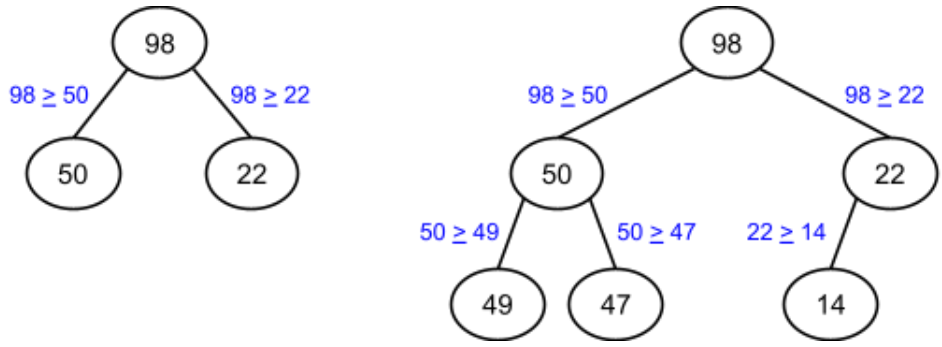
9.1 Heaps

Heap concept

Some applications require fast access to and removal of the maximum item in a changing set of items. For example, a computer may execute jobs one at a time; upon finishing a job, the computer executes the pending job having maximum priority. Ex: Four pending jobs have priorities 22, 14, 98, and 50; the computer should execute 98, then 50, then 22, and finally 14. New jobs may arrive at any time.

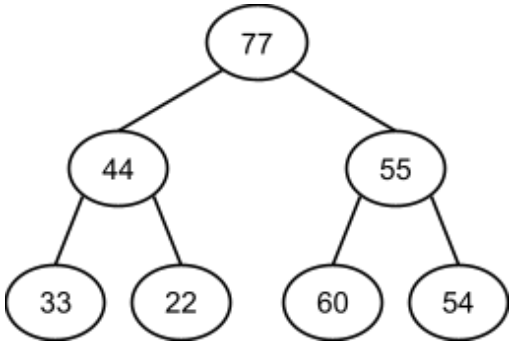
Maintaining jobs in fully-sorted order requires more operations than necessary, since only the maximum item is needed. A ***max-heap*** is a complete binary tree that maintains the simple property that a node's key is greater than or equal to the node's children's keys. (Actually, a max-heap may be any tree, but is commonly a binary tree). Because x ≥ y and y ≥ z implies x ≥ z, the property results in a node's key being greater than or equal to all the node's descendants' keys. Therefore, *a max-heap's root always has the maximum key in the entire tree.*

Figure 9.1.1: Max-heap property: A node's key is greater than or equal to the node's children's keys.



PARTICIPATION ACTIVITY 9.1.1: Max-heap property.

Consider this binary tree:



1) 33 violates the max-heap property due to being greater than 22.

- True
- False

2) 54 violates the max-heap property due to being greater than 44.

- True
- False

3) 60 violates the max-heap property due to being greater than 55.

- True
- False

4) A max-heap's root must have the maximum key.

- True
- False

Max-heap insert and remove operations

An ***insert*** into a max-heap starts by inserting the node in the tree's last level, and then swapping the node with its parent until no max-heap property violation occurs. Inserts fill a level (left-to-right) before adding another level, so the tree's height is always the minimum possible. The upward movement of a node in a max-heap is called ***percolating***.

A ***remove*** from a max-heap is always a removal of the root, and is done by replacing the root with the last level's last node, and swapping that node with its greatest child until no max-heap property violation occurs. Because upon completion that node will occupy another node's location (which was swapped upwards), the tree height remains the minimum possible.

PARTICIPATION ACTIVITY 9.1.2: Max-heap insert and remove operations.

Animation captions:

1. This tree is a max-heap. A new node gets initially inserted in the last level...
2. ...and then percolate node up until the max-heap property isn't violated.
3. Removing a node (always the root): Replace with last node, then percolate node down.

PARTICIPATION ACTIVITY

9.1.3: Max-heap inserts and deletes.

1) Given N nodes, what is the height of a max-heap?

- ○
- ○ N
- ○ Depends on the keys

2) Given a max-heap with levels 0, 1, 2, and 3, with the last level not full, after inserting a new node, what is the maximum possible swaps needed?

- ○ 1
- ○ 2
- ○ 3

3) Given a max-heap with N nodes, what is the worst-case complexity of an insert, assuming an insert is dominated by the swaps?

- ○ O( )
- ○ O( )

4) Given a max-heap with N nodes, what is the complexity for removing the root?

- ○ O( )
- ○ O( )

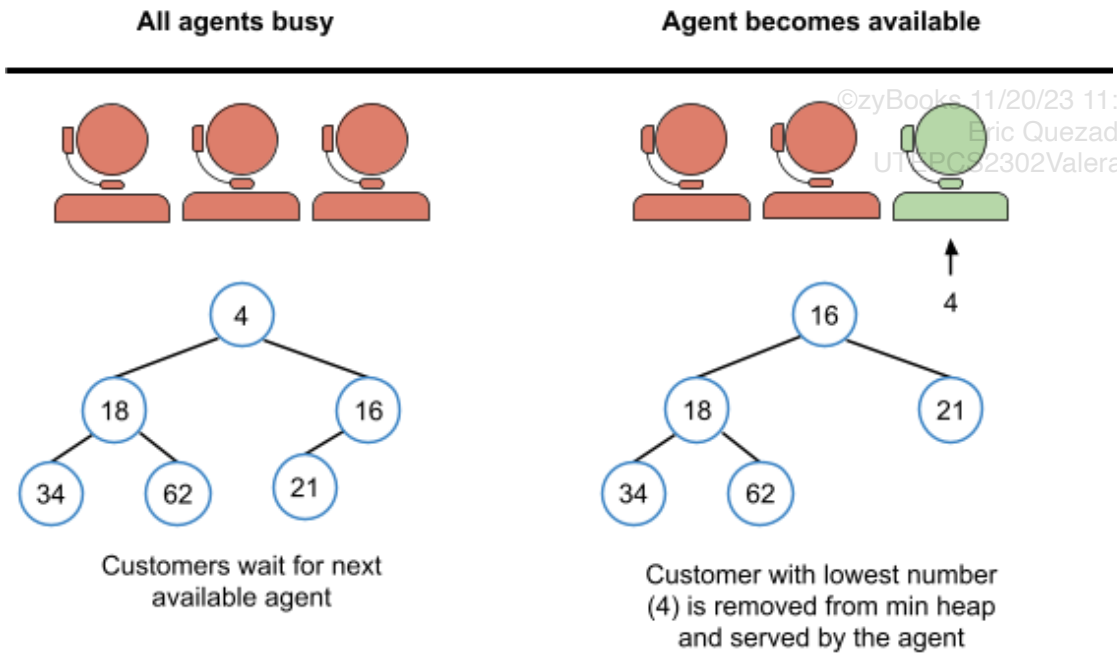
Min-heap

A ***min-heap*** is similar to a max-heap, but a node's key is less than or equal to its children's keys.

