Lab 3

Analys av SortedLinkedListSet

Nedan följer komplexitetsanalys av contains, add och remove i SortedLinkedListSet. Efter detta följer ett komplexitetsdiagram och kommentarer om denna. Härdanefter kommer n ange antalet noder/element i set:et.

Contains

Vi har först två if-satser och lite annat som har konstant komplexitet. Efter detta har vi en loop som plockar fram mitten elementet i listan, alltså som går från början av listan till det mittersta elementet. Om listan innehåller n element så kommer vi gå igenom hälften av dessa element, alltså n/2 element. Efter detta kollar vi i vilken halva av listan vi ska leta i. Oavsett vilken halva så måste vi gå igenom n/2 element och undersöka deras compareTo värden som är konstant.
Vi får alltså en komplexitet på O(n). Då varje n/2 komplexitet följer efter varandra och inte i varandra.

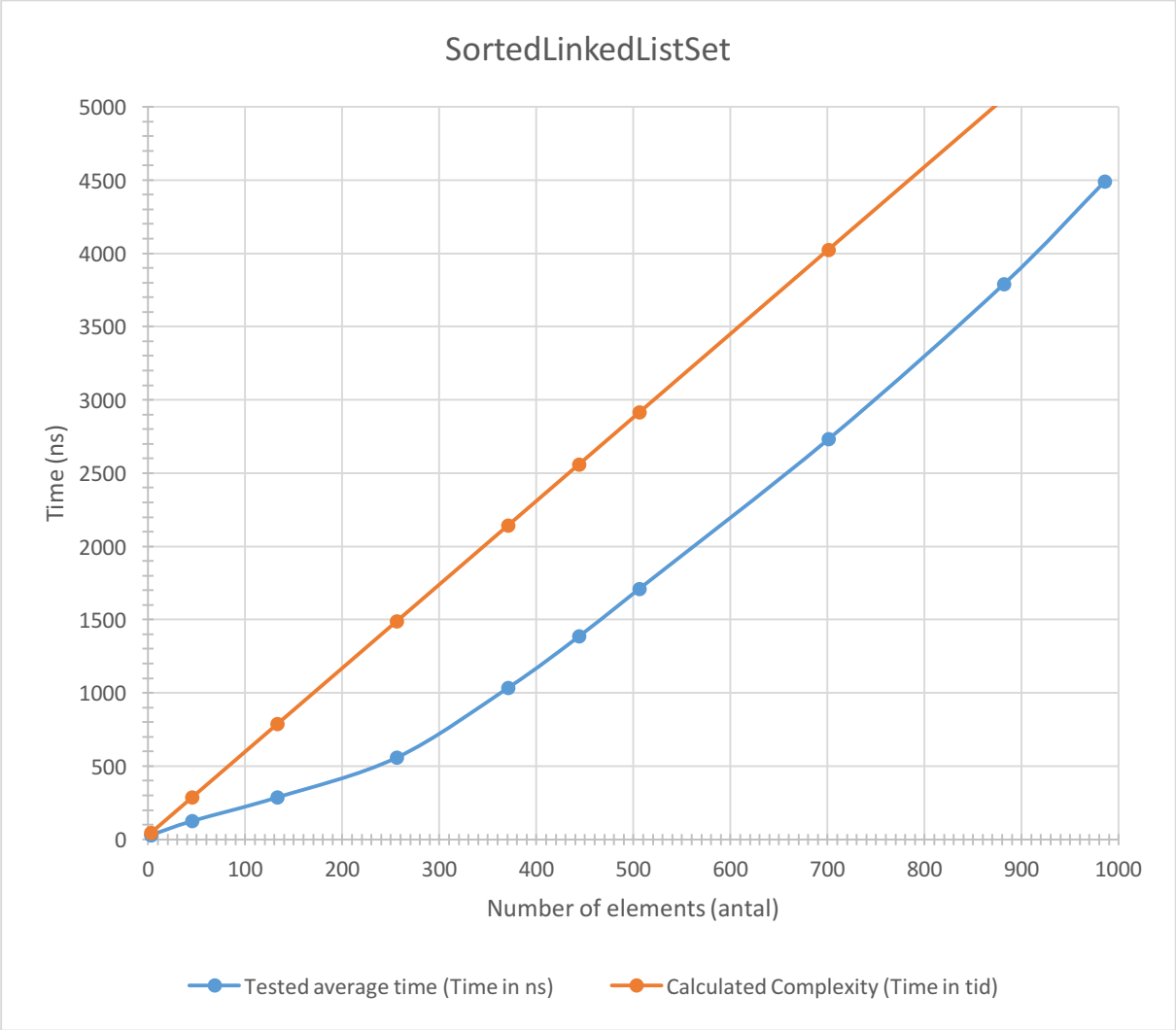
Add

Vi börjar med att fånga upp ett specialfall på konstant komplexitet. Sedan anropar vi på contains och där har vi en komplexitet på n. Vi kollar sedan om elementet ska placeras först eller sist (placeringen där sker på konstant komplexitet). Annars har vi en for-loop med komplexiteten (n-2)*(2) +6, vilket ändå har n i komplexitet. Det som ska läggas till är att för varje varv i den sista for-loopen genomför vi compareTo med konstant komplexitet. Vi genomför även compareTo när vi avgör om vi ska placera det nya elementet först eller sist. Vi får också här O(n) i komplexitet.

Remove

Vi har tre specialfall, ett ifall listan är tom, ifall det är första elementet och ett ifall det är sista elementet som vi ska ta bort. Alla dessa sker med konstant komplexitet. Om den vi ska ta bort inte är någon utav dessa tidigare fall så gör vi en for-loop där vi letar upp det elementet vi ska ta bort. Denna for-loop har komplexiteten (n-2)*2 + 3. Varje iteration av loopen kallar också på compareTo.
Sammanställt får vi komplexiteten O(n) här också.

Diagram



