LINKÖPINGS UNIVERSITET

TDDI16: Datastrukturer och algoritmer

Lab 1: AVL-träd

Tommy Färnqvist, Rita Kovordanyi, Filip Strömbäck

INSTITUTIONEN FÖR DATAVETENSKAP

Höstterminen 2018

2021-08-19

1 Uppgift

I den givna koden finns ett AVL-träd implementerat. Din uppgift är att komplettera AVL-trädet med operationen `remove`. I den givna koden är `remove` deklarerad i klassen `AVL_Tree`, men det enda den gör är att kasta ett undantag. Din uppgift är att implementera `remove` i `AVL_Tree`, vilket bland annat kräver att du lägger till en `remove`-funktion i `AVL_Tree_Node`.

Operationen `remove` ska ta bort det givna elementet om det finns, annars ska den antingen inte göra någonting alls, eller kasta ett undantag. Utgå från den givna funktionen som finns på filen `simple_remove.cpp` och komplettera den med balansjustering. Studera hur balanskontroll och justering görs i `insert`, och gör i princip samma sak i `remove`. Tänk dock noga igenom var en obalans ligger i förhållande till noden som har obalans, och hur man kan avgöra om en enkelrotation eller en dubbelrotation ska göras! På den punkten skiljer sig borttagning påtagligt från vad som gäller vid insättning, även om koden ska bli snarlik. Tänk också på att borttagningen resulterar i att vissa pekare kan bli `null` i `remove` där vi i `insert` vet att de alltid är giltiga.

2 Givna filer

- Filen `avl_tree.h` innehåller definitionen av AVL-trädet och några tillhärande funktioner som ej är medlemmar (`swap` och `operator <<`).
- Filen `avl_tree.cpp` innehåller definitionerna för medlemsfunktionerna i klassen `AVL_Tree` samt definitioner för den nodtyp som AVL-trädet implementeras med, `AVL_Tree_Node`.
- Filen `avl_tree-test.cpp` innehåller ett testprogram för AVL-trädet. Det sätter in några värden i ett AVL-träd och låter sedan en del av operationerna i AVL-trädet att testas interaktivt.
- En funktion för borttagning i ett enkelt binärt sökträd finns på filen `simple_remove.cpp`.
- För att kompilera programmet finns en make-fil, `Makefile`. Kommandot `make` kör den.

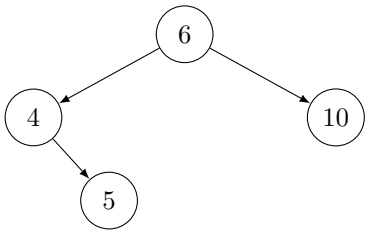
3 Fundera på

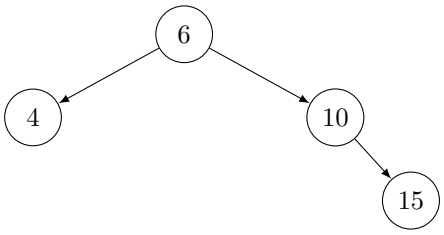
Vad får din `remove`-funktion för tidskomplexitet och minneskomplexitet? Vad borde den ha för tid- och minneskomplexitet?

4 Testfall

Nedan finns tre testfall din kod måste klara av. Observera att din kod naturligtvis måste fungera även i det generella fallet och inte bara på de tre testfallen, så tänk noga igenom vilka situationer som kan uppkomma och hur de ska hanteras!

Testfallen nedan antar att borttagning av en nod med två barn görs genom att ersätta den med efterföljande nod i en inorder-traversering. Det finns andra korrekta sätt att implementera borttagning på också, och det är okej att använda dessa. Dock stämmer inte alla testfall nedan då.



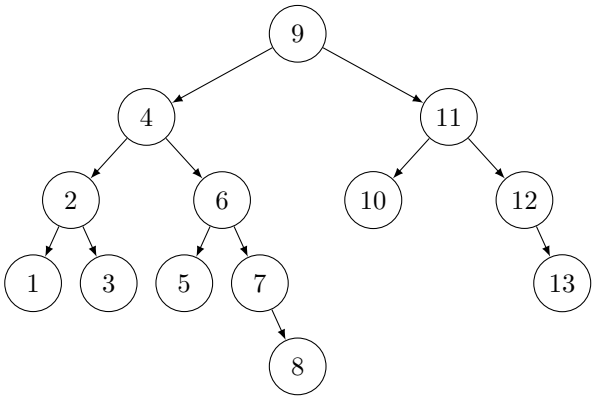


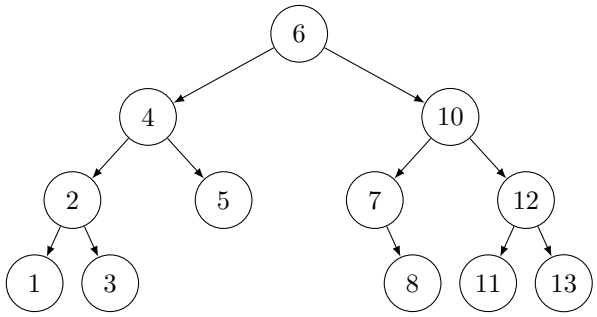
Testfall 1 är borttagning av nod 10 i trädet ovan till vänster och testfall 2 är borttagning av nod 4 i trädet ovan till höger. Träden ska efter borttagning vara korrekt balanserade sökträd och din implementation ska inte krascha. Notera att träden skrivs ut med rotnoden till vänster av testprogrammet. Träden ovan ser alltså ut som nedan när de skrivs ut i terminalen:

```
  10
 /
6
 \
   5
  /
 4
```

```
   15
  /
 10
/
6
 \
  4
```

Testfall 3 är borttagning av nod 9 i trädet nedan till vänster. Trädet kan byggas genom att köra `avl_tree-test` med följande indata: `1 9 1 4 1 11 1 2 1 6 1 10 1 12 1 1 1 3 1 5 1 7 1 13 1 8`. Resultatet av borttagningen ska bli som i trädet nedan till höger.





Använd gärna även exempel från lektion 2 som testfall till labben!