buildHeap() Sorted Input (Test in milliseconds)											
Input	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9	Test 10	Average
40,000	3	2	2	2	2	2	2	2	2	2	2.1
50,000	5	1	2	2	1	1	1	1	1	2	1.7
60,000	3	1	1	1	1	1	1	1	1	1	1.2
70,000	1	0	0	1	0	0	0	0	0	0	0.2
80,000	1	0	1	1	0	1	1	1	0	0	0.6
90,000	1	0	2	3	0	1	1	2	0	1	1.1
500,000	11	7	9	7	7	9	8	8	8	7	8.1
600,000	12	8	9	8	8	9	10	8	10	8	9
700,000	11	11	9	11	11	11	8	9	9	12	10.2
800,000	13	12	11	12	12	12	10	10	10	12	11.4
900,000	14	13	12	14	13	13	11	11	13	12	12.6
1,000,000	15	15	14	14	15	15	12	14	13	13	14
3,000,000	55	45	37	44	45	46	38	38	38	40	42.6
4,000,000	57	55	45	57	56	55	46	46	47	44	50.8
5,000,000	197	153	61	157	158	77	142	140	144	63	129.2
6,000,000	85	82	74	81	82	93	68	70	68	75	77.8
7,000,000	111	105	88	106	105	107	89	86	88	88	97.3
8,000,000	126	118	101	121	119	119	102	98	100	99	110.3
9,000,000	139	122	115	126	123	135	102	101	101	115	117.9
10,000,000	153	148	127	149	150	149	129	123	126	129	138.3
										Total AVG	41.82

The first piece of data that was tested was using the buildHeap() method with sorted data. Using buildHeap() with sorted data had the lowest average runtime out of all of the other tests ran, which was 41.82 milliseconds. Keep in mind that heaps have an order property, so this is undoubtedly due to the fact that the data was already sorted which means the heap does not have to sort the data while it is being entered. Depending on the number of items, this could become costly. The range of numbers entered were between 40,000-10,000,000. While running the initial tests I found that using too low of numbers resulted in the time being 0 which is essentially useless when comparing runtimes. 40,000 provided a good starting point for consistently getting numbers greater than 0 to average out and compare. Also, it is important to note that the program was ran a total of ten times. It was decided that the program needed to be ran a total of 10 times to gather a substantial amount of data to be compared and analyzed. Referring to the table, you can see that the higher the input gets the higher the average runtimes become because buildHeap() has an O(n) runtime.

insert() Sorted Input (Test in milliseconds)											
Input	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9	Test 10	Average
40,000	20	7	4	15	14	12	13	12	13	13	12.3
50,000	5	4	4	3	3	4	2	4	3	3	3.5
60,000	2	2	4	1	2	5	2	1	2	2	2.3
70,000	1	4	2	1	1	1	2	1	2	1	1.6
80,000	1	2	2	1	1	1	1	1	1	1	1.2
90,000	1	1	2	1	1	1	1	1	1	1	1.1
500,000	18	14	16	15	14	14	14	15	15	14	14.9
600,000	19	16	16	16	16	16	15	15	15	16	16
700,000	22	17	18	17	18	18	17	18	18	18	18.1
800,000	32	27	342	26	25	295	26	25	26	324	114.8
900,000	44	28	27	28	29	27	29	28	28	27	29.5
1,000,000	462	304	32	340	329	30	302	292	321	29	244.1
3,000,000	100	90	86	80	80	79	80	80	81	80	83.6
4,000,000	275	125	120	121	121	120	120	121	121	121	136.5
5,000,000	571	432	466	395	410	406	389	417	470	460	441.6
6,000,000	217	177	166	176	178	164	174	176	178	163	176.9
7,000,000	639	513	659	492	495	613	494	500	487	659	555.1
8,000,000	668	526	252	566	530	290	518	525	531	259	466.5
9,000,000	479	330	573	328	322	581	327	331	320	574	416.5
10,000,000	689	564	589	593	599	592	546	564	580	578	589.4
										Total AVG	166.275

