READ THIS NOW!

- Print your name in the space provided below.
- There are 7 short-answer questions, priced as marked. The maximum score is 100.
- When you have finished, sign the pledge at the bottom of this page and turn in the test.
- Aside from the allowed one-page fact sheet, this is a closed-book, closed-notes examination.
- No laptops, calculators, cell phones or other electronic devices may be used during this examination.
- You may not discuss this examination with any student who has not taken it.
- Failure to adhere to any of these restrictions is an Honor Code violation.

Name (Last, First) Solution

printed

Pledge: On my honor, I have neither given nor received unauthorized aid on this examination.

_____

signed

1. [12 points] Suppose that an implementation of the Quicksort algorithm selects the pivot value for a sublist by always taking the first element of the sublist. Under what circumstances would this approach to choosing the pivot value guarantee the worst possible performance? Explain clearly, using an example.

   The worst choice for the pivot value would be either the largest or smallest in the sublist, since this would result in a partitioning to one empty sublist and one containing every value except the pivot.

   So, if the list were already sorted in descending order the approach above would consistently pick the largest value for the pivot.

   For example, suppose we have the following list:

   `45 40 35 30 25 20 15 10 5`

   We would take `5` for the first pivot, resulting in the partitioning:

   `empty | 45 | 40 35 30 25 20 15 10 5`

   And at the next step we would take `10` for the pivot, and obtain:

   `empty | 45 | empty | 40 | 35 30 25 20 15 10 5`

   This would take N-1 partitioning steps to process the list, which leads to worst-case behavior.

   If the list were already sorted in descending order the approach above would consistently pick the largest value for the pivot, again producing worst-case behavior.

   This problem could be ameliorated if the Quicksort implementation checked for a sorted, or reverse sorted, list before the first partitioning step.

   However, it is still possible for a partially sorted, or reverse sorted, list to produce worst-case behavior.

2. [12 points]] Consider designing a hash function to be used in organizing a collection of phone company customer records. Each customer has a unique ID, of the form `<citycode>-<numeric>`, where the city code consists of three to five upper-case letters and the numeric part is a five-digit number in the range 0 to 99999. For example: ROA-37412 or BBURG-23512. The company has many customers, but no more than 99999 in any particular city.

   The hash function will be applied to customer IDs. The following hash scheme is suggested: represent each character in the city code by its ascii value and multiply those values together, then take the left-most three digits of the numeric part of the ID and add that in, and finally mod the result by the size of the hash table. The multiplication will very likely result in an integer overflow, but that is not objectionable.

   The size of the hash table will be a prime integer close to twice the number of customers.

   Give two good, unrelated reasons why this hashing scheme should not be adopted unless it is modified.

   1. The hash algorithm is not sensitive to the ordering of the characters in the alphabetic `citycode`. So, two different values that contained the same characters in different orders would yield the same numeric value.

   2. The hash algorithm ignores the low-significance digits of the numeric part of the key. Unless there is a good reason to expect a lack of variation, it is not good practice to discard portions of the key. Worse, it seems likely that codes will be assigned sequentially as customers are added within the same `citycode`, so the low-significance digits are the ones that we would expect to exhibit the highest variance.

   3. Customer IDs with the same `citycode` will tend to form clusters, since the `citycode` will produce the same "base" address for all of them, and the only difference will result from the addition of the upper three digits of the numeric field, which must fall in the range 0 to 999.

3. [20 points] Consider a hash table consisting of $N = 17$ slots, and suppose integer key values are hashed into the table using the hash function:

$$h(K) = K\%N$$

Suppose that collisions are resolved using the probe function: $p(i) = i^2 + K$

where i is the number of probes that have been attempted and the probe value is added to the base slot index (mod N).

