε σημείο r του R κυριαρχείται από τουλάχιστον ένα σημείο του B, τότε κάθε σημείο r του R κυριαρχείται από κάποιο σημείο του B′ και συνεπώς $k_{RB} \geq |R| \geq |B|$.
- Είναι χρήσιμο σε κάποιο αρχικό βήμα να απομονώσουμε τα σημεία του B που είναι maximal. Ο υπολογισμός αυτός μπορεί να γίνει εύκολα σε χρόνο O(|B|loglogn) και χώρο O(n) με ένα sweep προς τα κάτω ως προς τη συντεταγμένη z. κατά τη διάρκεια του sweep κρατάμε το «σύνορο» των προβολών στο επίπεδο xy των σημείων του B που έχουμε ήδη περάσει [15,16].

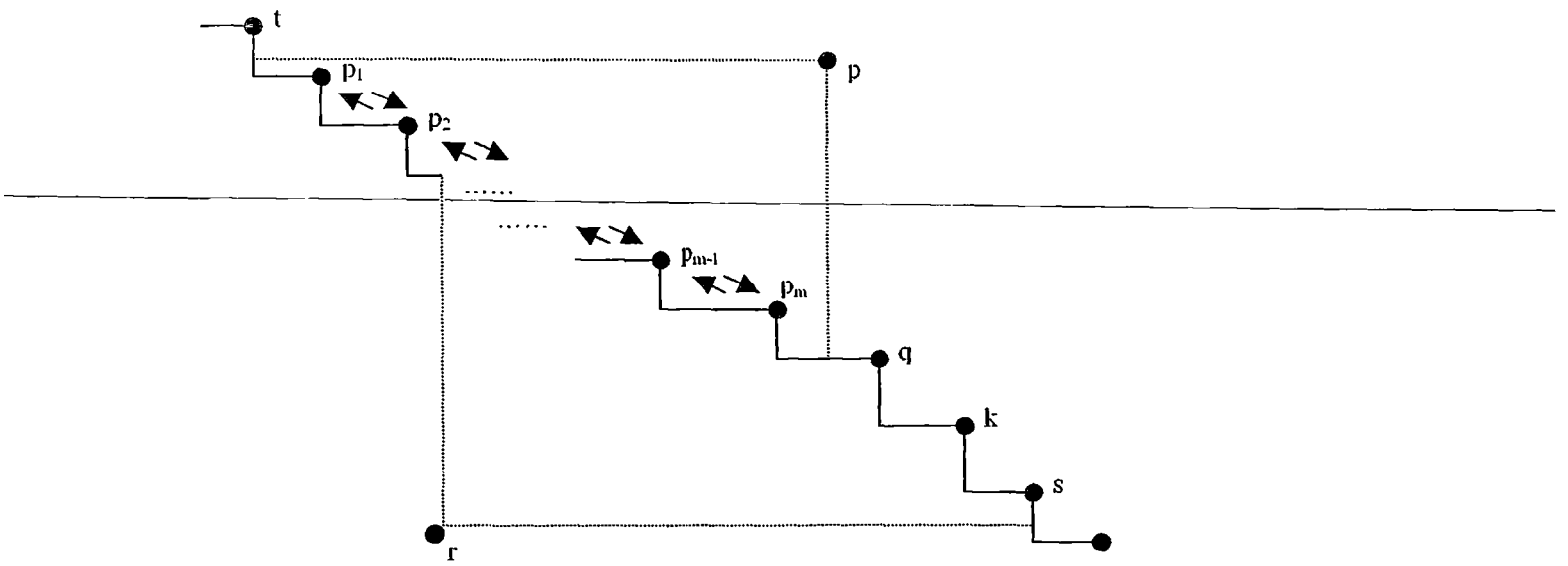
Στη συνέχεια ότι χειριζόμαστε απευθείας το σύνολο B′ των maximal σημείων του B. Θα χωματίσουμε επίσης τα σημεία του B′ μπλε και τα σημεία του R, κόκκινα. Τέλος θα θεωρήσουμε ότι έχουμε στη διάθεσή μας ένα αρχικά άδειο vEB δέντρο. Το κόστος για την αρχικοποίηση του δέντρου (O(n)) μπορεί να χεωθεί στο αρχικό βήμα προεπεξεργασίας του αλγορίθμου.

7.4 Ο αλγόριθμος γραμμικού χώρου.

