# Heaps

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 6

# Priority queue

- Need to maintain a collection of items with priorities to optimise the following operations
- `delete_max()`
  - Identify and remove item with highest priority
  - Need not be unique
- `insert()`
  - Add a new item to the list

# Priority queue

- Need to maintain a collection of items with priorities to optimise the following operations
- `delete_max()`
  - Identify and remove item with highest priority
  - Need not be unique
- `insert()`
  - Add a new item to the list

- Maintaining as a list incurs cost $O(N^2)$ across $N$ inserts and deletions

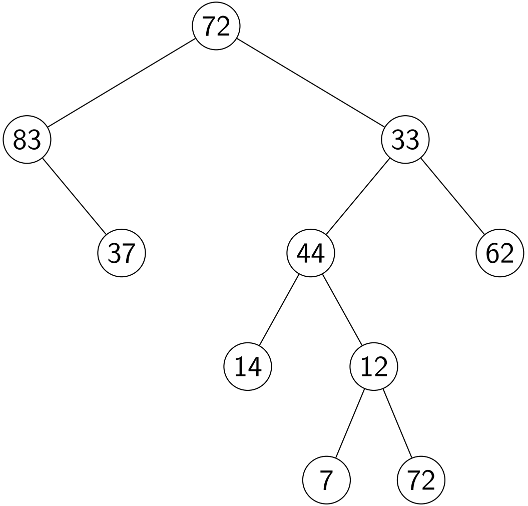
# Priority queue

- Need to maintain a collection of items with priorities to optimise the following operations
- `delete_max()`
  - Identify and remove item with highest priority
  - Need not be unique
- `insert()`
  - Add a new item to the list

- Maintaining as a list incurs cost $O(N^2)$ across $N$ inserts and deletions
- Using a $\sqrt{N} \times \sqrt{N}$ array reduces the cost to $O(\sqrt{N})$ per operations
  - $O(N\sqrt{N})$ across $N$ inserts and deletions

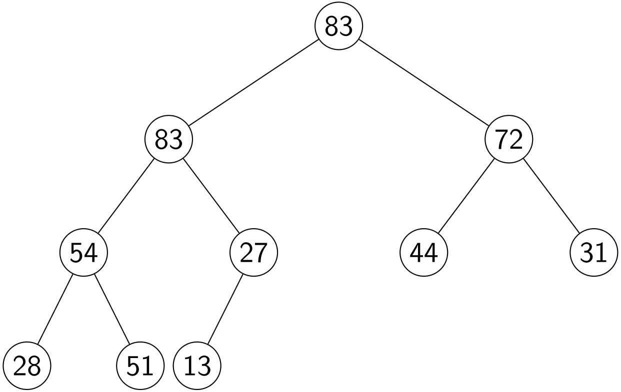
# Binary trees

- Values are stored as nodes in a rooted tree
- Each node has up to two children
  - Left child and right child
  - Order is important
- Other than the root, each node has a unique parent
- Leaf node — no children
- Size — number of nodes
- Height — number of levels



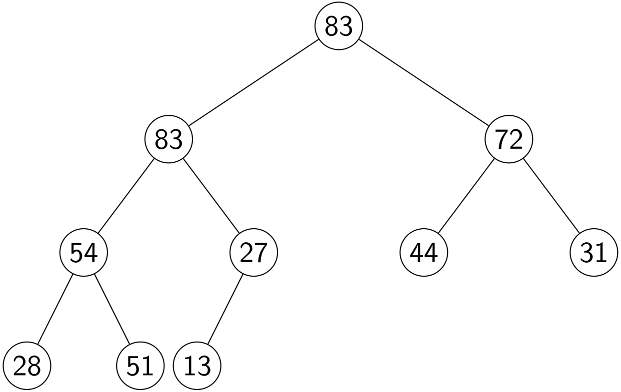
# Heap

- Binary tree filled level by level, left to right
- The value at each node is at least as big the values of its children
    - max-heap



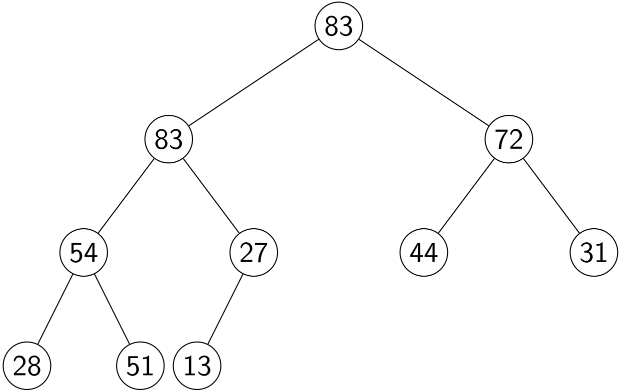
# Heap

- Binary tree filled level by level, left to right
- The value at each node is at least as big the values of its children
  - max-heap
- Binary tree on the right is an example of a heap



