HW3

Yao, Yi
sakfljklasdj

1. Application Introduction:
   This software simulates the operation of a north-south-one-lane bridge, whose capacity , as well as the north and south input buffers (queues) are in theory $+\infty$. When one side ends traversing the bridge, the other side starts traversing if cars are present in queue, otherwise, the same side starts traversing, if cars are present in queue. The model is implemented using Event Scheduling based Discrete Event Simulation.

2. Included is the file api.h:

```
typedef struct Event Event;
typedef struct FutureEventList FutureEventList;
typedef struct Engine Engine;
/*schedules an event to the engine*/
int schedule_event(Engine* engine, double time_stamp, int type, void* data);
/*creates a new engine*/
Engine* new_Engine(double duration);
/*prototype for event handler. to be implemented in application*/
int handle(Engine* engine, Event* event);
/*the "main loop". start the simulation using this function*/
int main_loop(Engine* e);
/*get the data from an event*/
void* get_data(Event* e);
/*get event type from an event*/
int get_type(Event* e);
/*get time stamp from an event*/
double get_time_stamp(Event* e);
/*get the current time from the engine*/
double get_time(Engine* e);
/*frees the memory allocated to the engine*/
int free_Engine(Engine* e);
/*generates a number from uniform distribution*/
double rand_unif(double lb, double ub);
/*generates a number from exponential distribution*/
double rand_exp(double mean);
```

   In this file that defines the api, there are function prototypes for creating the engine, scheduling events, starting simulation, generating random numbers and other utility functions.

3.

4.

5. Engine Introduction:
   The simulation engine implementation includes a future event list implemented with heap. The heapify up and heapify down functions are implemented with swapping the actual nodes.

6. Engine Testing Procedure: In order to make sure that the engine works, I wrote a trivial testing simulation.
   The testing program is in src/test_app

```
schedule_event(engine, 2.3, 1, new_Point(1, 1));
schedule_event(engine, 3, 1, new_Point(2, 2));
schedule_event(engine, 2, 1, new_Point(3, 3));
schedule_event(engine, 4, 1, new_Point(4, 4));
schedule_event(engine, 3, 1, new_Point(1, 1));
schedule_event(engine, 1, 1, new_Point(3, 3));
schedule_event(engine, 6, 1, new_Point(3, 3));
schedule_event(engine, 0.5, 1, new_Point(4, 4));
schedule_event(engine, 1, 1, new_Point(5, 5));
```

   The point starts at (0, 0). If the event type is 1, the new point is added to the point. If the type is 2, the new point is subtracted from the point. When an event is handled, an event will be scheduled at timestamp + 3 to subtract (1, 1)
   Here's the output from the test program:

```
0, 0
0.500000
4, 4
1.000000
7, 7
1.000000
12, 12
2.000000
15, 15
2.300000
16, 16
3.000000
18, 18
3.000000
19, 19
3.500000
18, 18
4.000000
22, 22
```

```
4.000000
21, 21
4.000000
20, 20
5.000000
19, 19
5.300000
18, 18
6.000000
17, 17
6.000000
16, 16
6.000000
19, 19
6.500000
18, 18
7.000000
17, 17
7.000000
16, 16
7.000000
15, 15
8.000000
14, 14
8.300000
13, 13
9.000000
12, 12
9.000000
11, 11
9.000000
10, 10
9.500000
9, 9
10.000000
8, 8
10.000000
7, 7
10.000000
6, 6
```

From the output we can be sure that the behavior of the simulation engine is well defined.

