

Sorting

CPT 204 - Advanced OO
Programming


AY 24/25

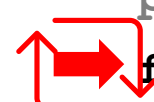
Objectives


- To study and analyze time complexity of various sorting algorithms
 - To design, implement, and analyze *bubble sort*
 - To design, implement, and analyze *merge sort*
 - To design, implement, and analyze *quick sort*
 - To design and implement a *binary heap*
 - To design, implement, and analyze *heap sort*

Bubble Sort

Repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order

```
// Outer loop controls the number of passes

for (int k = 1; k < list.length; k++) {

    // In each pass, the inner loop compares each
    // pair of adjacent elements
    
    for (int i = 0; i < list.length - k; i++) {

        
        if (list[i] > list[i + 1])
            // swap list[i] with list[i + 1]
    }
}
```

Bubble Sort

The image shows a software application titled "Bubble Sort Animator". At the top, there is a text input field labeled "Array:" containing the values "4,3,2,1". To the right of this field are two buttons: "Start" and "Next Step". Below the input field, four cyan-colored boxes represent the array elements, each with a number and a label underneath: "4" above "list[0]", "3" above "list[1]", "2" above "list[2]", and "1" above "list[3]". The bottom half of the window is divided into two panels. The left panel contains the text "Array Initialized: [4, 3, 2, 1]". The right panel displays a Java code snippet for the bubble sort algorithm. A mouse cursor is visible over the right panel.

Bubble Sort Animator

Array: 4,3,2,1

Start Next Step

4 3 2 1

list[0] list[1] list[2] list[3]

Array Initialized: [4, 3, 2, 1]

```
for (int k = 1; k < list.length; k++) {  
    for (int i = 0; i < list.length - k; i++)  
        if (list[i] > list[i + 1]) {  
            // swap list[i] and list[i + 1]  
        }  
}
```

Bubble Sort Optimization

What if a list is almost sorted or already sorted like [1,2,3,4]?

- A typical bubble sort would **keep doing passes** as long as $k < \text{list.length}$, even though **no swap is actually needed** in any of these passes. Just wasting time and resources.
- So, we do the following optimization

```
➡ boolean needNextPass = true;

for (int k = 1; k < list.length
➡ && needNextPass; k++) {

    // Array may be sorted and next
    pass not needed

    ➡ needNextPass = false;

    for (int i = 0; i <
        list.length - k; i++){

        if (list[i] > list[i + 1]){

            //swap list[i] with list[i +
            1];

            ➡ needNextPass = true;

        }

    }

}
```

Example: [1,2,3,4]

Will never be a $\text{list}[i] > \text{list}[i+1]$

Codes in the IF statement never implement \Rightarrow $\text{needNextPass} = \text{false}$

Stop in the 2nd pass (because of `&& needNextPass;`)

Bubble Sort Analysis

- In the **best case**, the bubble sort algorithm needs just the first pass to find that the array is already sorted—no next pass is needed.
- Since the number of comparisons is $n - 1$ in the first pass, the best-case time for a bubble sort is $O(n)$.

Worst case example: Initial list: [5,4,3,2,1]

➡ [5,4,3,2,1] -> [4,3,2,1,5] - 1st pass, 4 comparisons (5 v.s. 4,3,2,1, 5 is sorted)
➡ [4,3,2,1,5] -> [3,2,1,4,5] - 2nd pass, 3 comparisons (4 v.s. 3,2,1, 4 is sorted)
➡ [3,2,1,4,5] -> [2,1,3,4,5] - 3rd pass, 2 comparisons (3 v.s. 1,2, 3 is sorted)
➡ [2,1,3,4,5] -> [1,2,3,4,5] - 4th pass, 1 comparisons (2 v.s. 1, 2 is sorted)

- So, 5 elements, we do 4+3+2+1 comparisons
- If n elements, in the worst case, we compare $(n - 1) + (n - 2) + \dots + 2 + 1 = n(n - 1) / 2$ times. The Big(O) is $O(n^2)$

```

public class BubbleSort {
    public static void bubbleSort(int[] list) {
        boolean needNextPass = true;
        for (int k = 1; k < list.length && needNextPass; k++) {
            // Array may be sorted and next pass not needed
            needNextPass = false;
            // Perform the kth pass
            for (int i = 0; i < list.length - k; i++) {
                if (list[i] > list[i + 1]) {
                    int temp = list[i];
                    list[i] = list[i + 1];
                    list[i + 1] = temp;
                    needNextPass = true; // Next pass still needed
                }
            }
        }
    }

    public static void main(String[] args) {
        int size = 100000;
        int[] a = new int[size];
        randomInitiate(a);
        long startTime = System.currentTimeMillis();
        bubbleSort(a);
        long endTime = System.currentTimeMillis();
        System.out.println((endTime - startTime) + "ms");
    }

    private static void randomInitiate(int[] a) {
        for (int i = 0; i < a.length; i++)
            a[i] = (int) (Math.random() * a.length);
    }
}

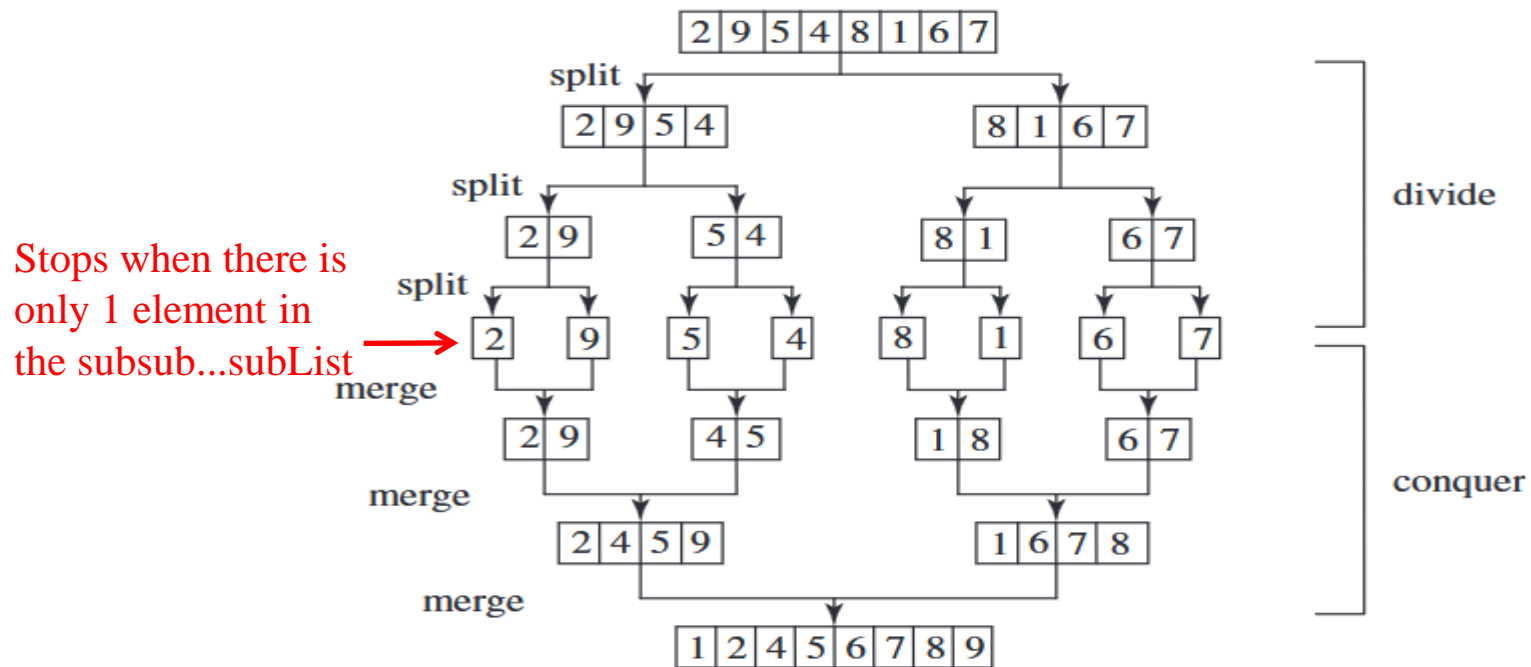
```