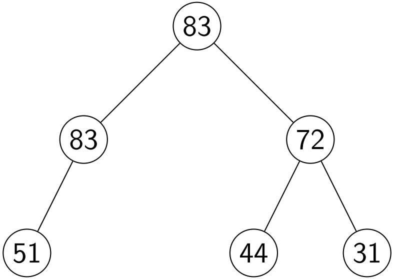
# Heap

- Binary tree filled level by level, left to right
- The value at each node is at least as big the values of its children
  - max-heap
- Binary tree on the right is an example of a heap
- Root always has the largest value
  - By induction, because of the max-heap property

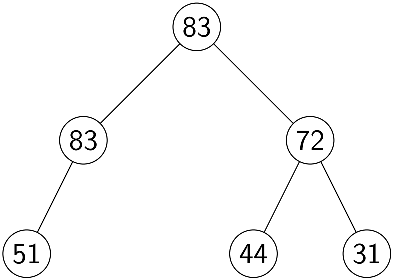


# Non-examples

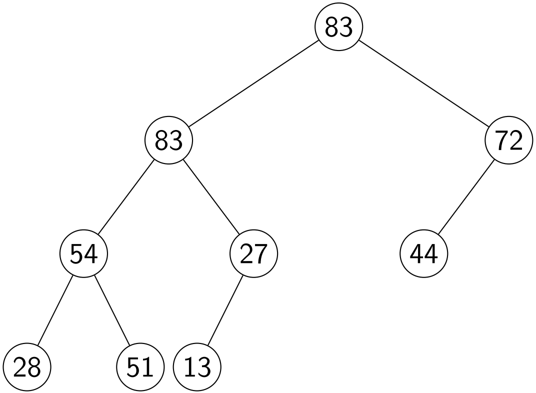
No "holes" allowed



No "holes" allowed

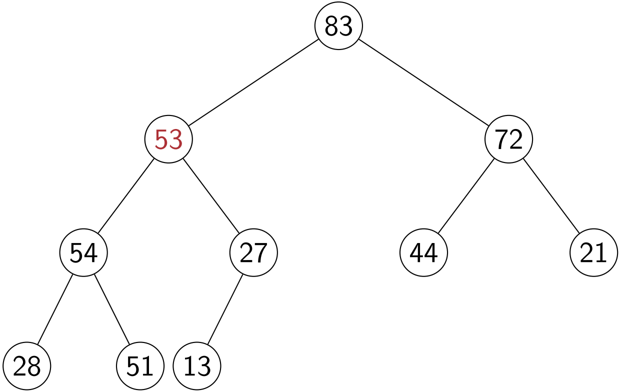



Cannot leave a level incomplete



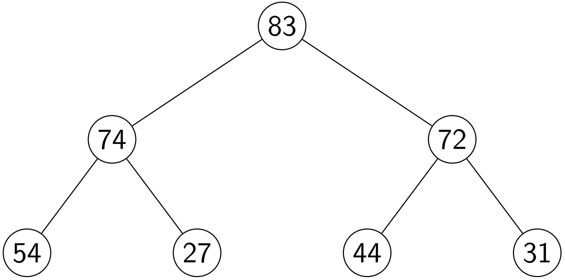
# Non-examples

Heap property is violated



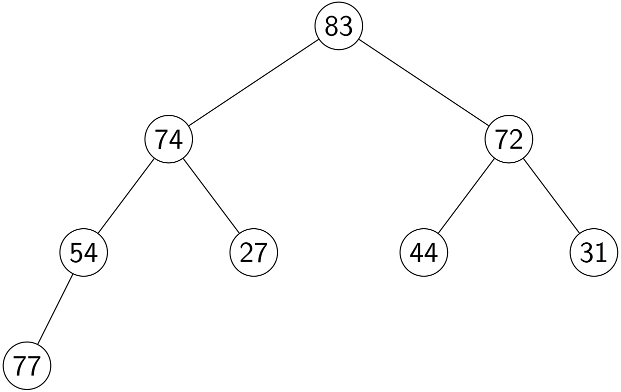
# insert()