7. Here's a sample output:

Dir	TimeArr	TimeMov	TimeOut	TIQ	TIS
NORTH	65.465061	65.465061	105.762517	0.000000	40.297456
SOUTH	75.440895	105.762517	148.818366	30.321622	73.377472
NORTH	125.699182	148.818366	193.804178	23.119185	68.104996
SOUTH	106.750896	193.804178	237.705550	87.053282	130.954654
SOUTH	114.222720	196.929366	240.830737	82.706645	126.608017
SOUTH	133.609873	199.878161	243.779533	66.268288	110.169660
SOUTH	148.220063	203.089423	246.990794	54.869359	98.770731
SOUTH	201.133643	204.515437	248.416809	3.381795	47.283166
NORTH	216.798590	248.416809	292.703603	31.618219	75.905014
NORTH	274.254592	292.703603	333.780362	18.449011	59.525770
SOUTH	309.802971	333.780362	376.752104	23.977391	66.949133
NORTH	300.015705	376.752104	419.152355	76.736399	119.136650
NORTH	321.762146	379.432453	421.832705	57.670307	100.070559
NORTH	393.771725	421.832705	463.735610	28.060980	69.963886
NORTH	405.871754	425.036287	466.939192	19.164533	61.067439
SOUTH	448.277374	466.939192	506.994708	18.661818	58.717334
NORTH	470.733064	506.994708	549.244130	36.261645	78.511067
SOUTH	492.720915	549.244130	592.009651	56.523215	99.288736
SOUTH	501.895626	552.526795	595.292315	50.631169	93.396690
NORTH	508.959399	595.292315	637.262944	86.332916	128.303545
NORTH	537.650725	598.634471	640.605099	60.983746	102.954375
SOUTH	561.635484	640.605099	683.964007	78.969616	122.328523
SOUTH	576.808752	643.944417	687.303325	67.135665	110.494572
NORTH	676.417298	687.303325	729.563604	10.886026	53.146305
NORTH	729.132733	729.563604	773.184284	0.430871	44.051551
NORTH	764.052622	773.184284	814.915784	9.131662	50.863163
SOUTH	811.802807	814.915784	855.019165	3.112977	43.216358
NORTH	813.499103	855.019165	897.215399	41.520062	83.716296
NORTH	823.792447	857.803768	900.000002	34.011321	76.207555
SOUTH	834.381065	900.000002	944.653557	65.618937	110.272493
SOUTH	893.403418	903.264656	947.918211	9.861238	54.514793
SOUTH	917.681540	947.918211	989.211762	30.236671	71.530222
SOUTH	922.471176	950.563446	991.856996	28.092269	69.385820

