



Sorting Algorithms

CptS 223 – Advanced Data Structures

Larry Holder

School of Electrical Engineering and Computer Science
Washington State University



Sorting Problem

- Given array $A[0\dots N-1]$, modify A such that $A[i] \leq A[i+1]$ for $0 \leq i < N-1$
- Internal vs. external sorting
- Stable vs. unstable sorting
 - Equal elements retain original order
- In-place sorting ($O(1)$ extra memory)
- Comparison sorting vs. ???



Sorting Algorithms

- Insertion sort
- Shell sort
- Heap sort
- Merge sort
- Quick sort
- ...
- Simple data structure; focus on analysis



InsertionSort

- In-place
- Stable
- Best case?
- Worst case?
- Average case?

```
InsertionSort (A)
  for p = 1 to N-1
  {
    tmp = A[p]
    j = p
    while (j > 0) and
          (tmp < A[j-1])
    {
      A[j] = A[j-1]
      j = j - 1
    }
    A[j] = tmp
  }
```



ShellSort

- In-place
- Unstable
- Best case

```
ShellSort (A)
  gap = N
  while (gap > 0)
    gap = gap / 2
    B = <A[0],A[gap],A[2*gap],...>
    InsertionSort (B)
```

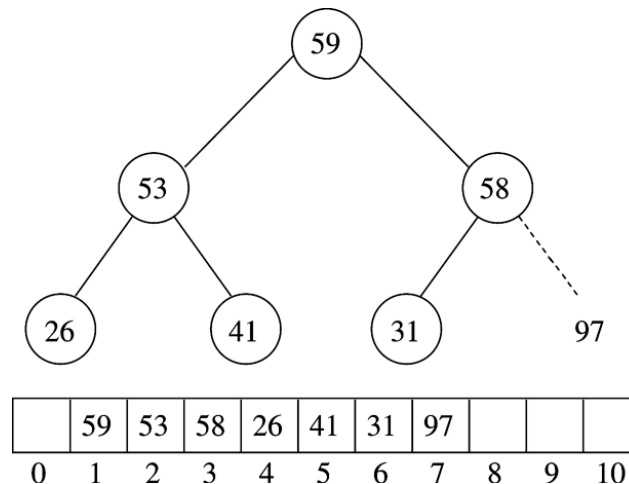
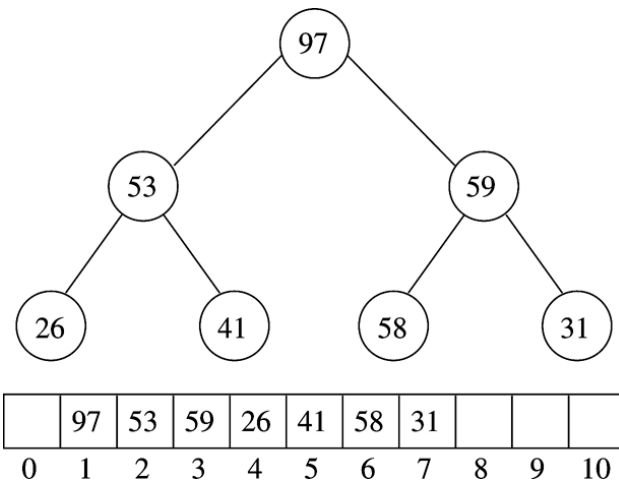
- Sorted: $\Theta(N \log_2 N)$
- Worst case
 - Shell's increments (by 2^k): $\Theta(N^2)$
 - Hibbard's increments (by 2^k-1): $\Theta(N^{3/2})$
- Average case: $\Theta(N^{7/6})$?

HeapSort

- In-place
- Unstable
- All cases
 - $\Theta(N \log_2 N)$

```
HeapSort (A)
  BuildHeap2 (A)
  for j = N-1 downto 1
    swap (A[0], A[j])
    PercolateDown2 (A, 0, j)
```

BuildHeap2 and PercolateDown2 same as before except maintain (parent > children).





