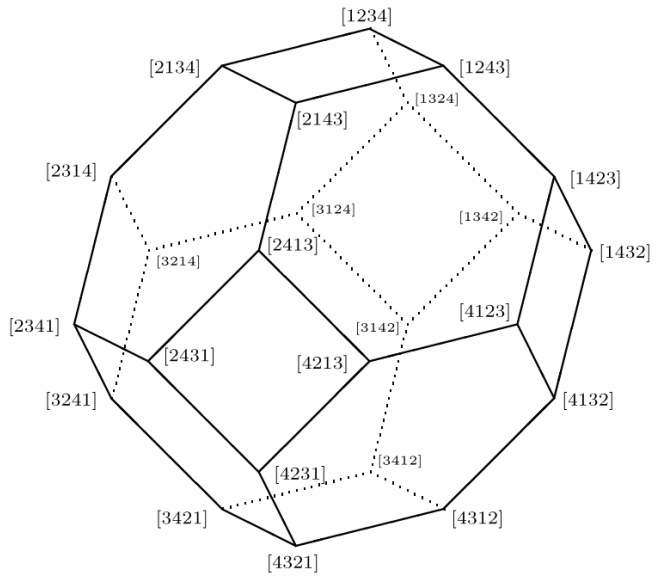
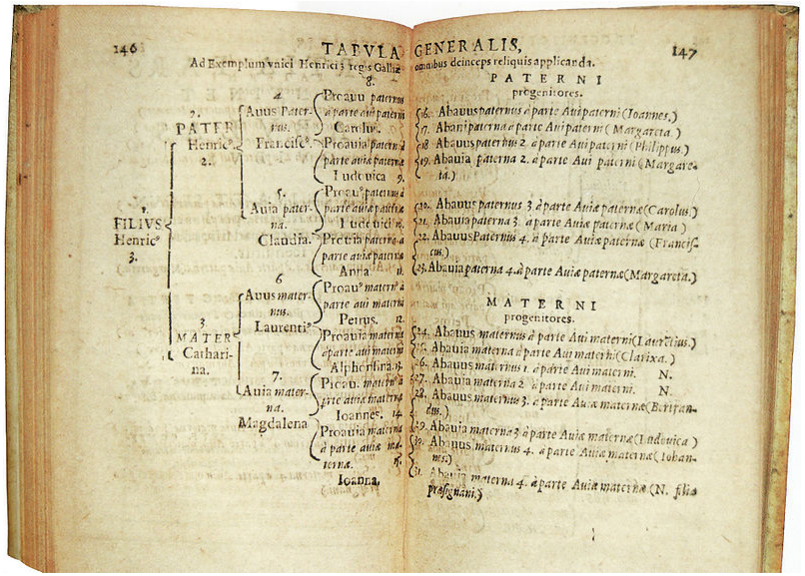
# CS3334 Data Structures

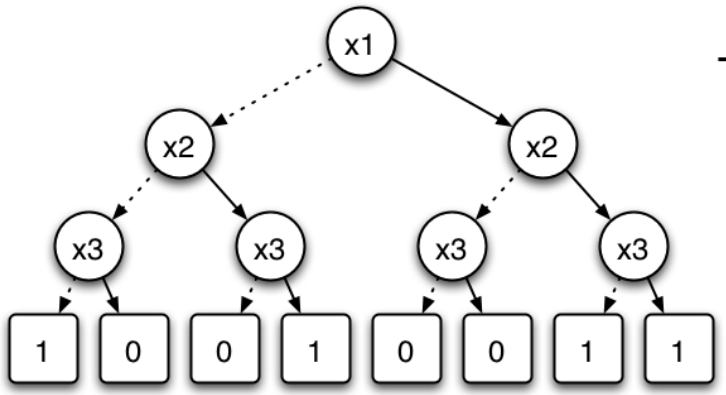
## Lecture 9: Binary Search Trees


Chee Wei Tan

# Binary Trees in Computer Science



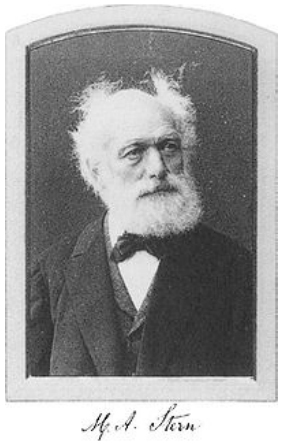



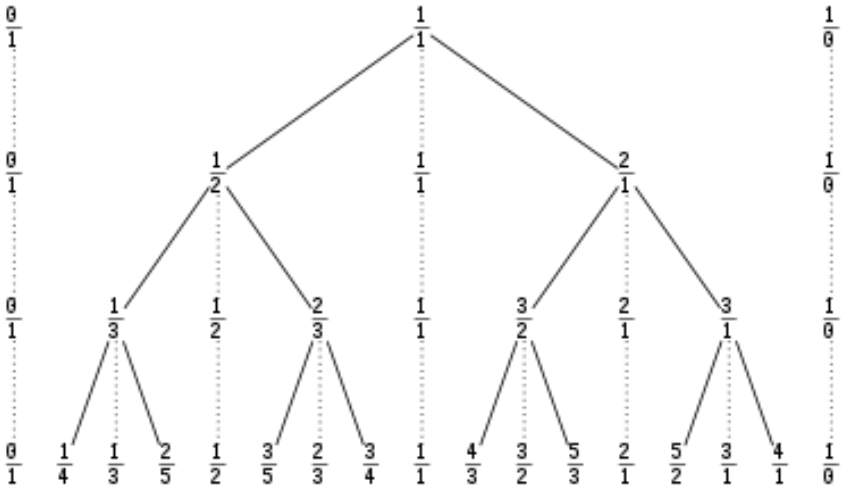

| x1 | x2 | x3 | f |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

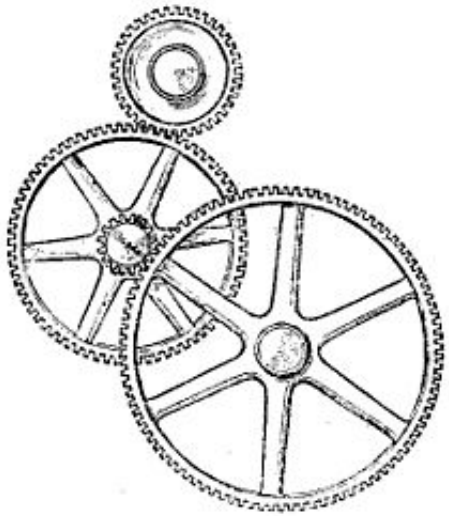
**Ahnentafel method** (Lecture 7)
https://en.wikipedia.org/wiki/Ahnentafel

**Binary decision tree**
Fundamental theorem of Boolean algebra (Shannon's expansion theorem). Binary decision trees lead to binary decision diagrams that Donald Knuth mentioned in 2018 as "one of the only really fundamental data structures that came out in the last twenty-five years".
Can you write down binary decision trees of a half- adder?
A full adder?

