13—Priority Queues
§9.1, §11.3

May 4, 2002

--- Frequency Assignment ---

--- The Text Engine ---

BST
AVL Tree
Hashtable(s)
Splay Tree
The Main Loop

```c
D = new DictAVL or DictHash or DictWhatever;
while ( still words left ) {
    w = next word;
}
D->PrintSorted(num_to_print);
```

Printing the List

```plaintext
my
and
you
a
in
of
the
of
the
1
521
466
466
my
383
you
970
the
708
and
666
of
32
to
521
1
466
a
466
in
466
my
383
you
```

- What if we want to print all \( n \) words?
- What if we want to print only the top ten words?

Priority Queues

- MakeEmpty(), isEmpty()
- Insert(key, info)
- FindMax(), DeleteMax()
How to Use

DictWhatever::PrintSorted(num_to_print)
{
    PQ pq;
    for (each record in dict)
        pq->Insert(record.freq, record.str);

    for (i = 1.. num_to_print)
        Print pq->DeleteMax();
}

Implementations of PQs

Linked List

• Insert

• FindMax

• DeleteMax

AVL Tree

• Insert

• FindMax

• DeleteMax
AVL Tree PQ

FindMax()

FindMin()

Removing the First $k$ Items

Running time for...

- Sorting and printing first $k$?
- Linked List PQ?
- AVL PQ?

<table>
<thead>
<tr>
<th>Hashtable</th>
<th>Top $k$ Freqs</th>
</tr>
</thead>
<tbody>
<tr>
<td>of</td>
<td>708</td>
</tr>
<tr>
<td>for</td>
<td>200</td>
</tr>
<tr>
<td>my</td>
<td>466</td>
</tr>
<tr>
<td>to</td>
<td>632</td>
</tr>
<tr>
<td>a</td>
<td>521</td>
</tr>
<tr>
<td>is</td>
<td>666</td>
</tr>
<tr>
<td>complete</td>
<td>279</td>
</tr>
<tr>
<td>the</td>
<td>970</td>
</tr>
<tr>
<td>nunnery</td>
<td>7</td>
</tr>
<tr>
<td>in</td>
<td>279</td>
</tr>
<tr>
<td>you</td>
<td>366</td>
</tr>
</tbody>
</table>

Detailed AVL PQ Creation

- $\approx n/2^h$ nodes inserted when tree is height $h$, for $h = 0, \ldots, \log n$
- The height 0 nodes take constant time to insert
- Time $h$ to insert node at height $h$, for $h \geq 1$
- Total time for rest is
  \[
  \sum_{h=1}^{\log n} h \cdot \frac{n}{2^h} = n \cdot \sum_{h=1}^{\log n} \frac{h}{2^h} 
  \leq n \cdot \left( \frac{1}{2} + 2 \cdot \frac{1}{4} + 3 \cdot \frac{1}{8} + \cdots \right) 
  = n \cdot \left( 1 + 2 \cdot \frac{1}{4} + 3 \cdot \frac{1}{8} + \cdots \right) 
  = n \cdot \left( 1 + 1 + \frac{1}{2} + \frac{1}{4} + \cdots \right) 
  = 2 \cdot n 
  \]
Towards an Easier Implementation

Partially Ordered Tree

- Parent *bigger* than children

A Hard Operation

FindMax()?

DeleteMax()? Insert()?

A Strange Operation

Decrease the Root
Let's Write Some Code

```cpp
DecreaseRoot(Node *n, int val) {
    n->key = val;
}
```

More Useful Than You Think

DeleteMin()
Deletion

Another way to do it

Why do we do it the other way?

Insertion

Fill out complete binary tree
Insertion

\[
\begin{array}{c}
  230 \\
  91 \quad 120 \\
  18 \quad 98 \quad 90 \quad 100 \\
  15 \quad 150
\end{array}
\begin{array}{c}
  230 \\
  91 \quad 120 \\
  15 \quad 150 \quad 91 \quad 120
\end{array}
\begin{array}{c}
  230 \\
  150 \quad 120 \\
  15 \quad 18 \quad 91
\end{array}
\]

Swap \textit{up}

Running Time?

Talking About Complete Trees

The space overhead of trees is annoying...

...but trees can be complicated, so it's necessary
Talking About Complete Trees

Complete trees are very regular

Do we really need all those pointers?

The Answer is No

Store as an array: *Heap* implementation of PQs

Adding and Removing

Heaps are easy!
**Indices of Heap in Array**

```
0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16
```

These are array indices, not key values

**Implementing Heaps**

```cpp
int Heap::LeftIndex(int i) {
    if (2 * i + 1 < array_size)
        return 2 * i + 1;
    return -1;
}

int Heap::RightIndex(int i) {
    if (2 * i + 2 < array_size)
        return 2 * i + 2;
    return -1;
}

void DecreaseRoot(Key new_root) {
    array[0] = new_root;
}
```
Our Application

```cpp
void PrintTopFreq(FreqHashTable& hash; int k)
{
    // Code...
}
```

In-Place Heap Creation

```
708 200 466 632 521 466 382 466 970 2 708 299 383
```

Heap Sort

1. Make heap, in-place

2. DeleteMax() by swapping root of heap to back of array

```
31 27 29 18 25 28 19 2 16 5
 5 27 29 18 25 28 19 2 16 31
```

3. Swap down to fix heap

```
5 27 29 18 25 28 19 2 16 31
29 27 5 18 25 28 19 2 16 31
29 27 28 18 25 5 19 2 16 31
```

4. Continue until sorted

```
Swap 29,16:
16 27 28 18 25 5 19 2 29 31
28 27 19 18 25 5 16 2 29 31
Swap 28,2:
2 27 19 18 2 5 16 28 29 31
27 25 19 18 2 5 16 28 29 31
```
Heap Sort Summary

- $O(n \log n)$ time, guaranteed

- In-place and efficient operations
  - No extra memory, only doing simple swaps

- What if initial array partially sorted?

- How is cache performance?