Kyler Little

CptS 223 Homework #3 - Heaps, Hashing, Sorting

Please complete the homework problems on the following page using a separate piece of paper. Note that this is an individual assignment and all work must be your own. Be sure to show your work when appropriate. Please scan the assignment and upload the PDF to Git, but also bring a printed out copy for the grading TAs. We've found that many of the scans are difficult to read and notate.

1. [6] Starting with an empty hash table with a fixed size of 11, insert the following keys in order into four distinct hash tables (one for each collision mechanism): {12, 9, 1, 0, 42, 98, 70, 3}. You are only required to show the final result of each hash table. In the **very likely** event that a collision resolution mechanism is unable to successfully resolve, simply record the state of the last successful insert and note that collision resolution failed. For each hashtable type, compute the hash as follows:

```
hashkey(key) = (key * key + 3) % 11
```

Separate Chaining (buckets)

0	1	2	3	4	5	6	7	8	9	10
	3		∅	12 → 1 → 98			9 → 42	70		

To probe, start at i = hashkey and do i++ if collisions continue

Linear Probing: probe(i') = (i + 1) % TableSize

0	1	2	3	4	5	6	7	8	9	10
		3		∅	12	1	98	9	42	70

Quadratic Probing: probe(i') = (i * i + 5) % TableSize

probe(hashkey + 6)

0	1	2	3	4	5	6	7	8	9	10
			9		98	42		1	∅	12

probe(hashkey+1)
probe(hashkey + 7)
probe(hashkey + 3)

70 unable to be inserted... Tried i ∈ [1, 12] & no success

2. [3] For implementing a hash table. Which of these would probably be the best initial table size to pick?

Table Sizes:

1 ✗ 100 ✗ 101 15 ✗ 500 ✗

Why? For every hashing function, you must mod by the table size. In picking a prime number, you significantly reduce the odds of collision because the prime number will not share any factors with the value trying to be inserted (unless of course it's a multiple)

3. [4] For our running hash table, you'll need to decide if you need to rehash. You just inserted a new item into the table, bringing your data count up to 53491 entries. The table's vector is currently sized at 106963 buckets.

- Calculate the load factor (λ):

$$\lambda = \frac{53491}{106963} = 0.50001$$

- Given a linear probing collision function should we rehash? Why?

Yes. Once $\lambda > 0.5$ for probing, we rehash. Otherwise, the behavior of inserts & searches begins to become $O(N)$. I.e. the probability is higher that we have to probe for a while to find an empty bucket.

- Given a separate chaining collision function should we rehash? Why?

No. We wait until $\lambda > 1.0$. Behavior is still $O(1)$ at this point if the hash table was implemented well.

4. [4] What is the Big-O of these actions for a well designed and properly loaded hash table with N elements?

Function	Big-O complexity
Insert(x)	$O(1)$
Rehash()	$O(N)$
Remove(x)	$O(1)$
Contains(x)	$O(1)$

5. [3] If your hash table is made in C++11 with a vector for the table, has integers for the keys, uses linear probing for collision resolution and only holds strings... would we need to implement the Big Five for our class? Why or why not?

Probably not... Copy & move semantics are already defined for STL data types, and we probably don't need a destructor unless we're using heap memory.

6. [6] Enter a reasonable hash function to calculate a hash key for these function prototypes:

```
int hashit( int key, int TS )
{
    return key % TS;
}
```

```
int hashit( string key, int TS )    //From p.195 of book
{
    int hashVal = 0;
    for (char ch : key)
        hashVal = 37 * hashVal + ch;
    return hashVal % TS;
}
```

