**Heap**

- Like a binary search tree, a heap is a binary tree in which the data entries can be compared using total order semantics
- Defining qualities of a heap:
  - the data entry contained in any node is greater than or equal to the data entry contained in either of its children
  - the tree is always complete

**Heap Example**

```
       45
      /   \
  32     41
   /\   / \   /\  \
 19 7 28 40
 / \ / \ / \ / \
10 12
```

**Heap Applications**

- Heap can be used to implement a sorting algorithm called heapsort
- Heap is also a handy way to implement a priority queue -- we’ll use this application to illustrate how heaps work

**Heap Implementation**

- Because a heap is by definition a complete tree, it is easy to use an array-based implementation
- Heap operations `reHeapUp` and `reHeapDown` are used to maintain the other defining quality of a heap -- each node’s data entry >= data entries of its children

**ReHeapUp operation**

```
       45
      /   \
  32     41
   /\   / \   /\  \
 19 7 28 40
 / \ / \ / \ / \
10 12 35
```

This maintains completeness of tree, but can violate the other defining characteristic of a heap
ReHeapUp operation

ReHeapUp restores the second heap condition: a parent node's data is always greater than or equal to the data in either of its child nodes.

The operation is accomplished by swapping the new node's data entry with that of its parent.

ReHeapDown operation

In a priority queue, the highest-priority item is always dequeued first -- this item would be the top item of the heap.

Since this is going to be an array implementation, we'll perform the usual trick of swapping the last array entry with the first, so we can minimize the amount of copying to be done.

ReHeapDown operation

Now we can effectively remove the item by simply diminishing the "used" portion of our array by 1 (already done here).

We solve this problem, as before, by swapping data between nodes. In this case, we swap the parent node's data with the largest data entry of the two children, continuing until the heap is restored.

Example: Priority Queue with heap implementation

- Priority queue has all operations associated with a queue: enqueue, dequeue and helper functions.
- Heap is a good fit for priority queue because highest-priority item should always be dequeued first, and this item would always be found at the top of a heap.

Code for Priority Queue

```java
public class PQueue {
    Heap queue;
    public void enqueue (int item) {
        queue.addEntry(item);
    }
    public PQueue (int cap) {
        queue = new Heap(cap);
    }
    public int deQueue () {
        return queue.getTop();
    }
}
```

Code for Heap class

```java
public class Heap {
    int [] data;
    int numItems;
    public Heap (int size) {
        numItems = 0;
        data = new int [size];
    }
}
```
Code for Heap class

```java
public int parent (int n) {
    return (n-1)/2;
}
public int leftChild (int n) {
    return 2*n + 1;
}
public int rightChild (int n) {
    return 2*n + 2;
}
```

Code for Heap class

```java
public void addEntry (int entry) {
    if (numItems < data.length) {
        data[numItems] = entry;
        reHeapUp(numItems);
        numItems++;
    }
}
```

Code for Heap class

```java
public void reHeapUp(int n) {
    int x = n;
    while (x>0 && data[x] > data[parent(x)]) {
        int tmp = data[x];
        data[x] = data[parent(x)];
        data[parent(x)] = tmp;
        x=parent(x);
    }
}
```

Code for Heap class

```java
public int getTop () {
    int value=data[0]; // save return value
    numItems--;
    data[0]=data[numItems]; // swap top & bottom
    reHeapDown(numItems); // restore heap
    return value;
}
```

Code for Heap class

```java
public void reHeapDown (int n) {
    int current = 0, bigChild;
    boolean heapOK = false;
    while (!heapOK && (leftChild(current) < n)) {
        if (rightChild(current) >= n)
            bigChild = leftChild(current);
        else if (data[leftChild(current)] > data[rightChild(current)])
            bigChild = leftChild(current);
        // continued next slide
    }
}
```
A good sorting algorithm is hard to find ...

- Quadratic sorting algorithms (with running times of $O(N^2)$), such as Selectionsort & Insertionsort are unacceptable for sorting large data sets.
- Mergesort and Quicksort are both better alternatives, but each has its own set of problems.

Heapsort to the rescue!

- Combines the time efficiency of Mergesort with the storage efficiency of Quicksort.
- Uses an element interchange algorithm similar to Selectionsort.
- Works by transforming the array to be sorted into a heap.

How Heapsort Works

- Begin with an array of values to be sorted.
- Heapsort algorithm treats the array as if it were a complete binary tree.
- Values are rearranged so that the other heap condition is met: each parent node is greater than or equal to its children.

