UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Advanced Data Structures

Computer Science Department

Random Binary Search Trees: An empirical analysis

Author:
Alex Herrero

Professor:
Amalia Duch

March 12, 2025

Contents

List of Figures

List of Tables

1 Introduction

Let T be a binary tree with subtrees T_l and T_r. We say that T is a *binary search tree* (BST) if it is either an empty binary tree or it contains at least one element x as its root such that

- T_l and T_r are also BSTs.
- $\forall y \in T_l, y < x$ and $\forall z \in T_r, z > x$.

Although it is well known that, in the worst case, a BST behaves like a linked list (with the height of the tree being $\Theta(n)$), in this report, we focus on *random BSTs*.

By *random BSTs*, we mean the following: Given a universe of keys U with $|U| = n$, we construct the BST by inserting each element of U exactly once, choosing the insertion order uniformly at random.

2 Analysis of the Average Cost of Insertions

2.1 Theoretical Study

Let us first analyze the expected cost of inserting a new element u into our BST T. For that, we will consider this cost as the cost of searching for u in our BST, which is valid since we can assume that, if u does not exist in T, our search terminates in any empty subtree with identical probability and then we will create a new node right there.

Let I_n be the expected cost of the insertion (expected number of nodes we need to traverse) for a new key u in a random BST of size n. Let, also, be $I_{n,q}$ be the expected cost of the insertion of a key u in a random BST which root is the $q-$th smallest element. Then, the expected cost of I_n is

$$\begin{aligned}
I_n &= \frac{1}{n}\sum_{q=1}^{n} I_{n,q} \\
&= 1 + \frac{1}{n}\sum_{k=1}^{n}(\frac{1}{n}0 + \frac{k-1}{n}I_{k-1} + \frac{n-k}{n}I_{n-k}) \\
&= 1 + \frac{1}{n}\sum_{k=0}^{n-1}(\frac{k}{n}I_k + \frac{n-k-1}{n}I_{n-k-1}) \\
&= 1 + \frac{1}{n^2}\sum_{k=0}^{n-1}(kI_k + (n-k-1)I_{n-k-1}) \\
&= 1 + \frac{2}{n^2}\sum_{k=0}^{n-1} kI_k
\end{aligned}$$

Where we consider a base case of $I_0 = 0$.

We can solve this recurrence using the continuous master theorem. The continuous master theorem solves recurrences of the form

$$F_n = t_n + \sum_{0 \le j < n} w_{n,j} F_j$$

with $t_n = \Theta(n^a (\log n)^b)$. We proceed as follows:

- Determine the values of a and b: Since $t_n = \Theta(1)$, it is straightforward to see that $a = b = 0$.
- Provide a shape function for the weights $w_{n,j}$: We use the following trick to determine the shape function:

$$w(z) = \lim_{n\to\infty} n \cdot w_{n,z\cdot n} = n \cdot \frac{2zn}{n^2} = 2z.$$

- Determine the value of

$$\mathcal{H} = 1 - \int_0^1 w(z) z^a dz.$$

  Substituting the values, we obtain:

$$\mathcal{H} = 1 - \int_0^1 2z dz = 1 - (1 - 0) = 0.$$

- Since $\mathcal{H} = 0$, we need to compute

$$\mathcal{H}' = -(b+1) \int_0^1 w(z) z^a \ln z \, dz.$$

  Substituting the known values,

$$\mathcal{H}' = -1 \int_0^1 2z \ln z \, dz.$$

  This integral can be solved using integration by parts. For the purpose of applying the theorem, we skip the detailed calculation, giving the result:

$$\mathcal{H}' = -(x^2 \ln x - \frac{x^2}{2})\Big|_0^1 = \frac{1}{2}.$$

Since $\mathcal{H} = 0$ and $\mathcal{H}' \neq 0$, we use the result

$$F_n = \frac{t_n}{\mathcal{H}'} \ln n + o(t_n \log n).$$

Substituting the values, we obtain

$$I_n = 2 \ln n + o(\log n).$$

Thus, the expected cost of an insertion into a random binary search tree is bounded by $O(\log n)$. We used the continuous master theorem to solve this recurrence, although it can be solved by solving the *Internal Path Length* recurrence in a more detailed way and, after that, get a more detailed solution of the recurrence. See Appendix A.

2.2 Experimental Study

Once we have theoretical results on the expected cost of an insertion in a random BST, we can provide experimental results to assess how closely they match the theoretical predictions. For this, we will conduct the following experiment (which can be replicated by executing the `execInsertion.sh` script file):

1. We create a random BST of size n by generating n random keys in the interval $[0, 1]$.
2. After constructing the BST, we generate $q = 2 \cdot n$ random numbers in the interval $[0, 1]$.
3. For each generated value, we perform a `find` operation in the BST, counting the number of nodes traversed during the operation.

