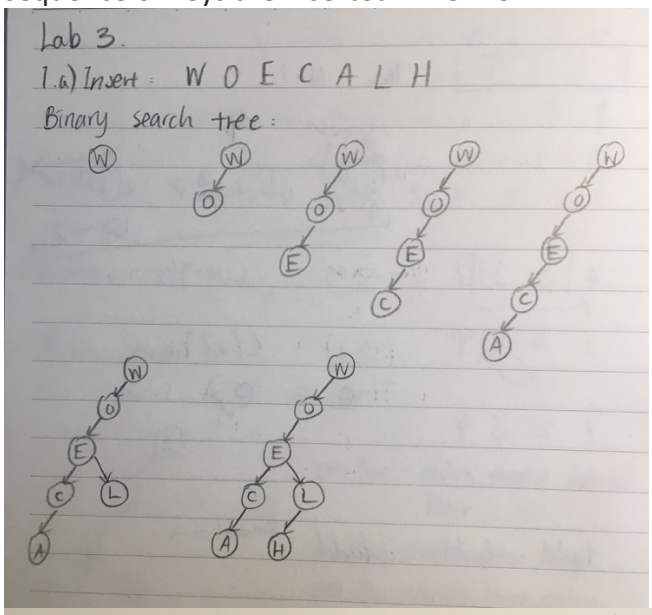
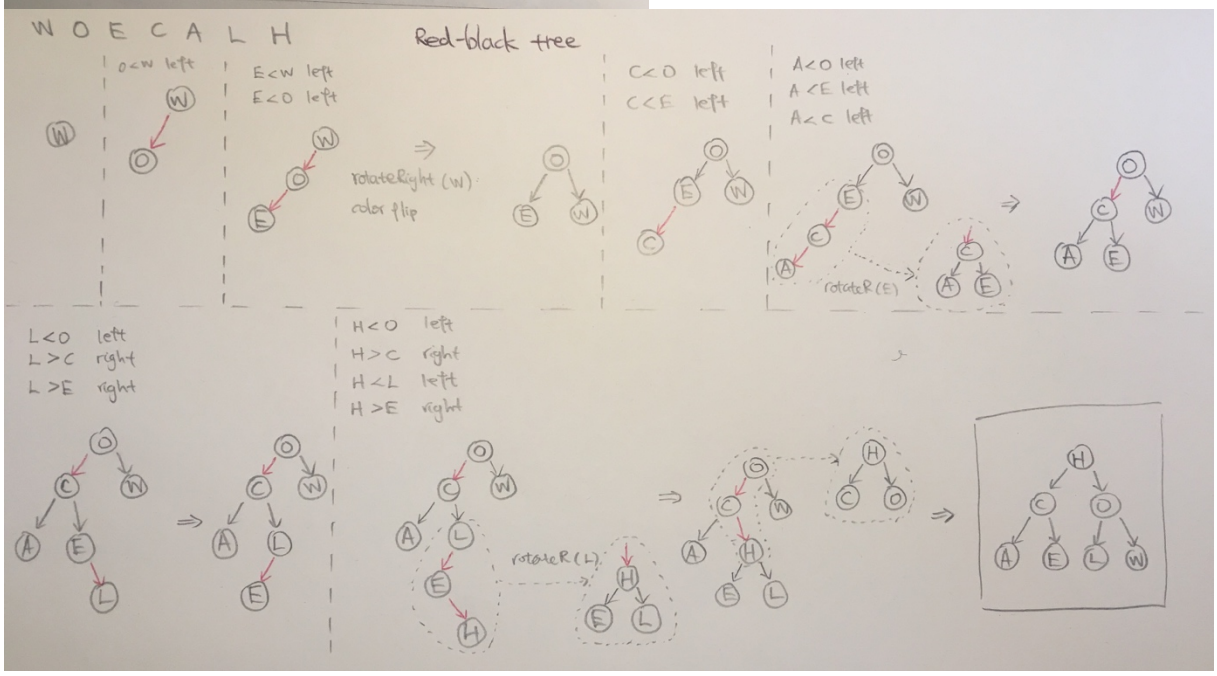
1.

a) Show how a binary search tree and a red-black tree is built when the following sequence of keys are inserted: W O E C A L H





b) Show the output if the content of the trees is printed in pre-, in- and postfix order
For the binary search tree:
Pre-fix: W O E C A L H
In-fix: A C E H L O W
Post-fix: A C H L E O W

For the red-black tree:
Pre-fix: H C A E O L W
In-fix: A C E H L O W
Post-fix: A E C L W O H

2. Explain which of the following could be good (or poor) choices for the initial size of an array used to implement a hash table assuming app. 2000 individual integer value keys and modular hashing: 3014, 4711, 4712, 4713.

