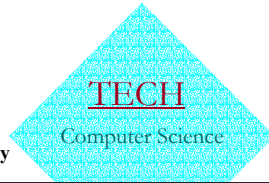


## Problem: Sorting

- arranging elements of set into order
- Algorithm design technique:
  - Divide and Conquer
- Solution:
  - Insertion Sort
  - Quicksort
  - Mergesort
  - Heapsort
  - Shellsort
  - Radix Sorting
- Optimality:
  - Lower bounds for Sorting by Comparison of Keys



## Application of Sorting

- For searching on unsorted data by comparing keys, optimal solutions require  $\theta(n)$  comparisons.
- For searching on sorted data by comparing keys, optimal solutions require  $\theta(\log n)$  comparisons.
- Sorting data for users
- More...

## Insertion Sort

- Strategy:
  - Insertion of an element in proper order:
  - Begin with a sequence E of n elements in arbitrary order
  - Initially assume the sorted segment contains first element
  - Let x be the next element to be inserted in sorted segment, pull x “out of the way”, leaving a vacancy
  - repeatedly compare x to the element just to the left of the vacancy, and as long as x is smaller, move that element into the vacancy,
  - else put x in the vacancy,
  - repeat the next element that has not yet examined.

## Insertion Sort: Algorithm

- Input
  - E, an array of elements, and  $n \geq 0$ , the number of elements. The range of indexes is 0, ..., n-1
- Output
  - E, with elements in nondecreasing order of their keys
- void insertionSort(Element[] E, int n)
  - int xindex;
  - for (xindex = 1; xindex < n; xindex++)
    - > Element current = E[xindex];
    - > key x = current.key
    - > int xLoc = shiftVacRec(E, xindex, x);
    - > E[xLoc] = current;
  - return;

## Insertion Sort: Specification for subroutine

- Specification
  - int shiftVacRec(Element[] E, int vacant, Key x)
  - Precondition
    - > Vacant is nonnegative
  - Postconditions
    - > 1. Elements in E at indexes less than xLoc are in their original positions and have keys less than or equal to x
    - > 2. Elements in E at positions xLoc+1, ..., vacant are greater than x and were shifted up by one position from their positions when shiftVacRec was invoked.

## Insertion Sort: Algorithm shiftVacRec

- int shiftVacRec(Element[] E, int vacant, Key x)
  - int xLoc;
  - if (vacant == 0)
    - > xLoc = vacant;
  - else if (E[vacant-1].key <= x)
    - > xLoc = vacant;
  - else
    - > E[vacant] = E[vacant-1];
    - > xLoc = shiftVacRec(E, vacant-1, x);
  - return xLoc

## Insertion Sort: Analysis

- Worst-Case Complexity
  - $W(n) = \sum_{i=1}^{n-1} \sum_{j=i}^{n-1} 1 = n(n-1)/2 \in \theta(n^2)$
- Average Behavior
  - average number of comparisons in shiftVacRec
  - $1/(i+1) \sum_{j=i}^{n-1} (j) + i/(i+1) = i/2 + 1 - 1/(i+1)$
  - $A(n) = \sum_{i=1}^{n-1} [1/2 + 1 - 1/(i+1)] \approx (n^2)/4$

## Insertion Sort: Optimality

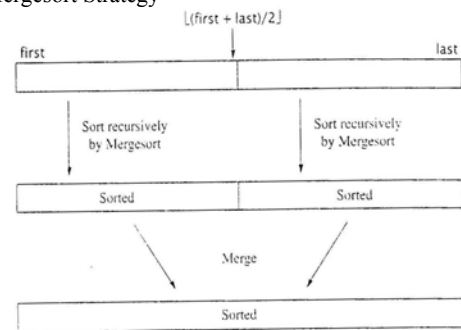
- Theorem 4.1
  - Any algorithm that sorts by comparison of keys and removes at most one inversion after each comparison must do at least  $n(n-1)/2$  comparisons in the worst case and at least  $n(n-1)/4$  comparisons on the average (for  $n$  elements)
- Proof...
- Insertion Sort is optimal for algorithms that works “locally” by interchanging only adjacent elements.
- But, it is not the best sorting algorithm.

