CSCI 335
Software Design and Analysis III

Sorting I

(Insertion sort, Shellsort, Heapsort, MergeSort, Quicksort), Quickselect,

Worst and average-case analysis

Visualizations

https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

Sorting algorithms

- Comparison-based sorting
- STL versions:

```
sort(v.begin(), v.end() );
sort(v.begin(), v.begin() + (v.end()-v.begin())/2);
sort(v.begin(),v.end(),greater<int>{ });
```

Also: stable_sort()

- Does not swap the positions of two elements with the same comparison key.

Insertion Sort

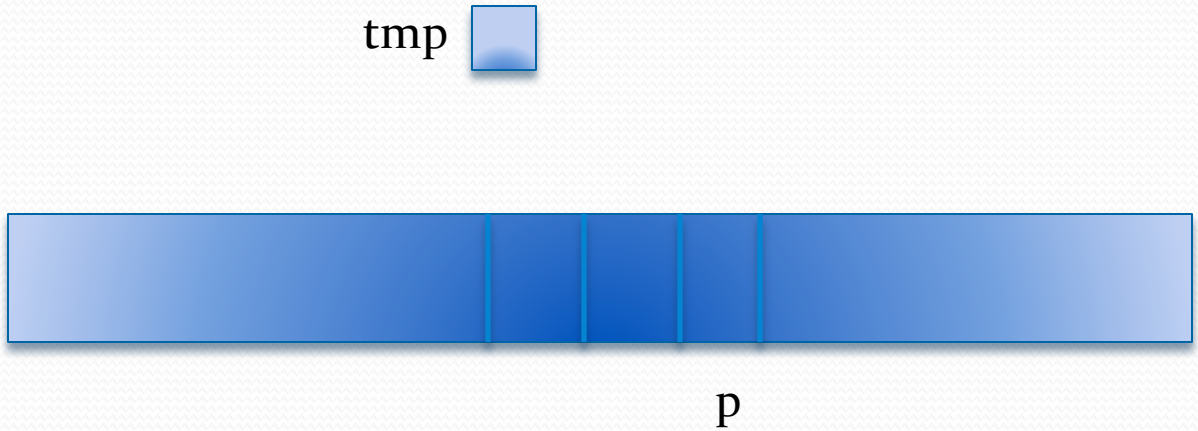
- Pass p=1 through N-1:

  items in positions 0 through p are in sorted order.

Original	34	8	64	51	32	21	Positions Moved
After $p = 1$	8	34	64	51	32	21	1
After $p = 2$	8	34	64	51	32	21	0
After $p = 3$	8	34	51	64	32	21	1
After $p = 4$	8	32	34	51	64	21	3
After $p = 5$	8	21	32	34	51	64	4

Insertion Sort

```
template <typename Comparable>
void InsertionSort(vector<Comparable> & a) {
    for (int p = 1; p < a.size(); ++p) {
        Comparable tmp = std::move(a[p]);
        int j;
        for (j = p; j > 0 && tmp < a[j - 1]; --j)
            a[j] = std::move(a[j - 1]);
        a[j] = std::move(tmp);
    }
}
```


tmp
p

Stl implementation

```
// Two-parameter version calls three-parameter version.
// Uses C++11 decltype.
template <typename Iterator>
void InsertionSort(const Iterator &begin, const Iterator &end) {
  InsertionSort(begin, end, less<decltype(*begin)>{});
}
```

Stl implementation

```
// Three-parameter version.
template <typename Iterator, typename Comparator>
void InsertionSort(const Iterator &begin, const Iterator &end, Comparator
                                                              less_than) {
  if (begin == end) return;
  for (Iterator p = begin + 1; p != end; ++p) {
    auto tmp = std::move(*p);
    Iterator j;
    for (j = p; j != begin && less_than(tmp, *(j - 1)); --j)
        *j = std::move(*(j - 1));
    *j = std::move(tmp);
}
---
You can call for instance:
InsertionSort(vec.begin(), vec.end()); // vec is of type vector<string>
```

Insertion Sort Analysis

- Worst case: $\Theta(N^2)$
- Best case (presorted input): $O(N)$
  - One comparison per item

Lower bound for simple sorting algorithms

Inversion in an array **a** is an ordered pair (i, j), 0 <= i, j <**a**.size():

i < j and **a**[i] > **a**[j]

Example array: 34, 8, 64, 51, 32, 21

9 inversions: (34,8), (34,32), (34,21),
(64,51), (64,32), (64,21),
(51,32),(51,21),
(32,21)

These are exactly the swaps performed by insertion sort!

=> Swapping two adjacent items that are out of place reduces one inversion

=> Sorted array needs no inversions

Lower bound for simple sorting algorithms

- Average number of inversions in permutation of N integers (assume no duplicates) provides

Average running time for insertion sort and for **all algorithms that are based on swapping adjacent elements !**

Lower bound for simple sorting algorithms

- **Theorem:** Average number of inversions in an array of n distinct elements is n(n-1)/4
- **Proof:**

  Consider list L of n integers and its reverse Lr.

  Total number of inversions for both lists is ?

  Average number of inversions is ?
- **Therefore:** Any algorithm that sorts by exchanging adjacent elements is **$\Omega(n^2)$ on average**.

Lower Bound for Simple Sorting

- *Theorem*. Average number of inversions in an array of n distinct elements is $\frac{n(n-1)}{4}$.

- *Proof*. Consider list L of n integers and its reverse L_r. For every pair of indicies (i, j), either (i, j) is an inversion in L or in L_r. Therefore the total number of inversions in both lists is $\binom{n}{2}$. The average number of insertions across all lists is $\frac{n!}{2n!}\binom{n}{2} = \frac{n(n-1)}{4}$.

- Thus, any algorithm that sorts by exchanging adjacent elements is $\Omega(n^2)$ on average.

Shellsort (by Donald Shell)

- Improves average time by exchanging non-adjacent elements
- **Diminishing increment sort**
- Uses increment sequence h1=1, h2, …, ht
  - All sequences work, but some perform better!
- Idea:

For k = t down to 1

[all elements spaced h_k apart are sorted] ==

[**a**[i] ≤ **a**[i+ h_k] for every i] ==

[perform insertion sort in h_k subarrays]

--each subarray has size approximately $\mathbf{n/h_k}$

Shellsort

Array index:	0	1	2	3	4	5	6	7	8	9	10	11	12
Original	81	94	11	96	12	35	17	95	28	58	41	75	15
After 5-sort	35	17	11	28	12	41	75	15	96	58	81	94	95
After 3-sort	28	12	11	35	15	41	58	17	94	75	81	96	95
After 1-sort	11	12	15	17	28	35	41	58	75	81	94	95	96

At increment h_k:

for every position $i=h_k, h_k+1, h_k+2, \ldots, N-1$

find its right spot among $i, i-h_k, i-2h_k, i-3h_k, \ldots$

Note that at h1 we have regular insertion sort

Shellsort

Array index:	0	1	2	3	4	5	6	7	8	9	10	11	12
Original	81	94	11	96	12	35	17	95	28	58	41	75	15
After 5-sort	35	17	11	28	12	41	75	15	96	58	81	94	95
After 3-sort	28	12	11	35	15	41	58	17	94	75	81	96	95
After 1-sort	11	12	15	17	28	35	41	58	75	81	94	95	96

For example for $h_k = 3$:

for i = 3, 3 + 1, 3 + 2, …, **a**.size() -1

find its right spot among i, i - 3, i – 6, i - 9 , … (i>=0)

5-sort: 5 “subarrays”

0	1	2	3	4	5	6	7	8	9	10	11	12
81	94	11	96	12	35	17	95	28	58	41	75	15
81					35					41		
35					**41**					**81**		
	94					17					75	
	17					**75**					**94**	
		11					95					15
		11					**15**					**95**
			96					28				
			28					**96**				
				12						58		
				12						**58**		
35	**17**	**11**	**28**	**12**	**41**	**75**	**15**	**96**		**58**	**81**	**94**	**95**

3-sort: 3 “subarrays”

0	1	2	3	4	5	6	7	8	9	10	11	12
35	17	11	28	12	41	75	15	96	58	81	94	95
35			28			75			58			95
28			**35**			**58**			**75**			**95**
	17			12			15			81		
	12			**15**			**17**			**81**		
		11			41			96			94	
		11			**41**			**94**			**96**	
28	**12**	**11**	**35**	**15**	**41**	**58**	**17**	**94**	**75**	**81**	**96**	**95**

1-sort: 1 “subarray”

0	1	2	3	4	5	6	7	8	9	10	11	12
28	12	11	35	15	41	58	17	94	75	81	96	95
11	**12**	**15**	**17**	**28**	**35**	**41**	**58**	**75**	**81**	**94**	**95**	**96**

- “subarrays” are conceptual (i.e. no need for extra storage)
- Implementation is extremely simple
- Best algorithm for small collections (up to 10000 elements)
- No recursion, good for embedded systems with small stack space.

Shellsort

- Original increment sequence:

  $h_t = \text{floor}(N / 2)$, $h_k = \text{floor}(h_{k+1} / 2)$

  Not a good selection though…

--Example for an array **a** of size N = 100

$h_6 = 50$, $h_5 = 25$, $h_4 = 12$, $h_3 = 6$, $h_2 = 3$, $h_1 = \mathbf{1}$.

Shellsort

```
// Shellsort, using Shell's (poor) increments.
template <typename Comparable>
void ShellSort(vector<Comparable> &a) {
  // gap is Shell's increment.
  for (int gap = a.size() / 2; gap > 0; gap /= 2) {
     for (int i = gap; i < a.size(); ++i) {
         // Insertion sort in subarrays.
         Comparable tmp = std::move(a[i]);
         int j = i;
         for (; j >= gap && tmp < a[j - gap]; j -= gap)
            a[j] = std::move(a[j - gap]);
         a[j] = std::move(tmp);
     }  // End of second for.
  }  // End of first for.
}
```

Worst Case Analysis

- Worst case running time using Shell's increments is $\Theta(N^2)$
- **Proof:**

(A) Provide an example input with behavior $\Omega(N^2)$:

Consider N is power of 2

Put the N/2 largest numbers at even positions

N/2 smallest numbers at odd positions

<See image of next slide>

(B) Prove that the algorithm is $O(N^2)$

Worst case analysis

Positions	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Start	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
After 8-sort	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
After 4-sort	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
After 2-sort	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
After 1-sort	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Input that provides worst behavior

Worst Case Analysis

- Worst case running time using Shell's increments is $\Theta(N^2)$
- **Proof of (B):** The algorithm is $O(N^2)$:
  - At pass $\mathbf{h_k}$: h_k insertion sorts => (since ins. sort is $O(n^2)$)

    $h_k\, O((N/h_k)^2) = O(h_k\, (N/h_k)^2) = O(\, N^2 / h_k)$ per pass.
  - Total cost for all passes:

Worst Case Analysis

- Worst case running time using Shell's increments is $\Theta(N^2)$
- **Proof of (B):** The algorithm is $O(N^2)$:
  - At pass $\mathbf{h_k}$: h_k insertion sorts => (since ins. sort is $O(n^2)$)

    $h_k\, O((N/h_k)^2) = O(h_k\, (N/h_k)^2) = O(\, N^2 / h_k)$ per pass.
  - Total cost for all passes:

$$O(\, N^2 / h_t) + O(\, N^2 / h_{t-1}) + \ldots + O(\, N^2 / h_k) + \ldots + O(\, N^2 / h_1) =$$

$$O(\, N^2 / h_t + N^2 / h_{t-1} + \ldots + N^2 / h_k + \ldots + N^2 / h_1\,) =$$

$$O(\, N^2\, (\, 1 / h_t + 1 / h_{t-1} + \ldots + 1 / h_k + \ldots + 1 / h_1\,) =$$

$$O(\, N^2\,).\ \ //\ \ 1 + \tfrac{1}{2} + \tfrac{1}{4} + 1/8 + \ldots = ?$$

Worst Case Analysis

- Worst case running time using Shell's increments is $\Theta(N^2)$
- **Proof of (B):** Example for N = 64 (power of 2)

$h_6 = 32, h_5 = 16, h_4 = 8, h_3 = 4, h_2 = 2, h_1 = 1.$

Total cost for all passes:

$O(N^2 / 32) + O(N^2 / 16) + O(N^2 / 8) + O(N^2 / 4) + O(N^2 / 2) + O(N^2 / 1) =$

$O(N^2 (1 / 32 + 1 / 16 + 1 / 8 + 1 / 4 + 1 / 2 + 1)) = ?$

In general

$O(N^2 (1 / h_t + 1 / h_{t-1} + \ldots + 1 / h_k + \ldots + 1 / h_1) = ?$

Better increments

Improvements over original insertion sort

- 1, 3, 7, …, 2^k-1 (Hibbard)
  - Consecutive increments have no common factors
  - **Theorem:** Worst case running time is $\Theta(N^{3/2})$
  - Proving average-case is still open.
- 1, 5, 19, 41, … (Sedgewick)
  - Worst case running time is $O(N^{4/3})$

