Stacks, Queues, Priority Queues, Heaps

Madhavan Mukund

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

09 December, 2021

Stacks

- Stack is a last-in, first-out list
  - `push(s,x)` — add `x` to stack `s`
  - `pop(s)` — return most recently added element
- Maintain stack as list, push and pop from the right
  - `push(s,x)` is `s.append(x)`
  - `s.pop()` — Python built-in, returns last element

Stacks

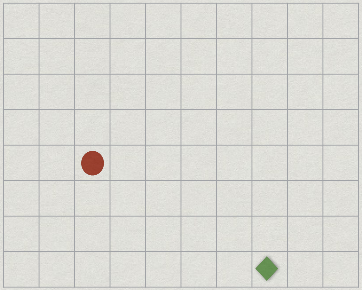
- Stacks are natural to keep track of recursive function calls
- In 8 queens, use a stack to keep track of queens added
  - Push the latest queen onto the stack
  - To backtrack, pop the last queen added

Queues

- First-in, first-out sequences
  - `addq(q,x)` — adds `x` to rear of queue `q`
  - `removeq(q)` — removes element at head of `q`
- Using Python lists, left is rear, right is front
  - `addq(q,x)` is `q.insert(0,x)`
    - `l.insert(j,x)`, insert `x` before position `j`
  - `removeq(q)` is `q.pop()`

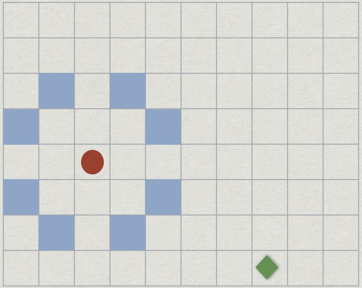
Systematic exploration

- Rectangular m x n grid
- Chess knight starts at (sx,sy)
  - Usual knight moves
- Can it reach a target square (tx,ty)?



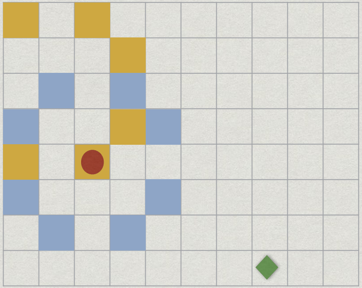
Systematic exploration

- Rectangular m x n grid
- Chess knight starts at (sx,sy)
  - Usual knight moves
- Can it reach a target square (tx,ty)?



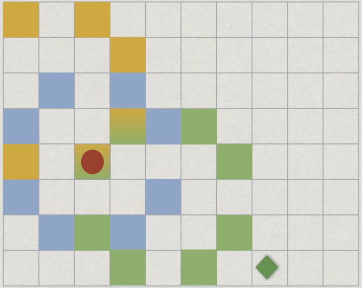
Systematic exploration

- Rectangular m x n grid
- Chess knight starts at (sx,sy)
  - Usual knight moves
- Can it reach a target square (tx,ty)? 



Systematic exploration

- Rectangular m x n grid
- Chess knight starts at (sx,sy)
  - Usual knight moves
- Can it reach a target square (tx,ty)?



Systematic exploration

- X1 — all squares reachable in one move from (sx,sy)
- X2 — all squares reachable from X1 in one move
- . . .
- Don’t explore an already marked square
- When do we stop?
  - If we reach target square
  - What if target is not reachable?

Systematic exploration

- Maintain a queue Q of cells to be explored
- Initially Q contains only start node (sx,sy)
  - Remove (ax,ay) from head of queue
  - Mark all squares reachable in one step from (ax,ay)
  - Add all newly marked squares to the queue
- When the queue is empty, we have finished

Dealing with priorities

Job scheduler

- A job scheduler maintains a list of pending jobs with their priorities

Dealing with priorities

Job scheduler

- A job scheduler maintains a list of pending jobs with their priorities
- When the processor is free, the scheduler picks out the job with maximum priority in the list and schedules it

Dealing with priorities

Job scheduler

- A job scheduler maintains a list of pending jobs with their priorities
- When the processor is free, the scheduler picks out the job with maximum priority in the list and schedules it
- New jobs may join the list at any time

Dealing with priorities

Job scheduler

- A job scheduler maintains a list of pending jobs with their priorities
- When the processor is free, the scheduler picks out the job with maximum priority in the list and schedules it
- New jobs may join the list at any time
- How should the scheduler maintain the list of pending jobs and their priorities?

Dealing with priorities

Job scheduler

- A job scheduler maintains a list of pending jobs with their priorities
- When the processor is free, the scheduler picks out the job with maximum priority in the list and schedules it
- New jobs may join the list at any time
- How should the scheduler maintain the list of pending jobs and their priorities?

