DATA STRUCTURES AND ALGORITHMS

Searching and Sorting

Introduction

- Searching data involves determining whether a value (referred to as the search key) is present in the data and, if so, finding the value's location.
  - Two popular search algorithms are the simple *linear search* and the faster but more complex *binary search.*
- Sorting places data in ascending or descending order, based on one or more sort keys.
- This also introduces Big O notation, which is used to characterize an algorithm's worst-case runtime—that is, how hard an algorithm may have to work to solve a problem.

Searching Algorithms

- Looking up a phone number, accessing a website and checking a word's definition in a dictionary all involve searching through large amounts of data.
- Searching algorithms all accomplish the same goal—finding an element that matches a given search key, if such an element does, in fact, exist.
- The major difference is the amount of effort they require to complete the search.
- One way to describe this effort is with Big O notation.

  For searching and sorting algorithms, this is particularly dependent on the number of data elements.

Searching Algorithms (cont.)

- In this section we present the linear search algorithm then discuss the algorithm's efficiency as measured by Big O notation.
- we introduce the binary search algorithm, which is much more efficient but more complex to implement.

Linear Search

Function Template linearSearch

- Function template linearSearch (lines 10–18) compares each element of an array with a search key (line 14).
- Because the array is not in any particular order, it's just as likely that the search key will be found in the first element as the last.
- On average, therefore, the program must compare the search key with half of the array's elements.
- To determine that a value is not in the array, the program must compare the search key to every array element.
- Linear search works well for small or unsorted arrays.
- However, for large arrays, linear searching is inefficient.
- If the array is sorted (e.g., its elements are in ascending order), you can use the high-speed binary search technique.

Linear Search

```
//compare key to every element of array until location is
found
//or until end of array is reached
//return position of element if key is found or -1 if key is not
found
void linearSearch(int [] items,int size,int key)
{
for(int i=0;i<size-1;++next)
{
if(key == items[i]) //if found
return i; //return  location of key
}
return -1;
}//end of function insetionSort
```

Linear Search

```
class Mainclass
{
static void main()
{
final int sz=10;
int data[]={34,56,4,10,77,51,93,30,5,52};
system.out.println("Array Elements");
for(int val:data)
system.out.print(val+"  ");
int element=linearSearch(data,10,93);
if(element!=-1)
system.out.println("Found value at Position "+element);
else
system.out.println("Value not found");
}

}
```

```
Enter integer search key: 37
Value not found
```

Efficiency of Linear Search

Big O: Constant Runtime

- Suppose an algorithm simply tests whether the first element of an array is equal to the second element.
- If the array has 10 elements, this algorithm requires only one comparison.
- If the array has 1000 elements, the algorithm still requires only one comparison.
- In fact, the algorithm is independent of the number of array elements.
- This algorithm is said to have a constant runtime, which is represented in Big O notation as O(1).
- An algorithm that is O(1) does not necessarily require only one comparison.
- O(1) just means that the number of comparisons is constant—it does not grow as the size of the array increases.
- An algorithm that tests whether the first element of an array is equal to any of the next three elements will always require three comparisons, but in Big O notation it's still considered O(1).
- O(1) is often pronounced "on the order of 1" or more simply "order 1."

Efficiency of Linear Search (cont.)

Big O: Linear Runtime

- An algorithm that tests whether the first element of an array is equal to any of the other elements of the array requires at most n – 1 comparisons, where n is the number of elements in the array.
- If the array has 10 elements, the algorithm requires up to nine comparisons.
- If the array has 1000 elements, the algorithm requires up to 999 comparisons.
- As n grows larger, the n part of the expression n – 1 “dominates,” and subtracting one becomes inconsequential.
- Big O is designed to highlight these dominant terms and ignore terms that become unimportant as n grows.

Efficiency of Linear Search (cont.)

- An algorithm that requires a total of n – 1 comparisons is said to be O(n) and is referred to as having a linear runtime.
- O(n) is often pronounced "on the order of n" or more simply "order n."

Efficiency of Linear Search (cont.)

Big O: Quadratic Runtime

- Now suppose you have an algorithm that tests whether any element of an array is duplicated elsewhere in the array.
- The first element must be compared with every other element in the array.
- The second element must be compared with every other element except the first (it was already compared to the first).
- The third element then must be compared with every other element except the first two.
- In the end, this algorithm will end up making (n – 1) + (n – 2) + … + 2 + 1 or n2/2 – n/2 comparisons.
- As n increases, the n2 term dominates and the n term becomes inconsequential.
- Again, Big O notation highlights the n2 term, leaving n2/2.

Efficiency of Linear Search (cont.)

- Big O is concerned with how an algorithm's runtime grows in relation to the number of items processed.
- Suppose an algorithm requires n2 comparisons.
- With four elements, the algorithm will require 16 comparisons; with eight elements, 64 comparisons.
- With this algorithm, doubling the number of elements quadruples the number of comparisons.
- Consider a similar algorithm requiring n2/2 comparisons.
- With four elements, the algorithm will require eight comparisons; with eight elements, 32 comparisons.
- Again, doubling the number of elements quadruples the number of comparisons.
- Both of these algorithms grow as the square of n, so Big O ignores the constant, and both algorithms are considered to be O(n2), which is referred to as quadratic runtime and pronounced "on the order of n-squared" or more simply "order n-squared."

Efficiency of Linear Search (cont.)

O(n2) Performance

- When n is small, O(n2) algorithms will not noticeably affect performance.
- As n grows, you'll start to notice the performance degradation.
- An O(n2) algorithm running on a million-element array would require a trillion "operations" (where each could actually require several machine instructions to execute).
- This could require hours to execute.
- A billion-element array would require a quintillion operations, a number so large that the algorithm could take decades! Unfortunately, O(n2) algorithms tend to be easy to write.

Efficiency of Linear Search (cont.)

- In this chapter, you'll see algorithms with more favorable Big O measures.
- These efficient algorithms often take a bit more cleverness and effort to create, but their superior performance can be worth the extra effort, especially as n gets large and as algorithms are compounded into larger programs.

Efficiency of Linear Search (cont.)

Linear Search's Runtime

- The linear search algorithm runs in O(n) time.
- The worst case in this algorithm is that every element must be checked to determine whether the search key is in the array.
- If the array's size doubles, the number of comparisons that the algorithm must perform also doubles.
- Linear search can provide outstanding performance if the element matching the search key happens to be at or near the front of the array.
- But we seek algorithms that perform well, on average, across all searches, including those where the element matching the search key is near the end of the array.
- If a program needs to perform many searches on large arrays, it may be better to implement a different, more efficient algorithm, such as the binary search which we present in the next section.

Binary Search

- The binary search algorithm is more efficient than the linear search algorithm, but it requires that the array first be sorted.
- This is only worthwhile when the vector, once sorted, will be searched a great many times—or when the searching application has stringent performance requirements.
- The first iteration of this algorithm tests the middle array element.
- If this matches the search key, the algorithm ends.

Binary Search (cont.)

- Assuming the array is sorted in ascending order, then if the search key is less than the middle element, the search key cannot match any element in the array’s second half so the algorithm continues with only the first half (i.e., the first element up to, but not including, the middle element).
- If the search key is greater than the middle element, the search key cannot match any element in the array’s first half so the algorithm continues with only the second half (i.e., the element after the middle element through the last element).
- Each iteration tests the middle value of the array’s remaining elements.
- If the element does not match the search key, the algorithm eliminates half of the remaining elements.
- The algorithm ends either by finding an element that matches the search key or by reducing the sub-array to zero size.

Binary Search (cont.)

- ***Efficiency of Binary Search***

- In the worst-case scenario, searching a sorted array of 1023 elements will take only 10 comparisons when using a binary search.
- Repeatedly dividing 1023 by 2 (because, after each comparison, we can eliminate from consideration half of the remaining elements) and rounding down (because we also remove the middle element) yields the values 511, 255, 127, 63, 31, 15, 7, 3, 1 and 0.
- The number 1023 (210 – 1) is divided by 2 only 10 times to get the value 0, which indicates that there are no more elements to test.
- (lines 11–22) to display the portion of the array that's currently being searched.

Binary Search (cont.)

```
class BinarySearchExample{
 public static void binarySearch(int arr[], int first, int last, in
t key){
   int mid = (first + last)/2;
   while( first <= last ){
     if ( arr[mid] < key ){
       first = mid + 1;
     }else if ( arr[mid] == key ){
       System.out.println("Element is found at index: " + mid);
       break;
     }else{
        last = mid - 1;
     }
     mid = (first + last)/2;
   }
   if ( first > last ){
     System.out.println("Element is not found!");
}  }
```

Binary Search (cont.)

```
public static void main(String args[]){
    int arr[] = {10,20,30,40,50};
    int key = 30;
    int last=arr.length-1;
    binarySearch(arr,0,last,key);
}
}
```

Binary Search (cont.)

```
class BinarySearchExample1{
  public static int binarySearch(int arr[], int first, int last, in
t key){
      if (last>=first){
         int mid = first + (last - first)/2;
         if (arr[mid] == key){
         return mid;
         }
         if (arr[mid] > key){
  return binarySearch(arr, first, mid-
1, key);//search in left subarray
         }else{
return binarySearch(arr, mid+1, last, key);//search in right s
ubarray
         }
      }
return -1;  }
```

Binary Search Using Recursion

```
public static void main(String args[]){
      int arr[] = {10,20,30,40,50};
      int key = 30;
      int last=arr.length-1;
      int result = binarySearch(arr,0,last,key);
      if (result == -1)
     System.out.println("Element is not found!");
      else
System.out.println("Element is found at index: "+result);
   }
}
```

Binary Search (cont.)

- Dividing by 2 is equivalent to one comparison in the binary search algorithm.
- Thus, an array of 1,048,575 (220 – 1) elements takes a maximum of 20 comparisons to find the key, and an array of approximately one billion elements takes a maximum of 30 comparisons to find the key.
- This is a tremendous performance improvement over the linear search.
- For a one-billion-element array, this is a difference between an average of 500 million comparisons for the linear search and a maximum of only 30 comparisons for the binary search!
- The maximum number of comparisons needed for the binary search of any sorted array is the exponent of the first power of 2 greater than the number of elements in the array, which is represented as log2 n.

Binary Search (cont.)

- *All logarithms grow at roughly the same rate*, so in Big O notation the base can be omitted.
- This results in a Big O of O(log *n*) for a binary search, which is also known as logarithmic runtime and pronounced "on the order of log n" or more simply "order log *n*."

Sorting Algorithms

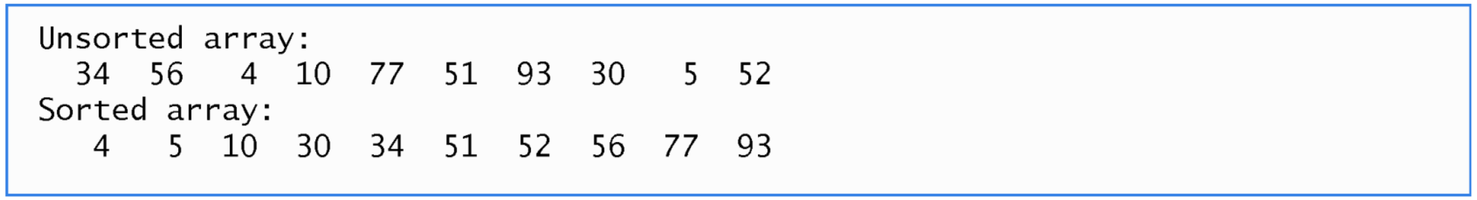
- Sorting data (i.e., placing the data into some particular order, such as ascending or descending) is one of the most important computing applications.
- Your algorithm choice affects only the algorithm’s runtime and memory use.
- The next two sections introduce the selection sort and insertion sort—simple algorithms to implement, but not efficient.
- In each case, we examine the efficiency of the algorithms using Big O notation.
- We then present the merge sort algorithm, which is much faster but is more difficult to implement.

Insertion Sort

```
void insertionSort(int [] items,int size)
{
for(int next=1;i<size-1;++next)
{
int insert=items[next]; // save value of next item to insert
int moveIndex=next; //initialize location to place element
//search for the location in which to put the current element
while((moveIndex > 0) && (items[moveIndex -1] > insert))
{
items[moveIndex] = items[moveIndex-1];
--moveIndex;
}
items[moveIndex=insert;
}//end of for
}//end of function insetionSort
```

Insertion Sort

```
class Mainclass
{
static void main()
{
final int sz=10;
int data[]={34,56,4,10,77,51,93,30,5,52};
system.out.println("Unsorted Array");
for(int val:data)
system.out.print(val+"  ");
insertionSort(data,sz);
system.out.println("Sorted Array");
for(int val:data)
system.out.print(val+"  ");
}
}
```



```
Unsorted array:
  34  56   4  10  77  51  93  30   5  52
Sorted array:
   4   5  10  30  34  51  52  56  77  93
```

Insertion Sort

Insertion Sort Algorithm

- The algorithm’s first iteration takes the array’s second element and, if it’s less than the first element, swaps it with the first element (i.e., the algorithm inserts the second element in front of the first element).
- The second iteration looks at the third element and inserts it into the correct position with respect to the first two elements, so all three elements are in order.
- At the ith iteration of this algorithm, the first i elements in the original array will be sorted.

Insertion Sort

First Iteration

- Lines 33–34 declare and initialize the array named data with the following values:

  34 56 4 10 77 51 93 30 5 52

- Line 42 passes the array to the insertionSort function, which receives the array in parameter items.
- The function first looks at items[0] and items[1], whose values are 34 and 56, respectively.
- These two elements are already in order, so the algorithm continues—if they were out of order, the algorithm would swap them.

Insertion Sort

Second Iteration

- In the second iteration, the algorithm looks at the value of items[2] (that is, 4).
- This value is less than 56, so the algorithm stores 4 in a temporary variable and moves 56 one element to the right.
- The algorithm then determines that 4 is less than 34, so it moves 34 one element to the right.
- At this point, the algorithm has reached the beginning of the array, so it places 4 in items[0].
- The array now is

  4   34   56   10   77   51   93   30   5   52

Insertion Sort

Third Iteration and Beyond

- In the third iteration, the algorithm places the value of items[3] (that is, 10) in the correct location with respect to the first four array elements.
- The algorithm compares 10 to 56 and moves 56 one element to the right because it’s larger than 10.
- Next, the algorithm compares 10 to 34, moving 34 right one element.
- When the algorithm compares 10 to 4, it observes that 10 is larger than 4 and places 10 in items[1].
- The array now is

  4 10 34 56 77 51 93 30 5 52
- Using this algorithm, after the ith iteration, the first i + 1 array elements are sorted.
- They may not be in their final locations, however, because the algorithm might encounter smaller values later in the array.

Insertion Sort

Function Template insertionSort

- Function template insertionSort performs the sorting in lines 13–27, which iterates over the array’s elements.
- In each iteration, line 15 temporarily stores in variable insert the value of the element that will be inserted into the array’s sorted portion.
- Line 16 declares and initializes the variable moveIndex, which keeps track of where to insert the element.
- Lines 19–24 loop to locate the correct position where the element should be inserted.
- The loop terminates either when the program reaches the array’s first element or when it reaches an element that’s less than the value to insert.
- Line 22 moves an element to the right, and line 23 decrements the position at which to insert the next element.
- After the while loop ends, line 26 inserts the element into place.
- When the for statement in lines 13–27 terminates, the array’s elements are sorted.