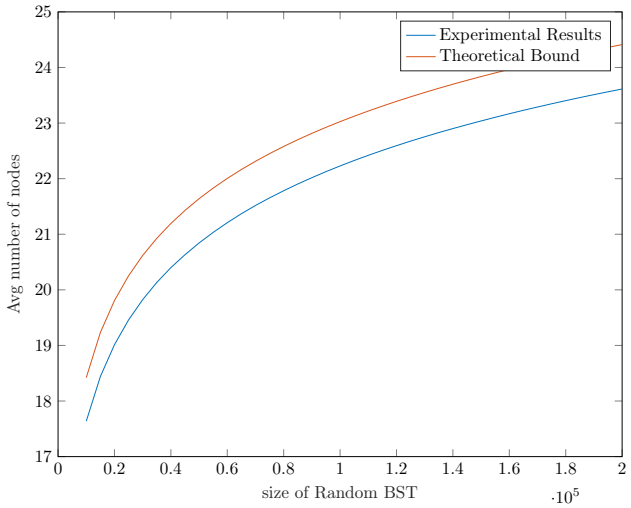


Figure 1: Plot average number of nodes visited respect theoretical bound

4. We sum up the total number of nodes traversed across all q search operations and compute the average.
5. We repeat all previous steps with 20 different seeds and compute the final average.
6. We repeat the entire experiment for different values of n.

We conducted the experiment previously explained with values of n ranging from $10,000$ to $200,000$ in increments of $5,000$, using seeds to generate random numbers from 1989 (in honor of the year of the first publication of the book *Introduction to Algorithms*, which was a great resource for refreshing my knowledge of BSTs and expected cost!) to 2008. Figure 1 provides a plot of the values obtained from this experiment, as well as the theoretical bound derived using the continuous master theorem. What is interesting to look is that the shapes of both plots appear to be the same, suggesting that the only difference between the two functions is a constant factor. Indeed, by applying the continuous master theorem, we know that this difference is $o(\log n)$, but a more detailed analysis, presented in Appendix A, shows that this difference is actually $O(1)$.

This constant factor is clearly visible in Table 1, where the column *diff* (computed as the difference between the theoretical value obtained using Roura's theorem and the experimental value) consistently yields basically a constant difference.

n	Exp.	Theor.	Diff.	n	Exp.	Theor.	Diff.
10000	17.5537	18.4207	0.867	105000	22.2487	23.1234	0.8747
15000	18.3697	19.2316	0.8619	110000	22.3422	23.2165	0.8743
20000	18.9473	19.8070	0.8597	115000	22.4310	23.3054	0.8744
25000	19.3916	20.2533	0.8617	120000	22.5167	23.3905	0.8738
30000	19.7552	20.6179	0.8627	125000	22.5978	23.4721	0.8743
35000	20.0657	20.9262	0.8605	130000	22.6755	23.5506	0.8751
40000	20.3299	21.1933	0.8634	135000	22.7498	23.6261	0.8762
45000	20.5641	21.4288	0.8647	140000	22.8217	23.6988	0.8771
50000	20.7729	21.6396	0.8667	145000	22.8918	23.7690	0.8772
55000	20.9616	21.8302	0.8686	150000	22.9589	23.8368	0.8779
60000	21.1332	22.0042	0.8710	155000	23.0249	23.9024	0.8775
65000	21.2937	22.1643	0.8705	160000	23.0877	23.9659	0.8782
70000	21.4409	22.3125	0.8716	165000	23.1479	24.0274	0.8795
75000	21.5790	22.4505	0.8715	170000	23.2068	24.0871	0.8803
80000	21.7084	22.5796	0.8712	175000	23.2646	24.1451	0.8805
85000	21.8297	22.7008	0.8711	180000	23.3213	24.2014	0.8801
90000	21.9421	22.8151	0.8730	185000	23.3752	24.2562	0.8811
95000	22.0499	22.9233	0.8734	190000	23.4287	24.3096	0.8809
100000	22.1519	23.0259	0.8739	195000	23.4810	24.3615	0.8805
				200000	23.5315	24.4121	0.8807

Table 1: Experimental vs Theoretical results on the expected cost of insertion

3 Analysis of Internal Path Length

3.1 Theoretical Study

Let T be a BST of size n with root r. We define the *Internal Path Length* of T as the sum of all distances between every node of the tree and the root. More formally:

$$\text{IPL}(T) = \sum_{v \in V(T)} d(r, v).$$

This metric should not be surprising, as we can estimate the distance between the root and every node by averaging the IPL of such a tree. Before establishing this relationship, let us first derive a recurrence for the IPL of a tree, solve this recurrence, and later analyze the obtained relation.