The next piece of data that was tested was using the insert() method with sorted data. The total average runtime was 166.275 milliseconds. So this method was obviously considerably slower than buildHeap() which will remain consistent for all data types throughout the analysis. This is due to the cost of having to loop through the data and insert one by one which is a lot more costly than the buildHeap() method. For the most part as the number of items being added to the heap becomes larger so do the average runtimes.

buildHeap() Reverse Input (Test in milliseconds)											
Input	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9	Test 10	Average
40,000	4	3	3	4	3	2	2	4	3	3	3.1
50,000	1	1	1	0	1	1	2	1	0	0	0.8
60,000	1	0	1	0	0	0	0	0	0	1	0.3
70,000	1	0	0	0	0	0	0	0	0	0	0.1
80,000	1	0	0	0	1	0	0	0	0	1	0.3
90,000	1	1	1	1	1	1	19	1	1	1	2.8
500,000	18	15	12	13	13	12	13	15	15	12	13.8
600,000	28	15	17	15	14	16	19	15	17	16	17.2
700,000	30	18	22	18	16	20	19	18	20	18	19.9
800,000	27	22	24	23	20	24	23	24	24	22	23.3
900,000	30	24	30	25	22	28	24	28	27	23	26.1
1,000,000	34	28	27	27	25	26	31	29	30	27	28.4
3,000,000	120	91	94	92	84	89	92	96	99	88	94.5
4,000,000	362	226	124	216	221	119	227	220	216	129	206
5,000,000	176	147	149	146	137	150	149	149	149	150	150.2
6,000,000	225	198	193	188	173	188	192	191	187	190	192.5
7,000,000	261	208	216	212	197	226	218	212	214	219	218.3
8,000,000	325	250	245	330	231	241	252	252	252	244	262.2
9,000,000	353	285	280	285	266	278	288	290	286	274	288.5
10,000,000	386	324	313	318	293	303	315	310	324	315	320.1
										Total AVG	93.42

The next piece of data analyzed is using the buildHeap() method with reversed data. This will become a little more costly due to the ordered property of heaps because now the data needs to be sorted. The total average runtime was 93.42. This is a little higher than the sorted data average runtime of 41.82 but not much. This is still pretty efficient. The trend of the average runtimes for each input growing larger as the input grows larger continues to be present.

insert() Reverse Input (Test in milliseconds)											
Input	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9	Test 10	Average
40,000	9	8	9	8	7	8	8	7	8	8	8
50,000	4	4	4	3	3	3	4	3	4	4	3.6
60,000	4	4	3	4	3	4	4	4	4	4	3.8
70,000	6	4	4	4	4	4	4	5	5	5	4.5
80,000	7	6	5	5	6	5	5	6	5	6	5.6
90,000	6	6	5	5	6	6	5	5	5	10	5.9
500,000	96	82	80	81	77	78	76	78	78	77	80.3
600,000	104	104	88	90	89	89	87	88	87	90	91.6
700,000	127	125	106	106	104	105	104	105	104	106	109.2
800,000	149	145	128	129	128	128	127	127	128	129	131.8
900,000	474	404	144	404	388	145	384	391	382	144	326
1,000,000	213	169	403	164	164	387	163	163	165	383	237.4
3,000,000	664	542	522	514	514	524	513	515	515	526	534.9
4,000,000	897	750	723	723	723	756	720	723	726	724	746.5
5,000,000	1448	1199	1228	1216	1212	1211	1236	1206	1198	1257	1241.1
6,000,000	1321	1082	1083	1091	1081	1084	1084	1081	1080	1084	1107.1
7,000,000	1986	1615	1690	1667	1593	1618	1627	1588	1607	1667	1665.8
8,000,000	2265	1949	1869	1898	1876	1845	1904	1893	1897	1845	1924.1
9,000,000	2042	1740	1711	1737	1711	1707	1711	1711	1723	1711	1750.4
10,000,000	2748	2355	2195	2364	2384	2208	2353	2341	2367	2241	2355.6
										Total AVG	616.66

