Priority Queues and Heaps

Computing 2 COMP1927 16x1

PRIORITY

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 Possible Implementations

Implementation	**insert**	**delete**
ordered array/list	O(N)	O(1)
unordered array/list	O(1)	O(N)

Can we implement BOTH operations efficiently?

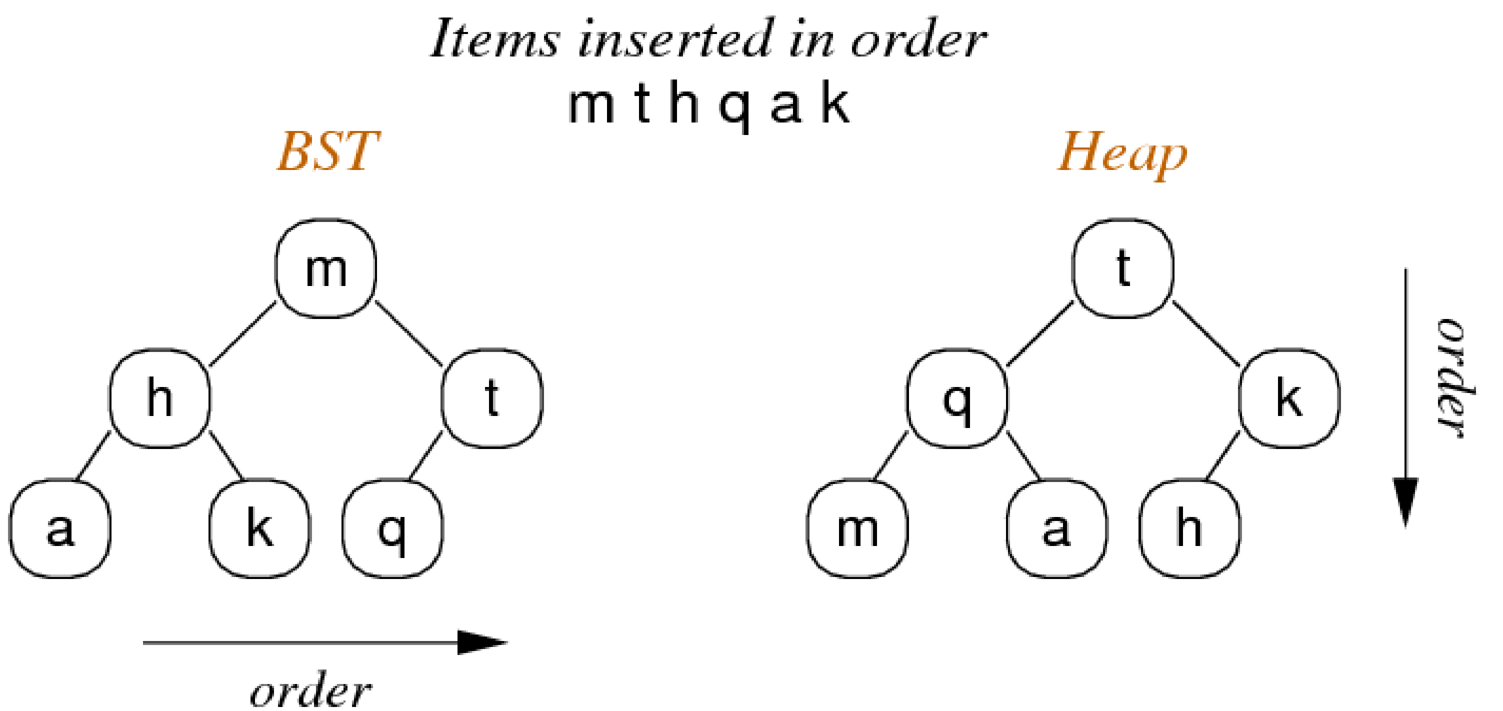
Yes with a heap

O(log N) for insert and delete

HEAP ORDER PROPERTY

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

