CSE 373
Data Structures & Algorithms

Lecture 13

Sorting (I)

Chapter 7 in Weiss

Sorting

- Input
  - an array A of data records
  - a key value in each data record
  - a comparison function which imposes a consistent ordering on the keys
- Output
  - reorganize the elements of A such that
    - For any i and j, if $i < j$ then $A[i] \leq A[j]$

Consistent Ordering

The comparison function must provide a consistent *ordering* on the set of possible keys

- You can compare any two keys and get back an indication of a < b, a > b, or a = b (trichotomy)
- The comparison functions must be consistent
  - If **`compare(a,b)`** says a<b, then **`compare(b,a)`** must say b>a
  - If **`compare(a,b)`** says a=b, then **`compare(b,a)`** must say b=a
  - If **`compare(a,b)`** says a=b, then **`equals(a,b) and equals(b,a)`** must say a=b

Why Sort?

- Allows binary search of an N-element array in O(log N) time
- Allows O(1) time access to *k*th largest element in the array for any *k*
- Sorting algorithms are among the most frequently used algorithms in computer science

Space

- How much space does the sorting algorithm require in order to sort the collection of items?
  - Is copying needed?
    - **In-place** sorting algorithms: no copying or at most O(1) additional temp space.
  - External memory sorting – data so large that does not fit in memory

Stability

A sorting algorithm is **stable** if:

– Items in the input with the same value end up in the same order as when they began.

Input		Unstable sort		Stable Sort	
Adams	1	Adams	1	Adams	1
Black	2	Smith	1	Smith	1
Brown	4	Washington	2	Black	2
Jackson	2	Jackson	2	Jackson	2
Jones	4	Black	2	Washington	2
Smith	1	White	3	White	3
Thompson	4	Wilson	3	Wilson	3
Washington	2	Thompson	4	Brown	4
White	3	Brown	4	Jones	4
Wilson	3	Jones	4	Thompson	4

[Sedgewick]

Time

How fast is the algorithm?

- The definition of a sorted array A says that for any i<j, A[i] ≤ A[j]
- This means that you need to at least check on each element at the very minimum
  - Complexity is at least:
- And you could end up checking each element against every other element
  - Complexity could be as bad as:

The big question is: How close to $O(n)$ can you get?

Sorting: *The Big Picture*

Given n comparable elements in an array, sort them in an increasing order.

Simple algorithms: $O(n^2)$	Fancier algorithms: $O(n \log n)$	Comparison lower bound: $\Omega(n \log n)$	Specialized algorithms: $O(n)$	Handling huge data sets
Insertion sort Selection sort Bubble sort	Heap sort Merge sort Quick sort …		Bucket sort Radix sort	External sorting

Selection Sort: idea

1. Find the smallest element, put it 1st
2. Find the next smallest element, put it 2nd
3. Find the next smallest, put it 3rd
4. And so on …

Try it out: Selection Sort

- 31, 16, 54, 4, 2, 17, 6

Selection Sort: Code

```
void SelectionSort (Array a[0..n-1]) {
    for (i=0; i<n; ++i) {
        j = Find index of
                smallest entry in a[i..n-1]
        Swap(a[i],a[j])
    }
}
```

Runtime:

worst case :

best case :

average case :

Bubble Sort Idea

- Take a pass through the array
  - If neighboring elements are out of order, swap them.
- Take passes until no swaps needed.

Try it out: Bubble Sort

- 31, 16, 54, 4, 2, 17, 6

Bubble Sort: Code

```
void BubbleSort (Array a[0..n-1]) {
    swapPerformed = 1
    while (swapPerformed) {
        swapPerformed = 0
         for (i=0; i<n-1; i++) {
           if (a[i+1] < a[i]) {
                 Swap(a[i],a[i+1])
                 swapPerformed = 1
             }
         }
    }
}
```

Can we decrease this ?

Runtime:

worst case :

best case :

average case :

Bubble Sort: Code

```
void BubbleSort (Array a[0..n-1]) {
    swapPerformed = 1
    while (swapPerformed) {
        swapPerformed = 0
         for (i=0; i < --n; i++) {
           if (a[i+1] < a[i]) {
                 Swap(a[i],a[i+1])
                 swapPerformed = 1
             }
         }
    }
}
```

Why

Can you do even better ?

