Homework 1: Models, Principles, Problems

You will turn in your solution to all 6 problems but we will grade only 3 of them 
(which I will pick semi-randomly).

1. **Implementing Chi-square** The Chi-Square statistic can be used to find 
if the overall set of observed character frequencies are unusually different 
(as compared to normal random variation) from the expected character 
frequencies. This is a more sophisticated test, statistically speaking, than 
the simple threshold detector used in the warm-up example. Assume that 
the thresholds represent the expected frequencies. The statistic is com- 
puted by finding the sum of \(( ExpectedFrequency[i] - ObservedFrequency[i] )^2 / ExpectedFrequency[i] \) 
for all values of character \( i \). The chip should alarm if the final statistic 
is above a specified threshold. (For example, a value of 14.2 implies that 
there is only a 1\% chance that the difference is due to chance variation.) 
Find a way to efficiently implement this statistic assuming once again that 
the length is known only at the end.

2. **Digital Design:** Multiplexers and barrel shifters are very useful in net- 
working hardware so working this problem can help even a software person 
to build up hardware intuition.

   - First show to design a 2-input multiplexer from basic gates (AND, 
   OR, NOT).

   - Next, generalize the idea shown in the chapter to design an N-input 
multiplexer from N/2 input multiplexers. Use this to describe a de-
   sign that takes log\( N \) gate delays and \( O(N) \) transistors.

   - Show how to design a barrel shifter using a reduction to multiplexers 
(i.e, use as many muxes as you need in your solution). Based on 
your earlier solutions what is the gate and time complexities of your 
solution?

3. **Memories and Pipelining Trees:** This problem studies how to pipeline 
a heap. A heap is important for applications like QoS where a router 
wishes to transmit the packet with the earliest timestamp first. This it 
makes sense to have a heap ordered on timestamps. To make it efficient, 
the heap needs to be pipelined in the same fashion as the binary search 
tree example in the chapter, though doing so for a heap is somewhat 
harder. The figure below shows an example of a P-heap capable of storing 15 keys. A P-heap is a full binary tree like a standard heap except that 
nodes anywhere in the heap can be empty as long as all children of the 
node are also empty (e.g., nodes 6, 12, 13).

   For the following explanations consult Figure 1 and Figure 2.

   Consider adding key 9 to the heap. Assume every node \( N \) has a count 
of the number of empty nodes in the subtree rooted at \( N \). Since 9 is less 
than the root value of 16, 9 must move below. Since both the left and
right children have empty nodes in their subtrees, we arbitrarily choose to add 9 to the left subtree (node 2). The index, value, and position values shown on the left of each tree are registers used to show the state of the current operation. Thus in Figure 1 Part (b), when 9 is added to the left subtree, the index represents the depth of the subtree (depth 2) and the position is the number of the node (i.e., node 2) that the value 9 is being added to.

Next, since 9 is less than 14, and since only the right child has space in its subtree, 9 is added to the subtree rooted at Node 5. This time 9 is greater than 7, so 7 is replaced with 9 (in node 5) and 7 is pushed down to the empty node 10. Thus in Figure 1 Part (d), the index value is 4 (i.e., operation is at depth 4) and the position is 10. Although in Figure 1, only one of the registers at any index/depth has non-empty information, keeping separate registers for each index will allow pipelining.

Consider next what is involved in removing the largest element (dequeue). Remove 16 and try to push down the hole created till an empty subtree
is created. Thus in Step 3, the hole is moved to node 2 (because its value 14 is larger than its sibling with value 10), then to node 4, and finally to node 9. Each time a hole is moved down, the corresponding non-empty value from below replaces the old hole.

- In order to make the enqueue operation work correctly, the count of empty subtree nodes must be maintained. Explain briefly how the count should be maintained for each enqueue and dequeue operation (the structure will be pipelined in a moment, so make sure the count values respect this goal).

- A logical thing to do is to pipeline by level as we did for the binary tree in the chapter. However, here we have a problem. At each level (say inserting 9 at the root) the operation has to consult the two children at the next level as well. Thus when the first operation moves down to Level 2, one cannot bring in a second operation to
Level 1 or there will be memory contention. Clearly waiting till one operation finishes completely will work but this reduces to sequential processing of operations. What is the fastest rate you can pipeline the heap?

- Consider the operations “Enqueue 9; Enqueue 4.5; Dequeue” pipelined as you have answered above. Show 6 consecutive snapshots of the tree supporting these 3 operations.

- Assume that each Level memory is an on-chip SRAM that takes 5 nsec for a memory access. Assume that you can read and write the value and count fields together in one access. Remember that some of the memories can be queried in parallel. What is the steady state throughput of the heap in operations per second?

- Could one improve the number of memory references by using a wider memory access and laying out the tree appropriately.

