CPSC 5031 : Data Structures & Algorithms

Lecture 6: Binary Search Trees

(Levitin, Chapter 4.5, 5.3)

Review: Linear Search

- Collection of data items to be searched is organized in a list

  $x_1, x_2, \ldots x_n$

  – Assume == and < operators defined for the type

- Linear search begins with item 1
  – continue through the list until target found
  – *or* reach end of list

Binary Search

- Two requirements?

Binary Search

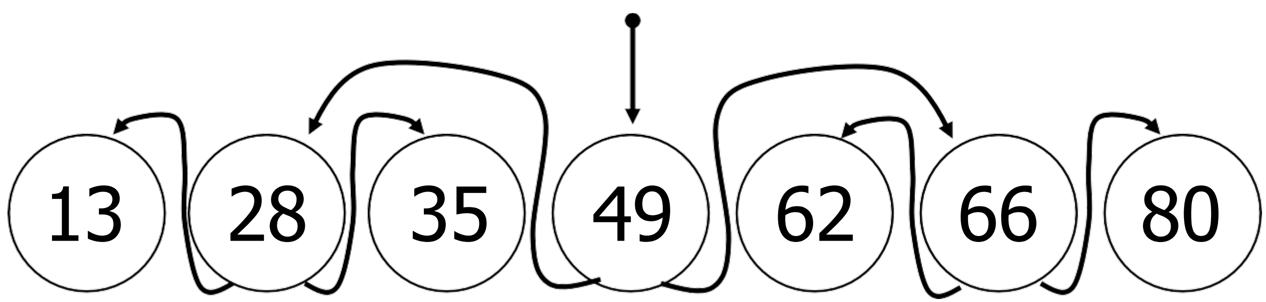
- Two requirements
  - The data items are in ascending order (can they be in decreasing order? 😀)
  - Direct access of each data item for efficiency (why linked-list is not good!)

Binary Search vs. Linear Search

- Usually outperforms Linear Search:
  O(log n) vs. O(n)
- Disadvantages
  - Sorted list of data items (O(n log n) best case)
  - Direct access of storage structure, not good for linked-list
- Good news: It is possible to use a linked structure which can be searched in a binary-like manner

Binary Search Tree (BST)

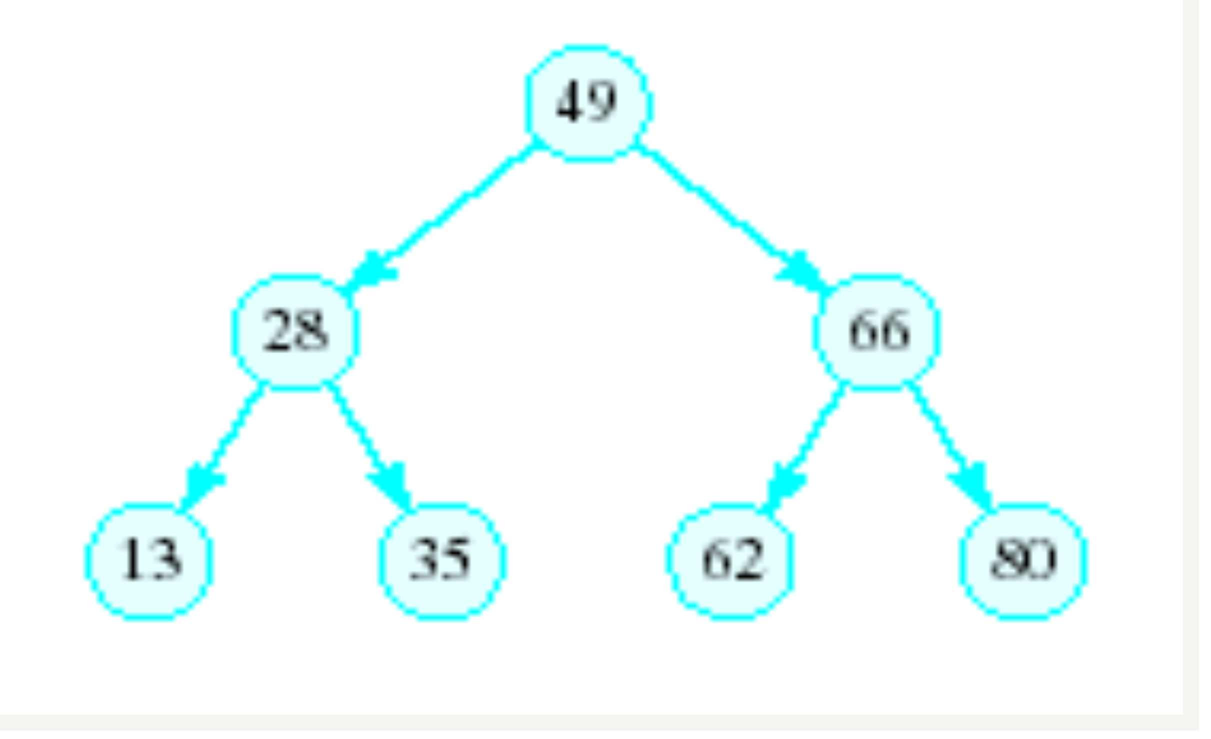
- Consider the following ordered list of integers



