Union-Find data structure

Madhavan Mukund

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

Programming, Data Structures and Algorithms using Python

Week 6

Kruskal's algoriththm for minimum cost spanning tree (MCST)

- Process edges in ascending order of cost
- If edge (u, v) does not create a cycle, add it
  - (u, v) can be added if u and v are in different components
  - Adding edge (u, v) merges these components
- How can we keep track of components and merge them efficiently?

Kruskal's algoriththm for minimum cost spanning tree (MCST)

- Process edges in ascending order of cost
- If edge (u, v) does not create a cycle, add it
  - (u, v) can be added if u and v are in different components
  - Adding edge (u, v) merges these components
- How can we keep track of components and merge them efficiently?

- Components partition vertices
  - Collection of disjoint sets
- Need data structure to maintain collection of disjoint sets
  - `find(v)` — return set containing `v`
  - `union(u,v)` — merge sets of `u`, `v`

Union-Find data strucrure

- A set S partitioned into components $\{C_1, C_2, \ldots, C_k\}$
  - Each $s \in S$ belongs to exactly one C_j

Union-Find data strucrure

- A set S partitioned into components $\{C_1, C_2, \ldots, C_k\}$
  - Each $s \in S$ belongs to exactly one C_j
- Support the following operations
  - `MakeUnionFind(S)` — set up initial singleton components $\{s\}$, for each $s \in S$
  - `Find(s)` — return the component containing s
  - `Union(s,s')` — merges components containing s, s'

Naive implementation

- Assume $S = \{0, 1, \ldots, n-1\}$

Naive implementation

- Assume $S = \{0, 1, \ldots, n-1\}$
- Set up a array/dictionary `Component`

Naive implementation

- Assume $S = \{0, 1, \ldots, n-1\}$
- Set up a array/dictionary `Component`
- `MakeUnionFind(S)` —
  - Set `Component[i] = i` for each `i`

Naive implementation

- Assume $S = \{0, 1, \ldots, n-1\}$
- Set up a array/dictionary `Component`
- `MakeUnionFind(S)` —
  - Set `Component[i] = i` for each `i`
- `Find(i)`
  - Return `Component[i]`

Naive implementation

- Assume $S = \{0, 1, \ldots, n-1\}$
- Set up a array/dictionary `Component`
- `MakeUnionFind(S)` —
  - Set `Component[i] = i` for each `i`
- `Find(i)`
  - Return `Component[i]`
- `Union(i,j)`

```
c_old = Component[i]
c_new = Component[j]
for k in range(n):
    if Component[k] == c_old:
        Component[k] = c_new
```

Naive implementation

- Assume $S = \{0, 1, \ldots, n-1\}$
- Set up a array/dictionary `Component`
- `MakeUnionFind(S)` —
  - Set `Component[i] = i` for each `i`
- `Find(i)`
  - Return `Component[i]`
- `Union(i,j)`

```
c_old = Component[i]
c_new = Component[j]
for k in range(n):
    if Component[k] == c_old:
       Component[k] = c_new
```

Complexity

- `MakeUnionFind(S)` — $O(n)$
- `Find(i)` — $O(1)$
- `Union(i,j)` — $O(n)$
- Sequence of m `Union()` operations takes time $O(mn)$

Improved implementation

- Another array/dictionary `Members`

Improved implementation

- Another array/dictionary `Members`
- For each component c, `Members[c]` is a list of its members
- `Size[c]` = `length(Members[c])` is the number of members

Improved implementation

- Another array/dictionary `Members`
- For each component c, `Members[c]` is a list of its members
- `Size[c]` $=$ `length(Members[c])` is the number of members

- `MakeUnionFind(S)`
  - Set `Component[i] = i` for all `i`
  - Set `Members[i] = [i]`, `Size[i] = 1` for all `i`

Improved implementation

- Another array/dictionary `Members`
- For each component c, `Members[c]` is a list of its members
- `Size[c]` = `length(Members[c])` is the number of members

- `MakeUnionFind(S)`
  - Set `Component[i] = i` for all `i`
  - Set `Members[i] = [i]`, `Size[i] = 1` for all `i`
- `Find(i)`
  - Return `Component[i]`

Improved implementation

- Another array/dictionary `Members`
- For each component c, `Members[c]` is a list of its members
- `Size[c]` = `length(Members[c])` is the number of members

- `MakeUnionFind(S)`
  - Set `Component[i] = i` for all `i`
  - Set `Members[i] = [i]`, `Size[i] = 1` for all `i`
- `Find(i)`
  - Return `Component[i]`
- `Union(i,j)`

```
c_old = Component[i]
c_new = Component[j]
for k in Members[c_old]:
    Component[k] = c_new
    Members[c_new].append(k)
    Size[c_new] = Size[c_new] + 1
```

Why does this help?

- `MakeUnionFind(S)`
  - Set `Component[i] = i` for all `i`
  - Set `Members[i] = [i]`, `Size[i] = 1` for all `i`
- `Find(i)`
  - Return `Component[i]`
- `Union(i,j)`

```
c_old = Component[i]
c_new = Component[j]
for k in Members[c_old]:
    Component[k] = c_new
    Members[c_new].append(k)
    Size[c_new] = Size[c_new] + 1
```

Why does this help?

- `MakeUnionFind(S)`
  - Set `Component[i] = i` for all `i`
  - Set `Members[i] = [i]`, `Size[i] = 1` for all `i`
- `Find(i)`
  - Return `Component[i]`
- `Union(i,j)`

```
c_old = Component[i]
c_new = Component[j]
for k in Members[c_old]:
    Component[k] = c_new
    Members[c_new].append(k)
    Size[c_new] = Size[c_new] + 1
```

- `Members[c_old]` allows us to merge `Component[i]` into `Component[j]` in time $O(Size[c\_old])$ rather than $O(n)$

Why does this help?

- `MakeUnionFind(S)`
  - Set `Component[i] = i` for all `i`
  - Set `Members[i] = [i]`, `Size[i] = 1` for all `i`
- `Find(i)`
  - Return `Component[i]`
- `Union(i,j)`

```
c_old = Component[i]
c_new = Component[j]
for k in Members[c_old]:
    Component[k] = c_new
    Members[c_new].append(k)
    Size[c_new] = Size[c_new] + 1
```

- `Members[c_old]` allows us to merge `Component[i]` into `Component[j]` in time $O(Size[c\_old])$ rather than $O(n)$
- How can we make use of `Size[c]`
  - Always merge smaller component into larger one
  - If `Size[c] < Size[c']` relabel `c` as `c'`, else relabel `c'` as `c`

Why does this help?

- `MakeUnionFind(S)`
  - Set `Component[i] = i` for all `i`
  - Set `Members[i] = [i]`, `Size[i] = 1` for all `i`
- `Find(i)`
  - Return `Component[i]`
- `Union(i,j)`

```
c_old = Component[i]
c_new = Component[j]
for k in Members[c_old]:
    Component[k] = c_new
    Members[c_new].append(k)
    Size[c_new] = Size[c_new] + 1
```

- `Members[c_old]` allows us to merge `Component[i]` into `Component[j]` in time $O(Size[c\_old])$ rather than $O(n)$
- How can we make use of `Size[c]`
  - Always merge smaller component into larger one
  - If `Size[c] < Size[c']` relabel `c` as `c'`, else relabel `c'` as `c`
- Individual merge operations can still take time $O(n)$
  - Both `Size[c]`, `Size[c']` could be about $n/2$
  - More careful accounting

Why does this help?

- `MakeUnionFind(S)`
  - Set `Component[i] = i` for all `i`
  - Set `Members[i] = [i]`, `Size[i] = 1` for all `i`
- `Find(i)`
  - Return `Component[i]`
- `Union(i,j)`

```
c_old = Component[i]
c_new = Component[j]
for k in Members[c_old]:
    Component[k] = c_new
    Members[c_new].append(k)
    Size[c_new] = Size[c_new] + 1
```

- Always merge smaller component into larger one

Why does this help?

- `MakeUnionFind(S)`
  - Set `Component[i] = i` for all `i`
  - Set `Members[i] = [i]`, `Size[i] = 1` for all `i`
- `Find(i)`
  - Return `Component[i]`
- `Union(i,j)`

```
c_old = Component[i]
c_new = Component[j]
for k in Members[c_old]:
    Component[k] = c_new
    Members[c_new].append(k)
    Size[c_new] = Size[c_new] + 1
```

- Always merge smaller component into larger one
- For each `i`, size of `Component[i]` at least doubles each time it is relabelled

Why does this help?

- `MakeUnionFind(S)`
  - Set `Component[i] = i` for all `i`
  - Set `Members[i] = [i]`, `Size[i] = 1` for all `i`
- `Find(i)`
  - Return `Component[i]`
- `Union(i,j)`

```
c_old = Component[i]
c_new = Component[j]
for k in Members[c_old]:
   Component[k] = c_new
   Members[c_new].append(k)
   Size[c_new] = Size[c_new] + 1
```

- Always merge smaller component into larger one
- For each `i`, size of `Component[i]` at least doubles each time it is relabelled
- After m `Union()` operations, at most $2m$ elements have been "touched"
  - Size of `Component[i]` is at most $2m$

Why does this help?

- `MakeUnionFind(S)`
  - Set `Component[i] = i` for all `i`
  - Set `Members[i] = [i]`, `Size[i] = 1` for all `i`
- `Find(i)`
  - Return `Component[i]`
- `Union(i,j)`

```
c_old = Component[i]
c_new = Component[j]
for k in Members[c_old]:
    Component[k] = c_new
    Members[c_new].append(k)
    Size[c_new] = Size[c_new] + 1
```

- Always merge smaller component into larger one
- For each `i`, size of `Component[i]` at least doubles each time it is relabelled
- After m `Union()` operations, at most $2m$ elements have been "touched"
  - Size of `Component[i]` is at most $2m$
- Size of `Component[i]` grows as $1, 2, 4, \ldots$, so `i` changes component at most $\log m$ times

Why does this help?

- Always merge smaller component into larger one
- For each `i`, size of `Component[i]` at least doubles each time it is relabelled
- After m `Union()` operations, at most $2m$ elements have been "touched"
  - Size of `Component[i]` is at most $2m$
- Size of `Component[i]` grows as $1, 2, 4, \ldots$, so `i` changes component at most $\log m$ times

Why does this help?

- Always merge smaller component into larger one
- For each `i`, size of `Component[i]` at least doubles each time it is relabelled
- After m `Union()` operations, at most $2m$ elements have been "touched"
  - Size of `Component[i]` is at most $2m$
- Size of `Component[i]` grows as $1, 2, 4, \ldots$, so `i` changes component at most $\log m$ times

- Over m updates
  - At most $2m$ elements are relabelled
  - Each one at most $O(\log m)$ times

Why does this help?

- Always merge smaller component into larger one
- For each `i`, size of `Component[i]` at least doubles each time it is relabelled
- After m `Union()` operations, at most $2m$ elements have been "touched"
  - Size of `Component[i]` is at most $2m$
- Size of `Component[i]` grows as $1, 2, 4, \ldots$, so `i` changes component at most $\log m$ times

- Over m updates
  - At most $2m$ elements are relabelled
  - Each one at most $O(\log m)$ times
- Overall, m `Union()` operations take time $O(m \log m)$

Why does this help?

