Priority Queues (Heap)

CPE202

Priority Queue ADT

- An item whose priority is the highest will be removed first.
- Two kinds of priority queues
  - Min Priority Queue
    - smallest key is always at the front
  - Max Priority Queue
    - largest key is always at the front
- Items are stored in a data structure so that an item with the highest/lowest priority can be found in constant time (i.e. O(1)).

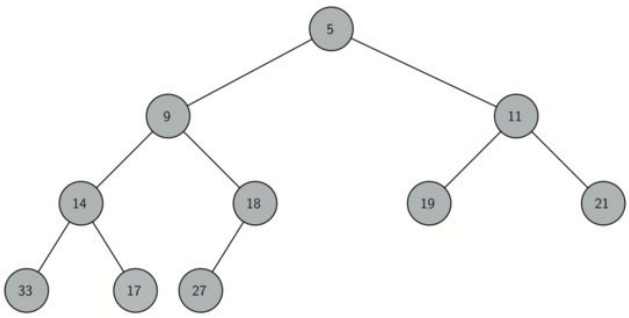
Priority Queue ADT

- Implemented with a data structure called binary heap.
- A priority queue implemented with a binary heap is conceptually represented by a complete binary tree but can be implemented with an array.

Binary Heap: The Structure Property

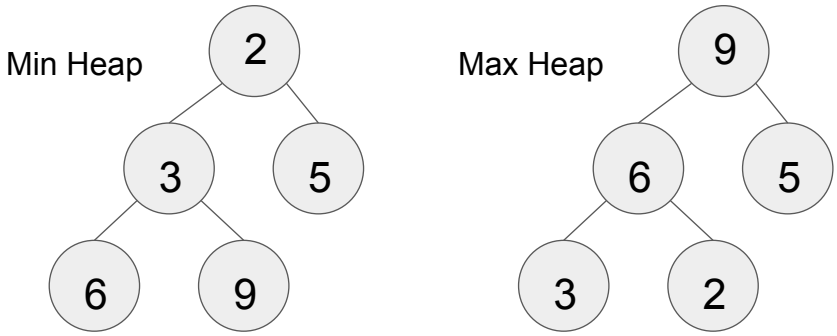
A complete binary tree

- All nodes are filled (have two child nodes)
- Except for the bottom level of the tree, which are filled in from left.



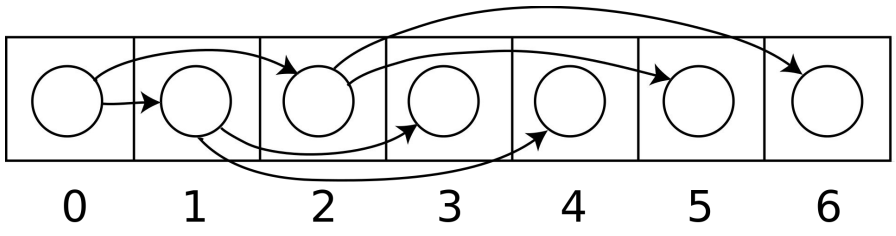
Binary Heap: The Heap Order Property

- Min Heap - The key of the parent node <= the node's key.
- Max Heap - The key of the parent node >= the node's key.



Internal Representation With Array

- Child nodes
  - Left : 2 * index + 1
  - Right : 2 * index + 2
- Parent node
  - (index - 1) // 2 for index > 0



Heap Operations

- insert
  - inserts an item at the end (bottom) of heap to maintain the structure property
  - Uses shift_up

Heap Operations

- dequeue (A.K.A. del_max or del_min)
  - Removes and returns an item from the top of heap
  - Move the last item in the heap to the top of heap to maintain the structure property
  - Uses shift_down