1. Examine middle element
2. Examine left, right sublist (maintain pointers)
3. (Recursively) examine left, right sublists

Binary Search Tree

- Redraw the previous structure so that it has a treelike shape – a binary tree

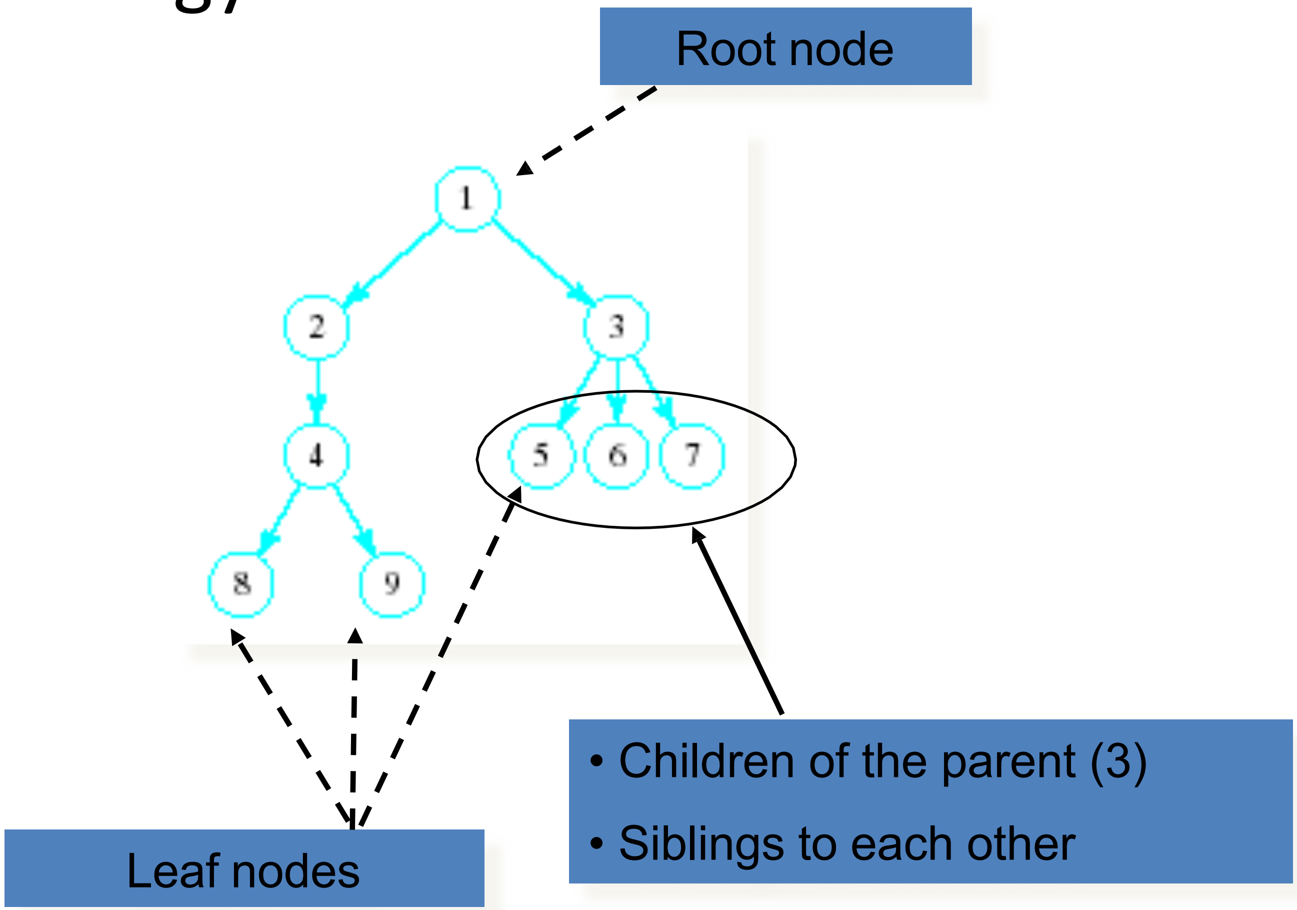


Trees

- A data structure which consists of
  - a finite set of elements called nodes or vertices
  - a finite set of directed arcs which connect the nodes
- If the tree is nonempty
  - one of the nodes (the root) has no incoming arc
  - every other node can be reached by following a unique sequence of consecutive arcs (or paths)

