CMSC 341
Lecture 18

## Announcements

Expect graded exams on Monday

Truncated office hours on Monday (until 1:30)

## Priority Queues

Priority: some property of an object that allows it to be prioritized WRT other objects (of the same type)

Priority Queue: homogeneous collection of Comparables with the following operations (duplicates are allowed)

- `void insert (const Comparable &x)`
- `void deleteMin()`
- `void deleteMin(Comparable &x) const`
  `Comparable &findMin() const`
- Construct from set of initial values
- `bool isEmpty() const;`
- `bool isFull() const;`
- `void makeEmpty();`

## Priority Queue Applications

Printer management: the shorter document on the printer queue, the higher its priority.

Jobs queue: users' tasks are given priorities. System priority high.

Simulations

Sorting

## Possible Implementations

Use sorted list. Sort by priority upon insertion.

- findMin() --> Itr.retrieve()
- insert() --> list.insert()
- deleteMin() --> list.delete(1)

Use ordinary BST

- findMin() --> tree.findMin()
- insert() --> tree.insert()
- deleteMin() --> tree.delete(tree.findMin())

Use balanced BST

- guaranteed O(lg n) for AVL, Red-Black

## Binary Heap

A binary heap is a CBT with the further property that at every vertex neither child is smaller than the vertex, called *partial ordering*.

Every path from the root to a leaf visits vertices in a non-decreasing order.

## Binary Heap Properties

For a node at index i

- its left child is at index 2i
- its right child is at index 2i+1
- its parent is at index $\lfloor i/2 \rfloor$

No pointer storage

Fast computation of 2i and $\lfloor i/2 \rfloor$

i << 1 = 2i

i >> 1 = $\lfloor i/2 \rfloor$

## Binary Heap Performance

Performance

- construction O(n)
- findMin O(1)
- insert O(lg n)
- deleteMin O(log n)

Heap efficiency results, in part, from the implementation

- conceptually a binary tree
- implementation in an array (in level order), root at index 1

## BinaryHeap.H

```
template <class Comparable>
class Binary Heap {
public:
   explicit BinaryHeap(int capacity = BIG);
   bool isEmpty() const;
   bool isFull() const;
   const Comparable & findMin() const;
   void insert (const Comparable & x);
   void deleteMin();
   void deleteMin(Comparable & min_item);
   void makeEmpty();
private:
   int currentSize;
   vector<Comparable> array;
   void buildHeap();
   void percolateDown(int hole);
   };
```

## BinaryHeap.C

```
template <class Comparable>
const Comparable & BinaryHeap::findMin() {
   if (isEmpty()) throw Underflow();
   return array[1];
   }
```

## Insert Operation

Must maintain

- CBT property (heap shape):
  - easy, just insert new element at the right of the array
- Heap order
  - could be wrong after insertion if new element is smaller than its ancestors
  - continuously swap the new element with its parent until parent is not greater than it
    - called *sift up* or *percolate up*

Performance O(lg n) worst case because height of CBT is O(lg n)

## BinaryHeap.C (cont)

```
template <class Comparable>
void BinaryHeap<Comparable>::insert(const Comparable &
  x) {
  if (isFull()) throw OverFlow();
  int hole = ++currentSize;
  // percolate up
  for (; hole > 1 && x < array[hole/2]; hole /= 2)
      array[hole] = array[hole/2];
  // put x in hole
  array[hole] = x;
  }
```

## Deletion Operation

Steps
- remove min element (the root)
- maintain heap shape
- maintain heap order

To maintain heap shape, actual vertex removed is last one
- replace root value with value from last vertex and delete last vertex
- sift-down the new root value
  - continually exchange value with the smaller child until no child is smaller

## BinaryHeap.C (cont)

```
template <class Comparable>
void BinaryHeap<Comparable>::deleteMin(Comparable
  &minItem) {
  if (isEmpty()) throw Underflow();
  minItem = array[1];
  array[1] = array[currentSize-];
  percolateDown(1);
  }
```

## BinaryHeap.C (cont)

```
template <class Comparable>
void BinaryHeap<Comparable>::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;
  }
```