# Mathematics and Clockwork





In number theory, the **Stern–Brocot tree** is an infinite complete binary tree in which the vertices correspond one-for-one to the positive rational numbers, whose values are ordered from the left to the right as in a search tree. The Stern–Brocot tree was discovered independently by **Moritz Stern** (1858) and **Achille Brocot** (1861). Stern was a German number theorist; Brocot was a French clockmaker who used the Stern–Brocot tree to design systems of gears with an engineered gear ratio.

This is a binary tree data structure useful for approximating real numbers by rational numbers. What problems can you solve with it? 🤔

https://en.wikipedia.org/wiki/Stern%E2%80%93Brocot_tree

# Binary Search Tree

## Some Combinatorial Properties of Certain Trees With Applications to Searching and Sorting*

THOMAS N. HIBBARD

*System Development Corporation, Santa Monica, California*

### INTRODUCTION

This paper introduces an abstract entity, the binary search tree, and exhibits some of its properties. The properties exhibited are relevant to processes occurring in stored program computers—in particular, to search processes. The discussion of this relevance is deferred until Section 2.

Section 1 constitutes the body of the paper. Section 1.1 consists of some mathematical formulations which arise in a natural way from the somewhat less formal considerations of Section 2.1. The main results are Theorem 1 (Section 1.2) and Theorem 2 (Section 1.3).

The initial motivation of the paper was an actual computer programming problem. This problem was the need for a list which could be searched efficiently and also changed efficiently. Section 2.1 contains a description of this problem and explains the relevance to its solution of the results of Section 1.

Section 2.2 contains an application to sorting.

The reader who is interested in the programming applications of the results but not in their mathematical content can profit by reading Section 2 and making only those few references to Section 1 which he finds necessary.

### 1. COMBINATORIAL PROPERTIES OF BINARY SEARCH TREES

#### 1.1 *Preliminary Definitions*

The central notion of this paper is that of a certain type of directed graph undergoing "random" operations. The present section is given to defining the graph and certain quantities associated with the graph. "Expected" values of these quantities will be defined combinatorially in Section 1.2; these "expected" values will then be calculated assuming "random insertions" (Section 1.2) and "random deletions" (Section 1.3).

DEFINITION. A *binary search tree* is a directed graph[1] having the following properties.

* Received March, 1961; revised July, 1961.

oduct

INFORMATION AND CONTROL 3, 327–334 (1960)

## On the Efficiency of a New Method of Dictionary Construction

A. D. BOOTH AND A. J. T. COLIN

*Department of Numerical Automation, Birkbeck College, University of London*

Programs for constructing dictionaries of texts, with computers, have sometimes been adaptations of methods suitable for manual construction with card indexes. With all card index methods it is customary to keep the different words collected in alphabetical order, for the structure of a card index lends itself to such a process: all that is necessary to insert a new word into the index between two existing ones is to make out a new card and to put it in the correct place. However, the insertion of a new word in the store of a computer where the words are kept in alphabetical order is a time-consuming process, for all the words below the one which is inserted have to be "moved down" by one place.

If, however, a computer method is used where the words are not stored in alphabetical order but in the order in which they occur, the position is even worse, for although the shifting of words is eliminated, the dictionary search which is necessary for each word in the text, to establish whether it is a new word or not, must involve all the words collected so far, and not just a small number of them, as would be the case if the words were in alphabetical order, and a logarithmic search (Booth, 1955) could be used.

The method of construction described in this paper overcomes both these problems. It is not an adaptation of a manual method, but is designed specifically for computers. The method is based on a tree structure, which is discussed below.

### THE LOGICAL TREE

The logical tree is based on the fact that if $\alpha$ and $\beta$ are two different words, $\alpha$ is either alphabetically less or alphabetically greater than $\beta$. The construction of a tree is illustrated by taking a sentence and making a tree from it.

# Dictionary Data Structure (1/2)

- To maintain a set S of items supporting the following core operations:
  - **Insert(x)**: insert x into S
  - **Delete(x)**: delete x from S
  - **Search(x)**: check if x is present in S
- Assume items can be compared for <, =, >

# Dictionary Data Structure (2/2)

- Support one or more of these operations:
  - **Predecessor(x)**: retrieve largest y in S s.t. $y \leq x$
  - **Successor(x)**: retrieve smallest y in S s.t. $y \geq x$
  - **Select(k)**: retrieve the k-th smallest x in S
- Some additional operations:
  - **Enumerate()**: list all elements of S in order
  - **Max()**: retrieve largest x in S
  - **Min()**: retrieve smallest x in S
  - **Count()**: compute the number of elements in S

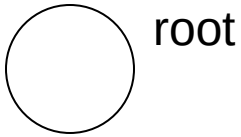
# Some Approaches

- Let n = the number of items in S
- **Sorted array**
  - O(n) time for insert & delete
  - O(logn) time for search (i.e., binary search)
- **Linked list (unsorted)**
  - O(1) time for insert
  - O(n) time for delete & search
- **Hash table**
  - O(1) time for insert, delete & search
  - Cannot support predecessor/successor queries
- **Balanced binary search tree**
  - O(logn) time for all operations