MergeSort

- Not in-place
- Stable

Analysis: All cases

$$T(1) = \Theta(1)$$

$$T(N) = 2T(N/2) + \Theta(N)$$

$$T(N) = \Theta(?)$$

```
MergeSort (A)
```

```
    MergeSort2 (A, 0, N-1)
```

```
MergeSort2 (A, i, j)
```

```
    if (i < j)
```

```
        k = (i + j) / 2
```

```
        MergeSort2 (A, i, k)
```

```
        MergeSort2 (A, k+1, j)
```

```
        Merge (A, i, k+1, j)
```

```
Merge (A, i, k, j)
```

```
    Create auxiliary array B
```

```
    Copy elements of sorted A[i..k] and  
    sorted A[k+1..j] into B (in order)
```

```
    A = B
```



QuickSort

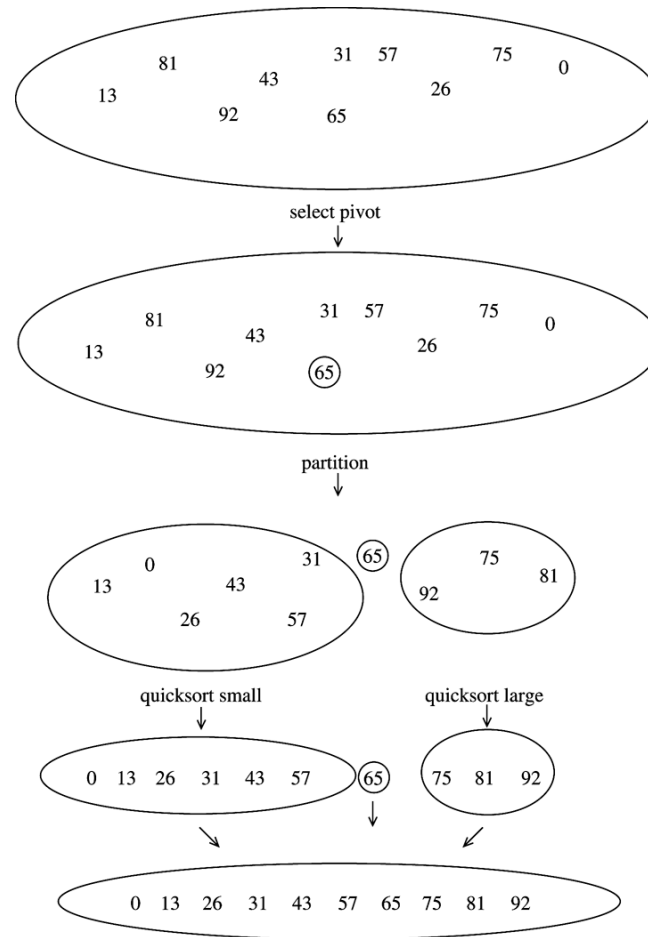
- In-place, unstable
- Like MergeSort, except
 - Don't divide the array in half
 - Partition the array based on elements being less than or greater than some element of the array (the pivot)
- Worst case running time $O(N^2)$
- Average case running time $O(N \log N)$
- Fastest generic sorting algorithm in practice
- Even faster if use simple sort (e.g., InsertionSort) when array is small



QuickSort Algorithm

- Given array S
- Modify S so elements in increasing order
 1. If size of S is 0 or 1, return
 2. Pick any element v in S as the pivot
 3. Partition $S - \{v\}$ into two disjoint groups
 - $S1 = \{x \in (S - \{v\}) \mid x \leq v\}$
 - $S2 = \{x \in (S - \{v\}) \mid x \geq v\}$
 4. Return QuickSort($S1$), followed by v , followed by QuickSort($S2$)

QuickSort Example





Why so fast?

