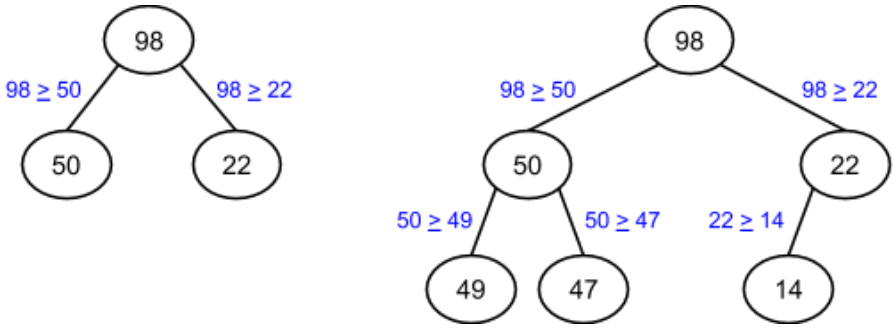
8.1 Heaps

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.

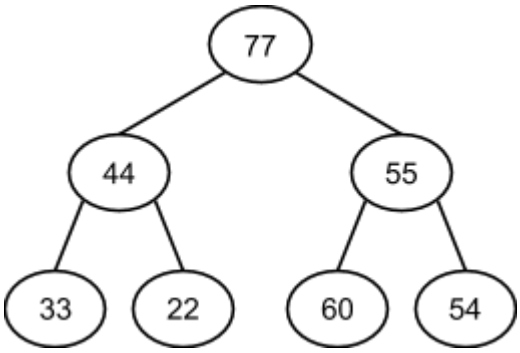
Maintaining jobs in fully-sorted order requires more operations than necessary, since only the maximum item is needed. A ***max-heap*** is a binary tree that maintains the simple property that a node's key is greater than or equal to the node's childrens' keys. (Actually, a max-heap may be any tree, 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 8.1.1: Max-heap property: A node's key is greater than or equal to the node's childrens' keys.



PARTICIPATION ACTIVITY 8.1.1: 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 sometime 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 8.1.2: 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 8.1.3: Max-heap inserts and deletes.

1) Given N nodes, what is the height of a max-heap?
   - $\lfloor logN \rfloor$
   - 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(N)
   - O($logN$)

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

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 8.1.1: Web server cache.

A ***web server*** is a computer that provides ("serves") web pages in response to Internet requests. A server may store millions of pages, on large slow hard drives. For speed, a server may keep copies of popular pages on a small fast drive known as a ***cache***, so that most accesses are fast. The cache may have room for a fraction of the pages, such as 10,000. When a page is requested that isn't in the cache and the cache is full, the server may make room by removing the page that has been cached the longest (i.e., has the oldest timestamp).

A heap provides a fast way to find the page which has the oldest timestamp, which serves as the key. Note that a complete ordering of those timestamps isn't needed; at any moment, the server only needs to know which one cached page is the oldest.

Source: By Victorgrigas (Own work), CC BY-SA 3.0 http://creativecommons.org/licenses/by-sa/3.0, via Wikimedia Commons

PARTICIPATION ACTIVITY 8.1.4: Web server cache example.

1) Assuming a web server cache holds 10,000 pages and uses a heap to find the oldest page, the time for an insertion or removal is proportional to what number?
   - ○ 10,000
   - ○ 13

2) Assume that a timer's output time is forever increasing, and when a page

enters a web server cache, a node is inserted into a heap having the current time as the key. To quickly find the oldest page, is a max-heap or min-heap needed?

- max-heap
- min-heap

How was this section? Provide feedback

8.2 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.

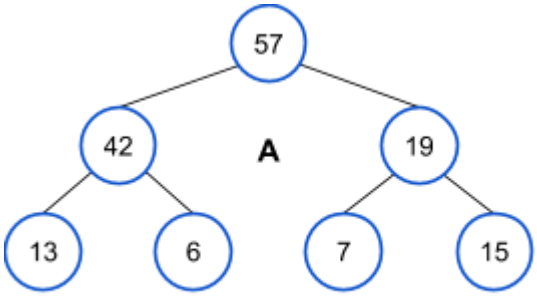
PARTICIPATION ACTIVITY 8.2.1: Max-heap stored using an array.

