CPSC 259: Data Structures and Algorithms for Electrical Engineers

Binary Trees

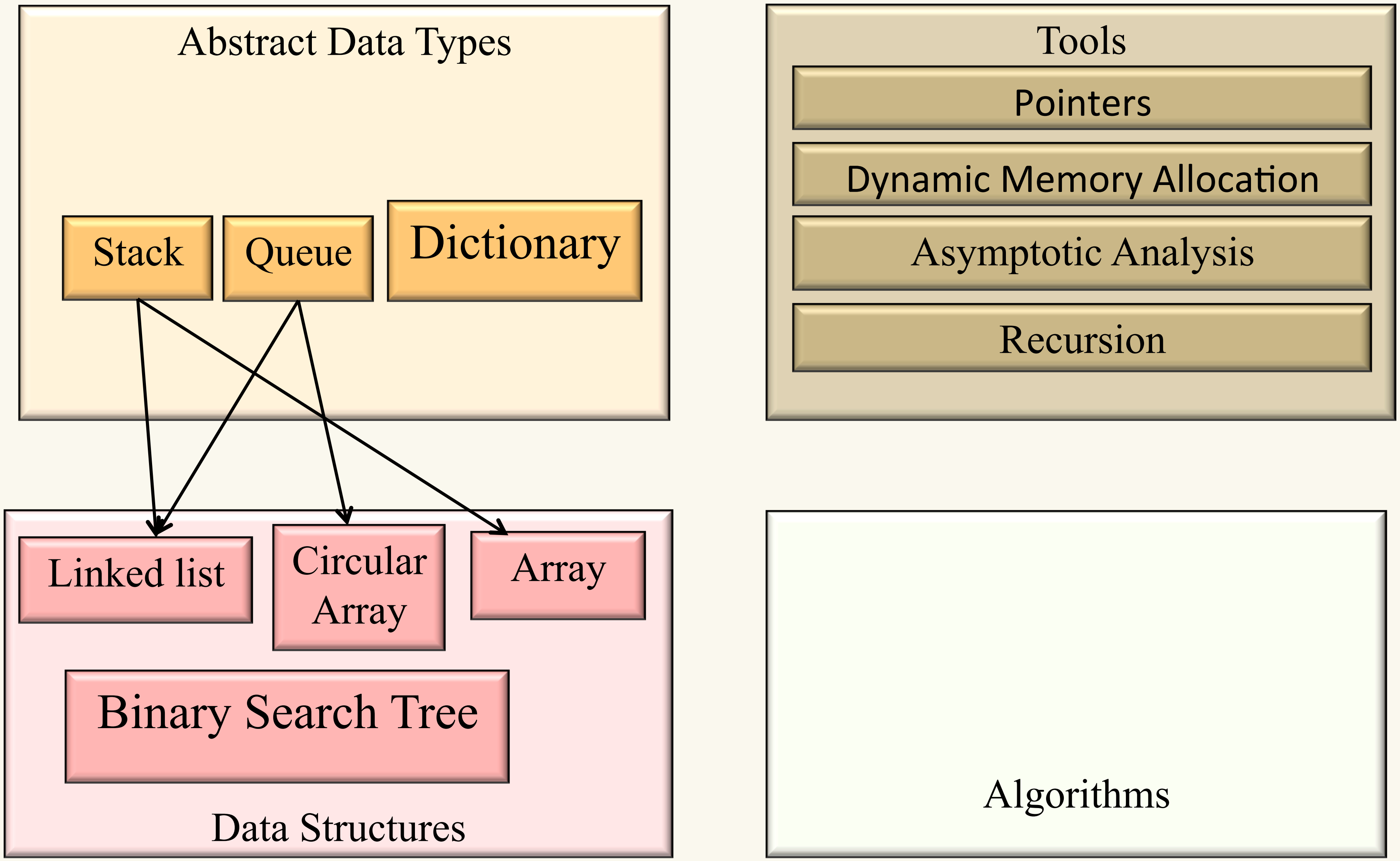
Thareja (1st edition): Chapter 10, pages 406-417
Thareja (1st edition): Chapter 11, pages 428-447
Thareja (2nd edition): Chapter 9, pages 279- 290
Thareja (2nd edition): Chapter 10, pages 298- 311

Hassan Khosravi

Learning goals

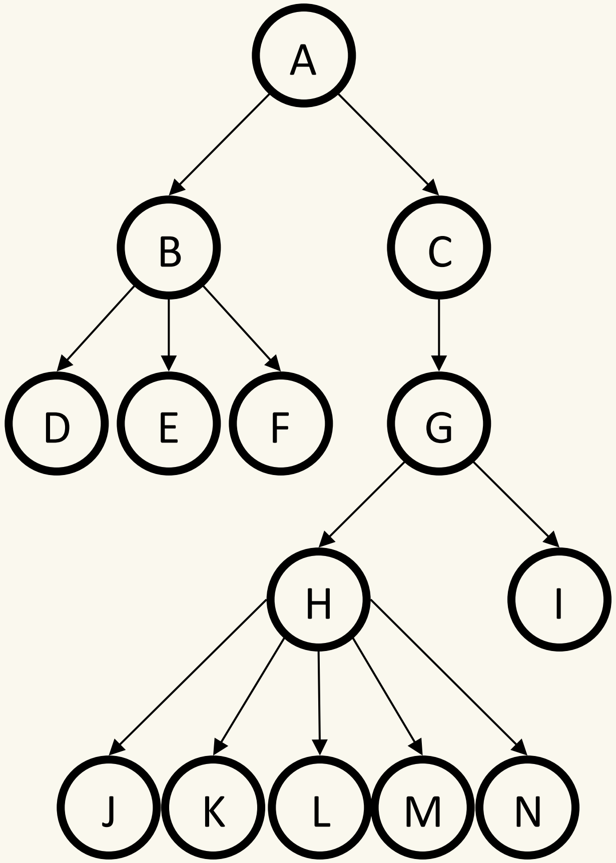
- Provide examples of the types of problems for which tree data structures are well-suited.
- Describe and use preorder, inorder, and postorder tree traversal algorithms.
- Perform a binary search on an array iteratively and recursively.
- Describe the properties of a binary search tree.
- Determine if a given tree is an instance of a binary search tree.
- Search for keys in a binary search tree.
- Insert and delete keys from a binary search tree.
- Describe the properties of binary trees and binary search trees; and algorithms for navigating (e.g., searching, adding, deleting) them in C.

CPSC 259 Journey



Tree Terminology

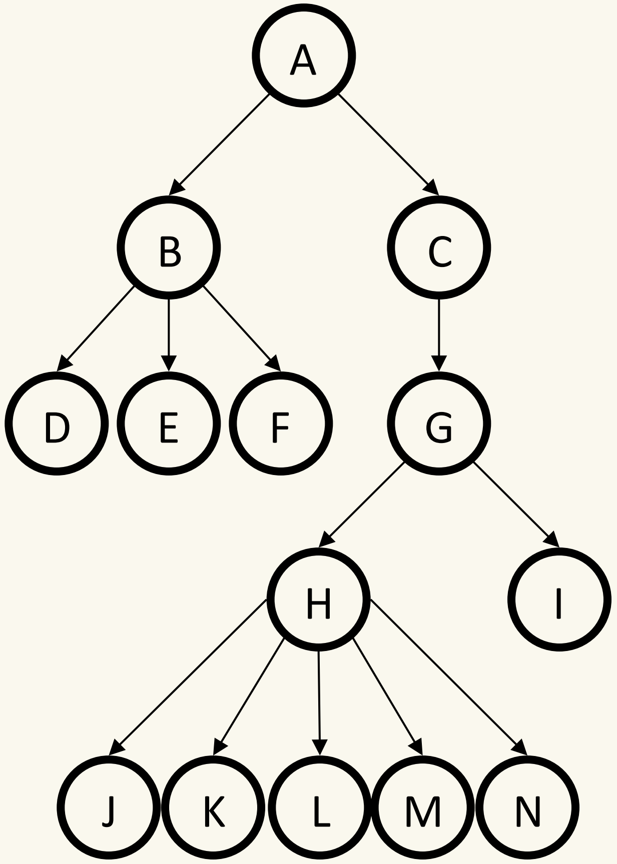
- *root:* the single node with no parent
- *leaf:* a node with no children
- *child:* a node pointed to by me
- *parent:* the node that points to me
- *Sibling:* another child of my parent
- *ancestor:* my parent or my parent's ancestor



- *descendent:* my child or my child's descendent
- *subtree:* a node and its descendants

Tree Terminology

- *depth:* # of edges along path from root to node
  - *depth of H?*
    - *3*
- height: # of edges along longest path from node to leaf or, for whole tree, from root to leaf
  - height of tree?
    - 4

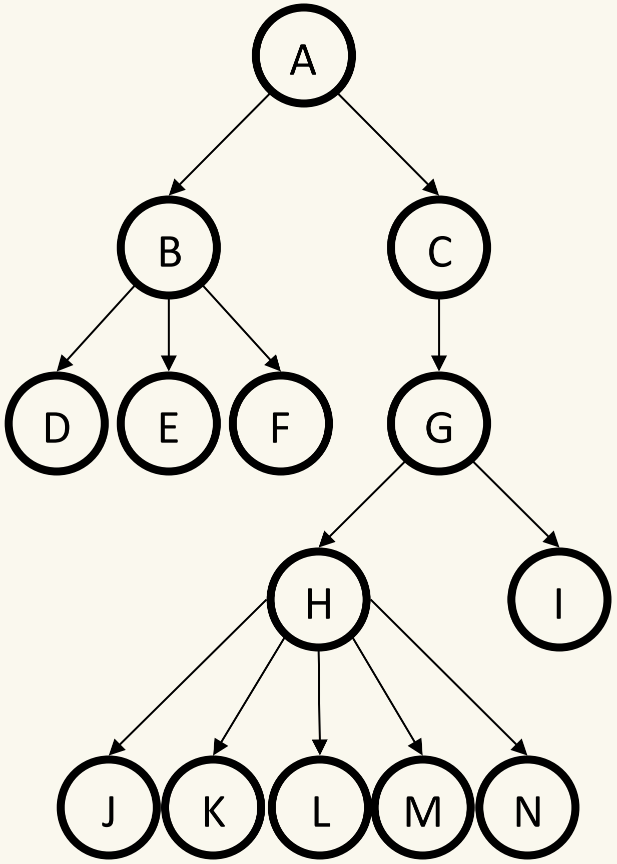


Tree Terminology

- *degree:* # of children of a node
  - *degree of B?*
    - *3*

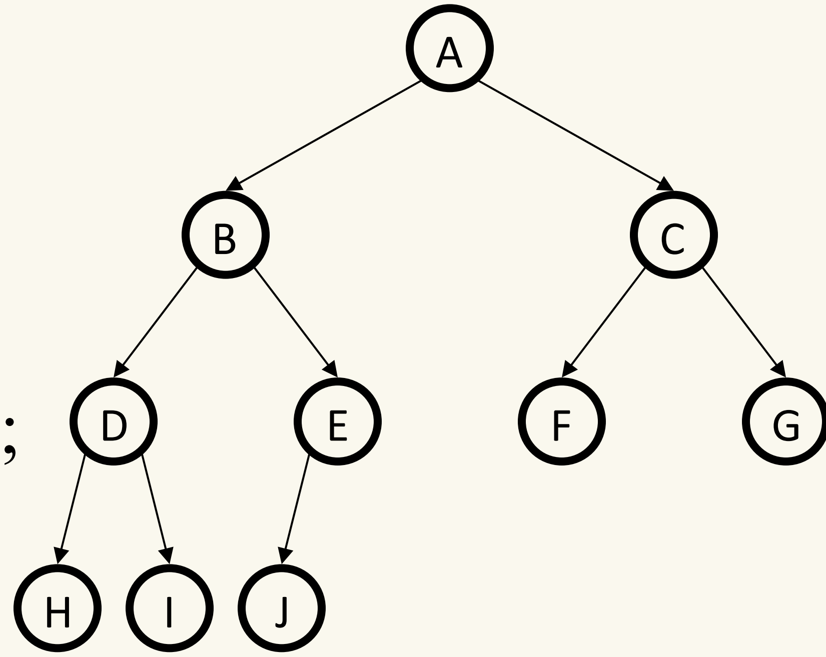
- branching factor: maximum degree of any node in the tree

2 for binary trees,
5 for this weird tree



One More Tree Terminology Slide

- binary: branching factor of 2 (each child has at most 2 children)
- n-ary: branching factor of n
- complete: "packed" binary tree;
  as many nodes as
  possible for its height
- nearly complete: complete plus some nodes on the left at the bottom



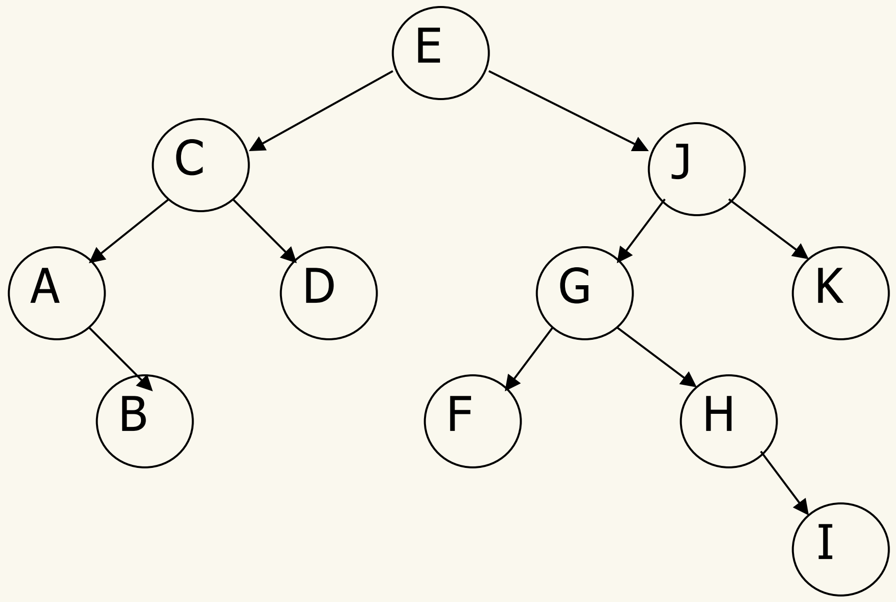
Trees and (Structural) Recursion

A tree is either:

- the empty tree
- a root node and an ordered list of subtrees

Trees are a recursively defined structure, so it makes sense to operate on them recursively.

Example


$$\begin{tikzcd}[column sep=small]
 & & & E \arrow[dll] \arrow[drr] & & & \\
 & C \arrow[dl] \arrow[dr] & & & & J \arrow[dl] \arrow[dr] & \\
A \arrow[dr] & & D & & G \arrow[dl] \arrow[dr] & & K \\
 & B & & F & & H \arrow[dr] & \\
 & & & & & & I
\end{tikzcd}$$

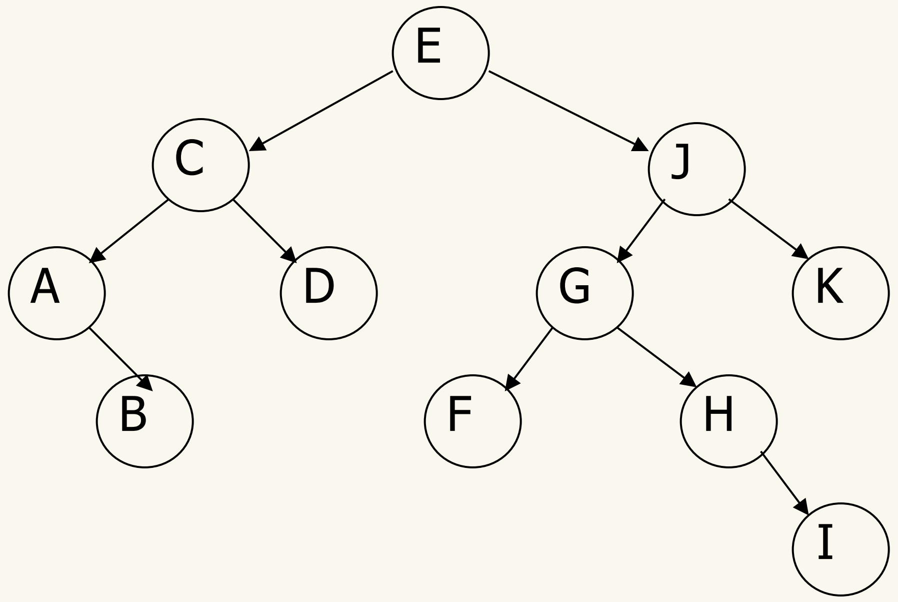
Path from E to H:

Height of tree:

Depth of node containing G:

Height of node containing G:

Example



Path from E to H: E J G H

Height of tree: 4

Depth of node containing G: 2

Height of node containing G: 2

Clicker question

- **What is the <u>height</u> of this tree?**

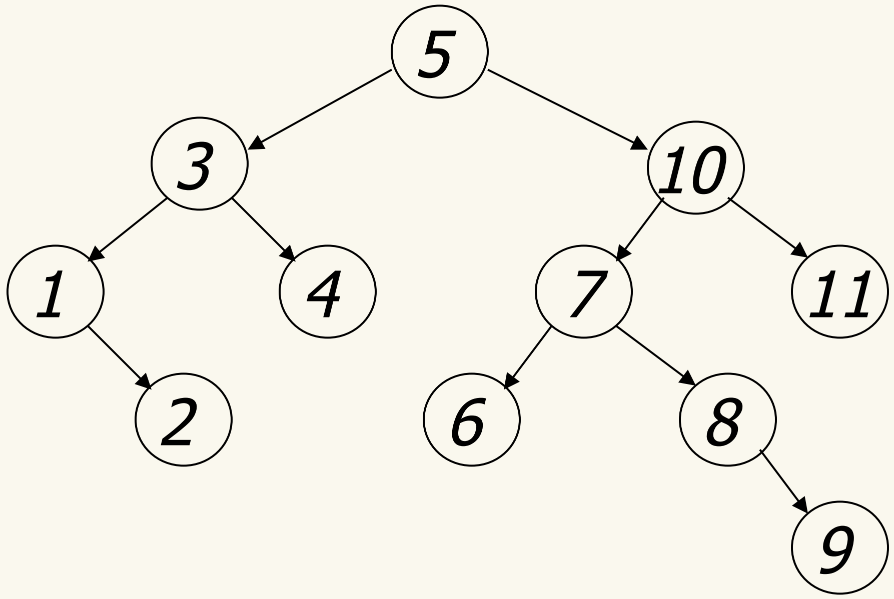
a) 0

b) 1

c) 2

d) 3

e) 4



Clicker question (answer)

- **What is the <u>height</u> of this tree?**

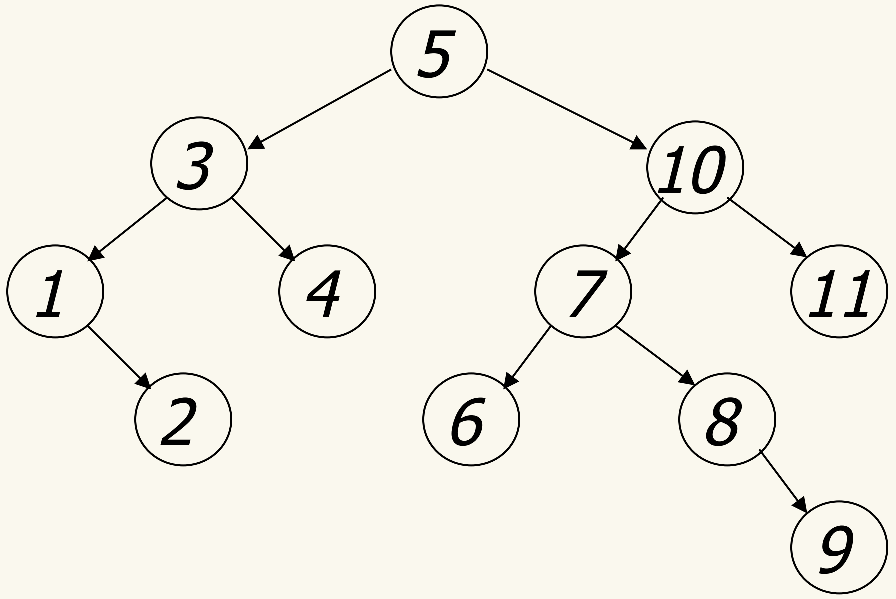
a) 0

b) 1

c) 2

d) 3

e) 4



Clicker Question

- **What is the <u>depth</u> of node 1?**

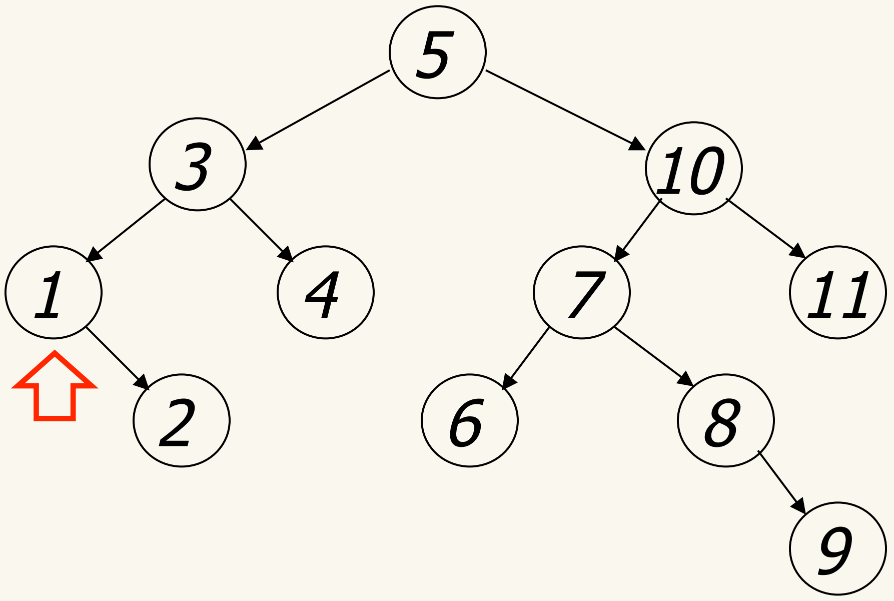
a) 0

b) 1

c) 2

d) 3

e) 4



Clicker Question (answer)

- **What is the <u>depth</u> of node 1?**

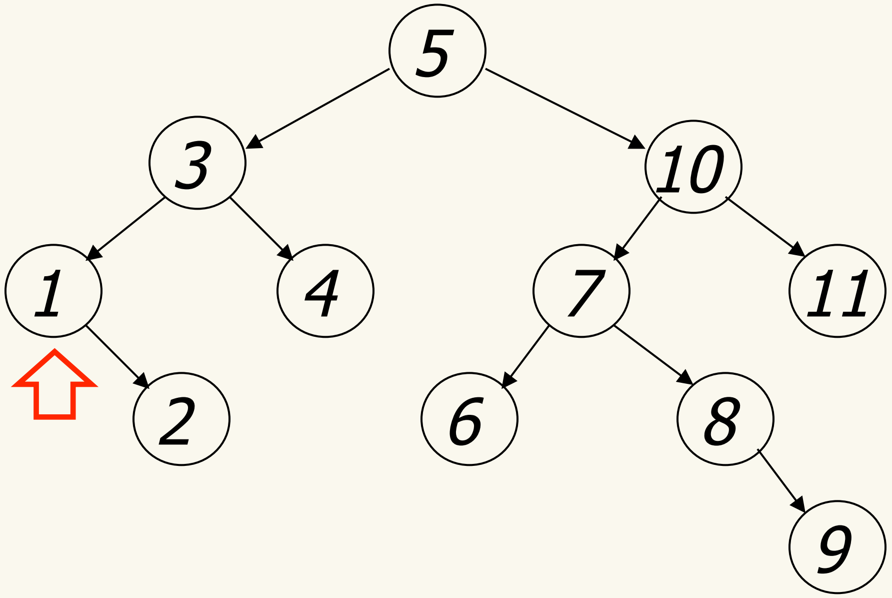
a) 0

b) 1

c) 2

d) 3

e) 4



CPSC 259 Administrative Notes

- Lab 4 – Week1 starts today
  - Recursion
- MT2: Next Friday
- Connect-Quiz and textbook exercises on Binary Search Trees are now available.
- PeerWise second call ends Friday, **November 6.**

