>>

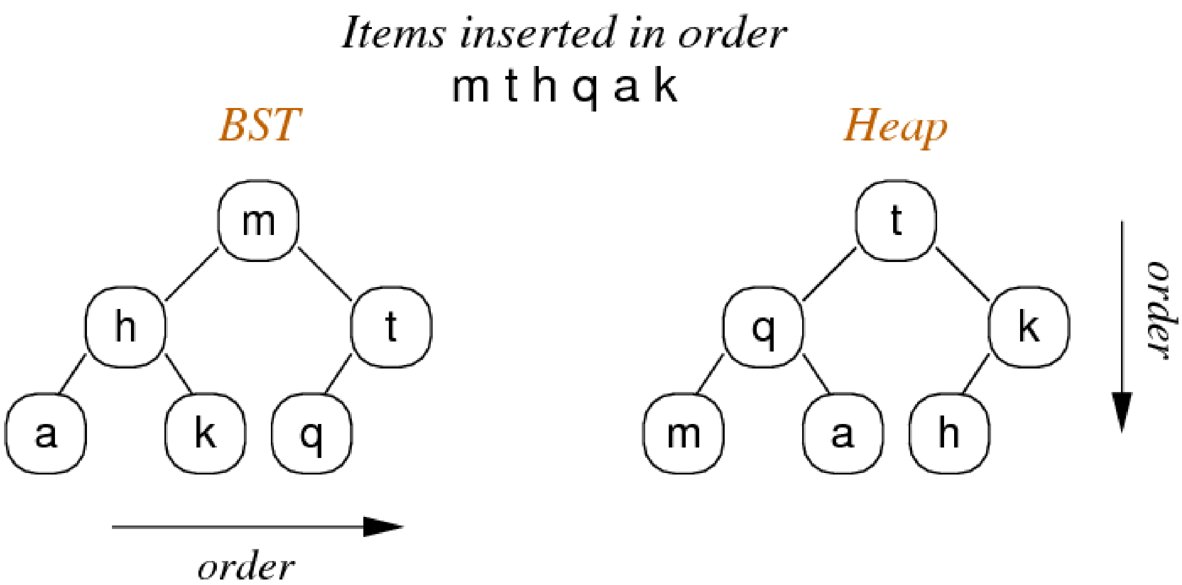
Heaps and Priority Queues

∧ >>

❖ Heaps

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

- cf. binary search trees which have left-to-right ordering



❖ ... Heaps

Heap characteristics ...

- 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)

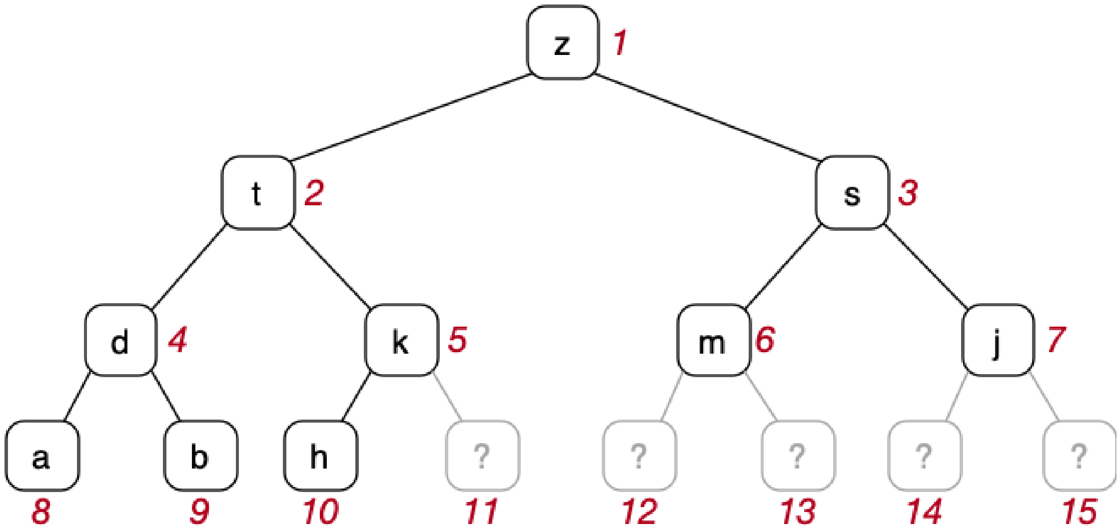
Since heaps are *dense* trees, depth = $floor(log_2N)+1$

