Heaps and Priority Queues

Saikrishna Arcot
M. Hudachek-Buswell

July 18, 2020

Heaps

- Sometimes, we want to have some way of keeping a set of items in ascending or descending order, so that we can quickly get either the largest or smallest item.

Heaps

- Sometimes, we want to have some way of keeping a set of items in ascending or descending order, so that we can quickly get either the largest or smallest item.
- This can be trivially done with a list, but insertion will be $O(n)$ in the worst case.

Heaps

- Sometimes, we want to have some way of keeping a set of items in ascending or descending order, so that we can quickly get either the largest or smallest item.
- This can be trivially done with a list, but insertion will be $O(n)$ in the worst case.
- Instead of using a list, a data structure known as a *binary heap*, can be used.

Heaps

- Heaps are commonly represented as a binary tree, where the root is either the largest or smallest item in the heap.

Heaps

- Heaps are commonly represented as a binary tree, where the root is either the largest or smallest item in the heap.
- We call this type of heap, a binary heap.

Heaps

- Heaps are commonly represented as a binary tree, where the root is either the largest or smallest item in the heap.
- We call this type of heap, a binary heap.
- A heap where the root is the largest item is called a maxheap.

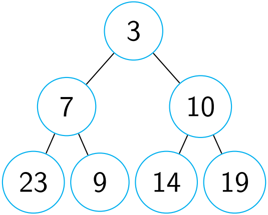
Heaps

- Heaps are commonly represented as a binary tree, where the root is either the largest or smallest item in the heap.
- We call this type of heap, a binary heap.
- A heap where the root is the largest item is called a maxheap.
- A heap where the root is the smallest item is called a minheap.

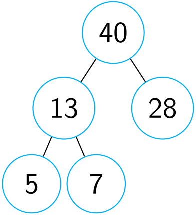
Heaps

- Heaps are commonly represented as a binary tree, where the root is either the largest or smallest item in the heap.
- We call this type of heap, a binary heap.
- A heap where the root is the largest item is called a maxheap.
- A heap where the root is the smallest item is called a minheap.
- Heaps can also be represented as an array.

Example Heaps



minheap



maxheap

Properties of a Heap

Heaps have two properties:

Order Property In a heap, the parent node is always larger (in the case of a maxheap) or smaller (in the case of a minheap) than its children. There is no relationship between the children.

Properties of a Heap

Heaps have two properties:

Order Property In a heap, the parent node is always larger (in the case of a maxheap) or smaller (in the case of a minheap) than its children. There is no relationship between the children.

Shape Property A heap is always complete. At any point, each level of the tree is completely filled, except for the last level which may be filled from left to right with no gaps.

Properties of a Heap

Heaps have two properties:

Order Property In a heap, the parent node is always larger (in the case of a maxheap) or smaller (in the case of a minheap) than its children. There is no relationship between the children.

Shape Property A heap is always complete. At any point, each level of the tree is completely filled, except for the last level which may be filled from left to right with no gaps.

Properties of a Heap

Heaps have two properties:

Order Property In a heap, the parent node is always larger (in the case of a maxheap) or smaller (in the case of a minheap) than its children. There is no relationship between the children.

Shape Property A heap is always complete. At any point, each level of the tree is completely filled, except for the last level which may be filled from left to right with no gaps.

In addition, only the root (the largest or the smallest) item of a heap can be accessed; you cannot search through a heap.

Binary Heap as an Array

- The properties of a heap allow us a clever way to implement heaps using arrays.

Binary Heap as an Array

- The properties of a heap allow us a clever way to implement heaps using arrays.
- The “root” of the heap would be in index 1 of the array. (Index 0 is skipped to make the math easier.)

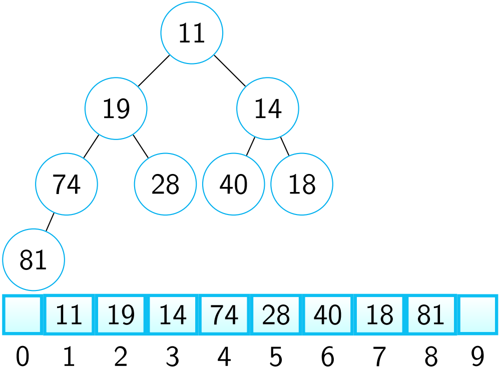
Binary Heap as an Array

- The properties of a heap allow us a clever way to implement heaps using arrays.
- The "root" of the heap would be in index 1 of the array. (Index 0 is skipped to make the math easier.)
- The left child of an item in index i would then be at index $2i$, and the right child would be at index $2i + 1$.