Heap Operations

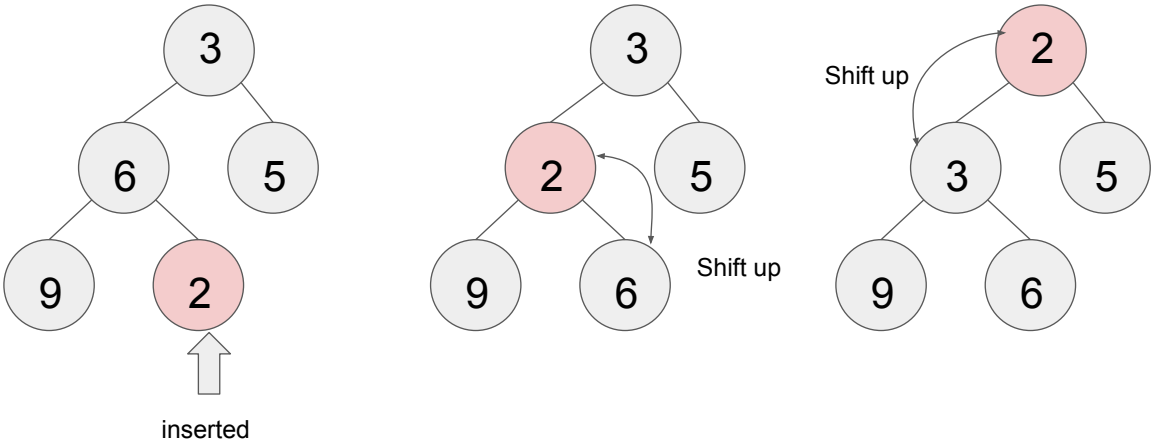
- heapify
  - Converts an array into a heap.
  - Uses shift_down

Heap Operations

- shift_up
  - Promotes an item up in heap to maintain the heap order property
- shift_down
  - Demotes an item down in heap to maintain the heap order property

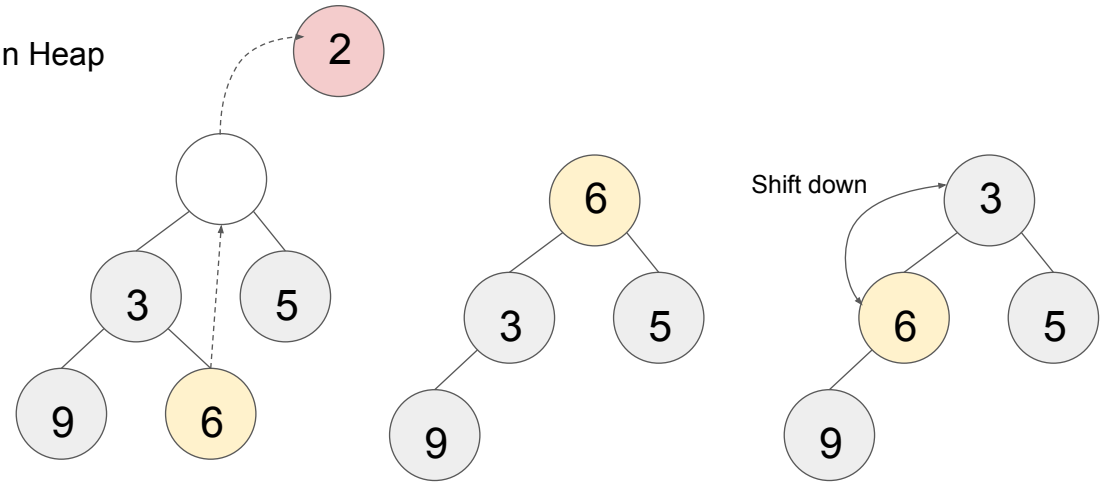
Insert O(log n)

