Priority Queues (Heaps)
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: Specification

- Main operations
  - **insert** (i.e., enqueue)
    - Dynamic insert
    - specification of a priority level (0-high, 1,2.. Low)
  - **deleteMin** (i.e., dequeue)
    - Finds the current minimum element (read: “highest priority”) in the queue, deletes it from the queue, and returns it

- Performance goal is for operations to be “fast”
Using priority queues

- insert(): Adds elements to the priority queue.
- deleteMin(): Dequeues the next element with the highest priority.

Dequeues the next element with the highest priority.
Can we build a data structure better suited to store and retrieve priorities?

Simple Implementations

- **Unordered linked list**
  - O(1) insert
  - O(n) deleteMin

- **Ordered linked list**
  - O(n) insert
  - O(1) deleteMin

- **Ordered array**
  - O(lg n + n) insert
  - O(n) deleteMin

- **Balanced BST**
  - O(log₂ n) insert and deleteMin
Binary Heap

A priority queue data structure
Binary Heap

- A binary heap is a binary tree with two properties
  - Structure property
  - Heap-order property
Structure Property

- A binary heap is a complete binary tree
  - Each level (except possibly the bottom most level) is completely filled
  - The bottom most level may be partially filled (from left to right)

- Height of a complete binary tree with \( N \) elements is \([\log_2 N]\)
**Binary Heap Example**

N=10

Every level (except last) saturated

Array representation:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

Cpt S 223. School of EECS, WSU
Heap-order Property

- Heap-order property (for a “MinHeap”)
  - For every node $X$, $\text{key(parent}(X)) \leq \text{key}(X)$
  - Except root node, which has no parent
- Thus, minimum key always at root
  - Alternatively, for a “MaxHeap”, always keep the maximum key at the root
- Insert and deleteMin must maintain heap-order property
Heap Order Property

- Duplicates are allowed
- No order implied for elements which do not share ancestor-descendant relationship
Implementing Complete Binary Trees as Arrays

- Given element at position i in the array
  - Left child(i) = at position 2i
  - Right child(i) = at position 2i + 1
  - Parent(i) = at position \( \left\lfloor \frac{i}{2} \right\rfloor \)
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 );
};

Just finds the Min without deleting it
insert
deleteMin
Note: a general delete() function is not as important for heaps but could be implemented
Stores the heap as a vector
Fix heap after deleteMin
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
Percolating Up

Heap Insert: Example

Insert 14:

13
  / \
21   16
 /   / \
24 31 19 68
 /    /    / \
65 26 32 14 hole
Percolating Up

Heap Insert: Example

Insert 14:

(1) 14 vs. 31

hole
Heap Insert: Example

Insert 14:

1. 14 vs. 31
   - Insert 14:
     - 13
     - 21
     - 24
     - 65
     - 26
     - 32
     - 31
     - 14
     - hole

2. 14 vs. 21
   - 14
   - 16
   - 21
   - 24
   - 65
   - 26
   - 32
   - 31
   - 19
   - 68

Cpt S 223. School of EECS, WSU
Percolating Up

Heap Insert: Example

Insert 14:

(1) 14 vs. 31

(2) 14 vs. 21

(3) 14 vs. 13

Path of percolation up

Heap order prop
Structure prop
Heap Insert: Implementation

// assume array implementation
void insert( const Comparable &x) {
    //
}

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;
}
```

O(log N) time
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

Percolating down...

Make this position empty
Heap DeleteMin: Example

- Copy 31 temporarily here and move it down.

- Make this position empty.

- Is 31 > min(14, 16)?
  - Yes - swap 31 with min(14, 16)
Is 31 > min(19,21)?
• Yes - swap 31 with min(19,21)
Heap DeleteMin: Example

Is 31 > min(19,21)?
- Yes - swap 31 with min(19,21)

Is 31 > min(65,26)?
- Yes - swap 31 with min(65,26)

Percolating down…

Cpt S 223. School of EECS, WSU
Heap DeleteMin: Example

Percolating down…
Heap DeleteMin: Example

Percolating down...

Heap order prop
Structure prop
Heap DeleteMin: Implementation

```
/**
 * 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);
}
```

O(log N) time
Heap DeleteMin: Implementation

```c
/**
 * 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 the current value of item \(p\) to new priority value \(v\)
  - Need to **percolate up**
  - E.g., promote a job

- **increaseKey**(\(p, v\))
  - Increases the current value of item \(p\) to new priority value \(v\)
  - Need to **percolate down**
  - E.g., demote a job

- **remove**(\(p\))
  - First, decreaseKey\((p, -\infty)\)
  - Then, deleteMin
  - E.g., abort/cancel a job

Run-times for all three functions?

\(O(\lg n)\)
Improving Heap Insert Time

- What if all N elements are all available upfront?

