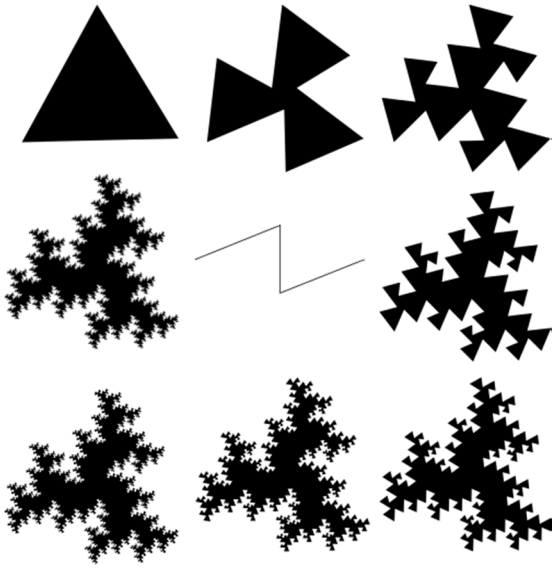
9 ERIOTSTARBELISED PUUD

Raamatu esimeses osas uuriti mitmesuguseid üldotstarbelisi andmestruktuure nagu massiivid, pinud ja järjekorrad. Enamasti võimaldavad need andmestruktuurid tööd üksikute elementidega ja nende tavalisteks operatsioonideks ongi elemendi lisamine, muutmine, lugemine või eemaldamine.



Nii arvutiteaduses üldiselt kui ka võistlusprogrammeerimises konkreetsemalt on aga sageli vaja andmestruktuure, mis võimaldavad mitte ainult üksikväärtuste, vaid ka suuremate väärtuste hulkadega manipuleerimist. Näitena võib tuua operatsioonid nagu „mis on suurima väärtusega element 25. ja 137. elemendi vahel?“ või „Suurenda kõiki väärtusi 33. elemendist 99. elemendini“.

Sellist laadi ülesannete lahendamiseks sobivad hästi mitmesugused puukujulised andmestruktuurid. Nende efektiivset esitamist ja erinevate operatsioonide realiseerimise võimalusi uurimegi käesolevas peatükis.

9.1 LÕIKUDE PUU

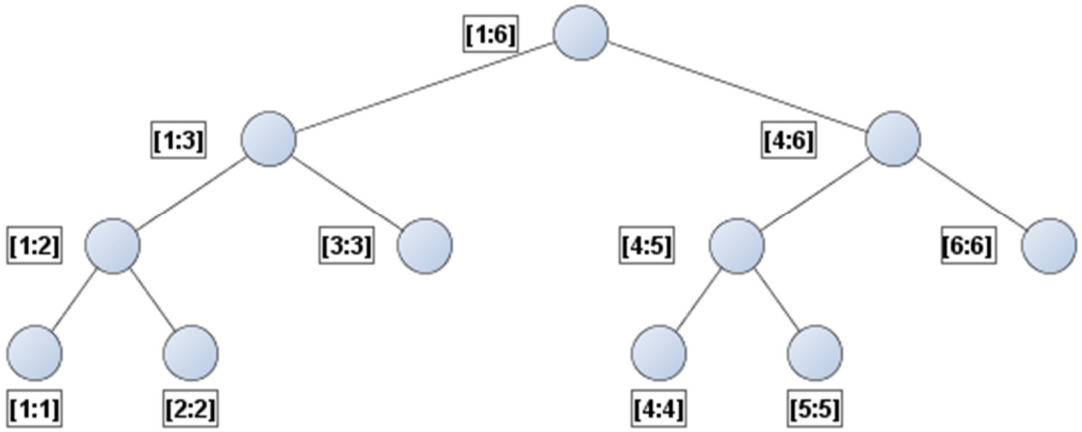
Vaatame ülesannet, kus on antud jada ja on vaja leida minimaalne element mingil suvalisel lõigul selles jadas. Sellist ülesannet tuntakse kui *Range Minimum Query (RMQ)*.

Kui sellist miinimumi leidmist on vaja teha vaid ühe korra, siis on loomulikult optimaalne käia antud lõik läbi ja leida minimaalne väärtus selles. Selle keerukus on lõigu pikkuse suhtes lineaarne: $O(n)$, kus n on jada pikkus. Kui aga päringuid on palju, siis muutub selline lähenemine liiga aeglaseks: m päringu korral on keerukuseks $O(m * n)$. Kui m on väike ja/või kui andmed muutuvad väga sagedasti, siis võib selline lihtne lähenemine olla optimaalne – kuna mingit uut struktuuri luua pole vaja, siis initsialiseerimisaeg puudub, samuti on andmete muutmine keerukusega $O(1)$.

Kui $m >> n$, siis on kasulikum täita tabel, kus kirjas kõikide lõikude miinimumid. Selle koostamise keerukuseks on $O(n^2)$, kuid igale päringule vastuse leidmise keerukus on $O(1)$. Aga ka see lähenemine võib osutuda liiga aeglaseks, eriti juhul, kui andmed päringute vahel muutuda võivad.

Võtame appi kahendpuu, mille igas tipus säilitatakse teatud lõigu miinimumi. Puu juurtipus on kogu jada $[1,\ n]$ miinimum, lehtedes jada üksikud väärtused, sest lõigu $[i,\ i]$ miinimum asub jadas kohal i.

Vahetippudes on vasaku alluva ja parema alluva lõikude ühend ehk lõik vasaku alluva vasakpoolsest indeksist parema alluva parempoolse indeksini:



Lõikude puu kuueelemendilise jada jaoks. Iga tipu juures on näidatud, millise lõigu kohta selles tipus informatsiooni säilitatakse.

Sellist puud nimetatakse **lõikude puuks** (i.k *segment tree*) ja selle mõtles välja Carnegie-Melloni ülikoolis töötanud ameerika arvutiteadlane Jon Louis Bentley aastal 1977.

9.1.1 Puu loomine jadast

Võtame näitejadaks järgmise jada:

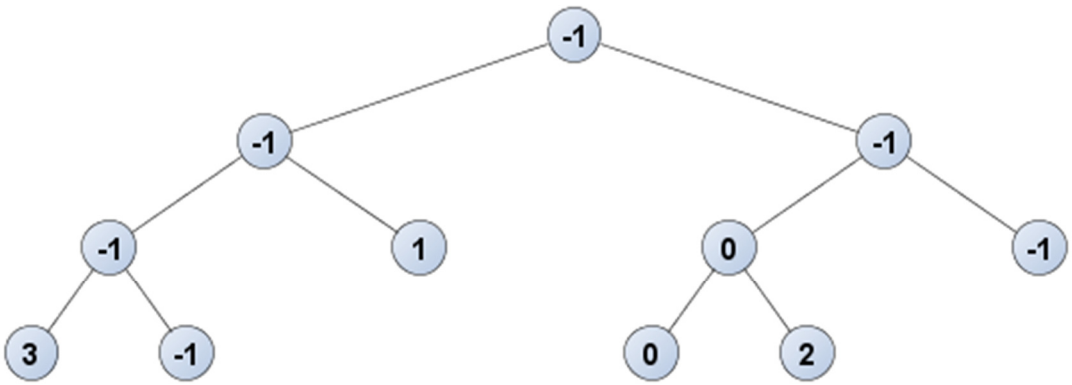
1	2	3	4	5	6
3	-1	1	0	2	-1

Esimeseks ülesandeks on algandmetest puu koostamine. Selleks on vaja kõigepealt otsustada, millist informatsiooni tippudes vaja hoida on. Antud ülesande puhul on mõttekas hoida iga lõigu kohta minimaalset väärtust sellel lõigul või uuritavas lõigus minimaalse väärtusega elemendi indeksit. Kasutame siin näidetes esimest.

Meeldetuletuseks teisest peatükist: kahendpuud, saab seda efektiivselt hoida massiivis. Kui alustada puud indeksilt 1, siis tipu k vasak alluv on massivis kohal $2 * k$ ja parem alluv kohal $2 * k + 1$. Igas tipus hoiame tema alluvate miinimumi indeksit.

Üheks sellise puu loomise võimaluseks on teha seda rekursiivselt:

```
void loo_puu(int tipp, int algus, int lopp)
{
    if (algus == lopp)
        LP[tipp] = A[algus]; //LP - lõikude puu
    else {
        //täida vasak ja parem alampuu
        loo_puu(2 * tipp, algus, (algus + lopp) / 2);
        loo_puu(2 * tipp + 1, (algus + lopp) / 2 + 1, lopp);
        //vasaku ja parema alampuu miinimum
        int k = 2 * tipp;
        LP[tipp] = min(LP[k], LP[k + 1]); //A - algandmed
    }
}
```



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-1	-1	-1	-1	1	0	-1	3	-1	∞	∞	0	2	∞	∞

Eelmisel joonisel toodud jadale vastav miinimumide indekseid sisaldav lõikude puu abstraktselt ja massiivina.

Puu loomise keerukuseks on $O(n)$, kuid tasub meeles pidada, et see on ühekordne protseduur.

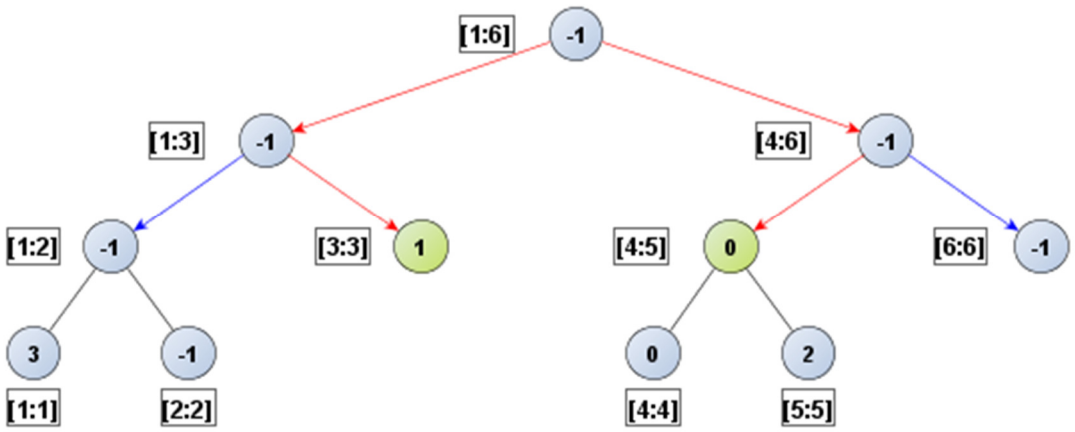
9.1.2 Lõigul väärtuse leidmine

Kui puu on valmis, siis järgmiseks protseduuriks on päringutele vastamine ehk väärtuse leidmine mingil suvalisel lõigul $[i,\ j]$.

Väärtuse leidmist alustame juurest, kus on olemas kogu jada miinimumi indeks. Kui aga lõik $[i,\ j]$ ei kata kogu jada, siis kogu jada miinimumiga ei ole midagi kasulikku peale hakata. Edasi otsimiseks on kolm võimalust:

1. Otspunkt j asub vasakus alampuus ($j \leq \frac{n}{2}$). Sellisel juhul asub kogu uuritav lõik vasakus alampuus ja võime sealt jätkata.
2. Otspunkt i asub paremas alampuus ($i > \frac{n}{2}$). Sellisel juhul asub kogu otsitav lõik paremas alampuus ja võime sealt jätkata.
3. Otspunkt i asub vasakus alampuus ja otspunkt j paremas alampuus. Sellisel juhul jagame lõigu kaheks $[i,\ \frac{n}{2}]$ ja $[\frac{n}{2}+1,\ j]$ ja otsime kummastki alampuust vastavat lõiku. Lõpuks on vaja leida miinimum vasakust ja paremast puust leitud miinimumidest.

Kui jõuame tippu, kus on säilitatud otsitava lõigu miinimum, siis tagastame kohe selle.



Lõigu [3:5] väärtuse leidmine. Rohelisega on märgitud need tipud, mille väärtus on oluline miinimumi leidmiseks uuritaval lõigul.

Otsime näiteks minimaalset väärtust lõigul [3, 5]. Alustame juurest. Esimene otspunkt 3 asub vasakus alampuus ja lõigu teine otspunkt 5 paremas alampuus. Vasak alampuu katab lõigu [1, 3]. Kuna ka see ei kuulu täielikult otsitavasse lõiku [3, 5], tuleb vaadata selle lõigu alampuid. Vasakul on lõik [1, 2], mis ei kattu üldse otsitava lõiguga. Selle väärtusega ei ole vaja arvestada ega siit edasi otsida. Lõigu [1, 3] parem alampuu [3:3] sisaldub tervenisti otsitavas lõigus [2, 5] ja seega tuleb selle väärtusega arvestada.

Juure parempoolses alampuus on lõik [4, 6], mis ei kuulu tervenisti otsitavasse lõiku, seega on taas vaja vaadata selle alampuid. Vasakpoolne alampuu [4, 5] aga kuulub tervenisti otsitavasse lõiku, saame kasutata siin salvestatud väärtust ilma selle alampuudele liikumata. Lõik [6, 6], mis on lõigu [4, 6] paremaks alampuuks, ei oma ühisosa lõiguga [3, 5], sellega pole vaja arvestada. Vastuse saab seega kokku tippudes [2, 2] ja [3, 4] salvestatud väärtustest. Miinimumi leidmise ülesandes on selleks vähim nendest.