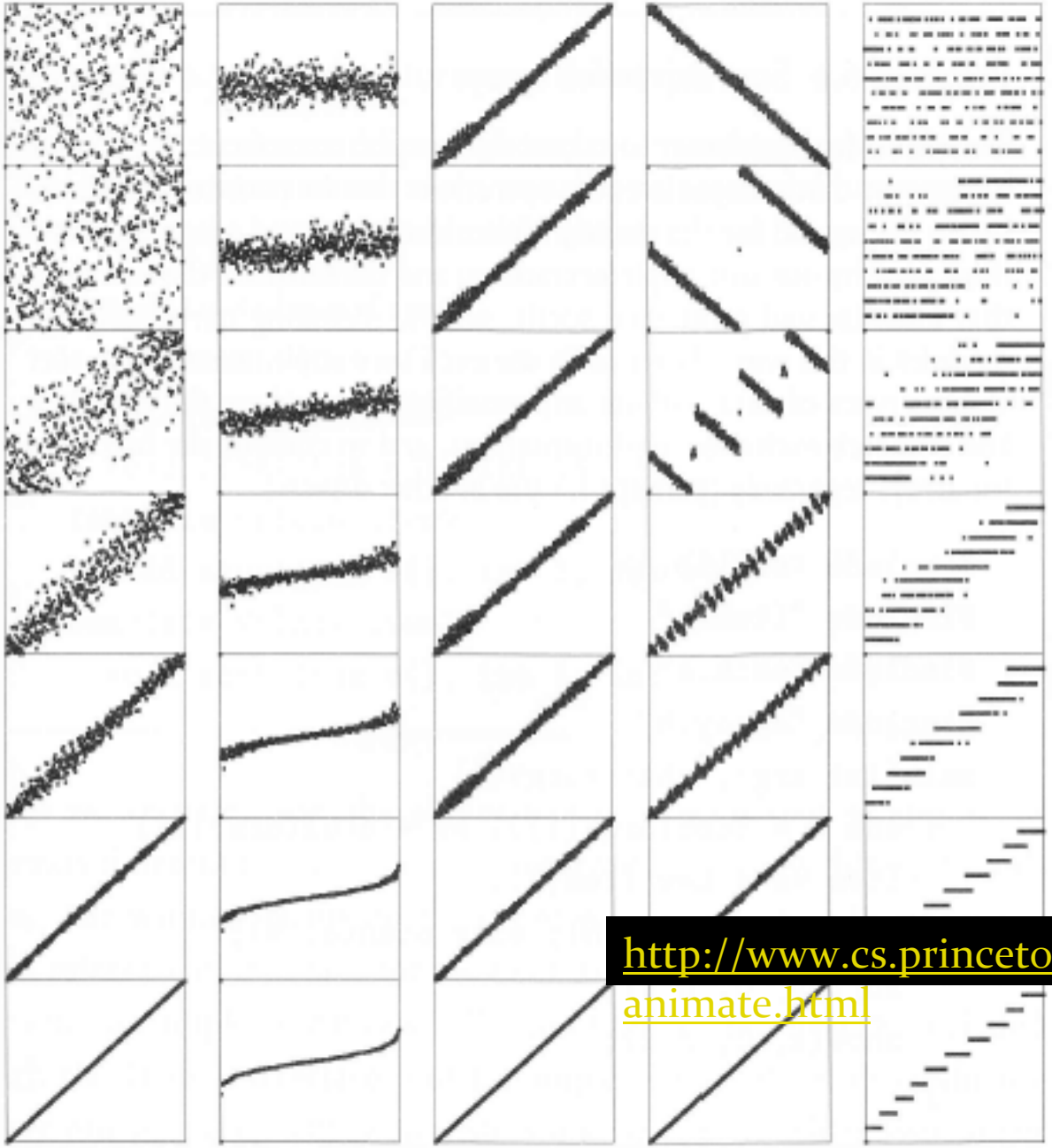


Figure 6.13
Dynamic characteristics of shellsort for various types of files

These diagrams show shellsort, with the increments 209 109 41 19 5 1, *in operation on files that are random, Gaussian, nearly ordered, nearly reverse-ordered, and randomly ordered with 10 distinct key values* (left to right, on the top). *The running time for each pass depends on how well ordered the file is when the pass begins. After a few passes, these files are similarly ordered; thus, the running time is not particularly sensitive to the input.*

http://www.cs.princeton.edu/~rs/shell/animate.html

From, Algorithms in C++, Robert Sedgewick

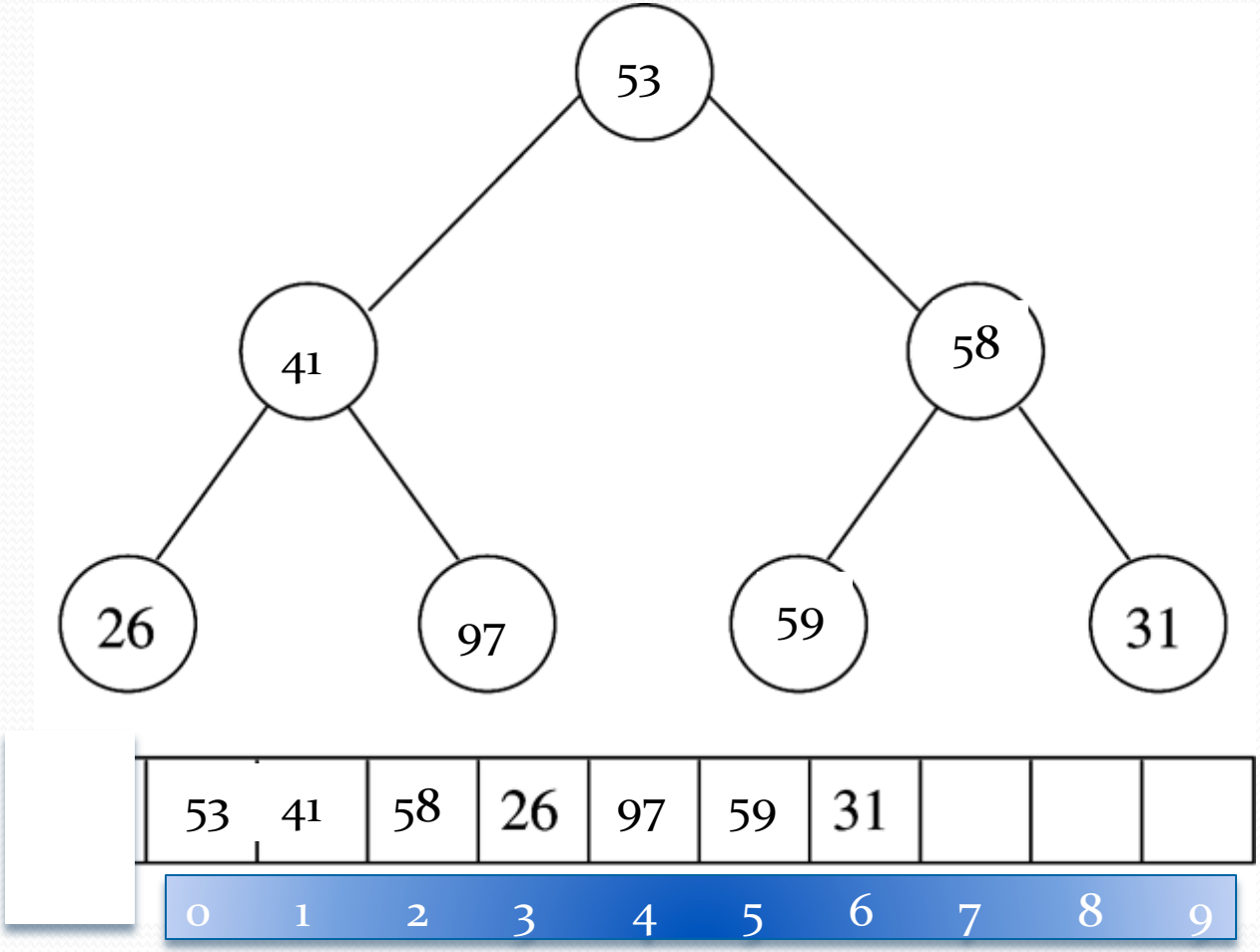
Heapsort

- Uses binary heap implemented as array
- O(N logN) time bound [worst and average]
- In-place algorithm, uses O(1) extra memory.
- Simple: Start with a given array:

    A) Build binary heap from N element: O(N)

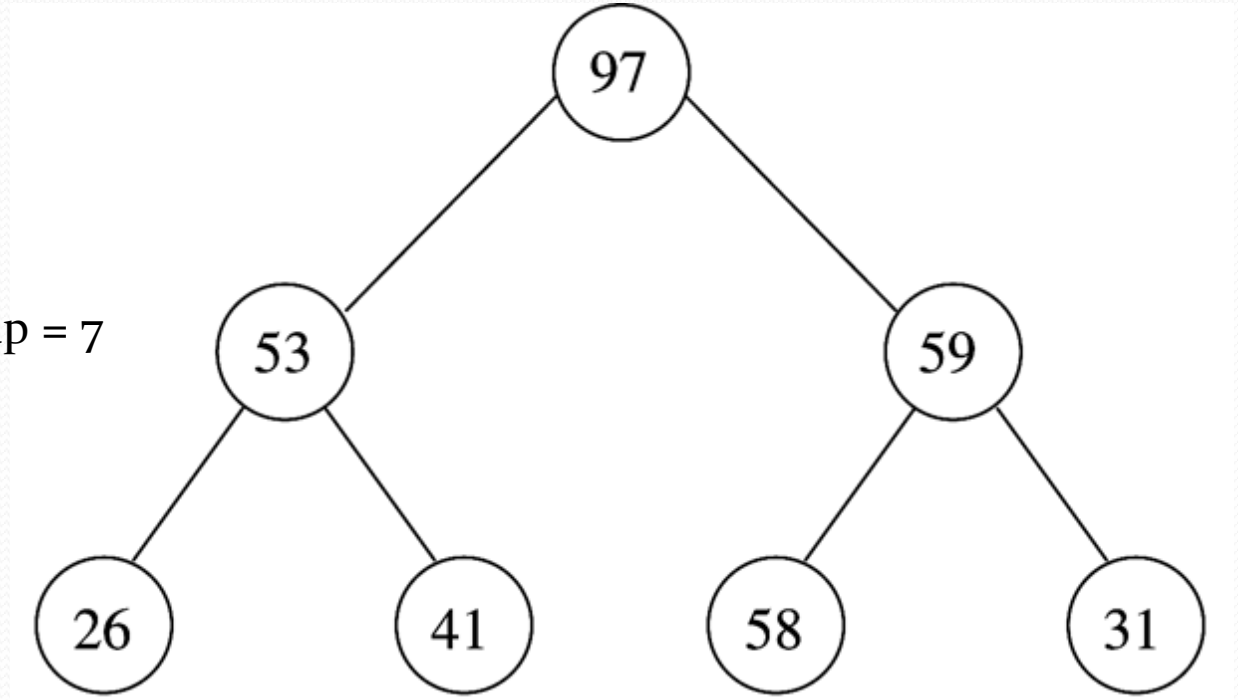
    B) Perform deleteMax() N times => O(NlogN)

