МАТЕМАТИЧКА ГИМНАЗИЈА

МАТУРСКИ РАД

из предмета

Програмирање и програмски језици

на тему

Компаративна анализа приоритетних редова

Ученик:
Коста Грујчић, IV_b

Ментор:
Петар Величковић

Београд, мај 2017.

Садржај

1 Увод **1**

2 Употреба приоритетних редова у неким алгоритмима **2**
2.1 Дајкстрин алгоритам . . . 2
2.2 Примов алгоритам . . . 2

3 Хип **3**
3.1 Дефиниција и својства . . . 3
3.2 Подржане операције . . . 4
3.2.1 Проналазак чвора са најмањим кључем . . . 4
3.2.2 Уклањање чвора са најмањим кључем . . . 4
3.2.3 Додавање новог чвора . . . 5
3.2.4 Смањивање кључа произвољног чвора . . . 5
3.2.5 Уклањање произвољног чвора . . . 5
3.2.6 Иницијализација хипа . . . 6
3.2.7 Спајање два хипа . . . 7

4 Биномни хип **8**
4.1 Дефиниција и својства . . . 8
4.2 Подржане операције . . . 9
4.2.1 Проналазак чвора са најмањим кључем . . . 9
4.2.2 Спајање два биномна хипа . . . 9
4.2.3 Додавање новог чвора . . . 10
4.2.4 Уклањање чвора са најмањим кључем . . . 10
4.2.5 Смањивање кључа произвољног чвора . . . 11
4.2.6 Уклањање произвољног чвора . . . 11
4.2.7 Иницијализација биномног хипа . . . 11
4.3 MST алгоритам употребом биномног хипа . . . 12

5 Фибоначијев хип **13**
5.1 Дефиниције и својства . . . 13
5.2 Подржане операције . . . 14
5.2.1 Проналазак чвора са најмањим кључем . . . 14
5.2.2 Додавање новог чвора . . . 14
5.2.3 Спајање два Фибоначијева хипа . . . 15
5.2.4 Уклањање чвора са најмањим кључем . . . 15
5.2.5 Смањивање кључа произвољног чвора . . . 17
5.2.6 Уклањање произвољног чвора . . . 18
5.3 Горње ограничење највећег степена . . . 18

6 Стабло бинарне претраге **21**
6.1 Дефиниције и својства . . . 21
6.2 Подржане операције . . . 21
6.2.1 Проналазак чвора са најмањим кључем . . . 21
6.2.2 Проналазак чвора одређене вредности . . . 22
6.2.3 Додавање новог чвора . . . 22
6.2.4 Уклањање произвољног чвора . . . 22
6.2.5 Иницијализација стабла бинарне претраге . . . 23
6.2.6 Спајање два стабла бинарне претраге . . . 24
6.3 Очекивана висина бинарног стабла претраге . . . 25

7 Сплеј стабло **27**
7.1 Подржане операције . . . 27

8 Ван Емде Боасово стабло **29**
8.1 Мотивација . . . 29
8.2 Прото ван Емде Боасово стабло . . . 30
8.2.1 Дефиниције и својства . . . 30
8.2.2 Подржане операције . . . 30
8.2.2.1 Провера присуства елемента . . . 30
8.2.2.2 Проналазак најмањег елемента . . . 31
8.2.2.3 Проналазак следбеника произвољног елемента . . . 31
8.2.2.4 Додавање новог елемента . . . 32
8.2.2.5 Уклањање произвољног елемента . . . 32
8.3 Ван Емде Боасово Стабло – коначна верзија . . . 32
8.3.1 Дефиниција и својства . . . 32
8.3.2 Подржане операције . . . 33
8.3.2.1 Проналазак најмањег и највећег елемента . . . 33
8.3.2.2 Провера присуства елемента . . . 33
8.3.2.3 Проналазак следбеника произвољног елемента . . . 34
8.3.2.4 Проналазак претходника произвољног елемента . . . 34
8.3.2.5 Додавање новог елемента . . . 35
8.3.2.6 Уклањање произвољног елемента . . . 36
8.4 Временска и меморијска сложеност . . . 37

9 Евалуација **38**
9.1 Дајкстрин алгоритам . . . 38
9.2 Примов алгоритам . . . 40

10 Закључак **42**
10.1 Резултати . . . 42
10.2 Стечено знање . . . 42
10.3 Даљи рад . . . 42
10.4 Захвалност . . . 42

11 Литература **43**

1 Увод

У основи свих рачунарских програма се налази нека обрада података. Како је неретко количина података велика, потребно их је организовати у структуре података. Специјално, приоритетни редови су своју примену пронашли у проблемима који се тичу опслуживања захтева према некој важности. Први такви проблеми су се јавили много пре рачунара, а данас се користе у рачунарским мрежама за контролу протока, симулацији дискретних догађаја, разним алгоритмима и др.

Дефиниција 1.1. Приоритетни ред је скуп у ком је сваком елементу придружен елемент неког потпуно уређеног скупа као његов *приоритет*.

Приметимо да једноставне структуре података попут низа, листе, реда или стека, премда неефикасне, могу бити приоритетни редови.

Како се сваком чвору придружује приоритет, сматраћемо да вредност кључа чвора (или само вредност чвора) представља његов приоритет. Уколико је неопходно да поред приоритета чувамо још неку вредност, чвор можемо представити слогом, па би приоритетни ред био представљен скупом слогова.

Операције које ћемо описати за сваки приоритетни ред су:

- проналазак чвора са најмањим кључем (Find-Min),
- уклањање чвора са најмањим кључем (Delete-Min),
- додавање новог чвора (Insert),
- смањивање кључа произвољног чвора (Decrease-Key) и
- спајање два приоритетна реда у један (Merge).

Поред директне примене у пракси, многи приоритетни редови служе за теоријски оптимално извршавање појединих алгоритама. Значајан је број структура података које су осмишљене са баш овим циљем. Структуре података које су обрађене не морају нужно да буду приоритетни редови, али испуњавају све услове за то. Овде ће бити анализирани Дајкстрин и Примов алгоритам употребом d–хипа, биномног хипа, Фибоначијевог хипа, стабла бинарне претраге, сплеј стабла и ван Емде Боасовог стабла. Времена извршавања употребом поменутих приоритетних редова ћемо на крају упоредити и утврдити колико се теоријски оптимална структура података показује у пракси. Сваки приоритетни ред је детаљно описан, уз доказане сложености за сваку операцију.

Све приоритетне редове описујемо тако да се кључеви уређују по мањој вредности, па ће и дефиниције бити у складу са тим.

Задржавамо енглеску терминологију за скраћенице, попут MST – минимално разапињуће стабло, SSSP – најкраћи пут из једног извора итд.

Сви кодови су писани у програмском језику C++ и доступни су на приложеном CD-у.

2 Употреба приоритетних редова у неким алгоритмима

Изложићемо два алгоритма која се могу имплементирати употребом приоритетних редова и објаснити којим приоритетним редом је њихову асимптотску сложеност могуће свести на оптималну.

Сложеност сваког алгоритма изразићемо у облику

$$O\left(\sum_{i=1}^{|S|} k_i \cdot O(P_i)\right)$$

где је S скуп функција које се користе у имплементацији алгоритма, P_i сложеност и-те функције, а k_i број позивања и-те функције.

Сматрамо да је број чворова у графу n, а број грана m.

2.1 Дајкстрин алгоритам

Нека је $dist[v]$ удаљеност чвора v од изворног чвора $source$. Иницијално је $dist[source] = 0$, а за све остале чворове износи ∞[1].

На почетку иницијализујемо приоритетни ред у ком су кључеви одговарајуће $dist$ вредности. Функцију Delete-Min ћемо позвати укупно n пута јер ће сваки чвор бити обрађен тачно једном. Најмањи чвор приоритетног реда је чвор којим покушавамо да поправимо што је више могуће постојећих удаљености. За сваки такав чвор покушаћемо да смањимо растојање свих његових директних суседа. Када за неки чвор постигнемо мању удаљност морамо да ажурирамо прироритетни ред па фунцкију Decrease-Key позивамо $O(m)$ пута.

Сложеност је $O(O(\text{Init}) + n \cdot O(\text{Delete-Min}) + m \cdot O(\text{Decrease-Key}))$.

2.2 Примов алгоритам

Приоритетни ред градимо над гранама датог графа и то тако да су тежине грана кључеви приоритетног реда. Сваки чвор ћемо обрадити тачно једном јер сви чворови морају да се налазе у минималном разапињућем стаблу. Најмањи чвор представља грану најмање тежине која спаја неки од чворова у MST-у са неким од преосталих чворова. Грану која спаја два чвора MST-а игноришемо.

Сложеност је $O(m \cdot O(\text{Delete-Min}) + m \cdot O(\text{Insert}))$.

[1] У пракси је ту вредност могуће представити целобројним типом података који користимо за представљање приоритета у реду и означава вредност од које су сви други приоритети мањи. Аналогно важи и за $-\infty$.

3 Хип

Доналд Џонсон[2] је осмислио d – хип 1975. године са циљем да унапреди извршавање алгоритама као што су Дајкстрин (SSSP) и Примов (MST) алгоритам [4].

Обрадићемо опште, d – хипове, а како је степен d важан, приликом изражавања сложености неће бити занемарен. Сматрамо да се хип односи на d – хип и да је број d степен хипа.

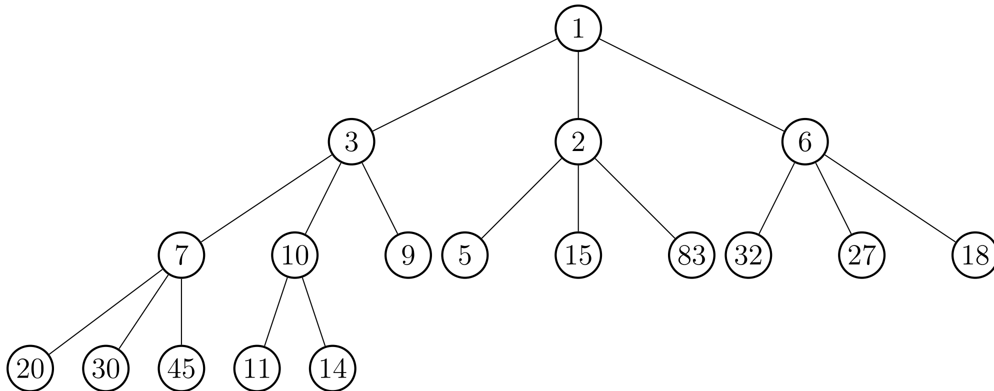
3.1 Дефиниција и својства

Дефиниција 3.1.1 (Хип својство)**.** Кључ сваког чвора, осим листова, је мањи од кључева своје деце.

Дефиниција 3.1.2. Уређено стабло у ком су сви кључеви различити и чворови имају степен d, осим листова и можда родитеља листова, и које испуњава хип својство је *хип*. На последњем нивоу сви постојећи чворови морају бити лево од оних који недостају.

Без умањења општости можемо претпоставити да су чворови хипа уређени у BFS поретку, да би хип могли једноставно да представимо. То можемо урадити помоћу низа, али би тада морали да прецизирамо величину низа приликом самог иницијализовања хипа. Зато се опредељујемо за употребу динамичког низа[3], јер се величина таквог низа може произвољно ширити. Такав низ ћемо означити са A. Подразумевамао да је динамички низ A индексиран од 0 и да се на тој позицији налази вредност корена хипа.

Приметимо да из хип својства следи да сваки чвор v мора имати кључ мањи од свих чворова у подстаблу са кореном v.



Слика 1: 3 – хип у ком су кључеви природни бројеви

Лема 3.1.1. Висина хипа је $O(\log_d n)$.

Доказ. Број чворова хипа је највећи уколико је стабло комплетно, па ћемо зато претпоставити да је хип комплетно d – стабло. Тада на основу формуле за збир првих m чланова геометријског низа важи

$$\sum_{k=0}^{h} d^k = \frac{d^{h+1}-1}{d-1} = n. \tag{3.1}$$

На крају добијамо тражено

$$h = O(\log_d nd) = O(1+\log_d n) = O(\log_d n). \tag{3.2}$$

[2]Donald B. Johnson – амерички информатичар. Најпознатији по раду из области дизајна и анализе алгоритама.

[3]Амортизоване сложености динамичког низа су $O(1)$.

□

Лема 3.1.2. Хип има највише $\frac{n}{d} + 1$ чворова који нису листови.

Доказ. Последњи чвор у хипу заузима позицију $n-1$, а његов родитељ позицију $\lfloor \frac{n-2}{d} \rfloor$. Како хип садржи n чворова, то су чворови на позицијама почев од $\lfloor \frac{n-2}{d} \rfloor + 1$ листови јер ником нису родитељи. Према томе, чворови на позицијама од 0 до $\lfloor \frac{n-2}{d} \rfloor$ нису листови. Њих укупно има $\lfloor \frac{n-2}{d} \rfloor + 1$. Трансформацијом израза добијамо тражено

$$\left\lfloor \frac{n-2}{d} \right\rfloor + 1 \leq \frac{n-2}{d} + 1 = \frac{n}{d} + \frac{d-2}{d} < \frac{n}{d} + 1. \tag{3.3}$$

□

Последица 3.1.2.1. До k – тог нивоа изнад нивоа листова постоји највише $\frac{n}{d^k} + 1$ чворова.

Имајући у виду дефиницију хипа, закључујемо да су за сваки чвор v вредности кључева деце $A[vd+1]$, $A[vd+2]$, ..., $A[vd+d]$. Уколико i – то дете чвора v не постоји, онда сматрамо да је $A[vd+i] = A[vd+i+1] = ... = A[vd+d] = \infty$, чак и када неке од поменутих позиција не постоје.

Будући да нам је за чување хипа od n чворова потребан динамички низ величине n, меморијска сложеност хипа је $O(n)$.