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

Heaps are typically used for implementing Priority Queues

❖ ... Heaps

Heaps grow in regular (level-order) manner:

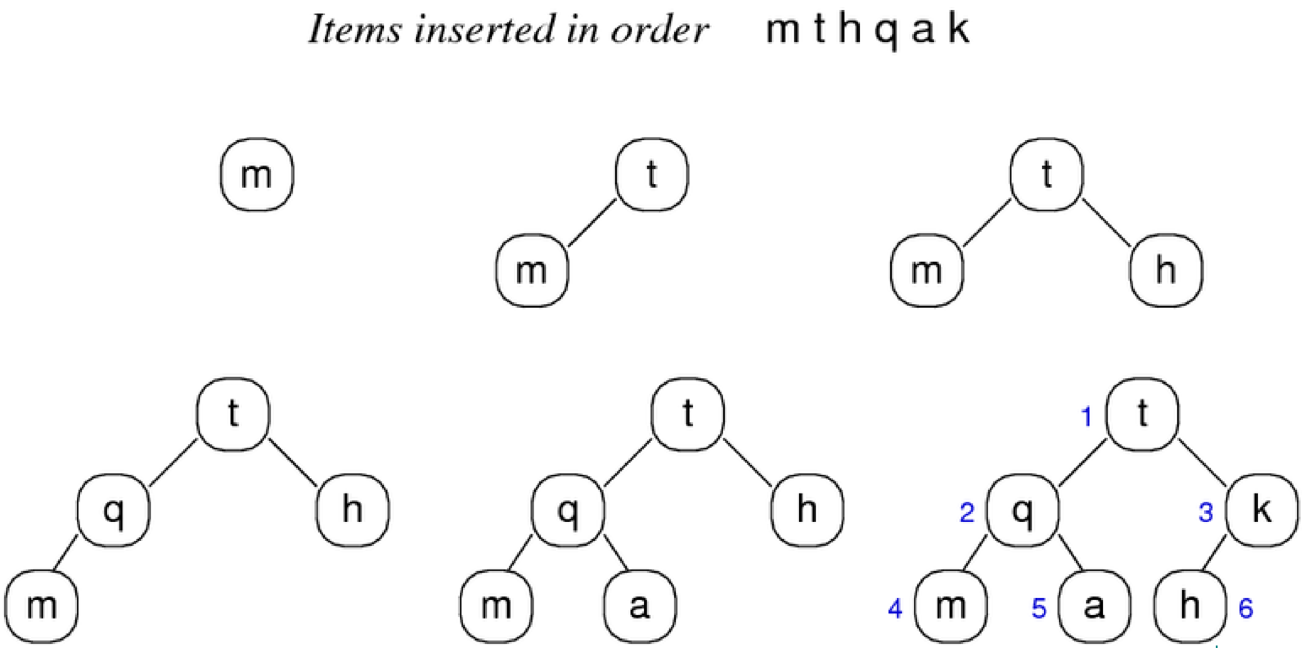


Nodes are always added in sequence indicated by numbers

<< ∧ >>

❖ ... Heaps

Trace of growing heap ...