- `insert(77)`



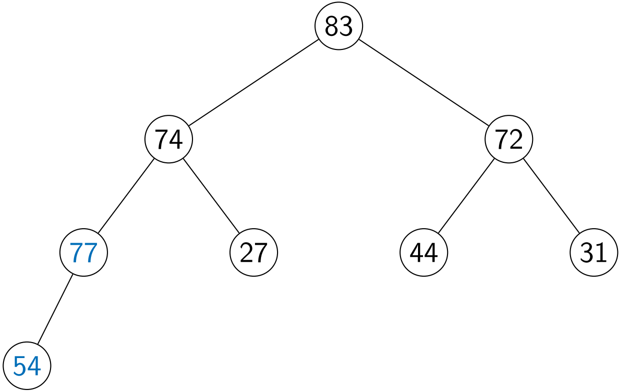
# insert()

- `insert(77)`
- Add a new node at dictated by heap structure



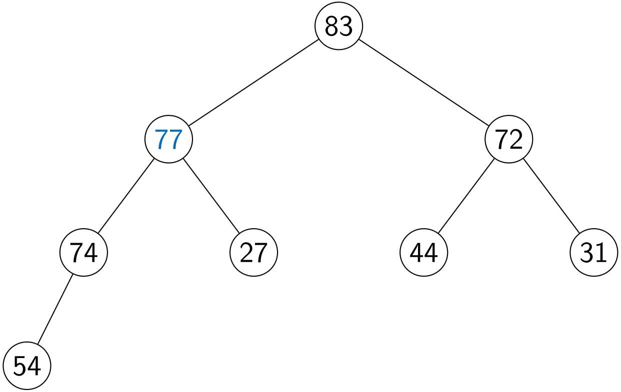
# insert()

- `insert(77)`
- Add a new node at dictated by heap structure
- Restore the heap property along path to the root



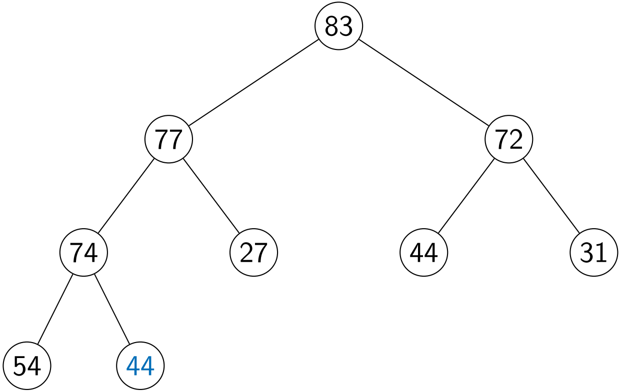
# insert()

- insert(77)
- Add a new node at dictated by heap structure
- Restore the heap property along path to the root



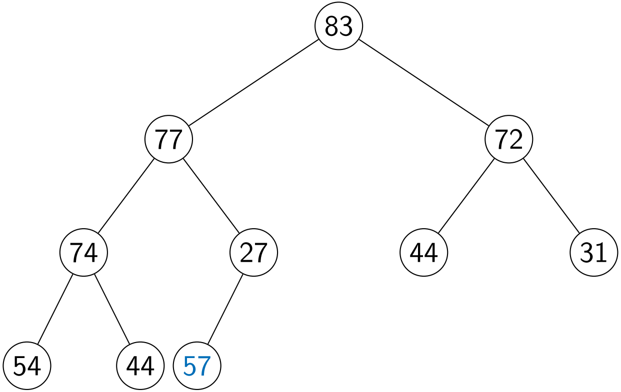
# insert()

- `insert(77)`
- Add a new node at dictated by heap structure
- Restore the heap property along path to the root
- `insert(44)`



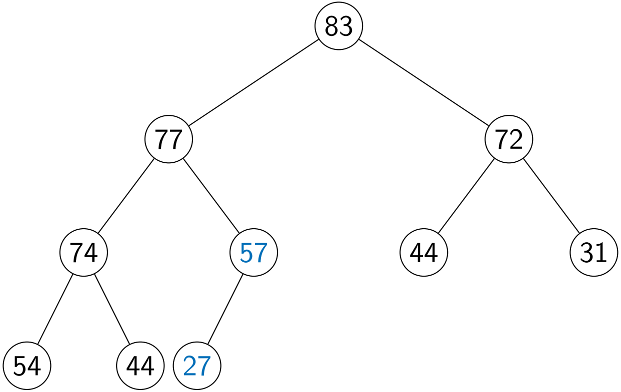
# insert()