3.2 Подржане операције

Увешћемо функције Heap-Parent и Heap-Child.

```
Heap-Parent(x, d)
1  return ⌊(x − 1)/d⌋
```

```
Heap-Child(x, d, i)
1  return dx + i
```

3.2.1 Проналазак чвора са најмањим кључем

Како је најмања вредност у хипу она коју има корен, најмањи чвор можемо пронаћи у $O(1)$.

```
Heap-Find-Min(A)
1  return A[0]
```

3.2.2 Уклањање чвора са најмањим кључем

Уклањањем корена хипа хип постаје неповезан. Због тога морамо да нађемо нови чвор тако да стабло остане скоро пуно, а да хип својство не буде нарушено.

Да би стабло остало скоро пуно, узимамо сасвим десни лист хипа и мењамо га са кореном. Будући да желимо да уклонимо корен, довољно је да само обришемо последњи елемент низа A. Како је сада хип својство можда нарушено, потребно је да уредимо хип. То радимо „спуштањем" корена докле је то потребно низ хип. Проверавамо да ли је кључ чвора мањи од неког кључа деце, водећи рачуна да децу проверавамо идући са лева на десно, и прво дете које то испуњава постаје нови корен. Поступак понављамо док или не дођемо до листа стабла или док ни једно дете нема мањи кључ.

Уводимо функцију Heap-Fix-Down да би лакше уредили хип после уклањања чвора, а која ће нам касније такође користити.

```
Heap-Fix-Down(A, d, x)
1  for i = 1 to d
2      if A[Heap-Child(x, i)] < A[x]
3          swap(A[Heap-Child(x, d, i)], A[x])
4          Heap-Fix-Down(A, d, Child(x, d, i))
5          return
```

HEAP-DELETE-MIN(A, d)

1 swap($A[0]$, $A[A.size-1]$)
2 A.pop()
3 HEAP-FIX-DOWN(A, d, 0)

Током проласка кроз стабло, у свакој итерацији смањујемо висину подстабла у ком се налазимо, па ћемо се у најгорем случају спустити за целу висину стабла. На свакој висини ћемо у најгорем случају проверити сву децу. Укупна временска сложеност функције HEAP-FIX-DOWN, а самим тим и функције HEAP-DELETE-MIN је $O(d \log_d n)$.

3.2.3 Додавање новог чвора

Идеја додавања чвора у хип је слична оној која се користи приликом уклањања чвора са најмањим кључем.

Када додамо чвор, хип својство мора да важи и стабло мора бити скоро пуно. Зато ћемо чвор додати као скроз десни лист, односно на крај низа A. Како сада хип својство може бити нарушено, потребно је да уредимо хип. Чвор ћемо „пењати” уз хип докле год је кључ родитеља већи од кључа чвора, простом заменом одговарајућих вредности. Поступак се завршава када или дођемо до корена или када родитељ има мањи кључ.

Уводимо функцију HEAP-FIX-UP којом олакшавамо додавање новог чвора, а која ће нам касније бити од користи.

HEAP-FIX-UP(A, d, x)

1 **if** $A[x] < A[$HEAP-PARENT(A, d, x)$]$
2   swap($A[x]$, $A[$HEAP-FIX-UP(A, d, x)$]$)
3   FIX-UP(A, HEAP-PARENT(A, d, x))

HEAP-INSERT-NODE(A, d, val)

1 A.push(val)
2 HEAP-FIX-UP(A, d, $A.size-1$)

Током проласка кроз стабло, у свакој итерацији увећавамо висину подстабла у ком се налазимо, па ћемо се у најгорем случају попети за целу висину стабла. Зато временска сложеност функције HEAP-FIX-UP, а самим тим и функције HEAP-INSERT-NODE износи $O(\log_d n)$.

3.2.4 Смањивање кључа произвољног чвора

Да би хип својство остало на снази, после смањивања кључа је можда потребно ажурирати хип. Како вредност кључа постаје мања, чвор или треба поставити на место неког од предака или оставити на старом месту. Приметимо да нам за то одговара већ коришћена метода HEAP-FIX-UP, која нам је служила да лист поставимо на одговарајуће место, што значи да сложеност функције HEAP-DECREASE-KEY износи $O(\log_d n)$.

HEAP-DECREASE-KEY(A, d, x, val)

1 $A[x] = val$
2 HEAP-FIX-UP(A, d, x)

3.2.5 Уклањање произвољног чвора

Након што смо размотрили уклањање најмањег чвора и смањивање кључа произвољног чвора, објаснићемо да је то довољно да се произвољан чвор уклони из хипа.

Идеја којом се водимо јесте да чвор који желимо да уклонимо начинимо кореном хипа. Да би то урадили, смањићемо кључ на вредност коју ни један други чвор не може да има, да би били сигурни да ће постати корен, који потом уклањамо из хипа.

Heap-Delete-Node(A, d, x)
1 Heap-Decrease-Key(A, d, x, $-\infty$)
2 Heap-Delete-Min(A, d)

Како се користимо већ описаним функцијама, сложеност функције Heap-Delete-Node износи $O(d \log_d n)$.

3.2.6 Иницијализација хипа

Претпоставимо да низ A садржи скоро пуно уређено стабло, али да оно не исупањава хип својство и да ми желимо да од таквог стабла начинимо хип. Лако можемо остварити сложеност $O(n \log_d n)$ тако што би правили нови хип додавањем чворова редом. Доказаћемо да је то могуће урадити ефикасније.

Хип ћемо изградити одоздо на горе, тако што у сваком кораку увећавамо висину формираних хипова све док се не постигне висина стабла.

Почињемо од листова, који сами за себе представљају хип висине 0. Затим се пењемо на следећи ниво и формирамо хипове висине 1 користећи се хиповима висине 0, и то тако да корен сваког хипа висине 1 има тачно d деце. Приликом формирања таквих хипова, тренутни корен је можда потребно спустити на место неког његовог листа да би испунили хип својство. Понављајући описан поступак и увећавајући висину у сваком кораку, почетно стабло постаје хип.

Heap-Init(A, d)
1 **for** $i = \lfloor (n-2)/d \rfloor$ **downto** 0
2 Heap-Fix-Down(A, d, i)

Функција Heap-Init неће бити позвана за листове. У најгорем случају ће сваки чвор који није лист бити спуштен једном. Сваки чвор на наредном нивоу ће у најгорем случају бити спуштен још једном. Према томе, корен ће у најгорем случају бити спуштен за целу висину стабла.

Како према леми 3.1.2 стабло има највише $\frac{n}{d} + 1$ листова, закључујемо да ће се прво спуштање десити на највише исто толико чворова и да је сложеност сваког спуштања $O(d)$. Како према последици леме 3.1.2 на следећем нивоу постоји највише $\frac{n}{d^2} + 1$ чворова, десиће се још не више од толико спуштања, свако сложености $O(d)$. Из наведеног можемо извести укупну сложеност функције Heap-Init

$$\sum_{k=1}^{\log_d n} \left(\frac{n}{d^k} + 1 \right) \cdot O(d) = O\left(n \cdot \sum_{k=0}^{\log_d n - 1} \frac{1}{d^k} \right). \tag{3.4}$$

Према формули за збир првих m чланова геометријског низа следи

$$\sum_{k=0}^{\log_d n - 1} \frac{1}{d^k} = \frac{1 - nd}{n - nd}. \tag{3.5}$$

Сада лако можемо изразити коначну сложеност

$$O\left(n \cdot \sum_{k=0}^{\log_d n - 1} \frac{1}{d^k} \right) = O\left(n \cdot \frac{1 - nd}{n - nd} \right) = O\left(\frac{nd - 1}{d - 1} \right) = O(n). \tag{3.6}$$

3.2.7 Спајање два хипа

Користећи функцију Heap-Init, можемо спојити два хипа у сложености $O(n+m)$, где n и m представљају број чворова хипова. Приметимо да хипови не морају бити истог степена и да нови хип може имати произвољан степен.

Heap-Merge(A, B, d)
1 $C = A + B$
2 Heap-Init(C, d)

4 Биномни хип

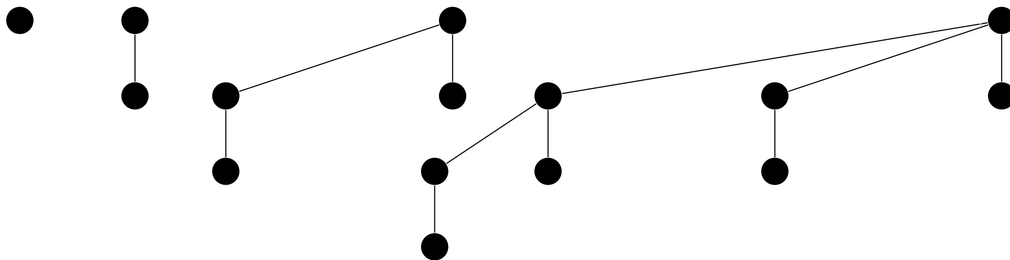
Жан Вилемин[4] је 1978. године представио биномни хип из потребе да се стандардном d–хипу убрза операција спајања [5].

4.1 Дефиниција и својства

Основа биномног хипа је биномно стабло, па ћемо прво објаснити ту врсту стабала. Биномно стабло реда k ћемо означити са B_k.

Дефиниција 4.1.1. Стабло од једног чвора је биномно стабло реда 0, док се стабло B_k добија спајањем два стабла B_{k-1} тако да је лево дете једног стабла корен другог стабла.

Из дефиниције директно следи да B_k садржи тачно 2^k чворова и да има висину k. Лако се увиђа да биномно стабло B_k садржи као децу корена сва стабла B_{k-1}, B_{k-2}, ..., B_0.



Слика 2: Винома стабла B_0, B_1, B_2 и B_3

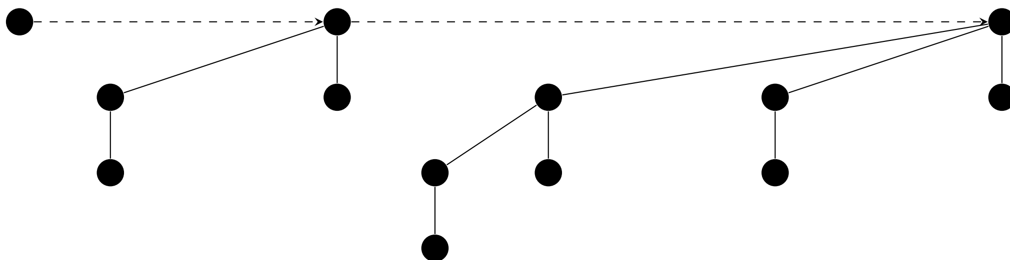
Дефиниција 4.1.2. Шума биномних стабала која испуњавају хип својство и у којој не постоје два биномна стабла истог реда се зове *биномни хип.*

Питање које се намеће јесте како формирати биномни хип од n чворова и како би такав хип изгледао? Одговор лежи у бинарној репрезентацији броја n јер је

$$n = \sum_{k=0}^{\lfloor \log n \rfloor} b_k \cdot 2^k \tag{4.1}$$

где је $b_k \in \{0, 1\}$ и означава k – ти бит у бинарној репрезентацији броја n. Према томе, биномни хип ће садржати биномно стабло B_k ако је $b_k = 1$. На пример, ако је $n = 13$ тада је $(13)_2 = 1101$, па се такав биномни хип представља шумом биномних стабала B_3, B_2 и B_0.

Биномни хип који има само једно биномно стабло се назива *прост биномни хип.*



Слика 3: Биномни хип од 13 чворова

[4] Jean Vuillemin, фраунцуски информатичар.

Биномни хип је погодно представити у „лево дете, десни брат” маниру. За сваки чвор ћемо чувати показивач на лево дете, десног брата и родитеља. Када неки чвор нема неки од атрибута, сматрамо да показивач показује на NIL. Како нам је важан ред биномног стабла, чуваћемо и тај податак. За сваки чвор чувамо $O(1)$ атрибута, па је меморијска сложеност биномног хипа $O(n)$, где је n број чворова у стаблу.

Атрибути *parent*, *child*, *sibling*, *degree* редом означавају родитеља, лево дете, десног брата и степен. Све корене биномног хипа ћемо организовати у *листу корена*.

4.2 Подржане операције

Свака операција се заснива на ефикасном спајању два проста биномна хипа истог реда. Реализација се ослања на дефиницију биномног стабла. За корен хипа постављамо мањи од корена који ће бити родитељ оном дугом.

```
BINOMIAL-LINK(p, q)
   // q.key < p.key
1  p.sibling = q.left
2  q.left = p
3  p.parent = q
4  q.degree = q.degree + 1
```

Како су све операције константне сложености, функција BINOMIAL-LINK захтева $O(1)$ времена.

4.2.1 Проналазак чвора са најмањим кључем

Најмањи чвор мора бити корен неког простог биномног хипа па је из тог разлога довољно да проверимо само те вредности. Сложеност функције BINOMIAL-HEAP-FIND-MIN очигледно износи $O(\log n)$.

```
BINOMIAL-HEAP-FIND-MIN(H)
1  ret = NIL
2  x = H.head
3  while x ≠ NIL
4       if ret == NIL or ret.key > x.key
5            ret = x
6       x = x.sibling
7  return ret
```

4.2.2 Спајање два биномна хипа

Спајање два биномна хипа у великој мери подсећа на сабирање у систему са основом 2. Повезујемо биномна стабла на начин на који се сабирају битови приликом сабирања два бинарна броја који представљају број чворова датих биномних хипова. Када се споје два биномна стабла истог реда, тада новодобијено стабло користимо као „пренос” за следећу позицију.