14650ms

Merge Sort

The merge-sort algorithm can be described recursively as follows:

- The algorithm divides the array into two halves and applies a merge sort on each half recursively.
- After the two halves are sorted, the algorithm then merges them.




```
public static void mergeSort(int[] list) {  
    if (list.length > 1) { // Recursive base case: stop when condition unsatisfied  
        // Split the 1st half (recursive step)  
        int[] firstHalf = new int[list.length / 2];  
        System.arraycopy(list, 0, firstHalf, 0, list.length / 2);  
        mergeSort(firstHalf);  
  
        // Split the 2nd half (recursive step)  
        int secondHalfLength = list.length - list.length / 2;  
        int[] secondHalf = new int[secondHalfLength];  
        System.arraycopy(list, list.length / 2, secondHalf, 0,  
            secondHalfLength);  
        mergeSort(secondHalf);  
  
        // SortMerge (only happens AFTER both recursive calls  
        finish)  
        merge(firstHalf, secondHalf, list);  
    }  
}
```

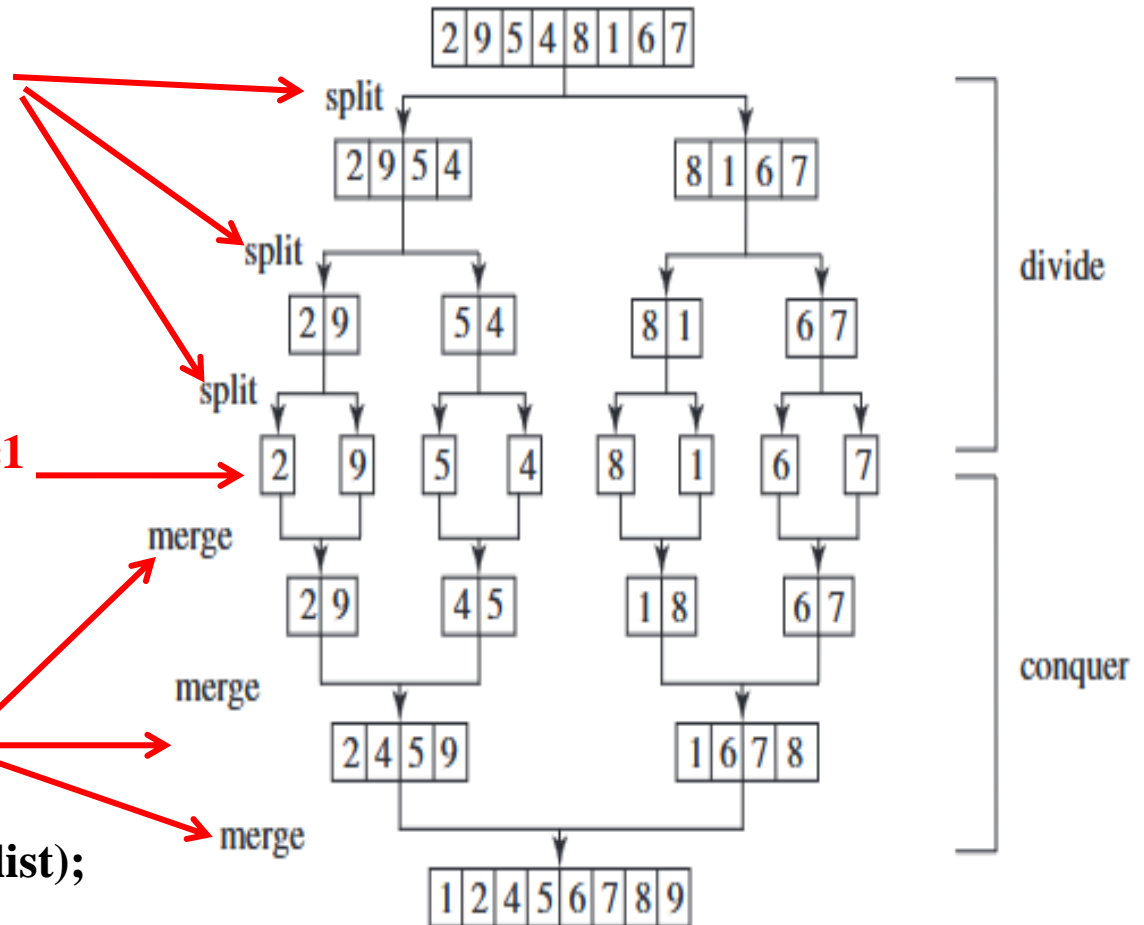
MergeSort()

// recursively calling itself
mergeSort(firstHalf);

// recursively calling itself
mergeSort(secondHalf);

// stops at base case length=1
if (list.length > 1)

// After the 2 recursive calls,
// step by step merge
merge(firstHalf, secondHalf, list);



```

public static void merge(int[] list1, int[] list2, int[] temp){
    int current1 = 0; // Current index in list1, the first half
    int current2 = 0; // Current index in list2, the 2nd half
    int current3 = 0; // Current index in temp, storing data temporarily

    // While the indices are in the list
    while (current1 < list1.length && current2 < list2.length) {
        if (list1[current1] < list2[current2])
            // If current element in list1 is smaller, add it to temp
            temp[current3++] = list1[current1++];
        else
            // Otherwise, add the current element in list2 to temp
            temp[current3++] = list2[current2++];
    }

    // list2 finished, but there are remaining elements in list1, add
    // them to temp
    while (current1 < list1.length)
        temp[current3++] = list1[current1++];

    // list1 finished, but there are remaining elements in list2, add
    // them to temp
    while (current2 < list2.length)
        temp[current3++] = list2[current2++];
}

