Priority queue ADT

- Like a dictionary, a priority queue stores a set of pairs (key, info)
- The rank of an object depends on its priority (key)

- Allows only access to
  - Object findMin() //returns info of smallest key
  - Object removeMin() // removes smallest key
  - void insert(key k, info i) // inserts pair

- Applications: customers in line, Data compression, Graph searching, Artificial intelligence...
Heap - Definition

- A **heap** is a binary tree such that:
  - For any node $n$ other than the root,
    $\text{key}(n) \geq \text{key}(\text{parent}(n))$
  - Let $h$ be the height of the heap
    - First $h-1$ levels are full:
      For $i = 0,...,h-1$, there are $2^i$ nodes of depth $i$
    - At depth $h$, the leaves are packed on the left side of the tree
Height

- Height of a node in a tree: the number of edges on the longest simple path down from the node to a leaf.
- Height of a heap = height of the root = $\Theta(lg n)$.
- Most Basic operations on a heap run in $O(lg n)$ time
- Shape of a heap
Array representation of heaps

• A heap with n keys can be stored in an array of length n+1:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>10</td>
<td>8</td>
<td>9</td>
<td>8</td>
<td>9</td>
<td>12</td>
</tr>
</tbody>
</table>

• For a node at index i,
  – The parent (if any) is at index ⌊i/2⌋
  – The left child is at index 2*i
  – The right child is at index 2*i + 1

• lastNode is the first empty cell of the array. To update it, either add or subtract one
Heaps as arrays

Max-heap as an array.

Map from array elements to tree nodes and vice versa

- Root – $A[1]$
- Left[$i$] – $A[2i]$
- Right[$i$] – $A[2i+1]$
- Parent[$i$] – $A[\left \lfloor i/2 \right \rfloor]$
Sorting with Heaps

• Use max-heaps for sorting.
• The array representation of max-heap is not sorted.

Steps in sorting
  – Convert the given array of size $n$ to a max-heap
  – Swap the first and last elements of the array.
    • Now, the largest element is in the last position – where it belongs.
    • That leaves $n – 1$ elements to be placed in their appropriate locations.
    • However, the array of first $n – 1$ elements is no longer a max-heap.
    • Float the element at the root down one of its subtrees so that the array remains a max-heap (MaxHeapify)
  • Repeat step 2 until the array is sorted.
Heapsort (overview)

• Combines the better attributes of merge sort and insertion sort.
  – Like merge sort, worst-case running time is $O(n \lg n)$.
  – Like insertion sort, sorts in place.

• Introduces an algorithm design technique
  – Create data structure (heap) to manage information during the execution of an algorithm.
Heapsort (Outline)

1. Builds a min-heap from the array.
2. Extract the minimum element (i.e. the root) and place it in the sorted array.
3. Remove this minimum node (i.e. the root) from the heap, and call MIN-HEAPIFY on the new root.
4. Repeat this process (goto 2) until only one node remains.
HeapSort (Implementation)

**Algorithm** heapSort(array A[0...n-1])

Heap h ← new Heap()

for i=0 to n-1 do
    h.insert(A[i])

for i=0 to n-1 do
    A[i] ← h.removeMin()

Running time: O(n log n) in worst-case

Easy to do in-place: Just use the array A to store the heap

Note: We can optimize the procedure to construct the initial heap (More in COMP251)
Heapsort – Example

|   1   |   2   |   4   |   3   |   7   |

Diagram:

1
  /   \
2     4
/     /
3     7
Heapsort – Example

Heapify

1. Initial array: 7 2 4 3
2. First heapify: 2 3 4 7
3. Final heap: 2 3 4 7
Heapsort – Example

Heapsort: 7 3 4 3 7 4

Heapify: 7 3 4 3 7 4

Heapify: 3 7 4 3 7 4
Heapsort – Example

4 7

Heapify

4

7

4

7

1 2 3
Heapsort – Example

7 1 2 3

7 1 2 3
Heap Procedures for Sorting

- BuildMinHeap $O(n)$
- for loop $n-1$ times (i.e. $O(n)$)
  - exchange elements $O(1)$
  - MinHeapify $O(lg \ n)$

$\Rightarrow$ HeapSort $O(n \ lg \ n)$
Sorting in place with max-heap

1. Builds a max-heap from the array.
2. Put the maximum element (i.e. the root) at the correct place in the array by swapping it with the element in the last position in the array.
3. “Discard” this last node (knowing that it is in its correct place) by decreasing the heap size, and call MAX-HEAPIFY on the new root.
4. Repeat this process (goto 2) until only one node remains.
HeapSort (Implementation)

**Algorithm** heapSort(array A[0...n-1])
Heap h ← new Heap()
for i=0 to n-1 do
    h.insert(A[i])
for i=0 to n-1 do
    A[i] ← h.removeMin()

Running time: O(n log n) in worst-case
Easy to do in-place: Just use the array A to store the heap
Note: We can optimize the procedure to construct the initial heap (More in COMP251)
Heapsort (with max-heap)

\[\text{HeapSort}(A)\]
1. Build-Max-Heap\((A)\)
2. \text{for } i \leftarrow \text{length}[A] \text{ downto } 2
3. \text{do exchange } A[1] \leftrightarrow A[i]
4. \text{MaxHeapify}(A, 1, i-1)
Heapsort – Example

| 7 | 4 | 3 | 1 | 2 |

```
    7
   / \
  4   3
 / \   \
1   2   
```
Heapsort – Example

Heapify

2 4 3 1 7

4 2 3 1 7

Heapify

2

4

3

1

7

7

1

3

2
Heapsort – Example

Heapsort

Heapify

1 2 3 4 7
3 2 1 4 7

Heapify
Heapsort – Example

1 2 3 4 7

2 1 3 4 7

Heapify

1 2

2 1

7 4 3
Heapsort – Example

1 2 3 4 7

1 2 3 4 7

1

7 4 3 2
A glimpse to the next lecture:
Come back to dictionary ADT

• Reminder: A dictionary stores pairs (key, information)

• Operations:
  – find(key k)
  – insert(key k, info i)
  – remove(key k)

• Binary Search Trees implement all these operations in time $O(h)$, where $h$ is the height of the tree, which is $O(\log n)$ if we maintain the tree balanced.

• We can sometimes do better...
Hash tables

Suppose keys are integers between 0 and K-1.
Hash tables

• Suppose keys are integers between 0 and K-1
• Then, use an array A[0...K-1] containing elements of type "info" to store the dictionary:
  – insert(key k, info i): A[k] = i;
  – remove(key k): A[k] = null;
  – find(key k): return A[k];
• Running time: All operations are O(1)
• It's a miracle! Except that...