Algorithms

Administrative

- Reminder: homework 3 due today
- Reminder: Exam 1 Wednesday, Feb 13
  - 1 8.5x11 crib sheet allowed
    - Both sides, mechanical reproduction okay
    - You will turn it in with the exam

Review Of Topics

- Asymptotic notation
- Solving recurrences
- Sorting algorithms
  - Insertion sort
  - Merge sort
  - Heap sort
  - Quick sort
  - Counting sort
  - Radix sort
- Medians/order statistics
  - Randomized algorithm
  - Worst-case algorithm
- Structures for dynamic sets
  - Priority queues
  - BST basics

Review: Induction

- Suppose
  - S(k) is true for fixed constant k
    - Often k = 0
  - S(n) → S(n+1) for all n >= k
- Then S(n) is true for all n >= k

Proof By Induction

- Claim:S(n) is true for all n >= k
- Basis:
  - Show formula is true when n = k
- Inductive hypothesis:
  - Assume formula is true for an arbitrary n
- Step:
  - Show that formula is then true for n+1

Induction Example: Gaussian Closed Form

- Prove $1 + 2 + 3 + \ldots + n = n(n+1) / 2$
  - Basis:
    - If $n = 0$, then $0 = 0(0+1) / 2$
  - Inductive hypothesis:
    - Assume $1 + 2 + 3 + \ldots + n = n(n+1) / 2$
  - Step (show true for $n+1$):

    $1 + 2 + \ldots + n + n+1 = (1 + 2 + \ldots + n) + (n+1)$

    $= n(n+1)/2 + n+1 = [n(n+1) + 2(n+1)]/2$

    $= (n+1)(n+2)/2 = (n+1)(n+1 + 1) / 2$

Induction Example: Geometric Closed Form

- Prove $a^0 + a^1 + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$ for all $a \neq 1$
  - Basis: show that $a^0 = (a^{0+1} - 1)/(a - 1)$

    $a^0 = 1 = (a^1 - 1)/(a - 1)$
  - Inductive hypothesis:
    - Assume $a^0 + a^1 + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$
  - Step (show true for n+1):

    $a^0 + a^1 + \ldots + a^{n+1} = a^0 + a^1 + \ldots + a^n + a^{n+1}$

    $= (a^{n+1} - 1)/(a - 1) + a^{n+1} = (a^{n+1+1} - 1)/(a - 1)$

Review: Asymptotic Performance

- *Asymptotic performance*: How does algorithm behave as the problem size gets very large?
    - Running time
    - Memory/storage requirements
  - Use the RAM model:
    - All memory equally expensive to access
    - No concurrent operations
    - All reasonable instructions take unit time
        - Except, of course, function calls
    - Constant word size

Review: Running Time

- Number of primitive steps that are executed
  - Except for time of executing a function call most statements roughly require the same amount of time
  - We can be more exact if need be
- Worst case vs. average case

Review: Asymptotic Notation

- Upper Bound Notation:
  - f(n) is O(g(n)) if there exist positive constants c and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$
  - Formally, $O(g(n)) = \{ f(n)\text{: } \exists$ positive constants c and n_0 such that $f(n) \leq c \cdot g(n)\ \forall\ n \geq n_0$
- Big O fact:
  - A polynomial of degree k is $O(n^k)$

Review: Asymptotic Notation

- Asymptotic lower bound:
  - f(n) is $\Omega(g(n))$ if ∃ positive constants c and n_0 such that $0 \le c \cdot g(n) \le f(n) \;\; \forall\, n \ge n_0$
- Asymptotic tight bound:
  - f(n) is $\Theta(g(n))$ if ∃ positive constants c_1, c_2, and n_0 such that $c_1\, g(n) \le f(n) \le c_2\, g(n) \; \forall\, n \ge n_0$
  - $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ AND $f(n) = \Omega(g(n))$