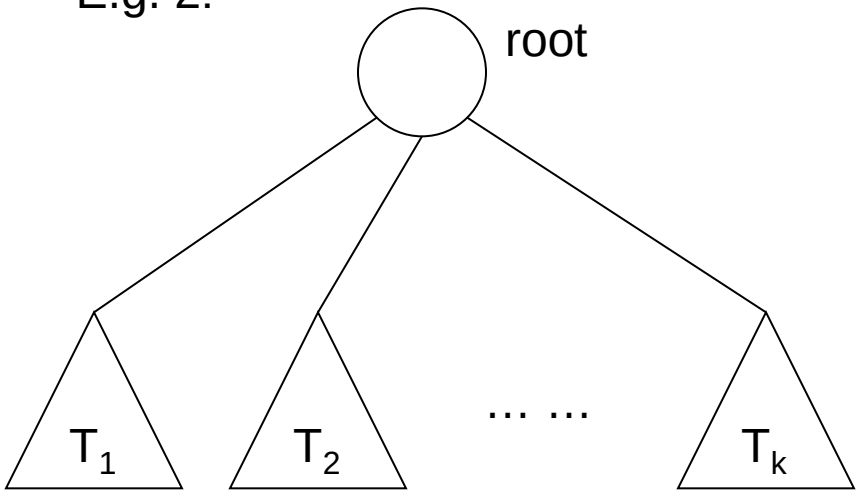
# What is a Tree?

- Recursive definition
  - A *rooted tree* T consists of a root node, and 0 or more nonempty subtrees $T_1$, $T_2$, …, and $T_k$
  - Root of T has an edge to root of each $T_i$

E.g. 1: A single root node is a tree

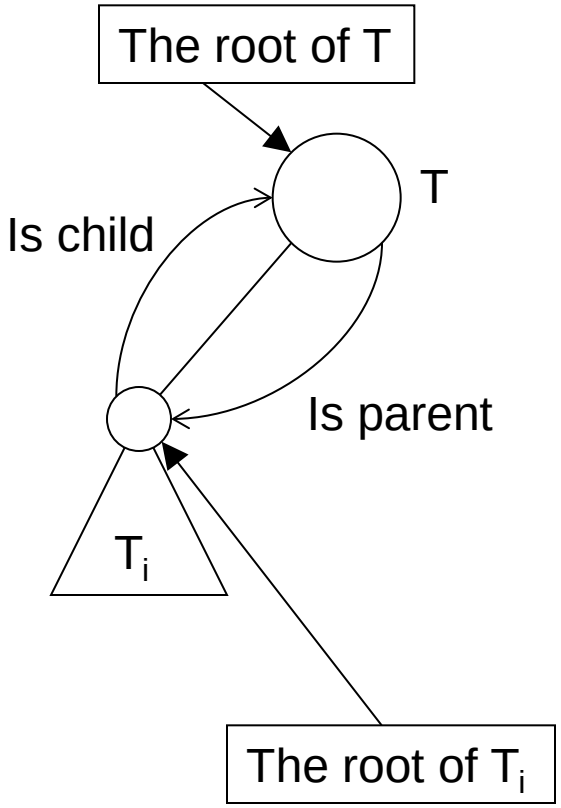



E.g. 2:



# Terminologies (1/2)

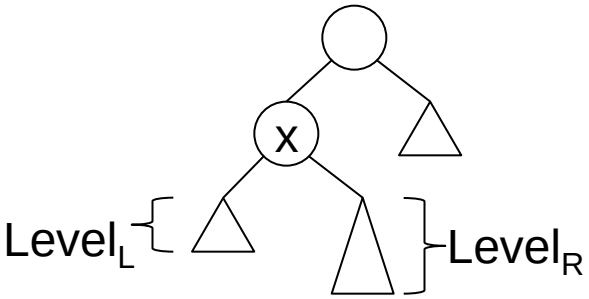
- Root of $T_i$ ($1 \leq i \leq k$) is called a **child** (**son**) of the root of T
- Root of T is a **parent** of root of $T_i$
- If $k \leq 2$ for every node (i.e., k=0, 1, or 2), called a **binary tree**
- Nodes without son are called **leaves**; other nodes are called **internal nodes**
- Path is a sequence of edges between 2 nodes



# Terminologies (2/2)

- Depth of a node (from root):
  - Length of path from root to that node
  - i.e., the number of edges along the path from root to that node
  - So, root has depth 0
- Level of a node (from bottom):
  - Leaf has level 0
  - Internal node has level
    = 1 + max{level of its sons}

Level of x = 1 + max{$Level_L$, $Level_R$}

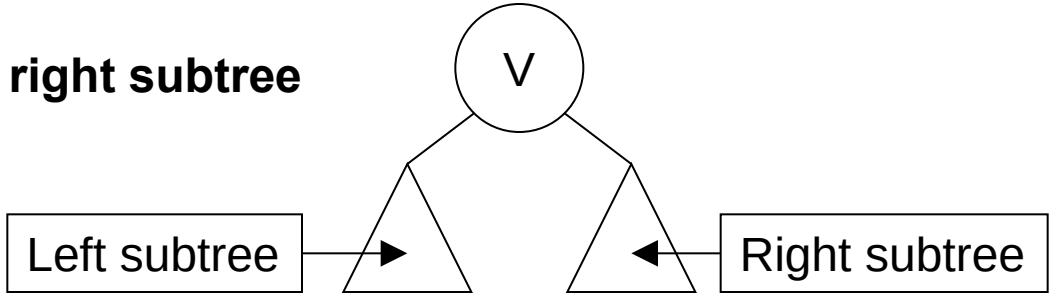



- Height of the tree:
  - 1 + max depth of a node
  - i.e., 1 + the number of edges in the longest root-leaf path
  - i.e., the number of nodes in the longest root-leaf path