Heapsort in Action

Original array: 42, 19, 33, 8, 12, 97, 54, 85, 29, 60, 26, 71

...arranged as binary tree (but not yet heap)

Heapsort in Action

Rearranged array: 97, 85, 71, 29, 60, 42, 54, 8, 19, 12, 26, 33

...rearranged as heap
Heapsort in Action

The goal of the sort algorithm is to arrange entries in order, smallest to largest. Since the root element is by definition the largest element in a heap, that largest element can be placed in its final position by simply swapping it with whatever element happens to be in the last position:

\[
\begin{array}{cccccccccc}
33 & 85 & 71 & 29 & 60 & 42 & 54 & 8 & 19 & 12 & 26 & 97
\end{array}
\]

As a result, the shaded area can now be designated the sorted side of the array, and the unsorted side is almost, but not quite, a heap.

Heapsort in Action

Again, the unsorted side is almost a heap, and must be restored to heap condition - this is accomplished the same way as before:

\[
\begin{array}{cccccccccc}
71 & 60 & 54 & 29 & 33 & 42 & 26 & 8 & 19 & 12 & 85 & 97
\end{array}
\]

The process continues: root is swapped with the last value and marked sorted; the unsorted portion is restored to heap condition

\[
\begin{array}{cccccccccc}
60 & 33 & 54 & 29 & 19 & 42 & 26 & 8 & 12 & 71 & 85 & 97
\end{array}
\]

Pseudocode for Heapsort

- Convert array of n elements into heap
- Set index of unsorted to n (unsorted = n)
- while (unsorted > 1)
  - unsorted--;
  - swap (array[0], array[unsorted]);
  - reHeapDown

Heapsort in Action

And so on ...

\[
\begin{array}{cccccccccc}
54 & 33 & 42 & 29 & 19 & 12 & 26 & 8 & 60 & 71 & 85 & 97
\end{array}
\]

And so forth ...

\[
\begin{array}{cccccccccc}
42 & 33 & 26 & 29 & 19 & 12 & 8 & 54 & 60 & 71 & 85 & 97
\end{array}
\]

Etc ...

\[
\begin{array}{cccccccccc}
8 & 12 & 19 & 26 & 29 & 33 & 42 & 54 & 60 & 71 & 85 & 97
\end{array}
\]

Tada!

makeHeap function

- Two common ways to build initial heap
- First method builds heap one element at a time, starting at front of array
- Uses reHeapUp process (last seen in priority queue implementation) to enforce heap condition - children are <= parents
Alternative makeHeap method

- Uses a function (subHeap) that creates a heap from a subtree of the complete binary tree
- The function takes three arguments: an array of items to be sorted, the size of the array (n), and a number representing the index of a subtree.
- Function subHeap is called within a loop in the makeHeap function, as follows:
  
  ```
  for (int x = (n/2); x>0; x--)
      subHeap(data, x-1, n);
  ```

Alternative makeHeap in Action

Original array

```
42 19 33 8 12 97 54 85 29 60 26 71
```

First call to subHeap, in the given example:

```
subHeap(data, 5, 12);
```

Algorithm examines subtree to determine if any change is necessary; since the only child of element 5 (97) is element 11 (71), no change from this call.

Second call to subHeap

```
42 19 33 8 60 97 54 85 29 12 26 71
```

The second call to subHeap looks at the subtree with root index 4
The children of this node are found at indexes 9 and 10
Both children are greater than the root node, so the root node’s data entry is swapped with the data entry of the larger child

Third call to subHeap

```
42 19 33 85 60 97 54 8 29 12 26 71
```

Continuing to work backward through the array, the next root index is 3
Again, the children of this node are examined, and the root data entry is swapped with the larger of its two children

The process continues ...

```
42 19 97 85 60 71 54 8 29 12 26 33
```

In this case, a further swap is necessary because the data entry’s new position has a child with a larger data entry

The process continues ...

```
42 85 97 29 60 71 54 8 19 12 26 33
```

Again, a further swap operation is necessary
**Last call to subHeap**

97 85 71 29 60 42 54 8 19 12 26 33

We now have a heap:

```
  97
 /    \
/      \
60      85
|      |
|      |
29      71
|      |
|      |
60      42
|      |
|      |
34      54
|      |
|      |
8       19
|      |
|      |
12      26
|      |
|      |
33      33
```

**ReHeapDown function**

- Both the original HeapSort function and the subHeap function will call reHeapDown to rearrange the array
- The reHeapDown function swaps an out-of-place data entry with the larger of its children

---

**Time analysis for HeapSort**

- The time required for a HeapSort is the sum of the time required for the major operations:
  - building the initial heap: $O(N \log N)$
  - removing items from the heap: $O(N \log N)$
- So total time required is $O(2N \log N)$, but since constants don’t count, the worst-case (and average case) for HeapSort is $O(N \log N)$