Heapsort

- HeapSort
- Cost Analysis

❖ HeapSort

Heapsort uses a priority queue (PQ) implemented as a heap.

Reminder: heap is a top-to-bottom ordered tree

- that has a simple implementation as an array of `Item`s

Reminder: priority queues ...

- implement a key-ordered queue structure
- items added to queue in arrival order
- items removed from queue in max-first order

❖ ... HeapSort

Heapsort (really PQ-sort) approach:

- insert all array items into priority queue
- one-by-one, remove all items from priority queue
- inserting each into successive array element

Priority queue operations ...

```
PQueue newPQueue();
void PQJoin(PQueue q, Item it);
Item PQLeave(PQueue q); // remove max Item
int  PQIsEmpty(PQueue q);
```

❖ ... HeapSort

Implementation of HeapSort:

```
void HeapSort(Item a[], int lo, int hi)
{
   PQueue pq = newPQueue();
   int i;
   for (i = lo; i <= hi; i++) {
      PQJoin(pq, a[i]);
   }
   for (i = hi; i >= lo; i--) {
      Item it = PQLeave(pq);
      a[i] = it;
   }
}
```

❖ ... HeapSort

Problem: requires an additional data structure (*O(N)* space)

Recall that earlier we defined **`fixDown()`**

- forces value at a[k] into correct position in heap

Allowed us to work with arrays as heap structures, hence as PQs.

Can we use these ideas to build an in-array PQ-sort?

❖ ... HeapSort

Reminder: **`fixDown()`** function

```
// force value at a[i] into correct position in a[1..N]
// note that N gives max index *and* number of 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;
   }
}
```

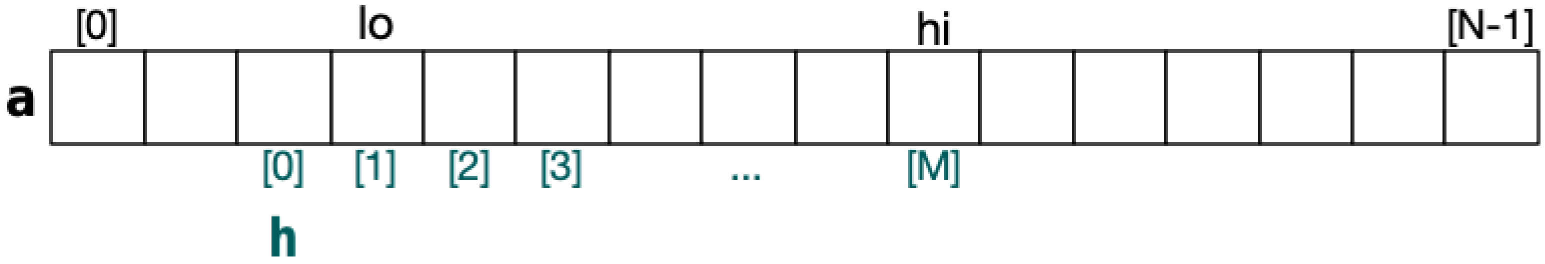
❖ ... HeapSort

Heapsort: multiple iterations over a shrinking heap

- initially use whole array as a heap
- uses **`fixDown`** to set max value at end
- reduce size of heap, and repeat

One minor complication: **`a[lo..hi]`** vs **`h[1..M]`** (where **`M=hi-lo+1`**)

To solve: pretend that heap starts one location earlier.



