Sorting (ii)

- Summary of Sorting Methods
- Lower Bound for Comparison-Based Sorting
- Radix Sort

❖ Summary of Sorting Methods

Sorting = arrange a collection of *N* items in ascending order ...

Elementary sorting algorithms: $O(N^2)$ comparisons

- selection sort, insertion sort, bubble sort

Advanced sorting algorithms: $O(N \log N)$ comparisons

most effective

- quicksort, merge sort, heap sort (priority queue)

Most are intended for use in-memory (random access data structure).

Merge sort adapts well for use as disk-based sort.

❖ ... Summary of Sorting Methods

Other properties of sort algorithms: stable, adaptive

Selection sort:

- stability depends on implementation
- not adaptive

Bubble sort:

- is stable if items don't move past same-key items
- adaptive if it terminates when no swaps

Insertion sort:

- stability depends on implementation of insertion
- adaptive if it stops scan when position is found

❖ ... Summary of Sorting Methods

Other properties of sort algorithms: stable, adaptive

Quicksort:

- easy to make stable on lists; difficult on arrays
- can be adaptive depending on implementation

Merge sort:

- is stable if merge operation is stable
- can be made adaptive (but version in slides is not)

Heap sort:

- is not stable because of top-to-bottom nature of heap ordering
- adaptive variants of heap sort exist (faster if data almost sorted)

❖ Lower Bound for Comparison-Based Sorting

All of the above sorting algorithms for arrays of *n* elements

- have comparing whole keys as a critical operation

Such algorithms cannot work with less than *O(n log n)* comparisons

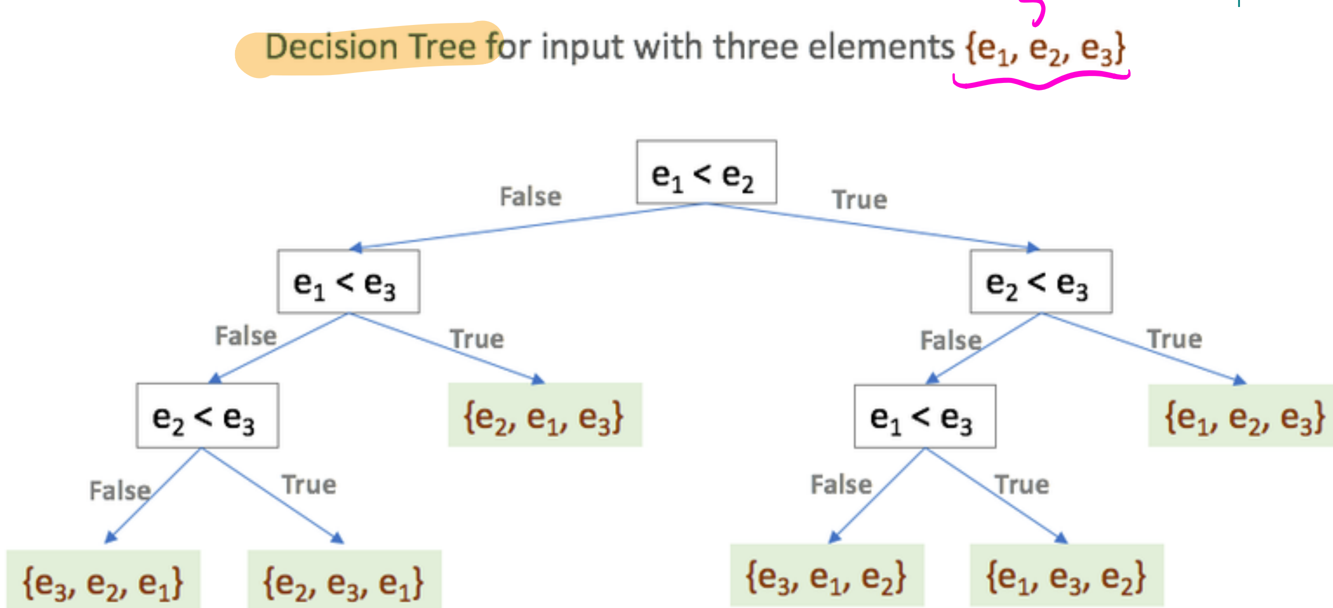
Informal proof (for arrays with no duplicates):

- there are *n!* possible permutation sequences
- one of these possible sequences is a sorted sequence
- each comparision reduces # possible sequences to be considered

(continued ...)