Η λύση που προτείνουμε αποτελείται από δύο φάσεις. Κάθε φάση είναι μία διαπέραση κατά μήκος της z- διάστασης, ή πρώτη προς τα κάτω και η δεύτερη προς τα πάνω.

Φάση 1: Κάνουμε διαπέραση του xy- επιπέδου, μετακινούμενοι προς τα κάτω στη z διάσταση, και σταματώντας σε κάθε σημείο του B′. Κατά τη διάρκεια της διαπέρασης το σύνολο των maximal ως προς τις δύο διαστάσεις προβολών των σημείων που έχουμε ήδη επισκεφτεί. Τα maximal αυτά στοιχεία αποθηκεύονται στα φύλλα ενός αρχικά άδειου vEB δέντρου, ταξινομημένα ως προς τη x συντεταγμένη. Όταν η γραμμή του sweep επισκεφτεί ένα σημείο p που βρίσκεται έξω από το σύνορο ($p \in B'$) το δέντρο T ενημερώνεται ως εξής: Ψάχνουμε στο T για τη x-συντεταγμένη του p

και σβήνουμε από το Τ όλα τα σημεία του συνόρου των οποίων οι xy-προβολές κυριαρχούνται από την προβολή του p. Στη συνέχεια εισάγουμε το p Σα νέο στοιχείο του συνόρου. Έστω p_1, p, ..., p_m τα σημεία που σβήστηκαν λόγω της εισαγωγής του p, ταξινομημένα ως προς τη x-συντεταγμένη. Αποθηκεύουμε τη διατεταγμένη ακολουθία p, p_2, ..., p_m σε μία διπλή διασυνδεδεμένη λίστα την οποία καλούμε L(p). Αποθηκεύουμε στο p δύο δείκτες, τους p->first και p->last που δείχνουν στο πρώτο και στο τελευταίο στοιχείο της λίστας L(p) αντίστοιχα (σχήμα 1)



Σχήμα 1

Ο χώρος που απαιτείται για όλες αυτές τις λίστες είναι O(|B'|). Σε κάθε σημείο p ∈ B' αποθηκεύονται τέσσερις δείκτες, p->first, p->last, p->succ και p-> pev. Οι δείκτες p->prev και p->succ χρησιμοποιούνται για την αναπαράσταση της διπλής διασυνδεδεμένης γραμμικής λίστας στην οποία ανήκει το p. Οι λίστες θα χρησιμοποιηθούν στη φάση 2 για την αναφορά των ζευγαριών κυριαρχίας. Είναι χρήσιμο εδώ να σημειώσουμε ότι μετά το τέλος της φάσης 1, τα στοιχεία που είναι αποθηκευμένα στο Τ και στις λίστες L() αναπαριστούν το σύνολο των B' εκδόσεων του Τ σε κάθε σημείο σταθμό της διαπέρασης. Μπορούμε να θεωρήσουμε ότι κάθε έκδοση του Τ είναι indexed από τη z-συντεταγμένη του αντίστοιχου σταθμού του sweep.

Φάση 2: Για κάθε σημείο p του B' αρχικοποιούμε μια άδεια λίστα C_p για την αποθήκευση των κόκκινων σημείων και αποθηκεύουμε σε κάθε κόκκινο σημείο r το δείκτη r->stop. Ο δείκτης r->stop χρησιμοποιείται για να designate τις διαφορετικές

χρονικές περιόδους στις οποίες το r είναι αποθηκευμένο στις λίστες διαφορετικών μπλέ σημείων. Διαπερνούμε τα σημεία $R \cup B'$ προς τα πάνω στη z-διάσταση χρησιμοποιώντας τις λίστες L() για την επανακατασκευή του συνόρου, και για την αναφορά των ζευγαριών κυριαρχίας. Αρχικά η γραμμή του sweep είναι στο σημείο που έχει τη μικρότερη z-συντεταγμένη και στο δέντρο T είναι αποθηκευμένο το τελικό σύνορο της φάσης 1.

