Joanna Klukowska
joannakl@cs.nyu.edu

Lecture 6: Priority Queues and Heaps

Reading materials

Goodrich, Tamassia, Goldwasser: Chapter 9

OpenDSA: chapter 12 (Heaps and Priority Queues and Huffman Coding)

Topics Covered

Joanna Klukowska
joannakl@cs.nyu.edu

1 Priority Queues and Heaps

A few weeks ago we looked at queue data structure. In a typical queue, the items that arrive at the queue first are removed from the queue first (first in, first out, or FIFO, structure). Our typical queue is a special case of a priority queue in which the priority is decided based on time of arrival. In more general priority queue, the priority of an element can be decided based on anything. The next element to be removed (or processed, or served, etc) is the element whose priority is highest.

Examples:

- Boarding the plane is done according to a (partial) priority queue - the passengers who paid the most for their tickets board first (they have higher priority).
- Job scheduling on a processor - all computer processes have some priority assigned to them and the ones with highest priority get the time on a processor first. The assignment of priorities for computer processes is a complicated and challenging problem - you'll see more about it in your operating systems course.
- Registering for classes on Albert (partial priority queue) - the seniors get to register sooner than the other students.

1.1 Priority Queue Interface

A sample priority queue interface may look as follows (it is practically the same as the interface provided by the regular queue). We will eventually modify it to explicitly represent priorities, but this is not essential just now.

`boolean isFull( )`	returns true if the priority queue is full (this may never happen), false otherwise
`boolean isEmpty( )`	returns true if the priority queue is empty, false otherwise
`void enqueue( T element)`	adds another element to the queue
`T dequeue( )`	removes and returns the element with highest priority from the queue (ties are resolved arbitrarily and are implementation dependent)

1.2 Implementation

Throughout the semester, we discussed several different structures that can be used for implementation of a priority queue. Below we review different options.

Joanna Klukowska
joannakl@cs.nyu.edu

An Array-based Sorted List

`enqueue()` operation can be implemented using binary search to find the correct locations - this can be done in O(log_2N) time, but then other elements in the array need to be shifted to accomodate the new element - this makes it O(N)

`dequeue()` operation is simple as we need to remove the item with highest priority which is at the front of the list which requires only constant time O(1) (well, assuming that we do not insist on keeping the front of the queue at the index 0 of the array - the front of the queue needs to be able to move through the array, otherwise the dequeue operation becomes O(N))

In addition we need to introduce finite size or resize the array when it becomes full.

A Reference-based Sorted List

`enqueue()` operation has to traverse through the list in order to find the correct place to insert the element, that is O(N) time

`dequeue()` operation requires removing the head of the list which is constant time operation, O(1)

The queue can grow in size indefinitely (well, within our memory limitations).

A Binary Search Tree

`enqueue()` operation is always proportional to the height of the tree

`dequeue()` operation is always proportional to the height of the tree

This is good and bad. If the tree is nice and bushy, that this means that we can perform both enqueue and dequeue in O(log_2N) time. But the binary search trees can become unbalance and the true worst case is O(N) for both operations. We could use a self-balancing binary search tree, but there is something better.

The queue can grow in size indefinitely (well, within our memory limitations).

1.3 Heaps as Implementation of Priority Queues

Our goal is to find an implementation that guarantees better than O(N) performance on both `enqueue` and `dequeue` operations, even in the worst case. This is where the **heap** data structure comes in.

1.3.1 Max-Heap

Max-heap is a complete binary tree with a property that each node contains a value (priority in our case) that is greater than or equal to the value of its children. When we talk about heaps, we need to make sure that a heap maintains two properties:

shape property the heap has to be a complete binary tree (a *complete binary* tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible),

Joanna Klukowska
joannakl@cs.nyu.edu

order property each node has a value greater than or equal to the values of both of its children.

`enqueue()` operation is always proportional to the height of the tree, but because of the shape property, the height is always guaranteed to be O(log_2N)

`dequeue()` operation can be done fast by accessing the node at the root of the tree, O(1), but since we will need to replace that node with something else, it also turns out to be O(log_2N)

Both of these running times are guaranteed even in the worst case.

Enqueing an element During enqueue operation, we add the new node as the last element in the bottom row of the tree. This maintains the shape property (the tree is still complete), but it may violate the order property. We need to *heapify* the tree, by moving the new node up (swapping with the parent node) until its parent is greater than, or equal to it.

