Chapter 9 Priority Queues

- Single- and Double-Ended Priority Queues
- **Leftist Trees**
- Binomial Heaps
- Fibonacci Heaps
- Pairing Heaps
- **Min-Max Heaps**
- Interval Heaps

Single- and Double-Ended Priority Queues

A priority queue is a collection of elements such that each element has an associated priority.

For single-ended priority queues (SEPQ), the operations supported by a min priority queue are:

- SP1: Return an element with minimum priority. 
  `minElement(); minKey();`
- SP2: Insert an element with an arbitrary priority.
  `insert();`
- SP3: Delete an element with minimum priority.
  `deleteMin();`

- Other ADTs: `size(); isEmpty();`
### Illustration of Priority Queue

<table>
<thead>
<tr>
<th>Operation</th>
<th>output</th>
<th>Priority Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert(5,A)</td>
<td>-</td>
<td>{(5,A)}</td>
</tr>
<tr>
<td>insert(9,C)</td>
<td>-</td>
<td>{(5,A),(9,C)}</td>
</tr>
<tr>
<td>insert(3,B)</td>
<td>-</td>
<td>{(3,B),(5,A),(9,C)}</td>
</tr>
<tr>
<td>insert(7,D)</td>
<td>-</td>
<td>{(3,B),(5,A),(7,D),(9,C)}</td>
</tr>
<tr>
<td>minElement()</td>
<td>B</td>
<td>{(3,B),(5,A),(7,D),(9,C)}</td>
</tr>
<tr>
<td>minKey()</td>
<td>3</td>
<td>{(3,B),(5,A),(7,D),(9,C)}</td>
</tr>
<tr>
<td>removeMin()</td>
<td>-</td>
<td>{(5,A),(7,D),(9,C)}</td>
</tr>
<tr>
<td>size()</td>
<td>3</td>
<td>{(5,A),(7,D),(9,C)}</td>
</tr>
<tr>
<td>minElement()</td>
<td>A</td>
<td>{(5,A),(7,D),(9,C)}</td>
</tr>
<tr>
<td>deleteMin()</td>
<td>-</td>
<td>{(7,D),(9,C)}</td>
</tr>
<tr>
<td>deleteMin()</td>
<td>-</td>
<td>{}</td>
</tr>
<tr>
<td>deleteMin()</td>
<td>“error”</td>
<td>{}</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>true</td>
<td>{}</td>
</tr>
</tbody>
</table>

### Heaps Applied for Priority Queue

- **Max heap tree**: a tree in which the key value in each node is no smaller than the key values in its children. A *max heap* is a complete binary tree that is also a *max tree*.

- **Min heap tree**: a tree in which the key value in each node is no larger than the key values in its children. A *min heap* is a complete binary tree that is also a *min tree*.
Double-Ended Priority Queues

For double-ended priority queues (DEPQ), the operations supported by a min priority queue are:

- SP1: Return an element with minimum priority.
- SP2: Return an element with maximum priority.
- SP3: Insert an element with an arbitrary priority.
- SP4: Delete an element with minimum priority.
- SP5: Delete an element with maximum priority.

Primary operations
- **Insert**
- **Delete Max**
- **Delete Min**

Note that a single-ended priority queue supports just one of the above remove operations.

Leftist Trees

Leftist trees provide an efficient implementation of meldable priority queues.

- Linked binary tree.
- **Can do everything a heap can do** and in the same asymptotic complexity.
- Can meld two leftist tree priority queues in $O(\log n)$ time.
Extended Binary Trees

- Two extended binary trees

![Diagram of Extended Binary Trees]

Internal node

External node

Height-Based Leftist Tree (HBLT)

- For any node \( x \) in an extended binary tree. Let \( \text{leftChild}(x) \) and \( \text{rightChild}(x) \) denote the left and right children of the internal \( x \), respectively. Define \( \text{shortest}(x) \) be the length of a shortest path from \( x \) to an external node in the subtree rooted at \( x \).

