Priority Queues and Heaps

15-211
Fundamental Data Structures and Algorithms

Ananda Guna
February 04, 2003

Priority Queues and Heaps

Definition of Priority Queue

Definition: An abstract data type to efficiently support finding the item with the highest priority across a series of operations. The basic operations are: insert, find-minimum (or maximum), and delete-minimum (or maximum).

Priority Queue

- P-queue is a data structure that allows:
  - Insertion and deleteMin in O(\log n)
  - O(1) findMin operation

- Applications
  - Operating System Design – resource allocation
  - Data Compression - Huffman algorithm
  - Discrete Event simulation
    - 1) Insertion of time-tagged events (time represents a priority of an event – low time means high priority)
    - 2) Removal of the event with the smallest time tag

- Implementation
  - Linked Lists
  - Using a binary Heap – a special binary tree with heap property

Priority queue Operations

- new
  - Create a new priority queue.

- insertItem(x)
  - Insert object x into the p-queue.

- minElement()
  - Return the minimum element from the p-queue.

- removeMin()
  - Return and remove the minimum element from the p-queue.

Some questions

- How do we ensure that there is a concept of a “minimum”?
- What should happen in minElement() and removeMin() if the priority queue is empty?
- How long does it take to perform operations like insertItem(x) and removeMin()?

Comparing Objects
Comparing objects

- In Java, objects can be compared for equality:

```java
public void doSomething (Person x, Person y) {
    if (x == y) { ... }
}
```

What does it mean for two objects to be equal?

Comparing objects

- Note that this is an issue only for objects.
- Values of base type (such as int, float, char, etc.) have built-in comparison operations ==, <, <=, ...
- But javac can’t possibly know how to compare objects.
  - E.g., Is a>b where a and b are objects

Suppose we want to put objects of class Person into our priority queue.
- What we can do is require that every Person object has a method that computes whether it is bigger, smaller, or equal to another Person object.
- The JDK has a built-in interface just for this purpose, called Comparable.

```
public interface Comparable {
    public int compareTo (Object obj);
}
```

Returns <0 if object is less than obj,
=0 if object is equal to obj,
>0 if object is greater than obj.

A caution

- Note that the compareTo() method takes any object (not just Person objects, for example).

```
Gorilla a = new Gorilla ("Freddy");
Person b = new Person ("Matt");
if (a.compareTo(b)) { ... }
```

- If a comparison makes no sense at all, then by convention the exception ClassCastException is raised.
Exceptional Conditions

**Some questions**

- How do we ensure that there is a concept of a “minimum”?
- What should happen in `minElement()` and `removeMin()` if the priority queue is empty?
- How long does it take to perform operations like `insertItem(x)` and `removeMin()`?

**One possibility**

- If `removeMin()` is applied to an empty priority queue, it could return null.
  
  - **Pro**: Simple.
  - **Con**: May require that all calls to `removeMin()` check for null.

**An alternative**

- A common approach is to raise an exception.

```java
public class PriorityQueue {
    ...
    public int removeMin() throws PriorityQueueEmptyException {
        ...
        if (isEmpty())
            throw new PriorityQueueException("Empty priority queue in removeMin()");
        ...
    }
    ...
}
```

**Exception classes**

```java
public class PriorityQueueException extends Exception {
    public PriorityQueueException() {
        super();
    }
    public PriorityQueueException(String s) {
        super(s);
    }
}
```

More on this later...

Implementation p-queue
Using Binary Trees

- We expect the \textit{find} and \textit{insert} operations to take \(O(\log N)\) time.
- In fact, operations like \textit{find} take time \(d\), where \(d\) is the depth of the item in the tree.
- Since the depth is not expected to be larger than \(\log(N)\), and each step down the tree requires constant time, we get \(O(\log N)\). More later..

Analysis of BSTs

- If all insertion sequences are equally likely (that is, the insertion order is random), then on average a binary search tree has depth \(O(\log_2(N))\).
- Define \(D(N)\) to be the sum of the depths of all nodes in a tree with \(N\) nodes.
  - \(D(1) = 0\).

Analysis, cont’d

- For a tree with \(N>1\) nodes:
  - \(i\) nodes in left subtree,
  - \(N-i-1\) nodes in the right subtree,
  - and one node at root. (for \(0 \leq i < N\))

Analysis, cont’d

- So,
  - \(D(N) = D(i) + D(N-i-1) + N - 1\)

Analysis, cont’d

- The average value of \(D(i)\) and \(D(N-i-1)\) is
  - \(\sum_{j=0}^{N} \frac{D(j)}{N}\)

- So, \(D(N) = 2\left(\sum_{j=0}^{N} \frac{D(j)}{N}\right) + N - 1\)

Analysis, cont’d

- There are methods for solving such \textit{recurrence equations}.
- We shall see later that this equation has the solution \(O(N\times \log N)\).
- Thus, on average the depth of any particular node is \(O(\log N)\).

