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

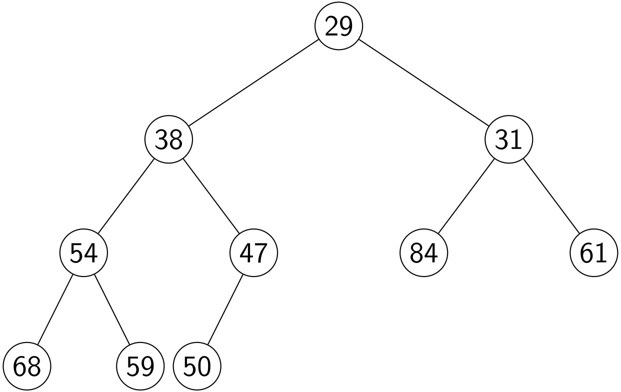
Bottleneck

- 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)
```

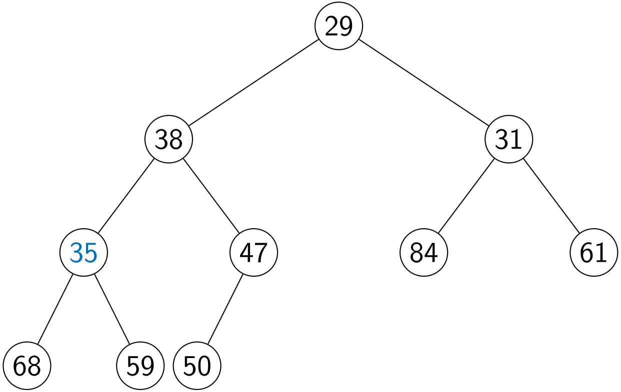
Updating values in a min-heap

- Change 54 to 35



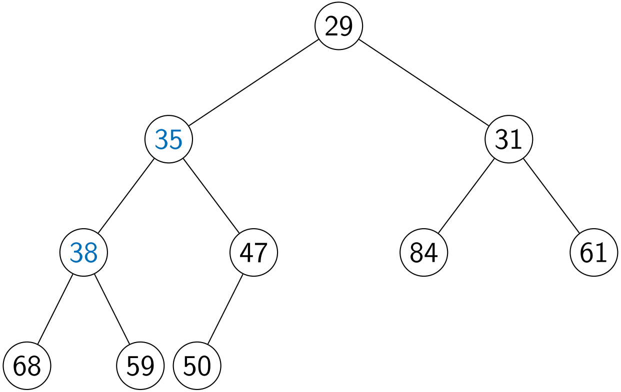
Updating values in a min-heap

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



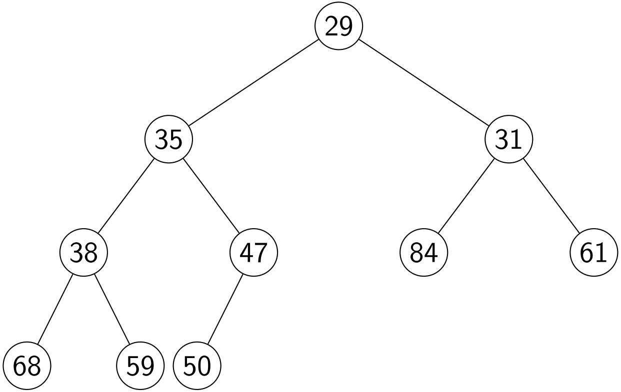
Updating values in a min-heap

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



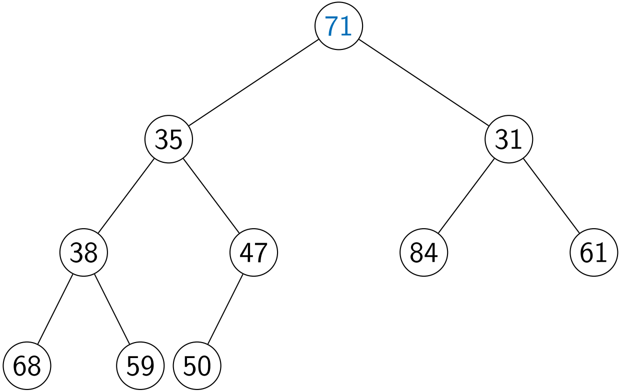
Updating values in a min-heap