- Before this design, previous designs used a memory element for each heap element as well as logic for each element. Thus the amount of logic required scaled directly with heap size, which scales poorly in terms of density and power. In this design, the memory scales with the number of heap elements and thus scales with SRAM densities and power, but the logic required scales much better. Explain.

4. **Architecture, Caches, and Fast Hash Functions**: The L1 cache in a CPU provides essentially a fast hash function that maps from a physical memory addresses to its contents via the L1 cache. Suppose that one wants to teach an old dog (the L1 cache) a new trick (to do IP lookups). The goal is to use the L1 cache as a hash table to map 32-bit IP addresses to 7 bit port numbers. Assume a 16 Kbyte L1 cache of which the first 4 Kbytes are reserved for the hash table, and a 32 byte cache block size. Assume a byte addressable machine, a 32 bit virtual address, and a page size of 4 Kbytes. Thus there are 512 32-byte blocks in the cache. Assume the L1 cache is directly indexed (called direct mapped). Thus bits 5 through 13 of a virtual address are used to index into one of 512 blocks, with bits 0 through 4 identifying the byte within each block.

- Given that pages of size 4 Kbytes and the machine is byte addressable, how many bits in a virtual address identify the virtual page? How many bits of the virtual page number intersect with bits 5 through 13 used to index into the L1 cache?

- The only way to ensure that the hash table is not thrown out of the L1 cache when some other virtual pages arrive is to mark any pages that could map into the same portion of the L1 cache as uncacheable at startup (this can be done). Based on your previous answer and the fact that the hash table uses the first 4 Kbytes of L1 cache, precisely identify which pages must be marked as uncacheable.
To do a lookup of a 32 byte IP address, first convert the address to a virtual address by setting to 0 all bits except bits 5 through 11 (bits 12 and 13 are zero because only the top quarter of the L1 cache is being used). Assume this is translated to the exact same physical address. When a read is done to this address, the L1 cache hardware will return the contents of the first 32 bit word of the corresponding cache block. Each 32 bit word will contain a 25 bit tag and a 7 bit port number. Next, compare all bits in the IP address other than bits 5 through 11 with the tag, and keep doing so for each 32 bit entry in the block. How many L1 cache accesses are required in the worst case for a hash lookup? Why might this be faster than a standard hash lookup in software?

5. **Relaxing Consistency Requirements in a Name Service:** The Grapevine system offers a combination of a name service (to translate user names to inboxes) and a mail service. To improve availability, Grapevine name servers are replicated. Thus any update to a registration record (e.g., Joe → MailSlot3) must be performed on all servers implementing replicas of that record. Standard database techniques for distributed databases require that each update be atomic; that is, the effect should be as if updates were done simultaneously on all replicas. Because atomic updates require that all servers be available, and registration information is not as important as say bank accounts, Grapevine only provides the following loose semantics (P3): all replicas will eventually agree if updates stop. Each update is timestamped and passed from one replica to the other in arbitrary order. The highest timestamped update wins.

- Give an example of how a user could detect inconsistency in Joe’s registration during the convergence process.
- If Joe’s record is deleted it should eventually be purged from the database to save storage. Suppose a server purges Joe’s record immediately after receiving a DELETE update. Why might ADD updates possibly cause a problem? Suggest a solution.
- The rule that the latest timestamp wins does not work well when two administrators try to create an entry with the same name. Because a later creation could be trapped in a crashed server, the administrator of the earlier creation can never know for sure that his creation has won. The Grapevine designers did not introduce mechanisms to solve this problem but relied on “some human-level centralization of name creation.” Explain their assumption clearly.

6. **Optimizing the Expected Case, Using algorithmic ideas, and Scavenging Files:** The Alto computer used a scavenging system that scans the disk after a crash to reconstruct file system indexes that map from file names and blocks to disk sectors. This can be done because each disk sector that contains a file block also contains the corresponding file
identifier. What complicates matters is that main memory is not large enough to hold information for every disk sector. Thus a single scan that builds a list in memory for each file will not work. Assume that the information for a single file will fit into memory. Thus a way that will work is to make a single scan of the disk for each file: but that would be obvious waste (P1) and too slow.

Instead, observe that in the expected case, most files are allocated contiguously. Thus suppose File X has pages 1-1000 located on disk sectors 301-1301. Thus the information about 1000 sectors can be compactly represented by 3 integers and a file name. Call this a run node.

- Assume the expected case holds and that all run nodes can fit in memory. Assume also that the file index for each file is an array (stored on disk) that maps from file block number to disk sector number. Show how to rebuild all the file indexes.

- Now suppose the expected case does not hold and the run nodes do not all fit into memory. Describe a technique, based on the algorithmic idea of divide-and-conquer (P15), that is guaranteed to work (without reverting to the naive idea of building the index for one file at a time unless strictly necessary.)