- `insert(77)`
- Add a new node at dictated by heap structure
- Restore the heap property along path to the root
- `insert(44)`
- `insert(57)`



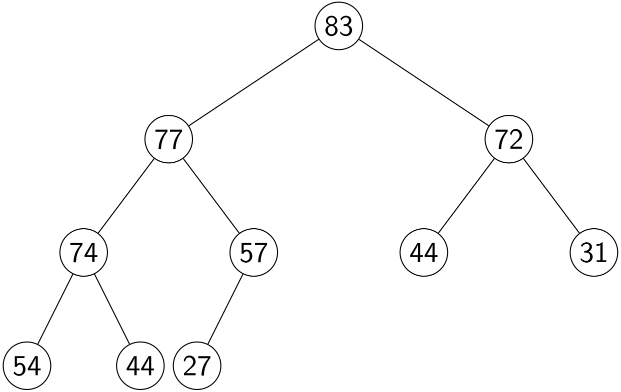
# insert()

- `insert(77)`
- Add a new node at dictated by heap structure
- Restore the heap property along path to the root
- `insert(44)`
- `insert(57)`



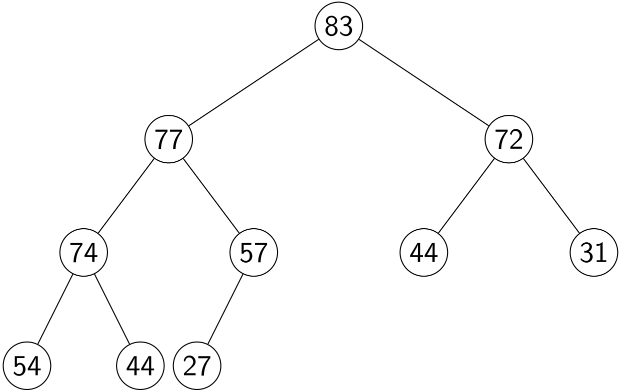
# Complexity of `insert()`

- Need to walk up from the leaf to the root
  - Height of the tree



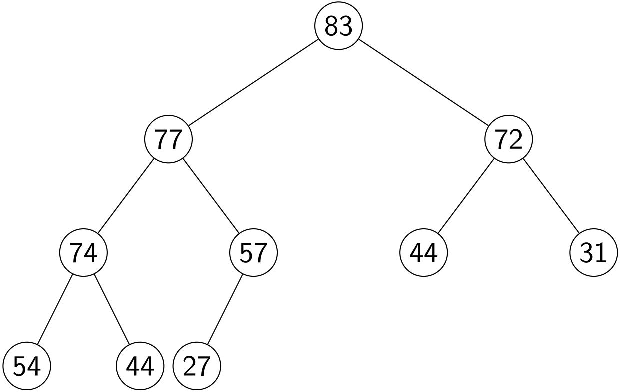
# Complexity of `insert()`

- Need to walk up from the leaf to the root
  - Height of the tree
- Number of nodes at level $0$ is $2^0 = 1$



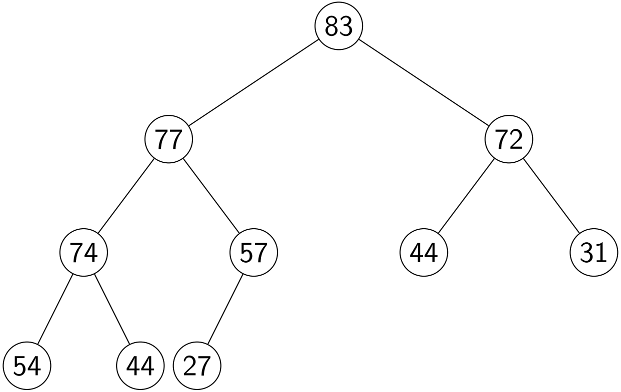
# Complexity of insert()

- Need to walk up from the leaf to the root
  - Height of the tree
- Number of nodes at level $0$ is $2^0 = 1$
- Number of nodes at level $j$ is $2^j$



