HEAPS

Heaps in Theory

- uses a graphical tree to represent an unsorted array
- the tree
  - is a RBT (Regular Binary Tree)
    - so only 2 children
  - is completed from the top down, left to right (called complete)
- each node in the tree represented in the corresponding array
- cannot have duplicates
- items are added to the array in order (or make a COMPLETE tree)
- order of inputs does have an effect on the overall order of the heap

<table>
<thead>
<tr>
<th>Tree Representation</th>
<th>Array Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image.png" alt="Tree Diagram" /></td>
<td><img src="image.png" alt="Array Representation" /></td>
</tr>
</tbody>
</table>

// JUST A GRAPHICAL REPRESENTATION!!

// Draw the value in the elements of the tree from the array representation

Was does complete mean?
Minimum Binary Heap

- same constructions as a heap, but the minimum value of the entire tree is stored at the root
- the further down we go in the min heap, the value increases
  - parent will ALWAYS be less than or equal in value than the kids
  - this is called partial ordering

The Min Heap Structure

Notice 4 is the smallest value so far in this heap
Anything below the parent (no matter where) is >= than the parent
Notice the max value will be SOMEWHERE near the bottom
Initial class setup – BinaryHeap

- code given uses an array
  - default size is 10
  - calls buildHeap() just to do that

```java
MinBH Construct(or)

/**
 * Construct the binary heap given an array of items.
 */
public BinaryHeap( AnyType [ ] items )
{
    currentSize = items.length;
    array = (AnyType[]) new Comparable[ (currentSize + 2) * 11 / 10 ];

    int i = 1;
    for( AnyType item : items )
        { array[ i++ ] = item; }
    buildHeap( );
}
```
Determining the relationships using code/array

- Determining who is parent/child of a certain node is easy!!
  - using array notation and structure!!

![Tree Representation]

// JUST A GRAPHICAL REPRESENTATION!!!

**Remember this is using an ARRAY representation!!!**

<table>
<thead>
<tr>
<th>To Find</th>
<th>Formula</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent index</td>
<td>( \text{floor}((\text{index})/2) )</td>
<td>([6]/2 = 3) 6’s Parent is 3</td>
</tr>
<tr>
<td>Left Child index</td>
<td>2(index)</td>
<td>(2*[3] = 6) 3’s Left Child is 6</td>
</tr>
<tr>
<td>Right Child index</td>
<td>2(index) + 1</td>
<td>(2*[3] + 1 = 7) 3’s Right Child is 7</td>
</tr>
<tr>
<td>9’s Parent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2’s Left Child</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4’s Parent</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Building and Inserting into a Min Heap**

- notice I do have to **specify** Minimum Binary Heap
- algorithm
  - place new node at END of array
    - next available complete spot in BT
  - at end, could be in wrong order (parent is larger!)
    - continuously swap with parent going up the tree until parent < new node
    - this is called sift up or percolate up
  - notice that “lighter” values do bubble up, (maybe not to the root), but are in a higher position

**Inserting into an establish Min Heap**

Where will the next value be first placed, no matter the value??
Inserting a 6

Start checking position!! (Check immediate Parent 17)

Swap since Parent is > new node!!
I will try this one: (answer on next page)

64, 12, 35, 28, 74, 24, 59
Answer to in-class example

Try these on your own, insert in the order given:
1. 56, 43, 12, 67, 92, 4, 87, 53, 44, 93
2. 61, 23, 57, 12, 68, 24, 14, 96, 75, 63
3. If I added 4 to #2, how many swaps would take place?

Answers:

**Insert – the function**

- notice it checks the size of the array first
  - adds more if not enough
- temporarily spaces our value in [0]
- `<0` is not the value, but if there is a parent that is greater, then keep swapping

---

### Array version of Insert for MinBH

```java
/**
 * Insert into the priority queue, maintaining heap order.
 * Duplicates are allowed.
 * @param x the item to insert.
 */
public void insert( AnyType x )
{
    // check if size of array is enough to hold new node
    if( currentSize == array.length - 1 )
        { enlargeArray( array.length * 2 + 1 ); }

    // Percolate up
    int hole = ++currentSize;
    for( array[ 0 ] = x; x.compareTo( array[ hole / 2 ] ) < 0; hole /= 2 )
        { array[ hole ] = array[ hole / 2 ]; }

    // now put our new value into the right place
    array[ hole ] = x;
}
```

Why is it /= 2?
Finding the minimum in a MinBH

- super easy!
- minimum value will ALWAYS be the root
  - if everything percolated correctly
  - [1] not [0]!!

