Sorting

CS61B Spring'15 Discussion 11

Some Interesting Properties

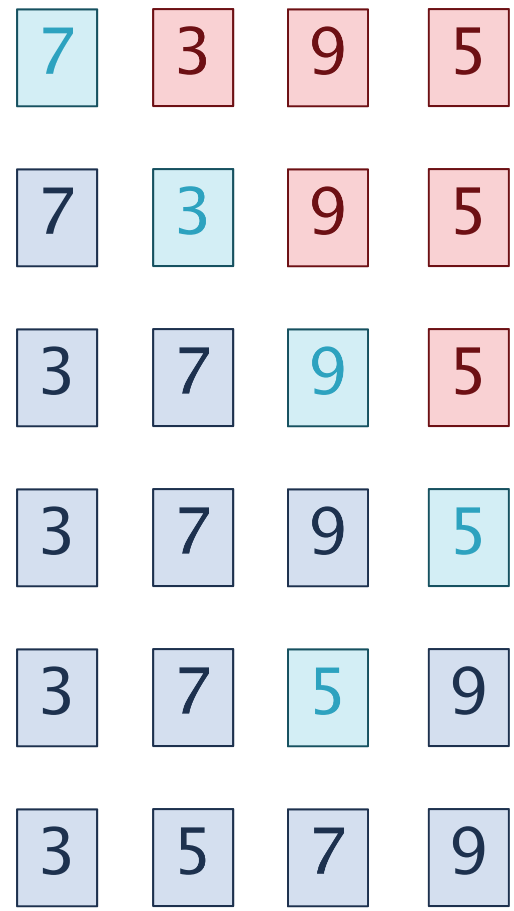
- **In-Place Sort:**
  - Keeps sorted items in original array (destructive)
  - No equivalent for linked list-based input
- **Stable Sort:**
  - Keeps elements with equal keys in same relative order in output as in input

Insertion Sort

Algorithm

- Partition array into unsorted portion *U* and sorted portion *S*.
- For each item *x* in *U*: swap *x* with left neighbor until left neighbor is $\leq x$.

Intuition: Insert *x* into correct position in *S*.



Insertion Sort

- Question 1(a): Insertion Sort.

```
106 351 214 873 615 172 333 564
```

Insertion Sort

- Question 1(a): Insertion Sort.

106 351 214 873 615 172 333 564
106|351 214 873 615 172 333 564
106 351|214 873 615 172 333 564
106 214 351|873 615 172 333 564
106 214 351 873|615 172 333 564
106 214 351 615 873|172 333 564
106 172 214 351 615 873|333 564
106 172 214 333 351 615 873|564
106 172 214 333 351 564 615 873|

Insertion Sort

Question 3.

- Worst-Case Runtime:
- Best-Case Runtime:
- In-Place?
- Stable?

Insertion Sort

Question 3.

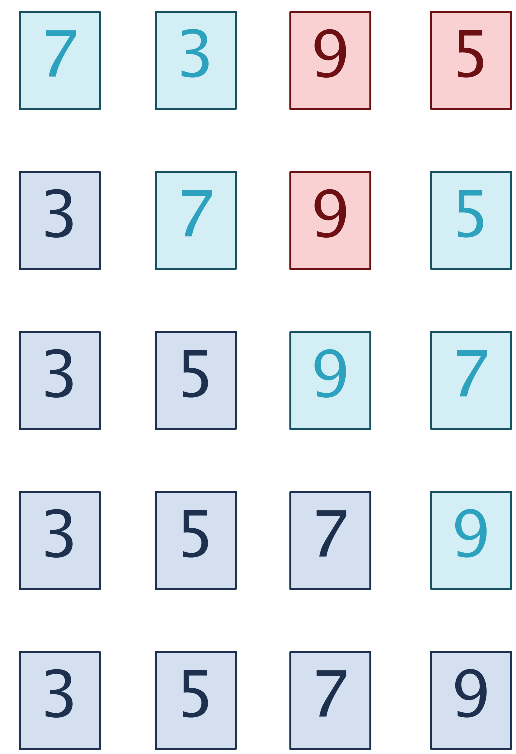
- Worst-Case Runtime: $\Theta(N^2)$
- Best-Case Runtime: $\Theta(N)$
- In-Place? ✓ Yes
- Stable? ✓ Yes

Selection Sort

Algorithm

For each position `i`:

- Find smallest element x from `i` to end.
- Swap x and `arr[i]`.



