Priority Queues (Heaps)

CptS 223 – Advanced Data Structures

Larry Holder
School of Electrical Engineering and Computer Science
Washington State University
Motivation

- Queues are a standard mechanism for ordering tasks on a first-come, first-served basis.
- However, some tasks may be more important or timely than others (higher priority).
- **Priority queues**
  - Store tasks using a partial ordering based on priority.
  - Ensure highest priority task at head of queue.
- **Heaps** are the underlying data structure of priority queues.
Priority Queues

- Main operations
  - `insert` (i.e., enqueue)
  - `deleteMin` (i.e., dequeue)
    - Finds the minimum element in the queue, deletes it from the queue, and returns it

- Performance
  - Goal is for operations to be fast
  - Will be able to achieve $O(\log_2 N)$ time insert/deleteMin amortized over multiple operations
  - Will be able to achieve $O(1)$ time insert amortized over multiple insertions
Simple Implementations

- **Unordered list**
  - O(1) insert
  - O(N) deleteMin

- **Ordered list**
  - O(N) insert
  - O(1) deleteMin

- **Balanced BST**
  - O(\(\log_2 N\)) insert and deleteMin

- **Observation**: We don’t need to keep the priority queue completely ordered
A binary heap is a binary tree with two properties

Structure property

- A binary heap is a complete binary tree
  - Each level is completely filled
  - Bottom level may be partially filled from left to right

Height of a complete binary tree with $N$ elements is $\lfloor \log_2 N \rfloor$
Binary Heap Example
Binary Heap

- Heap-order property
  - For every node $X$, $\text{key}(\text{parent}(X)) \leq \text{key}(X)$
  - Except root node, which has no parent
- Thus, minimum key always at root
  - Or, maximum, if you choose
- Insert and deleteMin must maintain heap-order property
Implementing Complete Binary Trees as Arrays

- Given element at position i in the array
  - i’s left child is at position 2i
  - i’s right child is at position 2i+1
  - i’s parent is at position \( \lfloor i/2 \rfloor \)
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 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 );
};
Heap Insert

- Insert new element into the heap at the next available slot ("hole")
  - According to maintaining a complete binary tree
- Then, "percolate" the element up the heap while heap-order property not satisfied
Heap Insert: Example

Insert 14:
Heap Insert: Implementation

```cpp
/**
 * Insert item x, allowing duplicates.
 */

void insert( const 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 ] = array[ hole / 2 ];
    array[ hole ] = x;
}
```
Heap DeleteMin

- Minimum element is always at the root
- Heap decreases by one in size
- Move last element into hole at root
- Percolate down while heap-order property not satisfied
Heap DeleteMin: Example

14
Heap DeleteMin: Example
Heap DeleteMin: Example
Heap DeleteMin: Implementation

```c
/**
 * Remove the minimum item.
 * Throws UnderflowException if empty.
 */
void deleteMin()
{
    if( isEmpty() )
        throw UnderflowException();

    array[1] = array[ currentSize-- ];
    percolateDown( 1 );
}

/**
 * Remove the minimum item and place it in minItem.
 * Throws UnderflowException if empty.
 */
void deleteMin( Comparable & minItem )
{
    if( isEmpty() )
        throw UnderflowException();

    minItem = array[1];
    array[1] = array[ currentSize-- ];
    percolateDown( 1 );
}
```
Heap DeleteMin: Implementation

```java
/**
 * 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 = 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 ] = array[ child ];
        else
            break;
    }
    array[ hole ] = tmp;
}
```
Other Heap Operations

- **decreaseKey**(p, v)
  - Lowers value of item p to v
  - Need to percolate up
  - E.g., change job priority

- **increaseKey**(p, v)
  - Increases value of item p to v
  - Need to percolate down

- **remove**(p)
  - First, **decreaseKey**(p, -∞)
  - Then, **deleteMin**
  - E.g., terminate job
Building a Heap

- Construct heap from initial set of N items
- Solution 1
  - Perform N inserts
  - \(O(N)\) average case, but \(O(N \log_2 N)\) worst-case
- Solution 2
  - Assume initial set is a heap
  - Perform a percolate-down from each internal node (\(H[\text{size}/2]\) to \(H[1]\))
BuildHeap Example

Leaves are all valid heaps
BuildHeap Example
BuildHeap Example

```
150
  /  \\
80   40
  /  \\
20   10
  /  \\
100  60
     /    \
    30  90
```

```
150
  /  \\
80   40
  /  \\
20   10
  /  \\
50   110
    /    \
   140   130
```

```
BuildHeap Example
BuildHeap Implementation