- Always merge smaller component into larger one
- For each `i`, size of `Component[i]` at least doubles each time it is relabelled
- After m `Union()` operations, at most $2m$ elements have been "touched"
  - Size of `Component[i]` is at most $2m$
- Size of `Component[i]` grows as $1, 2, 4, \ldots$, so `i` changes component at most $\log m$ times

- Over m updates
  - At most $2m$ elements are relabelled
  - Each one at most $O(\log m)$ times
- Overall, m `Union()` operations take time $O(m \log m)$
- Works out to time $O(\log m)$ per `Union()` operation
  - Amortised complexity of `Union()` is $O(\log m)$

Back to Kruskal's algorithm

- Sort $E = \{e_0, e_1, \ldots, e_{m-1}\}$ in ascending order

Back to Kruskal's algorithm

- Sort $E = \{e_0, e_1, \ldots, e_{m-1}\}$ in ascending order
- `MakeUnionFind(V)` — each vertex `j` is in component `j`

Back to Kruskal's algorithm

- Sort $E = \{e_0, e_1, \dots, e_{m-1}\}$ in ascending order
- `MakeUnionFind(V)` — each vertex `j` is in component `j`
- Adding and edge $e_k = (u, v)$ to the tree
  - Check that `Find(u) != Find(v)`
  - Merge components: `Union(Component[u],Component[v])`

Back to Kruskal's algorithm

- Sort $E = \{e_0, e_1, \ldots, e_{m-1}\}$ in ascending order
- `MakeUnionFind(V)` — each vertex `j` is in component `j`
- Adding and edge $e_k = (u, v)$ to the tree
  - Check that `Find(u) != Find(v)`
  - Merge components: `Union(Component[u],Component[v])`
- Tree has $n-1$ edges, so $O(n)$ `Union()` operations
  - $O(n \log n)$ amortised cost, overall

Back to Kruskal's algorithm

- Sort $E = \{e_0, e_1, \ldots, e_{m-1}\}$ in ascending order
- `MakeUnionFind(V)` — each vertex `j` is in component `j`
- Adding and edge $e_k = (u, v)$ to the tree
  - Check that `Find(u) != Find(v)`
  - Merge components: `Union(Component[u],Component[v])`
- Tree has $n - 1$ edges, so $O(n)$ `Union()` operations
  - $O(n \log n)$ amortised cost, overall
- Sorting E takes $O(m \log m)$
  - Equivalently $O(m \log n)$, since $m \leq n^2$

Back to Kruskal's algorithm

- Sort $E = \{e_0, e_1, \ldots, e_{m-1}\}$ in ascending order
- `MakeUnionFind(V)` — each vertex `j` is in component `j`
- Adding and edge $e_k = (u, v)$ to the tree
  - Check that `Find(u) != Find(v)`
  - Merge components: `Union(Component[u],Component[v])`
- Tree has $n - 1$ edges, so $O(n)$ `Union()` operations
  - $O(n \log n)$ amortised cost, overall
- Sorting E takes $O(m \log m)$
  - Equivalently $O(m \log n)$, since $m \leq n^2$
- Overall time, $O((m + n) \log n)$

Summary

- Implement Union-Find using arrays/dictionaries `Component`, `Member`, `Size`
  - `MakeUnionFind(S)` is $O(n)$
  - `Find(i)` is $O(1)$
  - Across m operations, amortised complexity of each `Union()` operation is $\log m$

Summary

- Implement Union-Find using arrays/dictionaries `Component`, `Member`, `Size`
  - `MakeUnionFind(S)` is $O(n)$
  - `Find(i)` is $O(1)$
  - Across m operations, amortised complexity of each `Union()` operation is $\log m$
- Can also maintain `Members[k]` as a tree rather than as a list
  - `Union()` becomes $O(1)$
  - With clever updates to the tree, `Find()` has amortised complexity very close to $O(1)$

Priority Queues

Madhavan Mukund

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

Programming, Data Structures and Algorithms using Python

Week 6

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		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

$N = 25$

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	5
	12	17	25	43	67	5
15	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
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		5
12	17	25	43			4
10	13	15	20			4
11	16	28	49			4
6	14					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			

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

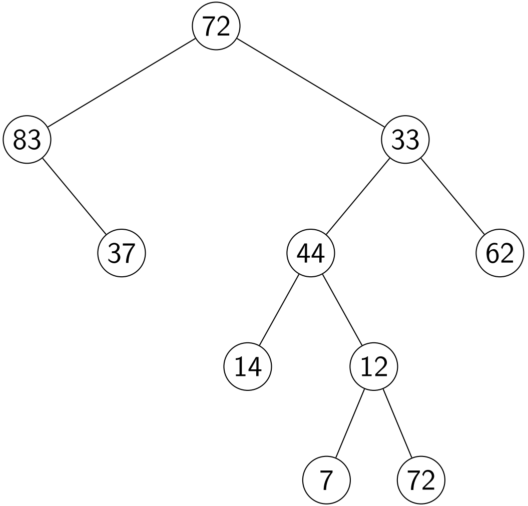
Priority queue

- 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

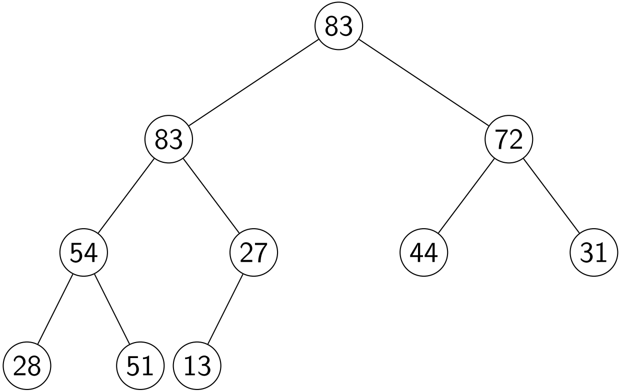
Binary trees

- 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

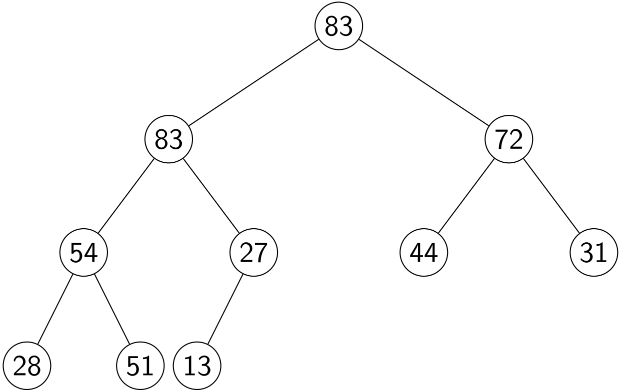


Heap

- 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

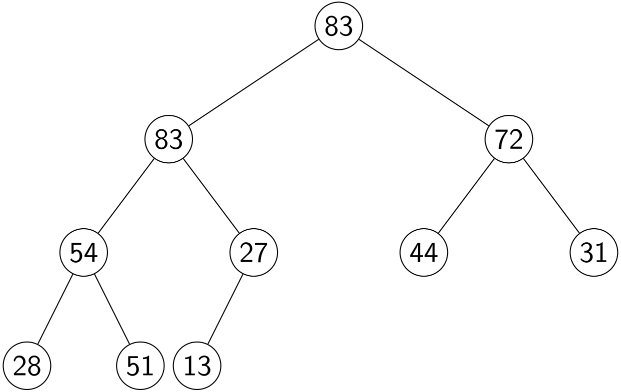


- Binary tree filled level by level, left to right
- The value at each node is at least as big the values of its children
  - max-heap
- Binary tree on the right is an example of a heap



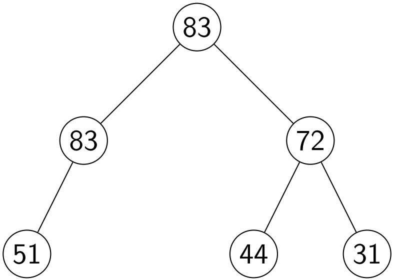
Heap

- 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



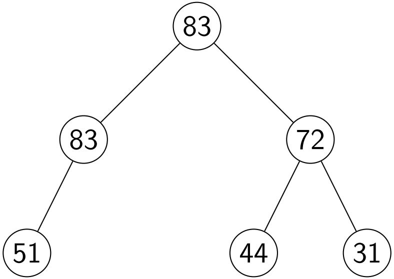
Non-examples

No "holes" allowed

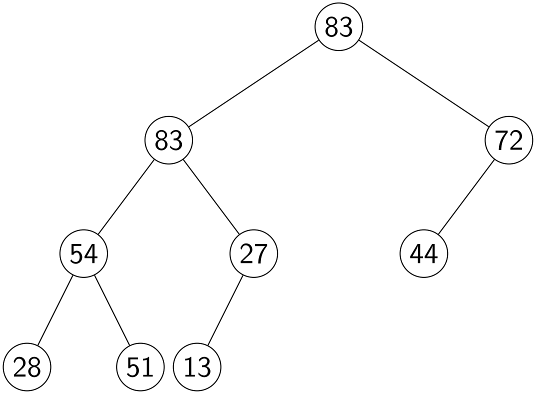


Non-examples

No "holes" allowed

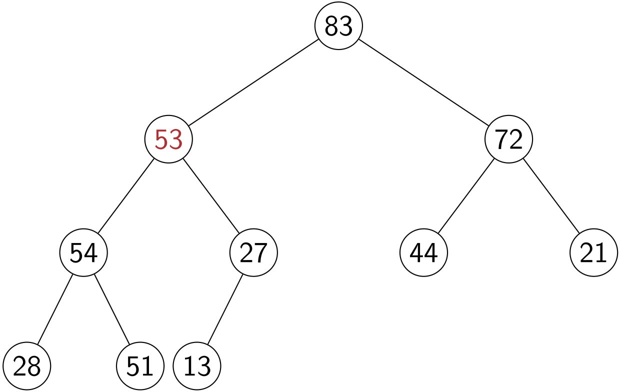


Cannot leave a level incomplete



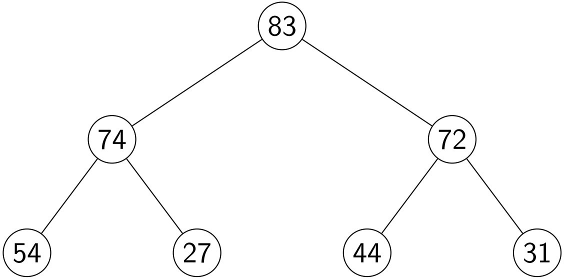
Non-examples

Heap property is violated



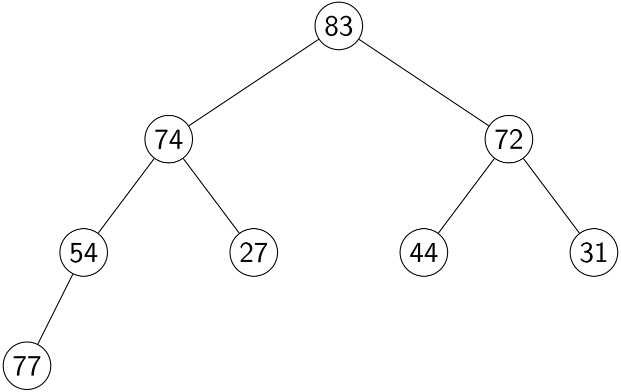
insert()

- `insert(77)`



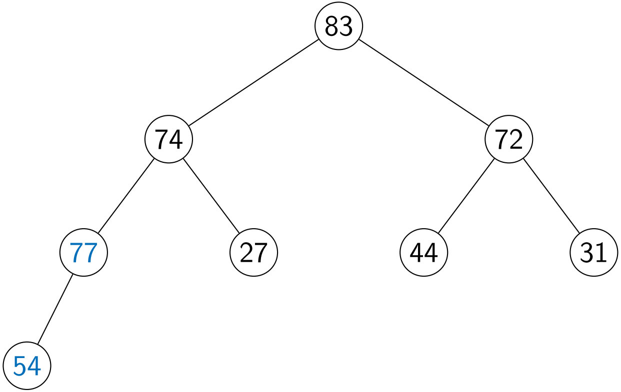
insert()

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



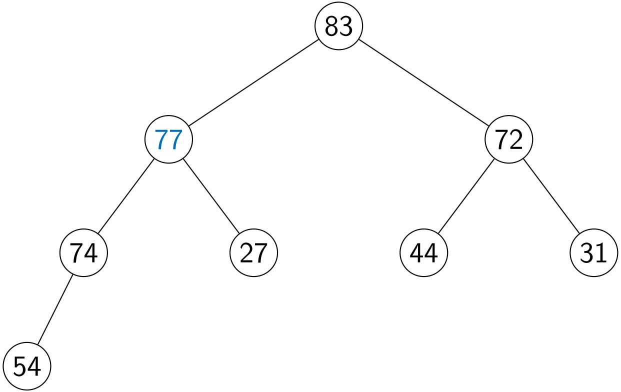
`insert()`

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



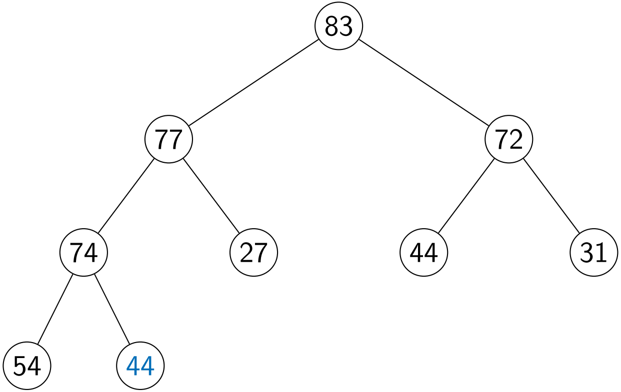
insert()

