2.4.1 Suppose that the sequence P R I O \* R \* \* I \* T \* Y \* \* \* Q U E \* \* \* U \* E (where a letter means insert and an asterisk means remove the maximum) is applied to an initially empty priority queue. Give the sequence of letters returned by the remove the maximum operations.

I O P I R R T Y E Q U U

2.4.2 Criticize the following idea: To implement find the maximum in constant time, why not use a stack or a queue, but keep track of the maximum value inserted so far, then return that value for find the maximum?

delMin() needs ~2N stack/queue operations and around ~N extra space at the worst case.

2.4.3 Provide priority-queue implementations that support insert and remove the maximum, one for each of the following underlying data structures: unordered array, ordered array, unordered linked list, and linked list. Give a table of the worst-case bounds for each operation for each of your four implementations.

Unordered array 1 N amortized

Ordered array N 1 amortized

Unordered linked list 1 N

Ordered linked list N 1

2.4.4 Is an array that is sorted in decreasing order a max-oriented heap?

Yes, because for any 0 <= i <= n/2, a[i] >= a[2i], a[i] >= [2i + 1]

2.4.5 Give the heap that results when the keys E A S Y Q U E S T I O N are inserted in that order into an initially empty max-oriented heap.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| A | E | E | I | O | U | S | Y | T | Q | N |

2.4.6 Using the conventions of Exercise 2.4.1, give the sequence of heaps produced when the operations P R I O \* R \* \* I \* T \* Y \* \* \* Q U E \* \* \* U \* E are performed on an initially empty max-oriented heap.

2.4.7 The largest item in a heap must appear in position 1, and the second largest must be in position 2 or position 3. Give the list of positions in a heap of size 31 where the kth largest (i) can appear, and (ii) cannot appear, for k=2, 3, 4 (assuming the values to be distinct).

K = 2. Position 2, 3;

K = 3. Position 2, 3, 4, 5, 6, 7

K = 4. Position 2 ~ 15

2.4.8 Answer the previous exercise for the kth smallest item.

K = 2. Position 8~31

K = 3. Position 8~31

K = 4. Position 8~31

For any 1 <= i <= n

Heap[i] can be:

largest

smallest

2.4.9 Draw all of the different heaps that can be made from the five keys A B C D E, then draw all of the different heaps that can be made from the five keys A A A B B.

A B C D E = total 12 kinds of heaps

A A A B B = total 3 cases.

2.4.10 Suppose that we wish to avoid wasting one position in a heap-ordered array pq[], putting the largest value in pq[0], its children in pq[1] and pq[2], and so forth, proceeding in level order. Where are the parents and children of pq[k]?

T Parent<T>(T[] a, int k) where T : IComparable<T>

{

return a[(k-1)/2];

}

T LeftChild<T>(T[] a, int k) where T : IComparable<T>

{

Return a[2 \* k + 1];

}

T RightChild<T>(T[] a, int k) where T : IComparable<T>

{

Return a[2 \* (k + 1)];

}

2.4.11 Suppose that your application will have a huge number of insert operations, but only a few remove the maximum operations. Which priority-queue implementation do you think would be most effective: heap, unordered array, or ordered array?

Suppose we have N insertion operation while M remove the maximum operation

Where N >> M

Heap ~NlgN M M + NlgN

Unordered 1 M \* N M \* N + 1

Ordered ~N^2/2 1 1 + N^2 /2

If M < lgN then unordered array will be the method to chose

2.4.12 Suppose that your application will have a huge number of find the maximum operations, but a relatively small number of insert and remove the maximum operations. Which priority-queue implementation do you think would be most effective: heap, unordered array, or ordered array?

Follow the argument from previous question then the Ordered array will be the method to choose.

2.4.13 Describe a way to avoid the j < N test in sink().

Test a[j+1] != null

Or use a sentinel, i.e. when the N is odd, place a minimum value (for max heap) at the position N + 1;

2.4.14 What is the minimum number of items that must be exchanged during a remove the maximum operation in a heap of size N with no duplicate keys? Give a heap of size 15 for which the minimum is achieved. Answer the same questions for two and three successive remove the maximum operations.

Minimum number of items that must be exchange floor(lgN) - 1

2.4.15 Design a linear-time certification algorithm to check whether an array pq[] is a min-oriented heap.

bool IsHeap(T[] pq) where T : IComparable<T>

{

for(int i = 1; i <= n ; i++)

If(pq[i] > pq[2\*i] || (2\*i+1<= n && pq[i] > pq[2\*i+1])

return false;

return true;

}

2.4.16 For N=32, give arrays of items that make heapsort use as many and as few compares as possible.

2.4.17 Prove that building a minimum-oriented priority queue of size k then doing N k replace the minimum (insert followed by remove the minimum) operations leaves the k largest of the N items in the priority queue.