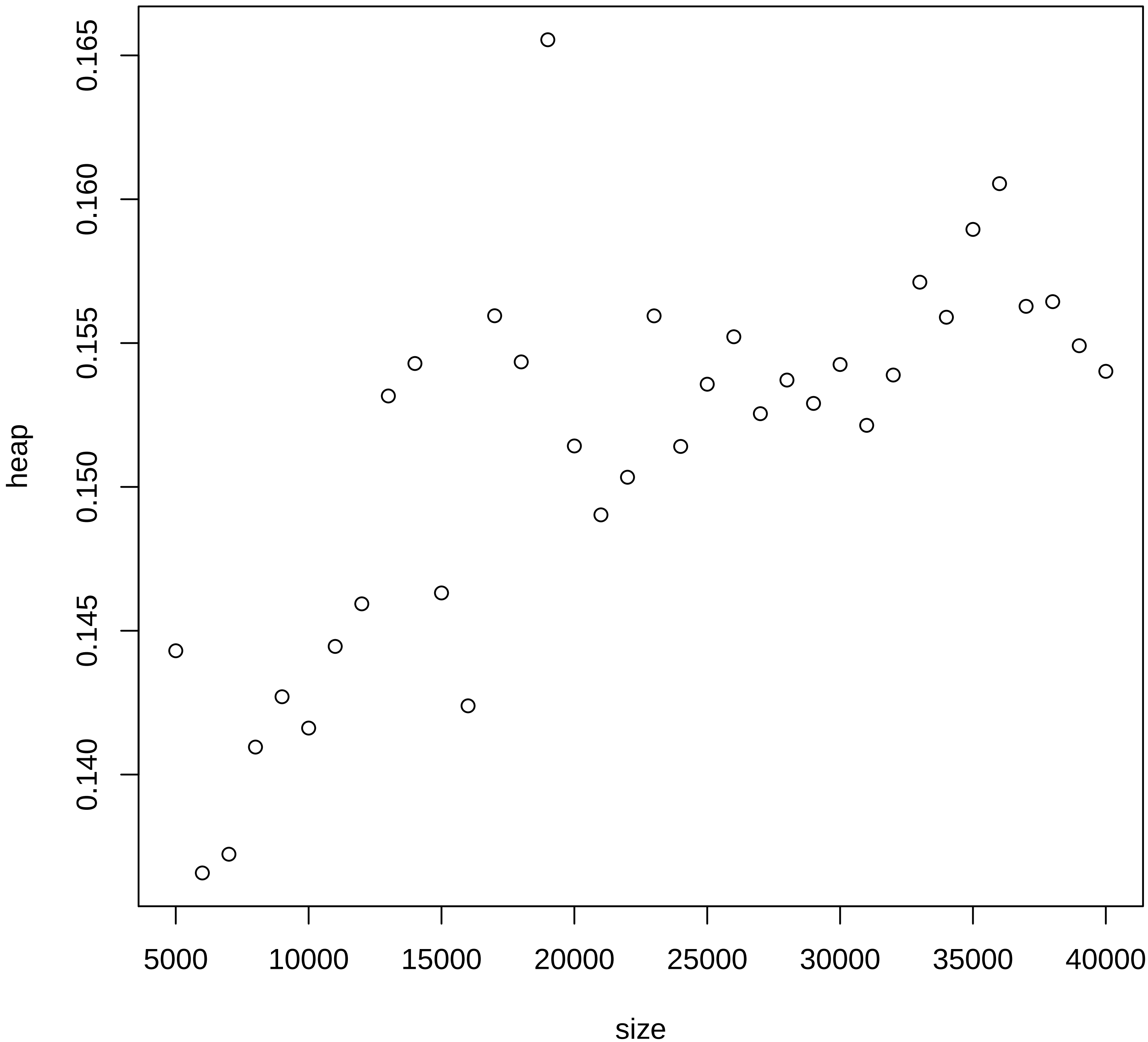
North interarrival times have a mean of 50 seconds, while south interarrival times have a mean of 60 seconds.
As is shown in the output, cars wait in queue when supposed to, traverse the bridge when supposed to and leaves the system when supposed to.

