

A heap of heaps, или купчина пирамиди

Андрей Дренски, 20 апр. 2021г.

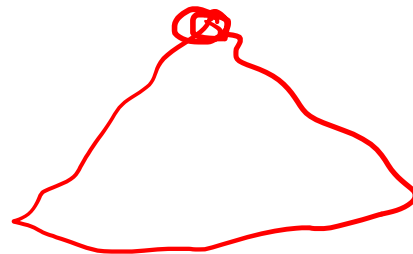
Що е то пирамида?

heap property

- дърво със свойството, че всеки възел е не по-голям (или не по-малък) от всеки свой наследник
- идват във всякакви форми и размери
- много често служат за бързи имплементации на приоритетна опашка
 - толкова често, че двата термина са взаимно-заменяеми

Добре, а що е то
приоритетна
опашка?

■ ~~това е пирамида~~



- абстрактна структура от данни, както „списък“ или „map“
- поддържа следните операции:
 - проверка дали е празна/колко елемента съдържа
 - вмъкване на елемент „с приоритет“ (**insert**)
 - изваждане на елемента с най-малък приоритет (**extractMin**)
 - само прочитане на същия, без модификации (**peek/findMin**)
 - намаляване на приоритета на даден елемент (**decreaseKey**)
 - често най-проблемната операция; понякога можем и без нея
 - сливане на две приоритетни опашки (**merge**)
 - или поне сливане на две „части“ от дадена опашка

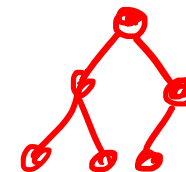
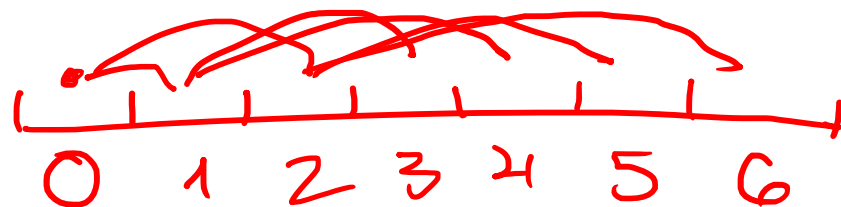
Добре, а що е то
приоритетна
опашка?

- по подразбиране ще говорим за **min**-приоритетни опашки
- в общия случай няма ограничения за приоритетите
 - ако имаше, могат да се използват различни структури

Baby's first heap

Двоична пирамида / BinaryHeap

$$\frac{i-1}{2} \leftarrow i \rightarrow 2i+1; 2i+2$$



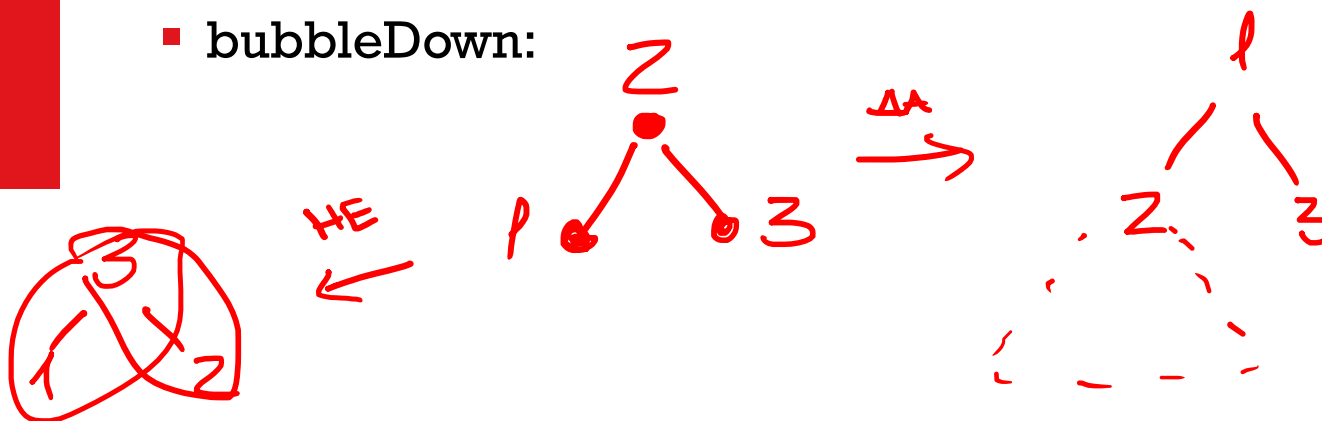
Binary heap

- най-простият и разпространен вариант
- представлява двоично попълнено дърво с пирамидалното свойство (heap property)
- възлите на дървото живеят в масив, нарастващ/намаляващ само отдясно → няма нужда да пазим „адреси“ на наследниците
 - това е implicit data structure → $\mathcal{O}(1)$
- най-важното следствие: височината на дървото винаги е точно $\lceil \lg n \rceil$

Binary heap

- всички операции се свеждат до **bubbleUp**/**bubbleDown**
- **bubbleUp**:

- **bubbleDown**:

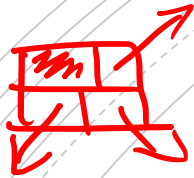




Binary heap

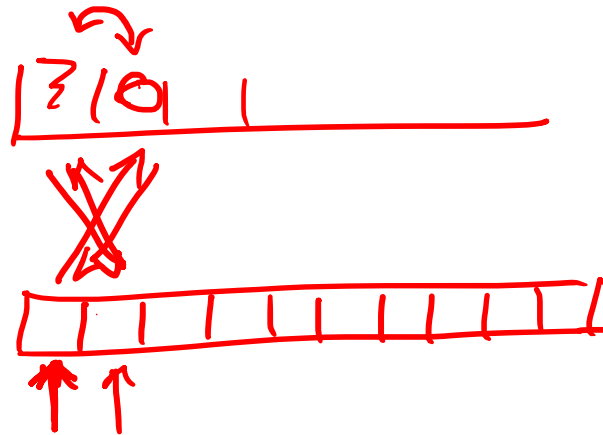
- **insert**: вмъкваме нов елемент вдясно; **bubbleUp** $O(\lg n)$
- **extractMin**: разменяме върха с последния елемент; **bubbleDown** $O(\lg n)$
- **peekMin**: прочитаем първия елемент (винаги върха) $O(1)$
- **decreaseKey**: проблем – как различаваме елементите? $O(\lg n)$
 - подробности след малко
- **merge**: няма добро решение $O(n)$
- Бонус: **in-place** конструиране от n елемента: **bubbleDown** на всички не-листа, започвайки от най-дясното $\rightarrow O(n)$, което е оптимално

Binary heap



■ за decreaseKey:

- проблемът е в разместването на самите елементи
- дървовидните структури се преорганизират само чрез смяна на вътрешните указатели към възлите
- тук нямаме такива

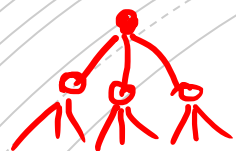


■ губим имплицитност :(

- това се води „компактна“ структура

Защо да спираме на двоично дърво?

d-ична пирамида / d-ary heap



$$\frac{i-1}{d} \leftarrow i \rightarrow d^*i+1 \rightsquigarrow d^*i+d$$

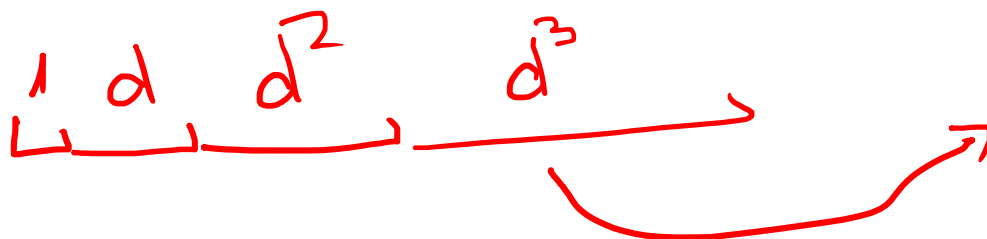
- напълно аналогична на двоичната пирамида
 - попълнено d -ично дърво \rightarrow височината ще е $\log_d n$
- отново имплицитна структура
- tradeoff между по-бърз bubbleUp и по-бавен bubbleDown

d-ary heap

$$O(d \log_d n) \stackrel{d=4}{\sim} 2 \lg n < 8 \log_8 n$$

$O(\log_d n)$

- $d=4$ постига достатъчно добър компромис



Един по-левашки пример

Skew heap

Skew heap

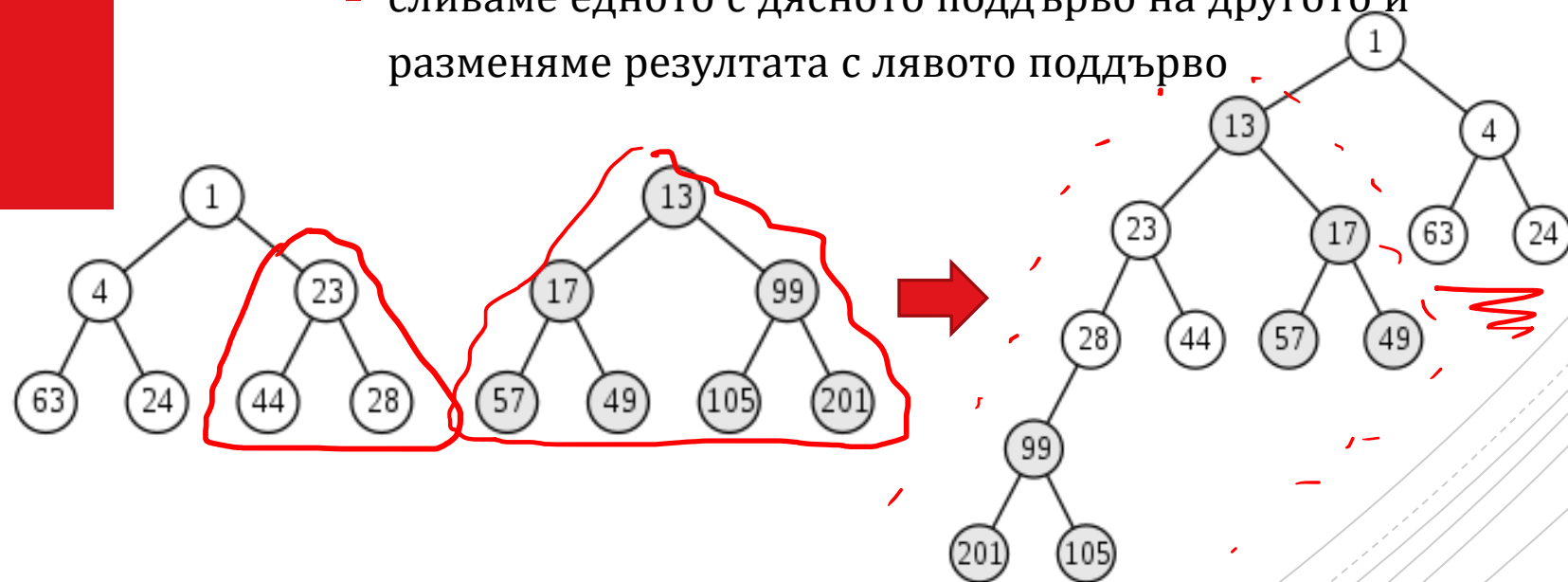
- представлява самонагласящо се ляво-наклонено дърво

→ най-краткият път до листо
започва надясно

- какво значи ляво-наклонено?
- а какво значи самонагласящо се?

