Algorithms

Medians and Order Statistics

Structures for Dynamic Sets

Homework 3

- On the web shortly…
  - Due Wednesday at the beginning of class (test)

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

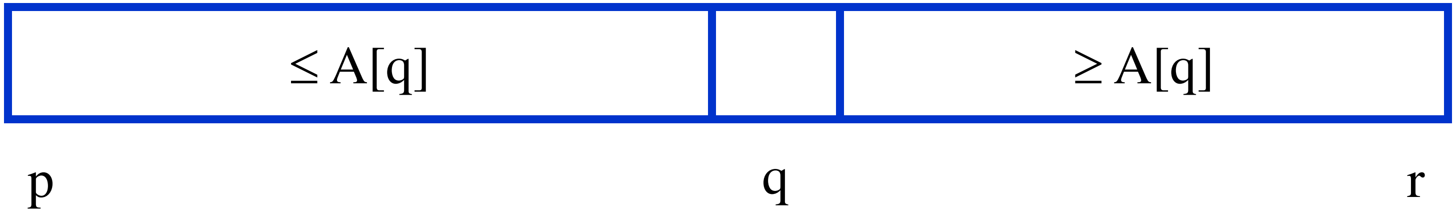
- The ith *order statistic* in a set of n elements is the ith smallest element
- The *minimum* is thus the 1st order statistic
- The *maximum* is (duh) the nth order statistic
- The *median* is the $n/2$ order statistic
  - If n is even, there are 2 medians
- Could calculate order statistics by sorting
  - 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)



$\leq A[q]$

$\geq A[q]$

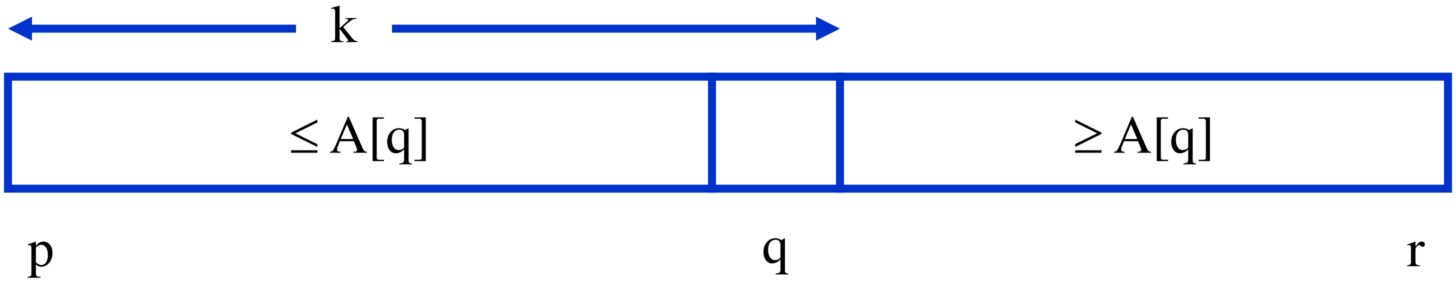
p

q

r

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

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

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

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 $\geq x$

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:

$$T(n) \le T(\lfloor n/5 \rfloor) + T(3n/4) + \Theta(n)$$

$$\le T(n/5) + T(3n/4) + \Theta(n)$$

$\lfloor n/5 \rfloor \le n/5$

$$\le cn/5 + 3cn/4 + \Theta(n)$$

Substitute T(n) = cn

$$= 19cn/20 + \Theta(n)$$

Combine fractions

$$= cn - (cn/20 - \Theta(n))$$

Express in desired form

$$\le cn \quad \text{if } c \text{ is big enough}$$

What we set out to prove

Worst-Case Linear-Time Selection

- Intuitively:
  - Work at each level is a constant fraction (19/20) smaller
    - Geometric progression!
  - Thus the O(n) work at the root dominates