Insertion Sort

Big O: Efficiency of Insertion Sort

- Insertion sort is simple, but inefficient, sorting algorithm.
- This becomes apparent when sorting large arrays.
- Insertion sort iterates n – 1 times, inserting an element into the appropriate position in the elements sorted so far.
- For each iteration, determining where to insert the element can require comparing the element to each of the preceding elements—n – 1 comparisons in the worst case.
- Each individual repetition statement runs in O(n) time.
- To determine Big O notation, nested statements mean that you must multiply the number of comparisons.
- For each iteration of an outer loop, there will be a certain number of iterations of the inner loop. In this algorithm, for each O(n) iteration of the outer loop, there will be O(n) iterations of the inner loop, resulting in a Big O of O(n * n) or O(n2).

Insertion Sort

- Figure uses the selection sort algorithm—another easy-to-implement, but inefficient, sorting algorithm—to sort a 10-element array’s values into ascending order.
- Function template selectionSort (lines 9–27) implements the algorithm.

Selection Sort

```
void selectionSort(int [] items,int size)
{
for(int i=0;i<size-1;++i)
{
int indexOfSmallest=i;
for(int index=i+1;index<size;++index)
{
if(items[index]
<items[indexOfSmallest])
indexOfSmallest=index;
}
//swap the elements at position i and
indexOfSmallest
int hold=items[i];
items[i]=items[indexOfSmallest];
items[indexOfSmallest]=hold;
}}
```

```
class Mainclass
{
public static void main()
{
final int sz=10;
int data[]={34,56,4,10,77,51,93,30,5,52};
system.out.println("Unsorted Array");
for(int val:data)
system.out.print(val+"  ");
selectionSort(data,sz);
system.out.println("Sorted Array");
for(int val:data)
system.out.print(val+"  ");
}
}
```

Selection Sort

Selection Sort Algorithm

- The algorithm’s first iteration of the algorithm selects the smallest element value and swaps it with the first element’s value.
- The second iteration selects the second-smallest element value (which is the smallest of the remaining elements) and swaps it with the second element’s value.
- The algorithm continues until the last iteration selects the second-largest element and swaps it with the second-to-last element’s value, leaving the largest value in the last element.
- After the ith iteration, the smallest i values will be sorted into increasing order in the first array elements.

Selection Sort (cont.)

First Iteration

- Lines 32–33 declare and initialize the array named data with the following values:
```
  34 56   4   10   77   51   93   30   5   52
```

- The selection sort first determines the smallest value (4) in the array, which is in element 2.

- The algorithm swaps 4 with the value in element 0 (34), resulting in
```
  4   56   34   10   77   51   93   30   5   52
```

Selection Sort (cont.)

Second Iteration

The algorithm then determines the smallest value of the remaining elements (all elements except 4), which is 5, contained in element 8.

The program swaps the 5 with the 56 in element 1, resulting in

```
    4    5    34    10    77    51    93    30    56    52
```

Selection Sort (cont.)

Third Iteration

- On the third iteration, the program determines the next smallest value, 10, and swaps it with the value in element 2 (34).

```
    4  5    10    34    77    51    93    30    56    52
```

- The process continues until the array is fully sorted.

```
    4  5    10    30    34    51    52    56    77    93
```

- After the first iteration, the smallest element is in the first position; after the second iteration, the two smallest elements are in order in the first two positions and so on.

Selection Sort (cont.)

Function Template ***`selectionSort`***

Function template `selectionSort` performs the sorting in lines 13–26.

Selection Sort (cont.)

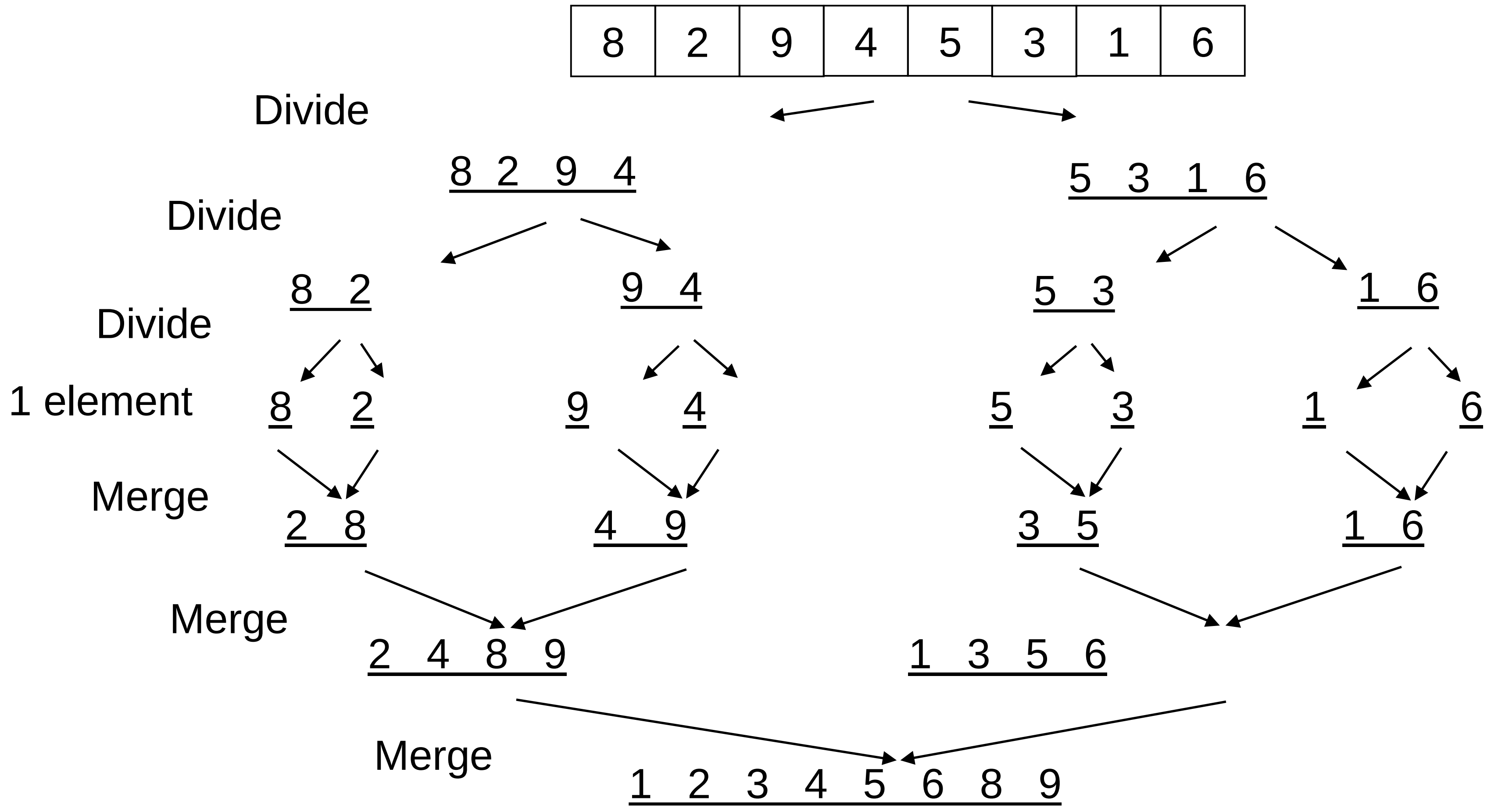
- The selection sort algorithm iterates n – 1 times, each time swapping the smallest remaining element into its sorted position.
- Locating the smallest remaining element requires n – 1 comparisons during the first iteration, n – 2 during the second iteration, then n – 3, … , 3, 2, 1.
- This results in a total of n(n – 1)/2 or (n2 – n)/2 comparisons.
- In Big O notation, smaller terms drop out and constants are ignored, leaving a Big O of O(n2).

Merge Sort

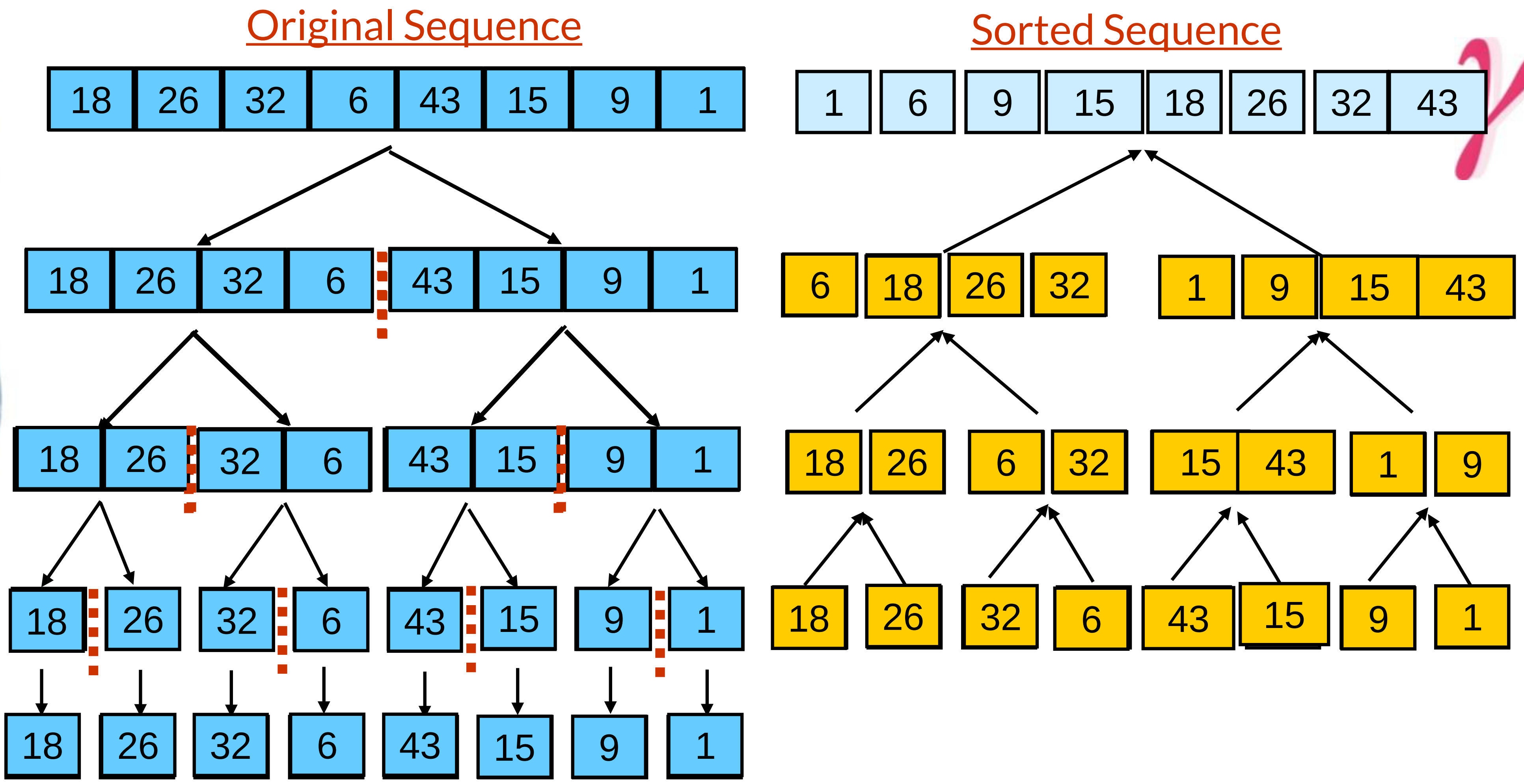
Divide And Conquer

- Merging two lists of one element each is the same as sorting them.
- Merge sort divides up an unsorted list until the above condition is met and then sorts the divided parts back together in pairs.
- Specifically this can be done by recursively dividing the unsorted list in half, merge sorting the left side then the right side and then merging the left and right back together.