<< ∧ >>

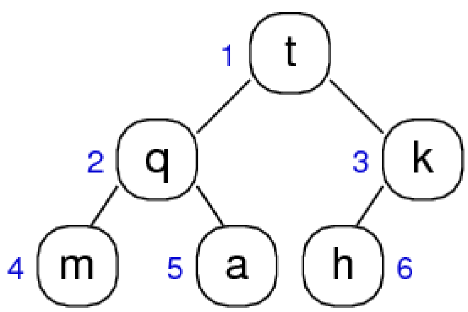
❖ ... Heaps

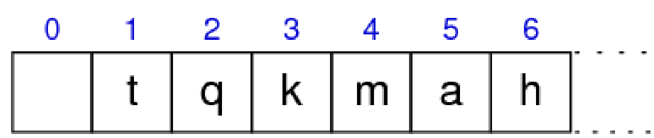
BSTs are typically implemented as linked data structures.

Heaps are often implemented via arrays (assumes we know max size)

Simple index calculations allow navigation through the tree:

- left child of **`Item`** at index *i* is located at *2i*
- right child of **`Item`** at index *i* is located at *2i+1*
- parent of **`Item`** at index *i* is located at *i/2*





<< ^ >>

❖ ... Heaps

Heap data structure:

```
typedef struct HeapRep {
   Item *items;  // array of Items
   int  nitems;  // #items in array
   int  nslots;  // #elements in array
} HeapRep;

typedef HeapRep *Heap;
```

Initialisation: **`nitems=0`**, **`nslots=`***ArraySize*

One difference: we use indexes from 1..**`nitems`**

Note: unlike "normal" C arrays, **`nitems`** also gives index of last item

❖ ... Heaps

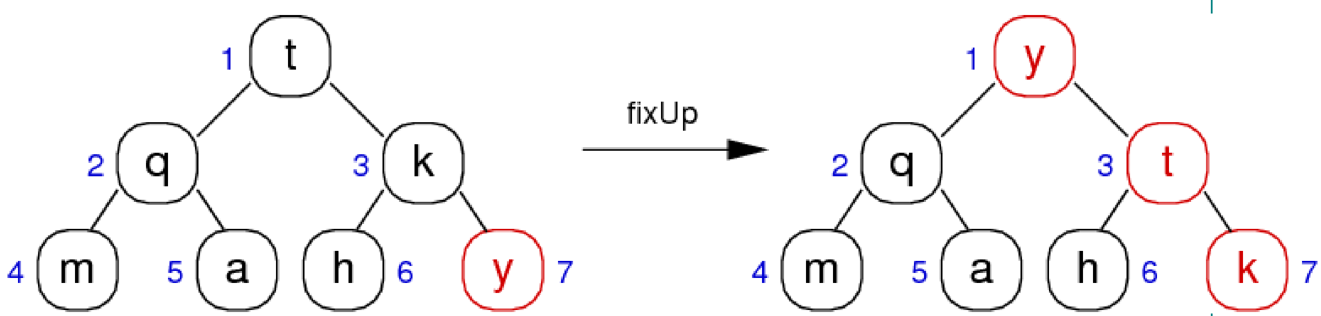
Creating new heap:

```
Heap newHeap(int N)
{
   Heap new = malloc(sizeof(HeapRep));
   Item *a = malloc((N+1)*sizeof(Item));
   assert(new != NULL && a != NULL);
   new->items = a;    // no initialisation needed
   new->nitems = 0;   // counter and index
   new->nslots = N;   // index range 1..N
   return new;
}
```

❖ Insertion with Heaps

Insertion is a two-step process

- add new element at next available position on bottom row
  (but this might violate heap property; new value larger than parent)
- reorganise values along path to root to restore heap property



❖ ... Insertion with Heaps

Insertion into heap:

```
void HeapInsert(Heap h, Item it)
{
   // is there space in the array?
   assert(h->nitems < h->nslots);
   h->nitems++;
   // add new item at end of array
   h->items[h->nitems] = it;
   // move new item to its correct place
   fixUp(h->items, h->nitems);
}
```