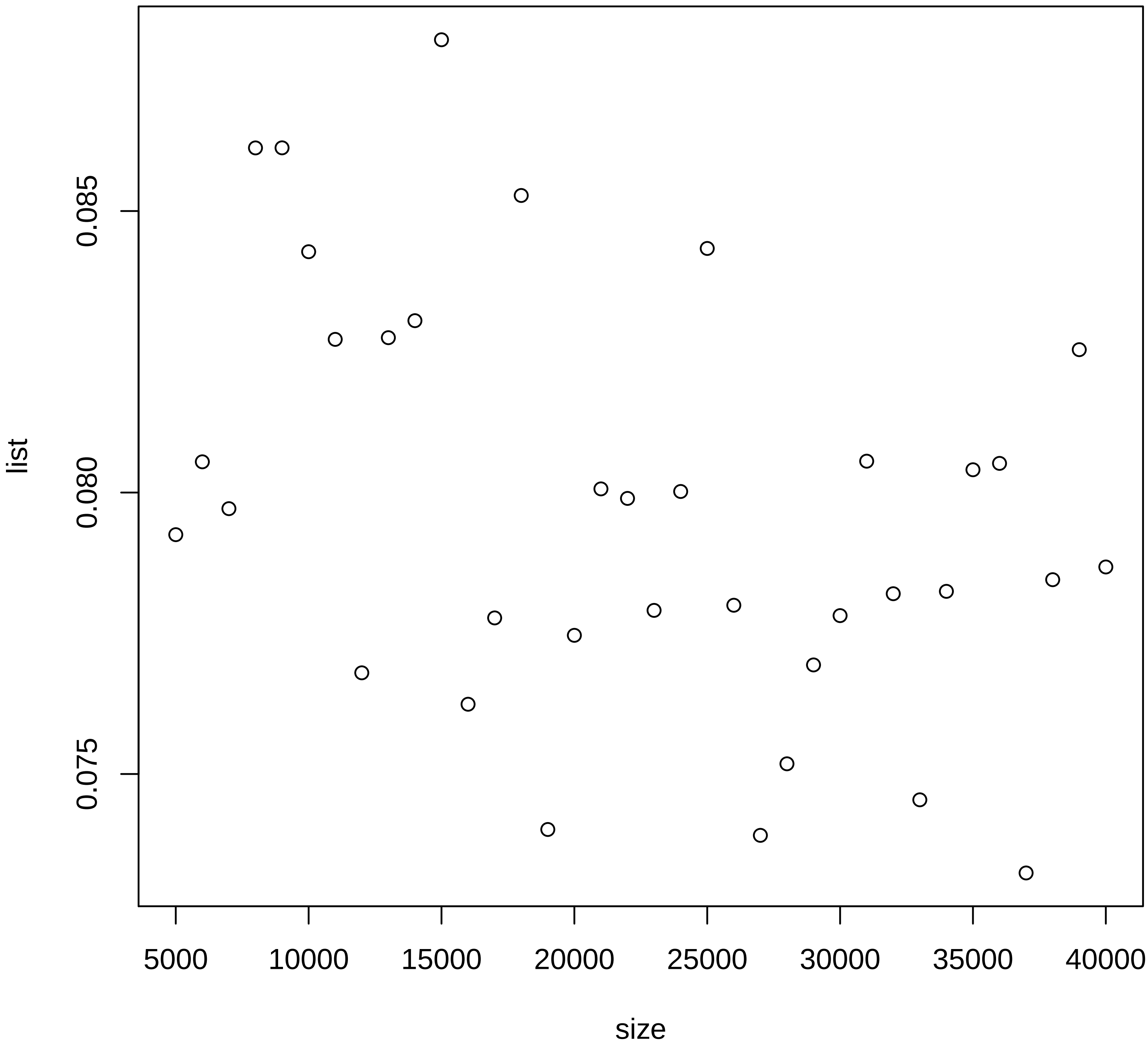
8. (a) Below are plots from the experiments on the engine.
The benchmark programs are in src/benchmark
Using the scripts benchmark and benchmark1 in project root, experiments are done. The first experiment is done with initial events whose difference in timestamps exponentially distributed with mean 1. It is supposed to be close to what the application is doing.
The results are in the first three plots. In the third plot, 1 represents