Tree Traversal

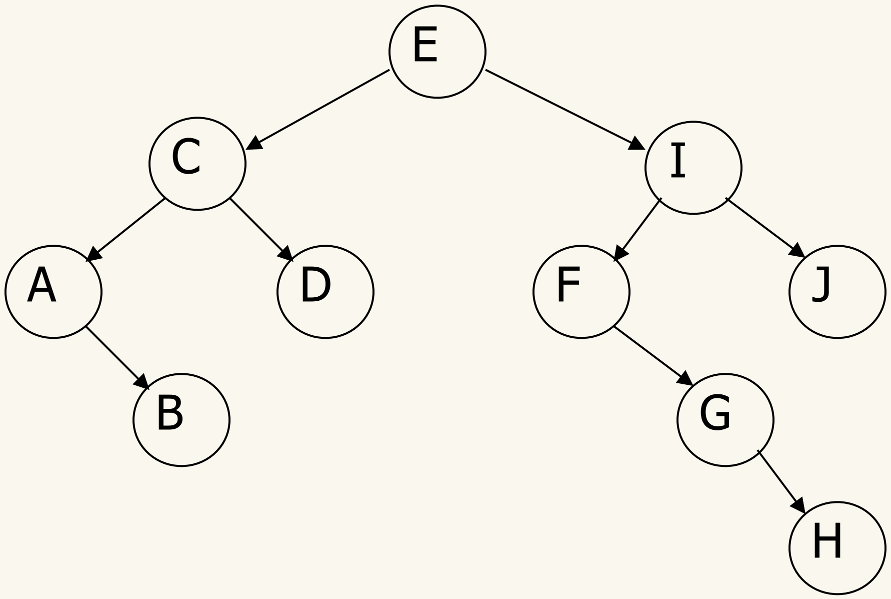
There are three common types of binary tree traversal:

Preorder: visit the current node, then its left sub-tree, then its right sub-tree

Inorder: visit the left sub-tree, then the current node, then the right sub-tree

Postorder: visit the left sub-tree, then the right sub-tree, then the current node

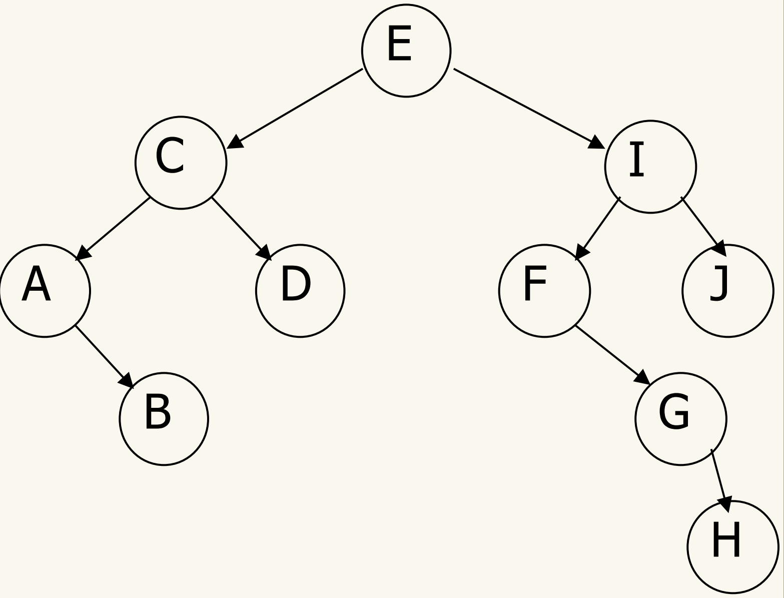
Preorder: visit the current node, then its left sub-tree, then its right sub-tree



Data printed using **preorder** traversal:

E C A B D I F G H J

Preorder: visit the current node, then its left sub-tree, then its right sub-tree

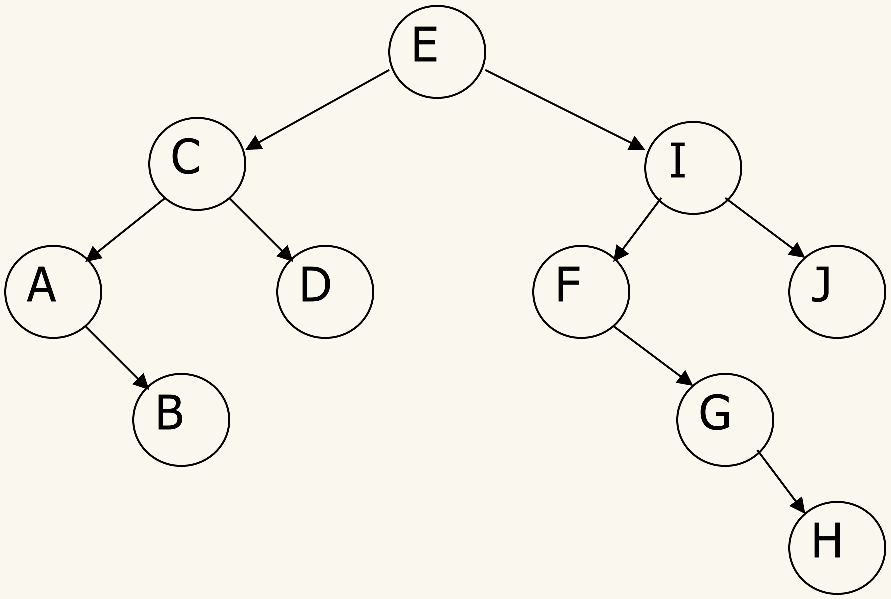


```
void printPreorder(BNode* node)
{
  if (node == NULL)
     return;

  /* first print data of node */
  printf("%d ", node->item);

  /* then recur on left sutree */
  printPreorder(node->left);

  /* now recur on right subtree */
  printPreorder(node->right);
}
```

Data printed using preorder traversal:

E C A B D I F G H J

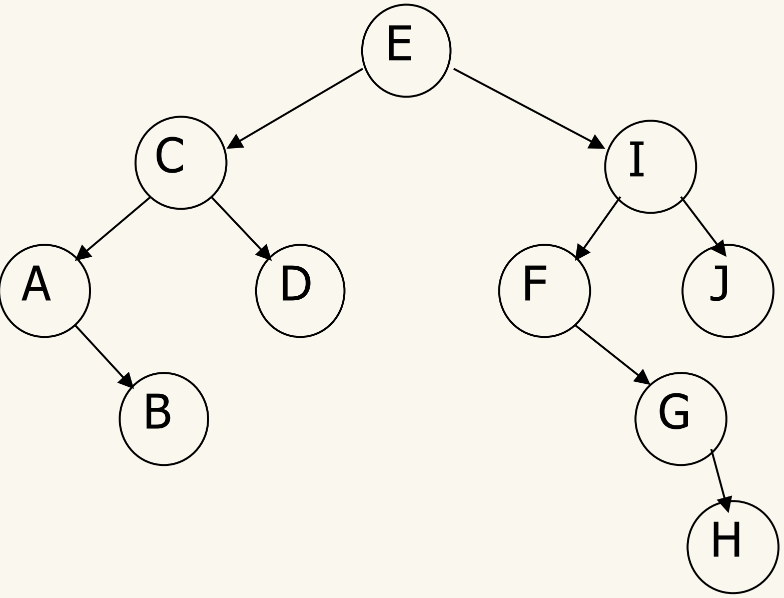
Inorder: visit the left sub-tree, then the current node, then the right sub-tree



Data printed using **inorder** traversal:

A B C D E F G H I J

Inorder: visit the left sub-tree, then the current node, then the right sub-tree

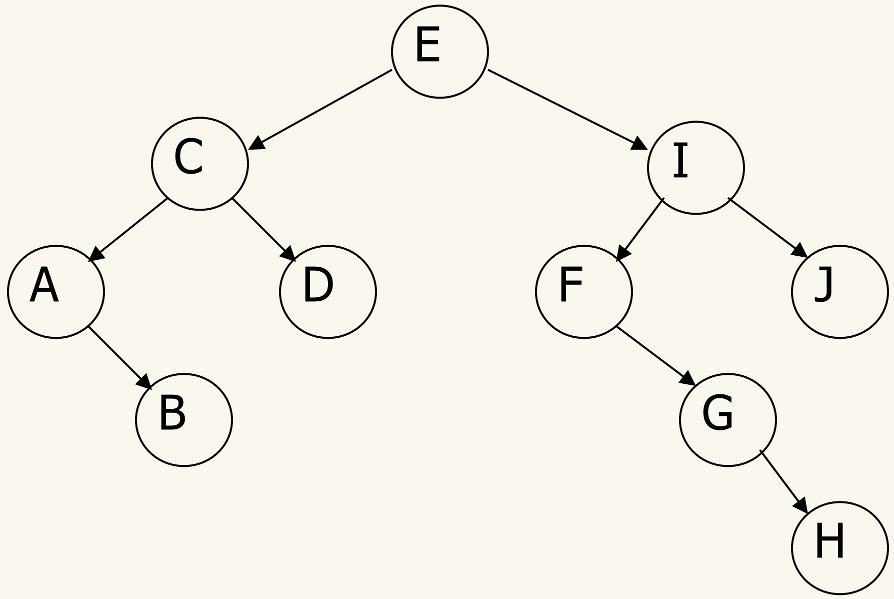


```
void printInorder(BNode* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder(node->left);

    /* then print the data of node */
    printf("%d ", node->item);

    /* now recur on right child */
    printInorder(node->right);
}
```

Data printed using inorder traversal:

A B C D E F G H I J

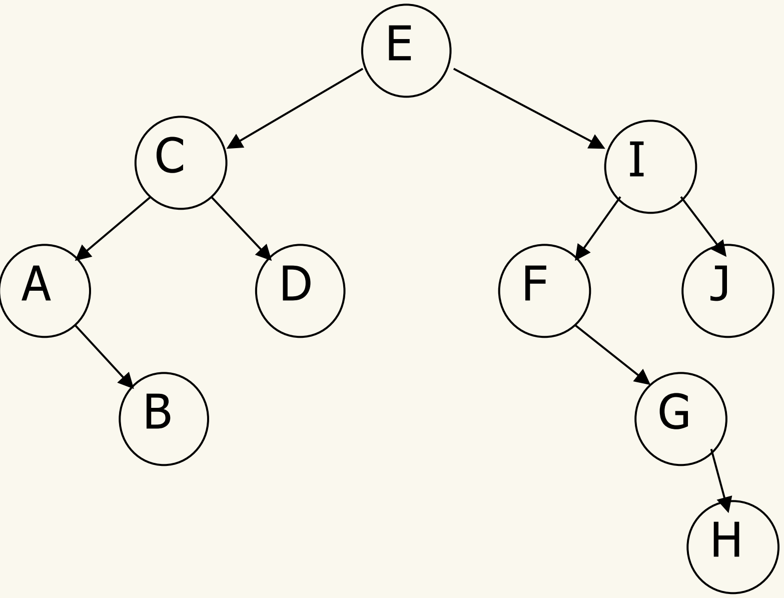
Postorder: visit the left sub-tree, then the right sub-tree, then the current node



Data printed using **postorder** traversal:

B A D C H G F J I E

Postorder: visit the left sub-tree, then the right sub-tree, then the current node

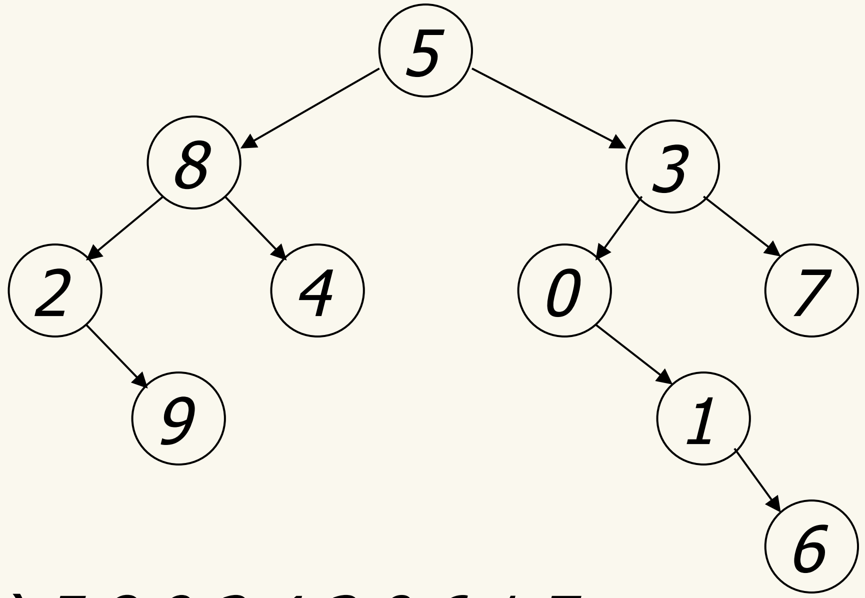


```
void printPostorder(BNode* node)
{
    if (node == NULL)
        return;

    // first recur on left subtree
    printPostorder(node->left);

    // then recur on right subtree
    printPostorder(node->right);

    // now deal with the node
    printf("%d ", node->item);
}
```

Data printed using postorder traversal:

B A D C H G F J I E

Preorder: visit the current node, then its left sub-tree, then its right sub-tree (this is NOT a Binary Search Tree!)



Nodes visited using preorder traversal:

a) 5 8 9 2 4 3 0 6 1 7

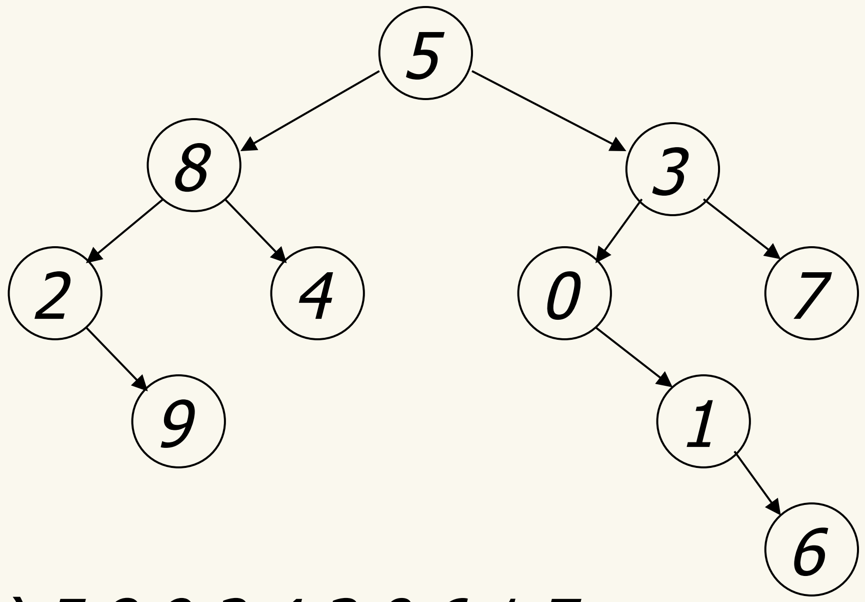
b) 5 8 2 9 4 3 0 1 6 7

c) 5 8 3 2 4 0 7 9 1 6

d) 6 1 0 7 3 5 9 2 4 8

e) 9 2 4 8 6 1 0 7 3 5

Preorder: visit the current node, then its left sub-tree, then its right sub-tree (this is NOT a Binary Search Tree!)



Nodes visited using preorder traversal:

a) 5 8 9 2 4 3 0 6 1 7

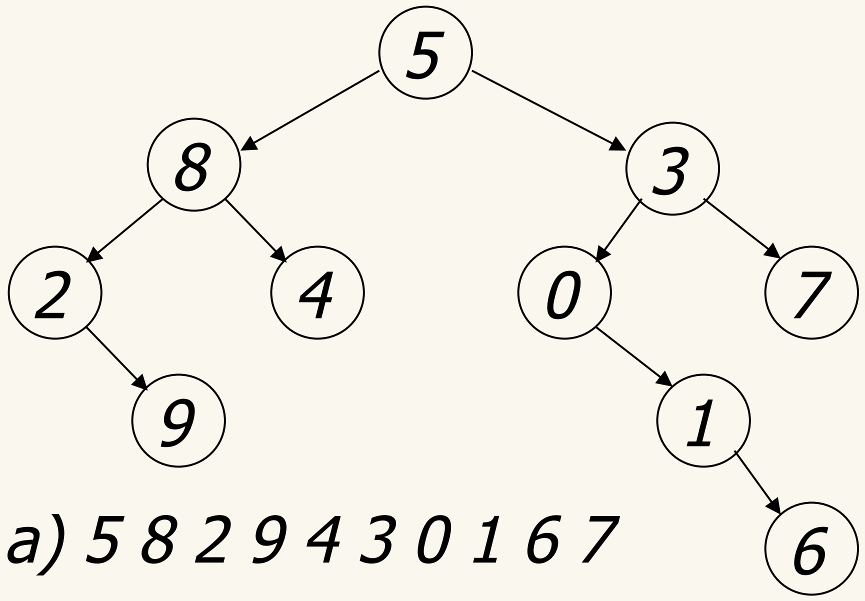
b) 5 8 2 9 4 3 0 1 6 7

c) 5 8 3 2 4 0 7 9 1 6

d) 6 1 0 7 3 5 9 2 4 8

e) 9 2 4 8 6 1 0 7 3 5

Inorder: visit the left sub-tree, then the current node, then the right sub-tree (this is NOT a BST!)



Nodes visited using inorder traversal:

a) 5 8 2 9 4 3 0 1 6 7

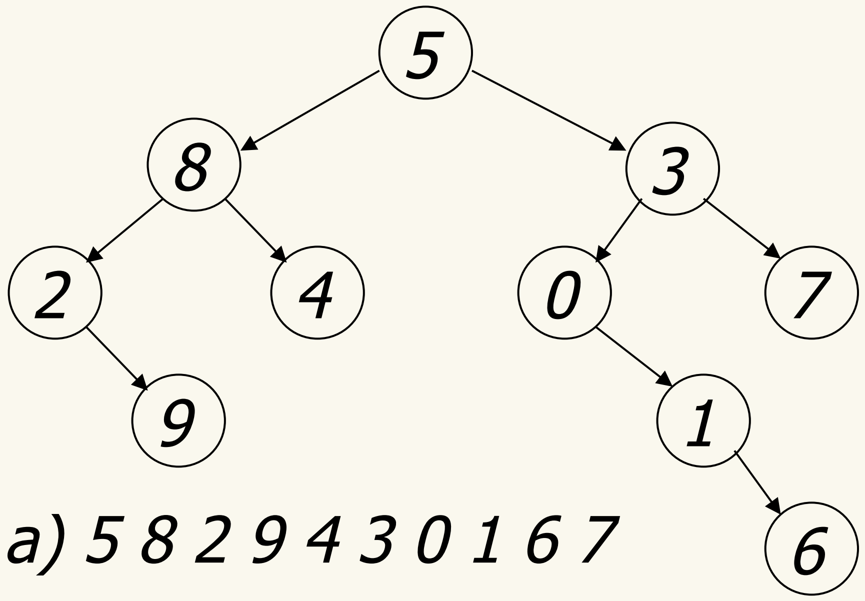
b) 5 8 3 2 4 0 7 9 1 6

c) 6 1 0 7 3 5 9 2 4 8

d) 2 9 8 4 5 0 1 6 3 7

e) 9 2 4 8 6 1 0 7 3 5

Inorder: visit the left sub-tree, then the current node, then the right sub-tree (this is NOT a BST!)



Nodes visited using inorder traversal:

a) 5 8 2 9 4 3 0 1 6 7

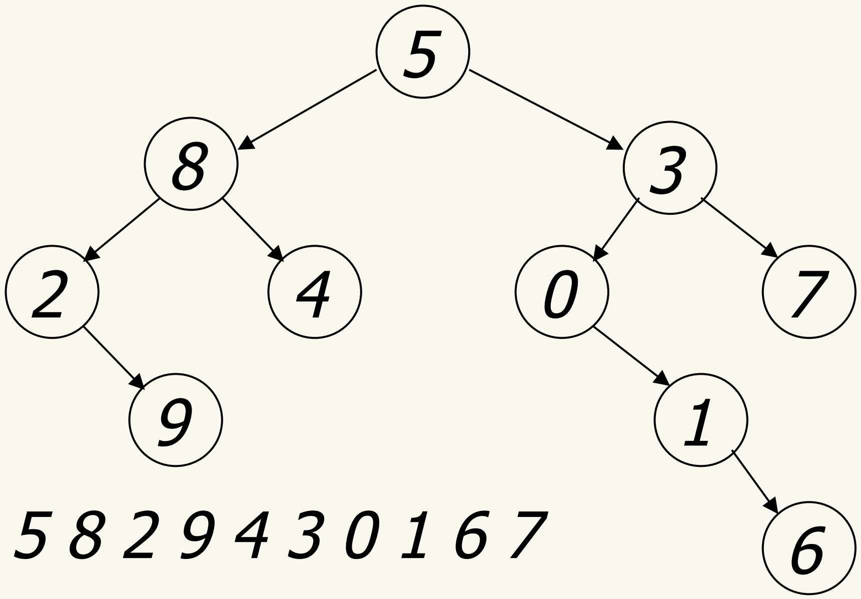
b) 5 8 3 2 4 0 7 9 1 6

c) 6 1 0 7 3 5 9 2 4 8

d) 2 9 8 4 5 0 1 6 3 7

e) 9 2 4 8 6 1 0 7 3 5

Postorder: visit the left sub-tree, then the right sub-tree, then the current node (this is NOT a BST!)



Nodes visited using postorder traversal:

a) 5 8 2 9 4 3 0 1 6 7

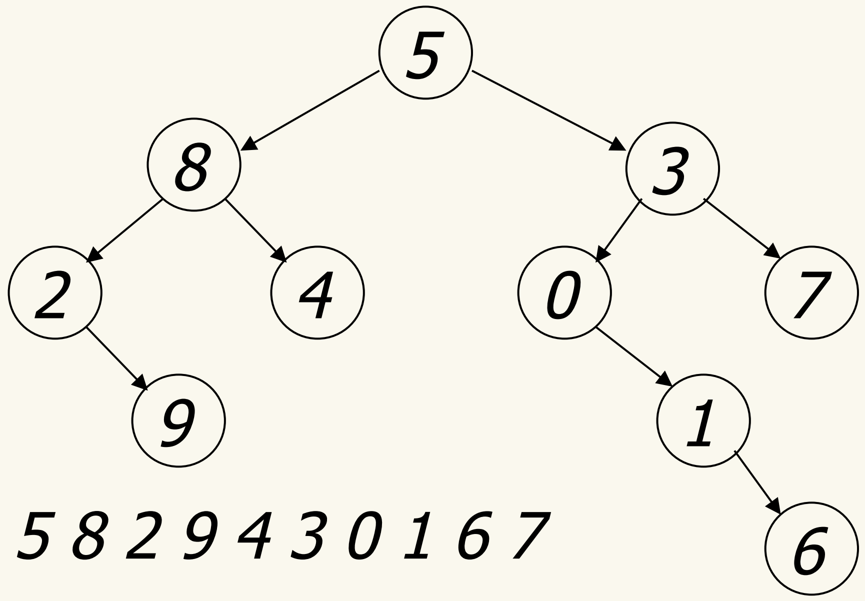
b) 2 9 8 4 0 1 6 7 3 5

c) 6 1 0 7 3 5 9 2 4 8

d) 9 2 4 8 6 1 0 7 3 5

e) 2 9 8 4 5 0 1 6 3 7

Postorder: visit the left sub-tree, then the right sub-tree, then the current node (this is NOT a BST!)



