Heaps

- Heaps
- Priority Queue Revisit
- HeapSort
Heaps

- A heap is a complete binary tree in which the nodes are organized based on their data values.

- For each non-leaf node $V$,
  - **max-heap**: the value in $V$ is greater than or equal to the value of its two children.
  - **min-heap**: the value in $V$ is smaller than or equal to the value of its two children.
Heap Properties

- **heap order property** – how the nodes in a heap or arranged.
- **heap shape property** – as a complete binary tree.

(a) min-heap

(b) max-heap
Heap Operations

- The heap is a specialized structure with limited operations.
  - Heap() construct an empty Heap
  - insert() insert a new element into the heap.
  - removeMin() or removeMax() remove the element from root node and return this min/max value.

The following slides explore operations for use with a max-heap. Min-heap operations share the same ideas.
Heap Insertions

- When an element is inserted into a heap, both properties must be maintained.
  - Example: add 90 to the max-heap.
  - There are only 2 places where 90 can be inserted.
Heap Insertions

- To properly insert an element into a heap involves several steps.
  - Create a new node for value.
Heap Insertions

- Link the node in as the last child in the complete tree.
Heap Insertions

- To restore the heap order property, the new element has to move up along its path:
  - the data is swapped with its parent's data.
  - **sift-up**
Heap Insertions

- 90 must move up another level since 84 is smaller.
- After which, it is in its correct position.
Heap Insert Example

- Insert 41 into the heap from the previous slide.
Heap Extractions

- When an element is extracted from the heap, it can only come from the root node.
  - Both heap properties must be maintained.
Heap Extractions

- To restore the tree to a heap:
  - another value will have to take the place of the extracted value in the root node.
  - a node has to be removed from the tree.
Heap Extractions

- There is only one node that can be removed and still maintain the heap shape property.
  - Copy the data from the last child node to the root.
  - Delete the last child node.
Heap Extractions

- To maintain the heap order property, the new root value has to be \textit{sifted-down}.

- Swap with the larger of the two children.
Heap Extractions

- The shifting continues until the value is placed into a leaf node, or it is larger than its children.
Heap Extractions

- After being swapped with 23, value 12 will be in a node that maintains the heap order property.
Heap Implementation

- While a heap is a binary tree, it's seldom implemented as a dynamically linked structure.
  - Use a sequence to physically store the nodes.
Heap – Node Access

- The complete tree will never contain “holes”.
  - The root will always be at position 0.
  - Its two children will always occupy positions 1 and 2.
  - The children of any node will always occupy the same position.
Heap – Node Access

- Given the array index \( i \)
  
  \[
  \text{parent} = \frac{(i-1)}{2} \\
  \text{left} = 2 \times i + 1 \\
  \text{right} = 2 \times i + 2
  \]

- A child link is null if the index is out of range.
Heap – Class Definition

class MaxHeap:
    def __init__(self, maxSize):
        self._elements = Array(maxSize)
        self._count = 0

    def __len__(self):
        return self._count

    def capacity(self):
        return len(self._elements)

    def _swap(self, item1, item2):
        tmp = item1
        item1 = item2
        item2 = tmp

    # ...
class MaxHeap :
    # ...
    def add( self, value ):
        assert self._count < self.capacity(),
            "Cannot add to a full heap."
        # Add the new value to the end of the list.
        self._elements[ self._count ] = value
        self._count += 1
        # Sift the new value up the tree.
        self._siftUp( self._count - 1 )

    def _siftUp( self, ndx ):
        if ndx > 0 :
            parent = (ndx - 1)// 2
            if self._elements[ndx] > self._elements[parent] :
                self._swap(self._elements[ndx], self._elements[parent] )
                self._siftUp( parent )

arrayheap.py
Heap – Class Definition

class MaxHeap:

# ...
def extract(self):
    assert self._count > 0,
    "Cannot extract from an empty heap."
    value = self._elements[0]
    self._count -= 1
    self._elements[0] = self._elements[self._count]
self._siftDown(0)
Heap – Class Definition

```python
class MaxHeap:
    # ...
    def _siftDown(self, ndx):
        left = 2 * ndx + 1
        right = 2 * ndx + 2
        largest = ndx
        if left < count and
            self._elements[left] >= self._elements[largest]:
            largest = left
        elif right < count and
            self._elements[right] >= self._elements[largest]:
            largest = right
        if largest != ndx:
            self._swap(self._elements[ndx], self._elements[largest])
            self._siftDown(largest)
```

arrayheap.py
Heap Example

- Physical view of adding value 90.
Heap Example

- Physical view of adding value 90.
Heap Analysis

- Assume a heap containing $n$ elements:
  - Insertion: $O(\log n)$
  - Extraction: $O(\log n)$
Priority Queue Revisited