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



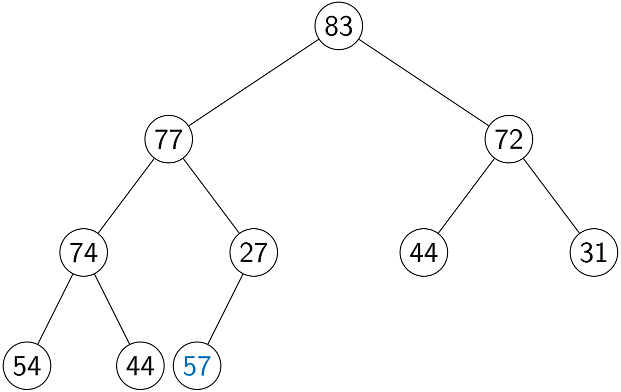
insert()

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



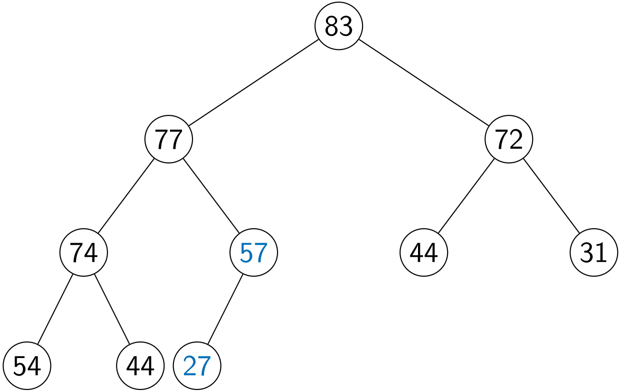
insert()

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



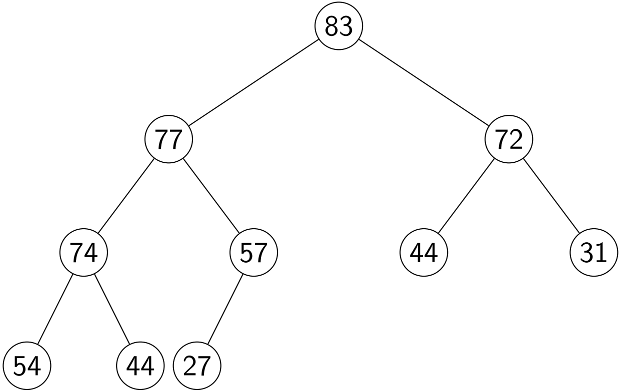
`insert()`

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



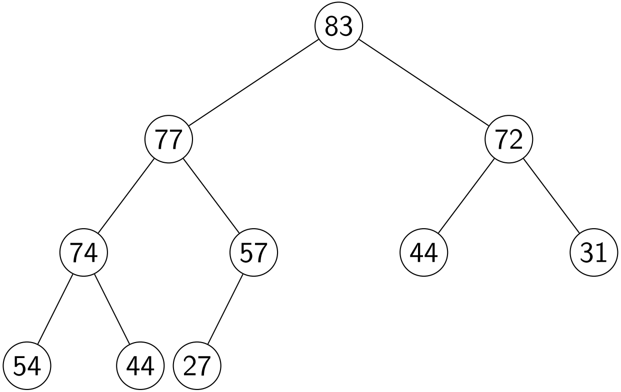
Complexity of `insert()`

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



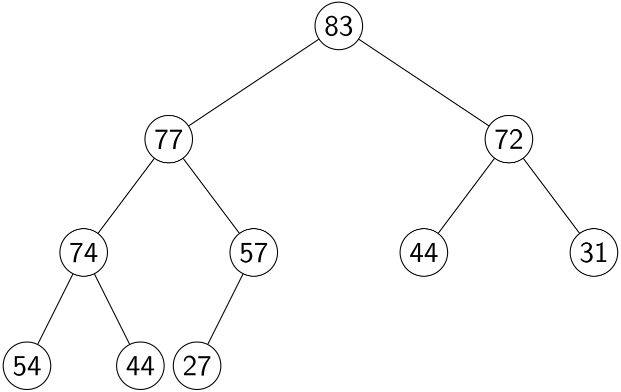
Complexity of `insert()`

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



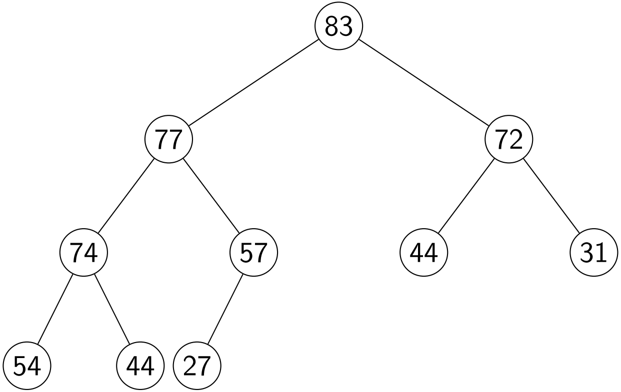
Complexity of `insert()`

- 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



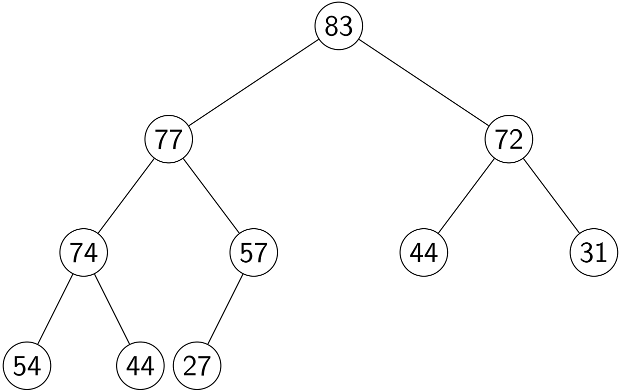
Complexity of `insert()`

- 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



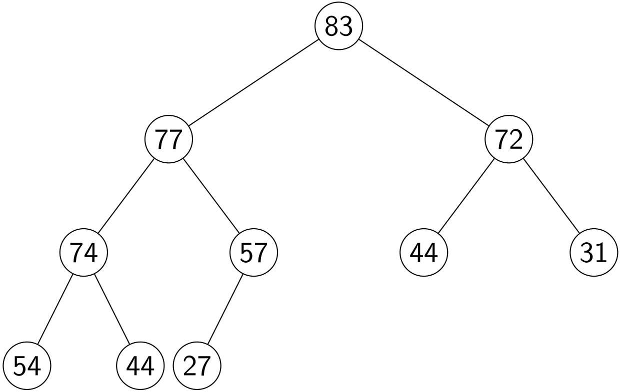
Complexity of `insert()`

- 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



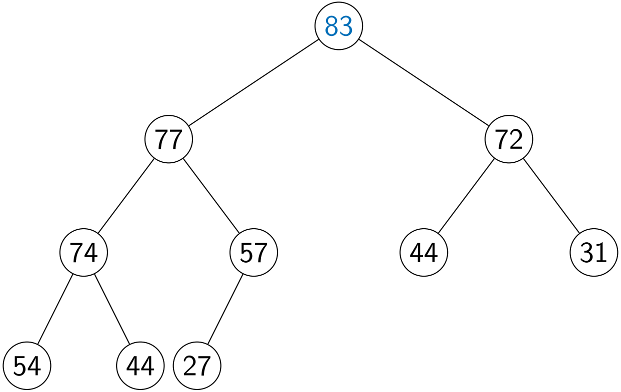
Complexity of `insert()`

- 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 $O(\log N)$



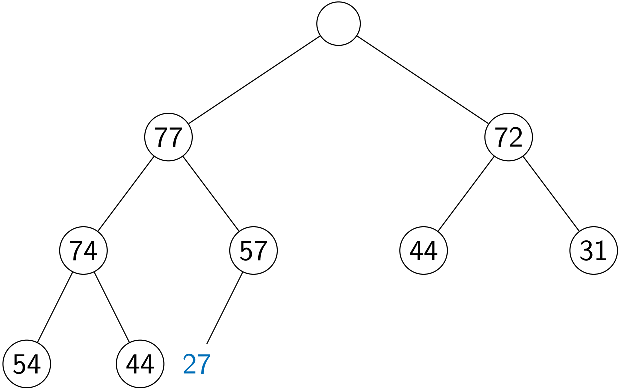
`delete_max()`

- Maximum value is always at the root



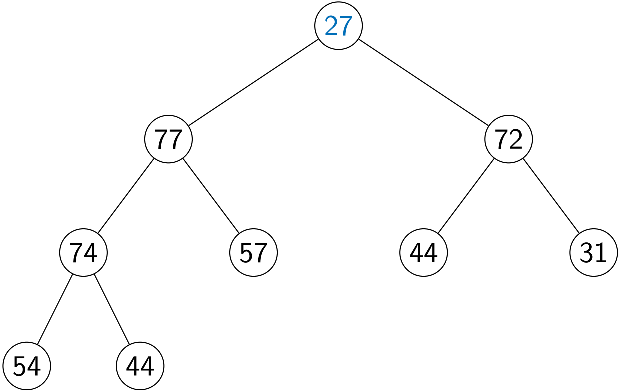
delete_max()

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



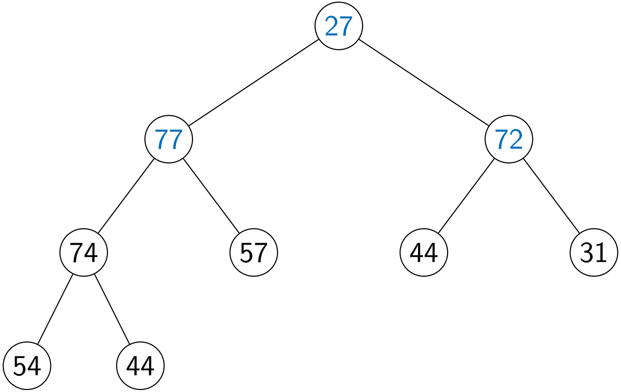
delete_max()

- 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



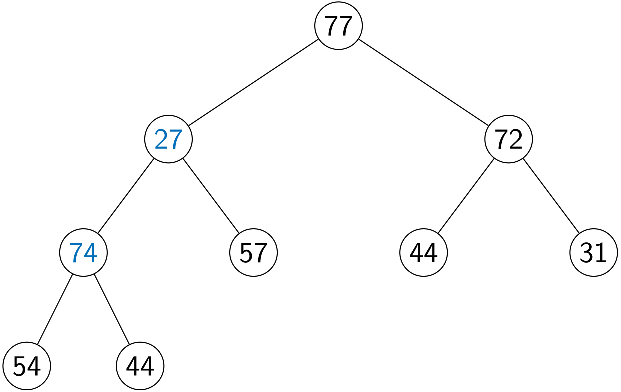
`delete_max()`

- 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



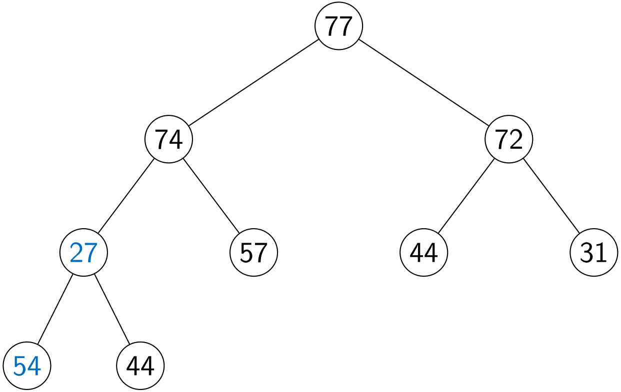
`delete_max()`

- 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 $O(\log N)$



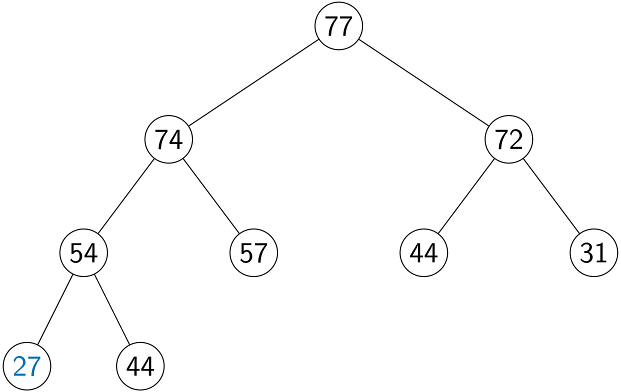
delete_max()

- 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 $O(\log N)$



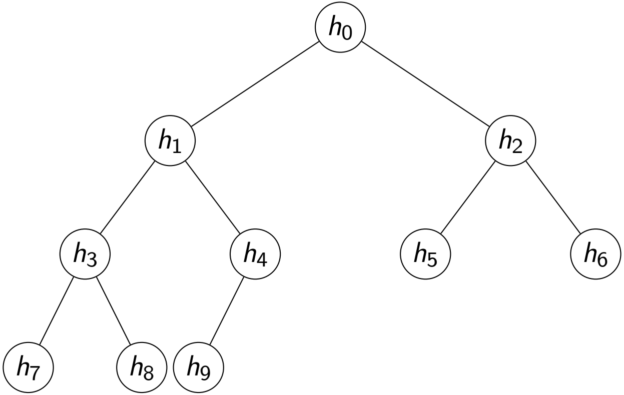
delete_max()

