Djupare och grundare titt på prioritetsköer

Tillämpad Algoritmisk Problemlösning (Våren 2026)

Martin Berglund, mbe@cs.umu.se

Låt oss först klura på ett problem

Take as input a sequence of $2n$ real numbers. Design an $O(n \log n)$ algorithm that partitions the numbers into n pairs, with the property that the partition minimizes the maximum sum of a pair.

For example, say we are given the numbers (1,3,5,9). The possible partitions are ((1,3),(5,9)), ((1,5),(3,9)), and ((1,9),(3,5)). The pair sums for these partitions are (4,14), (6,12), and (10,8). Thus the third partition has 10 as its maximum sum, which is the minimum over the three partitions.

Lösning

Algoritmen inte så svår, men argument för korrekthet?

Inte *komplicerat,* men sällan systematiskt- eller "kunna-utantill"-tänkande.

(På tavla...)

Antag $x_1, \ldots, x_n$ har minimal max-par-summa $\leq k$, då har vi tre fall:

1. Bas-fall...
2. "Verkligt" induktivt steg...
3. Steg som visar sig omöjligt...

Vi tar ett problem till...

Give an efficient algorithm to determine whether two sets (of size m and n) are disjoint. Analyze the complexity of your algorithm in terms of m and n. Be sure to consider the case where m is substantially smaller than n.

Snabb repetition: prioritetsköer och heaps

Heaps är den typiska implementationen av en *prioritetskö**

Definition

En *prioritetskö* P representerar mängd $X = \{x_1, \ldots, x_n\}$ med åtminstone dessa operationer definierade:

- *pop*(P) uppdaterar P till att representera mängden $X \setminus \{\min X\}$ och returnerar $\min X$,
- *insert*(P, e) uppdaterar P till att representera $X \cup \{e\}$.

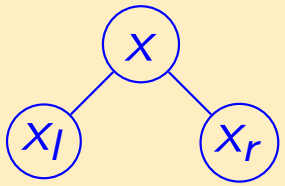
Andra vanliga: *find-min*, *delete-min*, *decrease-key*, *merge*.

Snabb repetition: prioritetsköer och heaps

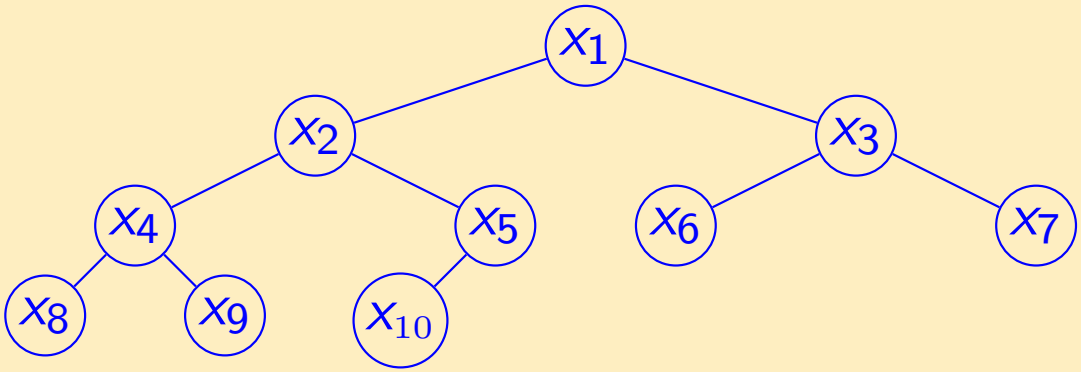
Definition

En binär heap är ett binärt träd med egenskaperna:

- Varje nod innehåller ett värde.
- Varje nods värde är större än dess förälders värde ($x_l > x$ och $x_r > x$).

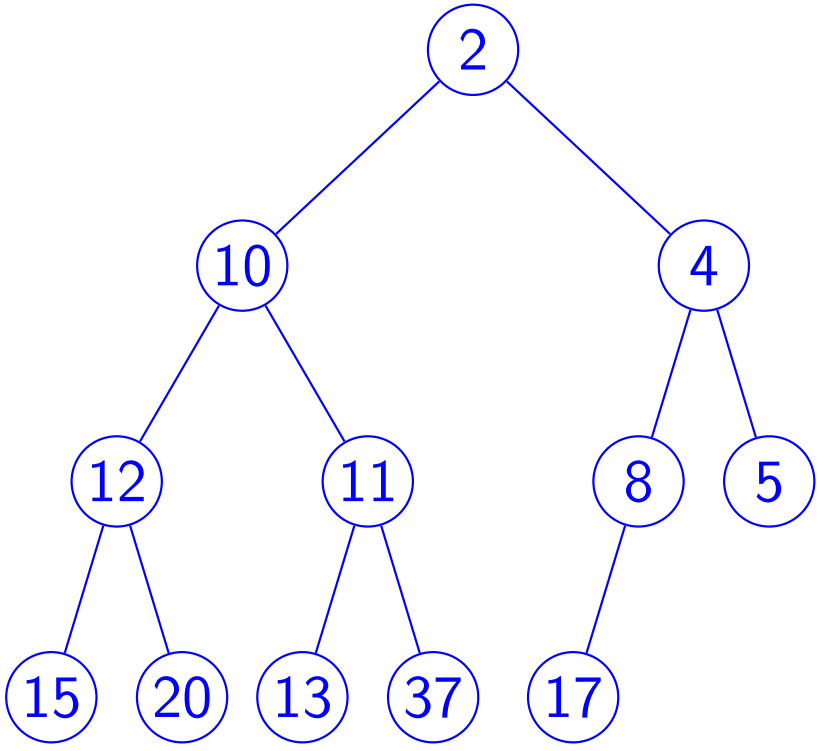


- Trädet är *komplett;* varje nivå (grupp av noder med samma avstånd från roten) av trädet, förutom möjligen den "lägsta", är fullt populerade. Om den sista nivån (längst från roten) inte är fullt populerad saknas noder på *höger* sida.

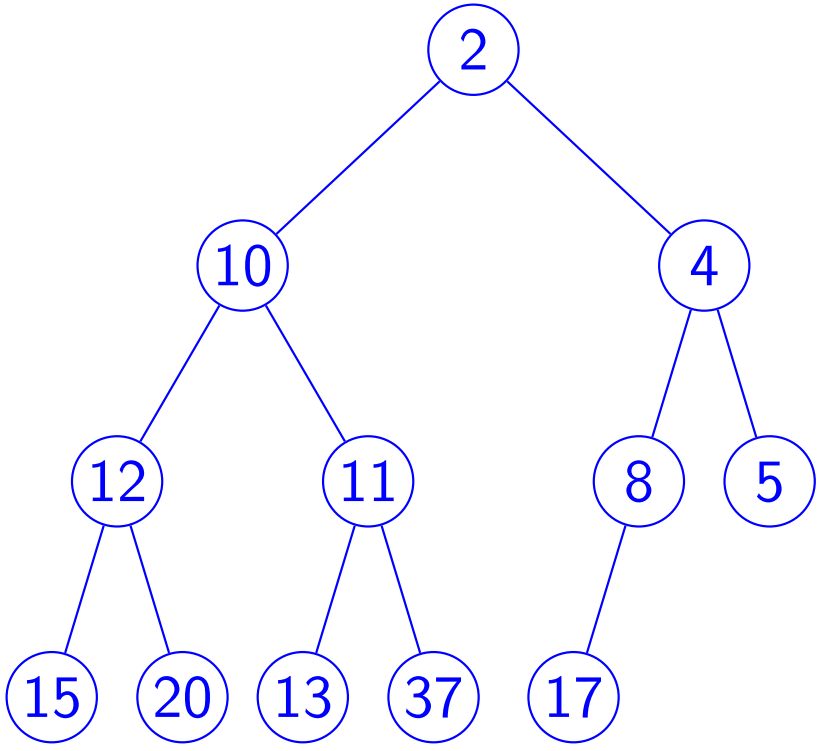


Snabb repetition: prioritetsköer och heaps

insert med värdet 3:

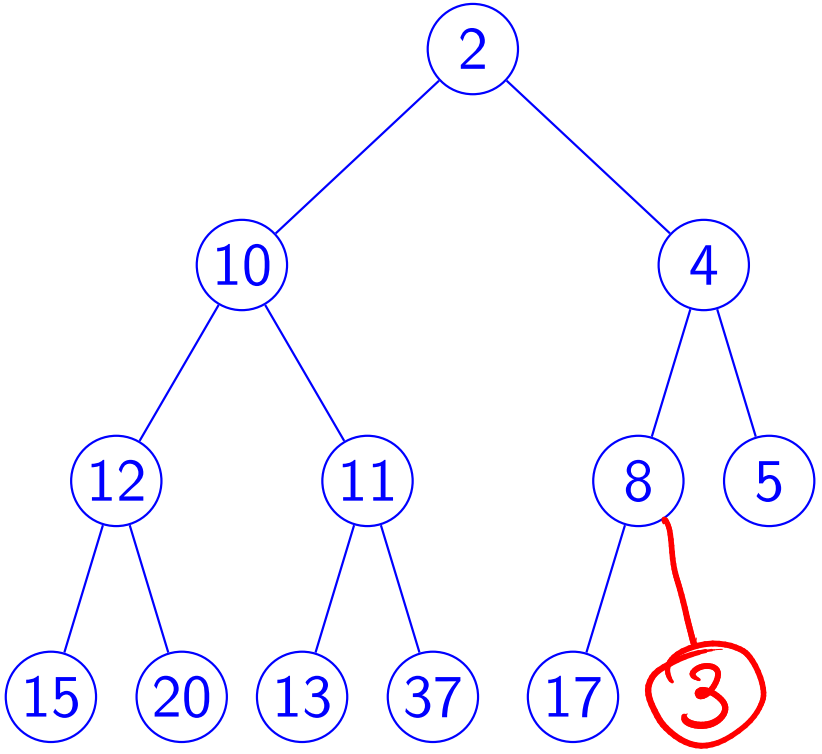


pop (värdet 2):



