Design and Analysis of Algorithms

Lecture 02 – Sorting and Data Structures Designed for Sorting

Where we left off…

An Example – Gossips

- If you wanted to simulate gossip propagation.
  - What would you be interested in? How would you measure it?
  - What kind of data structure would you need?
  - What kind of rules (algorithm) would you develop?

Sorting

Sorting is a fundamental and very important task in both scientific and commercial data processing.

Sorting is the process of rearranging a sequence of objects so as to put them in some logical order.

- To discuss sorting, we need to define “the logical order” in the first place.
  - Numbers are easy to sort ascending or descending.
  - Text sorting can be defined alphabetically. But the alphabet order is based on your computer’s settings. Turkish language sorting is always problematic because the letters are split into two different tables in the encoding.
  - Complex data types with multiple fields can prioritize fields so that we can sort “by field 1, then by field 2”, etc.

Sorting

There are more than one ways to define an order.

Partial ordering is, essentially the operator <=. If both a <= b and b <= a then you may say that a is equivalent to b. Note that, with only a single operator <= available, there could be cases where neither a <= b nor b <= a. This means that **the two elements are incomparable.**

Strict weak ordering works like the operator <. If a < b then it is impossible to have b < a as well. If neither a < b nor b < a (ie. they are both false) then we are sure that they are equivalent.

Total ordering works with an additional operator == so that **we can check equivalence directly in addition to strict weak ordering.**

Sorting

Sorting works with a strict weak ordering. However, you would benefit greatly from having total ordering.

Both strict weak ordering and total order relationship can infer: “If A > B and B > C then A > C.”

- Number sorting and text sorting have total order relationships.
- For sorting custom data types, you would need to define these relationships yourself.

Sorting

In this session we will introduce and discuss various fundamental sorting algorithms and practice with pencil and paper to sort some short arrays.

Then we will discuss

- How an algorithm would perform better or worse on some practical cases of data.
- If a data structure **that is guaranteed to be sorted at all times** can be designed.
- If a data structure that is sorted from time to time is better than being sorted at all times.

There are a lot of animations or videos showing how these algorithms work.

- Try this one – https://www.toptal.com/developers/sorting-algorithms

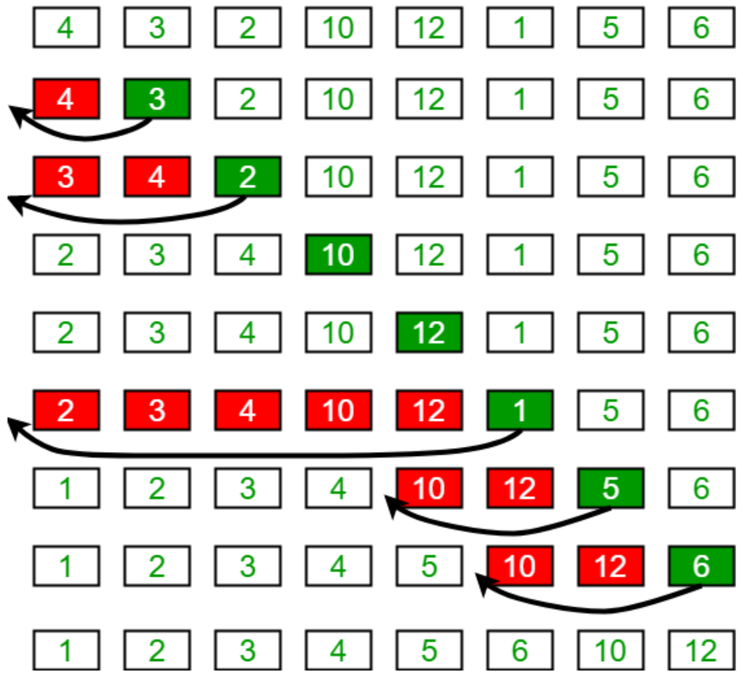
Sorting

Insertion Sort (on an array)

- You visualize there are two parts of the array, the sorted part (at the beginning) and the unsorted part.
- At the beginning of the algorithm, the first item in the array is itself the sorted part. As there is only one item, it is easy to assume it's sorted.
- From this point on, we repeat the following procedure.
  - We pick the first item on the "unsorted" part. It is also the next item just after the sorted part.
  - Then we try to insert it into the "correct" place in the sorted part.
  - Naturally, once we find the place to insert it, the part after the "insertion point" needs to be shifted.
  - Once shifting is done, the "sorted" part grows by one item. And the "unsorted" part shrinks by one item.
- We stop the procedure when the "unsorted" part has a size of zero.
- This is similar to how most people order a deck of cards.