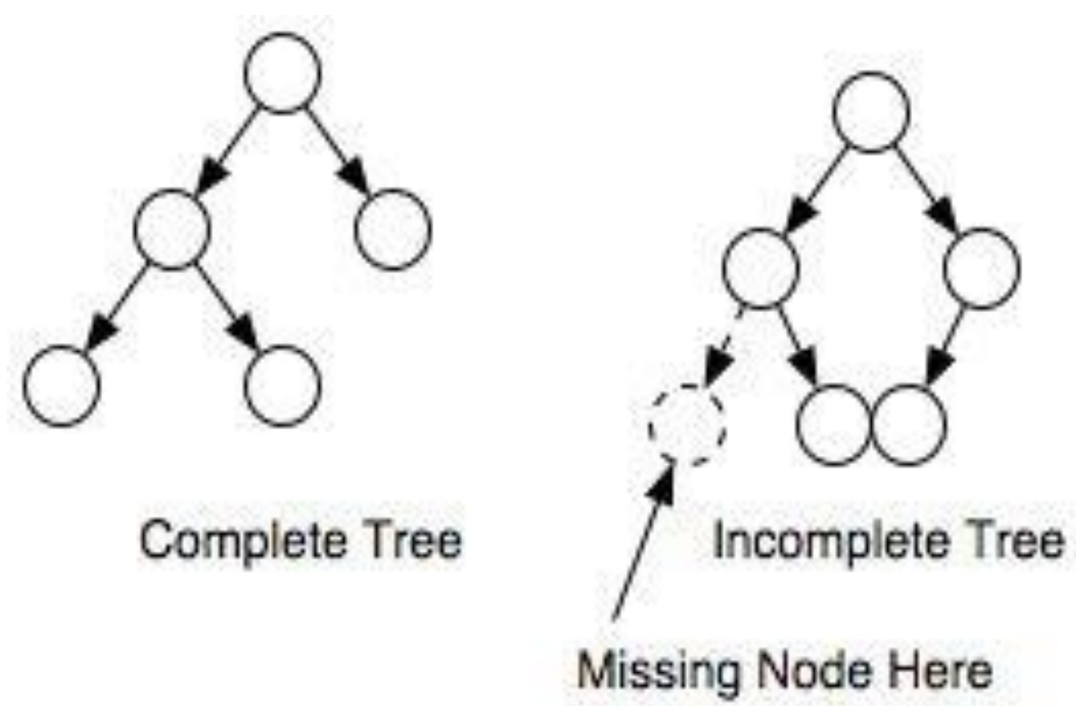
- for all keys both subtrees are ≤ root
- property applies to all nodes in tree (i.e. root contains largest value in that subtree)



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

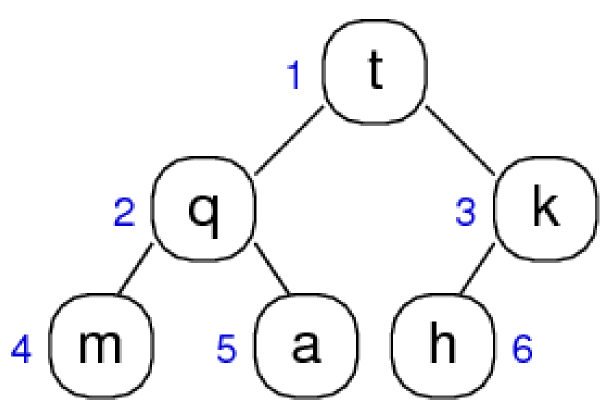
Heaps CAN be implemented as linked data structures

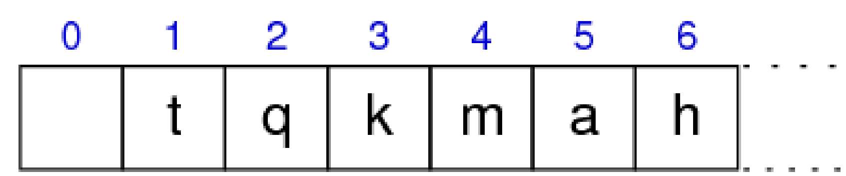
- Heaps are TYPICALLY implemented via arrays.
- The property of being **complete** makes array implementations suitable