Review:
Other Asymptotic Notations

- A function f(n) is o(g(n)) if ∃ positive constants c and n_0 such that

  $$f(n) < c\, g(n) \;\forall\; n \geq n_0$$

- A function f(n) is ω(g(n)) if ∃ positive constants c and n_0 such that

  $$c\, g(n) < f(n) \;\forall\; n \geq n_0$$

- Intuitively,
  - o() is like <
  - ω() is like >
  - Θ() is like =
  - O() is like ≤
  - Ω() is like ≥

Review: Merge Sort

```
MergeSort(A, left, right) {
   if (left < right) {
        mid = floor((left + right) / 2);
        MergeSort(A, left, mid);
        MergeSort(A, mid+1, right);
        Merge(A, left, mid, right);
   }
}
// Merge() takes two sorted subarrays of A and
// merges them into a single sorted subarray of A.
// Code for this is in the book.  It requires O(n)
// time, and *does* require allocating O(n) space
```

Review: Analysis of Merge Sort

Statement Effort

```
MergeSort(A, left, right) {                          T(n)
    if (left < right) {                              Θ(1)
        mid = floor((left + right) / 2);                 Θ(1)
        MergeSort(A, left, mid);                         T(n/2)
        MergeSort(A, mid+1, right);                      T(n/2)
        Merge(A, left, mid, right);                      Θ(n)
    }
}
```

- So $T(n) =$ $\Theta(1)$ when $n = 1$, and

  $2T(n/2) + \Theta(n)$ when $n > 1$

- This expression is a *recurrence*

Review: Solving Recurrences

- Substitution method
- Iteration method
- Master method

Review: Solving Recurrences

- The substitution method
  - A.k.a. the “making a good guess method”
  - Guess the form of the answer, then use induction to find the constants and show that solution works
  - Example: merge sort
    - $T(n) = 2T(n/2) + cn$
    - We guess that the answer is $O(n \lg n)$
    - Prove it by induction
  - Can similarly show $T(n) = \Omega(n \lg n)$, thus $\Theta(n \lg n)$

Review: Solving Recurrences

- The "iteration method"
  - Expand the recurrence
  - Work some algebra to express as a summation
  - Evaluate the summation
- We showed several examples including complex ones:

$$\boxed{T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}}$$

Review: The Master Theorem

- Given: a *divide and conquer* algorithm
  - An algorithm that divides the problem of size n into a subproblems, each of size n/b
  - Let the cost of each stage (i.e., the work to divide the problem + combine solved subproblems) be described by the function $f(\mathrm{n})$
- Then, the Master Theorem gives us a cookbook for the algorithm's running time:

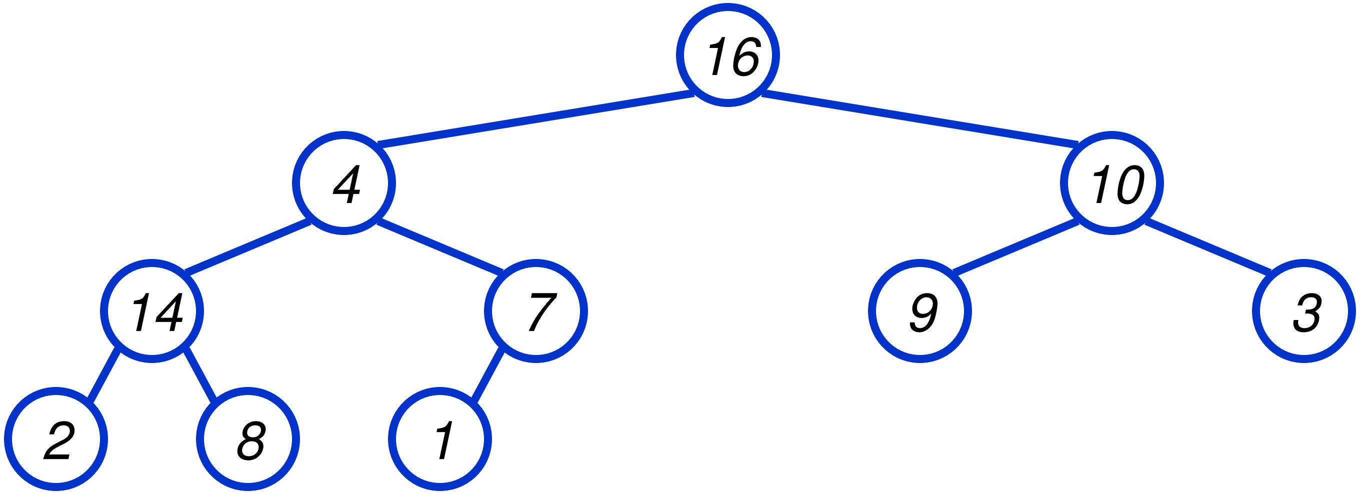
Review: The Master Theorem

- if T(n) = aT(n/b) + f(n) then

$$T(n) = \left\{ \begin{array}{ll} \Theta\left(n^{\log_b a}\right) & f(n) = O\left(n^{\log_b a - \varepsilon}\right) \\ \Theta\left(n^{\log_b a} \log n\right) & f(n) = \Theta\left(n^{\log_b a}\right) \\ \Theta\left(f(n)\right) & f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right) \text{AND} \\ & af(n/b) < cf(n) \text{ for large } n \end{array} \right\} \begin{array}{l} \varepsilon > 0 \\ c < 1 \end{array}$$

Review: Heaps

- A *heap* is a "complete" binary tree, usually represented as an array:



A =

16	14	10	8	7	9	3	2	4	1

Review: Heaps

- To represent a heap as an array:

```
Parent(i) { return ⌊i/2⌋; }
Left(i) { return 2*i; }
right(i) { return 2*i + 1; }
```

Review: The Heap Property

- Heaps also satisfy the *heap property*:

  A[*Parent*(i)] $\geq$ A[i]   for all nodes $i > 1$

  - In other words, the value of a node is at most the value of its parent
  - The largest value is thus stored at the root (A[1])
- Because the heap is a binary tree, the height of any node is at most $\Theta(\lg n)$

Review: Heapify()

- **`Heapify()`**: maintain the heap property
  - Given: a node *i* in the heap with children *l* and *r*
  - Given: two subtrees rooted at *l* and *r*, assumed to be heaps
  - Action: let the value of the parent node "float down" so subtree at *i* satisfies the heap property
    - If A[i] < A[l] or A[i] < A[r], swap A[i] with the largest of A[l] and A[r]
    - Recurse on that subtree
  - Running time: $O(h)$, h = height of heap = $O(\lg n)$

Review: BuildHeap()

- We can build a heap in a bottom-up manner by running **`Heapify()`** on successive subarrays
  - Fact: for array of length *n*, all elements in range A[$\lfloor n/2 \rfloor$ + 1 .. n] are heaps (*Why?*)
  - So:
    - Walk backwards through the array from n/2 to 1, calling **`Heapify()`** on each node.
    - Order of processing guarantees that the children of node *i* are heaps when *i* is processed

Review: BuildHeap()

```
// given an unsorted array A, make A a heap
BuildHeap(A)
{
  heap_size(A) = length(A);
  for (i = ⌊length[A]/2⌋ downto 1)
      Heapify(A, i);
}
```

Review: Priority Queues

- Heapsort is a nice algorithm, but in practice Quicksort (coming up) usually wins
- But the heap data structure is incredibly useful for implementing *priority queues*
  - A data structure for maintaining a set *S* of elements, each with an associated value or *key*
  - Supports the operations **`Insert()`**, **`Maximum()`**, and **`ExtractMax()`**
  - ***What might a priority queue be useful for?***

Review: Priority Queue Operations

- **Insert(S, x)** inserts the element x into set S
- **Maximum(S)** returns the element of S with the maximum key
- **ExtractMax(S)** removes and returns the element of S with the maximum key

Review:
Implementing Priority Queues

```
HeapInsert(A, key)     // what's running time?
{
    heap_size[A] ++;
    i = heap_size[A];
    while (i > 1  AND  A[Parent(i)] < key)
    {
        A[i] = A[Parent(i)];
        i = Parent(i);
    }
    A[i] = key;
}
```