```

Merge Sort Animator

List1: 2,4,5,9 List2: 1,6,7,8 Start Next Step Merge Started Code Window Height:

current1

| | | | |
|---|---|---|---|
| 2 | 4 | 5 | 9 |
|---|---|---|---|

current2

| | | | |
|---|---|---|---|
| 1 | 6 | 7 | 8 |
|---|---|---|---|

current3

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

List1: [2, 4, 5, 9]
List2: [1, 6, 7, 8]


```
public static void merge(int[] list1, int[] list2, int[] temp) {  
    int current1 = 0; // Current index in list1  
    int current2 = 0; // Current index in list2  
    int current3 = 0; // Current index in temp  
    while (current1 < list1.length && current2 < list2.length) {  
        if (list1[current1] < list2[current2])  
            temp[current3++] = list1[current1++];  
        else  
            temp[current3++] = list2[current2++];  
    }  
    while (current1 < list1.length)  
        temp[current3++] = list1[current1++];  
    while (current2 < list2.length)  
        temp[current3++] = list2[current2++];  
}
```

```

public class MergeSortTest {
    public static void mergeSort(int[] list) {
        if (list.length > 1) {
            // Merge sort the first half
            int[] firstHalf = new int[list.length / 2];
            System.arraycopy(list, 0, firstHalf, 0, list.length / 2);
            mergeSort(firstHalf);
            // Merge sort the second half
            int secondHalfLength = list.length - list.length / 2;
            int[] secondHalf = new int[secondHalfLength];
            System.arraycopy(list, list.length / 2, secondHalf, 0, secondHalfLength);
            mergeSort(secondHalf);
            // Merge firstHalf with secondHalf into list
            merge(firstHalf, secondHalf, list);
        }
    }

    public static void merge(int[] list1, int[] list2, int[] temp) {
        int current1 = 0; // Current index in list1
        int current2 = 0; // Current index in list2
        int current3 = 0; // Current index in temp
        while (current1 < list1.length && current2 < list2.length) {
            if (list1[current1] < list2[current2])
                temp[current3++] = list1[current1++];
            else
                temp[current3++] = list2[current2++];
        }
        while (current1 < list1.length)
            temp[current3++] = list1[current1++];
        while (current2 < list2.length)
            temp[current3++] = list2[current2++]
    }
}

```



```
public static void main(String[] args) {  
    int size = 100000;  
    int[] a = new int[size];  
    randomInitiate(a);  
    long startTime = System.currentTimeMillis();  
    mergeSort(a);  
    long endTime = System.currentTimeMillis();  
    System.out.println((endTime - startTime) + "ms");  
}  
  
private static void randomInitiate(int[] a) {  
    for (int i = 0; i < a.length; i++)  
        a[i] = (int) (Math.random() * a.length);  
}  
}
```

16ms (v.s. 14650ms in bubble sort)

Merge Sort Time Complexity

- Let $T(n)$ denote the time required for sorting an array of n elements using merge sort.

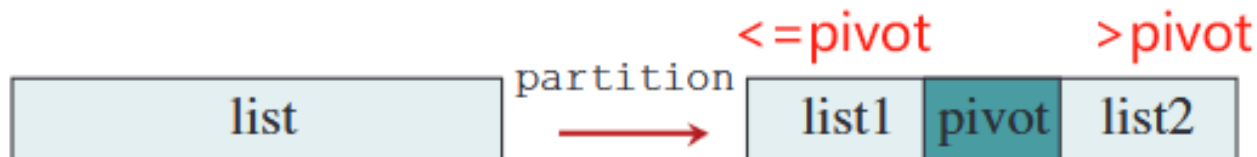
$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + 2n-1$$

- The first $T(n/2)$ is the time for sorting the first half of the array and the second $T(n/2)$ is the time for sorting the second half
- Merging cost $2n-1$ because $n - 1$ comparisons (for comparing the elements of the two subarrays) and n moves (to place each element into the temporary array)
- Repeatedly substitute $T(n/2)$ into the formula we will find the time complexity of merge sort is $O(n \log n)$.

Quick Sort

A quick sort works as follows:

- The algorithm selects an element, called the **pivot**, in the array.
- It partitions (divides) the array into two parts so all the elements in the **first part** are less than or equal to the pivot, and all the elements in the **second part** are greater than the pivot.
- The quick-sort algorithm is then **recursively** applied to the first part and then the second part to sort them out.




```
public static int partition(int[] list, int first, int last) {
```

```
    int pivot = list[first];
```

```
    int low = first + 1; // position, not value
```

```
    int high = last; // position, not value
```

```
    while (high > low) {
```

```
        //low pointer moves right when
```

```
        //1. low <= high and 2. list[low] <= pivot
```

```
        while (low <= high && list[low] <= pivot)
```

```
            low++;
```

```
        //high pointer moves left when
```

```
        //1. low <= high and 2. list[high] > pivot
```

```
        while (low <= high && list[high] > pivot)
```

```
            high--;
```

```
        // If low < high, swap the two elements.
```

```
        if (high > low) {
```

```
            int temp = list[high];
```

```
            list[high] = list[low];
```

```
            list[low] = temp;
```

```
        }
```

```
    }
```

```
    // Ensure high pointer point at an element
```

```
    // less than or equal to pivot
```

```
    while (high > first && list[high] >= pivot){
```

```
        high--;
```

```
    }
```

```
    // Swap pivot with list[high]
```

```
    if (pivot > list[high]) {
```

```
        list[first] = list[high];
```

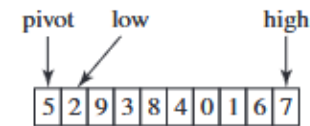
```
        list[high] = pivot;
```

```
        return high;
```

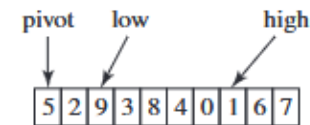
```
    } else // e.g., a sorted list
```

```
        return first;
```

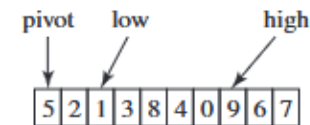
```
    }
```



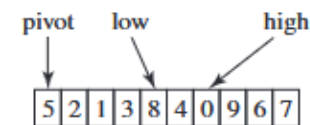
(a) Initialize pivot, low, and high



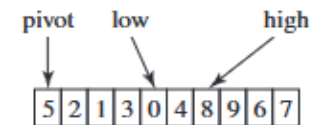
(b) Search forward and backward



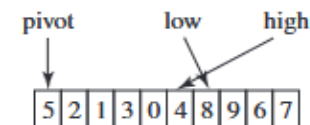
(c) 9 is swapped with 1



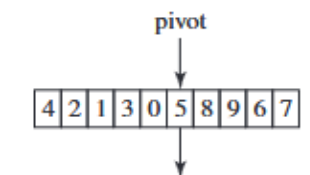
(d) Continue search



(e) 8 is swapped with 0



(f) When high < low, search is over



(g) Pivot is in the right place

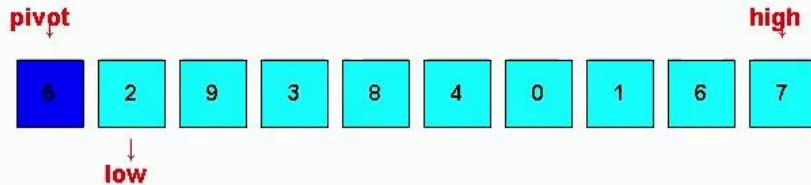
The index of the pivot is returned

Array: 5,2,9,3,8,4,0,1,6,7

Start

Next Step

Partition Started Code Window Height:



Initial Array: [5, 2, 9, 3, 8, 4, 0, 1, 6, 7]

Pivot = 5 at index 0

low pointer = 1, high pointer = 9

```
int pivot = list[first];
int low = first + 1; // position, not value
int high = last; // position, not value
while (high > low) {
    while (low <= high && list[low] <= pivot) {
        low++;
    }
    while (low <= high && list[high] > pivot) {
        high--;
    }
    if (high > low) {
        int temp = list[high];
        list[high] = list[low];
        list[low] = temp;
    }
}
while (high > first && list[high] >= pivot) {
    high--;
}
if (pivot > list[high]) {
    list[first] = list[high];
    list[high] = pivot;
    return high;
} else {
    return first;
}
```

```

public class QuickSortTest {
    public static void quickSort(int[] list) {
        quickSort(list, 0, list.length - 1);
    }
    public static void quickSort(int[] list, int first, int last) {
        if (last > first) {
            int pivotIndex = partition(list, first, last);
            quickSort(list, first, pivotIndex - 1);
            quickSort(list, pivotIndex + 1, last);
        }
    }
    public static int partition(int[] list, int first, int last) {
        int pivot = list[first]; // Choose the first element as pivot
        int low = first + 1; // Index for forward search
        int high = last; // Index for backward search
        while (high > low) {
            // Search forward from left
            while (low <= high && list[low] <= pivot)
                low++;

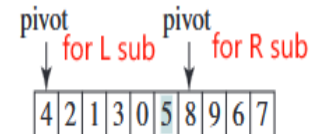
            // Search backward from right
            while (low <= high && list[high] > pivot)
                high--;

            // Swap two elements in the list
            if (high > low) {
                int temp = list[high];
                list[high] = list[low];
                list[low] = temp;
            }
        }
        // Account for duplicated elements:
        while (high > first && list[high] >= pivot)
            high--;
    }
}

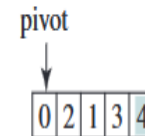
```

This **quickSort()** defines the recursive structure of Quick Sort, as it keeps calling **itself**. The **partition()** is also recursively called in the process, until the **base case** (**last>first**).

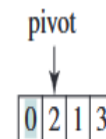
This ensures the list can be recursively divided into left and right sublist, subsublist, subsub...sublist.



(b) The original array is partitioned



(c) The subarray (4 2 1 3 0) is partitioned



(d) The subarray (0 2 1 3) is partitioned



(e) The subarray (2 1 3) is partitioned

```

    // Swap pivot with list[high]
    if (pivot > list[high]) {
        list[first] = list[high];
        list[high] = pivot;
        return high;
    } else
        return first;
}

public static void main(String[] args) {
    int size = 100000;
    int[] a = new int[size];
    randomInitiate(a);

    long startTime = System.currentTimeMillis();
    quickSort(a);
    long endTime = System.currentTimeMillis();

    System.out.println((endTime - startTime) + "ms");
}

private static void randomInitiate(int[] a) {
    for (int i = 0; i < a.length; i++)
        a[i] = (int) (Math.random() * a.length);
}
}

```


16ms (v.s. 16ms in merge sort and 14650ms in bubble sort)

Time Complexity - Best and Average Case

- To partition an array of **n** elements, it takes **n** comparisons and **n** moves. Thus, the time required for partition is $O(n)$.
- In the **best case**, each time the pivot divides the array into two parts of the same size
- In the **average case**, maybe not exactly the same, but the size of the two sub arrays are very close.

recursive quick sort on
two subarrays

partition time


$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n.$$

$$= O(n \log n)$$

Time Complexity - Worst Case

In the worst case, the pivot divides the array each time into one big subarray **with the other array empty**. The size of the big subarray is one less than the one before divided

For example (assuming the **1st element** is the pivot for partition):

- $[1, 2, 3, 4, 5 \dots n]$, size = n
- In the first partition, **pivot=1**, we have ->
 - left: [empty], right $[2, 3, 4, 5 \dots n]$, right sub array size = $n-1$
- In the next partition, **pivot=2**, we have ->
 - left: [empty], right $[3, 4, 5, 6 \dots n]$, right sub array size = $n-2$
- ... continues this way, we will have to recursively divide the array **$n-1$** times till the sub array size = 1
- Recall that the time complexity of each partition is $O(n)$, as it compares all elements during each division.
- So we did **$n-1$** times $O(n)$, so, we get the worst time complexity to be $O(n^2)$

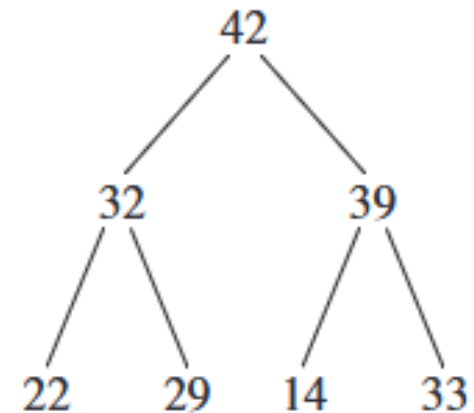
Heap Sort: Binary tree

A *binary tree* is a hierarchical structure: it either is empty or it consists of an element, called the *root*, and two distinct binary trees, called the *left subtree* and *right subtree*

- The *length* of a path is the number of the edges in the path
- The *depth* of a node is the length of the path from the root to that node

The length from 32 to 22: 1 (32 - 22)

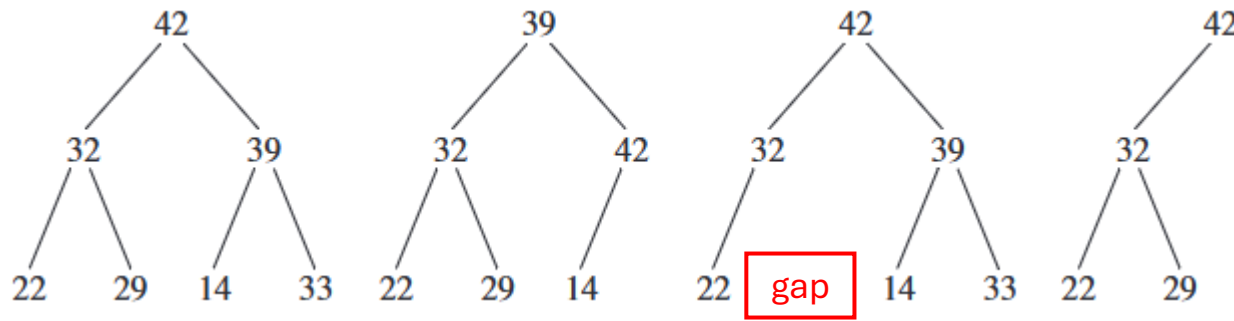
The depth of node 29: 2 (42(root)-32, 32-29)



