05. Rotfaste trestrukturer

Forelesning 5

Trær gjenspeiler rekursiv dekomponering. I binære søketrær er alt i venstre deltre mindre enn rota, mens alt i høyre er større, og det gjelder rekursivt! Hauger er enklere: Alt er mindre enn rota. Det begrenser funksjonaliteten, men gjør dem billigere å bygge og balansere.

Pensum

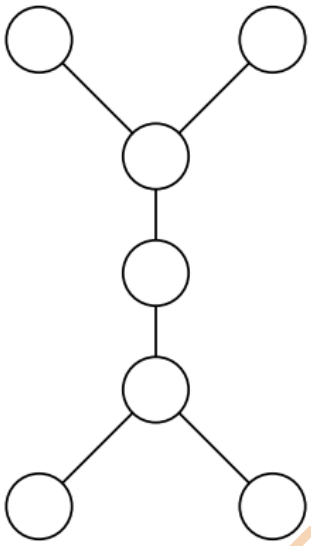
- [ ] Innledningen til del V
- [ ] Kap. 6. Heapsort
- [ ] Kap. 10. Elementary data structures: 10.3
- [ ] Kap. 12. Binary search trees

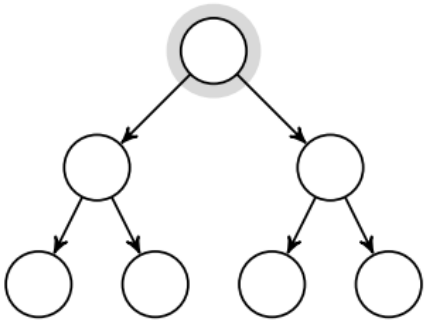
Læringsmål

- [E$_1$] Forstå hvordan *heaps* fungerer, og hvordan de kan brukes som *prioritetskøer*
- [E$_2$] Forstå HEAPSORT
- [E$_3$] Forstå hvordan *rotfaste trær* kan implementeres
- [E$_4$] Forstå hvordan *binære søketrær* fungerer
- [E$_5$] Vite at forventet høyde for et tilfeldig binært søketre er $\Theta(\lg n)$
- [E$_6$] Vite at det finnes søketrær med garantert høyde på $\Theta(\lg n)$

📚Trær

Trær er en betegnelse for grafer uten sykluser hvor antall kanter er antall noder minus 1, $|E| = |V| - 1$. Et binærtre er et tre hvor hver node har 0-2 barn. Et komplett binærtre er et binærtre hvor alle løvnodene ligger på samme høyde med null barn. Et komplett binærtre har alltid 2^x løvnoder når høyden er x. Derav ser vi også at høyden av et tre med n noder blir $\lg n$.





Det finnes flere klasser trær:

- Et ordnet tre definerer en ordning på barna
- Et posisjonstre har hvert barn en posisjon, og barn kan mangle.
- Et binærtre er et posisjonstre hvor hvert barn har to barneposisjoner.

CLRS definerer ikke binærtrær som trær, men vi kan godt tolke dem som ordnede trær med ekstra informasjon. Dessuten kan det være hensiktsmessing å se på binærtrær som grafer. I noen tilfeller kan kantene i binærtreet ha piler. Det er dermed også en directed-acyclic-graph (DAG).

Når vi skal traversere treet har vi et par muligheter. I hvilken rekkefølge skal vi håndtere den nåværende noden før vi fortsetter ned treet? Vi har tre muligheter:

- Inorder: besøk alle nodene til venstre, deretter seg selv, og til slutt de på høyre. Dette blir riktig rekkefølge for binære søketrær.
- Preorder: besøk seg selv, deretter noder til venstre og høyre
- Postorder: besøk noder til venstre og høyre, deretter seg selv

Det er kun $InorderTreeWalk$ som er implementert i pensum. Det er dermed åpenbart at kompleksiteten til traversering av et tre med n noder er $\Theta(n)$.

algdat/InorderWalk.scala at main · matsjla/algdat

This file contains bidirectional Unicode text that may be interpreted or compiled differently than what appears below. To review, open the file in an editor that reveals hidden Unicode characters. Learn

https://github.com/matsjla/algdat/blob/main/src/main/scala/com/supergrecko/dsa/tree/InorderWalk.scala

matsjla/**algdat**

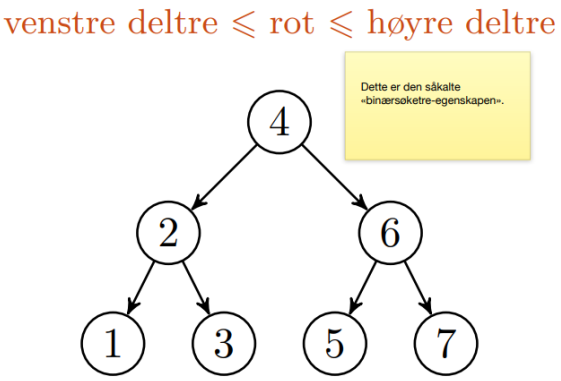
Implementation of all algoritms and data structures from TDT4120 at NTNU in Scala

1 Contributor · 0 Issues · 0 Stars · 0 Forks

📚Binære søktrær

Binære søketrær er funksjonelt sett binærsøk i form av en datastruktur. Barnet til venstre har lavere verdi, og det til høyre har en høyere verdi. På denne måten blir det veldig lett å navigere treet.

Det er også greit å merke seg at binære søketrær i pensum ikke håndterer duplikater.