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 collection

Implementing priority queues with one dimensional structures

- `delete_max()`
  - Identify and remove item with highest priority
  - Need not be unique
- `insert()`
  - Add a new item to the list

Implementing priority queues with one dimensional structures

- Unsorted list
  - `insert()` is $O(1)$
  - `delete_max()` is $O(n)$

- `delete_max()`
  - Identify and remove item with highest priority
  - Need not be unique
- `insert()`
  - Add a new item to the list

Implementing priority queues with one dimensional structures

- Unsorted list
  - `insert()` is $O(1)$
  - `delete_max()` is $O(n)$
- Sorted list
  - `delete_max()` is $O(1)$
  - `insert()` is $O(n)$

- `delete_max()`
  - Identify and remove item with highest priority
  - Need not be unique
- `insert()`
  - Add a new item to the list

Implementing priority queues with one dimensional structures

- Unsorted list
  - `insert()` is $O(1)$
  - `delete_max()` is $O(n)$
- Sorted list
  - `delete_max()` is $O(1)$
  - `insert()` is $O(n)$
- Processing n items requires $O(n^2)$

- `delete_max()`
  - Identify and remove item with highest priority
  - Need not be unique
- `insert()`
  - Add a new item to the list

Moving to two dimensions

First attempt

- Assume N processes enter/leave the queue

$N = 25$

3	19	23	35	58
12	17	25	43	67
10	13	20		
11	16	28	49	
6	14			

Moving to two dimensions

First attempt

- Assume N processes enter/leave the queue
- Maintain a $\sqrt{N} \times \sqrt{N}$ array

$N = 25$

3	19	23	35	58
12	17	25	43	67
10	13	20		
11	16	28	49	
6	14			

Moving to two dimensions

First attempt

- Assume N processes enter/leave the queue
- Maintain a $\sqrt{N} \times \sqrt{N}$ array
- Each row is in sorted order

$N = 25$

3	19	23	35	58
12	17	25	43	67
10	13	20		
11	16	28	49	
6	14			

insert()

- Keep track of the size of each row

$N = 25$

3	19	23	35	58
12	17	25	43	67
10	13	20		
11	16	28	49	
6	14			

5
5
3
4
2

insert()

- Keep track of the size of each row
- Insert into the first row that has space
  - Use size of row to determine

$N = 25$

3	19	23	35	58
12	17	25	43	67
10	13	20		
11	16	28	49	
6	14			

5
5
3
4
2

insert()

- Keep track of the size of each row
- Insert into the first row that has space
  - Use size of row to determine
- Insert 15

$N = 25$

3	19	23	35	58
12	17	25	43	67
10	13	20		
11	16	28	49	
6	14			

5
5
3
4
2

insert()

- Keep track of the size of each row
- Insert into the first row that has space
  - Use size of row to determine
- Insert 15

$N = 25$

15

3	19	23	35	58		5
12	17	25	43	67		5
10	13	20				3
11	16	28	49			4
6	14					2

insert()

- Keep track of the size of each row
- Insert into the first row that has space
  - Use size of row to determine
- Insert 15

$N = 25$

	3	19	23	35	58	5
15	12	17	25	43	67	5
	10	13	20			3
	11	16	28	49		4
	6	14				2

insert()

- Keep track of the size of each row
- Insert into the first row that has space
  - Use size of row to determine
- Insert 15

$N = 25$

	3	19	23	35	58
	12	17	25	43	67
15	10	13	20		
	11	16	28	49	
	6	14			

5
5
3
4
2

insert()

- Keep track of the size of each row
- Insert into the first row that has space
  - Use size of row to determine
- Insert 15

$N = 25$

3	19	23	35	58
12	17	25	43	67
10	13	15	20	
11	16	28	49	
6	14			

5
5
3
4
2

insert()

- Keep track of the size of each row
- Insert into the first row that has space
  - Use size of row to determine
- Insert 15

$N = 25$

3	19	23	35	58
12	17	25	43	67
10	13	15	20	
11	16	28	49	
6	14			

5
5
4
4
2

insert()

- Keep track of the size of each row
- Insert into the first row that has space
  - Use size of row to determine
- Insert 15
- Takes time $O(\sqrt{N})$
  - Scan size column to locate row to insert, $O(\sqrt{N})$
  - Insert into the first row with free space, $O(\sqrt{N})$

$N = 25$

3	19	23	35	58
12	17	25	43	67
10	13	15	20	
11	16	28	49	
6	14			

5
5
4
4
2

delete_max()

- Maximum in each row is the last element

$N = 25$

3	19	23	35	58
12	17	25	43	67
10	13	15	20	
11	16	28	49	
6	14			

5
5
4
4
2

delete_max()

- Maximum in each row is the last element
- Position is available through size column

$N = 25$

3	19	23	35	58
12	17	25	43	67
10	13	15	20	
11	16	28	49	
6	14			

5
5
4
4
2

delete_max()

- Maximum in each row is the last element
- Position is available through size column
- Identify the maximum amongst these

$N = 25$

3	19	23	35	58
12	17	25	43	67
10	13	15	20	
11	16	28	49	
6	14			

5
5
4
4
2

delete_max()

- Maximum in each row is the last element
- Position is available through size column
- Identify the maximum amongst these
- Delete it