Linear-Time Median Selection

- Given a “black box” O(n) median algorithm, what can we do?
  - *i*th order statistic:
    - Find median *x*
    - Partition input around *x*
    - if ($i \leq (n+1)/2$) recursively find *i*th element of first half
    - else find ($i - (n+1)/2$)th element in second half
    - $T(n) = T(n/2) + O(n) = O(n)$
  - *Can you think of an application to sorting?*

Linear-Time Median Selection

- Worst-case O(n lg n) quicksort
  - Find median *x* and partition around it
  - Recursively quicksort two halves
  - T(n) = 2T(n/2) + O(n) = O(n lg n)

Structures…

- Done with sorting and order statistics for now
- Ahead of schedule, so…
- Next part of class will focus on *data structures*
- We will get a couple in before the first exam
  - Yes, these will be on this exam

Dynamic Sets

- Next few lectures will focus on data structures rather than straight algorithms
- In particular, structures for *dynamic sets*
  - Elements have a *key* and *satellite data*
  - Dynamic sets support *queries* such as:
    - ***Search(S, k), Minimum(S), Maximum(S), Successor(S, x), Predecessor(S, x)***
  - They may also support *modifying operations* like:
    - ***Insert(S, x), Delete(S, x)***

Binary Search Trees

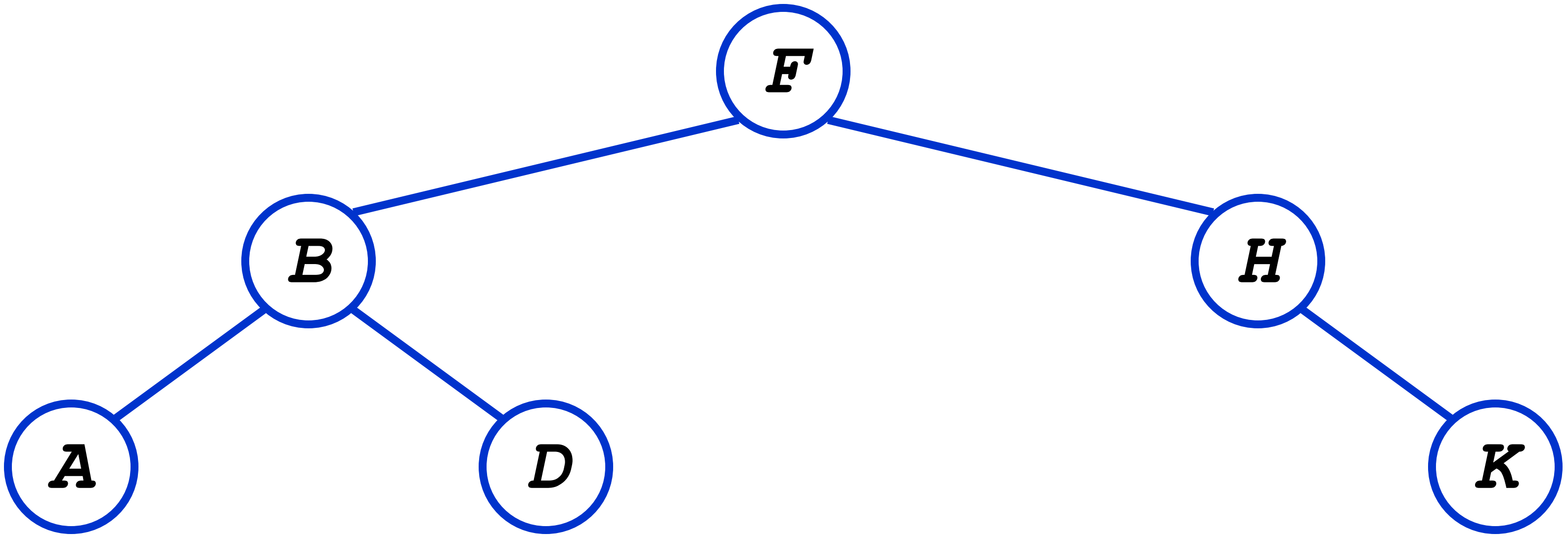
- *Binary Search Trees* (BSTs) are an important data structure for dynamic sets
- In addition to satellite data, eleements have:
  - *key*: an identifying field inducing a total ordering
  - *left*: pointer to a left child (may be NULL)
  - *right*: pointer to a right child (may be NULL)
  - *p*: pointer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