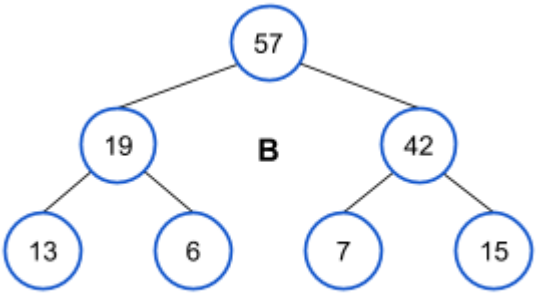
Animation captions:

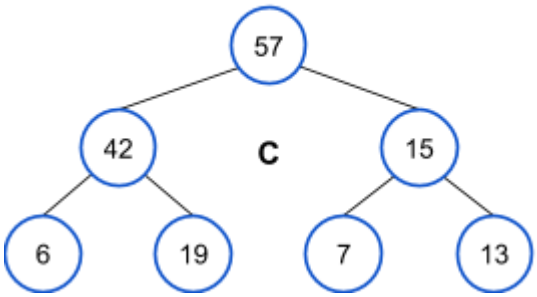
1. The max-heap's array form is produced by traversing levels left to right and top to bottom.
2. When 63 is inserted, the percolate-up operation happens within the array.

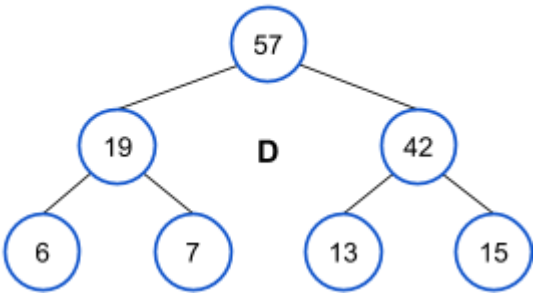
PARTICIPATION ACTIVITY 8.2.2: Heap storage.

Match each max-heap to the corresponding storage array.









Heap A Heap C Heap B Heap D

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 8.2.1: 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	floor((i - 1) / 2)	2 * i + 1, 2 * i + 2

PARTICIPATION ACTIVITY 8.2.3: 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 does not work on the root node.
   - True
   - False

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

node.

- True
- False

5) Heap implementations must check to see if a node is a leaf node before using the child index formula.

- 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 8.2.1: 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 8.2.2: 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 8.2.4: 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

How was this section? Provide feedback

8.3 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 8.3.1: 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 contai 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 floor(N / 2) - 1.

Table 8.3.1: 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	floor(N / 2) - 1

PARTICIPATION ACTIVITY 8.3.2: 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 8.3.3: Heapify operation - critical thinking.

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

True

- False
- 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 8.3.4: 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 8.3.5: 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 8.3.1: 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 8.3.6: 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

How was this section? Provide feedback

8.4 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 ***push*** 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 ***pop*** operation removes and returns the item at the front of the queue, which has the highest priority.

PARTICIPATION ACTIVITY 8.4.1: Priority queue push and pop.

Animation content:

undefined

Animation captions:

1. Pushing a single item with priority 7 initializes the priority queue with 1 item.
2. If a lower numerical value indicates higher priority, pushing 11 adds the item to the end of th queue.
3. Since 5 < 7, pushing 5 puts the item at the priority queue's front.
4. When pushing items of equal priority, the first-in-first-out rules apply. The 2nd item with priority 7 comes after the first.
5. Popping removes from the front of the queue, which is always the highest priority item.

PARTICIPATION ACTIVITY 8.4.2: Priority queue push and pop.

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 pushed?
   - ○ Before 54
   - ○ After 54
   - ○ After 86

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

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

Common priority queue operations

In addition to push and pop, 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 8.4.1: Common priority queue ADT operations.

Operation	Description	Example starting with priority queue: 42, 61, 98 (front is 42)
Push(PQueue, x)	Inserts x after all equal or higher priority items	Push(PQueue, 87). PQueue: 42, 61, 87, 98
Pop(PQueue)	Returns and removes the item at the front of PQueue	Pop(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 8.4.3: 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 Pop(servicePQueue) return?
   - 5
   - 11
   - 55

3) After popping an item, what will Peek(servicePQueue) return?

- 11
- 22
- 33

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

- -1
- 0
- Undefined

Pushing 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 ***PushWithPriority***: A push operation that includes an argument for the pushed item's priority.

PARTICIPATION ACTIVITY 8.4.4: Priority queue PushWithPriority operation.

Animation content:

undefined

Animation captions:

1. PushWithPriority calls push objects A, B, and C into the priority queue with the specified priorities.
2. In this implementation, the objects pushed into the queue do not have data members representing priority.
3. Priorities specified during PushWithPriority calls are stored alongside the queue's objects.

PARTICIPATION ACTIVITY 8.4.5: PushWithPriority operation.

1) A priority queue implementation that

requires objects to have a data member storing priority would implement the _____ function.

- Push
- PushWithPriority

2) A priority queue implementation that does not require objects to have a data member storing priority would implement the _____ function.