Review:
Implementing Priority Queues

```
HeapExtractMax(A)
{
    if (heap_size[A] < 1) { error; }
    max = A[1];
    A[1] = A[heap_size[A]]
    heap_size[A] --;
    Heapify(A, 1);
    return max;
}
```

Review: Quicksort

- Another divide-and-conquer algorithm
  - The array A[p..r] is *partitioned* into two non-empty subarrays A[p..q] and A[q+1..r]
    - Invariant: All elements in A[p..q] are less than all elements in A[q+1..r]
  - The subarrays are recursively sorted by calls to quicksort
  - Unlike merge sort, no combining step: two subarrays form an already-sorted array

Review: Quicksort Code

```
Quicksort(A, p, r)
{
    if (p < r)
    {
        q = Partition(A, p, r);
        Quicksort(A, p, q);
        Quicksort(A, q+1, r);
    }
}
```

Review: Partition

- Clearly, all the action takes place in the **`partition()`** function
  - Rearranges the subarray in place
  - End result:
    - Two subarrays
    - All values in first subarray $\leq$ all values in second
  - Returns the index of the "pivot" element separating the two subarrays

Review: Partition In Words

- Partition(A, p, r):
    - Select an element to act as the "pivot" (*which?*)
    - Grow two regions, A[p..i] and A[j..r]
        - All elements in A[p..i] <= pivot
        - All elements in A[j..r] >= pivot
    - Increment i until A[i] >= pivot
    - Decrement j until A[j] <= pivot
    - Swap A[i] and A[j]
    - Repeat until i >= j
    - Return j

Note: slightly different from old book's `partition()`*, very different from new book*

Review: Analyzing Quicksort

- *What will be the worst case for the algorithm?*
  - Partition is always unbalanced
- *What will be the best case for the algorithm?*
  - Partition is balanced
- *Which is more likely?*
  - The latter, by far, except...
- *Will any particular input elicit the worst case?*
  - Yes: Already-sorted input

Review: Analyzing Quicksort

- In the worst case:

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

  $T(n) = T(n - 1) + \Theta(n)$

- Works out to

  $T(n) = \Theta(n^2)$

Review: Analyzing Quicksort

- In the best case:

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

- What does this work out to?

  $T(n) = \Theta(n \lg n)$

Review: Analyzing Quicksort (Average Case)

- Intuitively, the O(n) cost of a bad split (or 2 or 3 bad splits) can be absorbed into the O(n) cost of each good split
- Thus running time of alternating bad and good splits is still O(n lg n), with slightly higher constants
- We can be more rigorous…

Analyzing Quicksort: Average Case

- So partition generates splits
  (0:n-1, 1:n-2, 2:n-3, … , n-2:1, n-1:0)
  each with probability 1/n
- If T(n) is the expected running time,

$$T(n)=\frac{1}{n}\sum_{k=0}^{n-1}\left[T(k)+T(n-1-k)\right]+\Theta(n)$$

- *What is each term under the summation for?*
- *What is the $\Theta(n)$ term for?*

Analyzing Quicksort: Average Case

- So partition generates splits
  (0:n-1, 1:n-2, 2:n-3, … , n-2:1, n-1:0)
  each with probability 1/n
- If T(n) is the expected running time,

$$T(n) = \frac{1}{n}\sum_{k=0}^{n-1}\left[T(k) + T(n-1-k)\right] + \Theta(n)$$

  - *What are terms under the summation for? the $\Theta(n)$?*
- Massive proof that you should look over

Sorting Summary

- Insertion sort:
  - Easy to code
  - Fast on small inputs (less than ~50 elements)
  - Fast on nearly-sorted inputs
  - $O(n^2)$ worst case
  - $O(n^2)$ average (equally-likely inputs) case
  - $O(n^2)$ reverse-sorted case

Sorting Summary

- Merge sort:
  - Divide-and-conquer:
    - Split array in half
    - Recursively sort subarrays
    - Linear-time merge step
  - O(n lg n) worst case
  - Doesn’t sort in place

Sorting Summary

- Heap sort:
  - Uses the very useful heap data structure
    - Complete binary tree
    - Heap property: parent key > children's keys
  - O(n lg n) worst case
  - Sorts in place
  - Fair amount of shuffling memory around