- 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



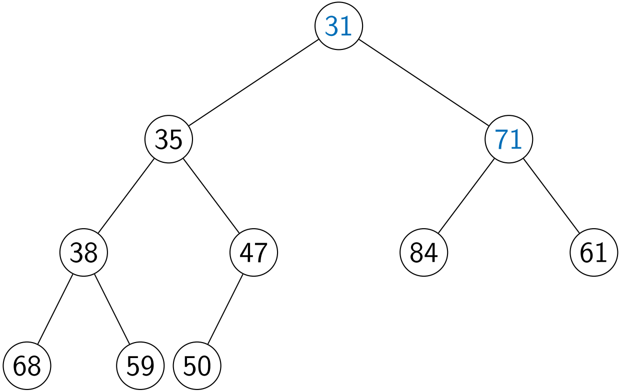
Updating values in a min-heap

- 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



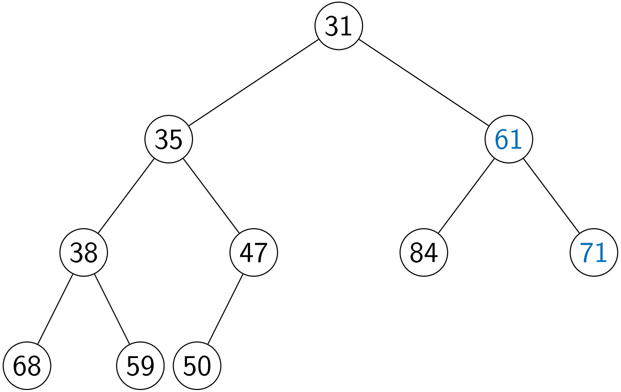
Updating values in a min-heap

- 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()`



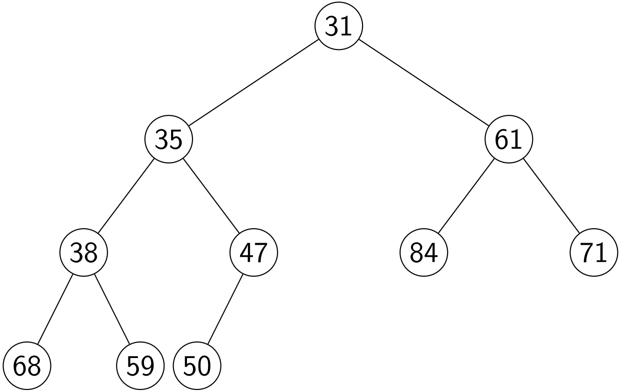
Updating values in a min-heap

- 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()`



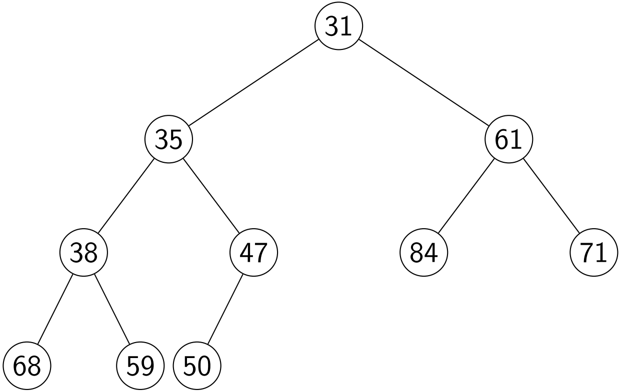
Updating values in a min-heap

- 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?