- To build a heap with N elements:
  - Default method takes $O(N \log N)$ time
  - We will now see a new method called buildHeap() that will take $O(N)$ time - i.e., optimal
Building a Heap

- Construct heap from initial set of N items

**Solution 1**
- Perform N inserts
- $O(N \log_2 N)$ worst-case

**Solution 2 (use buildHeap())**
- Randomly populate initial heap with structure property
- Perform a percolate-down from each internal node (H[size/2] to H[1])
  - To take care of heap order property
BuildHeap Example

Input: { 150, 80, 40, 10, 70, 110, 30, 120, 140, 60, 50, 130, 100, 20, 90 }

Leaves are all valid heaps (implicitly)

- Arbitrarily assign elements to heap nodes
- Structure property satisfied
- Heap order property violated
- Leaves are all valid heaps (implicitly)

So, let us look at each internal node, from bottom to top, and fix if necessary

Cpt S 223. School of EECS, WSU
BuildHeap Example

- Randomly initialized heap
- Structure property satisfied
- Heap order property violated
- Leaves are all valid heaps (implicit)

Swap with left child
BuildHeap Example

Dotted lines show path of percolating down

Swap with right child

Nothing to do
BuildHeap Example

Dotted lines show path of percolating down

Swap with right child & then with 60

Nothing to do

Cpt S 223. School of EECS, WSU
BuildHeap Example

Dotted lines show path of percolating down

Final Heap
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 );
}
```

Start with lowest, rightmost internal node
BuildHeap() : Run-time Analysis

- Run-time = ?
  - \( O(\text{sum of the heights of all the internal nodes}) \)
    because we may have to percolate all the way down to fix every internal node in the worst-case

- **Theorem 6.1** HOW?
  - *For a perfect binary tree of height \( h \), the sum of heights of all nodes is \( 2^{h+1} - 1 - (h + 1) \)*
  - Since \( h = \log N \), then sum of heights is \( O(N) \)
  - Will be slightly better in practice

**Implication:** Each insertion costs \( O(1) \) amortized time
Build Heap analysis

Claim: Runtime = O(sum of heights of all internal nodes) = O(sum of heights of all nodes) incl. leaves

Proof:

- Sum of heights of all nodes = sum of heights of nodes at each level
  - $2^0 \quad 2^1 \quad 2^2 \quad 2^3 \quad 2^n = (2^n - 1) = \sum_{i=0}^{n-1} 2^i$

- $\sum$ heights of all nodes

Let $S = \sum_{i=0}^{n-1} 2^i$

- $S = 1 + 2^1 + 2^2 + 2^3 + \cdots + 2^{n-1}$

- $2S = 2 + 2^2 + 2^3 + \cdots + 2^n$

- $2S - S = 2^n - 1$

- $S = 2^n - 1$

As $h = \log_2 n$,

- $\sum$ heights = $2^n - \log_2 n + 2 = O(n)$
Binary Heap Operations
Worst-case Analysis

- Height of heap is \( \lceil \log_2 N \rceil \)
- insert: \( O(\lg N) \) for each insert
  - In practice, expect less
- buildHeap insert: \( O(N) \) for \( N \) inserts
- deleteMin: \( O(\lg N) \)
- decreaseKey: \( O(\lg N) \)
- increaseKey: \( O(\lg N) \)
- remove: \( O(\lg N) \)
Applications

- Operating system scheduling
  - Process jobs by priority
- Graph algorithms
  - Find shortest path
- Event simulation
  - Instead of checking for events at each time click, look up next event to happen
An Application: The Selection Problem

- Given a list of $n$ elements, find the $k^{th}$ smallest element

Algorithm 1:
- Sort the list $\Rightarrow O(n \log n)$
- Pick the $k^{th}$ element $\Rightarrow O(1)$

A better algorithm:
- Use a binary heap (minheap)
Selection using a MinHeap

- **Input:** n elements
- **Algorithm:**
  1. buildHeap(n) ⇒ O(n)
  2. Perform k deleteMin() operations ⇒ O(k log n)
  3. Report the k\(^{th}\) deleteMin output ⇒ O(1)

Total run-time = O(n + k log n)

If k = O(n/log n) then the run-time becomes O(n)
Other Types of Heaps

- Binomial Heaps

- d-Heaps
  - Generalization of binary heaps (ie., 2-Heaps)

- Leftist Heaps
  - Supports merging of two heaps in $o(m+n)$ time (ie., sub-linear)

- Skew Heaps
  - $O(\log n)$ amortized run-time

- Fibonacci Heaps
## Run-time Per Operation

<table>
<thead>
<tr>
<th></th>
<th>Insert</th>
<th>DeleteMin</th>
<th>Merge (=H₁+H₂)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Binary heap</strong></td>
<td>O(log n) worst-case</td>
<td>O(log n)</td>
<td>O(n)</td>
</tr>
<tr>
<td></td>
<td>O(1) amortized for buildHeap</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Leftist Heap</strong></td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td><strong>Skew Heap</strong></td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td><strong>Binomial Heap</strong></td>
<td>O(log n) worst case</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td></td>
<td>O(1) amortized for sequence of n inserts</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Fibonacci Heap</strong></td>
<td>O(1)</td>
<td>O(log n)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>
Priority Queues in STL

- Uses **Binary heap**
- Default is MaxHeap
- Methods
  - Push, top, pop, empty, clear

#include <priority_queue>
int main ()
{