Binary Heap as an Array

- The properties of a heap allow us a clever way to implement heaps using arrays.
- The "root" of the heap would be in index 1 of the array. (Index 0 is skipped to make the math easier.)
- The left child of an item in index i would then be at index $2i$, and the right child would be at index $2i + 1$.
- The parent of an item in index i would be at $\frac{i}{2}$ (this is assuming you are doing integer division).

Binary Heap as an Array



Adding

- Add the new item into the next open slot in the tree/array. This may break the order property, which is fine because we will fix it.

Adding

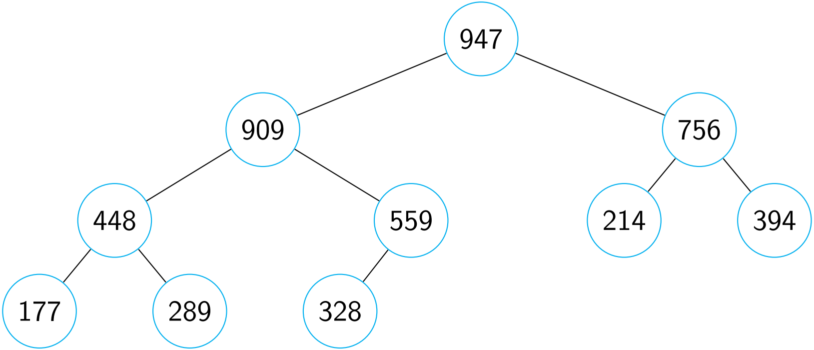
- Add the new item into the next open slot in the tree/array. This may break the order property, which is fine because we will fix it.
- Compare the newly added item to its the parent. If the order property is broken (i.e. the parent is larger than the child, but it is supposed to be a minheap, or the parent is smaller than the child, but it is supposed to be a maxheap), swap the parent and child. This is referred to as *heapify*, and the algorithm is recursive.

Adding

- Add the new item into the next open slot in the tree/array. This may break the order property, which is fine because we will fix it.
- Compare the newly added item to its the parent. If the order property is broken (i.e. the parent is larger than the child, but it is supposed to be a minheap, or the parent is smaller than the child, but it is supposed to be a maxheap), swap the parent and child. This is referred to as *heapify*, and the algorithm is recursive.
- Repeat the previous step until: 1) you don't make a swap, or 2) you reach the root of the heap.

Adding

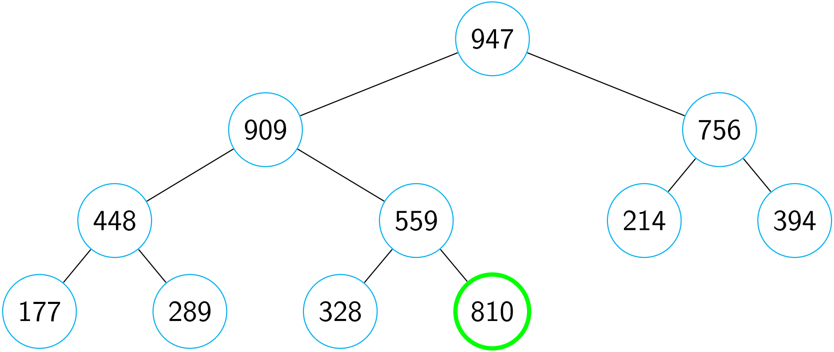
Example: Adding 810 into the heap below. Which heap is it? What is the size of the heap? Where do we add the new node? What index?



(Maxheap of size 10. Add to the right of 559, index 11)

Adding

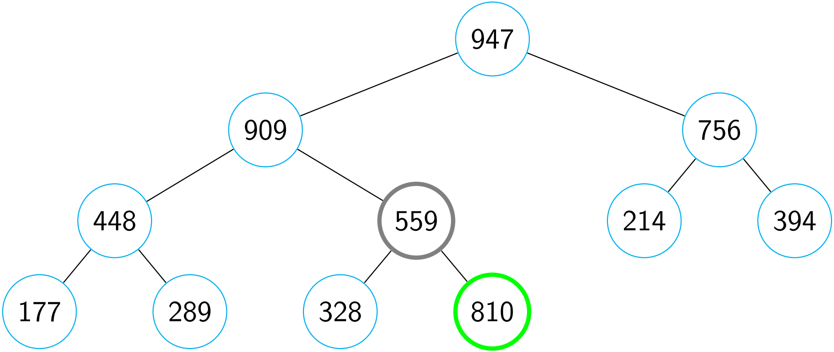
Example: Adding 810 into the heap below.



