Question of the day

- How can we organize data to efficiently locate and update the most important elements?
- How can we efficiently store and update group memberships?

Priority queues and Union-Find

William Hendrix

Lecture 5

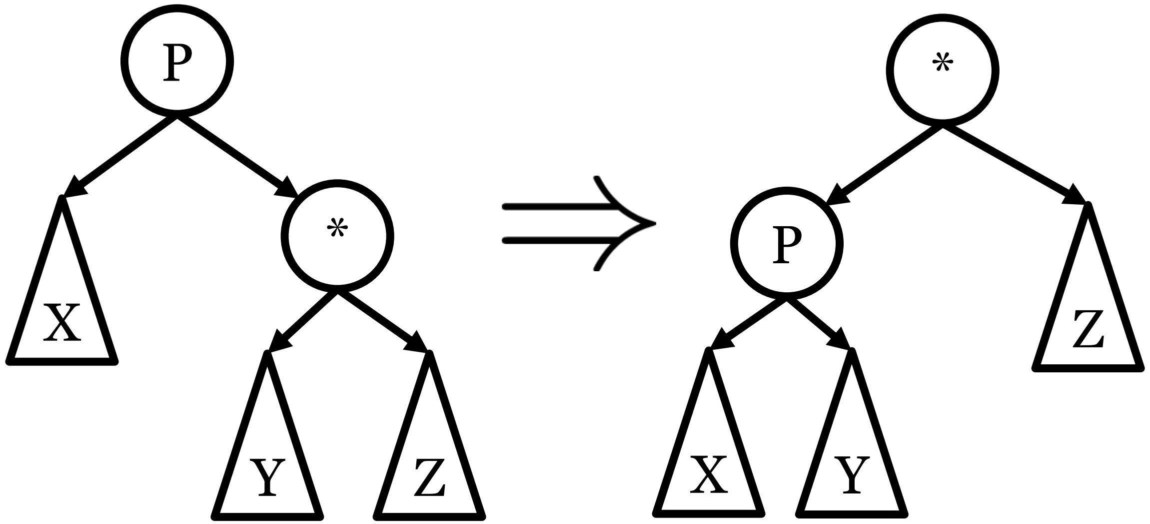
Outline

- Review
  - AVL trees
  - Hash tables
  - Sets
  - Maps
- Priority queues
- Heaps
- Union-Find

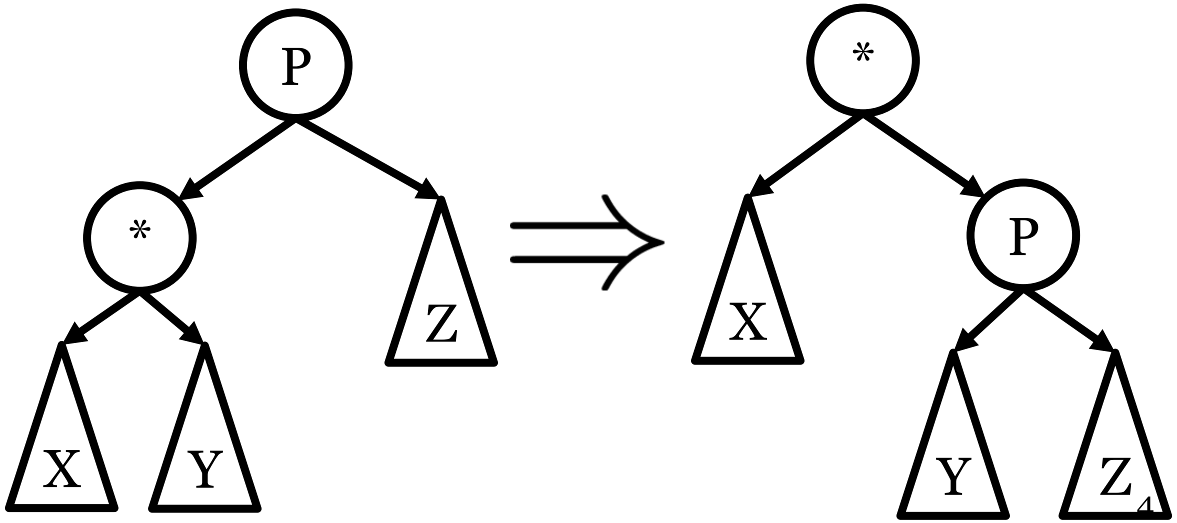
AVL tree review

- Self-balancing binary search tree
- All ops guaranteed O(lg *n*)
- Nodes keep track of *balance*
  - height(right) - height(left)
  - Always 0 or ±1
- Performs *rotations* as nodes are inserted or deleted to maintain balance
  - 4 cases: left-left, left-right, right-left, right-right

<u>Left rotation</u>



<u>Right rotation</u>



Hash table review

- Sparse array-based structure
  - Values inserted based on *hash function*
- **Separate chaining:** array of linked lists
- **Open addressing:** insert at "next available" space
  - Quadratic probing and double hashing reduce congestion
- Rehash once *load factor* exceeds threshold
  - **Load factor:** size / capacity
  - Double capacity (roughly) and reinsert using hash function
    - Often chosen to be prime
- Hash functions
  - Need to be fast, distribute values evenly, and separate nearby values
  - Often use multiplication, polynomials, or bitwise ops
  - Mod at end to ensure range is 0 to *cap* – 1
  - *Multi-byte inputs*: multiply or rotate before incorporating next values

Set review

- ADT for storing and retrieving values
  - May allow or disallow duplicates
- **Main operations:**
  - *Search(x)*: returns whether *x* is in the set
  - *Insert(x)*: adds *x* to the set (may or may not allow duplicates)
  - *Delete(x)*: removes *x* from the set
- **Two main implementations:** balanced BST and hash table
  - Hash table "usually" faster
    - $\Theta(1)$ expected complexity
    - Assumes $\Theta(1)$ collisions
  - BST has better worst-case complexity
    - $O(\lg n)$ vs. $O(n)$
  - BST can access elements in sorted order
    - min(), max(), predecessor(), successor()

Maps review

- Stores set of associations
- **Main operations:**
  - *Insert(key, value)*: associates *value* to *key*
  - *Delete(key)*: removes any association with *key*
  - *Search(key)*: returns value associated with *key*
- **Implementations**
  - Array map:
    - Store value in `arr[key]`
    - $\Theta(1)$ worst case (*very fast!*)
    - Keys must be relatively small ints
  - Hash map:
    - Insert (key, value) pairs into hash table
      - Only hash key
    - $\Theta(1)$ expected complexity
  - Tree map:
    - Insert (key, value) pairs into BBST based on key
    - $O(\lg n)$ worst case

Maps

- Abstraction of a function
- Main operations
  - **Insert(x, y):** declares that $f(x) = y$
  - **Delete(x):** declares that $f(x)$ does not have a value
  - **Search(x):** returns y such that $f(x) = y$, or NIL if $f(x)$ does not have a value
- **Example:** letter frequencies
  - Problem: count how many times a letter appears in a given text
    - Used in cryptography
  - Sample output

E	T	A	O	I	N	S	R	H	…
12	9	8	7	7	6	6	6	6	…

  - Need to associate count with letter
  - $f(\mathrm{E}) = 12$, etc.

