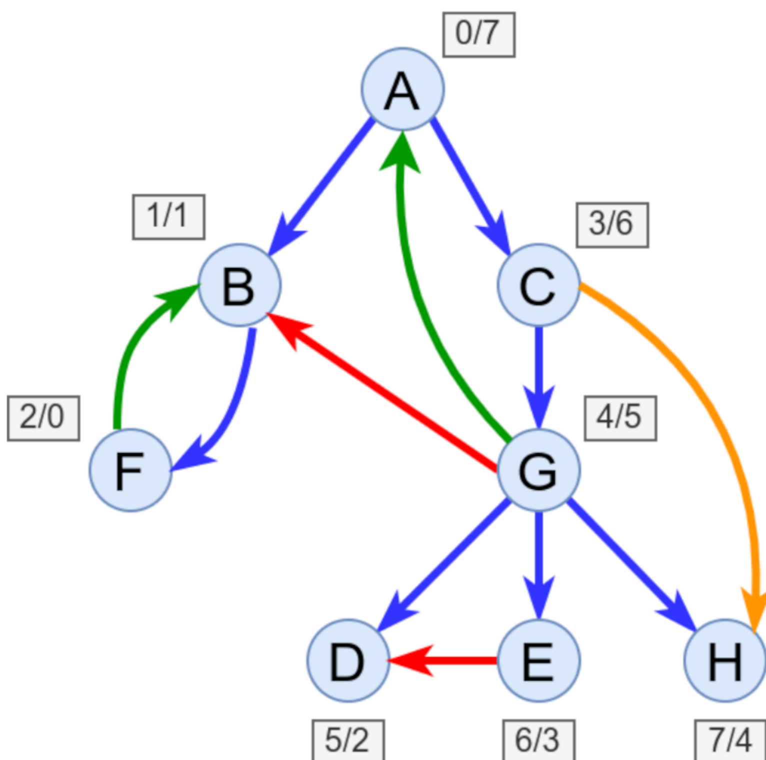
МАТФ
Универзитет у Београду
Математички факултет

ПРВО ИЗДАЊЕ



КОНСТРУКЦИЈА И АНАЛИЗА АЛГОРИТАМА

ВЕСНА МАРИНКОВИЋ

ФИЛИП МАРИЋ

Весна Маринковић Филип Марић

Конструкција и анализа алгоритама

Математички факултет, Универзитет у Београду

Београд
2024.

Аутори:
др Весна Маринковић,
доцент на Математичком факултету у Београду
др Филип Марић,
редовни професор на Математичком факултету у Београду

КОНСТРУКЦИЈА И АНАЛИЗА АЛГОРИТАМА
Прво издање, 2024.

Издавач:
Математички факултет Универзитета у Београду
Студентски трг 16, 11000 Београд,
(+381) 011 2027 801, `matf@matf.bg.ac.rs`
За издавача: *проф. др Зоран Ракић*, декан

Рецензенти:
др Миодраг Живковић,
редовни професор на Математичком факултету у Београду, у пензији
др Предраг Јаничић,
редовни професор на Математичком факултету у Београду
др Драган Урошевић,
научни саветник на Математичком институту САНУ

Обрада текста, илустрације и корице: *аутори*
Штампа: Скрипта интернационал, Београд
Тираж: 150

©2024. Весна Маринковић и Филип Марић
Ово дело заштићено је лиценцом Creative Commons CC BY-NC-ND 4.0 (Attribution-NonCommercial-NoDerivatives 4.0 International License). Детаљи лиценце се могу видети на веб-адреси `http://creativecommons.org/licenses/by-nc-nd/4.0/`. Дозвољено је умножавање, дистрибуција и јавно саопштавање дела, под условом да се наведу имена аутора. Употреба дела у комерцијалне сврхе није дозвољена. Прерада, преобликовање и употреба дела у склопу неког другог није дозвољена.

Садржај

Предговор . . . 7
1 Напредне структуре података . . . 9
1.1 Префиксно дрво 10
1.1.1 Структура префиксног дрвета . . . 11
1.1.2 Операције над префиксним дрветом . . . 13
Задатак: Пар који даје највећи XOR . . . 17
1.2 Фибоначијев хип 21
1.2.1 Структура хипа . . . 23
1.2.2 Операције над хипом . . . 25
1.2.2.1 Уметање . . . 26
1.2.2.2 Унија два хипа . . . 26
1.2.2.3 Одређивање минимума . . . 26
1.2.2.4 Брисање најмањег елемента . . . 26
1.2.2.5 Смањивање вредности кључа . . . 32
1.3 Структуре података за представљање дисјунктних скупова 36
1.3.1 Наивна имплементација . . . 37
1.3.2 Ефикасна имплементација . . . 40
1.3.2.1 Структура и операције . . . 40
1.3.2.2 Уравнотежавање дрвета . . . 44
1.3.2.3 Сажимање путева . . . 48
Задатак: Први пут кроз матрицу . . . 51

МАТФ Беог

1.4 Упити распона 54
1.4.1 Статички упити распона . . . 55
1.4.1.1 Збирови префикса . . . 55
1.4.1.2 Разлике суседних елемената . . . 57
1.4.2 Динамички упити распона . . . 58
1.4.2.1 Сегментна дрвета . . . 59
1.4.2.2 Фенвикова дрвета (BIT) . . . 73
Задатак: Максимални подсегмент . . . 83
Задатак: Број инверзија . . . 87
Задатак: К-ти парни број . . . 89
Задатак: Број различитих елемената у сегментима . . . 91
1.4.3 Ажурирање сегмената . . . 95
1.4.3.1 Дрво над низом разлика . . . 96
1.4.3.2 Два дрвета разлика . . . 97
1.4.3.3 Лењо ажурирање сегментног дрвета . . . 101
2 Графовски алгоритми . . . 111
2.1 Основни појмови 112
2.2 Представљање графа 116
2.2.1 Матрица повезаности . . . 116
2.2.2 Листе повезаности . . . 117
2.3 Обилазак графа 119
2.3.1 Обилазак у дубину . . . 121
2.3.1.1 Неусмерени графови . . . 121
2.3.1.2 Усмерени графови . . . 140
2.3.1.3 Провера постојања циклуса . . . 146
2.3.2 Обилазак у ширину . . . 147
2.3.2.1 Покретање обиласка из више чворова истовремено . . . 154
Задатак: Постављање рачунарске мреже . . . 155
Задатак: Провера да ли је граф бипартитан . . . 158
Задатак: Авионска преседања . . . 160
Задатак: Косе црте . . . 163
2.4 Тополошко сортирање 168
2.4.1 Канов алгоритам . . . 170
2.4.2 Алгоритам заснован на претрази у дубину . . . 174
Задатак: Организација пројектних активности . . . 177
2.5 Мостови и артикулационе тачке у неусмереном графу 180
2.5.1 Одређивање мостова . . . 182
2.5.1.1 Тарџанов алгоритам за одређивање мостова . . . 183

МАТФ Београд

2.5.2 Одређивање артикулационих тачака . . . 192
2.5.2.1 Тарџанов алгоритам за одређивање артикулационих тачака . . . 192
Задатак: Усмеравање путева . . . 196
2.6 Компоненте јаке повезаности графа 198
2.6.1 Алгоритам заснован на обиласку из свих чворова . . . 200
2.6.2 Тарџанов алгоритам . . . 201
2.6.2.1 Распоред компоненти у DFS дрвету . . . 201
2.6.2.2 Издвајање компоненти уз помоћ стека . . . 206
2.6.2.3 Одређивање базних чворова компоненти . . . 208
2.6.3 Косараџуов алгоритам . . . 220
Задатак: Докажи све формуле! . . . 226
2.7 Ојлерови и Хамилтонови путеви 229
2.7.1 Ојлерови путеви и циклуси . . . 229
2.7.1.1 Хирхолцеров алгоритам . . . 234
2.7.1.2 Флеријев алгоритам . . . 242
2.7.2 Хамилтонови путеви и циклуси . . . 242
Задатак: Де Брујнов низ . . . 244
2.8 Тежински графови 247
2.9 Најкраћи путеви из задатог чвора 248
2.9.1 Ациклички графови . . . 249
2.9.1.1 Рекурзивни приступ и динамичко програмирање . . . 250
2.9.1.2 Индуктивни приступ: истовремено тополошко сортирање и одређивање најкраћих путева . . . 251
2.9.2 Графови са ненегативним гранама: Дајкстрин алгоритам . . . 256
2.9.3 Графови са негативним гранама . . . 267
2.9.3.1 Општи алгоритам заснован на релаксацији грана . . . 269
2.9.3.2 Белман-Фордов алгоритам . . . 272
Задатак: Најмање напорна стаза . . . 279
Задатак: Мењачница . . . 282
2.10 Минимално повезујуће дрво 283
2.10.1 Индукција по броју чворова и грана . . . 284
2.10.2 Примов алгоритам . . . 286
2.10.3 Краскелов алгоритам . . . 292
Задатак: Вода до сваке куће . . . 296
2.11 Сви најкраћи путеви 298
2.11.1 Алгоритам заснован на индукцији по броју грана у графу . . . 299
2.11.2 Алгоритам заснован на индукцији по броју чворова у графу . . . 300
2.11.3 Флојд-Варшалов алгоритам . . . 302
2.11.4 Транзитивно затворење и транзитивна редукција . . . 311

МАТФ Београд

Задатак: Централни чвор . . . 316
3 Алгебарски алгоритми . . . 319
3.1 Еуклидов алгоритам 320
3.1.1 Основни Еуклидов алгоритам . . . 321
3.1.2 Проширени Еуклидов алгоритам . . . 326
3.1.2.1 Представљање НЗД преко узастопних остатака . . . 327
3.1.2.2 Представљање остатака преко a и b . . . 329
3.1.3 Решавање линеарних Диофантових једначина . . . 332
Задатак: Исти остаци свих бројева . . . 334
Задатак: Кутије за кликере . . . 337
3.2 Растављање на просте чиниоце (факторизација) 339
3.2.1 Факторизација испробавањем делилаца . . . 339
3.2.2 Фермаов алгоритам факторизације . . . 343
3.2.3 Факторизација више бројева помоћу Ератостеновог сита . . . 344
Задатак: Допуна до пуног квадрата . . . 348
Задатак: Највећи НЗД бројева чији је производ познат . . . 350
3.3 Мултипликативне функције 351
3.3.1 Ојлерова функција . . . 352
3.3.1.1 Директан алгоритам . . . 352
3.3.1.2 Рачунање Ојлерове функције броја свођењем на факторизацију . . . 353
3.3.1.3 Рачунање Ојлерове функције свих бројева до n . . . 356
3.3.2 Функције делилаца . . . 361
3.3.2.1 Број делилаца . . . 361
3.3.2.2 Збир делилаца . . . 364
Задатак: Број делилаца производа . . . 368
Задатак: Збир НЗД свих парова делилаца . . . 370
3.4 Модуларна аритметика 375
3.4.1 Модуларне групе . . . 382
3.4.2 Ојлерова и мала Фермаова теорема . . . 385
3.4.2.1 Фермаов тест да ли је број прост . . . 388
3.4.3 Цикличност модуларних мултипликативних група . . . 388
3.4.4 Израчунавање модуларног мултипликативног инверза . . . 389
3.4.4.1 Алгоритам грубе силе . . . 390
3.4.4.2 Алгоритам заснован на проширеном Еуклидовом алгоритму . . . 390
3.4.4.3 Алгоритам заснован на Ојлеровој и Малој Фермаовој теореми . . . 392
3.4.5 Кинеска теорема о остацима . . . 394
3.4.5.1 Груба сила . . . 395
3.4.5.2 Алгоритам заснован на просејавању . . . 395
3.4.5.3 Алгоритам заснован на Лагранжевом приступу . . . 396
3.4.5.4 Алгоритам заснован на Безуовој теореми . . . 401

МАТФ Београд

Задатак: Билијар . . . 402
3.5 **RSA криптографија** **406**
3.6 **Брза Фуријеова трансформација** **410**
3.6.1 Директна брза Фуријеова трансформација . . . 414
3.6.2 Инверзна Фуријеова трансформација . . . 425
3.6.3 Алгоритам брзе Фуријеове трансформације . . . 428
3.6.4 NTT: Фуријеова трансформација у модуларној аритметици . . . 434
Задатак: Поравнавање ниски . . . 437
Задатак: Дигитални бројач . . . 440
4 **Алгоритми за анализу и обраду текста** . . . **443**
4.1 **Хеширање ниски** **444**
4.1.1 Хеш-функције и њихова својства . . . 446
4.1.2 Дефинисање хеш-функције . . . 447
4.1.2.1 Полиномска хеш-функција слева-надесно . . . 447
4.1.2.2 Полиномска хеш-функција здесна-налево . . . 450
4.1.3 Тражење ниске у тексту (Рабин-Карпов алгоритам) . . . 451
4.1.4 Рачунање хеш-вредности сегмената ниске . . . 453
4.1.4.1 Полиномска хеш-функција слева-надесно . . . 453
4.1.4.2 Полиномска хеш-функција здесна-налево . . . 456
Задатак: Груписање једнаких ниски . . . 458
Задатак: Број различитих сегмената . . . 461
Задатак: Лексикографски најмања ротација . . . 464
4.2 **Z-низ** **466**
4.2.1 Конструкција z-низа . . . 467
4.2.1.1 Алгоритам грубе силе . . . 467
4.2.1.2 z-алгоритам . . . 468
4.2.2 Претрага текста применом z-низа . . . 473
4.3 **Алгоритам KMP** **474**
4.3.1 Предобрада ниске која се тражи . . . 477
4.3.2 Претраживање текста . . . 483
4.3.3 Испитивање периодичности ниске . . . 487
Задатак: Најкраћа допуна до палиндрома . . . 491
Задатак: Домине . . . 493
4.4 **Најдужи палиндромски сегмент - Маначеров алгоритам** **494**
4.4.1 Провера свих сегмената . . . 494
4.4.2 Провера свих сегмената редом према опадајућим дужинама . . . 494
4.4.3 Провера центара . . . 495
4.4.4 Експлицитна допуна речи и позиција . . . 497
4.4.5 Имплицитна допуна речи и позиција . . . 498

МАТФ Београд

4.4.6 Маначеров алгоритам . . . 500
5 Геометријски алгоритми . . . 507
5.1 Основе геометријских алгоритама 508
5.1.1 Тачке, координате . . . 508
5.1.1.1 Декартове координате . . . 508
5.1.1.2 Поларне координате . . . 509
5.1.2 Вектори . . . 510
Задатак: Стрелица . . . 512
5.1.3 Скаларни производ . . . 514
5.1.3.1 Пројекција вектора на правац вектора . . . 515
5.1.3.2 Пројекција тачке на праву . . . 516
5.1.4 Векторски производ . . . 517
5.1.4.1 Дводимензионални векторски производ . . . 518
5.1.5 Површина и примене . . . 521
5.1.5.1 Растојање тачке од праве . . . 524
5.1.6 Оријентација тројке тачака и примене . . . 525
5.1.6.1 Провера да ли су тачке са исте стране праве . . . 527
5.1.6.2 Испитивање да ли тачка припада унутрашњости троугла . . . 530
5.1.6.3 Пресек дужи . . . 532
5.1.7 Многоуглови . . . 536
5.1.7.1 Испитивање да ли је многоугао прост . . . 537
5.1.7.2 Испитивање да ли је многоугао конвексан . . . 537
5.1.7.3 Површина простог многоугла . . . 538
5.2 Пресеци хоризонталних и вертикалних дужи 538
5.3 Конструкција простог многоугла 545
5.4 Припадност тачке унутрашњости многоугла 551
5.4.1 Испитивање припадности тачке унутрашњости простог многоугла . . . 551
5.4.2 Испитивање припадности тачке унутрашњости конвексног многоугла . . . 556
5.4.2.1 Алгоритам заснован на оријентацији . . . 556
5.4.2.2 Алгоритам заснован на бинарној претрази . . . 557
5.5 Конструкција конвексног омотача 560
5.5.1 Директни индуктивни приступ . . . 562
5.5.2 Увијање поклона . . . 566
5.5.3 Грејемов алгоритам . . . 569
5.5.4 Брзи алгоритам за тражење конвексног омотача . . . 573
Задатак: Најдаље тачке . . . 578
Задатак: Пресек конвексних многоуглова . . . 582
Литература . . . 591

МАТФ Београд

Предговор

Уџбеник пред вама је намењен студентима друге године Математичког факултета Универзитета у Београду и користи се за предмет „Конструкција и анализа алгоритама” на другој години студијског програма Информатика.

Претпоставља се да су студенти у ранијем школовању успешно савладали основне елементе програмирања и да су овладали основама алгоритмике (асимптотском анализом сложености и O-нотацијом, основним техникама конструкције и анализе алгоритама и основним структурама података).

Теоријски прегледи алгоритама и структура података дају информације на основу којих би читалац требало да буде у могућности да самостално креира имплементацију у програмском језику који одабере. Ипак, илустрације ради, алгоритми су имплементирани у савременом језику C++ уз интензивно коришћење стандардне библиотеке тог програмског језика. Знање овог језика и библиотеке се, стога, подразумева. Приказане су функције које имплементирају ове алгоритме, док је писање потпуних програма који користе ове функције препуштено читоацу. Комплетна решења су доступна у онлајн издању уџбеника.

Материјал изложен у овом уџбенику увелико превазилази оквире једног једносеместралног курса и на наставнику је да одабере подскуп тема које жели да обради у току курса, имајући у виду примене, потребе наредних курсева, али и предзнање студената. Неке теме се могу и оставити студентима за самостални рад.

Након сваког поглавља дато је неколико задатака који илуструју примене описаних техника. Ти задаци представљају само илустрацију, а студентима који желе да стекну боље

програмерске вештине се саветује да прораде већи број задатака који се могу наћи у наменским збиркама задатака из области алгоритмике, али и на многим онлајн порталима посвећеним учењу програмирања (на пример, `petlja.org`, `codeforces.com`, `leetcode.com`, `geeksforgeeks.org` итд.).

Књига има и пратеће онлајн издање, доступно на адреси http://algoritmi.matf.bg.ac.rs/kiaa/index.html, у коме се налазе интерактивни аплети који могу помоћи студентима у разумевању неких тема.

Захвални смо професору Миодрагу Живковићу, који је кроз свој уџбеник и предавања утемељио приступ изучавању алгоритама на Математичком факултету. На веома пажљивом читању и коментарима и сугестијама које су допринеле квалитету књиге захваљујемо рецензентима Миодрагу Живковићу, Предрагу Јаничићу и Драгану Урошевићу и колегиници Тијани Шукиловић. Такође се захваљујемо сарадницима и студентима који су својим коментарима допринели исправљању уочених грешака и пропуста, посебно Матији Лојовићу, Лазару Јовановићу, Марку Цвијетиновићу, Жељку Зекавичићу и Милици Зубљић. Захвални смо и колеги Николи Ајзенхамеру на помоћи око графичког обликовања уџбеника.

Моле се читаоци да на све пропусте и евентуалне грешке укажу ауторима.

Аутори

МАТФ Београд

1. Напредне структуре података

У овом поглављу приказана је имплементација неких напредних структура података. Претпостављамо да је читалац упознат са коришћењем и имплементацијом основних структура података: секвенцијалних структура података (низа, једноструко и двоструко повезане листе), стека, реда, реда са два краја, реда са приоритетом, као и основних асоцијативних структура података (скупа, мултискупа и мапе, тј. речника) коришћењем хеш-табела и балансираних уређених бинарних дрвета. За разлику од ових елементарнијих структура података, напредне структуре по правилу нису део стандардних библиотека програмских језика (на пример, нису укључене у библиотеке језика C++, C#, Python, Java) и потребно их је посебно имплементирати (или преузети неку јавно доступну имплементацију).

У овом поглављу ћемо проучити следеће структуре података.

- **Префиксно дрво** (енгл. trie) омогућава да се на још један начин имплементирају асоцијативне структуре података (поред уређених бинарних дрвета и хеш-табела), код којих се уметање и претрага врши на основу кључева који су обично или ниске карактера или ниске декадних или бинарних цифара. Ова дрвета се обично користе у применама обраде текста, као што су провера правописа и обрада природног језика, захваљујући својој могућности да ефикасно складиште велике речнике који садрже мноштво сличних речи, прецизније речи које деле заједничке префиксе.

- **Фибоначијев хип** представља још један начин да се имплементира ред са приоритетом (поред коришћења класичног бинарног хипа), код којег се, за разлику од класичног хипа, операције уметања, али и смањивање приоритета неког елемента

и спајања два хипа врше у амортизованом константном времену (подсетимо се, ове операције се код класичних хипова врше у логаритамском времену).

- **Структура за представљање дисјунктних скупова** (енгл. disjoint set union или union find) омогућава да се одржавају колекције дисјунктних скупова (подскупова неког скупа), уз могућност ефикасног проналажења скупа коме дати елемент припада и ефикасног спајања два скупа у један.

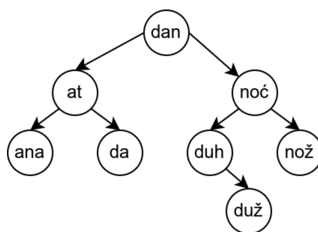
- **Структуре за ефикасно извршавање упита распона** (енгл. range queries) омогућавају да се након одређене предобраде (енгл. preprocessing) низа елемената ефикасно извршавају операције над његовим сегментима (поднизовима узастопних елемената), попут, на пример, израчунавања збира елемената сегмента, минимума или максимума елемената сегмента итд. Неке од ових структура допуштају ефикасно комбиновање ажурирања података (промену појединачних елемената или ажурирања целих сегмената увећањем свих елемената за неку вредност) и израчунавања поменутих статистика.

1.1 Префиксно дрво

Структура података са асоцијативним приступом (скуп, мапа тј. речник), код које се приступ елементима врши по кључу, ефикасно се имплементира коришћењем уређеног бинарног дрвета или хеш-табеле[1]. Кључ не мора бити целобројна вредност, већ ниска карактера, ниска битова или нешто друго.

Пример 1.1.1

На слици 1.1 приказано је уређено бинарно дрво[2] које садржи ниске ana, at, noć, nož, da, dan, duh и duž као кључеве.



Слика 1.1: Уређено бинарно дрво чији су кључеви ниске.

[1] Код асоцијативних структура података приступ елементима се врши на основу вредности кључа, а не на основу индекса, односно позиције елемента у структури података.

[2] Поред термина дрво (енгл. tree), често се користи и термин стабло.

МАТФ Београд

Приликом свих операција са уређеним бинарним дрветом (претрага, брисање, уметање) у сваком чвору се врши поређење кључева (оног који је записан у чвору и оног који се обрађује) и када су кључеви ниске (али и неки други већи подаци), то поређење може захтевати доста времена и лоше утицати на перформансе.

Још једна структура података у виду дрвета која омогућава ефикасан асоцијативни приступ када су кључеви ниске је *префиксно дрво* (енгл. prefix tree), такође познато под енглеским називом *trie* (од енглеске речи *reTRIEval*). Префиксно дрво ефикасно решава следећи проблем.

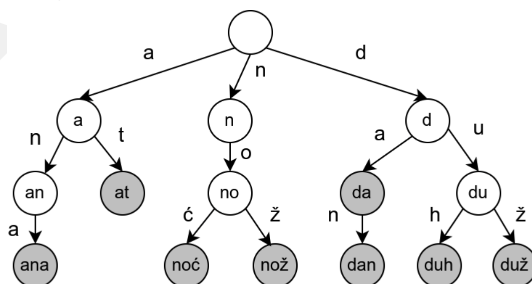
Проблем

Дефинисати асоцијативну структуру података која омогућава интерфејс скупа/мапе (додавање, тражење и брисање елемената) у којој су кључеви ниске или низови, а која подржава и ефикасно проналажење свих елемената чији кључеви имају задати префикс.

1.1.1 Структура префиксног дрвета

Основна идеја ове структуре података је да се кључ придружен чвору добија надовезивањем карактера који се налазе на гранама дуж путање од корена до тог чвора. Корен садржи празну реч, а преласком преко гране се на до тада формирану реч здесна надовезује још један карактер, па сваком чвору одговара неки префикс кључа. Притом, заједнички префикси различитих кључева су представљени истим путањама од корена до тачке разликовања (чвора који одговара најдужем заједничком префиксу). Чворови префиксног дрвета могу имати различит број деце, али максимални број деце одређен је величином азбуке која се користи за кодирање кључева. Ако се помоћу префиксног дрвета имплементира мапа (речник), тада се сваком кључу придружује вредност (податак који се чува у чвору дрвета којим се комплетира тај кључ).

Пример 1.1.2

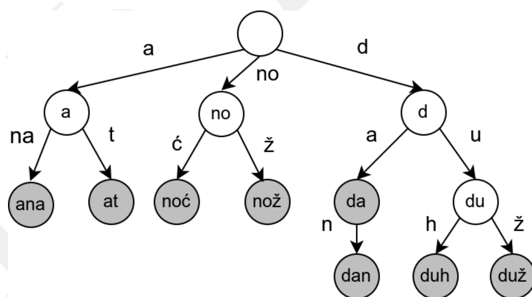


Слика 1.2: Пример префиксног дрвета.

МАТФ Београд

Пример префиксног дрвета дат је на слици 1.2. Кључеви које ово префиксно дрво чува су исти они које чува уређено бинарно дрво са слике 1.1: ana, at, noć, nož, da, dan, duh *и* duž *(без придружених података). Већина кључева које ово префиксно дрво чува се завршава у неком од листова. Међутим, то у општем случају не мора да важи: кључ* da *се не завршава у листу. Стога је потребно да сваки чвор префиксног дрвета чува и информацију о томе да ли се њиме комплетира неки кључ или не (што се лако имплементира додавањем одговарајуће логичке променљиве у чвор дрвета). Чворови којима се комплетира неки кључ су обојени. Илустрације ради, у чворовима су приказане ознаке акумулиране до тих чворова. Треба имати у виду да се оне не чувају експлицитно у чвору.*

Мана префиксног дрвета може бити то што уз податке које чува заузима и пуно додатне меморије (за чување показивача) и стога је пожељно компримовати га. Ако неки чвор има само једног потомка и не представља крај неког кључа, грана до њега и грана од њега се могу спојити у једну, њихови карактери надовезати, а чвор елиминисати. На овај начин се добија компактнија репрезентација префиксног дрвета, позната и под називом *радикс дрво* (енгл. radix tree), код које сваки унутрашњи чвор има бар два детета (слика 1.3). За разлику од регуларног префиксног дрвета, гране радикс дрвета могу бити означене нискама, а не само појединачним карактерима. Предност радикс дрвета је пре свега у томе што су просторно ефикаснија, посебно у случају када се у дрвету чувају дугачке ниске које имају дуге заједничке префиксе.



Слика 1.3: Пример компримованог префиксног дрвета, код кога су гране означене нискама.

Поред тога тога што се префиксним дрветом могу имплементирати опште асоцијативне структуре као скуп и мапа, префиксним дрветом може се имплементирати и коначни речник који омогућава аутоматско комплетирање или проверу исправности, односно аутоматско исправљање речи које корисник уноси на рачунару или мобилном телефону.

Кључеви у префиксном дрвету не морају бити искључиво ниске карактера. На пример, у префиксном дрвету можемо чувати и кључеве који су природни бројеви, при

MATF Београд

чему се тада користи низ цифара у њиховом декадном или бинарном запису. Поред ниски, најчешће се користе бинарне репрезентације бројева фиксне ширине (записаних са фиксним бројем бинарних цифара).

1.1.2 Операције над префиксним дрветом

Операције тражења и уметања елемената у префиксно дрво врше се на прилично очигледан начин, док је у случају брисања некада потребно брисати више од једног чвора.

Испитивање да ли се нека реч налази у префиксном дрвету може се реализовати рекурзивном функцијом која као аргументе прима корен дрвета и реч коју тражи у дрвету (током рекурзивних позива у питању је корен одговарајућег поддрвета префиксног дрвета и неки суфикс речи која се претражује). Ако је реч празна, она се налази у дрвету ако и само ако је корен обележен као крај кључа. Ако реч није празна, да би се она могла налазити у дрвету потребно је да постоји дете корена до ког се стиже преко њеног првог слова и тада претрагу настављамо рекурзивно од тог чвора за суфикс речи без тог првог слова.

И операцију уметања речи у префиксно дрво можемо имплементирати као рекурзивну функцију која као аргументе добија прима дрвета и реч коју треба убацити у то дрво. Ако је реч празна, обележавамо да се у корену налази крај речи. У супротном проверавамо да ли постоји дете корена до ког се стиже првим словом те речи и ако не постоји додајемо га. Затим рекурзивно настављамо уметање од тог чвора и умећемо суфикс речи без тог првог слова.

Брисање речи из префиксног дрвета се такође може имплементирати рекурзивном функцијом која као аргументе прима корен дрвета и реч коју треба обрисати (током рекурзивних позива аргумент ће бити корен одговарајућег поддрвета префиксног дрвета и неки суфикс речи која се брише). Ако је реч празна, означавамо да текући чвор више није крај речи. Ако реч није празна, проверавамо да ли постоји дете корена до ког се стиже првим карактером те речи и ако постоји рекурзивно бришемо суфикс речи без првог слова из дрвета чији је корен то дете. На крају рекурзивне функције, проверавамо да ли текући чвор има деце и ако нема и није крај речи, бришемо и тај чвор из дрвета.

