CS 234

Module 8

November 15, 2018
ADT Priority Queue

Data: (key, element pairs) where

- keys are orderable but not necessarily distinct, and
- elements are any data.

Preconditions: For all P is a priority queue, k is a key, and e is an element; for LookUpMin and DeleteMin, P is not empty.

Postconditions: Mutation by Add (add item with key k) and DeleteMin (delete an item with minimum key k).

<table>
<thead>
<tr>
<th>Name</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create()</td>
<td>a new empty priority queue</td>
</tr>
<tr>
<td>IsEmpty(P)</td>
<td>true if empty, else false</td>
</tr>
<tr>
<td>LookUpMin(P)</td>
<td>item with minimum key k</td>
</tr>
<tr>
<td>Add(P, k, e)</td>
<td></td>
</tr>
<tr>
<td>DeleteMin(P)</td>
<td>item with minimum key k</td>
</tr>
</tbody>
</table>
Array implementations of priority queues

Unsorted array:
- store full then empty (plus variable to locate first empty)
- $O(n)$ LookUpMin
- $O(1)$ Add (put in first empty, update variable)
- $O(n)$ DeleteMin (look through all elements, swap with last full, update variable)

Unsorted array with a variable storing the location of the min:
- store full then empty (plus variable to locate first empty)
- $O(1)$ LookUpMin
- $O(1)$ Add (like above but also compare to min variable)
- $O(n)$ DeleteMin (modify constant but linear to update min variable)

Sorted array:
- Store index of first empty in variable.
- $O(1)$ LookUpMin
- $O(n)$ Add (find location by binary search or linear, shift all)
- $O(1)$ DeleteMin (end of array)
Linked implementations of priority queues

Unsorted linked list:

- store items in arbitrary order
- $O(n)$ LookUpMin
- $O(1)$ Add (put front of linked list)
- $O(n)$ DeleteMin (look through all elements to find smallest)

Unsorted linked list with a pointer to min:

- $O(1)$ LookUpMin
- $O(1)$ Add (put at beginning of list, compare to min, update if needed)
- $O(n)$ DeleteMin (modify constant but linear to update min variable)

Sorted linked list:

- $O(1)$ LookUpMin
- $O(n)$ Add (scan in linked list)
- $O(1)$ DeleteMin
An improved implementation

Goal:

- Maybe not as fast as $\Theta(1)$ for all operations
- Never as slow as $\Theta(n)$

Using a tree:

- Modify BST to allow duplicate keys (e.g. values in left subtree $\leq$ value at node).
- Where is minimum element?
- What is cost of Add in general?
- $\Theta(\log n)$ time for balanced tree

Keeping FindMin cheap

- Store smallest at root.
- Bound on height of tree.
- Come up with less stringent condition than binary search order in order to make the condition cheap to maintain.
Data structure: heap

A binary tree satisfies the *heap-order property* if for each node, the value stored at the node is no greater than that stored in either child (if any).

A *heap* is a complete binary tree that satisfies the heap-order property.

Using a heap to implement the ADT Dictionary: store (key, element) pairs at each node, ensuring that the keys satisfy the heap-order property.
Observations about example of previous slide

Consequences of heap-order property:
- Path from root to leaf in nondecreasing order (remember that equal values are possible.)
- Relative values of left and right children unknown.
- Where are biggest and smallest?

Consequences of complete:
- Nice array implementation
- Logarithmic bound on height
Customizing ADT Binary Tree to implement a heap

Additional data: Store both keys and elements at nodes.

Note: Only keys are ordered using the heap order property.

Modified operations:
  • AddNode(B, parent, key, element, side)

Additional operations (also for BST):
  • KeyAtNode(B, n)
  • ElementAtNode(B, n)
  • StoreInNode(B, n, key, element) - changes what is stored in a node

Additional operations (just for heap):
  • LastLeaf(B) - node that is the last leaf in complete tree
  • PreviousLeaf(B) - leaf before last leaf in complete tree
  • NextLeaf(B) - node that will be the next leaf in complete tree
  • SwapValues(B, node1, node2) - exchanges both key and element

Store a variable lastleaf in either implementation.
Implementing a heap using an array implementation of ADT Binary Tree

- Illustrations show keys only, not elements.
- lastleaf is a pointer for a linked implementation of ADT Binary Tree.
Implementing ADT Priority Queue using a heap

LookUpMin(P)
  - Return the data item in the root of tree.
  - $\Theta(1)$ array implementation of ADT Binary Tree
  - $\Theta(1)$ linked implementation of ADT Binary Tree

Add(P, k, e)
  - To preserve completeness, add at next leaf position.
  - NextLeaf can be found in time $\Theta(1)$ for the array implementation of ADT Binary Tree.
  - NextLeaf can be found in time $\Theta(\log n)$ for the linked implementation of ADT Binary Tree.
  - Problem: Heap-order property violated.

