Tema NR 9: Avancerade datastrukturer

Emma Persson `empe5691`
Elise Edette `tero0337`

6 mars 2015

1 Förslag på muntafrågor

1. Förklara vad ett splayträd är, ge exempel på vad det kan användas till. Beskriv vilka operationer som kan utföras på det, och hur operationerna fungerar.

2. Vad är en paring heap, och hur fungerar den? Ge exempel på vad en pairing-heap kan användas till.

3. Vad är skillnaden mellan en treap och en heap? Vid vilka tillfällen är det bättre att använda en treap än en heap?

2 Röd-svarta träd

Ett Röd-svart träd är en typ av binärt sökträd. Trädet är uppbyggt av noder som innehåller den data som ska lagras, en nodfärg (antingen svart eller röd) samt två referenser till nästkommande noder(barn). Trädet utgår alltid ifrån en svart rotnod, och nya noder som stoppas in i trädet hamnar till vänster om roten, om dess värde är mindre än rotnoden och till höger om dess värde är större. Noder som är röda måste ha svarta noder som barn, och varje gren i trädet, från roten till den yttersta noden, måste ha lika många svarta noder. Dessutom måste varje röd nod ha två svarta barn-noder. Detta måste tas hänsyn till då nya noder ska sättas in i trädet, och då en insättning görs så att regeln inte stämmer måste det antingen ske en omfärgning av noder eller i värsta fall en rotation i trädet.

Detta sätt att placera noder gör att trädet blir relativt balanserat av sig självt. Ett balanserat träd, är ett träd där alla grenar är ungefär lika långa, vilket i de flesta fall är önskvärt då man konstruerar träd. Ett träd som inte är balanserat, exempelvis ett binärt sökträd, kan ha en eller flera grenar som är längre än andra, vilket gör att söktiden blir längre eftersom strukturen börjar likna en länkad lista. Söktiden för ett Röd-svart träd är O(log n), liksom tiden för insättning och borttag.

Det finns flera andra typer av träd som fungerar på ett liknande sätt. AVL-träd är den typ av träd som mest liknar rödsvarta träd. Fördelen med ett röd-svart träd framför ett AVL-träd är att antalet rotationer minimeras, och omfärgning används i en hög utsträckning. Ett AVL-träd liknar ett röd-svart träd, men med undantaget att det inte har färger på noderna.

Istället finns en regel som säger att skillnaden på grenarnas längder ej får vara större än ett. Då en insättning bryter mot denna regel behöver rotationer göras, som strukturerar om trädet så att regeln stämmer. Dessa rotationer är mer kostsamma än omfärgningar, men processen ger å andra sidan alltid ett balanserat träd. Ett Röd-svart träd har inte krav på sig att vara lika balanserat som ett AVL-träd, vilket teoretiskt sett skulle kunna göra att söktiden i trädet blir något längre, medan insättning och borttag blir enklare och mindre kostsamt, på grund av färre rotationer.

Ett röd-svart träd är lämpligt att använda då det finns specifika krav på den tid det ska ta att söka, ta bort eller lägga till element, eftersom trädet har en konstant tidskomplexitet på O(log n) för alla dessa operationer. Andra fördelar med röd-svarta träd, och binära sökträd i allmänhet, är att sökning går snabbare än i andra datastrukturer, såsom Linked-list och ArrayList. Bra användningsområden kan därför vara tillfällen då sökningar utförs frekvent.

De operationer som går att utföra på Röd-svarta träd är insättning, borttag samt sökning. Insättning fungerar så att Den nya nodens plats i trädet hittas genom att från roten jämföra om värdet på den nya noden är större eller mindre än den aktuella. Är värdet större hoppar man ett steg nedåt-åt höger och om det är mindre hoppar man nedåt-åt vänster. Detta upprepas till man kommit till trädets löv, och alltså hittat en nod som pekar på null. Den nya noden färgas därefter i den färg som behövs för att upprätthålla reglerna i trädet. Om föräldern är svart, blir den nya noden röd, men om föräldern är röd, skulle en insättning av en svart nod göra att det blir fler svarta noder i den aktuella

grenen än i de övriga. I detta fall behöver en korrigering göras, antingen genom att färga om ovanstående noder så att reglerna stämmer, eller i värsta fall göra en rotation, så att de nödvändiga noderna byter plats.

Borttag av noder i ett röd-svart träd sker genom att noden till en början hittas genom att en sökning utförs. Sökningen börjar från rotnoden, och utförs genom att se om den nod som ska tas bort har ett värde som är större eller mindre än rotnoden. Är värdet större flyttar man ner till rotnodens högra barn, och om det är mindre flyttar man ner till rotnodens vänstra barn. Detta upprepas tills den nod som ska tas bort är hittad. Då plockas noden bort, och dess barn blir barn till den tidigare föräldranoden. Efter detta kontrolleras om reglerna fortfarande stämmer, och om inte, färgas de kvarvarande noderna antingen om, eller om det behövs, så utförs rotationer.

3 Treaps

Treap är en form av binärt sökträd. Den använder sig av ett prioriteringsfält för noderna och trädet sorteras som en heap i fallande ordning; roten har lägsta prioritet och löven, som består av sentineller med null data, har en prioritet på infinity. Prioriteringen bestäms slumpmässigt.

Precis som ett vanligt binärt sökträd har noderna i en treap två barn, left och right. Vid insättning av en nod placeras den först som ett löv på den plats i trädet där värdet på dess data gör att den passar, precis som i ett binärt sökträd. Sedan roteras den upp beroende på dess prioritetsvärde, alltså den flyttas upp

i trädet tills den hamnar i rätt prioritetsordning, precis som i en heap.

När man vill ta bort en nod sätts dess prioritetsvärde till infinity, sedan roteras den ner så att den blir en löv och tas bort.

Sökning i en treap sker på samma sätt som i ett binärt sökträd, och prioriteringarna används inte. Sökningen börjar från rotnoden, och letar sig neråt i trädet, åt vänster om värdet på den eftersökta noden är lägre och åt höger om värdet är högre tills det eftersökta värdet är hittat.

Fördelar med att använda sig av treaps framför binära sökträd är att tiden för insättning, sökning och borttag är snabbare än tiden för dessa funktioner i ett binärt sökträd i värsta fall. Detta då det är möjligt, om än osannolikt, att noderna i ett binärt sökträd hamnar efter varandra i samma gren. Söktiden i detta träd blir då inte längre logaritmisk, utan närmar sig O(n). I en treap är sannolikheten för att detta ska hända mindre, eftersom de randomiserade prioriteringarna gör att noderna sprider ut sig i trädet, vilket gör att den förväntade tiden för insättning, borttag och sökning är O(log n).

Fördelen med en treap framför en heap är att det inte finns någon möjlighet att söka i en heap på ett effektivt sätt, eftersom det där inte finns någon ordning mellan en nods barn. I en treap, är barnen till vänster mindre än föräldern medan barnet till höger är större.