Користићемо функције BINOMIAL-HEAP-MERGE-ROOTS, која спаја листе корена H_1 и H_2 у једну листу која је уређена по степенима у монотоно растућем поретку, и MAKE-BINOMIAL-HEAP, која прави празан биномни хип.

```
Binomial-Heap-Merge(H1, H2)
 1  H = Make-Binomial-Heap()
 2  H.head = Binomial-Heap-Merge-Roots(H1, H2)
 3  if H.head == NIL
 4      return H
 5  prev = NIL
 6  x = H.head
 7  next = x.sibling
 8  while next ≠ NIL
 9      if (x.degree ≠ next.degree) or
            (next.sibling ≠ NIL and (next.sibling).degree == x.degree)
10              prev = x
11              x = next
12      else if x.key < next.key
13              x.sibling = next.sibling
14              Binomial-Link(next, x)
15          else if prev == NIL
16                  H.head = next
17              else prev.sibling == next
18              Binomial-Link(x, next)
19              x = x.next
20          next = x.sibling
21  return H
```

У функцији Binomial-Heap-Merge се **while** петља извршава тачно $\lfloor \log n \rfloor$ пута, па функција Binomial-Heap-Merge захтева $O(\log n)$ времена.

Уколико би чували показивч на најмањи чвор, који би ажурирали по потреби, најмањи чвор онда можемо пронаћи у $O(1)$. Приметимо да то нема утицај на сложеност осталих операција.

4.2.3 Додавање новог чвора

Чвор додајемо тако што га прикажемо као прост биномни хип B_0 и спојимо са датим биномним хипом. Функција Binomial-Heap-Insert-Node очигледно захтева $O(\log n)$ времена.

```
Binomial-Heap-Insert-Node(H, x)
1  G = Empty-Heap()
2  x.NIL
3  x.child = NIL
4  x.sibling = NIL
5  x.degree = 0
6  G.head = x
7  H = Binomial-Heap-Merge(H, G)
```

4.2.4 Уклањање чвора са најмањим кључем

Најмањи чвор умемо лако да нађемо. Нека је биномно стабло чији корен уклањамо B_k. Уклањањем корена стабло постаје неповезано, али подстабла која остају су редом B_{k-1}, B_{k-2}, ..., B_0. Свако од тих стабала ћемо спојити са биномним хипом из чије листе корена уклањамо најмањи корен и на тај начин добити биномни хип без претходно нађеног најмањег чвора.

BINOMIAL-HEAP-DELETE-MIN(H)
1 obrisati najmanji koren x iz liste korena binomnog hipa H
2 G = MAKE-BINOMIAL-HEAP()
3 preokrenuti listu dece korena x i
    postaviti $G.head$ na prvo dete takve liste
4 H = BINOMIAL-HEAP-MERGE(H, G)

Сам процес спајања подстабала са остатком хипа може да се посматра као спајање два биномна хипа са n чворова, што има сложеност $O(\log n)$. Треба нам $O(\log n)$ времена да формирамо листу корена G, што нас доводи до укупне сложености $O(\log n)$.

4.2.5 Смањивање кључа произвољног чвора

Као и приликом смањивања чвора код хипа, чвор чији кључ смањујемо можда треба померити у биномном стаблу навише да би хип својство и даље важило. Функција која нам је потребна доста подсећа на истоимену функцију коју смо користили у прошлом поглављу. Идеја је иста, чвор подижемо уз стабло докле год или чвор не постане корен или родитељ има мањи кључ.

BINOMIAL-HEAP-DECREASE-KEY(H, x, val)
1 $x.key = val$
2 $p = x$
3 $q = p.parent$
4 **while** $q \neq$ NIL and $p.key < q.key$
5     $p \leftrightarrow q$ // размена атрибута
6     $p = q$
7     $q = p.parent$

Како ће чвор у најгорем случају постати корен, сложеност функције BINOMIAL-HEAP-DECREASE-KEY износи $O(\log n)$.

4.2.6 Уклањање произвољног чвора

Након што смо објаснили како се произвољан чвор умањује и како се брише један од корена биномног хипа, можемо да размотримо брисање произвољног елемента.

BINOMIAL-HEAP-DELETE-NODE(H, x)
1 BINOMIAL-HEAP-DECREASE-KEY(H, x, $-\infty$)
2 BINOMIAL-HEAP-DELETE-MIN(H)

Идеја је да произвољан чвор начинимо кореном биномног стабла ком припада и то тако да у листи корена датог хипа буде најмањи чвор. Потом на већ описан начин бришемо најмањи чвор. Сложеност је очигледно $O(\log n)$.

4.2.7 Иницијализација биномног хипа

Наивна сложеност је $O(n \log n)$, јер биномни хип правимо тако што додајемо чвор по чвор и у сваком кораку спајамо већ постојећи хип са новим простим хипом B_0. Пажљивијом анализом се можемо уверити да је у питању ипак линеарно време [2].

Приметимо да је приликом спајања простог биномног хипа B_0 и било ког другог биномног хипа B_k јако важно колико се пута генерише пренос током операције BINOMIAL-HEAP-MERGE. Колико преноса има, толико има и позива функције BINOMIAL-LINK која формира сложеност иницијализације. Према томе, време које је потребно за иницијализацију хипа је пропорционално са бројем преноса који се генеришу током сукцесивног спајања почетно празног биномног

хипа и простог биномног хипа B_0 тачно n пута.

Парни бројеви тривијално не генеришу пренос. Бројеви који се у бинарном запису завршавају на 01 генеришу тачно један пренос, док бројеви који се завршавају на 011 тачно два. Настављајући резоновање на овај начин, добијамо да је укупан број генерисаних преноса ограничен редом

$$\sum_{k=0}^{\infty} k \cdot \frac{n}{2^{k+1}} = n.$$

Следи да се појединачно додавање n елемената у почетно празан биномни хип извршава за $O(n)$ времена.

4.3 MST алгоритам употребом биномног хипа

Поред Крушкаловог и Примовог алгоритма који су најпознатији MST алгоритми, представићемо алгоритам који користи биномне хипове.

Нека је $G(V, E)$ повезан граф. Извршићемо партицију скупа V на подскупове V_i, који иницијално садрже по један чвор. Сваки скуп V_i генерише скуп E_i у ком се налазе све оне гране којима је бар један чвор у V_i.

```
Binomial-Heap-MST(G)
 1  T = ∅
 2  foreach v_i ∈ V[G]
 3      V_i = {v_i}
 4      E_i = {(v_i, v) ∈ E[G]}
 5  while postoje bar dva skupa V_i
 6      biramo bilo koji V_i
 7          neka je e minimalna grana (u, v) iz E_i
 8          neka je u ∈ V_i i v ∈ V_j
 9          if i ≠ j
10              T = T ∪ {e}
11              V_i = V_i ∪ V_j
12              E_i = E_i ∪ E_j
13  return T
```

Сваки скуп V_i можемо представити биномним хипом који је уређен према тежини грана скупа E_i. Свако спајање два скупа V_i и V_j представља спајање одговарајућих биномних хипова. Укупно ће се десити n спајања хипова који укупно садрже m грана. У сваком кораку **while** петље може доћи до потпуног пражњења хипа, па ћемо $O(m)$ пута морати да узимамо најмању грану. Према томе, овај алгоритам захтева $O(m \log n)$ времена.

5 Фибоначијев хип

Мајкл Фредман[5] и Роберт Тарџан[6] су 1984. године представили Фибоначијев хип, који је у потпуности заснован на амортизованој анализи [6]. Будући да је већина операција амортизовано константа, ова стурктура података се намеће као оптималан избор када се те операције форсирају.

Потенцијални метод амортизоване анализе

Приликом амортозиваног анализирања желимо да упросечимо време извршавања низа операција над неком структуром података. Користећи се овом врстом анализе можемо да утврдимо да је просечна сложеност операције мала, иако је таква операција у најгорем случају изразито неефикасна.

Потенцијални метод је један од неколико могућих приступа амортизоване анализе. Потенцијал можемо схватити као „кредит” којим можемо да платимо цену неке операције. Дефинисаћемо *потенцијалну функцију* $\Phi : \mathbb{D} \to \mathbb{R}$, где је $\mathbb{D}$ скуп свих могућих стања структуре података D. Вредност $\Phi(D_k)$ представља потенцијал структуре података D после низа од k операција. Дефинишемо *амортизовану сложеност* $\hat{c_k}$ у односу на стварну сложеност операције c_k као $\hat{c_k} = c_k + \Phi(D_k) - \Phi(D_{k-1})$. Амортизована сложеност операције је, дакле, стварна сложеност у збиру са променом потенцијала.

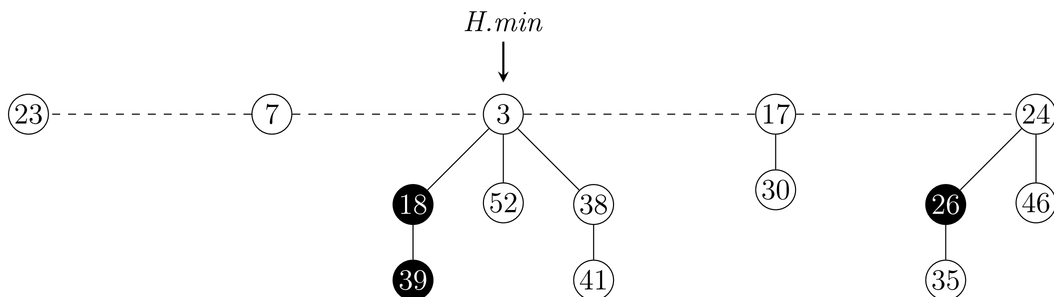
Сада можемо изразити укупну амортизовану сложеност низа од n операција

$$\begin{aligned} \sum_{k=1}^{n} \hat{c_k} &= \sum_{k=1}^{n} (c_k + \Delta\Phi_k) \\ &= \sum_{k=1}^{n} c_k + \Phi(D_n) - \Phi(D_0). \end{aligned} \tag{5.1}$$

5.1 Дефиниције и својства

Дефиниција 5.1.1. Фибоначијев хип је шума стабала у којима важи хип својство.

Чворови су на сваком нивоу повезани двоструко повезаном циркуларном листом. Уз то, за сваки чвор памтимо његовог родитеља и једно од његове деце, уколико их има. Поред тога, важни су нам број деце сваког чвора као и да ли је том чвору одстрањено једно дете (чију ћемо употребну вредност увидети касније). Дакле, за сваки чвор имамо следеће атрибуте: *left*, *right*, *parent*, child,*degree* и *mark*. Поред тога, чуваћемо показивач на најмањи чвор у $H.min$ као и број елемената у $H.n$. Листу која повезује корене зовемо **листа корена**.



Слика 4: Фибоначијев хип у ком су кључеви природни бројеви. Црни чворови су маркирани.

[5] Мајкл Фредман, амерички информатичар.
[6] Роберт Тарџан, амерички информатичар. Добитник Тјурингове награде.

Како за сваки чвор чувамо $O(1)$ атрибута, меморијска сложеност Фибоначијевог хипа је $O(n)$, где је n број чворова у стаблу.

Потенцијална функција коју ћемо користити за анализу Фибоначијевог хипа H је

$$\Phi(H) = t(H) + 2m(H), \tag{5.2}$$

где је *t(H)* број стабала, а *m(H)* број маркираних чворова у хипу. Укупан потенцијал скупа Фибоначијевих хипова је једнак збиру појединачних потенцијала. Како је потенцијал празног Фибоначијевог хипа 0, а потенцијална функција увек ненегативна, време извршавања сваке операције можемо ограничити одозго. Сматрамо да је јединица потенцијала довољно велика да покрије трошак било ког дела операције који се извршава за константно време.

Како нам је у анализи операција јако битан највећи степен који чвор може да има, означићемо са $D(n)$ ту вредност за хип од n чворова. Касније ћемо показати да је $D(n) = O(\log n)$.

5.2 Подржане операције

Операције спајања над Фибоначијевим хипом одлажу промене што је више могуће. Теоретски, могуће је да сви чворови буду у појединачним стаблима. Кључне операције које постижу ефикасност ове структуре су операције уклањања најмањег чвора и смањивања кључа произвољног чвора, па ћемо им се највише посветити.

5.2.1 Проналазак чвора са најмањим кључем

Како већ чувамо показивач на најмањи чвор, његов проналазак захтева $O(1)$ времена. С обзиром да се потенцијал није променио, амортизована сложеност такође износи $O(1)$.

```
FIB-HEAP-FIND-MIN(H)
1  return H.min
```

5.2.2 Додавање новог чвора

Поступак додавања новог чвора је врло једноставан, будући да само додајемо нови елемент у листу корена. Како је нови чвор можда мањи од најмањег, потребно је да и то проверимо.

```
FIB-HEAP-INSERT(H, x)
 1  x.degree = 0
 2  x.parent = NIL
 3  x.child = NIL
 4  x.mark = FALSE
 5  if H.min == NIL
 6      nova lista korena za H u kojoj je samo x
 7      H.min = x
 8  else dodaj x u listu korena od H
 9      if x.key < H.min.key
10          H.min = x
11  H.n = H.n + 1
```

Нека је H' хип који настаје после додатог чвора. Тада је $t(H') = t(H) + 1$ и $m(H) = m(H')$, па је промена потенцијала после функције FIB-HEAP-INSERT

$$\Delta\Phi(H) = ((t(H) + 1) + 2m(H) - (t(H) + 2m(H)) = 1. \tag{5.3}$$

Амортизована сложеност функције FIB-HEAP-INSERT је $O(1) + 1 = O(1)$, јер је сложеност $O(1)$.