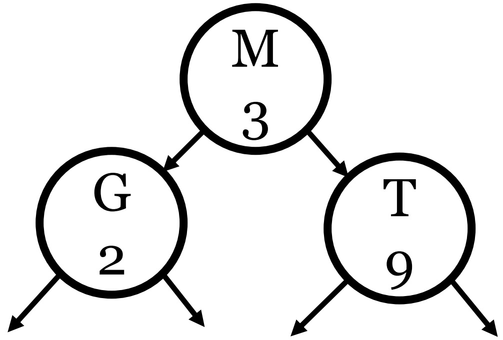
Map implementation

- Two main implementations
- Array-based map
  - Array of all possible x values
  - Stores $f(x)$ in arr[x] (NIL if not initialized)

8	2	3	4	12	2	2	…
A	B	C	D	E	F	G	…

  - All main operations are constant time
  - Only useful when input domain is small

- Set-based map
  - A.k.a., hash map
  - Set of ordered (x, y) pairs
  - Pairs added/searched according to x value
  - **Search** returns associated y value
  - Time complexity determined by hash table or BBST



Map complexity

Operation	Array-based map	Set-based (hash table)	Set-based (BBST)
Insert(x, y)	$\Theta(1)$	$\Theta(1)$*	$O(\lg n)$
Delete(x)	$\Theta(1)$	$\Theta(1)$*	$O(\lg n)$
Search(x)	$\Theta(1)$	$\Theta(1)$*	$O(\lg n)$
Build()	$\Theta(D)$	$\Theta(n)$	$O(n \lg n)$

D: size of domain (*x* values)
* Expected complexity for hash table

The power of maps

- Maps are very useful for storing values that we compute repeatedly
  - Especially when we can use direct maps
- **Example:** Discrete Fourier Transform
  - Given array *x* compute transformed array *c* such that

$$c_k = \sum_{j=1}^{n} x_j e^{\boxed{jk(-2\pi i/n)}} \longrightarrow \text{Store values in lookup table}$$

- Can also improve best-case performance for <u>any</u> algorithm

1. Build a map that contains problem instances and solutions
2. Before running another algorithm, test whether input is in map
3. If so, return the answer

- Best case typically constant or linear time
- Best case analysis not useful to compare algorithm quality

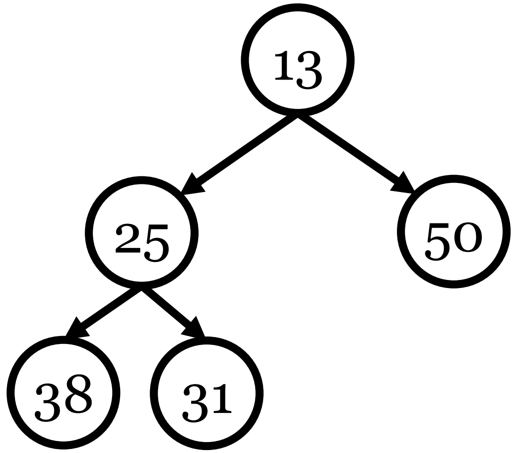
Priority queues

- Abstract data type
- Similar to a queue, but returns elements in *priority order*
  - Min-first or max-first
  - Very good at finding min/max

- 3 main operations (min)
- **Insert(x)**
  - Adds another element
- **Min()**
  - Returns min
- **DeleteMin()**
  - Returns min and deletes

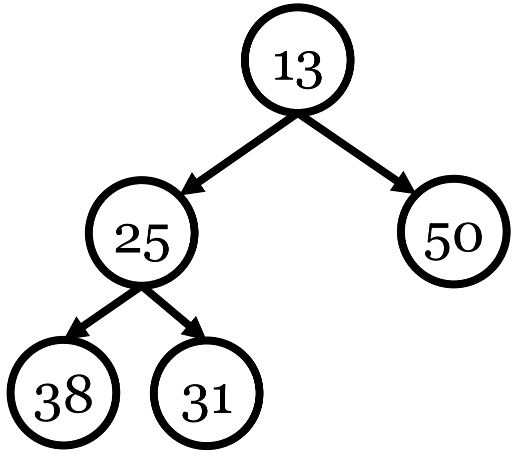
Heaps

- Main implementation of priority queue
- Complete binary tree that satisfies *heap property*
  - Min heap: every parent is ≤ its children
  - Max heap: every parent is ≥ its children
  - Unlike BST, left and right children not related
  - Also, children fill last level left-to-right
- **Min():**
  - Return root



Heap insertion

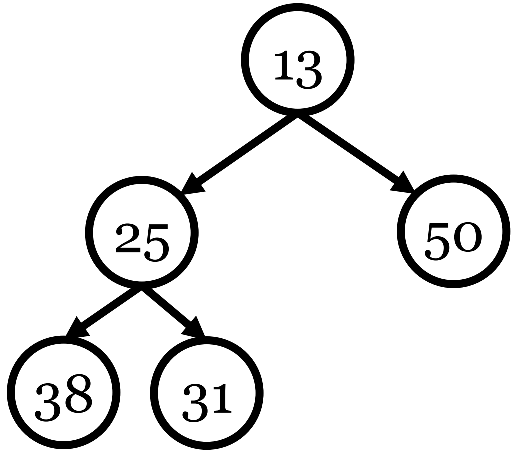
- **Insert(x):**
- Add x into next position on bottom level
  - Tree size dictates where to go
  - Increment size
  - Convert size to binary
  - Skip to just past first 1
  - Go left on 0
  - Go right on 1
- Restore heap property



Size = 5

Heap insertion

- **Insert(x):**
- Add x into next position on bottom level
  - Tree size dictates where to go
  - Increment size
  - Convert size to binary
  - Skip to just past first 1
  - Go left on 0
  - Go right on 1
- Restore heap property

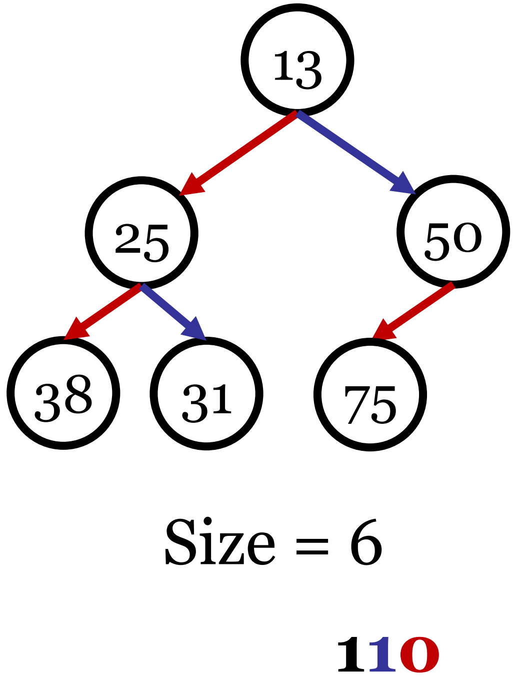


Size = 6

110

Heap insertion

- **Insert(x):**
- Add x into next position on bottom level
  - Tree size dictates where to go
  - Increment size
  - Convert size to binary
  - Skip to just past first 1
  - Go left on 0
  - Go right on 1
- Restore heap property

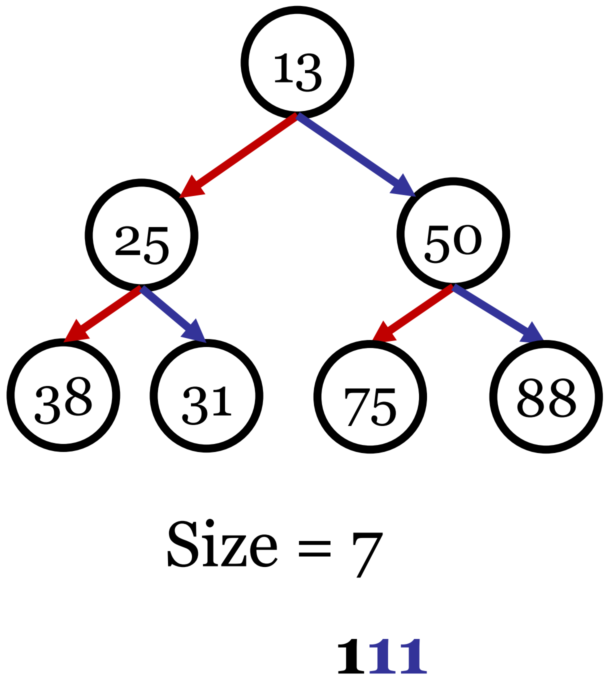


Heap insertion

- **Insert(x):**
- Add x into next position on bottom level
  - Tree size dictates where to go
  - Increment size
  - Convert size to binary
  - Skip to just past first 1
  - Go left on 0
  - Go right on 1