```cpp
explicit BinaryHeap( const vector<Comparable> & items )
  : array( items.size( ) + 10 ), currentSize( items.size( ) )
{
  for( int i = 0; i < items.size( ); i++ )
    array[ i + 1 ] = items[ i ];
  buildHeap( );
}

/**
  * 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 );
}
BuildHeap Analysis

- Running time of buildHeap proportional to sum of the heights of the nodes

Theorem 6.1

- For the perfect binary tree of height $h$ containing $2^{h+1} - 1$ nodes, the sum of heights of the nodes is $2^{h+1} - 1 - (h + 1)$

- Since $N = 2^{h+1} - 1$, then sum of heights is $O(N)$

- Slightly better for complete binary tree
Binary Heap Operations
Worst-case Analysis

- Height of heap is $\lceil \log_2 N \rceil$
- insert: $O(\log_2 N)$
  - 2.607 comparisons on average, i.e., $O(1)$
- deleteMin: $O(\log_2 N)$
- decreaseKey: $O(\log_2 N)$
- increaseKey: $O(\log_2 N)$
- remove: $O(\log_2 N)$
- buildHeap: $O(N)$
Applications

- Operating system scheduling
  - Process jobs by priority
- Graph algorithms
  - Find the least-cost, neighboring vertex
- Event simulation
  - Instead of checking for events at each time click, look up next event to happen
Priority Queues: Alternatives to Binary Heaps

- **d-Heap**
  - Each node has d children
  - insert in $O(\log_d N)$ time
  - deleteMin in $O(d \log_d N)$ time

- Binary heaps are 2-Heaps
Mergeable Heaps

- Heap merge operation
  - Useful for many applications
  - Merge two (or more) heaps into one
  - Identify new minimum element
  - Maintain heap-order property
  - Merge in $O(\log N)$ time
  - Still support insert and deleteMin in $O(\log N)$ time
    - Insert = merge existing heap with one-element heap
- d-Heaps require $O(N)$ time to merge
Leftist Heaps

- Null path length \( \text{npl}(X) \) of node \( X \)
  - Length of the shortest path from \( X \) to a node without two children

- Leftist heap property
  - For every node \( X \) in heap, \( \text{npl}(\text{leftChild}(X)) \geq \text{npl}(\text{rightChild}(X)) \)

- Leftist heaps have deep left subtrees and shallow right subtrees
  - Thus if operations reside in right subtree, they will be faster
Leftist Heaps

npl(X) shown in nodes

Leftist heap

Not a leftist heap
Leftist Heaps

- Theorem 6.2
  - A leftist tree with \( r \) nodes on the right path must have at least \( 2^r - 1 \) nodes.
  
  Thus, a leftist tree with \( N \) nodes has a right path with at most \( \lfloor \log(N + 1) \rfloor \) nodes.
Leftist Heaps

- Merge heaps H1 and H2
  - Assume root(H1) > root(H2)
  - Recursively merge H1 with right subheap of H2
  - If result is not leftist, then swap the left and right subheaps
  - Running time $O(\log N)$

- DeleteMin
  - Delete root and merge children
Leftist Heaps: Example

$H_1$

$H_2$
Leftist Heaps: Example

Merge H2 (larger root) with right sub-heap of H1 (smaller root).
Leftist Heaps: Example

Attach previous heap as H1’s right child. Leftist heap?
Leftist Heaps: Example

Swap root’s children to make leftist heap.
Skew Heaps

- Self-adjusting version of leftist heap
- Skew heaps are to leftist heaps as splay trees are to AVL trees
- Skew merge same as leftist merge, except we always swap left and right subheaps
- No need to maintain or test NPL of nodes
- Worst case is $O(N)$
- Amortized cost of $M$ operations is $O(M \log N)$
Binomial Queues

- Support all three operations in $O(\log N)$ worst-case time per operation
- Insertions take $O(1)$ average-case time
- Key idea
  - Keep a collection of heap-ordered trees to postpone merging
A binomial queue is a forest of binomial trees
  - Each in heap order
  - Each of a different height

A binomial tree $B_k$ of height $k$ consists of two $B_{k-1}$ binomial trees
  - The root of one $B_{k-1}$ tree is the child of the root of the other $B_{k-1}$ tree
Binomial Trees

$B_0 \quad B_1 \quad B_2 \quad B_3$

$B_4$
Binomial Trees

- Binomial trees of height $k$ have exactly $2^k$ nodes.
- Number of nodes at depth $d$ is $\binom{k}{d}$, the binomial coefficient.
- A priority queue of any size can be represented by a binomial queue.
  - Binary representation of $B_k$