❖ ... Lower Bound for Comparison-Based Sorting

Can view sorting as navigating a decision tree ...



(continued ...)

❖ ... Lower Bound for Comparison-Based Sorting

Can view the sorting process as

- following a path from the root to a leaf in the decision tree
- requiring one comparison at each level

For *n* elements, there are *n!* leaves

- height of such a tree is at least $log_2(n!)$

  ⇒ number of comparisions required is at least $log_2(n!)$

So, for comparison-based sorting, lower bound is $\Omega(n\ log_2\ n)$.

Are there faster algorithms not based on whole key comparison?

❖ Radix Sort

Radix sort is a non-comparative sorting algorithm.

Requires us to consider a key as a tuple (k_1, k_2, ..., k_m), e.g.

- represent key 372 as (3, 7, 2)
- represent key "**`sydney`**" as (s, y, d, n, e, y)

Assume only small number of possible values for k_i, e.g.

- numeric: 0-9 ... alpha: a-z

If keys have different lengths, pad with suitable character, e.g.

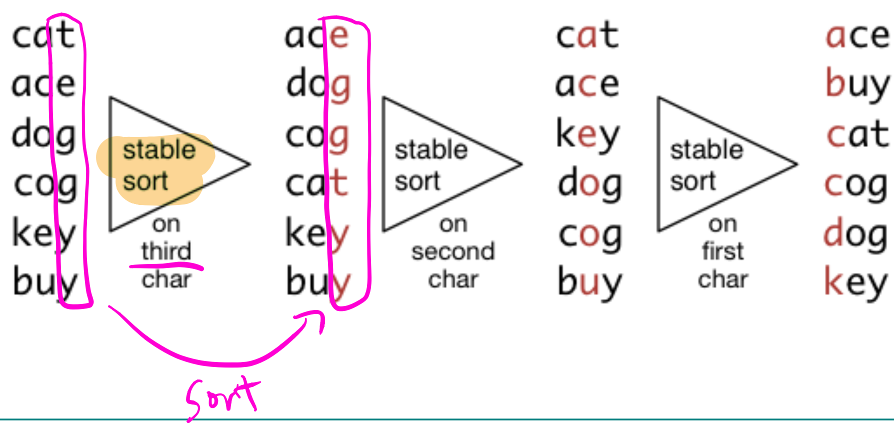
- numeric: 123, 002, 015 ... alpha: "**`abc`**", "**`zz␣`**", "**`t␣␣`**"

❖ ... Radix Sort

Radix sort algorithm:

- stable sort on k_m,
- then stable sort on $k_{(m-1)}$,
- continue until we reach k_1

Example:



❖ ... Radix Sort

Stable sorting (bucket sort):

```
// sort array A[n] of keys
// each key is m symbols from an "alphabet"
// array of buckets, one for each symbol
for each i in m .. 1 do
   empty all buckets
   for each key in A do
      append key to bucket[key[i]]
   end for
   clear A
   for each bucket in order do
      for each key in bucket do
         append to array
      end for
   end for
end for
```

❖ ... Radix Sort

Example:

- **m** = 3, alphabet = {'a', 'b', 'c'}, B[ ] = buckets
- A[ ] = {"abc", "cab", "baa", "a␣␣", "ca␣"}

After first pass (**i** = 3): end.

