Chapter 11-2, Lecture Notes

The ADT Priority Queue: A Variation of the ADT Table

The ADT table would be used for databases. However, there is another use for the ADT Table: Priority Queue. Imagine an emergency room (ER). The hospital staff would create a record for each patient and determine when they receive treatment. The ADT table could put the patients in alphabetic order or numerical order by ID. Then a queue would be used to treat patients in order of arrival. However, that simple scheme would not work well. Ms Zither with acute appendicitis would have to wait for Mr Able with a splinter. The next available doctor should treat the patient with the highest priority.

Another example of use of priorities would be:

- Send a birthday card to Aunt Mabel
- Start the research paper for world history
- Finish reading chapter 11 of Walls and Mirrors
- Make plans for Saturday nigh

You can definitely prioritize the above (reading the chapter, of course).

A "priority value" indicates, for example, a patient's priority for treatment or a task's priority for completion. What values should you use? There are many ways to do it. A simple ranking system would be to have values of 1 to 10. Let's arbitrarily decide the largest priority value means the highest priority. That would be used in the "Priority Queue".

ADT Priority Queue Operations

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

Operation Contract for the ADT Priority Queue

PQItemType is the type of the items stored in the priority queue

pqIsEmpty():boolean {query}

Determines whether this priority queue is empty

pqInsert(in newItem:PQItemType) throw PQException

Inserts newItem into this priority queue. THows PQException is the priority queue is full pqDelete(out priorityItem:PQItemType) throw PQException

Retrieves into priorityItem and then deletes the item in this priority queu with the highest priority value. Throws PQException if the queue is empty

The above looks like a regular queue except that pdDelete does not necessarily retrieve and delete the first item but rather the first item that is the highest priority.

Figure 11-9 on page 613 shows implementation of the priority queue. With the array, you have the familiar problem of inserting in the correct place and deleting from the correct place and shifting the values. Linked list would have the familiar problem of having to find the correct node. However, the binary tree implementation has some advantages: the highest priority will ALWAYS be the right most. With the binary search tree and that the ADT priority queue primarily does inserts and deletes, the shape of the tree could suffer.

Another form of binary tree would be the "heap". It is similar to the binary search tree but is different in two important ways: 1) the heap is ordered in a weaker way and 2) heaps are always complete binary trees.

A heap is a complete binary tree:

1. That is empty – or

2a. Whose root contains a search key greater than or equal to the search key in each of its children, and

2b. Whose root has heaps as its subtrees

A heap with the item of largest search key is "maxheap" while a heap with the item of smallest search key is "minheap"

Operational contract for the ADT Heap

HeapItemType is the type of the items stored in the heap

heapIsEmpty():boolean{query}

Determines whether this heap is empty

heapInsert(in newItem:HeapItemType) throw HeapException

Inserts newItem into this heap. Throws HeapException if the heap is full

heapDelete(out rootItem:HeapItemType) throw HeapException

Retrieves and then deletes this heap's root item. This item has the largest search key. Throws HeapException if the heap is empty.

Since the heap is a complete binary tree, you can use the array based implementation that figure 11-11 shows on page 615.

The array-based implementation of a heap would have:

- items: an array of heap items
- size: an integer equal to the number of items in the heap

When discussing a heap, let's assume the items are integers.

"heapDelete" would delete the node with the largest value. Where is that? At the root! So the first step would be:

```
// return the item in the root
rootItem = items[0]
```

The problem removing the root is that you have two disjoint heaps as figure 11-2a on page 616 shows. So to turn it back into a heap (or "reheap"), you take the last node, make it the root and remove the last root, like so:

```
// copy the item from the last node into the root
items[0]= items[size-1]

// remove the last node
--size
```

However, the above does NOT create a heap (not necessarily, at least). You have what is called a "semiheap". To rebuild the heap, you could do a "trickle down" until it reaches a node that will not be out of place (i.e. value is greater than its children). To do this, you take the root (can be a root of a subtree) and compare it to its children. If a child has a greater value, you exchange the value of the largest child with the parent. Call the routine recursively with the largest child to ensure you have the heap correct. The following pseudocode shows it:

```
heapRebuild(inout items:ArrayType, in root:integer, in size:integer)
// Converts a semiheap rooted at index root into a heap

        // Recursively trickle the item at index root down to its proper position
        // by swapping it with its larger child, if the child is larger than the item.
        // If the item is at a leaf, nothing needs to be done.

        if (the root is not a leaf)
        {       // root must have a left child
                child = 2 * root + 1     // left child index

                if (the root has a right child)
                {       rightChild = child + 1 // right chld index
                        if (items[rightChild].getKey() > items[child].getKey())
                                child = rightChild // larger child index
                } // end if

        // if the item in the root has a smaller search key than the search key of
        // the item in the larger child, swap items
        if (items[root].getKey() < items[child].getKey())
        {       Swap items[root] and items[child]
                heapRebuid(items, child, size)
        } // end if
} // end if

// else root is a leaf, so you are done
```

Figure 11-14 shows the heapRebuild's recursive calls. How you can use it in the heapDelete operation:

```
// return the item in the root
rootItem = items[0]

// copy the item from the last node into the root
items[0]= items[size-1]

// remove the last node
--size

// transform the semiheap back into a heap
heapRebuild(items, 0, size)
```

The heapDelete and heapRebuild are fairly efficient. With heapRebuild, you would, at maximum, have to travel down one path to reheap. The Big O would be O(log n).

The heapInsert would have the opposite strategy to the heapDelete. The new item is inserted at the bottom of the tree and it trickles up to its proper place as figure 11-15 shows. And with the array implementation, you can always tell the parent of the child being at items[(i-1]/2] which makes it handy. The pseudocode is:

```
// insert newItem into the bottom of the tree
items[size] = newItem

// trickle new item up to appropriate spot in the tree
place = size
parent = (place-1)/2
while (  (parent >= 0) and (items[place] > items[parent])  )
{       Swap items[place] and items[parent]
        place = parent
        parent = (place – 1)/2
} // end while

increment size
```

The efficiency of heapInsert is basically the same as heapDelete: O(log n).

Figure 11-15 on page 619 shows the heapInsert.