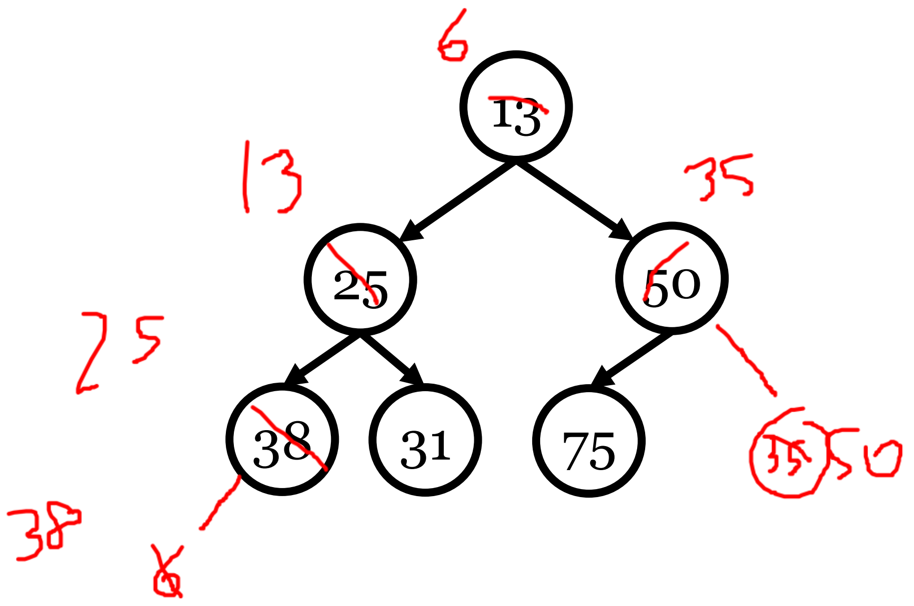
Bitwise ops not tested

- Restore heap property
  - Swap with parent if out-of-order
  - Repeat until satisfied or at root
  - "Percolate up"