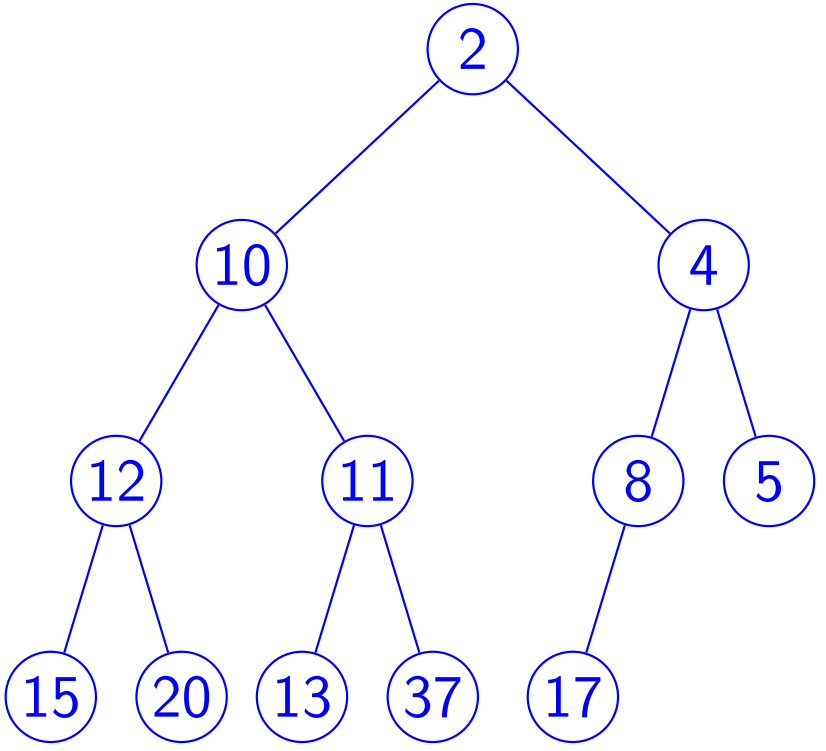
Snabb repetition: prioritetsköer och heaps

insert med värdet 3:



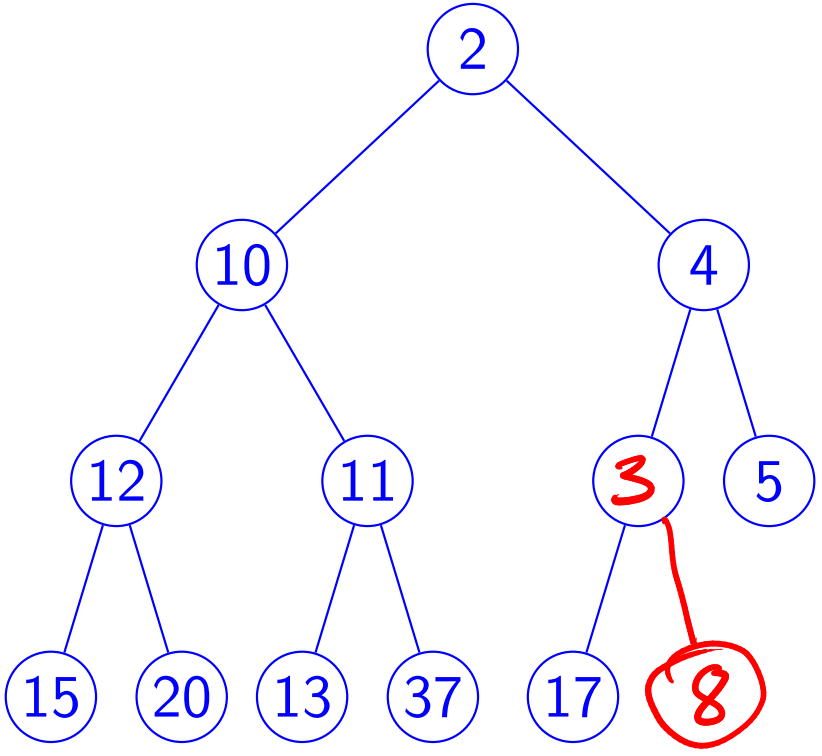
Men 8>3

pop (värdet 2):



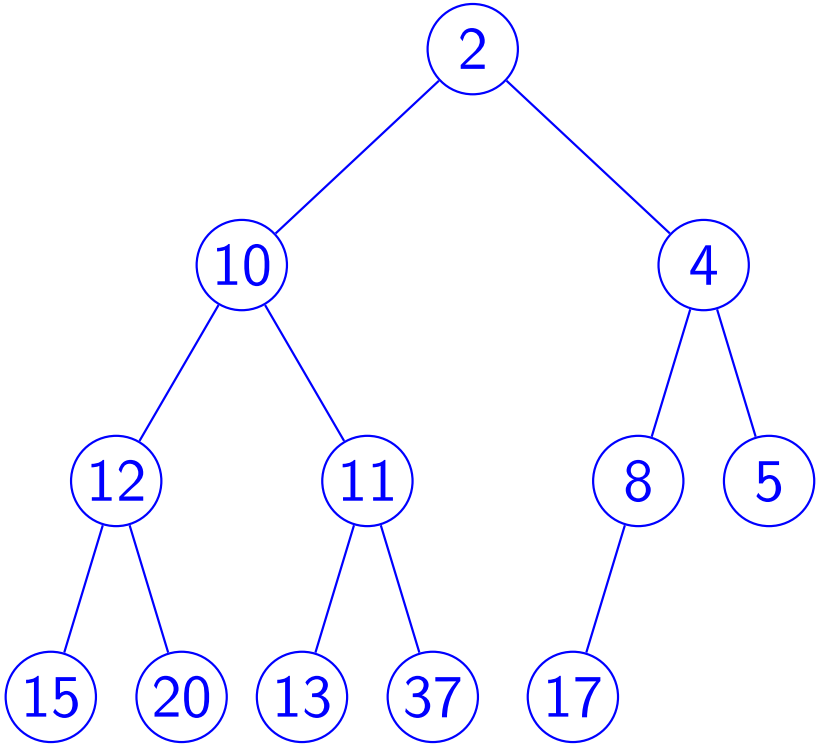
Snabb repetition: prioritetsköer och heaps

insert med värdet 3:



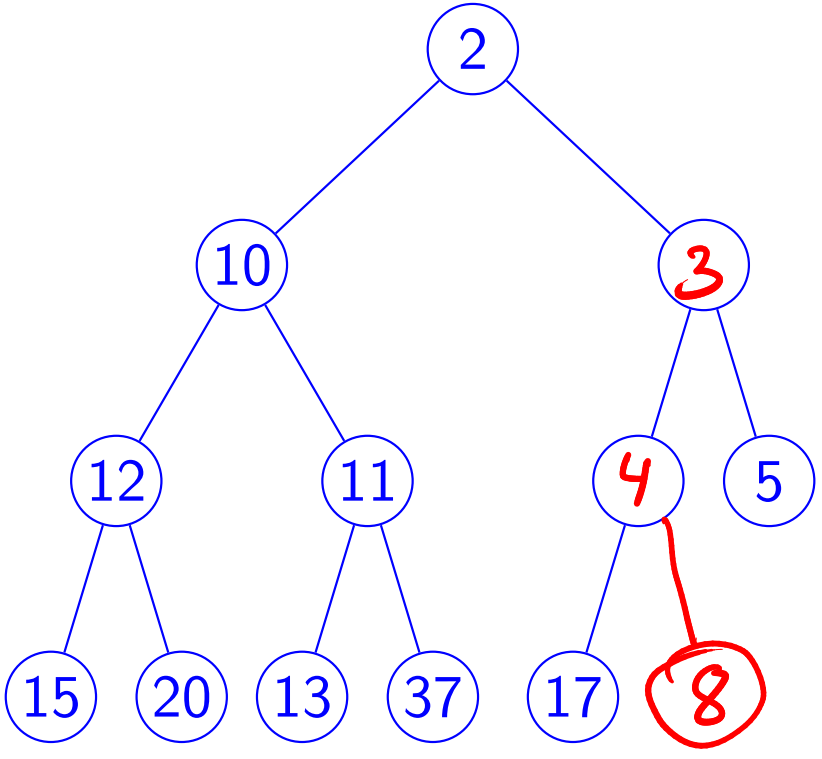
Men 4>3

pop (värdet 2):

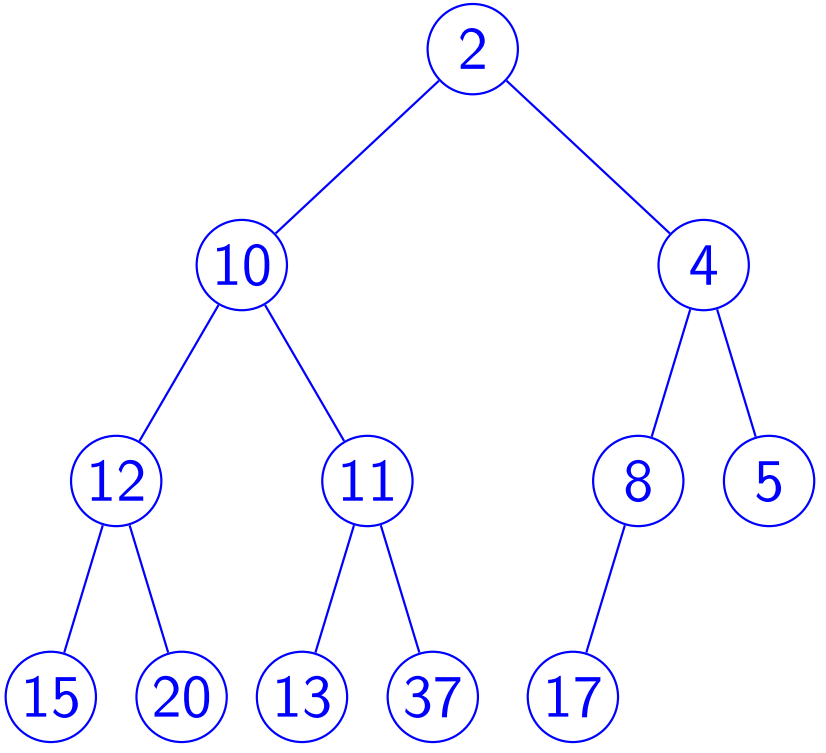


Snabb repetition: prioritetsköer och heaps

insert med värdet 3:



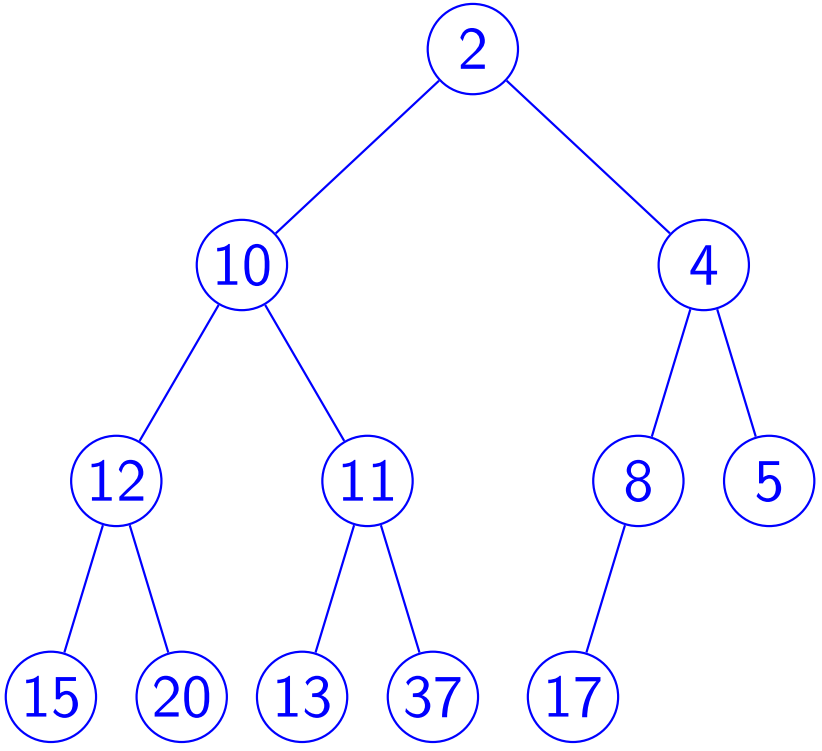
pop (värdet 2):



