Lecture Notes
Kenrick Mock
CS411

Non-Comparison Based Sorting

How fast can we sort?

Insertion-Sort $O(n^2)$

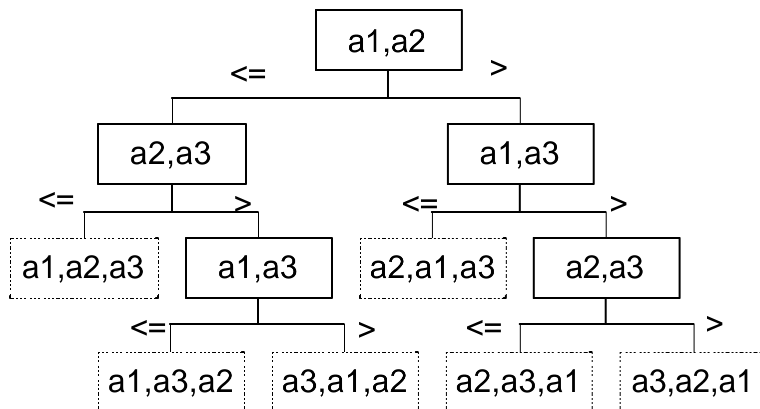
Merge-Sort, Quicksort (expected), Heapsort $:\Theta(n \lg n)$

Can we do faster? What is the theoretical best we can do?

So far we have done comparison sorts: A sort based only on comparisons between input elements. E1<E2, E1=E2, E1>E2. We will show that any comparison-based sort MUST make $\Omega(n \lg n)$ comparisons. This means that merge sort and heap sort are optimal. **This is important because it is not always possible that you can prove that your algorithm is the best one possible for a problem!**

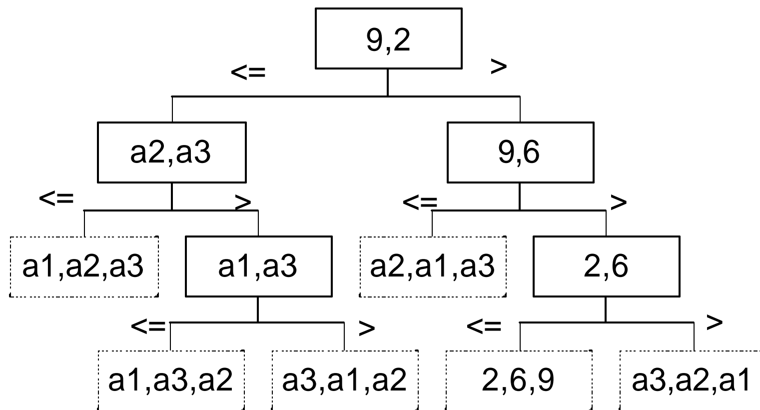
A decision tree is used to represent the comparisons of a sorting algorithm. Assume that all inputs are distinct. A decision tree compares all possible inputs to each other to determine the sequence of outputs.

Decision Tree for three elements a1,a2,a3 : If at the root, a1 ≤ a2 go left and compare a2 to a3, otherwise go right and compare a1 to a3. Each path represents a different ordering on a1,a2,a3.



This type of decision tree will have n! leaves – one for each permutation of the input. Any comparison-based sorting algorithm will have to go through the steps in the decision tree as a minimum (can do more comparisons if we want to, of course!)

Example of 9,2,6 :



The sorted elements are 2,6,9, in order of a2,a3,a1.

Decision trees can model comparison sorts. For any sorting algorithm:

1. One tree for each input length n
2. An algorithm "splits" at each decision/comparison unwinding the actual execution into a tree path
3. The tree is all possible execution traces

What is the height of the decision tree? This gives us the minimum number of comparisons necessary to sort the input.

For n inputs, the tree must have n! leaves. A binary tree of height h has no more than 2^h leaves:

$$n! \leq 2^h$$

Take log:

$$\lg(n!) \leq h$$

Stirling's approximation says that $n! > \sqrt{2n\pi}(n/e)^n > (n/e)^n$

So:
$$\lg(n/e)^n \leq h$$
$$n\lg(n/e) \leq h$$
$$n(\lg n - \lg e) \leq h$$
$$n\lg n - n\lg e \leq h$$

This means h= $\Omega(n \lg n)$ and we are DONE! We need to do at least nlgn comparisons to reach the bottom of the tree.