7. [3] I grabbed some code from the Internet for my linear probing based hash table because the Internet's always right. The hash table works, but once I put more than a few thousand entries, the whole thing starts to slow down. Searches, inserts, and contains calls start taking *way* longer than O(1) time and my boss is pissed because it's slowing down the whole application services backend I'm in charge of. I think the bug is in my rehash code, but I'm not sure where. Any ideas why my hash table starts to suck as it grows bigger?

```
/**
* Rehashing for linear probing hash table.
*/

void rehash( )
{
        vector<HashEntry> oldArray = array;

        // Create new double-sized, empty table
        array.resize( 2 * oldArray.size( ) );
        for( auto & entry : array )
                entry.info = EMPTY;

        // Copy table over
        currentSize = 0;
        for( auto & entry : oldArray )
                if( entry.info == ACTIVE )
                        insert( std::move( entry.element ) );
}
```

2 * oldArray.size() is NOT a prime number... it has 2 as a factor. Find the next prime AFTER this number.

8. [4] Time for some heaping fun! What's the time complexity for these functions in a binary heap of size N?

Function	Big-O complexity
insert(x)	O(log N)
findMin()	O(1)
deleteMin()	O(1)
buildHeap(vector<int>{1...N})	O(N)

9. [4] What would a good application be for a priority queue (a binary heap)? Describe it in at least a paragraph of why it's a good choice for your example situation.

I'm designing my own operating system. In my kernel, I use a priority queue to schedule processes. Because I want my OS to be responsive, I always want the tasks which are very short (like mouse clicking /movement) to be completed first. Therefore, I assign tasks like that with the highest priority. In this way, tasks are still completed sequentially, but w/ flexibility based on status.

(P.S. I'm not actually designing my own OS... not in 460 yet)

10. [4] For an entry in our heap (root @ index 1) located at position i, where are it's parent and children?

Parent: $i/2$

Children: left : $2i$ right : $2i+1$

binary heap

What if it's a d-heap?

Parent: i/d

Children: $di, di+1, \dots, di+(d-1)$

11. [6] Show the result of inserting 10, 12, 1, 14, 6, 5, 15, 3, and 11, one at a time, into an initially empty binary heap. Use a 1-based array like the book does. After insert(10):

0	1	2	3	4	5	6	7	8	9	10
	10									

After insert (12):

	10	12								

etc:

	1	12	10							

	1	12	10	14						

	1	6	10	14	12					

	1	6	5	14	12	10				

	1	6	5	14	12	10	15			

	1	3	5	6	12	10	15	14		

	1	3	5	6	12	10	15	14	11	

12. [4] Show the same result (only the final result) of calling buildHeap() on the same vector of values: {10, 12, 1, 14, 6, 5, 15, 3, 11}

	1	2	3	4	5	6	7	8	9	
	1	3	5	11	6	10	15	14	12	

6 15

14 12

1 3 10

5

11

good!

15 12 14

10

6 11

13. [4] Now show the result of three successive deleteMin operations from the prior heap:

	3	6	5	11	12	10	15	14		

	5	6	10	11	12	14	15			

	6	11	10	15	12	14				

14. [4] What are the average complexities and the stability of these sorting algorithms:

Algorithm	Average complexity	Stable (yes/no)?
Bubble Sort	N^2	yes
Insertion Sort	N^2	yes
Heap sort	$N \log N$	no
Merge Sort	$N \log N$	yes
Radix sort	kN	yes
Quicksort	$N \log N$	no

k is length of longest val (i.e. number of passes)

15. [2] What are the key differences between Mergesort and Quicksort? How does this influence why languages choose one over the other?