## Algorithm Design Technique: Divide and Conquer

- It is often easier to solve several small instances of a problem than one large one.
  - > divide the problem into smaller instances of the same problem
  - > solve (conquer) the smaller instances recursively
  - > combine the solutions to obtain the solution for original input
- Solve(I)
  - $n = \text{size}(I)$
  - if ( $n \leq \text{smallsize}$ )
    - > solution = directlySolve(I);
  - else
    - > divide I into  $I_1, \dots, I_k$ .
    - > for each  $i$  in  $\{1, \dots, k\}$ 
      - $S_i = \text{solve}(I_i)$ ;
    - > solution = combine( $S_1, \dots, S_k$ );
  - return solution;

## Using Divide and Conquer: Mergesort

- Mergesort Strategy



## Algorithm: Mergesort

- Input: Array  $E$  and indexes  $\text{first}$ , and  $\text{last}$ , such that the elements  $E[i]$  are defined for  $\text{first} \leq i \leq \text{last}$ .
- Output:  $E[\text{first}], \dots, E[\text{last}]$  is sorted rearrangement of the same elements
- void mergeSort(Element[]  $E$ , int  $\text{first}$ , int  $\text{last}$ )
  - if ( $\text{first} < \text{last}$ )
    - > int  $\text{mid} = (\text{first} + \text{last}) / 2$ ;
    - > mergeSort( $E$ ,  $\text{first}$ ,  $\text{mid}$ );
    - > mergeSort( $E$ ,  $\text{mid} + 1$ ,  $\text{last}$ );
    - > merge( $E$ ,  $\text{first}$ ,  $\text{mid}$ ,  $\text{last}$ );
  - return;
- $W(n) = W(n/2) + W(n/2) + W_{\text{merge}}(n) \in \theta(n \log n)$ 
  - $W_{\text{merge}}(n) = n - 1$
  - $W(1) = 0$

## Merging Sorted Sequences

- Problem:
  - Given two sequences  $A$  and  $B$  sorted in nondecreasing order, merge them to create one sorted sequence  $C$
- Strategy:
  - determine the first item in  $C$ : It is the minimum between the first items of  $A$  and  $B$ . Suppose it is the first items of  $A$ . Then, rest of  $C$  consisting of merging rest of  $A$  with  $B$ .

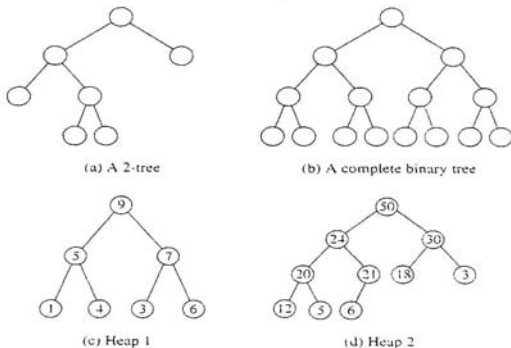
## Algorithm: Merge

- Merge(A, B, C)
  - if (A is empty)
    - > rest of C = rest of B
  - else if (B is empty)
    - > rest of C = rest of A
  - else if (first of A <= first of B)
    - > first of C = first of A
    - > merge (rest of A, B, rest of C)
  - else
    - > first of C = first of B
    - > merge (A, rest of B, rest of C)
  - return
- $W(n) = n - 1$

## Heap and Heapsort

- A Heap data structure is a binary tree with special properties:
  - Heap Structure
  - Partial order tree property
- Definition: Heap Structure
  - A binary tree T is a heap structure if and only if it satisfies the following conditions: (h = height of the tree)
    - > 1. T is complete at least through depth h-1
    - > 2. All leaves are at depth h or h-1
    - > 3. All paths to leaf of depth h are to the left of all parts to a leaf of depth h-1
  - Such a tree is also called a left-complete binary tree.
- Definition: Partial order tree property
  - A tree T is a (maximizing) partial order tree if and only if the key at any node is greater than or equal to the keys at each of its children (if it has any)