2.4.18 In MaxPQ, suppose that a client calls insert() with an item that is larger than all items in the queue, and then immediately calls delMax(). Assume that there are no duplicate keys. Is the resulting heap identical to the heap as it was before these operations?

Yes. Let its path to go up be n/2, n/4 … 1

Then when delMin is called, the one exchanged to 1 was a[n/2] and it will just follow the path the maximum went up because, all the items in this path are larger than its sibling for they were parent of its sibling before.

Answer the same question for two insert() operations (the first with a keylarger than all keys in the queue and the second for a key larger than that one) followed by two delMax() operations.

2.4.19 Implement the constructor for MaxPQ that takes an array of items as argument, using the bottom-up heap construction method described on page 323 in the text.

2.4.20 Prove that sink-based heap construction uses fewer than 2N compares and fewer than N exchanges.

2.4.21 Elementary data structures. Explain how to use a priority queue to implement the stack, queue, and randomized queue data types from Chapter 1.

2.4.22 Array resizing. Add array resizing to MaxPQ, and prove bounds like those of Proposition Q for array accesses, in an amortized sense.

2.4.23 Multiway heaps. Considering the cost of compares only, and assuming that it takes t compares to find the largest of t items, find the value of t that minimizes the coefficient of N lg N in the compare count when a t-ary heap is used in heapsort. First, assume a straightforward generalization of sink(); then, assume that Floyd’s method can save one compare in the inner loop.

2.4.24 Priority queue with explicit links. Implement a priority queue using a heapordered binary tree, but use a triply linked structure instead of an array. You will need three links per node: two to traverse down the tree and one to traverse up the tree. Your implementation should guarantee logarithmic running time per operation, even if no maximum priority-queue size is known ahead of time.

2.4.25 Computational number theory. Write a program CubeSum.java that prints out all integers of the form a3 + b3 where a and b are integers between 0 and N in sorted order, without using excessive space. That is, instead of computing an array of the N2 sums and sorting them, build a minimum-oriented priority queue, initially containing (03, 0, 0), (13, 1, 0), (23, 2, 0), . . . , (N3, N, 0). Then, while the priority queue is nonempty, remove the smallest item(i3 + j3, i, j), print it, and then, if j < N, insert the item (i3 + (j+1)3, i, j+1). Use this program to find all distinct mintegers a, b, c, and d between 0 and 106 such that a3 + b3 = c3 + d3.

**class** Cube **implements** Comparable<Cube>

{

**public** **long** Sum;

**public** **long** a;

**public** **long** b;

**public** **int** compareTo(Cube o) {

**long** ret = Sum - o.Sum;

**if**( ret > 0) **return** 1;

**else** **if**(ret < 0) **return** -1;

**else** **return** 0;

}

**public** Cube(**long** a2, **long** l)

{

**this**.a = a2;

**this**.b = l;

Sum = a2\*a2\*a2 + l\*l\*l;

}

}

**public** **class** CubeSum {

**public** **static** **void** main(String[] args) {

MinPQ<Cube> pq = **new** MinPQ<Cube>();

**int** upper = 1000000;

**for**(**int** i = 0 ; i <= upper ; i++)

pq.enqueue(**new** Cube(i, 0));

**long** prevSum = -1;

Cube prevCube = **null**;

**int** totalCount = 0;

**while**(!pq.isEmpty())

{

Cube c = pq.dequeue();

//System.out.println(c.a + c.b);

**if**(prevCube != **null** && c.Sum == prevSum && c.a != prevCube.b )

{

System.*out*.println("FOUND ONE: a = " + prevCube.a + " b = " + prevCube.b + " c = " + c.a + " d = " + c.b);

totalCount++;

}

**else**

{

prevCube = c;

prevSum = c.Sum;

}

**if**(c.b < upper)

pq.enqueue(**new** Cube(c.a, c.b + 1));

}

System.*out*.println(totalCount);

}

}

2.4.26 Heap without exchanges. Because the exch() primitive is used in the sink() and swim() operations, the items are loaded and stored twice as often as necessary. Give more efficient implementations that avoid this inefficiency, a la insertion sort (see Exercise 2.1.25).

2.4.27 Find the minimum. Add a min() method to MaxPQ. Your implementation should use constant time and constant extra space.

2.4.28 Selection filter. Write a TopM client that reads points (x, y, z) from standard input, takes a value M from the command line, and prints the M points that are closest to the origin in Euclidean distance. Estimate the running time of your client for N = 108 and M = 104.