Selection Sort

- Question 1(b): Selection Sort.

106 351 214 873 615 172 333 564

Selection Sort

- Question 1(b): Selection Sort.

```
106 351 214 873 615 172 333 564
106|351 214 873 615 172 333 564
106 172|214 873 615 351 333 564
106 172 214|873 615 351 333 564
106 172 214 333|615 351 873 564
106 172 214 333 351|615 873 564
106 172 214 333 351 564|873 615
106 172 214 333 351 564 615|873
```

Selection Sort

Question 3.

- Worst-Case Runtime:
- Best-Case Runtime:
- In-Place?
- Stable?

Selection Sort

Question 3.

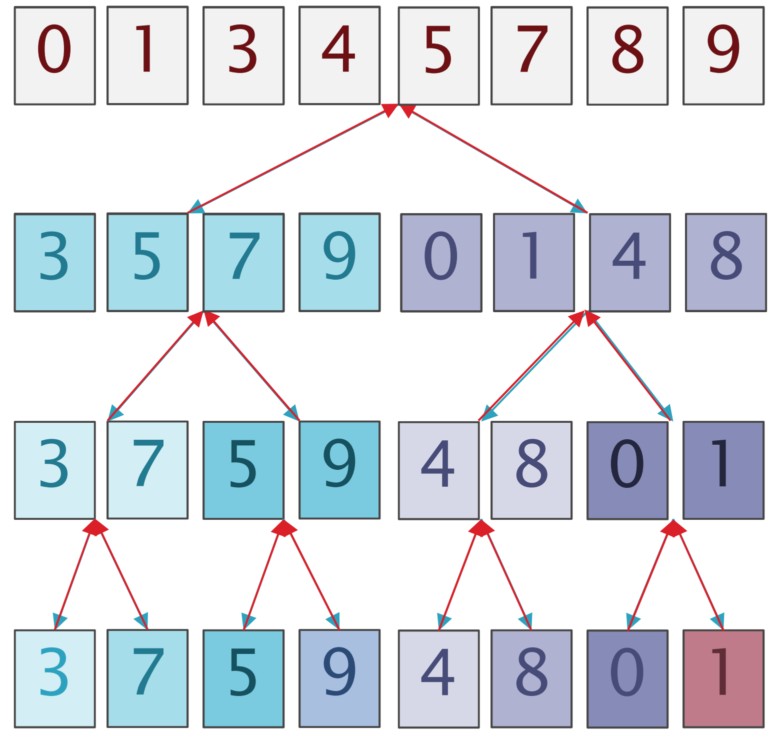
- Worst-Case Runtime: $\Theta(n^2)$
- Best-Case Runtime: $\Theta(n^2)$
- In-Place? ✓ Yes
- Stable? ✓ Yes

Merge Sort

Algorithm

- Given N items, split into left half and right half.
- Mergesort left half.
- Mergesort right half.
- Merge the sorted halves together.

How? See Q4: MergeTwo.