$N = 25$

3	19	23	35	58
12	17	25	43	
10	13	15	20	
11	16	28	49	
6	14			

5
4
4
4
2

delete_max()

- Maximum in each row is the last element
- Position is available through size column
- Identify the maximum amongst these
- Delete it
- Again $O(\sqrt{N})$
  - Find the maximum among last entries, $O(\sqrt{N})$
  - Delete it, $O(1)$

$N = 25$

3	19	23	35	58
12	17	25	43	
10	13	15	20	
11	16	28	49	
6	14			

5
4
4
4
2

Summary

- 2D $\sqrt{N} \times \sqrt{N}$ array with sorted rows
  - `insert()` is $O(\sqrt{N})$
  - `delete_max()` is $O(\sqrt{N})$
  - Processing N items is $O(N\sqrt{N})$

$N = 25$

3	19	23	35	58
12	17	25	43	67
10	13	20		
11	16	28	49	
6	14			

Summary

- 2D $\sqrt{N} \times \sqrt{N}$ array with sorted rows
  - `insert()` is $O(\sqrt{N})$
  - `delete_max()` is $O(\sqrt{N})$
  - Processing N items is $O(N\sqrt{N})$
- Can we do better?

$N = 25$

3	19	23	35	58
12	17	25	43	67
10	13	20		
11	16	28	49	
6	14			

Summary

- 2D $\sqrt{N} \times \sqrt{N}$ array with sorted rows
  - `insert()` is $O(\sqrt{N})$
  - `delete_max()` is $O(\sqrt{N})$
  - Processing N items is $O(N\sqrt{N})$
- Can we do better?
- Maintain a special binary tree — heap
  - Height $O(\log N)$
  - `insert()` is $O(\log N)$
  - `delete_max()` is $O(\log N)$
  - Processing N items is $O(N \log N)$

$N = 25$

3	19	23	35	58
12	17	25	43	67
10	13	20		
11	16	28	49	
6	14			

Summary

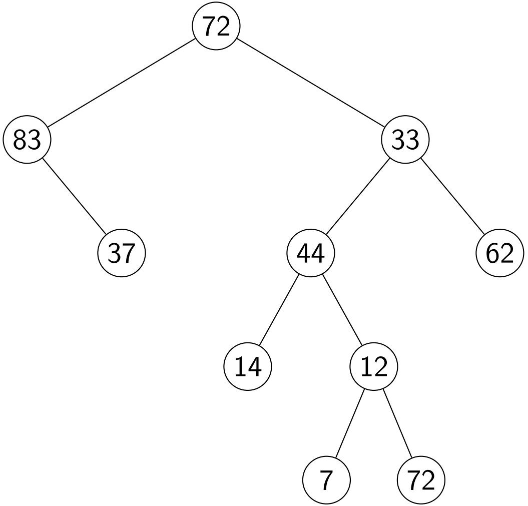
- 2D $\sqrt{N} \times \sqrt{N}$ array with sorted rows
  - `insert()` is $O(\sqrt{N})$
  - `delete_max()` is $O(\sqrt{N})$
  - Processing N items is $O(N\sqrt{N})$
- Can we do better?
- Maintain a special binary tree — heap
  - Height $O(\log N)$
  - `insert()` is $O(\log N)$
  - `delete_max()` is $O(\log N)$
  - Processing N items is $O(N \log N)$
- Flexible — need not fix N in advance

