Name: ______________________________

UW NetID: ______________________________

CSE 332 Summer 2019: Final Exam (pt. 1)

(closed book, closed notes, no calculators)

Instructions: Read the directions for each question carefully. We can only give partial credit based on the work you write down, so show your work. If you are running out of time, take note of which questions are weighted more and budget your time accordingly.

For questions where you are drawing pictures, please circle your final answer.

Good Luck!

Total: 66 points. Time: 60 minutes.

Question	**Max Points**
1 Clustering	7
2 Hashing	10
3 Best Sorts	12
4 Parallel Quicksort	11
5 Parallel Implementation	16
6 B-Trees	10
Total	66

1 Clustering [7 points]

Rob plans to "shake up" the hash table community with a "radical " new open addressing probing technique he calls "alternating probing." In this probing technique, the progression is as follows

- Probe 0: key % tableSize
- Probe 1: key % tableSize + 1
- Probe 2: key % tableSize - 1
- Probe 3: key % tableSize + 2
- Probe 4: key % tableSize - 2

For the ith probe, the location will be: key + $\begin{cases} -\frac{i}{2} & \text{if } i \text{ is even} \\ \lfloor \frac{i}{2} \rfloor & \text{if } i \text{ is odd} \end{cases}$ % tableSize

1. When put into practice, will this probing technique cause primary clustering? Briefly explain why.

   Solution: This will cause primary clustering, as the probing method will cause new elements to be added adjacent to each other in the table.

2. Will this probing technique show secondary clustering? Briefly explain why.

   Solution: This will cause secondary clustering as well, since each distinct element that hashes to the same location will follow the same probing path.

3. Would you expect worst-case hash table performance with alternating probing to be better, worse, or the same as using linear probing? Clearly state the expected worst-case time for a successful find with alternating probing. Briefly justify your answers.

   Solution: I would not expect this method to perform differently from linear probing, as it shows the same cluster patterns and is just as quick to calculate. The worst-case run time for a successful find would still be $O(n)$ just like with linear probing, with n as the number of elements in the table.

2 Hashing [10 points]

1. Insert the following values into the hash table below, using open addressing quadratic probing to handle potential collisions. Use the values themselves as the hash value. In the case quadratic probing doesn't find an open cell, write the value below the table.

$$5, 15, 24, 25, 9, 55$$

0	1	2	3	4	5	6	7	8	9
9	55			24	5	15			25

2. Up to what load factor is quadratic probing guaranteed to find an open location in a table with a prime number length?

Solution: $\lambda = 1/2$

3. Re-hash the table above with the same values into the new table below, but now use a secondary hash to resolve collisions via the double hashing technique we discussed in class. The secondary hash function is provided below. In the case double hashing probing doesn't find an open cell, write the value below the table.

index = (h(key) + i*g(key)) % tableSize
g(key) = 1 + (key/tableSize) % (tableSize - 1)

0	1	2	3	4	5	6	7	8	9
	55			24	5		15	25	9

Solution: We also accepted answers that hashed in the order of the elements in the table above.

4. Write a secondary hash function that would cause double hashing to act exactly like linear probing.

Solution: g(key) = 1

3 Best Sorts [12 points]

For each of the following situations below pick which sort would be the best option. Assume all input elements are comparable. Justify your answers in terms of expected run time, memory use, or other sorting method properties.

1. You have a large amount of uniformly random data where each element is an object and you have very little spare memory.

   Solution: QuickSort/Heapsort. Quicksort will likely run in the best-case asymptotic time with random data and both are in-place. $O(n \log n)$. Non-comparative sorts will not work with most objects either.

2. You have a large array of uniformly random two-digit integers.

   Solution: Bucket/Radix Sort. 100 possible values is small enough for non-comparative sort, and it will run in linear time.

3. You have a large amount of data where each element is an object and you want the sort to be stable.

   Solution: Merge Sort. Objects are not usually suitable for non-comparative sorts and quicksort/heapsort are not stable.

4. You have a large array of objects that has been pre-sorted with only one element out of place.

   Solution: Insertion Sort. Run in near-linear time for mostly-sorted input. Objects preclude the non-comparative sorts.

