CSc 3102: Priority Queues & Heaps

Supplementary Notes

- The ADT Priority Queue
- Heap
- HeapSort

1 The ADT Priority Queue

Definition 1. A **priority value** indicates a task's priority for completion. The item with the highest priority value is retrieved each time an item is removed from the queue.

Definition 2. A **priority queue** is an ADT that provides the following operations:

1. Create an empty priority queue.
2. Determine whether a priority queue is empty.
3. Insert a new item into a priority queue.
4. Retrieve and delete the item with the highest priority value.

1.1 Implementation Strategies

1. Sorted linear array-based implementation: highest priority value is at the end of the array. Insertion involves binary search and shifting. What are the worst-case time complexities of *insert* and *delete*?
2. Sorted linear linked-list based implementation: highest priority value is at the head of the list. Insertion however involves list traversal. What are the worst-case time complexities of *insert* and *delete*?

3. Binary search tree implementation: where is the highest priority value in the tree? What are the worst-case time complexities of *insert* and *delete*?

4. Heap array-based implementation: This implementation strategy is very often the most efficient.

2 Heap

Definition 3. A **heap** is a complete binary tree that is either empty or whose root contains a search key greater than or equal to the search key in each of its children, and whose root has heaps as its subtrees. Note that a heap is more weakly ordered than a binary search tree. Why?

Definition 4. When a heap contains the item with the largest search key in its root, it is called a **maxheap**.

Definition 5. A **minheap** places an item with the smallest search key in its root.

We now give pseudo-code descriptions for basic maxheap algorithms. For related minheap algorithms simply reverse the sense of the inequalities used when comparing two elements of the heap to determine whether a swap is necessary; that is, $<$ becomes $>$, vice versa.

Listing 1: Inserting into a Heap

```
1   Algorithm: maxHeapInsert(heap, newItem)
2   { heap - an array
3     newItem - the item to be inserted.}
4      heap[heapSize] <- newItem
5      place <- heapsize
6      parent <- (place -1)/2
7      while parent >= 0 and heap[place] > heap[parent]
8         swap(heap,place,parent)
9         place <- parent
10        parent <- (place - 1)/2
11     heapSize <- heapSize + 1
```

Listing 2: Deleting from a Heap

```
1     Algorithm: maxHeapRebuild(heap,root,heapSize)
2     {Reheapifies: restores the max-heap property
3      heap - an array representing a binary heap
4      root - the root of a sub-tree of the binary heap
5      heapSize - size of the heap}
6       if root is not a leaf
7          child <- 2 * root + 1
8          if root has a right child
9             if heap[child + 1] > Heap[child])
10               child <- child + 1
11         if Heap[root] < Heap[child]
12            swap(heap,root,child)
13            maxHeapRebuild(heap,child,heapSize)
14
15 Algorithm: maxHeapDelete(heap)
16 {Removes the item at the top of the heap
17  heap - an array representing the binary heap}
18    root <- heap[0]
19    heap[0] <- heap[heapSize-1]
20    heapSize  <- heapSize - 1
21    maxHeapRebuild(heap,0,heapSize)
22    return root
```