Bubble Sort: Code

```
void BubbleSort (Array a[0..n-1]) {
    m = n-1
    while (m > 0) {
        lastSwap = 0
         for (i=0; i<m; i++) {
           if (a[i+1] < a[i]) {
                 Swap(a[i],a[i+1])
                 lastSwap = i
             }
         }
        m = lastSwap
    }
}
```

Insertion Sort: Idea

1. Sort first 2 elements.
2. Insert 3rd element in order.
   - (First 3 elements are now sorted.)
3. Insert 4th element in order
   - (First 4 elements are now sorted.)
4. And so on…

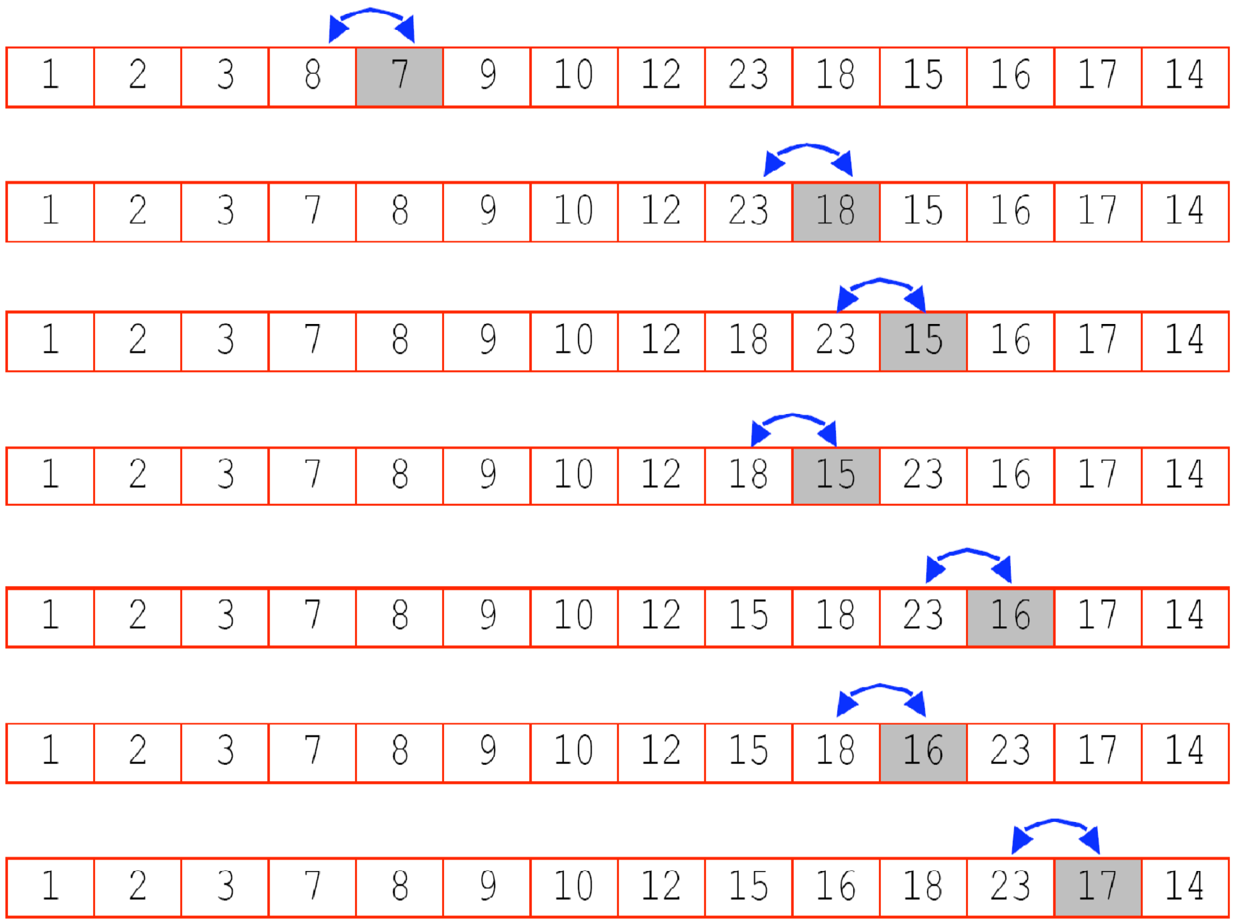
How to do the insertion?