Sorting

Insertion Sort Execution Example



Sorting

Insertion Sort (in Java)

In Java, the operators <, >, etc work only on numbers.

For objects of classes, we sort only data types which implement the **Comparable interface.** The comparable interface has the method compareTo(). This method compares one object against another and returns a result on the comparison. – https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html

Using the Comparable interface requires you to be modifying the class definition. If you cannot do it, then you define a new class that has the ability to compare items of the class. This new class is a utility class which implements the **Comparator interface.** – https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html

Insertion Sort (in C++)

In C++ the **operators <, >, etc can be overloaded** so that they are used for any type. However, overloading requires you add the overloading definitions as methods in the class. So you need to be able to modify the class definition.

If you cannot modify the class definition, you can **define a comparison function**, and then use std::sort library function by passing the comparison function as a parameter. See – https://en.cppreference.com/w/cpp/algorithm/sort and also https://en.cppreference.com/w/cpp/named_req/Compare

Note that the Java and C++ approaches are pretty much the same in idea.

They both **require strict weak ordering**.

They both benefit from total ordering (by implementing equals() in Java or overloading operator== in C++)

Sorting

Insertion Sort (in Java)

First **can we make sure** that an array of Comparable objects is sorted?

Code from our textbook. How does this work?

```
private static boolean isSorted(Comparable[] a, int lo, int hi) {
        for (int i = lo + 1; i < hi; i++)
            if (less(a[i], a[i-1])) return false;
        return true;
    }
```

Sorting

Insertion Sort (in Java)

Then comes how to implement the insertion sort itself.

Two important operations: (1) checking for the less than relationship, and (2) exchanging (swapping) two items.

Code from our textbook. How does this work?

```
private static boolean less(Comparable v, Comparable w) {
        return v.compareTo(w) < 0;
    }
private static void exch(Object[] a, int i, int j) {
        Object swap = a[i];
        a[i] = a[j];
        a[j] = swap;
    }
```

Sorting

Insertion Sort (in Java)

Then comes how to implement the insertion sort itself.

Code from our textbook. How does this work?

```
public static void sort(Comparable[] a) {
        int n = a.length;
        for (int i = 1; i < n; i++) {
            for (int j = i; j > 0 && less(a[j], a[j-1]); j--) {
                exch(a, j, j-1);
            }
            assert isSorted(a, 0, i);
        }
        assert isSorted(a);
    }
```

Sorting

Insertion Sort (discussion)

- How many times do we compare each unsorted item? All items?
  - Best case, once because it is already in position. All items, n obviously.
  - Worst case, the number of items before it. All items, 1 + 2 + …. + n-1, so it is related to n^2.
  - On average, somewhere in the middle, but still bounded by the n^2.
- So insertion sort needs O(n^2) comparisons.
  - For a straightforward explanation of the big-o notation please read – https://www.freecodecamp.org/news/big-o-notation-why-it-matters-and-why-it-doesnt-1674cfa8a23c/
- How about exchanges/swaps?
  - For each item one less than the number of comparisons.
  - For all items "n" less.
- So insertion sort needs O(n^2 – n) exchanges, which is still O(n^2).
- **We do not like O(n^2) unless n is sufficiently small.**

Sorting

Insertion Sort (discussion)

If we reduce the number of comparisons required to find the “correct place” for each item, we will be very happy.

Binary Insertion Sort

We can use **binary search** to reduce the number of comparisons in normal insertion sort.

Our assumption prior to searching for the correct place for the next unsorted item is that **the sorted part is already sorted**.

This allows us to use binary search to find the proper location to insert the selected item at each iteration.

This way we can **reduce O(n^2) to O(log n)**.

How does this work?