Trees

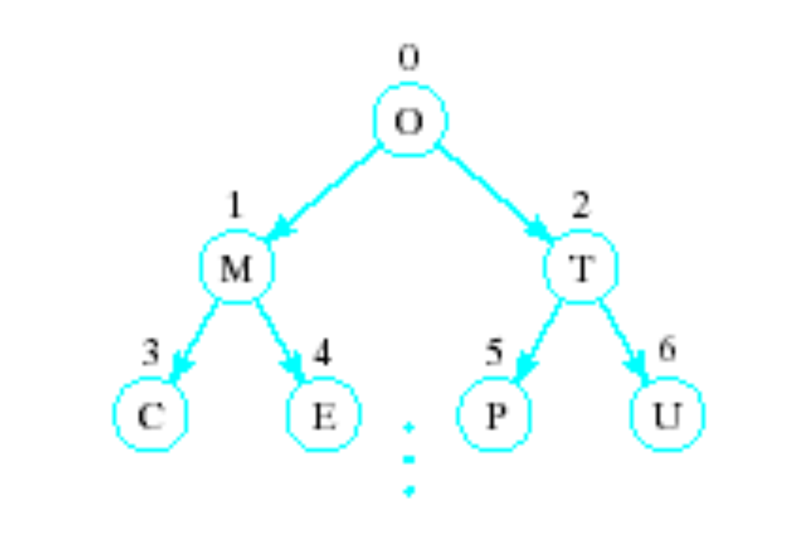
- Tree terminology



Binary Trees

- Each node has at most two children
- Useful in modeling processes where
  - a comparison or experiment has exactly two possible outcomes
  - the test is performed repeatedly
- Example
  - multiple coin tosses
  - encoding/decoding messages in dots and dashes such as Morse code

Array Representation of Binary Trees

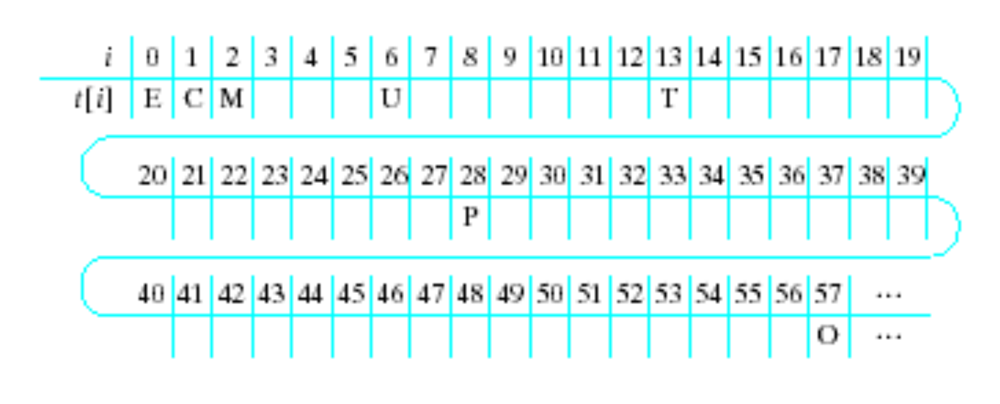


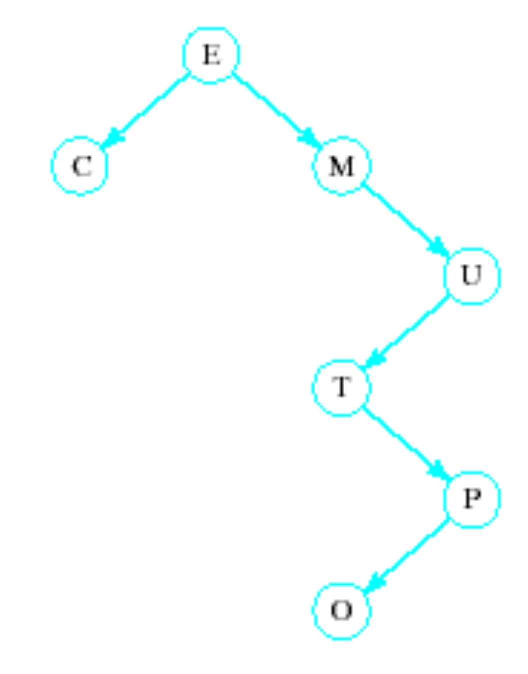
i	0	1	2	3	4	5	6	...
$t[i]$	O	M	T	C	E	P	U	...