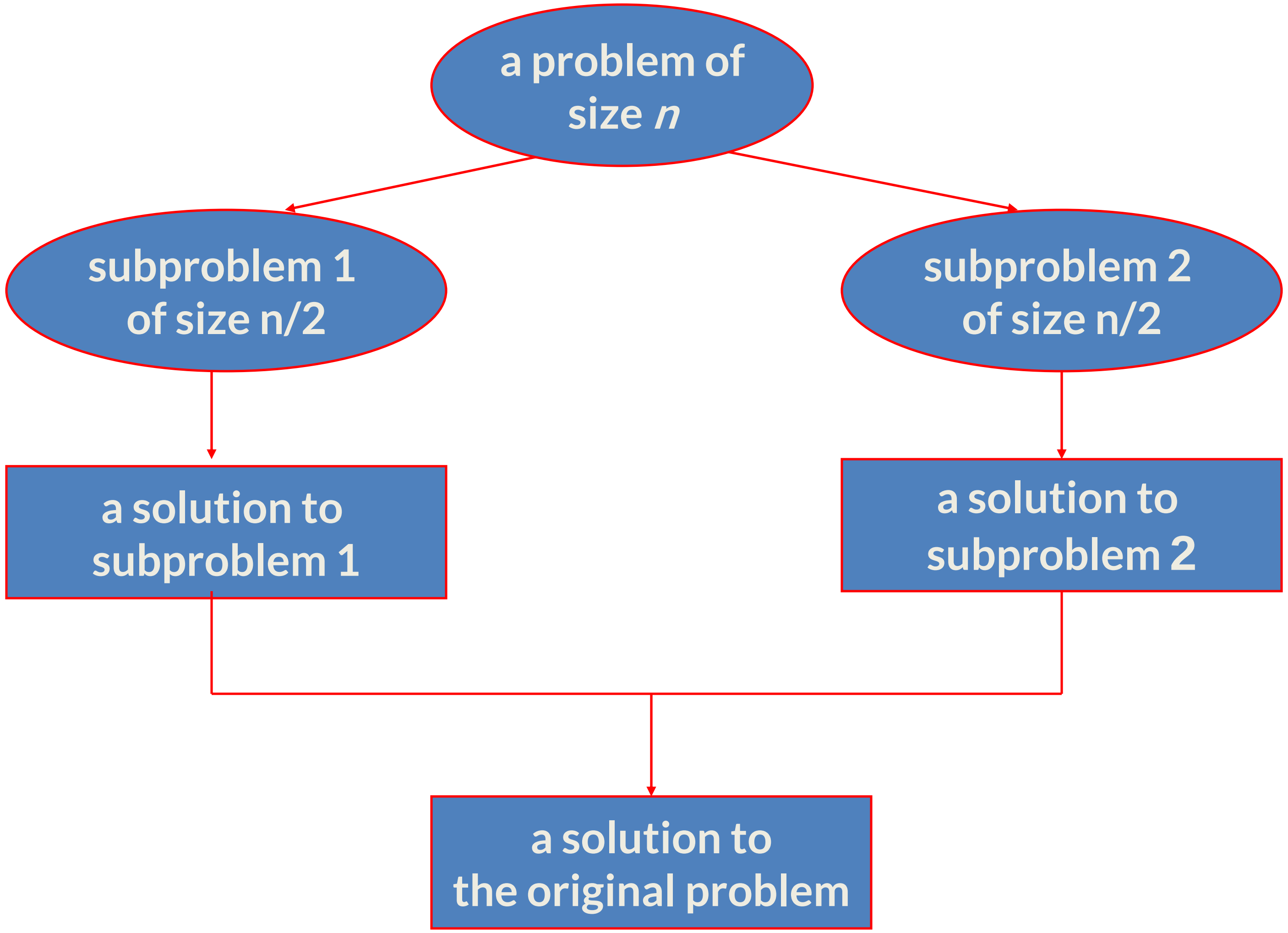
Mergesort Example



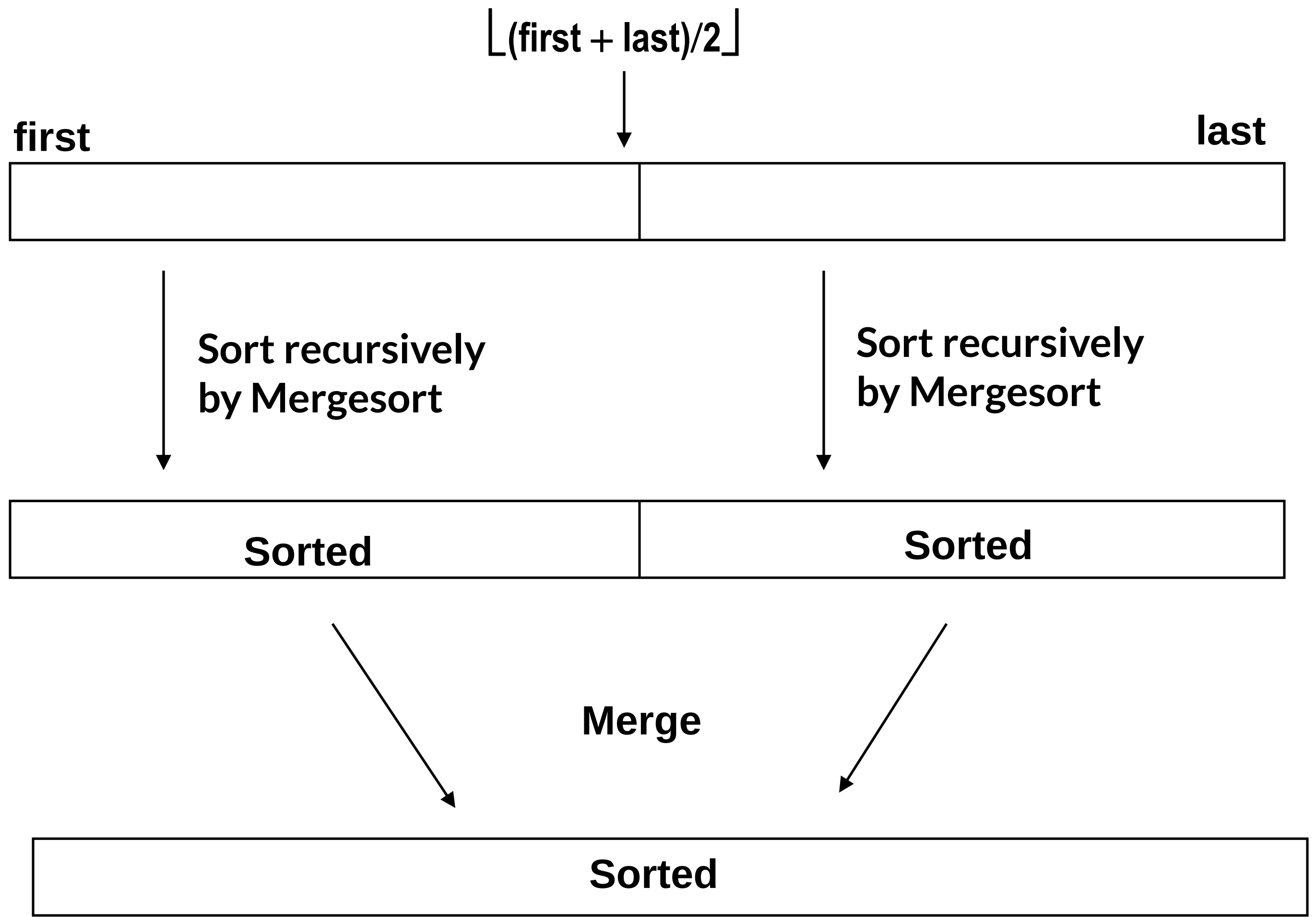
Merge Sort – Example



Divide-and-conquer Technique


a problem of size *n*
subproblem 1 of size n/2
subproblem 2 of size n/2
a solution to subproblem 1
a solution to subproblem 2
a solution to the original problem

