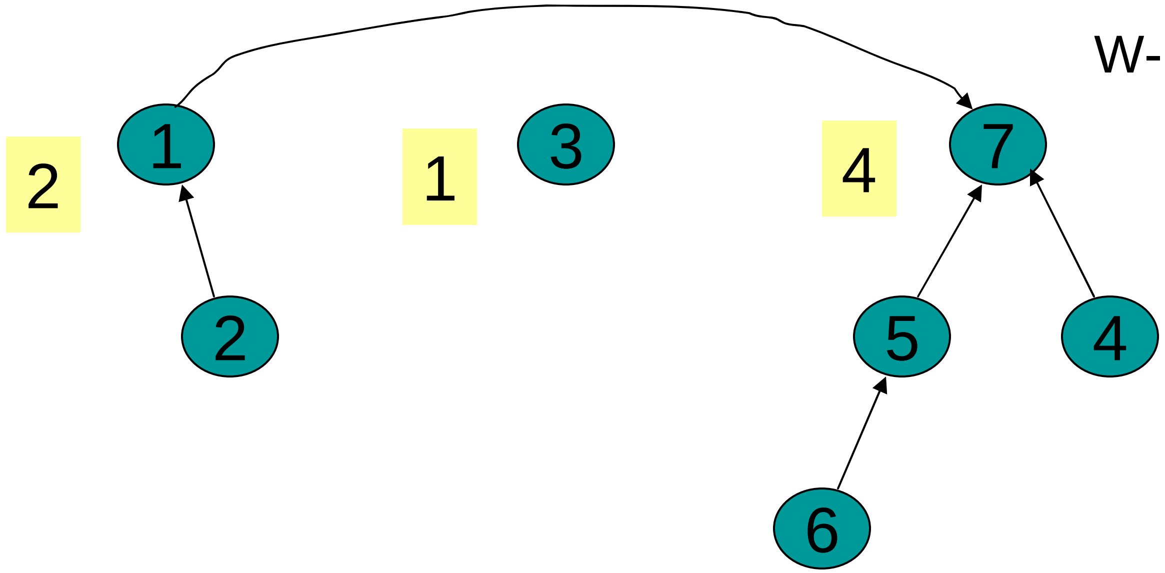
CSE 326: Data Structures
Disjoint Union/Find

James Fogarty

Autumn 2007

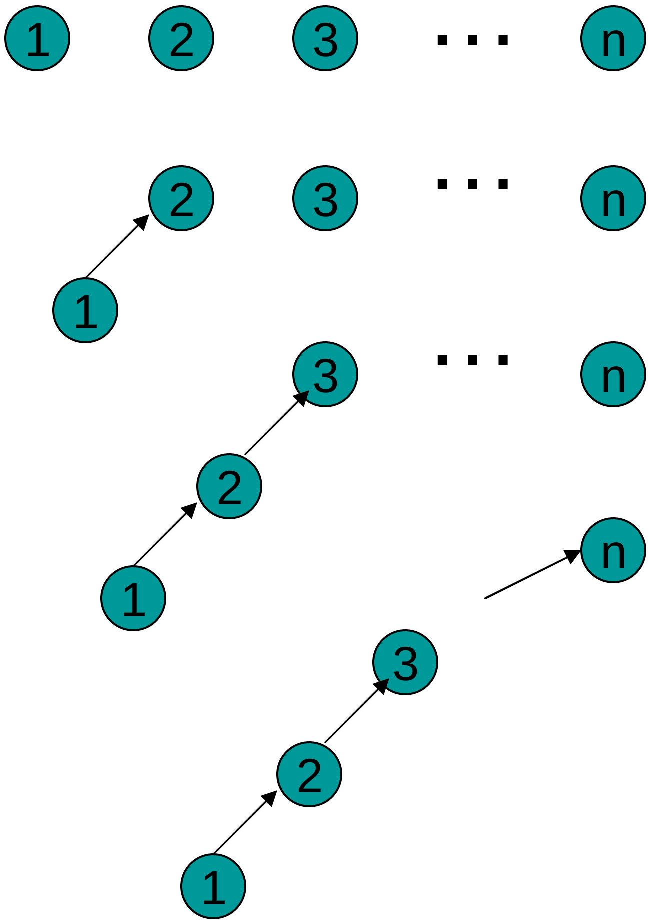
Weighted Union

- Weighted Union
  - – Always point the smaller tree to the root of the larger tree



W-Union(1,7)

A Bad Case



Union(1,2)

Union(2,3)

