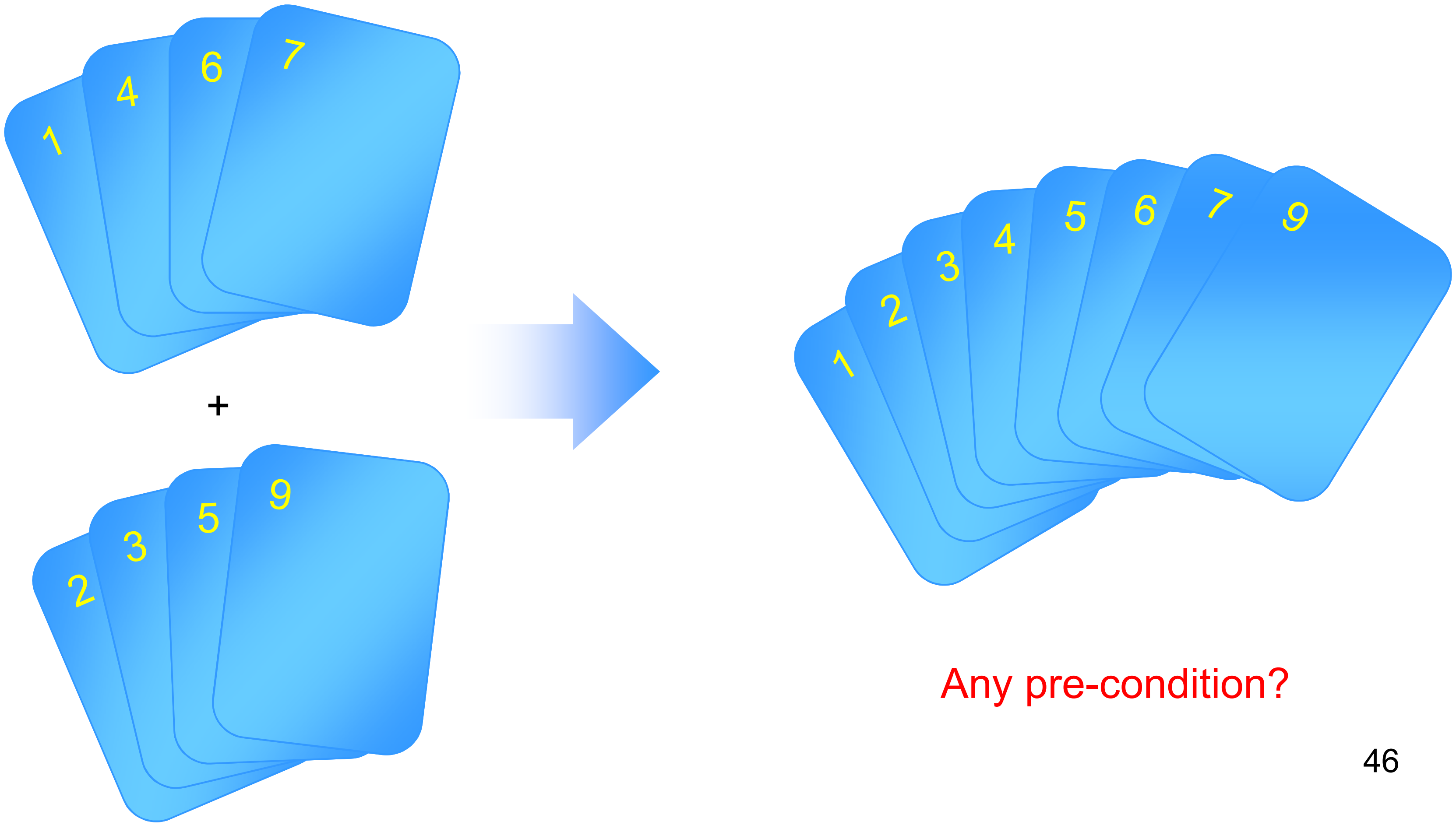
Merge Sort

Time Complexity: $O(n\log n)$

Space Complexity: $O(n)$

Daily Life Example

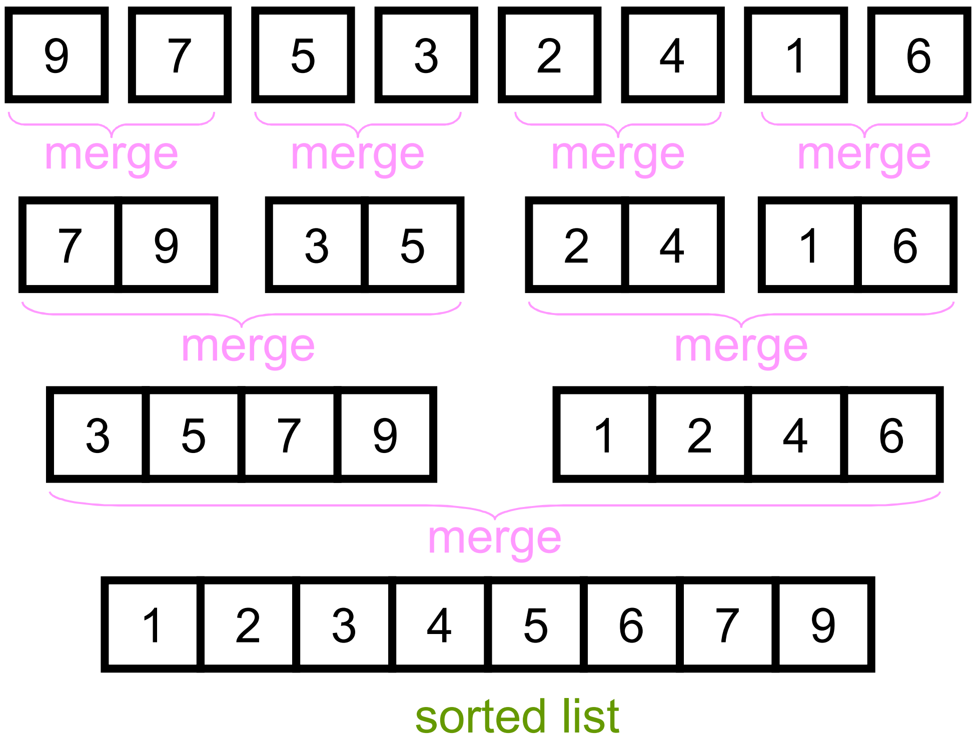
- The idea of merging



Any pre-condition?

The Algorithm

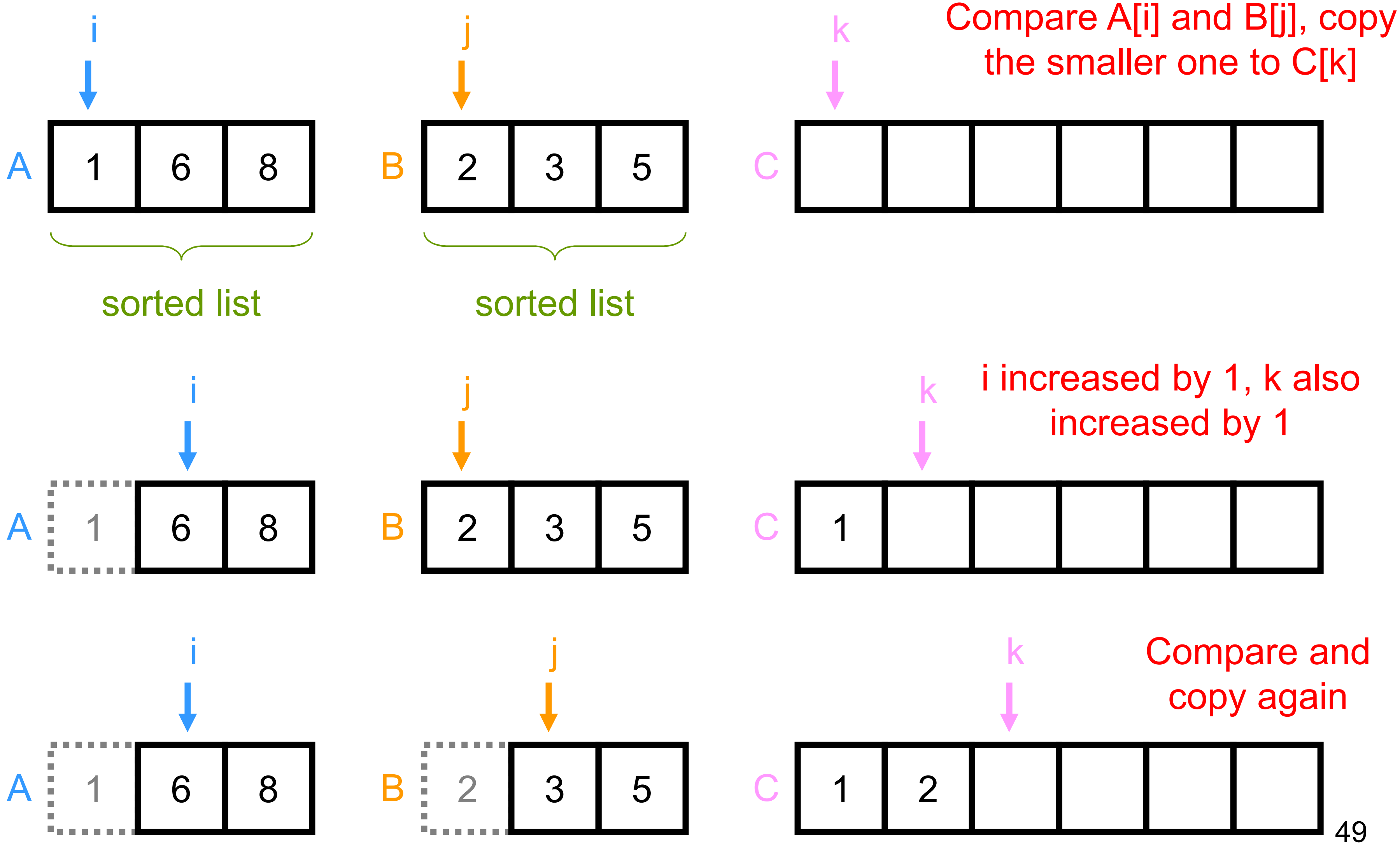
- Initially the input list is divided into *N* sublists of size 1
- Adjacent pairs of lists are merged to form larger sorted sublists
- The merging process is repeated until there is only one list