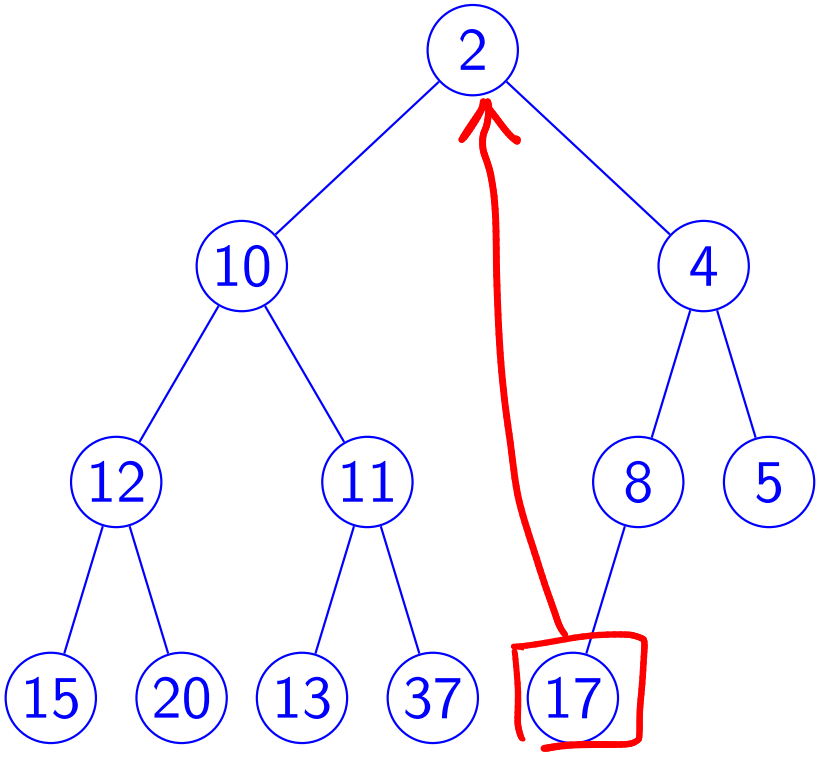
Ok, alla krav uppfyllda! $O(\log n)$ operationer.

Snabb repetition: prioritetsköer och heaps

insert med värdet 3:

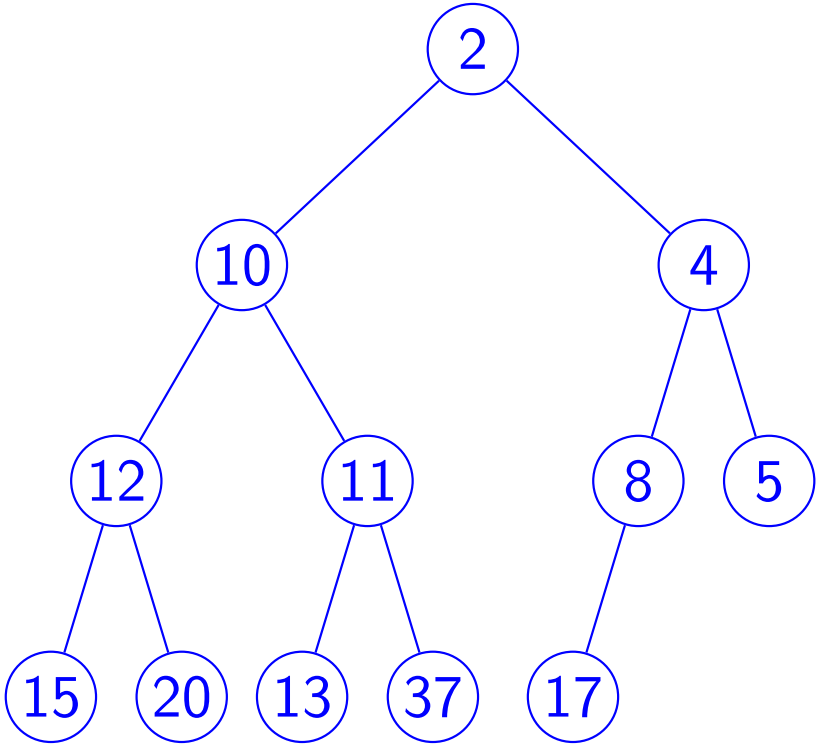


pop (värdet 2):

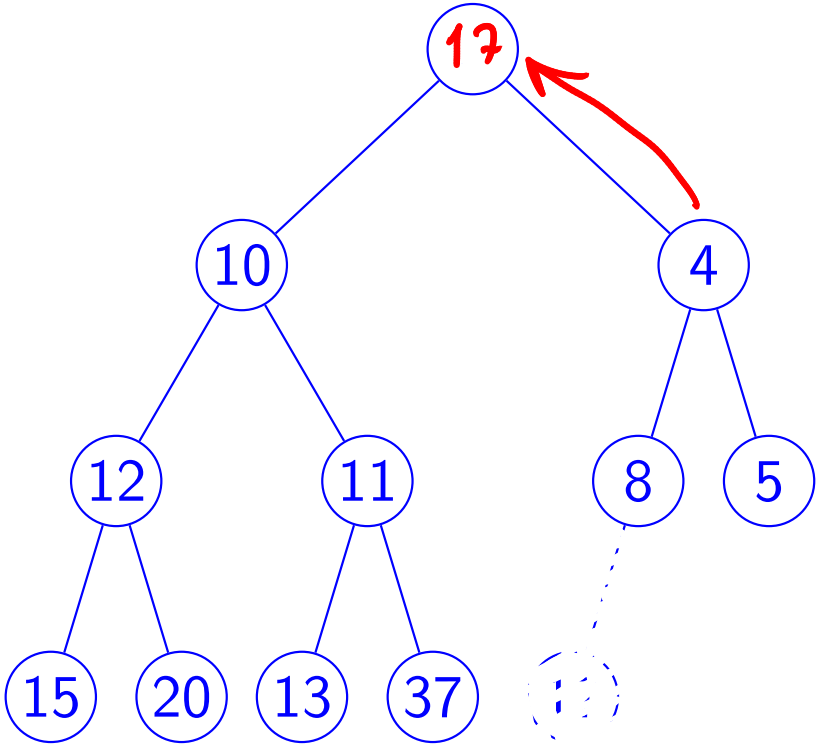


Snabb repetition: prioritetsköer och heaps

insert med värdet 3:



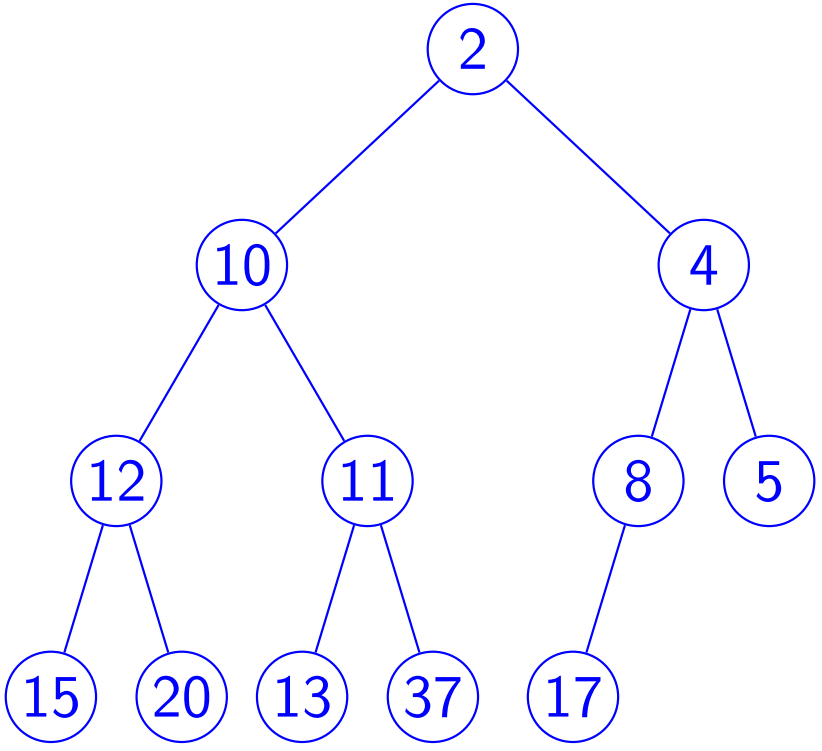
pop (värdet 2):



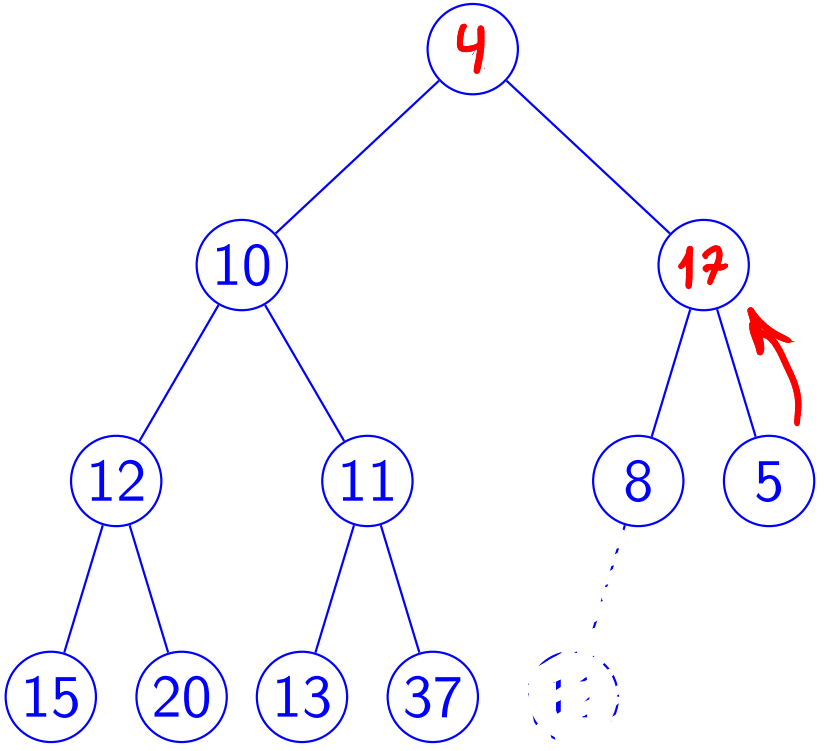
Men 17 > 4 och
17 > 10

Snabb repetition: prioritetsköer och heaps

insert med värdet 3:



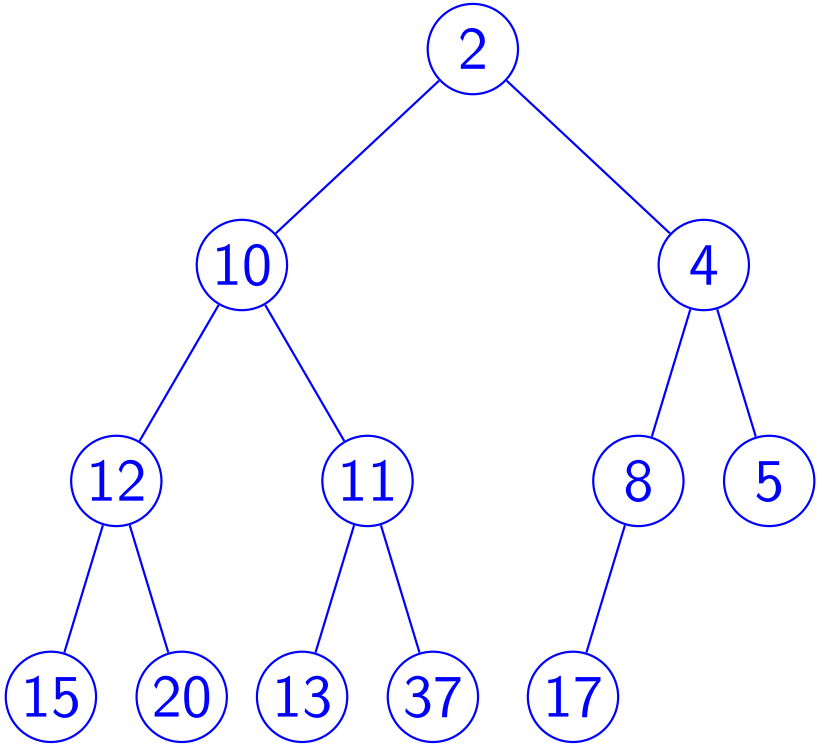
pop (värdet 2):



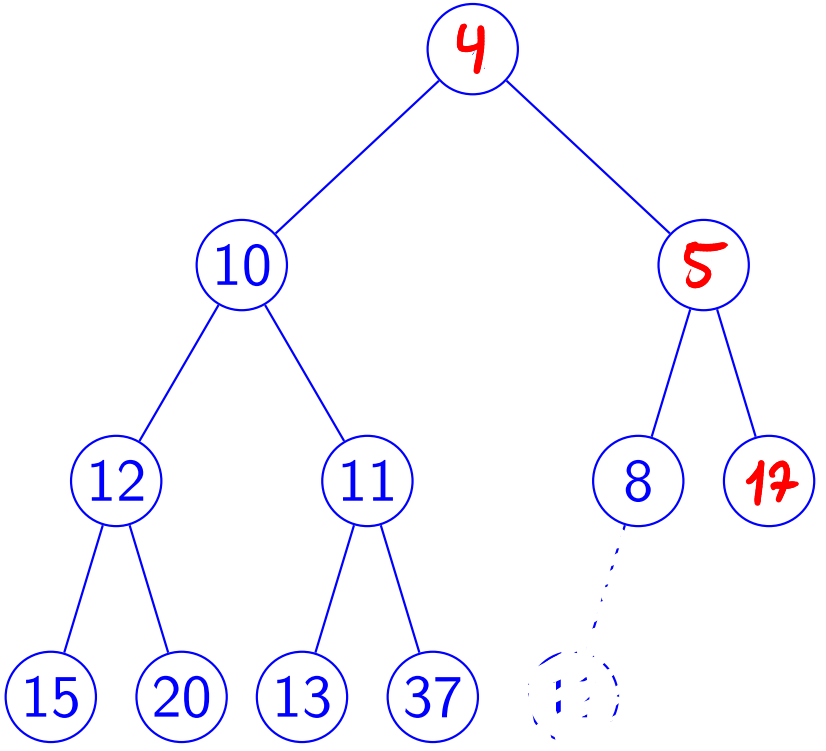
Men $17 > 8$ och $17 > 5$

Snabb repetition: prioritetsköer och heaps

insert med värdet 3:



pop (värdet 2):



Ok, alla krav uppfyllda efter $O(\log n)$ steg!

De många smakerna av heaps

Operation	find-min	delete-min	insert	decrease-key	meld
Binary[17]	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$\Theta(n)$
Leftist	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$	$\Theta(\log n)$
Binomial[17][18]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$[c]	$\Theta(\log n)$	$O(\log n)$[d]
Fibonacci[17][19]	$\Theta(1)$	$O(\log n)$[c]	$\Theta(1)$	$\Theta(1)$[c]	$\Theta(1)$
Pairing[20]	$\Theta(1)$	$O(\log n)$[c]	$\Theta(1)$	$o(\log n)$[c][e]	$\Theta(1)$
Brodal[23][f]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Rank-pairing[25]	$\Theta(1)$	$O(\log n)$[c]	$\Theta(1)$	$\Theta(1)$[c]	$\Theta(1)$
Strict Fibonacci[26]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
2–3 heap[27]	$O(\log n)$	$O(\log n)$[c]	$O(\log n)$[c]	$\Theta(1)$	?

De många smakerna av heaps

Sämst →

Bäst →

Operation	find-min	delete-min	insert	decrease-key	meld
Binary[17]	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$\Theta(n)$
Leftist	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$	$\Theta(\log n)$
Binomial[17][18]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$[c]	$\Theta(\log n)$	$O(\log n)$[d]
Fibonacci[17][19]	$\Theta(1)$	$O(\log n)$[c]	$\Theta(1)$	$\Theta(1)$[c]	$\Theta(1)$
Pairing[20]	$\Theta(1)$	$O(\log n)$[c]	$\Theta(1)$	$o(\log n)$[c][e]	$\Theta(1)$
Brodal[23][f]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Rank-pairing[25]	$\Theta(1)$	$O(\log n)$[c]	$\Theta(1)$	$\Theta(1)$[c]	$\Theta(1)$
Strict Fibonacci[26]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
2–3 heap[27]	$O(\log n)$	$O(\log n)$[c]	$O(\log n)$[c]	$\Theta(1)$	?

De många smakerna av heaps

Sämst →

Bäst →

Operation	find-min	delete-min	insert	decrease-key	meld
Binary[17]	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$\Theta(n)$
Leftist	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$	$\Theta(\log n)$
Binomial[17][18]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$[c]	$\Theta(\log n)$	$O(\log n)$[d]
Fibonacci[17][19]	$\Theta(1)$	$O(\log n)$[c]	$\Theta(1)$	$\Theta(1)$[c]	$\Theta(1)$
Pairing[20]	$\Theta(1)$	$O(\log n)$[c]	$\Theta(1)$	$o(\log n)$[c][e]	$\Theta(1)$
Brodal[23][f]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Rank-pairing[25]	$\Theta(1)$	$O(\log n)$[c]	$\Theta(1)$	$\Theta(1)$[c]	$\Theta(1)$
Strict Fibonacci[26]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
2–3 heap[27]	$O(\log n)$	$O(\log n)$[c]	$O(\log n)$[c]	$\Theta(1)$	?

Så, ingen använder binär-heaps?

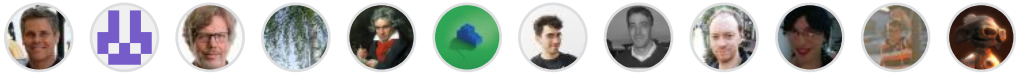
Johodå!

Heaps i bibliotek

```
 72  * ({@code peek}, {@code element}, and {@code size}).
 73  *
 74  * <p>This class is a member of the
 75  * <a href="{@docRoot}/../technotes/guides/collections/index.html">
 76  * Java Collections Framework</a>.
 77  *
 78  * @since 1.5
 79  * @author Josh Bloch, Doug Lea
 80  * @param <E> the type of elements held in this collection
 81  */
 82 public class PriorityQueue<E> extends AbstractQueue<E>
 83     implements java.io.Serializable {
 84
 85     private static final long serialVersionUID = -7720805057305804111L;
 86
 87     private static final int DEFAULT_INITIAL_CAPACITY = 11;
 88
 89     /**
 90      * Priority queue represented as a balanced binary heap: the two
 91      * children of queue[n] are queue[2*n+1] and queue[2*(n+1)].  The
 92      * priority queue is ordered by comparator, or by the elements'
 93      * natural ordering, if comparator is null: For each node n in the
 94      * heap and each descendant d of n, n <= d.  The element with the
 95      * lowest value is in queue[0], assuming the queue is nonempty.
 96      */
 97     transient Object[] queue; // non-private to simplify nested class acce
 98
 99     /**
100      * The number of elements in the priority queue.
101      */
102     private int size = 0;
103
104     /**
```

cpython / Lib / **heapq.py** / <> Jump to

adaggarwal Update: usage doc for heappushpop (GH-91451) ✓

18 contributors  +6

603 lines (533 sloc) | 22.5 KB

```
1   """Heap queue algorithm (a.k.a. priority queue).
2
3   Heaps are arrays for which a[k] <= a[2*k+1] and a[k] <= a[2*k+2] for
4   all k, counting elements from 0.  For the sake of comparison,
5   non-existing elements are considered to be infinite.  The interesting
6   property of a heap is that a[0] is always its smallest element.
7
8   Usage:
9
10  heap = []            # creates an empty heap
11  heappush(heap, item) # pushes a new item on the heap
12  item = heappop(heap) # pops the smallest item from the heap
13  item = heap[0]       # smallest item on the heap without popping it
14  heapify(x)           # transforms list into a heap, in-place, in linear time
```

Varför binär-heap?