$N = 25$

3	19	23	35	58
12	17	25	43	67
10	13	20		
11	16	28	49	
6	14			

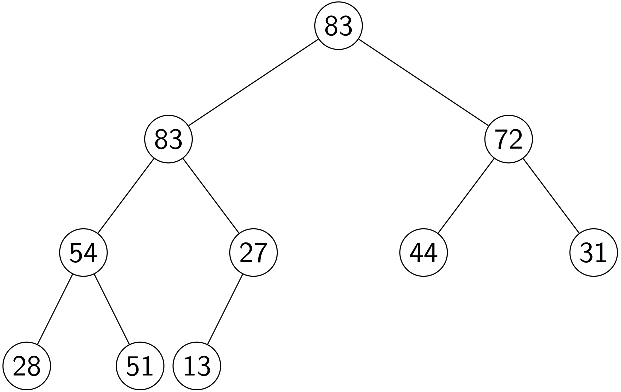
Binary trees

- 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



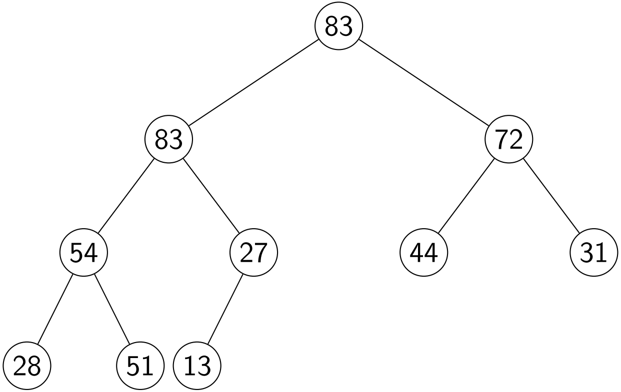
Heap

- 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



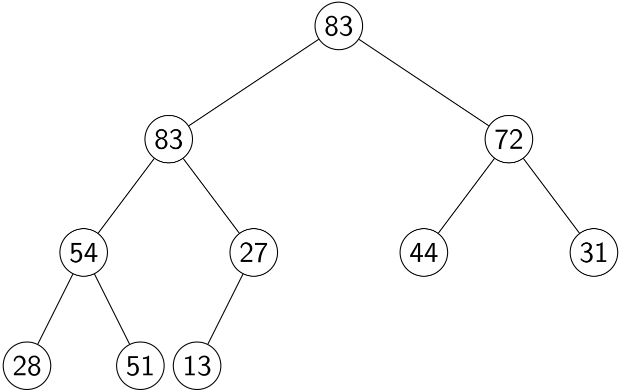
Heap

- 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



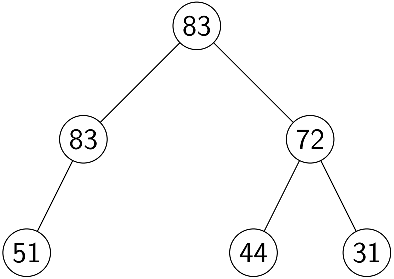
Heap

- 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



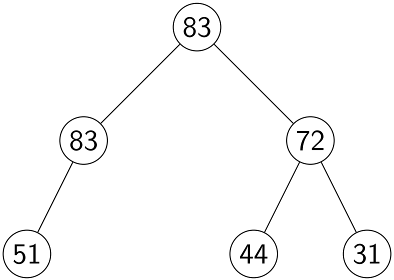
Non-examples

No “holes” allowed

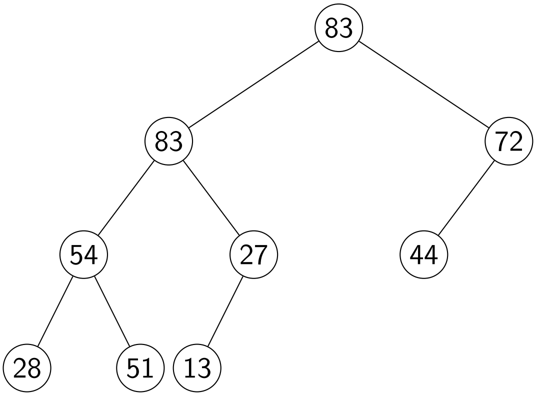


Non-examples

No "holes" allowed

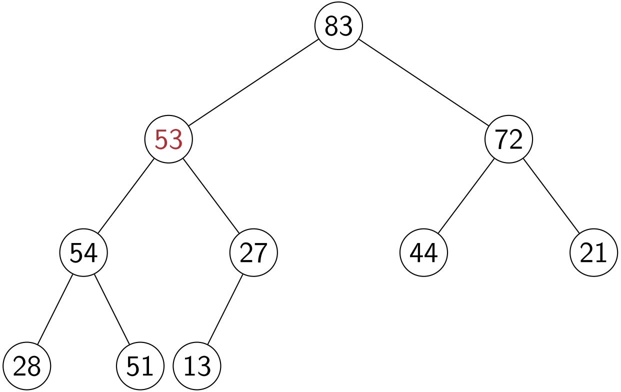


Cannot leave a level incomplete



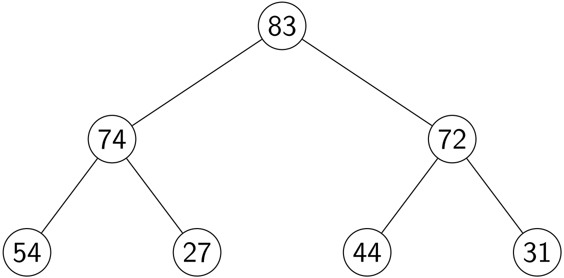
Non-examples

Heap property is violated



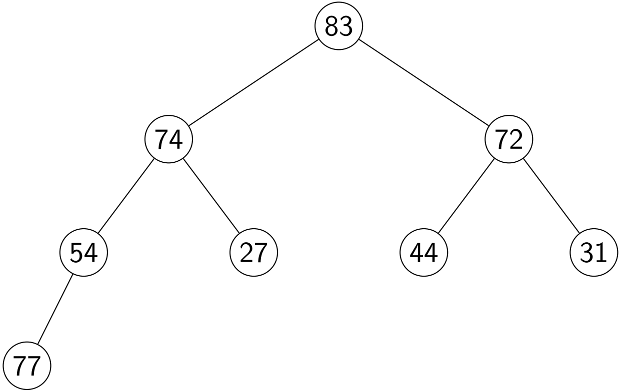
insert()