Suppose my sequence is:

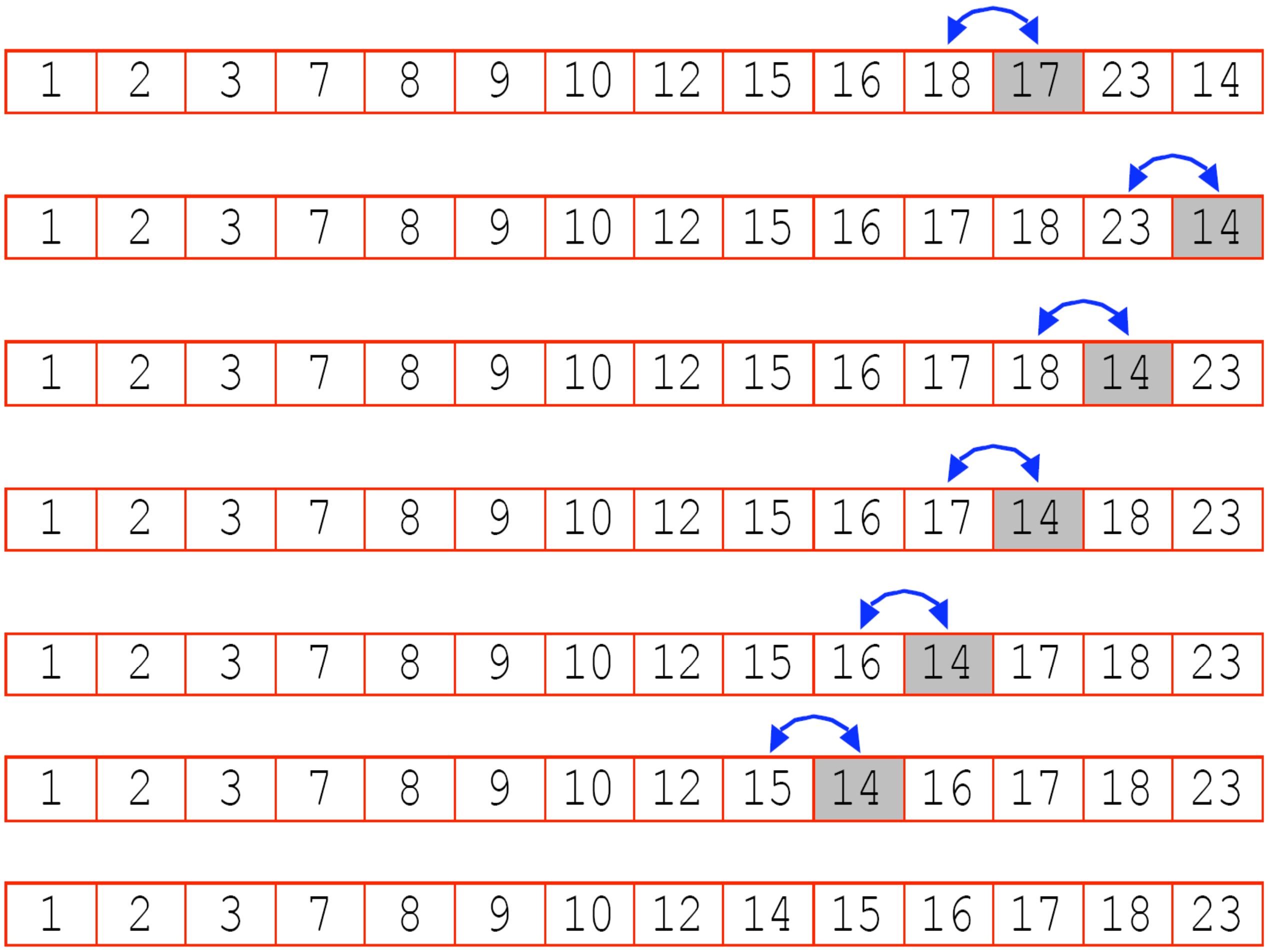
16, 31, 54, 78, 32, 17, 6

And I've already sorted up to 78. How to insert 32?