Trots skillnader: en heap tar ändå oftast $\mathcal{O}(n \log n)$ av en algoritms totala körtid (n antalet heap-operationer)

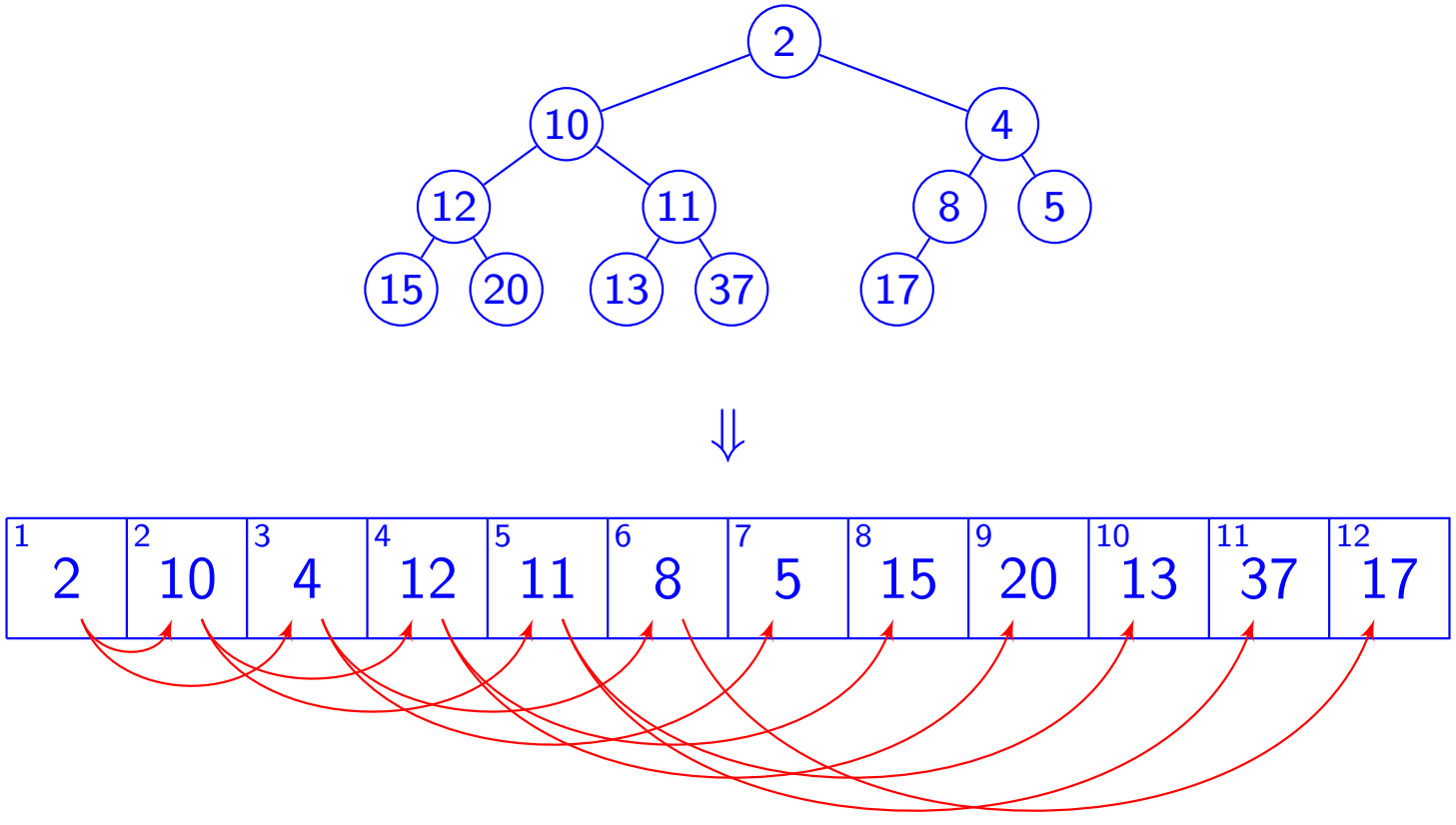
1. Fylla en heap med element man aldrig plockar ut: *extremt sällsynt!*
2. Mer än $\log n$ "melds" (unioner av heaps): *ovanligt, kan ofta fuskas!*
3. Sänka nyckel-värde mer än $\mathcal{O}(1)$ gånger per element (amorterat): *viktigt i vissa fall, men kan ofta fuskas!*

Varför fuska om vi har lösningen i Fibonacci/Brodal heaps?

Operation	find-min	delete-min	insert	decrease-key	meld
Binary[17]	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$\Theta(n)$
Leftist	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$	$\Theta(\log n)$
Binomial[17][18]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$[c]	$\Theta(\log n)$	$O(\log n)$[d]
Fibonacci[17][19]	$\Theta(1)$	$O(\log n)$[c]	$\Theta(1)$	$\Theta(1)$[c]	$\Theta(1)$
Pairing[20]	$\Theta(1)$	$O(\log n)$[c]	$\Theta(1)$	$o(\log n)$[c][e]	$\Theta(1)$
Brodal[23][f]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Rank-pairing[25]	$\Theta(1)$	$O(\log n)$[c]	$\Theta(1)$	$\Theta(1)$[c]	$\Theta(1)$
Strict Fibonacci[26]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
2–3 heap[27]	$O(\log n)$	$O(\log n)$[c]	$O(\log n)$[c]	$\Theta(1)$	?

Binär-heaps i praktiken

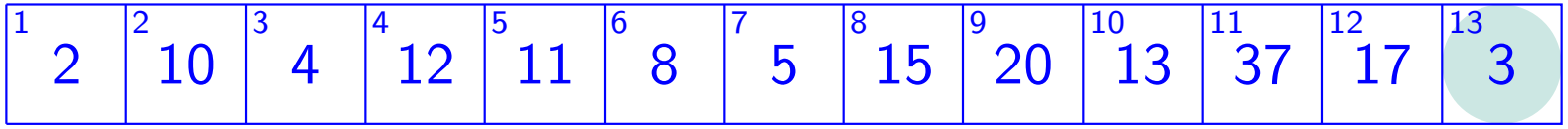
Binär-heaps är enkla att implementera och naturligt pekarfria, vilket gör dem kompakta i minne och cache



Nod is barn finns på plats $2i$ och $2i + 1$, och icke-existerande barn har index utanför sekvensen.

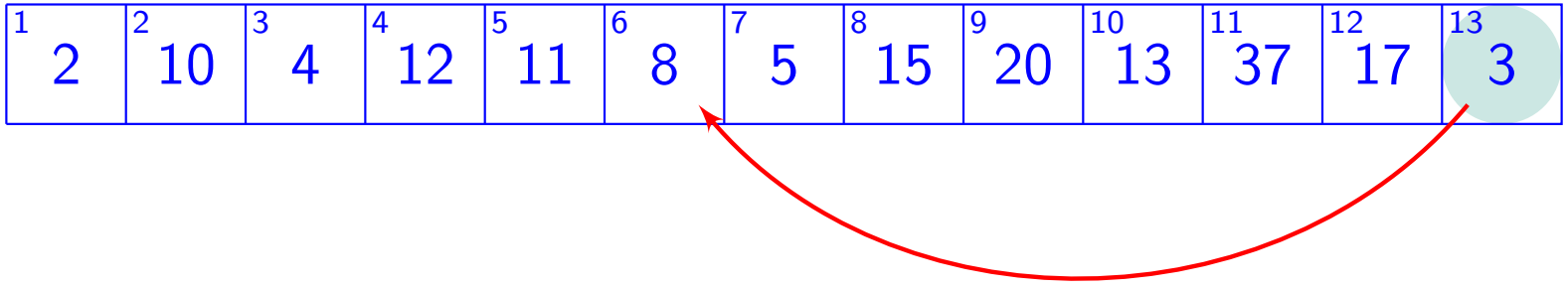
Binär-heaps i praktiken, operationer

Binär-heaps är enkla att implementera och naturligt pekarfria, vilket gör dem kompakta i minne och cache



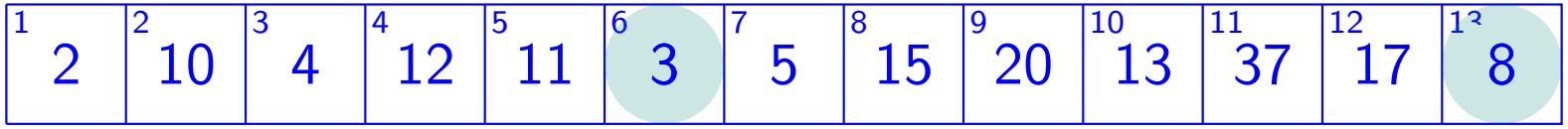
Binär-heaps i praktiken, operationer

Binär-heaps är enkla att implementera och naturligt pekarfria, vilket gör dem kompakta i minne och cache



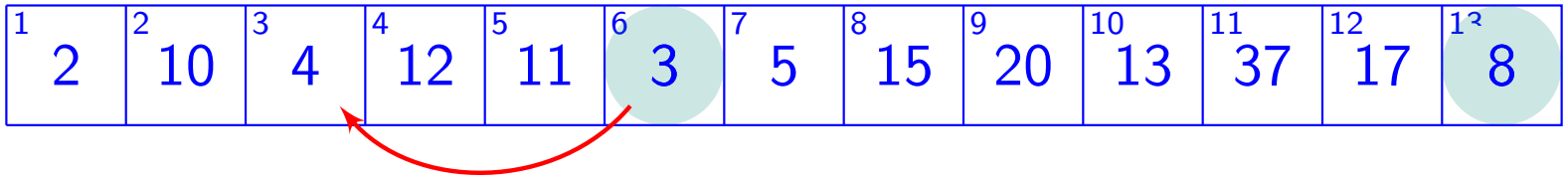
Binär-heaps i praktiken, operationer

Binär-heaps är enkla att implementera och naturligt pekarfria, vilket gör dem kompakta i minne och cache



Binär-heaps i praktiken, operationer

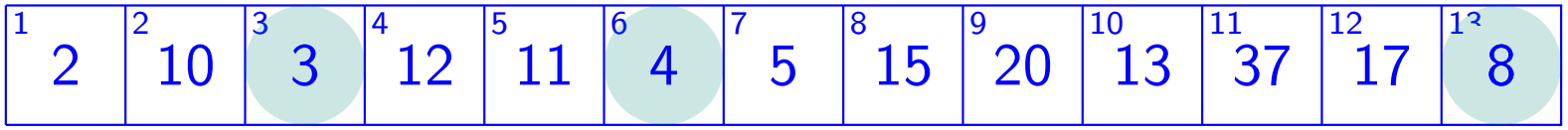
Binär-heaps är enkla att implementera och naturligt pekarfria, vilket gör dem kompakta i minne och cache



Dela index med två (trunkerat) för förälder, igen $4 > 3$

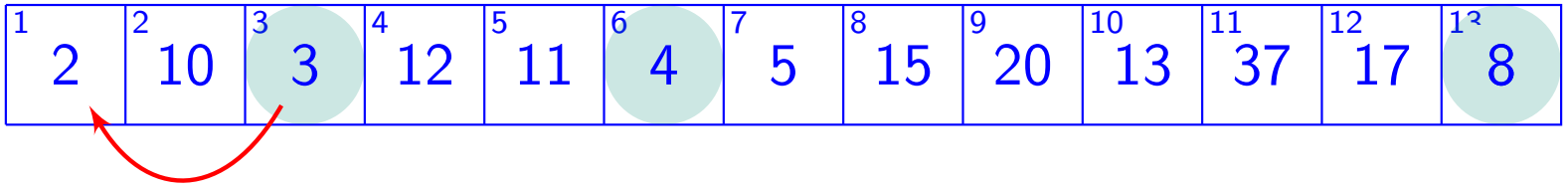
Binär-heaps i praktiken, operationer

Binär-heaps är enkla att implementera och naturligt pekarfria, vilket gör dem kompakta i minne och cache



Binär-heaps i praktiken, operationer

Binär-heaps är enkla att implementera och naturligt pekarfria, vilket gör dem kompakta i minne och cache



Dela index med två (trunkerat) för förälder, nu $2 < 3$, OK!

delete-min liknande övning:

1. Sätt `h[1]=h[length(h)]` (notera 1-indexering).
2. Rotera ner `h[1]` tills heap-invarianten håller.
3. Korta arrayen med 1.

