Topic 6: Concatenated Queues

Read: Chpt. 6, Weiss

Q: How can we effectively merge (concatenate) two priority queues Q1 and Q2 with size \( n \) and \( m \) together, \( n \geq m \)?

Approach 1:
We can merge Q1 and Q2 together by inserting the objects in Q2 into Q1.

Previous data structures ineffective since
BST: \( T_w(n,m) = O(nm) \).
Heap: \( T_w(n,m) = O(mlgn) \).
Minmax Heap: \( T_w(n,m) = O(mlgn) \).

Approach 2:
We can rebuild the data structure using objects from both Q1 and Q2.
BST: \( T_w(n,m) = O((n+m)\lg(m+n)) \), if distinct.
Heap: \( T_w(n,m) = O(n+m) \).
Minmax Heap: \( T_w(n,m) = O(n+m) \).

Conclusion:
Better off rebuilding and then restructuring the data structure, if \( m = \Theta(n) \), when we need to merge!
Q: Can we do it better?

ADT: Concatenated (Mergeable) queue.
   A collection class, whose items have all been assigned a priority, with the following operations:
   1. \textit{insert}(x,Q)
   2. \textit{findMin}(Q)
   3. \textit{deleteMin}(Q)
   4. \textit{concat}(Q_1,Q_2)
   5. create(Q)
   6. destroy(Q)
   7. isEmpty(Q)
   8. size(Q)

Remark: A concatenated queue is a priority queue that supports \textit{concat} operation effectively.

Q: How does one design this kind of concatenated queues?

Let’s consider a class of concatenated queues called \textit{mergeable/concatenated heaps} formed by using heap-ordered trees. If we have a function \textit{concat}(H_1,H_2) that allows us to merge the two concatenated heaps together and then stored the resulting heap as H_1, then we can perform the insert and delete operations as follow:
**insert(x,H):**
Treat x as a concatenated heap by itself and merge it with H.

**deleteMin(H):**
Delete min from H so as to decompose H – \{x\} into a collection of concatenated heaps and then merge them together.

1. **Leftist heap** (C.A. Crane, 1972):
A leftist heap is a heap-ordered leftist tree.

**Dfn:** Given a binary tree T. For each node x in T, define the rank of x as follow:
\[
\text{rank}(x) = \text{length of a shortest path going from } x \text{ to an external node in } T_E, \text{ where } T_E \text{ is the extended binary tree of } T.
\]

**Remark:** If y is an external node in \( T_E \), we can define \( \text{rank}(y) = 0 \).

**Dfn:** A leftist tree T is a binary tree such that

1. \( T = \emptyset \), or
2. For every node \( x \in T \), we have
   \[
   \text{rank}(\text{left_child}(x)) \geq \text{rank}(\text{right_child}(x)).
   \]
Examples: Binary trees with ranks.

T1:

```
   A 2
  /   \
 B 2   C 1
/ \
D 1 E 1 F 1
```

T1 is not a leftist tree.

T2:

```
   A 2
  /   \
 B 1   C 1
/ \
D 1 E 1
F 1
```

T2 is a leftist tree.
A **leftist heap** satisfies the following two properties:

- **Structural property:** A leftist tree
- **Relational property:** Heap-ordered tree

### Examples of leftist heaps:

**Min leftist heap:**

```
       2
      /|
     7 25
    / |
   11 32
  /   |
13    10
      /|
     11 9
```

**Max leftist heap:**

```
       18
      /|
     12 10
    /  |
   9   8
  /  |
2   5
```

**A skewed min leftist heap:**

```
       2
      /|
     4 6
    / |
   8 10
```
Remarks:
(1) A leftist tree may not be a balanced binary tree.
(2) If a leftist heap operation depends on the height of the tree, we will have $T_w(n) = O(n)$ complexity, which offers no gain in performance.
(3) Since a leftist tree is “left-heavy”, one should always avoid using the left subtree (path) of a node.
(4) No path from the root to an external node is shorter than the path that always uses the right child in going from the root to an external node.
(5) Leftist tree operations will always operate on the right subtree of a node in $T$.

Theorem: Let $x$ be the root of a leftist tree with $n$ nodes. We have $n \geq 2^{\text{rank}(x)} - 1$.

Corollary: A leftist tree with $n$ nodes has a right path going from the root to an external node containing at most $\lceil \lg(n+1) \rceil$ nodes.