<< ^ >>

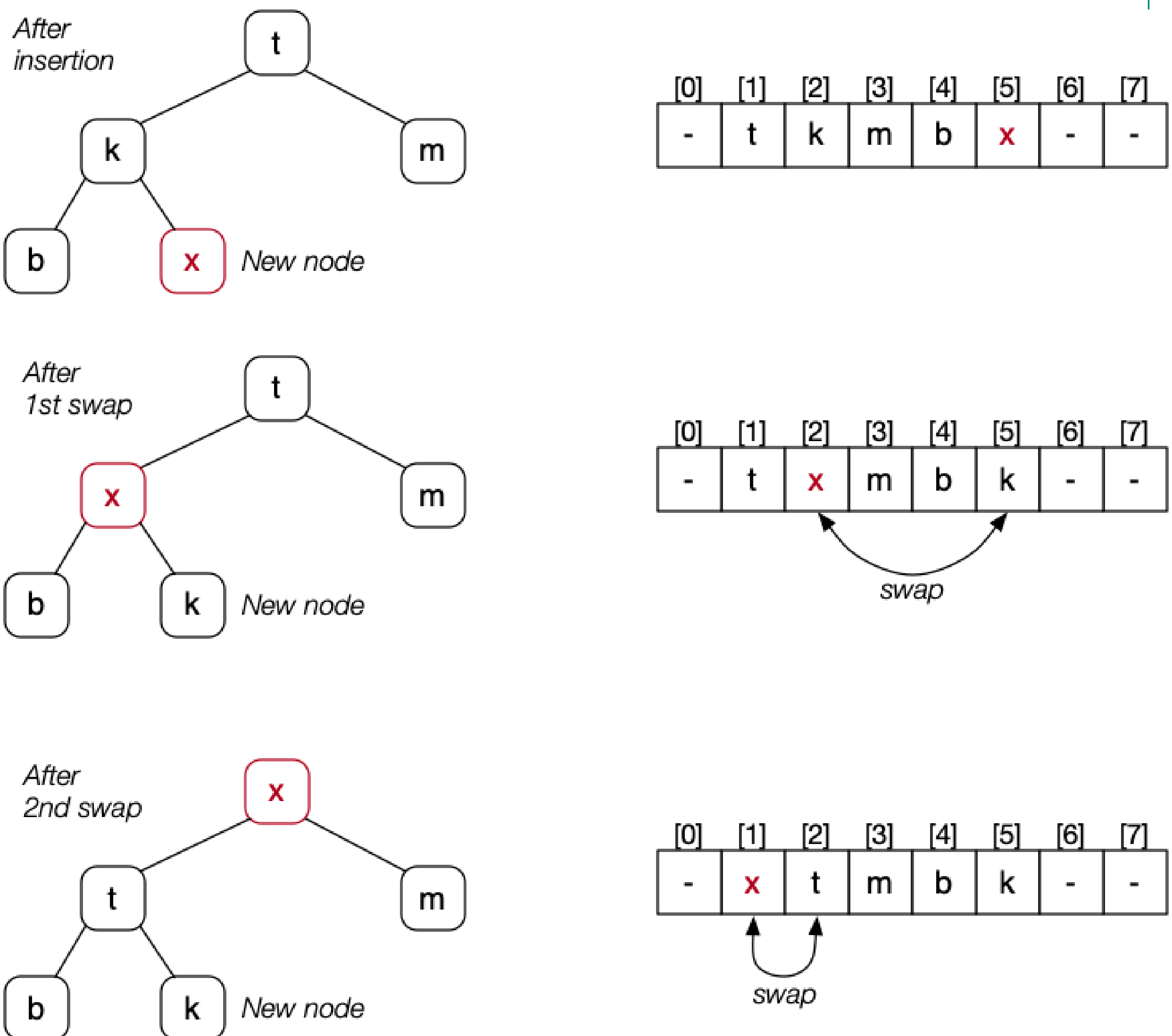
❖ ... Insertion with Heaps

Bottom-up heapify:

```
// force value at a[i] into correct position
void fixUp(Item a[], int i)
{
   while (i > 1 && less(a[i/2],a[i])) {
      swap(a, i, i/2);
      i = i/2;  // integer division
   }
}
void swap(Item a[], int i, int j)
{
   Item tmp = a[i];
   a[i] = a[j];
   a[j] = tmp;
}
```