Dequeueing an element During the dequeue operation, we remove and return the root of the tree. But that leaves a hole in the tree. To fill it, we take the rightmost node from the bottom level of the tree and place it at the root. This preserves the shape property, but we need to do some work to restore the order property. As with the enqueue, we heapify the tree by moving the new root down (swapping with the larger of its children) until we find the position at which both of its children are smaller.

As you remember, in a binary tree (complete or not) the parents have references to the children, but not the other way around. This means that the heapifying the tree might be a problem. Well, the heaps are not implemented as trees, they are implemented as arrays. It is actually much more efficient to implement them as arrays because the heaps are complete trees and the relationship between index of a parent and its children is easy to establish. On the other hand, it is much easier to think about the operations performed on the heaps when we think about them as trees. There is a one-to-one correspondence between an array and a tree representation of any heap.

1.3.2 Min-Heaps

It is intuitive to think of largest elements at the front of a priority queue because these are the elements with the highest priority. But often it is the smallest elements that have higher priority. We can implement a min-heap in exactly same way as the max-heap. The only difference is that the order property changes to ”each node has a value smaller than or equal to the values of both of its children”.

1.3.3 Using Arrays to Implement Heaps

We can populate an array with values stored in nodes of a complete binary tree by reading them off from the root one level at a time (this is breadth first search traversal of a tree). This gives us the following properties:

- the root of a tree is at index `0`

Joanna Klukowska
joannakl@cs.nyu.edu

- if an element is not a root, its parent is at position `(index-1)/2`
- if an element has a left child, the child is at position `index*2+1`
- if an element has a right child, the child is at position `index*2+2`

This allows us to move from parent to child and child to parent very quickly (no need to worry about storing extra references) and, because of the heap property, we do not need to traverse the entire array to find the right place for a node.

See simulations on the OpenDSA website: http://algoviz.org/OpenDSA/Books/OpenDSA/html/Heaps.html.

1.4 Java Libraries Implementation of a Priority Queue

Java implements a heap based priority queue in the class PriorityQueue<E>, see http://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html for details. You should also take a look at the actual implementation of this class (this can be downloaded with the source code for JDK).

The method names are slightly different than what we have been using. The enqueue operation is provided by `add(E element)` method and the dequeue operation is provided by `poll()` method.

2 File Compression

When a text file is stored on a computer, each letter is represented by a binary code corresponding to that letter. Depending on the file encoding (ASCII or UNICODE), we might be using anywhere from 1 to 4 bytes per character. The War and Peace text file, for example, has 3,226,614 characters, that is over 3MB (and that is if the file uses only 1 byte to represent each character, i.e., the ASCII encoding).

Do we really need to use all that space to save this file?

We compress files all the time. How can this be done and what does it have to do with data structures?

In 1952 David Huffman showed that we do not need all the bits to represent the text (well, I am not sure if he was thinking about bits, but his idea of minimal prefix encoding allows us to do this). Here is the basic idea.

Use different length of bit sequences to represent different letters. If some letters occur much more frequently than others, we want to use a very short bit sequence to represent them. If some letters occur very rarely, then we can afford to use a long sequence of bits to represent them. The necessary condition for this to work is the existence of a prefix-code: coding of letters such that no valid code is ever a prefix of another valid code, otherwise we could not recognize boundaries between letters.

Knowing statistical properties of letters used to produce English text, we can figure out best bit sequences for each letter. But we can do even better if the sequences are based on specific text that we want to make ”smaller” (i.e., compress).

The method to come up with such encoding (they are not unique) is as follows:

Joanna Klukowska
joannakl@cs.nyu.edu

1. compute frequencies of every single letter occurring in a given text
2. create individual nodes that contain a letter and its frequency (this is a forest of single nodes of binary trees)
3. create a priority queue of the trees (the priority is based on the frequencies: the lower frequency, the higher the priority of the tree)
4. as long as the priority queue has more than one tree in it
   (a) pick two trees with the smallest frequencies and merge them by creating a new node and settings its children to the two trees, assign the sum of the frequencies of the two trees to the new tree
   (b) put the new tree back on the priority queue

Once the Huffman tree is complete the actual letters (and their counts) are located at the leaves. To determine the binary sequences that should be assigned to them, label all the left children references (edges from parent to its left child) with 0 and all the right children references (edges from parent to its right child) with 1. For each leaf construct the sequence by following the edges from the root to the leaf and reading the labels on the edges.

Visualization of building a Huffman tree and establishing the binary sequences can be found on OpenDSA website: https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/Huffman.html.