Using Divide and Conquer: Mergesort strategy



Merge Sort Algorithm

Divide: Partition ‘n’ elements array into two sub lists with n/2 elements each

Conquer: Sort sub list1 and sub list2

Combine: Merge sub list1 and sub list2

Merge Sort Example

99	6	86	15	58	35	86	4	0

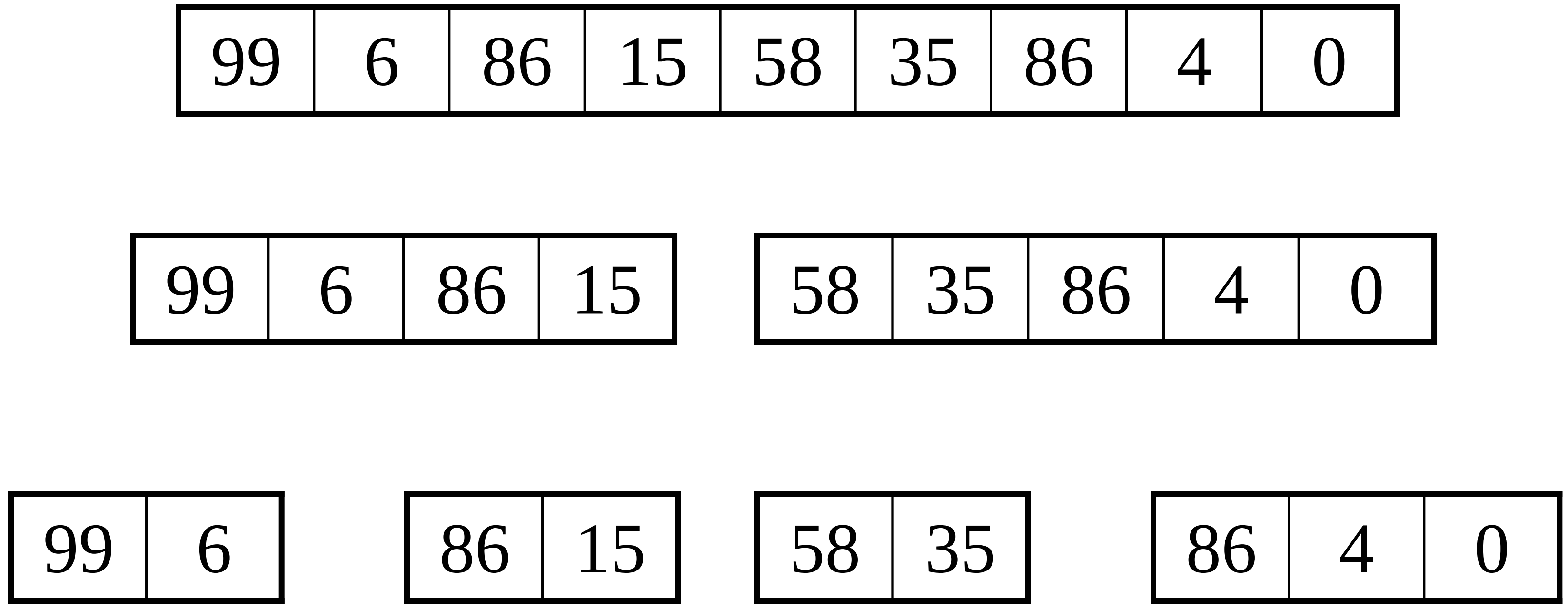
Merge Sort Example

99	6	86	15	58	35	86	4	0

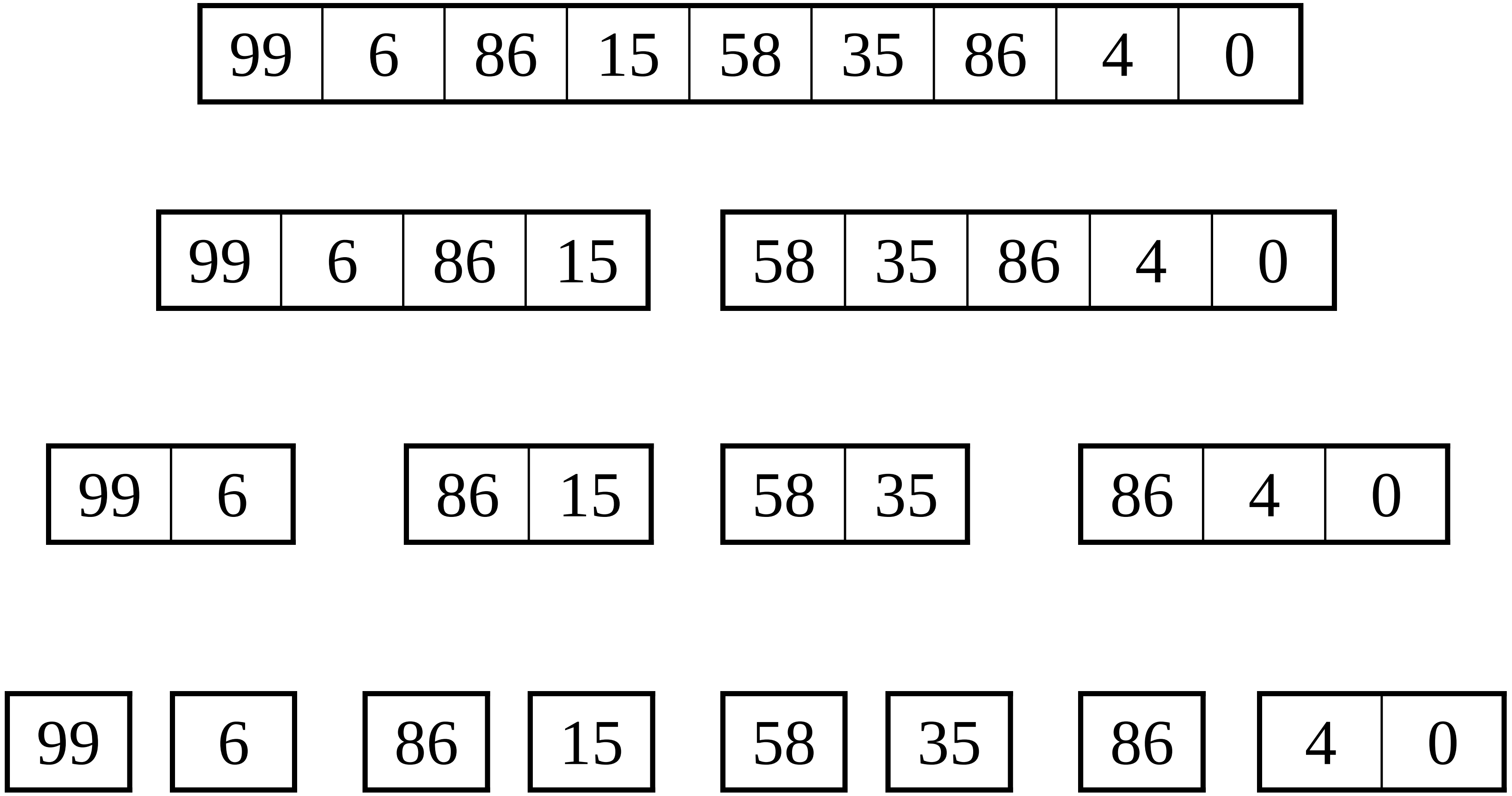
99	6	86	15

58	35	86	4	0

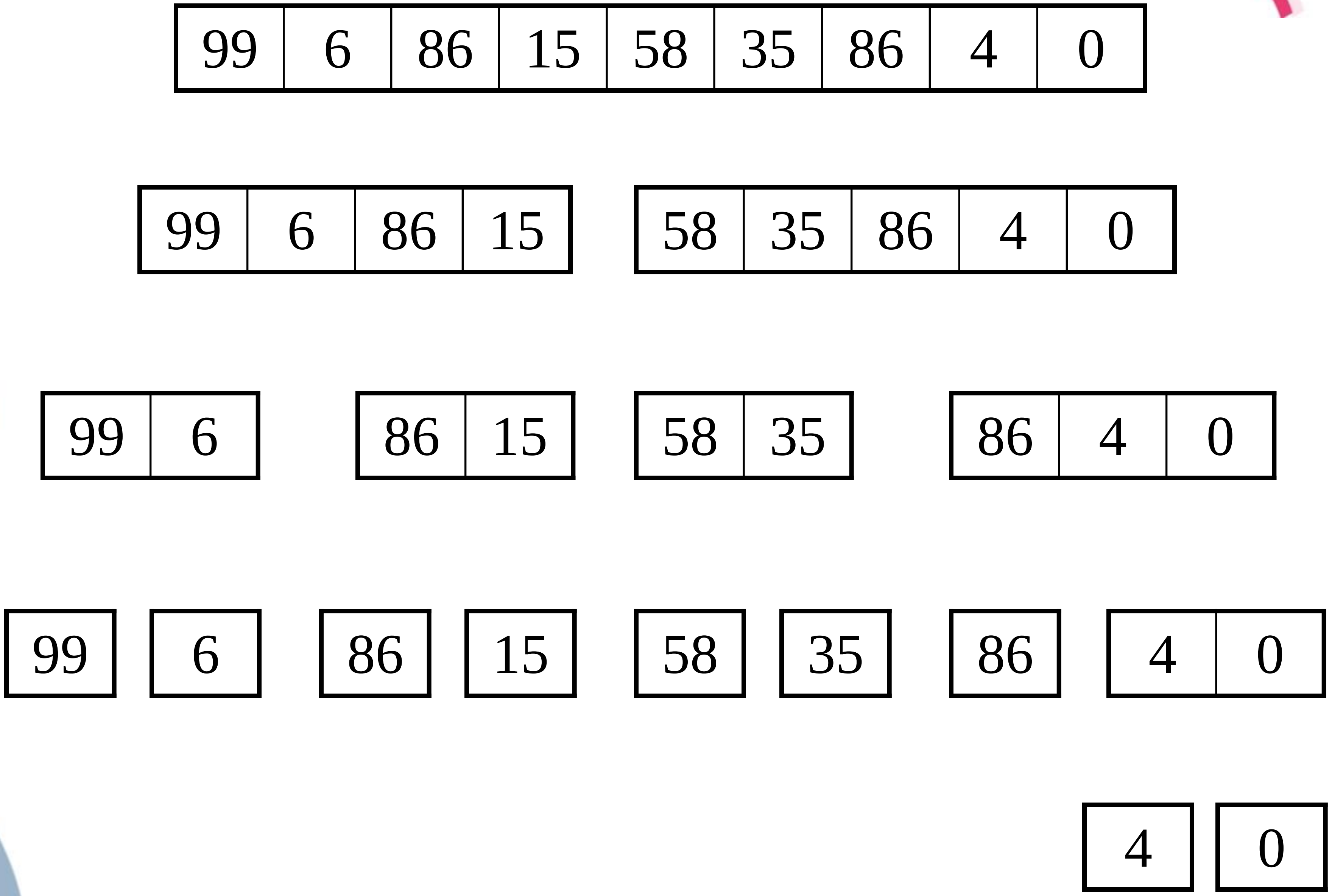
Merge Sort Example



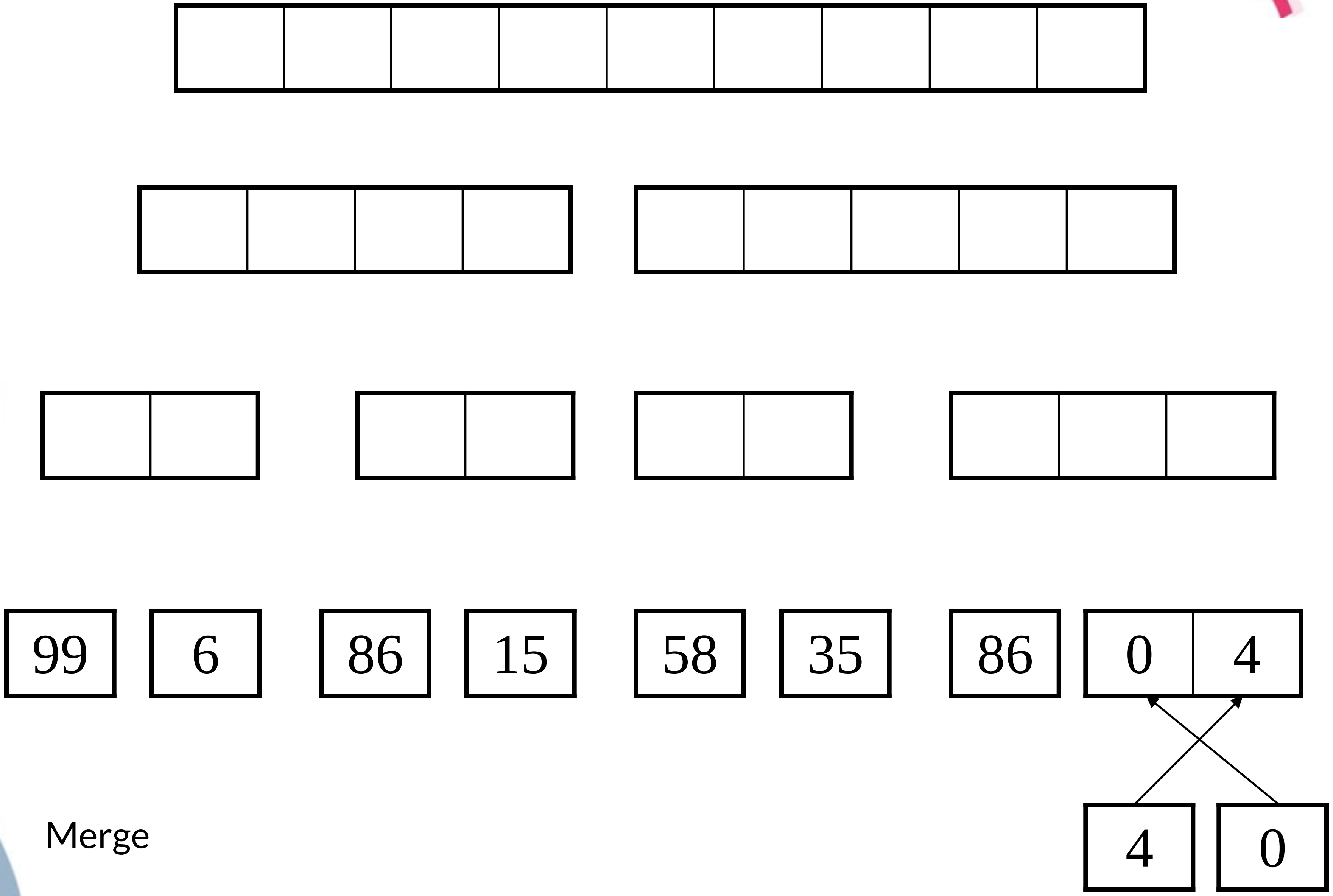
Merge Sort Example



Merge Sort Example

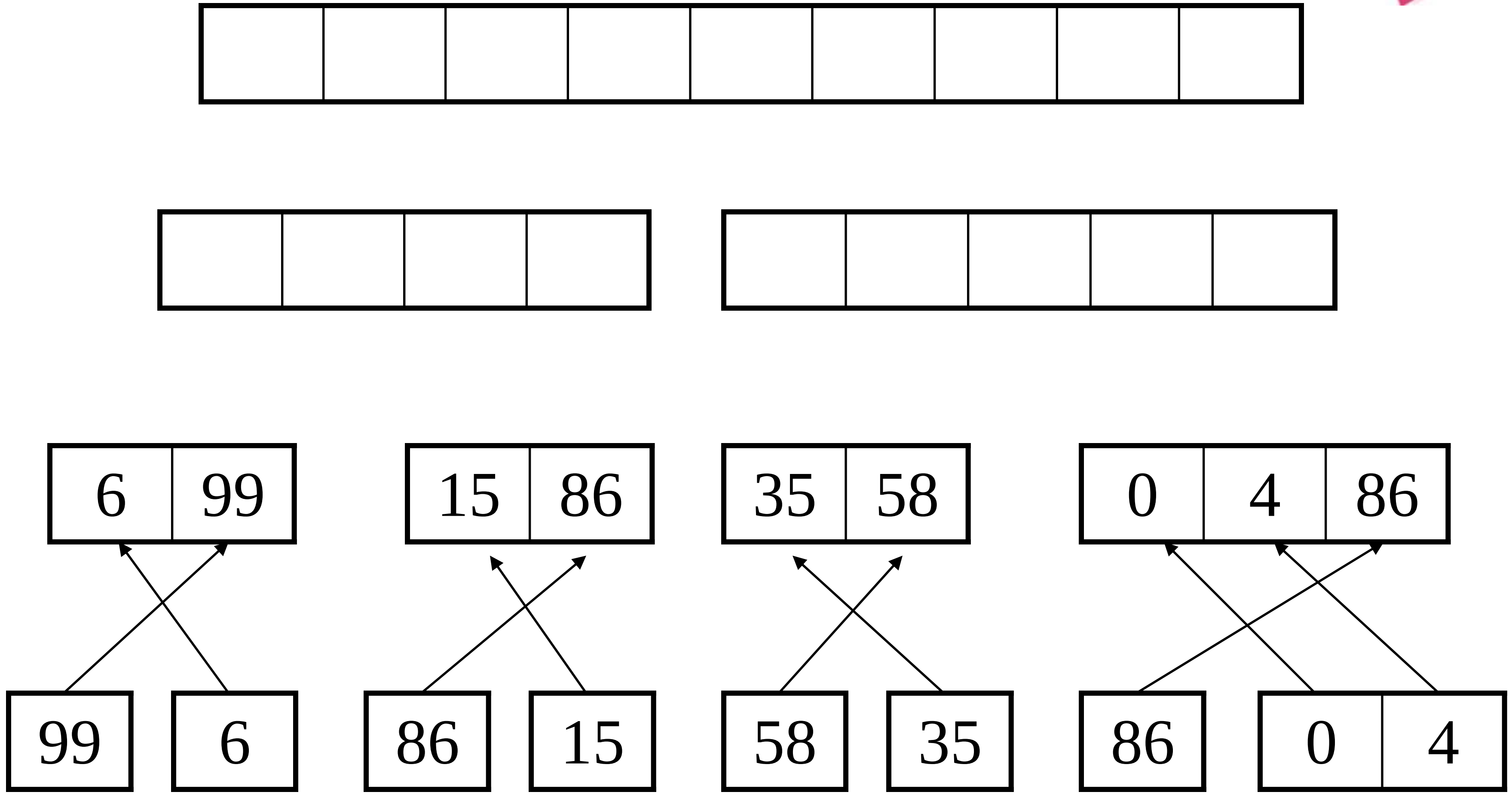


Merge Sort Example



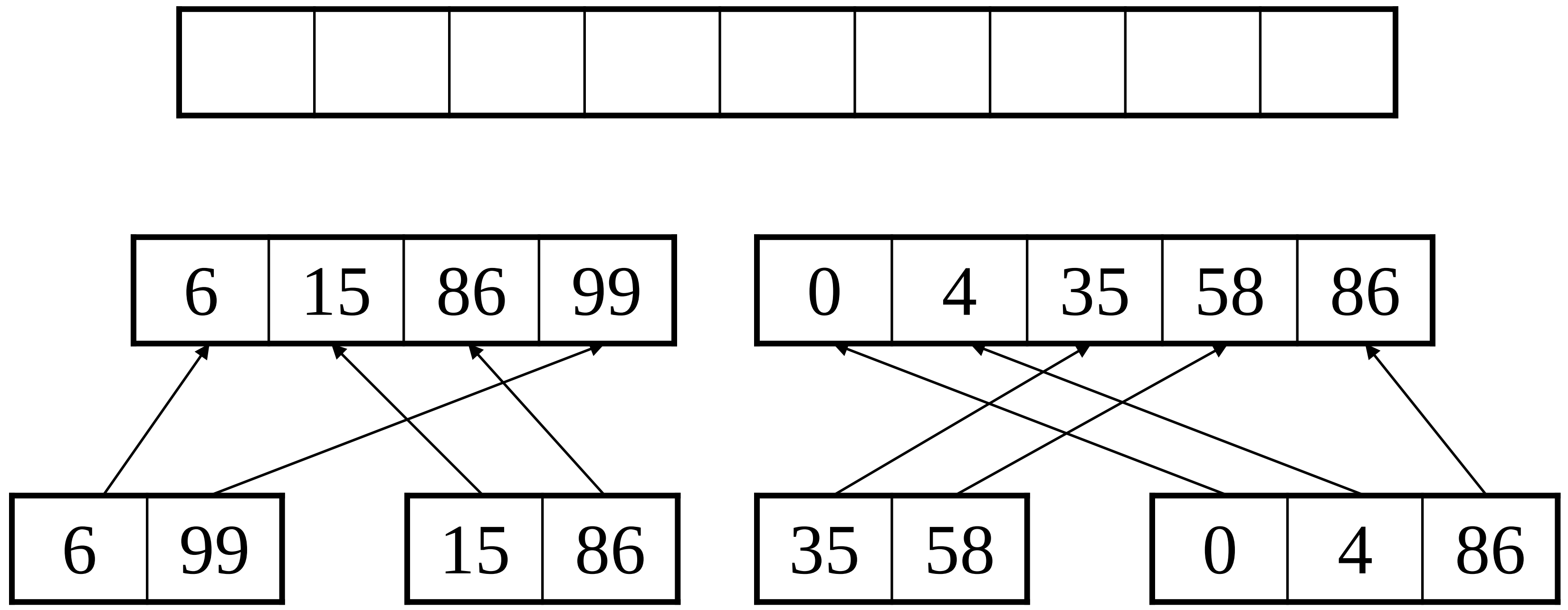
Merge

Merge Sort Example



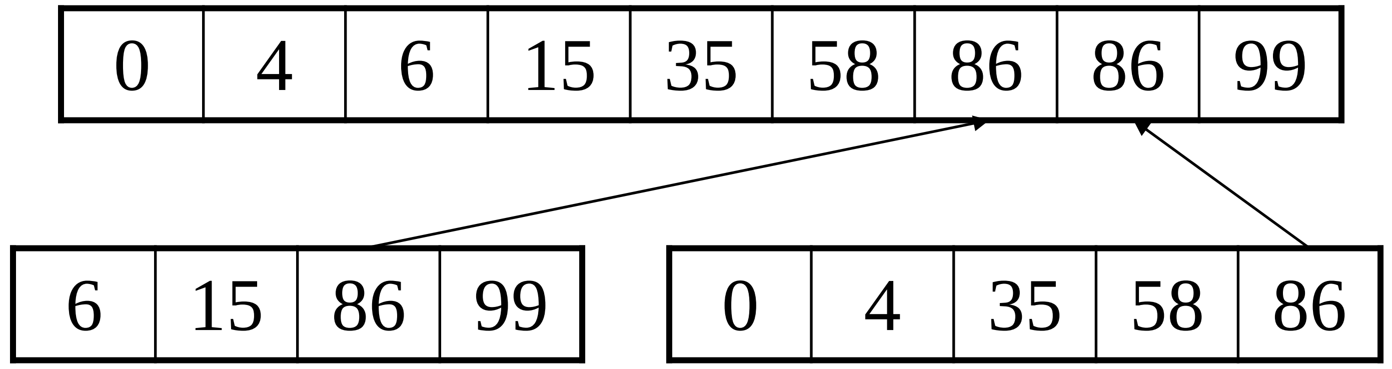
Merge

Merge Sort Example



Merge

Merge Sort Example



Merge

Merge Sort Example

0	4	6	15	35	58	86	86	99

Program

```
class MS
{

                        private int A[10];
                        public int n;
                        public int low, high;
            public void get_data()
            {
             println("Enter the length of the list");
            n=Scanner.nextInt();
             println(" Enter the list elements: ");
            for(int i = 0; i<n; i++)
            {
            A[i]=Scanner.nextInt();
            }
}

            public  void Combine(int low, int mid, int
high);
            public void Display();
```

```
}

void Mergesort(int low, int high)
{
            int mid;
            if(low<high)
            {
                        mid = (low+high)/2;             //split the list at
mid
                        Mergesort(low, mid);            //first sublist
                        Mergesort(mid+1, high);         //second sublist
                        Combine(low, mid, high);        //merging two
sublists
            }
}
```

Program (Contd.)

```
void MS::Combine(int low, int mid, int high)
{
        int i,j,k;
        int temp[10];
        i = low;
        j = mid+1;
        k = low;
        while(i <= mid && j <= high)
        {
                if(A[i] <= A[j])
                {
                        temp[k] = A[i];
                        i++;
                        k++;
                }
                else
                {
                        temp[k] = A[j];
                        j++;
                        k++;
                }
        }
        while(i<=mid)
        {
                temp[k] = A[i];
                i++;
                k++;
        }
        while(j<=high)
        {
                temp[k] = A[j];
                j++;
                k++;
        }
        for(k = low; k <= high; k++)
        {
                A[k] = temp[k];
        }
}
```

Program (Contd.)

```
void Display()
{
            println(" The sorted array
is:\n“);
            for(int i = 0; i<n; i++)
            {
                        println(A[i]);
            }
}
```

```
Public static int main()
{
            MS ms=new MS();
            ms.get_data();
            ms.low=0;
            ms.high=(obj.n)-1;
            ms.Mergesort(obj.low, obj.high);
            ms.Display();
}
```

Time Complexity Analysis

Worst case: O($nlog_2n$)

Average case: O($nlog_2n$)

Best case: O($nlog_2n$)

Quicksort

Introduction

Fastest known sorting algorithm in practice

Average case: O(N log N) (we don’t prove it)

Worst case: O(N^2)

But, the worst case seldom happens.