If \( x \) is an external node, \( \text{shortest}(x) = 0 \);
otherwise

\[
\text{shortest}(x) = 1 + \min \{ \text{shortest}(\text{leftChild}(x)), \text{shortest}(\text{rightChild}(x)) \}
\]

![Diagram of Height-Based Leftist Tree (HBLT)]
The Definition of Leftist Trees

A leftist tree is a binary tree such that if it is not empty, then for every internal node $x$:

$$\text{shortest} (\text{leftChild}(x)) \geq \text{shortest} (\text{rightChild}(x))$$


![Diagram of leftist trees](image)

Lemma of Leftist Trees

Let $x$ be the root of a leftist tree that has $n$ internal nodes.

a) $n \geq 2^{\text{shortest}(x)} - 1$

b) The rightmost root to external node path is the shortest root to external node path. Its length is

$$\text{shortest}(x) \leq \log_2(n+1)$$
Min-Leftist Trees

- **Definition:** A min-leftist tree (max-leftist tree) is a leftist tree in which the key value in each node is no larger (smaller) than the key values in its children (if any).

- Two min-leftist trees:

```
   2
  / \
 7   50
 /     /
11     80
  \
 13

   5
  / |
 9  8
 /  |
12  10
 /  / |
20 18 15
```

Combination of leftist trees

- **The combination steps (min-leftist trees)**
  1. **Choose minimum root** of the two trees, A and B.
  2. Leave the left subtree of smaller root (suppose A) unchanged and combine the right subtree of A with B. Back to step 1, until no remaining vertices.
  3. Compare shortest(x) and swap to make it satisfy the definition of leftist trees.
Example 1: Meld Two Leftist Trees

combine

Example 1: Meld Two Leftist Trees
Example 1: Meld Two Leftist Trees

[Diagram of two leftist trees being melded together]
Combination of leftist trees

Both insert and delete min operations can be implemented by using the combine operation.

- **Insert:*** Treat the inserting node as a single node binary tree. Combine with the original one.
- **Delete:*** Remove the node can get two separate subtrees. Combine the two trees.
**Weight-Based Leftist Tree (WBLT)**

- Define the weight $w(x)$ of node $x$ to be the number of internal nodes in the subtree with root $x$. The weight is 0 for an external node and is the sum of the children weights of an internal node.

- A binary tree is a WBLT iff at every internal node the $w$ value of the left child is greater than or equal to the $w$ value of the right child.

- For any node $x$ in an extended binary tree. The length, $\text{rightmost}(x)$, of the rightmost path from $x$ to an external node satisfies $\text{rightmost}(x) \leq \log_2(w(x)+1)$.

**Binomial Heaps**

- **Definition of Binomial trees:**
  - A binomial tree $B_k$ has $2^k$ nodes with height be $k$.
  - It has $\binom{i}{j}$ nodes at depth $i$.
  - The $i$th child of root is the root of subtree $B_{i-1}$.

- Depth 1: 4 nodes.
- Depth 2: 6 nodes.
- Depth 3: 4 nodes.
Definition of Binomial Heaps

- A min (max) binomial heap (B-heap) is a collection of min (max) trees.
- The min trees should be Binomial trees.

Example: a B-heap consists of three min trees

```
  1
 / \
 5   4
/     /
8   10  3
```

Representation of Binomial Heaps

- The representation of B-heap:
  - Degree: number of children a node has.
  - Child: point to any one of its children.
  - Left_link, Right_link: maintain doubly linked circular list of siblings.
- The position of pointer a is the min element.

```
  a
/    
 1
/    
5   4
/     /
8   10  3
```

- a → pointer
- Siblings
- parent
- child
Combination of Binomial Heaps

- Combine a min tree of 40

![Diagram of binomial heaps combining a min tree of 40](image_url)
Combination of Binomial Heaps

- Pairwise combine

\[ a \]
Combination of Binomial Heaps

- Pairwise2 combine

![Diagram of Binomial Heaps combination](image)
Combination of Binomial Heaps

- Pairwise2 combine

Combination of Binomial Heaps

- Pairwise3 combine
Combination of Binomial Heaps

- Pairwise3 combine

Diagram showing the combination of binomial heaps with nodes and arrows indicating the structure and connections.
Combination of Binomial Heaps

- The insertion is to combine a single vertex B-heap to original B-heap.
- After we delete the min element, we get several B-heaps that originally subtrees of the removed vertex. Then combine them together.