- 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 $O(\log N)$



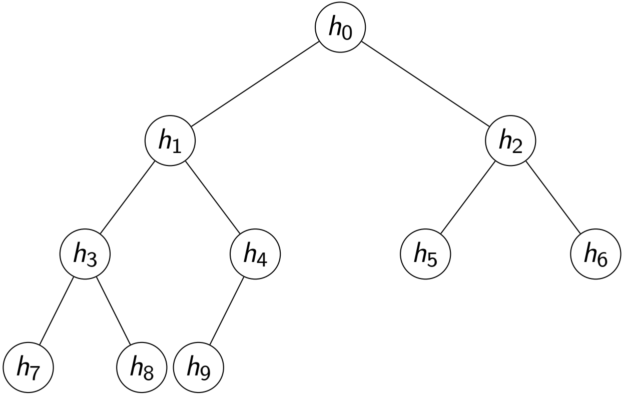
Implementation

- 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`



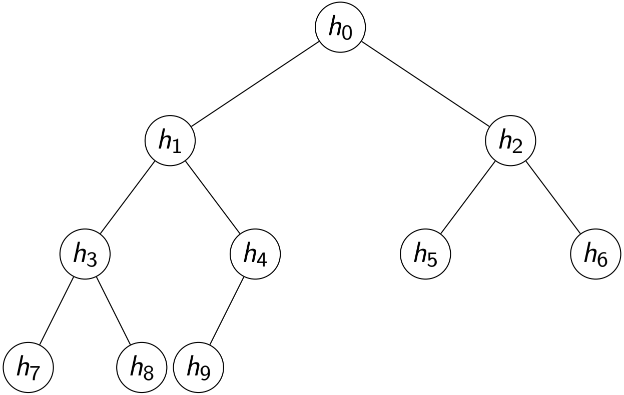
Building a heap — `heapify()`

- 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

...

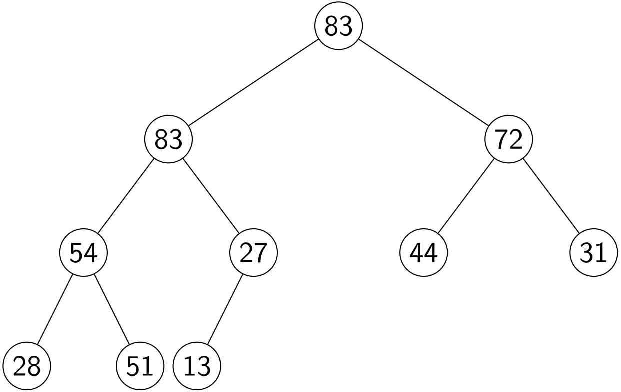
- Fix heap property at level 1
- Fix heap property at the root

- 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

...

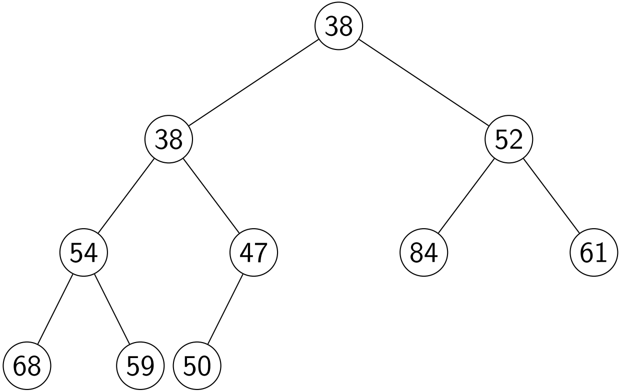
- Cost turns out to be $O(n)$

- 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()`



Using Heaps in Algorithms

Madhavan Mukund

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

Programming, Data Structures and Algorithms using Python

Week 6

Priority queues and heaps

- Priority queues support the following operations
  - `insert()`
  - `delete_max()` or `delete_min()`
- Heaps are a tree based implementation of priority queues
  - `insert()`, `delete_max()` / `delete_min()` are both $O(\log n)$
  - `heapify()` builds a heap from a list/array in time $O(n)$
- Heap can be represented as a list/array
  - Simple index arithmetic to find parent and children of a node
- What more do we need to use a heap in an algorithm?

Dijkstra's algorithm

- Maintain two dictionaries with vertices as keys
  - `visited`, initially `False` for all `v`
  - `distance`, initially `infinity` for all `v`
- Set `distance[s]` to `0`
- Repeat, until all reachable vertices are visited
  - Find unvisited vertex `nextv` with minimum distance
  - Set `visited[nextv]` to `True`
  - Recompute `distance[v]` for every neighbour `v` of `nextv`

```
def dijkstra(WMat,s):
  (rows,cols,x) = WMat.shape
  infinity = np.max(WMat)*rows+1
  (visited,distance) = ({},{})
  for v in range(rows):
    (visited[v],distance[v]) = (False,infinity)
  distance[s] = 0
  for u in range(rows):
    nextd = min([distance[v] for v in range(rows)
                    if not visited[v]])
    nextvlist = [v for v in range(rows)
                    if (not visited[v]) and
                       distance[v] == nextd]
    if nextvlist == []:
      break
    nextv = min(nextvlist)
    visited[nextv] = True
    for v in range(cols):
      if WMat[nextv,v,0] == 1 and (not visited[v]):
        distance[v] = min(distance[v],distance[nextv]
                                          +WMat[nextv,v,1])
  return(distance)
```

Dijkstra's algorithm

Bottleneck

- Find unvisited vertex j with minimum distance
  - Naive implementation requires an $O(n)$ scan

```
def dijkstra(WMat,s):
  (rows,cols,x) = WMat.shape
  infinity = np.max(WMat)*rows+1
  (visited,distance) = ({},{})
  for v in range(rows):
    (visited[v],distance[v]) = (False,infinity)
  distance[s] = 0
  for u in range(rows):
    nextd = min([distance[v] for v in range(rows)
                        if not visited[v]])
    nextvlist = [v for v in range(rows)
                        if (not visited[v]) and
                            distance[v] == nextd]
    if nextvlist == []:
      break
    nextv = min(nextvlist)
    visited[nextv] = True
    for v in range(cols):
      if WMat[nextv,v,0] == 1 and (not visited[v]):
        distance[v] = min(distance[v],distance[nextv]
                                          +WMat[nextv,v,1])
  return(distance)
```

Dijkstra's algorithm

Bottleneck

- Find unvisited vertex j with minimum distance
  - Naive implementation requires an $O(n)$ scan
- Maintain unvisited vertices as a min-heap
  - `delete_min()` in $O(\log n)$ time

```
def dijkstra(WMat,s):
  (rows,cols,x) = WMat.shape
  infinity = np.max(WMat)*rows+1
  (visited,distance) = ({},{})
  for v in range(rows):
    (visited[v],distance[v]) = (False,infinity)
  distance[s] = 0
  for u in range(rows):
    nextd = min([distance[v] for v in range(rows)
                if not visited[v]])
    nextvlist = [v for v in range(rows)
                if (not visited[v]) and
                   distance[v] == nextd]
    if nextvlist == []:
      break
    nextv = min(nextvlist)
    visited[nextv] = True
    for v in range(cols):
      if WMat[nextv,v,0] == 1 and (not visited[v]):
        distance[v] = min(distance[v],distance[nextv]
                                      +WMat[nextv,v,1])
  return(distance)
```

Dijkstra's algorithm

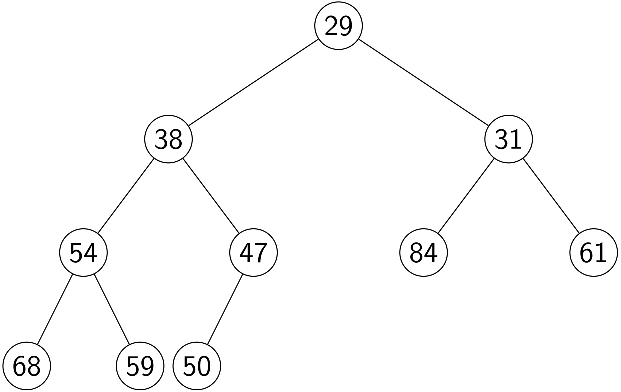
Bottleneck

- Find unvisited vertex j with minimum distance
  - Naive implementation requires an $O(n)$ scan
- Maintain unvisited vertices as a min-heap
  - `delete_min()` in $O(\log n)$ time
- But, also need to update distances of neighbours
  - Unvisited neighbours' distances are inside the min-heap
  - Updating a value is not a basic heap operation

```
def dijkstra(WMat,s):
  (rows,cols,x) = WMat.shape
  infinity = np.max(WMat)*rows+1
  (visited,distance) = ({},{})
  for v in range(rows):
    (visited[v],distance[v]) = (False,infinity)
  distance[s] = 0
  for u in range(rows):
    nextd = min([distance[v] for v in range(rows)
                if not visited[v]])
    nextvlist = [v for v in range(rows)
                if (not visited[v]) and
                    distance[v] == nextd]
    if nextvlist == []:
      break
    nextv = min(nextvlist)
    visited[nextv] = True
    for v in range(cols):
      if WMat[nextv,v,0] == 1 and (not visited[v]):
        distance[v] = min(distance[v],distance[nextv]
                                    +WMat[nextv,v,1])
  return(distance)
```

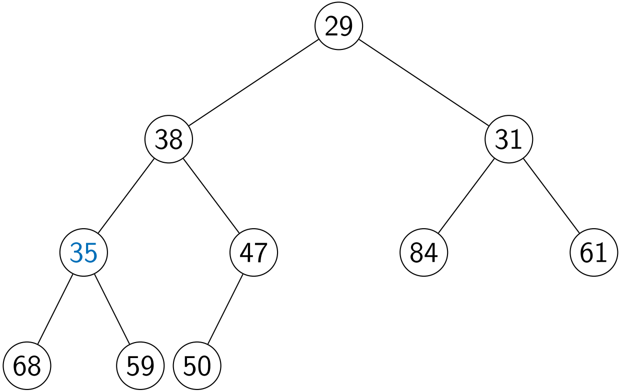
Updating values in a min-heap

- Change 54 to 35



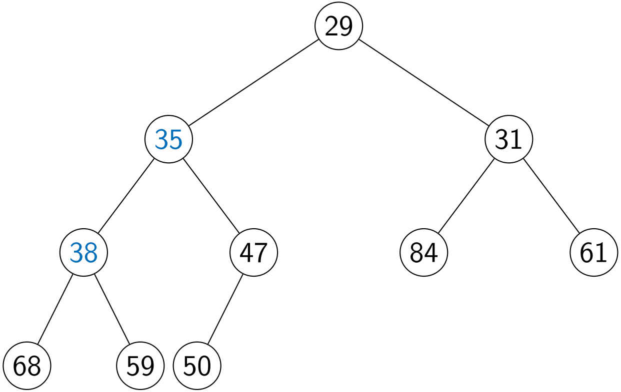
Updating values in a min-heap

- Change 54 to 35
  - Reducing a value can create a violation with parent



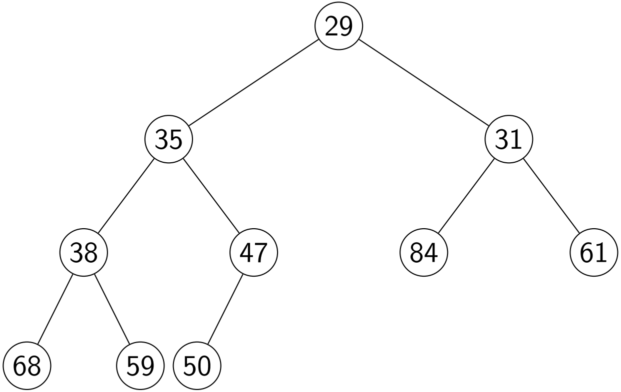
Updating values in a min-heap

- Change 54 to 35
  - Reducing a value can create a violation with parent
  - Swap upwards to restore heap, similar to `insert()`



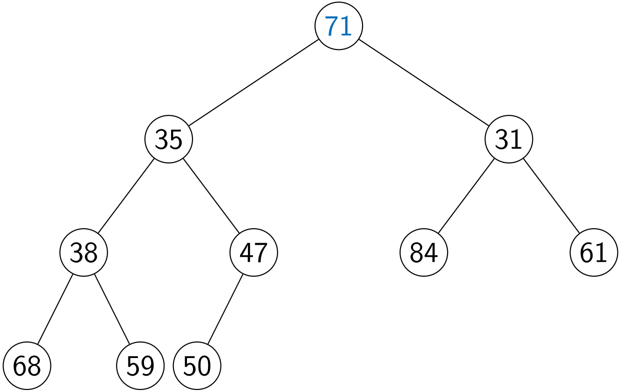
Updating values in a min-heap

- Change 54 to 35
  - Reducing a value can create a violation with parent
  - Swap upwards to restore heap, similar to `insert()`
- Change 29 to 71