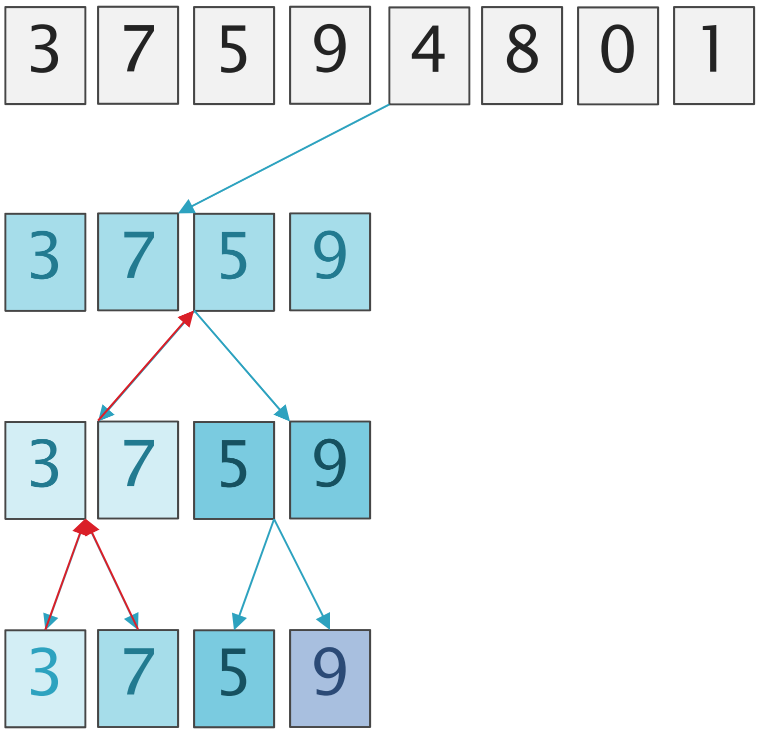


Merge Sort

Algorithm

- Given N items, split into left half and right half.
- Mergesort left half.
- Mergesort right half.
- Merge the sorted halves together.

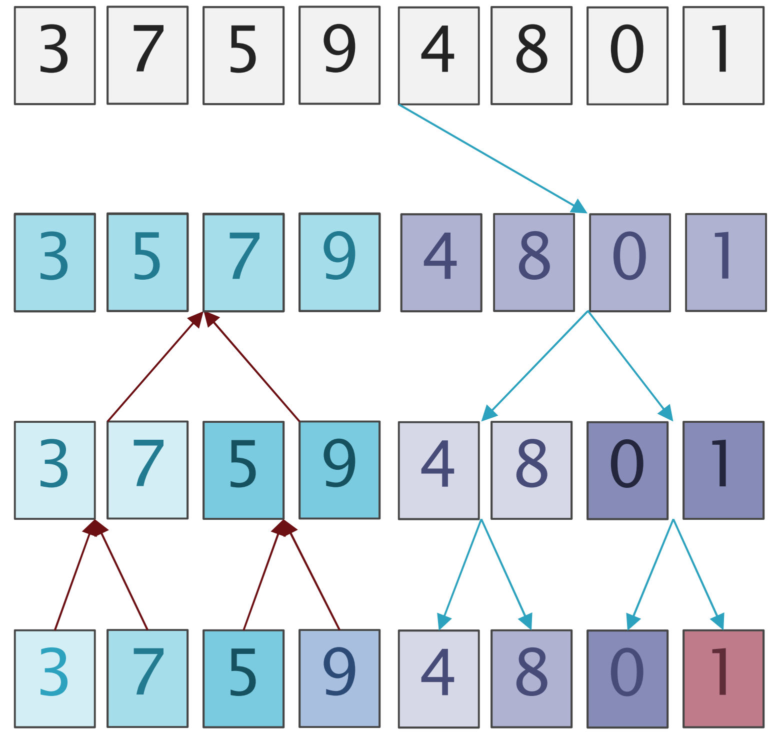
How? See Q4: MergeTwo.