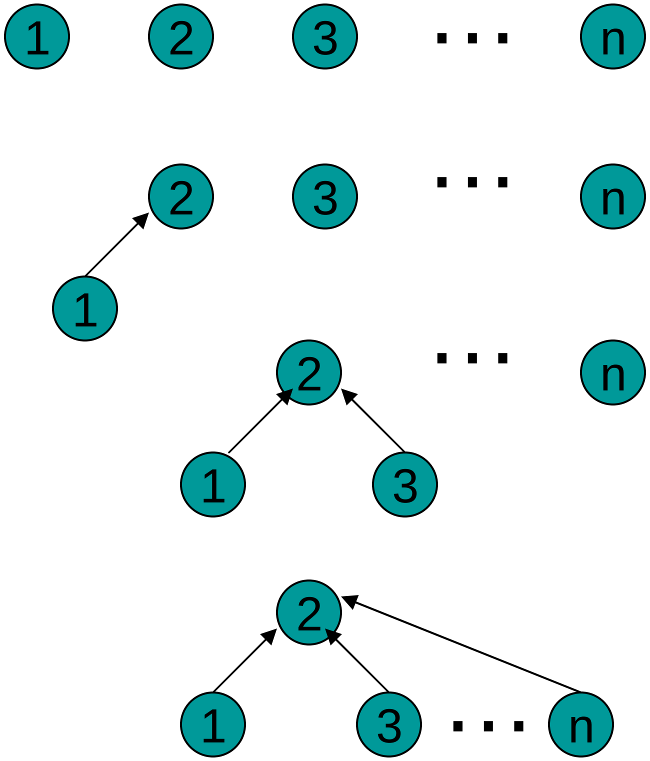
:

:

Union(n-1,n)

Find(1) n steps!!

Example Again



Union(1,2)

Union(2,3)

:

:

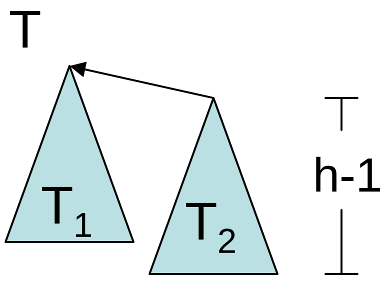
Union(n-1,n)

Find(1) constant time

Analysis of Weighted Union

- With weighted union an up-tree of height h has weight at least 2^h.
- Proof by induction
  - Basis: h = 0. The up-tree has one node, $2^0 = 1$
  - Inductive step: Assume true for all h’ < h.

Minimum weight up-tree of height h formed by weighted unions



$$W(T_1) \geq W(T_2) \geq 2^{h-1}$$

Weighted union

Induction hypothesis

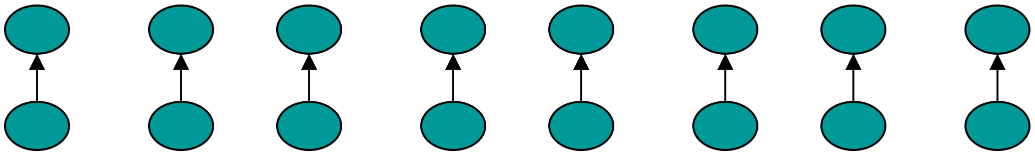
$$W(T) \geq 2^{h-1} + 2^{h-1} = 2^h$$

Analysis of Weighted Union

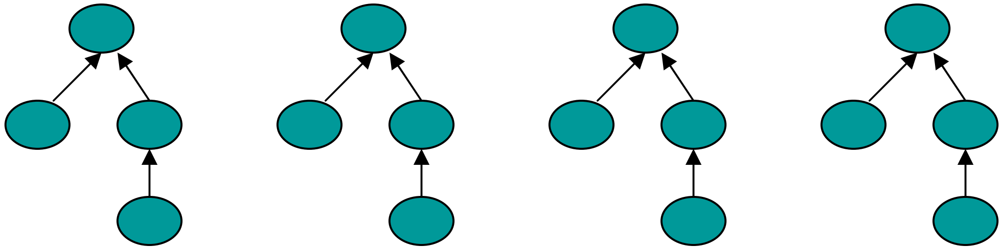
- Let T be an up-tree of weight n formed by weighted union. Let h be its height.
- $n \geq 2^h$
- $\log_2 n \geq h$
- Find(x) in tree T takes O(log n) time.
- Can we do better?