5.2.3 Спајање два Фибоначијева хипа

Листе корена хипа H_1 и H_2 спајамо у нову листу корена H чиме формирамо нови хип. Како листе корена хипова H_1 и H_2 више нећемо користити, уништавамо их да би ослободили заузету меморију.

```
Fib-Heap-Merge(H1, H2)
1  H = Make-Fib-Heap()
2  H.min = H1.min
3  spajanje liste korena hipova H1 i H2
4  if (H1.min == NIL) or (H2.min ≠ NIL and H2.min.key < H1.min.key)
5       H.min = H2.min
6  H.n = H1.n + H2.n
7  return H
```

Промена потенцијала је

$$\begin{aligned}\Delta\Phi(H) &= \Phi(H) - (\Phi(H_1) - \Phi(H_2)) \\ &= (t(H) + 2m(H)) - ((t(H_1) + 2m(H_1)) + (t(H_2) + 2m(H_2))) \\ &= 0,\end{aligned} \tag{5.4}$$

јер је $t(H) = t(H_1) + t(H_2)$ и $m(H) = m(H_1) + m(H_2)$. Према томе, амортизована сложеност функције Fib-Heap-Merge је $O(1)$.

5.2.4 Уклањање чвора са најмањим кључем

Као што је већ речено, ова операција је јако важна. Уклањањем најмањег чвора ажурирамо све потребне атрибуте листе корена, али показиваче самог чвора не мењамо.

Будући да имамо показивач на најмањи чвор, најмањи чвор можемо да уклонимо из листе корена у $O(1)$. Затим, сву децу најмањег чвора одвајамо од њега и додајемо у листу корена, чиме се листа корена увећава за највише $D(n)$ стабала. Потом ћемо консолидовати преостала стабла тако што два стабла спајамо у једно стабло у случају да су истог степена. Нови корен је корен оног стабла који је мањи, а његово дете постаје и корен стабла који је већи. На крају је неопходно да ажурирамо показивач на најмањи чвор.

```
Fib-Heap-Delete-Min(H)
 1  z = H.min
 2  if z ≠ NIL
 3       for za svako dete x čvora z
 4            dodaj x u listu korena hipa H
 5            x.p = NIL
 6       ukloni z is liste korena hipa H
 7       if z == z.right
 8            H.min = NIL
 9       else H.min = z.right
10                 Consolidate(H)
11       H.n = H.n − 1
12  return z
```

```
CONSOLIDATE(H)
 1  A[0..D(H.n)] // niz koji čuva pokazivače na stabla različitih stepena
 2  for i = 0 to D(H.n)
 3      A[i] = NIL
 4  for za svaki čvor w u listi korena hipa H
 5      x = w
 6      d = x.degree
 7      while A[d] ≠ NIL
 8          y = A[d] //  постоји чвор (стабло) истог степена као и x
 9          if x.key > y.key
10              x ↔ y // razmena atributa x i y
11          FIB-HEAP-LINK(H, y, x)
12          A[d] = NIL
13          d = d + 1
14      A[d] = x
15  H.min = NIL
16  for i = 0 to D(H.n)
17      if A[i] ≠ NIL
18          if H.min == NIL
19              napravi listu korena hipa H u kojoj je samo A[i]
20              H.min = A[i]
21          else dodaj A[i] u listu korena hipa H
22              if A[i].key < H.min.key
23                  H.min = A[i]
```

```
FIB-HEAP-LINK(H, y, x)
1  ukloni y iz liste korena hipa H
2  y postaje dete čvora x
3  x.degree = x.degree + 1
4  y.mark = FALSE
```

Навели смо и помоћну функцију FIB-HEAP-LINK која један чвор поставља за родитеља другог чвора.

Осврнућемо се на функцију CONSOLIDATE. Поменута функција нам служи да два стабла из листе корена која су истог степена спојимо у једно стабло. Поступак понављамо докле год постоје бар два таква стабла. Да би показали ваљаност ове операције, потребно је да анализирамо **while** петљу у линијама $7-14$.

Инваријанта петље: На почетку сваке итерације **while** петље је $d = x.degree$.

У линији 6 осигуравамо да ће по уласку у петљу инваријанта важити. У свакој итерацији **while** петље, $A[d]$ показује на неки корен y. Како је $d = x.degree = y.degree$, спајамо x и y. Који год чвор да има мањи кључ постаје нови корен. Позив функције FIB-HEAP-LINK у линији 11 увећава степен чвора x за 1, али не мења $y.degree$. Како y више није корен, уклањамо показивач на y из низа A у линији 12. Како је степен чвора x увећан, увећавамо и вредност d за један, чиме враћамо инваријанту. Понављамо **while** петљу све до $A[d] = \text{NIL}$, када више не постоји нити један чвор са истим степеном као и чвор x.

Издвајање деце најмањег чвора захтева $O(D(n))$ времена, као и реконструкција листе корена јер су сви чворови различитог степена. Остаје да се утврди колика је сложеност **for** петље у линијама $4-14$. Применићемо агрегатну анализу. Не знамо колика је тачна сложеност једне итерације, али знамо да свака итерација **while** петље умањује број корена из листе корена за

1, што значи да је укупан број итерација **while** петље $D(n) + t(H)$. Према томе, сложеност уклањања најмањег чвора је $O(D(n) + t(H))$.

Потенцијал пре уклањања најмањег чвора износи $t(H)+2m(H)$, а после уклањања највише $(D(n)+1)+2m(H)$, јер највише $D(n)+1$ чворова може остати у листи корена. Амортизована сложеност функције Fib-Heap-Delete-Min је

$$\begin{aligned} &O(D(n) + t(H)) + ((D(n) + 1) + 2m(H)) - (t(H) + 2m(H)) \\ &= O(D(n)) + O(t(H)) - t(H) \\ &= O(D(n)), \end{aligned} \tag{5.5}$$

јер можемо да увећамо вредност потенцијала колико је потребно да анулирамо константу иза $O(t(H))$.

5.2.5 Смањивање кључа произвољног чвора

Као и до сад, сматрамо да су структурни атрибути чвора који је уклоњен непромењени. Користићемо две нове функције Cut и Cascading-Cut које ћемо после објаснити.

```
Fib-Heap-Delete-Min(H, x, k)
   // k < x.key
1  x.key = k
2  y = x.p
3  if y ≠ NIL and x.key < y.key
4        Cut(H, x, y)
5        Cascading-Cut(H, y)
6  if x.key < H.min.key
7        H.min = x
```

```
Cut(H, x, y)
1  ukloni x iz liste dece čvora y, umanjujući y.degree
2  dodaj x u listu korena hipa H
3  x.p = NIL
4  x.mark = FALSE
```

```
Cascading-Cut(H, y)
1  z = y.p
2  if z ≠ NIL
3        if y.mark == FALSE
4              y.mark = TRUE
5        else Cut(H, y, z)
6              Cascading-Cut (H, z)
```

Уколико нови кључ наруши хип својство, мора доћи до извесних промена. Почињемо уклањањем чвора x из листе деце његовог родитеља, постављајући x за нови корен. Тај део обавља функција Cut.

Приликом ажурирања хипа, од кључног значаја је атрибут *degree*. Помоћу поменутог атрибута, имамо увид у следеће догађаје:

1. у неком тренутку, чвор x је постао корен,
2. чвор x је постао неком другом чвору дете,
3. чвору x су одстрањена два детета.

Чим се чвору одстране два детета, уклањамо га из листе деце његовог корена и постављамо за нови корен.

Родитељу чвора x је уклањање тог чвора можда друго уклањање детета, услед чега и тог родитеља морамо поставити за нови корен. За то нам служи функција CASCADING-CUT. Она ће се пењати до корена и уклонити потребне чворове, а чим наиђе на чвор коме ни једно дете није уклољено, маркира га и стаје. На крају је потребно проверити да ли је нови кључ мањи од тренутно најмањег.

Време извршавања функције FIB-HEAP-DECREASE-KEY је $O(1)$ не рачунајући сложеност резова. Претпоставимо да је укупан број позива функције CASCADING-CUT тачно c. Сваки позив те функције захтева $O(1)$ времена, према томе укупно време за све каскадне резове је $O(c)$.

Остаје нам да израчунамо промену потенцијала. Нека је H хип пре смањивања кључа. Први позив функције CUT можда мења $x.mark$ и поставља чвор x за нови корен. Сваки каскадни рез, осим последњег, одстрањује чвор и поставља његов $mark$ атрибут на FALSE. На крају хип има $t(H) + c$ корена у листи корена и највише $m(H) - c + 2$ маркираних чворова. Дакле, промена потенцијала је

$$\Delta\Phi(H) = ((t(H) + c) + 2m(H) - c + 2)) - (t(H) + 2m(H)) = 4 - c. \tag{5.6}$$

Према томе, аморизована сложеност функције FIB-HEAP-DECREASE-KEY износи

$$O(c) + 4 - c = O(1), \tag{5.7}$$

јер можемо да скалирамо вредност потенцијала колико нам је потребно да би поништили утицај константе иза $O(c)$.

5.2.6 Уклањање произвољног чвора

Као и до сада, ова операција подразумева смањивање кључа до најмање вредности, а онда и уклањање најмањег чвора.

```
FIB-HEAP-DELETE-NODE(H, x)
1  FIB-HEAP-DECREASE-KEY(H, x, −∞)
2  FIB-HEAP-DELETE-MIN(H)
```

Амортизована сложеност ове функције је очигледно $O(D(n))$.

5.3 Горње ограничење највећег степена

Показаћемо да је $D(n) \leq \lfloor \log_\phi n \rfloor$, где је ϕ златни пресек. Уводимо ознаку $size(x)$ за број чворова у стаблу са кореном x укључујући и x.

Теорема 5.3.1. За Фибоначијев хип од n чворова важи $D(n) = O(\log n)$.

Доказ. Прво ћемо доказати неколико лема, које су нам неопходне да комплетирамо доказ.

Лема 5.3.1. Нека је x чвор Фибоначијевог хипа и нека је $x.degree = k$. Нека су $y_1, y_2, ..., y_k$ деца тог чвора оним редом којим су постали његова деца. Тада важи да је $y_1.degree \geq 0$ и $y_i.degree \geq i - 2$ за $i = 2, 3, ...k$.

Доказ. Очигледно је $y_1.degree \geq 0$. За $i \geq 2$, када y_i постане дете чвора x, тада су чворови $y_1, y_2, ..., y_{i-1}$ већ његова деца, па мора бити $y_i.degree = i - 1$, да би их функција CONSOLIDATE спојила. Чвор y_i је од тад изгубио највише једно дете, јер би у случају да је игубио два био уклоњен из подстабла чвора x каскадним резом. Дакле, $y_i.degree \geq i - 2$. □

Фибоначијев број F_k ћемо дефинисати рекурентном везом $F_k = F_{k-1} + F_{k-2}$, за $k \geq 2$, док је $F_0 = 0, F_1 = 1$.

Лема 5.3.2. За сваки цео број $k \geq 0$, важи да је $F_{k+2} = 1 + \sum_{i=0}^{k} F_i$.

Доказ. Користићемо се математичком индукцијом по k. За $k = 0$ је

$$\begin{aligned} 1 + \sum_{i=0}^{0} &= 1 + F_0 \\ &= 1 + 0 \\ &= F2. \end{aligned} \tag{5.8}$$

За индуктивни корак имамо да је $F_{k+1} = 1 + \sum_{i=0}^{k-1} F_i$, из чега следи

$$\begin{aligned} F_{k+2} &= F_k + F_{k+1} \\ &= F_k + \left(1 + \sum_{i=0}^{k-1} F_i\right) \\ &= 1 + \sum_{i=0}^{k} F_i. \end{aligned} \tag{5.9}$$

□

Лема 5.3.3. За сваки цео број $k \geq 0$, важи да је $F_{k+2} \geq \phi^k$.

Доказ. Користићемо се математичком индукцијом корака 2 по k. База индукције је $F_2 = \phi^0$, $F_3 = 2 > 1.619 > \phi^1$. Индуктивни корак је за $k \geq 2$ и према индуктивној претпоставци имамо да је $F_{k+1} \geq \phi^{k-1}$ и $F_k \geq \phi^{k-2}$. Одатле следи

$$\begin{aligned} F_{k+2} &= F_{k+1} + F_k \\ &\geq \phi^{k-1} + \phi^{k-2} \\ &= \phi^{k-2}(\phi + 1) \\ &= \phi^{k-2} \cdot \phi^2 \\ &= \phi^k. \end{aligned} \tag{5.10}$$

□

Лема 5.3.4. Нека је x чвор Фибоначијевог хипа и нека је $x.degree = k$. Тада је $size(x) \geq F_{k+2} \geq \phi^k$.

Доказ. Нека је s_k најмања величина стабла неког чвора са степеном k у неком Фибоначијевом хипу. Тривијално је $s_0 = 1$ и $s_1 = 2$. Број s_k је највише $size(x)$, а како додавање новог детета увећава величину подстабла, вредност s_k строго монотоно расте. Посматрајмо неки чвор z, у неком Фибоначијевом хипу, тако да је $z.degree = k$ и $size(z) = s_k$. Како је $s_k \leq size(x)$, доње ограничење s_k је и доње ограничење $size(x)$. Као и у леми 5.3.1, нека су $y_1, y_2, ...y_k$ деца чвора z оним редом којим су постала његова деца. Тада, уз лему 5.3.1, имамо

$$\begin{aligned} size(x) &\geq s_k \\ &\geq 2 + \sum_{i=2}^{k} s_{y_i.degree} \\ &\geq 2 + \sum_{i-2}^{k} s_{i-2}, \end{aligned} \tag{5.11}$$

Сада ћемо трансфинитном индукцијом [3] по k показати да је $s_k \geq F_{k+2}$, за сваки ненегативан цео број k. База је за $k = 0$ и $k = 1$ тривијална. За индуктивни корак имамо да је $s_i \geq F_{i+2}$ за $i = 0, 1, ..., k - 1$. Важи да је

$$\begin{aligned} s_k &\geq 2 + \sum_{i=2}^{k} s_{i-2} \\ &\geq 2 + \sum_{i=2}^{k} F_i \\ &= 1 + \sum_{i=0}^{k} F_i \\ &= F_{k+2} \\ &\geq \phi^k. \end{aligned} \tag{5.12}$$

□

Нека је x чвор у Фибоначијевом хипу са n чворова, и нека је $x.degree = k$. Према леми 5.3.4. важи $n \geq size(x) \geq \phi^k$. Логаритмујући ову неједнакост, добијамо $k \leq \log_\phi n$, а како је k цео број, важи да је $k \leq \lfloor \log_\phi n \rfloor$. Одатле следи да је $D(n) = O(\log n)$. □

6 Стабло бинарне претраге

Стабло бинарне претраге је структура података која омогућава ефикасне операције над динамичким скупом података. Више научника[7] је 1960. године представила ову врсту стабала [8].