Merge Sort

Algorithm

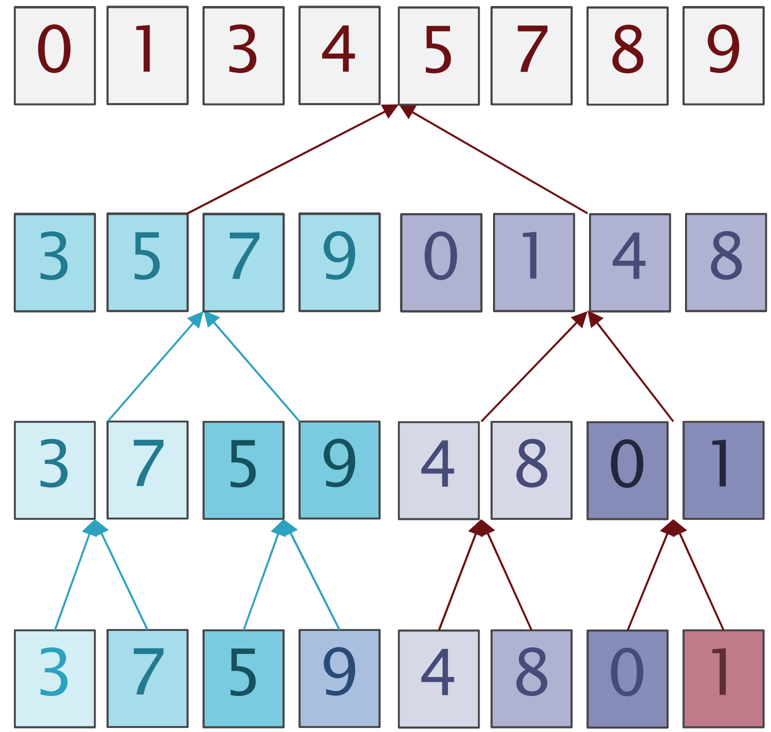
- Given N items, split into left half and right half.
- Mergesort left half.
- Mergesort right half.
- Merge the sorted halves together.



Merge Sort

Algorithm

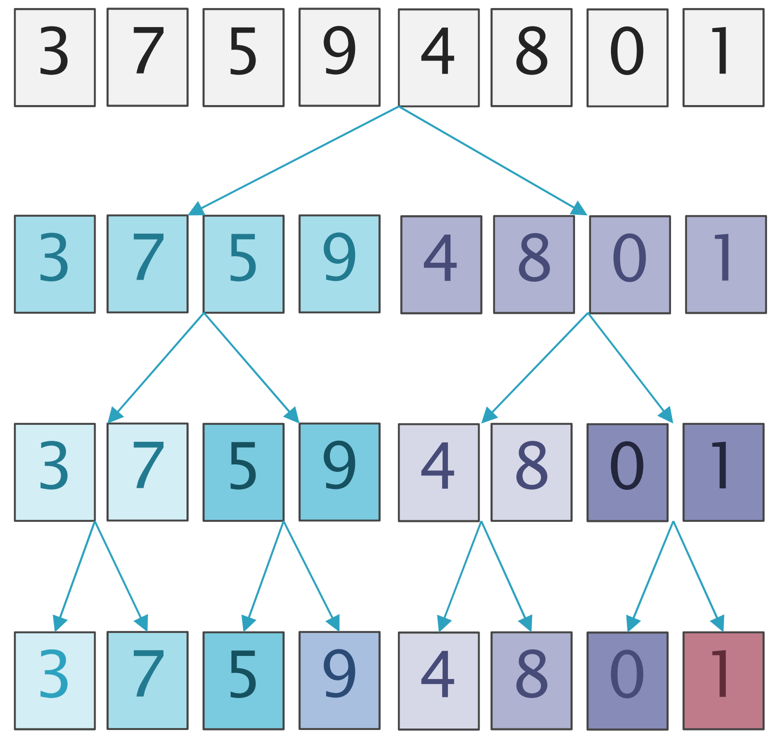
- Given N items, split into left half and right half.
- Mergesort left half.
- Mergesort right half.
- Merge the sorted halves together.



Merge Sort

Algorithm

- Given N items, split into left half and right half.
- Mergesort left half.
- Mergesort right half.
- Merge the sorted halves together.



Merge Sort

- Question 1(c): Merge Sort.

```
106 351 214 873 615 172 333 564
```

Merge Sort

- Question 1(c): Merge Sort.

106 351 214 873 615 172 333 564

106 351 214 873 | 615 172 333 564

106 351 | 214 873 | 615 172 | 333 564

106 | 351 | 214 | 873 | 615 | 172 | 333 | 564

106 351 | 214 873 | 172 615 | 333 564

106 214 351 873 | 172 333 564 615

106 172 214 333 351 564 615 873

Merge Sort

Question 3.