Size of the heap is 11. The order property is violated for parent node of 810.

Adding

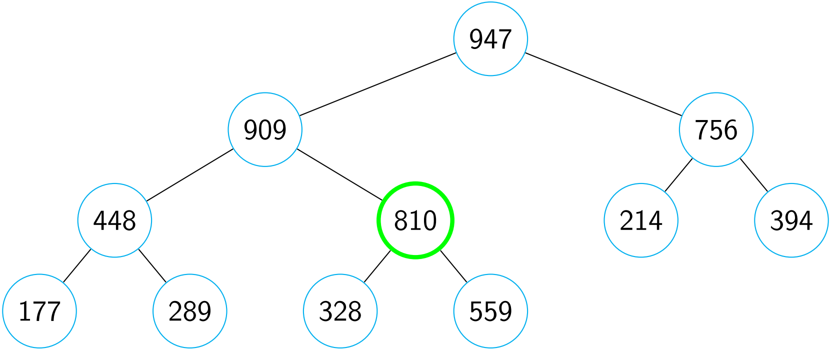
Example: Adding 810 into the heap below.



Need to restore order by heapifying parent and child.

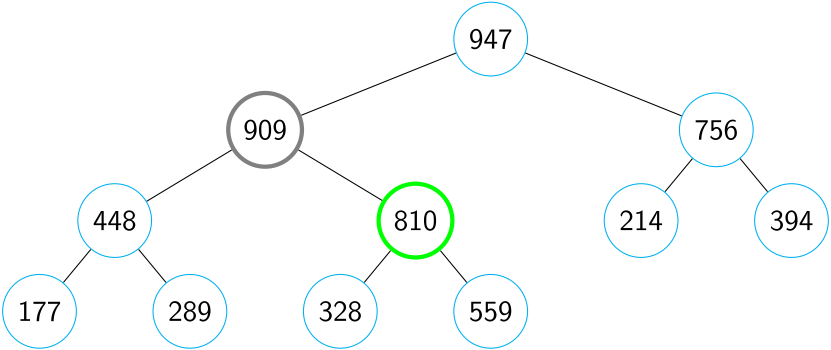
Adding

Example: Adding 810 into the heap below.



Adding

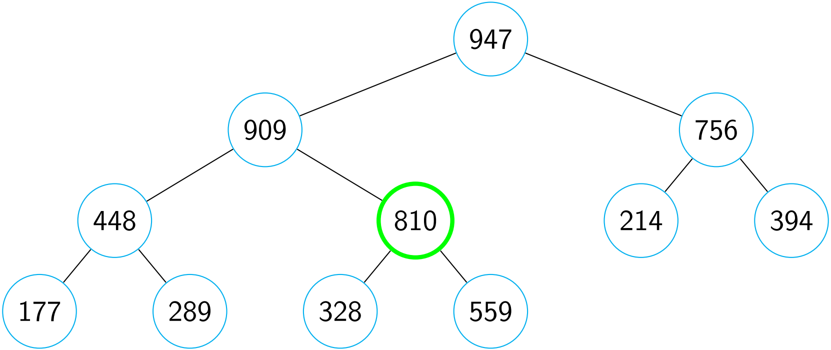
Example: Adding 810 into the heap below.



Now check order property again.

Adding

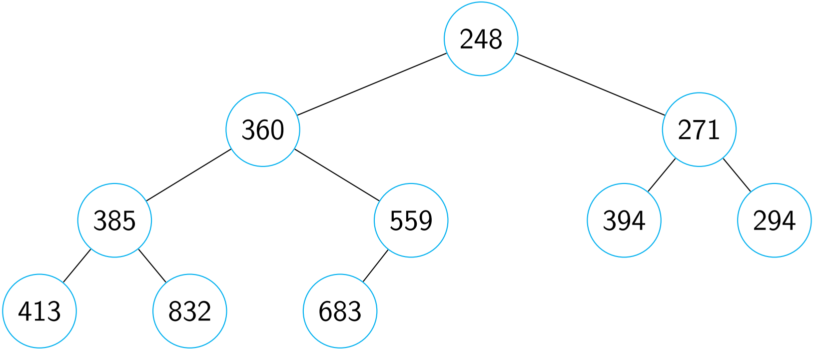
Example: Adding 810 into the heap below.