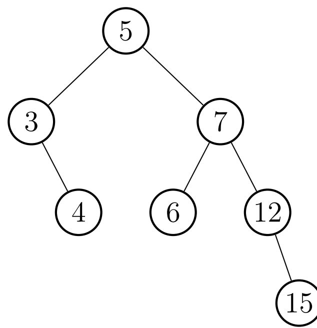
6.1 Дефиниције и својства

Дефиниција 6.1.1. Стабло бинарне претраге је или празно стабло или бинарно стабло за чији корен важи да је лево подстабло стабло бинарне претраге са мањим кључевима, а десно подстабло стабло бинарне претраге са већим кључевима.

Стабло бинарне претраге се лако може представити као повезана структура у којој се у сваком чвору налазе показивачи ка левом и десном детету, као и родитељу.

Атрибути $left$, $right$, p, key означавају редом лево дете, десно дете, родитеља и вредност чвора. Висину стабла означавамо са h.

Будући да за сваки чвор чувамо $O(1)$ атрибута, меморијска сложеност стабла бинарне претраге је $O(n)$, где је n број чворова у стаблу.



Слика 5: Бинарно стабло претраге у ком су кључеви природни бројеви

6.2 Подржане операције

Анализа сложености свих операција у великој мери зависи од висине самог стабла, због чега су Стабла бинарне претраге у својој основној варијанти ретко кад употребљива, али представљају основу за многе друге структуре података. Када је бинарно стабло комплетно тада су операције ефикасне, али како стабло може имати и форму ланца, сложености ћемо изражавати у односу на висину стабла.

6.2.1 Проналазак чвора са најмањим кључем

Дефиниција Стабла бинарне претраге нам дозвољава да се лако „позиционирамо” у стаблу. Како се у левом подстаблу налазе мањи чворови, искључиво ћемо ићи у лево подстабло. Последњи чвор на том путу је најмањи чвор.

```
BST-Find-Min(x)
1  if x.left == NIL
2      return x
3  return BST-Find-Min(x.left)
```

Како се у најгорем случају спуштамо за целу висину стабла, сложеност функције Find-Min је $O(h)$.

[7] P.F. Windley, A.D. Booth, A.J.T. Colin, и T.N. Hibbard

Приметимо да нам ова функција може послужити за проналазак следбеника неког чвора, уколико тај чвор има десно подстабло. Следбеник је чвор који је по вредности први после вредности чвора чији следбеник тражимо. Сви чворови који су кандидати за следбеника су у његовом десном подстаблу, а најмањи од њих је заправо најмањи чвор десног подстабла који је уједно и следбеник. Уколико чвор нема десно подстабло, тада је следбеник

6.2.2 Проналазак чвора одређене вредности

Посматрајмо неки чвор који није лист. У његовом десном подстаблу су сви они чворови који имају већу вредност од њега, као што се у левом подстаблу налазе сви они чворови који имају мању вредност. Зато, када тражимо чвор са неком вредношћу, у случају да тренутни чвор нема тражену вредност, претрагу настављамо у тачно једном подстаблу тог чвора.

BST-SEARCH(x, val)
1 **if** x == NIL or $x.key$ == val
2 **return** x
3 **if** $val < x.key$
4 **return** BST-SEARCH($x.left$, val)
5 **else return** BST-SEARCH($x.right$, val)

Како у сваком кораку смањујемо висину подстабла на ком се налазимо, у најгорем случају ћемо проћи целом висином стабла па сложеност функције BST-SEARCH износи $O(h)$.

6.2.3 Додавање новог чвора

Чвор који се додаје ће постати лист стабла. Његову позицију ћемо врло лако наћи користећи се дефиницијом.

BST-INSERT(x, v)
1 **if** $val < x.key$ and $x.right \neq$ NIL
2 $x.right = v$
3 $v.parent = x$
4 **else if** $val < x.key$
5 BST-INSERT($x.right$, v)
6 **else if** $x.left \neq$ NIL
7 $x.left = v$
8 $v.parent = x$
9 **else** BST-INSERT($x.left$, v)

Како се у сваком кораку спуштамо у подстабло мање висине, најгори случај подразумева да се спустимо низ целу висину стабла па функција BST-INSERT захтева $O(h)$ времена.

6.2.4 Уклањање произвољног чвора

Уклањање чвора x из стабла бинарне претраге има три могућа случаја:

- Ако чвор x нема деце, тада уместо да његов родитељ показује на чвор x као дете, сада показује на NIL.
- Ако чвор x има једно дете, тада је довољно заменити места чвора x и његовог детета.
- Ако чвор x има два детета, тада чвор x мењамо са његовим следбеником, што може бити проблематично уколико следбеник није десно дете чвора x.

Ради олакшања брисања чвора, увешћемо функцију TRANSPLANT која уместо једног подстабла поставља друго, односно на место једног чвора ставља други. Оваква функција очигледно захтева $O(1)$ времена.

```
TRANSPLANT(u, v)
1  if u.p == NIL
2      return
3  if u == (u.p).left
4      (u.p).left = v
5  else (u.p).right = v
6  if v ≠ NIL
7      v.p = u.p
```

Сада се операција уклањања чвора x извршава на следећи начин:

- Ако чвор x нема лево дете, тада чвор x мењамо са његовим десним дететом, које можда постоји, а можда и не. Ако не постоји тиме смо решили случај када чвор x нема децу, а ако постоји тада смо решили сучај када има само једно дете и то десно.
- Ако чвор x има само једно дете и то лево, тада чвор x мењамо са левим дететом.
- Иначе, чвор x има два детета. Тада налазимо следбеника чвора x који се налази у његовом десном подстаблу и сигурно нема лево дете. Нека је следбеник y.
  - Ако је чвор y десно дете чвора x, тада мењамо та два чвора.
  - Ако чвор y није десно дете чвора x, тада y мењамо са његовим десним дететом, затим га постављамо на место родитеља десног детета чвора x, а потом мењамо x и y.

```
BST-DELETE(x)
 1  if x.left == NIL
 2      TRANSPLANT(x, x.right)
 3  else if x.right == NIL
 4      TRANSPLANT(x, x.left)
 5  else y = BST-MIN(x.right)
 6      if y.parent ≠ x
 7          TRANSPLANT(x, y)
 8          y.right = x.right
 9          (y.right).p = y
10      TRANSPLANT(x, y)
11      y.left = x.left
12      (y.left).p = y
```

Како нам за проналажење следбеника треба $O(h)$ времена, а све остале операције су $O(1)$, закључујемо да нам за уклањање чвора из бинарног стабла претраге треба $O(h)$ времена.

6.2.5 Иницијализација стабла бинарне претраге

Почетно стабло је празно, а ми додајемо укупно n чворова. Распоред додавања чворова може произвести дегенеративну форму стабла, па наивна сложеност износи $O(n^2)$.

Нека је n кључева организовано у низ. Да ли би умели ефикасније да изградимо стабло ако би тај низ био сортиран? Нама је у интересу да стабло буде балансирано, јер је оно тада оптималне висине. Ако би корен имао вредност која заузима средишњу позицију у низу кључева,

тада би се величине левог и десног подстабла разликовале за највише 1, чиме постижемо балансираност. Да би стабло начинили стаблом бинарне претраге, лево подстабло ћемо изградити од кључева мањих од корена, десно подстабло од већих. Према томе, стабло ћемо изградити одозго на доле, тако да чворови узимају средишњи елемент одговарајућих поднизова низа кључева.

Користићемо и помоћну функцију BST-Empty-Node која прави нов, празан чвор стабла бинарне претраге.

```
BST-Init(x, a, L, R)
1  MID = (L + R)/2
2  x.key = a[MID]
3  if L ≤ MID − 1
4        x.left = BST-Empty-Node()
5        BST-Init(x.left, a, L, MID − 1)
6  if MID + 1 ≤ R
7        x.right = BST-Empty-Node()
8        BST-Init(x.right, a, MID + 1, R)
9  return x
```

Како ћемо сваки чвор посетити тачно једном, јасно је да функција BST-Init захтева $O(n)$ времена. Сортирање низа кључева троши $O(n \log n)$, па је укупна сложеност $O(n \log n)$. Иницијализација се може извршити у сложености $O(n)$ ако су испуњени услови за примену неког линеарног алгоритма сортирања, или ако је низ кључева већ сортиран.

6.2.6 Спајање два стабла бинарне претраге

Желимо да искористимо ефикасну операцију иницијализације стабла бинарне претраге. Ако би успели ефикасно да организујемо кључеве стабала, можемо ефикасно да их спојимо.

Приметимо да чворове стабла бинарне претраге можемо добити у сортираном поретку само једним пролазом кроз стабло, што можемо за линеарно време. Кренимо од корена стабла. Знамо да су сви чворови у левом подстаблу мањи, и они у сортираном низу кључева заузимају мању позицију од корена. Зато ћемо корен да поставимо на прву слободну позицију после постављених чворова левог подстабла. Десно подстабло садржи све веће чворове од корена, па сви ти чворови морају да буду после корена у сортираном низу кључева. Закључујемо да смо корен поставили на право место. Обилазећи чворове на описани, рекурзиван, начин, стабло обилазимо у *in-order* поретку, што значи да сортиран низ кључева стабла добијамо за време $O(n)$.

Како су оба низа кључева сортирани, њихово спајање у сортиран низ захтева $O(n + m)$ времена, где су n и m број чворова стабала. Потом, већ описаним поступком, градимо стабло бинарне претраге које садржи кључеве оба стабла. Према томе, спајање два стабла бинарне претраге захтева $O(n + m)$ времена.

Ради лакше имплементације, користићемо динамички низ за чување кључева стабала.

```
BST-In-Order(x)
1  ret = ∅
2  if x.left ≠ NIL
3        ret = BST-In-Order(x.left)
4  ret.push(x.key)
5  if x.right ≠ NIL
6        ret = ret + BST-In-Order(x.right)
7  return ret
```

BST-MERGE(x, y)
1 a = BST-IN-ORDER(x)
2 b = BST-IN-ORDER(y)
3 c = MERGE(a, b) // спајање два сортирана низа у један
4 $root$ = BST-EMPTY-NODE()
5 **return** BST-INIT($root$)

6.3 Очекивана висина бинарног стабла претраге

Показали смо да се све основне операције над стаблом бинарне претраге извршавају за време $O(h)$. Али како висина стабла може да се мења уклањањем или додавањем чворова, потребно је да проценимо висину бинарног стабла претраге у општем сучају.

У најгорем случају, сви чворови се могу налазити на једном путу, што би се десило када би n кључева додавали у растућем поретку у почетно празно стабло. Други дегенеративни случајеви су путеви који вијугају лево или десно код неких унутрашњих чворова. Дакле, проблем нам праве случајеви када је висина стабла $\Theta(n)$.

Највише би нам одговарало да је стабло комплетно бинарно стабло. Тада је висина $\log n$ па су све операције сложености $O(\log n)$. Према томе, операције су у најбољем случају сложености $O(\log n)$.

Висина стабла тешко може да се процени ако су заступљене операције уклањања и додавања чвора. Зато ћемо дефинисати **насумично генерисано стабло бинарне претраге** са n чворова као стабло које настаје после n операција додавања чвора у почетно празно стабло бинарне претраге.

Теорема 6.3.1. Математичко очекивање висине насумично генерисаног стабла бинарне претраге са n чворова је $O(\log n)$.

Доказ. Дефинисаћемо три случајне променљиве које ће нам помоћи да одредимо очекивану висину. Нека је X_n случајна променљива која представља висину насумично генерисаног стабла бинарне претраге од n чворова и нека је $Y_n = 2^{X_n}$ случајна променљива која представља експоненцијалну висину тог стабла. Нека је R_n случајна променљива која представља позицију кључа корена насумично генерисаног стабла бинарне претраге од n чворова у сортираном низу свих кључева тог стабла по мањој вредности. Једнако је вероватно да R_n узме било коју вредност из скупа $\{1, 2, ..., n\}$.

Ако је $R_n = i$ тада је лево подстабло изграђено од $i - 1$ кључева, а десно од $n - i$. Како је висина стабла за 1 већа од веће висине подстабала, експоненцијална висина је двоструко већа од висине вишег подстабла. Према томе, ако је $R_n = i$, имамо да је

$$Y_n = 2 \cdot \max(Y_{i-1}, Y_{n-i}). \tag{6.1}$$

Базни случај је $Y_1 = 1$, а додаћемо и да је $Y_0 = 0$.