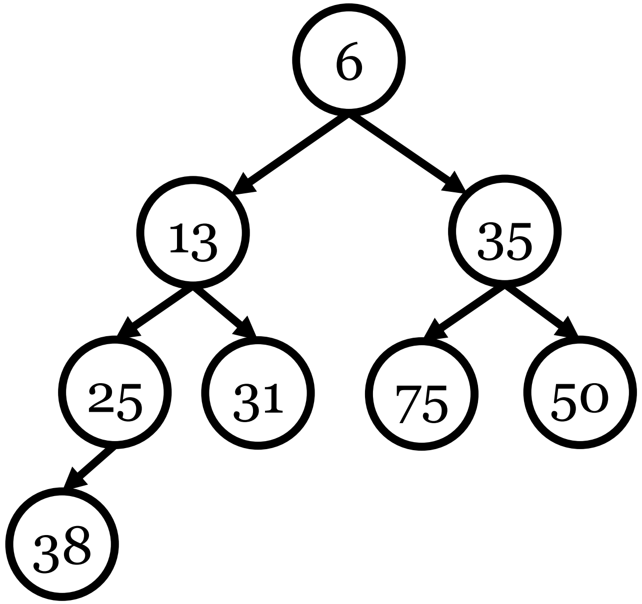
Heap example

- Add 35 to heap at right
- Then add 6



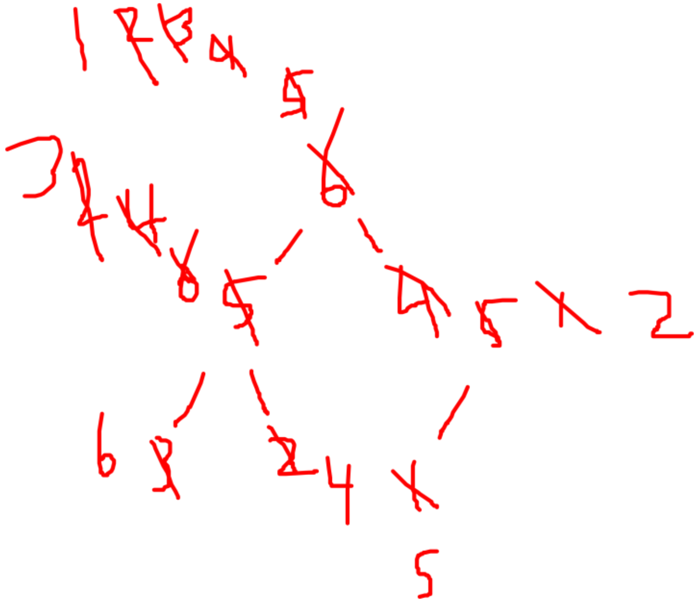
Heap example

- Add 35 to heap at right
- Then add 6



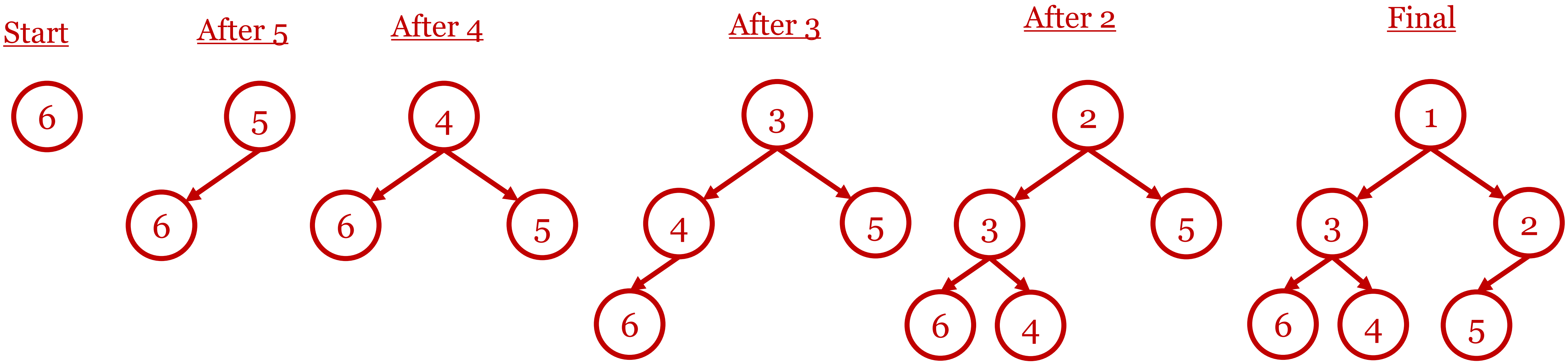
Heap exercise

- Sketch the result of inserting 6, 5, 4, 3, 2, and 1 into an empty min-heap



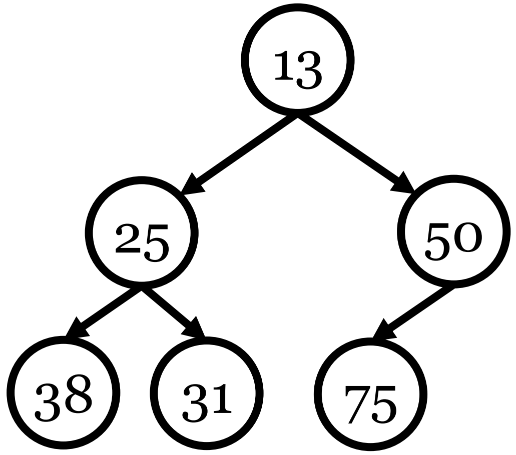
Heap exercise

- Sketch the result of inserting 6, 5, 4, 3, 2, and 1 into an empty min-heap



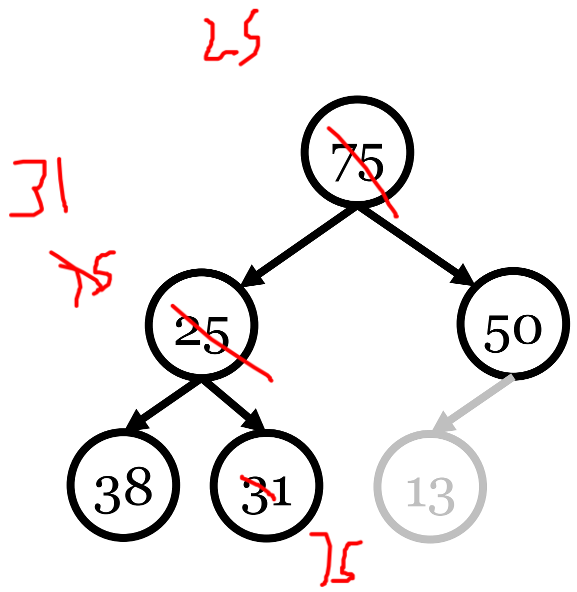
Heap deletion

- **DeleteMin():**
- Swap root value with last node
- Delete last node



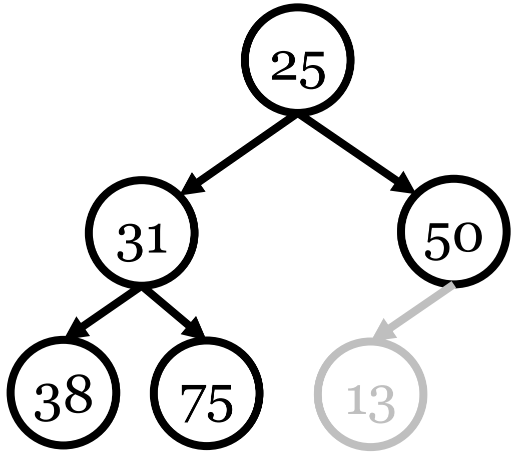
Heap deletion

- **DeleteMin():**
- Swap root value with last node
- Delete last node
- Fix heap property at root
  - Swap root with min child
    - Max child for max-heap
  - Stop if node greater than both children or at leaf
  - "Percolate down"



Heap deletion

- **DeleteMin():**
- Swap root value with last node
- Delete last node
- Fix heap property at root
  - Swap root with min child
    - Max child for max-heap
  - Stop if node greater than both children or at leaf
  - "Percolate down"

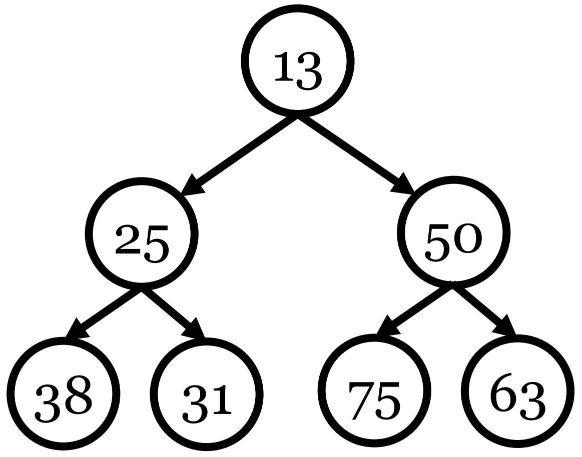


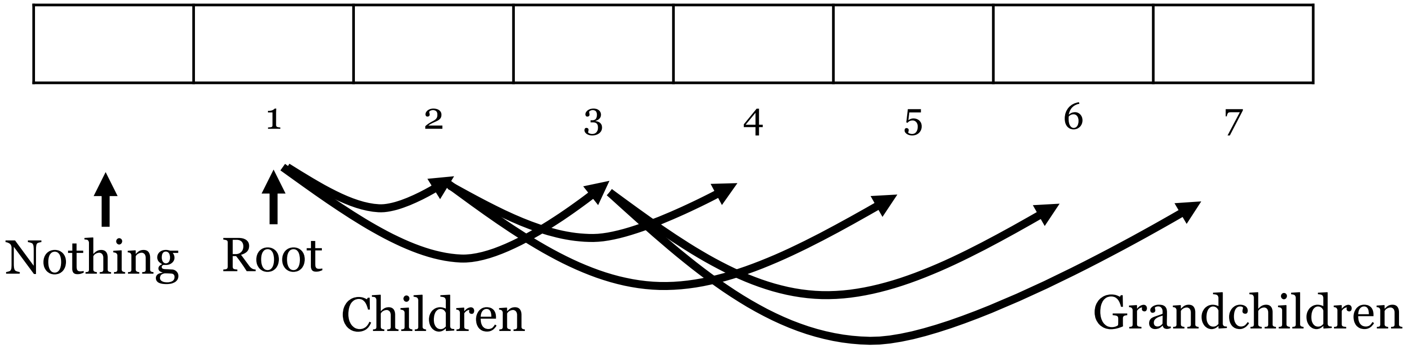
Heap complexity

- **Min():**
  - Return root
  - O(1)
- **Insert(x):**
  - Scan to bottom of tree
  - Percolate up
    - Worst case: go back to root
  - O(h) = O(lg n)
- **DeleteMin():**
  - Scan to bottom
  - Swap
  - Percolate down
    - Worst case: go down to leaf
  - O(lg n)

Array-based heap

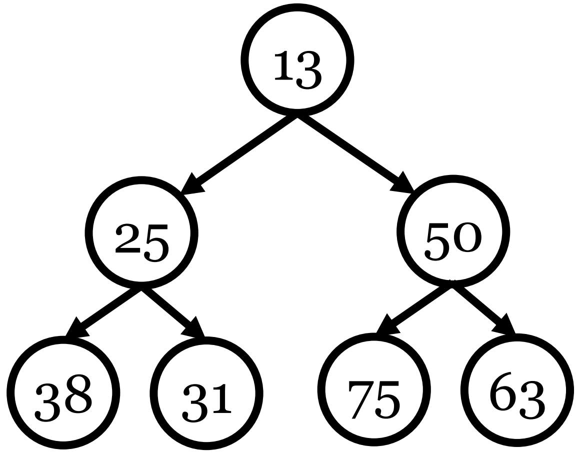
- Preferred implementation for a heap
- Store node values in an array
- Root at index 1
- Children of index *i* at 2*i* and 2*i*+1
  - Parent at floor(*i*/2)
- Complete tree fills array with no gaps or overlap

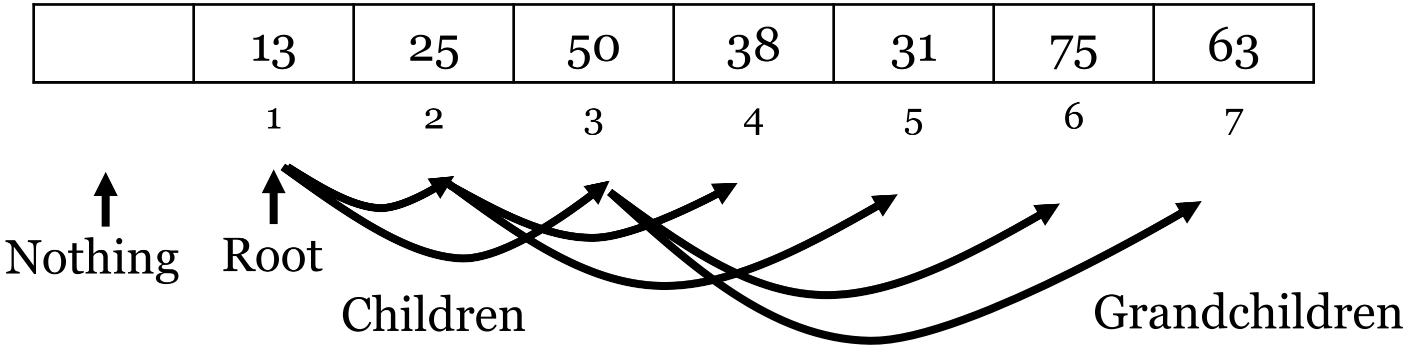




Array-based heap

- Store node values in an array
- Root at index 1
- Children of index *i* at 2*i* and 2*i*+1
  - Parent at floor(*i*/2)
- Complete tree fills array with no gaps or overlap
- Operations mostly the same
- **Min():** return arr[1]
- **Insert(x):**
  - Append to array
  - Percolate up
  - Increment size
  - Double capacity as needed
- **DeleteMin():**
  - Swap arr[1] and arr[size]
  - Percolate arr[1] down
  - Decrement size and return arr[size+1]





Heapification

- Convert unsorted array into heap
  - Faster than multiple calls to insert

- **Algorithm:**
  - Call PercolateDown(i) from end to beginning
  - *Optimization:* don't percolate the leaves down
    - Avoids half of the heap