- тук основната операция е сливане на две дървета:

- сливаме едното с дясното поддърво на другото и
разменяме резултата с лявото поддърво



Skew heap

- тривиална за имплементиране на функционален стил:

```
data SkewHeap a = Empty | Node a (SkewHeap a) (SkewHeap a)

insert :: Ord a => a -> SkewHeap a -> SkewHeap a
insert x sh = singleton x <> sh
  where singleton x = Node x Empty Empty

(<>) :: Ord a => SkewHeap a -> SkewHeap a -> SkewHeap a
sh1 <> Empty = sh1
Empty <> sh2 = sh2
sh1@(Node v1 l1 r1) <> sh2@(Node v2 l2 r2)
  .... | v1 < v2 ... = Node v1 (sh2 <> r1) l1
  .... | otherwise = Node v2 (sh1 <> r2) l2

extractMin :: Ord a => SkewHeap a -> Maybe (a, SkewHeap a)
extractMin Empty = Nothing
extractMin (Node val left right) = Just (val, left <> right)
```

merge: $\Theta(\lg n)$

Skew heap

- insert: конструираме дърво с 1 възел и merge

$\Theta(\lg n)$

- extractMin: премахваме корена и merge на двете поддървета

$\Theta(\lg n)$

- peekMin: прочитаме стойността в корена

$\Theta(1)$

- decreaseKey: bubbleUp

$\Theta(\lg n)$

→ трябва parent pointer

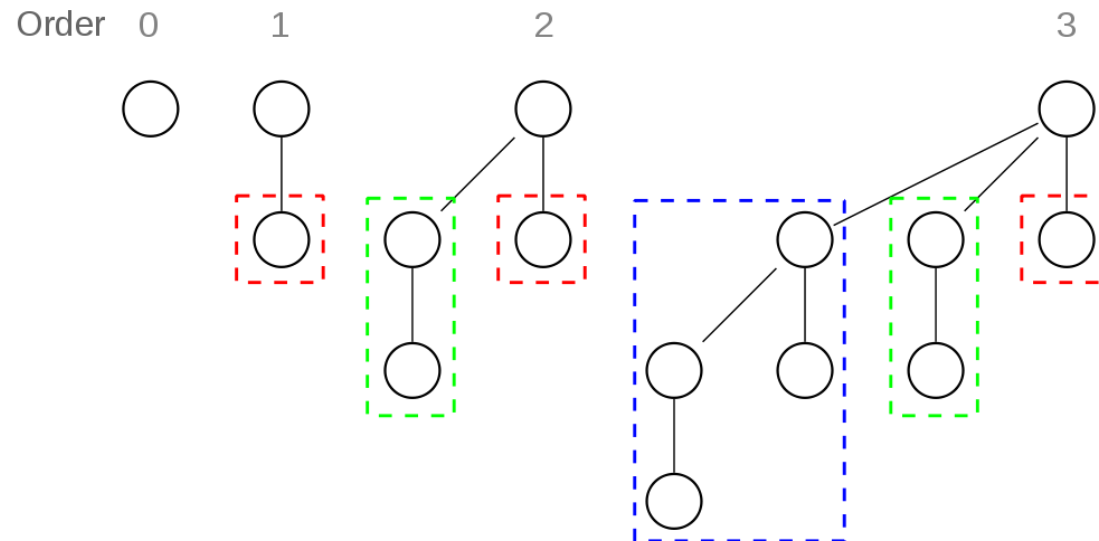
Биномна пирамида

Binomial heap

Binomial heap

- основна дефиниция: биномно дърво от ранг d

- също с **heap-property**



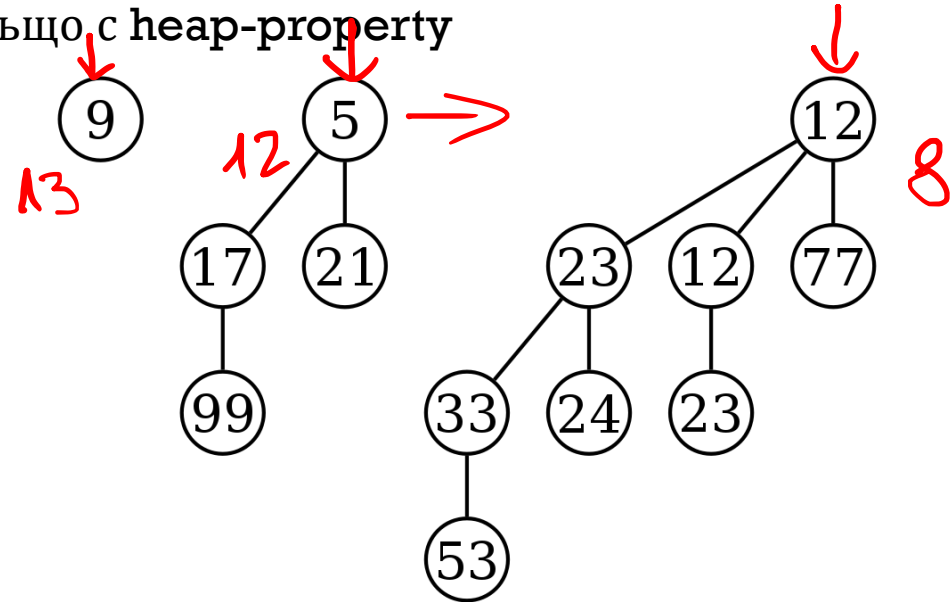
- пирамида с n елемента е списък от дървета с различни рангове
 - за всяко n има уникален такъв списък
 - по едно дърво с ранг = позицията на всеки вдигнат бит в двоичния запис на n
 - най-добре да се пазят в нарастващ ред на ранговете