- B['a'] = {"baa"}, B['b'] = {"cab"}, B['c'] = {"abc"}, B['␣'] = {"a␣␣","ca␣"}
- A[ ] = {"baa", "cab", "abc", "a␣␣", "ca␣"}

After second pass (**i** = 2): middle

- B['a'] = {"baa","cab","ca␣"}, B['b'] = {"abc"}, B['c'] = {}, B["␣"] = {"a␣␣"}
- A[ ] = {"baa", "cab", "ca␣", "abc", "a␣␣"}

After third pass (**i** = 1):

- B['a'] = {"abc","a␣␣"}, B['b'] = {"baa"}, B['c'] = {"cab","ca␣"}, B["␣"] = {}
- A[ ] = {"abc", "a␣␣", baa", "cab", "ca␣"}

❖ ... Radix Sort

Complexity analysis:

- array contains *n* keys, each key contains *m* symbols
- stable sort (bucket sort) runs in time *O(n)*
- radix sort uses stable sort *m* times

So, time complexity for radix sort = *O(mn)*

Radix sort performs better than comparison-based sorting algorithms

- when keys are short (small *m*) and arrays are large (large *n*)

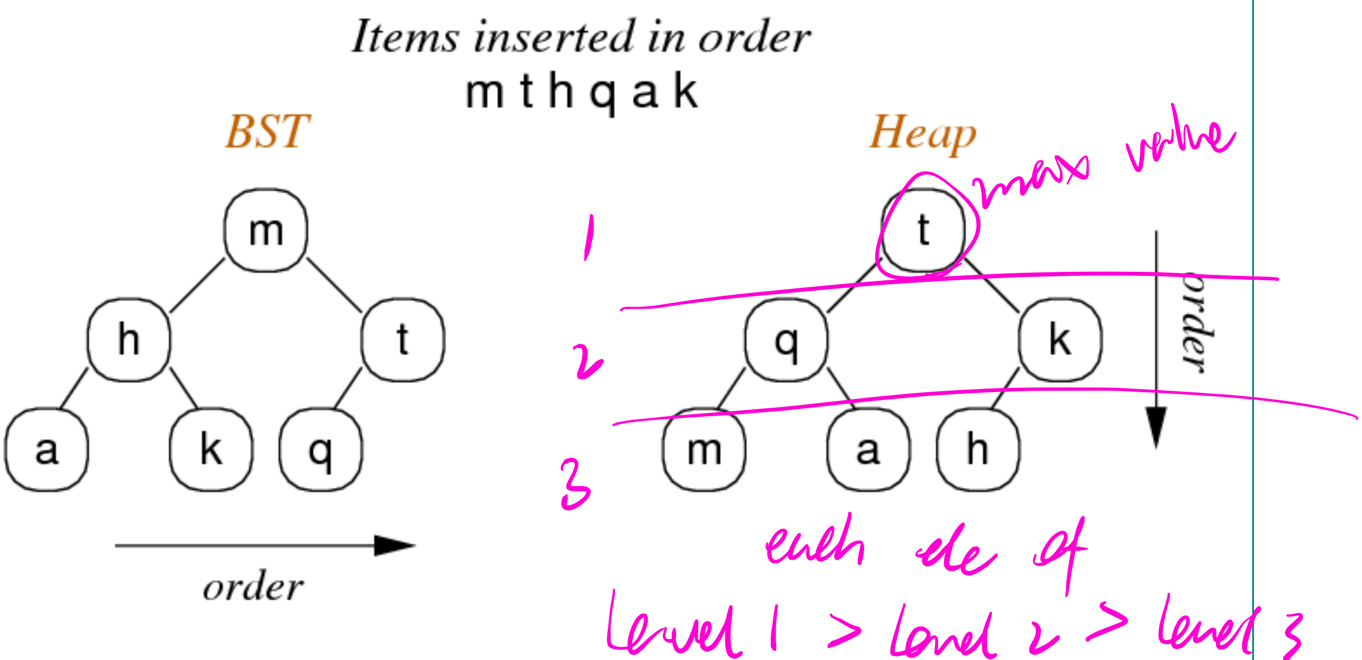
$m < \log_2 N$

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)