Sorting Summary

- Quick sort:
  - Divide-and-conquer:
    - Partition array into two subarrays, recursively sort
    - All of first subarray < all of second subarray
    - No merge step needed!
  - O(n lg n) average case
  - Fast in practice
  - $O(n^2)$ worst case
    - Naïve implementation: worst case on sorted input
    - Address this with randomized quicksort

Review: Comparison Sorts

- Comparison sorts: O(n lg n) at best
  - Model sort with decision tree
  - Path down tree = execution trace of algorithm
  - Leaves of tree = possible permutations of input
  - Tree must have n! leaves, so O(n lg n) height

Review: Counting Sort

- Counting sort:
  - Assumption: input is in the range 1..k
  - Basic idea:
    - Count number of elements $k \leq$ each element i
    - Use that number to place i in position k of sorted array
  - No comparisons! Runs in time O(n + k)
  - Stable sort
  - Does not sort in place:
    - O(n) array to hold sorted output
    - O(k) array for scratch storage

Review: Counting Sort

```
CountingSort(A, B, k)
    for i=1 to k
        C[i]= 0;
    for j=1 to n
        C[A[j]] += 1;
    for i=2 to k
        C[i] = C[i] + C[i-1];
    for j=n downto 1
        B[C[A[j]]] = A[j];
        C[A[j]] -= 1;
```

Review: Radix Sort

- Radix sort:
  - Assumption: input has *d* digits ranging from 0 to *k*
  - Basic idea:
    - Sort elements by digit starting with *least* significant
    - Use a stable sort (like counting sort) for each stage
  - Each pass over *n* numbers with *d* digits takes time O(*n*+*k*), so total time O(*dn*+*dk*)
    - When *d* is constant and *k*=O(*n*), takes O(*n*) time
  - Fast! Stable! Simple!
  - Doesn't sort in place

Review: Bucket Sort

- Bucket sort
  - Assumption: input is *n* reals from [0, 1)
  - Basic idea:
    - Create *n* linked lists (*buckets*) to divide interval [0,1) into subintervals of size 1/*n*
    - Add each input element to appropriate bucket and sort buckets with insertion sort
  - Uniform input distribution → O(1) bucket size
    - Therefore the expected total time is O(n)
  - These ideas will return when we study *hash tables*

Review: Order Statistics

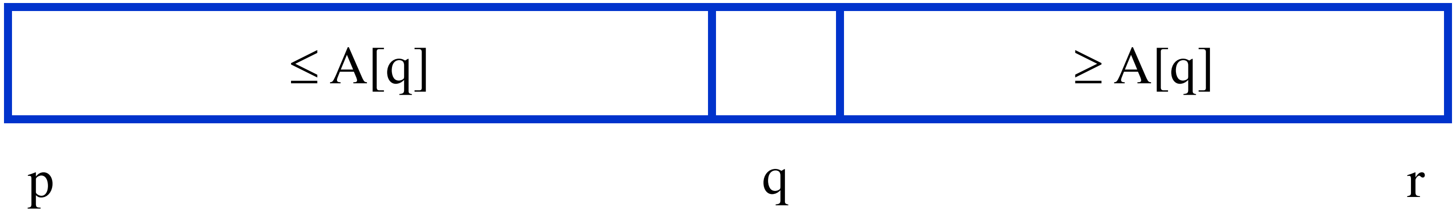
- The ith *order statistic* in a set of n elements is the ith smallest element
- The *minimum* is thus the 1st order statistic
- The *maximum* is (duh) the nth order statistic
- The *median* is the $n/2$ order statistic
  - If n is even, there are 2 medians
- Could calculate order statistics by sorting
  - Time: O(n lg n) w/ comparison sort
  - We can do better

Review: The Selection Problem

- The *selection problem*: find the *i*th smallest element of a set
- Two algorithms:
  - A practical randomized algorithm with O(n) expected running time
  - A cool algorithm of theoretical interest only with O(n) worst-case running time