У наставку су дате имплементације основних операција над префиксним дрветом на примеру формирања и претраге скупа речи на енглеском језику. У овом примеру кључеви су речи (ниске карактера) и нису им придружене никакве вредности. Потребно је подржати операцију додавања кључева у структуру података и провере да ли кључ постоји у структури. Чвор префиксног дрвета може се дефинисати тако да садржи низ показивача, чијим елементима одговарају сви могући карактери азбуке из које се формирају кључеви (нпр. пошто се у овом примеру кодирају само речи које се састоје од

Матф Београд

малих слова енглеске абецеде, можемо користити низ показивача дужине 26). Међутим, ефикасније је у сваком чвору чувати информације само о оним карактерима за које постоји грана из тог чвора, односно у ове сврхе искористити мапе. Дакле, у сваком чвору префиксног дрвета чувамо неуређену (хеш) мапу која карактерима придружује гране које крећу из тог чвора ка његовој деци, при чему користимо библиотечку имплементацију хеш-мапе (класу `unordered_map` декларисану у истоименом заглављу). Операције уметања и претраге се могу једноставно имплементирати било рекурзивно било итеративно (у наставку је приказана рекурзивна имплементација).

```
// osnovna struktura čvora prefiksnog drveta - u svakom čvoru čuvamo
// - mapu koja karakterima pridružuje pokazivače ka potomcima
// - informaciju o tome da li je u čvoru kraj neke reči
struct cvor {
  unordered_map<char, cvor*> grane;
  bool krajKljuca = false;
};

// sufiks koji počinje na poziciji i reči w tražimo
// u drvetu na čiji koren ukazuje pokazivač drvo
bool nadji(cvor *drvo, const string& w, int i) {
  // ako je sufiks prazan, on je u korenu akko je u korenu obeleženo
  // da je tu kraj reči
  if (i == w.size())
    return drvo->krajKljuca;

  // tražimo granu na kojoj piše w[i]
  auto it = drvo->grane.find(w[i]);
  // ako je nađemo, rekurzivno tražimo ostatak sufiksa od pozicije i+1
  if (it != drvo->grane.end())
    return nadji(it->second, w, i+1);

  // nismo našli granu sa w[i], pa reč ne postoji
  return false;
}

// tražimo reč w u drvetu na čiji koren ukazuje pokazivač drvo
bool nadji(cvor *drvo, const string& w) {
  return nadji(drvo, w, 0);
}
```

```
// umetanje sufiksa koji počinje na poziciji i reči w
// u drvo na čiji koren ukazuje pokazivač drvo
void umetni(cvor *drvo, const string& w, int i) {
  // ako je sufiks prazan samo u korenu beležimo da je tu kraj reči
  if (i == w.size()) {
    drvo->krajKljuca = true;
    return;
  }

  // tražimo granu na kojoj piše w[i]
  auto it = drvo->grane.find(w[i]);
  // ako takva grana ne postoji, dodajemo je kreirajući novi čvor
  if (it == drvo->grane.end())
    drvo->grane[w[i]] = new cvor();

  // sada znamo da grana sa w[i] sigurno postoji i preko te grane
  // nastavljamo dodavanje sufiksa koji počinje na poziciji i+1
  umetni(drvo->grane[w[i]], w, i+1);
}

// umetanje reči w u drvo na čiji koren ukazuje pokazivač drvo
void umetni(cvor *drvo, string& w) {
  return umetni(drvo, w, 0);
}

// brisanje drveta sa korenom u čvoru drvo
void obrisi(cvor *drvo) {
  if (drvo != nullptr) {
    for (const auto& p : drvo->grane)
      obrisi(p.second);
    delete drvo;
  }
}
```

Када азбука којом се кодирају кључеви има K елемената и када се у сваком чвору чува неуређена мапа грана, сложеност операција претраживања, уметања и брисања елемента из префиксног дрвета је у најгорем случају $O(mK)$, где је m дужина речи која се тражи, умеће или брише. Заиста, сложеност најгорег случаја претраге неуређене мапе је $O(K)$, а приликом обраде кључа дужине m врши се m таквих претрага.

Са друге стране, амортизована сложеност претраге неуређене мапе је $O(1)$, па је амортизована сложеност ових операција $O(m)$. Ако се ради са нискама карактера и ако се неуређена мапа имплементира помоћу низа од K елемената, онда је и сложеност најгорег случаја $O(m)$. Приметимо да је приликом претраге број операција ограничен и дужинама речи које се налазе у дрвету, па се, прецизније, сложеност може ограничити и са $O(\min(m, M))$, где је M дужина најдуже речи која се налази у дрвету.

Добра страна префиксног дрвета је то што сложеност уметања и претраге зависи од дужине записа кључа, а не од броја елемената који се чувају у дрвету. Мана је потреба за чувањем показивача уз сваки чвор у дрвету. Штавише, просторна сложеност префиксног дрвета у најгорем случају износи $O(MNK)$, где је са N означен број кључева који се чувају у префиксном дрвету, а са M максимална дужина кључа. Наиме, максимални могући број чворова префиксног дрвета једнак је $O(MN)$ и одговара случају када нема никаквог преклапања карактера међу кључевима, док је просторна сложеност сваког чвора једнака $O(K)$ због потребе чувања мапе у сваком чвору. Приметимо да је очекивана просторна сложеност мања јер ће се у случају реалне коначне азбуке (на пример, енглеске абецеде) прво преклапање јавити најкасније након 26 кључева.

Смањење меморијске сложености се може постићи и тако што се смањи величина азбуке (по цену повећања дужине кључа). На пример, уместо 256 различитих 8-битних карактера, можемо сваки карактер поделити на две 4-битне половине. Тако се само 16 различитих 4-битних карактера, али се дужина сваког кључа два пута повећава.

Када би се уместо префиксног дрвета користило балансирано уређено бинарно дрво које би чувало комплетне кључеве у чворовима (слика 1.1), временска сложеност операција претраживања, уметања и брисања би у најгорем случају била $O(M \cdot \log N)$, где је са N означен укупан број кључева који се чувају у дрвету, а са M максимална дужина кључа. Просторна сложеност ове структуре је $O(M \cdot N)$.

Када се користи хеш табела, приликом сваке операције уметања и претраге мора да буде израчуната хеш вредност кључа који се тражи за шта је потребно време $O(M)$. У зависности од броја колизија, врше се поређења хеш вредности (у најгорем случају њих $O(N)$, а амортизовано $O(1)$). На крају се кључ који се тражи и кључ слога који је пронађен на основу једнакости хеш-вредности експлицитно пореде, за шта је такође потребно време $O(M)$ (а ако постоје колизије ово поређење се врши неколико пута). Зато је сложеност најгорег случаја операција $O(N + M)$, а амортизована сложеност $O(M)$. Пошто кључ мора бити записан експлицитно у сваком слогу табеле, просторна сложеност је $O(M \cdot N)$.

У табели 1.1 приказана је (амортизована) сложеност разних имплементација скупова/мапа. Претпостављамо да је дужина речи која се обрађује m, максимална дужина речи M, број речи у колекцији N, а величина азбуке K.

MATF Beograd

Табела 1.1: Сложеност различитих имплементација речника

	Хеш табела	Балансирано бинарно уређено дрво	Префиксно дрво
Претрага	$O(m)$	$O(m \log N)$	$O(m)$
Уметање	$O(m)$	$O(m \log N)$	$O(m)$
Брисање	$O(m)$	$O(m \log N)$	$O(m)$
Простор	$O(MN)$	$O(MN)$	$O(MNK)$

Дакле, можемо приметити да префиксно дрво нема лошију сложеност од хеш-табела, али подржава нову операцију (проналажење свих речи које имају дати префикс) и стога се користи приликом решавања проблема у којима је та операција корисна.

Задатак: Пар који даје највећи XOR

Напиши програм који међу унетим неозначеним бројевима одређује онај пар који даје највећи резултат при операцији ексклузивне дисјункције (XOR) њихових бинарних записа.

Опис улаза

Са стандардног улаза се уноси број n ($1 \leq n \leq 100000$), а затим у наредном реду n природних бројева између 0 и 10^{18}.

Опис излаза

На стандардни излаз исписати максималну вредност која се може добити када се ексклузивна дисјункција примени на нека два унета броја.

Пример

Улаз	*Излаз*	*Објашњење*
`5` `1 2 3 4 5`	`7`	Највећи резултат 7 добија се ексклузивном дисјункцијом бројева 3 и 4 (њихови бинарни записи су `00...0000011` и `000...000100`). Исти резултат добија се и ексклузивном дисјункцијом бројева 2 и 5 (њихови бинарни записи су `0000..0000101` и `0000...00000010`).

Решење

Решење грубом силом подразумева да се на сваки пар бројева примени операција XOR и да се испише максимум добијених резултата. Сложеност овог приступа је $O(n^2)$.

Ефикасније решење можемо добити применом напредних структура података. Покушајмо да за сваки нови унети број ефикасно израчунамо највећи број који се може добити применом операције XOR на њега и неки од претходно унетих бројева (у решењу

МАТФ Београд

грубом силом, то се дешава у унутрашњој петљи). Покушајмо да тај број одредимо бит-по-бит и то кренувши од битова највеће тежине. На месту бита највеће тежине можемо добити 1 ако текући број почиње битом 0 и међу раније учитаним бројевима постоји неки који почиње битом 1 или ако текући број почиње битом 1 и међу раније учитаним бројевима постоји неки који почиње битом 0. У супротном, на месту највеће тежине резултата мора бити бит 0. Након одређивања првог бита, одређујемо наредни, али у случају да смо на водеће место резултата уписали 1, међу учитаним нискама задржавамо само оне које су на водећем месту имали бит супротан водећем биту текућег броја (у случају да смо на водеће место резултата уписали 0, тада су сви раније учитани бројеви почињали истим битом којим почиње текући број и сви се задржавају). Поступак сада понављамо за други бит, при чему разматрамо само ниске које нису раније одбачене.

Пример 1.1.3

Претпоставимо да су дати бројеви `1001`*,* `1010`*,* `0110` *и да је текући број* `0010`*.*

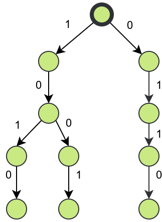
- *На водеће место резултата можемо уписати* `1`*, при чему задржавамо ниске* `1001`*,* `1010`*.*
- *На друго место резултата морамо уписати* `0`*, јер све задржане ниске на месту другог бита имају* `0`*, исто као и текући број. Обе ниске се задржавају.*
- *На треће место резултата можемо уписати* `1` *и тада задржавамо само ниску* `1001`*.*
- *На крају, на последње место резултата можемо уписати* `1`*.*

Коначан резултат је, дакле, `1011` *и он се добија применом операције XOR на ниске* `0010` *и* `1001`*.*

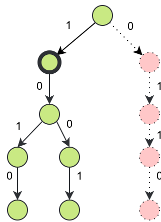
Претрагу можемо веома једноставно организовати ако све учитане бинарне записе један по један уписујемо у префиксно дрво. Приликом обраде текућег броја, његове битове пролазимо слева надесно, за сваки бит се спуштајући на наредни ниво дрвета. На сваком нивоу дрвета, дакле, одређујемо један бит резултата. Ако на текућем нивоу постоји грана обележена битом супротном од текућег бита тренутног броја, спуштамо се њоме и у резултат уписујемо јединицу, док се у супротном спуштамо граном на којој се налази бит једнак текућем биту тренутног броја и у резултат уписујемо нулу. Спуштањем на наредни ниво дрвета уједно елиминишемо све оне ниске које на одговарајућем месту немају потребан бит. Овај поступак је илустрован на наредној слици.

МАТФ Београд

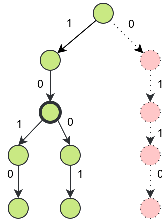
Tekući broj: 0010
Rezultat:
Preostale niske:
1010, 1001, 0110



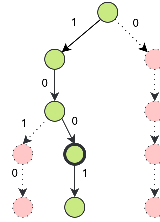
Tekući broj: 0010
Rezultat: 1
Preostale niske:
1010, 1001



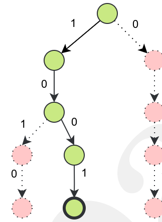
Tekući broj: 0010
Rezultat: 10
Preostale niske:
1010, 1001



Tekući broj: 0010
Rezultat: 101
Preostale niske:
1001



Tekući broj: 0010
Rezultat: 1011
Preostale niske:
1001



Сложеност овог алгоритма је $O(n)$, при чему је константа једнака броју битова којима се записују бројеви (с обзиром на ограничења дата у задатку, то је 64).

```
// cvor binarnog prefiksnog drveta
struct Cvor {
  Cvor* grane[2];
};

// kreiranje novog cvora drveta
Cvor* noviCvor() {
  Cvor* novi = new Cvor();
  novi->grane[0] = novi->grane[1] = nullptr;
  return novi;
}

// ubacivanje broja (njegov binarnog zapisa) u drvo
void ubaci(Cvor* koren, unsigned long long broj) {
  Cvor* cvor = koren;
  unsigned long long mask = 1ull << (8*sizeof(unsigned long long) - 1);
  while (mask != 0) {
    int bit = (broj & mask) != 0;
    if (cvor->grane[bit] == nullptr)
      cvor->grane[bit] = noviCvor();
    cvor = cvor->grane[bit];
    mask >>= 1;
  }
}
```

```
// brisanje drveta
void obrisi(Cvor* koren) {
  if (koren != nullptr) {
    obrisi(koren->grane[0]);
    obrisi(koren->grane[1]);
    delete koren;
  }
}

// maxXOR koji moze da se napravi izmedju elemenata drveta i datog broja
unsigned long long maxXOR(Cvor* koren, ull broj) {
  Cvor* cvor = koren;
  unsigned long long rez = 0;
  // analiziramo sve bitove datog broja
  // maska sadrzi tacno jednu binarnu jedinicu i ona se pomera nalevo
  unsigned long long mask = 1ull << (8*sizeof(unsigned long long) - 1);
  while (mask != 0) {
    // tekuci bit datog broja
    int bit = (broj & mask) != 0;
    if (cvor->grane[!bit] != nullptr) {
      rez = rez | mask;
      cvor = cvor->grane[!bit];
    } else
      cvor = cvor->grane[bit];
    mask >>= 1;
  }
  return rez;
}

int main() {
  // broj brojeva
  int n;
  cin >> n;
  // prefiksno drvo
  Cvor* koren = noviCvor();

  // ubacujemo prvi broj
  unsigned long long x;
  cin >> x;
```

```
  ubaci(koren, x);
  unsigned long long max = 0;
  // analiziramo naredne brojeve
  for (int i = 1; i < n; i++) {
    cin >> x;
    // trazimo maksimalni XOR koji se moze dobiti od broja x
    // i brojeva koji se do sada nalaze u drvetu
    unsigned long long rez = maxXOR(koren, x);
    if (rez > max)
      max = rez;
    ubaci(koren, x);
  }
  cout << max << endl;

  // brisemo drvo
  obrisi(koren);
  return 0;
}
```

1.2 Фибоначијев хип

Редови са приоритетом се обично имплементирају коришћењем структуре података хип. За разлику од бинарног хипа у ком операција уметања елемента има логаритамску сложеност, Фибоначијев[3] хип је структура података код које операција уметања новог елемента има константну амортизовану сложеност, што је чини бржом. Додатно, Фибоначијев хип омогућава и спајање два хипа у један као и смањивање вредности кључа у константној амортизованој сложености, док сложеност избацивања минимума остаје логаритамска. Дакле, Фибоначијев хип успешно решава следећи проблем.

Проблем

Дефинисати структуру података која подржава ефикасно извршавање (у амортизованом константном или логаритамском времену) следећих операција:

- `napravi_hip()` - *прави нови празни хип*
- `umetni(H, v)` - *умеће елемент са вредношћу кључа* `v` *у хип* `H`
- `minimum(H)` - *враћа елемент хипа* `H` *са минималном вредношћу кључа*

[3] Леонардо Пизано Фибоначи (ит. Leonardo Bonacci), (око 1170-1240), италијански математичар.

- `izbaci_minimum(H)` - *брише елемент хипа* `H` *са минималном вредношћу кључа и враћа га као резултат*
- `unija(H1,H2)` - *од два хипа* `H1` *и* `H2` *прави нови хип који садржи све елементе полазних хипова*
- `smanji_kljuc(H, x, v)` - *елементу* `x` *хипа* `H` *се тренутна вредност кључа смањује на нову дату вредност* `v`

Имплементација операције `smanji_kljuc` захтева памћење додатних података у структури, па је имплементација хипа који подржава ту операцију мало компликованија него имплементација хипа који подржава све остале наведене операције.

Амортизована сложеност операција прављења хипа, уметања елемента, одређивања минимума, прављења уније и смањивања вредности кључа је $O(1)$, док је амортизована сложеност операције брисања минималног елемента из хипа $O(\log n)$. Дакле, операције уметања елемента и прављења уније се ефикасније извршавају над Фибоначијевим хипом, него над класичним бинарним хипом (подсетимо се, сложеност додавања елемената у бинарни хип је $O(\log n)$).

Фибоначијев хип је користан када је број операција брисања минималног елемента из хипа мали у односу на остале поменуте операције. На пример, у графовским алгоритмима као што је одређивање разапињућег (повезујућег) дрвета минималне цене (види поглавље 2.10) или најкраћих путева из задатог чвора (види поглавље 2.9), ако је граф густ (садржи велики број грана), операција смањивања вредности кључа се може често јављати, те убрзање са $O(\log n)$ код бинарног хипа на $O(1)$ код Фибоначијевог хипа може донети осетно убрзање. Недостатак Фибоначијевог хипа је то што је доста тежи за имплементацију од класичног, бинарног хипа, то што захтева више меморије, као и то што је сложеност најгорег случаја неких операција велика.

Фибоначијев хип је добио назив према Фибоначијевим бројевима на основу којих се формулише централна инваријанта која гарантује добру сложеност операција. Ову структуру података осмислили су Мајкл Фредман[4] и Роберт Тарџан[5] са циљем да се побољша време извршавања Дајкстриног алгоритма за одређивање најкраћих путева из задатог чвора (види поглавље 2.9.2). Она има примене и у другим графовским алгоритмима, попут Примовог алгоритма за рачунање минималног повезујућег дрвета (види поглавље 2.10.2), за одређивање максималног тока кроз мрежу, али и у другим доменима, попут геометријских алгоритама, обраде слика и математичке оптимизације.

[4]Мајкл Фредман (енгл. Michael Fredman), амерички информатичар.

[5]Роберт Тарџан (енгл. Robert Tarjan), рођен 1948. године, амерички информатичар.

МАТФ Београд

1.2.1 Структура хипа

Фибоначијев мин-хип представља колекцију мин-хипова. Сваки мих-хип је дрво у ком чворови могу имати различит број наследника, а вредност у сваком чвору је мања или једнака вредности у његовим наследницима. Редослед дрвета у Фибоначијевом хипу је произвољан. Фибоначијевом хипу приступамо путем показивача на корен дрвета са минималном вредношћу у целом Фибоначијевом хипу – овај чвор зовемо *минималним чвором* Фибоначијевог хипа (ако постоји више чворова са минималном вредношћу било који од њих се може прогласити минималним чвором). Један пример Фибоначијевог мин-хипа приказан је на слици 1.5. Корени свих дрвета у Фибоначијевом хипу се повезују у кружну, двоструко повезану листу коју називамо *листа коренова* хипа. И сва деца било ког чвора су међусобно повезана у кружну, двоструко повезану листу. Сваки чвор (било да је корен или не) садржи показивач на левог и десног суседа и на неко дете (а ако је потребно вршити и операције смањивања вредности кључева, онда и показивач на родитеља и још неке помоћне податке које ћемо описати у склопу описа те операције). Коришћење кружних листа омогућава уметање новог чвора у листу и брисање чвора на који имамо показивач (самим тим и минималног чвора) у времену $O(1)$. Такође, две овакве листе можемо објединити у нову листу у времену $O(1)$ (видећемо да је ово битно за ефикасно извођење операције формирања уније). Ефикасност ових операција над листама гарантује ефикасно извршавање операција над хипом.

Број деце чвора назива се *степен чвора*. Наведимо сада кључну инваријанту (услов који важи након формирања структуре и након примене било које операције и за који ћемо видети да гарантује сложеност операција).

Инваријанта (број чворова поддрвета): Свако поддрво Фибоначијевог хипа чији је корен степена d садржи бар F_{d+2} чворова, где је са F_k означен k-ти Фибоначијев број ($F_0 = 0$, $F_1 = 1$, $F_k = F_{k-1} + F_{k-2}$).

Дакле, дрво чији је корен степена 0 има бар 1 чвор, степена 1 има бар 2 чвора, степена 2 има бар 3 чвора, степена 3 има бар 5 чворова, степена 4 има бар 8 чворова итд.. Пошто се лако (индукцијом) доказује да је $F_{d+2} \geq \varphi^d$, где је $\varphi = \frac{1+\sqrt{5}}{2}$, за степен d сваког чвора који је корен неког поддрвета у ком се налази m чворова важи $m \geq F_{d+2} \geq \varphi^d$ тј. $d \leq \log_\varphi m$. Дакле, број деце сваког чвора који је корен поддрвета са m елемената је $O(\log m)$. Ако у целом Фибоначијевом хипу има n елемената, степен било ког чвора је $O(\log n)$. Дакле, под условом да инваријанта важи, листе деце су релативно кратке и пролазак кроз све елементе такве листе врши се у логаритамској сложености (у односу на укупан број елемената у хипу).

Формулишимо сада другачију инваријанту, коју је лакше проверавати, а која даје гаранције да ће важити инваријанта о броју чворова у поддрветима.

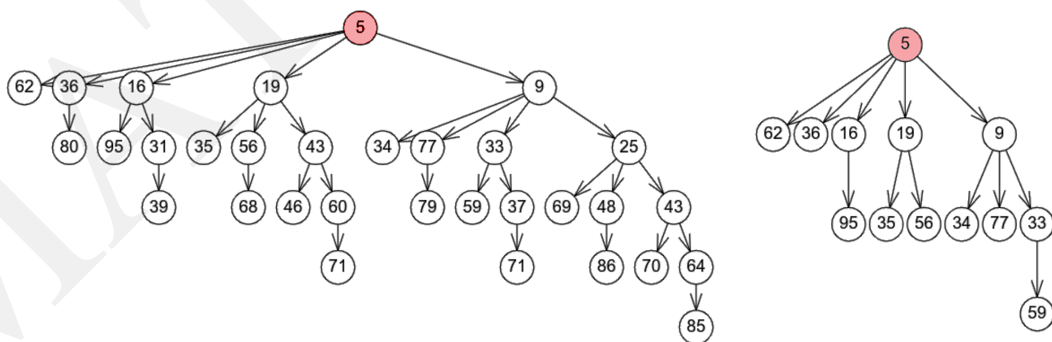
Матф Београд

Инваријанта (степени деце): За сваки чвор x степена d важи да су степени деце бар $0, 0, 1, 2, 3, \ldots, d-2$.

Индукцијом доказујемо да инваријанта о степенима деце гарантује инваријанту о броју чворова поддрвета. Ако је $d = 0$, чвор нема деце, а у његовом поддрвету се налази $F_2 = 1$ чвор. Ако је инваријанта о степенима деце испуњена, на основу индуктивне хипотезе важи да поддрвета чији су корени деца чвора x садрже редом $F_2, F_2, F_3, \ldots F_d$ чворова, па, пошто је $F_2 = F_1 = 1$ и $F_0 = 0$, важи да је укупан број чворова у дрвету чији је корен x једнак $1 + F_0 + F_1 + F_2 + F_3 \ldots + F_d$ (прва јединица долази од самог корена x). Индукцијом се рутински може доказати да је $1 + F_0 + F_1 + F_2 + \ldots + F_d = F_{d+2}$, па инваријанта о степенима деце заиста гарантује инваријанту о броју чворова поддрвета.

Ова инваријанта намеће одређене степене деце, али није прецизирано која деца треба да имају тражене степене. Деца се разматрају у редоследу њиховог додавања родитељском чвору и њихови степени у том редоследу задовољавају инваријанту (а не по редоследу којим су смештени у листу деце). Дакле, услов који је довољан да би хип био Фибоначијев је да су степени деце у редоследу додавања $0, 0, 1, 2, 3, \ldots$ тј. да за сваки чвор важи да његово дете са редним бројем додавања i (бројимо од нуле) има степен $d_i \geq i - 1$ (при чему је и $d_i \geq 0$, јер је d_i природан број).

Видећемо ускоро да ће процедура формирања хипа бити таква да су степени деце сваког чвора дрвета непосредно након његовог првог формирања $0, 1, 2, 3, \ldots$ (што је и више него што се захтева инваријантом о степенима деце). На основу овога се може доказати да свако поддрво чији је степен корена d има 2^d чворова, што такође гарантује повољну сложеност операција. Ипак, инваријанта је ослабљена да би било могуће уклањати чворове из формираног дрвета, што је, видећемо, веома важно за реализацију операције смањивања вредности кључа (`smanji_kljuc`).



Слика 1.4: Дрво у ком има $2^5 = 32$ чвора (степени чворова су редом, $0, 1, 2, 3, \ldots$). Дрво у ком има $F_7 = 13$ чворова (степени чворова су редом $0, 0, 1, 2, 3, \ldots$).

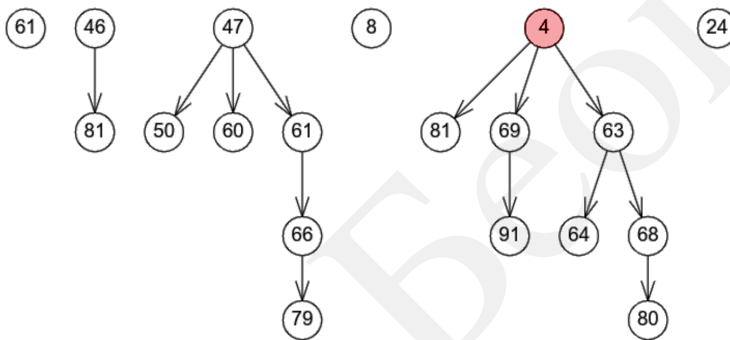
МАТФ Београд

На слици 1.4 лево приказано је најгушће могуће попуњено дрво (које се добија одмах након формирања дрвета које садржи $2^5 = 32$ чвора), док се десно види најређе могуће дрво које задовољава инваријанту и које садржи $F_{5+2} = 13$ чворова. Избацивањем било ког чвора из десног дрвета била би нарушена инваријанта.

Прикажимо, напокон, један комплетан пример Фибоначијевог хипа.

Пример 1.2.1

На слици 1.5 приказан је Фибоначијев хип који се састоји од 6 *мин-хипова. Минимални елемент у хипу је минимум свих коренова, а то је елемент са кључем* 4.



Слика 1.5: Пример Фибоначијевог хипа.

Директном провером се лако може показати инваријанта о степенима деце. Она је увек тривијално испуњена за чворове чији је степен 0, 1 *и* 2 *(јер су степени деце увек већи или једнаки од 0 што се инваријантом тражи). Деца чвора* 47 *имају редом степене* 0, 0 *и* 1, *а деца чвора* 4 *имају редом редом степене* 0, 1 *и* 2. *Дакле, за сваки чвор јесте задовољена инваријанта о степенима деце (степени деце су на неким местима већи него што је потребно), а она гарантује и инваријанту о величинама тј. укупном броју чворова тих дрвета. Заиста, дрвета која чине Фибоначијев хип редом имају* 1, 2, 6, 1, 8, 1 *и* 1 *чвор, што је понекад и више од одговарајућих Фибоначијевих бројева* 1, 2, 5, 1, 5, 1 *и* 1. *Исто важи и за сва њихова поддрвета, па је централна инваријанта о величинама свих дрвета задовољена.*

1.2.2 Операције над хипом

Једна од основних карактеристика операција над Фибоначијевим хипом је лењо извршавање операција, тј. посао се одлаже за што је могуће касније.

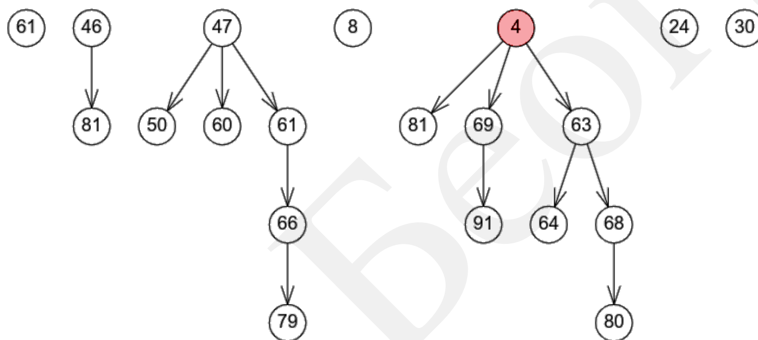
МАТФ Београд

1.2.2.1 Уметање

Уметање новог елемента у Фибоначијев хип (операција `insert(H, x)`) се врши тако што се нови елемент додаје у листу коренова, ажурирајући минимални чвор, ако је потребно, што је константне временске сложености. Дакле, уметањем k елемената у празан Фибоначијев хип добија се Фибоначијев хип чија листа коренова има k елемената.

Пример 1.2.2

Уметањем елемента са кључем 30 у претходни Фибоначијев хип добија се хип приказан на слици 1.6 (нови елемент је додат на крај листе коренова, али могао је бити додат и на било које друго место, најчешће је то непосредно иза или испред минималног чвора, на који, подсетимо се, указује посебан показивач).



Слика 1.6: Уметање елемента 30.

1.2.2.2 Унија два хипа

Унија два Фибоначијева хипа H_1 и H_2 (операција `unija(H1, H2)`) се прави надовезивањем листи коренова ова два хипа и одређивањем новог минималног чвора (то је мањи од два минимална чвора полазних хипова). И ово је операција константне сложености.

1.2.2.3 Одређивање минимума

Одређивање минималног елемента у Фибоначијевом хипу (операција `minimum(H)`) је тривијално, с обзиром на то да се у сваком тренутку чува показивач на минимални елемент у структури. Ово је такође операција константне сложености.

1.2.2.4 Брисање најмањег елемента

Операција `izbaci_minimum(H)`, којом се брише најмањи елемент из Фибоначијевог хипа почиње тако што се деца минималног елемента умећу у листу коренова, и он се брише из листе коренова. Да би се ажурирао минимум хипа потребно је потенцијално проћи кроз целу листу коренова, која може бити веома дуга. Зато се пре ажурирања минимума

МАТФ Београд

врши помоћна операција која се назива *консолидација* хипа. Њоме се хип доводи у стање у ком сви коренови имају различит степен, чиме се њихов број значајно смањује (на основу раније показане везе између степена коренова и величине дрвета, јасно је да су сви степени коренова $O(\log n)$, па их не може бити више од $O(\log n)$).

Консолидација се врши на следећи начин: све док нека два чвора из листе коренова имају исти степен врши се спајање коренова истог степена тј. ради се следеће:

1. проналазе се два чвора x и y у листи коренова која су истог степена (без смањења општости нека важи да је вредност кључа чвора x мања или једнака од вредности кључа чвора y)

2. Чвор y се избацује из листе коренова и проглашава се дететом чвора x. На овај начин, дрво са кореном x остаје хип и степен чвора x се повећава за један. На основу инваријанте о степенима деце знамо да важи да је последње додати наследник чвора x имао бар степен $d-2$, где је d степен чвора x. То значи да би наредни тј. последњи додати наследник, што је управо y, морао да има степен бар $d-1$. Међутим, ми знамо да он има степен d (јер му је степен једнак степену чвора x), тако да је инваријанта задовољена, уз један додатни степен слободе. Наиме, чак и када уклонимо једног наследника чвора y, инваријанта остаје задовољена. Са друге стране, уклањање два наследника би нарушило инваријанту и то не смемо да радимо. Ово је значајно за операцију `smanji_kljuc`, којом се смањује вредности кључа елемената хипа и вратићемо се на ово приликом описа те операције.