Time Complexity of Binomial Heaps

<table>
<thead>
<tr>
<th></th>
<th>Leftist trees</th>
<th>Binomial heaps</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Actual</td>
<td>Amortized</td>
</tr>
<tr>
<td>Insert</td>
<td>O(log n)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Delete min (or max)</td>
<td>O(log n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Meld</td>
<td>O(log n)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>
Fibonacci Heaps

- A min (max) Fibonacci heap (F-heap) is a collection of min (max) trees.
- The min trees need not be Binomial trees.
- B-heaps are a special case of F-heaps.
  - So that what B-heaps can do can be done in F-heaps.
  - More than that, F-heap may delete an arbitrary node and decrease key.

Deletion From an F-heap

- The deletion of 12
Deletion From an F-heap

- After the deletion of 12

![Diagram](image1)

Deletion From an F-heap

- After the deletion of 12

![Diagram](image2)
Decrease Key From an F-heap

- Decrease key of 15 to be 11

After decrease key of 15 => 11
Decrease Key From an F-heap

- After decrease key of 15 => 11

Cascading Cut

- When theNode is cut out of its sibling list in a remove or decrease key operation, follow path from parent of theNode to the root.
- Encountered nodes (other than root) with ChildCut = true are cut from their sibling lists and inserted into top-level list.
- Stop at first node with ChildCut = false.
- For this node, set ChildCut = true.
Cascading Cut Example

Decrease key by 2.

Cascading Cut Example
Cascading Cut Example

Tree Structure

C-C Tsai P.45

Cascading Cut Example

Tree Structure

C-C Tsai P.46
Cascading Cut Example

Actual complexity of cascading cut is $O(h) = O(n)$.

Time Complexity of Fibonacci Heaps

<table>
<thead>
<tr>
<th></th>
<th>Actual</th>
<th>Amortized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Delete min (or max)</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Meld</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Delete</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Decrease key (or increase)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
### Pairing Heaps

<table>
<thead>
<tr>
<th>Operation</th>
<th>Fibonacci Complex</th>
<th>Pairing Complex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Delete min (or max)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Meld</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Delete</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Decrease key (or increase)</td>
<td>O(n)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

**Actual Complexity**

### Pairing Heaps

<table>
<thead>
<tr>
<th>Operation</th>
<th>Fibonacci Complex</th>
<th>Pairing Complex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Delete min (or max)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>Meld</td>
<td>O(1)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>Delete</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>Decrease key (or increase)</td>
<td>O(1)</td>
<td>O(log n)</td>
</tr>
</tbody>
</table>

**Amortized Complexity**
Pairing Heaps

- Experimental results suggest that pairing heaps are actually faster than Fibonacci heaps.
  - Simpler to implement.
  - Smaller runtime overheads.
  - Less space per node.

Definition

- A min (max) pairing heap is a min (max) tree in which operations are done in a specified manner.
Node Structure

- **Child**
  - Pointer to first node of children list.

- **Left and Right Sibling**
  - Used for **doubly linked linked list** (not circular) of siblings.
  - Left pointer of first node is to parent.
  - \( x \) is first node in list iff \( x.\text{left.child} = x \).

- **Data**
- Note: No Parent, Degree, or ChildCut fields.

---

**Meld – Max Pairing Heap**

- **Compare-Link Operation**
  - Compare roots.
  - Tree with smaller root becomes leftmost subtree.

\[
\begin{array}{c}
\text{7} \\
\text{6}
\end{array}
\quad +
\quad \begin{array}{c}
\text{9} \\
\text{6} \\
\text{7} \\
\text{3}
\end{array}
\quad =
\quad \begin{array}{c}
\text{9} \\
\text{7} \\
\text{6} \\
\text{7} \\
\text{3} \\
\text{6}
\end{array}
\]

- \textbf{Actual cost} = \( O(1) \).
Insert

- Create 1-element max tree with new item and meld with existing max pairing heap.

\[ + \text{insert}(2) = \]

\[
\begin{array}{c}
\text{9} \\
\text{7} \\
\text{6} \\
\text{3} \\
\end{array}
\begin{array}{c}
\text{6} \\
\text{7} \\
\text{9} \\
\end{array}
\begin{array}{c}
\text{2} \\
\text{7} \\
\text{6} \\
\text{3} \\
\end{array}
\]

\[ + \text{insert}(14) = \]

\[
\begin{array}{c}
\text{14} \\
\text{9} \\
\text{7} \\
\text{6} \\
\text{3} \\
\end{array}
\begin{array}{c}
\text{7} \\
\text{6} \\
\text{7} \\
\end{array}
\begin{array}{c}
\text{6} \\
\text{3} \\
\end{array}
\]

*Actual cost = O(1).*
Worst-Case Degree

- Insert 9, 8, 7, ..., 1, in this order.