Нека је $I_{n,i}$ индикатор догађаја да за насумично генерисано стабло бинарне претраге од n чворова важи $R_n = i$. Како је у питању индикаторска променљива, за свако $i \in \{1, 2, ..., n\}$ важи

$$\mathbb{E}(I_{n,i}) = \frac{1}{n}. \tag{6.2}$$

Из истог разлога је

$$Y_n = \sum_{i=1}^{n} I_{n,i} \cdot (2 \cdot \max(Y_{i-1}, Y_{n-i})). \tag{6.3}$$

Лако се увиђа да је индикатор $I_{n,i}$ независан од Y_{i-1} и Y_{n-i}, па на основу тога имамо

$$\begin{aligned}
\mathbb{E}(Y_n) &= \mathbb{E}\left(\sum_{i=1}^{n} I_{n,i} \cdot (2 \cdot \max(Y_{i-1}, Y_{n-i}))\right) \\
&= \sum_{i=1}^{n} \mathbb{E}(I_{n,i} \cdot (2 \cdot \max(Y_{i-1}, Y_{n-i}))) \\
&= \sum_{i=1}^{n} \mathbb{E}(I_{n,i}) \cdot \mathbb{E}(2 \cdot \max(Y_{i-1}, Y_{n-i})) \\
&= \sum_{i=1}^{n} \frac{1}{n} \cdot \mathbb{E}(2 \cdot \max(Y_{i-1}, Y_{n-i})) \\
&= \frac{2}{n} \sum_{i=1}^{n} \mathbb{E}(\max(Y_{i-1}, Y_{n-i})) \\
&\leq \frac{2}{n} \sum_{i=1}^{n} (\mathbb{E}(Y_{i-1}) + \mathbb{E}(Y_{n-i})) \\
&= \frac{4}{n} \sum_{i=0}^{n} \mathbb{E}(Y_i)
\end{aligned} \tag{6.4}$$

Индукцијом ћемо показати да је $\mathbb{E}(Y_n) \leq \frac{1}{4}\binom{n+3}{3}$. Користићемо се идентитетом *хокејашког штапа*, који овде нећемо доказивати.

База индукције је испуњена јер је $0 = Y_0 = \mathbb{E}(Y_0) \leq \frac{1}{4}\binom{3}{3} = \frac{1}{4}$ и $1 = Y_1 = \mathbb{E}(Y_1) \leq \frac{1}{4}\binom{1+3}{3} = 1$. Примењујући индуктивну претпоставку на индуктивни корак добијамо тражено

$$\begin{aligned}
\mathbb{E}(Y_n) &\leq \frac{4}{n} \sum_{i=0}^{n} Y_i \\
&\leq \frac{4}{n} \sum_{i=0}^{n-1} \frac{1}{4}\binom{i+3}{3} \\
&= \frac{1}{n} \sum_{i=0}^{n} \binom{i+3}{3} \\
&= \frac{1}{n}\binom{n+3}{4} \\
&= \frac{1}{4}\binom{n+3}{3}.
\end{aligned} \tag{6.5}$$

Како је функција $f(x) = 2^x$ конвексна, применићемо Јенсенову неједнакост, па добијамо

$$\begin{aligned}
2^{\mathbb{E}(X_n)} &\leq \mathbb{E}\left(2^{X_n}\right) \\
&= \mathbb{E}(Y_n) \\
&\leq \frac{1}{4}\binom{n+3}{3} \\
&= \frac{1}{4} \cdot \frac{(n+3)(n+2)(n+1)}{6} \\
&= \frac{n^3 + 6n^2 + 11n + 6}{24}.
\end{aligned} \tag{6.6}$$

Логаритмујући ову неједнакост добијамо $\mathbb{E}(X_n) = O(\log n)$. □

7 Сплеј стабло

Сплеј стабло припада фамилији самобалансирајућих стабала бинарне претраге, са особином да је елементима којима је већ приступљено могуће приступити поново за значајно мање времена. Тарџан и Слетор[8] су 1985. године представили ову врсту стабала као потребну структуру за link/cut стабла [7]. У односу на обично стабло бинарне претраге, ово стабло користи *сплеј* операцију којом одржава балансираним.

Све операције су исте као и код обичног стабла бинарне претраге и њихове сложености су $O(h)$. Предност овог стабла јесте што су амортизоване сложености $O(\log n)$, што ћемо и доказати у даљем тексту.

Нека је $size(x)$ величина подстабла са кореном x и нека је $rank(x) = \log_2(size(x))$. Потенцијална функција коју ћемо користити за потенцијални метод амортизоване анализе је

$$\Phi(D) = \sum_{v \in V_D} rank(v).$$

7.1 Подржане операције

Детаљније ћемо описати само сплеј операцију којом се чвор поставља за корен стабла. Сплеј операција се користи кад год се неком чвору приступа, услед чега тај чвор постаје корен стабла. Остале операције су исте као и код стабла бинарне претраге.

Сплеј операција

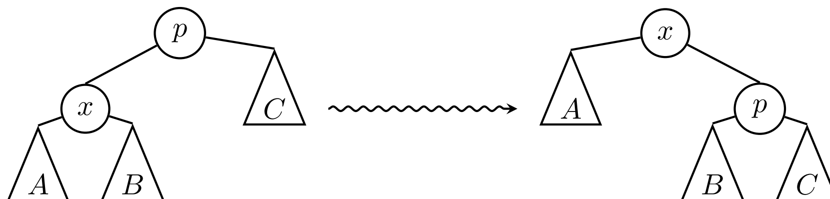
Када се приступи чвору x, низом корака се поставља за корен стабла. Скорије приступљени чворови су ближи корену стабла.

Разликујемо три основна случаја:

- да ли је чвор x лево или десно дете свог родитеља p,
- да ли је чвор p корен, а ако није
- да ли је чвор p лево или десно дете свог родитеља g.

Постоје три корака сплеј операције, и то за сваки од њих лева и десна варијанта, зависно од положаја у стаблу. Показаћемо само леву варијанту којој је десна аналогна.

- **Цик корак:** Дешава се када је чвор p корен. Стабло се ротира око гране (x, p). Може се догодити само једном и то на крају сплеј операције, под условом да је чвор x био на непарној дубини.

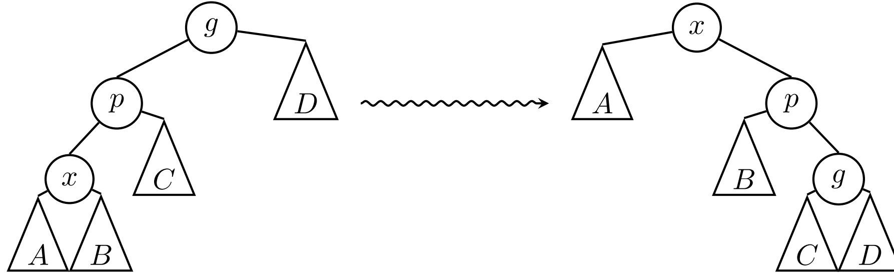


Нека је $rank'$ ранг после цик корака. Тада важи

$$\begin{aligned}\Delta\Phi &= rank'(p) - rank(p) + rank'(x) - rank(x) \\ &= rank'(p) - rank(x) \\ &\leq rank'(x) - rank(x)\end{aligned} \tag{7.1}$$

[8]Daniel Sleator – амерички информатичар познат по link/cut стаблима

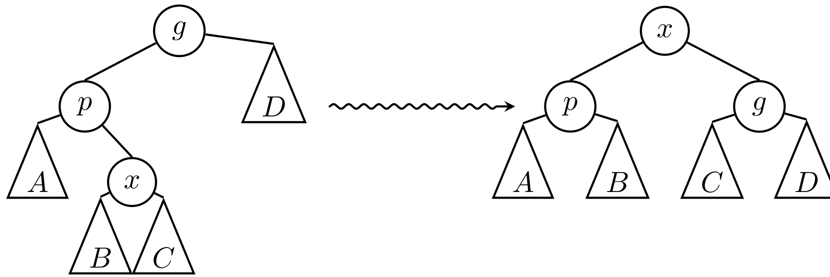
- **Цик–цик корак:** Дешава се када чвор p није корен и када x и p припадају левом подстаблу својих родитеља. Стабло се ротира око гране (p, g).



Нека је $rank'$ ранг после цик–цик корака. Тада важи

$$\begin{aligned}\Delta\Phi &= rank'(g) - rank(g) + rank'(p) - rank(p) + rank'(x) - rank(x) \\ &= rank'(g) + rank'(p) - rank(p) - rank(x) \\ &\leq rank'(g) + rank'(x) - 2 \cdot rank(x) \\ &\leq 3 \cdot (rank'(x) - rank(x)) - 2\end{aligned} \tag{7.2}$$

- **Цик–цак корак:** Дешава се када је x у десном подстаблу свог родитеља, а p у левом. Стабло се ротира око гране (x, p), а онда око гране (x, g).



Нека је $rank'$ ранг после цик–цак корака. Тада важи

$$\begin{aligned}\Delta\Phi &= rank'(g) - rank(g) + rank'(p) - rank(p) + rank'(x) - rank(x) \\ &\leq rank'(g) + rank'(p) - 2 \cdot rank(x) \\ &\leq 2 \cdot (rank'(x) - rank(x)) - 2\end{aligned} \tag{7.3}$$

Како се у цик–цик и цик–цак кораку дешавају две ротације амортизована сложеност те две операције је највише $3 \cdot (rank'(x) - rank(x))$, што је $O(\log n)$. Цик корак додаје амортизовану сложеност 1, али је највише једна таква операција. Дакле, за низ од m операција је потребно $O(m \log n)$ амортизованог времена. Да би време извршавања ограничили одозго потребно је да урачунамо промену потенцијала

$$\Phi_{init} - \Phi_{end} = \sum_{v \in V} rank_{init}(v) - \sum_{v \in V} rank_{end}(v) = O(n \log n). \tag{7.4}$$

Према томе, за извршавање низа од m операција је потребно $O(m \log n + n \log n)$ времена.

8 Ван Емде Боасово стабло

Петар ван Емде Боас[9] је 1975. године представио ову врсту стабала [8]. Ван Емде Боасово стабло има сложеност $O(\log\log n)$ по операцији, уместо досадашње сложености $O(\log n)$. За разлику од досадашњих структура, ово стабло подразумева да су елементи из тачно одређеног скупа вредности.

8.1 Мотивација

За почетак, желимо да подржимо операције Find, Insert и Delete за време $O(1)$. То се врло лако имплементира коришћењем логичког низа који има вредност 1 на позицји x ако је вредност x претходно убачена у низ, а 0 иначе. Нека је величина низа n. Проналазак одређене вредности и њено брисање се тривијално имплементира. Операције које су неефикасне над овим низом јесу Succesor и Predecessor јер захтевају $O(n)$ времена за проналазак следбеника, односно претходника.

Иако наизглед ни мало ефикасна структура података, логички низ је у основи ван Емде Боасовог стабла. Како се над овако примитивном структуром појави сложеност $O(\log\log n)$?

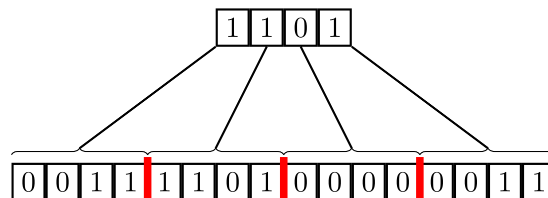
Претпоставимо да су елементи из неког универзума U јединствени и да могу да узму вредност из скупа $\{0, 1, 2, ..., u-1\}$. Ради једноставности, претпоставићемо да је u степен двојке. Интуитивно, логаритамска сложеност се јавља приликом бинарног претраживања. Уколико би имали стабло висине $\log u$, бинарним претраживањем би дошли до поменуте сложености. Према томе, основна рекурентна формула је

$$T(\log u) = T\left(\frac{\log u}{2}\right) + O(1). \tag{8.1}$$

Уколико би добијену формулу изразили у зависности од u добили би

$$\begin{aligned} T(u) &= T(\sqrt{u}) + O(1) \\ T(u) &= O(\log\log u). \end{aligned} \tag{8.2}$$

Добијена рекурентна формула је позната и асоцира на корен декомпозицију низа, што ћемо и урадити. Низ ћемо поделити на $\sqrt{u}$ кластера величине по $\sqrt{u}$. За сваки непразан кластер чувамо 1, а иначе 0. То ћемо да урадимо формирањем логичког низа величине $\sqrt{u}$ којег ћемо звати *сумирајући низ*. Када хоћемо да нађемо следбеника неког елемента, довољно је да у сумирајућем наиђемо на прву следећу TRUE вредност, а онда да у том кластеру нађемо најмањи елемент. Дакле, операција Successor захтева $O(\sqrt{u})$ времена. Потпуно аналоган поступак важи за тражење претходника неког елемента.



Слика 6: Низ од 16 елемената са сумирајућим низом и кластерима

[9] Peter van Emde Boas – дански информатичар

За неки елемент x важи да је редни број кластера којем припада $\lfloor x/\sqrt{u} \rfloor$, а да је његов редни број у кластеру $x \mod \sqrt{u}$. Увешћемо три помоћне функције

$$\begin{aligned} high(x) &= \lfloor x/\sqrt{u} \rfloor \\ low(x) &= x \mod \sqrt{u} \\ index(i,j) &= i \cdot \sqrt{u} + j. \end{aligned} \tag{8.3}$$

Имена $high$ и low нису случајна. Као што видимо из (1), а знајући да за репрезентацију броја x у бинарном бројном систему треба $\lfloor \log x \rfloor$ цифара, лако можемо да покажемо да прва половина броја x_2 одговара вредности $high(x)$, а друга половина вредности $low(x)$.

8.2 Прото ван Емде Боасово стабло

Иако не постиже жељене сложености, прото структура представља основу за ван Емде Боасово стабло, па ћемо зато размотрити ову „лакшу” верзију стабла.

8.2.1 Дефиниције и својства

Претпоставимо да је $u = 2^{2^k}$. Иако је ово прилично стриктна претпоставка, значајно ће олакшати описивање помоћне структуре.

Прото ван Емде Боасово стабло величине u ћемо означити са proto-vEB(u).

Дефиниција 8.2.1. За универзум величине 2, прото ван Емде Боасово стабло садржи, поред величине универзума u, само двобитовни логички низ A са већ описаном улогом, за скуп вредности $\{x, x+1\}$. За универзум величине $u \geq 4$, у прото ван Емде Боасовом стаблу постоје још два атрибута:

- низ $cluster[0..\sqrt{u}-1]$ – садржи показиваче на $\sqrt{u}$ прото структура величине $\sqrt{u}$,
- $summary$ – показивач на сумирајуће стабло proto-vEB($\sqrt{u}$). Сумирајуће стабло има исту улогу као и сумирајући низ и представља природно проширење на стаблу.