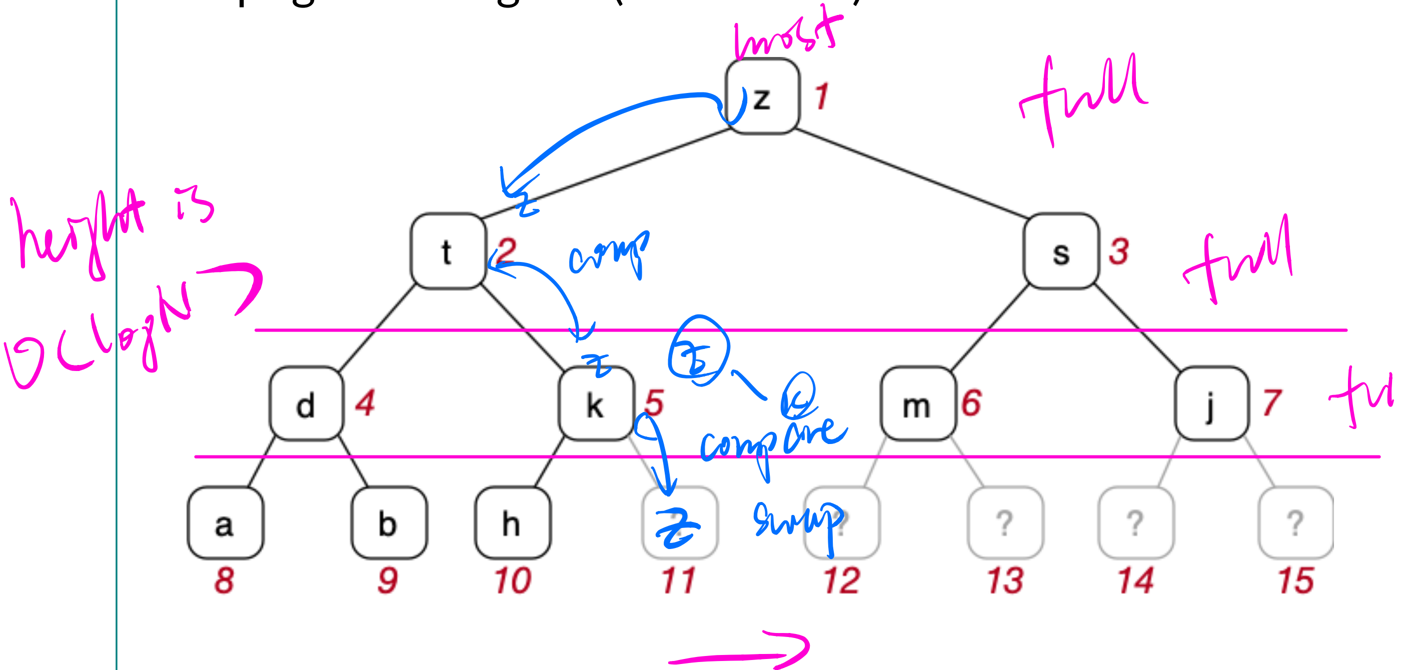
Since heaps are *dense* trees, depth = $floor(log_2N)+1$

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

Heaps are typically used for implementing Priority Queues

❖ ... Heaps

Heaps grow in regular (level-order) manner:

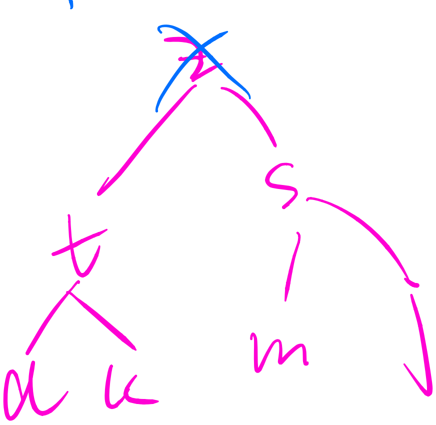


Nodes are always added in sequence indicated by numbers

if add z

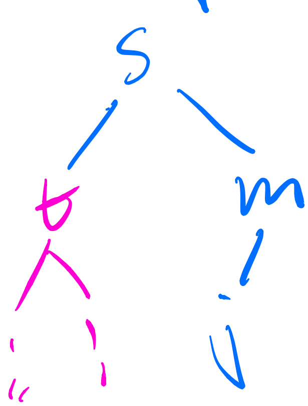
if delet z

m>j m↑



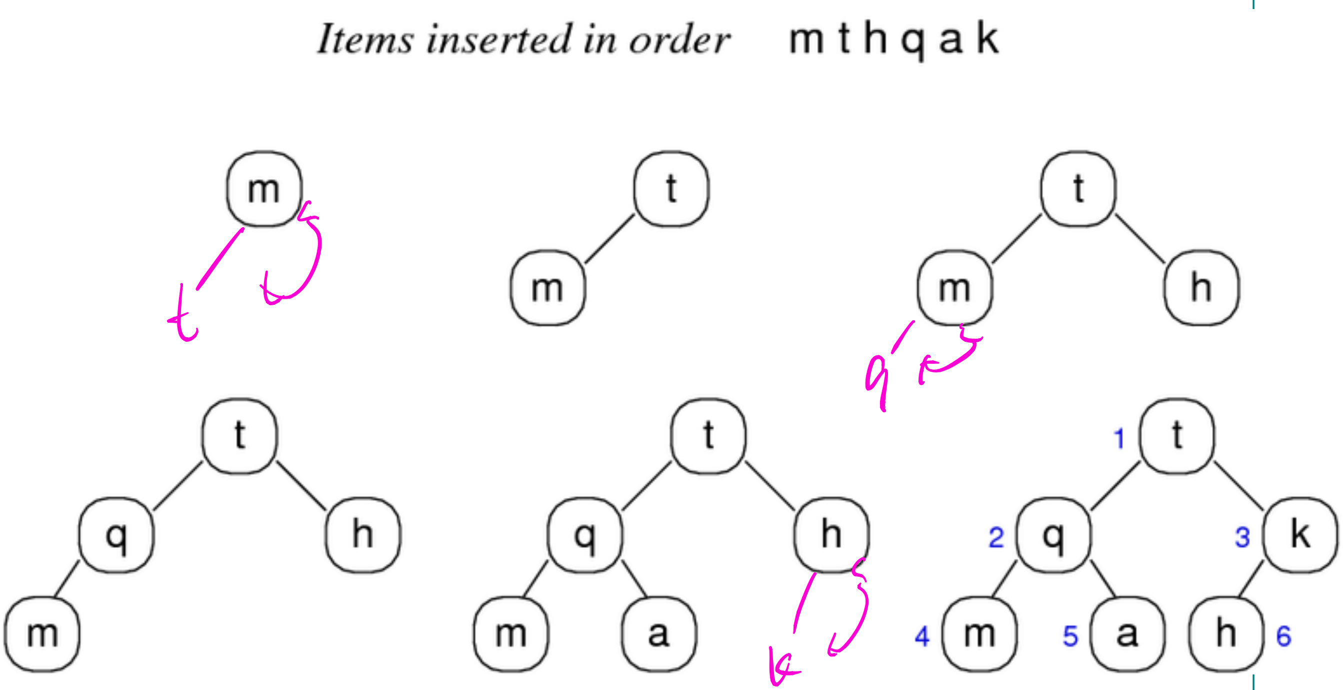
t<s

s↑



❖ ... Heaps

Trace of growing heap ...