- Worst-case degree = \( n - 1 \).

Worst-Case Height

- Insert 1, 2, 3, ..., n, in this order.

- Worst-case height = \( n \).
IncreaseKey(theNode, theAmount)

- Since nodes do not have parent fields, we cannot easily check whether the key in theNode becomes larger than that in its parent.
- So, detach theNode from sibling doubly-linked list and meld.

If theNode is not the root, remove subtree rooted at theNode from its sibling list.
IncreaseKey(theNode, theAmount)

Meld subtree with remaining tree.

• Actual cost = $O(1)$. 
Delete Max

- If empty => fail.
- Otherwise, remove tree root and meld subtrees into a single max tree.
- How to meld subtrees?
  - Good way => $O(\log n)$ amortized complexity for remove max.
  - Bad way => $O(n)$ amortized complexity.

Bad Way To Meld Subtrees

- $\text{currentTree} =$ first subtree.
- for (each of the remaining trees)
  \[ \text{currentTree} = \text{compareLink (currentTree, nextTree)}; \]
Example

- Delete max.
  - Meld into one tree.

Example

- Actual cost of insert is 1.
- Actual cost of delete max is degree of root.
- \(n/2\) inserts (9, 7, 5, 3, 1, 2, 4, 6, 8) followed by \(n/2\) delete maxs.
  - Cost of inserts is \(n/2\).
  - Cost of delete maxs is \(1 + 2 + \ldots + n/2 - 1 = \Theta(n^2)\).
  - If amortized cost of an insert is \(O(1)\), amortized cost of a delete max must be \(\Theta(n)\).
Good Ways To Meld Subtrees

- Two-pass scheme and Multipass scheme. Both have same asymptotic complexity.
- **Two-pass scheme** gives better observed performance.
  
  **Pass 1.**
  - Examine subtrees from left to right.
  - Meld pairs of subtrees, reducing the number of subtrees to half the original number.
  - If # subtrees was odd, meld remaining original subtree with last newly generated subtree.
  
  **Pass 2.**
  - Start with rightmost subtree of Pass 1. Call this the working tree.
  - Meld remaining subtrees, one at a time, from right to left, into the working tree.

---

**Two-Pass Scheme – Example**

**Pass 1**

```
T1   T2   T3   T4   T5   T6   T7   T8
S1   S2   S3   S4
```

**Pass 2**

```
R1   R2   R3
```
Multipass Scheme

- Place the subtrees into a FIFO queue.
- Repeat until 1 tree remains.
  - Remove 2 subtrees from the queue.
  - Meld them.
  - Put the resulting tree onto the queue.

Multipass Scheme – Example
Multipass Scheme--Example

Actual cost = $O(n)$.
Delete Nonroot Element

- Remove theNode from its sibling list.
- Meld children of theNode using either 2-pass or multipass scheme.
- Meld resulting tree with what’s left of original tree.

Delete(theNode)

Remove theNode from its doubly-linked sibling list.
Delete(theNode)

Meld children of theNode.

Delete(theNode)

Meld with what’s left of original tree.
Delete(theNode)

• Actual cost = $O(n)$.