- Worst-Case Runtime:
- Best-Case Runtime:
- In-Place?
- Stable?

Merge Sort

Question 3.

- Worst-Case Runtime: $\Theta(n \log n)$
- Best-Case Runtime: $\Theta(n \log n)$
- In-Place? ✗ No
- Stable? ✓ Yes

Heap Sort

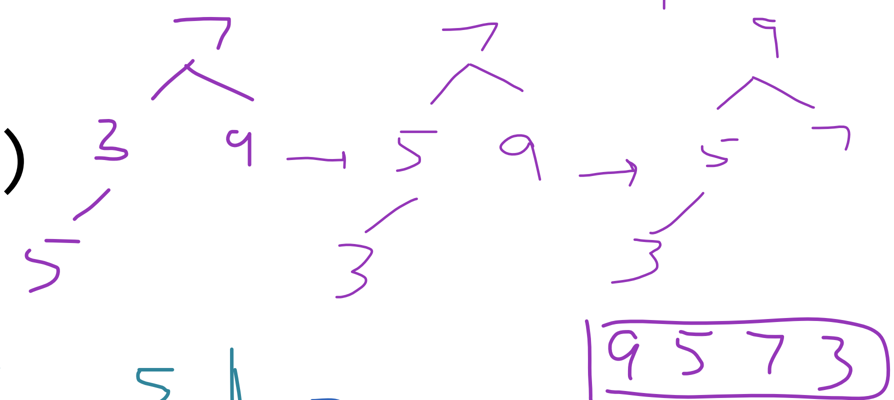
1. Start with first internal node and do reverse level-order traversal.
2. Bubble down to fix heap.

Algorithm

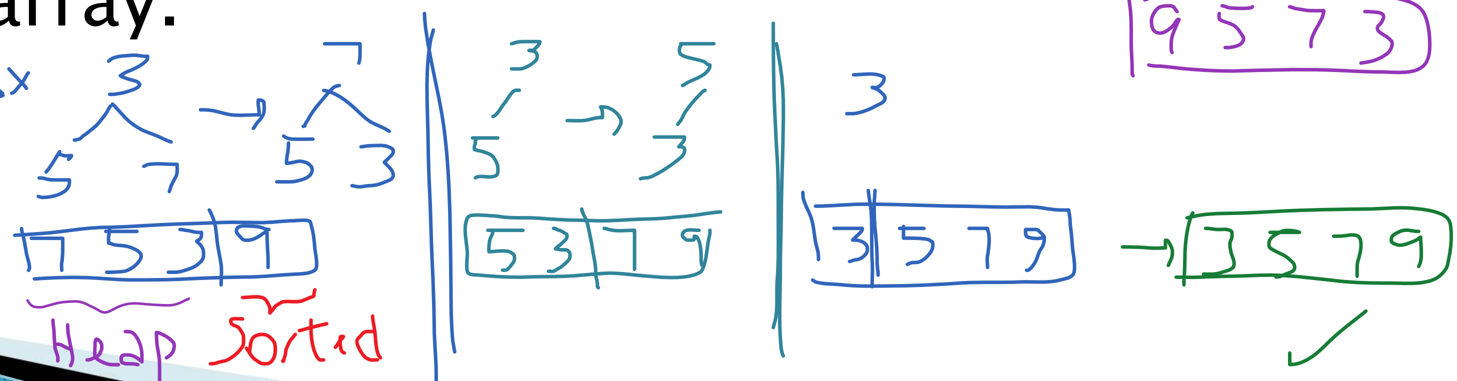
- Create max-heap with `bottomUpHeap()`.
- Repeatedly `deleteMax()` and place element at end of array.

Sort: 7 3 9 5

① Bottom-Up Heap



② Delete Max



Heap Sort

- Question 1(d): Heap Sort.