Нагласимо да је избор пара чворова са истим степенима у сваком кораку произвољан (консолидација се не мора вршити ни у каквом посебном редоследу парова чворова).

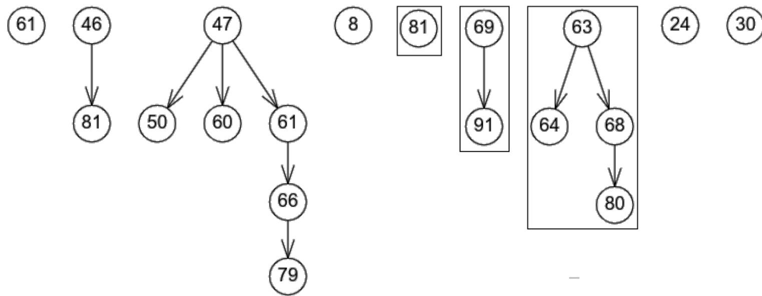
Обично се за реализовање консолидације користи помоћни низ у који се смештају показивачи на чворове из листе коренова, тако да се на позицији i чува показивач на неки чвор из листе коренова који је степена i. Димензија овог помоћног низа једнака је максималном степену $D(n)$ произвољног чвора у Фибоначијевом хипу – већ смо објаснили да је он једнак $O(\log n)$, где је са n означен број елемената хипа. Коренови који су истог степена се детектују на следећи начин: пролази се редом кроз листу коренова и ако је текући корен степена i, а i-ти елемент низа је још увек празан, он се иницијализује показивачем на дати корен, а ако је i-ти елемент већ постављен, пронађена су два корена истог степена и они се спајају.

Пример 1.2.3

Погледајмо како се мења претходни хип након извођења операције брисања елемента са минималном вредношћу кључа.

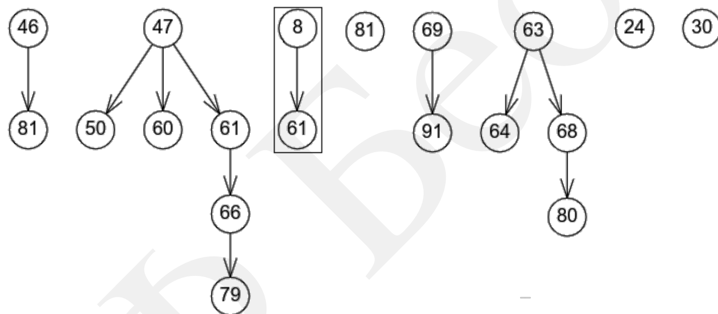
У првом кораку се сва деца уклоњеног минималног чвора 4 *умећу у низ коренова.*

МАТФ Београд

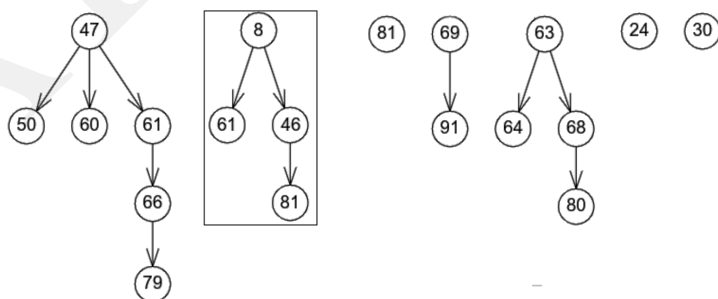


Након тога извршавамо консолидацију (спајање дрвета чији су корени истог степена).

- *Прво спајамо чворове* 8 *и* 61*, који имају степен* 0.

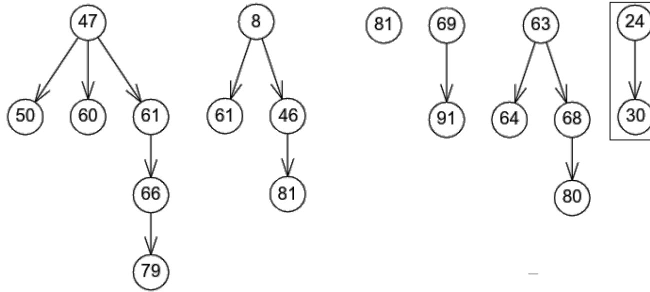


- *Корен добијеног дрвета има степен* 1*, оно се спаја са дрветом чији је корен* 46*, који такође има степен* 1.

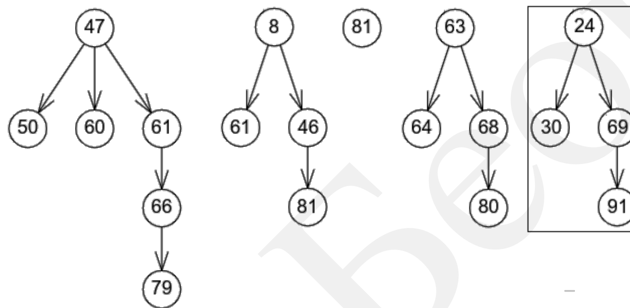


- *Након тога спајају се дрвета чији су корени* 24 *и* 30 *(оба су степена* 0*).*

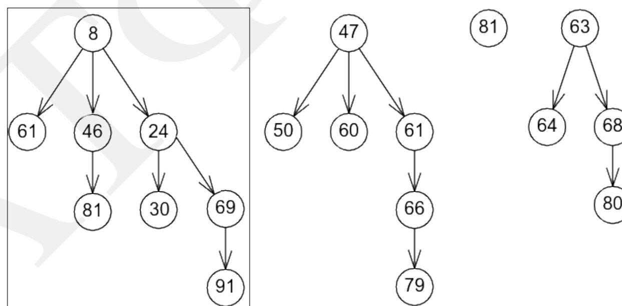
МАТФ Београд



- *Добијено дрво се спаја са дрветом чији је корен* 69 *(оба корена имају степен* 1*).*

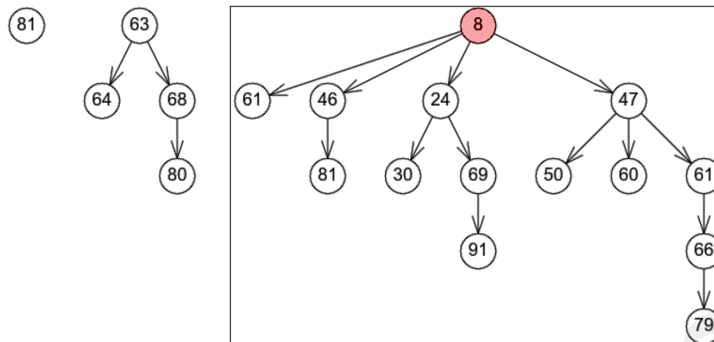


- *Добијено дрво се спаја са дрветом чији је корен* 8 *(оба дрвета имају степен* 2*).*



- *На крају се спаја добијено дрво са дрветом чији је корен* 47 *(оба корена имају степен* 3*). Након тог спајања, сва дрвета имају различите степене коренова (степени су* 0, 2 *и* 4*). Тиме се консолидација завршава, гарантовано је да је број дрвета* $O(\log n)$ *и затим се нови минимални чвор може ефикасно одредити (то је чвор* 8*).*

МАТФ Београд



Анализирајмо сложеност претходно описаног поступка. Деца се ефикасно, алгоритмом сложености $O(1)$ могу уметнути у листу коренова (ради се о спајању две листе). Након тога се врши консолидација, која у најгорем случају може бити и линеарне сложености (најгори случај је када се n једночланих дрвета спаја). Ипак, може се показати да је амортизована сложеност консолидације $O(\log n)$ – неформално, када се једном изврши консолидација и добије се мали број великих дрвета, сваки наредни корак избацивања најмањег елемента и консолидације биће прилично ефикасан. На крају се проналази нови минимални чвор, обрадом свих коренова. Њих након консолидације може бити највише $O(\log n)$, па је ово ефикасна операција.

Прикажимо сада једну једноставну имплементацију ових операција (једноставности ради, користимо глобалне промењиве и не водимо рачуна о успешности алокације, као ни о деалокацији меморије).

```
// Cvor drveta sadrzi podatak i dvostruko povezanu listu dece
struct Cvor {
  int podatak;
  list<Cvor*> deca;
};

// dvostruko povezana lista korenova
list<Cvor*> hip;
// pokazivac (iterator) koji ukazuje na najmanji cvor
list<Cvor*>::iterator minCvor;

// umetanje novog elementa u hip
void umetni(int podatak) {
  // pravimo novi cvor i upisujemo podatak u njega
  Cvor* novi = new Cvor();
```

```
  novi->podatak = podatak;

  if (hip.empty()) {
    // ako je hip prazan ubacujemo novi cvor u njega i taj novi cvor
    // je minimalan
    hip.push_back(novi);
    minCvor = hip.begin();
  } else {
    // ako hip nije prazana ubacujemo novi cvor u njega
    hip.push_front(novi);
    // minimum azuriramo samo ako je novi podatak manji od dotadasnjeg
    // minimalnog
    if (novi->podatak < (*minCvor)->podatak)
      minCvor = hip.begin();
  }
}

// najmanji element u hipu
int minimum() {
  return (*minCvor)->podatak;
}

// konsolidacija hipa (pomocna operacija kojom se skracuje lista korenova)
void konsoliduj() {
  // za svaki moguci broj dece pamtimo jedan koren koji ima toliko dece
  // u njemu ce se nalaziti svi koreni novog hipa (i svi koreni
  // ce imati razlicit broj dece)
  const int MAKS_DECE = 64;
  Cvor* pom[MAKS_DECE] = {};
  // obradjujemo jedan po jedan koren hipa
  for (Cvor* x : hip) {
    // dok postoji drugi koren (u novom hipu) sa istim brojem dece kao x
    int brojDece = x->deca.size();
    while (pom[brojDece] != nullptr) {
      // spajamo dva korena tako da je manji podatak iznad
      Cvor* y = pom[brojDece];
      if (x->podatak > y->podatak)
        swap(x, y);
      x->deca.push_back(y);
```

```
      // kada mu se pridruzi y, koren x ima jedno dete vise, a u novom
      // hipu vise ne postoji hip koji ima stari broj dece
      pom[brojDece] = nullptr;
      brojDece++;
    }
    // upisujemo x u novi hip
    pom[brojDece] = x;
  }
  // prebacujemo sve korene iz novog hipa u stari
  hip.clear();
  for (Cvor* cvor : pom)
    if (cvor != nullptr)
      hip.push_back(cvor);
}
```

1.2.2.5 Смањивање вредности кључа

Опишимо сада операцију смањивање вредности кључа чвору x (операцију `smanji_kljuc(H, x, v)`). Претпостављамо да је чвор x коме се вредност умањује већ познат тј. да је познат показивач на њега (јер је то случај у многим алгоритмима који користе Фибоначијеве хипове). Ако није, тј. ако претпостављамо да је позната само вредност кључа која се умањује, али не и чвор који садржи ту вредност (то је операција `smanji_vrednost(H, k, v)`) онда се чвор k са вредношћу кључа k може ефикасно пронаћи тако што се чува мапа која слика кључеве у показиваче на чворове (довољно је да се вредност кључа слика у показивач на било који чвор који је садржи).

Да би се ова операција могла ефикасно имплементирати, потребно је чворове проширити са неколико додатних података, које ћемо у тексту описати.

Смањивање вредности кључа се изводи на следећи начин: прво се датом чвору x промени вредност кључа на v, а затим се, ако x није корен, та вредност пореди са вредношћу кључа родитеља. Како би се могло приступити родитељу, у свим чворовима се чува информација о родитељском чвору (она не мора бити дефинисана једино у коренима дрвета). Ако је x корен или је вредност кључа родитеља чвора x мања или једнака v, онда услов хипа није нарушен те нису потребне никакве даље измене. Ако то није случај, услов хипа је нарушен те је хип потребно преуредити. Најпре се врши *одсецање* гране између чвора x и његовог родитеља и поддрво са кореном x се додаје у листу коренова.

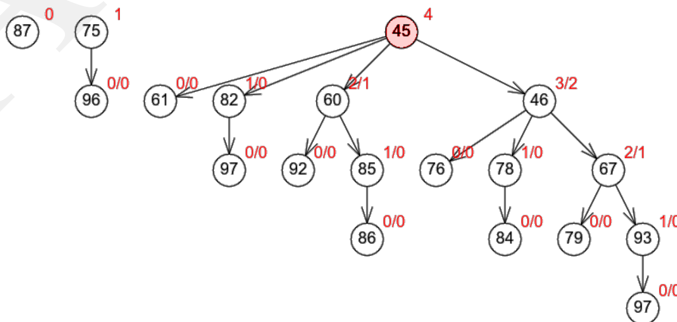
Поставља се питање да ли је дрво из којег је исечен чвор x потребно даље преуређивати. Одговор зависи од тога да ли је избацивањем поддрвета са кореном x нарушена инваријанта о броју чворова у том дрвету, тј. да ли је нарушена инваријанта о степенима

деце. Подсетимо се да су након консолидације чворови у дрвету обично такви да имају једно дете више од онога што би требало да имају на основу инваријанте тј. да им је могуће уклонити једно дете (тј. поддрво којем је оно корен), а да инваријанта за његовог родитеља и даље буде задовољена. Са друге стране, уклањање два детета нарушава инваријанту. Дакле, поставља се питање да ли смо родитељу чвора x већ уклонили неко дете од тренутка када је убачен у тренутно дрво. Да бисмо то знали, сваки чвор поред информације о свом степену (броју своје деце), садржи и *ознаку* да ли је „изгубио” дете од последњег тренутка када је постао дете неког другог чвора. Чворови се иницијално не означавају и сваки пут када чвор постане дете неког другог чвора или корен, уколико је био означен, ознака се уклања. Ако један чвор изгуби два детета, одсецамо и родитеља и поступак се наставља по истом принципу, навише, уз дрво. Прецизније, врши се следећи низ корака:

- Смањује се вредност кључа чвора x.
- Ако x није корен и ако је нова вредност v мања од вредности родитеља чвора x, одсеца се цело дрво чији је x корен, чвор x се додаје се у листу коренова (чиме то поддрво постаје самостално дрво) и уклања се ознака (уколико је чвор x био означен).
  - Ако је родитељ p чвора x неозначен (није му до сада било одсечено ниједно дете), он се означава.
  - У противном, одсеца се цело дрво чији је корен родитељски чвор p, чвор p се додаје у листу коренова (чиме и то поддрво постаје самостално дрво) и са њега се уклања ознака.
  - Рекурзивно се понавља претходни корак за све родитеље који нису већ коренови и којима су два пута одсечена деца.

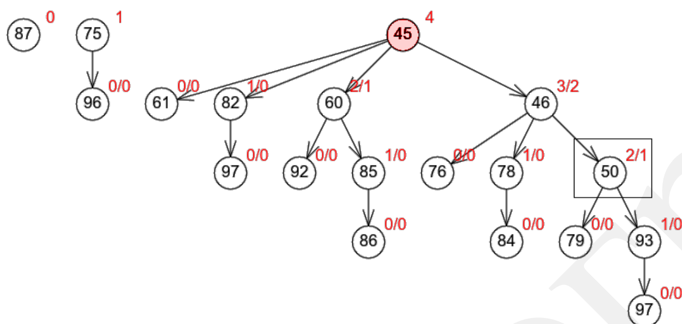
Пример 1.2.4

Размотримо хип приказан на наредној слици. Горе-десно од свакоī чвора написан је тренутни стеӣен чвора и минимални стеӣен који чвор мора имати на основу инваријанте. На почетку скоро сви чворови имају и већи стеӣен неīо што је то потребно.

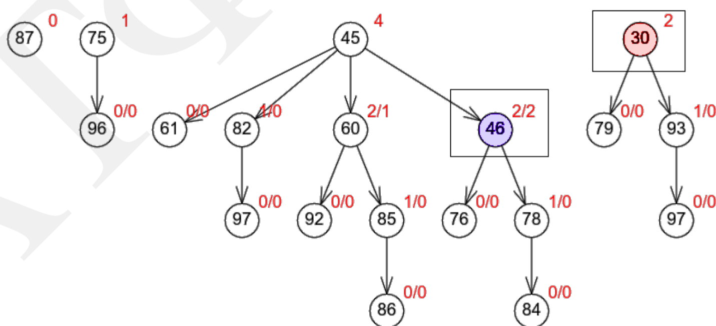


МАТФ Београд

- *Претпоставимо да желимо да смањимо вредност кључа са* 67 *на* 50. *Пошто је након тог смањивања вредност и даље већа од вредности у родитељском чвору, нема потребе вршити даље измене.*

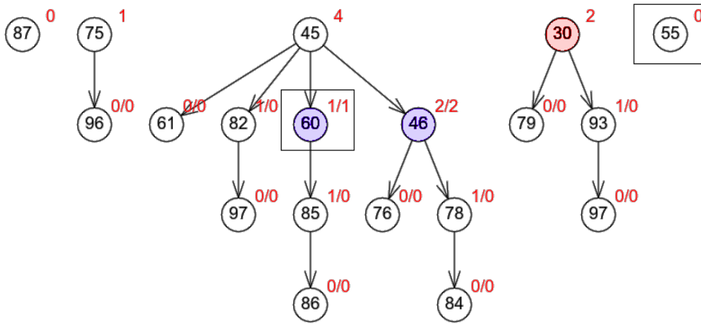


- *Даљим смањивањем вредности кључа* 50 *на* 30 *нарушава се однос вредности у том и родитељском чвору, па се чвор одсеца и ставља у низ коренова. Минимални чвор постаје* 30, *пошто је* 30 *мање од* 45. *Пошто је родитељски чвор* 46 *неозначен, нема потребе даље поправљати дрво након одсецања, већ се само родитељски чвор* 46 *означава. Приметимо да је његов степен у овом тренутку једнак минималној вредности степена на основу инваријанте и њему није могуће даље одсецати децу, а да инваријанта не буде нарушена. Добија се хип приказан на следећој слици.*

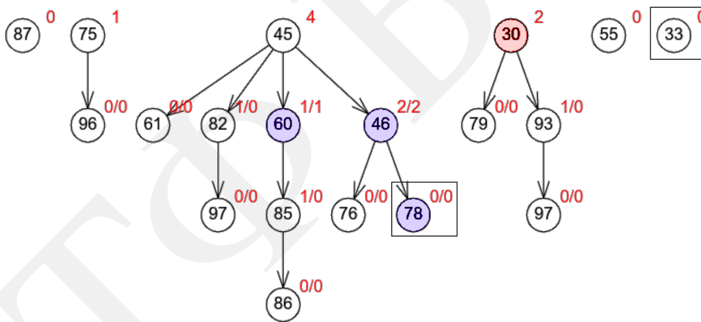


- *Смањимо сада вредност кључу* 92 *на* 55. *Чвор се одсеца и ставља у низ коренова. Пошто је код њему родитељског чвора* 60 *постојао један степен слободе (степен тог чвора је за* 1 *већи од минималне вредности) тј. пошто* 60 *није означен чвор, нема потребе даље модификовати дрво.*

МАТФ Београд

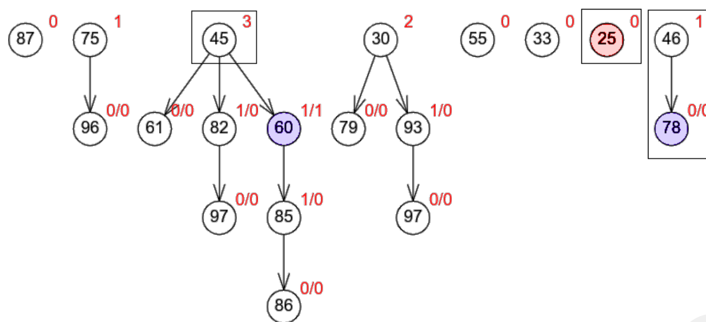


- *Смањимо сада вредност кључа* 84 *на* 33. *Он се одсеца и ставља у низ коренова. Пошто је код и њему родитељског чвора* 78 *постојао један степен слободе тј. пошто* 78 *није означен чвор, нема потребе даље модификовати дрво.*



- *На крају, смањимо вредност кључа* 78 *на* 25. *Он се одсеца и ставља у низ коренова. Међутим, овај пут имамо ситуацију да је његов родитљски чвор* 46 *означен, што значи да је достигао минимални степен, па се додатним одсецањем његовог детета добија степен мањи од минималног. Зато је потребно одсећи и цео родитељски чвор* 46. *Његов родитељски чвор је корен* 45, *па нема потребе за даљим модификацијама и добија се хип приказан на следећој слици (корене нема потребе означавати, јер њихов степен може бити произвољан тј. инваријанта не захтева било какву минималну вредност за степен корена).*

МАТФ Београд



1.3 Структуре података за представљање дисјунктних скупова

Понекад је у програму потребно одржавати неколико дисјунктних скупова (често подскупова неког скупа), при чему је потребно умети за дати елемент ефикасно пронаћи ком скупу припада (ту операцију зовемо `find` тј. `pronadji`) и ефикасно спојити два задата скупа у нови, већи скуп (ту операцију зовемо `union` тј. `unija`). Претпоставићемо да је сваки скуп једнозначно одређен неком ознаком (то може бити редни број скупа, назив скупа или неки канонски представник скупа). Аргументи операције `unija` не морају бити ознаке скупова чију унију треба креирати, већ могу бити произвољни елементи тих скупова. Приликом извођења уније полазни скупови се уклањају из колекције, и у колекцију се додаје њихова унија. Дакле, решавамо следећи проблем.

Проблем

Дефинисати структуру података која омогућава следећи интерфејс:

- `pronadji(x)` – *враћа ознаку скупа ком припада елемент* `x` *(та ознака може бити или неки идентификатор или неки канонски представник тог скупа);*
- `unija(x, y)` – *спаја скуп који садржи елемент* `x` *и скуп који садржи елемент* `y`.

Структура података која подржава овакав интерфејс се често назива по енглеском називу *union find* или *DSU* (енгл. disjoint set union), при чему се под тим именом обично подразумева и специфична, ефикасна имплементација (пре ње, у наставку ћемо приказати и једноставнију, али неефикаснију имплементацију).

Помоћу операције `pronadji` лако можемо за два елемента проверити да ли припадају истом скупу тако што за сваки од њих пронађемо ознаку скупа ком припада и проверимо да ли су ове ознаке једнаке.

Приметимо да је оваква структура података корисна када год радимо са неким релацијама еквиваленције и када је потребно представити класе еквиваленције (које су дис-

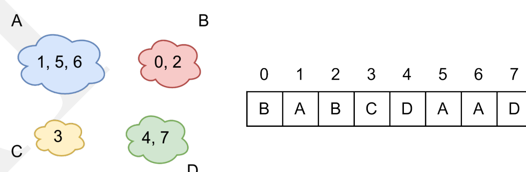
МАТФ Београд

јунктни подскупови скупа на ком је релација дефинисана). Провера да ли су два елемента у релацији се заснива на провери да ли припадају истој класи, а успостављање релације између било која два елемента доводи до спајања њихових класа еквиваленције. Често се употребљава за анализу повезаности неких елемената. На пример, корисници друштвених мрежа се могу груписати у класе еквиваленције на основу својих познанстава са другим корисницима (ако особа A познаје особу B, а особа B познаје особу C сматраћемо и да се особе A и C посредно познају) и коришћењем овакве структуре података можемо лако утврдити да ли су сви корисници међусобно повезани (преко заједничких познаника).

Структуре података за рад са дисјунктним скуповима имају различите примене. Једна од најзначајнијих је одржавање компоненти повезаности у Краскеловом алгоритму за конструкцију минималног повезујућег дрвета у графу (видети поглавље 2.10.3). Могу се користити и за сегментацију слика, тј. партиционисање слике на већи број дисјунктних региона, тако да су сви пиксели истог региона слични у погледу неког својства као што је боја, интензитет или текстура.

1.3.1 Наивна имплементација

Једна могућа имплементација структуре података са оваквим интерфејсом подразумева да се одржава пресликавање сваког елемента у ознаку скупа којем припада. Ако претпоставимо да разматрамо скуп од n елемената и да су сви елементи нумерисани бројевима од 0 до $n-1$, онда ово пресликавање можемо реализовати помоћу обичног низа где се на позицији сваког елемента налази ознака скупа којем он припада (уколико елементи нису нумерисани бројевима, можемо уместо низа да користимо мапу којом се ознаке елемената пресликавају у ознаке скупа ком елемент припада). Пример такве репрезентације приказан је на слици 1.7.



Слика 1.7: Представљање скупова обичним низом.

Операција `pronadji` је тада тривијална: довољно је из низа прочитати ознаку скупа ком елемент припада. Сложеност операције `pronadji` је тада $O(1)$.

Операција `unija` је много спорија јер подразумева да се ознаке свих елемената једног скупа мењају у ознаке другог, што захтева да се прође кроз цео низ. Сложеност операције `unija` је тада $\Theta(n)$.

МАТФ Београд

Да бисмо одредили амортизовану сложеност (што је веома релевантно код структура података код којих се одређене операције понављају пуно пута), потребно је да проценимо потребно време извршавања m операција (од којих је свака типа `unija` или `pronadji`). Операција `pronadji` се извршава у времену $O(1)$, али је операција `unija` сложености $O(n)$, где је n број елемената који одржавамо. Најгори случај наступа када стално вршимо уније, па време извршавања низа од m операција можемо да оценимо као $O(m \cdot n)$. Максимални број извршавања уније различитих скупова је $n - 1$, јер ће се након тога сви елементи објединити у исти скуп, па је и за велике вредности m укупно време свих тих операција $O(n^2)$.

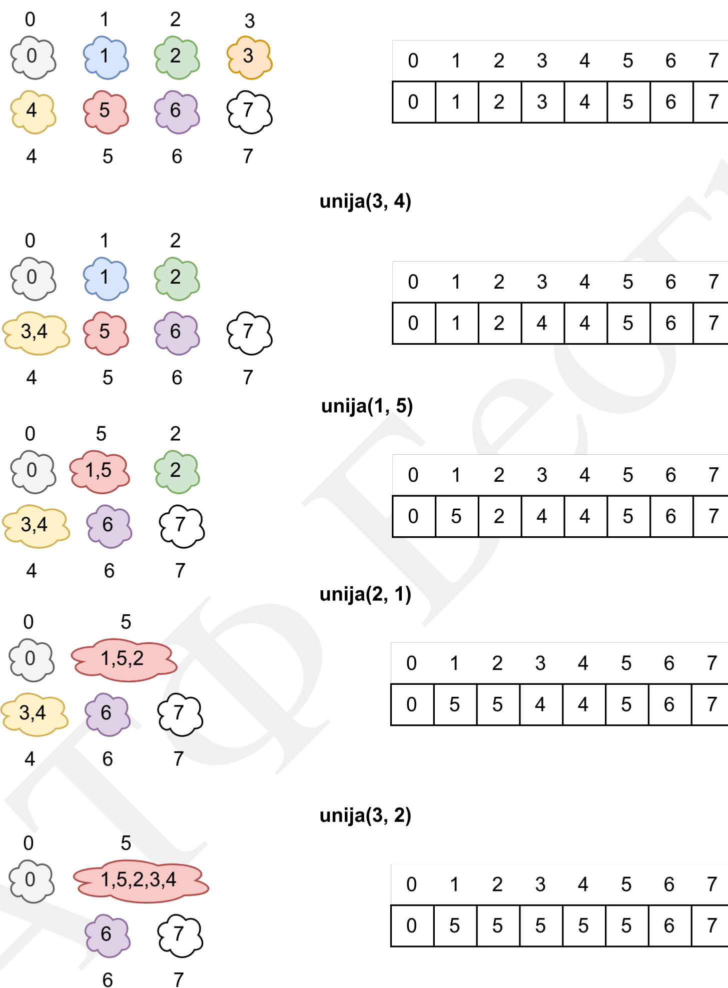
Пример 1.3.1

Илуструјмо спровођење операција `pronadji` и `unija` над описаном имплементацијом структуре података за дисјунктне скупове на једном примеру. Претпоставимо да је почетно стање такво да сваки елемент припада засебном скупу. Размотримо на који начин се мења садржај одговарајућег низа након извршавања операција уније. Извршавање неколико операција уније приказано је на слици 1.8. Лево су приказани скупови, а десно низ којим су ти скупови представљени (изнад сваког низа приказани су индекси). Претпостављамо да ће приликом извршавања операције `unija(x,y)` ознака новог скупа бити ознака скупа којем припада елемент `y`.

Прикажимо сада имплементацију ове технике (једноставности ради користимо глобалне променљиве).

```
// oznaka podskupa kome pripada element
vector<int> id;

// na pocetku svaki element pripada zasebnom skupu
void inicijalizuj(int n) {
  id.resize(n);
  for (int i = 0; i < n; i++)
    id[i] = i;
}

// oznaku podskupa kome pripada element x citamo sa pozicije x iz niza
int pronadji(int x) {
   return id[x];
}

// pravimo uniju podskupova kome pripadaju dati elementi
```

Матф Београд



Слика 1.8: Пример примена операције уније.

МАТФ Београд

```
void unija(int x, int y) {
  int idx = id[x], idy = id[y];
  // oznake svih elemenata prvog podskupa menjamo u oznaku drugog podskupa
  for (int i = 0; i < id.size(); i++)
    if (id[i] == idx)
      id[i] = idy;
}

// elementi su u istom podskupu ako su im oznake iste
int u_istom_podskupu(int x, int y) {
  return pronadji(x) == pronadji(y);
}
```

Обратимо пажњу да се приликом извођења уније морају користити помоћне променљиве (размислите зашто је наредна имплементација неисправна).

```
// pravimo uniju skupova kome pripadaju dati elementi
void unija(int x, int y) {
  // oznake svih elemenata prvog skupa menjamo u oznaku drugog skupa
  for (int i = 0; i < id.size(); i++)
    if (id[i] == id[x])
      id[i] = id[y];
}
```

1.3.2 Ефикасна имплементација

Размотримо нешто другачију имплементацију ове структуре података у којој је операција `unija` временски ефикаснија.

1.3.2.1 Структура и операције

Кључна идеја на којој се заснива ефикасније решење је да елементе не пресликавамо у ознаке скупова, већ да скупове чувамо у облику дрвета (не нужно бинарних) тако да сваки елемент сликамо у његовог родитеља у дрвету. Сваки корен дрвета сликамо самог у себе и сматрамо га представником скупа представљеног тим дрветом (дакле, представник сваког скупа је корен његовог дрвета). Нагласимо да су у чворовима ових дрвета показивачи усмерени од деце ка родитељима, за разлику од класичних дрвета где показивачи у чворовима указују од родитеља ка деци.