Binomial heap

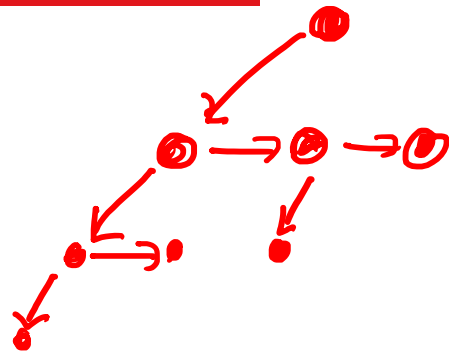
$$n = 13_{10} = 1101_2$$

- основна дефиниция: биномно дърво от ранг d

- също с heap-property

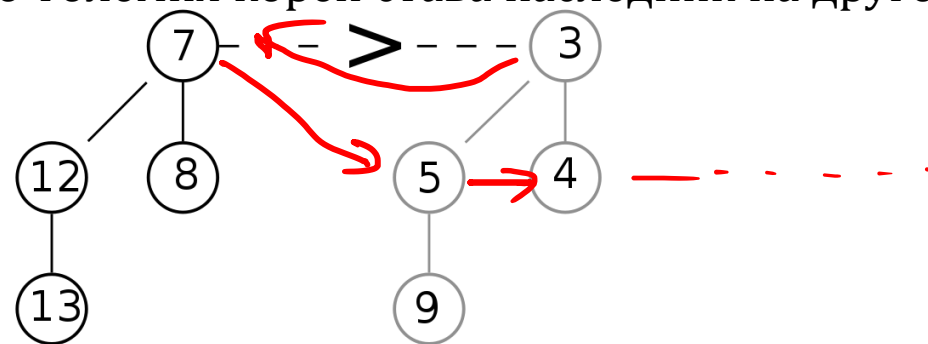


- пирамида с n елемента е списък от дървета с различни рангове
 - за всяко n има уникален такъв списък
 - по едно дърво с ранг = позицията на всеки вдигнат бит в двоичния запис на n
 - най-добре да се пазят в нарастващ ред на ранговете

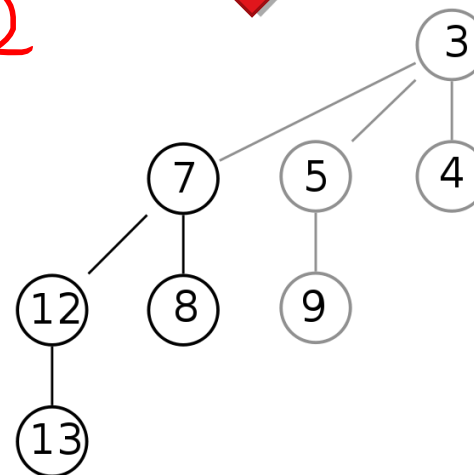


Binomial heap

- можем да обединяваме две дървета само ако са от един и същ ранг
- това с по-големия корен става наследник на другото:



$$2^d + 2^d = 2^{d+1}$$



$$\begin{array}{r}
 11 \\
 1011 \\
 11 \\
 \hline
 1110
 \end{array}$$

Binomial heap

- отново основната операция е сливане на две пирамиди
 - аналогична на сливането на сортирани списъци
 - не забравяме инвариантата
 - реално това е събиране на двоични числа



Binomial heap

- **insert**: конструираме дърво от ранг 0 и **merge**

→ $\Theta(1)$ аморт.

- **peekMin**: намираме най-малкия корен в списъка

- можем и да го кешираме

→ $\Theta(1)$

- **extractMin**: обръщаме наследниците на най-малкия във валидна пирамида и **merge** с остатъка

→ $\Theta(\lg n)$

- **decreaseKey**: bubbleUp → $\Theta(\lg n)$

- трябва **parent** поинтъри

- отново трябва **two-way referencing**, като **binary heap**

За тази дори нямам
име, камо ли шега

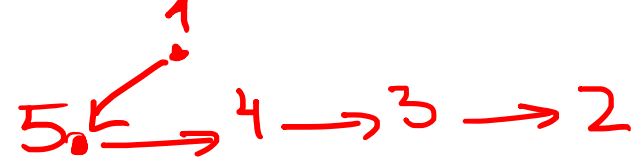
Pairing heap

Pairing heap



- дърво с произволен брой наследници и **heap property**
 - представяне: ляв син/десен брат
- много просто представяне, много прости операции (без една)

Pairing heap



- **merge**: добавяме дървото с по-голям корен в списъка с наследници на другото



- **insert**: познайте
 - конструираме дърво с 1 възел и **merge**

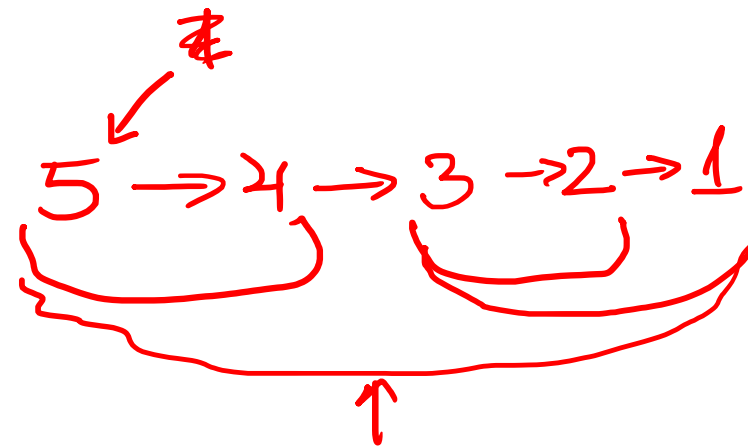
$\sim(1)$

- **findMin**: достъпваме корена на дървото

$\sim(1)$

Pairing heap

- **decreaseKey**: премахваме цялото поддърво, променяме му корена и **merge** обратно
 - трудно за анализ по сложност, доказано $\omega(1)$, но $o(\lg n)$
- **extractMin**: премахваме корена и сливаме децата му по специален начин
 - по двойки, и полученото насъбираме отзад-напред (!)