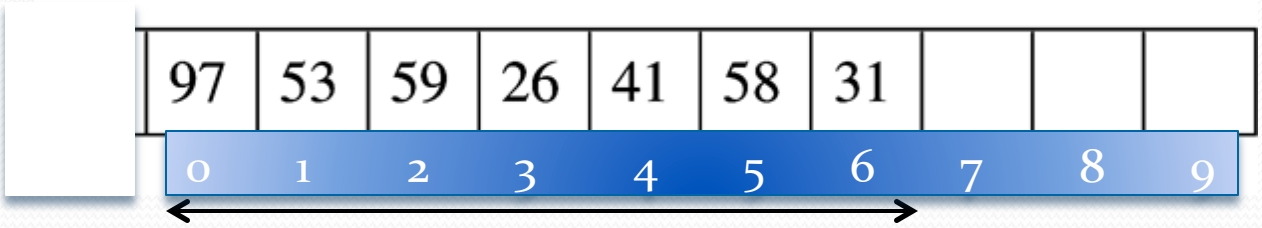
Original array



After build heap

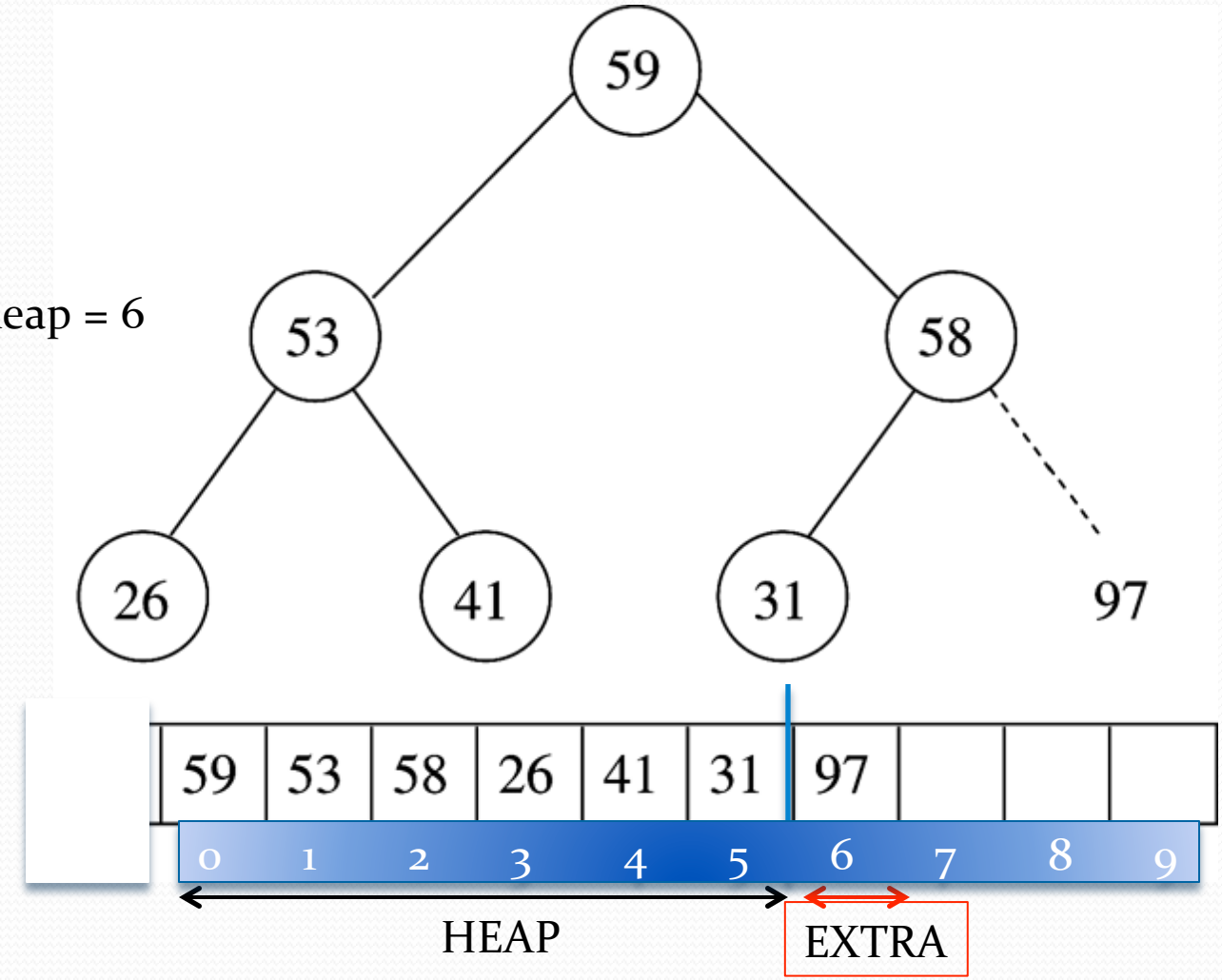
size_of_heap = 7





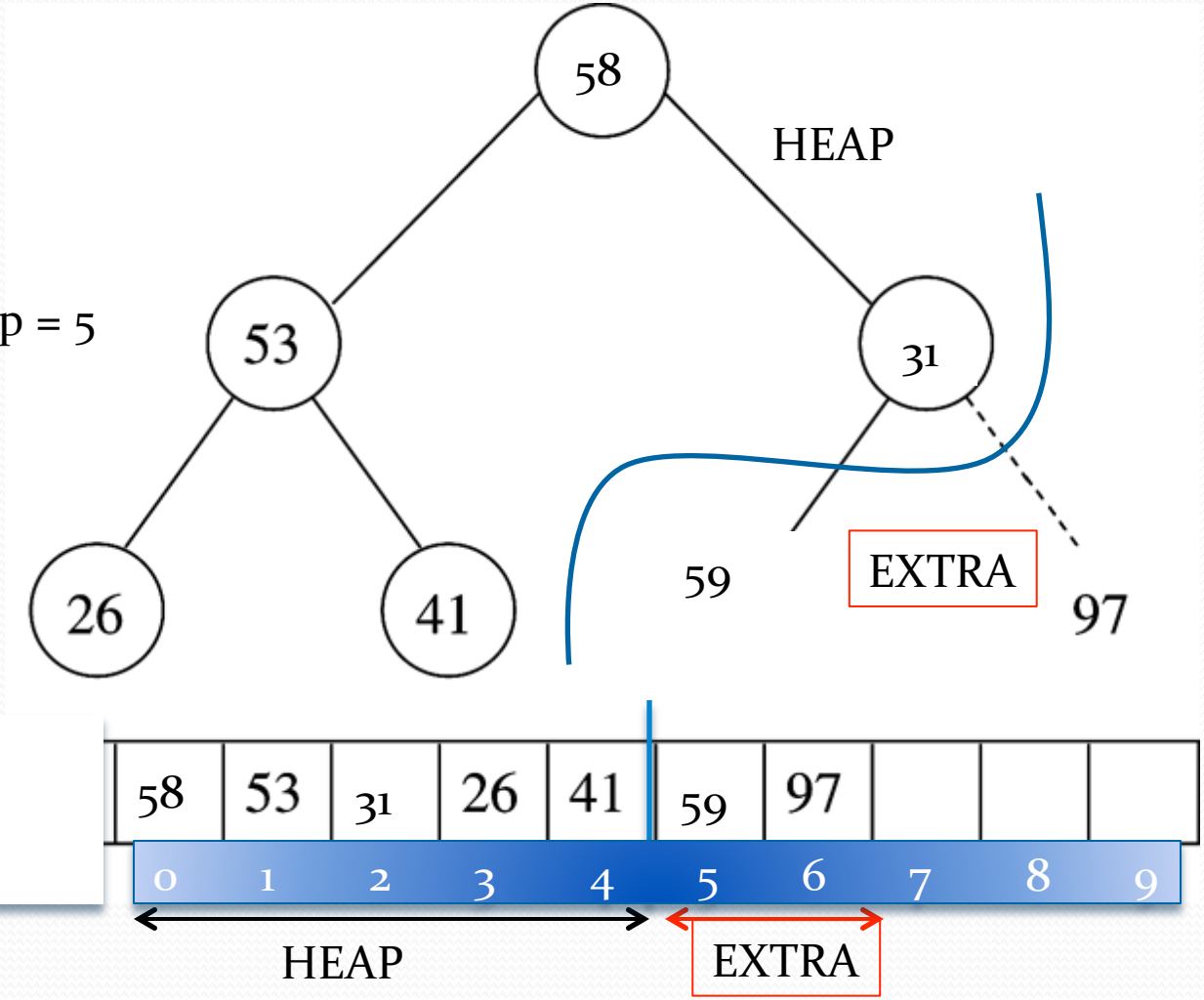
After first deleteMax()

size_of_heap = 6



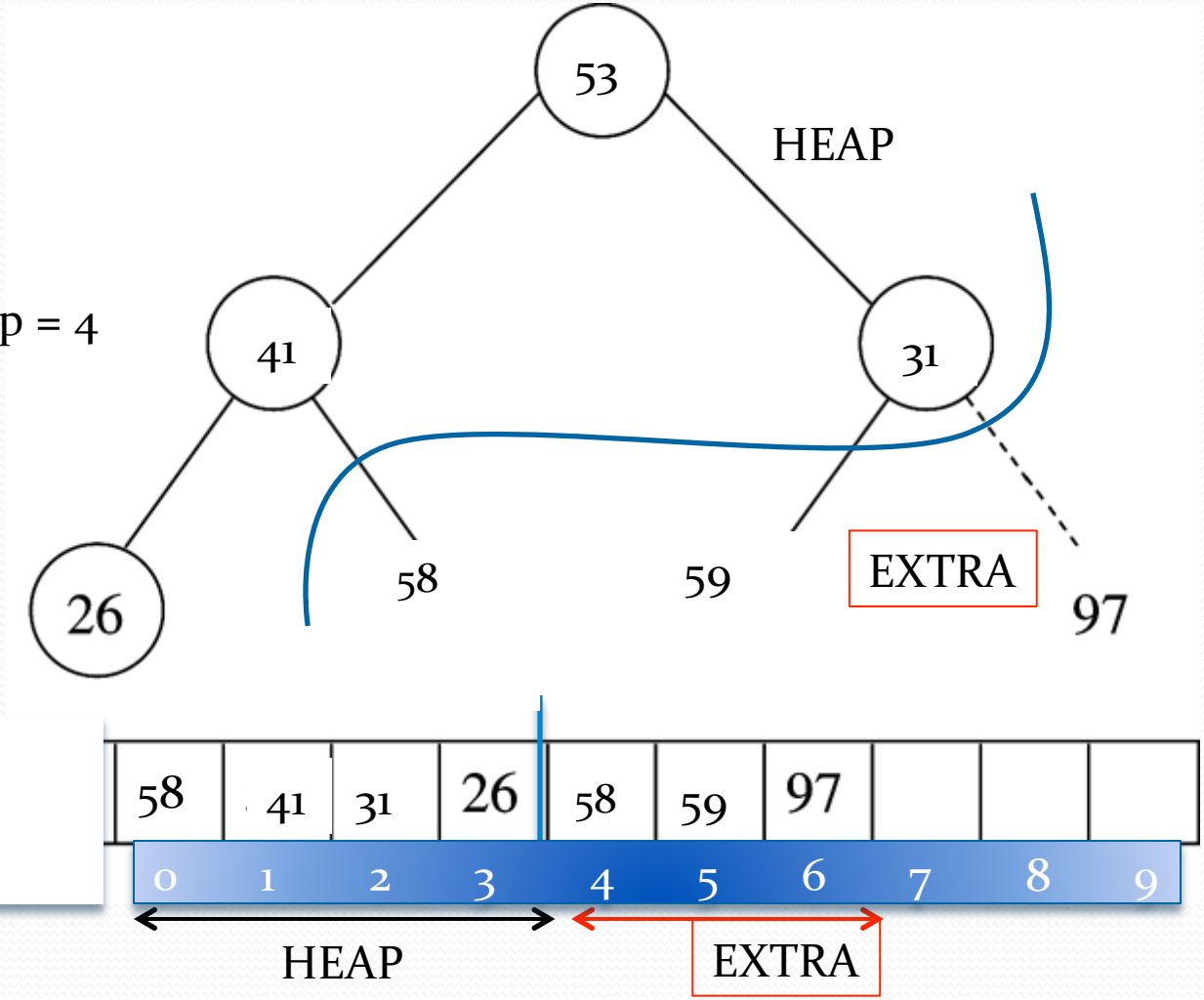
After second deleteMax()

size_of_heap = 5

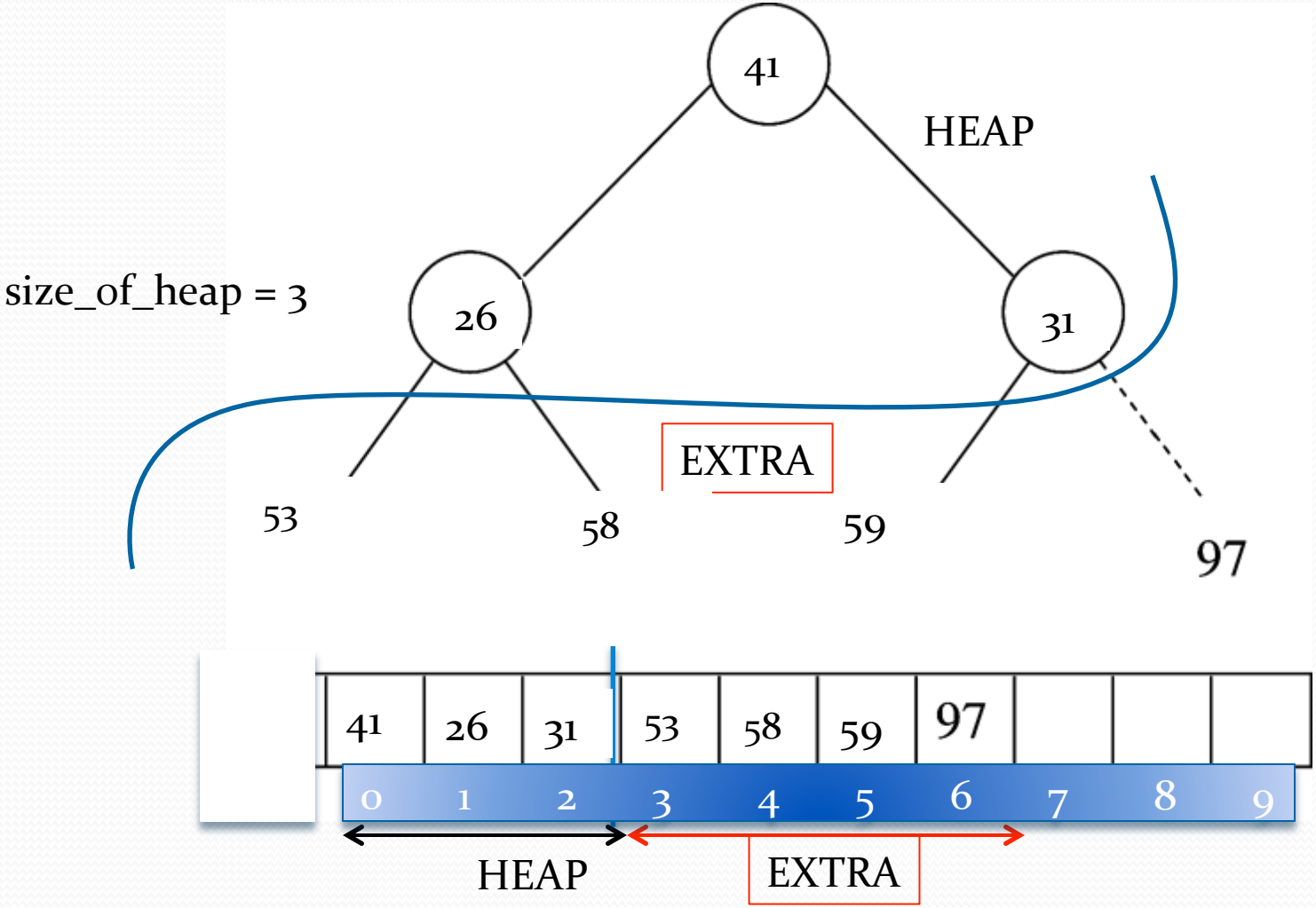


After third deleteMax()