Min is always at the top of a MinBH

```java
/**
 * Find the smallest item in the priority queue.
 * @return the smallest item, or throw an UnderflowException if empty.
 */
public AnyType findMin()
{
    if( isEmpty() )
    {
        throw new UnderflowException();
    }
    return array[ 1 ];
}
```
Deleting in a MinBH

- deletion is **ONLY** authorized for the MINIMUM value
  - not any other value
- we are not deleting the node, just replace the data inside
- now replaced with the NEXT lowest value
  - which SHOULD be close to the top of the tree
- tree must maintain it’s shape
- but we will delete the LAST complete node in the tree since
  - since now it will be empty

Deleting a node, and re-heaping

Deleting Min value
Replacing root with LAST value in tree

Comparing 91 (root) with whatever child is smallest
Percolating Down – Match

Percolating Down – Swap
Done since no more immediate nodes to compare to.

Delete the NEXT node using the result above.

After you’re done, click the link below for your answer: http://userpages.umbc.edu/~slupoli/notes/DataStructures/videos/Heaps/Deleting%20from%20Heap-%20Exercise.html
Delete – the function(s)

- deleteMin() and percolateDown()
  - deleteMin is the bootstrap to get things started
  - percolateDown is iterative in comparing and swapping
    - also called heapify

deleteMin() function
/**
 * Remove the smallest item from the priority queue.
 * @return the smallest item, or throw an UnderflowException if empty.
 */
public AnyType deleteMin( )
{
    if( isEmpty( ) ) { throw new UnderflowException( ); }

    AnyType minItem = findMin( );
    array[ 1 ] = array[ currentSize-- ];
    percolateDown( 1 );

    return minItem;
}

percolateDown() function
/**
 * Internal method to percolate down in the heap.
 * @param hole the index at which the percolate begins.
 */
private void percolateDown( int hole )
{
    int child;
    AnyType tmp = array[ hole ];

    for( ; hole * 2 <= currentSize; hole = child )
    {
        child = hole * 2;
        if( child != currentSize &&
            array[ child + 1 ].compareTo( array[ child ] ) < 0 )
            child++;
        if( array[ child ].compareTo( tmp ) < 0 )
            { array[ hole ] = array[ child ]; } 
        else { break; }
    }

    array[ hole ] = tmp;
Performance

- construction $O(n)$
  - even if data is out of order, we place in heap with partial ordering
  - still stored in a simple array!!
- findMin $O(1)$
- insert $O(\log n)$
- deleteMin $O(\log n)$

Heap Construction – the function

- lays all items into array first, no matter order in construction
  - done in constructor
- then “builds the heap” (sorts, partially) in buildHeap
  - notice that buildHeap uses percolateDown starting at middle of the array
  - this is enough to have the real minimum value “rise” to the top of the heap
- neither of these functions are recursive

```
/**
 * Establish heap order property from an arbitrary arrangement of items. Runs in linear time.
 */
private void buildHeap()
{
    for (int i = currentSize / 2; i > 0; i--)
        percolateDown(i);
}
```
Sorting a Heap – MinBH

- given a list of n values, we can build and sort in O(n log n)
  - insert *random* values = O(n)
  - heapify = O(n)
  - repeatedly delete min and re-heapify O(log n) * n times
- heapify
  - re-ordering the values so Parent is $\leq$ it’s kids in a MinBH
- delete
  - retrieves the CURRENT minimum node in a MinBH
    - value is saved in another array
  - *automatically calls percolateDown()*
- this looped OVER and OVER will return a sorted list of items
- this means we will need another array of the same size, just to hold the cast offs
  - UNLESS, we store the values casted back in the deleted code’s position
  - but this will have everything backwards!!