Да бисмо за произвољни елемент сазнали ознаку скупа ком припада тј. да бисмо имплементирали операцију `pronadji`, потребно је да почев од тог елемента прођемо кроз низ родитељских чворова све док не стигнемо до корена. Унију два скупа (тј. операцију `unija`) у овом приступу можемо једноставно реализовати тако што корен једног дрвета

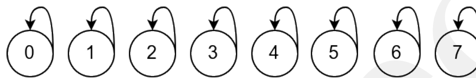
усмеримо ка корену другог.

Наивна имплементација која је описана у претходном поглављу одговара ситуацији у којој особа која промени адресу обавештава све друге особе о својој новој адреси, док имплементација коју тренутно описујемо одговара сценарију у коме само на старој адреси оставља информацију о својој новој адреси. Ово, наравно, мало успорава доставу поште, јер се мора прећи кроз низ преусмеравања, али ако тај низ није предугачак, може бити значајно ефикасније од првог приступа.

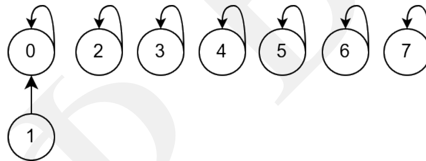
Пример 1.3.2

Прикажимо рад алгоритма на једном примеру. Скупове ћемо представљати дрветима.

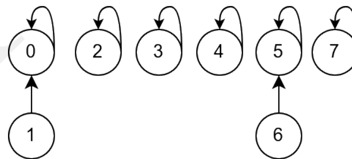
Претпостављамо да су на почетку сви скупови једночлани.



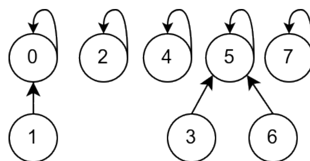
- *Извршава се операција* `unija(1, 0)`*. Представник скупа коме припада* 1 *је* 1*, па чвор* 1 *преусмеравамо ка представнику скупа коме припада* 0*, а то је* 0*.*



- *Извршава се операција* `unija(6, 5)`*. Представник скупа коме припада* 6 *је* 6*, па чвор* 6 *преусмеравамо ка представнику скупа коме припада* 5*, а то је чвор* 5*.*

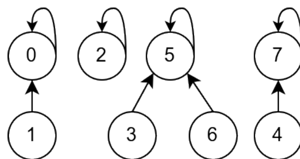


- *Извршава се операција* `unija(3, 6)`*. Представник скупа коме припада* 3 *је* 3*, па чвор* 3 *преусмеравамо ка представнику скупа коме припада* 6*, а то је чвор* 5*.*

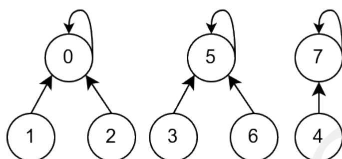


МАТФ Београд

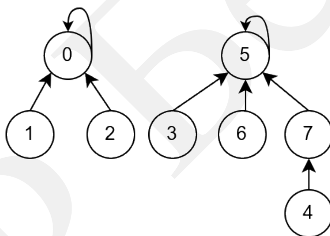
- *Извршава се операција* `unija(4, 7)`. *Представник скупа коме припада* 4 *је* 4, *па чвор* 4 *преусмеравамо ка представника скупа коме припада* 7, *а то је чвор* 7.



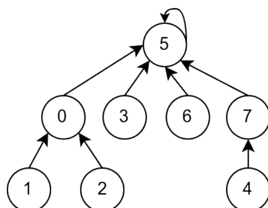
- *Извршава се операција* `unija(2, 0)`. *Представник скупа коме припада* 2 *је* 2, *па чвор* 2 *преусмеравамо ка представнику скупа коме припада* 0, *а то је чвор* 0.



- *Извршава се операција* `unija(4, 3)`. *Представник скупа коме припада* 4 *је* 7, *па чвор* 7 *преусмеравамо ка представнику скупа коме припада* 3, *а то је чвор* 5.



- *Извршава се операција* `unija(2, 6)`. *Представник скупа коме припада* 2 *је* 0, *па чвор* 0 *преусмеравамо ка представнику скупа коме припада* 6, *а то је чвор* 5.



Иако овако описана структура података има дрволику структуру, можемо је имплементирати коришћењем статичког низа. Наиме, пошто сваки елемент у дрвету има јединственог родитеља, на позицији неког елемента у низу можемо чувати индекс његовог родитеља. У случају да је елемент корен неког дрвета, његов родитељ је он сâм.

МАТФ Београд

Пример 1.3.3

За дрво из претходног примера, низ родитеља има следећи садржај:

0	1	2	3	4	5	6	7
5	0	0	5	7	5	5	5

Имајући у виду овакву репрезентацију, кôд је прилично једноставно написати (једноставности ради претпостављамо да се подаци смештају у глобалним променљивим).

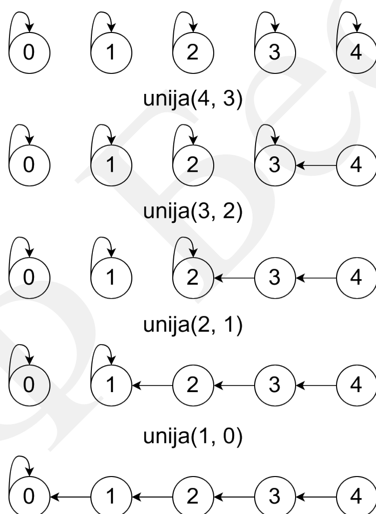
```
// roditelj svakog cvora u drvetu
vector<int> roditelj;

// na pocetku svaki element pripada zasebnom skupu
void inicijalizuj(int n) {
  roditelj.resize(n);
  for (int i = 0; i < n; i++)
    roditelj[i] = i;
}

// naziv podskupa kome element pripada dobijamo kao oznaku korena tog podskupa
int pronadji(int x) {
  // sve dok ne stignemo do korena
  while (roditelj[x] != x)
    // penjemo se u roditeljski cvor
    x = roditelj[x];
  return x;
}

// pravimo uniju podskupova kome pripadaju dati elementi
void unija(int x, int y) {
  int fx = pronadji(x), fy = pronadji(y);

  // x i y imaju istog predstavnika, pa su vec u istom podskupu
  if (fx == fy)
    return;

  // postavljamo da je koren prvog podskupa sin korena drugog podskupa
  roditelj[fx] = fy;
}
```

1.3.2.2 Уравнотежавање дрвета

Сложеност претходног приступа зависи од тога колико су дрвета којима се представљају скупови уравнотежена[6]. У најгорем случају се дрвета могу издегенерисати у листе и тада је сложеност најгорег случаја сваке од операција `unija` и `pronadji` $O(n)$.

Пример 1.3.4

Ако увек приликом извршавања операције `unija(x,y)` *усмеравамо показивач од представника скупа којем припада елемент x ка представнику скупа којем припада елемент y, извршиће се низ измена дрвета који је приказан на слици 1.9. Упит којим се тражи представник скупа којем припада елемент 4 се реализује низом корака којима се прелази преко елемената 3, 2, 1, 0 и у случају већег броја елемената се добија веома неефикасна имплементација проналажења представника (а самим тим и уније, чија имплементација подразумева проналажење представника).*



Слика 1.9: Илустрација извршавања низа операција `unija(x,y)` када се увек показивач од представника скупа коме припада елемент x усмерава ка представнику скупа коме припада елемент y.

Тренутни алгоритам, дакле, није у најгорем случају ефикаснији од наивног приступа. Наиме, у наивном приступу се проналажење представника врши у времену $O(1)$, али унија увек захтева време $O(n)$, док овде и операција `pronadji` може имати сложеност

[6]Појам уравнотеженог дрвета може се прецизно дефинисати на разне начине. У овом тексту ћемо уравнотеженост разматрати само неформално: дрво сматрамо уравнотеженијим што је растојање листова од корена уједначеније.

МАТФ Београд

$O(n)$. Сложеност тренутног алгоритма се може поправити ако се све време дрвета одржавају уравнотеженим. Доказаћемо да је у тој варијанти сложеност најгорег случаја сваке од операција `unija` и `pronadji` једнака $O(\log n)$. Тиме се постиже да је време извршавања низа од m операција типа `unija` или `pronadji` у најгорем случају једнако $O(m \log n)$ (док је у наивном приступу време извршавања овог низа операција, у случају када је број операција типа `unija` $O(m)$, једнако $O(mn)$).

Приликом прављења уније имамо слободу избора корена ког ћемо усмерити према другом корену и уравнотеженост се постиже тиме што се ова слобода на неки начин искористи. Постоје два уобичајена начина вршења уније: *унија на основу ранга (висине)* и *унија на основу броја елемената (величине)*.

Унија на основу ранга (висине)

Основна идеја вршења *уније на основу ранга (висине)* је да се приликом измена (а оне се врше само у склопу операције уније), ако је могуће, обезбеди да се висина[7] дрвета којим је представљена унија не повећа у односу на висине појединачних дрвета која представљају скупове чија се унија прави. Претпоставићемо да сваком чвору дрвета придружујемо број који представља висину тог чвора тј. дрвета са кореном у том чвору. Тај број ћемо назвати *ранг* (енгл. rank)[8] и рангове свих чворова ћемо одржавати у посебном низу `rang`. Ако се увек изабере да корен дрвета мањег ранга усмеравамо ка корену дрвета већег или једнаког ранга, тада се ранг уније повећава само ако су оба дрвета која унирамо истог ранга.

```
// roditelj svakog čvora drveta
vector<int> roditelj;
// rang (visina) svakog čvora drveta
vector<int> rang;

// na pocetku svaki element pripada zasebnom skupu
// i visina svakog drveta je 0
void inicijalizuj(int n) {
  roditelj.resize(n);
  rang.resize(n);
  for (int i = 0; i < n; i++) {
    roditelj[i] = i;
```

[7] Висину чвора можемо дефинисати као број грана на путањи од тог чвора до најудаљенијег листа у поддрвету са кореном у датом чвору. Висину дрвета рачунамо као висину његовог корена.

[8] У поглављу 1.3.2.3 посвећеном сажимању путева, видећемо да када се врше оптимизације, ранг не мора да буде увек једнак висини, али увек представља њену горњу границу тј. број од ког та висина сигурно није већа.

Математички факултет Београд

```
    rang[i] = 0;
  }
}

// pravimo uniju podskupova kojima pripadaju dati elementi
void unija(int x, int y) {
  int fx = pronadji(x), fy = pronadji(y);

  // x i y imaju istog predstavnika, pa su vec u istom podskupu
  if (fx == fy)
    return;

  // usmeravamo koren drveta nizeg ka korenu viseg ranga
  if (rang[fx] < rang[fy])
    swap(fx, fy);
  roditelj[fy] = fx;

  // ako su podskupovi istog ranga
  // unija ce biti za jedan veceg ranga
  if (rang[fx] == rang[fy])
    rang[fx]++;
}
```

Приметимо да су нам за извођење операције уније релевантне само вредности ранга представника скупова.

Докажимо лему која гарантује сложеност.

Лема 1.3.1 **[Унирање на основу ранга даје уравнотежена дрвета]**

Када се унија врши на основу ранга, у сваком дрвету чији је ранг h *налази се бар* 2^h *чворова.*

Доказ. Тврђење доказујемо математичком индукцијом.

- База индукције одговара полазном стању у коме је сваки чвор свој представник. Рангови свих дрвета су тада 0 и сва дрвета имају $2^0 = 1$ чвор, па тврђење важи.
- Покажимо да операција уније одржава ову инваријанту. По индуктивној хипотези знамо да ако два дрвета која представљају скупове који се унирају имају рангове r_1 и r_2, онда је број чворова у њима редом бар 2^{r_1} и 2^{r_2} чворова. Уколико се унирањем ранг не повећа, инваријанта је тривијално очувана јер се број чворова

увећао, а ранг је остао исти. Једини случај када се унирањем повећава ранг је када је $r_1 = r_2$ и тада унирано дрво има ранг $r = r_1 + 1 = r_2 + 1$ и бар $2^{r_1} + 2^{r_2} = 2^{r_1} + 2^{r_1} = 2 \cdot 2^{r_1} = 2^{r_1+1} = 2^r$ чворова.

Тиме је тврђење доказано. □

Дакле, ранг дрвета које има n чворова је $O(\log n)$. Ако је он увек већи или једнак од висине, и висина дрвета је $O(\log n)$, па је сложеност операције проналажења представника у скупу од n чворова, која директно зависи од висине дрвета, реда $O(\log n)$. Пошто унирање након проналажења представника врши још само $O(1)$ операција, и сложеност унирања два скупа је $O(\log n)$. Дакле, одржавање висина дрвета под контролом нам гарантује логаритамску сложеност операција `pronadji` и `unija` и време извршавања низа од m таквих операција је у најгорем случају једнако $O(m \log n)$.

Унија на основу броја елемената (величине)

У приступу *унија на основу броја елемената (величине)*, уместо висине се уз сваки од скупова одржава и број његових елемената тог скупа (тј. у сваком чвору дрвета чува се информација о броју чворова дрвета којем је тај чвор корен).

```
// pravimo uniju podskupova kome pripadaju dati elementi
void unija(int x, int y) {
  int fx = pronadji(x), fy = pronadji(y);

  // x i y imaju istog predstavnika, pa su vec u istom podskupu
  if (fx == fy)
    return;

  // usmeravamo koren manjeg drveta kao korenu veceg drveta
  if (velicina[fx] < velicina[fy])
    swap(fx, fy);
  roditelj[fy] = fx;

  // azuriramo velicinu novog korena
  velicina[fx] += velicina[fy];
}
```

Ако увек усмеравамо представника скупа са мањим бројем елемената ка представнику скупа са већим бројем елемената, поново добијамо логаритамску сложеност најгорег случаја за обе основне операције. Ово важи зато што и овај начин прављења уније гарантује да не можемо имати високо дрво са малим бројем чворова. Наиме, за добијање дрвета висине 1, потребна су бар два дрвета висине 0, односно бар 2 чвора; за дрво ви-

сине 2 потребна су бар два дрвета висине 1 која имају бар по 2 чвора, тј. дрво висине 2 има бар 4 чвора, итд. Доказимо ово.

Лема 1.3.2 [Унирање на основу броја елемената даје уравнотежена дрвета]
Када се унија прави на основу броја елемената, у дрвету чија је висина h налази се бар 2^h чворова тј. за сваки корен r дрвета висине h са s чворова важи $s \geq 2^h$.

Доказ. Докажимо ово тврђење математичком индукцијом.

- У почетку је број чворова у сваком дрвету $s = 1$, а висина сваког дрвета је $h = 0$, па важи $s = 2^h$.
- Претпоставимо да тврђење важи пре спајања два дрвета чији су корени r_1 и r_2, који имају редом s_1 и s_2 елемената, висине h_1 и h_2. На основу индуктивне хипотезе знамо да је $s_1 \geq 2^{h_1}$ и $s_2 \geq 2^{h_2}$. Претпоставимо да се дрво r_2 придружује дрвету r_1 (што значи да дрво са кореном r_2 има мање или једнако чворова него оно са кореном r_1 тј. да је $s_2 \leq s_1$), чиме се добија дрво са s чворова висине h.
  - Ако дрво са кореном r_1 има већу висину од оног са кореном r_2, тј ако је $h_1 > h_2$, након спајања се висина дрвета са кореном r_1 не мања, а број чворова му се повећава, па тврђење леме тривијално наставља да важи. Заиста, важи $s = s_1 + s_2 \geq s_1 \geq 2^{h_1} = 2^h$.
  - У супротном се висина дрвета са кореном дрвета r_1 повећава и постаје за један већа од висине дрвета са кореном r_2 тј. важи $h = h_2 + 1$. Тада важи $s = s_1 + s_2 \geq 2 \cdot s_2 \geq 2 \cdot 2^{h_2} = 2^{h_2+1} = 2^h$.

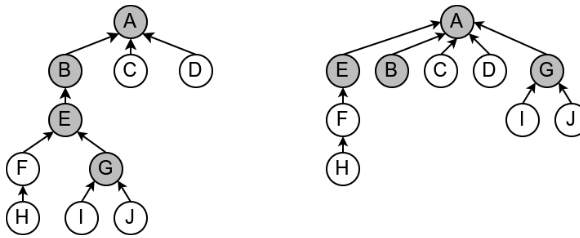
Тиме је тврђење доказано. □

Из ове леме следи да су и у овом приступу висине свих дрвета увек реда $O(\log n)$, што гарантује ефикасност обе операције.

1.3.2.3 Сажимање путева

Иако је сложеност претходно описаних варијанти алгоритма сасвим прихватљива (сложеност извршавања m операција уније је $O(m \log n)$), она се може додатно побољшати веома једноставном техником познатом као *сажимање путева* или *компресија путева* (енгл. path compression). Наиме, приликом проналажења представника можемо све чворове кроз које пролазимо усмерити ка корену. Приметимо да тада поред операције `unija` и операција `pronadji` мења структуру дрвета којим је представљен тај скуп. Један начин да се то уради је да се након проналажења корена, поново прође кроз низ елемената и сви показивачи усмере ка корену (слика 1.10). На овај начин се постиже да будуће операције над тим скупом буду ефикасније.

MATF Beograd



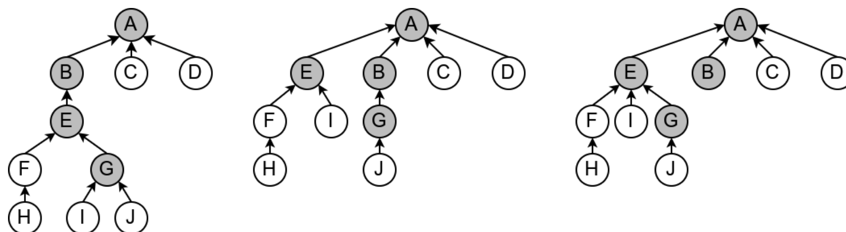
Слика 1.10: Илустрација поступка сажимања путева у два пролаза након тражења представника скупа коме припада елемент G.

```
// naziv podskupa kome element pripada dobijamo kao oznaku korena tog podskupa
int pronadji(int x) {
  int koren = x;
  // nalazimo oznaku podskupa kao koreni element podskupa
  while (koren != roditelj[koren])
    koren = roditelj[koren];
  // svim cvorovima na putanji od x do korena
  // postavljamo da je roditeljski cvor koren tog podskupa
  while (x != koren) {
    int tmp = roditelj[x];
    roditelj[x] = koren;
    x = tmp;
  }
  return koren;
}
```

За све чворове који се обилазе од полазног чвора до корена, дужине путања до корена се смањују на 1, а скраћују се и путање њихових потомака. Ако се врши унирање на основу броја елемената, приликом сажимања се не мењају бројеви елемената дрвета, па подаци придружени чворовима остају коректни. Ако вршимо унију на основу ранга и тумачимо рангове као висине дрвета, приликом сажимања низ рангова постаје неажуран (јер се неке висине смањују, а рангови се не ажурирају). Ипак, ни у овом случају низ не мора да се ажурира. Наиме, пошто се приликом сажимања висине смањују, а рангови се не мењају, рангови не представљају више висине чворова, али представљају њихове горње границе — висина чвора је увек мања или једнака од вредности његовог ранга. Лако се показује да се овим не нарушава сложеност операција. Наиме, пошто се унија и даље врши на основу ранга, и даље важи инваријанта гарантована лемом 1.3.1 да се у сваком дрвету које има ранг h налази бар 2^h чворова. Пошто је висина увек мања или једнака од ранга, важи и да је висина сваког дрвета које садржи n чворова највише $\log n$, што

гарантује сложеност најгорег случаја $O(\log n)$ за обе операције (видећемо да је услед сажимања амортизована сложеност још нижа).

У претходној имплементацији функције `pronadji` се два пута пролази кроз путању од чвора x до корена. Сличне перформансе се могу добити и у само једном пролазу. Постоје два начина на који се ово може урадити.



Слика 1.11: Илустрација два начина сажимања путева у једном пролазу током тражења представника скупа којем припада елемент I (лево је полазно дрво, у средини дрво које се добија првим алгоритмом, а десно дрво које се добија другим алгоритмом).

Први начин је да се сваки чвор кроз који се пролази током проналажења представника (осим детета корена) усмери ка родитељу свог родитеља. За све чворове који се обилазе (на путу од полазног чвора до корена), дужине путања до корена се након овога смањују двоструко (слика 1.11, средина), што је довољно за одличне перформансе.

```
// naziv podskupa kome element pripada dobijamo kao oznaku korena tog podskupa
int pronadji(int x) {
  int tmp;
  // za sve cvorove na putanji od x do korena
  while (x != roditelj[x]) {
    tmp = roditelj[x];
    // novi roditelj od x je roditelj njegovog roditelja
    roditelj[x] = roditelj[roditelj[x]];
    x = tmp;
  }
  return x;
}
```

Други начин подразумева да се, приликом проласка од чвора ка корену, сваки други чвор на путањи усмери ка родитељу свог родитеља (слика 1.11, десно).

```
// naziv podskupa kome element pripada dobijamo kao oznaku korena tog podskupa
int pronadji(int x) {
```

Merge Beograd

```
    // penjemo se do korena tako sto preskacemo po jedan cvor
    while (x != roditelj[x]) {
      // novi roditelj od x je roditelj njegovog roditelja
      roditelj[x] = roditelj[roditelj[x]];
      x = roditelj[x];
    }
    return x;
  }
```

Приметимо да је на овај начин додата само једна линија кода у првобитну имплементацију операције `pronadji`.

Када се примени било који од три наведена облика сажимања путева, амортизована сложеност операција постаје готово константна[9].

Задатак: Први пут кроз матрицу

Логичка матрица димензије $n \times n$ у почетку садржи све нуле. Након тога се насумично додаје једна по једна јединица. Кретање по матрици је могуће само по јединицама и то само на доле, на горе, на десно и на лево. Написати програм који учитава димензију матрице, а затим позицију једне по једне јединице и одређује након колико њих је први пут могуће сићи од врха до дна матрице (са произвољног поља прве врсте које садржи јединицу до произвољног поља последње врсте матрице које садржи јединицу).

Опис улаза

Са стандардног улаза се учитава димензија матрице $1 \leq n \leq 200$, затим број поља m ($1 \leq m \leq n^2$) у које се уписује јединица, а затим у наредних m редова координате тих поља (број врсте и број колоне од 0 до $n-1$, раздвојени размаком).

Опис излаза

На стандардни излаз исписати најмањи број додатих јединица након којих је постало могуће стићи од врха до дна.

[9] Прецизније, амортизована сложеност операција када се примени сажимање путева једнака је $O(\alpha(n))$, где је $\alpha(n)$ такозвана инверзна Акерманова функција која јако споро расте. Акерманова функција је позната по томе што јако брзо расте. Она је дефинисана рекурентним релацијама: $A(0,n) = n+1$, $A(m+1,0) = A(m,1)$, $A(m+1,n+1) = A(m, A(m+1,n))$. Функција $\alpha(n)$ је инверзна функција функције $A(n,n)$ и она јако споро расте. За било који број n који је мањи од броја атома у целом универзуму (око 10^{80}) важи да је $\alpha(n) < 5$, тако да је амортизована временска сложеност операција практично константна.

МАТФ Београд

Пример

Улаз	*Излаз*	*Објашњење*
4	8	После 8 учитаних поља, матрица постаје
9		1100
0 0		0101
0 1		1100
1 1		1001
3 3		и врх и дно постају спојени.
1 3		
2 0		
3 0		
2 1		
2 2		

Решење

Основна идеја је да се формирају сви подскупови поља матрице између којих постоји пут (они формирају класе еквиваленције тзв. компоненте повезаности). Сваки пут када се успостави веза између нека два поља матрице, подскупови којима она припадају се спајају. Провера да ли постоји пут између два поља матрице своди се онда на проверу да ли она припадају истом подскупу.

Путања од врха до дна постоји ако и само ако постоји путања од било ког поља у првој врсти матрице до било ког поља у последњој врсти матрице. Међутим, не морамо у сваком кораку морамо да проверавамо све парове елемената из прве и последње врсте. Додаћемо вештачки почетни чвор (назовимо га извор) и спојићемо га са свим чворовима у првој врсти матрице и завршни чвор (назовимо га ушће) који ћемо спојити са свим чворовима у последњој врсти матрице. Тада се у сваком кораку може проверити само да ли су извор и ушће спојени тј. да ли припадају истом подскупу. Приметимо да су додавањем ових помоћних чворова сви елементи прве врсте матрице обједињени у исти подскуп иако у њима пише нуле, међутим, то не нарушава коректност.

Подскупове можемо чувати помоћу структуре података за представљање дисјунктних подскупова.

```
int n;
cin >> n;

// alociramo matricu dimenzije n puta n
vector<vector<bool>> a(n);
for (int i = 0; i < n; i++)
  a[i].resize(n, false);
```

Матф Београд

Компоненте повезаности су обојене различитим бојама. Пошто су извор и ушће исто обојени, постоји пут од врха до дна.

```
// svakom polju matrice (x, y) dodeljujemo jedinstveni redni broj (kod)
auto kod = [n](int x, int y) { return x*n + y; };

// dva dodatna veštačka čvora
const int izvor = n*n;
const int usce = n*n+1;

// inicijalizujemo union-find strukturu za sve elemente matrice
// (njih n*n), izvor i ušće
inicijalizuj(n*n + 2);

// spajamo izvor sa svim elementima u prvoj vrsti matrice
for (int i = 0; i < n; i++)
  unija(izvor, kod(0, i));

// spajamo sve elemente u poslednjoj vrsti matrice sa ušćem
for (int i = 0; i < n; i++)
  unija(kod(n-1, i), usce);

// broj jedinica
int m;
cin >> m;
```

```
// korak u kom se spajaju izvor i usce
int korak = -1;

// ucitavamo i obrađujemo jednu po jednu jedinicu
for (int k = 1; k <= m; k++) {
  int x, y;
  cin >> x >> y;
  // ako je u matrici već jedinica, nema šta da se radi
  if (a[x][y]) continue;
  // upisujemo jedinicu u matricu
  a[x][y] = true;
  // povezujemo podskupove u sva četiri smera
  if (x > 0 && a[x-1][y])     unija(kod(x, y), kod(x-1, y));
  if (x + 1 < n && a[x+1][y]) unija(kod(x, y), kod(x+1, y));
  if (y > 0 && a[x][y-1])     unija(kod(x, y), kod(x, y-1));
  if (y + 1 < n && a[x][y+1]) unija(kod(x, y), kod(x, y+1));
  // proveravamo da li su izvor i ušće spojeni
  if (pronadji(izvor) == pronadji(usce)) {
    korak = k;
    break;
  }
}

cout << korak << endl;
```

1.4 Упити распона

Неке структуре података су посебно погодне за проблеме у којима се тражи да се над елементима низа израчунавају статистике (нпр. збир, минимум, максимум) неких сегмената тј. распона узастопних елемената низа. Захтеве тог типа називамо *упити распона* (енгл. range queries). *Статички упити распона* (енгл. static range queries), описани у поглављу 1.4.1, подразумевају да се низ једном иницијализује и након тога не мења, док *динамички упити распона* (енгл. dynamic range queries), описани у поглављу 1.4.2, омогућавају да се низ мења између израчунавања статистика. Ове структуре података обично подржавају неке од следећих врста упита:

- одређивање елемента низа на датој позицији (енгл. *point query*),
- одређивање статистике елемената у датом сегменту низа (енгл. *range query*),

- ажурирање елемента низа на датој позицији (енгл. *point update*),
- ажурирање елемената низа у датом сегменту (енгл. *range update*), обично или тако што се сви елементи поставе на неку дату вредност или тако што се сви елементи увећају за дату вредност.

Приметимо да могућност ефикасног одређивања збира елемената произвољног сегмента низа гарантује уједно и могућност одређивања елемента на датој позицији (јер се вредност тог елемента може одредити као збир једночланог сегмента који садржи само тај елемент). Слично, могућност ефикасног ажурирања произвољног сегмента гарантује могућност ефикасног ажурирања појединачних елемената низа.

- *Низ збирова префикса* (енгл. prefix sum array) омогућава ефикасно рачунање статистика сегмената и вредности појединачног елемента (*range query*, *point query*).
- *Низ разлика суседних елемената* (енгл. difference array) омогућава ефикасно ажурирање сегмената (*range update*).
- *Сегментно дрво* (енгл. segment tree) и *Фенвиково дрво* (енгл. Fenwick tree) омогућавају ефикасно извршавање статистика сегмената и одређивање вредности елемената (*range query*, *point query*), као и ажурирање вредности појединачног елемента (*point update*), при чему се Фенвикова дрвета користе обично само за рачунање збира, док се сегментна дрвета користе за шири спектар статистика.
- *Лењо сегментно дрво* (енгл. lazy segment tree) омогућава ефикасно извршавање све четири наведене врсте упита.

Иако ћемо све структуре података приказати у једнодимензионом облику, у реалним применама се веома често разматрају дводимензионална, па и тродимензионална уопштења. На пример, у дводимензионалном случају могуће је ефикасно очитавати збирове произвољних правоугаоних сегмената матрице.

1.4.1 Статички упити распона

За почетак ћемо разматрати статичке упите распона, код којих је задатак омогућити ефикасно извршавање потенцијално великог броја различитих упита над сегментима датог низа, при чему се вредности у низу не мењају.

1.4.1.1 Збирови префикса

Проблем

Дефинисати структуру података која обезбеђује ефикасно израчунавање збирова сегмената датог низа (сегмент одређен позицијама $[a, b]$ се састоји од узастопних елемената низа од позиције a до позиције b, укључујући и њих).

Матф Београд

У решењу грубом силом се сви елементи смештају у низ x и при сваком упиту се изнова рачуна збир елемената одговарајућег сегмента $[a, b]$. Укупно време да се сви упити изврше је реда $O(mn)$, где је са m означен број упита, а са n дужина низа, што је у случају дугачких низова и великог броја упита недопустиво неефикасно. И сложеност најгорег случаја и амортизована сложеност извршавања једног упита су реда $O(n)$.