- MergeSort always divides array in half
 - QuickSort might divide array into subproblems of size 1 and $N-1$
 - When?
 - Leading to $O(N^2)$ performance
 - Need to choose pivot wisely (but efficiently)
- MergeSort requires temporary array for merge step
 - QuickSort can partition the array in place
 - This more than makes up for bad pivot choices



Picking the Pivot

- Choosing the first element
 - What if array already or nearly sorted?
 - Good for random array
- Choose random pivot
 - Good in practice if truly random
 - Still possible to get some bad choices
 - Requires execution of random number generator



Picking the Pivot

- Best choice of pivot?
 - Median of array
- Median is expensive to calculate
- Estimate median as the median of three elements
 - Choose first, middle and last elements
 - E.g., $\langle 8, 1, 4, 9, 6, 3, 5, 2, 7, 0 \rangle$
- Has been shown to reduce running time (comparisons) by 14%



Partitioning Strategy

- Partitioning is conceptually straightforward, but easy to do inefficiently
- Good strategy
 - Swap pivot with last element $S[\text{right}]$
 - Set $i = \text{left}$
 - Set $j = (\text{right} - 1)$
 - While ($i < j$)
 - Increment i until $S[i] > \text{pivot}$
 - Decrement j until $S[j] < \text{pivot}$
 - If ($i < j$), then swap $S[i]$ and $S[j]$
 - Swap pivot and $S[i]$



Partitioning Example (cont.)

2 1 4 9 0 3 5 8 7 6 Before second swap
 i j

2 1 4 5 0 3 9 8 7 6 After second swap
 i j

2 1 4 5 0 3 9 8 7 6 Before third swap
 j i

2 1 4 5 0 3 6 8 7 9 After swap with pivot
 i p



Partitioning Strategy

- How to handle duplicates?
- Consider the case where all elements are equal
 - Current approach: Skip over elements equal to pivot
 - No swaps (good)
 - But then $i = (\text{right} - 1)$ and array partitioned into $N-1$ and 1 elements
 - Worst case $O(N^2)$ performance



Partitioning Strategy

- How to handle duplicates?
- Alternative approach
 - Don't skip elements equal to pivot
 - Increment i **while** $S[i] < \text{pivot}$
 - Decrement j **while** $S[j] > \text{pivot}$
 - Adds some unnecessary swaps
 - But results in perfect partitioning for array of identical elements
 - Unlikely for input array, but more likely for recursive calls to QuickSort



Small Arrays

- When S is small, generating lots of recursive calls on small sub-arrays is expensive
- General strategy
 - When $N < \text{threshold}$, use a sort more efficient for small arrays (e.g., InsertionSort)
 - Good thresholds range from 5 to 20
 - Also avoids issue with finding median-of-three pivot for array of size 2 or less
 - Has been shown to reduce running time by 15%



QuickSort Implementation

```
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  }
```

QuickSort Implementation


```
1  /**
2   * Return median of left, center, and right.
3   * Order these and hide the pivot.
4   */
5  template <typename Comparable>
6  const Comparable & median3( vector<Comparable> & a, int left, int right )
7  {
8      int center = ( left + right ) / 2;
9      if( a[ center ] < a[ left ] )
10         swap( a[ left ], a[ center ] );
11     if( a[ right ] < a[ left ] )
12         swap( a[ left ], a[ right ] );
13     if( a[ right ] < a[ center ] )
14         swap( a[ center ], a[ right ] );
15
16         // Place pivot at position right - 1
17     swap( a[ center ], a[ right - 1 ] );
18     return a[ right - 1 ];
19 }
```