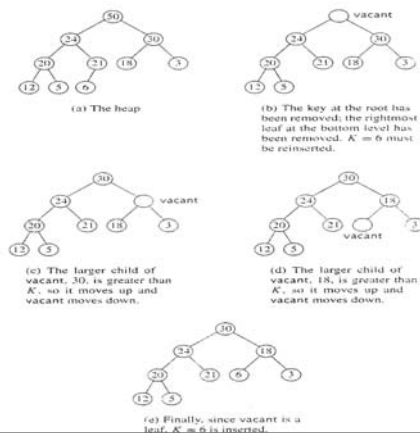
## e.g. Heaps (or not)



## Heapsort Strategy

- If the elements to be sorted are arranged in a heap,
- then we can build a sorted sequence in reverse order by repeatedly removing the element from the root,
- rearranging the remaining elements to reestablish the partial order tree property, and so on.
- How does it work?

## Heapsort in action



## Heapsort Outlines

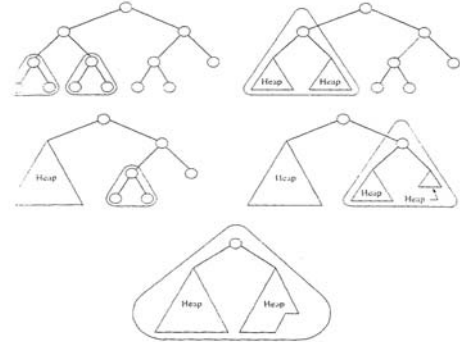
- heapSort(E, n) // Outline
  - construct H from E, the set of n elements to be sorted
  - for (i = n; i >= 1; i--)
    - > curMax = getMax(H)
    - > deleteMax(H);
    - > E[i] = curMax;
- deleteMax(H) // Outline
  - copy the rightmost element of the lowest level of H into K
  - delete the rightmost element on the lowest level of H
  - fixHeap(H, K);

## Fixheap Outline

- `fixHeap(H, K)` // Outline
  - **if (H is a leaf)**
    - > insert K in `root(H)`;
  - **else**
    - > set `largerSubHeap` to `leftSubtree(H)` or `rightSubtree(H)`, whichever has larger key at its root. This involves one key comparison.
    - > if  $(K.key \geq \text{root}(\text{largerSubHeap}.key))$ 
      - insert K in `root(H)`;
    - > **else**
      - insert `root(largerSubHeap)` in `root(H)`;
      - `fixHeap(largerSubHeap, K)`;
  - **return;**
- `FixHeap` requires  $2h$  comparisons of keys in the worst case on a heap with height  $h$ .  $W(n) \approx 2 \lg(n)$

## Heap construction Strategy (divide and conquer)

- base case is a tree consisting of one node



## Construct Heap Outline

- **Input:** A heap structure  $H$  that does not necessarily have the partial order tree property
- **Output:**  $H$  with the same nodes rearranged to satisfy the partial order tree property
- `void constructHeap(H)` // Outline
  - **if (H is not a leaf)**
    - > `constructHeap` (left subtree of  $H$ );
    - > `constructHeap` (right subtree of  $H$ );
    - > Element  $K = \text{root}(H)$ ;
    - > `fixHeap(H, K)`;
  - **return;**
- $W(n) = W(n-r-1) + W(r) + 2 \lg(n)$  for  $n > 1$
- $W(n) \in \theta(n)$ ; heap is constructed in linear time.

## Heapsort Analysis

- The number of comparisons done by `fixHeap` on heap with  $k$  nodes is at most  $2 \lg(k)$ 
  - so the total for all deletions is at most
  - $2 \sum_{k=1}^{n-1} (\lg(k)) \in \theta(2n \lg(n))$
- Theorem: The number of comparisons of keys done by Heapsort in the worst case is  $2n \lg(n) + O(n)$ .
- Heapsort does  $\theta(n \lg(n))$  comparisons on average as well. (How do we know this?)

## Implementation issue: storing a tree in an array

- Array  $E$  with range from  $1, \dots, n$
- Suppose the index  $i$  of a node is given, then
  - left child has index  $2i$
  - right child has index  $2i + 1$
  - parent has index  $\text{floor}(i/2)$
- e.g.