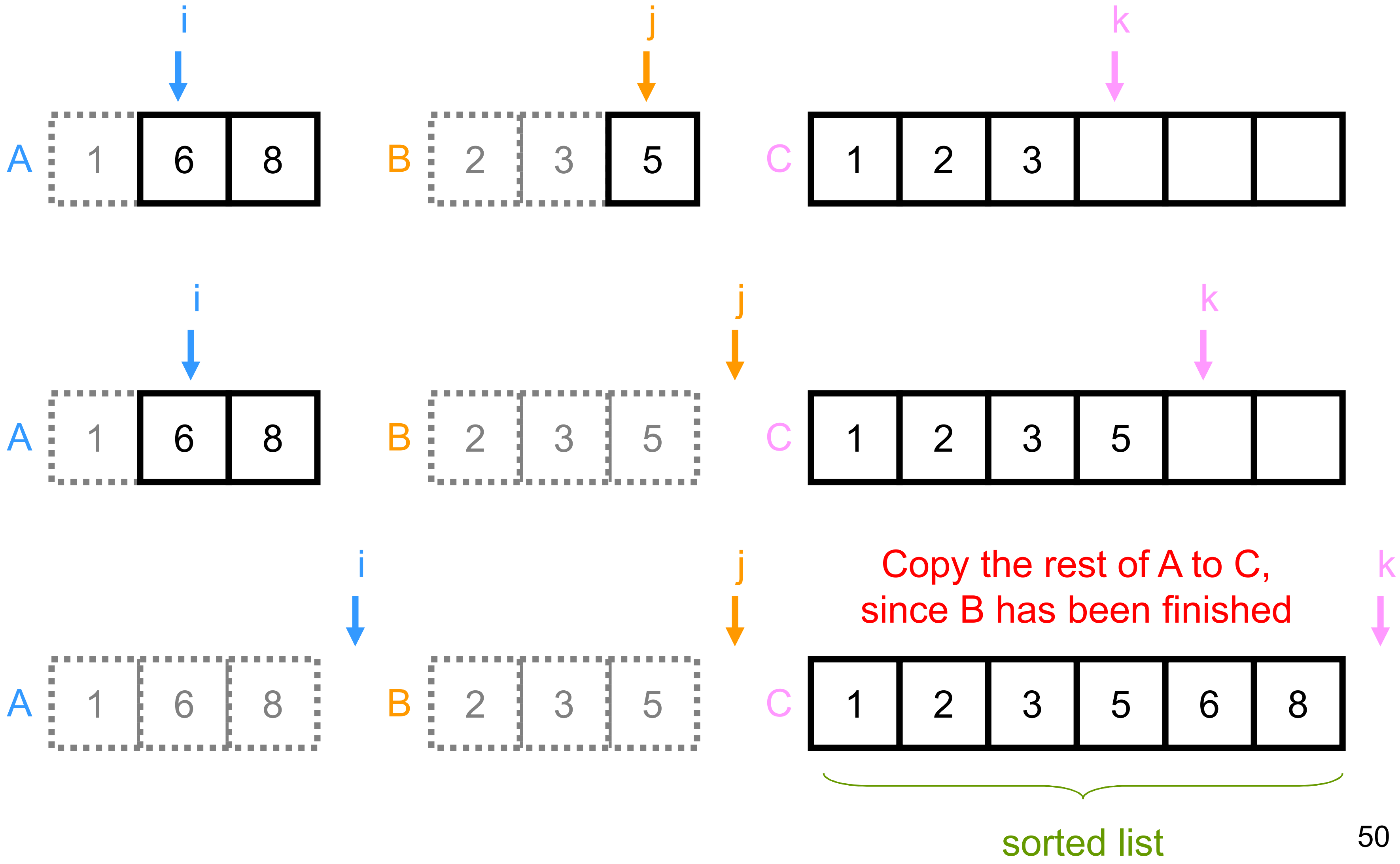
Merging

- To merge 2 **sorted** lists
- It takes 2 input arrays *A*[] & *B*[], 1 output array *C*[] and 3 counters (i, j, k) for the arrays respectively
- The smaller of *A*[i] and *B*[j] is copied to *C*[k], then the counters are advanced
- If either *A*[] or *B*[] finishes first, the reminder of the other array is copied to *C*[]

Merge Sort Example



Merge Sort Example



Merge Adjacent Lists

```
void merge(int data[], int first, int mid, int last) {
    int temp[SIZE], i = first, j = mid + 1, k = 0;
    while (i <= mid && j <= last) {
        if (data[i] <= data[j])
            temp[k++] = data[i++];
        else
            temp[k++] = data[j++];
    }
    while (i <= mid) temp[k++] = data[i++];
    while (j <= last) temp[k++] = data[j++];
    i = 0;
    while (i < k) data[first+i] = temp[i++];
}
```

Compare A[i] and B[j], copy the smaller one to temp[k]

A is data[first…mid]

B is data[mid+1…last]

C is temp[…]

The remaining A or B will be copied into temp

The sorted temp. array is copied back to data

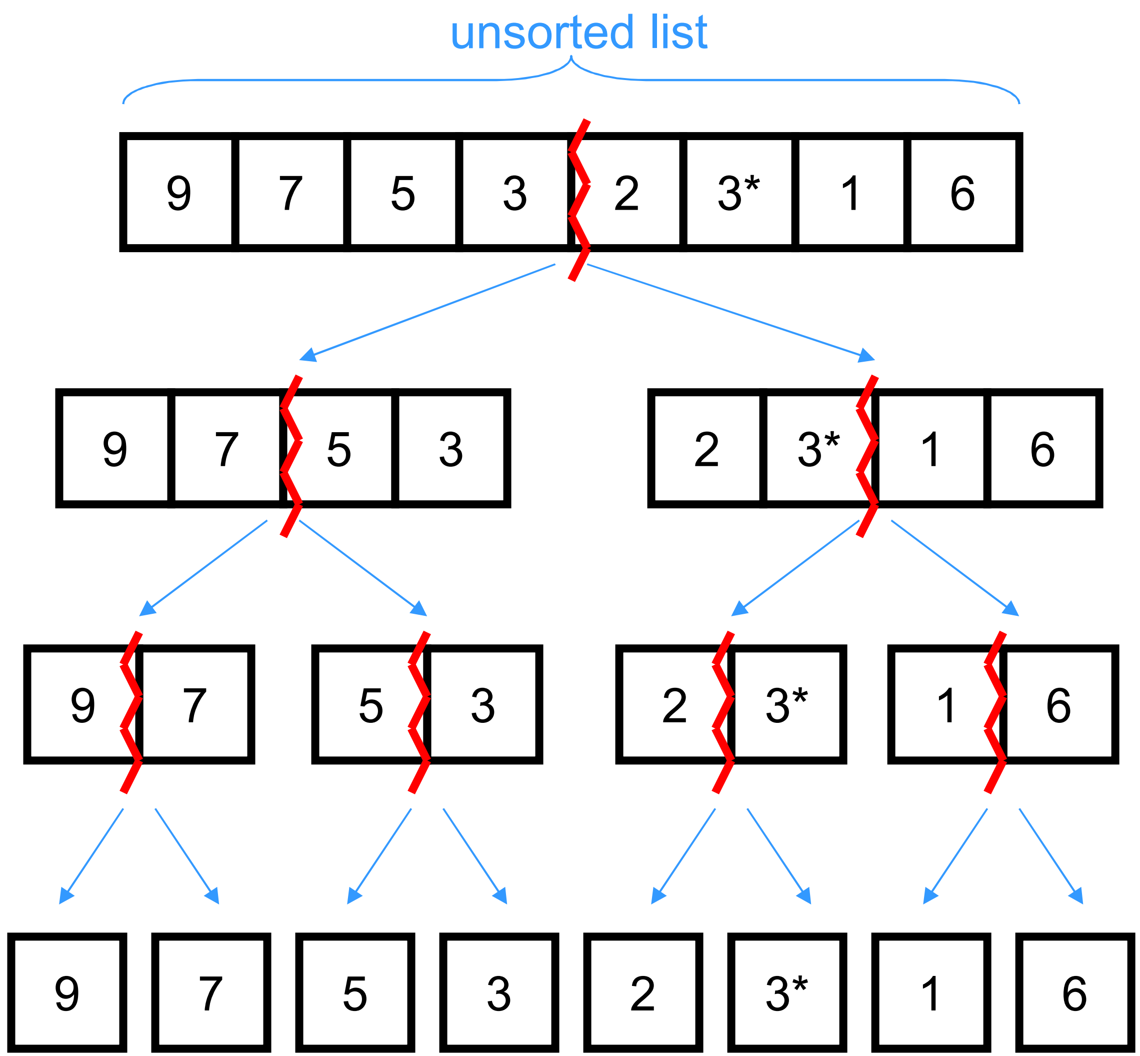
Divide-and-Conquer

- This algorithm is a classic divide-and-conquer strategy
- Very powerful use of recursion
- The problem is divided into smaller problems and solved independently and recursively
- The conquering phase consists of merging together the sorted lists