Једноставно решење је засновано на идеји да уместо да чувамо елементе низа x, чувамо *низ збирова префикса* низа $P_0 = 0$, $P_{i+1} = \sum_{k=0}^{i} x_k$ за $0 \leq i \leq n$. Дакле, збир P_i чува збир првих i елемената низа (пошто бројање позиција почиње од нуле, збир елемената $x_0, \ldots, x_i$ садржи $i + 1$ сабирак и једнак је P_{i+1}). Збир елемената сваког сегмента $[a, b]$ онда можемо разложити на разлику префикса низа x до елемента b и префикса до елемента $a - 1$:

$$\sum_{k=a}^{b} x_k = \sum_{k=0}^{b} x_k - \sum_{k=0}^{a-1} x_k = P_{b+1} - P_a.$$

Сви збирови префикса P_i низа x могу се израчунати алгоритмом сложености $O(n)$ и сместити у додатни (а ако је уштеда меморије битна, онда чак и у оригинални) низ. Након овакве предобраде, збир сваког сегмента се може израчунати алгоритмом сложености $O(1)$, па је укупна сложеност извршавања m упита рачунања збира елемената неког сегмента низа x једнака $O(n + m)$.

Полазни низ x се може реконструисати рачунањем низа разлика суседних елемената низа префикса P: $x_i = P_{i+1} - P_i$ за $0 \leq i < n$.

Дводимензионални збирови префикса

У дводимензионалном случају довољно је за сваки елемент матрице на позицији (v, k) памтити збир $P_{v+1,k+1}$ елемената правоугаоног сегмента којем је горње лево теме поље $(0, 0)$, а доње десно (v, k), при чему се у врсти и колони 0 налазе нуле. Тада се збир произвољног сегмента одређеног горњим-левим теменом (v_1, k_1) и доњим-десним теменом (v_2, k_2) може изразити као $P_{v_2+1,k_2+1} - P_{v_2+1,k_1} - P_{v_1,k_2+1} + P_{v_1,k_1}$.

Пример 1.4.1

Збир елемената у црвеном правоугаонику леве матрице са слике 1.12 једнак је $87-30-32+11$. *Број* 87 *је збир елемената матрице од њеног горњег левог угла, до доњег десног угла црвеног правоугаоника, тј. збир елемената у зеленом, жутом, плавом и црвеном делу матрице (ж+з+п+ц). Број* 30 *је збир елемената матрице од њеног горњег левог угла до доњег десног угла плавог правоугаоника, тј. збир елемената у зеленом и плавом делу матрице (з+п). Број* 32 *је збир елемената матрице од њеног горњег левог угла до доњег*

МАТФ Београд

1	3	2	3	2	3	1	1
4	3	1	4	2	4	2	3
1	2	1	3	2	3	2	4
1	4	2	3	2	1	3	1
3	2	4	3	2	2	2	2
4	2	2	1	4	1	3	4
1	3	4	1	2	2	1	3
1	3	2	4	3	3	4	2

0	0	0	0	0	0	0	0	0
0	1	4	6	9	11	14	15	16
0	5	***11***	14	21	25	***32***	35	39
0	6	14	18	28	34	44	49	57
0	7	19	25	38	46	57	65	74
0	10	24	34	50	60	73	83	94
0	14	***30***	42	59	73	***87***	100	115
0	15	34	50	68	84	100	114	132
0	16	38	56	78	97	116	134	154

Слика 1.12: Дводимензионални префиксни збирови за матрицу лево су приказани у матрици десно.

десног угла зеленог правоугаоника тј. збир елемената у зеленом и жутом правоугаонику (з+ж). Број 11 је збир елемената матрице од њеног горњег левог угла до доњег десног угла зеленог правоугаоника тј. збир елемената у зеленом правоугаонику (з). Изразом $87 - 32 - 30 + 11$*, се дакле, рачуна (з+ж+п+ц) - (ж+з) - (п+з) + з, што је једнако ц.*

Збирови префикса (и једнодимензиони и вишедимензиони) имају различите примене у анализи података, обради података, финансијској анализи и др. У обради слика се дводимензионални збирови префикса користе за израчунавање различитих прорачуна над регионима слике који омогућавају разне операције над сликом попут детекције ивица, замућења и других.

1.4.1.2 Разлике суседних елемената

Донекле сродан проблем је и следећи.

Проблем

Дефинисати структуру података која омогућава ефикасно извршавање упита облика $([a, b], c)$*, који подразумевају да се сви елементи низа* x *на позицијама из сегмента* $[a, b]$ *увећају за вредност* c*. Потребно је одредити садржај низа* x *након извршавања свих упита.*

Директно решење би за сваки упит у петљи увећавало све елементе низа x на позицијама из сегмента $[a, b]$. Сложеност тог наивног приступа је $O(mn)$, где је са m означен број упита, а са n дужина низа.

Много ефикасније решење се може добити ако се уместо елемената низа памти *низ разлика свака два суседна елемента низа*: $R_0 = x_0, R_i = x_i - x_{i-1}$ за свако i из-

МАТФ Београд

међу 1 и $n-1$. Током увећавања свих елемената сегмента $[a, b]$ низа x за вредност c, мењају се само разлике између елемената на позицијама a и $a-1$ (разлика R_a се увећава за c) као и између елемената на позицијама $b+1$ и b (разлика R_{b+1} се умањује за c), па је измену могуће извршити у времену $O(1)$. Ако знамо све елементе низа, тада низ разлика суседних елемената можемо веома једноставно израчунати алгоритмом сложености $O(n)$. Са друге стране, ако знамо низ разлика неког низа, тада тај низ можемо такође веома једноставно реконструисати, израчунавајући збирове префикса низа разлика, у времену $O(n)$. На тај начин, на основу разлика елемената финалног низа, можемо лако реконструисати тај финални низ.

Ако не желимо да у имплементацији први и последњи елемент низа третирамо другачије од осталих, можемо претпоставити да почетни низ и низ разлика проширујемо са по једном нулом са леве и десне стране (тада је $R_i = x_{i+1} - x_i$, за свако i од 0 до $n-1$).

Оригинални низа се реконструише израчунавањем префиксних збирова низа разлика суседних елемената, што указује на дубоку везу између ове две технике. Заправо, разлике суседних елемената представљају одређени дискретни аналогон извода функције, док префиксни збирови онда представљају аналогон одређеног интеграла. Израчунавање збира сегмента као разлике два збира префикса одговара Њутн-Лајбницовој формули.

1.4.2 Динамички упити распона

За разлику од претходних, статичких упита над распонима, овде ћемо разматрати тзв. динамичке упите над распонима који дозвољавају да се низ мења током времена, тако да је потребно развити напредније структуре података које омогућавају извршавање оба типа упита (читање и ажурирање) ефикасно.

Наиме, низ збирова префикса омогућава ефикасно израчунавање збирова сегмената низа код низова чији се садржај не мења. Са друге стране, низ збирова префикса не омогућава ефикасно ажурирање елемената низа, јер је при свакој измени неког елемента низа потребно ажурирати и збирове префикса, што је нарочито неефикасно када се ажурирају елементи близу почетка низа (сложеност најгорег случаја је $O(n)$), па се не могу примењивати у ситуацијама у којима су упити израчунавања збирова сегмената испреплетени са упитима ажурирања вредности елемената низа.

Слично, низ разлика суседних елемената допушта стална ажурирања елемената низа. Међутим, извршавање упита којима се захтева одређивање елемената низа подразумева реконструкцију садржаја низа на основу низа разлика, што је сложености $O(n)$. Стога низ разлика није пожељно употребљавати у ситуацијама када су упити увећавања сегмената и очитавања вредности елемената низа испреплетани.

Проблеми које ћемо разматрати у овом поглављу су специфични по томе што омогућа-

Матф Београд

вају да се упити ажурирања низа и очитавања његових статистика јављају испреплетано. За почетак размотримо мало једноставнији проблем.

Проблем

Дефинисати структуру података која обезбеђује ефикасно израчунавање збирова сегмената датог низа одређених позицијама $[a, b]$ (самим тим и појединачних елемената низа), као и ефикасно мењање вредности појединачних елемената низа.

Видећемо да неке структуре података допуштају да се уместо збира користе и неке друге операције. У наредним поглављима ћемо размотрити и сложенији проблем у ком ће бити допуштено ажурирање сегмената низа (а не само појединачних елемената).

Дакле, за почетак претпостављамо да имамо дат низ од m операција од којих је свака или израчунавање збира неког сегмента или ажурирање појединачног елемента низа и циљ је минимизовати укупно време извршавања овог низа операција. У овом случају нам није од користи структура података код које се једна од ове две операције извршава ефикасно, а друга неефикасно јер се може десити да је већина (или су све) од m датих операција баш тог другог типа. Наравно, треба узети у обзир и време иницијализације структуре података, међутим, пошто се иницијализација врши само једном, за разлику од упита за које претпостављамо да се извршавају пуно пута, фокусираћемо се на сложеност времена извршавања упита.

У наставку ћемо размотрити две различите, али донекле сличне структуре података које дају ефикасно решење претходног и њему сличних проблема: *сегментно дрво* и *Фенвиково дрво*. Идеја обе ове структуре података на неки начин наликује коришћењу префиксних сума: чувају се збирови неких унапред погодно изабраних сегмената и они се користе за ефикасно рачунање збира произвољног сегмента.

1.4.2.1 Сегментна дрвета

Једна структура података која омогућава прилично једноставно и ефикасно решавање претходно описаног проблема је *сегментно дрво* (енгл. segment tree). Слично као код низова префикса, током фазе предобраде израчунавају се збирови сегмената полазног низа, а онда се збир елемената произвољног сегмента полазног низа изражава путем тих унапред израчунатих збирова. Разлика је то што се не израчунавају збирови свих префикса, већ само посебно одабраних сегмената, као и то што се збир произвољног сегмента у општем случају израчунава обрадом неколико збирова сегмената (не само два). Сегментна дрвета се не користе само за рачунање збира елемената, већ се могу користити и за друге статистике сегмената које се израчунавају асоцијативним операцијама (на пример за одређивање производа, најмањег или највећег елемента, нзд-а или нзс-а свих елемената и слично). По истом принципу дефинисане су структуре података под називом *квад-дрвета* (енгл. quad tree) и *окт-дрвета* (енгл. oct tree) која представљају

МАТФ Београд

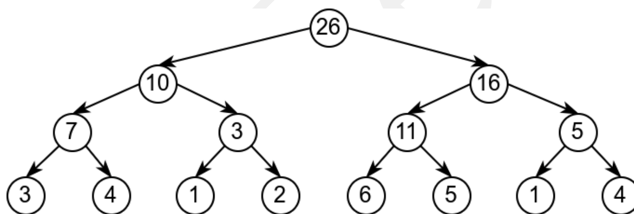
дводимензионална и тродимензионална уопштења сегментних древета.

Сегментна дрвета имају примену у области рачунарске геометрије, за проблеме који се решавају техником покретне праве, као што је проналажење пресечних тачака између дужи у равни, испитивање припадности тачке скупу интервала, израчунавање површине покривене скупом правоугаоника у равни и сл. Користе се и у другим областима, попут обраде слика и географских информационих система.

Опишимо процес конструкције сегментног дрвета. Претпоставимо да је дужина низа степен броја 2; ако није, низ се може допунити до најближег степена броја 2, у случају рачунања збирова нулама[10]. Чланови низа представљају листове дрвета. Групишемо два по два суседна чвора и на сваком претходном нивоу дрвета чувамо родитељске чворове који садрже збирове своја два детета.

Пример 1.4.2

Сегментно дрво којим се ефикасно могу рачунати збирови сегмената низа 3, 4, 1, 2, 6, 5, 1, 4 *приказано је на слици 1.13.*



Слика 1.13: Пример сегментног дрвета.

Пошто је дрво потпуно, најједноставнија имплементација подразумева да се чува имплицитно у низу (слично као у случају хипа). Претпоставићемо да елементе дрвета смештамо од позиције 1, јер је тада аритметика са индексима мало једноставнија (елементи полазног низа могу бити индексирани и уобичајено, кренувши од нуле). Корен се смешта на позицију 1, његова два детета на позиције 2 и 3, њихова деца на позиције 4, 5, 6 и 7 итд.

[10] На допуњена места треба уписати неутрални елемент за операцију која се спроводи, тј. елемент n који за свако x задовољава да је $f(x,n) = x$. На пример, ако се сегментним дрветом рачунају производи сегмената, низ се може допунити до степена двојке јединицама, ако се рачуна максимум, низ се може допунити вредностима $-\infty$, ако се рачуна минимум низ се може допунити вредностима ∞, ако се рачуна највећи заједнички делилац, низ се може допунити нулама, а ако се рачуна најмањи заједнички садржалац, низ се може допунити јединицама.

Матф Београд

Пример 1.4.3

Дрво из претходног примера се представља следећим низом (први ред садржи позиције, а други елементе тог низа):

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-	26	10	16	7	3	11	5	3	4	1	2	6	5	1	4

Корен дрвета је смештен на позицији 1. Ако полазни низ садржи n елемената, онда се у сегментном дрвету елементи полазног низа налазе се на позицијама $[n, 2n-1]$. Елемент који се у полазном низу налази на позицији p, се у сегментном дрвету налази на позицији $p+n$. Лево дете чвора на позицији k налази се на позицији $2k$, а десно на позицији $2k+1$. Дакле, на парним позицијама се налазе лева деца својих родитеља, а на непарним десна. Родитељ чвора k налази се на позицији $\lfloor \frac{k}{2} \rfloor$.

Пример 1.4.4

На слици 1.13 на којој је представљено сегментно дрво за низ од 8 елемената можемо уочити да се елементи полазног низа налазе на позицијама $[8, 15]$. Лево дете чвора са вредношћу 10 који се налази на позицији 2 је чвор са вредношћу 7 који се налази на позицији $2 \cdot 2 = 4$, а десно дете је чвор са вредношћу 3 који се налази на позицији $2 \cdot 2 + 1 = 5$. Родитељ и чвора на позицији 4 и чвора на позицији 5 је чвор који се налази на позицији $\lfloor \frac{4}{2} \rfloor = \lfloor \frac{5}{2} \rfloor = 2$.

Формирање сегментног дрвета

Формирање сегментног дрвета на основу датог низа је веома једноставно.

Итеративни поступак

Најпре се елементи полазног низа прекопирају у дрво, кренувши од позиције n. Затим се сви унутрашњи чворови дрвета (од позиције $n-1$, па уназад до позиције 1) попуњавају као збирови своје деце (на позицију k уписујемо збир елемената на позицијама $2k$ и $2k+1$). Коректност овог поступка лако се доказује индукцијом.

```
// na osnovu datog niza a dužine n u kom su elementi smešteni od
// pozicije 0 formira se segmentno drvo i elementi mu se smeštaju u
// niz drvo krenuvši od pozicije 1
vector<int> formirajSegmentnoDrvo(const vector<int>& a) {
  int n = stepenDvojke(a.size());
  vector<int> drvo(2*n, 0);
  // kopiramo originalni niz u listove (u niz drvo, od pozicije n nadalje)
```

Матф Београд

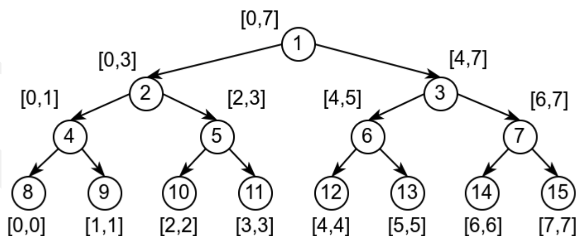
```
  copy(begin(a), end(a), next(begin(drvo), n));
  // ažuriramo roditelje već upisanih elemenata
  for (int k = n-1; k >= 1; k--)
    drvo[k] = drvo[2*k] + drvo[2*k+1];
  return drvo;
}
```

Сложеност ове операције је очигледно линеарна у односу на дужину низа n.

Претходни приступ формира дрво одоздо навише (прво се попуне листови, па онда унутрашњи чворови све док се не дође до корена).

Рекурзивни поступак

Још један начин да се сегментно дрво формира је рекурзивно, одозго наниже. Иако је ова имплементација компликованија и мало неефикаснија (додуше не асимптотски) услед рекурзивних позива, приступ одозго наниже је у неким каснијим операцијама неизбежан, па га илуструјемо на овом једноставном примеру. Сваки чвор дрвета представља збир одређеног сегмента позиција полазног низа. Сегмент је једнозначно одређен позицијом k у низу који одговара сегментном дрвету, али да бисмо олакшали имплементацију, границе тог сегмента можемо кроз рекурзију прослеђивати као параметар функције, заједно са вредношћу k (нека је то сегмент $[x, y]$). На слици 1.14 за сваки чвор сегментног дрвета дат је његов индекс у одговарајућем низу `drvo` и границе сегмента које тај чвор покрива.



Слика 1.14: Унутар сваког чвора је приказан његов индекс k у низу којим је дрво представљено, а поред чвора је приказан сегмент $[x, y]$ који тај чвор покрива.

Дрво крећемо да градимо од корена где је $k = 1$ и $[x, y] = [0, n-1]$. Ако родитељски чвор покрива сегмент $[x, y]$, тада лево дете покрива сегмент $[x, \lfloor \frac{x+y}{2} \rfloor]$, а десно дете покрива сегмент $[\lfloor \frac{x+y}{2} \rfloor + 1, y]$. Дрво попуњавамо рекурзивно, тако што најпре попунимо лево поддрво, затим десно поддрво и на крају вредност у корену израчунавамо као збир вредности у левом и десном детету. Излаз из рекурзије представљају листови, које препознајемо по томе што покривају сегменте дужине 1, и у њих само копирамо

елементе са одговарајућих позиција полазног низа.

```
// od elemenata niza a sa pozicija [x, y] formira se segmentno drvo i
// elementi mu se smeštaju u niz drvo krenuvši od pozicije k
void formirajSegmentnoDrvo(const vector<int>& a, vector<int>& drvo,
                           size_t k, size_t x, size_t y) {
  if (x == y)
    // u listove prepisujemo elemente polaznog niza
    drvo[k] = x < a.size() ? a[x] : 0;
  else {
    // rekurzivno formiramo levo i desno poddrvo
    int s = (x + y) / 2;
    formirajSegmentnoDrvo(a, drvo, 2*k, x, s);
    formirajSegmentnoDrvo(a, drvo, 2*k+1, s+1, y);
    // izračunavamo vrednost u korenu
    drvo[k] = drvo[2*k] + drvo[2*k+1];
  }
}

// na osnovu datog niza a duzine n u kom su elementi smesteni od
// pozicije 0 formira se segmentno drvo i elementi mu se smeštaju u
// niz drvo krenuvši od pozicije 1
vector<int> formirajSegmentnoDrvo(const vector<int>& a) {
  // niz implicitno dopunjujemo nulama tako da mu duzina postane
  // najblizi stepen dvojke
  int n = stepenDvojke(a.size());
  vector<int> drvo(n * 2);
  // krećemo formiranje od korena, koji se nalazi u nizu drvo
  // na poziciji 1 i pokriva elemente na pozicijama [0, n-1]
  formirajSegmentnoDrvo(a, drvo, 1, 0, n - 1);
  return drvo;
}
```

Временску сложеност претходне рекурзивне имплементације можемо описати наредном рекурентном једначином: $T(n) = 2T(n/2) + O(1), T(1) = O(1)$. Њено решење се добија директно из мастер теореме и износи $O(n)$, што је исто као и у случају итеративног формирања сегментног дрвета.

Рачунање збира елемената сегмента

И збир елемената можемо израчунати било итеративним, било рекурзивним поступком.

Итеративни поступак

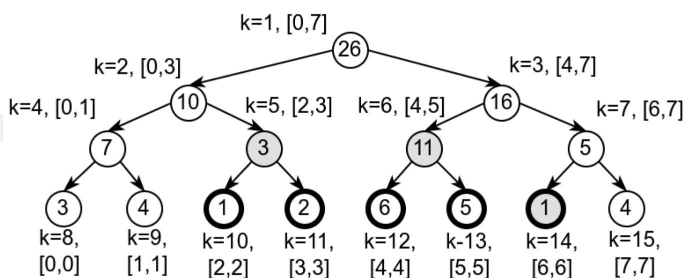
Илуструјмо итеративни поступак (кажемо и поступак одоздо навише) помоћу неколико примера.

Пример 1.4.5

Размотримо како бисмо нашли збир елемената низа $3, 4, 1, 2, 6, 5, 1, 4$ *на позицијама из сегмента* $[2, 6]$*, тј. збир елемената* $1, 2, 6, 5$ *и* 1*. У сегментном дрвету тај сегмент је смештен на позицијама из сегмента* $[2 + 8, 6 + 8] = [10, 14]$ *(слика 1.15). Јасно је да не треба ићи до листова и редом сабирати елементе, већ на паметан начин искористити већ израчунате збирове сегмената које се налазе у дрвету.*

Збир прва два елемента $1, 2$ *(који су на позицијама* 10 *и* 11*) се налази у чвору изнад њих (на позицији* 5*), збир наредна два елемената* 6 *и* 5 *(који су на позицијама* 12 *и* 13*) се налази такође у чвору изнад њих (на позицији* 6*), док се у родитељском чвору елемента* 1 *(који је на позицији* 14*) налази његов збир са елементом* 4 *(који је на позицији* 15 *и који не припада сегменту који сабирамо). Зато збир елемената на позицијама из сегмента* $[10, 14]$ *у сегментном дрвету можемо разложити на збир елемената на позицијама из сегмента* $[5, 6]$ *и елемента на позицији* 14 *који одмах засебно додајемо на тражени збир. На овај начин пењемо се на ниво изнад текућег (долазимо до родитеља тих чворова) и настављамо поступак рачунајући збир елемената на позицијама из сегмента* $[5, 6]$*.*

На позицији 5 *налази се елемент* 3*, а на позицији* 6 *елемент* 11*. У родитељском чвору елемента* 3 *(на позицији* 5*) налази се збир са елементом лево од њега, који не припада сегменту који треба да саберемо. Слично, у родитељском чвору елемента* 11 *(на позицији* 6*) налази се збир овог елемента са елементом десно од њега који не припада сегменту чији збир рачунамо. Стога оба елемента* 3 *и* 11 *додајемо засебно на збир и остајемо са празним сегментом, чиме се поступак рачунања збира датог сегмента завршава.*



Слика 1.15: Рачунање збира елемената из сегмента $[2, 6]$ приступом одоздо навише. Подебљани су листови који одговарају траженом сегменту, а обојени су чворови чије се вредности сабирају.

МАТФ Београд

Пример 1.4.6

Размотримо како бисмо рачунали збир елемената на позицијама из сегмента $[3, 7]$, *тј. збир елемената* $2, 6, 5, 1, 4$. *У сегментном дрвету тај сегмент је смештен на позицијама* $[3 + 8, 7 + 8] = [11, 15]$.

У родитељском чвору елемента 2 *(који је на позицији* 11*) налази се његов збир са елементом* 1*, лево од њега који не припада сегменту који сабирамо. Стога елемент* 2 *одмах додајемо на збир.*

Збирови елемената 6 *и* 5 *(на позицијама* 12 *и* 13*) и елемената* 1 *и* 4 *(на позицијама* 14 *и* 15*) се налазе у чворовима изнад њих (на позицијама* 6 *и* 7*), па се проблем своди на израчунавање збира елемената на позицијама* 6 *и* 7.

Тај збир је већ израчунат, износи 16 *и налази се на позицији* 3*, тако да је само потребно да и њега додамо на збир.*

Дакле, резултат се добија сабирањем елемента 2 *на позицији* 11 *и елемента* 16 *на позицији* 3.

Генерално, за све унутрашње елементе сегмента чији збир рачунамо смо сигурни да се њихов збир налази у чворовима изнад њих. Једини изузетак могу да буду елементи на крајевима сегмента.

- Ако је елемент на левом крају сегмента лево дете (што је еквивалентно томе да се налази на парној позицији) тада се у његовом родитељском чвору налази његов збир са елементом десно од њега који такође припада сегменту који треба сабрати (осим евентуално у случају једночланог сегмента).
- У супротном (ако се налази на непарној позицији), у његовом родитељском чвору је његов збир са елементом лево од њега, који не припада сегменту који сабирамо. У тој ситуацији, тај елемент ћемо посебно додати на збир и искључити из сегмента који сабирамо помоћу родитељских чворова.
- Ако је елемент на десном крају сегмента лево дете (ако се налази на парној позицији), тада се у његовом родитељском чвору налази његов збир са елементом десно од њега, који не припада сегменту који сабирамо. И у тој ситуацији, тај елемент ћемо посебно додати на збир и искључити из сегмента који сабирамо помоћу родитељских чворова.
- Коначно, ако се крајњи десни елемент налази у десном чвору (ако је на непарној позицији), тада се у његовом родитељском чвору налази његов збир са елементом лево од њега, који припада сегменту који сабирамо (осим евентуално у случају једночланог сегмента).

Матф Београд

```
// izračunava se zbir elemenata polaznog niza dužine n koji se
// nalaze na pozicijama iz segmenta [a, b] na osnovu segmentnog drveta
// koje je smešteno u nizu drvo, krenuvši od pozicije 1
int zbirSegmenta(const vector<int>& drvo, int a, int b) {
  int n = drvo.size() / 2;
  a += n; b += n;
  int zbir = 0;
  // sve dok je segment neprazan
  while (a <= b) {
    // ako je levi kraj segmenta desno dete, dodajemo ga posebno u zbir
    if (a % 2 == 1) zbir += drvo[a++];
    // ako je desni kraj segmenta levo dete, dodajemo ga posebno u zbir
    if (b % 2 == 0) zbir += drvo[b--];
    // penjemo se na nivo iznad, na roditeljske cvorove
    a /= 2;
    b /= 2;
  }
  return zbir;
}
```

Пошто се у сваком кораку дужина сегмента $[a, b]$ полови, а она је иницијално сигурно мања или једнака n, сложеност ове операције је $O(\log n)$.

Рекурзивни поступак

Претходна имплементација врши израчунавање одоздо навише. И за ову операцију можемо направити рекурзивну имплементацију која врши израчунавање одозго наниже. Функција као аргумент прима чвор дрвета (који је одређен позицијом k и који садржи збир елемената на позицијама из сегмента $[x, y]$). У општем случају, неки елементи на позицијама сегмента $[a, b]$ чији збир елемената рачунамо су лево од текућег чвора, а неки десно (или су сви лево или сви десно). За сваки чвор у сегментном дрвету функција враћа колики је допринос сегмента који одговара том чвору и његовим наследницима траженом збиру елемената на позицијама из сегмента $[a, b]$ у полазном низу. На почетку крећемо од корена и рачунамо допринос целог дрвета збиру елемената из сегмента $[a, b]$. Постоје три различита могућа односа између сегмента $[x, y]$ који одговара текућем чвору и сегмента $[a, b]$ чији збир елемената тражимо.

- Ако су сегменти дисјунктни, тј. ако је $y < a$ или је $x > b$, допринос текућег чвора збиру сегмента $[a, b]$ је нула.
- Ако је сегмент $[x, y]$ у потпуности садржан у сегменту $[a, b]$, тј. ако је $a \leq x$ и $y \leq b$, тада је допринос потпун, тј. цео збир сегмента $[x, y]$ (а то је број уписан у

низу на позицији k) доприноси збиру елемената на позицијама из сегмента $[a, b]$.

- У супротном, сегменти се секу и тада је допринос текућег чвора једнак збиру доприноса његовог левог и десног детета.

Из овог разматрања следи наредна имплементација.

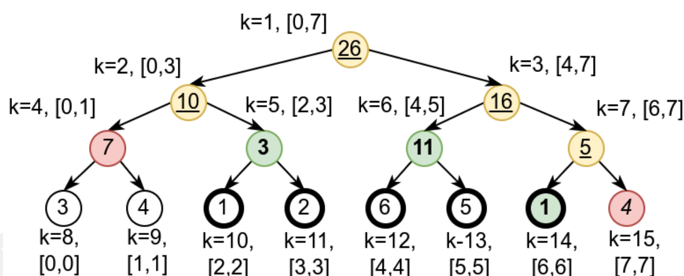
```
// izračunava se zbir onih elemenata polaznog niza, koji se nalaze na
// pozicijama iz segmenta [a, b] i koji se ujedno nalaze u delu
// segmentnog drveta ciji je koren u nizu drvo na poziciji k (taj deo
// drveta pokriva elemente na pozicijama segmenta [x, y] originalnog niza)
int zbirSegmenta(const vector<int>& drvo, int k, int x, int y, int a, int b) {
  // segmenti [x, y] i [a, b] su disjunktni
  if (b < x || a > y) return 0;
  // segment [x, y] je potpuno sadržan unutar segmenta [a, b]
  if (a <= x && y <= b)
      return drvo[k];
  // segmenti [x, y] i [a, b] se seku
  int s = (x + y) / 2;
  return zbirSegmenta(drvo, 2*k, x, s, a, b) +
         zbirSegmenta(drvo, 2*k+1, s+1, y, a, b);
}

// izračunava se zbir elemenata polaznog niza na pozicijama iz
// segmenta [a, b], na osnovu segmentnog drveta smeštenog u nizu drvo
int zbirSegmenta(const vector<int>& drvo, int a, int b) {
  int n = drvo.size() / 2;
  // krećemo od drveta smeštenog od pozicije 1, koje
  // pokriva elemente polaznog niza na pozicijama iz segmenta [0, n-1]
  return zbirSegmenta(drvo, 1, 0, n-1, a, b);
}
```

Пример 1.4.7

Размотримо како се овим приступом одређује збир елемената на позицијама из сегмента $[2, 6]$ *у оригиналном низу (у сегментном дрвету приказаном на слици 1.16, ти елементи су на позицијама* $[2+8, 6+8] = [10, 14]$*)*

- *Извршавање креће од корена. Сегмент* $[0, 7]$ *се сече са сегментом* $[2, 6]$ *те ће збир бити једнак суми доприноса сегмената* $[0, 3]$ *и* $[4, 7]$.
- *Сегмент* $[0, 3]$ *се сече са сегментом* $[2, 6]$ *те опет правимо два рекурзивна позива за сегменте* $[0, 1]$ *и* $[2, 3]$.

- *Сегмент* $[0, 1]$ *је дисјунктан са сегментом* $[2, 6]$ *па је његов допринос траженој суми* 0.
- *Сегмент* $[2, 3]$ *је садржан у сегменту* $[2, 6]$ *те је његов допринос потпун – једнак је вредности* 3 *коју тај чвор чува.*
- *С друге стране, сегмент* $[4, 7]$ *се сече са сегментом* $[2, 6]$ *те опет правимо два рекурзивна позива за сегменте* $[4, 5]$ *и* $[6, 7]$.
- *Сегмент* $[4, 5]$ *је у потпуности садржан у сегменту* $[2, 6]$ *те је његов допринос потпун и износи* 11.
- *Сегмент* $[6, 7]$ *се сече са сегментом* $[2, 6]$*, па из њега стартујемо два рекурзивна позива: за сегменте* $[6, 6]$ *и* $[7, 7]$.
- *Сегмент* $[6, 6]$ *је у потпуности садржан у сегменту чији збир рачунамо, те је његов допринос потпун и износи* 1.
- *Сегмент* $[7, 7]$ *је дисјунктан са сегментом чији збир рачунамо, па је његов допринос једнак нула. Дакле, укупна вредност суме сегмента биће једнака* $3 + 11 + 1 = 15$.