Binary Search Trees

- BST property:

  $$key[left(x)] \le key[x] \le key[right(x)]$$

- Example:



Inorder Tree Walk

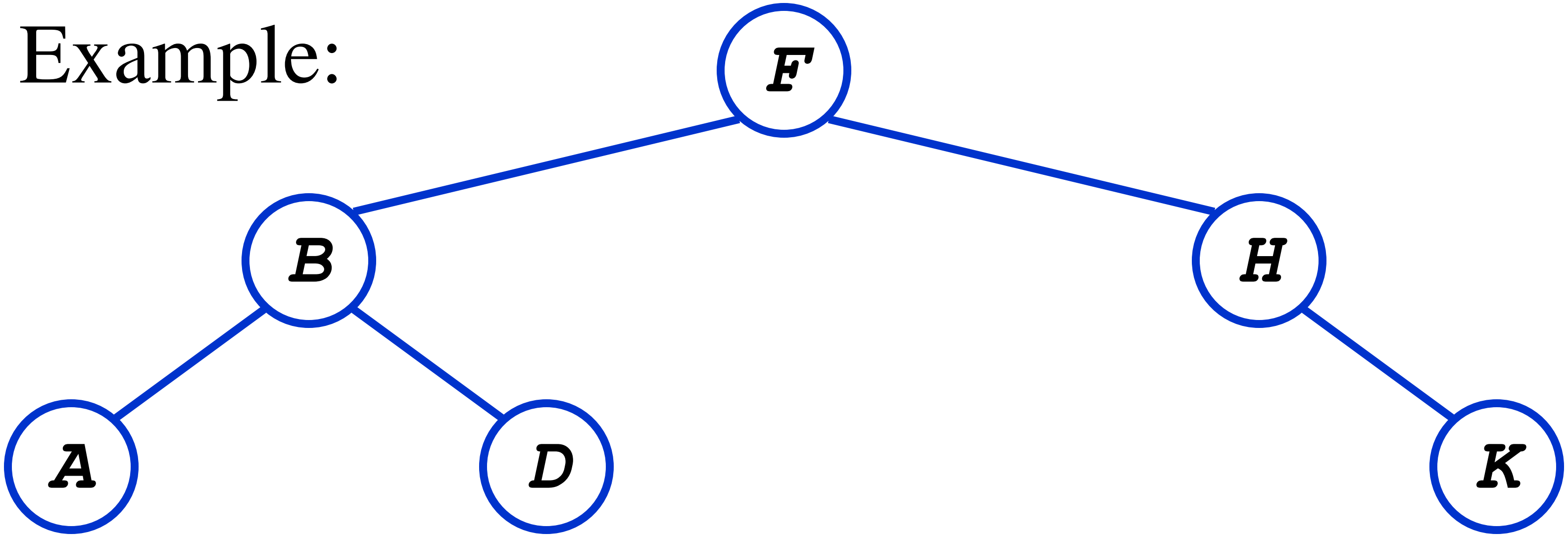
- *What does the following code do?*

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

- A: prints elements in sorted (increasing) order
- This is called an *inorder tree walk*
    - *Preorder tree walk*: print root, then left, then right
    - *Postorder tree walk*: print left, then right, then root

Inorder Tree Walk

- Example:



- *How long will a tree walk take?*
- *Prove that inorder walk prints in monotonically increasing order*