Worst Case for Weighted Union

n/2 Weighted Unions

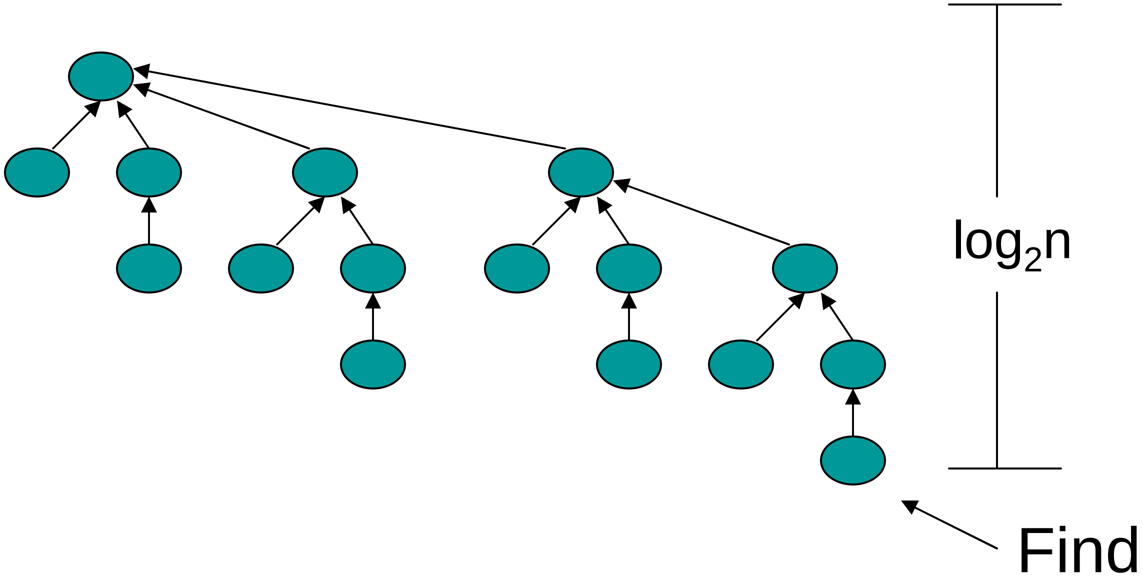


n/4 Weighted Unions



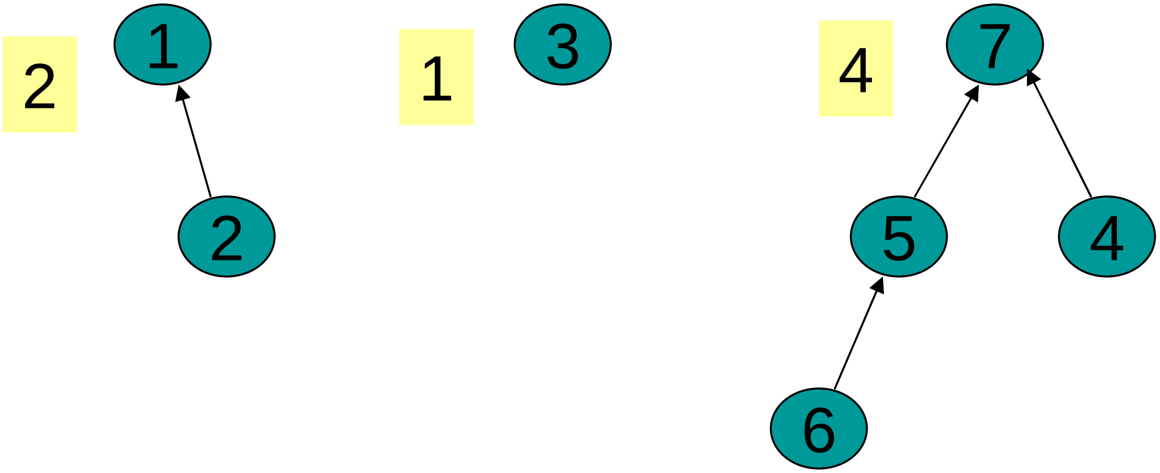
Example of Worst Cast (cont')

After n -1 = n/2 + n/4 + …+ 1 Weighted Unions



If there are $n = 2^k$ nodes then the longest path from leaf to root has length k.