Conclusion: $T_w(n) = O(\lg n)$ for leftist heap operations.
Implementation:
Array implementation infeasible, use pointers!

Node:

<table>
<thead>
<tr>
<th>lchild</th>
<th>rank</th>
<th>key</th>
<th>rchild</th>
</tr>
</thead>
</table>

Example: A min leftist heap.
Recall that if we know how to merge/concate, we will know how to insert and deletemin.

**Leftist heap operations:**
1. **insert(x,H):**
   Observe that a single element by itself is a leftist heap. Hence, we can perform concate(H1,H), where H1 is the leftist heap containing the single element x.

2. **deletemin(H):**
   After executing the deleteMin operation, we have left with two leftist heaps H1 & H2.

   \[
   \begin{align*}
   &x \\
   &y & z \\
   \Downarrow & \ \\
   &H \\
   \\
   &y & H1 \\
   &z & H2 \\
   \end{align*}
   \]

   Now, perform concate(H1,H2).

**Q:** How do we merge two leftist heaps together?
3. **concatenate**(H1,H2):

Recall that we will always operate on the right side of a leftist heap.

Assuming that we are using min leftist heap and the roots of H1 and H2 are x and y respectively such that \( x \leq y \).

**Algorithm:**

\[
\text{if } H1 \text{ or } H2 = \emptyset \\
\quad \text{then return the other heap;}
\]

\[
\text{else compare } x \text{ with } y;
\]

\[
\quad \text{if } x > y \\
\quad \quad \text{then } \text{swap}(H1,H2)
\]

\[
\text{endif;}
\]

\[
\text{merge } H2 \text{ with the } \text{RIGHT} \text{ subheap of } x;
\]

\[
\text{attach the resulting tree as the right child of } x;
\]

\[
\text{compute rank}(x);
\]

\[
\text{if } (\text{rank}(\text{lchild-of-x}) < \text{rank}(\text{rchild-of-x}))
\]

\[
\quad \text{then } \text{swap}(\text{lchild-of-x},\text{rchild-of-x})
\]

\[
\text{endif}
\]

**Q:** Do we always obtain a heap-ordered tree after the merging of H2 with the **right** subheap of H1?

Yes.
Q: Do we always obtain a leftist tree?
   Not necessarily!

Remedy:
   One must check the ranks of the two children of x after each re-attachment and swap the two subtrees of x whenever leftist heap property is not satisfied.

Merging two lefties heaps:
Merge(Node *H1, Node *H2)
   if H1 = null
      then return H2;
   else if H2 = null
      then return H1;
   else if (key(H1) > key(H2))
      then swap(H1,H2)
      endif;
   H1->rchild := Merge(H1->rchild, H2);
   adjust rank(H1);
   if rank(H1->lchild) < rank(H1->rchild)
      then swap(H1->lchild, H1->rchild)
      endif;
   return H1;
   endif;
end Merge;
Example: Merging two min leftist heaps H1 & H2.

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Example: Merging two min leftist heaps H1 & H2.

```

```
Final min leftist heap:

```
  2
 /  \
3   5
 /   /
4   8 7
 /  /
6   5  9
    /
    10
```

Complexity:
- `concat, insert, deleteMin: T_w(n) = O(lgn)`,
- `findMin: T_w(n) = O(1)`,
- `build_heap: T_w(n) = O(nlgn)`, using `insert` op.

**Q**: Can we omit the rank info so as to simplify the `concat` operation? How do we perform `concat(H1,H2)` if `H1` and `H2` are just two heap-ordered trees with no additional structural property imposed on them?
2. **Skew Heap** (Sleator & Tarjan, 1986):
   A heap-ordered binary tree with
   **Structural property:** A binary tree,
   **Relational property:** Heap-ordered tree.

Consider `concat(H1,H2)`: Merge two min skew heaps H1 and H2 together, the roots of H1 and H2 are x and y respectively with $x \leq y$.

**Observation:**
Since we do not have the rank info, if we merge H1 and H2 as in leftist heap, it may result in long right path!

**Possible Remedy:**
Swap `left_child` with `right_child` after *every* merge operation.
Merging two skew heaps:
Assuming that we are using min leftist heap and the roots of H1 and H2 are x and y respectively such that x < y.