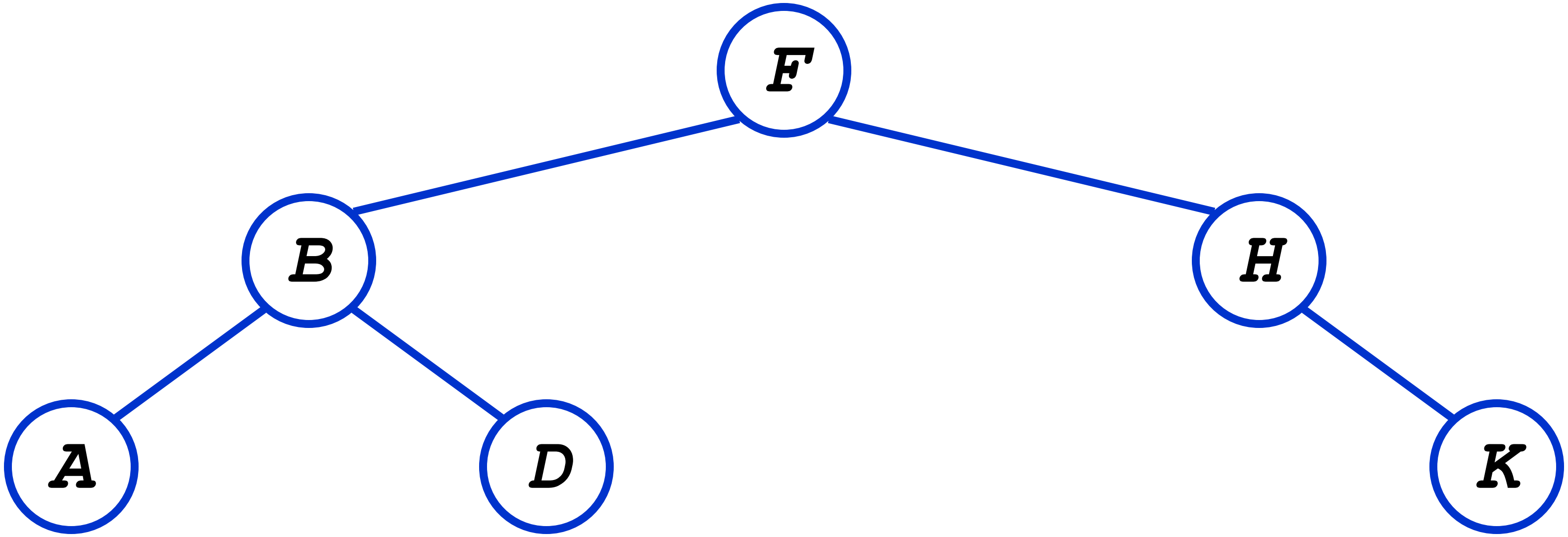
Operations on BSTs: Search

- Given a key and a pointer to a node, returns an element with that key or NULL:

```
TreeSearch(x, k)
    if (x = NULL  or  k = key[x])
        return x;
    if (k < key[x])
        return TreeSearch(left[x], k);
    else
        return TreeSearch(right[x], k);
```

BST Search: Example

- Search for *D* and *C*:



Operations on BSTs: Search

- Here’s another function that does the same:

```
TreeSearch(x, k)
    while (x != NULL  and  k != key[x])
        if (k < key[x])
            x = left[x];
        else
            x = right[x];
    return x;
```