9	5	7	1	4	3	6
---	---	---	---	---	---	---

Heap 1

50	24	30	20	21	18	3	12	5	6
----	----	----	----	----	----	---	----	---	---

Heap 2

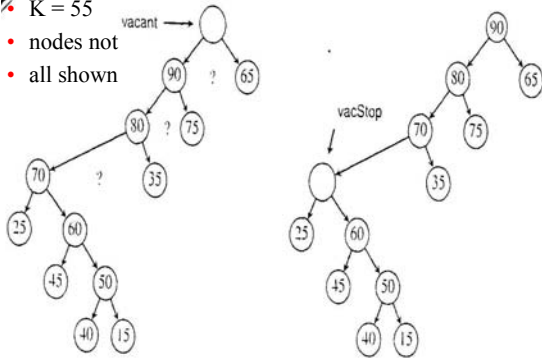
## Accelerated Heapsort

- Speed up Heapsort by about a factor of two.
- Normal `fixHeap` costs  $2h$  comparisons in the worst case. Can we do better?
- ...
- The solution is a surprising application of divide and conquer!
  - filter the vacant position halfway down the tree,  $h/2$
  - test whether  $K$  is bigger than the parent of vacant
  - yes: bubble the vacant back up to where  $K$  should be
  - no: repeat filter the vacant position another halfway down *recursively!*

## Accelerated Heapsort Strategy in Action

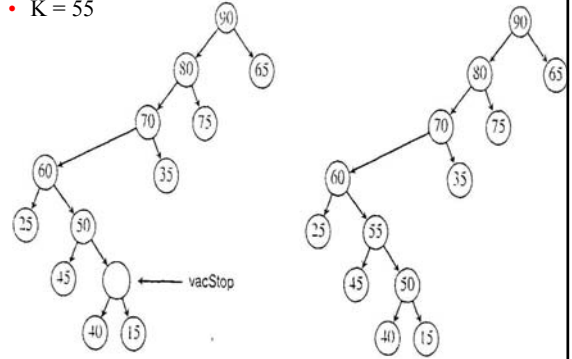
⚡  $K = 55$

- nodes not
- all shown



## Action continues

•  $K = 55$



## Accelerated Heapsort Algorithm

- void fixHeapFast(Element[] E, int n, Element K, int vacant, int h)
  - if ( $h \leq 1$ )
    - > Process heap of height 0 or 1
  - else
    - > int hStop =  $h/2$
    - > int vacStop = promote (E, hStop, vacant, h);
    - > // vacStop is new vacant location, at height hStop
    - > int vacParent =  $vacStop / 2$ ;
    - > if ( $E[vacParent].key \leq K.key$ )
      - $E[vacStop] = E[vacParent]$ ;
      - bubbleUpHeap (E, vacant, K, vacParent);
    - > else
      - fixHeapFast (E, n, K, vacStop, hStop);

## Algorithm: promote

- int promote (Element[] E, int hStop, int vacant, int h)
  - int vacStop;
  - if ( $h \leq hStop$ )
    - > vacStop = vacant;
  - else if ( $E[2*vacant].key \leq E[2*vacant+1].key$ )
    - >  $E[vacant] = E[2*vacant+1]$ ;
    - > vacStop = promote (E, hStop,  $2*vacant+1$ ,  $h-1$ );
  - else
    - >  $E[vacant] = E[2*vacant]$ ;
    - > vacStop = promote (E, hStop,  $2*vacant$ ,  $h-1$ );
  - return vacStop;

## Algorithm: bubbleUpHeap

- void bubbleUpHeap (Element[] E, int root, Element K, int vacant)
  - if (vacant == root)
    - >  $E[vacant] = K$ ;
  - else
    - > int parent =  $vacant / 2$ ;
    - > if ( $K.key \leq E[parent].key$ )
      - $E[vacant] = K$ ;
    - > else
      - $E[vacant] = E[parent]$ ;
      - bubbleUpHeap (E, root, K, parent);

## Analysis: fixHeapFast

- Essentially, there is one comparison each time vacant changes a level due to the action of either bubbleUpHeap or Promote. The total is  $h$ .
- Assume bubbleUpHeap is never call, so fixHeapFast reaches its base case. Then, it requires  $\lg(h)$  checks along the way to see whether it needs to reverse direction.
- Therefore, altogether fixHeapFast uses  $h + \lg(h)$  comparisons in the worst case.