Complete Binary Tree

A binary tree is *complete* if :

- All levels are completely filled except **possibly** the last level
- Even though the last level may not be full, it must be filled from left to right, **without gaps**

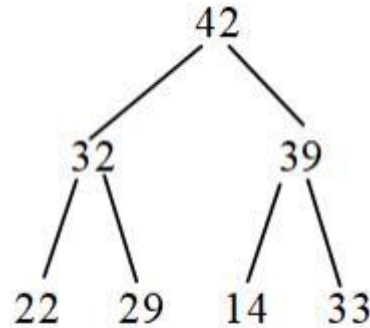


- 1st - complete, as all levels are completely filled
- 2nd - complete, although the last level is not full (missing one node next to 14), it is filled from left to right, without gaps
- 3rd, incomplete, a gap from left to right (next to node 22)
- 4th, incomplete, the 2nd level is not full, while we only allow the last level (level 3 in this case) not to be full

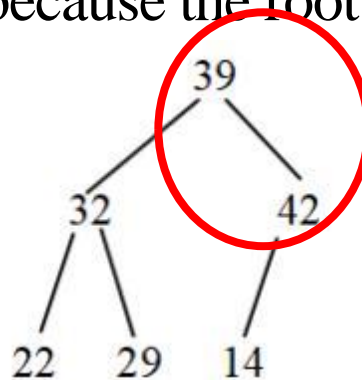
Binary Heap

- A *binary heap* is a binary tree with the following properties:
 - It is a **complete binary tree**, and
 - **Each node** is greater than or equal to any of its children

▪ Example heap:



- Example not a heap, because the root (39) is less than its right child (42)

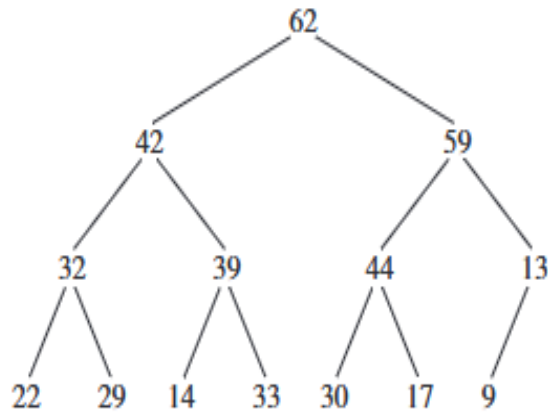


Heap Sort

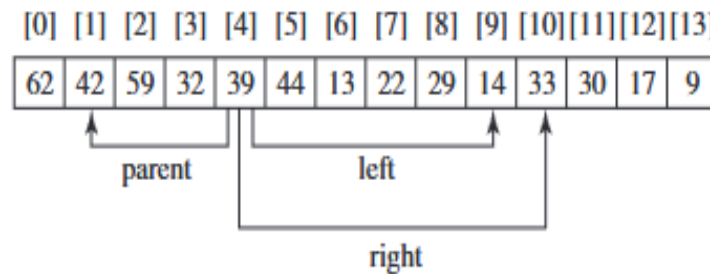
- *Heap sort* uses a binary heap and the process consists of **two** main phases:
 - **Heap construction:**
 - All the elements are first inserted into a max heap
 - **Repeated removal:**
 - Repeatedly remove the **root node**, which is the current largest element in the heap. The removed element is actually moved to the end of the array, forming a sorted array that grows from the back.
- For example: [10,5,3,4,1]
 - 1st removal: [..., 10]
 - 2nd removal: [..., 5, 10]
 - 3rd removal: [..., 4, 5, 10]
 - (do removal repeatedly)
 - Final: [1,3,4,5,10]

Storing a Heap

- A heap can be stored in an **ArrayList** or an **array** if the heap size is known in advance
- For a node at position i , its left child is at position $2i+1$ and its right child is at position $2i+2$, and its parent is at index $(i-1)/2$
- For example: the for a nood at position **4**, and its two children are at positions $2*4+1=9$ and $2*4+2=10$, and its parent is at index $(4-1)/2=1$ (not 1.5, integer division).



(a) A heap



(b) A heap stored in an array

Adding Elements to a Heap

- To add a new node to a heap, first add it to the end of the heap and then rebuild the tree with this algorithm:

```
Let the last node be the current node;  
while (the current node is greater than its parent) {  
    Swap the current node with its parent;  
    Now the current node is one level up;  
}
```

Adding Elements to the Heap

- Suppose a heap is initially empty. after adding numbers 3, 5, 1, 19, 11, and 22 **in this order**

heap is empty

Enter a key:

Add

Remove the root

Removing the Root and Rebuild the Heap

- Often we need to remove the maximum element, which is the **root in a heap**
- After the root is removed, the tree must be rebuilt to **maintain the heap property (e.g., $\text{parent} \geq \text{child}$)** using this algorithm:

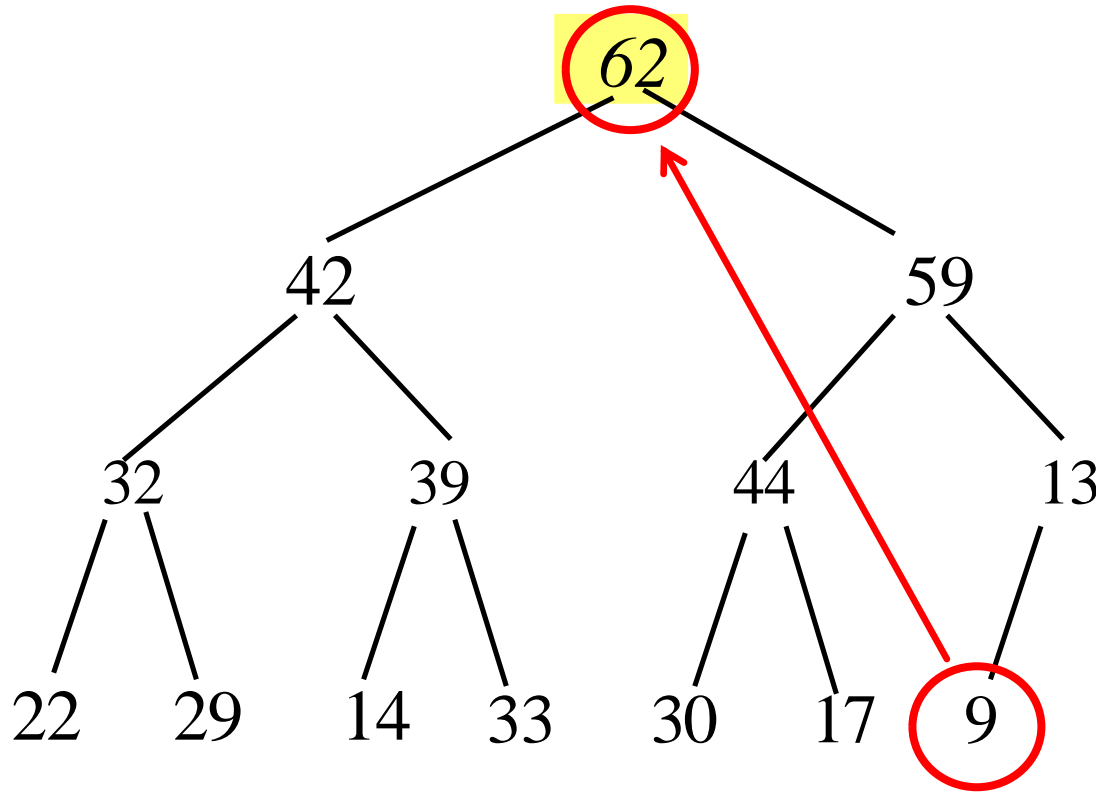
Move the last node to replace the root;