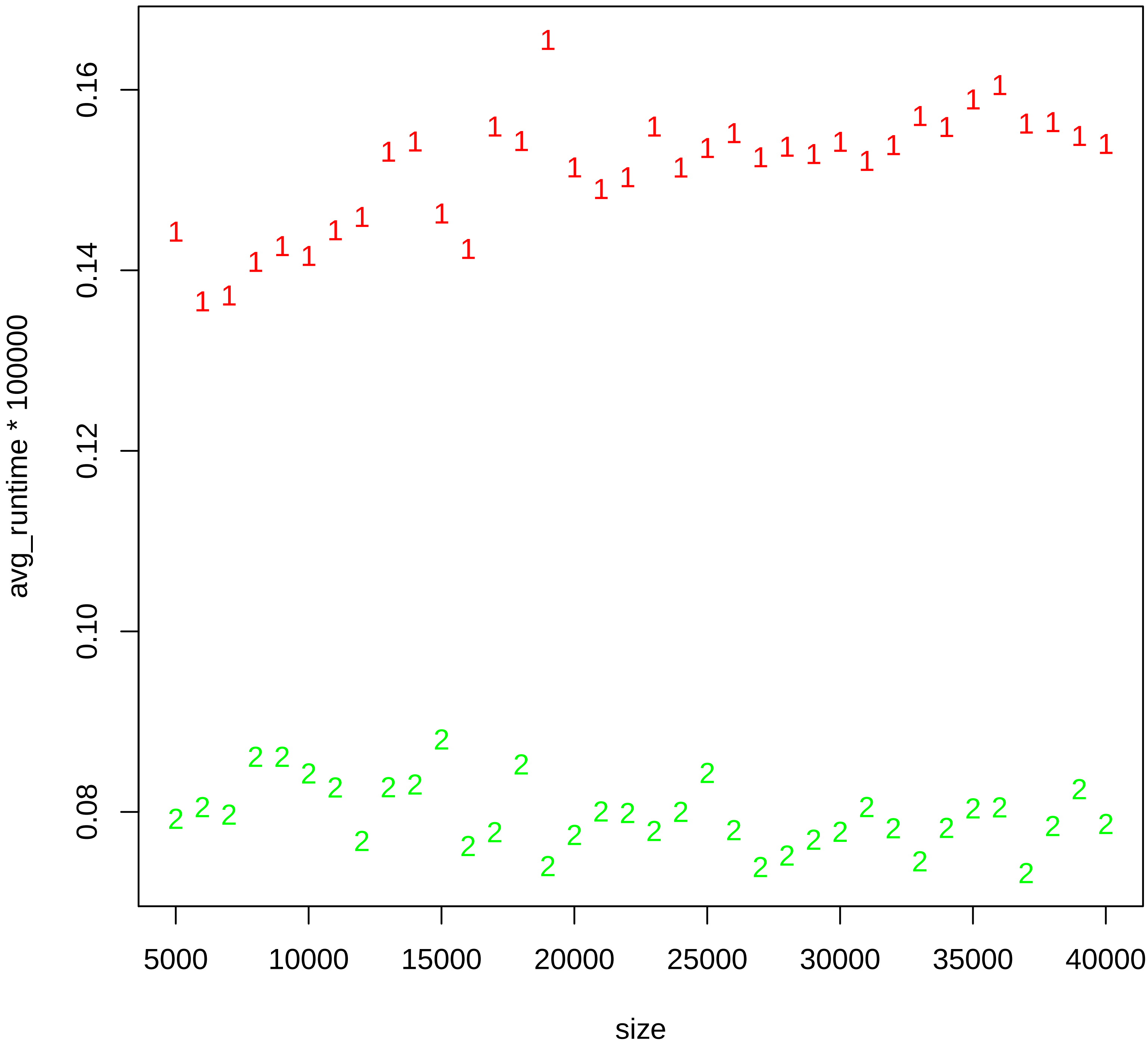
data points from the heap representation. We can see that the heap implementation is actually slower than the list implementation. The reason might be that the insertion point is close to the head of the "list," giving the list implementation an advantage.