För den beräknade komplexiteten använde vi oss av formeln 5,7n+30. Anledningen till att den blåa linjen avviker i början (från en linjär graf, alltså jämfört med senare värden för den blåa) är för att den ger ett genomsnittligt värde för alla funktioner och varierar i vilka operationer den utför på vad och med vilka tal. Avvikelsen skulle lika gärna kunna vara längre upp på kurvan som i början. Den beräknade komplexiteten är en beräkning av värsta fallet och alla operationer man utför på set:et är inte värstafalls operationer. Sammanfattningsvis så bildar de uppmätta värdena en förhållandevis linjär graf som överensstämmer med den beräknade komplexiteten.

Analys av SplayTreeSet

Nedan följer komplexitetsanalys av contains, add och remove i SplayTreeSet. Efter detta följer ett komplexitetsdiagram och kommentarer om denna. Härdanefter kommer n ange antalet noder/element i set:et.

Contains

Det första som görs är en if-sats som filtrerar bort ett specialfall med konstant och väldigt låg komplexitet. Nästa steg är en while-loop där vi letar efter elementet genom att gå ner i trädet.

Om trädet är rakt har vi en komplexitet på O(n), men om det är någorlunda jämt fördelat kommer det ha en komplexitet på O(log n). Alltså ju rakare träd vi har (t.ex. alla noder är högerbarn till varandra och roten) desto mer linjär komplexitet får vi. Detta gäller inte bara contains utan även add och remove. Det blir O(n) då för att det finns bara en väg genom trädet och oavsett till vilken nod vi ska till så måste vi gå samma väg.
Anledningen till att komplexiteten blir O(log n) är för att vi halverar antalet möjliga noder varje iteration. Första iterationen har vi n olika noder att gå till. Iteration två har vi istället 1+T(n/2) noder då vi redan har kollat på en nod. Tredje iterationen har vi istället 2+T(n/4) osv. Detta kan vi skriva om till en generell formel: $k + T(\frac{n}{2^k})$, vilket är O(log n). Alltså har contains komplexiteten O(log n). Detta gäller också för Add och Remove.