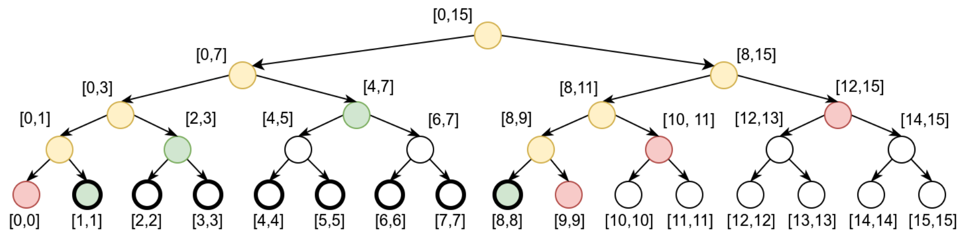


Слика 1.16: Рачунање збира елемената из сегмента $[2, 6]$ приступом одозго наниже. Жутом бојом и подвученим бројевима су означени чворови који одговарају сегментима који се секу са траженим сегментом, црвеном бојом и искошеним бројевима чворови који су дисјунктни, а зеленом бојом и подебљаним бројевима чворови који одговарају сегментима који су у потпуности садржани у сегменту који се сабира.

И ова имплементација има сложеност $O(\log n)$. Докажимо то.

Лема 1.4.1 **[Број посећених чворова на сваком нивоу]**

Приликом рекурзивног обиласка сегментног дрвета, за сваки ниво сегментног дрвета обилазе се највише по четири чвора.



Слика 1.17: Рачунање збира елемената из сегмента $[1, 8]$ приступом одозго наниже. Жутом бојом су означени чворови који одговарају сегментима који се секу са траженим сегментом, црвеном бојом чворови који су дисјунктни, а зеленом бојом чворови који одговарају сегментима који су у потпуности садржани у сегменту који се сабира.

Доказ. Докажимо ово тврђење принципом математичке индукције.

На првом нивоу се посећује само један чвор, корен дрвета, тако да се на овом нивоу посећује мање од четири чвора и база индукције важи.

Размотримо сада произвољни ниво дрвета: према индуктивној хипотези на њему се посећује највише четири чвора.

- Ако се посећује највише два чвора, у наредном нивоу се посећује највише четири чвора јер сваки чвор може да произведе највише два рекурзивна позива.

- Претпоставимо да се на текућем нивоу посећују три или четири чвора. Они могу бити суседни (слика 1.16, ниво 2), али и не морају (слика 1.17, ниво 3). Пошто чворови на једном нивоу сегментног дрвета садрже суме дисјунктних сегмената, једини чворови из којих се могу покренути рекурзивни позиви су они који садрже границе сегмента чију суму рачунамо (сиви чворови на слици): остали сегменти ће бити или у потпуности садржани или дисјунктни са сегментом чију суму рачунамо (или зелени, или црвени на слици). Стога, на сваком нивоу постоје само два чвора из којих се могу направити рекурзивни позиви, тако да ће и наредни ниво задовољавати полазно тврђење.

□

Пошто је висина сегментног дрвета $O(\log n)$, укупно се посећује највише $4 \log n$ чворова сегментног дрвета, те је сложеност операције израчунавања збира сегмента приступом одозго наниже такође $O(\log n)$.

МАТФ Београд

Ажурирање вредности елемента

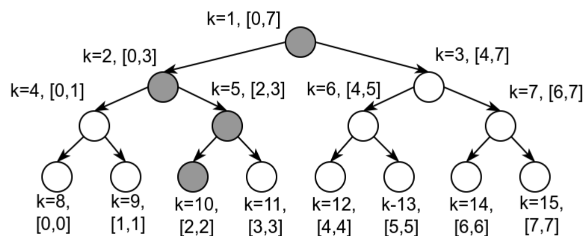
И функцију за ажурирање сегментног дрвета при ажурирању вредности неког елемента полазног низа можемо имплементирати итеративно и рекурзивно.

Итеративни поступак

Приликом ажурирања неког елемента потребно је ажурирати све чворове на путањи од тог листа до корена. С обзиром на то да знамо позицију родитеља сваког чвора, ова операција се може веома једноставно итеративно имплементирати.

Пример 1.4.8

На слици 1.18 је на једном примеру приказано које све чворове је потребно ажурирати након измене елемента на позицији 2 *у оригиналном низу.*



Слика 1.18: Ажурирање елемента на позицији 2 у полазном низу. Чворови чије се вредности ажурирају су обојени.

```
// ažurira segmentno drvo smešteno u niz drvo od pozicije 1
// koje sadrži elemente polaznog niza a dužine n u kom su elementi
// smešteni od pozicije 0, nakon što se na poziciju i polaznog
// niza upiše vrednost v
void upisi(vector<int>& drvo, int i, int v) {
  int n = drvo.size() / 2;
  // prvo ažuriramo odgovarajući list
  int k = i + n;
  drvo[k] = v;
  // ažuriramo sve roditelje izmenjenih čvorova
  for (k /= 2; k >= 1; k /= 2)
    drvo[k] = drvo[2*k] + drvo[2*k+1];
}
```

Пошто се k полови у сваком кораку петље, а креће од вредности највише $2n-1$, и сложеност ове операције је $O(\log n)$.

Рекурзивни поступак

И ову операцију можемо имплементирати одозго наниже.

```
// ažurira segmentno drvo smešteno u niz drvo od pozicije k,
// koje sadrži elemente polaznog niza a dužine n sa pozicija iz
// segmenta [x, y], nakon što se na poziciju i niza upiše vrednost v
void upisi(vector<int>& drvo, int k, int x, int y, int i, int v) {
   if (x == y)
     // ažuriramo vrednost u listu
     drvo[k] = v;
   else {
     // proveravamo da li se pozicija i nalazi levo ili desno
     // i u zavisnosti od toga ažuriramo odgovarajuće poddrvo
     int s = (x + y) / 2;
     if (x <= i && i <= s)
       upisi(drvo, 2*k, x, s, i, v);
     else
       upisi(drvo, 2*k+1, s+1, y, i, v);
     // pošto se promenila vrednost u nekom od dva poddrveta
     // moramo ažurirati vrednost u korenu
     drvo[k] = drvo[2*k] + drvo[2*k+1];
   }
}

// ažurira segmentno drvo smešteno u niz drvo od pozicije 1,
// koje sadrži elemente polaznog niza a dužine n u kom su elementi
// smešteni od pozicije 0, nakon što se na poziciju i polaznog
// niza upiše vrednost v
void upisi(vector<int>& drvo, int i, int v) {
  int n = drvo.size() / 2;
  // krećemo od drveta smeštenog od pozicije 1 koje
  // sadrži elemente polaznog niza na pozicijama iz segmenta [0, n-1]
  upisi(drvo, 1, 0, n-1, i, v);
}
```

Сложеност претходне имплементације можемо описати рекурентном једначином: $T(n) = T(n/2) + O(1), T(1) = O(1)$ и њено решење износи $O(\log n)$. Наиме, дужина интервала $[x, y]$ се у сваком наредном позиву смањује два пута, а његова почетна дужина је једнака n.

Уместо функције `upisi` често се разматра функција `uvecaj` која елемент на позицији `i`

полазног низа увећава за дату вредност `v` и у складу са тим ажурира сегментно дрво. Свака од ове две функције се једноставно изражава преко оне друге.

Друге операције

Осим за проналажење збира, сегментно дрво се може користити и за многе друге асоцијативне операције. Најчешће су то проналажење минимума или максимума, НЗД или НЗС свих елемената сегмента и слично. У тим случајевима се на свим местима у коду операција сабирања мења одговарајућом бинарном асоцијативном операцијом (`min`, `max`, `nzd`, `nzs` и слично). Притом, ако се низ допуњава до дужине која је степен броја 2, уместо вредности 0, користе се неутрални елементи одговарајућих операција (на пример, за `min` то је вредност $+\infty$, за `max` то је $-\infty$, за `nzd` то је 0, а за `nzs` то је 1).

Неке мало напредније модификације омогућавају и извршавање сложенијих упита. На пример, можемо да направимо дрво које за сваки сегмент одређује максималну вредност тог сегмента и уједно број појављивања те максималне вредности у том сегменту. Тада се у сваком чвору чува пар (m, n) сачињен од те две вредности. У листовима дрвета се чува уређен пар елемента низа и вредности 1. Парови се комбинују на следећи начин. Ако је вредност у левом детету чвора (m_l, n_l), а у десном (m_d, n_d), тада, ако је $m_l > m_d$, важи $(m, n) = (m_l, n_l)$, ако је $m_d > m_l$, важи $(m, n) = (m_d, n_d)$, а ако је $m_l = m_d$, важи $(m, n) = (m_l, n_l + n_d)$. Ако дужина низа није степен броја 2, низ се допуњава вредностима $(-\infty, 0)$, јер је $-\infty$ неутрал за операцију `max`, при чему се та вредност заправо не јавља у низу (ако се зна да су сви елементи низа ненегативни, уместо $-\infty$ се може користити вредност 0).

Наведимо још неке типичне операције: одређивање броја елемената у сегменту који задовољавају дати услов (на пример, одређивање броја нула у сваком сегменту или броја парних бројева у сваком сегменту), одређивање позиције k-тог по реду елемента у сегменту који задовољава дати услов, одређивање позиције првог елемента у сегменту који је већи од дате вредности, одређивање прве позиције у сегменту низа који садржи само ненегативне вредности такве да је збир префикса сегмента до те позиције већи или једнак од дате вредности и слично.

Напоменимо да се неке од ових операција могу реализовати помоћу обичног сегментног дрвета и бинарне претраге тако да им је сложеност $O(\log^2 n)$, а прилагођавањем сегментног дрвета сложеност постаје $O(\log n)$. На пример, у низу ненегативних бројева за дати сегмент одређен позицијама $[l, d]$, бинарном претрагом интервала $[l, d]$ можемо одредити најмању вредност $k \in [l, d]$ тако да је збир префикса одређеног позицијама $[l, k]$ већи од дате вредности x. Бинарна претрага захтева $O(\log{(d - l + 1)}) = O(\log n)$ корака, а у сваком кораку збир сегмента одређеног позицијама $[l, k]$ можемо одређивати помоћу сегментног дрвета у времену $O(\log n)$ (јер је префикс одређен позицијама $[l, k]$ сегмента одређеног позицијама $[l, d]$ уједно сегмент оригиналног низа). Са

МАТФ Београд

друге стране, можемо дефинисати рекурзивну функцију која у једном проласку кроз сегментно дрво проналази тражену вредност k. Наиме, ако је вредност левог детета (збира елемената у левој половини тренутног сегмента) већа или једнака од вредности x, онда префикс рекурзивно тражимо у левом детету, а ако је мања од вредности x, онда у десном детету тражимо најкраћи префикс чији је збир већи или једнак од разлике вредности x и вредности левог детета (што је збир елемената у левој половини тренутног сегмента). Тиме се упит извршава у времену $O(\log n)$.

1.4.2.2 Фенвикова дрвета (BIT)

Размотримо сада још једну структуру података која омогућава ефикасно рачунање статистика сегмената низа и ажурирање појединачних елемената: то су *Фенвикова*[11] *дрвета* (енгл. Fenwick tree), тј. *бинарно индексирана дрвета* (енгл. binary indexed tree, BIT). Видећемо да се она мало једноставније имплементирају од сегментних дрвета, користе мало мање меморије и могу да буду мало бржа од њих (иако су им и временска и просторна сложеност асимптотски једнаке). С друге стране, за разлику од сегментних дрвета, која су погодна за различите операције, Фенвикова дрвета су специјализована само за асоцијативне операције које имају одговарајуће инверзне (супротне) операције (нпр. збир или производ елемената сегмената се може налазити уз помоћ Фенвикових дрвета, јер сабирању одговара одузимање, а множењу дељење, али не и минимум, највећи заједнички делилац и слично). Потребу да за разматрану операцију постоји инверзна операција ћемо детаљније дискутовати када будемо говорили о реализацији самих операција. Сегментна дрвета могу да ураде све што и Фенвикова, док обратно не важи.

Слично као што је случај код сегментних дрвета, иако се назива дрветом, Фенвиково дрво заправо представља низ вредности збирова неких паметно изабраних сегмената оригиналног низа a. Избор сегмената је у тесној вези са бинарном репрезентацијом индекса. Кључна Фенвикова идеја је следећа:

Као што се сваки природан број може добити сабирањем степена двојке, одређених његовом бинарном репрезентацијом (свака јединица у бинарној репрезентацији одређује неки степен двојке), тако се и сваки префикс низа може добити надовезивањем неких сегмената, одређених бинарном репрезентацијом границе тог сегмента (свака јединица у бинарној репрезентацији одређује један такав сегмент).

Поново ћемо, једноставности ради, претпоставити да се вредности смештају од позиције 1 (вредност на позицији 0 је ирелевантна) и то и у полазном низу a, и у низу у ком се смешта Фенвиково дрво. Прилагођавање кода ситуацији у којој су у полазном низу елементи смештени од позиције нула веома је једноставно, само је на почетку сваке функције која ради са дрветом индекс полазног низа потребно увећати за један

[11] Структуру је иницијално предложио Борис Ријабко, а име је добила по Питеру Фенвику, информатичару са Универзитета Окланд у Новом Зеленду.

МАТФ Београд

пре даље обраде. Дакле, ако је полазни низ дужине n, елементи дрвета се смештају у посебан низ на позиције $[1, n]$.

Дефиниција Фенвиковог дрвета је следећа:

У дрвету се на позицији k чува збир вредности полазног низа са позиција из интервала $(f(k), k]$, где је $f(k)$ број који се добија од броја k тако што се у бинарном запису броја k прва јединица здесна замени нулом.

Пример 1.4.9

- *У Фенвиковом дрвету се на позицији $k = 21$ записује збир елемената полазног низа са позиција из интервала $(20, 21]$. Наиме, број 21 се бинарно записује као $(10101)_2$ и заменом прве јединице здесна нулом у његовом бинарном запису добија се бинарни запис $(10100)_2$, тј. број 20 (важи $f(21) = 20$).*
- *На позицији 20 у Фенвиковом дрвету налази се збир елемената полазног низа са позиција из интервала $(16, 20]$, јер се брисањем крајње десне јединице из његовог бинарног записа $(10100)_2$ добија бинарни запис $(10000)_2$ тј. број 16 (важи $f(20) = 16$).*
- *На позицији 16 у Фенвиковом дрвету чува се збир елемената полазног низа са позиција из интервала $(0, 16]$, јер се брисањем крање десне јединице из бинарног записа броја 16 добија 0 (важи $f(16) = 0$).*

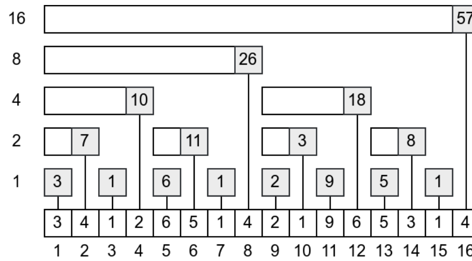
На слици 1.19 је за низ дужине 16 приказано који се збирови сегмената полазног низа чувају на свакој од позиција у Фенвиковом дрвету. Приметимо да непарне позиције одговарају сегментима дужине 1, а да за степене броја 2 сегменти представљају префиксе полазног низа до те позиције.

Пример 1.4.10

Размотримо низ a са вредностима $3, 4, 1, 2, 6, 5, 1, 4$.

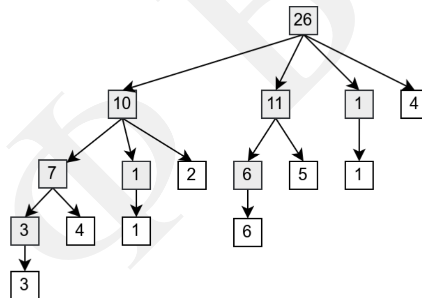
0	1	2	3	4	5	6	7	8	k
	0001	0010	0011	0100	0101	0110	0111	1000	*k бинарно*
	0000	0000	0010	0000	0100	0100	0110	0000	*$f(k)$ бинарно*
	0	0	2	0	4	4	6	0	*$f(k)$*
	$(0,1]$	$(0,2]$	$(2,3]$	$(0,4]$	$(4,5]$	$(4,6]$	$(6,7]$	$(0,8]$	*интервал*
	3	4	1	2	6	5	1	4	*низ a*
	3	7	1	10	6	11	1	26	*Фенвиково дрво*

МАТФ Београд



Слика 1.19: Приказ сегмената чији су збирови елемената смештени у елементима Фенвиковог дрвета дужине 16. На дну је приказан полазни низ, сегменти су означени правоугаоницима, а збир елемената сваког сегмента је приказан у сивом квадрату који се налази на позицији на којој је тај збир смештен у Фенвиковом дрвету када се оно представи помоћу низа.

На слици 1.20 приказана је зависност збирова сегмената од њихових подсегмената, одакле се јасно види да је заиста у питању дрволика структура (отуда и потиче назив Фенвиково дрво). У листовима ове структуре налазе се елементи полазног низа, док се у унутрашњим чворовима налазе елементи Фенвиковог дрвета.



Слика 1.20: Дрволика структура Фенвиковог дрвета.

Имплементација разних операција над Фенвиковим дрветом је веома једноставна, ако се зна начин да се из бинарног записа броја уклони прва јединица здесна тј. да се за дати број k израчуна вредност $f(k)$. Означимо са $b(k)$ бинарни број који садржи само једну јединицу и то на месту последње јединице у бинарном запису броја k. Важи да је $f(k) = k - b(k)$.

Под претпоставком да су бројеви записани у потпуном комплементу, вредност $b(k)$ се може израчунати изразом `k & -k`. Одузимањем те вредности од броја k тј. изразом `k - (k & -k)` добијамо ефекат брисања последње јединице у бинарном запису броја k и то представља имплементацију функције f.

МАТФ Београд

Пример 1.4.11

Броју $k = 20$ одговара бинарни запис $(00010100)_2$, а броју $-k$ бинарни запис $(11101100)_2$, те је `k & -k` једнако $(00000100)_2$ и представља број који садржи само једну јединицу и то на месту последње јединице у запису броја k. Одузимањем ове вредности од броја k добијамо број 16 коме одговара бинарна репрезентација $(00010000)_2$.

Докажимо исправност операције уклањање последње јединице из бинарног записа броја.

Лема 1.4.2 **[Рачунање броја $f(k)$]**

За било који неозначени бинарни број k већи од нуле изразом `k & -k` се израчунава број $b(k)$ тј. број који се састоји само од последње јединице у бинарном запису броја k, а изразом `k - (k & -k)` се израчунава број $f(k)$ добијен од броја k брисањем последње јединице из његовог бинарног записа.

Доказ. Потпуни комплемент броја k се добија тако што се броју k инвертују сви битови и добијени резултат се увећа за 1. Нека број k има своју последњу јединицу на позицији p (ако има битова десно од позиције p, они су сви нуле). Када се негирају битови добија се број који има нулу на позицији p иза које до краја следе јединице (ако их уопште има). Сабирањем са 1 добија се број који на позицији p има 1 иза чега се налазе нуле. Дакле, лево од позиције p битови броја k и $-k$ су супротни, на позицији p оба имају јединице, а десно од позиције p оба имају нуле. Битовска конјункција бројева k и $-k$ даје резултат $b(k)$ који има само једну јединицу и то на позицији p (то је једино место где и k и $-k$ имају вредност 1). □

Други начин да се израчуна вредност $f(k)$ јесте рачунањем вредности израза `k & (k-1)`. Исправност овог израза се такође може једноставно доказати. Наиме, број `k-1` има све исте битове од крајњег левог до позиције последњег постављеног бита у броју k, а све инвертоване битове после крајњег десног постављеног бита у броју k.

Пример 1.4.12

Бинарни запис броја $k = 20$ је $(00010100)_2$, а броја $k - 1 = 19$ је $(00010011)_2$ и изразу `k & (k-1)` одговара бинарни запис $(00010000)_2$.

Ипак, израз `k - (k & -k)` се чешће користи, због сличности са изразом `k + (k & -k)`, који се користи приликом израчунавања функције g која је дефинисана у поглављу 1.4.2.2 посвећеном ажурирању вредности елемената низа.

МАТФ Београд

Рачунање збира елемената сегмента

Збир елемената у било ком префиксу полазног низа може да се добије као збир неколико елемената записаних у Фенвиковом дрвету.

Пример 1.4.13

Интервал позиција $(0, 21]$ *тј. префикс низа до позиције* 21 *се добија надовезивањем интервала* $(0, 16]$, $(16, 20]$ *и* $(20, 21]$.

Ове интервале је једноставно одредити. На позицији k се налази збир елемената са позиција из интервала $(f(k), k]$, на позицији $f(k)$ збир елемената са позиција из интервала $(f(f(k)), f(k)]$, итд. Поступак се наставља сабирајући елементе са позиција k, $f(k)$, $f(f(k))$, $f(f(f(k)))$ итд., све док се не дође до позиције нула.

Број елемената Фенвиковог дрвета чијим се сабирањем добија збир префикса полазног низа је $O(\log n)$. Наиме, у сваком кораку се број јединица у бинарном запису текућег индекса смањује, а број n се бинарно записује са највише $O(\log n)$ бинарних јединица.

Збир елемената префикса на позицијама из интервала $(0, k]$ полазног низа a смештеног у Фенвиково дрво `drvo` можемо онда израчунати наредном функцијом.

```
// na osnovu Fenvikovog drveta smeštenog u niz drvo
// izračunava zbir prefiksa (0, k] polaznog niza
int zbirPrefiksa(const vector<int>& drvo, int k) {
  int zbir = 0;
  while (k > 0) {
     zbir += drvo[k];
     k -= k & -k; // k = f(k)
  }
}
```

Када умемо да израчунамо збирове префикса полазног низа, збир произвољног сегмента одређеног позицијама $[a, b]$ полазног низа можемо израчунати као разлику збира префикса одређеног позицијама $(0, b]$ и збира префикса одређеног позицијама $(0, a-1]$. Пошто се оба префикса рачунају у времену $O(\log n)$, и збир сваког сегмента можемо израчунати у времену $O(\log n)$. Истакнимо на овом месту да је због ове операције важно да асоцијативна операција која се користи у Фенвиковом дрвету има инверзну операцију (у овом случају, пошто се ради о операцији сабирања, да бисмо могли да одузимањем две вредности префикса добијемо збир произвољног сегмента[12]).

[12] Обратимо пажњу да ако се Фенвиковим дрветом рачунају производи сегмената, елементи дрвета морају бити различити од нуле, због операције дељења.

Ажурирање вредности елемента

Идеју рачунања збира елемената произвољног сегмента као разлике два збира префикса смо видели већ раније, када смо у те сврхе одржавали низ свих збирова префикса. Основна предност Фенвикових дрвета у односу на низ свих збирова префикса је то што се могу ефикасно ажурирати. Наиме, ажурирање једног елемента полазног низа може резултовати изменом вредности великог броја префиксних збирова, нарочито ако је елемент који мењамо близу почетка низа (у најгорем случају број збирова које је потребно ажурирати је $O(n)$). Стога је операција ажурирања елемента када се користе префиксни збирови у најгорем случају сложености $O(n)$.

Размотримо функцију која ажурира Фенвиково дрво након увећања елемента у полазном низу на позицији k за вредност x. Тада је за x потребно увећати све оне збирове у дрвету у којима се као сабирак јавља и елемент на позицији k. Потребно је ажурирати све оне позиције m чији придружени сегмент садржи вредност k, тј. све оне позиције m такве да је $k \in (f(m), m]$, тј. позиције m за које важи

$$f(m) < k \leq m \tag{1.1}$$

Те позиције се израчунавају веома једноставно кренувши од полазне позиције k, слично као у претходној функцији, с тим што се уместо да се од броја k одузима вредност $b(k)$, број k увећава за вредност израза $b(k)$. Обележимо са $g(k)$ број $k+b(k)$, који се добија од броја k тако што се k сабере са бројем који има само једну јединицу у свом бинарном запису и то на позицији на којој се налази последња јединица у бинарном запису броја k. У имплементацији се број $g(k)$ лако може израчунати по формули `k + (k & -k)`.

Пример 1.4.14

За број $k = (101100)_2$ *важи* $g(k) = (101100)_2 + (100)_2 = (110000)_2$.

Доказаћемо да је потребно и довољно ажурирати вредности на позицијама k, $g(k)$, $g(g(k))$ итд., све док се не дође до позиције која је строго већа од дужине низа n.

Пример 1.4.15

Ако би се у претходном примеру елемент на позицији 3 *у полазном низу увећао за вредност* 4, *било би потребно повећати за* 4 *вредности елемената Фенвиковог дрвета на позицијама* 3, 4 *и* 8. *До низа ових позиција бисмо могли да дођемо на следећи начин.*

- *Прва позиција је број* $k = 3$.

МАТФ Београд

- *Наредну позицију $g(3)$ добијамо тако што бинарном запису броја 3 који износи $(0011)_2$, додајемо број 1 (број $b(3)$ који садржи тачно једну јединицу на позицији последње јединице у бинарном запису датог броја) и добијамо бинарни запис $g(3) = (0100)_2$ који одговара броју 4.*
- *Наредну позицију $g(g(3)) = g(4)$ бисмо добили тако што бисмо вредност 4 сабрали са $(0100)_2$ (бројем $b(4)$ који садржи тачно једну јединицу у свом бинарном запису и то на позицији последње јединице), чиме бисмо добили $g(4) = (1000)_2$ (бинарни запис броја 8). Овде се процедура завршава пошто смо стигли до последњег елемента у Фенвиковом дрвету.*

С обзиром на то да се број n бинарно записује са $O(\log n)$ битова и да се у сваком кораку број крајњих нула у запису броја повећава (иако у сваком кораку вредност броја расте), укупан број корака је једнак $O(\log n)$. Дакле, број елемената Фенвиковог дрвета чије је вредности потребно изменити износи $O(\log n)$.

```
// ažurira Fenvikovo drvo smešteno u niz drvo nakon što se
// u originalnom nizu element na poziciji k uveća za x
void uvecaj(vector<int>& drvo, int n, int k, int x) {
  while (k <= n) {
    drvo[k] += x;
    k += k & -k;
  }
}
```

Докажимо коректност описаног поступка. Кренимо од наредне леме.

Лема 1.4.3

За свако x између 1 и n, најмањи број m који задовољава услов $f(m) < x < m$ је $g(x)$.

Доказ. Покажимо најпре да $g(x)$ задовољава овај услов. Очигледно важи $x < g(x)$. Пошто $g(x)$ има све нуле од позиције последње јединице у бинарном запису броја x (укључујући и њу), па до краја, брисањем његове последње јединице тј. израчунавањем вредности $f(g(x))$ се сигурно добија број који је строго мањи од x.

Покажимо сада да је $g(x)$ најмањи број строго већи од x који задовољава дати услов, односно да ниједан број m између x и $g(x)$ не може да задовољи услов $f(m) < x$. Наиме, сви бројеви из интервала $(x, g(x))$ се поклапају са бројем x на свим позицијама пре крајњих нула, а на позицијама крајњих нула броја x имају бар неку јединицу, чијим се брисањем добија број који је већи или једнак x. □

МАТФ Београд

Пример 1.4.16

У претходно разматраном примеру за $k = (101100)_2$*, као што смо већ видели важи* $g(k) = (110000)_2$ *и једини бројеви између* k *и* $g(k)$ *су* $(101101)_2$*,* $(101110)_2$ *и* $(101111)_2$*. Брисањем последње јединице у бинарном запису ових бројева добијају се редом бројеви* $(101100)_2$*,* $(101100)_2$ *и* $(101110)_2$ *и сви они су већи или једнаки од* k*.*

Докажимо и другу помоћну лему.

Лема 1.4.4

За свако $1 \leq x \leq n$ *важи* $f(g(x)) \leq f(x)$*.*

Доказ. Функцијом g се први низ узастопних јединица на који се наилази здесна у запису броја x мења нулама, а нула испред њих мења у јединицу, која се затим применом функције f враћа на нулу. С друге стране, ако је тај низ јединица дужи од 1 применом функције f на број x остаје нека од тих јединица, па је $f(g(x)) < f(x)$. Ако је низ најдеснијих јединица дужине 1, добија се $f(x) = f(g(x))$.

Формално, важи да је $f(g(x)) = g(x) - b(g(x)) = x + b(x) - b(g(x))$ и $f(x) = x - b(x)$. Зато је довољно доказати да је $x + b(x) - b(g(x)) \leq x - b(x)$ тј. да је $2b(x) \leq b(g(x))$. Нека $b(x)$ има јединицу на позицији p. Број $2b(x)$ има јединицу на позицији $p + 1$ (бројано здесна). Број $g(x)$ сигурно има све нуле од позиције p до краја, па се његова последња јединица налази или на позицији $p + 1$ или лево од ње, на основу чега лако следи да је он већи или једнак од $2b(x)$, који има јединицу на позицији $p + 1$. □

Докажимо сада и главно тврђење.

Теорема 1.4.1

Низ вредности k*,* $g(k)$*,* $g(g(k))$*, итд. које су мање или једнаке од дужине низа* n *одређује све вредности* m *такве да важи* $k \in (f(m), m]$ *тј. такве да важи једнакост (1.1).*

Доказ. Једнакост (1.1) не може да важи за бројеве $m < k$, а сигурно важи за број $m = k$, јер је $f(k) < k$ када је $k > 0$ (а ми претпостављамо да је $1 \leq k \leq n$). Дакле прва позиција у дрвету коју треба ажурирати је позиција k.

За све бројеве $m > k$, сигурно важи десна неједнакост и једино је потребно утврдити да ли важи лева тј. да ли је $f(m) < k$. На основу примене леме 1.4.3 на број k знамо да је $g(k)$ најмањи број m строго већи од k за који важи $f(m) < k$.

МАТФ Београд

Даље, на основу примене леме 1.4.3 на број $g(k)$ знамо да је најмањи број већи од $g(k)$ за који важи $f(m) < g(k)$ број $g(g(k))$. Он задовољава услов (1.1). Заиста, важи $k < g(k) < g(g(k))$. На основу леме 1.4.4 примењене на $g(k)$ важи је $f(g(g(k))) \leq f(g(k)) < k$. Број $g(g(k))$ је и најмањи број који већи од $g(k)$ који задовољава услов (1.1). Ако би неки број m између $g(k)$ и $g(g(k))$ задовољио услов (1.1), важило би да је $f(m) < k < g(k)$, што је у супротности са тиме да је $g(g(k))$ најмањи број m већи од $g(k)$ такав да је $f(m) < g(k)$.

