Week 5

Wednesday, February 19, 2020 11:06 PM

Linked Lists

Doubly Linked

→ next & previous ptrs

→ keep in mind extra links to change

→ prev & next are opposites:
x.prev.next = x.next.prev = x

* must make sure correctly linked in well-formed

Adding

→ at beginning: make sure to point old head's prev to new

→ at end: make sure to point new tail's prev to old

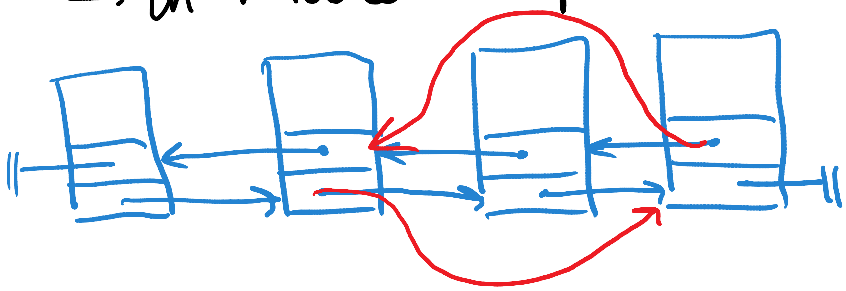
→ in middle: make sure updating in correct order!

```
Node p = new Node(v);
*add after current
p.next = current.next;
current.next = p;
p.prev = current;
* current.next.prev = p;
  current.next is p.
```

* current.next.prev = p
current.next is p.
→ create a cycle

Removal

→ at beginning: make new head's prev null
→ at end: make new tails next null
→ in middle: keep track of pts



n.next.prev = n.prev.
n.prev.next = n.next

Dummy Nodes

→ extra node typically at front of list w/ null data
→ precursor never null
→ simpler - fewer special cases

Cyclic Lists

→ head & tail is connected
→ no null pointers
→ precursor never null

→ no null pointers
  → precursor never null
    → fewer special cases, simpler code
→ adding first or removing last are special cases if there is no dummy node
  → no special cases w/ a dummy

Endogenous

→ links are part of the data
→ no nodes
→ in regular lists, a node acts as a box that data is put into
→ endogenous lists have no nodes & remain unboxed
→ objects point to eachother.
→ does not require additional objects — space efficient
→ simplifies code
→ an object cannot be in 2

→ an object cannot be in 2 lists simultaneously
  → when removing, must set next & prev to null
  → when adding, next & prev must be null prior
    → else throw exception