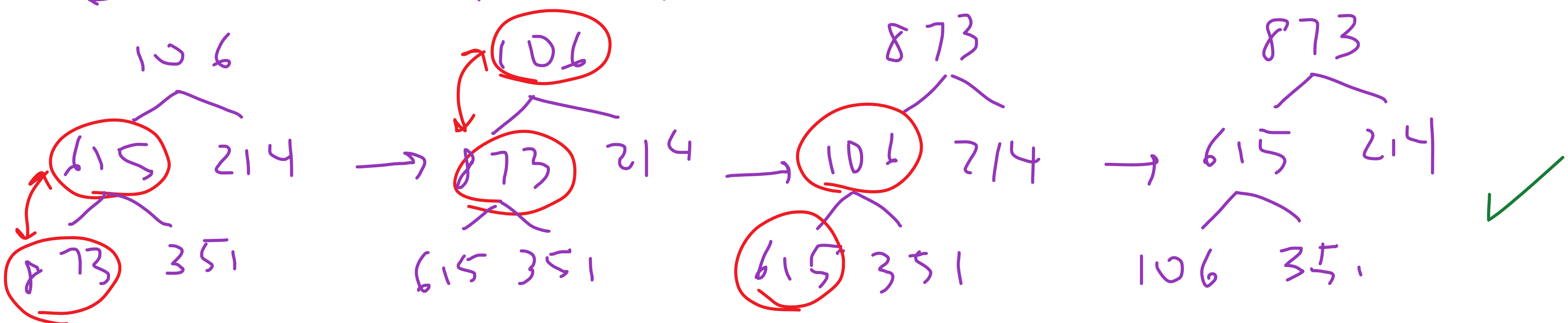
106 615 214 873 351

Heap Sort

- Question 1(d): Heap Sort.

106 615 214 873 351

① Bottom-up Heap.



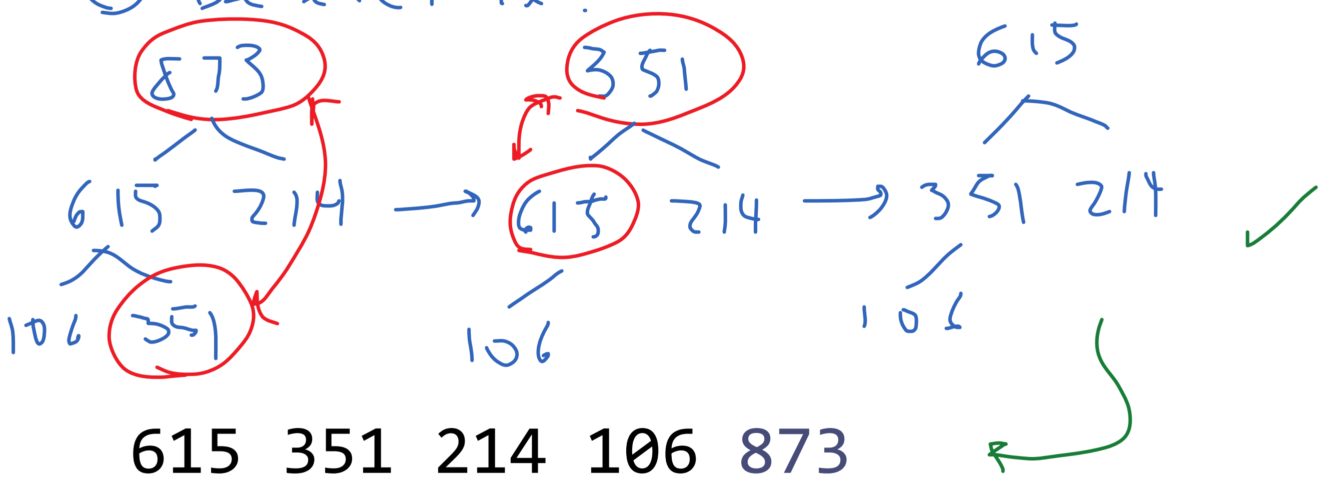
873 615 214 106 351

Heap Sort

- Question 1(d): Heap Sort.