Доказ се даље наставља и завршава по потпуно истом принципу (што се може формализовати индукцијом). □

Претходна теорема гарантује да су једине позиције које треба ажурирати у дрвету приликом ажурирања елемента на позицији k у полазном низу управо позиције из серије k, $g(k)$, $g(g(k))$, итд., све док су оне мање или једнаке n, па су претходни поступак и његова имплементација коректни.

Формирање Фенвиковог дрвета

Остаје још питање како иницијално формирати Фенвиково дрво. Формирање се може свести на то да се креира дрво попуњено само нулама, а да се затим увећава вредност једног по једног елемента низа претходном функцијом.

```
// na osnovu niza a u kom su elementi smešteni
// na pozicijama iz segmenta [1, n] formira Fenvikovo drvo
// i smešta ga u niz drvo (na pozicije iz segmenta [1, n])
vector<int> formirajDrvo(int n, const vector<int>& a) {
  vector<int> drvo(n+1);
  fill_n(drvo + 1, n, 0);
  for (int k = 1; k <= n; k++)
     uvecaj(drvo, n, k, a[k]);
  return drvo;
}
```

Пример 1.4.17

Размотримо проблем формирања Фенвиковог дрвета за низ вредности $3, 1, 9, 4, 6, 6, 2, 7$. *Кренувши од низа који садржи само нуле, увећавамо један по један елемент низа, како је приказано на слици 1.21.*

Ова имплементација n пута позива функцију `uvecaj` која је сложености $O(\log n)$, те је укупна сложеност овог алгоритма $O(n \log n)$. Међутим, показује се да можемо и боље од овог. Наиме, сваки елемент Фенвиковог дрвета садржи збир елемената неког

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

0	1	2	3	4	5	6	7	8
	0	0	0	0	0	0	0	0
	3	3	0	3	0	0	0	3
	3	4	0	4	0	0	0	4
	3	4	9	13	0	0	0	13
	3	4	9	17	0	0	0	17
	3	4	9	17	6	6	0	23
	3	4	9	17	6	12	0	29
	3	4	9	17	6	12	2	31
	3	4	9	17	6	12	2	38

Слика 1.21: Формирање Фенвиковог дрвета. Оригинални низ је представљен лево, а Фенвиково дрво десно.

сегмента полазног низа, па за израчунавање елемената Фенвиковог дрвета можемо искористити збирове префикса. Наиме, елемент на позицији k у Фенвиковом дрвету добијамо као разлику збира префикса $(0, k]$ и збира префикса $(0, f(k)]$, ако је $f(k) > 0$, док ако је $f(k) = 0$ елемент на позицији k биће једнак баш збиру префикса $(0, k]$.

```
vector<int> formirajDrvoPrefiksneSume(int n, const vector<int>& a) {
  vector<int> drvo(n+1);
  vector<int> sume_prefiksa(n+1,0);
  sume_prefiksa[1] = a[1];
  for (int k = 2; k <= n; k++)
    sume_prefiksa[k] = sume_prefiksa[k-1] + a[k];

  fill_n(drvo + 1, n, 0);
  for (int k = 1; k <= n; k++) {
    // racunamo f(k)
    int f_k = k - (k & -k);
    // od sume prefiksa do k oduzimamo sumu prefiksa do f(k)
    // ako je f(k)=0, onda ne vrsimo oduzimanje
    drvo[k] = sume_prefiksa[k] - (f_k > 0 ? sume_prefiksa[f_k] : 0);
  }
```

```
  return drvo;
}
```

Израчунавање збирова свих префикса полазног низа је временске сложености $O(n)$, али се након тога елементи Фенвиковог дрвета рачунају у времену $O(1)$, те је укупна временска сложеност овог приступа $O(n)$. Ова имплементација пак захтева додатни меморијски простор величине $O(n)$ за смештање префиксних збирова.

Приметимо да су за израчунавање k-тог елемента Фенвиковог дрвета потребни само елементи низа префиксних збирова на позицијама $j \leq k$. Ово опажање омогућава да користимо само један низ који ћемо инцијализовати на низ префиксних збирова, а затим почев од последњег елемента низа идући ка првом елементу мењати елемент по елемент са префиксног збира на елемент Фенвиковог дрвета.

```
vector<int> formirajDrvoPrefiksneSumeOpt(int n, const vector<int>& a) {
  vector<int> drvo(n+1);
  fill_n(drvo+1, n, 0);

  drvo[1] = a[1];
  for (int k = 2; k <= n; k++)
    drvo[k] = drvo[k-1] + a[k];

  for (int k = n; k >= 1; k--){
    int f_k = k - (k & -k);
    if (f_k > 0) drvo[k] -= drvo[f_k];
  }
  return drvo;
}
```

Ова имплементација алгоритма за формирање Фенвиковог дрвета је временске сложености $O(n)$, а додатне просторне сложености $O(1)$.

Задатак: Максимални подсегмент

Дат је низ целих бројева. Написати програм који омогућава извршавање следеће две врсте упита:

- Израчунавање максималног збира неког подсегмента датог сегмента одређеног позицијама $[a, b]$ (подсегмент је или празан или је одређен позицијама $[a', b']$ таквим да је $a \leq a' \leq b' \leq b$).
- Промена вредности елемента низа на некој датој позицији.

MATF Град

Опис улаза

Са стандардног улаза се учитава вредност n ($1 \leq n \leq 10^5$), а затим n целих бројева из интервала $[-100, 100]$, који представљају почетне вредности низа. Након тога се учитава природан број q ($1 \leq q \leq 10^5$) који представља број упита које је потребно обрадити. Након тога се учитава q упита. Упити могу бити следећег облика:

- `s a b` – потребно је одредити максималну вредност збира неког подсегмента сегмента одређеног позицијама $[a, b]$, за $0 \leq a \leq b < n$.
- `p k v` – потребно је елементу низа на позицији k доделити вредност v, за $0 \leq k < n$ и цео број v између -100 и 100.

Опис излаза

За сваки упит типа `s` на стандардни излаз исписати максималну вредност збира датог подсегмента.

Пример

Улаз

```
9
-2 1 -3 4 -1 2 1 -5 4
5
s 0 8
s 1 4
p 4 1
s 0 8
s 2 6
```

Излаз

```
6
4
8
8
```

Решење

Каданов алгоритам

Решење грубом силом подразумева да за сваки сегмент из почетка рачунамо вредност максималног збира неког његовог подсегмента. Чак и када се за то користи неки ефикасан алгоритам (на пример Каданов), ово решење је веома неефикасно. Ажурирање вредности низа врши се у времену $O(1)$, али се максимални подсегмент одређује у времену $O(n)$, па је сложеност најгорег случаја $O(qn)$.

Сегментно дрво

Ефикасно решење се може добити помоћу сегментног дрвета. Ако се неки сегмент подели на два мања подсегмента, његов максимални подсегмент може бити цео садржан у левом подсегменту, цео садржан у десном подсегменту или се састојати од максималног суфикса левог и максималног префикса десног подсегмента. Максимални префикс целог сегмента је или максимални префикс његовог левог подсегмента или садржи цео леви подсегмент и максимални префикс десног подсегмента. Слично, максимални су-

Матф Београд

фикс целог сегмента је или максимални суфикс његовог десног подсегмента или садржи цео десни подсегмент и максимални суфикс левог подсегмента. Зато сегментно дрво организујемо тако да сваки чвор садржи наредна 4 податка:

- збир свих елемената сегмента који одговара том чвору;
- максимални збир неког подсегмента сегмента који одговара том чвору;
- максимални збир префикса сегмента који одговара том чвору;
- максимални збир суфикса сегмента који одговара том чвору.

Формирање сегментног дрвета је сложености $O(n)$. Одређивање максималног збира подсегмента за дати сегмент захтева један пролаз кроз дрво и сложености је $O(\log n)$. Такође, и ажурирање елемента захтева један пролаз кроз дрво (одоздо навише) и сложености је $O(\log n)$. Дакле, сложеност овог приступа је $O(n + q \log n)$.

```
// cvor segmentnog drveta
typedef struct {
  int zbir;         // zbir segmenta
  int maksSegment; // maksimalni zbir podsegmenta
  int maksPrefiks; // maksimalni zbir prefiksa
  int maksSufiks;  // maksimalni zbir sufiksa
} Cvor;

// sve 4 vrednosti su inicijalizovane na nulu
Cvor nula = {};

// odredjuje cvor roditelja na osnovu poznatog levog i desnog deteta
Cvor kombinuj(const Cvor& l, const Cvor& d) {
  Cvor c;
  c.zbir = l.zbir + d.zbir;
  c.maksSegment = max({l.maksSegment, d.maksSegment,
                      l.maksSufiks + d.maksPrefiks});
  c.maksPrefiks = max({l.maksPrefiks, l.zbir + d.maksPrefiks});
  c.maksSufiks  = max({d.maksSufiks, d.zbir + l.maksSufiks});
  return c;
}

// kreira se list drveta na osnovu date vrednosti v
Cvor list(int v) {
  Cvor c;
  c.zbir = v;
```

```
  c.maksSegment = max(v, 0);
  c.maksPrefiks = max(v, 0);
  c.maksSufiks = max(v, 0);
  return c;
}

// formira segmentno drvo na osnovu datog niza
vector<Cvor> formirajDrvo(const vector<int>& a) {
  int n = stepenDvojke(a.size());
  vector<Cvor> drvo(2*n, nula);
  for (int k = 0; k < a.size(); k++)
    drvo[n + k] = list(a[k]);

  for (int k = n-1; k > 0; k--)
    drvo[k] = kombinuj(drvo[2*k], drvo[2*k+1]);
  return drvo;
}

// rekurzivna funkcija koja odredjuje doprinos cvora k u segmentnom
// drvetu (koji pokriva segment [x, y] u originalnom nizu) maksimalnom
// zbiru podsegmenta segmenta [a, b]
Cvor maksPodsegment(const vector<Cvor>& drvo,
                    int k, int x, int y, int a, int b) {
  // segmenti [x, y] i [a, b] su disjunktni pa ne doprinosi nista
  if (b < x || a > y)
    return nula;
  // segment [x, y] je ceo sadrzan unutar [a, b] pa se ceo uracunava
  if (a <= x && y <= b)
    return drvo[k];
  // segmenti [x, y] i [a, b] se preklapaju, pa se segment [x, y] deli
  // na dve polovine
  int s = (x + y) / 2;
  return kombinuj(maksPodsegment(drvo, 2*k, x, s, a, b),
                  maksPodsegment(drvo, 2*k+1, s+1, y, a, b));
}

// na osnovu segmentnog drveta odre]uje se maksimalni z
int maksPodsegment(const vector<Cvor>& drvo, int a, int b) {
  // pozivamo pomocnu rekurzivnu funkciju krenuvsi od korena drveta
```

```
  // koji se nalazi u cvoru 1 i pokriva ceo niz tj. segment [0, n-1]
  int n = drvo.size() / 2;
  Cvor c = maksPodsegment(drvo, 1, 0, n-1, a, b);
  return c.maksSegment;
}

// nakon upisivanja vrednosti v na poziciju k u originalnom nizu
// azurira se segmentno drvo
void azurirajDrvo(vector<Cvor>& drvo, int k, int v) {
  int n = drvo.size() / 2;
  // azuriramo list
  drvo[n+k] = list(v);
  // azuiramo sve njegove pretke
  int r = (n+k) / 2;
  while (r > 0) {
    drvo[r] = kombinuj(drvo[2*r], drvo[2*r+1]);
    r = r / 2;
  }
}
```

Задатак: Број инверзија

Напиши програм који одређује колико у низу има инверзија (позиција $0 \leq i < j < n$, таквих да је $a_i > a_j$).

Опис улаза

Са стандардног улаза се уноси број n ($1 \leq n \leq 10^5$) и затим n целих бројева, сваки у посебном реду.

Опис излаза

На стандардни излаз исписати само тражени број инверзија.

Пример

Улаз	*Излаз*
5 3 1 4 2 5	3

Решење

Један начин да се задатак ефикасно реши је да одржавамо структуру података у коју се ефикасно може уметнути нови елемент и којој се ефикасно може добити одговор на

МАТФ БЕОГРАД

питањем колико је елемената у структури строго мање од дате вредности. Ако имамо овакву структуру података на располагању, инверзије можемо пребројати на следећи начин. Низ у коме бројимо инверзије обилазимо са десног краја, додајући у структуру један по један елемент, након што га обрадимо. То значи да ће у тренутку обраде било ког елемента структура садржати тачно оне елементе који су у оригиналном низу десно од њега. За сваки елемент проверавамо колико је елемената у структури мање од њега, увећавамо бројач инверзија за тај број и након тога умећемо елемент у дрво.

Једна таква структура је Фенвиково дрво над низом који на позицији x чува број појављивања вредности x у структури. Бројање елемената мањих од x се онда своди на израчунавање префиксне суме до $x - 1$, а додавање елемента x се своди на увећавање вредности на позицији x за 1 (и ажурирање одговарајућих збирова у дрвету). Напоменимо да се у описаном алгоритму низ обилази здесна налево, јер нам Фенвиково дрво једноставније даје одговор на то колико је елемената мање од дате вредности, него колико је елемената веће од дате вредности.

Пошто величина Фенвиковог дрвета (па и меморијска, али и временска сложеност операција) зависи од вредности највећег елемента у њему, а нама за број инверзија нису важне апсолутне вредности елемената, него само њихов међусобни однос, пре примене алгоритма можемо сваки елемент заменити са његовим рангом у сортираном низу (исти елементи могу да имају исти ранг). Ово је могуће урадити, на пример, тако што сортирамо низ, а затим бинарном претрагом за сваки елемент пронађемо прву позицију његовог појављивања у сортираном низу.

```
long long brojInverzija(vector<int>& a) {
  int n = a.size();
  // svaki element u nizu a menjamo rangom tj. prvom pozicijom na
  // kojoj se javlja u sortiranom redosledu (brojimo pozicije od 1)
  vector<int> b = a;
  sort(begin(b), end(b));
  for (int i = 0; i < n; i++)
    a[i] = distance(begin(b), lower_bound(begin(b), end(b), a[i])) + 1;

  // Fenvikovo drvo
  vector<int> drvo(n+1, 0);
  // broj inverzija
  long long broj = 0;
  for (int i = n-1; i >= 0; i--) {
    // odredjujemo koliko elemenata u drvetu je strogo manje od a[i]
    broj += zbirPrefiksa(drvo, a[i]-1);
```

```
    // dodajemo a[i] u drvo
    dodaj(drvo, a[i], 1);
  }
  return broj;
}
```

Задатак: К-ти парни број

Написати програм који омогућава да се у низу природних бројева који је на почетку испуњен нулама, али чији се елементи често мењају током извршавања програма ефикасно проналази позиција k-тог парног броја по реду.

Опис улаза

Са стандардног улаза се учитава дужина низа n ($1 \leq n \leq 50000$), а затим и број упита m ($1 \leq m \leq 50000$). Извршавањем упита облика $u\ p\ x$ се у низ на позицију $1 \leq p \leq n$ уписује број x, док се упитом $c\ k$ на стандардни излаз исписује позиција k-тог парног броја у текућем садржају низа (позиције се броје од 1).

Опис излаза

На стандардном излазу приказати резултате извршавања упита `c`. Ако у неком случају у низу има мање парних бројева од вредности k, тада уместо позиције исписати `-`.

Пример

Улаз

```
5
8
u 3 1
c 4
u 1 7
u 2 5
c 1
u 1 2
c 2
c 4
```

Излаз

```
5
4
4
-
```

Објашњење

- Низ на почетку садржи 5 нула
- Након извршавање упита `u 3 1` низ садржи елементе `0 0 1 0 0`.
- Упитом `c 4` се израчунава позиција четвртог парног броја у низу и то је 5.
- Након извршавања упита `u 1 7`, па затим и `u 2 5` садржај низа је `7 5 1 0 0`.
- Упитом `c 1` се израчунава позиција првог парног броја у низу и то је 4.
- Након извршавања упита `u 1 2` садржај низа је `2 5 1 0 0`.
- Упитом `c 2` се израчунава позиција другог парног броја у низу и то је 4.
- Упитом `c 4` се израчунава позиција четвртог парног броја у низу. Он не постоји (низ садржи 3 парна броја: 2, 0 и 0).

МАТФ Београд

Решење

Директно решење подразумева да се елементи уписују у низ и да се позиција k-тог парног одређује применом линеарне претраге. Ако имамо m_1 операција ажурирања и m_2 операција претраге сложеност таквог решења је $O(m_1 + m_2 \cdot n)$, што може бити прилично неефикасно.

Можемо формирати низ нула и јединица такав да се јединице налазе на месту парних елемената у полазном низу. Тада је позиција k-тог по реду парног броја најмања позиција таква да је збир свих јединица закључно са том позицијом једнак k. Ако низ нула и јединица одржавамо у Фенвиковом или сегментном дрвету, врло ефикасно можемо да израчунавамо збирове сваког фиксираног префикса. Захваљујући чињеници да су збирови префикса монотоно неопадајући (када се префикси продужавају), тражену позицију можемо ефикасно одредити алгоритмом бинарне претраге. Ако имамо m_1 операција ажурирања и m_2 операција претраге укупна сложеност ће бити $O(m_1 \log n + m_2 \log^2 n)$, тј. $O(m \log^2 n)$.

```
// vraca prvu poziciju p tako da je zbir niza na pozicijama [1, p]
// veci ili jednak k
int prefiksK(const vector<int>& drvo, int k) {
  // poziciju pronalazimo binarnom pretragom po vrednosti zbira prefiksa
  int l = 1, d = drvo.size() - 1;
  while (l <= d) {
    int s = l + (d - l) / 2;
    if (zbirPrefiksa(drvo, s) < k)
      l = s + 1;
    else
      d = s - 1;
  }
  return l;
}

int main() {
  int n;
  cin >> n;
  // gradimo Fenvikovo drvo i inicijalizujemo ga jedinicama
  vector<int> drvo(n + 1, 0);
  for (int k = 1; k <= n; k++)
    dodaj(drvo, k, 1);
  // elementi niza
```

```
  vector<int> niz(n + 1, 0);

  // broj upita
  int m;
  cin >> m;
  for (int i = 0; i < m; i++) {
    char c;
    cin >> c;
    if (c == 'u') {
      // upit upisa elementa u niz
      int p, x;
      cin >> p >> x;
      if (x % 2 == 0 && niz[p] % 2 != 0)
        // upisan je novi paran element na poziciju p
        dodaj(drvo, p, 1);
      else if (x % 2 != 0 && niz[p] % 2 == 0)
        // upisan je novi neparan element na poziciju p
        dodaj(drvo, p, -1);
      niz[p] = x;
    } else if (c == 'c') {
      // upit odredjivanja k-tog parnog elementa
      int k;
      cin >> k;
      // trazimo najkraci prefiks ciji je zbir jednak k
      int p = prefiksK(drvo, k);
      // proveravamo da li takav prefiks zaista postoji
      if (p <= n)
        cout << p << "\n";
      else
        cout << "-" << "\n";
    }
  }
  return 0;
}
```

Задатак: Број различитих елемената у сегментима

Напиши програм који одређује број различитих елемената у сваком од m датих сегмената низа од n елемената.

Опис улаза

Са стандардног улаза се учитава број n ($1 \leq n \leq 50000$), а затим n целих бројева $a_1, \dots, a_n$, чије су вредности између 1 и 100000. Након тога се учитава број m ($1 \leq m \leq 50000$) а затим у наредних m редова лева и десна граница затвореног сегмента $[l_i, d_i]$, раздвојене са по једним размаком ($1 \leq l_i \leq d_i \leq n$).

Опис излаза

На стандардни излаз исписати m бројева од којих сваки представља број различитих елемената у сегменту $a_{l_i}, \dots, a_{d_i}$.

Пример

Улаз	*Излаз*
`5`	`3`
`1 1 2 1 3`	`2`
`3`	`3`
`1 5`	
`2 4`	
`3 5`	

Решење

Решење грубом силом подразумева да за сваки сегмент избројимо различите елементе (на пример, коришћењем структуре података скуп), што је прилично неефикасно.

Основна идеја ефикасног решења задатка је да је број различитих елемената низа једнак броју последњих појављивања тих елемената. Можемо креирати бинарни низ који садржи јединице на местима на којима се елемент јавља последњи пут и нуле на местима на којима се налазе елементи који се у том сегменту јављају и касније и број различитих елемената тог низа добити сабирањем тог бинарног низа.

Пример 1.4.18

На пример, ако је дато 10 *елемената низа* $3, 4, 1, 3, 2, 5, 4, 7, 2, 2$*, тада можемо направити бинарни низ* $0, 0, 1, 0, 0, 1, 1, 1, 0, 1$. *Збир његових елемената је* 5*, што значи да у оригиналном низу постоји* 5 *различитих елемената.*

Додатно, за сваки сегмент позиција облика $[l, n)$, тј. за сваки суфикс низа, број различитих елемената у том суфиксу можемо добити израчунавањем броја јединица у одговарајућем суфиксу бинарног низа. Заиста, за сваку групу једнаких елемената постоји само једна јединица и сваки елемент се броји само једном. За сваки елемент који припада суфиксу одређеном позицијама $[l, n)$ постоји бар једна јединица у том сегменту која му одговара (ако је тренутно појављивање елемента последње, онда је јединица на

МАТФ Београд

његовом месту, а ако није, онда сигурно постоји јединица негде иза њега која му одговара). Приметимо да то не мора да важи за сегменте који се не завршавају на последњој позицији у низу.

Пример 1.4.19

Збир последња три елемента у бинарном низу из претходног примера је 2*, што значи да у оригиналном низу постоје* 2 *различита елемента (то су* 7 *и* 2*).*

Бинарни низ можемо креирати инкрементално током проласка полазног низа слева надесно. Наиме, за сваки нови елемент полазног низа на крај низа додајемо јединицу (ово његово појављивање је сигурно последње). Додатно, проверавамо да ли се тај елемент раније појављивао и ако јесте проналазимо позицију његовог ранијег последњег појављивања и мењамо је у нулу. Позиције последњих појављивања елемената можемо чувати у засебној мапи.

Ово нам указује на то да би добро било унете сегменте сортирати на основу десних крајева и обрађивати их у том редоследу. Низ нула и јединица можемо чувати у Фенвиковом или сегментном дрвету, што нам омогућава да ефикасно ажурирамо појединачне вредности и одређујемо збирове његових суфикса.

Пример 1.4.20

Прикажимо како се извршава пример из поставке задатка. Упите обрађујемо у сортираном редоследу на основу десног краја.

- *Прво се обрађује упит* $[2, 4]$*. То значи да се креира бинарни низ закључно са позицијом* 4 *из оригиналног низа.*
  - *Додајемо елемент* 1 *(са позиције* 1*) и бинарни низ је* $[1]$*.*
  - *Додајемо елемент* 1 *(са позиције* 2*) и бинарни низ постаје* $[0, 1]$*.*
  - *Додајемо елемент* 2 *(са позиције* 3*) и бинарни низ постаје* $[0, 1, 1]$*.*
  - *Додајемо елемент* 1 *(са позиције* 4*) и бинарни низ постаје* $[0, 0, 1, 1]$*.*

  Упит се извршава сабирајући елементе са позиција $[2, 4]$ *у бинарном низу, чиме се добија резултат* 2*.*
- *Сада се обрађује упит* $[1, 5]$*. Ово захтева да се у бинарни низ дода и елемент са позиције* 5 *из оригиналног низа.*
  - *Додајемо елемент* 3 *(са позиције* 4*) и бинарни низ постаје* $[0, 0, 1, 1, 1]$*.*

  Упит се изршава сабирајући елементе са позиција $[1, 5]$ *у бинарном низу, чиме се добија резултат* 3*.*

МАТФ Београд

- *На крају се обрађује упит* $[3, 5]$. *Бинарни низ је већ проширен до позиције* 5.

  Упит се израшава сабирајући елементе са позиција $[3, 5]$ *у бинарном низу, чиме се добија резултат* 3.

На крају исписујемо резултате упита у редоследу у ком су унети (а не у редоследу у ком су обрађени).

Приметимо да смо одговарање на упите одложили за крај програма, што нам је омогућило да све упите учитамо и затим сортирамо. Ова техника се некада назива *офлајн обрада упита* (енгл. offline queries) и иако се понекада може употребити за ефикасно решење задатка, у реалним ситуацијама она није увек примењива (у неким применама се одговор на сваки упит тражи одмах чим се упит постави).

```
// ucitavamo elemente u niz a (od pozicije 1)
// ...
// ucitavamo broj upita i upite
struct Upit {
  int l, d, i;
};

int m;
cin >> m;
vector<Upit> upiti(m);
for (int i = 0; i < m; i++) {
  cin >> upiti[i].l >> upiti[i].d;
  upiti[i].i = i;
}

// sortiramo upite po desnom kraju
sort(begin(upiti), end(upiti),
     [](const auto& u1, const auto& u2) {
       return u1.d < u2.d;
     });

// prethodna pozicija na kojoj se javlja data vrednost iz niza
unordered_map<int, int> prethodna_pozicija;
// Fenvikovo drvo
vector<int> drvo(n + 1, 0);
// za svaki upit se cuva rezultat upita
```

```
vector<int> rezultat(m);
int tekuci_upit = 0;
for (int i = 1; tekuci_upit < upiti.size() && i <= n; i++) {
  auto it = prethodna_pozicija.find(a[i]);
  if (it != prethodna_pozicija.end()) {
    dodaj(drvo, it->second, -1);
    it->second = i;
  } else {
    prethodna_pozicija[a[i]] = i;
  }
  dodaj(drvo, i, 1);
  while (tekuci_upit < upiti.size() && upiti[tekuci_upit].d == i) {
    rezultat[upiti[tekuci_upit].i] =
      zbirSegmenta(drvo, upiti[tekuci_upit].l, i);
    tekuci_upit++;
  }
}

// ispisujemo rezultate upita smestene u niz rezultat
// ...
```

1.4.3 Ажурирање сегмената

И сегментна и Фенвикова дрвета подржавају ефикасно израчунавање збирова одређених сегмената низа (енгл. range query), па самим тим и одређивање појединачних вредности низа (енгл. point query), као и ажурирање[13] појединачних елемената низа (енгл. point update), док ажурирање целих сегмената низа одједном (енгл. range update) није директно подржано. Наиме, ако се оно сведе на појединачно ажурирање свих елемената унутар сегмента, добија се лоша сложеност (у најгорем случају врши се n ажурирања која имају појединачну сложеност $O(\log n)$, па је укупна сложеност $O(n \log n)$). Са друге стране, чување низа разлика омогућава ефикасно ажурирање целих сегмената (енгл. range update), па самим тим и појединачних елемената (енгл. point update), али не и ефикасно одређивање појединачних елемената низа (енгл. point query) нити збирова сегмената (енгл. range query).

[13]Под ажурирањем се подразумева или постављање вредности елемента на дату вредност или увећање тренутне вредности елемента за дату вредност. У наставку ћемо разматрати само увећање тренутне вредности.

	point query	*range query*	*point update*	*range update*
Разлике суседних елемената	лоше / $O(n)$	лоше / $O(n)$	добро / $O(1)$	добро / $O(1)$
Префиксни збирови	добро / $O(1)$	добро / $O(1)$	лоше / $O(n)$	лоше / $O(n)$
Фенвиково дрво	добро / $O(\log n)$	добро / $O(\log n)$	добро / $O(\log n)$	лоше / $O(n \log n)$
Сегментно дрво	добро / $O(1)$	добро / $O(\log n)$	добро / $O(\log n)$	лоше / $O(n \log n)$

У наставку ћемо видети неколико начина да се направе структуре које ефикасно подржавају све наведене операције.

1.4.3.1 Дрво над низом разлика

Проблем

Дефинисати структуру података која обезбеђује ефикасно ажурирање сегмената датог низа одређених позицијама $[a, b]$ *(*range update*), па самим тим и ажурирање појединачних елемената низа (*point update*) увећавањем свих елемената за дату вредност, као и ефикасно одређивање вредности појединачних елемената низа (*point query*).*

Приметимо да се не захтева могућност ефикасног одређивања статистика сегмената (*range query*).

Основна идеја решења је да се одржава низ разлика суседних елемената полазног низа и да се тај низ разлика чува у Фенвиковом дрвету (или, аналогно, у сегментном дрвету).

- Увећавање свих елемената сегмената полазног низа за неку вредност v, своди се на промену два елемента низа разлика, што се уз одржавање дрвета може урадити у времену $O(\log n)$.
- Реконструкција елемента полазног низа на основу низа разлика своди се на израчунавање збира одговарајућег префикса, што се помоћу Фенвиковог дрвета може урадити веома ефикасно, у времену $O(\log n)$.

На овај начин, и ажурирање целих сегмената низа одједном и очитавање појединачних елемената можемо постићи алгоритмима сложености $O(\log n)$, што није било могуће само уз коришћење низа разлика (увећавања свих елемената неког сегмента је тада било сложености $O(1)$, али је очитавање вредности из низа било сложености $O(n)$).

МАТФ Београд

Дакле, ако Фенвиково или сегментно дрво изградимо над низом разлика уместо над оригиналним низом, губимо могућност ефикасног израчунавања статистика сегмената (*range query*), али добијамо могућност ажурирања сегмената (*range update*).

	point query	*range query*	*point update*	*range update*
Дрво разлика	добро / $O(\log n)$	лоше / $O(n \log n)$	добро / $O(\log n)$	добро / $O(\log n)$

1.4.3.2 Два дрвета разлика

На крају, описаћемо неколико структура података које омогућавају ефикасно извршавање све четири врсте упита.

Проблем

Дефинисати структуру података која обезбеђује ефикасно ажурирање сегмената датог низа одређених позицијама $[a, b]$ *(самим тим и ажурирање појединачних елемената низа) увећавањем свих елемената за дату вредност, као и ефикасно одређивање збирова сегмената одређених позицијама* $[a, b]$ *(самим тим и одређивање вредности појединачних елемената низа).*

Ефикасно увећање свих елемената у датом сегменту за исту вредност и израчунавање збирова сегмената могуће је имплементирати одржавањем два Фенвикова или сегментна дрвета.

Прво од њих ће чувати разлике оригиналног низа и омогућиће нам да ефикасно увећавамо сегменте и израчунавамо појединачне елементе оригиналног низа.