Dividing Phase

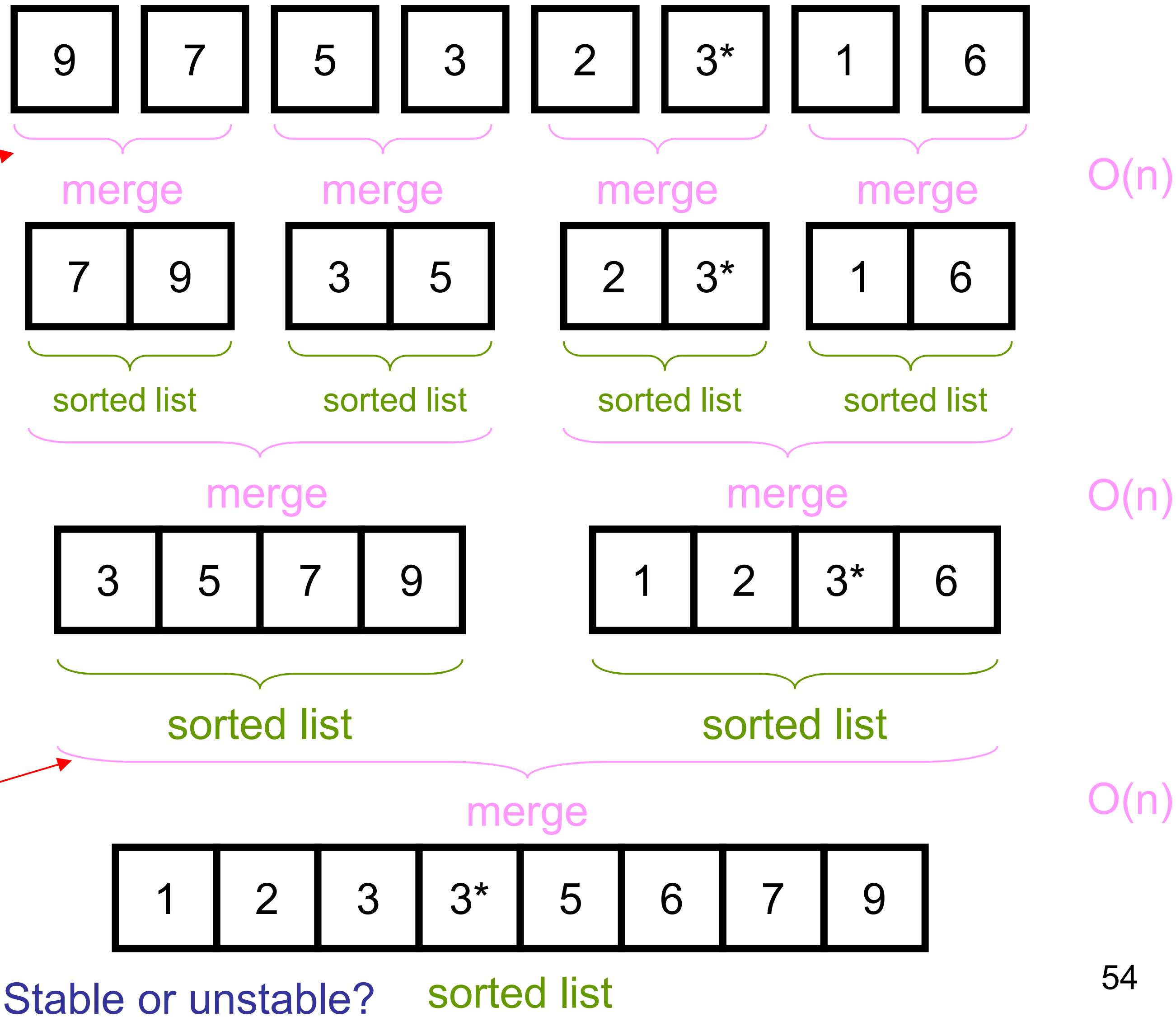
Divide the list into halves

Divide again and again until only one single element left in the list



Conquering Phase

Merge the small sorted lists into larger sorted lists



Merge again and again until one single list left

Stable or unstable?

Merge Sort (Using Recursion)

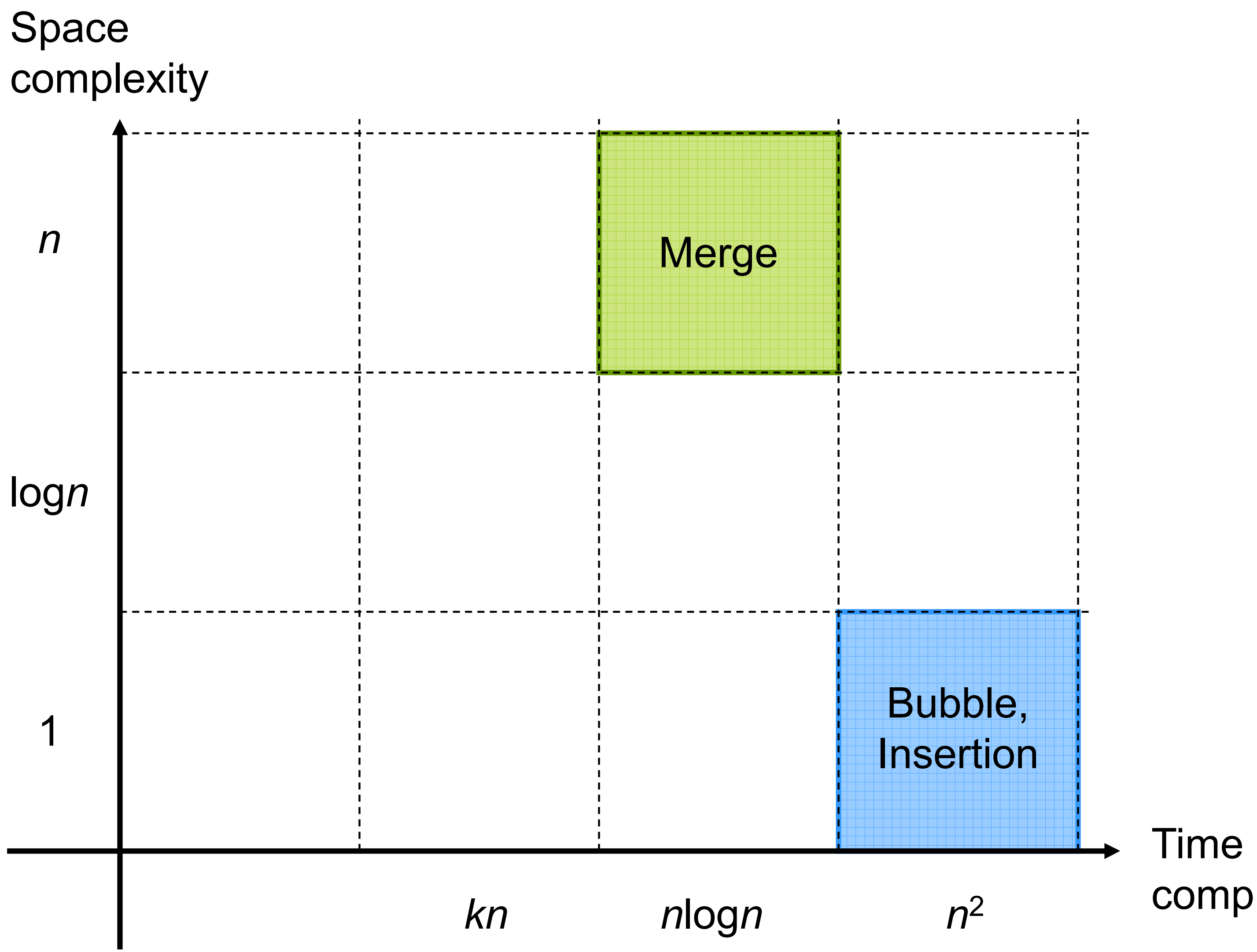
```
void mergesort(int data[], int first, int last) {
    int mid = (first + last ) / 2;
    if (first >= last) return;          //base case: size = 1
    mergesort(data, first, mid);        //recursion: divide the list into halves
    mergesort(data, mid+1, last);//recursion: divide the list into halves
    merge(data, first, mid, last);  //start merging the list: conquer
}
int main(…) {
    int data[] = {8, 5, 9, 6, 3};
    mergesort(data, 0, 4);
    return 0;
}
```

Complexity Analysis

- Merge sort goes through the same steps - independent of the data
  - Best case = Worst case = Average case
- For each runs, it requires O(n) time to finish
- There are $\log_2 n$ runs in total
- The time complexity is O($n\log n$)
- Faster than **bubble sort** and **insertion sort**!

- The trade-off is it needs extra memory to hold the temporary sorted result
- Space complexity = O(n)
- Improvement to the merge algorithm:
  - Instead of merging each set of lists from data[] to temp[] and then copy temp[] back to data[], alternate merge passes can be performed from data[] to temp[] and from temp[] to data[].