Sorting

Binary Insertion Sort (implementation in Java)

Code from our textbook. How does this work?

```
public static void sort(Comparable[] a) {
        int n = a.length;
        for (int i = 1; i < n; i++) {
            // binary search to determine index j at which to insert a[i]
            Comparable v = a[i];
            int lo = 0, hi = i;
            while (lo < hi) {
                int mid = lo + (hi - lo) / 2;
                 if (less(v, a[mid])) hi = mid;
                 else                 lo = mid + 1;
            }
            // insertion sort with "half exchanges"
            // (insert a[i] at index j and shift a[j], ..., a[i-1] to right)
            for (int j = i; j > lo; --j)
                a[j] = a[j-1];
            a[lo] = v;
        }
        assert isSorted(a);
}
```

Sorting

Binary Insertion Sort (discussion)

We now have a O (log n) algorithm, right? Not necessarily.

In the worst case (the array was in reverse order) the technique's performance degenerates a bit, into O(n log n) comparisons.

We only reduced the number of comparisons. But not the number of exchanges/swaps. **The swaps are still at O(n^2) level.**

So what is the cost of a comparison compared to that of a swap?

What are we comparing? How are we comparing?

If the comparisons are between numbers or strings, then the comparison is very fast. Otherwise, we may not be so sure.

In Java, **all items are referenced**, then the swaps are very fast. Why?

If the items themselves are to be swapped, and they are large items, we might run into **problems with memory allocation**. Why?

What if we are sorting on disk? Is there any difference?

Sorting

Merge Sort

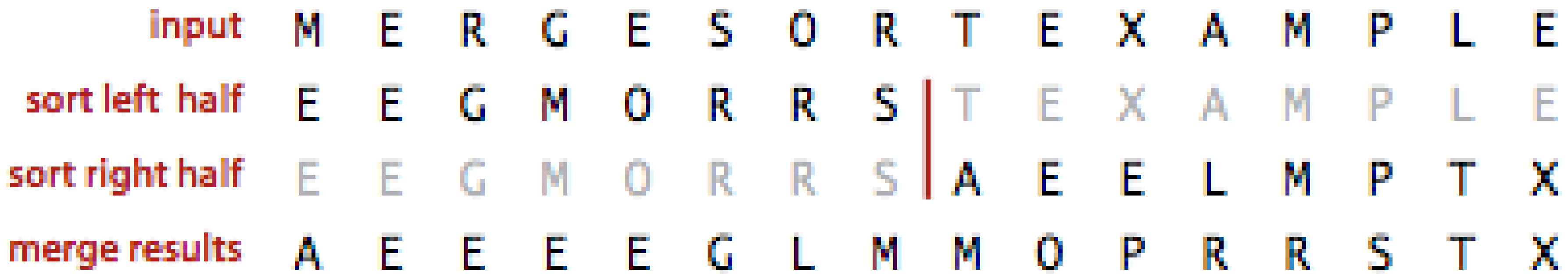
Assume we had two arrays of same size (or just 1 difference), and they were already sorted. Then we could very quickly merge the second array into the first array.

Any array of size 1 is already sorted.

By principle of induction.

- For any array of size 2, we have two "halves" of sizes 1, that are already sorted. We can merge the second array into the first one to construct a sorted array of size 2.
- For any array of size 3, we have two "halves" of size 1 and 2, that are sorted or can be sorted. We can sort the array of size 2, and then merge.
- …
- For any array of size n, we have two "halves" ….. we can sort both halves and the merge.