# Binary Search Tree (1/2)

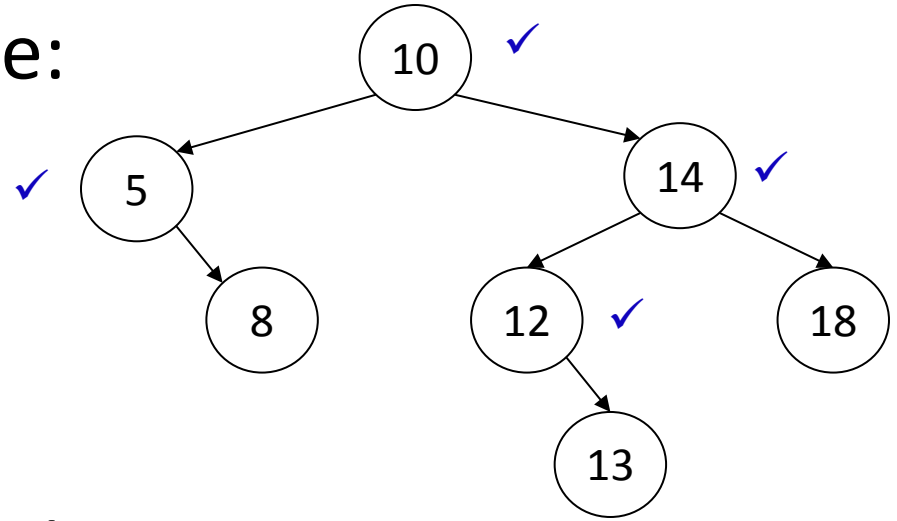
- Each internal node has at most 2 sons
- Elements are stored in nodes, one element per node, such that the *symmetric order* is satisfied:
  - **For each node v**,
    - **Elements stored in v's left subtree ≤ element stored in v, and**
    - **Elements stored in v's right subtree ≥ element stored in v**

**v's left subtree ≤ v ≤ v's right subtree**

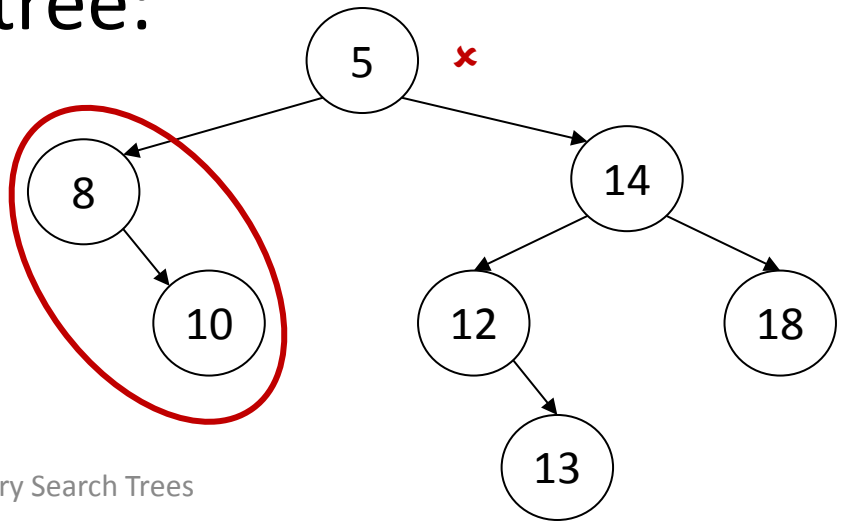


# Binary Search Tree (2/2)

- A binary search tree:

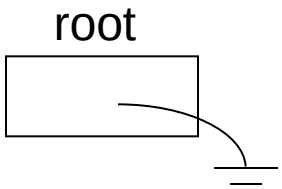



- Not a binary search tree:

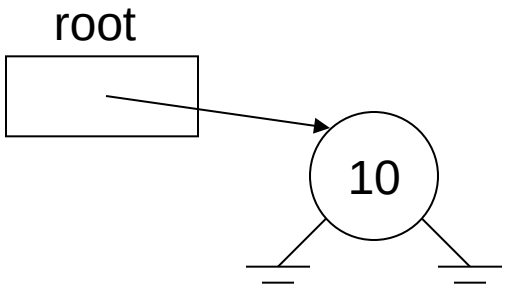


# Insertion (1/2)

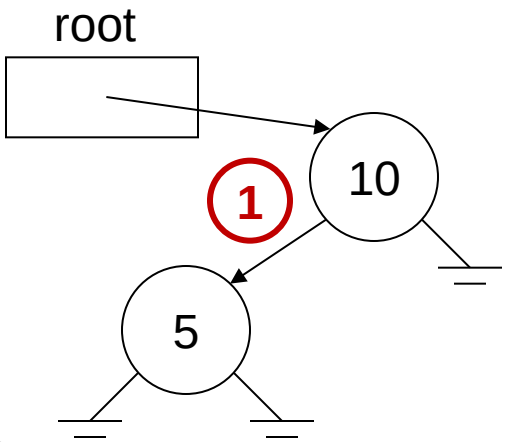
- Initially empty tree




- insert(10)

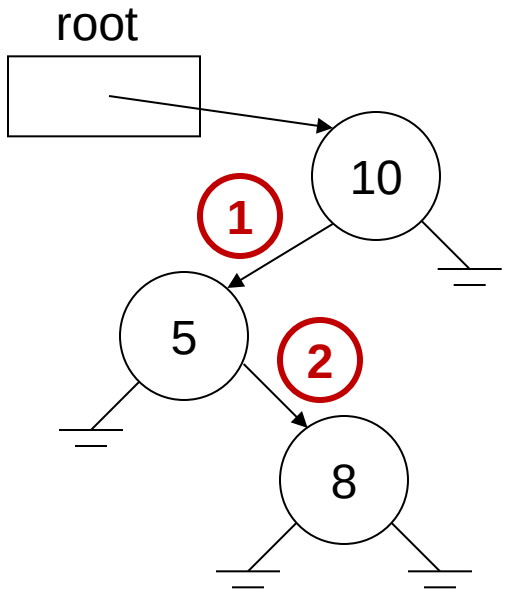



- insert(5)