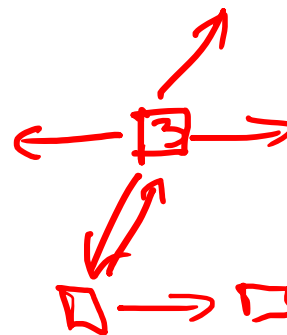


Последния бос на пирамидите

Фибоначиевата пирамида / Fibonacci heap

Fibonacci heap

- **tradeoff** между добри времена на теория и лоши на практика
 - сложната имплементация не помага
 - голям **memory overhead**
- отново списък от дървета с произволен брой наследници *и min-heap property*
 - този път в двусвързан списък



*210 байта
за един int*

Fibonacci heap

- **insert:** конструираме дърво с 1 възел и merge

→ $\Theta(1)$

- **findMin:** намиране на най-малкия корен от списъка

- отново може да бъде кеширан

→ $\Theta(1)$

- **merge:** залепяне на двата списъка от дървета

→ $\Theta(1)$

- **extractMin & decreaseKey:** magic

↓
 $\Theta(\lg n)$ аморт.

↓
 $\Theta(1)^*$ аморт

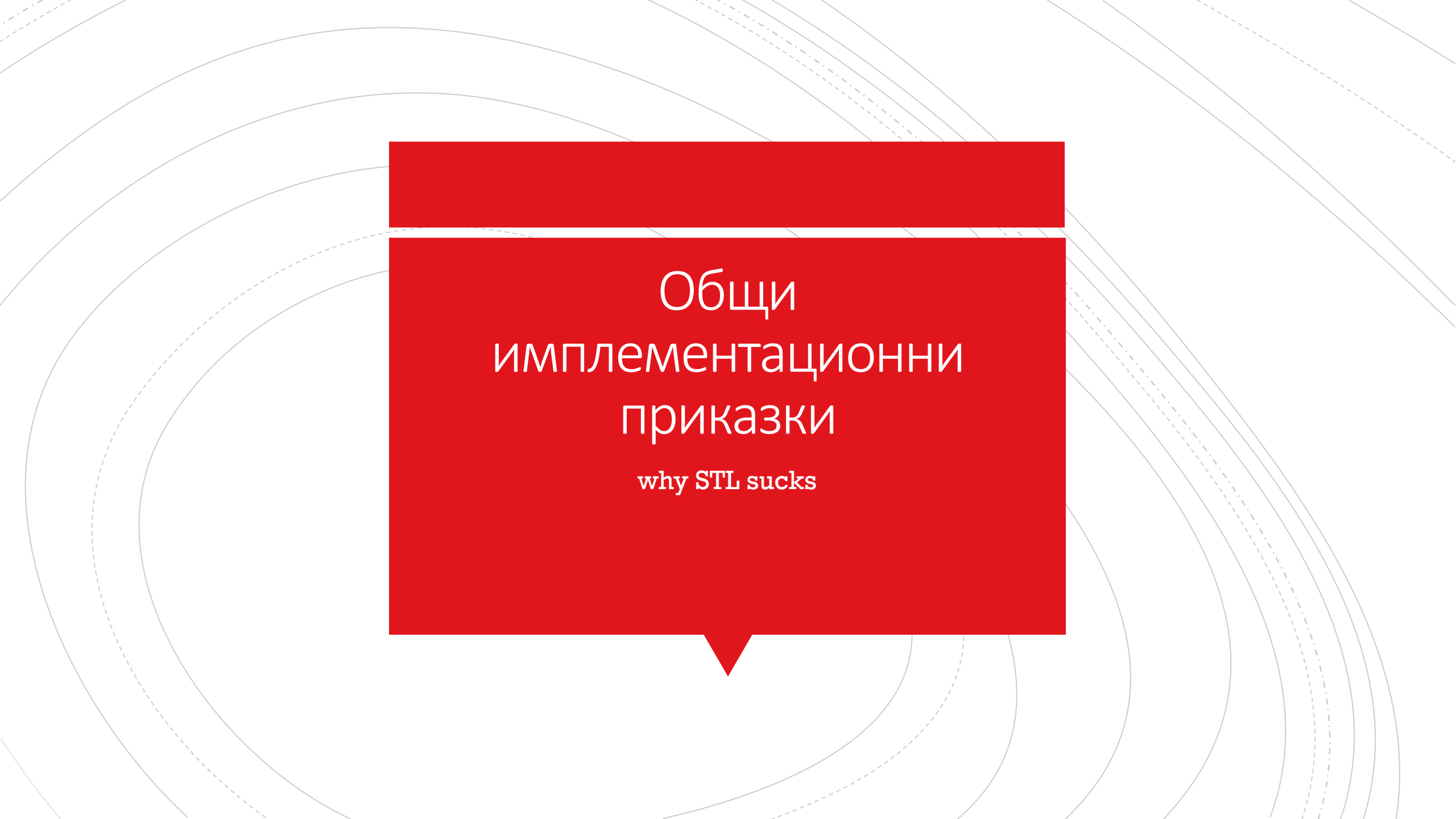
The background features a series of concentric circles in light gray, some solid and some dashed, creating a ripple effect. In the center, there is a large red speech bubble with a white border. The text is contained within this bubble.

