Union-Find Data Structure

Kruskal's Algorithm for Minimum Cost Spanning Tree (MCST)

- Process the 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 Structure

- 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 components containing s
  - `Union(s, s')` - merges components containing `s, s'`

Naive Implementation

- Assume $S = \{0, 1, \ldots, n-1\}$
- Set up an 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)` - $O(n)$

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] += 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] += 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 the larger one
  - If `Size[c] < Size[c']` re-label `c` as `c'`, else re-label `c'` as `c`
- Individual merge operatios can still take time $O(n)$
  - Both `Size[c], Size[c']` could be about $n/2$
  - More careful accounting
- Always merge smaller component into the larger one
- For each `i`, size of `Component[i]` at least doubles each time it is re-labelled
- 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 $2m$ updates are re-labelled
  - 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
  - Amortized 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
- `MakeUnionFind(V)` - each vertex `j` is in component `j`

- Adding an 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
  - $O(n. logn)$ amortized cost overall
- Sorting E takes $O(m. logm)$
  - Equivalently, $O(m. logm)$, since $m \leq n^2$
- Overall time, $O((m + n)logn)$

Summary

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

Priority Queues

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 optimize the following operations
- `delete_max()`
    - Identify and remove item with the 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

- 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 $O(n^2)$

Moving to 2 dimensions

First Attempt

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

Summary

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

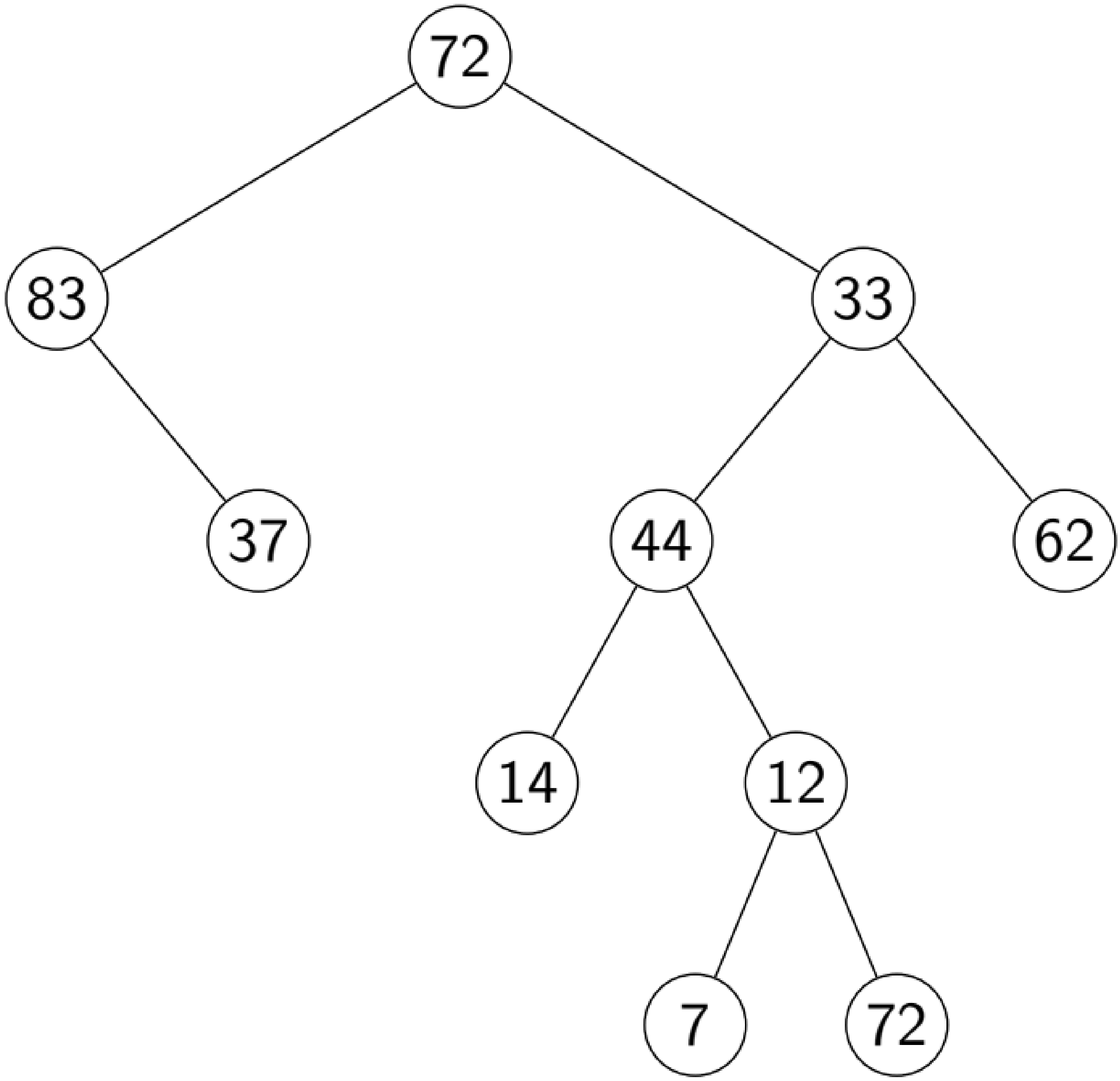
Heaps

Priority Queue

- Need to maintain a collection of items with priorities to optimize 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 oprations
    - $O(N\sqrt{N})$ across N inserts and deletions