The size of a hash table should be a prime number or power of 2, in this case none of the given number is a prime. So, the numbers with least prime factors are better choices, which are 4711(=7*674) and 4713(=3*1571).

3. Se kommentar i koden.

4.

	0 100 < filterad.txt	3 100 < filterad.txt	0 1000 < filterad.txt	0 5000 < filterad.txt
BinarySearchST	55ms	55ms	69ms	102ms
BST	54ms	55ms	72ms	97ms

Tiden mäts från när datastrukturen skapats, till FrequencyCounter hittat det mest frekventa ord. Tabellen visar medelvärden för varje test som kört 3 gånger.
Enligt teorin borde BST vara långsammare pga O(1,39log(n)) på average case och O(n) på worst case. BinarySearchST har O(log(n)) som average och worst case.
I FrequencyCounter itererar båda program igenom hela symbol tabell, så att tiden för att hitta det mest frekventa ord borda bli lika snabba. Men i detta fall mäts tiden från det när datastrukturen skapats, och put() operationen är olika i båda program. När en ny nyckel läggs till i BinarySearchST, måste alla befintliga värden i keys[] och vals[] arrayer flyttas ett steg för att kunna lägga in den nya nyckeln och dess värdet, O(n). Medans i put() operationen i BST räcker det med att skapa en ny node och sedan ligga den på rätt plats, O(log(n)). Skillnaden på tidskomplixitet på put() operationerna utgör därför den stora skillnaden.

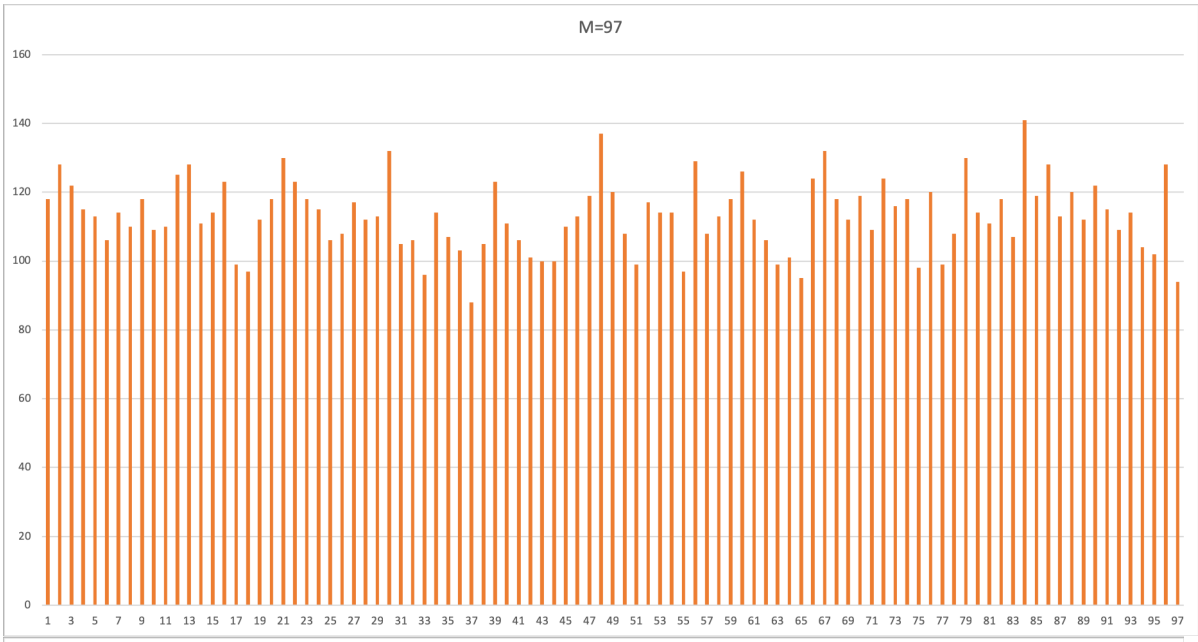
5. I denna uppgiften vald jag att implementera query metod i BinarySearchST. Det borde vara lika snabba att köra metoden som implementeras, printNtoNpX(), på både BinarySearchST och BST, då get() operationen på båda program körs på O(log(n)) tid.

