Algoritmer och datastrukturer 2

Av: Charley Ziegler, Bruno Teglas & Jonathan Gustafsson

ArrayMap

Designval: ArrayMap definieras av en tom lista som ska hantera tuples innehållande en nyckel och ett värde (’a value’). Alla funktioner söker igenom datastrukturen med linjärsökning, vi valde att använda linjärsök för att se hur tiden ökade linjärt jämfört med input.

Put: Inga specifika designval förutom att vi valde att sortera nycklarna i storleksordning (störst först).

Get: Inga specifika designval.

Delete: Inga specifika designval.

Length: Inga specifika designval.

Contains: Inga specifika designval.

Resultat: Resultatet blev som väntat, tidsåtgången ökade linjärt vilket gjorde att när antalet nycklar blev större så ökade tidsåtgången markant. Se även övrigt nedan.

HashMap

Designval: HashMap definieras av en hash funktion, den valda storleken som HashMap ska ha, samt en lista av listor som ska representera HashMap och dess celler. HashMap är en öppen hashning med statisk storlek. Vi valde att tilldela storleken på hash tabellen till 1/10 av den totala mängd element som vi hashar in i HashMap, detta val gjorde vi för att få en tydligare bild på hur funktionernas tidsåtgång påverkades när kollisioner inträffade i HashMap,

Put: Inga specifika designval.

Get: Inga specifika designval.

Delete: Inga specifika designval.

Length: Inga specifika designval.

Contains: Inga specifika designval.

HashFunc: Hash funktionen beräknar nyckeln % storleken på hashtabellen.

Resultat: Resultatet blev som väntat, när alla listor innehöll minst ett element och kollisioner inträffade mer frekvent så ökade tidsåtgången för samtliga funktioner.

BSTMap

Designval: Våra ursprungliga designval bestod av en class men efter problem vid rotering runt rooten så valde vi att låta BSTMap vara en yttre class bestående av flera noder som representerar ett nyckel-värde par.

Put: Inga specifika designval.

Get: Utnyttjar en rekursiv funktion _get som söker igenom BSTMap och returnerar node klassen till get som plockar ut värdet för nyckeln om den existerade och returnerar.

Delete: Inga specifika designval.

Length: Hålls av en variabel av den yttre klassen BSTMap, underlättar om denna kontrolleras frekvent.

Contains: Inga specifika designval. Likt Get utnyttjar contains _get och returnerar True respektive False beroende på om nyckeln fanns.

Resultat: Resultatet hade varit trevligare om vi inte stött på recursion depth error, men som väntat blir skillnaden mellan slumpade nycklar och sorterade stor när trädet inte är balanserat. Se även övrigt nedan.

BalancedBSTMap (AVL-Träd)

Designval: Likt BST så fick vi problem utan en yttre class, alltså består BalancedBSTMap också av en yttre class för att hålla koll på rooten till trädet. För uppdatering av balansen i trädet efter rotering valde vi att istället utnyttja en rekursiv funktion height vilket drar ner prestandan något i Put och Delete.

Put: Inga specifika designval.

Get: Utnyttjar en rekursiv funktion _get som söker igenom BalancedBSTMap och returnerar node klassen till get som plockar ut värdet för nyckeln om den existerade och returnerar.

Delete: Inga specifika designval.

Length: Hålls av en variabel av den yttre klassen BalancedBSTMap, underlättar om denna kontrolleras frekvent.

Contains: Inga specifika designval. Likt Get utnyttjar contains _get och returnerar True respektive False beroende på om nyckeln fanns.

Resultat: Resultatet blev som väntat, för sorterade nycklar blir det fler rotationer för både Put och Delete. Och som sagt uppdateringen av balansen är inte optimal och påverkar Put och Delete i exikveringstid.

Övrigt

Designval: Vi valde att plocka ut värdena i samma ordning som vi satte in dem, vilket som väntat visar sig tydligt i ArrayMap och BSTMap, men även i BalancedBSTMap.
Vi har lagt in en if-sats som bryter loopen vid testkörning av BSTMap, eftersom vi får recursion depth error för stora element. Se rad 53 i experimentRunner.