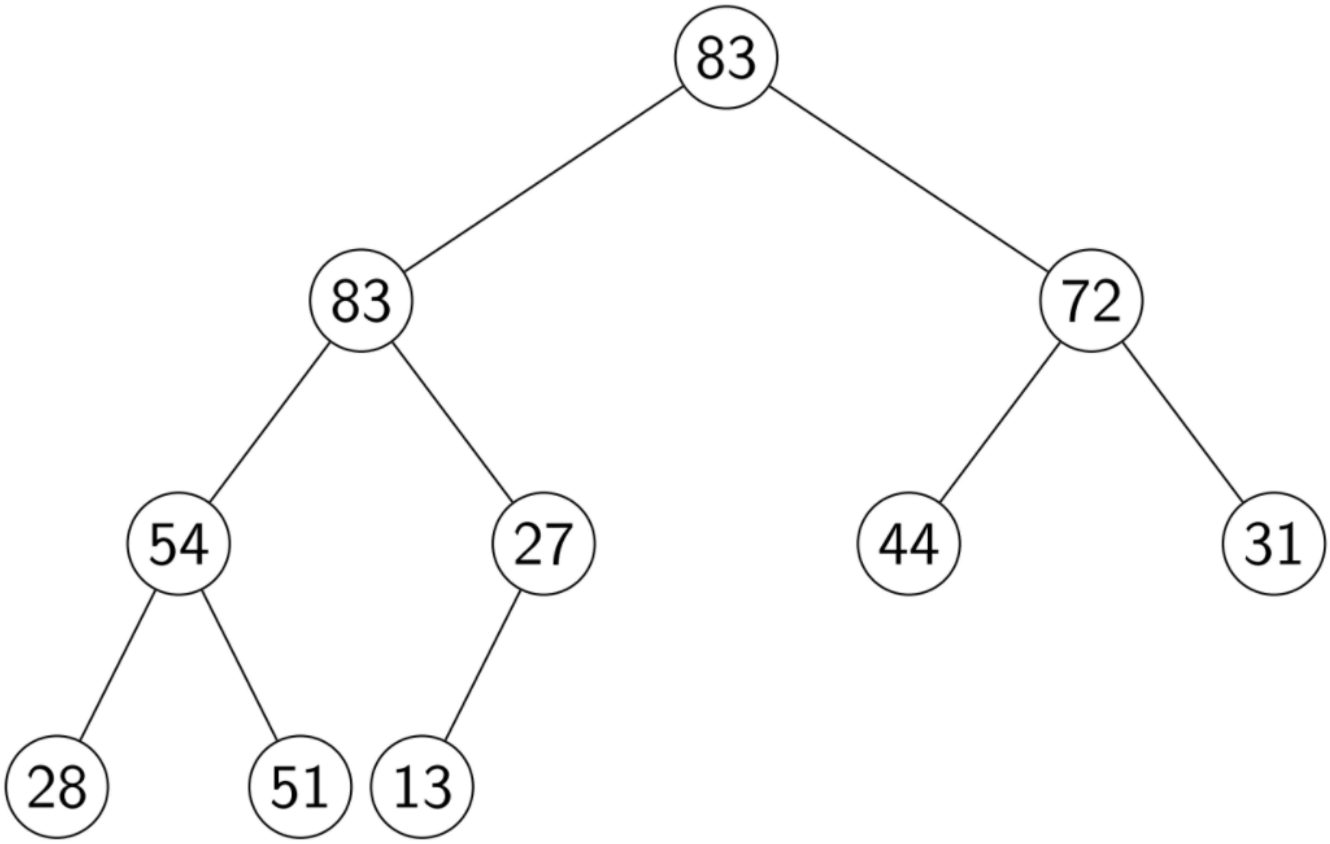
Binary Trees

- 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



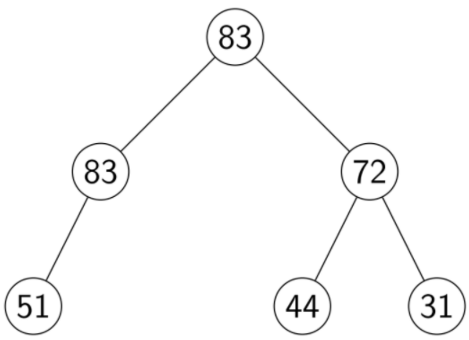
Heap

- Binary tree filled level-by-level, left-to-right
- The value at each node is at least as big as 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