Nodes visited using postorder traversal:

a) 5 8 2 9 4 3 0 1 6 7

b) 2 9 8 4 0 1 6 7 3 5

c) 6 1 0 7 3 5 9 2 4 8

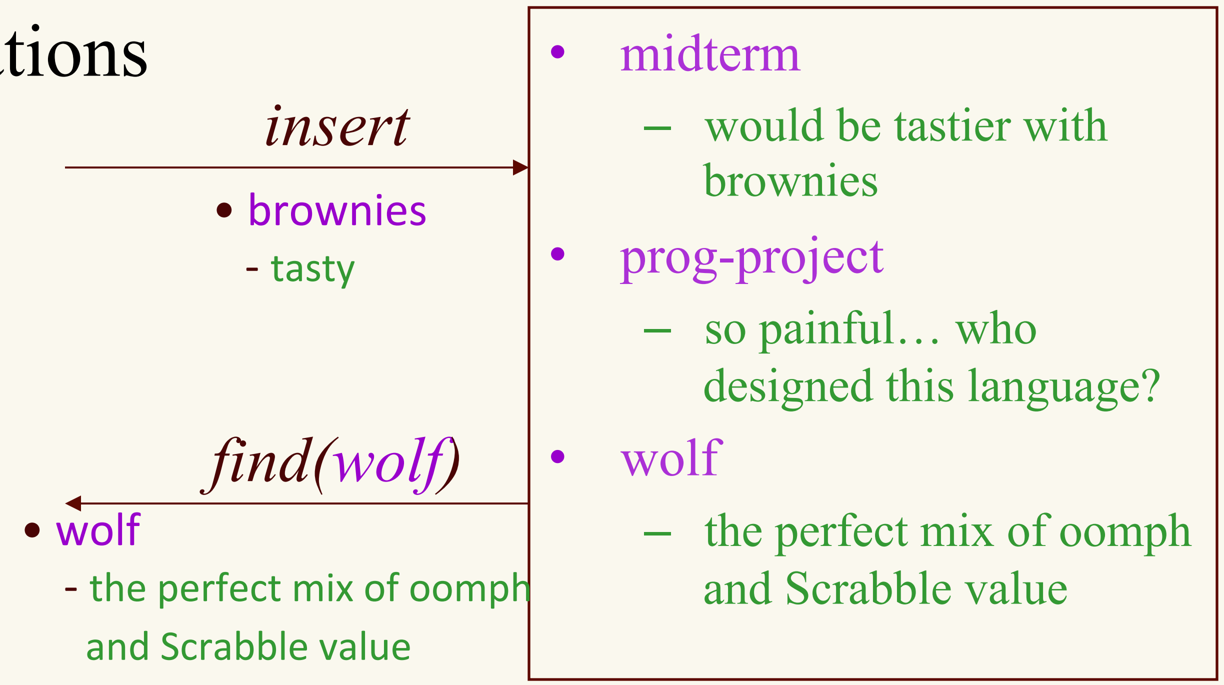
d) 9 2 4 8 6 1 0 7 3 5

e) 2 9 8 4 5 0 1 6 3 7

Tree: 5 → (8, 3); 8 → (2, 4); 2 → right 9; 3 → (0, 7); 0 → right 1; 1 → right 6

Dictionary ADT

- Dictionary operations
  - create
  - destroy
  - insert
  - find
  - Delete



- brownies
  - tasty

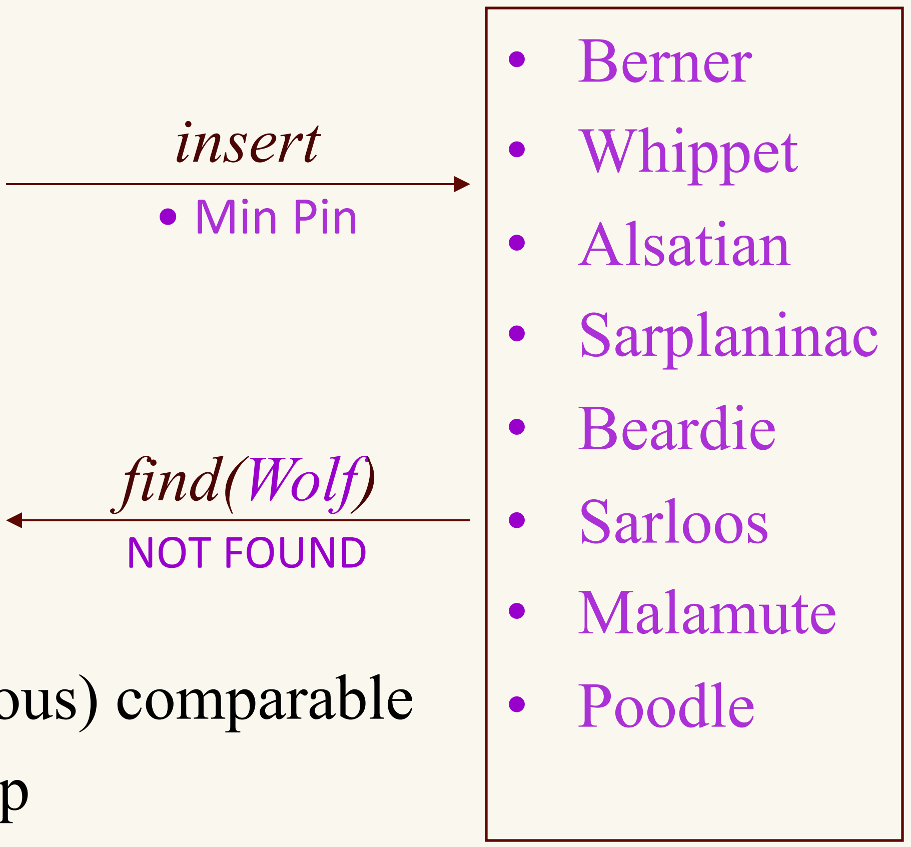
- midterm
  - would be tastier with brownies
- prog-project
  - so painful… who designed this language?
- wolf
  - the perfect mix of oomph and Scrabble value

- wolf
  - the perfect mix of oomph and Scrabble value

- Stores *values* associated with user-specified *keys*
  - values may be any (homogenous) type
  - keys may be any (homogenous) comparable type

Search/Set ADT

- Dictionary operations
  - create
  - destroy
  - insert
  - find
  - Delete



- Stores keys
  - keys may be any (homogenous) comparable
  - quickly tests for membership

A Modest Few Uses

- Arrays and “Associative” Arrays
- Sets
- Dictionaries
- Address books
- Credit card authorization
- Router tables
- Page tables
- Symbol tables
- C++ Structures

Naïve Implementations

	insert	*find*	*delete + find*	*delete after find*
• Linked list				
– Unsorted	$O(1)$	$O(n)$	$O(n)$	$O(1)$
– Sorted	$O(n)$	$O(n)$	$O(n)$	$O(1)$
• Array				
– Unsorted	$O(1)$	$O(n)$	$O(n)$	$O(1)$
– Sorted	$O(n)$	$O(\lg n)$	$O(n)$	$O(n)$

worst one… yet so close!

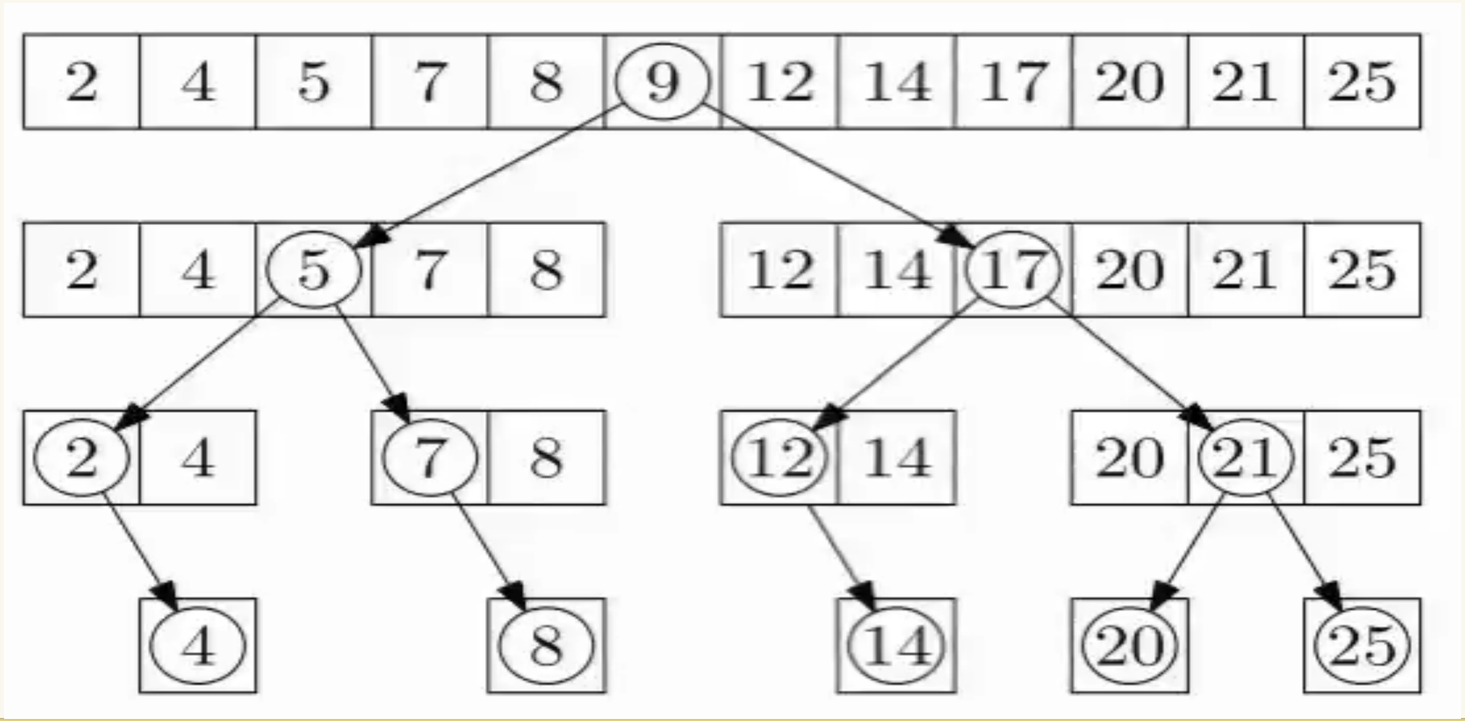
Binary Search (iterative version)

```
/* Search an array, iteratively, for a given search key */
int  search( int * array, int key, int size ){
    int  low = 0;
    int  high = size - 1;
    int  mid;

    /* Discard half of the array during each iteration. */
    while (low <= high){
        mid = (low + high) / 2;  /* middle of range */

        if (array[mid] == key)
            return mid;     /* found the search key */
        if (key < array[mid])
        /* focus on the left half of the remaining array */
            high = mid - 1;
        else
        /* focus on the right half of the remaining array */
            low = mid + 1;
    }
    return -1;     /* search key was not found */
}
```

Binary Search (recursive version)

```
/* Search an array, recursively, for a given search key. */
int  search( int * array, int key, int low_index, int high_index ){
    int mid = ( low_index + high_index ) / 2;

    if ( high_index < low_index )
        return -1;     /* key not found (base case) */

    if ( array[mid] > key )  /* search left half of array */
        return search( array, key, low_index, mid - 1 );

    else if ( array[mid] < key ) /* search right half of array */
        return search( array, key, mid + 1, high_index );

    else
        return mid; /* we found the search key */
}
```

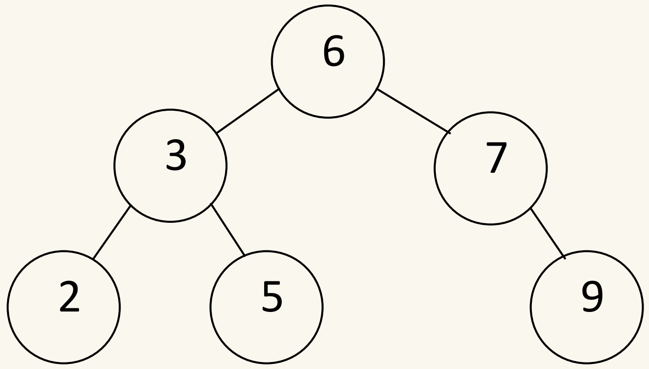
Binary Search into Binary Search Trees



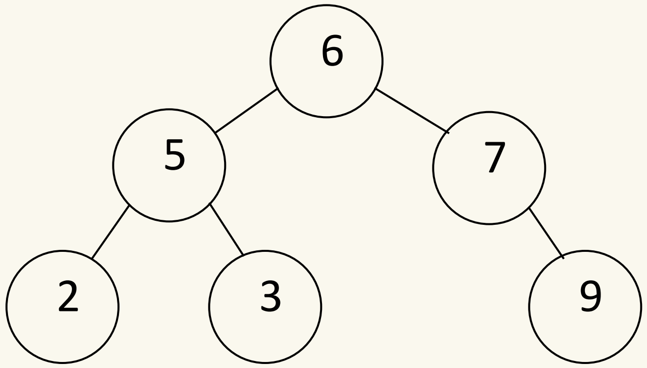
```
/* Search an array, recursively, for a given search key. */
int  search( int array[], int key, int low_index, int high_index ){
    int mid = ( low_index + high_index ) / 2;

    if ( high_index < low_index )  return -1;
    if ( array[mid] > key ) return search(array, key, low_index, mid-1);
    else if ( array[mid] < key) /* search right half of array */
        return search( array, key, mid + 1, high_index );
    else   return mid;
}
```

Binary search tree

A **binary search tree** is a binary tree such that for every node in the tree, all of the entries in the left sub-tree (if any) are smaller than the entry in the node and all of the entries in the right sub-tree (if any) are larger than the entry in the node.



a binary search tree

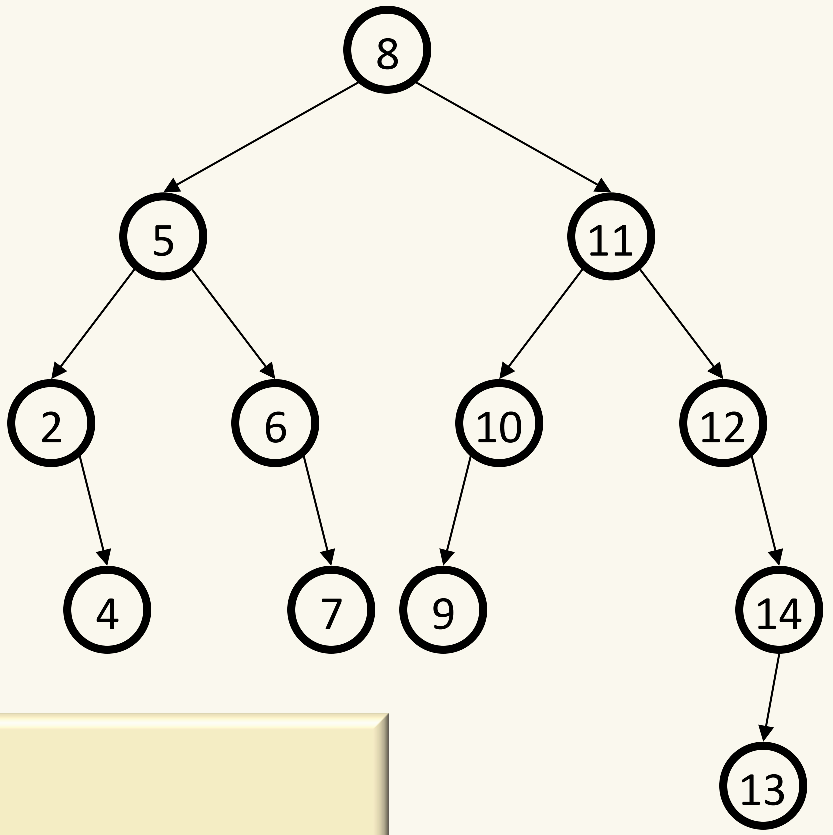


not a binary search tree

Note: there is no requirement that a binary search tree has to be a complete binary tree.

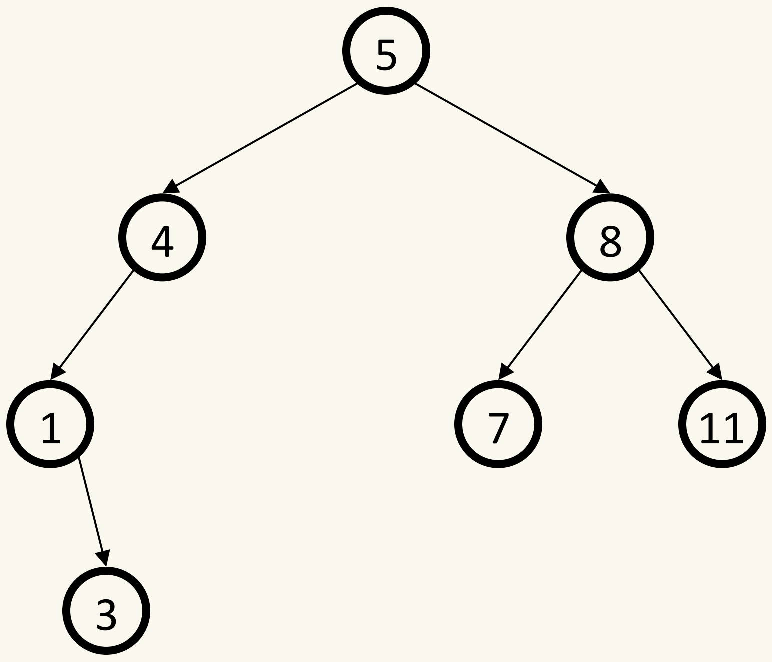
Binary Search Tree (Summary)

- Binary tree property
  - each node has ≤ 2 children
  - result:
    - operations are simple

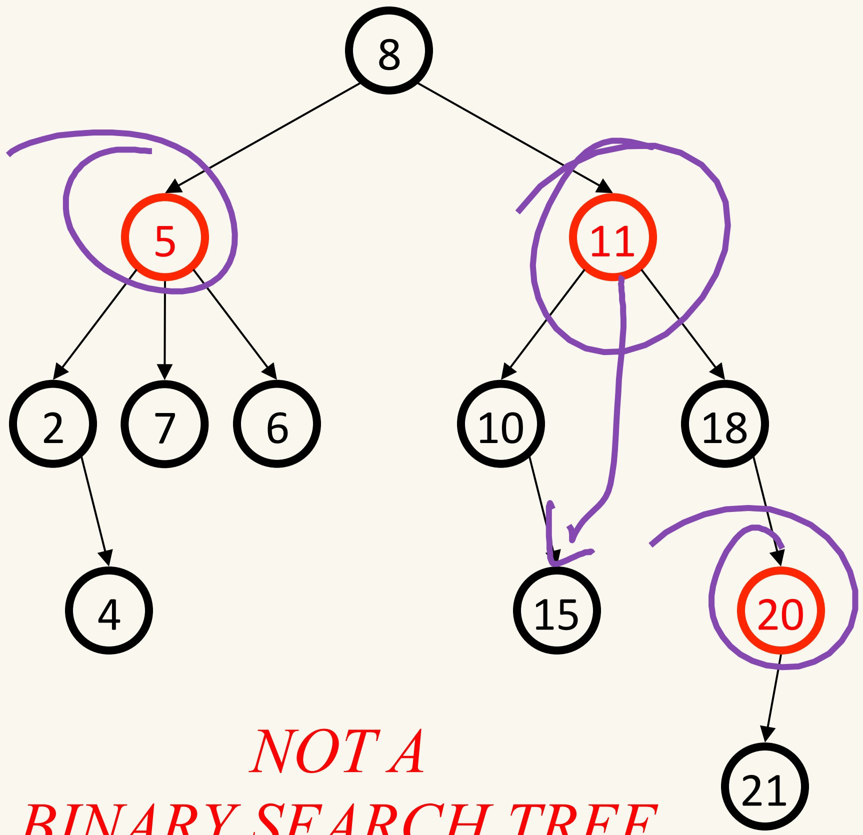


- Search tree property
  - **all** keys in left subtree smaller than root's key
  - **all** keys in right subtree larger than root's key
  - result:
    - easy to find any given key

Example and Counter-Example

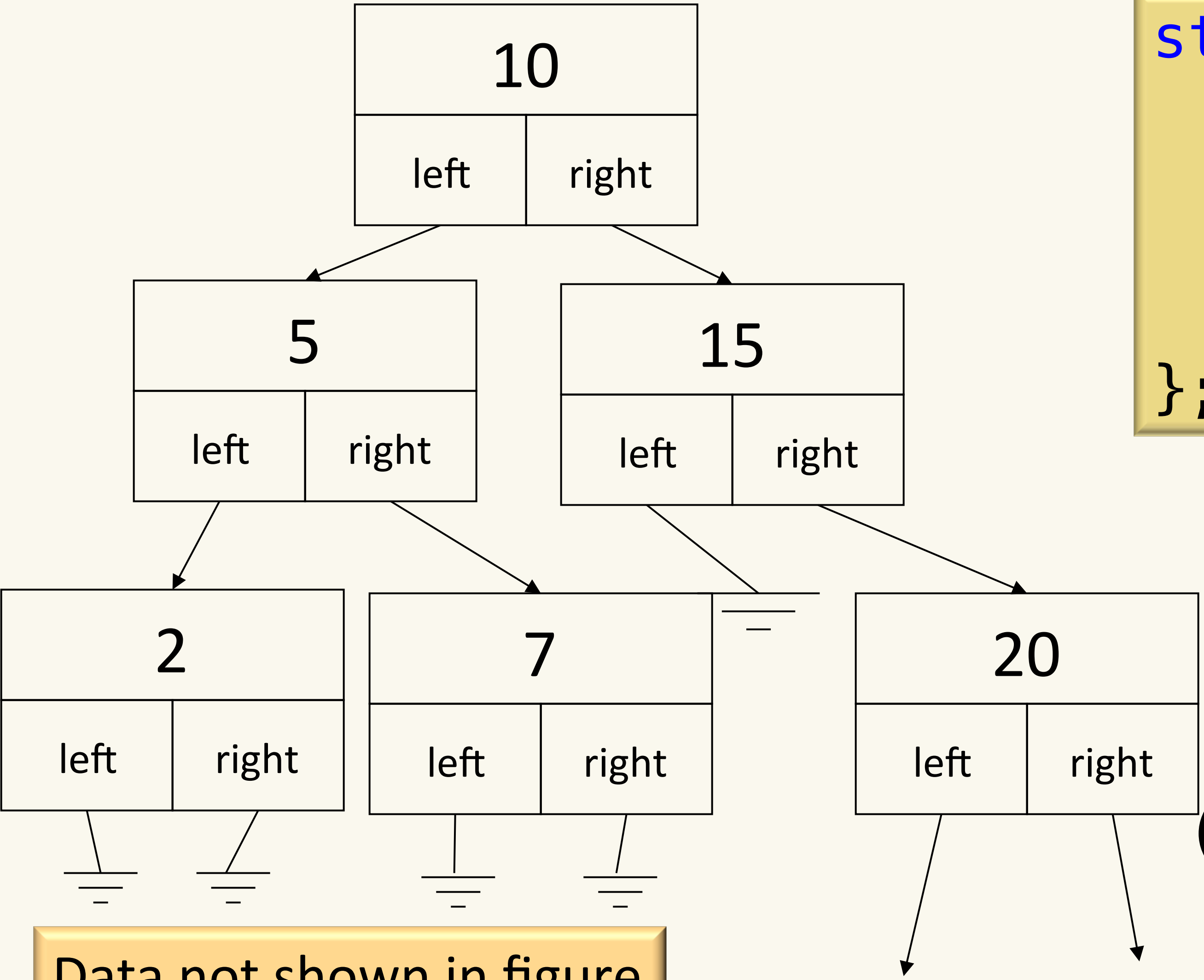


BINARY SEARCH TREE

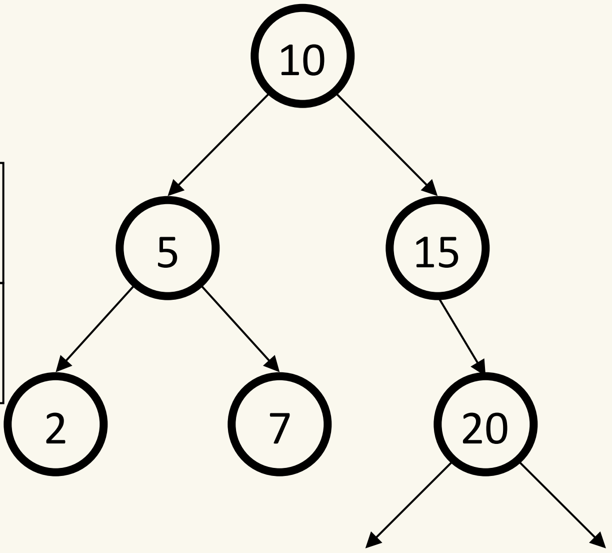


NOT A BINARY SEARCH TREE

Representing Binary Search Trees



```
struct Node {
  KTYPE key;
  DTYPE data;
  Node * left;
  Node * right;
};
```



