1. Purpose of a heap
   1. Priority queue: queue, but higher-priority cases are processed earlier.
   2. Heap sort: put everything in priority queue, delete everything.
      1. Same order time as MergeSort and QuickSort
2. Completing levels
   1. Every level is inserted left to right
3. Heap property: children nodes are higher than parent nodes.
   1. To keep tree together, take item that was in last row to root.
   2. Percolate down: reorganize into heap order.
4. Delete [O(*n*)]
   1. Percolate down
5. Array application
   1. Make heap large enough to avoid having to realloc()
   2. Put root in index 1
   3. Fill rest of array in left-right order, level to level
   4. Break ties
      1. Alphabetical order
      2. No extra ties
   5. Code
      1. typedef struct person {
         1. char name [MAXSIZE+1];
         2. int priority;
      2. } person;
      3. typedef struct heap {
         1. person\*\* list;
         2. int n; //Number of items in heap
      4. } heap;
   6. Functions
      1. void init(heap\* hPtr) {
         1. hPtr->list = malloc(sizeof(person\*)\*HEAPSIZE);
         2. for (i = 0; i<HEAPSIZE+1; i++)
            1. hPtr->list[i] = NULL;
         3. hPtr->n = 0;
      2. }
      3. int compare(person\* a, person\*b) {
         1. if(a->priority != b->priority)
            1. return a->priority – b->priority;
         2. else
            1. return strcmp(a->name, b->name);
      4. }
      5. void percolateUp(heap\* hPtr, int index) {
         1. //At root, can’t go up
         2. if (index == 1) return;
         3. int pIndex = index/2;
         4. //Node and parent are out of order; swap and recourse
         5. if(compare(hPtr->list[index], hPtr->list[pIndex] < 0) {
            1. person\* tmp = hPtr->list[index];
            2. hPtr->list[index] = hPtr->list[pIndex];
            3. hPtr->list[pIndex] = tmp;
            4. percolateUp(hPtr, pIndex);
         6. }
      6. }
      7. void percolateDown(heap\* hPtr, int index) {
         1. //Lead node
         2. if (2\*index > hPtr->n) return;
         4. //Left child exists; no right child
         5. if(2\*index == hPtr->n) {
            1. if(compare(hPtr->list[2\*index], hPtr->list[pIndex] < 0) {

person\* tmp = hPtr->list[2\*index];

hPtr->list[2\*index] = hPtr->list[pIndex];

hPtr->list[pIndex] = tmp;

* + - * 1. }
        2. return;
      1. }
      2. //Figure out whether or not the left or right child is better
      3. int swapIndex = compare(hPtr->list[2\*index],hPtr->list[2\*index+1]) < 0) ? 2\*index: 2\*index+1 //see code
      4. //Node and parent are out of order; swap and recourse
      5. if(compare(hPtr->list[swapIndex,hPtr->list[index]] < 0) {
         1. person\*tmp = hPtr->list[index];
         2. hPtr->list[index] = hPtr->list[swapIndex];
         3. hPtr->list[swapIndex] = tmp;
         4. percolateDown(hPtr, swapIndex);
      6. }
    1. }
    2. void insert(heap\* hPtr, person\* item) {
       1. //Increment size of heap
       2. hPtr->n++;
       3. //Copy new item into first open slot in the heap
       4. hPtr->list[hPtr->n] = item;
       5. //Percolate up
       6. percolateUp (hPtr, hPtr->n);
    3. }
    4. //Heap must be non-empty; otherwise crash
    5. void deleteMin(heap\* hPtr) {
       1. //Item to delete
       2. person\* retval = hPtr->list[1];
       3. //Copy last item into first slot
       4. hPtr->list[1] = hPtr->list[hPtr->n];
       5. //Percolate down
       6. percolateDown(hPtr, 1);
    6. }