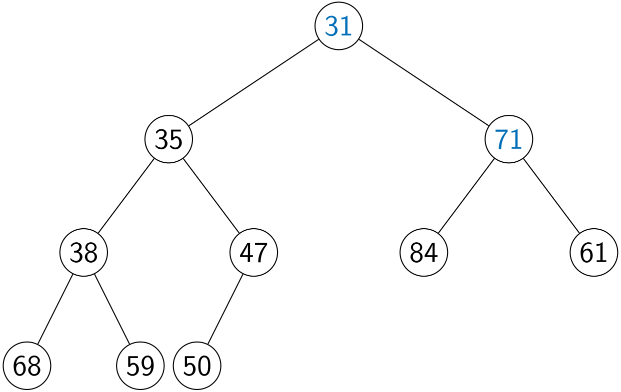
Updating values in a min-heap

- Change 54 to 35
  - Reducing a value can create a violation with parent
  - Swap upwards to restore heap, similar to `insert()`
- Change 29 to 71
  - Increasing a value can create a violation with child



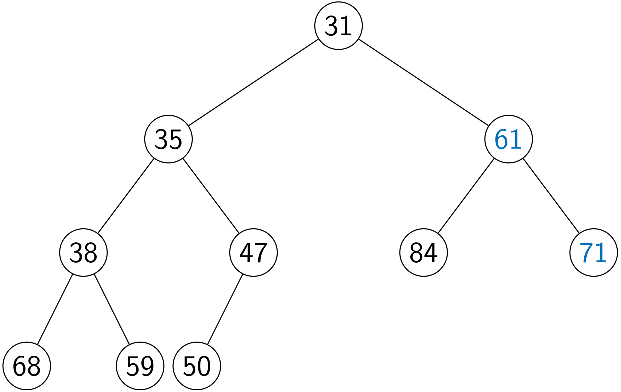
Updating values in a min-heap

- Change 54 to 35
  - Reducing a value can create a violation with parent
  - Swap upwards to restore heap, similar to `insert()`
- Change 29 to 71
  - Increasing a value can create a violation with child
  - Swap downwards to restore heap, similar to `delete_min()`



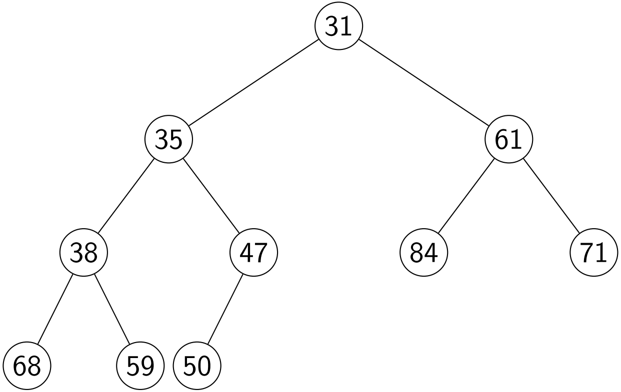
Updating values in a min-heap

- Change 54 to 35
  - Reducing a value can create a violation with parent
  - Swap upwards to restore heap, similar to `insert()`
- Change 29 to 71
  - Increasing a value can create a violation with child
  - Swap downwards to restore heap, similar to `delete_min()`



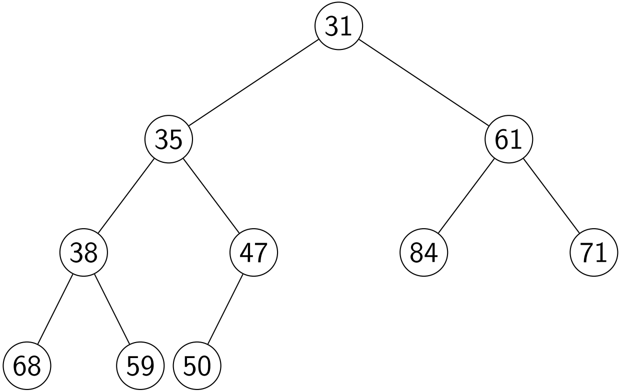
Updating values in a min-heap

- Change 54 to 35
  - Reducing a value can create a violation with parent
  - Swap upwards to restore heap, similar to `insert()`
- Change 29 to 71
  - Increasing a value can create a violation with child
  - Swap downwards to restore heap, similar to `delete_min()`
- Both updates are $O(\log n)$
  - Are we done?



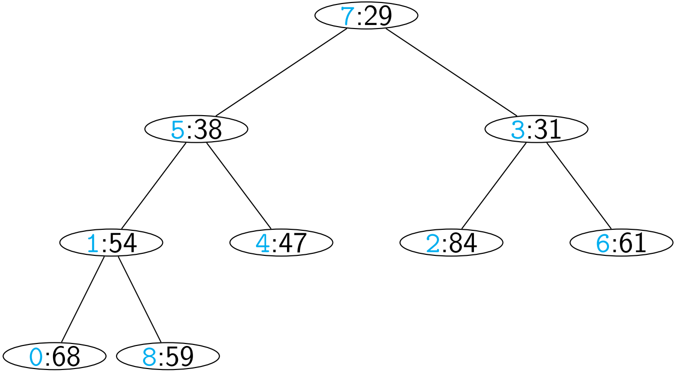
Updating values in a min-heap

- Change 54 to 35
  - Reducing a value can create a violation with parent
  - Swap upwards to restore heap, similar to `insert()`
- Change 29 to 71
  - Increasing a value can create a violation with child
  - Swap downwards to restore heap, similar to `delete_min()`
- Both updates are $O(\log n)$
  - Are we done?
- Locate the node to update?



Updating values in a min-heap

- Maintain two additional dictionaries
  - Vertices are `{0,1,...,n-1}`
  - Heap positions are $\{0, 1, \ldots, n-1\}$
  - `VtoH` maps vertices to heap positions
  - `HtoV` maps heap positions to vertices



VtoH

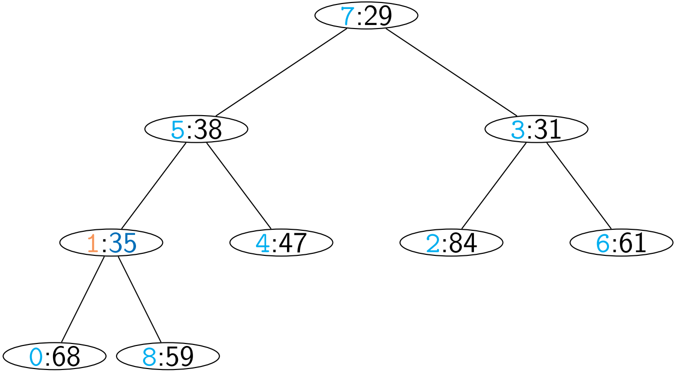
0	1	2	3	4	5	6	7	8
7	3	5	2	4	1	6	0	8

HtoV

0	1	2	3	4	5	6	7	8
7	5	3	1	4	2	6	0	8

Updating values in a min-heap

- Maintain two additional dictionaries
  - Vertices are `{0,1,...,n-1}`
  - Heap positions are $\{0, 1, \ldots, n-1\}$
  - `VtoH` maps vertices to heap positions
  - `HtoV` maps heap positions to vertices
- Update node 1 to 35



VtoH

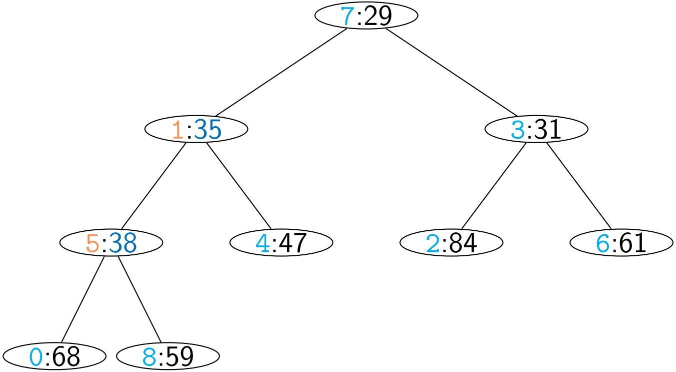
0	1	2	3	4	5	6	7	8
7	3	5	2	4	1	6	0	8

HtoV

0	1	2	3	4	5	6	7	8
7	5	3	1	4	2	6	0	8

Updating values in a min-heap

- Maintain two additional dictionaries
  - Vertices are `{0,1,...,n-1}`
  - Heap positions are $\{0, 1, \ldots, n-1\}$
  - `VtoH` maps vertices to heap positions
  - `HtoV` maps heap positions to vertices
- Update node 1 to 35
- Update `VtoH` and `HtoV` each time we swap values in the heap



VtoH

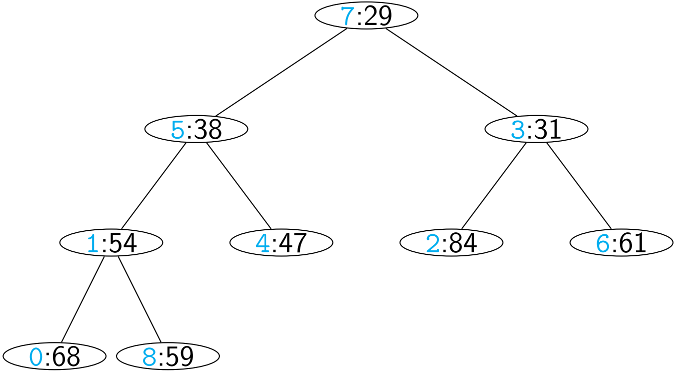
0	1	2	3	4	5	6	7	8
7	1	5	2	4	3	6	0	8

HtoV

0	1	2	3	4	5	6	7	8
7	1	3	5	4	2	6	0	8

Dijkstra's algorithm

- Using min-heaps
  - Identifying next vertex to visit is $O(\log n)$
  - Updating distance takes $O(\log n)$ per neighbour
  - Adjacency list — proportionally to degree



VtoH

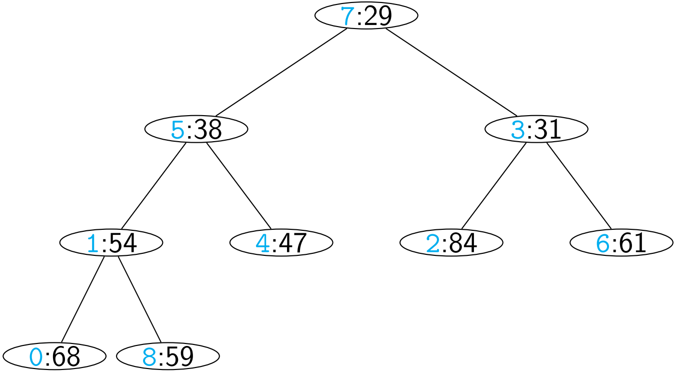
0	1	2	3	4	5	6	7	8
7	3	5	2	4	1	6	0	8

HtoV

0	1	2	3	4	5	6	7	8
7	5	3	1	4	2	6	0	8

Dijkstra's algorithm

- Using min-heaps
  - Identifying next vertex to visit is $O(\log n)$
  - Updating distance takes $O(\log n)$ per neighbour
  - Adjacency list — proportionally to degree
- Cumulatively
  - $O(n \log n)$ to identify vertices to visit across n iterations
  - $O(m \log n)$ distance updates overall



VtoH

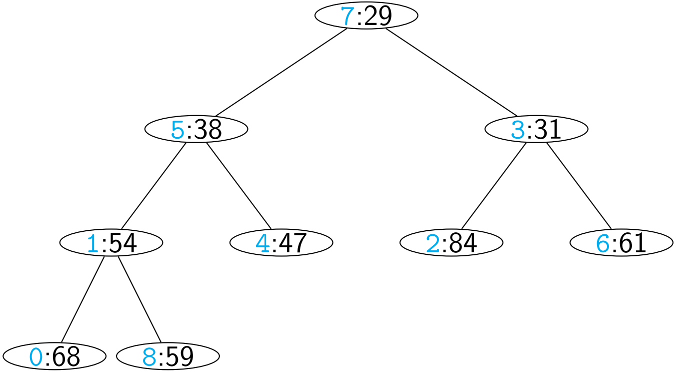
0	1	2	3	4	5	6	7	8
7	3	5	2	4	1	6	0	8

HtoV

0	1	2	3	4	5	6	7	8
7	5	3	1	4	2	6	0	8

Dijkstra's algorithm

- Using min-heaps
  - Identifying next vertex to visit is $O(\log n)$
  - Updating distance takes $O(\log n)$ per neighbour
  - Adjacency list — proportionally to degree
- Cumulatively
  - $O(n \log n)$ to identify vertices to visit across n iterations
  - $O(m \log n)$ distance updates overall
- Overall $O((m+n)\log n)$