size_of_heap = 4

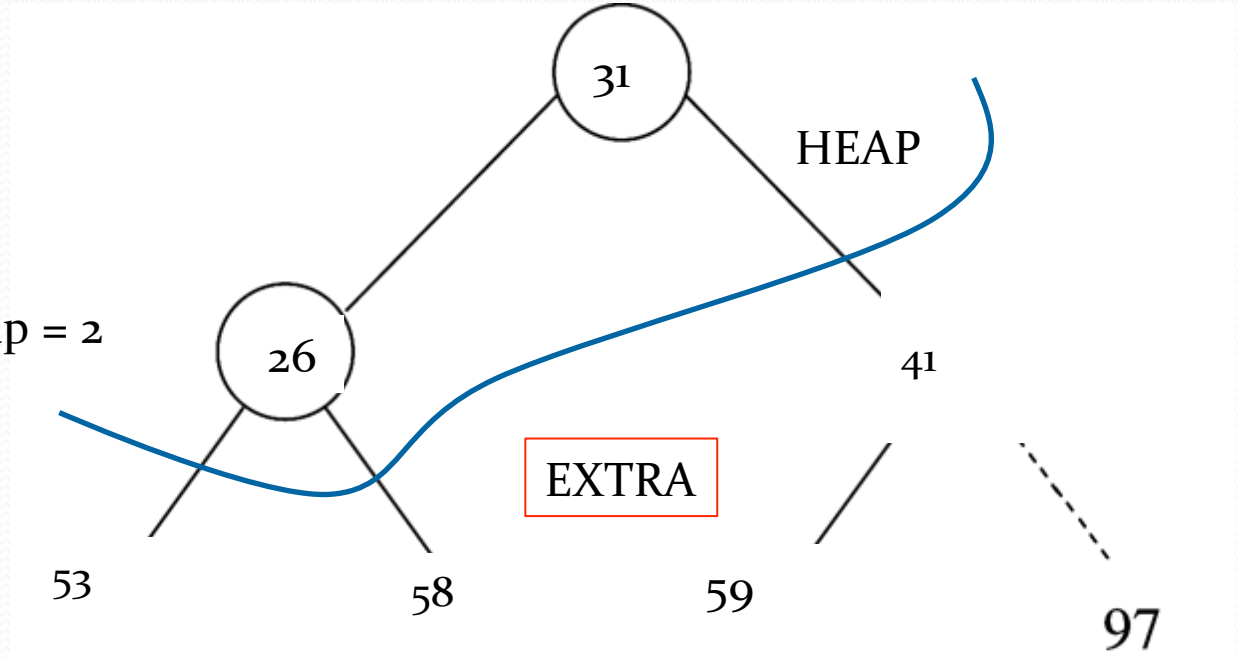


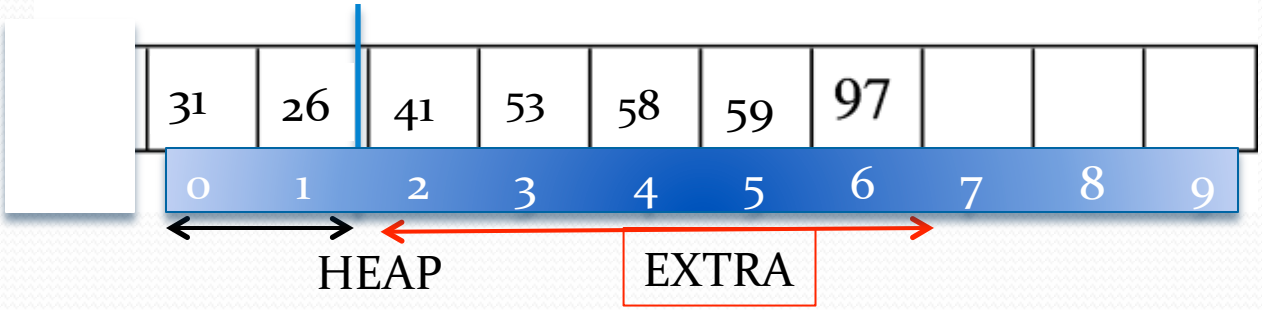
After next deleteMax()



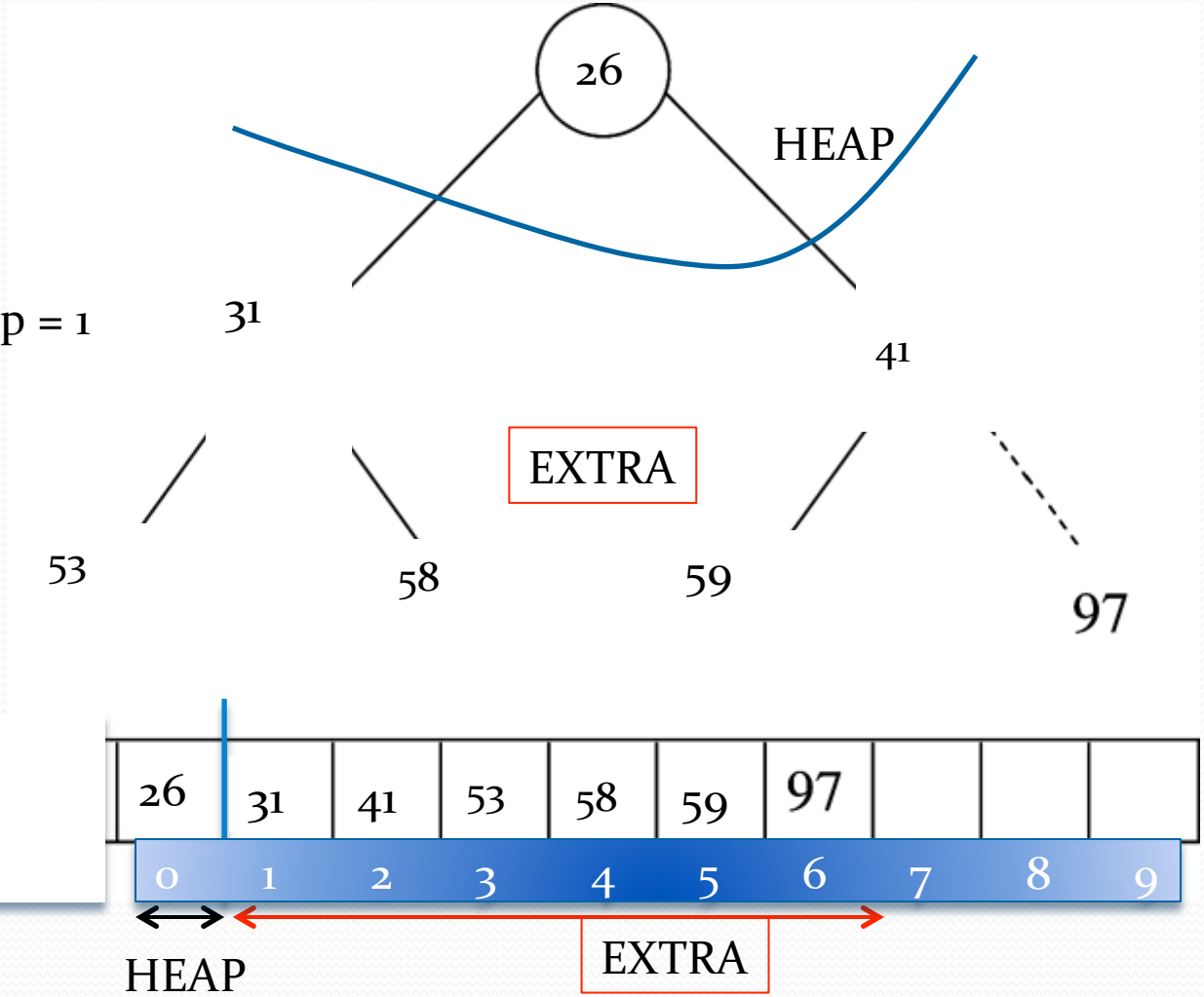
After next deleteMax()

size_of_heap = 2



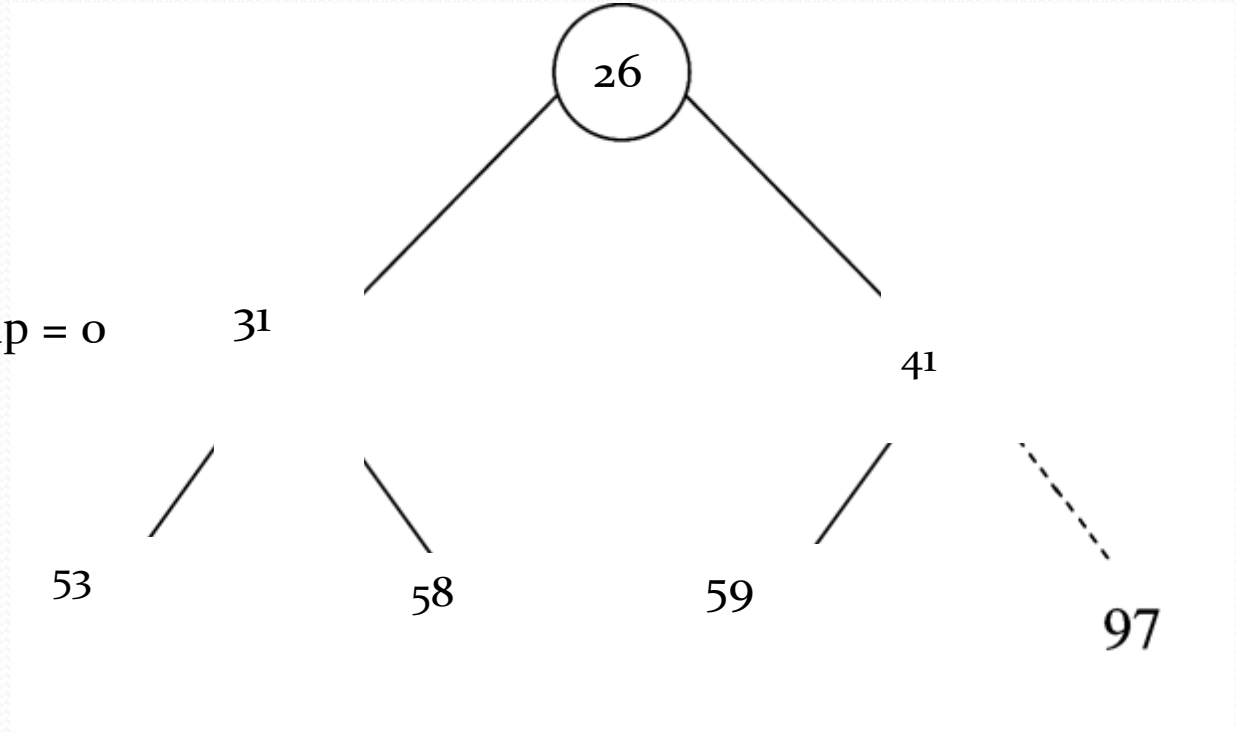


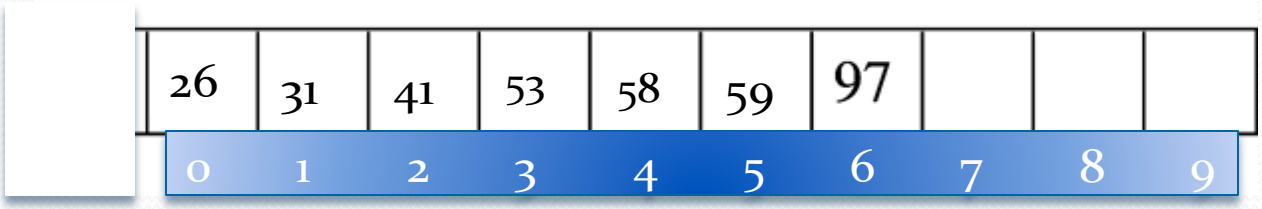
After next deleteMax()



Done!