8.2.2 Подржане операције

Прво ћемо размотрити операције које не мењају стабло.

8.2.2.1 Провера присуства елемента

Желимо да проверимо да ли се дати елемент x налази у стаблу. Случај када је $u = 2$ је тривијалан. Када је $u \geq 4$, прво проверавамо да ли кластер елемента x уопште садржи елементе, што можемо помоћу сумирајућег стабла. Даље рекурзивно улазимо у само једну структуру.

```
MEMBER(V, x)
1  if V.u == 2
2       return V.A[x]
3  return MEMBER(V.cluster[high(x)], low(x))
```

Рекурентна формула је $T(u) = T(\sqrt{u}) + O(1)$, па сложеност фунцкије MEMBER износи $O(\log \log u)$.

8.2.2.2 Проналазак најмањег елемента

Увек нам је у интересу да узмемо први непразан кластер, јер ће у њему сигурно бити мања вредност него у било ком другом непразном кластеру. Зато проналазимо најмањи елемент у сумирајућем стаблу, који заправо представља први непразан кластер. Рекурзивно тражећи најмањи елемент за тај кластер, налазимо најмањи елемент целог стабла.

```
PROTO-vEB-FIND-MIN(V)
 1  if V.u == 2
 2      if V.A[0] == 1
 3          return 0
 4      if V.A[1] == 1
 5          return 1
 6      return ∞
 7  else min-cluster = PROTO-vEB-FIND-MIN(V.summary)
 8      if min-cluster == NIL
 9          return ∞
10  else offset=PROTO-vEB-FIND-MIN(V.cluster[min-cluster])
11      return index(min-cluster, offset)
```

У функцији PROTO-vEB-FIND-MIN се дешавају два рекурзивна позива за proto-vEB($\sqrt{u}$), па зато имамо другачију рекурентну формулу

$$T(u) = 2T(\sqrt{u}) + O(1). \tag{8.4}$$

Уводећи смену $m = \log u$ имамо

$$T(2^m) = T(2^{m/2}) + O(1). \tag{8.5}$$

Уводећи смену $S(m) = T(2^m)$ добијамо

$$S(m) = 2S(m/2) + O(1), \tag{8.6}$$

што према мастер теореми следи да је $S(m) = \Theta(m)$. Враћајући смену $u = \log m$, добијамо да је сложеност функције PROTO-vEB-FIND-MIN $\Theta(\log u)$, уместо $O(\log\log u)$.

8.2.2.3 Проналазак следбеника произвољног елемента

Прво морамо да проверимо постоји ли већи елемент од x у његовом кластеру. Уколико то није испуњено, треба нам најмањи елемент првог непразног кластера.

```
PROTO-vEB-SUCCESSOR(V, x)
 1  if V.u == 2
 2      if x == 0 and V.A[1] == 1
 3          return 1
 4      else return ∞
 5  else offset = PROTO-vEB-SUCCESSOR(V.cluster[high(x)], low(x))
 6          if offset≠ ∞
 7              return index(high(x), offset)
 8          else succ-cluster = PROTO-vEB-SUCCESSOR(V.summary, high(x))
 9                  if succ-cluster == ∞
10                      return ∞
11              else offset = PROTO-vEB-FIND-MIN(V.cluster[succ-cluster])
12                      return index(succ-cluster, offset)
```

Лако долазимо до рекурентне формуле

$$\begin{aligned} T(u) &= 2T(\sqrt{u}) + \Theta(\log \sqrt{u}) \\ &= 2T(\sqrt{u}) + \Theta(\log u). \end{aligned} \tag{8.7}$$

Слично као у (8.4), добија се да сложеност функције следбеника износи $\Theta(\log u \log\log u)$.

8.2.2.4 Додавање новог елемента

Када додајемо нови елемент, неопходно да је променимо сумирајуће стабло и вредност у кластеру и то на сваком нивоу стабла.

PROTO-VEB-INSERT(V, x)

1 **if** $V.u == 2$
2     $V.A[x] = 1$
3 **else** PROTO-VEB-INSERT($V.cluster[high(x)]$, $low(x)$)
4   PROTO-VEB-INSERT($V.summary$, $high(x)$)

Како се дешавају два рекурзивна позива, сложеност функције PROTO-VEB-INSERT износи $\Theta(\log u)$.

8.2.2.5 Уклањање произвољног елемента

Операција брисања је компликованија од додавања елемента. За разлику од додавања, где 1 можемо да упишемо на место на ком можда већ постоји 1, са уписивањем нуле то није случај. Морамо да утврдимо да ли уклањањем тог елемента постоји неки други елемент, због чега вредност сумирајућег стабла не смемо да мењамо. Псеудокод ове функције изостављамо, јер се имплементација ове операције значајно разликује од прото варијанте.

8.3 Ван Емде Боасово Стабло – коначна верзија

Описаћемо финалну верзију ове структуре података. У великој мери ћемо се ослонити на прото структуру. Циљ је да смањимо број рекурзивних позива и да на тај начин достигнемо жељену сложеност.

8.3.1 Дефиниција и својства

Главна претпоставка у прото варијанти јесте $u = 2^{2^k}$. Од сада подразумевамо да је $u = 2^k$. Када се деси да $\sqrt{u}$ није цео број, поделићемо $\log u$ битова на водећих $\lceil(\log u/2)\rceil$ и осталих $\lfloor(\log u/2)\rfloor$ битова. Вредност $2^{\lceil(\log u/2)\rceil}$ ћемо означити са $\sqrt[\uparrow]{u}$, а вредност $2^{\lfloor(\log u/2)\rfloor}$ са $\sqrt[\downarrow]{u}$. У складу са тим ћемо додефинисати функције *high*, *low* и *index*:

$$\begin{aligned} high(x) &= \lfloor x/\sqrt[\downarrow]{u} \rfloor \\ low(x) &= x \mod \sqrt[\downarrow]{u} \\ index(i,j) &= i\sqrt[\downarrow]{u} + j. \end{aligned} \tag{8.8}$$

У складу са насталим променама, можемо да дефинишемо ван Емде Боасово стабло, као унапређену верзију прото ван Емде Боасовог стабла, које ћемо означити са $vEB(u)$ за универзум величине u.

Дефиниција 8.3.1. Уколико је $u \geq 4$, атрибут *summary* показује ка $vEB(\sqrt[\uparrow]{u})$ стаблу, а низ $cluster[0..\sqrt[\uparrow]{u}]$ садржи показиваче на $\sqrt[\uparrow]{u}$ $vEB(\sqrt[\downarrow]{u})$ стабала. Постоје још два атрибута:

- *min* – најмањи елемент у vEB стаблу и
- *max* – највећи елемент у vEB стаблу.

Елемент сачуван у min се неће појавити ни у једном рекурзивном $vEB(\sqrt[\downarrow]{u})$ стаблу на које $cluster$ показује, док елемент max хоће. Дакле, елементи $vEB(u)$ стабла V, су $V.min$ и сви елементи $vEB(\sqrt[\downarrow]{u})$ стабала на које показује $V.cluster[0..\sqrt[\uparrow]{u}]$.

Сада нам за $u = 2$ више не треба двобитовни логички низ A јер имамо атрибуте min и max. Ови атрибути нам помажу у следећим случајевима:

- Функције Find-Min и Find-Max могу само да врате атрибуте min и max.
- Функција Successor више не мора да прави рекурзиван позив ради провере да ли су x и његов следбеник у истом кластеру. То важи због тога што те две вредности могу бити у истом кластеру акко је вредност елемента x строго мања од највеће вредности тог кластера. Аналогно важи за функцију Predecessor.
- Лако утврђујемо да ли је стабло празно, да ли има само један или бар два елемента у сложености $O(1)$.
- Уколико је стабло празно или има један елемент, додавање и брисање елемента захтева $O(1)$ времена.

Чак и када је u непаран степен броја 2, разлика у величини сумирајућег стабла и кластера неће утицати на асимптотику операција. Главна рекурентна формула је

$$T(u) \leq T(\sqrt[\uparrow]{u}) + O(1). \tag{8.9}$$

Водећи се сличним поступком као у (8.4), добијамо да је $T(u) = O(\log\log u)$.

8.3.2 Подржане операције

Као и при описивању операција прото ван Емде Боасовог стабла, и овде ће прво бити размотрене опрације које не мењају стабло.

8.3.2.1 Проналазак најмањег и највећег елемента

Како се потребне информације налазе у атрибутима стабла, сложености су $O(1)$.

```
vEB-Tree-Find-Min(V)
1  return V.min
```

```
vEB-Tree-Find-Max(V)
1  return V.max
```

8.3.2.2 Провера присуства елемента

Ова ће функција, иако у прото структури оптимална, бити промењена. Разлог томе је што vEB стабло не чува битове као што то ради proto-vEB.

```
vEB-Tree-Member(V, x)
1  if x == V.min or x == V.max
2      return TRUE
3  elseif V.u == 2
4      return FALSE
5  else return vEB-Member(V.cluster[high(x)], low(x))
```

Како се дешава само један рекурзиван позив, рекурентна формула је једнака (8.9) па је сложеност функције vEB-Tree-Member $O(\log\log u)$.

8.3.2.3 Проналазак следбеника произвољног елемента

Главна разлика у односу на прото структуру је што су нам најмањи и највећи елементи доступни, те нам рекурзиван позив у случају да следбеник припада истом кластеру као и x више није потребан.

```
vEB-Tree-Successor(V, x)
 1  if V.u == 2
 2      if x == 0 and V.max == 1
 3          return 1
 4      else return ∞
 5  elseif V.min ≠ −∞ and x < V.min
 6      return V.min
 7  else max-low = vEB-Tree-Find-Max(V.cluster[high(x)])
 8      if max-low ≠ ∞ and low(x) < max-low
 9          offset = vEB-Tree-Successor(V.cluster[high(x)], low(x))
10          return index(high(x), low(x))
11      else succ-cluster = vEB-Tree-Successor(V.summary, high(x))
12              if succ-cluster == ∞
13                  return ∞
14          else offset = vEB-Tree-Find-Min(V.cluster[succ-cluster])
15                  return index(succ-cluster, offset)
```

Како је могуће применити (8.9), за функцију vEB-Tree-Succesor треба $O(\log \log u)$ времена.

8.3.2.4 Проналазак претходника произвољног елемента

Иако наизглед потпуно симетрична, функција vEB-Predecessor има једну битну разлику у односу на функцију следбеника. У случају када је претходник елемента минимум vEB стабла, тада тај елемент не може бити пронађен ни у једном кластеру.

```
vEB-Tree-Predecessor(V, x)
 1  if V.u == 2
 2      if x == 0 and V.min == 0
 3          return 0
 4      else return −∞
 5  elseif V.max ≠ ∞ and x > V.max
 6      return V.max
 7  else max-low = vEB-Tree-Find-Min(V.cluster[high(x)])
 8      if min-low ≠ −∞ and low(x) > min-low
 9          offset = vEB-Tree-Predecessor(V.cluster[high(x)], low(x))
10          return index(high(x), low(x))
11      else pred-cluster = vEB-Tree-Predecessor(V.summary, high(x))
12              if pred-cluster == −∞
13                  if V.min ≠ −∞ and x > V.min
14                      return V.min
15                  else return −∞
16              else offset = vEB-Tree-Find-Max(V.cluster[pred-cluster])
17                      return index(pred-cluster, offset)
```

Како је једина промена у односу на функцију следбеника сложености $O(1)$, закључујемо да функција vEB-Tree-Predecessor захтева $O(\log \log u)$ времена.

8.3.2.5 Додавање новог елемента

Приликом додавања елемента у прото стабло била су потребна два рекурзивна позива, један за сумирајуће стабло, а други за кластер. Приликом додавања елемента у vEB стабло довољан је само један позив. Ако је редни број кластера већ присутан у сумирајућем стаблу нема потребе за икаквим додавањем у то стабло. Уколико је кластер празан, додавање елемента у кластер захтева $O(1)$ времена и нема потребе за даљом рекурзијом.

Користићемо и помоћну функцију Empty-vEB-Tree-Insert, да би лакше решили проблем празног стабла.

```
Empty-vEB-Tree-Insert(V, x)
1  V.min = x
2  V.max = x
```

```
vEB-Tree-Insert(V, x)
 1  if V.min == −∞
 2        Empty-vEB-Tree-Insert(V, x)
 3  elseif x < V.min
 4        x ↔ V.min // размена вредности
 5        if V.u > 2
 6              if vEB-Tree-Find-Min(V.cluster[high(x)])== −∞
 7                    vEB-Tree-Insert(V.summary, high(x))
 8                    Empty-vEB-Tree-Insert(V.cluster[high(x)], low(x))
 9              else vEB-Tree-Insert(V.cluster[high(x)], low(x))
10        if x > V.max
11              V.max = x
```

Рекурзија улази у стабло величине универзума највише $\sqrt[4]{n}$. Према томе, рекурентна формула је (8.9), па функција vEB-Tree-Insert захтева $O(\log\log u)$ времена.