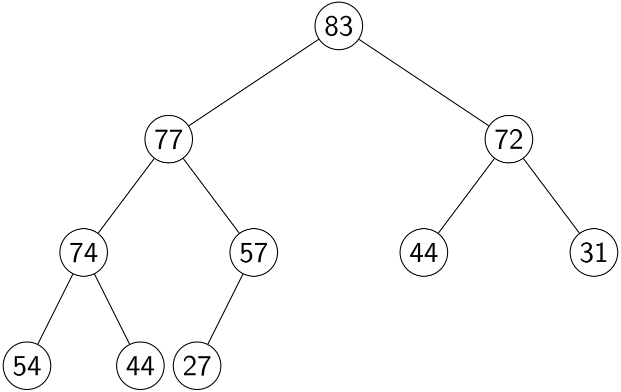
# Complexity of `insert()`

- Need to walk up from the leaf to the root
  - Height of the tree
- Number of nodes at level $0$ is $2^0 = 1$
- Number of nodes at level $j$ is $2^j$
- If we fill $k$ levels, $2^0 + 2^1 + \cdots + 2^{k-1} = 2^k - 1$ nodes



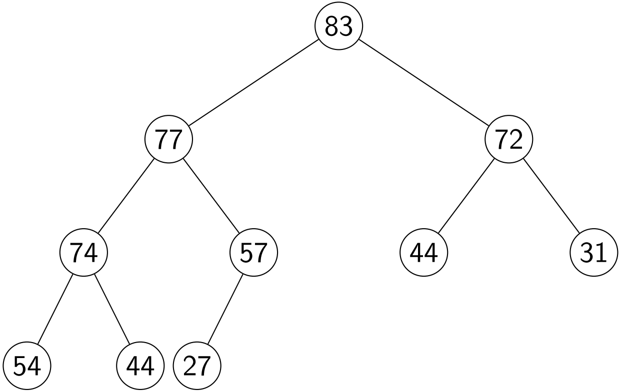
# Complexity of `insert()`

- Need to walk up from the leaf to the root
  - Height of the tree
- Number of nodes at level $0$ is $2^0 = 1$
- Number of nodes at level $j$ is $2^j$
- If we fill $k$ levels, $2^0 + 2^1 + \cdots + 2^{k-1} = 2^k - 1$ nodes
- If we have $N$ nodes, at most $1 + \log N$ levels



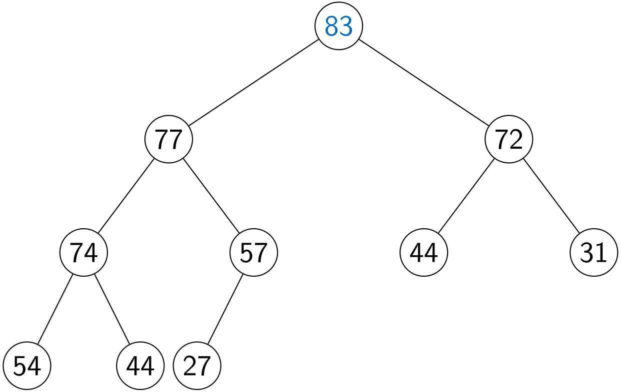
# Complexity of `insert()`

- Need to walk up from the leaf to the root
  - Height of the tree
- Number of nodes at level $0$ is $2^0 = 1$
- Number of nodes at level $j$ is $2^j$
- If we fill $k$ levels, $2^0 + 2^1 + \cdots + 2^{k-1} = 2^k - 1$ nodes
- If we have $N$ nodes, at most $1 + \log N$ levels
- `insert()` is $O(\log N)$



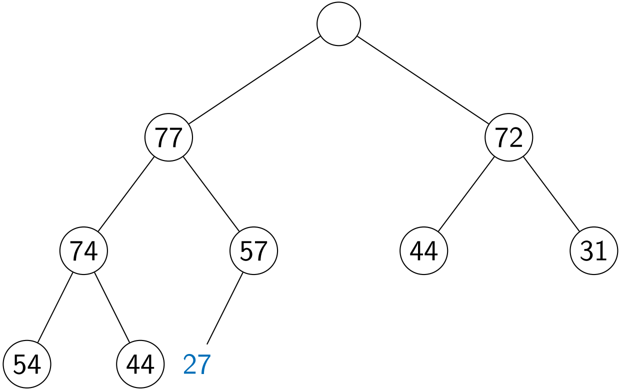
# delete_max()

- Maximum value is always at the root



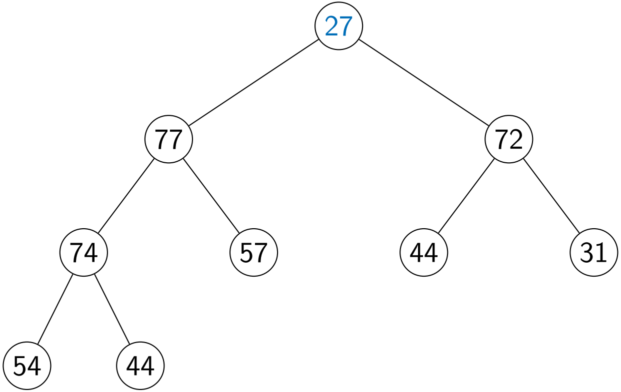
# delete_max()