Let the root be the current node;

```
while (the current node has children and the
current node is smaller than one of its children)
{   Swap the current node with the larger of its
    children;
    Now the current node is one level down;
}
```

Removing the Root and Rebuild the Heap

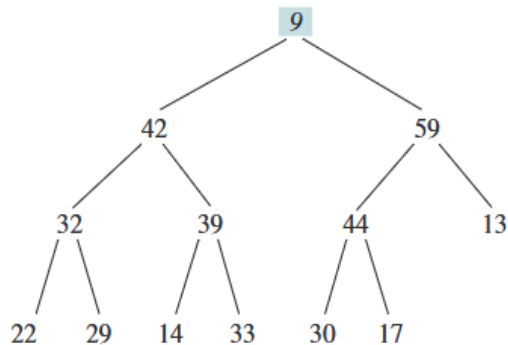
- Removing root 62 from the heap (replaces it with the last node in the heap: 9)



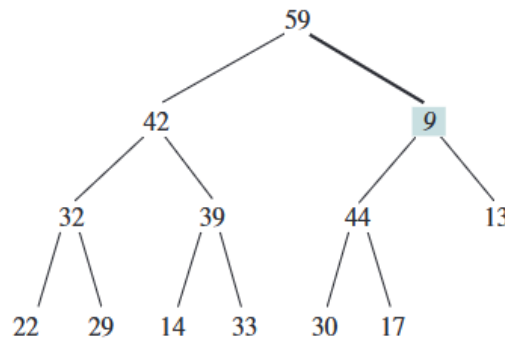
Move the last node to replace the root;
Let the root be the current node;

Removing the Root and Rebuild the Heap

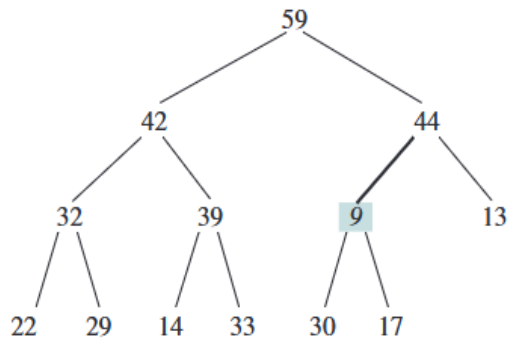
```
while (the current node has children and the  
current node is smaller than one of its children)  
{ Swap the current node with the larger of its  
children;  
}
```



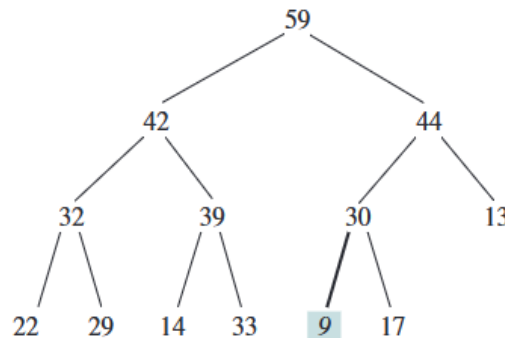
(a) After moving 9 to the root



(b) After swapping 9 with 59



(c) After swapping 9 with 44



(d) After swapping 9 with 30

(a-b) $9 < 42$; $9 < 59$; $59 > 42$;
Swap 9 and 59

(b-c) $9 < 44$; $9 < 13$; $44 > 13$;
Swap 9 and 44

(c-d) $9 < 30$; $9 < 17$; $30 > 17$;
Swap 9 and 30

The heap is rebuilt now, and
ready for the next removal

The Heap Class

Heap<E>

-list: java.util.ArrayList<E>
-c: java.util.comparator<E>

+Heap()
+Heap(c: java.util.Comparator<E>)
+Heap(objects: E[])
+add(newObject: E): void
+remove(): E
+getSize(): int
+isEmpty(): boolean

Creates a default empty Heap.

Creates an empty heap with the specified comparator.

Creates a Heap with the specified objects.

Adds a new object to the heap.

Removes the root from the heap and returns it.

Returns the size of the heap.

Returns true if the heap is empty.

```

public class Heap<E extends Comparable> {
    private java.util.ArrayList<E> list = new java.util.ArrayList<E>();
    /** Create a default heap */
    public Heap() {
    }
    /** Create a heap from an array of objects */
    public Heap(E[] objects) {
        for (int i = 0; i < objects.length; i++)
            add(objects[i]);
    }
    /** Add a new object into the heap */
    public void add(E newObject) {
        list.add(newObject); // Append to the end of the heap
        int currentIndex = list.size() - 1; // The index of the last node
        while (currentIndex > 0) {

            int parentIndex = (currentIndex - 1) / 2;
            // Swap if the current object is greater than its parent
            if (list.get(currentIndex).compareTo(
                list.get(parentIndex)) > 0) {
                E temp = list.get(currentIndex);
                list.set(currentIndex, list.get(parentIndex));
                list.set(parentIndex, temp);
            } else
                break; // the tree is a heap now
            currentIndex = parentIndex;
        }
    }
}