873 615 214 106 351

② Delete Max



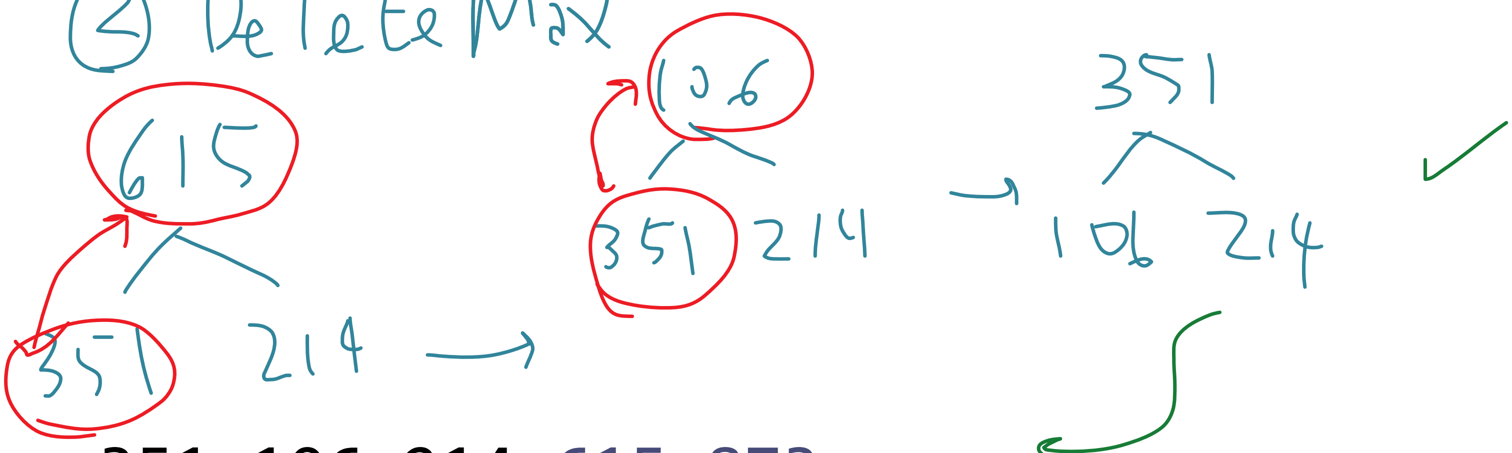
615 351 214 106 873

Heap Sort

- Question 1(d): Heap Sort.

615 351 214 106 873

② DeleteMax



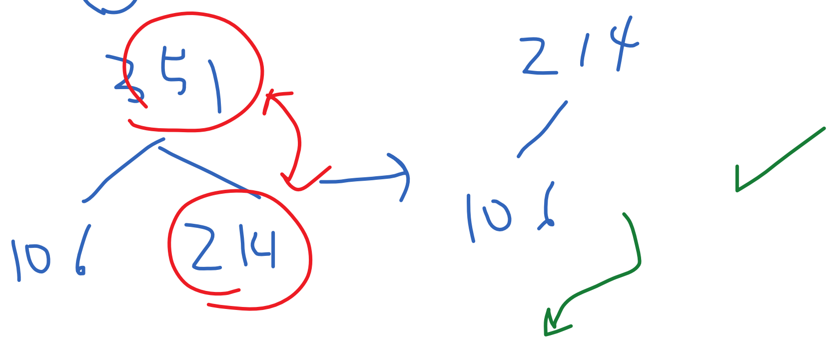
351 106 214 615 873

Heap Sort

- Question 1(d): Heap Sort.

351 106 214 615 873

2 deleteMax()



214 106 351 615 873

Heap Sort

- Question 1(d): Heap Sort.

214 106 351 615 873

③ Delete Max.

214

106

→ 106 ✓

↓

106 214 351 615 873 ← Final answer ☺

Heap Sort

Question 3.

- Worst-Case Runtime:
- Best-Case Runtime:
- In-Place?
- Stable?

Heap Sort

Question 3.

- Worst-Case Runtime: $\Theta(n \log n)$
- Best-Case Runtime: $\Theta(n)$
- In-Place? ✓ Yes
- Stable? ✗ No

  Unless you add a secondary key (a timestamp) to break ties.