size_of_heap = 6





Heapsort example

Figure 9.10
Heapsort example

From, Algorithms in C++, Robert Sedgewick

Visualization:
https://www.cs.usfca.edu/~galles/visualization/HeapSort.html

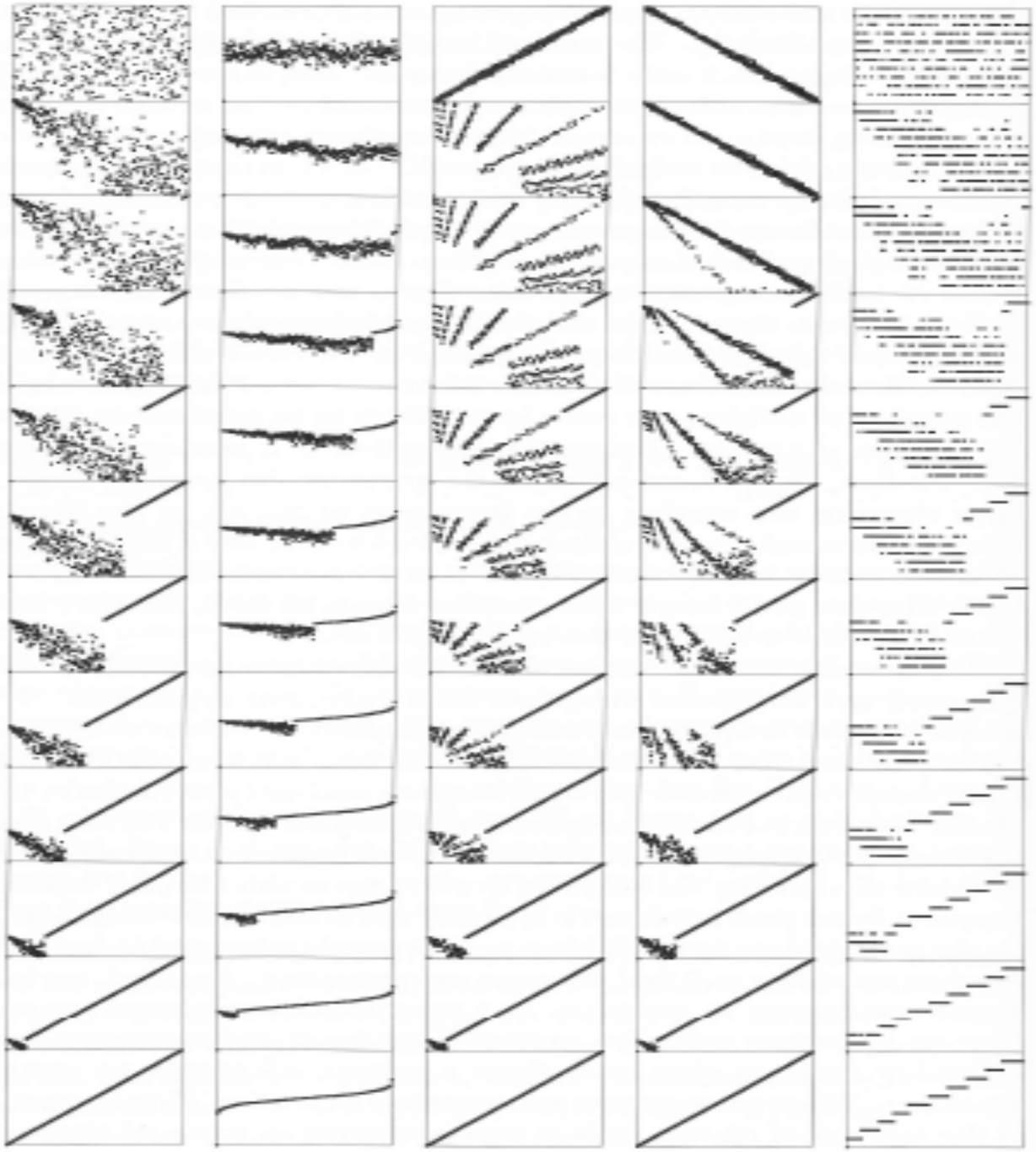
Heapsort

```
// Assume that input is in a[0], a[2], … a[a.size() – 1]
template <typename Comparable>
void heapsort(vector<Comparable> &a) {
  for (int i = a.size() / 2 - 1; i >= 0; --i)  // BuildHeap.
    PercolateDown(a, i, a.size() - 1);

  for (int j = a.size() - 1; j > 0; --j) {
    std::swap(a[0], a[j]);
    PercolateDown(a, 0, j - 1);
  }
}
// Percolates down a[pos] in heap of size size_of_heap
void PercolateDown(const vector<Comparable> &a, int pos, int
size_of_heap) { … }
```

```
inline int LeftChild(int i) {
    return 2 * i + 1;
}

// Heap starts from 0.
// @pos: Start percolating down from that index.
// @size_of_heap: number of elements in the heap.
template <typename Comparable>
void PercolateDown(const vector<Comparable> &a, int pos, int size_of_heap) {
    int child;
    Comparable tmp;

    for (tmp = std::move(a[pos]); LeftChild(pos) < size_of_heap; pos = child) {
        child = LeftChild(pos);
        if (child != size_of_heap - 1 && a[child] < a[child + 1])
            ++child;
        if (tmp < a[child])
            a[pos] = std::move(a[child]);
        else
            break;
    }
    a[pos] = std::move(tmp);
}
```

Figure 9.12
Dynamic characteristics of heapsort on various types of files

The running time for heapsort is not particularly sensitive to the input. No matter what the input values are, the largest element is always found in less than lg N steps. These diagrams show files that are random, Gaussian, nearly ordered, nearly reverse-ordered, and randomly ordered with 10 distinct key values (at the top, left to right). The second diagrams from the top show the heap constructed by the bottom-up algorithm, and the remaining diagrams show the sortdown process for each file. The heaps somewhat mirror the initial file at the beginning, but all become more like the heaps for a random file as the process continues.

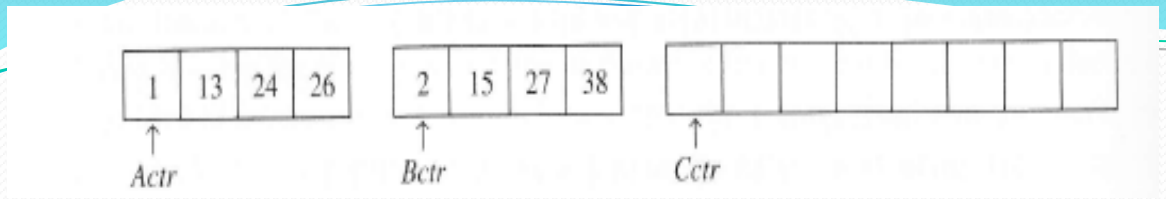


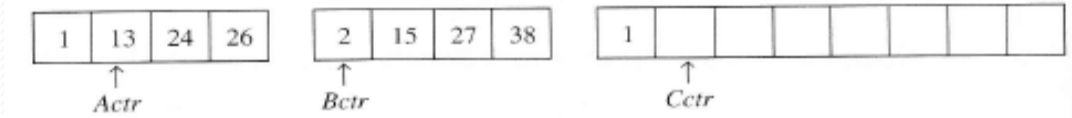
From, Algorithms in C++, Robert Sedgewick

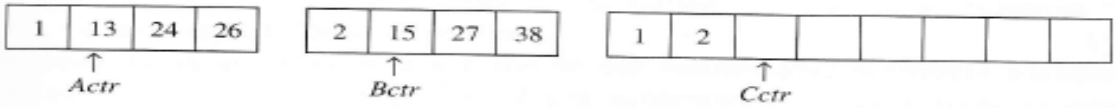
Mergesort

- O(N logN) worst-case running time
- Near optimal comparisons
- Recursive
- Key: O(N) for merging two sorted lists/arrays
- Need for extra O(N) space to merge arrays
- Good for linked lists, no need for random access
  - No need for extra space to merge sublists
- Stable sort
- May be better in C++11 with move semantics

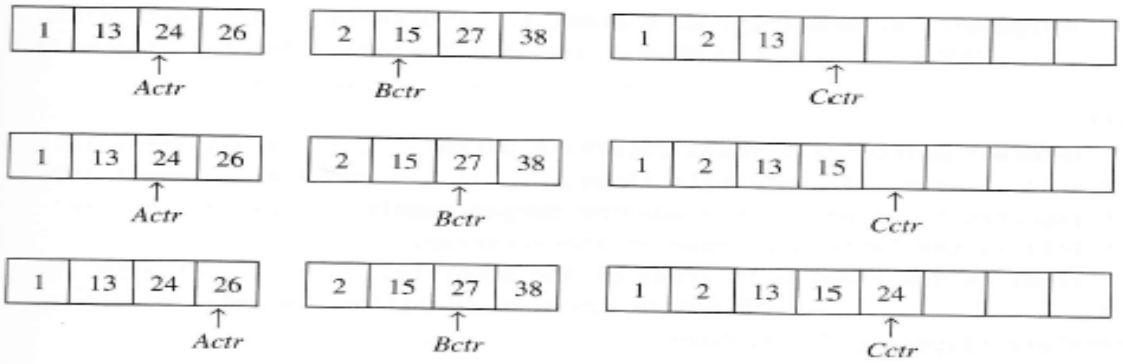
Merge two sorted arrays (or lists)



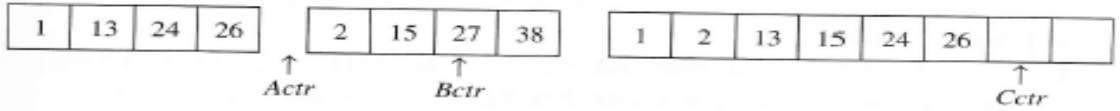




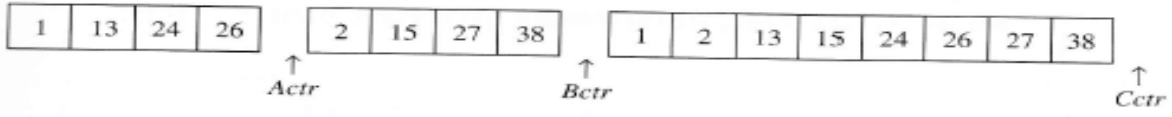
3 is added to C, and then 24 and 15 are compared. This proceeds until 26 and 27 ar
mpared.



is added to C, and the A array is exhausted.



e remainder of the B array is then copied to C.



MergeSort

```
// Mergesort algorithm (driver).
// @param a: input vector to sort.
template <typename Comparable>
void MergeSort(vector<Comparable> & a) {
    vector<Comparable> tmp_array(a.size());
    MergeSortInternal(a, tmp_array, 0, a.size() - 1);
}
// @param a: input vector to sort.
// @param tmp_array: temporary vector to store intermediate results.
// @left: left index in a.
// @right: right index in a (right > = left).
// Part of vector: a[left],…,a[right] will be sorted at the end of
// function.
template <typename Comparable>
void MergeSortInternal(vector<Comparable> & a,
                       vector<Comparable> & tmp_array, int left, int right) {
    if (left < right) {
       const int center = (left + right) / 2;
       MergeSortInternal(a, tmp_array, left, center);
       MergeSortInternal(a, tmp_array, center + 1, right);
       Merge(a, tmp_array, left, center + 1, right);
    }
}
```

```
// @param a: input vector for merging.
// @param tmp_array: temporary vector to store intermediate results.
// @left_pos: index in a.
// @right_pos: index in a.
// @right_end: index in a.
// Assumes that a[left_pos], … , a[right_pos – 1] is sorted.
// Assumes that a[right_pos], … , a[right_end] is sorted.
// Part of vector: a[left_pos], … ,a[right_end] will be the merged result
//   of the above two sorted subarrays.
template <typename Comparable>
Void Merge(vector<Comparable> &a,
           vector<Comparable> &tmp_array,
           int left_pos, int right_pos, int right_end) {
    const int left_end = righ_pos - 1;
    int tmp_pos = left_pos;  // Index in temporary array.
    while (left_pos <= left_end && right_pos <= right_end) {
        if (a[left_pos] <= a[right_pos])
            tmp_array[tmp_pos++] = std::move(a[left_pos++]);
        else
            tmp_array[tmp_pos++] = std::move(a[right_pos++]);
    }
    while (left_pos <= left_end)  // Copy rest of first half.
        tmp_array[tmp_pos++] = std::move(a[left_pos++]);
    while (right_pos <= right_end)  // Copy rest of right half.
        tmp_array[tmp_pos++] = std::move(a[right_pos++]);
    // Move from temp_array to a.
    const int num_elements = right_end - left_pos + 1;
    for (int i = 0; i < num_elements; ++i, --right_end)
        a[right_end] = std::move(tmp_array[right_end]);
}
```

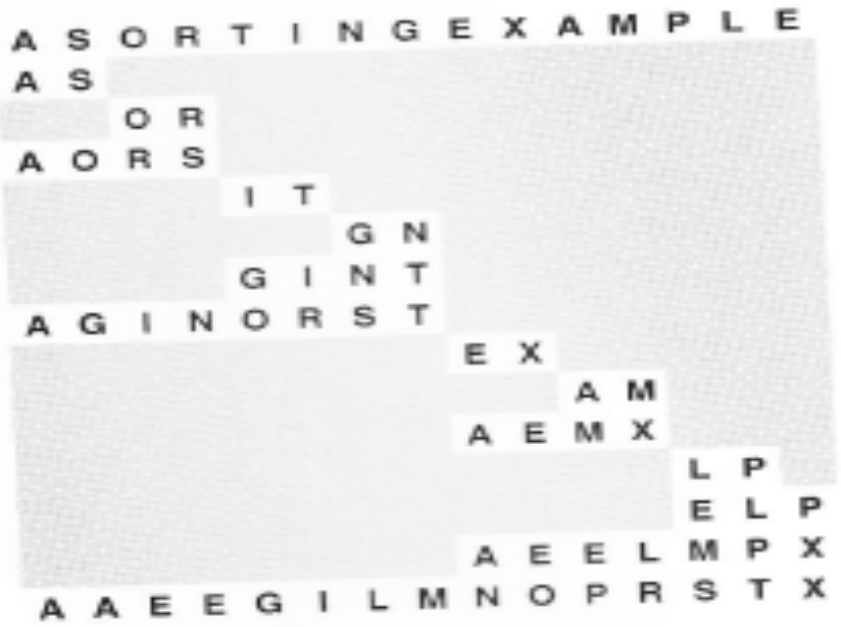


Figure 8.2
Top-down mergesort example

Each line shows the result of a call on `merge` *during top-down mergesort. First, we merge A and S to get A S; then, we merge O and R to get O R; then, we merge O R with A S to get A O R S. Later, we merge I T with G N to get G I N T, then merge this result with A O R S to get A G I N O R S T, and so on. The method recursively builds up small sorted files into larger ones.*

From, Algorithms in C++,
Robert Sedgewick

Mergesort Analysis

- $T(0) = T(1) = 1$
- $T(n) = 2T\left(\frac{n}{2}\right) + n$

We assume that n is a power of 2, i.e. n = 2^k for some positive k

- We will solve this recurrence relation by creating a telescoping sum. (The other way we've been doing it is also in the book)
- $\frac{T(n)}{n} = \frac{2T\left(\frac{n}{2}\right)}{n} + \frac{n}{n} = \frac{T(n/2)}{n/2} + 1$

Mergesort Analysis

Sort array of size n

$$\frac{T(n)}{n} = \frac{T(n/2)}{n/2} + 1$$

Sort array of size n/2

$$\frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + 1$$

Sort array of size n/4

$$\frac{T(n/4)}{n/4} = \frac{T(n/8)}{n/8} + 1$$

.
.
.