Under the assumption of having a random BST, given n keys, we consider every permutation among the $n!$ possible ones to be equally likely. Therefore, the sizes of the subtrees T_l and T_r will be completely random, provided that their combined sizes sum to $n-1$. Consequently, in our recurrence, we must consider every possible size distribution of the subtrees as equally likely.

Moreover, given the IPL of T_l and T_r, since we introduce a new root at the top of the tree, every node will need to traverse one additional edge to reach the new root. This results in adding 1 to the previous distances with respect to their original root. As a consequence, we must add the number of edges in the tree to the sum of the previous IPLs of both subtrees.

Hence, creating the following recurrence:

$$\begin{aligned} IPL_n &= n - 1 + \frac{1}{n} \sum_{k=0}^{n-1} IPL_k + IPL_{n-1-k} \\ &= n - 1 + \frac{1}{n} \sum_{k=0}^{n-1} 2 \cdot IPL_k \\ &= n - 1 + \frac{2}{n} \sum_{k=0}^{n-1} IPL_k \end{aligned}$$

Where we consider a base case of $IPL_0 = 0$.

We can, again, use the continuous master theorem in order to solve this recurrence. Again we need to determinate the following values:

- Determine the values of a and b: Since $t_n = \Theta(n)$, it is straightforward to see that $a = 1$ and $b = 0$.
- Provide a shape function for the weights $w_{n,j}$: We use the following trick to determine the shape function:

$$w(z) = \lim_{n \to \infty} n \cdot w_{n, z \cdot n} = n \cdot \frac{2}{n} = 2.$$

- Determine the value of

$$\mathcal{H} = 1 - \int_0^1 w(z) z^a dz.$$

Substituting the values, we obtain:

$$\mathcal{H} = 1 - \int_0^1 2z dz = 1 - (1 - 0) = 0$$

- Since $\mathcal{H} = 0$ we are in the case

$$\mathcal{H}' = -(b+1)\int_0^1 w(z)z^a \ln z \, dz.$$

Substituting the known values,

$$\mathcal{H}' = -1\int_0^1 2z \ln z \, dz.$$

This integral can be solved using integration by parts. For the purpose of applying the theorem, we skip the detailed calculation, giving the result:

$$\mathcal{H}' = -(x^2 \ln x - \frac{x^2}{2})\Big|_0^1 = \frac{1}{2}.$$

Since $\mathcal{H} = 0$ and $\mathcal{H}' \neq 0$, we use the result

$$F_n = \frac{t_n}{\mathcal{H}'} \ln n + o(t_n \log n).$$

Substituting the values, we obtain

$$I_n = 2(n-1)\ln n + o(n \log n) = 2n \ln n + o(n \log n)$$

Thus, the expected internal path length in a BST is bounded by $O(n \log n)$.

The IPL recurrence, as told before, is a handy recurrence to have an idea of the average distance between every element to the root of the BST (as we can average this path by dividing the IPL by n). With our current analysis we would guess that $I_n = 1 + \frac{IPL_n}{n} = 2 \ln n + o(n)$ but with a more detailed analysis, see Appendix A, we know that this $o(n)$ is, in fact, $O(1)$.

3.2 Experimental Study

Once we have theoretical results on the expected internal path length in a random BST, we can provide experimental results to assess how closely they match the theoretical predictions. For this, we will conduct the following experiment (which can be replicated by executing the `execIPL.sh` script file):

1. We create a random BST of size n by generating n random keys in the interval $[0, 1]$.
2. We calculate the internal path length of the random BST by doing a Breadth-first search algorithm
3. We repeat all previous steps with 20 different seeds and compute the final average.
4. We repeat the entire experiment for different values of n.

Again, we conducted the experiment with same values of n and same *seeds* as subsection 2.2. Figure 2 presents, as before, a plot comparing the expected values obtained with the continuous master theorem as well as experimental results. This time, however, the two curves do not exhibit similar behavior as observed previously, it appears that the theoretical bound increases faster than the experimental result, unlike the previous case. A more numerical perspective is provided in Table 2, where we observe that the difference is no longer constant. In fact, the discrepancy increases as the input size grows.