$H_1$:  
```
16
  18
12
  21
  24
  65
```
Binomial Queue Operations

- Minimum element found by checking roots of all trees
  - At most \((\log_2 N)\) of them, thus \(O(\log N)\)
  - Or, \(O(1)\) by maintaining pointer to minimum element
Binomial Queue Operations

- Merge (H1, H2) → H3
  - Add trees of H1 and H2 into H3 in increasing order by depth
  - Traverse H3
    - If find two consecutive $B_k$ trees, then create a $B_{k+1}$ tree
    - If three consecutive $B_k$ trees, then leave first, combine last two
    - Never more than three consecutive $B_k$ trees
- Keep binomial trees ordered by height
- $\min(H3) = \min(\min(H1), \min(H2))$
- Running time $O(\log N)$
Merge Example

$H_1$:  
16
  18

12
  21
  24
  65

$H_2$:  
13
  14
  26

23
  51
  24
  65

$H_3$:  
13

23
  51
  24
  65

12
  21
  24
  65
  65
  26
  16
  18
Binomial Queue Operations

- Insert \((x, H1)\)
  - Create single-element queue \(H2\)
  - Merge \((H1,H2)\)
- Running time proportional to minimum \(k\) such that \(B_k\) not in heap
- \(O(\log N)\) worst case
- Probability \(B_k\) not present is 0.5
  - Thus, likely to find empty \(B_k\) after two tries on average
  - \(O(1)\) average case
Binomial Queue Operations

- deleteMin (H1)
  - Remove min(H1) tree from H1
  - Create heap H2 from the children of min(H)
  - Merge (H1,H2)
- Running time $O(\log N)$
deleteMin Example

$H_3: 13$

Before deleteMin:

```
        23
       /  \
      51   24
       \   /  \
        65 24
```

After deleteMin:

```
        23
       /  \
      51   24
       \   /  \
        65 24
```

```
        12
       /  \
      21   24
       \   /  \
        65 65
```

```
        14
       /  \
      26   16
       \   /  \
        18 18
```
Binomial Queue Implementation

- Array of binomial trees
- Trees use first-child, right-sibling representation
template <typename Comparable>
class BinomialQueue
{
    public:
        BinomialQueue();
        BinomialQueue( const Comparable & item );
        BinomialQueue( const BinomialQueue & rhs );
        ~BinomialQueue();

        bool isEmpty() const;
        const Comparable & findMin() const;

        void insert( const Comparable & x );
        void deleteMin();
        void deleteMin( Comparable & minItem );

        void makeEmpty();
        void merge( BinomialQueue & rhs );

        const BinomialQueue & operator= ( const BinomialQueue & rhs );
private:
    struct BinomialNode
    {
        Comparable element;
        BinomialNode *leftChild;
        BinomialNode *nextSibling;

        BinomialNode( const Comparable & theElement,
                       BinomialNode *lt, BinomialNode *rt )
            : element( theElement ), leftChild( lt ), nextSibling( rt ) { }
    }

    enum { DEFAULT_TREES = 1 };  

    int currentSize;             // Number of items in priority queue
    vector<BinomialNode *> theTrees;  // An array of tree roots

    int findMinIndex( ) const;
    int capacity( ) const;
    BinomialNode * combineTrees( BinomialNode *t1, BinomialNode *t2 );
    void makeEmpty( BinomialNode * & t );
    BinomialNode * clone( BinomialNode *t ) const;
};
/**
 * Return the result of merging equal-sized t1 and t2.
 */
BinomialNode * combineTrees( BinomialNode *t1, BinomialNode *t2 )
{
    if( t2->element < t1->element )
        return combineTrees( t2, t1 );
    t2->nextSibling = t1->leftChild;
    t1->leftChild = t2;
    return t1;
}
/**
 * Merge rhs into the priority queue.
 * rhs becomes empty. rhs must be different from this.
 */

void merge( BinomialQueue & rhs )
{
    if( this == &rhs )    // Avoid aliasing problems
        return;

    currentSize += rhs.currentSize;

    if( currentSize > capacity() )
    {
        int oldNumTrees = theTrees.size( );
        int newNumTrees = max( theTrees.size( ), rhs.theTrees.size( ) ) + 1;
        theTrees.resize( newNumTrees );
        for( int i = oldNumTrees; i < newNumTrees; i++ )
            theTrees[ i ] = NULL;
    }
}
BinomialNode *carry = NULL;
for( int i = 0, j = 1; j <= currentSize; i++, j *= 2 )
{
    BinomialNode *t1 = theTrees[ i ];
    BinomialNode *t2 = i < rhs.theTrees.size( ) ? rhs.theTrees[ i ] : NULL;
    int whichCase = t1 == NULL ? 0 : 1;
    whichCase += t2 == NULL ? 0 : 2;
    whichCase += carry == NULL ? 0 : 4;