Summary



Space complexity
n
Merge
$\log n$
1
Bubble, Insertion
Time complexity
kn
$n\log n$
n^2

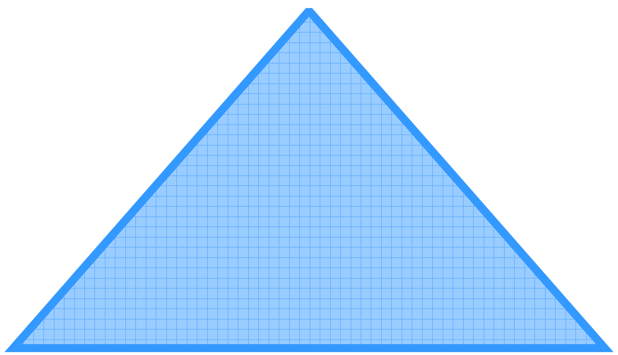
Heapsort

Time Complexity: O($n$log$n$)

Space Complexity: O(1)

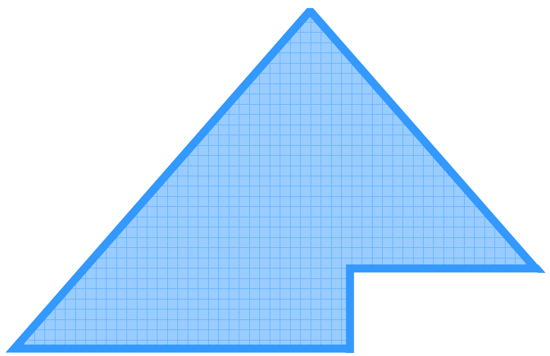
Heap Revision

- Max. heap tree is a binary tree with 2 properties
  - **Property 1**: The tree is complete
  - **Property 2**: The tree is descending



Full binary tree

OR



Almost complete binary tree

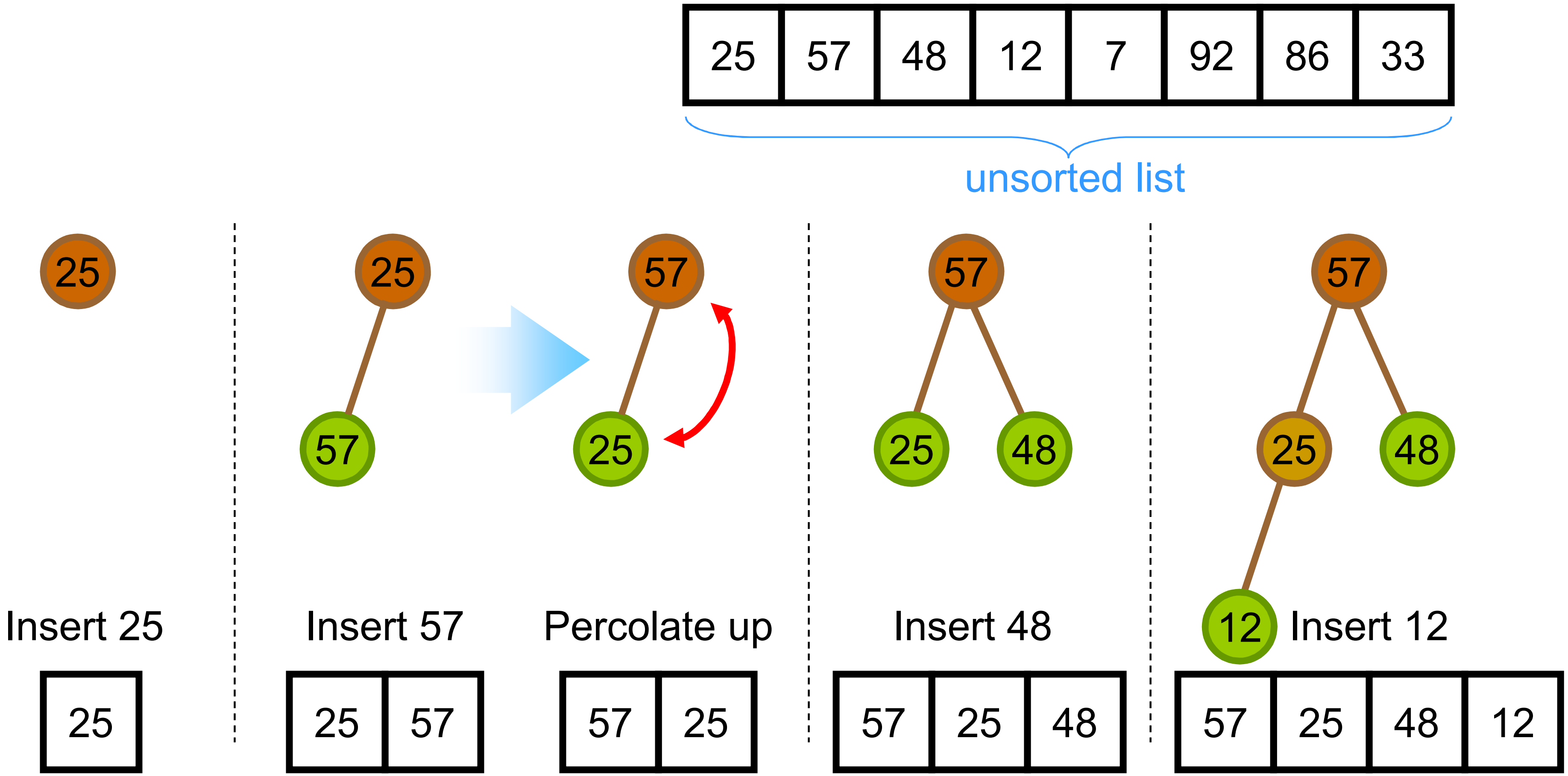
The Heapsort Algorithm

- Phase 1) Build Heap
  - Organize the input array as a max heap

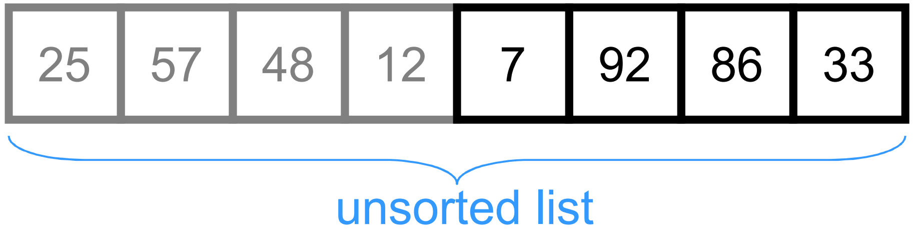
- Phase 2) Swap Node
  - Iteratively remove the root of the heap (the largest value in the subgroup of numbers to be sorted) and re-adjust the remaining numbers to form a max heap.