Does this mean that we can't do any better?? NO! (well, in some cases)
We can actually do some types of sorting in LINEAR TIME.

Counting Sort

This may work in O(n) time. How? Because it uses no comparisons! But we have to make assumptions about the size and nature of the input.

Input: A[1..n] where A[I]={1…k}
Output: B[1..n], sorted
Uses: C[1..k] auxiliary storage

Idea: Using random access array, count up number of times each input element appears and then collect them together.

Algorithm:

```
Count-Sort(A,n)
    for I← 1 to k do C[I] ← 0          ; Initialize to 0
    for j← 1 to n do C[A[j]] ++        ; Count
    j← 1
    for I← 1 to k do
        if (C[I]>0) then
            for z← 1 to C[I] do
                B[j]=I
                j++
```

Ex:
    A=[1 5 3 2 2 4 9]
    C=[0 0 0 0 0 0 0 0 0]
    C=[1 2 1 1 1 0 0 0 1]
    B=[1 2 2 3 4 5 9]

This works! How long does it take? O(n+k). If k=n, then this runs in O(n) time.
However, a bad example would be a input list like A[1,2,99999999].

One disadvantage of the current algorithm: it is not **stable**
An algorithm is stable if the occurrences of a value I appear in the same order in the output as they do in the input. That is, ties between two numbers are broken by the rule that whichever number appears first in the input array appears first in the output array.

Why do we want a stable algorithm? If the thing we are sorting is just a key of a record (perhaps a zip code, or a job indicating priority where we want the first one in to have precedence) then stability may be important.

Ex: A[3 5a 9 2 4 5b 6]
Sorts to A[2 3 4 5a 5b 6 9]
and not to A[2 3 4 5b 5a 6 9]

Can modify algorithm to make it stable:

```
Stable-Count-Sort(A,n)
    for I←1 to k do C[I] ←0              ; Initialize to 0
    for j←1 to n do C[A[j]] ++           ; Count
    for I←2 to k do
        C[I] ←C[I]+C[I-1]                ; Sum elements so far
                                         ; C[I] contains num elements <= I
    for j←n downto 1 do
        B[C[A[j]]] ←A[j]
        C[A[j]] ←C[A[j]]-1
```

Example: A=[1 5 3 2 2 4 9]
C=[0 0 0 0 0 0 0 0 0]
C=[1 2 1 1 1 0 0 0 1]
C=[1 3 4 5 6 6 6 6 7]
B=[. . . . . . 9]
C=[1 3 4 5 6 6 6 6 6]
B=[. . . . 4 . 9]
…
B=[1 2 2 3 4 5 9]

This version is stable, since we fetch from the original array.

Radix Sort

Works like the punch-card readers of the early 1900's. Only works on input items with digits! Idea somewhat counterintuitive: Sort on the least significant digit first.

```
Radix-Sort(A,d,n)            ; A is an n element array, each element d digits long
    for i←1 to d
        do   Use a stable sort to sort array A on digit i
```

Example:

A			
492	031	102	031
299	492	204	102

102		102		031		204
031	→	204	→	835	→	299
996		835		492		492
204		996		996		835
835		299		299		996

Sort must be stable so numbers chosen in the correct order! Assumes that lower order digits are already sorted to work.

If each digit is not large, counting sort is a good choice to use for the sort method.
If k is the maximum value of the digit, then counting sort takes $\Theta(k+n)$ time for one pass. We have to make d passes, so the total runtime is $\Theta(dk+dn)$.

If d is a constant and k is smaller than O(n), Radix-Sort runs in O(n) linear time!

Radix or Counting sorts are simple to code and the method of choice if the input is of the right form.