```

```
/** Remove the root from the heap */
public E remove() {
    if (list.size() == 0) return null;

    E removedObject = list.get(0);
    list.set(0, list.get(list.size() - 1));
    list.remove(list.size() - 1);

    int currentIndex = 0;
    while (currentIndex < list.size()) {
        int leftChildIndex = 2 * currentIndex + 1;
        int rightChildIndex = 2 * currentIndex + 2;

        // Find the maximum between two children
        if (leftChildIndex >= list.size())
            break; // The tree is a heap
        int maxIndex = leftChildIndex;
        if (rightChildIndex < list.size())
            if (list.get(maxIndex).compareTo(
                list.get(rightChildIndex)) < 0)
                maxIndex = rightChildIndex;
    }
}
```

```

    // Swap if the current node is less than the maximum
    if (list.get(currentIndex).compareTo(
        list.get(maxIndex)) < 0) {
        E temp = list.get(maxIndex);
        list.set(maxIndex, list.get(currentIndex));
        list.set(currentIndex, temp);
        currentIndex = maxIndex;
    }
    else
        break; // The tree is a heap
}
return removedObject;
}

```

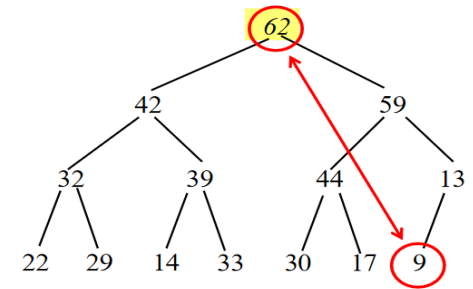
```

/** Get the number of nodes in the tree */
public int getSize() {
    return list.size();
}
}

```

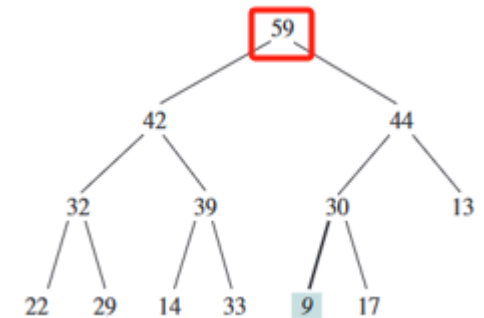
Heap Sort

```
public class HeapSort {  
    public static <E extends Comparable> void heapSort(E[] list) {  
        // Create a Heap of E  
        Heap<E> heap = new Heap<E>();  
  
        // Add elements to the heap  
        for (int i = 0; i < list.length; i++)  
            heap.add(list[i]);  
  
        // Remove the highest elements from the heap  
        // and store them in the list from end to start  
        for (int i = list.length - 1; i >= 0; i--)  
            list[i] = heap.remove();  
    }  
  
    /** A test method */  
    public static void main(String[] args) {  
        Integer[] list = {2, 3, 2, 5, 6, 1,  
            -2, 3, 14, 12};  
        heapSort(list);  
        for (int i = 0; i < list.length; i++)  
            System.out.print(list[i] + " ");  
    }  
}
```



1st removal

[, , , , 62]



2nd removal

[, , , 59, 62]

Storing from the largest one to the
smallest one 37

Heap Sort Time Complexity

Heap Sort Time: **$O(n \log n)$**

- **Space Complexity**

- Both merge and heap sorts require $O(n \log n)$ time.
- A merge sort requires a temporary array for merging two subarrays; a heap sort does not need additional array space.
- Therefore, a heap sort is more **space efficient** than a merge sort.

Life-Long Learning and EDI Principles in OOP

CPT 204 - Advanced OO
Programming

AY 24/25

Life-Long Learning in Object-Oriented Programming

- **Definition:**

- **Life-long learning** refers to the continuous, voluntary, and self-motivated pursuit of knowledge for personal or professional development.
- In the context of Object-Oriented Programming (OOP), life-long learning encourages developers to stay updated with evolving concepts, techniques, and tools to become proficient in designing, implementing, and maintaining software systems.

- **Importance:**

- **Ever-evolving Technology Landscape:** Life-long learning ensures that developers keep up with the latest best practices, frameworks, and design patterns.
- **Increased Professional Competence:** Lifelong learning enhances your ability to solve complex problems, optimize code, and design robust systems.
- **Adaptation to New Tools:** Learning new tools in OOP keeps you agile and opens up new career opportunities.
- **Personal Growth:** Continuously improving your OOP skills not only enhances your technical expertise but also develops other valuable skills such as critical thinking, problem-solving, and effective communication.

Recording and Reflecting on Your Learning

- *Maintain a reflective journal/log where you document:*
 - Key takeaways from each learning session.
 - Insights and thoughts gained after solving coding challenges.
 - Questions or areas where you need further clarification.
 - Code snippets and examples you've worked on. A journal helps consolidate your learning and provides a reference for future problems.
- *Regular Self-Assessment:*
 - Regularly evaluate your progress against your set goals. Are you mastering the basics, or do you need to revisit certain concepts.
 - Be honest about your challenges. Do you struggle with complex design patterns or the implementation of advanced OOP principles? Focus on these areas next.
 - Recognize your achievements. Completing a course or contributing to an OOP project is a big milestone worth celebrating
- *Join Communities and Forums*
 - Join OOP-focused communities on platforms like StackOverflow, Reddit, or GitHub.
 - Participate in discussions, ask questions, and share your knowledge with others.

EDI Principles in OOP

Introduction:

Equality, Diversity, and Inclusion (EDI) are essential principles that promote fairness, representation, and a welcoming environment within the software development community. These principles are crucial in the context of Object-Oriented Programming to ensure that:

- All individuals have equal access to opportunities, regardless of background.
- Diverse perspectives contribute to the improvement of the field.
- Everyone feels welcome, valued, and respected in the development process.

EDI Principles in OOP

Description:

- **Equality** in OOP means ensuring that all developers, regardless of their gender, race, sexual orientation, or disability, have equal access to career opportunities and the ability to contribute to the software development process.
- **Diversity** in OOP enriches problem-solving by bringing together people from various backgrounds, experiences, and perspectives. This diversity leads to more creative and innovative solutions to complex software design challenges.
- **Inclusion** in OOP goes beyond representation and equality; it emphasizes creating an environment where everyone feels respected and valued.

Practical Ways to Incorporate EDI into OOP Practices

Adopt Inclusive Coding Practices:

- Use inclusive language in code, comments, and documentation. Avoid terms or practices that could inadvertently alienate or exclude individuals.

Implement Accessible Design Patterns:

- Ensure that the software designs you build are accessible to a wide audience. Incorporate accessibility features such as customizable UI options, voice recognition, and high-contrast themes.