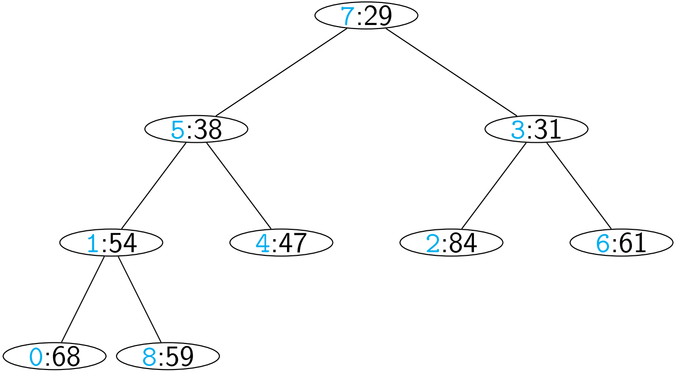
Updating values in a min-heap

- 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

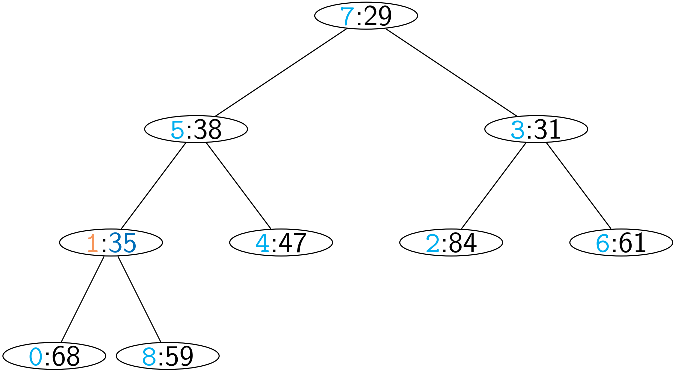
0	1	2	3	4	5	6	7	8
7	3	5	2	4	1	6	0	8

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

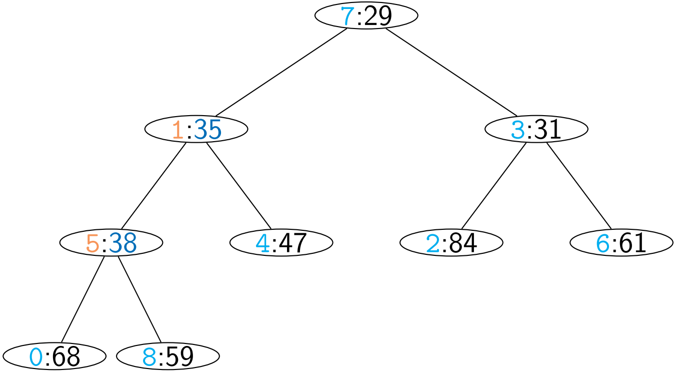
0	1	2	3	4	5	6	7	8
7	3	5	2	4	1	6	0	8

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



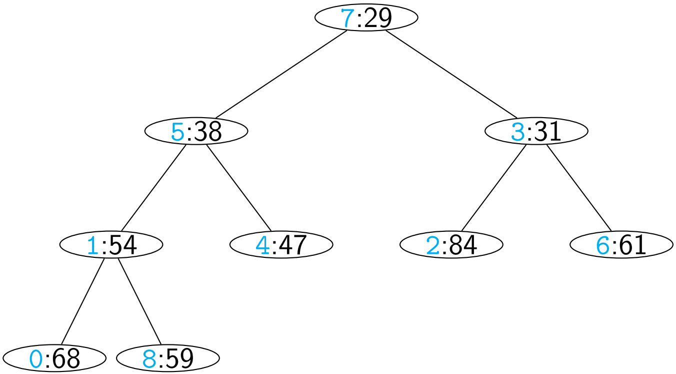
VtoH

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

- 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

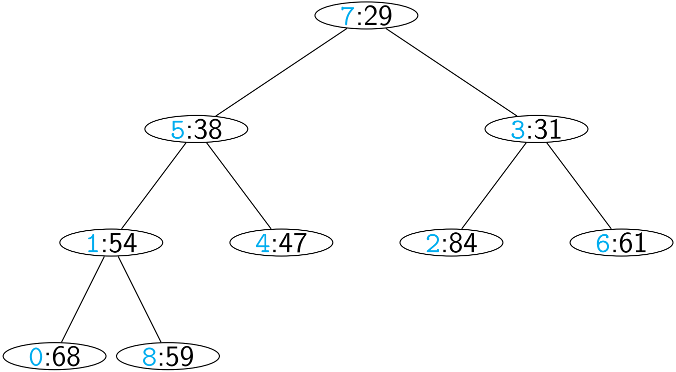
0	1	2	3	4	5	6	7	8
7	3	5	2	4	1	6	0	8

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

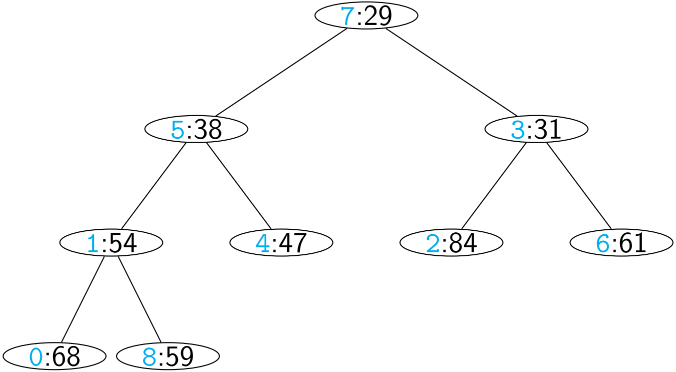
0	1	2	3	4	5	6	7	8
7	3	5	2	4	1	6	0	8

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 $O(n \log n)$