Using a MinBH to sort

```
    0026
   /   \
  0031 0058
 /      /  \
0041   0053 0059
       /    /  \
      /    /   0097
```

```
<table>
<thead>
<tr>
<th>INF</th>
<th>0026</th>
<th>0031</th>
<th>0058</th>
<th>0041</th>
<th>0053</th>
<th>0059</th>
<th>0097</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
</tbody>
</table>
```
Perform the next two deletions on the heap above. Make sure to draw the tree AND the array.
Sorting a Heap – MaxBH

- here we avoid the “backward” issue
- now the parent is ≥ its kids
  - so HIGHEST value is at the top of the heap

### Max Heap Example

![Max Heap Example Diagram](image)

### Deleting 97 (current Max), now heapify

![Deleting 97 Diagram](image)

Perform the next two deletions on the heap above

In its entirety
MinBH (using arrays) shortcomings

- sorting, wrong order
- merge
  - merging two arrays, no real shortcut
  - so \( n_1 + n_2 \)

Leftist Min Heaps

- uses a BT!!
- merging heaps is much easier and faster
  - may use already established links to merge with a new node
    - why so much faster
  - because we are using Binary Trees!!
- values STILL obey a heap order (partially ordered)
- uses a null path length to maintain the structure (covered later)
  - the null path of a node’s left child is \( \geq \) null path of the right child
- at every node, the shortest path to a non-full node is along the rightmost path
- this overall ADT supports
  - \( \text{findMin} = O(1) \)
  - \( \text{deleteMin} = O(\log n) \)
  - \( \text{insert} = O(\log n) \)
  - \( \text{construct} = O(n) \)
  - \( \text{merge} = O(\log n) \)

Example of a Leftist Heap

![Diagram of a Leftist Heap](image-url)
Null Path Length (npl)

- length of **shortest** path from current node (X) to a node **without** 2 children
  - value is store IN the node itself
- leafs = 0
- nodes with only 1 child = 0

### Determining the npl for a node

#### Determine the npls for the trees below. Are the left-ist?
The Leftist Node

- the node will have many data members this time
  - links (left and right)
  - element (data)
  - npl
- by default, the LeftistHeap sets and empty one as the root

The leftist Node Class and Code

```java
private LeftistNode<AnyType> root;    // root

private static class LeftistNode<AnyType>
{

  // Constructors
  LeftistNode( AnyType theElement )
  {
    this( theElement, null, null );
  }

  LeftistNode( AnyType theElement, LeftistNode<AnyType> lt, LeftistNode<AnyType> rt )
  {
    element = theElement;
    left     = lt;
    right    = rt;
    npl      = 0;
  }

  AnyType    element;    // The data in the node
  LeftistNode<AnyType> left;  // Left child
  LeftistNode<AnyType> right; // Right child
  int        npl;        // null path length
}
```
Building a Left-ist Heap

- value of node STILL matters, lowest value will be root, so still a min Heap
- data entered is random
- uses CURRENT npl of a node to determine where the next node will be placed
- algorithm
  - add new node to right-side of tree, in order
  - if new node is to be inserted as a parent (parent > children),
    - make new node parent
    - link children to it
    - link grandparent down to new node
  - if leaf, attach to right of parent
  - if no left sibling, push to left (hence left-ist)
    - why?? (answer in a second)
  - else left node is present, leave at right child
  - update all ancestors’ npls
  - check each time that all nodes left npl < right npls
    - if not, swap children or node where this condition exists
- this is really using heaps and links!!