Example 9.1.1: Online tech support waiting lines commonly use min-heaps.

Many companies have online technical support that lets a customer chat with a support agent. If the number of customers seeking support is greater than the number of available agents, customers enter a virtual waiting line. Each customer has a priority that determines their place in line. The customer with the highest priority is served by the next available agent.

A min-heap is commonly used to manage prioritized queues of customers awaiting support. Customers that entered the line earlier and/or have a more urgent issue get assigned a lower number, which corresponds to a higher priority. When an agent becomes available, the customer with the lowest number is removed from the heap and served by the agent.



PARTICIPATION ACTIVITY 9.1.4: Min-heaps and customer support.

1) A customer with a higher priority has a lower numerical value in the min-heap.
   - ○ True
   - ○ False

2) If 2,000 customers are waiting for technical support, removing a customer from the min-heap requires about 2,000 operations.
   - ○ True
   - ○ False

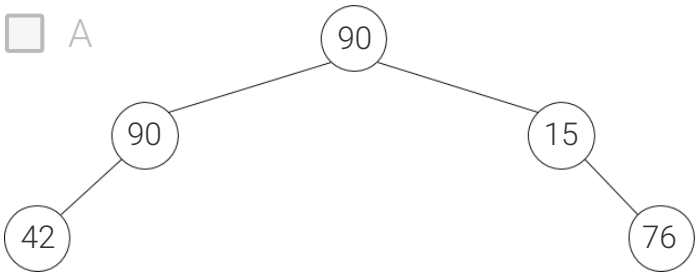
CHALLENGE ACTIVITY 9.1.1: Heaps.

502696.2096894.qx3zqy7

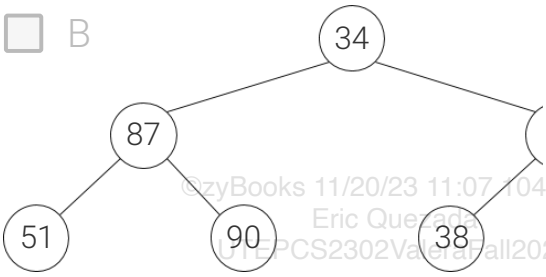
Start

Select all valid max-heaps.

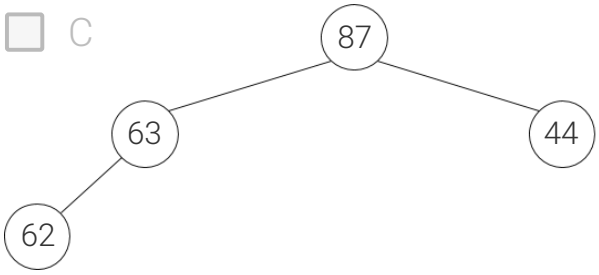
A



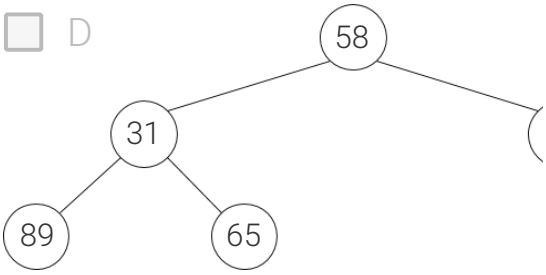
B



C



D



1 2 3 4

Check Next

9.2 Heaps using arrays

Heap storage

Heaps are typically stored using arrays. Given a tree representation of a heap, the heap's array form is produced by traversing the tree's levels from left to right and top to bottom. The root node is always the entry at index 0 in the array, the root's left child is the entry at index 1, the root's right child is the entry at index 2, and so on.

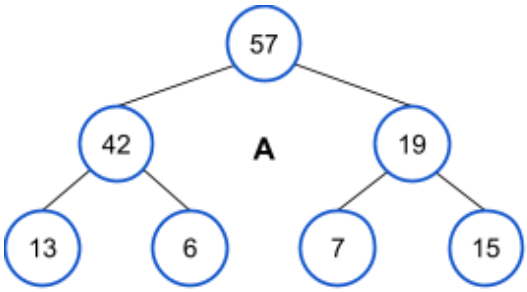
PARTICIPATION ACTIVITY 9.2.1: Max-heap stored using an array.

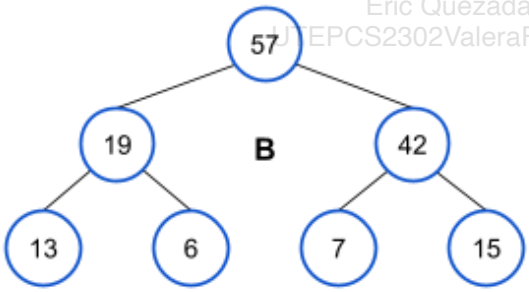
Animation captions:

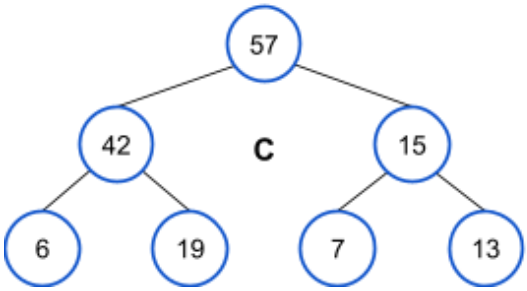
1. The max-heap's array form is produced by traversing levels left to right and top to bottom.
2. When 63 is inserted, the percolate-up operation happens within the array.