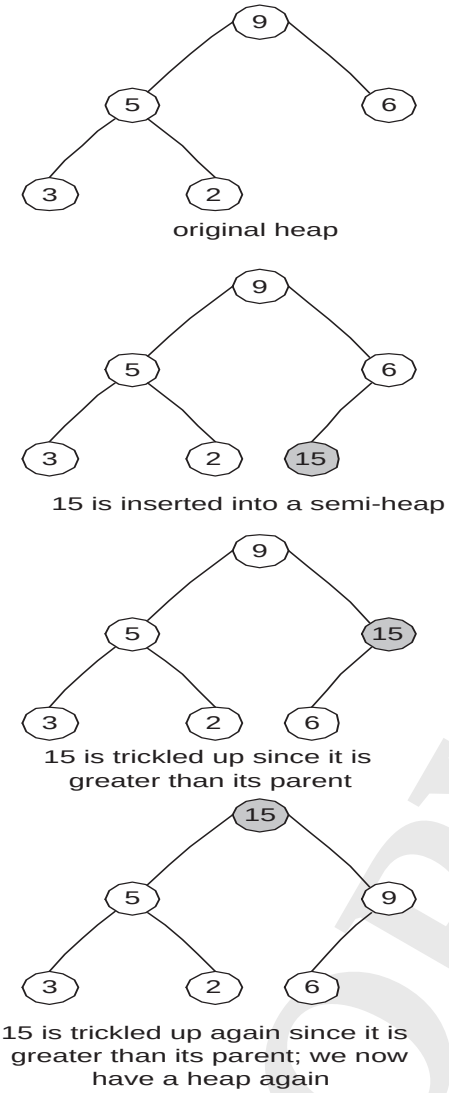

Figure 1: An illustration of inserting 15 into a maxheap

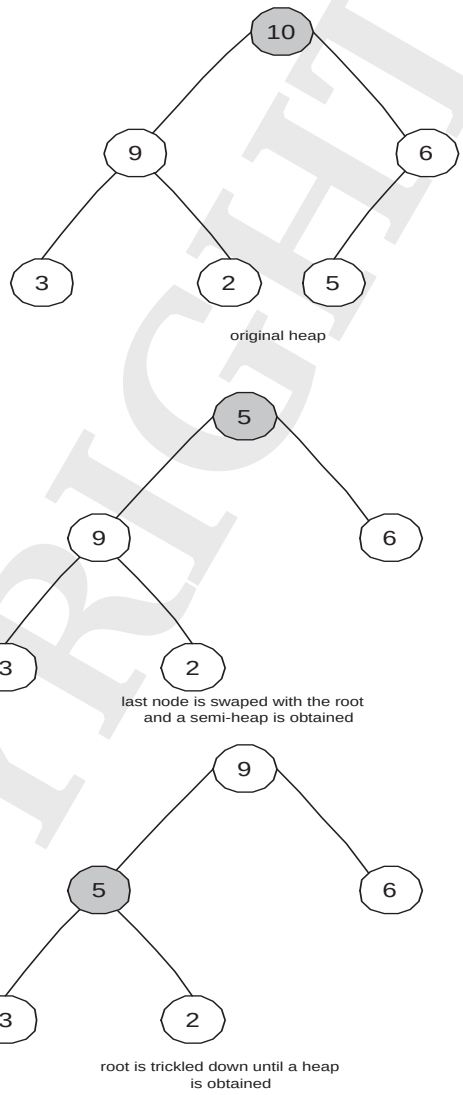

Figure 2: An illustration of how an item is removed from a maxheap

2.1 Analysis of Deletion and Insertion

Since the height of the complete binary tree is always $\lfloor \lg n \rfloor$, maxHeapInsert is also $O(\lg n)$. Convince yourself that in maxHeapDelete there are $3\left(\lfloor \lg\left(n-1\right) \rfloor + 1\right)$ data moves. So maxHeapDelete is $O(\lg n)$.

Listing 3: Heap Sort

```
1    heapSort(A, N)
2    {build heap from an array A}
3    for i <- N-1 to 0
4       maxHeapRebuild(A,i,N)
5    {the array is now 'heapified'}
6    last <- N-1
7    for j <- 1 to N
8       swap(A,0,last)
9       last <- last - 1
10      maxHeapRebuild(A,0,last+1)
```

heapSort is very efficient. In fact it is order optimal. In both the worst and average cases it is $O(n \lg n)$. It has a further advantage in that it is in-place. Although quicksort has the same average-case performance, its worst-case performance is $O(n^2)$. heapSort has the same worst and average case performance as mergeSort but unlike mergeSort it is in-place. In spite of quickSort's poor worst-case performance it is the preferred sorting algorithm because in practice, it performs very efficiently on random arrays. When implementing a priority queue in which the keys are not all distinct we may implement a heap with a queue at each node. In this way, items with identical keys are removed on a FIFO basis.