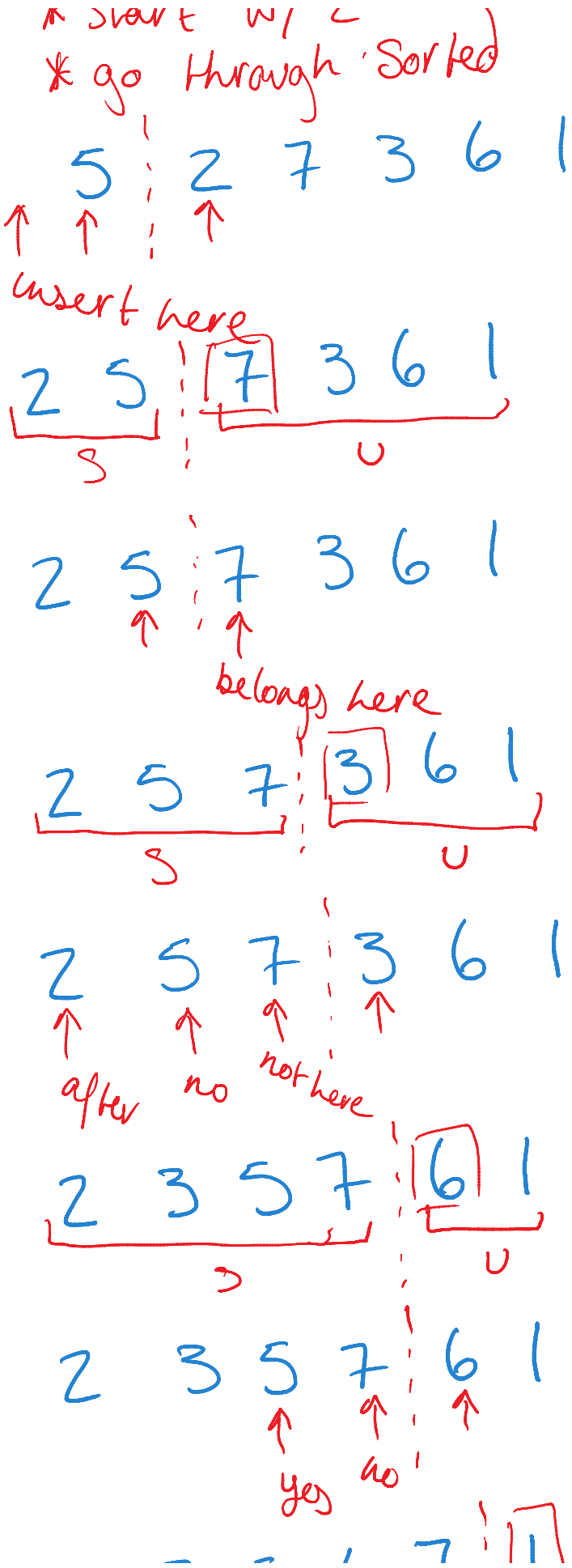
Sorting

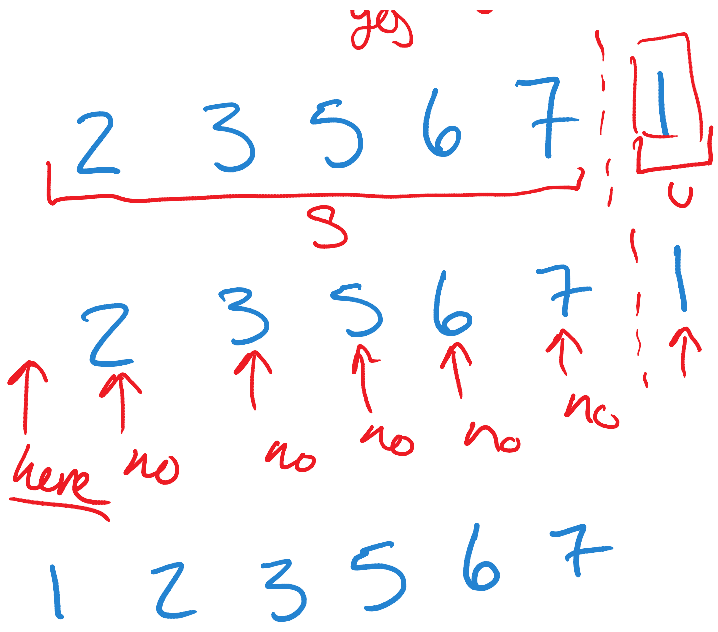
Insertion Sort

→ "sorted" section & "unsorted"
→ take each element from "unsorted" section
→ traverse "sorted" section backwards & find the spot the current element should be inserted into

ex: 5 | 2 7 3 6 1
S | U

* start w/ 2
* go through sorted


insert w/ c
* go through sorted
5 2 7 3 6 1
insert here
2 5 7 3 6 1
S U
2 5 7 3 6 1
belongs here
2 5 7 3 6 1
S U
2 5 7 3 6 1
after no not here
2 3 5 7 6 1
S U
2 3 5 7 6 1
yes no



