Heaps and Heapsort

Priority Queues

A *priority queue* is a data structure which contains any finite number of data and has the following operators.

1. Insert any datum.
2. Delete the datum of highest priority, and return that datum.
3. Determine whether the structure is empty.
4. Initialize an empty structure.

Priority can be defined in these three ways.

1. The most recently inserted item has priority, in which case the priority queue is called a *stack*. In this case, insert is call *push* and deletion is called *pop*. Originally, this structure was called a *push-down stack*.
2. The least recently inserted item has priority, in which case the priority queue is called a *queue*. Insert is called *enqueue* and deletion is called *dequeue*.
3. Data have an instrinsic priority measure, such as larger or smaller, in which case the priority queue is called a *heap*. If the larger datum has priority, it is called a *max-heap*, if smaller, *min-heap*. Implementation is the same, regardless of the definition of priority.

Array Implementation of Binary Tree Implementation of Heap

We now assume that our heaps are max-heaps.[1] A heap can be implemented as an *almost complete* binary tree.[2] The means that every level of the tree above the last level is filled, and the last level is left-justified. The tree must be in *heap order*, meaning that $x \leq y$ if x is a left or right child of y. The tree can be represented as (n, H), for $H[1], \ldots H[n]$, where the tree structure is defined as follows:

1. The root is $H[1]$.
2. For any $i \geq 2$, the parent of $H[i]$ is $H\left[\, \lfloor \frac{i}{2} \rfloor \,\right]$.

$H[i]$ could have two, one, or no children. The indices of those children are $2i$ and $2i+1$ if $2i+1 \leq n$, just $2i$ if $2i = n$, and none if $2i > n$.

[1] If we use min-heaps, everything is the same, except that we swap the words "smaller" and "larger."
[2] Not a binary search tree.

Insertion

For a heap H and a datum x, we implement insert(H, x) as follows.

$n++$
$H[n] = x$
Bubbleup(H, n)

The recursive procedure bubbleup (also called sift-up) moves a datum up in the tree to its proper place. Here is pseudocode for bubbleup(H, i). We write simply bubbleup(i) if H is understood.

If $(i > 1$ and $H[i] > H\big[\,\lfloor \frac{i}{2} \rfloor\,\big]$
  {
    Swap $\big(H[i], H\big[\,\lfloor \frac{i}{2} \rfloor\,\big]\big)$
    Bubbleup$\big(\lfloor \frac{i}{2} \rfloor\big)$
  }

Deletemax

If our heap H is not empty, the function deletemax(H) returns the maximum datum $H[1]$, and deletes it from H. We implement deletemax(H) as follows.

Return $H[1]$
Swap $(H[1], H[n])$
$n--$
If $(n > 0)$
  Bubbledown$(H, 1)$

The recursive procedure bubbledown(H, i), or siftdown(H, i) which we write as simply bubbledown(i) if H is understood, moves $H[i]$ down to its proper level.[3] Here is the pseudocode for bubbledown(i).

If $(2i \leq n)$
  {
    $j =$ index of the larger child of $H[i]$
    If $(H[j] > H[i])$
      {
        Swap $(H[i], H[j])$
        Bubbledown(j)
      }
  }

We define deletemin on a min-heap similarly.

[3] We sometimes write bubbledown$(H[i])$ instead of bubbledown(i).

Heapification

An array is put into heap order in by executing *heapify*. The following pseudocode heapifies an array H, by what I call "bottom-up bubbledown."

For(i from n downto 1)
  Bubbledown(i)

Note that bubbledown(i) is vacuous if $i > n/2$. The worst case time complexity of heapify is $\Theta(n)$, the same as that of the following code fragment, which models Heapify.

```
for(int i = n; i > 0; i--)
  for(int j = i; j <= n; j = 2*j)
    cout << "Hello"
```

Selection Sort

Given an array $A[1], A[2], \ldots A[n]$, we first identify the smallest entry and swap it with $A[1]$. Then, for each i from 2 to n, we swap the smallest entry of the subarray $A[i] \ldots A[n]$ with $A[i]$. The array is now sorted.

Alternatively, we could also sort in reverse order. The first step is to swap the largest entry with $A[n]$, then the next largest with $A[n-1]$, and so forth. If we use linear search for each subarray, each step takes $O(n)$ time, giving us an overall time complexity of $O(n^2)$ for selection sort.

Heapsort

Heapsort is a fast version of selection sort. Let $A[1], A[2], \ldots A[n]$ be the initial array. We first give an easy to explain version that requires workspace, that is, auxiliary memory.

1. Initialize H be an empty min-heap.
2. For i from 1 to n, insert($H, A[i]$).
3. For i from 1 to n, $A[i]$ = deletemin(H).

In our description below, we sort in reverse order, meaning we choose $A[n]$ to be the largest entry, then $A[n-1]$ the second largest, and so forth. Instead of using linear search, we maintain a max-heap structure on the as yet unsorted subarray. At each step, we identify the largest entry of the subarray in $O(\log n)$. time, yielding an overall time complexity of $O(n \log n)$ for heapsort.

We now describe the phases and steps of heapsort in more detail. Let $A[1], A[2], \ldots A[n]$ be the initial array.

- The first phase is to change A to a max-heap. We call this phase *heapify*, or *heap construction*. The time complexity of this phase is $\Theta(n)$.