А може ли по-бързо?

Достигнали ли сме долните граници за сложност
на операциите?

Съществуват ли
по-бързи
имплементации
?

- това, че някои сложности са амортизирани, не е проблем
- забелязваме, че винаги едно измежду `insert` и `extractMin` е $\omega(1)$ $\rightarrow \Theta(\lg n)$
 - не може ли и двете да са $O(1)$, дори и амортизирано..?
- НЕ – защото бихме получили $O(n)$ сортиране на n елемента, използвайки само директни сравнения
 \rightarrow това е доказано невъзможно :(

The background features a series of concentric circles in light gray, some solid and some dashed, creating a ripple effect. In the center, there is a red speech bubble with a pointed bottom. The text is contained within this bubble.

Общи имплементационни приказки

why STL sucks

Общи приказки

- така описаните структури са твърде общи
 - тоест, безполезни
- за всяка* конкретна употреба могат да се приложат отделни оптимизации
- много често работим само с една пирамида и всички промени са в нея
 - just use a vector
- много често знаем предварително максималния ѝ размер
 - allocate once

The background features a series of concentric circles in light gray, some solid and some dashed, creating a ripple effect. In the center, there is a red speech bubble with a white border. The text is contained within this bubble.

Code once,
measure thrice

Примери и сравнения на пирамиди

Пример 1:
проста
приоритетна
опашка

- проблем: измежду n елемента да намерим k -те най-малки (очевидно $k \ll n$)
- можем да ползваме пирамида за забързване на тривиалното решение

1, 3, 5, 8, 10 \leftrightarrow 4 $\rightarrow O(nk)$
max-heap $\rightarrow O(n \lg k)$

- естествено, можем да ползваме и друг алгоритъм

Пример 1:
проста
приоритетна
опашка

- приложение на проблема: reservoir sampling
- измежду n елемента да изберем k произволни с равна вероятност (!)

```
int getRandomNumber()  
{  
    return 4; // chosen by fair dice roll.  
              // guaranteed to be random.  
}
```

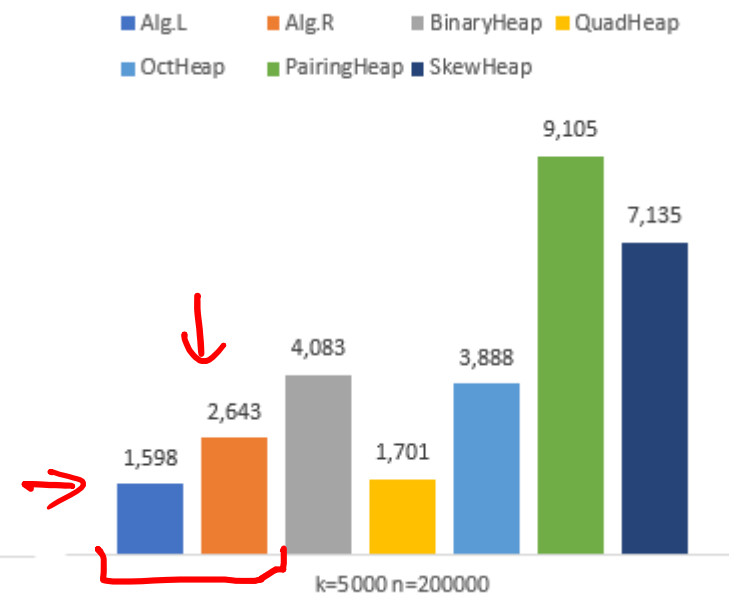
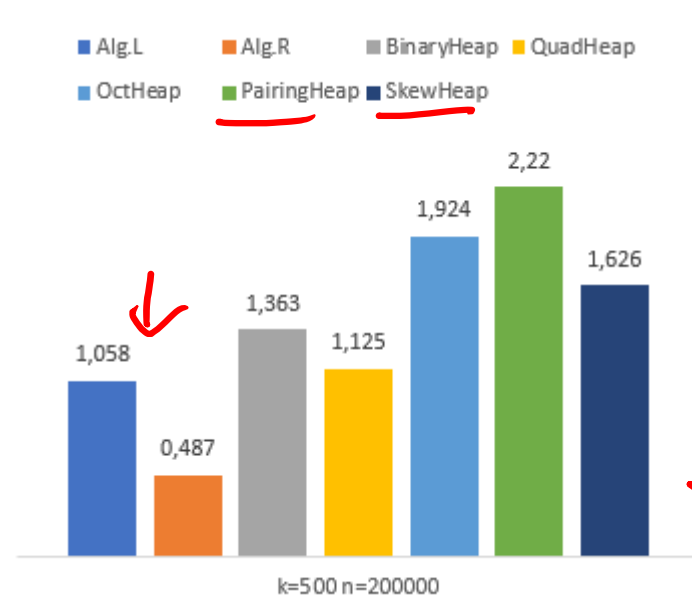
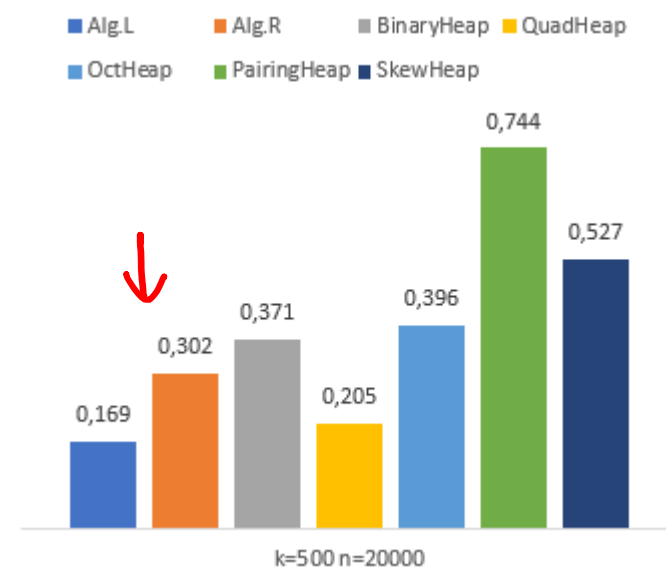
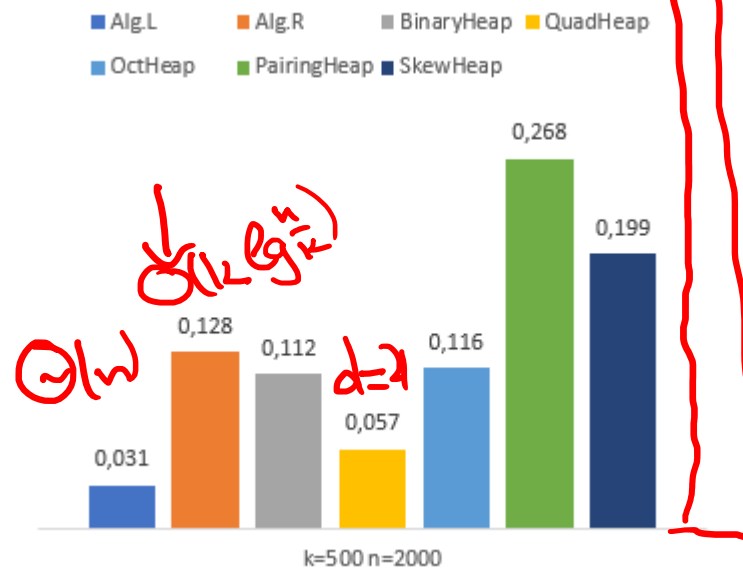
- решение: генерираме произволно число към всеки елемент и вмъкваме в пирамида, сравняваща приоритети

