Priority Queue ADT

- A priority queue stores entries in order
  - Each entry is a pair (key, value)
- Main methods of the Priority Queue ADT
  - `insert(k, x)`
    - inserts an entry with key k and value x
  - `deleteMax()` or `deleteMin()`
    - removes and returns the entry with the largest (smallest) key
  - `max()` or `min()`
    - returns, but does not remove, an entry with the largest (smallest) key
  - `size()`
  - `isEmpty()`

Sequence-based Priority Queue

- Implementation with an unsorted list
  - `insert`: $O(1)$ since we can insert the item at the beginning or end of the sequence
  - `removeMax` and `max`: $O(n)$ since we have to traverse the entire sequence to find the largest key
- Implementation with a sorted list
  - `insert`: $O(n)$ since we have to find the place where to insert the item
  - `removeMax` and `max`: $O(1)$ since the largest key is at the beginning
Heap

• A binary tree storing keys at its nodes
• Satisfy the following properties:
  ▪ Complete binary tree: Tree is full to level \( h - 1 \) and level \( h \) is filled from the left with contiguous nodes.
  ▪ Heap ordered: for every internal node \( v \) other than the root,
    \[
    \text{Max Heap } \quad \text{key}(v) \leq \text{key}(\text{parent}(v)) \text{ or } \text{key}(v) \geq \text{key}(\text{parent}(v))
    \]
    \[
    \text{Min Heap }
    \]

Properties of Heap

• The last node of a heap
  ▪ The rightmost node of maximum depth
• Height of a heap
  ▪ A heap storing \( n \) keys has height \( O(\log n) \).
  ▪ \( 2^i \) keys at depth \( i = 0, \ldots, h - 1 \)

Example of a (Max) Heap

Heap Implementation

• Representation of heap
  ▪ All you need to implement a heap is an array heap of items and the number of items stored in the heap (heapSize).
  ▪ The items are stored in positions 0 to heapSize - 1 in the array.
  ▪ No child pointers are needed.
  ▪ The root is always stored in heap[0], if the tree is not empty.
  ▪ The left child of heap[i] is heap[2i+1].
  ▪ The right child of heap[i] is heap[2i+2].
  ▪ The parent of heap[i] is heap[floor((i-1)/2)].
Insertion

- Insert a new element at the end of the array

```
100 19 36 17 3 25 18 2 7 1 80
```

heapSize = 11

- Shift up the element if the heap-order property is violated.

```
100 19 36 17 80 25 18 2 7 1 3
```

heapSize = 11

Insertion

- After the insertion of a new key
  - The heap-order property may be violated.
  - The heap-order property should be restored by swapping the new node with its parent.
  - This procedure continues until the key reaches the root or a node whose parent has a key smaller than or equal to the key.

- Time complexity
  - A heap has height: $O(\log n)$
  - Insertion of a heap runs in $O(\log n)$ time.
Insertion Example

- Insert 15, 5, 10, 30, 20, 0, 35, and 25 into an empty heap.

1. Insert 15:
   - Result: 15

2. Insert 5:
   - Result: 5, 15

3. Insert 10:
   - Result: 10, 5, 15

4. Insert 30:
   - Result: 30, 10, 5, 15

5. Insert 20:
   - Result: 20, 30, 10, 5, 15

6. Insert 0:
   - Result: 0, 20, 30, 10, 5, 15

7. Insert 35:
   - Result: 35, 0, 20, 30, 10, 5, 15

8. Insert 25:
   - Result: 25, 35, 0, 20, 30, 10, 5, 15
Insertion Example

• Insert 15, 5, 10, 30, 20, 0, 35, and 25 into an empty heap.

15 5 10 30 20 0 35 25

30 20 35 5 15 0 10

35 20 30 5 15 0 10 25

35 20 30 25 15 0 10 5

35 20 30 25 15 0 10 5
Implementation of Insertion

```java
public void insert(int priority) {
    int i;
    if (!isFull()) {
        heap[heapSize] = priority;
        i = heapSize++;
        while ((i > 0) && (heap[(i-1)/2] < heap[i])) {
            swap(heap[i], heap[(i-1)/2]);
            i = (i-1) / 2;
        } // end while
    } // end if
} // end insert
```

Deletion

- Delete the node with the highest priority (root)

- Replace the root with the last node

- Restore heap order in the heap by shifting down
  - Repeatedly swap the node with the larger one of its children
Deletion

- Restore heap order in the heap by shifting down
  - Repeatedly swap the node with the larger one of its children

```
80  19  36  17  3  25  18  2  7  1
```

Delete Max

- Removal of the root from the heap
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node \( w \)
  - Remove \( w \)
  - Restore the heap-order property
- Shift-down
  - swaping key \( k \) along a downward path from the root
  - Find the maximum child \( c \)
  - Swap \( c \) and \( k \) if \( c > k \)
- Time complexity: \( O(\log n) \)

Implementation of Deletion

```java
public int deleteMax()
{
    int max; // Highest priority to return.
    int i; // Current position in heap.

    if (!isEmpty())
    {
        max = heap[0];
        heap[0] = heap[--heapSize];
        i = 0;

        while (((2*i+2) < heapSize) && ((heap[i] < heap[2*i+1]) ||
                 (heap[i] < heap[2*i+2])))
            Exchange heap[i] with its larger child.
        if (((2*i+1) < heapSize) && (heap[i] < heap[2*i+1]))
            Exchange heap[i] with its left child.

        return max;
    } // end deleteMax
```