Funktsioon, mis seda teeb:

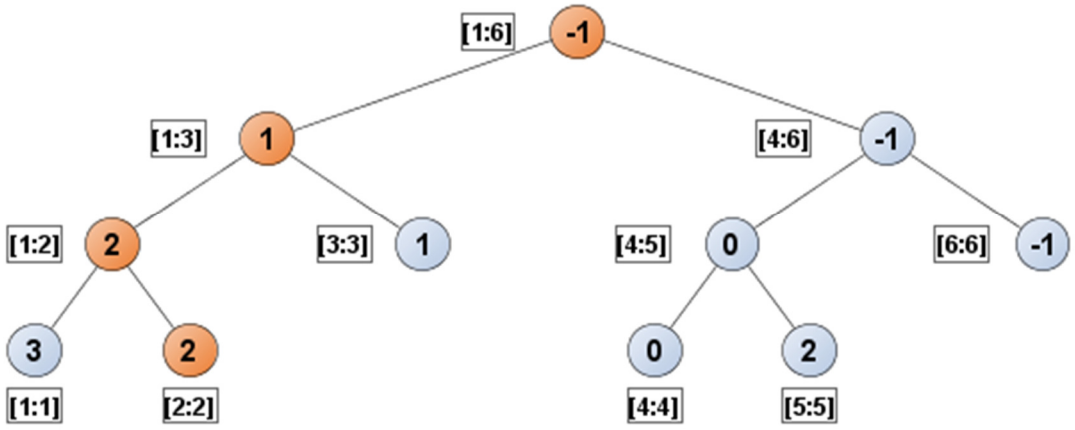
```
int otsi(int tipp, int algus, int lopp, int i, int j)
{
    int vv, pv;

    if (i > lopp || j < algus)
        return maxInt;    // seda intervalli ei kasuta

    if (algus >= i && lopp <= j) // Kui kogu intervall kattub,
        return LP[tipp]; // tagasta väärtus selles sõlmes.

    vv = otsi(2*tipp, algus, (algus+lopp) / 2, i, j); //leia sobiv intervall vasakust
    pv = otsi(2*tipp + 1, (algus+lopp) / 2 + 1, lopp, i, j); // ja paremast alampuust

    // tagastame sobiva

    if (vv <= pv)
        return vv;
    return pv;
}
```

Üksikoperatsiooni keerukuseks on $O(log\ n)$.

9.1.3 Väärtuse muutmine

Eriti oluline on lõikude puu siis, kui väärtused jadas võivad muutuda. Väärtuse muutmine on samuti keerukusega $O(log\ n)$, kuna peame uuendama ainult lehe ja selle eellaste väärtusi.

Näiteks soovime elemendile indeksil 2 anda väärtuseks -1 asemel 2. Kindlasti võib selline muudatus mõjutada juurt. Kuna element 2 kuulub juure vasakusse alampuusse, tuleb muuta ka seda ehk lõigu [1, 3] väärtust, parem alampuu [4, 6] jääb aga samaks. Samuti tuleb muuta lõikude [1, 2] ja [2, 2] väärtusi.



Väärtuse indeksil 2 muutmine. Punasega on märgitud need tipud, mille väärtused tuleb üle arvutada.

Klassikaline võte on taas tulla puus ülevalt alla ja rekursiivselt muuta nende lõikude väärust, mis sisaldavad muudetavat elementi:

```
void uuenda(int tipp, int algus, int lopp, int id)
{
    if (algus == lopp) { //Leht
        LP[tipp] = A[id]
        return;
    }
    else {
        int kk = (algus + lopp) / 2;
        if (algus <= id && id <= kk) {
            uuenda(2 * tipp, algus, kk, id); // vasak alampuu
        }
        else {
            uuenda(2 * tipp + 1, kk + 1, lopp, id); //parem alampuu
        }
        // vasak ja parem kokku
        LP[tipp] = min(LP[2*tipp], LP[2*tipp + 1]);
    }
}
```

9.1.4 Väärtuste muutmine tervel lõigul

Mõnikord on vaja korraga uuendada väärtusi terves mingis lõigus. Loomulikult on võimalik väärtustada kõik lehed ükshaaval. Veidi parem on kirjutada eraldi funktsioon:

```
void uuenda_loik(int tipp, int algus, int lopp, int algusID, int loppID)
{
    // ei ole piirides ja see haru ei vaja seega uuendamist
    if (algus>lopp || algus>loppID || lopp<algusID)
        return;

    // leht
    if (algus == lopp) {
         LP[tipp] = A[id]
         return;
    }

    // Kui ei ole leht, töötleme alluvad
    int kk = (algus + lopp) / 2;
    uuenda_loik(tipp * 2, algus, kk, algusID, loppID);
    uuenda_loik(tipp * 2 + 1, kk + 1, lopp, algusID, loppID);

    // alluvate tulemustest miinimum
    LP[tipp] = min(LP[2*tipp], LP[2*tipp + 1]);
}
```

9.2 LAISK VÄÄRTUSTAMINE LÕIKUDE PUUDES

Iga tipp lõikude puus hoiab endas infot mingi vahemiku kohta. Kui seda tippu on vaja uuendada, siis on vaja uuendada ka selle tipu alluvaid. Programmeerimisülesannetes tuleb sageli ette, et väärtusi muudetakse tervel lõigul sarnaselt, st kas antakse kogu lõigule üks ja sama väärtus või suurendatakse/vähendatakse väärtusi mingi väärtuse x võrra või korda. Sellistel juhtudel saab kergesti leida, mis on terve lõigu väärtuseks (näiteks miinimumiks või summaks), ilma et oleks vaja teada alamlõikude väärtusi.

Sellisel juhul on võimalus jätta nende alamlõikude väärtustamised lõikude puus tegemata ja teha need alles siis ja ainult siis, kui tekib vajadus kasutada konkreetseid (alam)lõike. Sellist meetodit nimetatakse **laisaks väärtustamiseks** (i.k *lazy propagation*).

Kuidagi on vaja ära tunda, millised tipud on uuendamata, et neid siis vajadusel õigete väärtustega varustada. Selleks kasutatakse eraldi massiivi, milles on iga tipu kohta vajalik informatsioon, kas ja kuidas selle tipu väärtust muuta tuleb. Algselt on iga elemendi väärtus selline, mis tähistab, et mingit muutust teha ei ole vaja.

Oletame, et on lõikude puu LP ja uuenduste jaoks massiiv U. Kui tipp t muutmist ei vaja, siis $U[t] = nil$. Muutmisfunktsiooniks on $F(U[t])$.

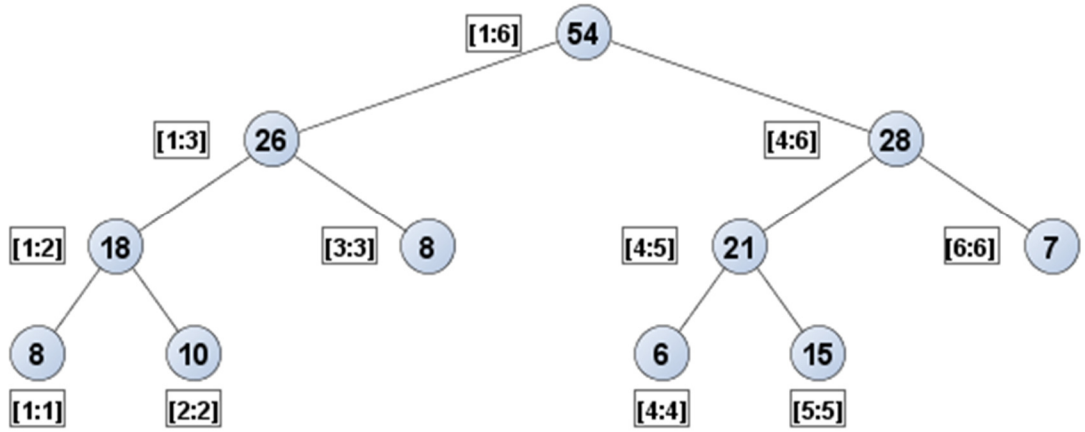
Lõikude puu elemendi muutmise algoritm on järgmine:

- Kui vaadeldaval tipp t ootab uuendamist, st $U[t] \neq nil$, muuda selle väärtus: $LP[t] = F(U[t])$.
- Kui vaadeldav vahemik sisaldub tervenisti päringus, uuenda vaadeldav tipp ja sea selle alluvatele ootel muutused.
- Kui vahemik kattub osaliselt, siis väärtusta vasak ja parem alampuu ja uuenda vastavalt saadud väärtustele.

Vaatame sellist lõikude puud, milles on igas tipus vastava lõigu elementide summa. Näiteks jadale

1	2	3	4	5	6
8	10	8	6	15	7

Vastab järgmine lõikude puu:

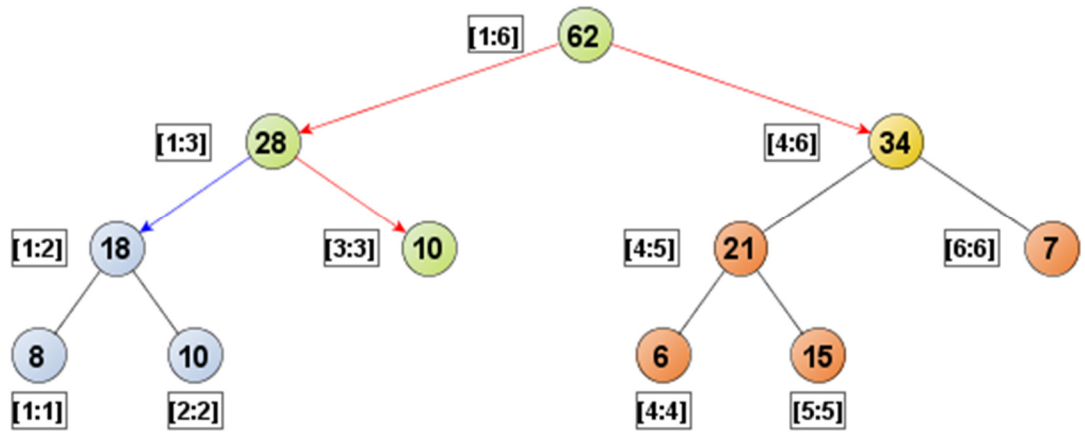


Lõikude puu, mille tippudes on vastava lõigu elementide summa. Juurtipus on kogu jada summa, lehtedes üksikud elemendid.

Edasi on meil vaja suurendada või vähendada iga elemendi väärtust teatud arvu võrra mingil etteantud lõigul. Oletame, et igale väärtusele lõigul [3, 6] liidetakse juurde 2.

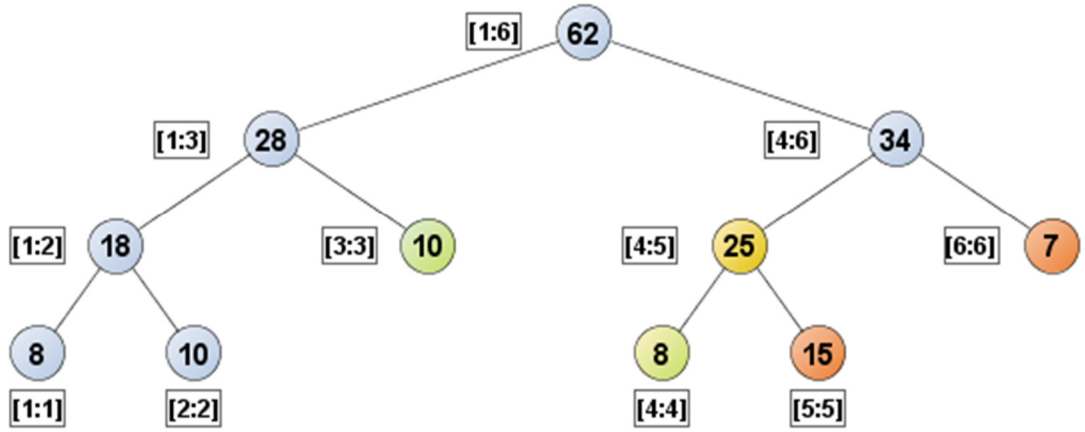
Alguses on meil uuenduste jada U elemendid määramata. Juurtipp ei asu tervenisti selles vahemikus, samuti kuna U[1] on määramata, ei ole sellel vaja teha uuendusi. Liigume juure alampuudele ning kontrollime, ega need uuendamist ei vaja. Praegu muidugi veel ei vaja. Vasak alampuu [1, 3] ei asu samuti tervenisti muudetavas vahemikus, liigume omakorda selle alampuudele. Vasak alampuu on tervenisti uuendatavast vahemikust väljas ega vaja uuendamist, sellega ei tee midagi. Parem alampuu aga on tervenisti muudetavas vahemikus, uuendame selle väärtuse ja samuti siis vahemiku [1, 3] väärtuse. Siiamaani toimub uuendamine tavapäraselt.

Vaatame aga paremat alampuud. Vahemik [4, 6] on tervenisti uuendatavas vahemikus, muudame selle väärtust ja seame alampuud muutuse ootele. Selleks U[6] = U[7] = 2. Seejärel uuendame tavapäraselt juurtipu.



Laisk väärtustamine. Igale tipule lõigus [3:6] lisatakse 2. Rohelisega on tipud, mis muudetakse tavapäraselt, kollasega on tipp, mille väärtus muudetakse (+6), aga mille alluvate väärtusi (punased) ei muudeta.

Oletame, et nüüd esitatakse päring lõigu [3, 4] kohta. Lõigu [3, 3] väärtuse saame tavapäraselt vasakust alampuust, paremas alampuus aga kui jõutakse lõiguni [4, 5], siis on kõigepealt vaja uuendada selle väärtust (+4) ning seada lõigud [4, 4] ja [5, 5] muutmist vajavateks (U[12]=U[13]=2). Seejärel liigume lõigule [4, 4], uuendame selle väärtuse (+2) ning saamegi kombineerida vastuse. Lõik [5, 5] jääb ikka muutmise ootele.



Laisk väärtustamine. Puu pärast päringut lõigu [3:4] kohta.

