Internal Sorting

- Two general types of sorting:
  - ☐ **Internal sorting**, in which all elements are sorted in main memory
  - ☐ **External sorting**, in which elements must be sorted on disk or tape
- The focus here is on internal sorting techniques
- Sorting Classifications:
  - ☐ Exchange sorts that all run in $O(n^2)$
    - ∘ Insert sort, bubble sort, selection sort
  - ☐ Shell sort: an in-the-middle sort that runs in $O(n^{1.5})$ or $o(n^2)$
  - ☐ Efficient sorts that run in $O(n \log n)$
    - ∘ Heap sort, merge sort, quicksort
  - ☐ Special-purpose sorts that run in quicker time:
    - ∘ Bin sort, bucket sort, radix sort
- The **problem** of sorting, in general, is $\Omega(n \log n)$
  - ☐ Special cases are allowed to take less time because they are special cases, not general

The Sorting Problem

- Each **record** contains a field called the **key**
- Definition of the sorting problem:
  - ☐ Given a sequence of records $r_1, r_2, \ldots, r_n$ with key values $k_1, k_2, \ldots, k_n$
  - ☐ Arrange the records into any order s such that
  - ☐ $r_{s_1}, r_{s_2}, \ldots, r_{s_n}$ have keys obeying the property $k_{s_1} \leq k_{s_2} \leq \ldots \leq k_{s_n}$
- Duplicate key values may be (are usually) allowed
  - ☐ Implicit ordering of duplicates:
    - ∘ After sorting, duplicate keys remain in the order in which they occurred in the input
    - ∘ This *may* be desirable, and the property is called **stability**

Comparing Performance of Sorting Algorithms

- Most obvious method: run two sorts on identical input and compare times
  - ☐ Problem: running time may depend on specifics of input values
  - ☐ Factors: number of records, key size, record size, range of key values, amount by which records are out of order
- Analytically, sorts are usually compared using two measures:
  - ☐ the number of comparisons
  - ☐ the number of swaps
- Common assumptions:
  - ☐ Each sort is passed an array containing the elements
  - ☐ n is the number of elements to be sorted
  - ☐ While `int` is the type in all examples, assume that any complex type implementing binary comparators (e.g. $<$ or $\leq$) can be sorted

Insertion Sort

One of the simplest sorting algorithms to implement.

- Characteristics:
  - ☐ Makes $n-1$ passes
  - ☐ For a given pass numbered p, elements in positions 0 through p are ensured to be sorted
- Code example:

```
//
// A hybrid of Shaffer's and other code
//
void insert_sort(int *array, int n) {
  for (int i = 1; i < n; i++) {
    for (int j = i;
          (j > 0) && (array[j] < array[j - 1]);
          j--)
      swap(array[j],array[j-1]);
  }
}
```

Insertion Sort

- Example sort

	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	$i=6$	$i=7$
42							
20							
17							
13							
28							
14							
23							
15							

- Time complexity
  - ☐ Best case:
  - ☐ average case
  - ☐ worst case

Bubble Sort

Another very simple sort.

- Characteristics:
  - ☐ Also makes $n-1$ passes, each pass represents a position
  - ☐ For a given pass numbered i, "bubble-up" the element in the upper part of the array belonging in position i
- Code example:

```
void bubble_sort(int *array, int n) {
  for (int i = 0; i < n - 1; i++)
    for (int j = n - 1; j > i; j--)
      if (array[j] < array[j - 1])
        swap(array[j], array[j - 1]);
}
```

Bubble Sort

- Example sort

	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	$i=6$	$i=7$
42							
20							
17							
13							
28							
14							
23							
15							

- Time complexity
  - ☐ Best case:
  - ☐ average case
  - ☐ worst case

Selection Sort

Yet another very simple sort.

