403: Algorithms and Data Structures

Heapsort and Priority Queues

Fall 2016
UAlbany
Computer Science

Some slides borrowed from David Luebke
Context

- We defined heaps
  - "almost" complete binary trees
  - $A[\text{Parent}(i)] \geq A[i]$ for all nodes $i > 1$

- Heap operations: Heapify()
  - Fix a single violation of the heap property
  - "Float" values down the tree
  - $O(\log n)$, where $n$ is the heap size

- What is the base of the log?

A = \begin{bmatrix} 16 & 14 & 10 & 8 & 7 & 9 & 3 & 2 & 4 & 1 \end{bmatrix}

\begin{tikzpicture}
  \node (16) at (0,0) {16};
  \node (14) at (-1,-1) {14};
  \node (10) at (-2,-2) {10};
  \node (8) at (-3,-3) {8};
  \node (7) at (-4,-4) {7};
  \node (9) at (-5,-5) {9};
  \node (3) at (-6,-6) {3};
  \node (2) at (-7,-7) {2};
  \node (4) at (-8,-8) {4};
  \node (1) at (-9,-9) {1};

  \draw (16) -- (14);
  \draw (14) -- (10);
  \draw (10) -- (8);
  \draw (10) -- (7);
  \draw (10) -- (9);
  \draw (10) -- (3);
  \draw (16) -- (2);
  \draw (16) -- (4);
  \draw (16) -- (1);
\end{tikzpicture}
Heap Operations: BuildHeap()

• Input: Array $A[1...n]$
• Required: Convert $A$ into a heap
• Idea: build a heap in a bottom-up manner by running $\text{Heapify}()$ on successive subarrays
Heap Operations: BuildHeap()

• Fact: for array of length $n$, all elements in range $A[\lceil n/2 \rceil + 1 .. n]$ are leaves ($\textit{Why?}$)
  – $\text{Left}(\lceil n/2 \rceil + 1) = 2 \times (\lceil n/2 \rceil + 1) > n$

• Another fact: Leaves are (trivially) heaps

• Walk backwards through the array from $\lceil n/2 \rceil$ to 1, calling $\texttt{Heapify}()$ on each node.
  – Order of processing guarantees that the children of node $i$ are heaps when $i$ is processed
  – Why is this important?
BuildHeap()

// given an unsorted array A, make A a heap
BuildHeap(A)
{
    heap_size(A) = length(A);
    for (i = ⌊length[A]/2⌋ downto 1)
        Heapify(A, i);
}

BuildHeap() Example

• Work through example
  A = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}
Crude Analysis of BuildHeap()

- Each call to `Heapify()` takes $O(lg n)$ time
- There are $O(n)$ such calls (specifically, $\lfloor n/2 \rfloor$)
- Thus the running time is $O(n \ lg n)$
  - Is this a correct asymptotic upper bound?
  - Is this an asymptotically tight bound?
- A tighter bound is $O(n)$
  - How can this be? Is there a flaw in the above reasoning?
Analyzing BuildHeap(): Tight

• To \texttt{Heapify} a subtree takes $O(h)$ time where $h$ is the height of the subtree
  – $h = O(\lg m)$, $m = \#$ nodes in subtree
  – Intuition: The height of most subtrees is small, i.e. $O(\log n)$ is too “generous”

• Fact 1: an $n$-element heap has at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes of height $h$
  – \textit{Proof?}

• Using Fact 1 we can show that \texttt{BuildHeap()} takes $O(n)$ time
  – \textit{Proof?}
Heapsort

• Given \texttt{BuildHeap()}, an in-place sorting algorithm is easily constructed:
  – Maximum element is at \(A[1]\)
  – Discard by swapping with element at \(A[n]\)
    • Decrement \texttt{heap\_size}[A]
    • \(A[n]\) now contains correct value
  – Restore heap property at \(A[1]\) by calling \texttt{Heapify()}
  – Repeat, always swapping \(A[1]\) for \(A[\texttt{heap\_size}(A)]\)
Example

```
    16
   / \   /   \
  14  10  8   9
 /    / /     / \
2    2  4  7   3
```
Heapsort

Heapsort(A)
{
    BuildHeap(A);
    for (i = length(A) downto 2)
    {
        Swap(A[1], A[i]);
        heap_size(A) -= 1;
        Heapify(A, 1);
    }
}
Analyzing Heapsort

• The call to \texttt{BuildHeap()} takes $O(n)$ time

• Each of the $n - 1$ calls to \texttt{Heapify()} takes $O(\lg n)$ time

• Thus the total time taken by \texttt{HeapSort()}
  \[= O(n) + (n - 1) \cdot O(\lg n)\]
  \[= O(n) + O(n \lg n)\]
  \[= O(n \lg n)\]
Priority Queues

• Heapsort is a nice algorithm, but in practice Quicksort (coming up) usually wins

• But the heap data structure is incredibly useful for implementing priority queues
  – A data structure for maintaining a set $S$ of elements, each with an associated value or key
  – Supports the operations $\text{Insert}()$, $\text{Maximum}()$, and $\text{ExtractMax}()$
  – What might a priority queue be useful for?
Assassin's prioritized TODO manager
Priority Queue Operations

- **Insert(S, x)** inserts the element x into set S
- **Maximum(S)** returns the element of S with the maximum key
- **ExtractMax(S)** removes and returns the element of S with the maximum key
- *How could we implement these operations using a heap?*
Priority Queue Operations

• **Insert(S, x)**
  – Increment heap size and add x at the end
  – move the new element “upwards” (reverse-heapify)
  – $O(\log n)$

• **Maximum(S)**
  – return $S[1]$
  – Time complexity?

• **ExtractMax(S)** removes and returns the element of S with the maximum key
  – Time?
## Heap vs All (for Priority queues)

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Pre-processing</th>
<th>Insert</th>
<th>Max</th>
<th>Extract Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linked List</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Sorted Array</td>
<td>$O(n \log n)$</td>
<td>$O(n)$ (shifting)</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Heap</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>
Announcements

• Read through Chapter 6
  – Next class: Build heap, Heap Sort, Priority Queues

• HW2 due next Wednesday