Priority queue ADT

Heaps

Lecture 21
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)

Rear of queue

Front 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

(4, O_4) (5, O_5) (8, O_8)

(4, O_4) (5, O_5) (8, O_8) (9, O_9)

(5, O_5) (8, O_8) (9, O_9)

(5, O_5) (6, O_6) (8, O_8) (9, O_9)

(2, O_2) (5, O_5) (6, O_6) (8, O_8) (9, O_9)
Implementation of priority queue

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. \( O(n) \)
Implementation of priority queue

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 $\neq$ binary search tree
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 $i = 0,\ldots,h-1$, 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.

\[\text{swap}(\text{lastNode}, \text{root})\]
Update lastNode
\[n \leftarrow \text{root}\]
\[\text{while } (n.\text{key} > \min(n.\text{getLeftChild()}.\text{key}, n.\text{getRightChild()}.\text{key})) \text{ do}\]
\[\text{if } (n.\text{getLeftChild()}.\text{key} < n.\text{getRightChild()}.\text{key}) \text{ then}\]
\[\text{swap}(n, n.\text{getLeftChild})\]
\[\text{else } \text{swap}(n, n.\text{getRightChild})\]
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[$i$] – $A[\lfloor i/2 \rfloor]$
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 construction of the heap (See COMP251)
Supplement

Implementing nextAvailableNode
Finding nextAvailableNode

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

n = lastNode;
while (n == (n.parent).rightChild && n.parent!=null) do
    n = n.parent
if (n.parent == null) then
    return left child of the leftmost node of tree
else
    n = n.parent // go up one more level
    if (n has no right child) then
        return (right child of n)
    else
        n = n.rightChild // go to right child
while (n has a left child) do
    n = n.leftChild
return (left child of n)