Priority Queues (Heaps)

October 11, 2016
Many applications require that we process records with keys in order, but not necessarily in full sorted order.

Often we collect a set of items and process the one with the current minimum value. e.g.
- Jobs sent to a printer,
- Operating system job scheduler in a multi-user environment.
- Simulation environments

An appropriate data structure is called a priority queue.
A priority queue is a data structure that supports two basic operations:
- insert a new item and
- remove the minimum item.
A simple linked list:
- Insertion at the front $O(1)$; deleteMin $O(N)$, or
- Keep list sorted; insertion $O(N)$, deleteMin $O(1)$

A binary search tree (BST):
- This gives an $O(\log N)$ average for both operations.
- But BST class supports many operations that are not required.

An array: Binary Heap
- Does not require links and will support both operations in $O(\log N)$ worst-case time.
The binary heap is the classic method used to implement priority queues.
We use the term heap to refer to the binary heap.
Heap is different from the term heap used in dynamic memory allocation.
Heap has two properties:
  - Structure property
  - Ordering property
A heap is a complete binary tree, represented as an array.

A complete binary tree is a tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right.
A complete binary tree of height $h$ has between $2^h$ and $2^{h+1} - 1$ nodes.

The height of a complete binary tree is $\lfloor \log N \rfloor$.

It can be implemented as an array such that:

- For any element in array position $i$:
  - the left child is in position $2i$,
  - the right child is in the cell after the left child ($2i + 1$), and
  - the parent is in position $\lfloor i/2 \rfloor$. 
A complete binary tree and its array representation
Heap-Order Property

- In a heap, for every node $X$ with parent $P$, the key in $P$ is smaller than or equal to the key in $X$.
- Thus the minimum element is always at the root.
  - Thus we get the extra operation `findMin` in constant time.
- A max heap supports access of the maximum element instead of the minimum, by changing the heap property slightly.
Two complete trees

(a) a heap; (b) not a heap
template <typename Comparable>
class BinaryHeap {

public:
    explicit BinaryHeap(int capacity = 100);
    explicit BinaryHeap(const vector<Comparable> & items);
    bool isEmpty() const;
    const Comparable & findMin() const;
    void insert(const Comparable & x);
    void insert(Comparable && x);
    void deleteMin();
    void deleteMin(Comparable & minItem);
    void makeEmpty();

private:
    int currentSize;  // Number of elements in heap
    vector<Comparable> array;  // The heap array
    void buildHeap();
    void percolateDown(int hole);
};
To insert an element $X$ into the heap:

- We create a hole in the next available location.
- If $X$ can be placed there without violating the heap property, then we do so and are done.
- Otherwise
  - we bubble up the hole toward the root by sliding the element in the hole’s parent down.
  - We continue this until $X$ can be placed in the hole.

This general strategy is known as a percolate up.
Attempt to insert 14, creating the hole and bubbling the hole up
The remaining two steps required to insert 14
/**
 * Insert item x, allowing duplicates.
 */

void insert( Comparable && x )
{
    if( currentSize == array.size() - 1 )
        array.resize( array.size() * 2 );

    // Percolate up
    int hole = ++currentSize;
    for( ; hole > 1 && x < array[ hole / 2 ]; hole /= 2 )
        array[ hole ] = std::move( array[ hole / 2 ] );
    array[ hole ] = std::move( x );
}
**Delete Minimum**

- `deleteMin` is handled in a similar manner as insertion:
  - Remove the minimum; a hole is created at the root.
  - The last element X must move somewhere in the heap.
    - If X can be placed in the hole then we are done.
    - Otherwise,
      - We slide the smaller of the hole’s children into the hole, thus pushing the hole one level down.
      - We repeat this until X can be placed in the hole.

- `deleteMin` is logarithmic in both the worst and average cases.
Creation of the hole at the root

Min = 13

13

14

16

19

68

19

65 26 32 31

16

14

19

31

21

65 26 32
The next two steps in the `deleteMin` operation
The last two steps in the `deleteMin` operation
// Remove the smallest item from the priority queue.
// Throw UnderflowException if empty.
void deleteMin( Comparable & minItem )
{
    if( isEmpty( ) )
        throw UnderflowException{ };

    minItem = std::move( array[ 1 ] );
    array[ 1 ] = std::move( array[ currentSize-- ] );
    percolateDown( 1 );
}
/**
 * Internal method to percolate down in the heap.
 * hole is the index at which the percolate begins.
 */

void percolateDown( int hole )
{
    int child;
    Comparable tmp = std::move( array[ hole ] );

    for( ; hole * 2 <= currentSize; hole = child )
    {
        child = hole * 2;
        if( child != currentSize && array[ child + 1 ] < array[ child ] )
            ++child;
        if( array[ child ] < tmp )
            array[ hole ] = std::move( array[ child ] );
        else
            break;
    }
    array[ hole ] = std::move( tmp );
}
Take as input $N$ items and place them into an empty heap.

Obviously this can be done with $N$ successive inserts: $O(N\log N)$ worst case.

However `buildHeap` operation can be done in linear time ($O(N)$) by applying a percolate down routine to nodes in reverse level order.
/**
 * Establish heap order property from an arbitrary
 * arrangement of items. Runs in linear time.
 */

void buildHeap( )
{
    for( int i = currentSize / 2; i > 0; --i )
        percolateDown( i );
}
Implementation of the linear-time `BuildHeap` method

Initial heap

After `percolateDown(7)`
(a) After \texttt{percolateDown}(6);  
(b) after \texttt{percolateDown}(5)
(a) After `percolateDown(4)`; (b) after `percolateDown(3)`
BuildHeap (Cont.)

(a) After `percolateDown(2)`;  
(b) after `percolateDown(1)` and `BuildHeap` terminates
The linear time bound of `BuildHeap`, can be shown by computing the sum of the heights of all the nodes in the heap, which is the maximum number of dashed lines.

For the perfect binary tree of height $H$ containing $N = 2^{H+1} - 1$ nodes, the sum of the heights of the nodes is $N - H - 1$.

Thus it is $O(N)$. 
Patients arrive at time $t$ with injury of criticality $C$

- If no patients are waiting and there is a free doctor, assign them to doctor and create a future departure event; else put patient in the Criticality priority queue.

Patient departs at time $t$

- If someone in Criticality queue, pull out most critical and assign to doctor; create a future departure event.