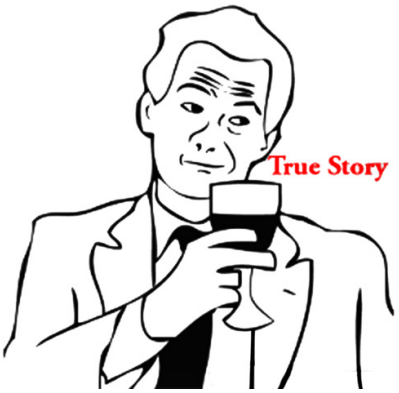
Oluline on meeles pidada, et info muudatuste kohta tuleb hoida värskena. Seega, kui muudatus on tehtud, tuleb see eemaldada muudatuste massiivist. Samuti tasub ka tähele panna, et sageli võivad muudatused kumuleeruda, enne kui neid tegelikele väärtustele rakendatakse. Näiteks kui nüüd soovime vähendada iga elemendi väärtust 3 võrra lõigul [4, 6], siis lõigu [6, 6] väärtust pole vaja vahepeal välja arvutada, lihtsalt väärtus massiivis U muutub 3 võrra väiksemaks. See tähendab, et kui enne oli U[7]=2, siis pärast on U[7] = 2-3=-1. Seega kui varem oli vaja selle tipu väärtust suurendada 2 võrra, siis pärast on seda hoopis vaja vähendada 1 võrra. Vahepealset väärtust (9) polnudki vaja välja arvutada.

Ühe mugavusena koodi kirjutamisel saab antud näite juures ära kasutada ka seda, et nulli võib vabalt liita ja lahutada, ilma et summa sellest muutuks, see tähendab, et võime alguses väärtustada muutuste massiivi nullidega ning igale tipule liita massiivis oleva väärtuse ilma eraldi kontrollimata, kas seda tegelikult vaja teha on.

Hetkel võib veel jääda segaseks, miks sellist keerukat süsteemi üldse vaja on. Päris mitmed võistlusülesanded on sellised, kus tehakse kõigepealt palju muudatusi lõikudel ning alles lõpuks esitatakse päring vastuse saamiseks. Nii muutuvad väärtused tervel lõigul mitmeid kordi, enne kui selle lõigu alampuid üldse vaja kasutada on. Vaatame selleks lähemalt kahte ülesannet toimunud võistlustelt.

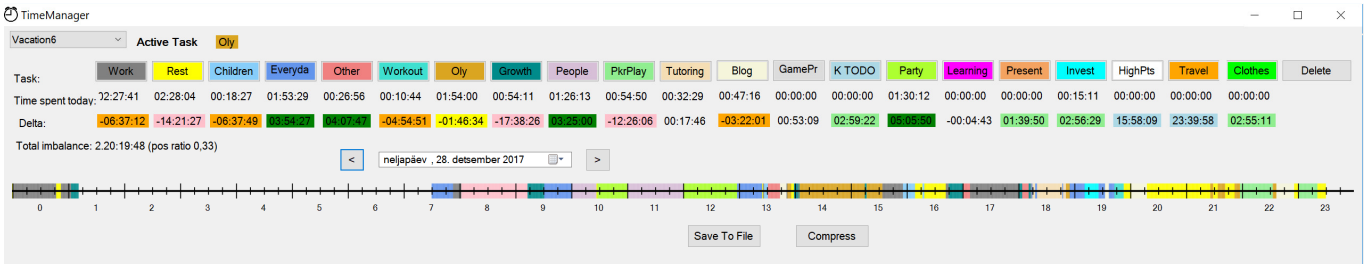
9.2.1 Paberiribade kleepimine

Selle ülesande panin ma 2014. aasta Eesti Informaatika Lahtiselt võistlusele. Originaalis on see aga hoopis päriselust, illustreerides asjaolu, et algoritmide tundmine aitab nii mõndagi „tavalist“ programmeerimisülesannet paremini lahendada.



Olen oma iseloomult alati olnud inimene, kes endale hulgaliselt erinevaid kohustusi, tegemisi, hobisid ja sebimisi teha võtab, mis viib kergesti ohuni, et mõni asi võtab liiga palju aega ning miski muu jääb seetõttu unarusse. Selle ennetamiseks kirjutasin endale programmi, kus saab erinevaid tegevusi kergesti ajajoonele märkida, näiteks hiirega algusest vastava ajalõigu algusest lõpuni vedades. Sageli on olukordi, kus mõne pikemaajalise tegevuse (nt tööpäeva) vahele tuleb märkida muid väikesi tegevusi (nt puhkepause), nii et kogutulemus on üpris fragmenteeritud. Andmete pealt saab teha igasugust huvitavat statistikat, nt uurida, millele erinevatel ajaperioodidel rohkem või vähem aega kulub.

Sisuliselt on tegu hulga ajaintervallidega, millega on vaja erinevaid lisamise, muutmise ja pärimise operatsioone sooritada. Kuni andmeid on vähe, pole väga suurt vahet, millist algoritmi kasutada, aga aastatega koguneb selliseid ajaintervalle palju tuhandeid ja ülesande lahendamiseks sobiski hästi lõikude puu.



Järgneb aga juba võistluse jaoks kohandatud sõnastus:

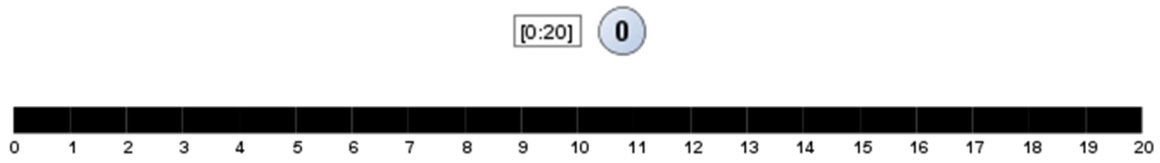
Jukul on pikk must pabeririba pikkusega L cm, millele ta kleebib teisi, värvilisi paberiribasid. Kleebitavad ribad mahuvad alati esialgse riba peale ära. Kõik ribad on sama laiusega. Iga järgmine riba varjab ära tema all olevad värvid. Leida, millised värvid ja milliste pikkustega jäävad lõppkokkuvõttes näha.
Sisendi esimesel real on antud musta riba pikkus L ($1 \le L \le 10^9$) ning kleebitavate ribade arv N ($0 \le N \le 500\ 000$). Järgmisel N real on igaühel kolm täisarvu: ühe riba värvikood K ($1 \le K \le 100$) ning selle alguse A ja lõpu B kaugus musta riba algusest ($0 \le A < B \le L$). Musta värvi kood on 0. Ainult alumine riba on must, teised on kõik värvilised.
Väljundisse väljastada lõpuks näha jäävate erinevat värvi lõikude värvid ja pikkused alates algse musta riba algusest kuni lõpuni. Järjestikused sama värvi ribad väljastada ühe lõiguna.
NÄIDE:

```
20 4
1 2 10
2 5 8
3 3 6
3 5 7
```

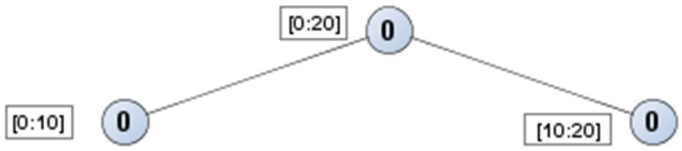
Vastus:

```
0 2
1 1
3 4
2 1
1 2
0 10
```

Iga pabeririba katab ära mingi teadaoleva lõigu ehk muudab värvi sellel lõigul. Siin toimub meil kõigepealt hulk kleepimisi, millest igaüks muudab värvi mingil lõigul. Meil on vaja kuidagi järge pidada nende muutuste üle, et saaksime väljastada lõppseisu. Selleks sobib suurepäraselt lõikude puu. Kuna väärtusi muudetakse tervete lõikude kaupa ja võib olla ka päris palju üle kleepimisi, siis on kaval kasutada laiska väärtustamist – konkreetsete ühiklõikude väärtused ei pruugigi kunagi kasutuses olla. Seetõttu pole vaja isegi luua tervet puud vaid võime luua mingi tipu t alampuud alles siis, kui neid on vaja. Ülevaatlikkuse mõttes ongi joonistel toodud ainult vajalikud tipud. Alustame ainult juurtipuga, mis on must (kood 0):

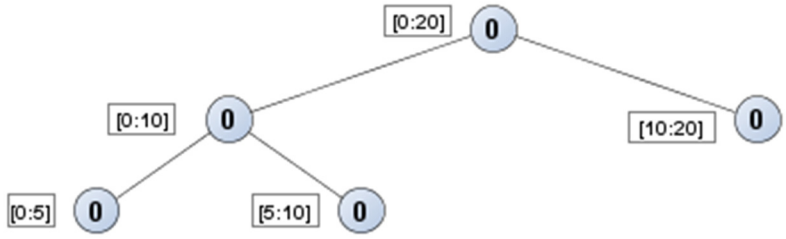


Nüüd on vaja lisada esimene riba värviga 1 (oletame, et see on punane). Kuna riba [2, 10] ei kata kogu musta riba, on vaja liikuda selle alampuudele. Alampuud pärivad esialgu sama värvi, mis vanemal:

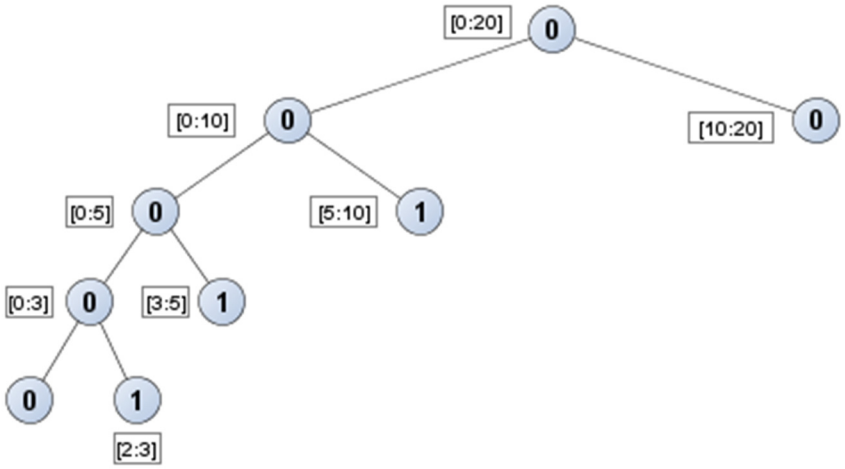


Vaatame kõigepealt, mis juhtub parempoolse alampuuga, mis vastab lõigule [10, 20]. Kuna see lõik jääb tervenisti kleebitavast ribast mõjutamata, siis jääb selle värviks must (0) ning sealt edasi liikuda

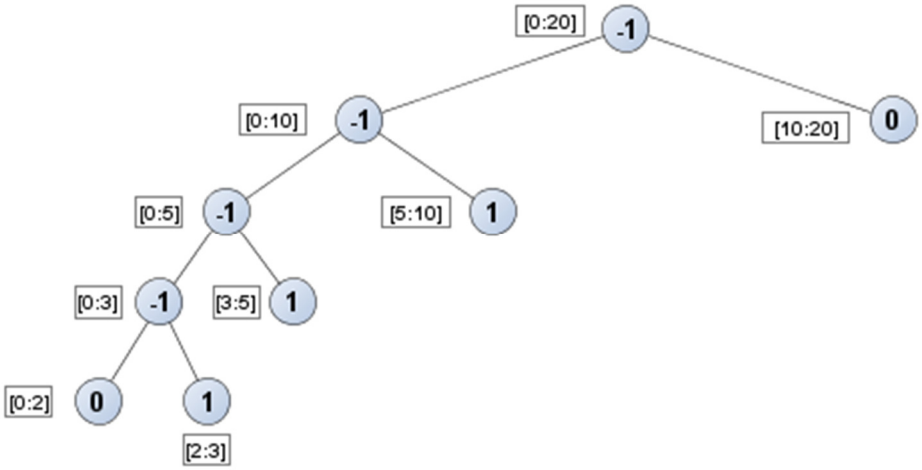
ei ole vaja. Vasakpoolsele alampuule vastav lõik [0, 10] on osaliselt kleebitava ribaga kaetud, seega tuleb liikuda selle alampuudele:



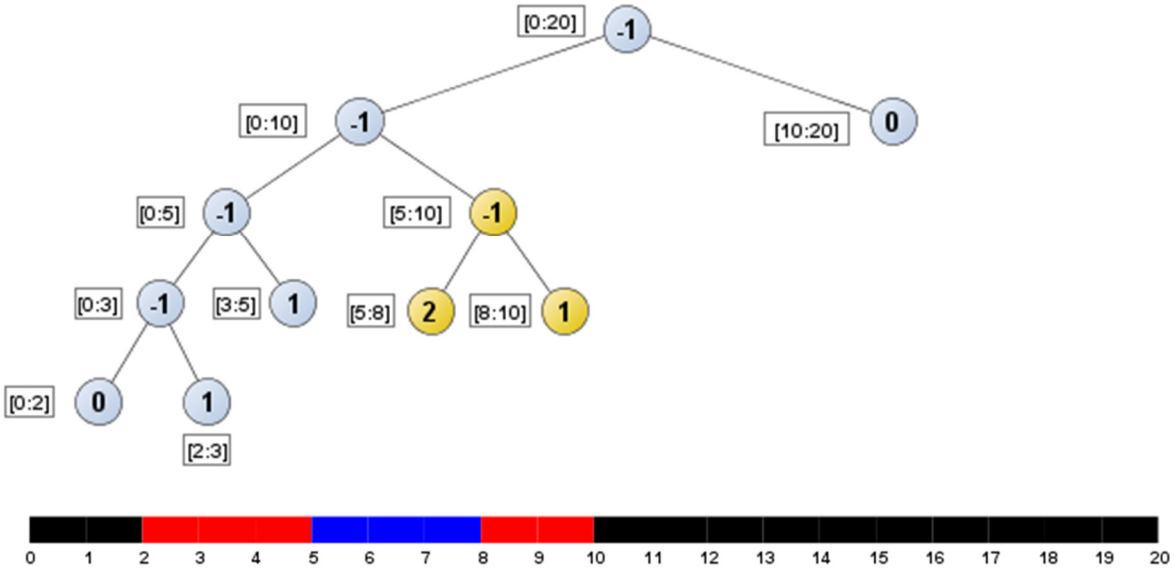
Vaatame taas enne lõigu [0, 10] parempoolset alampuud, mis vastab lõigule [5, 10]. See läheb tervenisti katmisele ja saamegi sellele määrata uue värvi ehk väärtuse 1. Siit edasi ei ole vaja liikuda. Vasakpoolne lõik [0, 5] aga on osaliselt kaetud ja seda tuleb taas poolitada ning pärast seda veel kord:



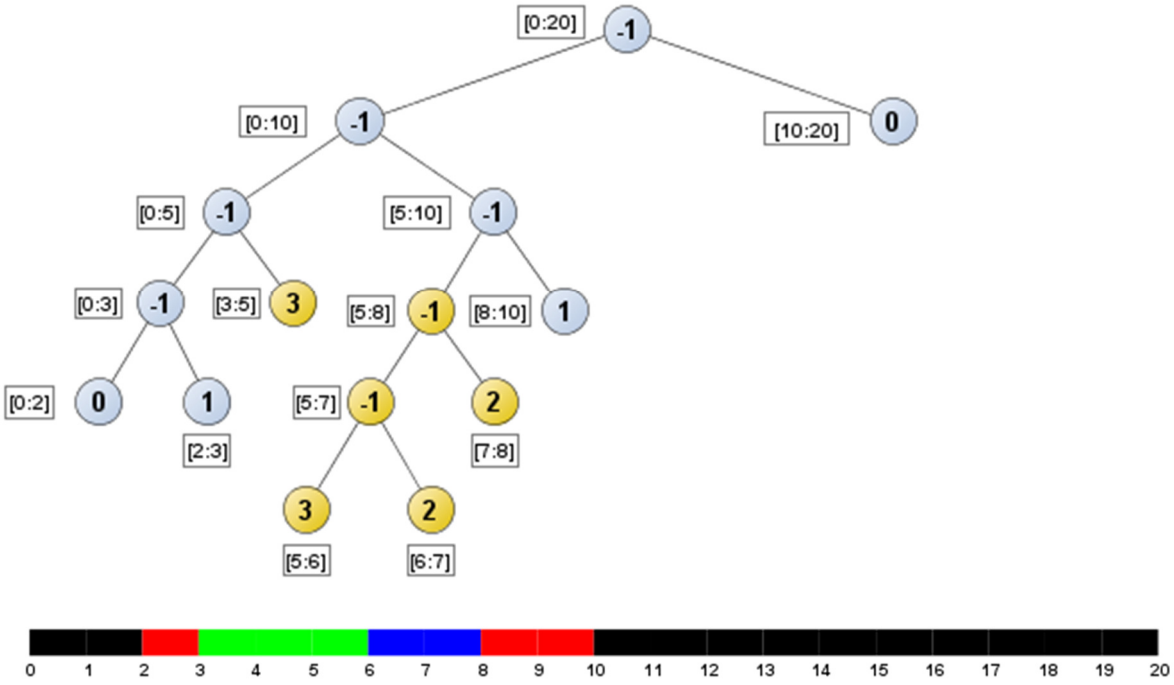
Nüüd on probleem selles, et mitmetel vanematel on värv erinev alluvate omast. Seega paneme selliste vanemate, millel on eri värvi alluvaid, värvi väärtuseks -1. Nii on puud läbi käies lihtne aru saada, mis toimub – kui lõigu värviks on -1, siis selle lõigu tegelikku värvi me ei tea ja peame uurima tema alamlõike. Kui aga lõigul on seatud mingi muu värv, siis on kogu see lõik justnimelt seda värvi ja edasi ei ole vaja seda lõiku jupiti uurida:



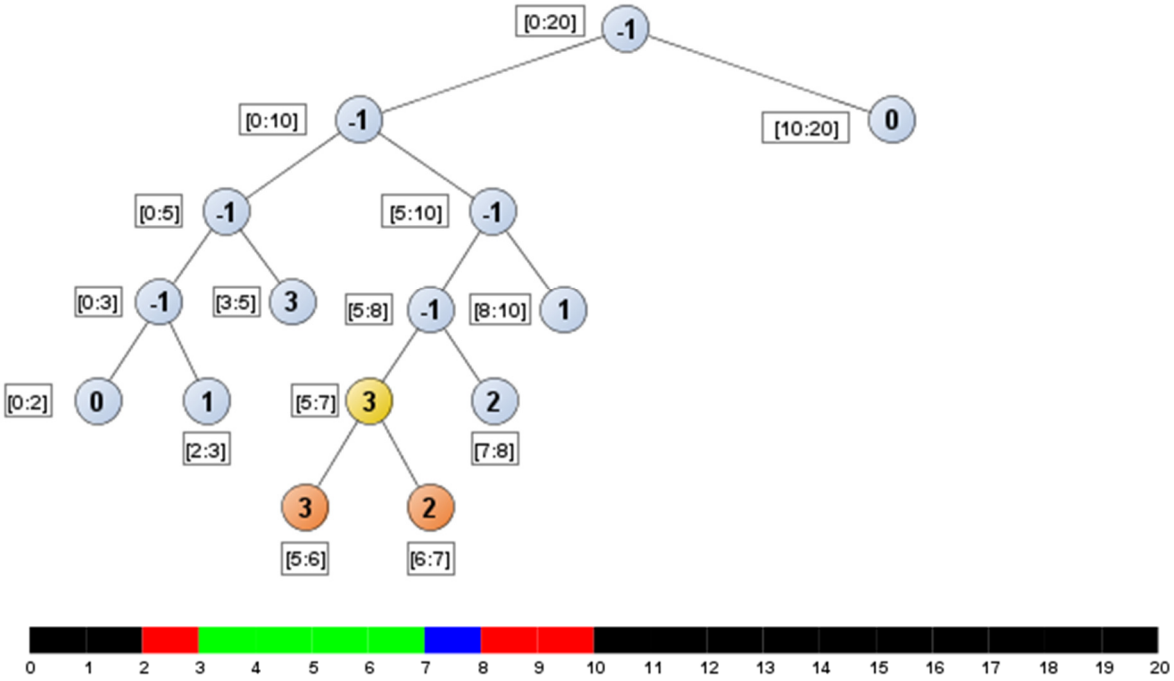
Teise riba (sinine) lisamise järgselt on pilt selline (kollased on muutunud väärtustega tipud):



Kolmas riba (roheline) annab seisu:



Pärast neljanda riba lisamist (samuti roheline):



Siin tekkis selline huvitavam olukord, kus sai üle värvida lõigu, mis oli enne mitmevärviline (värvi väärtus oli -1). See tipp on märgitud joonisel kollasega. Sellest jäävad ripakile tema alluvad (joonisel punased). Kas on vaja nendega midagi ette võtta? Esialgu puudub selleks vajadus, aga tuleb meeles pidada, et kui liigume ühevärviliselt tipult tema alluvatele, siis tuleb nende väärtust uuendada – nad pärivad vanemalt värvi väärtuse. Nii et kui see veel enne ilmne ei tundunud, siis nüüd on vast aru saada, et tegeleme laisa väärtustamisega.

Pane tähele, et kui puu kohe alguses valmis ehitada, tekib see probleem tegelikult juba esimese riba lisamisel, kuna aga joonisel neid alluvaid ei kujutatud, siis ei paistnud see veel välja.

9.2.2 Kood

```
#include <iostream>
#include <vector>
#include <set>
#include <algorithm>

using namespace std;
vector<int> _pointList;
vector<Ribbon> _result;
static TreeNode* MakeTree(int startIx, int endIx);
static void SetRibbon(TreeNode node, Ribbon ribbon);
static void WriteTree(TreeNode node);

class Ribbon
{
public:
    int Start;
    int End;
    int Color;
    void init(int start, int end, int color);
};
```

```
void Ribbon::init(int a, int l, int varv) {
    Start = a;
    End = l;
    Color = varv;
}

class TreeNode
{
public:
    int Color;

    int LeftPoint;
    int RightPoint;

    int LeftEnd;
    int RightEnd;

    bool CoversLeft;
    bool CoversRight;

    TreeNode* LeftNode;
    TreeNode* RightNode;
};

int main()
{
    set<int> pointSet;
    vector<Ribbon> ribbons;

    int l, n;
    cin >> l >> n;

    pointSet.insert(0);
    pointSet.insert(l);

    for (int i = 0; i < n; i++) {
        int varv, algus, lopp;
        cin >> varv >> algus >> lopp;
        Ribbon newRibbon;
        newRibbon.init(algus, lopp, varv);
        ribbons.push_back(newRibbon);
        pointSet.insert(newRibbon.Start);
        pointSet.insert(newRibbon.End);
    }

    _pointList.assign(pointSet.begin(), pointSet.end());
    sort(_pointList.begin(), _pointList.end());

    TreeNode* root = MakeTree(0, _pointList.size() - 1);

    for (int i = ribbons.size(); i > 0; --i) {
        Ribbon ribbon = ribbons[i];
        SetRibbon(root, ribbon);
    }

    WriteTree(root);

    for (int i = 0; i < _result.size(); i++) {
        Ribbon ribbon = _result[i];
        cout << ribbon.Color << " " << ribbon.End - ribbon.Start << endl;
    }
}
```

```
static void WriteTree(TreeNode node)
{
    if (node.LeftNode && !node.CoversLeft) {
        WriteTree(node.LeftNode);
    }

    Ribbon ribbon;
    ribbon.init(node.CoversLeft ? node.LeftEnd : node.LeftPoint,
        node.CoversRight ? node.RightEnd : node.RightPoint, node.Color);
    if (ribbon.Color == -1) {
        ribbon.Color = 0;
    }

    if (_result.size() > 0 && _result[_result.size() - 1].Color == ribbon.Color) {
        _result[_result.size() - 1].End = ribbon.End;
    }
    else {
        _result.push_back(ribbon);
    }

    if (node.RightNode && !node.CoversRight) {
        WriteTree(node.RightNode);
    }
}

static void SetRibbon(TreeNode node, Ribbon ribbon)
{
    // 0 == täiesti värvimata haru, -1 = osaliselt värvitud, värvimata juurtipuga haru
    if (node.Color <= 0) {
        if (node.Color==0&&ribbon.Start<=node.LeftEnd&&ribbon.End>=node.RightPoint) {
            node.CoversLeft = true;
        }
        if (node.Color==0&&ribbon.Start<=node.LeftPoint&&ribbon.End>=node.RightEnd) {
            node.CoversRight = true;
        }
        if (ribbon.Start <= node.LeftPoint && ribbon.End >= node.RightPoint) {
            node.Color = ribbon.Color;
        }
        else {
            node.Color = -1;
        }
    }

    if (ribbon.Start < node.LeftPoint && !node.CoversLeft && node.LeftNode) {
        SetRibbon(node.LeftNode, ribbon);
    }
    if (ribbon.End > node.RightPoint && !node.CoversRight && node.RightNode) {
        SetRibbon(node.RightNode, ribbon);
    }
}
```

```
static TreeNode* MakeTree(int startIx, int endIx)
{
    int medIx = (startIx + endIx) / 2;
    TreeNode node;
    node.LeftEnd = _pointList[startIx];
    node.RightEnd = _pointList[endIx];
    node.LeftPoint = _pointList[medIx];
    node.RightPoint = _pointList[medIx + 1];
    if (startIx < medIx) {
        node.LeftNode = MakeTree(startIx, medIx);
    }
    if (endIx > medIx + 1) {
        node.RightNode = MakeTree(medIx + 1, endIx);
    }
    return &node;
}
```

9.2.3 Ülesanne: Sein

See ülesanne on Taipeis toimunud 2014. aasta rahvusvaheliselt informaatikaolümpiaadilt.

Jian-Jia ehitab seina, ladudes üksteise otsa ühesuuruseid kive. Sein koosneb tulbast kividest, mis on nummerdatud vasakult paremale. Tulpadel kõrgused võivad olla erinevad. Tulba kõrguseks on kivide arv selles tulbas.
Jian-Jia ehitab seina järgneval viisil: alguses pole üheski tulbas ühtki kivi. Seejärel läbib Jian-Jia K kivide *lisamise* või *eemaldamise* etappi. Ehitamine lõpeb, kui kõik K etappi on läbitud. Igas etapis saab Jian-Jia lõigu järjestikuseid kivitulpasid ja kõrguse H ning sooritab järgmise protseduuri:

- *Lisamise* etapis lisab Jian-Jia etteantud lõigus kive nendesse tulpadesse, kus on vähem kui H kivi, nii et neis oleks täpselt H kivi. Tulpadega, kus juba on H või rohkem kivi, ei tee ta midagi.
- *Eemaldamise* etapis eemaldab Jian-Jia etteantud lõigus kive nendest tulpadest, kus on rohkem kui H kivi, nii et alles jääb H kivi. Tulpadega, kus juba on H või vähem kivi, ei tee ta midagi.