- Store the i^{th} node in the i^{th} location of the array

Array Representation of Binary Trees

- Works OK for complete trees, not for sparse trees





Some Tree Definition

- Complete trees
  - Each level is completely filled except the bottom level
  - The leftmost positions are filled at the bottom level
  - Array storage is perfect for them
- Balanced trees
  - Binary trees
  - |left_subtree| – |right_subtree|<=1

This means “depth”

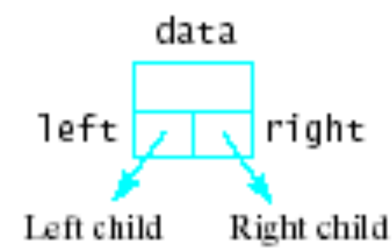
- Tree Height/Depth:
  - Number of levels

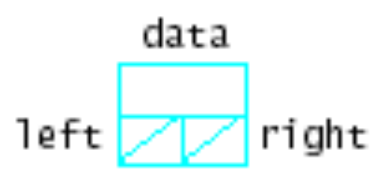
Tree Questions

- A complete tree must be a balanced tree?
- Give a node with position *i* in a complete tree, what are the positions of its child nodes?
  - Left child?
  - Right child?

Linked Representation of Binary Trees

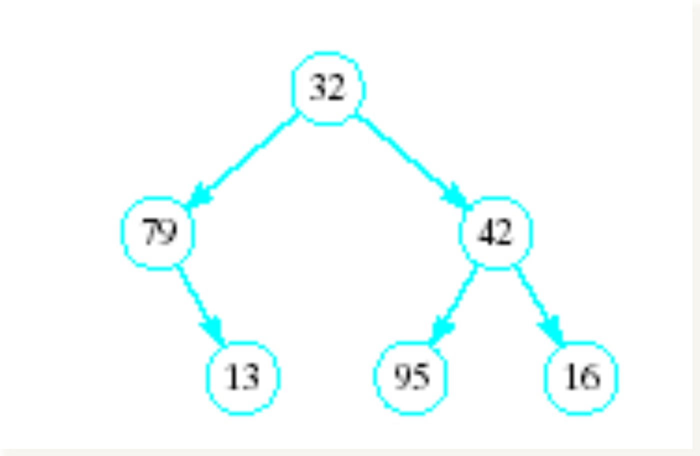
- Uses space more efficiently
- Provides additional flexibility
- Each node has two links
  - one to the left child of the node
  - one to the right child of the node
  - if no child node exists for a node, the link is set to NULL

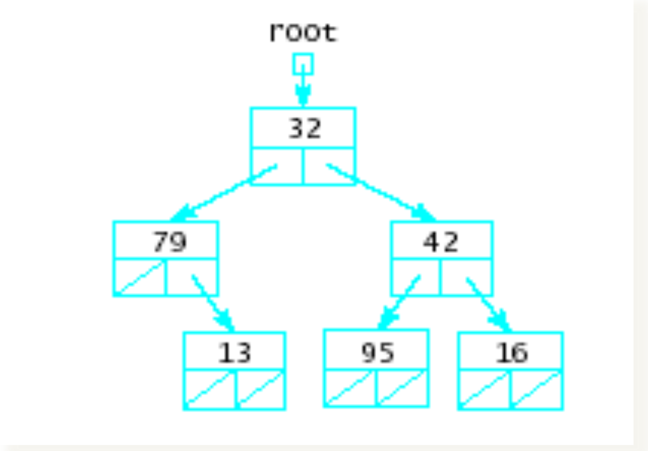




Linked Representation of Binary Trees

- Example





Binary Trees as Recursive Data Structures

- A binary tree is either empty …

  or

- Consists of
  - a node called the root
  - root has pointers to two disjoint binary (sub)trees called …
    - right (sub)tree
    - left (sub)tree

Basis step

Inductive step

Which is either empty … or …

Which is either empty … or …

Tree Traversal is Recursive

If the binary tree is empty then
do nothing

Else
N: Visit the root, process data
L: Traverse the left subtree
R: Traverse the right subtree

Basis step

The inductive step

Traversal Order

Three possibilities for inductive step …

- Left subtree, Node, Right subtree
  the <u>inorder</u> traversal

- Node, Left subtree, Right subtree
  the <u>preorder</u> traversal

- Left subtree, Right subtree, Node
  the <u>postorder</u> traversal

Binary Search Tree (BST)

- Collection of Data Elements
  - Binary tree
  - BST property: for each node x,
    - value in left child of x $<$ value in x $<$ in right child of x
- Basic operations
  - Construct an empty BST
  - Determine if BST is empty
  - Search BST for given item

Binary Search Tree (BST)

- Basic operations (ctd)
  - Insert a new item in the BST
    - Maintain the BST property
  - Delete an item from the BST
    - Maintain the BST property
  - Traverse the BST
    - Visit each node exactly once
    - The *inorder traversal* must visit the values in the nodes in ascending order

