“Good” characteristics of BST:
- Simplicity.
- Support general search/delete as well as special searchMin(Max)/deleteMin(Max) operations.
- Can easily be sorted (inorder traversal).
- Can easily be stored (preorder traversal).
- Good average performance, $T_a(n) = O(lgn)$.

“Bad” characteristics of BST:
- Worst-case complexity depends on height of tree. Hence, $T_w(n) = O(n)$.
- Inefficient when many items are having identical keys since height may increase.

Q: Can we design an efficient ADT with $T_w(n) = T_a(n) = O(lgn)$?

In many applications, a less powerful data structure may be sufficient. What if general delete operations are not required or if they are only performed infrequently? Can we design an efficient ADT that will allow us to remove a data object with maximum, or minimum, priority whenever a delete operation is
performed? These applications lead us to the design of a class of ADTs called *priority queues*.

**ADT: Priority Queue.**
A collection class, whose items have all been assigned a priority, supports the following operations:

1. \textit{PQInsert(in newItem: PQItemType)}
2. \textit{PQDelete(out priorityItem: PQItemType)}
3. \textit{createPQ():}
4. \textit{destroyPQ():}
5. \textit{PQIsEmpty():}
6. \textit{PQSize():}

**Simplest Approaches:**
Sorted and unsorted array/linked list.

**Better Approach:**
BST.

**Best Approach:**
k-Heap, \( k \geq 2 \).

**Defn:** A max (min) *k-heap* is a k-ary tree H such that it satisfies the following properties:

1. **Structural Property:** H is a complete k-ary tree, and
2. **Heap-Ordered Tree Property:** Priority of any node in H \( \geq (\leq) \) priority of all its descendants.
Example: A max 2-heap $H$.

Example: A min 3-heap $H$. 
Implementations of k-Heaps, $k \geq 2$:
1. Pointer-based implementation:
   Inefficiency! Why?

2. Array-based Implementation:
   Given a heap $H$ with $n$ nodes. $H$ can be represented using an array $A[0:\text{Max\_Queue}-1]$ such that
   1. Root of $T$ at $A[0]$,
   2. Parent of $A[i]$ at $A[(i-1)/k]$ if exists,
   3. The $j$th child of $A[i]$ at $A[ki+j]$, $1 \leq j \leq k$, if exist.

Remarks:
- When $i = 0$, $(i-1)/k = -1$, implying that $A[i]$ is the root of $H$.
- For $n \geq 1$, $A[i]$ is a leaf iff $ki \geq n-1$.
- Given $A[i]$, the parent and the children of $A[i]$ can be computed in $O(1)$ time.
- H can also be implemented using a similar array structure by storing the root at $A[1]$. (HW)

PQ Operations:
Two-steps process:
1. After insertion/deletion, try to maintain a complete k-ary tree structure for $H$.
2. Re-structure the complete k-ary tree from (1) so that it will satisfy the heap-ordered tree property.
1. Insert(x,H):

Finding location in H for insertion:
Consider inserting 9 into the following 2-heap H.

Q: Where will 9 go (in order to get back a complete binary tree)?
Heapifying/Restructuring resulting tree after insert:

After inserting a node with $x$ into $H$, the newly inserted node may or may not satisfy the heap-ordered tree property. If the heap-ordered tree property is violated, $x$ must have priority higher than its parent. Hence, we can simply swap $x$ with its parent and verify the heap-ordered tree property again by comparing the priority of $x$ with its new parent.

In general, for inserting a new item $x$ into $H$, we need to find a location to insert $x$ along the path from $x$ (after insertion) to the root of $H$ by repeatedly comparing $x$ with his parent, grandparent, ..., until either a node with priority $\geq x$ is found or the root of $H$ is reached. Once the final location for $x$ is found, it will then be inserted.

**Remark:** Do not insert and remove $x$ repeatedly. Find the final location for $x$ and then insert it. Once $x$ is inserted, it stays.
Example: Consider inserting 9 into the heap H.

Create a new location for 9:

Compare 9 with its parent 6; 6 moves down:
Compare 9 with its parent 7; 7 moves down:

Compare 9 with its parent 8; 8 moves down:
Insert 9 into its final location; process terminates:

Complexity Analysis:

Observe that a k-heap with m nodes has height \( \lceil \log_k m \rceil \) and requires at most \( \lceil \log_k (m+1) \rceil \) comparisons to insert a new node into this heap.