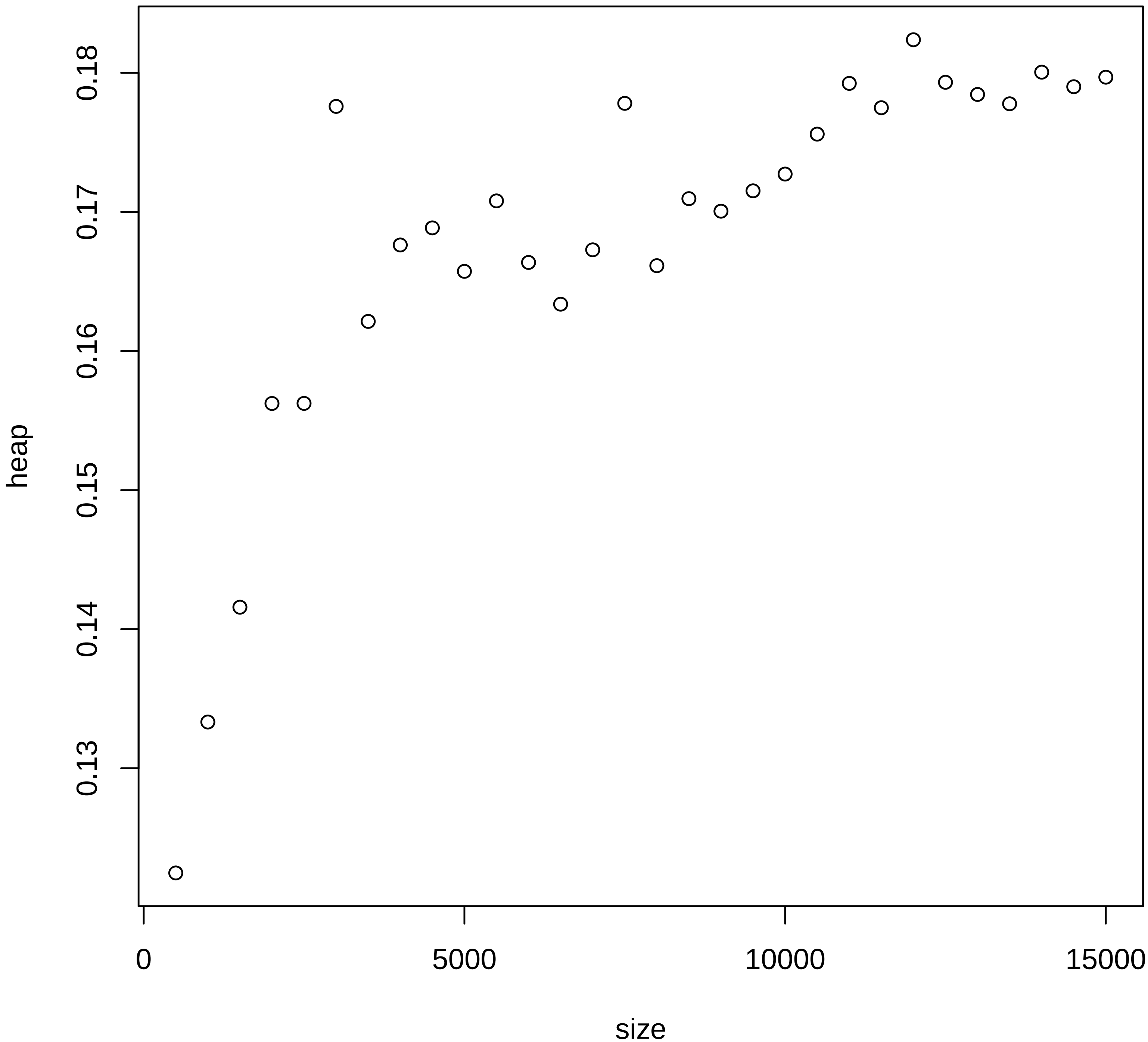
The second experiment is done with initial points whose timestamps are uniformly distributed between 0 and 2