    priority_queue<int> Q;
    Q.push (10);
    cout << Q.top () ;
    Q.pop () ;
}

Calls DeleteMax()

For MinHeap: declare priority_queue as:

    priority_queue<int, vector<int>, greater<int>> Q;

Refer to Book Chapter 6, Fig 6.57 for an example

Cpt S 223. School of EECS, WSU
Binomial Heaps
Binomial Heap

- A binomial heap is a *forest* of heap-ordered binomial trees, satisfying:
  i) Structure property, and
  ii) Heap order property

- A binomial heap is different from binary heap in that:
  - Its structure property is totally different
  - Its heap-order property (within each binomial tree) is the same as in a binary heap
Note: A binomial tree need not be a binary tree

Definition: A “Binomial Tree” $B_k$

- A binomial tree of height $k$ is called $B_k$:
  - It has $2^k$ nodes
  - The number of nodes at depth $d = \binom{k}{d}$

$\binom{k}{d}$ is the form of the co-efficients in binomial theorem

<table>
<thead>
<tr>
<th>Depth</th>
<th>$B_3$:</th>
<th>#nodes:</th>
</tr>
</thead>
<tbody>
<tr>
<td>d=0</td>
<td></td>
<td>$\binom{3}{0}$</td>
</tr>
<tr>
<td>d=1</td>
<td></td>
<td>$\binom{3}{1}$</td>
</tr>
<tr>
<td>d=2</td>
<td></td>
<td>$\binom{3}{2}$</td>
</tr>
<tr>
<td>d=3</td>
<td></td>
<td>$\binom{3}{3}$</td>
</tr>
</tbody>
</table>

Cpt S 223. School of EECS, WSU
What will a Binomial Heap with $n=31$ nodes look like?

- We know that:
  - i) A binomial heap should be a forest of binomial trees
  - ii) Each binomial tree has power of 2 elements
- So how many binomial trees do we need?

$$n = 31 = (11111)_2$$

$B_4B_3B_2B_1B_0$
A Binomial Heap w/ n=31 nodes

$B_i = B_{i-1} + B_{i-1}$

$n = 31 = (1 1 1 1 1)_2$
Binomial Heap Property

- **Lemma:** There exists a binomial heap for every positive value of \( n \)

- **Proof:**
  - All values of \( n \) can be represented in binary representation
  - Have one binomial tree for each power of two with co-efficient of 1
  - Eg., \( n=10 \implies (1010)_2 \implies \text{forest contains } \{B_3, B_1\} \)
Binomial Heaps: *Heap-Order Property*

- Each binomial tree should contain the minimum element at the root of every subtree
  - Just like binary heap, except that the tree here is a binomial tree structure (and not a complete binary tree)

- The order of elements across binomial trees is irrelevant
Definition: Binomial Heaps

- A binomial heap of $n$ nodes is:
  - (Structure Property) A forest of binomial trees as dictated by the binary representation of $n$
  - (Heap-Order Property) Each binomial tree is a min-heap or a max-heap
Binomial Heaps: Examples

Two different heaps:

$H_1$:  
16  
18

$H_2$:  
13
14
26

Cpt S 223. School of EECS, WSU
Key Properties

- Could there be multiple trees of the same height in a binomial heap?  
  
  no

- What is the upper bound on the number of binomial trees in a binomial heap of $n$ nodes? 
  
  $\lceil \log n \rceil$

- Given $n$, can we tell (for sure) if $B_k$ exists? 
  
  $B_k$ exists if and only if: 
  the $k^{th}$ least significant bit is 1 
  in the binary representation of $n$
An Implementation of a Binomial Heap

Example: \( n=13 \) == \((1101)_2\)

- \( B_7 \)
- \( B_6 \)
- \( B_5 \)
- \( B_4 \)
- \( B_3 \)
- \( B_2 \)
- \( B_1 \)
- \( B_0 \)

Maintain a linked list of tree pointers (for the forest)

Shown using the left-child, right-sibling pointer method

Analogous to a bit-based representation of a binary number \( n \)
Binomial Heap: Operations

- $x \leq \text{DeleteMin}()$
- Insert(x)
- Merge($H_1, H_2$)
DeleteMin()

- **Goal:** Given a binomial heap, H, find the minimum and delete it

- **Observation:** The root of each binomial tree in H contains its minimum element

- **Approach:** Therefore, return the minimum of all the roots (minimums)

- **Complexity:** $O(\log n)$ comparisons (because there are only $O(\log n)$ trees)
FindMin() & DeleteMin() Example

For DeleteMin(): After delete, how to adjust the heap?

New Heap: Merge \{B_0, B_2\} & \{B_0', B_1', B_2'\}
**Insert($x$) in Binomial Heap**

- **Goal:** To insert a new element $x$ into a binomial heap $H$

- **Observation:**
  - Element $x$ can be viewed as a single element binomial heap
  - $\Rightarrow$ Insert ($H, x$) $\Rightarrow$ Merge($H$, {$x$})

So, if we decide how to do merge we will automatically figure out how to implement both insert() and deleteMin()
Merge($H_1, H_2$)

- Let $n_1$ be the number of nodes in $H_1$
- Let $n_2$ be the number of nodes in $H_2$
- Therefore, the new heap is going to have $n_1 + n_2$ nodes
  - Assume $n = n_1 + n_2$

**Logic:**
- Merge trees of same height, starting from lowest height trees
- If only one tree of a given height, then just copy that
- Otherwise, need to do carryover (just like adding two binary numbers)
Idea: merge tree of same heights

Merge: Example

\[
\begin{align*}
H_1: & \quad B_0, B_1, \text{copy} \\
H_2: & \quad B_2
\end{align*}
\]
How to Merge Two Binomial Trees of the *Same* Height?

Simply compare the roots