<table>
<thead>
<tr>
<th>Building a leftist Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>21, 14, 17, 10, 3, 23, 26, 8</td>
</tr>
</tbody>
</table>

**inserting 14**

<table>
<thead>
<tr>
<th>Building a leftist Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>21, 14, 17, 10, 3, 23, 26, 8</td>
</tr>
</tbody>
</table>

**inserting 17**

<table>
<thead>
<tr>
<th>Building a leftist Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>21, 14, 17, 10, 3, 23, 26, 8</td>
</tr>
</tbody>
</table>

24
Why can we NOT have this??

• we can have it where a node’s left npl is greater than it’s right npl
  ○ simply, we swap children

Swapping children to save a leftist tree

Just inserted 52 into the leftist heap
Try creating these leftist heaps n your own:

75, 91, 97, 9, 39, 87, 34, 8, 86, 58
24, 80, 98, 30, 77, 35, 65, 2, 48, 92, 18, 37, 67, 96
71, 4, 13, 73, 52, 20, 50, 63, 85, 23, 1, 44, 32, 53, 14, 17, 82, 76, 27, 83, 11, 81, 90, 62

Answers:
Inserting – the function

• in the code, adding a single node is treated at merging a heap (just one node) with an established heap’s root
  ○ and work from that root as we just went over
• we will go over merging whole heaps momentarily

```java
/**
 * Insert into the priority queue, maintaining heap order.
 * @param x the item to insert.
 */
public void insert( AnyType x )
{
    root = merge( new LeftistNode<>( x ), root );
}
```
**Merging Left-ist Heaps**

- the heaps we are about to merge must be left-ist
- at end we will get a heap that is
  - a min-heap
  - left-ist
- algorithm
  - Start at the (sub) root, and finalize the node AND LEFT with the smallest value
  - REPEADLY, until no lists left unmerged.
    - Start at the **rightmost** root of the sub-tree, and finalize the node AND LEFT with the **next** smallest value in leftist lists.
    - Add to RIGHT of finalized tree.
  - Verify that it is a Min Heap!! (Parent < Children)
  - Verify a leftist heap! (left npl <= right npl)
    - if not, swap troubled node with sibling

I will try:

```
Initial Left-ist Heaps
```

```
<table>
<thead>
<tr>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>25</td>
</tr>
<tr>
<td>43</td>
</tr>
</tbody>
</table>
```

Start at the root, and finalize the node AND LEFT with the smallest value
<table>
<thead>
<tr>
<th>finalized</th>
<th>option 1</th>
<th>option 2</th>
</tr>
</thead>
</table>

Start at the root of the sub-tree, and finalize the node AND LEFT with the **next** smallest value. Add to RIGHT of finalized tree.

Start at the root of the sub-tree, and finalize the node AND LEFT with the **next** smallest value. Add to RIGHT of finalized tree.

Start at the root of the sub-tree, and finalize the node AND LEFT with the **next** smallest value. Add to RIGHT of finalized tree.

Verify that it is a Min Heap!! (Parent < Children)

*Yup*
Verify a leftist heap! (left npl <= right npl)

Switch problem node with sibling. (Start from root, work way to bottom). All links stay in same direction down.
Try these:

<table>
<thead>
<tr>
<th>#1</th>
<th><img src="image" alt="Two heaps" /></th>
</tr>
</thead>
<tbody>
<tr>
<td>DO THIS ONE AND STOP!!! Will go over together.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#2</th>
<th><img src="image" alt="Heaps" /></th>
</tr>
</thead>
<tbody>
<tr>
<td>a lot of work on this one</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#3</th>
<th><img src="image" alt="Heaps" /></th>
</tr>
</thead>
<tbody>
<tr>
<td>I know!! Heap is a MAX heap!! Try it anyway! (max first!)</td>
<td></td>
</tr>
</tbody>
</table>
Merging – the function

- notice it is recursive!
- merge()
  - version 1 – copy rhs to root
  - version 2 - is the function to set up the order between left and right heaps
- merge1() is the function to actually do the linking and swapping if left npl > right npl
  - notice npl is a private variable