    switch( whichCase )
    {
    case 0: /* No trees */
    case 1: /* Only this */
        break;
    case 2: /* Only rhs */
        theTrees[ i ] = t2;
        rhs.theTrees[ i ] = NULL;
        break;
    case 4: /* Only carry */
        theTrees[ i ] = carry;
        carry = NULL;
        break;
    }  
}
case 3: /* this and rhs */
    carry = combineTrees( t1, t2 );
    theTrees[ i ] = rhs.theTrees[ i ] = NULL;
    break;

case 5: /* this and carry */
    carry = combineTrees( t1, carry );
    theTrees[ i ] = NULL;
    break;

case 6: /* rhs and carry */
    carry = combineTrees( t2, carry );
    rhs.theTrees[ i ] = NULL;
    break;

case 7: /* All three */
    theTrees[ i ] = carry;
    carry = combineTrees( t1, t2 );
    rhs.theTrees[ i ] = NULL;
    break;

    }
    }

for( int k = 0; k < rhs.theTrees.size( ); k++ )
    rhs.theTrees[ k ] = NULL;
    rhs.currentSize = 0;

}
/*
 * Remove the minimum item and place it in minItem.
 * Throws UnderflowException if empty.
 */

void deleteMin( Comparable & minItem )
{
    if( isEmpty( ) )
        throw UnderflowException( );

    int minIndex = findMinIndex( );
    minItem = theTrees[ minIndex ]->element;
}
BinomialNode *oldRoot = theTrees[ minIndex ];
BinomialNode *deletedTree = oldRoot->leftChild;
delete oldRoot;

// Construct H''
BinomialQueue deletedQueue;
deletedQueue.theTrees.resize( minIndex + 1 );
deletedQueue.currentSize = ( 1 << minIndex ) - 1;
for( int j = minIndex - 1; j >= 0; j-- )
{
    deletedQueue.theTrees[ j ] = deletedTree;
    deletedTree = deletedTree->nextSibling;
    deletedQueue.theTrees[ j ]->nextSibling = NULL;
}

// Construct H'
theTrees[ minIndex ] = NULL;
currentSize -= deletedQueue.currentSize + 1;
merge( deletedQueue );
35       /**
36       * Find index of tree containing the smallest item in the priority queue.
37       * The priority queue must not be empty.
38       * Return the index of tree containing the smallest item.
39       */
40       int findMinIndex( ) const
41       {
42           int i;
43           int minIndex;
44
45           for( i = 0; theTrees[ i ] == NULL; i++ )
46               ;
47
48           for( minIndex = i; i < theTrees.size( ); i++ )
49               if( theTrees[ i ] != NULL &&
50                   theTrees[ i ]->element < theTrees[ minIndex ]->element )
51                   minIndex = i;
52
53           return minIndex;
54       }
Priority Queues in STL

- Binary heap
- Maintains maximum element
- Methods
  - Push, top, pop, empty, clear

```cpp
#include <iostream>
#include <queue>
using namespace std;

int main ()
{
    priority_queue<int> Q;
    for (int i=0; i<100; i++)
        Q.push(i);
    while (! Q.empty())
    {
        cout << Q.top() << endl;
        Q.pop();
    }
}
```
```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <functional>
#include <string>

using namespace std;

// Empty the priority queue and print its contents.
template<typename PriorityQueue>
void dumpContents( const string & msg, PriorityQueue & pq )
{
    cout << msg << ":" << endl;
    while( !pq.empty() )
    {
        cout << pq.top() << endl;
        pq.pop();
    }
}

// Do some inserts and removes (done in dumpContents).
int main()
{
    priority_queue<int> maxPQ;
    priority_queue<int,vector<int>,greater<int> > minPQ;
    minPQ.push( 4 ); minPQ.push( 3 ); minPQ.push( 5 );
    maxPQ.push( 4 ); maxPQ.push( 3 ); maxPQ.push( 5 );
    dumpContents( "minPQ", minPQ ); // 3 4 5
    dumpContents( "maxPQ", maxPQ ); // 5 4 3
    return 0;
}
Summary

- Priority queues maintain the minimum or maximum element of a set
- Support $O(\log N)$ operations worst-case
  - insert, deleteMin, merge
- Support $O(1)$ insertions average case
- Many applications in support of other algorithms