BST

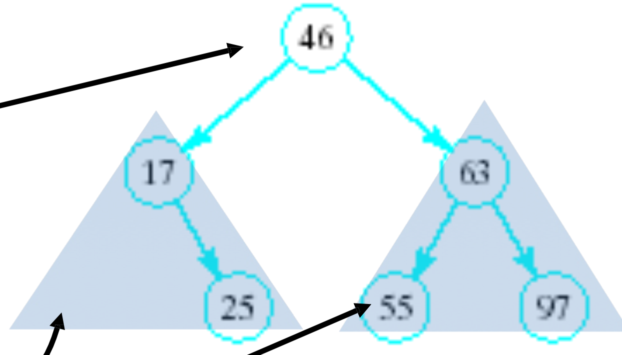
```
typedef  int  T;
class BST {
     public:
        BST() : myRoot(0) {}
        bool empty() { return myRoot==NULL; }
     private:
        class BinNode {
           public:
              T  data;
              BinNode* left, right;
              BinNode() : left(0), right(0) {}
              BinNode(T item): data(item), left(0), right(0) {}
        }; //end of BinNode
        typedef BinNode*  BinNodePtr;
        BinNodePtr  myRoot;
};
```

BST Traversals

- Inorder, preorder and postorder traversals (recursive)
- Non-recursive traversals
- Done in divide-and-conquer

BST Searches

- Search begins at root
  - If that is desired item, done
- If item is less, move down left subtree
- If item searched for is greater, move down right subtree
- If item is not found, we will run into an empty subtree



Insertion Operation

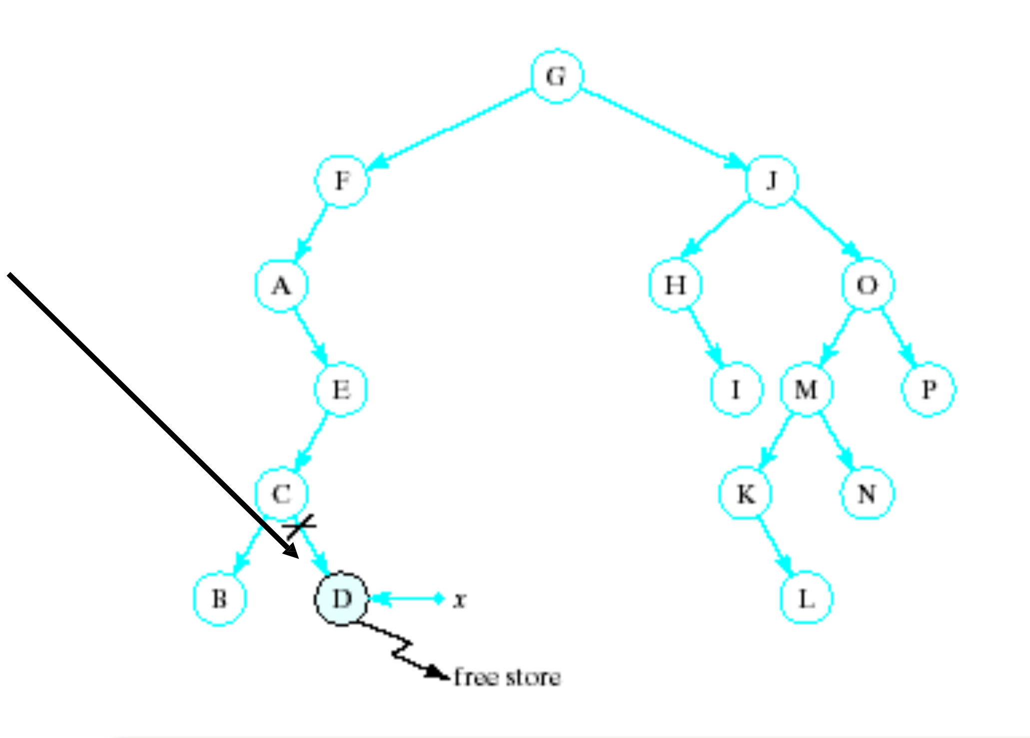
Basic idea:

- Use search operation to locate the insertion position or already existing item
- Use a parent point pointing to the parent of the node currently being examined as descending the tree

