CSE 326: Data Structures

Priority Queues – Binary Heaps

Hal Perkins
Spring 2007
Lectures 3 & 4

Outline

• Admin (proj #1)
• Math/Big-O – short summary
• Priority Queues (Binary Min Heaps)
  – Reading: Weiss, Ch. 6

Project #1 Turn-in

• The turnin page for project 1 is now linked to the project page.
• Turn in your electronic documents before midnight on Wednesday.
• Turn in hardcopies in sections on Thursday (whichever section you normally attend).

Simplifying Recurrences

Given a recursive equation for the running time, can sometimes simplify it for analysis.

• For an upper-bound analysis, can optionally simplify to something larger, e.g.
  \[ T(n) = T(\text{floor}(n/2)) + 1 \quad \text{to} \quad T(n) \leq T(n/2) + 1 \]

• For a lower-bound analysis, can optionally simplify to something smaller, e.g.
  \[ T(n) = 2T(n/2 + 5) + 1 \quad \text{to} \quad T(n) \geq 2T(n/2) + 1 \]
The One Page Cheat Sheet

• Calculating series:
  e.g. \[ \sum_{i=1}^{n} \frac{n(n+1)}{2} \]
  1. Brute force (Section 1.2.3)
  2. Induction (Section 1.2.5)
  3. Memorize simple ones!

• Solving recurrences:
  e.g. \[ T(n) = T(n/2) + 1 \]
  1. Expansion (example in class)
  2. Induction (Section 1.2.5)
  3. Telescoping (later…)

• General proofs (Section 1.2.5)
  e.g. How many edges in a tree with \( n \) nodes?
  1. Counterexample
  2. Induction
  3. Contradiction

Priority Queue ADT

• Checkout line at the supermarket ???
• Printer queues ???
• operations: insert, deleteMin

Priority Queue ADT

1. \textbf{PQueue data}: collection of data with \textit{priority}
2. \textbf{PQueue operations}
   – insert
   – deleteMin
   (also: create, destroy, is_empty)
3. \textbf{PQueue property}: for two elements in the queue, \( x \) and \( y \), if \( x \) has a \textbf{lower} priority value than \( y \), \( x \) will be deleted before \( y \)
Applications of the Priority Q

- Select print jobs in order of decreasing length
- Forward packets on network routers in order of urgency
- Select most frequent symbols for compression
- Sort numbers, picking minimum first

- Anything greedy

Implementations of Priority Queue ADT

<table>
<thead>
<tr>
<th>Method</th>
<th>Insert</th>
<th>Delete Min</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unssorted list (Array)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unssorted list (Linked-List)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorted list (Array)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorted list (Linked-List)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary Search Tree (BST)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tree Review

- root(T):
- leaves(T):
- children(B):
- parent(H):
- siblings(E):
- ancestors(F):
- descendents(G):
- subtree(C):

More Tree Terminology

- depth(T):
- height(G):
- degree(B):
- branching factor(T):

4/5/2007 9

4/5/2007 10

4/5/2007 11

4/5/2007 12
Some More Tree Terminology

- **T is binary if** …
- **T is n-ary if** …
- **T is complete if** …

How deep is a complete tree with \( n \) nodes?

**Full Binary Tree**

- A binary tree in which each node has **exactly zero or two children.**
- (also known as a proper binary tree)
- (we will use this later for Huffman trees)

**Binary Heap Properties**

1. Structure Property
2. Ordering Property

Brief interlude: Some Definitions:

**A Perfect** binary tree – A binary tree with all leaf nodes at the same depth. All internal nodes have 2 children.

- height \( h \)
- \( 2^{h-1} \) nodes
- \( 2^h - 1 \) non-leaves
- \( 2^h \) leaves

Full Binary Tree

- A binary tree in which each node has **exactly zero or two children.**
- (also known as a proper binary tree)
- (we will use this later for Huffman trees)
Heap **Structure** Property

- A binary heap is a *complete* binary tree.

**Complete binary tree** – binary tree that is completely filled, with the possible exception of the bottom level, which is filled left to right.