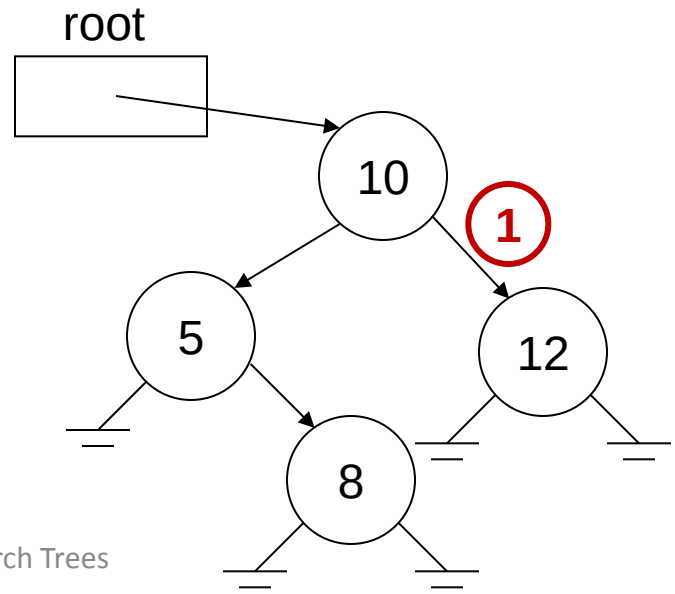


# Insertion (2/2)

- insert(8)




- insert(12)



# Implementation

- Definition for node:

```
struct Node
{
   Node(){}
   Node(item newData)
   { data=newData; lson=NULL; rson=NULL; }

   item data;
   Node* lson;
   Node* rson;
};
```

```
class BST
{
public:
   BST();
   BST(const BST& rhs);
   ~BST();
   const BST& operator = (const BST& rhs);

   bool search(item x) const;
   bool insert(item x);    // return false for duplication
   bool remove(item x);    // return false if not found
   void display() const;  // for debugging

private:
   ... Definition of struct Node placed here ...
   // other supporting functions to be specified
   Node *root;
};
```

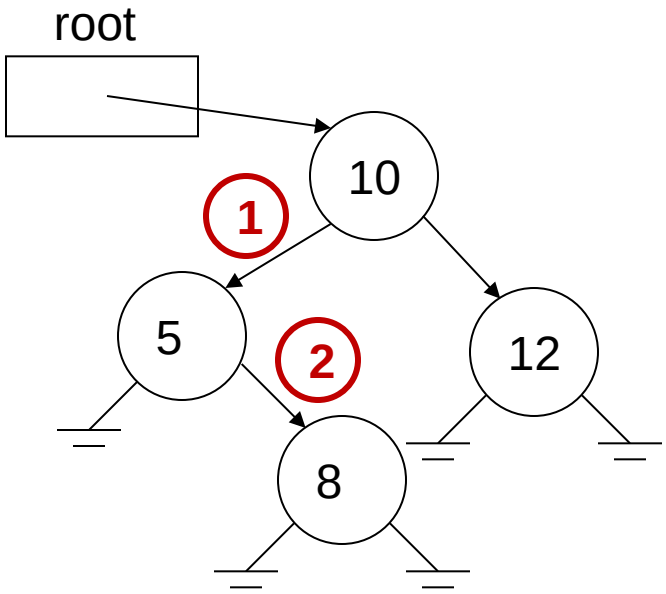
```
BST::BST(){       // default constructor
   root = NULL;  // to create an empty tree
}

BST::BST(const BST& rhs){  // copy constructor
   root = NULL;
   *this = rhs;  // to invoke the assignment operator
}

BST::~BST(){   // destructor
   deleteAll(root);
}

const BST& BST::operator=(const BST& rhs)
{
   if (this==&rhs) // avoid self-copying
      return *this;
   deleteAll(root);
   copyAll(root, rhs.root);
   return *this;
}
```

```
// To delete all nodes in the subtree pointed to by t
void BST::deleteAll(Node* t)
{
   if (t!=NULL)
   {  deleteAll(t->lson);
      deleteAll(t->rson);
      delete t;
   }
}

// To make a copy of the subtree pointed to by s and
// have t pointing to that copy
void BST::copyAll(Node*& t, Node* s)
{
   if (s!=NULL)
   {  t = new Node(s->data);
      copyAll(t->lson, s->lson);
      copyAll(t->rson, s->rson);
   }
}
```

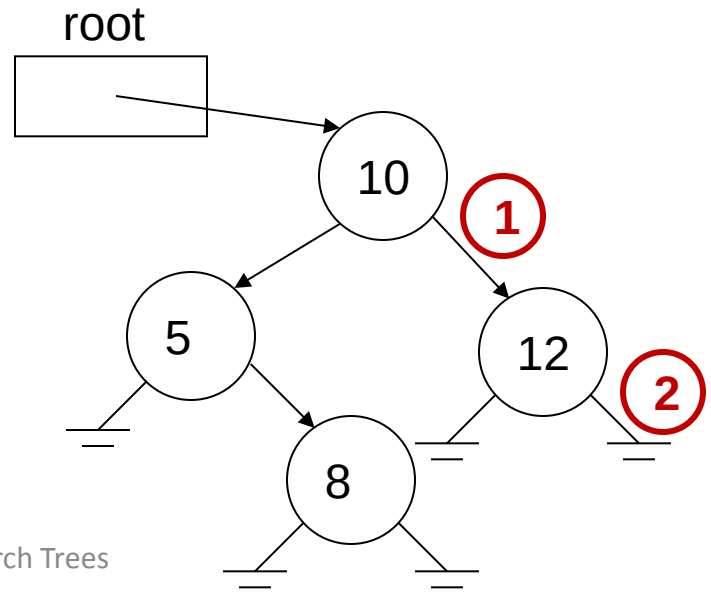
```
bool BST::insert(item x)
{  return insertR(root, x); }

// To insert x into the subtree pointed to by t
bool BST::insertR(Node*& t, item x)
{
  if (t==NULL)         // insert x into empty subtree
  {
    t = new Node(x);
    return true;
  }
  else if (x < t->data)      // recursively insert x
    return insertR(t->lson, x); // into left subtree
  else if (x > t->data)
    return insertR(t->rson, x);
  else
    return false;        // duplication, no operation
}
```