Deletion Operation

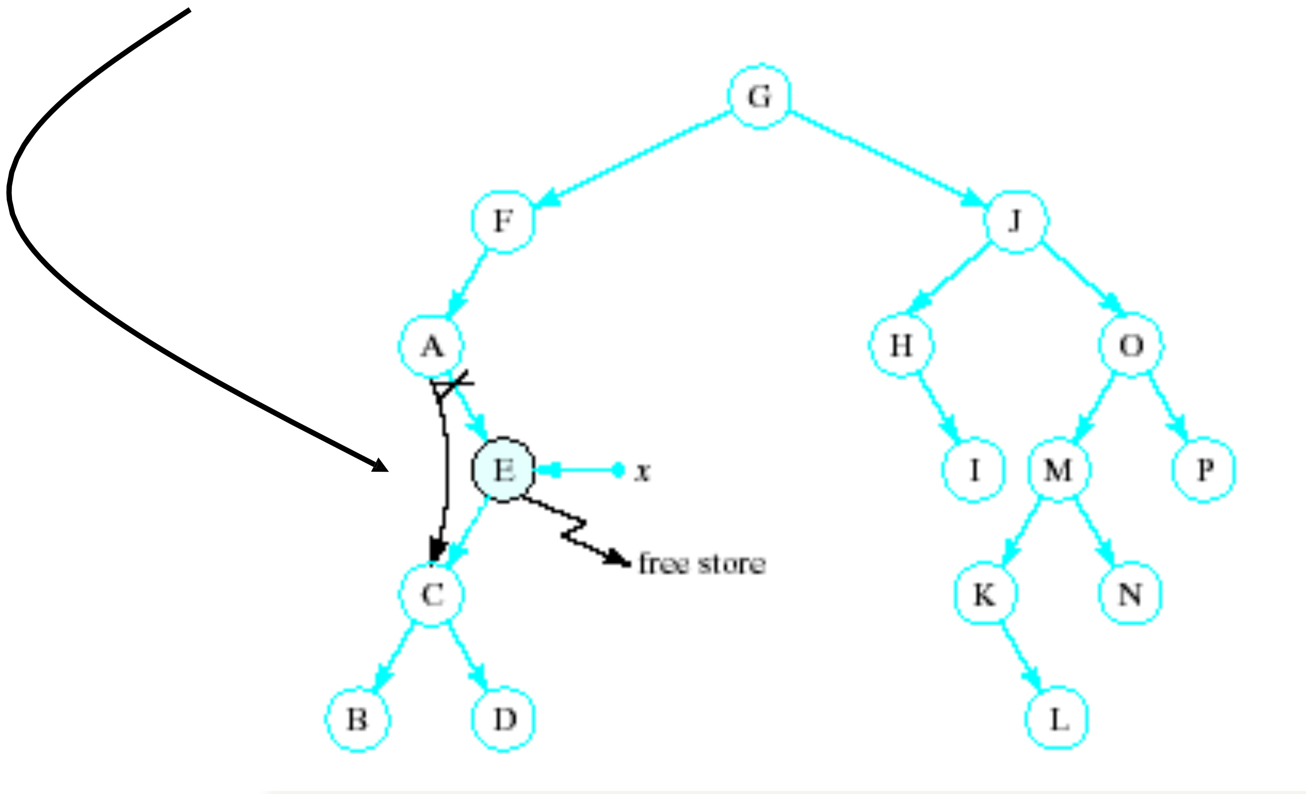
Three possible cases to delete a node, *x*, from a BST