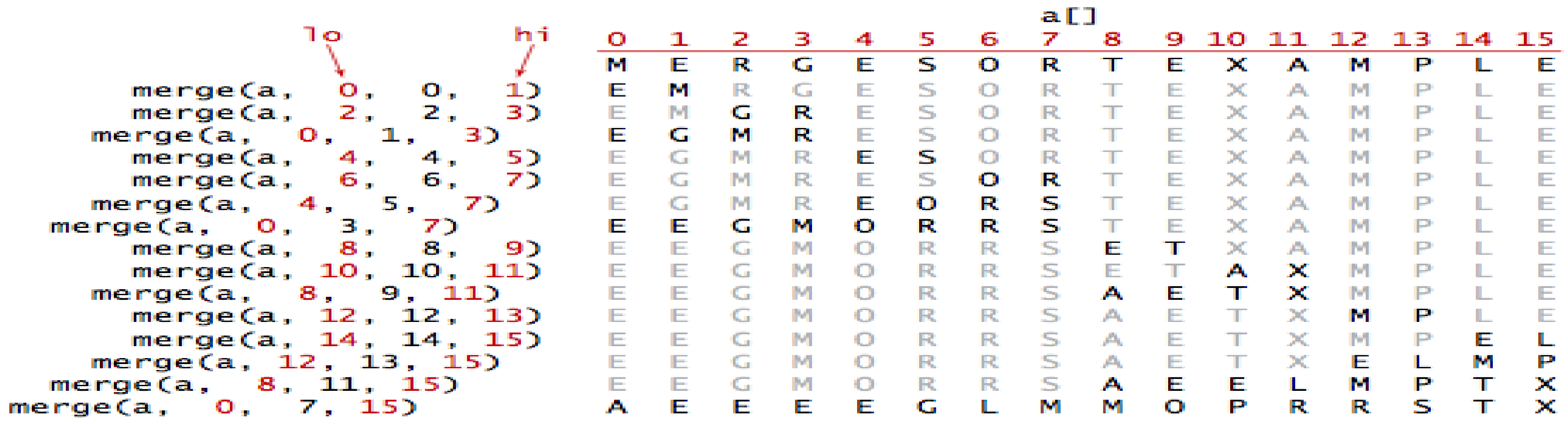
Sorting



input	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sort left half	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
sort right half	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge results	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Mergesort overview

Sorting



	a[]															
lo hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 0, 0, 1)	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 2, 2, 3)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 0, 1, 3)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 4, 4, 5)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 6, 6, 7)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 4, 5, 7)	E	G	M	R	E	O	R	S	T	E	X	A	M	P	L	E
merge(a, 0, 3, 7)	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
merge(a, 8, 8, 9)	E	E	G	M	O	R	R	S	E	T	X	A	M	P	L	E
merge(a, 10, 10, 11)	E	E	G	M	O	R	R	S	E	T	A	X	M	P	L	E
merge(a, 8, 9, 11)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a, 12, 12, 13)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a, 14, 14, 15)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	E	L
merge(a, 12, 13, 15)	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a, 8, 11, 15)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge(a, 0, 7, 15)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Trace of merge results for top-down mergesort

Sorting

Merge Sort

- Merge sort uses the <u>**principle of recursion**</u> which is the programmatic equivalent of mathematical inductive proofs.
- However, this algorithm requires <u>**additional memory space that is equivalent to the array**</u>. Why?
    - Can you identify the need for the additional memory when you are working on pencil-paper examples?

Sorting

Merge Sort (Implementation in Java)

First of all, how do we **merge** two sorted arrays?

Code from our textbook. How does this work?

```
// stably merge a[lo .. mid] with a[mid+1 ..hi] using aux[lo .. hi]
    private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi) {
        // precondition: a[lo .. mid] and a[mid+1 .. hi] are sorted subarrays
        assert isSorted(a, lo, mid);
        assert isSorted(a, mid+1, hi);
        // copy to aux[]
        for (int k = lo; k <= hi; k++) {
            aux[k] = a[k];
        }
        // merge back to a[]
        int i = lo, j = mid+1;
        for (int k = lo; k <= hi; k++) {
            if      (i > mid)              a[k] = aux[j++];
            else if (j > hi)               a[k] = aux[i++];
            else if (less(aux[j], aux[i])) a[k] = aux[j++];
            else                           a[k] = aux[i++];
        }
        // postcondition: a[lo .. hi] is sorted
        assert isSorted(a, lo, hi);
    }
```

Sorting

Merge Sort (Implementation in Java)

First of all, how do we merge two sorted arrays?

Code from our textbook. How does this work?

```
// mergesort a[lo..hi] using auxiliary array aux[lo..hi]
    private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
        if (hi <= lo) return;
        int mid = lo + (hi - lo) / 2;
        sort(a, aux, lo, mid);
        sort(a, aux, mid + 1, hi);
        merge(a, aux, lo, mid, hi);
    }
```

Sorting

Merge Sort (Discussion)

What is the computational cost of the merge operation? (n)

How many times do we merge? (log n)

So we end up with a **guaranteed** O (n log n). See for details. – https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/analysis-of-merge-sort

Sorting

Merge Sort (Discussion)

- Optimizations (all in favor of number of operations, nothing to be done about additional memory usage)
  - For smaller arrays, switch to insertion sort.
  - Always check if an array is already sorted.
  - Switch the auxiliary array with the original array to get smaller number of copying.