Όταν η γραμμή του sweep φτάσει σε ένα κόκκινο σημείο r κάνουμε τα ακόλουθα:

Αφού $r \in R$ τουλάχιστον ένα στοιχείο του B' κυριαρχεί πάνω στο R και έτσι το R βρίσκεται μέσα από το σύνορο. Αρχίζουμε να προχωράμε κατά μήκος του συνόρου ξεκινώντας από το σημείο που βρίσκεται ακριβώς δεξιά του r, μέχρι να φτάσουμε σε ένα μπλε σημείο του οποίου η xy-προβολή δεν κυριαρχεί της xy προβολής του r, ή μέχρι να φτάσουμε στο τέλος του συνόρου. Έστω $p_1, p_2, \ldots, p_m$ τα σημεία του συνόρου που κυριαρχούν πάνω στο r, και από αυτά τα σημεία έστω $q(q_x, q_y, q_z)$ το σημείο με τη μεγαλύτερη z-συντεταγμένη. Τότε:

Αποθήκευσε το r στο C_q και θέσε r->stop:=q_z.

Για κάθε σημείο $p \in \{p_1, p_2, .., p_m\}-\{q\}$ κάλεσε την Traverse(p, r).

Η ρουτίνα Traverse() χρησιμοποιείται για να προσπελάσει τις λίστες L() που περιέχουν τα στοιχεία που κυριαρχούν πάνω στο r και Δε βρίσκονται στην έκδοση του T που αντιστοιχεί στο r->stop. Η Traverse περιγράφεται από τον ακόλουθο ψευδοκώδικα.

```
Traverse(p,r)
Begin
If      pz<r->stop then {
        Report (r,p) as a dominance pair
        If      p->first dominates r then       {
                t:=p->first
                while t<>null do
                        Traverse(t, r);
                        t:=t->next;
                        if t does not dominate r then t:=null;
                od }
        else    if p->last dominates r τηεν {
                t:=p->last;
```

```
            while t<>null do
                Traverse(t,r);
                T:=t->prev;
                If t does not dominate r then t:=null;
            Od; }
        }
end;
```

Όταν η γραμμή της διαπέρασης επισκεφτεί ένα σημείο $b \in B'$, κάνουμε τα ακόλουθα:

- Χρησιμοποιώντας τη λίστα L(b) ακυρώνουμε τις αλλαγές που έγιναν στο σύνορο όταν επισκεφτήκαμε το p κατά τη διάρκεια της διαπέρασης της φάσης 1.
- Για κάθε $r \in C_b$

1. Αναφέρουμε το ζευγάρι (r, b) σαν ένα ζευγάρι κυριαρχίας
2. Ψάχνουμε με τη x-συντεταγμένη του r στο vEB tree αν η xy-συντεταγμένη του r βρίσκεται μέσα από το τρέχον σύνορο. Αν βρίσκεται μέσα τότε βρίσκουμε το σύνολο $\{p_1, p_2, \ldots, p_m\}$ των σημείων που κυριαρχούν πάνω στο r, όπως και στην προηγούμενη περίπτωση. Τότε:
   2.1 Αποθηκεύουμε το r στο C_q και θέτουμε r->stop = q_z.
   2.2 Για κάθε $p \in \{p_1, p_2, \ldots, p_m\}-\{q\}$ καλούμε την Traverse(p,r).

Είναι φανερό ότι αφού κάθε χρονική στιγμή κάθε κόκκινο σημείο είναι αποθηκευμένο στη λίστα ενός και μόνο ενός σημείου $p \in B'$, ο χώρος που απαιτείται για τη διατήρηση των λιστών είναι $O(|R|+|B'|)=O(n)$.

Στη συνέχεια θα δείξουμε ότι ο παραπάνω αλγόριθμος είναι σωστός και θα μελετήσουμε την πολυπλοκότητα χρόνου. Έστω p ένα σημείο του B′ και έστω L(p) η λίστα που περιέχει τα στοιχεία $p_1, p_2, \ldots, p_m$ τα οποία σβήστηκαν από το σύνορο όταν επισκεφτήκαμε το p κατά τη διάρκεια του sweep στη φάση 1. Αν θεωρήσουμε τα σημεία $p_1, p_2, \ldots, p_m$ σα γιους του p τότε το σύνολο των L() λιστών μπορεί να θεωρηθεί σαν ένα δάσος F από δέντρα. Η ρίζα κάθε δέντρου είναι ένα σημείο του B′ του οποίου η xy-προβολή δεν κυριαρχείται από τν προβολή κανενός άλλου σημείου του B′.

Λήμμα 6.1: Έστω r ένα σημείο του R και έστω $p_1, p_2, \ldots, p_m$, τα σημεία του B′ τα οποία κυριαρχούν πάνω στο r την πρώτη φορά που επισκεφτόμαστε το r κατά τη διάρκεια του sweep προς τα πάνω στη φάση 2. Τότε, όλα τα σημεία του B′ που

κυριαρχούν πάνω στο r είναι τα m υποδέντρα του F που έχουν ρίζες τα σημεία p_1, p_2, …, p_m.