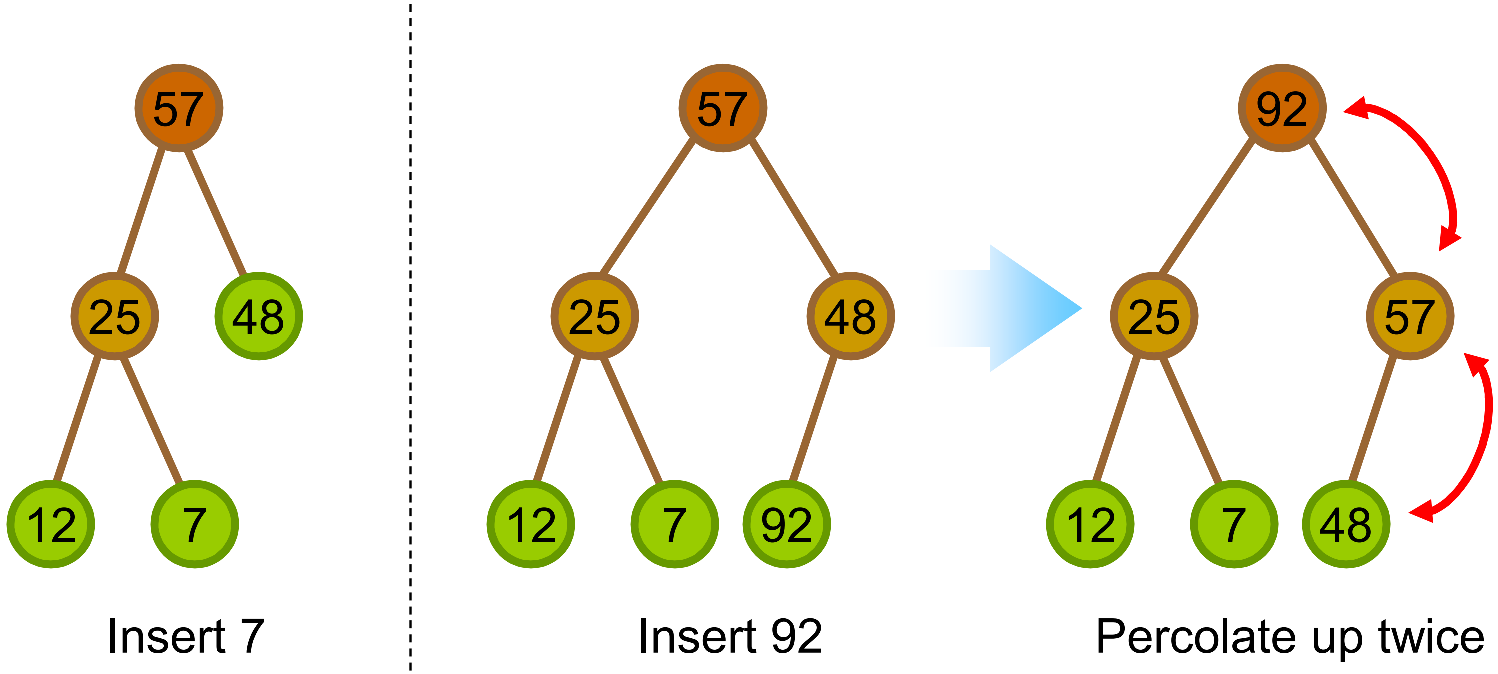
Heapsort Example: Phase 1

- Phase 1) Build the max. heap



Heapsort Example: Phase 1

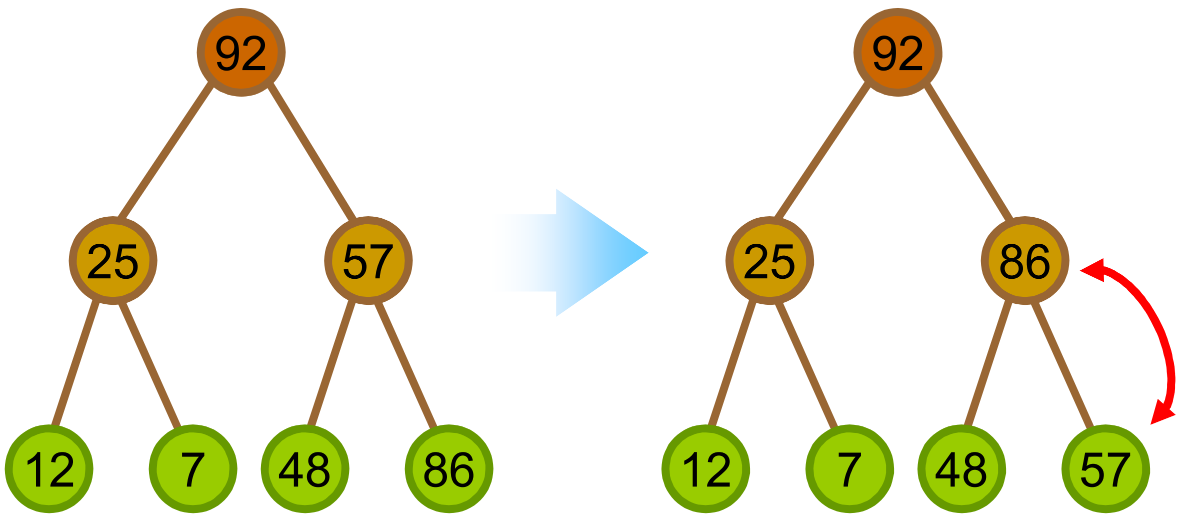




Heapsort Example: Phase 1

25	57	48	12	7	92	86	33

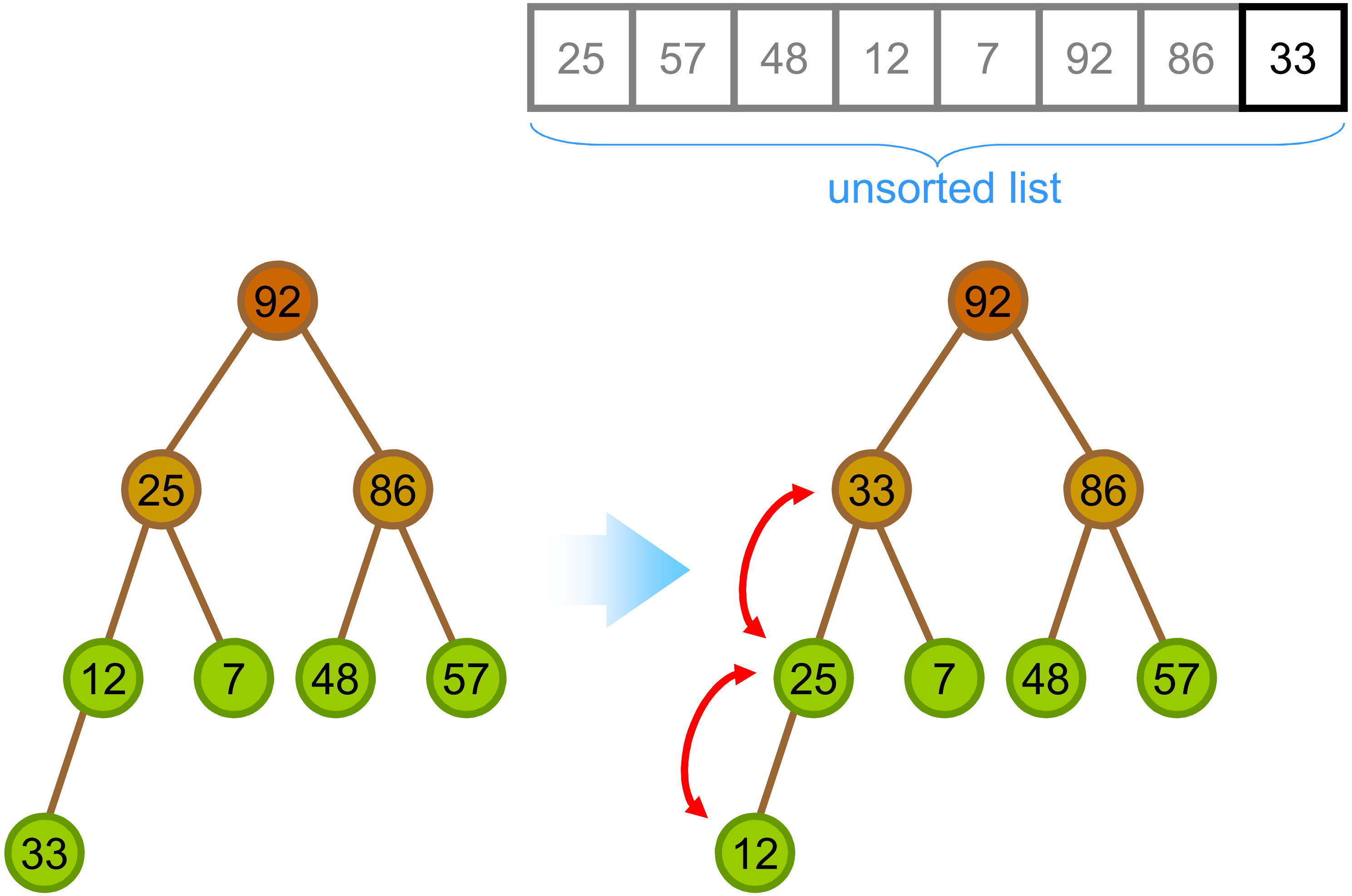
unsorted list



Insert 86

Percolate up

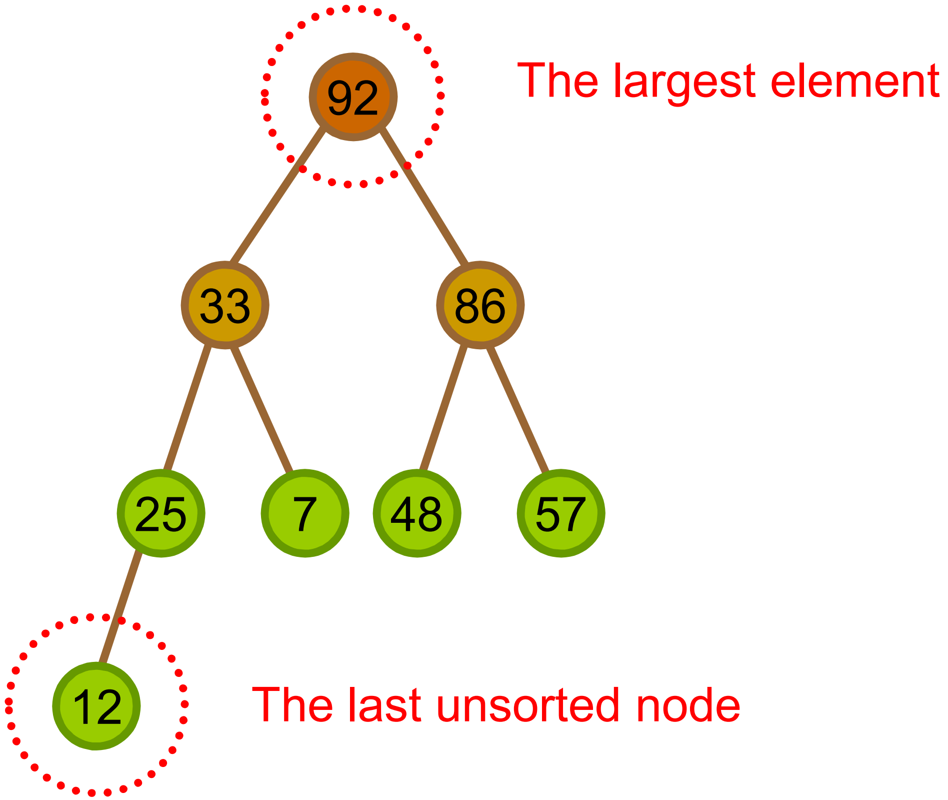
Heapsort Example: Phase 1



Insert 33

Percolate up twice

Heapsort Example: Phase 1

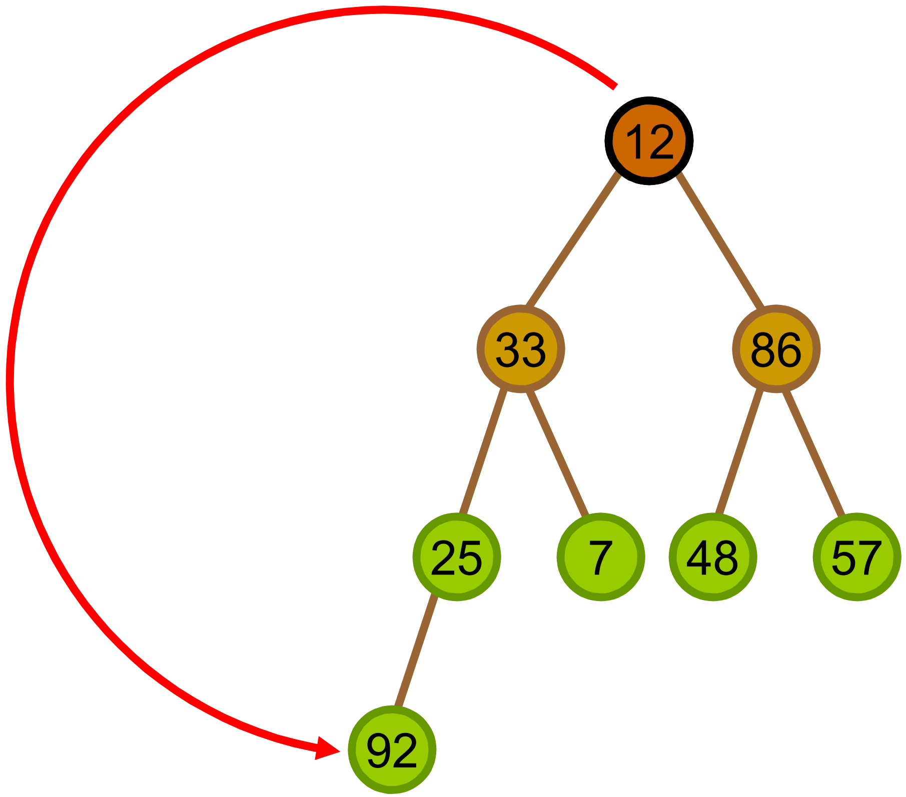


Heap array

92	33	86	25	7	48	57	12

unsorted list

Phase 2) 1st Pass



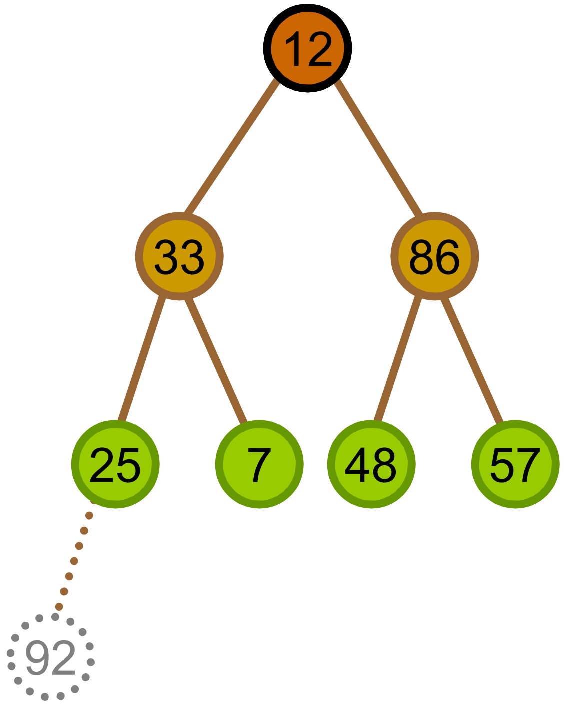
Swap root with the last unsorted node