```cpp
/**
 * 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;
}
```

Note: Merge is defined for only binomial trees with the same height.
Merge($H_1, H_2$) example

$H_1$:  

+  

$H_2$:  

16  18  

13  14  26  

12  21  24  65  

23  51  24  65  

14  26  16  26  

?  

Cpt S 223, School of EECS, WSU
How to Merge *more than two* binomial trees of the same height?

- Merging more than 2 binomial trees of the same height could generate carry-overs

![Diagram showing the merging of three binomial trees](image)

Merge any two and leave the third as carry-over
Merge\((H_1, H_2)\) : Example

Input:

\[ H_1: \]
\begin{align*}
16 & \quad 18 \\
12 & \quad 21 & \quad 24 & \quad 65
\end{align*}

\[ H_2: \]
\begin{align*}
13 & \quad 14 & \quad 23 & \quad 51 & \quad 24 & \quad 65
\end{align*}

Output:

\[ H_3: \]
\begin{align*}
13 & \quad 23 & \quad 51 & \quad 24 & \quad 65
\end{align*}

\[ 12 & \quad 21 & \quad 24 & \quad 14 & \quad 26 & \quad 16 & \quad 18
\]

There are two other possible answers

\[ \text{Merge cost } \leq \log(\max\{n_1, n_2\}) = O(\log n) \text{ comparisons} \]
Run-time Complexities

- Merge takes $O(\log n)$ comparisons

- **Corollary:**
  - Insert and DeleteMin also take $O(\log n)$

- It can be further proved that an uninterrupted sequence of $m$ Insert operations takes only $O(m)$ time per operation, implying $O(1)$ amortized time per insert

  - **Proof Hint:**
    - For each insertion, if $i$ is the least significant bit position with a 0, then number of comparisons required to do the next insert is $i+1$
    - If you count the #bit flips for each insert, going from insert of the first element to the insert of the last ($n^{th}$) element, then
      $\Rightarrow$ amortized run-time of $O(1)$ per insert
Binomial Queue Run-time Summary

- **Insert**
  - $O(\lg n)$ worst-case
  - $O(1)$ amortized time if insertion is done in an uninterrupted sequence (i.e., without being intervened by deleteMins)

- **DeleteMin, FindMin**
  - $O(\lg n)$ worst-case

- **Merge**
  - $O(\lg n)$ worst-case
## Run-time Per Operation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Insert</th>
<th>DeleteMin</th>
<th>Merge (=H₁+H₂)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Binary heap</strong></td>
<td>O(log n) worst-case</td>
<td>O(log n)</td>
<td>O(n)</td>
</tr>
<tr>
<td></td>
<td>O(1) amortized for buildHeap</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Leftist Heap</strong></td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td><strong>Skew Heap</strong></td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td><strong>Binomial Heap</strong></td>
<td>O(log n) worst case</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td></td>
<td>O(1) amortized for sequence of n inserts</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Fibonacci Heap</strong></td>
<td>O(1)</td>
<td>O(log n)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>
Summary

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