Απόδειξη: Έστω $w(w_x, w_y, w_z)$ ένα σημείο που κυριαρχεί πάνω στο r. Τότε, $w_x \geq r_x$ και $w_y \geq r_y$ και $w_z \geq r_z$. Έστω w_1 ο πατέρας του w στο F. Αν $w_{1z} < r_z$ τότε το w είναι σίγουρα ένα από τα p_1, p_2, …, p_m. Ας υποθέσουμε τώρα ότι $w_{1z} \geq r_z$. Αφού $w_{1x} \geq r_x$ και $w_{1y} \geq r_y$ συμπεραίνουμε ότι το w_1 κυριαρχεί πάνω στο r. Αντικαθιστούμε το w με το w_1 και συνεχίζουμε να το κάνουμε αυτό αναδρομικά μέχρι να βρούμε το πρώτο σημείο w_i του οποίου ο πατέρας w_{i+1} στο F έχει z-συντεταγμένη μικρότερη από την τιμή r_z. Τότε σίγουρα το w_i είναι ένα από τα p_1, p_2, .., p_m και επειδή τα σημεία w, w_1, w_2, …, w_i ορίζουν ένα μονοπάτι στο F, το λήμμα ισχύει. □

Το παραπάνω λήμμα μας λέει ότι το πρόβλημα της αναφοράς όλων των σημείων που κυριαρχούν πάνω στο r είναι η διαπέραση αυτών των υποδέντρων με ένα τρόπο που δεν είναι χρονοβόρος. Ας φανταστούμε, για παράδειγμα ε΄να συγκεκριμένο σημείο p_i η λίστα του οποίου είναι η $L(p_1)$. Η ιδανική λύση θα ήταν να διαπεράσουμε τη λίστα ξεκινώντας από το p->first ή από το p->last και να επισκεφτούμε όλα σημεία της που κυριαρχούν πάνω στο r. Με τον τρόπο αυτό το κόστος για τη διαπέραση θα ήταν O(k). Όμως αυτό δεν είναι πάντα δυνατό, αφού τα σημεία της λίστας $L(p_i)$ που κυριαρχούν πάνω στο R μπορούν να εμφανίζονται σα μία υποακολουθία της λίστας $L(p_i)$ που ξεκινάει και τελειώνει στο εσωτερικό της λίστας. Στην περίπτωση αυτή, η διαπέραση της λίστας από την αρχή ΄το τέλος της εμπεριέχει μεγάλο κόστος χρόνου. Η κρίσιμη παρατήρηση που λύνει το πρόβλημα αυτό είναι ότι για ένα συγκεκριμένο κόκκινο σημείο r και για τη χρονική περίοδο που το σημείο αυτό ανήκει σε μία συγκεκριμένη λίστα C_p η περίπτωση αυτή μπορεί να συμβεί για ένα μόνο σημείο και ειδικότερα για το σημείο p στου οποίου τη λίστα είναι αποθηκευμένο το r.

Λήμμα 6.2: Έστω p ένα σημείο του B′ που το επισκεφτήκαμε σε μία κλήση της υπορουτίνας Traverse(p, r). Αν πρέπει να επισκεφτούμε τη λίστα L(p) τότε τα σημεία της λίστας L(p) που κυριαρχούν πάνω στο r αποτελούν ία υποακολουθία της λίστας L(p) που ξεκινάει είτε από το p->first είτε από το p->last.

Απόδειξη: Αφού πρέπει να επισκεφτούμε τα σημεία της λίστας L(p), πρέπει να είναι p_z<r->stop. Έστω q το σημείο του B′ στου οποίου τη λίστα είναι αποθηκευμένο το r τη χρονική στιγμή που καλούμε τη ρουτίνα Traverse(). Είναι φανερό ότι r->stop=q_z. Αφού:

- Το σημείο q έχει μεγαλύτερη τιμή στη z-συντεταγμένη από το p και
- Τα σημεία q και p εμφανίζονται μαζί στην ίδια έκδοση του T (έστω t η έκδοση αυτή).