Show the contents of the hash table after the following key values have been inserted in the given order:

29 23 14 41 15 7 24 28

Slot number	Contents
0	
1	
2	
3	**28**
4	
5	
6	**23**
7	**41**
8	**7**
9	
10	
11	**24**
12	**29**
13	
14	**14**
15	**15**
16	

```
h(29) = 29 % 17 = 12

h(23) = 23 % 17 = 6

h(14) = 14 % 17 = 14

h(41) = 41 % 17 = 7

h(15) = 15 % 17 = 15

h(7) = 7 % 17 = 7 → collision!
   Probe:  7 + 1^2 = 8  free!

h(24) = 24 % 17 = 7 → collision!
   Probe:  7 + 1^2 = 8 collison!
           7 + 2^2 = 11 free!

h(28) = 28 % 17 = 11 -> collision!
   Probe:  11 + 1^2 = 12 collison!
   Probe:  11 + 2^2 = 15 collision!
   Probe:  11 + 3^2 = 20 % 17 = 3 free!
```

4. [12 points] B-trees and their relatives are generally used to index very large databases, in which even the index cannot be entirely stored in memory at once. Why is it better to use a B-tree, whose nodes store many values and have many children, rather than an AVL tree (assuming both would be stored on disk)?

Reading a single AVL node from disk yields only a single key value.

Reading a single B-tree node from disk typically yields lots of key values, and costs only slightly more than reading a single AVL node.

An average search in the AVL will require on the order of log N disk accesses.

An average search in the B-tree will require on the order of $\log_b N$ disk accesses, where b = m/2.

That's a LOT less disk accesses for the B-tree, typically no more than 4 or 5, and each of those takes only a little more time than a disk access for the AVL tree.

---

5. Recall that in a B+ tree internal nodes store ONLY key values and node pointers, and leaf nodes store complete records. Moreover, the number of key values stored in an internal node may be different than the number of data values stored in a leaf node. Consider using a B+ tree to store the index for a large disk file, so that each complete data record stored in the tree contains one 8-byte key value and 24 bytes of other data. Assume that file offsets occupy 4 bytes of space.

Suppose that the index will be stored on a disk where each sector occupies 2048 bytes, and that it is important for performance that each node of the B+ tree fit within a sector.

a) [8 points] What is the maximum number of key values that you can store in each internal node? Justify your answer.

We need for an internal node to fit within 2048 bytes. Internal nodes store N key values and N+1 file offsets (pointers) to child nodes. So, we need to satisfy the following inequality:

$$8N + 4(N + 1) \leq 2048$$
$$12N \leq 2044$$
$$N \leq 170.3$$

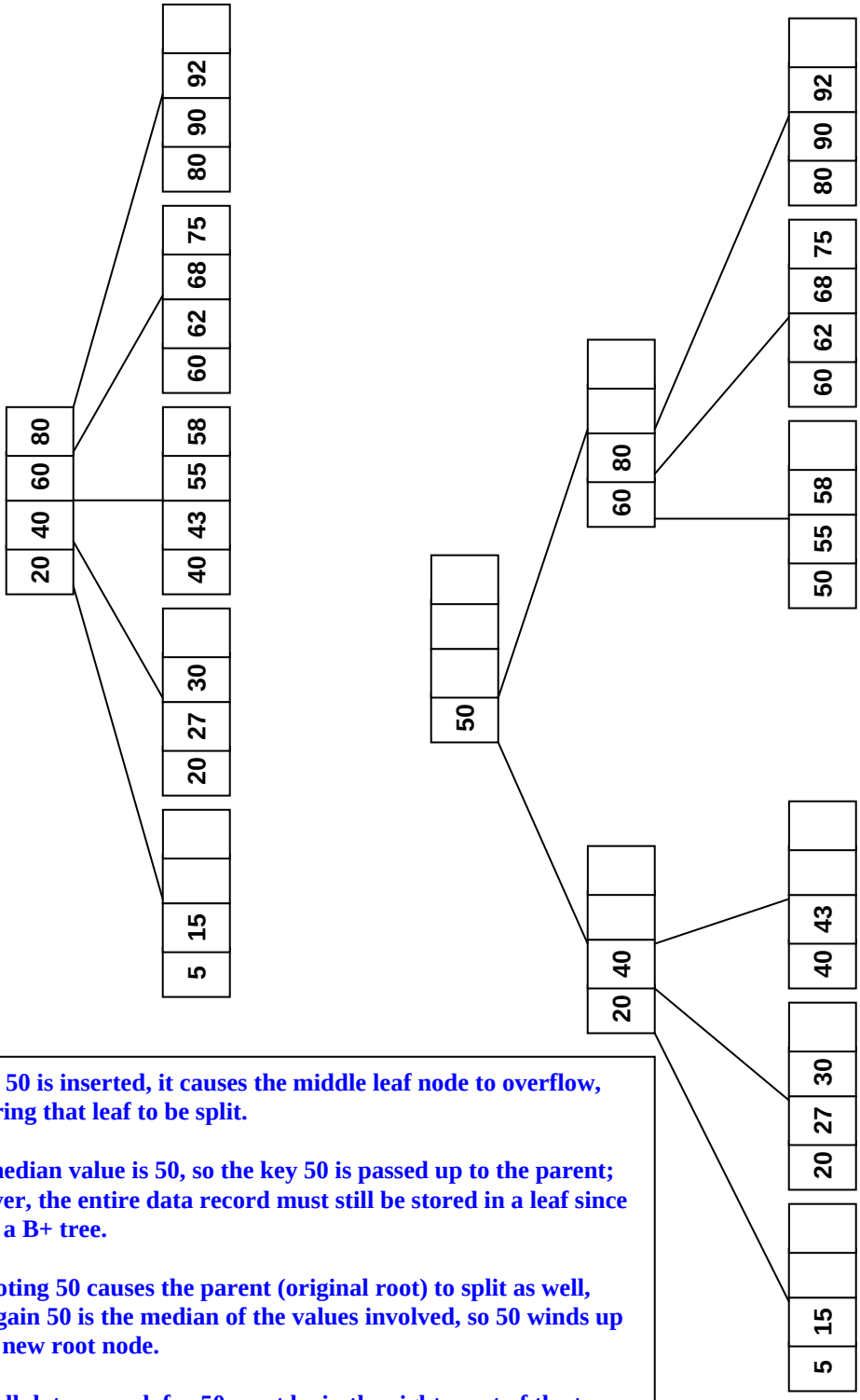
So, the internal nodes cannot store more than 170 key values. (You might note that it's probably necessary to also store a counter for the number of key values actually stored in the node, but that will easily fit in the 4 bytes that will be left over.)