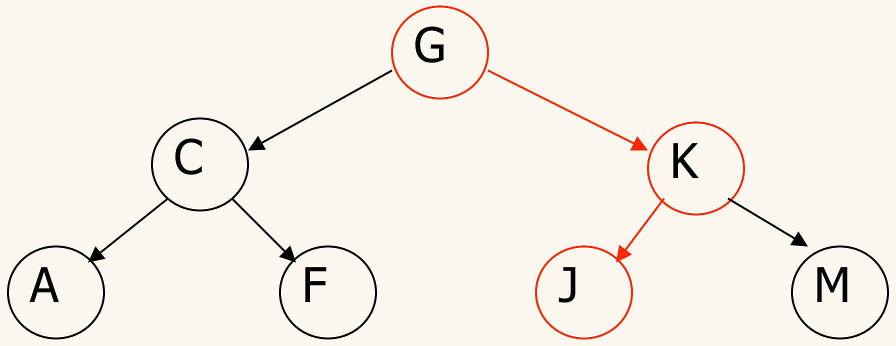
Data not shown in figure

Binary Search Tree

Binary search trees allow for fast insertion and removal of elements .They are specially designed for fast searching

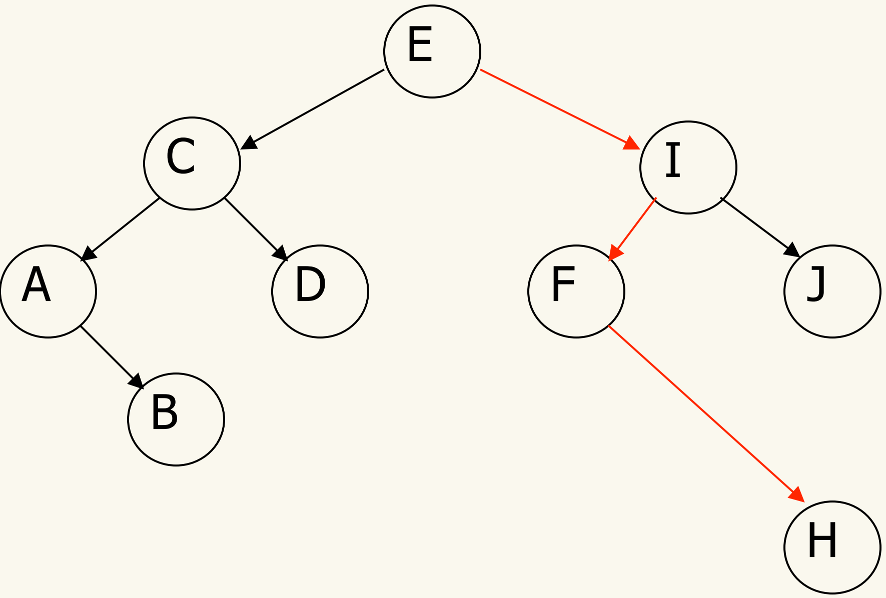
Searching for J:

- Start at the root of the tree (G)
- J is greater than G so discard the left half of the tree - just like binary search!
- Consider K, J is smaller than K, so discard the right half of the remaining tree.

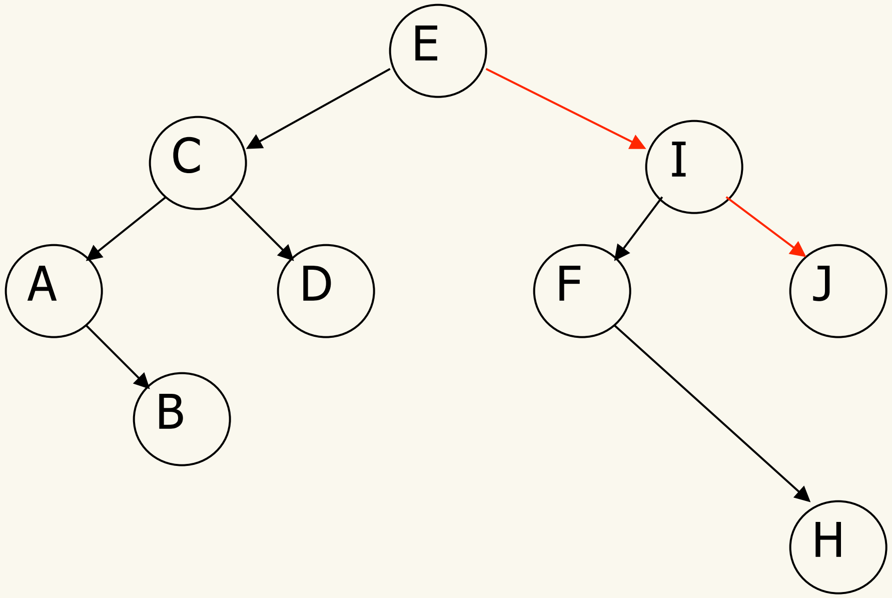


Finding a node

- Find H in the tree

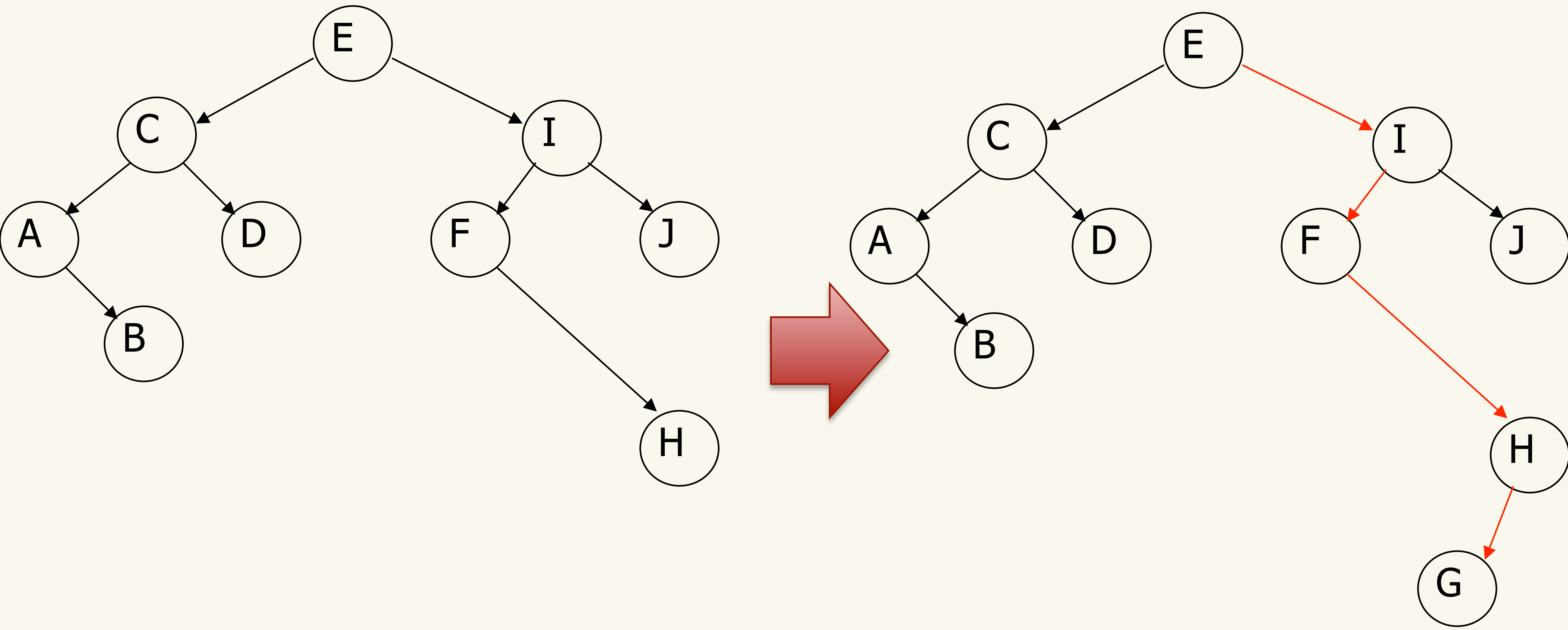


- Find K in the tree

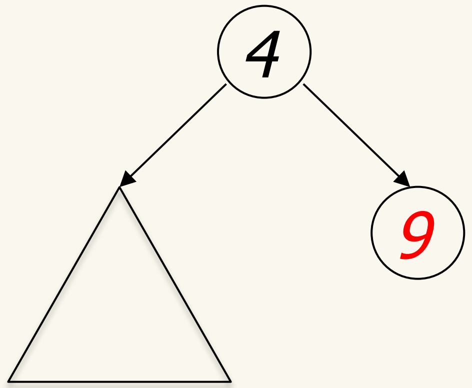


Adding a node

- We want to add G to the tree

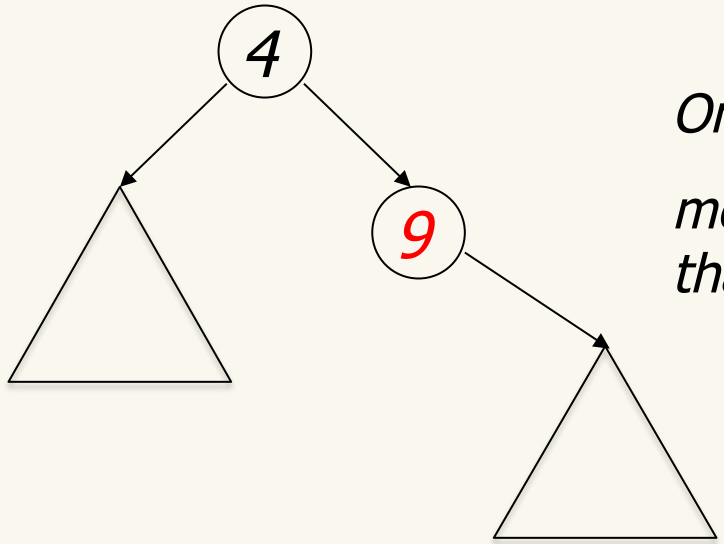


Removing a node from a BST



No children is an easy case:

Eliminate the node and the link to it



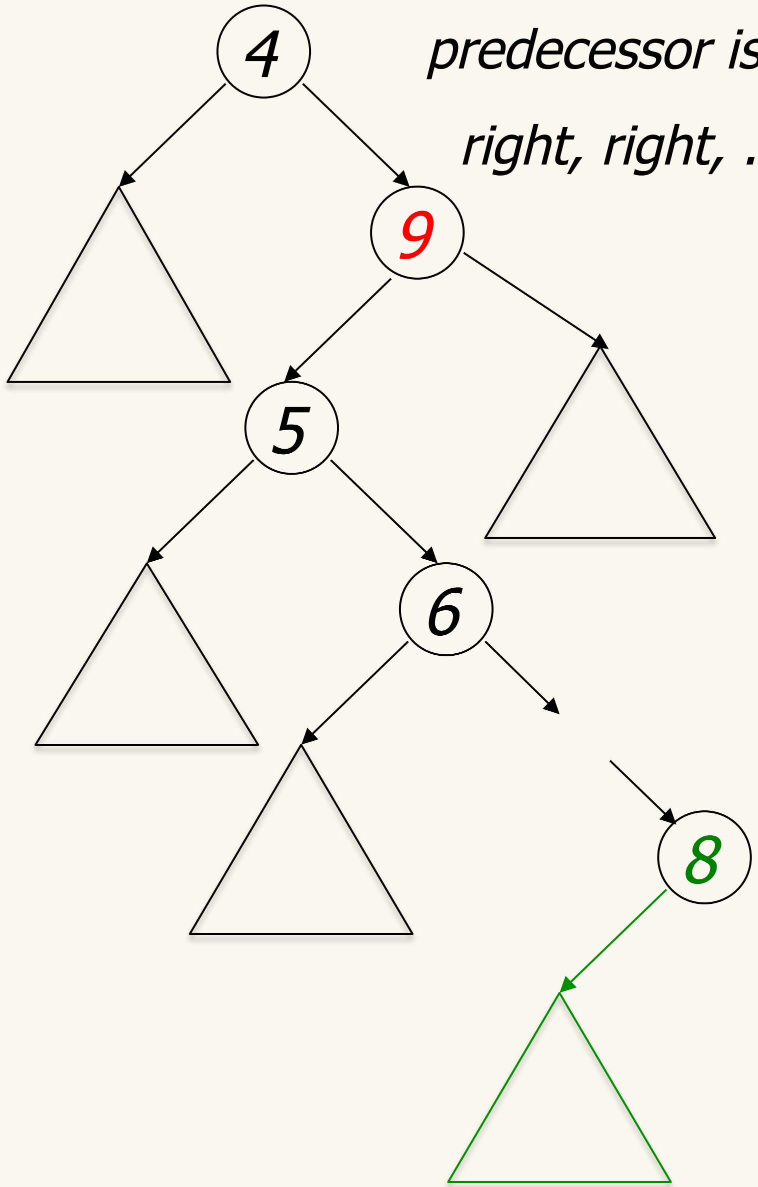
One child is an easy case:

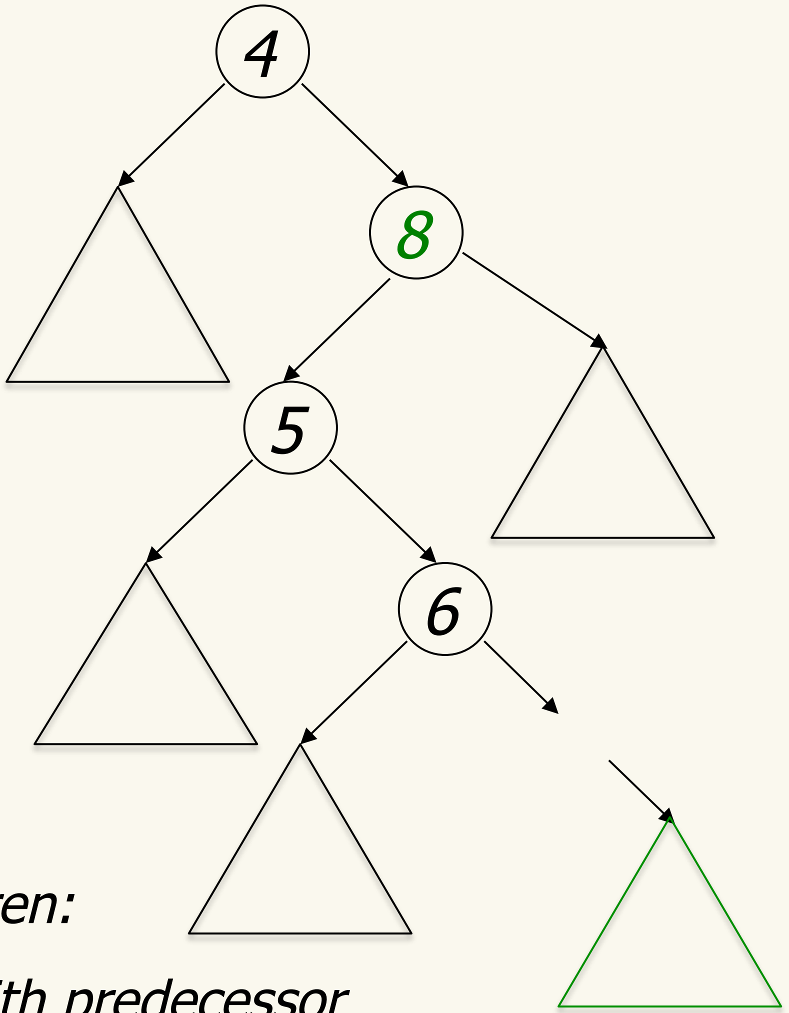
move the child up to replace the node that is erased

Removing a node from a BST

predecessor is left then

right, right, ... , right





Two children:

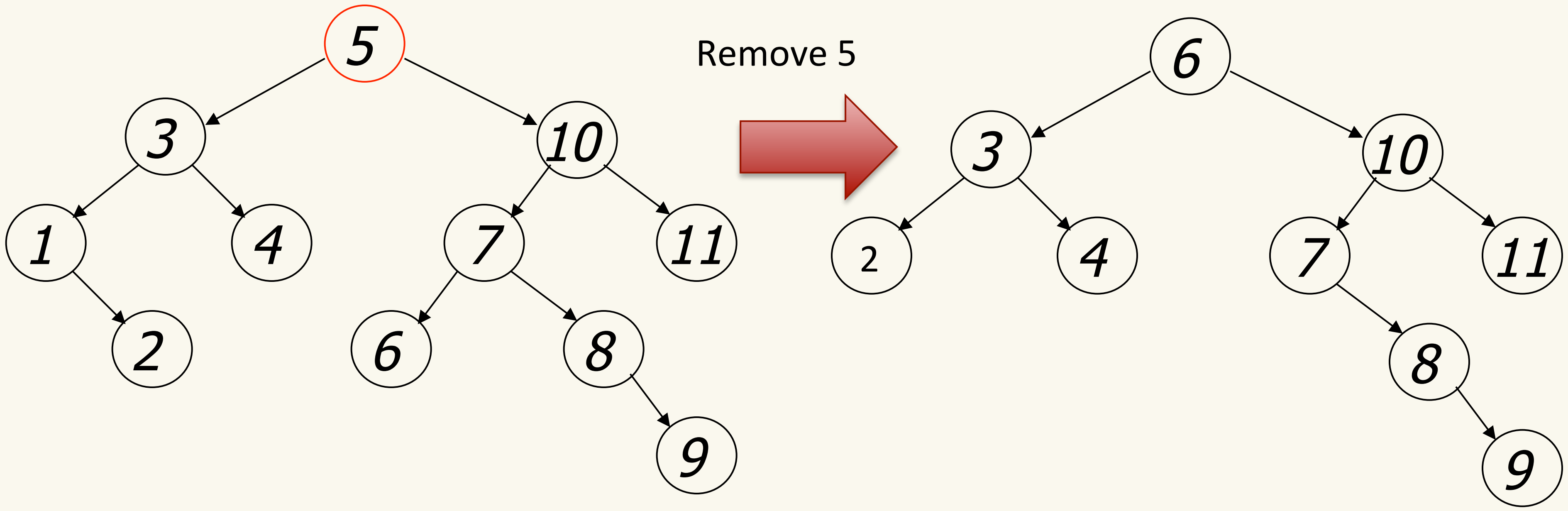
replace with predecessor

move predecessor subtree up

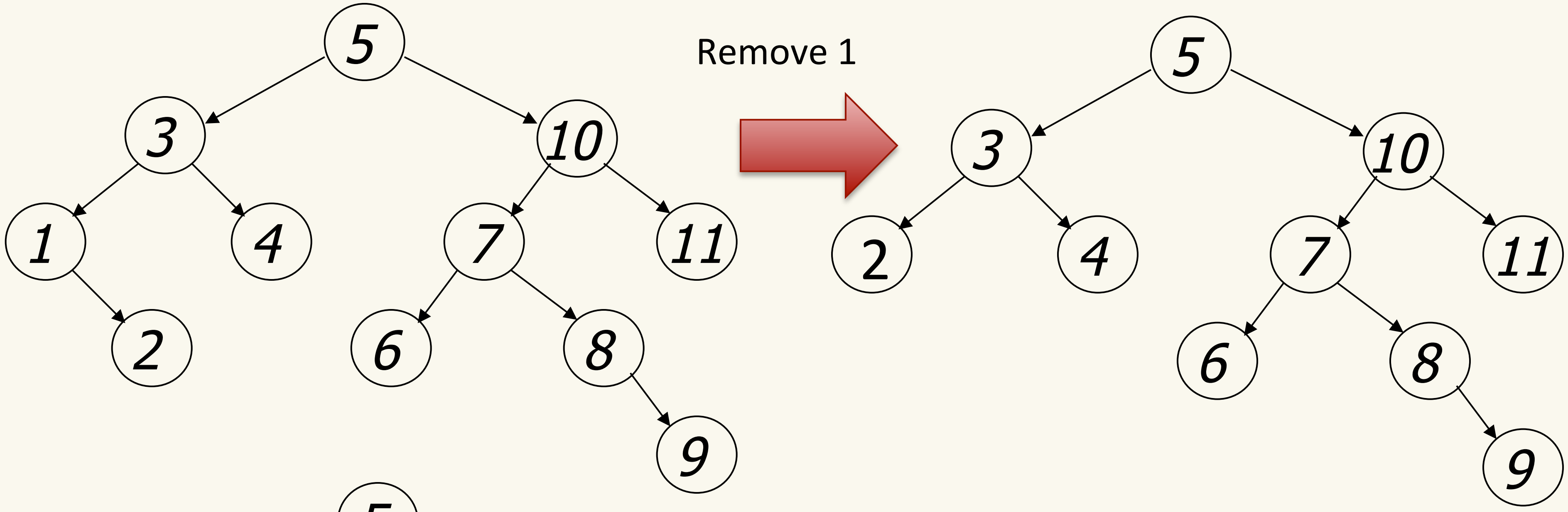
Removing a node from a BST

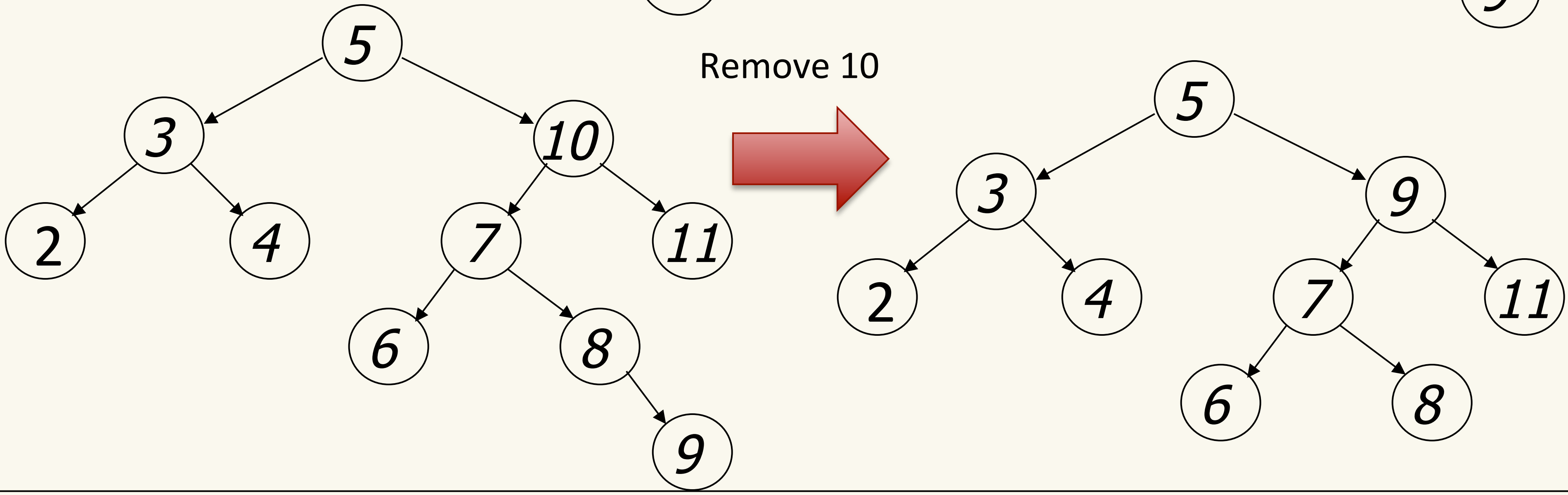
OR Successor, which is right then

left, left, … , left



Removing a node from a BST





BuildTree for BSTs

- Suppose the data 1, 2, 3, 4, 5, 6, 7, 8, 9 is inserted into an initially empty BST:
  - in order
  - in reverse order
  - median first, then left median, right median, etc. so: 5, 3, 8, 2, 4, 7, 9, 1, 6