We know that there is a factor $o(n \log n)$ (in fact, with a more detailed solution of the recurrence, see Appendix A, we know that this $o(n \log n)$ is $O(n)$) that must be taken into account in our study. As a result, even with a high number of nodes in a random BST, this theoretical bound may still be influenced by the $o(n \log n)$ factor until we reach a significantly larger number of nodes. To analyze this effect, we provide Figure 3, where we divide the experimental data by $n \log n$ to observe when this $o(n \log n)$ factor

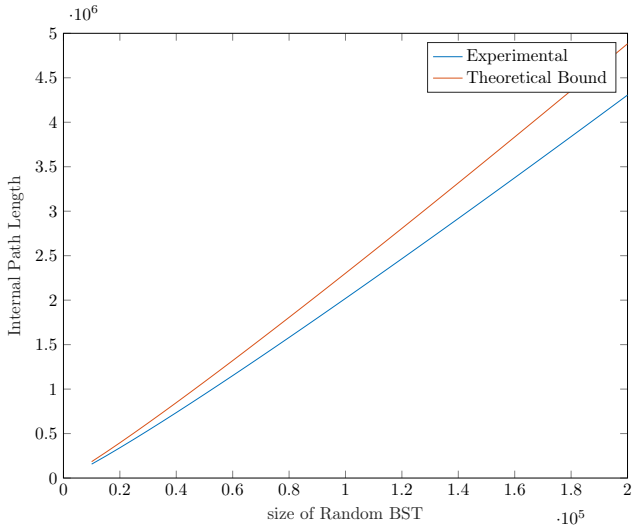


Figure 2: Plot average IPL respect the theoretical bound

stabilizes. As seen in the figure, even for large input sizes (over 200,000 nodes), this factor still plays a role in our analysis.

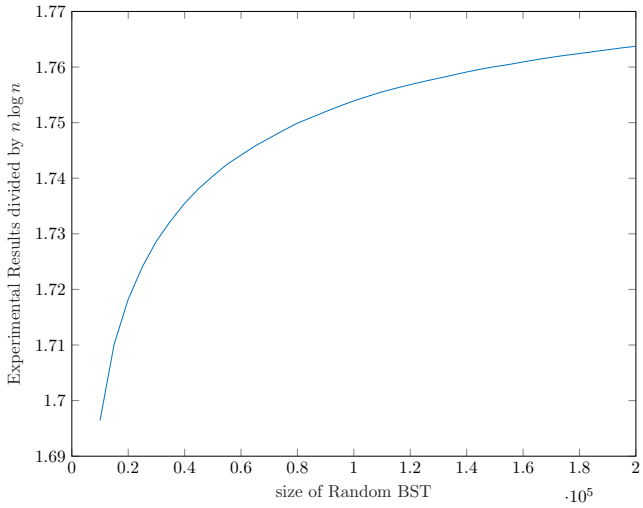


Figure 3: Plot experimental result divided by $n \log n$

So, that $o(n \log n)$ factor must be the reason why it is difficult to predict a more accurate theoretical solution to the IPL, as this factor still plays a significant role in large instances. To support this conjecture we provide in Table 2 a column with the relative error (the difference divided by the theoretical value) that we obtained in our meassure. As we see, even with 200000 nodes we have a relative error of, approx, 12%.

n	Exp.	Theor.	Diff.	Rel. Err	n	Exp.	Theor.	Diff.	Rel. Err
10000	155628.25	184206.81	28578.56	0.1551	105000	2122840.4	2427960.28	305119.88	0.1253
15000	245438.25	288474.16	43035.91	0.1492	110000	2233810.95	2553811.84	320000.89	0.1250
20000	338686.55	396139.50	57452.95	0.1450	115000	2345192.1	2680118.10	334926.00	0.1247
25000	434410.95	506331.56	71920.61	0.1420	120000	2456983.8	2806859.29	349875.49	0.1243
30000	532275.2	618537.16	86261.96	0.1395	125000	2569260.4	2934017.25	364756.85	0.1240
35000	631703.4	732417.23	100713.83	0.1375	130000	2681938.65	3061575.33	379636.68	0.1237
40000	732576.4	847730.78	115154.38	0.1358	135000	2794949.5	3189518.12	394568.62	0.1234
45000	834741.75	964297.60	129555.85	0.1344	140000	2908271.7	3317831.36	409559.66	0.1232
50000	937879.5	1081977.83	144098.33	0.1332	145000	3021837.5	3446501.82	424664.32	0.1230
55000	1042058.1	1200659.73	158601.63	0.1321	150000	3135701.5	3575517.17	439815.67	0.1228
60000	1147139.15	1320251.98	173112.83	0.1311	155000	3249947.05	3704865.92	454918.87	0.1226
65000	1252856.7	1440678.53	187821.83	0.1304	160000	3364578.0	3834537.31	469959.31	0.1223
70000	1359442.4	1561875.07	202432.67	0.1296	165000	3479561.8	3964521.25	484959.45	0.1221
75000	1466711.95	1683786.51	217074.56	0.1289	170000	3594722.3	4094808.26	500085.96	0.1219
80000	1574636.7	1806365.11	231728.41	0.1283	175000	3710163.6	4225389.44	515225.84	0.1218
85000	1683206.65	1929569.11	246362.46	0.1277	180000	3825827.85	4356256.37	530428.52	0.1216
90000	1792332.6	2053361.69	261029.09	0.1271	185000	3941848.05	4487401.11	545553.06	0.1214
95000	1902024.6	2177710.11	275685.51	0.1266	190000	4057893.55	4618816.15	560922.60	0.1213
100000	2012116.75	2302585.09	290468.34	0.1261	195000	4174221.25	4750494.39	576273.14	0.1212
					200000	4290876.65	4882429.06	591552.41	0.1212