# Search

- search(8)
  - Result: 8 is in BST



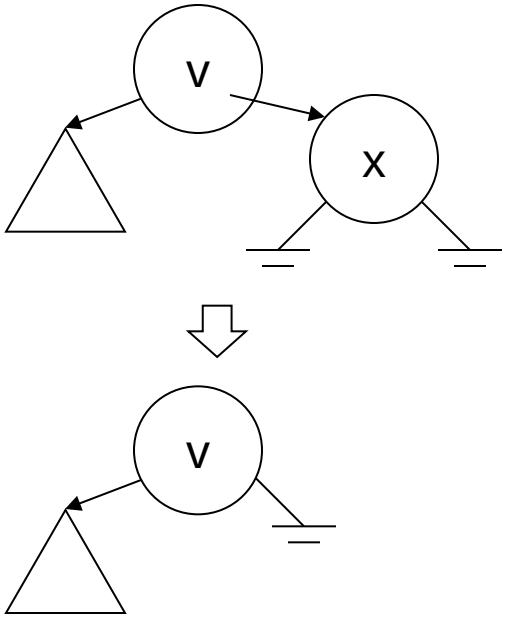

- search(13)
  - Result: 13 is not in BST



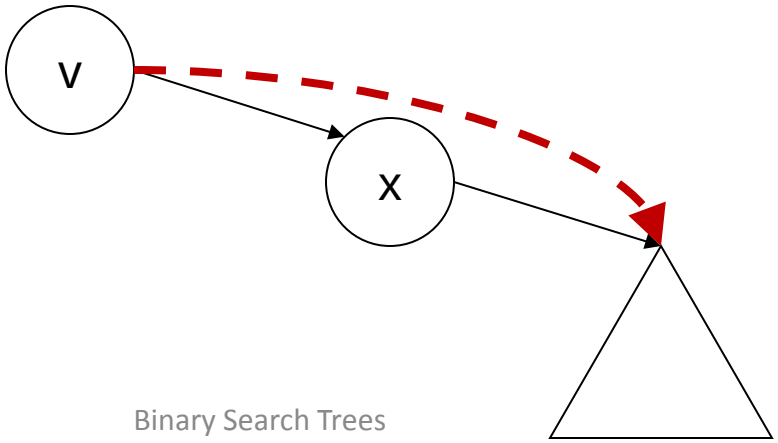
# Implementation

```
// To search for node storing x
bool BST::search(item x) const
{
   bool found=false;
   Node *t = root;
   while (!found && t!=NULL)
      if (x == t->data) found=true;
      else if (x < t->data) t=t->lson;
      else  t=t->rson;
   return found;
}
```

# Remove(x) (1/2)

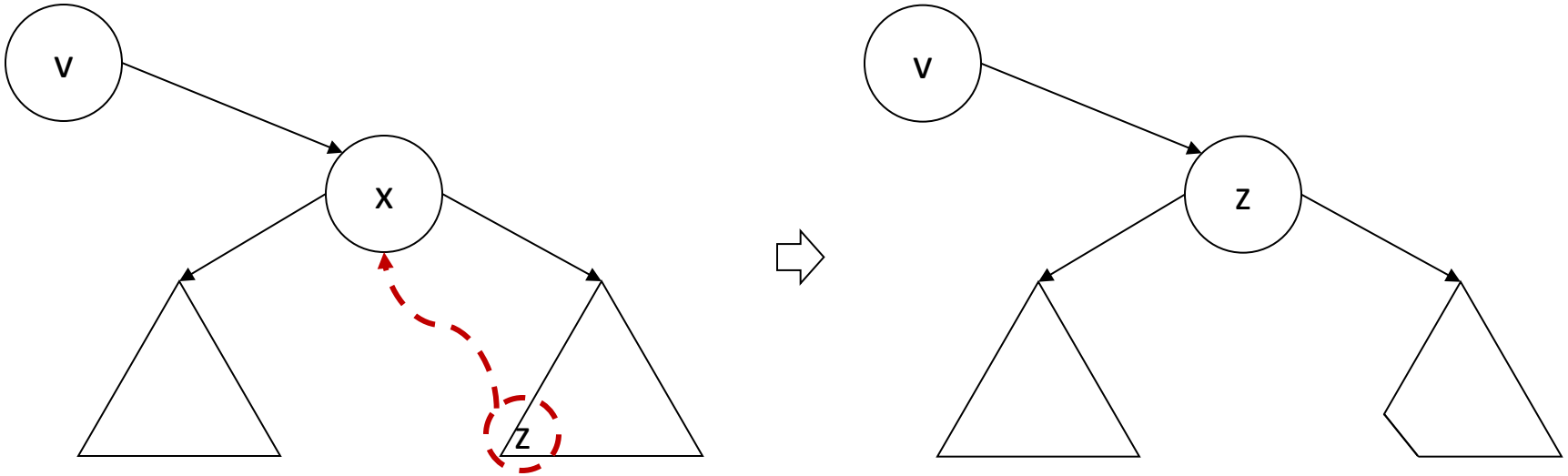



- **Case 1: x is a leaf (no son)**
  - Set pointer to x to NULL
- **Case 2: x has only one son**
  - Set parent of x, i.e., v, to link to x's son



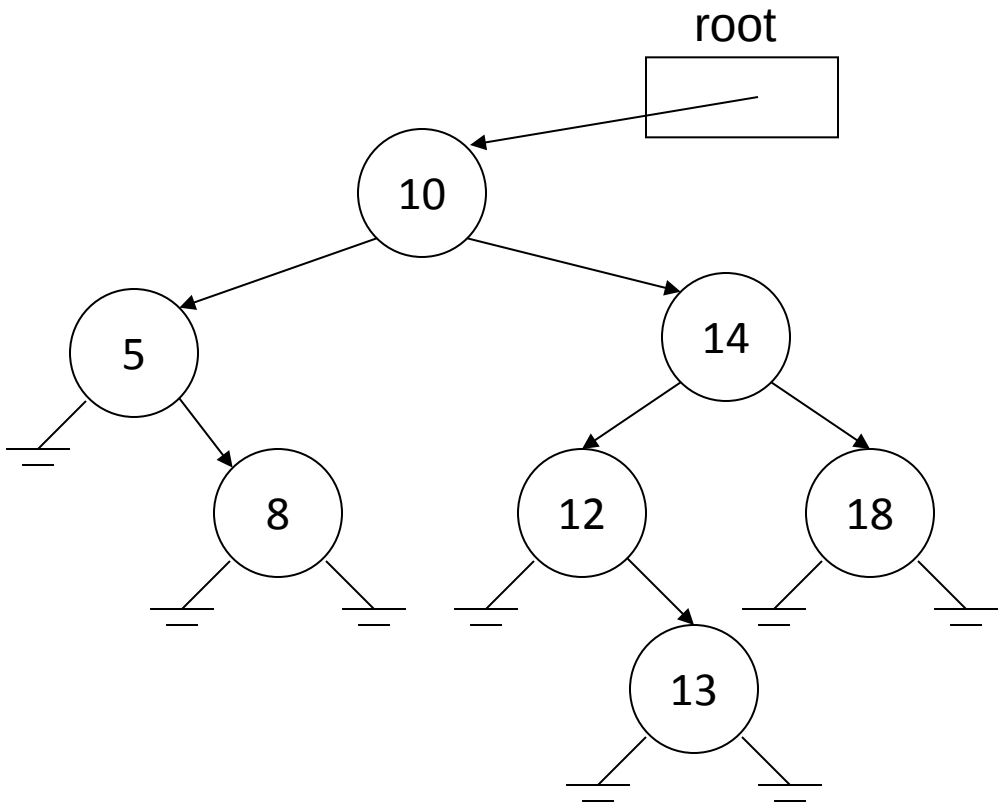
# Remove(x) (2/2)