Sorting

Merge Sort (Discussion)

- Suppose you have an array of n items which is nearly sorted. Each item at most k positions away from its position in the sorted order.
  - How would merge sort perform for this type of input?
- In Java and many other languages, arrays contain references so that the memory usage of an array (itself) is predictable.
  - Anything larger than a single number is larger than a single reference in memory.
  - Using references can help manage the memory usage of merge sort.
  - How important would this be for practical use?

Sorting

Quick Sort

Quick sort rivals insertion sort in its popularity.

Similar to merge sort, it is a divide-and-conquer method for sorting. It works by partitioning an array into two parts, then sorting the parts independently.

However, the way quick sort partitions the array into two parts is very different.

- **Select an index j**, so that a[j] would be in its "correct place".
- This **implies** for all entries from a[0] up to a[j-1] the entries are **not greater than a[j] (ie. less than or equal to)**, and at the same time, entries from a[j+1] up to a[n-1] would be **not less than a[j] (ie. greater than or equal to)**.

A complete sort is done by partitioning, then recursively applying the method to the partitions.

Note that the algorithm randomly shuffles the array before sorting it.

Sorting

Quick Sort

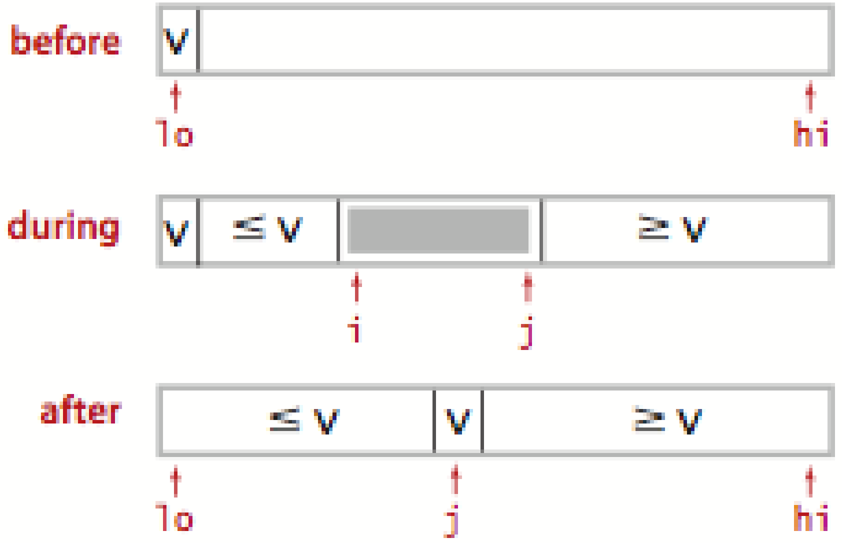
Partitioning is very important in quick sort.

First, we arbitrarily choose a[lo] to be the partitioning item—the one that will go into its final position.

Next, we scan from the left end of the array until we find an entry that is greater than (or equal to) the partitioning item.

Then we scan from the right end of the array until we find an entry less than (or equal to) the partitioning item.

The two items that stopped the scans are out of place in the final partitioned array, so we exchange them.



Quicksort partitioning overview

Sorting

Quick Sort

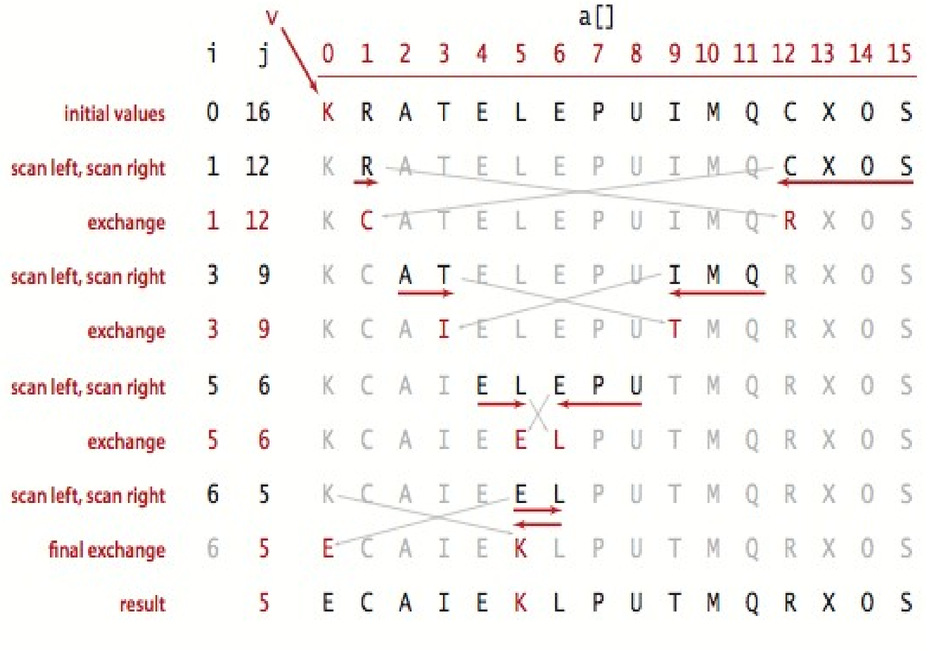
Partitioning is very important in quick sort.