Algorithm:
Merge(Node *H1, Node *H2)
   if H1 = null
      then return H2;
   else if H2 = null
      then return H1;
   else if (key(H1) > key(H2))
      then swap(H1,H2)
      endif;
   *Node temp := H1→rchild;
   H1→rchild := H1→lchild;
   H1→lchild := Merge(temp, H2);
   return H1;
   endif;
endif;
end Merge;
Example:

```
   2     4     2
  /     /     / \
 3 8   9 5   3 6
 /      \
 6     7 \\
```

```
  4     8     4
 /      /     / \
 9 5   9   9 6
 /      \
 7 \\
```

```
  5     8     5
 /      /     / \
 7     8   8 7 \\
```

```
  5     8     5 \
 /      /     / \
 7     8   8 7 \\
```

```
  5     9     2 \
 /      /     / \
 8   7   4 3 \\
```

```
  5     9     2 \
 /      /     / \
 8   7   4 3 \\
```

```
  5     9     2 \
 /      /     / \
 8   7   4 3 \\
```
```
Complexity:
\[ T_w(n) = O(n). \]

Amortized Cost:
For \( m \) insert, deletemin, or concate operations,
\[ T_w(m,n) = O(mlgn). \]

Amortized complexity per operation:
\[ T_w(n) = O(lgn). \] (Same as leftist heap)

Remark:
Skew heap is a class of self-adjusting concatenate queue. It uses less memory and is more efficient than leftist heap.

Comparing leftist heap with skew heap:

<table>
<thead>
<tr>
<th></th>
<th>Leftist Heap</th>
<th>Skew Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree structure</td>
<td>Leftist tree</td>
<td>Binary tree</td>
</tr>
<tr>
<td>Implementation</td>
<td>Children pointers</td>
<td>Children pointers</td>
</tr>
<tr>
<td>Node structure</td>
<td>Need info on rank</td>
<td>No info on rank</td>
</tr>
<tr>
<td>Merging</td>
<td>Verify rank info</td>
<td>No rank info</td>
</tr>
<tr>
<td>Re-attachment</td>
<td>May swap</td>
<td>Always swap</td>
</tr>
</tbody>
</table>
3. **Pairing Heap** (Fredman, Sedgewick, Sleator, Tarjan, 1986):

**Q:** Given two (min) heap-ordered trees H1 and H2 with roots x and y respectively, $x \leq y$, what is the simplest way to merge them together?

**Simplest Approach in Merging Two Heap-Ordered Trees:**
Assuming that $x \leq y$, make y a child of x.

![Diagram of merging two pairing heaps](image)

**Example:** Merging two pairing heaps.

![Example diagram](image)

**Q:** How can we incorporate this simple merge concept into the design of a concatenate queue?
Pairing Heap:
   Structural Property: Multiway tree.
   Relational Property: Heap-ordered tree.

Observations: Given a set of n objects to be represented using a pairing heap H.
   1. Maximum degree of a node (root) in H = n−1.

(Min) Pairing Heap Operations:
   1. Merge(H1,H2): Assume that root of H1 ≤ root of H2.
      Make H2 the new first child of H1.
   2. Insert(x,H):
      Let x be a single-node heap and merge it with H.
   3. DeleteMin(H):
      Delete the root to decompose H into ≤ n−1 heaps and then merge them together.

Worst-Case Time Complexity:
   If merging two heaps can be done in O(1) time, then insert can be done in O(1) time and deleteMin can be done in O(n) time.
**Implementation:**
Left-child right-sibling representation.

**Example:**

![Diagram of a Pairing Heap]

**Remark:** Using this left-child right-sibling representation, merge(H1,H2) can be performed in $T_w(n) = O(1)$ time as discussed above.

**Q:** How do we merge the subtrees together?

**Merging Subtrees in a Pairing Heap:**
1. Merging subtrees randomly:
   Merge subtrees two at a time in any order.
2. Two-Pass Method:

*First pass:*
Merge pairs of subtrees from left to right. If the number of subtrees is odd, merge the last remaining subtrees with the last newly merged subtree.

\[
\begin{align*}
T_1 & \quad T_2 & \quad T_3 & \quad T_4 & \quad \ldots & \quad T_{n-2} & \quad T_{n-1} & \quad T_n \\
\quad & \quad S_1 & \quad S_2 & \quad & \quad & \quad & \quad & \quad & \quad & S_{n/2} \\
\quad & \quad & \quad & \quad & \quad & \quad & \quad & \quad & \quad & S^*_{n/2}
\end{align*}
\]

*Second pass:*
Starting from the last (rightmost) tree obtained above, from right to left and one at a time, merge it with the remaining trees to form a single tree.

\[
\begin{align*}
S_1 & \quad \ldots & \quad S_{k-3} & \quad S_{k-2} & \quad S_{k-1} & \quad S_k \\
\quad & \quad \quad & \quad \quad & \quad \quad & \quad \quad & \quad \quad & \quad \quad & \quad \quad & \quad H_1 \\
\quad & \quad \quad & \quad \quad & \quad \quad & \quad \quad & \quad \quad & \quad \quad & \quad \quad & \quad H_2 \\
\quad & \quad \quad & \quad \quad & \quad \quad & \quad \quad & \quad \quad & \quad \quad & \quad \quad & \quad H_3 \\
\quad & \quad \quad & \quad \quad & \quad \quad & \quad \quad & \quad \quad & \quad \quad & \quad \quad & \quad \ldots \\
\quad & \quad \quad & \quad \quad & \quad \quad & \quad \quad & \quad \quad & \quad \quad & \quad \quad & \quad H_{k-1}
\end{align*}
\]
3. Multi-Pass method:
create a FIFO queue Q and store all trees in Q;
while Q ≠ ∅ do
    R ← deque(Q);
    if Q ≠ ∅
        then R1 ← deque(Q);
            enqueue(merge(R,R1));
    endif;
endwhile;
return R;

Example: Consider merging 7 trees.

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
<th>T6</th>
<th>T7</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>T3</th>
<th>T4</th>
<th>T5</th>
<th>T6</th>
<th>T7</th>
<th>R1</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>T5</th>
<th>T6</th>
<th>T7</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>T7</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>R2</th>
<th>R3</th>
<th>S1</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>S1</th>
<th>S2</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>H1</th>
</tr>
</thead>
</table>

Return R = H1.
Complexity on Merging Trees:
   Same for all three methods asymptotically with
   \( T_w(n) = O(n) \) in merging \( n \) subtrees.

Complexity on Building an Initial Heap:
Two BuildHeap Operations:
   1. Insert each element in \( S \) into an initially empty
      pairing heap using above merge operation.
   2. Consider each element being a heap with a single
      element and merge the \( n \) heaps together to form a
      single heap.

   \( T_w(n) = O(n) \).

Amortized Complexity of Pairing Heap Operations:
   Insert, deleteMin, merge:
   \( T(n) = O(\lg n) \). (amortized)
4. Binomial Queue
Let’s first introduce the following class of binomial trees (BT):

**A recursive definition of BT of order \( k \geq 0 \):**
- A single node is a BT of order 0, \( B_0 \).
- A BT of order \( k \), \( k > 0 \), \( B_k \) is formed by “melding” two BTs of order \( k-1 \) together by making the root of one tree be the child of the root of the other tree.