Algorithm: Heapify(i)
for $i = \lfloor n/2 \rfloor$ to 1 **step** -1 **do**
  PercolateDown(i)
end

- **Complexity:** *Θ(n)* time
  - *Intuition:* half have no children, half of rest have 1 child, etc.
    - Only root has lg *n* levels below it
    - Θ(1) "on average"

Priority queue implementations

Operation	Heap	Unsorted array	Sorted array	Balanced BST	Fibonacci heap
Insert(x)	*Θ(lg n)**	*Θ(1)**	*Θ(n)**	*O(lg n)*	*Θ(1)*
Max()	*Θ(1)*	*Θ(1)*	*Θ(1)*	*Θ(1)*	*Θ(1)*
DeleteMax()	*Θ(lg n)*	*Θ(n)*	*Θ(1)*	*O(lg n)*	*Θ(lg n)**
Build/Heapify	*Θ(n)*	*Θ(1)*	*Θ(n lg n)*	*Θ(n lg n)*	*Θ(n)*

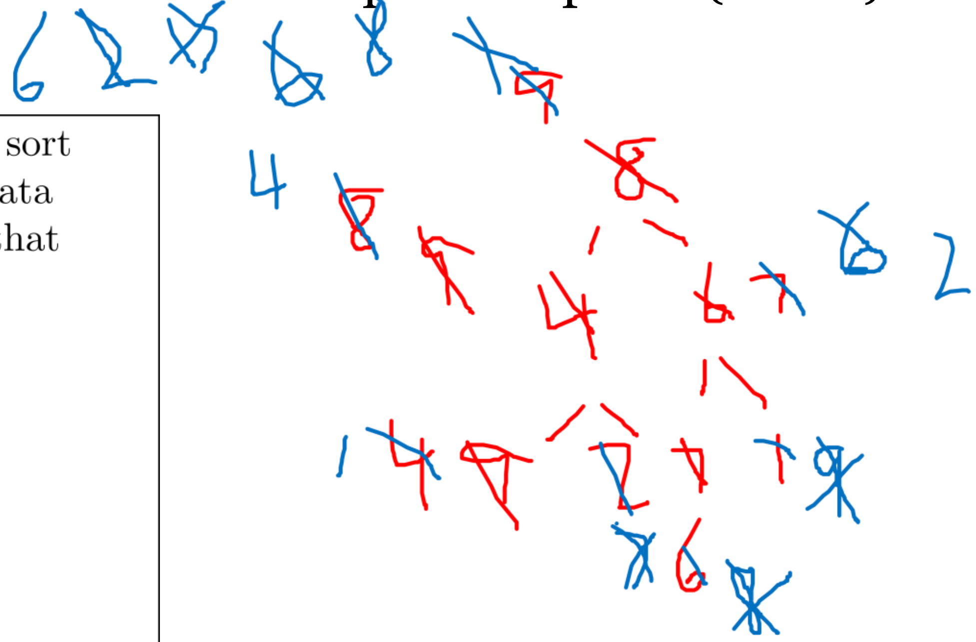
* amortized time

- BST has similar complexity, but higher coefficients
- Great at finding max (or min)
- Other operations (min/max, search, predecessor, etc.) are not good
  - Min-max heap can do either, but is more complex
- Fibonacci heap has even better complexity
  - More complex, higher coefficients, less space efficient
  - Fairly slow unless data is quite large

Heap exercise

1. Draw the max heap that is built from the array {8, 4, 6, 9, 2, 7, 1}.
2. Draw this heap at the end of every iteration of the **for** loop in HeapSort (below). You may ignore the effect of line 5.

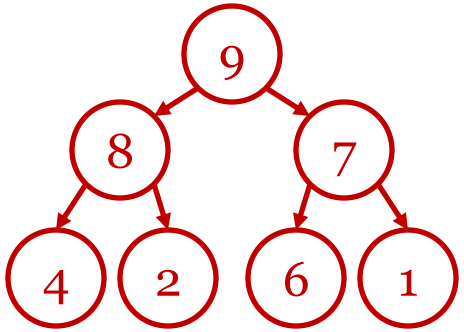
```
  Input: data: an array of integers to sort
  Input: n: the number of values in data
  Output: permutation of data such that
          data[1] ≤ ... ≤ data[n]
1 Algorithm: HeapSort
2 data = MaxHeap.Build(data)
3 for i = n to 2 step −1 do
4 |   m = data.DeleteMax()
5 |   data[i] = m
6 end
7 return data
```



Heap sample solution

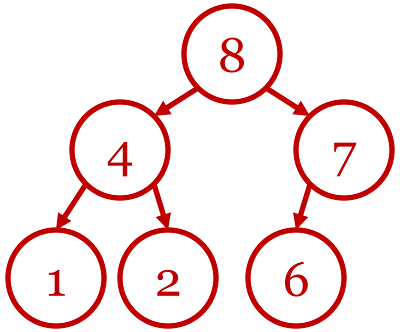
Tree view | **Array view**

Beginning:



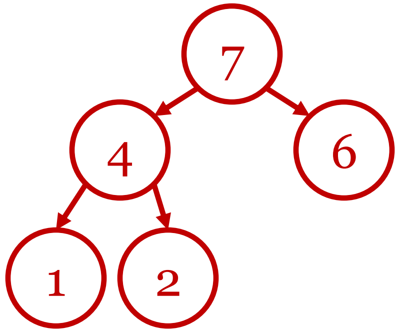
9	8	7	4	2	6	1

After iter 1:



8	4	7	1	2	6	9

After iter 2:

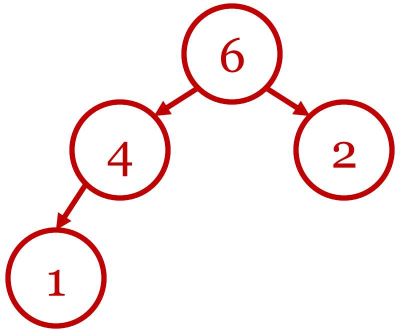


7	4	6	1	2	8	9

Heap sample solution

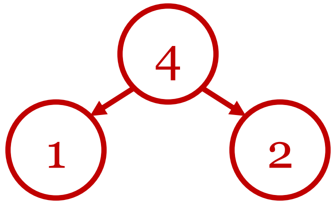
Tree view | **Array view**

After iter 3:



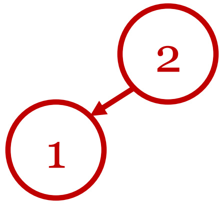
6	4	2	1	7	8	9

After iter 4:



4	1	2	6	7	8	9

After iter 5:



2	1	4	6	7	8	9

After iter 6:

1	2	4	6	7	8	9

Union-Find data structure

- A.k.a., disjoint set data structure
- **Purpose:** represent partition of dataset
  - Identify whether elements belong to the same subset or not

- **Operations**
  - Initialize(n): set up each element (1..n) in its own subset
  - Find(x): return a partition ID for a given element
  - Union(x, y): combine subsets containing x and y together

- **Representation:** array of "pointers" (integers)
  - Find: follow pointers until you find a self-loop
    - Self-loop is partition ID ("root")
  - Union: point Find(a) to b

Union-Find example

- Show how the data structure changes after each of the following iterations:
  - Initialize(7)
  - Union(1, 2)
  - Union(1, 4)
  - Union(6, 7)
  - Union(3, 5)
  - Union(3, 6)
  - Find(3)

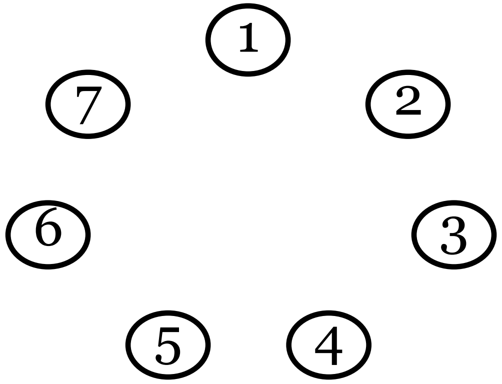
Union-Find example

- Show how the data structure changes after each of the following iterations:
  - Initialize(7)
  - Union(1, 2)
  - Union(1, 4)
  - Union(6, 7)
  - Union(3, 5)
  - Union(3, 6)
  - Find(3)