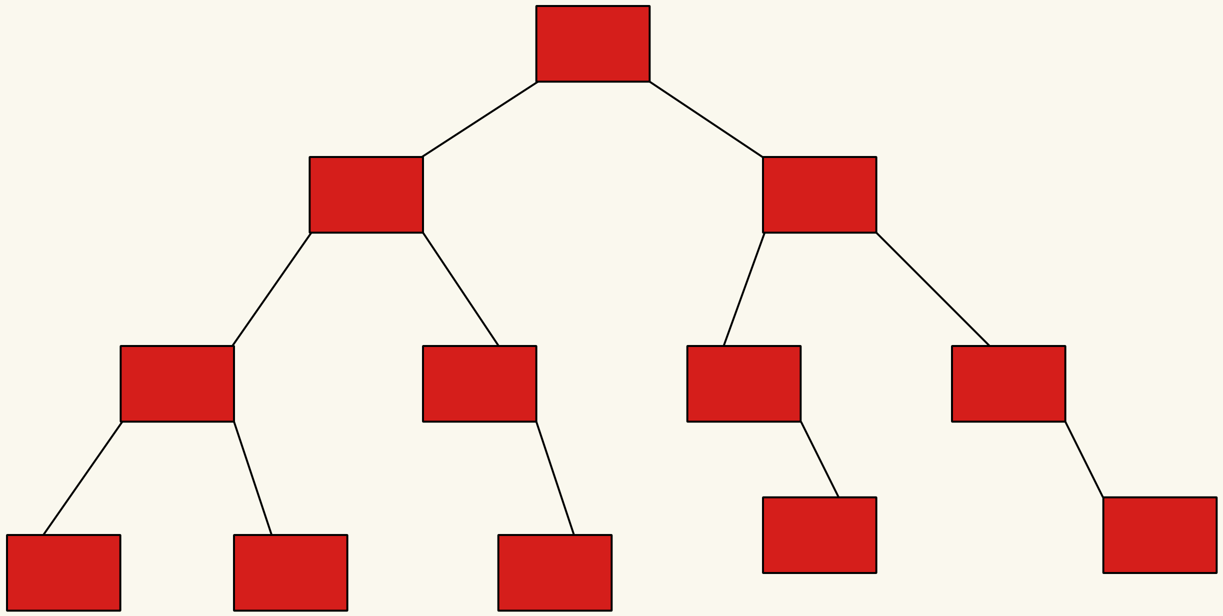
In-Class Exercise

What does this tell you about strengths and weaknesses of BSTs?

Using BSTs is only efficient if they are fairly balanced. Whether they are balanced is highly dependent on the order of the values being added. It is best to use BSTs when you are confident that your input values will be fairly random (e.g. not already sorted).

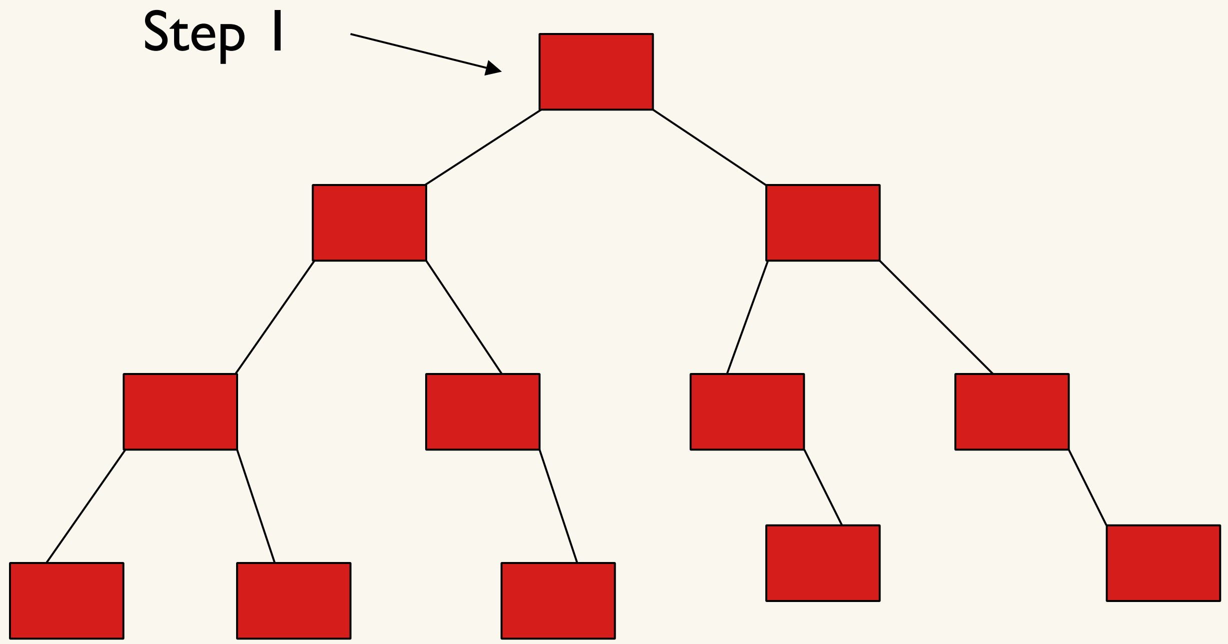
There are extensions and improvements to Binary Search Trees such as AVL trees or red-black trees. I encourage you to read about them, but they are outside of the scope of this course.

What makes a balanced BST efficient for searching?



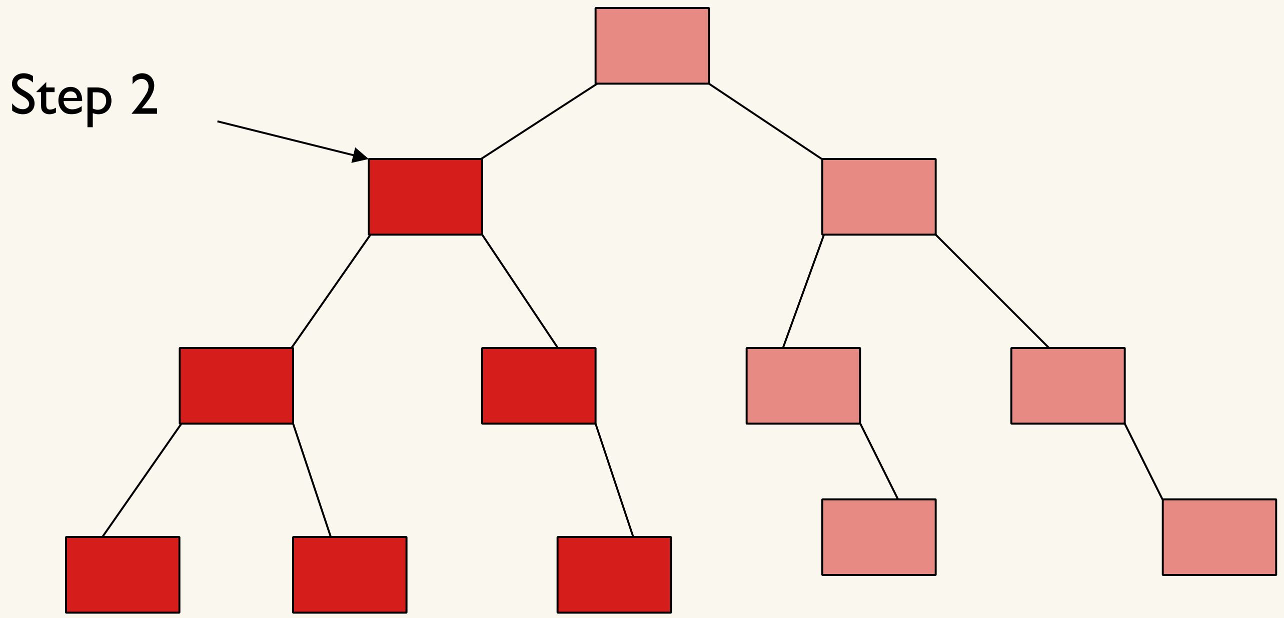
Each step we take as we search the tree reduces the remaining search space by half.

Sample Search



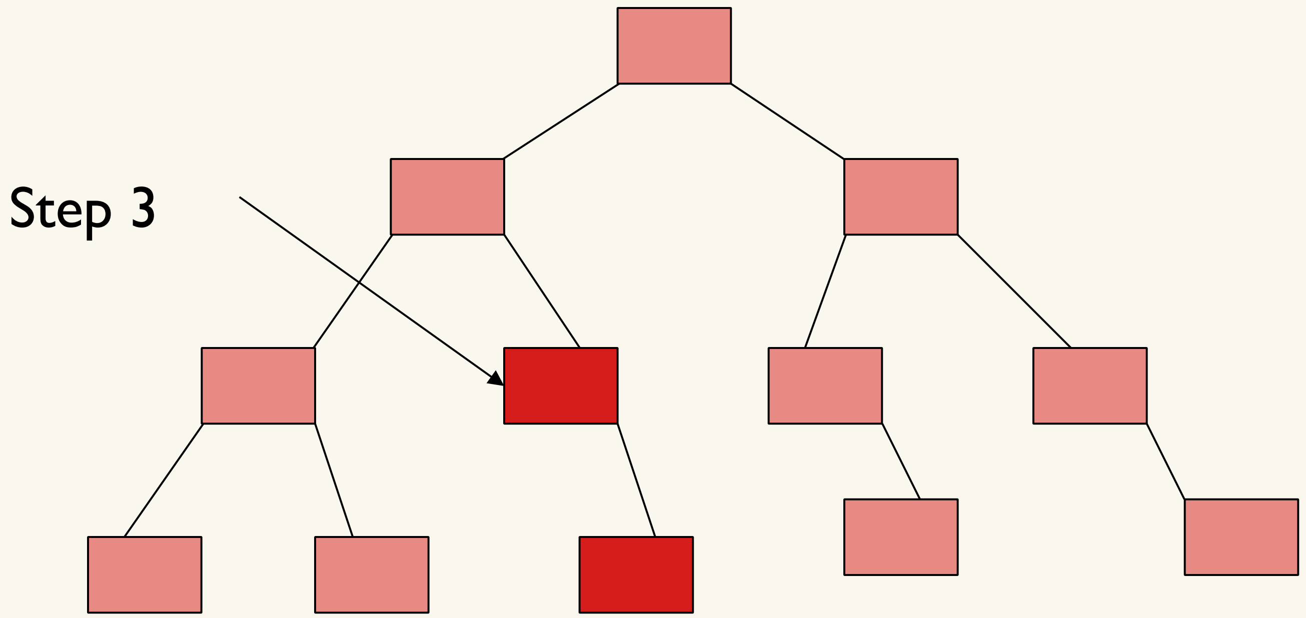
Each step we take as we search the tree reduces the remaining search space by half.

Sample Search



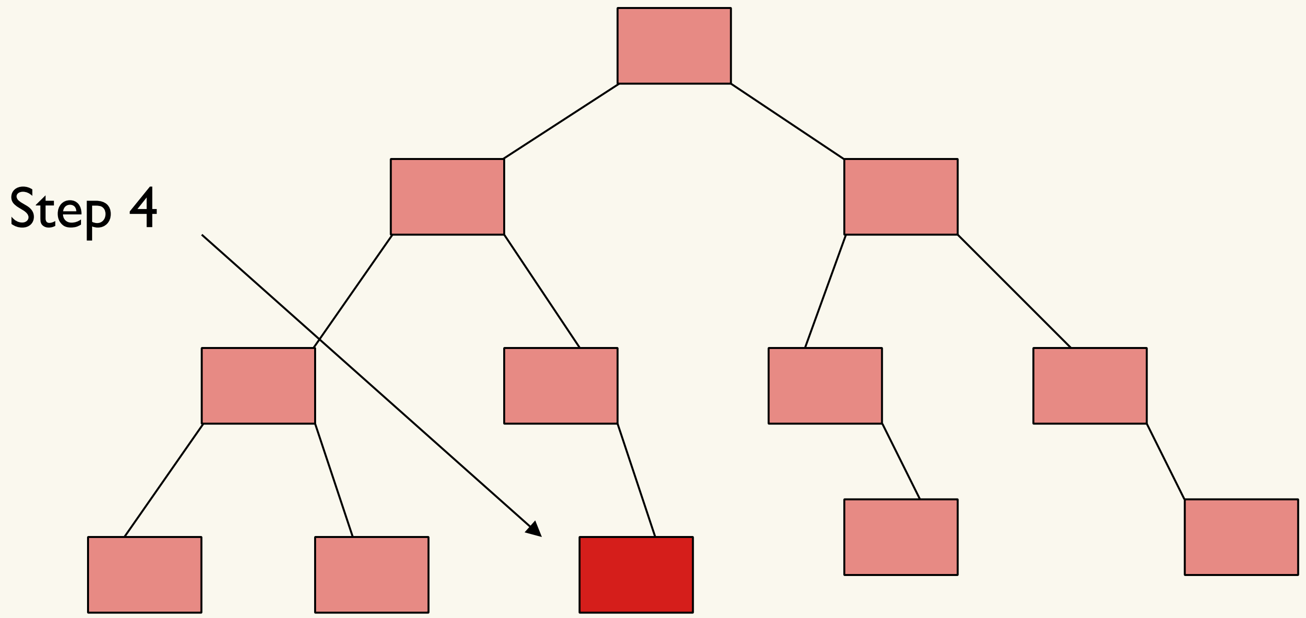
Each step we take as we search the tree reduces the remaining search space by half.

Sample Search



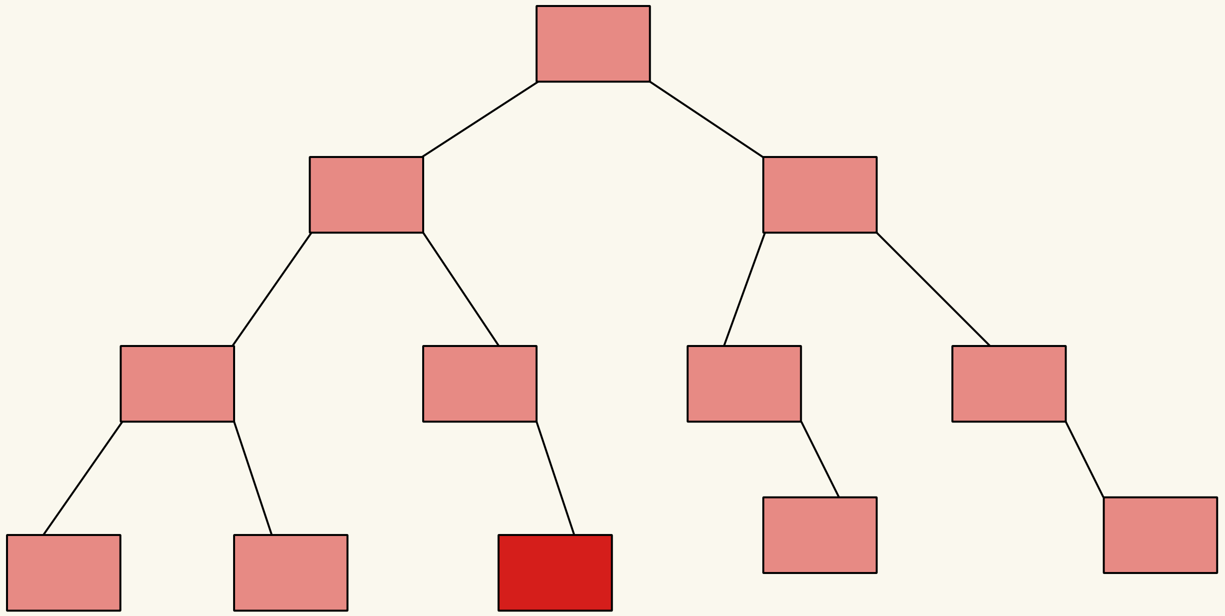
Each step we take as we search the tree reduces the remaining search space by half.

Sample Search



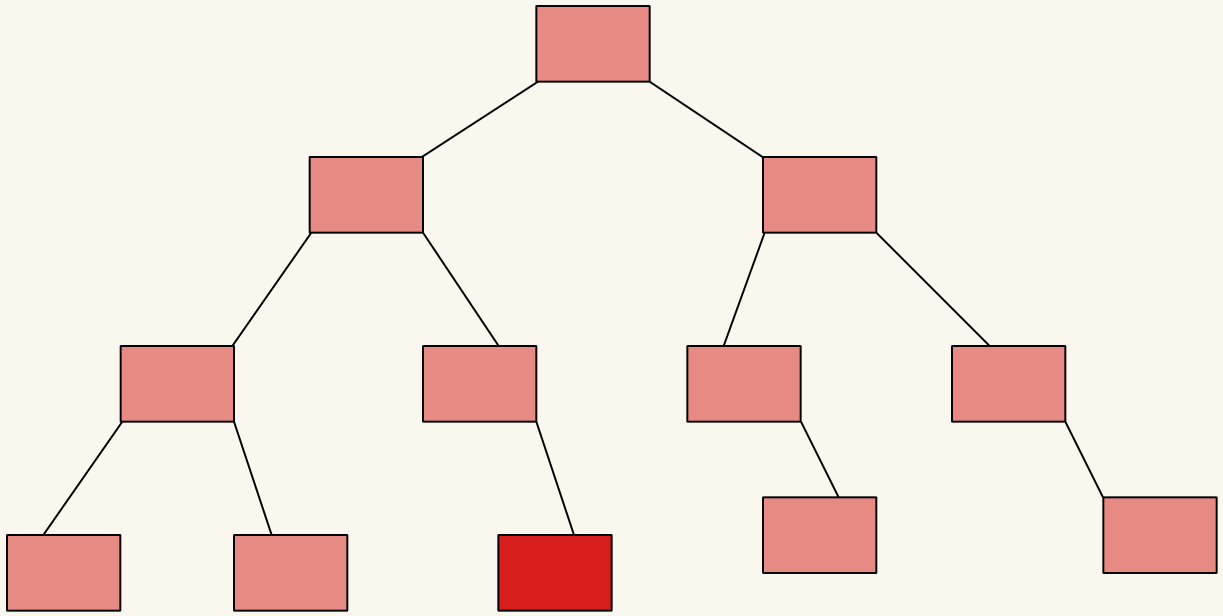
Each step we take as we search the tree reduces the remaining search space by half.

Sample Search



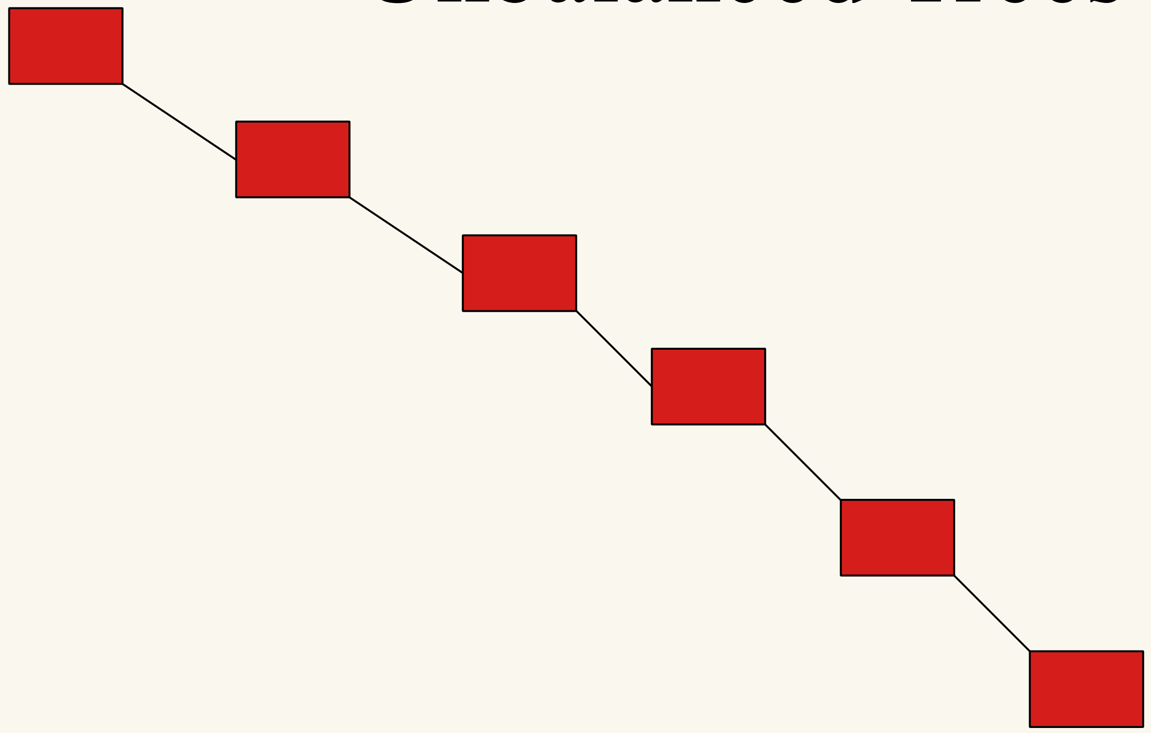
Each step we take as we search the tree reduces the remaining search space by half.