DeleteMin(P)
  - To preserve heap-order property, remove the root.
  - Problem: What remains is not a tree.
Implementing \text{Add}(P, k, e) \text{ in a heap}

“Bubble up”, fixing heap-order property on path from leaf to root by using Swap operation.

Here 18 has just been added.
Pseudocode for Add(P, k, e)

\[
\ell \leftarrow \text{NextLeaf}(B)
\]
\[
\text{lastleaf} \leftarrow \text{NextLeaf}(B)
\]
\[
\text{StoreInNode}(B, \ell, k, e)
\]
\[
curr \leftarrow \ell
\]
\[
\text{par} \leftarrow \text{Parent}(B, curr)
\]
while \(\text{par} \neq \text{false} \) and \(\text{KeyAtNode}(B, curr) < \text{KeyAtNode}(B, \text{par})\)
\[
\text{SwapValues}(B, curr, \text{par})
\]
\[
curr \leftarrow \text{par}
\]
\[
\text{par} \leftarrow \text{Parent}(B, curr)
\]
Observations about Add

Q: Why can we stop tracing up the path once we find a child that has a greater key value than its parent?

A: We verify that the heap property is satisfied everywhere. When we swap a child C with its parent P:

- C \preceq P
- P \preceq O (for O the other child)
- Thus C \preceq O (heap property OK at C)

Running time of Add:

- Number of iterations is height of heap in worst case.
- $\Theta(\log n)$ time due to height.

Various names:

- bubble-up
- sift
- percolate
Implementing \texttt{DeleteMin(P)} in a heap

Delete value in root, move value in last leaf to root.

“Bubble down”, fixing heap-order property on path from root to leaf using Swap operation
DeleteMin pseudocode

min ← Root(B)
SwapValues(B, Root(B), lastleaf)
DeleteNode(B, lastleaf)
lastleaf ← PreviousLeaf(B)
curr ← Root(B)
left ← LeftChild(B, curr)
right ← Rightchild(B, curr)
key ← KeyAtNode(curr)
stop ← false
if left ≠ false and right = false
   lkey ← KeyAtNode(B, left)
   if lkey < key
      SwapValues(B, curr, left)
      stop ← true
DeleteMin pseudocode, continued

while left \neq false and right \neq false and not stop)
   lkey ← KeyAtNode(B, left)
   rkey ← KeyAtNode(B, right)
   if key \leq lkey and key \leq rkey
      stop ← true
   else
      if lkey < rkey
         SwapValues(B, curr, left)
         curr ← left
      else
         SwapValues(B, curr, right)
         curr ← right
   left ← LeftChild(B, curr)
   right ← RightChild(B, curr)

return min
Observations about DeleteMin

PreviousLeaf:

- array-based implementation: $\Theta(1)$ time
- linked implementation: $\Theta(\log n)$ time

Running time of DeleteMin:

- Number of iterations is height of heap in worst case.
- $\Theta(\log n)$ time due to height.

Various names:

- bubble-down
- trickle
- percolate (again!)

Moral of story: careful about names

Why do we need to look at both children?

Did we need to look at both siblings in bubble-up?
Sorting using a priority queue

**User view**
Algorithm:
- Repeatedly use Add until all values entered.
- Repeatedly use DeleteMin.

Analysis:
- $\Theta(n \log n)$ for $n$ Add operations
- $\Theta(n \log n)$ for $n$ DeleteMin operations

**Provider view**
Can we find a faster way to make a heap out of $n$ elements?
Customizing ADT Priority Queue

Form heap out of a bunch of (key, element) pairs.
The **heapify** operation forms a heap out of an array of items.

Idea:

- Place all items into the structure.
- Fix heap-order property from bottom up.
- Observe that leaves are heaps of height 0.
- At phase $i$, form heaps of height at most $i$ from two heaps of height at most $i - 1$. 
Heapify example, phase 1
Heapify example, phase 2

```
7 1
11
2 5 3 9
6 4 10 8 12
```

```
2 1
11
4 5 3 9
6 7 10 8 12
```
Heapify example, phase 3

Initial heap:

```
2 1
11
4 5 3 9
6 7 10 8 12
```

After phase 3:

```
2 3
1
4 5 11 9
6 7 10 8 12
```
Linear-time heapify analysis

- Placement into structure takes $\Theta(n)$ time total.
- Logarithmic number of phases, in phase $i$ forming at most $n/2^{i+1}$ heaps of height $i$ each.
- Cost of making one heap of height $i$ is in $\Theta(i)$, bounded above by some $c_i$.
- Total cost of phases is at most $\sum_{i=1}^{\lfloor \log n \rfloor} c_i \cdot \frac{n}{2^{i+1}} \leq cn\left(\frac{1}{4} + \frac{2}{8} + \frac{3}{16} + \cdots\right)$, which is linear in $n$.
- Total cost in $O(n)$.