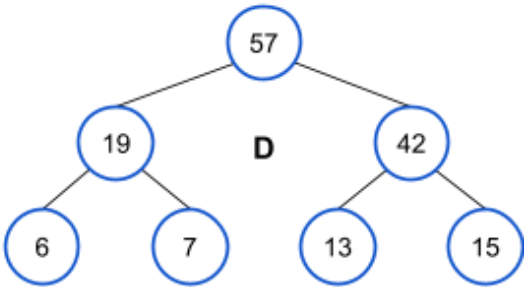
PARTICIPATION ACTIVITY 9.2.2: Heap storage.

Match each max-heap to the corresponding storage array.









If unable to drag and drop, refresh the page.

Heap D | Heap A | Heap C | Heap B

	57 19 42 13 6 7 15
	57 42 15 6 19 7 13
	57 42 19 13 6 7 15
	57 19 42 6 7 13 15

Reset

Parent and child indices

Because heaps are not implemented with node structures and parent/child pointers, traversing from a node to parent or child nodes requires referring to nodes by index. The table below shows parent and child index formulas for a heap.

Table 9.2.1: Parent and child indices for a heap.

Node index	Parent index	Child indices
0	N/A	1, 2
1	0	3, 4
2	0	5, 6
3	1	7, 8
4	1	9, 10
5	2	11, 12
...	...	...
i		2 * i + 1, 2 * i + 2

PARTICIPATION ACTIVITY 9.2.3: Heap parent and child indices.

1) What is the parent index for a node at index 12?

- 3
- 4
- 5
- 6

2) What are the child indices for a node at index 6?

- 7 and 8
- 12 and 13
- 13 and 14
- 12 and 24

3) The formula for computing parent node index should not be used on the root node.

- True
- False

4) The formula for computing child node indices does not work on the root node.

- True
- False

5) The formula for computing a child index evaluates to -1 if the parent is a leaf node.

- True
- False

Percolate algorithm

Following is the pseudocode for the array-based percolate-up and percolate-down functions. The functions operate on an array that represents a max-heap and refer to nodes by array index.

Figure 9.2.1: Max-heap percolate up algorithm.

```
MaxHeapPercolateUp(nodeIndex, heapArray) {
   while (nodeIndex > 0) {
      parentIndex = (nodeIndex - 1) / 2
      if (heapArray[nodeIndex] <= heapArray[parentIndex])
         return
      else {
         swap heapArray[nodeIndex] and
heapArray[parentIndex]
         nodeIndex = parentIndex
      }
   }
}
```

Figure 9.2.2: Max-heap percolate down algorithm.

```
MaxHeapPercolateDown(nodeIndex, heapArray, arraySize) {
   childIndex = 2 * nodeIndex + 1
   value = heapArray[nodeIndex]

   while (childIndex < arraySize) {
      // Find the max among the node and all the node's
children
      maxValue = value
      maxIndex = -1
      for (i = 0; i < 2 && i + childIndex < arraySize; i++)
{
         if (heapArray[i + childIndex] > maxValue) {
            maxValue = heapArray[i + childIndex]
            maxIndex = i + childIndex
         }
      }

      if (maxValue == value) {
         return
      }
      else {
         swap heapArray[nodeIndex] and heapArray[maxIndex]
         nodeIndex = maxIndex
         childIndex = 2 * nodeIndex + 1
      }
   }
}
```


PARTICIPATION ACTIVITY 9.2.4: Percolate algorithm.

1) MaxHeapPercolateUp works for a node index of 0.
   - True
   - False

2) MaxHeapPercolateDown has a precondition that nodeIndex is < arraySize.
   - True
   - False

3) MaxHeapPercolateDown checks the node's left child first, and immediately swaps the nodes if the left child has a greater key.

- True
- False

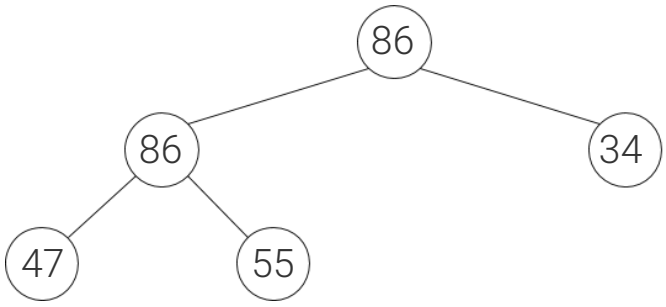
4) In MaxHeapPercolateUp, the while loop's condition nodeIndex > 0 guarantees that parentIndex is >= 0.

- True
- False

CHALLENGE ACTIVITY 9.2.1: Heaps using arrays.

502696.2096894.qx3zqy7

Start



What is the above max-heap's array form?

Ex: 86, 75, 30 (comma between values)

1	2	3	4	5

Check Next

9.3 Python: Heaps

Heaps and the MaxHeap class

Each level of a max-heap tree grows from left to right; a new level is added only after the current level fills up completely. Because the tree is nearly full (with at most one level not completely filled), an array implementation is efficient. The array implementation means that the root is always at index 0, and the index of any node's parent and children can be easily calculated:

- parent_index = (node_index - 1) // 2
- left_child_index = 2 * node_index + 1
- right_child_index = 2 * node_index + 2

Note: The `//` operator used in the parent_index calculation performs an integer division, where the quotient's decimal portion, if any, is dropped.

The MaxHeap class is defined with an initially-empty list as the only data member: `self.heap_array = []`.

Figure 9.3.1: The MaxHeap class.

```
class MaxHeap:
    def __init__(self):
        self.heap_array =
[]
```

The list's elements are the nodes of the tree; no actual `Node` class is used like in other tree implementations. Even though no `Node` class is used, the elements of the list are still called nodes.

PARTICIPATION ACTIVITY 9.3.1: Implementing a heap with an array.

A max heap is represented by the array:
[22, 10, 17, 7, 9, 8, 16, 6, 2, 3]
Calculate the requested node's index. If the requested node does not exist, enter "none".

1) The parent of node 9.

Index:

Check Show answer

2) The left child of node 17.