Sinu ülesanne on leida seina lõplik kuju.
Sisendi esimesel real on kaks täisarvu: n ($0 \le n \le 2\ 000\ 000$) – tulpade arv seinas ja k ($0 \le k \le 500\ 000$) – etappide arv. Järgmisel k real on igaühel neli täisarvu, mis kirjeldavad igat etappi: operatsiooni tüüp t (1-lisamine, 2-eemaldamine), vasak otspunkt v, parem otspunkt p, kusjuures $v \le p$, ja h ($0 \le h \le 100\ 000$) – seina kõrgus sellel etapil.
Väljundi ainsale reale kirjutada tühikutega eraldatult n täisarvu – iga tulba lõppkõrgus.
***NÄIDE*:**

```
10 6
1 1 8 4
2 4 9 1
2 3 6 5
1 0 5 3
1 2 2 5
2 6 7 0
```

Vastus:

```
3 4 5 4 3 3 0 0 1 0
```

Sellegi ülesande lahendamiseks saab kasutada lõikude puud. Siin aga tuleb väga hoolega silmas pidada, milline on ehitamise mehhanism: me ei pruugi muuta lõigul kõiki väärtusi, vaid ainult teatud osa nendest. Suurt probleemi see kohe ei tekita, sest lisamisel peame võtma maksimumi senisest kõrgusest ja uuest kõrgusest, eemaldamisel aga neist miinimumi.

Ühe võimalusena on hoida lisamise ja eemaldamise protseduuride jaoks eraldi massiive.

```
#include <algorithm>
#include <iostream>

using namespace std;

const int LEHED = (1 << 21);
const int PIKKUS = (1 << 22) + 5;

int D[PIKKUS], U[PIKKUS];
int N, K;

void yhenda(int n, int d, int u)
{
    D[n] = min(D[n], d);
    D[n] = max(D[n], u);
    U[n] = max(U[n], u);
    U[n] = min(U[n], d);
}

void tootle(int n, char t, int h)
{
    if (t == 2) {
        D[n] = min(D[n], h);
        U[n] = min(U[n], h);
    }
    if (t == 1) {
        D[n] = max(D[n], h);
        U[n] = max(U[n], h);
    }
}

void ehita(int v, int p, int t, int h, int n, int a, int b)
{
    if (a > p || b < v) return; // ei ole lõigus
    if (a >= v && b <= p) { // on tervenisti lõigus
        tootle(n, t, h);
        return;
    }

    yhenda(2 * n, D[n], U[n]);
    yhenda(2 * n + 1, D[n], U[n]);
    D[n] = PIKKUS;
    U[n] = 0;

    int mid = (a + b) / 2;
    ehita(v, p, t, h, 2 * n, a, mid);
    ehita(v, p, t, h, 2 * n + 1, mid + 1, b);
}

void lopeta()
{
    for(int i = 1; i <= LEHED - 1; i++) {
        yhenda(2 * i, D[i], U[i]);
        yhenda(2 * i + 1, D[i], U[i]);
    }
}
```

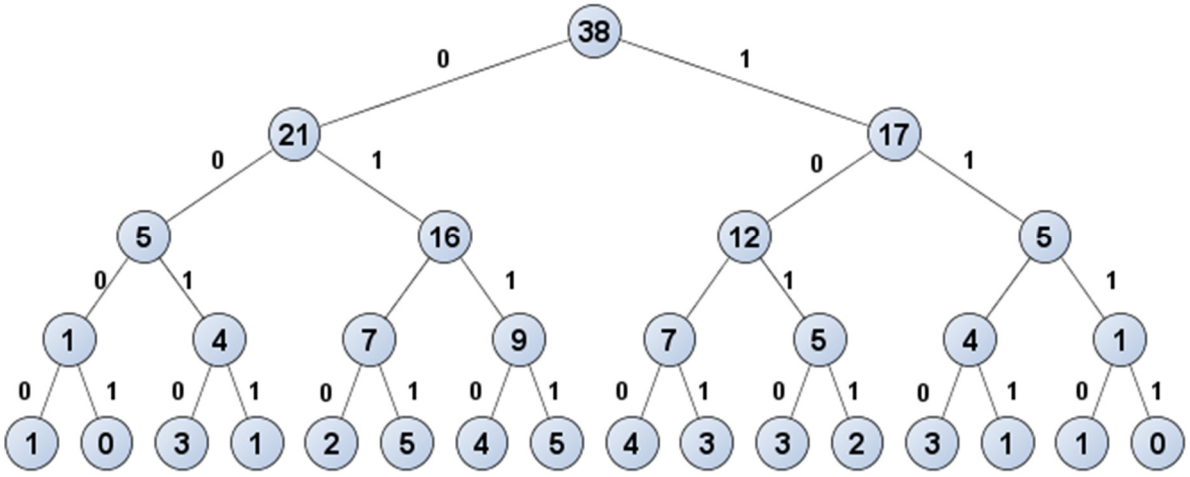
```
int main()
{
    cin >> N >> K;

    for (int i = 1; i <= K; i++) {
        int t, v, p, h;
        cin >> t >> v >> p >> h;
        //1-lisamine, 2-eemaldamine
        ehita(v + 1, p + 1, t, h, 1, 1, LEHED);
    }

    lopeta();
    for (int i = LEHED; i < LEHED + N; i++)
        cout << min(D[i], U[i]) << " ";
    cin >> K;
}
```

9.3 FENWICKI PUU

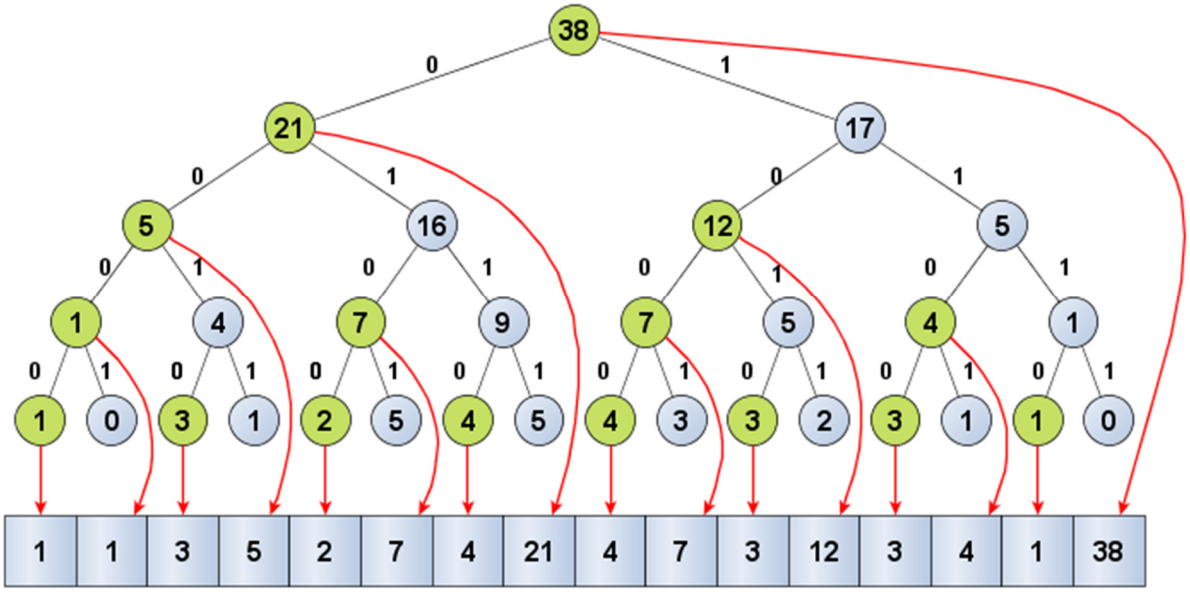
Fenwicki puu ehk **kahendindekseeritud puu** leiutas 1994. aastal Uus-Meremaa arvutiteadlane Peter Fenwick, kes uuris võimalusi andmepakkimisalgoritmide efektiivsuse suurendamiseks.

Vaatame ülesannet, kus tuleb leida kumulatiivne summa mingil lõigul. Jällegi – kui seda on vaja teha ainult üks kord, siis piisab sellest, kui liidamegi väärtused antud lõigul kokku. Kui päringuid on palju, kuid väärtused ei muutu, siis võib jadas hoida kumulatiivseid summasid: jada esimesel kohal on esimese elemendi väärtus, teisel kohal esimese ja teise summa, kolmandal esimese kolme summa jne. Kui on vaja leida summa lõigul $[a, b]$, siis selleks on lõikude $[0, b]$ ja $[0, a-1]$ vahe. Seega on sellise massiivi ülesseadmine keerukusega $O(n)$ ning sellest summa leidmine keerukusega $O(1)$.

Kolmandaks võimaluseks on kasutada lõikude puud, mille tippudes hoitakse alluvate summasid. Näiteks:



Lehtedeks on kõikide arvude 0-15 (0000-1111) sagedused. Igas tipus on tema alluvate summa. Paneme tähele, et summa leidmiseks on vaja vaadata ainult mõningaid tippe:



Kumulatiivsete summade leidmiseks on vaja hoida meeles ainult roheliste tippude väärtused.

9.3.1 Summa leidmine

Vaatame, kuidas leida esimese 13 elemendi summat. Iga summat saab esitada mingi mittekattuvate alamlõikude summana.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	0	3	1	2	5	4	5	4	3	3	2	3	1	1	0

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	1	3	5	2	7	4	21	4	7	3	12	3	4	1	38

Ülemisel joonisel on algses massiivis summeeritavad väärtused. Erinevate värvidega on tähistatud erinevad lõigud. Teises massiivis on Fenwicki puu ja rohelisega on tähistatud esimese 8 elemendi summa, kollasega 8.-12. elementide summa ja punasega 13. element. 13 esimese elemendi kumulatiivne summa on $21 + 12 + 3 = 36$.

Näiteks kui on vaja leida 13 esimese elemendi summat, siis kuna $13 = 2^0 + 2^2 + 2^3$, siis on meil vaja leida `FP[8] + FP[8+4] + FP[8+4+1]`, sest `FP[8]` katab lõigu 1…8, `FP[12]` lõigu 9..12 ja `FP[13]` üheelemendilise lõigu ehk elemendi kohal 13.

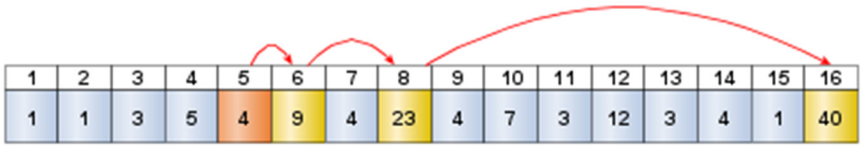
Järgnev funktsioon leiab esimese i elemendi summa:

```
int KumSumma(int i) {
    int sum = 0; // siin hoiame otsitavat summat
    while (i > 0) {
        sum += FP[i]; // Fenwicki puu
        i = i - (i & -i); // lahutame viimase biti
    }
    return sum;
}
```

Siin on kasutatud veidi kavalust bittidega. Negatiivsete arvude esitusest tulenevalt annab `i & -i` tulemuseks viimase seatud biti arvus. Näiteks 7 & -7 = 1, 6 & -6 = 2, 8 & -8 = 8 jne. `i - (i & -i)` nullib viimase seatud biti.

9.3.2 Elemendi lisamine ja muutmine

Iga kord, kui mingi elemendi väärtust muudetakse, tuleb muuta ka vastavad summad. Muutmise algoritm on sarnane kumulatiivse summa leidmise omale, aga liigume siin vastupidises suunas – muutmisel tuleb muuta järgnevaid elemente, mida muudatus mõjutab. Näiteks kui suurendada 5. elemendi väärtust 2 võrra, siis suurendame kõigepealt elementi FP[5], seejärel ka FP[6], kuna selles hoitakse FP[4] ja FP[5] summat. Edasi tuleb suurendada FP[8], kus hoitakse esimese kaheksa elemendi summat ja loomulikult ka FP[16], kus on kogu jada summa.



Kasutades taas viimase biti leidmist, saame kiirelt arvutada, millised väärtused tuleb muuta:

```
void lisaVaartus(int i, int v) {
    while (i <= N) {
        FP[i] += v;
        i = i + (i & -i);
    }
}
```

Lisamist võib ka vaadata kui muutmise operatsiooni, seega saab nii lihtsa vaevaga Fenwicki puu ehitada.

9.3.3 Üksiku elemendi leidmine

Kui mäluruumi jagub, siis võib muidugi iga elemendi tegeliku väärtuse hoida harilikus massiivis ja üksiku elemendi väärtuse leidmine on sel juhul keerukusega $O(1)$. Üksiku elemendi väärtust on võimalik aga taastada ka Fenwicki puust. Pole raske märgata, et $A[k] = FP[k] - FP[k-1]$, kus A on algandmetega massiiv.

```
int leiaVaartus(int i) {
    return KumSumma(i) - KumSumma(i - 1);
}
```

Vajadusel saab veidi kavalamalt ka, kasutades ära teadmist, et FP[i] väärtuseks on kas i. elemendi väärtus või i. elemendi ja mingite eelnevate elementide summa:

```
int leiaVaartus2(int i) {
    int sum = FP[i]; // summat hakkame vähendama (vajadusel)
    if (i > 0) { // erijuht kui i=0
        int z = i - (i & -i); // lahutame viimase biti maha
        i--;
        while (i != z) { // mingil hetkel z saab võrdseks indeksiga
            sum -= FP[i]; //vähendame summat
            i -= (i & -i); //lahutame viimase biti
        }
    }
    return sum;
}
```

9.3.4 Indeksi leidmine kumulatiivse summa järgi

Sageli on vaja ka leida etteantud kumulatiivsele summale vastavat indeksit. Üheks võimaluseks on proovida lihtsalt iga indeksi jaoks arvutada kumulatiivne summa ja võrrelda seda soovitud summaga. Kui väärtused võivad olla ka negatiivsed, siis see on tegelikult ainus moodus. Kui aga väärtused on kõik positiivsed, saab kasutada järgmist kahendotsimise algoritmi:

```
int otsiSumma(int summa) {
    int i = 0; // vastus
    int bitMask = esimeneBitt(N);
    while ((bitMask != 0) && (i < N)) {
        int mid = i + bitMask; // keskpunkt lõigul
        if (summa == FP[mid])
            return mid; // leidsime õige summa
        else if (summa > FP[mid]) { // kui mahub otsitavasse summasse
            i = mid;
            summa -= FP[mid]; // otsime, mis veel mahub
        }
        bitMask >>= 1; // poolitame otsimislõigu
    }
    if (summa != 0)
        return -1; // otsitavat summat ei olegi
    else
        return i;
}
```

9.3.5 Teatud lõigu summa leidmine

Kui on vaja leida summa teatud lõigul, näiteks elemendist 2 kuni elemendini 7, siis kõige lihtsamaks lahenduseks on

```
int loiguSumma(int i, int j) {
    return KumSumma(j) - KumSumma(i - 1);
}
```

9.4 2-MÕÕTMELINE FENWICKI PUU

Vaatame järgmiseks ülesannet, kus on vaja punkte tasandile lisada ja eemaldada ning leida parasjagu mingisse ristkülikukujulisse alasse jäävate punktide arv.