uniform distribution

- естествено, можем да ползваме и друг алгоритъм

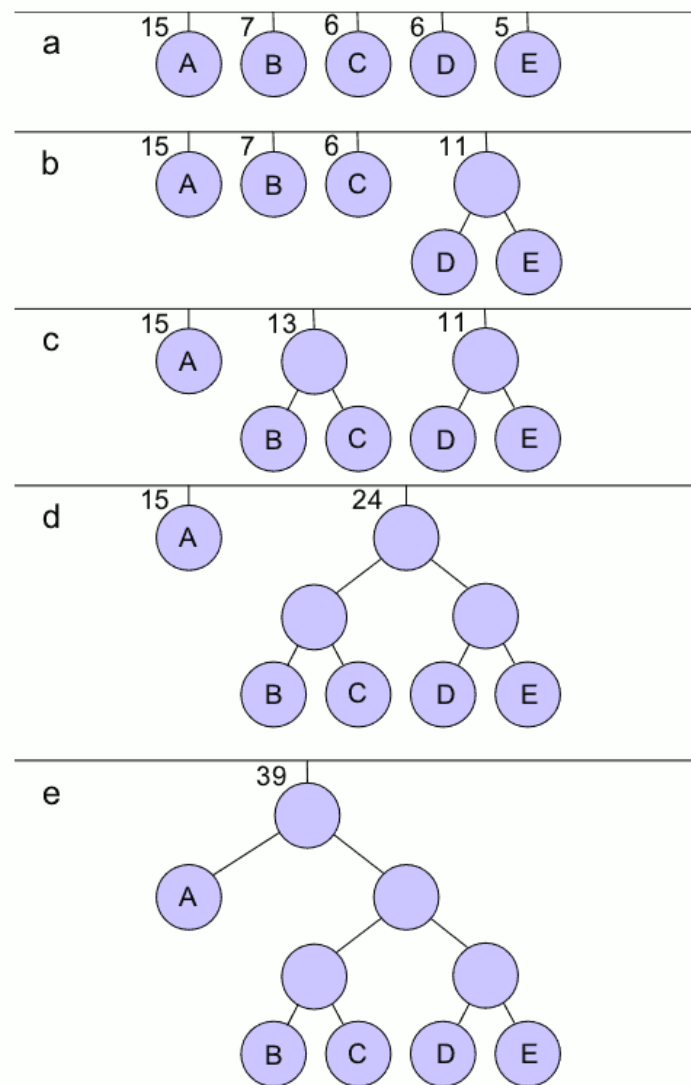
Пример 1: проста приоритетна опашка

результати:



Пример 2: Хъфманово кодиране

- проблем: генериране на хъфманово дърво по честотна таблица



Пример 3:
алгоритмите на
Прим/Дийкстра

$$\rightarrow T(n, m) = n \cdot T_{em} + m \cdot T_{dk} \begin{matrix} \nearrow (n+m) \lg n \\ \searrow n \lg n + m \end{matrix}$$

- тук можем да се възполваме от `decreaseKey`
- резултати: подобни на предишните :(

