Innlevering 2

Høst 2024

Effektive mengder

Den abstrakte datatypen for mengder kalles `Set`. Hvis `set` er av typen `Set`, så forventer vi at følgende operasjoner støttes:

`contains(set,` x`)`	er x med i mengden?
`insert(set,` x`)`	setter x inn i mengden (uten duplikater)
`remove(set,` x`)`	fjerner x fra mengden
`size(set)`	gir antall elementer i mengden

Husk at hverken rekkefølge eller antall forekomster noen rolle i mengder. Ved fjerning av et element som ikke er i mengden skal mengden forbli uforandret.

Implementasjon

Mengden skal implementeres som et *binært søketre*. Det betyr at du må sørge for at `contains`, `insert` og `remove` er i $\mathcal{O}(\log(n))$ så lenge vi antar at treet er balansert. Operasjonen `size` bør være i $\mathcal{O}(1)$.

Input

Input skal leses fra `stdin`.

Første linje av input består av et heltall N, der $1 \leq N \leq 10^6$, som angir hvor mange operasjoner som skal gjøres på megnden.

Hver av de neste N linjene er på følgende format

`contains` x	der x er et heltall $1 \leq N \leq 10^9$
`insert` x	der x er et heltall $1 \leq N \leq 10^9$
`remove` x	der x er et heltall $1 \leq N \leq 10^9$
`size`	

Merk at du ikke trenger å ta høyde for ugyldig input på noen som helst måte.

Output

Output skal skrives til `stdout`.

For hver linje av input som er på formen:

`contains` x

skal programmet skrive ut `true` dersom x er med i mengden, og `false` ellers.

For hver linje av input som er på formen:

`size`

skal programmet skrive ut antall elementer som er i mengden.

Eksempel input/output:

Eksempel-input	Eksempel-output

`9`
`insert 1`
`insert 2`
`insert 3`
`insert 1`
`contains 1`
`contains 0`
`remove 1`
`contains 1`
`size`
	true
	false
	false
	2

Det er publisert flere input- og outputfiler på semestersiden.

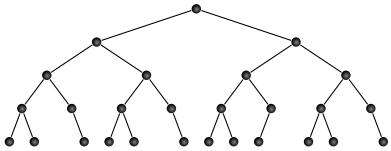
Oppgaver

(a) Skriv et Java eller Python-program som leser input fra `stdin` og skriver ut output *nøyaktig* slik som beskrevet ovenfor.

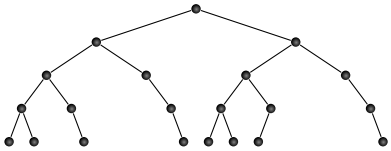
(b) Skriv et Java eller Python-program som er helt identisk, bortsett fra at det binære søketreet er erstattet med et AVL-tre.

Bygge balanserte søketrær

I denne oppgaven ønsker vi å bygge et *helt balansert binært søketre*. Vi definerer dette som et binært søketre hvor det er 2^d noder med dybde d, der $0 \leq d < h$ og h er høyden på treet[1]. En annen måte å si det samme på er at den korteste og den lengste stien fra roten til et tomt subtre (for eksempel representert med en `null`-peker) har en lengdeforskjell på 0 eller 1.



(a) Et helt balansert binærtre



(b) Et ikke helt balansert binærtre

Du trenger ikke implementere et binært søketre. Alt du trenger å gjøre er å printe ut elementene du får som input i en rekkefølge som garanterer at vi får et balansert tre dersom vi legger elementene inn i binærtreet ved bruk av vanlig innsetting. Dette binære søketreet er *ikke selvbalanserende*. Input består av heltall i sortert rekkeføge, der ingen tall forekommer to ganger (altså trenger du ikke ta høyde for duplikater).

Eksempel-input	Eksempel-output
0	5
1	8
2	10
3	9
4	7
5	6
6	2
7	4
8	3
9	1
10	0

(a) Du har fått et *sortert array* med heltall som input. Lag en algoritme som skriver ut elementene i en rekkefølge, slik at hvis de blir plassert i et binært søketre i den rekkefølgen så resulterer dette i et *balansert* søketre.

- Skriv pseudokode for algoritmen du kommer frem til.
- Skriv et Java eller Python-program som implementerer algoritmen din. Det skal lese tallene fra `stdin` og skrive dem ut som beskrevet ovenfor.

(b) Nå skal du løse det samme problemet kun ved bruk av *heap*. Altså: Algoritmen din kan ikke bruke andre datastrukturer enn heap, men til gjengjeld kan du bruke så mange heaper du vil!

- Skriv pseudokode for algoritmen du kommer frem til. Her kan du anta at elementene allerede er plassert på en heap, og at input kun består av en heap med heltall.
- Skriv et Java eller Python-program som implementerer algoritmen din. Programmet må først plassere elementene som leses inn på en heap, og derretter kalle på implementasjonen av algoritmen du har kommet frem til.

  For Java kan du bruke `PriorityQueue`[2]. De eneste operasjonene du trenger å bruke fra Java sin `PriorityQueue` er: `size()`, `offer()` og `poll()`. Merk at `offer()` svarer til `push()`, og `poll()` svarer til `pop()`.

  For Python kan du bruke `heapq`[3]. De eneste operasjonene du trenger er: `heappush()` og `heappop()`, samt kalle `len()` for å få størrelsen på heapen.

[1] Merk at dette er veldig likt definisjonen av et *komplett* binærtre, som forklares i forelesningen om heaps, men uten kravet om at noder med dybde h er plassert så langt til venstre som mulig.

[2] https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html

[3] https://docs.python.org/3/library/heapq.html