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

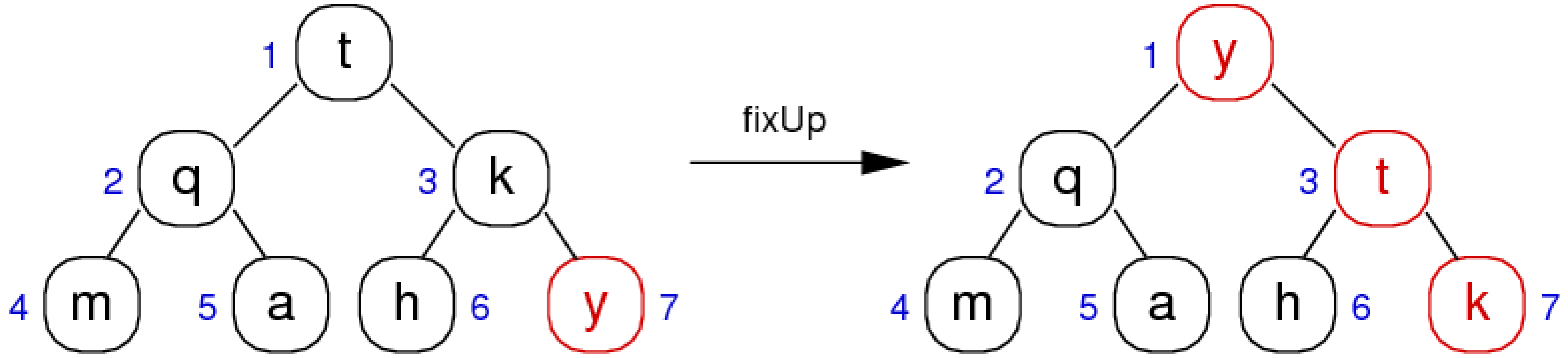




HEAP INSERTION

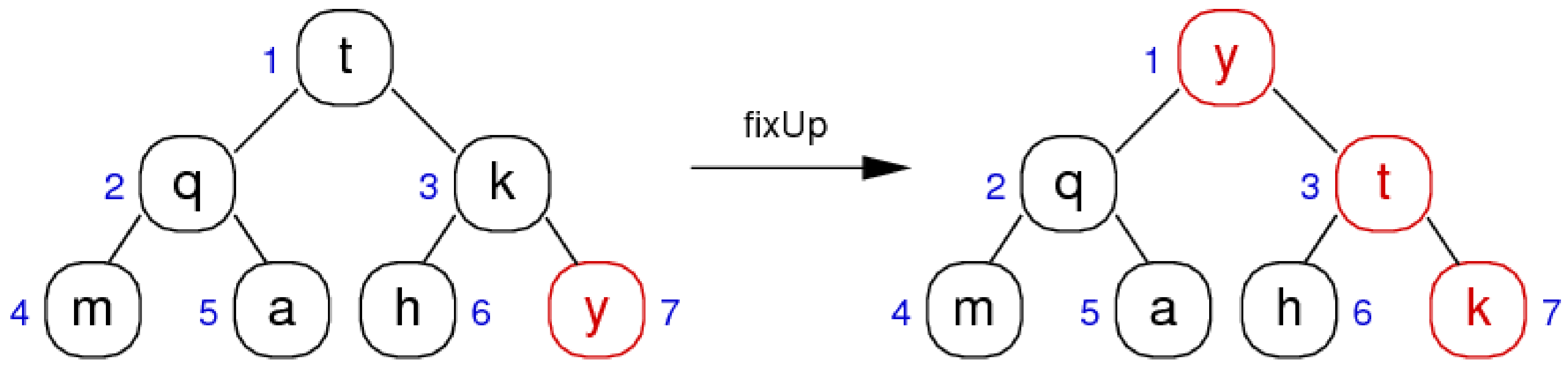
Insertion is a two-step process

1. add new element at bottom-most, rightmost position
2. reorganise values along path to root to restore heap property