Trivial i vanlig växande array: vi lägger alltid till/tar bort i slutet!

Skiss av Binomial-heaps

På tavla

Var *mycket* försiktig med att anta dem snabbare: *Ofta långsammare!*

Inte så ovanlig situation i att vi har två rätt olika förändringar:

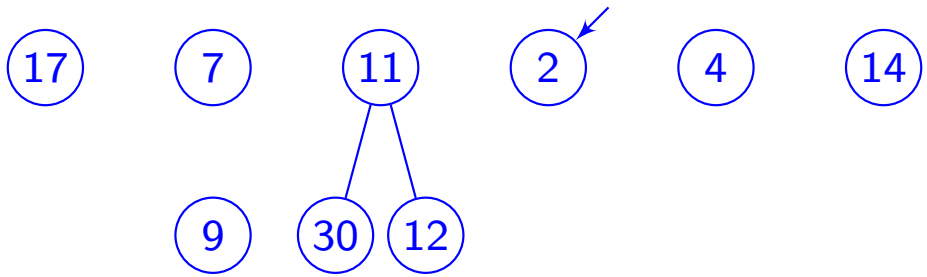
- Om applikationen verkligen behöver många merge/melds: då betalar det sig
  $\Rightarrow$ Klar funktionalitet, går ofta att fuska, men återuppfinner hjulet
- Fyller du ofta heapen med huvudsakligen element som aldrig tas ut?
  - Uppmärksamma att $\mathcal{O}(1)$-argumentet bara fungerar vid en *ren* sekvens av inserts.
  - Annan modifikation landar oss tillbaka i $\mathcal{O}(\log n)$ på nästa insert
  - Alltså: ren senareläggning av arbete, oftast görbart på andra sätt

Verklig slutsats: var medvetna om vad ni gör, vad som finns, och när det behövs: mäta.

Fibonacci-heaps i praktiken

En Fibonacci-heap består av en *ordnad skog:* en sekvens av träd

Varje träd uppfyller heap-invarianten, men har en obegränsad rank och inget inbördes sammanhang. Utöver skogen har datastrukturen en pekare till minsta roten



- *merge:* konkatenera skogarna godtyckligt, uppdatera minimum-pekare till den mindre; $\mathcal{O}(1)$
- *insert:* skapa den triviala Fibonacci-heapen bestående av en ensam nod, anropa *merge* med original-heapen; $\mathcal{O}(1)$
- *decrease-key:* ta bort noden, anropa *insert* med ny; $\mathcal{O}(1)$
- *delete-min:* ta bort minimum-roten, gör alla ($\mathcal{O}(\log n)$) barn till rötter på nya träd i sekvensen, och *optimera skogen*
  - Här slås träd ihop till större träd, nytt minimum identifieras
  - $\mathcal{O}(n)$ i värsta fall, amorterat $\mathcal{O}(\log n)$ (komplicerad analys)

Fibonacci-heaps i praktiken

Antal pekare att underhålla (syskon, föräldrar, barn, trädgrannar)

$\Rightarrow$ dåliga konstanter. Intressant nog ett större problem i *praktiken*, i tävling plågas tänkta implementationer asymptotiskt

Värsta-fall $\mathcal{O}(n)$ på *delete-min* ibland (sällan) problematiskt

Dock mer realistiskt: användbar främst då vi *vet* att mönstret av operationer vinner på Fibonacci

$\Rightarrow$ Dijkstra's med höga kantantal

$\Rightarrow$ Dijkstra's med tidigt avslut

Fibonacci-heaps i praktiken

Antal pekare att underhålla (syskon, föräldrar, barn, trädgrannar)

$\Rightarrow$ dåliga konstanter. Intressant nog ett större problem i *praktiken*, i tävling plågas tänkta implementationer asymptotiskt

Värsta-fall $\mathcal{O}(n)$ på *delete-min* ibland (sällan) problematiskt

Dock mer realistiskt: användbar främst då vi *vet* att mönstret av operationer vinner på Fibonacci

$\Rightarrow$ Dijkstra's med höga kantantal

$\Rightarrow$ Dijkstra's med tidigt avslut

Andra alternativ när binär heap inte räcker till:

- *Binomial heap* enklare men liknar Fibonacci
- *Binär heap* med extra fusk!

Clinic

You work at a clinic. The clinic factors in the waiting time when selecting patients to treat next. This approach was adopted to prevent patients from having to wait too long before being treated. Your task is to help the clinic perform 3 types of queries:

1. Process a patient arrival to the clinic. The patient will have an arrival time T, a name M, and a severity S that is accessed automatically by scanners at the entrance.
2. At time T, the doctor is ready to treat a patient. Every time this happens, the clinic will calculate priority values for every patient waiting in the clinic, and the patient with the highest priority value will be treated first. The priority value is computed as the following sum

   $$S + K \cdot W$$

   where S is the severity value of the patient, K is the constant that the clinic uses, and W is the total time the patient has been waiting in the clinic. If there are multiple patients with that value, the patient with the lexicographically smallest name is treated next. Your program will announce the name of that patient.
3. At time T, the clinic receives a notification stating that, due to unfortunate circumstances, a patient with name M has left the queue permanently. If no patient with name M exists in the queue, it is always a false alarm and will be ignored by the clinic. Otherwise, the notification is guaranteed to be valid and should be processed accordingly.

Input

The first line of the input contains 2 integers, $1 \leq N \leq 200\,000$, the number of queries to be processed, and $0 \leq K \leq 10\,000\,000$, the constant for the clinic. N lines follow, each line beginning with an integer Q where Q = 1, 2 or 3. $Q = 1$ denotes a query of the first type and will be followed by an integer T, a string M and an integer S. $Q = 2$ denotes a query of the second type and will be followed by an integer T. $Q = 3$ denotes a query of the third type and will be followed by an integer T and a string M. For all queries, $0 \leq T, S \leq 10\,000\,000$, and T values are strictly increasing. M is a non-empty alphanumeric string containing no spaces, and contains at most 10 characters. All patients have unique names. There is at least one query of the second type.

Problem ID: clinic

CPU Time limit: 1 second

Memory limit: 1024 MB

Difficulty: 3.5

Download:

Sample data files

Author: Matthew Ng Zhen Rui

Source: NUS Competitive Programming

characters. All patients have unique names. There is at least one query of the second type.

Output

For each query of the second type, output the name of the patient who will be treated on a new line. If the clinic is empty, print the string “doctor takes a break” (without quotes) on a new line instead.

Subtasks

1. **(28 Points)**: $1 \leq N \leq 10,000$. There is no query of type 3. You can assume that $K = 0$.
2. **(32 Points)**: $1 \leq N \leq 10,000$. There is no query of type 3.
3. **(20 Points)**: There is no query of type 3.
4. **(20 Points)**: No additional constraints.

Warning

The I/O files are large. Please use fast I/O methods.

Sample Input 1

```
5 1
1 10 Alice 5
1 15 Bob 15
2 20
2 25
2 30
```

Sample Output 1

```
Bob
Alice
doctor takes a break
```

Sample Input 2

```
5 5
1 10 Alice 5
1 15 Bob 15
2 20
2 25
2 30
```

Sample Output 2

```
Alice
Bob
doctor takes a break
```

Lite boilerplate

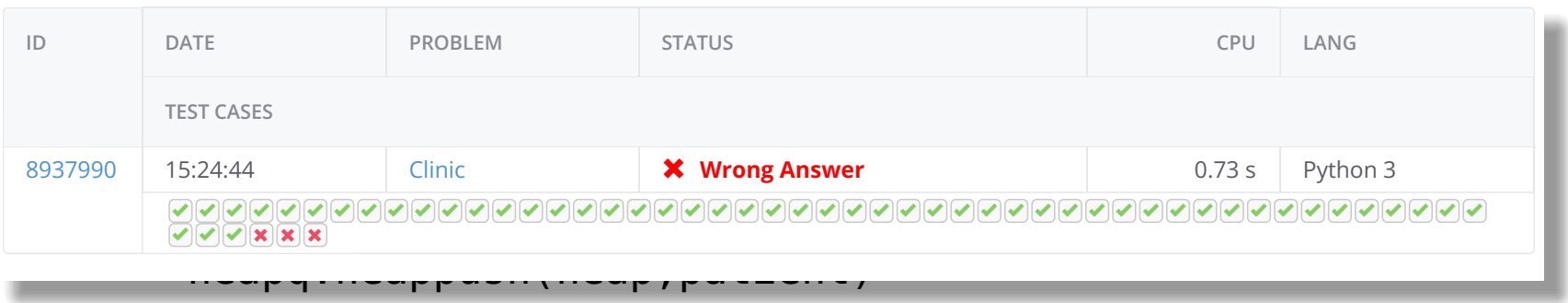
```
import sys
import pdb
import heapq   ⇐ new

if sys.stdin.isatty():
    with open("sample1.in") as f:
        inp=f.readlines()
else:
    inp=sys.stdin.readlines()

def tint(x):
    try:
        return int(x)
    except:
        return x
inp=[[tint(v.strip()) for v in l.split()] for l in inp]
```

Operation 1 och 2

```
k=inp[0][1] # clinic time factor
heap=[]

for instr in inp[1:]:
    if instr[0]==1:  # arrival
        _,time,name,sick=instr
        # Two tricks:
        #  * penalize by arrival instead of updating value
        #    over time, and,
        #  * negate "score" to account for Python min-heap.
        patient=[-(sick-time*k),name]
        heapq.heappush(heap,patient)
    elif instr[0]==2:  # doctor treats
        if len(heap)>0:
            print(heapq.heappop(heap)[1])
        else:
            print('doctor takes a break')
    elif instr[0]==3:  # patient dies
        # This does not exist in Python???
        pass
```