First, we arbitrarily choose a[lo] to be the partitioning item—the one that will go into its final position.

Next, we scan from the left end of the array until we find an entry that is greater than (or equal to) the partitioning item.

Then we scan from the right end of the array until we find an entry less than (or equal to) the partitioning item.

The two items that stopped the scans are out of place in the final partitioned array, so we exchange them.



			a[]															
	i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initial values	0	16	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
scan left, scan right	1	12	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
exchange	1	12	K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S
scan left, scan right	3	9	K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S
exchange	3	9	K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S
scan left, scan right	5	6	K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S
exchange	5	6	K	C	A	I	E	E	L	P	U	T	M	Q	R	X	O	S
scan left, scan right	6	5	K	C	A	I	E	E	L	P	U	T	M	Q	R	X	O	S
final exchange	6	5	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
result		5	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S

Partitioning trace (array contents before and after each exchange)

Sorting

Quick Sort (Discussion)

- Implementations vary by the way they select “j” for partitioning.
- Some things to consider when implementing partitioning.
  - If the smallest item or the largest item in the array is the partitioning item, we have to take care that the pointers (indexes i and j) do not run off the left or right ends of the array, respectively.
  - Properly testing whether the pointers have crossed is a bit trickier than it might seem at first glance. A common error is to fail to take into account that the array might contain other keys with the same value as the partitioning item.
- Quick sort is O (n log n).
  - Usually there are far less exchanges than there are compares.
  - Most implementations switch to insertion sort for shorter arrays.
- Median-of-three partitioning.
  - An easy way to improve the performance of quick sort is to use the median of a small sample of items taken from the array as the partitioning item.

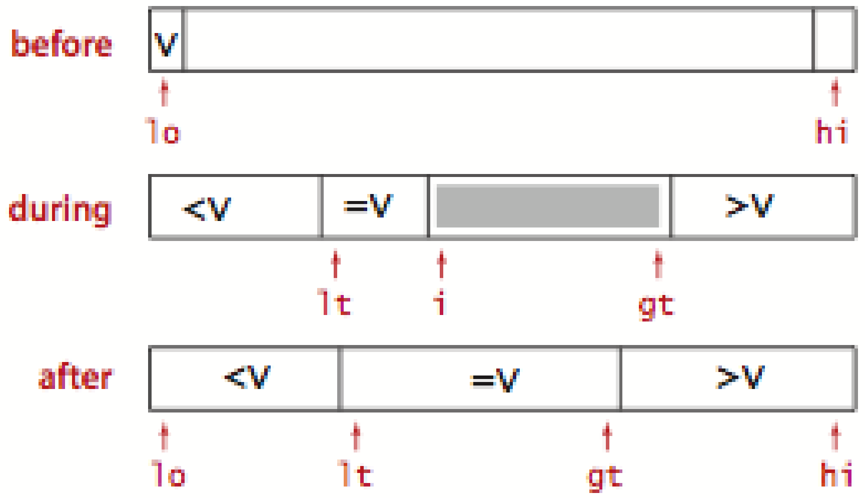
Sorting

Quick Sort (Discussion)

In cases where there are multiple items which will be equal, three-way partitioning is superior.

In three-way partitioning, we acknowledge the existing equal items and use three indexes instead of two to define an additional partition for those.

Java's default sorting method Arrays.sort() uses three-way partitioned quick sort for primitive types (ie. numbers) and merge sort for reference types (all objects, such as strings).