Heap array

12	33	86	25	7	48	57	92

$\underbrace{12 \; 33 \; 86 \; 25 \; 7 \; 48 \; 57}_{\text{unsorted list}} \; \underbrace{92}_{\text{sorted list}}$

Phase 2) 1st Pass



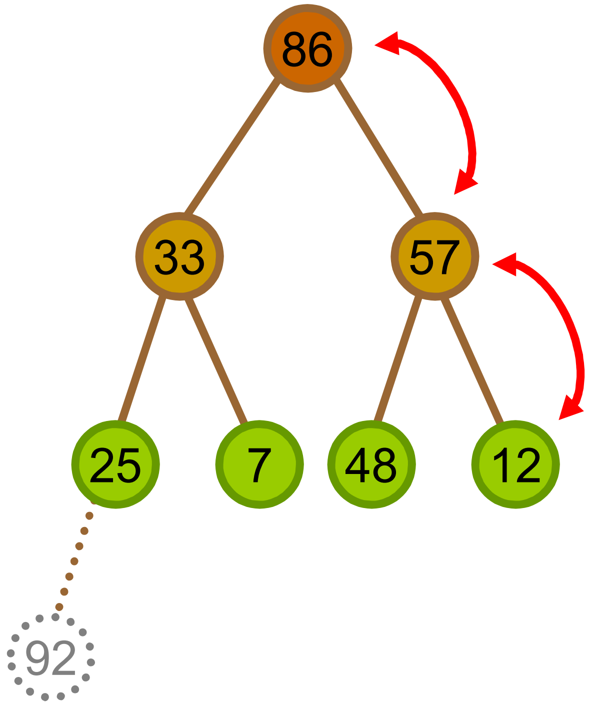
The original root (max node) can now be ignored

Heap array

12	33	86	25	7	48	57	92

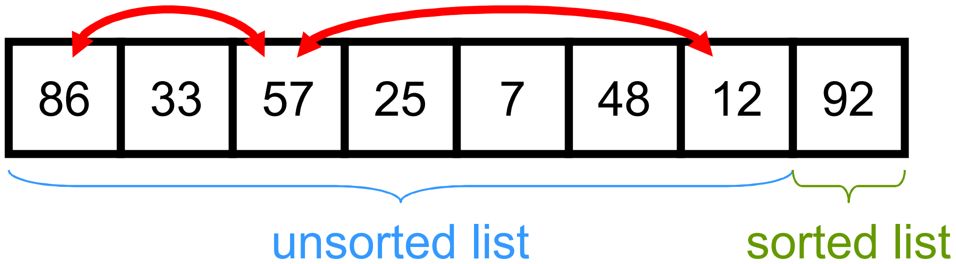
unsorted list sorted list

Phase 2) 1st Pass

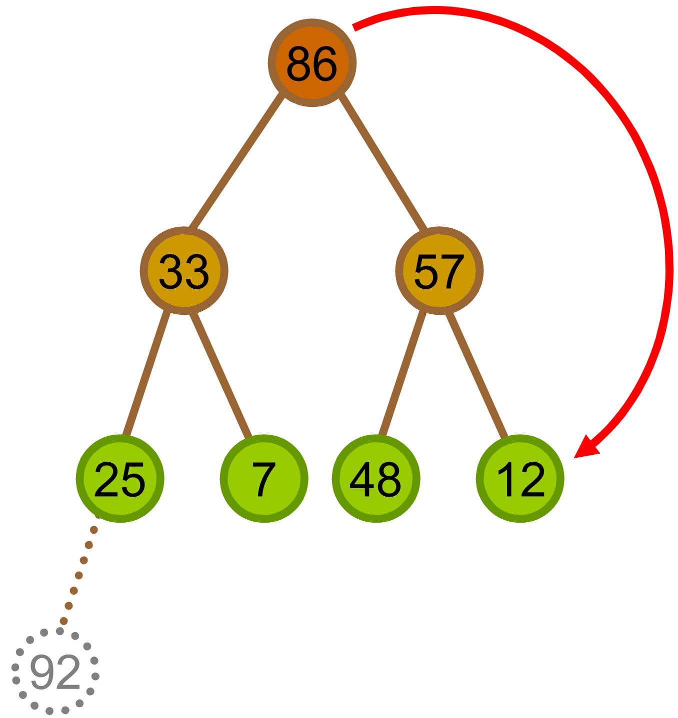


Percolate down the new root node if it violates property 2

Heap array



Phase 2) 2nd Pass



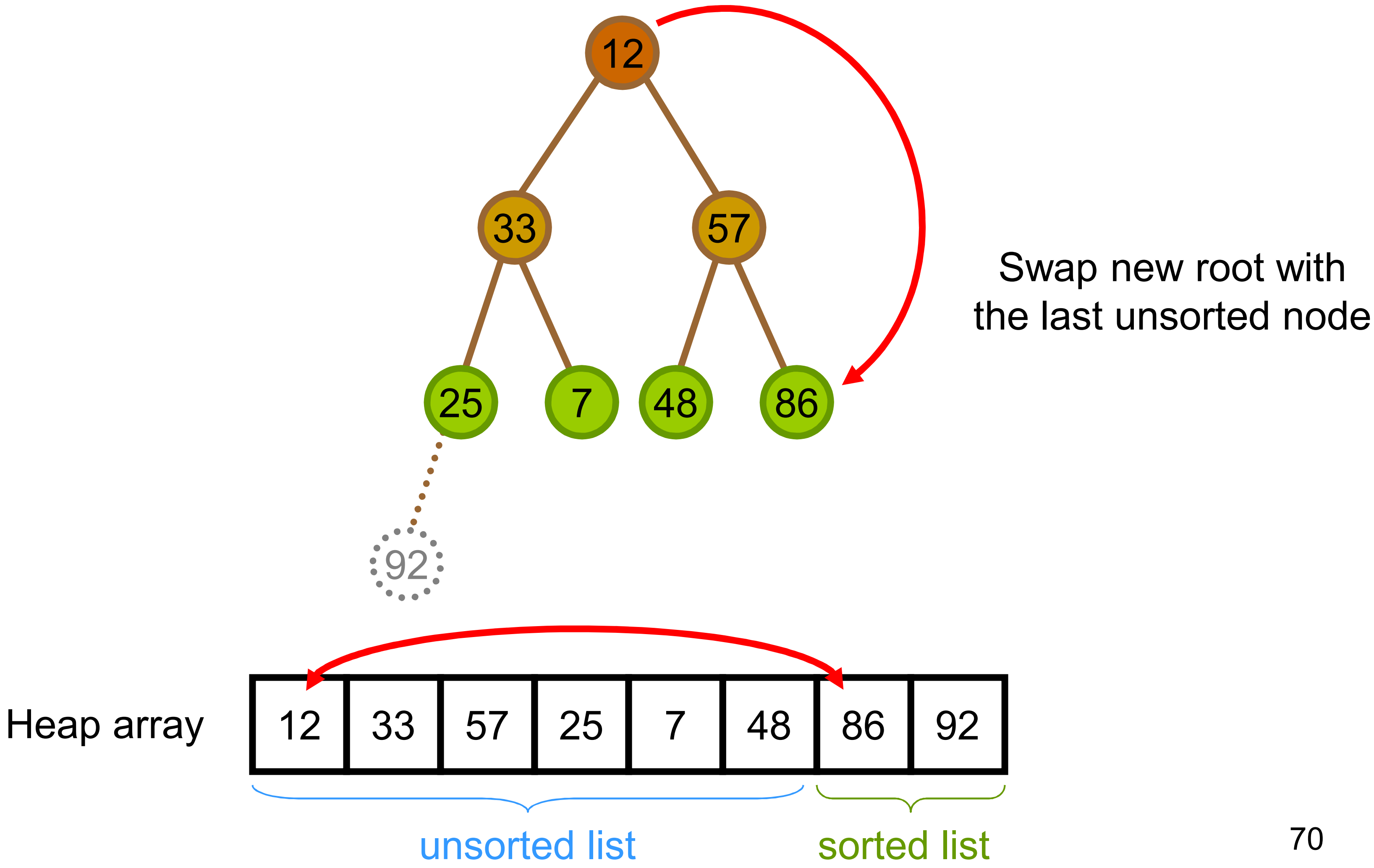
Swap new root with the last unsorted node