- Push
- PushWithPriority

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($logN$) time.

Table 8.4.2: Implementing priority queues with heaps.

Priority queue operation	Heap functionality used to implement operation	Worst-case runtime complexity
Push	Insert	O($logN$)
Pop	Remove	O($logN$)
Peek	Return value in root node	O(1)
IsEmpty	Return true if no nodes in heap, false otherwise	O(1)
GetLength	Return number of nodes (expected to be stored in the heap's member data)	O(1)

PARTICIPATION ACTIVITY 8.4.6: Implementing priority queues with heaps.

1) The Pop 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($logN$).
   - 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

How was this section? **Provide feedback**

8.5 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($logN$). 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:

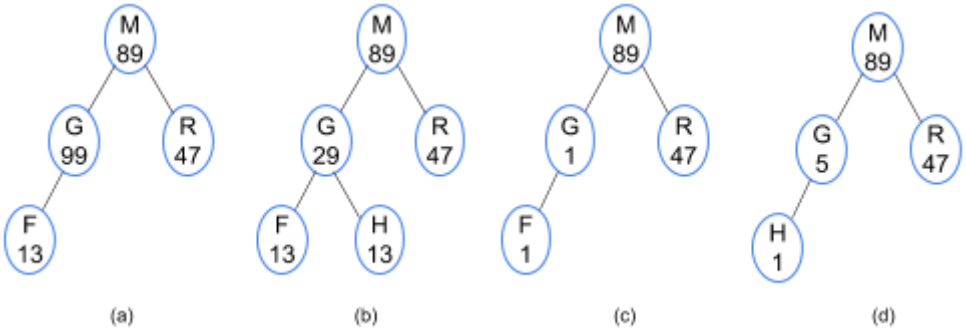
- A treap ***search*** is the same as a BST search using the main key, since the treap is a BST.
- A treap ***insert*** initially inserts a node as in a BST using the main key, then assigns a random priority to the node, and percolates the node up until the heap property is not violated. In a heap, a 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 8.5.1: 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 8.5.2: Recognizing treaps.



1) (a)

- Treap
- Not a treap

2) (b)

- Treap
- Not a treap

3) (c)

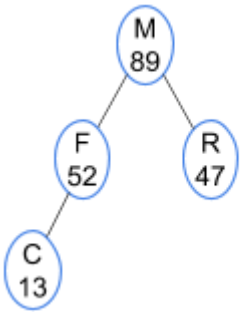
- Treap
- Not a treap

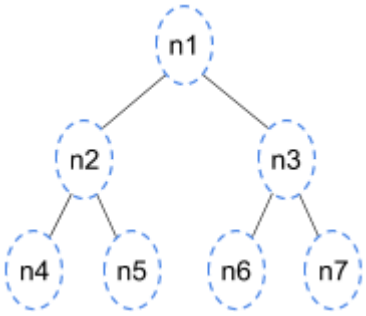
4) (d)

- Treap
- Not a treap

PARTICIPATION ACTIVITY 8.5.3: 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?

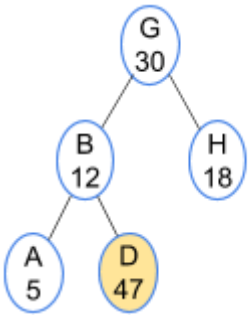
Check Show answer

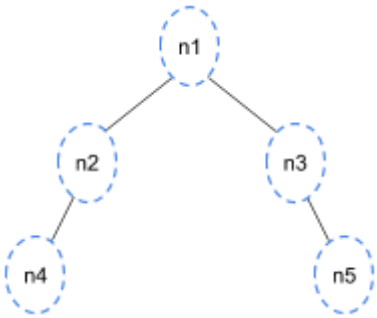
4) P is assigned a random priority of 65. To where does P percolate?

Check Show answer

PARTICIPATION ACTIVITY 8.5.4: 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.





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

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.

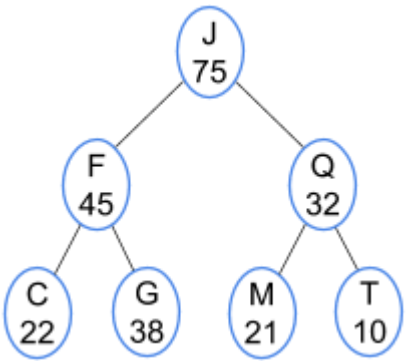
PARTICIPATION ACTIVITY 8.5.5: Treap delete: Set priority such that node must become a leaf, then percolate down using rotations.

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 8.5.6: 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 8.5.7: 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

How was this section? **Provide feedback**