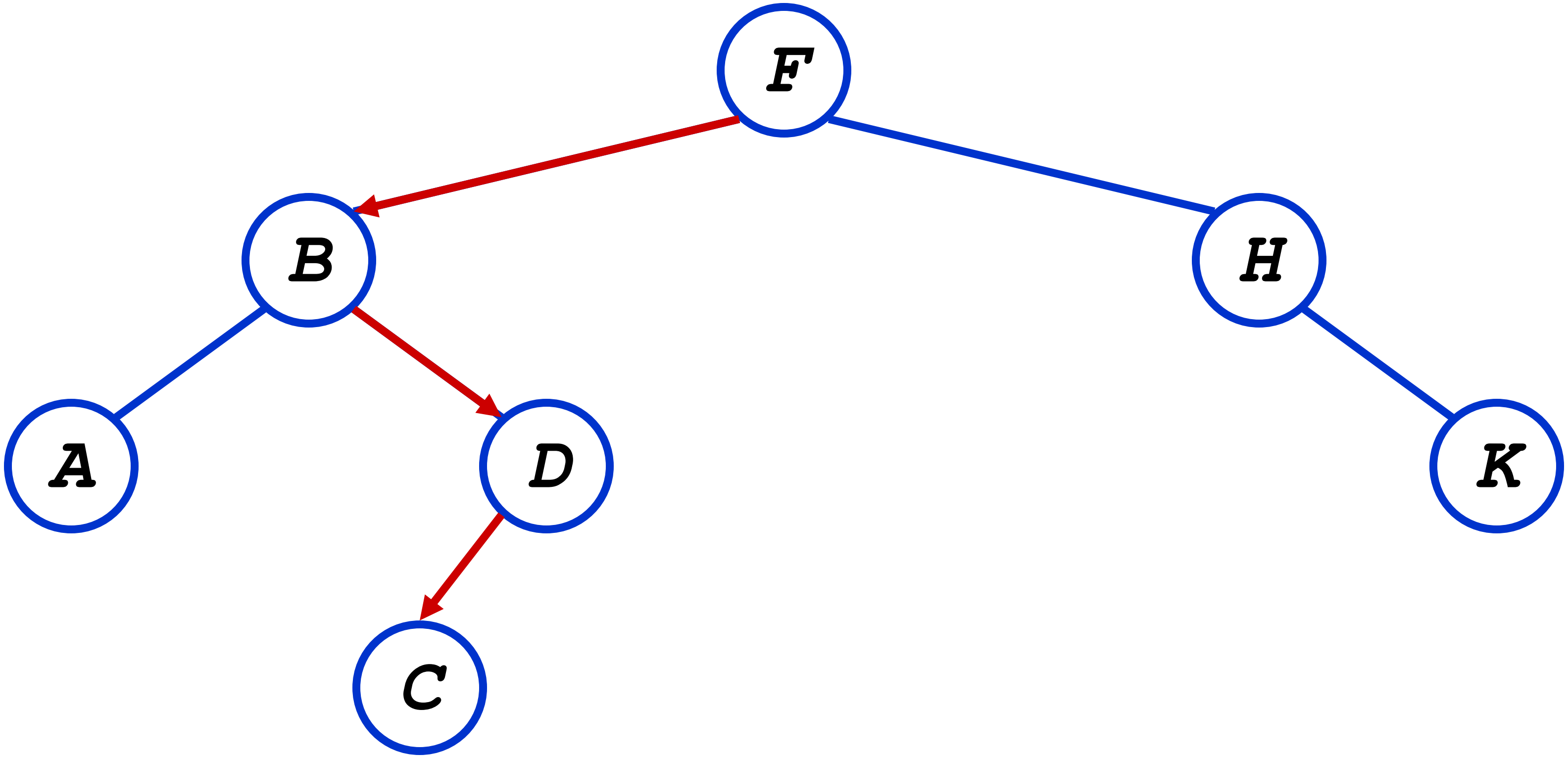
- *Which of these two functions is more efficient?*

Operations of BSTs: Insert

- Adds an element x to the tree so that the binary search tree property continues to hold
- The basic algorithm
  - Like the search procedure above
  - Insert x in place of NULL
  - Use a “trailing pointer” to keep track of where you came from (like inserting into singly linked list)

BST Insert: Example

- Example: Insert *C*



BST Search/Insert: Running Time

- *What is the running time of TreeSearch() or TreeInsert()?*
- A: O(h), where h = height of tree
- *What is the height of a binary search tree?*
- A: worst case: h = O(n) when tree is just a linear string of left or right children
  - We'll keep all analysis in terms of h for now
  - Later we'll see how to maintain h = O(lg n)

Sorting With Binary Search Trees

- Informal code for sorting array A of length *n*:

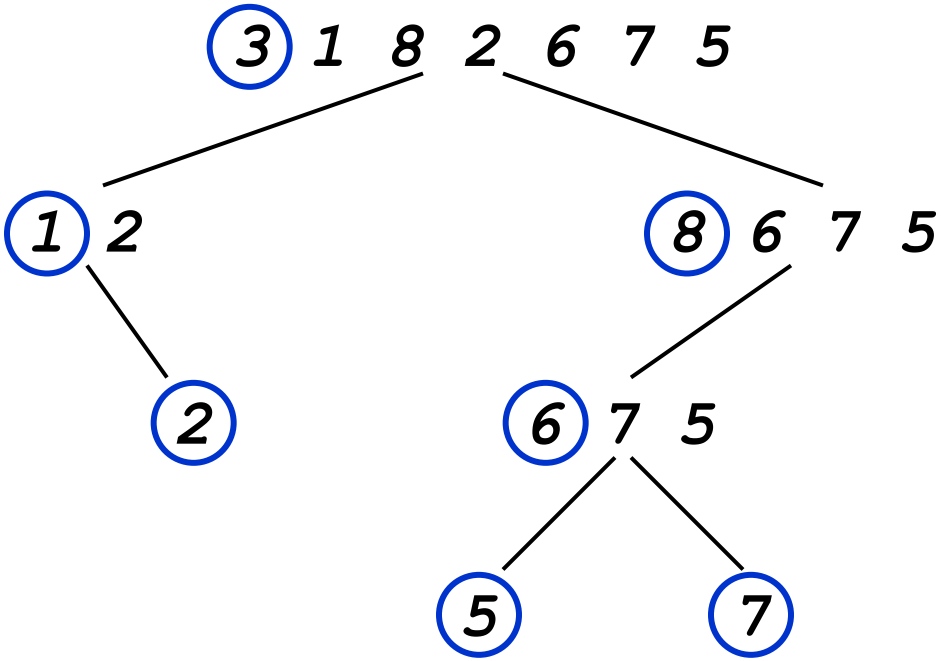
```
BSTSort(A)
    for i=1 to n
        TreeInsert(A[i]);
    InorderTreeWalk(root);
```

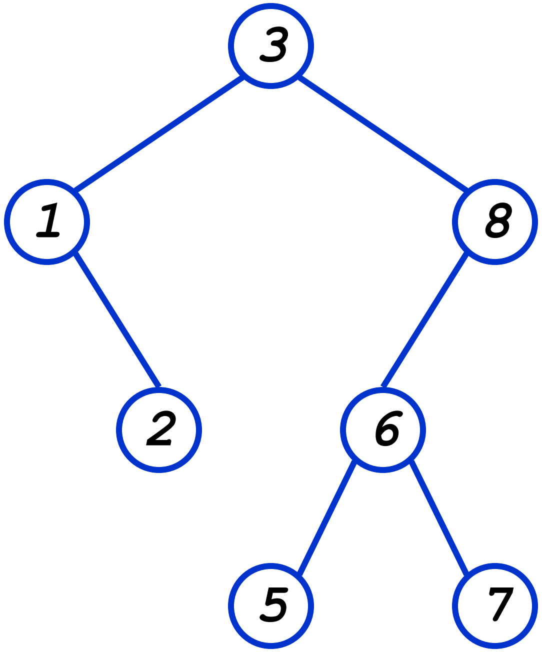
- *Argue that this is $\Omega(n \lg n)$*
- *What will be the running time in the*
    - *Worst case?*
    - *Average case? (hint: remind you of anything?)*

Sorting With BSTs

- Average case analysis
    - It's a form of quicksort!

```
for i=1 to n
    TreeInsert(A[i]);
InorderTreeWalk(root);
```





Sorting with BSTs

- Same partitions are done as with quicksort, but in a different order
  - In previous example
    - Everything was compared to 3 once
    - Then those items < 3 were compared to 1 once
    - Etc.
  - Same comparisons as quicksort, different order!
    - Example: consider inserting 5