VtoH

0	1	2	3	4	5	6	7	8
7	3	5	2	4	1	6	0	8

HtoV

0	1	2	3	4	5	6	7	8
7	5	3	1	4	2	6	0	8

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

- Updating a value in a heap takes $O(\log n)$

Summary

- Updating a value in a heap takes $O(\log n)$
- Need to maintain additional pointers to map values to heap positions and vice versa

Summary

- Updating a value in a heap takes $O(\log n)$
- Need to maintain additional pointers to map values to heap positions and vice versa
- With this extended notion of heap, Dijkstra's algorithm complexity improves from $O(n^2)$ to $O((m+n)\log n)$

Summary

- Updating a value in a heap takes $O(\log n)$
- Need to maintain additional pointers to map values to heap positions and vice versa
- With this extended notion of heap, Dijkstra's algorithm complexity improves from $O(n^2)$ to $O((m+n)\log n)$
- In a similar way, improve Prim's algorithm to $O((m+n)\log n)$

Summary

- Updating a value in a heap takes $O(\log n)$
- Need to maintain additional pointers to map values to heap positions and vice versa
- With this extended notion of heap, Dijkstra's algorithm complexity improves from $O(n^2)$ to $O((m+n)\log n)$
- In a similar way, improve Prim's algorithm to $O((m+n)\log n)$
- Heaps can also be used to sort a list in place in $O(n \log n)$

Search Trees

Madhavan Mukund

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

Programming, Data Structures and Algorithms using Python

Week 6

Dynamic sorted data

- Sorting is useful for efficient searching

Dynamic sorted data

- Sorting is useful for efficient searching
- What if the data is changing dynamically?
  - Items are periodically inserted and deleted

Dynamic sorted data

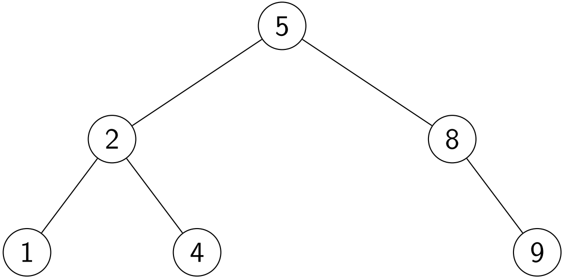
- Sorting is useful for efficient searching
- What if the data is changing dynamically?
  - Items are periodically inserted and deleted
- Insert/delete in a sorted list takes time $O(n)$

Dynamic sorted data

- Sorting is useful for efficient searching
- What if the data is changing dynamically?
  - Items are periodically inserted and deleted
- Insert/delete in a sorted list takes time $O(n)$
- Move to a tree structure, like heaps for priority queues

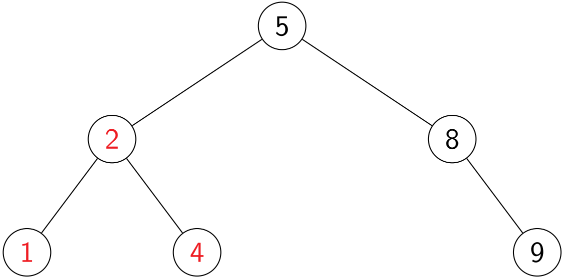
Binary search tree

- For each node with value v



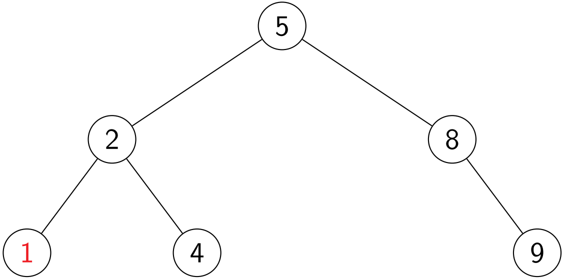
Binary search tree

- For each node with value v
  - All values in the left subtree are $< v$



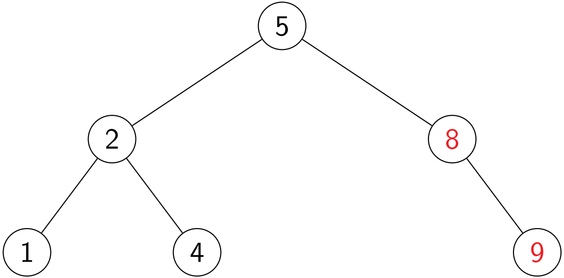
Binary search tree

- For each node with value v
  - All values in the left subtree are $< v$



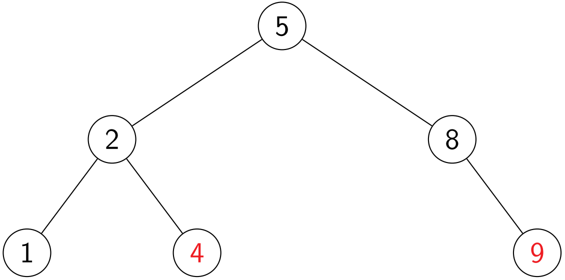
Binary search tree

- For each node with value v
  - All values in the left subtree are $< v$
  - All values in the left subtree are $> v$



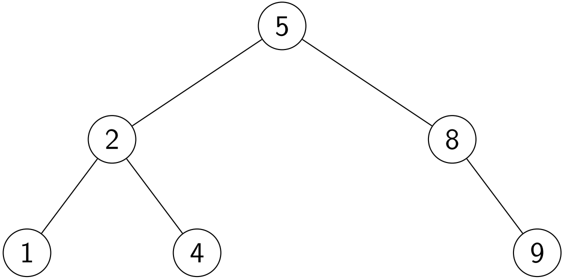
Binary search tree

- For each node with value v
  - All values in the left subtree are $< v$
  - All values in the left subtree are $> v$



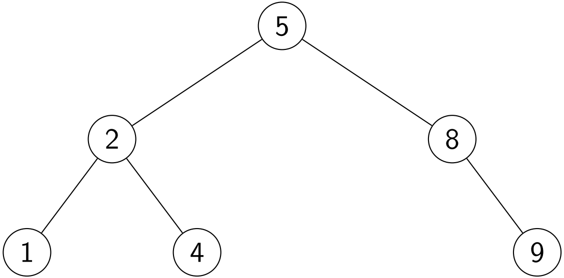
Binary search tree

- For each node with value v
  - All values in the left subtree are $< v$
  - All values in the left subtree are $> v$
- No duplicate values



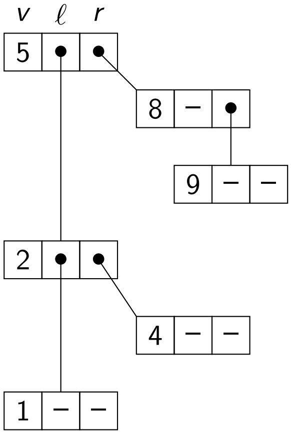
Implementing a binary search tree

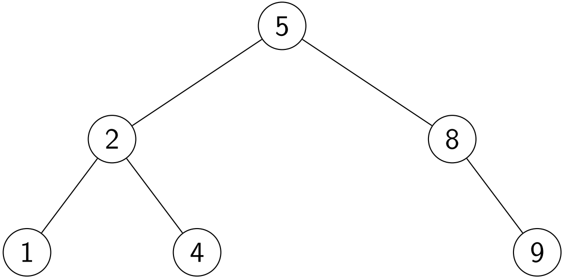
- Each node has a value and pointers to its children



Implementing a binary search tree

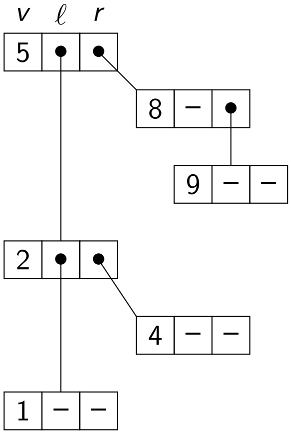
- Each node has a value and pointers to its children

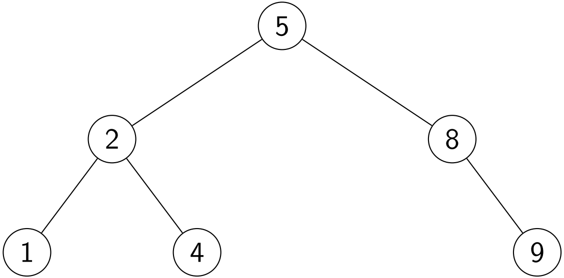




Implementing a binary search tree

- Each node has a value and pointers to its children

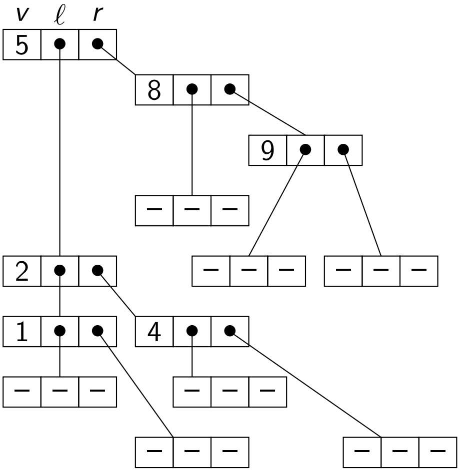


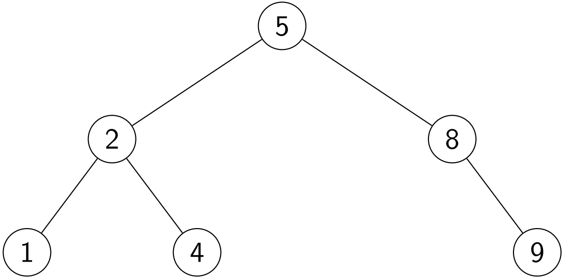


- Add a frontier with empty nodes, all fields –
  - Empty tree is single empty node
  - Leaf node points to empty nodes

Implementing a binary search tree

- Each node has a value and pointers to its children

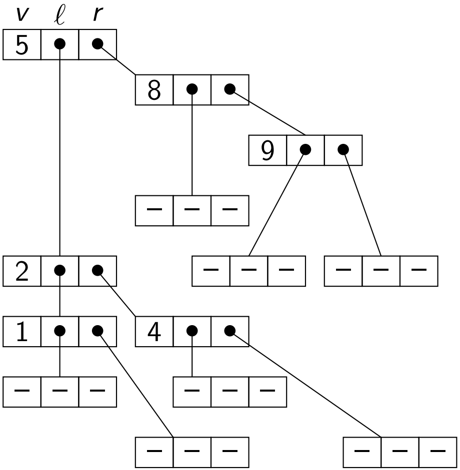


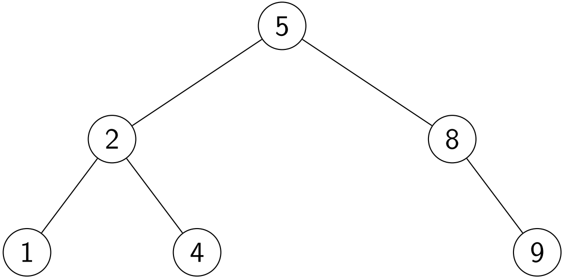


- Add a frontier with empty nodes, all fields –
  - Empty tree is single empty node
  - Leaf node points to empty nodes

Implementing a binary search tree

- Each node has a value and pointers to its children





- Add a frontier with empty nodes, all fields –
  - Empty tree is single empty node
  - Leaf node points to empty nodes
- Easier to implement operations recursively

The class Tree

- Three local fields, `value`, `left`, `right`
- Value `None` for empty value –
- Empty true has all fields `None`
- Leaf has a nonempty `value` and empty `left` and `right`

```
class Tree:

    # Constructor:
    def __init__(self,initval=None):
        self.value = initval
        if self.value:
            self.left = Tree()
            self.right = Tree()
        else:
            self.left = None
            self.right = None
        return

    # Only empty node has value None
    def isempty(self):
        return (self.value == None)

    # Leaf nodes have both children empty
    def isleaf(self):
        return (self.value != None and
                self.left.isempty() and
                self.right.isempty())
```