Table 2: Experimental vs Theoretical results on the expected Internal Path Length

4 Analysis of Interleaver Insertions and Deletions

Knuth, in Volume 3 of his well-known book *The Art of Computer Programming* [1], mentioned that random insertions and deletions indeed *destroy* the randomness of a BST. Knott [2] noticed this fact during his PhD thesis, where he experimentally showed that Hibbard's algorithm affects the expected cost of BSTs. Eppinger [3] demonstrated experimentally that the path length, although initially decreasing slightly, eventually increases and stabilizes after performing a quadratic number of deletions and insertions. Theoretical results have been accomplished for BSTs of size 3 and 4 [4, 5].

The following experiment will use Hibbard's algorithm. Thus, we aim to support the observations made by Eppinger and Knuth (which can be replicated by executing the `execDelete.sh` script):

1. We create a random BST of size n by generating n random keys in the interval $[0, 1]$.
2. We perform a quadratic number of insertions and deletions as follows: we alternately insert a random element from the interval $[0, 1]$ and delete a random element from the BST.
3. We compute the internal path length of the BST using a Breadth-First Search algorithm.
4. We repeat all the previous steps with 20 different seeds and compute the final average.
5. We repeat the entire experiment for different values of n.

This time, as we are performing a quadratic number of steps, due to memory and time performance considerations, we have decided to conduct this experiment with Random BSTs of size 1000 to 2000 in increments of 50 elements (using the same seeds as in Section 2.2). Figure 4 provides a plot of the final IPL with and without alternating between deletions and insertions. As expected, we observe that the IPL of a tree increases when performing these operations, indicating that we are unbalancing our random BST (hence, destroying its randomness).

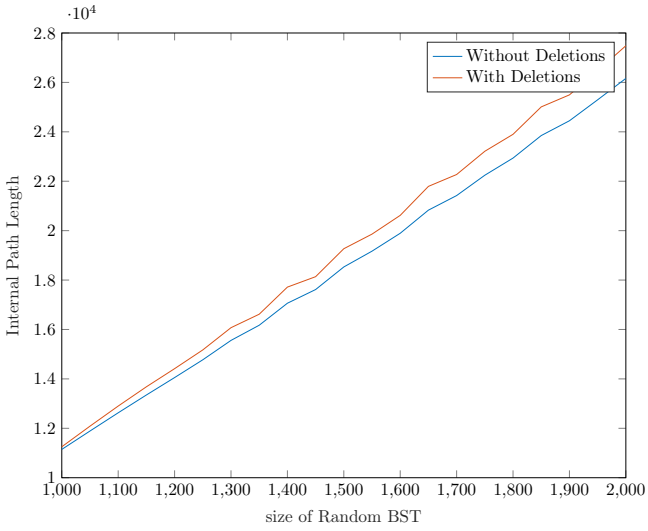


Figure 4: Plot average IPL with and without alternating insertions and deletions

Table 3 provides a more numerical view of such plot. As we see, the difference in IPL gets bigger as we increases the size of n, definitely destroying the initial randomness that we created.

