Priority Queues and Heaps

Computing 2 COMP1927 17x1
Some applications of queues require items processed in order of "key" or priority rather than in order of entry (FIFO)

Priority Queues (PQueues or PQs) provide this via:
- Insert item with a given priority into PQ
- Remove item with highest priority key
  - Highest priority key may be one with smallest or largest value depending on the application

Plus generic ADT operations:
- new, drop, empty, ...
**Priority Queue Interface**

typedef struct priQ * PriQ;

// We assume we have a more complex Item type that has
// a key and a value, where the key is the priority and the
// value is the data being stored

// Core operations
PriQ initPriQ(void);
void insert(PriQ q, Item i);
// retrieve and delete Item with highest priority
Item delete(PriQ q);

// Useful operations
int sizePriQ(PriQ q);
void changePriority(PriQ q, Key k, Item i);
void deleteKey(PriQ q, Key k);
int maxSize(PriQ q);
**Comparison of Priority Queue Implementations**

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Insert</th>
<th>Delete</th>
<th>IsEmpty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorted Array</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Unsorted Array</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Sorted List</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Unsorted List</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

Can we implement **BOTH** operations efficiently?

Yes with a heap

$O(log N)$ for insert and delete

| Heap     | $O(logN)$ | $O(logN)$ | $O(1)$ |
HEAP ORDER property

Heaps can be viewed as trees with top-to-bottom heap ordering

- for all keys both subtrees are $\leq$ root
- property applies to all nodes in tree (i.e. root contains largest value in that subtree)

*Items inserted in order*  
$m \ t \ h \ q \ a \ k$

**BST**

$m$

$\begin{array}{c}
  h \\
  a \ k \ q
\end{array}$

$\begin{array}{c}
  t \\
  q \ k
\end{array}$

**Heap**

$t$

$\begin{array}{c}
  q \\
  m \ a \ h
\end{array}$

order

$\downarrow$

order
COMPLETE TREE PROPERTY

Heaps are "complete trees"

  every level is filled in before adding a node to the next level

  the nodes in a given level are filled in from left to right, with no breaks.
**Heap Implementations**

BSTs are typically implemented as linked data structures.

Heaps can be implemented as linked data structures.

- Heaps are typically implemented via arrays.
- The property of being complete makes array implementations suitable.
ARRAY BASED HEAP IMPLEMENTATION

Simple index calculations allow navigation through the tree:
- left child of node at index i is located at 2i
- right child of node at index i is located at 2i+1
- parent of node at index i is located at i/2
ARRAY BASED HEAP IMPLEMENTATION

Heap data structure:

typedef struct HeapImp {
    Item *items; // array of Items
    int nitems;  // #items in array
} HeapImp;

typedef HeapImp *Heap;
**Heap Insertion:** Bottom-up Heapify

Insertion is a two-step process

1. add new element at bottom-most, rightmost position
2. reorganise values along path to root to restore heap property
// force value at a[k] into correct position
void fixUp(Item a[], int k) {
    while (k > 1 && less(a[k/2],a[k])) {
        swap(a, k, k/2);
        k = k/2; // integer division
    }
}
**Heap Insertion**

*Items inserted in order*  
$m, t, h, q, a, k$
**Deletion with Heaps – Top-Down Heapify**

Deletion is a three-step process

To delete node at position, \( k \)

1. replace node\([k]\) by bottom-most, rightmost value
2. remove bottom-most, rightmost value
3. restore heap by reorganizing values by moving down the heap, exchanging node\([k]\) with the larger of the node’s children, stopping when node\([k]\) is not smaller than either child or bottom is reached.

![Diagram of heap deletion process](image-url)
void fixDown(Item a[], int k) {
    int done = 0;
    while (2*k <= N && !done) {
        int j = 2*k; // choose larger of two children
        if (j < N && less(a[j], a[j+1])){
            j++;
        }
        if (!less(a[k], a[j])){
            done = 1;
        }else{
            swap(a, k, j);
            k = j;
        }
    }
}
EXERCISE:

Show the construction of the max heap produced by inserting

```
HEAPS FUN
```

Show the heap after an item is deleted.
Show the heap after another item is deleted.
Heaps as Priority Queues

Heaps are typically used for implementing Priority Queues

- priorities determined by order on keys
- new items added initially at lower-most, right-most leaf
- then new item "drifts up" to appropriate level in tree
- items are always deleted by removing root (top priority)

Since heaps are dense trees, depth = \( \lfloor \log_2 N \rfloor + 1 \)

Insertion cost = \( O(logN) \), Deletion cost = \( O(logN) \)