- Maximum value is always at the root
- After we delete one value, tree shrinks
  - Node to delete is rightmost at lowest level



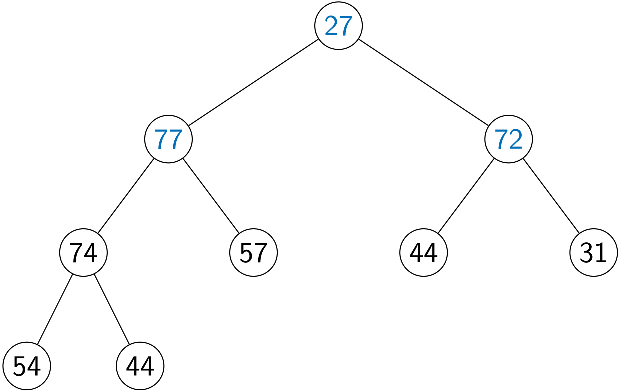
# delete_max()

- Maximum value is always at the root
- After we delete one value, tree shrinks
  - Node to delete is rightmost at lowest level
- Move "homeless" value to the root



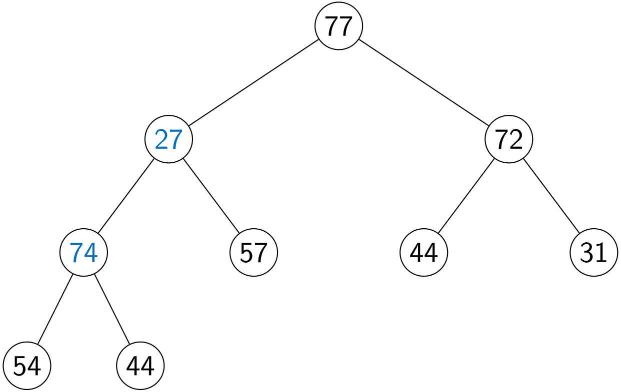
# delete_max()

- Maximum value is always at the root
- After we delete one value, tree shrinks
  - Node to delete is rightmost at lowest level
- Move "homeless" value to the root
- Restore the heap property downwards



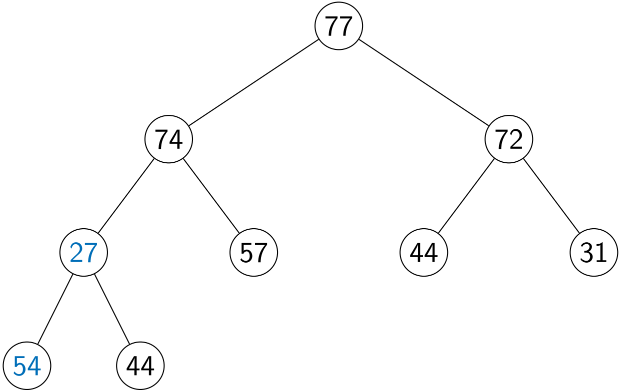
# `delete_max()`

- Maximum value is always at the root
- After we delete one value, tree shrinks
  - Node to delete is rightmost at lowest level
- Move "homeless" value to the root
- Restore the heap property downwards
- Only need to follow a single path down
  - Again $O(\log N)$



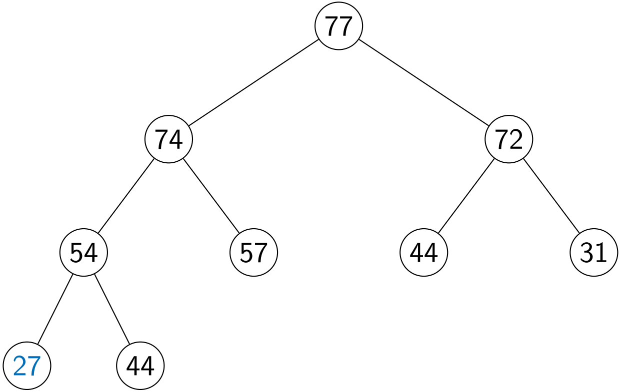
# delete_max()

- Maximum value is always at the root
- After we delete one value, tree shrinks
  - Node to delete is rightmost at lowest level
- Move "homeless" value to the root
- Restore the heap property downwards
- Only need to follow a single path down
  - Again $O(\log N)$