MIN-MAX Heaps

- Complete binary tree.
- Set root on a min level.
- A node $x$ in min level would have smaller key value than all its descendents. ($x$ is a min node.)
Insertion into a min-max heap

Insert 5

Insert 5
Insertion into a min-max heap

Insert 5

Check if it satisfies min heap.
(Compare with its parent.)
If no, move the key of current
parent to current position.
If yes, skip.
Insertion into a min-max heap

**Initial Heap={7,70,40,30,9,10,15,45,50,30,20,12}**

parent=7; We want to **insertion 80** into the position 13

*n=13, item=80;
80>10 $\rightarrow$ verify_max(heap,13,80)

Input:

verify_max(Heap,13,80)
grandparent=3
80>40

```
void verify_max(Heap, i, item) {
  /* follow the nodes from the max node i to the root and insert item into its proper place */
  int grandparent = i/4;
  while (grandparent) {
    if (item > heap[grandparent]) { // heap[i] > heap[grandparent];
      heap[i] = heap[grandparent];
      i = grandparent;
      grandparent /= 4;
    } else {
      break;
    }
  } // end while
  heap[1] = item;
}
```
Insertion into a min-max heap

- **Input**
  - `verify_max(Heap, 3, 80) → grandparent = null
  - break;
  - heap[3] = 80;

```c
void verify_max(element heap[], int i, element item)
{
    /* follow the nodes from the max node i to the root and
    insert item into its proper place */
    int grandparent = i/2;
    while (grandparent)
    {
        if (item.key > heap[grandparent].key) {
            heap[i] = heap[grandparent];
            i = grandparent;
            grandparent = i/2;
        }
        else
            break;
    }
    heap[i] = item;
```

- **The time complexity of insertion into a min-max heap with n elements is O(log n).**
  - A min-max heap with n elements has O(log n) levels.
Deletion of min element

- The smallest element is in the root.
- We do the deletion as follows:
  - Remove the root node and the node \( x \) which is the end of the heap.
  - Reinsert the key of \( x \) into the heap.

Image: A binary heap diagram showing the deletion of the minimum element 7.
Deletion of min element

Delete 7

The reinsertion may have 2 cases:

Case 1: No child. (Only one node in the heap)

It means the item is the only one element, so it should be at root in heap.
Deletion of min element

Case 2: The root has at least one child
Find the min value. (Let this be node $k$.)

a. $item.key \leq heap[k].key$
   It means the item is the min element, and the min element should be at root in heap.

```
        i
       /|
      / |   \\
     /  |   k
    /   |   \\
   /     |
  i------i
```

b. $item.key > heap[k].key$ and $k$ is child of root.
   Since $k$ is in max level, it has no descendants.

```
        i
       /|
      / |   \\
     /  |   k
    /   |   \\
   /     |
  k------i
```
Deletion of min element

c. item.key > heap[k].key and k is grandchild of root.
Example \((p>i)\): We should make sure the node in max level contains the largest key. Then redo insertion to the subtree with the root in red line.
Deletion of min element

Deletion of min element of a min-max heap with n elements need $O(\log n)$ time.

In each iteration, i moves down two levels. Since a min-max heap is a complete binary tree, heap has $O(\log n)$ levels.
Symmetric Min-Max Heaps

- Symmetric Min-Max Heap (SMMH) is a complete binary tree.
- SMMH is either empty or satisfies the properties:
  - The root contains no element.
  - The left subtree is a min-heap.
  - The right subtree is a max-heap.
  - If the right subtree is not empty. Let i be any node in left subtree, and j be the corresponding node in the right subtree. If no, choose the parent one. i_key <= j_key.

Example of a SMMH
SMMH

Let i be any node in left subtree and j be the corresponding node in the right subtree. The relation between i and j.

\[ j = i + 2 \left\lfloor \log_2 i \right\rfloor + 1; \]

if \((j > n)\)

\[ j/ = 2; \]

Ex1:

\[
i = 4; \\
j = 4 + 2^{(2-1)} = 6;
\]

Ex2:

\[
i = 9; \\
j = 9 + 2^{(3-1)} = 13; \\
j = 13 > 12 \Rightarrow j = 6;
\]

Insertion into a SMMH

The insertion steps

1. max_heap(n): Check iff n is a position in the max-heap of the SMMH.
2. min_partner(n) or max_partner(n): Compute the min-heap / max-heap node that corresponding to n.

\[
( n - 2^{\left\lfloor \log_2 n \right\rfloor - 1} ) / \left( n + 2^{\left\lfloor \log_2 n \right\rfloor - 1} \right) / 2
\]

1. Compare key of i and j to satisfy SMMH.
2. min_insert or max_insert.
Insertion into a SMMH

Insert 4: \( i = 13 - 2^{(3-1)} = 13 - 4 = 9 \) for min_heap

---

Insertion into a SMMH
Insertion into a SMMH

Step 1. max_heap.
Step 2. min_partner.
Step 3. Compare key of i and j.
Step 4. min_insert or max_insert.

