Lecture 7: Priority Queues and Heaps
Outline

- Priority queue ADT
- 2 Simple List-based implementation for Priority Queue
  - not as efficient as possible
- Heap Data structure
  - First moderately complex data structure that you haven’t seen before
- Heap-based implementation for Priority Queue
  - very efficient
Priority

- Frequently elements that we wish to store in a data structure have “priorities”
- Operations should be done in order of the priority
- Examples
  - air-traffic control, each flight to clear for landing has priority depending on its distance from the airport, amount of fuel left, etc.
  - standby passengers for a full flight have different based on their frequent-flyer status, check-in time, etc.
  - shared printer may assign priorities to documents based on time submitted, size of the document, seniority of the user, etc.
Priority Queue ADT

- A **priority queue** is an abstract data type for storing a collection of prioritized elements.
- Two main methods:
  - insertion of arbitrary element
  - removal of element of highest priority
- In our implementation, a priority queue stores **entries**
- Like for dictionaries, each **entry** is a pair (**key**, **value**)
  - **key** is the **priority** associated with the entry
  - unless stated otherwise, assume smaller **key** corresponds to higher priority
Priority Queue ADT

- Main methods
  - `insert(k, v)`
    inserts an entry with key `k` and value `v`
  - `removeMin()`
    removes and returns the entry with smallest key

- Additional methods
  - `min()`
    return, but do not remove, entry with smallest key
  - `size()`
  - `isEmpty()`
Keys and Comparators

- Two distinct entries in a priority queue can have the same key
- Keys in can be arbitrary objects with defined order
- General priority queue uses an external comparator object
  - comparator is external to the keys being compared
- Primary method of Comparator Interface
  - `compare(a, b)`: returns an integer i such that
    - $i < 0$ if $a < b$,
    - $i = 0$ if $a = b$,
    - $i > 0$ if $a > b$
Sequence-based Priority Queue

- Unsorted list implementation

  4 → 5 → 2 → 3 → 1

  - Performance
    - `insert` takes $O(1)$ time since can insert item at sequence beginning (or end)
    - `removeMin` and `min` take $O(n)$ time since have to traverse entire sequence to find the smallest key

- Sorted list implementation

  1 → 2 → 3 → 4 → 5

  - Performance
    - `insert` takes $O(n)$ time since have to find proper place to insert
    - `removeMin` and `min` take $O(1)$ time, since the smallest key is at sequence beginning
Sequence-based Priority Queue

- Implementation with an unsorted list
  - fast inserts $O(1)$
  - slow removals $O(n)$

- Implementation with a sorted list
  - slow inserts $O(n)$
  - fast removals $O(1)$

- Can we balance running time between insertion and removal to achieve better efficiency on both?
- Yes, with the data structure called heap!
Complete Binary Tree

- Note: different from full binary tree
- A binary tree with height $h$ is complete if
  1. Has the maximum possible number of nodes at levels 0, 1, ..., $h-1$
      - that is level $i$ has $2^i$ nodes
  2. at level $h$, all nodes must be as far to the left as possible
Theorem: A complete tree of size $n$ has height $O(\log n)$

- Let $h$ be the height of a complete binary tree of size $n$
- Exactly $2^i$ nodes at depth $i = 0, \ldots, h - 1$ and at least one node at depth $h$
- Thus $n \geq 1 + 2^1 + 2^2 + \ldots + 2^{h-1} + 1$
- Thus $n \geq 2^h$
- Take logarithm of both sides: $h \leq \log n$
Heaps

- A heap is a complete binary tree that satisfies Heap-Order
  - for every non-root node \(v\)
    \[\text{key}(v) \geq \text{key}(\text{parent}(v))\]
- Heap-order implies that each path in the tree is sorted
Heaps

- Since heap is a complete binary tree $h \leq \log n$
  - $n$ is the number of heap entries
  - $h$ is the heap height
- Path length is logarithmic in the number of nodes
- Define the last heap node as the rightmost node of depth $h$
Heaps and Priority Queues