Examples:

![Binary Tree Diagram](image)

Representing Complete Binary Trees in an Array

From node `i`:
- left child: `2i + 1`
- right child: `2i + 2`
- parent: `floor((i - 1) / 2)`

Implicit (array) implementation:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td>2</td>
<td></td>
<td>3</td>
<td></td>
<td>4</td>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>8</td>
<td>9</td>
<td></td>
<td>10</td>
<td></td>
<td>11</td>
<td></td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Why better than tree with pointers?

Heap **Order** Property

**Heap order property**: For every non-root node `X`, the value in the parent of `X` is less than (or equal to) the value in `X`.

![Heap Diagram](image)

not a heap
Heap Operations

- findMin:
- insert(val): percolate up.
- deleteMin: percolate down.

Heap – Insert(val)

Basic Idea:
1. Put val at “next” leaf position
2. Repeatedly exchange node with its parent if needed

Insert: percolate up

Insert pseudo/C++ Code (optimized)

```cpp
void insert(Object o) {
    assert(!isFull());
    size++;
    newPos = percolateUp(size, o);
    Heap[newPos] = o;
}
```

```cpp
int percolateUp(int hole, Object val) {
    while (hole > 1 && val < Heap[hole/2]) {
        Heap[hole] = Heap[hole/2];
        hole /= 2;
    }
    return hole;
}
```

runtime:
Heap – Deletemin

Basic Idea:
1. Remove root (that is always the min!)
2. Put “last” leaf node at root
3. Find smallest child of node
4. Swap node with its smallest child if needed.
5. Repeat steps 3 & 4 until no swaps needed.

DeleteMin pseudo/C++ Code (Optimized)

```cpp
Object deleteMin() {
    assert(!isEmpty());
    returnVal = Heap[1];
    size--;
    newPos =
        percolateDown(1,
                    Heap[size+1]);
    Heap[newPos] =
        Heap[size + 1];
    return returnVal;
}

int percolateDown(int hole,
                   Object val) {
    while (2*hole <= size) {
        left = 2*hole;
        right = left + 1;
        if (right <= size &&
            Heap[right] < Heap[left])
            target = right;
        else
            target = left;
        if (Heap[target] < val) {
            Heap[hole] = Heap[target];
            hole = target;
        } else
            break;
    }
    return hole;
}
```

runtime:

(Java code in book)
More Priority Queue Operations

• **decreaseKey**
  – given a pointer to an object in the queue, reduce its priority value
  Solution: change priority and ____________________________

• **increaseKey**
  – given a pointer to an object in the queue, increase its priority value
  Solution: change priority and _____________________________

Why do we need a pointer? Why not simply data value?

More Heap Operations

decreaseKey(objPtr, amount): raise the priority of a object, percolate up
increaseKey(objPtr, amount): lower the priority of a object, percolate down
remove(objPtr): remove a object, move to top, them delete. 1) decreaseKey(objPtr, ∞)
  2) deleteMin()

Worst case Running time for all of these:
FindMax?
ExpandHeap – when heap fills, copy into new space.

More Priority Queue Operations

• **Remove(objPtr)**
  – given a pointer to an object in the queue, remove it
  Solution: set priority to negative infinity, percolate up to root and deleteMin

• **buildHeap**
  Naïve solution: Running time:
  Can we do better?

More Heap Operations

buildHeap: Floyd’s Method

Add elements arbitrarily to form a complete tree.
Pretend it’s a heap and fix the heap-order property!
Buildheap pseudocode

```java
private void buildHeap() {
    for ( int i = currentSize/2; i > 0; i-- )
        percolateDown( i );
}
```

**BuildHeap: Floyd’s Method**

**Facts about Heaps**

**Observations:**
- finding a child/parent index is a multiply/divide by two
- operations jump widely through the heap
- each percolate step looks at only two new nodes
- inserts are at least as common as deleteMins

**Realities:**
- division/multiplication by powers of two are equally fast
- looking at only two new pieces of data: bad for cache!
- with huge data sets, disk accesses dominate
A Solution: $d$-Heaps

- Each node has $d$ children
- Still representable by array
- Good choices for $d$:
  - (choose a power of two for efficiency)
  - fit one set of children in a cache line
  - fit one set of children on a memory page/disk block

Operations on $d$-Heap

- Insert : runtime =

- deleteMin: runtime =

Does this help insert or deleteMin more?

One More Operation

- Merge two heaps. Ideas?