Elegant Array Implementation



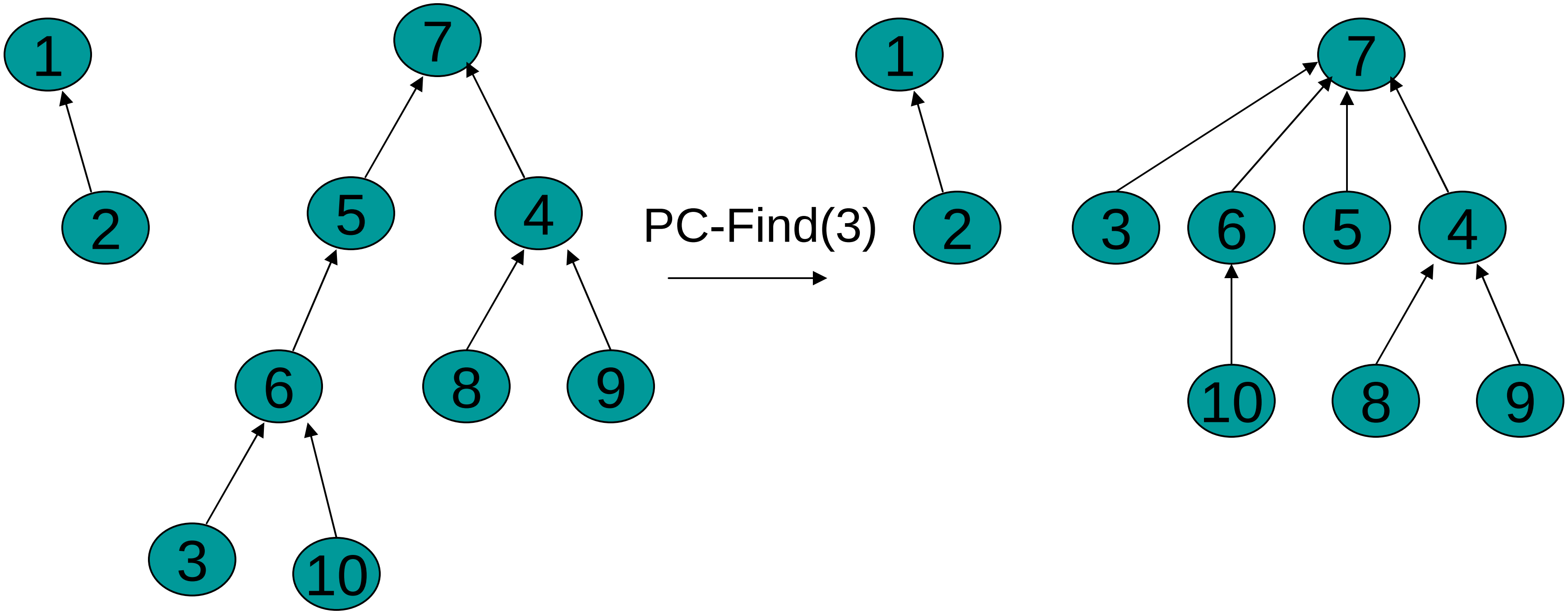
	1	2	3	4	5	6	7
up	0	1	0	7	7	5	0
weight	2		1				4

Weighted Union

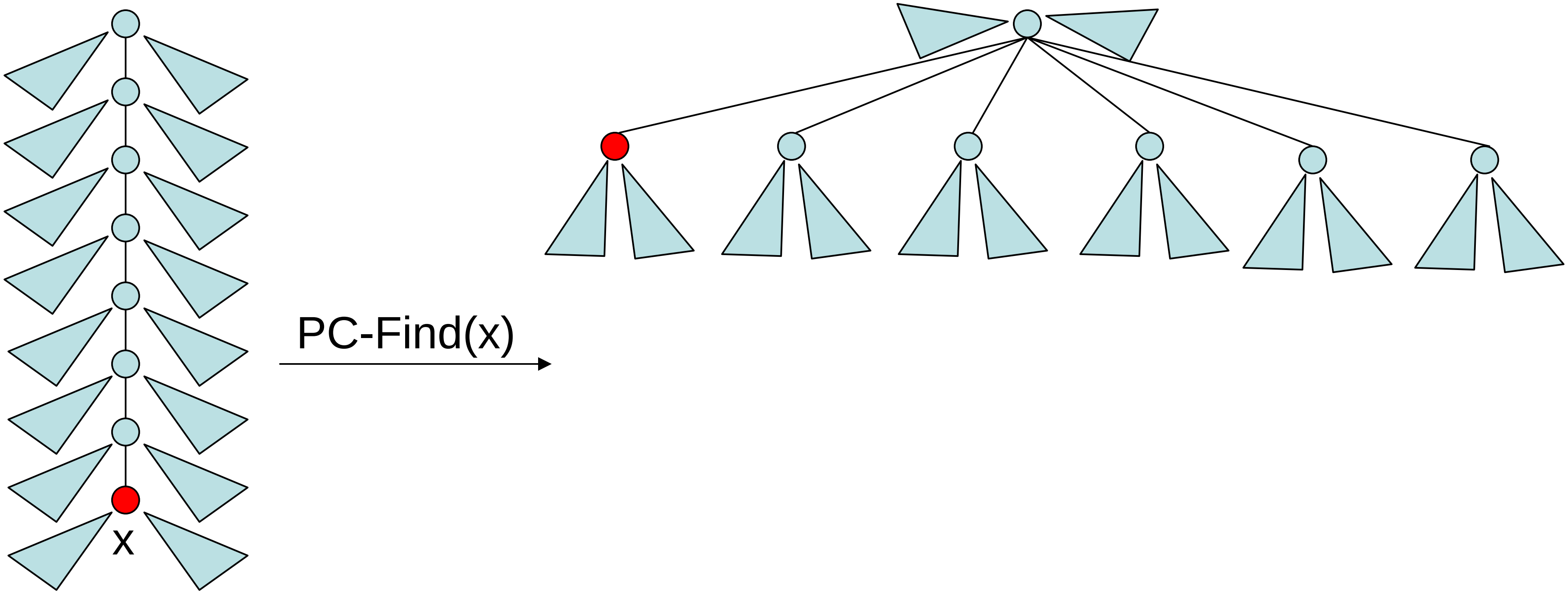
```
W-Union(i,j : index){
//i and j are roots//
  wi := weight[i];
  wj := weight[j];
  if wi < wj then
    up[i] := j;
    weight[j] := wi + wj;
  else
    up[j] :=i;
    weight[i] := wi + wj;
}
```

Path Compression

- On a Find operation point all the nodes on the search path directly to the root.



