Last Time

• Trees
  – Lots of definitions
  – General tree structure
  – Binary Trees
    • Nodes, insertion, retrieval, deletion
  – Binary Search Trees
Priority Queue

• What if we don’t need everything sorted from smallest to biggest; what if we just want the smallest/biggest item in the array right now?

• Three operations necessary:
  – Acquire new data (\texttt{insert()})
  – Retrieve the largest/smallest item (\texttt{get()})
  – 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**: $O(1)$
Heap

• 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()` (Also called **Bottom-Up Reheapify**)
Heap Functions – \texttt{sink()} (Also called Top-Down Reheapify)

![Diagram of heap structure with sink function example]
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.

- **get()**: Look at the root of the heap.

- **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()
Comparing Data Structures

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Add</th>
<th>Get</th>
<th>Remove</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>O(1) (unless resize)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Linked List</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Stack</td>
<td>O(1)</td>
<td>O(1) (as long as we want what’s on top)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Queue</td>
<td>O(1)</td>
<td>O(1) (as long as we want the front)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Hashtable</td>
<td>O(1) (unless resize)</td>
<td>O(1)</td>
<td>O(1) (unless resize)</td>
</tr>
<tr>
<td>Tree</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Binary Search Tree</td>
<td>O(log(n))</td>
<td>O(log(n))</td>
<td>O(log(n))</td>
</tr>
<tr>
<td>Heap</td>
<td>O(log(n))</td>
<td>O(1) (as long as we want what’s on top)</td>
<td>O(log(n))</td>
</tr>
</tbody>
</table>
Any Questions?