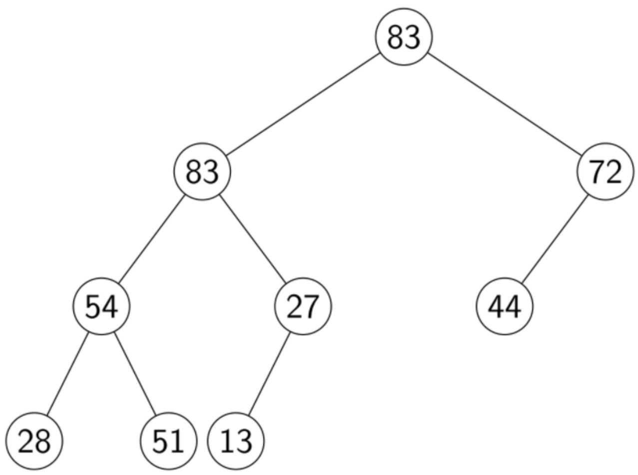


Non-Examples

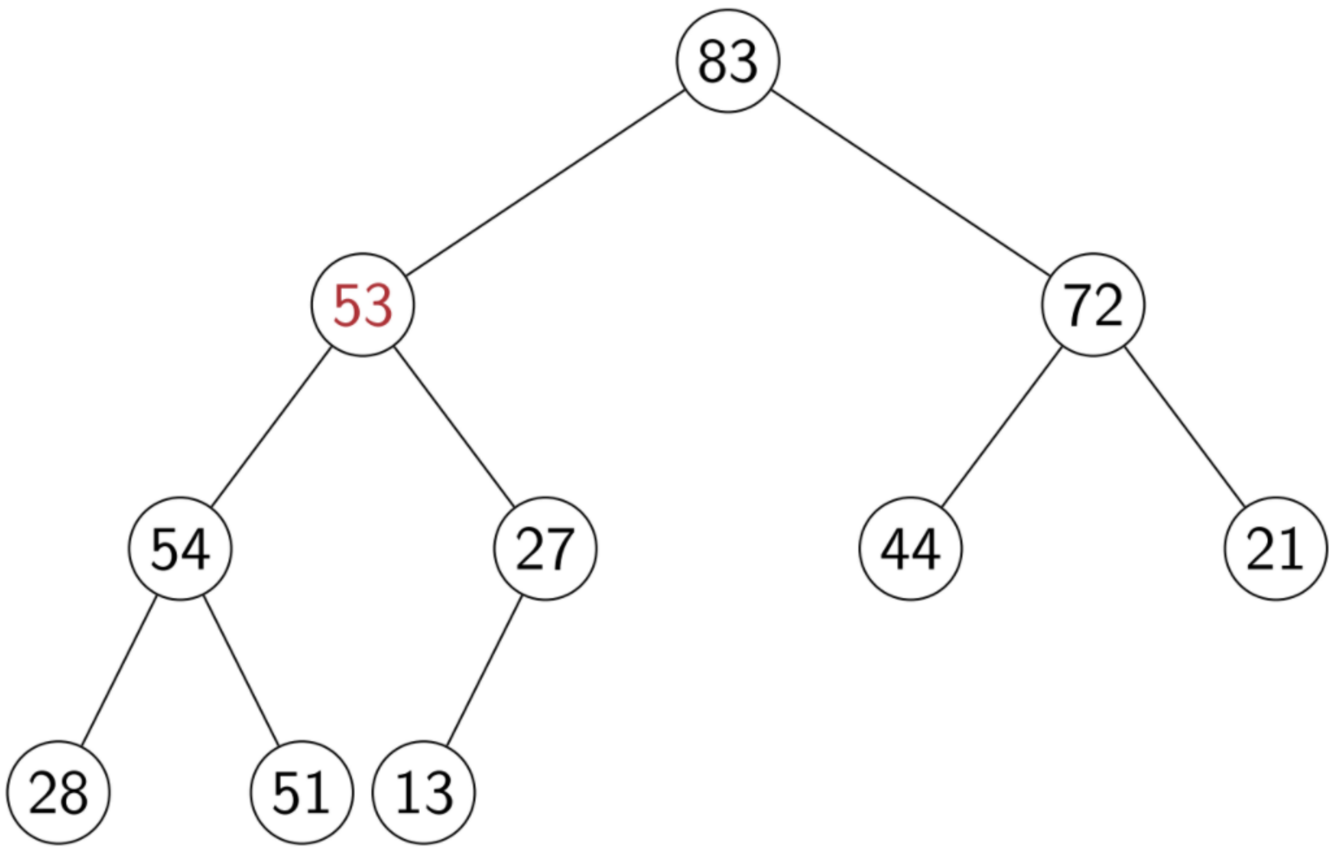
No “holes” allowed



Cannot leave a level incomplete



Heap property is violated



Complexity of `insert()`

- Need to walk up from the leaf to the root
  - Height of the tree
- Number of nodes at level 0 is $2^0 = 1$
- If we fill k levels, $2^0 + 2^1 + ... + 2^{k - 1} = 2^k - 1$ 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
- After we delete one value, tree shrinks
  - Node to delete is right-most 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)$$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[w * 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
- Simple strategy
  - Start with an empty heap
  - Repeatedly apply `insert(vj)`
  - Total time is $O(N.log \ N)$$O(N.log \ N)$

Better `heapify()`

- List `L = [v0, v1, ..., vN]`
- `mid = len(L)//2`, Slice `L[mid:]` has only leaf nodes
  - Already satisfy the 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 the heap peoperty
- However, number of nodes to fix halves
- Second last level, $n/4 \times 1$$n/4 \times 1$ steps
- Third last level, $n/8 \times 2$$n/8 \times 2$ steps
- Fourth last level, $n/16 \times 3$$n/16 \times 3$ steps
- ...
- Cost turns out to be $O(n)$$O(n)$

Summary

- Heaps are a tree implementation of priority queues

    - `insert()` is $O(log \ N)$$O(log \ N)$
    - `delete_max()` is $O(log \ N)$$O(log \ N)$
    - `heapify()` builds a heap in $O(N)$$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