Height



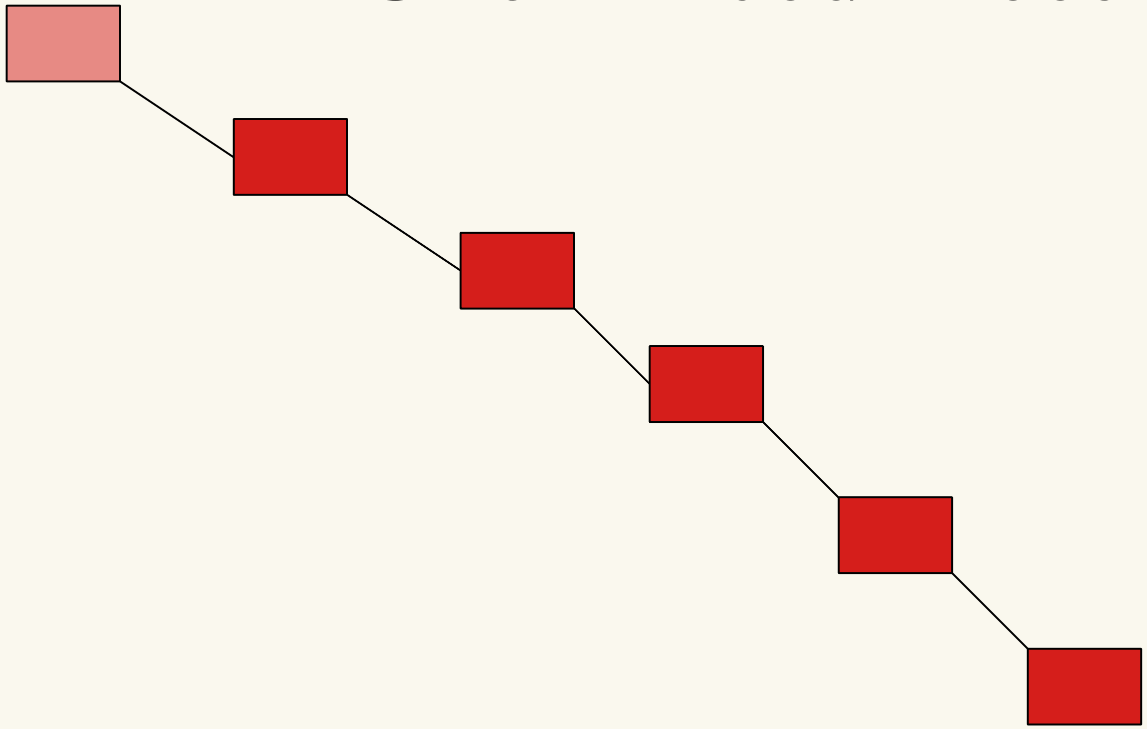
The tree has low height and all paths from the root node to other nodes are relatively short.

Unbalanced Trees



In contrast, this unbalanced tree is very high and has long paths from the root to other nodes. It essentially has degenerated to a linked list, which is very slow to search through.

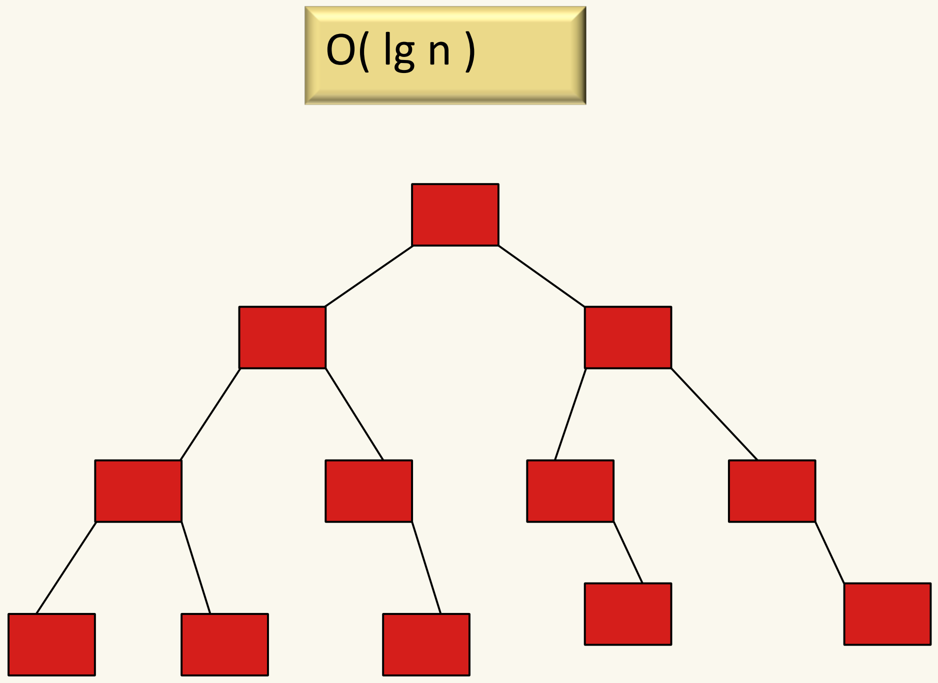
Unbalanced Trees

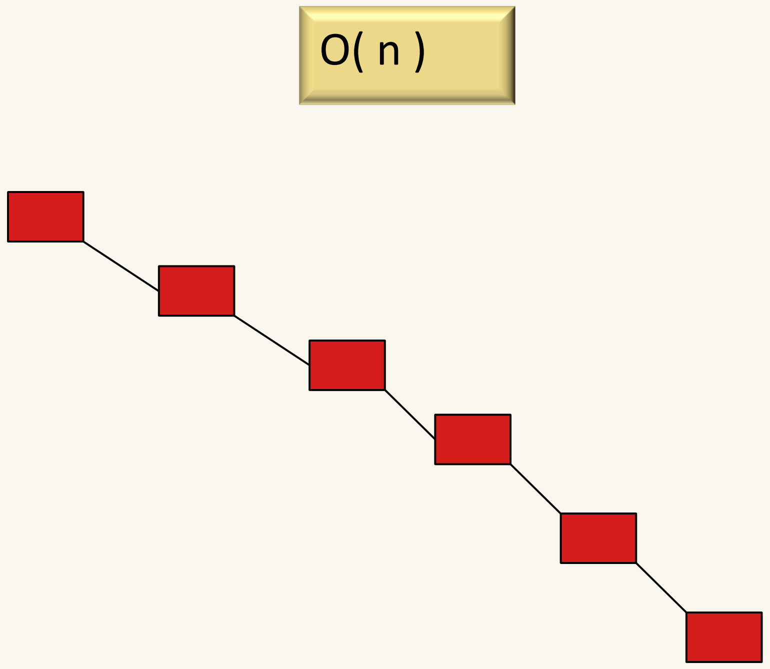


Now,with each step we take, we have only reduced the search space by one node.

Time of Search

- Time of search is proportional to the height of the tree





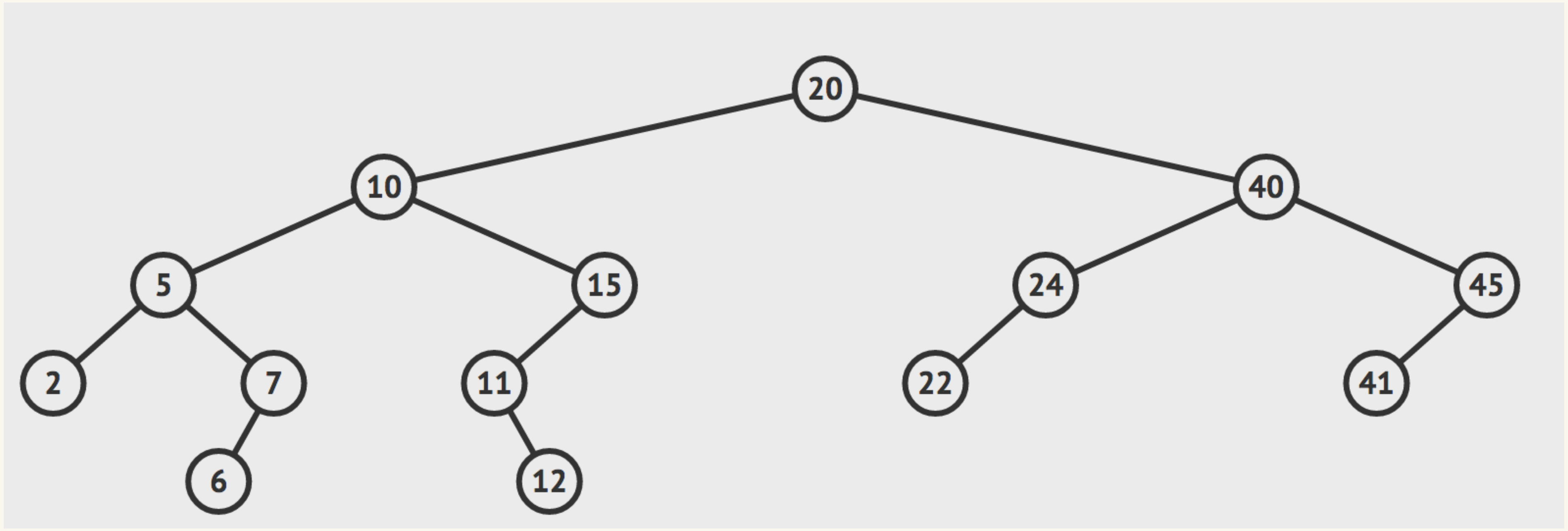
Analysis of BuildTree

- Worst case: O(n^2) as we've seen

- Average case assuming all orderings equally likely turns out to be O(n lg n).

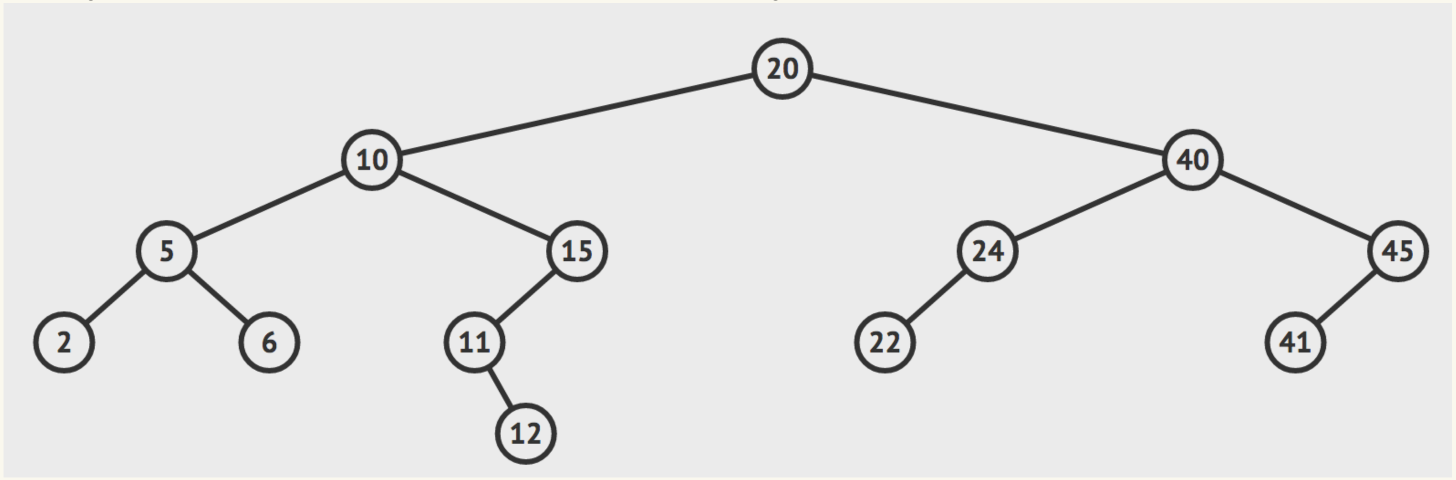
In-Class Exercise

- Draw the binary search tree, which results from adding the following keys in the given order:
  - 20, 10, 40, 5, 7, 2, 15, 11, 12, 6, 24, 22, 45, 41

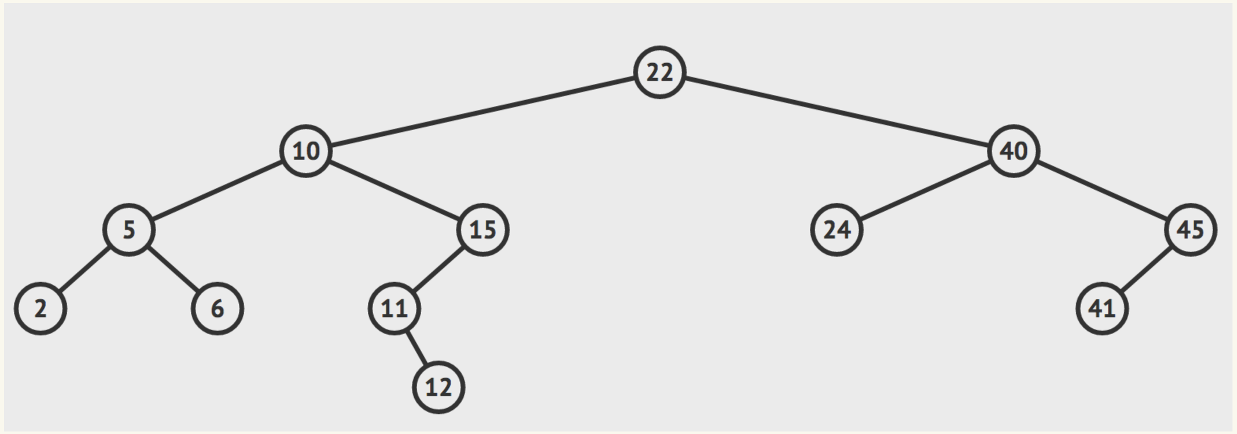


In-Class Exercise

- From your tree remove key with value 7

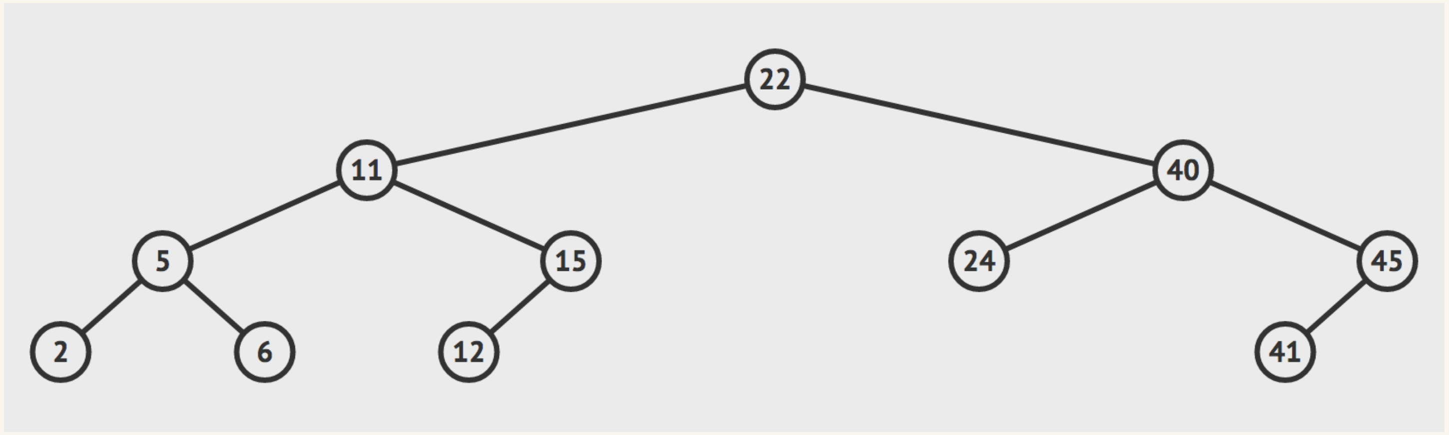


- From your tree remove key with value 20 (using successor)



In-Class Exercise

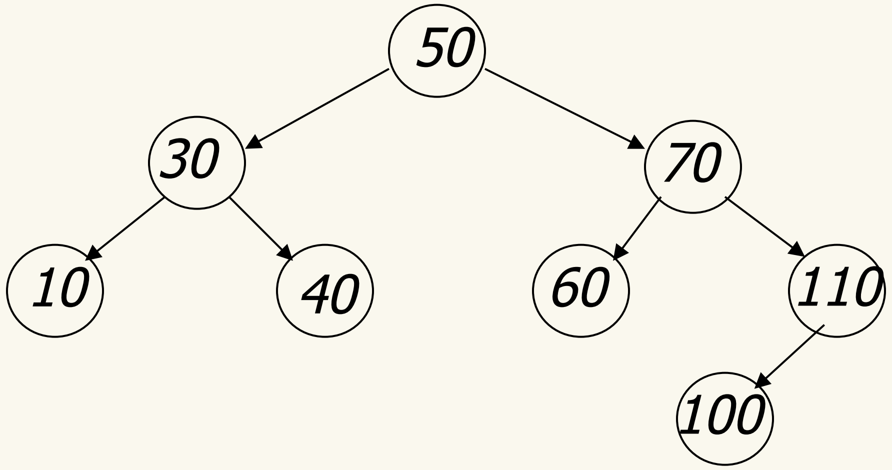
- From your tree remove key with value 10 (using successor)



- From your tree remove key with value 13
  - Item with key 13 does not exist
- What is the minimum/maximum values in the tree?
  - (2, 45)

BST with Arrays

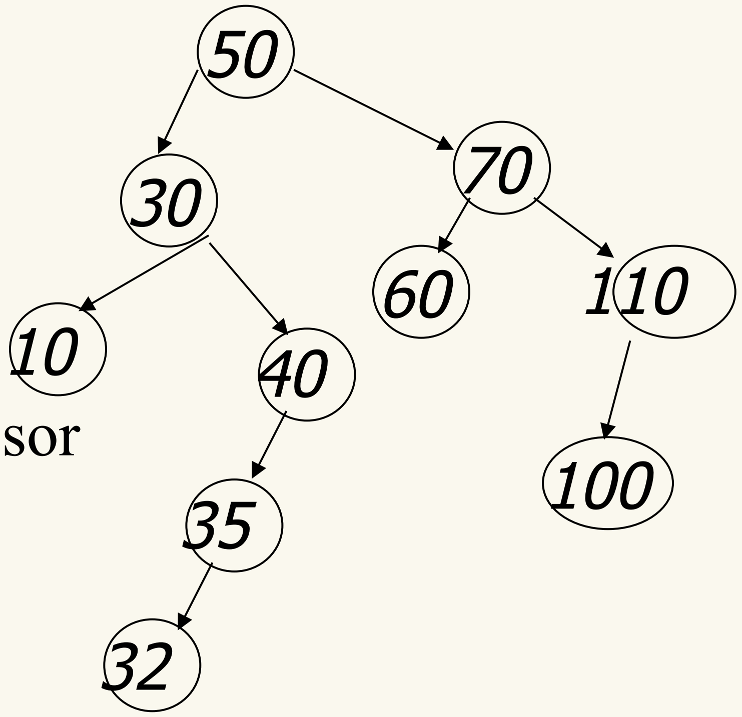
- Use an Array
  - `Root in element 1`
  - `leftChild(i) = 2i`
  - `rightChild(i) = 2i+1`
  - `parent(i) = i/2`



1	2	3	4	5	6	7	8	9	10	11	12	13	14
50	30	70	10	40	60	110							100

Implementation of a BST with Arrays

- Find
- Insert
  - Think about insert(105)
- Delete
  - Think about delete(50) using predecessor



1	2	3	4	5	6	7	8	9	10	11	12	13	14	20
50	30	70	10	40	60	110			35				100	32

Implementation of a BST with Arrays

- Deleting may be costly.
- Space complexity in the worst case is exponential -- $O(2^n)$

- What other data structure(s) can we use?
  – Linked lists

Implementation of a BST with linked lists

Each node contains an item (or an arbitrary amount of data), a pointer to its left subtree, and a pointer to its right subtree:

```
struct BNode{
    int                  item;
    struct BNode *  left;
    struct BNode *  right;
};
```

Key: item is usually a key, which is unique and allows entries all to be unique. There is usually a notion of data that comes with the key.

for example (student id key, student info data)

We will now look at the implementation of some of the operations listed earlier. To begin, we will write a `makeNode` function to create a new `BNode` as needed.