Hence,

\[
T_w(n) = \lceil \log_k (n+1) \rceil = O(\log_k n).
\]
2. \texttt{delete(H)}:
Consider deleting the highest priority item (root) from the original heap H.

After 8 is removed, in order to get back a complete 2-ary tree, one must replace the root of H with the last item (in level order) in H.
In general, we must replace the root (highest priority item) with the “last” item $x$ and then percolate down along a path from the root to a leaf by repeatedly compare $x$ with its child (children), swapping with the larger child if necessary, until $x \geq$ its children or a leaf is reached.

**Example:** Deleting the highest priority item (root) and then heapify the resulting complete binary tree.
Compare 5 with its two children, 7 moves up:

```
    7
   / \  
  5   5
 / \  /  
5   6 3   2
```

Compare 5 with its two children, 6 moves up:

```
    7
   / \  
  6   5
 / \  /  
5   3 5   2
```

12
**Insert 5 into its final location; process terminates:**

![Binary Heap Diagram](attachment:image.png)

**Complexity Analysis:**

Observe that to move a node down one level in a k-heap requires k comparisons. Since a heap with m nodes has height $\lceil \log_k m \rceil$, it requires at most $k \times \lceil \log_k m \rceil$ comparisons to delete its root. Hence,

$$T_w(n) = k \times \lceil \log_k n \rceil = O(k \log_k n).$$
Q: How do we build an initial heap $H$?

**Two possible build-heap methods:**
1. Top-down approach:
   Insert items of $S$ one at a time, in the order given, into an initially empty heap.

**Example:** Build a max 2-heap for $S = \{5,3,2,6,8,5,7,1,3,5\}$.

**Insert 5, 3, 2, 6:**

![Tree diagram showing the insertion of 5, 3, 2, 6 into the heap]

1. Insert 5:
   - $5$
2. Insert 3:
   - $5$
   - $3$
3. Insert 2:
   - $5$
   - $3$
   - $2$
4. Insert 6:
   - $5$
   - $6$
   - $3$
   - $2$

5. After inserting 6:
   - $5$
   - $6$
   - $3$
   - $2$

The heap is now complete.
Insert 8:

```
  6
 / \
5   2
/ \
3  8
      \
  8
 / \ \
6   2
/ \ 
3   5
```

Insert 5:
Insert 7:
**Insert 1, 3, 5:**

![Binary Tree Diagram]

2. **Bottom-up approach:**
   First form a complete binary tree H for S according to its given order. Observe that a leaf by itself is already a heap. If we scan the nodes of H in the reverse level order (leaf-to-root and right-to-left), two heaps can then be combined together by inserting a new element x as the new root of the resulting heap (as in delete operation). Hence, we can grow a heap for S in a bottom-up fashion by using *heapify* operations as in delete operation:

![New Element Insertion Diagram]
Build a max 2-heap for \( S = \{5,3,2,6,8,5,7,1,3,5\} \) using bottom-up approach with heapify operations:

**Form initial complete binary tree:**

![Binary Tree Diagram]

Observe that since \( n = 10 \), the first non-leaf node needs to be checked has array index \( \lfloor 10/2 \rfloor - 1 = 4 \), followed by nodes with index 3, 2, 1, 0.

**For \( A[4] \), compare 8 with 5; no swap:**

![Binary Tree Diagram]
For $A[3]$, compare 6 with 1 and 3; no swap:

```
  5
 /   \
3     2
/     /\n6     5
/     /\n1     7
3     5
```

For $A[2]$, compare 2 with 5 and 7, swap with 7:

```
  5
 /   \
3     7
/     /\n6     5
/     /\n1     2
3     5
```
For $A[1]$, compare 3 with 6 and 8, swap with 8:

Continued: Compare 3 with 5, swap with 5:
For $A[0]$, compare 5 with 8 and 7, swap with 8:

Continued: Compare 5 with 6 and 5, swap with 6:

Continued: Compare 5 with 1 and 3, process terminates!

HW: Redo the above examples using arrays.
Q: Which build-heap method should we use to build an initial heap?

