Lab 2

Här kommer rapporten som tillhör lab 1 för 1DT907 kursen från Samuel Berg

Hur man kör Main*.java programmen

Först och främst så tar vi och kör det följande kommandot i terminalen/consolen. Detta gör vi för att compile:a vår .java filer till filer så går att köra.

Notera: Eftersom jag kör på linux så är jag ej helt säker om det är samma kommandon på windows, hoppas det ändå ska funka då jag skickar med .class filerna också. Om jag känner till wondows korrekt lägg till `.java` när du ska köra filerna i wondows med `java MainX.java` exempelvis.

```
javac *.java
```

För att köra något av *Main* programmen, skriv en av de följande kommandona in i din terminal/console. I ordning från topppen ner.

1. kör main programmet för `Problem 1` algorithmen
2. kör main programmet för `Problem 2` algorithmen
3. kör main programmet för `Problem 3` algorithmen
4. kör main programmet för `Problem 4` algorithmen
5. går att köra `Problem 5` men ej bra implementerad eller totalt funktionell.
6. kör main programmet för `Problem 6` algorithmen

```
java Main1
java Main2
java Main3
java Main4
java Main5
java Main6
```

Problem 1

Jag löste resizningen för queue:n genom att vid varje enqueue kolla om `size == queue.length` om den var det så ökade jag köns storlek med en faktor av `2`. På liknande sätt vid en dequeue så kollar jag om `size == queue.lenght / 4` om snat så reducerade jag köns storlek med en faktor `2`. för redovisning så här ser min enqueue funktion ut:

```

```

```
public void enqueue(Obj obj) {
        if (obj == null) {
            throw new IllegalArgumentException("Cannot be null");
        }

        if (size == queue.length) {
            resize(2 * queue.length);
        }

        queue[size++] = obj;
    }
```

Skillnaden med enqueue implementationen och dequeue implementationen är att in enqueue kollas det om den behöver resize:as innan insättning medans i dequeue kollas det efter borttagning.

Problem 2

För detta experiment så byggde jag up ett AVL och ett BST träd med 10 000 000 noder i början för att han en förhoppningsvis tydligare tidsskillnad i resultaten. När det vart gjort så körde jag på dessa träd ett "general case test" vilket tar tiden då en mängnd olika tal skall i ordning sättas in, sökas upp och tas väck ur trädet och ta tiden för varje operation för hela mängnden tal. Efter detta är gjort så skrivs det ut i terminalen för BST respektive AVL tärdet deras tider för att gör de olika operationerna för den ängden tal vi hade samt att vi skriver ut höjden på trädet från root:en till yttersta noden. Ett exemple resultat som blev är det följande:

```
BST:
insert: 4.0109 s
search: 2.7219 s
delete: 3.4082 s
height: 60

AVL:
insert: 4.7135 s
search: 2.6945 s
delete: 4.0048 s
height: 28

Diff (AVL - BST):
insert: 702.6044 ms
search: -27.3676 ms
delete: 596.5343 ms
height: -32
```

Där vi också tar och reprensentera skillnaden i tid för operationerna för de olika träden. som vi kan se så har AVL trädet som förväntat en längre insättnings och uttagnings tid jämfört med BST trädet men en effektivare sök operation då AVL trädet balanserar sig under upp byggnaden jämfört med BST trädet vilket kan ses på `height` noteringen i Diff delen där vi kan se att höjden från root:en till yttersta node för AVL trädet är 32 noder närmare än för BST trädet vilket leder till den ökade prestandan i sökning. Detta vart

dock endast generela fallet om vi tar och tittar på bästa samt värsta fallet så kommer utskriften i terminalen möjligen se ut som följande:

```
Best case:
Time to insert, search & delete 16383 objects
BST test: completed
AVL test: completed

BST:
insert: 1.0566 ms
search: 3.1272 ms
delete: 2.0912 ms
height: 14

AVL:
insert: 2.0998 ms
search: 3.4103 ms
delete: 3.3879 ms
height: 14

Diff (AVL - BST):
insert: 1.0433 ms
search: 283.0750µs
delete: 1.2966 ms
height: 0
```

Medans värsta fallet:

```
Worst case:
Time to insert, search & delete 16383 objects
BST test: completed
AVL test: completed

BST:
insert: 2.2255 s
search: 1.7053 s
delete: 272.3140µs
height: 16383

AVL:
insert: 3.2341 ms
search: 2.0042 ms
delete: 2.2163 ms
height: 14

Diff (AVL - BST):
insert: -2.2223 s
search: -1.7033 s
delete: 1.9439 ms
height: -16369
```

Från detta kan vi se att om det som ska sättas in i träden är i korrekt ordning för att bygga upp trädet perfekt så är BST trädet ganska överlägset jämfört med AVL trädet, men om vi tar en titt på värsta fallet så ser vi att AVL trädet totalt sätt för mängden tal är effektivare på insättning samt upp sökning men förlorar när det kommer till borttagning då AVL trädet bygger om sig själv då det blir för obalanserat. Väldigt tydligt också med höjd skillnaden på träden är att AVL trädet är att föredra i detta fall.

