

1 Data Structures

Sets manipulated by algorithms often grow, shrink or change over time, so we call these *dynamic sets*. To represent such dynamic sets and to manipulate them efficiently, we need to design good *data structures*.

Over the next several classes we will look at several important data structures. We start with *priority queue*.

2 Priority Queue

A *priority queue* maintains a dynamic set S of elements from a totally ordered set, so that the following operations can be performed efficiently:

- $\text{INSERT}(S, x)$: inserts element x , whose key value is given.
- $\text{MINIMUM}(S)$: returns a pointer to an element in S with minimum key.
- $\text{EXTRACT-MIN}(S)$: returns an element with minimum key in S , and deletes that element from the priority queue.

We often use an *augmented* priority queue that supports the following two operations, in addition to the three operations listed above.

- $\text{DECREASE-KEY}(S, x, k)$: given a pointer to element x in S , updates the key value of element x to $\min(\text{key}(x), k)$.
- $\text{DELETE}(S, x)$: given a pointer to element x in S , removes x from S .

(One can also define a ‘max-priority-queue’ in which the ‘Minimum’ and ‘min’ in the above operations are replaced by ‘Maximum’ and ‘max’.)

2.1 Binary Heap

A *heap* structure on n elements is an ordered binary tree on n nodes that is as complete as possible; if the lowest level is not full, then it is filled in completely from the left end. For a node v in this binary tree, we denote its left child by $left(v)$, its right child by $right(v)$, and its parent by $parent(v)$ or $p(v)$. We use NIL to denote that a field is empty.

We represent a heap structure on n elements by an array $A[1..n]$. Then, $A[1]$ represents the element at the root node of the heap, and successive elements in A are located at nodes that are located left-to-right, level-by-level in the binary tree. With this numbering of the nodes of the heap we can show that

$$left(i) = 2i, \quad right(i) = 2i + 1, \quad \text{and } p(i) = \lfloor i/2 \rfloor$$

A heap structure maintains elements from a totally ordered set in either *max-heap-order* or *min-heap-order*. We will consider max-heap-order here, since that is the ordering used in heapsort. However, it should be noted that the heap is a commonly-used data structure, and in most data structure applications we use min-heap-order. We often simply say ‘heap’ when we mean a ‘min-heap.’

The max-heap-order property states that for every element $A[i]$ in the heap, $i > 1$,

$$A[i] \leq A[p(i)]$$

(Analogously, in a min-ordered heap, $A[i] \geq A[p(i)]$, for all $i > 1$.)

Recall that the *depth* of a node in a rooted tree is the length of the path from the root to that node; the *height* of the node is the length of a longest path from the node to a descendant leaf; the *height of the tree* is the height of its root. (*Please read up Appendix B.5 in your textbook to learn about standard properties and facts about trees.*)

Fact. A binary tree on n nodes that is as complete as possible has height $\lfloor \log n \rfloor$.

(Note: A *heap* is a data structure, and is not related in any way to garbage-collected storage.)

2.1.1 Binary Heap as Priority Queue

We can implement a priority queue efficiently by maintaining the set as a *min-heap*, which we will call simply a ‘heap’ from now onward. We create an array $A[1..r]$, where r is large enough to hold the elements in the set S as S changes over time. If we denote the size of the current set S as n , then the elements of S will be stored in the initial subarray $A[1..n]$ in min-heap order, and we will use *heapsize* to denote the current size of the set. The corresponding priority queue operations are

- $\text{HEAP-MINIMUM}(A)$
- $\text{HEAP-INSERT}(A, \textit{key})$
- $\text{HEAP-EXTRACT-MIN}(A)$
- $\text{HEAP-DECREASE-KEY}(A, i, \textit{key})$
- $\text{HEAP-DELETE}(A, i)$

Clearly $\text{HEAP-MINIMUM}(A)$ can be returned in one step — the minimum element is $A[1]$. It is also quite straightforward to see that HEAP-INSERT and HEAP-DECREASE-KEY can be supported in $O(\log n)$ time by repeatedly exchanging the current element with its parent until the heap order is restored (where $n = \textit{heapsize}$).

It is not too difficult to come up with an $O(\log n)$ time method to support $\text{HEAP-EXTRACT-MIN}(A)$ and $\text{HEAP-DELETE}(A, i)$. This is usually implemented using the following procedure MAX-HEAPIFY :

$\text{MAX-HEAPIFY}(A[1..n], k)$

Input. An array $A[1..n]$ of elements from a totally ordered set and an index k ($1 \leq k \leq n$) with the property that the elements in the subtree rooted at $\textit{left}(k)$ and the elements in the subtree rooted at $\textit{right}(k)$ each satisfy the max-heap property.

Output. The elements in array A corresponding to the subtree rooted at node k rearranged so that all elements in this subtree satisfy the max-heap property.

$i := k$

while i is not a leaf and $A[i]$ is not larger than $A[\textit{left}(i)]$ and $A[\textit{right}(i)]$ **do**

 let $A[j]$ be the larger of $A[\textit{left}(i)]$ and $A[\textit{right}(i)]$

 exchange $A[i]$ and $A[j]$

$i := j$

end while

We prove correctness of MAX-HEAPIFY using the following loop invariant for the **while** loop.

At the start of each iteration of the **while** loop, every node i' in the subtree rooted at k satisfies the max-heap property other than k and possibly $\text{left}(i)$ and $\text{right}(i)$, and the subtree rooted at k contains the same collection of elements as in the input array.

Running time of MAX-HEAPIFY. Each iteration of the **while** loop takes constant time. Also, in each iteration, the depth of node i in the subtree rooted at k increases by 1. Hence, if the height of node k is h_k , then the number of iterations is no more h_k , and the running time is $O(h_k)$.

Also, since height of any node k is \leq height of node 1, the running time is $O(\log n)$.

Back to HEAP-DELETE and HEAP-EXTRACT-MIN. To implement $\text{HEAP-DELETE}(A, i)$ we set $A[i] := A[n]$, we decrement *heapsize* to $n - 1$ and then call $\text{MAX-HEAPIFY}(A, i)$.
(Why is this correct?)

We implement HEAP-EXTRACT-MIN the same way, except that we first copy $A[1]$ so that we can return it, and then we use $i = 1$ in the procedure for HEAP-DELETE .

2.2 Heapsort

Since a binary heap on n elements can support all of the priority queue operations in $O(\log n)$ time, it leads to the following simple $O(n \log n)$ time algorithm: Given a collection of n elements to be sorted, insert the n elements into a binary heap using n HEAP-INSERT operations, and then list them in sorted order with n HEAP-EXTRACT-MIN operations.

Heapsort is a very efficient sorting algorithm based on the binary heap. The main difference with the simple method described above is that it is *in-place*, and it achieves this using a MAX-HEAP with a nice method BUILD-MAX-HEAP , which takes the input unsorted array $A[1..n]$ and rearranges the elements by repeatedly invoking MAX-HEAPIFY to form a binary heap; in addition to being in-place, this routine runs in $O(n)$ time. Using a MAX-HEAP also allows each element extracted with the HEAP-EXTRACT-MAX operation to be placed directly in the correct sorted position in the input array.

You can read up the details in your textbook.

2.3 DJP and Dijkstra's Algorithms

It is not difficult to see that using a binary heap we can implement the DJP minimum spanning tree algorithm and DIJKSTRA'S SSSP algorithm to run in $O(m \log n)$ time where n is the number of vertices and m is the number of edges in the input graph.