

# CS 372: Algorithms

## Binary Search Trees

(Based on slides by David Luebke)

1

## Dynamic Sets

- This lecture 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*( $S, x$ )

2

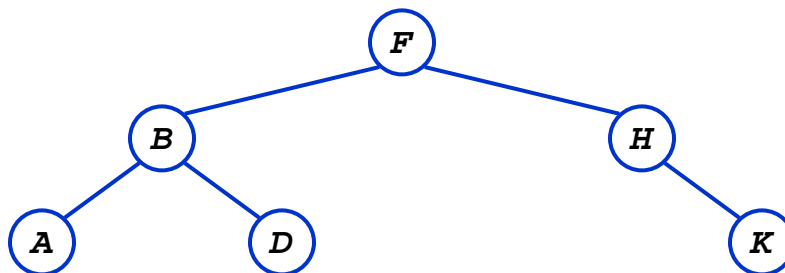
## Review: Binary Search Trees

- *Binary Search Trees* (BSTs) are an important data structure for dynamic sets
- In addition to satellite data, elements have:
  - *key*: an identifying field
  - *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)

3

## Review: Binary Search Trees

- BST property:  
 $key[leftSubtree(x)] \leq key[x] \leq key[rightSubtree(x)]$
- Example:



4

## Inorder Tree Walk

- *What does the following code do?*

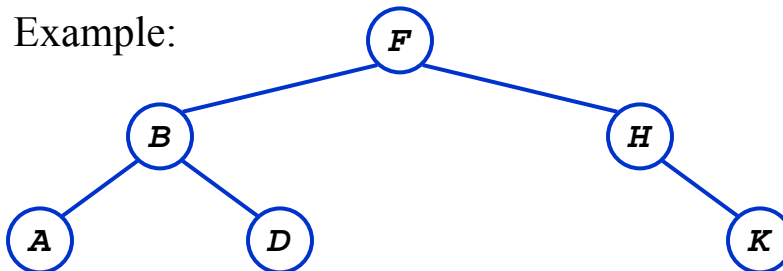
```
TreeWalk(x)
  TreeWalk(left[x]);
  print(x);
  TreeWalk(right[x]);
```

- 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

5

## Inorder Tree Walk

- Example:



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

6

## Operations on BSTs: Search

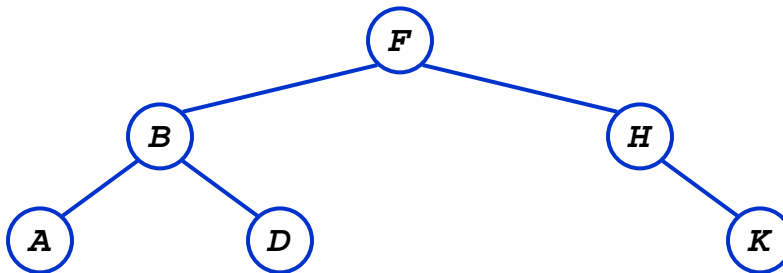
- Given a key and a pointer to a node, returns an element with that key or NULL:

```
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);
```

7

## BST Search: Example

- Search for *D* and *C*:



8

## 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?*

9

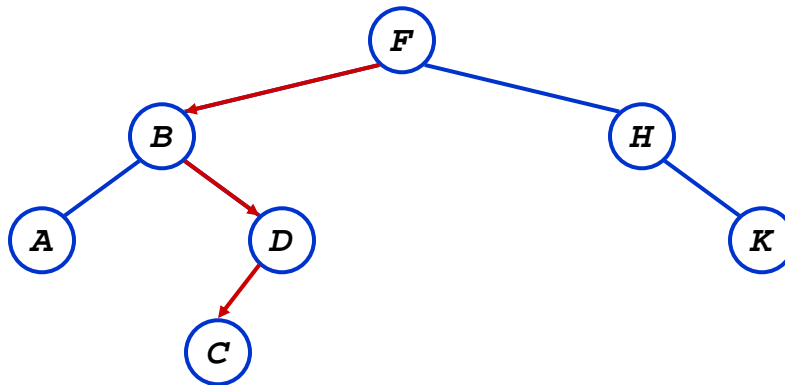
## Operations of BSTs: Insert

- 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)

10

## BST Insert: Example

- Example: Insert  $C$



11

## 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  $h = O(\lg n)$

12

## 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?)

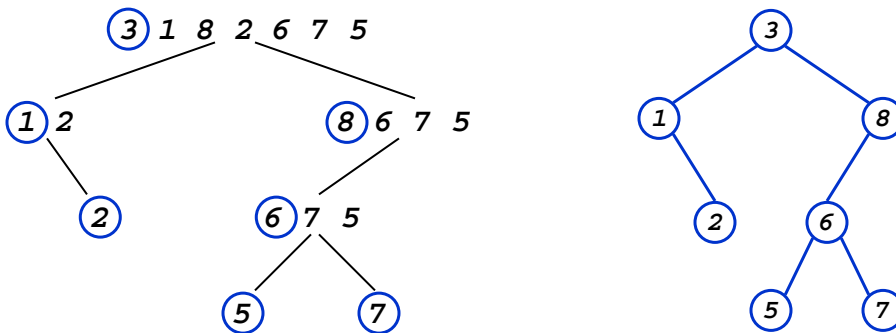
13

## Sorting With BSTs

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

```

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



14

## 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!

15

## Sorting with BSTs

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

16

## 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

17

## 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

18

## BST Operations: Minimum

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

19

## BST Operations: Successor

- For deletion, we will need a Successor() operation
- Draw Fig 12.2
- *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)*

20

## 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

21

## BST Operations: Delete

- Deletion is a bit tricky

- 3 cases:

- x has no children:

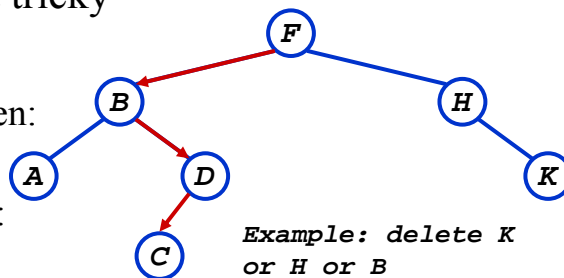
- Remove x

- x has one child:

- Splice out x

- x has two children:

- Swap x with successor
- Perform case 1 or 2 to delete it



22

## 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