1. The node, *x*, is a leaf



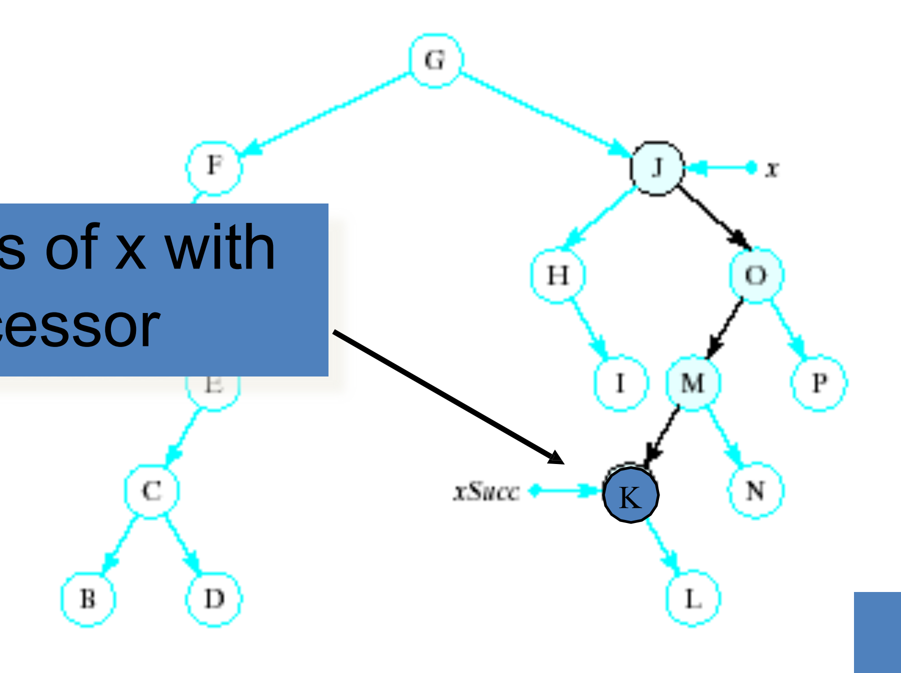
Deletion Operation

2. The node, *x* has one child



Deletion Operation

3. The node, *x* has two children



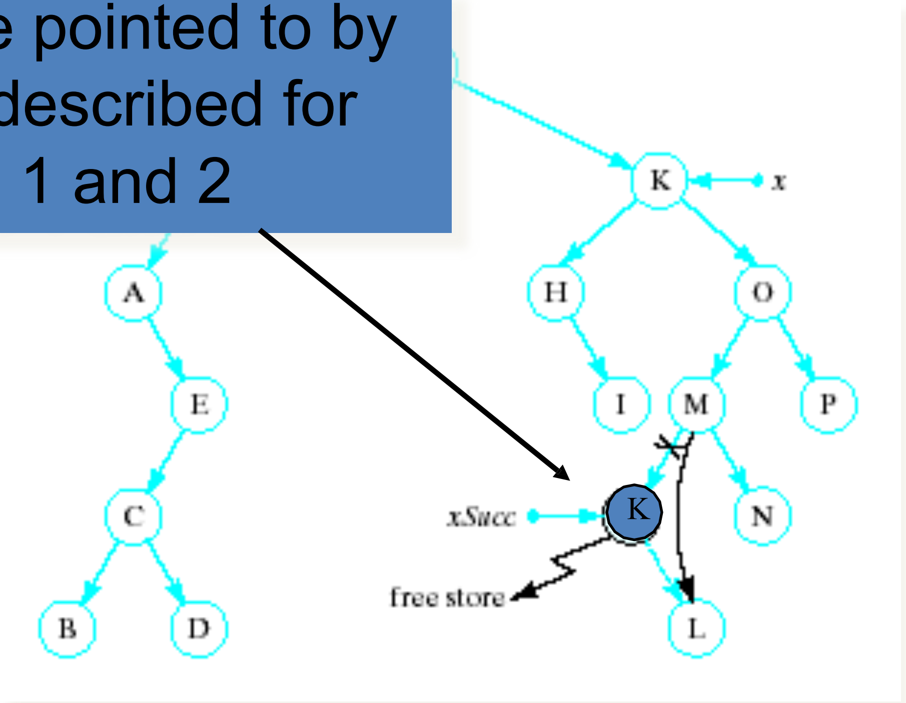
Replace contents of x with inorder successor

[View **`remove()`** function]

Deletion Operation

3. The node, *x* has two children (cont.)

Delete node pointed to by xSucc as described for cases 1 and 2



View **`remove()`** function

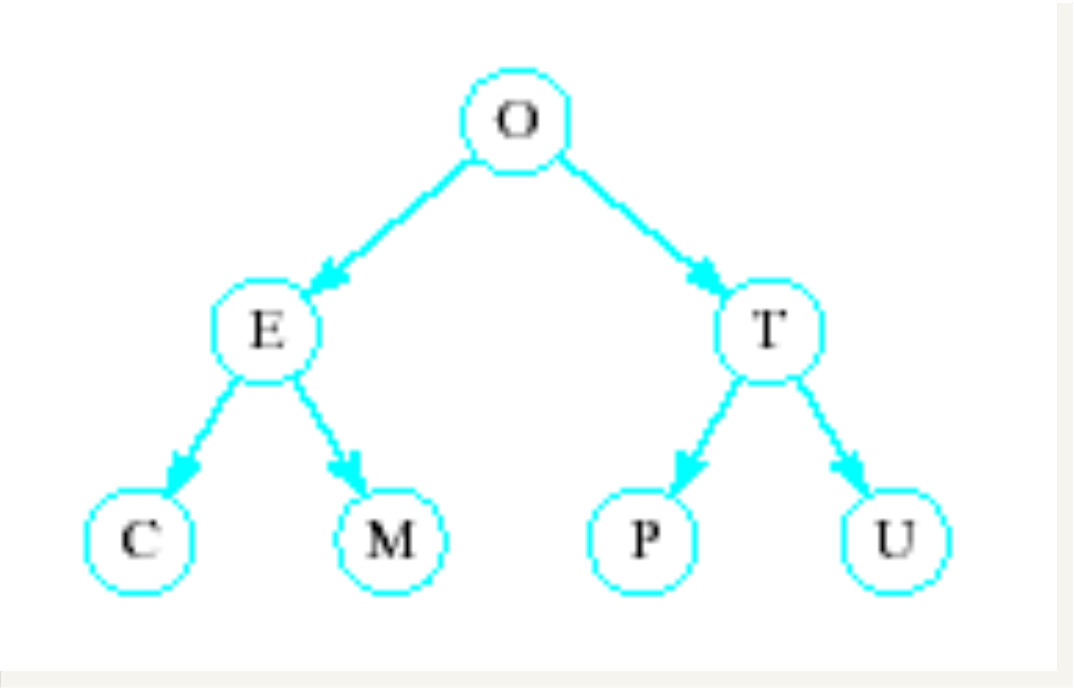
Questions?

