CSCI 6620 / 2226 Spring 2016

Heaps and Huffman Codes

1 Priority Queues.

A priority queue is a container with the following properties:

- Every data item in the queue has an associated priority.
- Items can be inserted in any order, and are stored within the container in order of priority.
- When we pop (or remove an item from) the priority queue, it is the always highest-priority item in the queue.

A priority queue can be implemented by a sorted vector or linked list, but a heap is a more efficient implementation because insertion into a heap is faster than insertion into a sorted list.

2 Heaps

A heap is a partially sorted data structure that stores a binary tree in an vector, without pointers. In this implementation, we use a vector for the storage and assume we are storing items of type Node, which is a Node of values, a data item and its associated priority.

- The heap class has one data member: a vector¡Node¿.
- The root of the tree is stored in slot 1 of the vector. (Position 0 is left empty.)
- At all times, vector slots $1 \ldots n$ contain data items and slots $n+1 \ldots$ are empty.
- For any tree-node k, the left son of k is in vector slot $2k$ and the right son is in slot $2k+1$.

There are two kinds of heaps: max-heaps have the greatest priority nodes at the root of the heap and min-heaps put the lowest-priority nodes at the root. This writeup describes a max-heap. The Huffman code uses a min-heap. Every mention of greater or less would need to be reversed to describe a min-heap.

There are three essential heap functions; we will call them *push* (or add), *pop* (or remove), and $heapify$ which, in turn, are implemented by two basic algorithms: $upHeap$ and $downHeap$. We describe these two algorithms first, followed by descriptions of the three heap functions.

- **upHeap(int k):** This is used when a new Node is added at the end of the heap, in position k, or when the priority of a heap node is increased. (In the Huffman algorithms, the priorities never change.)

  Readjust the position of the new Node by moving the Node up the tree until its priority is greater than its son(s) and less than its parent. In each comparison, the moving node is the son.

  1. Let $p = k/2$ be the position of the parent of slot k.
  2. If $p.priority() > k.priority()$, you are done.
  3. Otherwise swap the Nodes at positions p and k.
  4. Let $k = p$ and repeat from step 1.

- **downHeap(int k):** This function is called repeatedly during the $heapify()$ operation and once after popping a heap node, when the data from the last node is moved to the opening at position 1 and must be pushed down into its proper place. This algorithm would also be used if the priority of a heap node decreased. (Decreases do not happen in the Huffman algorithm.)

  Readjust the position of the Node at position k by moving the Node down the tree until its priority is greater than both sons but less than its parent. In each comparison, the moving node is the parent, so both sons must be considered.

  1. Let $s = 2 * k$ and let $r = s + 1$. These are the positions of the two sons of slot k.
  2. If $s.priority() > r.priority()$ set $s = r$.
  3. If $k.priority() > s.priority()$ you are done.
  4. Otherwise, swap the nodes at positions k and s, then set $k = s$.
  5. Repeat from step 1 until s is the subscript of a leaf node.

- **push(Node):** This function inserts a new Node into the container in a position determined by the priority. To do this:

  1. Put the new Node at the end of the heap and increment the heap counter.
  2. Starting at the new Node, do an $upHeap$ operation.

- **Node pop():** This function removes and returns the highest priority Node in the tree:

  1. Store a copy of the Node in heap position 1 so that it can be returned later.
  2. Move the last item in the heap to position 1 and decrement the heap counter.
  3. Starting at position 1, do a $downHeap$ operation.
  4. Return the item that was removed.

- **void heapify(Node[] vec):** We assume that n data Nodes are already stored in the Node-vector, *vec*, but not in legal heap-order. The goal is to correct the order so that every parent node in the tree has greater priority than both of its children.

  1. Let $k = n/2$, the position of the right-most parent node in the vector.
  2. Do a $downHeap$ operation starting with k.
  3. Decrement k and repeat step 2 until you have done a downHeap from position 1.

3 Huffman Codes

ASCII is a fixed-length 7-bit code that is capable of representing 128 different characters. We commonly use about 100 of these:

- 10 digits
- 52 upper-case and lower-case letters
- 24 operators and punctuation characters
- 8 grouping symbols: `()` {} `[]` `<>`
- 5 non-printing characters: newline, tab, carriage return, null, space

An 8-bit byte is capable of representing 256 characters. So if we are writing ordinary things, such as text or programs, we are using only about 40% of the representational capacity of each byte in the file.

The idea of a Huffman code is to exploit the unused capacity by using a variable length encoding for characters and, more than that, to encode only the characters that are used in a given text. Finally, frequently used characters are given shorter codes than rarely used characters. Thus, every coded document would use a different code, since the code is based on the frequency of the characters actually used in the document. Given a document, there are four steps in creating a Huffman code:

- Read the file and count how many times each character in it is used.
- Create a new Node for each character with a non-zero usage count. Store the Node references in an vector , starting with position 1. (Leave position 0 empty. In the diagrams, we use a dark background to indicate a box that is empty.)
- Make a min-heap out of these Nodes; Less frequently used characters will be closer to the root of the tree, commonly used letters will be near the leavess.
- Use the heap to build a binary tree by repeating the following steps until there is only one Node left on the heap:
  - Remove two Nodes from the heap.
  - Create a new Node whose left and right sons are the two Nodes you removed. The priority of this node equals the sum of the priorities of the two sons.
  - Add the new Node to the heap.
- Return that final node, which is a Huffman code tree. In this tree, all the original letter-nodes will be leaves, and all interior nodes have two sons. In the textbook's terminology, it is a 2-tree.