- `insert(77)`



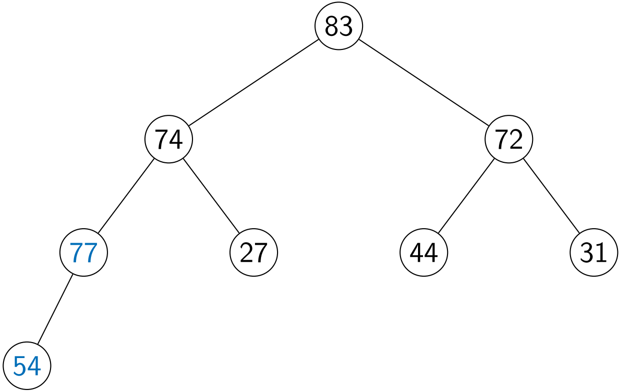
insert()

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



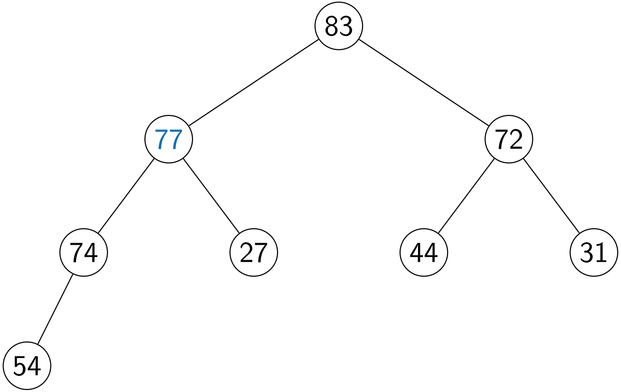
insert()

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



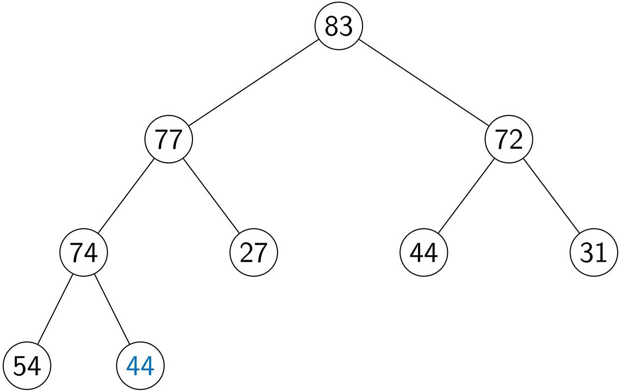
insert()

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



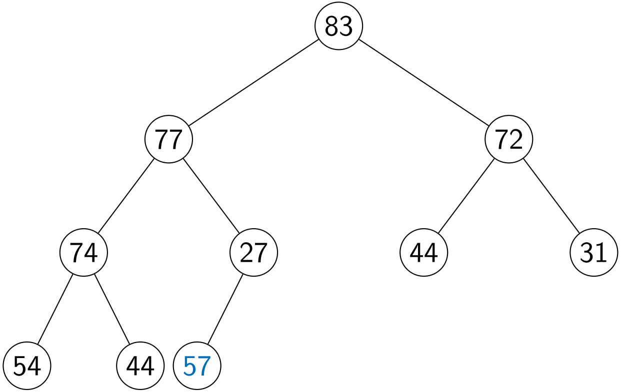
insert()

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



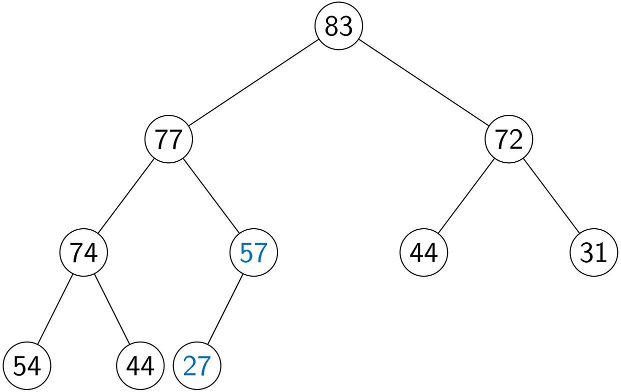
insert()

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



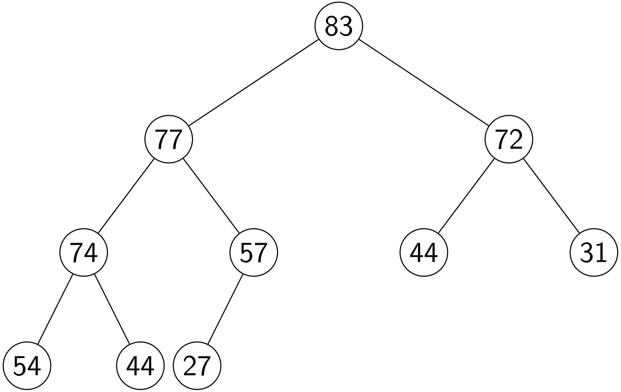
insert()

