CSE 332: Data Structures and Parallelism

Section 5: Hashing, Sorting, and Bad Experiments Solutions

0. Hash... Browns?

For the following scenarios, insert the following elements in this order: 7, 9, 48, 8, 37, 57. For each table, TableSize = 10, and you should use the primary hash function $h(k) = k$. If an item cannot be inserted into the table, please indicate this and continue inserting the remaining values.

(a) Linear Probing

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Solution:

0	8
1	37
2	57
3	
4	
5	
6	
7	7
8	48
9	9

(b) Quadratic Probing

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Solution:

0	
1	37
2	8
3	
4	
5	
6	57
7	7
8	48
9	9

(c) Separate chaining hash table - Use a linked list for each bucket. Order elements within buckets in any way you wish.

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Solution:

0	
1	
2	
3	
4	
5	
6	
7	57 $\rightarrow$37 $\rightarrow$7
8	8 $\rightarrow$48
9	9

1. Double Double Toil and Trouble

(a) Describe double hashing.

Solution:

The first hash function determines the original location where we should try to place the item. If there is a collision, then the second hash function is used to determine the probing step distance as 1*h2(key), 2*h2(key), 3*h2(key) etc. away from the original location.

(b) List 2 cons of quadratic probing and describe how one of those is fixed by using double hashing.

Solution:

In quadratic probing, 1) if the table is more than half full (load factor = 0.5) then you are not guaranteed to be able to find a location to place the item, 2) suffers from secondary clustering (items that initially hash to the same location resolve the collision identically).

Assuming a good second hash function is used, double hashing does not suffer from 1). Assuming a good second hash function is used, double hashing avoids secondary clustering because items that initially hash to the same location resolve the collision differently, which decreases the likelihood that two elements will hash to the same index after initial collision.

2. Sorting Hat

Suppose we sort an array of numbers, but it turns out every element of the array is the same, e.g.,{17, 17, 17, ..., 17}. (So, in hindsight, the sorting is useless.)

(a) What is the asymptotic running time of insertion sort in this case?

Solution:

O(n) - This is the best case runtime of insertion sort as it only requires one pass through the data. Insertion sort will traverse the array but since each element is not less than the one before it, no extra computations are necessary.

(b) What is the asymptotic running time of selection sort in this case?

Solution:

O(n^2) - Selection sort always has n^2 runtime, regardless of the nature of data

(c) What is the asymptotic running time of merge sort in this case?

Solution:

$O(n * log(n))$ - Merge sort always has $n * log(n)$ runtime, regardless of the nature of data

(d) What is the asymptotic running time of quick sort in this case?

Solution:

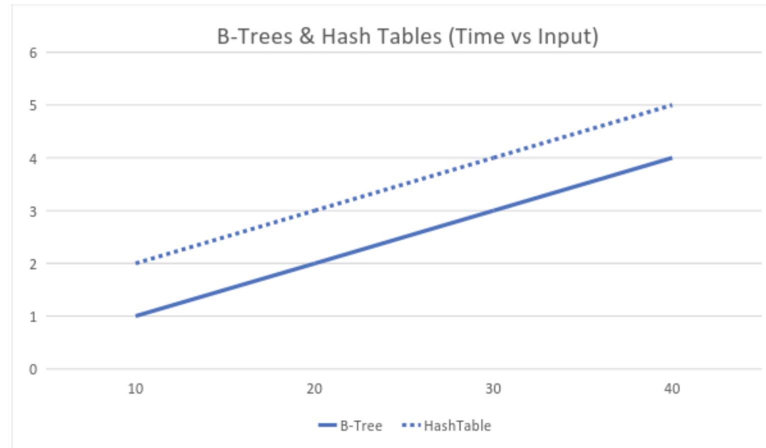
O(n^2) - This is the worst case runtime of quick sort. When partitioning, every element is going to fall to the same side of the pivot since they all have the same value which essentially only sorts 1 element per iteration of quicksort, leading to the n^2 runtime.

3. Bad Experiments

B-Trees vs HashTables

Construct input files for B-Trees and HashTables to demonstrate that a B-Tree is asymptotically better than a HashTable. To do this, we expect you to show trends. You might consider fitting a curve to your results. Explain your intuition on why your results are what they are.

(a) Bad Answer #1



As you can see here, the B-Tree clearly has better run time at many different inputs. The graphs show that both run times increase linearly, but the B-Tree always has a slightly lower run time, so its definitely the better data structure. All of our inputs are in the experiments file. We used different types of values: integers, strings and objects containing (x,y) coordinate pairs. We thought it was very interesting that both run times looked pretty linearly, we would have expected more differences, but data is data! For practical use, we dont see that there would be much of a difference in using either one of these since HashTables run time is only slightly worse than B-Trees.

What's wrong with this answer?