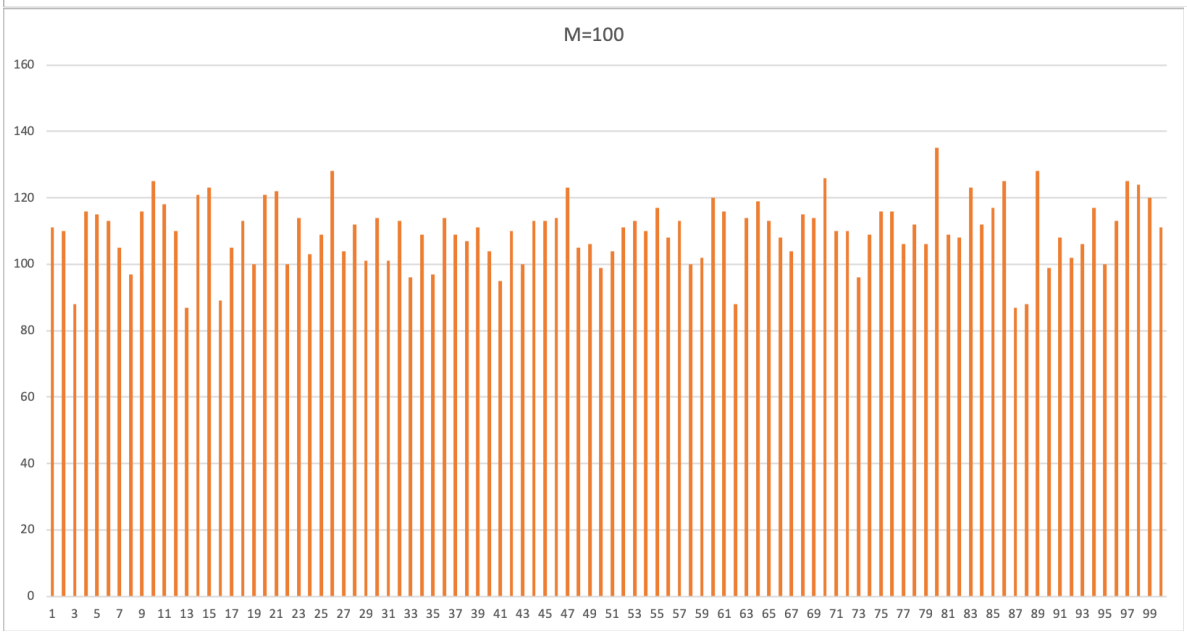
6.

	minlen = 0	minlen = 3	minlen = 10
BST	7ms	8ms	4ms
Red-black tree	10ms	9ms	5ms

Tiden mäts från när FrequencyCounter går igenom alla nycklar, till den hittat det mest frekventa ord. Då FrequencyCounter itererar igenom lika många ord i båda program, bör de vara lika snabba. Dvs trädets uppbyggnad kommer inte utgöra en stor skillnad exekveringstid på båda program.
Tabellen visar medelvärden för varje test som kört 3 gånger med olika minimum längd på orden.

7. Graferna nedan visar hur jämnt hashes fördelas mellan alla distinkta ord i the text, när storleken på hash tabeller är M=97 och M=100.





8. Se kommentar i koden.

9.

	minlen = 0	minlen = 5	minlen = 10
SeparateChainingHashST	354ms	308ms	259ms
(bara max)	5ms	6ms	4ms
LinearProbingHashST	303ms	291ms	259ms
(bara max)	5,6ms	5,6 ms	2,6 ms

Enligt teorin borde det tar lika lång tid O(n) för båda program.
När du använder separate chaining lagrar varje index på hashen en annan datastruktur, detta kan kräva mycket mer minne linear probing, vilket bara behöver en enda array. Dessutom, en annan fördel med en singel data structure är att den kan nås snabbare än separate chaining. Detta beror på att minnesadresserna som används för den arrayen i linear

probing är närmare varandra, medan separate chaining kan ha datastrukturen på olika platser långt ifrån varandra.