Creating a BNode

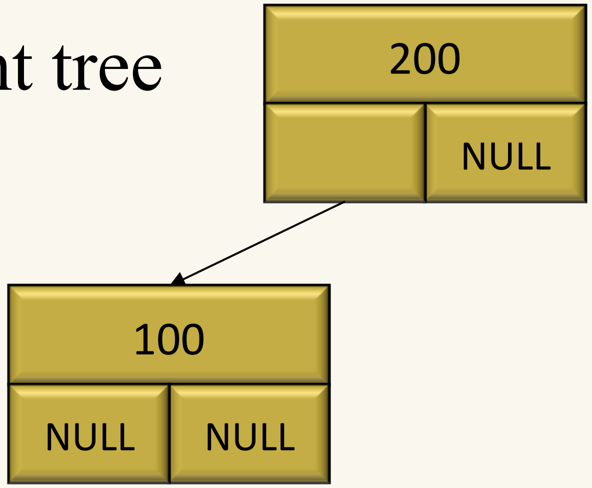
```
/* a new node is created and its address is returned */

BNode * makeNode(int item, BNode * leftChild, BNode * rightChild){
  BNode *  temp;
  temp = ( BNode *) malloc(sizeof(BNode));

  temp->item  = item;
  temp->left  = leftChild;
  temp->right = rightChild;

  return temp;
}
```

```
/* create a new tree */
BNode* createTree(int item){
  return makeNode(item, NULL, NULL);
}
```

Creating a Bnode exercise

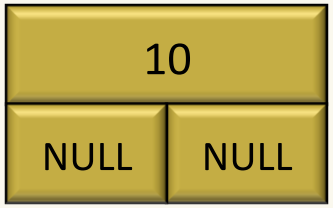
```
myTreeRoot = makeNode(200, makeNode(100, NULL, NULL), NULL);
```

- Draw the resultant tree



```
BNode* root = createTree(10);
```

- Draw the resultant tree

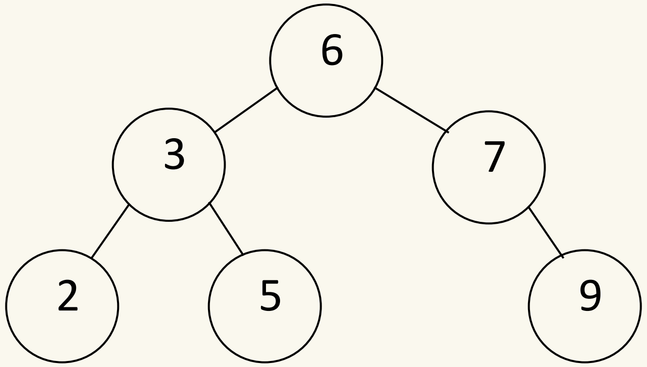


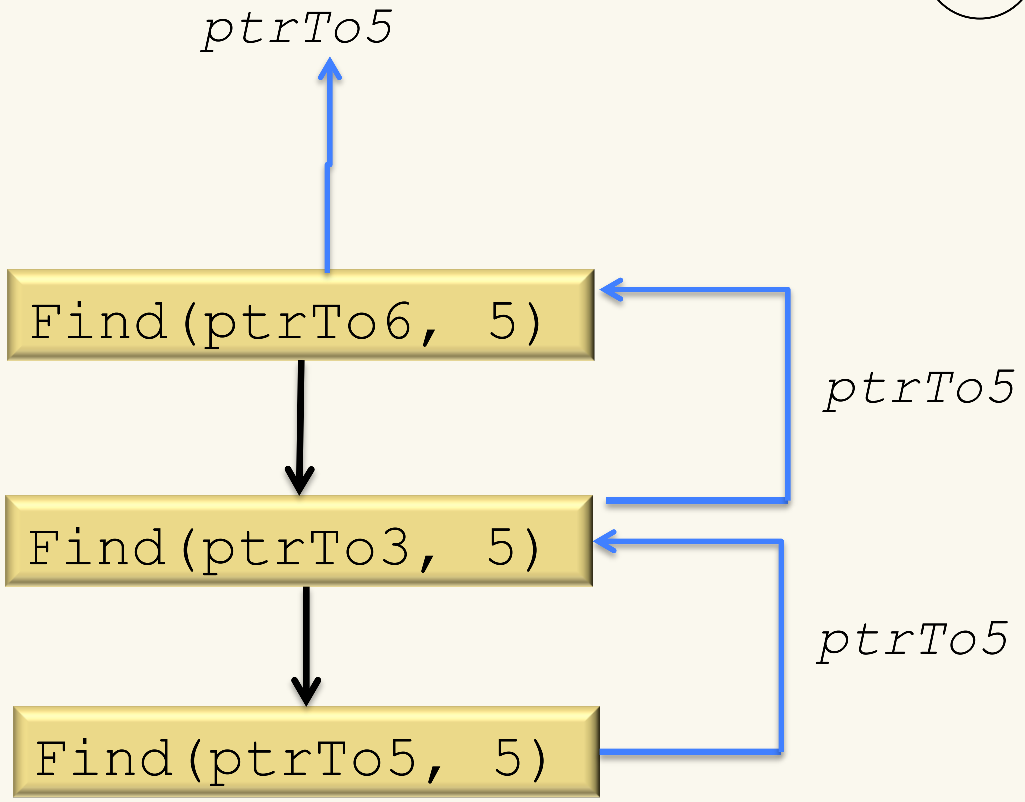
Finding a BNode

```
 /*  Assuming root points to the root of a binary tree, if item is
  *  in the tree, return the address of its node; otherwise, return NULL
  */
BNode * find(BNode * root, int item){
   if (root == NULL  ||  root->item == item)
      return root;

   if (item < root->item)
      return find(root->left, item);
   else
      return find(root->right, item);
}
```

Finding a Bnode exercise

- For the BST on the right draw the recursion tree of `find(ptrTo6, 5)`



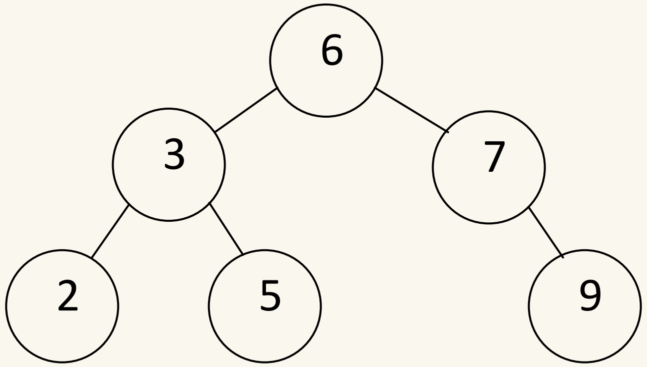


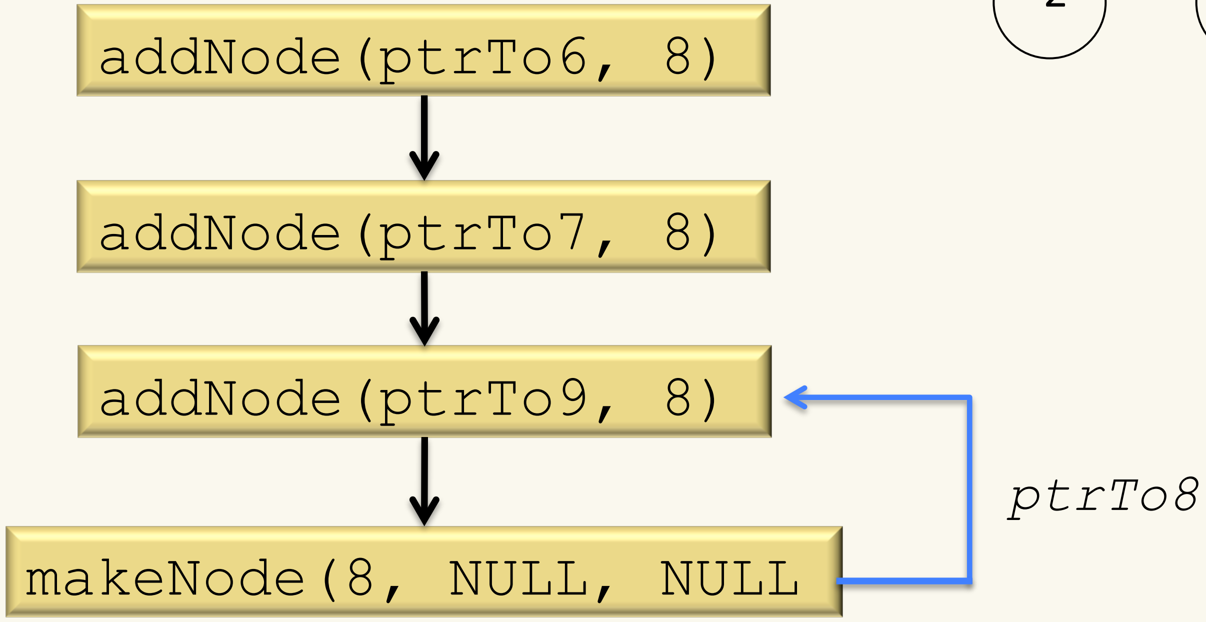
Adding a BNode

```
/*  Assuming root points to the root of a binary tree, and items are
 *   unique.
void addNode( BNode * root, int item){

  if (item < root->item){
    if (root->left)  /*  same as root->left!=NULL */
      addNode(root->left, item);
    else
      root->left = makeNode(item, NULL, NULL);
  }

  if (item > root->item){
    if (root->right) /*  same as root->right!=NULL */
      addNode(root->right, item);
    else
      root->right = makeNode(item, NULL, NULL);
  }
}
```

Adding a Bnode exercise

- For the BST on the right draw the recursion tree of `addNode(ptrTo6, 8)`





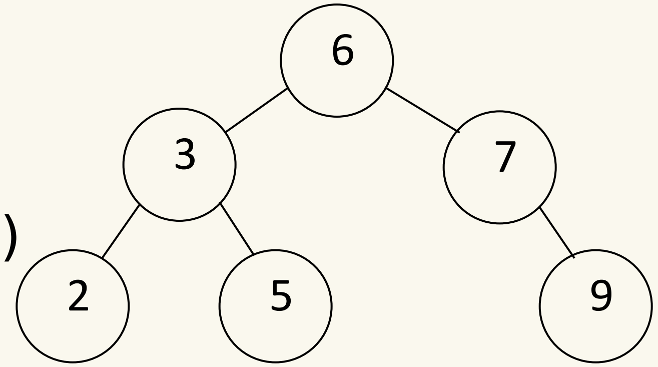
- What happens if you attempt to add an existing node ?

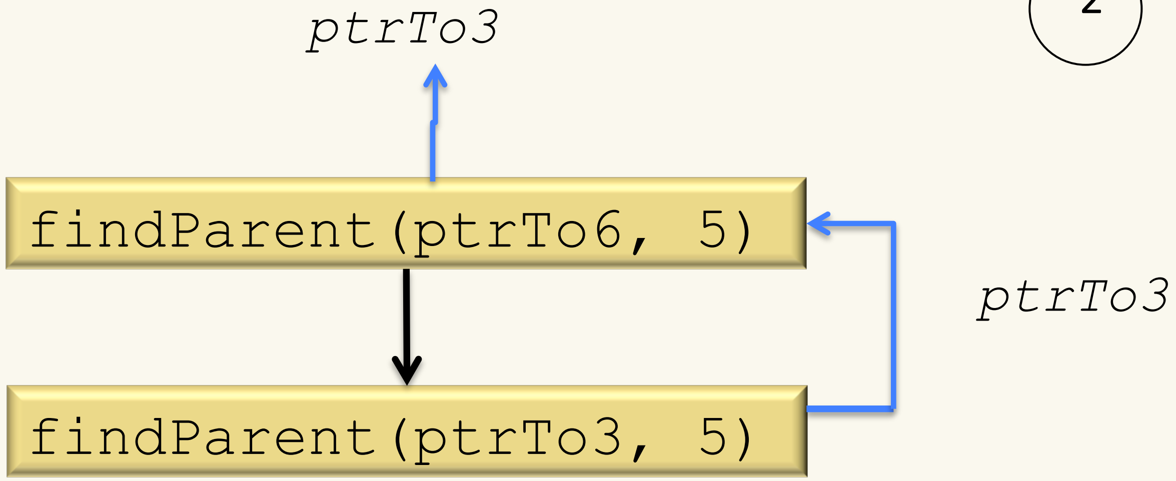
Finding parent of a BNode

```
/**
 * Finds the parent of node in the tree rooted at rootNode
 */
BNode* findParent( BNode * root, int item)
{
  if (root == NULL || root->item == item)
    return NULL;

  else if (root->left && (root->left->item == item))
    return root;

  else if (root->right && (root->right->item == item))
    return root;

  else if (item < root->item )
    return findParent(root->left, item);

  else
    return findParent(root->right, item);
}
```

Finding parent of a Bnode exercise

- For the BST on the right draw the recursion tree of `findParent(ptrTo6, 5)`





- What should be returned if we want to find parent of 6?

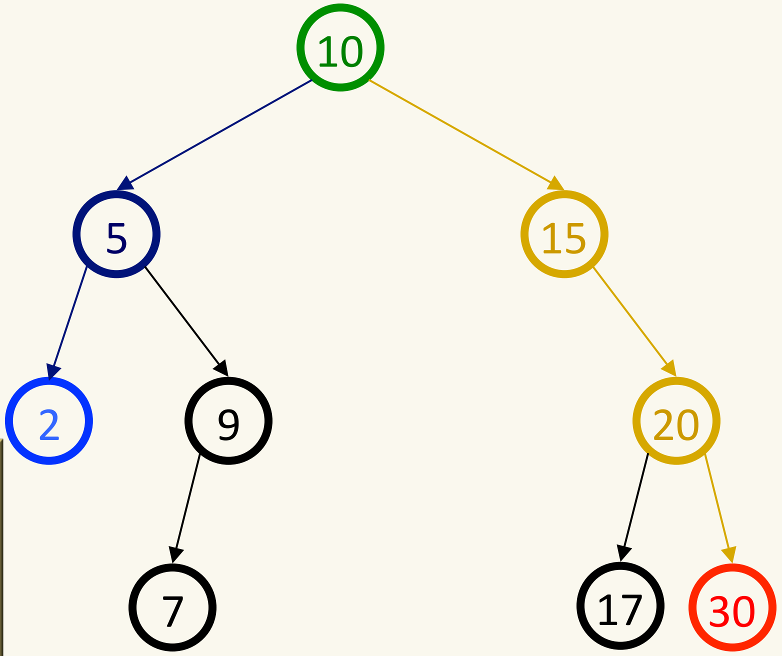
Bonus: FindMin/FindMax

- Find minimum

```
BNode* findMin(BNode* root){
  if (!root) return NULL;

  else if (!root->left) return root;

  else
    return findMin(root->left);
}
```

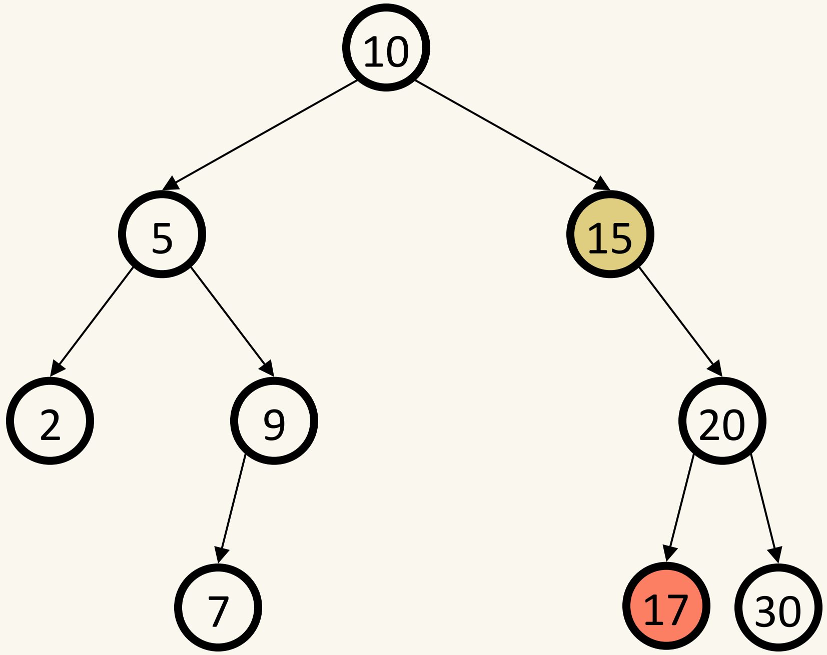
- Find maximum

```
BNode* findMax(BNode* root){
  if (!root) return NULL;

  else if (!root->right) return root;

  else
    return findMax(root->right);
}
```



Double Bonus: Successor

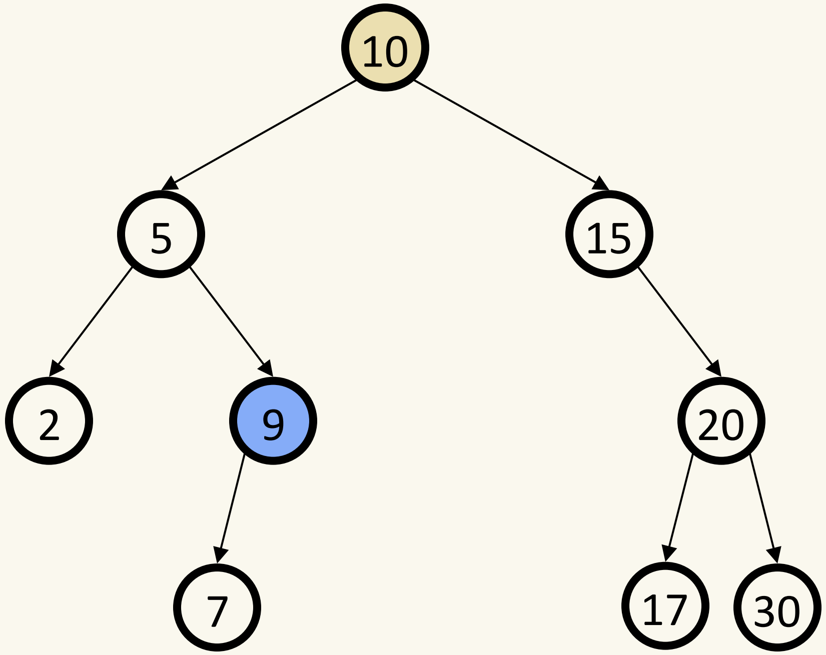
Find the next larger node in a node's subtree.



```
BNode* findSucessor( BNode * theNode){
  if (theNode==NULL || theNode->right==NULL)
    return NULL;

  else
    return findMin(theNode->right);
}
```

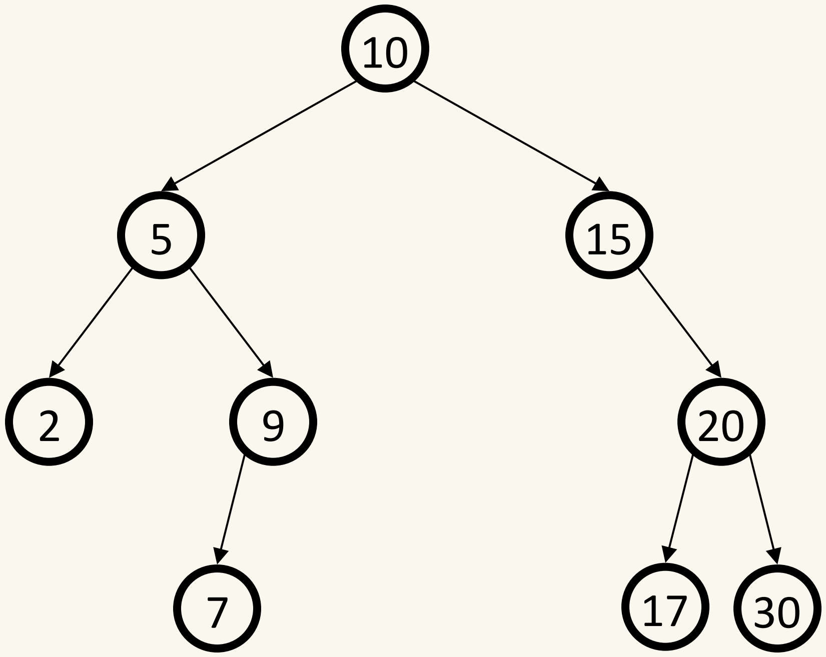
More Double Bonus: Predecessor

Find the next smaller node in a node's subtree.



```
BNode* findPredecessor( BNode * theNode){
  if (theNode==NULL || theNode->left==NULL)
    return NULL;

  else
    return findMax(theNode->left);
}
```

Deletion



Why might deletion be harder than insertion?

Deleting a BNode

- The task of removing a node from a binary tree is quite complicated; therefore, we will break the task into parts.

```
struct BNode* deleteNode(struct BNode* root, int item)
{
  struct BNode* target = find(root, item);
  struct BNode* parent = findParent(root, item);

  if (!target)
    return root; /* item not in tree */
```

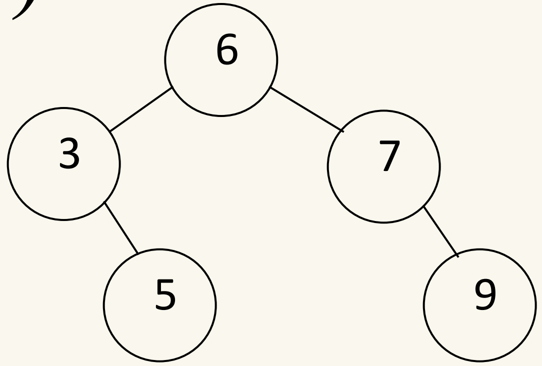
Deleting a BNode (leaf case)

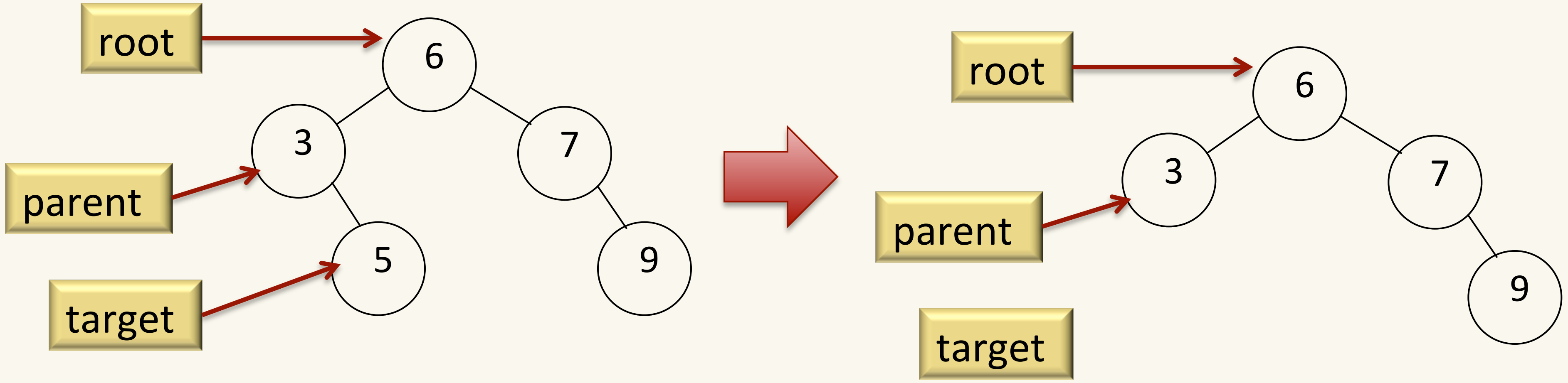
- Now that we have a pointer that points to the node to be deleted, we proceed according to one of 4 cases:
- Case 1: node to be deleted is a leaf

```
if (target->left == NULL && target->right ==NULL){
/* -------------------- leaf------------------ */
     if (parent != NULL)
         if (parent->left == target))
             parent ->left = NULL;
         else
             parent ->right = NULL;

    else // parent == NULL
        root = NULL;
    free(target);
    return root;
}
```

Deleting a BNode (leaf case) exercise

- Trace the code to see what the tree would look like after `deleteNode(ptrTo6, 5)has been executed`





check to see which child of parent needs to be updated to NULL

Deleting a BNode (one child)

- Case 2: node to be deleted has only a left child

```
if (target->left != NULL && target->right == NULL){
/* ---------------- Only left child------------ */

   if (parent != NULL) {
        if ( parent->left == target )
         parent->left = target->left;
        else
      parent->right = target->left;
   }
   else // parent == NULL
     root = target->left;

   free(target);
   return root;
}
```

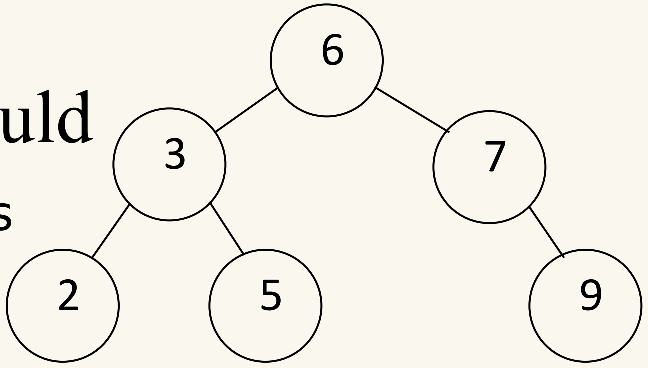
Deleting a BNode (one child)

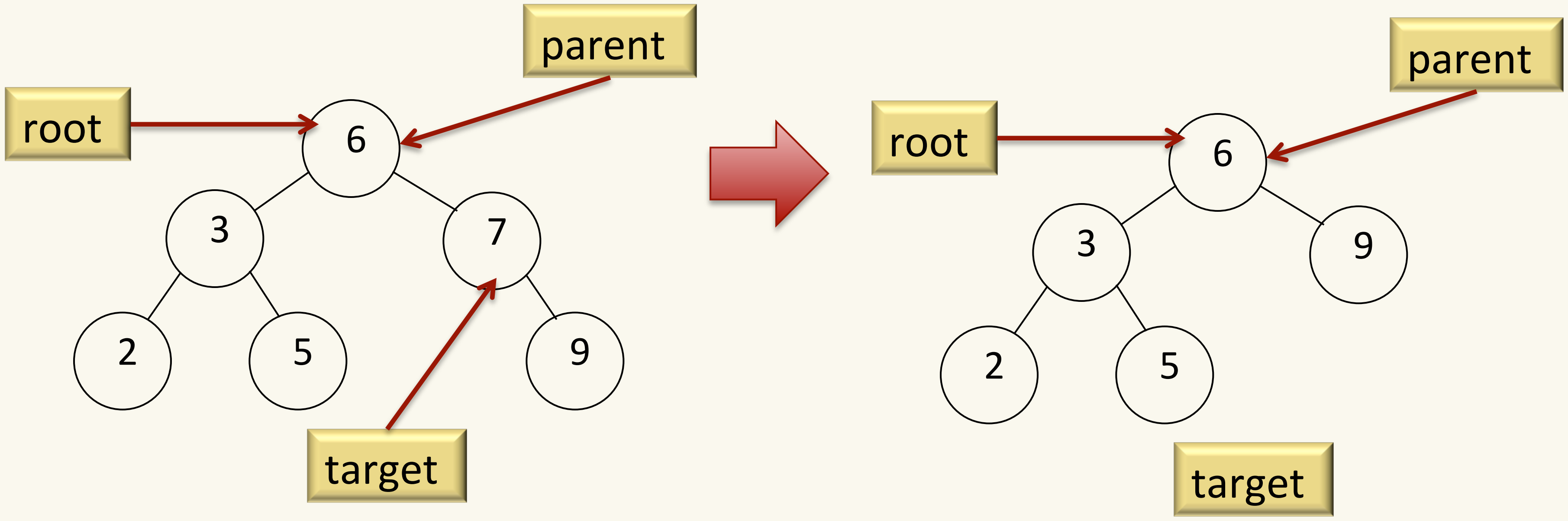
- Case 3: node to be deleted has only a right child

```
if (target->left == NULL && target->right !=NULL){
 /* ------------ Only right child------ */

    if (parent != NULL) {
        if ( parent->left == target )
               parent->left = target->right;
        else
               parent->right = target->right;
    }
    else
        root = target->right;

    free(target);
    return root;
}
```