## Accelerated Heapsort Analysis

- The number of comparisons done by fixHeapFast on heap with  $k$  nodes is at most  $\lg(k)$ 
  - so the total for all deletions is at most
  - $\sum_{k=1}^{n-1} (\lg(k)) \in \theta(n \lg(n))$
- Theorem: The number of comparisons of keys done by Accelerated Heapsort in the worst case is  $n \lg(n) + O(n)$ .

## Comparison of Four Sorting Algorithms

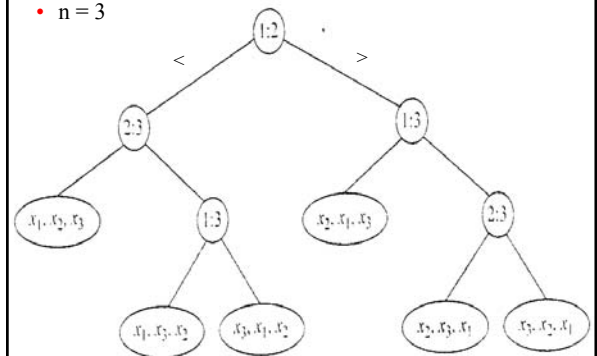
- | Algorithm | Worst case | Average            | Space Usage |
|-----------|------------|--------------------|-------------|
| Insertion | $n^2/2$    | $\theta(n^2)$      | in place    |
| Quicksort | $n^2/2$    | $\theta(n \log n)$ | $\log n$    |
| Mergesort | $n \lg n$  | $\theta(n \log n)$ | $n$         |
| Heapsort  | $2n \lg n$ | $\theta(n \log n)$ | in place    |
| Ac.Heaps. | $n \lg n$  | $\theta(n \log n)$ | in place    |
- Accelerated Heapsort currently is the method of choice.

## Lower Bounds for Sorting by Comparison of Keys

- The Best possible!
  - Lower Bound for Worst Case
  - Lower Bound for Average Behavior
- Use decision tree for analyzing the class of sorting algorithms (by comparison of keys)
  - Assuming the keys in the array to be sorted are distinct.
  - Each internal node associates with one comparison for keys  $x_i$  and  $x_j$ ; labeled  $i:j$
  - Each leaf nodes associates with one permutation (total  $n!$  permutations for problem size  $n$ )
  - The action of Sort on a particular input corresponds to following one path in its decision tree from the root to a leaf.

## Decision tree for sorting algorithms

- $n = 3$



## Lower Bound for Worst Case

- Lemma:
  - Let  $L$  be the number of leaves in a binary tree and let  $h$  be its height.
  - Then  $L \leq 2^h$ , and  $h \geq \lceil \lg L \rceil$
  - For a given  $n$ ,  $L = n!$ , the decision tree for any algorithm that sorts by comparison of keys has height as least  $\lceil \lg n! \rceil$ .
- Theorem:
  - Any algorithm to sort  $n$  items by comparisons of keys must do at least  $\lceil \lg n! \rceil$ ,
  - or approximately  $\lceil n \lg n - 1.443 n \rceil$ ,
  - key comparisons in the worst case.

## Lower Bound for Average Behavior

- Theorem:
  - The average number of comparisons done by an algorithm to sort  $n$  items by comparison of keys is at least  $\lg n!$
  - or approximately  $n \lg n - 1.443 n$
- The only difference from the worst-case lower bound is that there is no rounding up to an integer
  - the average needs not be an integer,
  - but the worst case must be.

## Improvement beyond lower bound?!

### Know more → Do better

- Up to now,
    - only one assumption was made about the keys: They are elements of linearly ordered set.
    - The basic operation of the algorithms is a comparison of two keys.
  - If we *know more* (or make more assumptions) about the keys,
    - we can consider algorithms that perform other operations on them.
- // Recall algorithms for searching from unordered data vs. searching from ordered data

## Using properties of the keys

- support the keys are names
- support the keys are all five-digit decimal integers
- support the keys are integer between 1 and m.
- For sorting each of these examples, the keys are
  - distributed into different piles as a result of examining individual letters or digits in a key or comparing keys to predetermined values
  - sort each pile individually
  - combine all sorted piles
- Algorithms that sort by such methods are not in the class of algorithms previously considered because
  - to use them we must know something about either the structure or the range of the keys.