Index:

Check Show answer

3) The right child of node 9.

Index:

Check Show answer

4) The right child of node 7.

Index:

Check Show answer

Heap percolate methods

The MaxHeap methods insert() and remove() make use of two methods, percolate_up() and percolate_down().

percolate_up() takes a starting index (node_index) as an argument and compares the value at node_index (the current value) to the parent value. If the parent value is smaller than the current value, the two items swap positions in the list. The process then repeats from the current value's new position. The algorithm stops when the current value reaches the root (at index 0), or when the parent value is greater than or equal to the current value, meaning no max heap violations exist in the list.

Figure 9.3.2: The MaxHeap's percolate_up() method.

```
def percolate_up(self, node_index):
    while node_index > 0:
        # compute the parent node's index
        parent_index = (node_index - 1) // 2

        # check for a violation of the max heap property
        if self.heap_array[node_index] <=
self.heap_array[parent_index]:
            # no violation, so percolate up is done.
            return
        else:
            # swap heap_array[node_index] and
heap_array[parent_index]
            temp = self.heap_array[node_index]
            self.heap_array[node_index] =
self.heap_array[parent_index]
            self.heap_array[parent_index] = temp

            # continue the loop from the parent node
            node_index = parent_index
```


The percolate_down() method takes a starting index as an argument and compares the value at node_index (the current value) to the current value's two children's values. If either child's value is larger than the current value, the current value swaps positions with whichever child is larger. The process continues from the current value's new position. The algorithm ends when the current value is at the tree's bottom level, or when both children's values are less than or equal to the current value, meaning no max heap violations exist in the list.

Figure 9.3.3: The MaxHeap's percolate_down() method.

```
def percolate_down(self, node_index):
    child_index = 2 * node_index + 1
    value = self.heap_array[node_index]

    while child_index < len(self.heap_array):
        # Find the max among the node and the node's children
        max_value = value
        max_index = -1
        i = 0
        while i < 2 and i + child_index < len(self.heap_array):
            if self.heap_array[i + child_index] > max_value:
                max_value = self.heap_array[i + child_index]
                max_index = i + child_index
            i = i + 1

        # check for a violation of the max heap property
        if max_value == value:
            return
        else:
            # swap heap_array[node_index] and
heap_array[max_index]
            temp = self.heap_array[node_index]
            self.heap_array[node_index] =
self.heap_array[max_index]
            self.heap_array[max_index] = temp

            # continue loop from the larger child node
            node_index = max_index
            child_index = 2 * node_index + 1
```


PARTICIPATION ACTIVITY 9.3.2: percolate_up() and percolate_down() methods.

1) Max heap array: `[ 15, 10, 8, 5, 2, 7, 14 ]`
How many swaps are done when percolate_up() is called on the last index?

- 0
- 1
- 2

2) Max heap array: `[ 20, 5, 20, 3, 1, 10 ]`
How many swaps are done when percolate_down() is called on the first index?

- 0
- 1
- 2

3) During percolate_down(), a node's key is smaller than both the left child's key and the right child's key. Which child does the node swap with?

- Left child
- Right child
- Whichever child is larger
- Either child is ok

insert() and remove() methods

The MaxHeap insert() method adds a new value to the end of the list, and then uses the percolate_up() method to restore the max heap property.

The MaxHeap remove() method has no parameters and returns the maximum value in the list (which always occurs at index 0). The list's index 0 position is assigned with the last item in the list, and the last item is removed using the list's pop() method. The max heap property is restored by calling percolate_down() from the index 0 position.

Figure 9.3.4: The MaxHeap class's insert() and remove() methods.

```
def insert(self, value):
    # add the new value to the end of the array.
    self.heap_array.append(value)

    # percolate up from the last index to restore heap
property.
    self.percolate_up(len(self.heap_array) - 1)


def remove(self):
    # save the max value from the root of the heap.
    max_value = self.heap_array[0]

    # move the last item in the array into index 0.
    replace_value = self.heap_array.pop()
    if len(self.heap_array) > 0:
        self.heap_array[0] = replace_value

        # percolate down to restore max heap property.
        self.percolate_down(0)

    # return the max value
    return max_value
```


PARTICIPATION ACTIVITY 9.3.3: Max heap insert() and remove() methods.

1) The insert() method starts by inserting at the max heap's root.
   - ○ True
   - ○ False

2) The insert() method uses percolate_up(), while the remove() method uses percolate_down().
   - ○ True
   - ○ False

3) The remove() method takes a search key as an argument and navigates through the tree looking for a node with a key that matches the argument to remove.
   - ○ True
   - ○ False

4) Both insert() and remove() are operations in the worst case.

- True
- False

zyDE 9.3.1: MaxHeap data structure and algorithms.

This program uses the MaxHeap class to insert some items from a list, and then remove the items. Notice that the items are removed in reverse sorted order. Some print statemer have been added to the MaxHeap class so the operations can be observed. Try to answer these questions:

- Which requires more swaps: when the input list starts sorted or starts reverse sorte
- Can you change the code to become a MinHeap instead? (Hint: ignoring variable na only two lines of code need to be changed to make a MinHeap!)

Heap.py Load default templat

```
1  class MaxHeap:
2
3      def __init__(self):
4          self.heap_array = []
5
6      def percolate_up(self, node_index):
7          while node_index > 0:
8              # compute the parent node's index
9              parent_index = (node_index - 1) // 2
10
11             # check for a violation of the max heap property
12             if self.heap_array[node_index] <= self.heap_array[parent_
13                 # no violation, so percolate up is done.
14                 return
15             else:
```

Run

9.4 Heap sort

Heapify operation

Heapsort is a sorting algorithm that takes advantage of a max-heap's properties by repeatedly removing the max and building a sorted array in reverse order. An array of unsorted values must first be converted into a heap. The ***heapify*** operation is used to turn an array into a heap. Since leaf nodes already satisfy the max heap property, heapifying to build a max-heap is achieved by percolating down on every non-leaf node in reverse order.

PARTICIPATION ACTIVITY 9.4.1: Heapify operation.