Προκύπτει ότι το σημείο q και τα σημεία της λίστας L(p) εμφανίζονται μαζί στην ίδια έκδοση του T με αριθμό ανάμεσα στους q_z και t. Έστω c ο αριθμός αυτός (ο αριθμός της έκδοσης του δέντρου T). Είναι προφανές ότι βρίσκεται είτε αριστερά είτε δεξιά από όλα τα σημεία της λίστας L(p) στην έκδοση του T με αριθμό c. Αν το q βρίσκεται δεξιά από όλα τα σημεία της λίστας L(p) τότε όλα αυτά τα σημεία κυριαρχούν πάν στο r εκτός από την περίπτωση όπου το p είναι το πιο αριστερό σημείο που κυριαρχεί στο r στο σύνορο (στην έκδοση του T με αριθμό t). Στην περίπτωση αυτή τα στοιχεία της λίστας που κυριαρχούν πάνω στο r εμφανίζονται σα μία συνεχής ακολουθία που ξεκινάει από το p->last (σχήμα 1). Αν το q βρίσκεται αριστερά από όλα τα σημεία της λίστας L(p), τότε όλα τα σημεία της λίστας L(p) κυριαρχούν πάνω στο r εκτός από την περίπτωση όπου το p είναι το πιο δεξιό σημείο που κυριαρχεί πάνω στο r, στο σύνορο (στην έκδοση του T με αριθμό t). Στην περίπτωση αυτή τα σημεία της λίστας L(p) που κυριαρχούν πάνω στο r είναι μία συνεχής ακολουθία που ξεκινάει από το p->first. Από τα παραπάνω προκύπτει ότι το λήμμα ισχύει. □

Ας δούμε τώρα την πολυπλοκότητα χρόνου. Για ένα σημείο p στο B′, ορίζουμε low(p) τη z-συντεταγμένη του σημείου q που είναι ο πατέρας του p στο δάσος F, και up(p) τη z-συντεταγμένη του p. Καλούμε [low(p) : up(p)) την περίοδο ζωής του p, αφού δηλώνει το διάστημα των εκδόσεων του T στις οποίες το p εμφανίζεται σα φύλλο του T. Θα φράξουμε το χρόνο του αλγορίθμου, μελετώντας κάθε σημείο του R ξεχωριστά.

Έστω r ένα σημείο του R και έστω k ο αριθμός των σημείων του B′ που κυριαρχούν πάνω στο R. Κατά τη διάρκεια του sweep της φάσης 2, το r εμφανίζεται σε έναν αριθμό από ξεχωριστές C_q λίστες. Έστω t_1, t_2, …, t_s οι χρονικές στιγμές κατά τη διάρκεια του sweep κατά τις οποίες το σημείο r αλλάζει C_q λίστες όπου t_1 είναι η πρώτη φορά που το r εμφανίζεται στο sweep, t_{s-1} είναι η τελευταία φορά που το t αλλάζει C_q λίστα και t_s είναι η up(q) τιμή για το σημείο q στού οποίου τη λίστα είναι αποθηκευμένο το r για τελευταία φορά. Έστω q_{ti} ($1 \le i \le s-1$) το σημείο στου οποίου τη λίστα είναι αποθηκευμένο το σημείο r τη χρονική στιγμή t_i. Τα t_1, t_2, …, t_s τα βλέπουμε και σαν z-συντεταγμένες και έτσι $t_1 < t_2 < \ldots < t_s$. έτσι προκύπτει η επόμενη παρατήρηση.

Λήμμα 6.3: Έστω q ένα σημείο που κυριαρχεί πάνω στο r. Τότε, το διάστημα ζωής του q περιέχει το πολύ έναν από τους χρονικούς σταθμούς t_1, t_2, …, t_s.

Απόδειξη: Αφού σε κάθε χρονικό σταθμό στον οποίο το r αλλάζει C_q λίστα αποφασίζουμε να αποθηκεύσουμε το r στη C_q λίστα του σημείου που κυριαρχεί πάν στο r στην τρέχουσα έκδοση του T, και έχει τη μεγαλύτερη z-συντεταγμένη, η απόδειξη προκύπτει εύκολα με επαγωγή. □

Το παραπάνω λήμμα μας λέει ότι μπορούμε να χωρίσουμε τα στοιχεία του B′ που κυριαρχούν πάνω στο r σε κλάσεις σύμφωνα με τη σχέση που έχει το διάστημα ζωής τους με τους χρονικούς σταθμούς t_1, t_2, .., t_s. Έσω k_{ti} ο αριθμός των στοιχείων που