Self-Adjustment Works

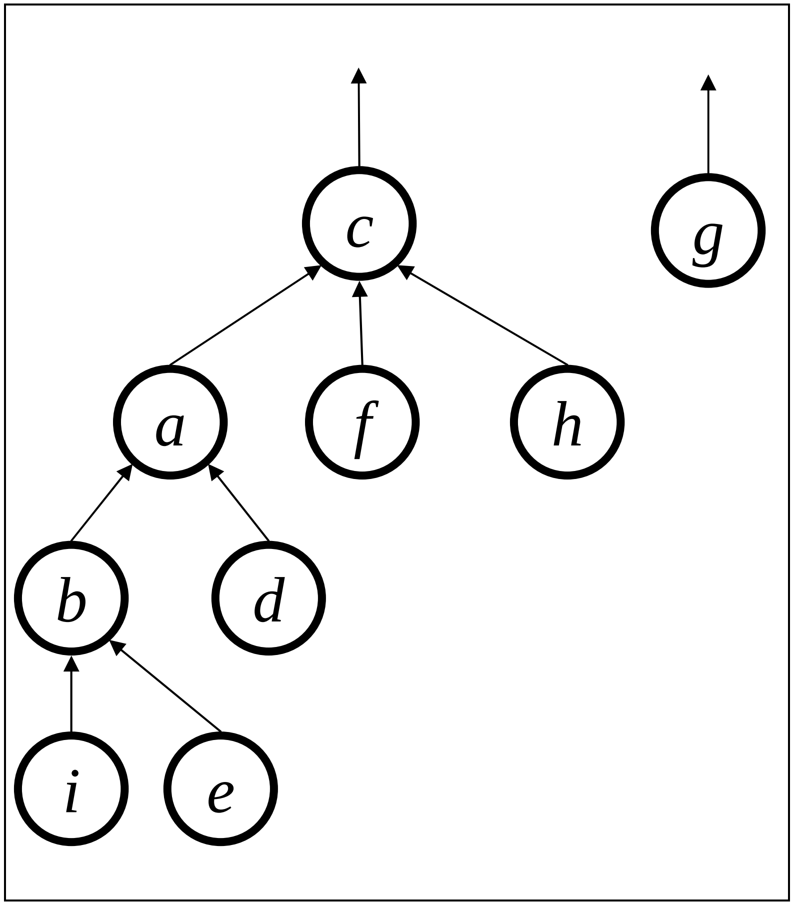


x

PC-Find(x)

Student Activity

Draw the result of Find(e):



Path Compression Find

```
PC-Find(i : index) {
  r := i;
  while up[r] ≠ 0 do //find root//
    r := up[r];
  if i ≠ r then  //compress path//
    k := up[i];
    while k ≠ r do
      up[i] := r;
      i := k;
      k := up[k]
  return(r)
}
```

Interlude: A Really Slow Function

Ackermann's function is a really big function $A(x, y)$ with inverse $\alpha(x, y)$ which is really small

How fast does $\alpha(x, y)$ grow?

$\alpha(x, y) = 4$ for x **far** larger than the number of atoms in the universe (2^{300})

α shows up in:

- Computation Geometry (surface complexity)
- Combinatorics of sequences

A More Comprehensible Slow Function

log* *x* = number of times you need to compute log to bring value down to at most 1

E.g. log* 2 = 1

log* 4 = log* 2^2 = 2

log* 16 = log* 2^{2^2} = 3 (log log log 16 = 1)

log* 65536 = log* $2^{2^{2^2}}$ = 4 (log log log log 65536 = 1)