b) [8 points] What is the maximum number of complete data records that you can store in each leaf node? Justify your answer.

This is similar, but leaf nodes do not store child pointers, and they do store full data objects and two pointers to adjacent leaves. So, we need to satisfy:

$$32N + 8 \leq 2048$$
$$32N \leq 2040$$
$$N \leq 63$$

6. [12 points] Consider the B+ tree of order 4 that is shown below. The diagram only shows key values but the leaf nodes store full data objects; the rest of the data object content is irrelevant. Draw the B+ tree that would result if a new data record with key value **50** were inserted.



When 50 is inserted, it causes the middle leaf node to overflow, requiring that leaf to be split.

The median value is 50, so the key 50 is passed up to the parent; however, the entire data record must still be stored in a leaf since this is a B+ tree.

Promoting 50 causes the parent (original root) to split as well, and again 50 is the median of the values involved, so 50 winds up in the new root node.

The full data record for 50 must be in the right-most of the two leaves involved in the splitting since the original tree shows that duplicate entries (key/full record pairs) are resolved to the right.

7. Consider the algorithm for external sorting used in Project 2. Suppose that during the run-building phase you use replacement selection with a heap that can store 1000 records and an input buffer that can store 100 records. Suppose that the file contains 100000 records.

a) [6 points] If the records in the file were initially in purely random order, how long would you expect the resulting runs to be? Why?

If the records were in purely random order we would reasonably expect that the root value in the heap (which is the smallest value and is being removed to be added to the current run) would be replaced by the next value from the input buffer about 50% of the time.

So, we would have to do about 2000 root removals before the heap would become empty (thus ending the current run).

So, we'd expect runs to average about 2000 data records.

Note: the minimum length for a run is always the capacity of the heap itself, since the algorithm will at least put all of the values used to initialize the heap into a single run.

b) [6 points] If the records in the file were initially in strictly descending order (sorted already but in backwards order), how long would you expect the resulting runs to be? Why?

If the file was sorted as described, then every element that is currently in the input buffer is guaranteed to be smaller than all the elements that are currently in the heap.

The old heap root is replaced by an element from the input buffer if and only if that element is larger than the old heap root.

So, we would NEVER replace the old root with the next value from the input buffer.

Therefore, the heap will always become empty after 1000 root removals, yielding a run of length 1000 every time.

Have a good break!