csci 210: Data Structures

Priority Queues and Heaps
Summary

• Topics
  • the Priority Queue ADT
  • Priority Queue vs Dictionary and Queues
  • implementation of PQueue
    • linked lists
    • binary search trees
    • heaps
  • Heaps
Priority Queues

- A Priority Queue is an abstract data structure for storing a collection of prioritized elements
- The elements in the queue consist of a value v with an associated priority or key k
  - element = (k,v)
- A priority queue supports
  - arbitrary element insertion: insert value v with priority k
    - insert(k, v)
  - delete elements in order of their priority: that is, the element with the smallest priority can be removed at any time
    - removeMin()
- Priorities are not necessarily unique: there can be several elements with same priority

- Examples: store a collection of company records
  - compare by number of employees
  - compare by earnings

- The priority is not necessarily a field in the object itself. It can be a function computed based on the object. For e.g., priority of standby passengers is determined as a function of frequent flyer status, fare paid, check-in time, etc.
Priority Queues

• Examples
  • Queue of jobs waiting for the processor
  • Queue of standby passengers waiting to get a seat
  • ...

• Note: the keys must be “comparable” to each other

• PQ Queue ADT
  • size()
    • return the number of entries in PQ
  • isEmpty()
    • test whether PQ is empty
  • min()
    • return (but not remove) the entry with the smallest key
  • insert(k, x)
    • insert value x with key k
  • removeMin()
    • remove from PQ and return the entry with the smallest key
Priority Queue example

(k,v) key=integer, value=letter

PQ={}

• insert(5,A)   PQ={(5,A)}
• insert(9,C)   PQ={(5,A), (9,C)}
• insert(3,B)   PQ={(5,A), (9,C), (3,B)}
• insert(7,D)   PQ={(5,A), (7,D), (9,C), (3,B)}
• min()         return (3,B)
• removeMin()   PQ = {(5,A), (7,D), (9,C)}
• size()        return 3
• removeMin()   return (5,A)   PQ={(7,D), (9,C)}
• removeMin()   return (7,D)   PQ={(9,C)}
• removeMin()   return (9,C)   PQ={}

return (3,B)
Priority queue implementations

- unsorted linked list
  - fast insertions, slow deletions

- sorted linked list
  - fast deletions, slow insertions

- (balanced) binary search trees
  - $O(lg n)$

- (binary) heaps
  - $O(lg n)$
  - simpler
Heaps

- A heap is an array viewed as a complete binary tree, level by level:
  - As a consequence, children positions can be computed without storing references
    - root has index 1
    - left(i) = 2i
    - right(i) = 2i + 1
    - parent(i) = i / 2

- Each node satisfies the heap property:
  - the keys of v’s children are >= the key of v

- As a consequence, the keys encountered on a root-to-leaf traversal are in increasing order (or equal); the smallest key is stored at the top.
Heaps

- Proposition: A heap $T$ storing $n$ elements has height $h = \lg_2 n$.

- insert($k,v$)
  - insert it at last position in the heap, and “trickle” it up (swap node with parent up the leaf-root path)

- deleteMin()
  - take the last element and put it in the root
  - this will violate the heap property, so “trickle” it down: swap the node with the smaller if its 2 children, and repeat

- insert and deleteMin take $O(h) = O(\lg n)$
Sort with a Priority Queue

- An important application of a priority queue is sorting.

**PriorityQueueSort** (collection S of n elements)
- put the elements in S in an initially empty priority queue by means of a series of n insert() operations on the pqueue, one for each element
- extract the elements from the pqueue by means of a series of n removeMin() operations

**Pseudocode for PriorityQueueSort(S)**
- input: a collection S storing n elements
- output: the collection S sorted
- P = new PQueue()
- while !S.isEmpty() do
  - e = S.removeFirst()
  - P.insert(e)
- while !P.isEmpty() do
  - e = P.removeMin()
  - S.addLast(e)
Heapsort

- sort with a heap
  - insert all elements
  - deleteMin n times

- time: \( O(n \lg n) \)

- Optimizations:
  - Constructing the heap can be improved so that it takes \( O(n) \) time (instead of \( O(n \lg n) \)), but the overall running time of the heapsort stays the same
    - idea: convert the array into a heap bottom up
  - The whole sort can be done “in place” (assume the input is stored in an array \( A \); you want to re-arrange the array \( A \) to be in sorted order, without creating a new array.)
    - use a max-heap instead of a min-heap (the heap property is reversed and the max element is stored at top)
    - repeatedly deleteMax
    - the heap shrinks by one every time, and at the end \( A[] \) is sorted