Homework 1: Hardware Models

You will turn in your solution to all 4 problems.

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 computed 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 1.4% 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 networking 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 design 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. **Memory Design:** For the design of the pipelined flow ID lookup scheme described above, draw the timing diagrams for the pipelined lookups. Use the numbers described in the chapter and clearly sketch a sample binary tree with 15 leaves and show how it can be looked up after 4 lookups on 4 different banks. Assume a binary tree not a ternary tree. Also, calculate the number of keys that can be supported using 16 banks of RAMBUS if the first k levels of the tree are cached in on-chip SRAM.

4. **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 make 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 **??** and Figure **??**.

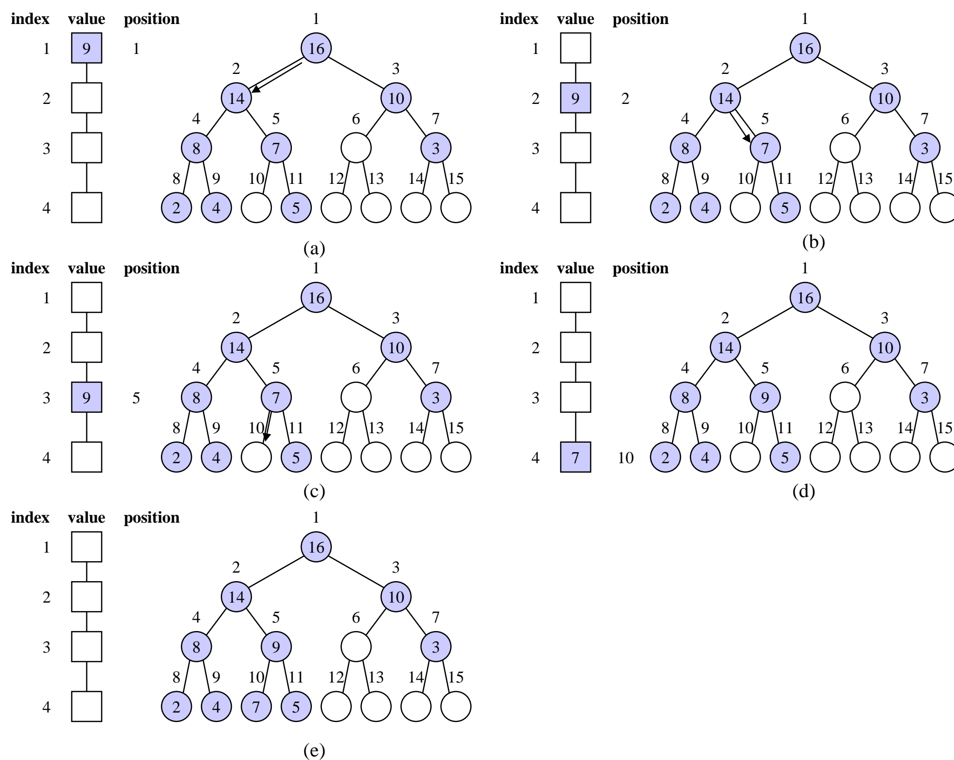

Figure 1: An Enqueue Example in 5 snapshots to be read from left to right, and then top down. In each snapshot, the index represents the depth of the subtree and the position is the number of the node that the value is being added to.

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 **??** 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 **??** Part (d), the index value is 4 (i.e., operation is at depth 4) and the position is 10. Although in Figure **??**, only one of the registers at any index/depth has non-empty information, keeping separate registers for each index will allow pipelining.

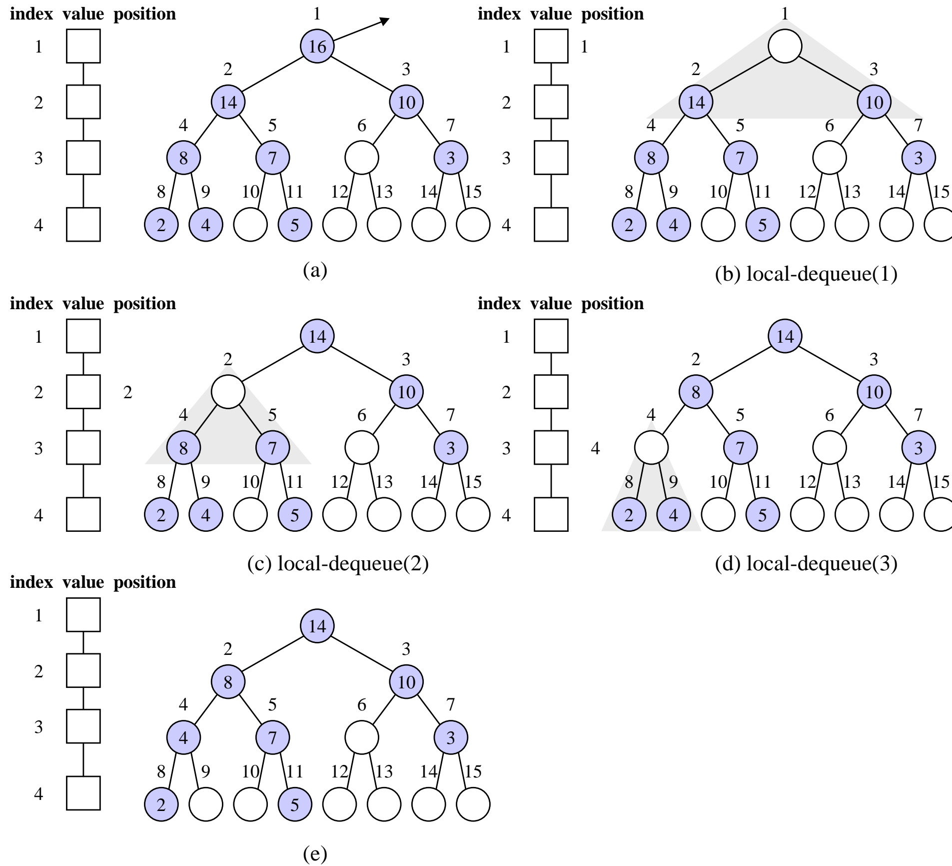

Figure 2: Dequeue Example

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.