Solution:

(a) The question asks the student to show that one is asymptotically better than the other, but the data collected shows linear graphs, meaning these would both be $O(n)$. Likely, bad data sets were chosen that did not demonstrate the differences between the data structures. For this example, youd need to use very large keys/values and/or very many of them to make the B-Tree advantageous (the whole point is bringing in whole pages from disk, inserting 10 ints takes up very few bytes and wont require any paging to disk). They should have used data that would have allowed the B-Tree to be properly utilized.

(b) There are no deep conclusions drawn, they just summarize the data thats already shown in the graph. The closest thing to a conclusion is the last sentence, but its a very superficial evaluation of the graph above and doesnt show any deeper thought or consideration. Also, it goes against the thing they were asked to show, which implies they probably gathered faulty data or used faulty test cases (see 1 above).

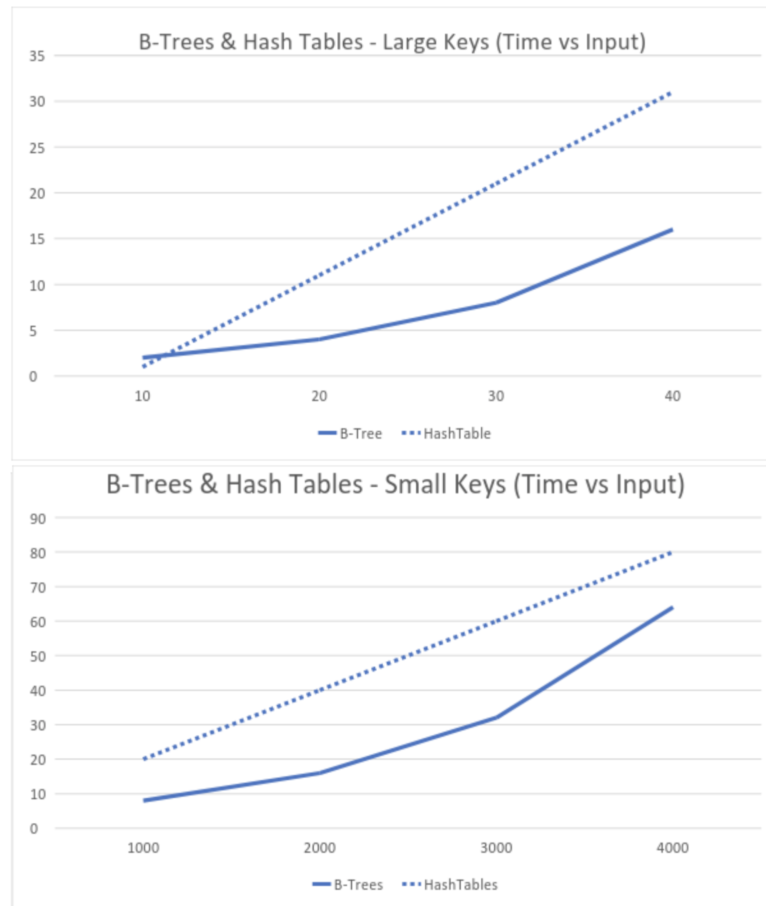
(c) They say they used many types of input, but they dont explain how these various types of input relate to the graph (is this an average of all inputs? Or is it just one of the inputs? What are they showing here?)

(d) Graph has no labels and no units.

(e) The graph does not clearly show the data points the trendlines is approximating. This is an issue because then we have no idea how many data points they gathered and also is generally wrong

because their data is not continuous. Trend lines are obviously fine, but these graphs dont give an honest representation of the data.

(b) Bad Answer #2



Our B-Tree did have better run time, as shown in the graph above. We think this is due to the fact that the B-Tree is paging to disk efficiently, but the HashTable is having to page to disk far more often. We tried two different kinds of input, one with very large keys with a lower M and L, and another with smaller keys and a larger M and L. We found that in both cases, the B-Tree performed better. This surprised us as HashTables usually work quite well with small values, but clearly if there are enough of them, having to page to disk multiple times really does make a difference. We did try with small numbers of small sized input as well, and as expected the HashTable performed better (we did not include this graph, we just did it out of curiosity). This made sense to us as the whole advantage of a B-Tree is more efficient disk look-ups, so the overhead in the B-Tree makes it perform worse than a HashTable with small inputs. Up to this point we had wondered why we dont hear more about B-Trees as they seem like such a better data structure to use, but now we see that there are only specific cases where it is really advantageous to use a B-Tree. (We know you said this in class, but now weve seen it in the data!) We think this sort of data structure would be good for things like large file systems on servers, or perhaps an application like GoogleDocs where a company is storing a lot of files within a lot of nested directories.

What's wrong with this answer?

Solution:

(a) Its subtle, the conclusions and discussion are good, BUT the data they gathered totally conflicts with all of their conclusions! The B-Tree graph looks like its going to be $O(n^2)$ but the HashTables

graph looks $O(n)$. This makes it seem as if they just ran experiments to produce data and then just wrote what they thought the grader wanted to hear rather than actually using the data to draw conclusions. (keep in mind this isnt a real question and Im just making stuff up, so dont take any of this as actual relations between B-Trees and HashTables!)