Ако је оригинални низ иницијализован нулама, након увећања свих елемената са позиција из сегмента $[a, b]$ за вредност v добијамо низ:

			a				b			
0	…	0	v	v	…	v	v	0	…	0

који се може представити низом разлика:

			a				b	$b+1$		
0	…	0	v	0	…	0	0	$-v$	…	0

МАТФ Београд

Сабирањем префикса низа разлика можемо добити било који појединачни елемент почетног низа (ако је низ разлика у Фенвиковом или сегментном дрвету, сложеност овог поступка је $O(\log n)$).

Циљ нам је да израчунамо префиксне збирове оригиналног низа (њега не можемо чувати у Фенвиковом или сегментном дрвету, зато што нам је потребно ажурирање његових сегмената, а не појединачних елемената).

Префиксни збирови оригиналног низа су:

			a				b			
0	…	0	v	$2v$	…	$(b-a)v$	$(b-a+1)v$	$(b-a+1)v$	…	$(b-a+1)v$

- Префиксни збирови P_k на позицјама k лево од позиције a (тј. $k < a$) једнаки су нули (јер су вредности A_k оригиналног низа лево од те позиције једнаки нули).
- Префиксни збирови P_k на позицијама k између a и b (тј. $a \leq k \leq b$) једнаки су $(k-a+1) \cdot v$, где је $v = A_k$ вредност која се налази на позицији k у оригиналном низу.
- Префиксни збирови P_k на позицијама k десно од позиције b (тј. $k > b$) једнаки су $(b-a+1) \cdot v$.

Дакле, важи:

$$P_k = \sum_{i=1}^{k} A_i = \begin{cases} 0 & k < a \\ (k-(a-1)) \cdot v & a \leq k \leq b \\ (b-a+1) \cdot v & k > b \end{cases}$$

Кључна идеја је да упоредимо ове тражене збирове са вредностима низа који на свакој позицији k садржи вредност $k \cdot A_k$, тј. да размотримо разлике $X_k = kA_k - P_k$.

- У првом случају (када је $k < a$) је $A_k = P_k = 0$, па је и $X_k = 0$.
- У другом случају (када је $a \leq k \leq b$) је $P_k = (k-a+1) \cdot A_k$, па је $X_k = kA_k - (k-a+1)A_k = (a-1)A_k$.
- У трећем случају (када је $k > b$) је $P_k = (b-a+1) \cdot v$, док је $A_k = 0$, па је $X_k = -(b-a+1) \cdot v$.

Другим речима, сваки збир префикса P_k смо разложили на разлику $kA_k - X_k$:

$$P_k = \sum_{i=1}^{k} A_i = k \cdot A_k - X_k = \begin{cases} k \cdot 0 - 0 & k < a \\ k \cdot v - (a-1) \cdot v & a \leq k \leq b \\ k \cdot 0 - (-(b-a+1)) \cdot v & k > b \end{cases}$$

МАТФ Београд

Низ X_k, дакле, након ажурирања оригиналног низа има следеће вредности:

			a			b			
0	$\ldots$	0	$(a-1)v$	$\ldots$	$(a-1)v$	$(a-1)v$	$-(b-a+1)v$	$\ldots$	$-(b-a+1)v$

Ако бисмо у сваком тренутку познавали вредности у низу X_k, тада бисмо вредности P_k лако могли израчунати помоћу формуле $P_k = kA_k - X_k$. Вредности A_k лако израчунавамо помоћу Фенвиковог или сегментног дрвета у коме чувамо разлике тог низа. Међутим, и низ X_k је такав да му се при сваком ажурирању повезани сегменти увећавају тј. умањују за исту вредност, тако да се и он може лако реконструисати ако би се његове разлике чувале у другом Фенвиковом или сегментном дрвету. Наиме, низ разлика овог низа се мења овако:

			a				b	$b+1$			
0	$\ldots$	0	$(a-1)v$	0	$\ldots$	0	0	$-bv$	0	$\ldots$	0

Дакле, одржавамо два дрвета: D_A у коме чувамо разлике низа A и D_X у коме чувамо разлике низа X. Прво иницијализујемо разликама низа A, а друго нулама.

Операција увећања свих елемената са позиција из $[a, b]$ за вредност v се своди на следеће операције над дрветима:

- *uvecaj*(D_A, a, v)
- ако је $b < n$ онда *uvecaj*$(D_A, b+1, -v)$
- *uvecaj*$(D_X, a, (a-1)v)$
- ако је $b < n$ онда *uvecaj*$(D_X, b+1, -bv)$

Операција израчунавање вредности низа A на позицији k тј. вредности A_k се своди на израчунавање префиксног збира првих k елемената Фенвиковог дрвета D_A.

- $A_k =$ *zbir_prefiksa*(D_A, k)

Операција израчунавања збира префикса елемената низа A дужине k се своди на израчунавање вредности

- $P_k = kA_k - X_k = k \cdot$ *zbir_prefiksa*$(D_A, k) -$ *zbir_prefiksa*(D_X, k)

Операција израчунавања зира сегмента $[a, b]$ низа A се своди на израчунавање вредности:

- $S_{ab} = P_b - P_{a-1}$

МАТФ Београд

При том је $P_0 = 0$.

Дакле, на овај начин можемо постићи добру сложеност за сва 4 типа операција.

	point query	*range query*	*point update*	*range update*
Два дрвета разлика	добро / $O(\log n)$	добро / $O(\log n)$	добро / $O(\log n)$	добро / $O(\log n)$

Пример 1.4.21

Прикажимо један пример употребе ове структуре података. Претпоставимо да низ A *има* 10 *елемената и да су у почетку сви једнаки нули. У програму би се одржавала само дрвета над низовима* D_A *и* D_X*, а илустрације ради ми ћемо приказати и садржај низа* A*, његових префиксних сума* P *и помоћног низа* X. *Елементи оригиналног низа (као и низова разлика) смештени су на позицијама од* 1 *до* 10.

	0	1	2	3	4	5	6	7	8	9	10
D_A	-	0	0	0	0	0	0	0	0	0	0
D_X	-	0	0	0	0	0	0	0	0	0	0
A	-	0	0	0	0	0	0	0	0	0	0
kA_k	-	0	0	0	0	0	0	0	0	0	0
X	-	0	0	0	0	0	0	0	0	0	0
P	0	0	0	0	0	0	0	0	0	0	0

Увећајмо елемент A_5 *за* 4. *То се може свести на увећавање елемената сегмента* $[a, b] = [5, 5]$ *за* $v = 4$.

	0	1	2	3	4	5	6	7	8	9	10
D_A	-	0	0	0	0	4	-4	0	0	0	0
D_X	-	0	0	0	0	16	-20	0	0	0	0
A	-	0	0	0	0	4	0	0	0	0	0
kA_k	-	0	0	0	0	20	0	0	0	0	0
X	-	0	0	0	0	16	-4	-4	-4	-4	-4
P	0	0	0	0	0	4	4	4	4	4	4

Умањимо све елементе низа A *за* 1. *То се може свести на увећавање елемената сегмента* $[a, b] = [1, 10]$ *за* $v = -1$.

МАТФ Београд

	0	1	2	3	4	5	6	7	8	9	10
D_A	-	-1	0	0	0	4	-4	0	0	0	0
D_X	-	0	0	0	0	16	-20	0	0	0	0
A	-	-1	-1	-1	-1	3	-1	-1	-1	-1	-1
kA_k	-	-1	-2	-3	-4	15	-6	-7	-8	-9	-10
X	-	0	0	0	0	16	-4	-4	-4	-4	-4
P	0	-1	-2	-3	-4	-1	-2	-3	-4	-5	-6

Одредимо $S_{[3,7]}$ *тј. збир елемената низа у сегменту* $[3, 7]$. *Важи да је* $S_{[3,7]} = P_7 - P_2$, *па је потребно да одредимо* P_7 *и* P_2.

P_7 *се одређује тако што се помоћу Фенвиковог дрвета ефикасно одреди* $A_7 = zbir_prefiksa(D_A, 7) = -1$, *затим* $X_7 = zbir_prefiksa(D_X, 7) = -4$ *и на крају се израчуна* $7A_7 - X_7 = 7 \cdot (-1) - (-4) = -3$.

P_2 *се одређује тако што се помоћу Фенвиковог дрвета ефикасно одреди* $A_2 = zbir_prefiksa(D_A, 2) = -1$, *затим* $X_2 = zbir_prefiksa(D_X, 2) = 0$ *и на крају се израчуна* $2A_2 - X_2 = 2 \cdot (-1) - 0 = -2$. *Тражени збир сегмента је онда* $-3 - (-2) = -1$. *Јасно је да је то тачан резултат, јер се у том сегменту налазе елементи* $[-1, -1, 3, -1, -1]$.

1.4.3.3 Лењо ажурирање сегментног дрвета

Опишимо сада још једно решење претходног проблема, тј. још једну структуру података која омогућава ефикасно извршавање све четири врсте упита. За разлику од претходне, ова структура омогућава и израчунавање неких других статистика (не само збира елемената).

Подсетимо се, сегментна дрвета омогућавају ефикасно ажурирање појединачних елемената и израчунавање збирова сегмената (самим тим и одређивање вредности појединачних елемената), међутим, не омогућавају ефикасно ажурирање свих елемената датог сегмента $[a, b]$ (тј. њихово увећање за дату вредност или њихову замену датом вредношћу). Ово се може постићи у времену $O(\log n)$ ако се на сегментно дрво примени техника *лење пропагације* (енгл. lazy propagation). Она је пример стратегије лењог извршавања којим се израчунавања одлажу све док одговарајуће вредности нису неопходне. Техника лење пропагације се доста ослања на рад са сегментним дрветом одозго наниже. Једноставности ради, технику ћемо представити кроз пример упита увећавања сегмената за дату вредност.

Сваки чвор у сегментном дрвету чува збир неког сегмента оригналног низа. Ако се тај сегмент у целости садржи унутар сегмента који се ажурира, можемо унапред изра-

МАТФ Београд

чунати за колико се повећава вредност у том чвору. Наиме, ако се свака вредност у сегменту повећава за v, тада се вредност збира тог сегмента повећава за $k \cdot v$, где је k број елемената у том сегменту. Вредност збира у том чвору тиме бива ажурирана у константном времену, али вредности збирова унутар поддрвета којима је тај чвор корен (укључујући и вредности у листовима које одговарају вредностима полазног низа) и даље остају неажурне. Њихово ажурирање захтевало би линеарно време, што је недопустиво скупо. Кључна идеја је да се ажурирање тих вредности одложи и да се оне не ажурирају одмах, већ само када затребају током неког накнадног упита, тј. током неке касније посете тим чворовима (која се иначе врши у склопу тог каснијег упита, а не посебно у циљу ажурирања вредности). Наиме, не желимо да те чворове посећујемо само због овог ажурирања, већ ћемо ажурирање урадити успут, током неке друге посете тим чворовима која би се свакако морала десити.

Поставља се питање како да сигнализирамо да вредности збирова у неком поддрвету нису ажурне и додатно оставимо упутство на који начин их треба ажурирати. У том циљу у сваком од чворова поред вредности збира сегмента чувамо и додатни *коефицијент лење пропагације*. Ако дрво у свом корену има коефицијент лење пропагације c који је различит од нуле, то значи да вредности збирова у целом том дрвету нису ажурне и да је сваки од листова тог дрвета потребно повећати за c и у односу на то ажурирати и вредности збирова у свим унутрашњим чворовима тог дрвета (укључујући и корен). Ажурирање се може одлагати све док вредност збира у неком чвору не постане заиста неопходна, а то је тек приликом упита израчунавања вредности збира неког сегмента. Ипак, вредности збирова у чворовима ћемо ажурирати и чешће и то заправо приликом сваке посете чвору – било у склопу операције увећања вредности из неког сегмента позиција полазног низа, било у склопу упита израчунавања збира неког сегмента. На почетку обе рекурзивне функције можемо проверити да ли је вредност коефицијента лење пропагације текућег чвора различита од нуле и ако јесте, ажурирати вредност збира у том чвору тако што ћемо га увећати за производ тог коефицијента и броја елемената који тај чвор покрива, затим коефицијенте лење пропагације оба његова детета увећати за тај коефицијент, а коефицијент лење пропагације тог чвора поставити на нулу (тиме корен дрвета који тренутно посећујемо постаје ажуран, а његовим поддрветима се даје упутство како их у будућности ажурирати). На овај начин се избегава ажурирање целог дрвета одједном, већ се ажурира само корен, што је операција сложености $O(1)$.

Размотримо како се може ажурирати чвор дрвета након увећања свих елемената неког сегмента. Инваријанта је да сви чворови у дрвету или садрже ажурне вредности збирова или су исправно обележени за каснија ажурирања (преко коефицијента лење пропагације). Након процеса ажурирања сегмента, чвор које се тренутно ажурира ће садржати актуелну вредност збира. Након почетног обезбеђивања да коефицијент лење пропагације у текућем чвору постане нула, могућа су три следећа случаја.

МАТФ Београд

- Ако је сегмент у текућем чвору дисјунктан у односу на сегмент који се ажурира, тада су сви чворови у поддрвету којем је он корен или већ ажурни или исправно обележени за касније ажурирање и није потребно ништа урадити.
- Ако је сегмент који одговара текућем чвору потпуно садржан у сегменту чији се елементи увећавају, тада се његова вредност ажурира (увећавањем за $k \cdot v$, где је k број елемената сегмента који одговара текућем чвору, а v вредност увећања), а његовој деци се коефицијент лење пропагације увећава за вредност v.
- На крају, ако се ова два сегмента секу, тада се прелази на рекурзивну обраду оба детета. Након извршавања функције над њима, сигурни смо да ће сви чворови у левом и десном поддрвету задовољавати услов инваријанте и да ће корени и левог и десног поддрвета имати ажурне вредности. Ажурну вредност у корену добијамо сабирањем вредности његова два детета.

Лењо сегментно дрво чуваћемо коришћењем два низа: `drvo`, у ком се чувају елементи сегментног дрвета и `lenjo`, у ком се чувају коефицијенти лење пропагације.

```
// ažurira elemente lenjog segmentnog drveta koje je smešteno
// u nizove drvo i lenjo od pozicije k u kome se čuvaju zbirovi
// elemenata originalnog niza sa pozicija iz segmenta [x, y]
// nakon što su u originalnom nizu svi elementi sa pozicija iz
// segmenta [a, b] uvećani za vrednost v
void promeni(vector<int>& drvo, vector<int>& lenjo, int k, int x, int y,
             int a, int b, int v) {
   // ažuriramo vrednost u korenu, ako nije ažurna
   if (lenjo[k] != 0) {
      drvo[k] += (y - x + 1) * lenjo[k];
      // ako nije u pitanju list, propagaciju prenosimo na decu
      if (x != y) {
         lenjo[2*k] += lenjo[k];
         lenjo[2*k+1] += lenjo[k];
      }
      // vrednost u korenu je sad ažurna
      lenjo[k] = 0;
   }
   // ako su intervali disjunktni, ništa nije potrebno raditi
   if (b < x || y < a) return;
   // ako je interval [x, y] sadržan u intervalu [a, b]
   // vrsimo uvecanje cvora za k * v
   if (a <= x && y <= b) {
```

```
      drvo[k] += (y - x + 1) * v;
      // ako nije u pitanju list, propagaciju prenosimo na decu
      if (x != y) {
         lenjo[2*k] += v;
         lenjo[2*k+1] += v;
      }
   } else {
      // u suprotnom se intervali seku,
      // pa rekurzivno obilazimo poddrveta
      int s = (x + y) / 2;
      promeni(drvo, lenjo, 2*k, x, s, a, b, v);
      promeni(drvo, lenjo, 2*k+1, s+1, y, a, b, v);
      // azurnu vrednost u korenu dobijamo kao zbir vrednosti dece
      drvo[k] = drvo[2*k] + drvo[2*k+1];
   }
}

// ažurira elemente lenjog segmentnog drveta koje je smešteno
// u nizove drvo i lenjo od pozicije 1 u kome se čuvaju zbirovi
// elemenata originalnog niza sa pozicija iz segmenta [0, n-1]
// nakon što su u originalnom nizu svi elementi sa pozicija iz
// segmenta [a, b] uvećani za vrednost v
void promeni(vector<int>& drvo, vector<int>& lenjo, int n,
             int a, int b, int v) {
   promeni(drvo, lenjo, 1, 0, n-1, a, b, v);
}
```

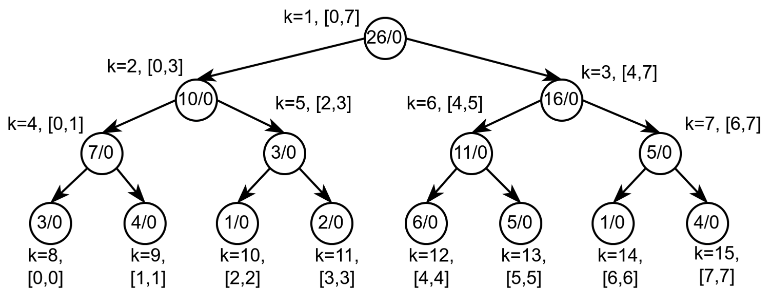
Сложеност функције ажурирања свих елемената датог сегмента у лењом сегментном дрвету износи $O(\log n)$. Наиме, као и у случају операције рачунања збира сегмента у сегментном дрвету, на сваком нивоу се разматра највише 4 чвора, те је максимални број чворова који се посећује једнак $4 \log n$.

Пример 1.4.22

Прикажимо рад ове функције на примеру лењоī сеīменшноī дрвеша са слике 1.22.

Прикажимо како бисмо све елеменше из сеīменша $[2, 7]$ увећали за 3.

- *Крећемо од корена који ӣокрива сеīменш $[0, 7]$. Сеīменши $[0, 7]$ и $[2, 7]$ се секу, ӣа сшоīа ажурирање ӣреӣушшамо деци и након њиховоī ажурирања, ӣри ӣовраšку из рекурзије вредносш корена одређујемо као збир ажурираних вредносши њеīове деце.*



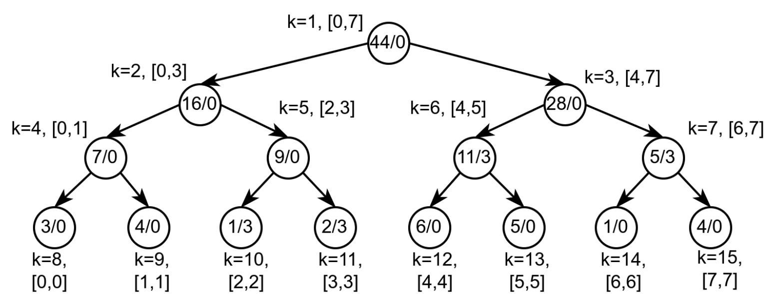
Слика 1.22: Лењо сегментно дрво: у сваком чвору чувамо вредности збира одговарајућег сегмента и коефицијента лење пропагације. Иницијално су вредности свих коефицијената лење пропагације једнаки нула.

- *На левој страни се сегмент* $[0, 3]$ *сече са сегментом* $[2, 7]$ *па и он препушта ажурирање наследницама и ажурира се тек при повратку из рекурзије.*
  - *Сегмент* $[0, 1]$ *је дисјунктан у односу на сегмент* $[2, 7]$ *и ту онда није потребно ништа радити.*
  - *Сегмент* $[2, 3]$ *је садржан у сегменту* $[2, 7]$*, и за њега директно знамо како се збир увећава: пошто овај чвор покрива два елемента и сваки се увећава за* 3*, збир овог сегмента се увећава укупно за* $2 \cdot 3 = 6$ *и поставља се на* 9*. У овом тренутку избегавамо ажурирање свих вредности у дрвету у ком је тај елемент корен, већ само наследницима уписујемо да је потребно пропагирати увећање за* 3*, али саму пропагацију одлажемо за тренутак када она постане неопходна.*
- *У повратку из рекурзије, вредност* 10 *ажурирамо и нова вредност је једнака* $7 + 9 = 16$*.*
- *Што се тиче десног поддрвета, сегмент* $[4, 7]$ *је садржан у сегменту* $[2, 7]$*, па можемо директно израчунати нову вредност збира у овом чвору. Наиме, пошто се* 4 *елемента увећавају за по* 3*, укупан збир се увећава за* $4 \cdot 3 = 12$*. Зато се вредност* 16 *мења у* $16 + 12 = 28$*. Пропагацију ажурирања кроз поддрво са кореном у овом чвору одлажемо и само његовој деци бележимо да је увећање за* 3 *потребно извршити у неком каснијем тренутку.*

• *При повратку из рекурзије вредност у корену ажурирамо са* 26 *на* $16 + 28 = 44$*.*

Након извршавања ових операција добија се дрво приказано на слици 1.23. Ово лењо сегментно дрво одговара низу $3, 4, 4, 5, 9, 8, 4, 7$*.*

МАТФ Београд



Слика 1.23: Лењо сегментно дрво након ажурирања свих елемената сегмента $[2, 7]$ за вредност 3.

МАТФ Београд

Претпоставимо да је у добијеном сегментном дрвету потребно елементе из сегмента $[0, 5]$ *увећавати за вредност* 2.

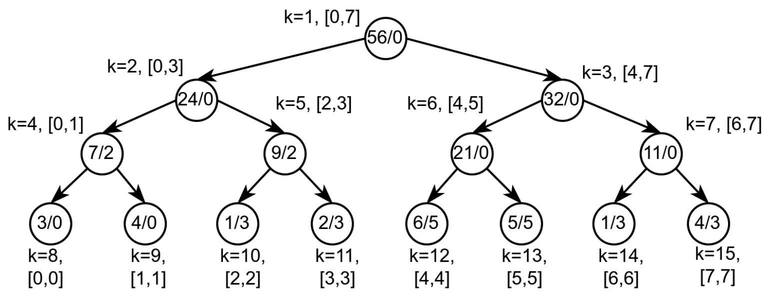
- *Поново се креће од корена лењог сегментног дрвета и када се установи да се сегмент* $[0, 7]$ *сече са сегментом* $[0, 5]$ *ажурирање се препушта деци и вредност у корену се ажурира тек при повратку из рекурзије.*
  - *Сегмент* $[0, 3]$ *је садржан у сегменту* $[0, 5]$*, па се вредност* 16 *увећава за* $4 \cdot 2 = 8$ *и поставља на* 24. *Поддрвета се не ажурирају одмах, већ се само њиховим коренима уписује да је све вредности потребно ажурирати за* 2.
  - *У десном поддрвету сегмент* $[4, 7]$ *се сече са* $[0, 5]$*, па се рекурзивно обрађују поддрвета.*
    - *При обради чвора са вредношћу* 11*, примећује се да је он требало да буде ажуриран јер је његов коефицијент лење пропагације различит од нула, међутим још није, па се најпре његова вредност ажурира и увећава за* $2 \cdot 3$ *и са* 11 *мења на* 17. *Његови наследници се не ажурирају одмах и њима се само уписује лења вредност* 3.

      Након тога се утврђује да се сегмент $[4, 5]$ *садржи у сегменту* $[0, 5]$*, па се вредност* 17 *увећава за* $2 \cdot 2 = 4$ *и поставља на* 21. *Поддрвета се не ажурирају одмах, већ само по потреби тако што се у њиховим коренима постави вредност лењог коефицијента. Пошто је у њима већ уписана вредност* 3*, она се сада увећава за* 2 *и поставља на* 5.
    - *Сада се прелази на обраду поддрвета у чијем је корену вредност* 5 *и пошто оно није ажурно, најпре се вредност* 5 *увећава за* $2 \cdot 3 = 6$ *и поставља на* 11*, а његовој деци се лењи коефицијент поставља на* 3.

Након тога се примећује да је сегмент $[6, 7]$ *дисјунктан са* $[0, 5]$ *и не ради се ништа.*

- *У повратку кроз рекурзију се ажурирају вредности родитељских чворова.*

Након ових операција добија се дрво приказано на слици 1.24. Ово лењо сегментно дрво одговара низу $5, 6, 6, 7, 11, 10, 4, 7$.



Слика 1.24: Лењо сегментно дрво након увећања свих елемената из сегмента $[0, 5]$ за 2.

Функција израчунавања вредности збира сегмента остаје практично непромењена, осим што се при уласку у сваки чвор врши његово ажурирање, ако је потребно. Стога и њена сложеност остаје непромењена и износи $O(\log n)$.

```
// na osnovu lenjog segmentnog drveta koje je smešteno
// u nizove drvo i lenjo od pozicije k u kome se čuvaju zbirovi
// elemenata polaznog niza sa pozicija iz segmenta [x, y]
// izračunava se zbir elemenata polaznog niza
// sa pozicija iz segmenta [a, b]
int saberi(vector<int>& drvo, vector<int>& lenjo, int k, int x, int y,
           int a, int b) {
  // ažuriramo vrednost u korenu, ako nije ažurna
  if (lenjo[k] != 0) {
     drvo[k] += (y - x + 1) * lenjo[k];
     if (x != y) {
       lenjo[2*k] += lenjo[k];
       lenjo[2*k+1] += lenjo[k];
     }
     lenjo[k] = 0;
  }
```

```
  // intervali [x, y] i [a, b] su disjunktni
  if (b < x || a > y) return 0;
  // interval [x, y] je potpuno sadržan unutar intervala [a, b]
  if (a <= x && y <= b)
      return drvo[k];
  // intervali [x, y] i [a, b] se seku
  int s = (x + y) / 2;
  return saberi(drvo, lenjo, 2*k, x, s, a, b) +
         saberi(drvo, lenjo, 2*k+1, s+1, y, a, b);
}

// na osnovu lenjog segmentnog drveta koje je smešteno
// u nizove drvo i lenjo od pozicije 1 u kome se čuvaju zbirovi
// elemenata polaznog niza sa pozicija iz segmenta [0, n-1]
// izračunava se zbir elemenata polaznog niza sa pozicija
// iz segmenta [a, b]
int saberi(vector<int>& drvo, vector<int>& lenjo, int n, int a, int b) {
  // računamo doprinos celog niza,
  // tj. elemenata iz intervala [0, n-1]
  return saberi(drvo, lenjo, 1, 0, n-1, a, b);
}
```

Приметимо да се у овој функцији дрво може мењати, па аргументи функције не смеју бити константни (иако се суштински вредности у дрвету не мењају, његова интерна репрезентација, тј. расподела података између низова `drvo` и `lenjo` се може мењати).

Пример 1.4.23

Прикажимо рад претходне функције на текућем примеру. Размотримо како се за дрво приказано на слици 1.24 израчунава збир елемената из сегмента $[3, 5]$.

- *Крећемо од корена дрвета које садржи суму сегмента* $[0, 7]$. *Сегмент* $[0, 7]$ *се сече са* $[3, 5]$, *па се рекурзивно обрађују деца.*
  - *У левом поддрвету сегмент* $[0, 3]$ *такође има пресек са* $[3, 5]$ *па прелазимо на наредни ниво рекурзије.*
    - *Приликом посете чвора у чијем је корену вредност* 7 *примећује се да његова вредност није ажурна, па се користи прилика да се она ажурира, тако што се увећа за* $2 \cdot 2 = 4$ *и постаје* 11, *а наследницима се лењи коефицијент поставља на* 2.

      Пошто је сегмент $[0, 1]$ *дисјунктан са* $[3, 5]$, *враћа се вредност* 0.

  - *Приликом посете чвора у чијем је корену вредност* 9 *примећује се да његова вредност није ажурна, па се користи прилика да се она ажурира, тако што се увећа за* $2 \cdot 2 = 4$ *и постаје* 13*, а наследницима се лењи коефицијент увећава за* 2*, односно постаје* 5*.*

    Сегмент $[2, 3]$ *се сече са* $[3, 5]$*, па се рекурзивно врши обрада поддрвета.*

    - *Вредност* 1 *се прво ажурира тако што се повећа за* $1 \cdot 5 = 5$ *и постаје* 6*, а онда, пошто је* $[2, 2]$ *дисјунктно са* $[3, 5]$ *враћа се вредност* 0*.*

    - *Вредност* 2 *се такође прво ажурира тако што се повећа за* $1 \cdot 5 = 5$ *и постаје* 7*, а пошто је сегмент* $[3, 3]$ *потпуно садржан у* $[3, 5]$ *враћа се вредност* 7*.*

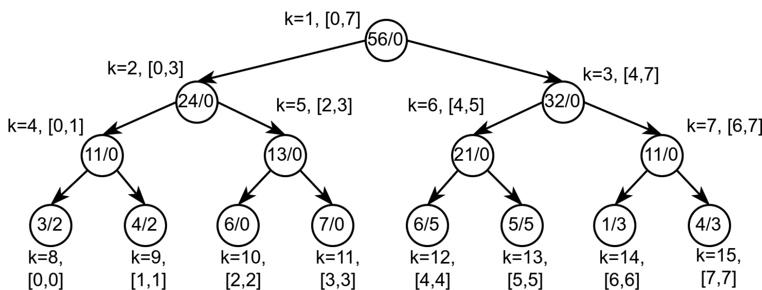
- *У десном поддрвету је чвор са вредношћу* 32 *ажуран, сегмент* $[4, 7]$ *се сече са* $[3, 5]$*, па се прелази на обраду наследника.*

  - *Чвор са вредношћу* 21 *је ажуран, сегмент* $[4, 5]$ *је садржан у* $[3, 5]$*, па се враћа вредност* 21*.*

  - *Чвор са вредношћу* 11 *је такође ажуран, али је сегмент* $[6, 7]$ *дисјунктан у односу на* $[3, 5]$*, па се враћа вредност* 0*.*

- *Дакле, чворови* 13 *и* 24 *враћају вредност* 7*, чвор* 32 *враћа вредност* 21*, па чвор* 56 *враћа вредност* $7 + 21 = 28$*.*

Дакле, збир сегмента $[3, 5]$ *у текућем дрвету је* 28*. Стање дрвета након извршавања упита је приказано на слици 1.25.*



Слика 1.25: Лењо сегментно дрво након рачунања збира елемената из сегмента $[3, 5]$.

Дакле, у лењом сегментном дрвету не можемо анализом појединачног чвора (нпр. листа) закључити која је тачна вредност тог чвора, међутим, када нам буде била потребна

вредност неког чвора у дрвету, ми ћемо се од корена спустити до тог чвора и, идући том путањом, све вредности на тој путањи ажурирати. На тај начин, када будемо стигли до жељеног чвора, имаћемо његову ажурну вредност, што је једино и важно.

	point query	*range query*	*point update*	*range update*
Лењо сегментно дрво	добро / $O(\log n)$	добро / $O(\log n)$	добро / $O(\log n)$	добро / $O(\log n)$

МАТФ Београд