- What is T(n) of search algorithm in a BST? Why?
- What is T(n) of insert algorithm in a BST?
- Other operations?

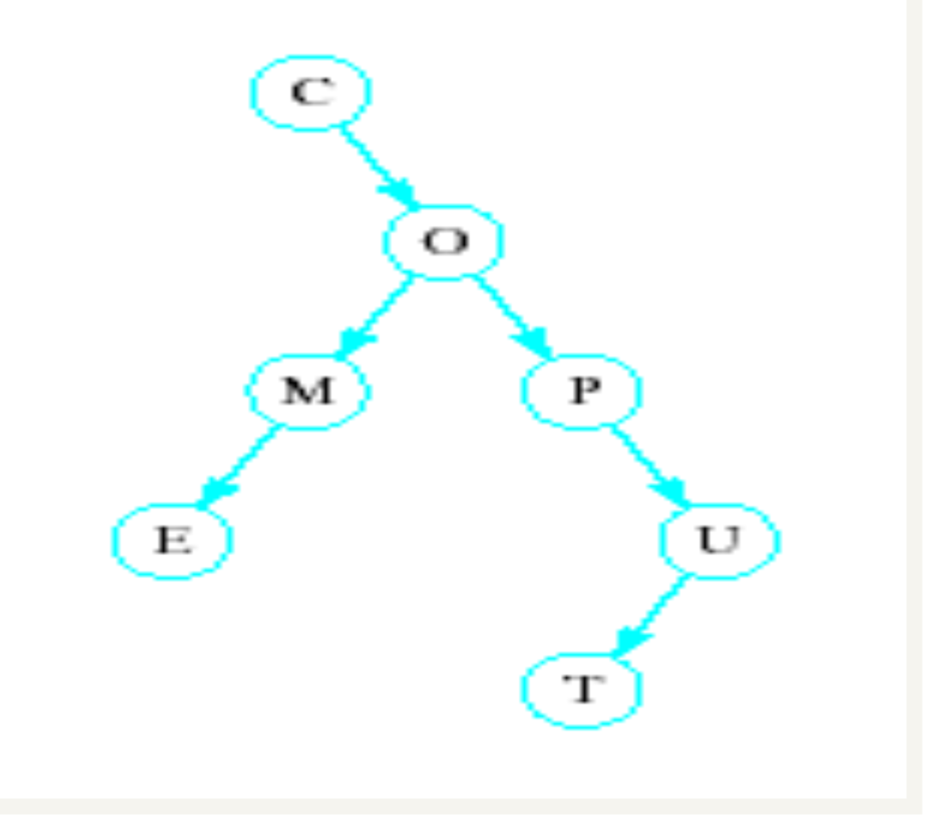
Problem of Lopsidedness

- The order in which items are inserted into a BST determines the shape of the BST
- Result in Balanced or Unbalanced trees
- Insert O, E, T, C, U, M, P
- Insert C, O, M, P, U, T, E

Balanced

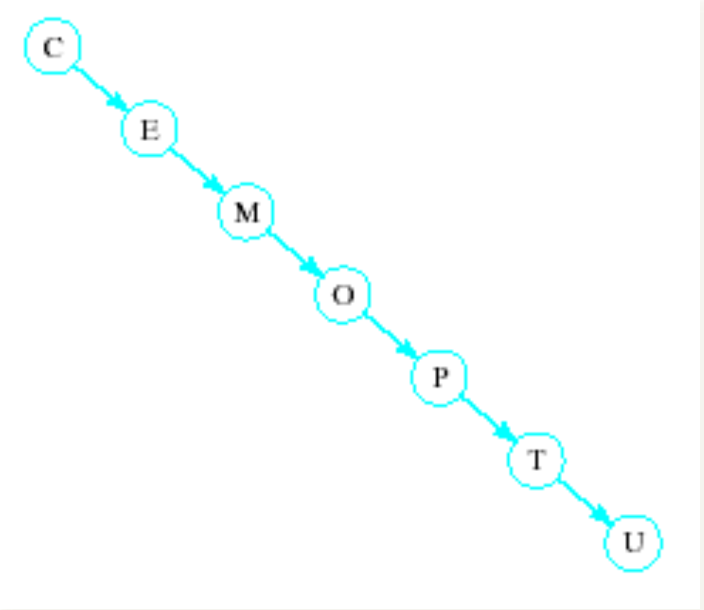


Unbalanced



Problem of Lopsidedness

- Trees can be totally lopsided
  - Suppose each node has a right child only
  - Degenerates into a linked list



Processing time affected by "shape" of tree

Next week…

- Heap, Heap Sort, and Priority Queues
  - Read 6.4