The time complexity is $O(\log n)$ as the height of the SMMH is $O(\log n)$. 

```
void deep_insert(element deep[], int *n, element x)
/* Insert x into the deep */
int i = 0;
if (n == MAX-SIZE) {
    printf("error, 'The heap is full'");
    exit(1);
}
    if (missing == -1)
        deep[missing] = x; /* Insert into empty deep */
    else
        for (i = 0; i < n; i++)
            if (deep[i] > x)
                deep[i] = deep[i] - 1;
            else
                deep[i] = x;
    n = n + 1;
```

C-C Tsai
P.103
Deletion of min element

The deletion steps

1. Save the last element as temp and remove this node from SMMH.
2. Find the node with smaller key from the children of removed minimum element and loop down until reaching leaves.
3. Insert temp into the left subtree of SMMH.

Delete 5:
Deletion of min element

```
<table>
<thead>
<tr>
<th>10</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>19</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>25</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>temp:20</td>
</tr>
</tbody>
</table>
```

Deletion of min element

```
<table>
<thead>
<tr>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
<tr>
<td>15</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>25</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>temp:20</td>
</tr>
</tbody>
</table>
```
Deletion of min element

Deletion of min element
Deletion of min element

Step 1. Save the min element.

Step 2. Find the node with smaller key.

Step 3. (Exercise 2).

The time complexity is $O(\log n)$ as the height of the SMMH is $O(\log n)$.

Interval Heaps

- Complete binary tree.
- Each node (except possibly last one) has 2 elements. Last node has 1 or 2 elements.
- Let $a$ and $b$ be the elements in a node $P$, $a \leq b$. $[a, b]$ is the interval represented by $P$.
- The interval represented by a node that has just one element $a$ is $[a, a]$.
- The interval $[c, d]$ is contained in interval $[a, b]$ iff $a \leq c \leq d \leq b$.
- In an interval heap each node’s (except for root) interval is contained in that of its parent.
Example of an Interval Heap

Left end points define a **min heap**. 
Right end points define a **max heap**.

Min and max elements are in the root.
Store as an array
Height is \( \sim \log_2 n \).
Insert An Element

Insert 27.
New element becomes a left end point.
Insert new element into min heap.

Another Insert

Insert 18.
New element becomes a left end point.
Insert new element into min heap.
Another Insert

Insert 18.
New element becomes a left end point.
Insert new element into min heap.
Yet Another Insert

Insert 82.
New element becomes a right end point.
Insert new element into max heap.

After 82 Inserted
One More Insert Example

Insert 8.
New element becomes both a left and a right end point.
Inserted new element into min heap.

After 8 Is Inserted
Remove Min Element

Remove Min element

- If \( n = 0 \) => fail.
- If \( n = 1 \) => heap becomes empty.
- If \( n = 2 \) => only one node, take out left end point.
- If \( n > 2 \) => not as simple.

Remove Min Element Example

Remove left end point from root.
Remove left end point from last node.
Delete last node if now empty.
Reinsert into min heap, begin at root.
Remove Min Element Example

Swap with right end point if necessary.
Remove Min Element Example

Swap with right end point if necessary.
Application Of Interval Heaps

- Complementary range search problem.
  - Collection of 1D points (numbers).
  - **Insert** a point. \(O(\log n)\)
  - **Remove** a point given its location in the structure. \(O(\log n)\)
  - Report all points not in the range \([a,b]\), \(a \leq b\).
    \(O(k)\), where \(k\) is the number of points not in the range.
    1. If the interval tree is empty, return.
    2. If the root interval is contained in \([a,b]\), then all points are in the range, return.
    3. Report the end points of the root interval that are not in the range \([a,b]\).
    4. **Recursively search the left subtree** of the root for additional points that are not in the range \([a,b]\).
    5. **Recursively search the right subtree** of the root for additional points that are not in the range \([a,b]\).
    6. return.

Example of Range Search

\[5,100\]: All points are in the range
\[2,65\]: Some points are in the range