- Characteristics:
  - ☐ Also makes $n-1$ passes, each pass represents a position
  - ☐ For a given pass numbered i, find the element in the upper part of the array belonging in position i. Only swap after that element is found.
- Code example:

```
void selection_sort(int *array, int n) {
  for (int i = 0; i < n - 1; i++) {
    int lowindex = i;
    for (int j = n - 1; j > i; j--)
      if (array[j] < array[lowindex])
        lowindex = j;
    swap(array[i], array[lowindex]);
  }
}
```

Selection Sort

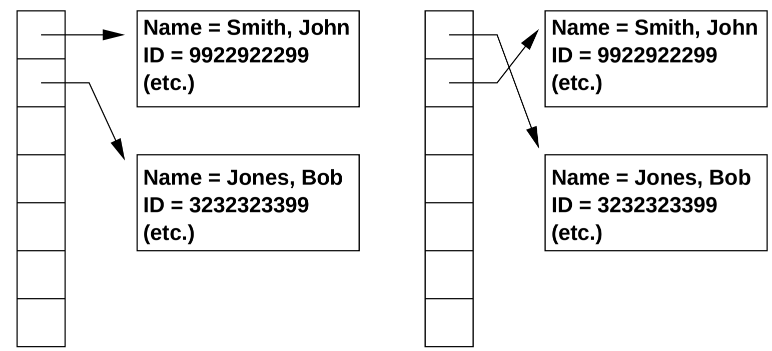
- Example sort

	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	$i=6$	$i=7$
42							
20							
17							
13							
28							
14							
23							
15							

- Time complexity
  - ☐ Best case:
  - ☐ average case
  - ☐ worst case

Keeping Swap Costs Low

- Some sorts aren't practical in an array
  - ☐ Desired: a means to swap objects without actually moving them
  - ☐ Pointer swapping can accomplish this



- Examples:
  - ☐ Array of integers: no pointers necessary
  - ☐ Array of student records: pointers necessary

Exchange Sorting

- An **exchange** is a swap of adjacent records.
  - ☐ Insert, bubble, and selection sort (basically) perform exchanges to move data
  - ☐ Therefore, they are sometimes called the **exchange sorts**.
- Exchange sort performance is measured based on **inversions**.
  - ☐ An inversion is any pair of array elements out of order with respect to each other
    - ∘ That is, consider an ordered pair (i, j) for which $i < j$ and $array[i] > array[j]$
  - ☐ Observation: the number of inversions is exactly the number of exchanges used by insert sort
    - ∘ If the number of inversions is n, then insert sort is $O(n)$
    - ∘ The average number of inversions in an array of n distinct elements is $n(n-1)/4$ (a theorem)
    - ∘ Exchange sorts are therefore $\Omega(n^2)$ (another theorem)

Shellsort

The first algorithm to break the quadratic time barrier.

- Characteristics:
  - ☐ Consists of $\log n$ phases
  - ☐ Works by comparing (and swapping) distant elements
  - ☐ Each phase reduces the distance between compared elements by half
  - ☐ Also called diminishing increment sort
- Code example:

```
void shellsort(int *array, int n) {
  int j;
  for (int gap=n / 2; gap > 0; gap /= 2)
    for (int i = gap; i < n; i++)
    {
      tmp = array[i];
      for (j = i; j >= gap && tmp < a[j - gap]; j -= gap)
        array[j] = array[j - gap];
      array[j] = tmp;
    }
}
```

Shellsort

- Shellsort has been shown to be $O(n^{1.5})$ or $o(n^2)$
- Example: sort the list 59, 20, 17, 13, 28, 14, 23, 83, 36, 98, 11, 70, 65, 41, 42, 15

Quicksort

Based on the concept of divide and conquer: a list is divided into sublists divided by a pivot.