Priority Queues and Heaps

- Priority Queues support the following operations
  - `insert()`
  - `delete_max()` or `delete_min()`
- Heaps are 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 2 dictionaries with vertices as keys
  - `visited`, initially `False` for all `v`
  - `distance`, initially `infinity` for all `v`
- Set `distance[v]` to 0
- Repeat, untill all the reachable vertices are visited
  - Find unvisited vertex `nextv` with minimum distance
  - Set `visited[nextv]` to `True`
  - Re-compute `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
```

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 the neighbours
    - Unvisited neighbour's distances are inside the min-heap
    - Updating a value is not a basic heap operation

Heap sort

- Start with an un-ordered 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 $O(log\ n)$
- 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 $O((m+n).\,log\ n)$
- Heaps can also be used to sort a list in place in $O(n.\,log\ n)$

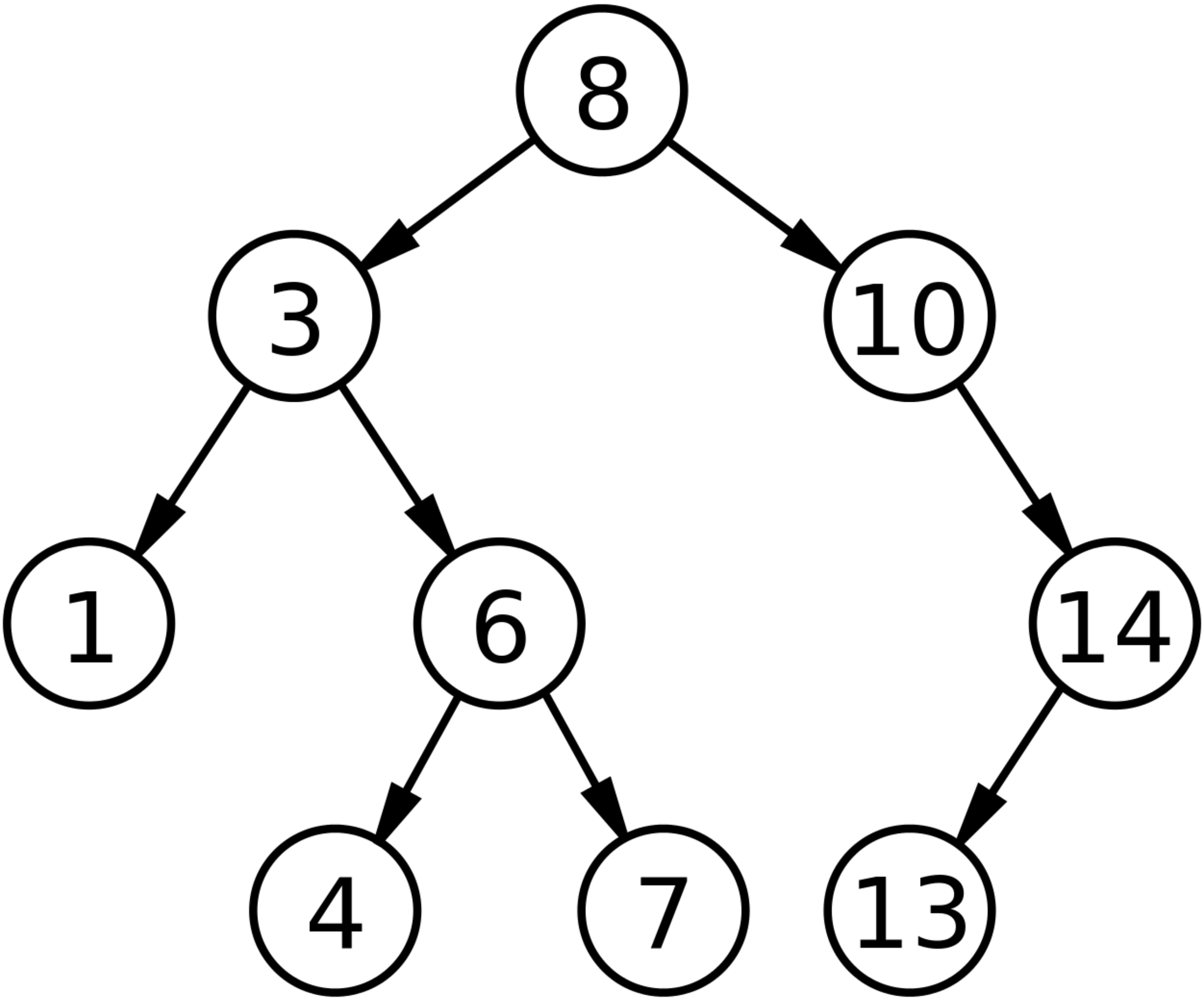
Search Trees

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 the value v
  - All values in the left sub-tree are $< v$
  - All values in the right sub-tree are $> v$
- No duplicate values



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 tree has all fields `None`
- Left has a non-empty `value` and empty `left` and `right`

```
class Tree:
```

```
  # Constructor
  def __init__(self, init_val = None):
    self.value = init_val

    if self.value:
      self.left = Tree()
      self.right = Tree()
    else:
      self.left = None
      Self.right = None

    return

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

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

In-order traversal

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

```
class Tree:
  ...
  # In-order traversal
  def in_order(self):