4 Parallel Quicksort [11 points]

1. Write the recurrence relation for best-case (i.e. balanced), sequential quicksort (you do not have to solve it into closed form).

$$T(n) = \begin{cases} 2T(n/2) + c_1 n & \text{if } n \geq c_3 \\ c_2 & \text{otherwise} \end{cases}$$

2. Now write the recurrence relation for the span of best-case (i.e. balanced), **parallel** quicksort (again, you do not have to solve this).

$$T_\infty(n) = \begin{cases} T_\infty(n/2) + c_1 \log n & \text{if } n \geq c_3 \\ c_2 & \text{otherwise} \end{cases}$$

3. What changes did we make to parallel quicksort to change it from its original recurrence to the span recurrence? Explain how the algorithm design was altered to create these changes.

Solution: We changed the recursive call to a parallel one. We also performed partition in parallel with two filter operations for the values less than and greater than the pivot.

4. What are the simplified, closed terms for the work and span of parallel quicksort? Using those, what is the parallelism of best-case quicksort?

Solution: Work: $T_1 = O(n \log n)$, Span: $T_\infty = O(log^2 n)$
Parallelism: $\frac{T_1}{T_\infty} = \frac{O(n \log n)}{O(\log^2 n)} = O(\frac{n}{\log n})$

5 Parallel Implementation [16 points]

Like with quick and merge sort in lecture, now we're going to implement parallel bucket sort. This will require you to recall knowledge of sorting techniques, parallelism and concurrency and mix them effectively. In this section you will write Java code. **You will not be graded on your java syntax as long as it is clear what you are trying to do.**

1. As a warm-up, first write code for a sequential version of bucket sort. Use the variable names provided in the function declaration. `lo` and `hi` define a range in the input array to perform the sort on (this will be useful in the next section). Assume `out[]` has been initialized with all `0`s and has the necessary number of buckets. You do not have to create a final sorted output format for this method, just increment the corresponding bucket when you see an instance of a value.

```
void bucketSort(int[] input, int[] out, int lo, int hi) {

    for (int i = lo; i < hi; i++) {}
        out[input[i]] = out[input[i]] + 1;
    }

}
```

2. What is the worst-case asymptotic run-time for general bucket sort with input size n and m buckets?

Solution: $O(n + m)$

3. Now create a parallel version of bucket sort by writing in code for a thread class below. You may call your previous sequential method if you want. Your design should have $O(\log(n))$ span ignoring issues related to concurrent access. Do not worry about concurrency for this question.

```
Class BucketThread extends RecursiveAction {
    int hi; int lo; int[] input; int[] out; int cutoff;
    public BucketThread(int[] i, int l, int h, int[] o, int c){
        input = i;
        lo = l;
        hi = h;
        out = o;
        cutoff = c;
    }

    public void compute() {
        if (hi-lo <= cutoff) {
        //sequential case here

            bucketSort(input, out, lo,hi);

        } else {
        //create and fork thread(s) here

            BucketThread left = new BucketThread(input,lo,(hi+lo)/2,out,cutoff)
            BucketThread right = new BucketThread(input,(hi+lo)/2,hi,out,cutoff
            left.fork();
            right.compute();
            left.join();

        }
    }
}
```

4. Now consider concurrent access. Why might there be a race condition in the code you just wrote? Provide an example interleaving of code called from two instances of your BucketThread class that show this condition and explain what is happening in it. Remember that more than operation can occur on a single line. You can use temporary variables in that case, if needed.

Solution: If multiple threads try to increment the bucket at the same time we may have a race condition where one write gets lost. e.g.

```
out[input[i]] + 1 (thread 1)
out[input[i]] + 1 (thread 2)
out[input[i]] = <thread 1 value> (thread 1)
out[input[i]] = <thread 2 value> (thread 2)
```

5. Consider two options for adding locks to your thread class:

   - Option 1: Lock the whole `out` array during any reads/writes to it.
   - Option 2: Lock only the single bucket (one cell of the `out` array) your sorting algorithm is trying to read/write to.

   In a sentence or two, explain if each of these approaches will eliminate data races or not.

Solution: Option 1: This will work, as only one thread can read/write to any location in the bucket array at one time.
Option 2: This will also work, as only one thread can read/write to each particular location in the bucket array at one time.

6. Which of the two locking options in the previous question do you think will result in better performance? Briefly explain why, and be clear how you are defining performance.