Selle lahendamiseks saab mugavalt kasutada kahemõõtmelist Fenwicki puud, mis sisuliselt on Fenwicki puu, mille tippudeks on omakorda Fenwicki puud.

9.4.1 Lisamine

Lisamise protseduur näeb välja järgmine:

```
void lisa(int x, int y, int val) {
    int y1;
    while (x <= max_x) {
        y1 = y;
        while (y1 <= max_y) {
            FP2D[x][y1] += val;
            y1 += (y1 & -y1);
        }
        x += (x & -x);
    }
}
```

Antud ülesandes võib siis kasutada uue punkti lisamisel väärtust 1 ja olemasoleva punkti eemaldamisel väärtust -1.

9.4.2 Loendamine

Analoogselt saab laiendada (kumulatiivse summa) leidmise algoritmi kahemõõtmelisele puule:

```
int loenda(int i, int j) {
    int j1;
    int sum = 0;
    while (i > 0) {
        j1 = j;
        while (j1 > 0) {
            sum += FP2D[i][j1];
            j1 = j1 - (j1 & -j1);
        }
        i = i - (i & -i);
    }
    return sum;
}
```

Kui on vaja loendada punkte suvalisest ristkülikust:

```
int loenda(int x1, int y1, int x2, int y2) {
    return loenda(x2, y2) - loenda(x1-1, y2) - loenda(y1-1, x2) + loenda(x1-1, y1-1);
}
```

Sama põhimõtte järgi saab koostada ka n-mõõtmelisi Fenwicki puid.

9.5 AJALOOGA ANDMESTRUKTUURID

Vahel on meil vaja andmestruktuuri, mis lisaks oma tavalisele funktsionaalsusele „mäletab“ ka neid andmeid, mis selles varem olid. Sel juhul ütleme, et andmestruktuur on **ajalooga** (i.k. *persistent data structure*). Andmestruktuur võib olla **täieliku** ajalooga (kui kõiki selle varasemaid versioone on võimalik muuta) või siis **osalise** ajalooga (kui muuta saab ainult viimast seisu).

Lihtsaim meetod andmestruktuuri ajaloo hoidmiseks on mõistagi sellest iga muudatuse puhul koopia tegemine, kuid see on jõudluse mõttes raiskav. Oluliselt parem on muutumatuks jäävaid andmeid versioonide vahel jagada.

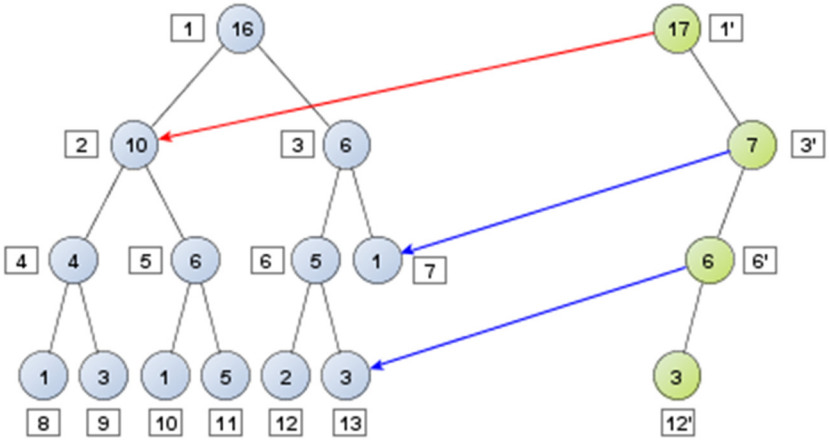
Üks võimalus selleks on nn **paksu elemendi** meetod – iga üksikelemendi asemel hoiame andmestruktuuris isekasvavat massiivi (nt vectorit või listi). Kui vastavat väärtust muudetakse, lisatakse massiivi järgmine element. Paksu elemendi meetod sobib näiteks ajalooga massiivi loomiseks.

Teine oluline võimalus on **teede kopeerimine**. Seda saab kasutada juhul, kui struktuur on ehitatud üksteisele viitavatest elementidest (nt ahel või puu). Selle meetodi korral:

1. Muuda ära viited muutunud elemendile.
2. Muuda ära viited eelmises sammus muutunud elementidele.
3. Korda sammu 2 kuni struktuuri alguseni.

9.5.1 Ajalooga lõikude puu

Ajalooga lõikude puu (i.k. *persistent segment tree*) tähendab seda, et säilitame kõik tehtud muudatused, kasutades teede kopeerimise meetodit. Kõige lihtsam on sellest mõelda kui versioneerimisest – iga muudatuse kohta tekib uus puu. Samas on terve uue puu loomine on nii mälu- kui ka ajamahukas. Teede kopeerimise meetod võimaldab aga teha muudatusi nii, et uuendatakse maksimaalselt $log(n)$ tippu. Põhimõtteliselt hoitakse meeles vaid see muudatuste puu – ülejäänud tipud on samad, mis eelmises verisoonis.



Ajalooga lõikude puu. Vasakul on eelmine versioon (hallid tipud) ja paremal uus versioon (rohelised tipud). Punane nool tähistab vasakut alluvat ja sinine paremat alluvat juhul, kui alluv on eelmises versioonipuus (ei ole muutunud).

Ülaltoodud joonisel on tippudes alluvate väärtuste summad. Elemendi asukohas 12 väärtust suurendatakse ühe võrra. Selle asemel, et see väärtus üle kirjutada, tekitatakse uus tipp 12'. Kuna see muudatus muudab ka tippude 6, 3 ja 1 väärtusi, tuleb nende jaoks luua samuti uued tipud 6', 3' ja 1'. Tipu 6' vasakuks alluvaks on tipp 12', kuid vasakuks alluvaks tipp 13. Samamoodi käitub ka tipp 3'. Juurtipu 1' vasakuks alluvaks saab tipp 2, paremaks 3'.

Alustades päringut tipust 1, saame seisu, mis oli enne muudatust, alustades aga tipust 1', läbime muudetud puud.

Peamiselt kerkib üles küsimus, kuidas pidada arvet versioonide üle. Piisab, kui teada versioonipuude juuri – neid võib meeles pidada eraldi massiivis. Pane tähele, et iga muudatus puus kandub edasi juureni, seega on igal versioonipuul oma versioon algse puu juurtipust.

Vaatame järgmise ülesande näitel, kuidas ajalooga lõikude puud rakendada. Ülesanne on pärit Balti Informaatikaolümpiaadilt (BOI) aastast 2015.

9.5.2 Ülesanne: Tekstiredaktor

Programmeerija Baitasar töötab uue revolutsioonilise tekstiredaktori kallal. Redaktoril on kaht tüüpi operatsioone: üks võimaldab redaktoris teksti *redigeerida*, teine võimaldab eelnevat operatsiooni *tagasi võtta* (undo). Üks redaktori innovatiivsetest võimalustest on *mitmetasemeline tagasivõtt*. See töötab järgmiselt: me ütleme, et teksti redigeerimine on *0 taseme* operatsioon. *i taseme tagasivõtuoperatsioon* (kus `i = 1, 2,...`) võtab tagasi viimase ülimalt `i−1` taseme operatsiooni, mis pole veel tagasi võetud. Näiteks 1. taseme tagasivõtt saab tagasi võtta ainult redigeerimis-operatsioone, aga 2. taseme tagasivõtt saab tagasi võtta nii redigeerimisoperatsioone kui ka esimese taseme tagasivõtte (aga mitte kõrgemate tasemete tagasivõtte).

Formaalsemalt saab iga juba sooritatud operatsioon olla kahes olekus: `aktiivne` või `tagasivõetud`. Olgu `X` üks neist operatsioonidest. Kohe pärast operatsiooni `X` sooritamist on ta olekus `aktiivne`. Kui `X` on `i.` taseme tagasivõtuoperatsioon, siis leiame viimase ülimalt `i−1` taseme (tähistame selle `X1`) operatsiooni seisus `aktiivne` ning muudame operatsiooni `X1` seisu `tagasivõetud`. Kui `X1` on samuti tagasivõtuoperatsioon, siis tuleb meil ka see operatsioon, mille `X1` tagasi võttis (tähistame selle `X2`), muuta seisu `aktiivne`. Jätkame samal viisil: kui tagasivõtuoperatsiooni X_j olek, mis oli enne tagasi võtnud operatsiooni X_j+1, muutub, siis peame muutma ka operatsiooni X_j+1 olekut (mis võib omakorda muuta järgmiste operatsioonide olekut). Kogu muutmiste jada lõpeb, kui jõutakse redigeerimisoperatsioonini.

Lihtsuse mõttes on tekstiredaktori hetkeseis antud üksiku täisarvuga `s`, mida nimetatakse redaktori olekuks (alguses väärtusega `0`). Iga redigeerimisoperatsioon annab meile uue redaktori oleku. Redaktori olek sõltub viimasest redigeerimisoperatsioonist, mille olek on `aktiivne`. Aita Baitasaril kirjutada programm, mis jälgib tekstiredaktori olekut.

Sisendi esimesel real on positiivne täisarv `n`, mis tähistab Baitasari poolt sooritatavate operatsioonide arvu. Järgmised `n` rida sisaldavad operatsioone, üks igal real. Iga operatsioon on täisarv a_i ($-n \le a_i \le n$, $a_i \neq 0$). Kui $a_i > 0$, siis see tähistab redigeerimisoperatsiooni, mis viib redaktori olekusse a_i. Kui $a_i < 0$, siis see tähistab tagasivõtuoperatsiooni tasemega $-a_i$. Võib eeldada, et iga tagasivõtuoperatsiooni jaoks on alati mõni eelnev madalama tasemega operatsioon, mille seis on aktiivne, mida saab tagasi võtta.

Väljasta `n` rida. Rida `i` peaks sisaldama ühe täisarvu, millel on redaktori seis pärast esimese `i` operatsiooni sooritamist.

NÄIDE:

```
11
1
2
5
-1
-1
-3
4
-2
-1
-1
1
```

Vastus:

```
1
2
5
2
1
2
4
2
1
0
1
```

Vaatame järgmist näidet: järgnev tabel näitab mõnesid operatsioone, mida Baitasar saab sooritada ning redaktori olekut pärast nende sooritamist. Sümbol E_s tähistab redigeerimisopratsiooni, mis viib redaktori olekusse s ja sümbol U_i tähistab i. taseme tagasivõtuoperatsiooni:

	1	2	3	4	5	6	7	8	9	10	11
Operatsioon	E_1	E_2	E_5	U_1	U_1	U_3	E_4	U_2	U_1	U_1	E_1
Olek pärast	1	2	5	2	1	2	4	2	1	0	1

Alguses sooritas Baitasar kolm redigeerimisoperatsiooni. Redaktor läks olekust 0 olekusse 1, siis 2 ja lõpuks 5. Järgmiseks sooritas ta kaks 1. taseme tagasivõtuoperatsiooni, mis võtsid tagasi operatsioonid E_5 ja E_2 (muutes nad olekusse tagasivõetud). Sellega oli redaktori seis läinud tagasi olekusse 1. Järgnev 3. taseme tagasivõtuoperatsioon võttis tagasi viimase U_1 operatsiooni (muutes selle olekusse tagasivõetud). Tagajärjena taastatakse operatsioon E_2 (see läheb olekusse aktiivne) ning redaktor läheb uuesti seisu 2. Operatsioon U_2 võtab tagasi operatsiooni E_4, operatsioon U_1 võtab jälle tagasi taastatud operatsiooni E_2, siis võtab viimane U_1 operatsioon tagasi operatsiooni E_1. Viimane operatsioon on E_1.