Bucket Sort

Similar to count sort, but uses a "bucket" to hold a range of inputs. Works for real numbers! Like the other sorts, bucket sort is fast because it assumes something about the input:

1. Input is randomly generated
2. Input elements randomly distributed over the interval [0..1]. In many cases we can divide by some "max" value to force the input key for comparison to be between 0 and 1. This assumption means that elements are generated with uniform probability over [0..1] or that each element has the same likelihood of being generated.

Idea:
1. Divide [0..1] into n equal sized parts or "buckets"
2. Put each of the n inputs into one of the buckets. Some buckets may be empty and some may have more than 1 element.
3. Sort each bucket.
4. To produce output, go through the buckets in order, listing the elements in each.

Linked Lists is a good mechanism for storing the buckets.

```
Bucket-Sort(A,n)
for i← 1 to n do
        Insert A[I] into list B[nA[I]]
for i← 0 to n-1 do
```

sort list B[I] with insertion sort
concatenate the lists B[0], B[1], … B[n-1] together in order

Buckets are automatically numbered in this case from 0..n-1

All the lines but line 5 take O(n) time in the worst case.
Line 5 is insertion sort which takes O(n^2) time but since the input is generated uniformly we dont expect any bucket to have many elements in it so Insertion-Sort should be called on very small lists.

Example:

A=[0.44 0.12 0.73 0.29 0.67 0.49]
Bucket I will get the values between I/n and (I+1)/n since buckets are numbered from 0 to n-1.

B
0..0.16 → 0.12
0.16..0.33 → 0.29
0.33..0.50 → 0.44 → 0.49
0.50..0.66 →
0.66..0.83 →0.73 →0.67
0.83..1 →

Sort the buckets with insertion sort and then combine buckets to get:

0.12 0.29 0.44 0.49 0.67 0.73

Informal Argument on the average time:

Since any element in A comes from [0..1] with an equal probability then the probability that an element e is in bucket B[I] is 1/n (each bucket covers 1/n of [0..1].

This means that the average number of elements that end up in bucket B[I] is 1. There is a little more to the analysis than this, but the basic idea is that the distribution of the input will cause the calls to Insertion-Sort to be on very short lists and so the other steps in the algorithm will use more time. The average running time of Bucket-Sort is then T(n)=O(n).

Postman Sort

There are many other sorting algorithms that have been proposed. Robert Ramey proposed the Postman Sort in the August 1992 issue of the C Programming journal. We will briefly discuss it here, as it will be the basis for a later exercise. The full article is available at http://www.rrsd.com/psort/cuj/cuj.htm . Although the article makes the sorting algorithm sound revolutionary, it is really just a variant on bucket sort.

To quote Ramey's article regarding a generalized distributed sorting algorithm:

> When a postal clerk receives a huge bag of letters he distributes them into other bags by state. Each bag gets sent to the indicated state. Upon arrival, another clerk distributes the letters in his bag into other bags by city. So the process continues until the bags are the size one man can carry and deliver. This is the basis for my sorting method which I call the postman's sort.
>
> Suppose we are given a large list of records to be ordered alphabetically on a particular field. Make one pass through the file. Each record read is added to one of 26 lists depending on the first letter in the field. The first list contains all the records with fields starting with the letter "A" while the last contains all the records with fields starting with the letter "Z". Now we have divided the problem down to 26 smaller subproblems. Now we address subproblem of sorting all the records in the sublist corresponding to key fields starting with the letter "A". If there are no records in the "A" sublist we can proceed to deal with the "B" sublist. If the "A" sublist contains only one record it can be written to output and we are done with that sublist. If the "A" sublist contains more that one record, it must be sorted then output. Only when the "A" list has been disposed of we can move on to each of the other sublists in sequence. The records will be written to the output in alphabetical order. When the "A" list contains more than one record it has to be sorted before it is output. What sorting algorithm should be used? Just like a real postman, we use the postman's sort. Of course we just apply the method to the second letter of the field. This is done to greater and greater depths until eventually all the words starting with "A" are written to the output. We can then proceed to deal with sublists "B" through "Z" in the same manner.