Complexity Analysis for buildHeap Operations:
Let’s compare the two buildHeap operations using a 2-heap:

1. Top-down approach using insert operations:
   Recall that it requires at most \[\lfloor \log(m+1) \rfloor\] comparisons to insert a new node into a heap with m nodes. Hence,

\[
T_w(n) = \lfloor \log 1 \rfloor + \lfloor \log 2 \rfloor + \ldots + \lfloor \log n \rfloor \\
\leq \log 1 + \log 2 + \ldots + \log n \\
= O(\log n!) \\
= O(n \log n)
\]

Observe that if we let I(H) be the set of internal nodes in H and d(x) be the depth of x, we have
\[
T_w(n) = \lfloor \log 1 \rfloor + \lfloor \log 2 \rfloor + \ldots + \lfloor \log n \rfloor \\
= \sum_{x \in I(H)} d(x), \\
= \text{internal path length of H.}
\]
2. Bottom-up approach using heapify operations:
Recall that it requires at most 2 comparisons to move a node down one level in a heap. For a node x of height h(x), it will require at most 2*h(x) to heapify x. Hence,

$$T_w(n) = \sum_{x \in I(H)} 2 \cdot h(x).$$

Observe that, in a complete binary tree, there are
- 1 = $2^0$ node of height h(H),
- 2 = $2^1$ nodes of height h(H)−1,
- 4 = $2^2$ nodes of height h(H)−2,
- $\ldots$
- $2^i$ nodes of height h(H)−i,
- $\ldots$
- $2^{h(H)−2}$ nodes of height 2,
- $2^{h(H)−1}$ nodes of height 1,
- \leq 2^{h(H)} nodes of height 0.

By summing all the nodes according to their height, we have
Conclusion:
Use O(n) bottom-up approach to build an initial heap.

A Simple Application of Priority Queue:
PQ Sorting:
1. Build a PQ Q for S.
2. Repeatedly delete until Q is empty.

Heap Sort:
1. Build a max-heap H for S.
2. Deletemax repeatedly until H is empty. (Swap max item with the current last item in array.)
Hence, T(n) = O(nlgn)
Final Remarks:
1. What is $k$ in the $k$-heap?
   Observe that there is a tradeoff between insert and delete operations since large $k$ implies faster insert but slower delete.

For $k$-heap, $k \geq 2$:
- **Insert**: $T_w(n) = O(\log_k n)$
- **Delete**: $T_w(n) = O(k \log_k n)$

2. Worst-Case Time Comparison of PQ Implementations:

<table>
<thead>
<tr>
<th>PQ Operation</th>
<th>Heap*</th>
<th>BST</th>
<th>Sorted IList</th>
<th>Unsorted Array</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Build/Organize</strong></td>
<td>O(n)</td>
<td>O(n lg n)*</td>
<td>O(n^2)</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>Insert</strong></td>
<td>O(lgn)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(1)</td>
</tr>
<tr>
<td><strong>Find</strong></td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>GeneralDelete</strong></td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>DeleteMax</strong></td>
<td>O(lgn)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>DeleteMin</strong></td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>

\*Using max 2-heap.

*To build a BST by first sorting, and then insert, the input into a complete BT requires $T_w(n) = O(n \text{lg} n)$.

3. Other operations such as find(x) and delete(x,H) possible but inefficient. (HW)
Extension:

How do we design an ADT that will support both deletemin(Q) and deletemax(Q) operations?

A double-ended priority queue (DEPQ) H is a collection of zero or more data objects together with the following operations:

1. findMin()
2. findMax()
3. insert(in newItem: DEQItemType)
4. deleteMin(out priorityItem: DEQItemType)
5. deleteMax(out priorityItem: DEQItemType)
6. createDEQ()
7. destroyDEQ()
8. DEQisEmpty()
9. DEQSize()

Simplest DEPQ:

Dual heap: A pair of min heap and a max heap with corresponding pointer between each pair of corresponding elements.
Example:

Operations:
1. inser(x): insert x into both min and max heaps, set corresponding pointers.
2. deleteMin: deletMin from min heap; follow pointer to max heap and delete corresponding element.
3. deleteMax: deletMax from max heap; follow pointer to min heap and delete corresponding element.

Complexity: Same as min or max heap.
$T_w(n) = O(\log n)$.

