CSE 332: Data Structures and Parallelism

Section 5: Hashing and Sorting 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.