Animation captions:

1. If the original array is represented in tree form, the tree is not a valid max-heap.
2. Leaf nodes always satisfy the max heap property, since no child nodes exist that can contain larger keys. Heapification will start on node 92.
3. 92 is greater than 24 and 42, so percolating 92 down ends immediately.
4. Percolating 55 down results in a swap with 98.
5. Percolating 77 down involves a swap with 98. The resulting array is a valid max-heap.

The heapify operation starts on the internal node with the largest index and continues down to, and including, the root node at index 0. Given a binary tree with N nodes, the largest internal node index is - 1.

Table 9.4.1: Max-heap largest internal node index.

Number of nodes in binary heap	Largest internal node index
1	-1 (no internal nodes)
2	0
3	0
4	1
5	1
6	2
7	2
...	...

N	- 1

PARTICIPATION ACTIVITY 9.4.2: Heapify operation.

1) For an array with 7 nodes, how many percolate-down operations are necessary to heapify the array?

Check Show answer

2) For an array with 10 nodes, how many percolate-down operations are necessary to heapify the array?

Check Show answer

PARTICIPATION ACTIVITY 9.4.3: Heapify operation - critical thinking.

1) An array sorted in ascending order is already a valid max-heap.

- True
- False

2) Which array could be heapified with the fewest number of operations, including all swaps used for percolating?

- (10, 20, 30, 40)
- (30, 20, 40, 10)
- (10, 10, 10, 10)

Heapsort overview

Heapsort begins by heapifying the array into a max-heap and initializing an end index value to the size of the array minus 1. Heapsort repeatedly removes the maximum value, stores that value at the end index,

and decrements the end index. The removal loop repeats until the end index is 0.

PARTICIPATION ACTIVITY 9.4.4: Heapsort.

Animation captions:

1. The array is heapified first. Each internal node is percolated down, from highest node index to lowest.
2. The end index is initialized to 6, to refer to the last item. 94's "removal" starts by swapping with 68.
3. Removing from a heap means that the rightmost node on the lowest level disappears before the percolate down. End index is decremented after percolating.
4. 88 is swapped with 49, the last node disappears, and 49 is percolated down.
5. The process continues until end index is 0.
6. The array is sorted.

PARTICIPATION ACTIVITY 9.4.5: Heapsort.

Suppose the original array to be heapified is (11, 21, 12, 13, 19, 15).

1) The percolate down operation must be performed on which nodes?
   - 15, 19, and 13
   - 12, 21, and 11
   - All nodes in the heap

2) What are the first 2 elements swapped?
   - 11 and 21
   - 21 and 13
   - 12 and 15

3) What are the last 2 elements swapped?
   - 11 and 19
   - 11 and 21
   - 19 and 21

4) What is the heapified array?

- (11, 21, 12, 13, 19, 15)
- (21, 19, 15, 13, 12, 11)
- (21, 19, 15, 13, 11, 12)

Heapsort algorithm

Heapsort uses 2 loops to sort an array. The first loop heapifies the array using MaxHeapPercolateDown. The second loop removes the maximum value, stores that value at the end index, and decrements the end index, until the end index is 0.

Figure 9.4.1: Heap sort.

```
Heapsort(numbers, numbersSize) {
   // Heapify numbers array
   for (i = numbersSize / 2 - 1; i >= 0; i--) {
      MaxHeapPercolateDown(i, numbers,
numbersSize)
   }

   for (i = numbersSize - 1; i > 0; i--) {
      Swap numbers[0] and numbers[i]
      MaxHeapPercolateDown(0, numbers, i)
   }
}
```

PARTICIPATION ACTIVITY 9.4.6: Heapsort algorithm.

1) How many times will MaxHeapPercolateDown be called by Heapsort when sorting an array with 10 elements?

- 5
- 10
- 14
- 20

2) Calling Heapsort on an array with 1 element will cause an out of bounds array access.

- True
- False

3) Heapsort's worst-case runtime is O(N log N).

- True
- False

4) Heapsort uses recursion.

- True
- False

CHALLENGE ACTIVITY 9.4.1: Heap sort.

502696.2096894.qx3zqy7

Start

Given the array:

74	51	41	14	95	22	52

Heapify into a max-heap.

Ex: 86, 75, 30

1 2 3

Check Next

9.5 Python: Heap sort

The heap sort algorithm can be implemented in Python to sort a list in place. Only the percolate down function is needed, not a full heap implementation.

The max_heap_percolate_down() function takes the following arguments:

- node_index: index of the node to be percolated down
- heap_list: list that stores the heap's contents
- list_size: an integer representing the heap's size

The heap sort algorithm requires operating on a subset of the heap contents, so the list_size argument is required and may not be equal to `len(heap_list)`.

The heap_sort() function takes a single argument: the list of items to be sorted. The list is heapified first, then the maximum is removed repeatedly, and the sorted list is built in reverse order.

Figure 9.5.1: Heap sort algorithm.

```
# Binary max heap percolate down
def max_heap_percolate_down(node_index, heap_list,
list_size):
    child_index = 2 * node_index + 1
    value = heap_list[node_index]

    while child_index < list_size:
        # Find the max among the node and all the node's
children
        max_value = value
        max_index = -1
        i = 0
        while i < 2 and i + child_index < list_size:
            if heap_list[i + child_index] > max_value:
                max_value = heap_list[i + child_index]
                max_index = i + child_index
            i = i + 1

        if max_value == value:
            return

        # Swap heap_list[node_index] and heap_list[max_index]
        temp = heap_list[node_index]
        heap_list[node_index] = heap_list[max_index]
        heap_list[max_index] = temp

        node_index = max_index
        child_index = 2 * node_index + 1


# Sorts the list of numbers using the heap sort algorithm
def heap_sort(numbers):
    # Heapify numbers list
    i = len(numbers) // 2 - 1
    while i >= 0:
        max_heap_percolate_down(i, numbers, len(numbers))
        i = i - 1

    i = len(numbers) - 1
    while i > 0:
        # Swap numbers[0] and numbers[i]
        temp = numbers[0]
        numbers[0] = numbers[i]
        numbers[i] = temp

        max_heap_percolate_down(0, numbers, i)
        i = i - 1
```


©zyBooks 11/20/23 11:07 1048447
Eric Quezada
UTEPCS2302ValeraFall2023

PARTICIPATION ACTIVITY 9.5.1: Heap sort algorithm.