Array:

1	2	3	4	5	6	7

Pointers:



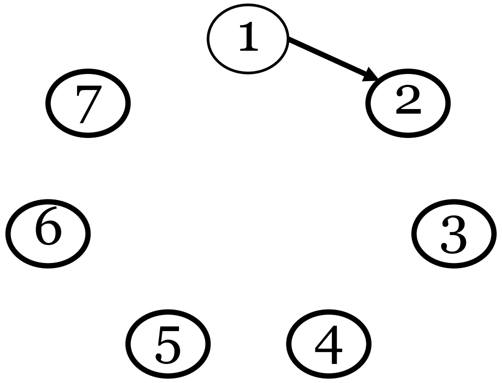
Union-Find example

- Show how the data structure changes after each of the following iterations:
  - Initialize(7)
  - Union(1, 2)
  - Union(1, 4)
  - Union(6, 7)
  - Union(3, 5)
  - Union(3, 6)
  - Find(3)

Array:

2	2	3	4	5	6	7

Pointers:



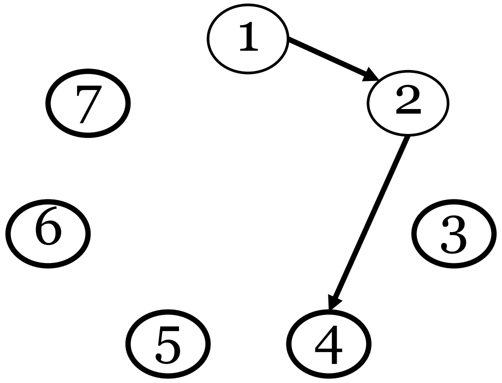
Union-Find example

- Show how the data structure changes after each of the following iterations:
  - Initialize(7)
  - Union(1, 2)
  - Union(1, 4)
  - Union(6, 7)
  - Union(3, 5)
  - Union(3, 6)
  - Find(3)

Array:

2	4	3	4	5	6	7

Pointers:



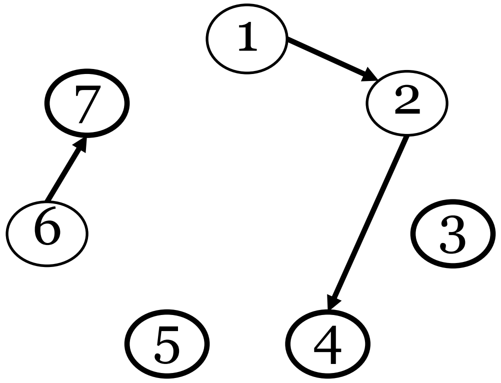
Union-Find example

- Show how the data structure changes after each of the following iterations:
  - Initialize(7)
  - Union(1, 2)
  - Union(1, 4)
  - Union(6, 7)
  - Union(3, 5)
  - Union(3, 6)
  - Find(3)

Array:

2	4	3	4	5	7	7

Pointers:



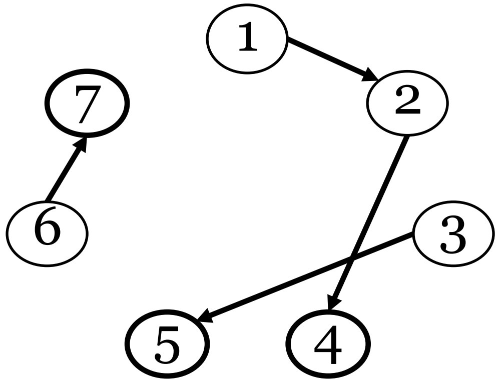
Union-Find example

- Show how the data structure changes after each of the following iterations:
  – Initialize(7)
  – Union(1, 2)
  – Union(1, 4)
  – Union(6, 7)
  – Union(3, 5)
  – Union(3, 6)
  – Find(3)

Array:

2	4	5	4	5	7	7

Pointers:



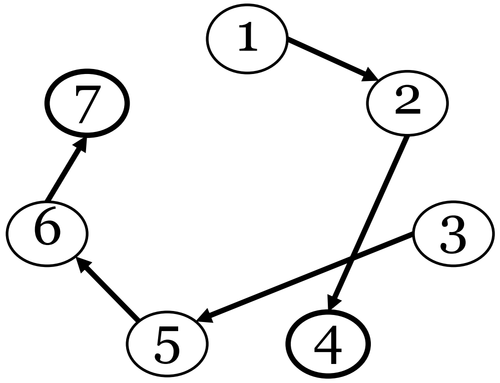
Union-Find example

- Show how the data structure changes after each of the following iterations:
  - Initialize(7)
  - Union(1, 2)
  - Union(1, 4)
  - Union(6, 7)
  - Union(3, 5)
  - Union(3, 6)
  - Find(3)

Array:

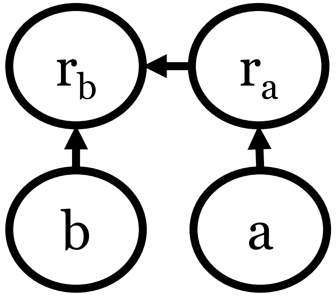
2	4	5	4	6	7	7

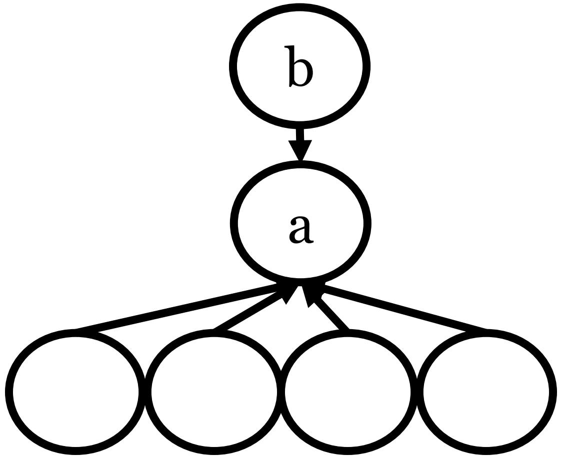
Pointers:

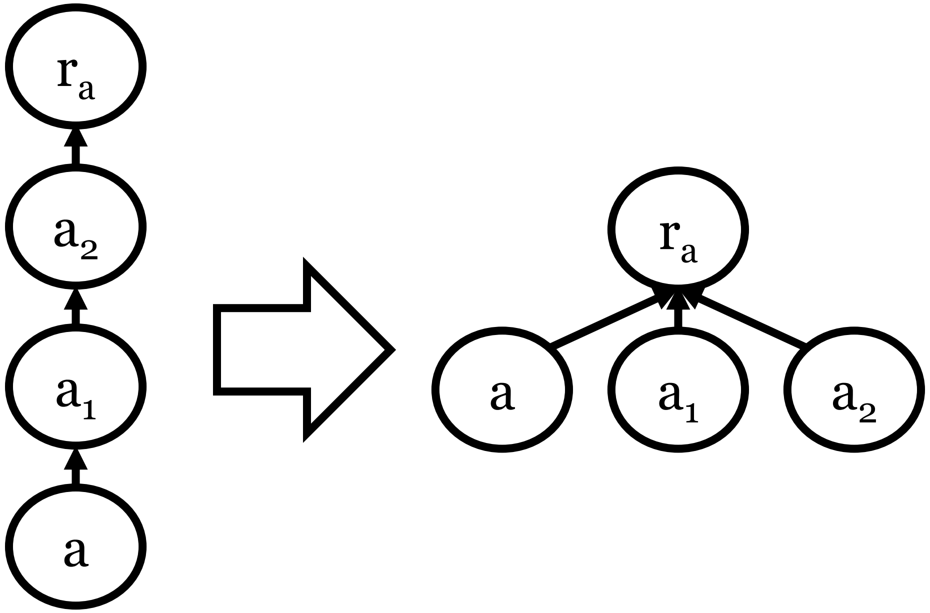


Optimizing Union-Find

- Union complexity depends on Find
- Find complexity depends on height of tree
- Worst case: $\Theta(n)$
- First idea: add Find(a) to Find(b) (or vice versa)
- Second idea: add the smaller tree to the larger
- Third idea: flatten structure when we call Find()