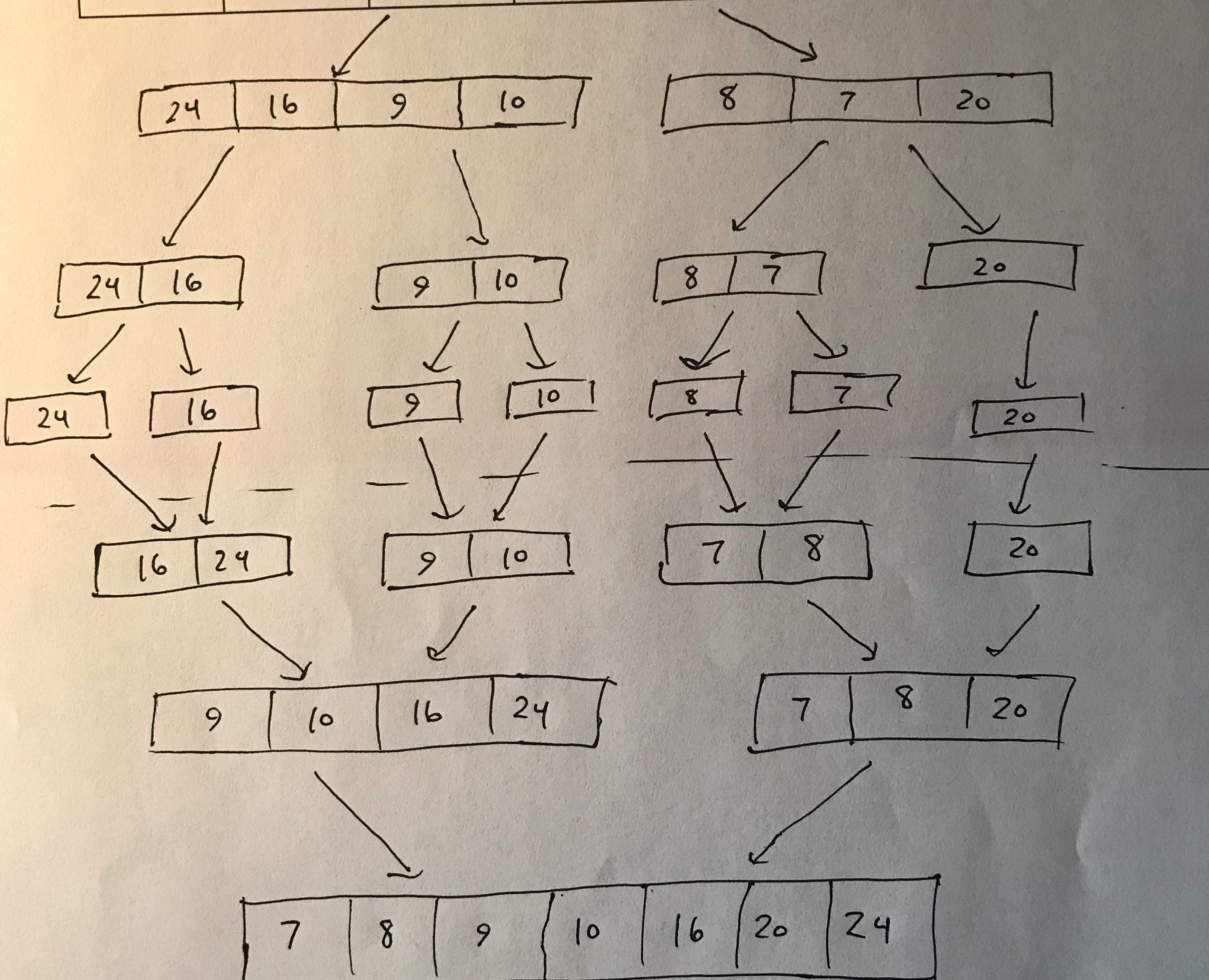
- Quicksort has a worse "worst case" — $O(N^2)$
- Merge sort is stable while QS is not
- merge sort is more dependent on how fast comparing/moving is...

$\Longrightarrow$ in Java, where moves are fast, merge sort is used by default since MS requires more moves than QS

$\Longrightarrow$ in C++, comparing is fast, so QS is default since it requires more comparisons

16. [4] Draw out how Mergesort would sort this list:

24	16	9	10	8	7	20



17. [4] Draw how Quicksort would sort this list:

24	16	9	10	8	7	20