- **Case 3: x has two sons**
  - Step 1: Remove the minimum element z of the right subtree (i.e., Case 1 or 2)
  - Step 2: Replace x by z



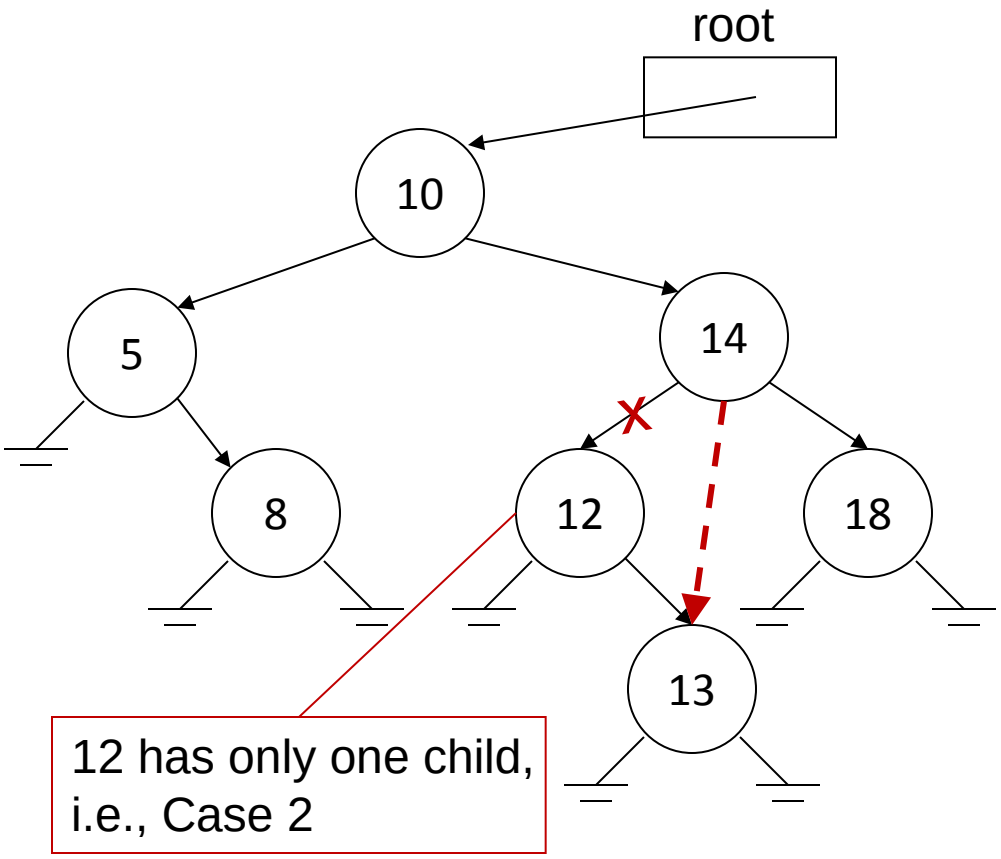
# Example: Remove(10) (1/4)

- Initial tree:



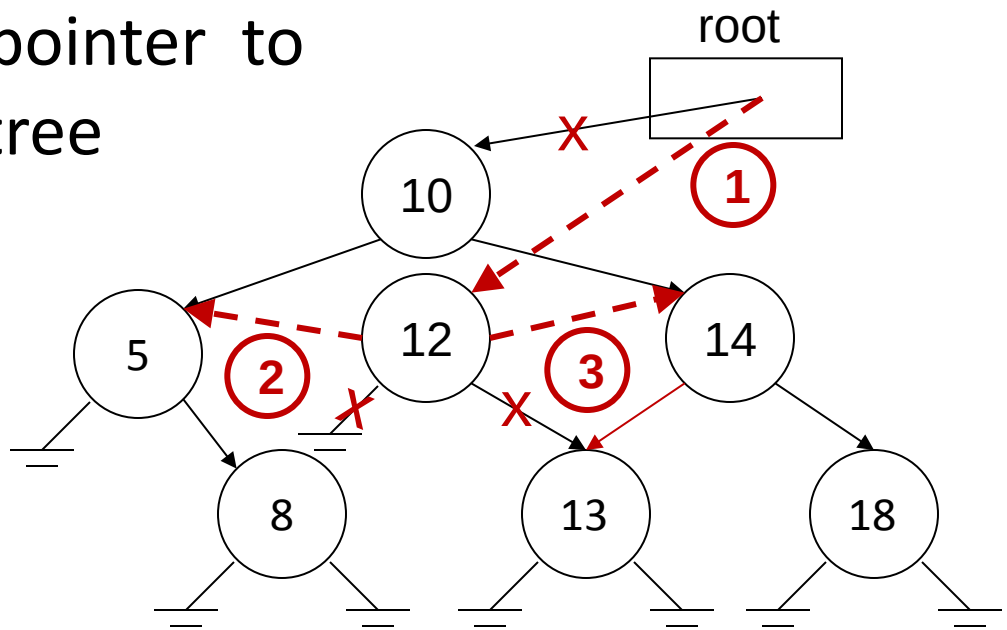
# Example: Remove(10) (2/4)