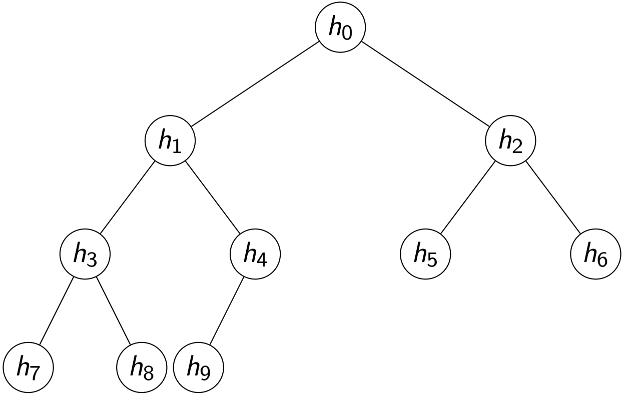
# `delete_max()`

- Maximum value is always at the root
- After we delete one value, tree shrinks
  - Node to delete is rightmost at lowest level
- Move "homeless" value to the root
- Restore the heap property downwards
- Only need to follow a single path down
  - Again $O(\log N)$



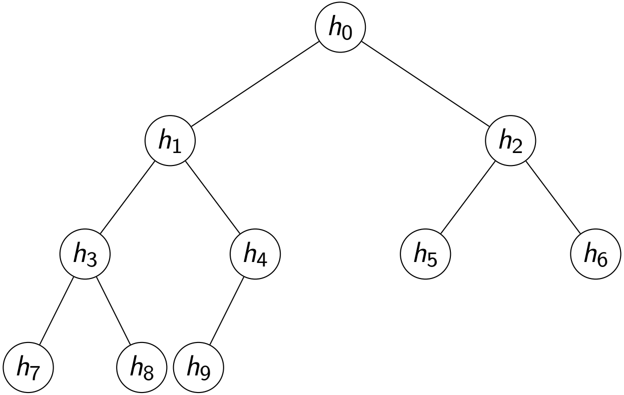
# Implementation

- Number the nodes top to bottom left right
- Store as a list
  `H = [h0,h1,h2,...,h9]`
- Children of `H[i]` are at `H[2*i+1]`, `H[2*i+2]`
- Parent of `H[i]` is at `H[(i-1)//2]`, for `i > 0`



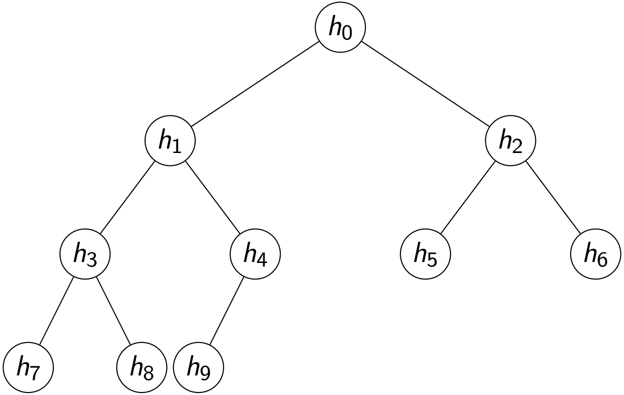
# Building a heap — `heapify()`

- Convert a list `[v0,v1,...,vN]` into a heap



# Building a heap — `heapify()`

- Convert a list `[v0,v1,...,vN]` into a heap
- Simple strategy
  - Start with an empty heap
  - Repeatedly apply `insert(vj)`
  - Total time is $O(N \log N)$



# Better `heapify()`

- List `L = [v0,v1,...,vN]`

# Better `heapify()`

- List `L = [v0,v1,...,vN]`
- `mid = len(L)//2`,
  Slice `L[mid:]` has only leaf nodes
  - Already satisfy heap condition

# Better `heapify()`

- List `L = [v0,v1,...,vN]`
- `mid = len(L)//2`,
  Slice `L[mid:]` has only leaf nodes
  - Already satisfy heap condition
- Fix heap property downwards for second last level

# Better `heapify()`

- List `L = [v0,v1,...,vN]`
- `mid = len(L)//2`,
  Slice `L[mid:]` has only leaf nodes
  - Already satisfy heap condition
- Fix heap property downwards for second last level
- Fix heap property downwards for third last level

# Better `heapify()`

- List `L = [v0,v1,...,vN]`
- `mid = len(L)//2`,
  Slice `L[mid:]` has only leaf nodes
  - Already satisfy heap condition
- Fix heap property downwards for second last level
- Fix heap property downwards for third last level