κυριαρχούν πάνω στο r και το διάστημα ζωής τους περιέχει το χρονικό σταθμό t_i. Επίσης, έστω k_{ti}' ο αριθμός των σημείων που κυριαρχούν στο r και το διάστημα ζωής τους βρίσκεται ακριβώς ανάμεσα στις τιμές t_i και t_{i+1}. Τότε έχουμε $k_{ts}=k_{ts}'=0$ και $(k_{t1}+\ldots+k_{ts-1}) + (k_{t1}'+\ldots+k_{ts-1}')=k$ όπου k είναι ο αριθμός των σημείων που κυριαρχούν πάνω στο r. Ας δούμε τώρα το κόστος χρόνου για το σημείο r. Είναι φανερό ότι ο τελευταίος χρονικός σταθμός δίνει κόστος O(loglogn). Σε κάθε άλλο χρονικό σταθμό t_i, έχουμε:

Το ψάξιμο στο δέντρο T, το οποίο κοστίζει $(\text{loglogn}+k_{ti})$.

Τις κλήσεις της ρουτίνας Traverse() οι οποίες κοστίζουν $O(k_{ti}+k_{ti}'+k_{ti+1})$. Αφού για $1\leq i\leq s-1$ έχουμε $k_{ti}\geq 1$ (το q_{ti} ανήκει στην έξοδο), προκύπτει ότι το συνολικό κόστος χρόνου για ένα σημείο είναι O(kloglogn). Αθροίζοντας το κόστος αυτό για όλα τα κόκκινα σημεία, συμπεραίνουμε ότι ο προτεινόμενος αλγόριθμος λύνει το πρόβλημα της παραγράφου 7.3 σε χρόνο $O(k_{RB}\text{loglon})$ και χρησιμοποιεί γραμμικό χώρο

8. Αναφορές

[1] A. Anderson. Sublogarithmic searching without multiplications. *Proc. 36th IEEE FOCS, pages* 655-663, 1995.

[2] D.T. Lee and F.P. Preparata, An improved algorithm for the rectangle enclosure problem, *Journal of Algorithms*, 3 (1982) 218-224.

[3] Dan E. Willard. Log-logarithmic worst case range queries are possible in space Θ(n). *Information processing Letters*. 17 (1983) 81-84.

[4] Dan E. Willard. New Trie Data Structures Which Support Very Fast Search Operations.

[5] H. Edelsbrunner, M.H. Overmars, On the equivalence of some rectangle problems, *Information Processing Letters*, 14, (1982) 124-127.

[6] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput Syst. Sci.*, 47:424-436, 1994.

[7] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. System Sci.*, 48 (1994), pp. 533-551.

[8] M. L. Fredman, J. Komolos and E. Szemeredi. Storing a sparce table with O(1) worst case access times. *Proceedings of the 23th IEEE Symposium on the Foundations of Computer Science*, pp 165-169, 1982.

[9] P. Bozanis, N. Kitsios , C. Makris and A. Tsakalidis, The space-optimal version of a known rectangle enclosure reporting algorithm, *Inform. Process. Lett.* 61 (1) (1997) 35-41.

[10] P. Gupta, R. Janardan, M. Smid, B. Dasgupta, The rectangle enclosure and point-dominance problems revisited. *Proceedings of the 11th Annual Sympoisum on Computational Geometry*, (1995), 162-171.

[11] P. Gupta, R. Janardan, M. Smid, B. Dasgupta, The rectangle enclosure and point-dominance problems revisited to appear in Int. *Journal of Comp. Geom. and Applications* 7(5) (1997) 437 - 457.

[12] P. van Emde Boas, R. Kaas and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Syst Theory*, 10:99-127, 1977.

[13] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters* 6(3):80-82, 1977.

[14] R. Carlsson. Algorithms in a Restricted Universe. *Ph. D. Thesis*, University of Waterloo, Canada, 1984.

[15] R. Karlsson and M. Overmars, Normalized Divide-and-Conquer: A Scaling Technique for solving multimdimensional problems. *Information Processing Letters*, (1987) 307-312.

[16] R. Karlsson and M. Overmars, Scanline Algorithms on a grid, BIT 28 (1988), 227-241.

[17] R. Raman. Improved data structures for predecessor queries in integer sets. *Manuscript*, 1995.

[18] Version of a known rectangle enclosure reporting algorithm. *Information Processing Letters*, 61 (1) (1997) 35 - 41.