Order property is intact so the add is complete.

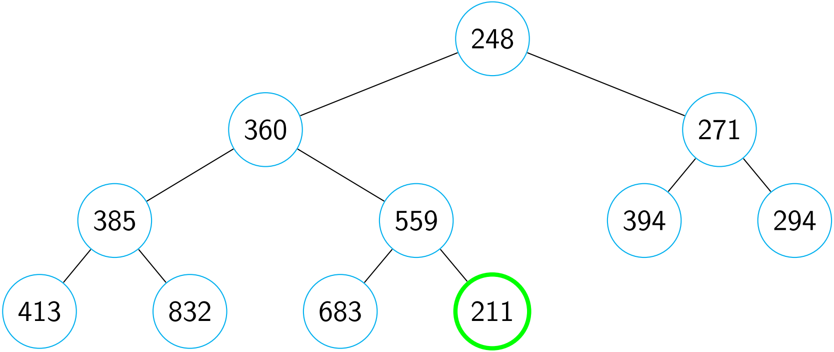
Adding

Example: Adding 211 into the minheap



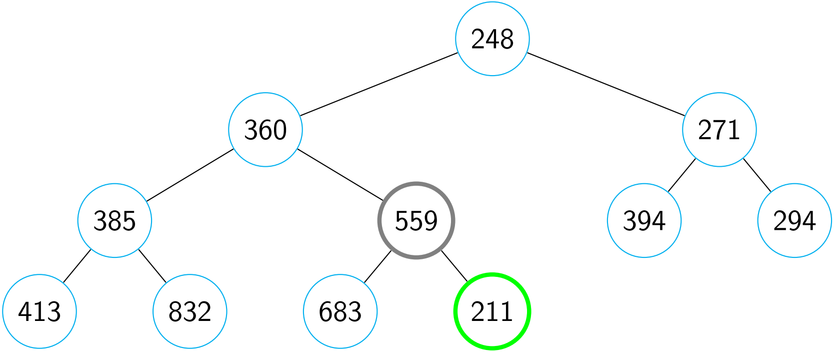
Adding

Example: Adding 211 into the minheap



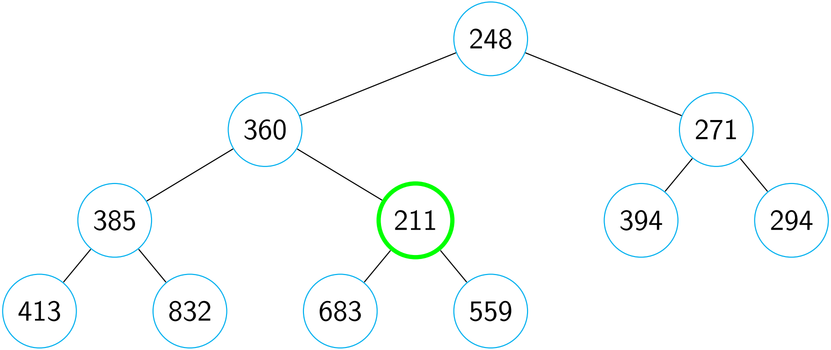
Adding

Example: Adding 211 into the minheap



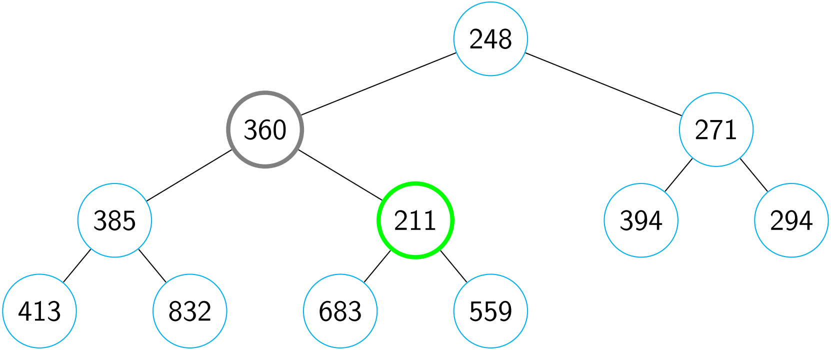
Adding

Example: Adding 211 into the minheap



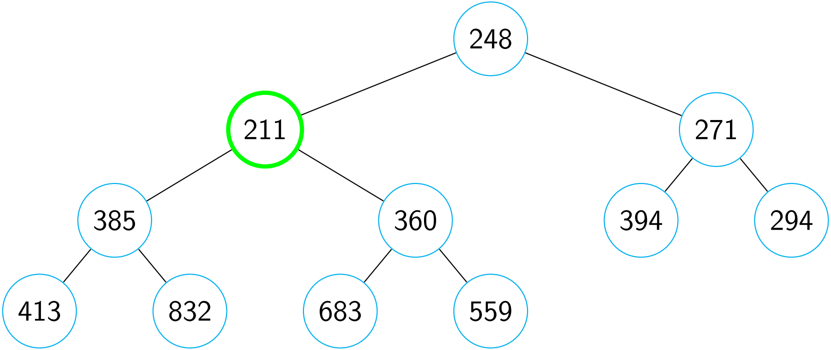
Adding

Example: Adding 211 into the minheap



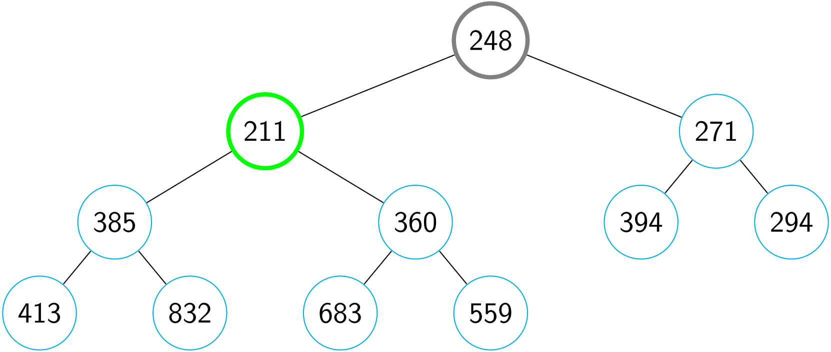
Adding

Example: Adding 211 into the minheap



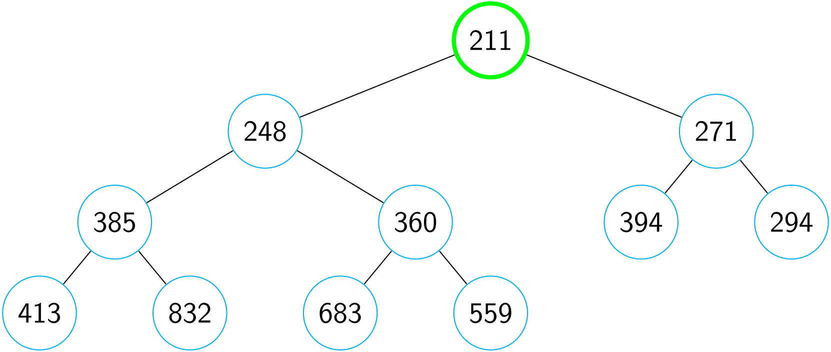
Adding

Example: Adding 211 into the minheap



Adding

Example: Adding 211 into the minheap



Adding

procedure $\textsc{Add}(\mathit{data})$

    index $\leftarrow$ next empty slot in the heap

    $\mathit{heap}[\mathit{index}] \leftarrow \mathit{data}$

    $\mathit{parentIndex} \leftarrow \mathit{index}/2$

    while $\mathit{parentIndex} >= 1$, and $\mathit{heap}[\mathit{parentIndex}]$ and $\mathit{heap}[\mathit{index}]$ are not in the correct order **do**

        Swap $\mathit{heap}[\mathit{parentIndex}]$ and $\mathit{heap}[\mathit{index}]$

        $\mathit{index} \leftarrow \mathit{parentIndex}$

        $\mathit{parentIndex} \leftarrow \mathit{parentIndex}/2$

    end while

end procedure

Removing

- You can only remove from the root.

Removing

- You can only remove from the root.
- Remove the item at the root, and move the item in the last slot of the heap to the root. This might break the order property, but that is fine.