Heap array	86	33	57	25	7	48	12	92

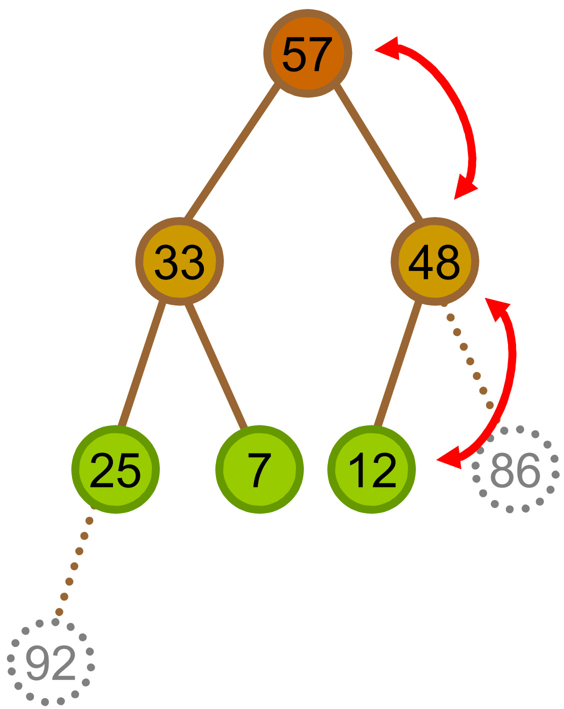
unsorted list

sorted list

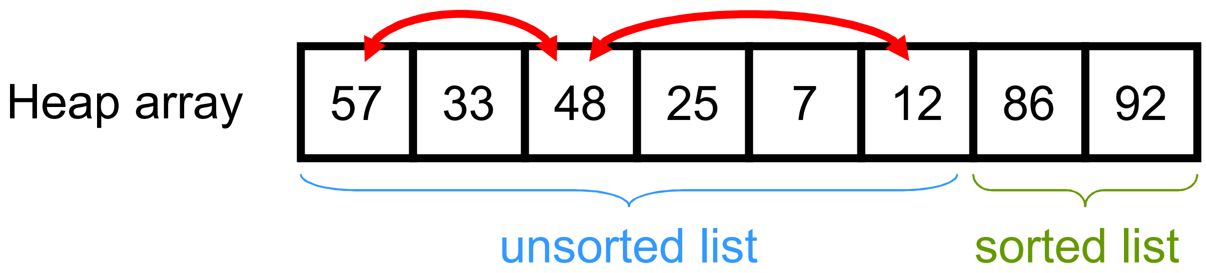
Phase 2) 2^{nd} Pass



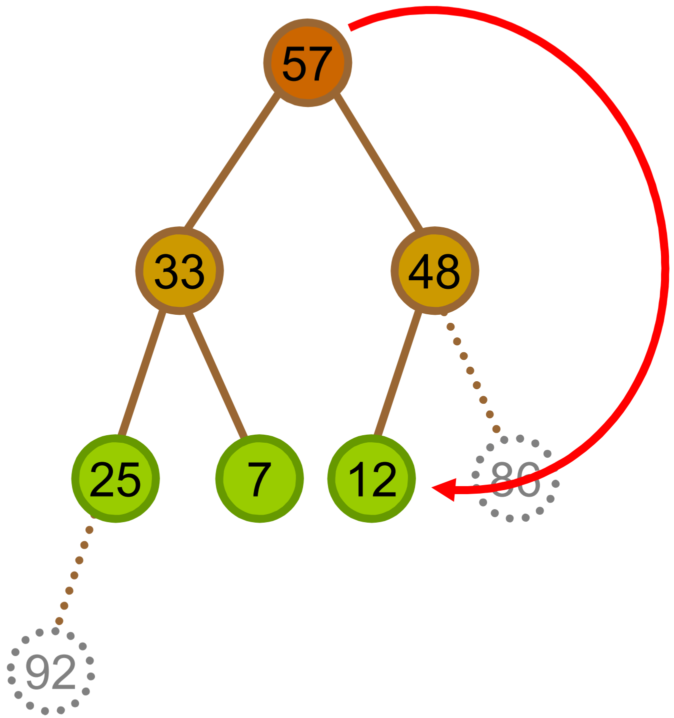
Phase 2) 2^{nd} Pass



Percolate down the new root node



Phase 2) 3rd Pass



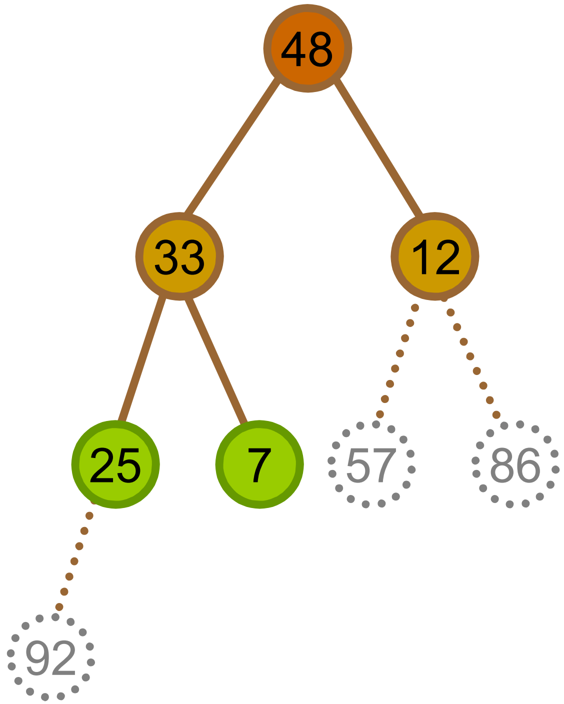
And so on…

Heap array

57	33	48	25	7	12	86	92

unsorted list sorted list

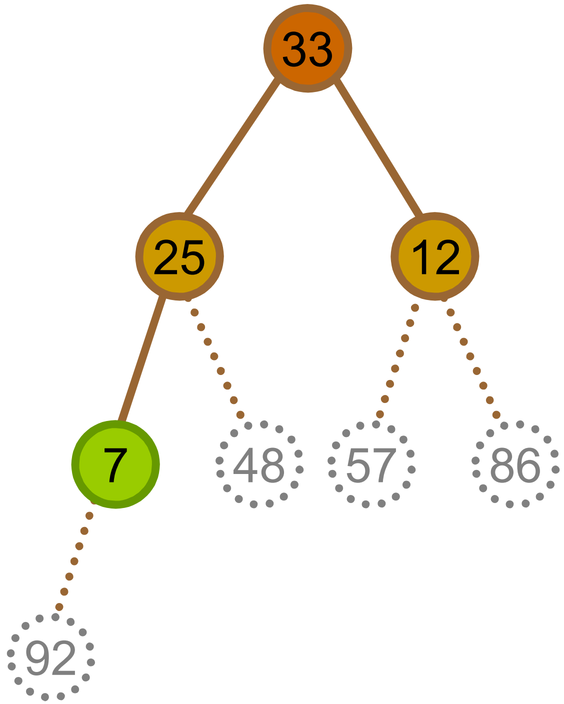
Phase 2) After 3rd Pass



Heap array	48	33	12	25	7	57	86	92

$\underbrace{48\ 33\ 12\ 25\ 7}_{\text{unsorted list}}\ \underbrace{57\ 86\ 92}_{\text{sorted list}}$