Operation 1 och 2

```
k=inp[0][1] # clinic time factor
heap=[]

for instr in inp[1:]:
    if instr[0]==1:  # arrival
        _,time,name,sick=instr
        # Two tricks:
```



ID	DATE	PROBLEM	STATUS	CPU	LANG
	TEST CASES				
8937990	15:24:44	Clinic	✖ Wrong Answer	0.73 s	Python 3

```
    elif instr[0]==2:  # doctor treats
        if len(heap)>0:
            print(heapq.heappop(heap)[1])
        else:
            print('doctor␣takes␣a␣break')
    elif instr[0]==3:  # patient dies
        # This does not exist in Python???
        pass
```

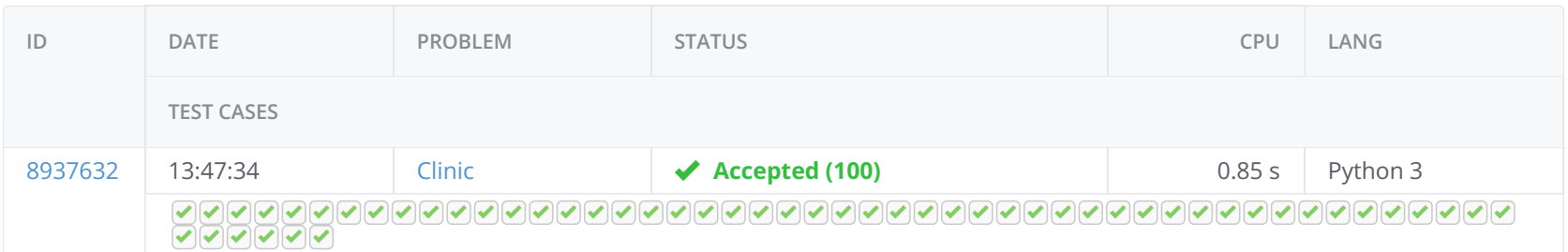
$\Longrightarrow$ Fungerar förstås inte, implementera ny heap?

Operation 3

```
k=inp[0][1] # clinic time factor
heap=[]

patientrefs={}

for instr in inp[1:]:
    if instr[0]==1:  # arrival
        _,time,name,sick=instr
        patient=[-(sick-time*k),name,True]
        patientrefs[name]=patient
        heapq.heappush(heap,patient)
    elif instr[0]==2:  # doctor treats
        while len(heap)>0 and not heap[0][2]:
            heapq.heappop(heap)
        if len(heap)>0:
            print(heapq.heappop(heap)[1])
        else:
            print('doctor takes a break')
    elif instr[0]==3:  # patient dies
        patientrefs[instr[2]][2]=False
```

Operation 3

```
k=inp[0][1] # clinic time factor
heap=[]

patientrefs={}

for instr in inp[1:]:
    if instr[0]==1:  # arrival
```



ID	DATE	PROBLEM	STATUS	CPU	LANG
	TEST CASES				
8937632	13:47:34	Clinic	✔ Accepted (100)	0.85 s	Python 3

```
        while len(heap)>0 and not heap[0][2]:
            heapq.heappop(heap)
        if len(heap)>0:
            print(heapq.heappop(heap)[1])
        else:
            print('doctor␣takes␣a␣break')
    elif instr[0]==3:  # patient dies
        patientrefs[instr[2]][2]=False
```

$\Longrightarrow$ Inga problem!

Fortsatt fusk med binär-heaps

Bra generella komplexiteter svårt, men inspiration från Fibonacci:

```
def decrease_key(heapplus,key,dec):
    # Get the element
    entry=heapplus.entry_map[key]
    # Make a copy with the change, insert that
    new_entry=copy(entry)
    new_entry.value-=dec
    heapq.heappush(heapplus.heap,new_entry)
    # Mark the original deleted (gets dropped on pop)
    entry.deleted=True
    heapplus.number_deleted+=1
    # If a "significant" fraction of the heap contents
    # is deleted we rebuild it
    if len(heapplus.heap)//10<heapplus.number_deleted:
        cleaned=[e for e in heapplus.heap if not e.deleted]
        heapplus.heap=heapq.heapify(cleaned)
        heapplus.number_deleted=0
```

Fortsatt fusk med binär-heaps

Bra generella komplexiteter svårt, men inspiration från Fibonacci:

```
def merge(heapplus1,heapplus2):
    # Put them together in a list, with the heap
    # with most elements put first

def pop(heapplus):
    # Find the heap in the list with the smallest
    # root, pop normally. Then also transfer some
    # number of elements from any of the non-first
    # heaps into the first.
```

Alehouse

Public Domain, Tavern Scene by David Teniers the Younger, via Wikimedia Commons

Making friends can seem impossible, but going to the alehouse makes it easy — it is actually the only way to make friendships. Luckily, the alehouse is extremely good at its task: if two people are inside simultaneously, they instantly become friends. People even become friends if they meet each other in the door as one leaves and one enters the alehouse!

In Consistentville, each of its n residents goes to the alehouse exactly once each week, and always during the same milliseconds as the week before. This is convenient for everyone, since then nobody needs to befriend new people all the time, which can be quite exhausting.

You are contemplating a move to Consistentville in order to adopt their well-ordered lifestyle, and have decided that you want as many friends as possible. However, you don't actually enjoy ale that much, so you decide to limit your weekly visit at the alehouse to at most k milliseconds. What is the maximum number of friends you can get?

Input

The first line of input contains two positive integers n ($1 \leq n \leq 100\,000$), and k ($0 \leq k < 604\,800\,000$). The next n lines describe at which millisecond each of the original residents of Consistentville enters and leaves the alehouse every week. Specifically, the i^{th} line consists of two integers a_i and b_i ($0 \leq a_i \leq b_i < 604\,800\,000$) indicating that the i^{th} resident enters the alehouse at millisecond a_i and leaves the alehouse at millisecond b_i each week.

Output

A single integer, the maximum number of friends you can get.

Sample Input 1

```
6 2
```

Sample Output 1

```
4
```

Submit | Stats

My Submissions

Problem ID: alehouse

CPU Time limit: 2 seconds

Memory limit: 1024 MB

Difficulty: 5.1

Download:

Sample data files

Author: Torstein Strømme

Source: Bergen Open 2019

The first line of input contains two positive integers n ($1 \leq n \leq 100\,000$), and k ($0 \leq k < 604\,800\,000$). The next n lines describe at which millisecond each of the original residents of Consistentville enters and leaves the alehouse every week. Specifically, the i^{th} line consists of two integers a_i and b_i ($0 \leq a_i \leq b_i < 604\,800\,000$) indicating that the i^{th} resident enters the alehouse at millisecond a_i and leaves the alehouse at millisecond b_i each week.

Output

A single integer, the maximum number of friends you can get.

Sample Input 1

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

Sample Output 1

```
4
```

Lösning

Egentligen inte svår lösning, men ändå 5.1 svårighetsgrad?

- *Misstag:* för många datastrukturer? Besvärlig aritmetik?
- Klassisk problemtyp för loggning/diagnostik/monitorering, t.ex.: *transaktioner senaste timmen*

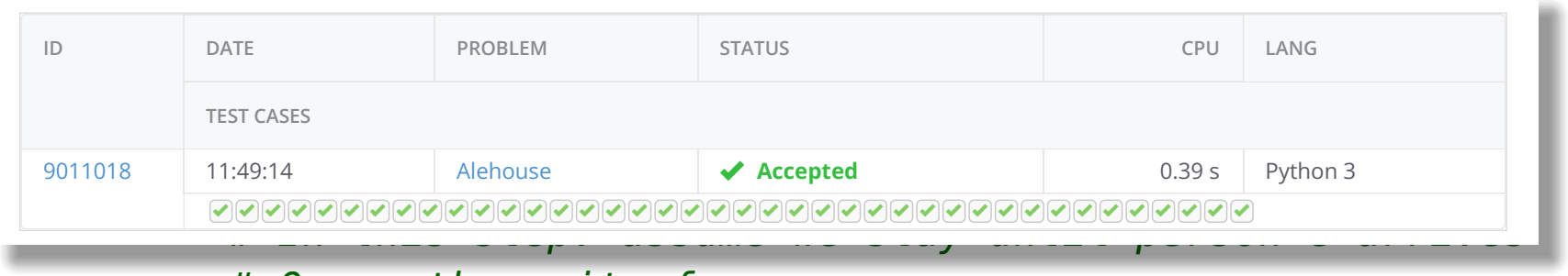
```
k=inp[0][1]
max_friends=-1
entries=sorted(inp[1:],key=lambda x: x[0])
heap=[]
for e in entries:
    # In this step: assume we stay until person e arrives
    # Queue the exit of person e
    heapq.heappush(heap,e[1])
    # So we leave e[0], does someone leave before we arrive?
    if e[0]-heap[0]>k:
        heapq.heappop(heap)
    # Then compute what friends we make in this timespan
    max_friends=max(max_friends,len(heap))
print(f'{max_friends}')
```

Lösning

Egentligen inte svår lösning, men ändå 5.1 svårighetsgrad?

- *Misstag:* för många datastrukturer? Besvärlig aritmetik?
- Klassisk problemtyp för loggning/diagnostik/monitorering, t.ex.: *transaktioner senaste timmen*

```
k=inp[0][1]
```



ID	DATE	PROBLEM	STATUS	CPU	LANG
	TEST CASES				
9011018	11:49:14	Alehouse	✔ Accepted	0.39 s	Python 3

```
    # Queue the exit of person e
    heapq.heappush(heap,e[1])
    # So we leave e[0], does someone leave before we arrive?
    if e[0]-heap[0]>k:
        heapq.heappop(heap)
    # Then compute what friends we make in this timespan
    max_friends=max(max_friends,len(heap))
print(f'{max_friends}')
```

Lösning: Hoppsan

Egentligen inte svår lösning, men ändå 5.1 svårighetsgrad?

- *Misstag:* för många datastrukturer? Besvärlig aritmetik?
- Klassisk problemtyp för loggning/diagnostik/monitorering, t.ex.: *transaktioner senaste timmen*

```
k=inp[0][1]
max_friends=-1
entries=sorted(inp[1:],key=lambda x: x[0])
heap=[]
for e in entries:
    # In this step: assume we stay until person e arrives
    # Queue the exit of person e
    heapq.heappush(heap,e[1])
    # So we leave e[0], does someone leave before we arrive?
    if e[0]-heap[0]>k:
        heapq.heappop(heap)
    # Then compute what friends we make in this timespan
    max_friends=max(max_friends,len(heap))
print(f'{max_friends}')
```

Lösning

Egentligen inte svår lösning, men ändå 5.1 svårighetsgrad?