- The second phase is a loop of length n. By a clever implementation first given by Robert Floyd (of Floyd-Warshall fame) no extra memory is needed to store data. During this phase, the array A is the concatenation of a max-heap and a sorted array. After t steps of this loop, the subarray $A[1] \dots A[n-t]$, is a maxheap, where $A[1]$ is the maximum of that subarray, and $A[i]$ is in its final position for each $n - t < i \leq n$. Each step, deletemax is executed. The first and last entries of the heap are swapped, the heap is decremented by 1, and bubbledown(1) is executed. Thus, the heap shrinks and the sorted subarray expands at each step.

We now give a more detailed explanation of the phases. The first phase, called *heapify*, is to place the items of A into a maxheap. Here is an example. As usual, we implement the heap as an almost complete binary tree stored in an array in level order. Suppose our file is THJESYIFWZGPZBRN. We simply start with an array containing those items, then execute bubbledown at each position, with decreasing indices starting from the middle. Here are the steps of heapify. (Remember, it's a maxheap.)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
T	H	J	E	S	Y	I	F	W	Z	G	P	X	B	R	N	
T	H	J	E	S	Y	I	N	W	Z	G	P	X	B	R	F	bubbledown(F)
T	H	J	E	S	Y	R	N	W	Z	G	P	X	B	I	F	bubbledown(I)
T	H	J	E	S	Y	R	N	W	Z	G	P	X	B	I	F	bubbledown(Y) nothing happens
T	H	J	E	Z	Y	R	N	W	S	G	P	X	B	I	F	bubbledown(S)
T	H	J	W	Z	Y	R	N	E	S	G	P	X	B	I	F	bubbledown(E)
T	H	Y	W	Z	J	R	N	E	S	G	P	X	B	I	F	bubbledown(J)
T	H	Y	W	Z	X	R	N	E	S	G	P	J	B	I	F	bubbledown(J), cont
T	Z	Y	W	H	X	R	N	E	S	G	P	J	B	I	F	bubbledown(H)
T	Z	Y	W	S	X	R	N	E	H	G	P	J	B	I	F	bubbledown(H), cont
Z	T	Y	W	S	X	R	N	E	H	G	P	J	B	I	F	bubbledown(T)
Z	W	Y	T	S	X	R	N	E	H	G	P	J	B	I	F	bubbledown(T), cont

Heap order is achieved.

The second, and longer, phase is the selection sequence. If we used the above example, it would be too long, so I'll start with something smaller.

During the second phase, the array has two parts. The left part is the maxheap, while the right part (shown with bold letters) is already sorted. Each iteration consists of the swap of the maximum item, in position 1, to the item in the last position of the heap, followed by restoration of heap order. The item that was in position 1 becomes the newest item in the sorted portion, which grows by one. The heap is decremented. Then the item in position 1, which is out of place, bubbles down until heap order is restored. After n iterations, the heap is empty and the sorted portion is the entire array.

Here is an entire computation of heapsort with initial array (H,U,B,J,E,R,P,W,Q). During the second phase, the members of the sorted subarray are written in boldface.

1	2	3	4	5	6	7	8	9	
H	U	B	J	E	R	P	W	Q	
H	U	B	W	E	R	P	J	Q	bubbledown(J)
H	U	R	W	E	B	P	J	Q	bubbledown(B)
H	W	R	U	E	B	P	J	Q	bubbledown(U)
W	H	R	U	E	B	P	J	Q	bubbledown(H)
W	U	R	H	E	B	P	J	Q	bubbledown(H), continued
W	U	R	Q	E	B	P	J	H	bubbledown(H), continued
W	U	R	Q	E	B	P	J	H	bubbledown(H) heap order is first established
H	U	R	Q	E	B	P	J	**W**	swap W and H
U	H	R	Q	E	B	P	J	**W**	bubbledown(H)
U	Q	R	H	E	B	P	J	**W**	bubbledown(H), continued
U	Q	R	J	E	B	P	H	**W**	bubbledown(H), continued, heap order restored
H	Q	R	J	E	B	P	**U**	**W**	swap U and H
R	Q	H	J	E	B	P	**U**	**W**	bubbledown(H)
H	Q	P	J	E	B	**R**	**U**	**W**	swap R and H
Q	H	P	J	E	B	**R**	**U**	**W**	bubbledown(H)
Q	J	P	H	E	B	**R**	**U**	**W**	bubbledown(H), continued, heap order restored
B	J	P	H	E	**Q**	**R**	**U**	**W**	swap Q and B
P	J	B	H	E	**Q**	**R**	**U**	**W**	bubbledown(B), heap order restored
E	J	B	H	**P**	**Q**	**R**	**U**	**W**	swap P and E
J	E	B	H	**P**	**Q**	**R**	**U**	**W**	bubbledown E
J	H	B	E	**P**	**Q**	**R**	**U**	**W**	bubbledown E, continued, heap order restored
E	H	B	**J**	**P**	**Q**	**R**	**U**	**W**	swap E and J
H	E	B	**J**	**P**	**Q**	**R**	**U**	**W**	bubbledown(E), heap order restored
B	E	**H**	**J**	**P**	**Q**	**R**	**U**	**W**	swap H and B
E	B	**H**	**J**	**P**	**Q**	**R**	**U**	**W**	bubbledown(B), heap order restored
B	**E**	**H**	**J**	**P**	**Q**	**R**	**U**	**W**	swap E and B
B	**E**	**H**	**J**	**P**	**Q**	**R**	**U**	**W**	sorted