1) What is the contents of `numbers_list` after the following code executes?

```
numbers_list = [ 10, 62, 97,
18, 4, 9, 36 ]
max_heap_percolate_down(0,
numbers_list,
len(numbers_list))
```

- [97, 62, 36, 18, 4, 9, 10]
- [62, 18, 97, 10, 4, 9, 36]

2) If the list of numbers passed to heap_sort() is [19, 87, 33, 22, 61, 12], what is the list's contents after the first loop heapifies?

- [19, 87, 33, 22, 61, 12]
- [87, 19, 33, 22, 61, 12]
- [87, 61, 33, 22, 19, 12]

3) If heap_sort's second loop always passed `len(numbers)` as the third argument to max_heap_percolate_down(), heap_sort() would _____.

- enter an infinite loop and never complete
- finish, but the resulting numbers list may not be sorted
- still correctly sort the list

zyDE 9.5.1: Heap sort.

This program uses heap_sort() to sort a list of numbers in ascending order. Try to answer these questions:

- What is the heapified list contents? Where can a single print statement be inserted i heap_sort() function to display the heapified list?
- What must change to sort the list in descending order instead of ascending? (Hint: 1 line of code needs to change!)

Heapsort.py Load default templat

```
1 # Binary max heap percolate down
```

```
# Binary max heap percolate down
def max_heap_percolate_down(node_index, heap_list, list_size):
    child_index = 2 * node_index + 1
    value = heap_list[node_index]

    while child_index < list_size:
        # Find the max among the node and all the node's children
        max_value = value
        max_index = -1
        i = 0
        while i < 2 and i + child_index < list_size:
            if heap_list[i + child_index] > max_value:
                max_value = heap_list[i + child_index]
                max_index = i + child_index
            i = i + 1
```


Run

9.6 Priority queue abstract data type (ADT)

Priority queue abstract data type

A ***priority queue*** is a queue where each item has a priority, and items with higher priority are closer to the front of the queue than items with lower priority. The priority queue ***enqueue*** operation inserts an item such that the item is closer to the front than all items of lower priority, and closer to the end than all items of equal or higher priority. The priority queue ***dequeue*** operation removes and returns the item at the front of the queue, which has the highest priority.

PARTICIPATION ACTIVITY 9.6.1: Priority queue enqueue and dequeue.

Animation content:

undefined

Animation captions:

1. Enqueueing a single item with priority 7 initializes the priority queue with 1 item.
2. If a lower numerical value indicates higher priority, enqueueing 11 adds the item to the end of the queue.
3. Since 5 < 7, enqueueing 5 puts the item at the priority queue's front.

4. When enqueueing items of equal priority, the first-in-first-out rules apply. The 2nd item with priority 7 comes after the first.
5. Dequeue removes from the front of the queue, which is always the highest priority item.

PARTICIPATION ACTIVITY 9.6.2: Priority queue enqueue and dequeue.

Assume that lower numbers have higher priority and that a priority queue currently holds items: 54, 71, 86 (front is 54).

1) Where would an item with priority 60 reside after being enqueued?
   - Before 54
   - After 54
   - After 86

2) Where would an additional item with priority 54 reside after being enqueued?
   - Before the first 54
   - After the first 54
   - After 86

3) The dequeue operation would return which item?
   - 54
   - 71
   - 86

Common priority queue operations

In addition to enqueue and dequeue, a priority queue usually supports peeking and length querying. A ***peek*** operation returns the highest priority item, without removing the item from the front of the queue.

Table 9.6.1: Common priority queue ADT operations.

Operation	Description	Example starting with priority queue: 42, 61, 98 (front is 42)
Enqueue(PQueue, x)	Inserts x after all equal or higher priority items	Enqueue(PQueue, 87). PQueue: 42, 61, 87, 98

Dequeue(PQueue)	Returns and removes the item at the front of PQueue	Dequeue(PQueue) returns 42. PQueue: 61, 98
Peek(PQueue)	Returns but does not remove the item at the front of PQueue	Peek(PQueue) returns 42. PQueue: 42, 61, 98
IsEmpty(PQueue)	Returns true if PQueue has no items	IsEmpty(PQueue) returns false.
GetLength(PQueue)	Returns the number of items in PQueue	GetLength(PQueue) returns 3.

PARTICIPATION ACTIVITY 9.6.3: Common priority queue ADT operations.

Assume servicePQueue is a priority queue with contents: 11, 22, 33, 44, 55.

1) What does GetLength(servicePQueue) return?
   - 5
   - 11
   - 55

2) What does Dequeue(servicePQueue) return?
   - 5
   - 11
   - 55

3) After dequeuing an item, what will Peek(servicePQueue) return?
   - 11
   - 22
   - 33

4) After calling Dequeue(servicePQueue) a total of 5 times, what will GetLength(servicePQueue) return?

- -1
- 0
- Undefined

Enqueueing items with priority

A priority queue can be implemented such that each item's priority can be determined from the item itself. Ex: A customer object may contain information about a customer, including the customer's name and a service priority number. In this case, the priority resides within the object.

A priority queue may also be implemented such that all priorities are specified during a call to ***EnqueueWithPriority***: An enqueue operation that includes an argument for the enqueued item's priority.

PARTICIPATION ACTIVITY 9.6.4: Priority queue EnqueueWithPriority operation.

Animation content:

undefined

Animation captions:

1. Calls to EnqueueWithPriority() enqueue objects A, B, and C into the priority queue with the specified priorities.
2. In this implementation, the objects enqueued into the queue do not have data members representing priority.
3. Priorities specified during each EnqueueWithPriority() call are stored alongside the queue's objects.

PARTICIPATION ACTIVITY 9.6.5: Enqueue-with-priority operation.

1) A priority queue implementation that implements only Enqueue() and not EnqueueWithPriority() requires enqueued items to have a data member representing priority.

- True
- False

2) A priority queue implementation that implements only EnqueueWithPriority() and not Enqueue() requires objects to have a data member representing priority.

- True
- False

Implementing priority queues with heaps

A priority queue is commonly implemented using a heap. A heap will keep the highest priority item in the root node and allow access in O(1) time. Adding and removing items from the queue will operate in worst-case O( ) time.