8	1	4	9	6	3	5	2	7	0
L				C					R
6	1	4	9	8	3	5	2	7	0
L				C					R
0	1	4	9	8	3	5	2	7	6
L				C					R
0	1	4	9	6	3	5	2	7	8
L				C					R
0	1	4	9	7	3	5	2	6	8
L				C				P	R

```

1  /**
2   * Internal quicksort method that makes recursive calls.
3   * Uses median-of-three partitioning and a cutoff of 10.
4   * a is an array of Comparable items.
5   * left is the left-most index of the subarray.
6   * right is the right-most index of the subarray.
7   */
8  template <typename Comparable>
9  void quicksort( vector<Comparable> & a, int left, int right )
10 {
11     if( left + 10 <= right )
12     {
13         Comparable pivot = median3( a, left, right );
14
15         // Begin partitioning
16         int i = left, j = right - 1;
17         for( ; ; )
18         {
19             while( a[ ++i ] < pivot ) { }
20             while( pivot < a[ --j ] ) { }
21             if( i < j )
22                 swap( a[ i ], a[ j ] );
23             else
24                 break;
25         }
26
27         swap( a[ i ], a[ right - 1 ] ); // Restore pivot
28
29         quicksort( a, left, i - 1 ); // Sort small elements
30         quicksort( a, i + 1, right ); // Sort large elements
31     }
32     else // Do an insertion sort on the subarray
33         insertionSort( a, left, right );
34 }

```



Swap should be compiled inline.



Analysis of QuickSort

- Let i be the number of elements sent to the left partition
- Compute running time $T(N)$ for array of size N
- $T(0) = T(1) = O(1)$
- $T(N) = T(i) + T(N - i - 1) + O(N)$



Analysis of QuickSort

- Worst-case analysis
 - Pivot is the smallest element ($i = 0$)

$$T(N) = T(0) + T(N-1) + O(N)$$

$$T(N) = O(1) + T(N-1) + O(N)$$

$$T(N) = T(N-1) + O(N)$$

$$T(N) = T(N-2) + O(N-1) + O(N)$$

$$T(N) = T(N-3) + O(N-2) + O(N-1) + O(N)$$

$$T(N) = \sum_{i=1}^N O(i) = O(N^2)$$



Analysis of QuickSort

- Best-case analysis

- Pivot is in the middle ($i = N/2$)

$$T(N) = T(N/2) + T(N/2) + O(N)$$

$$T(N) = 2T(N/2) + O(N)$$

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

- Average-case analysis

- Assuming each partition equally likely

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



Comparison Sorting

Sort	Worst Case	Average Case	Best Case	Comments
InsertionSort	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(N)$	Fast for small N
ShellSort	$\Theta(N^{3/2})$	$\Theta(N^{7/6})$?	$\Theta(N \log N)$	Increment sequence?
HeapSort	$\Theta(N \log N)$	$\Theta(N \log N)$	$\Theta(N \log N)$	Large constants
MergeSort	$\Theta(N \log N)$	$\Theta(N \log N)$	$\Theta(N \log N)$	Requires memory
QuickSort	$\Theta(N^2)$	$\Theta(N \log N)$	$\Theta(N \log N)$	Small constants



Comparison Sorting

N	Insertion Sort $O(N^2)$	Shellsort $O(N^{7/6})(?)$	Heapsort $O(N \log N)$	Quicksort $O(N \log N)$	Quicksort (opt.) $O(N \log N)$
10	0.000001	0.000002	0.000003	0.000002	0.000002
100	0.000106	0.000039	0.000052	0.000025	0.000023
1000	0.011240	0.000678	0.000750	0.000365	0.000316
10000	1.047	0.009782	0.010215	0.004612	0.004129
100000	110.492	0.13438	0.139542	0.058481	0.052790
1000000	NA	1.6777	1.7967	0.6842	0.6154

~3 hours

Good sorting applets

- <http://www.sorting-algorithms.com>
- <http://math.hws.edu/TMCM/java/xSortLab/>