**Observation:** On expanding, \( B_k \) can also be formed by making the roots of \( B_0, B_1, \ldots, B_{k-2}, B_{k-2} \) be the children of a new root.
**Example:** A BT $B_4$ of order 4.

![Diagram of a binomial tree]

**Observations:**

1. The root of $B_k$ has exactly $k$ children.
2. $B_k$ is of height $k$.

**Q:** Why the term “binomial” queue?

Recall that for given integers $n$ and $k$, $n \geq k \geq 0$, the binomial coefficient $C(n,k)$ is defined by:

$$C(n,k) = \binom{n}{k} = \frac{n!}{k!(n-k)!}.$$ 

If one counts the number of nodes at level $i$, $0 \leq i \leq k$, of a BT of order $k$, $B_k$, it is exactly equal to the binomial coefficient $C(k,i)$. Hence, the total number of nodes in $B_k$ is given by

$$\sum_{i=0}^{k} C(k,i) = \binom{k}{0} + \binom{k}{1} + \ldots + \binom{k}{k} = 2^k.$$
Theorems:
(1) The root of $B_k$ has exactly $k$ children.
(2) The height of $B_k$ is $k$.
(3) The binomial tree $B_k$ has exactly $2^k$ nodes.
(4) The number of nodes in $B_k$ at level (depth) $i$ is given by the binomial coefficient $C(k,i)$.

Example: The binomial tree $B_4$ illustrating the above properties.

BT Properties of $B_4$:
(1) The root of $B_4$ has exactly 4 children.
(2) $B_4$ has height 4.
(3) There are $2^4 = 16$ nodes in $B_4$.
(4) The number of nodes at each level of $B_4$ is given by the following binomial coefficients.

<table>
<thead>
<tr>
<th>Level</th>
<th># Nodes</th>
<th>Binomial Coeff.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>$C(4,0)$</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>$C(4,1)$</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>$C(4,2)$</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>$C(4,3)$</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>$C(4,4)$</td>
</tr>
</tbody>
</table>

Q: Given a set $S$ of $n$ records, can we always store the $n$ objects in a BT of order $k$?
No, unless $n = 2^k$ for some integer $k$. 
Remedy:
Use a collection of binomial trees with total number of nodes equal to n.

Binomial Queue/Heap:
A binomial queue (BQ) is a collection of uniquely specified heap-ordered binomial trees $Q = \{B_i \mid i \in I\}$ such that if $B_j$ and $B_k$ are two binomial trees in $Q$, then $i \neq j$.

Q: Using one node per record, how do we represent a set of $n$ records $S = \{x_1, x_2, \ldots, x_n\}$ using a BQ?

Given a positive integer $n$, $0 \leq n \leq 2^k$.
1. Integer $n$ can always be represented using a binary representation $(b_k, b_{k-1}, \ldots, b_1, b_0)$ with at most $k+1$ bits.
2. Since $n = 2^0 + 2^1 + \ldots + 2^k$, the i-bit, $0 \leq i \leq k$, contributes at most $2^i$ to $n$.
3. Since a BT of order $k$ contains exactly $2^k$ nodes and can be used to represent $2^k$ objects in $S$, each binary digit $b_i = 1$ in $(b_k \ldots b_1 b_0)_2$ corresponds to a unique BT $B_i$ in $Q$.
4. Given $n = (b_k \ldots b_1 b_0)_2$, $Q = \{B_i \mid i \in I\}$ can be represented using a collection of BT such that $B_i \in Q$ iff $b_i = 1$ in $(b_k \ldots b_1 b_0)_2$. We call $(b_k \ldots b_1 b_0)_2$ the binary representation of $Q$. 
**Theorem:** There are at most $\lceil \lg(n+1) \rceil$ BTs in $Q$ when representing a set of $n$ records.

**Example:** Given $S = \{3, 8, 6, 4, 2, 7, 9, 16, 1, 4, 10\}$.

Since $n = 11_{10} = 1011_2$, $Q = \{B_3, B_1, B_0\}$.

**Example of a BQ for $S$:**