3-way partitioning overview

Sorting

Quick Sort (Implementation)

```
private static void sort(Comparable[] a,
int lo, int hi) {
    if (hi <= lo) return;
    int j = partition(a, lo, hi);
    sort(a, lo, j-1);
    sort(a, j+1, hi);
    assert isSorted(a, lo, hi);
  }
```

Sorting

Quick Sort (Partitioning)

```
private static int partition(Comparable[] a, int lo, int hi) {
      int i = lo;
      int j = hi + 1;
      Comparable v = a[lo];
      while (true) {
         // find item on lo to swap
         while (less(a[++i], v)) {
            if (i == hi) break;
         }
         // find item on hi to swap
         while (less(v, a[--j])) {
            if (j == lo) break;      // redundant since a[lo] acts as sentinel
         }
         // check if pointers cross
         if (i >= j) break;
         exch(a, i, j);
      }
      //... omitted (next slide)
   }
```

Sorting

Quick Sort (Partitioning, cnt’d)

```
private static int partition(Comparable[]
a, int lo, int hi) {
        // omitted (see previous slide)
        // put partitioning item v at a[j]
        exch(a, lo, j);
        // now, a[lo .. j-1] <= a[j] <=
a[j+1 .. hi]
        return j;
    }
```

Sorting

Quick Sort (Another Implementation)

```
private static void sort(Comparable[] a, int lo, int
hi) {
    if (hi <= lo) return;
    exch(a, lo, (lo + hi) / 2);
    // use middle element as partition
    int last = lo;
    for (int i = lo + 1; i <= hi; i++)
        if (less(a[i], a[lo])) exch(a, ++last, i);
    exch(a, lo, last);
    sort(a, lo, last-1);
    sort(a, last+1, hi);
  }
```

Sorting

Sorting Stability

Sorting stability means that **records with the same key** (ie. that are considered equal) retain their relative order before and after the sort.

What does this actually mean? An example. Sorting by last names only.

- "Güngören, Bora" and "Güngören, Tufan" are considered equal.
- So a stable algorithm does not exchange/swap these items.
- Insertion sort and merge sort are both stable. Quick sort is not stable.

Stability matters, **if and only if**, the problem you're solving requires retention of that relative order.

Stability also matters if swapping operations may be prohibitive.

Sorting

In-place Sorting

In-place sorting means that for our algorithm, apart form a few variables, there is no additional space needed for sorting.

Out-of-place sorting means that our algorithm allocates temporary space with size similar o the original data set.

Data Structures Designed for Sorting

So far all our discussion has been about arrays. But we know there are other types of data structures.

Based on the requirements of our application, a data structure can be designed to better suit our sorting requirements.

- Do we need keys in full sorted order at all times?
- Do we need to access the minimum or maximum valued item only, or do we access any item?
- How do we collect items? Are they added one a at a time or as groups?

Data Structures Designed for Sorting

An application example – Priority Queues.

Many applications require that we process items having keys in order, but not necessarily in full sorted order and not necessarily all at once.

With some intervals, we process the item with the largest key, also removing at the same time from the group.

Also with some intervals, we add items to the group.

Therefore at any time, we only need to know the item with the largest key.

We can implement this in may ways.

Arrays (sorted) – When we add items, we use binary search to find the correct place (as in merge sort). The item with the largest key is at the end, hence easy to locate and remove.

Arrays (unsorted) – When we add items, we simply compare that with the item with the largest key. If the new item is larger, then we swap. The item with the largest key is in its place. When we remove, we find the largest item and exchange it to place in the correct position.

Linked lists (singly or doubly linked). – We find the correct place and insert in-between. No swaps are required.

Data Structures Designed for Sorting

Priority Queues using binary trees

If we implement the queue using a tree, many operations are easier.

An interesting tree is that of a **heap-ordered binary tree**. With heap-ordering, the item in a node is greater than its children.