Another divide-and-conquer recursive algorithm, like mergesort

Quicksort

Divide step:

Pick any element (***pivot***) v in S

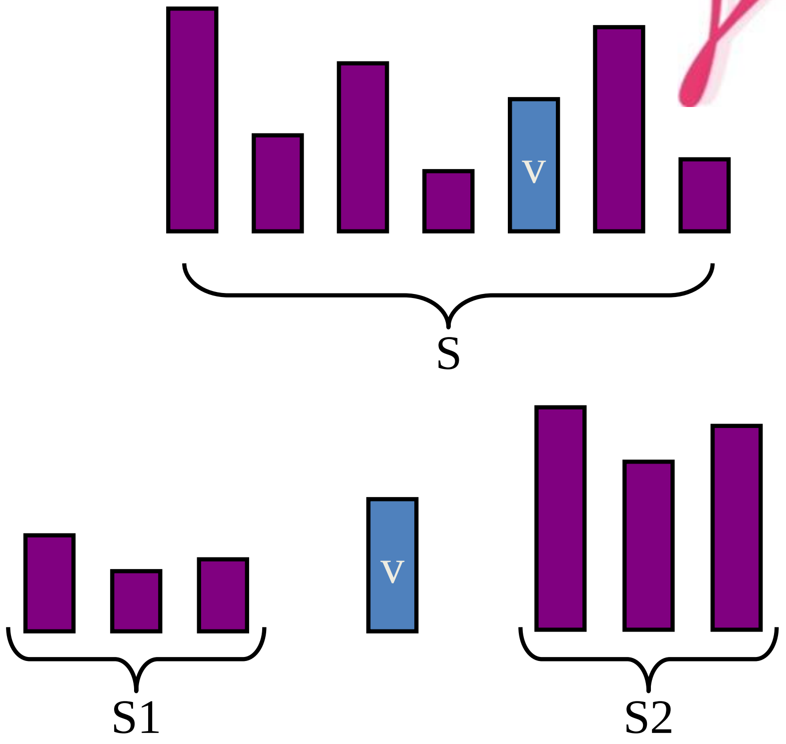
Partition S – {v} into two disjoint groups

$S1 = \{x \in S - \{v\} \mid x <= v\}$

$S2 = \{x \in S - \{v\} \mid x \geq v\}$

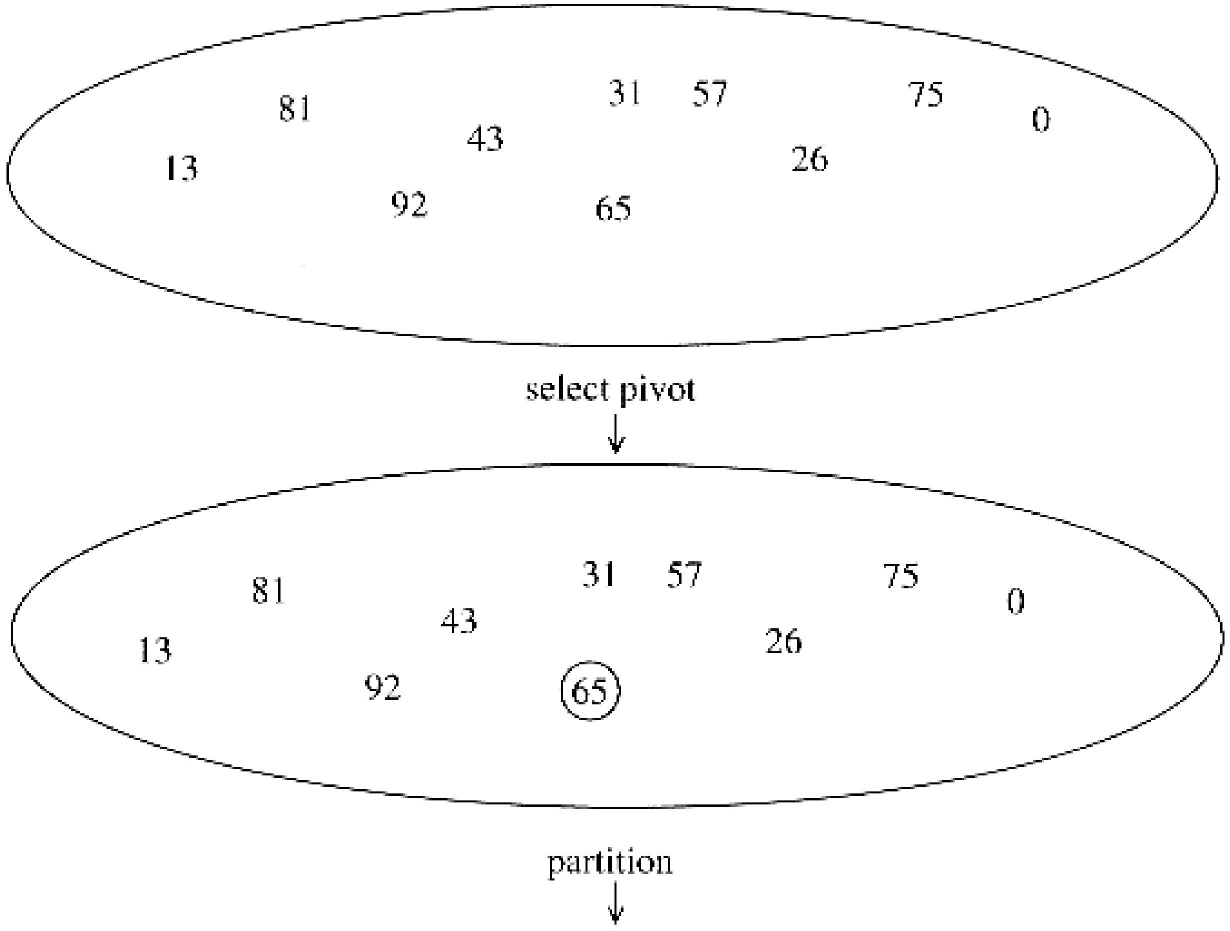
Conquer step: recursively sort S1 and S2

Combine step: the sorted S1 (by the time returned from recursion), followed by v, followed by the sorted S2 (i.e., nothing extra needs to be done)

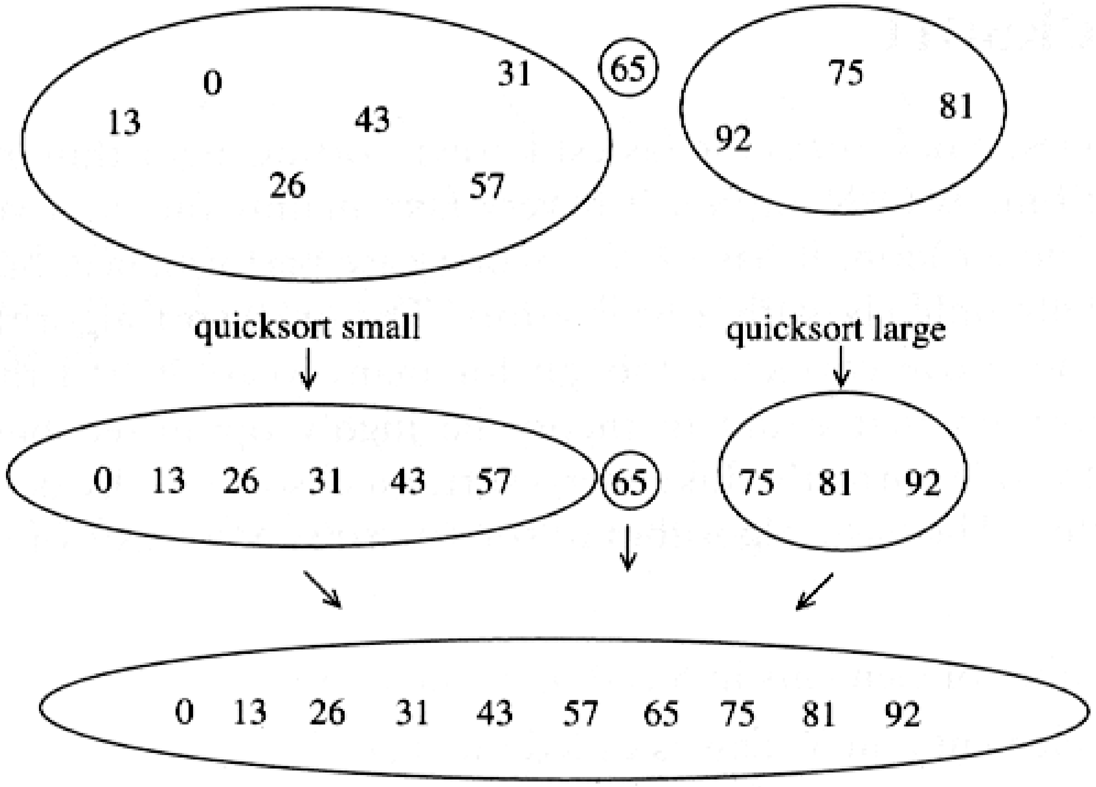


To simplify, we may assume that we don't have repetitive elements,
So to ignore the 'equality' case!

Example



81
31
57
75
0
13
43
26
92
65
select pivot
81
31
57
75
0
13
43
26
92
65
partition



0
31
65
75
13
43
81
92
26
57
quicksort small
quicksort large
0 13 26 31 43 57
65
75 81 92
0 13 26 31 43 57 65 75 81 92

Pseudo-code

```
Input: an array a[left, right]

QuickSort (a, left, right) {
        if (left < right) {
                pivot = Partition (a, left, right)
                Quicksort (a, left, pivot-1)
                Quicksort (a, pivot+1, right)
        }
}
```

Compare with MergeSort:

```
MergeSort (a, left, right) {
     if (left < right) {
          mid = divide (a, left, right)
          MergeSort (a, left, mid-1)
          MergeSort (a, mid+1, right)
          merge(a, left, mid+1, right)
     }
}
```

Two key steps

- How to pick a pivot?
- How to partition?

Pick a pivot

- Use the first element as pivot

  if the input is random, ok

  if the input is presorted (or in reverse order)

  all the elements go into S2 (or S1)

  this happens consistently throughout the recursive calls

  Results in O(n2) behavior (Analyze this case later)

- Choose the pivot randomly

  generally safe

  random number generation can be expensive

In-place Partition

If use additional array (not in-place) like MergeSort
- Straightforward to code like MergeSort (write it down!)
- Inefficient!

$\downarrow$

Many ways to implement

Even the slightest deviations may cause surprisingly bad results.
- Not stable as it does not preserve the ordering of the identical keys.
- Hard to write correctly ☹

An easy version of in-place partition to understand, but not the original form

```
int partition(a, left, right, pivotIndex) {
   pivotValue = a[pivotIndex];
   swap(a[pivotIndex], a[right]); // Move pivot to end

   // move all smaller (than pivotValue) to the begining
   storeIndex = left;
   for (i from left to right) {
        if a[i] < pivotValue
                swap(a[storeIndex], a[i]);
                storeIndex = storeIndex + 1 ;
   }

   swap(a[right], a[storeIndex]); // Move pivot to its final place
   return storeIndex;
}
```

Look at Wikipedia

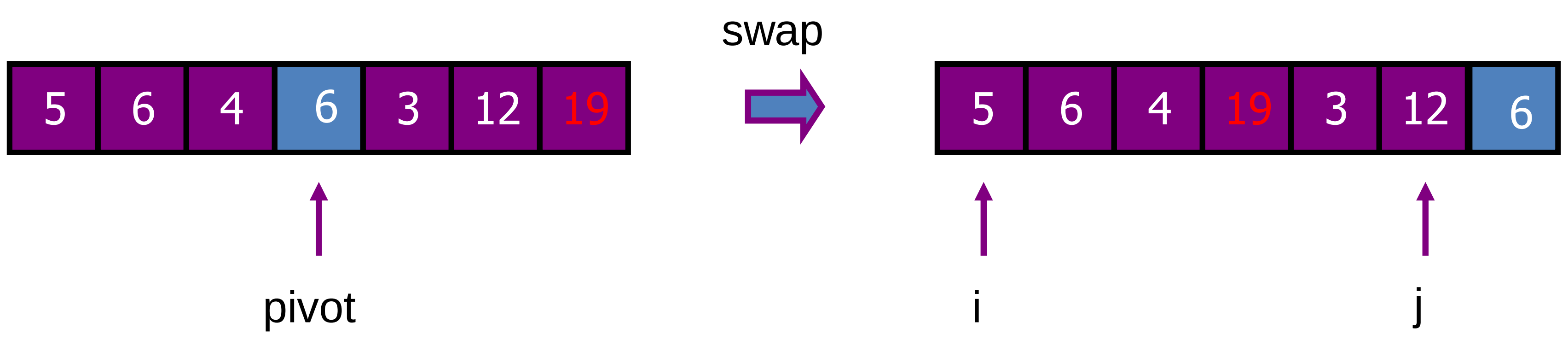
```
quicksort(a,left,right) {

   if (right>left) {
   pivotIndex = left;
   select a pivot value a[pivotIndex];

   pivotNewIndex=partition(a,left,right,pivotIndex);

   quicksort(a,left,pivotNewIndex-1);
   quicksort(a,pivotNewIndex+1,right);
   }
}
```

A better partition

Want to partition an array A[left .. right]

First, get the pivot element out of the way by swapping it with the last element. (Swap pivot and A[right])

Let i start at the first element and j start at the next-to-last element (i = left, j = right – 1)

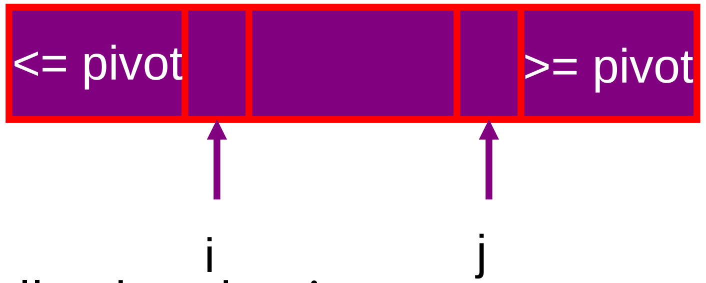


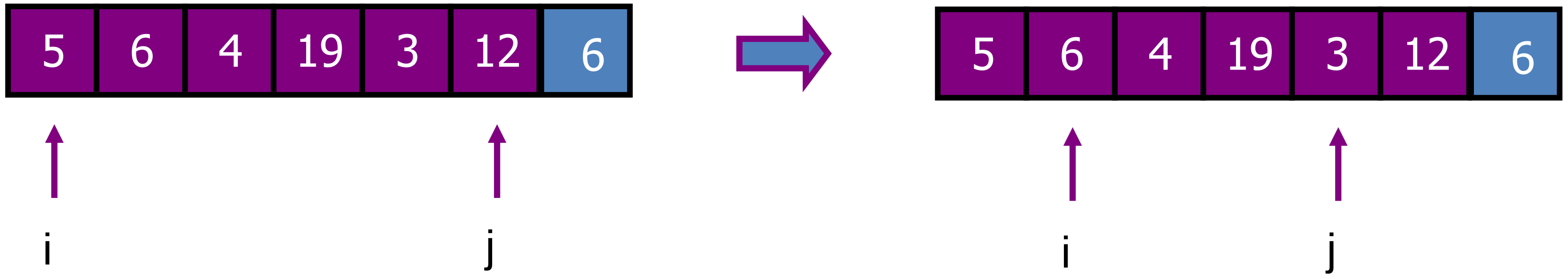
Want to have

- A[x] <= pivot, for x < i
- A[x] >= pivot, for x > j

When i < j

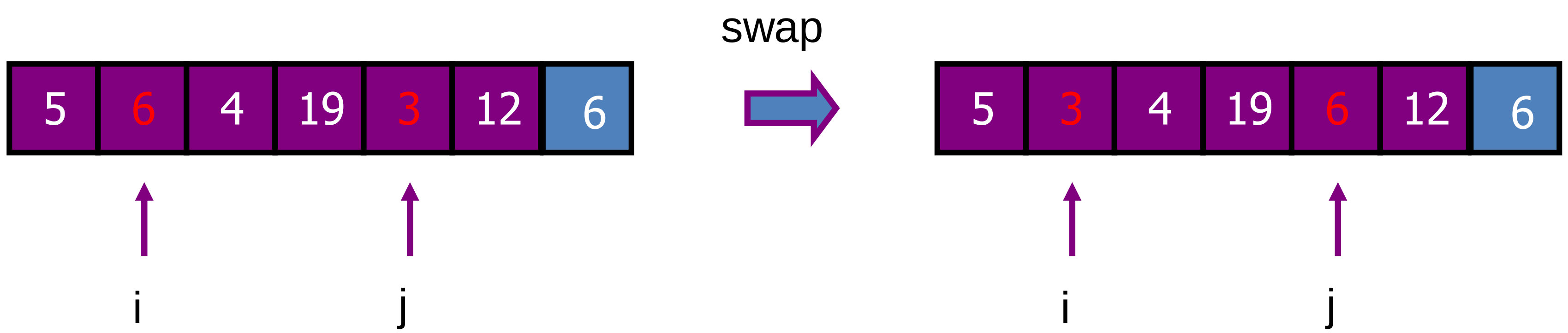
- Move i right, skipping over elements smaller than the pivot
- Move j left, skipping over elements greater than the pivot
- When both i and j have stopped
  - A[i] >= pivot
  - A[j] <= pivot





When i and j have stopped and i is to the left of j

- Swap A[i] and A[j]
  - The large element is pushed to the right and the small element is pushed to the left
- After swapping
  - A[i] <= pivot
  - A[j] >= pivot
- Repeat the process until i and j cross