Removing

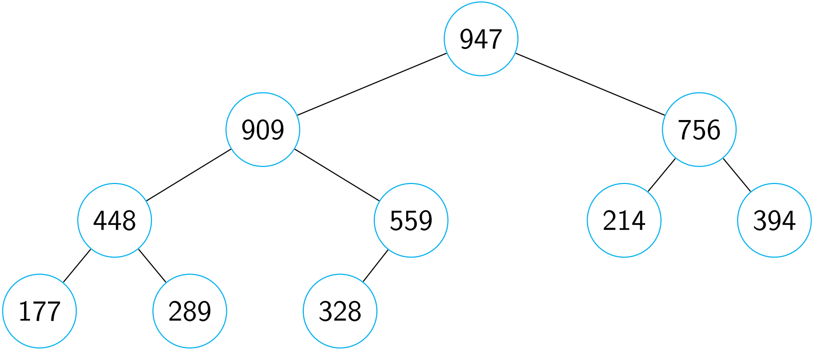
- You can only remove from the root.
- Remove the item at the root, and move the item in the last slot of the heap to the root. This might break the order property, but that is fine.
- Compare the item with the item in the left child and the item in the right child. If either the left child or the right child is smaller (in the case of a minheap) or larger (in the case of a maxheap), swap the item with the smaller/larger of the two children.

Removing

- You can only remove from the root.
- Remove the item at the root, and move the item in the last slot of the heap to the root. This might break the order property, but that is fine.
- Compare the item with the item in the left child and the item in the right child. If either the left child or the right child is smaller (in the case of a minheap) or larger (in the case of a maxheap), swap the item with the smaller/larger of the two children.
- Repeat the previous step until you don't make a swap or you reach the bottom of the heap.

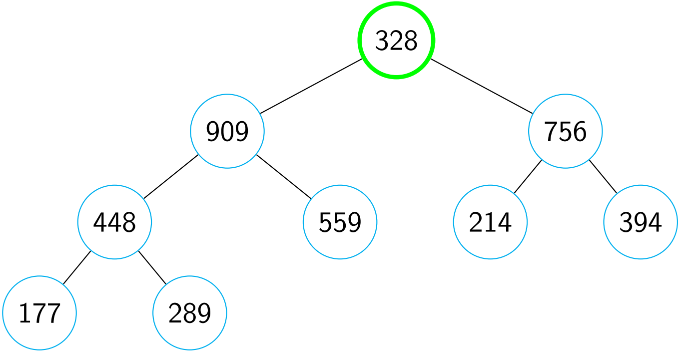
Removing

Example: Removing 947 from the *max*heap



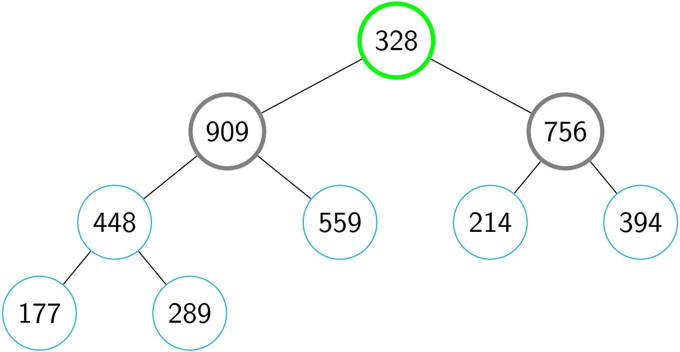
Removing

Example: Removing 947 from the *max*heap



Removing

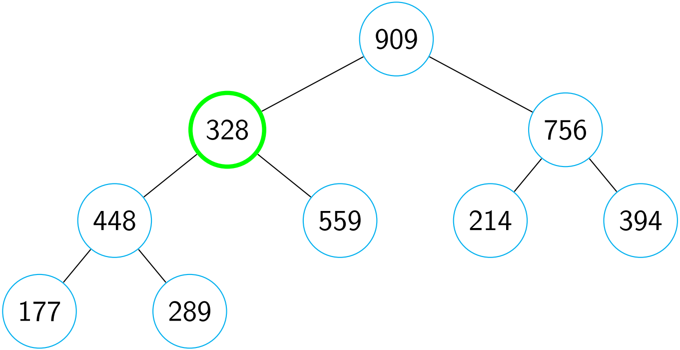
Example: Removing 947 from the *max*heap



Heapify

Removing

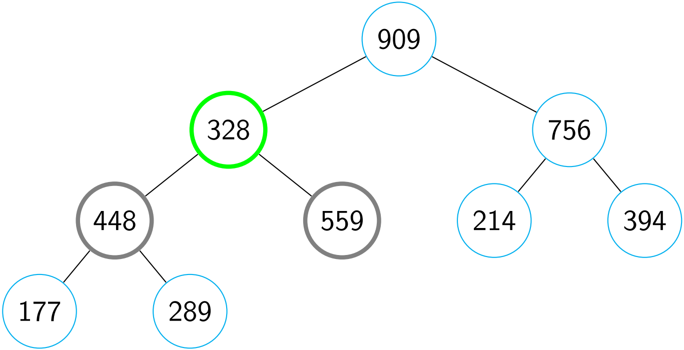
Example: Removing 947 from the *max*heap