I min åsikt så hade jag sagt att det är mycket mer lämpligt med ett balanserat träd då man hanterar stora data mängder på grund av att om det växer i snitt eller värsta fallet kommer BST trädet att bli så obalancerat att det kan ta uppåt flera minuter om inte mer att nå något som ligger ytterst i trädet jämfört med AVL trädet som kommer att ha byggt om sig själv till att ge oss snabbaste möjliga anslutning till värdet oavsett insättnings ordning.

Problem 3

Min implementation har de krävda funktionerna så här ser `insertPerson()` ut:

```
public void insertPerson(String name, int prio) {
        if (size == arr.length) {
            resize();
        }

        int place = size;

        arr[size] = new Person(name, prio, place);
        size++;
    }
```

Enligt mina tester så ser jag att jag har 3 av de 4 funktionerna som är beroende på antalet personer i kön den enda som är icke beroende av detta är `insertPerson()` detta är på grund av att för att jag sorterar arrayet i alla andra dunktioner men inte vid insättning av personer. De andra tre som är enligt mina experiment är beroende på sorterings algoritmen man använder för att sortera personerna i kön efter prioritet och placering i kön. Jag håller koll och sortera personerna korrekt med hjälp av denna biten kod i heapify:

```
if (l < n && arr[l].getPriority() > arr[smallest].getPriority()) {
    smallest = l;
} else if (l < n && arr[l].getPriority() == arr[smallest].getPriority()) {
    if (arr[l].getPlace() > arr[smallest].getPlace()) {
        smallest = l;
    }
}

if (r < n && arr[r].getPriority() > arr[smallest].getPriority()) {
    smallest = r;
} else if (r < n && arr[r].getPriority() == arr[smallest].getPriority()) {
    if (arr[r].getPlace() > arr[smallest].getPlace()) {
        smallest = r;
```

```
    }
}
```

Jag vill dock påpeka att min första tanke vart att göra en `@Override` på `compareTo` funktionaliteten hos objekt men vilket inte ville fungerar för mig. Detta har i slut ändan gett mig följande output då jag har en kö med 1 000 000 personer med 10 olika möjliga prioriteter.

```
Running tests for queue with: 1000000 persons
Testing getPerson(): 2.1509 s (Person{name='test22', prio=0, place=22})
Testing insertPerson(): 3.3500µs (Person{name='Alice', prio=0})
Testing swapPriority(): 1.6638 s (swaped Person{name='Alice', prio=0} &
Person{name='Adam', prio=7})
Testing deleteMinPriority(): 2.4902 s (Person{name='test22', prio=0,
place=22})

Next person to buy a ticket: Person{name='test32', prio=0, place=32}
Next person to buy a ticket after deleting: Person{name='test34', prio=0,
place=34}
Next person to buy a ticket after swapping: Person{name='test34', prio=0,
place=34}
0:Person{name='test34', prio=0, place=34}
1:Person{name='test47', prio=0, place=47}
2:Person{name='test55', prio=0, place=55}
3:Person{name='test60', prio=0, place=60}
4:Person{name='test67', prio=0, place=67}
5:Person{name='test71', prio=0, place=71}
6:Person{name='test73', prio=0, place=73}
7:Person{name='test118', prio=0, place=118}
8:Person{name='test122', prio=0, place=122}
9:Person{name='test125', prio=0, place=125}
            |
            |
            |
999989:Person{name='test999876', prio=9, place=999876}
999990:Person{name='test999877', prio=9, place=999877}
999991:Person{name='test999895', prio=9, place=999895}
999992:Person{name='test999899', prio=9, place=999899}
999993:Person{name='test999970', prio=9, place=999970}
999994:Person{name='test999974', prio=9, place=999974}
999995:Person{name='test999978', prio=9, place=999978}
999996:Person{name='test999980', prio=9, place=999980}
999997:Person{name='test999985', prio=9, place=999985}
999998:Person{name='test999997', prio=9, place=999997}
```

Dessa tider känns något rimliga då man tänker på hur allt vart implementerat samt att jag använder en egen quicksort variant för att sortera personerna i kön. De 3 so beror på storleken av kön enligt mina beräkningar har en tids komplexitet lik sorteringsalgorithmen vilket ger dem en ungifärlig tidskomplexitet på `O(c * log(n))`, konstanten c är det samma som basen för logaritmen vilket i detta fall är 2 vilket ger en ungifärlig tidskomplexitet av `O(2 * log(n))`.