Merge sort isn't always better!

- Question 1(e). Give an example of a situation when using insertion sort is more efficient than using merge sort.

Merge sort isn't always better!

- **Question 1(e). Give an example of a situation when using insertion sort is more efficient than using merge sort.**
- Insertion sort outperforms merge sort for lists that are "mostly sorted".
  - If list has only a few elements out of place
  - If all elements are within k positions of their proper place and $k < \log N$

2(a) Sorting II

- Which sorting algorithm?

```
12 7 8 4 10 2 5 34 14
2 4 5 7 8 12 10 34 14
```

2(a) Sorting II

- Which sorting algorithm?

```
12 7 8 4 10 2 5 34 14
2 4 5 7 8 12 10 34 14
```

- **Selection Sort.**

```
12  7  8  4 10  2  5 34 14
 2  7  8  4 10 12  5 34 14
 2  4  8  7 10 12  5 34 14
 2  4  5  7 10 12  8 34 14
 2  4  5  7  8 12 10 34 14
```

2(b) Sorting II

- Which sorting algorithm?

```
23 45 12 4 65 34 20 43
12 23 45 4 65 34 20 43
```

2(b) Sorting II

- Which sorting algorithm?

  23 45 12 4 65 34 20 43

  12 23 45 4 65 34 20 43

- Insertion Sort.

  23 45 12 4 65 34 20 43

  23 12 45 4 65 34 20 43

  12 23 45 4 65 34 20 43

2(c) Sorting II

- Which sorting algorithm?

  45 23 5 65 34 3 76 25

  23 45 5 65 3 34 25 76

  5 23 45 65 3 25 34 76

2(c) Sorting II

- Which sorting algorithm?

45 23 5 65 34 3 76 25

23 45 5 65 3 34 25 76

5 23 45 65 3 25 34 76

- **Merge Sort.**

45 23 5 65 34 3 76 25

23 45 5 65 3 34 25 76

5 23 45 65 3 25 34 76

2(d) Sorting II

- Which sorting algorithm?

12 32 14 34 17 38 23 11

12 14 17 32 34 38 23 11

2(d) Sorting II

- Which sorting algorithm?

12 32 14 34 17 38 23 11

12 14 17 32 34 38 23 11

- Insertion Sort.

12 32 14 34 17 38 23 11

12 14 32 34 17 38 23 11

12 14 32 17 34 38 23 11

12 14 17 32 34 38 23 11

4 MergeTwo

- Suppose you are given two sorted arrays of ints. Fill in the method mergeTwo to return a new array containing all of the elements of both arrays in sorted order. Duplicates are allowed (if an element appears s times in a and t times in b, then it should appear $s + t$ times in the returned array.

```java
public static int[] mergeTwo(int[] a, int[] b) {
  // YOUR CODE HERE
}
```

4 MergeTwo

- Create new array for result.
- Initialize three counters (for indices)
- Merge while both arrays have unmerged elements.
- Array b is done, so append rest of a.
- Array a is done, so append rest of b.

4 MergeTwo

```java
public static int[] mergeTwo(int[] a, int[] b) {
    int[] merged = new int[a.length + b.length];
    int aIndex = 0;  // Current index in a.
    int bIndex = 0;  // Current index in b.
    int mergedIndex = 0;  // Current index in merged.

    while (aIndex < a.length && bIndex < b.length) {
        if (a[aIndex] < b[bIndex]) {
            merged[mergedIndex] = a[aIndex];
            aIndex++;
        } else {
            merged[mergedIndex] = b[bIndex];
            bIndex++;
        }
        mergedIndex++;
    }                                           // Continued on next slide.
```

4 MergeTwo

```
    while (aIndex < a.length) {
        merged[mergedIndex] = a[aIndex];
        aIndex++;
        mergedIndex++;
    }

    while (bIndex < b.length) {
        merged[mergedIndex] = b[bIndex];
        bIndex++;
        mergedIndex++;
    }
}
```