Min Heap



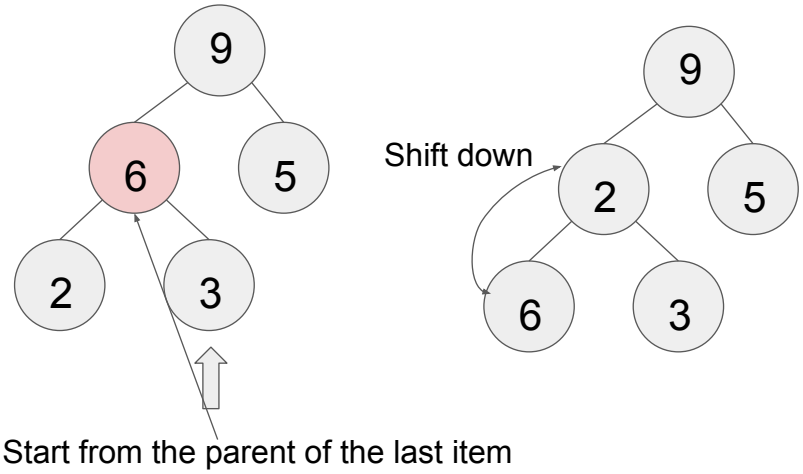
Dequeue O(log n)

Min Heap



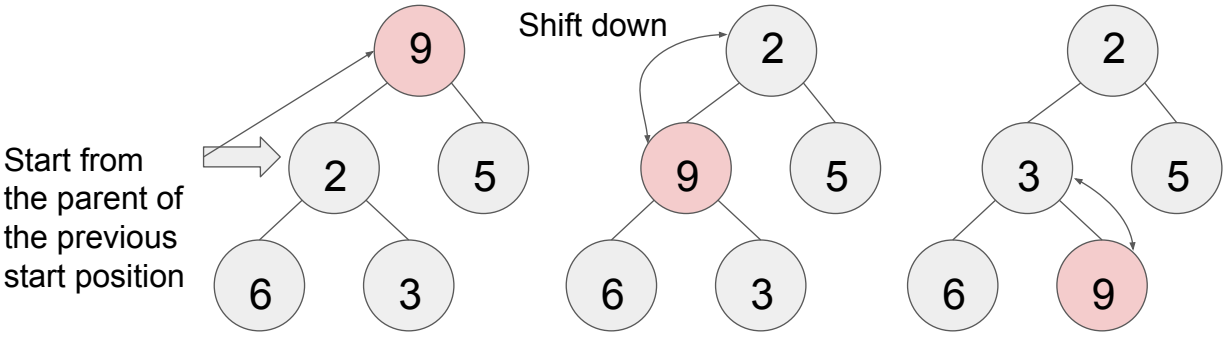
Heapify iteration #1

Min Heap



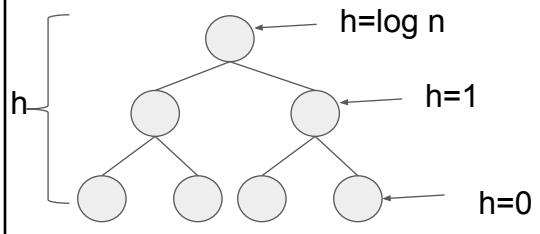
Heapify iteration #2

Min Heap



Heapify time complexity O(n)

The time complexity of Heapify



- At height (h) = 0, the number of nodes is at most $(n + 1)/ 2^{h+1}$
- At h = log n, # of nodes $\leqq (n + 1) / 2^{\log n+1}$
- At each height (h), the number of shift down operations is at most h.

# of steps =

$\sum_{h=0}^{\log n} ((n+1)/2^{h+1})*h \leqq (n+1) * \sum_{h=0}^{\log n} h/2^h$

$\sum_{h=0}^{\log n} h/2^h$ converges because the numerator (h) will be crushed by the denominator (2^h)as h grows.

Hence the time complexity is O(n).

Heap Sort

Selection Sort

- It does n loops and in each loop selecting the largest value takes n steps => $O(n^2)$

```
def selection_sort(arr):
    size = len(arr)
    for i in reversed(range(1, size)):
        max_idx = 0
        for j in range(1, i):
            if arr[j] > arr[max_idx]:
                max_idx = j
        arr[max_idx], arr[i] = arr[i], arr[max_idx]
    return arr
```

n iteration

Heap Sort

It is basically the selection sort, which does:

- While the unsorted part of the list is not empty
  a. Select the largest value from unsorted part of the list
  b. Prepend the value to the sorted part of the list

Heap Sort

Heap sort selects the largest value in O(log n) by using a better data structure, heap,
improving the time complexity to O(n * log n)

Heap Sort

1. Build a max heap from the list (No Copy. In-Memory.)
2. Divide the list into 2 sublists: unsorted and sorted sublist
   a. The unsorted sublist starts from index 0.
      i. This part is a max heap
      ii. It shrinks as items are removed from the heap
   b. The sorted sublist is initially empty and grows leftward from the end of the list.