- Fastest known sorting algorithm
- Basic algorithm: given a list S of numbers:
  - ☐ Choose a **pivot** v from some location in the list.
  - ☐ Partition the list into two sublists separated by the pivot:
    - ◦ S_1 is the sublist having values $> v$
    - ◦ S_2 is the sublist having values $< v$
  - ☐ Quicksort is called recursively on the sublists (when it returns, S_1 and S_2 will be sorted)
  - ☐ The sorted list is S_1 followed by v followed by S_2.
  - ☐ Recursion process stops when a list length of 0 or 1 is reached

Quicksort

- Code example: initial call would be

```
qsort(array,0,n-1);

void qsort(int *array, int left, int right) {
  int pivot = findpivot(array,left,right);
  swap(array,pivot,right);
  int k = partition(array,left-1,right,array[right]);
  swap(array,k,right);
  if ((k - left) > 1)
    qsort(array,left,k-1);
  if ((right - k) > 1)
    qsort(array,k+1,right);
}

int findpivot(int *array, int i, int j) {
  return (i + j) / 2;
}

int partition(int *a, int l, int r, int & pivot) {
  do {
    while (array[++l] < pivot);
    while (r && array[--r] > pivot);
    swap(array,l,r);
  } while (l < r);
  swap(array,l,r);
  return l;
```


Quicksort

- One pivot and partition run:

position	0	1	2	3	4	5	6	7	8	9
initial list	72	6	57	88	60	42	83	73	48	85
partition swap	72	6	57	88	85	42	83	73	48	60
partition:										
pass 1	72	6	57	88	85	42	83	73	48	60
	l								r	
swap 1	48	6	57	88	85	42	83	73	72	60
pass 2	48	6	57	88	85	42	83	73	72	60
				l		r				
swap 2	48	6	57	42	85	88	83	73	72	60
pass 3	48	6	57	42	85	88	83	73	72	60
				r	l					
swap 3	48	6	57	85	42	88	83	73	72	60
reverse swap	48	6	57	42	85	88	83	73	72	60
Partition relocation	48	6	57	42	85	88	83	73	72	60
					x					x
	48	6	57	42	60	88	83	73	72	85

- All values less than 60 are now to its left

- All values greater than 60 are now to its right

Cost of Quicksort

- Best case: always partition in half
  - ☐ Cost is $O(n \log n)$
- Worst case: a bad parttion
  - ☐ Cost is $O(n^2)$
- Average case:
  - ☐ Cost is $O(n \log n)$
- Quicksort Optimizations:
  - ☐ Choose a better pivot
  - ☐ Use a better algorithm for small sublists
  - ☐ Eliminate recursion

Mergesort

Based on the concept of merging two sorted lists.

- The general principles:
  - ☐ Each list is assumed sorted
  - ☐ Merge of two lists is accomplished in one pass
  - ☐ Output of merge is placed into a third list
- Pseudocode algorithm:

```
list mergesort(list inlist) {
  if (length(inlist) == 1)
    return inlist;
  list l1 = first half of inlist;
  list l2 = second half of inlist;
  return merge(mergesort(l1),mergesort(l2));
}
```

- Example:

36 20 17 13 28 14 23 15

20	36	13	17	14	28	15	23
13	17	20	36	14	15	23	28
13	14	15	17	20	23	28	36

Heapsort

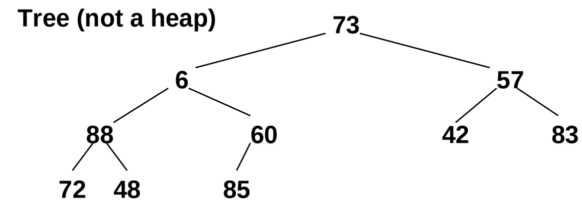
Heapsort uses a max heap.

