Reading Note: Chapter 6

Xitong Yang

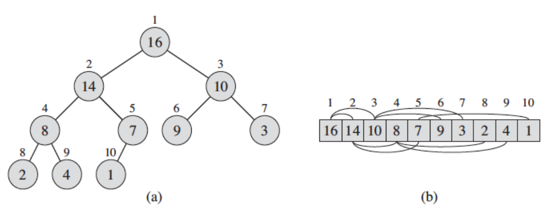
Jan.15 2016

1 Introduction

Chapter 5 gives a introduction to another sorting algorithm: **heapsort**. A new data structure, *heap*, is first introduced and its properties are explained. With this background in mind, we can see the algorithm of heapsort and its complexity in time and space.

2 Key Note

- The (binary) ***heap*** data structure is an array object that we can view as a nearly complete binary tree.
  - A *complete k-ary tree* is a k-ary tree in which all leaves have the same depth and all internal nodes have degree k. The number of leaves at depth h is k^h. Consequently, the number of internal nodes of a complete k-ary tree of height h is: $1+k+k^2+...+k^{h-1} = \sum_{i=0}^{h-1} k^i = \frac{k^h-1}{k-1}$. Thus a complete binary tree has $2^h - 1$ internal nodes.
  - An array A that represents a heap has two attributes: $A.length$ (number of elements in the array), $A.heap\text{-}size$ (represents how many elements in the heap are stored within array A)



- Given the index i of a node, we can easily compute: $PARENT(i) = \lfloor i/2 \rfloor$, $LEFT(i) = 2i$, $RIGHT(i) = 2i + 1$.
  - These procedures can be implemented as shifting operation as "macros" or "inline" procedures.

- ***Heap property***: In a *max-heap*, $A[PARENT(i)] \geq A[i]$. In a *min-heap*, $A[PARENT(i)] \leq A[i]$
- MAX-HEAPIFY: to maintain the heap property
  - Given an array A and an index i into the array (assume that the subtrees rooted at $LEFT(i)$ and $RIGHT(i)$ are max-heaps, but not sure about $A[i]$), we need to "float down" $A[i]$ if it violate the max-heap property.
  - The procedure mainly compares $A[i]$ with $A[LEFT(i)]$ and $A[RIGHT(i)]$ and swap $A[i]$ with the largest one if needed. The procedure is called recursively and the running time is $O(\lg n)$, deduced from recurrence $T(n) \leq T(2n/3) + \Theta(1)$.
- BUILD-MAX-HEAP: to build a heap
  - Begin with each of $A[(\lfloor n/2 \rfloor + 1)...n]$ as 1-element heap, then runs MAX-HEAPIFY on each of remaining ones.
  - The tight upper bound of the procedure is $O(n)$: we can build a max-heap from an unordered array in linear time.
- The **heapsort** algorithm
  - We use *max-heap* in heapsort algorithm.
  - Start by building a max-heap on $A[1...n]$. Exchange the largest element $A[1]$ with $A[n]$ and discard it from the heap. Call MAX-HEAPIFY(A,1). Loop for these procedures.
  - The algorithm takes time $O(n \lg n)$, and it is a *in place* algorithm (only a constant number of elements of the input array are ever stored outside the array).
- Heap data structure can be used as an efficient **priority queue**. For a max-priority queue, it supports following operations:
  - MAXIMUM(S): returns the element of S with the largest key. Solution: return the first element in the heap. ($\Theta(1)$ time)
  - EXTRACT-MAX(A): removes and return the largest element. Solution: similar to heapsort procedure, after popping the first element, substitute with the last one and call MAX-HEAPIFY(A,1).
  - INCREASE-MAX(S,x,k): increases the value of element x's key to the new value k. Solution: similar to INSERTION-SORT, traverses a simple path from the updated node to ward the root to find a proper place.
  - INSERT(S,x): inserts the element x into set S. Solution: insert the new element to the end of S with negative infinity key value, then call INCREASE-MAX.

3 Algorithms

- Heapsort Algorithm * (Time: $O(nlgn)$, Space: $O(1)$)

  - ```
    def MAX-HEAPIFY(A, i):
        l = LEFT(i)
        r = RIGHT(i)
        if l <= A.heap-size and A[l] > A[i]:
            largest = l
        else:
            largest = i
        if r <= A.heap-size and A[r] > A[largest]:
            largest = r
        if not largest == i:
            exchange A[i] with A[largest]
            MAX-HEAPIFY(A, largest)
    ```

  - ```
    def BUILD-MAX-HEAP(A):
        for i = A.length/2 downto 1:
            MAX-HEAPIFY(A, i)
    ```

  - ```
    def HEAPSORT(A):
        BUILD-MAX-HEAP(A)
        for i = A.length downto 2:
            exchange A[1] with A[i]
            A.heap-size -= 1
            MAX-HEAPIFY(A, 1)
    ```

* Sample codes implemented in *Codes* folder
    ```