2.4.29 Min/max priority queue. Design a data type that supports the following operations: insert, delete the maximum, and delete the minimum (all in logarithmic time); and find the maximum and find the minimum (both in constant time). Hint: Use two heaps.

2.4.30 Dynamic median-finding. Design a data type that supports insert in logarithmic time, find the median in constant time, and delete the median in logarithmic time. Hint: Use a min-heap and a max-heap.

**public** **class** DynamicMedianFinding<T **extends** Comparable<T>> {

**private** MaxPQ<T> lessers = **new** MaxPQ<T>();

**private** MinPQ<T> greaters = **new** MinPQ<T>();

**private** **int** size = 0;

**public** DynamicMedianFinding()

{

}

**public** **void** insert(T item)

{

**int** cmp = size > 0 ? item.compareTo(median()) : -1;

**if**(cmp <= 0) lessers.enqueue(item);

**else** greaters.enqueue(item);

size++;

balance();

}

**private** **void** balance() {

**if**(greaters.size() > lessers.size())

lessers.enqueue(greaters.dequeue());

**else** **if**(lessers.size() > greaters.size() + 1)

greaters.enqueue(lessers.dequeue());

}

**public** T median()

{

**return** lessers.size() > 0 ? lessers.peek() : **null**;

}

**public** T delMedian()

{

**if**(size > 0)

{

T ret = lessers.dequeue();

size--;

balance();

**return** ret;

}

**return** **null**;

}

**public** **static** **void** main(String[] args) {

DynamicMedianFinding<Integer> dmf = **new** DynamicMedianFinding<Integer>();

dmf.insert(10);

System.*out*.println(dmf.median());

dmf.insert(20);

System.*out*.println(dmf.median());

dmf.insert(20);

System.*out*.println(dmf.median());

dmf.insert(1);

System.*out*.println(dmf.median());

dmf.insert(100);

dmf.insert(100);

dmf.insert(100);

System.*out*.println(dmf.median());

dmf.insert(100);

System.*out*.println(dmf.median());

dmf.insert(100);

System.*out*.println(dmf.median());

System.*out*.println(dmf.delMedian());

System.*out*.println(dmf.delMedian());

System.*out*.println(dmf.delMedian());

}

}

2.4.31 Fast insert. Develop a compare-based implementation of the MinPQ API such that insert uses ~ log log N compares and delete the minimum uses ~2 log N compares. Hint : Use binary search on parent pointers to find the ancestor in swim().

**private** **void** fastSwim(**int** k)

{

T item = theHeap[k];

//initial: low = k's direct parent, high = the root;

**int** low = 1, high = floorLg(k);

**while**(low <= high)

{

**int** mid = low + (high - low)/2;

T midItem = theHeap[k / ( 1<< mid)];

**int** cmp = item.compareTo(midItem);

**if**(cmp < 0)

low = mid + 1;

**else**

high = mid - 1;

}

**int** insertPos = k / (1<< (low - 1));

**while**(k > insertPos)

{

theHeap[k] = theHeap[k / 2];

k /= 2;

}

theHeap[k] = item;

}

2.4.32 Lower bound. Prove that it is impossible to develop a compare-based implementation of the MinPQ API such that both insert and delete the minimum guarantee to use ~N log log N compares.

2.4.33 Index priority-queue implementation. Implement the basic operations in the index priority-queue API on page 320 by modifying Algorithm 2.6 as follows: Change pq[] to hold indices, add an array keys[] to hold the key values, and add an array qp[] that is the inverse of pq[] — qp[i] gives the position of i in pq[] (the index j such that pq[j] is i). Then modify the code in Algorithm 2.6 to maintain these data structures. Use the convention that qp[i] = -1 if i is not on the queue, and include a method contains() that tests this condition. You need to modify the helper methods exch() and less() but not sink() or swim().

2.4.34 Index priority-queue implementation (additional operations). Add minIndex(), change(), and delete() to your implementation of Exercise 2.4.33.

2.4.35 Sampling from a discrete probability distribution. Write a class Sample with a constructor that takes an array p[] of double values as argument and supports the following two operations: random()—return an index i with probability p[i]/T (where T is the sum of the numbers in p[])—and change(i, v)—change the value of p[i] to v. Hint: Use a complete binary tree where each node has implied weight p[i]. Store in each node the cumulative weight of all the nodes in its subtree. To generate a random index, pick a random number between 0 and T and use the cumulative weights to determine which branch of the subtree to explore. When updating p[i], change all of the weights of the nodes on the path from the root to i. Avoid explicit pointers, as we do for heaps.