  ...
- Fix heap property at level 1
- Fix heap property at the root

# Better `heapify()`

- List `L = [v0,v1,...,vN]`
- `mid = len(L)//2`,
  Slice `L[mid:]` has only leaf nodes
  - Already satisfy heap condition
- Fix heap property downwards for second last level
- Fix heap property downwards for third last level

  ...
- Fix heap property at level 1
- Fix heap property at the root

- Each time we go up one level, one extra step per node to fix heap property

# Better `heapify()`

- List `L = [v0,v1,...,vN]`
- `mid = len(L)//2`,
  Slice `L[mid:]` has only leaf nodes
  - Already satisfy heap condition
- Fix heap property downwards for second last level
- Fix heap property downwards for third last level

  ...
- Fix heap property at level 1
- Fix heap property at the root
- Each time we go up one level, one extra step per node to fix heap property
- However, number of nodes to fix halves

# Better `heapify()`

- List `L = [v0,v1,...,vN]`
- `mid = len(L)//2`,
  Slice `L[mid:]` has only leaf nodes
  - Already satisfy heap condition
- Fix heap property downwards for second last level
- Fix heap property downwards for third last level

  ...
- Fix heap property at level 1
- Fix heap property at the root

- Each time we go up one level, one extra step per node to fix heap property
- However, number of nodes to fix halves
- Second last level, $n/4 \times 1$ steps

# Better `heapify()`

- List `L = [v0,v1,...,vN]`
- `mid = len(L)//2`,
  Slice `L[mid:]` has only leaf nodes
  - Already satisfy heap condition
- Fix heap property downwards for second last level
- Fix heap property downwards for third last level

...

- Fix heap property at level 1
- Fix heap property at the root

- Each time we go up one level, one extra step per node to fix heap property
- However, number of nodes to fix halves
- Second last level, $n/4 \times 1$ steps
- Third last level, $n/8 \times 2$ steps

# Better `heapify()`

- List `L = [v0,v1,...,vN]`
- `mid = len(L)//2`,
  Slice `L[mid:]` has only leaf nodes
  - Already satisfy heap condition
- Fix heap property downwards for second last level
- Fix heap property downwards for third last level

  ...
- Fix heap property at level 1
- Fix heap property at the root

- Each time we go up one level, one extra step per node to fix heap property
- However, number of nodes to fix halves
- Second last level, $n/4 \times 1$ steps
- Third last level, $n/8 \times 2$ steps
- Fourth last level, $n/16 \times 3$ steps

# Better `heapify()`

- List `L = [v0,v1,...,vN]`
- `mid = len(L)//2`,
  Slice `L[mid:]` has only leaf nodes
  - Already satisfy heap condition
- Fix heap property downwards for second last level
- Fix heap property downwards for third last level

…

- Fix heap property at level 1
- Fix heap property at the root

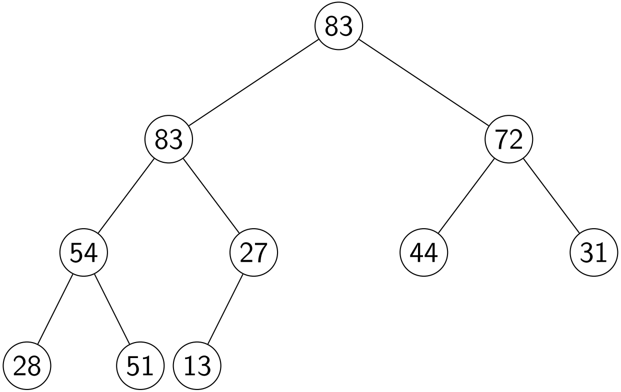
- Each time we go up one level, one extra step per node to fix heap property
- However, number of nodes to fix halves
- Second last level, $n/4 \times 1$ steps
- Third last level, $n/8 \times 2$ steps
- Fourth last level, $n/16 \times 3$ steps

…

- Cost turns out to be $O(n)$

# Summary

- Heaps are a tree implementation of priority queues
  - `insert()` is $O(\log N)$
  - `delete_max()` is $O(\log N)$
  - `heapify()` builds a heap in $O(N)$



# Summary

- Heaps are a tree implementation of priority queues
  - `insert()` is $O(\log N)$
  - `delete_max()` is $O(\log N)$
  - `heapify()` builds a heap in $O(N)$
- Can invert the heap condition
  - Each node is smaller than its children
  - min-heap
  - `delete_min()` rather than `delete_max()`