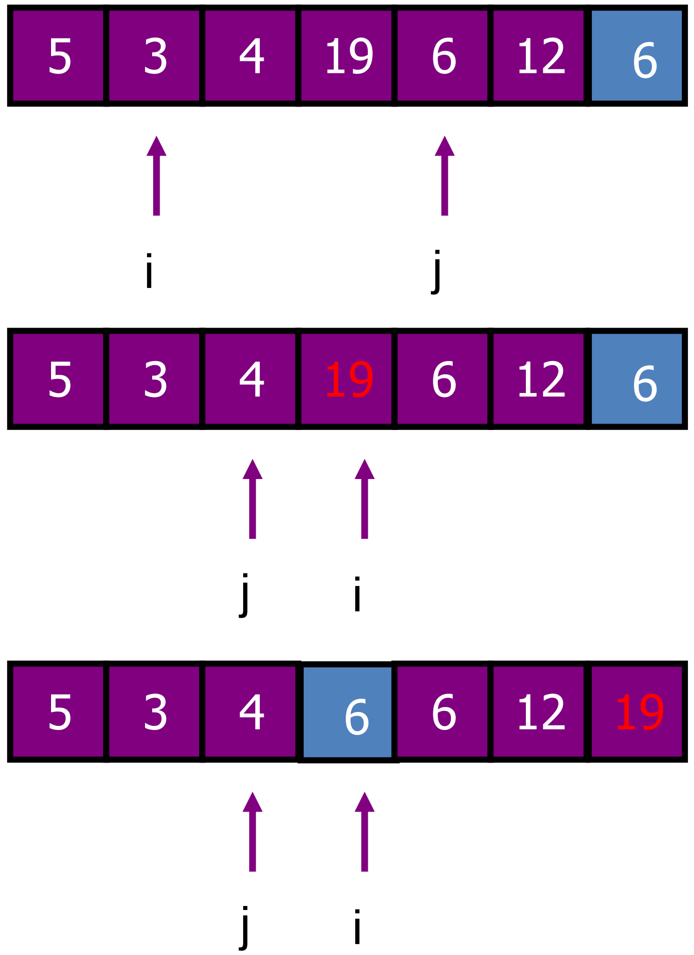
When i and j have crossed

Swap A[i] and pivot

Result:

A[x] <= pivot, for x < i

A[x] >= pivot, for x > i



Implementation (put the pivot on the leftmost instead of rightmost)

```
void quickSort(int array[], int start, int end)
{
    int i = start; // index of left-to-right scan
    int k = end; // index of right-to-left scan

    if (end - start >= 1) // check that there are at least two elements to sort
    {
          int pivot = array[start]; // set the pivot as the first element in the partition
          while (k > i) // while the scan indices from left and right have not met,
          {
                    while (array[i] <= pivot && i <= end && k > i)        // from the left, look for the first
                              i++;                                        // element greater than the pivot
                    while (array[k] > pivot && k >= start && k >= i) // from the right, look for the first
                              k--;                                        // element not greater than the pivot
                    if (k > i)                                            // if the left seekindex is still smaller than
                              swap(array, i, k);                          // the right index,
                                                                          // swap the corresponding elements
          }
          swap(array, start, k);                                          // after the indices have crossed,
                                                                          // swap the last element in
                                                                          // the left partition with the pivot
          quickSort(array, start, k - 1);                      // quicksort the left partition
          quickSort(array, k + 1, end);                        // quicksort the right partition
    }
    else // if there is only one element in the partition, do not do any sorting
    {
     return; // the array is sorted, so exit
    }
}
```

Adapted from http://www.mycsresource.net/articles/programming/sorting_algos/quicksort/

```
void quickSort(int array[])
// pre: array is full, all elements are non-null integers
// post: the array is sorted in ascending order
{
   quickSort(array, 0, array.length - 1); // quicksort all the elements in the array
}
void quickSort(int array[], int start, int end)
{
…
}
void swap(int array[], int index1, int index2) {…}
// pre: array is full and index1, index2 < array.length
// post: the values at indices 1 and 2 have been swapped
```

With duplicate elements …

Partitioning so far defined is ambiguous for duplicate elements (the equality is included for both sets)

Its ‘randomness’ makes a ‘balanced’ distribution of duplicate elements

When all elements are identical:

- both i and j stop $\rightarrow$ many swaps
- but cross in the middle, partition is balanced (so it’s n log n)

A better Pivot

Use the median of the array

- Partitioning always cuts the array into roughly half
- An optimal quicksort (O(N log N))
- However, hard to find the exact median (chicken-egg?)
  - e.g., sort an array to pick the value in the middle
- Approximation to the exact median: …

Median of three

We will use median of three

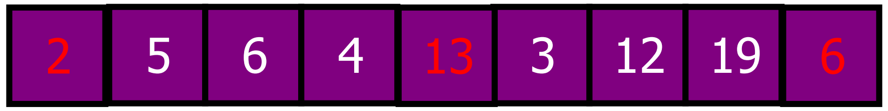
- Compare just three elements: the leftmost, rightmost and center
- Swap these elements if necessary so that

A[left]	=	Smallest
A[right]	=	Largest
A[center]	=	Median of three

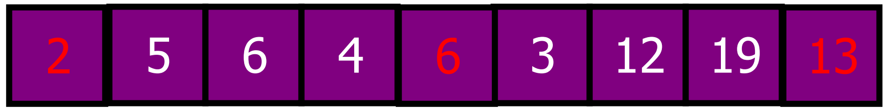
- Pick A[center] as the pivot
- Swap A[center] and A[right – 1] so that pivot is at second last position (why?)

median3

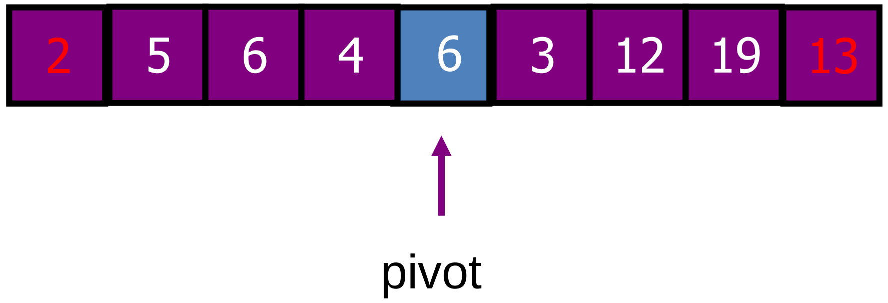
```
int center = ( left + right ) / 2;
if( a[ center ] < a[ left ] )
    swap( a[ left ], a[ center ] );
if( a[ right ] < a[ left ] )
    swap( a[ left ], a[ right ] );
if( a[ right ] < a[ center ] )
    swap( a[ center ], a[ right ] );

    // Place pivot at position right - 1
swap( a[ center ], a[ right - 1 ] );
```



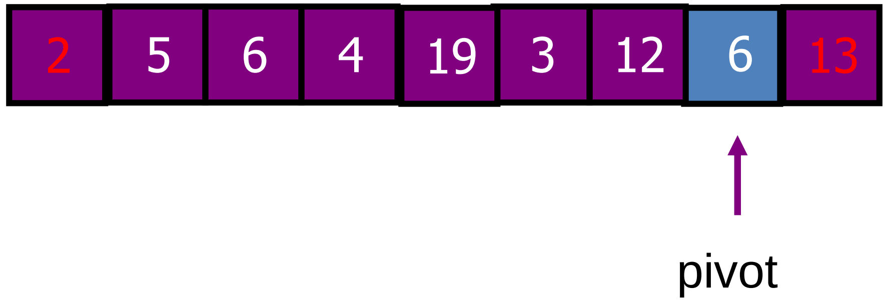
A[left] = 2, A[center] = 13, A[right] = 6



Swap A[center] and A[right]



Choose A[center] as pivot



Swap pivot and A[right – 1]

Note we only need to partition A[left + 1, …, right – 2]. Why?

Works only if pivot is picked as median-of-three.

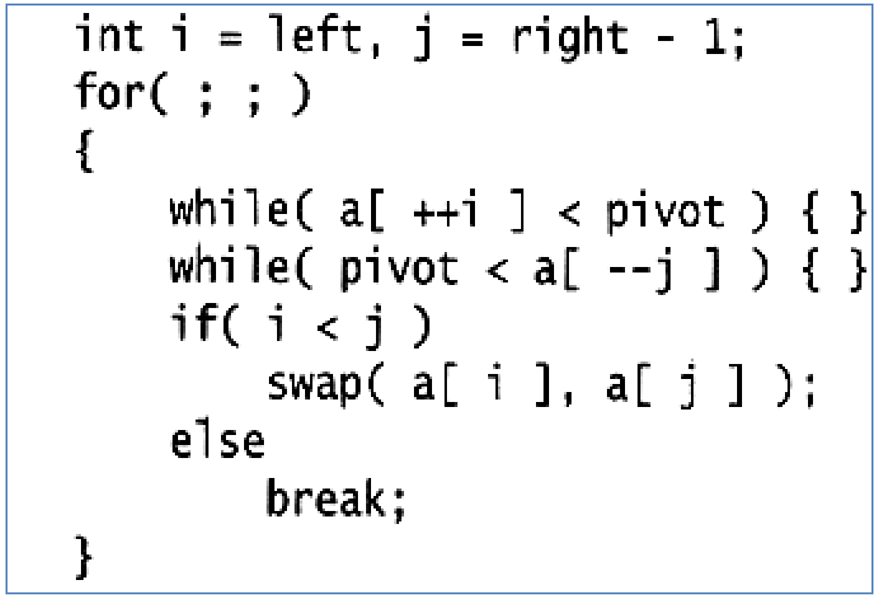
- A[left] <= pivot and A[right] >= pivot
- Thus, only need to partition A[left + 1, …, right – 2]

j will not run past the beginning

- because a[left] <= pivot

i will not run past the end

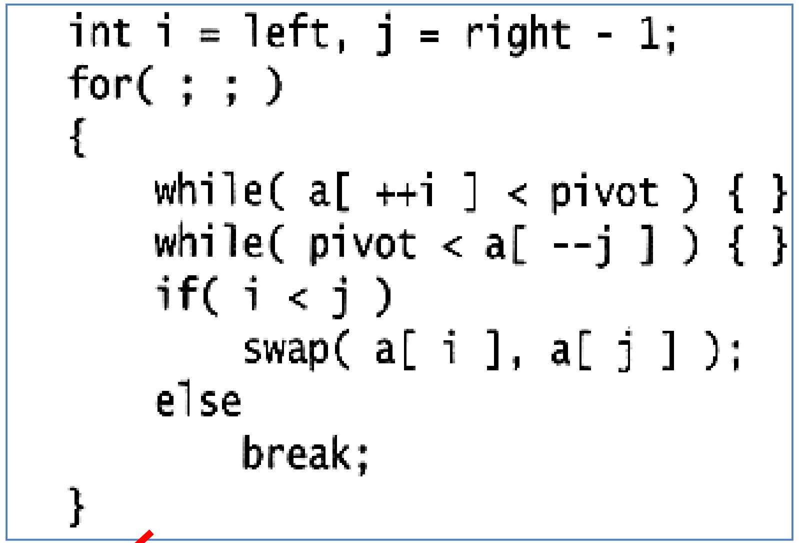
- because a[right-1] = pivot



```
int i = left, j = right - 1;
for( ; ; )
{
    while( a[ ++i ] < pivot ) { }
    while( pivot < a[ --j ] ) { }
    if( i < j )
        swap( a[ i ], a[ j ] );
    else
        break;
}
```

The coding style is efficient, but hard to read

```
i=left;
j=right-1;

while (1) {
   do i=i+1;
    while (a[i] < pivot);

   do j=j-1;
   while (pivot < a[j]);

   if (i<j) swap(a[i],a[j]);
   else break;
}
```



```
int i = left, j = right - 1;
for( ; ; )
{
    while( a[ ++i ] < pivot ) { }
    while( pivot < a[ --j ] ) { }
    if( i < j )
        swap( a[ i ], a[ j ] );
    else
        break;
}
```

Small arrays

For very small arrays, quicksort does not perform as well as insertion sort

- how small depends on many factors, such as the time spent making a recursive call, the compiler, etc

Do not use quicksort recursively for small arrays

- Instead, use a sorting algorithm that is efficient for small arrays, such as insertion sort

A practical implementation

```
if( left + 10 <= right )
{
    Comparable pivot = median3( a, left, right );

        // Begin partitioning
    int i = left, j = right - 1;
    for( ; ; )
    {
        while( a[ ++i ] < pivot ) { }
        while( pivot < a[ --j ] ) { }
        if( i < j )
            swap( a[ i ], a[ j ] );
        else
            break;
    }

    swap( a[ i ], a[ right - 1 ] );  // Restore pivot

    quicksort( a, left, i - 1 );     // Sort small elements
    quicksort( a, i + 1, right );    // Sort large elements
}
else  // Do an insertion sort on the subarray
    insertionSort( a, left, right );
```

Choose pivot

Partitioning

Recursion

For small arrays

Quicksort Analysis

- Assumptions:
  - A random pivot (no median-of-three partitioning)
  - No cutoff for small arrays
- Running time
  - pivot selection: constant time, i.e. O(1)
  - partitioning: linear time, i.e. O(N)
  - running time of the two recursive calls
- $T(N)=T(i)+T(N-i-1)+cN$ where c is a constant
  - i: number of elements in S1

Worst-Case Analysis

- What will be the worst case?
  - The pivot is the smallest element, all the time
  - Partition is always unbalanced

$$\begin{aligned}
T(N) &= T(N-1) + cN \\
T(N-1) &= T(N-2) + c(N-1) \\
T(N-2) &= T(N-3) + c(N-2) \\
&\vdots \\
T(2) &= T(1) + c(2) \\
T(N) &= T(1) + c\sum_{i=2}^{N} i = O(N^2)
\end{aligned}$$

Best-case Analysis

- What will be the best case?

  Partition is perfectly balanced.

  Pivot is always in the middle (median of the array)

$$
\begin{aligned}
T(N) &= 2T(N/2) + cN \\
\frac{T(N)}{N} &= \frac{T(N/2)}{N/2} + c \\
\frac{T(N/2)}{N/2} &= \frac{T(N/4)}{N/4} + c \\
\frac{T(N/4)}{N/4} &= \frac{T(N/8)}{N/8} + c \\
&\vdots \\
\frac{T(2)}{2} &= \frac{T(1)}{1} + c \\
\frac{T(N)}{N} &= \frac{T(1)}{1} + c\log N \\
T(N) &= cN\log N + N = O(N\log N)
\end{aligned}
$$

Average-Case Analysis

- Assume

  Each of the sizes for S1 is equally likely

- This assumption is valid for our pivoting (median-of-three) strategy

- On average, the running time is O(N log N) (covered in comp271)

Quicksort is ‘faster’ than Mergesort

- Both quicksort and mergesort take O(N log N) in the average case.

- Why is quicksort faster than mergesort?

The inner loop consists of an increment/decrement (by 1, which is fast), a test and a jump.