Optimized example

- Show how the data structure changes after each of the following iterations:
  - Initialize(7)
  - Union(1, 2)
  - Union(1, 4)
  - Union(6, 7)
  - Union(3, 5)
  - Union(3, 6)
  - Find(3)

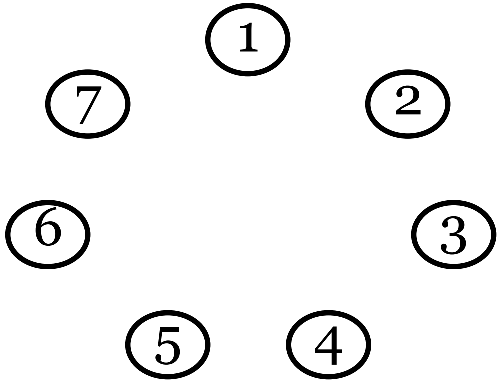
Optimized example

- Show how the data structure changes after each of the following iterations:
  – Initialize(7)
  – Union(1, 2)
  – Union(1, 4)
  – Union(6, 7)
  – Union(3, 5)
  – Union(3, 6)
  – Find(3)

Array:

1	2	3	4	5	6	7

Pointers:



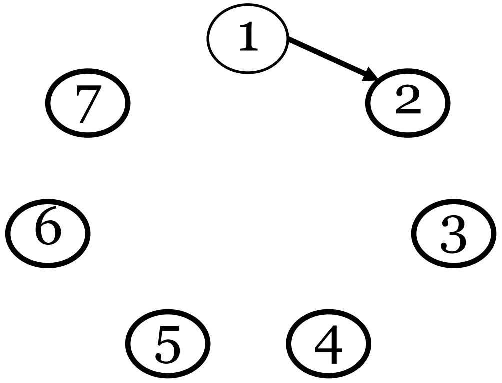
Optimized example

- Show how the data structure changes after each of the following iterations:
  - Initialize(7)
  - Union(1, 2)
  - Union(1, 4)
  - Union(6, 7)
  - Union(3, 5)
  - Union(3, 6)
  - Find(3)

Array:

2	2	3	4	5	6	7

Pointers:



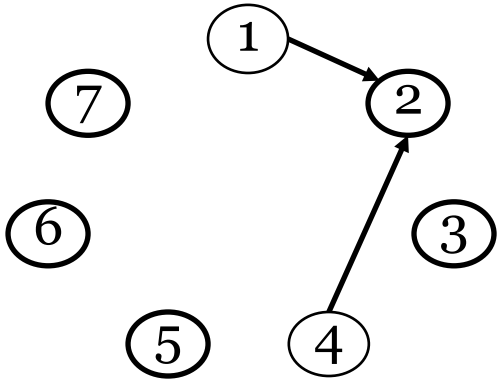
Optimized example

- Show how the data structure changes after each of the following iterations:
  - Initialize(7)
  - Union(1, 2)
  - Union(1, 4)
  - Union(6, 7)
  - Union(3, 5)
  - Union(3, 6)
  - Find(3)

Array:

2	2	3	2	5	6	7

Pointers:



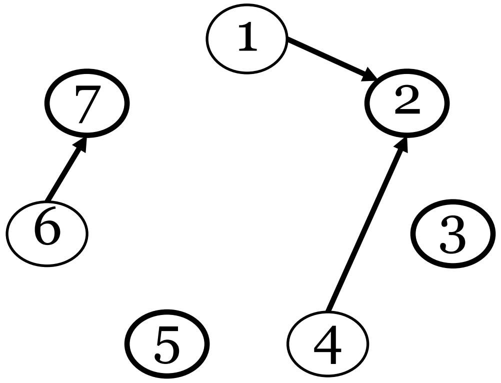
Optimized example

- Show how the data structure changes after each of the following iterations:
  - Initialize(7)
  - Union(1, 2)
  - Union(1, 4)
  - Union(6, 7)
  - Union(3, 5)
  - Union(3, 6)
  - Find(3)

Array:

2	2	3	2	5	7	7

Pointers:



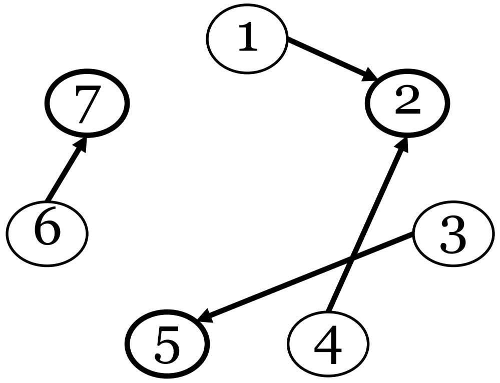
Optimized example

- Show how the data structure changes after each of the following iterations:
  - Initialize(7)
  - Union(1, 2)
  - Union(1, 4)
  - Union(6, 7)
  - Union(3, 5)
  - Union(3, 6)
  - Find(3)

Array:

2	2	5	2	5	7	7

Pointers:



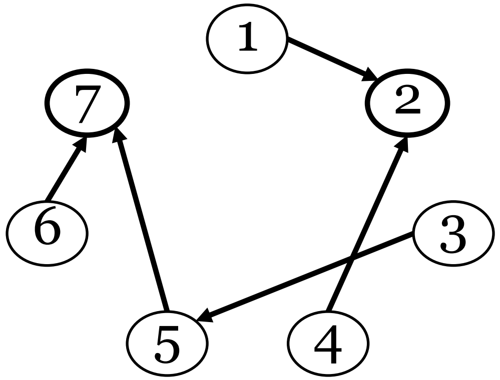
Optimized example

- Show how the data structure changes after each of the following iterations:
  - Initialize(7)
  - Union(1, 2)
  - Union(1, 4)
  - Union(6, 7)
  - Union(3, 5)
  - Union(3, 6)
  - Find(3)

Array:

2	2	5	2	7	7	7

Pointers:



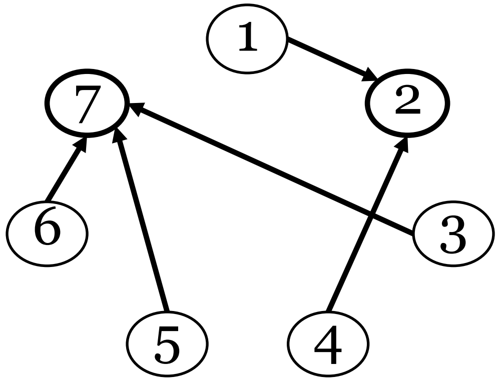
Optimized example

- Show how the data structure changes after each of the following iterations:
  - Initialize(7)
  - Union(1, 2)
  - Union(1, 4)
  - Union(6, 7)
  - Union(3, 5)
  - Union(3, 6)
  - Find(3)

Array:

2	2	7	2	7	7	7

Pointers:



Union-Find operations

- **Find(x)**
  - Recursively point to answer
  - $\Theta(\alpha(n))$, amortized
  - Generally less than 4 for conceivable *n*
- **Union(a, b)**
  - Call Find on both sides first
  - Always point to larger tree
  - $\Theta(\alpha(n))$

- The Ackermann function
  - Incredibly fast-growing function
  - First few values:
    - $3, 7, 61, 2^{2^{2^{65536}}} - 3, \ldots$

```
Algorithm: Find(x)
if unionfind[x] ≠ x then
    id = Find(unionfind[x]);
    unionfind[x] = id;
end
return unionfind[x];
```

```
Algorithm: Union(a, b)
ra = Find(a);
rb = Find(b);
if size[ra] > size[rb] then
    Swap ra and rb;
end
unionfind[ra] = rb;
size[rb] =
  size[ra] + size[rb];
```