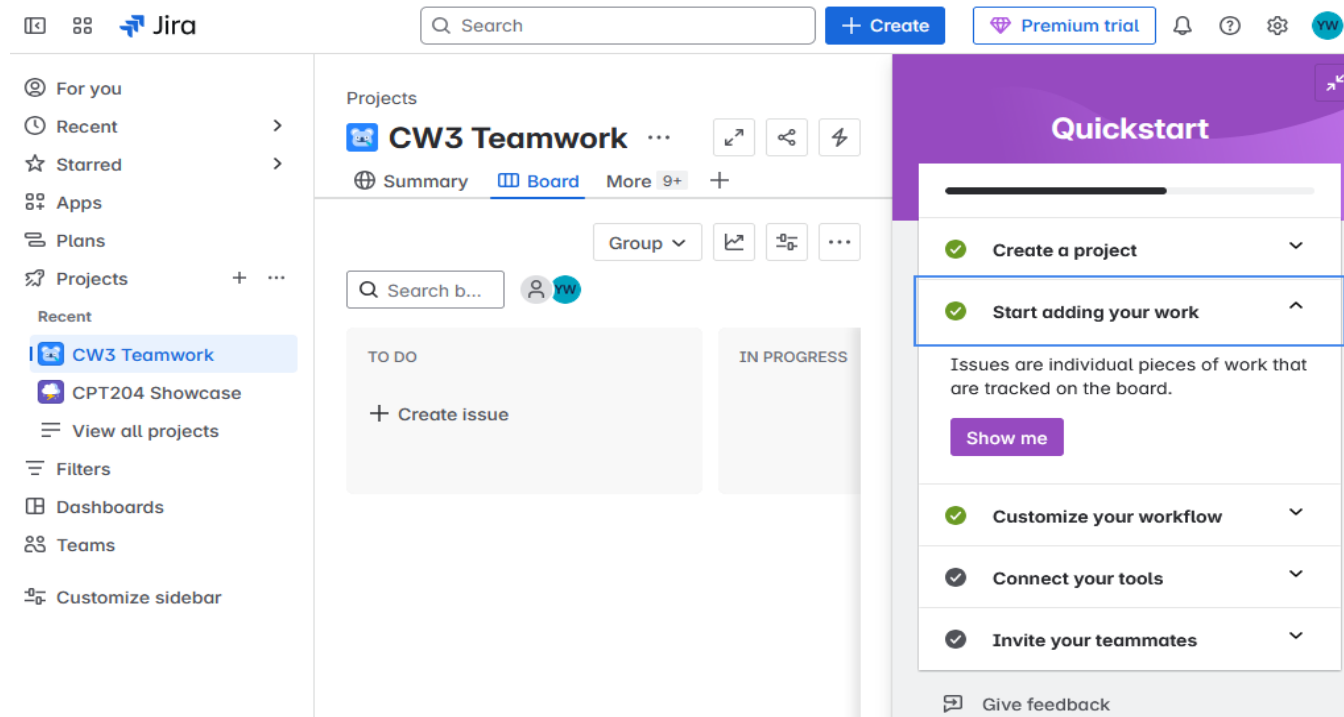
Encourage Collaboration Across Diverse Teams:

- Actively seek out team members from diverse backgrounds and experiences. This could mean promoting women in tech, supporting LGBTQ+ individuals, or working with teams that include people from different ethnicities and cultures.

Leverage EDI-Focused Resources:

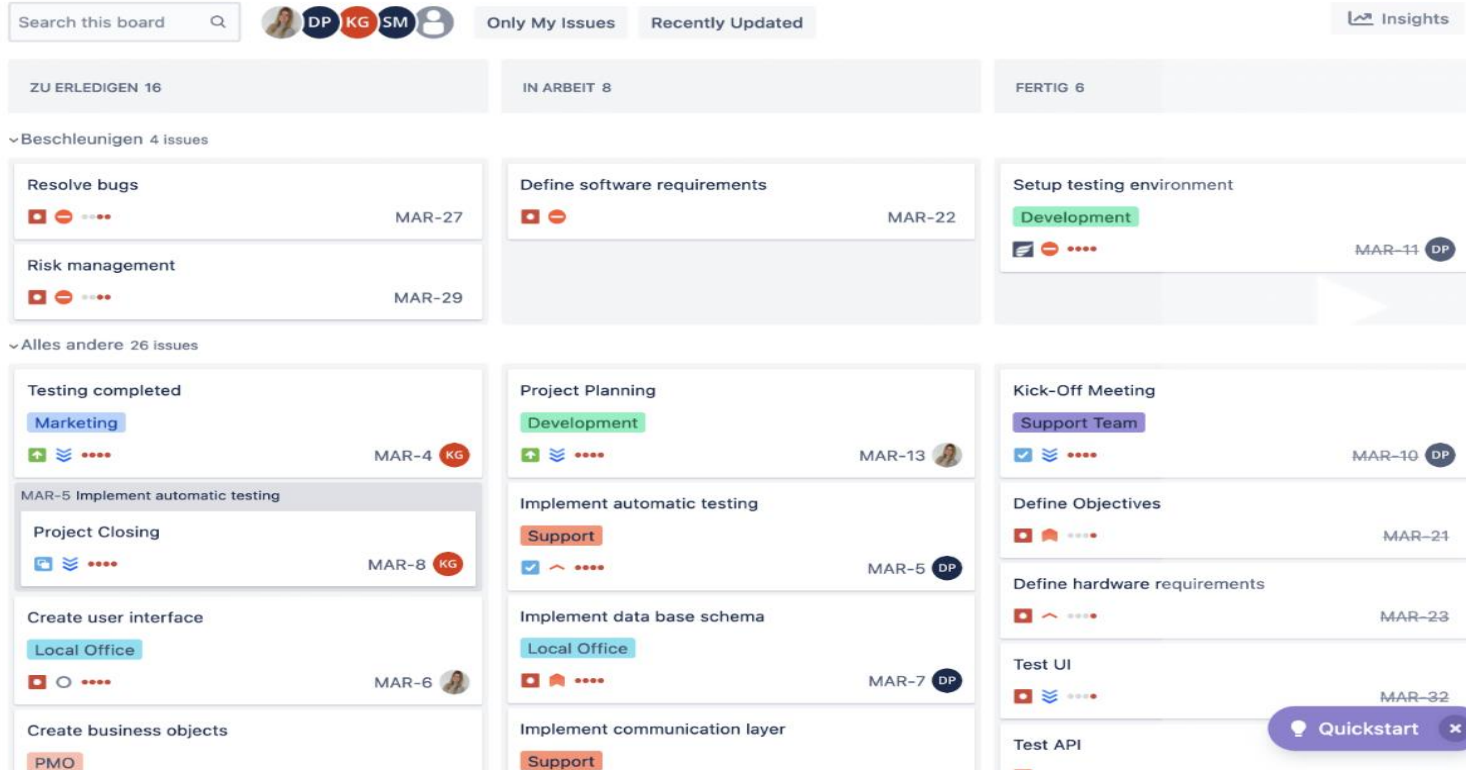
- Utilize resources, courses, and materials that focus on diversity and inclusion in the software development community. These resources help developers become more aware of potential biases in their coding practices and provide strategies to mitigate them.

Development with the AI-assisted Management Tools



- Develop advanced software components as a member or a leader of a software development team, **incorporating AI-assisted features.** (LO-C)
- Taking JIRA as an example, go through the **QuickStart** step by step

Development with the AI-assisted Management Tools



- **Basic Requirements:**
 - Show your planning board
 - Make it detailed
 - Screenshots+descriptions should just work