    if self.is_empty():
      return []
    else:
      return self.left.in_order() + [self.value] + self.right.in_order()

  # Display the tree as a string
  def __str__(self):
    return str(self.in_order())
```

Find a value `v`

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

```
class Tree:
  ...
  # Check if the value v occurs in the tree
```

```
  def find(self, v):
    if self.is_empty():
      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 the left most node in the tree
- Maximum is the right most node in the tree

```
class Tree:
  ...
  def min_val(self):
    if self.left.is_empty():
      return self.value
    else:
      return self.left.min_val()

  def max_val(self):
    if self.right.is_empty():
      return self.value
    else:
      return self.right.max_val()
```

Insert a value `v`

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

```
class Tree:
  ...
  def insert(self, v):
    if self.is_empty():
      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 sub-tree
- Otherwise, replace `v` with `self.left.max_val()` and delete `self.left.max_val()`
    - `self.left.max_val()` has no right child

```
class Tree:
  ...
  def delete(self, v):
    if self.is_empty():
      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.is_leaf():
        self.make_empty()
      elif self.left.is_empty():
        self.copy_right()
      elif self.right.is_empty():
        self.copy_left()
      else:
        self.value = self.left.max_val()
        self.left.delete(self.left.max_val())
      return

  # Convert left node to empty node
  def make_empty(self):
    self.value = None
    self.left = None
    self.right = None
    return

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

```
    self.left = self.left.left
    return

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

Summary

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