Example: Consider sorting "BOB", "BILL", BOY", "COW", "DOG"

How fast is it? (Exercise for the reader)

Binary Search Trees
Chapter 13

Skip section 13.4

Usually we think of binary trees as having two children at each node and the height h of the tree as being lgn. This is a specific case of a binary tree. We will look at more general trees. In fact, the height of a binary tree can be n if all children are to the left.

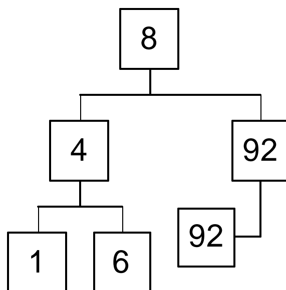
Basic Operations on a BST:

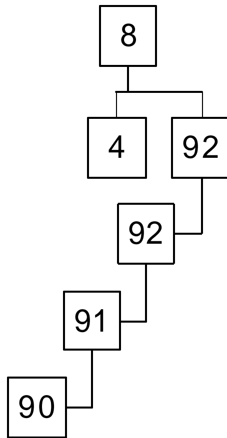
- Traversal
- Search
- Minimum
- Maximum
- Insert
- Delete
- Predecessor
- Successor

Claim: The basic operations on a binary tree take time proportional to the height of the tree.

A BST is a binary tree ordered such that the binary search property for a node I: All children in left subtree $\leq$ I $\leq$ all children in right subtree

Examples of valid binary trees:





Doesn't have to be balanced, just support the binary search property. If sorted, we could have a single tree of length n!

There are 3 ways to **traverse** a binary search tree:

1. Inorder tree walk : left child, root, right child (gives a sorted list)

```
Inorder(x)
        If X<>NIL
                InOrder(left[x])
                Print key(x)
                InOrder(right[x])
```

2. Preorder tree walk: root, left child, right child
3. Postorder tree walk: left child, right child, root

Each takes O(n) since every node in the tree is visited.

Search in a binary search tree takes O(h) time, where h is the height of the tree.
A search for x is done via a recursive algorithm that at node n looks down the left subtree of n if x<n and otherwise, if x>n, looks down the right subtree of n.

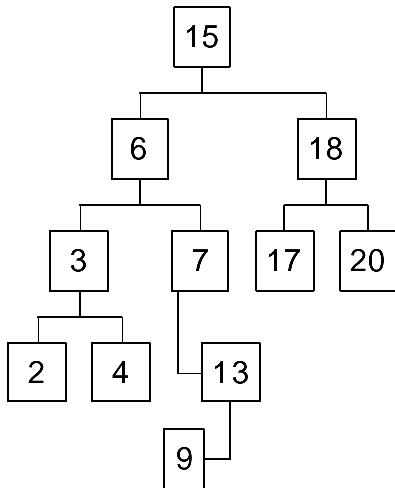
The binary-search tree property implies that no elements smaller than n are to the right of n, and no elements larger than n are to the left. The time to just travel down the root to the node which in the worst case will be a leaf, so O(h) time where h is the height of the tree.

```
        Tree-Search(x,k)
                if x=NIL or k=key[x] return x
                if k<key[x] then return(Tree-Search(left[x],k))
                            else return(Tree-Search(right[x],k))
```

Min can be found by always taking the left child until we reach a leaf. This takes O(h) time.
Max can be found by always taking the right child until we reach a leaf. This take O(h) time.

The **Successor** of an element X is the next largest element.
This can be found via 2 cases:

1. If x has a right subtree, then the successor of x is the leftmost node in the right subtree.
2. If x has no right subtree then the successor of x is the lowest ancestor of x whose left child is also an ancestor of x (go up parents until find a node whose left child is an ancestor of x)



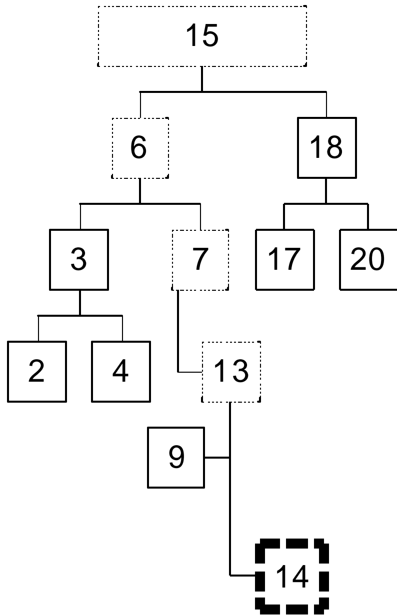
Succ of 15 is 17, Succ of 13 is 15, Succ of 9 is 13, Succ of 4 is 6
This works from the structure of the binary tree!