Solution: Option 2 will work faster, as multiple threads can write to the bucket array as long as they are not working at the same index.

6 B-Trees [10 points]

1. You are shown a B-Tree with a $M = 20$ maximum internal node branching factor. Assuming your block size is 175 bytes, and a pointer takes up 4 bytes, what is the largest size (in bytes) a single key could be for this B-Tree and still allow a node to fit on one block?

   Solution: $175 = 20 * 4 + (19) * key$
   $key = \frac{175-20*4}{19} = 5$ bytes

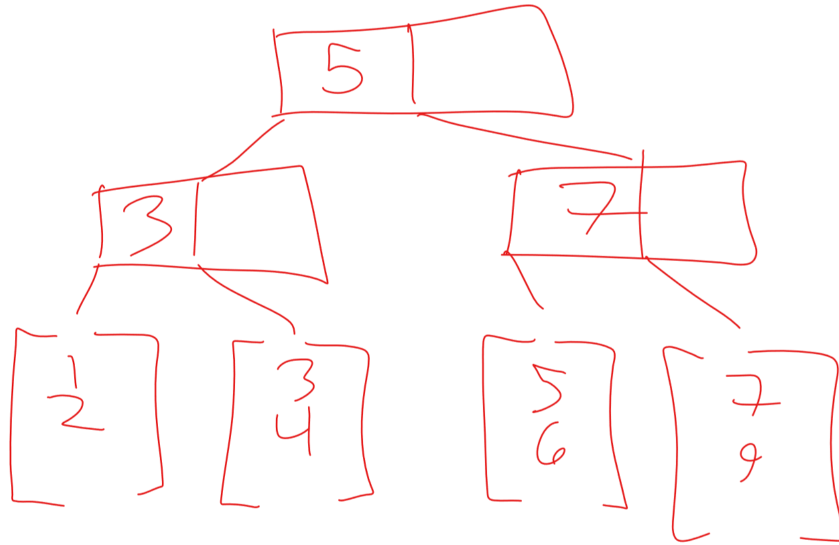
2. Using the key size you just calculated, find the maximum size (in bytes) of the values stored in the B-Tree. $L = 5$ for this B-Tree and leaf nodes also fit in one block exactly. Assume leaves can store the cost of just key+value pairs with no additional overhead.

   Solution: $175 = (value + 5) * 5$
   $value = 175/5 - 5 = 30$ bytes

3. From an initially empty B-Tree with $M = 3$ and $L = 3$ perform the following inserts in-order. Showing intermediate steps will greatly increase your chances of getting partial credit. Circle your final answer.

$$5, 3, 1, 2, 4, 7, 9, 6$$

(further space to do your B-Tree insertions)



Some Useful Facts

When we're using the tree method to solve a recurrence, we usually use the following steps:

0. Draw a few levels of the tree.
1. Let the root node be at level 0. Give a formula for the size of the input at level i.
2. What is the number of nodes at level i?
3. What is the work done at the ith recursive level?
4. What is the last level of the tree?
5. What is the work done at the base case?
6. Write an expression for the total work done.
7. Simplify until you have a "closed form" (i.e. no summations or recursion).

Geometric series identities:

$$\sum_{i=0}^{k} c^i = \frac{c^{k+1}-1}{c-1} \qquad \sum_{i=0}^{\infty} c^i = \frac{1}{1-c} \text{ if } |c| < 1$$

Common Summations:

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \qquad \sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \qquad \sum_{i=0}^{n} i^3 = \frac{n^2(n+1)^2}{4}$$

Log identities:

$$a^{\log_b(c)} = c^{\log_b(a)} \qquad \log_b(a) = \frac{\log_d(a)}{\log_d(b)}$$

Master Theorem:
Given a recurrence of the following form:

$$T(n) = \begin{cases} d & \text{if } n \leq \text{ some constant} \\ aT(n/b) + n^c & \text{otherwise} \end{cases}$$

with a, b, c are constants.
If $\log_b(a) < c$ then $T(n)$ is $\Theta(n^c)$
If $\log_b(a) = c$ then $T(n)$ is $\Theta(n^c \log n)$
If $\log_b(a) > c$ then $T(n)$ is $\Theta\left(n^{\log_b(a)}\right)$