- The unbounded priority queue does not place a restriction on the priority values.
  - Earlier we explored an implementation that used a linked list.
    - enqueue: $O(1)$
    - dequeue: $O(n)$
- Can we improve the efficiency using a heap?
Unbounded Priority Queue

- Use a min-heap:
  - Store both the priority and queue elements in each element of the heap.
  - The heap ordering is based on the priority.
# Priority Queue Comparison

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Worst Case</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Enqueue</td>
<td>Dequeue</td>
<td></td>
</tr>
<tr>
<td>Python List</td>
<td>O(n)</td>
<td>O(n)</td>
<td></td>
</tr>
<tr>
<td>Linked List</td>
<td>O(1)</td>
<td>O(n)</td>
<td></td>
</tr>
<tr>
<td>Heap (array)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td></td>
</tr>
<tr>
<td>Heap (list)</td>
<td>O(n)</td>
<td>O(n)</td>
<td></td>
</tr>
</tbody>
</table>
The Heapsort

- The simplicity and efficiency of the heap structure can be applied to the sorting problem.
  - Build a heap from a sequence of unsorted keys.
  - Extract the keys from the heap to create a sorted sequence.

- Very efficient: $O(n \log n)$
Heapsort Implementation

- A simple implementation is provided below.

```python
def simpleHeapSort( theSeq ):
    # Create an array-based max-heap.
    n = len(theSeq)
    heap = MaxHeap( n )

    # Build a max-heap from the list of values.
    for item in theSeq :
        heap.add( item )

    # Extract each value from the heap and store
    # them back into the list.
    for i in range( n-1, -1, -1 ) :
        theSeq[i] = heap.extract()
```
In-Place Heapsort

- The previous version required additional storage for the heap.
- The entire process can be done in-place within the original sequence.
  - Suppose we are given the following array

```
[10 51 2 18 4 31 13 5 23 64 29]
```
In-Place Heapsort

- The first step is to construct a max-heap.
  - The heap nodes occupy the array from front to back.
  - We can keep the heap elements in the front and those yet to be added at the back.
  - Keep track of where the heap ends.
In-Place Heapsort

- The first value in the array represents a max-heap of one element.
In-Place Heapsort

- The next value to be added, is the next in the array.
  - The value is added as a new leaf.
  - Then sifted up.
In-Place Heapsort
In-Place Heapsort

- The next step is to extract the values from the heap and build the sorted sequence.
  - When the root is extracted, the last child node is copied to the root.
  - The last child node is stored in the last element of the heap.
  - We can simply swap the two values.
In-Place Heapsort

(a) the original max-heap.

(b) swap the first and last items in the heap.

(c) remove the last item from the heap.

(d) sift the root value down the tree.
In-Place Heapsort

```
31 29 13 18 23 2 4 5 10 51 64
→
29 13 23 18 4 2 5 10 51 64
→
23 18 13 5 10 2 4 29 51 64
```

```
18
→
10
→
5 4 2
→
13
→
10
→
5 4
```

```
13 10 2 5 4 18 23 29 51 64
→
13 10 2 5 4 18 23 29 51 64
→
13 10 2 5 4 18 23 29 51 64
→
10 5 2 4
```

```
5 4 2
→
4
→
2
```

```
4 2 5 10 13 18 23 29 51 64
→
2 4 5 10 13 18 23 29 51 64
```

```
4
```

In-Place Heapsort Implementation

`siftUp()` and `siftDown()` are helper functions based on those used in the heap class.

```python
def heapsort(theSeq):
    n = len(theSeq)
    # Build a max-heap within the same array.
    for i in range(n):
        siftUp(theSeq, i)

    # Extract each value and rebuild the heap.
    for j in range(n-1, 0, -1):
        tmp = theSeq[j]
        theSeq[j] = theSeq[0]
        theSeq[0] = tmp
        siftDown(theSeq, j-1, 0)
```