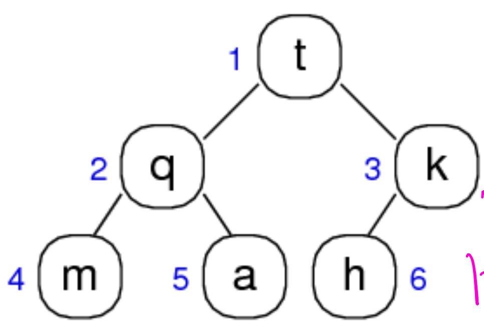
❖ ... Heaps

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* 2, 4, 6
- right child of **Item** at index *i* is located at *2i+1* 5, 3
- parent of **Item** at index *i* is located at *i/2*



no 0

0	1	2	3	4	5	6
	t	q	k	m	a	h

5 的 parent 5/2=2.5

i=3

2i=6 left

2i+1=7 right

取 2

2 → 5

❖ ... 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

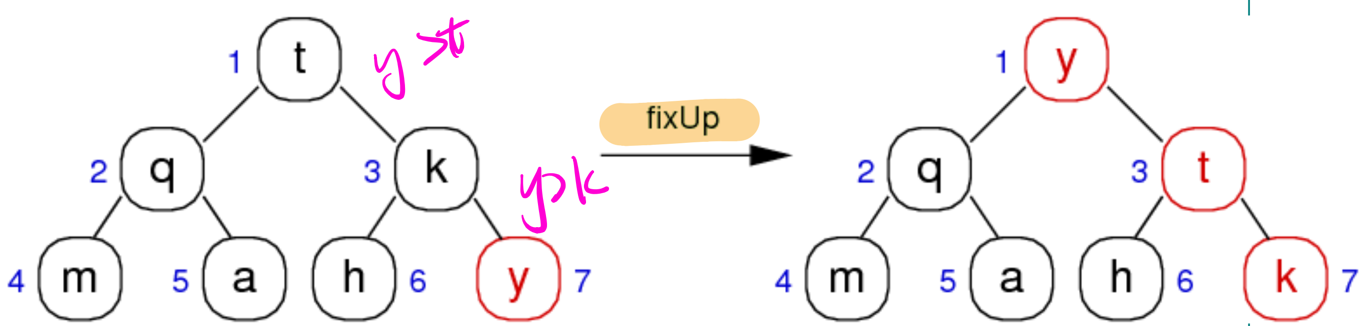
Creating new heap:

```
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);
}
```