Finding **Predecessor** is the opposite of Successor. The time for both is O(h) since we only move up or down the tree. In fact, no comparisons are necessary!

Tree-Insertion: Start at the root and move down the tree according to the binary search property. When at a leaf, add a new leaf in the correct position.

High-level pseudocode:

> Use Search algorithm until reach NIL; a leaf that does not yet exist.
> Add new element here and create pointer to it.



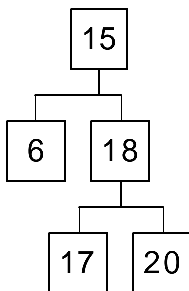
The dotted path indicates the path taken to insert the element "14" into the tree.

This operation takes O(h) time, as in search. What order of insertions ends up in a poorly balanced tree?

Tree-Deletion: To perform deletion we may need to rearrange the tree so that it is still a binary tree and maintains the binary search property.

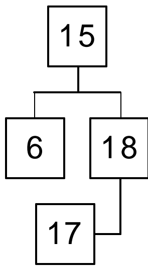
```
Tree-Delete(T,x)
    Search until find node with x
    if x has no children                        ; case 1
        then remove x
    if x has one child                          ; case 2
        then make Parent[x] point to child of x
    if x has two children                       ; case 3
        then swap x with its successor
        perform case 1 or case 2 to delete it
```

Case 1:

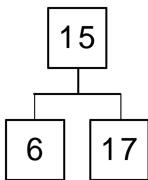


To delete the node with 20, just remove it.

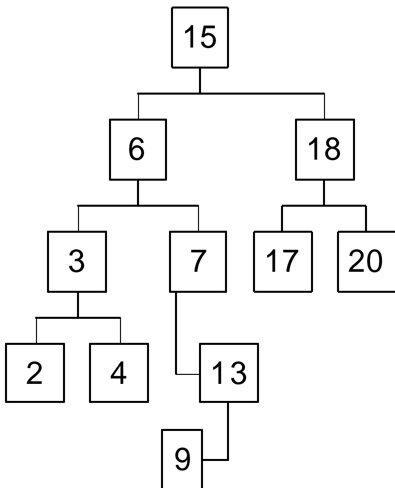
Case 2:



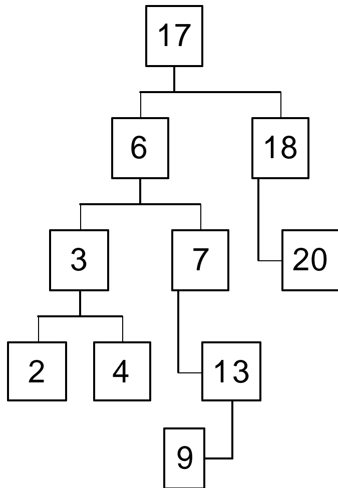
Removing node 18 becomes:



Case 3:



Deleting node 15 becomes



How do we know that we can do case 1 or case 2 with the successor's old node? Because when a node has two children, the successor is the leftmost node of the right subtree. The leftmost node is guaranteed to have no left child, so it will fall in case 1 or 2.

Time O(h) to perform deletion since just a Find-Successor operation and swap.
This delete operation may unbalance a balanced tree!

The moral of binary trees: Creating and maintaining a binary tree is O(h) time which will be fast if the tree is evenly balanced (lg n). If building the tree out of input data that is random, on average you will get balanced trees. It is possible to ensure the tree will be balanced on all operations without incurring extra cost (Ch 14, red-black trees. Still O(lgn) for all tree operations). We won't cover these, but you should be aware that they exist.