Lower Bound on Sorting

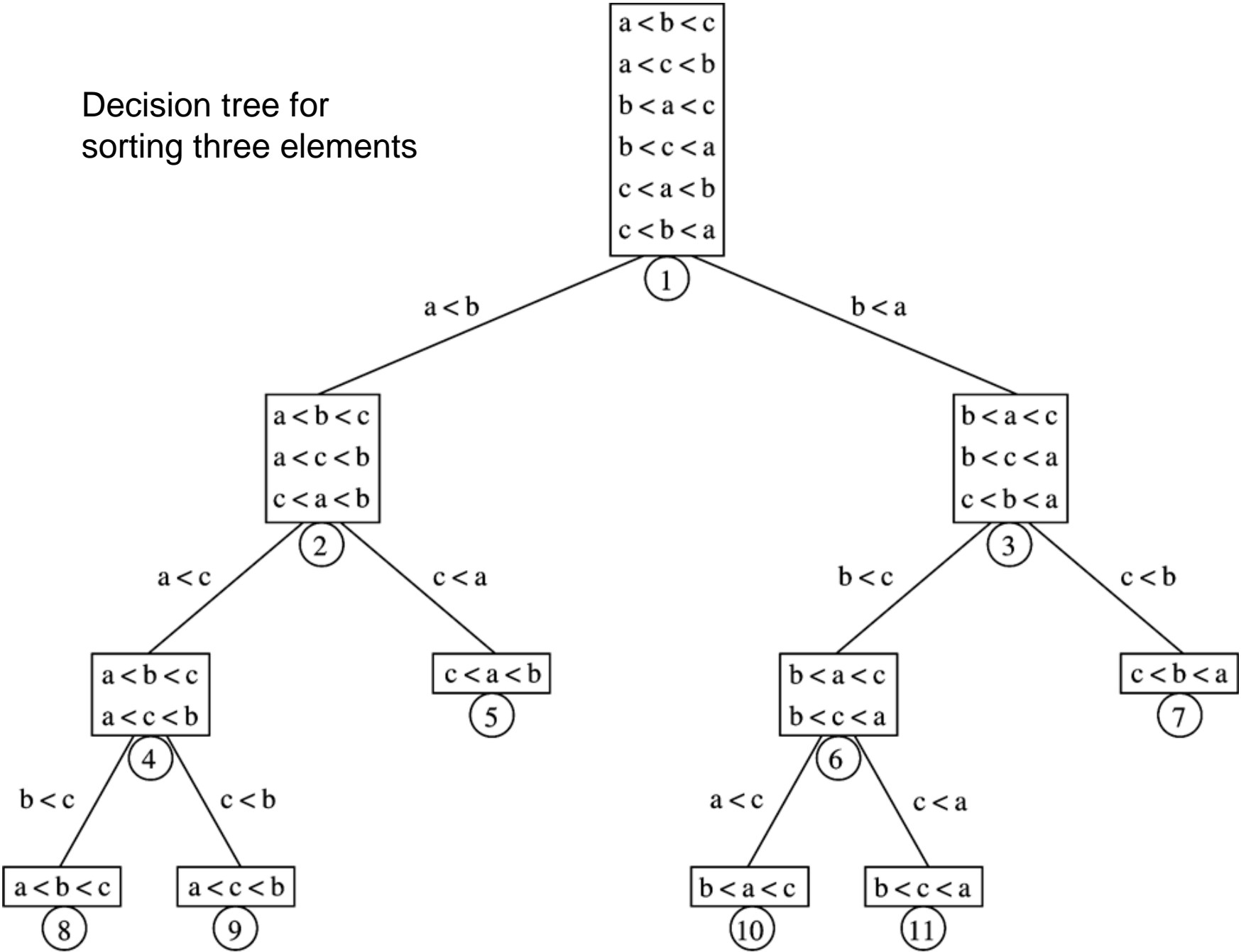
- Best worst-case sorting algorithm (so far) is $O(N \log N)$
- Can we do better?
- Can we prove a lower bound on the sorting problem?
- Preview
 - For comparison sorting, no, we can't do better
 - Can show lower bound of $\Omega(N \log N)$



Decision Trees

- A decision tree is a binary tree
 - Each node represents a set of possible orderings of the array elements
 - Each branch represents an outcome of a particular comparison
- Each leaf of the decision tree represents a particular ordering of the original array elements