```
10  1  2
  
  4
```

**BQ Operations:**

Let’s assume that we already have a function `concate(Q1, Q2)` that allows us to merge/concate two BQ’s $Q1$ and $Q2$.

1. **Insert($x, Q$):**

Since $x$ can be considered as a BQ with a BT $B_0$, $\text{insert}(x, Q) = \text{concate}(Q1, Q2)$, where $Q1$ is the BT containing $B_0$ with one element $x$. 
2. **Deletemin(Q):**

Find the min element \( m \) among the roots of BTs in \( Q \). If \( m \) is the root of a BT \( B_i \), form new BQ \( Q_1 = Q - \{ B_i \} \). Also, remove \( m \) from \( B_i \) to form a new BQ \( Q_2 \) by including all remaining BTs in \( B_i \) together. Perform \( \text{concat}(Q_1,Q_2) \).

**Q:** How do we perform \( \text{concat}(Q_1,Q_2) \)?

3. **Concatenate(Q1, Q2):**

Let’s first consider the merging of two BTs \( B_k \) of the same order.

**Simplest approach:**

Compare the roots of the two BTs and then make the root of the BT with smaller root the parent of the other BT.

Hence, \( T(n) = O(1) \).

**Q:** How can we extend this method to \( \text{concat}(Q_1,Q_2) \)?

Observe that if \( (b_p...b_1b_0)_2 \) and \( (c_p...c_1c_0)_2 \) are the binary representation of \( |Q_1| \) and \( |Q_2| \), then \( \text{concat}(Q_1,Q_2) \) results in a BQ \( Q_3 \) such that \( |Q_3| \) has binary representation \( (d_{p+1}d_p...d_1d_0)_2 \), where \( (d_{p+1}d_p...d_1d_0)_2 = (b_p...b_1b_0)_2 +(c_p...c_1c_0)_2 \).
Example:
Q1:
\[
\begin{array}{ccc}
10 & 1 & 2 \\
\downarrow & & \downarrow \\
4 & & \end{array}
\]
Q2:
\[
\begin{array}{ccc}
24 & 12 & 18 \\
\downarrow & & \downarrow \\
65 & 21 & 16 \\
\downarrow & & \downarrow \\
21 & 16 & 18 \\
\downarrow & & \\
3 & & \\
\end{array}
\]
Concate(Q1,Q2):
\[
\begin{array}{ccc}
10 & 1 & 2 \\
\downarrow & & \downarrow \\
4 & 24 & 12 \\
\downarrow & & \downarrow \downarrow \\
65 & 21 & 16 \\
\downarrow & & \downarrow \downarrow \\
18 & 16 & 8 \\
\downarrow & & \downarrow \\
3 & & 4 \\
\downarrow & & \downarrow \\
& & 6 \\
\end{array}
\]
Complexity for BQ Operations:
\[T_w(n) = O(\log n).\]
**Building a BQ**

Insert the given elements one by one into an initially empty BQ. Merge two BTs $B_i$ of order $i$ to form a BT $B_{i+1}$ of order $i+1$ if exist. Tie can be broken arbitrarily. Hence, $T_w(n) = O(n \log n)$.

**Example:** Building a BQ for $S = \{3, 8, 6, 4, 2, 7, 9, 16, 1, 4, 10\}$.

3 → 3 → 6 3 → 3 → 2 3

8

3

8

8

4

3

8

4

6

3

9

2

3

7

8

4

6

2

7

9

3

16

8

4

6
Implementation of BQ:
Node Structure:

<table>
<thead>
<tr>
<th>order</th>
<th>key</th>
<th>item</th>
</tr>
</thead>
<tbody>
<tr>
<td>l_sib</td>
<td>f_child</td>
<td>r_sib</td>
</tr>
</tbody>
</table>

A BT is implemented using the leftmostChild-rightSibling representation with circular doubly linked list.

For any node x in a BT:
- order: Order of BT root at x.
- f_child: pointer pointing at the lowest order child of x.
- r_child: pointer pointing at the right sibling of x.
- l_child: pointer pointing at the left sibling of x.

Observe that, with this structure, siblings of x are linked together from lower to higher order. A BQ Q
is then implemented by linking all binomial trees in Q together in *increasing order of their orders*.

**Example of Binomial Tree Implementation:**

$$B_2: \quad \begin{array}{c}
12 \\
21 \quad 16 \\
18
\end{array}$$

**Data Structure for B$_2$:**

```
B2
  12
  2
  21
  0
  16
  1
  18
  0
```
Example of Binomial Queue Implementation:

Q:

\[
\begin{array}{c}
24 \\
65 \\
\end{array} \quad \begin{array}{c}
12 \\
21 \\
16 \\
18 \\
\end{array}
\]

Data Structure for Q:

Warning: The BQ pointer Q is pointing at the root of the lowest-ordered BT, not a BT with minimum element.
**Node Implementation:**

```c
struct bqNode
{
    bqItemType: item;
    int key;
    int order;
    bqNode *f_child, *l_sibling, *r_sibling;
}
```

Consider the merging of two binomial trees T1 and T2 of order k, k ≥ 0:
Assuming that T1→key ≤ T2→key:
   if k = 0, then T1→f_child = T2;
      T1→order = T1→order + 1;

**Example:**

```
12 0
   ↓
21 0
```

```
12 1
   ↓
12 0
```

```
21 0
   ↓
21 0
```
if $k > 0$, then
(1) $T_2 \rightarrow l\_sibling = T_1 \rightarrow f\_child \rightarrow l\_sibling$;
(2) $T_2 \rightarrow l\_sibling \rightarrow r\_sibling = T_2$;
(3) $T_1 \rightarrow f\_child \rightarrow l\_sibling = T_2$;
(4) $T_1 \rightarrow order = T_1 \rightarrow order + 1$;

Example:
HW: Design and analysis an algorithm to merge two binomial trees $B_k$ of order $k$ together. Implement $\text{concat}(Q1, Q2)$ to merge two BQs.

10/15/2014