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



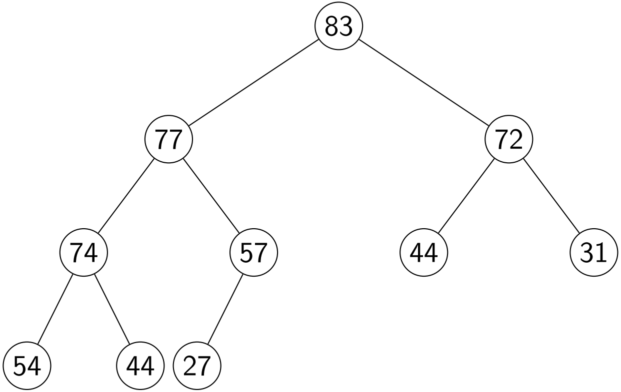
Complexity of `insert()`

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



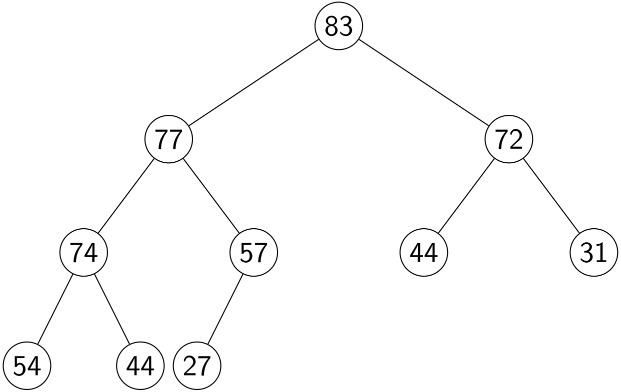
Complexity of `insert()`

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



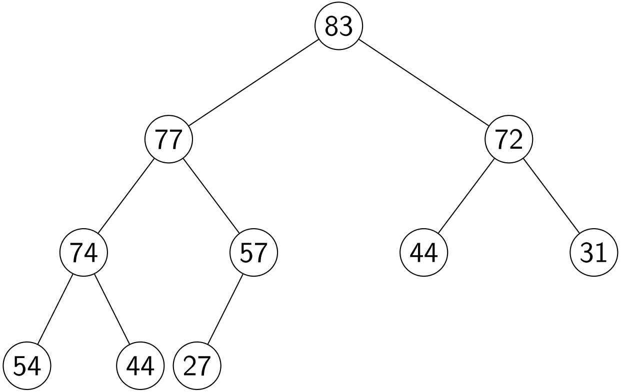
Complexity of `insert()`

- 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



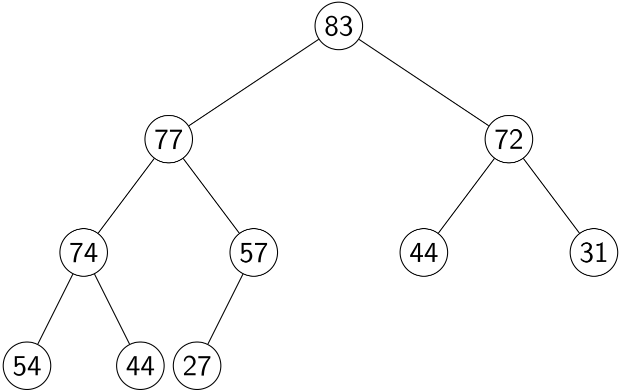
Complexity of `insert()`

- 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



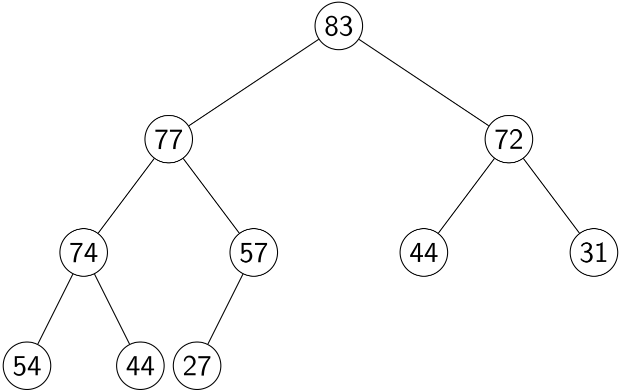
Complexity of insert()