Example: Insertion Sort



Example: Insertion Sort



Try it out: Insertion sort

- 31, 16, 54, 4, 2, 17, 6

Insertion Sort: Code

```
void InsertionSort (Array a[0..n-1]) {
    for (i=1; i<n; i++) {
        for (j=i; j>0; j--) {
            if (a[j] < a[j-1])
              Swap(a[j],a[j-1])
            else
                break
    }

}
```

Runtime:

worst case :
best case :
average case :

Note: can instead move the "hole" to minimize copying, as with a binary heap.

Sort with AVL Tree

Runtime:

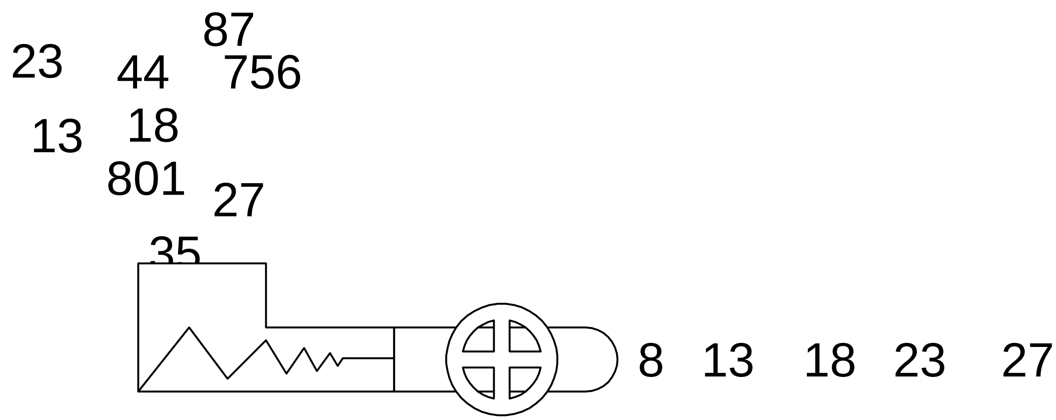
Try it out: Sort with AVL Tree

- 31, 16, 54, 4, 2, 17, 6

HeapSort

Runtime:

HeapSort



Shove all elements into a priority queue, take them out smallest to largest.

Runtime:

Try it out: HeapSort

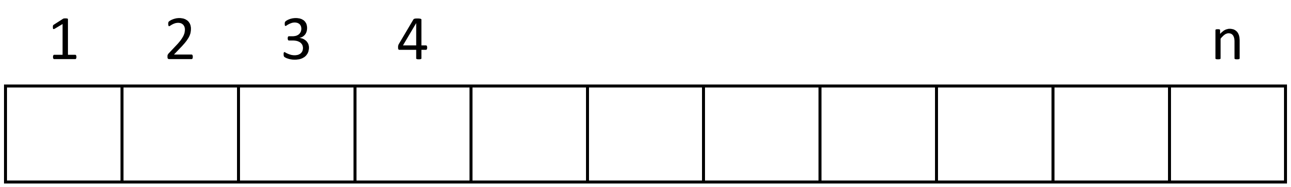
- 31, 16, 54, 4, 2, 17, 6

In Place HeapSort

1. Build Heap

2. Repeat:
   – DeleteMax and place it on the last leaf

Note: array entries are numbered 1..n !



HeapSort: Step 1

```
private void buildHeap(int a[ ], int n) {
   for ( int i = n/2; i > 0; i-- ) {
    percolateDown(i, a[i]);
   }
}
```

Lecture 8

Note: need to place the MAXIMUM element on the root

HeapSort: Step 2

```
private void sort(int a[ ], int n) {
   buildHeap(a, n);
  while ( n > 0 ) {
     a[n--] = a[1];
    DeleteMax(a, n);
   }
}
```

Lecture 7