Sort array of size 2

$$\frac{T(2)}{2} = \frac{T(1)}{1} + 1$$

Mergesort Analysis

Sum of left sides:

$T(n)/n + T(n/2)/(n/2) + T(n/4)/(n/4) + \ldots + T(4)/4 + T(2)/2$

Equals

Sum of right sides:

$T(n/2)/(n/2) + 1 + T(n/4)/(n/4) + 1 + T(n/8)/(n/8) + 1 + \ldots + T(2)/2 + 1 + T(1)/1 + 1$

Mergesort Analysis

Sum of left sides:

$T(n)/n + T(n/2)/(n/2) + T(n/4)/(n/4) + \ldots + T(4)/4 + T(2)/2$

Equals

Sum of right sides:

$T(n/2)/(n/2) + 1 + T(n/4)/(n/4) + 1 + T(n/8)/(n/8) + 1 + \ldots + T(2)/2 + 1 + T(1)/1 + 1$

Equal parts cancel out

Mergesort Analysis

Sum of left sides:

T(n)/n

Equals

Sum of right sides:

1 + 1 + 1 + … + 1 + T(1)/1 + 1

How many 1's ?

Sum of left sides:

T(n)/n

Equals

Sum of right sides:

1 + 1 + 1 + … + 1 + T(1)/1 + 1

How many 1's ? Answer: log(n)

[Hint: If n is a power of 2 how many times can we divide n until we reach 1?]

Mergesort Analysis

$T(n)/n = \log(n) + T(1)/1 \Rightarrow$

$T(n) = n \log(n) + n\, T(1) \Rightarrow$

$T(n) = n \log(n) + n \text{ [since } T(1) = 1] \Rightarrow$

$$\boldsymbol{T(n) = O(n \log(n))}$$

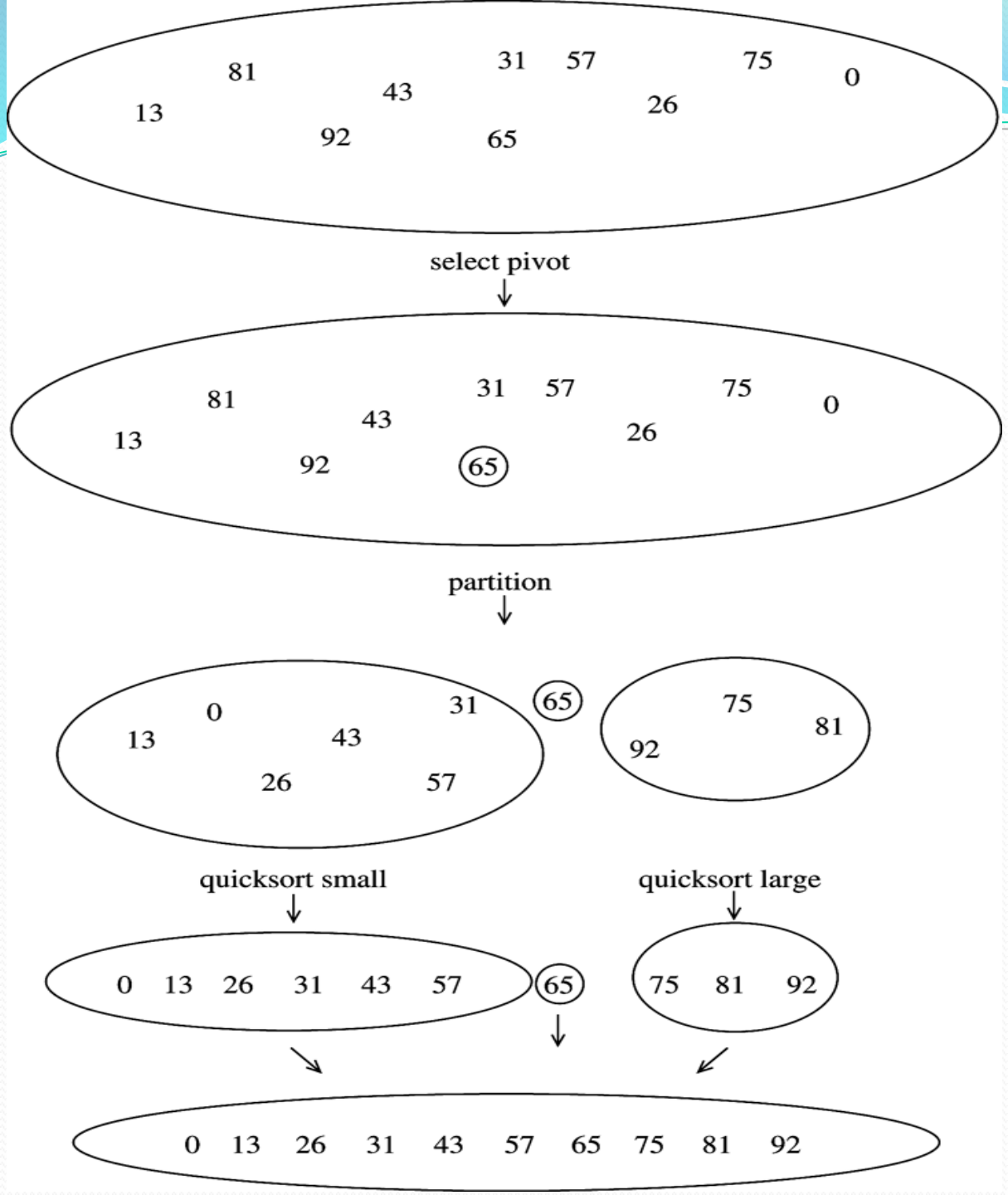
Quicksort

- Fastest for C++, more comparisons but fewer swaps
- O(NlogN) average, $O(N^2)$ worst
- To sort array S:

  quicksort(S)

  - If |S|=0, or 1 then return
  - **Pick** any element v in S. This is the **pivot**.
  - Partition S - {v} into two disjoint groups:

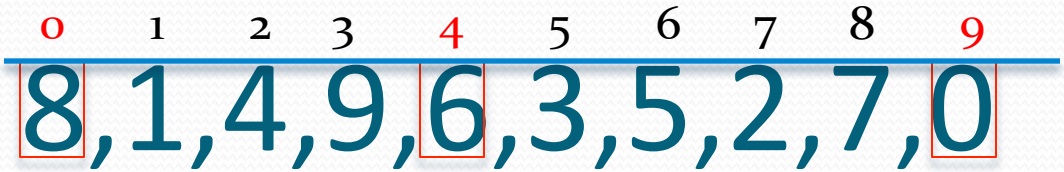
    $S_1=\{x \text{ in } S - \{v\} \mid x \leq v\}$ and $S_2=\{x \text{ in } S - \{v\} \mid x \geq v\}$
  - **Return** quicksort(S_1), v , quicksort(S_2)


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

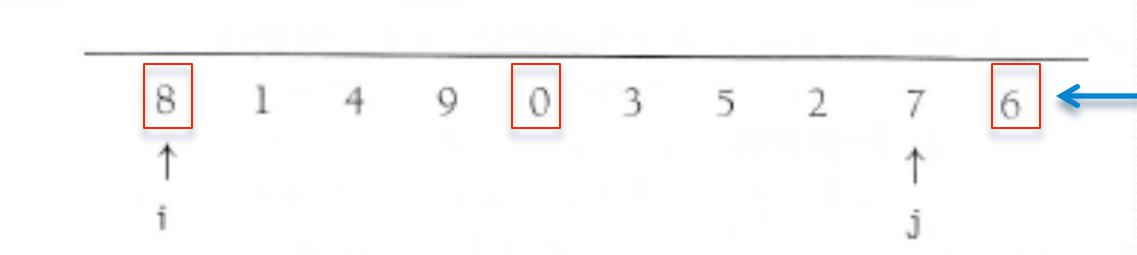
```
// A generic quick-sort type routine. Not the most efficient one.
// Not an in-place implementation.
// Pivot is always the middle item.
template <typename Comparable>
void SortGeneric(vector<Comparable> &items) {
    if (items.size() <= 1) return;

    vector<Comparable> smaller;
    vector<Comparable> same;
    vector<Comparable> larger;

    auto chosen_item = items[items.size() / 2];

    for (auto &x : items) {
       if (x < chosen_item)
           smaller.push_back(std::move(x));
       else if (chosen_item < x)
           larger.push_back(std::move(x));
       else
           same.push_back(std::move(x));
    }

    SortGeneric(smaller); // Recursive call.
    SortGeneric(larger);  // Recursive call.

    // Move from temporary storage to items.
    std::move(begin(smaller), end(smaller), begin(items));
    std::move(begin(same), end(same), begin(items) + smaller.size());
    std::move(begin(larger), end(larger), end(items) - larger.size());
}
```

Picking the pivot

- Choose first element as the pivot?
- Randomly select a pivot?
- Median-of-three pivot selection.
  - Pivot $=\text{median}(S[0], S[n-1/2\ ], S[n-1])$
  - E.g.,: 8,1,4,9,6,3,5,2,7,0
  - Pivot $=\text{median}(8, 6, 0)=6$
  - Good for sorted input.

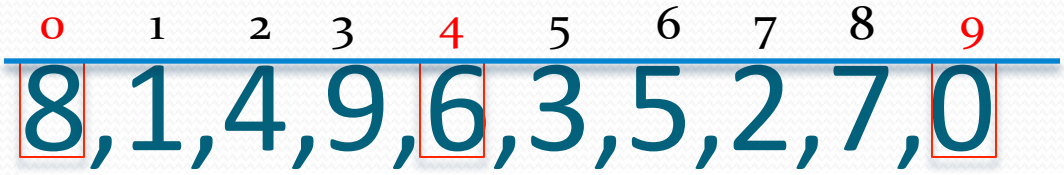


First index: **0**, Last index: **9**,
Middle index: (0+9)/2 = **4**

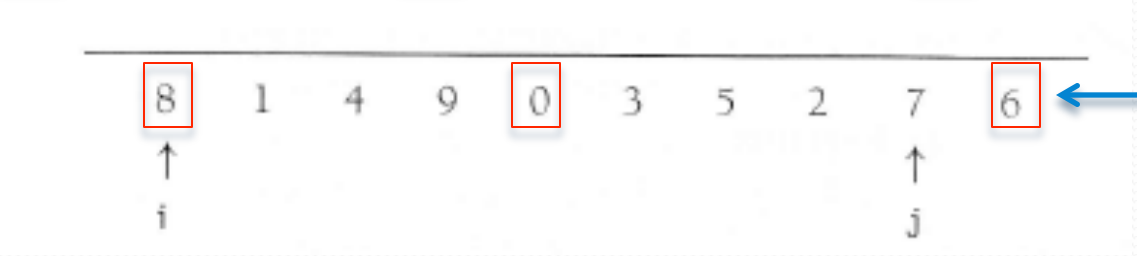


Take care of median

Compute median (6)



First index: **0**, Last index: **9**,
Middle index: (0+9)/2 = **4**



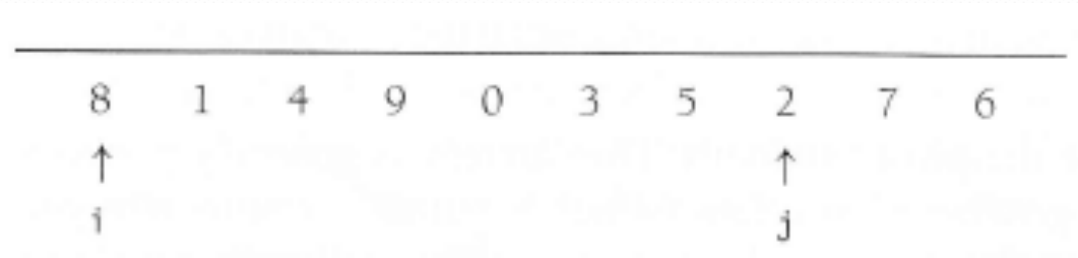
Take care of median

While i < j

move i to the right skipping over elements smaller than pivot

move j to the left skipping over elements larger than pivot

If i<j **swap** a[i] with a[j]



After First Swap

2	1	4	9	0	3	5	8	7	6
↑							↑		
i							j		

Before Second Swap

2	1	4	9	0	3	5	8	7	6
			↑			↑			
			i			j			

After Second Swap

2	1	4	5	0	3	9	8	7	6
			↑			↑			
			i			j			

Before Third Swap

2	1	4	5	0	3	9	8	7	6
					↑	↑			
					j	i			

At this stage, i and j have crossed, so no swap is performed. The final part of the partitioning is to swap the pivot element with the element pointed to by i.