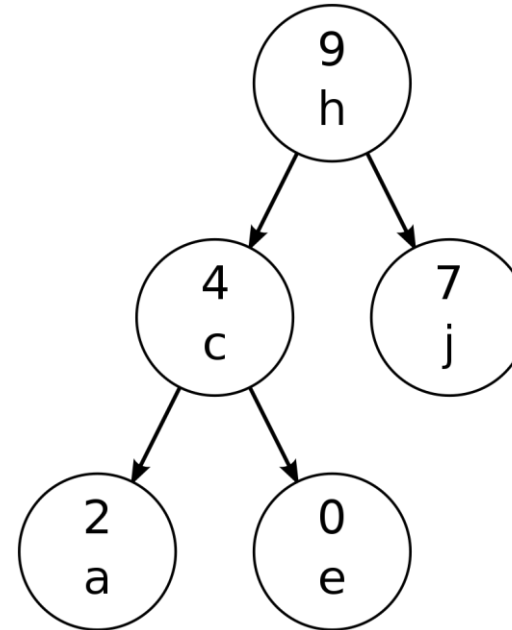
- ~~Витърбана~~ n е-та с ∞
и един с приоритет
- докато има елементи
 - `extractMin`
 - за \forall съседни натови
`decreaseKey` (пробвай)

tree + heap = ~~B.H.A.~~ treap

една рандомизирана структура от данни за общо ползване

Treap

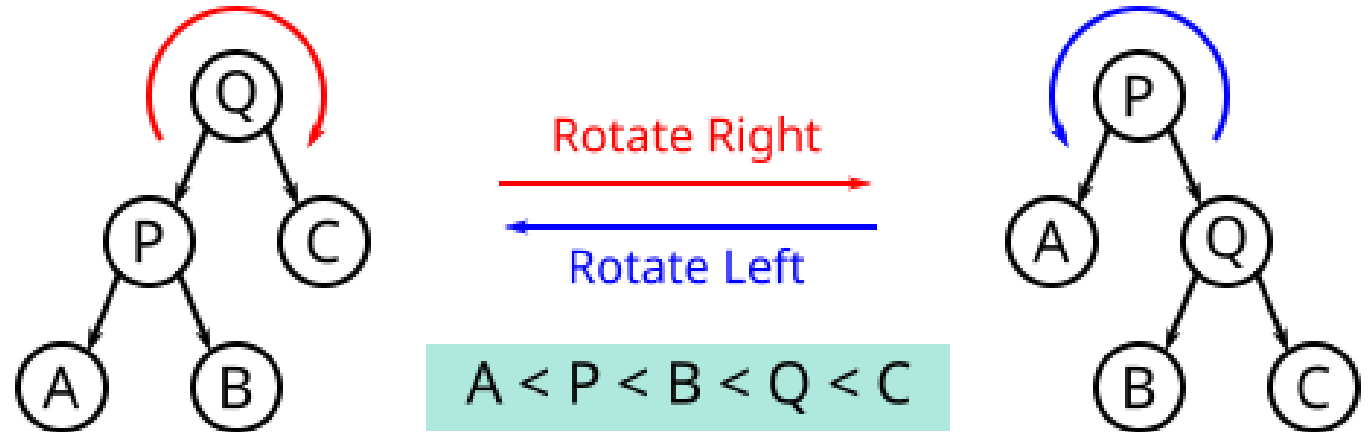
- двоично дърво, в което всеки възел получава произволен приоритет при създаване
- стойностите във възлите образуват **BST**
- приоритетите същевременно образуват **heap**



Binary search tree

- тайната: стандартни ротации
 - запазват BST-свойството
 - с тях разместваме само приоритетите

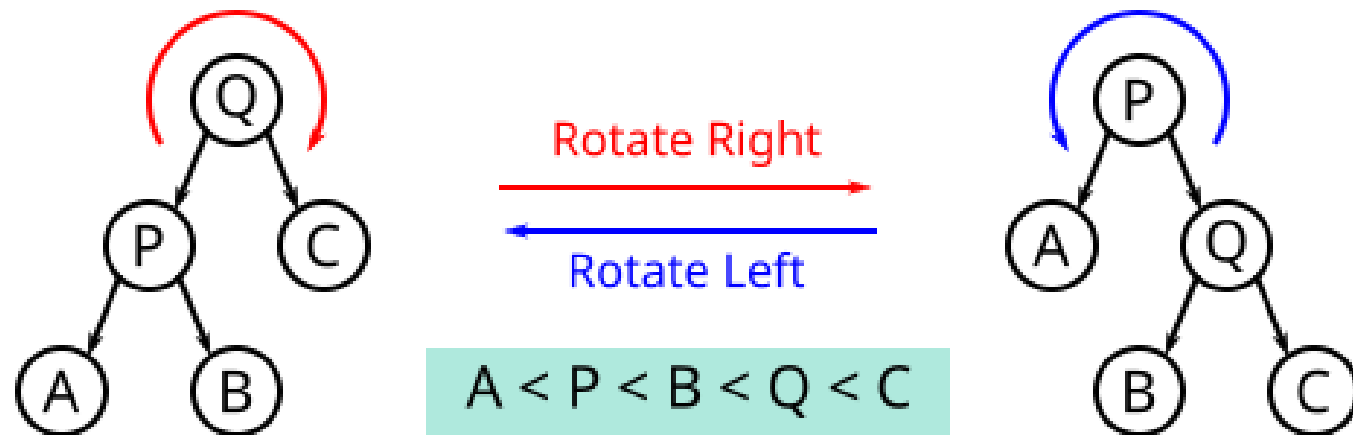
Treap



Treap

- insert:

- добавяме стойност като в обикновено BST
- използваме ротации за **bubbleUp** докато възстановим **heap**-свойството



- delete:

- използваме ротации за **bubbleDown** до листо
- кръцваме листото

отлично $O(\lg n)$

Treap

- search:

- като в стандартно двоично наредено дърво
- можем при всеки достъп да генерираме нов приоритет
→ ако е по-малък от текущия, **bubbleUp**
- бихме получили структура, подходяща за някакъв кеш

- обхождания:

- като в стандартно двоично наредено дърво

- бонус:

- обединение на два **treap**-а → **bubbleDown** на фалшив общ корен
- разделяне на **treap** по стойност → **bubbleUp** и изтриване

$O(\lg n)$



Благодаря за вниманието!

github.com/Andreshk

compile-time
verified →

Бонус:

доказано* коректна имплементация
на binomial heap