Heapify

Removing

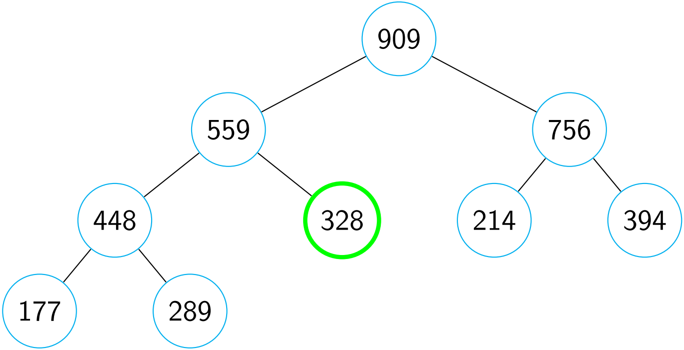
Example: Removing 947 from the *max*heap



Heapify

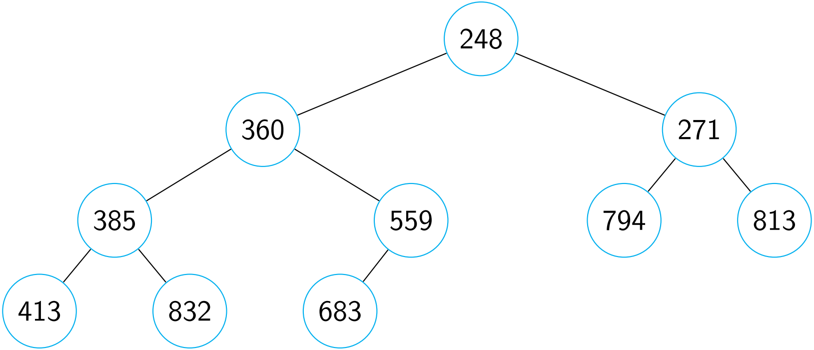
Removing

Example: Removing 947 from the *max*heap



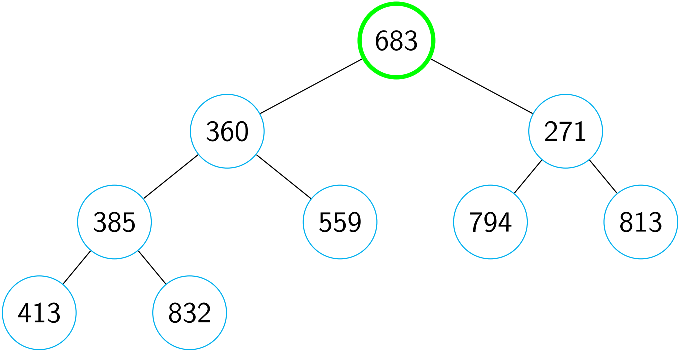
Removing

Example: Remove 248 from the minheap



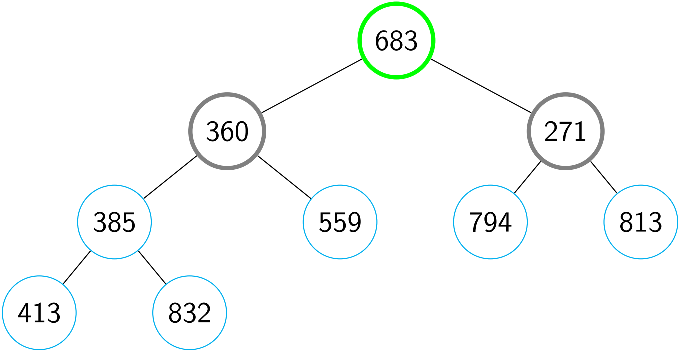
Removing

Example: Remove 248 from the minheap



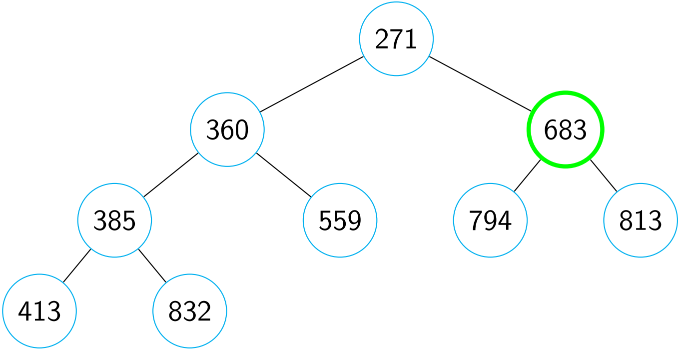
Removing

Example: Remove 248 from the minheap



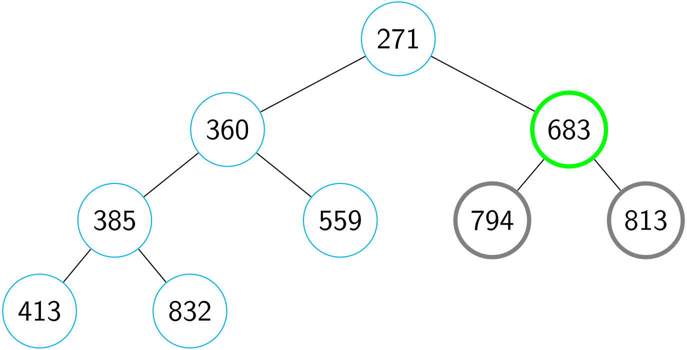
Removing

Example: Remove 248 from the minheap



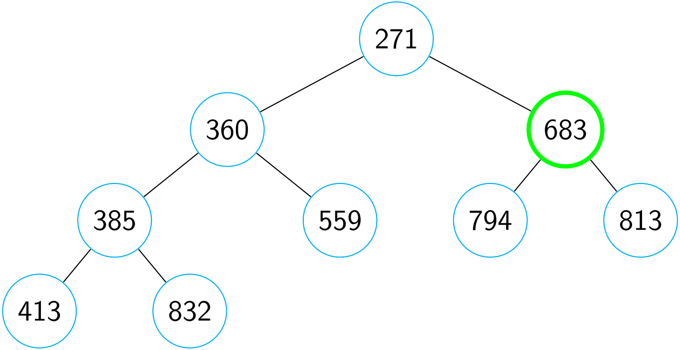
Removing

Example: Remove 248 from the minheap



Removing

Example: Remove 248 from the minheap



Removing

procedure REMOVING

index $\leftarrow$ last filled slot in the heap

data $\leftarrow$ *heap*[1]

heap[1] $\leftarrow$ *heap*[*index*]

heap[*index*] $\leftarrow$ `NULL`

index $\leftarrow$ 1

while *heap*[*index*] has a child, and *heap*[*index*] and its left and right children are not in the correct order **do**

Swap the largest/smallest child with *heap*[*index*]

index $\leftarrow$ index of the largest/smallest child

end while

return *data*

end procedure

Performance

- Adding to a heap will be $O(\log n)$ in the worst and average cases, because you may have to promote the item you just added up to the root of the heap, which would be $\log n$ steps.

Performance

- Adding to a heap will be $O(\log n)$ in the worst and average cases, because you may have to promote the item you just added up to the root of the heap, which would be $\log n$ steps.