There is no extra juggling as in mergesort.

```
int i = left, j = right - 1;
for( ; ; )
{
    while( a[ ++i ] < pivot ) { }
    while( pivot < a[ --j ] ) { }
    if( i < j )
        swap( a[ i ], a[ j ] );
    else
        break;
}
```

inner loop

Heapsort

Heap data structure

Binary tree

Balanced

Left-justified or Complete

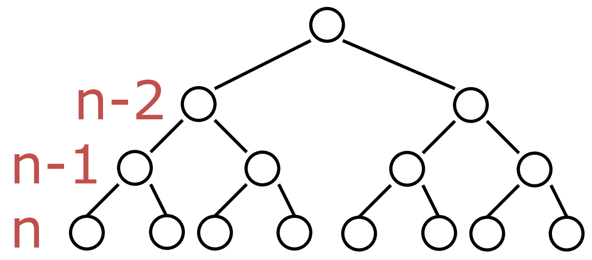
(Max) Heap property: no node has a value greater than the value in its parent

Balanced binary trees

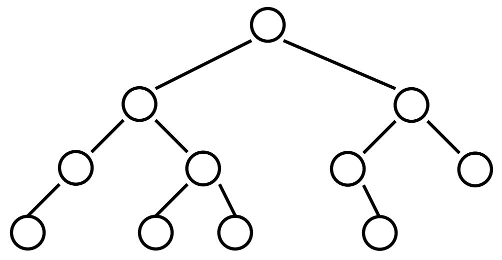
Recall:

- The depth of a node is its distance from the root
- The depth of a tree is the depth of the deepest node

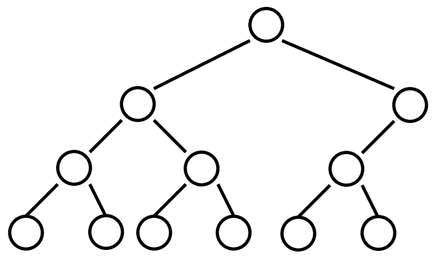
A binary tree of depth n is balanced if all the nodes at depths 0 through n-2 have two children



Balanced



Balanced

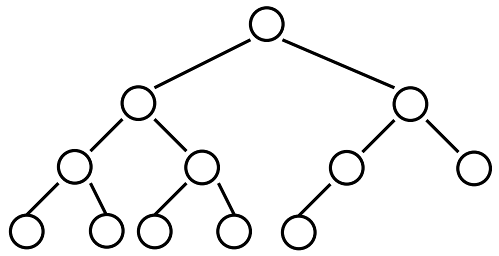


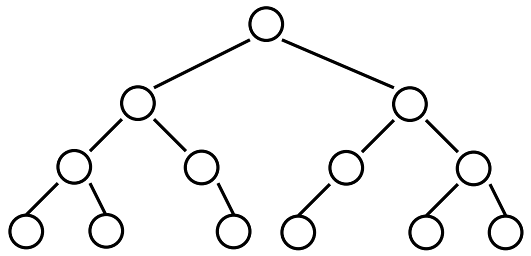
Not balanced

Left-justified binary trees

A balanced binary tree of depth n is left-justified if:

- it has 2^n nodes at depth n (the tree is "full"), or
- it has 2^k nodes at depth k, for all $k < n$, *and* all the leaves at depth n are as far left as possible


Left-justified

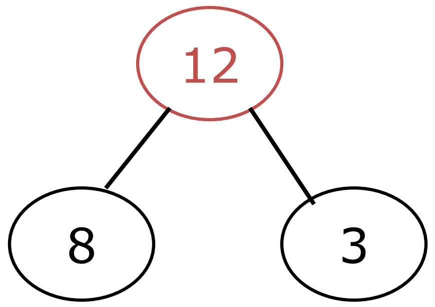

Not left-justified

Building up to heap sort

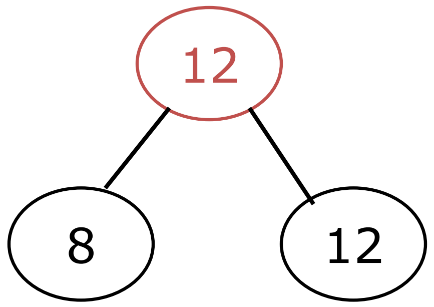
- How to build a heap
- How to maintain a heap
- How to use a heap to sort data

The heap property

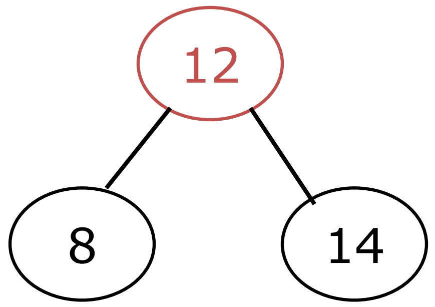
- A node has the heap property if the value in the node is as large as or larger than the values in its children



Blue node has heap property



Blue node has heap property



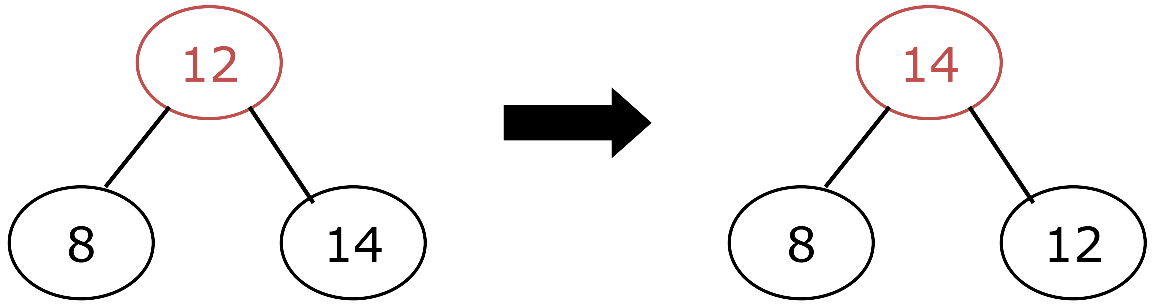
Blue node does not have heap property

- All leaf nodes automatically have the heap property

- A binary tree is a heap if all nodes in it have the heap property

siftUp

Given a node that does not have the heap property, you can give it the heap property by exchanging its value with the value of the larger child



Blue node does not have heap property

Blue node has heap property

This is sometimes called sifting up

Constructing a heap I

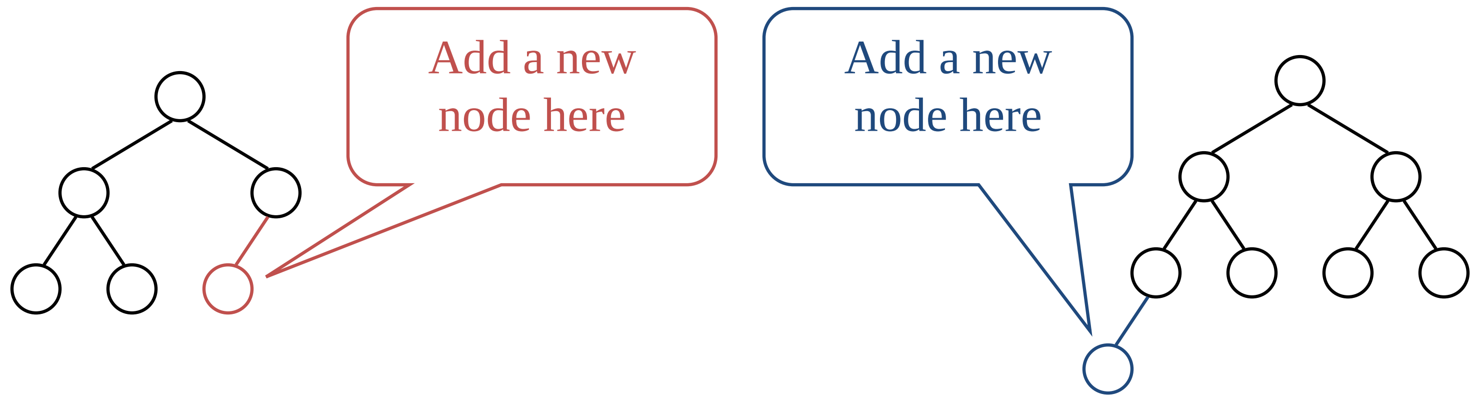
A tree consisting of a single node is automatically a heap

We construct a heap by adding nodes one at a time:

Add the node just to the right of the rightmost node in the deepest level

If the deepest level is full, start a new level

Examples:



Constructing a heap II

Each time we add a node, we may destroy the heap property of its parent node

To fix this, we sift up

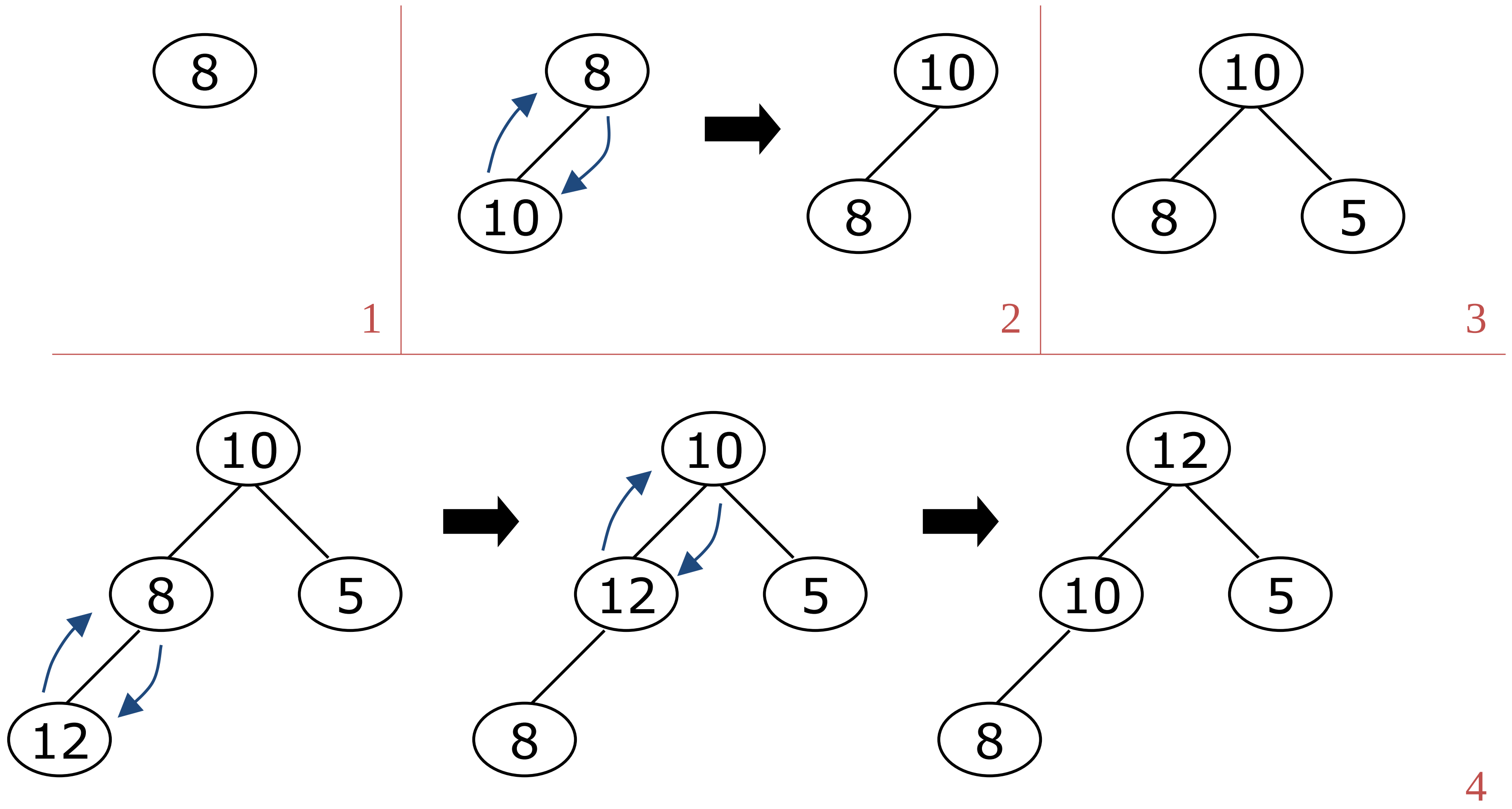
But each time we sift up, the value of the topmost node in the sift may increase, and this may destroy the heap property of its parent node

We repeat the sifting up process, moving up in the tree, until either

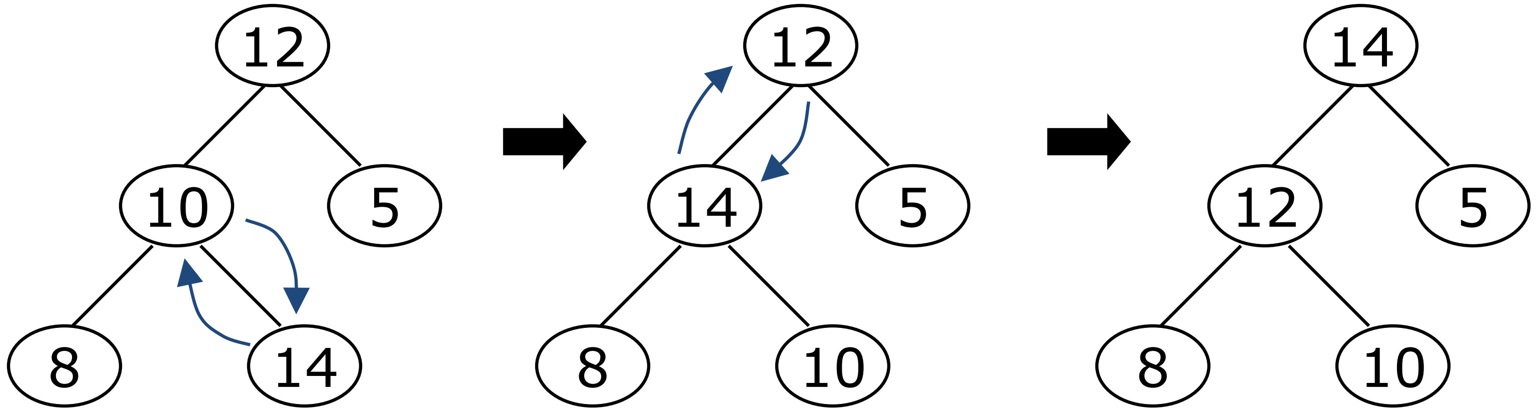
We reach nodes whose values don't need to be swapped (because the parent is still larger than both children), or

We reach the root

Constructing a heap III



Other children are not affected



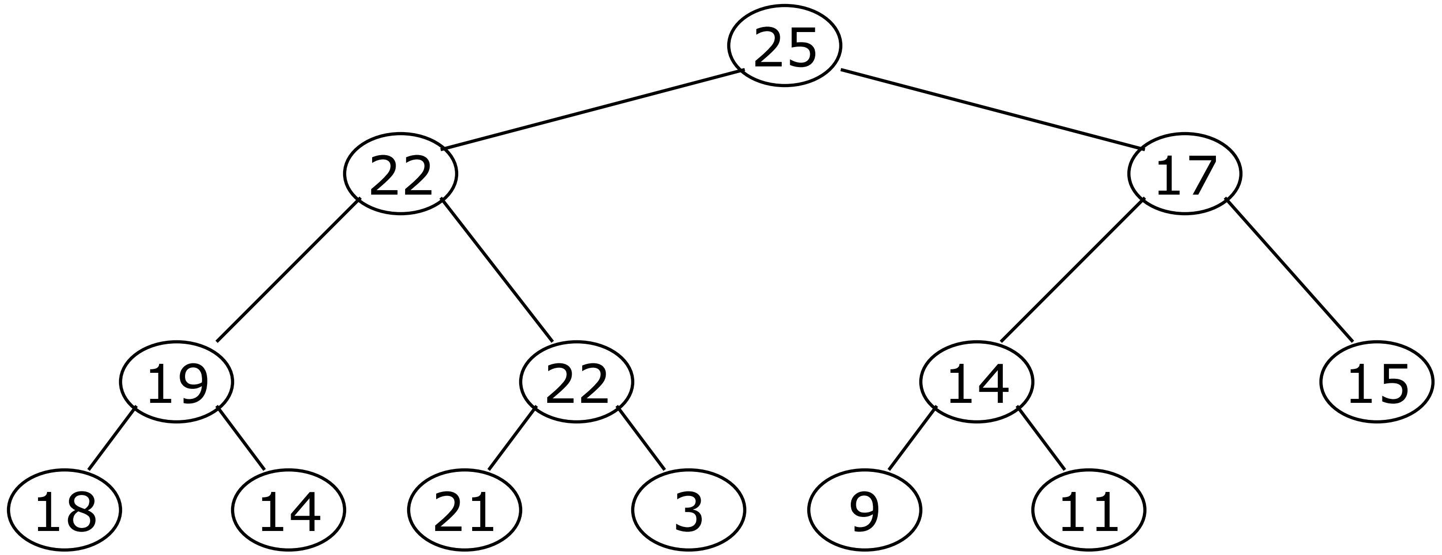
The node containing 8 is not affected because its parent gets larger, not smaller

