Readings and Homework

Read Chapter 26 “A Heap Implementation” to learn about heaps

Exercise: Salespeople often make matrices that show all the great features of their product that the competitor’s product lacks. Try this for a heap versus a BST. List some desirable properties of a BST that a heap lacks. Now be the heap salesperson: List some good things about heaps that a BST lacks. Can you think of situations where you would favor one over the other?

With ZipUltra heaps, you’ve got it made in the shade my friend!

The Bag Interface

A Bag:

```
interface Bag<E> {
    void insert(E obj);
    E extract(); //extract some element
    boolean isEmpty();
}
```

Like a Set except that a value can be in it more than once. Example: a bag of coins

Refinements of Bag: Stack, Queue, PriorityQueue

Stacks and Queues as Lists

• Stack (LIFO) implemented as list
  – `insert()`, `extract()` from front of list
• Queue (FIFO) implemented as list
  – `insert()` on back of list, `extract()` from front of list
• These operations are O(1)

Priority Queue

• A Bag in which data items are Comparable

• Lesser elements (as determined by `compareTo()`) have higher priority

• `extract()` returns the element with the highest priority = least in the `compareTo()` ordering

• break ties arbitrarily
Examples of Priority Queues

- Scheduling jobs to run on a computer
  default priority = arrival time
  priority can be changed by operator
- Scheduling events to be processed by an event handler
  priority = time of occurrence
- Airline check-in
  first class, business class, coach
  FIFO within each class

java.util.PriorityQueue<E>

- boolean add(E e) {...} //insert an element (insert)
- void clear() {...} //remove all elements
- E peek() {...} //return min element without removing
  // (null if empty)
- E poll() {...} //remove min element (extract)
  // (null if empty)
- int size() {...}

Priority Queues as Lists

- Maintain as unordered list
  - insert() put new element at front – O(1)
  - extract() must search the list – O(n)
- Maintain as ordered list
  - insert() must search the list – O(n)
  - extract() get element at front – O(1)
- In either case, O(n^2) to process n elements
  Can we do better?

Important Special Case

- Fixed number of priority levels 0,...,p – 1
- FIFO within each level
- Example: airline check-in
  - insert() – insert in appropriate queue – O(1)
  - extract() – must find a nonempty queue – O(p)

Heaps

- A heap is a concrete data structure that can be used to implement priority queues
- Gives better complexity than either ordered or unordered list implementation:
  - insert(): O(log n)
  - extract(): O(log n)
- O(n log n) to process n elements
- Do not confuse with heap memory, where the Java
  virtual machine allocates space for objects – different
  usage of the word heap

Heaps

- Binary tree with data at each node
- Satisfies the Heap Order Invariant:

  The least (highest priority) element of any subtree is found at the root of that subtree.

- Size of the heap is “fixed” at n. (But can usually double
  n if heap fills up)
Smallest element in any subtree is always found at the root of that subtree.

Note: 19, 20 < 35: Smaller elements can be deeper in the tree!

Ages of people in family tree
- parent is always older than children, but you can have an uncle who is younger than you

Salaries of employees of a company
- bosses generally make more than subordinates, but a VP in one subdivision may make less than a Project Supervisor in a different subdivision

Balanced Heaps

These add two restrictions:
1. Any node of depth < d – 1 has exactly 2 children, where d is the height of the tree
   - implies that any two maximal paths (path from a root to a leaf) are of length d or d – 1, and the tree has at least $2^d$ nodes
   - All maximal paths of length d are to the left of those of length d – 1

Example of a Balanced Heap

Store in an ArrayList or Vector

- Elements of the heap are stored in the array in order, going across each level from left to right, top to bottom
- The children of the node at array index n are at indices $2n + 1$ and $2n + 2$
- The parent of node n is node $(n - 1)/2$

Store in an ArrayList or Vector

- children of node n are found at $2n + 1$ and $2n + 2$
Store in an ArrayList or Vector

- children of node n are found at 2n + 1 and 2n + 2

**insert()**

- Put the new element at the end of the array
- If this violates heap order because it is smaller than its parent, swap it with its parent
- Continue swapping it up until it finds its rightful place
- The heap invariant is maintained!
• Time is $O(\log n)$, since the tree is balanced
  – size of tree is exponential as a function of depth
  – depth of tree is logarithmic as a function of size

```java
class PriorityQueue<E> extends java.util.Vector<E> {

  /** Insert e into the priority queue */
  public void insert(E e) {
    super.add(e); // add to end of array
    bubbleUp(size() - 1); // given on next slide
  }

  private void bubbleUp(int k) {
    int p = (k-1)/2; // p is the parent of k
    // inv: Every element satisfies the heap property
    // except element k might be smaller than its parent
    while (k>0 && get(k).compareTo(get(p)) < 0) {
      "swap elements k and p";
      k = p;
      p = (k-1)/2;
    }
  }
}
```

• Remove the least element – it is at the root
• This leaves a hole at the root – fill it in with the last element of the array
• If this violates heap order because the root element is too big, swap it down with the smaller of its children
• Continue swapping it down until it finds its rightful place
• The heap invariant is maintained!
extract()
extract()

4 5
6
21 14
22 38 55 10 20

extract()

4 5
6
21 14
22 38 55 10

extract()

4 5
6
21 14
22 38 55 10

extract()

4 5
6
21 14
22 38 55 10 20

extract()

4 5
6
21 14
22 38 55 10

extract()

4 5
6
21 14
22 38 55 20
**extract()**

Time is $O(\log n)$, since the tree is balanced.

```java
/** Remove and return the smallest element
 * (return null if list is empty) */

public E extract() {
    if (size() == 0) return null;
    E temp = get(0); // smallest value is at root
    set(0, get(size() - 1)); // move last element to root
    setSize(size() - 1); // reduce size by 1
    bubbleDown(0);
    return temp;
}
```

**HeapSort**

Given a `Comparable[]` array of length $n$,
- Put all $n$ elements into a heap – $O(n \log n)$
- Repeatedly get the min – $O(n \log n)$

```java
public static void heapSort(Comparable[] b) {
    PriorityQueue<Comparable> pq = new PriorityQueue<>(b);
    for (int i = 0; i < b.length; i++) {
        b[i] = pq.extract();
    }
}
```

One can do the two stages in the array itself, in place, so algorithm takes $O(1)$ space.

---

**Many uses of priority queues & heaps**

- Mesh compression: quadratic error mesh simplification
- Event-driven simulation: customers in a line
- Collision detection: "next time of contact" for colliding bodies
- Data compression: Huffman coding
- Graph searching: Dijkstra's algorithm, Prim's algorithm
- AI Path Planning: A* search
- Statistics: maintain largest $M$ values in a sequence
- Operating systems: load balancing, interrupt handling
- Discrete optimizations: bin packing, scheduling
- Spam filtering: Bayesian spam filter

---

**extract()**

```java
/** Remove and return the smallest element
 * (return null if list is empty) */

public E extract() {
    if (size() == 0) return null;
    E temp = get(0); // smallest value is at root
    set(0, get(size() - 1)); // move last element to root
    setSize(size() - 1); // reduce size by 1
    bubbleDown(0);
    return temp;
}
```