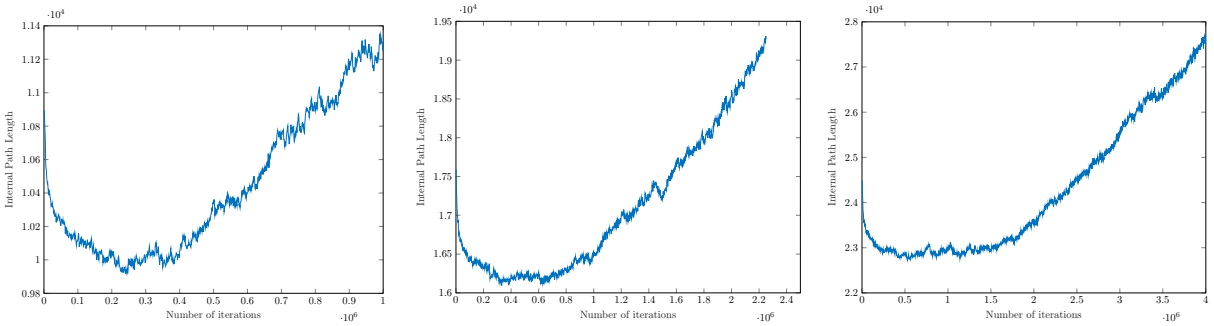
n	No Deletions	Deletions	Difference
1000	11145.85	11252.15	106.3
1050	11890.2	12086.95	196.75
1100	12627.65	12900.5	272.85
1150	13348.65	13680.2	331.55
1200	14053.475	14414.25	360.775
1250	14773.025	15172.7	399.675
1300	15557.95	16072.8	514.85
1350	16171.5	16617.6	446.1
1400	17062.5	17715.25	652.75
1450	17615.975	18136.7	520.725
1500	18530.425	19268.7	738.275
1550	19169.875	19860.2	690.325
1600	19894.3	20618.35	724.05
1650	20830.225	21792	961.775
1700	21419.875	22272.1	852.225
1750	22243.3	23213	969.7
1800	22937.55	23899.25	961.7
1850	23848.45	25009.05	1160.6
1900	24447.525	25495.45	1047.925
1950	25295.475	26473.65	1178.175
2000	26161.125	27483.5	1322.375

Table 3: Difference of IPL after doing deletions

For a more detailed view of the evolution of the IPL in a single BST during these operations we provide the following experiment:

1. We create a random BST of size n by generating n random keys in the interval $[0, 1]$.
2. We perform a quadratic number of insertions and deletions as follows: we alternately insert a random element in the interval $[0, 1]$ and delete a random element from the BST.
3. We compute the internal path length of the BST using a Breadth-First Search algorithm every n iterations.
4. We repeat all the previous steps with 20 different seeds and compute the final average.
5. We repeat the entire experiment for values of $n \in \{1000, 1500, 2000\}$.

Figure 5 provides a plot of such experiment. Indeed, we see what it was said: The IPL decrease slightly at the beginning but, after that, it starts increasing arriving to values greater than the initial ones. Again, indicating that we are unbalancing our tree and losing our randomness.



(a) Evolution of internal path length by doing insertions and deletions with $n = 1000$

(b) Evolution of internal path length by doing insertions and deletions with $n = 1500$

(c) Evolution of internal path length by doing insertions and deletions with $n = 2000$

Figure 5: Evolution of IPLs by doing insertions and deletions

5 Conclusion

Regarding the theoretical analysis of recurrences, although the continuous master theorem is a powerful tool for solving recurrences, it may not always provide a more accurate bound than solving them explicitly.

Regarding the practical part, randomness is a powerful tool for keeping a random BST balanced, but we must be careful in certain situations.

Last but not least, I would like to mention that this last section challenged my intuition: I was expecting that performing random insertions and deletions would not affect the structure of our tree, as every subtree is equally likely. Still, I had a great time glancing over different articles related to this term.

Special thanks to Conrado's slides on Data Structures and Algorithms, which, along with the book *Introduction to Algorithms*, were a great resource for refreshing my knowledge and assisting me when I struggled with certain calculations!

References

[1] D. E. Knuth, *The Art of Computer Programming: Sorting and Searching, volume 3.* Addison-Wesley Professional, 1998.

[2] G. D. Knott, *Deletion in binary storage trees.* Stanford University, 1975.

[3] J. L. Eppinger, "An empirical study of insertion and deletion in binary search trees," *Communications of the ACM*, vol. 26, no. 9, pp. 663–669, 1983.

[4] A. T. Jonassen and D. E. Knuth, "A trivial algorithm whose analysis isn't," *Journal of computer and system sciences*, vol. 16, no. 3, pp. 301–322, 1978.

[5] R. A. Baeza-Yates, "A trivial algorithm whose analysis is not: a continuation," *BIT Numerical Mathematics*, vol. 29, pp. 378–394, 1989.