Table 9.6.2: Implementing priority queues with heaps.

Priority queue operation	Heap functionality used to implement operation	Worst-case runtime complexity
Enqueue	Insert	O( )
Dequeue	Remove	O( )
Peek	Return value in root node	O( )
IsEmpty	Return true if no nodes in heap, false otherwise	O( )
GetLength	Return number of nodes (expected to be stored in the heap's member data)	O( )

PARTICIPATION ACTIVITY 9.6.6: Implementing priority queues with heaps.

1) The Dequeue and Peek operations both return the value in the root, and therefore have the same worst-case runtime complexity.

- True
- False

2) When implementing a priority queue with a heap, no operation will have a runtime complexity worse than O( ).

- True
- False

3) If items in a priority queue with a lower numerical value have higher priority, then a max-heap should be used to implement the priority queue.

- True
- False

4) A priority queue is always implemented using a heap.

- True
- False

CHALLENGE ACTIVITY 9.6.1: Priority queue abstract data type.

502696.2096894.qx3zqy7

Start

Assume that lower numbers have higher priority and that a priority queue numPQueue current holds items: 37, 42, 47, 60 (front is 37).

Where does Enqueue(numPQueue, 13) add an item?

Where does Enqueue(numPQueue, 60) add an item?

Where does Enqueue(numPQueue, 39) add an item?

1 2 3 4 5

Check Next

9.7 Treaps

Treap basics

A BST built from inserts of N nodes having random-ordered keys stays well-balanced and thus has near-minimum height, meaning searches, inserts, and deletes are O( ). Because insertion order may not be controllable, a data structure that somehow randomizes BST insertions is desirable. A ***treap*** uses a main key that maintains a binary search tree ordering property, and a secondary key generated randomly (often called "priority") during insertions that maintains a heap property. The combination usually keeps the tree balanced. The word "treap" is a mix of tree and heap. This section assumes the heap is a max-heap. Algorithms for basic treap operations include:

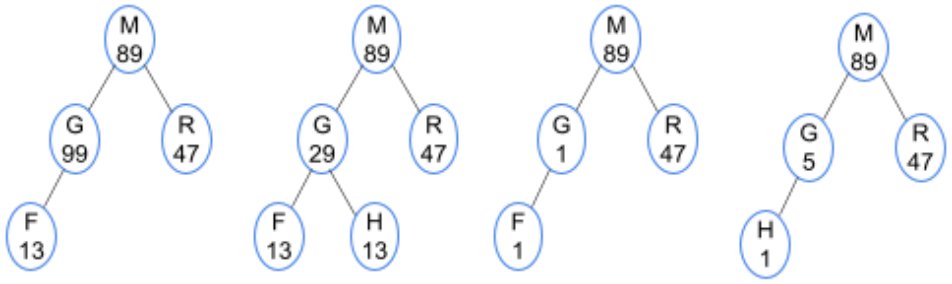
- A treap ***search*** is the same as a BST search using the main key, since the treap is a BST.
- A treap ***insert*** initially inserts a node as in a BST using the main key, then assigns a random priority to the node, and percolates the node up until the heap property is not violated. In a heap, a node is moved up via a swap with the node's parent. In a treap, a node is moved up via a *rotation at the parent*. Unlike a swap, a rotation maintains the BST property.
- A treap ***delete*** can be done by setting the node's priority such that the node should be a leaf (-∞ for a max-heap), percolating the node down using rotations until the node is a leaf, and then removing the node.

PARTICIPATION ACTIVITY 9.7.1: Treap insert: First insert as a BST, then randomly assign a priority and use rotations to percolate node up to maintain heap.

Animation captions:

1. The keys maintain a BST, the priorities a heap. Insert B as a BST...
2. Assign random priority (70). Rotate (which keep a BST) the node up until the priorities maintain a heap: 20 not > 70: Rotate. 47 not > 70: Rotate. 80 > 70: Done.

PARTICIPATION ACTIVITY 9.7.2: Recognizing treaps.



(a) (b) (c) (d)

1) (a)

- Treap
- Not a treap

2) (b)

- Treap
- Not a treap

3) (c)

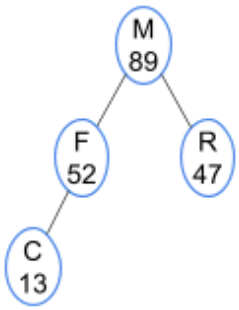
- Treap
- Not a treap

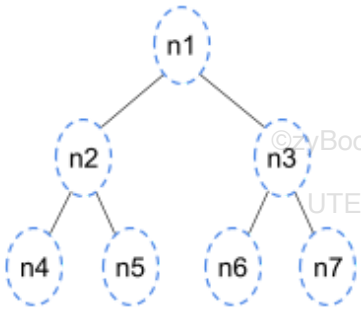
4) (d)

- Treap
- Not a treap

PARTICIPATION ACTIVITY 9.7.3: Treap insert.

When performing an insert, indicate each node's new location using the template tree's labels (n1...n7).





1) Where will a new node H first be inserted?

Check Show answer

2) H is assigned a random priority of 20. To where does H percolate?

Check Show answer

3) Where will a new node P first be inserted?

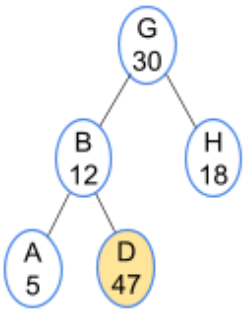
Check Show answer

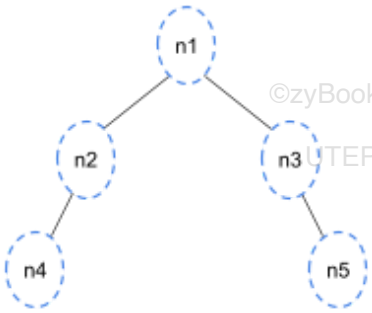
4) P is assigned a random priority of 65. To where does P percolate?

Check Show answer

PARTICIPATION ACTIVITY 9.7.4: Treap insert.

Node D was just inserted, and assigned a random priority of 47. Rotations are needed to not violate the heap property. Match the node value to the corresponding location in the tree template on the right after the rotations are completed.