After Swap with Pivot

2	1	4	5	0	3	6	8	7	9
						↑			↑
						i			pivot

While i < j

move i to the right skipping over elements smaller than pivot

move j to the left skipping over elements larger than pivot

If i<j **swap** a[i] with a[j]

Finally, swap a[i] with pivot

Sort left part:

0 1 2 3 4 5

2, 1, 4, 5, 0, 3

Take care of median:

2, 1, 4, 5, 0, 3

i j

Sort left part:

0 1 2 3 4 5

2, 1, 4, 5, 0, 3

2, 1, 4, 5, 0, 3

i j (move i to right until a[i] is larger than pivot)

Sort left part:

0 1 2 3 4 5

2, 1, 4, 5, 0, 3

Take care of median:

2, 1, 4, 5, 0, 3

i j (move j to left until a[j] is smaller than pivot)

Sort left part:

0 1 2 3 4 5

2, 1, 4, 5, 0, 3

2, 1, 0, 5, 4, 3

i j (swap a[i] with a[j])

Sort left part:

0 1 2 3 4 5

2, 1, 4, 5, 0, 3

2, 1, 0, 5, 4, 3

i j (move i to right until a[i] is larger than pivot)

Sort left part:

0 1 2 3 4 5

2, 1, 4, 5, 0, 3

2, 1, 0, 5, 4, 3

j i (move j to left until a[j] is smaller than pivot)

Sort left part:

0 1 2 3 4 5

2, 1, 4, 5, 0, 3

2, 1, 0, 5, 4, 3

j i (j crosses i (i < j))

Sort left part:

0 1 2 3 4 5

2, 1, 4, 5, 0, 3

2, 1, 0, 3, 4, 5

j i (swap a[i] with pivot)

Continue with 2, 1, 0 & 4, 5….

Important detail

- How to handle items that are EQUAL to pivot ?
  - Should i stop at element that = the pivot?
  - Should j stop at element that = the pivot?

  - 8 1 6 9 6 3 5 2 7 | 6
    i j

  - 4 4 4 4 4 4 4 4 | 4
    i j

Important detail

- **If both stop:** many swaps, but i, j will cross in middle
  - Good: even partitioning
- If none stop:
  - Prevent them from crossing boundaries
  - i will touch the end, and j will touch the beginning
  - Pivot will swap a[i]
  - Bad: uneven partitioning
- This detail not as silly as it seems.

Important detail

- **If both stop:** many swaps, but i,j will cross in middle
  - Good even partitioning is better for divide-and-conquer
  - $T(n) \approx 2T(n/2) + n = O(n \log n)$
- If none stop:
  - i will touch the end or j will touch the beginning
  - Pivot will swap a[i]
  - Bad uneven partitioning leads to worst-case behavior
  - $T(n) \approx T(n-1) + n = O(n^2)$

Small arrays

- For N <=20, we can use insertion sort => reduce number of recursive calls.

```
1   /**
2    * Quicksort algorithm (driver).
3    */
4   template <typename Comparable>
5   void quicksort( vector<Comparable> & a )
6   {
7       quicksort( a, 0, a.size( ) - 1 );
8   }
```

```
// Return median of left, center, and right.
// @param a: input vector.
// @param left: left index in a.
// @param right: right index in a.
// pivot is the median of a[left], a[(left + right) / 2] and a[right]
// The above three elements are being sorted in place.
// pivot is placed at a[right – 1] and is being returned.
template <typename Comparable>
const Comparable &
Median3(vector<Comparable> &a, int left, int right) {
    const int center = (left + right) / 2;
    if (a[center] < a[left])
        std::swap(a[left], a[center]);
    if (a[right] < a[left])
        std::swap(a[left], a[right]);
    if (a[right] < a[center])
        std::swap(a[center], a[right] );

    // Place pivot at position right - 1
    std::swap(a[center], a[right – 1]);
    return a[right - 1];
}
```

```
// @param a: input vector.
// @param left: left index in a.
// @param right: right index in a.
// Implements quicksort. At the end a[left], … , a[right] will be sorted.
template <typename Comparable>
void quicksort(vector<Comparable> &a, int left, int right) {
    if (left + 10 > right) {  // Do insertion sort if input <= 10 items.
       InsertionSort(a, left, right);
       return;
    }

    const Comparable &pivot = median3(a, left, right);
    int i = left, j = right - 1;
    while (true) {  // Main partitioning loop.
       while (a[++i] < pivot) {}
       while (pivot < a[--j]) {}
       if (i < j) std::swap(a[i], a[j]);
       else break;
    }
    std::swap(a[i], a[right – 1]);  // Restore pivot
    quicksort(a, left, i - 1);  // Sort elements smaller than pivot.
    quicksort(a, i + 1, right); // Sort elements larger than pivot.
}
```

Analysis of Quicksort

- Running time for array of N elements:

$$T(N) = T(i) + T(N-i-1) + cN, \quad i = |S1|$$

$$T(0) = T(1) = 1$$

Worst-Case

- i = 0 at every step (worst possible partitioning)

```
    T(N) = T(0)+T(N-1)+cN =>
    T(N) = T(N-1) + cN,          N>1
---------------------------------
   T(N-1)=T(N-2)+c(N-1)
   T(N-2)=T(N-3)+c(N-2)
       …
    T(2)    = T(1)    +c(2)
=============================
   T(N)=T(1) + c( N+(N-1)+…+2)

    =>T(N)=Θ(N^2)
```

Best-Case

- i=N/2 always (best possible partitioning)

  T(N)=T(N/2)+T(N/2)+cN=>

  T(N) = 2T(N/2)+cN

  --------same as mergesort-----→

  T(N)=O(NlogN)

Average-Case

$T(N) = T(i) + T(N-i-1) + cN$, i=size of S1, $N>=2$

$T(0)=T(1)=1$

How do we define average case?--→

Sizes of S1 are equally likely, i.e.

S1 can be of size 0, 1, 2, …, or N-1 with equal probability 1/N:

$Prob\{ |S1| = k \} = 1/N$ for k=0, …, N-1

|A| is the size of set A

Average-Case

$T(N) = T(i) + T(N-i-1) + cN$, i=size of S1, N>=2

$T(0)=T(1)=1$

How do we define average case?--→

It follows that sizes of **S2** are equally likely also, i.e.

S2 can be of size 0, 1, 2, …, N-1 with equal prob 1/N:

$Prob\{ |S2| = k \} = 1/N$ for every k=0, …, N-1

Average-Case

- Let us rewrite:

$$Tavg(|S|) = Tavg(|S1|) + Tavg\ (|S2|) + cN,\ N=|S|$$

$$N>=2$$

Average-Case

- **Expected** value of T(i) (|S1|=i) is

  $(1/N)\ T(0)+\ (1/N)\ T(1)+\ldots+\ (1/N)\ T(N-1)=$
  $(1/N)\ (T(0)+\ldots+T(N-1)\ =$
  $(1/N)\sum_{j=0}^{N-1} T(j)$

- The above is the expected value of T(N-i-1) (|S2|=n-i-1) as well

Average-Case

- Recurrence relation becomes:

$$T(N) = (2/N)[\sum_{j=0}^{N-1} T(j)] + cN,\ N>=2$$

or

$$NT(N) = 2\ [\sum_{j=0}^{N-1} T(j)] + cN^2$$

of course this is true for N-1 as well:

$$(N-1)T(N-1) = 2\ [\sum_{j=0}^{N-2} T(j)] + c(N-1)^2$$

Average-Case

- By subtraction:

$$NT(N) - (N-1)T(N-1) = 2\left[\sum_{j=0}^{N-1} T(j)\right] - 2\left[\sum_{j=0}^{N-2} T(j)\right] + cN^2 - c(N-1)^2$$

$$= 2\,T(N-1) + 2cN - c \Rightarrow \text{(-c not significant)}$$

$$NT(N) = (N+1)T(N-1) + 2cN \Rightarrow$$

$$T(N) = ((N+1)/N)\,T(N-1) + 2c \Rightarrow$$

$$\mathbf{T(N)/(N+1) = T(N-1)/N + 2c/(N+1)}$$

Average-case

- $T(N)/(N+1) = T(N-1)/N + 2c/(N+1)$ **[N]**

Telescope:

$T(N-1)/N = T(N-2)/(N-1)+2c/N$ **[N-1]**

$T(N-2)/(N-1)=T(N-3)/(N-2)+2c/(N-1)$ **[N-2]**

....

$T(2)/3 = T(1)/2 +2c/3$ **[2]**

---------- **+all**

$T(N)/(N+1)=T(1)/2 + 2c/(N+1) + 2c/N + \ldots +2c/3$

Average-case

- **$T(N)/(N+1)=T(1)/2 + 2c/(N+1) + 2c/N + \ldots + 2c/3$**

Compute: $2c/(N+1)+\ldots 2c/3=2c*(1/3+1/4+\ldots+1/(N+1))$

$1/3+1/4+\ldots+1/(N+1) = \log_e(N+1)+\gamma-3/2,$

$\gamma \sim 0.577$ (Euler's constant)

- Finally!! $T(N)/(N+1)=O(\log N)$ or

  $T(N)=O(N\log N)$

Linear-Expected-Time for Selection

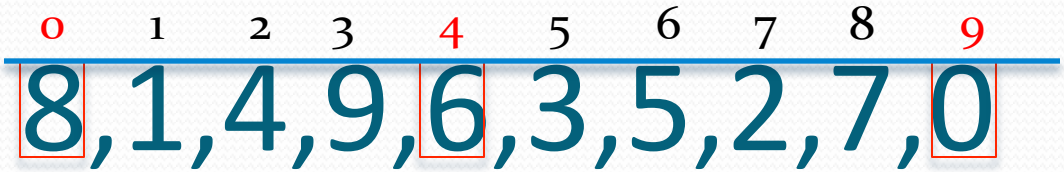
- Selection problem
- Using heap find kth largest (or smallest) in O(N+klogN) time => O(NlogN) for median

Linear-Expected-Time for Selection

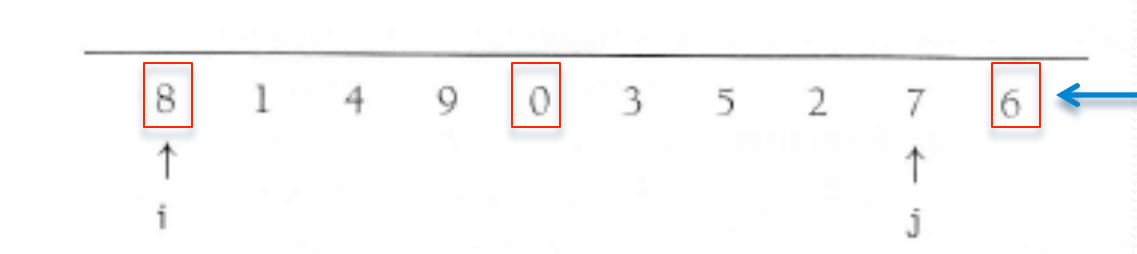
quickselect(S,k)

- If |S|=1, then k=1 and return element of S
  - You can also use CUTOFF for small arrays:

    if |S| <=CUTOFF then sort and return the k_th element
- Pick a pivot v in S
- Partition S-{v} into S1 and S2 as was done in quicksort
- If k<= |S1|, return quickselect(S1, k)
- Else if k==|S1|+1, return pivot (why?)
- Else, return quickselect(S2, k - |S1| -1)



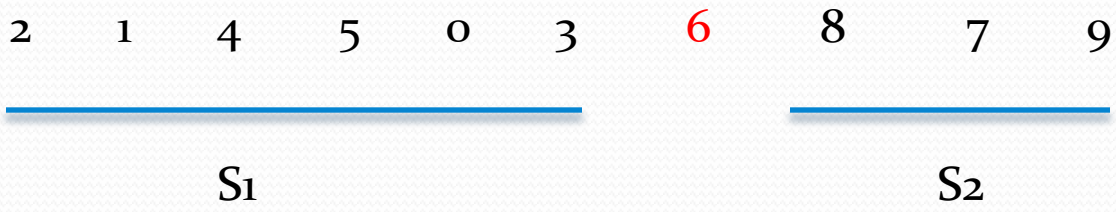
First index: **0**, Last index: **9**,
Middle index: (0+9)/2 = **4**



Take care of median

Compute median (6)

After partitioning:



If k was 6, then solution is pivot (6 is the kth smallest element)
Otherwise, you should only look in S1 or S2 (but not both).
For example if k = 3 you should look in ?
if k = 7 you should look in?

QuickSelect (Driver)

```
/**
 * Quick selection algorithm.
 * Places the kth smallest item in a[k-1].
 * a is an array of Comparable items.
 * k is the desired rank (1 is minimum) in the entire array.
 */
template <typename Comparable>
void QuickSelect( vector<Comparable> & a, int k) {
    QuickSelect(a, 0, a.size() - 1, k);
}
```

QuickSelect (Recursive)

```
void QuickSelect( vector<Comparable> & a, int left, int right, int k ) {
 if (left + 10 > right ) {
       insertionSort(a, left, right);
       return;
 }
 const Comparable &pivot = median3(a, left, right);
 int i = left, j = right - 1;
 while (true) {  // Main partitioning loop.
       while (a[++i] < pivot) {}
       while (pivot < a[--j]) {}
       if (i < j) std::swap(a[i], a[j]);
       else break;
 }
 std::swap(a[i], a[right – 1]);  // Restore pivot

 if (k <= i) QuickSelect(a, left, i – 1, k);
 else if (k > i + 1) QuickSelect(a, i + 1, right, k);
 // Otherwise: k == i + 1,
}
```

QuickSort Partitioning

QuickSelect (Recursive)

```
void QuickSelect( vector<Comparable> & a, int
left, int right, int k) {
 // If small array do insertion sort & exit.
 // QuickSort partitioning step.
 // pivot placed at position i.
 // pivot is the (i+1)th smallest item.

 if (k <= i) QuickSelect(a, left, i - 1, k);
 else if (k > i + 1) QuickSelect(a, i + 1, right, k);
 // Otherwise: k == i + 1 => i == k - 1 contains
 // the k^th smallest item (position a[k - 1]).
}
```

Quick Select analysis

- Worst-case ?
- Average-case?