- 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



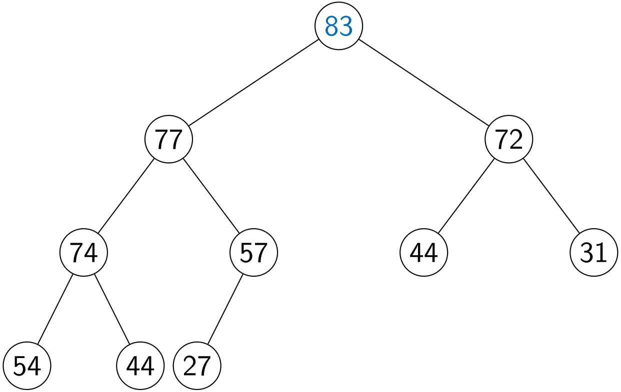
Complexity of `insert()`

- 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 $O(\log N)$



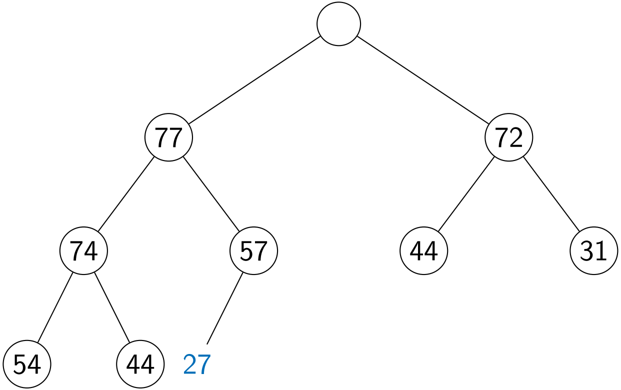
delete_max()

- Maximum value is always at the root



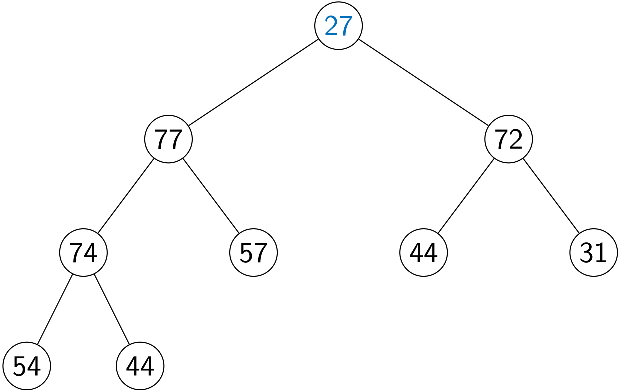
delete_max()

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



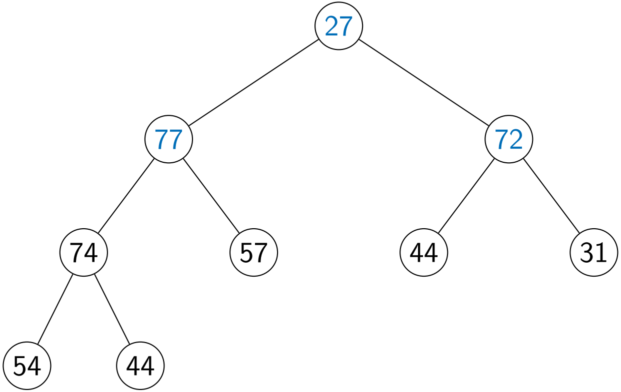
delete_max()

- 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



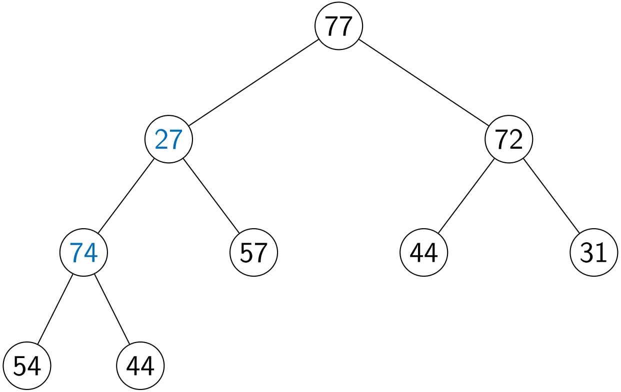
delete_max()

- 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



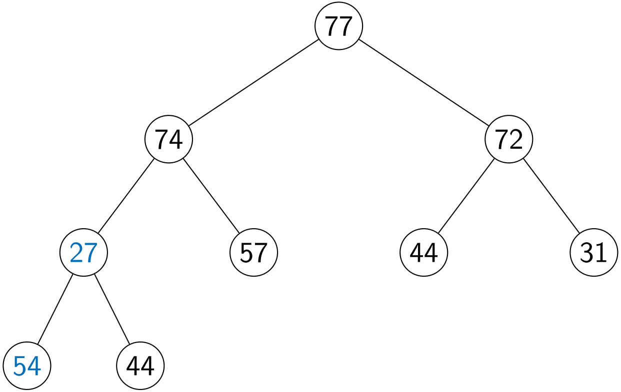
delete_max()

- 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 $O(\log N)$



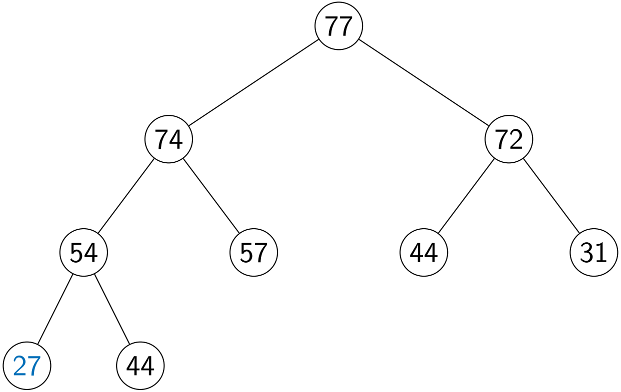
delete_max()