Sorting with BSTs

- Since run time is proportional to the number of comparisons, same time as quicksort: O(n lg n)
- *Which do you think is better, quicksort or BSTsort? Why?*

Sorting with BSTs

- Since run time is proportional to the number of comparisons, same time as quicksort: O(n lg n)
- *Which do you think is better, quicksort or BSTSort? Why?*
- A: quicksort
  - Better constants
  - Sorts in place
  - Doesn’t need to build data structure

More BST Operations

- BSTs are good for more than sorting. For example, can implement a priority queue
- *What operations must a priority queue have?*
  - Insert
  - Minimum
  - Extract-Min

BST Operations: Minimum

- *How can we implement a Minimum() query?*
- *What is the running time?*

BST Operations: Successor

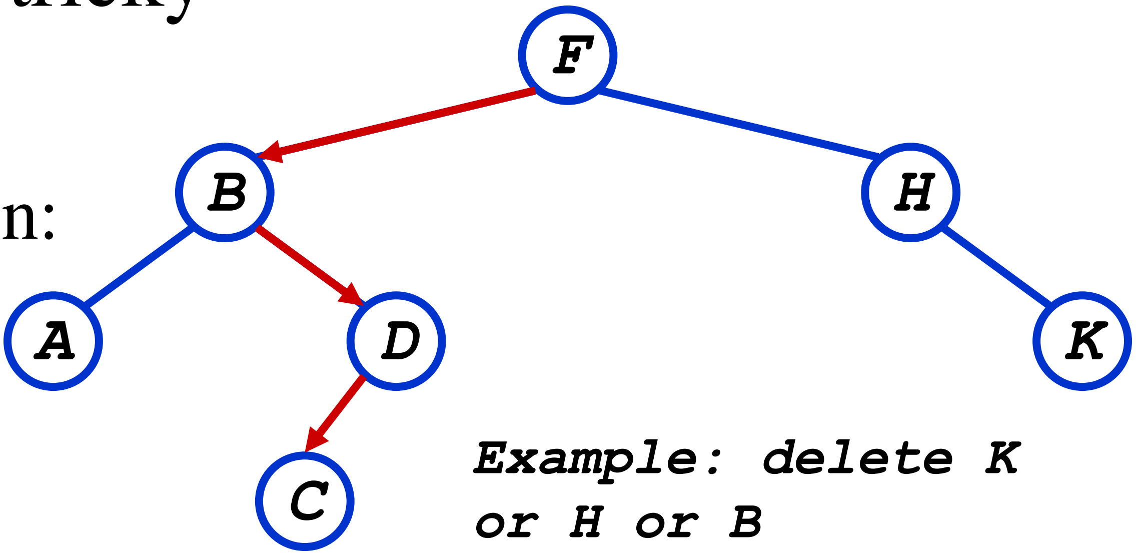
- For deletion, we will need a Successor() operation
- Draw Fig 13.2
- *What is the successor of node 3? Node 15? Node 13?*
- *What are the general rules for finding the successor of node x? (hint: two cases)*

BST Operations: Successor

- Two cases:
  - x has a right subtree: successor is minimum node in right subtree
  - x has no right subtree: successor is first ancestor of x whose left child is also ancestor of x
    - Intuition: As long as you move to the left up the tree, you’re visiting smaller nodes.
- Predecessor: similar algorithm

BST Operations: Delete

- Deletion is a bit tricky
- 3 cases:
  - x has no children:
    - Remove x
  - x has one child:
    - Splice out x
  - x has two children:
    - Swap x with successor
    - Perform case 1 or 2 to delete it



Example: delete K or H or B

BST Operations: Delete

- *Why will case 2 always go to case 0 or case 1?*
- A: because when x has 2 children, its successor is the minimum in its right subtree
- *Could we swap x with predecessor instead of successor?*
- A: yes. *Would it be a good idea?*
- A: might be good to alternate
- Up next: guaranteeing a O(lg n) height tree