- The general procedure:
  - ☐ Read in n elements
  - ☐ Build the heap
  - ☐ Call `deleteMax` n times in a row
  - ☐ The array is sorted at that point.
- Code example:

```
void heapsort(int *array, int n) {
  heap H(array,n,n);
  for (int i = 0; i < n; i++)
    H.deleteMax();
}
```


Heapsort Example

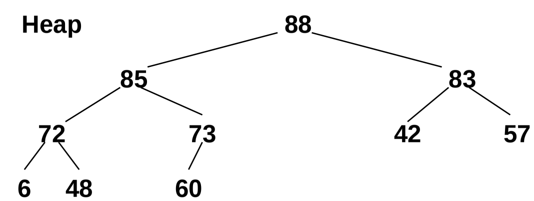
- Sort the list 73, 6, 57, 88, 60, 42, 83, 72, 48, 85
  - ☐ Before buildHeap:



73	6	57	88	60	42	83	72	48	85

Array contents

  - ☐ After buildHeap:

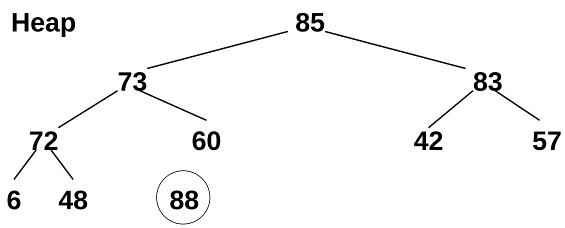


88	85	83	72	73	42	57	6	48	60

Array contents

Heapsort Example (cont.)

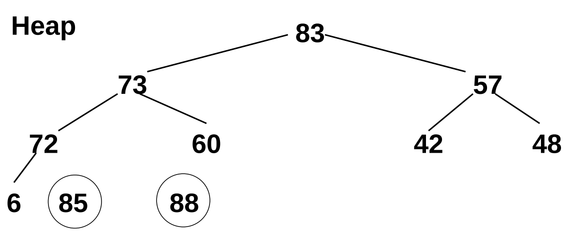
- After first `deleteMax`:



85	73	83	72	60	42	57	6	48	88

Array contents

- After second `deleteMax`:



83	73	57	72	60	42	48	6	85	88

Array contents

Binsort

This is a special-purpose, simple and very efficient sort.

- Given an unsorted array `A`
  - ☐ Let `B` be the array into which the data is sorted
  - ☐ The binsort code is:

```
for (i = 0; i < n; i++)
  B(A[i]) = A[i];
```

- Time complexity: $\Theta(n)$
- Why it isn't general:
  - ☐ No comparisons are performed
  - ☐ It only works on a specific list of numbers:
    - ∘ Array `A` contains exactly n elements
    - ∘ Array `A` must contain a permutation of the numbers from 0 to n
- Improvements:
  - ☐ Make each bin the head of a list
  - ☐ Allow more keys than records

Improved Binsort

- Code:

```
void binsort(int *A, int n) {
  List B[MaxKeyValue];  // an array of lists.
  int item;
  for (i = 0; i < n; i++)
    B[A[i]].append(A[i]);
  for (i = 0; i < MaxKeyValue; i++)
    for (B[i].setFirst(); B[i].isInList(); B[i].next())
      cout << B[i].value() << endl;
}
```

- Cost appears to be $\Theta(n)$
  - ☐ Actual cost also depends on `MaxKeyValue`

Bucket Sort

This is a simple generalization of Binsort

- Each bin is associated with a range of values:
- Assign records to bins
  - ☐ Rely on some other sorting technique to sort each bin
  - ☐ The other sorting technique is hopefully very efficient.
- Example:
  - ☐ Given a sequence of numbers between 0 and 99 inclusive
  - ☐ Use 10 bins
  - ☐ Assign numbers as follows: $bin = key$ mod 10

Radix Sort

This is one specific type of bin sort.