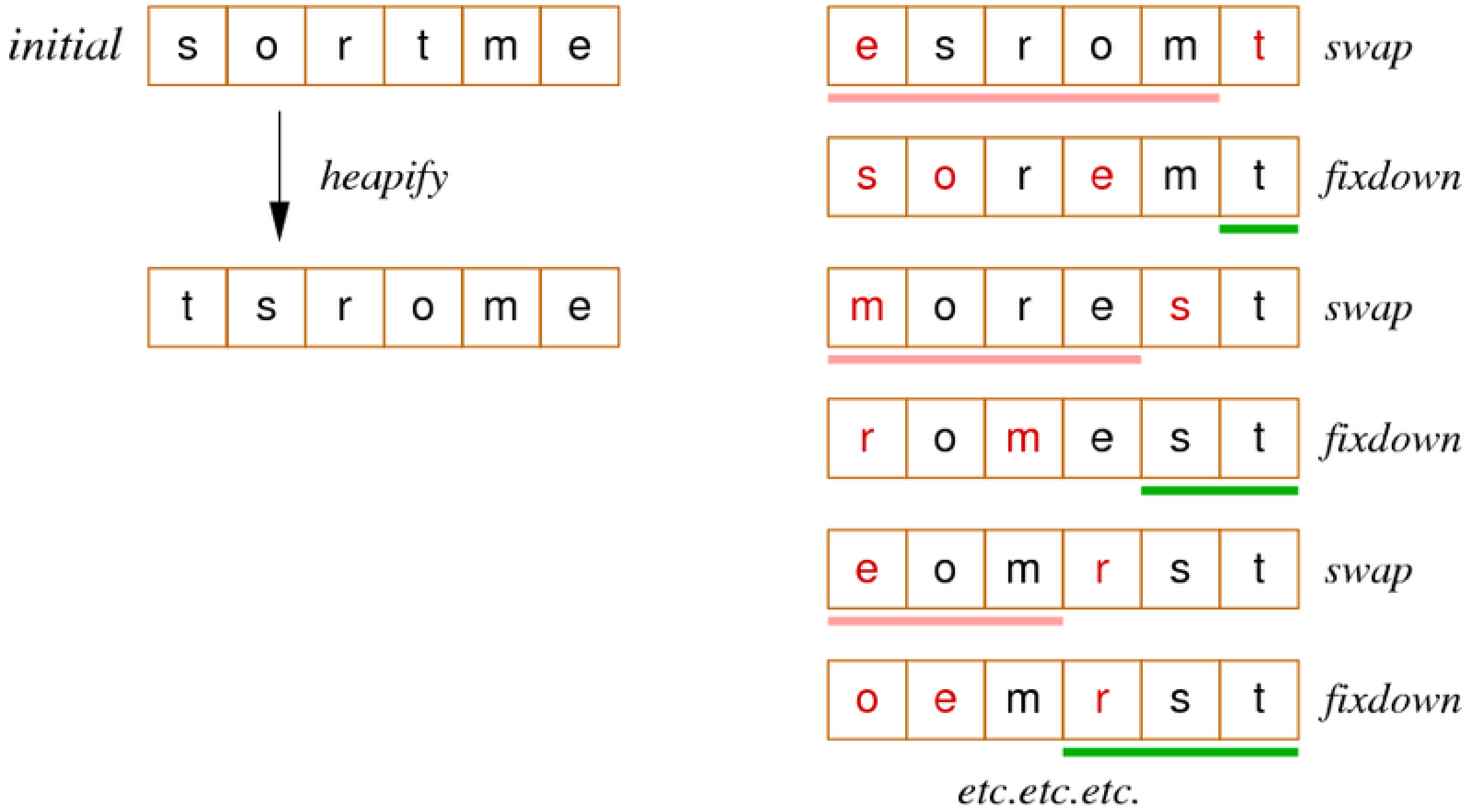
❖ ... HeapSort

Heapsort algorithm:

```
void heapsort(Item a[], int lo, int hi)
{
   int i, N = hi-lo+1;
   Item *h = a+lo-1;  //start addr of heap
   // construct heap in a[0..N-1]
   for (i = N/2; i > 0; i--)
      fixDown(h, i, N);
   // use heap to build sorted array
   while (N > 1) {
      // put largest value at end of array
      swap(h, 1, N);
      // heap size reduced by one
      N--;
      // restore heap property after swap
      fixDown(h, 1, N);
   }
}
```

❖ ... HeapSort

Trace of heapsort:



❖ Cost Analysis

Heapsort involves two stages

- build a heap in the array
  - iterates N/2 times, each time doing **`fixDown()`**
  - each **`fixDown()`** is *O(logN)*, so overall *O(NlogN)*
  - note: can write *heapify* more efficiently than we did *O(N)*
  - note: each **`fixDown()`** involves at most $log_2(2C + S)$
- use heap to build sorted array
  - iterates N times, each time doing **`swap()`** and **`fixDown()`**
  - **`swap()`** is *O(1)*, **`fixDown()`** is *O(logN)*, so overall *O(NlogN)*

Cost of heapsort = *O(NlogN)*