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)

Front of queue
Rear of queue

key: 9 8 6 5 4 2

• 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...
Outline

• Priority queues
• Heaps
• Operations on heaps
• Array-based implementation of heaps
• HeapSort
Priority queue ADT (as sorted array)

1. \((4, O_4)\) \((5, O_5)\) \((8, O_8)\)
2. \((4, O_4)\) \((5, O_5)\) \((8, O_8)\) \((9, O_9)\)
3. \((5, O_5)\) \((8, O_8)\) \((9, O_9)\)
4. \((5, O_5)\) \((6, O_6)\) \((8, O_8)\) \((9, O_9)\)
5. \((2, O_2)\) \((5, O_5)\) \((6, O_6)\) \((8, O_8)\) \((9, O_9)\)

- insert(9, O_9)
- remove()
- insert(6, O_6)
- insert(2, O_2)
Array implementation

**Unsorted array** of pairs (key, info)

- findMin(): Need to scan array \( O(n) \)
- insert(key, info): Put new object at the end \( O(1) \)
- removeMin(): First, findMin, then shift array \( O(n) \)

**Sorted array** of pairs (key, info)

- findMin(): Return first element \( O(1) \)
- insert(key, info):
  - Use binary-search to find position of insertion. \( O(\log n) \)
  - Then shift array to make space. \( O(n) \)
- removeMin(): findMin, then remove head \( O(1) \)
  (Note: using rotating arrays)
Doubly-linked list implementation

Using a sorted doubly-linked list of pairs (key, info)

**findMin()**: Return first element \( O(1) \)

**insert(key, info)**:

First, find location of insertion.

Binary Search?

No. Too slow on linked list.

Instead, we scan an array \( O(n) \)

Then insertion is easy \( O(1) \)

**removeMin()**: Remove first element of list \( O(1) \)
Heap data structure

• A heap is a data structure that implements a priority queue:
  – `findMin()`: $O(1)$
  – `removeMin()`: $O(\log n)$
  – `insert(key, info)`: $O(\log n)$

• A heap is based on a binary tree, but with a different property than a binary search tree

• heap ≠ binary `search` tree
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 $i = 0, \ldots, h-1$, there are $2^i$ nodes of depth $i$
  - At depth $h$, the leaves are packed on the left side of the tree
Height of a heap

What is the maximum number of nodes that fits in a heap of height $h$?

\[
\sum_{k=0}^{h} 2^k = 2^{h+1} - 1
\]

What is the minimum number?

\[
(2^h - 1) + 1 = 2^h
\]

Thus, the height of a heap with $n$ nodes is:

\[
\left\lfloor \log(n) \right\rfloor
\]
Heaps: findMin()

The minimum key is always at the root of the heap!
Heaps: Insert

Insert(key k, info i). Two steps:

1. Find the left-most unoccupied node and insert (k,i) there temporarily.
2. Restore the heap-order property (see next)
Heaps: Bubbling-up

Restoring the heap-order property:
- Keep swapping new node with its parent as long as its key is smaller than its parent’s key

Running time? \( O(h) = O(\log(n)) \)
Insert pseudocode

**Algorithm** insert(key k, info i)
**Input:** Key k and info i to add to the heap
**Output:** (k,i) is added

lastNode ← nextAvailableNode(lastNode)
lastNode.key ← k,
lastNode.info ← i
n ← lastNode
while (n.getParent()! =null and
    n.getParent().key > k) do
    swap (n.getParent(), n)
Heaps: RemoveMin()

- The minimum key is always at the root of the heap!
- Replace the root with last node

- Restore heap-order property (see next)
Heaps: Bubbling-down

Restoring the heap-order property:
- Keep swapping the node with its smallest child as long as the node’s key is larger than its child’s key

Running time? \( O(h) = O(\log(n)) \)
removeMin pseudocode

**Algorithm** removeMin()

**Input:** The heap

**Output:** A new heap where the node at the top of the input heap has been removed.

swap(lastNode, root)
Update lastNode
n ← root

while (n.key > min(n.getLeftChild().key, n.getRightChild().key)) do
  if (n.getLeftChild().key < n.getRightChild().key) then
    swap(n, n.getLeftChild)
  else swap(n, n.getRightChild)
Array representation of heaps

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

<table>
<thead>
<tr>
<th></th>
<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>-</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 $\lceil i/2 \rceil$
  – The left child is at index $2 \times i$
  – The right child is at index $2 \times i + 1$

• `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[\lfloor i/2 \rfloor]
HeapSort

**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)
Supplement

Implementing nextAvailableNode
NextAvailableNode - Example
Finding nextAvailableNode

nextAvailableNode(lastNode) finds the location where the next node should be inserted. It runs in time $O(n)$.

\[
\begin{align*}
n &= \text{lastNode}; \\
&\text{while } (n == (n.\text{parent}).\text{rightChild} \&\& n.\text{parent} != \text{null}) \text{ do} \\
&\quad n = n.\text{parent} \\
&\quad \text{if } (n.\text{parent} == \text{null}) \text{ then} \\
&\quad\quad \text{return left child of the leftmost node of tree} \\
&\quad \text{else} \\
&\quad\quad n = n.\text{parent} \quad /\quad \text{go up one more level} \\
&\quad\quad \text{if } (n \text{ has no right child}) \text{ then} \\
&\quad\quad\quad \text{return (right child of n)} \\
&\quad\quad \text{else} \\
&\quad\quad\quad n = n.\text{rightChild} \quad /\quad \text{go to right child} \\
&\quad\quad\quad \text{while } (n \text{ has a left child}) \text{ do} \\
&\quad\quad\quad\quad n = n.\text{leftChild} \\
&\quad\quad\quad \text{return (left child of n)}
\end{align*}
\]