- General idea:
  - ☐ Bin computations are based on the key's radix (its base)
  - ☐ Bins are computed in a series of steps using operations mod base
  - ☐ Example: for base 10 numbers, there are 10 bins and computations are done mod 10
- Example: sort 26, 93, 3, 97, 15, 77, 23, 48, 82, 87, 55, 36, 54, 46
  - ☐ All keys are in the range 0 through $r^2 - 1$ (i.e., $0 \leq \text{key} \leq 99$)
  - ☐ First pass: compute bin = key mod r (key mod 10), append to that bin
  - ☐ Second pass: compute bin = key/10 mod r (key/10 mod 10), append to that bin

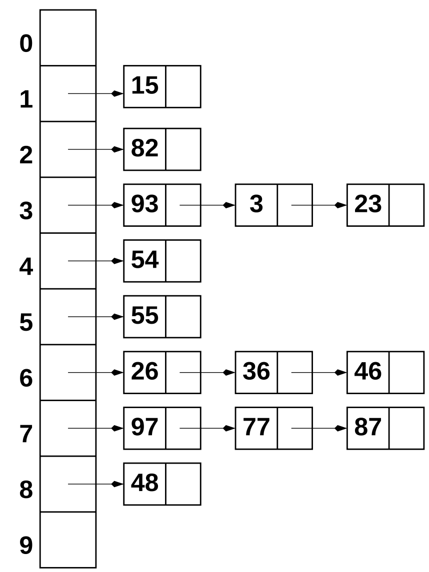
Radix Sort

- Example:

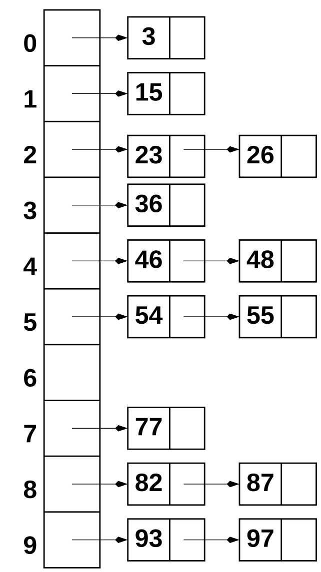
Initial list: 26, 93, 3, 97, 15, 77, 23, 48, 82, 87, 55, 36, 54, 46

List after second pass: 3, 15, 23, 26, 36, 46, 54, 55, 77, 82, 87, 93, 97

First pass (right digit)



Second pass (left digit)



Empirical Comparison

Which algorithm is the fastest?

- Analysis reveals several classes of algorithms but doesn't distinguish among those in a class
- Some times from Figure 7.13: (data is lists of integers, all times are in seconds)

	Size					
Sort	**1K**	**10K**	**100K**	**1M**	**U**	**D**
Insert	2.86	352.1	47241	-	0.0	803.0
Bubble	9.18	1066.1	123808	-	513.5	812.9
Selection	5.82	563.5	69437	-	577.8	560.8
Shell	5.50	9.9	170	3080	2.8	6.1
Quick	0.33	3.8	49	600	1.7	2.2
Quick/O	0.27	3.3	44	550	1.7	1.6
Merge	0.61	60.0	105	1430	6.0	6.1
Heap	0.38	47.2	94	1650	5.0	5.0
Radix/8	2.31	23.6	241	2470	23.6	23.6

- U and D columns: data consisted of 10,000 previously sorted in increasing (Up) or decreasing (Down) order
  - ☐ Note insert sort performance for U
  - ☐ Why is quicksort so good for these two?

General Lower Bound for Sorting

- It is possible to prove a lower bound for all general-purpose sorting algorithms
- Facts:
  - ☐ Sorting is $O(n \log n)$
  - ☐ Sorting I/O takes $\Omega(n)$ time
  - ☐ This give a "cheap" lower found of $\Omega(n)$
- It can be shown:
  - ☐ The **problem** of sorting is $\Omega(n \log n)$
  - ☐ Form of the proof:
    - ◦ Comparison-based sorting can be modeled by a binary tree
    - ◦ It can be shown the tree must have $\Omega(n!)$ leaves
    - ◦ Thus the tree must have $\Omega(n \log n)$ levels, representing the number of comparisons