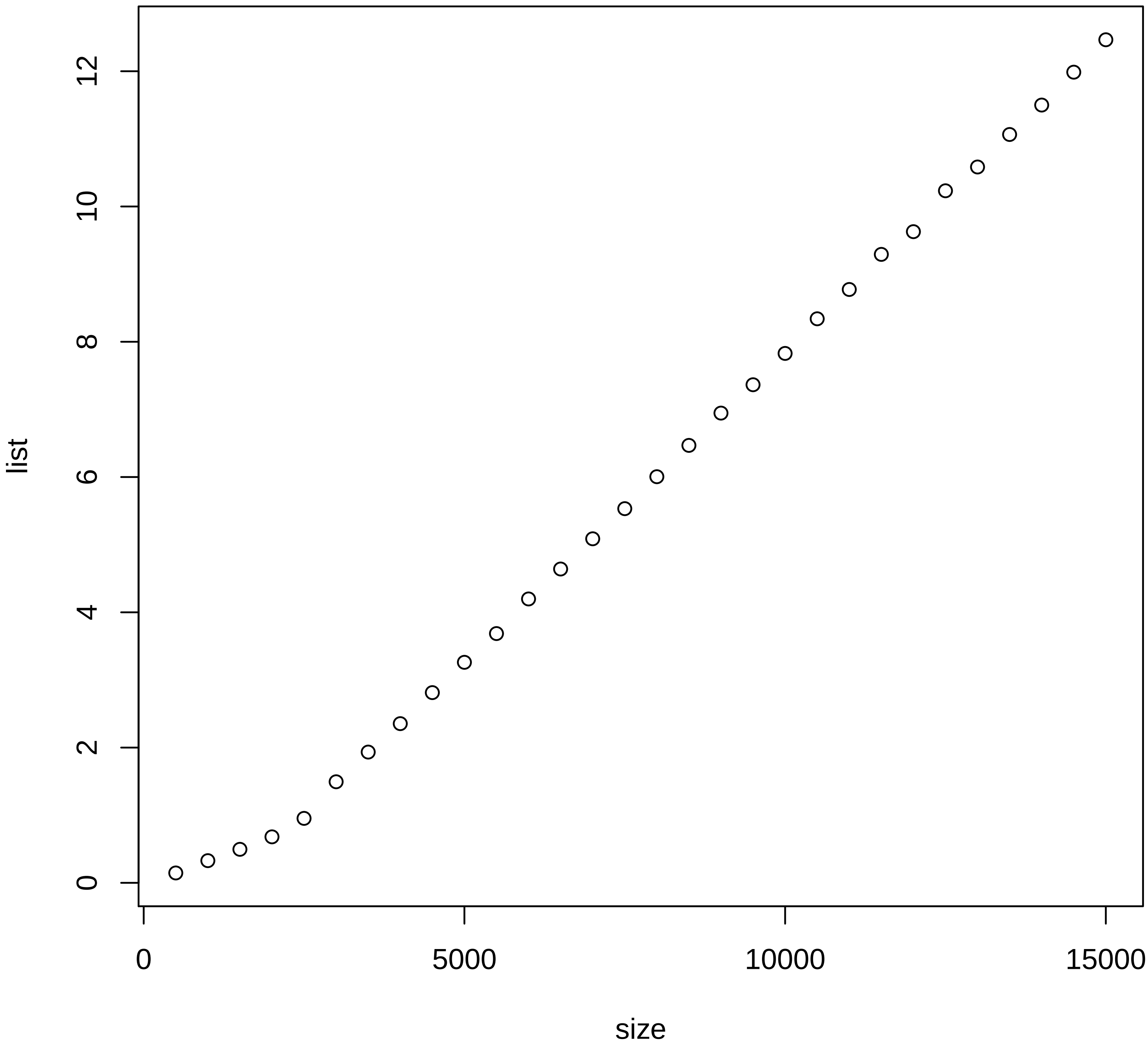
The results are in plots 4 through 6. In the sixth plot, 1 represents data points from the heap implementation. Since the insertion point is far from the head, we can see a log growth in the heap implementation and a linear growth in the list implementation clearly in the plots, giving the heap implementation a huge advantage.


heap
0.140
0.145
0.150
0.155
0.160
0.165
5000
10000
15000
20000
25000
30000
35000
40000
size