- Will use a heap to implement a priority queue
- Store entry = (key, element) at each node
- Will keep track of the position of the last node

```
last node

(2, Sue)

(5, Pat) (6, Mark)

(9, Jeff) (7, Anna)
```
Insertion into a Heap

- Method $\text{insert}(k,v)$
- Three steps
  - find insertion node $z$ (the new last node)
  - store $(k,v)$ at $z$, this likely violates the heap order
  - Restore heap-order with $\text{upheap}$

![Diagram of a heap with nodes labeled 1 to 9 and an insertion node Z at the bottom right.](image)
Upheap

- Restores heap-order by swapping $k$ along an upward path from the insertion node.
- Terminates when the key $k$ reaches the root or a node whose parent has a key smaller than or equal to $k$.
- Example
  - insert entry with key 1
Upheap: Another Example

- Insert entry with key 3
- Upheap terminates when key $k$ reaches the root or node whose parent has key smaller than or equal to $k$
- In the worst case, traverse a full path from last level to the root
  - perform a constant number of operations at each node
- All paths have height $O(\log n)$
- Upheap and insert have $O(\log n)$ worst case time
Removal from a Heap

- Method `removeMin()`
- Three steps
  - replace the root entry with the entry at the last node \( w \), and find the new “last node”
  - remove node \( w \)
  - restore the heap-order with `downheap`
Downheap

- Restores heap-order by swapping entry \((k, v)\) along a downward path from the root
  - most nodes have 2 children
  - swap with smallest key child
- Terminates when key \(k\) reaches a leaf or a node whose children have keys greater than or equal to \(k\)
- Traverses one path in the worst case
  - \(O(1)\) operations at each node
- downheap runs in \(O(\log n)\) time
- removeMin() runs in \(O(\log n)\) time
Downheap: Another Example
Array-based Heap Implementation

- Array-based implementation is the best choice
  - no need to store links between parent-child
- Store root at rank 1
- For the node at rank \( i \)
  - the left child is at rank \( 2i \)
  - the right child is at rank \( 2i + 1 \)
  - its parent is at rank \( i/2 \), integer division
- Last node is always at rank \( n \)
  - In insert, put new entry at rank \( n + 1 \) and perform upheap
  - In removeMin, replace entry at rank 1 with entry at rank \( n \), and perform downheap
- For \( n \) entries, need array of size \( n + 1 \)
  - use an “expandable” array
Heap Insertion

Algorithm InsertInHeap(k,v)
Input: A priority k, value v;
Output: none

size = size + 1  // Increase heap size
H[size] = new entry(k,v)  // Insert entry (k,v) at rank = array size

// Now perform upheap, starting at the last tree node
i = size
while i > 1 and H[i/2].key() > k
    swap(H[i], H[i/2])  // Swap entry (k,v) with parent node entry
    i = i/2  // Move to parent node
Algorithm `removeMin()`

Input: none

Output: entry with the smallest key

```plaintext
if size == 0 then ReportErrorSomehow
itemToReturn = H[1]    // minimum is always at rank 1
H[1] = H[size]         // put entry at last rank into the root location
size = size - 1       // decrease heap size
// Now restore heap order with downheap
i = 1
childIndex = findSmallestKeyChild(i)
while childIndex != 0 && H[childIndex].key < H[i].key
    swap(H[childIndex],H[i])
    i = childIndex
childIndex = findSmallestKeyChild(i)
return itemToReturn
```
Algorithm `findSmallestKeyChild (i)`

Input: index i of a node

Output: index of the child of node i with smallest key, or 0 if node i is a leaf

```
if (2*i < size)
    // node at index i has 2 children
    if H[2*i].key < H [2*i+1].key then
        return(2*i)
    else
        return(2*i+1)
else if (2*i == size)
    // node at index i has only a left child
    return(2*i)
else
    // node at index i is a leaf
    return(0)
```