Priority queue implementation

- Linked list
  - \(\text{removeMin} O(1)\) \(\text{insertItem} O(N)\)
- Heaps
  - \(\text{avg}\) \(\text{worst}\)
  - \(\text{deleteMin} O(\log N)\)
  - \(\text{insert} 2.6\)
  - \(\text{build} O(N)\)
  - \(\text{special case: insert*N}\)
  - \(\text{i.e., insert*N}\)
Heaps
- A binary tree.
- Representation invariant
  1. Structure property
     - Complete binary tree
  2. Heap order property
     - Parent keys less than children keys

Heaps
- Representation invariant
  1. Structure property
     - Complete binary tree
     - Hence: efficient compact representation
  2. Heap order property
     - Parent keys less than children keys
     - Hence: rapid insert, findMin, and deleteMin
     - $O(\log(N))$ for insert and deleteMin
     - $O(1)$ for findMin

Perfect binary trees
- How many nodes?
  - $N = 2^h - 1 = 15$
  - In general: $N = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$
  - Most of the nodes are leaves

Perfect binary trees
- What is the sum of the heights?
  \[ S = \sum_{i=0}^{h} 2^{(h-i)} = O(N) \]
  - prove this

Complete binary trees
Complete binary trees

Representing complete binary trees

• Linked structures?  No!
• Arrays!

Representing complete binary trees

• Arrays
  ➢ Parent at position $i$
  ➢ Children at $2i$ and $2i+1$.

Representing complete binary trees

• Arrays (1-based)
  ➢ Parent at position $i$
  ➢ Children at $2i$ and $2i+1$.

Representing complete binary trees

• Arrays (1-based)
  ➢ Parent at position $i$
  ➢ Children at $2i$ and $2i+1$. 

1  2  3  4  5  6  7  8  9  10
Representing complete binary trees

- **Arrays (1-based)**
  - Parent at position $i$
  - Children at $2i$ and $2i+1$.

```java
public class BinaryHeap {
    private Comparable[] heap;
    private int size;
    public BinaryHeap(int capacity) {
        size = 0;
        heap = new Comparable[capacity+1];
    }
    // ...}
```

**Example: find the leftmost child**

```java
int left = 1;
for(; left < size; left *= 2);
return heap[left/2];
```

**Example: find the rightmost child**

```java
int right = 1;
for(; right < size; right = right * 2 + 1);
return heap[(right - 1)/2];
```

---

**Heaps**

- **Representation invariant**
  1. **Structure property**
     - **Complete binary tree**
     - **Hence**: efficient compact representation
  2. **Heap order property**
     - **Parent keys less than children keys**
     - **Hence**: rapid insert, findMin, and deleteMin
     - $O(\log(N))$ for insert and deleteMin
     - $O(1)$ for findMin

---

**The heap order property**

- **Each parent is less than each of its children.**
- **Hence**: Root is less than every other node.
- **Proof by induction**

---

**Operating with heaps**

**Representation invariant**:

- **All methods must**:  
  1. Produce complete binary trees  
  2. Guarantee the heap order property

- **All methods may assume**
  1. The tree is initially complete binary  
  2. The heap order property holds
**findMin()**

- The code
  ```java
  public boolean isEmpty() {
      return size == 0;
  }
  public Comparable findMin() {
      if(isEmpty()) return null;
      return heap[1];
  }
  - Does not change the tree
    - Trivially preserves the invariant
  ```

**insert (Comparable x)**

- Process
  1. Create a "hole" at the next tree cell for x.
     ```java
     heap[size+1]
     ```
    This preserves the completeness of the tree.
  2. Percolate the hole up the tree until the heap order property is satisfied.
    This assures the heap order property is satisfied.

**insert (Comparable x)**

- Process
  1. Create a "hole" at the next tree cell for x.
     ```java
     heap[size+1]
     ```
    This preserves the completeness of the tree assuming it was complete to begin with.
  2. Percolate the hole up the tree until the heap order property is satisfied.
    This assures the heap order property is satisfied assuming it held at the outset.

**Percolation up**

```java
public void insert(Comparable x) throws Overflow {
    if(isFull()) throw new Overflow();
    int hole = ++size;
    for(; hole>1 && x.compareTo(heap[hole/2])<0; hole/=2)
        heap[hole] = heap[hole/2];
    heap[hole] = x;
}
```

**Percolation up**

- Bubble the hole up the tree until the heap order property is satisfied.
Percolation up
- Bubble the hole up the tree until the heap order property is satisfied.

Percolation down
- Bubble the transplanted leaf value down the tree until the heap order property is satisfied.

**deleteMin()**
```java
/**
* Remove the smallest item from the priority queue.
* @return the smallest item, or null, if empty.
*/
public Comparable deleteMin() {
    if(isEmpty()) return null;
    Comparable min = heap[1];
    heap[1] = heap[size--];
    percolateDown(1);
    return min;
}
```

Percolation down
- Bubble the transplanted leaf value down the tree until the heap order property is satisfied.

Percolation down
- Bubble the transplanted leaf value down the tree until the heap order property is satisfied.

**deleteMin()**
- Observe that both components of the representation invariant are preserved by deleteMin.
  1. Completeness
  -
  -
  2. Heap order property
deleteMin()

- Observe that both components of the representation invariant are preserved by deleteMin.
  1. Completeness
     - The last cell (heap[size]) is vacated, providing the value to percolate down.
     - This assures that the tree remains complete.
  2. Heap order property

buildHeap

- Equivalent to a sequence of inserts
  
  ```
  for (int i = 0; i < N; i++)
    insert(input[i]);
  ```

- Two steps:
  1. Fill the array (in no particular order).
  2. percolateDown, bottom up.
    
    ```
    for (int i = size/2; i > 0; i --)
      percolateDown(i);
    ```
    
    - This does a linear number of comparisons

Thursday

- We will talk about Greedy Algorithms
- Read Chapter 7
- HW3 is online now
  
  - You must read homework assignment before recitation tomorrow
- Start Early
- Ask Questions Early
- Go to Recitation Tomorrow