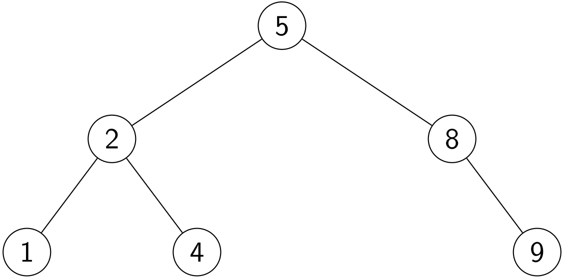
Inorder traversal

- List the left subtree, then the current node, then the right subtree
- Lists values in sorted order
- Use to print the tree

```
class Tree:
    ...
    # Inorder traversal
    def inorder(self):
        if self.isempty():
            return([])
        else:
            return(self.left.inorder()+
                   [self.value]+
                   self.right.inorder())

    # Display Tree as a string
    def __str__(self):
        return(str(self.inorder()))
```

Inorder traversal

- List the left subtree, then the current node, then the right subtree
- Lists values in sorted order
- Use to print the tree




```
class Tree:
    ...
    # Inorder traversal
    def inorder(self):
        if self.isempty():
            return([])
        else:
            return(self.left.inorder()+
                       [self.value]+
                       self.right.inorder())

    # Display Tree as a string
    def __str__(self):
        return(str(self.inorder()))
```

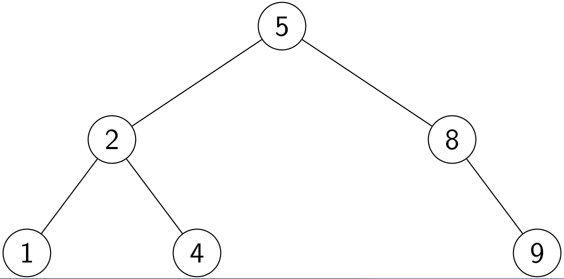
Find a value v

- Check value at current node
- If v smaller than current node, go left
- If v smaller than current node, go right
- Natural generalization of binary search

```
class Tree:
    ...
    # Check if value v occurs in tree
    def find(self,v):
        if self.isempty():
            return(False)

        if self.value == v:
            return(True)

        if v < self.value:
            return(self.left.find(v))

        if v > self.value:
            return(self.right.find(v))
```

Find a value v

- Check value at current node
- If `v` smaller than current node, go left
- If `v` smaller than current node, go right
- Natural generalization of binary search



```
class Tree:
    ...
    # Check if value v occurs in tree
    def find(self,v):
        if self.isempty():
            return(False)

        if self.value == v:
            return(True)

        if v < self.value:
            return(self.left.find(v))

        if v > self.value:
            return(self.right.find(v))
```

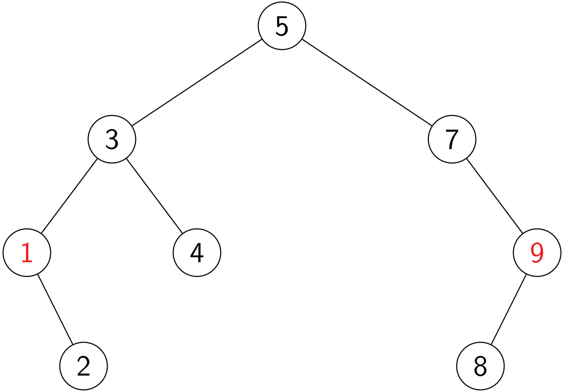
Minimum and maximum

- Minimum is left most node in the tree
- Maximum is right most node in the tree

```
class Tree:
    ...
    def minval(self):
        if self.left.isempty():
            return(self.value)
        else:
            return(self.left.minval())


    def maxval(self):
        if self.right.isempty():
            return(self.value)
        else:
            return(self.right.maxval())
```

Minimum and maximum

- Minimum is left most node in the tree
- Maximum is right most node in the tree




```
class Tree:
    ...
    def minval(self):
        if self.left.isempty():
            return(self.value)
        else:
            return(self.left.minval())


    def maxval(self):
        if self.right.isempty():
            return(self.value)
        else:
            return(self.right.maxval())
```

Insert a value v

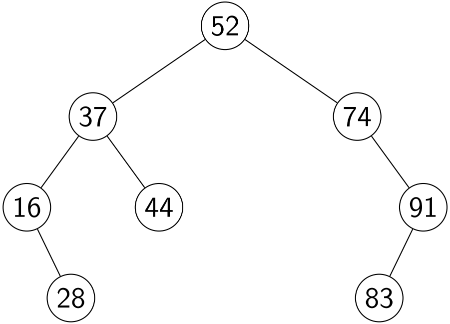
- Try to find v
- Insert at the position where find fails

```
class Tree:
    ...
    def insert(self,v):
        if self.isempty():
            self.value = v
            self.left = Tree()
            self.right = Tree()

        if self.value == v:
            return

        if v < self.value:
            self.left.insert(v)
            return

        if v > self.value:
            self.right.insert(v)
            return
```

Insert a value v

- Try to find `v`
- Insert at the position where `find` fails

Insert 21

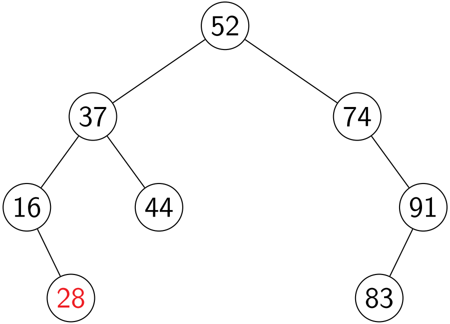



```
class Tree:
    ...
    def insert(self,v):
        if self.isempty():
            self.value = v
            self.left = Tree()
            self.right = Tree()

        if self.value == v:
            return

        if v < self.value:
            self.left.insert(v)
            return

        if v > self.value:
            self.right.insert(v)
            return
```

Insert a value v

- Try to find `v`
- Insert at the position where `find` fails

Insert 21

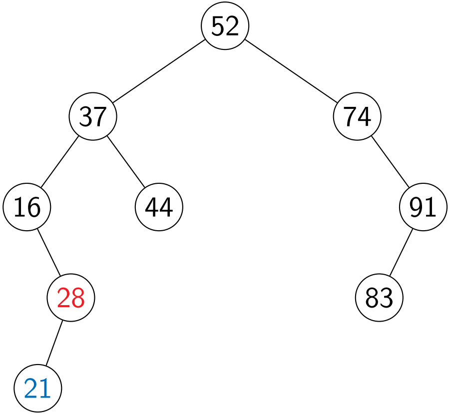



```
class Tree:
    ...
    def insert(self,v):
        if self.isempty():
            self.value = v
            self.left = Tree()
            self.right = Tree()

        if self.value == v:
            return

        if v < self.value:
            self.left.insert(v)
            return

        if v > self.value:
            self.right.insert(v)
            return
```

Insert a value v

- Try to find `v`
- Insert at the position where `find` fails

Insert 21

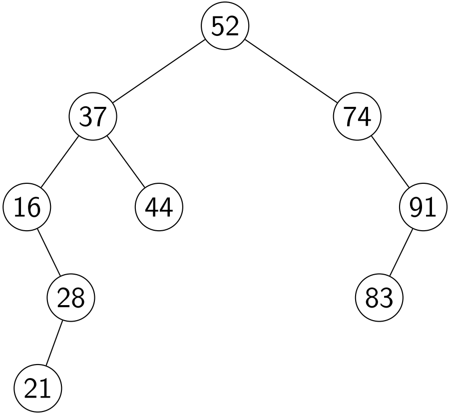



```
class Tree:
    ...
    def insert(self,v):
        if self.isempty():
            self.value = v
            self.left = Tree()
            self.right = Tree()

        if self.value == v:
            return

        if v < self.value:
            self.left.insert(v)
            return

        if v > self.value:
            self.right.insert(v)
            return
```

Insert a value v

- Try to find `v`
- Insert at the position where `find` fails

Insert 65

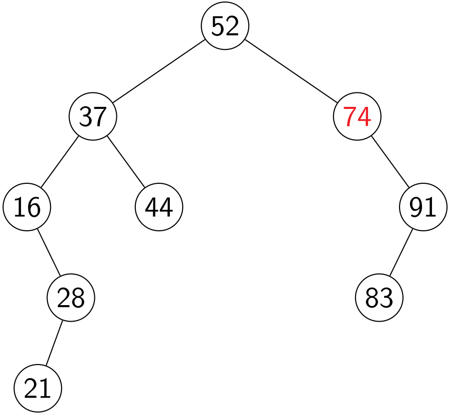



```
class Tree:
    ...
    def insert(self,v):
        if self.isempty():
            self.value = v
            self.left = Tree()
            self.right = Tree()

        if self.value == v:
            return

        if v < self.value:
            self.left.insert(v)
            return

        if v > self.value:
            self.right.insert(v)
            return
```

Insert a value v

- Try to find `v`
- Insert at the position where `find` fails

Insert 65

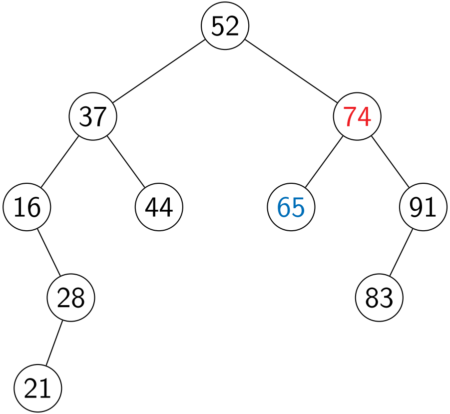



```
class Tree:
    ...
    def insert(self,v):
        if self.isempty():
            self.value = v
            self.left = Tree()
            self.right = Tree()

        if self.value == v:
            return

        if v < self.value:
            self.left.insert(v)
            return

        if v > self.value:
            self.right.insert(v)
            return
```

Insert a value v

- Try to find `v`
- Insert at the position where `find` fails

Insert 65

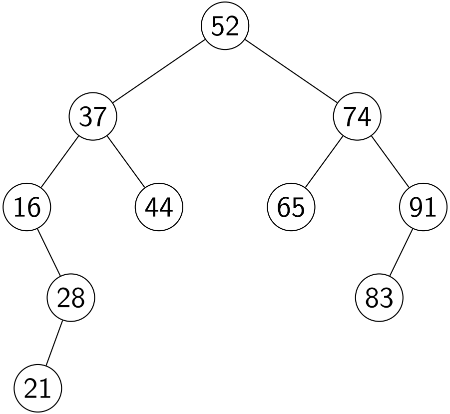



```
class Tree:
    ...
    def insert(self,v):
        if self.isempty():
            self.value = v
            self.left = Tree()
            self.right = Tree()

        if self.value == v:
            return

        if v < self.value:
            self.left.insert(v)
            return

        if v > self.value:
            self.right.insert(v)
            return
```

Insert a value v

- Try to find `v`
- Insert at the position where `find` fails

Insert 91

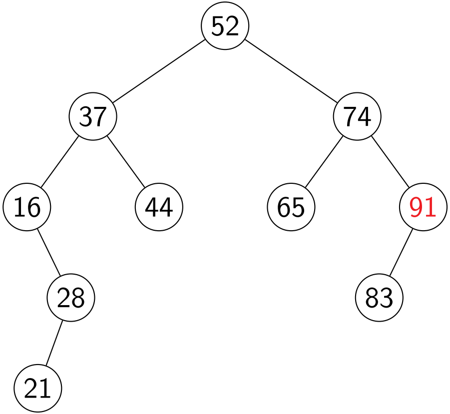


```
class Tree:
    ...
    def insert(self,v):
        if self.isempty():
            self.value = v
            self.left = Tree()
            self.right = Tree()

        if self.value == v:
            return

        if v < self.value:
            self.left.insert(v)
            return

        if v > self.value:
            self.right.insert(v)
            return
```

Insert a value v

- Try to find `v`
- Insert at the position where `find` fails

Insert 91




```
class Tree:
    ...
    def insert(self,v):
        if self.isempty():
            self.value = v
            self.left = Tree()
            self.right = Tree()

        if self.value == v:
            return

        if v < self.value:
            self.left.insert(v)
            return

        if v > self.value:
            self.right.insert(v)
            return
```

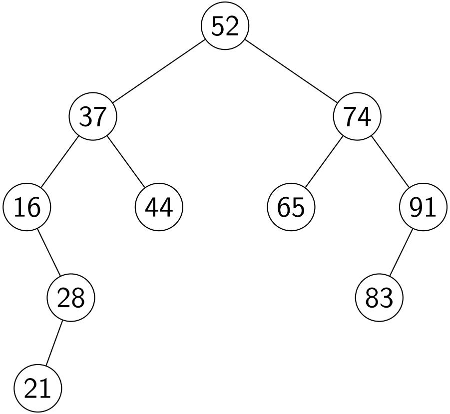
Delete a value v

- If `v` is present, delete
- Leaf node? No problem
- If only one child, promote that subtree
- Otherwise, replace `v` with `self.left.maxval()` and delete `self.left.maxval()`
  - `self.left.maxval()` has no right child

```
class Tree:
    ...
    def delete(self,v):
        if self.isempty():
            return
        if v < self.value:
            self.left.delete(v)
            return
        if v > self.value:
            self.right.delete(v)
            return
        if v == self.value:
            if self.isleaf():
                self.makeempty()
            elif self.left.isempty():
                self.copyright()
            elif self.right.isempty():
                self.copyleft()
            else:
                self.value = self.left.maxval()
                self.left.delete(self.left.maxval())
            return
```