log* 2^{65536} = …………….. = 5

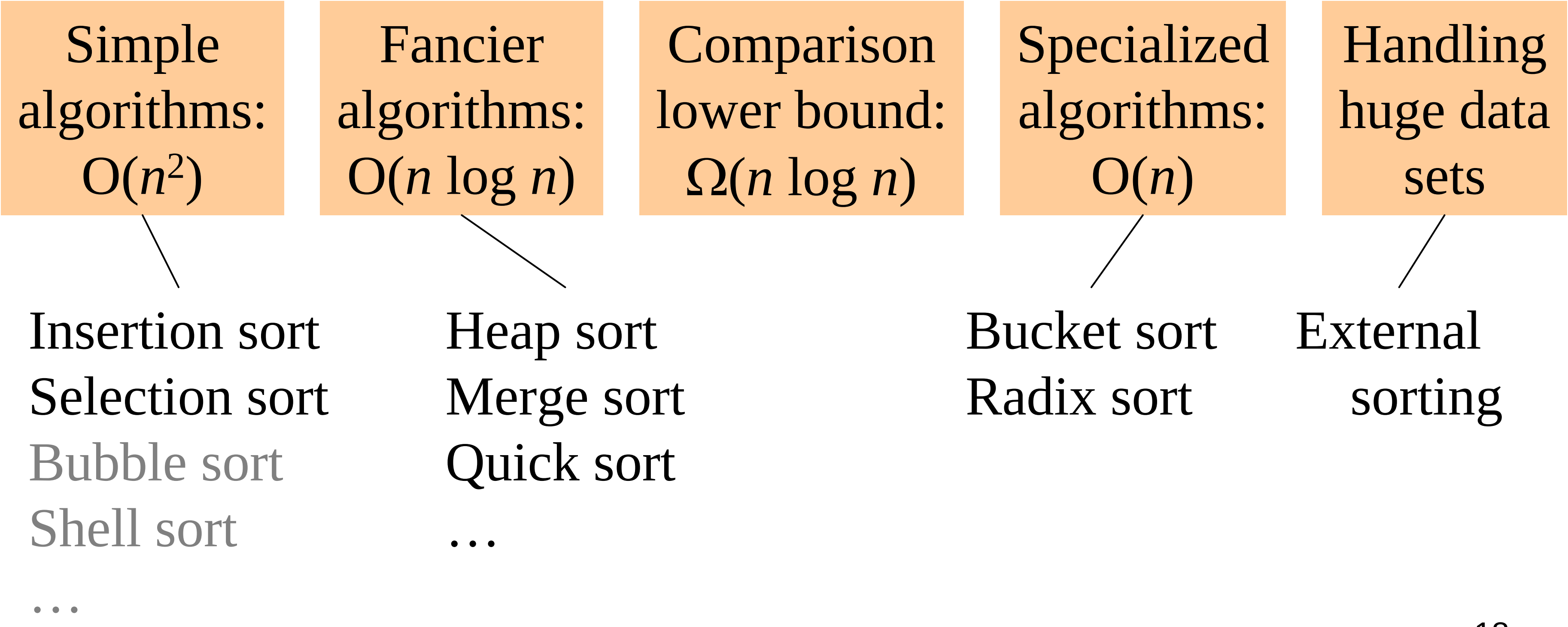
Take this: $\alpha(m,n)$ grows even slower than log* *n* !!

Disjoint Union / Find with Weighted Union and PC

- Worst case time complexity for a W-Union is O(1) and for a PC-Find is O(log n).
- Time complexity for $m \geq n$ operations on n elements is O(m log* n)
  - Log * n < 7 for all reasonable n. Essentially constant time per operation!
- Using “ranked union” gives an even better bound theoretically.

Sorting: *The* **Big** *Picture*

Given *n* comparable elements in an array, sort them in an increasing order.



Simple algorithms: O(n^2)

- Insertion sort
- Selection sort
- Bubble sort
- Shell sort
- …

Fancier algorithms: O($n \log n$)

- Heap sort
- Merge sort
- Quick sort
- …

Comparison lower bound: $\Omega(n \log n)$

Specialized algorithms: O(n)

- Bucket sort
- Radix sort

Handling huge data sets

- External sorting

Insertion Sort: Idea

- At the k^{th} step, put the k^{th} input element in the correct place among the first k elements
- Result: After the k^{th} step, the first k elements are sorted.

Runtime:

worst case :

best case :

average case :

Selection Sort: idea

- Find the smallest element, put it 1st
- Find the next smallest element, put it 2nd
- Find the next smallest, put it 3rd
- And so on …

Selection Sort: Code

```
void SelectionSort (Array a[0..n-1]) {
      for (i=0, i<n; ++i) {
            j = Find index of smallest entry in a[i..n-1]
            Swap(a[i],a[j])
      }

}
```

Runtime:

worst case :

