CSE 101 Class Notes

October 13, 2006

Today: Improving Iterative Algorithms

- Preprocessing
- Restructuring
- Data structures (i.e. heaps)

Example: Sorting

Aside: Why do we care about sorting? First, sorting is an important task in itself. Second, we have to sort under many different constraints (e.g. on-line, special input data, limited memory), which will suggest different algorithms. Finally, it is an important subpart of other algorithms.

Here is an abstract presentation of the sorting algorithm from last class:

```
A[1..n], B : array of number
S : set of numbers
S = { A[i] | 1 <= i <= n }
j <- 1
while S not empty
    B[j] <- min(S)
    j <- j + 1
    S <- S - { min(S) }
```

Here, min(S) is the smallest element of S. What structures are important? The (ordered) set `S` of ordered elements.

What operations?

- Find the minimum of `S`.
- Delete the minimum of `S`.
- Construct `S`.

Heaps

Definition: A *heap* is a binary tree such that a node n is less than all of its children (and, by implication, all of their children). Note that a heap is not totally ordered – while elements along a path from the root are strictly increasing, there is no relationship between elements in different branches. Heaps are typically implemented as arrays, where $H[i]$'s children are $H[2i]$ and $H[2i+1]$. Operations:

- Finding the minimum ($O(1)$): `H[1]`
- Inserting an element ($O(\log n)$):

```
insert(H, v)
    H[size(H) + 1] <- v
    size(H) <- size(H) + 1
    i <- size(H)
    while H[i] > H[i/2]         // percolate H[i] up to maintain
        swap(H[i], H[i/2])      // heap property
        i <- i / 2
```

- Removing the minimum ($O(\log n)$):

```
remove_max(H)
    H[1] <- H[size(H)]
    size(H) <- size(H) - 1
    i <- 1
    while H[i] > H[2*i] || H[i] > H[2*i + 1]
        swap(H[i], min(H[2*i], H[2*i + 1]))
        i <- min_index(H, 2*i, 2*i + 1)
```

Using a heap as our set representation in above sorting algorithm, lines 1 and 2 take $O(n)$; line 3, i.e. inserting n elements into a heap, is $O(n \log n)$; line 6 is $O(1)$; line 8 is $O(\log n)$. Therefore the entire algorithm takes $O(n \log n)$.

Skyline Problem

Input: A set of overlapping buildings. A building $b_i = (l_i, r_i, h_i)$ is a 3-tuple of left and right edges $l_i < r_i$ and a height h_i.

Output: a profile of the highest building at each position from $\min_i l_i$ to $\max_i r_i$, i.e. the list of heights at points at which the skyline height changes.

The straightforward algorithm is $O(n^2)$:

```
for i = 1 .. n
    insert (l[i], r[i]) into our skyline
```

Sweeping from left to right, however, we can do better. First, preprocess our data so that each left point l_i becomes a triple $(l_i, h_i, \text{“}start'')$, each right point a triple $(r_i, h_i, \text{“}end'')$. Sort the resulting set of triples.

Second, employ some data structure to keep track of the buildings that span each point along the horizon, $B_x = \{b_i, l_i \leq x \leq r_i\}$. The data structure needs to support the following operations:

(1) find $\max_i h_i \in B_x$.

(2) delete b_i with $r_i = x$.

(3) insert b_i with $l_i = x$.

This looks like a max-heap keyed on height, but how do we delete items other than the max in (2) and (3)? So we need to augment our basic data structure to support “named” elements, and operations to find and delete elements by name. One way to do this is with a supplemental array mapping names to heap indices. This array will have to be updated whenever the heap is adjusted, e.g. when inserting or removing items (note that this change will not change the order of heap operations). So the algorithm becomes:

```
1      S : array of labeled endpoints
2      A : list of skyline changes
3      H : named heap
4      sort S in increasing order
5      H <- { labeled endpoints of buildings in S }
6      for each unique coordinate x of S
7          for each building i with l[i] = x
8              insert(H, (h[i], i))
9          for each building i with r[i] = x
10             remove(H, (h[i], i))
11         insert(A, (max(H), x))
```

Running time: lines (4) and (5) each take $O(n \log n)$; lines (8) and (10) each take $O(\log n)$, and are executed no more than n times; line (11) also takes $O(\log n)$ and is executed $2n$ times. Therefore the total running time is $O(n \log n)$.