The node containing 5 is not affected because its parent gets larger, not smaller

The node containing 8 is still not affected because, although its parent got smaller, its parent is still greater than it was originally

A sample heap

Here's a sample binary tree after it has been heapified



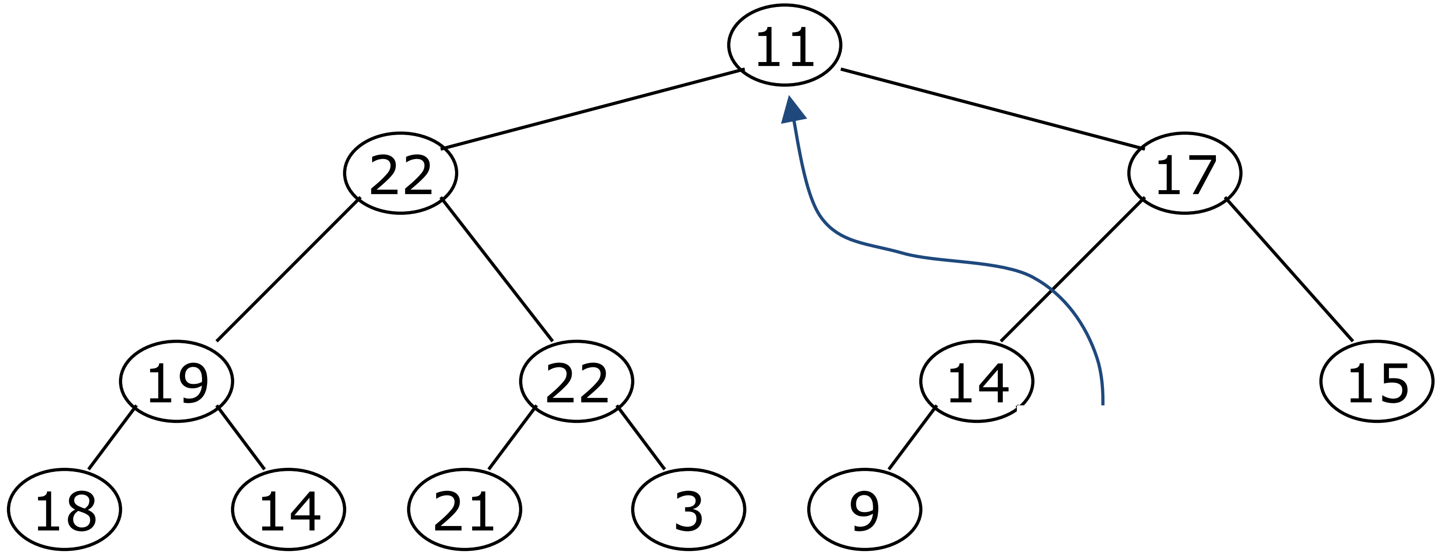
Notice that heapified does *not* mean sorted

Heapifying does *not* change the shape of the binary tree; this binary tree is balanced and left-justified because it started out that way

Removing the root (animated)

Notice that the largest number is now in the root

Suppose we *discard* the root:

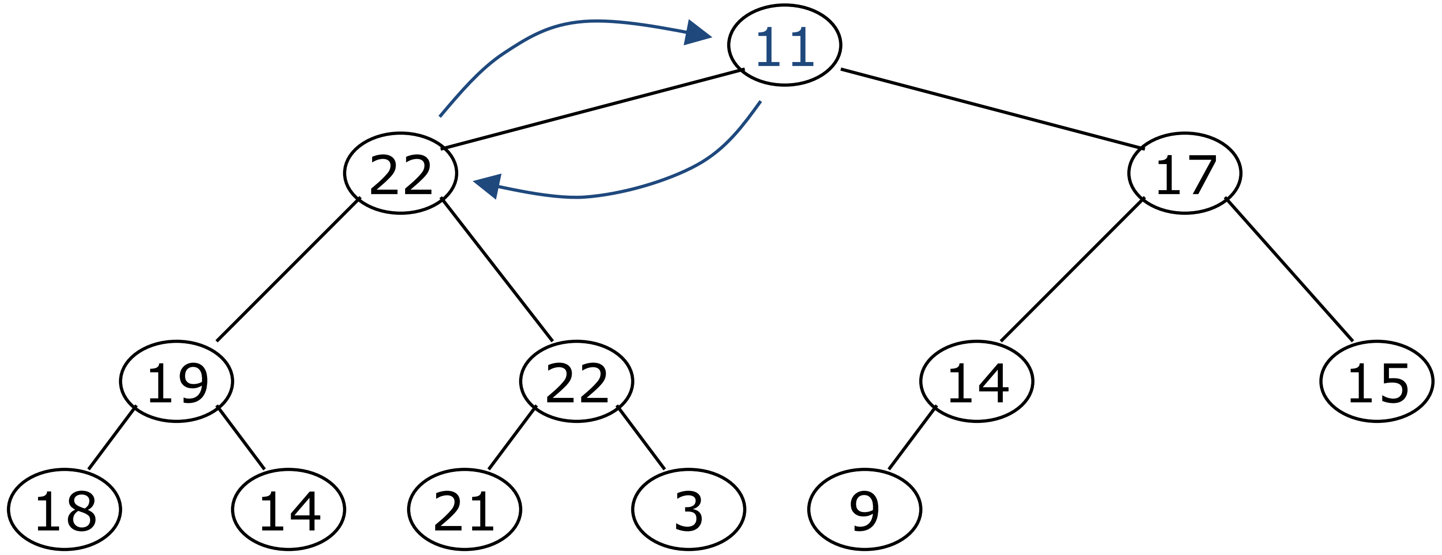


How can we fix the binary tree so it is once again *balanced and left-justified?*

Solution: remove the rightmost leaf at the deepest level and use it for the new root

The reHeap method I

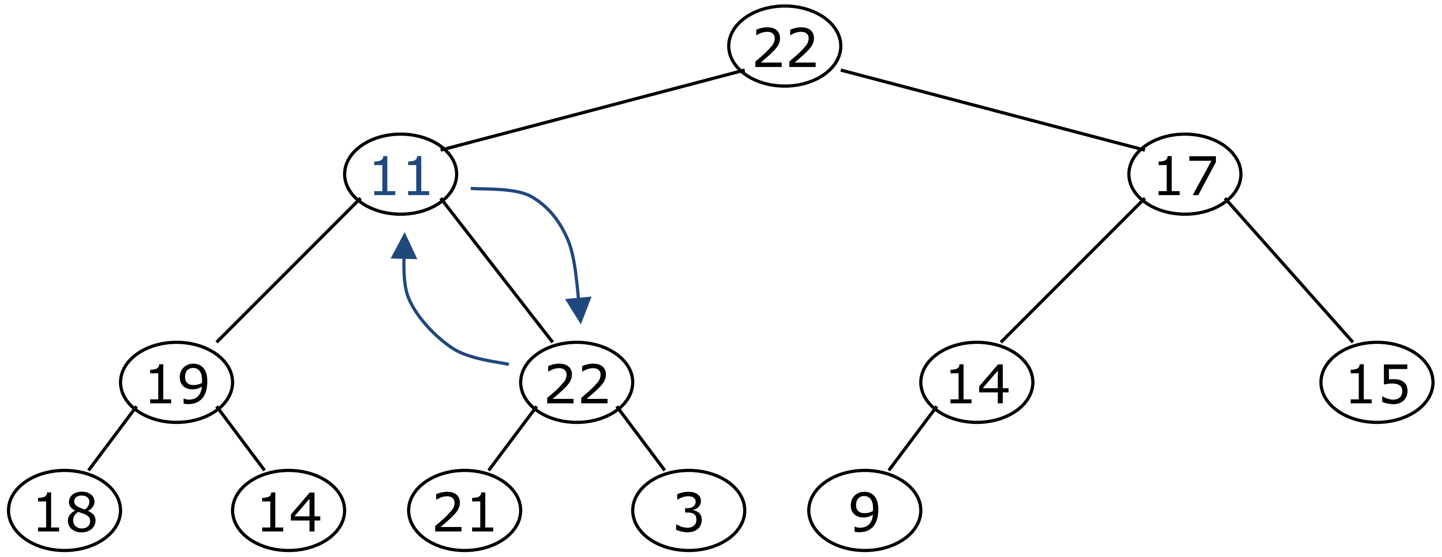
Our tree is balanced and left-justified, but no longer a heap
However, *only the root* lacks the heap property



We can siftDown() the root
After doing this, one and only one of its children may have lost the heap property

The reHeap method II

Now the left child of the root (still the number 11) lacks the heap property

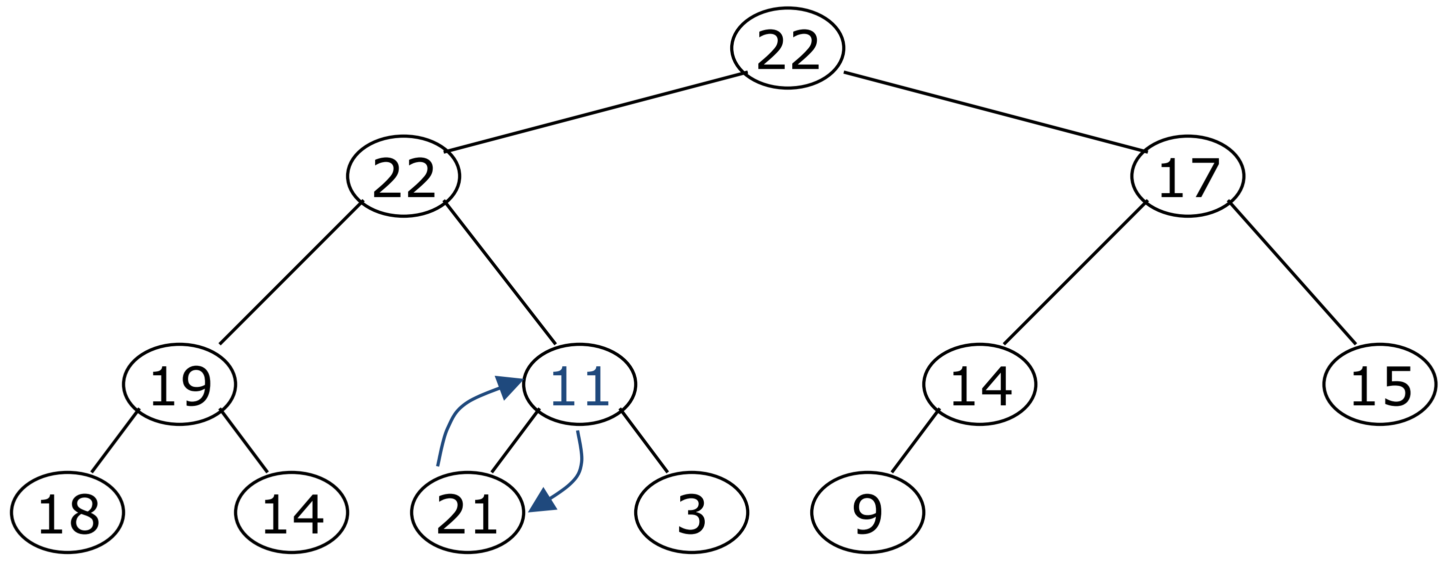


We can siftDown() this node

After doing this, one and only one of its children may have lost the heap property

The reHeap method III

Now the right child of the left child of the root (still the number 11) lacks the heap property:

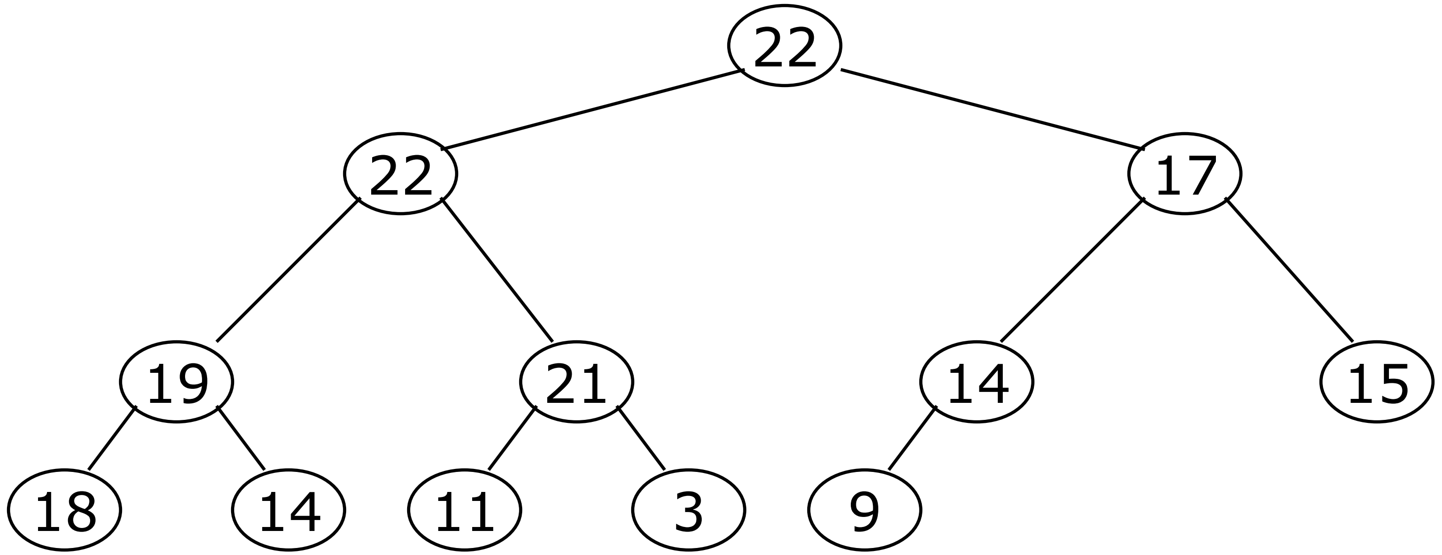


We can siftDown() this node

After doing this, one and only one of its children may have lost the heap property —but it doesn't, because it's a leaf

The reHeap method IV

Our tree is once again a heap, because every node in it has the heap property



Once again, the largest (or a largest) value is in the root

We can repeat this process until the tree becomes empty

This produces a sequence of values in order largest to smallest

Sorting

What do heaps have to do with sorting an array?

Here's the neat part:

Because the binary tree is *balanced* and *left justified,* it can be represented as an array

Danger Will Robinson: This representation works well ***only*** with ***balanced***, ***left-justified*** binary trees

All our operations on binary trees can be represented as operations on *arrays*

To sort:

```
heapify the array;
while the array isn't empty {
    remove and replace the root;
    reheap the new root node;
}
```

Key properties

- Determining location of root and “last node” take constant time
- Remove n elements, re-heap each time

Analysis

To reheap the root node, we have to follow *one path* from the root to a leaf node (and we might stop before we reach a leaf)

The binary tree is perfectly balanced

Therefore, this path is O(log n) long

- And we only do O(1) operations at each node
- Therefore, reheaping takes O(log n) times

Since we reheap inside a while loop that we do n times, the total time for the while loop is n*O(log n), or O(n log n)

Analysis

Construct the heap	O(n log n)
Remove and re-heap	O(log n)
Do this n times	O(n log n)
Total time	O(n log n) + O(n log n)

Thank You!