❖ ... Insertion with Heaps

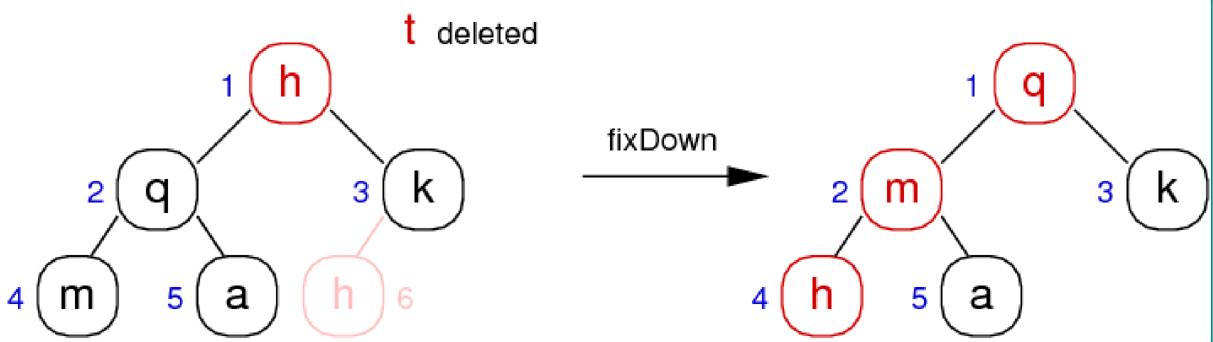
Trace of **`fixUp`** after insertion ..



❖ Deletion with Heaps

Deletion is a three-step process:

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



<< ∧ >>

❖ ... Deletion with Heaps

Deletion from heap (always remove root):

```
Item HeapDelete(Heap h)
{
   Item top = h->items[1];
   // overwrite first by last
   h->items[1] = h->items[h->nitems];
   h->nitems--;
   // move new root to correct position
   fixDown(h->items, 1, h->nitems);
   return top;
}
```

<< ∧ >>

❖ ... Deletion with Heaps

Top-down heapify:

```
// force value at a[i] into correct position
// note that N gives max index *and* # items
void fixDown(Item a[], int i, int N)
{
   while (2*i <= N) {
      // compute address of left child
      int j = 2*i;
      // choose larger of two children
      if (j < N && less(a[j], a[j+1])) j++;
      if (!less(a[i], a[j])) break;
      swap(a, i, j);
      // move one level down the heap
      i = j;
   }
}
```

<< ∧ >>

❖ Cost Analysis

Recall: tree is compact; max path length = $log_2 n$

For insertion ...

- add new item at end of array ⇒ $O(1)$
- move item up into correct position ⇒ $O(log_2 n)$

For deletion ...

- replace root by item at end of array ⇒ $O(1)$
- move new root down into correct position ⇒ $O(log_2 n)$

<< ^ >>

❖ Priority Queues

Heap behaviour is exactly behaviour required for Priority Queue ...

- **`join(PQ,it)`**: ensure highest priority item at front of queue
- **`it = leave(PQ)`**: take highest priority item from queue

So ...

```
typedef Heap PQueue;

void join(PQueue pq, Item it) { HeapInsert(pq,it); }

Item leave(PQueue pq) { return HeapDelete(pq); }
```

<< ∧

❖ ... Priority Queues

Heaps are not the only way to implement priority queues ...

Comparison of different Priority Queue representations:

	Array (sorted)	Array (unsorted)	List (sorted)	List (unsorted)	Heap
space usage	$O(N)$*	$O(N)$*	$O(N)$	$O(N)$	$O(N)$*
join	$O(N)$	$O(1)$	$O(N)$	$O(1)$	$O(\log N)$
leave	$O(N)$	$O(N)$	$O(1)$	$O(N)$	$O(\log N)$
is empty?	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$

for a Priority Queue containing N items

* If fixed-size array (no realloc), choose max N that might ever be needed

Produced: 5 Jul 2020