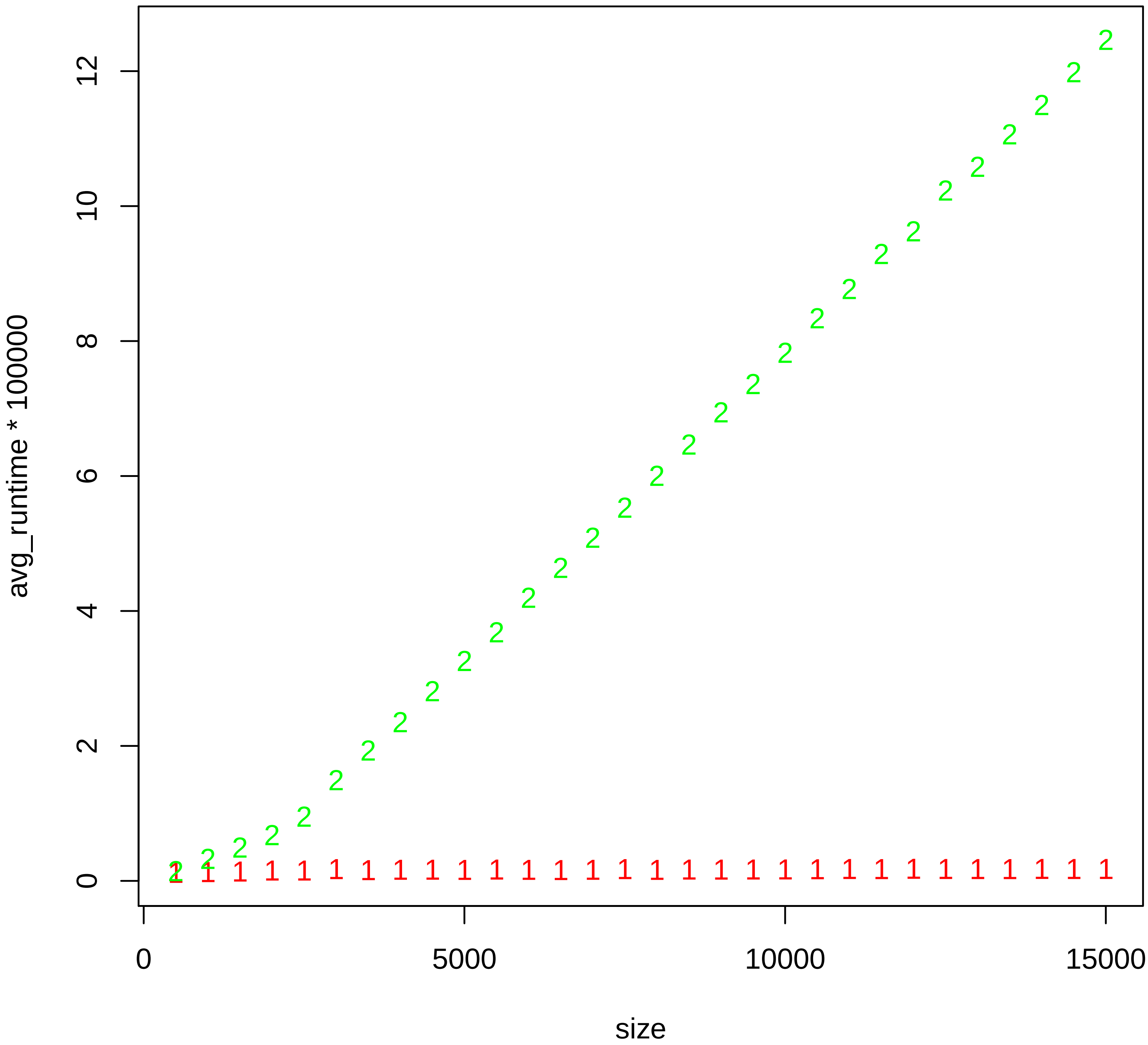

0.075
0.080
0.085
list
5000
10000
15000
20000
25000
30000
35000
40000
size


avg_runtime * 100000
0.08
0.10
0.12
0.14
0.16
5000
10000
15000
20000
25000
30000
35000
40000
size


heap
0.13
0.14
0.15
0.16
0.17
0.18
size
0
5000
10000
15000



list
0
2
4
6
8
10
12
size
0
5000
10000
15000



avg_runtime * 100000
0
2
4
6
8
10
12
0
5000
10000
15000
size

(b)