Union-Find applications

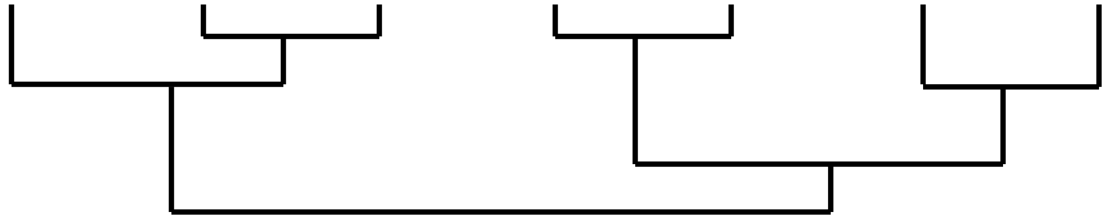
- Major application: algorithm later in semester…
- Single linkage hierarchical clustering
  - All points start as separate, individual clusters (groups)
  - Repeatedly join "closest" clusters together
  - *Single linkage:* distance between clusters = min dist b/w points
- **Input:**
  - *data:* array of *n* points
  - *c:* number of clusters
- **Output:** *c* clusters
- **Pseudocode:**

```
1.  Initialize union-find
2.  Calculate all distances between points
3.  Repeat:
4.      If points with next smallest distance are not in same group:
5.          Union them
6.  Until there are c groups in union-find
```

Hierarchical clustering example

- **Pseudocode**
  - Union next closest points if not unioned
  - Repeat until desired # of clusters
- **Example:** *data* are integers, distance is abs. value, $c = 1$

-4	-2	-1	3	4	7	9



dist = 1
dist = 2
dist = 3
dist = 4

Union-Find exercise

- Perform single-linkage hierarchical clustering on the array below until everything is in one cluster
  - Distance is abs. value
- Draw the Union-Find data structure after each step

-4	-2	-1	3	4	7	9

- Rules
  - If given a choice, merge the cluster with the leftmost element
  - Merge elements left-to-right

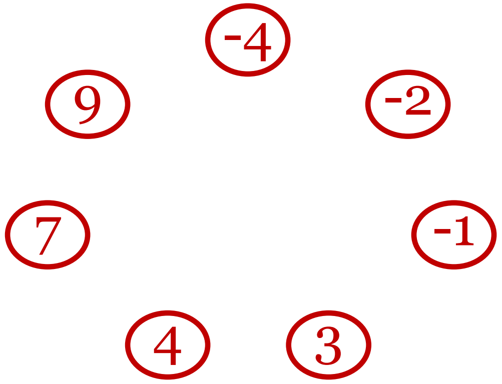
Union-Find exercise

- Perform single-linkage hierarchical clustering on the array below until everything is in one cluster
  - Distance is abs. value
- Draw the Union-Find data structure after each step

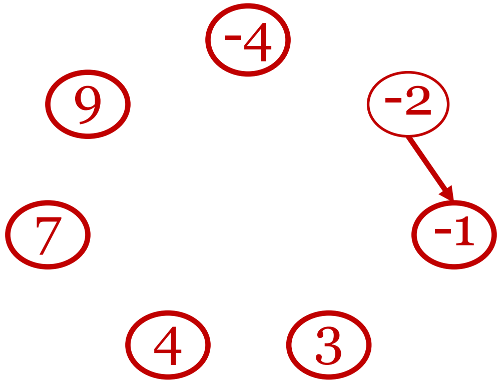
-4	-2	-1	3	4	7	9

- Rules
  - If given a choice, merge the cluster with the leftmost element
  - Merge elements left-to-right

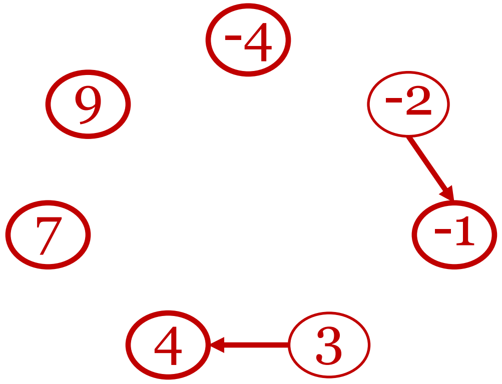
Initialize



Union(2, 3)
or: Union(-2, -1)



Union(4, 5)
or: Union(3, 4)



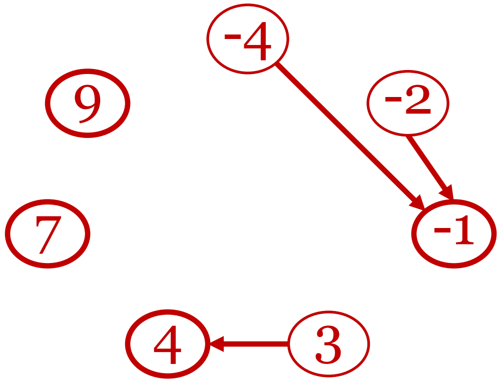
Union-Find exercise

- Perform single-linkage hierarchical clustering on the array below until everything is in one cluster
  - Distance is abs. value
- Draw the Union-Find data structure after each step

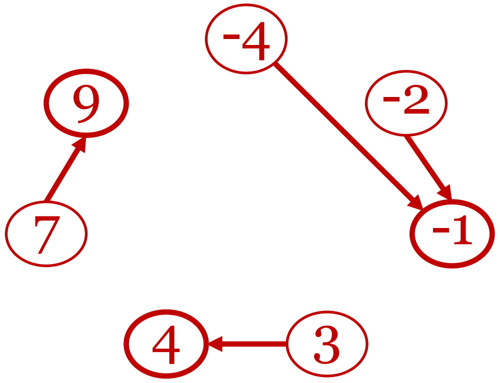
-4	-2	-1	3	4	7	9

- Rules
  - If given a choice, merge the cluster with the leftmost element
  - Merge elements left-to-right

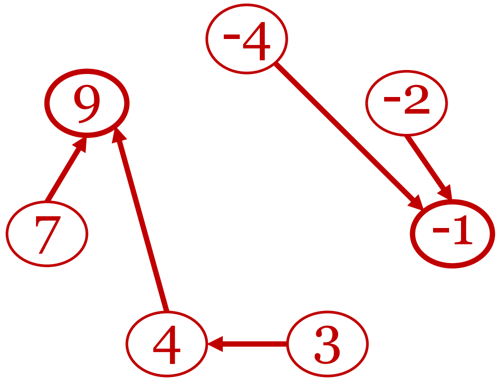
Union(1, 2)
or: Union(-4, -2)



Union(6, 7)
or: Union(7, 9)



Union(5, 6)
or: Union(4, 7)



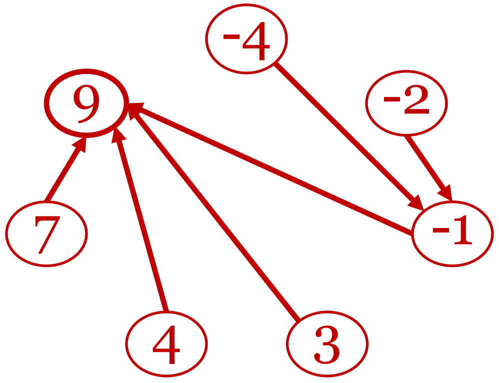
Union-Find exercise

- Perform single-linkage hierarchical clustering on the array below until everything is in one cluster
  - Distance is abs. value
- Draw the Union-Find data structure after each step

-4	-2	-1	3	4	7	9

- Rules
  - If given a choice, merge the cluster with the leftmost element
  - Merge elements left-to-right

Union(3, 4)
or: Union(-1, 3)



Array representation (final):

3	3	7	7	7	7	7

Note that array values represent the *position* (index) of the element, not its value.
For example, element #1 (-4) points to element #3 (-1), which points to element #7 (9).

Coming up

- Union-Find
- Sorting
- Search

- **Recommended reading:** Sections 5.1, 5.3, 5.4 (just "Bottom-Up Heap Construction" to the end), 7.1, and 7.3
  - *Practice problems:* R-5.8, R-5.9, R-5.10, R-5.12, C-5.5, A-5.1, R-7.7, R-7.8, C-7.8, A-7.1