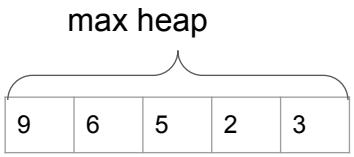
Heap Sort

3. While the size of the max heap is larger than 1:
   a. Dequeue an item from the max heap.
      i. Promote the last item in the max heap to the top.
      ii. Shift down the promoted item until it settles to a position which satisfies the max heap order.
   b. Add the dequeued item at the front of the sorted sublist.

9	2	5	6	3

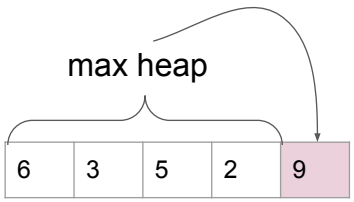
Heap Sort

3. While the size of the max heap is larger than 1:
   a. Dequeue an item from the max heap.
      i. Promote the last item in the max heap to the top.
      ii. Shift down the promoted item until it settles to a position which satisfies the max heap order.
   b. Add the dequeued item at the front of the sorted sublist.



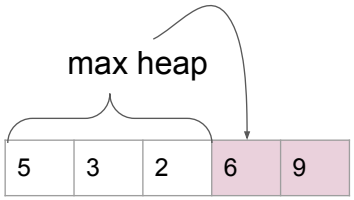
Heap Sort

3. While the size of the max heap is larger than 1:
   a. Dequeue an item from the max heap.
      i. Promote the last item in the max heap to the top.
      ii. Shift down the promoted item until it settles to a position which satisfies the max heap order.
   b. Add the dequeued item at the front of the sorted sublist.



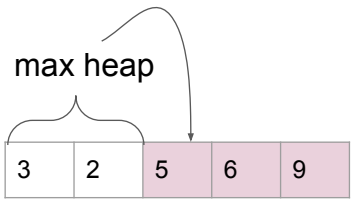
Heap Sort

3. While the size of the max heap is larger than 1:
   a. Dequeue an item from the max heap.
      i. Promote the last item in the max heap to the top.
      ii. Shift down the promoted item until it settles to a position which satisfies the max heap order.
   b. Add the dequeued item at the front of the sorted sublist.



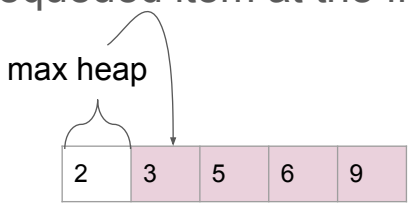
Heap Sort

3. While the size of the max heap is larger than 1:
   a. Dequeue an item from the max heap.
      i. Promote the last item in the max heap to the top.
      ii. Shift down the promoted item until it settles to a position which satisfies the max heap order.
   b. Add the dequeued item at the front of the sorted sublist.