Delete a value v

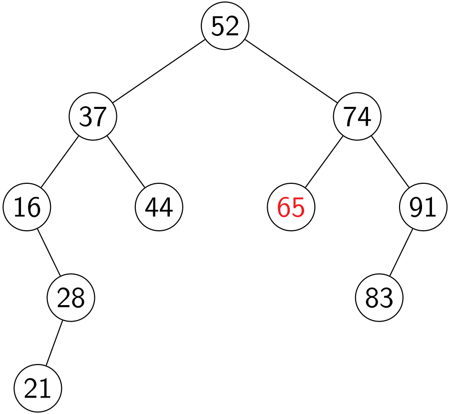
Delete 65




```
class Tree:
    ...
    def delete(self,v):
        if self.isempty():
            return
        if v < self.value:
            self.left.delete(v)
            return
        if v > self.value:
            self.right.delete(v)
            return
        if v == self.value:
            if self.isleaf():
                self.makeempty()
            elif self.left.isempty():
                self.copyright()
            elif self.right.isempty():
                self.copyleft()
            else:
                self.value = self.left.maxval()
                self.left.delete(self.left.maxval())
            return
```

Delete a value v

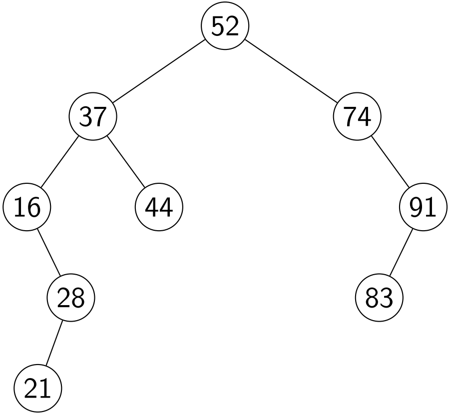
Delete 65




```
class Tree:
    ...
    def delete(self,v):
        if self.isempty():
            return
        if v < self.value:
            self.left.delete(v)
            return
        if v > self.value:
            self.right.delete(v)
            return
        if v == self.value:
            if self.isleaf():
                self.makeempty()
            elif self.left.isempty():
                self.copyright()
            elif self.right.isempty():
                self.copyleft()
            else:
                self.value = self.left.maxval()
                self.left.delete(self.left.maxval())
            return
```

Delete a value v

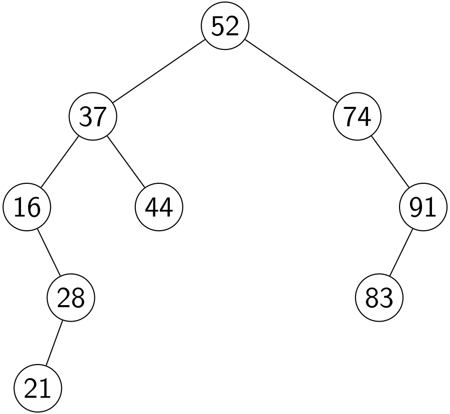
Delete 65




```
class Tree:
    ...
    def delete(self,v):
        if self.isempty():
            return
        if v < self.value:
            self.left.delete(v)
            return
        if v > self.value:
            self.right.delete(v)
            return
        if v == self.value:
            if self.isleaf():
                self.makeempty()
            elif self.left.isempty():
                self.copyright()
            elif self.right.isempty():
                self.copyleft()
            else:
                self.value = self.left.maxval()
                self.left.delete(self.left.maxval())
            return
```

Delete a value v

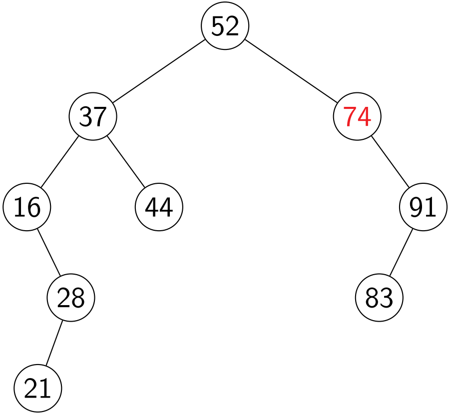
Delete 74




```
class Tree:
    ...
    def delete(self,v):
        if self.isempty():
            return
        if v < self.value:
            self.left.delete(v)
            return
        if v > self.value:
            self.right.delete(v)
            return
        if v == self.value:
            if self.isleaf():
                self.makeempty()
            elif self.left.isempty():
                self.copyright()
            elif self.right.isempty():
                self.copyleft()
            else:
                self.value = self.left.maxval()
                self.left.delete(self.left.maxval())
            return
```

Delete a value v

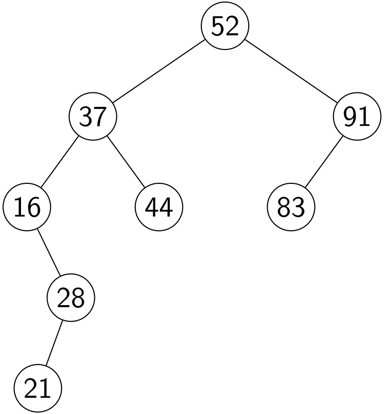
Delete 74




```
class Tree:
    ...
    def delete(self,v):
        if self.isempty():
            return
        if v < self.value:
            self.left.delete(v)
            return
        if v > self.value:
            self.right.delete(v)
            return
        if v == self.value:
            if self.isleaf():
                self.makeempty()
            elif self.left.isempty():
                self.copyright()
            elif self.right.isempty():
                self.copyleft()
            else:
                self.value = self.left.maxval()
                self.left.delete(self.left.maxval())
            return
```

Delete a value v

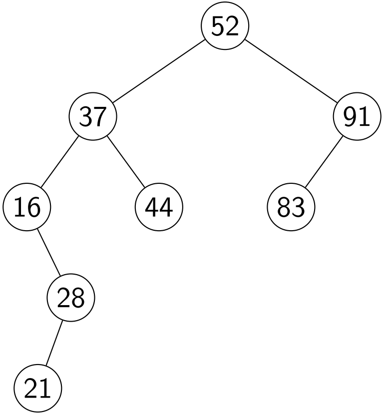
Delete 74




```
class Tree:
    ...
    def delete(self,v):
        if self.isempty():
            return
        if v < self.value:
            self.left.delete(v)
            return
        if v > self.value:
            self.right.delete(v)
            return
        if v == self.value:
            if self.isleaf():
                self.makeempty()
            elif self.left.isempty():
                self.copyright()
            elif self.right.isempty():
                self.copyleft()
            else:
                self.value = self.left.maxval()
                self.left.delete(self.left.maxval())
            return
```

Delete a value v

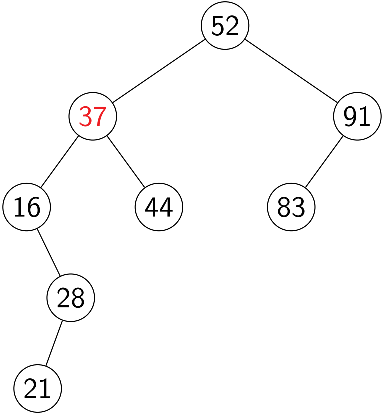
Delete 37




```
class Tree:
    ...
    def delete(self,v):
        if self.isempty():
            return
        if v < self.value:
            self.left.delete(v)
            return
        if v > self.value:
            self.right.delete(v)
            return
        if v == self.value:
            if self.isleaf():
                self.makeempty()
            elif self.left.isempty():
                self.copyright()
            elif self.right.isempty():
                self.copyleft()
            else:
                self.value = self.left.maxval()
                self.left.delete(self.left.maxval())
            return
```

Delete a value v

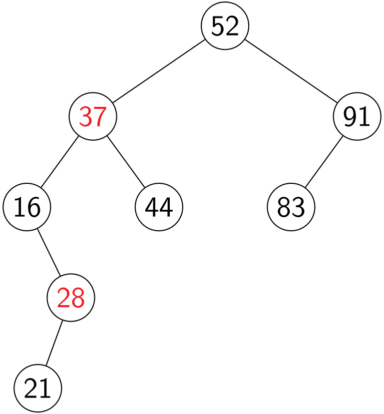
Delete 37




```
class Tree:
    ...
    def delete(self,v):
        if self.isempty():
            return
        if v < self.value:
            self.left.delete(v)
            return
        if v > self.value:
            self.right.delete(v)
            return
        if v == self.value:
            if self.isleaf():
                self.makeempty()
            elif self.left.isempty():
                self.copyright()
            elif self.right.isempty():
                self.copyleft()
            else:
                self.value = self.left.maxval()
                self.left.delete(self.left.maxval())
            return
```

Delete a value v

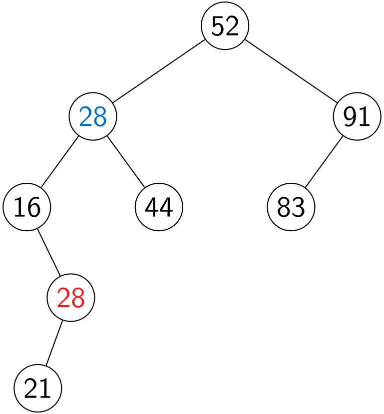
Delete 37




```
class Tree:
    ...
    def delete(self,v):
        if self.isempty():
            return
        if v < self.value:
            self.left.delete(v)
            return
        if v > self.value:
            self.right.delete(v)
            return
        if v == self.value:
            if self.isleaf():
                self.makeempty()
            elif self.left.isempty():
                self.copyright()
            elif self.right.isempty():
                self.copyleft()
            else:
                self.value = self.left.maxval()
                self.left.delete(self.left.maxval())
            return
```

Delete a value v

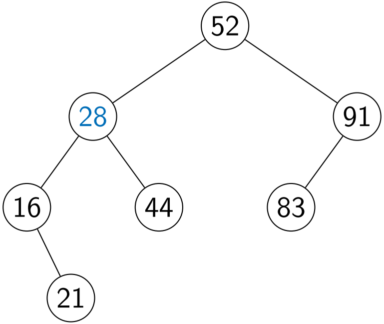
Delete 37




```
class Tree:
    ...
    def delete(self,v):
        if self.isempty():
            return
        if v < self.value:
            self.left.delete(v)
            return
        if v > self.value:
            self.right.delete(v)
            return
        if v == self.value:
            if self.isleaf():
                self.makeempty()
            elif self.left.isempty():
                self.copyright()
            elif self.right.isempty():
                self.copyleft()
            else:
                self.value = self.left.maxval()
                self.left.delete(self.left.maxval())
            return
```

Delete a value v

Delete 37




```
class Tree:
    ...
    def delete(self,v):
        if self.isempty():
            return
        if v < self.value:
            self.left.delete(v)
            return
        if v > self.value:
            self.right.delete(v)
            return
        if v == self.value:
            if self.isleaf():
                self.makeempty()
            elif self.left.isempty():
                self.copyright()
            elif self.right.isempty():
                self.copyleft()
            else:
                self.value = self.left.maxval()
                self.left.delete(self.left.maxval())
            return
```

Delete a value v

```python
class Tree:
    ...
    def delete(self,v):
        if self.isempty():
            return
        if v < self.value:
            self.left.delete(v)
            return
        if v > self.value:
            self.right.delete(v)
            return
        if v == self.value:
            if self.isleaf():
                self.makeempty()
            elif self.left.isempty():
                self.copyright()
            elif self.right.isempty():
                self.copyleft()
            else:
                self.value = self.left.maxval()
                self.left.delete(self.left.maxval())
            return
```

```python
# Convert leaf node to empty node
def makeempty(self):
    self.value = None
    self.left = None
    self.right = None
    return

# Promote left child
def copyleft(self):
    self.value = self.left.value
    self.right = self.left.right
    self.left = self.left.left
    return

# Promote right child
def copyright(self):
    self.value = self.right.value
    self.left = self.right.left
    self.right = self.right.right
    return
```

Complexity

- `find()`, `insert()` and `delete()` all walk down a single path
- Worst-case: height of the tree
- An unbalanced tree with n nodes may have height $O(n)$
- Balanced trees have height $O(\log n)$
- Will see how to keep a tree balanced to ensure all operations remain $O(\log n)$