# <2T>

# Heapsort Description

The ***(binary) heap*** data structure is an array object that we can view as a

nearly complete binary tree (see Section B.5.3), as shown in Figure 6.1. Each

node of the tree corresponds to an element of the array. The tree is completely

filled on all levels except possibly the lowest, which is filled from the

left up to a point. An array A that represents a heap is an object with two attributes:

A.*length*, which (as usual) gives the number of elements in the array, and

A.*heap*-*size*, which represents how many elements in the heap are stored within

array A. That is, although A[1..A:*length*] may contain numbers, only the elements

in A[1..A.*heap*-*size*], where 0 <= A.*heap*-*size* <= A.*length*, are valid elements

of the heap. The root of the tree is A[1], and given the index i of a node, we

can easily compute the indices of its parent, left child, and right child:

Parent(i)  
 return[i/2]

Left(i)  
 return 2i

Right  
 return 2i + 1

There are two kinds of binary heaps: max-heaps and min-heaps. In both kinds,

the values in the nodes satisfy a ***heap property***, the specifics of which depend on

the kind of heap. In a ***max-heap***, the ***max-heap property*** is that for every node i

other than the root,

A[PARENT(i)] => A[i] .  
that is, the value of a node is at most the value of its parent. Thus, the largest

element in a max-heap is stored at the root, and the subtree rooted at a node contains  
values no larger than that contained at the node itself. A ***min-heap*** is organized in

the opposite way; the ***min-heap property*** is that for every node i other than the

root,

A[PARENT(i)] <= A[i]   
The smallest element in a min-heap is at the root.

The MAX-HEAPIFY procedure, which runs in O(lg n) time, is the key to maintaining

the max-heap property.

! The BUILD-MAX-HEAP procedure, which runs in linear time, produces a maxheap

from an unordered input array.

! The HEAPSORT procedure, which runs in O(n lg n) time, sorts an array in

place.

! The MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY,

and HEAP-MAXIMUM procedures, which run in O(lg n) time, allow the heap

data structure to implement a priority queue.

# Heapsort Pseudo-Code

Max-Heapify(A, i)  
1 I = Left(i)  
2 r = Right(i)  
3 if I <= A.heap-size and A[l] > A[i]  
4 largest = j  
5 else largest = i  
6 if r <= A.heap-size and A[r] > A[largest]  
7 largest = r  
8 if largest != i  
9 exchange A[i] wth A[largest]  
10 Max-Heapify(A, largest)

Build-Max-Heap(A)  
1 A.heap-size = A.length  
2 for I = [A.length / 2] downto 1  
3 Max-Heapify(A, i)

Heapsort(A)  
1 Build-Max-Heap(A)  
2 for I = A.length downto 2  
3 exchange A[1] with A[j]  
4 A.heap-size = A.heap-size – 1  
5 Max-Heapify(A, 1)

# Heapsort code

 def heapsort( aList ):

  # convert aList to heap

  length = len( aList ) - 1

  leastParent = length / 2

  for i in range ( leastParent, -1, -1 ):

    moveDown( aList, i, length )

  # flatten heap into sorted array

  for i in range ( length, 0, -1 ):

    if aList[0] > aList[i]:

      swap( aList, 0, i )

      moveDown( aList, 0, i - 1 )

def moveDown( aList, first, last ):

  largest = 2 \* first + 1

  while largest <= last:

    # right child exists and is larger than left child

    if ( largest < last ) and ( aList[largest] < aList[largest + 1] ):

      largest += 1

    # right child is larger than parent

    if aList[largest] > aList[first]:

      swap( aList, largest, first )

      # move down to largest child

      first = largest;

      largest = 2 \* first + 1

    else:

      return # force exit

def swap( A, x, y ):

  tmp = A[x]

  A[x] = A[y]

  A[y] = tmp