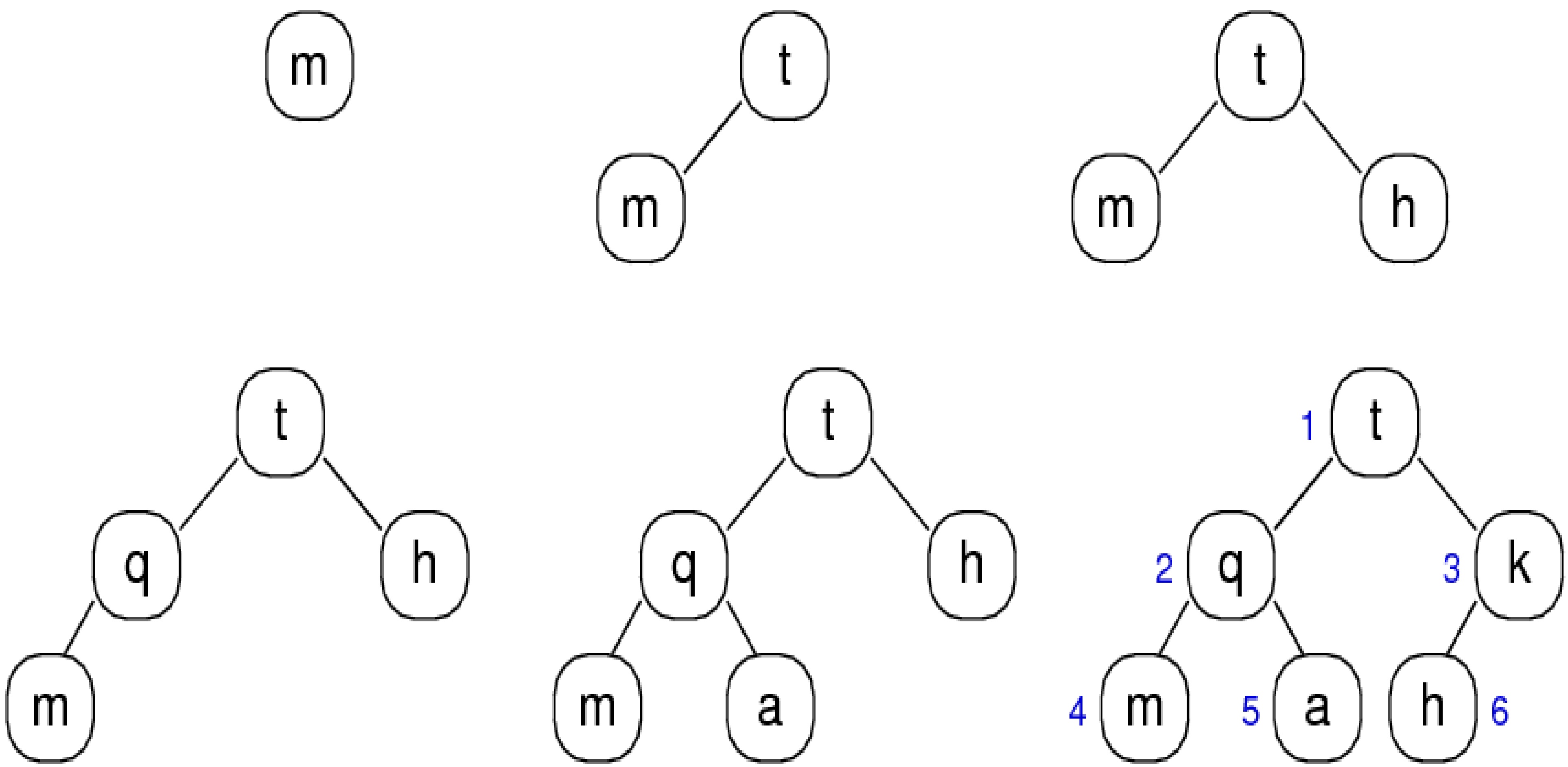
HEAP INSERTION FIX-UP CODE

```
// 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

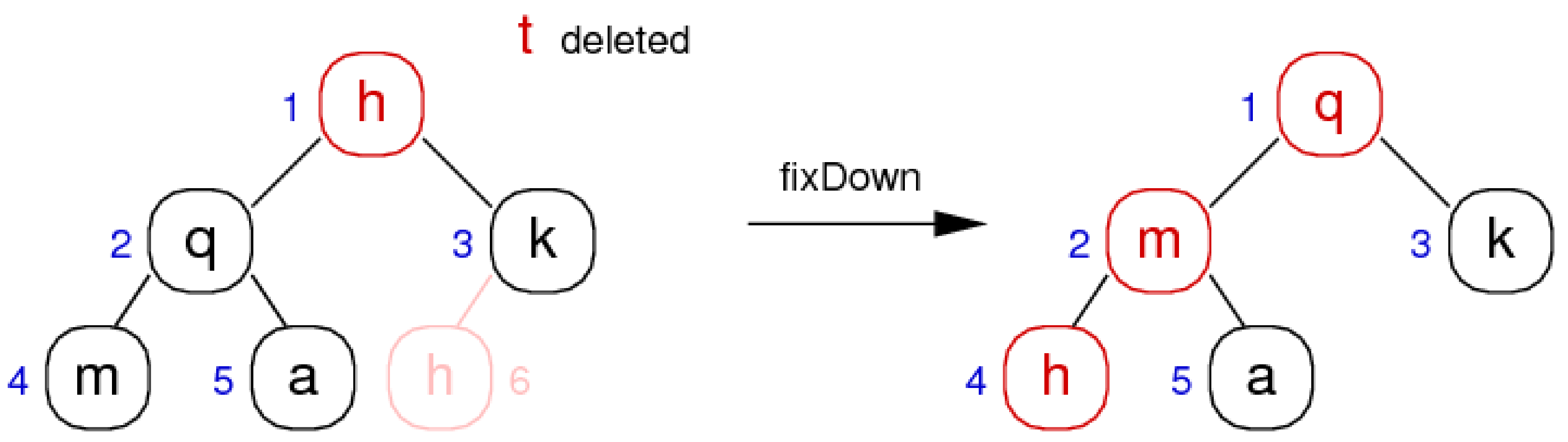
Items inserted in order m t h q a k



DELETION WITH HEAPS

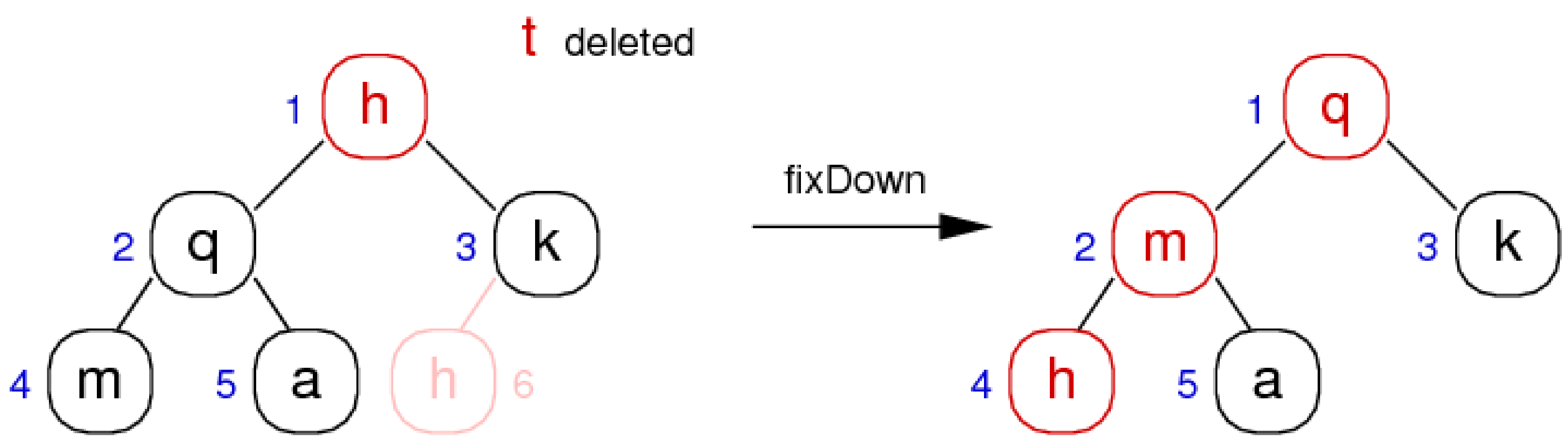
Deletion is a three-step process

1. replace root value by bottom-most, rightmost value
2. remove bottom-most, rightmost value
3. reorganise values along path from root to restore heap



HEAP DELETION FIX-DOWN CODE

```
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

H E A P S F U N

Show the heap after an item is deleted.

Show the heap after another item is deleted.