A More accurate solutions to the previous recurrences

Let us consider the previous recurrence for the IPL of a Random BST:

$$I_n = n - 1 + \frac{2}{n}\sum_{k=0}^{n-1} I_k$$

We need to rewrite this recurrence in terms of just one call instead of multiple calls. Let us express this recurrence in terms of $n+1$:

$$I_{n+1} = n + \frac{2}{n+1}\sum_{k=0}^{n} I_k$$

Now, we need to subtract both recurrences carefully. To do this, let us first multiply each recurrence by n and $n+1$:

$$(n+1)I_{n+1} = (n+1)n + 2\sum_{k=0}^{n} I_k$$
$$nI_n = n^2 - n + 2\sum_{k=0}^{n-1} I_k$$

Now we are ready to subtract both recurrences:

$$\begin{aligned}(n+1)IPL_{n+1} - nIPL_n &= (n+1)n + 2IPL_n - n^2 + n\\ (n+1)IPL_{n+1} &= n^2 + n + 2IPL_n - n^2 + n + nIPL_n\\ &= 2n + 2IPL_n + nIPL_n\\ &= 2n + (2+n)IPL_n\end{aligned}$$

And now we can solve IPL_{n+1}

$$\begin{aligned}IPL_{n+1} &= \frac{2n}{n+1} + \frac{2+n}{n+1}IPL_n\\ &= \frac{2n}{n+1} + \frac{2(2+n)(n-1)}{(n+1)n} + \frac{(2+n)(n+1)}{(n+1)n}IPL_{n-1}\\ &= \frac{2n}{n+1} + \frac{2(2+n)(n-1)}{(n+1)n} + \frac{2+n}{n}\left(\frac{2(n-2)}{n-1} + \frac{n}{n-1}IPL_{n-2}\right)\\ &= \frac{2n}{n+1} + 2(n+2)\sum_{i=1}^{n}\frac{n-i}{(n-i+1)(n-i+2)}\\ &= \frac{2n}{n+1} + 2(n+2)\sum_{i=1}^{n}\frac{i}{(i+1)(i+2)}\\ &= \frac{2n}{n+1} + 2(n+2)\sum_{i=1}^{n}\frac{2}{i+2} - \frac{1}{i+1}\\ &= \frac{2n}{n+1} + 2(n+2)\left(\frac{2}{n+2} + \frac{1}{n+1} + H_n - 2\right)\\ &= 2nH_n - 4n + 4H_n + O(1)\end{aligned}$$

We know that harmonic numbers grow similarly to the natural logarithm, so $H_n = \ln n + O(1)$. Hence,

$IPL_n = 2n \ln n + O(n)$. Using the IPL, we can estimate the average number of nodes required for an insertion by averaging the IPL over the number of nodes. This gives the following estimation:

$$I_n = 1 + \frac{IPL_n}{n} = 2 \ln n + O(1).$$

B Main file implementation

We have created three different main files for each different section. Both will require as an input the size of the BST as well as a seed for generating random numbers. Then a random BST is going to be created and it will perform a single execution of the required experiment. One can execute all experiments by running the `execAll.sh` script file.

B.1 Average Cost of Insertions

The provided `Makefile` can be used to compile this code. To do so, execute the following command in the terminal: `make insertion`. This will compile the main file along with the BST class.

To run the executable, the size and the seed must also be provided via the command line, following the format:

`./insertion n seed`.

Here is the main file provided for this experiment:

```
#include <iostream>
#include <random>
#include "bst.hh"
using namespace std;

int main(int argc, char** argv) {

    unsigned int n = atoi(argv[1]);
    unsigned int seed = atoi(argv[2]);

    BST t;

    mt19937 generator(seed);
    uniform_real_distribution<float> distribution(0.0, 1.0);
    for (unsigned int i = 0; i < n; ++i) t.insert(distribution(generator));


    unsigned int q = 2*n;
    unsigned int tpl = 0;
    for (unsigned int i = 0; i < q; ++i) tpl += t.find(distribution(generator));

    cout << n << "," << float(tpl) / q << endl;
}
```

B.2 Analysis of Internal Path Length

The provided `Makefile` can be used to compile this code. To do so, execute the following command in the terminal: `make ipl`. This will compile the main file along with the BST class.

To run the executable, the size and the seed must also be provided via the command line, following the format:

`./ipl n seed`.

Here is the main file provided for this experiment:

```
#include <iostream>
#include <random>
#include "bst.hh"
using namespace std;

int main(int argc, char** argv) {

    unsigned int n = atoi(argv[1]);
    unsigned int seed = atoi(argv[2]);

    BST t;

    mt19937 generator(seed);
    uniform_real_distribution<float> distribution(0.0, 1.0);
    for (unsigned int i = 0; i < n; ++i) t.insert(distribution(generator));

    cout << n << "," << t.ipl() << endl;
}
```

B.3 Analysis of Interleaved Insertions and Deletions

The provided `Makefile` can be used to compile this code. In this case, we either want to obtain the IPL of the final BST or track instances of the IPL to observe how it evolves during the procedure. To compute the final IPL, compile the code using `make delete`. Otherwise, to track the IPL evolution, compile using `make deleteSingle`.

