CMPSC250
Lecture 7: Priority Queues and Heapsort

Prof. John Wenskovitch
02/03/2015
Last Time

• What is the best sorting that we can expect to achieve?
  – Compare-based algorithms and expressing those comparisons as a tree.

• QuickSort and Partitioning

• Survey
## Sorting Summary

<table>
<thead>
<tr>
<th>Sort</th>
<th>Compares</th>
<th>Exchanges/Array Accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Sort</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Insertion Sort (Worst)</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Insertion Sort (Average)</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Insertion Sort (Best)</td>
<td>$O(n)$</td>
<td>0</td>
</tr>
<tr>
<td>Shell Sort (Worst)</td>
<td>$O(n^{3/2})$</td>
<td>$O(n^{3/2})$</td>
</tr>
<tr>
<td>Shell Sort (Average)</td>
<td>It depends</td>
<td>It depends</td>
</tr>
<tr>
<td>Top-Down Mergesort (Worst)</td>
<td>$O(n \log(n))$</td>
<td>$O(n \log(n))$</td>
</tr>
<tr>
<td>Top-Down Mergesort (Average)</td>
<td>$O(n \log(n))$</td>
<td>$O(n \log(n))$</td>
</tr>
<tr>
<td>Bottom-Up Mergesort</td>
<td>$O(n \log(n))$</td>
<td>$O(n \log(n))$</td>
</tr>
<tr>
<td>Quicksort (Worst)</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Quicksort (Average)</td>
<td>$O(n \log(n))$</td>
<td>$O(n \log(n))$</td>
</tr>
</tbody>
</table>
Online Data Acquisition

• Thus far, we have assumed that we have all of the data before we begin to sort.
  – What if we expect to continue to receive data throughout the sorting process? We’ll never get a totally sorted list.

• We also assumed that producing a fully sorted list was the end goal.
  – What if we don’t need everything sorted from smallest to biggest; what if we just want the smallest item in the array right now?
Priority Queue

• Two operations necessary:
  – Acquire new data (\texttt{insert()})
  – Remove the largest item (\texttt{delMax()}) or the smallest item (\texttt{delMin()}).

• As we acquire new data, we assign it a priority (typically its value, but could be something calculated or defaulted).
  – Items with high/low priority will rise/fall in the data structure, so that they can be accessed quickly.
PQ Implementation #1: Unordered Array

• `insert()`: Simply add the new item to the end of the array.

• `delMax/Min()`: Find the largest (smallest) item in the array, swap it with what is at the end (single iteration of Selection Sort) and pop it off the end.

• **Insert cost**: $O(1)$

• **Remove cost**: $O(n)$
PQ Implementation #2: Ordered Array

- `insert()`: Add the new item to the end of the array, then push it to the correct location in the array (single iteration of Insertion Sort).
- `delMax/Min()`: Simply remove the item from the end of the list.

- **Insert cost**: $O(n)$
- **Remove cost**: $O(1)$
• Best of both worlds:
  – **Insert cost**: $O(\log(n))$
  – **Remove cost**: $O(\log(n))$

• A binary tree, where the key of each node is larger than or equal to the key of both of its children.
  – The largest key then will be located at the root.
Storing a Heap

• Represented sequentially in an array in **level order**, with the root at \( a[1] \), its children at \( a[2] \) and \( a[3] \), ...
  – The children of node \( k \) located at \( 2k \) and \( 2k+1 \).
  – The height of a tree with \( N \) items is \( \log(N) \)
  – Moving items is a called **reheapifying**
Heap Functions – `swim()`

```java
while (k > 1 && less(k/2, k)) {
    exch(k/2, k);
    k = k/2;
} //while

(Also called **Bottom-Up Reheapify**)  

private boolean less(int i, int j) {
    return pq[i].compareTo(pq[j]) < 0; }

private void exch(int i, int j) {
    Key t = pq[i]; pq[i]=pq[j]; pq[j]=t; }
```
Heap Functions – `swim()`

violates heap order (larger key than parent)
Heap Functions – \texttt{sink()}

\begin{verbatim}
while (2\times k \leq N) {
    int j = 2\times k;
    if (j < N \&\& \text{less}(j, j+1)) {
        j++;
    } //if
    if (!\text{less}(k, j)) {
        break;
    } //if
    \text{exch}(k, j);
    k = j;
} //while
\end{verbatim}

(Also called \textbf{Top-Down Reheapify})
Heap Functions – `sink()`

![Diagram of a heap with annotations indicating violations of heap order](image)

- `sink()` operation fails when a node is smaller than one of its children in a min-heap or larger in a max-heap.
Heaps and Priority Queues

- **insert()**: Add the new item to the end of the array, increment the size of the heap, and then **swim()** up through the heap to position the new item appropriately.

- **delMax()**: Remove the root of the heap, replace it with the item at the end of the array and decrement the size of the heap, then let that item **sink()** down through the heap to the appropriate position.
Building a Heap

1. Insert(P)
2. Insert(Q)
3. Insert(E)
4. RemoveMax()
5. Insert(X)
6. Insert(A)
7. Insert(M)
8. RemoveMax()
9. Insert(P)
10. Insert(L)
11. Insert(E)
12. RemoveMax()
Heapsort

```java
int N = a.length
for (int k = N/2; k>=1; k--) {
    sink(a, k, N);
} //for
while (N > 1) {
    exch(a, 1, N--);
    sink(a, 1, N);
} //while
```
Heapsort Visual

<table>
<thead>
<tr>
<th>N</th>
<th>k</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>initial values</strong></td>
<td></td>
<td>SORT</td>
<td>EXAMPLE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>heap-ordered</strong></td>
<td></td>
<td>TPS</td>
<td>SOL</td>
<td>LRAM</td>
<td>EEX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>sorted result</strong></td>
<td></td>
<td>AEE</td>
<td>ELM</td>
<td>OPR</td>
<td>STX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

02/03/2015

Priority Queues and Heapsort

Allegheny College
Theorem: Sink-based heap construction uses fewer than $2N$ compares and fewer than $N$ exchanges to construct a heap from $N$ items.

Proof: Most of the subheaps that we process are small. Consider a heap of 127 items:

- We process 32 heaps of size 3, 16 heaps of size 7, 8 heaps of size 15, 4 heaps of size 31, 2 heaps of size 63, and 1 heap of size 127.
- Multiply the number of heaps of each size by the height of each heap. $(32)(1) + (16)(2) + \ldots + (1)(6) = 120$ exchanges, and 240 compares.
Heapsort Evaluation

• **Theorem:** Heapsort uses fewer than $2n \times \log(n) + 2n$ compares (and half as many exchanges) to sort $n$ items.

• **Proof:** The $2n$ term covers the heap construction (previous slide). We run 1 sink operation per item in the array, and the max cost of each sink is $\log(n)$.

• Heapsort grows at a rate of $O(n \times \log(n))$ with respect to both compares and exchanges.
Any Questions?