*runtime? $O(n^2)$

→ for each element, look @ rest

* if already sorted $O(n)$

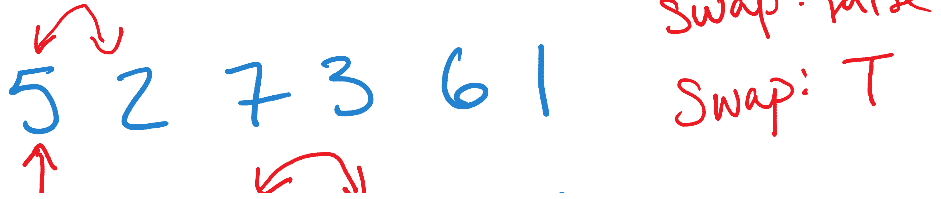
Bubble Sort

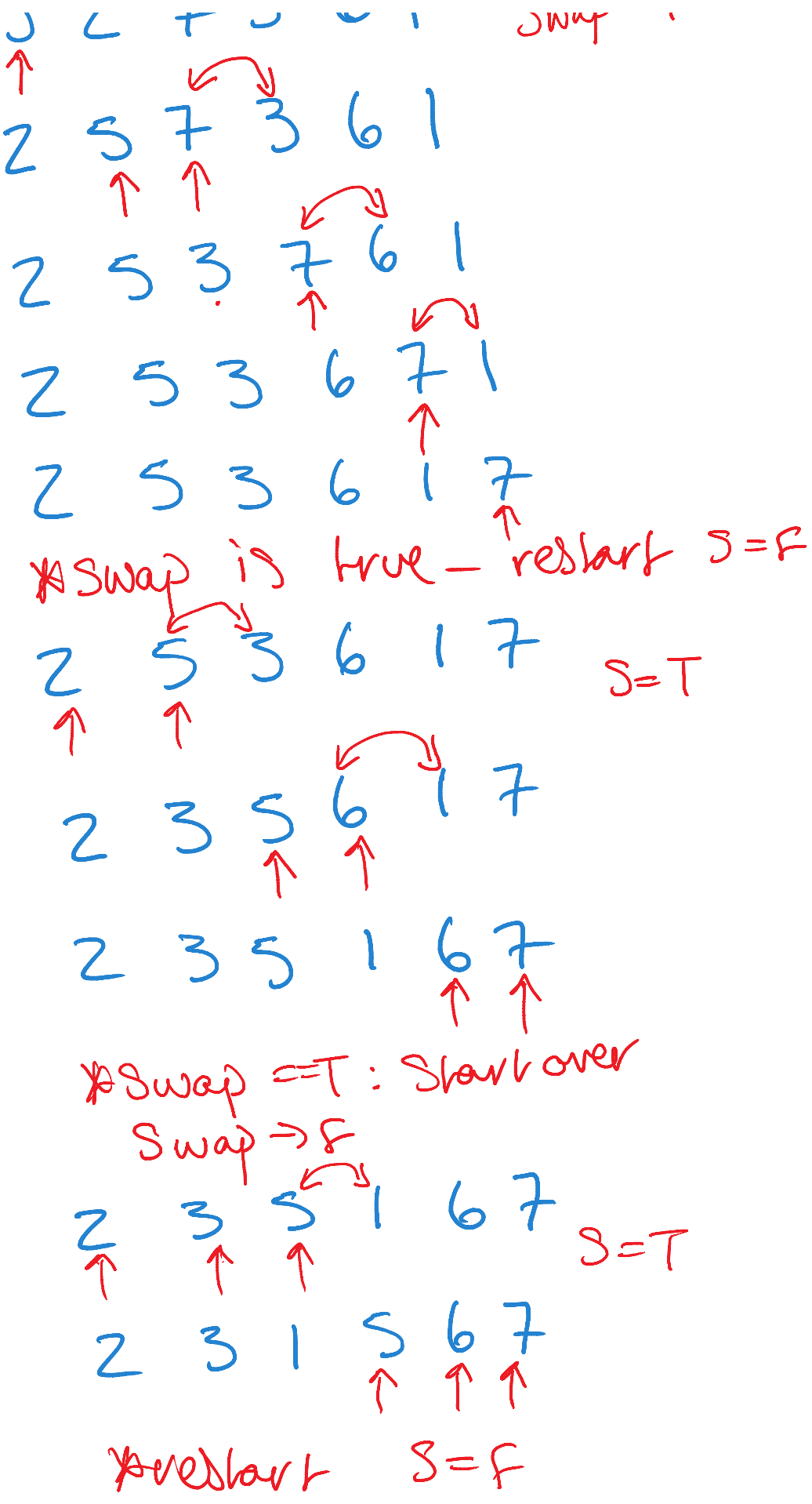
→ very simple & inefficient

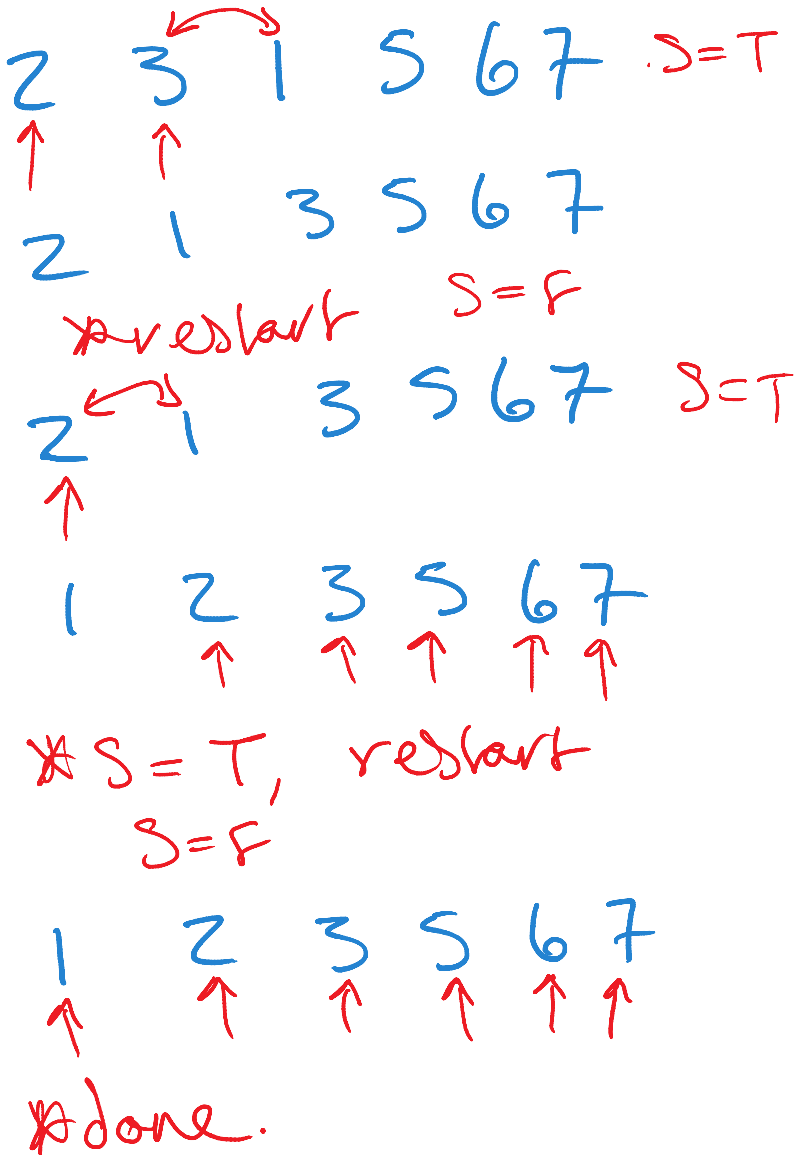
→ start at beginning

→ if 2 next to each other are out of order, swap

→ keep swapping until no more




2 5 7 3 6 1
2 5 3 7 6 1
2 5 3 6 7 1
2 5 3 6 1 7
*Swap is true – restart S=F
2 5 3 6 1 7
S=T
2 3 5 6 1 7
2 3 5 1 6 7
*Swap ==T : Start over
Swap -> F
2 3 5 1 6 7
S=T
2 3 1 5 6 7
*restart S=F



*runtime: $O(n^2)$

best: $O(n)$

Comparator

⇒ generic interface to compare objects

→ create classes that implement

→ create classes that implement Comparator
→ Single method: compare(o1, o2)
  → returns an int
    <0 if o1 < o2
    =0 if o1 = o2
    >0 if o1 > o2

Anonymous Class

→ can create an anonymous class — one that only exists in a specific context
→ instantiate when used
→ has no name — cannot be used elsewhere
→ common to make some objects, particularly comparators, as anonymous classes

Ex:
Comparator<String> c = new

```
Comparator<String> c = new
  Comparator<String>(){
      public int compare(String s1, String s2){
        //etc
      }
};
```

"Pass the Buck" Recursion

→ if it isn't your problem, pass it on to someone else

* don't update arguments

→ pass entire problem to someone else

→ if you can solve the problem, solve it.

* make sure someone can solve it & you don't pass it back to the person who originally had the problem.