To execute the program, we also need to specify the size and the seed via the command line, following the format:

`./delete n seed` or `./deleteSingle n seed`.

Here is the main file provided for this experiment:

```
#include <iostream>
#include <random>
#include "bst.hh"
using namespace std;

int main(int argc, char** argv) {

    unsigned int n = atoi(argv[1]);
    unsigned int seed = atoi(argv[2]);

    float *v = (float *) malloc(n*n*sizeof(float));

    BST t;
    unsigned int lastElem = n - 1;

    mt19937 generator(seed);
    uniform_real_distribution<float> distribution(0.0, 1.0);
    for (unsigned int i = 0; i < n; ++i) {
        v[i] = distribution(generator);
```

```
        t.insert(v[i]);
    }

    unsigned int erasedElems = 0;
    srand(seed);
    bool ins = true;
    unsigned int count = 0;
    for (unsigned int i = 0; i < n*n; ++i) {

        if (ins) {
            ++lastElem;
            v[lastElem] = distribution(generator);
            t.insert(v[lastElem]);
        }

        else {
            unsigned int cand = rand() % (lastElem - erasedElems + 1) + erasedElems;
            float aux = v[cand];
            v[cand] = v[erasedElems];
            v[erasedElems] = aux;
            t.erase(aux);
            ++erasedElems;
        }

        ins = !ins;
        ++count;
#ifdef SINGLE
        if (count == n) {
            count = 0;
            cout << i << "," << t.ipl() << endl;
        }
#endif
    }

#ifndef SINGLE
    cout << n << "," << t.ipl() << endl;
#endif
    free(v);
}
```

C Binary Search Tree implementation

A C++ implementation has been developed for efficiency purposes. A C++ class has been created to represent the Binary Search Tree data structure (refer to `bst.hh` for a more detailed exploration of different methods of the class). It consists of a node that stores its value along with two pointers to its respective subtrees. In C++, this can be achieved by using a struct that holds two pointers to its own type.

C.1 Find operation

Classical `find` operation in a BST that will return the number of nodes traversed to arrive to certain key (if it exists).

```
unsigned int BST::find(float x, node* n) {
    if (n == nullptr) return 0;
    if (n -> x == x) return 1;
    if (n -> x > x) return 1 + find(x, n->l);
    else return 1 + find(x, n->r);
}
```

C.2 Insert operation

Classical `insertion` operation in a BST that will return a new BST with the key inserted.

```
BST::node* BST::insert(float x, node* n) {

    if (n == nullptr) {
        node* newNode = new node;
        newNode -> x = x;
        return newNode;
    }

    else if (n -> x == x) return n;

    else {
        if (n -> x > x) n -> l = insert(x, n -> l);
        else n -> r = insert(x, n -> r);
        return n;
    }
}
```

C.3 Erase Operation

We implemented Hibbard's algorithm for `deletion` operations.

```
BST::node* BST::erase(node* n, float x) {

    if (n == nullptr) return nullptr;

    if (n -> x < x) {
        n -> r = erase(n -> r, x);
        return n;
    }
```

```
    if (n -> x > x) {
        n -> l = erase(n -> l, x);
        return n;
    }

    if (n -> l == nullptr) {
        node* aux = n -> r;
        delete n;
        return aux;
    }

    if (n -> r == nullptr) {
        node* aux = n -> l;
        delete n;
        return aux;
    }

    float val = 0;
    n -> r  = eraseSuccessor(n -> r, val);
    n -> x = val;
    return n;
}

BST::node* BST::eraseSuccessor(node* n, float& x) {

    if (n -> l == nullptr) {
        node* aux = n -> r;
        x = n -> x;
        delete n;
        return aux;
    }

    else {
        n -> l = eraseSuccessor(n -> l, x);
        return n;
    }
}
```

C.4 Internal Path Length calculation

```
unsigned int BST::ipl() {
    queue<pair<node*,unsigned int>> q;
    unsigned int iplLocal = 0;
    q.push({root,0});

    while (not q.empty()) {
        pair<node*, unsigned int> p = q.front();
        q.pop();

        if (p.first != nullptr) {
            unsigned int d = p.second;
            iplLocal += d;
            q.push({p.first -> l, d+1});
            q.push({p.first -> r, d+1});
        }
    }

    return iplLocal;
}
```