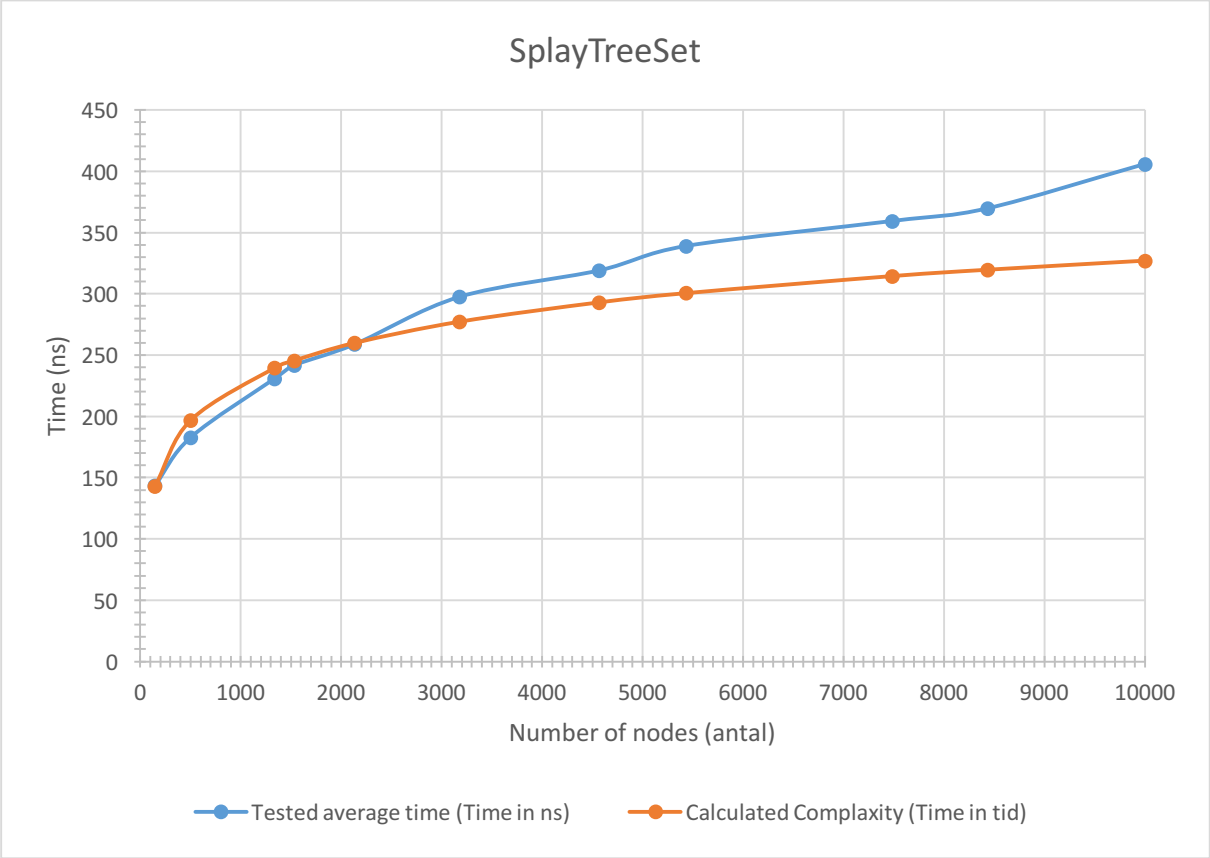
Add

Vi börjar också här med ett specialfall med konstant komplexitet. Efter det påbörjar vi en while-loop som fortsätter till dess att vi har hittat en plats åt den nya noden, under förutsättningen att noden inte redan finns i trädet. Denna har komplexiteten O(log n) enligt samma argument som för contains.

Remove

Vi börjar i remove med att ta hand om 2 specialfall med konstant komplexitet. Sedan genomför vi splay och som tidigare har splay log n i komplexitet. Efteråt kontrollerar vi 3 specialfall till som också dem genomförs med konstant komplexitet. Nu till huvudfallet: Vi börjar med att leta efter största värdet på ena delträdet med rotens vänstra barn som rot. Detta görs i en while-loop med komplexiteten O(log n) enligt samma argument som för Contains. Sedan sker ett antal saker med konstant komplexitet innan vi igen kommer till en while-loop. Denna while-loop går igenom ett delträd för att hitta det minsta elementet i trädet. Detta sker också med O(log n) enligt samma argument som för contains.
Sammantaget får vi att remove tillhör O(log n).

Diagram



För den orangea grafen gäller funktionen 100log(n)-73.
De stämmer ganska så bra överens med varandra. Det som gör att den blåa inte är exakt logaritmisk är ju för att det är i en testmiljö, med många faktorer som påverkar, bland annat vilka operationer som görs, vilka värden som används med mera. Den orangea kurvan är däremot perfekt vilket vi aldrig kommer kunna uppnå i praktiken och representerar värsta fallet.