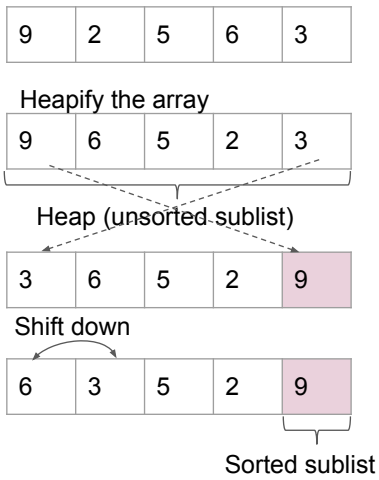


Heap Sort

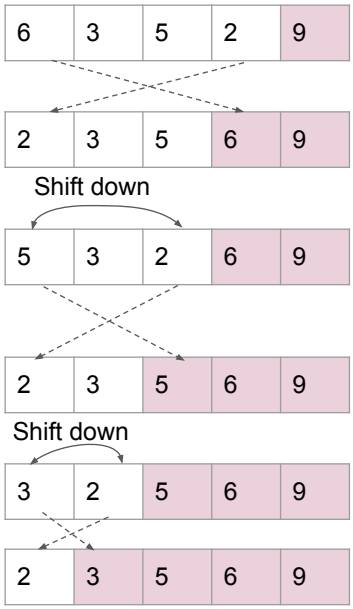
3. While the size of the max heap is larger than 1:
   a. Dequeue an item from the max heap.
      i. Promote the last item in the max heap to the top.
      ii. Shift down the promoted item until it settles to a position which satisfies the max heap order.
   b. Add the dequeued item at the front of the sorted sublist.



Heap Sort in Action



Heap Sort in Action



- Time complexity
  - Best: O(n log n)
  - Average: O(n log n)
  - Worst: O(n log n)
- Space complexity O(1)
- Unstable

Done!

Code Listings For Min Heap

```
def insert(items, item):
    """insert an item to min heap
    Args:
        items (list) : a list of integers (min heap)
        item (int) : an integer value to be inserted to the min heap
    """
    items.append(item)
    shift_up(items, len(items) - 1)
    return
```

```
def shift_up(items, i):
    """shift up an item in the list to maintain the min heap order
    Args:
        items (list) : a list of integers (min heap)
        i (int) : the index of the item of interest
    """
    iparent = index_parent(i)
    if iparent < 0 or items[iparent] <= items[i]:
        return
    temp = items[i]
    items[i] = items[iparent]
    items[iparent] = temp
    return shift_up(items, iparent)
```

```
def del_min(items, end):
    """pop the minimum valued item from the min heap
    Args:
        items (list) : a list of int
        end (int) : the index of the end of the heap
    Return:
        int : the smallest integer value in the min heap
        int : a new index of the end of the heap
    """
    min_item = items[0]
    items[0] = items[end]
    shift_down(items, 0, end-1)
    return min_item, end-1
```

```
def shift_down(items, i, end):
    """shift down an item in the list to keep the min heap order
    Args:
        items (list) : a list of int
        i (int) : the index of the item of interest
        end (int) : the end index of the heap
    """
    imin = index_minchild(items, i, end)
    if imin < 0 or items[imin] >= items[i]:
        return None
    temp = items[i]
    items[i] = items[imin]
    items[imin] = temp
    return shift_down(items, imin, end)
```

```
def heapify(items):
    """convert a list into a min heap
    Args:
        items (list) : a list of int
    """
    length = len(items)
    i = index_parent(length - 1)
    while i >= 0:
        shift_down(items, i)
        i = i - 1
    return
```

```
def index_parent(i):
    """compute the index of the parent
    Args:
        i (int) : index
    Returns:
        int : the index of the parent
    """
    return (i - 1) // 2
```

```
def index_left(i):
    """compute the index of the left child
    Args:
        i (int) : index
    Returns:
        int : the index of the left child
    """
    return 2 * i + 1
```

```
def index_right(i):
    """compute the index of the right child
    Args:
        i (int) : index
    Returns:
        int : the index of the right child
    """
    return 2 * i + 2
```

```
def index_minchild(items, i, end):
    """compute the index of child node with minimum key value
    Args:
        items (list) : a list of int
        i (int) : index of the node
        end (int) : the last index of the heap
    Returns:
        int : the index of the child node with minimum key value.
    """
    …
```

Code for Max Heap and Heap sort

Left as your exercises.

Priority Queue ADT (Minimum Priority Queue)

The textbook uses Object Oriented Approach and the list is a member variable

- PriorityQueue() creates a new, empty, priority queue.
- insert(k) adds a new item to the queue.
- find_min() returns the item with the minimum key value, leaving item in the queue.
- del_min() returns the item with the minimum key value, removing the item from the queue.
  a. Equivalent to our dequeue function.
- is_empty() returns true if the queue is empty, false otherwise.
- size() returns the number of items in the queue