- removeMin of subtree rooted at 14 (Case 2)



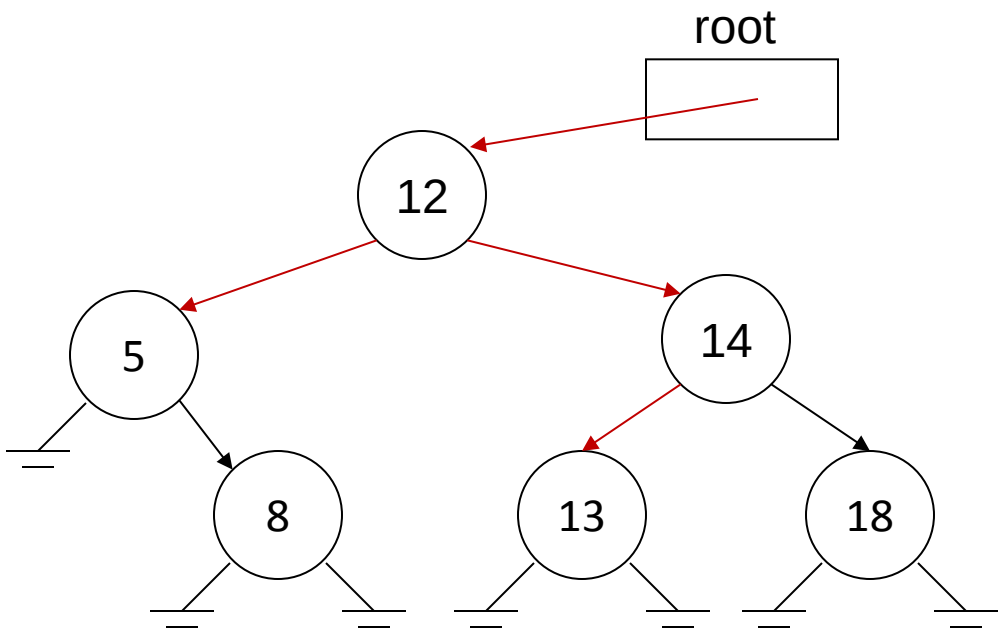
# Example: Remove(10) (3/4)

- Link 12 in place of 10
  1. Set pointer to 10 to 12
  2. Set 12's left pointer to 10's left subtree
  3. Set 12's right pointer to 10's right subtree



# Example: Remove(10) (4/4)

- Done



# Implementation

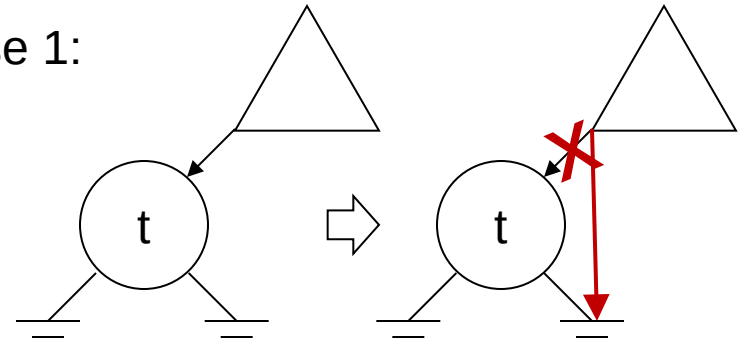
```
bool BST::remove(item x)
{  return removeR(root,x);  }

bool BST::removeR(Node*& t, item x)
{
   if (t==NULL)                        // x not found
      return false;
   else if (x < t->data)
      return removeR(t->lson, x);  // left subtree
   else if (x > t->data)
      return removeR(t->rson, x);  // right subtree
```

```
    else        // x is found
    {  Node* p = t;
       if (t->lson==NULL && t->rson==NULL) // t is leaf
          t=NULL;
       else if (t->lson==NULL) // t has rson only
          t=t->rson;
       else if (t->rson==NULL) // t has lson only
          t=t->lson;
       else {                          // t has both sons
          t=removeMin(t->rson);  1
          t->lson=p->lson;       2
          t->rson=p->rson;       3
       }
       delete p;          // delete old node
       return true;
    }
}
```

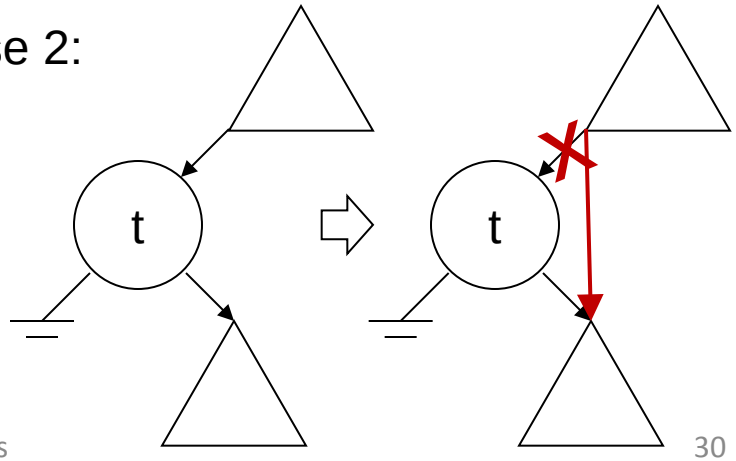
The 3 steps in Slide #23

```
// To remove min from subtree pointed to by t
// precondition: t is not NULL
BST::Node* BST::removeMin(Node*& t)
{
   if (t->lson==NULL) // t is a leaf or has one son
   {
      Node* p=t;
      t=t->rson; // See examples below
      return p;
   }
   else return removeMin(t->lson);
}
```

Case 1:




Case 2:



# Time Complexities

- Consider search(x)
  - Constant time to go down 1 level (or edge); so total time O(d) where d=depth of node storing x
  - Intuitively, good if tree is short & fat; bad if tree is long & thin
  - Worst case: O(n)
- Same for insert(x), remove(x)
  - Worst case: O(n)
  - E.g., insert a sequence 1, 2, 3, 4, and 5 into an empty binary search tree