9.5.3 Kood

```
#include <cstdio>

int max(int x, int y) {
    if (x > y)
        return x;
    else
        return y;
}
int max(int x, int y, int z) {
    return max(x, max(y, z));
}

struct PersTree {
    PersTree *l, *r;
    int key;
    int value;
    int max_value;

    int find(int) const;
    PersTree *add(int, int) const;

    PersTree(PersTree *_l, PersTree *_r, int _key, int _value, int _max_value) :
l(_l), r(_r), key(_key), value(_value), max_value(_max_value) {}

};

int safe_max_value(PersTree *x) {
    if (x == NULL)
        return 0;
    return x->max_value;
}

int PersTree::find(int mxkey) const {
    if (mxkey < key)
        return l->find(mxkey);
    else if (mxkey == key)
        return max(safe_max_value(l), value);
    else // if (mxkey > key)
        return max(safe_max_value(l), value, r->find(mxkey));
}

PersTree *PersTree::add(int _key, int nvalue) const {
    PersTree *_l = l;
    PersTree *_r = r;
    int _value = value;

    if (_key < key)
        _l = l->add(_key, nvalue);
    else if (_key == key)
        _value = nvalue;
    else // if (_key > key)
        _r = r->add(_key, nvalue);

    return new PersTree(_l, _r, key, _value, max(safe_max_value(_l), _value,
safe_max_value(_r)));
}
```

```
PersTree *create_pers_tree(int left, int right) {
    if (left > right)
        return NULL;
    int s = (left + right) / 2;
    return new PersTree(create_pers_tree(left, s - 1), create_pers_tree(s + 1, right),
s, 0, 0);
}

int n;
const int M = 3e5 + 5;
PersTree *akt[M];
int wyn[M];
int main() {
    scanf("%d", &n);
    akt[0] = create_pers_tree(0, n);
    wyn[0] = 0;

    for (int i = 1; i <= n; ++i) {
        int op;
        scanf("%d", &op);
        if (op > 0) {
            wyn[i] = op;
            akt[i] = akt[i - 1]->add(0, i);
        }
        else {
            int cof = akt[i - 1]->find(-op - 1) - 1;
            wyn[i] = wyn[cof];
            akt[i] = akt[cof]->add(-op, i);
        }
        printf("%d\n", wyn[i]);
    }
}
```

9.6 HÕRE TABEL

Kui andmed ei muutu, siis on võimalik lõikude puu asemel kasutada „hõredat tabelit“ (i.k. *sparse table*). Tabeli veergudeks on lõigu alguspunktid ning tabeli veergudeks lõikude pikkuseid, nii et reas indeksiga j on lõikudele pikkusega 2^j vastavad väärtused. Esimeses reas, mille indeksiks on 0, on seega üheelemendise pikkusega lõigud ehk töödeldava jada algväärtused.

Siin on tabel, mis vastab kumulatiivse summa ülesandele:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	0	3	1	2	5	4	5	4	3	3	2	3	1	1	0
1	1	3	4	3	7	9	9	9	7	6	5	5	4	2	1	
2	5	6	11	12	16	18	16	15	12	11	9	7	5			
3	21	24	27	27	28	29	25	22	17							
4	38															

9.6.1 Tabeli koostamine

Tabeli koostamise üldreegel:

$$T[i,j] = F_{k=i\ldots i+2^j-1}\{Jada[k]\}, kus\ 1 \le i \le n, 1 \le j \le \lfloor \log n \rfloor$$

Kiiremaks koostamiseks saab kasutada dünaamilist programmeerimist. Näiteks minimaalse elemendi leidmise korral on rekursioonivalem järgmine:

$$T[i,j] = F(T[i,j-1], T\left[i+2^{j-1}, j-1\right]$$

Tabeli ehitamiseks kuluv aeg on $O(n\ log\ n)$.

9.6.2 Tabelist lugemine

Tabelist lugemine sõltub mõnevõrra ülesande spetsiifikast.

Näiteks minimaalse väärtuse leidmiseks lõigul [i, j] on vaja leida kaks lõiku, mis tervenisti katavad otsitava lõigu [i, j]. Pikim tabeli lõik, mis mahub lõiku [i, j] on pikkusega $2^{\lfloor \log(j-i+1) \rfloor}$.

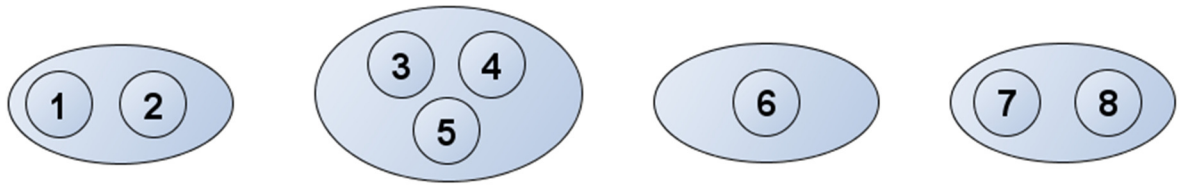
$$RMQ[i,j] = \begin{cases} T[i,k], & kui\ Jada[T[i,k]] \leq Jada[T[j-2^k+1,k]] \\ T[T[j-2^k+1,k] \end{cases}$$

Samas näiteks kumulatiivse summa ülesandes on oluline, et arvestatavad lõigud ei kattuks, vaid järgneksid vahetult üksteisele.

9.7 ÜHISOSATA HULGAD

Hulk (i.k *set*) on andmestruktuur, mille kõige olulisemaks tunnuseks on, et iga element saab esineda selles maksimaalselt ühe korra.

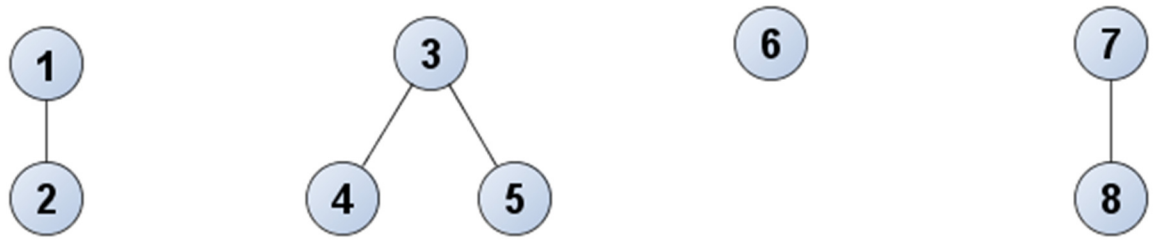
Mõnikord on aga vaja tervet hulkade komplekti nii, et iga element esineb mitte rohkem kui ühes hulgas. Sellist struktuuri võib nimetada ühisosata hulkadeks (i. k. *disjoint sets*).



Peamised operatsioonid, mida on vaja sooritada, on hulga loomine, kahe hulga ühendamine ning leidmine, millisesse hulka element kuulub.

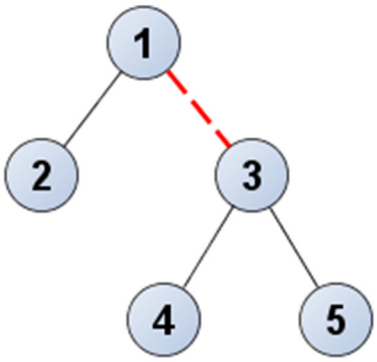
Abstraktselt on tegemist küllaltki lihtsa struktuuriga, küsimus on, et kuidas hoida hulki ja nende elemente nii, et mainitud operatsioonid oleksid võimalikult efektiivsed.

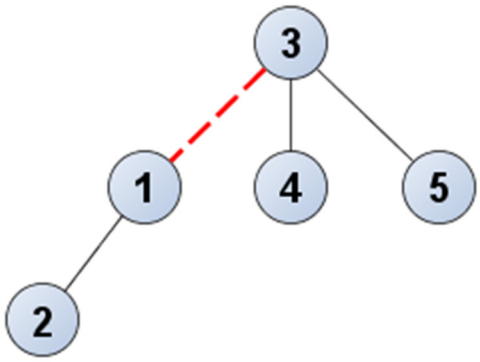
Tavaliselt esitatakse iga hulk elementide puuna ning kogu ühisosata hulkade struktuur on seega mets.



Peamiselt on meil vaja teada, kes on kelle vanem, seega võib kogu struktuuri hoida loendis, kus kohal i on i-nda elemendi vanem. Juurelemendi vanemaks loetakse enamasti element ise.

Kahe puu ühendamiseks piisab, kui muuta ühe puu vanemat:





Elemendi kuuluvuse leidmine on samuti lihtne:

```
int leia(int x) {
      if (hulgad[x] == x) return x;
      return leia(hulgad[x]);
}
```

Probleem on see, et halvimal juhul on selle keerukuseks $O(n)$ – seda küll juhul, kui kogu mets koosneb ühest puust sügavusega n ja soovime leida elemendi n hulka. Seetõttu on kasulik hoida puud võimalikult madalad.

Elemendi lisamisel on seda kerge teha, kuna meil ei ole piiratud, mitu alluvat tipul olla võib, ja seega saab uue elemendi lisada alati otse juure alla. Puu kõrgus kasvab seega just puude ühendamise operatsiooni käigus. Siingi on esmaseks lihtsaks lahenduseks, et madalam puu läheb alati kõrgema alla, mitte vastupidi. Nii jääb ka ühendatud puu kõrgus muutumatuks. Muutus kõrguses toimub vaid siis, kui ühendada tuleb kaks sama kõrgusega puud.

Puu kõrgust iga kord leida on tülikas, selle asemel peetakse seda meeles. Selleks antakse igale tipule veel üks väärtus – **aste** (i.k *rank*). Kahe puu ühendamisel lisame väiksema astmega puu suurema astmega puu alla ja nii jääb uue puu aste muutmatuks. Kui aga ühendatavad puud on võrdse kõrgusega, siis ühendame puud ja suurendame juure astet ühe võrra. Nii ei saa puu aste kasvada kunagi suuremaks kui log n.

Puus, mille juure aste on n, on vähemalt 2^n tippu. Näitame seda induktsiooniga. Baasjuhuks on ühetipuline puu, mille asteks on 0: $2^0 = 1$. Puu aste muutub ühe võrra, kui liidame kaks sama astmega puud. Olgu selleks astmeks k ja hüpoteesi järgi on siis kummaski puus vähemalt 2^k tippu. Tippude arv uues puus on siis vähemalt $2^k + 2^k = 2^{k+1}$.

Teiseks võtteks, et veelgi kiirendada elementide leidmiseks on nn **tee pakkimine** (i.k. *path compression*). Nagu juba lisamisel sai mainitud, siis parim juht on, kui puu on võimalikult madal ehk kõik hulga liikmed on puus vahetult juurtipu all. Kui me hakkaks seda ühendamisel tegema, siis kaotaksime oluliselt ühendamise operatsiooni efektiivsuses. Samas hulga leidmise operatsiooni ajal saame lihtsalt kõikide teele jäävate tippude vanemat muuta:

```
int leia2(int x) {
      if (hulgad[x] != hulgad[hulgad[x]]) hulgad[x] = leia2(hulgad[x]);
      return hulgad[x];
}
```

Pane tähele, et me ei vaevu muutma astmeid, mistõttu need ei pruugi enam vastata puu tegelikele kõrgustele. Sellele ei ole lihtsalt praktiline aega kulutada. Kuna tee pakkimine ei muuda tippude arvu

puus, siis jääb kehtima ka varem tõestatud väide, et puus astmega n on vähemalt 2^n tippu. Puu tegelik kõrgus ei saa kunagi olla suurem kui puu aste.

Koos tee pakkimisega m ühendamise või leidmise operatsiooni on keerukusega $O((m+n)log^*n)$, kus log^*n on pöördfunktsioon funktsioonist $2^{2^{2^{\cdots 2}}}$, kus siis astendatakse n korda. Selle funktsiooni väärtus kasvab väga kiiresti: 2, 4, 16, 65 536, $2*10^{19728}$. Vastava pöördfunktsiooni väärtus on jällegi väga väike ja ei lähe praktilises kasutuses üle viie, mis tähendab, et lisamise ja ühendamise operatsioonide keerukus on m operatsiooni korral $O(m+n)$.

```
bool yhenda(int x, int y) {
     int xx = leia2(x), yy = leia2(y);
     if (xx == yy) return false;
     if (rank[xx] > rank[yy]) { int t = xx; xx = yy; yy = t; }
     if (rank[xx] == rank[yy]) rank[yy]++;
     hulgad[xx] = yy;
     return true;
}
```

9.7.1 Kasutamine

Ühisosata hulki kasutatakse peamiselt graafiülesannete juures, nimelt minimaalse toese leidmiseks. Sellest tuleb täpsemalt juttu järgmises peatükis.

9.8 KONTROLLÜLESANDED

9.8.1 Laoseisud

Reas on n (1 ≤ n ≤ 200000) laohoonet. i-ndas laohoones on r (0 ≤ r ≤ 1000) võistlusprogrammeerimise õpikut. Vahetevahel lisatakse õpikuid laohoonesse juurde või võetakse neid sealt ära. Samuti tahetakse aeg-ajalt teada, mitu õpikut on mingites järjestikku asuvates laohoonetes kokku.
Sisendi esimesel real on täisarv n. Järgmisel n real on laohoonetes olevate raamatute arv. Sellele järgneb ühel real arv q (1 ≤ q ≤ 200000) – reas tehtavate operatsioonide arv. Järgmised q rida on kas kujul „L x r“ (1 ≤ x ≤ n; 1 ≤ r ≤ 1000) – tehakse operatsioon, mille tagajärjel on laohoones x r raamatut, või „U x y“ (1 ≤ x ≤ y ≤ n) – uuritakse, mitu õpikut on laohoonetes vahemikus x…y. Väljundisse kirjutada mingi hulk ridu. Igale reale on vaja väljastada üks täisarv: vastava teist tüüpi operatsiooni vastus.

NÄIDE 1:

```
3
100
100
100
6
U 1 1
U 1 3
L 2 200
U 1 2
L 3 0
U 2 3
```

Vastus:

```
100
300
300
200
```

NÄIDE 2:

```
10
1
2
3
4
5
6
7
8
9
10
1
U 1 10
```

Vastus:

```
55
```

9.8.2 Korrutamismäng

Sa mängid sõpradega mängu, kus sul on arvujada pikkusega N ning kus sulle antakse K käsku. Iga käsk on kas muutmise käsk, kus sa pead üht arvu jadas muutma, või korrutamise käsk, kus sa pead ütlema, kas mingis vahemikus olevate arvude korrutis on positiivne, negatiivne või 0. Seega otsustasid teha programmi, mis seda mängu mängib.

