**Priority Queues**

The heap is used to implement a special kind of queue called a priority queue

The heap is not very useful as an ADT on its own

* We will not create a Heap interface or code a class that implements it
* Instead, we will incorporate its algorithms when we implement a priority queue class (which implements priority queue ADT) and heapsort

Sometimes a FIFO queue may not be the best way to implement a waiting line

A priority queue is a data structure in which only the highest-priority item is accessible

In a print queue, sometimes it is more appropriate to print a short document that arrived after a very long document.

* If we use insertion time as the priority value, then priority queue becomes regular (FIFO) queue.
* If we say highest priority is highest time, then it becomes stack.
* We can say smaller jobs have higher priority, then it becomes priority queue.

A priority queue is a data structure in which only the highest-priority item is accessible (as opposed to the first item entered)

Insertion into a Priority Queue

Diagram

Description automatically generated

**PriorityQueue Class**

Java provides a PriorityQueue<E> class that implements the Queue<E> interface.

Table

Description automatically generated

Using a Heap as the Basis of a Priority Queue

In a priority queue, just like a heap, the smallest item always is removed first

Because heap insertion and removal is O(logn), a heap can be the basis of a very efficient implementation of a a priority queue

While the java.util.PriorityQueue uses an Object array, we will use an ArrayList for our custom priority queue, KWPriorityQueue

Text, letter

Description automatically generated

Using a Comparator

Comparator<E> is an interface that has only 1 method (compare method that takes 2 elements of type E, compares them and returns positive or negative or 0 value).

comparator field is used to compare 2 items’ priorities, not the values.

To use an ordering that is different from the natural ordering, provide a constructor that has a Comparator<E> parameter

Graphical user interface, text, application, email

Description automatically generated

- offer()

Text, letter

Description automatically generated

- poll()

Text, letter

Description automatically generatedText, letter

Description automatically generated

Other Methods

The iterator and size methods are implemented via delegation to the corresponding ArrayList methods

Method isEmpty tests whether the result of calling method size is 0 and is inherited from class AbstractCollection

- compare()

If data field comparator references a Comparator<E> object, method compare delegates the task to the object’s compare method

If comparator is null, it will delegate to method compareTo of E

Text, letter

Description automatically generated

PrintDocuments Example

The class PrintDocument is used to define documents to be printed on a printer

We want to order documents by a value that is a function of both size and time submitted

In the client program, use:

Queue printQueue = new PriorityQueue(new ComparePrintDocuments());

Text, letter

Description automatically generated

**HUFFMAN TREES**

Diagram

Description automatically generated

A Huffman tree can be implemented using a binary tree and a PriorityQueue

A straight binary encoding of an alphabet assigns a unique binary number to each symbol in alphabet

* Unicode is an example of such a coding

The message “go eagles” requires 144 bits in Unicode but only 38 bits using Huffman coding

In binary search tree, top-down built up is used.

In Huffman tree, we built bottom-up. We merge smaller trees together and end up with tree at former page.

We always merge 2 trees which has lowest frequency and to be able to find the lowest frequency trees, we will use priority queue. Frequency value will be used as priority.

**Building a Custom Huffman Tree**

Suppose we want to build a custom Huffman tree for a file

We first have to build the table below from the file and use that table for construction

**Input** : an array of objects such that each object contains a reference to a symbol occurring in that file and the frequency of occurrence (weight) for the symbol in that file

Table

Description automatically generated

Analysis

Each node will have storage for 2 data items:

1. the weight of the node
2. the symbol associated with the node

All symbols will be stored in leaf nodes

For nodes that are not leaf nodes, the symbol part has no meaning

The weight of a leaf node will be the frequency of the symbol stored at that node

The weight of an interior node will be the sum of frequencies of all leaf nodes in the subtree rooted at the interior node

A priority queue will be the key data structure in our Huffman tree

We will store individual symbols and subtrees of multiple symbols in order by their priority (frequency of occurrences)

2 Huffman trees can be compared based on their frequencies at the root node (70 has higher priority than 100).

In the beginning, we just create leaf nodes with symbols and frequencies as single tree. So if there are 29 symbols at the beginning, we will have 29 trees of 1 node only and each of them is a Huffman tree.

Diagram

Description automatically generated

A picture containing diagram

Description automatically generated

Design

Text

Description automatically generated

If you use ArrayList as PriorityQueue, you get linear time in line 4.

If you use BinarySearchTree as PriorityQueue, you get linear time in line 4.

If you use BinaryHeap as PriorityQueue, you get (logn) in line 4.

If you use sorted array as PriorityQueue, you get constant time in line 4, but linear time in line 6.

PriorityQueue is ADT, there are several implementations of it. As in List has LinkedList, ArrayList, … implementations.

Graphical user interface

Description automatically generated with medium confidence