- *Misstag:* för många datastrukturer? Besvärlig aritmetik?
- Klassisk problemtyp för loggning/diagnostik/monitorering, t.ex.: *transaktioner senaste timmen*

```
k=inp[0][1]
max_friends=-1
entries=sorted(inp[1:],key=lambda x: x[0])
heap=[]
for e in entries:
    # In this step: assume we stay until person e arrives
    # Queue the exit of person e
    heapq.heappush(heap,e[1])
    # Perform all exits which would happen before we arrive
    while e[0]-heap[0]>k:
        heapq.heappop(heap)
    # Then compute what friends we make in this timespan
    max_friends=max(max_friends,len(heap))
print(f'{max_friends}')
```

Ett ämne till: binär sökning

Givet en sorterad sekvens $v_1, \ldots, v_n$, vilken position skulle ett värde v få om det lades till?

Alternativt: finns v i denna sorterade lista?

Klassiskt, förekommer ofta! Binär sökning, $\mathcal{O}(\log n)$ steg. Tänk leken, jag tänker på ett tal mellan 1 och 100, ni skall gissa:

- 50 (mitt mellan 1 och 100)! Nej, *lägre.*
- 25 (mitt mellan 1 och 49)! Nej, *högre.*
- 37 (mitt mellan 26 och 49)! Nej, *högre.*
- 43! *... etc.*

Påminner då den dyker upp i övningar, men också: var försiktiga med indexering, en enkel algoritm med mycket möjlighet till off-by-ones!

Se också: *ternär sökning,* och allmänt *sökning sökning sökning!*

Låt oss lösa ett problem

Firefly

A Japanese firefly has flown into a cave full of obstacles: stalagmites (rising from the floor) and stalactites (hanging from the ceiling). The cave is N units long (where N is even) and H units high. The first obstacle is a stalagmite after which stalactites and stalagmites alternate.

Here is an example cave 14 units long and 5 units high (the image corresponds to the second example):

MEMORY LIMIT
1024 MB

CPU TIME LIMIT
1 second

2.8 DIFFICULTY
Medium

LINKS
Statistics

DOWNLOADS
Sample data files
Problem PDF

SOURCE & LICENSE
Croatian Regional Competition in Informatics 2007
For educational use only

This is not the best choice because the firefly will end up less tired if it chooses either level one or five, as that would require destroying only seven obstacles.

You are given the width and length of the cave and the sizes of all obstacles.

Write a program that determines the **minimum number** of obstacles the firefly needs to destroy to reach the end of the cave, and on how many distinct levels it can achieve that number.

Input

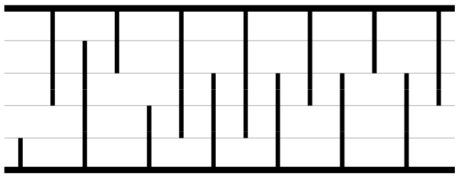
The first line contains two integers N and H, $2 \leq N \leq 200\,000$, $2 \leq H \leq 500\,000$, the length and height of the cave. N will always be even.

The next N lines contain the sizes of the obstacles, one per line. All sizes are positive integers less than H.

Firefly

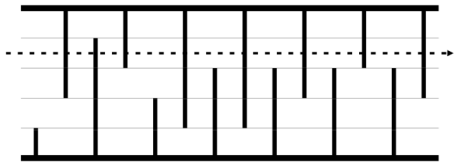
A Japanese firefly has flown into a cave full of obstacles: stalagmites (rising from the floor) and stalactites (hanging from the ceiling). The cave is N units long (where N is even) and H units high. The first obstacle is a stalagmite after which stalactites and stalagmites alternate.

Here is an example cave 14 units long and 5 units high (the image corresponds to the second example):



The Japanese firefly is not the type that would fly around the obstacle, instead it will choose a single height and ram from one end of the cave to the other, **destroying all obstacles** in its path with its mighty kung-fu moves.

In the previous example, choosing the 4:th level from the ground has the firefly destroying eight obstacles:



This is not the best choice because the firefly will end up less tired if it chooses either level one or five, as that would require destroying only seven obstacles.

You are given the width and length of the cave and the sizes of all obstacles.

Write a program that determines the **minimum number** of obstacles the firefly needs to destroy to reach the end of the cave, and on how many distinct levels it can achieve that number.

Input

The first line contains two integers N and H, $2 \leq N \leq 200\,000$, $2 \leq H \leq 500\,000$, the length and height of the cave. N will always be even.

The next N lines contain the sizes of the obstacles, one per line. All sizes are positive integers less than H.

Överslag

Idé: vi simulerar alla höjder (testa att flyga på höjd 0.5, sedan 1.5, sedan 2.5, osv., kom ihåg hur många kung-fu-moves som behövs)

Överslag

Idé: vi simulerar alla höjder (testa att flyga på höjd 0.5, sedan 1.5, sedan 2.5, osv., kom ihåg hur många kung-fu-moves som behövs)

Nja. 500 000 höjder, 200 000 hinder. Multiplicerar till 10^{11}, kommer inte vara en sekund hur vi än kodar.

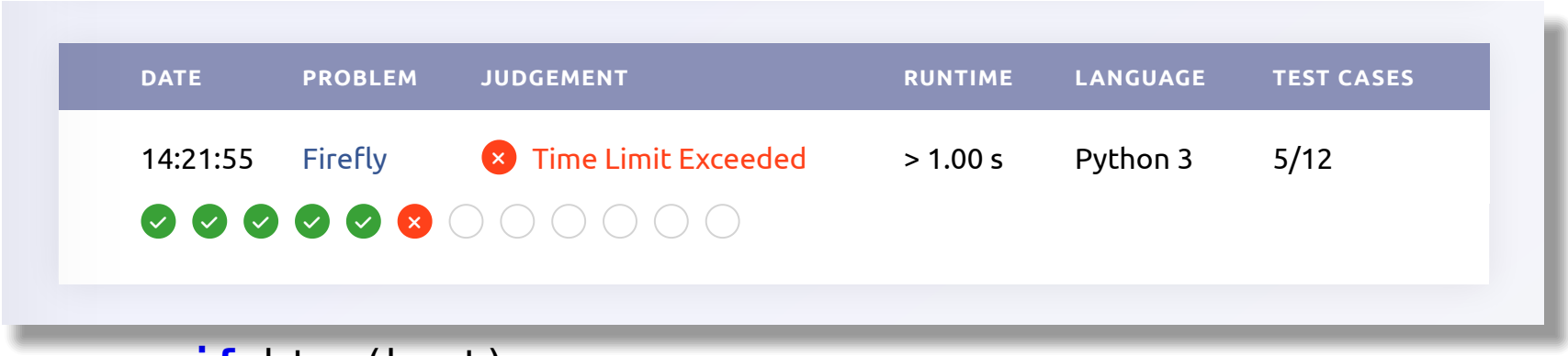
Men låt oss koda ut det ändå, ibland bra att testa en *korrekt* lösning och sedan fundera hur man optimerar.

Långsam men korrekt

```
n,h=inp[0]
stalagmites=[v[0] for v in inp[1::2]]
stalactites=[v[0] for v in inp[2::2]]

# simulate height ht
def sim(ht):
    kungfu=0
    for st in stalagmites:
        if ht<=st:
            kungfu+=1  # pow!
    # stalactites hang down, so need to subtract from h
    for st in stalactites:
        if ht>=(h-st):
            kungfu+=1  # kabom!
    return kungfu

# simulate all heights and then do stats on it
sims=[sim(ht+.5) for ht in range(0,h)]
min_kungfu=min(sims)
n_min=sum(v==min_kungfu for v in sims)
print(min_kungfu,n_min)
```

Långsam men korrekt

```
n,h=inp[0]
stalagmites=[v[0] for v in inp[1::2]]
stalactites=[v[0] for v in inp[2::2]]

# simulate height ht
def sim(ht):
```



DATE	PROBLEM	JUDGEMENT	RUNTIME	LANGUAGE	TEST CASES
14:21:55	Firefly	Time Limit Exceeded	> 1.00 s	Python 3	5/12

```
        if ht>=(h-st):
            kungfu+=1  # kabom!
    return kungfu

# simulate all heights and then do stats on it
sims=[sim(ht+.5) for ht in range(0,h)]
min_kungfu=min(sims)
n_min=sum(v==min_kungfu for v in sims)
print(min_kungfu,n_min)
```

Men hur optimerar vi?

Insikten: ordningen på stalagmiter/stalaktiter spelar ingen roll

Men hur optimerar vi?

Insikten: ordningen på stalagmiter/stalaktiter spelar ingen roll

1. Sortera hindren (var för sig)
2. När vi *börjar* träffa hinder träffar vi resten
3. Så binärsök efter var vi börjar träffa (två gånger, en för stalagmiter, en för stalaktiter)

Simulera alla upp till $500\,000$ höjderna, men nu $2 * \log_2 200\,000$ tittar på hinder, just över 35. Säg totalt $18\,000\,000$ steg på en sekund. Låter *lätt!*

Binärsökning

Jag finner det lättast att resonera genom att minnas vad som *inte* längre är möjligt: så *lo* och *hi* utgör första *omöjliga* position.

```
# compute index at which v should be inserted into
# the sorted list ls if we are to retain sorted'ness
def bins(ls,v):
    lo=-1
    hi=len(ls)
    while lo+1<hi:
        mid=lo+(hi-lo)//2
        if v>ls[mid]:
            lo=mid
        elif v<ls[mid]:
            hi=mid
        else:
            return mid
    # hi if we imagine we push the existing elements
    # to the right, which is usually the interpretation
    return hi
```

Kompletta lösningen

```
n,h=inp[0]
stalagmites=sorted([v[0] for v in inp[1::2]])
stalactites=sorted([v[0] for v in inp[2::2]])

def bins(ls,v):
    Förra sliden...

# simulate height ht
def sim(ht):
    # find where we start hitting them with bins
    return ((n//2)-bins(stalagmites,ht)) + \
    markstart ((n//2)-bins(stalactites,h-ht))

sims=[sim(ht+.5) for ht in range(0,h)]
min_kungfu=min(sims)
n_min=sum(v==min_kungfu for v in sims)
print(min_kungfu,n_min)
```

Kompletta lösningen

```
n,h=inp[0]
stalagmites=sorted([v[0] for v in inp[1::2]])
stalactites=sorted([v[0] for v in inp[2::2]])

def bins(ls,v):
    Förra sliden...
```

DATE	PROBLEM	JUDGEMENT	RUNTIME	LANGUAGE	TEST CASES
15:07:58	Firefly	Accepted	0.52 s	Python 3	12/12


```
    markstart ((n//2)-bins(stalactites,h-ht))

sims=[sim(ht+.5) for ht in range(0,h)]
min_kungfu=min(sims)
n_min=sum(v==min_kungfu for v in sims)
print(min_kungfu,n_min)
```