Quick Select analysis

- Worst-case

$T(N) = T(N-1) + cN, N>=2 => ?$

- Average-case?

Average-Case Quick Select

$$T(|S|) = Tavg(|S_1| \text{ or } |S_2|) + cN,\ N=|S|$$

$$N>=2$$

Average-Case Quick Select

- Recurrence relation becomes:

$$T(N) = (1/N)[\sum_{j=0}^{N-1} T(j)] + cN,\ N>=2$$

or

$$NT(N) = 1\ [\sum_{j=0}^{N-1} T(j)] + cN^2$$

of course this is true for N-1 as well:

$$(N-1)T(N-1) = 1\ [\sum_{j=0}^{N-2} T(j)] + c(N-1)^2$$

Average-Case Quick Select

- By subtraction:

$$NT(N) - (N-1)T(N-1) = 1\left[\sum_{j=0}^{N-1} T(j)\right] - 1\left[\sum_{j=0}^{N-2} T(j)\right] + cN^2 - c(N-1)^2$$

$$= 1\,T(N-1) + 2cN - c \Rightarrow \text{(-c not significant)}$$

$$NT(N) = N\,T(N-1) + 2cN \Rightarrow$$

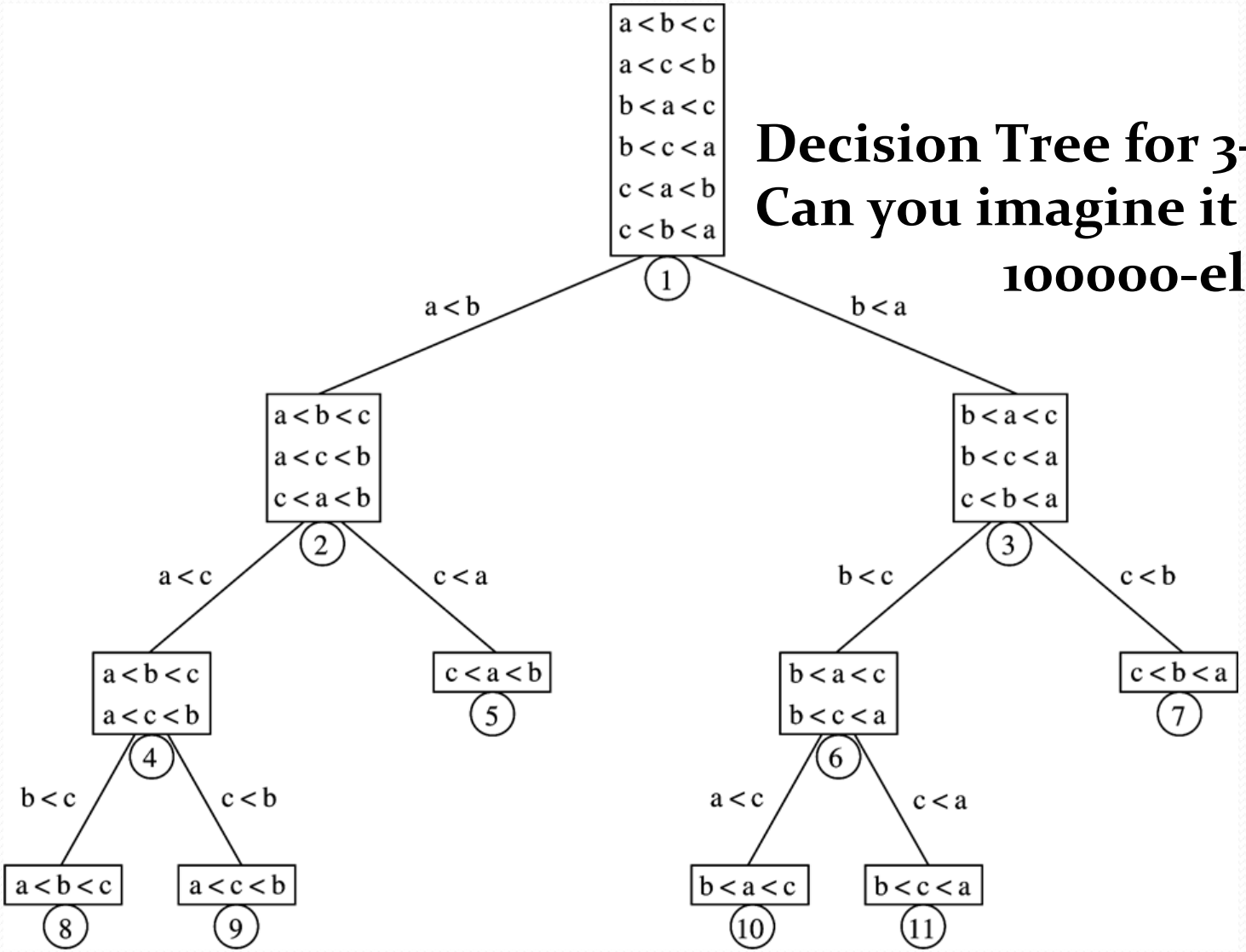
$$T(N) = T(N-1) + 2c \Rightarrow$$

$T(N) = O(N)$

A General Lower Bound for Sorting

- **Theorem:** Any sorting algorithm that uses **only comparisons** requires **Ω(N logN)** comparisons in the worst case !
- Holds true for **average** case as well.
- What does this mean for mergesort, heapsort, quicksort?

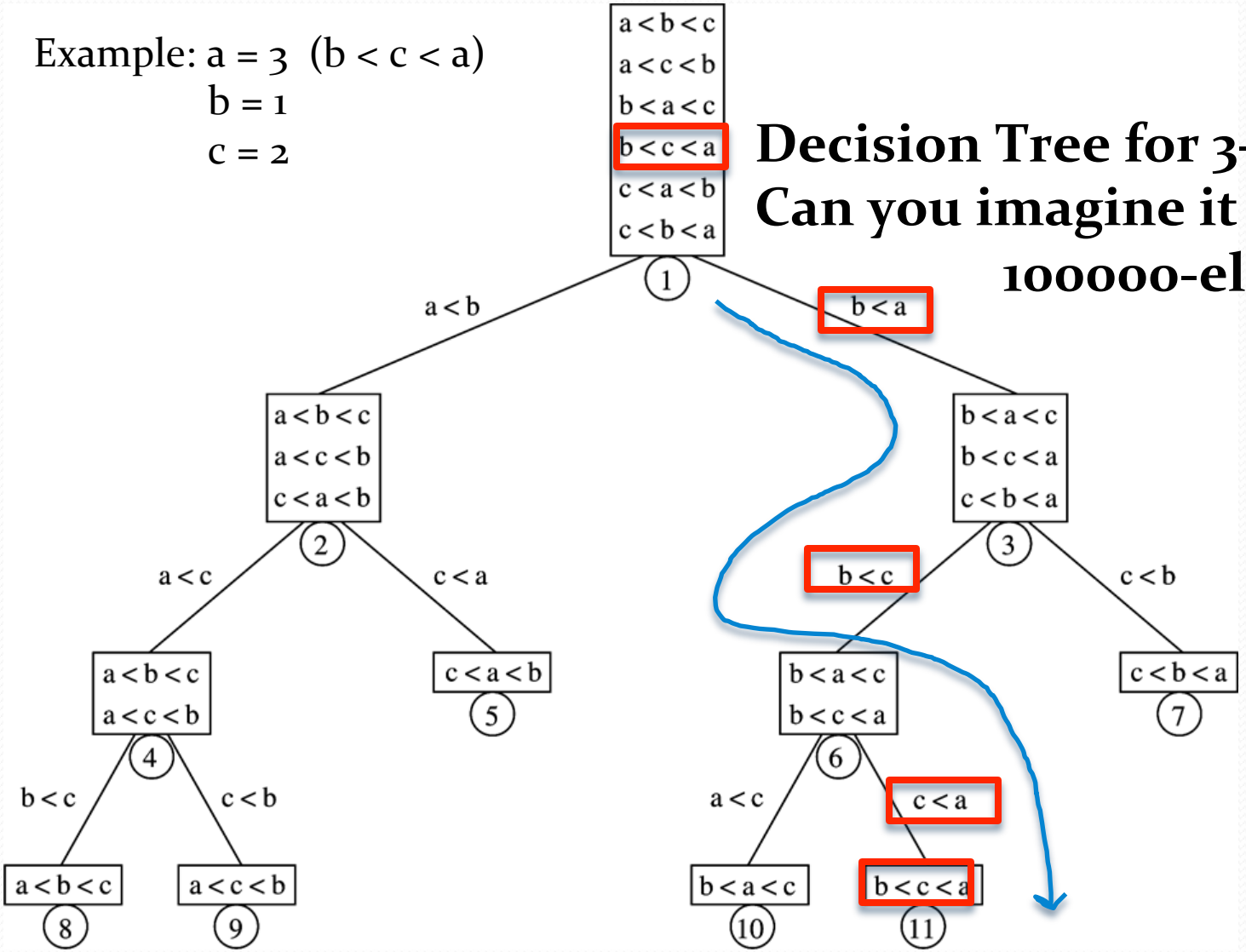
Decision Trees



Decision Tree for 3-element sort
Can you imagine it for
100000-element sort?

Decision Trees

Example: a = 3 (b < c < a)
b = 1
c = 2



Decision Tree for 3-element sort
Can you imagine it for
100000-element sort?

Decision trees

- Every algorithm using comparisons, can be represented by decision tree.
- The length of each path to leaves is the minimum number of comparisons any algorithm needs for a particular input.
- **Information-theoretic bound**
  - Does not give the algorithm to achieve the NlogN bound.
  - It tells you though that this is the **minimum number of comparisons any algorithm will require.**

Proof flow

- Lemma 1: Let T be a binary tree of depth d.
  It has at most 2^d **leaves**.

- Lemma 2: A binary tree of **L** leaves must have
  depth of at least ceil(**logL**),

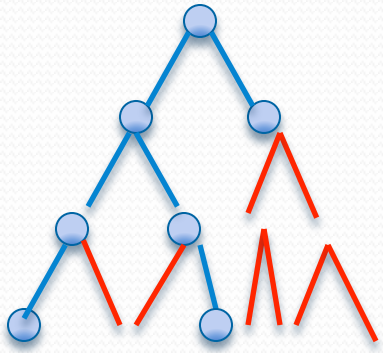
  i.e. d ≥ ceil(logL).

So, if you only know the number of leaves of a tree, you can figure out its minimum depth.

For example, if L = 17, then d >= ceil(log17) = ceil(4.08) =>
d >= 5.

If L = 2^16, then d >=ceil(log2^16) = 16.

Example



Binary tree (blue) of depth d = 3

Full binary tree (blue & red) of depth = 3

Number of nodes of full binary tree:
2^{d+1} = 16

Number of leaves of full binary tree: 2^d = 8

If L is the number of nodes of any tree of depth d

Then L <= 2^d

In this example L (=3) <= 8

Therefore d >= ceil(log L) = ceil(log 3) = ceil(1.5) = 2,
i.e. d >= 2 in this example.

Proof

Lemma 1. A binary tree of depth d has at most 2^d leaves.

Proof (by induction). If $d = 0$ then the tree has one leaf node, the root. If $d > 0$ then the tree has two subtrees of depth up to $d - 1$ with at most 2^{d-1} leaves, by the inductive hypothesis. Since the root itself is not a leaf this means the number of leaves is at most $2 \cdot 2^{d-1} = 2^d$.

□

Proof flow

Theorem 1:

Any sorting algorithm that is using comparisons between elements requires at least **ceil(log(N!))** comparisons in the **worst case**, i.e.

Number of comparisons ≥ **ceil(log(N!))**

Proof flow

Theorem 1:

Any sorting algorithm that is using comparisons between elements requires at least **ceil(log(N!))** comparisons in the **worst case**, i.e.

Number of comparisons ≥ **ceil(log(N!))**

Proof:

A decision tree of N items has N! leaves.

Use Lemma 2 to show that the depth of the decision tree is ≥ ceil(log(N!))

Proof flow

Theorem 2:

Any sorting algorithm that uses only comparisons requires $\Omega(N \log N)$ comparisons in the worst case.

Proof

Theorem 2. Any sorting algorithm that is using comparisons requires $\Omega(n \log n)$ in the worst-case.

Proof. From the previous theorem, $\log(n!)$ comparisons are required.

$$\begin{aligned}
\log(n!) &= \log n + \log(n-1) + \cdots + \log 2 + \log 1 \\
&\geq \log n + \log(n-1) + \cdots \log\left(\frac{n}{2}\right) \\
&\geq \frac{n}{2} \cdot \log\left(\frac{n}{2}\right) \\
&\geq \frac{n}{2}(\log n - \log 2) \\
&\geq \frac{1}{2} n \log n - \frac{1}{2} n \\
&= \Omega(n \log n)
\end{aligned}$$

□

Linear-Time Sorting Algorithms

- Does not only use comparisons to sort.
- Depends on fixed length of inputs
- Can be inefficient for arbitrarily large input values.
- Counting sort, bucket sort, radix sort (4^{th} edition)