If unable to drag and drop, refresh the page.

D, 47 | B, 12 | H, 18 | A, 5 | G, 30

	n1
	n2
	n3
	n4
	n5

Reset

Treap delete

A treap delete could be done by first doing a BST delete (copying the successor to the node-to-delete, then deleting the original successor), followed by percolating the node down until the heap property is not violated. However, a simpler approach just sets the node-to-delete's priority to -∞ (for a max-heap), percolates the node down until a leaf, and removes the node. Percolating the node down uses rotations, not swaps, to maintain the BST property. Also, the node is rotated in the direction of the lower-priority child, so that the node rotated up has a higher priority than that child, to keep the heap property.

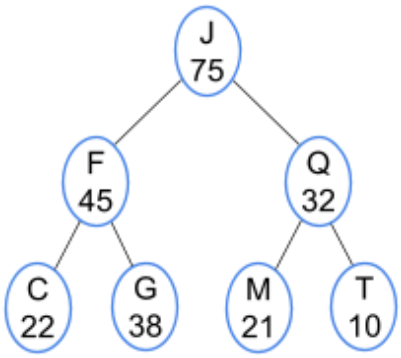
PARTICIPATION ACTIVITY 9.7.5: Treap delete: Set priority such that node must become a leaf, then percolate down using rotations.

Animation captions:

1. Node F is to be deleted. First set F's priority to -∞.
2. Rotate (to keep a BST) until the node becomes a leaf node. 29 > 13: Rotate right.
3. Rotate until node becomes a leaf node. Rotate left (the only option).
4. Remove leaf node.

PARTICIPATION ACTIVITY 9.7.6: Treap delete algorithm.

Each question starts from the original tree. Use this text notation for the tree: (J (F (C, G), Q (M, T))). A - means the child does not exist.



1) What is the tree after removing G?

- (J (F (C, -), Q (M, T)))
- (J (C (-, F), Q (M, T)))

2) What is the tree after removing Q?

- (J (F (C, G), M(-, T))
- (J (F (C, G), T(M, -))

PARTICIPATION ACTIVITY 9.7.7: Treaps.

1) A treap's nodes have random main keys.

- True
- False

2) A treap's nodes have random priorities.

- True
- False

3) Suppose a treap is built by inserting nodes with main keys in this order: A, B, C, D, E, F, G. The treap will have 7 levels, with each level having one node with a right child.

- True
- False

9.8 LAB: Timeout manager with PriorityQueue

Timeout manager overview

A timeout manager stores a priority queue of timeout items, each a (callback function, callback time) pair. Each callback function is called approximately N milliseconds after the timeout is set, where N is the delay specified when adding the timeout item. Ex:

- At time t = 0, a 500 millisecond timeout is set for function A
- At time t = 100, a 3000 millisecond timeout is set for function B
- At time t = 2000, a 1000 millisecond timeout is set for function C

So the timeout manager should call the callbacks as follows:

- Call function A at time t = 0 + 500 = 500
- Call function C at time t = 2000 + 1000 = 3000
- Call function B at time t = 100 + 3000 = 3100

A timeout item with a callback time <= the current time is said to be "expired".

Millisecond-level callback precision is often unfeasible. So a timeout manager typically has an update function that is called by external code every so often, ex: every 100 milliseconds. The manager's update function calls each expired timeout's callback function.

Step 1: Inspect TimeoutItem.py

Inspect the TimeoutItem class declaration in the read only TimeoutItem.py file. Access TimeoutItem.py by clicking on the orange arrow next to main.py at the top of the coding window. The `callback_time` attributes stores the time the item was added plus the item's delay. Ex: A TimeoutItem created at t=500 with a delay of 1000 has callback_time = 500 + 1000 = 1500. The `callback` attributes is the function to call after the timeout expires.

TimeoutItem's comparison operators are implemented so that TimeoutItems can be put in a PriorityQueue that gives higher priority to lesser callback_time values.

Step 2: Inspect Clocks.py

Inspect class declarations in the read only Clocks.py file. MillisecondClock is an abstract base class for a clock. The abstract get_time() method returns an integer indicating the clock's current time, in milliseconds.

Times are simulated during grading using the TestClock class. TestClock inherits from MillisecondClock and allows the clock's time to be set via the set_time() method.

Step 3: Implement TimeoutManager's add_timeout() and update() methods

Complete the TimeoutManager class implementation in TimeoutManager.py. add_timeout() must add a new TimeoutItem to the priority queue that expires `delay_before_callback` milliseconds after the current time. update() must dequeue and call the callback function for each expired timeout item. Use TimeoutManager's `clock` attributes to get the current time.

Step 4: Test in develop mode, then submit

File main.py contains two automated tests that can be run in develop mode. Each adds timeouts and invokes updates at various times, then verifies that the proper callback functions are called during each update. Additional tests can be added, if desired. Ensure that the two tests in develop mode pass before submitting your code.

LAB ACTIVITY | 9.8.1: LAB: Timeout manager with PriorityQueue | 10 / 10

Downloadable files

`main.py` , `Clocks.py` , `TimeoutItem.py` , and `TimeoutManager.py`

Download

Current file: **main.py** **Load default template...**

```
1  import sys
2  from Clocks import TestClock
3  from TimeoutManager import TimeoutManager
4
5  def main():
6      test1_result = test1(sys.stdout)
7      print()
8      test2_result = test2(sys.stdout)
9      print()
10
11     print(f"""Local test 1: {'PASS' if test1_result else 'FAIL'}""")
12     print(f"""Local test 2: {'PASS' if test2_result else 'FAIL'}""")
13
14 def callback_function(callbacks_list, callback_name, msg, output_writer):
15     callbacks_list.append(callback_name)
```

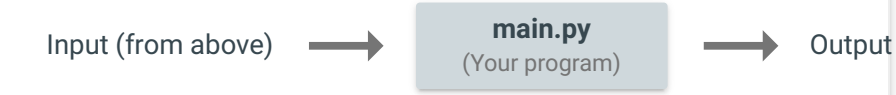
Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program



Program output displayed here

Coding trail of your work What is this?

```
10/18 W-10 T- min:6
```