## Radix Sort

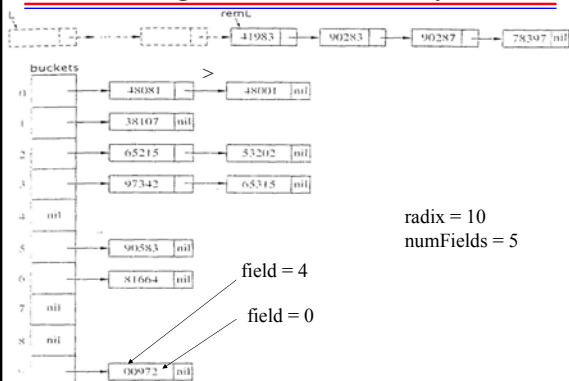
- Strategy: It is *startling* that
  - if the keys are distributed into piles (also called buckets) first according to their *least significant digits* (or bits, letters, or fields),
    - and the piles are combined in order
    - and the relative order of two keys placed in the same pile is not changed
  - then the problem of sorting the piles has been completely eliminated!

## Radix Sort e.g

### Start from least significant digit

Unsorted file	First		Second		Third		Fourth		Fifth		Sorted file
	bkt	Pass	bkt	Pass	bkt	Pass	bkt	Pass	bkt	Pass	
48081	1	48081	0	48001	0	48001	0	90283	0	00972	00972
97342		48001		53202		48081		90287			38107
90287	2	97342		38107	1	38107		90583	3	38107	41983
90583		53202	1	65215	2	53202		00972	4	41983	48001
53202		00972		65315		65215	1	81664		48081	48081
65215	3	90583				90283		41983		48081	53202
78397		41983	4	97342		90287	3	53202	5	53202	65215
48001		90283	6	81664	3	65315		65215	6	65215	65315
00972	4	81664	7	00972		97342	5	65315		65315	78397
65315	5	65315	8	48081		78397		65315	7	78397	81664
41983				41983	5	90583	7	97342	8	81664	81664
90283	7	90287		90283	6	81664		48001	9	90283	90283
81664		78397		90287				48081		90287	90287
38107		38107	9	00972				38107		90583	90583
38107				78397		41983		78397		97342	97342

## Radix Sort e.g. Data Structure, array of lists



## Radix Sort: Algorithm

- List radixSort (List L, int radix, int numFields)
  - List[] buckets = new List[radix];
  - int field; // filed number within the key
  - List newL;
  - newL = L;
  - For (field = 0; field < numFields; field++)
    - Initialize buckets array to empty lists.
    - distribute (newL, buckets, radix, field);
    - newL = combine (buckets, radix);
  - return newL;

## Radix Sort: distribute

---

- void distribute (List L, List[] buckets, int radix, int field)
  - //distribute keys into buckets
  - List remL;
  - remL = L;
  - while (remL != nil)
    - > Element K = first (remL);
    - > int b = maskShift (field, radix, K.key);
      - // maskShift(f, r, key) selects field f (counting from the right) of key,
      - // based on radix r. the result, b, is the range 0 ... radix - 1,
      - // and is the bucket number for K
    - > buckets[b] = cons(K, buckets[b]); // construct list
    - > remL = rest (remL);
  - return

## Radix Sort: Combine

---

- List combine (List[] buckets, int radix)
  - // Combine linked lists in all buckets into one list L
  - int b; // bucket number
  - List L, remBucket;
  - L = nil;
  - for (b = radix-1; b>=0; b--)
    - > remBucket = buckets[b];
    - > while (remBucket != nil)
      - key K = first (remBucket);
      - L = cons (K, L);
      - remBucket = rest (remBucket);
  - return L;

## Radix Sort: Analysis

---

- distribute does  $\theta(n)$  steps
- combine does  $\theta(n)$  steps
- if number of field is constant,
  - then the total number of steps done by radix sort is *linear in n*.
- radix sort use  $\theta(n)$  extra space for link fields, provided the radix is bounded by n.

## Exercise

---

- Do it or loss it!