Remark: Inefficient in memory; not as fast as min/max heap.
A Better DEPQ: Minmax Heap

A minmax heap $H$ is an extension of 2-heap by fusing a min heap and a max heap together such that

1. Each node $x$ in $H$ is either a min node ($x$ is $\leq$ all its descendants), or a max node ($x$ is $\geq$ all its descendants).
2. All nodes having the same level order must be of the same type. A min-level (max-level) in $H$ is a collection of all the min (max) nodes having the same level number.
3. Starting with the root of $H$ at the min-level, nodes are alternating between min-level and max-level.

Remark: A minmax heap is a tree satisfying the following properties:

1. Structural Property: $H$ is a complete binary tree.
2. Relational Property: $H$ satisfies the minmax heap property with which, starting at the root at min-level, nodes are alternating between min-level and max-level.
**Example:** A minmax heap H.

![Minmax heap diagram](attachment:heap_diagram.png)

**HW:** Define and explore maxmin heap.

**Implementation:**
Sequential array implementation with root of H at A[1].

**Example:** Array implementation of H.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>75</td>
<td>65</td>
<td>36</td>
</tr>
<tr>
<td>36</td>
<td>48</td>
<td>12</td>
<td>58</td>
</tr>
<tr>
<td>65</td>
<td>38</td>
<td>56</td>
<td>58</td>
</tr>
</tbody>
</table>

**Minmax heap operations:**
Two-step process as in binary heap:
1. Maintain complete binary tree structure after each insert/deleteMin/deleteMax operation.
2. Restructure resulting tree to restore minmax heap property.
1. Insert(x,H):
   As in heap, insert x into the last+1 position and then restore the minmax heap property.
Consider the following two cases:
(1) H = ∅: Return heap with x.
(2) H ≠ ∅: Consider the parent, p(x), of x, after inserting x into H.
   If x = p(x), done.
   else consider x < p(x), or x > p(x).
   (a) Assume x < p(x): If p(x) is a min node, then x < y for all max node y on the path from x to the root of H. Similarly, if p(x) is a max node, then x < z for all max node z on the path from x to the root of H. Hence, we need only compare x with the min nodes along the path from x to the root in order to restore the minmax heap property as in a min heap.
   (b) Assume x > p(x): If p(x) is a min node, then x > y for all min node y on the path from x to the root of H. Similarly, if p(x) is a max node, then x > z for all min node z on the path from x to the root of H. Hence, we need only compare x with the max nodes along the path from x to the root in order to restore the minmax heap property as in a max heap.
**Example:** Build a minmax heap and a maxmin heap by inserting \(<6, 8, 5, 2, 7, 8, 2, 9, 12, 1>\) into an initially empty heap.

**Minmax heap:**

```
                               1
                              / \
                             12   8
                            /   / \
                           5   2   6   2
                          /     /   /   / \
                         8   9   7   6   8   2
```

**Maxmin heap:**

```
                               12
                              /   \
                             1   2
                            /     \
                           9   7   8   5
                          /       /   /   /   /   \
                         6   8   2   9   7   8   5
```

**Remark:** A minmax heap can also be built using a modified bottom-up approach.
Q: Given a node x at A[i]. How do you determine whether x is a min node or a max node?
   Min node: \(\lfloor \log(i) \rfloor = \text{even} \)
   Max node: \(\lfloor \log(i) \rfloor = \text{odd} \)

Q: How do you locate the grandparent of x if exists?
   Grandparent of A[i] at A\(\lceil \frac{i}{4} \rceil \).

HW: Implement insert(H) for minmax heap and maxmin heap.

Consider deleteMin/deleteMax operations.
Q: Where is the min (max) element in a minmax heap?
   Min element: Root of H.
   Max element: A child of root if |H| > 1.

General approach: Replace the min (max) element of H by an element in H.
   (1) Find the second smallest/largest element s in H and use it, or the last element in H, to patch up the hole left by the deletion of the min (max) element.
   (2) Remove and then re-insert the last element x (in level order traversal) back into H to patch up the “hole” left by the deletion of s if used.
2. **Deletemin(H):**

Consider the following four cases:

1. $\text{H} = \emptyset$: Return error.
2. $|\text{H}| = 1$: Return $\emptyset$.
3. $|\text{H}| = 2$: Replace root with its only child.
4. $|\text{H}| \geq 3$: Delete root $r$ and then find the second smallest element $s$ in $\text{H}$, which is either a child ($|\text{H}| = 3$), or a grandchild, of $r$ to patch up the hole. Compare the last element $x$ in $\text{H}$ with $s$ (before removing any element from $\text{H}$).
   