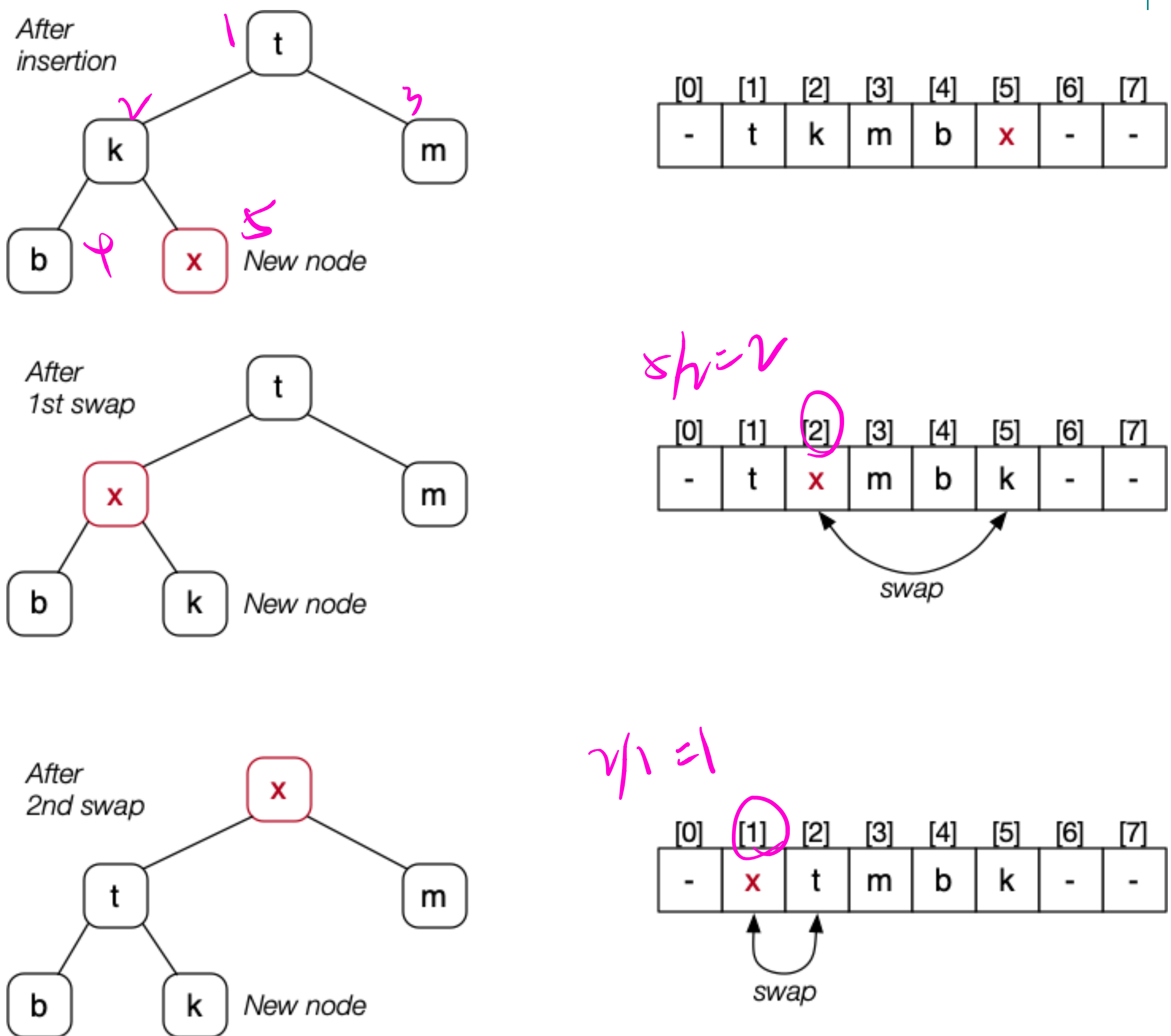
❖ ... Insertion with Heaps

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); — parent
      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

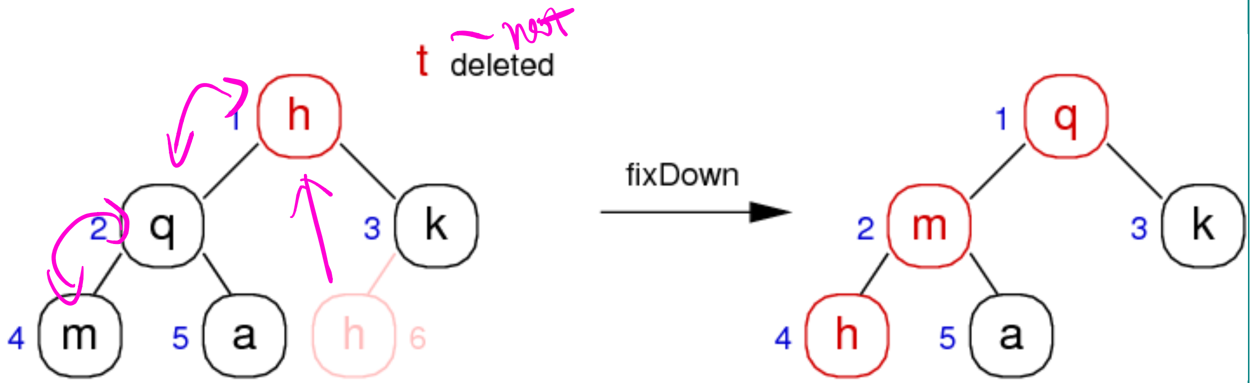
Trace of **`fixUp`** after insertion ..



❖ 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):

delet

last

```
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_2n

For insertion ...

- add new item at end of array ⇒ $O(1)$
- move item up into correct position ⇒ $O(log_2n)$

For deletion ...

- replace root by item at end of array ⇒ $O(1)$
- move new root down into correct position ⇒ $O(log_2n)$

❖ 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(logN)* best
leave	*O(N)*	*O(N)*	*O(1)*	*O(N)*	*O(logN)*
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