When comparing insert() with buildHeap() with reversed data we really see a huge jump in average runtime. The total average runtime is 616.66 milliseconds compared to buildHeap()'s 93.42. The insert() method is now doing even more additional steps to sort the data which slows this method down severely. It is important to note that when the input size is low there is not a huge difference between insert() and buildHeap() but when the input ranges between 1,000,000 and 10,000,000 there is a more significant difference. So when using a small input the difference between the two would probably be unrecognizable. On the other hand, when using a large input like 10,000,000 the average runtime of buildHeap() is 320.1 milliseconds compared to insert()'s 2,355.6 millisecond runtime.

buildHeap() Random Input (Test in milliseconds)											
Input	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9	Test 10	Average
40,000	1	0	4	2	2	2	1	5	1	1	1.9
50,000	2	4	2	1	2	1	2	1	1	2	1.8
60,000	2	1	2	1	1	1	1	1	1	1	1.2
70,000	1	1	1	1	1	1	1	1	2	1	1.1
80,000	2	1	1	1	1	1	1	1	2	1	1.2
90,000	2	1	1	1	1	1	1	2	1	1	1.2
500,000	19	16	14	16	12	17	15	15	16	15	15.5
600,000	23	20	20	21	16	19	16	20	18	19	19.2
700,000	28	20	20	19	22	20	24	24	20	21	21.8
800,000	49	26	24	23	25	24	26	24	22	23	26.6
900,000	32	37	27	31	31	29	42	27	29	29	31.4
1,000,000	42	29	37	29	29	34	29	29	31	29	31.8
3,000,000	113	95	96	91	94	93	101	94	98	94	96.9
4,000,000	165	132	143	124	129	141	121	123	123	141	134.2
5,000,000	223	175	287	175	176	291	173	176	187	291	215.4
6,000,000	257	203	187	196	197	187	198	202	194	193	201.4
7,000,000	477	402	243	380	403	224	382	394	393	231	352.9
8,000,000	315	254	260	258	258	245	255	255	251	249	260
9,000,000	361	309	296	303	301	321	307	302	295	299	309.4
10,000,000	430	363	340	359	365	332	350	373	345	348	360.5
										Total AVG	104.27

The runtimes for random input varied because of the fact the input is being added in a random order so the difference between insert() and buildHeap() was not huge. For the buildHeap() method the size average runtimes grew consistently as the size of the input grew. The total average runtime of 104.27 milliseconds is still pretty low.

insert() Random Input (Test in milliseconds)											
Input	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9	Test 10	Average
40,000	2	2	2	2	2	2	2	2	2	2	2
50,000	3	3	3	3	3	3	3	3	3	3	3
60,000	2	2	2	2	2	2	2	2	2	2	2
70,000	3	2	2	2	2	2	2	2	2	2	2.1
80,000	3	3	3	3	3	3	3	3	3	3	3
90,000	4	3	4	4	3	4	3	3	3	3	3.4
500,000	30	26	29	25	25	27	26	24	26	25	26.3
600,000	35	28	30	29	29	30	29	29	28	28	29.5
700,000	42	34	416	33	33	405	35	33	34	453	151.8
800,000	375	45	44	47	47	43	48	47	47	43	78.6
900,000	63	288	49	279	284	49	281	295	290	50	192.8
1,000,000	69	54	54	56	50	51	51	53	52	52	54.2
3,000,000	182	145	140	142	141	138	145	144	144	146	146.7
4,000,000	264	209	201	205	200	204	204	201	203	202	209.3
5,000,000	775	589	622	605	605	606	600	609	620	610	624.1
6,000,000	356	275	278	277	272	273	275	305	279	273	286.3
7,000,000	977	770	763	736	750	750	684	657	733	746	756.6
8,000,000	1001	883	785	843	864	765	858	829	843	769	844
9,000,000	612	453	798	489	439	776	451	445	452	791	570.6
10,000,000	1046	922	752	907	912	747	828	842	875	735	856.6
										Total AVG	242.145

As stated earlier the insert() method is still less efficient than buildHeap(). The total average runtime of insert() with random input is 242.145 milliseconds. The randomness of the input seems a little more prevalent in the average runtimes with insert(). For example, the average runtime for 700,00 is 151.8 and periodically from test to test the runtime jumps from around 35 milliseconds to 400 milliseconds but when the input size is 1,000,000 the average runtime is 54.2 milliseconds and stays more consistent from test to test.