Merging Heaps

```java
/**
 * Merge rhs into the priority queue.
 * rhs becomes empty. rhs must be different from this.
 * @param rhs the other leftist heap.
 */
public void merge( LeftistHeap<AnyType> rhs )
{
    if( this == rhs ) // Avoid aliasing problems
        return;

    root = merge( root, rhs.root );
    rhs.root = null;
}

/**
 * Internal method to merge two roots.
 * Deals with deviant cases and calls recursive merge1.
 */
private LeftistNode<AnyType> merge( LeftistNode<AnyType> h1,
                                     LeftistNode<AnyType> h2 )
{
    if( h1 == null )
        return h2;
    if( h2 == null )
        return h1;
    if( h1.element.compareTo( h2.element ) < 0 )
        return merge1( h1, h2 );
    else
        return merge1( h2, h1 );
}

/**
 */
```
* Internal method to merge two roots.
* Assumes trees are not empty, and h1's root contains smallest item.
* /

```java
private LeftistNode<AnyType> merge1( LeftistNode<AnyType> h1, LeftistNode<AnyType> h2 )
{
    if( h1.left == null ) // Single node
        h1.left = h2; // Other fields in h1 already accurate
    else
    {
        h1.right = merge( h1.right, h2 );
        if( h1.left.npl < h1.right.npl )
            swapChildren( h1 );
        h1.npl = h1.right.npl + 1;
    }
    return h1;
}

/**
* Swaps t's two children.
*/
private static <AnyType> void swapChildren( LeftistNode<AnyType> t )
{
    LeftistNode<AnyType> tmp = t.left;
    t.left = t.right;
    t.right = tmp;
}
```
So why did we do this?

• fast!
  o merge with two trees of size n
    ▪ O(log n), we are not creating a totally new tree!!
    ▪ some was used as the LEFT side!
  o inserting into a left-ist heap
    ▪ O(log n)
    ▪ same as before with a regular heap
  o deleteMin with heap size n
    ▪ O(log n)
    ▪ remove and return root (minimum value)
    ▪ merge left and right subtrees

• real life application
  o priority queue
    ▪ homogenous collection of comparable items
    ▪ smaller value means higher priority
Answers:

**Inserting into a Heap Exercise #1**

```
<table>
<thead>
<tr>
<th>004</th>
<th>0044</th>
<th>0012</th>
</tr>
</thead>
<tbody>
<tr>
<td>0063</td>
<td>0002</td>
<td>0043</td>
</tr>
<tr>
<td>0087</td>
<td>0087</td>
<td>0068</td>
</tr>
<tr>
<td>INF</td>
<td>0004</td>
<td>0044</td>
</tr>
</tbody>
</table>
```

```
0067 0056 0093
```

```
0004
  0044
    0063
      0067
      0056
      0093
    0092
      0043
      0087
  0012
```

**Inserting into a Heap Exercise #2**

```
<table>
<thead>
<tr>
<th>0012</th>
<th>0023</th>
</tr>
</thead>
<tbody>
<tr>
<td>0014</td>
<td>0081</td>
</tr>
<tr>
<td>0083</td>
<td>0067</td>
</tr>
<tr>
<td>0067</td>
<td>0068</td>
</tr>
<tr>
<td>0024</td>
<td></td>
</tr>
</tbody>
</table>
```

```
0096 0075 0088
```

```
0012
  0023
    0081
      0096
      0075
      0088
    0083
      0067
      0068
  0014
```


Merging Left-ist Heaps #1
Merging Left-ist Heaps #2

\[ H = H^1 + H^2 \]
Merging Left-ist Heaps #3

Diagram of merging left-ist heaps with numbers 12, 8, 10, 13, 5, 9, 6, 4, and 2.
Sources:

In General
http://courses.cs.washington.edu/courses/cse332/13wi/lectures/cse332-13wi-lec04-BinMinHeaps-6up.pdf

Maximum HeapSort
http://www.cs.usfca.edu/~galles/visualization/HeapSort.html

Show and Tell Heap building
http://www.cs.usfca.edu/~galles/visualization/Heap.html

Random Heapsort
http://www.cse.iitk.ac.in/users/dsrkg/cs210/applets/sortingII/heapSort/heapSort.html
http://nova.umuc.edu/~jarc/idsv/lesson3.html

Building a Leftist Heap
http://www.cs.usfca.edu/~galles/visualization/LeftistHeap.html