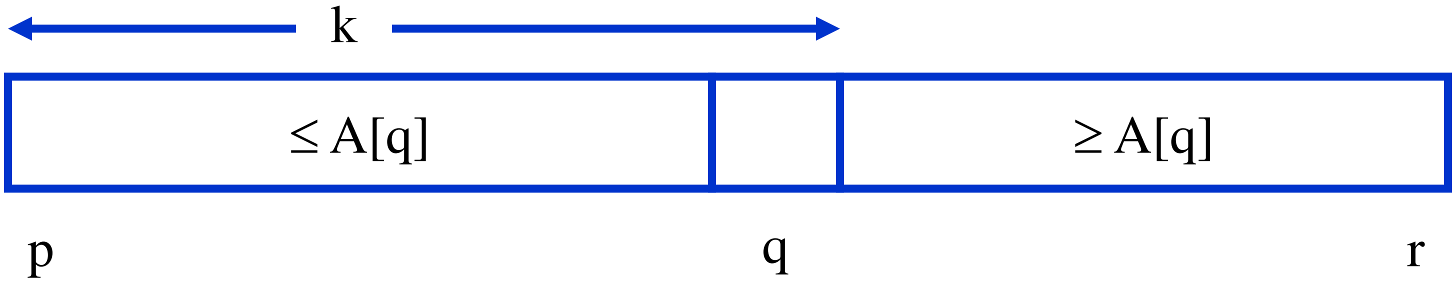
Review: Randomized Selection

- Key idea: use partition() from quicksort
  - But, only need to examine one subarray
  - This savings shows up in running time: O(n)



Review: Randomized Selection

```
RandomizedSelect(A, p, r, i)
    if (p == r) then return A[p];
    q = RandomizedPartition(A, p, r)
    k = q - p + 1;
    if (i == k) then return A[q];    // not in book
    if (i < k) then
        return RandomizedSelect(A, p, q-1, i);
    else
        return RandomizedSelect(A, q+1, r, i-k);
```



Review: Randomized Selection

- Average case
  - For upper bound, assume *i*th element always falls in larger side of partition:

$$T(n) \le \frac{1}{n}\sum_{k=0}^{n-1} T(\max(k, n-k-1)) + \Theta(n)$$

$$\le \frac{2}{n}\sum_{k=n/2}^{n-1} T(k) + \Theta(n)$$

  - We then showed that T(n) = O(n) by substitution

Review:
Worst-Case Linear-Time Selection

- Randomized algorithm works well in practice
- What follows is a worst-case linear time algorithm, really of theoretical interest only
- Basic idea:
  - Generate a good partitioning element
  - Call this element x

Review: Worst-Case Linear-Time Selection

- The algorithm in words:
  1. Divide *n* elements into groups of 5
  2. Find median of each group (*How? How long?*)
  3. Use Select() recursively to find median *x* of the $\lfloor n/5 \rfloor$ medians
  4. Partition the *n* elements around *x*. Let *k* = rank(*x*)
  5. **if** (i == k) **then** return x

     if (i < k) **then** use Select() recursively to find *i*th smallest element in first partition

     else (i > k) use Select() recursively to find (*i*-*k*)th smallest element in last partition

Review: Worst-Case Linear-Time Selection

- (Sketch situation on the board)
- *How many of the 5-element medians are $\leq x$?*
    - At least 1/2 of the medians = $\lfloor \lfloor n/5 \rfloor / 2 \rfloor = \lfloor n/10 \rfloor$
- *How many elements are $\leq x$?*
    - At least $3 \lfloor n/10 \rfloor$ elements
- For large n, $3 \lfloor n/10 \rfloor \geq n/4$ *(How large?)*
- So at least $n/4$ elements $\leq x$
- Similarly: at least $n/4$ elements $\geq x$