- 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 $O(\log N)$



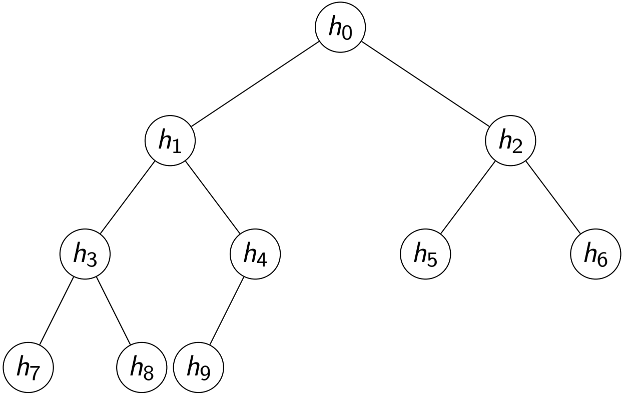
delete_max()

- 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 $O(\log N)$



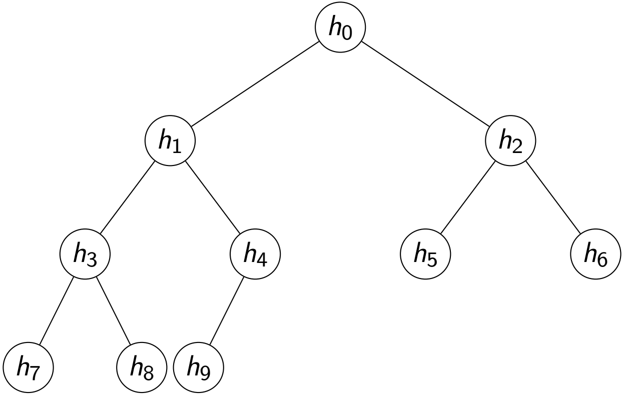
Implementation

- 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`



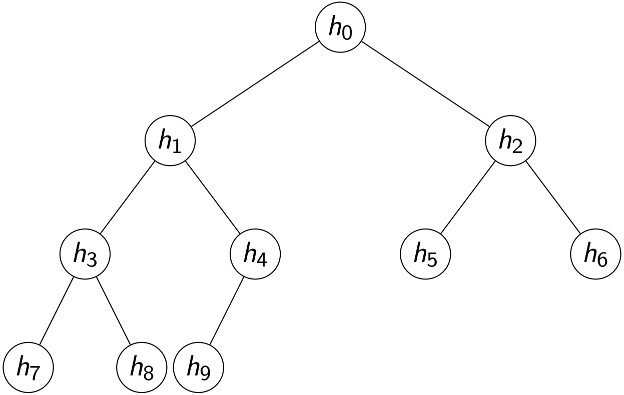
Building a heap — heapify()

- 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 $O(n)$

Heap sort

- Start with an unordered list

Heap sort

- Start with an unordered list
- Build a heap — $O(n)$

Heap sort

- Start with an unordered list
- Build a heap — $O(n)$
- Call `delete_max()` n times to extract elements in descending order — $O(n \log n)$

Heap sort

- Start with an unordered list
- Build a heap — $O(n)$
- Call `delete_max()` n times to extract elements in descending order — $O(n \log n)$
- After each `delete_max()`, heap shrinks by 1

Heap sort

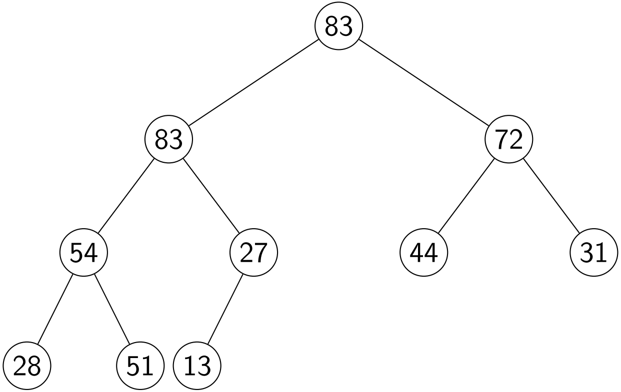
- Start with an unordered list
- Build a heap — $O(n)$
- Call `delete_max()` n times to extract elements in descending order — $O(n \log n)$
- After each `delete_max()`, heap shrinks by 1
- Store maximum value at the end of current heap

Heap sort

- Start with an unordered list
- Build a heap — $O(n)$
- Call `delete_max()` n times to extract elements in descending order — $O(n \log n)$
- After each `delete_max()`, heap shrinks by 1
- Store maximum value at the end of current heap
- In place $O(n \log n)$ sort

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()`