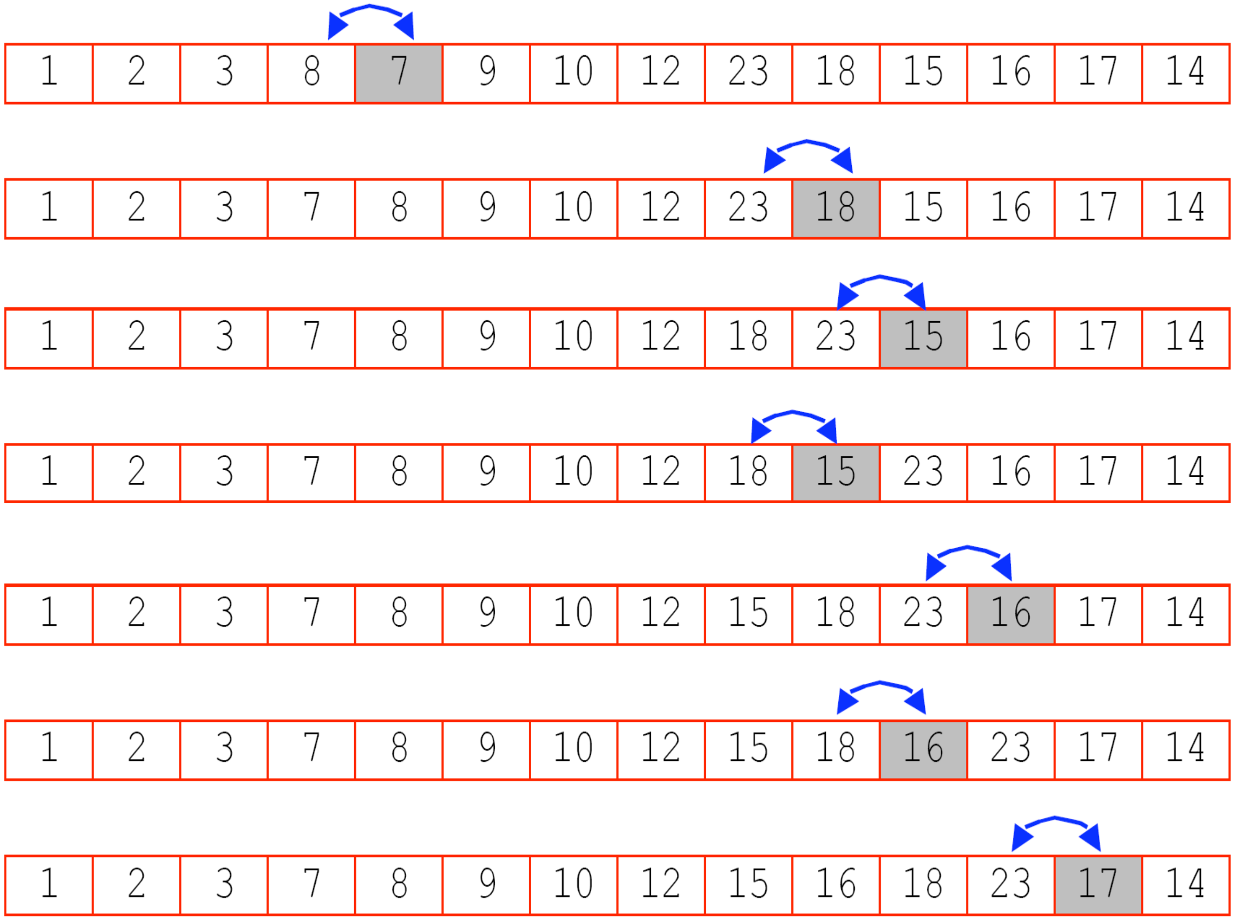
best case :

average case :