   - (a) $x \leq s$: Remove $x$; $x$ becomes new root of $\text{H}$.
   - (b) $x > s$: Remove $s$; $s$ become the new root of $\text{H}$. Observe that a new “hole” is now generated in $\text{H}$ and we will delete $x$ (the very first time) and use it to patch up the hole.

   - (i) If $s$ is a child of $r$, then $x$ can be used to patch up the hole vacated by $s$. (Why?)
   - (ii) If $s$ is a grandchild of $r$, $s$ is a min node and it has a parent $p(s)$, which is a max node, in $\text{H}$. Compare $x$ with $p(s)$.
     
     - (a) If $x \leq p(s)$, recursively use $x$ to patch up the root of the minmax heap rooted at the original $s$ location.
(b) If $x > p(s)$, replace $p(s)$ with $x$ in $H$ and then recursively use $p(x)$ to patch up the root of the minmax heap rooted at the original $s$ location.

**Q:** How about `deletemax(H)`?

   Similar method except using maxmin heap concept instead.

**HW:** Implement the `deletemin(H)` and `deletemax(H)` operations.

**Complexity Analysis:**

   For `insert`, `deletemin`, `deletemax` operations,
   $T_w(n) = O(h(H)) = O(\log n)$.

3. **Build-Heap(H):**

   Two approaches similar to building a binary heap:
   1. Top-down $O(n\log n)$ approach using insert operations.
   2. Bottom-up $O(n)$ approach using a modified heapify operations.
**Example:** Build a minmax heap for the set $S = \{6, 8, 5, 2, 7, 8, 2, 9, 12, 1\}$ using a modified bottom-up $O(n)$ buildMinMaxHeap operation.

\begin{center}
\begin{tikzpicture}[level distance=1.5cm, level 1/.style={sibling distance=4.5cm}, level 2/.style={sibling distance=2.5cm}, level 3/.style={sibling distance=1.5cm}]
    \node (root) {6} \nodepart{right} \text{min-level}
    child {node {8} \nodepart{right} \text{max-level}
        child {node {2} \nodepart{right} \text{min-level}
            child {node {9} \nodepart{right} \text{max-level}}
            child {node {12} \nodepart{right} \text{max-level}}
        } child {node {7}}
    } child {node {5} \nodepart{right} \text{max-level}
        child {node {8} \nodepart{right} \text{min-level}
            child {node {2}}
        } child {node {1}}
    }
\end{tikzpicture}
\end{center}

Considering delete and then re-insert the node as in deleteMin or deleteMax operations:

\begin{center}
\begin{tikzpicture}[level distance=1.5cm, level 1/.style={sibling distance=4.5cm}, level 2/.style={sibling distance=2.5cm}, level 3/.style={sibling distance=1.5cm}]
    \node (root) {6} \nodepart{right} \text{min-level}
    child {node {8} \nodepart{right} \text{max-level}
        child {node {2} \nodepart{right} \text{min-level}
            child {node {9}}
            child {node {12}}
        } child {node {1}}
    } child {node {5} \nodepart{right} \text{max-level}
        child {node {8}} child {node {2}}
    }
\end{tikzpicture}
\end{center}
HW:

1. Build a minmax heap by inserting <3, 8, 10, 2, 7, 24, 5, 12, 26, 1, 28> into an initially empty heap.
2. Build a minmax heap for {13, 8, 10, 22, 7, 24, 5, 12, 26, 1, 28, 6} using the bottom-up O(n) approach.
3. Repeat (1) & (2) by building a maxmin heap.
4. Given a minmax heap [1, 28, 24, 12, 6, 10, 15, 18, 22, 7, 8]. Perform deletemax until the heap is empty.
5. Given a minmax heap [1, 28, 24, 12, 6, 10, 15, 18, 22, 7, 8]. Perform deletemax until the heap is empty.

9/30/2014