Problem 4

Jag gjorde 100 stycken experiment och fick fram i snitt fallet det följande:

```
Heapsort is the most efficient algoritm with time: 28.6400 ms and for
recursion depth: 5
Insertionsort is the most efficient algorithm with time: 12.2572 ms and for
recursion depth: 10
Insertionsort is the most efficient algorithm with time: 3.0289 ms and for
recursion depth: 15
Insertionsort is the most efficient algorithm with time: 891.8350µs and for
recursion depth: 20
Insertionsort is the most efficient algorithm with time: 538.3060µs and for
recursion depth: 25
Insertionsort is the most efficient algorithm with time: 517.2630µs and for
recursion depth: 30
Insertionsort is the most efficient algorithm with time: 524.9120µs and for
recursion depth: 35
Insertionsort is the most efficient algorithm with time: 580.2780µs and for
recursion depth: 40
Insertionsort is the most efficient algorithm with time: 503.6210µs and for
recursion depth: 45
Insertionsort is the most efficient algorithm with time: 504.2510µs and for
recursion depth: 50
Insertionsort is the most efficient algorithm with time: 472.2160µs and for
recursion depth: 55
Insertionsort is the most efficient algorithm with time: 554.5380µs and for
recursion depth: 60
Insertionsort is the most efficient algorithm with time: 503.3370µs and for
recursion depth: 65
Insertionsort is the most efficient algorithm with time: 502.5890µs and for
recursion depth: 70
Insertionsort is the most efficient algorithm with time: 503.3980µs and for
recursion depth: 75
Insertionsort is the most efficient algorithm with time: 471.0200µs and for
recursion depth: 80
Insertionsort is the most efficient algorithm with time: 504.1580µs and for
recursion depth: 85
Insertionsort is the most efficient algorithm with time: 508.2830µs and for
recursion depth: 90
Insertionsort is the most efficient algorithm with time: 503.7300µs and for
recursion depth: 95
Insertionsort is the most efficient algorithm with time: 504.6870µs and for
recursion depth: 100
```

Detta gäller då vi kollar på en array med 100 000 element och går i steg av 5 med depth med quicksort innan jag skapar kopior av array:et för att sortera det sorterade array:et med heap- respektive insertionsort.

Problem 6

Från min tester med mina olika shellsort sekvenser så får jag att tidskomplexiteten stämmer någlunda väl överäns med vad som representerads i wikipedia artiklen som medgavs med uppgiften. Jag får resulta liknande följande:

```
Testing for array size of 100000
ShellSort w seq: hibbard, time: 47.2076 ms
ShellSort w seq: sedgewick, time: 30.7012 ms
ShellSort w seq: knuth, time: 36.5370 ms

Testing for array size of 200000
ShellSort w seq: hibbard, time: 75.7807 ms
ShellSort w seq: sedgewick, time: 68.1272 ms
ShellSort w seq: knuth, time: 76.2070 ms

Testing for array size of 300000
ShellSort w seq: hibbard, time: 99.6089 ms
ShellSort w seq: sedgewick, time: 105.0120 ms
ShellSort w seq: knuth, time: 120.5488 ms

Testing for array size of 400000
ShellSort w seq: hibbard, time: 137.0622 ms
ShellSort w seq: sedgewick, time: 144.4007 ms
ShellSort w seq: knuth, time: 156.3455 ms

Testing for array size of 500000
ShellSort w seq: hibbard, time: 168.0282 ms
ShellSort w seq: sedgewick, time: 180.3242 ms
ShellSort w seq: knuth, time: 217.6204 ms

Testing for array size of 600000
ShellSort w seq: hibbard, time: 263.0259 ms
ShellSort w seq: sedgewick, time: 238.5287 ms
ShellSort w seq: knuth, time: 230.5272 ms

Testing for array size of 700000
ShellSort w seq: hibbard, time: 241.2289 ms
ShellSort w seq: sedgewick, time: 267.4571 ms
ShellSort w seq: knuth, time: 298.7250 ms

Testing for array size of 800000
ShellSort w seq: hibbard, time: 333.1191 ms
ShellSort w seq: sedgewick, time: 314.2258 ms
ShellSort w seq: knuth, time: 351.3146 ms

Testing for array size of 900000
ShellSort w seq: hibbard, time: 392.4427 ms
ShellSort w seq: sedgewick, time: 405.3004 ms
ShellSort w seq: knuth, time: 462.7835 ms

Testing for array size of 1000000
ShellSort w seq: hibbard, time: 471.7961 ms
```

```
ShellSort w seq: sedgewick, time: 503.8592 ms
ShellSort w seq: knuth, time: 574.7555 ms
```

Dessa resultaten tycker jag verkar rimliga vid denna storlek på input array och verkar också skala på ett sätt enligt den adviserade tidskomplexiteten.