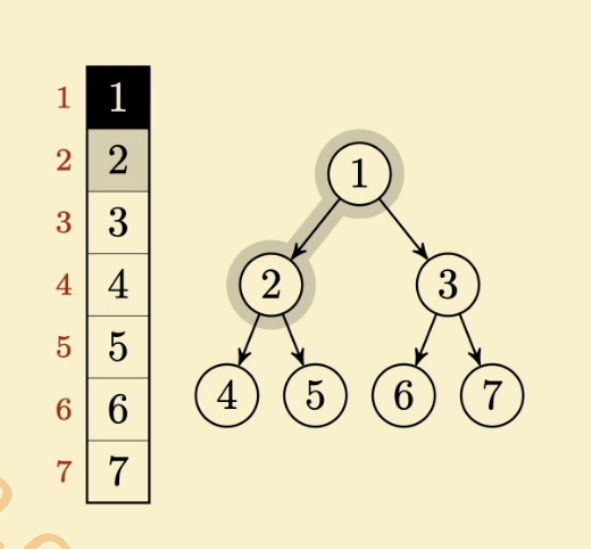


Relevant funksjonalitet på et BinarySearchTree fra pensum er:

- $TreeSearch$ - søk for en verdi i treet på kjøretid $O(\lg n)$, eller $O(1)$ hvis rot er tom eller søkeverdi.
- $TreeInsert$ - sett inn et nytt tall i treet i $O(\lg n)$ tid, eller $O(1)$ hvis treet er tomt.
- $TreeMinimum$ - finn minimumverdien treet i $O(\lg n)$ tid, eller $O(1)$ hvis treet er tomt.
- $TreeMaximum$ - finn maksverdien i treet i $O(\lg n)$ tid, eller $O(1)$ hvis treet er tomt.
- $TreeSuccessor$ - finn noden med minste verdi som er større en den gitte noden i $O(\lg n)$ tid.
- $TreeDelete$ - slett et element fra treet i $O(\lg n)$ tid.

📚Hauger

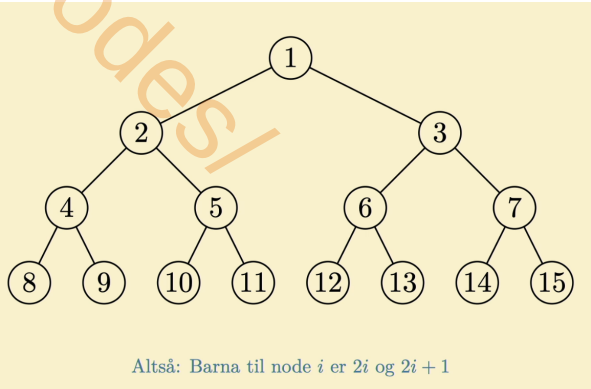
En haug er en datastruktur. I bunn og grunn er en haug en liste eller et array, men vi forestiller oss at haugen er et tre.



Med denne tolkningen kan vi få tilgang til barna eller foreldrene til en node. Hauger er automatisk så balanserte som mulig.

Vi ser ofte på $MaxHeap$ eller $MinHeap$ versjoner av hauger. Dette gir oss en gylden mulighet til å implementere prioritetskøer.

Hauger trenger ikke å være sorterte, og siden det finnes eksponensielt mange lovlige rekkefølger, gjelder ikke sorteringsgrensen.



Relevant funksjonalitet på en Heap fra pensum er:

- $HeapLeft$: finn posisjonen barnet til venstre til en node i konstant tid $O(1)$.
- $HeapRight$: finn posisjonen barnet til høyre til en node i konstant tid $O(1)$.
- $HeapParent$: finn posisjonen foreldrenoden til en node i konstant tid $O(1)$.

- $MaxHeapify$: omplasserer en gitt node og barna til noden slik at den følger max-heap egenskapen i $O(\lg n)$ tid, men konstant $O(1)$ i beste tilfelle.
- $BuildMaxHeap$: bygg et MaxHeap av et array i lineær tid $O(n)$.

I tilfellet hvor haugen brukes som en prioritetskø ønsker vi oss følgende funksjonalitet (som også er pensum):

- $HeapMax$: finn den største verdien i haugen i konstant tid $O(1)$.
- $HeapExtractMax$: finn og fjern det største elementet i haugen i logaritmisk tid $O(\lg n)$.
- $HeapIncreaseKey$: øk verdien til en nøkkel i haugen i logaritmisk tid $O(\lg n)$.
- $MaxHeapInsert$: sett inn en ny verdi inn i haugen i logaritmisk tid $O(\lg n)$.

💻HeapSort

Til slutt kan vi bruke disse funksjonene til å implementere en sorteringsalgoritme ved hjelp av en haug. Denne algoritmen heter $HeapSort$.

- Best case: $O(n)$
- Worst case: $O(n \lg n)$
- Average case: $O(n \lg n)$

Algoritme	WC	AC/E	BC
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heap sort	$\Theta(n \lg n)$	—	$\Theta(n)$
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)^*$	$\Theta(n \lg n)$
Counting sort	$\Theta(n + k)$	$\Theta(n + k)$	$\Theta(n + k)$
Radix sort	$\Theta(d(n + k))$	$\Theta(d(n + k))$	$\Theta(d(n + k))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)^\dagger$	$\Theta(n)$

*Expected, RANDOMIZED-QUICKSORT

†Average-case

Kan få alle i én bøtte, og de sorteres med insertion sort