Sisendi esimesel real on kaks täisarvu N ja K ($1 \le N, K \le 10^5$). Teisel real on N täisarvu – esialgse jada väärtused x ($-100 \le x \le 100$). Järgmisel K real on käsud, mis on kas muutmise või korrutamise käsud. Muutmise käsk antakse ette kujul M x y ($1 \le x \le N$; $-100 \le y \le 100$), mis tähendab, et arv kohal x on nüüd väärtusega y. Korrutamise käsk antakse ette kujul K x y ($1 \le x \le y \le N$), mis tähendab, et on vaja uurida väärtust vahemikus x…y.

Väljundisse kirjutada ühele reale üks string: iga korrutamise käsu kohta peab olema kas +, kui korrutis on positiivne, -, kui korrutis on negatiivne, või 0, kui korrutis on 0.

NÄIDE 1:

```
4 6
-2 6 0 -1
M 1 10
K 1 4
M 3 7
K 2 2
M 4 -5
K 1 4
```

Vastus:

```
0+-
```

NÄIDE 2:

```
5 9
1 5 -2 4 3
K 1 2
K 1 5
M 4 -5
K 1 5
K 4 5
M 3 0
K 1 5
M 4 -5
M 4 -5
```

Vastus:

```
+-+-0
```

9.8.3 Külaskäigud

Mäe nõlval on tee, mille ääres on n maja. Igal majal on kõrgus merepinnast ning on teada, et majade kõrgused moodustavad mittekahaneva rea (Järgmine maja ei ole kunagi eelmisest majast madalamal). Samuti teame, et iga maja laius on 1 ning kahe erineva kõrgusega maja vahel on aste. Majades elab q paari lapsi, kes käivad üksteisel külas. Lapsed tahavad teada, kui pikk on pikim teelõik, mis on ühel kõrgusel, mida nad külla minnes läbivad. Laps läbib külla minnes ka oma maja pikkuse ning ka oma sõbra maja pikkuse.
Sisendi esimesel real on kaks täisarvu: n ja q ($1 \le n,q \le 100000$). Järgmisel real on n täisarvu vahemikus $-100000 \ldots 100000$ – majade kõrgused merepinnast. Järgmisel q real on kaks täisarvu: x ja y ($1 \le x \le y \le n$) – laste elukohad.
Väljundisse kirjutada q rida: Pikima ühel kõrgusel oleva teelõigu pikkus, mida laps peab külla minnes läbima.

NÄIDE:

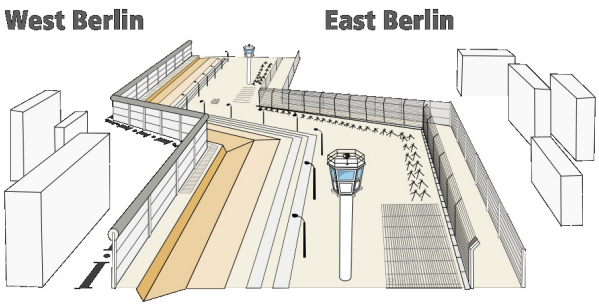
```
10 3
-1 -1 1 1 1 1 3 10 10 10
2 3
1 10
5 10
```

Vastus:

```
1
4
3
```

9.8.4 Berliini müür

Tulevane Berliini müür jagati enne ehitamist tulpadeks. Müüri otsustati ehitada osade kaupa, see tähendab, et valiti mingid lõigud, mille kohta oli teada, et müür peab ilmtingimata olema sellel lõigul teatud kõrgusel. Selle jaoks alustati tööd selles lõigus olevate tulpade, mis olid lühemad nõutud kõrgusest, kõrgendamisega. Iga sellise lõigu ehitamise järel tehti viieminutiline paus. Sinu eesmärk on teada, mitu korda alustati kokku tööd mingisuguse tulba üles ehitamisega.
Sisendi esimesel real on arv n ($1 \le n \le 100000$) – vajalike müürilõikude arv.
Järgmisel n real on kolm täisarvu: l, r ja h ($1 \le l < r \le 100000$; $1 \le h \le 10^9$) – müürijupi otspunktid ning kõrgus.
Väljundisse kirjutada üks täisarv: töö alustamiste arv.



NÄIDE:

```
3
5 11 3
1 10 1
3 13 2
```

Vastus:

```
14
```

9.8.5 Lambipirnid

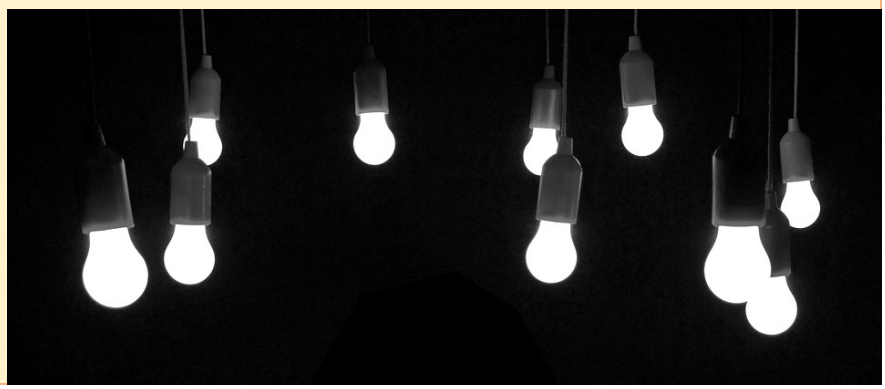
Reas on `N` (`1 ≤ N ≤ 1000000`) lambipirni. Iga pirn kas põleb või ei põle. Vahetevahel pannakse kõik mingis vahemikus olevad pirnid põlema, vahetevahel kustutatakse kõik mingis vahemikus olevad pirnid, vahel muudetakse kõigi mingis vahemikus olevate pirnide olekut ning vahel küsitakse sinu käest, mitu pirni mingis vahemikus põlevad.
Sisendi esimesel real on arv `n`. Järgmisel real on `n` märgist koosnev sõne. `i`-s märk sõnes on `0`, kui vastav pirn ei põle, ning `1`, kui vastav pirn põleb. Järgneb üksikul real arv `q` (`1 ≤ q ≤ 1000`) – päringute arv, mis tehakse. Järgmisel `q` real on tähemärk ning kaks arvu kujul `c a b` (`0 ≤ a ≤ b <` `n`). Kui `c = 'F'`, pannakse põlema kõik pirnid vahemikus a…b. Kui `c = 'E'`, kustutatakse kõik pirnid vahemikus a…b. Kui `c = 'I'`, muudetakse kõikide pirnide olek vahemikus a…b. Kui `c = 'S'`, küsitakse sinu käest, mitu pirni põlevad vahemikus a…b.
Väljundisse kirjutada mingi hulk ridu. Iga korra kohta, kui küsitakse, mitu pirni mingis vahemikus põlevad, väljastada ühele reale vastav pirnide arv.

NÄIDE 1:

```
18
101010101010001000
5
F 0 17
I 0 5
S 1 10
E 4 9
S 2 10
```

Vastus:

```
5
1
```



NÄIDE 2:

```
9
111000000
2
I 0 2
S 0 8
```

Vastus:

```
0
```

9.8.6 Taani valimised

Taanis on valimiste eel kogu maa plakateid täis. Üks kohtadest, kuhu plakateid riputati, on plakatikõrgune sein Holbækvej tänaval Sorøs. See nn Sorø müür on $l = 10^7$ millimeetrit pikk, kus iga millimeetri järel on tehtud märge. Plakat, mis seinale riputatakse, peab olema riputatud täpselt märkmest märkmeni ning selle pikkus peab olema vähemalt kaks millimeetrit. Valimiste eel riputatakse Sorø müürile n ($1 \le n \le 40000$) plakatit. Kuna plakat või osaliselt või täielikult katta ka teist plakatit, leida, mitu plakatit on valimispäeval osaliselt või täielikult nähtaval.
Sisendi esimesel real on täisarv n. Järgneval n real on kaks täisarvu:
v ja p ($1 \le v < p \le l$) – plakati esimese ning viimase kaetud millimeetri järjekorranumber. Plakatid on ette antud nende riputamise järjekorras.
Väljundisse kirjutada üksikule reale üks täisarv: Plakatite arv, mis on kas osaliselt või täielikult katmata.

NÄIDE:

```
5
1 4
2 6
8 10
3 4
7 10
```

Vastus:

```
4
```

9.8.7 Rivistus

n last seisavad reas. Sul on vaja m korda vastata küsimusele, kui pikk on k-s kõige lühem laps vahemikus i…j.
Sisendi esimesel real on kaks arvu: n ja m ($1 \le n \le 100000$, $1 \le m \le 5000$). Sisendi teisel real on jada pikkusega n – reas seisvate laste pikkused. Järgneval m real on kolm arvu:
i, j ja k ($1 \le i \le j \le n$, $1 \le k \le j - i + 1$).
Väljundisse kirjutada eraldi ridadele m täisarvu: vastused päringutele.

NÄIDE:

```
7 3
1 5 2 6 3 7 4
2 5 3
4 4 1
1 7 3
```

Vastus:

```
5
6
3
```

9.8.8 Madalad majad

Linnas on n hoonet ning n − 1 teed hoonete vahel, nii et iga hoone juurest saab iga hoone juurde sõita. Sinult küsitakse m korda, mis on k-nda kõige lühema hoone kõrgus, millest möödutakse teel hoonest u hoonesse v.
Esimesel real on kaks täisarvu: N ja M (N,M ≤ 100000). Järgmisel real on N täisarvu: hoonete kõrgused. Järgneval N − 1 real on kaks täisarvu: u ja v – hoonete u ja v vahel on tee. Järgneval M real on kolm tühikutega eraldatud täisarvu: u, v ja k – küsitakse k kõige lühemat hoonet teel hoonest u hoonesse v.
Väljundisse kirjutada erinevatele ridadele M täisarvu: vastused päringutele.

NÄIDE:

```
8 5
105 2 9 3 8 5 7 7
1 2
1 3
1 4
3 5
3 6
3 7
4 8
2 5 1
2 5 2
2 5 3
2 5 4
7 8 2
```

Vastus:

```
2
8
9
105
7
```

9.8.9

Ruudukujulisel toal, mille küljepikkus on N meetrit, on põrand jaotatud N^2 ruutmeetriseks platvormiks, mille kõrgus võib olla vabalt valitud täisarv millimeetrites. Sulle antakse ette selle ruumi algseis ning sinu eesmärk on öelda, mis on selle toa mingi alamristküliku kõige kõrgema ning kõige madalama platvormi kõrgus teatud ajahetkedel. Selliste küsimuste vahel võidakse muuta mingite üksikplatvormide kõrgusi.

Sisendi esimesel real on täisarv N ($0 \le N \le 500$) – toa küljepikkus. Järgneval N real on igal real N tühikutega eraldatud täisarvu – platvormide esialgsed kõrgused. Järgmisel real on täisarv Q ($0 \le Q \le 40000$). Järgmised Q rida on antud ette kas kujul q x_1 y_1 x_2 y_2, mis tähendab, et tahetakse teada, mis on suurima ning vähima kõrgusega platvorm ristkülikus, mille ülemine vasak nurk on (x_1, y_1) ning alumine parem nurk on (x_2, y_2), või siis c x y v, mis tähendab, et platvormi, mille koordinaadid on (x, y), on nüüd kõrgusega v.

Väljundisse kirjutada mingi hulk ridu. Iga esimest tüüpi päringu kohta väljastada kaks tühikutega eraldatud täisarvu: suurima ning vähima kõrgusega platvorm nõutud vahemikus.

NÄIDE:

```
5
1 2 3 4 5
0 9 2 1 3
0 2 3 4 1
0 1 2 4 5
8 5 3 1 4
4
q 1 1 2 3
c 2 3 10
q 1 1 5 5
q 1 2 2 2
```

Vastus:

```
9 0
10 0
9 2
```

9.8.10 Paberi lõikamine

Sa lõikad üht pikka värvilist pabeririba. Pabeririba on n millimeetrit pikk ning iga tükk, mida sa selle küljest lõikad, peab olema pikkusega, mis on täisarv millimeetreid. Samuti vaheldub riba värv ainult täisarvu millimeetrite järel. Sinu ülemused on öelnud, et ükski sinu lõigatud tükk ei tohi sisaldada rohkem kui k erinevat värvi, aga nad unustasid sulle k väärtust öelda. Leia iga võimaliku k väärtuse jaoks, kus $1 \leq k \leq n$, mis on minimaalne arv lõikeid, mida sa pead ribasse tekitama.
Sisendi esimesel real on täisarv n ($1 \leq n \leq 100000$). Järgmisel real on n tühikutega eraldatud täisarvu a ($1 \leq a \leq n$) – riba iga millimeetri värv.
Väljundisse kirjutada n tühikutega eraldatud täisarvu: iga k (väiksemast suuremani) väärtuse jaoks minimaalne lõikuste arv.

NÄIDE 1:

```
5
1 3 4 3 3
```

Vastus:

```
3 1 0 0 0
```

NÄIDE 2:

```
8
1 5 7 8 1 7 6 1
```

Vastus:

```
7 3 2 1 0 0 0 0
```

9.9 VIITED LISAMATERJALIDELE

Kaasasolevas failis VP_lisad.zip, peatükk10 kaustas on abistavad failid käesoleva peatüki materjalidega põhjalikumaks tutvumiseks:

Fail	Kirjeldus
RMQ.cpp, RMQ.java, RMQ.py	Lõikude puu
Ribad.cpp, Ribad.java, Ribad.py	Lõikude puu laisa väärtustamisega
Sein.cpp, Sein.java, Sein.py	Lõikude puu laisa väärtustamisega
RCS.cpp, RCS.java, RCS.py	Kumulatiivne summa (Fenwicki puu)
Punktid.cpp, Punktid.java, Punktid.py	2-mõõtmeline Fenwicki puu
Redaktor.cpp, Redaktor.java, Redaktor.py	Ajalooga lõikude puu
UF.cpp, UF.java, UF.py	Ühisosata hulgad