Decision tree for sorting three elements





Decision Tree for Sorting

- The logic of every sorting algorithm that uses comparisons can be represented by a decision tree
- In the worst case, the number of comparisons used by the algorithm equals the depth of the deepest leaf
- In the average case, the number of comparisons is the average of the depths of all leaves
- There are $N!$ different orderings of N elements



Lower Bound for Comparison Sorting

- Lemma 7.1: A binary tree of depth d has at most 2^d leaves
- Lemma 7.2: A binary tree with L leaves must have depth at least $\lceil \log L \rceil$
- Thm. 7.6: Any comparison sort requires at least $\lceil \log(N!) \rceil$ comparisons in the worst case



Lower Bound for Comparison Sorting

- Thm. 7.7: Any comparison sort requires $\Omega(N \log N)$ comparisons
- Proof (recall Stirling's approximation)

$$N! = \sqrt{2\pi N} (N/e)^N (1 + \Theta(1/N))$$

$$N! > (N/e)^N$$

$$\log(N!) > N \log N - N \log e = \Theta(N \log N)$$

$$\log(N!) > \Theta(N \log N)$$

$$\therefore \log(N!) = \Omega(N \log N)$$



Linear Sorting

- Some constraints on input array allow faster than $\Theta(N \log N)$ sorting (no comparisons)
- CountingSort¹
 - Given array A of N integer elements, each less than M
 - Create array C of size M , where $C[i]$ is the number of i 's in A
 - Use C to place elements into new sorted array B
 - Running time $\Theta(N+M) = \Theta(N)$ if $M = \Theta(N)$

¹ Weiss incorrectly calls this BucketSort.



Linear Sorting

- BucketSort

- Assume N elements of A uniformly distributed over the range $[0,1)$
- Create N equal-sized buckets over $[0,1)$
- Add each element of A into appropriate bucket
- Sort each bucket (e.g., with InsertionSort)
- Return concatenation of buckets
- Average case running time $\Theta(N)$
 - Assumes each bucket will contain $\Theta(1)$ elements



Sorting in the STL

■ Vectors

```
#include <algorithm>
void sort (iterator start, iterator end);
void sort (iterator start, iterator end, Comparator cmp);
void stable_sort (iterator start, iterator end);
void stable_sort (iterator start, iterator end, Comparator cmp);
```

■ STL **sort** uses IntrospectiveSort

- QuickSort until recursion depth of $(\log N)$
 - Median-of-3 pivot selection
- Then HeapSort

■ STL **stable_sort** uses MergeSort



Sorting in the STL

- Lists

```
#include <list>  
void sort ();  
void sort (Comparator cmp);
```

- Uses MergeSort
 - Stable
 - No auxiliary array needed
- Iterators left intact



External Sorting

- What is the number of elements N we wish to sort do not fit in memory?
- Obviously, our existing sort algorithms are inefficient
 - Each comparison potentially requires a disk access
- Once again, we want to minimize disk accesses



External MergeSort

- N = number of elements in array A to be sorted
- M = number of elements that fit in memory
- $K = \lceil N / M \rceil$
- Approach
 - Read in M amount of A , sort it using QuickSort, and write it back to disk: $O(M \log M)$
 - Repeat above K times until all of A processed
 - Create K input buffers and 1 output buffer, each of size $M/(K+1)$
 - Perform a K -way merge: $O(N)$
 - Update input buffers one disk-page at a time
 - Write output buffer one disk-page at a time



External MergeSort

- $T(N, M) = O(K * M \log M) + N$
- $T(N, M) = O((N/M) * M \log M) + N$
- $T(N, M) = O(N \log M) + N$
- $T(N, M) = O(N \log M)$
- Disk accesses (all sequential)
 - $P =$ page size
 - Accesses = $4N/P$ (read-all/write-all twice)



Sorting: Summary

- Need for sorting is ubiquitous in software
- Optimizing the sort algorithm to the domain is essential
- Good general-purpose algorithms available
 - QuickSort
- Optimizations continue...
 - Sort benchmark
<http://www.hpl.hp.com/hosted/sortbenchmark>