Deleting a BNode (one child) exercise

- Trace the code to see what the tree would look like after `deleteNode(ptrTo6, 7) has been executed.`





Deleting a BNode (both children)

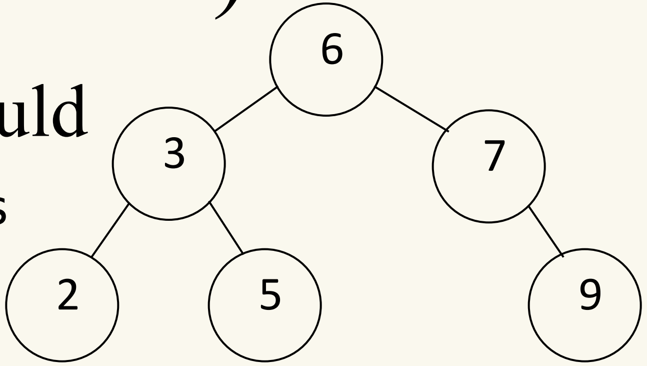
- Case 4: node to be deleted has both a left and right child
- This is the tricky case. There is no obvious way to remove a node having two children and re-connect the tree. Instead, we can choose not to delete the node, but rather copy data from either a leaf node or one with just the left child (which is easy to remove)

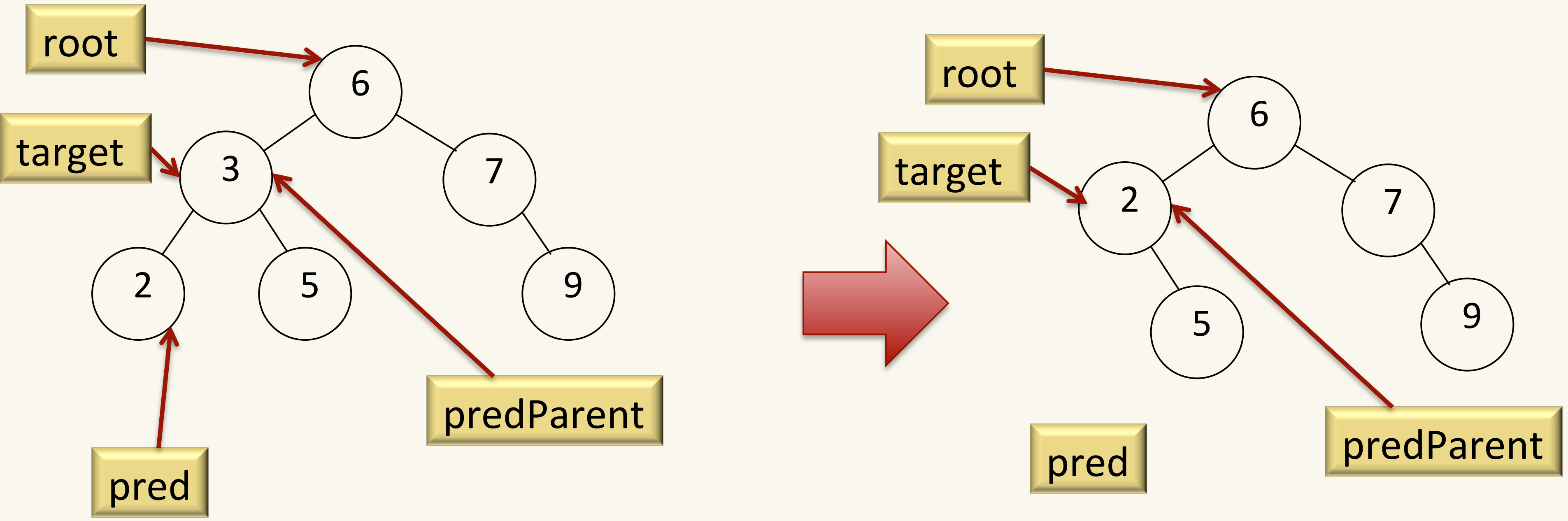
Deleting a BNode (both children)

```
if (target->left != NULL && target->right != NULL) {

   /* find the replacing node and its parent */
   BNode* pred = findPredecessor(target);
   BNode* predParent = findParent(root, pred->item);

   target->item = pred->item;
   if (target == predParent) /* replaced by left child */
     predParent->left = pred->left;
   else
   /* could either be Null or temp has a left child */
     predParent->right = pred->left;

   free(pred);
   pred = NULL;
 }

 return root; // return root when you're done.
```

Deleting a BNode (both children) exercise

- Trace the code to see what the tree would look like after `deleteNode(ptrTo6, 3)` has been executed.

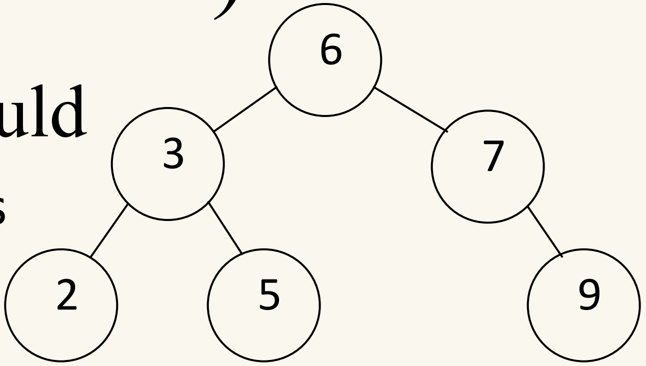


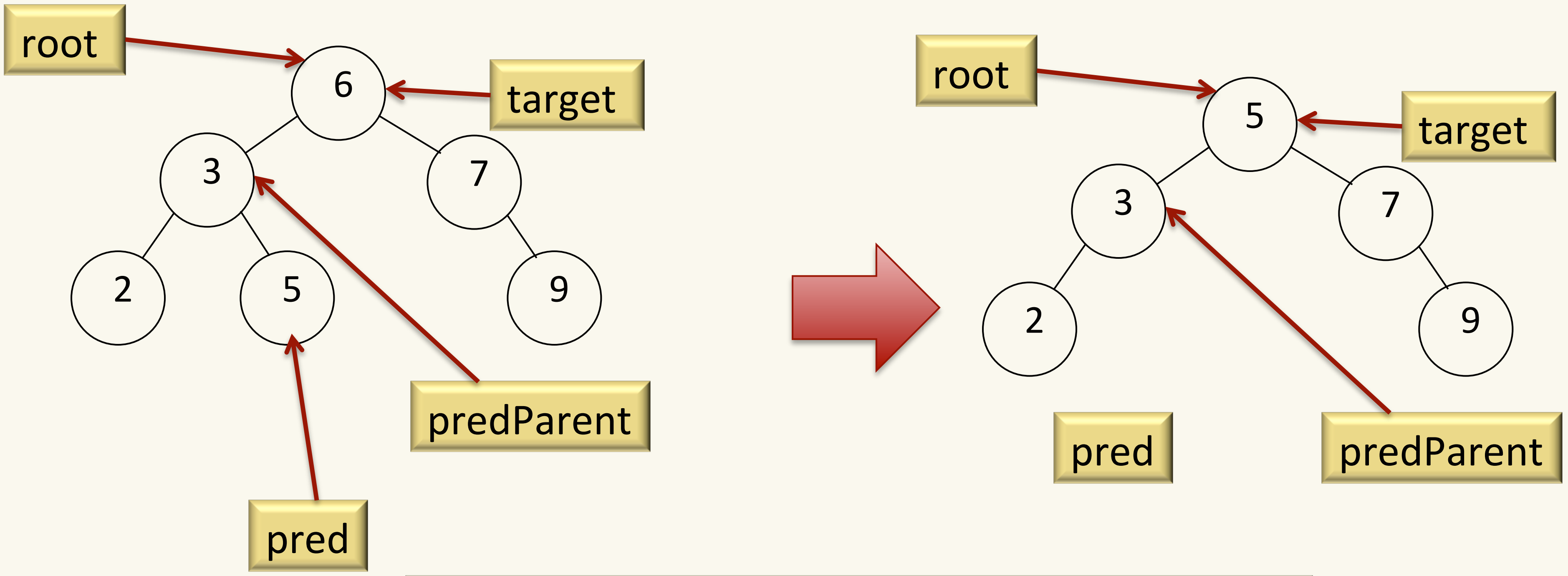


```
predParent->left = pred->left; /* replaced by left child */
```

Deleting a BNode (both children) exercise

- Trace the code to see what the tree would look like after `deleteNode(ptrTo6, 6)` has `been executed.`

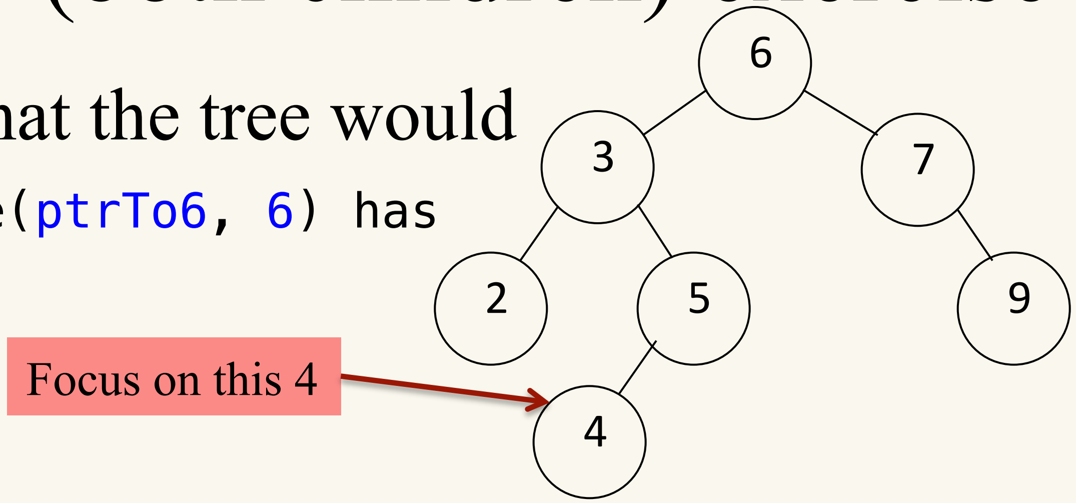


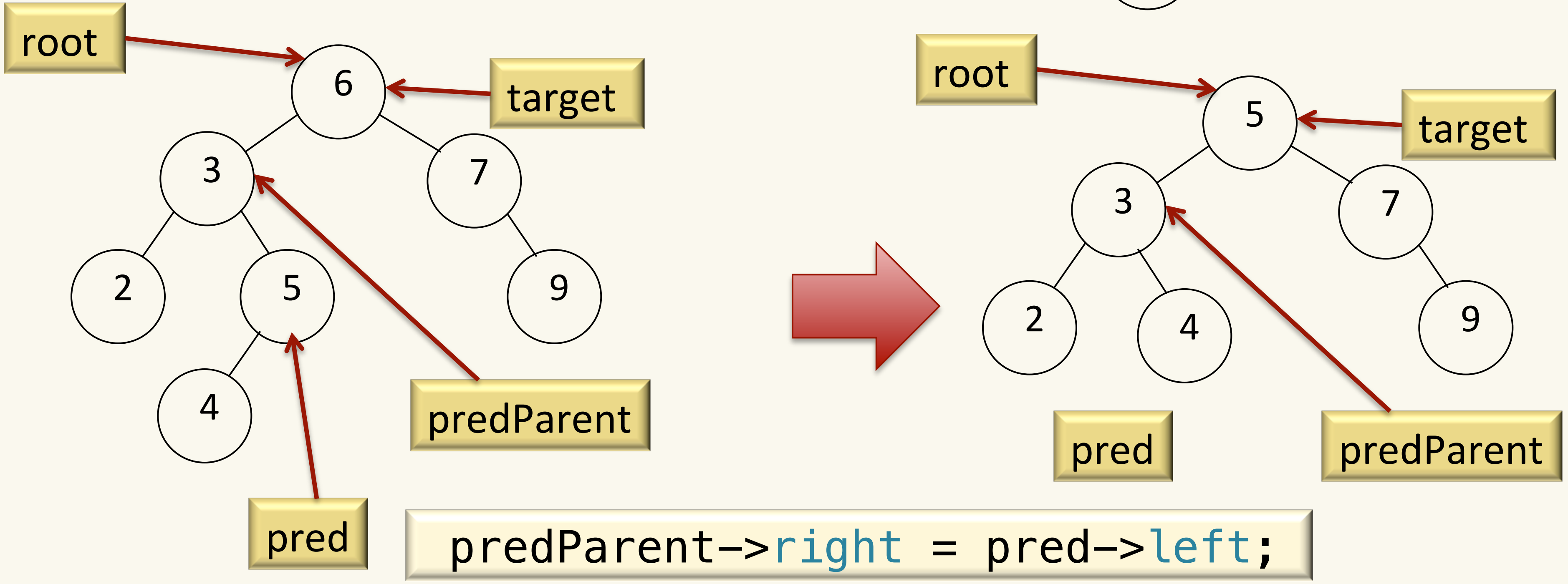


```
predParent->right = pred->left;
```

Deleting a BNode (both children) exercise

- Trace the code to see what the tree would look like after `deleteNode(ptrTo6, 6)` has `been executed.`





```
predParent->right = pred->left;
```

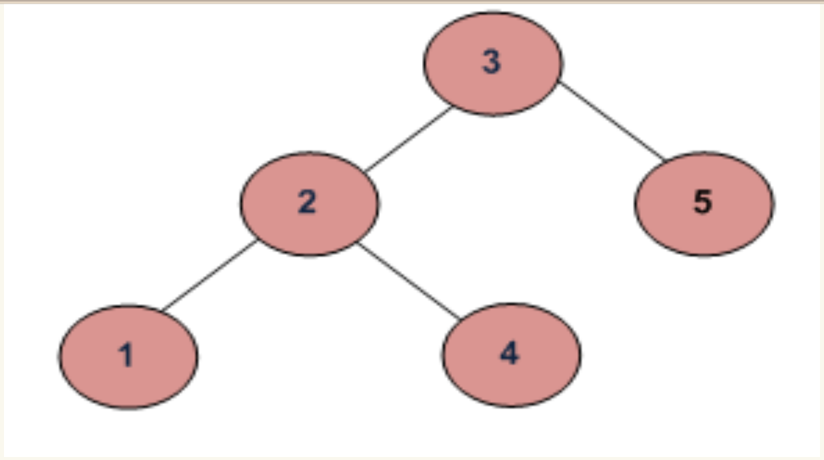
In-class Exercise

- Is this code correct? If yes, briefly justify your answer; if no, draw a small tree for which the code gives the wrong answer.

```c
// Returns 1 if the tree rooted at x is a BST, 0 otherwise
int check_bst(Node * x){
  if (x == NULL) return 1;

  if ((x->left != NULL) && (x->left->key > x->key))
    return 0;
  if ((x->right != NULL) && (x->right->key < x->key))
    return 0;

  return check_bst(x->left) && check_bst(x->right);
}
```

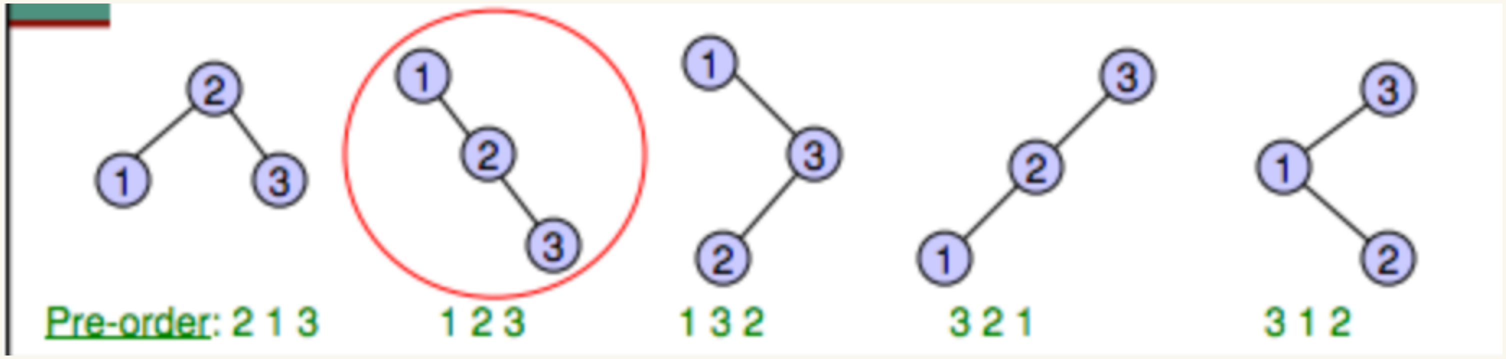
In-class Exercise

```
// Returns 1 if the tree rooted at x is a BST, 0 otherwise
int check_bst(Node * x){
  if (x == NULL) return 1;

  if ((x->left != NULL) && (x->left->key > x->key))
    return 0;
  if ((x->right != NULL) && (x->right->key < x->key))
    return 0;

  return check_bst(x->left) && check_bst(x->right);
}
```



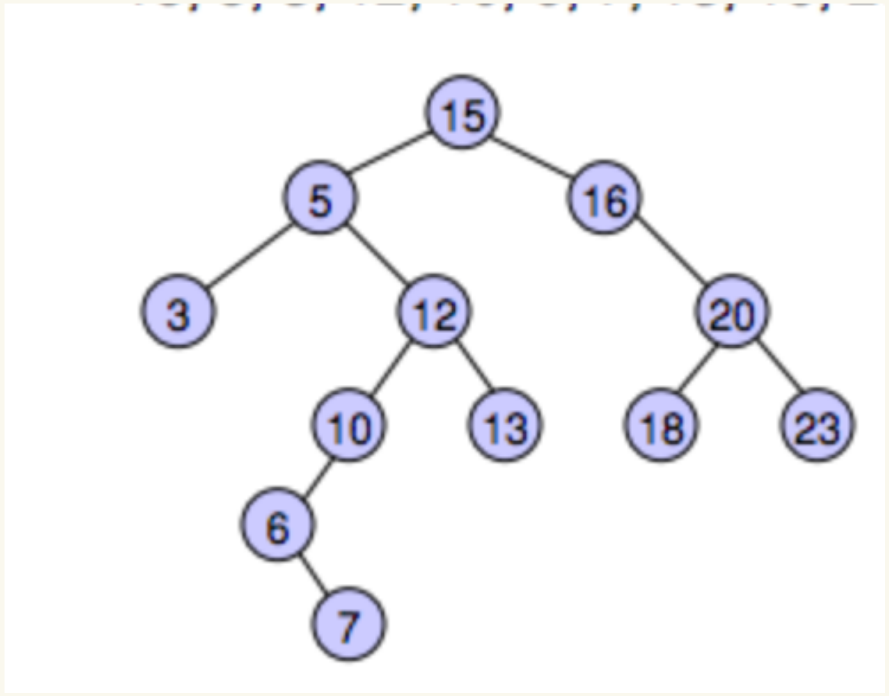
An Application of pre-order traversing

- Suppose we want to transmit our tree across the country to another programmer. Sending the in-order list would tell them the values, but would not communicate how the tree is built.
- All of the tree below have the in-order walk: 1 2 3. But only one of the trees below has the pre-order walk 1 2 3.
  - Note that we expect the tree to hold the BST property



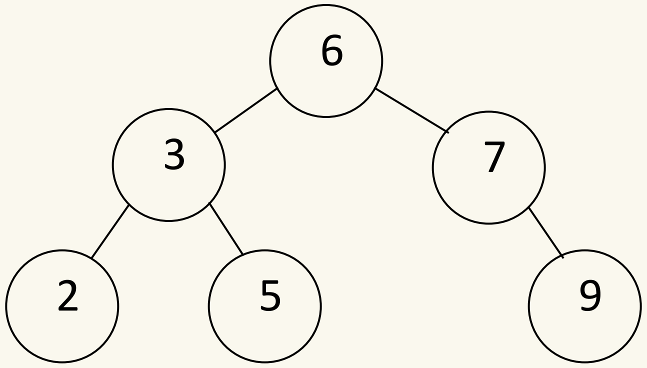
In-class exercise

- Can you recover the binary search tree from its pre-order traversal?
  - 15, 5, 3, 12, 10, 6, 7, 13, 16, 20, 18, 23



An Application of in-order traversing

Sorting values in a binary search tree



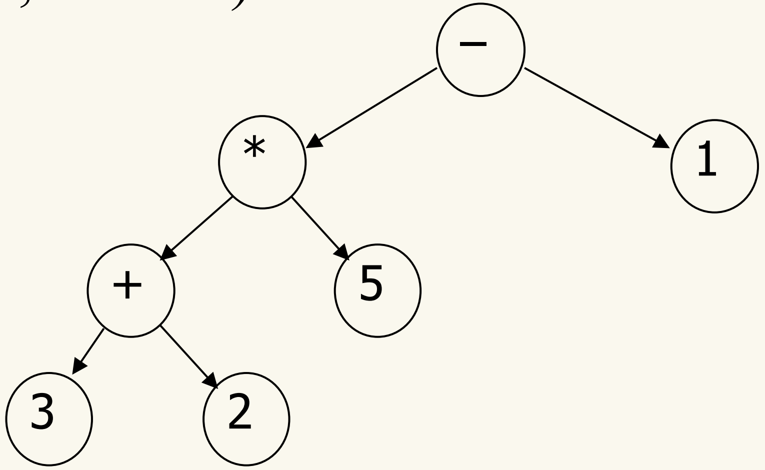
In-order = 2, 3, 5, 6, 7, 9

An Application of post-order traversing

Traverse the tree in post-order (left, right, current)

3 2 + 5 * 1 –

Use a stack to compute the value



Character scanned	Stack
3	3
2	3, 2
+	5
5	5, 5
*	25
1	25,1
-	24

Learning goals revisited

- Provide examples of the types of problems for which tree data structures are well-suited.
- Describe and use preorder, inorder, and postorder tree traversal algorithms.
- Perform a binary search on an array iteratively and recursively.
- Describe the properties of a binary search tree.
- Determine if a given tree is an instance of a binary search tree.
- Search for keys in a binary search tree.
- Insert and delete keys from a binary search tree.
- Describe the properties of binary trees and binary search trees; and algorithms for navigating (e.g., searching, adding, deleting) them in C.