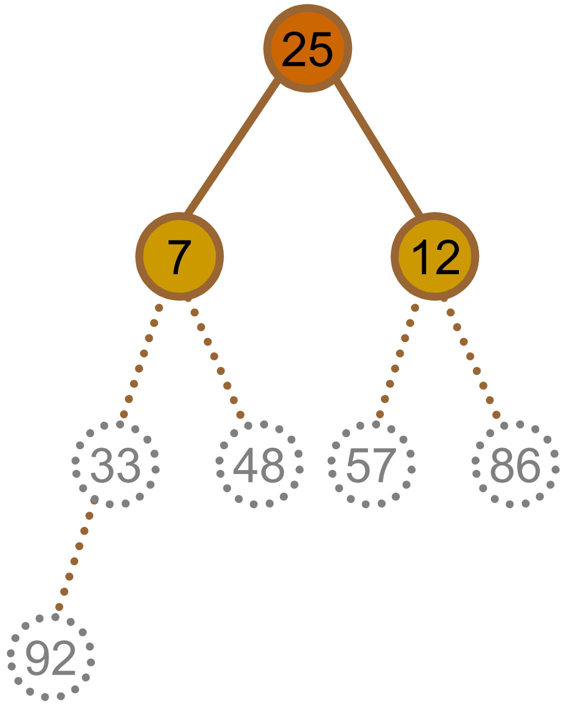
Phase 2) After 4th Pass



Heap array								
Heap array	33	25	12	7	48	57	86	92

unsorted list sorted list

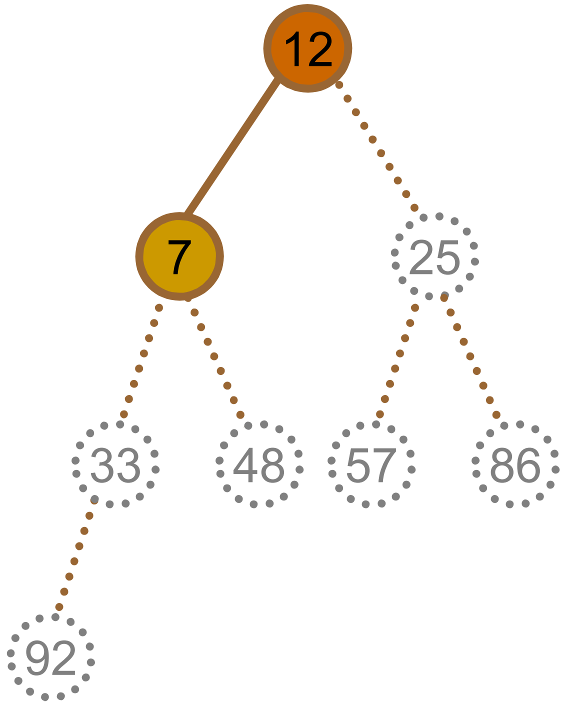
Phase 2) After 5th Pass



Heap array	25	7	12	33	48	57	86	92

unsorted list sorted list

Phase 2) After 6th Pass

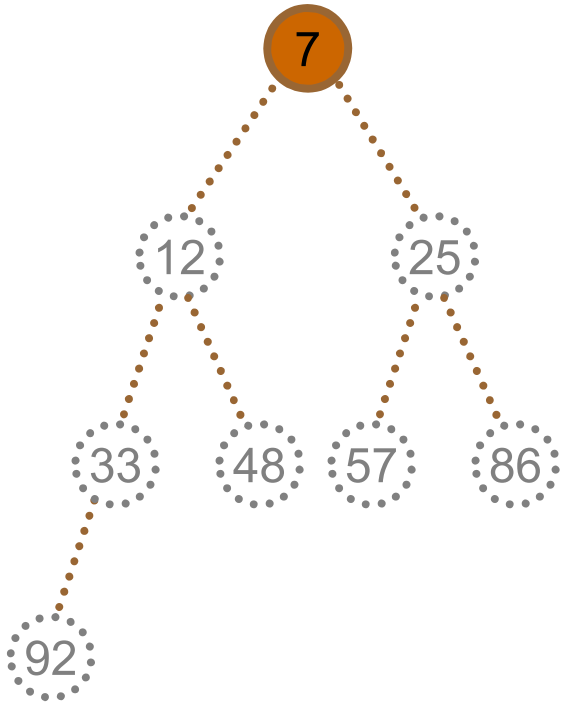


Heap array

12	7	25	33	48	57	86	92

unsorted list　　sorted list

Phase 2) After 7th Pass



Until only one node left

Heap array

7	12	25	33	48	57	86	92

sorted list

Complexity Analysis

- Time to build the heap tree
  - Suppose there are *n* nodes
  - The depth of the tree is $\log_2 n$
  - So at most $\log_2 n$ comparison for each percolate up
  - Total $n \cdot \log_2 n$
- Time to sort the data
  - About $\log_2 n$ time for each percolate down process
  - Total $(n\text{-}1)\log_2 n$
- Time complexity: $O(n \cdot \log n)$
  - Go through the same steps in the second phase (percolate down)
  - Best case = Worst case = Average case
- Extra space is required for swapping the nodes
  - Space Complexity: O(1)

Heapsort (Recursive Version)

```
void percolateUp(int data[], int index) {
    int parent = (index - 1) / 2;

    if (parent < 0) return;                //base case
    //note: if parent >= 0, index also >= 0

    if (data[index] > data[parent]) {      //general case
        swap(&data[index], &data[parent]);
        percolateUp(data, parent);
    }
}
```

Heapsort (Recursive Version)

```
void percolateDown(int data[], int n, int index) {
    int left, right, maxIndex;

    if (index < 0 || index >= n) return;            //base case 1

    left = 2 * index + 1;
    right = left + 1;
    if (left >= n) return;                          //base case 2

    maxIndex = right < n && data[left] < data[right] ? right : left;
    if (data[index] < data[maxIndex]) {             //general case
        swap(&data[index], &data[maxIndex]);
        percolateDown(data, n, maxIndex);
    }
}
```

Heapsort

```
void heapsort(int data[], int n) {
    int i, last;
    for (i = 1; i < n; i++) {             //start from index 1
        percolateUp(data, i);             //build the max. heap
    }
    for (last = n – 1; last > 0; last--) {
        swap(&data[0], &data[last]);      //sort the sequence
        percolateDown(data, last, 0);
    }
}
```

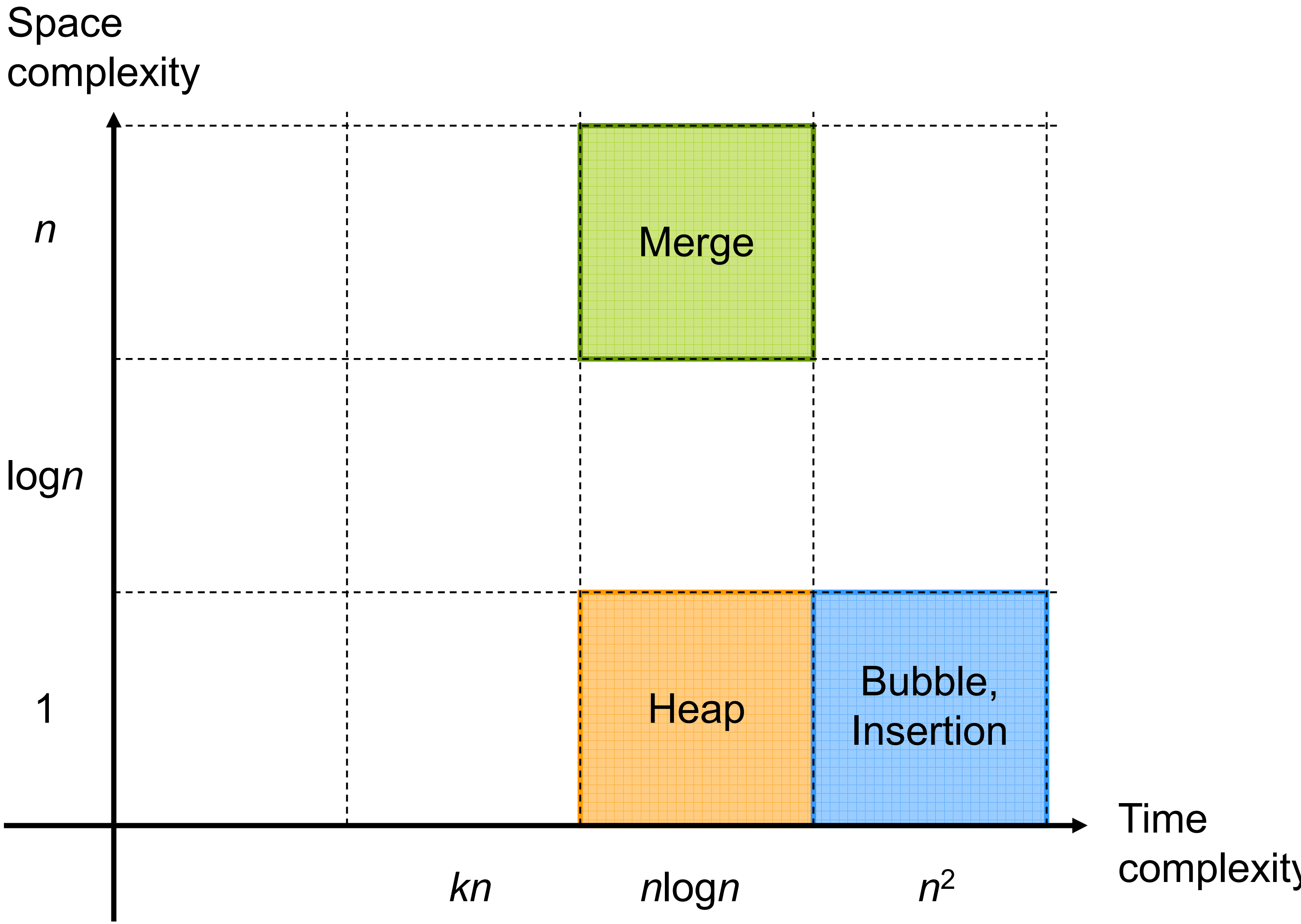
Heapsort (Iterative Version)

```
void percolateUp(int data[], int index) {

}
```

Heapsort (Iterative Version)

```
void percolateDown(int data[], int n, int index) {

}
```

Summary


Space complexity
n
logn
1
Merge
Heap
Bubble, Insertion
Time complexity
kn
nlogn
n2