Review: Worst-Case Linear-Time Selection

- Thus after partitioning around *x*, step 5 will call Select() on at most 3*n*/4 elements
- The recurrence is therefore:

$$
\begin{aligned}
T(n) &\le T(\lfloor n/5 \rfloor) + T(3n/4) + \Theta(n) \\
&\le T(n/5) + T(3n/4) + \Theta(n) && \lfloor n/5 \rfloor \le n/5 \\
&\le cn/5 + 3cn/4 + \Theta(n) && \text{Substitute } T(n) = cn \\
&= 19cn/20 + \Theta(n) && \text{Combine fractions} \\
&= cn - (cn/20 - \Theta(n)) && \text{Express in desired form} \\
&\le cn \quad \text{if } c \text{ is big enough} && \text{What we set out to prove}
\end{aligned}
$$

Review: Binary Search Trees

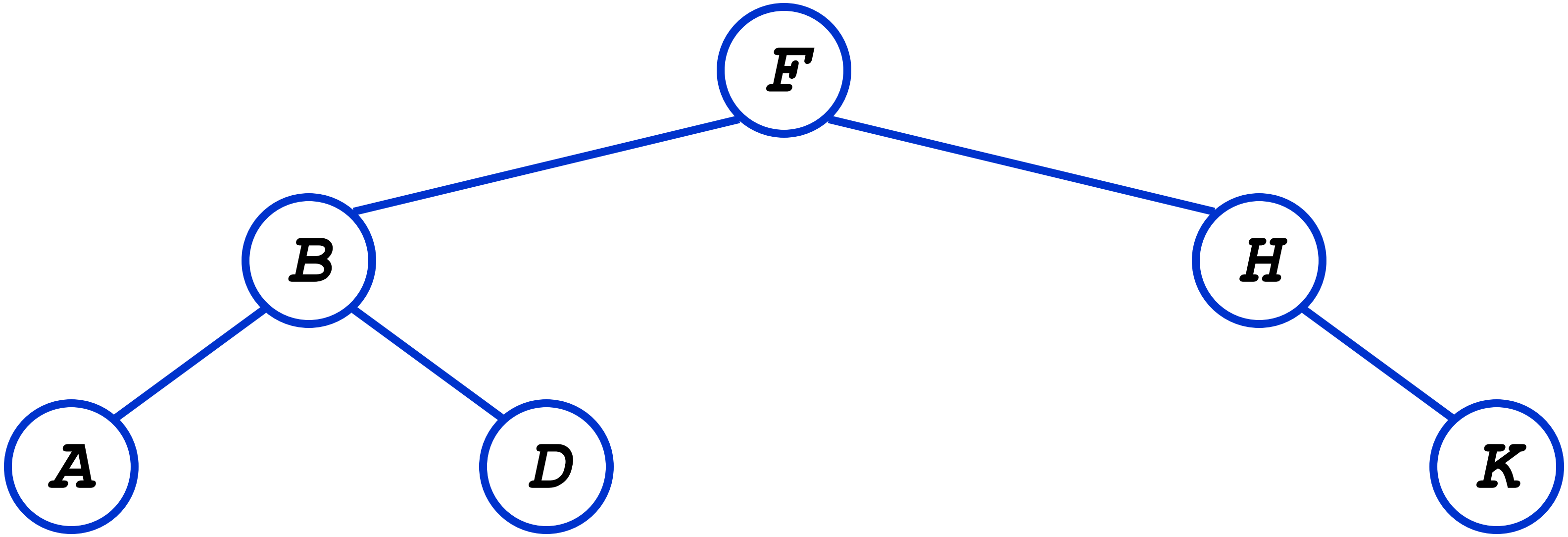
- *Binary Search Trees* (BSTs) are an important data structure for dynamic sets
- In addition to satellite data, elements have:
  - *key*: an identifying field inducing a total ordering
  - *left*: pointer to a left child (may be NULL)
  - *right*: pointer to a right child (may be NULL)
  - *p*: pointer to a parent node (NULL for root)

Review: Binary Search Trees

- BST property:

  key[left(x)] ≤ key[x] ≤ key[right(x)]

- Example:



Review: Inorder Tree Walk

- An *inorder walk* prints the set in sorted order:

```
TreeWalk(x)
    TreeWalk(left[x]);
    print(x);
    TreeWalk(right[x]);
```

  - Easy to show by induction on the BST property
  - *Preorder tree walk*: print root, then left, then right
  - *Postorder tree walk*: print left, then right, then root