- In the best case, however, adding to a heap could be $O(1)$ if you add the items in ascending order (for a minheap) or descending order (for a maxheap), because the item you just added will never get promoted.

Performance

- Adding to a heap will be $O(\log n)$ in the worst and average cases, because you may have to promote the item you just added up to the root of the heap, which would be $\log n$ steps.
- In the best case, however, adding to a heap could be $O(1)$ if you add the items in ascending order (for a minheap) or descending order (for a maxheap), because the item you just added will never get promoted.
- Removing from a heap will be $O(\log n)$ because the item that gets moved to the root may have to go down $\log n$ levels.

Priority Queue

- A priority queue is a data structure that, when given some items, returns the items in ascending or descening order (depending on the implementation).

Priority Queue

- A priority queue is a data structure that, when given some items, returns the items in ascending or descening order (depending on the implementation).
- Priority queues may be used in printer jobs, CPU schedulers, flight boarding, bandwidth management in routers, and as an auxillary data structure in other algorithms.

Priority Queue

- A priority queue is a data structure that, when given some items, returns the items in ascending or descening order (depending on the implementation).
- Priority queues may be used in printer jobs, CPU schedulers, flight boarding, bandwidth management in routers, and as an auxillary data structure in other algorithms.
- Priority queues can be efficiently implemented using a heap.