8.3.2.6 Уклањање произвољног елемента

Ова операција је насложенија, али се могу уочити главни случајеви:

- Стабло има само један елемент па је довољно само променити атрибуте min и max.
- Елемент који се уклања је и најмањи елемент стабла. Тада морамо да нађемо следећи најмањи елемент и да њега уклонимо из стабла.
- Елемент који се уклања је и највећи елемент стабла. Тада морамо да нађемо следећи највећи елементи да њега поставимо за највећи елемент стабла, водећи рачуна о томе да стабло може остати са само једним елементом.
- Елемент који се брише није ни најмањи ни највећи елемент стабла. Тада се елемент брише из кластера и евентуално врши ажурирање сумирајућег стабла.

```
vEB-Tree-Delete(V, x)
 1  if V.min == V.max
 2      V.min = −∞
 3      V.max = ∞
 4  elseif V.u == 2
 5      if x == 0
 6          V.min = 1
 7      else V.min = 0
 8          V.max = V.min
 9  elseif x == V.min
10      first-cluster =vEB-Tree-Find-Min(V.summary)
11      x = index(first-cluster, vEB-Tree-Find-Min(V.cluster[first-cluster]))
12      V.min = x
13      vEB-Tree-Delete(V.cluster[high(x)], low(x))
14      if vEB-Tree-Find-Min(V.cluster[high(x)]) == −∞
15          vEB-Tree-Delete(V.summary, high(x))
16              if x == V.max
17                  summary-max = vEB-Tree-Find-Max(V.summary)
18                  if summary-max == ∞
19                      V.max = V.min
20              else V.max = index(summary-max, vEB-Tree-Find-Max(V.cluster[summary-max])
21      elseif x == V.max
22          V.max = index(high(x), vEB-Tree-Find-Max(V.cluster[high(x)])
```

Наизглед се (8.9) не може применити. Проблем праве линије 13 и 15, јер могу да направе два рекурзивна позива. То јесте тачно, али је потребно додатно анализирање.

Пре него што се деси брисање елемента у линији 15, линија 14 осигурава да је кластер елемента x празан. Кластер може остати празан једино ако је елемент x био једини елемент свог кластера када се десило брисање у линији 13. Како је x био једини елемент свог кластера, уклањање троши $O(1)$ времена. Према томе, дошли смо до два случаја:

- Брисање у линији 13 се извршило за константно време.
- Брисање у линији 15 се није десило.

У оба случаја важи да се сложеност може изразити помоћу (8.9), па закључујемо да функција vEB-Tree-Delete захтева $O(\log\log u)$ времена.

8.4 Временска и меморијска сложеност

Све сложености операција које смо изразили су зависиле од величине унверзума, а не од броја елемената у стаблу. Међутим, ако за број елемената у стаблу n важи да је $u = n^{O(1)}$ или $u = n^{\log^{O(1)} n}$, тада је $\log \log u = O(\log \log n)$, па је сложеност сваке операције $O(\log \log n)$.

За меморијску сложеност ћемо прво одредити рекурентну формулу. У стаблу величине универзума u постоји сумирајуће стабло величине $\sqrt[\uparrow]{u}$. Постоји још $\sqrt[\uparrow]{u}$ показивача на $\sqrt[\uparrow]{u}$ $vEB(\sqrt[\downarrow]{u})$ стабала. Према томе, рекурентна формула је

$$M(u) = M(\sqrt[\uparrow]{u}) + \sqrt[\uparrow]{u} \cdot M(\sqrt[\downarrow]{u}) + O(\sqrt[\uparrow]{u}). \tag{8.10}$$

Мењајући $u = 2^m$ долазимо до коначне рекурентне формуле

$$\begin{aligned} M(u) &= (1 + 2^m)M(2^m) + O(\sqrt{u}) \\ M(u) &= (1 + \sqrt{u})M(\sqrt{u}) + O(\sqrt{u}). \end{aligned} \tag{8.11}$$

Да би решили ову рекурентну формулу, неопходно је да покажемо колико је пута потребно применити операцију квадратног корена пре него што број који коренујемо постане мањи или једнак 2.

Нека је број $n = 2^k$. Кореновање таквог броја представља смањивање експонента k за 1 у свакој итерацији. Број n после $\log k$ корака постаје 2. Како је $k = \log_2 n$, закључујемо да је потребно $O(\log \log n)$ корака. Узмемо ли произвољан број n, број корака се може ограничити са бројем корака првог већег броја који је степен двојке. Према томе, број потребног броја кореновања пре него што број постане мањи или једнак 2 је $O(\log \log n)$.

Користећи се добијеним резултатом, рекурентну формулу можемо написати на следећи начин

$$M(u) \leq \left(\prod_{k=1}^{\log \log u} \left(u^{\frac{1}{2^k}} + 1 \right) \right) M(2) + \sum_{k=1}^{\log \log u} O\left(u^{\frac{1}{2^k}} \right) \left(u^{\frac{1}{2^k}} + 1 \right). \tag{8.12}$$

У првом члану овог збира се јавља производ бинома. Моном са највећим степеном који одређује први члан датог збира јесте u^ϵ где је $\epsilon = \sum_{k=1}^{\log \log n} \frac{1}{2^k}$. Та сума се може ограничити геометријским редом $\sum_{k=1}^{\infty} \frac{1}{2^k} = 1$. Према томе, први члан овог збира је $o(u)$.

У другом члану збира се за $k = 1$ јавља $O(u)$, јер је $\frac{2}{2^k} = 1$, а за свако друго $k > 1$ важи $\frac{2}{2^k} < 1$ па су ти чланови $o(u)$.

Дакле, рекурентну формулу можемо упростити, па добијамо $M(u) = o(u) + O(u) = O(u)$. Према томе, меморијска сложеност ван Емде Боасовог стабла је $O(u)$.

9 Евалуација

Као што је већ речено у уводу, обрађене структуре података ћемо корисити за Дајкстрин и Примов алгоритам. Оба алгоритма ће као улазне податке имати различите врсте графова и то: стабло, ланац, звезда граф, комплетан граф и насумично генерисан граф. За сваку врсту графа постоји више тест примера са различитим бројем чворова и грана.

Сва времена су изражена у секундама, а за сваки граф су тежине грана произвољно биране.

9.1 Дајкстрин алгоритам

Таблица 1: Времена извршавања бинарног хипа

n, m	ланац	стабло	звезда граф	комплетан граф	произвољан граф
10^2	0.022	0.018	0.025	0.05	0.028
10^3	0.033	0.028	0.035	0.53	0.06
10^4	0.057	0.058	0.05		0.1
10^5	0.12	0.14	0.17		0.37
$2 \cdot 10^5$	0.25	0.27	0.28		0.6

Таблица 2: Времена извршавања d–хипа

n, m	ланац	стабло	звезда граф	комплетан граф	произвољан граф
10^2	0.02	0.02	0.024	0.06	0.024
10^3	0.032	0.024	0.04	0.47	0.09
10^4	0.06	0.06	0.05		0.08
10^5	0.13	0.16	0.14		0.34
$2 \cdot 10^5$	0.25	0.31	0.28		0.54

Таблица 3: Времена извршавања биномног хипа

n, m	ланац	стабло	звезда граф	комплетан граф	произвољан граф
10^2	0.14	0.47	0.45	0.36	1.17
10^3	1.51	1.65	1.76	8.46	5.22
10^4	14.21	15.78	16.02		17.42
10^5	66.84	69.88	72.74		79.19
$2 \cdot 10^5$	150.1	166.33	158.64		181.63

Таблица 4: Времена извршавања Фибоначијевог хипа

n, m	ланац	стабло	звезда граф	комплетан граф	произвољан граф
10^2	0.021	0.12	0.21	0.24	2.22
10^3	0.27	0.47	0.46	10.81	5.78
10^4	1.24	5.12	5.18		8.15
10^5	32.93	56.91	57.73		75.05
$2 \cdot 10^5$	58.74	72.88	86.44		24.04

Таблица 5: Времена извршавања сплеј стабла

n, m	ланац	стабло	звезда граф	комплетан граф	произвољан граф
10^2	0.03	0.019	0.027	0.04	0.024
10^3	0.03	0.031	0.034	0.65	0.05
10^4	0.62	0.067	0.1		0.2
10^5	0.14	0.19	0.4		0.72
$2 \cdot 10^5$	0.27	0.41	0.52		0.78

Таблица 6: Времена извршавања ван Емде Боасовог стабла

n, m	ланац	стабло	звезда граф	комплетан граф	произвољан граф
10^2	0.03	0.71	0.37	0.38	4.2
10^3	2.35	3.31	1.88	22.23	12.16
10^4	18.79	35.28	27.93		19.62
10^5	95.64	87.42	89.19		159.83
$2 \cdot 10^5$	169.39	160.74	170.44		216.12

9.2 Примов алгоритам

Таблица 7: Времена извршавања бинарног хипа

n, m	ланац	стабло	звезда граф	комплетан граф	произвољан граф
10^2	0.028	0.035	0.033	0.47	0.021
10^3	0.041	0.047	0.049	0.93	0.15
10^4	0.074	0.079	0.066		0.23
10^5	0.45	0.37	0.38		1.11
$2 \cdot 10^5$	0.67	0.71	0.74		1.86

Таблица 8: Времена извршавања d–хипа

n, m	ланац	стабло	звезда граф	комплетан граф	произвољан граф
10^2	0.035	0.033	0.034	0.44	0.029
10^3	0.044	0.045	0.052	0.84	0.12
10^4	0.071	0.08	0.069		0.25
10^5	0.48	0.37	0.39		1.1
$2 \cdot 10^5$	0.73	0.68	0.74		1.68

Таблица 9: Времена извршавања Фибоначијевог хипа

n, m	ланац	стабло	звезда граф	комплетан граф	произвољан граф
10^2	0.027	0.09	0.07	2.14	3.13
10^3	0.32	0.22	0.36	223.06	25.81
10^4	3.02	1.82	0.066		19.74
10^5	57.89	22.52	29.756		169.84
$2 \cdot 10^5$	113.4	23.09	59.93		248.91

Таблица 10: Времена извршавања биномног хипа

n, m	ланац	стабло	звезда граф	комплетан граф	произвољан граф
10^2	0.2	0.41	0.45	1.36	1.79
10^3	2.03	1.98	1.76	188.87	12.22
10^4	13.17	16.01	16.02		14.24
10^5	78.42	70.77	72.74		179.19
$2 \cdot 10^5$	170.56	176.5	161.06		237.27

Таблица 11: Времена извршавања сплеј стабла

n, m	ланац	стабло	звезда граф	комплетан граф	произвољан граф
10^2	0.03	0.031	0.035	0.04	0.027
10^3	0.051	0.048	0.055	0.65	0.14
10^4	0.87	0.067	0.089		0.37
10^5	0.64	0.42	0.53		1.28
$2 \cdot 10^5$	0.7	0.87	0.81		2.02

Таблица 12: Времена извршавања ван Емде Боасовог стабла

n, m	ланац	стабло	звезда граф	комплетан граф	произвољан граф
10^2	0.05	0.89	0.77	2.44	3.84
10^3	3.23	3.76	3.54	232.39	18.96
10^4	24.54	33.82	29.29		47.26
10^5	95.64	81.68	92.9		191.22
$2 \cdot 10^5$	165.9	160.74	170.44		224.63

10 Закључак

Циљ овог матурског рада је био да представи и анализира приоритетне редове који би могли да се искористе у имплементацији Дајкстриног и Примовог алгоритма.

10.1 Резултати

Евалуација је обављена успешно. Постигнути су жељени резултати и утврђено је да бинарни хип представља најбољи избор приоритетног реда за поменуте алгоритме, у случају када број чворова и број грана графа не прелази 10^5. Остали приоритетни редови су се показали као изузетно неефикасни, изузев сплеј стабла које негде чак и надмашује бинарни хип. Међутим, како је бинарни хип садржан у готово свим модерним програмским језицима, представља логичан избор при имплементацији.

10.2 Стечено знање

Приликом израде матурског рада сам се детаљније упознао са израчунавањем сложености операција као и потпуно новим концептом за мене – амортизованом анализом. Увидео сам да чак и теоријски оптимална струткура података не мора да има практичну корист. Такође сам побољшао своје знање C++-а, а специјално употребу показивача.

10.3 Даљи рад

Многи приоритетни редови нису обрађени. Неки од њих су *Red-black tree*, *Pairing heap*, *Brodal queue*, *Skew heap*. Резултати овог матурског рада могу послужити за компарацију са поменутим приоритетним редовима, као и са било којом другом структуром података.

Приликом израде самог рада сам се сусрео са недостатком адекватне литературе на српском језику. Сматрам да овај матурски рад може послужити као квалитетна литература за описане структуре података.

10.4 Захвалност

Желео бих да изразим захвалност

- **Петру Величковићу** – мом ментору, који ми је значајно помогао при изради матурског рада. Дао ми је низ сугестија, критика и савета и увек је био на располагању за све што ми је требало.
- **Владимиру Миленковићу и Филипу Весовићу** – другарима из школе, захваљујући којима сам значајно напредовао на пољу такмичарског програмирања.

11 Литература

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms (3rd ed.)*, The MIT Press, 2009.

[2] Дејан Живковић, *Основе дизајна и анализе алгоритама*, Рачунарски факултет, 2007.

[3] Зоран Каделбург, Владимир Мићић, Срђан Огњановић, *Анализа са алгебром 2*, Круг, 2014.

[4] Johnson B. Donald, *Priority queues with update and finding minimum spanning trees*, Information Processing Letters, 1975.

[5] Jean Vuillemin, *A Data Structure for Manipulating Priority Queues*, Journal of the ACM, 1978.

[6] Michael L. Fredman, Robert E. Tarjan, *Fibonacci heaps and their uses in improved network optimization algorithms*, Journal of the ACM, 1987.

[7] Daniel D. Sleator, Robert E. Tarjan, *Selft–Adjusting Binary Search Trees*, Journal of the ACM, 1985.

[8] Donald Knuth, *The Art of Computer Programming (3rd ed.)*, Addison–Wesley, 1997.

[9] Peter van Emde Boas, *Preserving order in a forest in less than logarithmic time*, Proceedings of the 16th Annual Symposium on Foundations of Computer Science, 1975.

[10] Peter van Emde Boas, *Preserving order in a forest in less than logarithmic time and linear space*, Information Processing Letters, 1977.

[11] Peter van Emde Boas, Rob Kaas, Eduard Zijlstra, *Design and implementation of an efficient priority queue*, Mathematical Systems Theory, 1976.