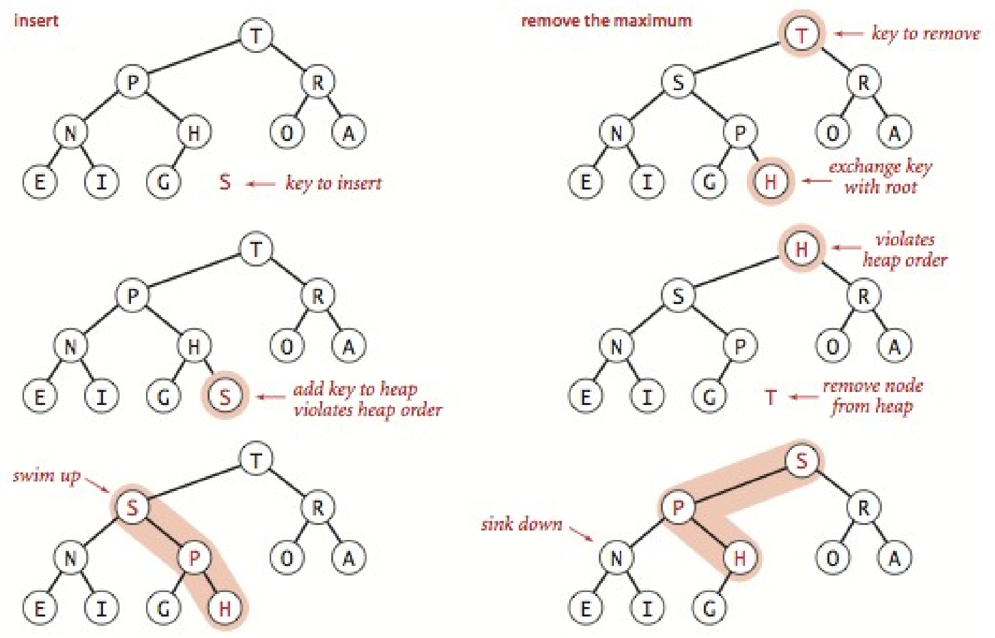
Therefore the largest item is at the root of the tree.

When we need to remove the largest item, it is an O(1) operation to locate the item. So the **response time** for the request for the largest item is very short.

Then we re-balance the tree within the interval of waiting.

We have the terms (1) swimming up and (2) sinking down in re-balancing.

Data Structures Designed for Sorting



Heap operations

Data Structures Designed for Sorting

Priority Queues Using Binary Trees (Discussion)

- Each time we add or remove an item, there is an O(log n) operation.
- This gives us **predictability in terms of the completion time** of the operation.
- Adding n items, we end up with O (n log n) to sort an arbitrary array bu constructing a heap-ordered tree.

Heap based algorithms are unstable with respect to sorting.

Data Structures Designed for Sorting

General Discussion

Many graph algorithms rely on sorting to find the least-costly / most-awarding path.

Combinatorial optimization algorithms use sorting algorithms to efficiently solve their sub-problems.

String processing applications rely on sorting.

In commercial applications, the latest transactions are usually more important, so having data sorted at all times increases performance.

Data Structures Designed for Sorting

Assignment 1 (A Prioritized Event Simulation)

Consider ABC Bank which is a typical deposit bank. Therefore the main task is to handle money transactions (EFT, or deposit/withdrawal at cashier).

- Each money transaction request is first handled by a load balancer (LB) that forwards the transaction to an available server.
- Each transaction takes time t, a function of the size of the transaction.
- Between two transactions, servers need some random duration to re-order their internal records.
- There are S servers serving banking transactions.
- There are queues inside each server, so that each Server can be assigned a number of transactions. However, the load balancer also has its own queue, so that servers' queues need not be utilized.
- All servers update their queues as they process requests.

ABC Bank so far has been assigning new requests on the least busy server. This is defined as the server with least number of active requests assigned. In case of equality, a random server is selected.

Data Structures Designed for Sorting

Assignment 1 (A Prioritized Event Simulation)

<u>Questions</u>

1. Try to model the load balancer's server selection process. Sketch the data structure for the problem.

2. Is there a sorting-related problem here?

3. Which type of algorithm would be better suited for this problem? Why?

4. Suppose ABC Bank decided to implement a "high priority customer" program. In this program these high priority customers' transactions are processed earlier than those of the ordinary customers. How would you handle this change in your data structure and your algorithm?

5. Suppose banking regulations now enforce all banks to prioritize transactions that have been waiting the longest duration. How would you handle this change in your data structure and your algorithm?

6. How would you classify the current design and your proposed designs in terms of sorting stability?

<u>Note:</u> In this assignment you are not required to code in Java.