Try it out: Selection sort

- 31, 16, 54, 4, 2, 17, 6

Example



1	2	3	8	7	9	10	12	23	18	15	16	17	14

1	2	3	7	8	9	10	12	23	18	15	16	17	14

1	2	3	7	8	9	10	12	18	23	15	16	17	14

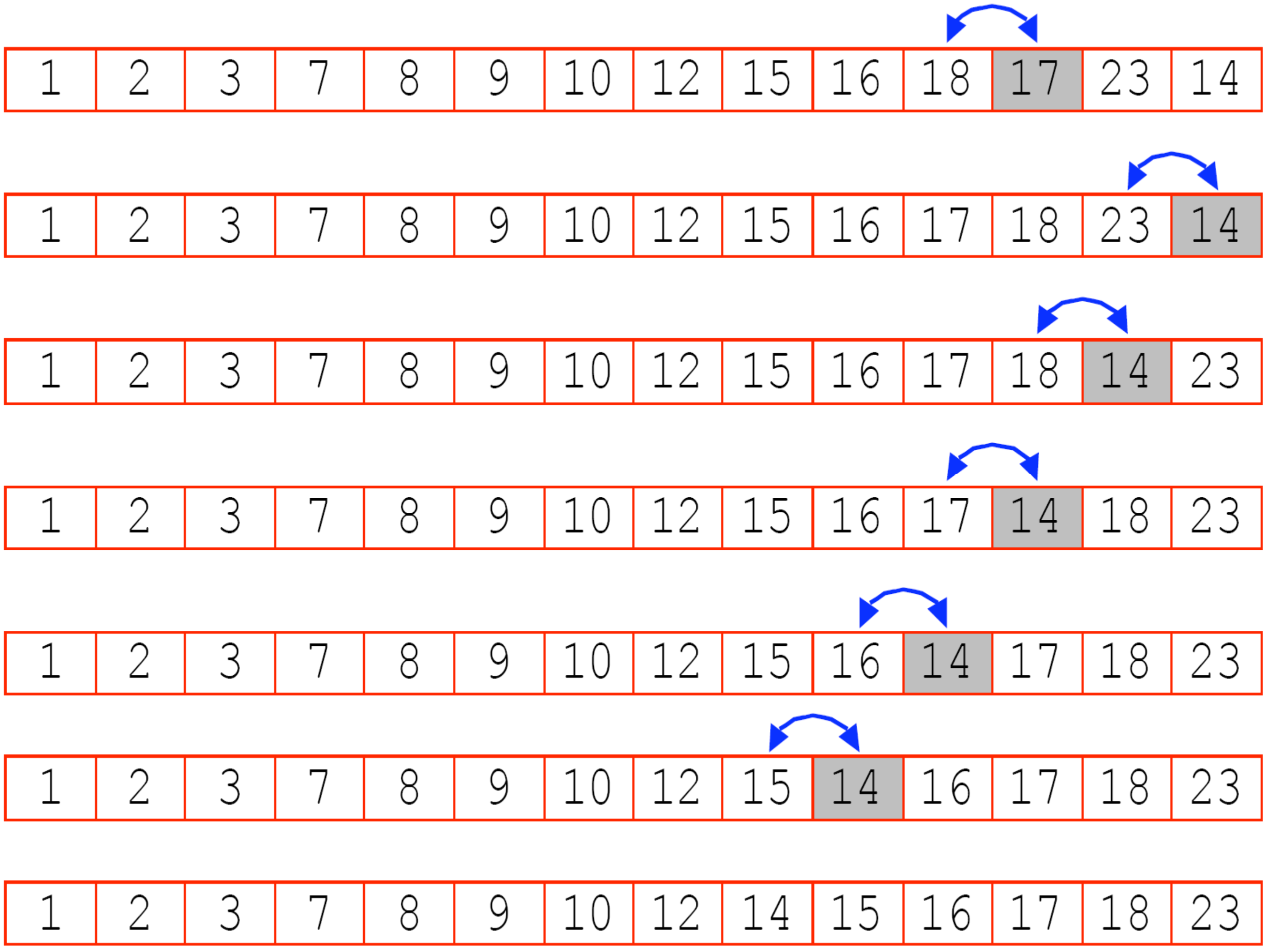
1	2	3	7	8	9	10	12	18	15	23	16	17	14

1	2	3	7	8	9	10	12	15	18	23	16	17	14

1	2	3	7	8	9	10	12	15	18	16	23	17	14

1	2	3	7	8	9	10	12	15	16	18	23	17	14

Example

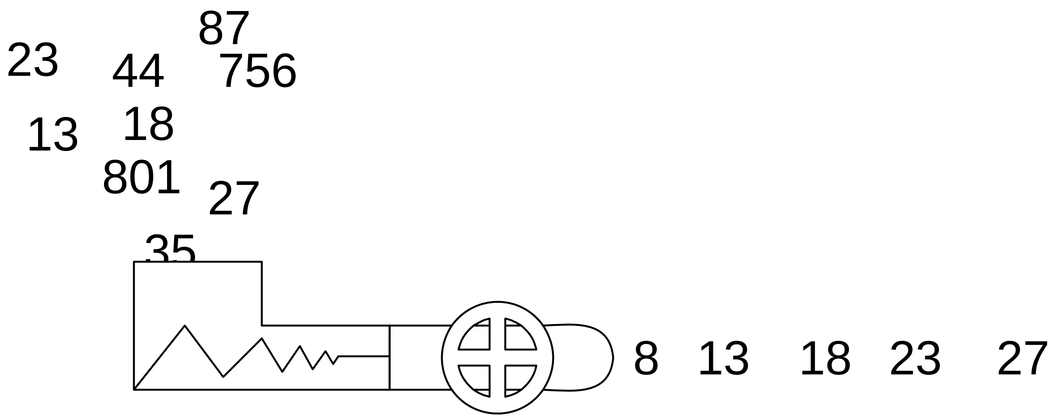


1	2	3	7	8	9	10	12	15	16	18	**17**	23	14
1	2	3	7	8	9	10	12	15	16	17	18	23	**14**
1	2	3	7	8	9	10	12	15	16	17	18	**14**	23
1	2	3	7	8	9	10	12	15	16	17	**14**	18	23
1	2	3	7	8	9	10	12	15	16	**14**	17	18	23
1	2	3	7	8	9	10	12	15	**14**	16	17	18	23
1	2	3	7	8	9	10	12	14	15	16	17	18	23

Try it out: Insertion sort

- 31, 16, 54, 4, 2, 17, 6

HeapSort:
Using Priority Queue ADT (heap)



Shove all elements into a priority queue,
take them out smallest to largest.

Runtime:

Try it out: Heap sort

- 31, 16, 54, 4, 2, 17, 6